# DRoC: Elevating Large Language Models for Complex Vehicle Routing via Decomposed Retrieval of Constraints

Xia Jiang[1], Yaoxin Wu[1,*], Chenhao Zhang[2], Yingqian Zhang[1]

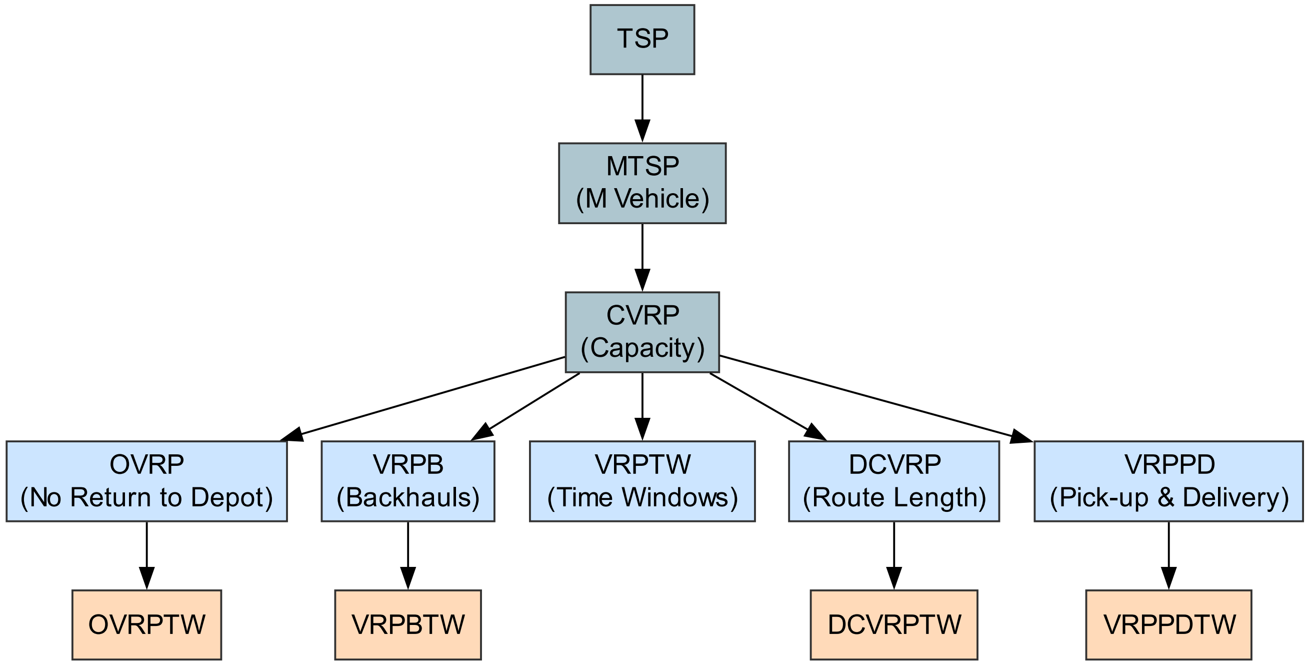
Eindhoven University of Technology, The Netherlands [1]
Southeast university, P.R., China [2]

## Overview

- Background
- Framework
- Experiment Results

## Background

# Vehicle Routing Problem (VRP)

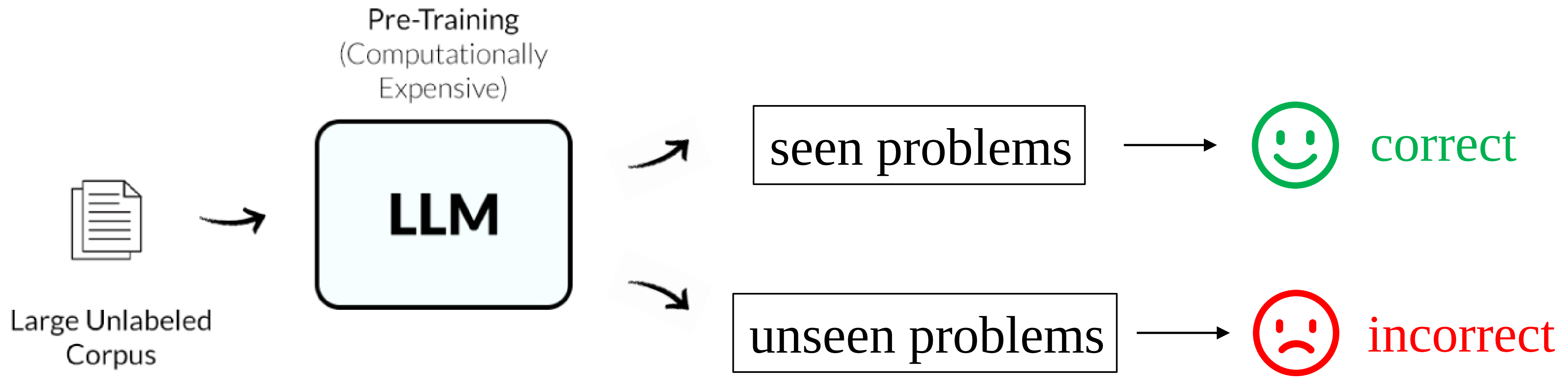


The goal of an VRP is to determine the most efficient routes for a fleet of vehicles to service a set of locations, while meeting constraints like vehicle capacity and time windows.

