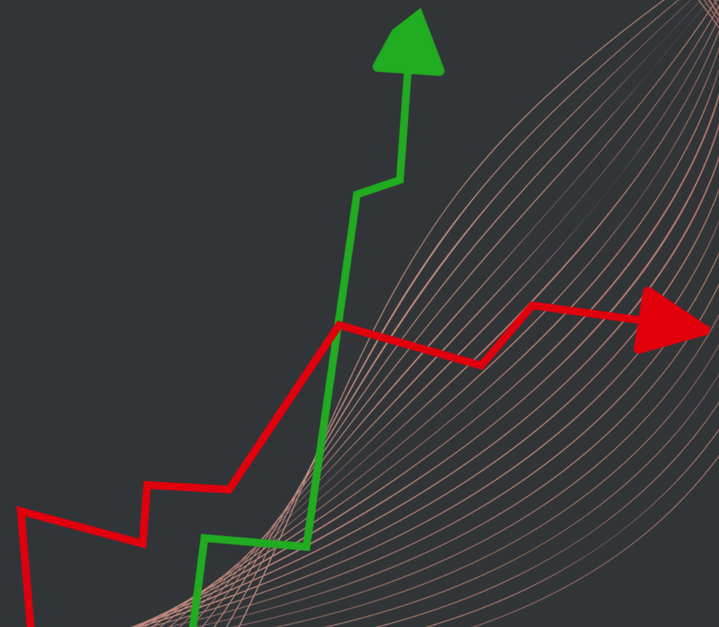
StockFlow Analytics

Data engineering (DS43 7) I I sem 144 6 AH

dr.shakila bashir

Haya alnashwan 444008713
lubna bin taleb 444008721
fajer alshuwier 444008709
nourah albarrak 444008746
lina aljsir 444008726



Contents

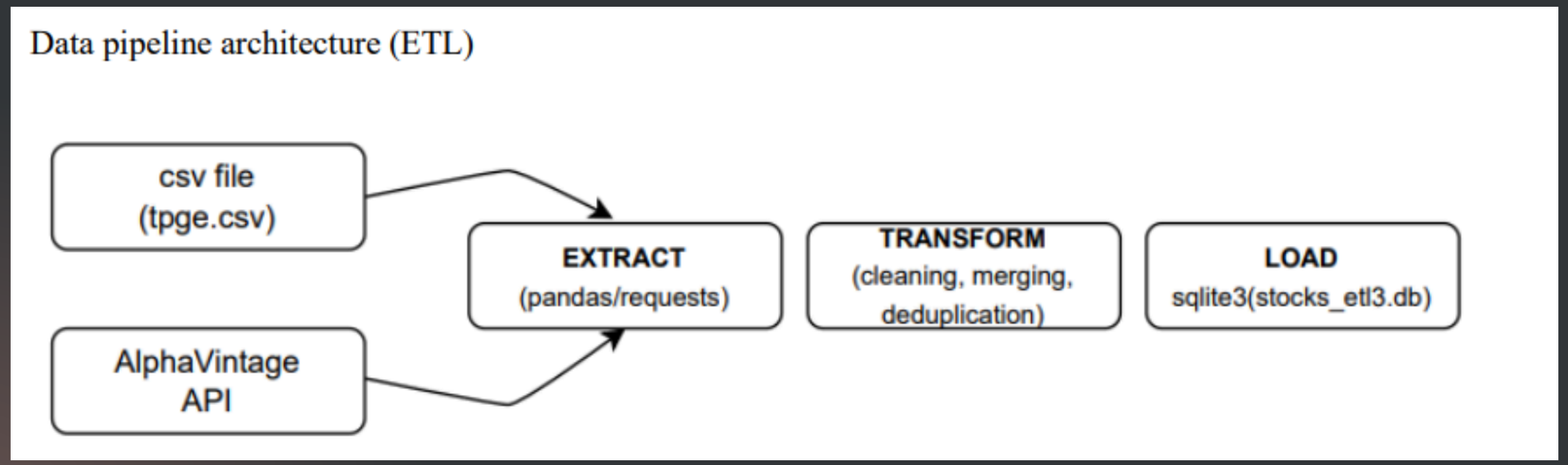
Intorduction:

We built an ELT and ETL pipeline to process stock market data from two sources: real-time prices via an API and historical data from a CSV file. We faced challenges like missing values and inconsistent formats, so we designed an automated pipeline to extract, validate, and load data into a database, ensuring high data quality.

