CoME: Empowering LLMs to Solve Complex Vehicle Routing Problems via Cross-Constraint Modeling Experience

Anonymous authors
Paper under double-blind review

Abstract

Modeling and solving real-world vehicle routing problems (VRPs) with intricate constraints has long required substantial domain expertise. Large language models (LLMs) have shown promise in solving VRPs, but existing approaches face a quality-coverage trade-off. Those that integrate mature solvers preserve solution quality but lack the composition knowledge to model composite constraints; those that generate their own solvers handle composite constraints but sacrifice solution quality. Experience reuse improves LLMs without retraining by storing reflections and rationales, and can supply this knowledge. However, it requires a reliable source, which is precisely what the model's own attempts lack. To supply this knowledge reliably, we propose CoME, which automatically learns cross-constraint modeling experience from verified solver code, enabling LLMs to intelligently exploit mature solvers for VRPs with composite constraints. This experience spans constraint-level and interaction knowledge and is stored in a structured library. CoME applies this learned experience through a three-stage multi-agent pipeline with cooperative agents verifying each stage's output to ensure correctness throughout the generation. CoME outperforms existing LLM-based baselines across 88 VRP variants and standard benchmarks in both accuracy rate and solution quality. Experiments on other operations research problems and across a range of solvers validate the effectiveness and generality of our method.

1 Introduction

Vehicle Routing Problems (VRPs) are a fundamental class of combinatorial optimization problems that underpin logistics (Konstantakopoulos et al., 2022; Bochtis & Sørensen, 2010), transportation (Zhang et al., 2022), and retail distribution (Toth & Vigo, 2014), with direct consequences for cost, emissions, and service quality. While the canonical VRP is already NP-hard, real-world settings compound this difficulty by imposing additional constraints such as time windows, capacities, and multiple depots. In response, decades of operations research effort have produced mature solvers, including Gurobi, OR-Tools, PyVRP, and LKH, with flexible, general-purpose modeling interfaces that handle diverse problem variants. However, using these solvers still demands considerable expertise: a practitioner must map their specification to a precise, domain-specific formulation and implement it as solver code. This expertise barrier prevents non-experts from adopting optimization, even where it could substantially improve their operations.

Researchers have begun leveraging LLMs to lower this barrier by translating natural-language VRP descriptions into executable code or generating solutions directly. Early attempts (Yang et al., 2024; Liu et al., 2024c) prompt LLMs to generate solutions directly but fall short in optimality and feasibility. Some approaches (Romera-Paredes et al., 2024; Liu et al., 2024a; Ye et al., 2024; Liu et al., 2026) integrate LLMs with evolutionary computation to evolve basic heuristics; however, they remain limited to conventional VRPs and require domain-specific knowledge. Recent works aim to extend beyond conventional VRPs, broadly falling into two directions. The first direction emphasizes the automation of the pipeline, from extracting instance-specific information to generating the solver itself, including (Zhang et al., 2025a; Li et al., 2026). This approach improves flexibility in modeling, but sacrifices the solution quality that mature solvers provide. The second direction maintains solution quality by invoking mature solvers but is limited by the LLM's difficulty in mod-

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.

eling composite constraints. Yet real-world settings demand both consistent solution quality and the ability to handle composite constraints.

Given that solution quality remains paramount, we adopt the solver-integrated approach and tackle its primary bottleneck: not modeling individual constraints in isolation, but correctly composing heterogeneous constraints within solver APIs. To mitigate the limited knowledge of LLMs, recent work (Jiang et al., 2025b) incorporates external domain knowledge via Retrieval-Augmented Generation (RAG). Although it reduces the knowledge gap, it primarily retrieves per-constraint knowledge, thereby shifting the burden of compositional modeling entirely onto the LLM. Consequently, an effective method must capture cross-constraint composition knowledge and reuse it across problems.

Addressing this gap requires reusable knowledge of both how to model individual constraints and how their implementations must change when constraints interact. Since such interaction knowledge is rarely available in isolated documentation and cannot generally be recovered from per-constraint retrieval, it must either be manually engineered by experts, which is costly, or automatically accumulated from the LLM's successful trajectories. Recent approaches such as ExpeL (Zhao et al., 2024) and AgentKB (Tang et al., 2025) show that LLMs improve by reusing such trajectories, such as reflections or code edits from past attempts. These methods have proven effective on reasoning tasks such as information retrieval (Yang et al., 2018) and web browsing (Yao et al., 2022). In VRP domain, however, LLMs often handle individual constraints well but struggle when they interact. As Figure 1 demonstrates, modeling failures are driven more by cross-constraint interactions than by constraint count. At the same constraint count, variants involving more constraint interactions exhibit substantially lower solve rates, with performance falling to zero on the most complex variants. Therefore, relying on self-generated experience alone cannot reliably provide the compositional knowledge needed for these cases.

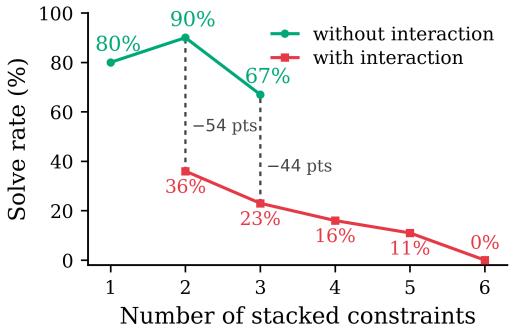


Figure 1: Modeling failures are driven more by cross-constraint interactions than by constraint count (Claude Opus 4.5 under standard prompting). Details of this pilot experiment can be found in Appendix D.1.

In this work, we automatically learn compositional modeling knowledge from gold-standard implementations. This knowledge covers both individual constraints and cross-constraint interactions. We propose a multi-agent inference framework that integrates this modeling experience through three sequential stages with evaluation and revision loops. Our contributions are summarized as follows:

- We propose **CoME** (Cross-Constraint Modeling Experience), which automatically learns modeling hints for individual constraints and their interactions from verified solver code and accumulates them into a structured library.
- We develop a multi-agent inference framework that composes the learned hints into solver code for complex VRP variants with composite constraints. The framework decomposes the task into three stages, with each including cooperative agents that evaluate and revise the output in a loop.
- We evaluate the multi-agent framework with CoME on 88 VRP variants, including standard benchmarks, constrained TSP and VRPs, and other OR problems across multiple solvers. Extensive results demonstrate its effectiveness and generality, while achieving higher accuracy and better solution quality than existing LLM-based approaches.

2 Preliminaries

2.1 Vehicle Routing Problems

Vehicle Routing Problems concern the optimisation of routes for a fleet of vehicles that serve a set of geographically distributed customers from one or more depots, subject to operational constraints

and with the objective of minimising total cost. A VRP instance is defined on a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where the node set $\mathcal{V} = \{0, 1, \ldots, n\}$ comprises a depot indexed 0, and n customers; the edge set $\mathcal{E} = \{e_{ij} \mid i, j \in \mathcal{V}\}$ represents the road segments between them, each carrying a traversal cost c_{ij}. The problem is formulated as

$$\min \sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} c_{ij}\, x_{ij}, \qquad \text{subject to} \quad \mathbf{x} \in \mathcal{C}, \tag{1}$$

where $\mathbf{x} = \{x_{ij} \mid i, j \in \mathcal{V}, i \neq j\}$ is the set of binary decision variables, with $x_{ij} = 1$ if the edge from i to j is used in a route and 0 otherwise. Almost all VRP variants share a common core of base constraints: each customer is visited exactly once, and every vehicle departs from and returns to a depot while conserving flow along its route. Beyond the base constraints, practical VRPs impose additional operational requirements, each of which further restricts the feasible region $\mathcal{C}$. We consider an additional taxonomy of thirteen canonical VRP constraints defined in full in Table 5 of Appendix C. Each VRP variant is defined by a subset of constraints drawn from the taxonomy.

2.2 Problem Formulation

Unlike prior optimization-modeling methods (Xiao et al., 2024; AhmadiTeshnizi et al., 2024), CoME solves a code-generation problem without a mathematical-model-formulation step. The input to the LLM is a problem description P and a function signature F, which specifies the function's name, its parameters, and their types, with each parameter described in a docstring (see Appendix C for examples of P and F). The LLM is responsible for implementing the `solve` function by invoking a designated solver. Once the function is generated, it applies to all instances of that VRP variant. In this paper, we propose that the code generation is conditioned on modeling hints retrieved from the library $\mathcal{L}$ that is collected and structured in an offline learning fashion. Formally, given a description P and signature F, a retrieval function selects a relevant subset $H_s \subseteq \mathcal{L}$, and an LLM π_θ generates a program $y \sim \pi_\theta(y \mid P, F, H_s)$ that a solver can execute to solve the variant.

3 Methodology

Our method comprises offline learning and online inference phases. In the offline phase, CoME automatically learns constraint and interaction modeling experience from reference implementations and stores it in a schema-based structured library. In the online phase, given a new task, our multi-agent inference framework retrieves the relevant hints and composes them to generate solver code. It operates in three stages: problem identification, library-guided planning, and code generation, each employing the same generation-evaluation-revision loop.

3.1 Learning Cross-Constraint Modeling Experience

CoME builds on experience learning, which reuses structured experience from past problem-solving. Since self-generated experience is unreliable for composite VRPs, as shown in Figure 1, we instead learn from verified ground-truth solver code that correctly models each variant's constraints and objective. Grounding the hints in correct code ensures they encode valid modeling knowledge, which transfers reliably to new tasks. A further limitation of existing methods (Shinn et al., 2023; Zhao et al., 2024; Tang et al., 2025) is that they preserve experience as free-form text or edits, a flat list without explicit applicability semantics, whereas solving VRPs inherently requires a more efficient organization of acquired knowledge. We organize the modeling experience into a structured schema with explicit semantic applicability.

Modeling hints are learned from verified gold-standard solver implementations in three steps, as shown in Figure 2. First, an LLM analyzes the description P and signature F, identifying the active constraint set $\mathcal{C}_v \subseteq \mathcal{C}$ and the objective. Second, given the description P, ground-truth code $\mathbf{g}_v$, and the constraint set $\mathcal{C}_v$, the LLM distills modeling hints for the individual constraints, their interactions, and the search strategy; see Figure 6 in Appendix C for an example. We detail the resulting schema below. Third, because LLMs can misinterpret source code and produce unfaithful descriptions (Wan et al., 2025), each hint is verified by using it to solve the problem and added to the library $\mathcal{L}$ only if the generated code reproduces the same ground-truth objective; otherwise, generation is repeated until it does. The modeling hint learning prompt template is provided in Appendix H.1.

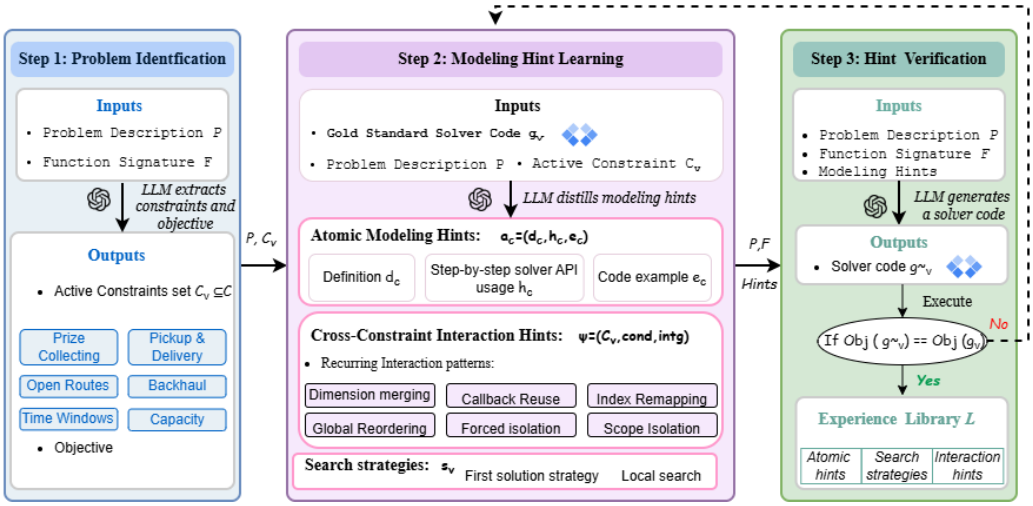


Figure 2: Overview of CoME's modeling hint learning process. It comprises three steps, namely problem identification, modeling hint learning, and hint verification.

Inspired by how NP-hard VRPs are solved through external solver APIs, we design a structured schema around three factors that determine correctness and optimality: per-constraint implementation, inter-constraint interaction, and search strategy. The modeling hint learning accordingly comprises the following three sections.

Atomic modeling hints. Atomic modeling hints capture how each constraint is implemented in isolation within the solver API. Each constraint c is captured in isolation: $a_c = (d_c, h_c, e_c)$ records the definition d_c, a natural-language hint h_c for step-by-step solver API usage, and a concrete code example e_c.

Interaction hints. Two or more constraints require an interaction hint when their correct joint implementation is not the union of their individual implementations: $\mathcal{I}(c_1 \cup c_2) \neq \mathcal{I}(c_1) \cup \mathcal{I}(c_2)$. Each hint captures the modeling requirement that closes this gap, knowledge that cannot be derived from any single constraint in isolation. Formally, an interaction hint is a triple $\psi = (C_v, \text{cond}, \text{intg})$, where C_v is the set of participating constraints, cond is the activation condition, and intg is the integration logic. These hints span several recurring forms, including dimension merging, callback reuse, forced separation, index remapping, scope isolation, and global reordering. For example, service time and time windows constraints share a single callback, capacity and open start-open end constraints require index remapping, and mixed backhaul and multi-depot constraints demand a global reordering of the problem data.

Search strategies. Correct modeling is necessary but not sufficient; solution quality also depends on the solver's search strategy. For each variant v, we record a search-strategy section s_v specifying the initial route construction heuristic and local search operators extracted directly from the ground-truth code $\mathbf{g}_v$. Together, this three-section schema comprehensively capture how individual constraints are expressed, how cross-constraint interactions compose, and how the resulting model is solved.

3.2 Multi-Agent Inference Framework

Figure 3 provides an overview of our framework. At inference, the framework takes a natural-language description problem and function signature (P, F) and produces executable solver code through three sequential stages: problem identification, library-guided planning, and code generation. It first identifies the active constraints and objective, then retrieves the relevant modeling knowledge from the offline-learned library $\mathcal{L}$ and composes it into an implementation plan, and finally generates solver code. Each stage runs a generation-evaluation-revision loop that validates its output before advancing to the next.

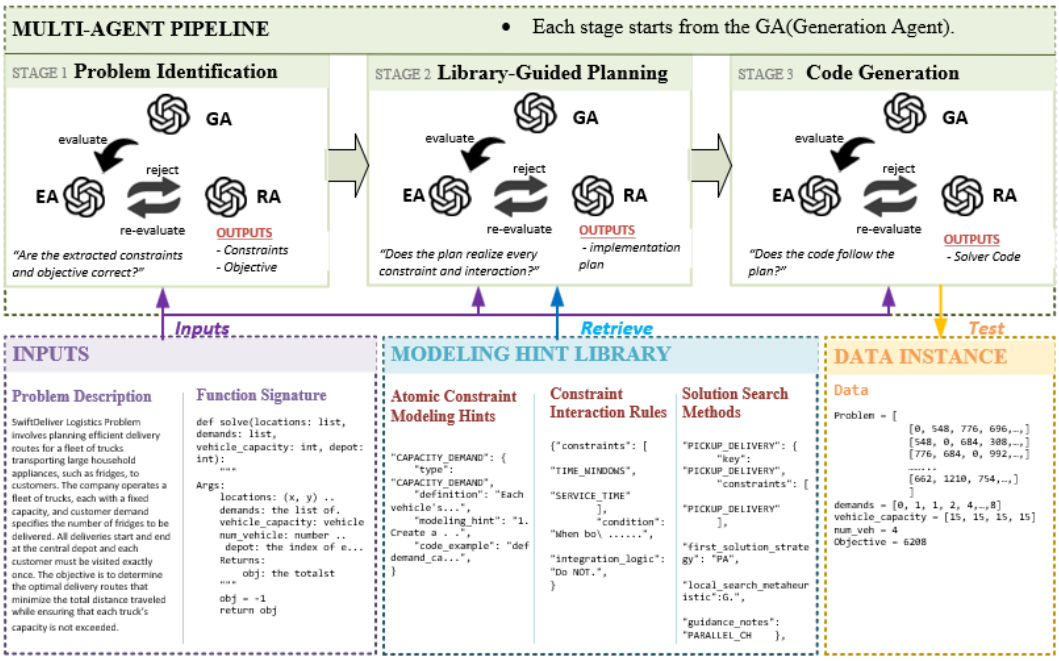


Figure 3: Our proposed multi-agent inference framework.

