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

Minwei Kong*
London School of Economics and Political Science

Ao Qu†
Massachusetts Institute of Technology

Xiaotong Guo
Massachusetts Institute of Technology

Wenbin Ouyang
Massachusetts Institute of Technology

Chonghe Jiang
Massachusetts Institute of Technology

Han Zheng
Massachusetts Institute of Technology

Yining Ma
Massachusetts Institute of Technology

Dingyi Zhuang
Massachusetts Institute of Technology

Yuhan Tang
Massachusetts Institute of Technology

Junyi Li
Singapore-MIT Alliance for Research and Technology

Hai Wang
Singapore Management University

Cathy Wu
Massachusetts Institute of Technology

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

Abstract

Optimization modeling enables critical decisions across industries but remains hard to automate: informal language must be mapped to precise mathematical formulations and executable solver code, while prior LLM approaches either rely on brittle prompting or costly retraining with limited generalization. We present **AlphaOPT**, a self-improving *experience library* that enables an LLM to learn from limited demonstrations (i.e, even answers alone without gold-standard program) and solver feedback without annotated reasoning traces or parameter updates. AlphaOPT operates a continual two-phase cycle: (i) a *Library Learning* phase that reflects on failed attempts, extracts solver-verified, structured insights as $\{$*taxonomy*, *condition*, *explanation*, *example*$\}$; and (ii) a *Library Evolution* phase that diagnoses retrieval misalignments and refines the applicability conditions of stored insights, improving transfer across tasks. This design (1) learns efficiently from limited demonstrations without curated rationales, (2) expands continually without costly retraining by updating the library rather than model weights, and (3)

*Equal contribution.

†Correspondence: qua@mit.edu

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: ScaleOPT: GPU-Accelerated and Scalable Optimization.

makes knowledge explicit and interpretable for human inspection and intervention. Experiments show that AlphaOPT steadily improves with more data (65% → 72% from 100 to 300 training items) and surpasses the strongest baseline by 7.7% on the out-of-distribution OptiBench dataset when trained only on answers. AlphaOPT code and data are available at https://github.com/Minw913/AlphaOPT.

1 INTRODUCTION

Optimization models support critical decision-making in finance, manufacturing, marketing, transportation, and logistics (AhmadiTeshnizi et al., 2023; Bertsimas & Tsitsiklis, 1997; Ramamonjison et al., 2022). Beyond improving efficiency, automating the optimization workflow lowers the barrier to operations research expertise in industry, enabling non-experts to prototype faster, iterate on formulations, and deploy solver-backed decisions at scale. Yet this process has long been challenging, as informal and often ambiguous specifications must be mapped to precise, domain-specific formulations and paired with appropriate code and solvers, creating major bottlenecks for end-to-end automation (Jiang et al., 2025).

Advances in large language models (LLMs) make this vision increasingly feasible: they can parse natural language requirements (Ouyang et al., 2022), generate executable programs (Nijkamp et al., 2022; Jimenez et al., 2024), and orchestrate downstream tools (Qin et al., 2024). Two main lines of work have emerged. Prompt-based systems steer general LLMs with structured prompts and tool use (Xiao et al., 2023; Thind et al., 2025; AhmadiTeshnizi et al., 2024; Zhang & Luo, 2025). Fine-tuning approaches adapt models on domain corpora and benchmarks (Huang et al., 2025; Yang et al., 2024). Despite this progress, both families face limitations: prompt-based systems stop improving once they run out of fixed templates, and they are fragile to small wording changes and shifts in the domain; fine-tuned models require costly retraining and, critically, most benchmarks and datasets in the community (e.g., NLP4LP (AhmadiTeshnizi et al., 2024), MAMO (Huang et al., 2024), IndustryOR (Huang et al., 2025)) contain only programs/solutions rather than the intermediate reasoning that governs modeling choices, thereby limiting 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.

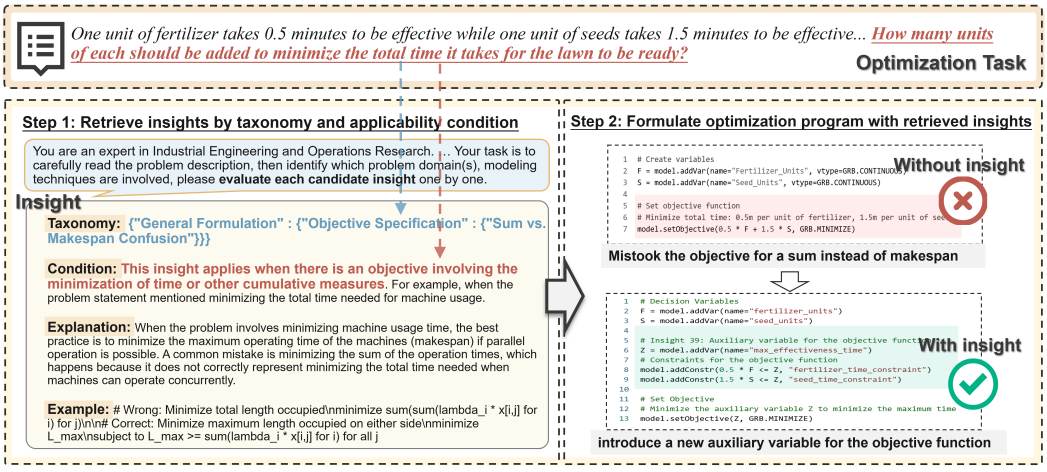


Figure 1: AlphaOPT identifies and retrieves relevant insights to guide problem solving. In this example, it avoids the common mistake of minimizing the sum of process times and instead introduces an auxiliary variable to correctly minimize the makespan, leading to the correct solution.

We propose **AlphaOPT**, a self-improving framework that builds and refines a structured library of solver-verified insights for optimization formulation, as exemplified in Figure 1. Each insight encodes a reusable modeling rule in the form of a 4-tuple (*taxonomy*, *(applicability) condition*, *explanation*, *example*), which specifies not only what to reuse but also when and why it applies. We remark that

our library learning framework does not require backpropagation to update framework parameters and can be regarded as the evolutionary mechanism. **AlphaOPT** improves through a continual two-phase cycle. Library Learning acquires new insights from both gold programs (when available) and solver-verified answer-only supervision, organizing them into a dynamically updating hierarchical taxonomy. Library Evolution then diagnoses misalignments between tasks and insight applicability, and refines conditions using aggregate evidence, ensuring that insights remain neither too narrow nor overly general. This design yields a principled optimization view: library construction corresponds to maximizing expected task success induced by task–insight matching while regularizing size to maintain efficiency and prevent redundancy.

We conduct quantitative experiments across multiple benchmarks and baselines, as well as qualitative analyses of the learned library. The results show that, compared to prompt-based or fine-tuning approaches, **AlphaOPT** (1) learns efficiently from limited demonstrations (i.e., it can learn from answers without recalling formulation) without requiring annotated reasoning traces or even gold-standard programs, (2) achieves stronger out-of-distribution generalizability and more consistent continual growth than learning-based methods, and (3) makes knowledge explicit and interpretable for human inspection and involvement.

Beyond these advantages, **AlphaOPT** also achieves state-of-the-art performance on multiple benchmarks. These results demonstrate the efficacy and potential of self-improving experience-library learning for optimization formulation, paving the way toward more challenging settings, such as efficient program formulation and large-scale optimization.

