Escaping the Sisyphus Dilemma: Experience Replay for Robust Text-to-Optimization Modeling

Wantong Xie[1]*, **Yinghao Chen**[2]*, **Yi-Xiang Hu**[2], **Feng Wu**[2], **Jieyang Xu**[2]
Sijia Zhang[2]† and **Xiang-Yang Li**[2]
[1]Institute of Advanced Technology, University of Science and Technology of China
[2]School of Computer Science and Technology, University of Science and Technology of China
{wantongxie,chenyinghao,yixianghu,jieyangxu,sxzsj}@mail.ustc.edu.cn
{wufeng02,xiangyangli}@ustc.edu.cn

Abstract

Large Language Models have shown promise in translating natural language into executable optimization models, yet they often suffer from the Sisyphus Dilemma: a memoryless cycle where identical errors are repeated across structurally similar problems. Existing retrieval-augmented strategies primarily fetch static problem-model pairs as few-shot demonstrators, failing to capture the dynamic reasoning required to resolve execution failures. To bridge this gap, we propose **EOM**, a framework that implements Experience Replay to transform transient rectification steps into persistent knowledge. EOM distills interaction histories into Causal Correction Mappings, indexing both diagnostic insights and prohibitive traps. By utilizing a structure-aware retrieval mechanism that aligns semantic intent with abstract syntax trees and solver tracebacks, the system enables models to recall specific correction strategies for isomorphic errors. Extensive experiments across seven benchmarks demonstrate that EOM improves modeling accuracy by 8.45% on complex tasks while reducing token consumption by 28.65% and interaction turns by 25.82%, validating the efficiency of a "Rectify Once, Solve Many" paradigm.

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable potential in democratizing Operations Research (OR) by automatically translating natural language descriptions into mathematical optimization models (Wasserkrug et al., 2025; Xie et al., 2025). Unlike general code generation tasks, optimization modeling requires the construction of strict executable artifacts that must align with the rigorous protocols of external solvers (e.g., Gurobi (Gurobi Optimization, LLC, 2025)). A minor syntactic slip or a subtle logical misalignment often renders the entire model infeasible. Consequently, the prevailing paradigm relies heavily on iterative refinement where the model engages in multi-turn dialogues with a compiler or solver to diagnose and fix errors (Chen et al., 2023).

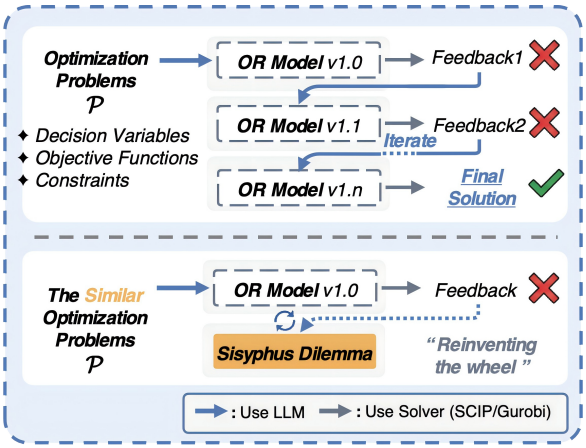


Figure 1: **The Sisyphus Dilemma.** The model relies on a memoryless iterative loop to resolve errors. Lacking experience persistence, it fails to transfer insights to similar problems, forcing it to "reinvent the wheel" by repeating futile rectification steps.

However, this iterative process suffers from a fundamental inefficiency which we term the **Sisyphus Dilemma** (Camus, 2013). As illustrated in Figure 1, current methods operate as memoryless solvers. They may successfully resolve a complex variable definition error after multiple rounds of trial-and-error, yet they fail to retain this diagnostic expertise. When confronted with a structurally identical problem in a subsequent session, the model resets to its initial state and repeats the same futile trajectory of mistakes (Zhao et al., 2024). This phenomenon not only inflates computational costs but also hinders the model from evolving its expertise over time (Huang et al., 2024a).

Existing solutions primarily address this chal-

*These authors contributed equally to this work.
†Corresponding author.

Findings of the Association for Computational Linguistics: ACL 2026, pages 14100–14116

lenge by retrieving static knowledge (Asai et al., 2024). Retrieval-Augmented Generation (RAG) frameworks typically fetch similar problem-model pairs to serve as few-shot demonstrators (AhmadiTeshnizi et al., 2024a). While effective for syntactic reference, standard retrieval fails to capture the state transition logic required. Retrieving a correct final solution does not teach the model how to transition from an erroneous state to a valid one, nor does it warn against latent pitfalls that lead to dead ends (Lightman et al., 2023).

To bridge this gap, we propose a framework called **EOM** (**E**xperience for **O**ptimization **M**odeling), which implements Experience Replay for executable model generation. Inspired by how humans extract experience from errors, EOM transforms transient rectification steps into persistent knowledge. We introduce the concept of Causal Correction Mappings, where valid rectification trajectories are distilled into diagnostic insights and failed attempts are summarized as prohibitive traps. By indexing these experiences using a structure-aware key that combines semantic intent, Abstract Syntax Trees (ASTs), and solver tracebacks. This allows the model to identify isomorphic error patterns across different problem narratives, shifting the paradigm from "guess-and-check" to "recall-and-refine".

Our contributions are summarized as follows:

- **Experience Replay for Optimization Modeling.** We propose EOM, which integrates a memory mechanism into the optimization modeling lifecycle. This enables the transition from memoryless generation to a continuous learning system that accumulates diagnostic expertise from every interaction.

- **Structure-Aware Retrieval.** We design a multi-view retrieval system that utilizes ASTs and execution tracebacks. This allows the accurate location of historical insights for specific logical or syntactic errors across different problem narratives.

- **Accuracy and Efficiency Gains.** Extensive experiments show that EOM improves modeling accuracy by 8.45% on complex benchmarks while reducing token consumption by 28.65% and interaction turns by 25.82%, validating the principle of "Rectify Once, Solve Many".

2 Problem Definition

We focus on the Text-to-Optimization Modeling task, where the objective is to translate a natural language problem description $\mathcal{P}$ into an executable optimization model $\mathcal{M}$ (Pan et al., 2025). This process is governed by an external solver $\mathcal{S}$ which provides deterministic feedback. Detailed mathematical formulations of the target Mixed-Integer Linear Programming (MILP) problems and variable definitions are provided in Appendix A.1.

2.1 Iterative Modeling Process

The modeling process entails a sequential interaction between an LLM π_θ and an external solver $\mathcal{S}$. At the t-th turn, π_θ generates an optimization model $\mathcal{M}_t$ conditioned on the problem description $\mathcal{P}$ and the accumulated interaction history $\mathcal{H}_{t-1}$. We define the history as the sequence of prior model attempts paired with their corresponding solver feedback:

$$\mathcal{H}_{t-1} = \{(\mathcal{M}_i, \mathcal{F}_i)\}_{i=0}^{t-1} \tag{1}$$

Here, $\mathcal{F}_i$ denotes the deterministic feedback (e.g., error tracebacks or infeasibility reports) returned by $\mathcal{S}$. If $\mathcal{F}_t$ signifies a successful execution, the process terminates. Otherwise, $\mathcal{F}_t$ acts as a correction signal, and π_θ attempts to refine the model based solely on the local context $(\mathcal{P}, \mathcal{H}_{t-1})$:

$$\mathcal{M}_t \sim \pi_\theta(\mathcal{M}_t \mid \mathcal{P}, \mathcal{H}_{t-1}) \tag{2}$$

2.2 Experience-Augmented Objective

The core challenge, which we term the Sisyphus Dilemma, is that π_θ treats each problem instance as an isolated event. This concept draws from the philosophical metaphor of Sisyphus, who was condemned to ceaselessly roll a boulder uphill only to watch it fall back, symbolizing the futility of repetitive labor without retention. It lacks a mechanism to recall successful rectification trajectories from prior sessions, leading to repetitive error patterns across structurally similar problems.