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).



Step-by-step ETL process execution using python program

1-Extract

- **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.

```python
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))
```

Step-by-step ETL process execution using python program

2-Transform
cleaning CSV data

In this Transform step we clean the CSV: lower-case headers, combine the Date and Time columns into a single timestamp, cast prices and volume to numeric types, drop invalid rows, and pass the tidy frame to the merge routine.

```python
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()
```

Step-by-step ETL process execution using python program

2-Transform clean API data

we standardised column names and timestamps, cast prices + volume to numbers, dropped bad or duplicate rows, merged CSV with API on datetime (favoring API values), and recast volume to integer—leaving one clean DataFrame ready for loading.

```
        })
    )
    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()
```

```
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"
```

Step-by-step ETL process execution using python program

2-Transform

In this step we merge the two cleaned data sets: we first drop any duplicate timestamps, then perform a left-join on datetime so every CSV row is preserved, and whenever both sources provide a value we keep the fresher API number, falling back to the CSV only when the API field is empty.

```
)

# 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
```

```
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")
```

Step-by-step ETL process execution using python program

3- Load

In the Load phase we first persist the merged DataFrame to a safety copy merged_ibm_5min.csv, then open stocks_etl3.db, overwrite (or create) the table stocks_intraday, bulk-insert all 403 cleaned rows, and finally close the connection—leaving the data ready for the Visualization.

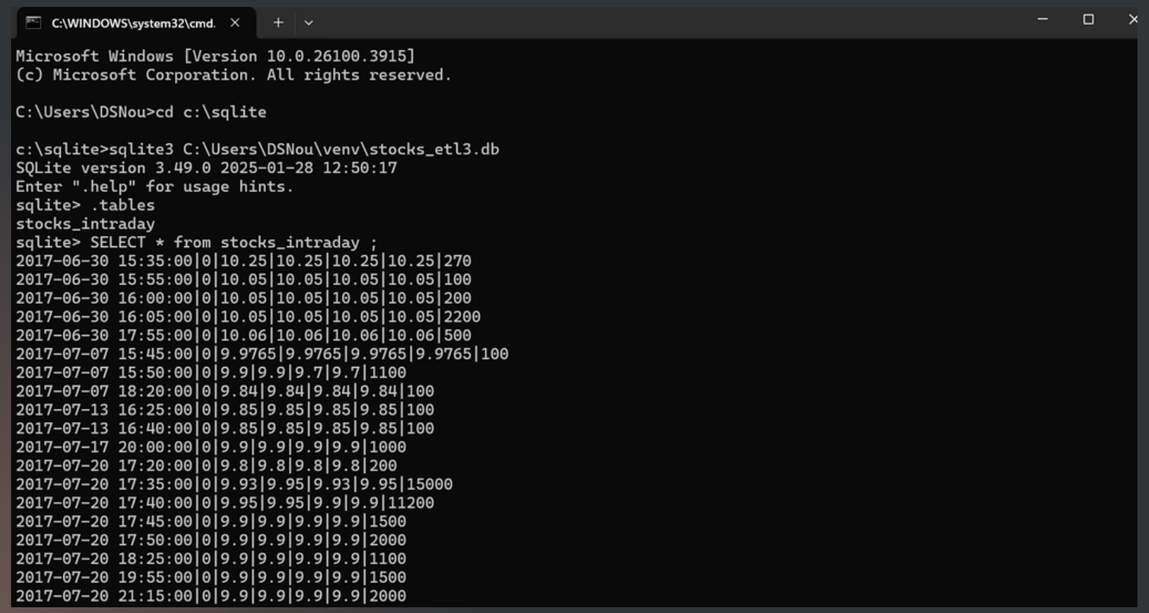
```
def main():
    print("Starting ETL process...")

    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()
```

