CT-592

Big Data Analytics

# Integrated Financial Data ETL Pipeline

Project Report By

Muhammad Ismail

## Table of Contents

1. **Introduction**
2. **Pipeline Design**
   - 2.1 Conceptual Overview
   - 2.2 Detailed Workflow (Extract, Transform, Load)
   - 2.3 Data Flow Diagram
   - 2.4 Component Interaction Diagram
3. **Technology Choices Justification**
   - 3.1 Core Language: Python
   - 3.2 Data Manipulation: Pandas & NumPy
   - 3.3 Data Source Connectors (Requests, Psycopg2, Pymongo, Gspread)
   - 3.4 Target Database: PostgreSQL
   - 3.5 Configuration: JSON
   - 3.6 Scheduling: Python `schedule` Library
   - 3.7 CI/CD Platform: GitHub Actions
4. **CI/CD Implementation**
   - 4.1 Overview
   - 4.2 Typical Workflow Description
5. **CI/CD and Reliability**
   - 5.1 Reduction of Manual Errors
   - 5.2 Rapid Feedback Loops
   - 5.3 Improved Data Integrity
   - 5.4 Accelerated Development & Deployment
6. **Conclusion**
7. **Future Enhancements**

## 1. Introduction

This report details an Extract, Transform, Load (ETL) pipeline designed to integrate financial data from diverse sources into a unified analytical database. The pipeline sources data from CSV files (stock market data), a public API (cryptocurrency data), a PostgreSQL database (economic indicators), a MongoDB database (financial news sentiment), and Google Sheets (company financial reports).

The primary goal is to extract raw data, clean and standardize it, perform feature engineering, aggregate key metrics, and load the resulting insights into a PostgreSQL database for analysis and reporting. The pipeline is designed to be automated using a scheduler and incorporates Continuous Integration/Continuous Deployment (CI/CD) principles for enhanced reliability and maintainability.

## 2. Pipeline Design

### 2.1 Conceptual Overview

The pipeline follows a standard ETL pattern:

1. Extraction: Retrieve data from various sources (CSV, API, SQL DB, NoSQL DB, Google Sheets).
2. Transformation: Clean, standardize, enrich, and aggregate the extracted data into a consistent format.
3. Load: Insert the processed, aggregated data into a target PostgreSQL database table.
4. Scheduling: Automate the execution of the entire ETL process on a predefined schedule (daily).
5. CI/CD: Automate testing and validation during development to ensure code quality and pipeline reliability.

### 2.2 Detailed Workflow (Extract, Transform, Load)

The core logic resides in `etl_pipeline.py`:

