docs/report.md (paste this in `docs/report.md`, then export to PDF)

```
# Polar Parcels — Lab 2 Report

**Name:** _<Your Name>_
**Course/Section:** _<Course>_
**Date:** _<Date>_

---

## Design Rationale (≤ 1 page)

**Suitability & Simplicity.**
I modeled the MVP directly from the requirements: load a workday, compute quotes,
assign feasible (driver, vehicle) pairs, and print/export outputs. I kept the
structure minimal and readable:
- **Domain models** are plain types (`Job`, `Vehicle`, `Driver`, `Weather`,
`Assignment`, `Quote`).
- **Pricing** is a small engine that composes **pluggable rules** behind an
interface (`IPriceRule`). This cleanly matches “base + per-km + surcharges +
multipliers”.
- **Assignment** is a single strategy (`GreedyAssignment`) behind
`IAssignmentStrategy`, focused only on feasibility and a cheap choice among
feasible options.
- **I/O** (JSON load and printing) is isolated from domain logic so
pricing/assignment stay framework-free.
- No DI container: wiring is done explicitly in `app.ts` for clarity at MVP scale.

**Information Hiding & Cohesion.**
Each module does one job. Rules return only a `PriceEffect` (`{ add?, multiply?
}`), hiding how the engine attributes costs. The assignment strategy contains
constraint checks and ETA math, but it doesn’t know anything about file I/O or
CLI. Public APIs are small and purposeful.

**Loose Coupling & DI.**
The engine depends on `IPriceRule[]` and the strategy depends on a `quoteFor(job,
vehicle)` callback; both are injected in `app.ts`. Adding new pricing behavior or
swapping strategies doesn’t ripple through other code.

**Extension Seam (intentionally left).**
New pricing rules or vehicle policies fit behind interfaces. I proved this by
adding `LakeEffectAlertRule` as a separate class and **only registering** it in
`app.ts`—no edits to the calculator.

---

## UML (current MVP)
```

~~~
> Include the image `docs/uml.png` in your PDF, or render this Mermaid block if
your PDF pipeline supports Mermaid.

```mermaid
classDiagram
  direction LR

  class Job
  class Vehicle
  class Driver
  class Weather

  class PricingContext {
    +job: Job
    +vehicle: Vehicle
    +weather: Weather
    +baseFee: number
  }
  class PriceEffect { +add?: number +multiply?: number }
  class Quote { +jobId: string +vehicleId: string +price: number +breakdown: {...}
}
  class Assignment { +jobId: string +driverId: string +vehicleId: string
+etaMinutes: number +price: number }
  class DayData { +drivers: Driver[] +vehicles: Vehicle[] +weather: Weather +jobs:
Job[] }

  class IPriceRule { <<interface>> +apply(ctx: PricingContext): PriceEffect }
  class BasePricingEngine { -rules: IPriceRule[] +quote(ctx): Quote }

  class IAssignmentStrategy { <<interface>>
+assign(jobs,drivers,vehicles,weather,quoteFor): Assignment[] }
  class GreedyAssignment { +assign(...): Assignment[] }

  class Loader { +loadDay(dataDir): DayData }
  class Printer { +printManifest(assignments,outDir): string
+exportSummary(assignments,outDir) }

  BasePricingEngine o--> IPriceRule : rules (DI)
  GreedyAssignment ..|> IAssignmentStrategy
  Loader --> DayData
  Printer --> Assignment
~~~

Correctness Mapping

Pricing Order (implemented): Base fee ($5) Distance cost by vehicle: car $0.80/km, snowmobile $0.60/km, cargo-bike $0.35/km Weather surcharge: +10% if conditions === "snow" or tempC < -10 Lake-effect alert: +8% if weather.alerts contains "lake-effect-snow" (extension rule) Priority multiplier: ×1.25 when priority === "express"

Assignment Constraints:

capacityKg ≥ job.kg rangeKm ≥ job.distanceKm × 2 × 1.2 (round-trip + 20% buffer) Terrain fit: backcountry: car disallowed, snowmobile allowed, cargo-bike disallowed by default urban: all vehicles allowed ETA: one-way time = distanceKm / speed_kph (car 50, snowmobile 35, cargo-bike 18) → minutes rounded.

Separation of Concerns

Domain has no file paths/printing. Pricing and assignment are pure computations. I/O (io/loader.ts, io/printer.ts) moves bytes and formats text, nothing more. Composition root (app.ts) wires concrete rules/strategy and exposes CLI commands. This keeps domain logic UI-agnostic.

Extension Proof

What I added: LakeEffectAlertRule (+8% multiplier if alerts include "lake-effect-snow"). How it was integrated: I created a new class implementing IPriceRule and only registered it in the engine construction list in app.ts. No edits to BasePricingEngine or other rules were required, demonstrating a true plug-in seam.

Observed effect with the sample data:

J-1001 cargo-bike with the alert → ~$11.14 (without the rule) → ~$10.31 J-1002 snowmobile (express) with the alert → ~$83.16 (without the rule) → ~$77.00

Short Reflection

Where coupling is loosest. The pricing pipeline: rules are independent, unaware of each other, and are composed by the engine. The engine depends only on the IPriceRule interface. This allows adding/removing rules by registration with zero ripple effects.

Trade-off made. I chose a simple greedy assignment for clarity and speed. It is easy to reason about and extend with more feasibility checks, but it may not minimize global cost. For the MVP and rubric goals (suitability, simplicity, clear seams), the trade-off is appropriate. A future version could swap in a more optimal strategy (e.g., Hungarian algorithm over a feasibility graph) without changing the CLI or pricing code.

Future improvements (not required now).

Cache quotes to avoid recompute in assignment loops. Add logs/metrics per rule for transparency. Support per-region policies by injecting rule sets based on config.

Demo / Test Script

import data price all assign all print manifest export summary quit