



PROJECT
DATA ENGINEERING
(DS437)

II sem 1446 AH

StockFlow Analytics

| Student name | ID | Section |
|-----------------|-----------|---------|
| Haya Alnashwan | 444008713 | 63S |
| Lubna Bin Taleb | 444008721 | 63S |
| Fajer alshuwier | 444008709 | 63S |
| Noura Albarak | 444008746 | 63S |
| lina aljasir | 444008726 | 63S |

Table of Contents

| | |
|---|-----------|
| Chapter 1: Introduction | 4 |
| Chapter 2: System Design and Architecture | 4 |
| 2.1 Data Sources | 4 |
| 2.2 Ingestion Method | 4 |
| 2.3 Raw Data | 5 |
| 2.4 Data Validation | 5 |
| Chapter 3: ETL Pipeline Design | 6 |
| 3.1 Introduction | 6 |
| 3.2 ETL Architecture Overview | 6 |
| 3.3 Extract: Data Collection | 7 |
| 3.4 Transform: Data Cleaning and Integration | 8 |
| 3.5 Load: Database Storage | 11 |
| Chapter 4: Implementation and Testing | 12 |
| 4.1 Introduction | 12 |
| 4.2 ETL Execution Summary | 12 |
| 4.3 Visualize Using Jupyter | 13 |
| 4.4 Performance Challenges Solutions | 18 |
| Chapter 5: (Phase 2) Introduction to ELT | 18 |
| 5.1 Extract, Transform, Load, or ETL: | 18 |
| 5.2 Extract, Load, Transform, or ELT: | 19 |
| 5.3 The Use of ELT | 19 |
| 5.4 Tools for ELT Implementation | 19 |
| Chapter 6: ELT Pipeline Design | 20 |
| 6.1 Data Pipeline Architecture (ELT) | 20 |
| 6.2 Data Sources: Collecting Raw Data | 21 |
| 6.3 Validation and Quality Check Using Great Expectations (GE) | 22 |
| 6.4 Load the raw data in datawarehouse | 22 |
| 6.5 Transform: SQL based transformation | 23 |
| Chapter 7: Implementation & Results | 23 |
| Comparison of ETL vs. ELT (speed, scalability, efficiency) | 27 |
| ELT Data Preprocessing Challenges and Solutions | 27 |
| Sustainability Goals Achieved | 27 |
| Conclusion | 28 |
| References | 29 |

List of Figures

| | |
|--|----|
| Figure 1 System architecture | 6 |
| Figure 2 ETL architecture..... | 6 |
| Figure 3: Code to define CSV file path and stock API URL. | 7 |
| Figure 4: Function to read, clean, and convert raw CSV stock data into a structured Data Frame.... | 8 |
| Figure 5: Function to retrieve, clean, and prepare stock data from the Alpha Vantage API..... | 11 |
| Figure 6: Function to merge and clean data from both CSV and API sources, handling duplicates and prioritizing API values. | 12 |
| Figure 7: Main function to execute the full ETL process, including extraction, transformation, CSV save, and SQLite loading..... | 12 |
| Figure 8: Terminal output showing successful ETL execution and data preview. | 12 |
| Figure 9: SQLite command-line output confirming correct table creation and data storage..... | 13 |
| Figure 10: Volume histogram before cleaning..... | 18 |
| Figure 11: Volume histogram after cleaning..... | 18 |
| Figure 12: Line Plot of Close Price Over Time (After Cleaning)..... | 18 |
| Figure 13: Sample of Original (Raw) CSV Data | 19 |
| Figure 14: Raw CSV Data Info Output..... | 19 |
| Figure 15: Null Value Check in Raw CSV | 19 |
| Figure 16: Null Value Check In Cleansed Data Base..... | 20 |
| Figure 17 ELT Pipeline | 20 |

List of Tables

| | |
|---|----|
| Table 1: Challenges and their solutions | 18 |
|---|----|

Chapter 1: Introduction

Data engineering is essential for transforming messy, raw data into structured, reliable information. Without it, organizations like Netflix and Amazon couldn't deliver personalized recommendations or operate efficiently. In our project, we worked with stock market data from two sources: live prices via the Alpha Vantage API and historical data from a CSV file. This mirrors real-world scenarios where companies combine real-time and historical data for smarter decision-making. Through careful extraction, validation, and transformation, we ensured the system was accurate and dependable.

Managing data pipelines comes with challenges like missing values, inconsistent formats, and duplicate records. APIs can change unexpectedly, and CSV files may contain outdated information. Our main goal was to keep the data clean, complete, and ready for analysis. To solve this, we built a pipeline that automatically extracts, validates, and loads data into a database, minimizing manual errors and preserving data quality.

We chose to build an ELT pipeline, where raw data is first loaded into the database, then transformed within it. This method offers flexibility, faster loading, and better scalability compared to ETL, which transforms data before loading. Given the need to quickly handle data from both API and CSV sources, ELT was the ideal choice for our project.

Chapter 2: System Design and Architecture

2.1 Data Sources

Data sources can include APIs, databases, flat files like CSVs, or web scraping. In our project, we used two main sources:

- **Alpha Vantage API** for real-time stock data
- **CSV file** for historical stock data

This setup simulates a real-world financial analysis environment, allowing us to test the pipeline's flexibility and robustness.

2.2 Ingestion Method

We applied **batch ingestion**, where data is collected at scheduled intervals rather than streamed continuously.

This method is suitable for stock market data that does not require millisecond-level updates and is more resource-efficient and manageable.

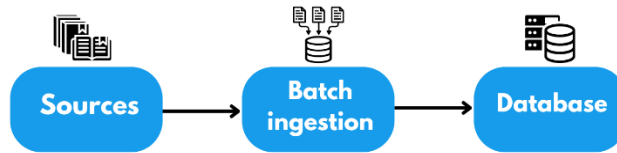


Figure 1 System architecture

2.3 Raw Data

Preparation and Transformation

Preparation steps included:

- Converting fields (open, high, low, close, volume) into numeric types
- Parsing timestamps correctly
- Checking for missing or null values

For transformation, we aggregated data by **calculating the average closing price per hour**, applied type casting, and ensured timestamp consistency.

Since the data was organized by timestamp, deduplication wasn't necessary in this case.

2.4 Data Validation

We used the **Great Expectations** library to validate data quality. Validation checks included:

- No null values in critical columns (timestamp, close)
- Closing prices within a reasonable range (100–1000)

Records failing validation were automatically rejected, maintaining database cleanliness similar to how invalid emails are filtered out.

Chapter 3: ETL Pipeline Design

3.1 Introduction

This chapter outlines the design and architecture of our ETL (Extract, Transform, Load) pipeline, which integrates data from a local CSV file and a stock market API. The ETL process is responsible for ensuring data consistency, quality, and readiness for storage and analysis. The implementation is done in Python, leveraging libraries like pandas, requests, and sqlite3.

