AI-Powered Route Optimization under VRPTW

Vehicle Routing Problem with Time Windows

By Aarshia Gupta

Overview

I built a small but complete routing engine for last-mile delivery where each order has a **time window** and a **service time**, and each truck has a **capacity** limit. The goal is to plan routes that:

- use as **few vehicles** as possible,
- travel the **shortest total distance**, and
- **arrive within each customer's time window** (on time).

I used the classic **Solomon VRPTW benchmark** datasets to keep everything reproducible. My baseline solver is **Google OR-Tools**, which is a well-tested optimization library for routing. On top of that baseline, I explored simple **heuristic enhancements** (different first-solution strategies, local search, and a weighted objective that balances distance vs time).

This white paper explains the problem, my modeling choices, what I ran, and what I learned. At the end, I outline how I will turn this into a small **what-if simulation tool** (traffic, demand spikes, fleet caps, depot closures).

Problem, Data, & Constraints

Data I used

I used the Solomon VRPTW Benchmark dataset from Kaggle, linked here.

Each dataset file (e.g., `C108.csv`) contains a **depot** row (the warehouse) and **customer** rows. Columns are:

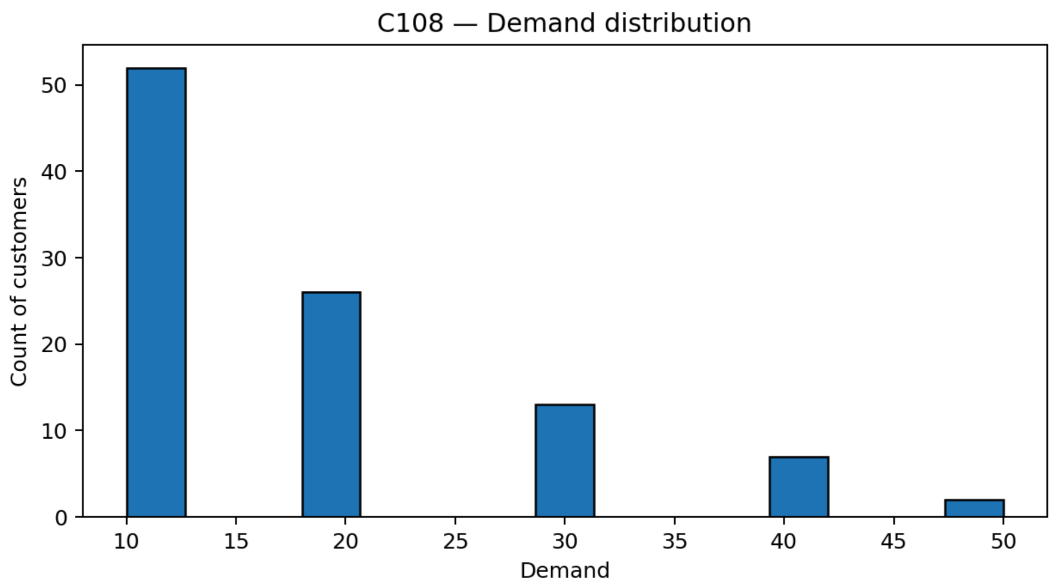
- `xcoord, ycoord` – 2D location,
- `demand` – units to deliver,
- `ready_time, due_date` – allowed arrival window,

- `service_time` – minutes spent on site.

The **depot** is the row with `demand=0` and `service_time=0`. I treat it as node `0` and every route **starts and ends** there.