VRPs are inherently with **different constraints**, such as time windows, backhauls, or route length limits, reflecting **real-world logistical considerations**.

## LLM for Optimization

### Dilemma of current solutions



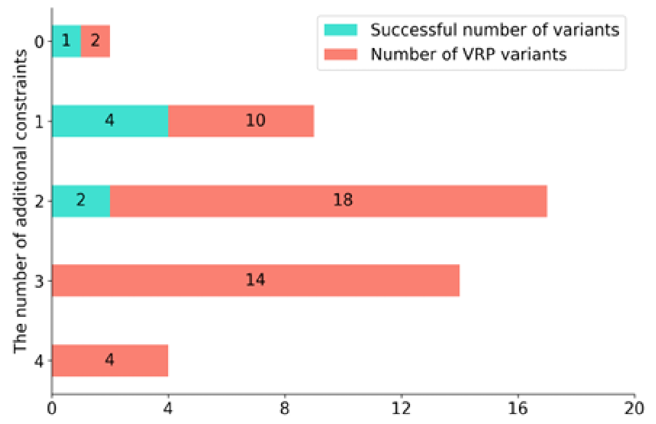


Figure 1: The evaluation of gpt-3.5-turbo on 48 VRP variants with different numbers of composite constraints. Performance declines with increased constraints.

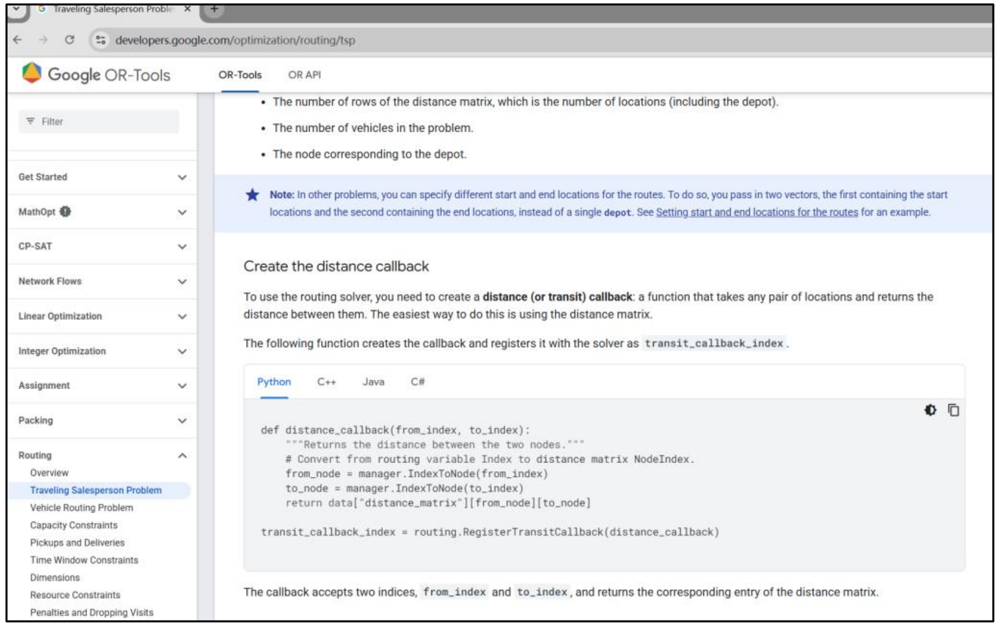
LLMs tend to successfully solve simple problems because:

- Simple problems are **usually contained in the corpus** during pre-training stage.
- It is challenging to reason how to **integrate heterogeneous constraints** within a generated program.

**Relying on external knowledge is important!**

# Framework

## Retrieval Augmented Generation



- The number of rows of the distance matrix, which is the number of locations (including the depot).
- The number of vehicles in the problem.
- The node corresponding to the depot.

**Note:** In other problems, you can specify different start and end locations for the routes. To do so, you pass in two vectors, the first containing the start locations and the second containing the end locations, instead of a single `depot`. See Setting start and end locations for the routes for an example.