To address this, we introduce an experience set $\mathcal{E}$ containing historical rectification insights. Our goal is to shift the generation probability to be conditioned on retrieved experiences that are causally relevant to the current error state. The goal is to maximize the likelihood of generating the correct

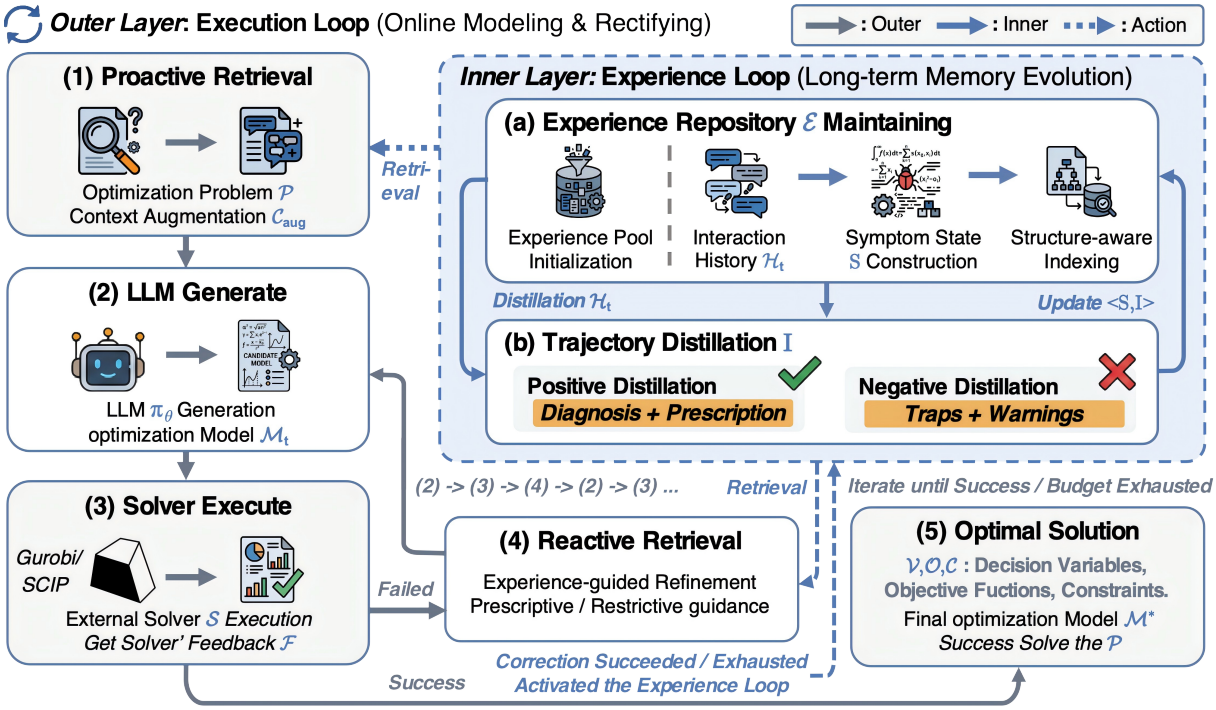


Figure 2: **Overview of the EOM framework.** The architecture couples two loops: an *Outer Execution Loop* for online modeling, where the LLM π_θ iteratively refines candidates $\mathcal{M}_t$ using solver feedback and retrieved context; and an *Inner Experience Loop* for long-term memory evolution. Here, interaction histories $\mathcal{H}_t$ are distilled into causal correction mappings, encoding both successful diagnoses and failure traps, which are indexed to guide future optimization tasks.

Figure 3: **EOM** LLM π_θ $\mathcal{M}_t$ $\mathcal{H}_t$

model $\mathcal{M}^*$ by leveraging a subset of relevant experiences $\mathbf{e} \subset \mathcal{E}$:

$$\max_{\theta} \sum_{(\mathcal{P},\mathcal{M}^*)\in\mathcal{D}} \log \pi_\theta(\mathcal{M}^* \mid \mathcal{P}, \mathcal{H}, \mathbf{e}) \quad (3)$$

where $\mathbf{e}$ bridges the gap between the current error state and the correct solution, effectively bypassing redundant trial-and-error iterations.

3 The EOM Framework

3.1 Overview

As shown in Figure 3, the proposed EOM framework transforms the transient trial-and-error process into a persistent learning cycle. The architecture operates through two coupled feedback loops: 1) an outer Execution Loop responsible for immediate code refinement, and 2) an inner Experience Loop responsible for long-term knowledge accumulation.

The workflow commences with the **Structure-Aware Retrieval** phase. Given a new problem description or a runtime error from the solver, the system queries the vectorized experience pool. It seeks historical trajectories that share similar semantic contexts or isomorphic error structures. These retrieved insights are then injected into the LLM via **Context Augmentation**. Then π_θ utilizes this borrowed wisdom to generate or refine the optimization model $\mathcal{M}$, bypassing previously known pitfalls.

Upon the conclusion of a modeling session, the **Experience Acquisition** mechanism activates. Regardless of whether the final outcome was a success or a failure, the system employs a distillation process to extract the causal logic behind the result. Successful rectification sessions yield prescriptive fixes, while failed attempts yield prohibitive warnings. These distilled Symptom-Insight pairs are indexed back into the experience pool. This closes the outer loop, ensuring that the system continuously evolves and adheres to the principle of "Rectify Once, Solve Many".

3.2 Experience Representation: Causal Correction Mapping

To bridge the gap between erroneous modeling attempts and valid solutions, we formalize an ex-

perience unit not as a static data record, but as a dynamic Causal Correction Mapping. Each unit $e \in \mathcal{E}$ encapsulates the trajectory from a specific error context to a verified correction or a recognized dead-end. Formally, we define the experience unit as a tuple:

$$e = \langle \mathbf{S}_{\text{symptom}}, \mathbf{I}_{\text{insight}} \rangle \tag{4}$$

Symptom State ($\mathbf{S}_{\text{symptom}}$). Effective retrieval requires precise error localization. Relying solely on natural language similarity is insufficient for code generation tasks where semantic intent is correct, but implementation details fail. Therefore, we construct a hybrid symptom state $\mathbf{S}_{\text{symptom}}$ comprising three distinct feature vectors:

$$\mathbf{S}_{\text{symptom}} = \phi_{\text{sem}}(\mathcal{P}) \oplus \phi_{\text{ast}}(\mathcal{C}_{\text{err}}) \oplus \phi_{\text{trace}}(\mathcal{F}_{\text{err}}) \tag{5}$$

Here, $\phi_{\text{sem}}(\mathcal{P})$ represents the semantic embedding of the problem description. $\phi_{\text{ast}}(\mathcal{C}_{\text{err}})$ encodes the structural signature of the erroneous code segment using a linearized AST. This structural encoding ensures the representation is invariant to variable naming differences while remaining sensitive to logic flow. Finally, $\phi_{\text{trace}}(\mathcal{F}_{\text{err}})$ embeds the deterministic solver traceback, capturing the precise nature of the violation, such as dimensionality mismatch or infeasible constraints.

Corrective Insight ($\mathbf{I}_{\text{insight}}$). The insight component provides the actionable knowledge required to resolve the symptom. Unlike standard solutions that only provide the final code, $\mathbf{I}_{\text{insight}}$ includes the reasoning process. The content of the insight depends on the outcome of the rectification session:

1. Success Trajectories. When a valid model is derived, the insight contains a Diagnosis which explains the root cause of the error and a Prescription which provides the corrected code snippet. This pair teaches the model not just what to write, but why the previous attempt was invalid.

