

Beyond Objective Equivalence: Constraint Injection for LLM-Based Optimization Modeling on Vehicle Routing Problems

Xizi Luo^{1,3}, Changhong He^{2,3}, Dongdong Geng³, Chenggong Shi³, Yu Mei^{3*}

¹School of Economics and Management, Beihang University, Beijing, China

²School of Mechanical Engineering and Automation, Beihang University, Beijing, China

³Baidu Inc., Beijing, China

Abstract

Large language models (LLMs) can generate executable solver code from natural-language descriptions of optimization problems. However, existing verification signals focus on whether the generated formulation produces the correct objective value. Such signals can overlook constraint-level errors: incorrect formulations may still produce the same optimum when extra or missing constraints do not affect the tested instance. We propose constraint injection, a verification method that directly tests whether the generated code implements the intended constraints. We use feasible solutions to detect incorrectly added constraints and one-constraint-violating solutions to detect omissions. Together with the optimal value check, these tests verify both the objective and the constraint set. We evaluate this approach on vehicle routing problems (VRPs), which provide a challenging testbed with diverse, tightly coupled operational constraints. We develop VRPCoder, an 8B model for generating Gurobi code from natural-language VRP descriptions, and VRPBench, an expert-verified benchmark spanning 21 VRP variants across four subsets. The verifier serves as a rejection-sampling criterion for synthetic data construction and as a rollout-level reward for reinforcement learning (RL). On VRPBench, VRPCoder-RL reaches 93% overall Pass@1, outperforms Gemini-3.1-Pro Preview on three subsets, exceeds Claude-Sonnet-4.5 by 28 percentage points, and exceeds the strongest prior OR-LLM by 78 percentage points.

1 Introduction

Optimization problems are often described in natural language, but solving them requires translating these descriptions into formal mathematical models and executable solver code (Ramamonjison et al., 2022). Large language models (LLMs) provide a promising way to automate this process.

*Corresponding author: whqyq@hotmail.com

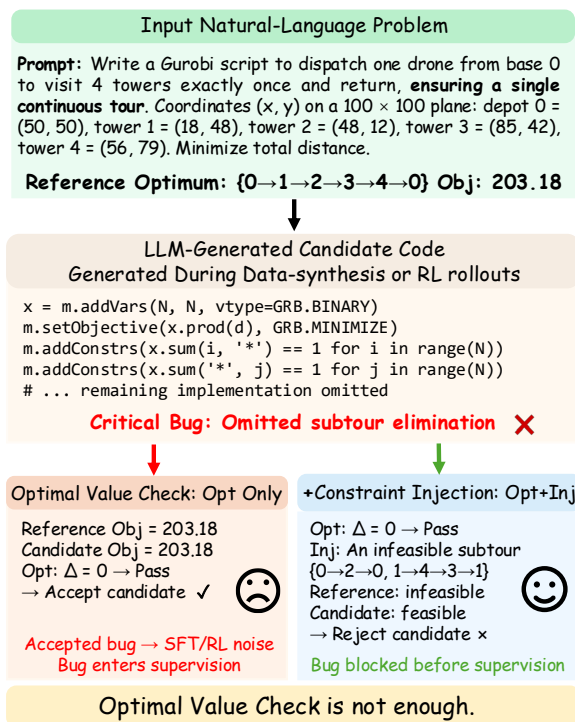


Figure 1: A candidate missing the subtour-elimination constraint still matches the reference optimal value, so the optimal value check accepts it. Constraint injection injects a disconnected-subtour solution: correct code rejects it as infeasible, while the buggy candidate accepts it, exposing the missing constraint.

However, verifying the correctness of the generated formulation remains difficult: code that runs successfully and produces the expected objective value may still fail to faithfully encode the intended constraints. Such errors are particularly concerning in optimization modeling, where an incorrect constraint set can yield seemingly valid solutions while misrepresenting the original problem.

Existing LLM-based optimization modeling methods mainly fall into three categories. Inference-time enhancement methods improve frozen LLMs via prompting, retrieval, self-

debugging, or agentic workflows (Jiang et al., 2025c; Li et al., 2025; Zhang et al., 2026). Supervised fine-tuning (SFT) trains operations research LLMs (OR-LLMs) on synthetic problem-code pairs, often filtered by executability, optimal value checks, or self-consistency (Huang et al., 2025a; Lu et al., 2025; Zhang et al., 2025). Reinforcement learning (RL) methods use solver feedback such as objective matching or answer agreement (Chen et al., 2025; Zhou et al., 2026; Ding et al., 2026). Despite their differences, both SFT filtering and RL rewards often reduce to objective equivalence, typically assessed by comparing optimal objective values.

However, objective equivalence does not guarantee constraint fidelity. A generated formulation can match the reference optimum even if it adds redundant constraints or omits required ones, provided that the affected constraints are non-binding on the tested instance. We call these two failure modes *spurious over-constraint* and *silent constraint omission*. Since both can survive executability and optimal value checks, they can contaminate SFT data and be rewarded during RL. Figure 1 shows such a case in a routing instance where the omission of a subtour-elimination constraint leaves the optimum unchanged.

We address this gap with constraint injection, a verification operator that checks whether generated code accepts a feasible solution and rejects one-constraint-violating solutions. Combined with the optimal value check, it forms a dual verifier whose constraint-level signal is decoupled from the optimum on any single instance. We instantiate the dual verifier on vehicle routing problems (VRPs), a constraint-dense testbed whose capacities, time windows, depots, and service rules are coupled into natural-language scenarios (Toth and Vigo, 2014; Laporte, 2009; Lahyani et al., 2015). The verifier is reused first as a rejection-sampling filter during data synthesis and then as a per-rollout reward during RL. The resulting model, VRPCoder, is an 8B end-to-end model that translates natural-language VRP scenarios into Gurobi code, accompanied by VRPBench¹, an expert-verified benchmark covering 21 VRP variants across four subsets.

Our contributions are as follows: (1) We identify objective equivalence as a shared blind spot of SFT data filtering and RL rewards, with two

failure modes: spurious over-constraint and silent constraint omission. (2) We propose constraint injection, a constraint-level verifier that uses feasible solutions to detect spurious over-constraints and one-constraint-violating solutions to detect silent omissions, and combine it with the optimal value check into a dual verifier reused for data synthesis and RL. (3) We introduce VRPCoder together with VRPBench, an expert-verified benchmark covering 21 VRP variants across four subsets. VRPCoder-RL reaches 93.00% overall Pass@1 on VRPBench, outperforms Gemini-3.1-Pro Preview on three subsets, and substantially outperforms Claude-Sonnet-4.5 and prior OR-LLMs.

2 Related Work

2.1 Supervised Fine-Tuning of OR-LLMs

Data synthesis has been widely adopted to train OR-LLMs (Xiao et al., 2025). ORLM (Huang et al., 2025a) expands industry-case seeds with GPT-4; OptMATH (Lu et al., 2025) generates complexity-controlled problems and filters them via bidirectional re-modeling; LLMOPT (Jiang et al., 2025a) defines a five-element unified formulation for multi-instruction tuning; ReSocratic (Yang et al., 2025) derives problem-code pairs from structured demonstrations. Across many such pipelines, sample verification relies mainly on executability, re-modeling consistency, or objective equivalence. While useful for filtering invalid or clearly incorrect code, such signals do not verify whether the generated code faithfully implements the intended constraints, and can therefore miss constraint omissions that leave the optimum unchanged.

2.2 Reinforcement Learning for OR-LLMs

Beyond SFT, RL with solver-based verifiable rewards has become a natural direction for OR-LLMs, since generated code can be executed, solved, and evaluated by feasibility, objective values, or answer agreement (Le et al., 2022). SIRL (Chen et al., 2025) uses solver execution as outcome feedback, FOARL (Jiang et al., 2025b) introduces feasibility-and-optimality-aware RL for combinatorial optimization, StepORLM (Zhou et al., 2026) uses a generative process reward model for step-wise supervision, and OR-R1 (Ding et al., 2026) performs test-time group relative policy optimization (GRPO) using majority-voting pseudo-labels as the reward. Although

¹<https://github.com/ccccziiii/VRPBench>

these rewards improve OR reasoning beyond SFT, they mainly operate at the solution or answer level and do not directly verify whether each required constraint is implemented in the generated code. For constraint-dense VRP solver-code generation, rewarding constraint-level correctness beyond solver outcomes or answer agreement remains underexplored.

2.3 LLMs for Vehicle Routing Problems

VRP is a representative testbed for LLM-based optimization modeling under coupled constraints. NLCO (Jiang et al., 2026) benchmarks LLM reasoning on natural-language combinatorial optimization including routing tasks; Huang et al. (2024) show that general-purpose LLMs exhibit preliminary VRP code-generation ability but remain limited under complex constraints. Another line of work improves VRP solving through inference-time mechanisms: DRoC (Jiang et al., 2025c) combines constraint decomposition with retrieval and self-debugging, while ARS (Li et al., 2025) and AFL (Zhang et al., 2026) adopt agentic workflows over frozen LLMs. These studies do not directly address whether generated solver code faithfully encodes intended constraints; this is the focus of our work.

3 Preliminaries

3.1 Problem Definition

We formulate the task of *automated optimization modeling*: given a natural-language problem description p , an LLM generates solver code y that produces an optimal solution and its optimal value upon execution:

$$y = \text{LLM}(p), \quad (s^*, z^*) = \text{Solve}(y) \quad (1)$$

where s^* denotes the optimal solution and z^* the corresponding optimal value. A correct y must (1) produce a feasible optimal solution with the correct optimal value z^* and (2) faithfully encode the explicit constraints in p together with the implicit constraints of the corresponding problem variant. We use *objective equivalence* to denote agreement in the optimal objective value between generated and reference code on the same instance; such agreement alone does not guarantee faithful constraint encoding. In this work, y is instantiated as Gurobi solver code for VRPs.

3.2 Vehicle Routing Problem

The Capacitated Vehicle Routing Problem (CVRP) (Braekers et al., 2016) is adopted as the foundation for VRP variants. Consider a graph with customer set $N = \{1, \dots, n\}$, depot node 0, node set $V = N \cup \{0\}$, vehicle set M , and arc set A . Each customer i has demand d_i , each vehicle has capacity Q , and each arc (i, j) has cost c_{ij} . Let $x_{ij}^k \in \{0, 1\}$ indicate whether vehicle k traverses arc (i, j) , and let $v_i \in [1, n]$ denote the position of customer i in its route. The CVRP minimizes total routing cost (2) subject to: each customer is visited exactly once (3), each vehicle departs at most once (4), flow conservation (5), vehicle capacity (6), and subtour elimination via Miller–Tucker–Zemlin (MTZ) ordering (7):

$$\min \sum_{k \in M} \sum_{(i,j) \in A} c_{ij} x_{ij}^k, \quad (2)$$

$$\sum_{k \in M} \sum_{j: (i,j) \in A} x_{ij}^k = 1, \quad \forall i \in N, \quad (3)$$

$$\sum_{j: (0,j) \in A} x_{0j}^k \leq 1, \quad \forall k \in M, \quad (4)$$

$$\sum_{j: (i,j) \in A} x_{ij}^k = \sum_{j: (j,i) \in A} x_{ji}^k, \quad (5)$$

$$\forall i \in V, k \in M,$$

$$\sum_{i \in N} d_i \sum_{j: (i,j) \in A} x_{ij}^k \leq Q, \quad \forall k \in M, \quad (6)$$

$$v_i - v_j + n \sum_{k \in M} x_{ij}^k \leq n - 1, \quad (7)$$

$$\forall i, j \in N, i \neq j.$$

VRP variants. We extend CVRP with constraint modules such as time windows, pickup-delivery, backhauls, multi-depot, heterogeneous fleets, and open routes, yielding 21 variants in total: 18 used for training and 3 held out for generalization testing. Full mapping is provided in Appendix A.

3.3 Abstract and Concrete Code

To enable scalable data synthesis, we decompose Gurobi solver code into two layers based on the observation that a VRP instance is fully determined by a parameter set (nodes, vehicles, demands, capacities, costs, and variant-specific fields such as time windows or fleet attributes).

Abstract code C_{abs}^v . For each variant v , an OR expert writes a Gurobi code template that defines

the decision variables, constraints, and objective, with instance-specific parameters left as inputs. C_{abs}^v serves as the reusable template for variant v .

Concrete code C_{con} . Instantiating C_{abs}^v with a sampled parameter set $\xi \sim \mathcal{P}_v$ yields executable concrete code:

$$C_{\text{con}} = C_{\text{abs}}^v(\xi), \quad \xi \sim \mathcal{P}_v \quad (8)$$

where $\mathcal{P}_v$ is the parameter distribution for variant v . Since each ξ yields a distinct instance, arbitrarily many C_{con} can be generated from a single template.

4 Method

4.1 Overview

Our framework consists of a data synthesis pipeline and a training pipeline. In the data synthesis pipeline, we instantiate expert-written abstract code into concrete code, construct feasible and one-constraint-violating solutions, back-translate the concrete code into natural-language problems, and regenerate solver code with a teacher LLM. Each generated sample is then filtered by the dual verifier, which jointly applies the optimal value check and the constraint injection check. In the training pipeline, accepted samples are first used for SFT; the same dual verifier is then reused as the reward signal during RL with GRPO. The following sections detail each component.

4.2 Data Synthesis Pipeline

Figure 2 summarizes our data synthesis pipeline.

