PEARL: Solver-in-the-Loop Interactive Optimization Modeling from Natural Language

Hongliang Lu[1*] **Zhong Li**[2*]
Yuxuan Chen[1] **Yuan Lan**[3] **Fan Zhang**[3] **Zaiwen Wen**[1(✉)]
[1]Peking University [2]Great Bay University
[3]Huawei Technologies Co., Ltd
[(✉)]Corresponding author: wenzw@pku.edu.cn (Zaiwen Wen)
[*]Equal contributions.

Abstract

Optimization modeling is the process of translating real-world decision problems, often described in natural language, into formal mathematical formulations and executable solver code. While recent advances in large language models have shown promise in automating this process, many approaches focus on one-shot generation or rely on predefined execution-and-repair workflows, without explicitly learning a policy for tool use and revision. This stands in sharp contrast to real-world optimization modeling, which is inherently interactive and proceeds through repeated solve–debug–revise cycles. We introduce PEARL, a system for multi-turn solver-interactive optimization modeling that uses Python execution and mathematical programming solvers inside this loop. Rather than relying on a fixed repair workflow, PEARL learns when to test partial models, how to revise from solver diagnostics, and when to stop. It operates in a multi-turn tool-integrated setting where intermediate execution results, feasibility signals, and solution checks are used to improve both formulations and solver code before finalization. Across diverse optimization benchmarks, PEARL substantially improves executable objective agreement over the same-scale base model and is competitive with strong one-shot and tool-augmented baselines. Notably, PEARL-Qwen3-**4B** outperforms the much larger DeepSeek-V3.2-**685B** baseline under direct prompting on both macro- and micro-averaged Pass@1. Code and datasets are provided at https://github.com/AuroraLHL/PEARL.

1 Introduction

Optimization modeling is the process of translating real-world decision problems, often described in natural language (NL), into formal mathematical models and generating corresponding solver code. It provides the critical link between high-level problem descriptions and the low-level solvers [Anand et al., 2017] that compute optimal or feasible decisions. Across domains [Singh, 2012, Antoniou and Lu, 2007] ranging from energy systems to logistics, finance, and manufacturing, the quality of this modeling step largely determines whether optimization can be applied. However, modeling typically requires substantial domain knowledge, careful specification of variables and constraints, and repeated debugging of data pipelines and solver implementations [Huang et al., 2025, Jiang et al., 2025]. In practice, even seemingly minor issues—such as a missing constraint, an incorrect index, or inconsistent data bindings—can make a model infeasible or lead to silent but severe errors.

Recent advances in large language models (LLMs) have spurred growing interest in automating optimization modeling from natural language [Xiao et al., 2025]. Existing approaches can be broadly grouped into three categories. *Prompting-based* methods [Prasath and Karande, 2023]

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

treat optimization modeling as a single-pass or weakly iterative generation task. *Workflow-based* systems [Xiao et al., 2024, Ahmaditeshnizi et al., 2024] decompose modeling into multiple stages such as drafting, execution, and repair, typically orchestrated by fixed prompting logic or search procedures. *Training-based* approaches [Huang et al., 2025, Lu et al., 2025, Chen et al., 2025a] improve base models through supervised fine-tuning or reinforcement learning (RL) on curated optimization-modeling data.

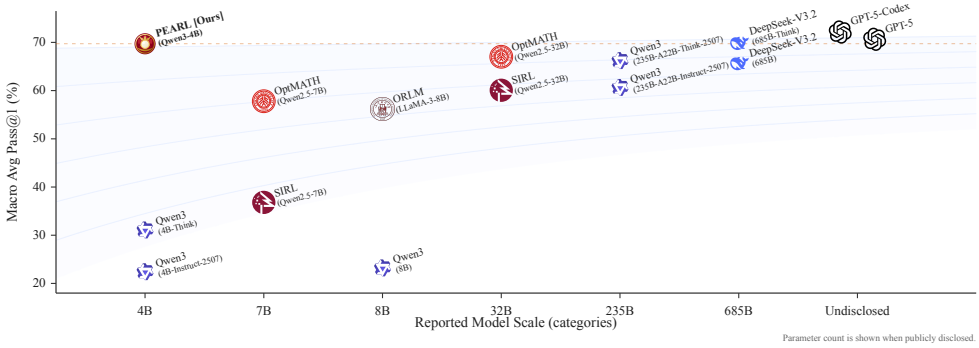


Figure 1: Macro-averaged Pass@1 versus reported model scale across representative optimization-modeling methods. PEARL (Ours) uses a 4B backbone yet matches or exceeds much larger general-purpose models such as DeepSeek-V3.2-685B, highlighting that solver-mediated interaction and optimization-specific training can deliver strong accuracy in a compact model regime that is more practical for repeated agentic invocation [Kwon et al., 2023, Abdin et al., 2024, Belcak et al., 2025].

Across these paradigms, LLMs demonstrate a nontrivial ability to capture modeling structure. However, many training-based approaches still optimize single-pass generation, while workflow-based systems often prescribe the interaction policy. Even in workflow-based systems [Xiao et al., 2024, Ahmaditeshnizi et al., 2024], the control flow that governs tool use and revision is typically hand-designed, and models are not explicitly trained to make multi-turn decisions based on solver feedback. In the single-pass setting, given a natural-language (NL) specification, the model is expected to produce a complete formulation and solver implementation in a single pass, without executing partial models, conditioning on solver diagnostics, or revising errors. As a result, generated outputs frequently hallucinate constraints [Li et al., 2026a, Wu et al., 2026], produce infeasible formulations, or fail at runtime. Some recent systems [Chen et al., 2025a] iteratively execute and repair solver code using tool feedback, but these interactions are orchestrated by heuristics rather than learned policies. Crucially, the underlying models are not trained to decide when to invoke tools, how to interpret solver feedback, or whether to revise or terminate. This mismatch between training objectives and real-world modeling workflows limits performance, particularly as problem descriptions become longer, noisier, and more structurally complex.

In contrast, real-world optimization modeling is inherently interactive [Zhang et al., 2024]. Human modelers rarely produce a correct model in a single pass; instead, they follow a solve–debug–revise loop. Partial formulations are tested, solver errors or infeasibility reports are inspected, constraints or data processing are adjusted, and the process repeats until the model passes verification. This workflow naturally involves temporally extended decision-making, partial observability, and tool-mediated feedback. These characteristics align closely with the emerging paradigm of *agentic reinforcement learning* [Zhang et al., 2025], which treats LLMs as policies operating in multi-turn, tool-integrated environments rather than as one-shot generators.

Motivated by this perspective, we argue that optimization modeling should be formulated not as static text generation, but as an *interactive decision process*. To this end, we introduce PEARL (interactive o**p**timization mod**e**ling with tool-integrated **a**gentic **r**einforcement **l**earning), a method for modeling and solving optimization problems via tool-integrated agentic reinforcement learning (Agentic RL). In PEARL, an agent incrementally constructs mathematical formulations and solver implementations, invokes tools such as Python execution and optimization solvers, observes execution traces and solver feedback, and revises its outputs across multiple turns before finalizing its answer. Rather than relying on fixed heuristics or prompting logic, the agent is trained to learn *how* to interact with tools, *when* to test partial models, *how* to repair failures, and *when* to terminate. Here, interaction refers to

iterative execution and revision using solver feedback; it does not include clarification of missing requirements with a human user.

To support this behavior, we design an optimization-specific Agentic RL environment with structured tool interfaces and solver-based observations. We further introduce a verifiable reward decomposition that combines interaction-format validity with objective agreement after successful execution, while execution traces and feasibility signals are fed back as observations for revision. This design enables long-horizon credit assignment over modeling decisions, rather than only over final outputs. Building on recent advances in tool-integrated Agentic RL, we adapt training recipes and curricula to the unique challenges of optimization tasks, including sparse terminal rewards, solver nondeterminism, and heterogeneous failure modes arising from data processing, modeling, and implementation errors.

We evaluate PEARL on a diverse set of optimization modeling benchmarks, ranging from standard natural-language-to-optimization (NL-to-Opt) tasks to long-context and industrial-style problem descriptions. Across these settings, training optimization modeling as an interactive agent leads to substantial improvements in verified solve rates over strong one-shot and tool-augmented baselines. In particular, PEARL reduces code execution errors and output-format violations, highlighting the benefits of explicitly learning solve–debug–revise behavior. As previewed in Figure 1, this is not merely a scale effect: a 4B PEARL policy trained for solver-mediated interaction outperforms the much larger DeepSeek-V3.2-685B in macro-averaged Pass@1, while its compact backbone makes repeated agentic invocation more practical by reducing the serving burden of multi-turn tool use [Kwon et al., 2023, Abdin et al., 2024, Chen and Li, 2024, Belcak et al., 2025].

Contributions. Our main contributions are threefold: (1) We reconceptualize optimization modeling as an inherently *interactive* process rather than a one-shot generation task, and formalize it as a multi-turn, tool-integrated decision problem in which solver execution and diagnostic feedback are treated as part of the agent's observable state; (2) Building on this formulation, we introduce an optimization-specific agentic reinforcement learning approach, enabling the agent to learn when to execute, revise, and terminate through solver-grounded rewards and an optimization-specific training curriculum; (3) Through extensive experiments across diverse NL-to-Opt benchmarks, we demonstrate that PEARL substantially improves verified solve rates over existing baselines; notably, our PEARL-Qwen3-**4B** outperforms the much larger DeepSeek-V3.2-**685B** [Liu et al., 2025] in both macro- and micro-averaged accuracy on optimization modeling tasks (see Figure 1). Our methodological contribution lies in the optimization-specific interaction environment and policy training, built on established RL optimization methods.

2 Related Work

LLMs for optimization modeling. A growing body of work studies natural-language-to-optimization modeling, covering decision variables, objectives, constraints, and executable solver code [Xiao et al., 2025]. Prompting-based methods rely on templates or in-context examples to elicit formulations from pretrained LLMs [Prasath and Karande, 2023]. Workflow-based systems decompose modeling into drafting, execution, and repair, using hand-designed control logic or search, as in Autoformulation [Astorga et al., 2025], Chain-of-Experts [Xiao et al., 2024], and OptiMUS [Ahmaditeshnizi et al., 2024]. Supervised fine-tuning methods such as ORLM [Huang et al., 2025], LLMOPT [Jiang et al., 2025], and OptMATH [Lu et al., 2025] improve formulation and code generation. OptiMind [Chen et al., 2025b] uses class-specific error analysis to clean fine-tuning data and guide multi-turn inference with hints and solver feedback.

Recent RL-based methods use solver outcomes as verifiable supervision. SIRL [Chen et al., 2025a] optimizes complete generations using executability, feasibility, and correctness rewards, while OR-R1 [Ding et al., 2026], StepORLM [Zhou et al., 2026], and STORM [Tang et al., 2025] explore richer objectives, curricula, or step-level signals. MIND [Liu et al., 2026c] pairs error-targeted data synthesis with reinforcement learning and supervised updates on teacher-corrected rollouts. Several recent systems also use iterative or agent-style optimization workflows, including OptLLM [Zhang et al., 2024], OptiTree [Liu et al., 2026b], LLMOPT [Jiang et al., 2025], and MM-Agent [Liu et al., 2026a]. MURKA [Xie et al., 2025] combines a GRPO-trained Extractor, a distilled Solver, and Checker-guided refinement using solver feedback. PEARL instead uses multi-turn RL to train the interaction policy itself, including tool invocation, revision, and termination. A systematic comparison with prior RL-based methods is provided in Table 2 (Appendix A.1).

For post-hoc verification and repair, OptArgus [Li et al., 2026a] uses specialist agents to audit problem–model–code consistency, while OPT-Kraken [Wu et al., 2026] uses source-grounded evidence to constrain repair scope and preserve unaffected model and code components.

Agentic reinforcement learning. Agentic reinforcement learning formulates LLMs as policies in temporally extended, partially observable environments whose actions may include reasoning steps and tool calls [Zhang et al., 2026]. Recent systems such as rStar2-Agent [Shang et al., 2025], ReTool [Feng et al., 2026], SimpleTIR [Xue et al., 2026], and Agent-R1 [Cheng et al., 2025] show that end-to-end RL can induce tool use, self-correction, and verification behavior with code interpreters, calculators, or search engines. Optimization modeling adds domain-specific structure: solvers expose partially verifiable signals such as feasibility, status, and objective values, while failures may arise from data processing, formulation, implementation, or solver limits. PEARL adapts the Agentic RL paradigm to this setting by making solver interaction, revision, and termination trainable decisions in a multi-turn optimization-modeling environment. This emphasis on defining states, actions, rewards, and termination conditions follows the MDP-design perspective discussed by Yang et al. [2026].

3 Methodology: Optimization Modeling via Agentic Reinforcement Learning

Problem Statement. We study automated optimization modeling from natural language as a sequential decision-making problem. Given a natural-language problem specification and associated data artifacts, the goal is to produce a mathematically correct optimization formulation together with executable solver code that passes verification. Crucially, correctness cannot be reliably assessed from the generated text alone: execution, solver feedback, and reference-based checks provide operational evidence for this goal. This setting naturally induces multi-step interaction, partial observability, and delayed rewards.

Optimization Modeling as a POMDP. We formalize agentic optimization modeling as an episodic partially observable Markov decision process (POMDP) defined by the tuple [Zhang et al., 2026]

$$\langle \mathcal{S}, \mathcal{O}, \mathcal{A}, \mathcal{P}, \mathcal{R}, T \rangle.$$

$\mathcal{S}$ denotes the latent environment state, which includes the true optimization problem semantics, hidden modeling errors, solver-internal states, and execution side effects that are not directly observable to the agent. The agent does not observe $s_t \in \mathcal{S}$ directly; instead, it receives observations $o_t \in \mathcal{O}$ consisting of the natural-language specification, the conversation history, intermediate code artifacts, and tool feedback such as runtime outputs, solver logs, feasibility status, and validation reports.

The action space $\mathcal{A}$ consists of two classes of actions: (i) natural-language actions that revise or extend the current mathematical formulation and implementation plan, and (ii) tool-invocation actions that execute code or trigger validation procedures. Transitions $\mathcal{P}(s_{t+1} \mid s_t, a_t)$ are induced by the effects of code execution, solver behavior, and internal environment updates. The scalar reward combines interaction-format rewards with terminal objective agreement. Runtime errors, solver status, and feasibility signals are observations for revision, not additional scalar reward terms. An episode terminates when the agent emits a final answer (formulation plus solver code) or when a turn or wall-clock budget T is exhausted.

This problem formalization explicitly distinguishes agentic optimization modeling from traditional one-shot generation, which corresponds to a degenerate MDP with horizon $T = 1$ and no intermediate observations. We next describe the environment and tool interface (Section 3.1), action space and output format (Section 3.2), reward design (Section 3.3), and training objective and policy optimization (Section 3.4). The overall workflow is demonstrated in Figure 2.

3.1 Environment and Tool Interface

At each step, the agent conditions on the full observation history $o_{\leq t}$, including the NL specification, intermediate modeling artifacts, and feedback from prior tool interactions. The environment exposes a Python execution interface through `<code>...</code>` actions.

The execution interface runs model-generated Python code for data parsing, model construction (e.g., via Pyomo), solver invocation, and solution extraction. It returns execution output and errors inside `<interpreter>...</interpreter>` blocks; solver status, objective values, solutions, and constraint checks are visible when the executed code reports them. Such checks are performed by

model-generated code, not by a separate trusted *Validate* action. The agent can use this feedback to revise its implementation or terminate. Reference-objective comparison is performed separately by the terminal scorer, which does not append the label or comparison result to the interaction history.