Step-by-step ETL process execution using python program

4-ETL Execution Summary

Run completed end-to-end: both sources pulled, data cleaned and merged, and the final 403-row dataset successfully written to stocks_intraday in stocks_etl3.db

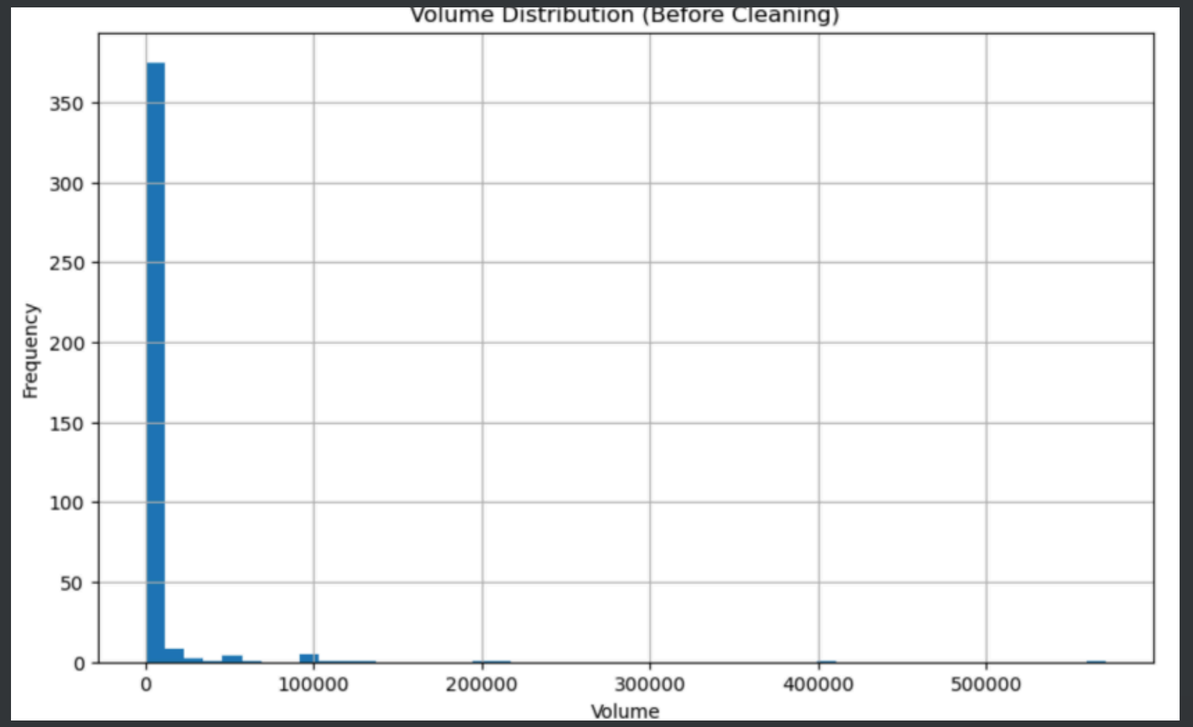



```
CSV data extracted successfully
API data extracted and ready for merging
Data cleaned and merged successfully
Data loaded into SQLite database successfully
ETL process completed! Sample:
             datetime  openint   open   high    low  close  volume
0 2017-06-30 15:35:00         0  10.25  10.25  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>
```

Visualize Using Jupyter

In this histogram:

- The horizontal axis represents trading volume per period.
- The vertical axis shows how often each volume range occurs.
- Most volumes fall into a low-to-moderate range, with bars clustered near smaller values, indicating light to moderate trading activity.