2 Methodology

Optimization tasks arrive with diverse natural-language descriptions, yet they share recurring modeling rules that activate under identifiable conditions. We identify three major challenges in building reliable systems that leverage LLMs to formulate and solve optimization problems using existing technologies and resources. First, gold-standard programs are scarce and may contain annotation errors (Jiang et al., 2025; Yang et al., 2025a), while datasets with only answer labels remain underutilized (Huang et al., 2024, 2025; Lu et al., 2025). Second, fine-tuned models (Huang et al., 2025; Jiang et al., 2025) struggle to generalize because the crucial when-to-apply-what knowledge is weakly represented (or missing) in training data; they can mimic syntax without mastering applicability. Third, the performance of prompt-based agent systems AhmadiTeshnizi et al. (2023); Xiao et al. (2023); Yang et al. (2025a) stagnates as the number of exemplars increases: they rely on human empirical curation and lack the capacity to adapt or to continually learn from larger datasets.

We propose AlphaOPT, an experience-library learning framework that learns from both gold programs (when available) and answer-only supervision. AlphaOPT iteratively builds a structured, solver-verified repository of reusable insights with explicit applicability conditions and evolves these conditions at the population level to improve generalization while avoiding redundancy. Specifically, as illustrated in Figure 2, each iteration consists of two complementary phases that form a continual cycle of acquisition and refinement:

Library Learning. The first phase extracts insights from individual tasks under either gold-program or answer-only supervision while minimizing redundancy.

Library Evolution. The second phase diagnoses misalignments between insights and tasks and refines applicability conditions to enhance generalization while reducing confusion caused by over-generalization.

The design follows three guiding principles: it is failure-driven (every error becomes a learning opportunity), locally validated (an insight must solve its source task before being admitted), and compact yet generalizable (redundant insights are merged and conditions refined to prevent uncontrolled growth that hinders retrieval and execution). We elaborated the specific methodological components and provide a mathematical interpretation that frames library construction as maximizing task success with a size regularizer in Appendix B. Additionally, we compare our method with prior works on learning from experience and self-evolving problem-solving agents in Appendix C. The prompts for all LLM modules are provided in Appendix H.

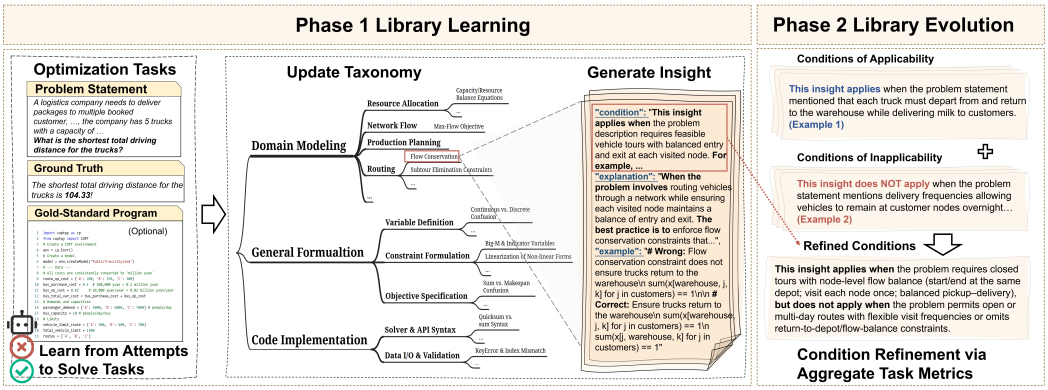


Figure 2: AlphaOPT consists of a library learning stage, which produces insights from attempts to solve tasks, and a library evolution stage, which adjusts the applicability of insights to avoid being too narrow or too general.

3 EXPERIMENTS

Our experiments are designed to reflect the requirements that arise in real-world optimization and operations research (OR) applications. In these settings, methods are expected not only to perform well on standard benchmarks, but also to transfer across domains, to remain effective when limited supervision is available, to improve steadily as more data becomes available, and to offer results that can be inspected and audited. We therefore organize our evaluation around four questions: (1) How well does the method generalize across domains? (2) Can it learn effectively with limited supervision? (3) Does performance improve consistently with more training data? (4) How does it compare overall with strong baselines? Finally, we examine the interpretability of the insight library to assess whether the outputs are understandable and actionable to practitioners.

3.1 Experimental Setup

Datasets. We conduct our experiments on a dataset consisting of 454 problem instances. Detailed descriptions of these datasets and the sampling procedure are provided in Appendix D.

Backbone Model and Metrics. 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. (2025a), where a task is considered successful if the LLM-generated optimal value closely aligns with the provided ground-truth solution.

Baselines. We evaluate against two families of baselines. (i) **Prompt-based**: a vanilla baseline that directly generates the mathematical model from a simple prompt, as well as Reflexion (Shinn et al., 2023), OptiMUS (AhmadiTeshnizi, Gao, and Udell, 2024), and ORThought (Yang et al., 2025a). (ii) **Learning-based**: ORLM (Huang et al., 2025), built on LLaMa3-8B, and LLMOPT (Jiang et al., 2025), built on Qwen2.5-14B (the latest open-source version available after their paper).

3.2 Out-of-Distribution Generalization

We evaluate how well different methods generalize beyond their training distribution. For this purpose, we use two benchmarks that were not included during training: LogiOR (Yang et al., 2025a) and OptiBench (Yang et al., 2024). Details are provided in Appendix D. These datasets were either released after the baseline model (ORLM) or explicitly designed in baseline model's experiment setting to avoid overlap with their training set (LLMOPT).

Figure 3 summarizes the results. Fine-tuned models such as ORLM and LLMOPT show strong in-distribution performance but exhibit a noticeable drop on unseen datasets. For example, ORLM falls to 19.6% on LogiOR and 13.3% on OptMath, while LLMOPT performs better but still degrades compared to its in-distribution performance. By contrast, AlphaOPT maintains higher accuracy across all three benchmarks, reaching 51.1% on LogiOR and 91.8% on OptiBench. These results

support our hypothesis: fine-tuned models tend to learn the syntax of solutions and may perform well when problems are very similar, but they struggle to capture the underlying principles needed for broader problem solving. In contrast, the learned experience library equips **AlphaOPT** with stronger out-of-distribution generalization capability.

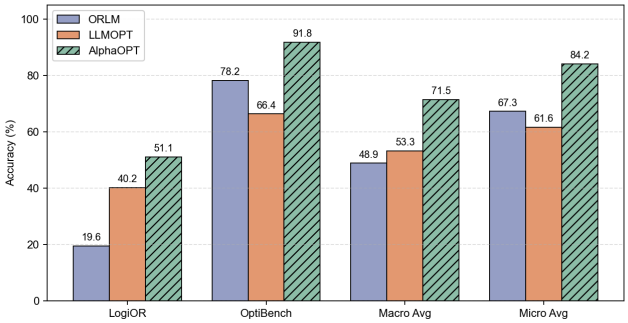


Figure 3: Performance on out-of-distribution datasets. Numbers report test accuracy on LogiOR, OptiBench, and OptMath.

