AlphaOPT: Formulating Optimization Programs with a Self-Improving LLM Experience Library

Minwei Kong*
Singapore-MIT Alliance for Research and Technology
Singapore

Ao Qu*†
Massachusetts Institute of Technology
USA

Xiaotong Guo
Massachusetts Institute of Technology
USA

Wenbin Ouyang
Massachusetts Institute of Technology
USA

Chonghe Jiang
Massachusetts Institute of Technology
USA

Han Zheng
Massachusetts Institute of Technology
USA

Yining Ma
Massachusetts Institute of Technology
USA

Dingyi Zhuang
Massachusetts Institute of Technology
USA

Yuhan Tang
Massachusetts Institute of Technology
USA

Junyi Li
Singapore-MIT Alliance for Research and Technology
Singapore

Shenhao Wang
University of Florida
USA
Singapore-MIT Alliance for Research and Technology
Singapore

Haris Koutsopoulos
Northeastern University
USA

Hai Wang
Singapore Management University
Singapore
Singapore-MIT Alliance for Research and Technology
Singapore

Cathy Wu
Massachusetts Institute of Technology
USA

Jinhua Zhao
Massachusetts Institute of Technology
USA
Singapore-MIT Alliance for Research and Technology
Singapore

arXiv:2510.18428v3 [cs.AI] 15 Feb 2026

Abstract

Optimization modeling underlies critical decision-making across industries, yet remains difficult to automate: natural-language problem descriptions must be translated into precise mathematical formulations and executable solver code. Existing LLM-based approaches typically rely on brittle prompting or costly retraining, both of which offer limited generalization. Recent work suggests that large models can improve via experience reuse, but how to systematically acquire, refine, and reuse such experience in structurally constrained settings remains unclear. We present **AlphaOPT**, a self-improving experience library that enables LLMs to learn optimization modeling knowledge from limited supervision, including answer-only feedback without gold-standard programs, annotated reasoning traces, or parameter updates. AlphaOPT operates in a continual two-phase cycle: a *Library Learning* phase that extracts solver-verified, structured insights from failed attempts, and a *Library Evolution* phase that refines the applicability of stored insights based on aggregate evidence across tasks. This design allows the model to accumulate reusable modeling principles, improve transfer across problem instances, and maintain bounded library growth over time. Evaluated on multiple optimization benchmarks, AlphaOPT steadily improves as more training data become available (65% → 72% from 100 to 300 training items) and outperforms the strongest baseline by 9.1% and 8.2% on two out-of-distribution datasets. These results demonstrate that structured experience learning, grounded in solver feedback, provides a practical alternative to retraining for complex reasoning tasks requiring precise formulation and execution. AlphaOPT code and data are available at https://github.com/Minw913/AlphaOPT.

CCS Concepts

• **Applied computing** → **Operations research**; • **Computing methodologies** → **Artificial intelligence**.

Keywords

Large Language Models, Optimization Problems, Experience Learning

1 Introduction

Optimization models support critical decision-making in finance, manufacturing, marketing, transportation, and logistics [1, 3, 21]. Beyond improving efficiency, automating the optimization workflow lowers the barrier to operations research expertise in industry

*Equal contribution.
†Corresponding author.

and enables non-experts to prototype faster, iterate on formulations, and deploy solver-backed decisions at scale. Yet this process has long been challenging, because informal and often ambiguous specifications must be mapped to precise, domain-specific formulations and paired with appropriate code and solvers. This creates major bottlenecks for end-to-end automation [9].

Advances in large language models (LLMs) make this vision increasingly feasible: They can parse natural language requirements [16], generate executable programs [11, 14], and orchestrate downstream tools [20]. Two main lines of work have emerged. Prompt-based systems steer general LLMs with structured prompts and tool use [1, 24, 27, 35]. Fine-tuning approaches adapt models on domain corpora and benchmarks [6, 31]. Despite this progress, both families face important limitations. Prompt-based systems improve only within the scope of their fixed prompt templates and are sensitive to small wording changes and domain shifts. Fine-tuned models, on the other hand, require retraining as new data arrives and, critically, most benchmarks and datasets in the community (e.g., NLP4LP [1], MAMO [7], IndustryOR [6]) provide only final programs or solutions rather than the intermediate reasoning that governs modeling decisions, which limits the generalizability of fine-tuning approaches.

This motivates a new learning paradigm for optimization formulation: Instead of relying solely on prompts or retraining, LLMs should continually improve by accumulating, refining, and reusing solver-verified modeling insights. Indeed, recent approaches such as Reflexion [22], STaR [34], ExpeL [37], and AlphaEvolve [15] demonstrate that large models can improve through experiential reuse, storing reflections, rationales, or code edits and applying them in new tasks. These methods have been effective in common reasoning tasks such as information retrieval [30] and web browsing [32], but they face limitations for optimization problems. First, their experiences are largely preserved as free-form text or edits, without explicit applicability semantics or refinement mechanisms to represent, verify, and refine applicability-sensitive modeling knowledge; in optimization tasks, applying mismatched experiences can therefore have detrimental effects. Second, when generating experiences, these methods often lack rigorous replay-based validation, or rely on verifications limited to task-level outcomes (e.g., rewards, final answers, or test-case success), which does not guarantee that the underlying knowledge is structurally valid or transferable. Third, their experiences tend to be organized as a flat list of items, whereas operations research involves diverse insights across different modeling stages and problem types, inherently requiring a more efficient organization of acquired knowledge.

Regarding these challenges, we introduce **AlphaOPT**, an experience learning framework for optimization formulation that builds and continually refines a library of solver-verified modeling insights. In contrast to prior approaches that store experience as unstructured text or static edits, **AlphaOPT** formulates experience as structured, reusable insights with explicit applicability semantics. Each insight is defined as a four-tuple(*taxonomy*, *condition(applicability)*, *explanation*, *example*), specifying not only what modeling rule to reuse, but when and why it should be applied given a natural-language problem description. **AlphaOPT** improves performance through a continual two-phase cycle without updating model parameters. In the *Library Learning* phase, the framework self-explores optimization tasks and autonomously constructs a hierarchical taxonomy to organize insights across modeling stages, while relying on solver verification to ensure correctness even in the absence of gold-standard programs. In the *Library Evolution* phase, **AlphaOPT** diagnoses systematic misalignments between tasks and insights and refines applicability conditions based on aggregate evidence, preventing insights from being either overly narrow or overly general. Compared to existing experience learning methods, **AlphaOPT** introduces **three key advances**: (1) a self-updating hierarchical taxonomy for organizing insights; (2) solver-verified insight generation and updating; and (3) an aggregate-driven refinement mechanism that continuously improves the precision and generalizability of insight applicability.

We conduct quantitative experiments across multiple benchmarks and baselines. Compared with other approaches, the results show that **AlphaOPT** not only achieves state-of-the-art performance on multiple benchmark datasets without any gold-standard formulation or program, but also demonstrates strong generalization on out-of-distribution datasets, and continual performance growth with converging library size. These results demonstrate the efficacy of experience learning for optimization formulation, which paves the way toward more challenging settings, such as efficient program formulation and large-scale optimization. Compared to existing approaches, **AlphaOPT** makes the following contributions:

- We introduce the first experience learning framework for natural-language optimization modeling, which represents solver-verified modeling knowledge as condition-aware insights that are continually refined over time.
- Our experiments show that AlphaOPT generalizes strongly beyond training distributions, achieving state-of-the-art performance on the LogiOR and OptiBench benchmarks.
- We provide extensive qualitative and quantitative analyses of the learned experience library and its refinement dynamics, demonstrating explicit, interpretable, and human-inspectable knowledge transfer in operations research domains.

2 Related Work

2.1 LLMs for Solving Optimization Problems

Recent work increasingly uses LLMs as autonomous agents for optimization, mainly following two paradigms: formulation and heuristic generation [26, 29, 33]. LLM-driven formulation tasks aim to translate natural-language problem statements into solver-ready mathematical programs and executable solving code. Multiple benchmark datasets have been proposed to cover diverse problem families such as LP, MILP, and NLP [1, 27, 28, 31].

Existing approaches can be broadly grouped into prompt-based and training-based methods. Prompt-based methods use multi-step prompting and structured reasoning to guide LLMs in parsing problems, extracting variables/constraints, and synthesizing models. Representative examples include Chain-of-Experts [27], which leverages multi-role decomposition, and OptiMUS [1], which employs a self-reflect agentic pipeline with retrieval-augmented prompting to produce mathematical formulations. LEAN-LLM-OPT [12] further integrates a three-agent workflow with RAG and few-shot learning for problem typing, example retrieval, and

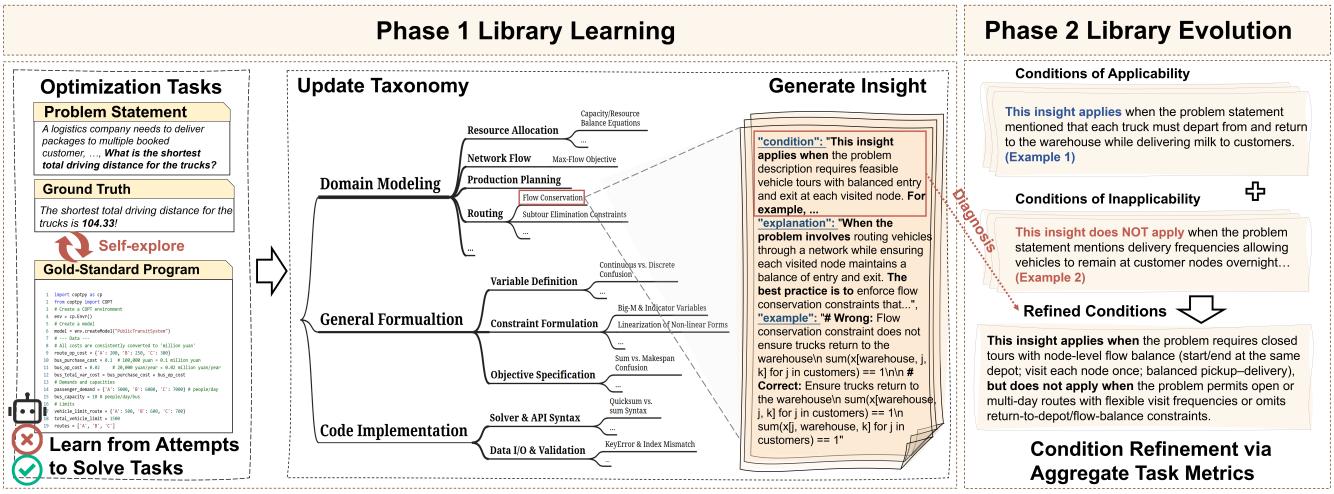


Figure 1: AlphaOPT iteratively learns an experience library through two phases: a Library Learning phase that extracts structured insights from failed attempts and organizes them in a hierarchical taxonomy, and a Library Evolution phase that refines insight applicability based on all associated tasks avoid over- or under-generalization.

data grounding. However, these methods lack sustained knowledge injection and long-term memory, which can lead to performance plateaus under distribution shift. Training-based methods improve robustness by building instruction corpora and fine-tuning open-source models. For example, ORLM [6] synthesizes natural-language optimization problem data used for fine-tuning, and LLMOPT [8] adopts a five-element schema with multi-instruction SFT on expert-labeled and augmented data. Recent post-training work further applies RL-style refinement, such as OR-R1 [4], to improve single-shot performance and robustness. Nonetheless, these pipelines rely heavily on high-quality, diverse data, and are often hard to interpret or audit.

Overall, despite progress in prompting, data synthesis, fine-tuning, and post-training, existing methods remain insufficient for reliable deployment and sustained improvement in real-world operations research systems.

2.2 Decision-making Tasks with Experience Learning

Experience Learning studies how LLM-based agents extract and reuse transferable structured experience from historical interactions to continuously improve downstream decision-making and execution [22, 36]. Such experience can take the form of episodic traces [18] (e.g., tool/action sequences), semantic insights abstracted from successful or failed attempts [36]. This line of work is primarily evaluated on long-horizon, interactive, feedback-rich decision-making tasks, including multi-tool planning and web interaction [19, 38], repository-level program repair/code generation [10], and multi-step reasoning problem solving [22, 25].