Stage 1: Code Instantiation and Solution Construction. For each of the 18 training variants, we sample multiple parameter sets and substitute them into C_{abs}^v , producing multiple concrete code instances $\{C_{\text{con}}\}$ per abstract template. Specifically, each instance is parameterized by customer count $|N|$, vehicle count $|M|$, customer coordinates, and base demand range. Variant-specific parameters such as capacities, time windows, distance limits, and fleet types are then derived accordingly. We define 12 regular scale profiles and 3 boundary profiles that test extreme geometric layouts, tight capacities, and narrow time-resource budgets. Parameters are randomly sampled or selected under a fixed random seed per instance (see Appendix B.1 and Appendix B.2). We then construct a feasible solution s^+ and a set of infeasible solutions $\{s_i^-\}$ for each C_{con} as follows.

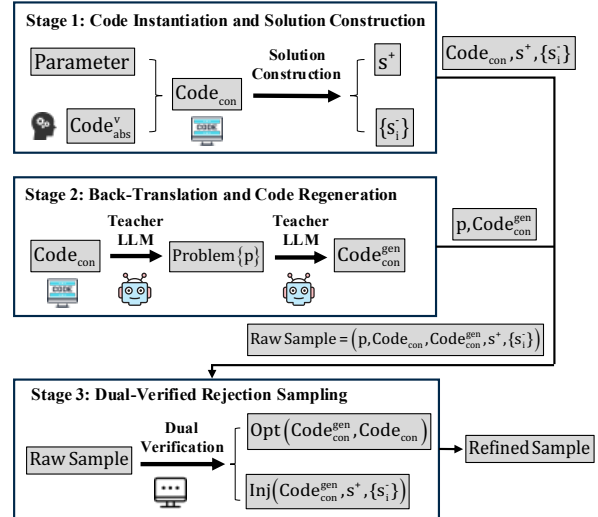


Figure 2: Overview of the three-stage data synthesis pipeline.

Feasible solution s^+ . We construct s^+ with a greedy nearest-neighbor heuristic that, starting from a depot, iteratively appends the closest feasible customer to the current route, opens a new vehicle when no extension satisfies capacity or distance limits, and inserts any remaining customers into routes with the largest slack. If the heuristic fails to produce a feasible solution, we replace the objective of C_{con} with a constant and invoke Gurobi to return any feasible solution under a short time limit.

Infeasible solutions $\{s_i^-\}$. For each variant v , we define a set of perturbation operators $\mathcal{A}_v$ that perturb s^+ into infeasible solutions $\{s_i^-\}$. Each s_i^- violates exactly one class of constraints while satisfying all others, which we term a *one-constraint-violating* solution. This design isolates the effect of specific categories of constraints, enabling the model to learn fine-grained feasibility reasoning. The operators fall into two categories depending on whether they modify the parameters of C_{con} .

- **Structural perturbations ($\mathcal{A}_v^{\text{struct}}$)** keep all parameters of C_{con} unchanged and perturb s^+ alone, introducing a local structural change that renders the resulting s_i^- infeasible with respect to exactly one class of constraints. They are applicable to specific categories of constraints—subtour elimination, precedence, and coverage—whose violations are determined by solution topology alone, independent of numerical parameters such as capacities or time windows. Figure 3(a) illus-

trates a CVRP example: the *subtour cycle* operator removes the depot connections from route $0 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 0$ to form the closed cycle $3 \rightarrow 4 \rightarrow 5 \rightarrow 3$, which disconnects these nodes from the depot and thereby violates the MTZ subtour elimination constraints (Eq. (7)).

- *Parameter perturbations* ($\mathcal{A}_v^{\text{param}}$) construct s_i^- by modifying s^+ together with the relevant parameter settings in C_{con} , producing a perturbed concrete code $\tilde{C}_{\text{con}}$. They are needed for resource-bound categories of constraints—capacity, distance, compartment, and hard time windows—where s^+ leaves slack under the original parameters, so structural changes alone may not trigger infeasibility. The relevant parameter settings are adjusted so that s^+ remains feasible under $\tilde{C}_{\text{con}}$ while s_i^- becomes infeasible. Figure 3(b) illustrates a CVRP example: the *move overload* operator moves customer 5 between routes, changing route loads from (14.5, 16.1) to (18.0, 12.6), and adjusts capacity to $\tilde{Q} = 17.48$ so that s^+ stays feasible while s_i^- violates Eq. (6).

Across the 18 training variants, our operator catalogue contains 5 structural and 7 parameter perturbation operators in total. Per-variant parameter ranges and the full operator catalogue are listed in Appendices B.2 and B.4, respectively.

Stage 2: Back-Translation and Code Regeneration. A teacher LLM back-translates each C_{con} into a set of natural-language problems $\{p\}$ via one core path and two augmentation paths, so that a single C_{con} yields multiple training samples: (1) *Scenario Instantiation* (core path) builds a baseline problem p_0 across 60 scenarios via a generate–critique–repair loop (Lu et al., 2025); (2) *Condensation* (augmentation) eliminates solver-exclusive modeling elements or simplifies business expressions, producing a concise p_{cond} ; (3) *Index Rewriting* (augmentation) varies node-ID conventions, producing p_{idx} (Jiang et al., 2026). Subsequently, the same teacher maps each p to a target code $C_{\text{con}}^{\text{gen}}$, which serves as the SFT ground truth. Using $C_{\text{con}}^{\text{gen}}$ rather than C_{con} as the training target introduces cross-sample diversity in code structure. Accordingly, each sample now takes the form $(p, C_{\text{con}}, C_{\text{con}}^{\text{gen}}, s^+, \{s_i^-\})$. Detailed back-translation procedures and prompts are

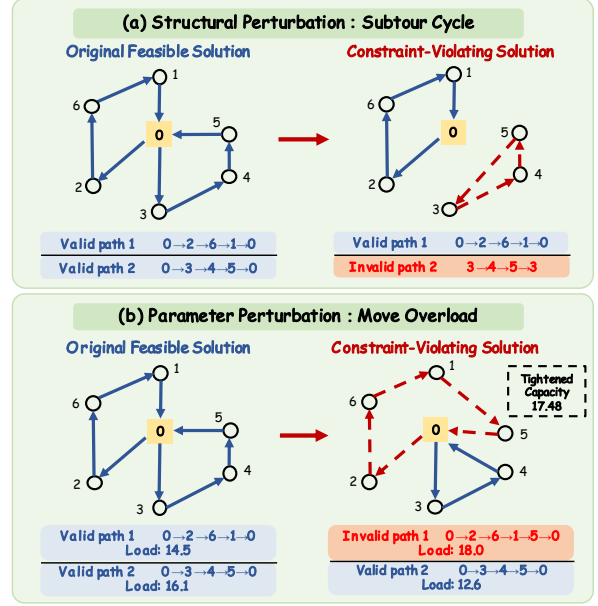


Figure 3: Two illustrative perturbation operators: (a) Subtour Cycle (structural perturbation); (b) Move Overload (parameter perturbation).

provided in Appendix C and Appendix H.

Stage 3: Dual-Verified Rejection Sampling. Since $C_{\text{con}}^{\text{gen}}$ is generated by the teacher LLM rather than derived from verified templates, it may contain modeling errors. To filter out incorrect samples, we apply a dual verification mechanism that checks each $C_{\text{con}}^{\text{gen}}$ with two complementary checks against the ground-truth C_{con} and the solution set $\mathcal{S} = \{s^+\} \cup \{s_i^-\}_{i=1}^m$.

Optimal value check Opt. Let $z^*(C)$ denote the optimal value returned by executing code C . We independently solve both $C_{\text{con}}^{\text{gen}}$ and the ground-truth C_{con} , then compare their optimal values:

$$\text{Opt}(C_{\text{con}}^{\text{gen}}, C_{\text{con}}) \triangleq |z^*(C_{\text{con}}^{\text{gen}}) - z^*(C_{\text{con}})|. \quad (9)$$

The check passes if $\text{Opt} \leq \varepsilon_{\text{obj}}$, where ε_{obj} is a small numerical tolerance accounting for floating-point differences in solver outputs; if $C_{\text{con}}^{\text{gen}}$ fails to compile or yields an infeasible model, the check fails immediately. However, matching the optimal value alone cannot guarantee faithful constraint encoding in $C_{\text{con}}^{\text{gen}}$. This motivates the following constraint injection check.

Constraint injection check Inj. Given generated code $C_{\text{con}}^{\text{gen}}$, constraint injection checks whether the encoded constraints are consistent with the labeled solutions $\mathcal{S}$ constructed from C_{con} . Specifically, we do not modify the constraints originally generated by $C_{\text{con}}^{\text{gen}}$, but replace

the objective function with a constant and, for each solution $s \in \mathcal{S}$, append arc-level constraints $\Phi(s)$ via Gurobi addConstr calls so that the solver answers a pure feasibility query. Let the ground-truth label be $\ell(s)$, where $\ell(s^+) = \text{Feasible}$ and $\ell(s_i^-) = \text{Infeasible}$. The constraint injection check passes if and only if:

$$\text{Inj}(C_{\text{con}}^{\text{gen}}, s) = \ell(s), \quad \forall s \in \mathcal{S}. \quad (10)$$

Recall that N denotes the customer set and V denotes the full node set of the current VRP variant. Let $E(s) \subseteq N \times N$ denote the customer-to-customer arcs used by solution s , and $U(s) \subseteq N$ the customers left unserved by s . Since a solution contains multiple routes, we further define $E_r(s)$ as the customer-to-customer arcs on route r . Below we describe how to construct the injection constraints $\Phi(s)$ from s . For each solution $s \in \mathcal{S}$, we first fix its customer-level routing structure with the following constraint set, denoted by $\Phi_{\text{cust}}(s)$:

$$\begin{aligned} \sum_{k \in M} x_{uv}^k &= \mathbb{I}[(u, v) \in E(s)], \quad \forall u, v \in N, \\ \sum_{u \in V \setminus \{c\}} \sum_{k \in M} x_{uc}^k &= 0, \quad \forall c \in U(s). \end{aligned} \quad (11)$$

That is, each customer-to-customer arc (u, v) is fixed according to the indicator $\mathbb{I}[(u, v) \in E(s)]$: arcs used in s must be traversed by exactly one vehicle, whereas all customer arcs not appearing in $E(s)$ are deactivated. Moreover, every unserved customer $c \in U(s)$ has its incoming arcs blocked entirely. By injecting s^+ , we can detect whether $C_{\text{con}}^{\text{gen}}$ introduces spurious constraints that incorrectly exclude a feasible solution. For a homogeneous fleet, this gives $\Phi(s^+) = \Phi_{\text{cust}}(s^+)$.

For an infeasible solution s_i^- under a homogeneous fleet, the customer-level injection $\Phi_{\text{cust}}(s_i^-)$ alone is insufficient: the solver can reassign arcs across vehicles to silently repair the violation. We therefore add route-level consistency constraints:

$$\begin{aligned} \Phi_{\text{route}}(s_i^-): \quad x_{ij}^k &= x_{ab}^k, \\ \forall k \in M, \forall (i, j), (a, b) &\in E_r(s_i^-), \forall r. \end{aligned} \quad (12)$$

This ensures that all customer arcs on the same route r are assigned to a single vehicle, without specifying which one. For a homogeneous fleet, the full injection for s_i^- is thus:

$$\Phi(s_i^-) = \Phi_{\text{cust}}(s_i^-) \cup \Phi_{\text{route}}(s_i^-). \quad (13)$$

In this way, the violation introduced by perturbation is preserved on its original route, enabling us to check whether $C_{\text{con}}^{\text{gen}}$ misses the corresponding category of constraint. For heterogeneous fleets, both s^+ and $\{s_i^-\}$ instead use $x_{uv}^{\kappa_s(u,v)} = 1$ for every used customer arc (u, v) , where $\kappa_s(u, v)$ is its assigned vehicle in s ; unused arcs and unserved customers are handled as above, with details in Appendix D.

4.3 Model Training Pipeline

Stage 1: Supervised Fine-Tuning. Based on the data synthesis pipeline described above, we construct the verified SFT dataset:

$$\mathcal{D} = \{(p, C_{\text{con}}^{\text{gen}})\}. \quad (14)$$

For the SFT stage, we fine-tune a pre-trained LLM on $\mathcal{D}$, where the input is the problem p and the target is the tokenization $y = (y_1, \dots, y_T)$ of $C_{\text{con}}^{\text{gen}}$. The training objective minimizes the standard autoregressive negative log-likelihood:

$$\mathcal{L}_{\text{SFT}}(\theta) = -\mathbb{E}_{(p,y) \sim \mathcal{D}} \left[\sum_{t=1}^T \log p_{\theta}(y_t \mid p, y_{<t}) \right], \quad (15)$$

where $p_{\theta}(y_t \mid p, y_{<t})$ denotes the probability of generating token y_t conditioned on the problem description p and all preceding tokens $y_{<t}$. The fine-tuned model is denoted **VRPCoder-SFT**.

Stage 2: Reinforcement Learning with GRPO.

While SFT teaches the model to imitate teacher-generated code, it cannot leverage execution feedback to further improve constraint modeling accuracy. To address this, we apply GRPO (Shao et al., 2024) to fine-tune the policy π_{θ} , initialized from VRPCoder-SFT, using reward signals derived from solver execution. Figure 4 summarizes the model training pipeline.

