CityRetail Data Engineering Term Project

ITC 6050 - DATA ENGINEERING

Spring Term 2025

Panagiotis Vergas 283051

Contents

1. Data Understanding

1.1 Project Context

The goal of this project is to produce a clean, reliable, and analysis-ready data warehouse using a star schema implemented in PostgreSQL. Cleaned and standardized data from raw CSVs will populate dimension and fact tables with enforced foreign key constraints. Business-relevant KPIs (e.g., Revenue, Quantity Sold, Profit Margin) will be computed using SQL views or materialized views. The data will then be visualized through a Power BI dashboard. This enables users to track trends, compare store performance, and make data-driven decisions.

1.2 Data Sources and Description

The following raw CSV files were provided:

File Name	Description
`sales.csv`	Transaction-level sales data
`products.csv`	Metadata for each product
`stores.csv`	Store metadata with store identifiers, names, and city names, regions
`calendar.csv`	Date dimension file
`cities_lookup.csv`	Mapping table used to standardize inconsistent or malformed city names.

These datasets collectively support a star schema, with sales.csv serving as the Fact table and the others as Dimensions.

1.3 Initial Observations

- **Missing Data:**
  - No standard or hidden null values (NaN, empty strings, "NULL") were found in any dataset using custom inspection functions.
- **Duplicates:**
  - No duplicate rows were detected across all datasets during EDA.

- **Inconsistent City Names:**
  - The stores.csv file may contains raw city names that may be misspelled, inconsistently cased, or contain trailing whitespace.
  - Examples:
    - "athens" → lowercase
    - "Athens " → trailing space
    - "Heraklio" → misspelling of "Heraklion"
  - A city consistency check was implemented to compare values in stores['City'] against approved raw values in cities_lookup['RawCity'], using lowercase + whitespace trimming for comparison only.
  - The cities_lookup.csv serves as a **controlled mapping reference** and can be **expanded** in the future to include additional raw variants or new cities as business needs evolve.
- **Referential Integrity Checks:**
  - Confirmed all foreign keys in `sales.csv` (`ProductID`, `StoreID`, `DateID`) exist in their corresponding dimension tables.
  - No mismatches found, ensuring a clean load into the OLAP schema.

1.4 Key Design Principles

- **Audit before Clean:** City names were only analyzed — not modified — during EDA. All standardization occurs later in a centralized transformation pipeline.
- **Modular Pipeline:** Code is structured in reusable components (src/, notebooks/, etc.).
- **Robust Null Handling:** Explicitly checked for both visible and hidden missing values.
- **Business-Aware Analysis:** EDA focused on features that directly affect business KPIs and reporting logic

1.5 Expected Outcome

The ETL pipeline is designed to:

- Ingest and clean raw datasets

- Standardize inconsistent values (e.g., city names)
- Load data into a PostgreSQL star schema with enforced foreign key constraints
- Enable efficient KPI calculation and interactive dashboard reporting

2. Pipeline Planning

2.1 Overview

The goal of the pipeline was to transform loosely structured retail data into clean, analysis-ready datasets that can be reliably loaded into a PostgreSQL star schema and consumed by downstream BI tools like Power BI. To ensure maintainability and auditability, we implemented a modular Extract-Transform-Load (ETL) process using Python, with optional containerization using Docker and Docker Compose.

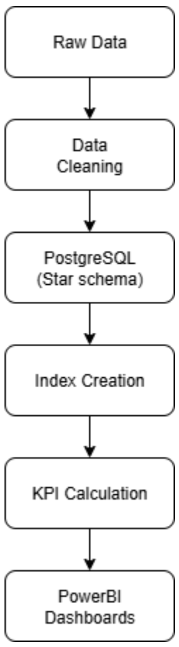


Figure 1. End-to-End Data Pipeline Architecture.

2.3 Handling Missing Values and Inconsistencies

- **Null Handling**
  - Explicit checks were implemented to detect both standard and hidden null values (e.g., "NULL", empty strings, "NaN").
  - None of the datasets contained missing values in their raw form, but checks were retained for robustness.

- **Referential Integrity Checks**
    - Ensured that all ProductID, StoreID, and DateID values in sales.csv match corresponding entries in products.csv, stores.csv, and calendar.csv.
    - This step prevents orphaned foreign keys during database loading.

- **City Name Standardization**
    - City names in stores.csv were often inconsistent (e.g., "Athens ", "athens", "Heraklio").
    - These were not corrected manually during EDA. Instead:
        - We built a function standardize_city_names() that merges stores.csv with cities_lookup.csv.
        - It replaces raw city names with their business-approved versions from the StandardCity column.
        - Unmatched rows are logged as warnings for traceability.

- **Date Parsing and Enrichment**
    - The calendar.csv dataset appeared clean but was still parsed using `pandas.to_datetime()` to enforce format consistency.
    - Two additional fields were added:
        - WeekNumber: ISO week number (1–53)
        - IsWeekend: Boolean flag indicating Saturday or Sunday
    - These enrichments support week-level trend analysis and weekend-vs-weekday comparisons in reporting.
    - Additional enrichments such as quarterly sales analysis aligning with financial reporting cycles, or if isholiday for analyzing revenue impact of public holidays or events, etc… could enable more meaningful insights

Special Handling for City Names, Dates, Numbers

Element	Handling Strategy
City Names	Merged with cities_lookup.csv for safe replacement of raw names with standardized ones.
Dates	Parsed explicitly using pd.to_datetime with errors='coerce'. Enriched with WeekNumber and IsWeekend.
Quantities/Revenue	No format issues found. Numeric columns retained original types but are validated during ingestion.
Currencies	All revenue is in a single currency — no conversion required, but column was explicitly validated as float.
String Fields	No encoding issues or inconsistencies were detected in products.csv or calendar.csv.

2.4 Tools, Technologies, and Target Database

The project used Python for ETL, PostgreSQL (via Docker) for OLAP storage, and Power BI for business-facing dashboards. Modular scripts and Docker Compose ensured reproducibility, while SQL views and Power BI interactivity provided clarity.

