# Programming for Artificial Intelligence

## Individual Assignment

Takuya Otsuki

Task 1:

# 1. Project overview

## 1.1 Aim and scope

The goal of this project is to support researchers to analyse the enormous public health dataset by implementing python-based data-insight dashboard. This system covers complete data pipeline from importing raw data to visualization, and focuses on pre-processing and exploratory data analysis instead of predictive modelling.

The scope of this project is the four core capabilities below:

- data access and load: import CSV with validation and error handling
- data cleaning and structuring: handle missing values, normalize schema formats and convert data type
- filtering and summary views: filter by country code and date range and calculate the statistics
-presentation layer: command-line interface

Three extension features (CSV export, CRUD operation and activity log) were designed but intentionally not implemented in this prototype to start with minimum viable product (MVP) (Section 2.1.2) and extend the features after discussing with the researchers. So, the three extensions are included in the future implementation plan and were only designed for now.

## 1.2 Key functionality

The implemented systems are:

1. CSV Data import
2. Data Validation and Cleaning
3. SQLite Storage
4. Flexible Filtering
5. Statistical analysis
6. Visual Output
7. CLI Interface

# 2. Software engineering and design

## 2.1 Requirements analysis and how the code is organized

### 2.1.1 Requirements analysis

The requirements were separated into functional and non-functional requirement. Each functional requirement is connected to the implementation evidence to follow Requirements Traceability Matrix (GeeksforGeeks, 2025c), resulting in preventing overlooked requirements. And FURPS+ Analysis were also applied to the requirements in order to categorise them into functionality, usability, reliability, performance and supportability dimensions. As a result, it

figured out that the core value of this project is data transformation and statistical summarisation, and advanced features such as UI and performance optimisation can be pursued later. Furthermore, MoSCoW Prioritisation classified features into Must-Have, Should-Have, Could-Have and Won't-Have. The Should-Have and Could-Have features appears in design diagrams and the reflection section, but they were consciously not implemented to demonstrate scope management. The tables of the functional (FR) and non-functional (NFR) requirements with Requirements Traceability Matrix and FURPS+ Analysis and MoSCoW Prioritisation are shown below respectively.

| ID | Requirement | Priority | Notes |
|---|---|---|---|
| FR1 | Load dataset from CSV | MUST | Core functionality |
| FR2 | Validate and parse CSV schema | MUST | Includes handling World Bank CSV metadata rows (first 2 rows) and mapping/normalising dataset-specific columns into the expected schema |
| FR3 | Clean missing / invalid values | MUST | Cleaning requirement |
| FR4 | Convert data types (date, numeric) | MUST | Cleaning requirement |
| FR5 | Store cleaned data in SQLite | MUST | LO3 (SQL/database) |
| FR6 | Filter data (country/date) | MUST | Summary requirement |
| FR7 | Summary statistics (mean/min/max/count) | MUST | Summary requirement |
| FR8 | Trend analysis (time series) | MUST | Summary requirement |
| FR9 | Group analysis (country/indicator) | MUST | Summary requirement |
| FR10 | CLI menu interface | MUST | Presentation requirement |
| FR11 | Display tables (formatted) | MUST | Presentation requirement |
| FR12 | Visualisations (line & bar plots) | MUST | Presentation requirement |
| FR13 | Data export to CSV | SHOULD | Extension |
| FR14 | CRUD update of data | SHOULD | Extension |
| FR15 | Activity logging | COULD | Extension |

| ID | Requirement | Category | Success Metric |
|---|---|---|---|
| NFR1 | Process ~20k rows under 5 seconds | Performance | Use `time.perf_counter()` on a sample WHO-style dataset. **Acceptance:** CSV load + cleaning + DB insert all complete in < 5.0 seconds on a standard laptop. |
| NFR2 | Graceful error handling | Reliability | No crashes in common edge cases (missing file, malformed CSV, empty filters). Clear error messages are shown instead. |
| NFR3 | Modular OOP structure | Maintainability | ≤ 3 main classes per module; clear separation between data, analysis and presentation layers. |
| NFR4 | Unit tests for core pipeline | Testability | ≥ 60% line coverage across core modules ( `csv_source` , `cleaner` , `repository` , `filters` , `analyzer` ). UI modules ( `visualizer` , `cli` ) should have smoke tests but are not included in coverage calculation due to I/O-heavy nature. |
| NFR5 | Clear CLI instructions | Usability | A new user can import, filter, summarise, and visualise data without reading the source code. |
| NFR6 | Cross-platform operation | Portability | Prototype tested on at least Windows and macOS. |
| NFR7 | Extensible design | Supportability | A new data source (e.g. another CSV) can be added and wired into the pipeline in < 1 hour. |
| NFR8 | Documentation | Supportability | README + docstrings for all public methods. |