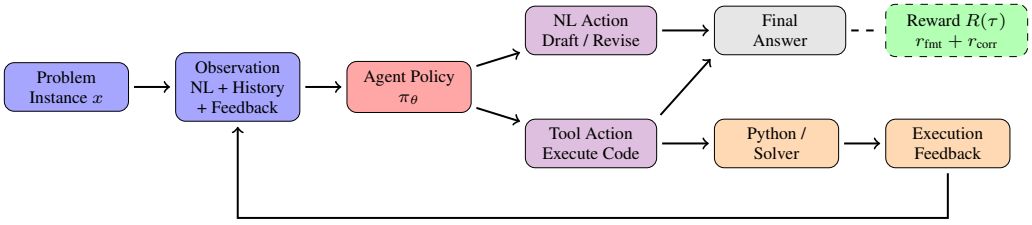


Figure 2: Workflow of PEARL. Colors indicate elements of the POMDP $\langle \mathcal{S}, \mathcal{O}, \mathcal{A}, \mathcal{P}, \mathcal{R}, T \rangle$: observations $\mathcal{O}$ (blue), agent policy π_θ (red), actions $\mathcal{A}$ (purple), environment transitions $\mathcal{P}$ (orange), reward $\mathcal{R}$ (green, dashed), and termination T (gray). The latent state $\mathcal{S}$ is not shown. Terminal scoring executes the final code and compares its objective with the reference; this comparison is not an in-loop tool observation.

3.2 Action Space and Output Format

At each turn t, the agent selects an action $a_t \in \mathcal{A}$ that explicitly controls the optimization modeling workflow. The action space consists of two categories: (1) *natural-language actions* modify or extend the current modeling state, including revising decision variables, objectives, constraints, indexing schemes, or implementation plans. These actions correspond to reasoning and editing steps typically performed by human modelers prior to execution; and (2) *tool-invocation actions* trigger interaction with the environment via the Python execution interface, enabling the agent to test partial formulations, inspect solver behavior, and run checks on its candidate solution.

The choice between natural-language revision and tool invocation is part of the learned policy, allowing the agent to decide *when* to execute, *what* to test, and *whether* further revision is warranted. An episode terminates when the agent emits a designated terminal action, which outputs a final answer consisting of: (1) a complete mathematical formulation—including decision variables, objective, and constraints—expressed in a standardized template, and (2) executable solver code that produces a solution and an accompanying verification report. This explicit termination decision enables learning over long-horizon solve–debug–revise trajectories rather than fixed-length generation. Training and inference allow at most eight generation turns, including the final-answer turn. The agent may terminate earlier with an `<answer>...</answer>` response. Thus, a trajectory ending with this action can contain at most seven preceding tool calls; exhausting all eight turns on tool calls does not leave an extra final-answer turn.

3.3 Reward Design

We design rewards to be *verifiable*, *solver-grounded*, and aligned with the semantics of optimization modeling. Let $\tau = (a_{1:T})$ denote a multi-turn interaction trajectory and $R(\tau)$ its return. Because the correctness of an optimization model cannot be reliably assessed from text alone and typically only becomes evident after execution, reward signals are sparse and delayed.

In practice, we define the return as the sum of two components: $R(\tau) = r_{\text{fmt}}(\tau) + r_{\text{corr}}(\tau)$, which jointly encourage correct interaction protocols and solver-verifiable modeling outcomes.

Format reward. The format reward r_{fmt} is computed once per trajectory: $+0.1$ if the response contains both opening and closing `interpreter` tags, and $+0.2$ if the final-answer parser extracts a nonempty code block in the required format. The 0.1 bonus is neither accumulated per call nor conditional on tool success. This component stabilizes training by discouraging invalid interactions that would otherwise prevent execution-based evaluation.

Objective-agreement reward. The objective-agreement reward r_{corr} is grounded in solver execution. After extracting and executing the generated optimization code, we obtain the solver-reported objective value v_{pred}. All retained training instances have a verified reference optimal objective value v_{label}. If execution succeeds and an objective value is extracted, the trajectory earns $+1.0$

when $|v_{\text{pred}} - v_{\text{label}}| \leq \epsilon$, where $\epsilon = 10^{-3}$ is the absolute tolerance for training rewards and online evaluation during training; the final benchmark evaluation protocol is specified separately in Section 4. Otherwise, this reward is zero. Code extraction and execution are prerequisites, not separate rewards, and the scorer does not independently enforce optimal solver termination. Objective agreement is a practical proxy, not a certificate of semantic correctness; see Appendix I.

By grounding rewards in executable code and solver-verifiable outcomes, this design avoids text-based correctness proxies. While sparse at the modeling level, the reward provides a clear supervision signal aligned with the end goal of optimization modeling: producing executable models that yield correct solver behavior. Further parsing and scoring details are provided in Appendix A.2.

3.4 Training Objective and Policy Optimization

We train the optimization-modeling agent using agentic reinforcement learning with multi-turn rollouts in a tool-augmented POMDP. The policy is initialized from a pretrained language model and subsequently optimized via GRPO-style reinforcement learning with DAPO-inspired stabilization details such as decoupled clipping [Shao et al., 2024, Yu et al., 2025]. This recipe is well suited for sparse, delayed rewards and high-variance tool-augmented trajectories. The learning procedure is summarized in Algorithm 1 (see Appendix A.3).

Objective. We train the agent to maximize expected performance over multi-turn optimization-modeling episodes. Let $x \sim \mathcal{D}$ denote a single optimization modeling instance, consisting of a natural-language problem description and associated data artifacts. Given x, the agent induces a multi-turn trajectory $\tau = (o_{1:T}, a_{1:T}) \sim \pi_\theta(\cdot \mid x)$, where each observation o_t encapsulates the conversation history and tool feedback accumulated up to turn t, and each action a_t corresponds to either a natural-language modeling step or a tool invocation. The episode terminates after T turns or upon producing a final formulation and solver implementation. Each trajectory τ receives a scalar return $R(\tau)$ derived from solver execution and validation signals, as described in Section 3.3. We optimize the expected trajectory-level return:

$$\max_\theta \; J(\theta) = \mathbb{E}_{x\sim\mathcal{D}}\, \mathbb{E}_{\tau\sim\pi_\theta(\cdot|x)}[R(\tau)]\,. \tag{1}$$

where π_θ is the agent policy being optimized. This objective encourages improvement in long-horizon, solver-verified performance in a high-variance, tool-augmented environment.

Policy update with decoupled clipping. We perform on-policy policy optimization using trajectories sampled from the current policy. Let $\pi_{\theta_{\text{old}}}$ denote the policy used to generate rollouts in the current iteration. For each token-level action a_t conditioned on observation o_t, we define the standard importance sampling ratio $\rho_t(\theta) = \frac{\pi_\theta(a_t|o_t)}{\pi_{\theta_{\text{old}}}(a_t|o_t)}$, where π_θ is the updated policy being optimized.

Given an advantage estimate A_t for action a_t, we adopt the decoupled clipping strategy introduced in DAPO [Yu et al., 2025], which applies asymmetric clipping depending on the sign of the advantage:

$$\bar{\rho}_t(\theta) = \begin{cases} \min(\rho_t(\theta),\, 1+\epsilon_+), & A_t \geq 0, \\ \max(\rho_t(\theta),\, 1-\epsilon_-), & A_t < 0, \end{cases} \tag{2}$$

$$\mathcal{L}_{\text{pg}}(\theta) = \mathbb{E}[\min(\rho_t(\theta)\, A_t,\; \bar{\rho}_t(\theta)\, A_t)]\,, \tag{3}$$

where ϵ_+ and ϵ_- control the clipping range for positive and negative advantages. This asymmetric design stabilizes learning under high-variance returns in tool-augmented, multi-turn environments.

Because rewards in optimization modeling are computed at the trajectory level, for each problem x_i we sample a group of $G = 8$ trajectories $\tau_{i,j} \sim \pi_{\theta_{\text{old}}}(\cdot \mid x_i)$. In the main training configuration, returns are normalized within each problem's group:

$$A_{i,j} = \frac{R(\tau_{i,j}) - \mu_i}{s_i + \delta}, \qquad \mu_i = \frac{1}{G}\sum_{j=1}^{G} R(\tau_{i,j}),$$

$$s_i^2 = \frac{1}{G-1}\sum_{j=1}^{G}\big(R(\tau_{i,j}) - \mu_i\big)^2, \tag{4}$$

where s_i is the sample standard deviation used by the implementation and $\delta = 10^{-6}$. The same $A_{i,j}$ is assigned to the model-generated tokens of trajectory $\tau_{i,j}$, with tool-observation tokens masked out. No additional batch-wide advantage normalization is enabled.

Optimization-specific structure. Optimization modeling exhibits properties uncommon in generic tool-use tasks. Solver nondeterminism and time limits are treated as environment noise. Problem difficulty varies sharply with context length and specification completeness, motivating a curriculum from short, fully specified instances to long-context and industrial-style inputs. Finally, execution errors, solver diagnostics, and checks performed by the generated code can help distinguish implementation failures from candidate modeling errors, enabling feedback-conditioned revision rather than undifferentiated terminal supervision.

4 Experiments

Research questions. Our experiments aim to answer the following questions: **RQ1.** Does modeling optimization as an interactive, agentic decision process improve verified solve rates over one-shot generation and heuristic tool-augmented baselines across diverse benchmarks? **RQ2.** How do key training choices influence performance and stability in agentic optimization modeling? **RQ3.** How does agentic training alter the distribution of failure modes compared to non-agentic approaches?

4.1 Experimental Setup

Benchmarks and tasks. We evaluate on a range of established optimization modeling benchmarks, including NL4Opt [Ramamonjison et al., 2022], MAMO EasyLP and ComplexLP [Huang et al., 2024], NLP4LP [Ahmaditeshnizi et al., 2024], OptMATH [Lu et al., 2025], IndustryOR [Huang et al., 2025], ComplexOR [Xiao et al., 2024], and Resocratic [Yang et al., 2024]. Across all benchmarks, the task is end-to-end optimization modeling: given a natural-language specification and associated data, the model must produce a mathematically correct formulation and executable solver code that yields a feasible (ideally optimal) solution and passes verification.

The suite spans linear, integer, mixed-integer, nonlinear, and second-order cone programs, but it does not exhaust real-world optimization modeling. Stochastic, robust, and highly domain-specific enterprise settings remain under-represented, so our claims should be read as evidence for the current public NL-to-Opt regime rather than universal coverage.

For PEARL training, we use a mixed corpus of roughly 14k optimization-modeling instances: about 10k cleaned open-source instances drawn from OR-Instruct/ORLM [Huang et al., 2025], OptMATH [Lu et al., 2025], and OptiBench-style data [Yang et al., 2024], plus about 3.8k newly constructed and manually curated instances. Appendix G describes how the training data were collected and prepared.

Compared methods. Our primary system is PEARL-Qwen-3-4B-Instruct-2507. We compare it with baselines spanning different model scales, training regimes, and inference settings. *Direct Prompt* baselines include open-weight and closed-source LLMs evaluated without task-specific training; for models that support different reasoning behaviors, either via distinct variants or inference-time configurations, we report both standard and reasoning-enabled results when available. These include Qwen-3-4B-Instruct/Think, Qwen-3-235B-A22B-Instruct/Think, DeepSeek-V3.2-685B/Think, and the GPT-5-family endpoints used during our method development and evaluation, GPT-5 and GPT-5-CodeX. *Fine-Tuned* baselines are prior optimization-modeling systems adapted via supervised fine-tuning and/or reinforcement learning, including SIRL-Qwen2.5-7B/32B, ORLM-LLaMA-3-8B, and OptMATH-Qwen2.5-7B/32B. For all methods in Table 1, we execute their final code and check its objective value against the reference. The baselines generate code in one response, while PEARL learns to revise its code using solver feedback. We also compare PEARL with the untrained Qwen3-4B-Instruct-2507 model using the same tools and turn limit for both (Appendix D.4; Table 7).

Evaluation metrics. Evaluation is performed at both the instance and dataset levels. At the instance level, a generated program passes the objective-agreement criterion if it executes successfully and its output satisfies the normalized objective-error threshold, i.e., $\frac{|y_{\text{pred}} - y_{\text{label}}|}{|y_{\text{label}}| + 1} < 10^{-6}$. At the dataset level, we report pass@1 accuracy, defined as the fraction of instances for which a single model-generated solution satisfies this criterion. Each evaluation run produces one final candidate per instance. By default, we run each pass@1 evaluation three times and report mean $\pm$ standard deviation. Unless otherwise specified, inference is conducted with a sampling temperature of 0.6. Additional evaluation details and parameter settings are provided in Appendix B. Code and datasets are provided at `https://github.com/AuroraLHL/PEARL`.

For the main Qwen3-4B experiment, we train on four high-performance GPUs with a learning rate of 10^{-6}. Each rollout batch contains 16 problems, with eight interaction trajectories sampled for each problem. Full hyperparameters are provided in Table 3 (Appendix A).

4.2 Experimental Results and Analysis

The main text answers RQ1 by measuring overall effectiveness against prompting and fine-tuned baselines (Sec. 4.2.1; setup details, baseline-selection caveats, and extended comparisons in Appendix B and Appendix B.4), RQ2 by analyzing training choices for stable agentic optimization modeling (Sec. 4.2.2; ablations and training dynamics in Appendix D), and RQ3 by attributing how failures change after PEARL training (Sec. 4.2.3; diagnostic definitions in Appendix C).

4.2.1 RQ1: Overall Effectiveness of Agentic Optimization Modeling

Table 1 answers RQ1 by comparing PEARL with direct prompting and prior task-specific fine-tuning baselines across all benchmarks. We draw the following takeaways.

Takeaway 1: PEARL substantially improves performance at a fixed model scale. PEARL substantially improves over same-scale direct prompting baselines: the 4B PEARL model reaches 69.71 macro average, compared with 22.20 for Qwen-3-4B-Instruct and 30.86 for Qwen-3-4B-Think. It also achieves the highest macro average among fine-tuned models in Table 1 under our Pyomo evaluation protocol. PEARL also has a higher macro average than its base model with the same tools and turn budget (Appendix D.4; Table 7), and than SIRL with the same backbone and training data (Appendix H.1; Table 10). Thus, tool access or extra data alone cannot explain the gain.

Takeaway 2: large gains appear on harder modeling benchmarks. On simpler datasets such as NL4Opt and MAMO EasyLP, improvements are modest because many strong models already achieve high solve rates. In contrast, on harder benchmarks such as MAMO ComplexLP and Resocratic, PEARL achieves large absolute gains, suggesting improved robustness to complex modeling and implementation demands. This pattern is consistent with the benefits of learning solve–debug–revise behavior.

Table 1: Pass@1 accuracy (%) across benchmarks (higher is better). Each entry reports the mean ± standard deviation over three runs.