3.2 ETL Architecture Overview

Our ETL architecture follows a linear pipeline structure:

- Extract: Retrieve raw data from a local file (tpge.csv) and from the Alpha Vantage Intraday API.
- Transform: Clean, normalize, and merge data from both sources.
- Load: Store the final cleaned dataset in a local SQLite database (stocks_etl3.db).

Data pipeline architecture (ETL)

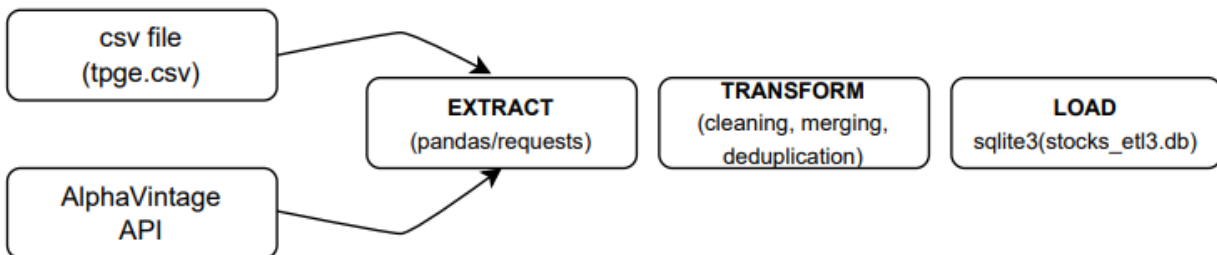


Figure 2 ETL architecture

3.3 Extract: Data Collection

```
import requests

import pandas as pd

import os

import sqlite3

# 1. Define file path and API URL

CSV_FILE_PATH = r"C:\Users\DSNou\venv\tpge.csv"

API_URL = [

    "https://www.alphavantage.co/query"

    "?function=TIME_SERIES_INTRADAY"

    "&symbol=IBM"

    "&interval=5min"

    "&apikey=demo"

]

print("Reading CSV_PATH:", CSV_FILE_PATH, "exists?", os.path.exists(CSV_FILE_PATH))
```

Figure 3: Code to define CSV file path and stock API URL.

- **requests:** for communicating with the API.
- **pandas:** for handling data in DataFrame structures.
- **os:** for interacting with the file system (e.g., checking if a file exists).
- **sqlite3:** for working with a SQLite database.
- We define the path to the CSV file and the query URL for the Alpha Vantage service (using a demo API key).
- We print a message to confirm that the file exists.

3.4 Transform: Data Cleaning and Integration

Transformation involves:

```
def extract_csv(file_path):  
    try:  
        df = pd.read_csv(file_path)  
  
        df.columns = df.columns.str.lower().str.strip()  
  
        df["datetime"] = pd.to_datetime(df["date"] + " " + df["time"], errors="coerce")  
  
        df.drop(columns=["date", "time"], inplace=True)  
  
        df = df[["datetime", "open", "high", "low", "close", "volume", "openint"]]  
        df["volume"] = df["volume"].astype('int64')  
  
        for c in ["open", "high", "low", "close", "openint"]:  
            df[c] = pd.to_numeric(df[c], errors="coerce")  
  
        df.dropna(subset=["datetime"], inplace=True)  
  
        print("CSV data extracted successfully")  
  
        return df  
    except Exception as e:  
        print(f"Error reading CSV file: {e}")  
  
        return pd.DataFrame()
```

Figure 4: Function to read, clean, and convert raw CSV stock data into a structured Data Frame.

- Reads the file into a Data Frame.
- Normalizes column names to lowercase and strips whitespace.
- Constructs a datetime column from the date and time columns, then drops the original date and time columns.
- Select only the required columns in a logical order.
- Convert volume to a 64-bit integer, convert the other fields to numeric types, and drop rows where the datetime conversion failed.


```

def extract_api(api_url):
    try:
        resp = requests.get(api_url, timeout=10)
        resp.raise_for_status()
        data = resp.json()
        raw = data.get("Time Series (5min)", {})
        if not raw:
            print("No time-series data found in API response.")
            return pd.DataFrame()

        df_api = {}

        pd.DataFrame.from_dict(raw, orient="index")
        .reset_index()
        .rename(columns={
            "index": "datetime",
            "1. open": "open",
            "2. high": "high",
            "3. low": "low",
            "4. close": "close",
            "5. volume": "volume"
        })

        df_api["datetime"] = pd.to_datetime(df_api["datetime"], errors="coerce")
        for c in ["open", "high", "low", "close", "volume"]:
            df_api[c] = pd.to_numeric(df_api[c], errors="coerce")
        df_api.dropna(subset=["datetime"], inplace=True)
        df_api.drop_duplicates(subset=["datetime"], inplace=True)
        print("API data extracted and ready for merging")
        return df_api[["datetime", "open", "high", "low", "close", "volume"]]
    except Exception as e:
        print(f"Warning: Could not fetch data from API - {e}")
        return pd.DataFrame()

```

Figure 5: Function to retrieve, clean, and prepare stock data from the Alpha Vantage API.