Only Must-Have, NFR1-NFR5 and NFR8 were implemented to follow MVP method.

### 2.1.2 Product strategy

The project follows Minimum Viable Product (MVP) methodology (GeeksforGeeks, 2025b). This is why some features (FR13-FR15) were deferred to be implemented, which is justified as:

- Time constraints: realistic 4-week timeline meant implementing FR1-FR12 with the high quality guaranteed by ≥60% test coverage.
- Core value Delivery: researchers can already get data insights with the MVP's workflow (import, clean, filter, analyse and visualise).
- Extensibility demonstration: including these features in the design diagrams in section 2.2.3 shows forward-thinking architecture design and scope management skill. Furthermore, it is essential for a software engineer to ask users what they want and feel once the MVP was completed because redundant functions can reduce the usability and they might want to change the functionalities after seeing the MVP.

### 2.1.3 Architecture patterns

The project applied three-layer system, which separate an application into a presentation layer, domain layer and data source (Fowler, 2002), while MVC design pattern, which separates an application into Model, View and Controller components, was not chosen. The reason why MVC was not used is that, MVC is suitable for complex applications with frequent UI changes while this CLI-based project does not require it. It is better to avoid the unnecessary complexity and three-layer system can be easily maintained because having layers can reduces dependencies

on each other (Fowler, 2002).

How the three-layer is adopted in the project can be seen in the folder directory. The data folder is in charge of data cleaning, loading and storing, so this is a data source layer. The analysis layer has responsibility for statistical calculation and filtering, indicating domain layer. And presentation file is all related to user interaction such as CLI and data visualization, so this is a presentation layer.

```
Individual-Assignment-Task-1-for-PAI/
├── data/             # Data Source Layer
│   ├── csv_source.py
│   ├── cleaner.py
│   └── repository.py
├── analysis/         # Domain Layer
│   ├── analyzer.py
│   └── filters.py
├── presentation/     # Presentation Layer
│   ├── cli.py
│   └── visualizer.py
├── utils/
│   └── config.py
├── tests/
└── main.py          # Entry point (composes all layers)
```

2.1.4 Design Principles and Patterns

This project employs Object-Oriented Programming (OOP) as its core paradigm, which can handle multiple responsibilities and improve complexity management, reusability, testability and maintainability. To enjoy these advantages, the project follows SOLID Principles – five fundamental object-oriented design principles.

Single-Repository Principle (SRP) is one of the SOLID principles that is followed in the project, and SRP means a class should do only one thing with a good quality. In fact, each class has a single, concrete responsibility:

CSVDataSource: only loads CSV files

DataCleaner: only transforms and cleans data

DatabaseRepository: only handles database operations

Analyzer: only performs statistical calulation

Visualizer: only creates charts and tables

FilterCriteria: only filters data

CLIController: only manages user interaction

As an example, the Analyzer class (analysis/analyzer.py) focuses on data analysis.

```python
class Analyzer:
    """Handles data analysis operations."""

    def summary_stats(self, df: pd.DataFrame, value_col: str = "value") -> dict
        """
        Calculate summary statistics for a column.
```

```python
def trend_over_time(self, df: pd.DataFrame, date_col: str = "report_date",
                    value_col: str = "value") -> pd.DataFrame:
    """
    Calculate trend over time (mean value per date).
```

```python
def group_aggregate(self, df: pd.DataFrame, group_cols: list,
                    agg_col: str = "value") -> pd.DataFrame:
    """
    Group by specified columns and aggregate.
```

It is clear that the Analyzer class does not load data, save to database, or present the results because they are not its responsibility. As a result, if statistical logic needs to be changed, for example adding a median calculation, only Analyzer class is affected, indicating a less modification.

