OPTROUTE: EVOLVING TREES THAT COUPLE SKILL ORGANIZATION AND RETRIEVAL FOR LLM-BASED OPTIMIZATION MODELING

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) have shown strong potential for automating optimization modeling from natural-language problem descriptions. Prompt-based methods are particularly attractive because they can reuse external modeling knowledge at inference time without updating model parameters. However, as reusable modeling knowledge grows, it becomes increasingly challenging to organize related skills while ensuring that the right knowledge can be reliably retrieved for new problems. Existing approaches typically organize skills using flat catalogs or predefined hierarchies, leaving skill organization disconnected from how the stored knowledge is subsequently retrieved. We propose **OptRoute**, a framework that couples skill organization and retrieval through an evolving tree. Within the tree, paradigm nodes organize related formulation families and provide hierarchical routing, while leaf nodes preserve complete, solver-ready skills. As new reference-grounded skills are added, OptRoute jointly updates skill content and tree structure, refining existing skills and evolving paradigms that distinguish formulation families relevant to future retrieval. At inference time, a new problem is routed through the resulting hierarchy to retrieve relevant skills for solver-code generation. Experiments on eight OR benchmarks show that OptRoute achieves 79.2% macro-average Pass@1 accuracy with DeepSeek-V4, outperforming the evaluated baselines. These results demonstrate the effectiveness of coupling skill organization and retrieval for scalable reuse of optimization modeling knowledge.

1 INTRODUCTION

Optimization modeling is a fundamental task in operations research (OR), translating real-world decision problems into mathematical formulations or solver-ready code (Saaty, 2004; Hillier et al., 1995). In practice, this translation requires identifying decision variables, objectives, constraints, and their relationships from natural-language descriptions. It is a challenging process that demands substantial domain knowledge and OR modeling expertise.

Recent advances in large language models (LLMs) have enabled automated OR modeling from natural-language problem descriptions (AhmadiTeshnizi et al., 2024; Huang et al., 2025; Astorga et al., 2024). Existing methods mainly fall into two categories: fine-tuned models that acquire OR modeling knowledge through domain-specific training (Huang et al., 2025; Shu et al., 2025; Zhou et al., 2026; Chen et al., 2026), and prompt-based agentic methods that improve test-time modeling while keeping the backbone LLM fixed (Xiao et al., 2024; AhmadiTeshnizi et al., 2024; Astorga et al., 2024). The latter are particularly attractive because they can incorporate external modeling knowledge at inference time without updating model parameters.

Recent methods therefore augment LLMs with reusable external knowledge. OptiTree organizes problems through hierarchical subproblem relations (Liu et al., 2026); AlphaOPT maintains solver-verified insights in a hierarchical taxonomy (Kong et al., 2026); and OptSkills groups problems by optimization archetypes and distills reusable skills (Yang et al., 2026). These methods demonstrate the value of organizing and reusing modeling knowledge. However, as the knowledge base grows, related skills may overlap in problem semantics while differing in important formulation details.

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.

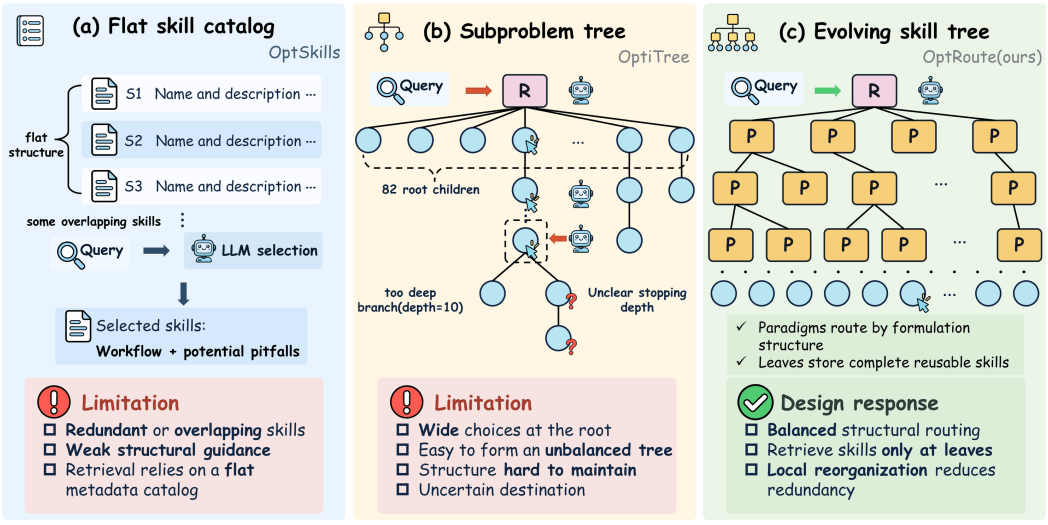

Figure 1: **Comparison of skill-memory organizations for OR modeling.** Existing approaches use flat skill catalogs or hierarchies defined by problem relations. OptRoute instead evolves a shared tree in which paradigm nodes organize related formulation families and provide the structure for retrieving reusable skill leaves.

Effective reuse thus depends not only on what knowledge is stored, but also on how it is organized for retrieval.

Existing memory structures only partially address this challenge. As illustrated in Figure 1, flat skill catalogs provide limited structure for distinguishing related skills, while problem-centric hierarchies organize knowledge according to relations that may not directly support skill retrieval. Moreover, these relations can change as new skills are added, calling for a memory structure that evolves with the accumulated knowledge. **This raises a central question: how can a growing skill memory for OR modeling organize related knowledge while making that organization directly useful for retrieval?**

To address this challenge, we propose OptRoute, an evolving tree framework that couples skill organization and retrieval for LLM-based optimization modeling. OptRoute assigns complementary roles within a shared tree: paradigm nodes organize related formulation families and support hierarchical routing, while leaf nodes store complete, reusable formulation skills. As new reference-grounded skills are added, OptRoute updates both skill content and tree structure, refining existing skills and reorganizing paradigms to capture formulation distinctions useful for retrieval. The resulting tree therefore organizes accumulated modeling knowledge while simultaneously providing the structure through which that knowledge is retrieved.

Concretely, OptRoute extracts and validates reusable skills from reference-grounded cases, incrementally integrates them into the evolving tree, and retrieves relevant skills through the resulting hierarchy at inference time. The retrieved skills are synthesized into a modeling plan for solver-code generation, while the backbone LLM remains frozen. Our contributions are summarized as follows:

- We propose OptRoute, an evolving tree framework that couples skill organization and retrieval, with paradigm nodes organizing formulation families and leaf nodes storing complete, reusable modeling skills.
- We develop a skill tree evolution mechanism that integrates and reorganizes skills as knowledge grows, allowing the hierarchy to emerge and evolve without a predefined taxonomy.
- Experiments on eight OR benchmarks show that OptRoute achieves strong performance across diverse optimization modeling tasks. Controlled retrieval ablations and scaling studies further demonstrate the benefits of the evolving hierarchy for skill retrieval and memory growth.

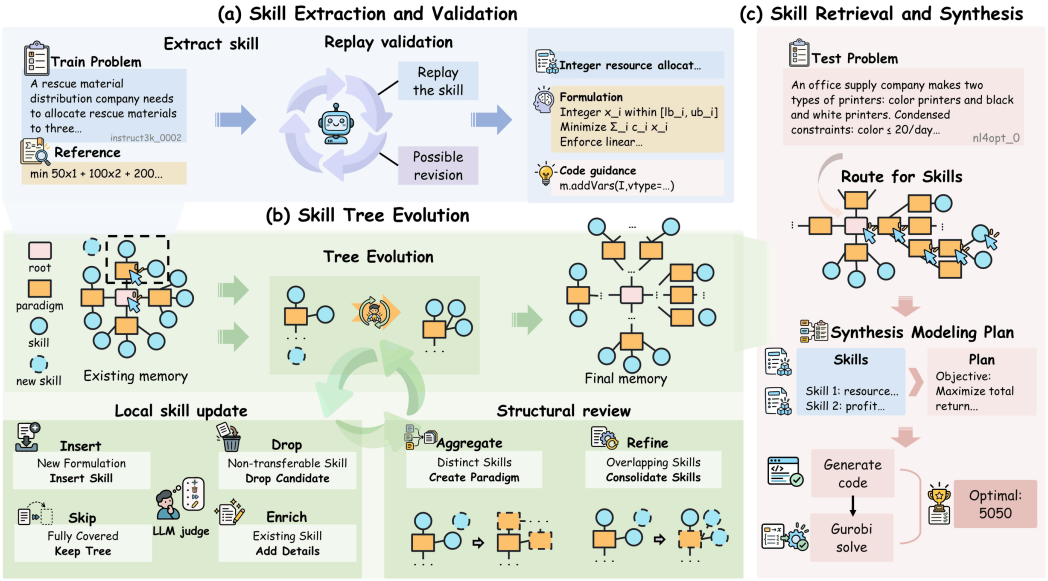

Figure 2: **Overview of the OptRoute framework.** OptRoute extracts and validates reusable formulation skills, organizes them in an evolving paradigm–skill tree, and uses the same hierarchy to retrieve relevant skills for solver-code generation.

2 METHOD

Given a natural-language problem p and an external skill memory $\mathcal{M}$, OptRoute retrieves relevant skills and uses them to condition a frozen LLM for solver-code generation:

$$\mathcal{S}_p = \mathcal{R}(p; \mathcal{M}), \qquad z \sim \pi_\theta(\cdot \mid p, \mathcal{S}_p), \tag{1}$$

where $\mathcal{R}$ denotes the retrieval process, $\mathcal{S}_p$ is the retrieved skill set, and z is the generated solver code. The backbone parameters θ remain fixed. Our goal is to evolve $\mathcal{M}$ such that its organization both captures relations among accumulated skills and provides the structure for retrieving them. We represent the memory as a rooted typed tree

$$\mathcal{M} = (V, E, r, \tau, \phi), \qquad V = P \sqcup S, \tag{2}$$

where P and S denote paradigm and skill nodes, respectively, $r \in P$ is the root, τ specifies the node type, and $\phi(v)$ stores the semantic content of node v. Paradigm nodes organize related formulation families and provide the hierarchy for routing, while skill nodes are leaves that store complete, reusable formulation knowledge. The children of a paradigm node may be paradigm or skill nodes. A paradigm with no paradigm children defines a terminal region, whose skill children form the candidate set for retrieval. As shown in Figure 2, OptRoute consists of three stages: (i) **Skill Extraction and Validation** extracts reusable formulation skills from reference cases and validates them through replay; (ii) **Skill Tree Evolution** integrates validated skills and evolves their organization; and (iii) **Skill Retrieval and Synthesis** routes a new problem through the evolved tree, retrieves relevant leaf skills, and synthesizes them into guidance for solver-code generation.