Shinn et al. [22] propose Reflexion, which enables language agents to learn from trial-and-error by converting task feedback into verbal self-reflections that are stored as episodic memory and reused in subsequent attempts. ExpeL [36] jointly extracts similar exemplars/trajectories together with their associated insights to guide task execution and further employs a voting-based mechanism to downweight mutually contradictory insights, thereby improving single-shot performance and cross-task generalization. At a more symbolic end, Zhu et al. [39] explicitly induces a rule library and maintains it using quantitative filters such as coverage and confidence, enhancing controllability and robustness of experience reuse. AgentKB [23] and ReasoningBank [17] move toward organizing an evolving reasoning memory to support self-improving agents across tasks.

However, current frameworks face systematic limitations in three key aspects. First, regarding experience applicability, most methods still rely on embedding-based similarity retrieval or coarse metadata to match experiences [18, 25, 36], while lacking explicit, verifiable preconditions (when an experience applies, when it should not, and what assumptions it requires), which increases the risk of negative transfer under distribution shift. Second, for strict verification of experience generation, merging, and editing, existing mechanisms often depend on LLM self-assessment [5, 22, 36]; for instance, ExpeL's voting-based filtering can remove contradictions but does not guarantee semantic correctness. Finally, regarding cross-task generalization, existing experience libraries typically support experience accumulation and local pruning [17, 23, 36], but lack a systematic mechanism to assess whether an experience that helps on local source tasks may degrade performance on broader related task sets.

3 Methodology

Building reliable LLM-based systems for solving optimization problems remains challenging. To begin with, gold-standard formulations and programs are scarce and may contain annotation errors [9, 28], while datasets with only answer-level supervision are underexploited [6, 7, 13]. Moreover, although optimization tasks arrive with diverse natural-language descriptions, they often share

recurring modeling rules that activate under identifiable conditions; however, existing methods still struggle to learn such when-to-apply-what knowledge. Fine-tuned models [6, 9] often overfit to syntax without mastering applicability, whereas prompt-based agent systems [1, 27, 28] rely on human empirical curation and lack the capacity to adapt or continually learn from larger datasets.

To this end, we propose AlphaOPT, which constructs an experience library of reusable insights with explicit applicability conditions, and then iteratively evolves these conditions at population level to improve generalization. As illustrated in Figure 1, it consists of two complementary phases that form a continual cycle of acquisition and refinement. The first phase, Library Learning, extracts structured insights from individual failed tasks under answer-only supervision while minimizing redundancy. The second phase, Library Evolution, diagnoses misalignments between insights and tasks and refines applicability conditions to enhance generalization while reducing confusion caused by overgeneralization. In Section 3.3, we provide a mathematical interpretation that frames library construction as maximizing task success with a size regularizer. Implementation details for the different phases are provided in the Appendix A.

To the best of our knowledge, our framework is the first to adapt experience learning to operations research (OR) with three key innovations: (1) structured knowledge for interpretability and auditability: Each insight is represented with taxonomy, condition, explanation, and example, which renders its applicability explicit, reviewable, and even revisable in practice; (2) solver-guided verifiability: If a program generated by LLM achieves the optimal objective under the solver, it is highly likely to be valid and can serve as a reliable anchor for extracting insights, which broadens the sources of experience collection. Newly generated insights are retest to be retrievable in the library and to succeed when replayed on its source task, which ensures that they are valid before integration; (3) refinement of experience applicability for generalizability and preciseness: Applicability conditions are refined using cross-task evidence, so insights neither over-generalize nor remain too narrow, which improves safe transfer across problem families.

3.1 Library Learning

This stage is designed to generate reusable insights as structured 4-tuples (*Taxonomy*, *Condition*, *Explanation*, *Example*) and organize them in a hierarchical taxonomy dictionary for efficient retrieval while minimizing redundancy in the library. Figure 2 (Library Learning Flow) clearly illustrates the overall workflow.

Insight Extraction and Representation. For each arriving task, the system first constructs a mathematical formulation, then generates an executable solver program and invokes the solver. When the library is non-empty, both steps are guided by retrieved insights. If the generated program does not achieve the correct optimal value, the system performs solver-guided self-exploration: It iteratively proposes executable programs, reuses prior failures as context, and receives verification from the solver. Once a program achieves its correct objective, it is treated as a proxy for the gold standard in anchor insight extraction.

Each insight is represented as a structured 4-tuple: *Taxonomy*, hierarchical labels for indexing and retrieval; *Condition*, an explicit description of the applicable prerequisites and triggering signals in the problem; *Explanation*, the underlying principle of applying this insight; and *Example*, a concrete demonstration such as corresponding mathematical expressions or code snippet.

Insight Verification. To ensure the retrievability and correctness of newly generated natural-language insights and to prevent the uncontrolled accumulation of idle or invalid entries in the library, each insight undergoes two stages of strict verification before being stored: *Retrieval verification*: temporarily incorporate the insight into a copy of the library, and check whether the source task can match it via LLM-driven retrieval; *Execution verification*: reapply the new insight together with other retrieved insights to its source task to verify that the new insight can resolve the original failure and lead to optimality.

At each stage, a limited number of rewrite attempts is allowed until the insight passes verification, during which the LLM performs targeted revisions of the taxonomy, condition or the whole insight to improve its efficacy.

Library Storage. Experience library is maintained as a dynamically updating taxonomy dictionary organized into three main tracks: *Domain Modeling* (problem-specific structures and assumptions), *General Formulation* (reusable mathematical patterns), and *Code Implementation* (solver-specific coding practices). Within each track, the library organizes insights using a two-level labeling scheme, where Level-1 captures a broad category and Level-2 specializes it into a more specific subcategory. The taxonomy dictionary is initialized with few-shot labels and expands online: Each new insight is either mapped to an existing category or, if no suitable label exists, prompts the LLM to propose new Level-1 or Level-2 labels. Each label is also assigned a condition, written by the LLM, that specifies when the category should be retrieved.

To prevent uncontrolled growth of the experience library, we incorporate an online merging mechanism throughout training to prevent degradation in retrieval and execution efficiency. For each newly generated insight, the system performs redundancy checking with existing insights, merging when appropriate, followed by correctness validation on all associated tasks.

Library Retrieval. During solution generation, retrieved insights from the *Domain Modeling* and *General Formulation* tracks guide the construction of the mathematical model, while insights from the *Code Implementation* track guide solver-code generation. To align a target task with relevant insights, we employ a two-step LLM-driven retrieval procedure: Quick label matching, then full applicability check. The system first scans the taxonomy dictionary to identify labels that are potentially relevant to the context of the tasks. For example, a Level-2 label such as Fixed Charge (Big-M Linking) will probably be detected when the problem description specifies that service or flow is allowed only if a facility is opened. After candidate labels are identified, the system rigorously evaluates each associated insight by examining its condition, and only the most applicable insights are retained. By employing a progressive disclosure strategy that reveals only task-relevant information when it becomes applicable, this procedure mitigates context overflow while maintaining the effectiveness of retrieval.

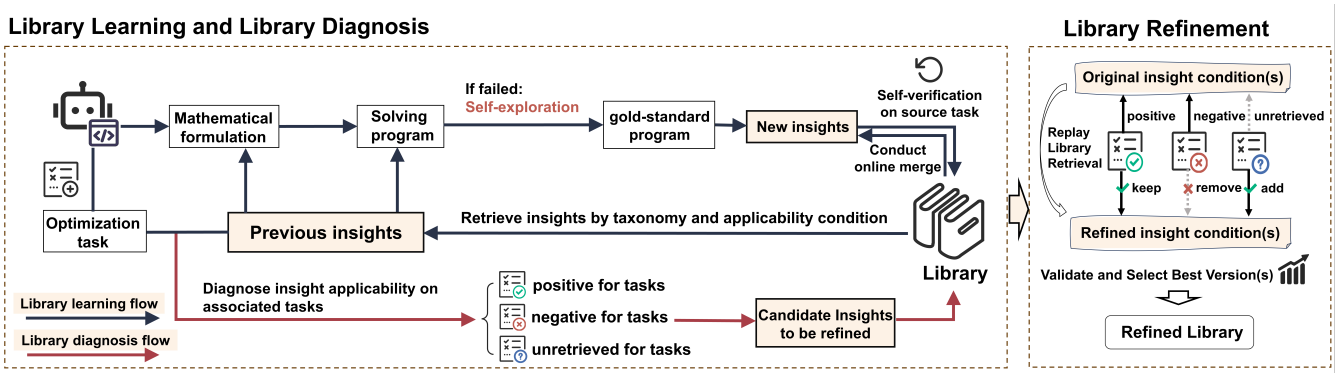


Figure 2: The overall workflow of AlphaOPT. The left panel depicts two complementary flows: library learning, which self-explores correct patterns and extracts new insights by trial and error; library diagnosis, which categorizes interactions between failed tasks and retrieved insights into positive, negative, and unretrieved cases. The right panel shows library refinement, whereby the LLM refines insight applicability, validates via retrieval replay, and reintegrates them into the library.

3.2 Library Evolution

While Library Learning expands the repository of insights, Library Evolution aims to transform task-specific lessons into broadly applicable knowledge. Since each insight's applicability is defined by a condition induced by a specific task, early conditions are often too narrow (failing to trigger on relevant tasks) or too broad (causing misretrieval). Left unchecked, these misalignments lead to widespread misguidance or missed opportunities. Library Evolution counters this with a diagnosis–refinement cycle: It detects misaligned insights, aggregates evidence across tasks, and refines conditions at the end of each iteration. The refinement is guided by an aggregate metric rather than ad hoc fixes. As illustrated in Figure 3, library refinement can be understood as adjusting each insight's condition toward the correct retrieval boundary in the problem space.

Library Diagnosis. After an initial pass over the training tasks, the system continuously diagnoses task failures and traces their interactions with library insights through iterative cycles of insight retrieval and solution generation across tasks. By comparing the formulation and program generated under the guidance of retrieved insights with the ground truth to locate discrepancies, the diagnostic LLM agent partitions the relationship between each insight i and its associated tasks into three disjoint categories: $\Pi(i) = \{\text{Positive} : S_i^+, \text{Negative} : S_i^-, \text{Unretrieved} : S_i^u\}$ where S_i^+ contains tasks for which the insight was applicable and contributed to the correct formulation and program, S_i^- contains tasks for which it was misleading and degraded performance, and S_i^u contains tasks for which it was not retrieved but would have been beneficial. By maintaining these partitions across iterations, the system continuously builds a performance profile for each insight. If a failed task is subsequently solved after removing a misleading (negative) insight or by injecting a previously unretrieved one, the system attributes the failure to condition misalignment rather than lack of knowledge and thus avoids redundant insight generation. Figure 2 (Library Diagnosis Flow) illustrates the workflow for this stage.

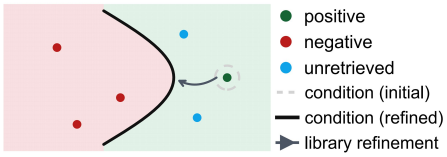


Figure 3: A locally verified *initial* condition is refined into a broader *applicability* boundary through LLM-driven refinement.

Library Refinement. Building on the diagnosed interaction of each insight, the LLM agent first strengthens or prunes its applicability condition. Negative tasks contribute explicit *inapplicability clauses* (e.g., constraints or contexts that block use), while unretrieved tasks highlight missing applicability signals. Multiple refinement strategies are asked to be proposed (e.g., adding preconditions, introducing keyword anchors, merging overlapping triggers) and produce candidate conditions with the goal of preserving correct cases, eliminating mismatches, and recovering previously missed tasks. Then, each candidate condition replaces the original and is tested over the union $R_i = S_i^+ \cup S_i^- \cup S_i^u$. A performance score

$$p_i = \frac{|\text{kept positives}| + |\text{corrected negatives}| + |\text{recovered unretrieved}|}{|R_i|}$$

quantifies improvement. Here, *kept positives* are tasks that remain correctly retrieved after refinement; *corrected negatives* are tasks that were misled by the insight before and are no longer retrieved; and *recovered unretrieved* are tasks that become correctly retrieved after refinement. We accept refinements that increase p_i and keep the one with the highest p_i. The workflow for this stage is illustrated in the right panel of Figure 2.

3.3 Optimization Perspective

The framework can be viewed as an iterative solution to the optimization problem in the library space. Let $\mathcal{L}$ denote a candidate library and $\mathcal{T}$ the distribution of the optimization problems we

Table 1: Accuracy (%) on *Test Split* and *Out-of-Distribution* datasets, along with micro- and macro-averaged accuracy across datasets. Best per column in bold and second best underlined (ablation variants excluded from highlighting).