Generation-Evaluation-Revision. Single-shot code generation is prone to semantic faults, such as omitted sub-steps, misaligned index transformations, or extraneous constraints, and frequently yields executable code that implements incorrect logic (Lee et al., 2025). To address this, we structure each stage as a generation-evaluation-revision loop carried out by three cooperative agents. The generation agent (GA) progressively generates the required stage outputs, while the evaluation agent (EA) and revision agent (RA) iteratively evaluate and refine the output until it meets all requirements and constraints. This iterative process continues until the output passes validation or an iteration limit is reached.

3.2.1 THREE SEQUENTIAL STAGES

Problem Identification. Given the problem description P and function signature F, GA produces problem type, a set of constraints and the objective $E = (\text{type}, \{c_i\}_{i=1}^{n}, \text{obj})$ along with their explanations. Each constraint $c_i = (\tau_i, r_i)$ pairs a canonical type $\tau_i \in \mathcal{C}$ with a natural-language rationale r_i grounding it in P. The extraction prompt instructs GA to use signature parameters such as `time_windows` or `demands` as confirmatory evidence and also enforces mutual-exclusion rules to distinguish closely related variants such as backhaul, mixed backhaul, and pickup-and-delivery. EA then verifies that the extracted constraints and objective are correct against P. Upon failure, it returns a structured list of issues, and RA re-extracts under this feedback until the output is accepted. The identified constraints are then used to query the library $\mathcal{L}$ for relevant modeling knowledge. The verified representation E^* is passed on only once it faithfully reflects (P, F).

Library-Guided Planning. Directly composing retrieved modeling knowledge into executable solver code is error-prone, as it requires the LLM to reconcile isolated constraint hints and their interactions at once. We instead introduce an explicit planning stage, since decomposing a complex task into steps simplifies each step and improves overall generation accuracy (Jiang et al., 2024; Lei et al., 2024). While self-planning methods rely on the LLM's intrinsic knowledge to formulate a plan, we ground our plan in an external modeling library that provides verified hints. Using E^*, the framework queries $\mathcal{L}$ to construct guide G: for each constraint c, it retrieves the corresponding atomic hint a_c; for each interaction, it retrieves the rule ψ only if all participating constraints are present. To assign an optimal configuration to unseen compositions, the framework determines the search strategy s^* by cascading through three matching criteria in order: (1) exact set equality, (2) maximal subset containment, and (3) weighted Jaccard similarity above a predefined threshold.

The planning generation prompt provides a solver-specific workflow with examples, instructing the GA to ground the plan in G and F, align variable names with the signature arguments, and override isolated constraint hints with interaction rules where overlaps occur, shown in the Appendix H.3. Injecting this compositional knowledge ensures H specifies not only how individual constraints are implemented, but also how their interactions are resolved. The GA takes (P, F, E^*, G) then generates a plan H that instantiates this workflow using the retrieved knowledge. Next, the EA audits H against the retrieved hints and interaction logic, verifying that every constraint in G is realized without omitting steps, additional parameters, or misapplying interaction rules. Upon failure, the RA repairs H using G as the sole ground truth, repeating the loop until the plan is approved. The validated plan H^* is then passed to code generation.

Code Generation and Execution. GA implements the verified plan H^* as executable solver code y using the target solver's API. Because H^* already specifies constraint modeling and composition logic, this stage realizes the plan rather than solving the problem from scratch. Nevertheless, code synthesis remains challenging as LLMs frequently misuse solver APIs by invoking non-existent methods or passing invalid parameters, faults that evade linters and surface only at execution (Shao et al., 2026). We mitigate this by applying a generation-evaluation-revision loop, where the EA assesses y for adherence to H^*, while the RA repairs plan deviations using H^* as the blueprint. Once accepted by the EA, the code is executed in a sandboxed subprocess. If a runtime traceback ϵ occurs, the error is passed back to the GA via a repair prompt. This execution-repair loop repeats until the code executes successfully or hits a maximum iteration threshold, yielding solver code y^*. We provide the prompts for these three sequential stages in Appendix H.

4 EXPERIMENTS

To verify the effectiveness and generalization of CoME, we conduct extensive experiments across multiple solvers, LLM families, and benchmarks. We further provide generalization to unseen variants, failure-case analysis, and efficiency analysis, together with ablations that isolate each component's contribution. Since we aim to let users specify their problem in natural language and have the LLM automatically generate executable solver code, evaluating CoME requires problems stated in natural language. However, most existing benchmarks supply only numerical instances, and those with descriptions cover only standard variants. We thus assemble a test set of 88 VRP variants spanning 13 constraint types by combining the variant sets of DRoC (Jiang et al., 2025b) and RouteFinder (Berto et al., 2025). To each variant, we add a natural-language description and a function signature specifying the function's name, parameters, and their types. Modeling hints are learned from only 17 of these variants. Full details of the variants and their description are provided in Appendix C.

We generate code for two widely used routing solvers, Google OR-Tools and PyVRP, chosen for their prevalence in industrial practice, active maintenance, and comprehensive APIs. To probe solver-agnostic transfer, we additionally target Gurobi and LKH, with results in Appendix G. Performance is measured by accuracy rate (AR) and success rate (SR), with average percentage gap reported as an additional metric; definitions are given in Appendix B. All experiments run on an Intel Xeon Gold 6348 (Ice Lake-SP) 16-core CPU. We evaluate generality across proprietary and open-source LLMs for code generation: Claude Opus 4.5, Gemini 3 Pro, Qwen 3.6-35B, and GPT-3.5 Turbo. All iterative loops in CoME are bounded to $t = 3$ iterations.

4.1 BASELINES

Since existing approaches target different sets of VRP variants, we compare each on the variants it actually solves rather than forcing all methods into a single comparison. Among methods that invoke external solvers, we compare against standard prompting, OptiTree (Liu et al., 2025), and DRoC (Jiang et al., 2025b). Among approaches, which produce a solver or learn a solution directly rather than solver-invoking code, we compare against AFL (Zhang et al., 2025b), an LLM-based automatic solver-design method, and RF-POMO (Berto et al., 2025), a neural solver. We further compare CoME against DRoC and AFL on standard VRP benchmarks. We extend this comparison to the LLM-with-evolutionary-computation methods, FunSearch (Romera-Paredes et al., 2024), EoH (Liu et al., 2024a), ReEvo (Ye et al., 2024), and EoH-S (Liu et al., 2026) on variants solvable by common heuristic algorithms in the Appendix D.3. Besides the comparison on VRPs, we evaluate

Table 1: Comparison of AR and SR between CoME and external solver integrated baselines on DRoC's 48-variant benchmark. OptiTree uses MILP modeling with Gurobi.

Method	Solver	Claude Opus 4.5		Qwen3.6-35b	
		AR	SR	AR	SR
Standard	OR-Tools	33.33	35.42	31.25	31.25
	PyVRP	55.89	55.89	35.29	35.29
OptiTree	Gurobi	18.75	18.75	16.67	16.67
DRoC	OR-Tools	56.25	58.33	41.67	45.83
CoME	OR-Tools	**83.30**	**98.95**	**81.25**	**95.83**
	PyVRP	**94.11**	**97.05**	**88.24**	**91.17**

CoME's generalization on the facility location problem and on constrained TSP and VRP, with full results in Appendix G.

4.2 Comparison against LLM Methods Using External Solvers

We first evaluate CoME against LLM-based approaches that invoke an external solver on DRoC's 48 VRP variants (Table 1). Under standard prompting, state-of-the-art LLMs solve only about one-third of the variants, underscoring the difficulty of these NP-hard problems. OptiTree, a general LLM-based OR method that formulates the problem as an MILP solved via Gurobi, exhibits even lower performance. This result highlights the limitations of generic mathematical modeling for complex VRP variants. DRoC improves through retrieval, but because it retrieves each constraint's implementation in isolation, it leaves the composition of constraints to the LLM. CoME substantially outperforms all baselines across both solvers and both LLMs, solving over 80% of variants. Moreover, the small performance gap between the two LLMs indicates that externalizing modeling knowledge reduces reliance on the model's intrinsic reasoning capabilities. We further illustrate CoME's advantage qualitatively through examples of generated solutions in Appendix F.

4.3 Comparison with End-to-End VRP Methods

We compare CoME against end-to-end methods, which produce the solver or learn a solution directly rather than invoking a mature external solver, namely the LLM-based method AFL (Zhang et al., 2025a) and the neural solver RF-POMO (Berto et al., 2025). We evaluate on the test data of RF-POMO (Berto et al., 2025), spanning 48 VRP variants with 1,000 instances at two problem sizes, $n \in \{50, 100\}$. Following the RF-POMO evaluation protocol, we report the average optimality gap. CoME generates PyVRP solver code, while AFL runs 10,000 iterations of solution improvement. To ensure a fair comparison, we match CoME's time budget to AFL's per-instance runtime and parallelize both across 16 CPU cores. Both CoME and AFL generate code with Claude Opus 4.5.

Table 2 reports the performance of CoME and the two baselines in terms of gap and runtime. CoME achieves a 100% success rate and the lowest gap across all variants at both sizes, remaining consistently near-optimal with matching runtime. AFL's solution quality is highly variable, ranging from under 1% to over 18% (on MDOCVRPTW). Such inconsistency undermines its reliability in practice, where operations depend on consistent, predictable quality rather than occasionally strong results. It achieves a 100% success rate only through a costly revision loop and, even at its best, does not match an expert-engineered solver. RF-POMO runs faster but exceeds 40% on several multi-depot variants, as it does not generalize to complex compositions. See Table 8 for additional results. The results follow the same trend, with CoME maintaining consistent solution quality across all variants.

4.4 Comparison Across Approaches on Standard Benchmarks

To position our method within the full landscape of competing approaches, we compare against DRoC and AFL on five standard routing benchmarks. We conduct experiments on TSPLIB (Reinelt, 1991), a real-world TSP benchmark containing 50 instances with sizes ranging from 50 to 1,000 customers; CVRPLIB (Uchoa et al., 2017), a real-world CVRP benchmark containing 43 instances

Table 2: Comparison results on RF-POMO 48 VRP variants ($n \in \{50, 100\}$). A dash (**-**) denotes a zero gap. More results are shown in Table 8.

	RF-POMO				AFL				CoME			
	n=50		n=100		n=50		n=100		n=50		n=100	
Variant	Gap	Time	Gap	Time	Gap	Time	Gap	Time	Gap	Time	Gap	Time
CVRP	2.41	0.03	2.50	0.12	1.45	1.30	1.83	3.51	**0.10**	1.04	-	4.17
CVRPB	3.17	0.02	4.47	0.10	3.30	2.68	3.62	7.45	-	2.08	**0.07**	7.29
CVRPBL	3.94	0.02	5.70	0.10	4.71	4.05	5.48	16.23	-	4.17	-	15.63
CVRPL	1.52	0.02	2.17	0.10	2.17	1.78	2.54	4.39	**0.28**	2.08	**0.32**	4.17
CVRPMB	8.68	0.04	10.64	0.11	2.64	5.38	2.51	23.18	-	5.21	-	20.80
CVRPMD	45.82	0.02	27.08	0.10	6.20	1.33	5.13	3.15	**0.12**	1.04	**0.17**	3.13
CVRPTW	2.09	0.02	3.58	0.12	1.18	3.38	1.57	11.51	**1.00**	10.40	**0.39**	10.40
CVRPTWB	1.67	0.02	2.96	0.12	1.15	5.38	1.70	21.27	**0.66**	5.21	-	20.80
CVRPTWBL	1.85	1.00	6.09	0.12	3.76	4.92	9.69	18.37	**1.74**	5.21	-	17.71
CVRPTWL	2.38	0.02	3.95	0.12	1.77	5.28	2.80	19.00	**0.31**	5.21	-	18.75
CVRPTWMDBL	33.38	0.06	41.00	1.00	2.46	6.02	2.72	21.40	**0.07**	6.25	-	20.80
CVRPTWMDL	44.81	0.03	40.22	0.20	1.33	6.67	1.50	24.98	**0.08**	6.25	-	20.80
OCVRP	2.90	0.02	4.66	0.10	2.15	2.22	2.77	6.57	-	2.08	**0.10**	6.25
OCVRPB	2.69	0.02	4.82	0.12	2.17	4.93	2.80	20.18	-	4.17	-	19.79
OCVRPBL	2.69	1.00	4.83	0.12	4.93	2.87	5.32	9.73	-	2.08	**0.09**	9.38
OCVRPL	2.95	0.02	4.66	0.10	2.15	2.33	2.98	7.25	-	2.08	**0.21**	7.29
OCVRPTW	1.38	0.02	2.72	0.12	0.67	4.58	1.06	17.85	**0.19**	4.17	-	17.71
OCVRPTWB	1.15	1.00	2.34	0.12	0.34	7.20	0.63	27.58	-	7.29	-	20.80
OCVRPTWBL	1.16	1.00	2.34	0.13	1.80	7.32	1.89	28.60	-	7.29	-	20.80
OCVRPTWL	1.38	0.02	2.73	0.12	0.67	6.03	1.00	22.98	-	6.25	-	20.80
OCVRPTWMD	32.02	0.02	34.90	0.12	10.51	6.50	18.62	23.77	-	6.25	-	20.80
OCVRPTWMDBL	29.12	0.05	30.06	1.00	2.07	8.28	4.31	32.40	-	8.33	-	20.80
OCVRPTWMDL	31.90	0.04	34.90	1.00	6.20	5.67	5.69	21.90	-	5.21	-	20.80
OCVRPTWMDMBL	37.16	0.03	44.21	0.90	0.51	15.50	2.93	64.63	-	10.40	-	20.80

CoME's per-instance time budget is 1–10 s at size 50 and 4–20 s at size 100. AFL and CoME both attain a 100% success rate. All times are in minutes.

Table 3: Average optimality gap (%) and runtime (minutes) on five standard benchmarks. Dashes indicate a method cannot model the corresponding variant.

	TSP		CVRP		CVRPL		CVRPB		CVRPMB	
Method	Gap%	Time	Gap%	Time	Gap%	Time	Gap%	Time	Gap%	Time
DRoC	3.25	13.53	6.18	7.17	5.87	3.33	-	-	-	-
AFL	2.25	13.53	3.85	7.20	3.34	18.33	2.89	1.75	3.10	12.53
CoME	**1.24**	13.53	**0.66**	7.17	**0.38**	3.33	**0.00**	1.65	**0.32**	3.50

Time limits per instance are matched to AFL's runtime: 10 s for CVRP and CVRPMB, 20 s for CVRPL, and 3 s for CVRPB; on TSP: DRoC and CoME match AFL's per-instance runtime.

with sizes ranging from 100 to 300 customers; Golden instances (Golden et al., 1998), a distance-constrained CVRP (CVRPL) benchmark containing 8 instances with 200 to 480 customers excluding instances with service time; the Toth-Vigo instances (Toth & Vigo, 1997), a VRP-with-backhauls (CVRPB) benchmark containing 33 instances with 20 to 100 customers; and the Salhi-Nagy instances (Salhi & Nagy, 1999), a VRP-with-mixed-pickup-and-delivery (CVRPMB) benchmark containing 21 instances with 50 to 199 customers. Our method generates PyVRP solvers, and we report the average percentage gap to the solutions of LKH (Helsgaun, 2017). AFL runs for 10,000 iterations, and to ensure a fair comparison, we set the time limit for DRoC and our method based on AFL's runtime and the number of nodes on each benchmark.

As reported in Table 3, CoME achieves the lowest optimality gap across all benchmarks within a time budget comparable to the baselines. These results demonstrate that CoME's superior solution quality extends to standard VRP benchmarks beyond our evaluation set. We also compare with methods that integrate evolutionary computation with LLMs in the Appendix D.3.

Table 4: Ablation results on DRoC's 48 VRP variants. AR and SR are reported (%) for each configuration on OR-Tools and PyVRP.