2.1 SKILL EXTRACTION AND VALIDATION

We construct the skill memory from solved optimization cases $\mathcal{D} = \{(p_i, r_i, y_i)\}_{i=1}^{N}$, where p_i is a natural-language problem, r_i is its reference formulation, and y_i is the reference objective value. For each case, we extract at most one reusable skill and retain it only after replay validation.

Skill extraction. Given (p_i, r_i), an LLM extracts a candidate skill s_i containing a formulation title, applicability description, formulation skeleton, solver-code guidance, and retrieval cues. The extraction focuses on transferable modeling structure while removing source-specific entities and numerical values. The candidate is then checked against the reference formulation for consistency.

Replay validation. The candidate skill is replayed on its source problem, producing solver status σ_i and objective value $\hat{y}_i$. We accept s_i when

$$\mathrm{Valid}(s_i) = \mathbb{I}[\sigma_i = \mathrm{optimal} \ \wedge \ |\hat{y}_i - y_i| \leq \epsilon]\,. \tag{3}$$

If validation fails, the skill is revised once using the reference formulation and execution feedback and then replayed. Candidates that still fail are discarded.

2.2 Skill Tree Evolution

After validation, each skill is integrated into the current memory $\mathcal{M}_t$. Rather than assigning skills to a predefined OR taxonomy, OptRoute evolves the tree according to relations among accumulated skills. Each update consists of insertion routing, a local skill update, and structural review.

Insertion routing. Given source problem p_t, candidate skill s_t, and current memory $\mathcal{M}_t$, insertion routing identifies a local paradigm region

$$q_t = \mathcal{R}_{\mathrm{ins}}(p_t, s_t; \mathcal{M}_t). \tag{4}$$

Starting from the root, the LLM compares the candidate with the semantic scope of the current paradigm children and follows the most compatible branch. Routing continues until a region suitable for skill-level integration is reached. If the candidate matches no existing child paradigm, it is temporarily assigned to an *Other* bucket rather than forced into an unsuitable branch. Such skills can later be reorganized during structural review. The resulting region q_t defines the local context for integrating the candidate.

Local skill update. Within the routed region, the incoming skill is compared with existing skills and incorporated through

$$\widetilde{\mathcal{M}}_t = \mathcal{U}(\mathcal{M}_t, q_t, s_t). \tag{5}$$

The update operator $\mathcal{U}$ considers four actions. INSERT adds s_t as a new leaf when it introduces formulation knowledge not covered by the current region. SKIP leaves the memory unchanged when the candidate is already sufficiently represented by an existing skill. DROP discards the candidate when it is insufficiently transferable or useful despite passing source-level validation. ENRICH incorporates complementary information from s_t into an existing skill when they capture the same underlying formulation pattern but the candidate provides additional useful detail. These actions update the stored skill content without changing the routing structure.

Structural review. After the local update, OptRoute reviews region q_t to determine whether its organization remains suitable for distinguishing and retrieving the accumulated skills:

$$\mathcal{M}_{t+1} = \mathcal{O}(\widetilde{\mathcal{M}}_t, q_t). \tag{6}$$

The structural operator $\mathcal{O}$ considers three outcomes. KEEP preserves the current organization when the region already forms a coherent retrieval group. REFINE consolidates redundant or weakly differentiated skills without introducing a new routing level. AGGREGATE introduces new paradigm nodes and redistributes skills when the region contains distinct formulation families that should be separated during retrieval. Together, insertion routing, local skill update, and structural review define one memory-evolution step:

$$\mathcal{M}_{t+1} = \mathcal{O}(\mathcal{U}(\mathcal{M}_t, q_t, s_t), q_t)\,, \qquad q_t = \mathcal{R}_{\mathrm{ins}}(p_t, s_t; \mathcal{M}_t). \tag{7}$$

After all candidates are processed, a bottom-up pass updates paradigm descriptions and merges redundant sibling paradigms while preserving skill content. Local skill updates determine what knowledge is retained, while structural review determines how that knowledge is organized and subsequently retrieved. As skills accumulate, the tree therefore evolves as both a knowledge organization and a retrieval structure.

2.3 Skill Retrieval and Synthesis

Given a new problem x, OptRoute routes the query through the evolved paradigm hierarchy. Starting from the root, the LLM evaluates the child paradigms of each visited node and follows the relevant branches until terminal regions are reached. Let

$$T_x = \mathrm{Traverse}(x; \mathcal{M}) \tag{8}$$

denote the resulting terminal paradigms. Their skill children form the candidate pool

$$C_x = \bigcup_{q \in T_x} \big(\mathrm{Ch}(q) \cap S\big), \tag{9}$$

from which the LLM selects a small set of relevant skills. The selected skills are then combined with the original problem and synthesized into a modeling plan. When multiple skills are retrieved, the synthesis step integrates their complementary formulation knowledge. The resulting plan guides the frozen LLM in generating solver code.

3 EXPERIMENTS

Our experiments address five questions: (1) Does OptRoute improve optimization-modeling performance over general-purpose LLMs and existing agentic methods? (2) Does evolving skill organization improve retrieval and modeling performance? (3) How does the skill memory evolve as more modeling experience is incorporated? (4) How does the evolved structure affect retrieval on individual problems? (5) What inference cost does OptRoute incur compared with multi-stage agentic methods?

3.1 EXPERIMENTAL SETUP

Benchmarks. We evaluate OptRoute on eight public benchmarks for optimization modeling, covering standard LP/MILP problems, structurally complex formulations, and industrial applications: `NL4Opt` (Ramamonjison et al., 2023), `MAMO-Easy` (Huang et al., 2024), `NLP4LP` (AhmadiTeshnizi et al., 2024), `OptiBench` (Yang et al., 2025), `MAMO-Complex` (Huang et al., 2024), `ComplexOR` (Xiao et al., 2024), `IndustryOR` (Huang et al., 2025), and `OptMATH` (Lu et al., 2025). We report Pass@1 accuracy as the primary evaluation metric and use the macro average across benchmarks for overall comparison. Dataset statistics and evaluation details are provided in Appendix A.1.

Baselines. We compare OptRoute with representative baselines from three categories:

- **General-Purpose LLMs:** General-purpose LLMs without optimization-specific fine-tuning, including `DeepSeek-V4-flash` (Xu et al., 2026) and `DouBao-Seed-2.0-lite` (Seed, 2026), evaluated under the same base prompting protocol.
- **Fine-Tuned Models:** Specialized models trained for optimization modeling, including `ORLM` (Huang et al., 2025), LLMOPT (Shu et al., 2025), OptMATH (Lu et al., 2025), SIRL (Chen et al., 2026), and StepORLM (Zhou et al., 2026).
- **Agentic Methods:** Prompt-based OR modeling methods that improve inference through structured workflows or reusable external knowledge. CoE (Xiao et al., 2024) coordinates expert roles across modeling stages; OptiMUS (AhmadiTeshnizi et al., 2024) uses a formulation–coding–debugging workflow; OptiTree (Liu et al., 2026) organizes knowledge through hierarchical problem relations; AlphaOPT (Kong et al., 2026) maintains solver-verified modeling insights in a hierarchical memory; and OptSkills (Yang et al., 2026) distills reusable skills from optimization archetypes.

Implementation Details. We construct the skill memory from 800 examples sampled from the ORLM Instruct3K training set (Huang et al., 2025). We use `DeepSeek-V4` (Xu et al., 2026) as the primary backbone for skill extraction, memory construction, and evaluation, and additionally evaluate `DouBao-Seed-2.0` (Seed, 2026) to test backbone generality. Agentic baselines are re-evaluated under matched backbone and evaluation settings whenever possible. Results for specialized models that cannot be reproduced are taken from the original publications and marked with (*) in Table 1. Further implementation details are provided in Appendix A.3.

3.2 MAIN RESULTS

We first evaluate whether OptRoute improves optimization-modeling performance over direct prompting and existing agentic methods. Table 1 summarizes the results across eight OR benchmarks. Compared with direct prompting, OptRoute improves macro-average Pass@1 from 71.3%

Table 1: **Pass@1 accuracy on eight optimization-modeling benchmarks.** All agentic methods are evaluated under unified backbone-specific protocols. Results from original publications are marked with (*), and the best and second-best results are shown in **bold** and underlined. Avg. denotes the macro average.