The remaining SOLID principles were intentionally not applied because they have only limited relevance within the scope of the MVP project.

2.1.5 Coding Standards and best practices

Since consistency within one module, function, and project is important for a better readability, the project explicitly employs three coding standards and best practices below:

a. PEP8 (Python Enhancement Proposal 8) (van Rossum *et al*., 2001)

a.1 Naming conventions

- Functions and variables:

```python
# Reorder columns
df_long = df_long[["country_code", "country_name", "indicator_code",
```

```python
def handle_missing(self, df: pd.DataFrame, strategy: str = "drop")
    """
```

- classes

```python
class DataCleaner:
```

- constants

```python
DEFAULT_CSV_PATH = "data/raw/API_SP.DYN.LE00.IN_DS2_en_csv_v2_2505.csv"
```

- Private/internal indicators

These were not used because this is a prototype and secret information is not needed yet.

a.2 Indentation

- Continuation lines and 4-space rule

```
def __init__(
    self,
    repo: DatabaseRepository,
    analyzer: Analyzer,
    visualizer: Visualizer,
    cleaner: DataCleaner
) -> None:
    """
```

a.3 Imports

- imports on separate lines
- Groped in order
- Absolute imports

```
"""Command-line interface controller."""
import pandas as pd
from typing import Optional
from data.repository import DatabaseRepository
from data.cleaner import DataCleaner
```

a.4 Whitespace in Expressions

Single space around operators: x = 1 not x=1

```
result = result[result["report_date"] >= self.date_from]
```

a.5 Blank Lines

- Two blank lines between top-level function and class definitions
- One blank line between method definitions within a class

```
class CSVDataSource:
    """Loads and validates CSV files in World Bank WDI format."""

    def __init__(self, file_path: str) -> None:
        """
        Initialize CSVDataSource with a file path.

        Args:
            file_path: Path to the CSV file.
        """
        self.file_path = file_path

    def validate(self) -> bool:
```

a.6 Type annotation

Type hints improve code readability and enable bugs to be identified earlier.

```
def __init__(self, file_path: str) -> None:
```

a.7 Documentation strings

This is essential for developers who use the methods, modules, functions, classes because they can search the comment in documentation strings.

```python
class CSVDataSource:
    """Loads and validates CSV files in World Bank WDI format."""

    def __init__(self, file_path: str) -> None:
        """
        Initialize CSVDataSource with a file path.

        Args:
            file_path: Path to the CSV file.
        """
        self.file_path = file_path
```

b. DRY (Don't Repeat Yourself) (GeeksforGeeks, 2025a)

Reusable methods are usually created to avoid the unorganised codes. Since this project has multiple methods, DRY was met. But this project is too small to reuse the method. When the features were extended, these reusable methods will be used multiple times.

## 2.2 Data structures and modelling

### 2.2.1 Data structures used

**Pandas DataFrame**:

Pandas DataFrame was chosen as the main in-memory data structure because:

- Missing value handling: enable to use NaN to express missing values with built-in methods.
- Vectorised operations: it enables fast numerical calculation without explicit loops.
- Rich API: it enables to easily notate operations for analysis, such as aggregation, transformation and filtering.

```python
stats = df[value_col].agg(["mean", "min", "max", "count"])
```

All analysis operations work on DataFrames. Data flows is CSV, DataFrame, cleaned DataFrame, SQLite, queried DataFrame and finally analysis results.

**SQLite Relational Schema (3NF)**:

Third Normal Form (3NF) was adopted to database in order to remove redundancy and guarantee data integrity. This was justified as:

- Data integrity: foreign key constraints can ensure that every report references valid country and indicator.
- Zero configuration: separate database server is not needed.
- Full SQL support: it supports transactions, indexes and parameterised queries.
-Portability: it is easy to copy and share a single .db file.