- Sends a GET request to the API and verifies it succeeded.
- Extract the time-series section and convert it into a Data Frame.
- Renames the columns to simple names, then parses dates and numeric fields, and removes duplicates.

```

def clean_and_merge(csv_df, api_df):
    if csv_df.empty and api_df.empty:
        print("No data to clean or merge!")
        return pd.DataFrame()

    if csv_df.empty:
        print("CSV empty--returning API data only")
        return api_df

    if api_df.empty:
        print("API empty--returning CSV data only")
        return csv_df

    # drop duplicates
    csv_df = csv_df.drop_duplicates(subset=["datetime"])
    api_df = api_df.drop_duplicates(subset=["datetime"])

    # merge
    merged = csv_df.merge(
        api_df,
        on="datetime",
        how="left",
        suffixes=("_csv", "_api")
    )

    # prefer API values, fallback to CSV
    for field in ["open", "high", "low", "close", "volume"]:
        merged[field] = merged[f"{field}_api"].fillna(merged[f"{field}_csv"])
        merged.drop(columns=[f"{field}_csv", f"{field}_api"], inplace=True)

    print("Data cleaned and merged successfully")
    return merged

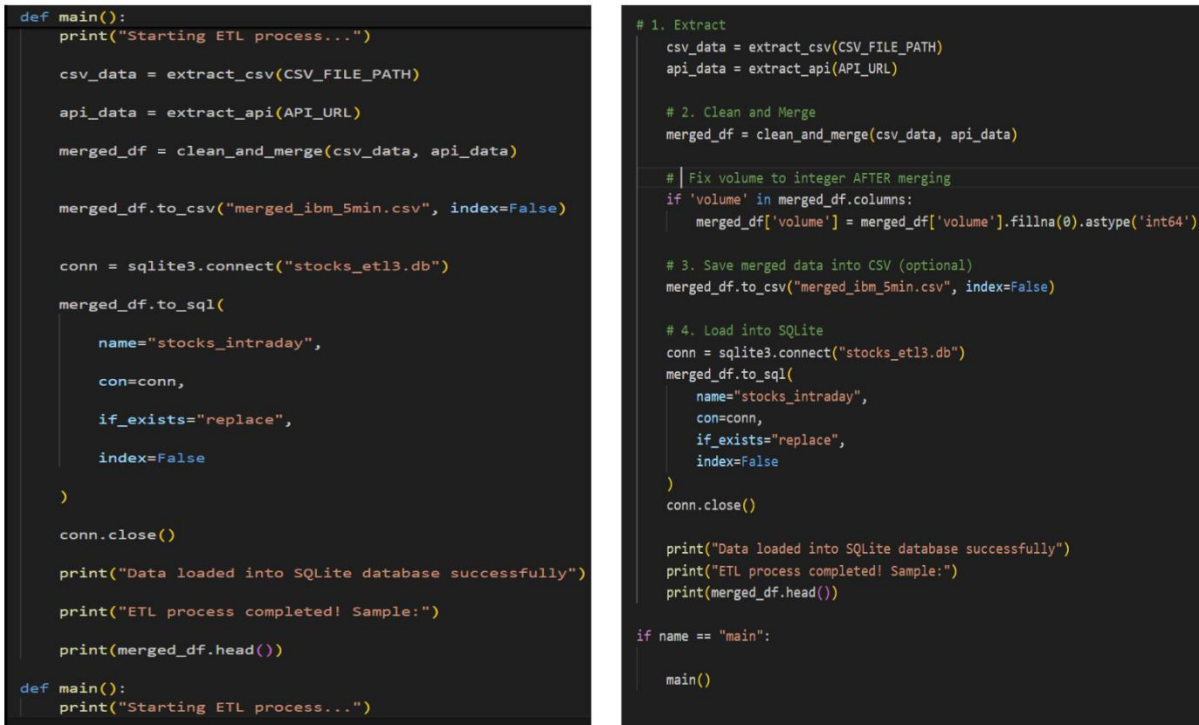
```

Figure 6: Function to merge and clean data from both CSV and API sources, handling duplicates and prioritizing API values.

This function takes two DataFrames one from the CSV (csv_df) and one from the API (api_df) and:

1. **Handles empty inputs**
 - a. If both are empty, return an empty DataFrame.
 - b. If only one is empty, return the non-empty one.
2. **Deduplicates**
 - a. Drops any rows with the same datetime in each DataFrame to ensure unique timestamps.
3. **Merges**
 - a. Performs a left join on datetime, so every CSV row is preserved, and API values are brought in where available.
4. **Resolves conflicts**
 - a. For each numeric field (open, high, low, close, volume), prefers the API value if present, otherwise falls back to the CSV value.
 - b. Removes the now redundant _csv and _api columns.
5. **Returns**
 - a. The cleaned, deduplicated, and merged DataFrame.

3.5 Load: Database Storage



```
def main():
    print("Starting ETL process...")

    csv_data = extract_csv(CSV_FILE_PATH)
    api_data = extract_api(API_URL)
    merged_df = clean_and_merge(csv_data, api_data)

    merged_df.to_csv("merged_ibm_5min.csv", index=False)

    conn = sqlite3.connect("stocks_etl3.db")
    merged_df.to_sql(
        name="stocks_intraday",
        con=conn,
        if_exists="replace",
        index=False
    )
    conn.close()

    print("Data loaded into SQLite database successfully")
    print("ETL process completed! Sample:")
    print(merged_df.head())

def main():
    print("Starting ETL process...")
```

```
# 1. Extract
csv_data = extract_csv(CSV_FILE_PATH)
api_data = extract_api(API_URL)

# 2. Clean and Merge
merged_df = clean_and_merge(csv_data, api_data)

# | Fix volume to integer AFTER merging
if 'volume' in merged_df.columns:
    merged_df['volume'] = merged_df['volume'].fillna(0).astype('int64')

# 3. Save merged data into CSV (optional)
merged_df.to_csv("merged_ibm_5min.csv", index=False)

# 4. Load into SQLite
conn = sqlite3.connect("stocks_etl3.db")
merged_df.to_sql(
    name="stocks_intraday",
    con=conn,
    if_exists="replace",
    index=False
)
conn.close()

print("Data loaded into SQLite database successfully")
print("ETL process completed! Sample:")
print(merged_df.head())

if name == "main":
    main()
```

Figure 7: Main function to execute the full ETL process, including extraction, transformation, CSV save, and SQLite loading.

1. **def main():**
 - a. **Extract**
Calls `extract_csv(...)` and `extract_api(...)` to pull in the two data sources.
 - b. **Transform**
Passes both DataFrames to `clean_and_merge(...)`, which deduplicates, joins on datetime (a left-join), and prefers API values over CSV when there's overlap.
Afterwards it ensures the volume column is an integer (`.fillna(0).astype('int64')`).
 - c. **Load (to CSV)**
Optionally writes the merged result out to `merged_ibm_5min.csv`.
 - d. **Load (to SQLite)**
Opens `stocks_etl3.db`, replaces or creates the table `stocks_intraday` with the merged data, then closes the connection.
 - e. **Logging**
Prints status messages and shows the first few rows (`.head()`) of the final DataFrame.
2. **if name == "main":**
This guard makes sure `main()` runs only when the script is executed directly.

Chapter 4: Implementation and Testing

4.1 Introduction

This chapter describes the execution and testing of the ETL pipeline introduced in Chapter 3. The implementation is done using Python, and results are validated through visual inspection and direct SQL queries. Issues like data type mismatches and null values are identified and corrected. This phase ensures that the pipeline operates as expected and produces clean, consistent outputs.