Method	Easy Tasks				Complex Tasks				Avg.
	NL4OPT	MAMO Easy	NLP4LP	OptiB.	MAMO Cpx.	CpxOR	IndOR	OptM.	
General-Purpose LLMs									
DeepSeek-V4	76.5	96.7	89.9	80.9	55.9	55.6	71.4	43.4	71.3
DouBao-Seed-2.0	81.7	97.6	93.8	83.9	55.9	50.0	73.8	42.2	72.4
Fine-Tuned Models									
ORLM*	73.8	90.4	76.4	61.8	59.5	50.0	42.9	2.6	57.2
LLMOPT*	80.3	89.5	73.4	53.8	44.1	35.3	29.0	12.5	52.2
OptMATH*	94.7	86.5	–	57.9	51.2	–	20.0	24.4	–
SIRL*	94.8	98.0	96.6	89.6	81.1	44.4	50.0	27.1	72.7
StepORLM*	96.7	95.4	97.2	88.6	79.3	50.0	52.4	13.9	71.7
Agentic Frameworks (DouBao-Seed-2.0)									
CoE	82.2	96.1	87.6	85.1	61.3	55.6	69.0	45.2	72.8
OptiMUS	83.6	89.2	92.7	78.9	51.4	38.9	61.9	45.2	67.7
OptiTree	83.6	**97.8**	**96.1**	86.1	80.2	**61.1**	71.4	47.6	78.0
AlphaOPT	83.6	96.3	95.5	88.6	54.1	55.6	69.0	40.4	72.9
OptSkills	76.5	97.2	88.2	80.1	55.0	55.6	**73.8**	**51.8**	72.3
OptRoute(ours)	**84.5**	95.6	93.8	**88.8**	**82.9**	**61.1**	**73.8**	50.6	**78.9**
Agentic Frameworks (DeepSeek-V4)									
CoE	72.8	94.1	86.0	79.9	59.5	50.0	66.7	47.6	69.6
OptiMUS	74.2	89.9	93.8	85.9	55.0	50.0	61.9	37.3	68.5
OptiTree	78.9	**97.2**	91.6	84.6	82.9	50.0	64.3	45.8	74.4
AlphaOPT	77.0	96.0	91.6	80.9	54.1	**55.6**	**76.2**	46.4	72.2
OptSkills	77.9	94.7	93.8	87.8	81.1	**55.6**	69.0	49.4	76.2
OptRoute(ours)	**85.5**	94.9	**96.6**	**90.3**	**86.5**	**55.6**	71.4	**52.4**	**79.2**

Abbreviations: OptiB.: OptiBench; OptM.: OptMATH; CpxOR: ComplexOR; IndOR: IndustryOR; MAMO Cpx.: MAMO Complex.

to 79.2% with DeepSeek-V4 and from 72.4% to 78.9% with DouBao-Seed-2.0. The improvements are particularly clear on several complex benchmarks, showing the benefit of reusable external skills for test-time modeling without parameter updates. Under matched backbones and evaluation protocols, OptRoute also achieves the highest macro-average accuracy among the evaluated agentic methods. With DeepSeek-V4, it reaches 79.2%, 3.0 percentage points above the strongest agentic baseline, with notable gains on MAMO-Complex and OptMATH. These results show that OptRoute effectively reuses external modeling knowledge across diverse optimization tasks.

3.3 Analysis of Skill Organization and Retrieval

We next examine the two components underlying OptRoute: how accumulated skills are organized as the memory evolves, and how the resulting structure affects retrieval.

Effect of Evolving Skill Organization. We isolate the effect of evolving skill organization using the same 609 validated skills produced by skill extraction and replay validation. All variants share the same synthesis and code-generation pipeline and differ only in how the validated skills are organized and consolidated. Flat LLM maintains a flat skill catalog, while OptiTree-like introduces a hierarchy without the skill and structural updates of OptRoute. In contrast, OptRoute jointly updates skill content and paradigm structure as new skills are incorporated. As shown in Table 2, the differences are concentrated on the complex benchmarks. OptiTree-like remains competitive on the easy tasks but drops substantially on MAMO-Complex, IndustryOR, and OptMATH; the flat

Table 2: **Ablation of evolving skill organization.** Results on eight optimization-modeling benchmarks with DeepSeek-V4. Easy-4 (Avg.) and Overall Avg. denote macro averages over four and eight benchmarks, respectively. Δ denotes the performance difference from OptRoute.

Method	Ablation Factors		Easy-4 (Avg.)	Complex Tasks				Overall Avg.
	Hier.	Evol.		MAMO-Cpx.	CpxOR	IndOR	OptM.	
OptRoute (Ours)	✓	✓	91.8	86.5	55.6	71.4	52.4	79.2
OptiTree-like	✓	×	93.0 $\uparrow$1.2	62.2 $\downarrow$24.3	55.6	61.9 $\downarrow$9.5	46.4 $\downarrow$6.0	74.8 $\downarrow$4.4
Flat LLM	×	×	91.8	72.1 $\downarrow$14.4	55.6	59.5 $\downarrow$11.9	44.0 $\downarrow$8.4	74.8 $\downarrow$4.4

Abbreviations: Hier.: hierarchy; Evol.: skill tree evolution; OptiTree-like: OptiTree-like Hierarchy.

Table 3: **Controlled comparison of retrieval strategies with fixed skill memory.** Results on eight optimization-modeling benchmarks with DeepSeek-V4 using the same 247 leaf skills. Easy-4 (Avg.) and Overall Avg. denote macro averages over four and eight benchmarks, respectively. Δ denotes the performance difference from OptRoute.

Method	Easy-4 (Avg.)	Complex Tasks				Overall Avg.
		MAMO-Cpx.	CpxOR	IndOR	OptM.	
OptRoute (Ours)	91.8	86.5	55.6	71.4	52.4	79.2
BM25	91.2 $\downarrow$0.6	75.7 $\downarrow$10.8	55.6	66.7 $\downarrow$4.7	45.8 $\downarrow$6.6	76.0 $\downarrow$3.2
Dense	91.6 $\downarrow$0.2	83.8 $\downarrow$2.7	50.0 $\downarrow$5.6	69.0 $\downarrow$2.4	50.6 $\downarrow$1.8	77.5 $\downarrow$1.7
Flat LLM	92.9 $\uparrow$1.1	81.1 $\downarrow$5.4	50.0 $\downarrow$5.6	61.9 $\downarrow$9.5	48.2 $\downarrow$4.2	76.6 $\downarrow$2.6
Static Clustering	87.9 $\downarrow$3.9	62.2 $\downarrow$24.3	55.6	61.9 $\downarrow$9.5	45.8 $\downarrow$6.6	72.1 $\downarrow$7.1

variant shows similar degradation. Overall, both alternatives reach 74.8% macro-average accuracy, compared with 79.2% for OptRoute. These results indicate that a hierarchy alone does not account for the gains: reorganizing accumulated skills as the memory evolves is important for downstream modeling.

Effect of the Evolved Retrieval Structure. We next isolate the retrieval benefit of the evolved tree. After memory construction, OptRoute contains 247 active leaf skills. We fix this skill set across all variants and vary only how the skills are retrieved. BM25 and Dense retrieve skills by lexical and embedding similarity, respectively (Robertson & Zaragoza, 2009; Su et al., 2025; Karpukhin et al., 2020; Das et al., 2025); Flat LLM selects directly from the full catalog; and Static Clustering partitions skills by embedding similarity before cluster-level selection (Liu & Croft, 2004). OptRoute instead retrieves through the evolved paradigm hierarchy before selecting leaf skills. As shown in Table 3, OptRoute achieves 79.2% macro-average accuracy, compared with 76.0% for BM25, 77.5% for Dense, 76.6% for Flat LLM, and 72.1% for Static Clustering. The differences are again more pronounced on several complex benchmarks. Because all variants use the same 247 leaf skills and the same downstream generation pipeline, this comparison isolates the effect of retrieval structure. The results show that organizing skills into an evolved paradigm hierarchy provides a more effective basis for retrieval than flat similarity matching or static partitions.

3.4 Memory Evolution and Structural Growth

We next examine how the memory evolves as more modeling experience is incorporated. We build memories from 100, 200, 400, and 800 construction cases under the same retrieval and generation protocol, and measure both downstream performance and the number of active leaf skills. As shown in Figure 3(a), macro-average Pass@1 increases from 76.3% to 79.2% as the number of construction cases grows, while the ratio of active leaf skills to construction cases decreases from 45.0% to 30.9%. Thus, additional modeling experience improves downstream performance without proportional growth in active leaf skills, indicating that the evolution process consolidates overlapping

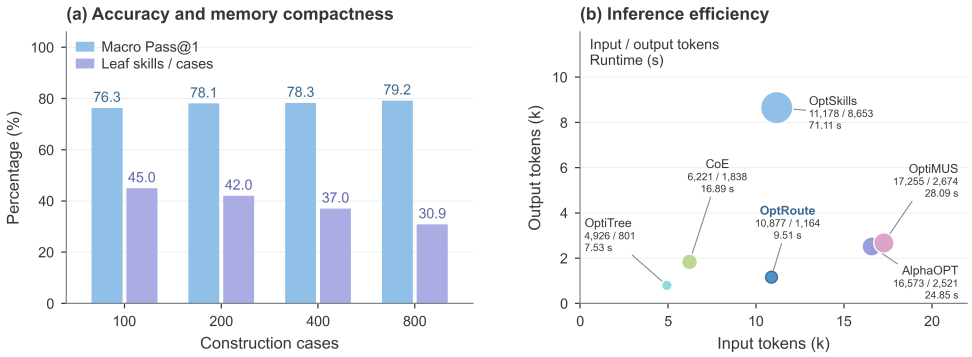

Figure 3: **Memory scaling and inference efficiency of OptRoute.** **(a)** Pass@1 accuracy and active leaf-skill ratio under different memory scales. **(b)** Token usage and cumulative LLM invocation time of representative agentic methods; bubble size indicates total LLM time.

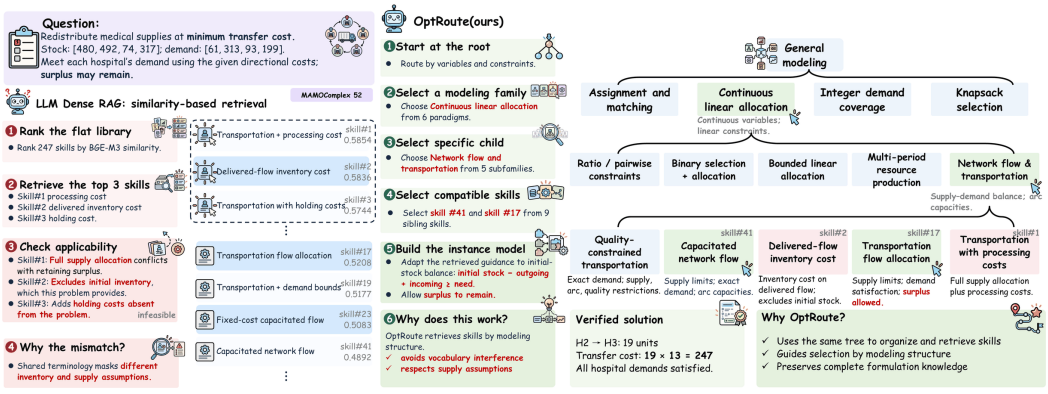

Figure 4: **Representative retrieval case of OptRoute.** The evolved paradigm hierarchy guides the query to a compatible formulation region, avoiding mismatched skills from semantically similar alternatives.

knowledge rather than simply accumulating one skill per case. Detailed checkpoint statistics are reported in Appendix C.