Create the distance callback

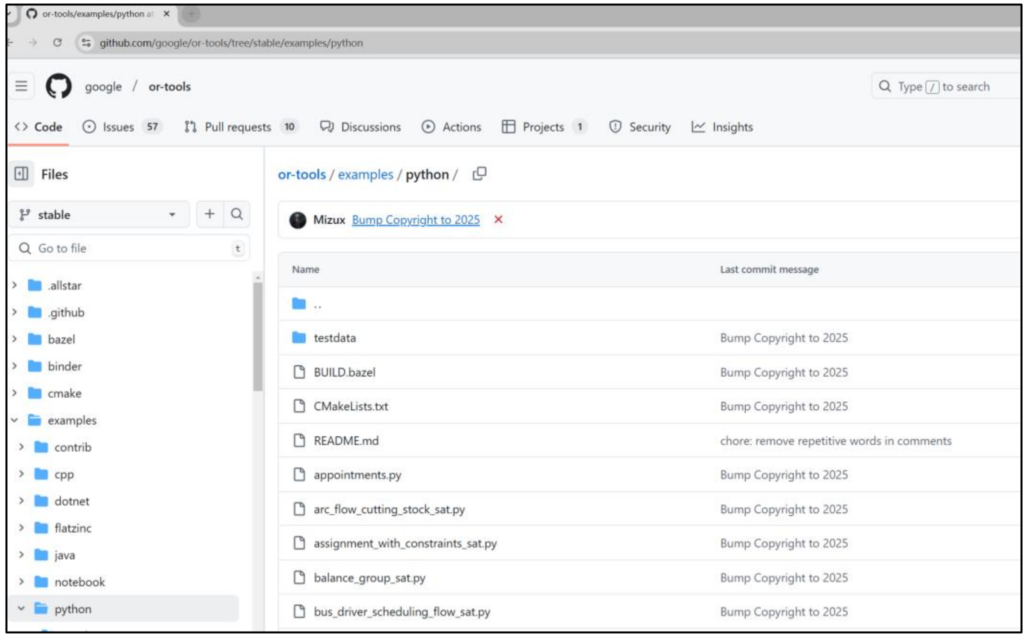
To use the routing solver, you need to create a **distance (or transit) callback**: a function that takes any pair of locations and returns the distance between them. The easiest way to do this is using the distance matrix.

The following function creates the callback and registers it with the solver as `transit_callback_index`.

```
def distance_callback(from_index, to_index):
    """Returns the distance between the two nodes."""
    # Convert from routing variable Index to distance matrix NodeIndex.
    from_node = manager.IndexToNode(from_index)
    to_node = manager.IndexToNode(to_index)
    return data["distance_matrix"][from_node][to_node]

transit_callback_index = routing.RegisterTransitCallback(distance_callback)
```

The callback accepts two indices, `from_index` and `to_index`, and returns the corresponding entry of the distance matrix.



or-tools / examples / python /

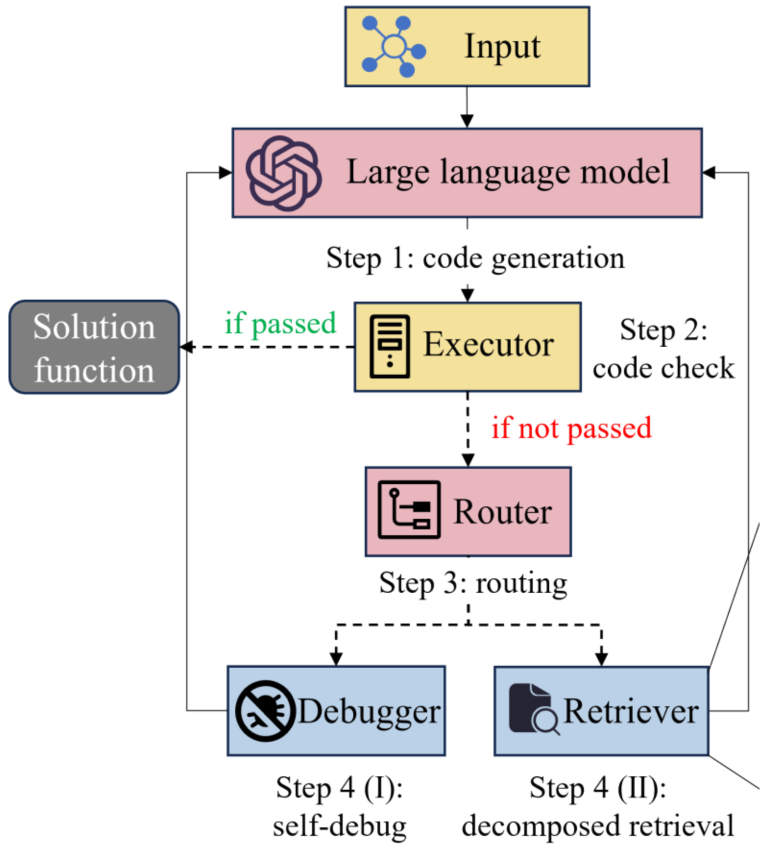
Mizux Bump Copyright to 2025

| Name | Last commit message |
|---|---|
| .. | |
| testdata | Bump Copyright to 2025 |
| BUILD.bazel | Bump Copyright to 2025 |
| CMakeLists.txt | Bump Copyright to 2025 |
| README.md | chore: remove repetitive words in comments |
| appointments.py | Bump Copyright to 2025 |
| arc_flow_cutting_stock_sat.py | Bump Copyright to 2025 |
| assignment_with_constraints_sat.py | Bump Copyright to 2025 |
| balance_group_sat.py | Bump Copyright to 2025 |
| bus_driver_scheduling_flow_sat.py | Bump Copyright to 2025 |

**Solver online documentation** **Open-source code repository**

There are various **online knowledge sources** can be used to enhance the LLM in generation.