Reward function. For a code y generated for problem p , the reward is a weighted combination of three components:

$$\begin{aligned} r(p, y) &= \lambda_{\text{build}} r_{\text{build}}(y) + \lambda_{\text{opt}} r_{\text{opt}}(y, C_{\text{con}}) \\ &\quad + \lambda_{\text{inj}} r_{\text{inj}}(y, \mathcal{S}), \end{aligned} \quad (16)$$

where the individual terms are defined as follows: (1) $r_{\text{build}}(y) = \mathbb{I}[\text{build}(y)]$, indicating whether y successfully constructs a valid Gurobi model; (2) $r_{\text{opt}}(y, C_{\text{con}}) = \mathbb{I}[\text{Opt}(y, C_{\text{con}})] \leq$

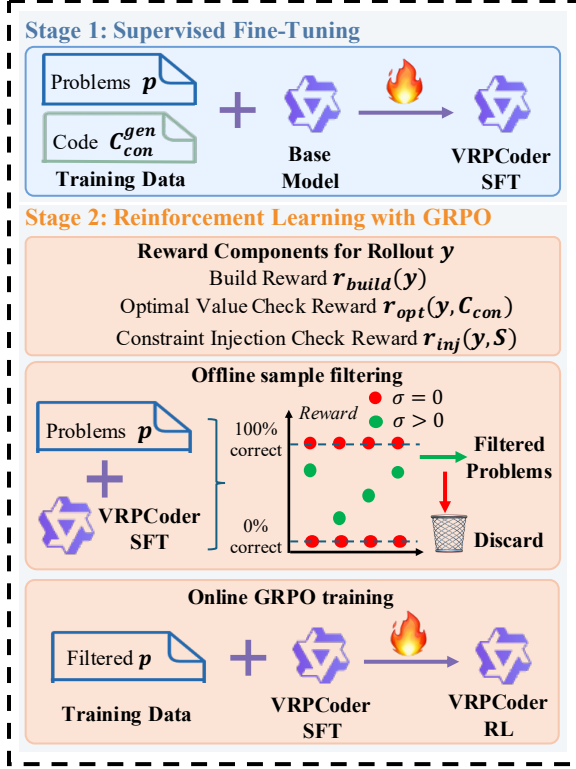


Figure 4: Overview of the model training pipeline.

ε_{obj}], indicating whether the optimal value produced by y matches that returned by the reference concrete code C_{con} ; and (3) $r_{\text{inj}}(y, \mathcal{S}) = |\mathcal{S}|^{-1} \sum_{s \in \mathcal{S}} \mathbb{I}[\text{Inj}(y, s) = \ell(s)]$, measuring the fraction of solutions in $\mathcal{S}$ that pass the constraint injection check. Here, $\mathbb{I}[\cdot]$ denotes the indicator function, and λ_{build} , λ_{opt} , and λ_{inj} denote the weights assigned to the three reward components, respectively.

Offline sample filtering. To ensure effective policy optimization, we filter training samples that provide meaningful learning signal. Specifically, for each problem p , we pre-generate H rollouts using VRPCoder-SFT and compute the within-group reward standard deviation:

$$\sigma_p = \sqrt{\frac{1}{H} \sum_{h=1}^H (r(p, y_h) - \bar{r}_p)^2}, \quad (17)$$

where $\bar{r}_p = \frac{1}{H} \sum_{h=1}^H r(p, y_h)$. We retain only samples with $\sigma_p > 0$, discarding groups whose rollouts receive identical rewards, as they provide no within-group learning signal.

Online GRPO training. On the dataset $\mathcal{D}_{\text{GRPO}}$ obtained after sample filtering, for each problem p we sample G rollouts from $\pi_{\theta_{\text{old}}}$, compute group-normalized advantages, and update π_{θ} via:

$$\mathcal{J}(\theta) = \mathbb{E} \left[\frac{1}{G} \sum_{i=1}^G \frac{1}{|o_i|} \sum_{t=1}^{|o_i|} (\mathcal{L}_{i,t}^{\text{clip}} - \beta D_{\text{KL}}^{(i,t)}) \right], \quad (18)$$

where the expectation is over prompts $p \sim \mathcal{D}_{\text{GRPO}}$ and rollout groups $\{o_i\}_{i=1}^G$ sampled from $\pi_{\theta_{\text{old}}}(\cdot | p)$. For token t in rollout o_i , the importance ratio is $\rho_{i,t} = \pi_{\theta}(o_{i,t} | p, o_{i,<t}) / \pi_{\theta_{\text{old}}}(o_{i,t} | p, o_{i,<t})$. The clipped surrogate is $\mathcal{L}_{i,t}^{\text{clip}} = \min(\rho_{i,t} \hat{A}_i, \bar{\rho}_{i,t} \hat{A}_i)$, where $\bar{\rho}_{i,t} = \text{clip}(\rho_{i,t}, 1 - \epsilon, 1 + \epsilon)$. The rollout-level group-normalized advantage is $\hat{A}_i = (r(p, o_i) - \mu_r) / (\sigma_r + \varepsilon_{\text{adv}})$, where μ_r and σ_r are the mean and standard deviation of $\{r(p, o_j)\}_{j=1}^G$, and $\varepsilon_{\text{adv}} > 0$ is a small constant for numerical stability; the same $\hat{A}_i$ is shared across all tokens in o_i . Finally, $D_{\text{KL}}^{(i,t)}$ denotes the token-level KL penalty against the reference policy π_{ref} , a frozen copy of VRPCoder-SFT.

Here β is the KL coefficient, ϵ is the clipping range, and G is the number of rollouts per problem. The resulting model is denoted **VRPCoder-RL**.

5 Experiments

5.1 Experimental Setup

Benchmark. The training dataset $\mathcal{D}$ contains 6,797 $(p, C_{\text{con}}^{\text{gen}})$ pairs produced by the synthesis pipeline in Section 4.2. For evaluation, we introduce **VRPBench**, a 700-problem expert-verified benchmark comprising four subsets: (1) *In-domain synthetic* (310 problems), generated from the 18 training variants with unseen random seeds, evaluating in-distribution generalization; (2) *Out-of-domain synthetic* (66 problems), generated from the 3 held-out compositional variants, evaluating whether the model can combine familiar constraint primitives into unseen VRP variants; (3) *Expert-written* (74 problems), sampled from public optimization libraries and formulated by OR experts, evaluating performance on human-authored problem descriptions; (4) *Externally-sourced* (250 problems), collected from MAMO (Huang et al., 2025b), IndustryOR (Huang et al., 2025a), OptMATH (Lu et al., 2025), and NLCO (Jiang et al., 2026), evaluating cross-source robustness.

Baselines. We compare against four baseline groups: (1) closed-source frontier LLMs, including Gemini-3.1-Pro Preview (Google DeepMind, 2026) and Claude-Sonnet-4.5 (Anthropic,

Model	VRPBench				
	Subset 1	Subset 2	Subset 3	Subset 4	AVG
<i>Closed-source frontier LLMs</i>					
Gemini-3.1-Pro Preview	<u>95.81</u>	<u>89.39</u>	<u>91.89</u>	96.40	95.00
Claude-Sonnet-4.5	70.65	42.42	79.73	58.80	64.71
<i>Open-source general-purpose LLMs</i>					
DeepSeek-V3.2	31.29	7.58	51.35	45.60	36.29
MiniMax-M2.5	21.61	9.09	17.57	32.80	24.00
Qwen2.5-72B-Instruct	15.81	6.06	12.16	27.20	18.57
Qwen3-8B	0.00	0.00	1.35	1.20	0.57
<i>OR-LLMs</i>					
ORLM-LLaMA-3-8B	3.87	0.00	12.16	7.60	5.71
OptMATH-Qwen2.5-7B	0.97	0.00	2.70	8.40	3.71
SIRL-Gurobi-8B	1.94	0.00	0.00	13.20	5.57
SIRL-Gurobi-32B	8.71	3.03	1.35	30.00	15.00
<i>Our 8B model</i>					
VRPCoder-SFT	92.26	<u>89.39</u>	82.43	85.20	88.43
VRPCoder-RL	96.13	96.97	93.24	<u>88.00</u>	<u>93.00</u>

Table 1: Pass@1 (%) on the four subsets of VRPBench. Best scores are **bolded**; second-best are underlined. AVG is computed over all samples in VRPBench.

Method	VRPBench					Δ AVG
	Subset 1	Subset 2	Subset 3	Subset 4	AVG	
VRPCoder-SFT w/o Injection	91.61	81.82	83.78	79.60	85.57	–
VRPCoder-SFT	92.26	89.39	82.43	85.20	88.43	+2.86
VRPCoder-RL w/o Injection	94.19	93.94	91.89	80.40	89.00	–
VRPCoder-RL	96.13	96.97	93.24	88.00	93.00	+4.00

Table 2: Ablation of constraint injection for SFT and RL on VRPBench. Best scores within each stage are **bolded**. Δ AVG reports the percentage-point gain in overall Pass@1 over the corresponding model without injection.

2025); (2) open-source general-purpose LLMs, including DeepSeek-V3.2 (DeepSeek-AI, 2025), MiniMax-M2.5 (MiniMax Team, 2026), Qwen2.5-72B-Instruct (Qwen Team, 2024), and Qwen3-8B (Qwen Team, 2025); (3) SFT-based OR-LLMs, including ORLM-LLaMA-3-8B (Huang et al., 2025a) and OptMATH-Qwen2.5-7B (Lu et al., 2025); (4) RL-based OR-LLMs, including SIRL-Gurobi-8B and SIRL-Gurobi-32B (Chen et al., 2025). We evaluate groups (1) and (2) with the shared code-regeneration prompt (Appendix H), and evaluate groups (3) and (4) with their original prompts and solver settings.

Implementation Details. Both stages use a LoRA adapter with rank $r = 16$, $\alpha = 32$, and dropout 0.1 on Qwen3-8B, trained on $4 \times A100$ -40GB GPUs. We first apply LoRA SFT on $\mathcal{D}$ for 3 epochs with a maximum sequence length of 8192, using AdamW with learning rate 2×10^{-4} and cosine scheduling, to obtain VRPCoder-SFT. We then perform offline sample filtering with $H = 6$ pre-generated rollouts per prompt to select frontier

prompts $\mathcal{D}_{\text{GRPO}}$, and run GRPO with $G = 8$ roll-outs per prompt, temperature 0.4, top- p 0.95, and KL coefficient $\beta = 0.02$. The composite reward weights are 0.2, 0.5, and 0.3 for r_{build} , r_{opt} , and r_{inj} , respectively, yielding VRPCoder-RL. Full settings are provided in Appendix E.

5.2 Main Results

Table 1 reports Pass@1 on the four subsets of VRPBench. VRPCoder-RL outperforms Gemini-3.1-Pro Preview on Subsets 1, 2, and 3, achieving 96.13%, 96.97%, and 93.24% versus 95.81%, 89.39%, and 91.89%, but trails on Subset 4, which includes the traveling salesman problem with time windows (TSPTW), a variant absent from training. It exceeds Claude-Sonnet-4.5 by a large margin, with 93.00% versus 64.71% overall Pass@1. Existing OR-LLMs achieve notably low performance across all subsets, where the strongest, SIRL-Gurobi-32B, reaches only 15.00% average Pass@1, indicating limited applicability to real-world VRP scenarios and further demonstrating

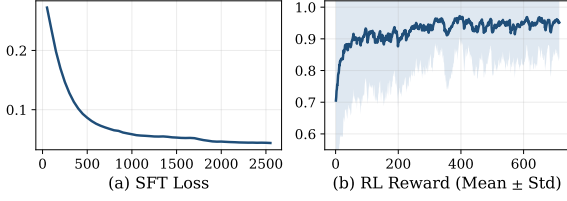


Figure 5: Training dynamics of SFT and RL. The left panel shows the SFT training loss, while the right panel shows the RL mean reward. The dark curve indicates the smoothed trend, and the shaded region visualizes local variation over training steps.

the value of VRPBench as a challenging evaluation benchmark. Before fine-tuning, the base model Qwen3-8B is essentially non-functional, achieving only 0.57% overall Pass@1; after our full pipeline, it achieves 93.00% overall Pass@1, confirming the effectiveness of the proposed training framework.

5.3 Ablation Studies

Table 2 isolates the contribution of constraint injection. The no-injection setting removes the injection signal from both stages: SFT retains samples that pass the optimal value check but fail injection, and RL scores rollouts using only build success and the optimal value check. Removing injection lowers overall Pass@1 by 2.86 percentage points for SFT (88.43% to 85.57%) and by 4.00 points for RL (93.00% to 89.00%). The largest gain appears on Subset 4, where RL improves from 80.40% to 88.00%. Subset 2 also improves substantially at the SFT stage, from 81.82% to 89.39%. These results indicate that constraint-level supervision provides information beyond the optimal value check, especially under distribution shift and unseen constraint combinations.

5.4 Model Analysis

Training dynamics. Figure 5 shows the training curves of both stages. In the SFT stage, the training loss decreases steadily from 0.272 to 0.043 over 2,550 steps, indicating stable convergence. In the RL stage, the mean reward shows a clear overall upward trend over 714 steps, with its late-stage level around 0.94.

Sensitivity Analysis. Table 3 shows the reward weight sensitivity analysis. We keep the three weights summing to 1, fix $\lambda_{\text{build}}=0.2$, and vary λ_{opt} and λ_{inj} . Pass@1 remains above 90% across all configurations and peaks at 93.00% for

λ_{build}	λ_{opt}	λ_{inj}	AVG Pass@1
0.2	0.6	0.2	91.57
0.2	0.5	0.3	93.00
0.2	0.4	0.4	92.86
0.2	0.3	0.5	90.71

Table 3: Sensitivity analysis of the three reward weights (λ_{build} , λ_{opt} , λ_{inj}) for VRPCoder-RL. Best score is **bolded**.

$\lambda_{\text{opt}}/\lambda_{\text{inj}} = 0.5/0.3$. It is slightly lower when injection is down-weighted (0.6/0.2: 91.57%) or made dominant (0.3/0.5: 90.71%). This suggests that constraint injection works best as a complement to the optimal value check rather than as the dominant reward. All weight configurations outperform removing injection entirely (Table 2).