2. Failure Trajectories. For sessions that exhaust the maximum interaction depth without success, we extract Negative Constraints. The insight comprises Traps, summarizing strategies that were attempted but failed, and Warnings, which are explicit instructions to avoid specific syntax or logic patterns. These negative insights effectively prune the search space for future iterations.

3.3 Experience Acquisition via Trajectory Distillation

Raw interaction histories typically contain redundant conversation turns, generic chit-chat, and intermediate incorrect attempts. Direct retrieval of such noisy data yields a suboptimal context. To address this, we propose Trajectory Distillation, a process that employs an LLM as a synthesizer to compress the raw history $\mathcal{H}$ into the structured corrective insight $\mathbf{I}_{\text{insight}}$. This distillation operates through two distinct pathways based on the execution outcome.

Positive Distillation: Retrospective Analysis. When a modeling session successfully produces a valid solution $\mathcal{M}^*$, the system performs a retrospective analysis. The distiller model reviews the transition from the first erroneous attempt $\mathcal{M}_{\text{err}}$ to the final correct code. It identifies the critical pivot point where the logic was fixed. The distillation objective is to generate a Diagnosis that articulates the discrepancy between the user's intent and the solver's requirements, followed by a Prescription. Crucially, the prescription is not the entire code block but the specific logical patch or corrected constraint formulation. This minimizes token consumption during retrieval and focuses the model's attention on the delta required for correction.

Negative Distillation: Dead-End Pruning. Trajectories that reach the maximum interaction depth without a valid solution are equally valuable as they delimit the boundaries of the solution space. For these failed sessions, the distillation process functions as a post-mortem analysis. The model summarizes the sequence of failed strategies into Traps, effectively labeling paths that lead to infeasibility or syntax loops. Additionally, it synthesizes Warnings, which serve as explicit negative constraints. By storing these negative insights, the system prevents future iterations from exploring known dead ends, thereby pruning the search space efficiently.

Automated Pipeline. Formally, let $\mathcal{D}_{\text{pos}}$ and $\mathcal{D}_{\text{neg}}$ denote the prompt-based distillation functions for success and failure cases, respectively. The acquisition process is automated as follows:

$$\mathbf{I}_{\text{insight}} = \begin{cases} \mathcal{D}_{\text{pos}}(\mathcal{H}, \mathcal{M}^*) & \text{, if } \mathcal{F}_{\text{final}} = \text{Success} \\ \mathcal{D}_{\text{neg}}(\mathcal{H}) & \text{, otherwise} \end{cases} \tag{6}$$

The extracted insight is then paired with the corresponding symptom state $\mathbf{S}_{\text{symptom}}$ and indexed into

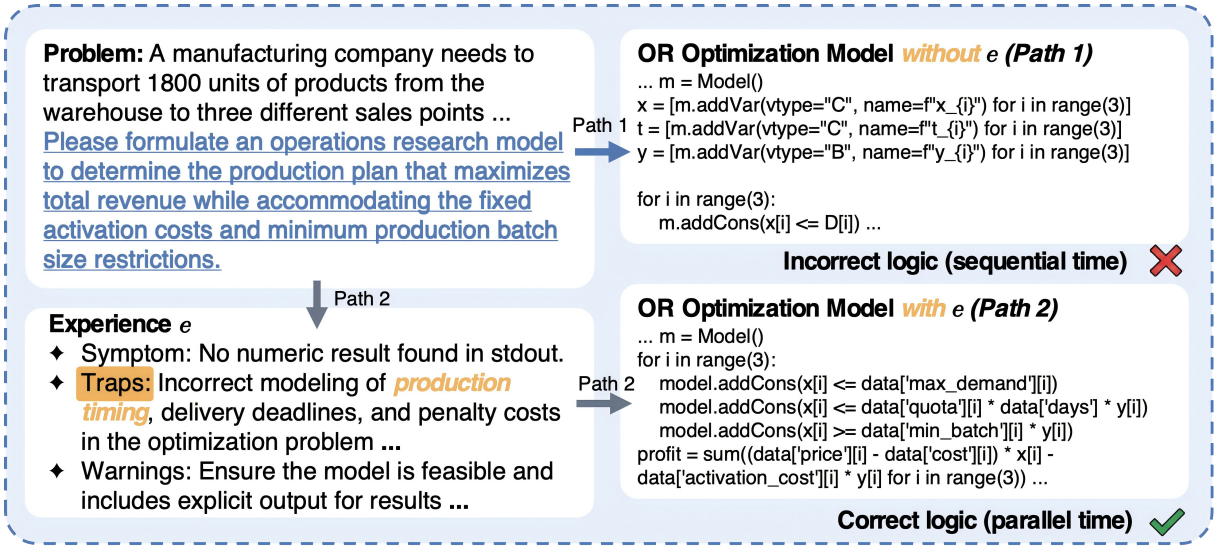


Figure 4: **Qualitative case study comparing modeling trajectories.** Path 1 illustrates a baseline attempt that fails due to a logical misalignment in time constraints. In contrast, Path 2 demonstrates our EOM framework, which retrieves an experience containing specific Traps regarding production timing. Guided by this insight, the model rectifies the formulation by correctly incorporating the time factor, leading to a valid solution.

the experience pool, completing the experience acquisition loop.

3.4 Structure-Aware Indexing and Retrieval

Once experiences are distilled into the experience pool $\mathcal{E}$, the retrieval mechanism serves as the bridge between past insights and current problems. To maximize the utility of historical data, we employ a dual-phase retrieval strategy: Proactive Retrieval prior to the initial modeling attempt, and Reactive Retrieval upon encountering execution errors.

Adaptive Query Formulation. The query vector $\mathbf{Q}$ adapts dynamically to the availability of context. In the Proactive Phase, the model has not yet generated code. The query relies solely on the problem description to identify similar tasks and their associated traps: $\mathbf{Q} = \langle \phi_{\text{sem}}(\mathcal{P}), \mathbf{0}, \mathbf{0} \rangle$. In the Reactive Phase, triggered by a solver error, we construct a full-state query to locate specific rectification solutions:

$$\mathbf{Q} = \langle \phi_{\text{sem}}(\mathcal{P}), \phi_{\text{ast}}(\mathcal{C}_{\text{err}}), \phi_{\text{trace}}(\mathcal{F}_{\text{err}}) \rangle \quad (7)$$

Multi-View Relevance Scoring. To quantify experience relevance, we employ a parameter-free scoring mechanism that directly sums the cosine similarities across semantic, structural, and execution views. This approach ensures balanced attention to both high-level intent and low-level rectification logic without heuristic tuning. The additive metric naturally adapts to varying context availability by implicitly assigning zero weight to missing features, such as absent tracebacks during initial generation. To minimize context pollution, we strictly retrieve the single nearest neighbor for augmentation.

3.5 Experience-Guided Refinement

The final component of the EOM framework is the utilization of the retrieved experience e^* to steer the current generation. We employ a Context Augmentation strategy, where the distilled insights are formatted into a structured instructional block and injected into the LLM's inference context.

Insight Integration. The augmented input context $\mathcal{C}_{\text{aug}}$ is constructed by synthesizing the problem description $\mathcal{P}$, the current interaction history $\mathcal{H}$, and the retrieved insight $\mathbf{I}_{\text{insight}}$. To ensure the model prioritizes this external knowledge over its internal biases, we wrap the insight in a dedicated prompt section:

$$\mathcal{C}_{\text{aug}} = \mathcal{P} \oplus \mathcal{H} \oplus \textsc{Format}(\mathbf{I}_{\text{insight}}) \quad (8)$$

where $\oplus$ denotes concatenation and $\textsc{Format}(\cdot)$ is a template function that translates the structural fields of the experience unit into natural language directives.