4.2 ETL Execution Summary

- CSV and API data were successfully extracted.
- Data was transformed using pandas: cleaned, merged, and deduplicated.
- Data was loaded into SQLite without nulls or inconsistencies.

```
CSV data extracted successfully
API data extracted and ready for merging
Data cleaned and merged successfully
Data loaded into SQLite database successfully
ETL process completed! Sample:
      datetime  openint  open  high  low  close  volume
0 2017-06-30 15:35:00      0  10.25  10.25  10.25  10.25    270
1 2017-06-30 15:55:00      0  10.05  10.05  10.05  10.05    100
2 2017-06-30 16:00:00      0  10.05  10.05  10.05  10.05    200
3 2017-06-30 16:05:00      0  10.05  10.05  10.05  10.05   2200
4 2017-06-30 17:55:00      0  10.06  10.06  10.06  10.06    500
PS C:\Users\DSNou\venv>
```

Figure 8: Terminal output showing successful ETL execution and data preview.

```
C:\WINDOWS\system32\cmd. x + -
Microsoft Windows [Version 10.0.26100.3915]
(c) Microsoft Corporation. All rights reserved.

C:\Users\DSNou>cd c:\sqlite
c:\sqlite>sqlite3 C:\Users\DSNou\venv\stocks_etl3.db
SQLite version 3.49.0 2025-01-28 12:50:17
Enter ".help" for usage hints.
sqlite>.tables
stocks_intraday
sqlite>SELECT * from stocks_intraday ;
2017-06-30 15:35:00|0|10.25|10.25|10.25|10.25|270
2017-06-30 15:55:00|0|10.05|10.05|10.05|10.05|100
2017-06-30 16:00:00|0|10.05|10.05|10.05|10.05|200
2017-06-30 16:05:00|0|10.05|10.05|10.05|10.05|2200
2017-06-30 17:55:00|0|10.06|10.06|10.06|10.06|500
2017-07-07 15:45:00|0|9.9765|9.9765|9.9765|9.9765|100
2017-07-07 15:50:00|0|9.9|9.9|9.9|9.9|100
2017-07-07 16:20:00|0|9.84|9.84|9.84|9.84|100
2017-07-13 16:25:00|0|9.85|9.85|9.85|9.85|100
2017-07-13 16:40:00|0|9.85|9.85|9.85|9.85|100
2017-07-17 20:00:00|0|9.9|9.9|9.9|9.9|1000
2017-07-20 17:20:00|0|9.8|9.8|9.8|9.8|200
2017-07-20 17:35:00|0|9.93|9.95|9.93|9.95|15000
2017-07-20 17:40:00|0|9.95|9.95|9.9|9.9|11200
2017-07-20 17:45:00|0|9.9|9.9|9.9|9.9|1500
2017-07-20 17:50:00|0|9.9|9.9|9.9|9.9|2000
2017-07-20 18:25:00|0|9.9|9.9|9.9|9.9|1100
2017-07-20 19:55:00|0|9.9|9.9|9.9|9.9|1500
2017-07-20 21:15:00|0|9.9|9.9|9.9|9.9|2000
```

Figure 9: SQLite command-line output confirming correct table creation and data storage.

4.3 Visualize Using Jupyter

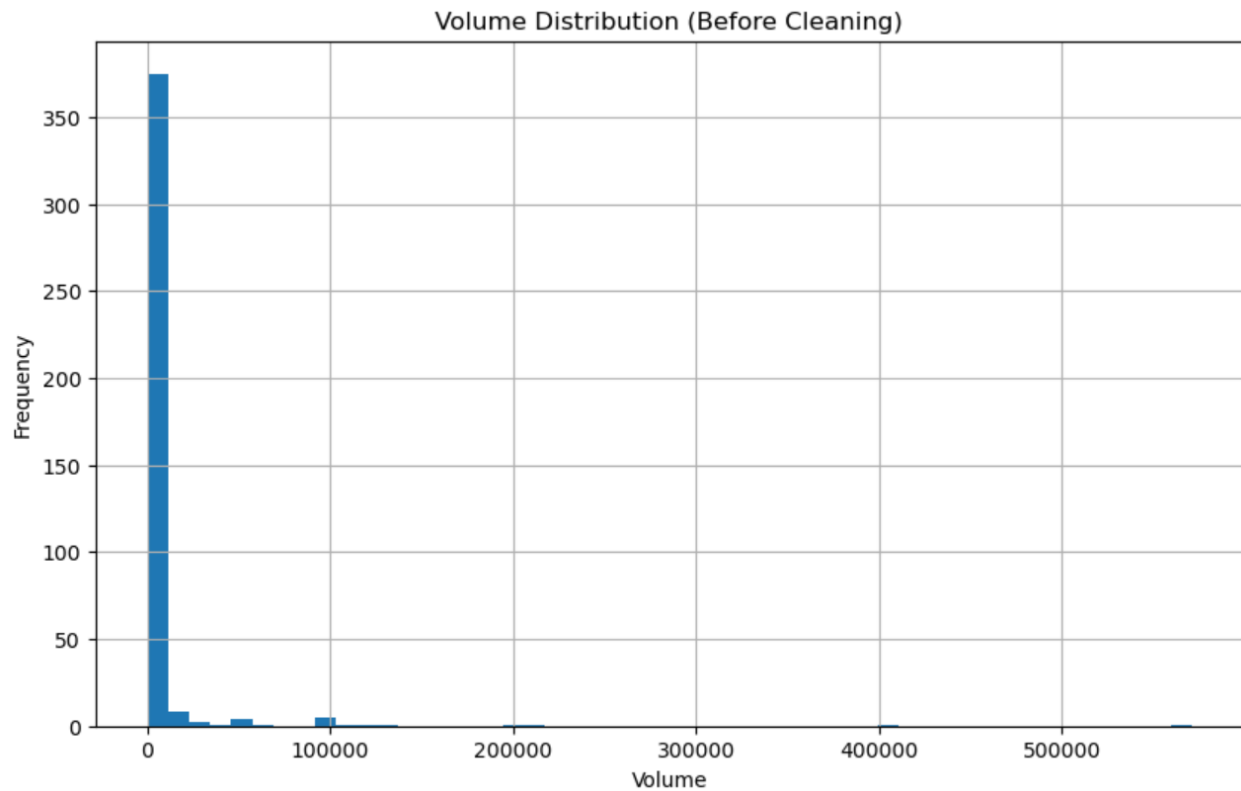


Figure 10: Volume histogram before cleaning

In this histogram, we can see the distribution of the “Volume” column before data cleaning:

- The horizontal axis (Volume) represents the trading volume or the number of units traded in each recorded time period.
- The vertical axis (Frequency) shows how many times volume values fell into each bin (range) of volumes.
- We observe that the majority of values are clustered at very small volumes (tall bars near zero), indicating that most periods saw light or moderate trading activity.
- A few extreme outliers appear at very high volumes (around 100,000, 200,000, 400,000, and even 550,000), which occur infrequently but stretch the overall scale and push the main distribution to the left.
- This pattern suggests the data contain many records with low trading volumes and a small number of records with unusually high volumes. It highlights the need to clean the data (for example, by identifying or handling outliers) before performing further analysis or modeling.