Method		Test Split						Out-of-Distribution			
		NLP4LP (73)	NL4OPT (64)	IndustryOR (25)	MAMO (ComplexLP) (34)	Micro	Macro	LogiOR (92)	OptiBench (403)	Micro	Macro
Prompt-based	Standard	68.5	54.7	52.0	44.1	57.7	54.8	46.7	72.7	67.9	59.7
	OptiMus	71.2	73.4	36.0	29.4	60.2	52.5	17.4	74.7	64.1	46.0
	ORThought	69.9	75.0	**60.0**	41.2	65.3	61.5	44.6	<u>84.4</u>	<u>77.0</u>	<u>64.5</u>
Fine-tuning	ORLM	86.3	**87.5**	36.0	55.9	75.0	66.4	19.6	78.2	67.3	48.9
	LLMOPT	**97.3**	<u>86.5</u>	44.0	**85.8**	**85.0**	**78.4**	40.2	66.4	61.5	53.3
Experience-learning	Reflexion	76.7	64.1	<u>56.0</u>	47.1	64.8	61.0	43.5	76.9	70.7	60.2
	Expel	79.5	67.2	48.0	70.4	69.9	66.3	<u>47.8</u>	80.4	74.3	64.1
	AlphaOPT	<u>87.7</u>	81.2	**60.0**	<u>76.5</u>	<u>80.1</u>	<u>76.4</u>	**56.0**	**93.5**	**86.5**	**74.8**
Ablation (AlphaOPT)	w/o Refinement	82.2 (↓ 4.1)	80.7 (↓ 0.5)	44.0 (↓ 16.0)	61.8 (↓ 14.7)	73.3 (↓ 6.8)	67.2 (↓ 9.2)	48.9 (↓ 7.1)	92.1 (↓ 1.4)	84.1 (↓ 2.4)	70.5 (↓ 4.3)
	w/o Insight Example	86.3 (↓ 1.4)	78.1 (↓ 3.1)	56.0 (↓ 4.0)	58.8 (↓ 17.7)	76.0 (↓ 4.1)	69.8 (↓ 6.6)	45.7 (↓ 10.3)	91.0 (↓ 2.5)	82.9 (↓ 3.6)	68.4 (↓ 6.4)

want to solve. The objective is to maximize task success while penalizing library complexity to mitigate retrieval inefficiency and long-context degradation in LLM inference:

$$\max_{\ell \in \mathcal{L}} \ \mathbb{E}_{t \sim \mathcal{T}}[\text{Success}(t \mid \ell)] \ - \ \lambda\, \Omega(\ell),$$

where $\text{Success}(t \mid \ell)$ indicates whether ℓ enables the system to produce a program that achieves the correct optimal objective for task t, and $\Omega(\ell)$ quantifies library complexity (e.g., number of insights or redundancy-adjusted size). Under our problem design—bounded and continuous property of $\text{Success}(\cdot)$ and $\Omega(\cdot)$, sufficient exploration under solver verification, and bounded merging—the refinement dynamics converge to a locally optimal library. In Appendix A, we provide a conceptual sketch that shows that convergence holds: Since refinement in the second phase strictly improves the first term, while verified merging in the first phase reduces the second term without diminishing the first, sufficient exploration combined with iterative cycles of library learning and evolution ensures convergence to a local optimum. Given the inherent ambiguity of natural language and stochasticity in LLM outputs, we present this perspective not as a strict theorem but as a principled justification for the acquisition–refinement design and redundancy-reduction operations.

4 EXPERIMENTS

Our experiments are designed to reflect the requirements that arise in real-world optimization and operations research applications. In these settings, methods are expected not only to perform well on standard benchmarks but also to transfer across domains, remain effective when limited supervision is available, improve steadily as more data become available, and offer results that can be inspected and audited. We organize our empirical evaluation around four questions: (1) How well does the method generalize across domains? (2) How effective are the insight content structure and library refinement mechanisms? (3) Does performance improve consistently as more training data become available? (4) Does the library size converge as training progresses?

4.1 Experimental Setup

Our experiments are conducted on a dataset of 452 problem instances, aggregated from four real-world optimization task datasets—the NLP4LP [1], NL4OPT [21], IndustryOR [6], MAMO (ComplexLP) [7]. These collections span various formulation types and originate from diverse sources, which include academic papers, textbooks, and real-world industry scenarios. Detailed descriptions of these datasets are provided in Appendix B.1.

We perform stratified sampling within each dataset, randomly partitioning 70% for training and 30% for testing. The experience library is constructed only from training tasks, which ensures that training-derived insight examples do not leak into the test set. To assess out-of-distribution generalization, we also evaluate on LogiOR [28] and Optibench [31].

Unless otherwise specified, gpt-4o (OpenAI, 2024) with temperature 0 is used as the backbone. We use the success rate as the primary evaluation metric, following the evaluation protocol of Yang et al. [28], whereby a task is considered successful if the LLM-generated optimal value closely aligns with the provided ground-truth solution. We also report micro- and macro-averaged accuracy for both test-split and out-of-distribution datasets (see Appendix B.2 for definitions).

Baselines. We evaluate against three families of baselines. (i) **Prompt Designing**: directly generate the mathematical model from manually crafted prompts, which include OptiMUS [1], and ORThought [28]. (ii) **Fine-tuning**: adapt model parameters through fine-tuning, including ORLM [6], built on LLaMa3-8B, and LLMOPT [9], built on Qwen2.5-14B (the latest open-source version available after their paper). (iii) **Experience Learning**: Accumulate reusable knowledge from past task-solving attempts to improve decision

quality across tasks, including two representative works Reflexion [22] and ExpeL [37]. For ExpeL, we adapt the original framework to operations research tasks by making minor changes to task-related prompts and evaluation criteria.

4.2 Overall Performance

Table 2 summarizes the overall performance across test and out-of-distribution benchmarks. **AlphaOPT** consistently outperforms prompt-based and experience-learning baselines on the test splits and achieves the best performance on the more challenging LogiOR, IndustryOR, and out-of-distribution OptiBench datasets, exceeding the second-best method by an average of 7%. Although fine-tuned models (ORLM and LLMOPT) achieve higher scores on certain in-distribution benchmarks such as NLP4LP and NL4OPT, these gains are less conclusive, as their training data either overlap with our evaluation splits or are not publicly available. Moreover, many existing benchmarks are derived from a small set of seed problems [7, 21], which favors fine-tuning approaches that excel at pattern memorization. In contrast, while fine-tuning methods require detailed annotations of mathematical formulations and code to achieve strong performance, **AlphaOPT** attains competitive results using only answer labels. This advantage is particularly significant in OR practice, where high-quality, expert-annotated datasets are generally scarce.

Compared with baseline experience-learning methods, our approach achieves superior performance; our analysis in Appendix B.3 further shows that methods such as ExpeL tend to learn generic insights that lack the detailed modeling and coding guidance needed to prevent common errors.

In particular, we evaluate robustness to distribution shift using two benchmarks not seen during training: LogiOR [28] and OptiBench [31]. Fine-tuned models show strong in-distribution performance but degrade substantially on these unseen datasets. For example, ORLM drops to 19.6% accuracy on LogiOR and 13.3% on OptiBench, while LLMOPT also exhibits noticeable degradation. In contrast, **AlphaOPT** maintains substantially higher accuracy, achieving 56.0% on LogiOR and 93.5% on OptiBench. These results suggest that fine-tuning tends to capture dataset-specific solution patterns, whereas the experience library in **AlphaOPT** enables the learning of transferable modeling principles, leading to stronger robustness under distribution shift.

4.3 Ablation Study

We examine two ablations of AlphaOPT: removing the library refinement stage (**w/o Refinement**) and removing concrete examples from insights (**w/o Insight Example**). In **w/o Refinement**, insights are stored after extraction without further adjustment of their applicability conditions. In **w/o Insight Example**, insights exclude concrete mathematical or code-level demonstrations.
Removing refinement leads to consistent performance drops across both test-split and out-of-distribution datasets. On the test split, micro- and macro-averaged accuracy decrease by 6.8 and 9.2 points respectively, with the largest degradation on structurally complex datasets such as IndustryOR and MAMO (ComplexLP). Out-of-distribution performance also declines, indicating that unrefined applicability conditions are insufficient for reliable cross-task reuse.

Table 2: AlphaOPT steadily improves in both Micro and Macro averages with increasing training size.

Training Size	MicroAvg (%)	MacroAvg (%)
100	83.24	65.80
200	85.09	69.22
300	85.21	72.12

Removing insight examples results in a smaller but systematic drop in aggregate accuracy, with a notably larger impact on formulation-to-code–heavy datasets.

4.4 Continual Growth and Library Converge

We test whether **AlphaOPT** can improve its performance as more data become available. We incrementally sample sets of 100, 200, and 300 data items from our training set and train **AlphaOPT** on each subset. As shown in Table 2, when evaluated on out-of-distribution datasets (LogiOR, OptiBench), we observe that **AlphaOPT** steadily improves its performance with increasing data size, without requiring updates to its model parameters.

Also, we provide empirical validation of the theoretical analysis presented in Section 3.3, under which **AlphaOPT** is expected to converge to a locally optimal experience library during training. As illustrated in Figure 4, the cumulative training accuracy increases rapidly with the number of visited tasks in the early stage and gradually saturates thereafter. Meanwhile, library size gradually slows and eventually stabilizes as training proceeds. This behavior results from the combined effects of online insight merging, solver-based verification, and the refinement mechanism, under which the number of distinct and verifiable insights that can be incorporated in the library progressively saturates.

Taken together, the joint observation of continual performance improvement and library convergence indicates that **AlphaOPT** achieves scalable learning without unbounded accumulation of redundant or marginally useful insights.

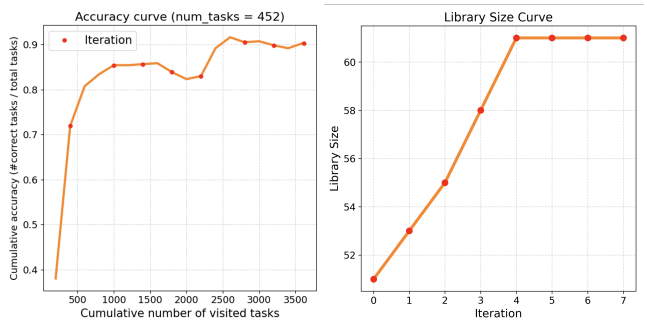


Figure 4: Training dynamics of the experience library. As training progresses, accuracy converges while library growth stabilizes.

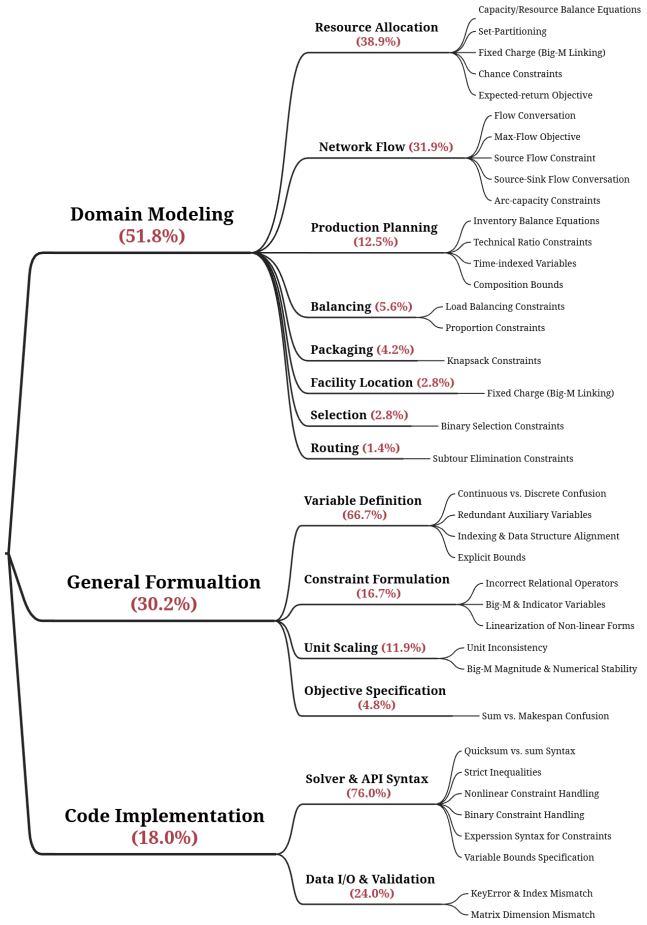


Figure 5: The learned experience library taxonomy consists of three main tracks, 14 level-1 labels, and 38 level-2 labels. Percentages shown in red indicate each track's or level-1 label's proportion.