3.3 Learning with Limited Supervision

In practical OR applications, gold-standard programs are rarely available. We therefore test whether AlphaOPT can learn solely from answers. Since two datasets in our training set contain gold-standard programs, we remove them in this experiment and let **AlphaOPT** learn exclusively from answer labels through self-exploration, as introduced in Section **??**. As shown in the last two rows of Table 2, remarkably, when trained with answer-only supervision, **AlphaOPT** achieves accuracy comparable to when it is trained with gold-standard programs. **AlphaOPT** (self-exploration) outperforms all prompt-based methods on test splits of the training data and even achieves the best performance on the OOD OptiBench dataset (92.1% accuracy). This demonstrates another advantage of **AlphaOPT** over fine-tuning–based methods, which require detailed annotations of mathematical formulations and code in order to achieve strong performance.

3.4 Continual Growth with Data

We test whether **AlphaOPT** can improve its performance as more data becomes 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 1, 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.

Table 1: AlphaOPT steadily improves in both Micro and Macro averages with increasing training size, while maintaining a compact library.

Training Size	MicroAvg	MacroAvg	Library Size
100	83.24%	65.80%	38
200	85.09%	69.22%	103
300	85.21%	72.12%	110

3.5 Overall Performance

AlphaOPT achieves the best accuracy on out-of-distribution datasets, reaching 51.1% on LogiOR and 91.8% on OptiBench. On in-distribution test splits, fine-tuned models such as ORLM and LLMOPT achieve higher scores on certain datasets (e.g., LLMOPT obtains 97.3% on NLP4LP and 85.8% on MAMO). However, these advantages are less conclusive, since LLMOPT's training data are not publicly available and may overlap with our test splits. Moreover, many existing benchmarks are derived from a small set of seed problems (Ramamonjison et al., 2022; Huang et al., 2024), which favors fine-tuning approaches that excel at pattern memorization. In contrast, **AlphaOPT** performs

competitively across all in-distribution datasets, matches or exceeds baselines on IndustryOR and MAMO (ComplexLP), and maintains a clear margin on out-of-distribution generalization. These results demonstrate that the experience library enables **AlphaOPT** to learn transferable modeling principles rather than dataset-specific syntax, resulting in stronger robustness to distribution shifts. Additionally, we provide the results of the ablation study in Appendix G.1 to assess the effectiveness of separate components of our framework.

Table 2: Accuracy on in-distribution *Test Split* and *Out-of-Distribution* datasets (higher is better). Best per column in **bold**.

Method		Test Split				Out-of-Distribution	
		NLP4LP (73)	NL4OPT (64)	IndustryOR (25)	MAMO (ComplexLP) (34)	LogiOR (92)	OptiBench (403)
Prompt-based	Standard	68.5	54.7	52.0	44.1	46.7	72.7
	Reflexion	76.7	64.1	56.0	47.1	43.5	76.9
	OptiMus	71.2	73.4	36.0	29.4	17.4	74.7
	ORThought	69.9	75.0	**60.0**	41.2	44.6	84.4
Fine-tuning-based	ORLM	86.3	**87.5**	36.0	55.9	19.6	78.2
	LLMOPT	**97.3**	86.5	44.0	**85.8**	40.2	66.4
Ours	AlphaOPT (full)	83.6	79.7	**60.0**	85.3	**51.1**	91.8
	AlphaOPT (self-exploration)	86.3	79.7	**60.0**	76.5	50.0	**92.1**

4 Library Analysis

To fully interpret the experience library, we visualize its structure and the distribution of insights under a hierarchical taxonomy in the Appendix F.1 and F.2. Additionally, we select several representative insights to analyze the effectiveness of library refinement, which are provided in the Appendix F.3.

5 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 and the out-of-distribution sets), comparing the performance of solving optimization tasks with and without library retrieval enabled, while excluding tasks that succeed in both settings from the evaluation. The details are provided in the Appendix G.2.

6 Conclusion

This paper addresses the limitations of previous methods by presenting a novel self-improving library learning framework, **AlphaOPT**, for formulating optimization programs. It 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 across domain-specific modeling, mathematical formulation, and solver syntax handling. Case studies show that high-success-rate insights primarily address fundamental modeling errors with clear structures, while high-mismatch or low-effectiveness insights expose overgeneralization in more complex knowledge and highlight the need to further improve the clarity and applicability of insights.

Looking ahead, we highlight three promising directions. First, reasoning-oriented test-time scaling—already powerful in other domains—could be particularly effective for OR formulations, where results are inherently verifiable. Second, strengthening datasets with both academic and large-scale industrial problems will move LLM systems beyond the toy examples common in current benchmarks. Third, going beyond correctness toward improving formulation efficiency is crucial for real-world deployment, and our self-improving library learning approach offers a promising path in that direction.

References

Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. Optimus: Optimization modeling using mip solvers and large language models. *arXiv preprint arXiv:2310.06116*, 2023.

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

Nicolás Astorga, Tennison Liu, Yuanzhang Xiao, and Mihaela van der Schaar. Autoformulation of mathematical optimization models using llms, 2025. URL https://arxiv.org/abs/2411.01679.

Dimitris Bertsimas and John N Tsitsiklis. *Introduction to linear optimization*, volume 6. Athena scientific Belmont, MA, 1997.

Erhu Feng, Wenbo Zhou, Zibin Liu, Le Chen, Yunpeng Dong, Cheng Zhang, Yisheng Zhao, Dong Du, Zhichao Hua, Yubin Xia, et al. Get experience from practice: Llm agents with record & replay. *arXiv preprint arXiv:2505.17716*, 2025.

Arya Grayeli, Atharva Sehgal, Omar Costilla Reyes, Miles Cranmer, and Swarat Chaudhuri. Symbolic regression with a learned concept library. *Advances in Neural Information Processing Systems*, 37: 44678–44709, 2024.

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

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

Caigao Jiang, Xiang Shu, Hong Qian, Xingyu Lu, Jun Zhou, Aimin Zhou, and Yang Yu. Llmopt: Learning to define and solve general optimization problems from scratch. In *Proceedings of the Thirteenth International Conference on Learning Representations (ICLR)*, Singapore, Singapore, 2025. URL https://openreview.net/pdf?id=9OMvtboTJg.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

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

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

Fangwen Mu, Junjie Wang, Lin Shi, Song Wang, Shoubin Li, and Qing Wang. Experepair: Dual-memory enhanced llm-based repository-level program repair. *arXiv preprint arXiv:2506.10484*, 2025.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*, 2022.

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. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

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

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. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *ICLR*, 2024. URL https://openreview.net/forum?id=dHng2O0Jjr.

Rindranirina Ramamonjison, Timothy Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong Zhang. Nl4opt competition: Formulating optimization problems based on their natural language descriptions. In Marco Ciccone, Gustavo Stolovitzky, and Jacob Albrecht (eds.), *Proceedings of the NeurIPS 2022 Competitions Track*, volume 220 of *Proceedings of Machine Learning Research*, pp. 189–203. PMLR, 28 Nov–09 Dec 2022. URL https://proceedings.mlr.press/v220/ramamonjison23a.html.

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

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

Raghav Thind, Youran Sun, Ling Liang, and Haizhao Yang. Optimai: Optimization from natural language using llm-powered ai agents. *arXiv preprint arXiv:2504.16918*, 2025.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Forty-second International Conference on Machine Learning*, 2024.

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

Beinuo Yang, Qishen Zhou, Junyi Li, Chenxing Su, and Simon Hu. Automated optimization modeling through expert-guided large language model reasoning, 2025a. URL https://arxiv.org/abs/2508.14410.

