

# DATA INTEGRATION WITH APIS AND ETL PROCESSES

## PROJECT : ETL PIPELINE WITH PYTHON

### Introduction

To enhance my understanding of the ETL (Extract, Transform, Load) process, I decided to retrieve financial data from both Alpha Vantage and Yahoo Finance. Each source has its own data retrieval method: Alpha Vantage requires an API key, whereas Yahoo Finance provides direct access without a formal API, requiring a different extraction approach.

Given these differences, I implemented two distinct data extraction methods. The goal of this project was to process the extracted data, transform it into a structured format, and ultimately store it in a PostgreSQL database via pgAdmin 4, using a CSV file as an intermediary.

### Alpha-Vantage model

#### 1. Extraction Process

As previously mentioned, Alpha Vantage requires an API key, which can be obtained directly from their website. However, the free API key comes with limitations on the number of requests and accessible functionalities.

To leverage Alpha Vantage effectively, I chose to retrieve Forex EUR/USD exchange rates on a monthly basis over a period of 230 months. Compared to Yahoo Finance, Alpha Vantage is particularly advantageous when dealing with commodities and Forex data, as it provides a more extensive dataset for these asset classes.

```
EXTRACTION

We define the API key and specify the currency pair (EUR/USD) for which we will retrieve foreign exchange data.

# API ID
API_KEY = "J20ZYB9ZL1RBF9ZP"
FROM_SYMBOL = "EUR"      # Devise de départ
TO_SYMBOL = "USD"        # Devise cible

✓ 0.0s Python

We construct the request URL with the necessary parameters and send a GET request to retrieve the monthly foreign exchange data. If the request fails, an exception is raised to indicate the error. We use a scalable solution in the ID so it's easy to change the currencies.

# URL
url = f"https://www.alphavantage.co/query?function=FX_MONTHLY&from_symbol={FROM_SYMBOL}&to_symbol={TO_SYMBOL}&apikey={API_KEY}"

# GET Request
response = requests.get(url)
if response.status_code != 200:
    raise Exception(f"Erreur API: {response.status_code} - {response.text}")

data = response.json()

✓ 0.7s Python

We extract the "Time Series FX (Monthly)" section from the API response, which contains the monthly exchange rate data.

fx_data = data["Time Series FX (Monthly)"]

✓ 0.0s Python
```

Figure 1

It illustrates the extraction process for Alpha Vantage, highlighting the API key definition and the various URL parameters required to retrieve the data.

#### 2. Transformation Process

The first step in the transformation process was to convert the extracted JSON data into a Pandas DataFrame. To achieve this, I used a dictionary structure in Python, following the method described in the Alpha Vantage documentation. Since I had previously encountered data quality issues with Alpha Vantage, I decided to implement key validation checks. I tested for missing values to detect any incomplete data points and examined

the dataset for outliers by verifying whether exchange rates fell within a reasonable range of 0.5 to 2.0. These steps were necessary to ensure that the dataset was reliable and suitable for further analysis.

```
## Convert
df = pd.DataFrame.from_dict(fx_data, orient="index")
df.index = pd.to_datetime(df.index)
df.columns = ["Open", "High", "Low", "Close"]
df = df.astype(float)
df = df.sort_index()

## Missing Values
missing_values = df.isnull().sum()
print("Check missing values :\n", missing_values)

## Outlier Detection
for col in ["Open", "High", "Low", "Close"]:
    if (df[col] < 0.5).any() or (df[col] > 2.0).any():
        print("Warning : Outliers {col} !")
```

✓ 0.0s

Check missing values :  
Open 0  
High 0  
Low 0  
Close 0  
dtype: int64

Python

**Figure 2**  
The Transformation Process.

To enhance our analysis, I computed several key metrics based on the FX monthly data. These indicators provide insights into price movements, market volatility, and potential trading opportunities. Since it is significantly easier to compute such metrics in Python rather than in SQL, I decided to add them directly to the DataFrame before loading the data into PostgreSQL :

- Monthly Change (Open to Close) %  
Measures the percentage change from the opening to the closing price.  
Formula:  $((\text{Close} - \text{Open}) / \text{Open}) * 100$
- Range %  
Represents the percentage difference between the highest and lowest price within the month.  
Formula:  $((\text{High} - \text{Low}) / \text{Low}) * 100$
- Closing Momentum  
Indicates whether the closing price is closer to the highest or lowest price of the month.  
Formula:  $(\text{Close} - \text{Low}) / (\text{High} - \text{Low})$
- Volatility Index  
Measures the relative volatility of the exchange rate based on the high-low range.  
Formula:  $((\text{High} - \text{Low}) / \text{Close}) * 100$
- Momentum Score  
Evaluates the strength of the price movement relative to its range.  
Formula:  $(\text{Close} - \text{Open}) / (\text{High} - \text{Low})$

These additional columns allow for a more comprehensive evaluation of exchange rate dynamics, facilitating deeper analysis before storing the data in the SQL database.

```
## Monthly Change (Open to Close) %
df["Monthly Change (Open to Close) %"] = ((df["Close"] - df["Open"]) / df["Open"]) * 100

## Range %
df["Range %"] = ((df["High"] - df["Low"]) / df["Low"]) * 100

## Closing Momentum
df["Closing Momentum"] = (df["Close"] - df["Low"]) / (df["High"] - df["Low"])

## Volatility Index
df["Volatility Index"] = ((df["High"] - df["Low"]) / df["Close"]) * 100

## Momentum Score
df["Momentum Score"] = (df["Close"] - df["Open"]) / (df["High"] - df["Low"])
```

✓ 0.0s

**Figure 2**  
The calculations.

Before exporting the data to a database, certain columns were renamed to prevent conflicts with SQL reserved keywords and improve readability. The column "Open" was changed to "open\_price", "High" to "high\_price", "Low" to "low\_price", and "Close" to "close\_price". Additionally, the "Range" column was renamed to "range\_value" to avoid conflicts with the SQL RANGE function. After renaming, the column names were printed to verify the modifications.

