OPTWEAVER: LEARNING WHAT TO REMEMBER FOR OPTIMIZATION MODELING

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models have advanced automatic optimization modeling from natural-language descriptions. Recent knowledge-augmented methods improve performance by retrieving relevant knowledge from solved problems. However, retrieval relevance does not guarantee modeling applicability or specify how the retrieved knowledge should participate in model construction. Moreover, knowledge updates after failures may not match the localized source of the error. We propose OptWeaver, a bidirectional framework that organizes accumulated modeling knowledge into reusable skills for forward model construction and backward knowledge evolution. For each problem, OptWeaver first adapts reusable skills into temporary task-specific guidance. It then turns this guidance into concrete modeling decisions and weaves them through mathematical dependencies into a modeling graph for solver code generation. When execution fails, OptWeaver traces the same construction structure backward to localize the responsible source and guide targeted repair. Reusable principles distilled from validated repairs are incorporated into the skill system for reuse, with changes localized to relevant modeling knowledge. This design makes knowledge use explicit during model construction and aligns knowledge updates with the attributed source of modeling failures. Across six benchmarks, OptWeaver achieves an average accuracy of 78.35%, exceeding the strongest average baseline by 6.40%. The learned skills also transfer across model families, model scales and modeling frameworks.

1 INTRODUCTION

Mathematical models are fundamental tools in operations research (OR) for supporting complex decisions across modern industry, including production scheduling, logistics, vehicle routing and power-system operation (Ghasemi et al., 2024; Gu et al., 2024; Motta et al., 2024). Yet translating a real-world problem into a rigorous optimization model remains challenging. Practical requirements are often communicated in natural language and intertwined with implicit assumptions, ambiguous conditions and domain-specific rules. Formulating these requirements mathematically therefore demands substantial expertise and often iterative collaboration between domain experts and optimization modelers (Mostajabdaveh et al., 2024; Zhang et al., 2024). The expertise-intensive process creates a major bottleneck to the broader use of optimization, particularly in unfamiliar domains (Xiao et al., 2024; Huang et al., 2025a).

Recent advances in large language models (LLMs) offer a promising way to lower this barrier by translating natural-language problem descriptions into mathematical formulations and solver code (Ramamonjison et al., 2023; Xiao et al., 2024). Existing methods mainly follow two directions. One focuses on domain-specific training or fine-tuning (Xiao et al., 2026; Ding et al., 2026), while the other enhances fixed models at inference time through prompting, structured reasoning, search, external tools and solver feedback (Xiu et al., 2026; Chen et al., 2026a). We focus on the latter. Within this line of work, early approaches elicited knowledge already encoded in the model, whereas recent methods increasingly accumulate modeling knowledge from previously solved problems and reuse it on new ones (Xie et al., 2026; Lian et al., 2026). In these methods, accumulated knowledge is typically reused by retrieving relevant knowledge for a new problem based on semantic similarity or applicability conditions (Xie et al., 2026; Kong et al., 2026).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

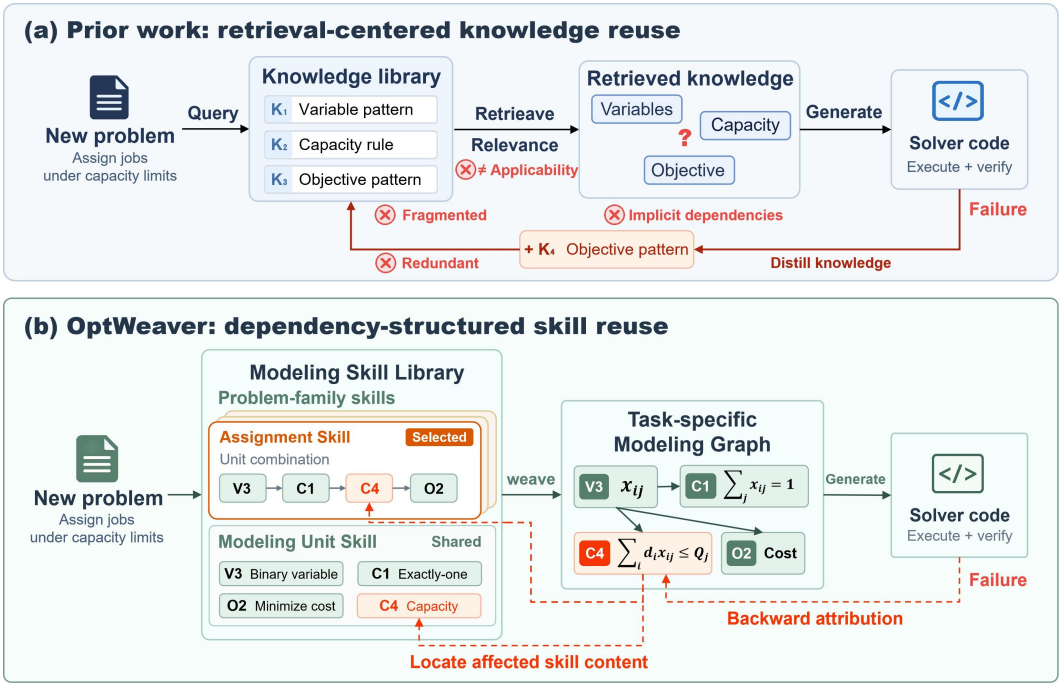


Figure 1: Comparison of modeling-knowledge reuse paradigms. (a) Prior work relies on knowledge accumulation and similarity-based reuse, with knowledge accumulating as separate pieces while construction dependencies remain implicit. (b) OptWeaver uses structured, reusable Modeling Skills and explicit modeling dependencies for structured construction, targeted repair and knowledge evolution. Targeted local updates refine the affected C4 content and implicated dependencies.

Despite this progress, existing retrieval-centered methods still leave two related limitations unresolved, as illustrated in Figure 1. First, existing methods provide limited support for grounding retrieved knowledge in concrete modeling decisions and organizing those decisions into a coherent formulation. Retrieval relevance does not guarantee modeling applicability, and useful knowledge may also be missed when problem descriptions or structures shift. Even when appropriate knowledge is retrieved, existing methods offer limited guidance on where it should be applied or how the resulting decisions should interact through mathematical dependencies. Second, knowledge updates may not match the localized nature of modeling failures. Because a failure may originate from only a local variable definition, constraint or dependency, updating an entire knowledge item may affect content beyond the actual source of the error. This can alter otherwise valid content or overspecialize the knowledge to a particular case. Alternatively, introducing a new item for each local correction can create redundancy when it differs from existing knowledge only in local modeling details. Both limitations arise because reusable knowledge is not explicitly connected to the concrete modeling decisions through which it is applied, evaluated and revised.

To address these two limitations, we introduce OptWeaver, which organizes accumulated modeling knowledge as structured, reusable Modeling Skills in a persistent library and reuses them to guide new model construction. For each problem, the selected problem-family skill and shared modeling unit skill are adapted into temporary task-specific guidance. This guidance drives the construction of a task-specific Modeling Graph, which explicitly records dependencies among modeling decisions and supports solver code generation. When execution fails, OptWeaver traces the same construction structure backward to locate the responsible modeling source and guide targeted repair, rather than directly revising the associated skill. Reusable modeling principles extracted from validated corrections are then used to locally update the knowledge implicated by backward attribution, while preserving unaffected content. New knowledge is introduced only when the existing skills cannot represent these principles. During training, reference solutions are used only when autonomous repair fails, while the learned skill library is frozen at test time. Across multiple optimization modeling benchmarks, OptWeaver achieves an average accuracy of 78.35%, outperforming the strongest aver-

age baseline by 6.40%. The accumulated modeling knowledge also transfers across model families, model scales and other modeling frameworks.

Our main contributions are threefold.

- We introduce OptWeaver, which organizes accumulated modeling knowledge into complementary problem-family skills and a shared modeling unit skill. Their knowledge is adapted into task-specific guidance that instantiates concrete modeling decisions, which are woven through explicit mathematical dependencies to construct optimization models.
- We develop a bidirectional learning mechanism in which forward skill weaving constructs mathematical models, while backward attribution reuses the same construction structure to localize failures, guide targeted repair, and evolve the implicated reusable knowledge while preserving unaffected content.
- Extensive experiments show that OptWeaver improves optimization modeling performance and transfers learned knowledge across model families, scales and frameworks.

2 Related Work

2.1 Learning-based Optimization Modeling

Learning-based approaches improve optimization modeling by incorporating domain knowledge into model parameters. LLMOPT (Shu et al., 2025) uses unified formulation and instruction tuning, while OptMATH (Lu et al., 2025) scales training through synthesized natural language problems and mathematical formulations. Recent methods further introduce solver-grounded supervision. SIRL (Chen et al., 2026b) uses solver-verified rewards, while StepORLM (Zhou et al., 2026) and OR-PRM (Wang et al., 2026) provide finer process-level supervision. These methods show that training can strengthen optimization modeling. In contrast, we explicitly accumulate and reuse modeling knowledge without further parameter updates.

2.2 Knowledge-Augmented Optimization Modeling

Knowledge-augmented methods improve optimization modeling by providing reusable knowledge during inference. OptiTree (Liu et al., 2026b) organizes modeling thoughts through a problem hierarchy, while AlphaOPT (Kong et al., 2026) learns solver-verified insights and refines their applicability for reuse. Recent methods such as Opt-Miner (Liu et al., 2026a), OptMaster (Lin et al., 2026a) and Agora-Opt (Lin et al., 2026b) further explore active retrieval, cross-branch sharing and persistent reuse. However, these methods mainly organize knowledge by problem taxonomy, semantic relevance, applicability or search context, offering limited guidance on how knowledge should be used in model construction or updated after localized failures. OptWeaver instead organizes modeling knowledge into complementary problem-family skills and a shared modeling unit skill. It then connects task-specific modeling decisions through explicit dependencies and reuses the same construction structure for targeted repair and localized knowledge evolution.

3 Method

This section details the forward–backward learning process of OptWeaver. As illustrated in Figure 2, OptWeaver maintains complementary problem-family skills and a shared modeling unit skill in a persistent library for modeling knowledge (Section 3.1). For each problem, reusable knowledge is adapted into temporary task-specific guidance. This guidance instantiates modeling decisions, which are woven into a task-specific Modeling Graph through explicit mathematical dependencies (Section 3.2). Solver feedback is traced backward through the same construction structure to locate responsible modeling sources, guide targeted repair and extract reusable modeling principles from validated corrections (Section 3.3). Reusable principles distilled from validated corrections drive localized skill evolution during learning, while the library remains frozen at test time (Section 3.4).

In implementation, these stages use stage-specific modules driven by dedicated prompts. Complete algorithms are provided in Appendix C, a learning-process case study in Appendix D and representative prompts in Appendix F.

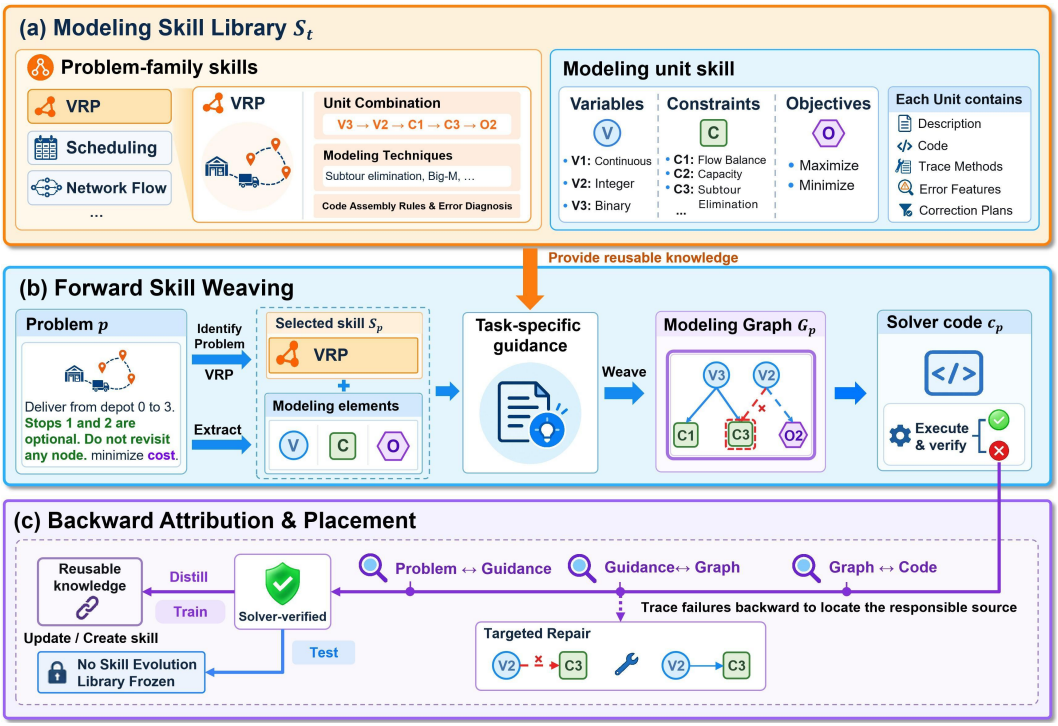


Figure 2: Overview of OptWeaver. Forward skill weaving adapts the retrieved problem-family skill and shared modeling unit skill into temporary task-specific guidance, then instantiates concrete modeling decisions and weaves them through mathematical dependencies into a Modeling Graph for solver code generation. Solver feedback is traced backward through the same construction structure for targeted repair. During training, validated corrections further evolve the skill system. At test time, the learned skill library is frozen, while local repair remains enabled without persistent knowledge evolution.