5 Library Analysis

5.1 Library Taxonomy and Knowledge

To fully interpret the experience library, we visualize its hierarchical taxonomy and the distribution of insights (as shown in Figure 5). Detailed specification and distribution of level-2 labels are provided in the Appendix B.5 and B.7.

Overall, LLMs tend to make mistakes more frequently on mathematical modeling than in code implementation when solving optimization problems, especially when problem structures are complex or the modeling assumptions and conventions are implicit in the context. Specifically, challenges in LLM-driven optimization can be summarized as threefold:

Structural Coupling and Balance. In the Domain Modeling track, LLMs exhibit two primary difficulties: (i) capturing structural coupling across variables or stages, and (ii) maintaining global balance constraints over system resources or quantities. These challenges are particularly pronounced in problem classes such as *Resource Allocation* (38.9%), *Network Flow* (31.9%), and *Production Planning* (12.5%). For example, in production planning scenarios, LLMs often fail to correctly couple machine-level production decisions with inter-temporal inventory balance constraints; they rely instead on aggregated variables to constrain total processing time, and thus violate essential structural dependencies.

From Intuition to Rigorous Formulation. In the General Formulation track, LLMs are prone to pitfalls when translating intuitive or contextual descriptions into mathematically rigorous and solver-consistent formulations, with mistakes concentrated in *Variable Definition* (66.7%), *Constraint Formulation* (16.7%), and *Unit Scaling* (11.9%). These errors include unclear variable domains, confusion between continuous and discrete variables, misuse of relational operators, improper handling of Big-M or indicator variables, and unit inconsistencies. For example, in production or resource constraints expressed as *at least* or *at most* (e.g., minimum production requirements or budget limits), LLMs often model such conditions as equalities (=) or strict inequalities (< or >), leading to overly restrictive formulations.

Bridging Models and Executable Code. In the Code Implementation track, although such errors are less frequent, LLLMs still exhibit gaps when translating symbolic mathematical models into executable solver code. Among these errors, 76% stem from *Solver & API Syntax* (e.g., improper handling of nonlinear constraints, confusion between `quicksum` and `sum`), while the remaining 24% arise from *Data I/O & Validation* issues (e.g., `KeyError`, index mismatches, and matrix dimension errors).

5.2 Examples of Library Refinement

Representative refined insights are selected from the library to illustrate how their applicability conditions are adjusted based on the associated tasks.

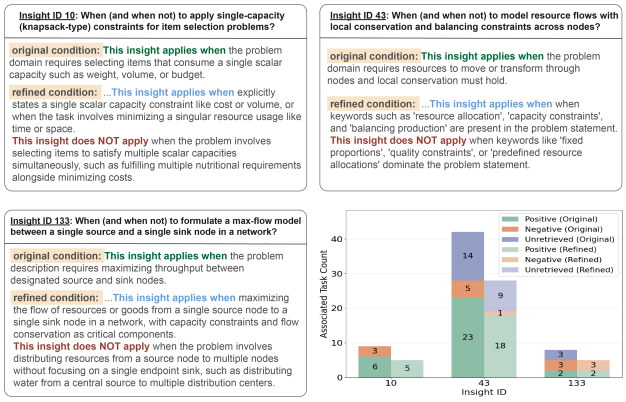


Figure 6: Examples of insight applicability conditions before and after refinement, along with the corresponding changes in task associations.

As shown in the Figure 6, *Insight ID 10* targets single–scalar-capacity selection; the refinement broadens coverage by adding equivalent formulations such as minimizing a single resource (time/space), while explicitly excluding multi-capacity settings. *Insight ID*

43 captures flow/conservation and production balancing across nodes; refinement adds lexical anchors (e.g., capacity constraints, balancing production) and excludes statements dominated by fixed proportions, quality constraints, or predefined allocations, and gating suppresses spurious retrievals whereby the structure is not true flow/conservation. *Insight ID 133* focuses on max-flow between a single source and a single sink; refinement tightens the structural requirement and rules out source-to-many distribution tasks.

Across the cases, four LLM-driven refinement strategies recur: (i) generalization via equivalent phrasings; (ii) lexical anchoring with positive keywords; (iii) explicit exclusions to reduce misalignment; and (iv) structural qualifiers to prevent misuse. As shown in Figure 6, these semantic refinements reduce both negative and unretrieved cases while preserving most positive cases.

6 Conclusion and Discussion

This paper addresses the limitations of previous methods by presenting a novel self-improving library learning framework, **AlphaOPT**, for formulating optimization programs. **AlphaOPT** can learn from answer labels only, achieves much stronger out-of-distribution generalization than fine-tuning–based methods, and provides interpretable and auditable structured knowledge to support human involvement in real-world practice. The learned experience library reveals LLMs' characteristic failure patterns and underlying causes across domain-specific modeling, mathematical formulation, and solver syntax handling. Case study results show that insights with a high success rate primarily address fundamental modeling errors with clear structures, while insights with a high mismatch rate or low effectiveness expose the overgeneralization of structurally complex knowledge and highlight the need to improve the pedagogical clarity and applicability conditions of insights.

Looking ahead, we highlight three promising directions. First, reasoning-oriented test-time scaling, which is already powerful in other domains, could be particularly effective for OR formulations, whose results are inherently verifiable. Second, strengthening datasets with both academic research and large-scale real-world industry problems will move LLM systems beyond the toy examples that dominate current benchmarks, and thus enable progress toward truly large-scale optimization tasks. Third, moving beyond correctness toward improving the efficiency of formulations is crucial for real-world deployment, and our self-improving library learning approach offers a promising path to that goal.

Acknowledgments

To Robert, for the bagels and explaining CMYK and color spaces.

References

[1] Ali AhmadiTeshnizi, Wenzhi Gao, Herman Brunborg, Shayan Talaei, and Madeleine Udell. 2024. OptiMUS-0.3: Using Large Language Models to Model and Solve Optimization Problems at Scale. *arXiv preprint arXiv:2407.19633* (2024).

[2] Nicolás Astorga, Tennison Liu, Yuanzhang Xiao, and Mihaela van der Schaar. 2025. Autoformulation of Mathematical Optimization Models Using LLMs. In *Proceedings of the 42nd International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 267)*. arXiv:2411.01679 [cs.LG] doi:10.48550/arXiv.2411.01679

[3] Dimitris Bertsimas and John N Tsitsiklis. 1997. *Introduction to linear optimization*. Vol. 6. Athena scientific Belmont, MA.

[4] Zezhen Ding, Zhen Tan, Jiheng Zhang, and Tianlong Chen. 2025. OR-R1: Automating Modeling and Solving of Operations Research Optimization Problem via Test-Time Reinforcement Learning. (2025). arXiv:2511.09092 [cs.AI] doi:10.48550/arXiv.2511.09092

[5] Erhu Feng, Wenbo Zhou, Zibin Liu, Le Chen, Yunpeng Dong, Cheng Zhang, Yisheng Zhao, Dong Du, Zhichao Hua, Yubin Xia, and Haibo Chen. 2025. Get Experience from Practice: LLM Agents with Record & Replay. *arXiv preprint arXiv:2505.17716* (2025). doi:10.48550/arXiv.2505.17716

[6] 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* (2025).

[7] Xuhan Huang, Qingning Shen, Yan Hu, Anningzhe Gao, and Benyou Wang. 2024. Mamo: a Mathematical Modeling Benchmark with Solvers. arXiv:2405.13144 [cs.AI]

[8] Caigao Jiang, Xiang Shu, Hong Qian, Xingyu Lu, Jun Zhou, Aimin Zhou, and Yang Yu. 2024. LLMOPT: Learning to Define and Solve General Optimization Problems from Scratch. (2024). arXiv:2410.13213 [cs.AI] doi:10.48550/arXiv.2410.13213

[9] 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 *Proceedings of the Thirteenth International Conference on Learning Representations (ICLR)*. Singapore, Singapore. https://openreview.net/pdf?id=9OMvtboTJg

[10] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *arXiv preprint arXiv:2310.06770* (2023). doi:10.48550/arXiv.2310.06770

[11] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[12] Kuo Liang, Yuhang Lu, Jianming Mao, Shuyi Sun, Chunwei Yang, Congcong Zeng, Xiao Jin, Hanzhang Qin, Ruihao Zhu, and Chung-Piaw Teo. 2026. Large-Scale Optimization Model Auto-Formulation: Harnessing LLM Flexibility via Structured Workflow. arXiv:2601.09635 [cs.AI] https://arxiv.org/abs/2601.09635

[13] 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* (2025).

[14] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. [n. d.]. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations*.

[15] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131* (2025).

[16] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems* 35 (2022), 27730–27744.

[17] Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. 2025. ReasoningBank: Scaling Agent Self-Evolving with Reasoning Memory. *arXiv preprint arXiv:2509.25140* (2025). doi:10.48550/arXiv.2509.25140

[18] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative Agents: Interactive Simulacra of Human Behavior. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery. doi:10.1145/3586183.3606763

[19] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. *arXiv preprint arXiv:2307.16789* (2023). doi:10.48550/arXiv.2307.16789

[20] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. In *ICLR*. https://openreview.net/forum?id=dHng2O0Jjr

[21] Rindranirina Ramamonjison, Timothy Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong Zhang. 2022. NL4Opt Competition: Formulating Optimization Problems Based on Their Natural Language Descriptions. In *Proceedings of the NeurIPS 2022 Competitions Track (Proceedings of Machine Learning Research, Vol. 220)*, Marco Ciccone, Gustavo Stolovitzky, and Jacob Albrecht (Eds.). PMLR, 189–203. https://proceedings.mlr.press/v220/ramamonjison23a.html

[22] 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 (2023), 8634–8652.

[23] Xiangru Tang, Tianrui Qin, Tianhao Peng, Ziyang Zhou, Daniel Shao, Tingting Du, Xinming Wei, Peng Xia, Fang Wu, He Zhu, Ge Zhang, Jiaheng Liu, Xingyao Wang, Sirui Hong, Chenglin Wu, Hao Cheng, Chi Wang, and Wangchunshu Zhou. 2025. Agent KB: Leveraging Cross-Domain Experience for Agentic Problem Solving. *arXiv preprint arXiv:2507.06229* (2025). doi:10.48550/arXiv.2507.06229

[24] Raghav Thind, Youran Sun, Ling Liang, and Haizhao Yang. 2025. OptimAI: Optimization from Natural Language Using LLM-Powered AI Agents. *arXiv preprint arXiv:2504.16918* (2025).

[25] Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. 2024. Agent Workflow Memory. *arXiv preprint arXiv:2409.07429* (2024). doi:10.48550/arXiv.2409.07429

[26] Xuan Wu, Di Wang, Chunguo Wu, Lijie Wen, Chunyan Miao, Yubin Xiao, and You Zhou. 2025. Efficient Heuristics Generation for Solving Combinatorial Optimization Problems Using Large Language Models. *arXiv preprint arXiv:2505.12627* (2025). https://arxiv.org/abs/2505.12627

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

[28] Beinuo Yang, Qishen Zhou, Junyi Li, Chenxing Su, and Simon Hu. 2025. Automated Optimization Modeling through Expert-Guided Large Language Model Reasoning. arXiv:2508.14410 [cs.AI] https://arxiv.org/abs/2508.14410

[29] Xianliang Yang, Ling Zhang, Haolong Qian, Lei Song, and Jiang Bian. 2025. HeurAgenix: Leveraging LLMs for Solving Complex Combinatorial Optimization Challenges. (2025). arXiv:2506.15196 [cs.AI] doi:10.48550/arXiv.2506.15196

[30] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (Eds.). Association for Computational Linguistics, Brussels, Belgium, 2369–2380. doi:10.18653/v1/D18-1259

[31] 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. (2024). arXiv:2407.09887 [cs.AI] doi:10.48550/arXiv.2407.09887 Published as a conference paper at ICLR 2025.

[32] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems* 35 (2022), 20744–20757.

[33] Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. 2024. Reevo: Large language models as hyper-heuristics with reflective evolution. *Advances in neural information processing systems* 37 (2024), 43571–43608.

[34] Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. 2022. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems* 35 (2022), 15476–15488.

[35] Bowen Zhang and Pengcheng Luo. 2025. Or-llm-agent: Automating modeling and solving of operations research optimization problem with reasoning large language model. *arXiv preprint arXiv:2503.10009* (2025).