		OR-Tools		PyVRP	
Configuration	LLM	AR	SR	AR	SR
w/o Interaction Hints	Claude Opus 4.5	66.67	72.92	73.53	76.47
	Qwen3.6-35b	56.25	62.50	61.76	67.65
w/o Planning	Claude Opus 4.5	83.30	98.95	91.18	97.05
	Qwen3.6-35b	77.08	85.42	76.47	76.47
w/o Evaluation-Revision	Claude Opus 4.5	83.30	98.95	94.11	97.05
	Qwen3.6-35b	81.25	95.83	88.24	91.17
CoME	Claude Opus 4.5	**85.41**	**98.95**	**94.11**	**97.05**
	Qwen3.6-35b	**83.30**	**95.83**	**91.17**	**94.11**

4.5 Ablation Study

We conduct ablation experiments on the solver code generation pipeline of CoME using both OR-Tools and PyVRP with Claude Opus 4.5 and Qwen3.6-35b. To analyze the impact of interaction hints, the planning stage, and the evaluation-revision loop on CoME, experiments are performed by removing each of these components separately. As shown in the Table 4, removing interaction rules and search methods causes the largest performance drop, confirming composition knowledge as the critical factor. Performance degrades without library-guided planning, especially on weaker models, proving that the plan acts as a critical abstraction between translating high-level problem constraints into executable API code. In contrast, disabling the evaluation-revision loop causes only a minimal performance drop, demonstrating that it serves as a safeguard rather than a performance gain. We therefore disable it by default, enabling it only when correctness is critical. These results show that each component in our method is crucial for maintaining the accuracy and trustworthiness of the generated code. We further analyze failure cases on an open-source LLM in Appendix E.

4.6 Efficiency Analysis

Beyond success rate and solution quality, CoME is markedly more efficient than both baselines. It uses, on average, 2.3× fewer tokens and 2.6× fewer LLM calls than DRoC, and roughly two orders of magnitude fewer tokens than AFL, which often consumes around one million tokens per variant (Figures 4a and 4b). These gaps stem from how each method handles composite constraints. DRoC repeatedly runs a retrieve-and-debug loop to reconcile interactions, yet its retrieved hints never jointly demonstrate, and AFL iteratively refines a synthesized solver over many calls. In contrast, CoME composes stored modeling hints in a single pass, keeping cost largely decoupled from constraint complexity. This comparison is, moreover, conservative, as CoME's cost includes a one-time hint-learning overhead that the baselines do not incur. To further demonstrate that CoME is LLM-agnostic, we evaluate it across diverse LLM families on all 88 VRP variants in Appendix D.4.

5 Conclusion

In this work, we presented CoME, a framework that automatically learns modeling hints from verified solver code, capturing both isolated constraints and their interactions, and composes them into correct solver code via a multi-agent inference pipeline. Extensive experiments on 88 VRP variants, standard benchmarks, and other operations research problems across a range of solvers demonstrate the effectiveness, applicability, and trustworthiness of CoME, with performance superior to existing LLM-based methods. By grounding experience reuse in verified implementations rather than the model's own attempts, CoME captures the composition knowledge that composite problems demand, pointing toward reliable, solver-grade automation of constraint-rich optimization. In future work, we plan to integrate CoME with solver selection and metaheuristic generation to enhance its versatility and reliability across a wider range of optimization tasks. A promising direction is to formalize CoME's learned modeling experience as reusable agent skills, self-contained, composable knowledge packages that agents discover and load on demand, allowing the accumulated cross-constraint experience to be shared and extended across tasks and solvers.

REFERENCES

Ali AhmadiTeshnizi, Wenzhi Gao, Harald Brunborg, Shahab Talaei, Chris Lawless, 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 large language models. *arXiv preprint arXiv:2410.01783*, 2024.

Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. In *International Conference on Learning Representations (ICLR) Workshop Track*, 2017.

Federico Berto, Chuanbo Hua, Nayeli Gast Zepeda, André Hottung, Niels Wouda, Leon Lan, Junyoung Park, Kevin Tierney, and Jinkyoo Park. Routefinder: Towards foundation models for vehicle routing problems. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856.

D. D. Bochtis and C. G. Sørensen. The vehicle routing problem in field logistics: Part II. *Biosystems Engineering*, 105(2):180–188, 2010. doi: 10.1016/j.biosystemseng.2009.10.006.

Yitian Chen, Jingfan Xia, Siyu Shao, Dongdong Ge, and Yinyu Ye. Solver-informed RL: Grounding large language models for authentic optimization modeling. *arXiv preprint arXiv:2505.11792*, 2025.

Bruce L. Golden, Edward A. Wasil, John P. Kelly, and I-Ming Chao. The impact of metaheuristics on solving the vehicle routing problem: Algorithms, problem sets, and computational results. In Teodor Gabriel Crainic and Gilbert Laporte (eds.), *Fleet Management and Logistics*, pp. 33–56. Springer, Boston, 1998.

Keld Helsgaun. An extension of the lin-kernighan-helsgaun TSP solver for constrained traveling salesman and vehicle routing problems. Technical report, Roskilde University, 2017. URL `http://akira.ruc.dk/~keld/research/LKH-3/LKH-3_REPORT.pdf`.

André Hottung and Kevin Tierney. Neural large neighborhood search for routing problems. *Artificial Intelligence*, 313:103786, 2022.

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. *arXiv preprint arXiv:2405.17743*, 2024a.

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

Zhehui Huang, Guangyao Shi, and Gaurav S Sukhatme. From words to routes: Applying large language models to vehicle routing. *arXiv preprint arXiv:2403.10795*, 2024c.

Caigao Jiang, Xiang Shu, Hong Qian, Xingyu Lu, Jun Zhou, Aimin Zhou, and Yang Yu. Llmopt: Learning to define and solve general optimization problems from scratch. In *International Conference on Learning Representations (ICLR)*, 2025a. arXiv:2410.13213.

X. Jiang et al. Droc: Elevating large language models for complex vehicle routing via decomposed retrieval of constraints. In *International Conference on Learning Representations (ICLR)*, 2025b.

Xue Jiang, Yihong Dong, Lecheng Wang, Zishuo Ding, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 2024. doi: 10.1145/3672456.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? *arXiv preprint arXiv:2310.06770*, 2023. doi: 10.48550/arXiv.2310.06770.

Chaitanya K. Joshi, Quentin Cappart, Louis-Martin Rousseau, and Thomas Laurent. Learning TSP requires rethinking generalization. In *International Conference on Principles and Practice of Constraint Programming (CP)*, 2021.

Minsu Kim, Junyoung Park, and Jinkyoo Park. Sym-NCO: Leveraging symmetricity for neural combinatorial optimization. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pp. 1936–1949, 2022.

Grigorios D. Konstantakopoulos, Sotiris P. Gayialis, and Evripidis P. Kechagias. Vehicle routing problem and related algorithms for logistics distribution: a literature review and classification. *Operational Research*, 22(3):2033–2062, 2022. doi: 10.1007/s12351-020-00600-7.

Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations (ICLR)*, 2019.

Yeong-Dae Kwon, Jinho Choo, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min. POMO: Policy optimization with multiple optima for reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pp. 21188–21198, 2020.

Yunseo Lee, John Youngeun Song, Dongsun Kim, Jindae Kim, Mijung Kim, and Jaechang Nam. Hallucination by code generation llms: Taxonomy, benchmarks, mitigation, and challenges. *arXiv preprint arXiv:2504.20799*, 2025.

C. Lei et al. Planning-driven programming: A large language model programming workflow. *arXiv preprint arXiv:2411.14503*, 2024.

Han Li, Fei Liu, Zhi Zheng, Yu Zhang, and Zhenkun Wang. CaDA: Cross-problem routing solver with constraint-aware dual-attention. *arXiv preprint arXiv:2412.00346*, 2024.

K. Li et al. Ars: Automatic routing solver with large language models. In *International Conference on Learning Representations (ICLR)*, 2026.

F. Liu et al. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *International Conference on Machine Learning (ICML)*, 2024a.

F. Liu et al. Eoh-s: Evolution of heuristic set using llms for automated heuristic design. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 37090–37098, 2026.

Fei Liu, Xi Lin, Zhenkun Wang, Qingfu Zhang, Tong Xialiang, and Mingxuan Yuan. Multi-task learning for routing problem with cross-problem zero-shot generalization. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 1898–1908, 2024b.

Haoyang Liu, Jie Wang, Yuyang Cai, Xiongwei Han, Yufei Kuang, and Jianye Hao. Optitree: Hierarchical thoughts generation with tree search for LLM optimization modeling. *arXiv preprint arXiv:2510.22192*, 2025.

S. Liu et al. Large language models as evolutionary optimizers. In *IEEE Congress on Evolutionary Computation (CEC)*, 2024c.

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.

Fu Luo, Xi Lin, Fei Liu, Qingfu Zhang, and Zhenkun Wang. Neural combinatorial optimization with heavy decoder: Toward large scale generalization. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pp. 8845–8864, 2023.

Yining Ma, Zhiguang Cao, and Yeow Meng Chee. Learning to search feasible and infeasible regions of routing problems with flexible neural k-opt. *Advances in Neural Information Processing Systems (NeurIPS)*, 36:49555–49578, 2023.

Y. Ouyang et al. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2509.25140.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. *arXiv preprint arXiv:2307.16789*, 2023. doi: 10.48550/arXiv.2307.16789.

Gerhard Reinelt. TSPLIB—a traveling salesman problem library. *ORSA Journal on Computing*, 3 (4):376–384, 1991. doi: 10.1287/ijoc.3.4.376.

B. Romera-Paredes et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Saïd Salhi and Gábor Nagy. A cluster insertion heuristic for single and multiple depot vehicle routing problems with backhauling. *The Journal of the Operational Research Society*, 50(10): 1034–1042, 1999. doi: 10.1057/palgrave.jors.2600808.

Zhenyu Shao et al. Detecting and correcting hallucinations in llm-generated code via deterministic ast analysis. *arXiv preprint arXiv:2601.19106*, 2026.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. arXiv:2303.11366.

Y. Tan, X. Zhang, et al. DELTAMEM: Incremental experience memory for LLM agents via residual trees. *arXiv preprint arXiv:2606.03083*, 2026.

Xiangru Tang et al. Agent kb: Leveraging cross-domain experience for agentic problem solving. *arXiv preprint arXiv:2507.06229*, 2025.

Paolo Toth and Daniele Vigo. An exact algorithm for the vehicle routing problem with backhauls. *Transportation Science*, 31(4):372–385, 1997. doi: 10.1287/trsc.31.4.372.

Paolo Toth and Daniele Vigo (eds.). *Vehicle Routing: Problems, Methods, and Applications*. MOS-SIAM Series on Optimization. Society for Industrial and Applied Mathematics, 2 edition, 2014. doi: 10.1137/1.9781611973594.

Eduardo Uchoa, Diego Pecin, Artur Pessoa, Marcus Poggi, Thibaut Vidal, and Anand Subramanian. New benchmark instances for the capacitated vehicle routing problem. *European Journal of Operational Research*, 257(3):845–858, 2017. doi: 10.1016/j.ejor.2016.08.012.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 28, 2015.

Guanghua Wan, Yuanning Feng, Yao Wan, Zhaoyang Chu, Zhangqian Bi, Junxiao Han, Zhou Zhao, Hongyu Zhang, Pingpeng Yuan, Xuanhua Shi, and Hai Jin. Hallucinations in llm-based code summarization: Unveiling, detection, and mitigation. *Proceedings of the ACM on Software Engineering*, 2025. doi: 10.1145/3808139. To appear in FSE 2026.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024. doi: 10.48550/arXiv.2409.07429.

Ziyang Xiao, Dongxiang Zhang, Yangjun Wu, Lilin Xu, Yuan Jessica Wang, Xiongwei Han, Xiaojin Fu, Tao Zhong, Jia Zeng, Mingli Song, and Gang Chen. Chain-of-experts: When LLMs meet complex operations research problems. In *International Conference on Learning Representations (ICLR)*, 2024.

Zidi Xiong et al. How memory management impacts LLM agents: An empirical study of experience-following behavior. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2026. arXiv:2505.16067.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations (ICLR)*, 2024.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380. Association for Computational Linguistics, 2018. doi: 10.18653/v1/D18-1259.

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

Haoran Ye et al. Reevo: Large language models as hyper-heuristics with reflective evolution. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Haifei Zhang, Hongwei Ge, Jinlong Yang, and Yubing Tong. Review of vehicle routing problems: Models, classification and solving algorithms. *Archives of Computational Methods in Engineering*, 29(1):195–221, 2022.

N. Zhang et al. An agentic framework with llms for solving complex vehicle routing problems. In *International Conference on Learning Representations (ICLR)*, 2025a.

Ni Zhang, Zhiguang Cao, Jianan Zhou, Cong Zhang, and Yew-Soon Ong. An agentic framework with llms for solving complex vehicle routing problems. *arXiv preprint arXiv:2510.16701*, 2025b.

Xinzhi Zhang, Zeyi Chen, Humishka Zope, Hugo Barbalho, Konstantina Mellou, Marco Molinaro, Janardhan Kulkarni, Ishai Menache, and Sirui Li. Optimind: Teaching LLMs to think like optimization experts. *arXiv preprint arXiv:2509.22979*, 2025c.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 19632–19642, 2024. arXiv:2308.10144.

Jianan Zhou, Zhiguang Cao, Yaoxin Wu, Wen Song, Yining Ma, Jie Zhang, and Chi Xu. MVMoE: Multi-task vehicle routing solver with mixture-of-experts. *arXiv preprint arXiv:2405.01029*, 2024.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023. doi: 10.48550/arXiv.2307.13854.

Xinyu Zhu, Chen Xue, Di Chen, Zhen Zhou, Jiahua Tang, Dale Schuurmans, and Hanjun Dai. Large language models can learn rules. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.07064.

APPENDIX

A RELATED WORKS

A.1 LLMS FOR VRPS

Recent work applies LLMs to VRPs by leveraging their ability to encode and process complex optimization tasks (Huang et al., 2024c). Early work employs LLMs as direct optimization engines through prompt-driven black-box search. OPRO (Yang et al., 2024) maintains a trajectory of past solutions and their scores in a meta-prompt to steer the LLM toward better candidates, while LMEA (Liu et al., 2024c) casts the LLM as a genetic operator that performs crossover and mutation over a textual population of solutions. Because the search proceeds implicitly via token generation rather than through a principled optimization procedure, solution quality and feasibility degrade rapidly as the instance size grows. A more effective line couples LLMs with evolutionary computation, using the model as a variation operator to evolve the heuristics that construct solutions. FunSearch (Romera-Paredes et al., 2024) pioneered this by searching over program space with an automated evaluator. Another work EoH (Liu et al., 2024a) adopts a population-based framework with fixed templates for heuristic evolution. ReEvo (Ye et al., 2024) further combines evolutionary search with LLM reflections to provide verbal feedback and enhance search efficiency. EoH-S (Liu et al., 2026) evolves a cooperative portfolio of complementary heuristics. These methods remain tailored to specific problems, require domain knowledge to fix the heuristic template, and incur many LLM invocations to maintain a population. A third line has the LLM generate a solver itself: ARS (Li et al., 2026) injects LLM-generated constraint-scoring functions into a fixed destroy-and-repair backbone, while AFL (Zhang et al., 2025b) uses cooperative agents to synthesize a self-contained neighborhood-search solver. Because the solver is generated by the LLM, it carries no solution-quality guarantee, with gaps that widen on heavily constrained variants. Such methods are limited to a single fixed search algorithm, whereas mature solvers offer a variety of configurable algorithms. This motivates alternative methods generating code that invokes a mature solver rather than replacing it.

A distinct research direction keeps a mature solver and employs the LLM to translate a natural-language description into solver-ready code, retaining the solution quality that solvers provide. Most prior work addresses general mathematical programming rather than domain-specific VRP formulations. We classify them into two categories. The first finetunes open-source LLMs on curated synthetic data, through supervised fine-tuning on synthetic problem-model pairs, as in ORLM (Huang et al., 2024a) and OptMATH (Lu et al., 2025), and preference alignment, as in LLMOPT (Jiang et al., 2025a) and SIRL (Chen et al., 2025). Both depend on large amounts of data that are costly to build and often unreliable. The second steers a fixed LLM through structured prompting, via tree-based search over modeling stages, as in Autoformulation (Astorga et al., 2024) and OptiTree (Liu et al., 2025), and multi-expert collaboration with specialized formulator and programmer roles, as in OptiMUS (AhmadiTeshnizi et al., 2024) and Chain-of-Experts (Xiao et al., 2024). Both rely on the LLM's intrinsic knowledge, which limits their efficacy on problems beyond their training data. Closest to our work is DRoC (Jiang et al., 2025b), which alleviates the LLM's limited modeling knowledge through retrieval-augmented generation, improving solver-code correctness over standard prompting. However, it retrieves each constraint in isolation and does not supply their joint implementation, leaving composition to the LLM's own knowledge, precisely where it is least reliable. We instead alleviate both the intrinsic difficulty of individual constraints and the challenge of composing them through a knowledge-grounded approach that learns modeling hints from verified implementations.