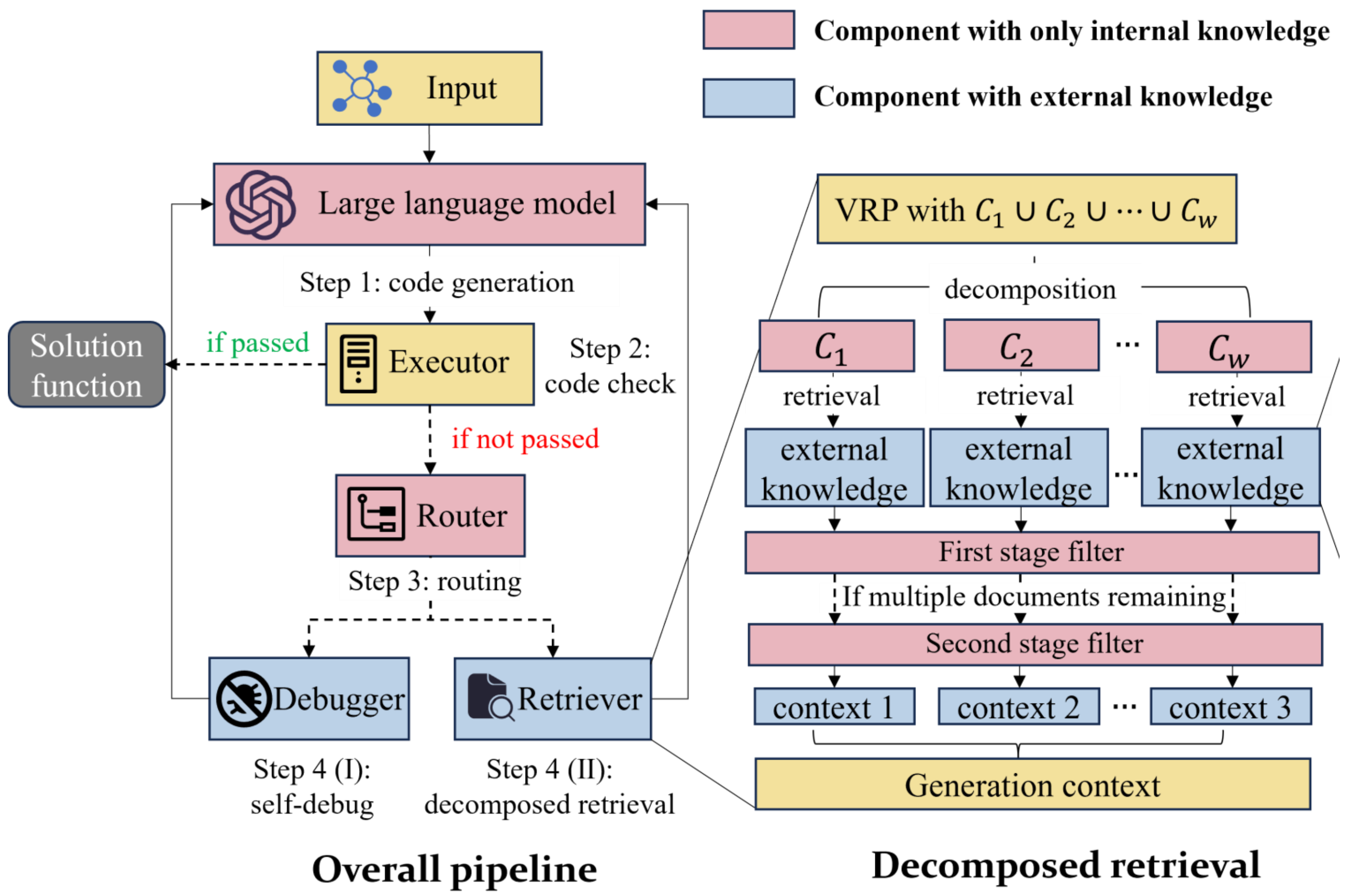
## Framework

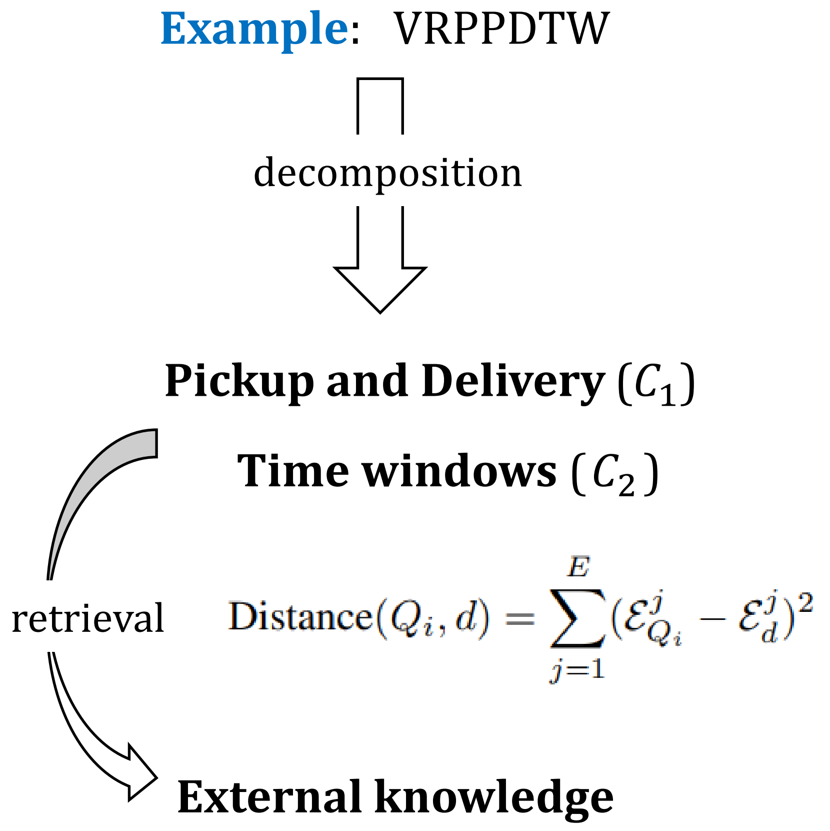


**Overall pipeline**

- Step 1: An LLM as the is prompted **directly by the input (i.e., a VRP) to generate a program.**
- Step 2: The program generated in Step 1 is **run by a code executor**, invoking a solver to solve the VRP. The LLM will be provided with execution traceback if the code contains errors.
- Step 3: According to the execution traceback, an **LLM as a router** determines the operation in Step 4, either self-debugging (I) or RAG (II).