6 Conclusion

We identified objective equivalence as a shared blind spot in SFT- and RL-based optimization-modeling pipelines, where spurious over-constraints and silent constraint omissions can be indistinguishable from correct code when the affected constraints are non-binding at the optimum. We proposed **constraint injection**, a constraint-level verification operator that uses feasible and one-constraint-violating solutions; combined with the optimal value check, it forms a dual verifier. We applied this verifier as a rejection-sampling filter during data synthesis and a per-rollout reward in GRPO. When instantiated on vehicle routing as a constraint-dense testbed, our framework yields VRPCoder-RL, which reaches 93.00% overall Pass@1 on VRPBench. It is competitive with frontier LLMs and surpasses prior OR-LLMs by a large margin. Ablations show that constraint injection is a key component: the largest improvement appears on the most distribution-shifted subset, and constraint-level supervision provides benefits beyond the optimal value check.

Limitations

Our empirical validation focuses on vehicle routing, and transferring constraint injection to other optimization domains requires expert-written abstract-code templates, constraint-family-specific perturbation operators, and compatible solution-encoding rules. This effort is incurred once per recurring constraint family rather than per instance, but transferability beyond routing

remains to be evaluated. Our one-constraint-violating solutions isolate individual constraint families while the other constraints in the generated model remain active, but they may not expose errors that arise only through interactions among multiple constraints. We report instance-level Pass@1 to enable consistent comparison across all four subsets of VRPBench. Although Appendix F reports constraint-level failures during data synthesis, benchmark-wide constraint-level evaluation of final model outputs remains future work.

Acknowledgments

This research was supported by the National Natural Science Foundation of China under Grant 72571017.

References

- Anthropic. 2025. Claude Sonnet 4.5 System Card. <https://www.anthropic.com/claude-sonnet-4-5-system-card>. Accessed: 2026-05-26.
- Anthropic. 2026. Claude Opus 4.6 System Card. <https://www.anthropic.com/claude-opus-4-6-system-card>. Accessed: 2026-05-26.
- Philippe Augerat, José Manuel Belenguer, Enrique Benavent, Angel Corberán, Denis Naddef, and Giovanni Rinaldi. 1995. Computational Results with a Branch and Cut Code for the Capacitated Vehicle Routing Problem. Technical Report 949-M, Université Joseph Fourier, Grenoble.
- José M. Belenguer, M. Cristina Martinez, and Ernesto Mota. 2000. A Lower Bound for the Split Delivery Vehicle Routing Problem. *Operations Research*, 48(5):801–810.
- Kris Braekers, Katrien Ramaekers, and Inneke Van Nieuwenhuysse. 2016. The Vehicle Routing Problem: State of the Art Classification and Review. *Computers & Industrial Engineering*, 99:300–313.
- Yitian Chen, Jingfan Xia, Siyu Shao, Dongdong Ge, and Yinyu Ye. 2025. Solver-Informed RL: Grounding Large Language Models for Authentic Optimization Modeling. In *Advances in Neural Information Processing Systems*, volume 38, pages 106027–106069. Curran Associates, Inc.
- Jean-François Cordeau, Michel Gendreau, and Gilbert Laporte. 1997. A Tabu Search Heuristic for Periodic and Multi-Depot Vehicle Routing Problems. *Networks*, 30(2):105–119.
- DeepSeek-AI. 2025. DeepSeek-V3.2: Pushing the Frontier of Open Large Language Models. *Preprint*, arXiv:2512.02556.
- Zezen Ding, Zhen Tan, Jiheng Zhang, and Tianlong Chen. 2026. OR-R1: Automating Modeling and Solving of Operations Research Optimization Problem via Test-Time Reinforcement Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 228–236.
- Bruce L. Golden, Edward A. Wasil, James P. Kelly, and I-Ming Chao. 1998. The Impact of Metaheuristics on Solving the Vehicle Routing Problem: Algorithms, Problem Sets, and Computational Results. In Teodor Gabriel Crainic and Gilbert Laporte, editors, *Fleet Management and Logistics*, pages 33–56. Springer, Boston, MA.
- Google DeepMind. 2026. Gemini 3.1 Pro Model Card. <https://deepmind.google/models/model-cards/gemini-3-1-pro/>. Accessed: 2026-05-26.
- Chenyu Huang, Zhengyang Tang, Shixi Hu, Ruqing Jiang, Xin Zheng, Dongdong Ge, Benyou Wang, and Zizhuo Wang. 2025a. ORLM: A Customizable Framework in Training Large Models for Automated Optimization Modeling. *Operations Research*, 73(6):2986–3009.
- Xuhan Huang, Qingning Shen, Yan Hu, Anningzhe Gao, and Benyou Wang. 2025b. LLMs for Mathematical Modeling: Towards Bridging the Gap between Natural and Mathematical Languages. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 2678–2710, Albuquerque, New Mexico. Association for Computational Linguistics.
- Zhehui Huang, Guangyao Shi, and Gaurav S. Sukhatme. 2024. From Words to Routes: Applying Large Language Models to Vehicle Routing. *arXiv preprint arXiv:2403.10795*.
- Caigao Jiang, Xiang Shu, Hong Qian, Xingyu Lu, Jun Zhou, Aimin Zhou, and Yang Yu. 2025a. LLMOPT: Learning to Define and Solve General Optimization Problems from Scratch. In *International Conference on Learning Representations*, volume 2025, pages 101580–101606.
- Xia Jiang, Jing Chen, Cong Zhang, Jie Gao, Chengpeng Hu, Chenhao Zhang, Yaoxin Wu, and Yingqian Zhang. 2026. Reasoning in a combinatorial and constrained world: Benchmarking LLMs on natural-language combinatorial optimization. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 30592–30648, San Diego, California, United States. Association for Computational Linguistics.
- Xia Jiang, Yaoxin Wu, Minshuo Li, Zhiguang Cao, and Yingqian Zhang. 2025b. Large language models as end-to-end combinatorial optimization solvers. In *Advances in Neural Information Processing Systems*, volume 38, pages 164787–164826. Curran Associates, Inc.
- Xia Jiang, Yaoxin Wu, Chenhao Zhang, and Yingqian Zhang. 2025c. DRoC: Elevating Large Language

- Models for Complex Vehicle Routing via Decomposed Retrieval of Constraints. In *International Conference on Learning Representations*, volume 2025, pages 46731–46752.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. **Efficient Memory Management for Large Language Model Serving with PagedAttention**. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, Koblenz, Germany. Association for Computing Machinery.
- Rahma Lahyani, Mahdi Khemakhem, and Frédéric Semet. 2015. **Rich Vehicle Routing Problems: From a Taxonomy to a Definition**. *European Journal of Operational Research*, 241(1):1–14.
- Gilbert Laporte. 2009. **Fifty Years of Vehicle Routing**. *Transportation Science*, 43(4):408–416.
- Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. **Coderl: Mastering code generation through pretrained models and deep reinforcement learning**. In *Advances in Neural Information Processing Systems*, volume 35, pages 21314–21328. Curran Associates, Inc.
- Kai Li, Fei Liu, Zhenkun Wang, Xialiang Tong, Xiongwei Han, Mingxuan Yuan, and Qingfu Zhang. 2025. **ARS: Automatic Routing Solver with Large Language Models**. *Preprint*, arXiv:2502.15359v3.
- Hongliang Lu, Zhonglin Xie, Yaoyu Wu, Can Ren, Yuxuan Chen, and Zaiwen Wen. 2025. **OptMATH: A Scalable Bidirectional Data Synthesis Framework for Optimization Modeling**. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 40769–40802. PMLR.
- MiniMax Team. 2026. MiniMax-M2.5 Model Card. <https://huggingface.co/MiniMaxAI/MiniMax-M2.5>. Accessed: 2026-05-26.
- Qwen Team. 2024. **Qwen2.5 Technical Report**. *Preprint*, arXiv:2412.15115.
- Qwen Team. 2025. **Qwen3 Technical Report**. *Preprint*, arXiv:2505.09388.
- Rindranirina Ramamonjison, Timothy Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong Zhang. 2022. **NL4Opt Competition: Formulating Optimization Problems Based on Their Natural Language Descriptions**. In *Proceedings of the NeurIPS 2022 Competitions Track*, volume 220 of *Proceedings of Machine Learning Research*, pages 189–203. PMLR.
- Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. **DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters**. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506. Association for Computing Machinery.
- Gerhard Reinelt. 1991. **TSPLIB—A Traveling Salesman Problem Library**. *ORSA Journal on Computing*, 3(4):376–384.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. **DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models**. *Preprint*, arXiv:2402.03300.
- Marius M. Solomon. 1987. **Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraints**. *Operations Research*, 35(2):254–265.
- Paolo Toth and Daniele Vigo. 1997. **An Exact Algorithm for the Vehicle Routing Problem with Backhauls**. *Transportation Science*, 31(4):372–385.
- Paolo Toth and Daniele Vigo, editors. 2014. **Vehicle Routing: Problems, Methods, and Applications**, 2 edition. Number 18 in MOS-SIAM Series on Optimization. Society for Industrial and Applied Mathematics, Philadelphia, PA.
- Eduardo Uchoa, Diego Pecin, Artur Pessoa, Marcus Poggi, Thibaut Vidal, and Anand Subramanian. 2017. **New Benchmark Instances for the Capacitated Vehicle Routing Problem**. *European Journal of Operational Research*, 257(3):845–858.
- Ziyang Xiao, Jingrong Xie, Lilin Xu, Shisi Guan, Jingyan Zhu, Xiongwei Han, Xiaojin Fu, WingYin Yu, Han Wu, Wei Shi, Qingcan Kang, Jiahui Duan, Tao Zhong, Mingxuan Yuan, Jia Zeng, Yuan Wang, Gang Chen, and Dongxiang Zhang. 2025. **A Survey of Optimization Modeling Meets LLMs: Progress and Future Directions**. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-25*, pages 10742–10750. International Joint Conferences on Artificial Intelligence Organization. Survey Track.
- Zhicheng Yang, Yiwei Wang, Yinya Huang, Zhijiang Guo, Wei Shi, Xiongwei Han, Liang Feng, Linqi Song, Xiaodan Liang, and Jing Tang. 2025. **OptiBench Meets ReSocratic: Measure and Improve LLMs for Optimization Modeling**. In *The Thirteenth International Conference on Learning Representations*.
- Ni Zhang, Zhiguang Cao, Jianan Zhou, Cong Zhang, and Yew-Soon Ong. 2026. **An Agentic Framework with LLMs for Solving Complex Vehicle Routing Problems**. In *The Fourteenth International Conference on Learning Representations*.
- Xinzhi Zhang, Zeyi Chen, Humishka Zope, Hugo Barbalho, Konstantina Mellou, Marco Molinaro, Janardhan Kulkarni, Ishai Menache, and Sirui Li. 2025.

OptiMind: Teaching LLMs to Think Like Optimization Experts. *Preprint*, arXiv:2509.22979v2. Version 2.

Chenyu Zhou, Tianyi Xu, Jianghao Lin, and Dongdong Ge. 2026. **StepORLM: A Self-Evolving Framework With Generative Process Supervision For Operations Research Language Models**. In *The Fourteenth International Conference on Learning Representations*.

A VRP Variant Overview

We build 21 VRP variants. The 21-variant count refers to the variants constructed for Subsets 1–3; the externally sourced Subset 4 additionally includes TSPTW.

MD (multi-depot) introduces multiple depots, with each vehicle departing from and returning to its assigned depot. **MC (multi-compartment)** partitions each vehicle into multiple non-mixing compartments, each with an independent capacity. **PD (pickup-and-delivery)** pairs pickup and delivery nodes that must be visited by the same vehicle in order. **Open (open routing)** allows each route to terminate at the last customer without returning to the depot. **HTW (hard time window)** imposes hard time windows $[a_i, b_i]$ on service start times; early arrival is allowed through waiting, but service cannot start before a_i or after b_i . **STW (soft time window)** relaxes time-window violations into objective penalties. **Dist (distance limit)** imposes a maximum travel distance $D_{\max}$ on each route. **BH (backhaul)** splits customers into linehaul and backhaul segments and requires all linehaul customers to precede backhaul customers on each route. **Split (split delivery)** allows a customer to be served by multiple vehicles. **HF (heterogeneous fleet)** allows capacities, fixed costs, and per-distance costs to vary by vehicle.

The Cap. column in Table 4 specifies capacity semantics: Q for homogeneous capacity, Q_k for heterogeneous capacity, and “–” for no effective capacity limit. VRPB accounts for linehaul and backhaul loads separately, whereas MCVRP and HFMCVRP account for capacities separately for each compartment.

B Constraint-Level Solution Construction

In the three-stage data synthesis pipeline (Section 4.2), Stage 1 constructs feasible and one-constraint-violating solutions with auditable feasibility labels for each training sample. Given a variant v and a profile, we first determine the customer count, fleet size, coordinate distribution, and base demand range (Appendix B.1). Variant-specific rules then specify capacities, time windows, route-distance limits, heterogeneous vehicle types, pickup–delivery pairs, compartments, and backhaul labels, yielding a complete instance and its reference concrete code C_{con} (Appendix B.2). We then construct a feasible solution s^+ (Ap-

Variant	Cap.	MD	MC	PD	Open	HTW	STW	Dist	BH	Split	HF	Train
TSP	—											✓
CVRP	Q											✓
MDVRP	Q	✓										✓
MCVRP	Q		✓									✓
PDP	Q			✓								✓
OVRP	Q				✓							✓
DCVRP	Q							✓				✓
VRPB	Q								✓			✓
SDVRP	Q									✓		✓
VRPHTW	Q					✓						✓
VRPSTW	Q						✓					✓
HFVRP	Q_k										✓	✓
HFMDVRP	Q_k	✓									✓	✓
HFMCVRP	Q_k		✓								✓	✓
HFVRPHTW	Q_k					✓					✓	✓
MDVRPSTW	Q	✓					✓					✓
PDPSTW	Q			✓			✓					✓
PDPTW	—			✓		✓						✓
<i>OVRPHTW</i>	Q				✓	✓						Held-out
<i>MCVRPSTW</i>	Q		✓				✓					Held-out
<i>DCVRPHTW</i>	Q					✓		✓				Held-out

Table 4: Structural module matrix for the 21 VRP variants. A check mark (✓) indicates the module is enabled. The *Train* column specifies the participation in SFT/GRPO training. ***Bold and italicized*** variants are strictly held out from the training mixture: their constituent modules are present in the training set, but their pairwise combinations are never seen, forming Subset 2 for evaluating compositional generalization.

pendix B.3) and derive one-constraint-violating solutions $\{s_i^-\}$ from s^+ (Appendix B.4). Finally, bounded resources are slightly relaxed and each labeled solution set $\mathcal{S}$ is checked by C_{con} before assembly (Appendix B.5). The main depot is fixed at (50, 50); multi-depot variants use two depots at (25, 50) and (75, 50); all customer coordinates are clipped to the coordinate range $[5, 95]^2$.