A.2 NCO FOR VRPS

An extensive line of research automates end-to-end solutions for VRPs through deep (reinforcement) learning, known collectively as neural combinatorial optimization (NCO) (Kim et al., 2022; Kool et al., 2019; Luo et al., 2023). These methods employ Transformer-like architectures that encode instance features, such as customer coordinates and demands, and decode a solution tour directly, learning problem-solving strategies from data rather than relying on hand-designed heuristics. NCO generally follows two paradigms. Construction methods build a solution from scratch, either generating a feasible tour directly (Vinyals et al., 2015; Bello et al., 2017) or sampling one from a predicted

edge heatmap (Joshi et al., 2021); AM (Kool et al., 2019) first introduced attention to this setting, and POMO (Kwon et al., 2020) exploited multiple optima to improve solution quality, inspiring a series of refinements. Much recent effort targets generalization, scalability, complex constraints, and the handling of multiple VRP variants within a single framework: MTPOMO (Liu et al., 2024b) represents variants as combinations of shared attributes to solve unseen ones zero-shot, MVMoE (Zhou et al., 2024) uses a mixture-of-experts architecture to cover ten variants, CaDA (Li et al., 2024) introduces a constraint-aware dual-attention mechanism that reaches sixteen, and RF-POMO (Berto et al., 2025) provides a foundation model spanning many variants. Improvement methods instead refine an initial solution iteratively through learned local-search (Ma et al., 2023) or destroy-and-repair operators (Hottung & Tierney, 2022). Despite this progress, NCO models are typically trained separately for each variant at substantial cost, and adapting them to a new variant generally requires manual architectural modifications and retraining; even unified multi-variant frameworks cover a fixed, predetermined set of constraint combinations rather than arbitrary compositions.

A.3 Experience Learning and Knowledge Reuse

Experience learning studies how LLM agents extract and reuse transferable structured experience from historical interactions to improve downstream decision-making without parameter updates (Shinn et al., 2023; Zhao et al., 2024). Such experience may take the form of episodic traces, such as tool or action sequences, or semantic insights abstracted from successful and failed attempts (Zhao et al., 2024), and is primarily evaluated on long-horizon, feedback-rich tasks including multi-tool planning (Qin et al., 2023) and web interaction (Zhou et al., 2023), repository-level code generation and repair (Jimenez et al., 2023), and multi-step reasoning (Wang et al., 2024). Reflexion (Shinn et al., 2023) converts task feedback into verbal self-reflections stored as episodic memory and reused in later attempts, while ExpeL extracts insights from past trajectories and downweights contradictory ones through voting to improve single-shot performance and cross-task generalization. At a more symbolic end, HtT (Zhu et al., 2024) induces an explicit rule library maintained by quantitative filters such as coverage and confidence, and AgentKB (Tang et al., 2025) and ReasoningBank (Ouyang et al., 2026) organize an evolving reasoning memory to support self-improvement across tasks. Two limitations persist: experience is verified weakly, relying on self-assessment such as ExpeL's voting, which removes contradictions but not semantic errors that then propagate to later tasks (Xiong et al., 2026); and their experiences tend to be organized as a flat list of items (Tan et al., 2026; Ouyang et al., 2026; Tang et al., 2025).

B Evaluation Metrics

We use accuracy rate (AR) and success rate (SR) as primary metrics. AR measures the percentage of generated programs that find the mathematically optimal solution, while SR measures the percentage that return a feasible solution satisfying all constraints, even if suboptimal due to algorithmic or time-limit limitations. Formally, let V_t denote the total number of generated programs, V_a the number attaining optimality, and V_{succ} the number producing feasible solutions. Then

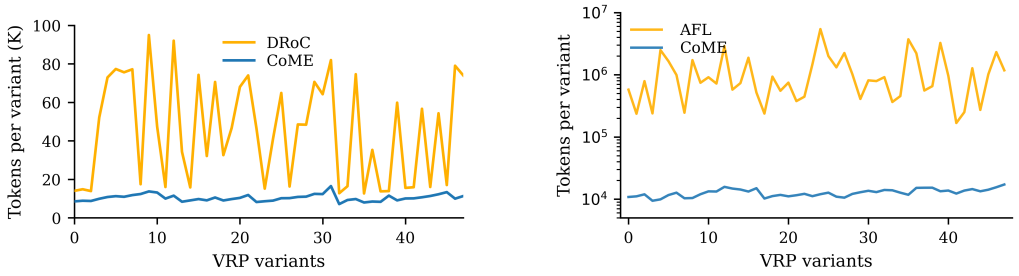
$$\text{AR} = \frac{V_a}{V_t} \times 100\%, \qquad \text{SR} = \frac{V_{\text{succ}}}{V_t} \times 100\%.$$

Thus SR captures programs that execute without error and yield feasible solutions, whereas AR additionally requires optimality; by construction, $\text{AR} \leq \text{SR}$. For methods that produce approximate rather than exact solutions, we report the optimality gap over feasible instances:

$$\text{gap} = \frac{1}{|V_{\text{feas}}|} \sum_{v \in V_{\text{feas}}} \frac{z_v - z_v^*}{z_v^*} \times 100\%,$$

where z_v is the objective value of the feasible solution and z_v^* is the optimum. This metric credits solutions by their proximity to the optimum rather than penalizing any deviation.

The choice of metric depends on the method being compared. Solver-invoking methods are evaluated using AR and SR. End-to-end neural methods and solver-generating methods rarely produce exactly optimal solutions, so they are evaluated by the average optimality gap. In principle, we follow the evaluation protocol of the methods we compare against.



(a) DRoC versus CoME.

(b) AFL versus CoME.

Figure 4: Token consumption analysis. (a) DRoC versus CoME; (b) AFL versus CoME. DRoC and AFL target different VRP variants.

Table 5: Thirteen VRP constraint types and their definitions.

Constraint	Definition
Capacity (C)	Each vehicle has a limited carrying capacity that the total demand on its route may not exceed.
Distance Limit (DL)	Each vehicle is restricted by a maximum total travel distance.
Duration Limit (L)	Each vehicle is restricted by a maximum total travel time.
Time Window (TW)	Each customer must be visited within a specified time interval.
Multi Depot (MD)	Vehicles may start and end their routes at different depots.
Open End (O)	Each vehicle starts at a depot but does not return to one after its last delivery.
Open Start-End (O_{SE})	Each vehicle has an unspecified start and end node.
Prize Collecting (PC)	Each customer carries a penalty if left unvisited, and routes balance these penalties against travel cost.
Pickup and Delivery (PD)	Each request pairs a pickup and a delivery that must be served by the same vehicle.
Service Time (S)	Each customer requires a fixed service duration that is added to the route time upon visit.
Resource Constraints (RC)	Each depot allows only a limited number of vehicles to load or unload simultaneously, potentially delaying departure or return.
Backhaul (B)	Each route serves delivery (linehaul) customers before pickup (backhaul) customers.
Mixed Backhaul (MB)	Each route may serve linehaul and backhaul customers in any order.

C VRP VARIANTS

We assemble the evaluation set of 88 VRP variants from the DRoC (Jiang et al., 2025b) and RouteFinder (Berto et al., 2025) papers. Since CoME takes a natural-language problem and generates solver code, evaluation requires problems stated in natural language. We adopt a model-data-separated format because large customer-node sets exceed LLM context limits, degrading performance as the input grows. Prior approaches that pass numerical data directly are restricted to small toy problems (AhmadiTeshnizi et al., 2024). Each problem comprises three components: a problem description (`description.txt`) specifying the scenario, objective, and constraints in natural language, without numerical data; a function signature (`solve.py`) giving the function name, typed parameters, and docstrings; and problem data (`data.py`) holding the numerical instance (Figure 5). Experts manually curate the problem descriptions.

We consider 13 canonical constraints: capacity, distance, duration limit, time window, service time, open route, open start-open end, multi-depot, backhaul, mixed backhaul, pickup-and-delivery, prize collecting, and resource constraints, defined in Table 5. Variants are formed by composing these

constraints, and the composition of each variant is given in Table 6. OR-Tools models all 88 VRP variants, whereas PyVRP covers 74, as it does not support pickup-and-delivery and resource constraints. We use 17 variants for hint learning (shaded in Table 6), each with a human-curated reference implementation. We illustrate the modeling hint distillation process in Figure 6, showing the hints extracted from a gold-standard program. Extracting hints from verified ground truth rather than the LLM's own attempts ensures the correctness of the modeling knowledge.

Problem Description

SwiftDeliver Logistics Problem involves planning efficient delivery routes for a fleet of trucks transporting large household appliances, such as fridges, to customers. The company operates a fleet of trucks, each with a fixed capacity, and customer demand specifies the number of fridges to be delivered. All deliveries start and end at the central depot and each customer must be visited exactly once. The objective is to determine the optimal delivery routes that minimize the total distance traveled while ensuring that each truck's capacity is not exceeded.

Problem Data

```
problem = [
        # fmt: off
      [0, 548, 776, 696, 582, 274, 502, 194, 308, 194, 536, 502, 388, 354, 468, 776, 662],
      [548, 0, 684, 308, 194, 502, 730, 354, 696, 742, 1084, 594, 480, 674, 1016, 868, 1210],
      [776, 684, 0, 992, 878, 502, 274, 810, 468, 742, 400, 1278, 1164, 1130, 788, 1552, 754],
      [696, 308, 992, 0, 114, 650, 878, 502, 844, 890, 1232, 514, 628, 822, 1164, 560, 1358],
      [582, 194, 878, 114, 0, 536, 764, 388, 730, 776, 1118, 400, 514, 708, 1050, 674, 1244],
      [274, 502, 502, 650, 536, 0, 228, 308, 194, 240, 582, 776, 662, 628, 514, 1050, 708],
      [502, 730, 274, 878, 764, 228, 0, 536, 194, 468, 354, 1004, 890, 856, 514, 1278, 480],
      [194, 354, 810, 502, 388, 308, 536, 0, 342, 388, 730, 468, 354, 320, 662, 742, 856],
      [308, 696, 468, 844, 730, 194, 194, 342, 0, 274, 388, 810, 696, 662, 320, 1084, 514],
      [194, 742, 742, 890, 776, 240, 468, 388, 274, 0, 342, 536, 422, 388, 274, 810, 468],
      [536, 1084, 400, 1232, 1118, 582, 354, 730, 388, 342, 0, 878, 764, 730, 388, 1152, 354],
      [502, 594, 1278, 514, 400, 776, 1004, 468, 810, 536, 878, 0, 114, 308, 650, 274, 844],
      [388, 480, 1164, 628, 514, 662, 890, 354, 696, 422, 764, 114, 0, 194, 536, 388, 730],
      [354, 674, 1130, 822, 708, 628, 856, 320, 662, 388, 730, 308, 194, 0, 342, 422, 536],
      [468, 1016, 788, 1164, 1050, 514, 514, 662, 320, 274, 388, 650, 536, 342, 0, 764, 194],
      [776, 868, 1552, 560, 674, 1050, 1278, 742, 1084, 810, 1152, 274, 388, 422, 764, 0, 798],
      [662, 1210, 754, 1358, 1244, 708, 480, 856, 514, 468, 354, 844, 730, 536, 194, 798, 0],
        # fmt: on
    ]
demands = [0, 1, 1, 2, 4, 2, 4, 8, 8, 1, 2, 1, 2, 4, 4, 8, 8]
vehicle_capacity = [15, 15, 15, 15]
num_veh = 4

depot = 0
```

Function Signature

```
def solve(distance_matrix: list, demands: list, num_vehicle: int, vehicle_capacity: list, depot: int):
        """
        Args:
                distance_matrix: contains the integer distance between customers\n
                demands: the list of integer customer demands
                num_vehicle: the number of the vehicle\n
                vehicle_capacities: the capacity of each vehicle\n
                depot: the index of the depot node\n\n
        Returns:\n
                obj: a number representing the objective value of the solution\n

        """
        # To be implemented
        obj = -1\n

return obj
```

Figure 5: A sample CVRP problem in our model-data-separated format.

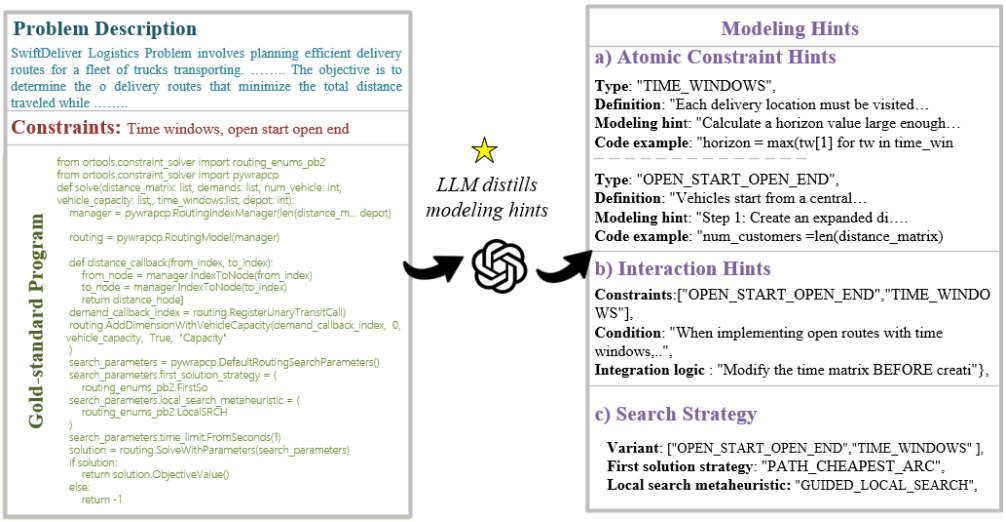


Figure 6: Modeling hint distillation example. From a gold-standard solver implementation, the LLM distills atomic constraint hints, cross-constraint interaction rules, and a search strategy.

Table 6: Constraint composition of the 88 VRP variants. Shaded rows indicate the variants used for hint learning. Variants with the `_SE` suffix use the open start-open end route type.