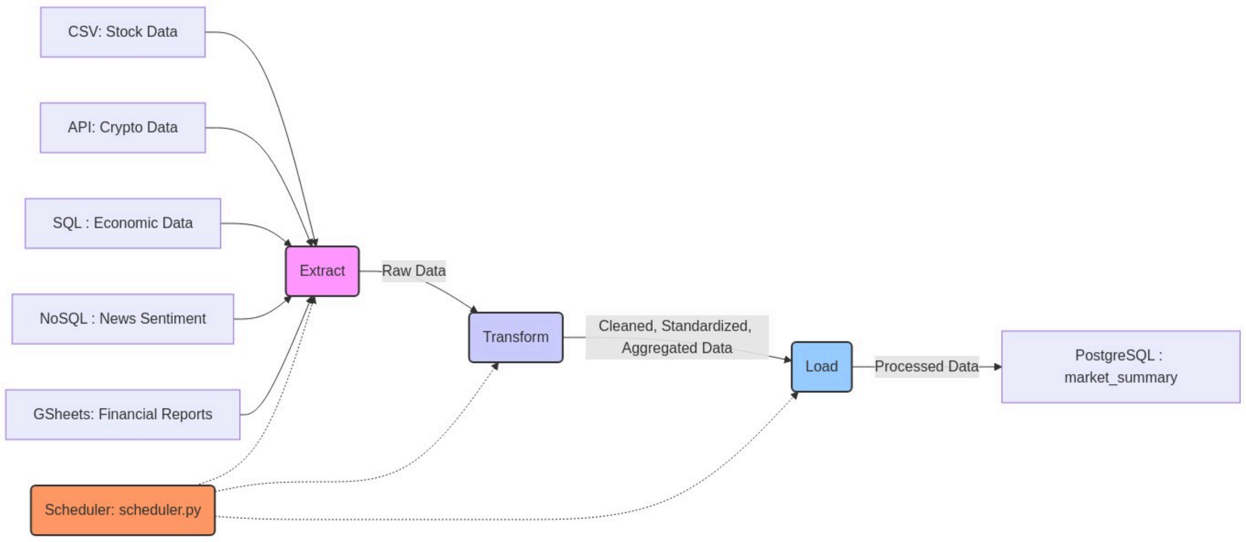
- **Extraction Phase:**
  - `extract_stock_data_csv()`: Reads stock data from `data/stock_market_data.csv` using Pandas. Validates required columns.
  - `extract_crypto_data_api()`: Fetches cryptocurrency market data from the CoinGecko API using the `requests` library. Handles API key management via `config/api_keys.json`.
  - `extract_economic_data_sql()`: Queries economic indicators from a source PostgreSQL database using SQLAlchemy and configuration from `config/db_config.json`.

- `extract_news_data_mongodb()`: Retrieves recent financial news sentiment data from a MongoDB collection using `pymongo` and configuration from `config/db_config.json`.
- `extract_financial_reports_gdrive()`: Accesses and reads company financial data from a Google Sheet using `gspread` and service account credentials (`gspread-creds.json`).

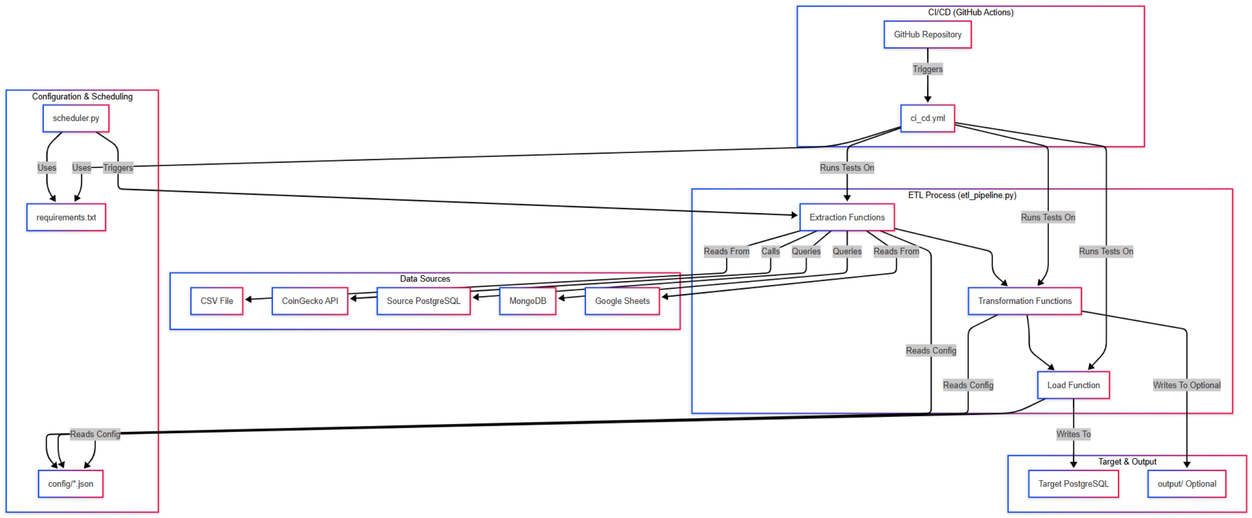
- **Transformation Phase:**
  - Cleaning (`clean_...` functions): Each data source undergoes specific cleaning:
    - Date parsing and standardization.
    - Duplicate removal based on relevant keys (e.g., Date/Symbol for stocks, ID/Date for crypto).
    - Handling missing values (dropping critical NaNs, filling/forward-filling others).
    - Ensuring correct numeric data types, and handling errors (e.g., currency symbols).
  - Standardization (`standardize_data`):
    - Converts all date/timestamp columns to a common `date` column (timezone-naive).
    - Attempts currency standardization to USD (Note: Requires actual exchange rate implementation).
    - Adds `asset_type` and `data_source` columns for clarity and future filtering.
  - Feature Engineering (`engineer_features`):
    - Calculates stock metrics: daily returns, moving averages, volatility.
    - Calculates crypto metrics: market cap to price ratio.
    - Calculates news metrics: rolling average sentiment score per entity.
    - Calculates financial report metrics: profit margin, debt-to-equity ratio.
  - Aggregation (`aggregate_data`):
    - Combines insights from all cleaned and engineered data sources into a single `daily_market_summary` Pandas DataFrame.
    - Groups data by date and calculates aggregate statistics (mean returns, total volume, average sentiment, etc.).
    - Uses forward-filling (`ffill`) to handle missing values, especially for less frequent data like economic indicators or quarterly reports.

