**Project setup, data, and baseline results (Steps 1–4)**

**Goal**

Build a reproducible pipeline to evaluate Vehicle Routing Problem with Time Windows (VRPTW) heuristics on the Solomon benchmark (56 instances). KPIs: (i) total distance, (ii) number of vehicles, with hard capacity and time-window feasibility.

**Dataset**

We use the 56 Solomon VRPTW instances, provided as CSVs with columns:  
CUST NO., XCOORD., YCOORD., DEMAND, READY TIME, DUE DATE, SERVICE TIME.  
Row 1 is the depot. Families encode structure and time-window tightness:

* **C** (clustered customers)
* **R** (random customers)
* **RC** (mix)
* **1** = short scheduling horizon (tight time windows)
* **2** = long scheduling horizon (looser windows)

So:

* **C1, R1, RC1** = short horizon
* **C2, R2, RC2** = long horizon

All instances have vehicle **capacity = 200** and Euclidean travel distance (speed = 1, so travel time = distance). Service times are respected at each stop.

**Folder structure (created once for reproducibility)**

PFE/

configs/

data/

raw/ <- original 56 CSVs

solutions/ <- JSON per instance + summary.csv

reports/ <- aggregated CSV/TXT

figures/ <- plots and charts

scripts/ <- pipeline scripts

**Step 1 — “Freeze” the raw data (why and what)**

**Why:** guarantee the original instances never change (so results are reproducible).  
**What we did:** scripts/freeze\_raw.py set all files in data/raw/ to read-only and logged a message with the number of files.

**Step 2 — Build a manifest (why and what)**

**Why:** keep a machine-readable inventory (filename, family/group, file size, SHA-256) to prove we always solved the same data.  
**What we did:** scripts/make\_manifest.py produced data/manifest.csv with:

* filename, inferred family (C/R/RC),
* size\_bytes, sha256 (content hash),
* head\_preview (first row) for quick sanity checking.

This confirms we have **56 instances** and lets anyone verify file integrity later.

**Step 3 — QA of the raw instances (why and what)**

**Why:** detect corrupted rows, missing values, impossible horizons, etc., before solving.  
**What we did:** scripts/qa\_check.py scanned all CSVs and wrote data/qa\_report.json with, for each instance:

* rows loaded and dropped (e.g., C104 had 1 NaN row dropped → 100 rows loaded),
* detected depot row,
* min/max of time-window bounds,
* total customer demand.

**Result:** all 56 instances are clean and usable.

**Step 4 — Baseline solver + analysis (why and what)**

**4.1 Baseline solver**

**Why:** create a consistent, fast reference to compare future methods against.  
**What we did:** ran scripts/vrptw\_baseline.py on single instances and also with --all.  
The solver uses a simple constructive heuristic (greedy insertion by earliest feasible time) with local improvements inside each route (2-opt on geometry). It enforces:

* vehicle capacity = 200,
* customer time windows (start service ≥ ready time, finish ≤ due date),
* depot time window (overall horizon from depot’s due date).

**Outputs:**

* One **solution JSON per instance** in data/solutions/, e.g.:
* {
* "instance": "C101",
* "vehicles": 10,
* "total\_distance": 878.362,
* "capacity": 200,
* "horizon": 1236.0,
* "feasible": true,
* "routes": [[21,25,26,...],[68,66,...], ...]
* }

Here, each route is a sequence of customer IDs (the depot is implicit at start/end), and feasible confirms all constraints were met.

* A **global table** data/solutions/summary.csv with one row per instance (instance, vehicles, total\_distance, capacity, horizon, feasible).

**Key baseline facts so far:**

* **Feasibility:** 56/56 feasible.
* **C instances** generally need **fewer vehicles** and **less distance** than R/RC.
* **Short horizons** (group “1”) typically force **more vehicles** and/or longer detours than long horizons (“2”).