The schema (summarised here; the full ER diagram is presented in Section 2.2.2) is below:

| |
|---|
| countries (country_code PK, country_name, region) |
| indicators (indicator_id PK, indicator_code UNIQUE, indicator_name, category) |
| reports (report_id PK, country_code FK, indicator_id FK, report_date, value) |

The following design decisions were made to balance data integrity, query performance, and maintainability within the scope of the prototype.

- Surrogate keys: indicator_id and report_id use auto-incrementing integers to make internal references stable
- Natural keys preservation: country_code and indicator_code are set for human readability and CSV matching
- Date representation: report_date is stored as TEXT in "YYYY-01-01" format for portability and filtering.
- Indexing strategy: country_code, indicator_id, report_date supports the common queries.

**Python Module-Level Constants**:

The system uses module-level constants to configure settings. It is used to define file paths and configuration in utils/config.py.

```
# CSV data source
DEFAULT_CSV_PATH = "data/raw/API_SP.DYN.LE00.IN_DS2_en_csv_v2_2505.csv"

# Database configuration
DEFAULT_DB_PATH = "health_insights.db"
```

**In-memory data structures**:

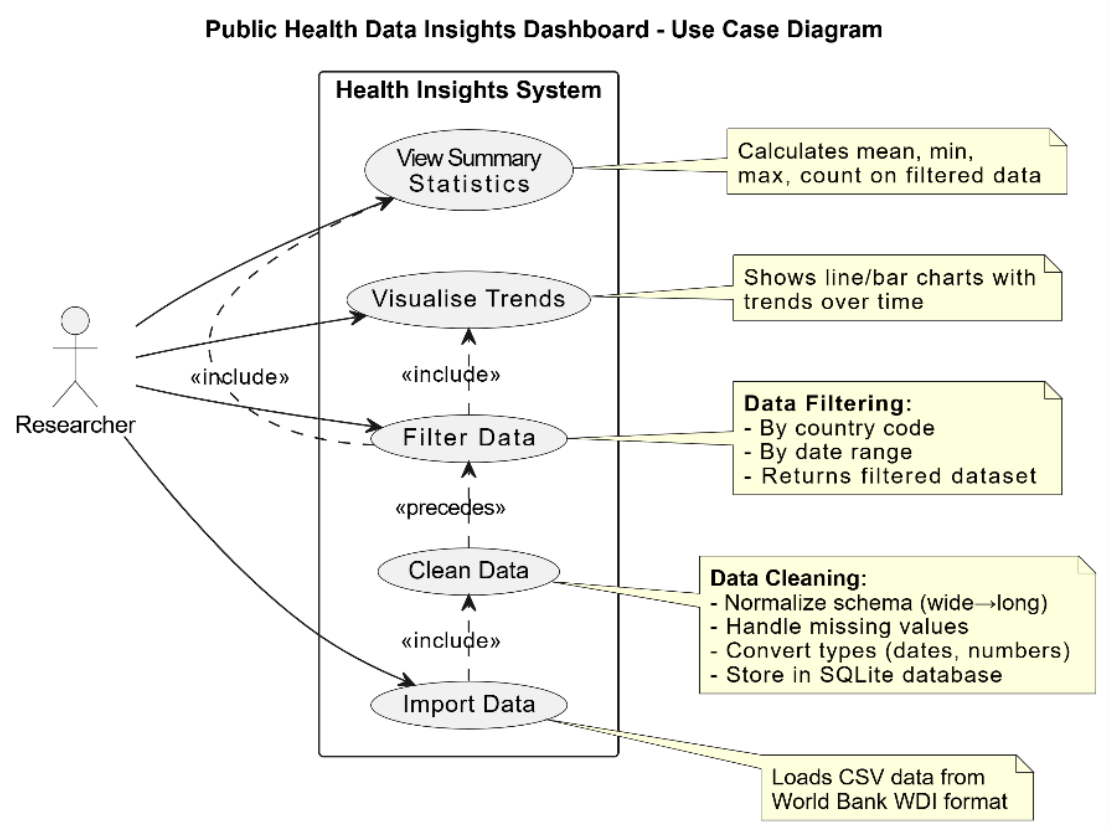
Pandas DataFrame operations created unique countries and indicators during CSV import.

```
countries = df[["country_code", "country_name"]].drop_duplicates()
indicators = df[["indicator_code", "indicator_name"]].drop_duplicates()
```

These are then iterated and inserted into the database in batches. Using INSERT OR IGNORE ensures that each country and indicator is inserted only once even if duplicates exist in the CSV.