	Method	NL4Opt	MAMO-E	MAMO-C	NLP4LP	OptMATH	IndustryOR	ComplexOR	Resocratic	Macro Avg	Micro Avg
Direct Prompt	Qwen-3-4B-Instruct	48.13±1.40	2.20±0.45	13.51±2.20	55.62±1.25	2.41±0.50	16.66±3.10	5.56±2.78	33.50±1.80	22.20±0.75	22.42±0.55
	Qwen-3-4B-Think	48.60±2.80	27.34±2.50	6.31±1.90	59.55±2.10	1.81±0.45	30.95±4.10	22.22±5.56	50.12±2.50	30.86±0.95	35.06±0.80
	Qwen3-235B-A22B-Instruct-2507	74.92±1.77	91.62±0.69	38.74±5.41	88.95±0.33	20.88±3.09	58.73±5.50	38.89±0.00	70.64±1.41	60.42±1.13	72.26±0.69
	Qwen3-235B-A22B-Think-2507	79.59±5.42	89.79±3.07	40.84±6.64	91.20±5.54	30.72±1.59	71.43±4.12	50.00±5.56	74.36±4.50	65.99±1.68	74.95±1.39
	DeepSeek-V3.2-685B	83.18±3.07	94.07±2.29	50.75±9.38	92.70±2.81	25.10±3.03	58.73±11.98	51.85±6.42	75.68±2.52	66.51±1.88	77.07±1.22
	DeepSeek-V3.2-Think-685B	85.20±1.18	95.53±0.59	53.15±3.12	97.38±0.86	30.32±1.25	64.29±2.39	50.00±5.56	77.67±0.43	69.19±0.65	79.58±0.41
	GPT-5	83.80±0.97	94.31±0.32	59.16±0.52	94.76±0.65	31.33±0.61	73.02±4.95	53.70±8.49	81.14±1.29	**71.40±1.55**	**80.31±0.41**
	GPT-5-Codex	80.37±0.94	95.60±0.19	61.56±0.52	97.94±0.32	32.13±0.69	69.05±2.38	55.56±5.56	78.99±0.38	**71.40±0.37**	80.27±0.15
Fine-Tuned	SIRL-Qwen2.5-7B	76.17±1.45	82.20±1.20	2.70±1.35	93.26±0.75	6.02±0.80	21.43±3.60	11.11±3.92	63.03±1.60	36.81±0.90	55.34±0.70
	SIRL-Qwen2.5-32B	87.38±1.10	98.90±0.45	70.27±3.20	97.75±0.50	22.89±1.40	59.52±4.10	33.33±5.56	78.16±1.10	60.07±0.75	73.65±0.60
	ORLM-LLaMA-3-8B	78.50±1.80	82.30±1.60	41.44±3.80	89.54±1.20	6.02±0.90	42.86±4.70	38.89±5.56	70.22±1.50	56.22±0.95	67.97±0.75
	OptMATH-Qwen2.5-7B	84.11±1.30	77.61±1.80	33.33±3.40	89.47±1.30	22.29±1.50	30.95±4.10	33.33±5.56	91.32±0.80	57.80±0.85	72.94±0.65
	OptMATH-Qwen2.5-32B	85.51±1.05	95.05±0.80	44.14±3.60	95.51±0.65	37.35±1.70	45.24±4.60	50.00±5.56	83.37±0.95	67.02±0.70	80.26±0.45
	PEARL-Qwen-3-4B-Instruct (Ours)	85.28±0.99	91.65±0.65	76.12±3.19	93.25±1.59	16.87±1.70	67.86±8.42	41.66±3.92	84.99±0.52	69.71±0.13	79.84±0.25

Takeaway 3: the tool scaffold alone is insufficient. Appendix D.4 reports a controlled scaffold comparison in which the base Qwen3-4B-Instruct-2507 model uses the same tool interface, execution-feedback loop, and repair budget as PEARL. The base model can invoke tools, but it uses them more sparsely and achieves lower accuracy across all benchmarks, so providing tools without training does not reproduce PEARL's performance. The untrained model achieves 36.32% macro-averaged Pass@1, compared with 69.71% for PEARL (Appendix D.4; Table 7). The training gains also extend to another backbone: with Qwen3-8B, PEARL improves over the untrained model, reaching 67.73% macro-averaged and 79.61% micro-averaged Pass@1 (Appendix B.4; Table 5).

Takeaway 4: PEARL-4B is competitive with much larger general-purpose models, but not a universal winner. We compare PEARL against substantially larger models, including Qwen-3-235B-A22B, DeepSeek-V3.2-685B, DeepSeek-V3.2-Think-685B, GPT-5, and GPT-5-CodeX. Despite being orders of magnitude smaller, PEARL-4B outperforms DeepSeek-V3.2-685B in both macro-averaged accuracy (69.71 vs. 66.51) and micro-averaged accuracy (79.84 vs. 77.07), with particularly strong results on MAMO ComplexLP and Resocratic. At the same time, GPT-5-family models and DeepSeek-V3.2-Think remain highly competitive or stronger on some aggregate metrics.

The result is therefore not that small models always dominate, but that optimization-specific agentic training can close much of the scale gap for solver-grounded modeling tasks.

Takeaway 5: compact backbones make repeated agentic inference more practical. Agentic optimization modeling amplifies inference cost because each problem may require several policy calls interleaved with code execution and solver feedback, making serving memory and throughput constraints increasingly salient [Kwon et al., 2023]. Recent reports show that 3–4B models can be capable enough for local deployment [Abdin et al., 2024], 2B on-device models can support function-calling agents with strong latency advantages [Chen and Li, 2024], and specialized repetitive agentic routines are natural targets for small language models [Belcak et al., 2025]. We therefore view PEARL-4B as evidence that optimization-specific agentic training can move solver-grounded modeling into a more deployable parameter regime, while not making a universal wall-clock or dollar-cost claim across different serving stacks. PEARL averages 9,868.75 generated tokens per instance, compared with 1,465.51 for DeepSeek-V3.2 and 2,323.56 for DeepSeek-V3.2-Think (Appendix H.2; Table 12). Token count alone, however, is not a measure of serving cost. PEARL's 4B backbone supports local deployment with lower model-memory requirements and potentially lower per-token serving costs [Abdin et al., 2024, Belcak et al., 2025]. These advantages can help offset the extra generation needed for multi-turn modeling, especially for repeated local use.

4.2.2 RQ2: Impact of Training Choices

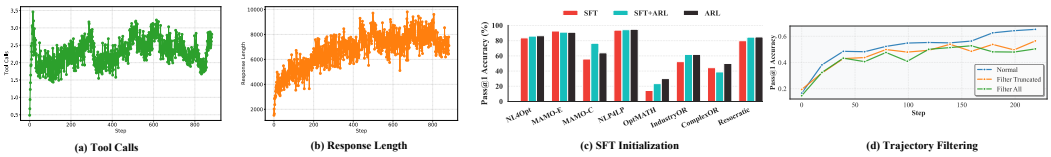


Figure 3: Main ablations for PEARL training choices. Tool use converges early, response length continues to grow under GRPO, SFT mainly serves as a warm-start for less aligned models, and aggressive trajectory filtering is not beneficial in this solver-feedback setting.

We study training dynamics and design choices that affect the stability and efficiency of agentic optimization modeling; Figure 3 summarizes the main evidence, with additional details in Appendix D.1.2, Appendix D.2.2, and Appendix D.3. We make four observations: (1) **Tool-use behavior is learned early.** Figure 3(a) shows that the average number of tool calls stabilizes early in training, suggesting that PEARL quickly learns when external execution and solver feedback are useful. (2) **Later training mainly refines solution traces.** Figure 3(b) shows that response length continues to grow after tool usage has stabilized, indicating that GRPO training increasingly improves the detail and structure of the generated modeling trajectory rather than simply increasing tool frequency. (3) **SFT is a warm-start, not the final driver.** Figure 3(c) indicates that supervised fine-tuning is important for less aligned starting models, but direct agentic RL reaches comparable performance to SFT+ARL when starting from Qwen3-4B-Instruct; thus, SFT helps initialize the policy, while solver-grounded agentic RL determines the final interaction behavior. (4) **Aggressive trajectory filtering is unnecessary in this solver-feedback setting.** Figure 3(d) shows that, unlike some general tool-use domains [Xue et al., 2026, Shang et al., 2025], filtering truncated or format-violating trajectories does not improve performance in this setting and can slightly reduce it. The limited tool-use depth and capped observations may contribute to the observed stability.

4.2.3 RQ3: Failure Mode Analysis

To understand how agentic training changes model behavior, we compare the failure attribution of the base model and PEARL under the main pass@1 evaluation setting. We make three observations: (1) **PEARL shifts failures away from execution and parsing.** Compared with the base model, PEARL fixes many code execution errors and formatting or extraction failures, indicating improved ability to emit executable solver programs, follow the required output protocol, and recover from tool feedback. (2) **The corrected base-model failures are mostly repairable implementation failures.** Among base-model failures that PEARL later solves, most are code execution failures, suggesting that the learned policy mainly teaches the model to execute, inspect, and repair solver programs rather than merely produce longer answers. (3) **The remaining failures are more modeling-centric and expose**

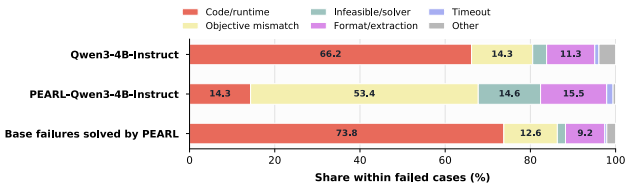


Figure 4: Failure attribution before and after PEARL training under the main pass@1 evaluation setting. Bars are normalized within failed cases; attribution setup, category definitions, and detailed interpretation are provided in Appendix C.

the limits of solver feedback. After PEARL removes many basic execution failures, a larger share of the remaining errors comes from objective mismatch and solver-side issues. These residual cases are more likely to reflect formulation quality, ambiguous specifications, or optimization difficulty. Solver and compiler messages are useful but incomplete evidence for structural correctness, so PEARL combines execution feedback with deterministic validation and, when available, objective-value agreement against references; nevertheless, these checks remain partial oracles.

4.2.4 Additional Comparisons and Data Ablation

Training on the same problems. Using the same Qwen2.5-7B-Instruct backbone, we retrain SIRL on PEARL's 13.8k training problems. Both use Pyomo and the same eight benchmarks, evaluation prompt, and objective check. PEARL reaches 67.86% macro and 77.82% micro Pass@1, versus 57.22% and 73.03% for SIRL (Appendix H.1; Table 10).

Testing SIRL with Gurobi. The released SIRL-Qwen2.5-7B checkpoint, evaluated with its native Gurobi API and our evaluation-prompt template, achieves 66.84% macro and 82.29% micro Pass@1 (Appendix H.1; Table 10). It is competitive with PEARL in macro accuracy and stronger in micro accuracy, but uses different training data.

Contribution of the curated data. In a separate Qwen3-4B experiment, we keep the training recipe unchanged and add 3.8k curated problems to the public 10k pool. Macro Pass@1 rises from 64.12% to 70.40% and micro Pass@1 from 78.95% to 80.38% (Appendix H.2; Table 11).

5 Conclusion

We presented PEARL, an agentic RL framework that treats optimization modeling from natural language as a multi-turn solver-interaction process. By integrating solver execution and feedback into rollouts, PEARL learns solver-aware behaviors such as when to test, how to revise formulations, and when to terminate. PEARL substantially improves over the same-scale base model across the evaluated benchmarks. Notably, compact PEARL models are competitive with the much larger general-purpose baselines evaluated in this study, showing that learned interaction and feedback utilization can complement model scale in optimization modeling. Our current evidence is still limited by bounded interaction budgets, benchmark coverage, and the partial nature of solver-based validation. Future work should extend agentic optimization modeling to stochastic, robust, and domain-specific settings with richer validators and more adaptive interaction policies. Limitations and broader impact are discussed in Appendix I.

Acknowledgments and Disclosure of Funding

The computational resources were supported by the Center for Intelligent Computing and Song-Shan Lake HPC Center (SSL-HPC) in Great Bay University, Dongguan, China. This work was supported in part by National Key Research and Development Program of China under the grant number 2024YFA1012900, and the National Natural Science Foundation of China under the grant numbers 12331010 and 12288101. We also thank the anonymous reviewers for their valuable feedback.

Competing interests. The authors declare no competing interests.

References

Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Martin Cai, Qin Cai, Vishrav Chaudhary, Dong Chen, Dongdong Chen, Weizhu Chen, Yen-Chun Chen, Yi-Ling Chen, Hao Cheng, Parul Chopra, Xiyang Dai, Matthew Dixon, Ronen Eldan, Victor Fragoso, Jianfeng Gao, Mei Gao, Min Gao, Amit Garg, Allie Del Giorno, Abhishek Goswami, Suriya Gunasekar, Emman Haider, Junheng Hao, Russell J. Hewett, Wenxiang Hu, Jamie Huynh, Dan Iter, Sam Ade Jacobs, Mojan Javaheripi, Xin Jin, Nikos Karampatziakis, Piero Kauffmann, Mahoud Khademi, Dongwoo Kim, Young Jin Kim, Lev Kurilenko, James R. Lee, Yin Tat Lee, Yuanzhi Li, Yunsheng Li, Chen Liang, Lars Liden, Xihui Lin, Zeqi Lin, Ce Liu, Liyuan Liu, Mengchen Liu, Weishung Liu, Xiaodong Liu, Chong Luo, Piyush Madan, Ali Mahmoudzadeh, David Majercak, Matt Mazzola, Caio César Teodoro Mendes, Arindam Mitra, Hardik Modi, Anh Nguyen, Brandon Norick, Barun Patra, Daniel Perez-Becker, Thomas Portet, Reid Pryzant, Heyang Qin, Marko Radmilac, Liliang Ren, Gustavo de Rosa, Corby Rosset, Sambudha Roy, Olatunji Ruwase, Olli Saarikivi, Amin Saied, Adil Salim, Michael Santacroce, Shital Shah, Ning Shang, Hiteshi Sharma, Yelong Shen, Swadheen Shukla, Xia Song, Masahiro Tanaka, Andrea Tupini, Praneetha Vaddamanu, Chunyu Wang, Guanhua Wang, Lijuan Wang, Shuohang Wang, Xin Wang, Yu Wang, Rachel Ward, Wen Wen, Philipp Witte, Haiping Wu, Xiaoxia Wu, Michael Wyatt, Bin Xiao, Can Xu, Jiahang Xu, Weijian Xu, Jilong Xue, Sonali Yadav, Fan Yang, Jianwei Yang, Yifan Yang, Ziyi Yang, Donghan Yu, Lu Yuan, Chenruidong Zhang, Cyril Zhang, Jianwen Zhang, Li Lyna Zhang, Yi Zhang, Yue Zhang, Yunan Zhang, and Xiren Zhou. Phi-3 technical report: A highly capable language model locally on your phone, 2024. URL `https://arxiv.org/abs/2404.14219`.

Ali Ahmaditeshnizi, Wenzhi Gao, and Madeleine Udell. OptiMUS: Scalable optimization modeling with (MI)LP solvers and large language models. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 577–596. PMLR, 21–27 Jul 2024. URL `https://proceedings.mlr.press/v235/ahmaditeshnizi24a.html`.

Rimmi Anand, Divya Aggarwal, and Vijay Kumar. A comparative analysis of optimization solvers. *Journal of Statistics and Management Systems*, 20(4):623–635, 2017.

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

Nicolás Astorga, Tennison Liu, Yuanzhang Xiao, and Mihaela van der Schaar. Autoformulation of mathematical optimization models using LLMs. In *Forty-second International Conference on Machine Learning*, 2025. URL `https://openreview.net/forum?id=33YrT1j0O0`.

Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. Small language models are the future of agentic ai. *arXiv preprint arXiv:2506.02153*, 2025.

Michael L Bynum, Gabriel A Hackebeil, William E Hart, Carl D Laird, Bethany L Nicholson, John D Siirola, Jean-Paul Watson, David L Woodruff, et al. *Pyomo-optimization modeling in python*, volume 67 of *Springer Optimization and Its Applications*. Springer, 2021.

Wei Chen and Zhiyuan Li. Octopus v2: On-device language model for super agent, 2024. URL `https://arxiv.org/abs/2404.01744`.

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. URL `https://openreview.net/forum?id=80L235oVBe`.

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, September 2025b. URL `http://arxiv.org/abs/2509.22979`. arXiv:2509.22979 [cs].

Mingyue Cheng, Jie Ouyang, Shuo Yu, Ruiran Yan, Yucong Luo, Zirui Liu, Daoyu Wang, Qi Liu, and Enhong Chen. Agent-R1: Training Powerful LLM Agents with End-to-End Reinforcement Learning, November 2025. URL http://arxiv.org/abs/2511.14460. arXiv:2511.14460 [cs].

Zezhen Ding, Zhen Tan, Jiheng Zhang, and Tianlong Chen. Or-r1: Automating modeling and solving of operations research optimization problem via test-time reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 228–236, 2026.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. Retool: Reinforcement learning for strategic tool use in LLMs. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=tRk1nofSmz.

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, 2024. URL https://arxiv.org/abs/2405.13144. Publisher: arXiv Version Number: 2.

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*, 2025. URL https://openreview.net/forum?id=9OMvtboTJg.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626, 2023.

Zhong Li, Zihan Guo, Xiaohan Lu, Juntao Wang, Jie Song, Chao Shen, Jiageng Wu, and Mingyang Sun. OptArgus: A multi-agent system to detect hallucinations in LLM-based optimization modeling, 2026a. URL https://arxiv.org/abs/2605.11738. arXiv:2605.11738.

Zhong Li, Xin Huang, Jinhui Wan, Xiangyi Wang, Shenkai Zhang, Ruiqi Chen, Wenyu Liu, Zaiwen Wen, and Ziyan Luo. Right answers, costly models: The efficiency gap in LLM-based optimization modeling, 2026b. URL https://arxiv.org/abs/2609.38884. arXiv:2609.38884.

Zhong Li, Hongliang Lu, Tao Wei, Yuxuan Chen, Wenyu Liu, Yuan Lan, Fan Zhang, and Zaiwen Wen. Constructing industrial-scale optimization modeling benchmark. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, pages 68993–69036. PMLR, 2026c. URL https://proceedings.mlr.press/v306/li26dz.html.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

Fan Liu, Zhe-Rui Yang, Cancheng Liu, Tianrui SONG, Xiaofeng Gao, and Hao Liu. MM-agent: LLM as agents for real-world mathematical modeling problem. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2026a. URL https://openreview.net/forum?id=o8n5oNDsiq.

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*, 2026b. URL https://openreview.net/forum?id=Ej20yjWMCj.

Weiting Liu, Han Wu, Yufei Kuang, Xiongwei Han, Tao Zhong, Jianfeng Feng, and Wenlian Lu. Automated optimization modeling via a localizable error-driven perspective, 2026c. URL https://arxiv.org/abs/2602.11164. arXiv:2602.11164.

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.

Ganesh Prasath and Shirish Karande. Synthesis of mathematical programs from natural language specifications. *arXiv preprint arXiv:2304.03287*, 2023.

Rindranirina Ramamonjison, Timothy Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong Zhang. NL4Opt Competition: Formulating Optimization Problems Based on Their Natural Language Descriptions. In *Proceedings of the NeurIPS 2022 Competitions Track*, volume 220 of *Proceedings of Machine Learning Research*, pages 189–203. PMLR, 2022. URL https://proceedings.mlr.press/v220/ramamonjison23a.html.

Ning Shang, Yifei Liu, Yi Zhu, Li Lyna Zhang, Weijiang Xu, Xinyu Guan, Buze Zhang, Bingcheng Dong, Xudong Zhou, Bowen Zhang, Ying Xin, Ziming Miao, Scarlett Li, Fan Yang, and Mao Yang. rStar2-Agent: Agentic Reasoning Technical Report, August 2025. URL http://arxiv.org/abs/2508.20722. arXiv:2508.20722 [cs].

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

Ajay Singh. An overview of the optimization modelling applications. *Journal of Hydrology*, 466: 167–182, 2012.

Zhengyang Tang, Zihan Ye, Chenyu Huang, Xuhan Huang, Chengpeng Li, Sihang Li, Guanhua Chen, Ming Yan, Zizhuo Wang, Hongyuan Zha, Dayiheng Liu, and Benyou Wang. CALM Before the STORM: Unlocking Native Reasoning for Optimization Modeling. In *Submitted to The Fourteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=DILQqCQIJ3.

Jiageng Wu, Chao Shen, Zihan Guo, Junyan Shao, Zongyan Zhang, Jie Song, Zhong Li, and Mingyang Sun. Fix what's wrong, keep what's right: Hallucination repair for LLM-based optimization modeling, 2026. URL https://arxiv.org/abs/2610.12529. arXiv:2610.12529.

Ziyang Xiao, Dongxiang Zhang, Yangjun Wu, Lilin Xu, Yuan Wang, Xiongwei Han, Xiaojin Fu, Tao Zhong, Jia Zeng, Mingli Song, and Gang Chen. CHAIN-OF-EXPERTS: WHEN LLMS MEET COMPLEX OPERATIONS RESEARCH PROBLEMS. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=HobyL1B9CZ.

Ziyang Xiao, Jingrong Xie, Lilin Xu, Shisi Guan, Jingyan Zhu, Xiongwei Han, Xiaojin Fu, WingYin Yu, Han Wu, Wei Shi, Qingcan Kang, Jiahui Duan, Tao Zhong, Mingxuan Yuan, Jia Zeng, Yuan Wang, Gang Chen, and Dongxiang Zhang. A survey of optimization modeling meets LLMs: Progress and future directions. In James Kwok, editor, *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-25*, pages 10742–10750. International Joint Conferences on Artificial Intelligence Organization, 8 2025. doi: 10.24963/ijcai.2025/1192. URL https://doi.org/10.24963/ijcai.2025/1192. Survey Track.

Wantong Xie, Yi-Xiang Hu, Jieyang Xu, Feng Wu, and Xiangyang Li. Murka: Multi-reward reinforcement learning with knowledge alignment for optimization tasks. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Zhenghai Xue, Longtao Zheng, Qian Liu, Yingru Li, Xiaosen Zheng, Zejun MA, and Bo An. SimpleTIR: End-to-end reinforcement learning for multi-turn tool-integrated reasoning. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=EplNy91Xqh.

Zhao Yang, Yuxuan Jiang, Ting-Chih Chen, Lincen Yang, Annie Wong, Chao Gao, Jacob E. Kooi, Zhong Li, Jiayang Shi, Kevin Qiu, Qi Huang, Xinrui Zu, Shiping Yang, Hengyuan Zhang, Ngai Wong, Filip Ilievski, Shujian Yu, Aske Plaat, Zhaochun Ren, Mark Hoogendoorn, and Vincent François-Lavet. Modularized reinforcement learning on LLMs: From MDP creation to exploration and learning, 2026. URL https://arxiv.org/abs/2606.21943. arXiv:2606.21943.

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, October 2024. URL http://arxiv.org/abs/2407.09887. arXiv:2407.09887 [cs].

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan 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, Mu Qiao, Yonghui Wu, and Mingxuan Wang. DAPO: An Open-Source LLM Reinforcement Learning System at Scale, May 2025. URL http://arxiv.org/abs/2503.14476. arXiv:2503.14476 [cs].

Guibin Zhang, Hejia Geng, Xiaohang Yu, Zhenfei Yin, Zaibin Zhang, Zelin Tan, Heng Zhou, Zhong-Zhi Li, Xiangyuan Xue, Yijiang Li, Yifan Zhou, Yang Chen, Chen Zhang, Yutao Fan, Zihu Wang, Songtao Huang, Francisco Piedrahita Velez, Yue Liao, Hongru WANG, Mengyue Yang, Heng Ji, Jun Wang, Shuicheng YAN, Philip Torr, and LEI BAI. The landscape of agentic reinforcement learning for LLMs: A survey. *Transactions on Machine Learning Research*, 2026. ISSN 2835-8856. URL https://openreview.net/forum?id=RY19y2RI1O. Survey Certification.

Hanchen Zhang, Xiao Liu, Bowen Lv, Xueqiao Sun, Bohao Jing, Iat Long Iong, Zhenyu Hou, Zehan Qi, Hanyu Lai, Yifan Xu, Rui Lu, Hongning Wang, Jie Tang, and Yuxiao Dong. AgentRL: Scaling Agentic Reinforcement Learning with a Multi-Turn, Multi-Task Framework, October 2025. URL https://openreview.net/forum?id=zq3vAmuUk9.

Jihai Zhang, Wei Wang, Siyan Guo, Li Wang, Fangquan Lin, Cheng Yang, and Wotao Yin. Solving General Natural-Language-Description Optimization Problems with Large Language Models. In Yi Yang, Aida Davani, Avi Sil, and Anoop Kumar, editors, *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 6: Industry Track)*, pages 483–490, Mexico City, Mexico, June 2024. Association for Computational Linguistics. URL https://aclanthology.org/2024.naacl-industry.42.

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. URL https://openreview.net/forum?id=ZrgxU8WMmG.

Zilin Zhu, Chengxing Xie, Xin Lv, and slime Contributors. slime: An llm post-training framework for rl scaling. https://github.com/THUDM/slime, 2025. GitHub repository. Corresponding author: Xin Lv.

Appendix for *PEARL: Solver-in-the-Loop Interactive Optimization Modeling from Natural Language*

Table of Contents

A Design Details on PEARL

- A.1 Comparison with RL-Based Methods
- A.2 Reward Function Design
- A.3 PEARL Training Algorithm
- A.4 Training and Inference Loop
- A.5 Training Hyperparameters

B Experiment Setup

- B.1 Workflow Overview
- B.2 Training Setup and Loss Masking
- B.3 Inference Setup
- B.4 8B Model Comparison

C Failure Mode Attribution

D Ablation Study and Sensitivity Analysis

- D.1 Supervised Fine-Tuning Initialization
  - D.1.1 Theoretical Analysis
  - D.1.2 Experiments
- D.2 Adaptive Sampling with Trajectory Filtering
  - D.2.1 Theoretical Analysis
  - D.2.2 Experiments
- D.3 Training Dynamics
- D.4 Tool-Use Depth and Termination

E PEARL Rollout Example

F Prompt Templates

- F.1 Training Prompt
- F.2 Direct Testing Prompt

G Training Data and Benchmark Documentation

H Additional Experiments and Protocols

I Limitations and Broader Impact

A Design Details on PEARL

A.1 Comparison of PEARL and other RL-based Methods

Table 2 compares PEARL with recent reinforcement learning–based methods for optimization modeling, including SIRL [Chen et al., 2025a], OR-R1 [Ding et al., 2026], MURKA [Xie et al., 2025], StepORLM [Zhou et al., 2026], and STORM [Tang et al., 2025]. These methods make important progress by replacing purely supervised imitation with solver- or preference-grounded learning signals. Their training targets and interaction architectures differ: for example, MURKA combines GRPO-trained extraction, a distilled Solver, and an iterative Checker. Thus, the distinction is not that prior systems never perform iterative verification, but how solver interaction enters the learned policy.

The key distinction is where RL is applied. Many prior RL-based methods optimize *outputs* or individual workflow components: the learned behavior is to make a response more likely to be correct, executable, or process-consistent. PEARL instead applies RL to the *modeling process*. The trajectory consists of multiple interaction steps, where the policy can decide to draft, revise, execute code, inspect solver feedback, and terminate. Thus, PEARL does not merely learn what a good final solver program looks like; it learns a control policy for navigating the solve–debug–revise loop.

PEARL treats solver interaction as part of the environment. In SIRL [Chen et al., 2025a], solver results provide rewards for completed programs during training. In MURKA [Xie et al., 2025], a

dedicated Checker tests candidate solutions and sends corrective feedback to the Solver. In PEARL, multi-turn RL trains the model to manage the interaction itself: it chooses when to run code and uses the output to decide whether to revise or finish. The model receives execution output and errors, including solver diagnostics or checks printed by its code. Only the external scorer compares the final objective value with the reference; the model does not receive the reference or the comparison result.

This agentic formulation gives PEARL three practical advantages. First, it supports *adaptive computation*: the policy can stop early on easy instances and spend additional tool calls on harder ones. Second, it supports *solver-aware repair*: errors are not only penalized after the fact, but are observed and acted upon during the trajectory. Third, it supports *long-horizon credit assignment*: choices such as indexing schemes, data parsing, or early execution can receive credit based on their downstream effect on final solver-verified correctness. These advantages are especially relevant for optimization modeling, where many failures are not visible from the generated text alone and only emerge through execution.

The comparison in Table 2 should therefore be read as a methodological distinction rather than a claim that prior RL methods are ineffective. Prior methods improve the quality of optimization-modeling outputs under static or weakly interactive generation. PEARL targets a different training problem: learning an agent that can use solver feedback during generation as an observable signal for deciding when to execute, how to revise, and when to terminate.

Table 2: Comparison of RL-based methods for optimization modeling. The table summarizes what each method learns and how it uses solver feedback during training or inference.

Method	**Learning target**	**Role of solver feedback**
SIRL	Outcome-based learning of complete generations	Terminal execution, feasibility, and objective-based scoring.
OR-R1	Output-level RL	Solver outcomes score generated solutions.
MURKA	GRPO-trained Extractor and distilled Solver	A separate Checker iteratively verifies solutions through solver feedback.
StepORLM	Generative process supervision and preference learning	Solver outcomes and process feedback guide reasoning-trajectory learning.
STORM	Curriculum-guided output learning	Outcome signals support curriculum-shaped alignment.
PEARL	Multi-turn policy for execution, revision, and termination	Solver output conditions later actions; trajectory rewards combine format and objective agreement.

A.2 Details on Reward Function Design

The reward function is designed to encourage correct formatting, proper tool usage, and accurate optimization modeling. The total reward is decomposed into format rewards and an objective-agreement reward:

- **Format Reward (r_{fmt}):**
  - A reward of 0.1 is assigned once if the complete response contains both `<interpreter>` and `</interpreter>`. This is a tag-presence check, not a check of execution success or a per-call bonus.
  - A reward of 0.2 is assigned when the parser extracts a nonempty `'''python ... '''` block from the last complete `<answer>...</answer>` block. Nested answer tags and interaction tags inside the extracted code are rejected. The implementation uses triple single quotes for this format.
- **Objective-Agreement Reward (r_{corr}):** We evaluate objective agreement by executing the extracted code and comparing the resulting objective value with the reference value. Specifically, the system appends a value probe to the generated code to capture the solver's output.
  - If the extracted code executes successfully (return code 0) and the appended probe yields an objective value v_{pred}, we compare it with v_{label}. The probe reads a Pyomo

objective value or a Gurobi incumbent objective; the scorer does not independently require an optimal termination status.

- If the absolute difference $|v_{\text{pred}} - v_{\text{label}}| \leq 10^{-3}$, the solution passes the objective-agreement criterion, and a reward of 1.0 is granted.