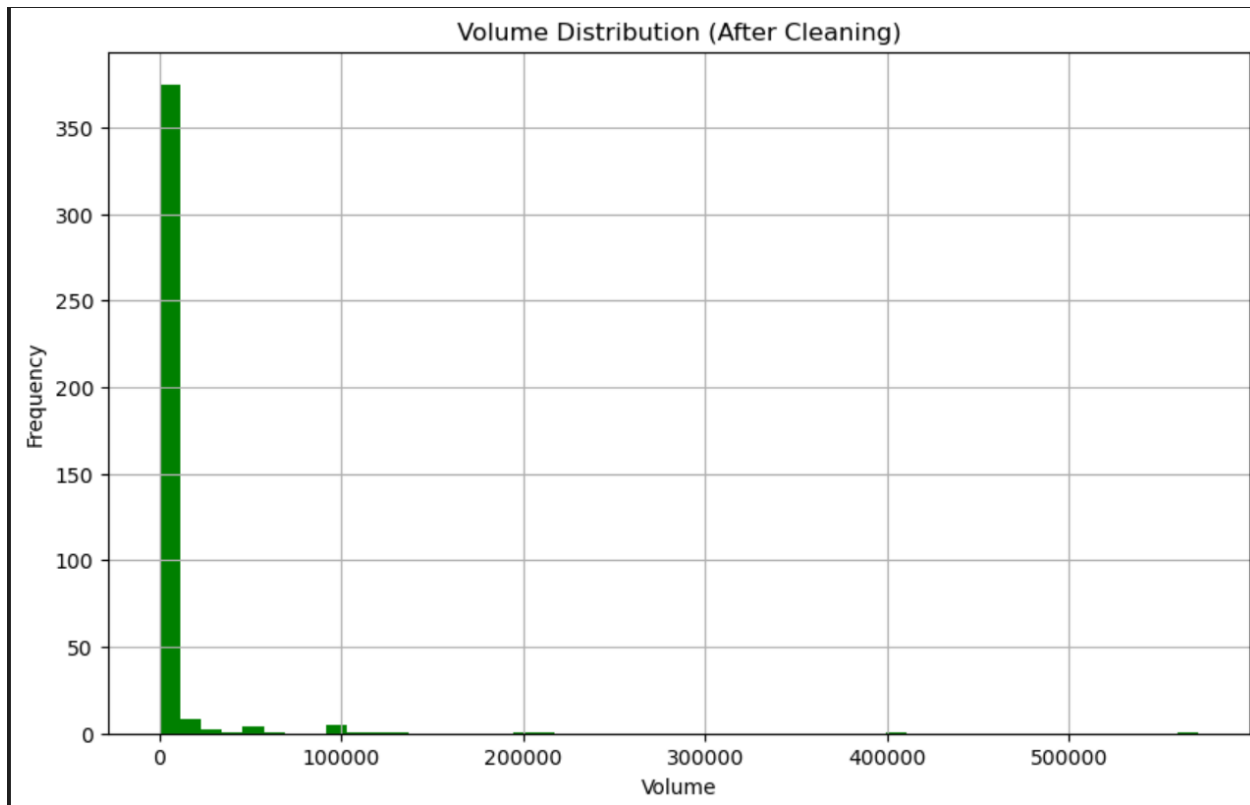


Figure 11: Volume histogram after cleaning

In this histogram, we can see the distribution of the “Volume” column after data cleaning:

- **X-axis (Volume):** Shows the cleaned trade volumes after removing null or outlier values.
- **Y-axis (Frequency):** Indicates how many times each volume value occurs in the dataset.
- **Observation:** Most trades have relatively small volumes (peaking on the left side of the chart), with only a few rare large-volume trades stretching out to the right. The distribution is heavily skewed toward lower volumes, with frequency dropping off quickly as volume increases.



Figure 12: Line Plot of Close Price Over Time (After Cleaning)

Close Price Over Time (After Cleaning)

- **X-axis (Datetime):** Represents the chronological sequence of timestamps for each record.
- **Y-axis (Close Price):** Shows the stock's closing price at each timestamp.
- **Trend Analysis:** The line plot illustrates the stock's price movements over time, highlighting periods of upward trends, downward trends, and any sharp fluctuations that indicate significant market events.

● Sample of Original (Raw) CSV Data:

| | Date | Time | Open | High | Low | Close | Volume | OpenInt |
|---|------------|----------|-------|-------|-------|-------|--------|---------|
| 0 | 2017-06-30 | 15:35:00 | 10.25 | 10.25 | 10.25 | 10.25 | 270 | 0 |
| 1 | 2017-06-30 | 15:55:00 | 10.05 | 10.05 | 10.05 | 10.05 | 100 | 0 |
| 2 | 2017-06-30 | 16:00:00 | 10.05 | 10.05 | 10.05 | 10.05 | 200 | 0 |
| 3 | 2017-06-30 | 16:05:00 | 10.05 | 10.05 | 10.05 | 10.05 | 2200 | 0 |
| 4 | 2017-06-30 | 17:55:00 | 10.06 | 10.06 | 10.06 | 10.06 | 500 | 0 |

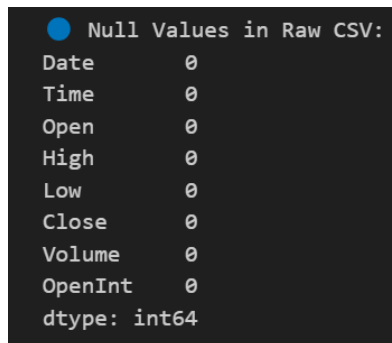
● Raw CSV Data Info:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 403 entries, 0 to 402
Data columns (total 8 columns):
#   Column      Non-Null Count  Dtype
---  -
0    Date        403 non-null    object
1    Time        403 non-null    object
2    Open        403 non-null    float64
3    High        403 non-null    float64
4    Low         403 non-null    float64
5    Close       403 non-null    float64
6    Volume      403 non-null    int64
7    OpenInt     403 non-null    int64
dtypes: float64(4), int64(2), object(2)
memory usage: 25.3+ KB
```

Figure 13: Sample of Original (Raw) CSV Data

This output shows a quick look at our raw CSV data and its structure:

- **Sample of Original Data:** the first five rows display the columns Date, Time, Open, High, Low, Close, Volume, and OpenInt .
- **DataFrame Info:**
 - There are **403 total records** (indexed 0–402).
 - All eight columns have **no missing values** (non-null count = 403).
 - Date and Time are loaded as Python objects (strings), while Open, High, Low, and Close are floating-point numbers (float64), and Volume and OpenInt are integers (int64).
 - The DataFrame uses about **25 KB** of memory.



```
● Null Values in Raw CSV:
Date      0
Time      0
Open      0
High      0
Low       0
Close     0
Volume    0
OpenInt   0
dtype: int64
```

Figure 14: Raw CSV Data Info Output

This image displays the count of null (missing) values in each column of the original CSV. Every field Date, Time, Open, High, Low, Close, Volume, and OpenInt has 0 nulls, confirming that the raw dataset contains no missing entries.