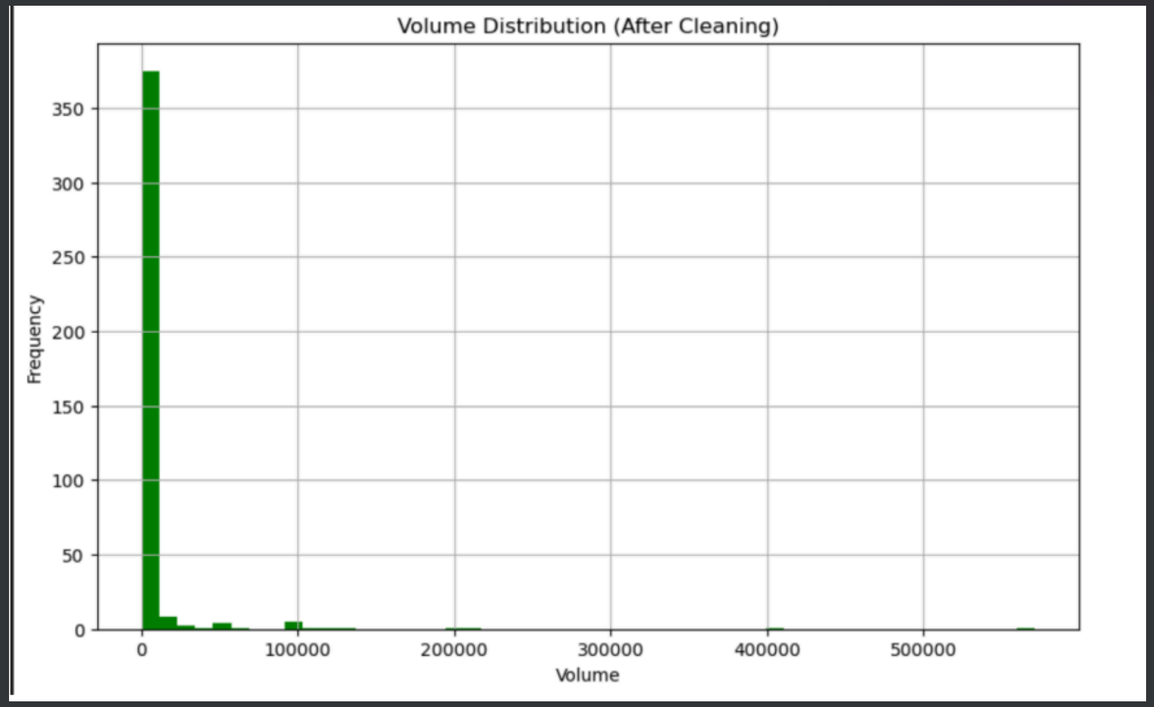


Sample of Original (Raw) CSV Data:

	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

Visualize Using Jupyter

In this histogram:
• X-axis (Volume): Shows the cleaned trade volumes after removing null .
• Y-axis (Frequency): Indicates how many times each volume value occurs in the dataset.
The histogram is heavily skewed to the left, showing that most trades occur at low volumes, with frequency rapidly decreasing as volume increases.



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	10.25	270.0
1	2017-06-30 15:55:00	0	10.05	10.05	10.05	10.05	100.0
2	2017-06-30 16:00:00	0	10.05	10.05	10.05	10.05	200.0
3	2017-06-30 16:05:00	0	10.05	10.05	10.05	10.05	2200.0
4	2017-06-30 17:55:00	0	10.06	10.06	10.06	10.06	500.0

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

Introduction to 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 database

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

Tools for ETL implementation

1. Extrac Phase
Requests – Used to extract real-time stock data from Alpha Vantage or MarketStack API

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, mergin 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.