Xianliang Yang, Ling Zhang, Haolong Qian, Lei Song, and Jiang Bian. Heuragenix: Leveraging llms for solving complex combinatorial optimization challenges. *arXiv preprint arXiv:2506.15196*, 2025b.

Zhicheng Yang, Yiwei Wang, Yinya Huang, Zhijiang Guo, Wei Shi, Xiongwei Han, Liang Feng, Linqi Song, Xiaodan Liang, and Jing Tang. Optibench meets resocratic: Measure and improve llms for optimization modeling. *arXiv preprint arXiv:2407.09887*, 2024.

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

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

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

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

Zhaocheng Zhu, Yuan Xue, Xinyun Chen, Denny Zhou, Jian Tang, Dale Schuurmans, and Hanjun Dai. Large language models can learn rules. *arXiv preprint arXiv:2310.07064*, 2023.

Appendix

A Related Work

LLMs for Solving Optimization Problems. Related work can be categorized into prompt-based and learning-based approaches. Prompt-based methods guide reasoning and modeling through multi-step prompts using proprietary LLMs (AhmadiTeshnizi et al., 2024; Xiao et al., 2023). AhmadiTeshnizi et al. (2023) first introduced OptiMUS, demonstrating how LLMs can generate optimization formulations from natural language descriptions, and OptiMUS-0.3 (AhmadiTeshnizi et al., 2024) extends this line of work to large-scale instances, introducing retrieval-augmented prompting and solver-integrated verification to improve scalability.

In contrast, learning-based methods construct training datasets and apply instruction tuning to open-source LLMs. Huang et al. (2025) proposed a semi-automated pipeline to synthesize training data, which is then used to fine-tune an open-source ORLM model. LLMOPT (Jiang et al., 2025) combines both paradigms by modeling optimization with five elements and fine-tuning on expert-annotated data via multi-instruction SFT. More recently, ORThought (Yang et al., 2025a) introduced the LogiOR benchmark and an expert-guided chain-of-thought framework, providing a systematic dataset and evaluation pipeline for optimization tasks that require harder logic. In terms of multi-agent design, Xiao et al. (2023) employs a collaborative multi-expert framework to enhance reasoning, Zhang & Luo (2025) integrates sandbox-based code execution and self-repair/self-verification.

Several benchmarks now exist that cover LP, MILP, NLP, and other optimization problem types (Xiao et al., 2023; AhmadiTeshnizi et al., 2024; Huang et al., 2025; Yang et al., 2024). Yet, none of the prior work has investigated strengthening LLMs' optimization capabilities by *learning and reusing structured modeling experience*.

Decision-making tasks with Library Learning. Library Learning refers to the process where reusable patterns or modules are automatically extracted from past experiences to improve future problem-solving. These experiences include concrete trajectories or demonstrations, as well as abstracted rules generalized from successful or failed attempts (Zhao et al., 2024; Mu et al., 2025; Feng et al., 2025; Wang et al., 2024; Zhu et al., 2023). In terms of experience improvement, Zhao et al. (2024) and Mu et al. (2025) leverage an LLM to prune the library by checking if a newly added insight duplicates or conflicts with existing insights, or merges and generalizes from those overlapping insights. Feng et al. (2025) uses check functions to ensure that LLM-translated action sequences remain within the generalization boundaries of the original experiences.

LLM-driven Evolutionary Methods. Recent LLM-driven evolutionary frameworks have achieved remarkable advances in scientific discovery, showcasing LLM's capacity for broad generative exploration on solutions. Romera-Paredes et al. (2024) introduces FunSearch, a genetic programming driven by LLMs to search for feasible or optimal solutions of mathematical problems. AlphaEvolve (Novikov et al., 2025) extends the FunSearch system to provide the ability to perform multiobjective optimization using rich forms of natural-language context and feedback. Grayeli et al. (2024) applies LLMs to discover abstract concepts from high-performing hypotheses, combining symbolic regression with LLM-guided exploration within a concept library. ReEvo (Ye et al., 2024) frames LLMs as hyper-heuristics with a reflective evolution mechanism, enabling the generation and iterative refinement of heuristics across multiple NP-hard problems. HeurAgenix (Yang et al., 2025b) further develops this direction by evolving a pool of heuristics and dynamically selecting the most suitable one for each problem state, highlighting LLMs' role in adaptive heuristic discovery. Besides, LLM-guided evolution has also found use in discovering heuristics for combinatorial optimization (Liu et al., 2024).

B Additional Methodology Details

B.1 Library Learning

The objective of this stage is to generate reusable insights as structured 4-tuples (*Taxonomy*, *Condition*, *Explanation*, *Example*) and organize them in a hierarchical taxonomy for efficient retrieval, while minimizing redundancy in the library. The left panel of Figure 4 (see the Library learning flow label) illustrates the workflow for this stage.

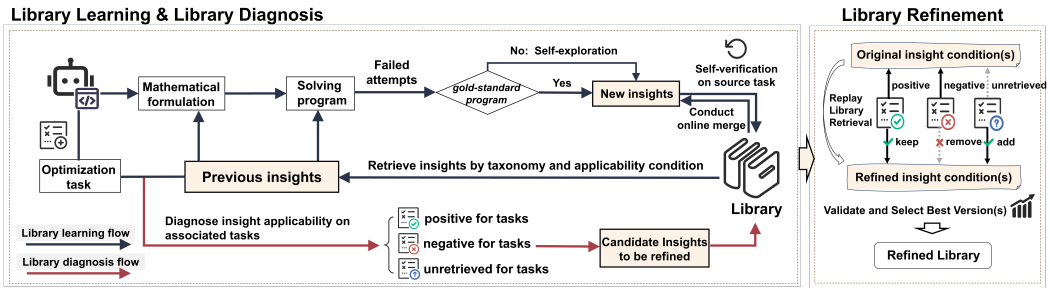


Figure 4: The overall workflow of library learning and evolution. The left panel depicts two complementary flows: library learning, which extracts new insights by generating and consolidating insights from failed optimization attempts, and library diagnosis, which analyzes interactions between failed tasks and retrieved insights to collect negative and unretrieved cases for refinement. The right panel illustrates library refinement, where the LLM refines each insight's applicability conditions, validates them via retrieval replay, and integrates the updated insights back into the library.

Insight Extraction, Representation, and Supervision Mode. Insights can be learned from either problems with a gold-standard program or with the answer alone. For each 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, two supervision modes are used to guide the generation of new insights. When a *gold program* is available, the system compares the candidate formulation and program against the reference, diagnosing discrepancies (e.g., missing variables, misformulated constraints, incorrect objective terms) and distilling them into insights. When only the *answer* (i.e., final optimal objective) is provided, 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. Before being stored in the library, each insight is locally verified by reapplying it to its source task to ensure that it resolves the original failure. In addition, to mitigate stochastic successes that could obscure useful lessons, we conduct three independent trials per task, allowing errors from probabilistic generation to serve as signals for learning.

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

Library Storage and Retrieval. Insights are stored in a dynamically updated hierarchical taxonomy organized into three main tracks: *Domain Modeling* (problem-specific structures and assumptions), *General Formulation* (reusable mathematical patterns), and *Code Implementation* (solver-specific coding practices). Under each track, insights are further classified with two-level labels, where Level-1 captures a broad category and Level-2 refines it into a more specific subcategory. The taxonomy 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. When storing insights, to reduce redundancy, the LLM also checks whether a similar insight already exists and performs merging when appropriate. 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, Level-2 label such as Fixed Charge (Big-M Linking) will be probably 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.