Variant	C	DL	L	TW	MD	O	O_{SE}	PC	PD	S	RC	B	MB
CVRP	✓												
CVRPB	✓											✓	
CVRPBL	✓		✓									✓	
CVRPL	✓	✓											
CVRPMB	✓												✓
CVRPMBL	✓		✓										✓
CVRPMD	✓				✓								
CVRPMDB	✓				✓							✓	
CVRPMDBL	✓		✓		✓							✓	
CVRPMDL	✓	✓			✓								
CVRPMDMB	✓				✓								✓
CVRPMDMBL	✓		✓		✓								✓
CVRPTW	✓			✓									
CVRPTWB	✓			✓								✓	
CVRPTWBL	✓		✓	✓								✓	
CVRPTWL	✓	✓		✓									
CVRPTWMB	✓			✓									✓
CVRPTWMBL	✓		✓	✓									✓
CVRPTWMD	✓			✓	✓								
CVRPTWMDB	✓			✓	✓							✓	
CVRPTWMDBL	✓		✓	✓	✓							✓	
CVRPTWMDL	✓	✓		✓	✓								
CVRPTWMDMB	✓			✓	✓								✓
CVRPTWMDMBL	✓		✓	✓	✓								✓
CVRPTWRC	✓			✓							✓		
CVRPTWRCL	✓	✓		✓							✓		
OCVRP	✓					✓							
OCVRPB	✓					✓						✓	
OCVRPBL	✓		✓			✓						✓	
OCVRPL	✓		✓			✓							
$OCVRPL_{SE}$	✓	✓					✓						

OCVRPMB	✓				✓						✓
OCVRPMBL	✓		✓		✓						✓
OCVRPMD	✓			✓	✓						
OCVRPMDB	✓			✓	✓					✓	
OCVRPMDBL	✓		✓	✓	✓					✓	
OCVRPMDL	✓		✓	✓	✓						
OCVRPMDMB	✓			✓	✓						✓
OCVRPMDMBL	✓		✓	✓	✓						✓
OCVRPTW	✓			✓	✓						
OCVRPTWB	✓			✓	✓					✓	
OCVRPTWBL	✓		✓	✓	✓					✓	
OCVRPTWL	✓		✓	✓	✓						
OCVRPTWMB	✓			✓	✓						✓
OCVRPTWMBL	✓		✓	✓	✓						✓
OCVRPTWMD	✓			✓	✓	✓					
OCVRPTWMDB	✓			✓	✓	✓				✓	
OCVRPTWMDBL	✓		✓	✓	✓	✓				✓	
OCVRPTWMDL	✓		✓	✓	✓	✓					
OCVRPTWMDMB	✓			✓	✓	✓					✓
OCVRPTWMDMBL	✓		✓	✓	✓	✓					✓
$\text{OCVRPTW}_{\text{SE}}$	✓			✓			✓				
OCVRP_{SE}	✓						✓				
OVRPL_{SE}		✓					✓				
$\text{OVRPTW}_{\text{SE}}$				✓			✓				
OVRP_{SE}							✓				
PCTSP								✓			
PCTSPTW				✓				✓			
PCVRP								✓			
PCVRPMD					✓			✓			
PCVRPTW				✓				✓			
PCVRPTWMD				✓	✓			✓			
PDP									✓		
PDPL		✓							✓		
PDPMD					✓				✓		
PDPSL		✓							✓	✓	
PDPTW				✓					✓		
PDPTWL		✓		✓					✓		
PDPTWMD				✓	✓				✓		
PDPTWMDL		✓		✓	✓				✓		
PDPTWS				✓					✓	✓	
PDPTWSL		✓		✓					✓	✓	
TSP											
TSPTW				✓							
TSPTWS				✓						✓	
VRP											
VRPL		✓									
VRPMD					✓						
VRPS										✓	
VRPSL		✓								✓	
VRPTW				✓							
VRPTWL		✓		✓							
VRPTWMD				✓	✓						
VRPTWMDL		✓		✓	✓						
VRPTWRC				✓						✓	
VRPTWRCL		✓		✓						✓	
VRPTWS				✓						✓	
VRPTWSL		✓		✓						✓	

Table 7: Success rate (%) of standard prompting (Claude Opus 4.5, OR-Tools) grouped by constraint count and split by whether the constraints interact. At matched constraint counts, interaction sharply reduces the success rate. Variants with four or more constraints always interact.

# Constraints	Without interaction	With interaction
1	80.0	–
2	90.0	36.4
3	66.7	22.7
4	–	15.8
5	–	11.1
6	–	0.0

D Additional Experimental Results

D.1 Pilot Experiment: Isolating the Effect of Constraint Interaction

To motivate this study, we conduct experiments to examine the causes of LLM solver code generation failures on complex VRP variants. Under standard prompting, Claude Opus 4.5 generates OR-Tools solver code for the 88 variants and solves only 37.5% of them.

A VRP variant is a composition of constraints. When two constraints can each be implemented correctly in isolation but their joint implementation is not the union of the two, we say they *interact*. Interaction arises because the correct implementation of one constraint affects how another is implemented. These cross-constraint interactions take one of the following forms: shared dimension, callback reuse, forced separation, index remapping, scope isolation, and global reordering.

To isolate the effect of interaction from that of constraint count, we group variants by constraint count and split each group into variants with interactions and those without. Table 7 reports the result. At a constraint count of one, no interaction is possible, and 80% of the variants are solved. At a count of two, the LLM solves 90.0% of variants without interaction but only 36.4% of those with interaction. At three, it solves 66.7% without interaction and 22.7% with. Variants with four or more constraints always contain interactions; solve rates fall to 15.8% at four, 11.1% at five, and 0% at six. At matched constraint counts, variants involving interactions exhibit substantially lower success rates. This demonstrates that modeling failures are driven more by cross-constraint interactions than by the constraint count, motivating our method to resolve these interaction failures.

D.2 Comparison with End-to-End VRP Methods (Extended)

The results of the additional 24 RF-POMO variants are presented in Table 8. Consistent with the main results, CoME attains the lowest optimality gap on most variants at both problem sizes, frequently reaching a zero gap. In contrast, AFL's performance varies widely across variants, while RF-POMO exhibits larger gaps, mainly on multi-depot variants.

D.3 Comparison with LLM+EC Methods

While LLMs have also been used to iteratively evolve heuristics for solving VRPs, we conduct an experiment to assess how this approach compares to CoME. We compare with FunSearch, EoH, ReEvo, and EoH-S on the TSP and CVRP instances, using the same settings as in Table 3. The results for these methods are taken from Liu et al. (2026). As shown in Table 9, the LLM+EC methods leave large optimality gaps (9% to 27%), whereas CoME reaches approximately 1%. These gaps arise because EC+LLM methods build solutions greedily without further refinement, whereas CoME invokes a mature solver directly. This confirms that external-solver integration achieves far higher solution quality than LLM-evolved heuristics.

Table 8: Extended comparison with end-to-end VRP methods on the remaining 24 RouteFinder variants.

	RF-POMO				AFL				CoME			
	n=50		n=100		n=50		n=100		n=50		n=100	
Variant	Gap	Time	Gap	Time	Gap	Time	Gap	Time	Gap	Time	Gap	Time
CVRPMBL	7.38	0.02	10.96	0.10	2.63	6.20	3.19	22.22	-	6.25	-	20.80
CVRPMDB	54.34	1.00	29.81	0.12	6.79	7.13	6.56	29.12	-	7.29	-	20.80
CVRPMDBL	53.93	1.00	34.50	0.14	4.49	3.49	4.88	14.51	-	3.13	-	14.58
CVRPMDMB	64.82	0.03	41.29	0.13	3.41	3.31	3.37	9.85	-	3.13	**0.09**	9.38
CVRPMDMBL	64.09	1.00	43.51	0.12	4.07	7.80	3.83	36.15	-	7.29	-	20.80
CVRPMDL	46.04	0.02	31.02	0.10	0.99	2.92	2.94	8.55	-	2.08	-	8.33
CVRPTWMB	7.56	1.00	11.84	0.15	1.00	7.75	1.41	31.22	**0.37**	7.29	-	20.80
CVRPTWMBL	7.30	1.00	11.76	0.13	1.23	5.62	1.78	21.57	**0.61**	5.21	-	20.80
CVRPTWMD	43.49	0.03	37.61	0.12	2.91	5.57	2.85	22.41	**0.17**	5.21	-	20.80
CVRPTWMDMB	43.66	0.04	48.68	0.12	3.73	5.53	4.04	21.56	**0.17**	5.21	-	20.80
CVRPTWMDMBL	44.17	1.00	53.96	0.14	1.16	10.30	1.50	40.69	-	10.40	-	20.80
CVRPTWMDB	32.70	1.00	38.30	0.11	1.84	7.59	2.00	24.78	-	7.29	-	20.80
OCVRPMB	14.08	0.03	20.31	0.13	1.30	5.70	1.78	25.46	-	5.21	-	20.80
OCVRPMBL	13.91	1.00	20.31	0.12	1.15	6.10	1.66	29.00	-	6.25	-	20.80
OCVRPMD	21.72	0.02	28.36	0.10	2.06	1.30	3.14	3.19	-	1.04	**0.25**	3.13
OCVRPMDB	26.96	1.00	28.20	0.13	1.22	4.76	1.73	19.62	-	4.17	-	19.79
OCVRPMDBL	26.78	1.00	25.32	0.13	6.26	6.96	8.98	29.64	-	6.25	-	20.80
OCVRPMDL	21.72	0.04	28.36	0.11	2.25	3.79	2.89	12.92	-	5.21	**0.13**	12.50
OCVRPMDMB	38.00	0.03	42.30	0.13	1.74	5.42	1.96	23.33	-	5.21	-	20.80
OCVRPMDMBL	38.80	0.04	42.30	0.14	0.39	10.59	1.04	49.37	-	10.40	-	20.80
OCVRPTWMB	6.11	0.03	9.79	0.14	0.48	7.77	0.71	32.31	-	7.29	-	20.80
OCVRPTWMBL	6.11	0.04	9.72	0.13	0.48	10.57	0.71	44.13	-	10.40	-	20.80
OCVRPTWMDMB	36.74	1.00	44.21	0.12	0.38	9.16	0.62	36.42	-	9.38	-	20.80
OCVRPTWMDB	28.79	1.00	30.06	0.14	0.58	6.49	1.18	25.82	-	6.25	-	20.80

Table 9: Average optimality gap (%) of LLM-with-evolutionary-computation methods on TSP and CVRP. Results for the evolutionary methods are taken from Liu et al. (2026).

Method	TSP	CVRP
FunSearch	15.20	27.50
EoH	17.30	27.00
ReEvo	16.50	27.00
EoH-S	9.30	19.60
CoME (ours)	**1.24**	**0.66**

D.4 Evaluation with Different LLMs

We next assess CoME's generalization across three proprietary and an open-source LLMs (Claude Opus 4.5, Gemini 3 Pro, GPT-3.5 Turbo, and Qwen3.6-35b) on 88 VRP variants. Specifically, we test whether performance gains stem from the library's supplied knowledge rather than from its parameters. Across all LLMs and both solvers, CoME outperforms standard prompting, with the largest gains on the weakest models as shown in Figure 7. These results confirm that the library actively supplies the necessary domain knowledge rather than merely amplifying existing parametric priors. Most notably, GPT-3.5 Turbo solves 0% of PyVRP variants under standard prompting, as PyVRP's 2023 release postdates the model's September 2021 training cutoff, yet achieves 91.89% accuracy with our method. These findings highlight that CoME's grounded experience-learning mechanism enables LLMs to generalize to unseen solvers using only their instruction-following capabilities.

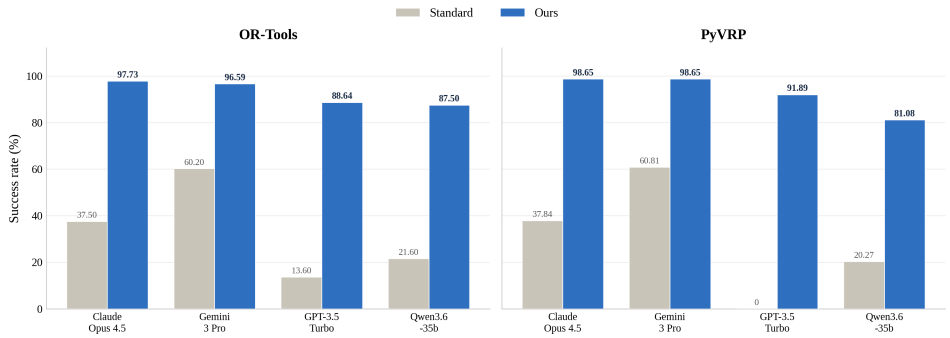


Figure 7: The performance of standard prompting and our method across various LLMs on 88 VRP variants. Our method improves performance across LLMs on both solvers.

	OR-Tools (71 unseen)		PyVRP (60 unseen)	
	Opus 4.5	Qwen 3.6-35B	Opus 4.5	Qwen 3.6-35B
Standard prompting	30.99	15.49	33.33	20.00
CoME	97.18	84.50	98.33	74.19

Table 10: Success rate on variants never seen during hint learning. CoME's gains persist on held-out variants, with no drop relative to the full benchmark.

D.5 Performance on Unseen VRP Variants

To further evaluate CoME's generalization, we conduct experiments on unseen variants (71 for OR-Tools, 60 for PyVRP) that were excluded from offline hint learning. As shown in Table 10. CoME achieves success rates comparable to its full 88-variant results shown in the Figure 7 and outperforms standard prompting. This confirms that CoME generalizes to unseen compositions rather than memorizing the variants seen during hint learning.

E Failure Cases Analysis

To understand our method's strengths and weaknesses, we analyze the variants it leaves unsolved using the open-source Qwen3.6-35b and attribute each failure to the stage at which it occurs: problem identification, library-guided planning, or code generation. We use Qwen because Claude leaves too few failures to analyze. Across both solver backends, 22 failures remain, of which 12 arise in problem identification, 5 in library-guided planning, and 5 in code generation, making problem identification the dominant failure mode (Figure 8b). Identification failures come mostly from confusing constraints with closely related demand semantics, such as backhaul, mixed backhaul, and pickup-and-delivery, or from omitting a multi-depot constraint. Planning and code-generation failures share a common pattern, either dropping a required sub-step or adding one the variant does not need, which points to the LLM's limited instruction-following ability rather than to any single stage. The distribution of failures across stages shifts systematically with constraint count (Figure 8a). At low-to-moderate counts, problem identification dominates. Modeling and implementation are straightforward, so nearly all failures come from misreading the problem. As composition grows, failures spread more evenly across the three stages, with identification failures declining while planning and code-generation failures rise as the number of interacting sub-steps grows.

F Generated Solutions

We present several examples of solutions produced by our method and the baselines in Figure 9. Figures 9a and 9b present two challenging variants, an open-route variant (OVRP) and a prize-collecting multi-depot variant (PCVRPMD), that standard prompting and DRoC (both using OR-Tools with Claude Opus) fail to solve. The baselines struggle with the open route in Figure 9a, which the

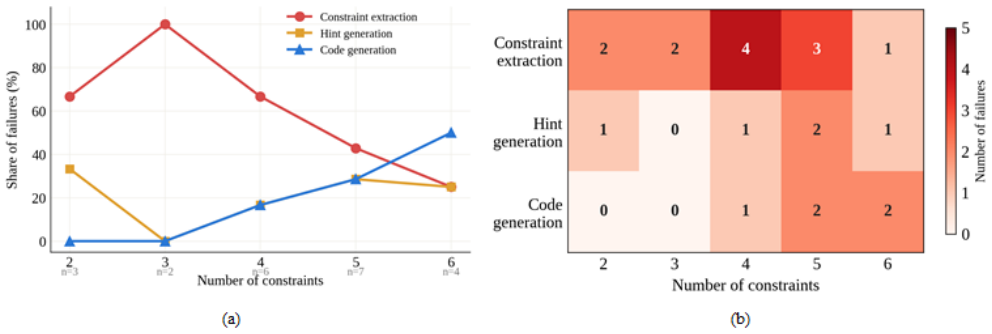


Figure 8: Failure distribution by pipeline stage and constraint count (Qwen-3.6). (a) Share of failures per stage; (b) failure counts per stage and constraint count.

external documentation they retrieve from never covers. On the composite variant in Figure 9b, retrieval covers each constraint separately but not their interaction, so the model must compose them unaided and produce redundant or infeasible code. In contrast, our method solves both variants and achieves the optimal solution. Figures 9c and 9d show that our approach also scales to complex variants: on OCVRPLTW, our PyVRP-generated solution achieves a lower objective than AFL, which requires an additional route and introduces an extra route crossing. Finally, Figures 9e–9g extend the comparison to CVRP across three paradigms: LLM with evolutionary computation (EoH-S), automatic routing-solver synthesis (AFL), and external-solver integration (ours), with the objective value shown above each subfigure. The ordering reflects the extent of optimization achieved by each paradigm. EoH-S builds routes greedily with no refinement, while AFL has the LLM write a metaheuristic solver that adds search but cannot match expert-designed functions. Our method invokes a mature solver directly and achieves the lowest objective value.

G Generalization Across Solvers and Problem Classes

Our method applies broadly to combinatorial optimization problems featuring interacting constraints and diverse objectives. This adaptability is demonstrated through evaluations on facility location using Gurobi, and constrained TSP and VRP variants using LKH-3. These experiments span both exact and heuristic solvers across programmatic and file-based interfaces.

G.1 Facility Location Problems

The Facility Location Problem (FLP) is a classic operations research and optimization task that selects sites for p facilities to minimize the total transportation cost from facilities to customers. Depending on the objective type, it gives rise to several well-known variants: *p-median* minimizes the total distance from each customer to its nearest facility; *p-center* minimizes the maximum customer-to-facility cost; *p-maxian* maximizes the total weighted travel cost; and *p-dispersion* maximizes the smallest distance between any pair of facilities. We study capacity constraints across all four variants and additionally consider the Maximum Coverage Location Problem to broaden the scope.