**4.2 Route plots (qualitative insight)**

**Why:** visually check that routes make sense and understand structural differences.

* **C101** (C1, short horizon): strongly **clustered** stops; many short local tours; total dist ≈ 878; **10 vehicles**.
* **C201** (C2, long horizon): still clustered but with more freedom; some longer arcs are allowed; **10 vehicles**, dist ≈ 1210.
* **R104** (R1): customers scattered; many criss-crossing spokes from depot; **11 vehicles**, dist ≈ 1257.
* **R206** (R2): random + long horizon; **8 vehicles** covering the plane; longer continuous paths; dist ≈ 1495.
* **RC202, RC208** (mixed): mixed patterns; **10** and **9** vehicles respectively; more long “spokes” and cross-region connections.

**4.3 Aggregated analysis & figures**

**Why:** quantify difficulty by family (C/R/RC) and by group (C1,C2,R1,R2,RC1,RC2) to motivate the next experiments.

We ran scripts/analyze\_results.py, which wrote:

* data/reports/family\_summary.csv (means by family),
* data/reports/overall\_summary.txt (quick textual summary),
* Figures in data/figures/:
  + avg\_distance\_by\_family.png
  + avg\_vehicles\_by\_family.png
  + vehicles\_vs\_distance\_scatter.png
  + (extra) avg\_distance\_by\_group.png, avg\_vehicles\_by\_group.png

**What the charts show (high level):**

* **Average total distance by family:** C < R < RC.  
  Interpretation: clustered customers are easier; mixed (RC) are hardest.
* **Average vehicles by family:** C < R ≲ RC.  
  Interpretation: the baseline needs the fewest vehicles on clustered instances.
* **By group:**
  + **C1 < C2** in distance, but both use ~10 vehicles on average.
  + **R1 > R2** in both vehicles and distance → **tight windows** (R1) are clearly harder.
  + **RC1 > RC2** strongly in both metrics → short horizons with mixed geography are the **hardest** set.
* **Vehicles vs total distance (scatter):** positive correlation (more vehicles → more distance). Outliers (e.g., **R101** with 21 vehicles) confirm R1’s difficulty; RC2 points cluster lower in vehicles than RC1.

**Why these steps matter (summary)**

* **Freezing** and **manifest** make the study **reproducible** (no silent file changes).
* **QA** catches data issues before they waste solver time.
* A **baseline** gives a fair reference to beat and a uniform set of feasibility checks.
* **Plots & aggregates** let us understand *where* the difficulty comes from (structure vs time windows) and justify the experimental plan.

**Current status (end of Step 4)**

* Pipeline works end-to-end on **all 56 instances**.
* All solutions are **feasible**.
* Clear difficulty ranking observed: **RC1 > R1 > RC2 ≈ R2 > C2 > C1** (qualitative order from our baseline).
* We have **per-instance JSONs**, a consolidated **summary.csv**, and multiple **figures** and **reports** ready to include in the thesis.

**What’s next (preview of Step 5)**

* Implement a stronger method (e.g., **OR-Tools** VRPTW or a metaheuristic like **Tabu Search** / **Simulated Annealing** with cross-route moves).
* Keep the same pipeline and metrics, rerun on all 56, and **compare** against the baseline using the same reports and plots.
* Optional: add run-time limits, seed control, and repeats to report **average and variance**.

Files produced so far that you can cite in your manuscript:

* data/manifest.csv
* data/qa\_report.json
* data/solutions/\*.json, data/solutions/summary.csv
* data/reports/family\_summary.csv, data/reports/overall\_summary.txt
* data/figures/\*.png (distance/vehicles by family & group, scatter, route plots)

**Step 5 — Exact solver (OR-Tools) vs. baseline heuristic.**  
We integrated Google OR-Tools (CP-SAT) to solve the Solomon VRPTW instances (C, R, RC with short/long horizons). The model minimizes a weighted objective 10000 × (# vehicles) + total distance. Distances are Euclidean and time windows are enforced via a time dimension with waiting.

