Appendix 1 VRP Variants

We adopt the VRP experimental configuration established in DRoC [1]. As Table 1 is shown, the 48 VRP variants are combined by various combinations of fundamental constraints, such as service time and capacity limits. Considering the inherent difficulty in generating large-scale instances that guarantee feasible solutions, we use a representative instance for each variant created by human experts. To establish a reliable performance benchmark, we employ the Hybrid Genetic Search (HGS) algorithm [2] to obtain the ground truth for each instance. HGS is widely acknowledged in the operations research community for its superior ability to generate high-quality, near-optimal solutions for complex VRPs. However, some of the constraints like Pickup and Delivery and Resource Constraints are unsupported by HGS, so we employ OR-Tools as an alternative, setting a search time limit of 100 seconds to establish the ground truth. To enhance the complexity and informational density of the benchmark, we diversify the underlying graph representations by randomly assigning either a distance matrix or a time matrix to each instance. Following this principle, we meticulously curated our training dataset to align with the proposed framework while preserving the integrity of the original test suite from DRoC [1]. This consistency ensures a fair and direct performance comparison against existing state-of-the-art methods. A generated program is formally validated as correct only if its execution yields a solution value identical to the pre-determined optimal baseline. Crucially, these validated programs possess structural generalization capabilities. They are able to solve any instance within the same VRP class, remaining robust to variations in input parameters such as customer coordinates or demand distributions.

The input of the instance is composed by the name of the problem and the function signature. Under this setting, the LLM is required to infer the underlying constraints and objective logic solely from the semantic cues of the function signature. Once a generalized program for a specific VRP variant is synthesized, it functions as a robust solver template applicable to all instances within that category.

Appendix 2 Prompts

Prompts are classified into three functional groups, aligning with the three core actions of our framework. Each group is follows a consistent structure comprising three key components: role, task description, and output format.

- **Refine** The Refine action is designed to facilitate a self-correction mechanism, guiding the Executor to autonomously diagnose and rectify errors in the generated code.

Table 1. The 48 VRP tasks with 9 constraints.

VRP task	Capacity (C)	Open Route (O)	Distance Limit (L)	Service Time (S)	Time Window (TW)	Multi-Depot (MD)	Resource Const. (RC)	Prize Coll. (PC)	Pickup Delivery (PD)
CVRP	✓								
CVRPL	✓		✓						
CVRPTW	✓				✓				
CVRPMD	✓					✓			
CVRPTWL	✓		✓		✓				
CVRPMDL	✓		✓			✓			
CVRPTWMD	✓				✓	✓			
CVRPTWMDL	✓		✓		✓	✓			
CVRPTWRC	✓				✓		✓		
CVRPTWRCL	✓		✓		✓		✓		
OVRP		✓							
OVRPL		✓	✓						
OVRPTW		✓			✓				
OCVRP	✓	✓							
OCVRPL	✓	✓	✓						
OCVRPTW	✓	✓			✓				
PCTSP								✓	
PCTSPTW					✓			✓	
PCVRP	✓							✓	
PCVRPTW	✓				✓			✓	
PCVRPMD	✓					✓		✓	
PCVRPTWMD	✓				✓	✓		✓	
PDP									✓
PDPL			✓						✓
PDPTW					✓				✓
PDPMD						✓			✓
PDPTWL			✓		✓				✓
PDPTWMD					✓	✓			✓
PDPSL			✓	✓					✓
PDPTWS				✓	✓				✓
PDPTWSL			✓	✓	✓				✓
PDPTWMDL			✓		✓	✓			✓
TSP									
TSPTW					✓				
TSPTWS				✓	✓				
VRP	✓								
VRPL	✓		✓						
VRPMD	✓					✓			
VRPS	✓			✓					
VRPSL	✓		✓	✓					
VRPTW	✓				✓				
VRPTWL	✓		✓		✓				
VRPTWMD	✓				✓	✓			
VRPTWS	✓			✓	✓				
VRPTWMDL	✓		✓		✓	✓			
VRPTWSL	✓		✓	✓	✓				
VRPTWMRC	✓				✓	✓	✓		
VRPTWMRCL	✓		✓		✓	✓	✓		

Prompt for Refine

You are given a Python code snippet with errors, which may fail or exceed runtime limits (e.g., solver takes too long, constraints make problem infeasible).

The code snippet is as `<prep_code>`.