Step-by-step ELT process execution using python program

Extract:

In this step, we used the Alpha Vantage API to extract real-time stock data for IBM. The data includes important fields like timestamp, open, high, low, close, and volume. We sent a request using Python and received the data in JSON format, which we then converted into a DataFrame using Pandas.

```
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')


df_raw = df_raw.head(15)

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

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

Step-by-step ELT process execution using python program

Validation and Quality Check Using Great Expectations (GE):

Before transforming the data, we used a data validation tool called Great Expectations. We defined simple rules to make sure the data quality is acceptable. For example, we checked that the 'close' price is within a valid range (between 50 and 250), and that there are no missing values in the timestamp or price columns.

```
print("-------------------------------------------------")
print("Step 2: Validating data using Great Expectations...")
print("-------------------------------------------------")

# Convert to Great Expectations DataFrame
ge_df = ge.from_pandas(df_raw)

# Perform validation checks
assert ge_df.expect_column_values_to_not_be_null('timestamp').success, "Validation Failed: Missing values in 'timestamp'."
assert ge_df.expect_column_values_to_not_be_null('close').success, "Validation Failed: Missing values in 'close'."
assert ge_df.expect_column_values_to_not_be_null('volume').success, "Validation Failed: Missing values in 'volume'."
assert ge_df.expect_column_values_to_be_between('close', min_value=50, max_value=250).success, "Validation Failed: 'close' price out

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

# Save validated raw data to CSV with comma separator
df_raw.to_csv('raw_ibm_data.csv', index=False, sep=',')

print("Raw data saved to 'raw_ibm_data.csv'.\n")
```

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

Step-by-step ELT process execution using python program

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.

```
# ------------------------------------------------------------
# Step 5: Load - Insert raw data into SQLite
# ------------------------------------------------------------

print("-----------------------------------------------------")
print("Step 3: Loading raw data into SQLite database...")
print("-----------------------------------------------------")

# Connect to SQLite database
conn = sqlite3.connect('ibm_elt.db')

# Save raw data into a table
df_raw.to_sql('raw_ibm_data', conn, if_exists='replace', index=False)

print("Raw data loaded into 'raw_ibm_data' table.\n")
```

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

Step-by-step ELT process execution using python program

Transform: SQL based transformation

This step involves cleaning and converting the raw data using SQL queries inside the database. We changed data types to ensure consistency — for example, prices were cast to REAL and volume to INTEGER. We also reformatted timestamps

```
print("-------------------------------------------------")
print("Step 4: Transforming raw data inside SQLite...")
print("-------------------------------------------------")

# Drop old table if exists
conn.execute("DROP TABLE IF EXISTS clean_ibm_data;")

# SQL Query to create clean data table
transform_query = """
CREATE TABLE clean_ibm_data AS
SELECT
    datetime(timestamp) AS timestamp,
    CAST(open AS REAL) AS open,
    CAST(high AS REAL) AS high,
    CAST(low AS REAL) AS low,
    CAST(close AS REAL) AS close,
    CAST(volume AS REAL) AS volume
FROM raw_ibm_data;
"""

# Execute the query
conn.execute(transform_query)
conn.commit()

print("Data transformation completed. Cleaned data stored in 'clean_ibm_data' table.\n")
```

```
-------------------------------------------------
Step 4: Transforming raw data inside SQLite...
-------------------------------------------------
Data transformation completed. Cleaned data stored in 'clean_ibm_data' table.
```

Execution of ETL workflow

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) o 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. o 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.