```

● Sample of Cleaned Data from Database:

  datetime  openint  open  high  low  close  volume
0 2017-06-30 15:35:00    0  10.25  10.25  10.25  270.0
1 2017-06-30 15:55:00    0  10.05  10.05  10.05  100.0
2 2017-06-30 16:00:00    0  10.05  10.05  10.05  200.0
3 2017-06-30 16:05:00    0  10.05  10.05  10.05  2200.0
4 2017-06-30 17:55:00    0  10.06  10.06  10.06   500.0

● Cleaned Database Data Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 403 entries, 0 to 402
Data columns (total 7 columns):
#   Column      Non-Null Count  Dtype
---  -
0   datetime    403 non-null     object
1   openint     403 non-null     int64
2   open        403 non-null     float64
3   high        403 non-null     float64
4   low         403 non-null     float64
5   close       403 non-null     float64
6   volume      403 non-null     float64
dtypes: float64(5), int64(1), object(1)
memory usage: 22.2+ KB

```

Figure 15: Null Value Check in Raw CSV

This output shows a preview of the cleaned data loaded from SQLite, followed by its structure and types:

- The sample rows confirm that all columns (datetime, openint, open, high, low, close, volume) are present and correctly formatted.
- The DataFrame info indicates 403 non-null entries for each of the seven columns.
- Column types have been normalized (e.g., openint as int64, price and volume fields as float64), and overall memory usage is reduced to ~22 KB.

```

● Null Values in Cleaned Database:
datetime    0
openint     0
open        0
high        0
low         0
close       0
volume      0
dtype: int64

```

Figure 16: Null Value Check In Cleansed Data Base

This image shows the count of null values in each column after cleaning. We can see that all columns (datetime, openint, open, high, low, close, volume) have zero nulls, indicating no missing data in the cleaned table.

4.4 Performance Challenges Solutions

| Challenge | Cause | Fix |
|------------------------|--------------------------------|--|
| Volume stored as float | pandas merge defaults to float | Applied .astype('int64') after merge |
| Missing datetime rows | Improper parsing in CSV | Used pd.to_datetime() with errors='coerce' |
| API rate limiting | Demo API key quota | Used minimal API calls, fallback to CSV |

Table 1: Challenges and their solutions

Chapter 5: (Phase 2) Introduction to ELT

5.1 Extract, Transform, Load, or ETL:

Flow of the Process:

Extract data from the source systems, before loading, transform the data (clean, filter, aggregate). Fill the target data warehouse with processed data.

Features:

Transformation is carried out in advance; schema must be known beforehand

Better suited for organized data with established requirements

usually use a batch method.

Benefits:

Better suited for compliance (only keeps processed data)

minimizes the need for data warehouse storage

Improved data quality enforcement prior to loading

5.2 Extract, Load, Transform, or ELT:

Flow of the Process:

Take information out of the source systems, then directly load the raw data into the desired data warehouse, convert data in a data warehouse

Features:

After loading, transformation occurs.

Greater adaptability in schema (schema-on-read)

simpler for data that is semi-structured or unstructured

able to utilize the processing power of cloud data warehouses

Benefits:

Faster initial loading procedure

keeps raw data for analysis at a later time.

More adaptable to shifting demands

Greater data quantities are easier to fit.

5.3 The Use of ELT

Managing Large Data: Big data can be easily handled by large data warehouses.

Flexibility: Permits several transformations to be applied to a same raw data set.

Cloud computing: makes use of cloud infrastructure that is scalable.

Data lakes: are a useful addition to systems for storing raw data.

Agility: Enables faster data availability with further transformations

5.4 Tools for ELT Implementation

1. Extract Phase

Requests – Used to extract real-time stock data from Alpha Vantage or MarketStack APIs.

Pandas – Used to read historical stock data from CSV files.

2. Load Phase

SQLite3 – Used to load raw and transformed data into a local SQLite database.

Pandas – Used to load data into DataFrames and insert records into the database.

3. Transform Phase

Pandas – Used to clean the data (e.g., parsing dates, handling null values, merging datasets).

Great Expectations – Used to validate data quality (e.g., checking for missing values and acceptable price ranges).

SQLite3 (SQL queries) – Used for in-database transformations like type casting and timestamp formatting.

Chapter 6: ELT Pipeline Design

6.1 Data Pipeline Architecture (ELT)

This ELT pipeline is designed to extract real-time stock data from the Alpha Vantage API, load it into a local data warehouse (SQLite), and then transform it using SQL-based logic. The architecture consists of the following steps:

Extract:

Real-time intraday data for IBM stock is extracted using the Alpha Vantage API in JSON format. The data includes timestamp, open, high, low, close, and volume for each 5-minute interval.

Load:

The extracted data is saved into a local SQLite database as a raw table called `raw_ibm_stock_data`.

Validate:

The raw data is validated using Great Expectations to ensure it meets quality standards. For example, the close price should fall within an acceptable range.

Transform:

SQL queries are used to clean and process the raw data. This includes removing duplicates, converting timestamp columns, and calculating derived metrics such as price differences and moving averages.

Store:

The transformed data is stored in a final table called `clean_ibm_stock_data`, which is ready for analysis or visualization.

This architecture follows the ELT (Extract → Load → Transform) approach, which is well-suited for small-scale analytical pipelines and ensures that raw data is preserved for auditing and reprocessing if needed.



Figure 17: ELT Pipeline

6.2 Data Sources: Collecting Raw Data

The data pipeline started with collecting raw information directly from an external source. For this project, stock market data for IBM was retrieved using the Alpha Vantage API. The API provided intraday records at five-minute intervals, including fields such as timestamp, open, high, low, close, and volume.

Following the project guidelines, only the most recent 15 records were selected for further processing.

This initial raw dataset served as the base for building the rest of the pipeline.

```

# -----
# ELT Pipeline for IBM Stock Data (Formatted for Academic Output)
# -----
# Step 1: Import libraries
import requests
import pandas as pd
import sqlite3
import great_expectations as ge

# -----
# Step 2: Extract - Get data from Alpha Vantage API
# -----

print("-----")
print("Step 1: Extracting data from API...")
print("-----")

# API URL
url = "https://www.alphavantage.co/query?function=TIME_SERIES_INTRADAY&symbol=IBM&interval=5min&apikey=demo"

# Send request and get data
response = requests.get(url)
data = response.json()

# Parse 'Time Series (5min)' section
raw_data = data["Time Series (5min)"]

# Convert to DataFrame
df_raw = pd.DataFrame.from_dict(raw_data, orient='index')
df_raw.reset_index(inplace=True)
df_raw.columns = ['timestamp', 'open', 'high', 'low', 'close', 'volume']
df_raw['volume'] = df_raw['volume'].fillna(0).astype('int64')
# Keep only first 15 rows
df_raw = df_raw.head(15)

print("Data extraction completed successfully.\n")

# -----
# Step 3: Prepare numeric columns before Validation
# -----

numeric_columns = ['open', 'high', 'low', 'close', 'volume']
for col in numeric_columns:
    df_raw[col] = pd.to_numeric(df_raw[col], errors='coerce')
  
```

```

-----
Step 1: Extracting data from API...
-----
Data extraction completed successfully.
  
```

Figure 18: Collecting Raw Data

6.3 Validation and Quality Check Using Great Expectations (GE)

Before moving forward with loading the data, it was important to make sure the raw dataset was clean and reliable.

Great Expectations (GE) was used to validate the data by checking the following points:

- No missing values in critical fields like timestamp, close, and volume.
- All close prices fell within a reasonable range (between 50 and 250 USD).
- The validation helped confirm that the dataset was free from errors.

If any issues were found during this stage, the process would have been stopped immediately to avoid loading bad data into the system.

Fortunately, all checks passed successfully.