Dual-Mode Guidance. The refinement mechanism operates differently depending on the nature of the retrieved experience:

- **Prescriptive Guidance:** If e^* contains a Diagnosis and Prescription, the system acts as

Method	Level 1		Level 2			Level 3	
	NL4Opt	MAMO Easy	OptiBench	ComplexOR	ComplexLP	OptMATH	IndustryOR
Standard	70.5	84.3	52.4	52.6	39.8	16.2	29.0
Reflexion	85.7	85.9	63.2	61.1	46.9	34.3	32.0
CoT (2022)	74.0	82.9	53.1	52.6	40.7	21.1	35.0
CoE (2023)	79.2	85.9	55.2	63.2	43.1	24.1	33.0
OptiMUS (2024b)	80.6	87.1	58.8	79.0	45.2	32.5	36.0
MCTS (2025)	89.6	88.0	67.9	79.0	51.6	38.6	46.0
OptiTree (2025a)	98.3	96.9	74.7	**84.2**	**81.5**	52.4	54.0
EOM (Ours)	**99.6**	96.3	83.1	83.3	73.9	69.3	53.0

Table 1: **Performance comparison of prompt-based optimization methods.** We evaluate accuracy using DeepSeek-V3 across three difficulty levels categorized by problem complexity. The notation EOM-k denotes our framework with a maximum interaction depth of k. Our approach demonstrates robust scalability, consistently outperforming search-based baselines like OptiTree, particularly on the most challenging Level 3 benchmarks. The best results are highlighted in **bold** and the second best are underlined.

a "teacher". It explicitly highlights the root cause of the potential error and provides the verified code logic. This encourages the model to mimic the successful correction pattern, effectively short-circuiting the trial-and-error process.

- **Restrictive Guidance:** If e^* contains Traps and Warnings, the system functions as a "guardrail". The insight serves as a set of negative constraints, strictly prohibiting specific modeling strategies or API usages that previously led to dead ends. This effectively prunes invalid branches from the search space before generation begins.

Refined Generation. Finally, the model generates the refined solution $\mathcal{M}_{\text{new}}$ conditioned on this experience-enriched context:

$$\mathcal{M}_{\text{new}} \sim \pi_\theta(\mathcal{M} \mid \mathcal{C}_{\text{aug}}) \tag{9}$$

By explicitly integrating past diagnostic expertise, the model transcends its immediate local context. It avoids repeating identical mistakes and navigates complex constraints with the borrowed foresight of its past self.

4 Experiments

4.1 Experimental Setup

Benchmarks. We evaluate the robustness of our framework across seven optimization modeling datasets, categorized into three difficulty levels based on problem length and logical complexity. **Level 1** comprises NL4Opt (Ramamonjison et al., 2023) and MAMO-Easy (Huang et al., 2024b), which focus on standard linear and integer programming tasks with concise descriptions. **Level 2** includes OptiBench, ComplexOR, and ComplexLP (Yang et al., 2024; Xiao et al., 2023). These datasets feature longer narratives and denser constraint sets requiring multi-step reasoning. **Level 3** consists of OptMATH (Lu et al., 2025) and IndustryOR (Huang et al., 2025), representing the most challenging scenarios with intricate variable dependencies and real-world industrial contexts.

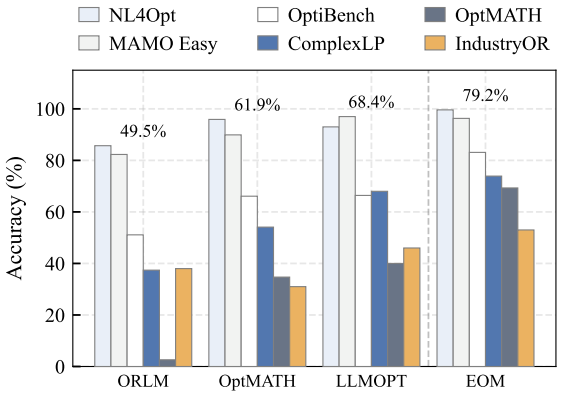


Figure 5: **Performance comparison of learning-based optimization methods.** We report accuracy results for our model against state-of-the-art supervised baselines across three difficulty levels.

Baselines. We compare EOM against a comprehensive set of baselines categorized into three paradigms: Learning-based Methods, Prompting Strategies, and RAG Frameworks. Please refer to Section 5 for a detailed discussion and categoriza-

Method	Level 1		Level 2			Level 3		Average
	NL4Opt	MAMO Easy	OptiBench	ComplexOR	ComplexLP	OptMATH	IndustryOR	
Average Token Savings Rate								*28.65% ↓*
w/o E	5874.7	9922.8	7203.6	9485.0	7240.8	14987.5	14147.3	9837.4
w/ E	3128.0	8655.8	6108.0	5189.0	5580.5	10725.7	9747.8	**7019.3**
Average Turn Savings Rate								*25.82% ↓*
w/o E	2.06	2.19	2.18	2.25	2.16	2.35	2.33	2.22
w/ E	1.56	1.96	1.86	1.25	1.86	1.70	1.83	**1.72**

Table 2: **Ablation study on efficiency: Token and turn consumption.** We analyze the average token usage and conversation turns specifically for samples requiring multiple iterations. The results demonstrate that our proposed method (w/ E.) significantly reduces overhead compared to the baseline (w/o E.), achieving substantial savings in both computational cost and interaction depth.

tion of these specific methods.

Metrics and Implementation. The primary evaluation metric is Accuracy. A generated model is considered correct only if it is syntactically valid and yields the ground-truth optimal objective value when executed by the solver. Unless otherwise stated, we employ DeepSeek-V3 (DeepSeek-AI, 2025) as the backbone LLM with a seed of 42 to ensure reproducibility. The external solver is SCIP (Maher et al., 2016). For the experience retrieval, we use a dense vector index with cosine similarity.

4.2 Main Results

We present a comparative analysis of EOM against the three baseline categories defined in the setup.

Comparison with Learning-based Methods. Table 5 illustrates the performance of EOM against supervised baselines. While fine-tuned models like LLMOPT achieve competitive results on simpler Level 1 tasks, their performance degrades significantly on complex benchmarks. For instance, ORLM and OptMATH struggle to generalize to the intricate logic of IndustryOR. In contrast, our inference-time method maintains robust performance across all difficulty levels. Notably, on the Level 3 OptMATH dataset, EOM achieves 69.3% accuracy, surpassing the best supervised baseline by a substantial margin. This demonstrates that retrieving relevant rectification experiences is more effective for logical generalization than fitting parameters on static training sets.

Comparison with Prompting Strategies. Table 1 reports the results using DeepSeek-V3 as the backbone. Standard prompting and CoT often fail to navigate the high-dimensional search space of complex optimization problems. While search-based agents like OptiTree show improvement by exploring multiple reasoning paths, they lack a mechanism to retain successful strategies. Our method effectively closes this gap. Even with a limited interaction budget, EOM outperforms the computation-heavy OptiTree on the majority of datasets. With 5 iterations, the performance gap widens further, validating our hypothesis that historical foresight reduces the need for extensive random exploration.

Comparison with RAG Frameworks. Table 5 contrasts EOM with OptiMUS-0.3, a representative RAG-based baseline. OptiMUS-0.3 employs a retrieval mechanism to fetch natural language problem descriptions paired with their corresponding optimization code from open-source datasets, utilizing them as few-shot demonstrators. While this approach provides syntactic reference, our method consistently outperforms it, achieving gains of 5.1% on NL4Opt and 6.4% on NLP4LP. This performance gap highlights a critical insight: providing static "gold" code samples is often insufficient for complex reasoning tasks. In contrast, EOM retrieves correction trajectories, explicitly teaching the model how to resolve conflicts and transition from an erroneous state to a valid solution, rather than merely imitating a final answer.