2.2.3 System Modelling

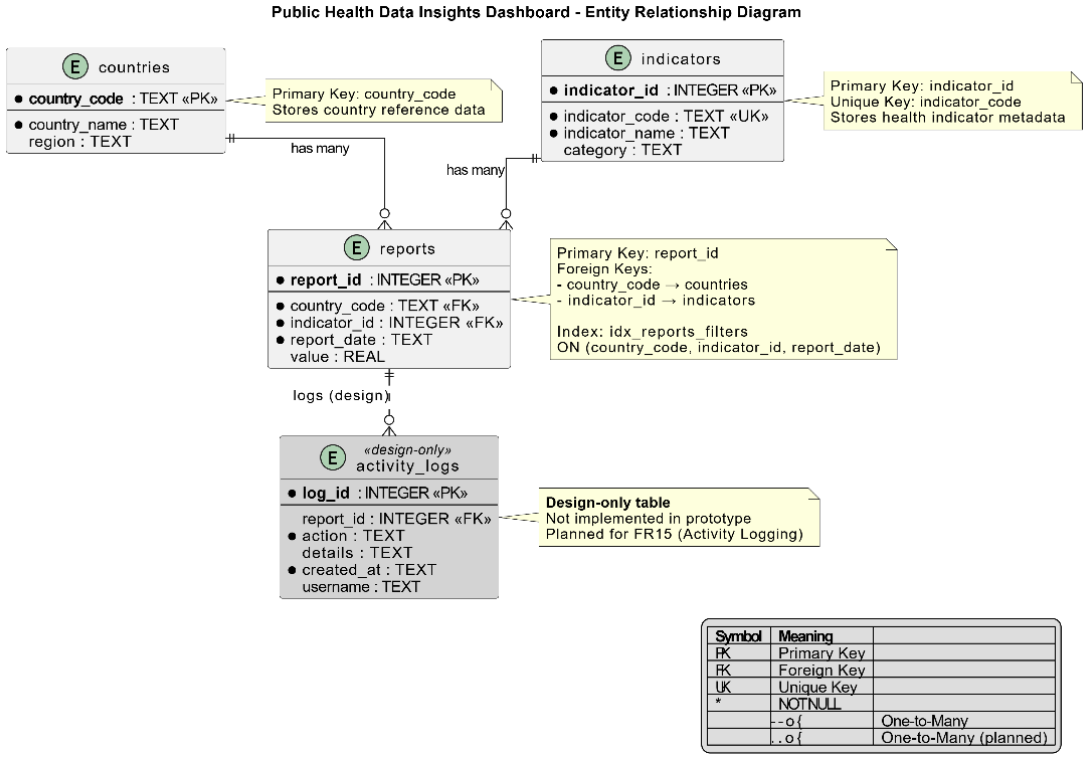
**Use Case Diagram**:




The use case diagram shows five use cases that covers the system workflow. UC1 always (Import Data) includes UC2 (Clean Data), so that is why <<include>> is put in the diagram. Five use cases were defined rather than merging them into broader workflows because:

- Each use case is directly connected to CLI menu. So, it enables clear traceability between functional requirements to user workflows.
- separating use cases can improve testability. Since UC3 (Filter Data) is separated from UC4-5(View Summary Statistics and Visulise Trend), filter logic is independent of statistical calculation, preventing the series of fails in tests.
- it also improves reusability. UC2(Clean Data) is independent of UC1, so data cleaning logic can be reused in the future use cases such as data export and data import by using API.

**Entity Relationship (ER) diagram**:




As the activity logs table is only designed (linked by dashed line to reports table) and not implemented for MVP, the ER diagram is composed of three main tables (countries, indicators and reports) and designed based on Third Normal Form (3NF). This level of normalization was selected to balance data integrity and complexity of queries.