```python
# 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())
```

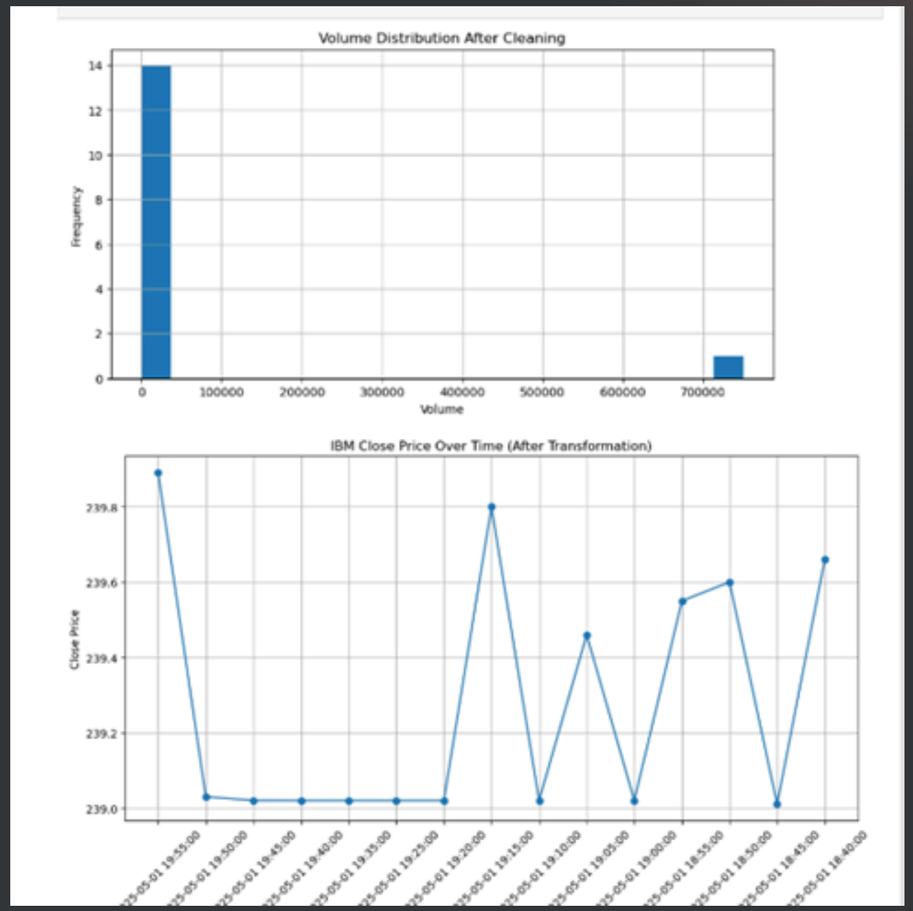
Visualize Using Jupyter

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

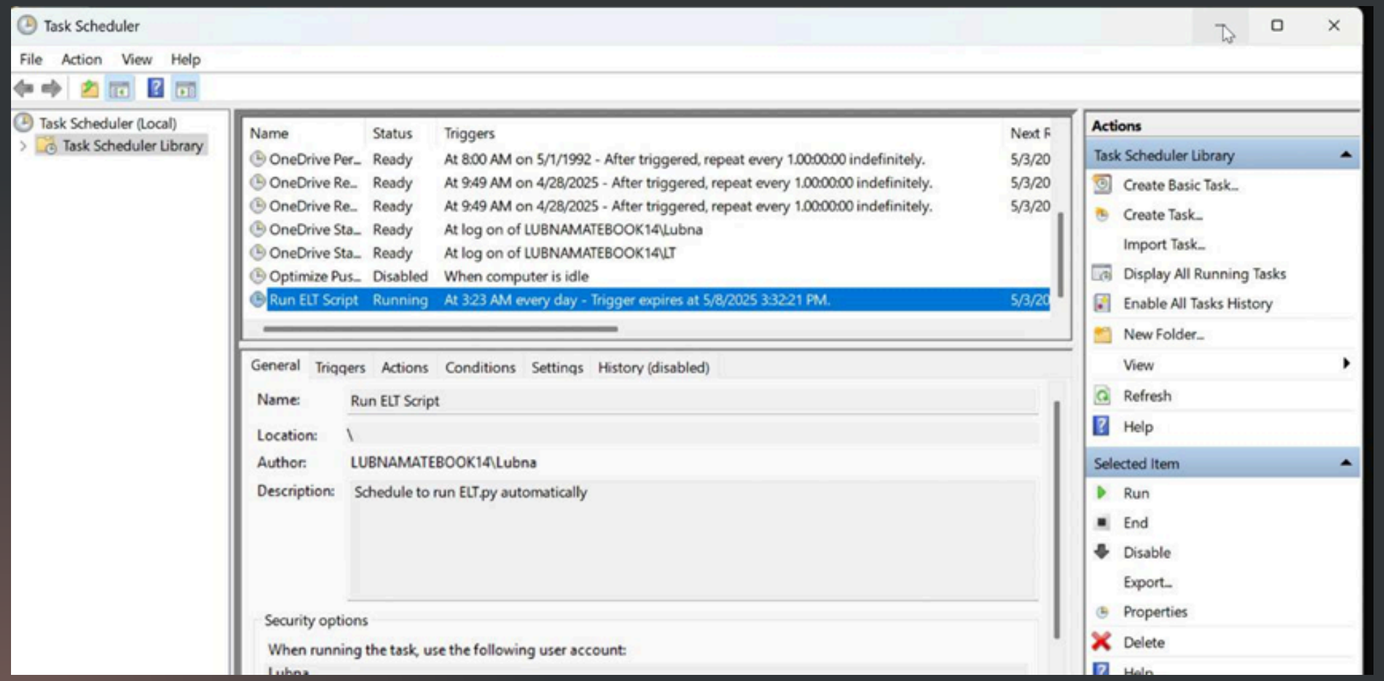
Key Benefits:
- Validates data quality after cleaning
- Identifies market trends and anomalies
- Provides clear visual representation for analysis
- Uses adjustable sizing (10x5 and 12x6 inches) for optimal display

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.



Automation of ELT pipeline using Task scheduler

To simulate a real data pipeline, we automated the ELT process using Windows Task Scheduler. Although the demo API does not return new data, we scheduled the Python script to