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. This two-step procedure ensures that insights are applied

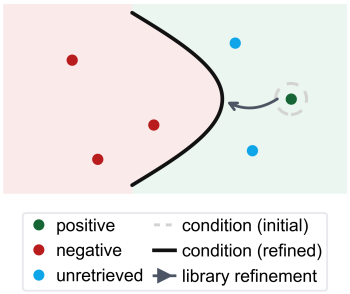


Figure 5: A locally verified *initial* condition (light-grey dashed circle) is refined into a broader *applicability* boundary (solid black) through evolutionary prompt optimization guided by the aggregate metric.

appropriately, while the hierarchical taxonomy provides an extensible structure for organizing and retrieving insights as the library grows.

Operational Flow. Training proceeds in an online regime over minibatches of data, starting from an empty library. For each batch, the system retrieves candidate insights, generates and executes programs, and upon failures extracts insights and immediately commits those that pass local self-verification, allowing newly added insights to benefit subsequent tasks and preventing the generation of repetitive insights. To reduce redundancy, tasks are clustered and reordered by problem type and semantic similarity, and overlapping insights within a batch are merged prior to integration. The process iterates until overall accuracy plateaus, at which point the library is archived and used for evaluation.

In implementation, for the sake of efficiency, training follows two coordinated data flows. The first processes minibatches of tasks in parallel for insight extraction. The second maintains a centralized queue of all generated insights, storing them into the library sequentially. This queue does not allow asynchronous updates, as concurrent modifications could lead to conflicts if two insights attempt to update the library simultaneously. This design balances parallelism in problem-solving with strict serialization in library updates, ensuring both efficiency and consistency.

B.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 from 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 missed opportunities or systematic failures. Library Evolution counters this with a diagnostic–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-hox fixes. As illustrated in Figure 5, library refinement can be understood as adjusting each insight's condition toward the correct retrieval boundary in the problem space.

Diagnostic: Library Diagnosis. After each training round, we trace failed tasks and analyze their interaction with the library. The diagnostic 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 where the insight was applicable and contributed to the correct formulation, S_i^- contains tasks where it was misleading and degraded performance, and S_i^u contains tasks where 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, thus avoiding redundant insight generation. Unretrieved tasks are identified by first comparing the model's generated program with the ground-truth (or a reference program from self-exploration) to locate discrepancies. These discrepancies guide the search for

candidate insights, which are then verified for their ability to resolve the errors. Verified insights are flagged as relevant but unretrieved, allowing the system to diagnose retrieval gaps without resorting to intractable combinatorial search. The left panel of Figure 4 (see the Library diagnosis flow label) illustrates the workflow for this stage.

Evolver: Library Refinement. Building on the diagnosis, the Evolver agent refines insights in two steps: condition refinement and refinement verification. First, for each diagnosed insight, the agent 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. The Evolver then proposes multiple refinement strategies (e.g., adding preconditions, introducing keyword anchors, merging overlapping triggers) and produces 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 still remain correctly retrieved after refinement; "corrected negatives" are tasks that were misled by the insight before and 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 4.

B.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 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 E, we provide a conceptual sketch showing that convergence holds: As 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 the redundancy-reduction operations.

C Comparative Analysis of AlphaOPT against Prior Experience-Learning Methods

Recent approaches such as Reflexion (Shinn et al., 2023), STaR (Zelikman et al., 2022), ExpeL (Zhao et al., 2024), and AlphaEvolve (Novikov et al., 2025) 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 open-ended reasoning and programming, but they face limitations for optimization problems. First, their experiences are largely preserved as free-form text or edits without explicit applicability semantics, yet in optimization tasks, applying such experiences inappropriately can have detrimental effects. Second, their verification is limited to task outcomes such as checking rewards, final answers, or passing test cases, which does not guarantee that the underlying knowledge is structurally valid or transferable.

Our framework adapts experience learning to operations research (OR) with three key innovations: (1) solver-guided verifiability: correctness is judged at the program level. If a program achieves the

optimal objective under the solver, it is highly likely to be valid and can serve as a reliable anchor for extracting insights, broadening the sources of experience collection. New and refined insights are explicitly re-tested on associated tasks, ensuring they are valid before integration; (2) structured knowledge for interpretability and auditability: each insight is represented with taxonomy, condition, explanation, and example, making its applicability explicit, reviewable, and even revisable in practice; (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, improving safe transfer across problem families. See Table 3 for detailed comparisons.

Table 3: Comparison of experience-learning methods. Prior works improve through experiential reuse but rely on free-form knowledge and outcome-level verification. Our framework introduces structured insights, solver-guided verification, and refined applicability, which are crucial for OR.

Method	Structured knowledge	Explicit applicability	Verification	Applicability refinement	Application domain
Reflexion	✗	✗	Reward signal	✗	General agents
STaR	✗	✗	Answer correctness	✗	QA / reasoning
ExpeL	✗	(✓) minimal	Task success assumed	✗	General agents
AlphaEvolve	✗	(✓) implicit	Test harness (partial)	✗	Code synthesis / evolution
AlphaOPT	✓	✓	Solver optimality + insight verification	✓	OR formulation and solver code

D Datasets

We have collected the publicly available optimization problem datasets listed in the Table 4, which include both natural language problem descriptions and their optimal solutions. They are aggregated from four real-world optimization and operation task datasets, namely the NLP4LP (AhmadiTeshnizi et al., 2024), NL4OPT (Ramamonjison et al., 2022), IndustryOR (Huang et al., 2025), MAMO (ComplexLP) (Huang et al., 2024), with any invalid entries discarded. These collections span various formulation types and originate from diverse sources, including academic papers, textbooks, and real-world industry scenarios.

NLP4LP, NL4OPT, IndustryOR, and MAMO (ComplexLP) are used to construct our training and test datasets. The gold-standard programs for the training datasets NLP4LP and IndustryOR are obtained from Yang et al. (2025a). We perform stratified sampling within each dataset, randomly partitioning 70% for training and 30% for testing. We maintain a strict separation between training and test data. The experience library is constructed only from training tasks, ensuring that training-derived insight examples do not leak into the test set. To assess out-of-distribution (OOD) generalization, we additionally evaluate on LogiOR (Yang et al., 2025a) and Optibench (Yang et al., 2024).

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 4 indicates the size of each dataset after cleaning. Specifically, for NLP4LP, IndustryOR and LogiOR we use the cleaned versions provided by Yang et al. (2025a); for NL4OPT, MAMO (ComplexLP) and Optibench we use the cleaned releases from Astorga et al. (2025), obtained from their GitHub repository.

E 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 $\mathrm{Success}(t \mid \ell)$ (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:

Table 4: The statistics of the optimization problem datasets

Dataset	Size	Formulation Type(s)	Completion
NL4OPT (Ramamonjison et al., 2022)	289	LP	solution
NLP4LP (AhmadiTeshnizi et al., 2024)	242	LP, MILP, MINLP	solution, program
MAMO (complex LP) (Huang et al., 2024)	111	LP	solution
IndustryOR (Huang et al., 2025)	82	LP, IP, MILP, NLP, others	solution, program
Optibench (Yang et al., 2024)	403	LP, MILP, MINLP	solution
LogiOR (Yang et al., 2025a)	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.