With a 10-second time limit and Guided Local Search (GLS), OR-Tools produced feasible solutions for all **C** instances and most **R/RC** instances. On the solved cases, route lengths were consistently shorter than our baseline heuristic: the family-level boxplots of % distance change are centered well below zero, with several instances showing 30–35% reductions. A sample of the largest gains includes R206 (-35.6%), C108 (-34.7%), RC206 (-34.5%), R203 (-33.1%), and R103 (-32.9%).

Six short-horizon instances (R101, R102, R105, RC101, RC102, RC105) timed out at 10 s with no solution; these are known to be the tightest cases. Increasing the time limit to 60–120 s and trying alternative local-search metaheuristics (Tabu, Simulated Annealing) is expected to resolve feasibility while preserving the distance advantage.

We saved a comparison table (data/reports/method\_comparison.csv) with vehicles, distances, and deltas for every instance, plus figures showing (i) the percent distance improvement by family and (ii) the count of timeouts by family. This establishes that OR-Tools gives a strong improvement in route quality on clustered, random, and mixed datasets, especially once adequate time is allowed for the hardest short-horizon cases.

**Step 1 — Goal & success metrics (abstract)**

We framed the PFE objective as “build high-quality VRPTW routes on the original 56 Solomon CSV instances, then improve robustness to variability.” Success is measured by either (a) improving on-time service by ~8–15% at ≤3% extra cost, or (b) cutting total distance/cost by ~5% while keeping on-time unchanged, with practical runtimes (≤30 min on 100-customer sets). This step fixes the scope, KPIs (distance, #vehicles, on-time %, tardiness, runtime), and the evaluation rules we follow later.

**Step 2 — Data inventory & freezing (abstract)**

We consolidated all 56 original CSVs under data/raw/, set them read-only, and produced a manifest (data/manifest.csv) plus a lightweight QA profile (data/qa\_report.json). The manifest records file family (C/R/RC), size, checksum, and a header preview; the QA confirms row counts, candidate depot rows, and demand/time ranges. Nothing was edited—this step locks the dataset and makes every later result reproducible (same inputs, same checksums).

**Step 3 — Schema & loader validation (abstract)**

We standardized one robust parser for your exact headers (CUST NO., XCOORD., YCOORD., DEMAND, READY TIME, DUE DATE, SERVICE TIME) and validated depot detection (the unique row with zero demand/service). Families and horizons were recognized automatically (C, R, RC × short/long), so the same loader works across all 56 files without per-file tweaks. This guarantees that every downstream script (distance matrices, solvers, plots) reads the data identically.

**Step 4 — Deterministic baseline & diagnostics (abstract)**

Using the loader, we solved each instance with a simple deterministic baseline (greedy/meta-light) and wrote per-instance JSON solutions plus a global table at data/solutions/summary.csv. We also generated quick analytics by family (data/reports/family\_summary.csv, overall\_summary.txt) and route/metric visualizations (avg\_distance\_by\_family.png, avg\_vehicles\_by\_family.png, vehicles\_vs\_distance\_scatter.png, plus optional route plots). This gave us a clean reference (feasibility, typical distances, fleet sizes) and helped sanity-check time windows and capacity behavior before heavier methods.

**Step 5 — OR-Tools baseline & method comparison (abstract)**

We implemented an OR-Tools VRPTW solver (RoutingModel + TimeDimension), fixed type/units issues (vehicle-penalized objective vs pure distance), and ran it across all instances, saving results to data/solutions\_ortools/summary.csv and per-instance JSONs. Then compare\_methods.py merged both summaries into data/reports/method\_comparison.csv and plotted performance deltas (pct\_distance\_change\_by\_family.png) and feasibility gaps (no\_solution\_by\_family.png). Result: the OR-Tools plans substantially reduced total distance versus the greedy baseline (often with equal or fewer vehicles), establishing a stronger deterministic reference for the tuning phase.