B.1 Profiles

A profile serves as a parameter envelope rather than a training split or an algorithmic component. It is applied before instance generation and specifies the customer count n , vehicle count k , spatial distribution, and base range for scalar customer demands, while variant-specific modules fill in capacities, time windows, pickup-delivery demand, compartment demand, and other constraints. Profiles are designed so that small-scale instances, regular layouts, and geometric, capacity, or time-boundary cases all lie in a single auditable parameter space, instead of being listed instance by instance. We use 15 profiles in total: 12 regular profiles and 3 boundary profiles; this appendix specifies their parameter coverage. Unless otherwise stated, an interval $[a, b]$ denotes a sampling envelope, and a discrete set lists the admissible choices.

We use d for scalar customer demand, Q for ho-

mogeneous vehicle capacity or a capacity bound specified by a boundary-profile grid, H for the time horizon in time-window variants, and C for the number of compartments. A grid tuple such as (n, k, Q, d) , (n, k, Q, H) , or (n, k, Q, C) denotes an instance-level configuration. If an entry contains an interval, such as $d = [3, 6]$, the tuple is first selected and each customer demand is then sampled from that interval. In Table 5, “locked” means that the profile family directly uses the resource values from its grid rather than the default rules in Appendix B.2; “—” means that the profile specifies only scale and geometry, and the resource parameters are supplied by the variant-specific rules.

Uniform, Clustered, and Radial distributions are inspired by the VRP instance designs of Augerat et al. (1995) and Uchoa et al. (2017); time-window generation follows Solomon (1987); and regular geometric arrays follow Golden et al. (1998). These references serve only as sources of design factors and do not indicate that our samples are drawn from the corresponding benchmarks.

The five spatial distributions define geometric families for customer coordinates only, rather than complete VRP instances. Capacities, time windows, pickup-delivery pairs, compartments, and other constraint modules are supplied by Ap-

pendix B.2. **Uniform** samples customer coordinates independently in the plane. **Clustered** first samples 2–3 centers and then draws compact customer groups around them. **Radial** samples angles and radii around the depot. **Elongated** forms a narrow corridor along a random direction. **Split** places customers in two separated groups.

For profile 13, only the hexagon subfamily fixes $Q = 100$, keeping capacity safely non-binding so that the resulting solutions focus on depot connectivity and subtour structure; other geometry-boundary subfamilies still take capacity values from Appendix B.2. Profiles 14 and 15 are marked as locked because they are generated by directly sampling grid tuples such as (n, k, Q, d) , (n, k, Q, H) , or (n, k, Q, C) .

The boundary profiles specify parameter arrangements rather than new VRP variants or training subsets. Profile 13 covers deterministic geometric structures. Its hexagon subfamily uses $n = 6, k = 1$, places customers on a regular hexagon with radius $r \in [15, 25]$, places the depot at the midpoint of one hexagon edge, and fixes $Q = 100$. The polygon subfamily samples $n \in \{4, 6, 8\}$ and places customers on a regular polygon centered at $c \in [45, 55]^2$, with radius $r \in [25, 40]$ and a random rotation. The collinear / coverage subfamily places $n \in \{5, 6, 7, 8\}$ customers at equal spacing on a randomly rotated segment, with spacing in $[3, 8]$. The local-cluster subfamily places customers within radius $r \in [1, 3]$ around a center in $[20, 80]^2$.

Profile 14 samples from several (n, k, Q, d) grids so that the average route demand $\bar{d}n/k$ is close to the capacity bound. The small-grid settings include $(5, 2, 12, [3, 6])$, $(6, 2, 14, [3, 7])$, $(6, 3, 12, [2, 5])$, $(8, 3, 16, [2, 6])$, $(6, 3, 8, [3, 4])$, $(8, 4, 8, [3, 4])$, $(6, 2, 12, [3, 5])$, and $(8, 3, 12, [3, 5])$. To cover split delivery and heavy-route cases, the extended grids allow n up to 12, k up to 4, scalar demand up to $[2, 20]$, and capacities within the envelope specified in Appendix B.2. Customer coordinates use uniform or split-cluster layouts.

Profile 15 includes time-window, pickup-delivery, and compartment subfamilies. The time-window subfamily arranges customers in a radial sequence, uses $d \in [1, 4]$, and samples (n, k, Q, H) from four configurations: $(8, 3, 60, 240)$, $(10, 3, 70, 260)$, $(10, 4, 60, 260)$, and $(12, 4, 80, 300)$. The pickup-delivery time-window subfamily uses one vehicle and a longer

time horizon, with $n \in \{4, 6, 8, 10, 12\}$; pairing is supplied by the variant rules. The compartment subfamily uses a split-cluster layout and samples (n, k, Q, C) from four configurations: $(8, 3, 24, 2)$, $(10, 3, 26, 2)$, $(10, 4, 24, 2)$, and $(12, 4, 28, 2)$.

B.2 Variant-Specific Parameters

After a profile determines (n, k) , the spatial distribution, and the base demand, the variant generator generates the additional parameters required by each constraint module. Unless otherwise specified, continuous values are sampled within the reported intervals and discrete values are selected from the reported sets.

Capacity. For homogeneous-capacity variants, effective vehicle capacities lie within the envelope $[8, 100]$. The capacity-boundary family does not sample from the default envelope; it directly uses the (n, k, Q, d) grid from Appendix B.1, making the average route demand close to the capacity bound. TSP has no effective capacity constraint and assigns zero demand to nodes. SDVRP uses split-delivery demand, with scalar demand in $[2, 20]$ and vehicle capacity in $[30, 85]$. MCVRP, HFMCVRP, and MCVRPSTW use two compartments. Each customer has per-compartment demand sampled from $[0, 6]$; per-compartment capacity is obtained by splitting vehicle capacity across compartments and applying a mild perturbation, yielding values within the range $[5, 55]$. The total vehicle capacity equals the sum of compartment capacities.

Heterogeneous fleets. HFVRP, HFVRPHTW, HFMDVRP, and HFMCVRP use three vehicle types. Type-specific capacities, fixed costs, and per-distance costs fall within $[10, 65]$, $[20, 70]$, and $[0.8, 1.2]$, respectively. This covers small, medium, and large vehicles, requiring the model to index both capacity and cost by vehicle type rather than reducing the instance to a homogeneous fleet. In HFMCVRP, the three total base capacities are $\{15, 30, 50\}$ and then split into two compartments within each type.

Route-distance limits. DCVRP and DCVRPHTW use route-level distance limits $D_{\max}$ within the range $[60, 1000]$. When a distance-boundary perturbation is constructed, $D_{\max}$ is tightened by the operator according to the route lengths of s^+ and the corresponding solution s_i^- .

#	Profile	n	k	Spatial distribution	Demand d	Q
1	Tiny uniform	5	2	Uniform	[2, 10]	—
2	Tiny clustered	6	2	Clustered	[1, 10]	—
3	Small radial	7	2	Radial	[2, 8]	—
4	Small elongated	7	3	Elongated	[1, 10]	—
5	Medium uniform	8	3	Uniform	[1, 10]	—
6	Medium clustered	9	3	Clustered	[2, 12]	—
7	Medium split	10	3	Split	[1, 8]	—
8	Medium light	8	3	Uniform	[0.5, 3]	—
9	Medium heavy	9	4	Uniform	[5, 15]	—
10	Large uniform	12	4	Uniform	[1, 8]	—
11	Large clustered	12	4	Clustered	[2, 10]	—
12	Large mixed	12	4	Elongated	[1, 12]	—
13	Geometry boundary	4–8	1–4	Deterministic geometry	Family-specific	Partially locked
14	Capacity boundary	5–12	1–4	Uniform / split clusters	Family-specific	Locked
15	Time / multi-resource boundary	4–12	1–4	Radial / split clusters	Family-specific	Locked

Table 5: Compact profile definitions. The demand column gives the default scalar demand range. Zero scalar demand for TSP, pickup-delivery demand, and SDVRP split-delivery demand are supplied by the variant-specific rules. The Q column indicates only whether capacity-related values are fixed at the profile level; the time horizon H and compartment count C are specified only in the family descriptions for profile 15.

Time windows. Regular time-window variants use $H = 500$, except PDPTW, which uses a chain-length-dependent horizon $H = \max(1200, 120p + 600)$, where p is the number of pickup-delivery pairs, yielding $H = 1200$ for $p \leq 5$ and $H = 1320$ for $p = 6$. After resource relaxation, customer window endpoints remain within $[0, 1500]$. Depot windows are $[0, H]$. Customer service times lie within the effective envelope $[0, 10]$: PDPTW uses zero service time, while the remaining time-window variants sample positive values, with DCVRPHTW allowing values up to 10. Let d_0 be the Euclidean distance from a customer to the main depot. Customer window starts are sampled from intervals proportional to d_0 , lie in $[0, 250]$, and have widths in $[30, 120]$. For MDVRPSTW, d_0 is instead the distance to the nearest depot when generating each customer’s soft time window; this does not assign the customer to that depot. During initial instance generation, PDPSTW and PDPTW first sample pickup windows and then derive the paired delivery windows. PDPSTW initializes each delivery-window start as the pickup start plus pickup service and 0.5 times the direct pickup-delivery distance, whereas PDPTW uses the pickup start plus 0.3 times that distance with zero service time. These windows may subsequently be widened during feasibility-preserving relaxation. The time-boundary family directly uses the (n, k, Q, H) grid in Appendix B.1.

Depots, pickup-delivery pairs, backhauls, and open routes. MDVRP, MDVRPSTW, and

HFMDVRP use two depots. PDP and PDPSTW set scalar node demand to zero and sample paired pickup-delivery demand from $[1, 8]$, whereas PDPTW uses unit demand for every pair. All three construct pickup-delivery pairs; their customer counts are even and lie in $\{4, 6, 8, 10, 12\}$. VRPB labels customers as linehaul or backhaul and requires linehaul-first service. OVRP and OVRPHTW allow routes to terminate at the last customer without counting a return-to-depot leg.

B.3 Feasible Solution Construction

For each complete instance, we first run a variant-specific heuristic to obtain s^+ . The heuristic is mainly based on nearest-neighbor insertion and checks common VRP resource constraints, such as capacity, route-distance limits, and time windows, at each step. If customers remain uninserted, they are preferentially inserted into the route with the largest remaining slack. If the heuristic fails, we invoke the reference concrete code C_{con} for a time-limited feasibility solve with $\text{TimeLimit} = 15$ seconds and $\text{MIPFocus} = 1$. The returned solution is then checked by C_{con} through a feasibility assertion. If any assertion fails, the candidate sample is discarded. The resulting s^+ is used to construct the one-constraint-violating solutions $\{s_i^-\}$.

B.4 One-Constraint-Violating Solution Construction

Each variant v is associated with a perturbation-operator set $\mathcal{A}_v = \mathcal{A}_v^{\text{struct}} \cup \mathcal{A}_v^{\text{param}}$. Each operator transforms s^+ into a one-constraint-violating so-

lution s_i^- that targets one active constraint family. Structural perturbations change only the customer visiting sequence and do not modify instance parameters. Parameter perturbations change the visiting sequence and tighten one family of resource bounds. We construct these solutions for logical structure constraints such as customer coverage, depot connectivity, and pickup–delivery precedence, and for common resource constraints such as capacity, distance, compartments, and hard time windows. Constraints that mainly affect objective penalties rather than producing deterministic infeasibility, such as soft time-window penalties, are handled by the optimal value check.

The perturbation operators only generate candidate solutions. The final labels are assigned by checking each candidate instance–solution pair with C_{con} . Each s_i^- is derived from s^+ by one operator and targets one constraint family by construction. If s^+ is infeasible on the corresponding instance, or if any s_i^- is not rejected by C_{con} , the candidate is discarded.

Each s_i^- is injected separately into the complete generated model, while all other generated constraints remain active. Combining several violations in one solution would make the verdict non-diagnostic, because the model could reject that solution by correctly enforcing any one of the violated constraints while still omitting another. We therefore require $C_{\text{con}}^{\text{gen}}$ to accept s^+ and reject every s_i^- separately. This design isolates individual constraint-family errors but does not exhaustively test errors that emerge only through interactions among multiple constraints.

SDVRP is handled separately because split delivery changes the semantics of coverage, subtours, and capacity. A customer can be served by multiple vehicles; removing a customer must remove all of its delivered portions; a subtour customer may otherwise be reconnected to a depot through another route; and overload can be repaired by redistributing demand across vehicles. Therefore, SDVRP uses split-delivery versions of the remove-customer and isolated-subtour perturbations, together with two capacity-oriented perturbations: quarantine overload and phantom delivery. Repeatedly visiting the same customer is not necessarily a violation under split delivery and is therefore not used to construct an s_i^- .