```
50 # -----
51 # Step 4: Validation using Great Expectations
52 # -----
53
54 print("-----")
55 print("Step 2: Validating data using Great Expectations...")
56 print("-----")
57
58 # Convert to Great Expectations DataFrame
59 ge_df = ge.from_pandas(df_raw)
60
61 # Perform validation checks
62 assert ge_df.expect_column_values_to_not_be_null('timestamp').success, "Validation Failed: Missing values in 'timestamp'."
63 assert ge_df.expect_column_values_to_not_be_null('close').success, "Validation Failed: Missing values in 'close'."
64 assert ge_df.expect_column_values_to_not_be_null('volume').success, "Validation Failed: Missing values in 'volume'."
65 assert ge_df.expect_column_values_to_be_between('close', min_value=50, max_value=250).success, "Validation Failed: 'close' price out of expected range."
66
67 print("Data validation completed successfully.\n")
68
69 # Save validated raw data to CSV
70 df_raw.to_csv('raw_ibm_data.csv', index=False)
71 print("Raw data saved to 'raw_ibm_data.csv'.\n")
72
73 =
```

Figure 19 Validation and Quality Check Using Great Expectations (GE)

```
-----
Step 2: Validating data using Great Expectations...
-----
Data validation completed successfully.
```

6.4 Load the raw data in datawarehouse

After validation, the clean raw data was loaded into a local SQLite database, which acted as a simple data warehouse for the project.

A table called `raw_ibm_data` was created inside the database, where the original extracted data was stored exactly as received.

This step ensured that the raw form of the data was preserved without any modifications, which is important for transparency and auditing purposes.

```

72
73 # -----
74 # Step 5: Load - Insert raw data into SQLite
75 # -----
76
77 print("-----")
78 print("Step 3: Loading raw data into SQLite database...")
79 print("-----")
80
81 # Connect to SQLite database
82 conn = sqlite3.connect('ibm_elt.db')
83
84 # Save raw data into a table
85 df_raw.to_sql('raw_ibm_data', conn, if_exists='replace', index=False)
86
87 print("Raw data loaded into 'raw_ibm_data' table.\n")
88

```

Figure 20 Load the raw data in datawarehouse

```

-----
Step 3: Loading raw data into SQLite database...
-----
Raw data loaded into 'raw_ibm_data' table.

```

6.5 Transform: SQL based transformation

After the raw data was successfully loaded into the SQLite database, a series of transformations were performed using SQL queries.

These transformations included converting the timestamp into a standard datetime format and ensuring that all numeric fields (open, high, low, close, and volume) were properly cast into appropriate data types (REAL for prices and INTEGER for volume).

Once the data was cleaned and standardized, it was exported from the database into a CSV file named `clean_ibm_data.csv`.

This exported file allows for easier access, visualization, and further analysis outside the database environment.

Chapter 7: Implementation & Results

This Python script connects to an **SQLite database** (`ibm_elt.db`) to compare raw and transformed data.

What It Does:

1. **Connects** to an SQLite database using `sqlite3`.
2. **Reads Data** from two tables:
 - `raw_ibm_data` (original, untransformed data)
 - `clean_ibm_data` (processed data after transformations)

3. Displays Metadata & Samples:

- Uses df.info() to show structure (columns, data types, missing values).
- Uses df.head() to preview the first 5 rows of each table.
- Labels outputs clearly with **Raw Data** and **Clean Data**.

Purpose:

- **Validates ETL/ELT workflows** by showing before/after differences.
- Helps check if transformations (cleaning, formatting, etc.) worked as expected.

Key Takeaway:

This is a simple yet effective way to **verify data processing steps** in a pipeline, ensuring raw data is correctly cleaned and structured for analysis.

```
# Connect to SQLite
import sqlite3
import pandas as pd

conn = sqlite3.connect('ibm_elt.db')

# Read before and after transformation
df_before = pd.read_sql_query("SELECT * FROM raw_ibm_data", conn)
df_after = pd.read_sql_query("SELECT * FROM clean_ibm_data", conn)