Your task is to reflect and generate a corrected, fully executable version. Follow these steps:

1. Restate the VRP problem requirements and constraints based on the given input.
2. Analyze whether the current code handles the constraints correctly. If correct, think about how to improve the solution.
   Optimization example:
   - Current code:
     `routing.AddDimension(transit_callback_index, 0, 30, True, 'Time')`
   - Improved code:
     `routing.AddDimension(transit_callback_index, 0, 3000, True, 'Time')`
   - Rationale: Increased max time per vehicle to allow feasible solutions.
3. Potential improvement directions include:
   - Adjusting dimension limits to allow feasible solutions
   - Optimizing time windows, vehicle capacities, or maximum route lengths
   - Improving callback function logic to enhance performance
   - Tuning solver parameters for faster or better-quality solutions
   - Ensure no undefined or unavailable functions from `{solver}` are used
   - Verifying that the correct functions are invoked according to the problem requirements
   - Check whether the code includes all the parameters. If not, add them
4. If the current code executes successfully, prioritize obtaining a feasible solution, and then optimize solver settings to move the solution closer to optimality.

Parameters: `<params>`

Template: `<code_example>`

Structure your answer with a description of the code solution, then list the imports, and finally list the functioning code block.

–Output Format Requirement (STRICT):
You must output your answer as a single, valid JSON object. Do **NOT** include any text before or after the JSON. Please provide the solution directly, without lengthy step-by-step reasoning.
The JSON object MUST have exactly the following fields:

- `"prefix"`: Description of the problem and the modeling approach
- `"imports"`: All required import statements only
- `"code"`: The complete code excluding import statements

– **Retrieval Augmented Generation** The RAG-based prompting strategy is executed in two distinct stages. In the initial iteration, the Executor retrieves relevant documentation to guide the generation of the baseline code. In subsequent iterations, the Executor utilizes the retrieved context to refine any erroneous code, iteratively addressing execution failures or constraint violations.

Prompt for RAG 1

You are an expert in Python programming for operations research and combinatorial optimization. You are good at calling `{solver}` in Python and solving problems.

Respond with the syntactically correct code for solving a `{problem}` using `{solver}`. Make sure you follow these rules:

- Read the template. First understand the meaning of the parameters in the `solve` function, and then complete the code inside the function.
- The context provides example codes of addressing each constraint of `{problem}` by `{solver}`. Learn to model each constraint and solve the problem accordingly.
- Do not give additional examples or define a main function for testing.
- Return the objective value of the problem by the `solve` function.
- Ensure any code you provide can be executed with all required imports and variables defined.
- If the current code can run, adjust it to make the solution approach a feasible or near-optimal result.

Current code: `<prep_code>`

Template: `<code_example>`

Context: `<context>`

Structure your answer with a description of the code solution, then list the imports, and finally list the functioning code block.

–**Output Format Requirement (STRICT):** You must output your answer as a single, valid JSON object. Do **NOT** include any text before or after the JSON. The JSON object must have exactly the following fields:

- `"prefix"`: Description of the problem and the modeling approach
- `"imports"`: All required import statements only
- `"code"`: The complete code excluding import statements

Prompt for RAG 2

You are responsible for refining the code with errors, which tries to solve `{problem}` by calling `{solver}` in Python.

The code snippet with the bug is as `<prep_code>`.

Here is the error message of the code: `<message>`.

Make sure you follow these rules:

- You can first reason about the error, and then refine the code and return the whole fixed function.
- The context provides tool descriptions for solving problems with different constraints. Refer to the relevant parts and modify the code accordingly: `<context>`.
- Do not give additional examples or define the main function for testing.
- Return the objective value of the problem by the `solve` function.
- Ensure any code you provide can be executed with all required imports and variables defined.

Structure your answer with a description of the code solution, then list the imports, and finally list the functioning code block.

–**Output Format Requirement (STRICT):**
You must output your answer as a single, valid JSON object. Do **NOT** include any text before or after the JSON. Please provide the solution directly, without lengthy step-by-step reasoning.
The JSON object MUST have exactly the following fields:

- `"prefix"`: Description of the problem and the modeling approach
- `"imports"`: All required import statements only
- `"code"`: The complete code excluding import statements

– **Meta-learning with Evolved Memory** To enable the agent to learn from previous modeling attempts, we design two memory-evolution prompts that summarize both successful and failed code refinements.
The success-memory prompt extracts reusable modeling strategies, solver-specific implementation patterns, and effective constraint formulations from revisions that produce valid solutions.

Prompt for Success Memory Evolution

You are an expert with extensive experience in modeling.

Below is the code for successfully modeling the `{problem}`: `<code>`

Your task is to summarize the successful modeling strategy in **plain English**. The summary should distill reusable modeling experience from the successful code, with particular attention to how the constraint is implemented and how similar strategies can be applied to related problems.