3.1 Modeling Skills as Structured Knowledge Carriers

Instead of retaining complete solved problems or isolated textual corrections, OptWeaver maintains accumulated modeling knowledge in a persistent Modeling Skill Library,

$$\mathcal{S}_t = \mathcal{S}_t^{\mathrm{PF}} \cup \{s_t^{\mathrm{MU}}\}, \tag{1}$$

where t denotes the current learning stage, $\mathcal{S}_t^{\mathrm{PF}}$ denotes the set of problem-family skills and s_t^{MU} denotes the shared modeling unit skill. The two components organize reusable modeling knowledge at complementary levels. Problem-family skills capture domain-specific formulation guidance for different classes of optimization problems, while the modeling unit skill provides reusable knowledge for constructing variables, constraints and objectives.

Complementary Problem-Family Skills and the Shared Modeling Unit Skill. Problem-family skills are organized by optimization problem class. For a new problem, OptWeaver first identifies its problem class and retrieves the corresponding problem-family skill. Because such knowledge is closely tied to problem structure and domain semantics, the retrieved skill provides high-level formulation guidance specific to that problem family, covering unit combinations, modeling techniques, code assembly rules and error diagnosis.

However, high-level formulation guidance alone does not guarantee correct model construction. Even when the overall modeling approach is appropriate, local errors may still occur, such as omitting a necessary variable index, applying a constraint to the wrong object set or misinterpreting a local relation that subsequently affects other constraints or the objective.

OptWeaver therefore uses the shared modeling unit skill to capture reusable OR knowledge at the level of variables, constraints and objectives. Each modeling unit contains a description of the mod-

eling principle, implementation code, trace methods, characteristic error features and corresponding correction plans. These components jointly support the construction, diagnosis and correction of local modeling decisions, helping reduce localized but consequential formulation errors. When applied to a task, the modeling unit skill supports the construction of task-specific variables, constraints and objectives, which form the basic components of the Modeling Graph. The resulting modeling decisions are connected through task-specific mathematical dependencies, making the model construction process explicit and providing a structural basis for subsequent backward attribution.

A Persistent Skill System. The library $\mathcal{S}_t$ serves as the persistent knowledge state shared by forward model construction and backward knowledge evolution. During forward modeling, it provides reusable knowledge from both problem-family skills and the modeling unit skill to support task-specific model construction. The skill system is initialized with a small amount of general modeling knowledge and progressively evolves during learning through validated reusable knowledge. The attribution and knowledge evolution mechanisms are detailed in Sections 3.3 and 3.4.

3.2 Forward Skill Weaving

Given a natural-language optimization problem p and the current skill library $\mathcal{S}_t$, OptWeaver constructs a task-specific model and executable solver code through forward skill weaving,

$$(S_p, G_p, c_p) = \mathcal{F}(p, \mathcal{S}_t), \tag{2}$$

where $S_p \subseteq \mathcal{S}_t$ denotes the skills used for p, G_p is the task-specific Modeling Graph and c_p is the resulting solver code. The forward process moves from identifying relevant knowledge to adapting it into task-specific guidance, instantiating concrete modeling decisions and organizing these decisions through explicit mathematical dependencies.

From Reusable Knowledge to Task-Specific Guidance. OptWeaver first identifies the modeling requirements in p, including decision variables, constraints and objectives. These elements serve as task-specific anchors for identifying the problem class. OptWeaver then retrieves the corresponding problem-family skill from $\mathcal{S}_t^{\mathrm{PF}}$. Rather than using the retrieved problem-family skill directly, OptWeaver adapts its formulation logic, modeling structures, techniques and diagnostic guidance to the current problem. The shared modeling unit skill further supplements this process with reusable construction knowledge. This adaptation yields temporary task-specific guidance that bridges reusable skill knowledge and problem-specific model construction. It specializes general modeling knowledge to the current modeling objects. The resulting guidance remains local to the current problem and does not modify or become part of the persistent skill library.

Weaving Modeling Decisions through Mathematical Dependencies. Task-specific guidance provides modeling principles for the current problem, but does not explicitly specify how concrete modeling decisions should be organized into a complete formulation. Without an explicit representation of their dependencies, the interactions among variables, constraints and objectives may remain implicit, increasing the risk of disconnected or inconsistent formulations. For example, an allocation variable may be defined correctly but remain disconnected from its activation variable if the required linking constraint is omitted.

OptWeaver therefore instantiates concrete modeling decisions from the guidance and organizes them through task-specific mathematical dependencies in the Modeling Graph. We represent the resulting structure as

$$G_p = (N_p, D_p), \tag{3}$$

where N_p contains the task-specific modeling decisions instantiated for variables, constraints and objectives, and D_p contains the mathematical dependencies among them. The Modeling Graph serves as an intermediate representation between task-specific guidance and solver code. By making such dependencies explicit, the graph supports coherent formulation construction and provides the structural information needed to trace solver feedback to responsible modeling decisions.

When a required modeling structure is not explicitly covered by the reusable knowledge in the current skill library, OptWeaver can derive the required task-specific modeling decision directly from the problem and incorporate it into G_p. The resulting decision remains local to the current problem unless subsequent validation shows that its underlying modeling principle is reusable, in which case the principle can be incorporated into the persistent skill system during knowledge evolution.

From Modeling Structure to Solver Code. Once the Modeling Graph is complete, OptWeaver converts G_p into executable solver code c_p. The Modeling Graph specifies the required mathematical structure, while code generation focuses on its solver implementation. As a result, solver failures can be distinguished according to whether they arise from the mathematical structure or its implementation.

3.3 Backward Knowledge Attribution and Placement

The generated solver program is executed to obtain

$$r_p = \mathrm{Solver}(c_p), \tag{4}$$

where r_p contains the execution outcome and available solver feedback. When the formulation is verified successfully, no correction is required. If verification fails, OptWeaver does not immediately attribute the failure to the associated skill, since the error may arise from only a localized inconsistency in the construction process. Instead, the construction structure established during forward skill weaving is reused to trace failures backward to the responsible modeling source. The identified source is then repaired, and validated corrections are distilled into reusable modeling principles for knowledge evolution.

Tracing Failures through the Construction Structure. OptWeaver traces solver feedback backward through the forward construction chain. It checks whether the code correctly realizes the Modeling Graph and whether the graph reflects the temporary task-specific guidance and its dependencies. Further upstream, it examines the correctness of task-specific bindings, the match between the selected problem-family skill and the problem structure, and the consistency of the extracted modeling elements with the requirements of p. This backward tracing localizes the inconsistency to its responsible modeling source. By tracing the same construction structure established during forward weaving, backward attribution remains aligned with the model construction process.

Repairing the Responsible Modeling Source. Once the source of the failure is identified, OptWeaver performs a targeted repair while avoiding unnecessary changes to unaffected modeling decisions. The repair may revise extracted modeling elements or task-specific binding, correct a dependency in G_p, introduce a missing modeling decision or modify the solver realization in c_p. When an upstream modeling decision is changed, the affected downstream structure is regenerated accordingly.

Targeted repair avoids unnecessary changes to unaffected modeling decisions, which could otherwise turn correct parts of the formulation into new errors. OptWeaver also records previous modifications for the current problem to avoid repeated ineffective repairs within the repair budget. This localized repair further provides a more specific basis for subsequent knowledge evolution, preventing a local failure from being treated as evidence that the entire associated skill should be revised.

Learning What to Remember. A validated repair provides a correction for the current problem, but only part of that correction may be useful beyond the current instance. OptWeaver therefore distills reusable modeling knowledge from the diagnosed failure and its successful repair,

$$\Delta k_p = \mathcal{B}(r_p, G_p, S_p), \tag{5}$$

where $\mathcal{B}$ denotes backward attribution, targeted repair and knowledge distillation. Δk_p preserves the general modeling principle revealed by the correction rather than the specific variables, coefficients or data of p. For example, correcting a missing relation such as $x_{ij} \leq y_i$ should not result in storing the particular expression. Instead, the reusable knowledge is that a decision controlled by binary activation requires an explicit linking relation. Instance-specific corrections are not retained as reusable knowledge, while the distilled principles are passed to knowledge placement.

Placing Knowledge Where It Belongs. Retaining useful knowledge alone is insufficient because its role within the modeling structure must also be preserved. OptWeaver therefore uses the backward attribution result and the scope of Δk_p to determine whether the reusable knowledge should be incorporated into the existing skill system for future model construction.

If the relevant knowledge already exists, OptWeaver updates the implicated skill content, including applicability conditions, modeling rules or relations, and merges closely related knowledge when

appropriate. New knowledge is introduced only when no suitable representation exists. A new problem-family skill is created when no existing one adequately matches the current problem, while a new modeling unit is added when reusable unit-level knowledge is absent from the shared modeling unit skill.

This targeted update is designed to reduce unnecessary modifications and the risk of overspecializing reusable knowledge to a single failure case, thereby preserving the generality and reusability of accumulated knowledge while limiting redundant accumulation.

3.4 Learning-time Skill Evolution and Test-time Reuse

During learning, reusable knowledge obtained through backward attribution and placement progressively updates the persistent skill system,

$$\mathcal{S}_{t+1} = \mathcal{U}(\mathcal{S}_t, \Delta k_p), \tag{6}$$

where $\mathcal{U}$ denotes the localized update determined by the knowledge-placement process. Repeated problem solving therefore allows validated reusable knowledge to accumulate in the persistent skill system and influence future model construction.

Solver-Verified Skill Evolution. During training, persistent knowledge is updated only with corrections that pass solver verification, preventing unsuccessful or unsupported repairs from accumulating in the skill library. Some training problems may remain unresolved after the autonomous repair budget is exhausted. In this case, OptWeaver uses a reference solution only to help locate the remaining discrepancy. The reference does not directly replace the generated formulation and is not copied into the final solver code. Once the unresolved modeling decision is identified, the correction still follows the same targeted repair, solver verification, knowledge extraction, and placement procedure before it can modify $\mathcal{S}_t$.

Frozen Skill Reuse at Test Time. After learning, the final skill library $\mathcal{S}_T$ is frozen. For each unseen problem, OptWeaver uses $\mathcal{S}_T$ to perform forward skill weaving, construct the Modeling Graph, and generate solver code. If verification fails, solver feedback can trigger local repair within the predefined budget. These repairs apply only to the current problem and do not modify $\mathcal{S}_T$.

The difference is that test-time corrections do not modify the persistent library, and reference-guided validation is disabled. The same $\mathcal{S}_T$ is therefore used throughout evaluation. This separation ensures that performance on unseen problems reflects the transferability of the modeling knowledge acquired during learning rather than additional knowledge acquisition from the test set.

4 Experiments

This section presents a comprehensive empirical evaluation of OptWeaver. We begin by describing the experimental setup in Section 4.1. We then assess overall modeling performance (Section 4.2), component contributions through ablation studies (Section 4.3), and skill transferability across model families, model scales and modeling frameworks (Section 4.4). Further experiments in Appendix B report full ablation and transfer results, study the effect of training scale on performance and skill-library growth, and evaluate within- and cross-distribution skill reuse under different training data sources. Representative modeling cases are provided in Appendix E.

4.1 Experimental Setup

Benchmarks and Baselines. We evaluate OptWeaver on six optimization modeling benchmarks, including NL4Opt (Ramamonjison et al., 2023), MAMO EasyLP, MAMO ComplexLP (Huang et al., 2025b), IndustryOR (Huang et al., 2025a), OptiBench (Yang et al., 2025) and OptMATH (Lu et al., 2025). These benchmarks span diverse optimization structures, ranging from relatively standard linear programs to problems involving integer decisions, logical dependencies and coupled constraints. We compare with three groups of methods. General LLMs include GPT-4o (Hurst et al., 2024), OpenAI-o3 (Singh et al., 2025), DeepSeek-V3 (Liu et al., 2024) and DeepSeek-R1 (Guo et al., 2025). Optimization specific fine-tuned models include ORLM (Huang et al., 2025a), LLMOPT (Shu et al., 2025), OptMATH (Lu et al., 2025) and SIRL (Chen et al., 2026b). Prompt-based

Table 1: Accuracy (%) on six optimization modeling benchmarks. Best results within the prompt-based group are shown in bold. Avg. denotes the unweighted average across the six benchmarks.

Method	NL4Opt	MAMO-E	MAMO-C	IndustryOR	OptiBench	OptMATH	Avg.
General LLMs							
GPT-4o	79.2	81.3	41.2	38.0	48.4	17.5	50.93
OpenAI-o3	86.8	79.2	53.5	43.0	62.8	43.4	61.45
DeepSeek-V3	79.8	88.3	50.2	37.0	55.1	44.0	59.07
DeepSeek-R1	82.4	87.2	67.9	45.0	67.2	34.9	64.10
Fine-tuned Methods							
ORLM	85.7	82.3	37.4	24.0	51.1	2.6	47.18
LLMOPT	80.3	89.5	44.1	29.0	53.8	12.5	51.53
OptMATH	95.9	89.9	54.1	31.0	66.1	34.7	61.95
SIRL	98.0	94.6	61.1	48.0	67.4	45.8	69.15
Prompt-based Methods							
CoT	74.0	82.9	40.7	35.0	53.1	21.1	51.13
CoE	79.2	85.9	43.1	33.0	55.2	24.1	53.42
OptiMUS	80.6	87.1	45.2	36.0	58.8	32.5	56.70
MCTS	89.6	88.0	51.6	46.0	67.9	38.6	63.62
AlphaOPT	93.0	92.5	61.6	49.0	69.9	39.2	67.53
OptiTree	97.1	92.4	75.9	47.0	69.4	**49.9**	71.95
OptWeaver	**98.0**	**95.8**	**81.8**	**70.0**	**76.9**	47.6	**78.35**

baselines include CoT (Wei et al., 2022), CoE (Xiao et al., 2024), OptiMUS (Ahmaditeshnizi et al., 2024), MCTS (Astorga et al., 2025), AlphaOPT (Kong et al., 2026) and OptiTree (Liu et al., 2026b).

