perfect — we’ll use your original 56 CSVs exactly as they are (no edits), and build everything around them. here’s your start-to-finish PFE plan, step-by-step, with the why for each step. once you read this, we can jump straight into execution.

1) project goal & success metrics

what: optimize VRPTW routes that stay on time under variability, using only your original Solomon-style CSVs.

why: clear targets keep the project focused and evaluable.

success (pick one):

+8–15% on-time vs deterministic baseline at ≤+3% cost, or

−5% cost at the same on-time rate, runtime ≤ 30 min for 100-customer sets.

2) data inventory & freezing (read-only originals)

what: put all 56 CSVs in data/raw/ and treat them as read-only. Create data/processed/ for any derived files (distance/time matrices, scenario files).

why: preserves your originals; everything we compute is reproducible and separable.

3) schema agreement (parser for your exact headers)

what: write a single parser that reads the original headers you showed (e.g., CUST NO., XCOORD., YCOORD., DEMAND, READY TIME, DUE DATE, SERVICE TIME). Detect the depot row (usually demand==0 & service==0).

why: one robust loader handles all 56 files identically and documents the data contract.

4) units & baseline time matrix (from original coordinates)

what: compute a baseline travel-time matrix directly from coordinates (Euclidean). In Solomon, distance units double as time units; we’ll use that default (1 unit ≈ 1 time unit).

why: this gives a deterministic baseline that respects your original instances without adding external data.

5) deterministic VRPTW baseline (OR-Tools)

what: solve each instance with OR-Tools (RoutingModel + TimeDimension): capacity, time windows, service time, depot, number of vehicles as per the instance.

KPIs: total distance/time, #vehicles, % on-time (strict windows), avg/95th tardiness, runtime.

why: establishes a reference to beat; validates our parser and feasibility handling.

6) uncertainty layer (needed because originals are deterministic)

what: define plausible variability around the baseline times to evaluate robustness (no edits to originals; we just simulate test conditions). Two options (we can do both):

Profile noise: peak vs off-peak multipliers by hour band (e.g., +0%, +20%, +35%), plus small random noise.

Log-normal arc noise: for each arc, mean = baseline time, CV = 10–25%; add a global traffic factor per scenario so adjacent arcs co-move (correlation).

why: your CSVs are deterministic; to study “robust” routing, we need test scenarios that mimic real variation without altering originals.

7) scenario generator (dependence-aware)

what: generator that outputs K scenarios per instance (start with K=32), preserving correlation via a shared traffic state per time band (blocks) + per-arc residual noise.

why: independent link noise underestimates route variance; correlated scenarios are more realistic for lateness risk.

8) fast robust method (quantile-buffer) — “vite” baseline

what: build a conservative time matrix by inflating baseline times (e.g., 0.90 quantile ≈ baseline × 1.2, or median + fixed buffer per leg/stop). Solve with OR-Tools.

why: extremely quick to implement, usually big boost to on-time with small cost impact; serves as your first strong method.

9) stochastic optimization method (SAA, small K)

what: implement Sample Average Approximation (two-stage flavor) with mini-batch evaluation inside a metaheuristic (e.g., ALNS/HGS): routes shared across scenarios; moves scored by expected cost + tardiness across a small batch of scenarios (rotate batches).

why: captures variability directly and often beats simple buffers while staying scalable.

10) robust budget method (Γ-robust)

what: add a per-route delay budget Γ (assume up to Γ arcs hit worst deviation). Enforce buffers accordingly during feasibility checks.

why: one intuitive knob for conservatism; easy to compare vs quantile and SAA.

11) (optional) chance-constraint surrogate

what: enforce that arrival upper quantiles (built from scenario samples) do not exceed due times for a target service level (e.g., 95%).

why: aligns directly with “on-time probability ≥ 95%” guarantees; good if you want SLA-style results.

12) evaluation protocol (fixed & fair)

what: for each method (Deterministic, Quantile-buffer, SAA, Γ-robust), on each instance:

Use common random numbers: the same 100 test scenarios to evaluate every plan.

Report KPIs: cost, on-time %, avg/95th tardiness, #vehicles, runtime.

Sensitivity: K ∈ {16, 32, 64}; buffer factor ∈ {1.1, 1.2, 1.3}; Γ ∈ {0,1,2}.

why: apples-to-apples comparison and statistically meaningful summaries.

13) experiment matrix (cover all your data efficiently)

what: run all 56 files grouped by family (C1/C2, R1/R2, RC1/RC2). At minimum:

Phase A (fast): Deterministic vs Quantile-buffer on all 56.

Phase B (deeper): Add SAA (K=32) and Γ-robust on a representative subset (e.g., 2 per family), then roll out the best settings to all.

why: quick full-coverage results first; deeper methods where it matters; then scale.

14) code layout & naming (so everything is tidy)

pfe/

data/ raw/ processed/

io/ (parse\_solomon.py, build\_matrices.py)

vrp/ (solve\_deterministic.py, solve\_quantile.py, solve\_saa.py, solve\_robust\_gamma.py)

sim/ (make\_scenarios.py, traffic\_profiles.py, evaluate\_plan.py)

eval/ (aggregate\_results.py, plots.py)

configs/ (exp.yaml)

README.md requirements.txt

why: clean separation: parsing, solving, simulation, and evaluation.

15) reporting & deliverables

what:

Tables per instance + family aggregates (mean ± std).

Plots: cost vs on-time trade-off; tardiness distributions; runtime bars.

Short report (10–15pp): problem, data, methods, KPIs, results, sensitivity, conclusions.

why: a professional PFE package your jury can read quickly.

16) acceptance checklist

✅ All 56 originals used untouched (read-only).

✅ Deterministic baseline reproduced.

✅ At least one robust method (quantile-buffer) and one stochastic method (SAA) compared fairly.

✅ Scenario generator documented (how correlation is modeled).

✅ Clear gains on on-time % or cost, with runtimes reported.

17) risks & mitigations

Too slow on big instances: use smaller K (16–32), mini-batch evaluation, and cache transit computations.

Noisy results: increase test scenarios to 200 for evaluation only (not for solving).

Windows too tight: allow soft tardiness with a penalty for analysis; still report strict on-time %.

if this plan looks good, we’ll start with Step 3 (parser + deterministic baseline) on one file (e.g., C101.csv) and then roll it to all 56.