4.3 Qualitative Analysis

Figure 4 provides a concrete example of how retrieved experiences rectify logical reasoning. In this manufacturing optimization task, the baseline method constructs a syntactically valid but logically flawed constraint, neglecting the temporal dimension of production quotas. This "silent error"

leads to an infeasible plan without triggering immediate solver warnings. Conversely, our EOM framework retrieves a targeted experience containing a specific Trap regarding "incorrect modeling of production timing". Conditioned on this negative insight, the model explicitly incorporates the time factor into the constraint logic (multiplying quota by days). This correction demonstrates that historical Traps serve as effective cognitive guardrails, preventing the repetition of subtle domain-specific errors that standard prompting strategies overlook.

4.4 Ablation Studies

Beyond modeling accuracy, we evaluate the computational efficiency of the framework to assess its practical viability. We conduct an ablation study to quantify the impact of the experience replay mechanism on interaction overhead, with detailed statistical results and analysis presented in Appendix A.4. The experimental data demonstrates that integrating historical insights significantly accelerates convergence. Specifically, EOM reduces average token consumption by 28.65% and decreases interaction turns by 25.82%. These substantial reductions confirm that retrieved guidance effectively short-circuits repetitive trial-and-error loops and validates the efficiency gains of the proposed paradigm.

5 Related Work

LLMs for Optimization. The application of LLMs to OR has evolved from simple translation to complex workflows. Early approaches focused on direct prompting or Chain-of-Thought (CoT) strategies (Wei et al., 2022) to interpret natural language descriptions. Recent frameworks have integrated external solvers and multi-stage reasoning, such as Chain-of-Experts (CoE) (Xiao et al., 2023). To enhance reliability, agentic workflows like OptiMUS (AhmadiTeshnizi et al., 2024b) employ modular agents to formulate and solve problems, while search-based frameworks like MCTS (Astorga et al., 2025) and OptiTree (Liu et al., 2025a) utilize exploration strategies to navigate the solution space. Parallel to prompting, supervised approaches including ORLM (Huang et al., 2025), OptMATH (Lu et al., 2025), and LLMOPT (Jiang et al., 2025) seek to improve performance by fine-tuning models on domain-specific corpora. Unlike these approaches, our EOM framework aims to minimize exploration by leveraging historical diagnostic expertise.

Iterative Refinement. Iterative refinement is a standard paradigm in code generation. Models generate code, execute it, and use the compiler feedback or tracebacks to correct errors. This Self-Repair mechanism has been formalized in various frameworks such as Self-Refine and Reflexion (Madaan et al., 2023; Shinn et al., 2023). In the specific context of optimization, OptiTree and CoE utilize step-by-step verification to catch errors early. Despite their effectiveness within a single session, these methods suffer from a lack of persistence. The model solves a syntax error in one instance but fails to transfer this correction logic to a subsequent, structurally similar problem. Our work addresses this "amnesia" by converting transient rectification steps into persistent knowledge.

RAG and Experience Learning. RAG typically enhances LLMs by providing relevant documents or code snippets from a static knowledge base. In the OR domain, RAG is often used to retrieve similar problem-model pairs as few-shot demonstrators (AhmadiTeshnizi et al., 2024a). However, retrieving a "correct solution" is often insufficient for rectification subtle logical conflicts (Zhong et al., 2024). Recent advancements in experience replay, inspired by reinforcement learning, suggest storing interaction trajectories (Liu et al., 2025b). Our approach aligns with this trend but introduces a structure-aware retrieval mechanism tailored for OR. We distinguish between prescriptive insights that guide success and prohibitive warnings that prune the search space, a distinction rarely made in standard code-retrieval systems.

6 Conclusion

In this paper, we introduced the EOM framework to address the "Sisyphus Dilemma" in optimization modeling by transforming transient error tracebacks into persistent causal correction mappings. Through a structure-aware retrieval mechanism, our approach enables LLMs to recall and apply past diagnostic expertise to isomorphic bug patterns. Extensive experiments demonstrate that EOM not only outperforms state-of-the-art baselines on complex industrial benchmarks but also significantly reduces computational overhead by short-circuiting repetitive trial-and-error

loops. Ultimately, this work validates the "Rectify Once, Solve Many" principle, advocating for a paradigm shift towards continuous learning agents that evolve expertise through every execution failure.

Limitations

The efficacy of the EOM framework currently relies on explicit feedback signals from the external solver. Consequently, the system may struggle to address silent semantic errors where the generated model executes successfully but misinterprets the underlying constraints, yielding incorrect objective values without triggering a traceback. Furthermore, the framework is sensitive to the quality of the distilled experience pool. Inaccurate diagnoses or hallucinated prescriptions accumulated during the acquisition phase could potentially introduce noise into the retrieval mechanism, propagating misleading guidance to future modeling sessions. Future work will focus on developing automated verification metrics to filter low-fidelity experiences and extending the approach to nonlinear optimization domains.

Ethical Considerations

Our research contributes to energy-efficient AI by significantly reducing the computational overhead associated with iterative rectification in LLMs. However, the automated generation of optimization models for critical infrastructure, such as logistics networks and energy grids, warrants caution. An incorrectly formulated constraint that is syntactically valid could lead to suboptimal or hazardous decision-making in real-world operations. Therefore, we emphasize that human oversight remains essential prior to the deployment of any generated code. Additionally, the experience replay mechanism involves storing problem descriptions and execution traces. Implementers must establish strict data privacy protocols to ensure that sensitive or proprietary industrial information is not inadvertently retained or leaked through the shared experience pool.

Acknowledgments

The research is partially supported by Quantum Science and Technology-National Science and Technology Major Project (QNMP) 2021ZD0302900, China National Natural Science Foundation with No. 92567301, 62132018, 62231015, "Pioneer" and "Leading Goose" R&D Program of Zhejiang, 2023C01029 and 2023C01143, and Anhui Provincial Science and Technology Innovation Program 202523o09050019.

References

Ali AhmadiTeshnizi, Wenzhi Gao, Herman Brunborg, Shayan Talaei, Connor Lawless, and Madeleine Udell. 2024a. Optimus-0.3: Using large language models to model and solve optimization problems at scale. *arXiv preprint arXiv:2407.19633*.

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

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2024. Self-rag: Learning to retrieve, generate, and critique through self-reflection. In *The Twelfth International Conference on Learning Representations*.

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

Albert Camus. 2013. *The myth of Sisyphus*. Penguin UK.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*.

DeepSeek-AI. 2025. Deepseek-v3.2: Pushing the frontier of open large language models.

Gurobi Optimization, LLC. 2025. Gurobi Optimizer Reference Manual.

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

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2024a. Large language models cannot self-correct reasoning yet. In *The Twelfth International Conference on Learning Representations*.

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

Caigao Jiang, Xiang Shu, Hong Qian, Xingyu Lu, JUN ZHOU, Aimin Zhou, and Yang Yu. 2025. Llmopt: Learning to define and solve general optimization problems from scratch. In *The Thirteenth International Conference on Learning Representations*.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*.

Haoyang Liu, Jie Wang, Yuyang Cai, Xiongwei Han, Yufei Kuang, and Jianye HAO. 2025a. Optitree: Hierarchical thoughts generation with tree search for llm optimization modeling. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Yitao Liu, Chenglei Si, Karthik R Narasimhan, and Shunyu Yao. 2025b. Contextual experience replay for self-improvement of language agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14179–14198.