The total reward is at most 1.3. A valid final answer with an accepted objective can receive 1.2 without any tool call. If the final code cannot be extracted, both the 0.2 final-format bonus and the objective-agreement reward are zero; the interpreter-tag bonus is evaluated independently. Thus, although the components are additive, objective scoring depends on successful final-code extraction.

Multiple optima and missing references. The scorer compares objective values, not a canonical decision vector, so distinct optimal solutions (including degenerate optima) are accepted when they return the reference objective. Any internal transformation of the objective must be converted back to the problem's original objective definition before comparison. This does not check every intended constraint. Without a reference objective, the current training scorer can award only format bonuses; such instances are excluded from the reported training corpus. Pure constraint-satisfaction problems require a separate assignment verifier and are not evaluated here.

A.3 PEARL Training Algorithm

Algorithm 1 gives the training loop corresponding to the objective in Section 3.4. The algorithm starts from a reference policy π_{ref}, which is either an instruction-tuned backbone or an SFT-warm-started model depending on the experiment. Each training iteration samples optimization-modeling instances from $\mathcal{D}$ and rolls out the current policy in the PEARL environment. A rollout is not a single completion: it is a multi-turn trajectory in which the agent may draft a formulation, invoke Python or solver tools, observe execution feedback, revise the formulation or implementation, and eventually emit a final answer.

The return $R(\tau_{i,j})$ is computed at the trajectory level using the reward components described in Appendix A.2: format rewards check interpreter-tag presence and final-code extraction, while the terminal objective-agreement reward requires successful execution and an accepted objective; execution and feasibility are not separately rewarded. We then convert returns into normalized advantages, so credit is assigned to complete solver-grounded interaction traces rather than to isolated text spans. This design is important in optimization modeling because early actions, such as choosing an indexing scheme or deciding when to execute, may only reveal their value after several tool calls.

Finally, the policy is updated with the GRPO-style objective described in Section 3.4, using DAPO-inspired decoupled clipping with thresholds ϵ_+ and ϵ_-. In the main PEARL runs, KL and entropy coefficients are set to zero after initialization; the update is therefore driven by solver-grounded returns and stabilized by clipping and within-problem group normalization. The high-level loop in Algorithm 1 omits engineering details such as loss masking over tool observations, execution timeouts, and distributed rollout scheduling, which are summarized in Appendix B.2 and Table 3.

Algorithm 1: PEARL: NL-to-Opt via Agentic RL

Input: Dataset $\mathcal{D}$ of optimization modeling instances; reference policy π_{ref}; clipping thresholds ϵ_+, ϵ_-
Initialize policy $\pi_\theta \leftarrow \pi_{\text{ref}}$
while *not converged* **do**
  foreach $x_i \sim \mathcal{D}$ **do**
    Sample $G = 8$ multi-turn, tool-augmented trajectories $\{\tau_{i,j}\}_{j=1}^{G} \sim \pi_\theta(\cdot \mid x_i)$
      // NL modeling + execution + solver calls
    Compute each trajectory return $R(\tau_{i,j})$
      // Format and terminal objective agreement
    Compute $A_{i,j}$ by normalizing returns within group i
      // Credit assignment for optimization modeling
  Update π_θ via decoupled-clipped policy gradient
    // Agentic RL over multi-turn optimization modeling

Figure 5 complements the pseudo-code: both training and inference expose execution output to the policy during generation. After generation, the training scorer separately executes the final code and compares its objective with the reference to compute the terminal reward.

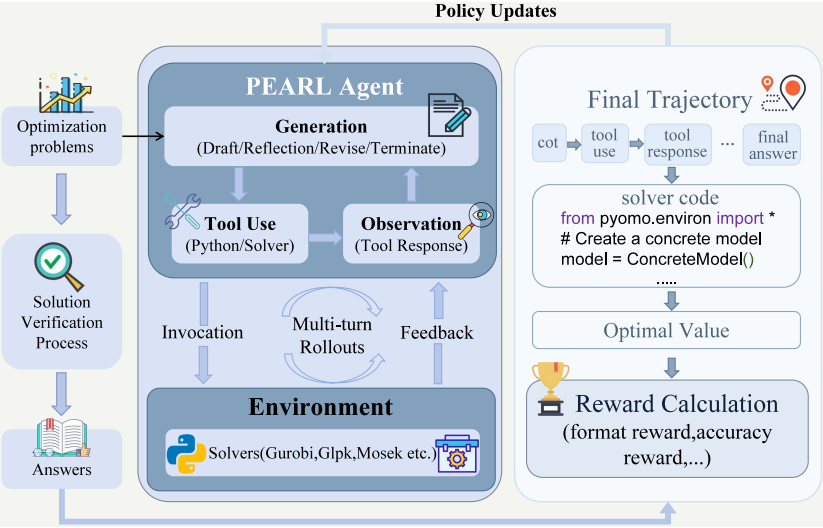


Figure 5: Overview of the PEARL training and inference loop for optimization modeling with tool use. The right-hand branch depicts external training-time scoring; its objective-value probe does not certify optimal solver termination.

Table 3: Main hyperparameters for Qwen3-4B-Instruct-2507 PEARL training.

Hyperparameter	**Value**
Base model	Qwen3-4B-Instruct-2507
Hardware	1 node, 4 high-performance GPUs, Ray-based training
Rollout engine	SGLang tool-call rollout (colocated), 4 GPUs per engine
Algorithm	GRPO with decoupled clipping (ϵ=0.2, ϵ_{high}=0.28)
KL regularization	disabled in main runs, coefficient 0.0
Entropy regularization	coefficient 0.0
Configured rollout iterations	3000, shuffle enabled
Samples per prompt	8
Rollout batch size	16 prompts per rollout batch
Global batch size	128
Max response length	16384 tokens
Sampling temperature	0.8
Optimizer	Adam, constant LR
Learning rate	1×10^{-6}
Adam betas	$(0.9, 0.98)$
Weight decay	0.1
Parallelism	tensor MP=2, context MP=2, pipeline MP=1, sequence parallel on
Memory / efficiency	dynamic batch size, max tokens per GPU 9216, activation recompute enabled
Checkpointing	save interval 20
PEARL-specific tool-use settings	
Max generation turns per rollout	8, including the final-answer turn
Tool-output truncation	First 2000 characters, plus truncation notice
Tool concurrency	64
Training Python execution timeout	120 seconds
Python tool memory limit	4 GB
Return log probabilities	enabled

B Experiment Setup

B.1 Workflow overview

As shown in Figure 5, we cast optimization modeling from natural language as an interactive decision process. Given a problem description, the PEARL agent incrementally constructs the final solution via staged generation (Draft/Reflection/Revise/Terminate) and queries external tools (Python / solvers) to obtain verifiable feedback such as execution traces, solver status, and objective values. Each interaction yields a multi-turn trajectory that contains intermediate reasoning, tool calls, tool observations, and the final modeling write-up with runnable code. After generation, the external scorer extracts and executes the final code and compares the extracted objective value with the

reference. The trajectory-level format bonuses and objective-agreement reward are then combined for policy updates; the reference comparison is not returned as an in-loop observation.

The maximum turn budget is only an upper bound: the policy can terminate early by emitting the final <answer> action. In practice, tool use stabilizes at a small number of calls for most instances, supporting a bounded-horizon interaction policy.

B.2 Training setup

Implementation and tool environment. We implement training with the SLIME framework [Zhu et al., 2025], using its tool-augmented rollout engine and distributed training stack. Rollouts are generated with an engine that supports multi-turn tool calling, where the model can emit <code>...</code> blocks to run Python in a sandbox and observe results in <interpreter>...</interpreter> blocks, and finally return a single <answer>...</answer> block containing the finalized formulation and verified runnable code. We use Pyomo Bynum et al. [2021] as the modeling interface for optimization formulations. An example of PEARL Agent solving an optimization problem can be found in Appendix E.

We train the policy with GRPOShao et al. [2024] . Table 3 summarizes the main hyperparameters for our Qwen3-4B-Instruct-2507 experiments.

The configured 3,000 rollout iterations are not 3,000 individual trajectories: a complete iteration contains 16 problem groups of eight trajectories. Returns are normalized within each group. The implementation has a batch-wide fallback when the sample count differs from the configured full batch; this fallback is distinct from the main full-batch procedure. Online evaluation during training uses the absolute objective-error tolerance 10^{-3} without format bonuses. Its accuracy curves should not be treated as evaluations under the final benchmark threshold in Section 4.

Loss masking for observation tokens. In our policy optimization procedure, the token-level losses are computed over multi-turn rollout sequences. In PEARL, each rollout sequence consists of both agent-generated tokens (modeling steps, reasoning, tool invocations in <code>...</code> blocks) and observation tokens returned by external tools (execution traces, solver outputs, error messages in <interpreter>...</interpreter> blocks). While optimizing agent-generated tokens enhances the model's ability to formulate optimization problems, interact with solvers, and perform multi-turn reasoning, applying the same optimization to observation tokens can lead to unintended learning dynamics, as these tokens represent deterministic tool outputs rather than learned agent behavior. To address this, we introduce loss masking for observation tokens, ensuring the policy gradient objective is computed only over agent-generated tokens, excluding tool-returned observations from the optimization process. This approach stabilizes training while preserving the agent's ability to leverage solver feedback for iterative model refinement.

B.3 Inference setup

For evaluation, we deploy the trained PEARL model using vLLM for efficient inference. Table 4 summarizes the inference configuration.

Table 4: Inference parameters for Qwen3-4B-Instruct-2507 PEARL evaluation.

Parameter	**Value**
Provider	vLLM (optrl_vllm)
Model	Qwen/Qwen3-4B-Instruct-2507
Tokenizer	Qwen/Qwen3-4B-Instruct-2507
Max model length	16384 tokens
Max output tokens	16384 tokens
LLM configuration timeout	300 seconds (see note below)
Engine configuration	
Tensor parallel size	1
GPU memory utilization	0.8
Sampling parameters	
Temperature	0.6
Top-p	0.95
Top-k	50

The 300-second entry is an LLM configuration field, not a whole-trajectory time budget. The local `optrl_vllm` client does not enforce this field around generation. Python tool execution has a separate timeout; the inference tool configuration sets it to 120 seconds per call.

B.4 8B Model Comparison

Table 5: Pass@1 accuracy (%) for 8B-related comparisons. The 8B rows are reported in the appendix to keep the main table focused on the primary 4B PEARL model and representative baselines.

Method	NL4Opt	MAMO-E	MAMO-C	NLP4LP	OptMATH*	IndustryOR	ComplexOR	Resocratic	Macro Avg	Micro Avg
Qwen-3-8B	38.32±1.60	38.53±2.20	12.61±2.30	46.07±1.90	3.01±0.55	2.38±1.35	5.56±2.78	37.47±1.70	22.99±0.70	32.56±0.60
PEARL-Qwen-3-8B	86.45±1.10	94.68±0.75	57.66±3.40	94.38±1.20	24.10±1.65	64.29±6.20	38.89±4.80	81.39±0.90	67.73±0.55	79.61±0.30
PEARL-Qwen-3-4B-Instruct-2507	85.28±0.99	91.65±0.65	76.12±3.19	93.25±1.59	16.87±1.70	67.86±8.42	41.66±3.92	84.99±0.52	69.71±0.13	79.84±0.25

Table 5 shows that PEARL training yields a large improvement over the raw Qwen-3-8B baseline, confirming that the learned interaction policy is beneficial beyond the 4B backbone. Comparing PEARL-Qwen-3-8B with PEARL-Qwen-3-4B-Instruct-2507, the 8B model improves on some easier or scale-sensitive benchmarks but does not dominate the 4B model overall, suggesting that both backbone choice and training data/interface stability matter. These backbone comparisons do not isolate model size from differences in initialization and training behavior.

C Failure Mode Attribution

Figure 4 in the main text summarizes the failure attribution analysis. We include the detailed setup and interpretation here to avoid overloading the compact main-text figure.

Attribution setup. We analyze failures from the Qwen3-4B-Instruct base model and the PEARL-Qwen3-4B-Instruct model under the main pass@1 evaluation setting. For each instance that fails the pass@1 correctness criterion, we assign one primary failure category using the execution-and-verification pipeline. The categories are: *code/runtime*, where the generated program crashes, imports fail, variables are undefined, parsing fails, or no executable solver result can be obtained; *format/extraction*, where the response violates the required answer or code format or the evaluator cannot reliably extract the intended implementation; *timeout*, where execution exceeds the configured budget; *infeasible/solver*, where the generated model reaches the solver but produces an infeasible, unbounded, or otherwise solver-side invalid status; *objective mismatch*, where the code executes and produces a candidate solution but the objective value does not match the reference within tolerance; and *other*, which covers residual failures that do not cleanly fit the preceding categories. Each bar in Figure 4 is normalized within the corresponding set of failed cases, so the figure describes the composition of failures rather than the overall pass rate.

Rows in the figure. The first two rows compare the failure composition before and after PEARL training. The first row aggregates failures of Qwen3-4B-Instruct under the PEARL evaluation pipeline, while the second row aggregates failures of PEARL-Qwen3-4B-Instruct. The third row is a paired subset analysis: it considers only instances that the base model failed but PEARL later solved, and attributes those solved cases according to the base model's original failure mode. This row therefore answers a different question from the first two rows: instead of describing what PEARL still gets wrong, it identifies which base-model failure modes PEARL most often repairs.

Main interpretation. The base model's failures are dominated by code execution errors, which account for 66.2% of its failed cases. This means that many errors occur before the evaluator can even assess the mathematical quality of the generated optimization formulation: the model may emit non-runnable code, inconsistent variable names, invalid data handling, or solver calls that fail at runtime. After PEARL training, the share of code execution failures drops to 14.3%, indicating that interactive execution and solver feedback teach the model to produce substantially more executable solver programs.

The PEARL failure distribution is instead dominated by objective mismatch, which accounts for 53.4% of PEARL's remaining failed cases. This increase should be interpreted as a conditional shift among failures, not as evidence that PEARL creates more objective errors overall. Because PEARL fixes many low-level execution and formatting failures, the residual error mass becomes concentrated in harder cases where the code runs but the formulation, objective, indexing, or constraints still do

not match the reference semantics. In other words, agentic training moves the dominant failure mode from "cannot execute or parse" toward "executes but is mathematically wrong," which is a more substantive optimization-modeling failure.

The paired subset in the third row further clarifies what PEARL repairs. Among base-model failures that PEARL later solves, 73.8% were originally code execution failures, followed by objective mismatch (12.6%) and formatting or extraction errors (9.2%). This supports the claim that the learned interaction policy is especially effective at implementation repair: PEARL learns to run partial code, inspect exceptions or solver messages, and revise the program until it reaches a valid execution path. It also repairs some modeling-level failures, but the dominant improvement comes from making the solver-grounded workflow operational.

Finally, the remaining PEARL failures show the limits of solver feedback as an oracle. Solver logs and compiler errors are highly useful for detecting runtime, feasibility, and formatting problems, but they do not fully identify all semantic mismatches between the intended optimization problem and the generated formulation. Objective-value checks catch many such mismatches when references are available, yet they remain partial: different formulations can agree on an objective for one instance but differ structurally, while ambiguous problem statements may admit multiple plausible interpretations. This motivates the broader validation agenda discussed in Appendix I.