2.5 Directory Configuration and Path Management

To maintain a clean and portable structure across environments, a dedicated configuration script (config.py) was implemented to define all project paths. This script dynamically builds paths using the Python os module and an optional DATA_PATH environment variable. It creates key folders like data/raw/, data/cleaned/, and data/logs/ if they do not already exist. This approach ensures:

- Environment flexibility — the pipeline runs seamlessly whether deployed locally or on a cloud environment.
- Centralized structure — all file paths are defined in one place, reducing code duplication and preventing path-related errors.
- Automatic setup — required directories are created at runtime, improving robustness and simplifying first-time execution.

This method strengthens reproducibility and is especially valuable when packaging the pipeline for containerized or team-based execution environments.

2.6 Docker Integration

- Used a multi-container setup with:
    - A PostgreSQL container for hosting the cleaned OLAP schema
    - A Python container for executing main.py, the ETL script
- Docker ensures the project runs identically on any machine and simplifies future testing, deployment, and teamwork.
- PostgreSQL credentials, database name, and ports were managed using environment variables.

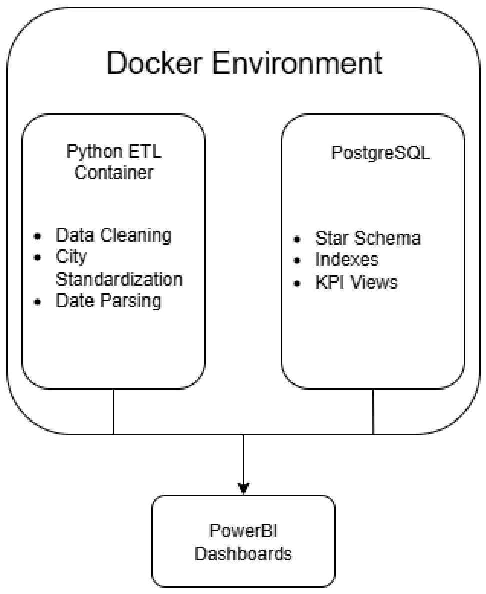


Figure 2. Docker-Based Architecture for ETL and KPI Delivery

2.7 Design Summary

- ETL modules are reusable and testable (`standardize_city_names()`, `parse_calendar_dates()`)
- City name standardization and date enrichment are handled in clean_data.py
- All cleaned files are saved to data/cleaned/ and then loaded into the database
- Docker enables fully portable execution: `docker-compose up --build` spins up both the DB and the ETL pipeline
- Only necessary files (`calendar.csv`, `stores.csv`, `products.csv`, `sales.csv`) are output for loading. Mapping tables like `cities_lookup.csv` remain internal.

2.8 Future Consideration for Standardized Development

A `.devcontainer/` setup enables a fully containerized VS Code environment, ensuring consistency across machines. While this approach benefits team collaboration and onboarding, it was intentionally excluded here to maintain simplicity.

The existing Docker Compose setup already provides full reproducibility for running the ETL and database containers. For solo development and classroom settings, this streamlined approach avoids the added complexity of containerized editors and remains robust for future scaling.

3. Reliability & Performance

3.1 Reliability Strategy

The pipeline is designed with modularity, reusability, and auditability in mind. Each phase (Extract, Transform, Load) is implemented in independent Python functions and executed from a single orchestration point (`main.py`). Logs are written for every major step to enable traceability and debugging.

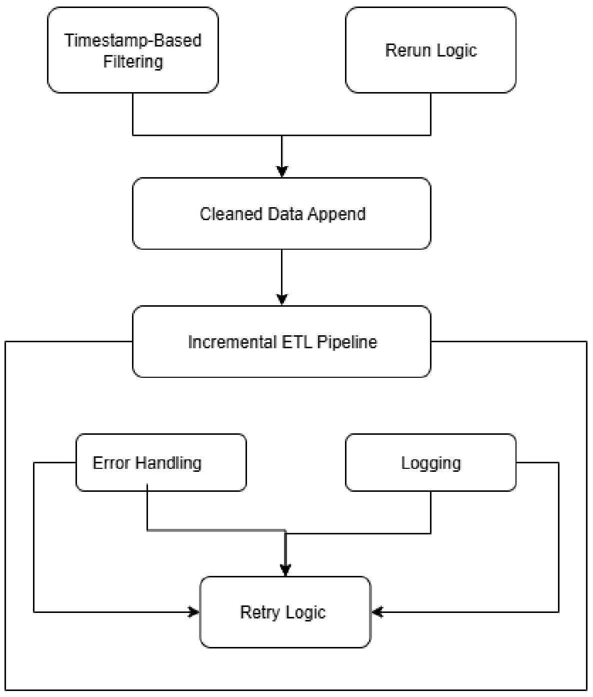


Figure 3. Incremental ETL Pipeline with Error Handling and Retry Logic

Incremental Loads and Reruns

Although the initial implementation uses full loads, the pipeline got to support **incremental loading** via future extensions:

- **Raw Timestamp-Based Filtering**:
    - New records can be filtered during extraction using a LastUpdated timestamp (e.g., sales.csv with transaction dates).
    - Only rows after a saved checkpoint would be transformed and appended.
- **Cleaned Data Append**:
    - In main.py, a future flag (e.g., --incremental) could trigger logic to:
        - Load only new rows
        - Clean and standardize them
        - Append them to existing PostgreSQL tables
- **Reruns**:
    - Since save_cleaned_dataframes() saves outputs deterministically to /data/cleaned/, re-running the pipeline regenerates the same results unless upstream data changes.
    - Docker's volume setup enables complete reset of the environment via docker-compose down -v and re-run with --build.

Incremental logic follows same methodology, but extended to support:

- Incremental logic
- Primary key-based filtering
- Optional UPSERT behavior
- Auditability per insert step

Error Handling and Logging

- **Logging**
    - A custom logger is implemented via `logger.py`, configured in every cleaning function.
    - Warnings are issued for:

        - Unmapped city names (standardization issues)
        - Unparsable dates in the calendar
        - Failed file loads
    - All logs are terminal-visible redirected to a file (`etl.log`) for auditability.