[36] Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. 2023. ExpeL: LLM Agents Are Experiential Learners. *arXiv preprint arXiv:2308.10144* (2023). doi:10.48550/arXiv.2308.10144

[37] Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. 2024. ExpeL: LLM Agents Are Experiential Learners. arXiv:2308.10144 [cs.LG] https://arxiv.org/abs/2308.10144

[38] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2023. WebArena: A Realistic Web Environment for Building Autonomous Agents. *arXiv preprint arXiv:2307.13854* (2023). doi:10.48550/arXiv.2307.13854

[39] Zhaocheng Zhu, Yuan Xue, Xinyun Chen, Denny Zhou, Jian Tang, Dale Schuurmans, and Hanjun Dai. 2023. Large Language Models can Learn Rules. *arXiv preprint arXiv:2310.07064* (2023). doi:10.48550/arXiv.2310.07064

A Proof of the Library Convergence

Recall the optimization problem in the library training phase

$$F(\ell) \;=\; \mathbb{E}_{t\sim\mathcal{T}_{\text{train}}}\big[\, r(t \mid \ell)\,\big] \;-\; \lambda\,\Omega(\ell),$$

where $r(t,\ell)$ is a bounded reward function that implements the role of the original Success$(t \mid \ell)$ (i.e., it measures the matching quality between optimization problem t and library ℓ), and $\Omega(\ell)$ is a bounded complexity penalty.

According to the problem setting, the iterative refinement algorithm satisfies:

(1) Monotone update: At iteration k, from ℓ_k, the algorithm considers a set of admissible refinements $R(\ell_k) \subseteq \mathcal{L}$. Each accepted iteration consists of one of two types of operations:
   - *Merge step:* decreases $\Omega(\ell)$ while leaving $r(t,\ell)$ non-decrease for all relevant tasks;
   - *Exploration step:* improves $r(t,\ell)$ for some tasks without increasing $\Omega(\ell)$.

   Therefore, every accepted refinement strictly increases $F(\ell)$; otherwise, the algorithm keeps $\ell_{k+1} = \ell_k$.

(2) Sufficient exploration: Any improving neighbor $\tilde{\ell} \in R(\ell_k)$ (i.e. one with a strictly larger objective) will eventually be discovered and executed. Empirically, this is achieved through iterative prompt optimization with LLMs.

(3) Boundedness: $r(t,\ell)$ and $\Omega(\ell)$ are bounded; hence $F(\ell)$ is bounded above and below.

The following theorem establishes that, under the assumption that the training and testing distributions are identical, the refinement procedure yields libraries that are locally optimal for the testing objective.

Theorem A.1. *Assume $\mathcal{T}_{\text{train}} = \mathcal{T}_{\text{test}}$. If the library space $\mathcal{L}$ is finite, the algorithm terminates in finitely many steps at a library ℓ^* that is a local maximizer for the testing objective. Moreover, the algorithm cannot terminate at a saddle point.*

Proof. Every accepted merge or exploration step strictly increases $F(\ell)$; otherwise, the library remains unchanged. Since F is bounded above, the sequence $\{F(\ell_k)\}$ is monotone non-decreasing and bounded, and hence convergent to some limit F^*. Furthermore, since $\mathcal{L}$ is finite, define

$$\delta \;=\; \min\big\{F(\tilde{\ell}) - F(\ell) : \tilde{\ell} \in R(\ell),\; F(\tilde{\ell}) > F(\ell)\big\}.$$

Finiteness guarantees $\delta > 0$, so only finitely many strict improvements are possible. The algorithm halts at some ℓ^*. By sufficient exploration, no improving neighbor of ℓ^* exists. Therefore, ℓ^* is a local maximizer for both training and testing objectives. Saddle points are excluded.

Since the training and testing distributions coincide, the training objective equals the testing objective; thus any local optimality statement directly applies to testing. □

Although the assumption of a finite library is reasonable, we also provide a proof for the case of an infinite library for completeness and rigor.

Theorem A.2 (Infinite compact library case). *Assume $\mathcal{T}_{\text{train}} = \mathcal{T}_{\text{test}}$. If the library space $\mathcal{L}$ is compact (closed and bounded) and F*

Table 3: Statistics of the optimization problem datasets

Dataset	Size	Formulation Type(s)	Completion
NL4OPT [21]	289	LP	solution
NLP4LP [1]	242	LP, MILP, MINLP	solution, program
MAMO (complex LP) [7]	111	LP	solution
IndustryOR [6]	82	LP, IP, MILP, NLP, others	solution, program
Optibench [31]	403	LP, MILP, MINLP	solution
LogiOR [28]	92	LP, IP, MIP, NLP	solution

Abbreviations: LP – Linear Programming; IP - Integer Programming; NLP – Nonlinear Programming; MI – Mixed-Integer; others - Quadratic Programming, Dynamic&Stochastic Programming, etc.

is continuous, then the sequence $\{F(\ell_k)\}$ converges, and any subsequential limit point ℓ^∞ is a local maximizer for the testing objective. Saddle points are excluded for all such limit points.

Proof. Each accepted step strictly increases $F(\ell)$, so $\{F(\ell_k)\}$ is monotone non-decreasing. Since F is bounded above, $\{F(\ell_k)\}$ converges to some F^*. By compactness of $\mathcal{L}$, there exists a convergent subsequence $\ell_{k_j} \to \ell^\infty$. The continuity of F ensures $F(\ell_{k_j}) \to F(\ell^\infty) = F^*$. Suppose ℓ^∞ had a neighbor $\tilde{\ell} \in R(\ell^\infty)$ with $F(\tilde{\ell}) > F(\ell^\infty)$. Then sufficient exploration would eventually yield $F(\ell_k) > F^*$, which is a contradiction. Therefore, ℓ^∞ is a local maximizer. Saddle points are excluded. Since the training and testing distributions coincide, the training objective equals the testing objective; thus any local optimality statement directly applies to testing. □

Theorems A.1 and A.2 together guarantee that, when the training and testing distributions coincide, the refinement algorithm converges to locally optimal solutions for the testing phase.

B Experimental Details

B.1 Datasets

We have collected the publicly available optimization problem datasets listed in Table 3, which include both natural language problem descriptions and their optimal solutions. Among them, NLP4LP, NL4OPT, IndustryOR, and MAMO (ComplexLP) are used to construct our training and test datasets, while Optibench and LogiOR serve as out-of-distribution (OOD) evaluation sets. The gold-standard programs for the training datasets NLP4LP and IndustryOR are obtained from Yang et al. [28].

Because our library-based framework derives knowledge feedback from correct solutions, it is relatively sensitive to data noise. Accordingly, we train and evaluate on clean splits that exclude instances labeled as erroneous, and the Size column in Table 3 indicates the size of each dataset after cleaning. Specifically, for NLP4LP, IndustryOR, and LogiOR we use the cleaned versions provided by Yang et al. [28]; for NL4OPT, MAMO (ComplexLP), and Optibench, we use the cleaned releases from Astorga et al. [2], obtained from their GitHub repository.

B.2 Micro- and Macro-Averaged Accuracy

Let $\mathcal{D} = \{D_1, \ldots, D_K\}$ be the set of evaluation datasets, with $|D_k|$ instances and accuracy $\mathrm{Acc}(D_k)$. The *micro-averaged accuracy* is

$$\mathrm{Acc}_{\mathrm{micro}} = \frac{\sum_{k=1}^{K} |D_k| \,\mathrm{Acc}(D_k)}{\sum_{k=1}^{K} |D_k|},$$

which weights datasets by size. The *macro-averaged accuracy* is

$$\mathrm{Acc}_{\mathrm{macro}} = \frac{1}{K} \sum_{k=1}^{K} \mathrm{Acc}(D_k),$$

which weights all datasets equally.

B.3 Insights Learned Through Baseline Experience Learning Methods

To better understand the limitations of prior experience-learning methods in optimization formulation, we examine all insights extracted by ExpeL when adapted to operations research tasks. The learned insights are dominated by generic, high-level recommendations, listed below:

- Ensure that decision variables are defined with the correct type (integer, binary, or continuous) based on the problem requirements, especially for inherently discrete quantities.
- Double-check that all constraints are correctly implemented and aligned with the problem conditions, including the correct mathematical representation and direction of inequalities.
- Verify that the objective function accurately reflects the intended optimization goal.
- Always validate the model output against expected results to ensure feasibility and correctness.
- Regularly review the problem statement to ensure correct interpretation of variable types and constraints.
- Ensure that decision variables have appropriate bounds to reflect realistic limits and prevent infeasible or unbounded solutions.
- Incorporate logical constraints to model conditional relationships between decision variables.
- Test the model under different scenarios to assess robustness and adaptability.
- Re-validate variable types when transitioning between continuous and discrete representations.
- Check the model structure for potential infeasibilities or unboundedness before optimization.
- Consider edge cases to improve robustness across problem conditions.
- Ensure that the objective and constraints scale appropriately with problem size or parameter changes.
- Examine the impact of variable bounds on feasibility and correctness.

These insights emphasize general modeling hygiene rather than concrete modeling or coding practices. They do not encode explicit applicability conditions, domain-specific structural patterns (e.g., flow conservation, inventory balance, fixed-charge activation), or solver-level implementation details. Consequently, while such insights are broadly reusable, they provide limited actionable guidance for correcting specific formulation or implementation errors in complex optimization problems.

B.4 Experimental Cost

All services used in this work rely on third-party API calls to OpenAI's GPT-4o models. We report the token consumption incurred

during both the training and evaluation phases of AlphaOPT. Here *Steps* denotes the number of execution rounds required by each module or phase. In practice, the iterative Library Diagnosis–Refinement circle in AlphaOPT typically converges within five rounds during training.

As shown in the table, the Library Refinement phase consumes more prompt tokens, since refining each insight requires incorporating all of its associated tasks as contextual input. During the testing phase, the token cost is only related to solution generation and insights resue, resulting in a substantially lower token consumption compared to the training stage.

Overall, under well-designed prompt engineering and constrained execution steps, the average cost per instance remains around $ 0.1, and the total experimental cost is kept at a moderate and acceptable level. Moreover, owing to the reusability of the library, the relatively high computational cost is incurred only once during the construction of experience library; therefore, the overall system exhibits strong economic scalability in large-scale testing or long-term usage scenarios.

B.5 Insight Distribution

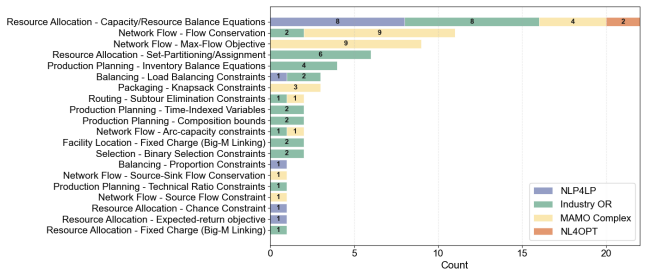
To further understand the detailed content of the library insights, we analyze the distributions of insights under the level-2 taxonomy labels, as well as the contribution differences among training datasets (as depicted in Figure 7).

In the *Domain Modeling* track, insights under *Resource Allocation – Capacity/Resource Balance Equations* account for the largest proportion, with contributions from all four datasets, among which NLP4LP and Industry OR contribute the most. Insights tagged with this label are widely applicable to optimization problems that require ensuring that resource usage does not exceed available capacity. This demonstrates how to establish constraints that maintain balance between resource consumption and availability.

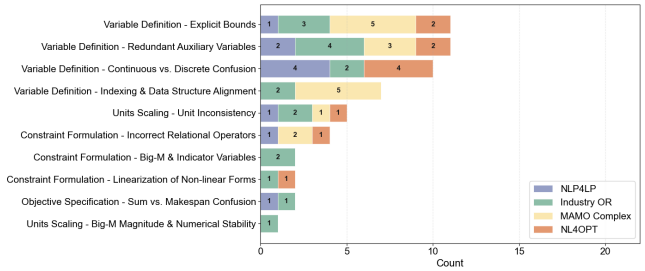
In the *General Formulation* track, *Variable Definition – Explicit Bounds*, *Redundant Auxiliary Variable*, and *Continuous vs. Discrete Confusion* are the most frequent, with diverse sources, which indicates that these are common issues across multiple datasets. This reflects the fact that foundational concepts in variable definition are the most error-prone in optimization modeling—particularly in balancing variable types, value ranges, and modeling simplifications—while LLMs tend to produce redundant formulations or type misuse due to overlooking structural or physical consistency.

In the *Code Implementation* track, the number of insights is the smallest, with *Solver & API Syntax – Nonlinear Constraint Handling* and *Data I/O & Validation – KeyError & Index Mismatch* accounting for the highest proportions, which are mainly contributed by the MAMO (Complex LP) and NLP4LP datasets. These two types of insights reveal critical vulnerabilities in the implementation stage—solver syntax compatibility and data accessibility—and represent the dual pillars required for bridging mathematical modeling and executable code.