D Ablation Study and Sensitivity Analysis

D.1 Supervised Fine-Tuning Initialization

D.1.1 Theoretical Analysis

Alternatively, before agentic reinforcement learning, we can initialize the policy using supervised fine-tuning (SFT) to obtain a stable and competent reference model for subsequent multi-turn optimization. This initialization follows standard practice in optimization modeling and tool-augmented language modeling, and is not intended to encode interactive decision-making or solver-driven control behaviors.

Training data. The SFT stage uses a mixture of publicly available optimization modeling datasets and curated instruction-style data, including OR-Instruct from ORLM [Huang et al., 2025], OptMATH [Lu et al., 2025], and OptiBench-style modeling instances [Yang et al., 2024]. These datasets consist of paired natural-language problem descriptions and corresponding solver-ready formulations or executable code, covering a broad range of optimization domains, constraint structures, and problem scales.

Training objective. Supervised fine-tuning is performed using standard autoregressive next-token prediction over *complete* solution traces, which may include natural-language reasoning, mathematical formulations, and solver code. Formally, let $\mathcal{D}_{\mathrm{SFT}}$ denote the supervised dataset of input–output pairs $(x, \mathbf{y})$, where x is a natural-language optimization prompt and $\mathbf{y} = (y_1, \ldots, y_{|\mathbf{y}|})$ is the target output sequence. The SFT objective is defined as:

$$\mathcal{L}_{\mathrm{SFT}}(\theta) = -\mathbb{E}_{(x,\mathbf{y})\sim\mathcal{D}_{\mathrm{SFT}}}\left[\sum_{t=1}^{|\mathbf{y}|} \log p_\theta(y_t \mid x, \mathbf{y}_{<t})\right], \tag{5}$$

where θ denotes the model parameters and $p_\theta(y_t \mid x, \mathbf{y}_{<t})$ is the conditional probability of generating the t-th token given the input prompt and all preceding tokens.

Importantly, SFT training is restricted to *single-pass generation*: the model is not exposed to multi-turn interaction, intermediate solver feedback, partial execution results, or revision behavior during this stage.

Role of SFT in PEARL. The purpose of SFT is to endow the base model with fundamental optimization modeling competence and syntactic validity, ensuring that generated formulations and solver code are executable at the outset of reinforcement learning. The resulting model defines the reference policy π_{ref}, which serves solely as an initialization for agentic reinforcement learning. All interactive behaviors in PEARL—such as deciding when to invoke solvers, how to interpret solver diagnostics, and whether to revise or terminate—are learned exclusively during the agentic reinforcement learning stage described in Section 3.4.

D.1.2 Experiments

In our main experiments, we directly apply reinforcement learning to instruction-tuned models. However, when we attempted to train Qwen3-8B-Base (a base model without instruction tuning), we observed poor performance and training instability. As shown in Figure 6, the model struggles to follow the required output format in early training stages, leading to training collapse with rewards declining after approximately 60 steps. This suggests that for base models without instruction tuning, a supervised fine-tuning (SFT) warm-start is necessary to establish basic format-following capabilities.

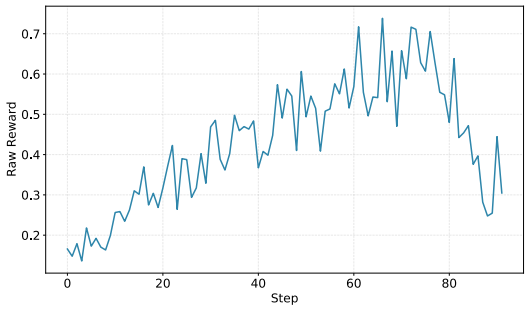


Figure 6: Training reward curve when directly applying PEARL to Qwen3-8B-Base without prior instruction tuning. The reward begins to decline after 60 steps, indicating training collapse due to the model's inability to consistently follow the required output format.

Recall that, to obtain SFT data, we use a curated set of supervised solution traces. We conduct an ablation study to evaluate whether performing SFT before RL benefits the PEARL agent. As shown in Table 6, SFT alone (SFT-Qwen-3-4B-Instruct) substantially improves the model's capabilities compared to the base model. However, the SFT-only model does not reach the performance ceiling of PEARL. Under this controlled ablation setting, adding PEARL after SFT improves MAMO-C from 55.86 to 76.57, while direct PEARL from the instruction-tuned model reaches 63.96 on MAMO-C but remains comparable in Macro/Micro average (70.40/80.38 versus 69.63/80.32 for PEARL after SFT). This appendix ablation is separate from the final main-result checkpoint reported in Table 1, where the PEARL MAMO-C result is 76.12 ± 3.19. The results indicate that while SFT provides a useful warm-start for base models, instruction-tuned models can achieve similar final performance through direct RL training, bypassing the need for an intermediate SFT stage.

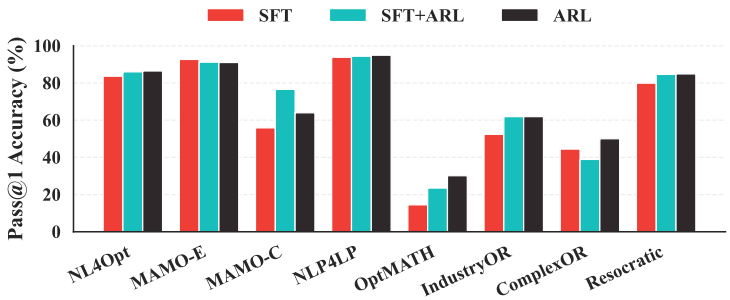


Figure 7: Analyzing the role of SFT prior to agentic RL. Results are obtained using the same base model, Qwen3-4B-Instruct, under three training regimes: (i) SFT only, (ii) SFT followed by agentic RL (SFT+ARL), and (iii) direct agentic RL without SFT (ARL).

D.2 Adaptive Sampling with Trajectory Filtering

D.2.1 Theoretical Analysis

Agentic RL training, which incorporates external feedback from tool interactions, is highly susceptible to training instability and model collapse [Xue et al., 2026, Shang et al., 2025]. Tool-augmented

Table 6: Ablation study on the role of supervised fine-tuning (SFT) before PEARL training. We compare three controlled ablation settings using Qwen3-4B-Instruct as the base: (1) SFT only, (2) SFT followed by PEARL training, and (3) direct PEARL training without SFT. This ablation is separate from the final main-result checkpoint in Table 1. Results show that while SFT provides substantial gains, the ultimate performance ceiling is similar whether SFT is used as an intermediate step or not.

Method	NL4Opt	MAMO-E	MAMO-C	NLP4LP	OptMATH	IndustryOR	ComplexOR	Resocratic	Macro Avg	Micro Avg
SFT-Qwen-3-4B-Instruct	83.64	**92.66**	55.86	93.82	14.46	52.38	44.44	79.90	64.64	76.86
PEARL-SFT-Qwen-3-4B-Instruct	85.98	91.19	**76.57**	94.38	23.49	**61.90**	38.89	84.62	69.63	80.32
PEARL-Qwen-3-4B-Instruct	**86.45**	91.01	63.96	**94.93**	**30.12**	**61.90**	**50**	**84.86**	**70.40**	**80.38**

rollouts may fail for reasons unrelated to modeling quality, including malformed tool calls, runtime crashes, and solver timeouts. Treating such failures as ordinary negative rewards introduces substantial variance and biases the policy update toward avoiding tool use altogether. To ensure stable training and achieve better model performance, prior work has adopted trajectory filtering strategies to exclude obviously anomalous negative samples [Xue et al., 2026, Shang et al., 2025]. To verify whether such filtering is effective in the optimization modeling scenario, we investigate a dynamic sampling strategy with trajectory filtering, inspired by DAPO-style filtering ideas [Yu et al., 2025] but adapted to optimization modeling.

For each modeling instance x_i, we repeatedly sample trajectories $\tau_{i,j}$ from the current policy until a minimum number of valid trajectories is obtained. Formally, let $\tau_{i,j}$ denote the j-th sampled trajectory for instance x_i, and let $q(\tau_{i,j}) \in \{0, 1\}$ be a binary quality indicator that tests whether the trajectory is usable for learning. We define the set of retained trajectories as

$$\begin{aligned} \mathcal{V}_i &= \{\, j \le K_i \;:\; q(\tau_{i,j}) = 1 \,\}, \\ K_i &= \min\left\{ k \;:\; \sum_{j \le k} q(\tau_{i,j}) \ge K_{\min} \right\} \wedge K_{\max}, \end{aligned} \tag{6}$$

where $K_{\min}$ specifies the minimum number of valid trajectories required for instance x_i, and $K_{\max}$ caps the total number of rollouts to control computational cost.

The quality gate $q(\tau)$ filters out trajectories that are structurally invalid or uninformative, including malformed tool-call schemas, incomplete final outputs, or execution failures that occur before any meaningful solver interaction. By conditioning learning on $\mathcal{V}_i$ rather than on all sampled trajectories, this mechanism reduces gradient noise while allocating more sampling budget to difficult instances with low validity rates.

D.2.2 Experiments

As described above, we adopt an adaptive sampling strategy with trajectory filtering to mitigate training instability in agentic RL. Here, we empirically evaluate whether such filtering is necessary in the optimization modeling domain. We use Qwen3-4B-Instruct-2507 as the base model and train on a combination of curated and open-source samples. Due to the large context requirement with `max_response_length` set to 32k tokens, we use 8 high-performance GPUs with context parallelization set to 4.

We explore three trajectory filtering strategies: (1) *Normal*: no filtering of negative samples; (2) *Filter Truncated*: filtering only truncated negative samples; and (3) *Filter All*: filtering both truncated and format-violating negative samples. As shown in Figure 9, under the Normal setting, the model's responses grow increasingly longer during training (reflected in extended rollout time in Figure 9c), leading to a progressively higher truncation ratio during the rollout process (Figure 9d). This results in more truncated negative samples being included in the training data.

Interestingly, despite the increasing presence of truncated negative samples, the model does not exhibit training collapse. In fact, as shown in Figure 9a, the *Normal* setting achieves slightly better performance on the benchmarks (average of MAMO-C and IndustryOR) compared to the filtering strategies. The limited number of tool calls (2--4 per problem) may help explain the stability observed in this setting. Training rewards also stabilize (Figure 9b). These results suggest that aggressive filtering is unnecessary under the evaluated conditions.

Algorithm 2: PEARL-Filter: NL-to-Opt via Agentic RL with adaptive sampling

Input: Dataset $\mathcal{D}$ of optimization modeling instances; reference policy π_{ref}; $K_{\min}, K_{\max}$; clipping thresholds ϵ_+, ϵ_-

Initialize policy $\pi_\theta \leftarrow \pi_{\text{ref}}$

while *not converged* **do**

  foreach $x_i \sim \mathcal{D}$ **do**

    $\mathcal{V}_i \leftarrow \emptyset$, $j \leftarrow 0$ //Valid solver-reaching trajectories

    while $|\mathcal{V}_i| < K_{\min}$ ***and*** $j < K_{\max}$ **do**

      Sample trajectory $\tau_{i,j} \sim \pi_\theta(\cdot \mid x_i)$ //Multi-turn NL modeling with execution and solver calls

      if $q(\tau_{i,j}) = 1$ **then**

        $\mathcal{V}_i \leftarrow \mathcal{V}_i \cup \{j\}$ //Executable and reaches solver/validation

      $j \leftarrow j + 1$

    foreach $j \in \mathcal{V}_i$ **do**

      Compute return $R(\tau_{i,j})$ //Trajectory-level reward

    Compute group-normalized advantages $A_{i,j}$ //Instance-wise credit assignment

  Update π_θ via decoupled-clipped policy gradient //Agentic RL over multi-turn optimization modeling

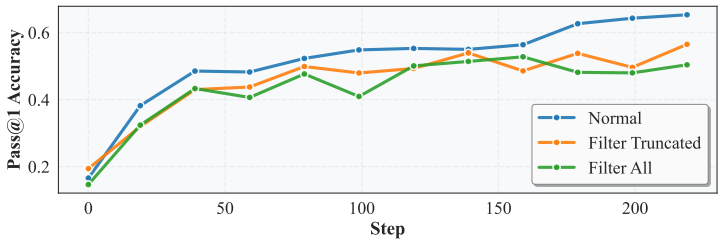


Figure 8: Impact of trajectory filtering strategies on PEARL training. We compare three settings—Normal, Filter Truncated, and Filter All—and report average Pass@1 accuracy on MAMO-C and IndustryOR throughout training.

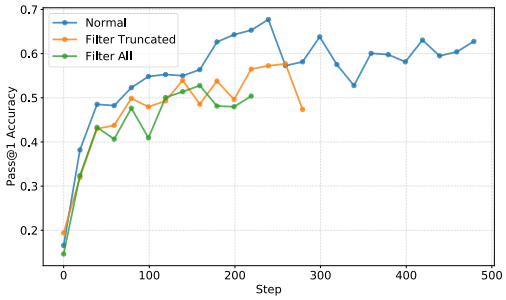
D.3 Training Dynamics

We visualize the training dynamics of PEARL in Figure 10. The model is trained using Qwen3-4B-Instruct-2507 as the base model, with open-source and curated data as the training corpus. Throughout the training process, we monitor three key benchmarks (IndustryOR, MAMO-C, and NL4LP) as evaluation metrics. As shown in Figure 10a, the Pass@1 accuracy on all three benchmarks consistently increases during training, demonstrating the model's improved problem-solving capabilities.

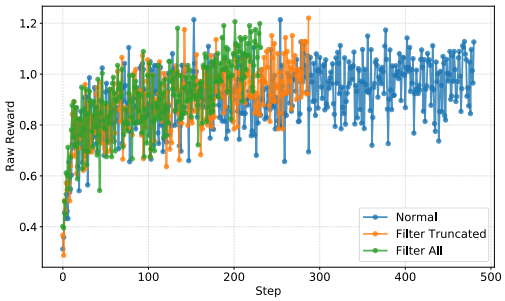
The training set reward (Figure 10b) also exhibits an upward trend, though it begins to saturate in the later stages of training. This plateau indicates limited further improvement in the measured training reward. Notably, the model rapidly learns to utilize tool calls (Figure 10d), with this metric converging early in the training process. This suggests that tool usage is one of the first capabilities the model acquires. Additionally, under GRPO training, we observe that the response length (Figure 10c) continues to grow throughout the training process, reflecting the model's tendency to generate more detailed and comprehensive solutions.