Your summary must:

- Be concise and informative
- Be no more than 100 words
- Highlight how the constraint was implemented

- Provide insights that help apply this method to similar problems

–**Output Format Requirement (STRICT):**
You must output your answer as a single, valid JSON object. Do **NOT** include any text before or after the JSON. Please provide the solution directly, without lengthy step-by-step reasoning.
The JSON object MUST have exactly the following fields:

- `"code"`: Code related to the modeling of the constraint
- `"description"`: Concise summary, no more than 100 words, highlighting the key modeling approach and the specific parameter settings related to constraint definition, solver configuration, or search strategy

In contrast, the failure-memory prompt identifies common error causes, invalid modeling choices, missing constraints, and ineffective repair behaviors from unsuccessful attempts.

Prompt for Failure Memory Evolution

You are an expert with extensive experience in modeling.

Below is the code for attempting to model the `{problem}`: `<code>`

The model encountered the following error: `<error>`

Your task is to analyze why the modeling or optimization attempt failed and summarize the failure experience in **plain English**. The summary should identify the key modeling, implementation, or optimization issue and provide practical insights for avoiding similar failures in future tasks.

Specifically, your analysis should:

- Identify the exact logic or implementation issue that caused the failure or poor performance
- Analyze why the solution quality degraded if the model did not reach or approach the optimal solution
- Consider possible causes such as incorrect objective formulation, missing constraints, unstable solver settings, poor initialization, or solver-specific API misuse
- Provide actionable insights on what part of the code or modeling logic should be revised
- Suggest alternative modeling strategies or improvements to avoid similar issues in future refinements
- Be concise and practical, with the description no more than 100 words

–**Output Format Requirement (STRICT):**
You must output your answer as a single, valid JSON object. Do **NOT** include any text before or after the JSON. Please provide the solution directly, without lengthy step-by-step reasoning.
The JSON object MUST have exactly the following fields:

- `"code"`: Code related to the modeling of the constraint

- `"description"`: A concise summary of the failed modeling approach and insights on how to avoid similar issues, no more than 100 words

Examples of the output of the two memory-evolution prompts are shown below.

Examples of Evolved Memories

The following examples illustrate the evolved memories generated by the success-memory and failure-memory prompts. Each memory summarizes reusable modeling experience from previous attempts and provides guidance for subsequent code refinement.

Successful Memories.

- **Simple Vehicle Routing Problem (VRP).** The Simple Vehicle Routing Problem (VRP) is modeled using OR-Tools. A routing index manager and routing model are created to handle the nodes and vehicles. A distance callback function is registered to compute distances between nodes, which is used to set the arc cost evaluator for all vehicles. The first solution strategy is set to `PATH_CHEAPEST_ARC` to find an initial feasible solution. This approach can be adapted to similar routing problems by adjusting the distance matrix and vehicle parameters.
- **Vehicle Routing Problem with Service Time (VRPS).** The VRPS model uses OR-Tools to optimize vehicle routes considering service times. A time callback function calculates travel and service time between nodes and is registered as a transit callback. The `Time` dimension restricts total time per vehicle, ensuring feasible routes. The `PATH_CHEAPEST_ARC` strategy finds initial solutions efficiently. This approach is adaptable to similar routing problems by adjusting time matrices, service times, and constraints.
- **Capacitated Vehicle Routing Problem with Multiple Depots (CVRPMD).** The CVRPMD is modeled using OR-Tools by defining distance and demand callbacks to handle routing and capacity constraints. The `RoutingIndexManager` manages node indices, while the `RoutingModel` sets up the problem. Vehicle capacity constraints are added using `AddDimensionWithVehicleCapacity`, ensuring demands do not exceed vehicle capacities. This approach can be adapted by adjusting distance matrices, demand lists, and vehicle capacities.
- **Vehicle Routing Problem with Multiple Depots (VRPMD).** The VRPMD is modeled using OR-Tools, leveraging the `RoutingIndexManager` to handle multiple depots. A distance callback function calculates travel costs between nodes and is registered with the `RoutingModel` to evaluate arc costs. The solver uses the `PATH_CHEAPEST_ARC` strategy for initial solutions, making the approach adaptable to similar multi-depot or multi-vehicle routing problems.
- **Vehicle Routing Problem with Service Time and Duration Limit (VRPSL).** The VRPSL is modeled using OR-Tools, focusing on service time and duration limits. A routing index manager and model are created,

with a transit callback registered to calculate travel times. The arc cost is set for all vehicles, and a time dimension is added using `AddDimension` to enforce maximum duration per vehicle. This approach can be adapted by adjusting the time matrix, service times, and duration limits.

Failure Memories.