- Step 4 (I) : An LLM as the **self-debugger analyzes the execution traceback** and attempts to refine the code.
- Step 4 (II): The retrieval is decomposed to **refer LLM to external documentation or example codes** for seeking relevant documents on separate constraints.

## Framework





$$\text{Distance}(Q_i, d) = \sum_{j=1}^{E} (\mathcal{E}_{Q_i}^j - \mathcal{E}_d^j)^2$$

## Framework

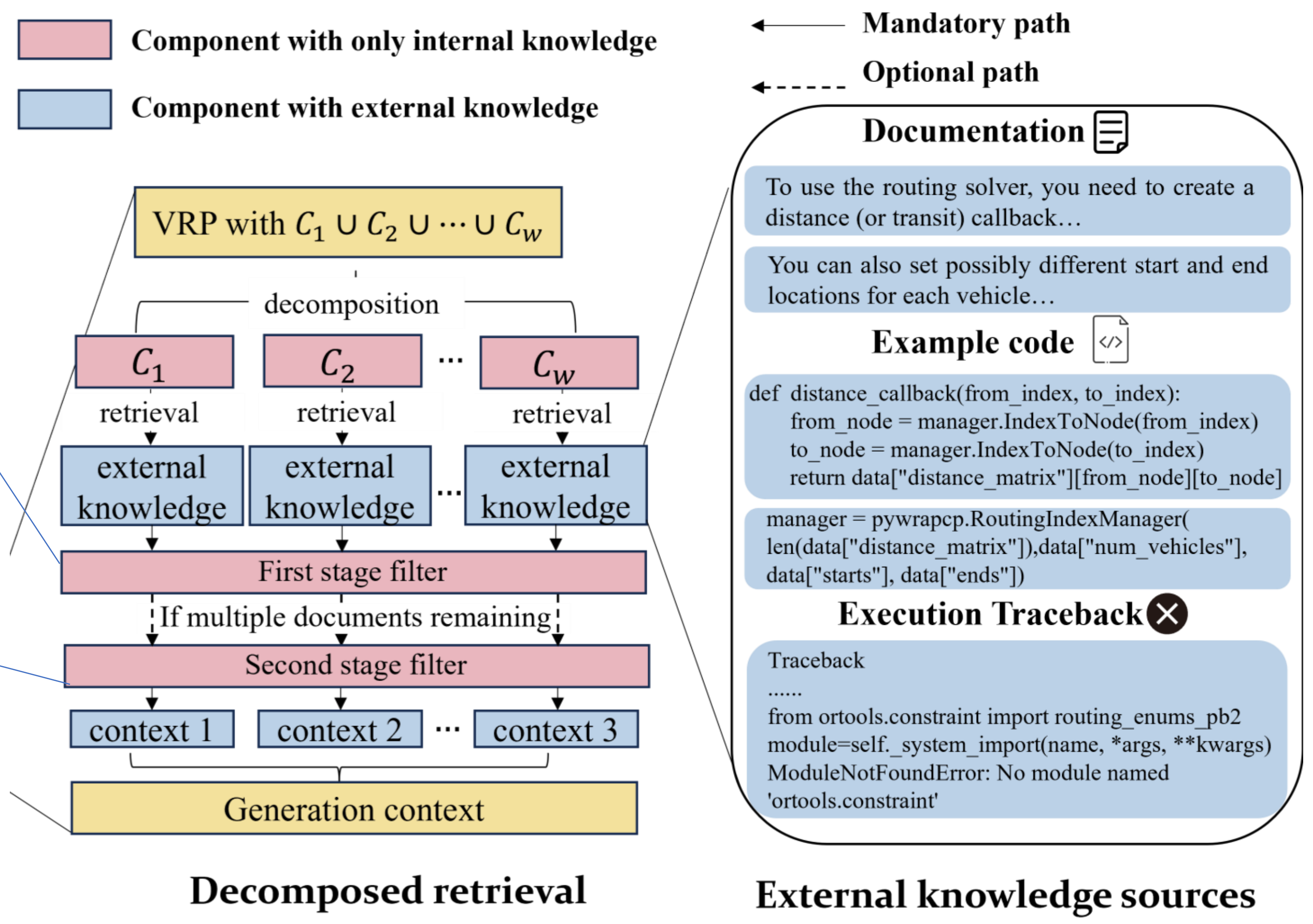
### Pickup and Delivery ( $C_1$ )