B.4.1 Structural Perturbations

Remove customer targets coverage or demand-satisfaction constraints. For standard non-split variants, every customer $c \in N$ must be visited exactly once, e.g., $\sum_{i,k} x_{i,c,k} = 1$. The operator selects a visited customer from s^+ and removes it from its route; for example, $[0, 1, 2, 0]$ becomes $[0, 1, 0]$, leaving customer 2 unserved. In SDVRP, where a customer may be split across vehicles, the selected customer is removed from every route that serves any portion of it. If $C_{\text{con}}^{\text{gen}}$ omits the coverage/demand-satisfaction constraint, relaxes it to an inequality, or covers only a subset of customers, the resulting s_i^- may be accepted.

Subtour cycle targets depot connectivity constraints: every route must be connected through a depot and cannot contain a customer-only cycle. The operator extracts at least two customers from a feasible route and forms a cycle that does not pass through the depot, e.g., a detached cycle such as $[1, 2, 1]$. If $C_{\text{con}}^{\text{gen}}$ omits subtour elimination or only enforces local flow balance, the detached cycle may be accepted. SDVRP uses an isolated-subtour solution, which additionally removes the cycle customers from other routes so that split service cannot reconnect them to the depot.

Backhaul order violation targets the VRPB linehaul-first rule: all linehaul customers must precede all backhaul customers on each route, equivalently forbidding any transition from a backhaul customer to a linehaul customer. The operator moves a backhaul customer before a linehaul customer, e.g., changing $[0, l_1, l_2, b_1, 0]$ into $[0, l_1, b_1, l_2, 0]$. The optimal value check alone cannot verify that this rule is encoded, because the optimum in an enlarged feasible region may not use a reversed edge.

Precedence violation targets pickup–delivery precedence. All pickup–delivery variants require each pickup to precede its paired delivery on the same route. PDPTW enforces this through service-start times, whereas PDP and PDPSTW use route-order variables. The operator swaps the pickup and delivery within the same route, e.g., $0, p, d, 0$ becomes $0, d, p, 0$.

Split pickup–delivery pair targets same-vehicle coupling for pickup–delivery pairs: each pair (p, d) must be served by the same vehicle, expressible as $\sum_j x_{p,j,k} = \sum_j x_{d,j,k}, \forall k$. The operator keeps

the pickup on the original vehicle and moves the delivery into another vehicle route. It exposes models that encode precedence but omit same-vehicle coupling.

B.4.2 Parameter Perturbations

Parameter perturbations construct a boundary case for one resource. Let g be the resource usage of the target route, vehicle, or compartment, and let b be the corresponding resource usage of a perturbed solution s_i^- , with $b > g$. Let $\tilde{B}$ be the tightened resource bound in the perturbed instance, such as capacity $\tilde{Q}$, route-distance limit $\tilde{D}_{\max}$, or compartment bound $\tilde{Q}_c$. If $\tilde{B} \leq g$, the feasible solution is incorrectly excluded; if $\tilde{B} \geq b$, the one-constraint-violating solution no longer violates the resource. Thus, except for time-window violations, parameter perturbations set $\tilde{B}$ inside the open interval (g, b) :

$$\tilde{B} = \max(g + \varepsilon, \min(g + 0.85(b - g), b - \varepsilon)). \quad (19)$$

Here $\varepsilon > 0$ is a small numerical margin ensuring that $\tilde{B}$ is strictly between g and b . The 0.85 interpolation places the one-constraint-violating solution close to the boundary, testing whether the regenerated model truly encodes that upper bound. Clipping prevents numerical rounding from moving $\tilde{B}$ onto g or b . SDVRP uses a more conservative margin because split-delivery quantities are more sensitive to rounding. The time-window violation operator does not use this scalar bound-tightening rule, because it swaps the visiting order to create hard time-window lateness.

Capacity overload targets capacity constraints $\sum_i d_i \sum_j x_{i,j,k} \leq Q$ for all vehicles k . It moves one or more customers from other routes into a target route until the target route exceeds the tightened capacity. We set g to the target route's original load under s^+ and b to the load of the overloaded target route. We then tighten Q , or the target vehicle's capacity, to a value in (g, b) . For homogeneous fleets, the tightened bound is additionally kept above the largest route load in s^+ ; for vehicle-specific bounds, it is applied only to the target vehicle. The resulting s_i^- checks whether capacity is present and applied per vehicle. VRPB computes linehaul and backhaul loads separately and uses their maximum. PDP and PDPSTW move whole pickup-delivery pairs, and rebuild the target route in pickup-before-delivery order, to

avoid confounding capacity with same-vehicle or precedence violations.

Heterogeneous fleet overload targets vehicle-specific capacity constraints, where each vehicle has its own capacity and cost parameters. For HFVRP and HFVRPHTW, we construct an extreme-overload solution: customers are greedily accumulated into a single route until their total demand exceeds the maximum capacity of any vehicle, making the route infeasible under any re-assignment. If this construction fails, the implementation falls back to the generic move-overload perturbation. For heterogeneous multi-depot or multi-compartment variants, the generic overload operator tightens only the target vehicle's capacity, while compartment-specific variants additionally use compartment-overload solutions. These perturbations test whether $C_{\text{con}}^{\text{gen}}$ respects vehicle-specific capacities rather than replacing them with a single homogeneous bound.

Distance overrun targets route-distance limits $\sum_{i,j} c_{i,j} x_{i,j,k} \leq D_{\max}$ for all vehicles k . It moves customers into the longest route in s^+ so that the perturbed route length b exceeds the largest route length in s^+ , denoted by g , and then tightens $D_{\max}$ to a value in (g, b) . It tests whether the distance limit is enforced separately for each vehicle and computed from all traveled legs of its route.

Compartment overload targets compartment capacity constraints $\sum_i d_{i,c} \sum_j x_{i,j,k} \leq Q_c$ for each vehicle k and compartment c . Multi-compartment routing is not captured by total vehicle load: one compartment can overflow while the total load still appears feasible. The operator concentrates demand from one compartment on a vehicle and tightens only that compartment bound while keeping total vehicle capacity non-binding.

Quarantine overload targets the SDVRP per-vehicle capacity constraint $\sum_i d_i^k \leq Q$ for every vehicle k , where d_i^k is the portion of customer i served by vehicle k . Ordinary capacity perturbations can be repaired in SDVRP by redistributing demand. This operator fixes the service assignment of the target customers so the diagnostic question becomes whether the model still enforces the capacity bound for that vehicle.

Phantom delivery targets the coupling between vehicle visits and delivered quantities in SDVRP.

Split delivery models commonly use delivery-allocation variables to indicate how much demand of customer i is served by vehicle k , and these quantities must be coupled to whether vehicle k actually visits customer i . The operator grafts the customers of one active vehicle onto another capacity-tightened vehicle and deactivates the original vehicle’s route. The resulting s_i^- is infeasible when visit–delivery coupling is enforced, because the tightened vehicle cannot carry the combined demand. If the regenerated model allows a deactivated vehicle to deliver positive demand without visiting the customer, the same solution can be incorrectly accepted. This exposes models that encode aggregate customer demand satisfaction but omit the coupling between routing variables and split-delivery quantities.

Time window violation targets hard time window constraints $a_i \leq \tau_i^k \leq b_i$. It swaps adjacent customers in a feasible route while preserving the customer set, vehicle assignment, and load, and then sets narrow windows around the legal arrival times in the original route. The resulting s_i^- is retained only if the swapped route violates at least one hard time window under the same windows. For PDPTW, the operator is precedence-aware and avoids swaps that directly invert a pickup-delivery pair, so that the violation remains attributable to time windows rather than precedence. Soft-time-window variants do not use this operator because time-window violations are penalized rather than infeasible.

B.5 Instance Relaxation and Assembly

Resource-bound perturbations require an open interval between the resource usage of the feasible solution s^+ and that of the corresponding one-constraint-violating solution s_i^- . After obtaining s^+ , we therefore apply one-way relaxation to the bounded resource families present in each variant, such as vehicle capacity, route-distance limits, and time windows. Capacities and route-distance limits are widened relative to the load or route length of s^+ , while time-window variants receive additional slack around the service times induced by s^+ . This step only enlarges the feasible region, so s^+ remains feasible while later operator-specific tightening has room to place the target bound between the usage of s^+ and s_i^- .

Let I^* denote the relaxed instance used as the sample instance before perturbation-specific tight-

ening. Structural perturbations are evaluated directly on I^* , since their violations are induced by route structure rather than numerical resource bounds. For each resource-bound operator a , we create a perturbation-specific copy I_a^* and tighten only the intended resource bound on that copy. Non-target resources may be adjusted only to avoid confounding violations, ensuring that s_i^- targets the intended constraint family. We retain a labeled solution only when C_{con} verifies s^+ as feasible and the corresponding s_i^- as infeasible on the same instance: I^* for structural perturbations and I_a^* for resource-bound perturbations. For each resource-bound sample, its natural-language description and reference concrete code are generated from the same I_a^* .

C Natural-Language Rewriting and Code Regeneration

This appendix describes Stage 2 of the data synthesis pipeline. We use Gemini-3.1-Pro Preview (Google DeepMind, 2026) and Claude Opus 4.6 (Anthropic, 2026) as the teacher LLMs in the synthesis pipeline. Within Stage 2, the teacher first rewrites the reference concrete code into natural-language descriptions and then regenerates $C_{\text{con}}^{\text{gen}}$ from each description.

C.1 Natural-Language Rewriting

Scenario instantiation. The main path rewrites C_{con} into a natural-language problem statement p_0 organized into business background, resources, constraints, and objective. It uses a generate–critique–repair loop: the teacher first produces a draft, a critic then checks numerical completeness, constraint coverage, objective direction, absence of fabrication, ID–attribute binding, and unit plausibility, and a repair call edits only the parts identified as inconsistent. Each call is given a deterministic scene–style hint, described below.

Condensation. The first augmentation path removes solver-side or formulation-side wording from p_0 while preserving the optimization problem. Typical deletions include self-loop prohibitions, explicit in-degree or out-degree formulations, subtour-elimination wording, and explanatory references to virtual depots or sink nodes when they are only modeling devices. Two prompt variants are used: a permissive version that allows such deletions, and a stricter version that forces virtual-node wording to be replaced by natu-

Variant	Structural perturbations	Parameter perturbations
TSP	Base pair	–
CVRP	Base pair	Capacity overload
MDVRP	Base pair	Capacity overload
MCVRP	Base pair	Capacity overload; compartment overload
PDP	Base pair; precedence violation; split pickup-delivery pair	Capacity overload
OVRP	Base pair	Capacity overload
DCVRP	Base pair	Capacity overload; distance overrun
VRPB	Base pair; backhaul order violation	Capacity overload
SDVRP	Split delivery base pair	Quarantine overload; phantom delivery
VRPHTW	Base pair	Capacity overload; time window violation
VRPSTW	Base pair	Capacity overload
HFVRP	Base pair	Capacity overload; heterogeneous fleet overload
HFMDVRP	Base pair	Capacity overload
HFMCVRP	Base pair	Capacity overload; compartment overload
HFVRPHTW	Base pair	Capacity overload; heterogeneous fleet overload; time window violation
MDVRPSTW	Base pair	Capacity overload
PDPSTW	Base pair; precedence violation; split pickup-delivery pair	Capacity overload
PDPTW	Base pair; precedence violation	Time window violation
<i>OVRPHTW</i>	Base pair	Capacity overload; time window violation
<i>MCVRPSTW</i>	Base pair	Capacity overload; compartment overload
<i>DCVRPHTW</i>	Base pair	Capacity overload; distance overrun; time window violation

Table 6: Perturbation operators enabled for the 21 variants. Base pair denotes the remove-customer and subtour-cycle perturbations. Split-delivery base pair denotes the SDVRP-specific remove-customer and isolated-subtour perturbations. Structural perturbations modify only the visiting sequence, whereas parameter perturbations additionally tighten one target resource family.

ral route-termination statements. Outputs that fail static fidelity checks are repaired through minimal local edits.

Index rewriting. The second augmentation path changes node identifiers in p_0 or p_{cond} while preserving all other semantics. The target style is deterministically selected from three options: consecutive one-based integer identifiers, consecutive integer identifiers starting from 2, or uppercase letter identifiers. The rewrite call must also return a one-to-one node_id_map. This map is required downstream because s^+ and $\{s_i^-\}$ are constructed on I^* using source node IDs; the verifier must translate the reference solutions into the rewritten naming space before applying Inj.

Scene and style configuration. Scenario instantiation appends a deterministic scene–style hint to the rewriting prompt. The scene pool contains 60 entries and the style pool contains three entries. Both are selected from a seed derived from the sample triple (variant, seed, profile), so rerunning the same sample yields the same scene and style. The template contains no additional random perturbation. The unit hints are drawn from

a small set of base unit families, including tons, kilograms, liters, kWh, seat-kilometers, and order-equivalents, sometimes with domain-specific modifiers. The three style instructions are: use a concise academic-problem style similar to a benchmark or methods paper; use a formal optimization-problem style with stable ordering from background to data, constraints, and objective; and use precise conservative wording that avoids storytelling and remains easy to parse into a mathematical model.

C.2 Code Regeneration

Each problem description $p \in \{p_0, p_{\text{cond}}, p_{\text{idx}}\}$ is then sent to the code-regeneration prompt to obtain $C_{\text{con}}^{\text{gen}}$. The same code-generation instruction template and output protocol are used when constructing SFT examples, during GRPO rollouts, and in final evaluation.

D Constraint-Injection Encoding Details

This appendix provides implementation details for the Inj check introduced in Section 4.2. Given $(C_{\text{con}}^{\text{gen}}, s)$, the verifier executes $C_{\text{con}}^{\text{gen}}$ to obtain the Gurobi model, copies the model, replaces the ob-

jective with a constant, and appends the solution-encoding constraints $\Phi(s)$. The resulting model is solved only as a feasibility query; its objective value is ignored.