- **Retry Logic**
    - While retries were not required in this static data version, the design allows for:
        - Wrapping DB inserts with retry decorators or exponential backoff
        - Adding try/except blocks in ETL steps (e.g., `parse_calendar_dates`, `standardize_city_names`)
    - With Docker in place, retry strategies can also be extended using tools like Airflow or a custom retry wrapper script.

- **Future Integration with Queues**
    - The ETL is designed as a Python module, which allows easy wrapping inside:
        - **Redis Queue (RQ)** for task scheduling
        - **RabbitMQ** or **Celery** for distributed processing or real-time updates
        - This sets the stage for scalable production-level ingestion.

3.2 Parallel Processing Potential

The CityRetail pipeline was initially designed to support parallel loading using Python's ThreadPoolExecutor to speed up the ETL process. However, during testing, this approach led to PostgreSQL deadlocks. Even though different tables were being loaded in parallel, their foreign key relationships caused locking conflicts, resulting in failures during table truncation or deletion.

To ensure data integrity and avoid such concurrency issues, parallelism was replaced with safe, serial loading. Instead of using TRUNCATE, the pipeline now applies DELETE FROM table operations in a fixed referential order—dimension tables first, then the fact table. This design avoids locking conflicts, maintains referential integrity, and ensures the pipeline can be rerun reliably without manual cleanup.

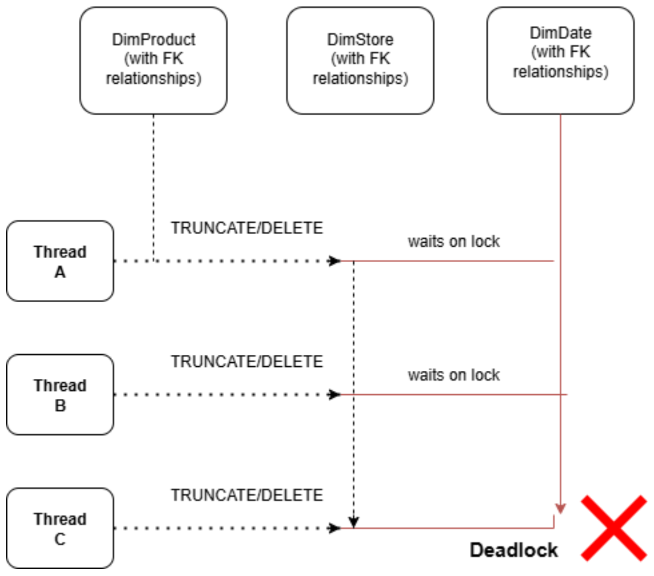


Figure 4. PostgreSQL Deadlock Caused by Concurrent TRUNCATE on FK-Linked Tables

- Multiple threads tried to truncate tables like `DimProduct`, `DimStore`, and `DimDate` at the same time.
- PostgreSQL detected **circular wait conditions** and raised `psycopg2.errors.DeadlockDetected`.

- **Final Implementation: Safe Serial Loading**

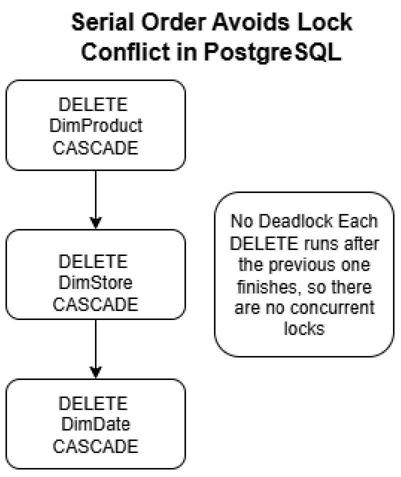

Serial Order Avoids Lock Conflict in PostgreSQL
DELETE DimProduct CASCADE
DELETE DimStore CASCADE
DELETE DimDate CASCADE
No Deadlock Each DELETE runs after the previous one finishes, so there are no concurrent locks

Figure 5. Serial Table Loading with DELETE Avoids Deadlocks in PostgreSQL

To ensure data integrity and avoid deadlocks:

- **Parallelism was removed**, and tables are now loaded **sequentially** in referential order (dimensions first, then fact).

- Instead of TRUNCATE, the pipeline uses **DELETE FROM table** during load_csv_to_table() and clear_table().

This combination avoids locking issues while maintaining safe resets before insert.

While parallelism offers speed, it introduced deadlocks due to interdependent foreign key constraints. The current serial + DELETE strategy is slower but ensures consistent, safe loads, making it more suitable for this pipeline's reliability goals. Parallel loading can be revisited in the future with added safeguards.

Future Outlook

If performance becomes a bottleneck:

- Parallel loading can be reintroduced by replacing all DELETE operations with **batch deletes** and managing insert concurrency more carefully.
- Alternatively, row-level staging and partitioned loads can be implemented with retry-safe mechanisms.

3.3 Monitoring and Bottlenecks

While production grade monitoring tools (e.g., Prometheus, Grafana) were out of scope, the pipeline incorporates the following monitoring strategies:

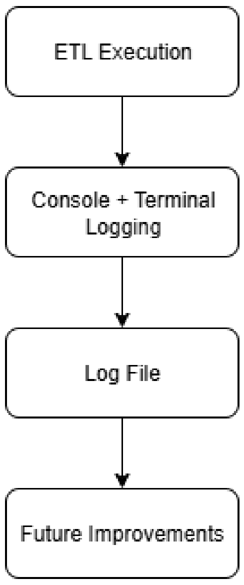


Figure 6. Logging and Monitoring Architecture in the ETL Pipeline

- **Console-based logging**: Each major step prints status messages and warnings

- **Error logs**: Any exceptions (file not found, parsing issues) are logged with detail
- **Docker isolation**:
    - Ensures environment-specific failures (e.g., port conflicts, dependency issues) don't affect host machine
    - PostgreSQL logs are visible via docker logs cityretail-db

In a future setup, these mechanisms could be extended to include:

- ETL run status reports
- Log shipping to ELK stack (Elasticsearch, Logstash, Kibana)
- Task monitoring via Airflow UI or Prefect Cloud
-

Summary Table

Feature	Current Support	Scalable Design
Logging	Terminal logs via logger	Easily extensible to file-based logging
Retry logic	Planned via decorators	Compatible with Python retry libraries
Incremental loads	Planned (via date filtering)	Designed for timestamp-based deltas
Parallel processing	Not yet implemented	Supported via function-level modularity
Monitoring/Bottleneck detection	Manual via Docker/logs	Extensible with Prometheus or Airflow
Queue/message integration	Not active	Future-ready for implementation

4. Security & Compliance

4.1 Sensitive Data Inspection

An audit of the CityRetail datasets confirmed that no personally identifiable information (PII) or sensitive financial data is present. All datasets contain operational or metadata only:

Dataset	Sensitive Data?	Notes
`sales.csv`	No	Transaction-level info — no customer names or user details
`stores.csv`	No	Store and city metadata — no addresses or geo-coordinates
`products.csv`	No	Product metadata — no vendor or confidential pricing details
`calendar.csv`	No	Standard date fields — no behavioral or timestamp-level data
`cities_lookup.csv`	No	Internal mapping for standardization — not exposed to end-users

No masking or encryption was needed, but the pipeline remains extendable for future compliance scenarios involving customer data.

4.2 Use of Environment Variables for Security

A feature of the project is the use of `.env` and `.env.example` files for secure configuration management:

- All database credentials are accessed via `os.getenv()` no hardcoding in scripts.
- A sample `.env.example` is provided for setup consistency without exposing secrets.
- The real `.env` file is excluded from Git via `.gitignore`.
- Docker Compose uses the `.env`_file directive to inject credentials into the Python and PostgreSQL containers at runtime.

It is important not to expose credentials to public repositories.

Alternative Use of Environment Variables for Security

In addition to using a .env file for local development with Docker, environment variables can also be set permanently via the system settings. On Windows, users can navigate to **"Edit the system environment variables"** and manually add credentials like `POSTGRES_USER, POSTGRES_PASSWORD, DB_USER, and DB_PASS` under the **User variables** section.

This method ensures that the variables persist across terminal sessions and reboots, making them accessible to scripts, Docker containers, and CI/CD pipelines without requiring a .env file.

4.3 Recommended Access Controls for Production

To scale the CityRetail pipeline securely, the following access control strategies are recommended:

Control Type	Description	Purpose
RBAC	Use separate DB roles like etl_writer and dashboard_reader	Prevents privilege escalation and accidental changes
Secrets Management	Store credentials using Docker Secrets or cloud vaults	Avoids exposing sensitive info in .env files
Encrypted Connections (TLS)	Enforce SSL/TLS for all PostgreSQL and ETL communication	Protects credentials and data in transit
JWT Authentication	Authenticate users via JSON Web Tokens for APIs or dashboards	Enables secure, stateless access control

4.4 Key Takeaway

While the current pipeline processes only non-sensitive data, it already follows good security practices .env-based configuration, modular logging, Git hygiene, and isolated credentials. These decisions make it easy to scale, supporting future inclusion of customer data, cloud deployment, and compliance audits.

5. Iteration & Automation

5.1 Iteration Progress

The CityRetail ETL pipeline began as a simple, manually-triggered process with sequential table loads, no retry handling, and minimal logging. As development progressed, the focus shifted toward robustness, modularity, and scalability.

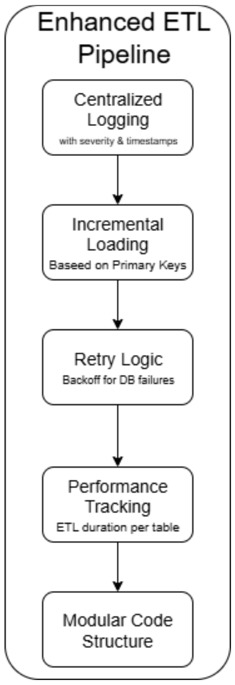


Figure 7. Key Enhancements in the ETL Pipeline

Key enhancements included:

- Centralized logging with timestamps and severity levels (logs/etl.log)
- Incremental loading based on primary keys to avoid duplicate inserts
- Retry logic with exponential backoff (`wait_for_postgres()`) for database connections
- Performance tracking with ETL duration logs per table
- Modular code structure across dedicated scripts for cleaning, loading, and orchestration

These improvements transformed the project from a basic script into a foundation suitable for automation, monitoring, and future scaling.

5.2 Automation Opportunities and Tools

To reduce manual effort and ensure consistent execution, several steps in the pipeline are ideal for automation:

Step	Recommendation
Running ETL on a schedule	Use tools like **n8n** or **cron jobs**
Monitoring and retrying	Integrate with **Prefect** or **Airflow**
Logging and notifications	Add alerting on success/failure
File-based triggers	Automate ETL launch on file arrival or update

Among the tools explored, n8n stands out for this project due to its lightweight, visual interface and easy integration with file systems, databases, and shell commands. For future scalability or cloud integration, Prefect or Apache Airflow are more suitable due to their Python-native orchestration, rich observability.

5.3 Summary

The project evolved iteratively, from a basic script into a more robust, semi-automated pipeline. With the current structure in place, it's well-positioned for full automation using open-source orchestrators. n8n is recommended for immediate use, while Prefect or Airflow would be appropriate as the pipeline grows in complexity or moves into a production cloud environment.

6. Data Presentation, KPIs & Reporting

6.1 Business Objective

The goal of the Power BI dashboard is to help CityRetail stakeholders make informed, data-driven decisions related to revenue performance, product strategy, regional investment, and seasonal planning. All visuals and KPIs were designed around realistic business questions and backed by SQL views optimized for analytical performance.