We evaluate on 62 facility-location problems drawn from three benchmark datasets, IndustryOR (Huang et al., 2024a), MAMO (Huang et al., 2024b), and OptMATH (Lu et al., 2025), curated by the OptiMIND (Zhang et al., 2025c) author to assess cross-modeling and cross-solver generalization capability. We compare against standard prompting and OptiTree (Liu et al., 2025), a state-of-the-art LLM-based optimization modeling method that organizes problems into a hierarchical modeling tree and applies tree search to decompose each problem into subproblems, whose high-level modeling thoughts are synthesized. All methods employ the Gurobi MILP solver. As shown in Figure 10, CoME substantially outperforms both OptiTree and standard prompting. CoME's automated hint learning and multi-agent inference pipeline extend beyond VRPs to broader OR problem classes.

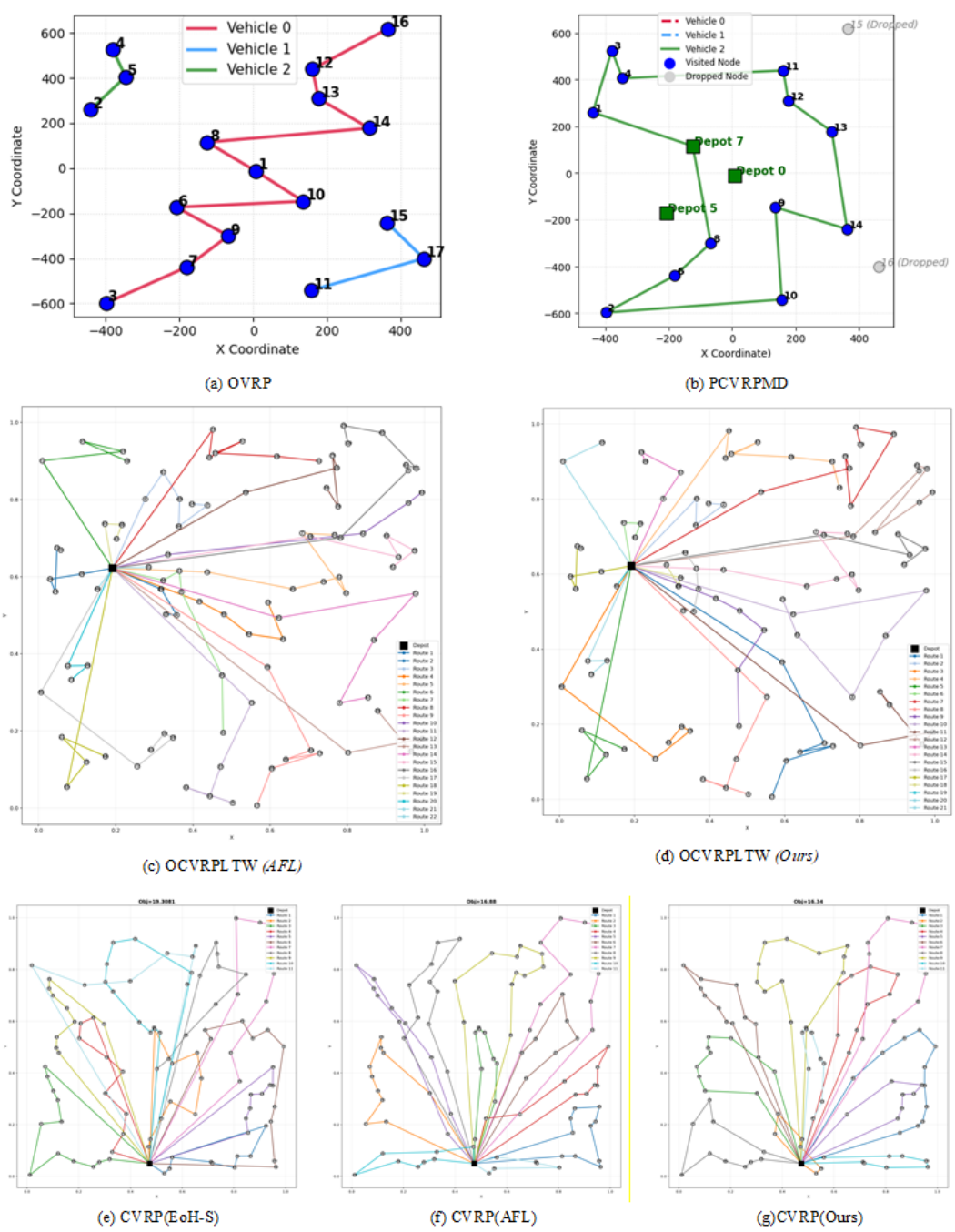


Figure 9: Qualitative comparison of VRP solutions generated by CoME and other baselines.

G.2 Constrained Traveling Salesman and Vehicle Routing Problems

We further evaluate CoME's generalization on constrained traveling salesman and vehicle routing problems solved using LKH-3 (Helsgaun, 2017), an open-source heuristic solver developed by Keld Helsgaun. Unlike the Python solver interfaces used elsewhere in the paper, LKH-3 relies entirely on file-based execution. Solving problems requires data preprocessing and generating custom parameters and data files programmatically, making it a markedly different code-generation target. We conduct experiments on 55 constrained TSP and VRP variants listed in Table 12. These variants are drawn from a taxonomy of 10 objectives and 39 constraints defined in Table 11. Instances sourced

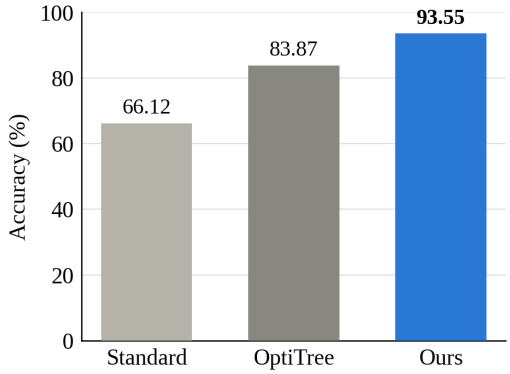


Figure 10: Performance on 62 facility-location problems from IndustryOR, MAMO, and OptMATH, solved with Gurobi. Our method outperforms OptiTree and standard prompting.

from the LKH-3 website and paired with human-curated problem descriptions. Claude Opus 4.5 is employed for code generation.

Table 11: The taxonomy of 10 objectives and 39 constraints from which the constrained TSP and VRP variants are composed.

Code	Objective / Constraint	Description
Objectives		
O1	Min total distance	Minimize the sum of travel costs of all edges used, across every route.
O2	Min path cost	Minimize the cost of a single open path from a given origin to a distinct given destination; no closing edge is counted.
O3	Min total latency	Minimize the sum over all nodes of the arrival time at that node (cumulative waiting time).
O4	Min expected length	Minimize the expected cost of a fixed a-priori route over the distribution of which nodes actually need service.
O5	Min max route	Minimize the cost of the single longest route.
O6	Min route range	Minimize the difference between the longest and the shortest route.
O7	Max min spacing	Maximize the smallest distance between any two consecutively visited nodes.
O8	Max collected prize	Maximize the total prize (score) collected from the nodes actually visited, while keeping the total travel cost within a stated budget; visiting every node is neither required nor possible.
O9	Max net profit	Maximize the total prize collected from the visited nodes minus the total travel cost; visiting each node is optional and worthwhile only when its prize outweighs the added travel.
O10	Min travel and skip penalty	Minimize the total travel cost plus a penalty charged for every node left unvisited, subject to collecting at least a required total prize; visiting a node is optional.
Constraints		
C1	Fixed fleet size	A specified number of routes must be built, all from a common depot, and every node belongs to exactly one.
C2	Route size bounds	The number of nodes on any route must lie between a stated lower and upper bound.
C3	Open routes	Routes start at the depot but do not return; the closing edge is neither traversed nor costed.
C4	Fixed distinct endpoints	The route is a path whose origin and destination are both specified and are different nodes.
C5	Multi depot return	Several depots exist; each route begins and ends at the same depot.
C6	Asymmetric costs	The cost from one node to another may differ from the cost of the reverse direction.
C7	Forbidden transitions	Specified ordered pairs of nodes may not be visited consecutively.
C8	Node class partition	Nodes are split into a mandatory/special class and an ordinary class, treated differently by the routing rules.

C9	Max ordinary between special	At most a stated number of ordinary nodes may fall between two consecutive special nodes.
C10	Max distance between special	At most a stated accumulated cost may fall between two consecutive special nodes.
C11	Capacity limit	Each vehicle has a limited load; each node's demand is served in full by one vehicle and no route exceeds capacity.
C12	Route length limit	The total length of any single route may not exceed a stated maximum.
C13	Service time in route length	Serving a node consumes a fixed time that counts toward the route-length limit or schedule.
C14	Risk exposure threshold	Risk accrues with the load carried and the distance carried; each route's total must stay below a threshold.
C15	Draft limit	Each node caps how much load the vehicle may still be carrying when it arrives there.
C16	Load feasibility pickup delivery	A single vehicle redistributes goods between nodes with NO paired requests – each node simply has a net surplus to collect or a net shortfall to fill – so the amount on board rises and falls and must stay within capacity at every point of the route.
C17	Multi commodity demands	Several distinct commodities, each with its own suppliers and consumers; units are not interchangeable between them.
C18	Paired pickup delivery	Each request pairs a pickup node with a specific delivery node, served by the same vehicle, pickup first.
C19	Mixed pickup delivery	A fleet serves customers who each need either a delivery brought from the depot or a pickup taken back to it, and a route may interleave the two in any order.
C20	Simultaneous pickup delivery	Each node requires a delivery and a pickup at the same visit; the load changes by the net of the two.
C21	Fifo loading	Items are unloaded in the order they were loaded (first in, first out).
C22	Lifo loading	Items are unloaded in reverse order of loading (last in, first out).
C23	Backhaul precedence	On any route, all delivery nodes must be served before any pickup node.
C24	Time windows	Each node must be reached within a stated earliest/latest interval.
C25	Exclusive color assignment	Every restricted node has exactly one eligible salesman – it belongs to a single color, and only that color's salesman may serve it; remaining nodes are shared and open to all.
C26	General eligibility matrix	A restricted node may have several eligible salesmen – each node carries its own arbitrary subset of qualified servers, of any size.
C27	Cluster contiguous service	All nodes of a cluster are served consecutively by one and the same vehicle.
C28	Cluster same vehicle	All nodes of a cluster are served by one and the same vehicle, but not necessarily consecutively.
C29	Group relaxation level	Nodes belong to groups whose ordering requirement is enforced at a tunable strictness level.
C30	Visit exactly k nodes	Visit exactly k of the n nodes and omit the rest; which nodes to include is part of the decision.
C31	Required nodes subset	Only a designated subset of nodes must be visited; the rest are optional pass-throughs.
C32	Stochastic node presence	Each node independently needs service with a common probability; the route is fixed before this is known.
C33	Fixed edges	Specified edges must appear in the solution.
C34	Node score	Each node carries a non-negative prize (score) that is collected only when the node is visited; these per-node prizes feed a prize-collecting, orienteering, or profit objective.
C35	Node penalty	Each node carries a penalty that is incurred only when the node is left unvisited (skipped); present whenever skipping a node has an explicit cost, distinct from any prize it would have yielded.
C36	Drone endurance	One or more drones launch from and return to the vehicle (truck) to serve nodes cooperatively, and each drone sortie is bounded by a fixed flight endurance – a range/time limit that a single launch-serve-return trip may not exceed.
C37	Precedence ordering	The visiting order is constrained by precedence rules – certain nodes must be visited (completed) before certain other nodes may be started, and every such before/after relation must hold in the final sequence.
C38	Landmark node sets	The nodes are organized into landmark groups – each landmark owns its own specific list of nodes, partitioning the customers into per-landmark sets that the truck/drones serve for that landmark.

C39	Tour cost budget	The tour's total travel cost may not exceed a fixed overall budget, and because that budget is too small to reach every node, only a chosen subset of nodes can be visited within it (as in orienteering / prize-collecting).

A modeling hint for LKH-3 specifies how a single objective or constraint is expressed in the solver's input files, whether through a file header, a named data section, a solver parameter, the problem type, or a transformation of the distance matrix. Composition is not mere concatenation, however: certain constraints interact, meaning their encodings cannot be applied independently. These interactions take several forms: section fusion, mathematical modification of a section, row-basis changes in granularity, combined or reinterpreted parameters, packed parameter strings, and pre-processing of the distance matrix. Hints learned from 27 variants cover the full benchmark, generating correct solvers for all 55. Since the LKH problem type is determined by LKH-3's implementation rather than derivable from a variant's objective and constraints, the model cannot infer it, so we precompute a type lookup table and fetch each variant's type from it at inference. Because the method learns an encoding per objective and per constraint rather than per variant, it can compose correct solvers for combinations it never encountered during learning, extending to combinatorial optimization problems that combine many constraints and objectives.

Table 12: The 55 constrained TSP and VRP variants and their objective-constraint compositions.

Variant	**Obj.**	**Constraints**	**Variant**	**Obj.**	**Constraints**
1-PDTSP	O1	C11, C16	OP	O8	C34, C39
ACVRP	O1	C6, C11, C1	OVRP	O1	C11, C1, C3
ADCVRP	O1	C6, C1, C12	PC-TSP	O10	C35, C34
ATSP	O1	C6	PCTSP	O1	C6, C1, C7, C26
BMTSP	O1	C1, C2	PDPTW	O1	C11, C1, C18, C24
BWTSP	O1	C10, C9, C8	PDTSP	O1	C18
CATSPP	O2	C6, C4, C7	PDTSPF	O1	C21, C18
CBnTSP	O5	C25, C1	PDTSPL	O1	C22, C18
CBTSP	O6	C25, C1	PTP	O9	C6, C34
CCCTSP	O3	C11, C1, C26	PTSP	O4	C32
CCVRP	O3	C11, C1	RCTVRP	O1	C1, C14
CluVRP	O1	C11, C27, C1	RCTVRPTW	O1	C1, C14, C13, C24
CTSP	O1	C25, C1	SoftCluVRP	O1	C11, C28, C1
CTSP-d	O1	C29	SOP	O1	C6, C37
CVRP	O1	C11, C1	STTSP	O1	C31
CVRPTW	O1	C11, C1, C13, C24	TRP	O3	C3
DCVRP	O1	C11, C1, C12, C13	TSP	O1	—
GCTSP	O1	C1, C26	TSP_mD	O5	C15, C36, C38, C13
k-TSP	O1	C30	TSPDL	O1	C15
m-PDTSP	O1	C11, C16, C17	TSPPD	O1	C11, C33, C16
m1-PDTSP	O1	C11, C17, C18	TSPTW	O1	C24
MDMTSP	O1	C6, C1, C5	VRPB	O1	C23, C11, C1
MLP	O3	—	VRPBTW	O1	C23, C11, C1, C13, C24
MSCTSP	O7	C25, C1	VRPMPD	O1	C11, C1, C19
MTRP	O3	C1, C3	VRPMPDTW	O1	C11, C1, C19, C13, C24
MTRPD	O3	C1, C3, C12	VRPSPD	O1	C11, C1, C20
mTSP	O1	C1	VRPSPDTW	O1	C11, C1, C13, C20, C24
O-k-TSP	O1	C1, C3			