3.5 Case Study on Evolving Skill Tree Retrieval

Figure 4 illustrates how the evolved tree distinguishes semantically similar skills with different formulation assumptions. We consider MAMO-Complex instance 52, which requires medical-supply redistribution under minimum transfer cost while allowing surplus stock to remain. Dense retrieval ranks several semantically similar transportation skills highly, but their formulation assumptions do not match the instance, leading to an infeasible formulation. In contrast, OptRoute routes the problem through continuous linear allocation to the network-flow and transportation family, where it retrieves skills consistent with supply limits, demand satisfaction, and residual surplus. The resulting formulation yields the verified objective value of 247. This example illustrates how coupling skill organization with retrieval can reduce ambiguity among semantically similar skills.

3.6 Inference Cost and Efficiency

Finally, we examine the inference overhead of OptRoute across the eight benchmarks. Figure 3(b) compares input and output tokens together with cumulative LLM invocation time. OptRoute requires 9.5 seconds of LLM time per problem, remaining close to lightweight agentic methods and below several multi-stage baselines. This indicates that hierarchical retrieval and skill synthesis add moderate inference overhead.

4 Related Work

4.1 LLMs for Operations Research Modeling

Existing LLM-based OR modeling methods can be broadly divided into fine-tuned and prompt-based approaches. Among fine-tuned methods, ORLM, OptMATH, and LLaMoCo primarily construct domain-specific training data and improve formulation or code generation through supervised learning (Huang et al., 2025; Lu et al., 2025; Ma et al., 2026). SIRL and OR-R1 emphasize reinforcement learning with solver-derived or execution-based feedback (Chen et al., 2026; Ding et al., 2026), while LLMOPT and StepORLM combine supervised learning with preference optimization using solution-quality judgments or process supervision (Shu et al., 2025; Zhou et al., 2026).

Among prompt-based methods, OptiMUS, Chain-of-Experts, and OR-LLM-Agent coordinate formulation, programming, and debugging through structured agentic workflows (AhmadiTeshnizi et al., 2024; Xiao et al., 2024; Zhang & Luo, 2025). SolverLLM (Li et al., 2026) and the formulation-search framework of Astorga et al. (2024) explore and evaluate alternative modeling decisions at inference time. AlphaOPT, OptiTree, and OptSkills instead emphasize the reuse of external modeling knowledge through structured experiences, hierarchical modeling thoughts, or distilled skills (Kong et al., 2026; Liu et al., 2026; Yang et al., 2026). ReLoop focuses on detecting and repairing formulation errors through execution and behavioral verification (Lian et al., 2026). OptRoute also builds on reusable external knowledge, but organizes formulation skills in an evolving tree whose structure is directly used for retrieval.

4.2 Hierarchical Knowledge Organization and Retrieval

Retrieval-augmented generation (RAG) augments LLMs with external knowledge retrieved at inference time and has become a standard paradigm for knowledge-intensive generation (Lewis et al., 2020; Fan et al., 2024). A common approach is to maintain a flat collection and rank candidates using sparse or dense similarity, as in BM25 and dense passage retrieval (Robertson & Zaragoza, 2009; Karpukhin et al., 2020). Cluster-based retrieval instead groups related items before retrieval to reduce the candidate space (Liu & Croft, 2004). Recent work has further studied retrieval for reasoning-intensive queries. BRIGHT shows the limitations of surface-level relevance on such queries (Su et al., 2025), while RaDeR incorporates reasoning signals into dense retriever training (Das et al., 2025).

Another line of work structures knowledge hierarchically to support retrieval. RAPTOR recursively clusters and summarizes text into a multi-level retrieval tree (Sarthi et al., 2024), while dynamic tree memory supports incremental insertion and hierarchy updates as new memories are added (Rezazadeh et al., 2025). Tree-based indexing has also been explored in recommender systems and document retrieval, where hierarchical structures progressively narrow the search space (Zhu et al., 2018; Wang et al., 2022). More recently, LLM-guided hierarchical search uses LLMs to navigate tree-structured indexes for reasoning-intensive retrieval (Gupta et al., 2025). These studies show that retrieval depends not only on how candidates are matched, but also on how knowledge is organized and navigated. OptRoute extends this perspective to optimization modeling by evolving a shared tree in which paradigm nodes organize related formulation families and simultaneously define the routing structure for retrieving reusable skill leaves.

5 Conclusion

We presented OptRoute, an evolving skill tree framework that couples skill organization and retrieval for LLM-based optimization modeling. Paradigm nodes organize related formulation families and provide the structure for retrieval, while leaf nodes preserve complete, solver-ready skills. As new skills are added, the tree evolves to consolidate reusable knowledge and refine its organization for subsequent retrieval. Experiments on eight OR benchmarks show that OptRoute outperforms the evaluated agentic baselines in macro-average Pass@1, while controlled comparisons with fixed leaf skills demonstrate the benefit of the evolved hierarchy for retrieval. Overall, our results show that evolving a shared structure for both skill organization and retrieval provides an effective way to reuse growing optimization modeling knowledge.

AI USE STATEMENT

LLMs are used as components of OptRoute and the evaluated baselines for the experiments in this work; their roles are described in the Method and Experiments sections. We also used generative AI tools to assist with literature search and discovery, manuscript drafting, and language editing. All AI-assisted content, citations, and experimental claims were reviewed by the authors, who take full responsibility for the final manuscript.

ETHICS STATEMENT

This work uses publicly available operations research benchmarks and training data and does not involve private or personally identifiable information.

REPRODUCIBILITY STATEMENT

Section 2 describes the construction and retrieval procedures. Appendix A reports benchmark composition, baseline configurations, construction data, and model settings. Appendix B specifies operation boundaries, structural constraints, routing, and objective-based scoring. Appendix C reports structural statistics and performance at four memory scales. Appendix D provides the principal prompts. The complete project code and associated data are included in the supplementary materials.

REFERENCES

Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. Optimus: Scalable optimization modeling with (mi) lp solvers and large language models. *arXiv preprint arXiv:2402.10172*, 2024.

Nicolás Astorga, Tennison Liu, Yuanzhang Xiao, and Mihaela Van Der Schaar. Autoformulation of mathematical optimization models using llms. *arXiv preprint arXiv:2411.01679*, 2024.

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, 2026.

Debrup Das, Sam O'Nuallain, and Razieh Rahimi. Rader: Reasoning-aware dense retrieval models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 19981–20008, 2025.

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.

Wenqi Fan, Yujuan Ding, Liangbo Ning, Shijie Wang, Hengyun Li, Dawei Yin, Tat-Seng Chua, and Qing Li. A survey on rag meeting llms: Towards retrieval-augmented large language models. In *Proceedings of the 30th ACM SIGKDD conference on knowledge discovery and data mining*, pp. 6491–6501, 2024.

Nilesh Gupta, Wei-Cheng Chang, Ngot Bui, Cho-Jui Hsieh, and Inderjit S Dhillon. Llm-guided hierarchical search for end-to-end reasoning intensive retrieval. *arXiv preprint arXiv:2510.13217*, 2025.

Frederick S Hillier, Gerald J Lieberman, Karl Schmedders, Molly Stephens, Mark Hillier, and Michael O'Sullivan. *Introduction to operations research*, volume 9. McGraw-Hill New York, 1995.

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, 2025.

Xuhan Huang, Qingning Shen, Yan Hu, Anningzhe Gao, and Benyou Wang. Mamo: a mathematical modeling benchmark with solvers. *arXiv preprint arXiv:2405.13144*, pp. 189–203, 2024.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 conference on empirical methods in natural language processing (EMNLP)*, pp. 6769–6781, 2020.

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.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33: 9459–9474, 2020.

Dong Li, Xujiang Zhao, Linlin Yu, Yanchi Liu, Wei Cheng, Zhengzhang Chen, Zhong Chen, Feng Chen, Chen Zhao, and Haifeng Chen. Solverllm: Leveraging test-time scaling for optimization problem via llm-guided search. *Advances in Neural Information Processing Systems*, 38:100028–100058, 2026.

Junbo Jacob Lian, Yujun Sun, Huiling Chen, Chaoyu Zhang, Hanzhang Qin, and Chung-Piaw Teo. Reloop: Structured modeling and behavioral verification for reliable llm-based optimization. *arXiv preprint arXiv:2602.15983*, 2026.

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, 2026.

Xiaoyong Liu and W Bruce Croft. Cluster-based retrieval using language models. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 186–193, 2004.

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

Zeyuan Ma, Yue-Jiao Gong, Hongshu Guo, Jiacheng Chen, Yining Ma, Zhiguang Cao, and Jun Zhang. Llamoco: Instruction tuning of large language models for optimization code generation. *IEEE Transactions on Evolutionary Computation*, 2026.

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.

Alireza Rezazadeh, Zichao Li, Wei Wei, and Yujia Bao. From isolated conversations to hierarchical schemas: Dynamic tree memory representation for llms. In *International Conference on Learning Representations*, volume 2025, pp. 990–1023, 2025.

Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Foundations and trends® in information retrieval*, 4(1-2):1–174, 2009.

Thomas L Saaty. *Mathematical methods of operations research*. Courier Corporation, 2004.

Parth Sarthi, Salman Abdullah, Aditi Tuli, Shubh Khanna, Anna Goldie, and Christopher Manning. Raptor: Recursive abstractive processing for tree-organized retrieval. In *International Conference on Learning Representations*, volume 2024, pp. 32628–32649, 2024.

Bytedance Seed. Seed2. 0 model card: Towards intelligence frontier for real-world complexity. *arXiv preprint arXiv:2607.00248*, 2026.

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.

Hongjin Su, Howard Yen, Mengzhou Xia, Weijia Shi, Niklas Muennighoff, Han-yu Wang, Liu Haisu, Quan Shi, Zachary Siegel, Michael Tang, et al. Bright: A realistic and challenging benchmark for reasoning-intensive retrieval. In *International Conference on Learning Representations*, volume 2025, pp. 48941–48991, 2025.