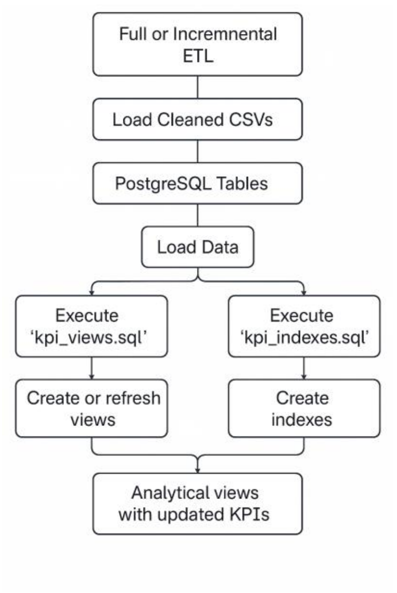


Figure X. Analytical View Refresh Workflow Supporting Dashboard KPIs

6.2 Key Business Questions Addressed

The Power BI dashboard is designed to help CityRetail stakeholders make strategic decisions based on questions such as:

- How is revenue and profitability trending over time?
- Which regions and stores contribute most to overall performance?
- Which products and categories are high or low performers?
- Are there seasonal, monthly, or weekly sales patterns?
- How does weekend performance compare to weekdays?
- Can users interactively drill into specific products, regions, or time frames?

These questions are supported through SQL views, optimized data models, and a highly interactive Power BI report.

6.3 Highlighted KPIs and Metrics

The following KPIs were chosen to support **sales, marketing, and supply chain decisions**. All KPIs are derived from SQL views for reuse and fast Power BI query performance:

KPI	Why It Matters	SQL View Used
Total Revenue	Tracks top-line business performance	`v_monthly_sales`
Quantity Sold	Measures demand and sales volume	`v_sales_by_region`, `v_product_performance`
Average Profit Margin	Evaluates product profitability	`v_product_performance`
Top Products by Revenue	Identifies key revenue drivers	`v_product_performance` (Power BI filtered)
Revenue by Category/Subcategory	Assesses product line strength	`v_monthly_sales_by_category`
Top Performing Region	Prioritizes regional investments	`v_top_region`
Weekend vs Weekday Performance	Analyzes behavioral sales patterns	`v_weekend_sales_comparison`

All KPIs are queried from reusable, index-optimized SQL views and visualized within Power BI using native features for filtering, sorting, and interaction.

6.4 Visualization Choices and Rationale

To make insights accessible and business-relevant, Power BI visuals were selected based on the nature of the data and user interaction goals.

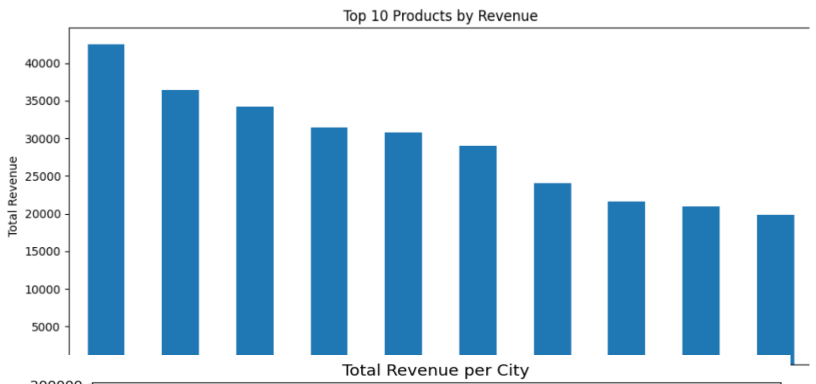
Visual Type	Purpose	KPI Focus
KPI Cards	Quick snapshot of essential metrics	Revenue, Margin, Top Region
Line Charts	Show trends and seasonality	Monthly and weekly revenue patterns
Bar Charts	Compare and rank entities (Top N Products)	Revenue, Quantity Sold (Power BI visual filter)
Stacked Columns	Reveal category-level structure and contributions	Product Category/Subcategory
Maps (Filled)	Visualize regional distribution of sales	Region-wise Quantity Sold
Slicers	Enable user-driven filtering	Dimensions: Region, Category, Month, etc.
Tooltip Drilldowns	Offer contextual detail on hover	Product and region deep dives

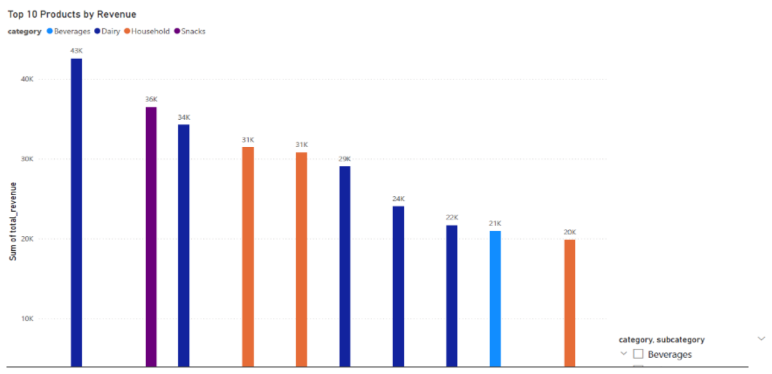
Top N logic (e.g., Top 10 Products) is applied **inside Power BI visuals**, not in SQL. This approach enables full flexibility to adjust the number of items shown and reuse a single base view (v_product_performance) across different filters and visuals.