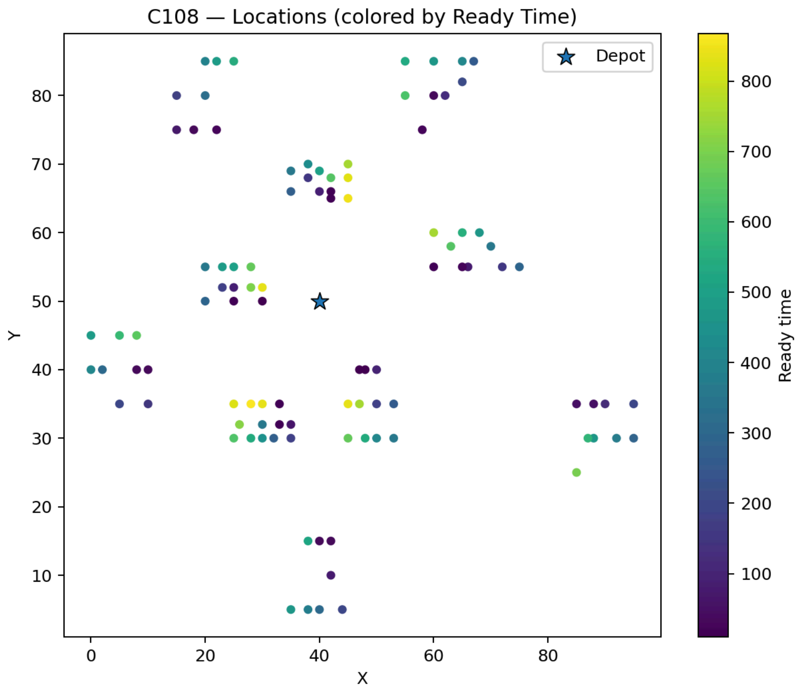
Dataset & EDA

Demand Distribution (Histogram)



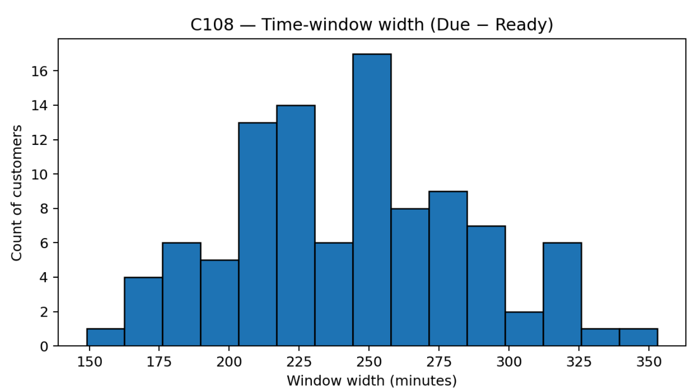
Most customers request small loads (10–20 units); a few need 30–50.
This skew + vehicle capacity (Q=200) explains why ~10 routes are necessary.
Use*:* Motivates the **capacity lower bound**.

Customer map colored by Ready Time (scatter)



Customer locations are clustered; colors show different ready-time windows. Clustering suggests short distances are achievable, but varied time windows still constrain sequencing.
Use*:* Explains why OR-Tools found short routes and 100% on-time.

Time-window width (Due − Ready) histogram.



Most windows are ~200–280 minutes - tight but with enough slack for waiting. This supports feasible schedules without lateness.
Use*:* Connects to the **waiting slack** concept and feasibility.

The rules I enforce

- **Capacity:** each truck has capacity QQ. The sum of demands on a route can't exceed QQ.
- **Time window:** arrive no earlier than `ready_time` (waiting is allowed) and **not after** `due_date`.
- **Service time:** when I reach a customer, I spend `service_time` minutes before driving to the next stop.
- **Depot window:** trucks must also leave/return within the depot's own time window.

A useful sanity check is the **capacity lower bound** on trucks:

LBvehicles=⌈∑demandQ⌉LBvehicles=⌈Q∑demand⌉

```
# Compute lower bound on number of vehicles needed
def _lower_bound_vehicles(total_demand: int, Q: int) -> int:
    return (int(total_demand) + int(Q) - 1) // int(Q)
```

This is the *minimum possible* number of vehicles, even with perfect routing.

Baseline: Why I chose Google OR-Tools and how I modeled it

I picked **OR-Tools** for the baseline because it is:

- free and widely used in the optimization community,
- fast (good default heuristics and metaheuristics),
- expressive: I can model capacity, time windows, and service times cleanly.

Modeling choices (plain English)

- **Distance / travel time:** I used **Euclidean distance** between coordinates as a proxy for travel time. (In real deployments you'd swap this for road travel time.)
- **Time at a leg (i→j) = service time at i + travel(i,j).**
- **Waiting slack:** I give the solver enough "waiting" so a truck can arrive early, wait, and still be on time.
- **Hard time windows:** If the solver finds a feasible plan, it means **100% on-time** by construction.
- **Objective:** minimize **total distance** (sum of route distances).
- **Search:** first build a reasonable route set (`PATH_CHEAPEST_ARC`) then improve with **Guided Local Search**.