H Prompt Templates

H.1 Modeling Hint Learning Prompt

Listing 1: Hint learning prompt.

```
You are a Lead {solver} Developer. Your task is to extract
    implementation rules from code, not just describe behavior. You
    are writing a guide for another programmer who needs to implement
    these constraints from scratch.
```

```
### Inputs:
Problem Description:
{problem_description}
Ground-Truth Solver Code:
{code}
Target Constraints:
{target_constraints}
### Output Format (JSON only):
Return a single JSON object with this structure:
1. For each target constraint, identify the following sections:
- Type: The standardized name of the constraint.
- Definition: A clear, one-sentence explanation of what this
    constraint enforces in a logistics context.
- Modeling Hint: A step-by-step technical explanation of how to
    implement this in {solver}. Mention specific classes and methods
    used in the ground truth code.
- Code Example: Extract the exact, functional snippet from the
    Ground-Truth code that implements this constraint.  Keep it
    concise.
2. Analyze Interactions (Crucial):
- Look for shared resources.
- Look for data modifications.
- Actionable Logic: The integration_logic must be an instruction.
    - Bad: "They share the time dimension."
    - Good: "Do NOT create a new dimension. Reuse the existing 'Time'
        dimension created for Time Windows."
If there is no technical interaction (e.g., they just both affect the
    route), leave the list empty.
3. Search Guidance: Identify which first_solution_strategy or
    local_search_metaheuristic was used in the code to handle this
    specific complexity.
### Output Example:
'''json
{{
    "constraint_implementations":[
        {{
        "type": "CAPACITY_DEMAND",
        "definition": "Vehicle load must not exceed capacity.",
        "modeling_hint": "Register a demand callback and add a
            capacity dimension.",
        "code_example": "..."
        }}
    ],
    "constraint_interactions":  [
        {{
            "constraints": ["OPEN_ROUTE", "CAPACITY_DEMAND"],
            "condition": "The specific scenario triggering the
                interaction (e.g., 'When both Time Windows and
                Duration Limits are present').",
            "integration_logic": "The specific coding action required
                (e.g., 'Reuse Dimension X', 'Modify Callback Y')."
        }}
    ],
    "search_method": {{
    "first_solution_strategy": "PATH_CHEAPEST_ARC",
    "local_search_metaheuristic": "GUIDED_LOCAL_SEARCH"
    }}
}}
```

H.2 Problem Identification Prompts

Listing 2: Problem identification prompt.

```
You are an expert in vehicle routing problems (VRP). Your task is to
    analyze a given problem description and function signature, then
    extract the constraints and objective of the VRP instance.
    Function signature specifies the function's name, its parameters,
    and parameter types. With each parameter in the function
    described by the docstring. First, determine if this is a
    **Single Vehicle** or **Multiple Vehicle** routing problem:
## Additional Constraints to Identify
From the problem description and function signature, identify which
    of the following advanced constraints apply. Return ONLY the
    constraints that are explicitly present or strongly implied.
{constraint _objective_definitions}
# Instructions
{instructions}
# Input Data
## Problem Description
{problem_description}
## Function Signature
{function_signature}
# Output Format
Return ONLY a JSON object in the following format:
```json
{{
  "problem_type": {{
    "vehicle_count": "single" | "multiple",
    "reasoning": "Explanation based on text and signature indicators"
  }},
"constraints": [
{{
    "type": "<constraint_keys>",
    "reasoning": "<quote from description or signature>"
}}
],
"Objective": {{
"type": "<objective_type>",
"reasoning": "<brief explanation>"
}}
}}
```

H.3 LIBRARY-GUIDED PLANNING PROMPTS

Listing 3: Planning Prompt

```
You are an expert in {target_solver} for vehicle routing problems.
    You will be given a problem description, a function signature
    with parameter descriptions, and a list of constraints that have
    been extracted from the problem, along with brief notes on how
    each constraint is typically implemented in {target_solver}. Your
    task is to generate a strategic hint - a step-by-step
    implementation plan - that guides a developer in solving this
    specific VRP using {target_solver}.
The hint should be practical and clear, focusing on the necessary
    steps.
## INPUTS
#1. Problem Description:
{PROBLEM_DESCRIPTION}

#2. Function Signature:
{FUNCTION_SIGNATURE}

#3. Extracted Problem Information:
{CONSTRAINTS_OBJECTIVES}
```
```

```
#4. Constraint Modeling Guides From Library:
{CONSTRAINT_HINTS}

#5. Solver-Specific Instructions:
{SOLVER_INSTRUCTIONS}

Wrap your final strategic hint inside <strategic_hint> and
    </strategic_hint> tags. Do not put anything else inside these
    tags except the step-by-step implementation plan. Do not try to
    write the whole code.
```

H.4 Code Generation and Execution Prompts

Listing 4: Code Generation Prompt

```
You are an expert operations research engineer specializing in
    {target_solver}. Your task is to implement the body of a Python
    function that solves a Vehicle Routing Problem (VRP) using
    {target_solver}' routing solver.
You will be given:
- A natural language description of the VRP (no numerical values
    included).
- A solve function signature specifying input types and return type.
- A strategic hint describing the exact {target_solver} modeling
    approach for the given problem. The strategic hint is not a
    suggestion. It is the full implementation specification.
## INPUT
#1. Problem Description:
{PROBLEM_DESCRIPTION}
#2. Function Signature:
{FUNCTION_SIGNATURE}
# 3. Strategic Hint:
{STRATEGIC_HINT}
CRITICAL RULES:
The strategic hint is the ONLY allowed implementation specification.
    You must implement exactly what it says, no more and no less.
- Use ONLY the function parameters for numerical data
- NEVER hardcode values from the problem description
- Return the objective value as specified in the function docstring
**DO NOT** "improve" the code with extra safety checks or additional
    constraints.
OUTPUT: Complete Python function implementation with imports.
Implement the strategic hint step by step.
Generate the Python {target_solver} code to solve this.
```

I Output Example

I.1 Library Hint Modeling Example

Listing 5: Library hint example for a multi-constraint VRP variant.

```
"CAPACITY_DEMAND": {
    "type": "CAPACITY_DEMAND",
    "definition": "Each vehicle's cumulative load from picked-up
        demands must never exceed its maximum carrying capacity
        throughout the route.",
    "modeling_hint": "1. Create a unary transit callback that returns
        the demand at each node using `RegisterUnaryTransitCallback`.
        2. Use `AddDimensionWithVehicleCapacity` to create a 'Capacity'
         dimension. 3. Set slack to 0 (no slack needed for capacity).
```

```
        4. Pass the list of vehicle capacities. 5. Set `
        start_cumul_to_zero=True` so vehicles start empty at the depot
        .",
    "code_example": "def demand_callback(from_index):\n    from_node =
        manager.IndexToNode(from_index)\n    return demands[from_node]\
        ndemand_callback_index = routing.RegisterUnaryTransitCallback(
        demand_callback)\nrouting.AddDimensionWithVehicleCapacity(\n
           demand_callback_index,\n    0,  # null capacity slack\n
        vehicle_capacities,  # vehicle maximum capacities\n    True,  #
         start cumul to zero\n    'Capacity')",
    "constraint_interactions": [
      {
        "constraints": [
          "CAPACITY_DEMAND",
          "BACKHAUL"
        ],
        "condition": "Both constraints require tracking load on the
            same vehicle, with delivery demands decreasing load ahead
            and backhaul demands increasing load after pickups.",
        "integration_logic": "Implement two separate dimensions for
            deliveries and pickups using distinct callbacks, ensuring
            each respects the vehicle capacity individually; modify the
             cost matrix to prevent backhaul-to-linehaul arcs,
            enforcing the route order constraint without merging
            dimensions."
      },
      {
        "constraints": [
          "OPEN_END",
          "CAPACITY_DEMAND"
        ],
        "condition": "Open routes affect how capacity is accumulated
            and route cost is computed.",
        "integration_logic": "No modification needed to demand
            callbacks or capacity dimension since demands are agnostic
            to route end; just ensure distance costs reflect open route
             returns by modifying the distance matrix to zero for depot
             columns."
      }
    ]
  },
  "TIME_WINDOWS": {
    "type": "TIME_WINDOWS",
    "definition": "Each delivery location must be visited within its
        specified earliest and latest arrival times, with vehicles
        allowed to wait if they arrive early.",
    "modeling_hint": "1. Calculate a horizon value large enough to
        accommodate all possible delivery times. 2. Use `AddDimension`
        with the transit callback to create a 'Time' dimension. 3. Set
        `slack_max` to horizon to allow waiting at nodes. 4. Set `
        fix_start_cumul_to_zero=False` to allow flexible start times.
        5. For each customer node (excluding depot), call `CumulVar(
        index).SetRange(earliest, latest)`. 6. For each vehicle's start
         index, constrain to depot's time window. 7. Leave vehicle end
        nodes unconstrained to allow flexible return times. 8. Use `
        AddVariableMinimizedByFinalizer` on start and end cumul vars to
         encourage the solver to minimize actual times.",
    "code_example": "horizon = max(tw[1] for tw in time_windows)\
        nhorizon = 2*horizon\nrouting.AddDimension(\n
        transit_callback_index,\n    horizon,  # slack_max: allows
        vehicles to wait\n    horizon,  # capacity: maximum total time
        duration\n    False,    # fix_start_cumul_to_zero\n    \"Time
        \"\n)\ntime_dimension = routing.GetDimensionOrDie(\"Time\")\
        nfor node_idx in range(len(time_windows)):\n    if node_idx ==
        depot:\n        continue\n    index = manager.NodeToIndex(
```

```
        node_idx)\n    time_dimension.CumulVar(index).SetRange(\n
               time_windows[node_idx][0],\n        time_windows[
        node_idx][1]\n    )\nfor vehicle_id in range(num_vehicle):\n
           start_index = routing.Start(vehicle_id)\n    time_dimension.
        CumulVar(start_index).SetRange(\n        time_windows[depot
        ][0],\n        time_windows[depot][1]\n    )\nfor i in range(
        num_vehicle):\n    routing.AddVariableMinimizedByFinalizer(
        time_dimension.CumulVar(routing.Start(i)))\n    routing.
        AddVariableMinimizedByFinalizer(time_dimension.CumulVar(routing
        .End(i)))",
    "constraint_interactions": [
      {
        "constraints": [
          "TIME_WINDOWS",
          "SERVICE_TIME"
        ],
        "condition": "When patients have both time windows and service
            durations that must be tracked together.",
        "integration_logic": "Do NOT create separate dimensions or
            callbacks. Embed service_time directly into the single
            time_callback by returning travel_time + service_time[
            from_node]. Register this combined callback ONCE and use it
             for BOTH SetArcCostEvaluatorOfAllVehicles() AND
            AddDimension(). The Time dimension then automatically
            tracks cumulative travel + service time, ensuring time
            window constraints account for service durations."
      },
      {
        "constraints": [
          "MULTI_DEPOT",
          "TIME_WINDOWS"
        ],
        "condition": "When vehicles have different starting depots and
            time windows apply to all locations including depots.",
        "integration_logic": "When applying time window constraints,
            SKIP start depot nodes in the general location loop (check
            `if location_idx in starts: continue`). Instead, apply
            depot time windows separately using `routing.Start(
            vehicle_id)` to get the correct internal index for each
            vehicle's starting position."
      },
    ]
  },
  "SERVICE_TIME": {
    "type": "SERVICE_TIME",
    "definition": "Each patient visit requires a fixed duration of care
         that must be spent at the location before the nurse can travel
         to the next patient, effectively adding to the time consumed
        on the route.",
    "modeling_hint": "1. Do NOT create a separate dimension for service
         time. 2. Incorporate service time directly into the time
        callback by adding service_time[from_node] to the travel time.
        3. The callback returns: time_matrix[from_node][to_node] +
        service_time[from_node]. 4. This ensures the cumulative time
        tracked by the Time dimension includes both travel and service.
         5. Set service_time[depot] = 0 since no service occurs at the
        depot.",
    "code_example": "def time_callback(from_index, to_index):\n
        \"\"\"Returns the travel time between the two nodes.\"\"\"\n
           # Convert from routing variable Index to time matrix
        NodeIndex.\n    from_node = manager.IndexToNode(from_index)\n
           to_node = manager.IndexToNode(to_index)\n\n    return
        time_matrix[from_node][to_node] + service_time[from_node]\n\
        ntransit_callback_index = routing.RegisterTransitCallback(
```

```
      time_callback)\nrouting.SetArcCostEvaluatorOfAllVehicles(
      transit_callback_index)",
  "constraint_interactions": [
    {
      "constraints": [
        "TIME_WINDOWS",
        "SERVICE_TIME"
      ],
      "condition": "When patients have both time windows and service
          durations that must be tracked together.",
      "integration_logic": "Do NOT create separate dimensions or
          callbacks. Embed service_time directly into the single
          time_callback by returning travel_time + service_time[
          from_node]. Register this combined callback ONCE and use it
           for BOTH SetArcCostEvaluatorOfAllVehicles() AND
          AddDimension(). The Time dimension then automatically
          tracks cumulative travel + service time, ensuring time
          window constraints account for service durations."
    }
  ]
},
"MULTI_DEPOT": {
  "type": "MULTI_DEPOT",
  "definition": "Each vehicle must start from its assigned starting
      depot and end at its assigned ending depot, where start and end
       depots may differ per vehicle.",
  "modeling_hint": "Pass the `starts` and `ends` lists directly to
      the RoutingIndexManager constructor instead of a single depot
      index. The manager signature is `RoutingIndexManager(
      num_locations, num_vehicles, starts, ends)` where starts and
      ends are lists of node indices with length equal to
      num_vehicles.",
  "code_example": "manager = pywrapcp.RoutingIndexManager(len(
      time_matrix), num_vehicle, starts, ends)",
  "constraint_interactions": [
    {
      "constraints": [
        "MULTI_DEPOT",
        "TIME_WINDOWS"
      ],
      "condition": "When vehicles have different starting depots and
          time windows apply to all locations including depots.",
      "integration_logic": "When applying time window constraints,
          SKIP start depot nodes in the general location loop (check
          `if location_idx in starts: continue`). Instead, apply
          depot time windows separately using `routing.Start(
          vehicle_id)` to get the correct internal index for each
          vehicle's starting position."
    },
    {
      "constraints": [
        "OPEN_END",
        "MULTI_DEPOT"
      ],
      "condition": "When vehicles start at multiple depots and routes
           end openly without returning to depots.",
      "integration_logic": "Set vehicle ends to the same depot node
          indices as their start nodes in RoutingIndexManager, but
          modify the distance matrix to zero out costs returning to
          depot nodes, allowing open-ended routes while satisfying
          start/end node requirements."
    },
  ]
},
"BACKHAUL": {
```

```
    "type": "BACKHAUL",
    "definition": "Requires each vehicle route to serve all delivery (
        linehaul) customers before any pickup (backhaul) customers,
        preventing interleaving of deliveries and pickups within the
        same route.",
    "modeling_hint": "1. Identify linehaul (delivery) and backhaul (
        pickup) customer indices.\n2. Modify the distance matrix by
        assigning prohibitively high costs (e.g. 1e9) to arcs from
        backhaul nodes to linehaul nodes, effectively forbidding those
        transitions.\n3. Create a separate dimension with
        AddDimensionWithVehicleCapacity for backhaul demands, allowing
        capacity tracking of collected pickups.\nThis cost modification
         enforces that in routing optimization, backhauls are scheduled
         only after all linehauls.",
    "code_example": "\n    linehaul_idx = [i for i in range(n) if
        demands[i] > 0]\n    backhaul_idx = [i for i in range(n) if
        backhauls[i] > 0]\n    for b in backhaul_idx:\n        for l in
         linehaul_idx:\n            distance_matrix[b][l] = 1
        _000_000_000\n\n    def backhaul_callback(from_index):\n
               from_node = manager.IndexToNode(from_index)\n
        return backhauls[from_node]\n    backhaul_callback_index =
        routing.RegisterUnaryTransitCallback(backhaul_callback)\n
        routing.AddDimensionWithVehicleCapacity(\n
        backhaul_callback_index, \n        0, \n        [capacity] *
        num_vehicles, \n        True, \n        \"Backhaul\",\n    )\n
        ",
    "constraint_interactions": [
      {
        "constraints": [
          "CAPACITY_DEMAND",
          "BACKHAUL"
        ],
        "condition": "Both constraints require tracking load on the
            same vehicle, with delivery demands decreasing load ahead
            and backhaul demands increasing load after pickups.",
        "integration_logic": "Implement two separate dimensions for
            deliveries and pickups using distinct callbacks, ensuring
            each respects the vehicle capacity individually; modify the
             cost matrix to prevent backhaul-to-linehaul arcs,
            enforcing the route order constraint without merging
            dimensions."
      }
    ]
  },
  "OPEN_END": {
    "type": "OPEN_END",
    "definition": "Vehicles start routes at the depot but are not
        required to return there, allowing routes to end at the last
        customer to reduce return travel costs.",
    "modeling_hint": "1. Set up start and end nodes so vehicles start
        at the depot but end at any node.\n2. Modify the distance
        matrix to set travel costs to the depot to zero, effectively
        removing return arcs.\n3. Pass start and end node lists to the
        RoutingIndexManager with starts all at the depot and ends also
        set to depot but these will be ignored due to zero return costs
        .\n4. This makes the solver treat the problem as an open route
        problem without additional explicit end constraints.",
    "code_example": "distance_matrix[:, starting_depot] = 0\nstarts = [
        starting_depot] * num_vehicles\nends = [starting_depot] *
        num_vehicles\nmanager = pywrapcp.RoutingIndexManager(n,
        num_vehicles, starts, ends)",
    "constraint_interactions": [
      {
        "constraints": [
          "OPEN_END",
```

```
          "MULTI_DEPOT"
        ],
        "condition": "When vehicles start at multiple depots and routes
            end openly without returning to depots.",
        "integration_logic": "Set vehicle ends to the same depot node
            indices as their start nodes in RoutingIndexManager, but
            modify the distance matrix to zero out costs returning to
            depot nodes, allowing open-ended routes while satisfying
            start/end node requirements."
      },
      {
        "constraints": [
          "OPEN_END",
          "CAPACITY_DEMAND"
        ],
        "condition": "Open routes affect how capacity is accumulated
            and route cost is computed.",
        "integration_logic": "No modification needed to demand
            callbacks or capacity dimension since demands are agnostic
            to route end; just ensure distance costs reflect open route
            returns by modifying the distance matrix to zero for depot
            columns."
      }
    ]
  },
```