Yujing Wang, Yingyan Hou, Haonan Wang, Ziming Miao, Shibin Wu, Qi Chen, Yuqing Xia, Chengmin Chi, Guoshuai Zhao, Zheng Liu, et al. A neural corpus indexer for document retrieval. *Advances in Neural Information Processing Systems*, 35:25600–25614, 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, Jingrong Xie, Lilin Xu, Shisi Guan, Jingyan Zhu, Xiongwei Han, Xiaojin Fu, WingYin Yu, Han Wu, Wei Shi, et al. A survey of optimization modeling meets llms: Progress and future directions. *arXiv preprint arXiv:2508.10047*, 2025.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Haochen Yang, Ke Zhao, Mengyuan Ma, Xingyu Lu, Xiangfeng Wang, and Hong Qian. Optskills: Learning generalizable optimization skills from problem archetypes via cluster-based distillation. *arXiv preprint arXiv:2605.29829*, 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.

Bowen Zhang and Pengcheng Luo. Or-llm-agent: Automating modeling and solving of operations research optimization problem with reasoning large language model. *arXiv e-prints*, pp. arXiv–2503, 2025.

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.

Han Zhu, Xiang Li, Pengye Zhang, Guozheng Li, Jie He, Han Li, and Kun Gai. Learning tree-based deep model for recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 1079–1088, 2018.

A Experiment Details

We evaluate on eight public OR modeling benchmarks covering standard LP/MILP, complex formulations, and industrial problems. Since existing benchmarks may contain ambiguous descriptions or incorrect references, we use cleaned or provider-verified versions where available (Xiao et al., 2025). Together, these benchmarks provide a diverse test set for OR modeling.

A.1 Benchmark Details

NL4Opt. NL4Opt (Ramamonjison et al., 2023) was introduced for the NeurIPS NL4Opt competition and contains natural-language descriptions of linear programming problems. Its original tasks include entity recognition and mathematical formulation, but some instances contain ambiguous descriptions or unreliable references. Following the cleaned benchmark setting, we retain cases with consistent problem statements and verifiable reference answers, resulting in 213 evaluation problems.

MAMO MAMO (Huang et al., 2024) evaluates mathematical modeling from natural-language descriptions and separates problems into Easy and Complex subsets. The Easy subset mainly contains standard LP and MILP formulations, while the Complex subset includes longer descriptions and more interacting constraints. We remove instances with ambiguous or inconsistent references and retain 545 MAMO-Easy and 111 MAMO-Complex problems.

NLP4LP. NLP4LP (AhmadiTeshnizi et al., 2024) contains human-authored LP and MILP problems originally used to evaluate OptiMUS. The benchmark covers classical OR settings such as scheduling, allocation, and production planning, and requires translating natural-language requirements into executable formulations. After removing cases with unclear descriptions or unreliable references, we retain 178 verified problems.

OptiBench. OptiBench (Yang et al., 2025) is a large-scale benchmark for evaluating optimization modeling from natural-language descriptions. It covers LP, MILP, and related formulation settings across multiple domains and includes problems with different input structures. We retain 403 cases whose problem statements and reference answers can be consistently evaluated under our unified protocol.

ComplexOR. ComplexOR (Xiao et al., 2024) was introduced with Chain-of-Experts to evaluate LLM-based modeling on more challenging OR problems. Its instances require multiple interacting modeling decisions, including complex constraints, scheduling structures, and mixed-integer reasoning. Following verification of the problem statements and reference solutions, we retain 18 reliable cases.

IndustryOR. IndustryOR (Huang et al., 2025) contains optimization problems derived from industrial scenarios, including manufacturing, logistics, finance, and energy. The benchmark covers several mathematical programming classes and includes domain-specific constraints and decision rules. Because some instances are not suitable for reliable automatic evaluation, we retain 42 cases with consistent problem descriptions and reference answers.

OptMATH. OptMATH-Bench (Lu et al., 2025) was introduced with the OptMATH data-synthesis framework and contains challenging optimization-modeling problems with longer descriptions and diverse formulation structures. It includes LP, MILP, scheduling, and other structured optimization tasks. We retain 166 cases that can be evaluated consistently under the same solver-based protocol.

A.2 Baseline Details

Specialized optimization models. ORLM (Huang et al., 2025) trains optimization-specific models on instruction data for formulation and solver-code generation. LLMOPT (Shu et al., 2025) combines a structured five-element formulation representation with multi-instruction tuning and alignment. OptMATH (Lu et al., 2025) fine-tunes models on synthetic problem–formulation pairs generated and checked through bidirectional modeling. SIRL (Chen et al., 2026) uses solver-derived

rewards to supervise syntax, feasibility, and solution quality. StepORLM (Zhou et al., 2026) introduces generative process supervision and self-evolution to improve step-level modeling reasoning. We report their published results, marked with an asterisk in Table 1.

Agentic methods. CoE (Xiao et al., 2024) decomposes optimization modeling across multiple expert roles, coordinating problem understanding, formulation, and solution generation through staged interaction. OptiMUS (AhmadiTeshnizi et al., 2024) uses a structured workflow that separates mathematical formulation, solver-code generation, and debugging. OptiTree (Liu et al., 2026) organizes modeling thoughts through hierarchical problem relations and retrieves relevant subproblem knowledge to guide formulation of a new instance. AlphaOPT (Kong et al., 2026) maintains a solver-verified experience library, where reusable modeling insights are organized hierarchically and their applicability is refined through accumulated solving experience. OptSkills (Yang et al., 2026) groups problems by optimization archetypes and distills solution trajectories into reusable formulation skills, which are retrieved to guide new problems.

For controlled comparison, we evaluate local implementations of these methods under matched backbone and evaluation settings whenever possible. All methods produce Gurobi solver code as the final output. To isolate the modeling and retrieval components, the local adapters do not use execution-feedback code repair.

Retrieval baselines. We compare four alternative retrieval strategies over the same leaf-skill set. BM25 ranks skills by lexical similarity between the problem and each skill's title, summary, and retrieval cues, using $k_1 = 1.2$ and $b_{\mathrm{BM25}} = 0.75$, and returns the top three skills. Dense retrieval encodes the problem and the same skill text with BGE-M3, normalizes the embeddings, and retrieves the three skills with the highest cosine similarity. Flat LLM selection presents the LLM with the identifiers and titles of all leaf skills and allows it to select at most three; the full content of the selected skills is then passed to guidance synthesis. Static clustering partitions normalized skill embeddings into 16 clusters using k-means with seed 42 and ten initializations. The LLM first selects at most four clusters from their summaries and then selects at most three skills from the resulting candidate pool. The clustering structure is fixed throughout evaluation.

A.3 Implementation Details

Memory-construction data. We sample 800 cases without replacement from the 3,000-example Instruct3K training set (Huang et al., 2025) using seed 42 and process them in source order. Two cases with invalid references are replaced by unused examples from the same training set. Reference formulations are taken from the provided solutions, and objective labels are obtained by executing the corresponding reference-derived models first. We manually screened the construction set against all evaluation benchmarks to remove overlapping problem statements. The final 800 construction problems have no exact overlap with any evaluation instance.

Model and solver settings. We use DeepSeek-V4-flash for memory construction and evaluate both DeepSeek-V4-flash and DouBao-Seed-2.0-lite. Generation uses temperature zero with explicit thinking disabled. Generated Gurobi/Python programs are executed with a 60-second process timeout. The memory is fixed throughout evaluation. During retrieval, each visited paradigm retains at most four child paradigms, and at most three leaf skills are selected for synthesis. The final memory contains 247 skill nodes and 52 paradigm nodes, including the root.

B Method Details

This section clarifies the operational boundaries used in skill validation, memory evolution, retrieval, and evaluation.

Extraction and validation. A candidate skill is retained only when it captures a reusable formulation pattern consistent with the reference formulation. The consistency check examines variable semantics, objective accounting, constraints, domains, indexing, and applicability conditions. Source-specific entities or numerical values alone are not treated as reusable knowledge. Replay

Table 4: **Decision boundaries for skill memory evolution.** The table summarizes when local skill updates and structural reviews apply during the skill tree evolution process.

Stage	Operation	Boundary
Local update	INSERT	Introduces a new formulation pattern not covered in the routed region.
	SKIP	Duplicates an existing formulation pattern without adding useful detail.
	DROP	Valid on the source case but too specific or weakly transferable for reuse.
	ENRICH	Matches an existing formulation pattern while adding complementary detail.
Structural review	KEEP	The current region already forms a coherent retrieval group.
	REFINE	Local redundancy exists, but the current routing level remains appropriate.
	AGGREGATE	Multiple formulation families require separation into new paradigm nodes.

generates a fresh solution from the candidate skill; a failed candidate may be revised once and replayed.

Memory evolution. After validation, the candidate is routed to a local region and processed by a local update followed by structural review. The decision boundaries for the four local-update actions and three structural-review actions are summarized in Table 4. In the implementation, the insertion prompt refers to ENRICH as `refine`; this is distinct from the structural REFINE operation.

Structural constraints. Aggregation creates two to six child paradigms, assigns each source skill to exactly one child, and requires at least two skills per used child. A large sibling set is treated only as a signal for review; aggregation still requires distinct formulation families. All children of a paradigm share the same node type, and each nonroot node has one parent. After construction, a bottom-up pass updates paradigm descriptions and merges redundant sibling paradigms without rewriting skill content.

Routing and synthesis. If an incoming skill does not match an existing child paradigm, it is temporarily assigned to an *Other* bucket rather than forced into an incompatible branch. At inference time, routing retains at most $b = 4$ child paradigms per visited parent. Selections from different parents are combined, so the total frontier may exceed four nodes. Terminal regions contribute their direct skill children to the candidate pool, from which at most $K = 3$ skills are selected. If no candidates are reached, retrieval falls back to the full skill set. The selected skills are then synthesized with the original problem into a modeling plan for code generation.