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

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594.

Stephen Maher, Matthias Miltenberger, João Pedro Pedroso, Daniel Rehfeldt, Robert Schwarz, and Felipe Serrano. 2016. PySCIPOpt: Mathematical programming in python with the SCIP optimization suite. In *Mathematical Software – ICMS 2016*, pages 301–307. Springer International Publishing.

Xiaotian Pan, Junhao Fang, Feng Wu, Sijia Zhang, Yi-Xiang Hu, Shaoang Li, and Xiang-Yang Li. 2025. Guiding large language models in modeling optimization problems via question partitioning. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, pages 2657–2665.

Rindranirina Ramamonjison, Timothy Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, and 1 others. 2023. Nl4opt competition: Formulating optimization problems based on their natural language descriptions. In *NeurIPS 2022 Competition Track*, pages 189–203. PMLR.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652.

Segev Wasserkrug, Leonard Boussioux, Dick Den Hertog, Farzaneh Mirzazadeh, Ş Ilker Birbil, Jannis Kurtz, and Donato Maragno. 2025. Enhancing decision making through the integration of large language models and operations research optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 28643–28650.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Ziyang Xiao, Dongxiang Zhang, Yangjun Wu, Lilin Xu, Yuan Jessica Wang, Xiongwei Han, Xiaojin Fu, Tao Zhong, Jia Zeng, Mingli Song, and 1 others. 2023. Chain-of-experts: When llms meet complex operations research problems. In *The twelfth international conference on learning representations*.

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

Zhicheng Yang, Yiwei Wang, Yinya Huang, Zhijiang Guo, Wei Shi, Xiongwei Han, Liang Feng, Linqi Song, Xiaodan Liang, and Jing Tang. 2024. Optibench meets resocratic: Measure and improve llms for optimization modeling. In *The Thirteenth International Conference on Learning Representations*.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. 2024. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 19632–19642.

Lily Zhong, Zilong Wang, and Jingbo Shang. 2024. Ldb: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*.

A Appendix

A.1 Optimization Problem Formulation

Operations Research (OR) optimization focuses on determining the optimal allocation of scarce resources to maximize specific objectives under strict constraints. In the context of this work, we primarily address MILP problems. An optimization model $\mathcal{M}$ is formally defined as a tuple $\langle \mathcal{V}, \mathcal{O}, \mathcal{C} \rangle$, comprising decision variables, an objective function, and a set of constraints.

The decision variables $\mathcal{V} = \{x_1, x_2, \ldots, x_n\}$ represent the unknown quantities that the solver must determine. Each variable x_j is associated with a domain $\mathcal{D}_j$, which may be continuous (real values) or discrete (integers or binary values). The integration of discrete variables distinguishes MILP from standard Linear Programming and introduces NP-hard complexity, thereby necessitating sophisticated branch-and-bound algorithms for resolution.

The objective function $\mathcal{O}$ defines the goal of the optimization, typically formulated as a maximization or minimization of a linear combination of decision variables. The constraints $\mathcal{C}$ impose restrictions on the feasible region of the solution space. A canonical MILP formulation can be expressed as follows:

$$\begin{aligned} \min_{\mathbf{x}} \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \\ & x_j \in \mathbb{Z}, \quad \forall j \in \mathcal{I} \end{aligned} \tag{10}$$

Here, $\mathbf{x} \in \mathbb{R}^n$ denotes the vector of decision variables. The vector $\mathbf{c} \in \mathbb{R}^n$ represents the objective coefficients, while $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{b} \in \mathbb{R}^m$ define the constraint matrix and the right-hand side vector, respectively. The set $\mathcal{I} \subseteq \{1, \ldots, n\}$ identifies the subset of variables restricted to integer values.

The task of Text-to-Optimization modeling requires the Language Model to strictly map the natural language problem description $\mathcal{P}$ into this mathematical structure. It must accurately identify the dimensions of $\mathbf{A}$, correct the logic within the inequalities, and rigorously distinguish between continuous and integer variable domains. A feasible solution satisfies all constraints in $\mathcal{C}$, whereas an optimal solution is a feasible solution that achieves the best value for $\mathcal{O}$.

A.2 Notations

For clarity and ease of reference, we summarize the primary mathematical notations and symbols utilized throughout the EOM framework and the problem formulation in Table 3.

Symbol	Definition
$\mathcal{P}, \mathcal{M}$	The natural language problem description and the corresponding executable optimization model.
$\mathcal{S}, \mathcal{F}$	The external solver (e.g., Gurobi/SCIP) and its deterministic feedback signal.
π_θ	The Large Language Model policy parameterized by θ.
$\mathcal{H}_t$	The interaction history sequence accumulated up to turn t.
$\mathcal{E}, e$	The experience pool and a single experience unit containing symptom and insight.
$\mathbf{S}, \mathbf{I}$	The hybrid symptom state vector and the distilled corrective insight.
$\mathbf{Q}$	The query vector constructed for structure-aware retrieval.
$\mathcal{C}_{aug}$	The augmented input context injected with retrieved experience.
Optimization Formulation	
$\mathcal{V}, \mathcal{O}, \mathcal{C}$	The components of an optimization model: decision variables, objective function, and constraints.
$\mathbf{x}, \mathbf{c}$	The vector of decision variables and the objective coefficient vector.
$\mathbf{A}, \mathbf{b}$	The constraint matrix and the right-hand side vector defining the feasible region.
n, m	The dimensions representing the number of variables and constraints, respectively.
$\mathcal{I}$	The subset of indices denoting variables restricted to integer domains.
$\mathcal{D}_j$	The specific domain associated with the j-th decision variable.

Table 3: **Nomenclature and definitions used in the EOM framework.**

A.3 Datasets and Benchmarks

We evaluate our framework on seven widely recognized optimization modeling benchmarks. To ensure a comprehensive assessment of robustness, we categorize these datasets into three difficulty levels based on problem length, constraint density, and logical complexity:

- **Level 1:** Includes NL4Opt, NLP4LP, and MAMO-Easy. These datasets feature concise descriptions and standard linear/integer programming structures, primarily focusing on variable identification and simple arithmetic constraints.

Dataset	Size	Word Count		
		Min	Max	Mean
Level 1: Basic Standard Programming				
NL4Opt	230	58	153	100.1
NLP4LP	242	58	153	100.2
MAMO Easy	652	92	316	172.9
Level 2: Complex Constraints				
OptiBench	605	11	356	117.1
ComplexOR	18	130	298	218.8
ComplexLP	211	85	631	317.9
Level 3: Industrial Scale & Reasoning				
OptMATH	166	181	1351	507.3
IndustryOR	100	49	2450	209.9

Table 4: **Statistics of the optimization benchmarks.** The datasets are grouped by difficulty level. *Size* denotes the number of problem samples. Word counts reflect the length of the natural language problem descriptions.

- **Level 2:** Comprises OptiBench, ComplexOR, and ComplexLP. These tasks involve longer narratives and require multi-step reasoning to model complex dependencies effectively.
- **Level 3:** Consists of OptMATH and IndustryOR. These represent the most challenging scenarios, characterized by intricate industrial contexts, high-dimensional variables, and lengthy problem statements.

All datasets utilized in this study are publicly available and adhere to standard open-source licenses, specifically MIT or CC-BY-SA 4.0. Detailed statistics regarding the scale and length of problem descriptions are provided in Table 4.

A.4 Additional Experimental Analysis

In this section, we provide a deeper analysis of the computational efficiency of the EOM framework and a supplementary comparison against RAG baselines.