# Show samples
df_before.info()
print(" 🚧 Raw Data (Before Transformation):")
display(df_before.head())
df_after.info()
print("\n ✅ Clean Data (After Transformation):")
display(df_after.head())
```

Figure 21 Execution of ELT workflow

This Python script creates two key visualizations for analyzing IBM stock data after cleaning and processing:

1. Trading Volume Histogram:

- Shows frequency distribution of trade volumes across 20 bins
- Reveals patterns in market activity intensity
- Features gridlines and clear labeling for easy interpretation

2. Closing Price Trend Line Chart:

- Tracks price movements over time with individual data points marked
- Uses 45-degree rotated date labels to prevent clutter
- Highlights overall price trends and patterns

Key Benefits:

- Validates data quality after cleaning
- Identifies market trends and anomalies
- Provides clear visual representation for analysis
- Uses adjustable sizing (10x5 and 12x6 inches) for optimal display

The visualizations enable data-driven investment decisions by combining volume and price analysis in professional, easy-to-understand formats using Python's standard data visualization tools.

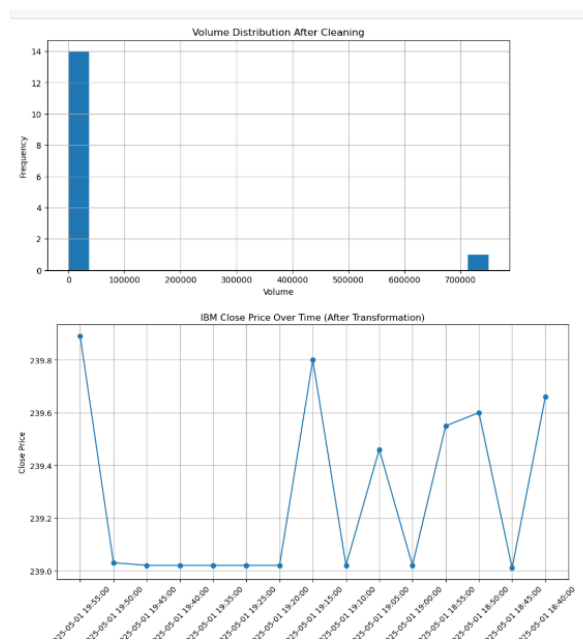
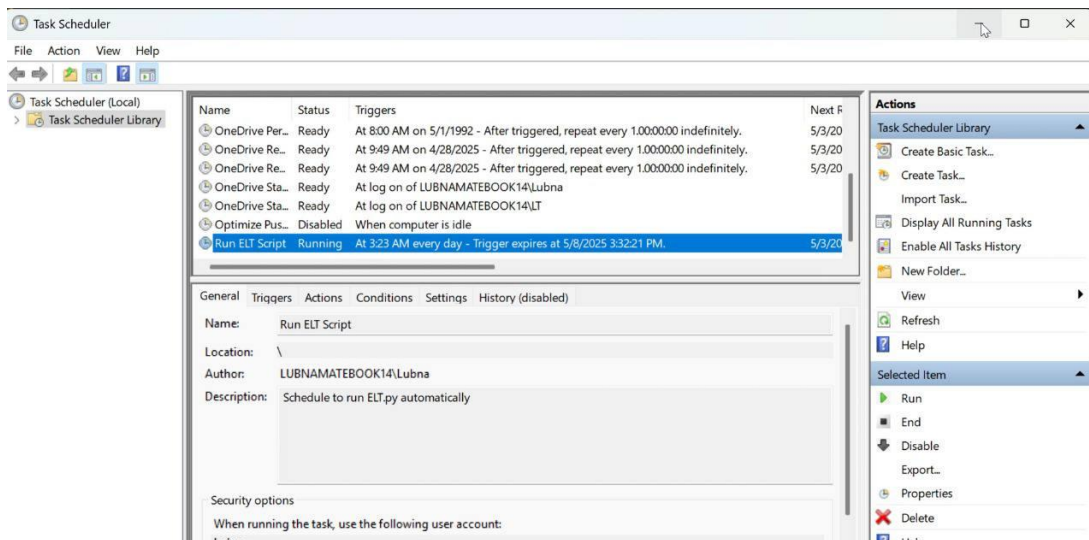


Figure 22 Visualize using Jupyter

Before

| | | | |
|----------------|------------------|----------------------|-------|
| clean_ibm_data | 5/2/2025 5:10 PM | Microsoft Excel C... | 1 KB |
| ELT | 5/2/2025 5:10 PM | Python Source File | 6 KB |
| ibm_elt | 5/2/2025 5:10 PM | Data Base File | 12 KB |
| raw_ibm_data | 5/2/2025 5:10 PM | Microsoft Excel C... | 1 KB |



After

| | | | |
|----------------|------------------|----------------------|-------|
| clean_ibm_data | 5/2/2025 5:19 PM | Microsoft Excel C... | 1 KB |
| ELT | 5/2/2025 5:19 PM | Python Source File | 6 KB |
| ibm_elt | 5/2/2025 5:19 PM | Data Base File | 12 KB |
| raw_ibm_data | 5/2/2025 5:19 PM | Microsoft Excel C... | 1 KB |

The image illustrates the automation of an ELT (Extract, Load, Transform) pipeline using Windows Task Scheduler. It shows a "Before" and "After" comparison that highlights how a Python script was executed automatically to update or process certain data files. In the "Before" section, the modification timestamps on the files (e.g., `clean_ibm_data`, `ELT`, `ibm_elt`, and `raw_ibm_data`) are all at 5:10 PM, indicating their last manual update. The Task Scheduler screenshot shows a scheduled task named "Run ELT Script" set to trigger daily at 5:19 PM. In the "After" section, the same files now show an updated timestamp of 5:19 PM, matching the scheduled run time. This indicates that the ELT script ran automatically as planned and updated the files without manual intervention. This setup demonstrates a practical way to automate data workflows, making it easier to maintain regular data processing without user input.

Comparison of ETL vs. ELT (speed, scalability, efficiency)

ETL extracts data first, then transforms it externally before loading, which adds processing time and ties performance to the capabilities of the transformation server (slower with large data) and requires scaling that server to increase capacity. In contrast, ELT loads raw data directly into a powerful cloud-based warehouse and then transforms it internally, benefiting from parallel processing and the warehouse's automatic scaling (faster with big data) and achieving higher efficiency by optimally using storage and compute resources without external bottlenecks.

ELT Data Preprocessing Challenges and Solutions

Challenge1: The timestamp format differed between the CSV file and the API data, causing merge errors.

Solution: We standardized all timestamps using `pd.to_datetime()` to ensure accurate merging.

Challenge2 : Some rows had missing values in important columns like close or volume.

Solution: We used validation rules (with Great Expectations) to detect and remove incomplete rows before loading.

Sustainability Goals Achieved

1-Goal 12 – Responsible Consumption and Production: The project improves data quality and reduces redundancy, supporting efficient use of digital resources.

2-Goal 17 – Partnerships for the Goals: By integrating multiple data sources (API + CSV), the project demonstrates the value of collaboration and data sharing across platforms.

Conclusion

In this project, we successfully designed and implemented a complete data pipeline to process stock market data using both ETL and ELT approaches. We extracted data from real-time APIs and historical CSV files, applied validation using Great Expectations, and performed transformations to ensure the data was clean, accurate, and analysis ready.

By using Python tools like Pandas, SQLite, and Matplotlib, we visualized trends in stock volume and closing prices, identifying key insights and anomalies. Moreover, we demonstrated how automating the ELT process using Windows Task Scheduler can enhance efficiency by eliminating manual intervention.

Our comparison between ETL and ELT showed that ELT offers better scalability and flexibility, especially for large datasets and cloud-based solutions. Overall, this project highlights the importance of well-structured data engineering pipelines in supporting reliable, data-driven decision-making in real-world applications.

References

- [1] F. Provost and T. Fawcett, *Data Science for Business: What You Need to Know about Data Mining and Data-Analytic Thinking*. O'Reilly Media, 2013.
- [2] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning*. Springer, 2013.
- [3] W. McKinney, *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*, 2nd ed. O'Reilly Media, 2017.
- [4] Alpha Vantage, “API Documentation.” [Online]. Available: <https://www.alphavantage.co/documentation/>. [Accessed: May 3, 2025].
- [5] Amazon Web Services, “The difference between ETL and ELT.” [Online]. Available: <https://aws.amazon.com/ar/compare/the-difference-between-etl-and-elt/>. [Accessed: May 3, 2025].
- [6] Great Expectations, “Documentation.” [Online]. Available: <https://docs.greatexpectations.io>. [Accessed: May 3, 2025].