Variable parsing and index alignment. The verifier searches for routing variables named x , following the output protocol in Appendix H. It supports both two-dimensional arc variables and three-dimensional arc-vehicle variables, provided that their indices follow the arc-first convention specified by the output protocol. For index-rewritten statements, the recorded `node_id_map` is used to translate the routes in s back to the internal node indices expected by $C_{\text{con}}^{\text{gen}}$. If the routing variables or node indices cannot be matched reliably, the case is not used as a valid Inj verdict and the sample fails rejection sampling.

Homogeneous-fleet injection. For homogeneous fleets, s^+ is encoded through a vehicle-agnostic customer-customer projection: customer-customer arcs appearing in s^+ are required, customer-customer arcs absent from s^+ are forbidden, customer self-loops are blocked, and intentionally unvisited customers have their incoming arcs blocked. Vehicle identities remain free. For each s_i^- , the verifier additionally applies the route-level consistency constraints in Eq. (12), so that the customer-customer edges of the same diagnostic route cannot be distributed across interchangeable vehicles.

Heterogeneous-fleet injection. For heterogeneous fleets, the verifier uses the vehicle-specific fixing described in Section 4.2. This preserves the vehicle assignment carried by the injected solution, which is necessary when capacities, costs, or resource bounds depend on vehicle type. In this case, both s^+ and $\{s_i^-\}$ are injected with vehicle identities fixed.

Depot edges. The main injection encoding focuses on customer-customer transitions and does not use depot-departure or depot-return patterns as primary evidence for the injected solution. It therefore does not directly test every depot-structure constraint; such errors are detected by the optimal value check only when they change the optimum. For a small number of one-constraint-violating solutions, the verifier adds solution-specific depot-edge restrictions to prevent degenerate repairs, such as serving an intentionally unvisited customer outside the encoded route structure or reas-

signing an isolated SDVRP customer to another vehicle. These restrictions preserve the intended solution semantics and are not used as a general test of depot-structure constraints.

Split-delivery solutions. For SDVRP, the verifier must preserve the service assignment in the injected solution because customer demand may be split across vehicles. The SDVRP-specific one-constraint-violating solutions in Appendix B.4, including quarantine overload and phantom delivery, therefore prevent the violation from being repaired by redistributing the same customer’s demand to another vehicle. This keeps the Inj query aligned with the targeted capacity or visit-delivery coupling constraint.

Verdict handling. The feasible solution s^+ passes Inj when the solution-encoded model is feasible. A one-constraint-violating solution s_i^- passes Inj when the solution-encoded model is infeasible. Cases with model-construction failures, unsupported variable formats, unreliable node matching, or unresolved solver statuses are treated as verification failures during rejection sampling.

E Training Details

Implementation environment. All optimization solves in data synthesis, verification, training-time reward computation, and evaluation are conducted with Gurobi Optimizer under an academic license. The main SFT and GRPO runs share one LoRA adapter configuration with rank $r = 16$, $\alpha = 32$, and dropout 0.1. All model calls use the shared code-regeneration system prompt in Appendix H.

SFT. We apply LoRA SFT to Qwen3-8B on 6,797 $(p, C_{\text{con}}^{\text{gen}})$ pairs. Training uses four A100-40GB GPUs, per-device batch size 1, gradient accumulation 2, and global effective batch size 8. We train for 3 epochs with sequence length 8192. The optimizer is AdamW with learning rate 2×10^{-4} , cosine scheduling, warmup ratio 0.1, weight decay 0.01, and maximum gradient norm 1.0.

Frontier filtering. For each prompt in the SFT pool, we sample $H = 6$ offline rollouts from VRPCoder-SFT under the same decoding settings as online GRPO (temperature 0.4, top- p 0.95), score each rollout with the dual verifier under the GRPO reward weights, and retain only prompts

whose within-group reward standard deviation is strictly positive. This yields the 716 frontier prompts used in the main GRPO run.

GRPO. Starting from the SFT checkpoint, we train on 716 frontier prompts. Policy training uses three A100-40GB GPUs with DeepSpeed ZeRO-2 (Rasley et al., 2020), while one additional A100-40GB GPU runs a vLLM (Kwon et al., 2023) server for rollouts. The per-device batch size is 1, gradient accumulation is 8, and the global effective batch size is 24. We train for 3 epochs. For each prompt, we sample $G = 8$ rollouts with a maximum completion length of 3072, temperature 0.4, top- p 0.95, and KL coefficient $\beta = 0.02$. The optimizer is AdamW with $\beta_2 = 0.95$, learning rate 2×10^{-5} , cosine scheduling, 5 warmup steps, weight decay 0.01, and maximum gradient norm 0.5.

No-injection ablation. For the no-injection ablation, we retrain the SFT checkpoint on 7,347 samples. Compared with the main 6,797-sample SFT set, the additional 550 samples pass the optimal value check but fail constraint injection and are therefore rejected by the main pipeline. All other SFT hyperparameters are unchanged. For GRPO, we remove r_{inj} from the reward and keep only build success and the optimal value check, so the maximum reward becomes 0.7 under the original weights. We then generate new rollouts from the 7,347-sample no-injection SFT checkpoint and apply the same positive-variance frontier filtering without injection verification, yielding 855 frontier prompts. Decoding, learning rate, scheduling, LoRA configuration, and effective batch size remain aligned with the main GRPO run.

F Constraint-Level Error Analysis

Across several rounds of synthesis and regeneration, 7,347 candidates passed the optimal value check, among which 550 (7.49%) failed constraint injection. Among the 450 silent-omission cases, the failures were associated with the base structural pair (268), hard time windows (56), backhaul ordering (44), capacity (30), distance limits (24), compartment capacity (21), precedence (5), and split pickup-delivery coupling (2). Here, the base structural pair denotes the remove-customer and subtour-cycle solutions. Among the 100 spurious-over-constraint cases, the failures involved distance limits (40), vehicle capacity (38), hard time

Source or slice	Group	N	Pass@1 (%)
Subsets 1–2	0 active modules	25	100.00
Subsets 1–2	1 active module	20	100.00
Subsets 1–2	2 active modules	191	95.81
Subsets 1–2	3 active modules	140	95.71
TSPLIB	TSP	16	100.00
CVRPLIB	CVRP	15	100.00
Solomon	VRPTW	11	81.82
Cordeau-MDVRP	MDVRP	7	85.71
SDVRPLIB	SDVRP	13	92.31
VRPB	VRPB	12	91.67
Cleaned SIRL	TSP/VRP	50	100.00
NLCO	TSP	50	92.00
NLCO	CVRP	50	90.00
NLCO	TSPTW [†]	50	74.00
NLCO	PDPTW	50	84.00

Table 7: VRPCoder-RL Pass@1 by active-module count and evaluation source or task. [†] denotes a variant absent from training.

windows (21), and compartment capacity (1). To examine whether these synthesis-time patterns persist after training, Table 7 reports final performance by active-module count and evaluation source or task.

Hard time windows appear in both synthesis failure modes and remain challenging in final evaluation: VRPCoder-RL obtains 81.82% Pass@1 on Solomon VRPTW and 74.00% on the unseen NLCO TSPTW task. Aggregating Subsets 1 and 2 by variant complexity, VRPCoder-RL achieves 95.81% and 95.71% Pass@1 on variants with two and three active constraint modules, respectively.

G Evaluation Benchmarks and Protocol

VRPBench comprises four subsets with 700 total instances, ranging from in-domain synthesis to externally sourced problems. All problem statements and reference answers are manually verified by operations-research experts.

Subset 1: In-domain synthetic. This subset contains 310 instances generated by the three-stage data synthesis pipeline on the 18 training variants with a separate set of random seeds. Samples are retained after the same dual verifier procedure used for training data.

Subset 2: Out-of-domain synthetic. This subset contains 66 instances generated by the same pipeline on the three held-out variants OVRPHTW, MCVRPSTW, and DCVRPHTW. The individual constraint modules in these variants appear in training, but the corresponding pairwise combina-

tions are never observed during training, making this benchmark a test of compositional generalization.

Subset 3: Public OR libraries. This subset contains 74 instances derived from public OR libraries, including TSPLIB (Reinelt, 1991), CVRPLIB (Uchoa et al., 2017), Solomon (1987), Cordeau-MDVRP (Cordeau et al., 1997), SDVRPLIB (Belenguer et al., 2000), and VRPB (Toth and Vigo, 1997). It covers TSP, CVRP, VRPTW, MDVRP, SDVRP, and VRPB. Since the original public instances often contain tens to hundreds of customers, we extract 5–12-customer subinstances while preserving real coordinates, demands, time windows, multi-depot structure, split delivery, or backhaul structure when present. OR experts then write the natural-language statements after down-sampling, ensuring that all numerical and structural constraints remain consistent.

Subset 4: Externally sourced. This subset contains 250 instances. The first 200 come from the routing subset of NLCO (Jiang et al., 2026), covering TSP, CVRP, TSPTW, and PDPTW with 50 small instances per type. We make two lightweight adaptations: removing the JSON output-format and reasoning instructions from the original prompts, and adding the annotated vehicle count to CVRP statements because code-generation tasks require the fleet size to be explicit. The surface format and indexing scheme of NLCO are otherwise preserved. The remaining 50 instances come from the cleaned SIRL release (Chen et al., 2025), which aggregates OptMATH (Lu et al., 2025), IndustryOR (Huang et al., 2025a), and MAMO (Huang et al., 2025b); we extract all TSP and VRP variants, consisting of 9 OptMATH, 2 IndustryOR, and 39 MAMO ComplexLP cases.

Evaluation. All four subsets use instance-level Pass@1 with absolute objective tolerance 10^{-3} , a 600-second Gurobi time limit per instance, and greedy decoding. Trained models use the shared code-regeneration system prompt in Figure 11; OR-LLM baselines use their original prompts and solver settings.

H Prompts Used in Experiments

This appendix collects all prompt texts used in the paper. Placeholders in braces are filled by the data-generation scheduler at runtime.

Scenario Instantiation: System Prompt

You are an operations research expert. Given Gurobi solver code, produce a formal English business problem description suitable as code-generation training data.

Data fidelity:

1. Preserve ALL numerical data exactly (coordinates, demands, capacities, fleet size, time windows, service times, max distances, etc.). Do not omit, invent, or round any number.
2. State the optimization objective and every constraint in business language. No math symbols, variable names, or code jargon.
3. Index preservation (critical): every vehicle, customer, and depot must carry its explicit numeric ID from the code. Write "Truck 0 (capacity 50.0, fixed cost 70.0)" — never describe entities by type alone without their ID. Never renumber or reorder.

Unit consistency:

4. Demand/capacity values may be fractional (e.g. 3.4). Use continuous units only (tons, kilograms, liters). Never use bare countable nouns (pallets, boxes, items) — "3.4 pallets" is nonsensical.
5. If the problem has time-window or scheduling constraints, do NOT also use time-based units for demand/capacity — this creates a logical contradiction. Use a tangible goods unit instead.

Style:

6. Structure: background → resources/data → operational constraints → objective.
7. Plain prose only. No Markdown, no bullets/tables/dialogue/email tone.
8. If a scene hint is given, adapt terminology naturally while preserving the optimization structure.

Scenario Instantiation: User Prompt

Generate a natural-language business problem statement from the code below.

===== Scene and Style Instructions =====

{scene_hint}

===== Gurobi Code (with complete data) =====

{code}

===== Critical Numbers That Must Appear =====

Embed each number naturally in context (e.g. "a capacity of 50.0"), do not list them as a bare table:

{critical_numbers_str}

Output the problem description directly:

Figure 6: Scenario Instantiation Prompts.

Self-Critique: Critic System Prompt

You are a strict reviewer. Compare the Gurobi code with the generated NL description for full consistency.

Check these dimensions:

1. Numerical completeness: all critical values (depot/customer/vehicle counts, capacities, demands, coordinates, time windows, etc.) present?
2. Constraint coverage: every code constraint has a corresponding business description?
3. Objective: optimization direction correctly stated?
4. No fabrication: no numbers or conditions invented beyond the code?
5. Index–attribute binding: every entity carries its numeric ID, and ID–to–attribute mappings (ID → capacity, cost, type) are faithful to the code? Missing or ambiguous IDs = INCOMPLETE.
6. Unit plausibility: demand/capacity units are continuous (not countable nouns) and do not conflict with time–window dimensions?

Output exactly one of:

- COMPLETE
- INCOMPLETE: <discrepancy_1> | <discrepancy_2> | ...

Self-Critique: Critic User Prompt

Please compare the following content and determine whether the natural–language description fully and accurately reflects the optimization problem in the code:

===== Gurobi Code (with complete data) =====

{code}

===== Generated Natural–Language Description =====

{nl_description}

Please provide the evaluation result:

Repair: System Prompt

You are an operations research expert.

Revise the natural–language problem description based on the review feedback so that it is fully consistent with the original mathematical problem.

Rules:

1. Only modify the parts identified as problematic.
2. Preserve the correct and fluent parts of the original description.
3. Ensure all critical numbers appear accurately after revision.
4. Do not introduce new errors.

Repair: User Prompt

Review feedback: {criticism}

Original code and data: {code}

Current natural–language description: {nl_description}

Please output the revised complete natural–language description (only modify the parts identified as problematic; preserve the correct parts):

Figure 7: Self-Critique and Repair Prompts. Critic and repair prompts used in the generate–critique–repair loop.