Implementation and Evaluation. Unless otherwise specified, DeepSeek-V3 is used as the backbone of OptWeaver. All main experiments start from eight initial problem-family skills together with a shared modeling unit skill, and the library is evolved using 400 training problems. After learning, the resulting skill library is frozen for evaluation. Test problems may use solver feedback for up to three rounds of local repair, while reference-guided validation and persistent skill evolution are disabled. We report solving accuracy under a unified solver-based evaluation protocol. Detailed benchmark statistics, baseline descriptions, training configurations and evaluation protocols are provided in Appendix A.

4.2 Main Results

Table 1 compares OptWeaver with general LLMs, optimization specific fine-tuned models and prompt-based methods across the six benchmarks.

Overall Performance. OptWeaver achieves the strongest average performance across the evaluated benchmarks, with particularly clear gains on several structurally complex tasks. The improvements are smaller on more regular linear programming benchmarks, where strong baselines already perform well. This suggests that OptWeaver benefits not merely from additional optimization knowledge, but from structuring that knowledge for systematic reuse.

Comparisons with knowledge-augmented baselines further support this interpretation. AlphaOPT and OptiTree show that accumulated knowledge can improve optimization modeling, while OptWeaver further connects retrieved knowledge to concrete modeling decisions through mathematical dependencies. Although it does not dominate every benchmark, its stronger performance across heterogeneous problem structures indicates more robust knowledge reuse. Further analyses of training scale, data source, computational cost and structural coverage are provided in Appendix B.

4.3 Ablation Study

Figure 3 presents the ablation results for the main components of OptWeaver. Figure 3(a) reports benchmark-wise accuracy under component ablations, while Figure 3(b) shows average accuracy across variants. Component effects vary across benchmarks. Ablating problem-family skill selection

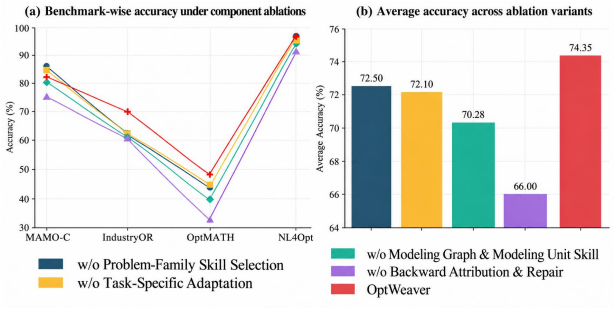


Figure 3: Ablation results for OptWeaver. (a) Benchmark-wise accuracy under component ablations on four representative benchmarks. (b) Average accuracy of each ablation variant across the four reported benchmarks.

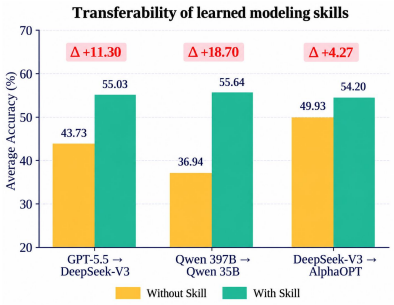


Figure 4: Transferability of learned Modeling Skills across model families, model scales and modeling frameworks.

or task-specific adaptation can occasionally improve performance on regular problem distributions, but such simplifications are less reliable when generalizing across more diverse problem structures.

As shown in Figure 3(b), the largest degradation occurs when backward attribution and repair are removed, indicating that this backward module is an important contributor to overall performance. This result is consistent with the design of OptWeaver, where solver feedback is traced through the construction structure to localize failures and guide targeted repair. Removing both the Modeling Graph and the modeling unit skill also weakens performance, especially on structurally demanding benchmarks in Figure 3(a), suggesting that unit-level modeling knowledge and explicit dependency structures jointly support effective model construction. Together, these results support OptWeaver's bidirectional design. Forward weaving organizes knowledge for model construction, while backward attribution uses the same structure to localize and repair modeling failures.

4.4 Knowledge Transferability

We further isolate the learned skills from the full OptWeaver inference procedure and evaluate their transfer across model families, scales and modeling frameworks, as summarized in Figure 4. Across model families, skills learned by one model still guide a different backbone when other OptWeaver components are removed. Similarly, skills learned by a larger model improve a smaller model. These results suggest that modeling capability acquired during learning can be externalized as reusable skills rather than remaining tied to the original model parameters.

The cross-framework experiment provides complementary evidence. As shown in Figure 4, skills learned with DeepSeek-V3 still improve AlphaOPT when its original solving procedure is preserved. This suggests that the skills are not merely intermediate states tailored to OptWeaver, but capture modeling principles reusable by other solving processes. Transfer remains uneven across benchmarks, indicating that effectiveness depends on whether the learned skills cover the structures required by the target distribution.

5 Conclusion

We presented OptWeaver, a bidirectional framework for optimization modeling that organizes reusable knowledge as structured skills. It weaves this knowledge into model construction through mathematical dependencies, while backward attribution supports targeted repair and localized skill evolution. Unlike retrieval-centered reuse, OptWeaver explicitly connects reusable knowledge to the modeling decisions through which it is applied, evaluated and revised. This enables targeted repair while preserving unaffected knowledge for future reuse. Experiments across six benchmarks show strong overall performance and demonstrate that the learned skills transfer across model families, model scales and modeling frameworks.

AI USE STATEMENT

Generative AI tools were used solely for translation and language editing to improve the clarity, fluency and readability of the manuscript. They were not used for methodological development, scientific reasoning, experimental design, implementation or result interpretation. All AI-assisted revisions were reviewed and approved by the authors, who remain responsible for the final manuscript.

REFERENCES

Ali Ahmaditeshnizi, Wenzhi Gao, and Madeleine Udell. Optimus: Scalable optimization modeling with (mi)lp solvers and large language models. In *International Conference on Machine Learning*, pp. 577–596. PMLR, 2024.

Nicolas Astorga, Tennison Liu, Yuanzhang Xiao, and Mihaela Van Der Schaar. Autoformulation of mathematical optimization models using llms. In *Proceedings of the 42nd International Conference on Machine Learning*, pp. 1864–1886. PMLR, 2025.

Yitian Chen, Cheng Cheng, Yinan Sun, Zi Ling, and Dongdong Ge. Opt-engine: Benchmarking the limits of llms in optimization modeling via complexity scaling. In *Forty-third International Conference on Machine Learning*, 2026a.

Yitian Chen, Jingfan Xia, Siyu Shao, Dongdong Ge, and Yinyu Ye. Solver-informed rl: Grounding large language models for authentic optimization modeling. *Advances in Neural Information Processing Systems*, 38:106027–106069, 2026b.

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

Amir Ghasemi, Fatemeh Farajzadeh, Cathal Heavey, John Fowler, and Chrissoleon T Papadopoulos. Simulation optimization applied to production scheduling in the era of industry 4.0: A review and future roadmap. *Journal of Industrial Information Integration*, 39:100599, 2024.

Wenjuan Gu, Claudia Archetti, Diego Cattaruzza, Maxime Ogier, Frédéric Semet, and M Grazia Speranza. Vehicle routing problems with multiple commodities: A survey. *European Journal of Operational Research*, 317(1):1–15, 2024.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645:633–638, 2025.

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*, 73(6):2986–3009, 2025a.

Xuhan Huang, Qingning Shen, Yan Hu, Anningzhe Gao, and Benyou Wang. Llms for mathematical modeling: Towards bridging the gap between natural and mathematical languages. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 2678–2710. Association for Computational Linguistics, 2025b.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

Minwei Kong, Ao Qu, Xiaotong Guo, Wenbin Ouyang, Chonghe Jiang, Han Zheng, Yining Ma, Dingyi Zhuang, Yuhan Tang, Junyi Li, et al. Alphaopt: Formulating optimization programs with self-improving llm experience library. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pp. 2389–2400, 2026.

Junbo Jacob Lian, Huiling Chen, Hanzhang Qin, and Chung-Piaw Teo. Admission without answers: Label-free certification and experience learning for llm-based optimization modeling. *arXiv preprint arXiv:2608.15565*, 2026.

Hang Lin, Yuanpeng Gao, Yuzhi Zhang, Kun Yuan, Gang Yan, Siheng Chen, Linfeng Zhang, et al. Optmaster: A dag-based framework for formulation and heuristic discovery in optimization. In *Forty-third International Conference on Machine Learning*, 2026a.

Jianghao Lin, Zi Ling, Chenyu Zhou, Tianyi Xu, Ruoqing Jiang, Zizhuo Wang, and Dongdong Ge. From soliloquy to agora: Memory-enhanced llm agents with decentralized debate for optimization modeling. *arXiv preprint arXiv:2604.25847*, 2026b.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Haoyang Liu, Yuyang Cai, Jie Wang, Xiongwei Han, Minyang Hu, Shuqi Liu, Mingxuan Yuan, Jianye Hao, and Feng Wu. Opt-miner: Empowering information-seeking agent with tree-guided data synthesis for optimization modeling. In *Forty-third International Conference on Machine Learning*, 2026a.

Haoyang Liu, Jie Wang, Yuyang Cai, Xiongwei Han, Yufei Kuang, and Jianye Hao. Optitree: Hierarchical thoughts generation with tree search for llm optimization modeling. *Advances in Neural Information Processing Systems*, 38:120713–120781, 2026b.

Hongliang Lu, Zhonglin Xie, Yaoyu Wu, Can Ren, Yuxuan Chen, and Zaiwen Wen. Optmath: A scalable bidirectional data synthesis framework for optimization modeling. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 40769–40802. PMLR, 2025.

Mahdi Mostajabdaveh, Timothy T Yu, Rindranirina Ramamonjison, Giuseppe Carenini, Zirui Zhou, and Yong Zhang. Optimization modeling and verification from problem specifications using a multi-agent multi-stage llm framework. *INFOR: Information Systems and Operational Research*, 62(4):599–617, 2024.

Vinicius N Motta, Miguel F Anjos, and Michel Gendreau. Survey of optimization models for power system operation and expansion planning with demand response. *European Journal of Operational Research*, 312(2):401–412, 2024.

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

Xiang Shu, Hong Qian, Xingyu Lu, JUN ZHOU, Aimin Zhou, Yang Yu, et al. Llmopt: Learning to define and solve general optimization problems from scratch. In *International Conference on Learning Representations*, volume 2025, pp. 101580–101606, 2025.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

Yilin Wang, Heng Zhou, Dongxing Mao, Linjie Li, Jingru Tan, Haochen Han, Zhengyuan Yang, Alex Jinpeng Wang, and Min Li. Or-prm: A process reward model for algorithmic problem in operations research. In *International Conference on Learning Representations*, volume 2026, pp. 42343–42369, 2026.

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

Ziyang Xiao, Dongxiang Zhang, Yangjun Wu, Lilin Xu, Yuan Wang, Xiongwei Han, Xiaojin Fu, Tao Zhong, Jia Zeng, Mingli Song, et al. Chain-of-experts: When llms meet complex operations research problems. In *International Conference on Learning Representations*, volume 2024, pp. 48519–48537, 2024.

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

Wantong Xie, Yinghao Chen, Yi-Xiang Hu, Feng Wu, Jieyang Xu, Sijia Zhang, and Xiangyang Li. Escaping the sisyphus dilemma: Experience replay for robust text-to-optimization modeling. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 14100–14116, 2026.

Xianchao Xiu, Jianhao Li, Jun Fan, and Wanquan Liu. Large language models for operations research: A comprehensive survey. *arXiv preprint arXiv:2605.20849*, 2026.

Zhicheng Yang, Yiwei Wang, Yinya Huang, Zhijiang Guo, Shi Shi, Xiongwei Han, Liang Feng, Linqi Song, Xiaodan Liang, and Jing Tang. Optibench meets resocratic: Measure and improve llms for optimization modeling. In *International Conference on Learning Representations*, volume 2025, pp. 24726–24759, 2025.

Jihai Zhang, Wei Wang, Siyan Guo, Li Wang, Fangquan Lin, Cheng Yang, and Wotao Yin. Solving general natural-language-description optimization problems with large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 6: Industry Track)*, pp. 483–490, 2024.

Chenyu Zhou, Tianyi Xu, Jianghao Lin, and Dongdong Ge. Steporlm: A self-evolving framework with generative process supervision for operations research language models. In *International Conference on Learning Representations*, volume 2026, pp. 6914–6940, 2026.

Appendix Contents

A Experimental Details **14**

A.1 Benchmarks and Evaluation Protocol 14

A.2 Baseline Details 14

A.3 Training and Inference Configuration 15

A.4 Initial Skill Library 15

B Additional Experimental Results **16**

B.1 Detailed Ablation Results 16

B.2 Transferability of Learned Skills 17

B.3 Effect of Training Scale 18

B.4 Effect of Training Data Source 18

B.5 Computational Cost 19

B.6 Skill Coverage and Problem Structure Analysis 20

C Algorithm Details **21**

C.1 Forward Skill Weaving 22

C.2 Backward Attribution and Skill Evolution 22

D Case Study of the OptWeaver Learning Process **24**

E Case Studies of Modeling with Learned Skills **29**

E.1 Using Advanced Modeling Knowledge 29

E.2 Enhancing Structured Modeling 31

F Prompts **33**

F.1 Modeling Element Extraction and Problem-Type Recognition 33

F.2 Task-Specific Guidance Construction 34

F.3 Modeling Subgraph Construction 35

F.4 Solver Code Generation 36

F.5 Reflection and Targeted Repair Prompts 37

F.6 Training-Only Validation Prompt 38

F.7 Knowledge Evolution Prompts 39

A Experimental Details

This section specifies the evaluation setting used throughout the paper, including benchmark coverage, comparison methods, learning and inference configurations, and the initialization of the Modeling Skill Library.