**Doc 1**: In the General Pickup and Delivery Problem (GPDP) a set of routes has to be constructed in order to satisfy transportation requests…

**Doc 2**: for request in data["pickups_deliveries"]: pickup_index =manager.NodeToIndex(request[0]) delivery_index = manager.NodeToIndex(request[1])

**Doc 3**: In the Vehicle Routing Problem (VRP), the goal is to find optimal routes for multiple vehicles…

**Doc 1 and Doc 3 are filtered out**



To use the routing solver, you need to create a distance (or transit) callback…

You can also set possibly different start and end locations for each vehicle…

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

```
manager = pywrapcp.RoutingIndexManager(
len(data["distance_matrix"]),data["num_vehicles"],
data["starts"], data["ends"])
```

```
Traceback
......
from ortools.constraint import routing_enums_pb2
module=self._system_import(name, *args, **kwargs)
ModuleNotFoundError: No module named
'ortools.constraint'
```

## Experiment Results

### Considered constraints

Vehicle capacity

Distance limit

Time windows

Multiple depots

Open route

Prize collecting

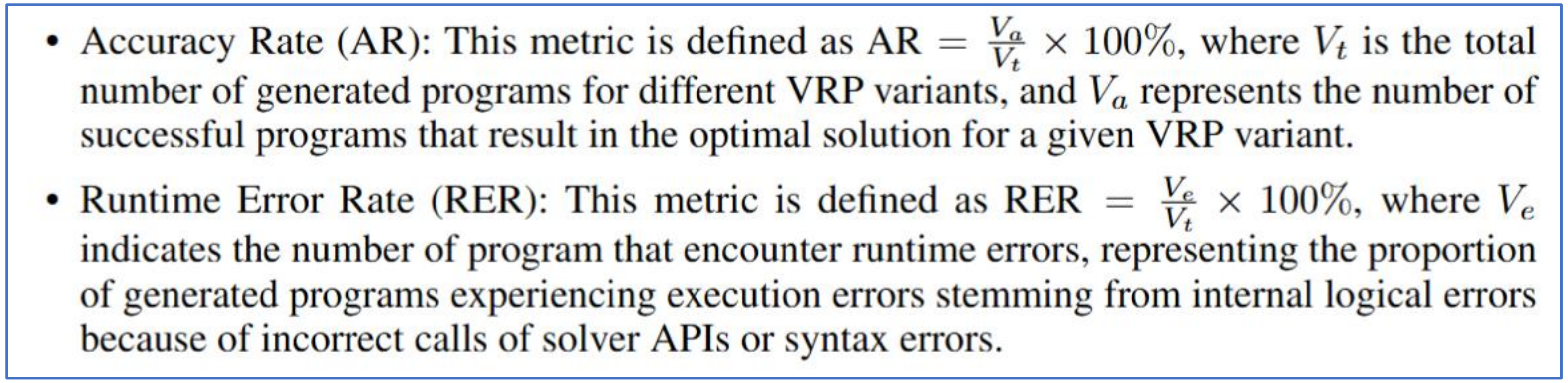
Pickups and deliveries

Service time

Resource constraints

**48 VRP variants are formulated by integrating different constraints**

### Evaluation metrics



- Accuracy Rate (AR): This metric is defined as $AR = \frac{V_a}{V_t} \times 100\%$, where $V_t$ is the total number of generated programs for different VRP variants, and $V_a$ represents the number of successful programs that result in the optimal solution for a given VRP variant.
- Runtime Error Rate (RER): This metric is defined as $RER = \frac{V_e}{V_t} \times 100\%$, where $V_e$ indicates the number of program that encounter runtime errors, representing the proportion of generated programs experiencing execution errors stemming from internal logical errors because of incorrect calls of solver APIs or syntax errors.