I.2 OCVRPTWMDB End-to-End Example

Listing 6: Problem Description.

```
A medical-supply distributor operates multiple depots that deliver
    sterile equipment to clinics and hospitals, while also collecting
    used or expired equipment from other facilities for sterilization
    or disposal. On each route, all deliveries of sterile supplies
    must be completed before any used equipment is collected.
    Customers have strict time windows during which they must be
    served. Each delivery or collection stop requires a fixed service
    time, which must be completed within the customer's time window.
    Each truck has a limited cargo capacity. Drivers start at one of
    the depots and end their shifts at the last collection site,
    without returning to a depot. Every customer must be visited
    exactly once. The distributor wants to assign customers to trucks
    and plan routes that minimize total travel distance while
    respecting time windows, capacity, depot assignments, the
    delivery-before-pickup sequence, and the open-end policy.
```

Listing 7: Function Signature.

```
def solve(distance_matrix: list, demands: list, capacity: int,
    num_vehicles: int, starting_depots: list, backhauls: list,
    time_windows: list, service_times: list):
        """
            Args:
        distance_matrix: contains the integer distance between
            locations (depots + customers).
        demands: the list of integer customer demands.
        capacity: the maximum cargo capacity of each vehicle.
        num_vehicles: the number of vehicles available (length of
            starting_depots).
```

```
        starting_depots: the per-vehicle list of depot node indices
            where each vehicle starts its route (length = number of
            vehicles).
        backhauls: the list of integer customer pickup (backhaul)
            demands.
        time_windows: the list of customer time windows [start, end]
            for service.
        service_times: the list of service times (durations) at each
            customer location.

            Returns:
                obj: a number representing the objective value of the
                    solution.
        """
        obj = -1
        # To be implemented
        return obj
```

Listing 8: Problem Identification Output.


```
{
  "problem_type": {
    "vehicle_count": "multiple",
    "reasoning": "The function signature includes 'num_vehicles: int'
        and 'starting_depots: list' with length equal to number of
        vehicles, indicating a fleet of multiple vehicles that must be
        assigned to customers."
  },
  "constraints": [
    {
      "type": "CAPACITY_DEMAND",
      "reasoning": "Function signature includes 'demands: list' and '
          capacity: int'. Description states 'Each truck has a limited
          cargo capacity.'"
    },
    {
      "type": "MULTI_DEPOT",
      "reasoning": "Function signature includes 'starting_depots: list'
           (per-vehicle depot assignments). Description states '
          operates multiple depots' and 'Drivers start at one of the
          depots.'"
    },
    {
      "type": "TIME_WINDOWS",
      "reasoning": "Function signature includes 'time_windows: list'.
          Description explicitly states 'Customers have strict time
          windows during which they must be served' and 'must be
          completed within the customer's time window.'"
    },
    {
      "type": "SERVICE_TIME",
      "reasoning": "The description states 'Each delivery or collection
           stop requires a fixed service time' at the customer location
          . The function signature includes 'service_times: object'
          with docstring 'service times (durations) at each customer
          location', confirming time spent serving customers on-site."
    },
    {
      "type": "BACKHAUL",
      "reasoning": "Function signature includes 'backhauls: list' for
          pickup demands. Description explicitly states 'all deliveries
           of sterile supplies must be completed before any used
```

```
          equipment is collected' and 'the delivery-before-pickup
          sequence,' which defines the backhaul ordering rule."
    },
    {
      "type": "OPEN_END",
      "reasoning": "Description states 'Drivers start at one of the
          depots and end their shifts at the last collection site,
          without returning to a depot' and mentions 'the open-end
          policy.' Vehicles have anchored starts at depots but free
          ends."
    }
  ],
  "Objective": {
    "type": "Minimize total travel distance",
    "reasoning": "Description explicitly states the goal is to '
        minimize total travel distance while respecting time windows,
        capacity, depot assignments, the delivery-before-pickup
        sequence, and the open-end policy.'"
  }
}
```

Listing 9: Planning Output.

```
## Step 1: Initialize Routing Model

Since this is a multi-depot problem with open-end routes, initialize
    the routing index manager with separate start and end depot lists.
    However, because routes are open-ended (vehicles don't return to
    depot), set both starts and ends to the depot indices from `
    starting_depots`, but modify the distance matrix to zero out return
     costs to depot nodes.

```python
# Modify distance matrix to implement open-end routes
n = len(distance_matrix)
for depot_idx in set(starting_depots):
    for i in range(n):
        distance_matrix[i][depot_idx] = 0

**Backhaul Constraint (enforce delivery-before-pickup sequence):**
```python
# Identify linehaul and backhaul customer indices
linehaul_nodes = [i for i in range(len(demands)) if demands[i] > 0]
backhaul_nodes = [i for i in range(len(backhauls)) if backhauls[i] > 0]

# Prohibit arcs from backhaul to linehaul nodes
for b in backhaul_nodes:
    for l in linehaul_nodes:
        distance_matrix[b][l] = 1_000_000_000
```

# Initialize with multi-depot configuration
manager = pywrapcp.RoutingIndexManager(
    len(distance_matrix),
    num_vehicles,
    starting_depots, # each vehicle's starting depot
    starting_depots   # set ends to same depots (will be ignored due to
         zero return costs)
)

routing = pywrapcp.RoutingModel(manager)
```
```

````
## Step 2: Create Distance Callback and Set Arc Costs

Create the main transit callback for distance and set it as the arc
    cost evaluator:

```python
def distance_callback(from_index, to_index):
    from_node = manager.IndexToNode(from_index)
    to_node = manager.IndexToNode(to_index)
    return distance_matrix[from_node][to_node]

distance_callback_index = routing.RegisterTransitCallback(
    distance_callback)
routing.SetArcCostEvaluatorOfAllVehicles(distance_callback_index)
```

## Step 3: Create Additional Callbacks

Create separate callbacks for time (including service time), delivery
    demands, and backhaul demands:

**Time Callback (combines travel time with service time):**
```python
def time_callback(from_index, to_index):
    from_node = manager.IndexToNode(from_index)
    to_node = manager.IndexToNode(to_index)
    # Travel time based on distance + service time at from_node
    return distance_matrix[from_node][to_node] + service_times[
        from_node]

time_callback_index = routing.RegisterTransitCallback(time_callback)
```

**Delivery Demand Callback (unary):**
```python
def demand_callback(from_index):
    from_node = manager.IndexToNode(from_index)
    return demands[from_node]

demand_callback_index = routing.RegisterUnaryTransitCallback(
    demand_callback)
```

**Backhaul Demand Callback (unary):**
```python
def backhaul_callback(from_index):
    from_node = manager.IndexToNode(from_index)
    return backhauls[from_node]

backhaul_callback_index = routing.RegisterUnaryTransitCallback(
    backhaul_callback)
```

## Step 4: Add Dimensions

**Time Dimension** (for time windows and service time tracking):
```python
horizon = max(tw[1] for tw in time_windows) * 2
routing.AddDimension(
    time_callback_index,
    horizon,  # allow waiting
    horizon,  # maximum time per route
    False,    # don't fix start to zero
    'Time'
````
````

````
)
time_dimension = routing.GetDimensionOrDie('Time')
```

**Capacity Dimension** (for delivery demands):
```python
routing.AddDimensionWithVehicleCapacity(
    demand_callback_index,
    0,  # no slack
    [capacity] * num_vehicles,
    True,  # start cumul to zero
    'Capacity'
)
```

**Backhaul Dimension** (for pickup demands):
```python
routing.AddDimensionWithVehicleCapacity(
    backhaul_callback_index,
    0,  # no slack
    [capacity] * num_vehicles,
    True,  # start cumul to zero
    'Backhaul'
)
```

## Step 5: Enforce Constraints

**Time Window Constraints:**
```python
# Apply time windows to all customer nodes (skip depot nodes)
depot_nodes = set(starting_depots)
for location_idx in range(len(time_windows)):
    if location_idx in depot_nodes:
        continue
    index = manager.NodeToIndex(location_idx)
    time_dimension.CumulVar(index).SetRange(
        time_windows[location_idx][0],
        time_windows[location_idx][1]
    )

# Apply depot time windows to vehicle start nodes
for vehicle_id in range(num_vehicles):
    start_index = routing.Start(vehicle_id)
    depot_idx = starting_depots[vehicle_id]
    time_dimension.CumulVar(start_index).SetRange(
        time_windows[depot_idx][0],
        time_windows[depot_idx][1]
    )

# Minimize start and end times
for vehicle_id in range(num_vehicles):
    routing.AddVariableMinimizedByFinalizer(
        time_dimension.CumulVar(routing.Start(vehicle_id))
    )
    routing.AddVariableMinimizedByFinalizer(
        time_dimension.CumulVar(routing.End(vehicle_id))
    )
```
## Step 6: Configure Search Parameters

```python
search_parameters = pywrapcp.DefaultRoutingSearchParameters()
search_parameters.first_solution_strategy = (
    routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC
````

```
)
search_parameters.local_search_metaheuristic = (
    routing_enums_pb2.LocalSearchMetaheuristic.GUIDED_LOCAL_SEARCH
)
search_parameters.time_limit.seconds = 30
```

Step 7: Solve and Return Objective Value

```python
solution = routing.SolveWithParameters(search_parameters)
if solution:
    return solution.ObjectiveValue()
return -1
```

Listing 10: Generated OR-Tools Solver Code.

```
from ortools.constraint_solver import pywrapcp
from ortools.constraint_solver import routing_enums_pb2

def solve(distance_matrix: list, demands: list, capacity: int,
    num_vehicles: int, starting_depots: list, backhauls: list,
    time_windows: list, service_times: list):
    """
    Args:
        distance_matrix: contains the integer distance between
            locations (depots + customers).
        demands: the list of integer customer demands.
        capacity: the maximum cargo capacity of each vehicle.
        num_vehicles: the number of vehicles available (length of
            starting_depots).
        starting_depots: the per-vehicle list of depot node indices
            where each vehicle starts its route (length = number of
            vehicles).
        backhauls: the list of integer customer pickup (backhaul)
            demands.
        time_windows: the list of customer time windows [start, end]
            for service.
        service_times: the list of service times (durations) at each
            customer location.

    Returns:
        obj: a number representing the objective value of the solution.
    """

    # Step 1: Initialize Routing Model
    # Modify distance matrix to implement open-end routes
    n = len(distance_matrix)
    for depot_idx in set(starting_depots):
        for i in range(n):
            distance_matrix[i][depot_idx] = 0

    # Backhaul Constraint (enforce delivery-before-pickup sequence)
    # Identify linehaul and backhaul customer indices
    linehaul_nodes = [i for i in range(len(demands)) if demands[i] > 0]
    backhaul_nodes = [i for i in range(len(backhauls)) if backhauls[i]
        > 0]

    # Prohibit arcs from backhaul to linehaul nodes
    for b in backhaul_nodes:
        for l in linehaul_nodes:
            distance_matrix[b][l] = 1_000_000_000
```

```
    # Initialize with multi-depot configuration
    manager = pywrapcp.RoutingIndexManager(
        len(distance_matrix),
        num_vehicles,
        starting_depots,  # each vehicle's starting depot
        starting_depots   # set ends to same depots (will be ignored
            due to zero return costs)
    )

    routing = pywrapcp.RoutingModel(manager)

    # Step 2: Create Distance Callback and Set Arc Costs
    def distance_callback(from_index, to_index):
        from_node = manager.IndexToNode(from_index)
        to_node = manager.IndexToNode(to_index)
        return distance_matrix[from_node][to_node]

    distance_callback_index = routing.RegisterTransitCallback(
        distance_callback)
    routing.SetArcCostEvaluatorOfAllVehicles(distance_callback_index)

    # Step 3: Create Additional Callbacks
    # Time Callback (combines travel time with service time)
    def time_callback(from_index, to_index):
        from_node = manager.IndexToNode(from_index)
        to_node = manager.IndexToNode(to_index)
        # Travel time based on distance + service time at from_node
        return distance_matrix[from_node][to_node] + service_times[
            from_node]

    time_callback_index = routing.RegisterTransitCallback(time_callback
        )

    # Delivery Demand Callback (unary)
    def demand_callback(from_index):
        from_node = manager.IndexToNode(from_index)
        return demands[from_node]

    demand_callback_index = routing.RegisterUnaryTransitCallback(
        demand_callback)

    # Backhaul Demand Callback (unary)
    def backhaul_callback(from_index):
        from_node = manager.IndexToNode(from_index)
        return backhauls[from_node]

    backhaul_callback_index = routing.RegisterUnaryTransitCallback(
        backhaul_callback)

    # Step 4: Add Dimensions
    # Time Dimension (for time windows and service time tracking)
    horizon = max(tw[1] for tw in time_windows) * 2
    routing.AddDimension(
        time_callback_index,
        horizon,  # allow waiting
        horizon,  # maximum time per route
        False,    # don't fix start to zero
        'Time'
    )
    time_dimension = routing.GetDimensionOrDie('Time')

    # Capacity Dimension (for delivery demands)
    routing.AddDimensionWithVehicleCapacity(
        demand_callback_index,
```

```
        0,  # no slack
        [capacity] * num_vehicles,
        True,  # start cumul to zero
        'Capacity'
    )

    # Backhaul Dimension (for pickup demands)
    routing.AddDimensionWithVehicleCapacity(
        backhaul_callback_index,
        0,  # no slack
        [capacity] * num_vehicles,
        True,  # start cumul to zero
        'Backhaul'
    )

    # Step 5: Enforce Constraints
    # Time Window Constraints
    # Apply time windows to all customer nodes (skip depot nodes)
    depot_nodes = set(starting_depots)
    for location_idx in range(len(time_windows)):
        if location_idx in depot_nodes:
            continue
        index = manager.NodeToIndex(location_idx)
        time_dimension.CumulVar(index).SetRange(
            time_windows[location_idx][0],
            time_windows[location_idx][1]
        )

    # Apply depot time windows to vehicle start nodes
    for vehicle_id in range(num_vehicles):
        start_index = routing.Start(vehicle_id)
        depot_idx = starting_depots[vehicle_id]
        time_dimension.CumulVar(start_index).SetRange(
            time_windows[depot_idx][0],
            time_windows[depot_idx][1]
        )

    # Minimize start and end times
    for vehicle_id in range(num_vehicles):
        routing.AddVariableMinimizedByFinalizer(
            time_dimension.CumulVar(routing.Start(vehicle_id))
        )
        routing.AddVariableMinimizedByFinalizer(
            time_dimension.CumulVar(routing.End(vehicle_id))
        )


    # Step 6: Configure Search Parameters
    search_parameters = pywrapcp.DefaultRoutingSearchParameters()
    search_parameters.first_solution_strategy = (
        routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC
    )
    search_parameters.local_search_metaheuristic = (
        routing_enums_pb2.LocalSearchMetaheuristic.GUIDED_LOCAL_SEARCH
    )
    search_parameters.time_limit.seconds = 30

    # Step 7: Solve and Return Objective Value
    solution = routing.SolveWithParameters(search_parameters)
    if solution:
        return solution.ObjectiveValue()
    return -1
```

J THE USE OF LARGE LANGUAGE MODELS

In this work, LLMs were used to polish the texts, specifically for improving grammar and prose. LLMs are also a core component of our method, where they automate VRP solving. The authors, however, conceived, prepared, and finalized all aspects of the manuscript.