A.1 Benchmarks and Evaluation Protocol

Benchmark Coverage. We evaluate OptWeaver on NL4Opt (Ramamonjison et al., 2023), MAMO EasyLP and MAMO ComplexLP (Huang et al., 2025b), IndustryOR (Huang et al., 2025a), OptiBench (Yang et al., 2025) and OptMATH (Lu et al., 2025). The six benchmarks differ in both application context and formulation structure. NL4Opt and MAMO EasyLP mainly contain relatively regular linear programming patterns, while MAMO ComplexLP introduces more coupled variables and constraints. IndustryOR emphasizes application-oriented modeling from realistic problem descriptions. OptiBench broadens the range of optimization structures. OptMATH contains a larger proportion of mixed variable types, logical relations and combinatorial modeling patterns.

Taken together, the evaluation covers problems ranging from standard linear formulations to models that require several interacting decisions to be constructed consistently. This distinction is important for OptWeaver because its main objective is not only to retrieve relevant knowledge, but also to compose reusable knowledge when multiple modeling structures must coexist in the same formulation.

Solver-based Evaluation. We report solving accuracy as the primary metric. A generated solution is counted as correct only when the produced optimization program can be executed and satisfies the benchmark specific correctness criterion. For an evaluation set $\mathcal{B}$, we compute

$$\mathrm{Acc}(\mathcal{B}) = \frac{1}{|\mathcal{B}|} \sum_{p \in \mathcal{B}} \mathbb{I}[\mathrm{Correct}(p)]. \tag{7}$$

The overall result is the unweighted average over the six benchmarks.

For OptWeaver, solver feedback may be used to revise an unsuccessful formulation for at most three rounds. The repair is restricted to the current problem and does not alter the persistent skill library. An instance is marked as unsuccessful when no correct formulation is obtained within this budget.

Evaluation Isolation. The skill library is frozen before evaluation begins. Test instances do not participate in skill evolution and cannot access reference-guided validation. Solver feedback is available only for local diagnosis and repair of the current formulation. The same frozen library is therefore reused across the complete evaluation set, preventing test problems from becoming additional sources of persistent modeling knowledge.

A.2 Baseline Details

Comparison Methods. Our baselines cover three complementary ways of obtaining optimization modeling capability. The first group consists of general-purpose or reasoning-oriented LLMs, including GPT-4o (Hurst et al., 2024), OpenAI-o3 (Singh et al., 2025), DeepSeek-V3 (Liu et al., 2024), and DeepSeek-R1 (Guo et al., 2025). These models provide a reference for the optimization capability available without explicit external skill learning.

The second group contains optimization-specific trained models. ORLM (Huang et al., 2025a) and LLMOPT (Shu et al., 2025) acquire domain-specific modeling capability through supervised training, while OptMATH (Lu et al., 2025) expands optimization training data through scalable synthesis. SIRL (Chen et al., 2026b) further incorporates solver-grounded reinforcement learning. These methods represent approaches in which optimization knowledge is primarily absorbed into model parameters.

The third group improves optimization modeling at inference time without updating the underlying model. CoT (Wei et al., 2022) provides explicit intermediate reasoning, CoE (Xiao et al., 2024) decomposes the problem across specialized agents, OptiMUS (Ahmaditeshnizi et al., 2024) combines model construction with iterative solver interaction, and MCTS (Astorga et al., 2025) searches over

alternative formulations. AlphaOPT (Kong et al., 2026) and OptiTree (Liu et al., 2026b) additionally retain knowledge from previously solved problems. AlphaOPT maintains solver-verified modeling insights, while OptiTree organizes reusable modeling thoughts through a hierarchical problem structure.

AlphaOPT and OptiTree are particularly relevant comparisons because they also reuse accumulated modeling knowledge. OptWeaver differs in how this knowledge is organized after acquisition. Rather than treating reusable knowledge primarily as retrieval context, it associates Modeling Skills with the modeling decisions and mathematical dependencies through which they participate in a new formulation.

A.3 Training and Inference Configuration

Learning Configuration. DeepSeek-V3 is used as the default backbone unless a different model is explicitly specified. The main knowledge learning setting uses 400 training problems. Every run begins from the same initial library described in Appendix A.4, and independently evolves its own skill system. Skills acquired in one experimental run are not reused by another configuration unless the experiment explicitly studies knowledge transfer.

For each training problem, OptWeaver identifies the required modeling elements and activates relevant skills from the current library. These skills are adapted to the problem and composed into the task specific Modeling Graph before solver code is produced. Unsuccessful formulations are traced backward through this construction structure and repaired at the responsible modeling decision. A correction contributes to persistent knowledge only after the revised formulation has been validated by the solver.

During skill evolution, newly validated knowledge can refine an existing skill, modify its applicability, change its relations with other skills, merge with related knowledge, or introduce a new skill. The choice depends on how the correction relates to the current skill system rather than on the identity of the training instance.

Fallback Validation. Reference information is used only during training and only when autonomous diagnosis cannot resolve a problem within the allowed repair budget. Its role is to expose the unresolved difference between the current model and a valid formulation. The reference solution is not directly substituted for the generated model. Any correction identified with reference assistance must still be applied to the current construction and verified by the solver before it can affect the persistent library.

Inference Configuration. After training, the final Modeling Skill Library is fixed. An unseen problem follows the same forward construction process used during learning, including skill selection, task specific adaptation, Modeling Graph construction and solver code generation. Solver feedback can trigger local backward attribution and repair for up to three rounds.

Persistent evolution is disabled throughout evaluation. In particular, test-time corrections are not written back into the skill library, and reference-guided validation is unavailable. The evaluation therefore measures whether the knowledge acquired during training can be reused on new optimization problems rather than whether the system can continue learning from the test set.

A.4 Initial Skill Library

Initialization. All main experiments start from eight general Modeling Skills that provide coarse coverage of common optimization problem families. The initialization is intentionally compact so that most problem specific and structural knowledge must be acquired through subsequent solver grounded learning.

The initial library should be interpreted as a set of broad starting points rather than a fixed taxonomy. A training problem may require modeling structures that are only partially covered by the retrieved problem-family skill and the shared modeling unit skill, or may contain reusable knowledge not yet represented in the library. After a validated correction, OptWeaver can update the implicated skill content and introduce new knowledge only when no suitable representation exists. This may involve creating a new problem-family skill or adding a new modeling unit to the shared modeling unit skill.

Table 2: Initial problem-family skills used before knowledge learning.

Skill	Covered Modeling Pattern
LP	Linear programming
IP	Integer programming
VRP	Vehicle routing
Scheduling	Scheduling structures
Assignment	Assignment structures
Knapsack	Capacity constrained selection
Network Flow	Network flow structures
Facility Location	Facility location structures

This design also prevents library growth from becoming equivalent to memorizing training problems. New instances do not automatically produce new skills. Persistent changes occur only when a correction exposes reusable knowledge that meaningfully extends the existing skill system. We examine how this library evolves as the amount of training data increases in Appendix B.3.

B Additional Experimental Results

This section provides additional experimental analyses of OptWeaver, including detailed component ablations, skill transferability, the effects of training scale and data source, computational cost and skill coverage across different problem structures. These experiments complement the main results by examining how the learned Modeling Skills are formed and under which conditions they can be effectively reused.

B.1 Detailed Ablation Results

We first provide the complete ablation results for the major components of OptWeaver. All variants use the same backbone and evaluation setting, with only the corresponding component removed.

Table 3: Ablation results on four representative benchmarks. Avg. denotes the unweighted average over the four benchmarks.

Variant	MAMO-C	IndustryOR	OptMATH	NL4Opt	Avg.
w/o Problem-Family Skill Selection	**85.2**	63.0	44.2	**97.6**	72.50
w/o Task-specific Adaptation	84.7	62.0	45.8	95.9	72.10
w/o Modeling Graph and Modeling Unit Skill	81.8	63.0	40.4	95.9	70.28
w/o Backward Attribution and Repair	75.4	62.0	33.1	93.5	66.00
OptWeaver	81.8	**70.0**	**47.6**	98.0	**74.35**

The results reveal different roles for the forward and backward components. Removing problem-family skill Selection or task-specific adaptation does not uniformly reduce performance on every benchmark. In particular, simpler selection strategies can occasionally perform better on a single problem distribution. However, their lower average performance indicates weaker robustness across heterogeneous modeling structures.

Removing both the Modeling Graph and the modeling unit skill reduces average performance, with larger declines on IndustryOR and OptMATH. This result is consistent with their complementary roles in providing reusable unit-level modeling knowledge and explicitly organizing dependencies among modeling decisions. Among the tested ablations, removing Backward Attribution and Repair produces the largest average degradation. This result is consistent with the intended role of the backward module, which traces solver feedback through the construction structure to localize failures and guide targeted repair.

B.2 TRANSFERABILITY OF LEARNED SKILLS

A useful Modeling Skill should remain valuable beyond the model and solving procedure from which it was acquired. We therefore isolate the learned skills from the remaining OptWeaver components and evaluate transfer across model families, model scales and modeling frameworks. During these experiments, the transferred skills are provided only as external modeling knowledge. Modeling element extraction, task-specific guidance construction, Modeling Graph construction, backward repair and persistent evolution are disabled.

Transfer across Model Families. We first construct a skill library using GPT-5.5 on 200 training problems and transfer the resulting knowledge to DeepSeek-V3. The target model is evaluated either directly or with the transferred skills while all other inference conditions remain unchanged.

Table 4: Transfer of Modeling Skills from GPT-5.5 to DeepSeek-V3.

Setting	MAMO-C	IndustryOR	OptMATH	Avg.
DeepSeek-V3	50.2	37.0	**44.0**	43.73
DeepSeek-V3 + Skill	**70.9**	**52.0**	42.2	**55.03**

The transferred skills provide substantial gains on MAMO ComplexLP and IndustryOR even though the target model never participates in the original skill learning process. Performance on OptMATH does not improve, showing that transfer depends on whether the acquired knowledge covers the structures required by the target problems. The clear average improvement nevertheless indicates that reusable modeling knowledge can move across model families rather than remaining tied to the original generator.

Transfer across Model Scales. We next examine whether knowledge learned by a substantially larger model can support a smaller model from a related family. Qwen3.5-397B-A17B is used to acquire the skills, while Qwen3.6-35B-A3B is used for evaluation.

Table 5: Transfer of Modeling Skills from a large Qwen model to a smaller Qwen model.

Setting	MAMO-C	IndustryOR	OptMATH	Avg.
Qwen3.6-35B-A3B	44.8	38.0	28.0	36.94
Qwen3.6-35B-A3B + Skill	**71.9**	**44.0**	**51.0**	**55.64**

The smaller model benefits strongly from the transferred skills across all three benchmarks. This suggests that part of the modeling capability exposed by a high capacity model can be represented externally as reusable knowledge and subsequently applied by a much smaller model.

Transfer across Modeling Frameworks. We finally introduce OptWeaver learned skills into AlphaOPT while preserving the original AlphaOPT solving procedure. This setting tests whether the learned knowledge remains useful when the surrounding reasoning and retrieval framework is changed.

Table 6: Transfer of learned Modeling Skills to AlphaOPT.

Setting	MAMO-C	IndustryOR	OptMATH	Avg.
AlphaOPT	61.6	49.0	39.2	49.93
AlphaOPT + Skill	**65.2**	**56.0**	**41.4**	**54.20**

The improvement persists even though the original OptWeaver construction and repair mechanisms are absent. This provides complementary evidence that the learned skills encode reusable modeling principles rather than framework specific intermediate states.

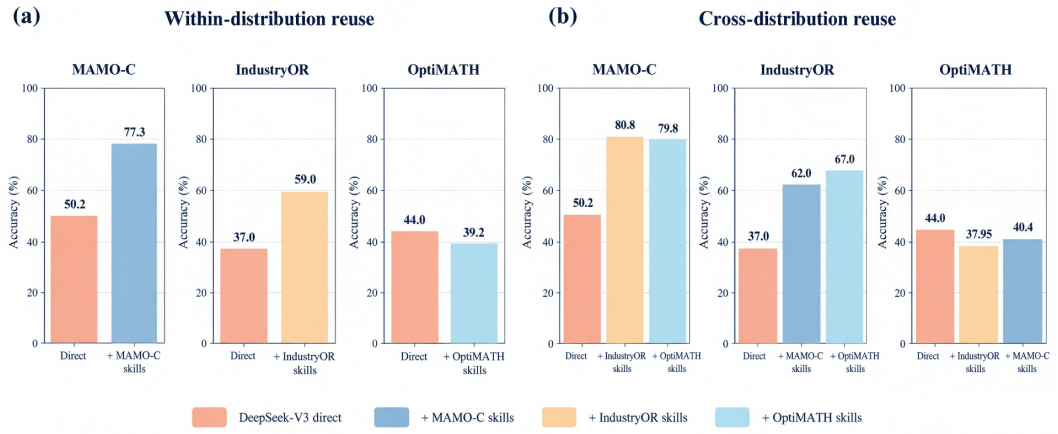


Figure 5: Effect of training data source on Modeling Skill reuse. (a) Within-distribution reuse, where skills are acquired from 25% of each benchmark and evaluated on the remaining 75%. (b) Cross-distribution reuse, where the skill library acquired from one dataset is frozen and directly transferred to the other benchmarks. DeepSeek-V3 direct performance is shown as the baseline.