D.4 Tool-Use Depth and Termination

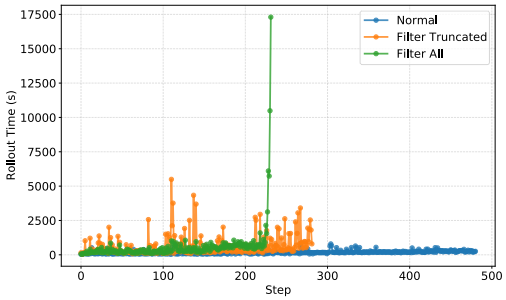
Average interaction depth alone is not sufficient evidence that PEARL benefits from a learned multi-turn policy: a fixed wrapper could also create additional opportunities for tool interaction. To isolate this effect, we evaluate the base Qwen3-4B-Instruct-2507 model under the same PEARL inference



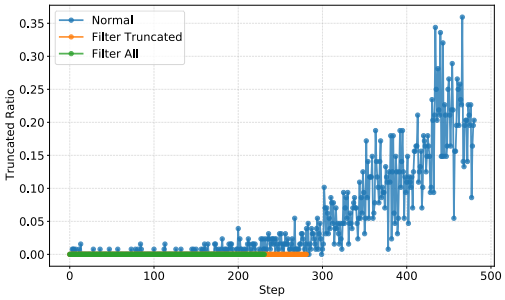
(a) Pass@1 Accuracy (Avg. of MAMO-C & IndustryOR)



(b) Raw Reward



(c) Rollout Time



(d) Truncated Ratio

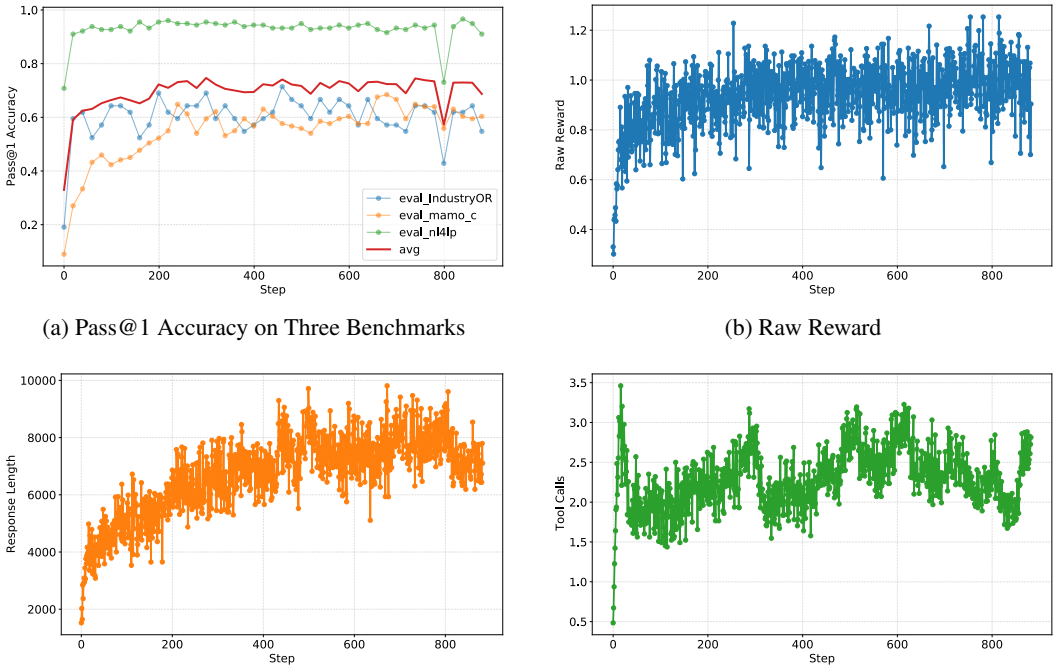
Figure 9: Impact of trajectory filtering strategies on PEARL training. We compare three settings: Normal (no filtering), Filter Truncated (filtering only truncated negative samples), and Filter All (filtering both truncated and format-violating negative samples). (a) Average Pass@1 accuracy on MAMO-C and IndustryOR benchmarks. (b) Training set raw reward. (c) Rollout time per step. (d) Truncated ratio during training.

Table 7: Scaffold-control comparison under the same PEARL inference process. Accuracy is pass@1 (%); tool calls are averaged per instance.

Benchmark	**Acc.**		**Tool Calls**	
	Base	**PEARL**	**Base**	**PEARL**
NL4Opt	68.22	85.28	0.47	1.31
MAMO-Easy	41.28	91.65	0.85	1.42
MAMO-Complex	12.61	76.12	0.15	1.55
NLP4LP	81.29	93.25	0.19	1.29
OptMATH	5.42	16.87	0.23	4.43
IndustryOR	19.05	67.86	0.69	1.62
ComplexOR	11.11	41.66	0.28	2.28
Resocratic	51.61	84.99	0.35	1.83

process, keeping the tool interface, execution-feedback loop, and repair budget fixed. For PEARL, we report the final checkpoint values from Table 1 so that the accuracy column is aligned with the main result table. Table 7 reports both pass@1 accuracy and the average number of tool calls per instance.

The base model occasionally uses the available tools, but it does so much more sparsely and less effectively than trained PEARL. PEARL achieves higher accuracy on every benchmark, with especially large gains on MAMO-C, MAMO-E, IndustryOR, and Resocratic. These results suggest that the improvement is not due to the hand-written multi-turn scaffold alone. Rather, agentic training teaches a control policy for deciding when to execute code, how to interpret solver feedback, and whether to continue revising.



(a) Pass@1 Accuracy on Three Benchmarks

(b) Raw Reward

(c) Response Length

(d) Tool Calls

Figure 10: Training dynamics of PEARL using Qwen3-4B-Instruct-2507 as the base model. The model is trained on open-source and curated data. (a) Pass@1 accuracy across three evaluation benchmarks (IndustryOR, MAMO-C, and NL4LP), showing consistent improvement during training. (b) Raw reward from the training set, demonstrating increasing performance that saturates in later stages. (c) Response length increases throughout training under GRPO optimization. (d) Tool calls rapidly converge, indicating the model quickly learns to utilize tools effectively. Online accuracy uses absolute objective error $\leq 10^{-3}$, without format bonuses.

The eight-turn inference budget is an upper bound on generation turns, not a target number of tool calls. A valid final `<answer>` action permits early termination and consumes one turn. The inference implementation can also extract fallback code when the budget is exhausted, without generating an additional answer. Figure 11 shows a concentration at one or two calls and a tail extending to eight. Counts alone do not distinguish successful final-answer termination from budget exhaustion or fallback extraction. The benchmark-level averages in Table 7 similarly remain small for most datasets, rising most noticeably on OptMATH.

Figure 3(a) and Figure 10d show tool use emerging early and stabilizing around 2–4 calls, while the training reward in Figure 10b levels off later in training. Together with the concentration of evaluation trajectories at one or two tool calls in Figure 11, these observations support the emergence of early termination behavior within the allowed budget.

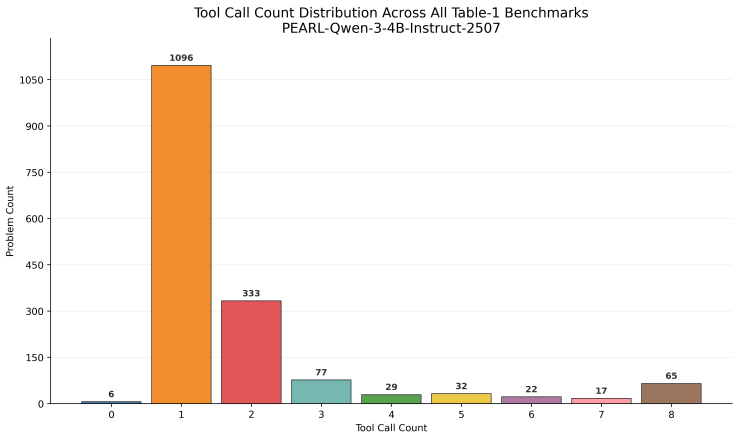


Figure 11: Distribution of tool-call counts for PEARL across the Table 1 benchmarks. Counts are concentrated at one or two calls. The eight-turn budget includes final-answer generation, so the eight-call bin must not be interpreted as eight calls followed by an additional final-answer turn.

E PEARL Rollout Example

PEARL Agent Response (Sanitized Excerpt)

We are given a complex production system with multiple stages. Let's model this as a ***mixed-integer linear program****.*

Decision Variables:

- R: raw material units; $e_{\text{sold}}, e_{\text{toT}}, e_{\text{toBS}}$: engine block allocation
- $t_{\text{sold}}, t_{\text{toBS}}$: transmission allocation; b_{sold}: brake systems sold

Objective: Maximize profit: $25e_{\text{sold}} + 35t_{\text{sold}} + 45b_{\text{sold}} - 68R - 5e_{\text{toT}} - 10e_{\text{toBS}} - 12t_{\text{toBS}}$
Key Constraints: Engine/transmission/brake system material balance, contract requirements ($b_{\text{sold}} \geq 4000$), market demand caps, machine time limit.

Now implementing in Pyomo...

`<code>` `from pyomo.environ import *; model = ConcreteModel(); ...`
`<interpreter>` `Error: Constraint 'eb_balance[1]' does not have proper value. Found function object instead of expression.`

▷ **Reflection:** *Ah, I see the issue! I'm passing the function object instead of calling it. Let me fix the constraint definition...*

`<code>` `model.eb_balance.add(eb_balance_rule(model)) # Fixed: call the rule`
`<interpreter>` `Optimal Solution: R=3772.0, e_sold=20000.0, e_toT=2399.0, ... Maximum Profit: 851534.0`

▷ **Reflection:** *We're getting non-integer values like 3772.0. However, these are actually integers displayed with .0 suffix. Let's verify the solution satisfies integrality and all constraints...*

Checking: $R = 3772$ *(integer ✓),* $e_{toT} = 2399$ *(integer ✓), all constraints satisfied ✓*

`<answer>`

Modeling: 7 decision variables (all $\mathbb{Z}^+$), 1 objective (maximize profit), 6 constraint groups (material balance, contract, demand, machine time)

Implementation: Complete Pyomo code with constraint rules, GLPK solver

```
from pyomo.environ import *
model = ConcreteModel()
# Variables
model.R = Var(domain=NonNegativeIntegers)
model.e_sold = Var(domain=NonNegativeIntegers)
...
# Constraints
model.eb_balance = ConstraintList()
model.eb_balance.add(eb_balance_rule(model))
...
# Solve
solver = SolverFactory('glpk')
results = solver.solve(model)
print(f"Maximum Profit: {value(model.objective)}")
```

Problem Description

```
AutoParts Ltd. manufactures three types of automotive components: Engine Block,
↪  Transmission, and Brake System. The company sources raw materials at a cost of $60
↪  per unit. Each unit of raw material can be processed into 7 units of Engine Block
↪  and 3 units of Transmission. Processing one unit of raw material requires $8 in
↪  energy costs and 7 hours of machine time.\n\nEngine Block can be used in three
↪  ways:\n- Sold directly for $25 per unit.\n- Processed into 1 unit of Transmission,
↪  requiring 4 hours of machine time and $5 in additional costs.\n- Processed into 1
↪  unit of Brake System, requiring 6 hours of machine time and $10 in additional
↪  costs.\n\nTransmission can be used in two ways:\n- Sold directly for $35 per
↪  unit.\n- Processed into 1 unit of Brake System, requiring 5 hours of machine time
↪  and $12 in additional costs.\n\nBrake System is sold for $45 per unit. The company
↪  has a maximum machine time capacity of 60,000 hours per year. Additionally, there
↪  is a contractual obligation to produce at least 4,000 units of Brake System
↪  annually. The maximum market demand for each component is as follows:\n- Engine
↪  Block: 20,000 units\n- Transmission: 15,000 units\n- Brake System: 12,000
↪  units\n\nDetermine how AutoParts Ltd. can maximize its annual profit while meeting
↪  all constraints.
```

F Prompt Templates

F.1 Training Prompt

RL Training Prompt (System, Pyomo)

```
You are an optimization modeling agent.

Goal:
- Convert the user problem statement into a correct optimization
↪  model and a runnable Python implementation.
- Use Python tool calls to compute, sanity-check, and verify your
↪  formulation when needed.

- The final code must be verified runnable, and you must validate the
↪  optimal value/solution against the problem statement: check all
↪  constraints are satisfied and no requirement is violated.

Modeling notes:
- Check variable meaning: if x is a quantity (integer/continuous), do
↪  NOT use binary-only logic like x_i + x_j <= 1.
```

```
- For logical constraints about whether a product is produced (x_i >
↪  0), introduce indicator y_i in {0,1}.
- Link quantity and indicator with tight bounds: x_i <= M_i * y_i;
↪  add x_i >= y_i only if selecting implies x_i >= 1.
- Encode implications with indicators: (A => B) becomes y_A <= y_B;
↪  (A => not B) becomes y_A + y_B <= 1.
- Choose M_i as small as possible using known bounds (e.g., from
↪  resource constraints) to keep the MIP stable.
- Carefully distinguish variable types. Common nouns -> types:
↪  amount/production/flow/distance/time/cost = nonnegative
↪  continuous; counts
↪  (trips/machines/workers/vehicles/orders/batches) = nonnegative
↪  integer; yes/no decisions (select/open/visit/build) = binary;
↪  assignment/matching/edge-used (i->j) = binary; sequence/rank =
↪  integer (with subtour-elimination / uniqueness constraints).
- Keep units consistent across constraints (e.g., hours vs. units, kg
↪  vs. g); unit mistakes often cause 10x/1000x objective gaps.
- Implementation sanity: check solver termination is optimal before
↪  reading values; access indexed vars as model.x[i] (not getattr).

Implementation requirement:
- Use Pyomo to implement and solve the optimization model. Your code
↪  should follow this skeleton:
from pyomo.environ import *

model = ConcreteModel()

# Solve
You may use SolverFactory('glpk') for linear/MIP problems.
You may use SolverFactory('ipopt') or SolverFactory('gurobi') for
↪  nonlinear programming (NLP).
solver = SolverFactory('glpk')
results = solver.solve(model)

Tool calling:
- When you need to run Python, output exactly:
<code>
...python code...
</code>
- After a tool call, you will receive the execution result inside:
<interpreter>
...tool output...
</interpreter>
- You may call the tool multiple times.

Final answer:
- When you are done, output exactly one final block:
<answer>
# Modeling:
define decision variables, objective, constraints (clear and
↪  complete).
# Implementation:
include runnable code wrapped as:
'''python
...python code...
'''
</answer>
```

F.2 Direct Prompts

Direct Testing Prompt (Pyomo)

````
You are a professional mathematical optimization expert. Please use
↪  Python and Pyomo to solve the following optimization problem.

# Question:
{description}

# Note:
- Use Pyomo to implement and solve the optimization model. Your code
↪  should follow this skeleton:
```python
import pyomo.environ as pyo

model = pyo.ConcreteModel()

solver = pyo.SolverFactory("glpk")
results = solver.solve(model)
```
- Make sure the model variable is named `model`.
- Carefully determine whether the variable is an integer or a
↪  continuous variable.
````

G Training Data and Benchmark Documentation

Training data. The corpus contains approximately 13.8k underlying problems: about 10k cleaned public instances from OR-Instruct/ORLM [Huang et al., 2025], OptMATH [Lu et al., 2025], and OptiBench-style resources [Yang et al., 2024], plus about 3.8k expert-curated instances. Optimization experts collected the curated problems from publicly available operations-research articles and websites. DeepSeek-V3.2 assisted with rewriting and diversifying descriptions, followed by expert formulation checks, manual curation, and solver verification. All retained training instances have valid reference objective values. These counts describe the problem pool; multi-turn trajectories are generated online during RL.

Table 8: Training problem pools and documented preparation steps. Counts refer to underlying problems, not online RL trajectories.

Problem pool	Approx. count	Sources and preparation
Cleaned public data	10k	OR-Instruct/ORLM, OptMATH, and OptiBench-style resources; cleaning and reference-objective verification.
Expert-curated data	3.8k	Public OR articles and websites; LLM-assisted description rewriting, expert formulation checks, filtering, and solver verification.

Evaluation data. For NL4Opt, MAMO, NLP4LP, IndustryOR, ComplexOR, and Resocratic, we use cleaned public benchmark resources associated with the LLM4OR survey [Xiao et al., 2025]. OptMATH-Bench is the official 166-instance ICML-2025 test release [Lu et al., 2025]. Its release file is `benchmark/OptMATH_Bench.json` in `https://github.com/optsuite/OptMATH`. Table 9 lists the eight evaluation sets and their JSONL copies in the PEARL repository's `benchmark/` directory. Macro accuracy averages their scores; micro accuracy weights them by the instance counts.