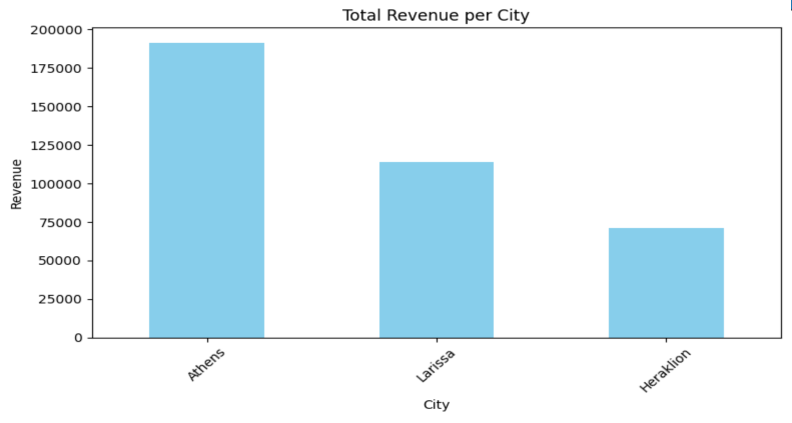
6.5 Visual Traceability from EDA to Power BI

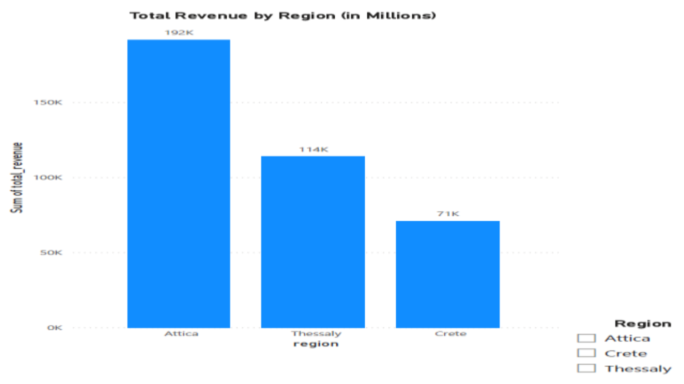
During the exploratory phase, Python visualizations were used to validate data quality, value distributions, and transformation logic — especially for city names, product revenue, and date-based trends. These early insights were later cross-verified in Power BI dashboards to ensure consistency and accuracy between raw inputs and business-ready outputs. This traceability approach validated not just cleaning steps (e.g., city-to-region mapping), but also metric rollups and categorical enrichment, aligning technical validation with stakeholder-facing KPIs.

For example, product-level revenue plots revealed early sales concentration, which translated into a Power BI bar chart grouping the top 10 products by name and category. Similarly, city-level revenue totals from EDA (e.g., Athens, Heraklion) were rolled up by region in the dashboard (e.g., Attica, Crete), with values matching precisely. These side-by-side comparisons confirmed correct transformations and aggregation logic, offering confidence that all final KPIs reflect clean, verified source data.





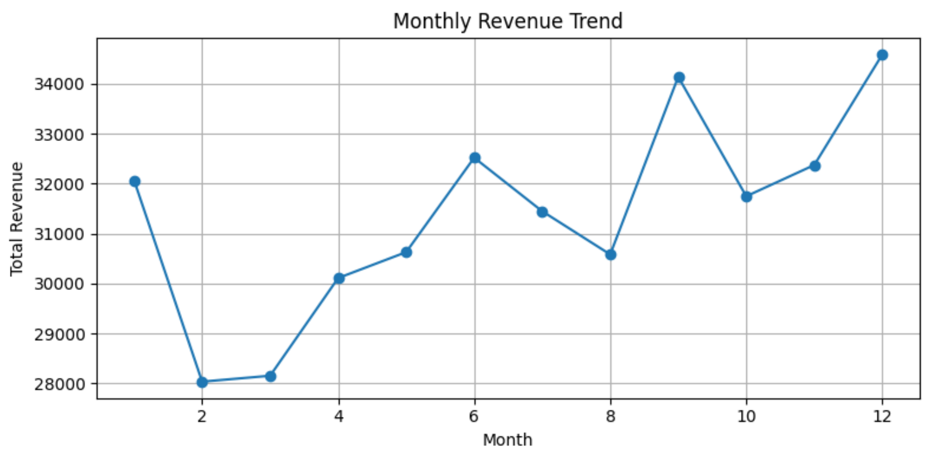




6.5.1 Visual Traceability: Monthly Revenue Trend

These line plots were used to explore **temporal patterns** in sales data and validate whether **monthly aggregation logic** during EDA was correctly reflected in the final Power BI dashboard. This helped confirm the time-based rollups and detect any inconsistencies or gaps in date mappings.

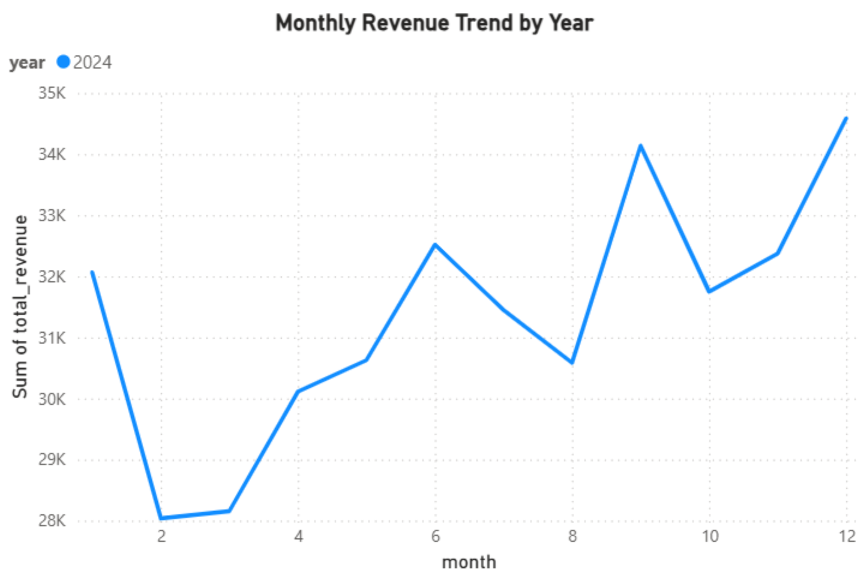
Python Plot: Monthly Revenue Trend



Interpretation:

- The plot shows monthly revenue fluctuations over the course of the year.
- Revenue dips in **February and March**, then shows a steady climb toward **June**, followed by slight dips in **July and August**.
- Notable peaks occur in **September and December**, possibly indicating **seasonal promotions or holiday spikes**.
- This EDA plot helped:
    - Verify completeness of date fields
    - Identify **monthly seasonality or trends**
    - Detect potential data gaps or misassigned dates (e.g., wrong month)

Power BI Plot: Monthly Revenue Trend by Year



Interpretation:

- This dashboard plot replicates the monthly revenue trend for the year **2024** using interactive Power BI visuals.
- The pattern is **identical to the EDA plot**, with low points in early months and peaks around **mid and end of the year**.
- The cleaner formatting and make this ideal for **executive reporting and temporal comparisons** across years (if extended).
- Useful for:
    - Forecasting and planning
    - Understanding revenue cycles
    - Presenting trend summaries to non-technical audiences

Traceability and Validation

The perfect alignment between Python and Power BI confirms:

- Correct parsing of date fields
- Accurate **monthly grouping/aggregation** (although not much preprocessing was needed for dates from raw -> cleaned with our dataset)
- No time-based anomalies or mismatches between stages

This traceability ensures that **temporal KPIs**, such as monthly sales trends or seasonal benchmarks, are backed by well-structured and validated source data. It reinforces confidence in both the **data pipeline** and the **final insights**.

6.6 Conclusion

The dashboard translates SQL KPIs into business-ready visualizations. By handling ranking, filtering, and interactivity inside Power BI, the report offers maximum flexibility, clarity, and decision-making power to users, without inflating the number of views at the database level.

Appendix

KPI Overview Panel

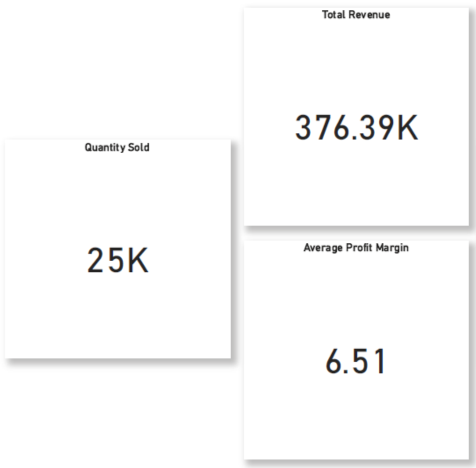

Total Revenue
376.39K
Quantity Sold
25K
Top Performing Region
Attica
Average Profit Margin
6.51

Figure A1. Power BI KPI Overview showing core business metrics.

Monthly Revenue Trend/ Weekend vs Weekday Performance

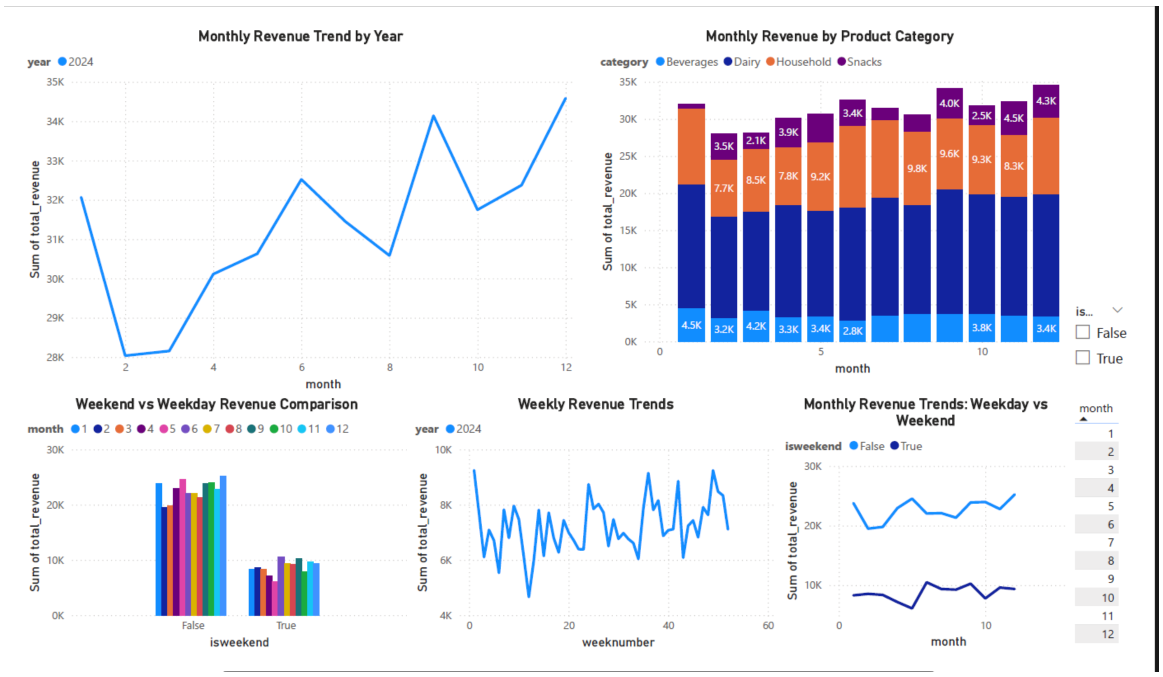


Figure A2. Monthly Revenue Trend visualizing seasonality and performance spikes /

Revenue Comparison Between Weekend and Weekday Sales.