B.3 Effect of Training Scale

We study how the amount of knowledge acquisition data affects both modeling performance and the evolution of the skill library. All settings use DeepSeek-V3 and the same initialization. The training sets are constructed in a nested manner so that larger settings extend the smaller ones rather than replacing them with unrelated samples.

Table 7: Effect of the number of training problems on modeling performance and the size of the resulting skill library.

Training Size	MAMO-C	IndustryOR	OptMATH	Avg.	# Skills
100	74.4	62.0	31.3	55.90	18
150	73.9	67.0	34.9	58.60	20
200	76.9	69	35.5	60.46	27
400	**81.8**	**70.0**	**47.6**	**66.45**	42

Increasing the number of training problems generally improves average performance while also expanding the learned skill system. The improvement is not strictly monotonic on every benchmark, which suggests that the benefit is determined by the modeling structures covered by the additional data rather than by data volume alone.

The library grows from the eight initial skills to 42 skills in the largest setting. We do not interpret this growth itself as the cause of improved performance. Instead, the larger training collection exposes a broader range of reusable modeling patterns and failure modes, allowing the skill system to cover more structures that may recur during evaluation.

B.4 Effect of Training Data Source

Training scale determines how much knowledge can be acquired, while the source of the training problems determines which modeling structures are exposed. We construct separate skill libraries from IndustryOR, MAMO ComplexLP and OptMATH. We then evaluate their reuse within the source distribution and across different data distributions.

Within-distribution Reuse. For each dataset, 25% of the instances are used for knowledge acquisition and the remaining 75% are used for evaluation. Figure 5(a) compares within-distribution reuse against the DeepSeek-V3 direct baseline.

Knowledge acquisition produces clear gains on MAMO ComplexLP and IndustryOR, but the same behavior does not appear on OptMATH. This difference indicates that simply learning from problems drawn from the same benchmark does not guarantee improvement. The usefulness of the resulting skills depends on whether recurring structures can be extracted and reused reliably.

Cross-distribution Reuse. We next freeze the library learned from one dataset and directly apply it to the other benchmarks without additional persistent learning. Figure 5(b) shows the resulting cross-distribution reuse across the three benchmarks.

The transfer pattern is strongly asymmetric. Knowledge learned from IndustryOR or OptMATH transfers well to MAMO ComplexLP, while knowledge transferred to OptMATH is considerably less effective. This suggests that transfer is governed more by overlap in underlying modeling structures than by dataset identity. A source containing reusable allocation, capacity or dependency structures can support another benchmark when these patterns recur, even if the natural language domains differ.

B.5 Computational Cost

OptWeaver introduces additional computation during knowledge acquisition because a training problem may require skill adaptation, model construction, solver feedback, repair and persistent skill evolution. We analyze where this cost occurs and how it differs from subsequent skill reuse.

Knowledge Acquisition Cost. Table 8 reports token consumption for the 200 problem training setting. Both input and output tokens are included.

Table 8: Token consumption during knowledge acquisition with 200 training problems.

Stage	Total Tokens	Avg. Tokens / Instance	Share
Modeling Element Extraction	1.22M	6.09K	5.06%
Problem Type Recognition	0.37M	1.86K	1.55%
Task-specific Skill Construction	3.31M	16.54K	13.74%
Modeling Graph Construction	3.18M	15.92K	13.22%
Code Generation	5.36M	26.79K	22.26%
Backward Attribution / Repair	3.21M	16.05K	13.34%
Reference-Guided Validation	1.29M	6.45K	5.36%
Skill Evolution	6.13M	30.67K	25.48%
Total	**24.07M**	**120.36K**	**100%**

The cost is concentrated in model construction and persistent knowledge formation rather than in preliminary problem recognition. Skill Evolution accounts for the largest individual share, while code generation is the second largest component. Reference-Guided Validation and Skill Evolution together account for roughly one third of the training cost. These operations are directly associated with acquiring and maintaining persistent modeling knowledge.

Cost of Reuse. Once training is complete, persistent evolution and reference-guided validation are no longer executed. On the IndustryOR evaluation set, 80 problems consume approximately 7.89M tokens in total, corresponding to about 98.66K tokens per problem. This is lower than the average training cost, although the difference should not be attributed entirely to skill reuse because the training and evaluation problems differ in complexity.

A more direct distinction is that persistent knowledge acquisition incurs one time operations that are absent during deployment. Let T_{evo} denote the total token cost of Skill Evolution. For the 200 problem training run,

$$T_{\text{evo}} = 6{,}133{,}306. \tag{8}$$

If the learned library is subsequently reused for n problems, its amortized evolution cost per problem is

$$\frac{T_{\text{evo}}}{n}. \tag{9}$$

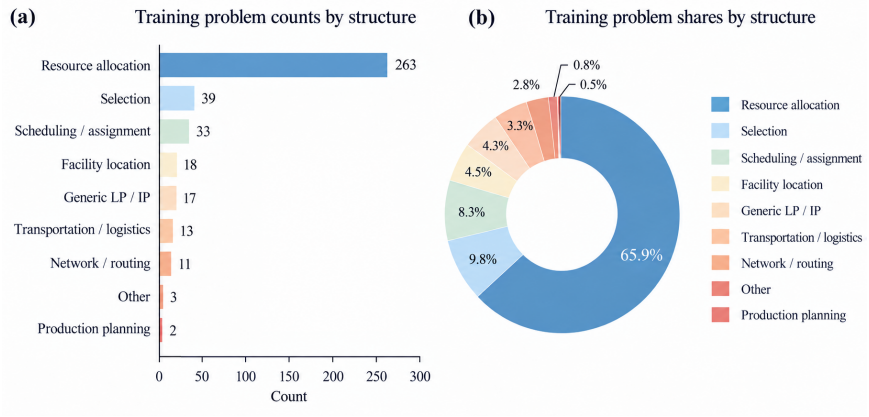


Figure 6: Structural distribution of the 400 problems used for knowledge acquisition.

OptWeaver therefore follows a front loaded cost structure in which additional computation is spent to acquire reusable modeling knowledge, while the persistent knowledge formation process does not need to be repeated for every future problem.

B.6 Skill Coverage and Problem Structure Analysis

We finally examine what types of modeling knowledge are most strongly represented in the learned skill system and how this coverage relates to performance across different problem structures. The categories below are semantic groupings introduced only for analysis and are not additional labels used internally by OptWeaver.

Training Knowledge Distribution. Figure 6 shows the structural distribution of the 400 training problems across broad modeling families.

The training distribution is strongly concentrated on allocation related structures. This tendency is also reflected in the evolved library. Among the 34 additional problem family skills produced beyond the eight initial skills, 17 can be grouped as allocation related patterns. The learned knowledge therefore provides substantially denser coverage for allocation style modeling than for temporal or scheduling structures.

Performance across Problem Structures. To examine whether this difference appears during evaluation, we group test problems from MAMO ComplexLP, IndustryOR and OptMATH according to their dominant modeling structure. Figure 7 reports the resulting solve rates across these structural groups.

The strongest performance appears on resource allocation, transportation and production-planning structures. Resource allocation is also the dominant structure in the training distribution, while the strong results on transportation and production planning suggest that some learned modeling knowledge transfers beyond the most frequent training families. Scheduling and assignment problems remain substantially more difficult. These problems often require several decisions to be coupled through temporal ordering, precedence, mutual exclusion, or state transitions, placing greater demands on the composition of multiple skills.

Fine-grained Structural Analysis. Table 9 provides a more detailed breakdown of representative problem types.

The detailed results reinforce the same pattern. OptWeaver performs reliably when a problem is dominated by allocation, transportation, capacity or familiar integer structures. Performance becomes weaker when several temporal and logical dependencies must be coordinated simultaneously. In particular, precedence constrained scheduling and joint assignment scheduling structures remain difficult despite the presence of individually relevant skills.

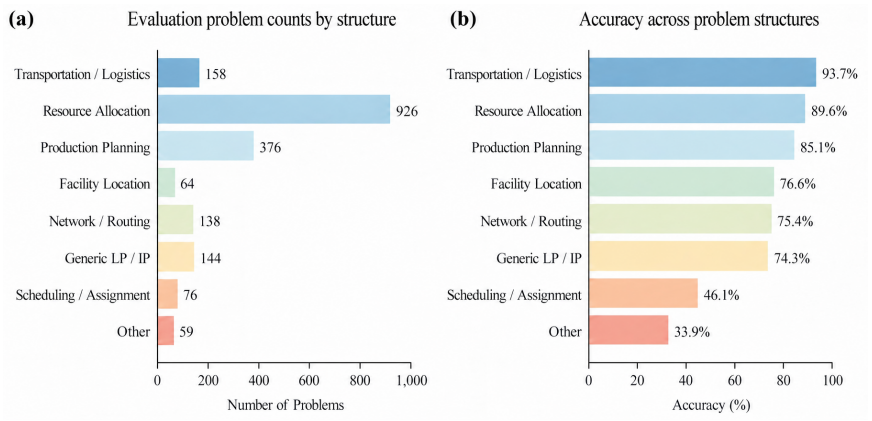


Figure 7: Performance of OptWeaver across broad modeling structures, with the number of evaluation problems shown for each structure.

Table 9: Fine-grained solve rates across representative optimization problem structures.

Problem Type	# Problems	# Solved	Solve Rate
VRP	52	50	96.2%
Integer transportation with ratio constraints	152	142	93.4%
Integer resource allocation	524	484	92.4%
Integer resource allocation with dependencies	84	77	91.7%
Integer resource allocation with bounds	105	95	90.5%
Integer production planning	373	317	85.0%
Continuous resource allocation	79	59	74.7%
LP	140	104	74.3%
Network flow	84	52	61.9%
Precedence constrained scheduling	54	29	53.7%
Other	59	20	33.9%
Binary assignment with scheduling	10	0	0.0%

Understanding the Difficulty of OptMATH. This structural perspective also helps explain the relatively weaker performance on OptMATH. Many OptMATH problems combine multiple variable types with precedence relations, linking constraints, temporal interactions and several constraint families in the same formulation. The main challenge is therefore not simply the absence of a relevant individual skill. Instead, several local modeling principles must be bound to the correct objects and woven together without introducing inconsistent dependencies.

This observation highlights an important limitation of the current skill system. The learned library provides strong coverage for frequently observed local structures, but composing several less frequent temporal and logical patterns remains substantially harder. Improving knowledge acquisition for such underrepresented compositions is a promising direction for future work.

C Algorithm Details

This section provides the complete procedures underlying the forward and backward processes described in Section 3. We follow the same notation as the main text. Given a problem p and the current Modeling Skill Library $\mathcal{S}_t$, forward skill weaving identifies the skills used for p, adapts their reusable knowledge into temporary task-specific guidance, and constructs the task-specific Modeling Graph G_p and executable solver code c_p. Solver feedback r_p is then used for backward attribution, targeted repair and extraction of reusable knowledge Δk_p. During learning, the validated knowledge is integrated into the persistent library to obtain $\mathcal{S}_{t+1}$.

C.1 FORWARD SKILL WEAVING

Forward skill weaving adapts reusable Modeling Skills into temporary task-specific guidance and uses this guidance to construct a task-specific formulation for the current problem.

$$(S_p, G_p, c_p) = \mathcal{F}(p, \mathcal{S}_t). \tag{10}$$

where $S_p \subseteq \mathcal{S}_t$ denotes the skills used for p, rather than the task-specific guidance itself. The procedure contains three closely connected operations. OptWeaver first identifies the modeling requirements of p and and uses them to identify the problem class. It then retrieves the corresponding problem-family skill and combines its knowledge with the shared modeling unit skill to construct temporary task-specific guidance. The resulting guidance is instantiated into concrete modeling decisions, which are organized through their mathematical dependencies in the Modeling Graph. The resulting structure is finally instantiated as executable solver code.

From Reusable Knowledge to Task-Specific Guidance. OptWeaver parses p to identify the decision variables, parameters, constraints, objectives and semantic relations required by the problem. These modeling requirements serve as task-specific anchors for identifying the problem class. OptWeaver retrieves the corresponding problem-family skill from $\mathcal{S}_t^{\mathrm{PF}}$, while the shared modeling unit skill provides reusable construction knowledge at the level of variables, constraints, and objectives. We denote the skills used for the current problem by S_p. Their reusable knowledge is adapted to the concrete modeling objects of p, producing temporary task-specific guidance.

The adaptation step is important because a reusable skill typically describes a general modeling rule rather than a problem-specific expression. For example, a generic activation skill must be bound to the particular binary decision and dependent variable appearing in p before it can contribute to the formulation. The resulting task-specific guidance is used only for the current problem and does not modify the persistent skill library.

Dependency-aware Graph Construction. The temporary task-specific guidance is instantiated into concrete modeling decisions, which are organized in the task-specific Modeling Graph

$$G_p = (N_p, D_p), \tag{11}$$

where N_p contains the task-specific modeling decisions instantiated for variables, constraints and objectives, and D_p records their mathematical dependencies.

For each modeling decision, OptWeaver records both the resulting mathematical object and the skill that supports it. Dependencies are then introduced according to how these objects participate in the formulation. A linking constraint, for example, connects an activation variable with a dependent decision, while the same decision may later participate in capacity or assignment constraints. These dependencies are explicitly represented in G_p and is reused during backward attribution, while the construction structure established during forward weaving is reused during backward attribution.

If a required local structure is not directly covered by the current library, OptWeaver can derive the required task-specific modeling decision directly from the problem and incorporate it into G_p. Such a structure is not immediately added to $\mathcal{S}_t$. The resulting decision remains local to the current problem unless subsequent validation shows that its underlying modeling principle is reusable, in which case the principle can be incorporated into the persistent skill system during knowledge evolution.