Condensation: Shared System-Prompt Skeleton

You condense verbose vehicle–routing problem statements into shorter business–requirement descriptions. The rules are specified as follows:

1. Preserve every numeric value exactly as written. Do not delete, alter, round, merge, or restate numbers in a different form.
 2. Preserve every explicit entity ID exactly as written. Do not renumber, reorder, relabel, or collapse depots, customers, vehicles, compartments, pickup–delivery pairs, or any other named entities.
 3. Preserve every fact that changes the optimization problem: objective terms, route start/end behavior, depot assignment, fleet size, capacities, coordinates, demands, service times, time windows, maximum distance or duration, fixed or travel costs, heterogeneous vehicle attributes, compartment rules, pickup–delivery logic, backhaul ordering, split–delivery permission, and any other operational restriction.
 4. Keep the original business scene and terminology. This is condensation, not stylistic rewriting.
 5. <See variant–specific block below>
 6. Micro–examples of allowed compression: (1) "each node has exactly one incoming arc and exactly one outgoing arc" can usually be shortened to "each customer/location must be visited exactly once" if route start/end behavior is already clear elsewhere. (2) "no self–loops, no arcs into depot 0, no arcs out of sink 13" can be deleted if these are only solver–hygiene details and the route endpoints are already clear.
 7. Merge duplicated statements into one concise statement. Remove tutorial tone, repeated reminders, and repeated restatements of the same requirement.
 8. Prefer plain prose. Do not output bullets, markdown, JSON, XML, commentary, or surrounding quotation marks.
 9. If the source is already concise, make only small cuts.
- Output only the condensed problem statement.

Condensation: Rule 5 for Pilot 1

5. You MAY delete solver–side or formulation–side wording when it is not real business content. Typical deletable items include self–loop prohibitions, forbidden–arc wording such as "no arcs into the start depot" or "no arcs out of the sink", exact in–degree / out–degree phrasing, explicit subtour–elimination wording, and explanations of why a virtual depot or sink is introduced.

Condensation: Rule 5 for Pilot 2

5. You MUST delete ALL virtual–node wording. Specifically remove: (1) Any mention of "virtual depot", "virtual sink", "virtual return node", "dummy node" and their node IDs when used only as route endpoints. (2) Sentences like "Upon completing their routes, all trucks must proceed to Virtual Distribution center N". (3) Replace with natural route termination: "All vehicles return to the depot after completing deliveries" (closed routes) or "Vehicles do not need to return to the depot" (open routes). You may also delete other solver–side wording (self–loop prohibitions, forbidden–arc phrasing, explicit in–degree / out–degree constraints, subtour–elimination wording).

Condensation: User Prompt

Variant: {variant}; Condensation task: Turn the following source description into a shorter business–requirement description. Keep the same scene, terminology, numbers, IDs, objective, and operational constraints. You may compress duplicated explanations and solver–side wording such as self–loop or forbidden–arc phrasing when the business meaning is already clear. Do not introduce a new writing style.

Source description: {original_nl}

Output only the condensed problem statement.

Figure 8: Condensation prompts. The prompt template used for condensation, consisting of a shared system-prompt skeleton, pilot-specific Rule 5 instructions, and the corresponding user prompt.

Index Rewriting: System Prompt

Rewrite the VRP problem statement by replacing node/location identifiers only.

Rules:

1. First identify every node/location identifier that appears in the source text: depots, customers, pickup nodes, delivery nodes, virtual depots, sink nodes, endpoint nodes, start nodes, and end nodes.
2. Replace all of those node/location identifiers consistently using the requested target style.
3. Create the node-ID mapping yourself from the source text. Do not rely on any external mapping table.
4. Do not change vehicle IDs or any non-ID number.
5. Preserve all constraints and business meaning.
6. Do not add nodes or virtual-node details that are absent from the source.
7. Report the exact mapping you used in the JSON field `node_id_map`.
8. Do not output markdown, explanations, or commentary.

Output JSON only:

```
{
  "rewritten_nl": "...",
  "node_id_map": [
    {"source_id": "0", "target_id": "A", "role": "start_depot"},
    {"source_id": "7", "target_id": "B", "role": "virtual_end_depot"}
  ]
}
```

Index Rewriting: User Prompt

Variant: {variant}

Target style: {target_style}

Source description: {original_nl}

Rewrite the source by replacing every node/location identifier that appears in it. Include physical depots, customer nodes, pickup/delivery nodes, and virtual/sink/end depots if they appear. Decide the mapping yourself from the source text and keep it consistent. Preserve everything else. Return JSON only with `rewritten_nl` and `node_id_map`. `node_id_map` must list every source node/location identifier you replaced and the exact target identifier used in `rewritten_nl`.

Figure 9: Index Rewriting Prompts.

Index Rewriting Check: System Prompt

You check node–identifier rewrites for VRP problem statements.

Return JSON only with these fields:

```
{
  "passed": true/false,
  "missing_replacements": [],
  "leaked_source_ids": [],
  "added_absent_nodes": [],
  "mapping_issues": [],
  "issues": []
}
```

Compare the source statement, rewritten statement, and `node_id_map` directly. Check only node/location/depot/customer/pickup/delivery/virtual/sink/end identifiers. Do not treat vehicle IDs, coordinates, capacities, demands, costs, times, route counts, vehicle counts, or objective values as node IDs. If a source virtual/sink/end node is present, it must be replaced and included in `node_id_map`. If a virtual/sink/end node is absent from the source, it must not be added in the rewrite. The rewrite should consistently use the requested target style. `node_id_map` must describe the actual replacements used in `rewritten_nl`, must be one–to–one for node identifiers, and must not include vehicle IDs or non–node numeric values.

Index Rewriting Check: User Prompt

Target style: {target_style}

Source statement: {original_nl}

Rewritten statement: {rewritten_nl}

Reported node_id_map: {node_id_map}

Check whether every node/location identifier that appears in the source statement has been consistently rewritten in the rewritten statement and correctly recorded in `node_id_map`. Pay special attention to virtual depots, sink nodes, endpoint nodes, start depots, and end depots. Return JSON only.

Figure 10: Index Rewriting Check Prompts.

Code Regeneration: System Prompt

You are an operations research expert. Given a natural-language business problem, generate complete Gurobi Python solver code.

Output ONLY code wrapped in a Python fenced code block. No explanations, no commentary.

Code structure:

1. import gurobipy as gp and from gurobipy import GRB.
2. Define def build_model(): containing all model logic. It must call m.optimize() and return the model object.
3. Set m.setParam('OutputFlag', 0) before optimize.
4. Model ALL constraints described in the problem — omit none, fabricate none.
5. Use exact numerical values from the problem description.

Index convention:

- Use internal integer node indices 0..N-1 for routing variables, where N is the total number of routing nodes: depots, customers, pickup/delivery nodes, and explicit virtual/end nodes if present.
- If the problem already uses natural zero-based node IDs, use them directly as model indices. In this case, depot is usually node 0, customers are the following nodes, and no node ID mapping is needed.
- If the problem uses any other node ID scheme, such as one-based IDs or simple letter IDs, keep those original IDs only in raw data, then map them to internal indices with node_ids = [...], id2idx = {bid: i for i, bid in enumerate(node_ids)}, idx2id = {i: bid for bid, i in id2idx.items()}, and N = len(node_ids).
- Use input order as node_ids order. Convert every node-indexed attribute through id2idx when a mapping is used: depot IDs, customer IDs, coordinates, demands, service times, time windows, pickup-delivery pairs, virtual/sink depots, and start/end nodes.
- Use node IDs exactly as stated in the problem. Do not invent labels such as Customer A, Node A, Location A, Depot A, Outlet 7, or Hub 0.
- Index fidelity: preserve every ID-to-attribute mapping exactly as described, e.g., Truck 0 capacity 30 means vehicle 0 gets capacity 30. Never reorder or reassign.
- Vehicle IDs are not part of this identifier augmentation requirement; keep the vehicle convention 0..K-1 unless the problem explicitly specifies otherwise.

Variable format (critical for verification):

- Route variable MUST be named x with name='x' in addVars, indexed arc-first: x[i,j] or x[i,j,k] (i=origin, j=destination, then vehicle/depot). Never x[k,i,j] or x[(i,j),k].
- x MUST span the full node set, e.g., m.addVars(N, N, K, vtype=GRB.BINARY, name='x'). Do NOT pre-filter arcs with tuplelist; forbid invalid arcs via constraints instead.
- When a mapping is used, set N = len(node_ids) and define x over internal integer node indices. Do not use string/letter business IDs directly as x indices.

Code Regeneration: User Prompt

{nl_description}

Figure 11: Shared code-regeneration prompt used to produce $C_{\text{con}}^{\text{gen}}$ during data synthesis, to generate GRPO roll-outs, and to evaluate trained models.

ID	Scene	Depot term	Customer term	Vehicle term
1	Classical Distribution	distribution center	customer	vehicle
2	Retail Store Replenishment	warehouse	retail store	delivery truck
3	E-commerce Order Fulfillment	fulfillment center	delivery address	delivery van
4	Wholesale Grocery Supply	regional distribution hub	grocery outlet	supply truck
5	Cold-Chain Food Distribution	cold storage facility	supermarket	refrigerated truck
6	Dairy Product Delivery	dairy processing plant	convenience store	insulated vehicle
7	Urban Parcel Last-Mile Delivery	local sorting station	residential address	courier van
8	Meal Delivery Dispatch	cloud kitchen hub	diner location	delivery rider
9	Factory Parts Supply	parts warehouse	assembly workstation	AGV cart
10	Construction Material Delivery	building material yard	construction site	flatbed truck
11	Pharmaceutical Distribution	pharma distribution center	hospital pharmacy	medical transport vehicle
12	Home Healthcare Visiting	community health center	patient residence	nurse dispatch vehicle
13	Municipal Waste Collection	waste treatment plant	collection point	garbage truck
14	School Bus Routing	school campus	student pickup stop	school bus
15	Power Grid Maintenance	maintenance depot	substation	service crew vehicle
16	Telecom Equipment Repair	telecom operations center	base station site	field engineer van
17	Agricultural Produce Collection	produce purchasing station	farm	collection truck
18	Livestock Feed Distribution	feed mill	livestock farm	bulk feed truck
19	Disaster Relief Supply	relief staging area	shelter site	relief vehicle
20	Hazardous Waste Transport	licensed disposal facility	industrial generator site	hazmat transport vehicle
21	Urban Logistics Delivery	distribution center	delivery address	freight truck
22	Food Delivery Dispatch	delivery station	merchant/customer	delivery rider
23	Express Parcel Pickup	courier hub	pickup point	pickup van
24	Furniture Last-Mile Delivery	furniture warehouse	customer apartment	moving van
25	Laundry Pickup and Delivery	laundry processing center	hotel/household	laundry van
26	Fresh Food Cold Chain	cold storage warehouse	grocery supermarket	refrigerated truck
27	Seafood Cross-City Transport	seafood wholesale market	seafood restaurant	live fish transport truck
28	Bakery Product Distribution	central bakery	cafe/pastry shop	bread delivery van
29	Frozen Meal Delivery	frozen food warehouse	vending kiosk	freezer van
30	Agricultural Product Collection	produce purchasing station	farm plantation	farm transport vehicle
31	Flower Wholesale Distribution	flower auction center	florist shop	climate-controlled van
32	Vending Machine Restocking	restocking warehouse	vending machine location	restocking van
33	Bookstore Chain Replenishment	book distribution center	bookstore branch	delivery truck
34	Bike-Sharing Battery Swap	battery swap station	bike parking spot	maintenance dispatch vehicle
35	Ride-Hailing / Custom Bus	vehicle hub	boarding stop	shuttle bus
36	Drone Last-Mile Delivery	drone base station	community receiving tower	logistics drone
37	Autonomous Vehicle Delivery	AGV dispatch station	office building drop-off	AGV delivery robot
38	EV Charging Truck Dispatch	mobile charging depot	stranded EV location	mobile charger truck
39	Medical Supply Delivery	pharmaceutical distribution center	community clinic/hospital	medical cold chain vehicle
40	Blood Bank Distribution	central blood bank	surgical hospital	blood transport vehicle

Table 8: Scene hint pool, part I. Entries 1–40 of the 60-entry scene pool.

ID	Scene	Depot term	Customer term	Vehicle term
41	Mobile Home Nursing	community health center	patient household	visiting nurse
42	Mobile Vaccination Service	disease control center	temporary vaccination site	mobile medical vehicle
43	Medical Waste Collection	medical waste treatment plant	fever clinic	special collection truck
44	Organ Transport Coordination	transplant coordination center	recipient hospital	emergency medical vehicle
45	Municipal Waste Collection	landfill site	residential waste station	garbage truck
46	Snow Removal / De-icing	municipal works depot	major traffic intersection	snowplow
47	School Bus Service	school campus	student pickup point	school bus
48	Mobile Library Tour	city central library	remote community branch	mobile library van
49	Street Light Maintenance	city lighting depot	faulty lamppost location	maintenance bucket truck
50	Armored Cash Transport	vault center	bank branch/ATM	armored transport vehicle
51	Factory JIT Parts Delivery	line-side warehouse	assembly line station	AGV transport robot
52	Port Container Dispatch	container yard	berth/cargo ship	straddle carrier
53	Semiconductor Wafer Transport	cleanroom storage	fabrication tool bay	overhead transport robot
54	High-Voltage Grid Inspection	power maintenance base	transmission tower/substation	drone/inspection vehicle
55	Telecom Base Station Repair	telecom operations center	faulty base station	repair engineer
56	Wind Farm Turbine Maintenance	wind farm service center	wind turbine	maintenance crew vehicle
57	Offshore Oil Platform Supply	onshore supply base	offshore drilling platform	offshore supply vessel
58	Mine Ore Transport	ore processing plant	mining site	heavy-duty mining truck
59	Earthquake Relief Distribution	emergency relief tent/depot	displaced persons shelter	rescue helicopter/off-road vehicle
60	Forest Fire Patrol	forestry fire prevention center	key forest observation point	forest patrol vehicle

Table 9: Scene hint pool, part II. Entries 41–60 of the 60-entry scene pool.