Data ablation. Table 11 compares public-only, curated-only, and combined-data training with Qwen3-4B. Under the same PEARL training and evaluation protocol, adding the curated data

Table 9: Benchmark instance counts used in the evaluation.

Benchmark	Instances	Released PEARL file
NL4Opt	214	`nl4opt.jsonl`
MAMO EasyLP	545	`mamo_easy_lp_clean.jsonl`
MAMO ComplexLP	111	`mamo_complex.jsonl`
NLP4LP	178	`nl4lp.jsonl`
OptMATH-Bench	166	`OptMATH_Bench.jsonl`
IndustryOR	42	`IndustryOR.jsonl`
ComplexOR	18	`complexor.jsonl`
Resocratic	403	`ReSocratic.jsonl`
Total	1,677	

improves macro- and micro-averaged Pass@1 by 6.28 and 1.43 percentage points, respectively, relative to public-only training.

H Additional Experiments and Protocols

This appendix provides additional comparisons and analyses. Appendix H.1 compares PEARL with SIRL using the same backbone and training data, and also evaluates the released SIRL checkpoint with Gurobi. Appendix H.2 reports training-data ablations and generated-token usage. Appendix H.3 tests Qwen3-4B in the LLMOPT inference workflow without LLMOPT-specific fine-tuning and discusses comparisons with other workflow-based systems.

H.1 SIRL retraining on PEARL's training data

To test whether additional training data alone explain PEARL's gains, we retrain SIRL on PEARL's 13.8k training instances using SIRL's released training pipeline. Both methods start from Qwen2.5-7B-Instruct and generate Pyomo code. For SIRL, we convert the data into the required problem–answer format without adding instances. We evaluate both models on the same eight benchmarks, with the same evaluation prompt, objective-value check, and tolerance.

Table 10: Qwen2.5-7B comparisons (Pass@1, %). Official SIRL uses its native Gurobi API and original training data; retrained SIRL and PEARL use Pyomo and the same 13.8k problems. Evaluation shares benchmark instances, validator, tolerance, and a common prompt template with interface-specific instructions. Results are point estimates, reported separately from Table 1.

Benchmark	n	PEARL Pyomo	Official SIRL Gurobi	Retrained SIRL Pyomo
NL4Opt	214	90.19	87.85	84.51
MAMO-E	545	94.13	96.15	94.50
MAMO-C	111	87.39	79.28	44.14
NLP4LP	178	91.01	94.38	89.33
OptMATH-Bench	166	3.61	27.71	1.20
IndustryOR	42	57.14	35.71	42.86
ComplexOR	18	44.44	27.78	27.78
Resocratic	403	74.94	85.86	73.45
Macro	–	67.86	66.84	57.22
Micro	1,677	77.82	82.29	73.03

Table 10 shows that PEARL reaches 67.86% macro-averaged and 77.82% micro-averaged Pass@1, compared with 57.22% and 73.03% for retrained SIRL. These are gains of 10.64 and 4.79 percentage points, respectively, with PEARL performing better on seven of eight benchmarks. Since both models use the same backbone and training data, the result supports a benefit from PEARL's training approach beyond access to additional data.

The two methods use the same training instances but learn differently: SIRL assigns terminal rewards to complete responses, whereas PEARL trains on multi-turn sequences of tool calls, feedback, and

revisions. The comparison matches the backbone, training data, and modeling interface, but does not match training compute or hyperparameter-tuning effort.

Table 10 also includes the released SIRL checkpoint evaluated with Gurobi. We retain our evaluation-prompt template rather than SIRL's original prompt. The released checkpoint achieves 66.84% macro-averaged and 82.29% micro-averaged Pass@1. It was trained on a different dataset, which was not fully available to us, so it is a separate comparison rather than the data-matched baseline above. Table 1 reports its results with Pyomo instead of Gurobi.

Retrained SIRL performs worse than the released checkpoint, but the two use different training data and modeling interfaces. Training budget, checkpoint selection, and hyperparameter tuning may also contribute to the difference. Further controlled experiments are needed to separate these effects.

H.2 Data ablation and generated-token usage

Training-data ablation. We train PEARL with Qwen3-4B on three datasets: the cleaned public 10k instances, the curated 3.8k instances, and their combination. The data sources are described in Appendix G. We keep the training and evaluation settings fixed. Table 11 shows that adding the curated data to the public set raises macro-averaged Pass@1 from 64.12% to 70.40% and micro-averaged Pass@1 from 78.95% to 80.38%. The combined set gives the best results, showing that the curated instances are useful additions to the public data.

Generated-token usage. In a separate experiment, we evaluate PEARL, DeepSeek-V3.2, and DeepSeek-V3.2-Think on the same instances. All methods use the same solver, code parser, and objective-value check, but follow their respective inference procedures. PEARL averages 9,868.75 generated tokens per instance, compared with 1,465.51 for DeepSeek-V3.2 and 2,323.56 for DeepSeek-V3.2-Think (Table 12). For PEARL, the count includes all model responses across the interaction, not just the final answer.

These counts measure how much text the models generate, not how much they cost to run. PEARL's 4B backbone supports local deployment and potentially lower per-token serving costs [Abdin et al., 2024, Belcak et al., 2025], which can help offset the additional generation required for multi-turn modeling. Total cost also depends on hardware, batching, and the time spent constructing and solving the generated optimization models [Li et al., 2026b]. We do not measure end-to-end latency or monetary cost in this experiment. The additional experiments in this appendix are reported as point estimates, without standard deviations.

Table 11: Effect of training data on PEARL-Qwen3-4B. Training and evaluation settings are unchanged across rows. Scores are macro- and micro-averaged Pass@1 (%). The combined-data row matches the direct-RL results in Table 6.

Training problem pool	Macro	Micro
Cleaned public 10k	64.12	78.95
Curated 3.8k	67.61	79.23
Combined 13.8k	70.40	80.38

Table 12: Mean generated tokens, execution success, and Pass@1 on the same evaluation instances. All methods use the same solver, code parser, and objective-value check, but retain their own inference procedures. Token counts cover all model responses for each instance, including intermediate turns, and are averaged across instances.

Method	Mean tokens	Execution success (%)	Pass@1 (%)
PEARL-Qwen3-4B	9,868.75	90.88	79.43
DeepSeek-V3.2	1,465.51	91.29	80.68
DeepSeek-V3.2-Think	2,323.56	91.06	79.90

H.3 LLMOPT and other workflow-based methods

We run Qwen3-4B within LLMOPT's inference workflow [Jiang et al., 2025], without LLMOPT-specific fine-tuning. The model reaches 5.30% macro-averaged and 6.14% micro-averaged Pass@1 (Table 13). Using the workflow alone therefore does not give strong results with this backbone. These scores are not an evaluation of the released LLMOPT-Qwen2.5-14B model, which has undergone task-specific fine-tuning.

Table 13: Pass@1 (%) for Qwen3-4B using LLMOPT's inference workflow without LLMOPT-specific fine-tuning. These scores do not represent the released LLMOPT-Qwen2.5-14B model.

Benchmark	Pass@1 (%)
NL4Opt	4.21
MAMO-E	8.62
MAMO-C	7.21
NLP4LP	5.62
OptMATH-Bench	0.00
IndustryOR	4.76
ComplexOR	5.56
Resocratic	6.45
Macro	5.30
Micro	6.14

Other released workflows require inputs beyond a natural-language description. OptiMUS [Ahmaditeshnizi et al., 2024] expects preprocessed target and data files, together with a Gurobi executor for linear and mixed-integer linear programs. Chain-of-Experts [Xiao et al., 2024] requires a function signature and input/output examples for each instance. Applying these releases to our benchmarks would require additional code to prepare those inputs; the results would depend in part on how this conversion is implemented.

OptiMind [Chen et al., 2025b] also uses multi-turn correction, but with a different backbone and training dataset. Its published scores therefore do not provide a comparison in which those factors are held fixed.

To test the benefit of PEARL training without changing the backbone or tool access, Appendix D.4 (Table 7) compares PEARL with the base Qwen3-4B model using the same tools and turn budget. We do not report equivalent tool-enabled comparisons for the larger DeepSeek and Qwen models, or a comparison with a general-purpose coding agent. Our results therefore do not establish that optimization-specific training is required for every agent workflow.

I Limitations and Broader Impact

Our evaluation provides evidence for tool-integrated agentic optimization modeling, but it has several limitations. First, PEARL operates under a bounded turn budget and fixed observation limits. This makes training and evaluation tractable, but it may under-estimate problems that require long debugging chains, richer data inspection, or interaction with multiple domain-specific tools. Multi-turn inference also incurs additional serving cost relative to one-shot generation, so future systems should learn when interaction is worth the cost and when a direct solution is sufficient.

Second, the benchmark suite covers linear, integer, mixed-integer, nonlinear, and second-order cone programs, but it still under-represents stochastic, robust, online, and highly domain-specific enterprise optimization problems. Real deployments may involve longer documents, richer data interfaces, domain-specific modeling languages, interactive human oversight, or larger industrial instances, motivating broader evaluation on industrial-scale benchmarks [Li et al., 2026c].

Third, solver execution and objective-value checks are strong but partial validation signals. They make evaluation auditable and practically meaningful, but they may fail to detect ambiguous specifications or formulations that satisfy the tested instance while violating broader requirements. In particular, an under- or over-constrained model may attain the reference optimum and receive the objective-agreement reward despite being semantically incorrect. We do not quantify this false-positive rate.

Constraint-satisfaction problems without an optimization objective are outside the current evaluation and would require a verifier for the predicted assignment. This limitation is common in optimization-modeling evaluation: objective agreement, executability, formulation matching, and solver-checked outputs are practical proxies for correctness, but none can fully score all mathematically equivalent formulations or latent structural defects. Future work should therefore develop richer validators, uncertainty-aware evaluation, and training procedures for broader classes of real-world optimization workflows.

The potential positive impact of this line of research is substantial. Reliable interactive optimization modeling could lower the expertise barrier that currently limits the use of mathematical optimization in domains such as logistics, manufacturing, planning, energy systems, and resource allocation. By emphasizing solver execution and validation feedback, PEARL may also reduce modeling errors and encourage a more transparent style of LLM-based decision-support generation centered on verifiable artifacts rather than surface-level text.

At the same time, this capability carries meaningful risks. Optimization models are often used in high-stakes settings, and flawed formulations may yield infeasible, unsafe, unfair, or economically harmful decisions while still appearing superficially plausible. A system that generates convincing but incorrect solver code could create a false sense of reliability, especially for non-expert users. PEARL should therefore be understood as a sandboxed modeling and verification system, not as evidence that current agents are ready for autonomous deployment in decision-critical environments. All outputs require human interpretation and validation before real-world use. We hope this work helps promote stronger solver-grounded verification, better structural interpretability, clearer human oversight, and safer standards for optimization-oriented AI systems.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract states the main contribution as an interactive PEARL system for optimization modeling that uses Python execution, solver feedback, and multi-turn revision; Sec. 1 gives the full Agentic RL framing and contribution statement. The empirical claims are tied to verified solve rates in Sec. 4.2.1 and Table 1, including the PEARL-Qwen3-4B comparison against DeepSeek-V3.2-685B, while scope limitations are discussed in Appendix I.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Appendix I includes a dedicated Limitations and Broader Impact section discussing the bounded interaction horizon, additional inference cost, data and label limitations, benchmark coverage, and limits of solver-based validation; Sec. 5 also points readers to this discussion.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best

judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: The paper is empirical and methodological: Sec. 3 presents the agentic modeling formulation and training objective, while Appendix D.1.1 and Appendix D.2.1 provide informal analysis of design choices rather than formal theorem statements or proofs.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Sec. 3 specifies the environment, reward, and training objective, and Sec. 4 specifies the benchmark task and validator. Appendix B, Appendix F, and Appendix H document the main configuration, prompts, and additional controls. The public repository links a trained PEARL-Qwen3-4B checkpoint for independent evaluation; complete retraining additionally requires the training corpus and run-specific configurations.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.

(c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: Sec. 4 links the public code, datasets, and a trained checkpoint at `https://github.com/AuroraLHL/PEARL`. The release includes the full approximately 13.8k-instance training corpus, provenance information, and instructions for data preparation, training, and evaluation. Appendix G documents the data sources and preparation, while Appendix B and Appendix F provide experimental settings and prompt templates.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Sec. 4 specifies the benchmark suite, compared methods, evaluation metric, repeated pass@1 protocol, and solver-based correctness criterion. Appendix B, Appendix B.2, Appendix B.3, Table 3, and Table 4 provide the GRPO training setup, inference configuration, tool-use limits, and key hyperparameters.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Sec. 4 specifies three pass@1 evaluation runs by default, and Table 1 reports benchmark-wise mean $\pm$ standard deviation. The additional controls in Appendix H are explicitly reported as point estimates without variance estimates. We do not claim significance tests for these differences.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
   - If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Appendix B and Table 3 report the PEARL training hardware, GPU count, Ray/SGLang rollout setup, tensor/context parallelism, token and memory-related settings, rollout batch sizes, tool concurrency, Python timeout and memory limits, and checkpointing. Table 4 reports the vLLM inference configuration, and Appendix D.2.2 states the larger eight-GPU setting used for the trajectory-filtering ablation.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

Justification: Sec. 3 describes PEARL as a sandboxed optimization-modeling agent using execution and solver feedback, Sec. 4 describes public/curated optimization tasks and solver-grounded evaluation, and Appendix I reports limitations, broader impact, and the fact that PEARL is not evidence for autonomous deployment in decision-critical environments.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: Appendix I discusses potential benefits for optimization modeling and decision-support workflows, as well as risks from flawed formulations or misleading solver code; it also emphasizes that outputs require human validation and should not be treated as automated real-world decision deployment.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: The submission does not release a high-risk pretrained foundation model, image generator, or scraped dataset. Sec. 3 and Appendix B.1 describe the studied agent as operating through a sandboxed solver/code-execution workflow, and Appendix I frames the system as a modeling and verification resource rather than an autonomous deployment tool.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.

- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: Sec. 2 and Sec. 4 credit the existing optimization-modeling benchmarks, Qwen/DeepSeek/GPT-family baselines, prior systems, solvers, and serving/training frameworks used in the experiments. Appendix B names the vLLM, Ray, SGLang/slime-based rollout and training setup, and the public repository linked in Sec. 4 documents the released PEARL code and data assets with attribution and license information.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: Sec. 4 links the public repository, which documents the released PEARL code assets, repository structure, core rollout/reward components, benchmark/example data format, and setup/evaluation entry points. Appendix B, Appendix A.2, and Appendix F further document the experimental configuration, reward implementation, and prompts associated with these assets.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The work described in Sec. 3 and Sec. 4 concerns algorithmic training and evaluation on optimization-modeling instances; it does not involve crowdsourcing experiments or research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The method and experiments in Sec. 3 and Sec. 4 do not involve human subjects, so IRB approval or equivalent human-subjects review is not applicable.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: Sec. 3 describes the LLM policy and tool-integrated RL; Sec. 4 reports LLM-based evaluation. Appendix G also discloses LLM-assisted data construction, and Appendix B and Appendix F document model and prompt settings.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.