Solver Code Instantiation. Once the Modeling Graph is complete, OptWeaver generates c_p from G_p. The Modeling Graph specifies the required mathematical structure, while code generation focuses on its solver implementation. This separation allows errors in the mathematical structure to be distinguished from errors introduced during implementation.

Algorithm 1 summarizes the complete forward procedure.

C.2 BACKWARD ATTRIBUTION AND SKILL EVOLUTION

The backward process starts from solver execution,

$$r_p = \mathrm{Solver}(c_p). \tag{12}$$

Algorithm 1 Forward Skill Weaving

Require: Problem p, current Modeling Skill Library $\mathcal{S}_t$
Ensure: Skills used for p, S_p, Modeling Graph G_p, solver code c_p
1: Extract the modeling elements required by p
2: Identify the problem class of p from the extracted modeling elements
3: Retrieve the corresponding problem-family skill from $\mathcal{S}_t^{\mathrm{PF}}$
4: Access the shared modeling unit skill s_t^{MU}
5: Let S_p denote the problem-family skill and modeling unit skill used for p
6: Adapt their reusable knowledge to form temporary task-specific guidance H_p
7: Initialize the Modeling Graph G_p
8: **for** each required modeling decision implied by p and H_p **do**
9: Instantiate the corresponding variable, constraint, or objective from H_p in G_p
10: Add the required mathematical dependencies among modeling decisions to G_p
11: **end for**
12: Derive any required modeling decisions not covered by the reusable knowledge directly from p and incorporate them into G_p
13: Generate executable solver code c_p from G_p
14: **return** S_p, G_p, c_p

If the formulation is correct, no persistent update is required. Otherwise, OptWeaver reuses the construction structure established during forward skill weaving to trace the failure backward to the responsible modeling source, performs a targeted repair, and extracts reusable knowledge from the validated correction.

Backward Attribution and Targeted Repair. OptWeaver first determines whether the observed failure originates from the solver implementation or from an earlier modeling decision. It traces the failure backward through c_p, G_p, the temporary task-specific guidance, task-specific bindings, the selected problem-family skill, and the extracted modeling elements. The goal is to identify the earliest decision that is inconsistent with the original problem rather than repairing only the downstream symptom.

The identified source may lie in the extracted modeling elements, a mismatch between the selected problem-family skill and the problem structure, incorrect task-specific binding, a missing or incorrect modeling decision or dependency in G_p, or an implementation error in c_p. OptWeaver modifies the responsible component while preserving unaffected parts of the formulation. Previous unsuccessful modifications for the current problem are recorded so that the same ineffective repair is not repeatedly applied.

After each repair, the revised solver program is executed again. The process continues until the formulation passes verification or the allowed repair budget is exhausted.

Reusable Knowledge Extraction. When a repair is successfully validated, OptWeaver distills reusable modeling knowledge from the diagnosed failure and its successful repair,

$$\Delta k_p = \mathcal{B}(r_p, G_p, S_p). \tag{13}$$

The extracted knowledge removes problem-specific variables, coefficients, and descriptions whenever possible and retains the underlying modeling principle. A correction therefore becomes persistent only when it can be expressed as knowledge that may apply to future problems.

Knowledge Placement and Skill Evolution. During learning, Δk_p is integrated into the persistent skill system through

$$\mathcal{S}_{t+1} = \mathcal{U}(\mathcal{S}_t, \Delta k_p). \tag{14}$$

OptWeaver first determines how the new knowledge relates to the existing library. If the relevant knowledge already exists, OptWeaver updates the implicated skill content, including applicability conditions, modeling rules, or relations, and merges closely related knowledge when appropriate. New knowledge is introduced only when no suitable representation exists. A new problem-family skill is created when no existing one adequately matches the problem, while a new modeling unit

is added when the shared modeling unit skill lacks the required reusable unit-level knowledge. The update therefore changes both the knowledge stored in the library and the structure that determines how this knowledge can participate in future model construction.

If autonomous attribution and repair fail during training, reference-guided validation can be used as a fallback to locate the unresolved discrepancy. The reference solution is not directly inserted into the generated formulation. Any correction identified with reference assistance must still be applied to the current model and pass solver verification before it can produce Δk_p and update $\mathcal{S}_t$.

At test time, the same attribution and local repair procedure can be applied to the current problem, but persistent evolution is disabled. The final library $\mathcal{S}_T$ remains fixed, and no test-time correction is used to update the skill system.

Algorithm 2 summarizes the backward learning procedure.

Algorithm 2 Backward Attribution and Skill Evolution

Require: Problem p, current library $\mathcal{S}_t$, skills used for p, S_p, Modeling Graph G_p, solver code c_p, mode $m \in \{\text{learn}, \text{test}\}$
Ensure: Repaired Modeling Graph G_p, solver code c_p, and updated or unchanged skill library

Execute c_p and obtain $r_p = \text{Solver}(c_p)$
if r_p indicates a correct formulation **then**
  return $G_p, c_p, \mathcal{S}_t$
end if
Initialize the modification history for the current problem
while solver verification fails and the repair budget remains **do**
  Trace r_p backward through the construction structure established during forward weaving
  Localize the responsible modeling source
  Determine whether the error arises from modeling element extraction, problem-family skill matching, task-specific binding, a modeling decision or dependency in G_p, or solver implementation
  Repair the responsible modeling source while preserving unaffected modeling decisions
  Avoid previously unsuccessful modifications recorded for the current problem
  Regenerate the affected downstream structure when necessary
  Execute the revised c_p and update $r_p = \text{Solver}(c_p)$
end while
if solver verification fails and $m = \text{learn}$ **then**
  Use reference information only to locate the unresolved modeling discrepancy
  Apply the corresponding targeted repair and regenerate the affected downstream structure
  Execute the revised c_p and update $r_p = \text{Solver}(c_p)$
end if
if solver verification still fails **then**
  return $G_p, c_p, \mathcal{S}_t$
end if
if $m = \text{test}$ **then**
  return $G_p, c_p, \mathcal{S}_t$
end if
Distill reusable modeling knowledge $\Delta k_p = \mathcal{B}(r_p, G_p, S_p)$
Determine the implicated skill content according to the attribution result and the scope of Δk_p
Evolve the persistent skill library as $\mathcal{S}_{t+1} = \mathcal{U}(\mathcal{S}_t, \Delta k_p)$
return $G_p, c_p, \mathcal{S}_{t+1}$

D Case Study of the OptWeaver Learning Process

This case illustrates how OptWeaver constructs a model during training, traces an incorrect solver outcome back through the construction structure, performs a targeted repair, and converts the verified correction into reusable modeling knowledge. We use a facility-selection instance in which the central difficulty is not the choice of optimization technique, but preserving the directionality of pairwise coefficients throughout modeling. The first construction treats pairwise dispersion as sym-

metric and therefore collapses ordered interactions into unordered pairs. A single backward repair corrects the indexing scope, after which the reconstructed model reaches the reference objective of 49.

Problem

Five candidate locations are available and exactly two facilities must be selected. Let $y_i \in \{0, 1\}$ indicate whether location i is selected. Selecting two facilities activates pairwise dispersion terms with coefficients d_{ij}, and the objective is to maximize the total activated dispersion. Unlike a standard symmetric dispersion problem, the pairwise values are directional. For example,

$$d_{04} = 24, \qquad d_{40} = 25.$$

Thus, choosing locations 0 and 4 contributes both directional terms and yields

$$d_{04} + d_{40} = 49.$$

The reference optimal objective is therefore **49**.

Forward Construction. OptWeaver first grounds the natural-language problem into modeling elements and recognizes it as an integer facility-selection problem. The initial extraction identifies binary facility-selection variables, a cardinality constraint, pair-activation variables, and a pairwise-dispersion objective. It assigns the pair variable z_{ij} to the scope $i < j$, thereby treating each facility pair as unordered.

Extracted Modeling Elements

Variables:

- $y_i \in \{0, 1\}$: whether candidate location i is selected.
- $z_{ij} \in \{0, 1\}$: whether locations i and j are both selected, extracted over pairs satisfying $i < j$.

Constraints:

- Exactly two locations must be selected:

$$\sum_i y_i = 2.$$

- Pair activation is linked to the corresponding selection decisions:

$$z_{ij} \le y_i, \qquad z_{ij} \le y_j, \qquad z_{ij} \ge y_i + y_j - 1, \qquad i < j.$$

Objective:

$$\max \sum_{i<j} d_{ij} z_{ij}.$$

Parameter semantics: d_{ij} denotes the pairwise dispersion associated with locations i and j.

Problem Type Recognition

Problem type: `integer_facility_selection` **Confidence:** 0.95

A relevant skill is retrieved and adapted to the current modeling objects. The resulting temporary skill binds selection decisions to locations, pair-activation decisions to facility pairs, and the objective to activated pairwise dispersion. Because the extracted pair scope is restricted to $i < j$, the temporary construction inherits the same scope when constructing the task-specific Modeling Graph. This task-specific guidance is used only for the current problem and is not written back to the persistent skill library at this stage.

Retrieved Integer Facility Selection Skill

The retrieved skill was previously learned from an integer facility selection problem and provides reusable guidance for constructing facility-selection models. We show only the parts relevant to the current problem.

Relevant Skill Excerpt.

- Represent facility decisions using integer or binary variables with domains consistent with the problem semantics.
- Preserve the facility types, indices, and problem parameters exactly when instantiating the formulation.
- Construct constraints from the problem-specific relations among the selected facilities rather than reusing source-problem constraints directly.
- Attach all problem-specified contribution terms to the objective with their original coefficients and optimization direction.
- During validation, check variable domains, constraint structure, objective composition, and parameter accuracy before attributing a failure to solver implementation.

Skill Activation and Temporary Adaptation

The retrieved facility-selection skill is then adapted to the current problem by binding its reusable rules to the extracted modeling objects.

Task-Specific Binding.

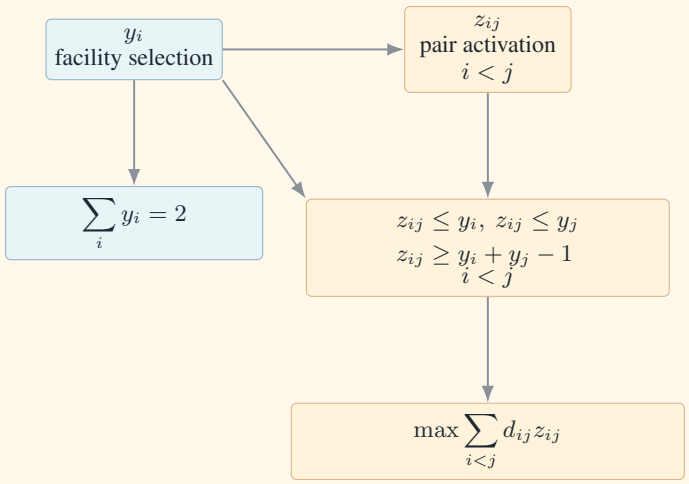
$$\begin{aligned} \text{facility decision} &\mapsto y_i, \\ \text{joint selection of a facility pair} &\mapsto z_{ij}, \\ \text{pairwise contribution} &\mapsto d_{ij} z_{ij}. \end{aligned}$$

Temporary Modeling Guidance. For the current problem, the adapted skill specifies that:

- y_i represents whether candidate location i is selected
- z_{ij} is activated only when both corresponding facilities are selected
- pair-activation variables are linked to facility decisions before contributing to the dispersion objective
- the current formulation follows the pair scope supplied by the extracted modeling elements, namely $i < j$

The resulting Modeling Graph records the dependencies among the instantiated modeling decisions. In the initial graph, facility-selection decisions are linked to the pair-activation variable z_{ij}, which is instantiated only for $i < j$ and then contributes to the pairwise-dispersion objective.

Initial Modeling Graph



Initial pair scope. The pair variable z_{ij}, its linking constraints, and the corresponding objective terms are all instantiated only for $i < j$.

The graph is then realized as solver code. The relevant implementation pattern is equivalent to constructing only unordered pairs and attaching one dispersion coefficient to each activated pair.

Initial Solver Realization

```
unordered_pairs = [(i, j) for i in nodes for j in nodes if i < j]
z = {p: model.addVar(vtype=GRB.BINARY) for p in unordered_pairs}

for i, j in unordered_pairs:
    model.addConstr(z[i,j] <= y[i])
    model.addConstr(z[i,j] <= y[j])
    model.addConstr(z[i,j] >= y[i] + y[j] - 1)

model.setObjective(
    gp.quicksum(d[i,j] * z[i,j] for i,j in unordered_pairs),
    GRB.MAXIMIZE
)
```

Under this formulation, only the ten coefficients with $i < j$ are representable. Their largest value is 24, whereas the correct directional formulation can obtain 49 by selecting locations 0 and 4 and counting both d_{04} and d_{40}.

Backward Attribution and Repair. The incorrect objective triggers the training-time backward procedure. Rather than treating the solver code as the sole source of failure, OptWeaver traces the discrepancy through the construction provenance. The diagnosis identifies the pair-indexing decision in the extracted modeling elements as the responsible upstream error: the source problem supplies asymmetric values for ordered pairs, while the extracted variable z_{ij}, its linking constraints, and the objective all use unordered pairs.

Backward Attribution

Observed issue: the constructed model discards reverse-direction dispersion terms.

Responsible modeling decision:

pair scope of z_{ij} : $i < j$

Attributed error class: `extracted_units_fix`.

Repair plan: replace unordered pairs with ordered pairs so that the variable definition, linking constraints, and objective preserve asymmetric values for both (i, j) and (j, i).

Only the faulty scope is changed. The facility-selection variable, cardinality requirement, pair-activation logic, and objective role remain unchanged. This makes the repair both local and structurally traceable.