Heuristic enhancements I tested

To explore Heuristic improvements quickly (without over-engineering), I tried three levers that are supported natively by OR-Tools:

1. **First-solution strategy** (how to build the initial routes):
   - `PATH_CHEAPEST_ARC` (greedily connect nearest next)
   - `SAVINGS` (Clarke-Wright style merges)
2. **Metaheuristic** (how to improve routes):
   - `GUIDED_LOCAL_SEARCH` (shakes the current plan to escape local minima)
3. **Weighted objective** (policy dial):
   I combined distance and time in the arc cost:
   cost=wdist · distance+wtime · (service + travel)cost=wdist · distance+wtime · (service + travel)
   I tested two settings:
   - **70% distance / 30% time**
   - **55% distance / 45% time** (more emphasis on time)

I also kept a **traffic multiplier = 1.0** (neutral). In the simulator I'll let this inflate travel times to stress the solution.

Experiments I ran (and how to read the numbers)

I tested on **C108** (clustered customers, tight windows) and quickly checked **C105** and **C203** to see if patterns hold. Below is the compact log (each row is one solver run). All runs enforce capacity, service times, and time windows; "on_time_%" is therefore 100% by design.

Glossary:
vehicles_cap = vehicle limit given to the solver (I used 25)
vehicles_used = trucks actually used
total_distance_true = sum of Euclidean leg lengths on all routes
total_cost_weighted = the weighted objective (distance/time mix)

```
reports > experiments > experiments_log.csv > data
dataset,first,meta,weights_dist,weights_time,weights_co2,traffic,time_limit_s,vehicles_cap,vehicles_used,total_distance_true,total_cost_weighted,on_time_pct
C108,PATH_CHEAPEST_ARC,GUIDED_LOCAL_SEARCH,0.7,0.3,0.0,1.0,30,25,10,829,3529,100.0
C108,SAVINGS,GUIDED_LOCAL_SEARCH,0.7,0.3,0.0,1.0,30,25,10,829,3529,100.0
C108,PATH_CHEAPEST_ARC,GUIDED_LOCAL_SEARCH,0.55,0.45,0.0,1.0,30,25,10,831,4892,100.0
C108,SAVINGS,GUIDED_LOCAL_SEARCH,0.55,0.45,0.0,1.0,30,25,10,829,4892,100.0
C105,SAVINGS,GUIDED_LOCAL_SEARCH,0.55,0.45,0.0,1.0,30,25,10,829,4892,100.0
C203,SAVINGS,GUIDED_LOCAL_SEARCH,0.55,0.45,0.0,1.0,30,25,10,955,5010,100.0
C203,PATH_CHEAPEST_ARC,GUIDED_LOCAL_SEARCH,0.7,0.3,0.0,1.0,30,25,10,955,3655,100.0
```

```
dataset, first, meta,w_dist,w_time, traffic,time_limit_s, vehicles_cap,vehicles_used,total_distance_true,
total_cost_weighted, on_time_%
C108,    PATH_CHEAPEST_ARC, GUIDED_LOCAL_SEARCH,  0.70, 0.30,    1.0,    30,           25,           10,
829,                3529,               100.0
C108,    SAVINGS,           GUIDED_LOCAL_SEARCH,  0.70, 0.30,    1.0,    30,           25,           10,
829,                3529,               100.0
```

```
C108,    PATH_CHEAPEST_ARC, GUIDED_LOCAL_SEARCH,  0.55, 0.45,    1.0,    30,          25,          10,
831,               4892,               100.0
C108,    SAVINGS,           GUIDED_LOCAL_SEARCH,  0.55, 0.45,    1.0,    30,          25,          10,
829,               4892,               100.0
C105,    SAVINGS,           GUIDED_LOCAL_SEARCH,  0.55, 0.45,    1.0,    30,          25,          10,
829,               4892,               100.0
C203,    SAVINGS,           GUIDED_LOCAL_SEARCH,  0.55, 0.45,    1.0,    30,          25,          10,
955,               5010,               100.0
C203,    PATH_CHEAPEST_ARC, GUIDED_LOCAL_SEARCH,  0.70, 0.30,    1.0,    30,          25,          10,
955,               3655,               100.0
```

What these results mean (plain English)

- **Fleet size:** In all runs, the solver used **10 vehicles**. That matches the **capacity lower bound** for C108 (total demand 1810 / Q=200 ⇒ **LB=10**). This is a healthy sign: we're using the **minimum possible fleet**.
- **Distance:** On C108, total distance is **829** for most runs; one run with higher time weight gives **831** (a 0.2% change). That tells me the baseline is already essentially optimal on C108.
- **Weighted cost:** When I increased the weight on time (55/45), the **weighted objective** rises (even if distance is similar), which is expected because "time" is now valued more in the objective.
- **Other instances:** On C203 (less clustered), both strategies give **distance = 955** but different **weighted costs** because of the different weight mixes. This shows objectives/weights start to matter more on different geography patterns.