Acceptance and evaluation. Replay validation and benchmark evaluation use the same objective-matching criterion:

$$a_i = \mathbb{I}[\sigma_i = \text{optimal} \wedge |\hat{y}_i - y_i| \leq 0.05]\,. \tag{10}$$

The tolerance is absolute. Missing outputs, nonoptimal status, execution errors, and timeouts count as failures. Pass@1 is the fraction of successful cases, and the overall score is the macro average across the eight benchmarks.

C MEMORY ANALYSIS

C.1 STRUCTURAL GROWTH ACROSS MEMORY SCALES

Table 5 reports the memory structure at four construction scales. As the number of construction cases increases from 100 to 800, the memory grows from 45 to 247 leaf skills and from 12 to 52 paradigm nodes. The mean leaf depth increases from 2.36 to 3.70, while the average paradigm branching factor stabilizes around 5–6. The number of terminal regions also grows from 10 to 39,

Table 5: Structural growth of the skill memory across construction scales. Average branching denotes the mean number of direct children per paradigm.

Cases	Skills	Paradigms	Terminal regions	Avg. branching	Mean leaf depth
100	45	12	10	4.67	2.36
200	84	19	15	5.37	2.77
400	148	31	24	5.74	3.16
800	247	52	39	5.73	3.70

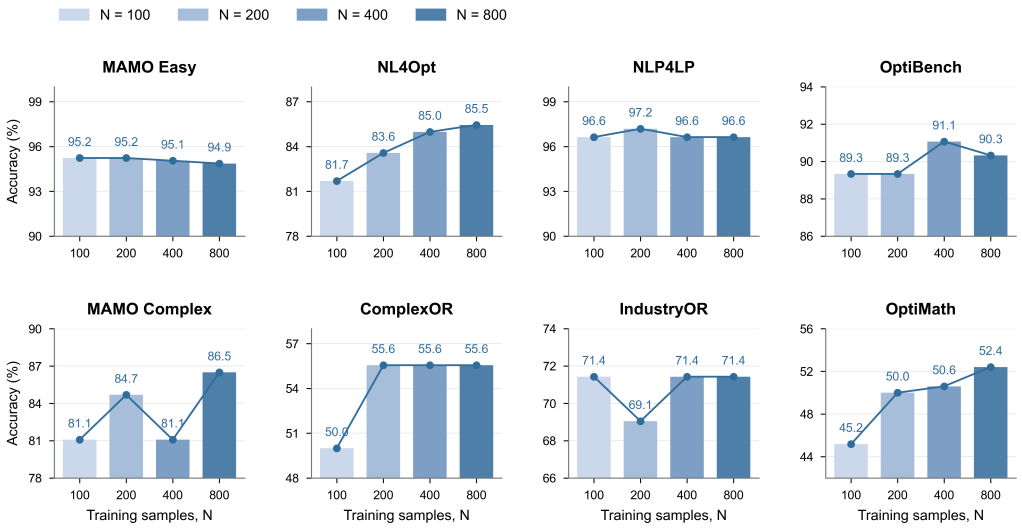

Figure 5: Pass@1 accuracy on each benchmark using memories constructed from 100, 200, 400, and 800 cases. All checkpoints use the same retrieval and generation protocol.

showing that additional skills are organized through progressively finer routing structure rather than only expanding existing leaf sets.

C.2 Performance Across Memory Scales

Figure 5 reports benchmark-level performance for the four memory checkpoints. Scaling has limited effect on benchmarks that already achieve high accuracy, such as MAMO-Easy and NLP4LP, while larger gains appear on NL4Opt, MAMO-Complex, ComplexOR, and OptMATH. In particular, NL4Opt improves from 81.69% to 85.45%, MAMO-Complex from 81.08% to 86.49%, and OptMATH from 45.18% to 52.41%. Although individual checkpoints are not strictly monotonic, the 800-case memory provides the strongest overall performance.

D Prompts

We provide the main prompts used by OptRoute in four stages: skill construction and validation, memory evolution, inference, and auxiliary recovery and evaluation. Braced fields denote runtime inputs, and doubled braces escape JSON in the templates.

Skill extraction and validation. The following prompt extracts one reusable formulation skill from a reference-grounded construction case.

D.2a: Reference-first induction

```
You extract one high-quality ordinary optimization-modeling skill from a
reference formulation. Read the problem and reference formulation. Decide
```

D.2a: Reference-first induction (continued)

```
whether the reference contains a complete, transferable ordinary formulation
skeleton. If yes, extract exactly one ordinary skill. If not, return null.

Problem: {problem}

Reference formulation: {reference}

Return one valid JSON object only:
{{
  "gate": {{
    "complete_formulation_skeleton": true, "transferable_beyond_case": true,
    "separates_common_nearby_error": true
  }},
  "skill": {{
    "id": "short_snake_case_skill_name", "kind": "skill", "scope": "ordinary",
    "title": "problem family + variable granularity + objective logic +
constraint skeleton",
    "summary": "when to use + nearest exclusions",
    "thought": "sets, parameters, variables, objective, constraints, domains,
boundary",
    "code_prompt": "Gurobi implementation skeleton without case-specific
numbers or entity names",
    "cues": ["short retrieval cue"]
  }}
}}

If any gate answer is false, return:
{{
  "gate": {{
    "complete_formulation_skeleton": false, "transferable_beyond_case": false,
    "separates_common_nearby_error": false
  }},
  "skill": null
}}

Gates. A complete skeleton specifies variables, objective, main constraints,
domains, and data/index support. Transferability requires reuse under another
business wrapper without copying case-specific names, constants, or entities.
The nearby-error gate requires preventing a plausible modeling mistake:
integer quantity versus binary selection, transportation flow versus TSP
tour, sparse valid tuples versus full Cartesian assignment, fixed-duration
scheduling versus pure sequencing, net versus gross flow, or inventory timing
versus immediate liquidation.

Skill content. Return at most one skill with `scope` set to `ordinary`, using
the reference formulation as authoritative. Lead title, summary, and thought
with formulation structure rather than the business shell; business words may
appear in `cues`. Make the title routeable and discriminative through problem
family, variable granularity, objective logic, and constraint skeleton. State
applicability and nearest exclusions in the summary. Use the thought labels
Sets, Parameters, Variables, Objective, Constraints, Domains, Boundary when
applicable. Specify named sets and parameters, indexed decision variables and
domains, objective expression and timing/accounting, core constraints, data
support, and all required structural constraints, including
conservation/transition, linking, valid arcs/tuples, non-overlap, or subtour
elimination.

Implementation. Translate the thought into a corresponding Gurobi skeleton:
model/addVars granularity, vtype and bounds, setObjective sense, core
quicksum constraints, structural constraints, and code-level exclusions. Keep
code_prompt isomorphic to the reference and thought. For tours, specify
binary arc variables, degree constraints, and required subtour elimination
(MTZ, flow-based SEC, or lazy SEC). For scheduling, specify start/completion
variables, ordering binaries only for stated conflicts or shared resources,
Big-M non-overlap from valid bounds, and objective type (makespan, tardiness,
or total completion). For network flow, specify balance direction, net/gross
convention, valid arcs, capacities, commodity indices when present, and
```

D.2a: Reference-first induction (continued)

```
source/sink treatment. For conic/quadratic models, specify the algebraic
relation and necessary bounds without brittle solver syntax.

Restrictions. Do not infer variables, objectives, or constraints from an
unsupported nearby formulation; supply complete runnable code or
case-specific numbers/entity names in code_prompt; or output tips, failure
reasons, dataset labels, ground-truth answers, or case-specific numeric
results. Include nearest-error exclusions only when the reference makes the
boundary clear. Prefer ASCII such as `<=`, `>=`, `sum`, and `times`.
```

The consistency prompt checks whether the extracted skill faithfully preserves the reference formulation and revises it when necessary.

D.2b: Reference consistency

```
You check whether one ordinary optroute skill faithfully follows the
reference formulation. Read the problem, reference formulation, direct
attempt, direct execution signal, and candidate ordinary skill. Decide
whether the candidate skill's thought and code_prompt are isomorphic to the
reference formulation. If the candidate imported wrong variables, objective
terms, constraints, domains, or accounting from the direct attempt, revise it
or reject it.

Problem: {problem}

Reference formulation: {reference}

Direct attempt code: {direct_code}

Direct execution signal: {direct_signal}

Candidate ordinary skill: {candidate_skill}

Return one valid JSON object only:
{{ "consistent": true, "reason": "short reason", "critical_differences": [],
"skill": null }}

If the candidate is inconsistent but can be repaired, return:
{{
  "consistent": false, "reason": "short reason", "critical_differences":
["difference"],
  "skill": {{
    "id": "short_snake_case_skill_name", "kind": "skill", "scope": "ordinary",
    "title": "problem family + variable granularity + objective logic +
constraint skeleton",
    "summary": "when to use + nearest exclusions",
    "thought": "sets, parameters, variables, objective, constraints, domains,
boundary",
    "code_prompt": "Gurobi implementation skeleton without case-specific
numbers or entity names",
    "cues": ["short retrieval cue"]
  }}
}}

If the candidate is inconsistent and has no reusable ordinary formulation
value, return:
{{
  "consistent": false, "reason": "short reason", "critical_differences":
["difference"],
  "skill": null
}}

Consistency. Treat the reference as authoritative and problem wording/direct
code as secondary. Match variable meaning and granularity, objective sense
and accounting, constraint skeleton, data/index support, domains, and
```

D.2b: Reference consistency (continued)

```
boundary. Keep code_prompt isomorphic to both thought and reference, with the
same variables, domains, objective sense, and constraints. Remove extra
variables or constraints imported from the direct attempt; mention them only
as useful nearest exclusions.

Corrections. Reject or revise substitutions of distinct used categories for
total quantity, TSP tours for transportation flow, full Cartesian products
for sparse valid tuples, invented conflicts for stated conflict pairs, or a
different time/unit basis for reference accounting. A repaired skill must
remain a transferable ordinary complete formulation skeleton.

Restrictions. Do not turn the skill into a global warning, local semantic
patch, code repair note, or case-specific answer, or include tips, markdown,
scores, dataset labels, ground-truth answers, or case-specific numeric
results. Prefer ASCII such as `<=`, `>=`, `sum`, and `times`.
```

If replay fails, the following prompt revises the skill using the reference formulation and execution feedback.