Targeted Repair

Before repair

$z_{ij},\ i < j$
$z_{ij} \le y_i,\ z_{ij} \le y_j,\ i < j$
$z_{ij} \ge y_i + y_j - 1,\ i < j$
$\max \sum_{i<j} d_{ij} z_{ij}$

After repair

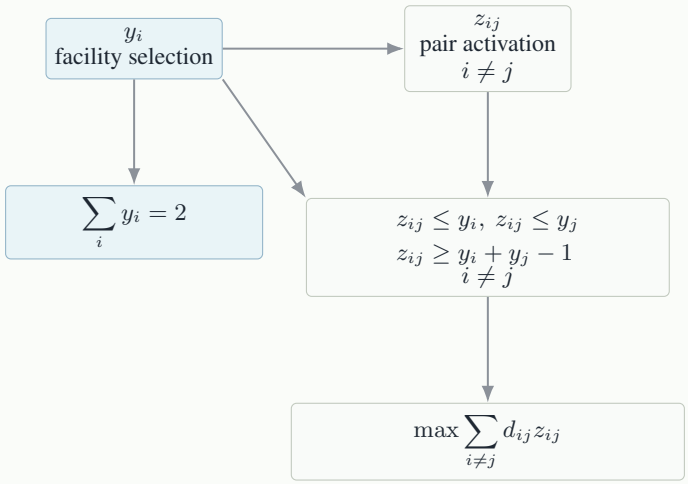
$z_{ij},\ i \neq j$
$z_{ij} \le y_i,\ z_{ij} \le y_j,\ i \neq j$
$z_{ij} \ge y_i + y_j - 1,\ i \neq j$
$\max \sum_{i \neq j} d_{ij} z_{ij}$

Because the repair changes an upstream modeling element rather than only the code, OptWeaver reconstructs the Modeling Graph and regenerates the solver program. The repaired graph instantiates directional pair activations, allowing both z_{04} and z_{40} to contribute to the objective. The resulting model selects locations 0 and 4 and obtains

$$d_{04} z_{04} + d_{40} z_{40} = 24 + 25 = \mathbf{49},$$

matching the reference objective. The training trace therefore terminates successfully after **one backward repair**.

Reconstructed Modeling Graph



Updated pair scope. After repair, the pair variable z_{ij}, its linking constraints, and the objective terms are all reconstructed over ordered pairs satisfying $i \neq j$, thereby preserving the directional dispersion coefficients.

Regenerated Code and Verification

```
ordered_pairs = [(i, j) for i in nodes for j in nodes if i != j]
z = {p: model.addVar(vtype=GRB.BINARY) for p in ordered_pairs}

for i, j in ordered_pairs:
    model.addConstr(z[i,j] <= y[i])
    model.addConstr(z[i,j] <= y[j])
    model.addConstr(z[i,j] >= y[i] + y[j] - 1)

model.setObjective(
    gp.quicksum(d[i,j] * z[i,j] for i,j in ordered_pairs),
    GRB.MAXIMIZE
)
```

Training-Only Knowledge Evolution. Once the repaired formulation is verified, the correction becomes a reusable training signal rather than remaining a problem-specific patch. The reusable lesson is not the numerical pair $(0, 4)$, but the modeling condition that determines how pair variables should be indexed: directionality in coefficient data must be preserved by the corresponding modeling objects and objective terms. This verified correction is passed to the post-repair evolution stage, where OptWeaver can place it into the persistent skill and unit system according to its modeling role.

Reusable Knowledge from the Verified Repair

Problem-specific correction

$$z_{ij} : i < j \quad \Longrightarrow \quad z_{ij} : i \neq j.$$

Reusable modeling principle

When pairwise coefficients are directional or asymmetric, preserve ordered-pair indexing throughout variable construction, linking constraints, dependency construction, and objective assembly. Do not collapse the relation to unordered pairs.

Training versus Test-Time Use. This case expands the training path because training is where backward attribution, repair, and persistent knowledge evolution are enabled. At test time, the learned skill system is frozen. OptWeaver uses the accumulated knowledge for forward construction, but does not update the persistent skill system from test-instance outcomes.

Test-Time Setting

Training: forward construction → backward repair → verified knowledge evolution
Test: frozen skill system → forward construction only

Overall, this case shows why OptWeaver retains explicit construction provenance during training. A seemingly small extraction decision about whether a pair is ordered or unordered propagates through temporary skill binding, Modeling Graph construction, and solver realization. Backward attribution makes this propagation traceable, allowing the system to repair the responsible upstream modeling decision rather than patching only its downstream code manifestation. The verified correction can then be retained as structural modeling knowledge for subsequent problems.

E Case Studies of Modeling with Learned Skills

We present two representative cases to illustrate how learned Modeling Skills affect concrete modeling decisions. To isolate the effect of the learned knowledge, we follow the transfer setting in Appendix B.2. DeepSeek-V3 is used either directly or with the learned skills provided as external modeling guidance, while the remaining OptWeaver inference components are disabled. The two cases highlight two complementary modeling difficulties arising from correctly scoping advanced formulation techniques and consistently composing modeling components across periods.

E.1 Using Advanced Modeling Knowledge

The first case is a job-shop scheduling instance in which binary sequencing variables and a sufficiently large constant are already specified. The key challenge is therefore not recognizing the Big-M technique itself, but determining which operation pairs require the corresponding disjunctive constraints. Such pairwise relations are easy to overgeneralize, leading to unnecessary or incorrect constraints when Big-M is applied beyond its intended scope.

Problem

Job 0 contains five operations with processing times $4, 1, 6, 6$, and 8, while Job 1 contains four operations with processing times $9, 1, 4$, and 2. Operations within each job must follow their specified precedence relations. In addition, only particular operation pairs share a machine. For example, Job 0–Operation 1 conflicts with Job 0–Operation 3, and Job 0–Operation 1 conflicts with Job 1–Operation 2. These pairs require explicit sequencing decisions so that the

corresponding operations cannot overlap. The optimization objective is

$$\min T,$$

where T denotes the makespan. The reference optimal makespan is **25**.

Direct Modeling. DeepSeek-V3 correctly recognizes the need for precedence constraints, makespan minimization, and Big-M-based disjunctive sequencing. However, it applies the sequencing formulation too broadly, expanding the conflict relation to essentially every operation pair rather than restricting it to true shared-machine conflicts. This implicitly treats otherwise independent operations as if they were competing for the same machine, introducing unsupported ordering dependencies and unnecessarily restricting the feasible scheduling space. As a result, the Direct formulation yields a makespan of 40 instead of the reference value of 25.

The failure therefore does not arise from missing the appropriate modeling technique, but from correctly identifying it while incorrectly determining its scope of application.

Direct Modeling

$$\mathcal{P}_{\text{Direct}} = \{(i,j) \mid i < j,\ i,j \in \mathcal{O}\}.$$

For every $(i,j) \in \mathcal{P}_{\text{Direct}}$,

$$\begin{aligned} &s_i + p_i \le s_j + M(1 - z_{ij}), \\ &s_j + p_j \le s_i + M z_{ij}, \end{aligned} \qquad M = 100000.$$

Modeling with Learned Skills. With the learned skill provided as external guidance, DeepSeek-V3 retains the same high-level disjunctive formulation but changes how it is bound to the problem structure. Instead of applying sequencing logic globally, the model restricts binary ordering variables and Big-M no-overlap constraints to the actual shared-machine conflict set, while leaving nonconflicting operations independent except for their original precedence relations. The resulting formulation is substantially smaller and recovers the correct makespan of 25. The improvement therefore does not come from introducing a new modeling technique, but from applying an existing one at the correct structural scope. In this sense, the learned skill provides reusable guidance about *when and where* a formulation pattern should be instantiated.

DeepSeek-V3 + Learned Skill

$$\mathcal{C} = \{((J_0,O_1),(J_0,O_3)),\ ((J_0,O_1),(J_1,O_2))\}.$$

For each actual conflict $(i,j) \in \mathcal{C}$,

$$\begin{aligned} s_j &\ge s_i + p_i + \delta_{ij} - M(1 - z_{ij}), \\ s_i &\ge s_j + p_j + \delta_{ji} - M z_{ij}. \end{aligned}$$

Table 10: Comparison of modeling decisions in the job-shop scheduling case.

	DeepSeek-V3	DeepSeek-V3 + Learned Skill
Formulation technique	Big-M sequencing	Big-M sequencing
Conflict identification	All operation pairs	Shared-machine pairs only
Dependency structure	Unsupported global ordering	Conflict-specific ordering
Model size	46 variables / 88 constraints	14 variables / 22 constraints
Makespan	40	25

Thus, the error is structural rather than syntactic, the model selects a valid advanced formulation, but binds it to the wrong modeling objects. The learned scheduling skill directly addresses this issue by specifying the structural conditions under which ordering and no-overlap logic should be introduced.

Relevant Learned Skill

The learned scheduling skill provides reusable modeling guidance on when and where sequencing logic should be introduced.

- Introduce binary ordering and disjunctive no-overlap constraints only for explicit machine conflicts or sequencing choices.
- Keep unrelated operations uncoupled, except for required predecessor–successor relations.
- Link makespan to operation completion times after the relevant precedence and conflict structure has been established.

The key contribution of the transferred skill is therefore not a reusable code template, but a modeling principle for determining where a formulation pattern should be instantiated within the problem structure. Operation identities, processing times, and machine assignments may change, while the same principle remains applicable. This case shows that reusable optimization knowledge should capture not only which modeling technique is relevant, but also the structural conditions governing its application.

E.2 Enhancing Structured Modeling

The second case is a six-month production and workforce planning problem involving workforce management, production, outsourcing, inventory, sales, and backorders. Unlike the first case, the Direct program is executable and Gurobi successfully reaches an optimal solution. The challenge therefore lies not in solver compatibility, but in assigning different quantities appropriate semantic roles and composing them consistently through cross-period dependencies.

Problem

A manufacturer of foldable tables must construct a monthly production and workforce plan from January to June. The decisions include workforce size, hiring, firing, in-house production, overtime, outsourcing, inventory, and order fulfillment.
The initial workforce is 1000 employees and the initial inventory is 15000 units. Each in-house unit requires five labor hours. Each worker provides 160 regular hours per month and at most 20 overtime hours. Unfulfilled demand is backordered and must be satisfied in later periods.
At the end of June,

$$I_{\text{Jun}} \geq 10{,}000, \qquad B_{\text{Jun}} = 0.$$

The objective is to maximize total net profit, including sales revenue and the costs of raw materials, outsourcing, inventory, backorders, labor, hiring, and firing.
The reference optimal objective value is

$$10349920.$$

Direct Modeling. DeepSeek-V3 identifies the main decision quantities and generates a solver-compatible model. However, several quantities with distinct semantic roles are coupled too tightly. In particular, realized sales are not represented as a separate flow, causing inventory evolution, demand fulfillment, and backlog propagation to be combined within the same cross-period relations. Moreover, all major quantities are assigned the same integer domain despite their different modeling roles. As a result, the mathematical semantics of inventory, fulfilled demand, and backlog are not cleanly separated, yielding an objective value of 35792500.

Although the generated program can be solved, its mathematical structure does not faithfully preserve the intended state–flow semantics. This case therefore exposes a different failure mode from simple code invalidity, where the relevant components and temporal relations are present but organized inconsistently. Solver executability alone does not guarantee structural correctness.

Direct Modeling

For $t > 1$,

$$I_t = I_{t-1} + P_t^{\text{reg}} + P_t^{\text{ot}} + X_t - D_t + B_{t-1} - B_t.$$

Modeling with Learned Skills. With the learned skill provided as external guidance, DeepSeek-V3 reorganizes the planning problem around explicit state transitions and distinct state–flow roles. Workforce evolves across periods through hiring and firing decisions, while production is linked to the labor capacity provided by the current workforce. More importantly, physical inventory and demand fulfillment are separated into distinct balance relations by introducing realized sales explicitly, allowing inventory and backlog to propagate consistently across consecutive periods.

The resulting formulation does not merely add more constraints, but changes how familiar modeling components are composed. Workforce, production, inventory, sales, and backlog are represented as separate modeling objects and connected through explicit temporal and capacity dependencies. This structurally reorganized model recovers the reference objective value of 10349920.

DeepSeek-V3 + Learned Skill

Workforce transition:

$$W_t = W_{t-1} + H_t - F_t.$$

Production capacity:

$$5P_t \leq 160W_t + OT_t.$$

Inventory balance:

$$I_t = I_{t-1} + P_t + X_t - S_t.$$

Demand and backlog balance:

$$S_t + B_t = D_t + B_{t-1}.$$

The improvement is therefore structural rather than syntactic. The Direct model contains plausible variables and temporal recursions, but several semantic roles are compressed into the same relation. With the learned skill, these roles are separated and connected through explicit dependencies, yielding the structure

$$\text{Workforce} \rightarrow \text{Labor Capacity} \rightarrow \text{Production} \rightarrow \text{Inventory/Sales} \rightarrow \text{Backlog}.$$

The learned skill directly supports this type of decomposition by providing reusable guidance on how variables with different semantic roles should be represented and how balance, capacity, and state-transition relations should be composed.

Relevant Learned Skill

The learned skill provides reusable modeling guidance on how state, flow, and capacity components should be separated and connected.

- Use distinct variable families when quantities play different semantic roles or require different variable domains.
- Use balance and conservation constraints when inventory, flow, demand, or state-transition semantics are present.
- Connect capacity-providing decisions, such as workforce, to the operational quantities they support through explicit capacity relations.
- Attach objective terms to variables whose economic meaning and units match the corresponding revenue or cost.