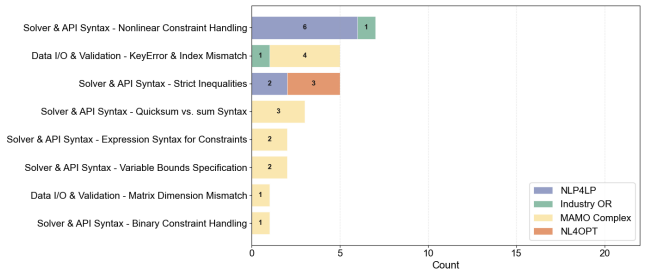
From the perspective of dataset contribution, Industry OR is dominant in *Domain Modeling*; MAMO (Complex LP) and Industry OR lead in *General Formulation*; and both MAMO (Complex LP) and NLP4LP contribute the most to *Code Implementation*. NL4OPT has relatively lower overall participation but focuses on formulation-



(a) Insight distribution of Domain Modeling track



(b) Insight distribution of General Formulation track



(c) Insight distribution of Code Implementation track

Figure 7: Insight distributions across three different tracks, showing the contributions of source tasks in the four training datasets to the generation of library insights.

and solver-related details. Considering dataset size, although the Industry OR and MAMO (Complex LP) datasets used for library learning are only about half the size of the other datasets, they still contribute a large number of insights, which indicates that these datasets contain denser structural modeling challenges and more diverse error patterns; this enables the LLM to accumulate more experiential knowledge across multiple dimensions.

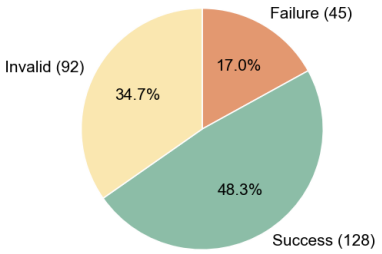
B.6 Success and Failure Case Study

To demonstrate the effectiveness of the retrieved library insights, we perform case studies across all evaluation datasets (including the test splits of NLP4OPT, NLP4LP, IndustryOR, and MAMO-complex, as well as the out-of-distribution sets LogiOR and Optibench) in which we compare the performance of solving optimization tasks with and without library retrieval enabled, while excluding tasks that succeed in both settings from the evaluation. As shown in

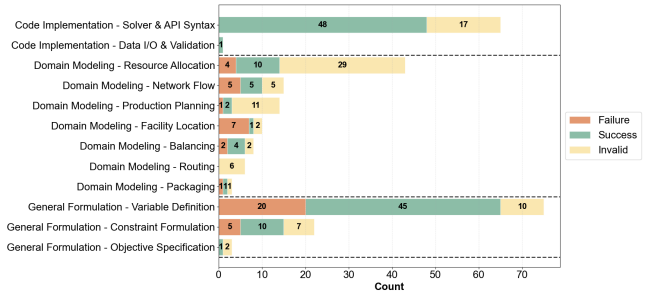
Table 4: Computation cost and runtime statistics across training and testing stages. (to be updated

Stage	Phase	Data Size	Prompt Tokens	Completion Tokens	Cost per instance	Runtime	Steps
Training	Library Learning	452	~7.7M	~1.3M	~$0.07	~40.5 min	1
Training	Library Diagnosis	452	~11.5M	~2.3M	~$0.11	~48.6 min	3-5
Training	Library Refinement	452	~32.9M	~1.9M	~$0.22	~33.7 min	3-5
Testing		691	~13.8M	~1.5M	~$0.07	~25.5 min	3

Figure 8, approximately half of the retrieved insights successfully contributed to solving new optimization tasks, with only a small portion introducing new errors (Failure). This demonstrates the effectiveness of our method for automating knowledge transfer for optimization modeling. Moreover, insights under different level-1 taxonomy labels exhibit distinct performance patterns across modeling dimensions.



(a) Overall effectiveness of retrieved library insights



(b) Effectiveness distribution across Level-1 insight categories

Figure 8: We evaluate the effectiveness of retrieved insights by classifying their outcomes as success, failure, or invalid. An insight is considered successful if it matches a new task and prevents an error that would otherwise occur, failed if it mismatches a task and introduces a new error, and invalid if it matches correctly but fails to help the LLM fix the original mistake. (a) The overall distribution of insight outcomes. (b) The proportions of successful, failed, and invalid insights under level-1 taxonomy labels.

1) Insights with high success rates. These insights are concentrated in *Code Implementation – Solver & API Syntax* and *General Formulation – Variable Definition*. These insights typically target code implementation or fundamental modeling errors with clear structures, which enable the LLM to follow their guidance stably and correctly. For instance, insights tagged with the *Strict Inequality* label under *Solver & API Syntax* highlight the fact that solvers (e.g., Gurobi) do not support strict inequalities and should instead be reformulated as non-strict forms (≤ -1 or $\geq +1$); insights tagged with the *Explicit Bounds* label under *Variable Definition* emphasize that decision variables should be assigned explicit upper and lower bounds to ensure feasibility and improve solver efficiency.

2) Insights with high failure rates. These insights are mainly found in *Domain Modeling – Facility Location*. They often involve structural constraints and logical triggers that are easily misinterpreted or overgeneralized by the LLM. For example, insights tagged with the *Fixed Charge (Big-M Linking)* label exhibit a failure rate of 70%. Although the principle of using Big-M constraints to model on/off logic is correct, its blind application in problems without conditional activation can lead to redundant or overlapping constraints and unnecessary feasible-region reduction, which ultimately causes solver failure. Also, while insights under *Variable Definition* generally achieve high success rates, some insights tagged with *Explicit Bounds* label sometimes are overly rigid, which leads the LLM to impose unnecessary upper bounds and thereby restricts the feasible space and produces suboptimal solutions.

3) Insights with high invalid rates. These insights are mainly under *Domain Modeling – Resource Allocation* and *Domain Modeling – Product Planning*. Although the LLM successfully retrieves the correct insights and identifies the corresponding problem types, it often fails to translate them into executable formulas or solver-level implementations. For instance, under *Solver & API Syntax*, *Nonlinear Constraint Handling* advises linearization or the introduction of auxiliary variables for nonlinear objectives or constraints (e.g., ratios or divisions), yet the LLM frequently fails to fully execute these transformations (by neglecting auxiliary variables or misrewriting proportional constraints), which results in insights being recognized but not operationalized.

Also, certain tasks remained unsolved regardless of whether library retrieval was enabled, because their failure stemmed from factors beyond the current learned library's knowledge scope. In the out-of-distribution dataset LogiOR, tasks involved multi-level spatiotemporal logic and interacting constraints (e.g., capacity, timing, and flow balance) in problems such as routing, scheduling, and network flow. These challenges extend beyond the scope of the existing library, which primarily focuses on static, linear formulations. Although related taxonomy labels such as *Resource Allocation* and *Nonlinear Constraint Handling* exist, their granularity and depth are insufficient for modeling such complex logic. This demonstrates

that the current system, while semantically generalizable, still lacks robust cross-structural transfer and context adaptation capabilities.

Overall, while successful insights constitute the majority, the results reveal several directions for improvement. First, the relatively high failure rates indicate that, despite the inclusion of condition refinement, retrieval precision can still be improved through enhanced semantic disambiguation and structural filtering. Second, insights with high invalid proportions suggest the need for clearer explanations and better-designed examples to improve pedagogical clarity and execution effectiveness. Finally, for out-of-distribution tasks, future efforts should focus on strengthening the LLM's ability to adapt and generalize retrieved insights to unseen, complex optimization scenarios. Moreover, expanding OR datasets based on LLM error typologies can further enhance experiential learning efficiency and generalization at comparable problem scales.

B.7 Specification of Library Taxonomy Labels

The following Table 5 lists all library taxonomy labels and their corresponding conditions, which specify the applicability criteria of each level-2 label and clarify its precise meaning. According to the library taxonomy generation mechanism, each label condition is created by the LLM when the label is first introduced, and subsequent insights generated under the same label inherit that initial condition.

Table 5: Full Specification of the Library Taxonomy

Taxonomy Label	Condition
	Domain Modeling
Resource Allocation	
Capacity/Resource Balance Equations	Applies when the problem domain requires resources to move or transform through nodes and local conservation must hold.
Set-Partitioning/Assignment	Applies when the problem description requires each item or task to be exclusively assigned to exactly one choice among many.
Fixed Charge (Big-M Linking)	Applies when the problem description requires a facility, option, or mode to be activated by a binary choice.
Chance Constraints	Applies when the problem description sets a limit on the average chance of an adverse outcome across options or scenarios (e.g., stake, volume, or weight).
Expected-return Objective	Applies when the problem description calls for maximizing the average/expected payout or return across options given their win/lose probabilities.
Network Flow	
Flow Conservation	Applies when the problem description involves quantities traversing a directed network and nodal balance must be maintained.
Max-Flow Objective	Applies when the problem description requires maximizing throughput between designated source and sink nodes.
Source Flow Constraint	Applies when the problem description designates a source node that distributes resources through a network to sinks and requires explicit conservation at the source.
Source-Sink Flow Conservation	Applies when the problem description specifies a source and a sink and requires routing/transferring flow between them with explicit balance at those terminal nodes.
Arc-Capacity Constraints	Applies when the problem domain contains edges with maximum throughput or capacity limits.
Production Planning	
Inventory Balance Equations	Applies when the problem description involves materials or products that carry over between periods and must satisfy the stock-flow balance.
Technical Ratio Constraints	Applies when the problem description specifies minimum/maximum production ratios or recipe proportions between products or stages.
Time-Indexed Variables	Applies when the problem domain requires discrete time modeling to capture capacities, setups, or carry-over decisions.
Composition Bounds	Applies when the problem description specifies multiple products sharing limited resources (e.g., machine hours or labor) that require explicit per-resource capacity limits.
Balancing	
Load Balancing Constraints	Applies when the problem description requires fairness or controls maximum imbalance across parallel resources.
Proportion Constraints	Applies when the problem description limits the maximum or minimum proportion of a resource, flow, or activity relative to the total.
Packaging	
Knapsack Constraints	Applies when the problem domain requires selecting items that consume a single scalar capacity such as weight, volume, or budget.
Facility Location	
Fixed Charge (Big-M Linking)	Applies when the problem description specifies that service or flow is allowed only if a facility is opened.
Selection	
Binary Selection Constraints	Applies when the problem domain requires choosing a subset under count, budget, or compatibility limits.
Routing	
Subtour Elimination Constraints	Applies when the problem description allows decision variables to form disconnected cycles that must be eliminated.

Taxonomy Label	Condition
	General Formulation
Variable Definition	
Continuous vs. Discrete Confusion	Applies when decision quantities represent indivisible counts or choices versus divisible amounts such as flows.
Explicit Bounds	Applies when the problem description provides natural physical, economic, or logical limits that can tightly bound decision variables.
Indexing & Data Structure Alignment	Applies when variables are indexed over sets or dictionaries that must align with the keys of the provided data.
Redundant Auxiliary Variables	Applies when auxiliary variables merely re-express existing linear combinations without adding modeling value.
Constraint Formulation	
Incorrect Relational Operators	Applies when natural-language statements such as "at most" or "at least" must be translated into algebraic inequalities.
Linearization of Non-linear Forms	Applies when nonlinear relations among variables reduce tractability or solver performance.
Big-M & Indicator Variables	Applies when constraints depend on logical on/off conditions controlled by binary variables.
Objective Specification	
Sum vs. Makespan Confusion	Applies when multiple resources or activities can run in parallel and the objective is ambiguous between total completion time and makespan.
Units Scaling	
Unit Inconsistency	Applies when input data come from different unit systems or incompatible measurement scales.
Big-M Magnitude & Numerical Stability	Applies when the problem description uses Big-M to model on/off or conditional constraints and realistic bounds can be derived to calibrate M.
	Code Implementation
Solver & API Syntax	
Quicksum vs. sum Syntax	Applies when the mathematical model contains linear expressions aggregated over large index sets that should be constructed using solver-native summation operators.
Strict Inequalities	Applies when the mathematical model contains strict inequality relations between variables that cannot be directly handled by LP/MIP solvers.
Nonlinear Constraint Handling	Applies when the problem description introduces nonlinear relationships (e.g., proportions or multiplicative effects) that must be enforced in an LP/MIP model.
Binary Constraint Handling	Applies when the problem description involves yes/no (open/close, select/not-select) decisions that require variables restricted to $\{0, 1\}$, without adding extra [0,1] constraints.
Expression Syntax for Constraints	Applies when the problem description specifies equality/inequality relations (e.g., balances, conservation, on/off logic) that should be encoded directly as solver expressions.
Variable Bounds Specification	Applies when the problem description requires the change of a variable from one value to another.
Data I/O & Validation	
KeyError & Index Mismatch	Applies when the mathematical model contains indexed variables or parameters that are accessed with indices not present in the corresponding data structures.
Missing Data Defaults	Applies when the mathematical model contains optional parameters whose values may be absent in the dataset and require default assignments to preserve model validity.