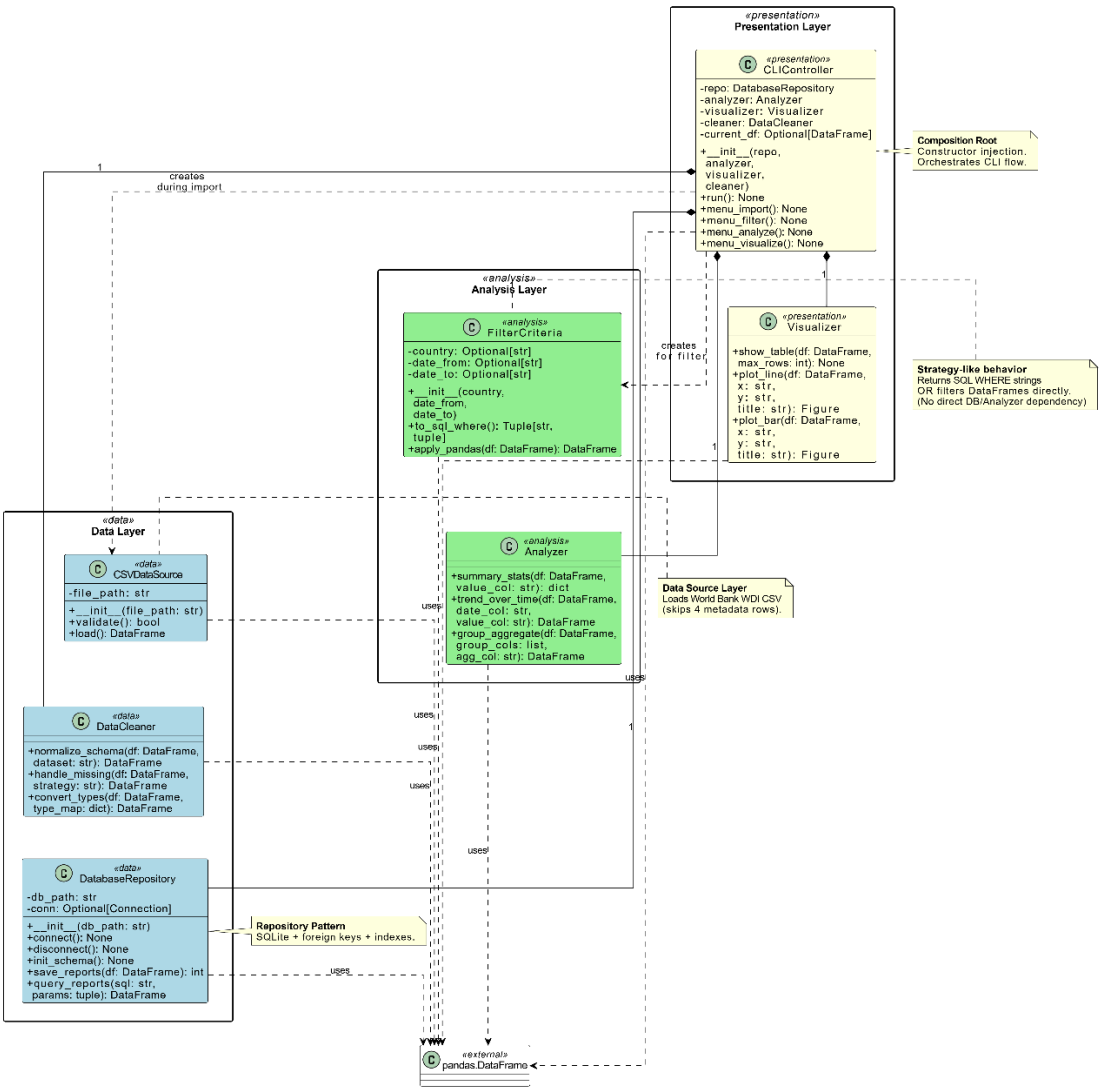
A denormalised schema was not chosen because it would cause redundancy such as repeated country names and inconsistent updates. Higher normal forms like 4NF and 5NF were also rejected because it requires too much JOIN operations for common queries such as filtering by country and date.