D.2c: Candidate revision after failed replay

```
You revise one candidate optimization-modeling skill after it failed its
source-case replay. Read the problem, reference formulation, candidate skill,
replay code, and replay execution signal. Rewrite the skill so that it can
guide a fresh code generator to the correct formulation. Revise the skill,
not the code. If the replay failure shows the skill has no reusable modeling
value, return null.

Problem: {problem}

Reference formulation: {reference}

Candidate skill: {candidate_skill}

Replay code generated from the candidate skill: {replay_code}

Replay execution signal: {replay_signal}

Return one valid JSON object only, with exactly this shape:
{{
  "diagnosis": "one short sentence naming what the candidate skill still
failed to specify",
  "skill": {{
    "id": "same_skill_id_or_better_short_snake_case_name", "kind": "skill",
    "scope": "ordinary",
    "title": "problem family + structural discriminator + variable
granularity + objective logic + constraint skeleton",
    "summary": "short routing summary with applicability boundary and
exclusion boundary",
    "thought": "minimal executable formulation skeleton with sets, indexed
variables, objective, key constraints, structural pitfalls, and boundary",
    "code_prompt": "Gurobi implementation skeleton without case-specific
numbers or entity names",
    "cues": ["short retrieval cue"]
  }}
}}

If the skill should not be kept, return:
{{ "diagnosis": "why the candidate should be rejected", "skill": null }}

Revision. Return at most one complete reusable skill with `scope` set to
`ordinary`. Preserve only information that helps a fresh code generator
choose and implement the correct model. Name any missing correctness-critical
modeling decision in title, summary, or thought; keep the summary short. Make
the thought solver-ready: sets and indices, decision variables, objective
```

D.2c: Candidate revision after failed replay (continued)

```
accounting, constraints, domains, data support, and implementation-sensitive
structures.

Implementation. Translate the thought into a Gurobi skeleton with addVars
indices, vtype and bounds, setObjective sense, quicksum constraints, required
structural constraints, and code-level exclusions. Keep code_prompt
isomorphic to the thought and reference, without unsupported variables,
objective terms, or constraints.

Restrictions. Do not output a code patch, error explanation, or case-specific
answer; copy replay code; rely on legacy tips for correctness-critical
variables, objectives, constraints, domains, or boundaries; or include tips,
dataset labels, ground-truth answers, predicted values, or case-specific
numeric results. Keep the skill self-contained in title, summary, thought,
and cues. Prefer ASCII such as `<=`, `>=`, `sum`, and `times`.
```

Memory evolution prompts. The following prompts determine how validated skills are integrated and how the local routing structure is reorganized.

The insertion prompt determines how a validated skill should affect the routed local region.

D.2d: Insertion decision

```
You decide whether one candidate optroute skill should enter the tree. Read
the candidate skill, routed insertion parent, ancestor path, sibling skills,
child paradigms, and nearby child skill summaries. Decide whether to insert,
skip, drop, or refine.

Input JSON: {context}

Return one valid JSON object only:
{{
  "decision": "insert|skip|drop|refine", "target_skill_ids": ["skill_id"],
  "instruction": "short instruction for refine if needed", "reason": "short
decision reason"
}}

Decisions. `insert` admits a new transferable complete skeleton only if it
changes variable design, objective accounting, constraint skeleton,
index/time structure, variable domain, or applicability boundary. `skip`
applies when an existing skill covers the candidate at equal or better
specificity, with no new formulation-critical information. `drop` applies to
low-information, non-transferable, overly local, generic LP/IP, or incomplete
skills, including generic continuous resource allocation with linear
constraints, integer cost minimization with capacity constraints, or linear
programming with bounds. `refine` merges useful additions only into targets
with the same variables, objective accounting, constraints, domains,
data/index support, and boundary.

Restrictions. Do not insert solely because business-scenario words differ, or
include markdown, scores, or extra keys. Use only sibling-skill or
nearby-child-skill ids in `target_skill_ids`; the list may be empty for
`insert` and `drop`.
```

When local consolidation is required, this prompt updates only the targeted skills without modifying unrelated siblings.

D.2e: Targeted skill refinement

```
You refine a targeted set of sibling optroute skills. Use the instruction,
target skills, optional candidate skill, and sibling context to refine
sibling skill nodes: merge duplicates, rewrite vague but useful skills, drop
low-information skills, or absorb a candidate into an existing skill. Do not
freely edit the whole bucket.
```

D.2e: Targeted skill refinement (continued)

```
Input JSON: {context}

Return one valid JSON object only:
{{
  "updated_nodes": [
    {{
      "id": "existing_or_new_skill_id",
      "fields": {{
        "title": "problem family + variable granularity + objective logic +
constraint skeleton",
        "summary": "when to use + nearest exclusions",
        "thought": "sets, parameters, variables, objective, constraints,
domains, boundary",
        "code_prompt": "Gurobi implementation skeleton without case-specific
numbers or entity names",
        "cues": ["cue"], "support": 2
      }},
      "source_ids": ["skill_id"]
    }}
  ],
  "deleted_node_ids": [], "reason": "short reason"
}}

Scope and usefulness. Operate only on `target_skill_ids` and the optional
`candidate_skill`. A useful skill changes variable design, objective
accounting, constraint skeleton, index/time structure, variable domain, or
applicability boundary. Same formulation means matching decision-variable
meaning and granularity, objective accounting, constraint skeleton,
data/index support, domains, and applicability/exclusion boundary.

Refinement. Merge duplicates or same-meaning skills, rewrite vague useful
skills into routeable skeletons, drop low-information skills, and absorb the
optional candidate when its formulation meaning matches an existing skill.
Delete generic LP/IP, resource-allocation, bounds, or linear-constraint
skills without independent routing value. For equal formulations, retain the
stronger executable skeleton and useful details in title, summary, thought,
and cues. For nearby distinct formulations, preserve both and clarify
title/summary boundaries if they are targets. Preserve or improve ordinary
code_prompt while keeping its variables, domains, objective sense, and
constraints isomorphic to the thought.

Restrictions. Do not modify unrelated siblings; merge different variable
designs, objective timing/accounting, constraint families, data/index
support, domains, boundaries, or Gurobi variable/structural-constraint
requirements; or output tips, markdown, scores, or extra keys.
```

After the local update, the bucket-review prompt decides whether the region should be kept, refined, or structurally aggregated.

D.2f: Bucket review

```
You review one current optroute bucket and decide whether an organization
event is needed. Read the parent, ancestor path, sibling paradigms, current
children, and the most recent insert decision. You choose one operation for a
bucket of sibling skills. Trigger an event when refinement or aggregation
would improve routing. Do not create final paradigms or final skill text here.

Input JSON: {context}

Return one valid JSON object only:
{{
  "action": "none|refine|aggregate", "target_skill_ids": ["skill_id"],
  "instruction": "short instruction for the next operation", "reason": "short
decision reason"
}}
```

D.2f: Bucket review (continued)

```
Definitions. A useful skill is a transferable complete skeleton that changes
variables, objective accounting, constraint skeleton, index/time structure,
variable domain, or applicability boundary. A useful paradigm improves
routing between complete formulation families. Review triggers events; refine
writes skill changes; aggregate creates child paradigms. About 30 sibling
skills is a routing-pressure warning, not a hard threshold.

‘none‘. Retain a clear bucket with no obvious duplicates, low-information
skills, or mixed complete families. Size alone does not require an event when
all skills share variable granularity, objective accounting, constraint
skeleton, index/time structure, domains, and applicability boundary.

‘refine‘. Target duplicates, near-duplicates, vague useful skills,
low-information skills, or local fragments; absorb a candidate into an
existing same-meaning skill. Choose this when targets can be repaired without
child paradigms, including crowded buckets needing generic, duplicate, or
weakly routeable skills merged, rewritten, or deleted before splitting.
Prefer refine over none above about 30 skills when obvious generic LP/IP,
near duplicates, or vague route labels remain. List only ids needing work.

‘aggregate‘. Split mixed complete families when flat storage confuses
routing. Search actively for real splits in large buckets, and prefer
aggregate over none above roughly 30 skills when confusable cores differ in
variable granularity, objective timing/accounting, constraint skeleton,
valid-index support, flow balance, routing/subtour logic, scheduling
disjunctions, inventory balance, assignment, or facility/network design. For
crowded buckets, require at least 2 child families, each receiving multiple
skills with clear sibling exclusions. ‘target_skill_ids‘ may be empty because
aggregation inspects the full bucket.

Restrictions. Do not split solely by size, business names, or isolated
binary, integer, continuous, capacity, ratio, bounds, or objective-weight
keywords; chase ‘preferred_child_paradigms‘ (only an upper preference for the
later operation); or output markdown, scores, final paradigms, rewritten
skills, or extra keys. Make ‘instruction‘ operational about what to preserve,
merge, drop, or separate.
```

When aggregation is selected, this prompt induces child paradigms and assigns the current skills to distinct formulation families.

D.2g: Aggregation into child paradigms

```
You create high-quality routing paradigms for one optroute bucket. Read the
parent, ancestor path, sibling paradigms, bucket skills, and review
instruction. You split one bucket of skills into child modeling families only
when they improve routing between complete formulation families. Assign every
bucket skill to exactly one new child paradigm.

Input JSON: {context}

Return one valid JSON object only:
{{
  "new_paradigms": [
    {{
      "id": "short_snake_case_name", "kind": "paradigm",
      "title": "compact formulation family boundary",
      "summary": "use when ...; exclude when ...; sibling distinction ...",
      "thought": "routing description of variables, objective logic,
constraint skeleton, boundaries, and nearest exclusions",
      "cues": ["cue"]
    }}
  ],
  "assignments": [
    {{ "skill_id": "skill_id", "paradigm_id": "paradigm_id" }}
  ],
```

D.2g: Aggregation into child paradigms (continued)

```
  "reason": "short reason"
}}

Families. Use the review instruction as diagnosis, not fixed child
directions. Create 2 to 6 child paradigms, preferring fewer real boundaries.
Each must represent a complete formulation family with variables, objective
logic, and constraint skeleton, rather than one component such as binary
variables, capacity, ratios, objective weights, bounds, or generic LP/IP.
Each summary must state `use when`, `exclude when`, and a sibling
distinction. Reduce routing pressure in large buckets; a child should usually
contain no more than about 30 skills unless they truly share a formulation
core.

Assignment. Every input in `bucket_skills` must appear exactly once in
`assignments`, using only new ids from `new_paradigms`. Each paradigm must
receive at least `min_skills_per_new_paradigm` skills.

Restrictions. Do not duplicate/paraphrase ancestors or repeat the parent
title with minor wording changes; use pure, classic, general, miscellaneous,
other, or additional constraints as child names unless the title also
identifies the variable, objective, and constraint difference; split solely
by business names or isolated binary, integer, continuous, capacity, ratio,
bounds, or objective-weight keywords; or include markdown, scores, or extra
keys.
```

This prompt assigns a formulation-based name to a temporary *Other* bucket before further organization.