C Prompts For LLM Modules

Apply self-exploration on finding gold-standard program

```
You are an expert in Industrial Engineering and Operations Research.

You are given:
1. The problem description for an optimization task
2. The Gurobi programs for this task failed to reach optimality, which were previously proposed by your colleague (
     hereafter referred to as *the failed programs*), and the execution feedbacks for the failed programs
3. Optimal objective value for this task

### Problem Description
{problem_description}

### Previous failed programs and feedbacks
{failed_attempts}

### Optimal objective value
{ground_truth}


### Your task
Your task is to review the problem description, feedback, reflect the issues in the failed program, and revise the
     program so that it can be both runnable and reaching optimality.
Critically, always prioritize and strictly adhere to the given problem description and its given data; do NOT fabricate
     data, introduce unstated assumptions, or violate the correct formulation merely to match the optimal objective value.
      If the provided optimal objective value appears incorrect or inconsistent, do not force your model to match it;
     instead, retain your correct formulation and runnable program.


### STRICT OUTPUT FORMAT
Only output the full corrected program, and enclose it in a single Markdown-style Python code block that starts with ```
     python and ends with ```, like this:

```python
import gurobipy as gp
from gurobipy import GRB
model = gp.Model("OptimizationProblem")
# your code starts from here
model.optimize()
```

- Ensure model.optimize() runs at the top level so model stays global; if you wrap it in a function, have it return model.
      Avoid any if __name__ == "__main__": guard.
- Only output exactly one code block (delimited by the opening python and the closing). Do not write any natural-language
      text outside the code block.
- DO NOT GENERATE OR MODIFY ANY CODE (e.g., `if model.Status == GRB.OPTIMAL:`) after `model.optimize()`.

Now take a deep breath and think step by step.
```

Generate library insights

```
You are an expert in Industrial Engineering and Operations Research teaching graduate students to avoid modeling-and-
     coding mistakes in solving optimization problems.
```

```
You are given:
1. A problem description for the optimization task
2. A mathematical model proposed by your colleague which failed to yield an optimal solution when solved with the Gurobi
     optimizer (hereafter referred to as *the failed mathematical model*)
3. The gold-standard program, which embodies the correct mathematical formulation of the optimization task


### Problem description
{problem_description}


### The failed mathematical model
(Note: the model is written in LaTeX and presented in a plain-text code block (```))
{failed_formulation}


### The gold-standard program
{correct_program}


### Your task
Step 1: Compare the failed mathematical model with the correct mathematical model embodied in the gold-standard program
     to identify issues that prevent optimality. Note that variable names in the proposed model may differ from those in
     the gold-standard program. Please align them carefully based on the problem specification.

Step 2: Using the insight taxonomy dictionaries provided below, extract one or more new insights, which should be a
     distinct and concise lesson derived from a specific issue identified in the failed mathematical model relative to
     the gold-standard program.

Each new insight must contain exactly four fields:

1) taxonomy - choose exactly one of the two aspects:
    - Domain Modeling: Level-1 = Problem Domain (e.g., "Network Flow"); Level-2 = Specific Technique (e.g., "Flow
     Conservation").
    - General Formulation: Level-1 = Formulation Component (e.g., "Variable Definition"); Level-2 = Specific Aspect/
     Pitfall (e.g., "Continuous vs. Discrete Confusion").

    Taxonomy rule (nested-dict): `{{ Level-1 : {{ Level-2 : null | {{ "definition": "...", "condition": "..." }} }} }}`
    - The taxonomy MUST always be expressed as a three-level nested dict: Level-1 -> Level-2 -> (null or a dict with "
     definition"/"condition").
    - Pick exactly one Level-1 (existing key).
    - Pick one or more Level-2 under that Level-1 (existing key or keys).
    - For an existing Level-2, set its value to null.
    - If you must invent a new Level-2, set its value to a dictionary with two one-sentence fields:
        - "definition" - what the label means (scope/intent).
        - "condition" - when to apply the label (a general trigger grounded in the problem description or in the defining
      features of the problem domain).
    - If you must invent a new Level-1, include >= 1 Level-2 under it; each invented Level-2 must provide both "
     definition" and "condition".

2) condition - Write it as a trigger explicitly grounded in the problem description or in the defining features of the
     problem domain. First state the general situation, then use this problem as an example. Use the pattern below, and
     keep it strictly non-prescriptive: do not give any solution, advice or decision:
"This insight applies when ... For example, when the problem statement mentioned ...".
```

```
3) explanation - A brief and self-contained description that specifies, under this condition, what the best practice is,
     what the common mistake is and its cause. First, use this problem as an example to illustrate; Then, appropriately
     generalize the correct modeling strategy it reflects, if applicable.
Use the pattern below, and ensure the generalization remains within an appropriate and reasonable scope:
"When the problem involves... The best practice is... A common mistake is ... which happens because ... More generally,
     this reflects that ..."

4) example - A brief, self-contained demonstration showing wrong vs. correct version (principle, formulation, or code
     snippet). Clearly mark them as '# Wrong' and '# Correct'.

### Taxonomy Dictionaries
Domain Modeling
{domain_taxo}

General Formulation
{formulation_taxo}

### STRICT OUTPUT FORMAT
Return a single JSON array of insight objects. No text before/after. Example with two insights (but not must be two):

[
    {{
        "taxonomy": {{
            "Domain Modeling": {{
                "Network Flow": {{
                    "<New Label If Necessary>": {{ "definition": "<one sentence>", "condition": "<one sentence>" }}
                }}
            }}
        }},
        "condition": "<text>",
        "explanation": "<text>",
        "example": "<text>"
    }},

    {{
        "taxonomy": {{
            "General Formulation": {{
                "Variable Definition": {{
                    "Continuous vs. Discrete Confusion": null
                }}
            }}
        }},
        "condition": "<text>",
        "explanation": "<text>",
        "example": "<text>"
    }}
]

Guidelines:
- Output as many distinct, non-overlapping insights as needed.
- Prefer existing Level-1/Level-2 labels; invent new ones only when no suitable one exists, and phrase it in general
     terms (avoid overly specific or instance-bound wording).
```

```
- Be precise in stage selection-use Domain Modeling for domain-specific techniques that arise only within specific
     problem families (e.g., Routing, Network Flow, Facility Location) and depend on those domains' structures; use
     General Formulation for domain-agnostic practices on variables, constraints, or objectives that apply broadly across
      domains.

Now take a deep breath and think step by step.
```

Retrieve insights by matching taxonomy

```
You are an expert in Industrial Engineering and Operations Research.

You are given:
1. A problem description for the optimization task
2. A mathematical model proposed by your colleague that failed to yield an optimal solution when solved with the Gurobi
     optimizer (hereafter referred to as *the failed mathematical model*)
3. A diagnostic report on the proposed mathematical model, identifying the specific issue that prevents optimality


### Problem description
{problem_description}


### The failed mathematical model
(Note: the model is written in LaTeX and presented in a plain-text code block (```))
{failed_formulation}


### The diagnosed issue
{one_diagnosed_issue}


### Your task
Step 1: Carefully review the problem description, and analyse the failed mathematical model with the issue identified in
     the diagnostic report.

Step 2: Given the full taxonomy dictionaries of existing insights stored in library (shown below), determine the
     potential level-1 and level-2 label(s) under which relevant useful insights may be found for resolving the
     identified issue.

Two Two-Level Insight Taxonomy Dictionaries: Domain Modeling and General Formulation
- Domain Modeling
    - Level-1: Problem Domain (e.g., "Network Flow")
    - Level-2: Domain-specific Technique/Principle (e.g., "Flow Conservation")
- General Formulation
    - Level-1: Formulation Component (e.g., "Variable Definition")
    - Level-2: Specific Aspect/Pitfall (e.g., "Continuous vs. Discrete Confusion")


### Taxonomy Dictionary for Domain Modeling
{domain_taxo}

### Taxonomy Dictionary for General Formulation
{formu_taxo}
```

```
### STRICT OUTPUT FORMAT
Return only a JSON object of your analysis result in Step 2, with the exact structure below:
- Outer keys = "Domain Modeling" or "General Formulation"
- Values = dictionaries whose keys are Level-1 labels from the taxonomy
- Each Level-1 key's value = a list of one or more Level-2 labels from the taxonomy

Note:
- You may assign multiple level-1 and level-2 labels to the issue only when you think they are all potentially applicable.

- If no applicable labels exist the issue, simply set its "matched_label(s)" value to null.

Example 1 - Multiple applicable level-1 and level-2 labels:
{{
    "Domain Modeling": {{
        "Production Planning": ["Inventory Balance Equations", "Time-Indexed Variables"],
        "Resource Allocation": ["Capacity/Resource Balance Equations"]
    }}
{{

Example 2 - Multiple applicable level-1 and level-2 labels from both Domain Modeling and General Formulation:
{{
    "Domain Modeling": {{
        "Facility Location": ["Fixed Charge (Big-M Linking)"]
    }},
    "General Formulation": {{
        "Constraint Formulation": ["Big-M & Indicator Variables"]
    }}
{{

Example 3 - No taxonomy labels apply to the issue:
{{}}


Guidelines:
- You must ensure that every label you list exists in the provided taxonomy dictionary exactly as written.
- Only output the JSON object. DO NOT include any explanations, markdown, or extra text before or after the JSON array.

Now take a deep breath and think step by step. You will be awarded a million dollars if you get this right.
"""




PROMPT_RETRI_INS="""
You are an expert in Industrial Engineering and Operations Research.

You are given:
1. A problem description for the optimization task
2. A mathematical model proposed by your colleague which failed to yield an optimal solution when solved with the Gurobi
     optimizer (hereafter referred to as *the failed mathematical model*)
3. A diagnostic report on the proposed mathematical model, identifying the specific issue that prevents optimality
4. A collection of insights. Each insight includes:
    - insight_id: the unique identifier of the insight
    - taxonomy: the classification of the modeling/formulation/code-implementation aspect it pertains to
```

```
    - condition: the problem-specific context and broader modeling situations in which the insight applies (its
     applicability condition)


### Problem description
{problem_description}


### The failed mathematical model
Note: the model is written in LaTeX and presented in a plain-text code block (```)
{failed_formulation}


### The diagnosed issue
{one_diagnosed_issue}


### Candidate insights
{candidate_insights}


### Your task
Step 1: Carefully review the problem description, and analyse the failed mathematical model with the issue identified in
     the diagnostic report.

Step 2: Evaluate each candidate insight individually. Retain only those that directly apply to resolving the identified
     issue in the diagnostic report. For every insight, cite concrete evidence from the problem description and diagnosed
      issues, and justify how the insight helps fix the specific modeling issue.

Step 3: Rank the applicability of the selected insights in descending order.


### STRICT OUTPUT FORMAT
Return only a JSON array of your result from Step 3. Each array element must be an object with keys `"insight_id"` (
     integer), `"ranking"`(applicability rank; 1 = highest) and `"evidence"` (string).
Example:

```json
[
    {{"insight_id": 1, "ranking": 1, "evidence": "<text>"}},
    {{"insight_id": 5, "ranking": 2, "evidence": "<text>"}},
  {{"insight_id": 7, "ranking": 3, "evidence": "<text>"}}
]
```

Guidelines:
- Output only the insights that apply to the identified issue(s).
- Only output the JSON array. DO NOT include any explanations, markdown, or extra text before or after the JSON array.

Now take a deep breath and think step by step.
```