run daily. If we were using a live API key, this would keep the data fresh automatically without manual work.



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

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.

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.

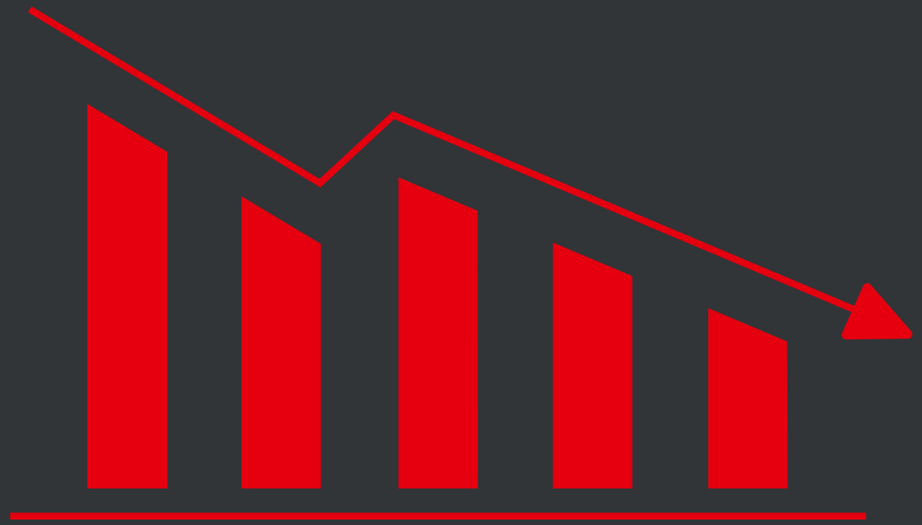
Sustainability Goals Achieved

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

We built a data pipeline using ETL and ELT to process real-time and historical stock data. With Python tools, we cleaned, validated, and visualized the data. Automating the ELT process improved efficiency. ELT proved more scalable, showing how strong pipelines support real-world, data-driven decisions.



Thank you for listening