D.2h: Temporary-bucket naming

```
You rename a temporary routing bucket before it is organized. Read the
sibling skills inside this temporary bucket and give the bucket a real
modeling-family name.

Input JSON: {context}

Return one valid JSON object only:
{{
  "updated_parent": {{
    "fields": {{
      "title": "specific modeling family", "summary": "short routing summary",
      "thought": "what shared complete modeling boundary this bucket
represents",
      "cues": ["cue"]
    }}
  }},
  "updated_nodes": [], "deleted_node_ids": []
}}

Naming. Rename only the bucket, using a shared distinguishing property
visible in its skills and a title more specific than the parent. Prefer a
name that remains useful after splitting into child families, with a short
routing summary.

Restrictions. Do not add, delete, merge, or move skills; title the bucket
"Other"; repeat or paraphrase the parent family; or include markdown, scores,
or extra keys.
```

After construction, this prompt refines paradigm descriptions and merges redundant sibling paradigms while preserving skill content.

D.2i: Bottom-up paradigm refinement

```
You refine sibling paradigm nodes for routing clarity. Update only paradigm
wording and duplicate paradigm structure. Do not modify, merge, delete, or
reorganize skills.

Input JSON: {context}

Return one valid JSON object only:
{{
  "updated_parent": {{
    "fields": {{ "summary": "short routing summary", "thought": "updated
parent thought", "cues": ["cue"] }}
  }},
  "updated_child_paradigms": [
    {{
      "id": "existing_survivor_id",
      "fields": {{
        "title": "compact modeling boundary",
        "summary": "use when ...; exclude when ...; sibling distinction ...",
        "thought": "updated paradigm thought", "cues": ["cue"], "support": 2
      }},
      "source_ids": ["paradigm_id"]
    }}
  ],
  "deleted_paradigm_ids": []
}}

Scope. Operate only on the supplied sibling paradigm children. Rewrite
titles, summaries, thoughts, and cues as needed; include in
`updated_parent.summary` a concise one-layer map of direct-child boundaries
and distinctions without copying full child thoughts. Rename generic `Other`
only when sibling context supports a real modeling boundary.

Merging. Merge duplicate siblings only if the survivor does not mix skill
children with paradigm children. Keep one existing survivor id, put all
merged sibling ids in `source_ids`, and list removed ids in
`deleted_paradigm_ids`.

Restrictions. Do not create new paradigm ids; change skill titles, summaries,
thoughts, parents, or existence; or include markdown, scores, or extra keys.
```

Inference prompts. At test time, OptRoute first navigates the evolved hierarchy, then synthesizes retrieved knowledge into guidance for solver-code generation.

The navigation prompt selects relevant paradigm branches or leaf skills from the current candidate set.

D.2j: Navigation and skill selection

```
You route an optimization problem through a compact tree of modeling families
and skills. Select candidate node ids using only the provided titles,
summaries, cues, and support counts.

Input JSON: {context}

Return one valid JSON object only:
{{ "decision": "descend", "selected_ids": ["candidate_id"] }}

Selection. Use only ids in `candidates` and respect `min_count` and
`max_count`. For `paradigm`, select the closest 1 or 2 child families for the
primary formulation. For `insert_parent`, descend only when one child is
clearly more specific for the new skill; use `stay_here` with empty
`selected_ids` when the parent fits but no child is clearly suitable. For
`skill`, prefer fewer, stronger complete ordinary skills directly matching
the problem. `max_count` is an upper bound, not a quota.
```

D.2j: Navigation and skill selection (continued)

```
Relevance. Omit irrelevant or weakly related skills that could harm
generation. Prefer summaries matching the full modeling family rather than
one variable, objective, or constraint component, with defining relationships
present in the problem that distinguish it from superficial alternatives.
Treat exclusion boundaries as hard negatives.

Restrictions. Do not select a skill whose exclusion matches the problem, or
include explanations, scores, markdown, or ids outside the candidate list.
```

The synthesis prompt combines the retrieved skills into a problem-specific modeling plan.

D.2k: Modeling-plan synthesis

```
You synthesize retrieved optimization-modeling knowledge into one coherent
modeling plan. Use the problem statement as the source of truth. Use
retrieved modeling skills as candidate complete formulation frames. Produce
one complete modeling plan for code generation.

Problem: {problem}

Retrieved knowledge: {retrieval}

Return plain text only.

Plan. Identify what the problem actually asks, then state decision-variable
granularity, the real quantity to optimize, and required resource, capacity,
logic, time, assignment, balance, or feasibility constraints. Use retrieved
skills only when applicable; reject conflicts with the statement or matching
exclusions even when words overlap. Preserve each selected skill's structural
discriminator and defining relationships; adapt a skill that is too strong
for the exact details rather than copying it as a hard rule.

Restrictions. Do not substitute a superficially similar template, broaden
relationships in ways that change intended feasible decisions, or reveal
labels, reference answers, or ground-truth objective values. Include concise
cautions about likely mis-modeling; keep the complete plan compact and
directly useful for solver-code generation.
```

The final generation prompt converts the problem and synthesized plan into executable Gurobi code.

D.2l: Gurobi code generation

```
You are an optimization engineer writing executable Gurobi Python code.
Generate executable code for the problem using the problem statement and
retrieved optroute guidance.

Problem: {problem}

Retrieved optroute guidance: {modeling_plan}

Return Python code only. Do not include markdown or prose.

Required stdout contract
After solving, the final stdout line must be exactly one JSON object and
nothing else:
{{ "status": "optimal|infeasible|unbounded|error", "objective_value": 0.0 }}

JSON contract. Use `import json` and `print(json.dumps(result))` with exactly
the keys `status` and `objective_value`. Status must be "optimal",
"infeasible", "unbounded", or "error". Set a numeric objective only for
"optimal"; otherwise use Python `None`. Do not print markdown, explanations,
logs, solver output, variable values, or other text before or after the JSON.

Modeling. Use `import gurobipy as gp` and `from gurobipy import GRB`. Build
directly from numeric problem data, matching the stated decision process in
```

D.2l: Gurobi code generation (continued)

```
variable granularity, objective, and constraints. Use applicable retrieved
ordinary skills as complete formulation skeletons; follow their Gurobi
variable creation, objective sense, quicksum constraints, and
structural-constraint patterns. Before coding, internally map sets/indices,
parameters, variables/domains, objective, core constraints, and indispensable
structural constraints to this problem. If the skill does not match, derive
the structure from the statement rather than force it or guess a nearby
formulation.

Structure. Preserve the selected skill's structural discriminator and
explicitly implement the defining relationships needed for decisions and
feasibility: dependencies, permitted combinations, and requirements on the
complete solution. Separate local conditions do not suffice when
relationships across decisions are required.

Implementation. Use `model = gp.Model("model")`, immediately followed by
`model.setParam("OutputFlag", 0)`. Use explicit loops or dictionaries for
indexed data and simple runnable Gurobi syntax. For index-specific bounds,
use dictionaries or set `.LB` / `.UB` after creation.

Restrictions. Do not use callable `lb=...`, `ub=...`, or `obj=...` in
addVar/addVars; assemble unrelated variable, objective, or constraint
fragments; add constraints solely from a nearby retrieved pattern; replace
specific structure with a broader template; read/write files, call network
APIs, spawn processes, or ask for input. Unless optimal, serialize
`objective_value` as `null` (Python `None`).
```

Auxiliary prompts. The following prompts are used only for recovering malformed organization outputs or evaluating cases that cannot be scored automatically.

This prompt repairs malformed JSON returned by a memory-organization step.

D.2m: Organization JSON recovery

```
You repair one invalid JSON response from a optroute organization step.
Convert the malformed response into the exact JSON object requested by the
original prompt. Preserve the intended content; fix only JSON syntax,
truncation artifacts, markdown fences, stray prose, trailing commas, missing
quotes, or unfinished braces.

Input JSON: {context}

Recovery. Return one valid JSON object following the schema in
`operation_prompt` and `original_prompt`. If the malformed response is
incomplete, retain only complete safely recoverable fields and return the
smallest valid object for the operation.

Restrictions. Do not invent skill ids, paradigm ids, assignments, source ids,
or modeling content beyond what repair requires, or include markdown,
comments, scores, or explanations.
```

This prompt repairs a parsed organization output that violates structural validation constraints.

D.2n: Organization validation recovery

```
You repair one parsed JSON payload that failed optroute organization
validation. Revise the payload so it satisfies the validation errors while
preserving the intended organization decision and using only ids available in
the original context.

Input JSON: {context}

Validation. Return one valid JSON object with the `operation_prompt` schema.
Use only existing skill ids in assignments, updates, source_ids, and
```

D.2n: Organization validation recovery (continued)

```
deleted_node_ids. For aggregation, assign every input skill exactly once to a
newly created paradigm, give each used paradigm enough skills, and ensure new
paradigm ids do not duplicate existing ids. For final skill refinement, merge
only exact/near duplicates, keep merge-source lists small, and leave
`deleted_node_ids` empty because `source_ids` removes merged sources. For
paradigm refinement, keep child kinds pure and delete only paradigms absorbed
into a surviving sibling.

Restrictions. Do not create new skill ids in final skill refinement or
include markdown, comments, scores, or explanations. Preserve distinct
complete modeling schemes; repair structure without broad semantic rewriting.
```