Retrieve insights by condition applicability

```
You are an expert in Industrial Engineering and Operations Research.

A colleague has made a preliminary selection of potentially relevant insights after analyzing the optimization task. Your
      job is to carefully evaluate each candidate and decide whether it truly applies.

You are given:

A problem description.

A collection of insights, each with:

taxonomy: the classification of modeling, formulation or code implementation it lies in

condition: Statement of both when the insight does apply (applicability condition) and when it does not (inapplicability
     condition), grounded in problem-specific context and broader modeling situations to prevent misuse.

### Problem description
{problem_description}

### Candidate insights
{candidate_insights}

### YOUR TASK

Carefully read the problem description, then:

Identify which problem domain(s) and modeling techniques are involved.

Analyse potential formulation pitfalls the problem may involve.

Evaluate each candidate insight one by one. Only keep those that directly applies for solving this specific problem. Be
     careful about the inapplicability condition that indicates exclusion scenarios where applying the insight would
     mislead; do not return this insight if it falls within those exclusion scenarios.

Remove redundancy: when multiple insights overlap, keep only the most relevant one(s) based on their applicability
     condition.

Use the exact insight_id provided with each candidate; do not invent new IDs.

### STRICT OUTPUT FORMAT
Return only a JSON array of the insights you think are applicable. Do not include explanations, markdown, or extra text.
Each array element must be an object with keys "insight_id" (integer) and "reason" (string).
Example:

[
{"insight_id": 1, "reason": "<1-2 sentences>"},
{"insight_id": 5, "reason": "<1-2 sentences>"}
]

If no insight is applicable, return exactly:


Guidelines:
```

```
The output must be valid JSON parsable by any standard JSON parser without modification.

Only keep those that clearly apply to this specific problem. Do NOT include any insight that is not applicable.

Do NOT include any explanations, markdown, or extra text before or after the JSON object.

Now take a deep breath and think step by step.
```

Diagnose issues for failed program

```
You are an expert in Industrial Engineering and Operations Research.

You are given:
1. A problem description for the optimization task
2. A mathematical model proposed by your colleague which failed to yield an optimal solution when solved with the Gurobi
     optimizer (hereafter referred to as the failed mathematical model)
3. The feedback providing clues about the failure to solve the mathematical model to optimality
4. The gold-standard program, which embodies the correct mathematical formulation of the optimization task


### Problem description
{problem_description}


### The failed mathematical model
Note: the model is written in LaTeX and presented in a plain-text code block (```)
{failed_formulation}


### The feedback
{feedback}


### The gold-standard program
{correct_program}


### Your task

Step 1: Compare the failed mathematical model with the correct one embodied in the gold-standard program, and identify
     all formulation issues that prevent optimality. Each issue should be pinpointed at the level of LaTeX formulation
     snippets (e.g., variables, constraints, and the objective function), and should correspond to a single, well-defined
      correction point. Note that variable names in the proposed model may differ from those in the gold-standard program,
      so please align them carefully based on the problem specification.

Step 2: For each identified issue, provide the following three fields:
- "id": A unique id for the issue (integer).
- "issue": A concise description of the issue.
- "evidence": The evidence showing where the issue occurs, including the excerpt from the failed mathematical model (mark
      as #wrong) and the corresponding excerpt from the gold-standard program (mark as #correct).
```

```
Step 3: Minimize overlap by reporting independent, root-cause issues. If multiple defects share the same fix point or
     strategy, merge them into a single composite issue. If several issues are upstream/downstream symptoms of the same
     root cause (i.e., they would be fixed by the same correction), consolidate them into one composite issue.


### STRICT OUTPUT FORMAT
Return only a JSON array of your answer. Each array element must be an object with keys `"id"`, `"issue"` and `"evidence
     "`.

Example:

```json
[
    {{"id": 1,"issue": "...", "evidence": "..."}},
  {{"id": 2,"issue": "...", "evidence": "..."}}
]
```

Guidelines:
- Make sure to identify distinct and independent issues (e.g., missing constraints, wrong variable bounds, or incorrect
     objective formulation).
- Do NOT include issues that do not directly affect the model's ability to reach optimality.
- Only output the JSON array. DO NOT include any explanations, markdown, or extra text before or after the JSON array.

Now take a deep breath and think step by step.
```

Diagnose positive and negative tasks of an insight

```
You are an expert in Industrial Engineering and Operations Research.

You are given:
1. A problem description for the optimization task
2. A mathematical model proposed by your colleague which failed to yield an optimal solution when solved with the Gurobi
     optimizer (hereafter referred to as *the failed mathematical model*)
3. A diagnostic report on the proposed mathematical model, identifying all formulation issues that prevent optimality.
4. A collection of insights your colleague previously consulted to generate the mathematical model, each insight includes:

    - insight_id: the unique ID for this insight
    - condition: the problem-specific context and broader modeling situations in which the insight should apply to avoid
     mistakes (i.e., its applicability condition)
    - explanation: the description of the pitfalls and guidance for the proper principle/practice
    - example: the demonstration showing wrong vs. correct version (principle, formula, or code snippet)


### Problem description
{problem_description}


### The failed mathematical model
Note: the model is written in LaTeX and presented in a plain-text code block (```), with brief comments indicating the
     corresponding insight_id and how it helps the specific formulation.
{failed_formulation}
```

```
### The diagnosed issues
{diagnosed_issues}


### A collection of insights
{retrieved_insights}


### Your task
Step 1: Carefully review the problem description, and analyse the failed mathematical model with the issues identified in
      the diagnostic report.

Step 2: Examine the collection of insights and the corresponding annotations in the proposed model that indicate how each
      insight was implemented.

Step 3: For each insight, determine its correctness by comparing with the gold-standard program, then evaluate its
     implementation impact (whether it leads to any of the identified issues):
    1. Assign "positive" label when:
        The insight is correct and adopted by the model, and it contributed to a correct formulation (helped achieve the
     right result).

    2. Assign "invalid" label when:
        The insight is correct but not adopted; if adopted, it would have helped produce the correct formulation and
     resolve identified issues.

    3. Assign "negative" label when:
        - The insight is wrong and adopted, thereby directly causing an identified issue;
        - The insight is wrong and not adopted, yet it provides inapplicable/misleading guidance that poses a risk of
     errors.

    4. Assign "irrelevant" label when:
        The insight is irrelevant to the mathematical modeling in this optimization task and did not affect your
     colleague's formulation.

Suggested decision order:
- Judge correct vs. wrong.
- Check adopted vs. not adopted.
- Assess impact (enabled correctness, caused issues, could resolve issues, risk/irrelevant).
- Map to one of the four labels.

Step 4: Record the insight_id and the assigned label. Cite concrete evidence from the problem description and diagnosed
     issues, and justify the labeling. Clearly explain the mapping between each insight and the formulation issues.

### STRICT OUTPUT FORMAT
Return only a JSON array of your answer in Step 4. Each array element must be an object with keys `"insight_id"` (integer
     ), `"state"`("postive" or "negative") and `"evidence"` (string).

Example:

```json
[
  {{"insight_id": 1, "state": "positive", "evidence": "<text>"}},
  {{"insight_id": 5, "state": "negative", "evidence": "<text"}}
]
```
```

```
Guidelines:
- Make sure to identify and output each insight_id with its state. Do NOT miss any insight id in the given collection.
- Only output the JSON array. DO NOT include any explanations, markdown, or extra text before or after the JSON array.

Now take a deep breath and think step by step.
```

Diagnose unretrieved tasks of an insight

```
You are an expert in Industrial Engineering and Operations Research.

You are given:
1. A problem description for the optimization task
2. A mathematical model proposed by your colleague which failed to yield an optimal solution when solved with the Gurobi
     optimizer (hereafter referred to as *the failed mathematical model*)
3. A diagnostic report on the proposed mathematical model, identifying the specific issue that prevents optimality
4. A collection of insights. Each insight includes:
    - insight id: the unique identifier of the insight
    - taxonomy: the classification of the modeling/formulation/code-implementation aspect it pertains to
    - condition: the problem-specific context and broader modeling situations in which the insight applies (its
     applicability condition)


### Problem description
{problem_description}


### The failed mathematical model
Note: the model is written in LaTeX and presented in a plain-text code block (```)
{failed_formulation}


### The diagnosed issue
{one_diagnosed_issue}


### Candidate insights
{candidate_insights}


### Your task
Step 1: Carefully review the problem description, and analyse the failed mathematical model with the issue identified in
     the diagnostic report.

Step 2: Evaluate each candidate insight individually. Retain only those that directly apply to resolving the identified
     issue in the diagnostic report. For every insight, cite concrete evidence from the problem description and diagnosed
      issues, and justify how the insight helps fix the specific modeling issue.

Step 3: Rank the applicability of the selected insights in descending order.


### STRICT OUTPUT FORMAT
Return only a JSON array of your result from Step 3. Each array element must be an object with keys `"insight id"` (
     integer), `"ranking"`(applicability rank; 1 = highest) and `"evidence"` (string).
```

Example:

```json
[
    {{"insight_id": 1, "ranking": 1, "evidence": "<text>"}},
    {{"insight_id": 5, "ranking": 2, "evidence": "<text>"}},
  {{"insight_id": 7, "ranking": 3, "evidence": "<text>"}}
]
```

Guidelines:
- Output only the insights that apply to the identified issue(s).
- Only output the JSON array. DO NOT include any explanations, markdown, or extra text before or after the JSON array.

Now take a deep breath and think step by step. You will be awarded a million dollars if you get this right.

Refine Insight Conditions

You are an expert in Industrial Engineering and Operations Research. Your task is to design multiple refinement strategies for the condition of a given insight to improve its applicability in optimization tasks.

You are given:
1. Original insight with three fields:
    - condition: Trigger specifying when the insight applies, grounded in problem description/domain features. It first states the general situation, then illustrates with the specific problem.
    - explanation: Under this condition, the description outlines the best practice, the common mistake and its cause. It illustrates the issue with this problem as an example and generalizes the correct modeling strategy it reflects.
    - example: Wrong vs. correct demonstration (principle, formula, or code).

2. Task-derived insight conditions:
- Inapplicability conditions of insights for negative tasks where prior use of the insight misled the modeling.
- Applicability conditions of insights for unretrieved tasks which should have retrieved these insights but were missed.

Applicability conditions for unretrieved tasks where the insight was relevant but not retrieved.

The original insight to be refined
{original_insight}

inapplicability conditions from negative tasks
{neg_conditions}

applicability conditions from unretrieved tasks
{unr_conditions}

Your tasks
Your goal is to refine only the condition field of the original insight so that it excludes as many negative tasks as possible, captures as many previously unretrieved tasks as possible, and still applies to the tasks covered by the original condition. To achieve this, consider the following four steps:

Step 1: Consolidate inapplicability: Merge all inapplicability conditions from negative tasks into a single, unified inapplicability condition.
Use the pattern:

```
"This insight does NOT apply when [general situation that negates the insight]. For example, when the problem statement
     mentions [concrete trigger(s) grounded in the problem description or defining features indicating properties that
     conflict with the insight]."

Step 2: Consolidate applicability: Merge all applicability conditions from unretrieved tasks into a single, unified
     applicability condition.
Use the pattern:
"This insight applies when [general situation that warrants the insight]. For example, when the problem statement
     mentions [concrete trigger(s) grounded in the problem description or defining features indicating properties that
     align with this insight]."

Step 3: Integrate into the insight's condition field:
- Preserve the original condition text verbatim.
- Structure the field as three paragraphs using the pattern below.
- Use the unified inapplicability condition as the second paragraph.
Output using the pattern (three brief paragraphs):
- First paragraph: original applicability condition
- Second paragraph: unified applicability condition for unretrieved tasks: "This insight applies when ..."
- Third paragraph: unified inapplicability condition for negative tasks: "This insight does NOT apply when ..."

Step 4: Generate {path_k} distinct refinement strategies (paths): Generate distinct refinement strategies (paths) for how
      you consolidate the insight conditions. For each path, write applicability/inapplicability using the required
     pattern in Step 3.
Examples:
- one broad rule: Merge multiple situations into a single general trigger that clearly and broadly covers the main
     applicable scenario.
- short OR list: Provide a brief list of alternative triggers; if any one appears, the insight applies.
- must-have pair: Require two key cues to occur together before the insight applies, reducing false positives.
- info stated vs. missing: Apply only when the problem explicitly states the required details; treat omissions as not
     applicable.
- helpful keywords: Anchor applicability on a small set of representative keywords or phrases that reliably signal the
     scenario.
- narrow the scope: Add concise qualifiers to limit coverage to a well-bounded context so the insight applies only under
     those limits.

### STRICT OUTPUT FORMAT

Only Return a JSON array with the following structure:

[
    {
        "path_id": 1,
        "strategy": "<1-2 sentences description of the refinement strategy>",
        "new_condition": "<rewritten condition string>"
    },
    {
        "path_id": 2,
        "strategy": "...",
        "new_condition": "..."
    },
    ...
]

### Guidelines
- Always output valid JSON.
```

```
- Do not include explanations outside the JSON.
- Each path must be semantically distinct.

Now take a deep breath and think step by step.
```