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 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 1. *Assume $\mathcal{T}_{\text{train}} = \mathcal{T}_{\text{test}}$. If the library space $\mathcal{L}$ is finite, then the algorithm terminates in finitely many steps at a library ℓ^* which 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 $F(\ell)$, and otherwise the library remains unchanged. Since F is bounded above, the sequence $\{F(\ell_k)\}$ is monotone non-decreasing and bounded, 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, so 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 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 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 $\ell_{k_j} \to \ell^\infty$. 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, so any local optimality statement directly applies to testing. □

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

F Library Analysis

F.1 Library Taxonomy Structure

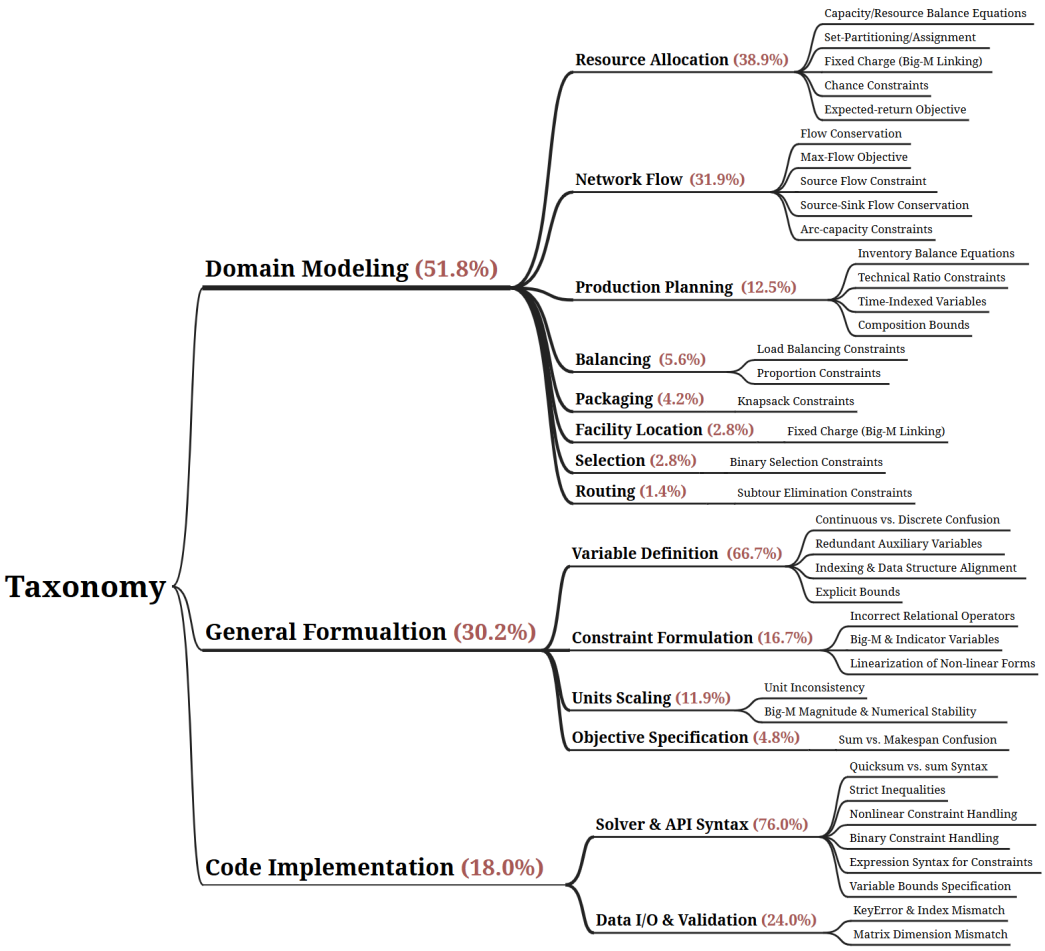


Figure 6: The learned experience-library taxonomy has three hierarchy levels: 3 main tracks, 14 level-1 labels, and 38 level-2 labels. This figure presents the level-1 and level-2 terms and their assignments. Percentages in red, shown in parentheses after each track and level-1 term, denote that category's percentage within its parent. The detailed specification and distribution of level-2 labels are provided in the Appendix F.2 and Appendix F.4.

As illustrated in Figure 6,

1) The library mainly captures insights that address LLMs' difficulties with domain-specific modeling structures, particularly those involving structural coupling and constraint balance. In the library taxonomy, the number of insights under *Domain Modeling* track accounts for 52% of the total. Within this track, the most frequent level-1 labels are *Resource Allocation* (38.9%), *Network Flow* (31.9%), and *Production Planning* (12.5%). Structural coupling, which refers to the model's difficulty in capturing cross-variable or cross-stage dependencies, is reflected in level-2 labels such as *Fixed Charge (Big-M Linking)*, *Set-Partitioning/Assignment*, and *Time-Indexed Variables*. Constraint balance, which refers to the model's failure to maintain system-wide conservation and resource equilibrium, is reflected in level-2 labels such as *Capacity/Resource Balance Equations*, *Flow Conservation*, and *Inventory Balance Equations*.

2) The library captures insights that help transform intuitive or context-based reasoning into mathematically rigorous and solver-consistent formulations, particularly in defining variables, formalizing constraints, and maintaining numerical consistency. *General Formulation* track accounts for 30% of all insights. Within this track, the the most frequent level-1 labels are *Variable Definition* (66.7%), followed by *Constraint Formulation* (16.7%) and *Units Scaling* (11.9%). Variable definition difficulty lies in specifying variable domains and maintaining structural consistency, as reflected in level-2 labels such as *Explicit Bounds*, *Continuous vs. Discrete Confusion*, and *Indexing & Data Structure Alignment*. Constraint formulation difficulty lies in the model's inaccuracy in representing logical and numerical relationships, as reflected in level-2 labels such as *Incorrect Relational Operators*, *Big-M & Indicator Variables*, and *Linearization of Non-linear Forms*. Units scaling difficulty lies in maintaining numerical coherence and stability, as reflected in level-2 labels such as *Unit Inconsistency*, *Big-M Magnitude & Numerical Stability*.

3) The library captures insights that help bridge the gap between symbolic formulations and executable solver code, particularly in handling solver syntax and maintaining data consistency. *Code Implementation* track accounts for 18% of all insights, this proportion is lower than formulation-level insights though, it still exposes critical weaknesses in the execution stage. Within this track, most errors originate from *Solver & API Syntax* (76.0%), while the remaining issues arise from *Data I/O & Validation* (24.0%). Solver & API syntax errors reflect the model's lack of solver awareness and insufficient control over formal expression generation, as evidenced by level-2 labels such as *Nonlinear Constraint Handling*, *Strict Inequalities*, *Quicksum vs. sum Syntax*. Data I/O & validation errors, on the other hand, reveal instability in aligning mathematical indices with data structures, as reflected in level-2 labels such as *KeyError & Index Mismatch* and *Matrix Dimension Mismatch*.