This example shows that the value of learned skills extends beyond producing solver-compatible formulations. The transferred knowledge helps determine which modeling components should remain distinct, which components must interact, and how these interactions should be represented across periods. In this case, the important reusable knowledge is the dependency structure connecting workforce, production, inventory, sales, and backlog through explicit state and flow relations.

Taken together, the two cases illustrate complementary roles of learned modeling skills. The first helps determine where an advanced formulation pattern should be applied within the problem structure, while the second helps organize familiar modeling components through explicit state–flow and cross-period dependencies. Both cases support the central premise of OptWeaver that reusable modeling knowledge should capture not only which modeling techniques or components are relevant, but also where they should be instantiated and how they should interact during model construction.

F PROMPTS

We provide the core prompts used by OptWeaver for forward model construction, backward failure attribution and repair, and training-time knowledge evolution. In the implementation, each LLM call is instantiated from a corresponding task-specific skill prompt. For readability, we report only the task-defining instructions and runtime information exposed to the model, while omitting auxiliary implementation details such as repeated formatting constraints, JSON parsing safeguards, retry messages, and execution wrappers. The abbreviated prompts preserve the information available to the model and the output semantics used by the method.

F.1 MODELING ELEMENT EXTRACTION AND PROBLEM-TYPE RECOGNITION

OptWeaver first separates problem understanding into two LLM calls. The modeling-element extractor identifies the variables, constraints, and objective implied by the problem description, while the problem-type recognizer assigns the instance to a reusable modeling family. The extracted elements provide the problem-specific modeling objects used by subsequent construction, whereas the recognized type is used to retrieve a related persistent Skill.

Modeling Element Extraction

You are an expert **in** mathematical optimization modeling.

Problem:
{problem_description}

Extract the modeling elements required to formulate the problem.

Identify:
1. Decision variables
  – semantic role
  – index **or** entity scope
  – variable domain when specified **or** implied

2. Constraints
  – the modeling relation being enforced
  – the variables **or** quantities involved
  – important coupling, logical, temporal, **or** resource dependencies

3. Objective
  – optimization direction
  – quantities **and** terms participating **in** the objective

Focus on mathematical modeling semantics rather than implementation details.
Do **not** generate solver code.

Return a structured representation containing:
{
 "variables": [...],
 "constraints": [...],
 "objectives": [...]
}

Problem-Type Recognition

You are an expert **in** operations research **and** mathematical programming.

Problem:
{problem_description}

Classify the problem into the most appropriate modeling family **from** the available problem–**type set**.

Base the decision on the mathematical structure of the problem, including:
– variable domains
– constraint structure
– temporal **or** network dependencies
– routing **or** scheduling structure
– **and** the form of the objective

Return:
{
"problem_type": "<modeling family>",
"confidence": <0–1>,
"reason": "<brief justification>"
}

The two calls serve different purposes. Element extraction preserves instance-specific modeling semantics, while problem-type recognition provides the retrieval key for reusable knowledge. In particular, problem-type classification is not treated as the final formulation. It only determines which existing Skill should be considered during subsequent Skill construction.

F.2 Task-Specific Guidance Construction

After problem-type recognition, OptWeaver retrieves the most relevant persistent Skill for the recognized modeling family. The retrieved Skill is not applied directly as a complete solution. Instead, the model synthesizes a problem-specific Temporary Skill by jointly considering the current problem, the extracted modeling elements, the retrieved Skill, and the available modeling unit skill.

Temporary Skill Construction

You are an expert optimization modeler.

Current problem:
{problem_description}

Problem **type**:
{problem_type}

Extracted modeling elements:
{extracted_units}

Related reusable Skill:
{related_skill}

Available Modeling Unit Skill:
{unit_skill}

Construct a Temporary Skill specialized to the current problem.

The Temporary Skill should determine how the current modeling elements should be organized **and** should contain four modules:

1. Unit Combination Logic
   – identify the required variable, constraint, **and** objective units

– determine primary, auxiliary, **and** linking components
– specify a valid modeling order **and** their dependencies

2. Python Code Assembly Rules
– specify data **and** index binding
– variable/constraint/objective assembly order
– **and** solver–oriented implementation requirements

3. Advanced Modeling Techniques
– introduce techniques such **as** Big–M, activation logic, propagation constraints, **or** subtour elimination only when required by the current problem structure

4. Full–Process Error Diagnosis Chain
– identify likely formulation failures
– indicate how errors should be traced through modeling decisions
– **and** provide targeted correction principles

Adapt reusable knowledge to the current problem rather than copying the retrieved Skill mechanically. Do **not** invent modeling structure unsupported by the problem.

Return the complete Temporary Skill.

The Temporary Skill acts as a task-specific modeling policy. In particular, it determines how reusable knowledge should be bound to the extracted problem elements before the formulation is converted into an explicit construction structure.

F.3 Modeling Subgraph Construction

OptWeaver next converts the extracted elements and Temporary Skill into a task-specific Modeling Subgraph. The purpose of this prompt is not to generate final code, but to make the formulation structure explicit by assigning modeling units, ordering construction steps, and recording mathematical dependencies.

Modeling Subgraph Construction

You are an expert optimization modeler **and** formulation planner.

Problem:
{problem_description}

Extracted modeling elements:
{extracted_units}

Temporary Skill:
{skill_text}

Available Modeling Unit Skill:
{unit_skill}

Construct one ordered Modeling Subgraph **for** the current problem.

For each extracted variable, constraint, **and** objective:
1. Map it to the most appropriate reusable modeling unit when possible.
2. Introduce a new placeholder unit only when no existing unit represents the required modeling semantics.
3. Add auxiliary modeling steps only when they are mathematically necessary.
4. Order **all** steps according to their construction dependencies.
5. Preserve the parameters **and** bindings required **for** downstream code generation.

The graph should normally follow a dependency–aware order such **as**:
primary variables

```
–> auxiliary variables
–> core constraints
–> linking/support constraints
–> objective.

Return:
{
 "unit_chain": "<ordered unit sequence>",
 "nodes": [
  {
   "step": <integer>,
   "unit_id": "<unit identifier>",
   "unit_type": "<variable|constraint|objective|auxiliary>",
   "description": "<modeling role>",
   "depends_on": [...],
   "params": {...}
  }
 ],
 "notes": [...]
}
```

The resulting subgraph serves as an intermediate representation of the task-specific formulation. Its dependencies make explicit how individual modeling decisions participate in the complete model, while its parameters retain the concrete semantics required by downstream code generation.

F.4 Solver Code Generation

Code generation is intentionally separated from problem interpretation. Once the Modeling Subgraph has been constructed, the code-generation agent receives the subgraph together with implementation snippets associated with the selected modeling unit skill. The original problem description and Temporary Skill are not passed directly to this stage.

Gurobi Code Generation

You are an expert **in** Python **and** Gurobi modeling.

Modeling Subgraph:
{subgraph}

Relevant Modeling Unit Skill:
{unit_skill}

Generate a complete executable Python/Gurobi program that faithfully realizes the provided Modeling Subgraph.

Requirements:
1. Treat the Modeling Subgraph **as** the source of truth **for** the mathematical semantics of the model.
2. Use Modeling Unit Skill only **as** reusable implementation guidance.
3. Preserve the variable domains, indices, coefficients, constraint directions, dependencies, **and** objective semantics specified by the graph.
4. Do **not** introduce additional decision logic **or** modeling assumptions that are **not** supported by the graph.
5. Assemble variables before the constraints **and** objectives that depend on them.
6. Optimize the model **and** expose the resulting objective value **in** the execution output.

Return only the complete runnable Python/Gurobi implementation.

This separation reduces the burden on the code generator to reinterpret the natural-language problem. Instead, it translates an already structured mathematical representation into solver code, while modeling unit skill supplies reusable implementation patterns rather than problem semantics.

F.5 Reflection and Targeted Repair Prompts

When execution does not produce the expected result, OptWeaver invokes a Reflection prompt. Reflection receives the original problem, extracted elements, current Temporary Skill, generated code, solver feedback, known error features from the selected modeling units, and the history of previous modifications. It then attributes the failure to the earliest modeling layer that requires correction and routes the result to a targeted repair action.

Backward Failure Attribution

You are given a failed optimization modeling trajectory.

Problem:
{problem_description}

Extracted modeling elements:
{extracted_units}

Temporary Skill:
{temporary_skill}

Current generated code:
{current_code}

Execution result:
{execution_result}

Current result state:
{result_state}

Known unit–level error features:
{unit_error_features}

Previous modification history:
{modification_history}

Trace the observed failure back through the modeling process.

Determine whether the failure **is** caused by:
– incorrect **or** incomplete extracted modeling elements
– incorrect modeling guidance **or** binding **in** the Temporary Skill
– **or** incorrect solver realization **in** the generated code

Prefer the earliest responsible modeling layer when a downstream error **is** only a consequence of an upstream modeling decision.

Return:
{
"error_class": ["extracted_units_fix" | "skill_fix" | "code_fix"],
"repair_plan": "<targeted correction>",
"history_record": "<concise diagnosis for later attribution>"
}

The returned error class determines which artifact is repaired. A code error is corrected locally without rebuilding the formulation, whereas repairs to the extracted elements or Temporary Skill trigger reconstruction of the Modeling Subgraph and regeneration of the code.

Targeted Repair

Apply only the repair identified by the Reflection result.

Reflection diagnosis:

{reflection_result}

If the target **is** code:
Input:
{current_code}

Correct the identified implementation error **while** preserving the mathematical formulation.

If the target **is** the Temporary Skill:
Input:
{problem_description}
{temporary_skill}

Revise the incorrect modeling guidance **or** problem–specific binding identified by Reflection.

If the target **is** the extracted modeling elements:
Input:
{problem_description}
{extracted_units}

Correct only the missing **or** incorrect variables, constraints, objective semantics, **or** associated bindings.

Do **not** rewrite unaffected components.
Return the repaired artifact **in** the same **format as** its **input**.

This routing implements targeted repair rather than regenerating the entire trajectory after every failure. It also preserves the provenance of each correction in the modification history, which is later used for knowledge evolution.

F.6 Training-Only Validation Prompt

During training, if the self-reflection and repair budget is exhausted, OptWeaver can additionally use the reference formulation to identify modeling errors that cannot be reliably inferred from solver feedback alone. This reference-guided Validation stage is not available at test time.

Reference-Guided Validation

You are given a failed optimization formulation **and** a reference solver implementation.

Problem:
{problem_description}

Extracted modeling elements:
{extracted_units}

Temporary Skill:
{temporary_skill}

Current generated code:
{current_code}

Reference code:
{ground_truth_code}

Known unit–level error features:
{unit_error_features}

Previous modification history:
{modification_history}

Compare the current formulation **with** the reference only to identify material modeling discrepancies.

Determine the earliest modeling decision responsible **for** the discrepancy.
Distinguish among:
– extracted modeling elements
– Temporary Skill / modeling guidance
– **and** solver code realization

Do **not** copy the reference implementation **as** the answer. Use it only **as** evidence **for** locating the missing **or** incorrect modeling decision.

Return:
{
"error_class": ["extracted_units_fix" | "skill_fix" | "code_fix"],
"repair_plan": "<targeted correction>",
"history_record": "<reusable diagnosis>"
}

Validation therefore provides privileged supervision only during training. Its purpose is backward attribution of an unresolved formulation error, rather than direct reference-conditioned code generation.

F.7 Knowledge Evolution Prompts

For successful training trajectories, OptWeaver evaluates how the problem-specific Temporary Skill relates to the retrieved persistent Skill. The evaluation uses the final Temporary Skill together with the modification history, so that changes introduced during successful repair can influence which knowledge should be retained.

Persistent Skill Evaluation

You are given the effective Temporary Skill obtained **for** a successfully modeled problem.

Problem **type**:
{problem_type}

Temporary Skill:
{temporary_skill}

Previously retrieved Skill:
{related_skill_text}

Modification history:
{modification_history}

Assess how the Temporary Skill relates to the existing reusable Skill.

Determine whether the persistent Skill should:
– be kept unchanged
– be modified **with** transferable knowledge **from** this trajectory
– **or** be recreated when the current knowledge represents a substantially different modeling family

Base the decision on reusable modeling structure rather than problem–specific constants **or** variable names.

Return:
{
"related_skill_action": "<keep|modify|recreate>",
"similarity_score": <0–1>,
"kb_reason": "<brief justification>",

```
  "new_problem_type_name": "<new name when recreation is required>"
}
```

When modification is selected, a second prompt integrates only transferable changes into the persistent Skill.

Persistent Skill Evolution

Existing reusable Skill:
{related_skill_text}

Effective Temporary Skill:
{temporary_skill}

Successful modification history:
{modification_history}

Evolution decision:
{decision_result}

Update the reusable Skill using only knowledge that can transfer to future problems of the same modeling family.

Preserve valid existing guidance. Incorporate corrected modeling principles, applicability conditions, dependency relations, **and** reusable failure–avoidance knowledge when they are supported by the successful trajectory.

Do **not** retain instance–specific variable names, numerical coefficients, **or** problem–specific solutions unless they are necessary to illustrate a general modeling principle.

Return the complete updated reusable Skill.

OptWeaver applies analogous evolution prompts to the modeling unit skill, the solver code Skill, and, when a new problem family is created, the problem-type classifier. These updates are expressed as compact patches when possible rather than regenerating the complete knowledge artifact, reducing unnecessary changes to previously validated knowledge.

Train–Test Separation. The forward prompts for element extraction, problem-type recognition, Temporary Skill construction, Modeling Subgraph construction, code generation, self-reflection and repair are shared between training and testing. Reference-guided Validation and knowledge-evolution prompts are enabled only during training. At test time, the learned Skill system are frozen, and no reference solution is exposed to the model.