### Baselines

**Standard Prompting:** Directly prompt the LLM for code generation.

**Chain-of-Thought (CoT):** Add "Let’s think step by step“ in the prompt.

**Progressive-Hint Prompting (PHP):** Iteratively guide the LLM.

**Self-debug:** Use the error information and traceback to teach the LLM.

**Vanilla RAG:** Directly retrieve relevant context without decomposing.

**Self-RAG:** A retriever is used to obtain relevant documents, followed by a grader to assess whether each document is pertinent to the target VRP.

## Experiment Results

### Evaluation on OR-Tools

Table 1: Performance of different methods with gpt-3.5-turbo and gpt-4o. The reported values are averaged over the results of 48 VRP variants.

| Method (gpt-3.5-turbo) | AR ↑ | RER ↓ | Method (gpt-4o) | AR ↑ | RER ↓ |
|---|---|---|---|---|---|
| Standard Prompting | 14.58% | 66.67% | Standard Prompting | 20.83% | 58.33% |
| CoT | 8.33% | 79.17% | CoT | 22.92% | 50.00% |
| PHP | 12.50% | 79.17% | PHP | 22.92% | 54.17% |
| Self-debug | 14.58% | 66.67% | Self-debug | 31.25% | 37.50% |
| Vanilla RAG | 10.42% | 60.42% | Vanilla RAG | 37.50% | 31.25% |
| Self-RAG | 16.67% | 56.25% | Self-RAG | 33.33% | 39.58% |
| DRoC (Ours) | **20.83%** | **47.92%** | DRoC (Ours) | **45.83%** | **20.83%** |

- Solving VRPs represents a **challenging task** for the current LLMs (only 45.83% problems are accurately solved).
- Using RAG does not necessarily improve the performance (because of the irrelevant documents are harmful).
- Our method leads to the best performance by constraint decomposing and retrieving.

```
# Define the prize collection callback
def prize_callback(from_index):
    from_node = manager.IndexToNode(from_index)
    return prizes[from_node]
prize_callback_index = routing.RegisterUnaryTransitCallback(
    prize_callback)

routing.AddDimensionWithVehicleCapacity(
    prize_callback_index,
    0,  # no slack
    [sum(prizes)] * num_vehicle,  # vehicle maximum prize capacity
    True,  # start cumul to zero
    'Prize')
# Setting the objective to maximize the prize collection
prize_dimension = routing.GetDimensionOrDie('Prize')
for vehicle_id in range(num_vehicle):
    routing.SetFixedCostOfVehicle(-sum(prizes), vehicle_id)
```

(a) Generated code snippet by Standard Prompting. (Incorrect)

```
# Allow to drop nodes.
for node in range(1, len(distance_matrix)):
    routing.AddDisjunction([manager.NodeToIndex(node)], prizes[node])
```

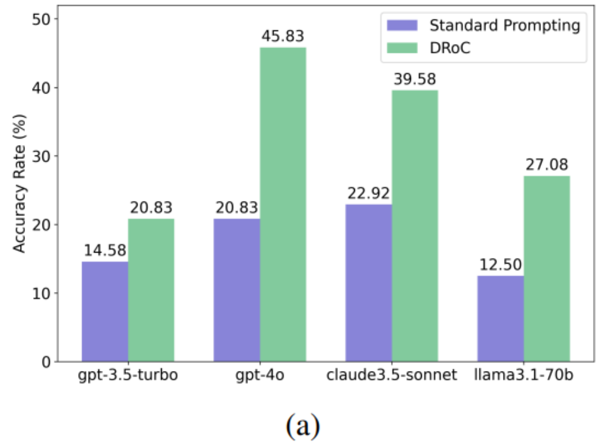
(b) Generated code snippet by DRoC. (Correct)

- The ‘prize callback’ is added to capacity dimension because of the **hallucination** of the LLM.
- Our method deal with prize collection by allowing to **drop nodes** during vehicle travelling.