To lower the influence of normalization overhead, a composite index is used on country_code, indicator_id and report_date, reducing the filter query time.

Surrogate keys like indicator_id and report_id provide stable foreign key references while natural keys such as country_code and indicator_code are kept for readability and CSV matching.

This schema can be extended in the future, such as adding country metadata or indicator categories without schema changes on reports table.

**class diagram**:




The class diagram shows that seven classes making up of three layers: Presentation Layer, Analysis Layer and Data Layer. Dependencies flow is limited to downward, preventing circular dependencies and enabling independent testing on each layer.

The primary decision on class design is separation of CSVDataSource and DataCleaner. Separating the I/O process and data conversion logic makes the product meet SRP, achieving higher reusability and simpler unit testing. The two tables were not merged because the changes in file format would affect to cleaning logic.

Analyzer class has only three methods to follow MVP method. The number of methods can be extended for future features.

Filtering logic is not integrated to DatabaseRepository class and independent as FilterCriteria class. It enables the same filtering rules to be applied to both SQL query and in-memory DataFrame, improving reusability, testability and separation of responsibilities.

3. External Tools

3.1 Libraries Used

- pandas (v2.3.3): data cleaning and manipulation.
- matplotlib (v3.10.8): visualization.
- SQLite (sqlite3 stdlib): database storage.
- unittest (stdlib): testing.

1.2 Alternative Libraries Considered

- Plotly: Interactive web-based charts; modern look
- SQLAlchemy ORM: cleaner Python code without raw SQL
- pytest: More concise syntax; parametrized tests
- Tkinter: GUI

4. Testing and debugging

4.1 Test-Driven Development (TDD)

The project strictly followed Test-Driven Development (TDD) with the Red-Green-Refactoring workflow. Feature branches such as feature/csv-loading and feature/data-cleaning were also built and merged to development branch using --no--ff clear history. Development branch was merged to main branch after the development branch was tested to work properly. This is essential to always keeping the product working properly.

4.2 Logging and Exception Handling

Exception Handling Strategy:

Try/except blocks with exception handling is demonstrated below:

```python
def validate(self) -> bool:
    """
    Validate that the CSV file exists and is readable.

    Returns:
        True if file is valid, False otherwise.
    """
    try:
        # Check if file exists
        if not os.path.exists(self.file_path):
            return False

        # Check if file is not empty
        if os.path.getsize(self.file_path) == 0:
            return False

        # Try reading the file
        pd.read_csv(self.file_path, skiprows=4, nrows=0)
        return True
    except Exception:
        return False
```

Transaction Safety:

Database inserts use BEGIN/COMMIT with ROLLBACK on failure for atomicity.

```
try:
    cursor.execute("BEGIN TRANSACTION;")
```

```
    cursor.execute("COMMIT;")
    return report_count

except Exception as e:
    cursor.execute("ROLLBACK;")
    raise
```

This was inserted to ensure if any insert fails all are rolled back, enabling database is never left in partial state and the process is completed faster.

Logging:

The system was designed with a future logging framework in mind:

- activity_logs table schema designed (FR15 - not implemented, can check in the ER diagram)
- Would log all user actions (IMPORT, FILTER, EXPORT) with timestamps

5. Use of AI-generated code

AI was used for most of coding part, for making the design more robust by checking any lacks. The all the output from AI were definitely critically reviewed and modified. For design creating support, ChatGPT and Claude were used to develop the idea. For coding, Claude Code was used. When the Claude Code was used, it gave a unit of code and a commit message based on TDD. All AI outputs were verified and modified to ensure originality.

6. Reflection

Even though this is a prototype, exception handling can be more improved while features are enough as a prototype. Specific exceptions or clear error messages were not used and try/exception block was also scarce. However, as explained in Section 2, the justification of structures, design patterns and strategy such as MVC, a denormalised schema, higher normal forms, OLID in SOLID was concrete. That would be better if their drawbacks of the methods used were also discussed. For future extension, following MoSCoW, the CSV export, CRUD, activity logging would be implemented as it was designed. Furthermore, private/internal indicators, UI could be also implemented. GPU with CUDA operation can be considered for the more robust system against huge transactions in the future.