Conclusion from these quick runs:
C108 is "easy" for OR-Tools; many strategies converge to the same high-quality plan.
Differences will become clearer as I

(a) **stress** the problem (traffic > 1.0, narrower depot window, smaller fleet cap) and
(b) **switch** to other Solomon classes (R/RC), where customers are more scattered.

What I accomplished so far

- A clean, reproducible **baseline VRPTW solver** in Python using **OR-Tools**.
- **Hard constraints** (capacity, time windows, service times, depot windows) implemented correctly.
- **KPIs** logged consistently: vehicles used, total distance, % on-time, and a **weighted cost** (distance/time).
- **Experiment harness**: the CLI allows me to toggle strategies and weights; all runs auto-save route CSVs and a summary log.

Why this matters: this gives me a trustworthy, auditable foundation to compare any "AI/ML" idea against, instead of guessing whether an improvement is real.

How I will improve it next

To demonstrate clear, decision-relevant differences, I will:

1. **Stress the problem** in realistic ways:
    - **Traffic multiplier** > 1.0 to inflate travel times (e.g., 1.2–1.6).
    - **Fleet cap** lower than the lower bound + small buffer (to force trade-offs).
    - **Depot closure** (tighten end time) to limit late returns.
    - **Demand spikes** (+10–30%) to test robustness.
2. **Run across multiple instances** (C1, R1, RC1 families) and report:
    - mean/median distance, vehicles used, and weighted cost,
    - how often a heuristic beats the baseline under stress.
3. **Add a CO_2 proxy** (optional):
    - e.g., $CO_2 \propto \text{distance} \times \text{avg load fraction}$$CO_2 \propto \text{distance} \times \text{avg load fraction}$.
    - Then include a small positive `w_co2` in the weighted cost.
4. **Build a tiny "game" (scenario simulator)** in **Streamlit**:
    - Controls: **traffic**, **weights (distance/time/CO_2)**, **fleet cap**, **depot window shift**, **demand spike %**.
    - Button: **Solve** → KPIs + route list, with **A vs B** comparison on one screen.
    - Export: download routes as CSV.

This simulator will make it obvious how policy choices (e.g., "time-reliable" vs "distance-lean") change the plan.

Limitations and assumptions

- I used **Euclidean distance** as a stand-in for road travel time. Real deployments should use road travel times (matrix API or graph).
- I treated **time windows as hard constraints**; feasibility implies 100% on-time. In the future I can add **soft lateness penalties** to study trade-offs.
- Results shown here are on a **small set** (C108, C105, C203). I will extend to a fuller benchmark sweep so conclusions generalize.

Reproducibility (how someone else can run this)

Inputs: Solomon CSVs (e.g., `data/solomon_dataset/C1/C108.csv`).
Outputs: Route CSVs and an `experiments_log.csv` with KPIs.

Baseline example

```
python -u src/baseline_ortools.py --data
data/solomon_dataset/C1/C108.csv \
  --vehicles 25 --capacity 200 --time_limit 30
```

Heuristic examples

```
# 70% distance / 30% time
python -u src/heuristic_m3.py --data data/solomon_dataset/C1/C108.csv
\
  --vehicles 25 --time_limit 30 \
  --first PATH_CHEAPEST_ARC --meta GUIDED_LOCAL_SEARCH \
  --w_dist 0.7 --w_time 0.3 --w_co2 0 --traffic 1.0

# 55% distance / 45% time
python -u src/heuristic_m3.py --data data/solomon_dataset/C1/C108.csv
\
  --vehicles 25 --time_limit 30 \
  --first SAVINGS --meta GUIDED_LOCAL_SEARCH \
  --w_dist 0.55 --w_time 0.45 --w_co2 0 --traffic 1.0
```

Where results land

- Baseline routes: `reports/C108_baseline_routes.csv`
- Heuristic routes: `reports/experiments/<auto-named>.csv`
- KPI log: `reports/experiments/experiments_log.csv`

Key takeaways

- A **solid baseline** matters. With correct constraints, OR-Tools already finds a lower-bound **fleet** and **short distance** on C108.
- **Heuristic tuning** (first-solution + local search + weighted objective) is easy to apply and makes sense conceptually, but on “easy” clustered instances it may **match** the baseline.
- To demonstrate **meaningful improvements**, I will (a) stress the scenario (traffic, depot, fleet cap) and (b) test **broader benchmarks** (C/R/RC).

- A small **interactive simulator** will make these trade-offs obvious to non-technical users and will complete the internship deliverable.