## Experiment Results

Table 2: The performance evaluated on Gurobi solver with and without DRoC.

| LLM | AR ↑ | RER ↓ |
|---|---|---|
| gpt-4o (Standard Prompting) | 2.08% | 45.83% |
| claude-3.5-sonnet (Standard Prompting) | 12.50% | 52.08% |
| gpt-4o (DRoC) | **31.25%** | **25.00%** |
| claude-3.5-sonnet (DRoC) | **31.25%** | **37.50%** |



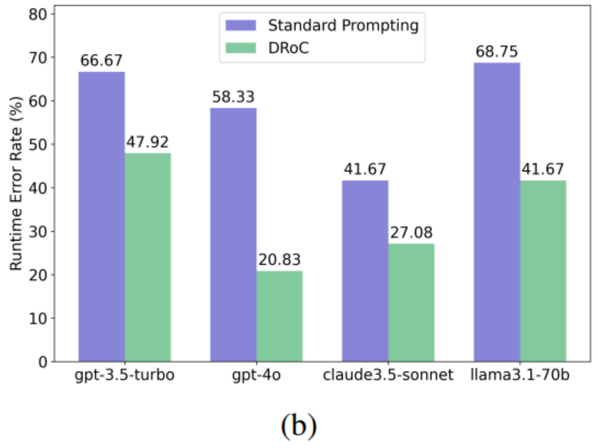


Figure 3: Performance of DRoC and Standard Prompting with different LLMs: (a) AR metric (b) RER metric. The DRoC is applicable to varied LLMs, showing clear performance enhancements.

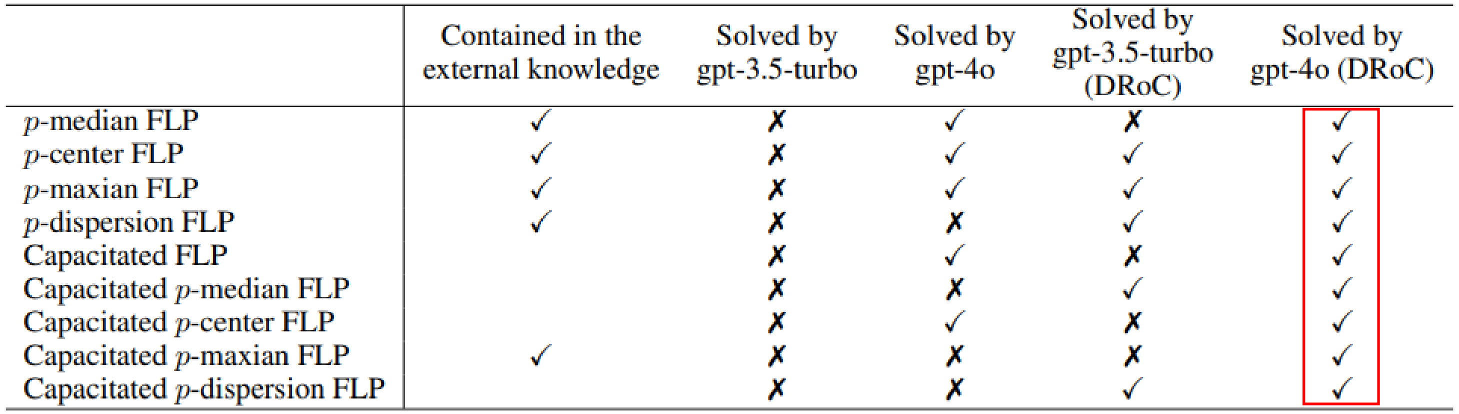
### DRoC as a generic tool

- It is very challenging to use Gurobi for modelling and programming by LLMs.
- DRoC can enhance LLMs in calling **different optimization solvers.**
- Compared to the standard prompting approach, DRoC successfully solves 25% more VRP variants by gpt-4o.
- DRoC can enhance **different LLMs** for generating correct programs and reducing runtime errors.

## Experiment Results

### Generalizability to other OR problems

Table 6: The evaluation results on the FLPs.



| | Contained in the external knowledge | Solved by gpt-3.5-turbo | Solved by gpt-4o | Solved by gpt-3.5-turbo (DRoC) | Solved by gpt-4o (DRoC) |
|---|---|---|---|---|---|
| $p$-median FLP | ✓ | ✗ | ✓ | ✗ | ✓ |
| $p$-center FLP | ✓ | ✗ | ✓ | ✓ | ✓ |
| $p$-maxian FLP | ✓ | ✗ | ✓ | ✓ | ✓ |
| $p$-dispersion FLP | ✓ | ✗ | ✗ | ✓ | ✓ |
| Capacitated FLP | | ✗ | ✓ | ✗ | ✓ |
| Capacitated $p$-median FLP | | ✗ | ✗ | ✓ | ✓ |
| Capacitated $p$-center FLP | | ✗ | ✓ | ✗ | ✓ |
| Capacitated $p$-maxian FLP | ✓ | ✗ | ✗ | ✗ | ✓ |
| Capacitated $p$-dispersion FLP | | ✗ | ✗ | ✓ | ✓ |

The **Facility Location Problem** aims at selecting optimal sites for $p$ facilities to minimize the conveying cost from facilities to customers. It usually encompasses **different optimization objectives** for different variants.

- $p$-median: minimizing the total distance from each customer to their nearest facility.
- $p$-center: minimizing the maximum cost any customer spend to reach the nearest facility.
- $p$-maxian: maximizing the total weighted travel cost incurred by all customers.
- $p$-dispersion: maximizing the smallest distance between any pair of facilities.
- Facility capacity: the total demand assigned to a facility must not exceed its capacity.

# Thanks for listening