- **Prize Collecting Vehicle Routing Problem (PCVRP).** The failed attempts at solving PCVRP primarily stem from incorrect objective formulation, such as maximizing prizes without considering travel costs or penalties for unvisited nodes. This misalignment leads to suboptimal solutions. Future models should use a weighted objective that integrates prize maximization and travel cost minimization, incorporate penalties for unvisited nodes, and ensure all constraints are correctly implemented.
- **Prize Collecting Vehicle Routing Problem with Multiple Depots (PCVRPMD).** The modeling failures primarily stem from incorrect initialization of `RoutingIndexManager`, where starts and ends should be lists of node indices rather than single indices. Misalignment of objectives and constraints, particularly in maximizing collected prizes while respecting distance limits, further contributed to poor performance. Future refinements should correctly initialize depot lists, verify objective-constraint alignment, and tune prize-related dimensions.
- **Vehicle Routing Problem with Time Windows and Multiple Depots and Duration Limit (VRPTWMDL).** The failed modeling attempts for VRPTWMDL primarily stem from incorrect handling of time windows, duration limits, and multiple depots, leading to infeasibility or suboptimal solutions. Key issues include improper initialization of vehicle routes, insufficient constraints, and inadequate solver time limits. Future models should correctly implement time-window and duration constraints, use robust initialization strategies, and increase solver time limits when necessary.
- **Capacitated Vehicle Routing Problem with Distance Limit (CVRPL).** The modeling failures stemmed from inadequate objective function definition, improper constraint handling, and ineffective solver settings. Key issues included missing or incorrectly implemented distance constraints and arbitrarily set span cost coefficients. Future models should explicitly define objectives, correctly apply distance constraints, and adjust solver settings to better guide the search process.
- **Vehicle Routing Problem with Time Windows and Duration Limit (VRPTWL).** Modeling failures stem from incorrect handling of time-window constraints, particularly at the depot, and misapplication of duration limits. These mistakes can lead to infeasible solutions. To avoid such errors, future models should exclude the depot from inappropriate time-window constraints, establish separate dimensions for duration limits, ensure correct index mapping, and debug smaller instances before scaling.
- **Vehicle Routing Problem with Time Windows and Open Start and End locations (OVRPTW).** The modeling failures stemmed from missing `starts` and `ends` parameters in the

`solve` function, which are crucial for defining open routes in the `RoutingIndexManager`. To improve solution quality, future refinements should correctly define and pass start and end locations, adjust time windows for feasibility, and use stronger initialization strategies such as `PARALLEL_CHEAPEST_INSERTION`.

- **Vehicle Routing with Pickups and Deliveries and Distance Limit (PDPL).** The modeling failures primarily stem from incorrect constraint handling, particularly applying distance limits to individual nodes rather than the entire route. Other issues include insufficient constraints, poor initialization, and an ineffective objective function. Future models should enforce route-level distance limits, revise the objective to minimize total distance, and adjust solver parameters for better search exploration.

This prompt is designed to support meta-learning–driven code refinement for operations research problems solved via a given solver. Instead of performing isolated debugging, the model is instructed to leverage accumulated experience from prior successful and unsuccessful code revisions to guide the current refinement process.

Prompt for MLEM

You are an expert in Python programming for operations research by calling `{solver}`.

Your task is to analyze the provided erroneous code that calls the `{solver}` to solve the `{problem}`, identify errors, and modify it to make it correct and functional. If the current code executes successfully, prioritize obtaining a feasible solution, and then optimize solver settings to move the solution closer to optimality.

To guide your reasoning, positive examples (successful code fixes from similar tasks) and negative examples (failed attempts or common pitfalls from similar tasks) may be provided. Use meta-learning principles: extract general patterns, strategies, and lessons from these examples (e.g., common error types like incorrect parameter passing, data formatting issues, or solver-specific constraints), then adapt and migrate those insights to the current problem. Focus on transferring successful approaches while avoiding the pitfalls in the negative examples.

Examples: `<examples>`

Current erroneous code: `<code>`

Template: `<code_example>`

Parameters: `<params>`

Structure your answer with a description of the code solution, then list the imports, and finally list the functioning code block.

–Output Format Requirement (STRICT):

You must output your answer as a single, valid JSON object. Do **NOT** include any text before or after the JSON. Please provide the solution directly, without lengthy step-by-step reasoning.
The JSON object MUST have exactly the following fields:

- `"prefix"`: Description of the problem and the modeling approach
- `"imports"`: All required import statements only
- `"code"`: The complete code excluding import statements

References

1. Jiang, X., Wu, Y., Zhang, C., Zhang, Y.: DRoC: Elevating large language models for complex vehicle routing via decomposed retrieval of constraints. In: ICLR (2025)
2. Wouda, N.A., Lan, L., Kool, W.: Pyvrp: A high-performance VRP solver package. INFORMS J. Comput. **36**(4), 943–955 (2024)