A.4.1 Efficiency and Overhead Analysis

To assess the practical viability of EOM, we analyze the trade-off between the additional overhead introduced by the retrieval mechanism and the savings gained from reduced trial-and-error iterations. Figure 6 visualizes the impact of Experience Replay on computational resources across varying difficulty levels.

Method	NL4Opt	NLP4LP	IndustryOR
OptiMUS-0.3 (RAG)	86.6%	73.7%	37.0%
EOM (Ours)	**91.7%**	**80.1%**	**40.0%**

Table 5: **Performance comparison against RAG.** Using GPT-4o as the backbone, EOM outperforms standard RAG methods that retrieve static code examples. This highlights the superiority of retrieving *dynamic rectification trajectories* over static solutions.

Accelerated Convergence. As shown in Figure 6b, integrating experience significantly mitigates the "Sisyphus" looping behavior. On average, the experience-augmented model reduces the conversation length from 2.22 to 1.72 turns. This confirms that retrieved *Prescriptions* effectively guide the model to fix errors in a single step, while *Warnings* preemptively prune invalid search paths.

Token Economy. Although retrieving and injecting prompts introduces a marginal input cost, it yields substantial net savings by short-circuiting futile rectification loops. As illustrated in Figure 6a, EOM achieves an average token saving rate of 28.65%. Notably, in Level 3 tasks where standard baselines often exhaust the maximum interaction budget, our method maintains a stable token footprint, validating its scalability for industrial-scale problems.

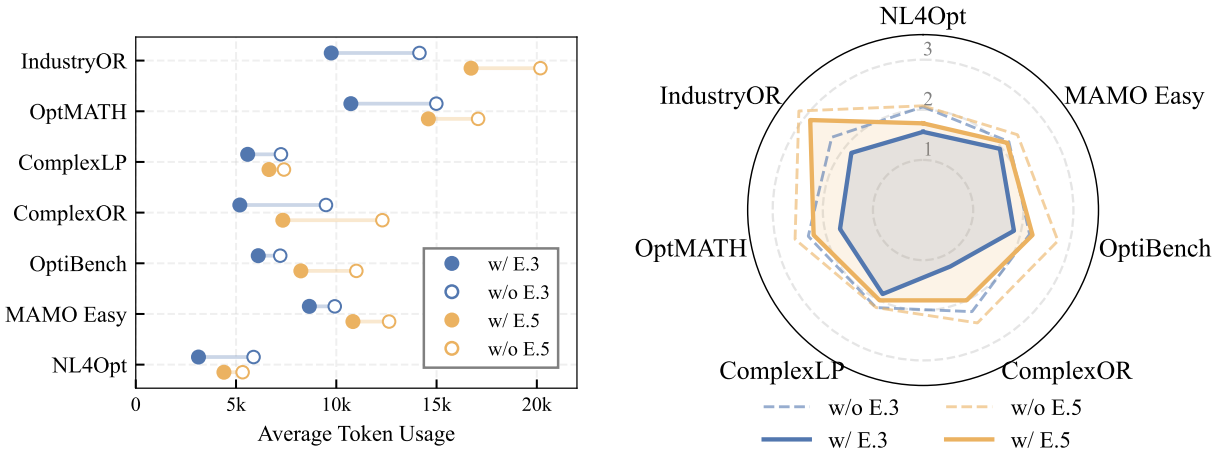
A.4.2 Comparison with RAG Baselines

We further contrast EOM with a strong RAG-based baseline, OptiMUS-0.3 (AhmadiTeshnizi et al., 2024a), which utilizes GPT-4o to retrieve static problem-code pairs as few-shot demonstrators. As detailed in Table 5, EOM consistently outperforms the RAG approach.

The performance gap highlights a critical insight: for complex reasoning tasks, simply providing a "correct final answer" is insufficient. EOM's advantage lies in retrieving the *correction logic*explicitly teaching the model how to transition from an erroneous state to a valid solution.

A.5 Algorithm Details

We decouple the EOM framework into two coordinated processes. Algorithm 2 details the **EOM-Solver**, which handles the online interaction between the LLM and the solver, utilizing retrieved insights to navigate the solution space. Algorithm 1 outlines the **EOM-Acquisition** mecha-



(a) Computational Cost (Tokens)

(b) Interaction Depth (Turns)

Figure 6: **Ablation study on efficiency: Token and turn consumption.** Comparison of (a) average token usage and (b) conversation turns. The results demonstrate that our proposed method significantly reduces overhead compared to the baseline.

nism, which acts as the memory consolidation layer, distilling transient interaction histories into persistent structural knowledge.

Algorithm 1 Acquisition: Experience Learning

Require:
History $\mathcal{H}$, Final Model $\mathcal{M}_{final}$, Status S
Distillation Prompts $\mathcal{D}_{pos}, \mathcal{D}_{neg}$
Memory Bank $\mathcal{E}$
Ensure: Updated Memory $\mathcal{E}'$
{**Phase 1: Trajectory Distillation**}
if S is Success **then**
  // Retrospective Analysis
  $\mathbf{I}_{insight} \leftarrow \mathcal{D}_{pos}(\mathcal{H}, \mathcal{M}_{final})$
  Extract: Diagnosis + Prescription
else
  // Dead-End Pruning
  $\mathbf{I}_{insight} \leftarrow \mathcal{D}_{neg}(\mathcal{H})$
  Extract: Traps + Warnings
end if
{**Phase 2: Representation Encoding**}
// Extract first error state for indexing
Let $(\mathcal{M}_{err}, \mathcal{F}_{err})$ be the pivotal error in $\mathcal{H}$
$\mathbf{S}_{symptom} \leftarrow \phi_{\text{sem}}(\mathcal{P}) \oplus \phi_{\text{ast}}(\mathcal{M}_{err}) \oplus \phi_{\text{trace}}(\mathcal{F}_{err})$
{**Phase 3: Indexing**}
$e_{new} \leftarrow \langle \mathbf{S}_{symptom}, \mathbf{I}_{insight} \rangle$
$\mathcal{E}' \leftarrow \mathcal{E} \cup \{e_{new}\}$

Algorithm 2 Solver: Online Execution Loop

Require:
Problem $\mathcal{P}$, Solver $\mathcal{S}$, Policy π_θ, Memory $\mathcal{E}$
Interaction Budget T_{max}
Ensure: Optimal Model $\mathcal{M}^*$ or Failure $\emptyset$
{**Phase 1: Proactive Initialization**}
$\mathcal{H}_0 \leftarrow \emptyset$
$\mathbf{Q}_{init} \leftarrow \langle \phi_{\text{sem}}(\mathcal{P}), \mathbf{0}, \mathbf{0} \rangle$
$e^* \leftarrow \text{Retrieve}(\mathcal{E}, \mathbf{Q}_{init})$ {Fetch similar tasks}
{**Phase 2: Iterative Refinement**}
for $t = 1$ **to** T_{max} **do**
  // Context Construction
  $\mathcal{C}_t \leftarrow \text{Augment}(\mathcal{P}, \mathcal{H}_{t-1}, e^*)$
  $\mathcal{M}_t \sim \pi_\theta(\cdot \mid \mathcal{C}_t)$
  // Deterministic Execution
  $\mathcal{F}_t \leftarrow \mathcal{S}(\mathcal{M}_t)$
  if $\mathcal{F}_t$ is Success **then**
    Call EOM-ACQUISITION($\mathcal{H}_{t-1}, \mathcal{M}_t$, Success)
    return $\mathcal{M}_t$
  end if
  // Reactive Retrieval on Error
  $\mathbf{Q}_{err} \leftarrow \langle \phi_{\text{sem}}(\mathcal{P}), \phi_{\text{ast}}(\mathcal{M}_t), \phi_{\text{trace}}(\mathcal{F}_t) \rangle$
  $e^* \leftarrow \text{Retrieve}(\mathcal{E}, \mathbf{Q}_{err})$ {Find rectification fix}
  $\mathcal{H}_t \leftarrow \mathcal{H}_{t-1} \cup \{(\mathcal{M}_t, \mathcal{F}_t)\}$
end for
Call EOM-ACQUISITION($\mathcal{H}_{T_{max}}, \emptyset$, Failure)
return $\emptyset$

A.6 Examples

A.6.1 OptiBench

Example

Problem Description: Logistics Optimization with Fuel Efficiency

Scenario: A logistics company distributes three types of goods (X, Y, Z) using a fleet of trucks. The goal is to maximize profit by optimizing truck allocation, trip counts, and investments in fuel efficiency upgrades.