F.2 Details of Library Insight Distribution

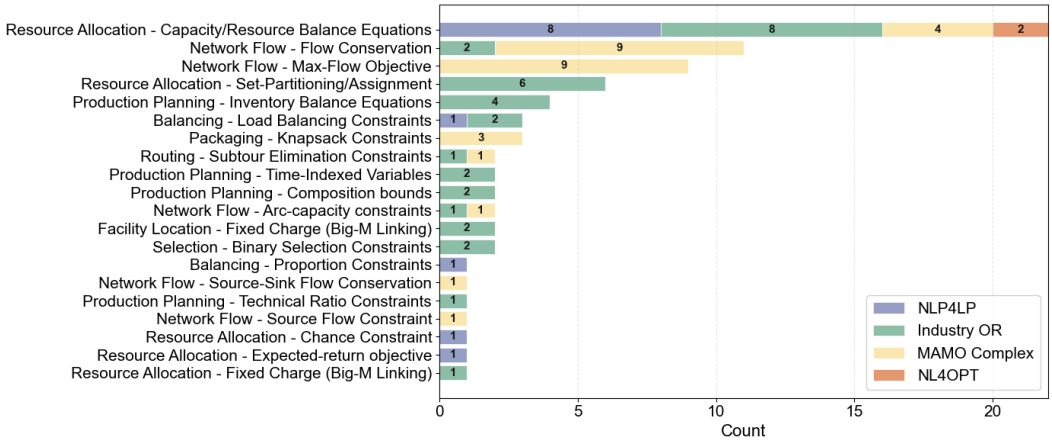
To further understand the detailed content of the library insights, we analyze the distributions of insights under the level-2 taxonomy labels, as well as the contribution differences among training datasets (as depicted in Figure 7).

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 resource usage does not exceed available capacity, demonstrating how to establish constraints that maintain balance between resource consumption and availability.

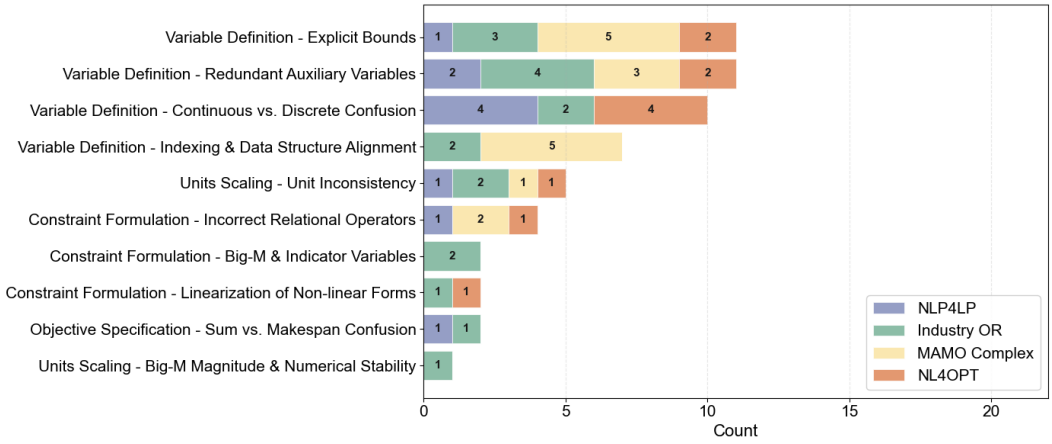
In the *General Formulation* track, *Variable Definition – Explicit Bounds*, *Redundant Auxiliary Variable*, and *Continuous vs. Discrete Confusion* are the most frequent, with diverse sources, indicating that these are common issues across multiple datasets. This reflects that foundational concepts in variable definition are the most error-prone in optimization modeling—particularly in balancing variable types, value ranges, and modeling simplifications—where 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, 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—representing the dual pillars required for bridging mathematical modeling and executable code.

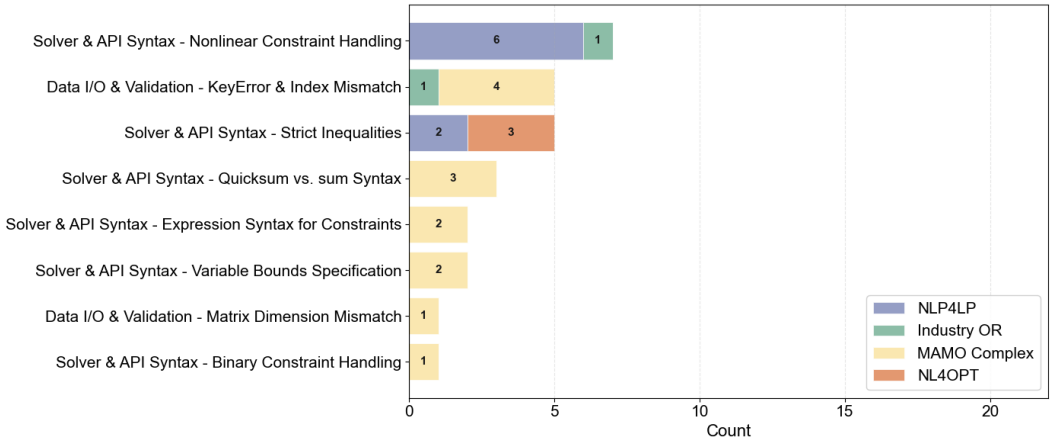
From the perspective of dataset contribution, Industry OR is dominant in *Domain Modeling*; MAMO (Complex LP) and Industry OR lead in *General Formulation*; and both MAMO (Complex LP) and NLP4LP contribute the most to *Code Implementation*. NL4OPT has relatively lower overall participation but focuses on formulation- 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, indicating that these datasets contain denser structural modeling challenges and more diverse error patterns, enabling the LLM to accumulate more experiential knowledge across multiple dimensions.



(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.

F.3 Examples of Library Refinement

We selected five representative refined insights from the library and explained how their applicability conditions were adjusted based on the associated tasks.

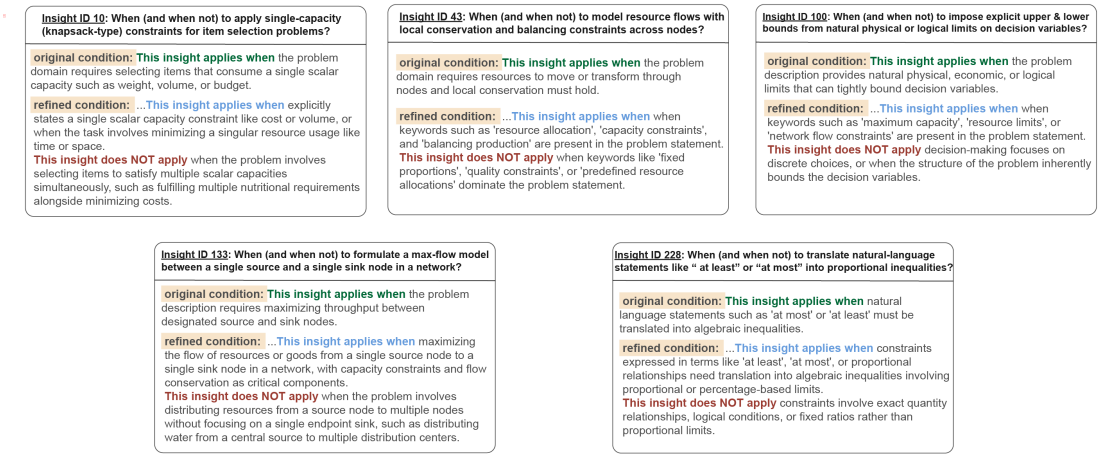


Figure 8: The examples of insight conditions before and after refinement, with green text indicating the original applicability conditions, and blue and red marking the newly added applicable and non-applicable conditions after refinement.