- **Load Phase:**
    - `load_to_database()`:
        - Connects to the target PostgreSQL database using `psycopg2` and configuration from `config/db_config.json`.
        - Ensures the target table (`market_summary`) exists, creating it if necessary.
        - Filters the aggregated DataFrame columns to match the target table schema.
        - Iterates through the `daily_market_summary` DataFrame and inserts/updates records into the `market_summary` table using an `INSERT ... ON CONFLICT DO UPDATE` statement (upsert logic) to handle existing dates gracefully.

## 2.3 Data Flow Diagram



### 2.4 Component Interaction Diagram



## 3. Technology Choices Justification

- ### 3.1 Core Language: Python:
  - **Justification:** Python is the de facto standard for data science and data engineering tasks due to its extensive ecosystem of libraries (Pandas, NumPy, Requests, database connectors), readability, ease of integration, and large community support. It allows for rapid development and handling diverse data formats efficiently.

- ### 3.2 Data Manipulation: Pandas & NumPy:
  - **Justification:** Pandas provides high-performance, easy-to-use data structures (DataFrame) and data analysis tools, essential for cleaning, transforming, aggregating, and analyzing structured data. NumPy provides the underlying numerical computation capabilities.

- ### 3.3 Data Source Connectors (Requests, Psycopg2, Pymongo, Gspread):
  - **Justification:** These are standard, well-maintained libraries for interacting with specific data sources:
    - `requests`: Simplifies making HTTP requests to APIs (CoinGecko).

        - `psycopg2` (and `SQLAlchemy`): Robust and widely used drivers for interacting with PostgreSQL databases. SQLAlchemy adds an abstraction layer (engine) for easier connection management.
        - `pymongo`: The official driver for interacting with MongoDB databases.
        - `gspread` / `oauth2client`: Standard libraries for authenticating and interacting with the Google Sheets and Google Drive APIs.

- **3.4 Target Database: PostgreSQL:**
    - **Justification:** PostgreSQL is a powerful, open-source object-relational database system known for its reliability, feature robustness, and ACID compliance. It is well-suited for storing the structured, aggregated `market_summary` data, allowing for complex analytical queries later.

- **3.5 Configuration: JSON:**
    - **Justification:** JSON (`db_config.json`, `api_keys.json`) is used for storing database credentials and API keys. It's human-readable, lightweight, and easily parsed by Python, making it ideal for separating configuration from code, which enhances security and maintainability.