Revenue by Region

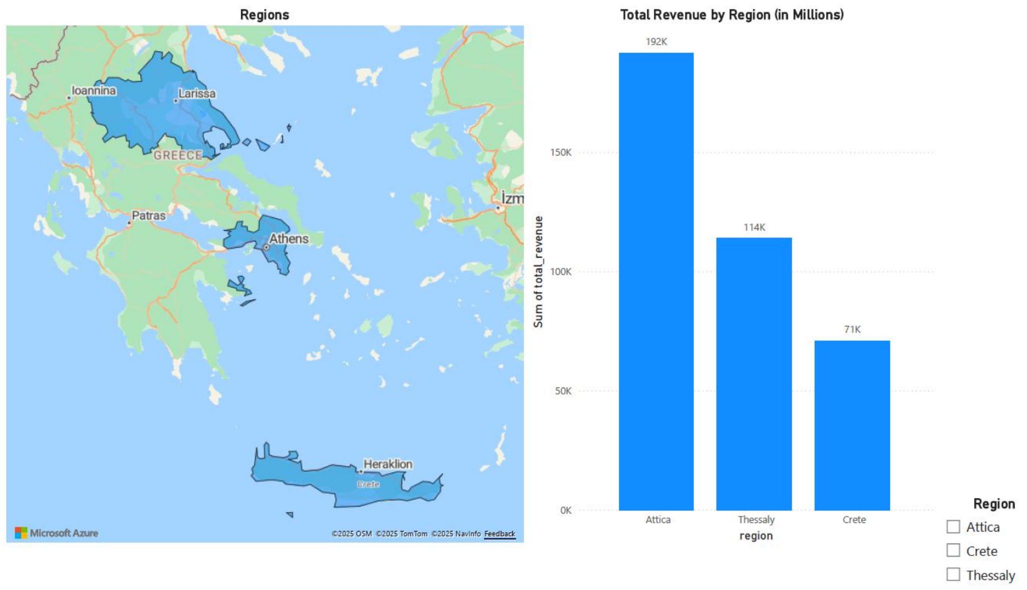


Figure A4. Regional Revenue Distribution visualized by map or stacked bar.

Top 10 Products by Revenue

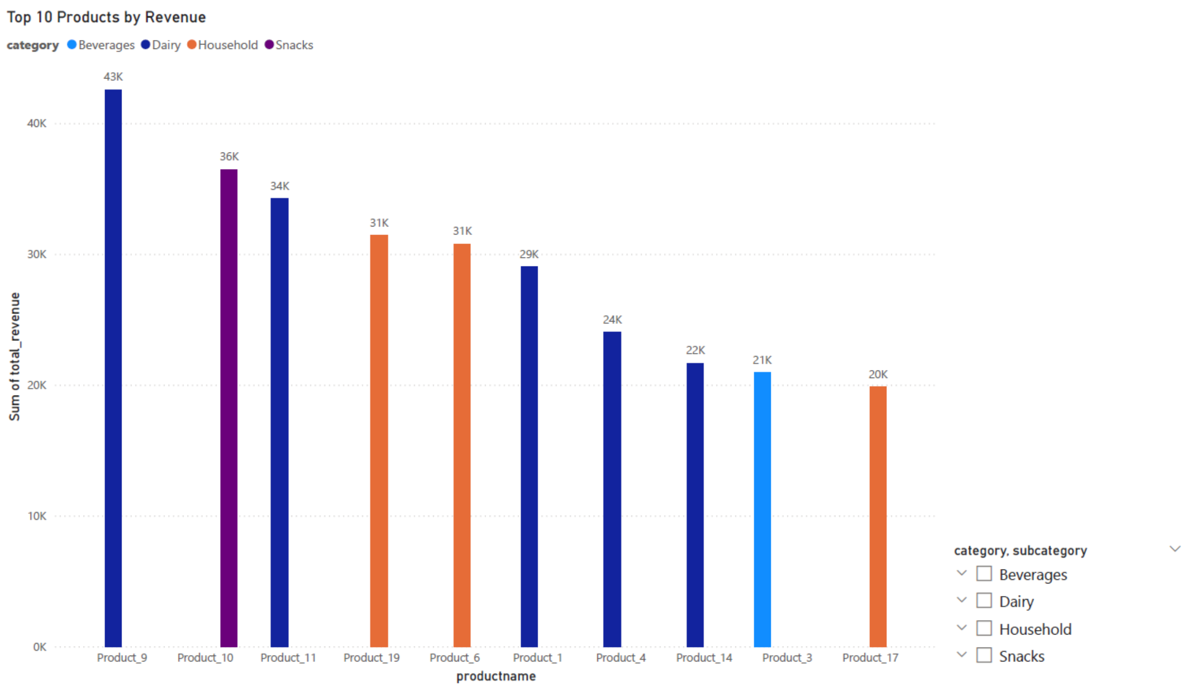


Figure A3. Top 10 Revenue-Generating Products grouped by category.

Revenue by Product Category

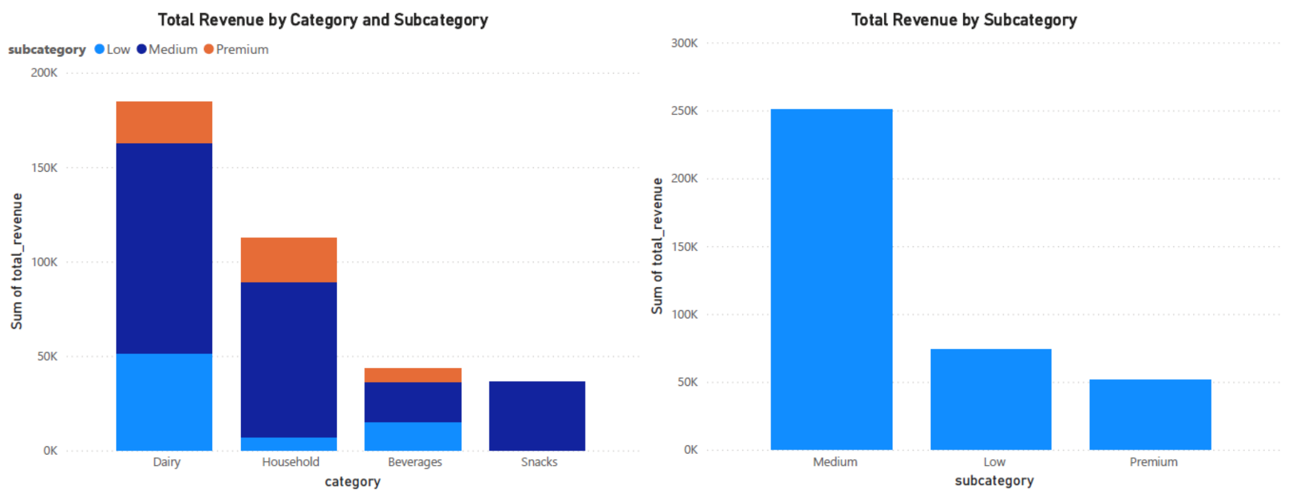


Figure A5. Revenue Contribution by Product Category and Subcategory.