As shown in the Figure 8, *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, gating suppresses spurious retrievals where the structure is not true flow/-conservation. *Insight ID 100* supports adding tight explicit bounds derived from natural limits; refinement strengthens positive cues (e.g., maximum capacity, resource limits) and excludes cases that are purely discrete or already inherently bounded. *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. *Insight ID 228* covers translating *at least/at most* expressed as proportions into inequalities; refinement broadens to general percentage phrasing, clarifies the mapping, and excludes exact quantities, logical relations, or fixed ratios that are not proportion limits.

Across the five cases, four strategies proposed by LLM agent recur: (i) generalize with equivalent phrasings (e.g., minimize a single resource); (ii) lexical anchoring with positive keywords to raise recall where appropriate; (iii) explicit exclusion lists to reducing misalignment with tasks; (iv) structural qualifiers (single-scalar capacity; single-source–single-sink) to prevent misuse. A refinement is considered effective when the reduction in negative and unretrieved tasks outweighs the decrease in positive tasks. As shown in the Figure 9, these semantic refinements reduce both negative and unretrieved cases while preserving as many positive cases as possible.

F.4 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 generated insights under the same label inherit that initial condition.

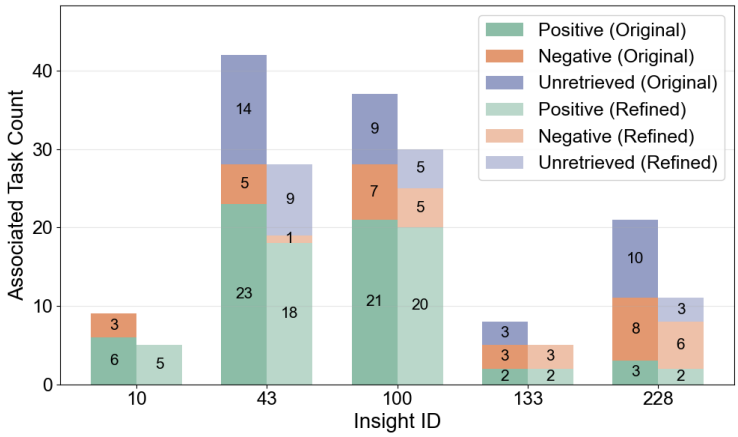


Figure 9: Changes in the number of positive, negative, and unretrieved tasks associated with each insight before and after applicability condition refinement. A decrease in negative and unretrieved tasks indicates that, the insight no longer mismatches unrelated tasks (negative) and successfully matches previously applicable but missed tasks (unretrieved) after refinement.

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 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.

G Additional Results

G.1 Ablation Study

We assess the effectiveness of library insight retrieval/application and self-debug:

- **w/o self-debug:** Remove the model's self-debugging section.
- **w/o taxonomy:** Remove the library taxonomy and directly match insights by checking all conditions.
- **w/o insight example:** Use only the explanation as input, excluding exemplar cases

Table 5 shows that the full AlphaOPT achieves the best scores on all out-of-distribution datasets. Removing self-debug yields the largest drop on Logior (15.21%), indicating it plays an important role in the system. Dropping the library taxonomy reduces accuracy by 4.34% on Logior and by 1.09% on Optibench, suggesting that structured matching meaningfully improves retrieval. Excluding insight examples similarly lowers performance, showing that concrete, worked snippets aid application beyond textual explanations alone.

Table 6: The ablation results of AlphaOPT without: performance on benchmarks.

Dataset	Logior (92)	Optibench (403)
AlphaOPT (full)	**51.08**%	**91.81**%
w/o self-debug	35.87%	89.26%
w/o taxonomy	46.74%	90.72%
w/o insight example	45.65%	91.06%

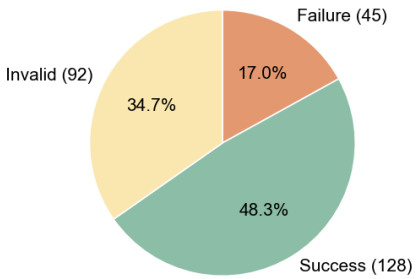
G.2 Success And Failure Case Study

As shown in Figure 10, approximately half of the retrieved insights successfully contributed to solving new optimization tasks, with only a small portion introducing new errors (Failure), demonstrating the effectiveness of our method in automating knowledge transfer for optimization modeling. Moreover, insights under different level-1 taxonomy labels exhibit distinct performance patterns across modeling dimensions.

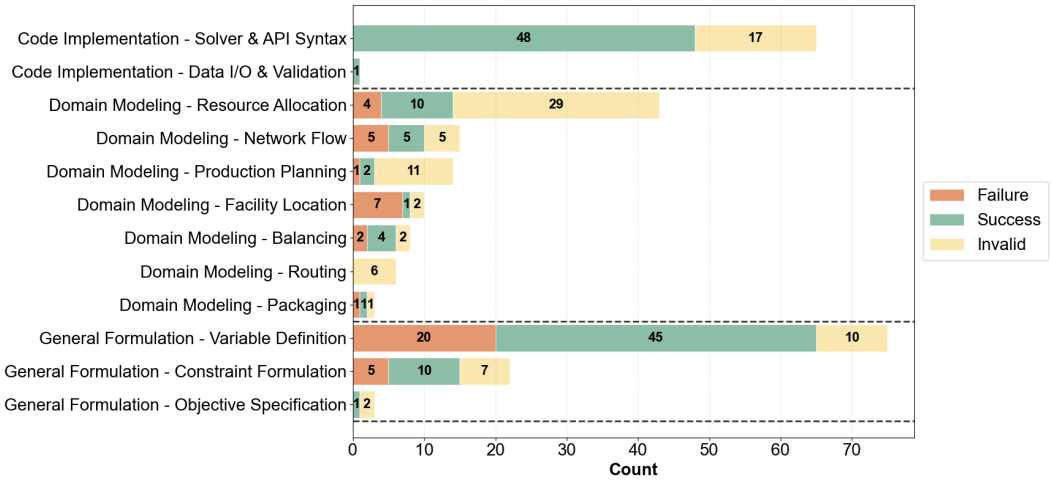
1) Insights with high success rates These insights are concentrated in *Code Implementation – Solver & API Syntax* and *General Formulation – Variable Definition*. These insights typically target code implementation or fundamental modeling errors with clear structures, enabling the LLM to follow their guidance stably and correctly. For instance, insights tagged with *Strict Inequality* label under *Solver & API Syntax* highlights 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 *Explicit Bounds* label under *Variable Definition* emphasizes 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 *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, ultimately causing solver failure. Additionally, while insights under *Variable Definition* generally achieves high success rates, some insights tagged with *Explicit Bounds* label sometimes are overly rigid, leading the LLM to impose unnecessary upper bounds, thereby restricting the feasible space and producing 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



(a) Overall effectiveness of retrieved library insights



(b) Effectiveness distribution across level-1 insight categories

Figure 10: 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) Overall distribution of insight outcomes. (b) Proportions of successful, failed, and invalid insights under level-1 taxonomy labels.

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 (neglecting auxiliary variables or mis-rewriting proportional constraints), resulting in insights being recognized but not operationalized.

Additionally, certain tasks remained unsolved regardless of whether library retrieval was enabled, as 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 remain insufficient for modeling such complex logic, revealing 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.

H 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.
```