- **3.6 Scheduling: Python `schedule` Library:**
    - **Justification:** The `schedule` library provides a simple, human-friendly way to schedule periodic jobs purely within Python. It's suitable for straightforward scheduling needs like running a single ETL script daily. While less robust than OS-level `cron` or full orchestrators (like Apache Airflow), it's easy to set up and manage for this specific project scope, running directly alongside the ETL code.

- **3.7 CI/CD Platform: GitHub Actions:**
    - **Justification:** The presence of `.github/workflows/ci_cd.yml` suggests GitHub Actions. It's tightly integrated with GitHub repositories, offers a generous free tier, uses declarative YAML configuration, and provides a wide range of triggers and community actions for building, testing, and deploying workflows.

## 4. CI/CD Implementation

### 4.1 Overview

Continuous Integration (CI) and Continuous Deployment (CD) practices are implemented using GitHub Actions, triggered by events in the GitHub repository (e.g., pushes or pull requests). The goal is to automate the testing and validation process, ensuring that code changes do not break the pipeline's functionality and maintain data quality.

*(Note: The specific `ci_cd.yml` file content was not provided. This section describes a typical implementation for such a project.)*

### 4.2 Typical Workflow Description

A common CI workflow (`.github/workflows/ci_cd.yml`) for this project would likely include the following steps:

1. Trigger: Define when the workflow runs (e.g., `on: [push, pull_request]`).
2. Checkout Code: Fetch the latest code from the repository branch.
3. Setup Python: Install the correct Python version.
4. Install Dependencies: Install required packages using `pip install -r requirements.txt`.
5. Linting: Run code linters (e.g., `flake8`, `black`) to enforce code style and catch simple errors.
6. Testing:
   - Unit Tests: Run tests on individual functions (extractors, transformers, loaders) using mock data or test databases/APIs. (Requires creating a separate `tests/` directory and test files using a framework like `pytest`).
   - Integration Tests: Run tests that verify the interaction between different components, potentially running a mini-version of the pipeline with sample data and asserting the output format or database state.
7. (Optional) Deployment: If the scheduler needs to be deployed to a server or cloud service, this step would handle packaging and deployment. For this project structure, deployment might be manual or a simpler process.

## 6. Conclusion

This ETL project successfully implements a pipeline to integrate diverse financial data sources into a unified PostgreSQL database. It leverages Python and its rich data science ecosystem for efficient extraction, transformation, and loading. The design emphasizes modularity, configuration management, and automation through scheduling.

The incorporation of CI/CD practices (even if described conceptually here) is critical for ensuring the long-term reliability, maintainability, and data integrity of the pipeline. By automating testing and validation, the project minimizes manual errors, provides rapid feedback, and builds confidence in the deployed code and the resulting data.

## 7. Future Enhancements

- **Comprehensive Testing:** Develop a full suite of unit and integration tests using `pytest` to cover edge cases and ensure robustness. Mock external services (APIs, databases) for isolated unit testing.
- **Monitoring and Alerting:** Implement logging throughout the pipeline and add monitoring tools (e.g., Prometheus/Grafana, Datadog) to track pipeline health, performance, and data quality metrics. Set up alerts for failures or anomalies.
- **Orchestration:** For more complex dependencies or error handling, consider migrating from the `schedule` library to a workflow orchestrator like Apache Airflow.
- **Data Validation Framework:** Integrate a data validation library (e.g., Great Expectations, Pandera) to define and enforce explicit data quality rules during the transformation phase.
- **Secrets Management:** Use a more secure secrets management solution (e.g., HashiCorp Vault, AWS Secrets Manager, GCP Secret Manager) instead of storing credentials directly in config files, especially in production environments.
- **Containerization:** Package the ETL application and its dependencies into a Docker container for easier deployment and environment consistency.
- **Idempotency:** Ensure all pipeline steps, especially the Load step, are fully idempotent (running them multiple times with the same input produces the same result). The current `ON CONFLICT DO UPDATE` helps, but other steps should also be reviewed.