Parameters

- **Fleet Constraints:**
  - Total trucks available: 50
  - Allocation requirements: At least 10 for X, 15 for Y.
  - Trip limit: Max 20 trips per truck per month.
- **Financials (per trip):**
  - **Goods X:** Revenue \$500, Base Fuel Cost \$200
  - **Goods Y:** Revenue \$600, Base Fuel Cost \$250
  - **Goods Z:** Revenue \$700, Base Fuel Cost \$300
- **Fuel Efficiency Investment:**
  - Mechanism: \$10,000 investment $\rightarrow$ \$10 cost reduction per trip.
  - Budget: Total investment cannot exceed \$60,000.
  - Modeling: Let r be the reduction amount. Total allowed reduction capacity is 60 (representing \$60,000 budget).

Mathematical Formulation

Let x, y, z be the number of trucks. Let P_x, P_y, P_z be the *total* number of trips for each goods type. Let r_x, r_y, r_z be the fuel cost reduction per trip ($r_i \geq 0$).
Objective: Maximize Total Profit Z. Profit consists of Base Profit plus Savings from fuel efficiency ($Savings = Reduction \times Trips$).

$$\max Z = (300P_x + 350P_y + 400P_z) + (r_x P_x + r_y P_y + r_z P_z)$$

Constraints:

$$\begin{aligned} x + y + z &= 50 && \text{(Total fleet size)} \\ x \geq 10, \quad y &\geq 15 && \text{(Minimum allocation)} \\ P_x \leq 20x, \quad P_y \leq 20y, \quad P_z &\leq 20z && \text{(Trip capacity constraint)} \\ r_x + r_y + r_z &\leq 60 && \text{(Investment budget limit)} \\ x, y, z, P_i, r_i &\geq 0 \end{aligned}$$

Executable Optimization Model

```
from pyscipopt import Model

data = {}

model = Model()
```

```
# Variables
# Truck allocation
x = model.addVar("x", lb=10)  # trucks for X (min 10)
y = model.addVar("y", lb=15)  # trucks for Y (min 15)
z = model.addVar("z", lb=0)   # trucks for Z

# Total trips variables
Px = model.addVar("Px", lb=0)  # total trips for X
Py = model.addVar("Py", lb=0)  # total trips for Y
Pz = model.addVar("Pz", lb=0)  # total trips for Z

# Fuel cost reduction variables (related to investment)
rx = model.addVar("rx", lb=0)
ry = model.addVar("ry", lb=0)  # for Y
rz = model.addVar("rz", lb=0)  # for Z

# Constraints
# 1. Total trucks must be 50
model.addCons(x + y + z == 50)

# 2. Operational limits: Max 20 trips per truck
model.addCons(Px <= 20 * x)
model.addCons(Py <= 20 * y)
model.addCons(Pz <= 20 * z)

# 3. Investment budget: Total reduction units <= 60
# Note: Text says $10k inv -> $10 reduction.
# Max inv $60k -> Max reduction sum = 60 units)
model.addCons(rx + ry + rz <= 60)

# Objective: maximize profit
# Base Profit + (Reduction * Trips)
# Note: rx * Px creates a quadratic term (Non-Linear)
obj = 300 * Px + 350 * Py + 400 * Pz
obj += rx * Px + ry * Py + rz * Pz
model.setObjective(obj, "maximize")

model.optimize()

if model.getStatus() == "optimal":
    print(f"{model.getObjVal()}")
```

A.6.2 OptMATH

Example

Problem Description: Traveling Salesperson Problem (ATSP)

Scenario: A logistics company needs to find the optimal delivery route covering 4 cities (0, 1, 2, 3). The route must start at a city, visit every other city exactly once, and return to the start

(forming a single loop), minimizing the total delivery cost.

Parameters

The problem involves 4 cities ($n = 4$). The delivery costs C_{ij} between cities are represented by the following cost matrix:

$$C = \begin{pmatrix} 0 & 616 & 567 & 143 \\ 388 & 0 & 947 & 628 \\ 495 & 650 & 0 & 604 \\ 77 & 630 & 23 & 0 \end{pmatrix}$$

Note: Row i to Column j. e.g., Cost from 0 to 1 is 616.

Mathematical Formulation

Let x_{ij} be a binary variable equal to 1 if the path from i to j is used, 0 otherwise. Let u_i be an integer variable representing the position of city i in the tour (to prevent subtours).
Objective: Minimize Total Cost.

$$\min \sum_{i=0}^{n-1} \sum_{j=0, j \neq i}^{n-1} C_{ij} x_{ij}$$

Constraints:

$$\sum_{j \neq i} x_{ij} = 1, \quad \forall i \qquad \text{(Leave each city exactly once)}$$

$$\sum_{i \neq j} x_{ij} = 1, \quad \forall j \qquad \text{(Enter each city exactly once)}$$

$$u_0 = 0 \qquad \text{(Fix starting city position)}$$

$$u_i - u_j + n \cdot x_{ij} \leq n - 1, \quad \forall i, j \in \{1, \ldots, n-1\}, i \neq j \qquad \text{(MTZ Subtour Elimination)}$$

$$x_{ij} \in \{0, 1\}$$

$$0 \leq u_i \leq n - 1$$

Executable Optimization Model

```
from pyscipopt import Model

# Data
n = 4
# Cost Matrix (C_ij)
cost = [[0, 616, 567, 143],
        [388, 0, 947, 628],
        [495, 650, 0, 604],
        [77, 630, 23, 0]]

# Model
model = Model("ATSP")

# Variables
x = {}
for i in range(n):
    for j in range(n):
```

```
        if i != j:
            # Binary variable for route i -> j
            x[i, j] = model.addVar(vtype="B", name=f"x_{i}_{j}")

# Position variables for subtour elimination (u_i)
u = [model.addVar(vtype="I", name=f"u_{i}", lb=0, ub=n-1) for i
   in range(n)]

# Constraints
# 1. Leave each city exactly once
for i in range(n):
    model.addCons(sum(x[i, j] for j in range(n) if i != j) == 1,
        name=f"out_{i}")

# 2. Enter each city exactly once
for j in range(n):
    model.addCons(sum(x[i, j] for i in range(n) if i != j) == 1,
        name=f"in_{j}")

# 3. Fix start position
model.addCons(u[0] == 0, name="fix_start")

# 4. Miller-Tucker-Zemlin (MTZ) constraints
# u_i - u_j + n*x_ij <= n - 1 for i, j != 0
for i in range(1, n):
    for j in range(1, n):
        if i != j:
            model.addCons(u[i] - u[j] + n * x[i, j] <= n - 1,
                name=f"mtz_{i}_{j}")

# Objective: Minimize total cost
model.setObjective(sum(cost[i][j] * x[i, j] for i in range(n)
   for j in range(n) if i != j), "minimize")

# Solve
model.hideOutput()
model.optimize()

# Print objective value
print(f"{model.getObjVal():.1f}")
```