Design Decisions

1) Overview

Goal: Two-tier application that exposes a REST API over a sales dataset and a Dash UI consuming that API.

High-level architecture

- **Backend (FastAPI):** /health, /meta, /data, /metrics
  - Serves aggregated metrics (category, store, article, time series, predicted values)
  - Performs all filtering (date range, top) server-side for consistency
- **Frontend (Dash/Plotly):**
  - Reads from /metrics with query params (start, end, top)
  - Renders KPIs, bar/pie charts, and time series
- **Containerization (Docker Compose):**
  - Separate services for api (port 8000) and dashboard (port 8050)

This separation keeps concerns clean: the API owns business logic, the UI focuses on presentation.

---

2) Technology Choices (and Why)

- **FastAPI** (REST API)
  - Modern, fast
  - Great dev ergonomics → maintainability
- **Pandas**
  - Ideal for CSV ingestion and straightforward groupby/aggregation
- **Plotly Dash**
  - Python-native dashboards
  - Interactivity with minimal code → faster implementation, good readability
- **Plotly Express**
  - Concise, declarative chart API (less code, fewer bugs)
- **Docker & docker-compose**

    - Reproducible runtime
    - Clear boundary between services
    - One command to run everything

---

3) API Design

Endpoints

- GET /health — liveness check
- GET /meta — schema metadata for debugging/inspectability
- GET /data?start&end — preview of raw, filtered data (capped to 50 rows)
- GET /metrics?start&end&top
    - **Server-side filtering** by date range
    - **Server-side Top-N** for category/store/article (top=0 means “all”)
    - **Aggregations returned:**
        - sales_by_category
        - sales_over_time
        - sales_by_store
        - sales_by_article
        - predicted_over_time
        - KPIs: total_rows, total_sales, avg_sales_per_day, distinct_articles

---

4) Frontend (Dash)

- **Inputs:** DatePickerRange, Top-N dropdown (3/5/10/All), reset button for date
- **KPI bar:** total sales, records in period, average per day, number of articles
- **Charts:**
    - Bar + Pie: Category
    - Line: Sales over Time
    - Bar + Pie: Store
    - Bar + Pie: Article
    - Line: Predicted over Time

5) Maintainability

- **Readable, typed Python** with clear constants for column names
- **Small, focused endpoints** and minimal branching
- **English comments** and a **README** with structure and usage
- Separation of concerns (UI vs aggregation) → easier to change either layer independently
- Dockerized services

6) Scalability

Current solution is optimized for the assignment (single CSV, in-memory). Clear path to scale:

- **Move to a DB** (e.g., Postgres):
  - Store raw data in tables with proper indexes (date, category, store, article)
  - Replace pandas groupbys with SQL aggregations or materialized views
- **Horizontal scaling**
  - Run multiple API instances
- **Pre-aggregation**
  - Daily ETL to per-day/per-category/store/article tables reduces query time

7) Performance

- **Single aggregated response** keeps UI fast and reduces chattiness
- **Date filtering before groupby** reduces compute

10) External Libraries Justification

- **FastAPI**: high performance, built-in validation & docs, maintainable codebase
- **Pandas**: pragmatic for CSV analytics and rapid development
- **Plotly Dash + Express**: fast to build interactive analytics, minimal code, good defaults

- **Docker Compose**: reproducible, portable multi-service setup

---

Mapping to Requirements

- **Two-tier app ✓** REST API + Dash UI
- **At least 5 metrics ✓** KPIs + category/store/article + time + predicted time series
- **Date range filter ✓** applied server-side and used by UI
- **Document design decisions ✓** this file
- **Smaller commits ✓** repo history structured in incremental, testable steps