To ensure the DataFrame was correctly formatted for SQL export, several transformations were applied. Since the date was originally used as an index, it was reset and stored as a regular column named "Date", replacing the generic "index" label. The "Date" column was also converted to the datetime64[ns] format to ensure compatibility with SQL databases. Furthermore, all numerical columns were converted to float and rounded to four decimal places to maintain precision while optimizing storage.

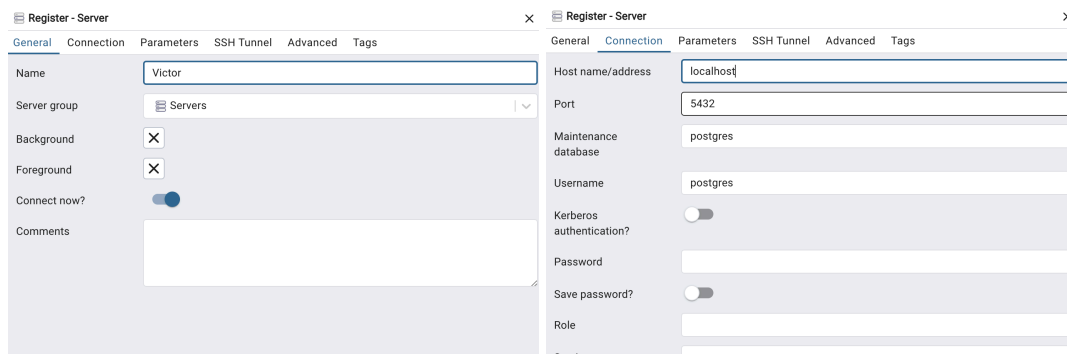
Before exporting the final dataset, a data type verification step was conducted to confirm that all values were properly formatted. This last phase of the transformation process ensured that the DataFrame was fully SQL-compatible, recognizing that SQL has its own structure for storing and handling data efficiently.

### 3. Loading Process

The final step of the ETL process involved exporting the cleaned and processed DataFrame to a CSV file to facilitate further use and integration into a database. The file was saved under the name "API.csv" using a comma as a separator, following the standard CSV format. To ensure compatibility across different systems, the file was encoded in "UTF-8". Additionally, the index was excluded from the export to prevent Pandas from adding an extra column that could interfere with database integration. Once the export was completed, a success message was displayed to confirm that the file had been saved correctly.

Then, I needed to integrate the CSV file into SQL. To achieve this, I installed PostgreSQL, which provides a local server for managing databases, and pgAdmin, which offers a user-friendly interface for executing SQL queries. The integration process involved several steps to ensure that the data was correctly imported into the database. Below is a step-by-step explanation of how I proceeded with the CSV integration into SQL.

#### 1. Created a server.



#### 2. Create the Table in the database

```
15/03/2025 21:52:39      26 msec
Date                    Rows affected  Duration
Copy  Copy to Query Editor
CREATE TABLE alpha_vantage_data (
  id SERIAL PRIMARY KEY,
  date DATE NOT NULL,
  open_price FLOAT NOT NULL,
  high_price FLOAT NOT NULL,
  low_price FLOAT NOT NULL,
  close_price FLOAT NOT NULL,
  monthly_change FLOAT,
  range_percentage FLOAT,
  closing_momentum FLOAT,
  volatility_index FLOAT,
  momentum_score FLOAT
);
```

#### 3. Import the csv files into SQL with the query :

COPY alpha\_vantage\_data (date, open\_price, high\_price, low\_price, close\_price, monthly\_change,

```
range_percentage, closing_momentum, volatility_index, momentum_score)
FROM '/Users/victorbrivet/Documents/GitHub/API/Alpha_Vantage_Forex.csv'
DELIMITER ','
CSV HEADER
```

#### 4. Run a simple query to ensure the proper integration

```
1 SELECT * FROM alpha_vantage_data LIMIT(10);
2
```

Showing rows: 1 to 10   Page No: 1 of 1													
id	date	open_price	high_price	low_price	close_price	monthly_change	range_percentage	closing_momentum	volatility_index	momentum_score			
[PK] integer	date	double precision	double precision	double precision	double precision	double precision	double precision	double precision	double precision	double precision			
1	2006-02-28	1.2154	1.2166	1.1825	1.192	-1.9253	2.8837	0.2786	2.8607	-0.6862			
2	2006-03-31	1.1921	1.2207	1.1857	1.2121	1.6777	2.9518	0.7543	2.8876	0.5714			
3	2006-04-28	1.2116	1.2635	1.2029	1.2613	4.102	5.0378	0.9637	4.8046	0.8201			
4	2006-05-31	1.2628	1.2971	1.2553	1.2812	1.4571	3.3299	0.6196	3.2626	0.4402			
5	2006-06-30	1.2817	1.298	1.2475	1.2789	-0.2185	4.0481	0.6218	3.9487	-0.0554			
6	2006-07-31	1.2786	1.2859	1.2458	1.2766	-0.1564	3.2188	0.7681	3.1412	-0.0499			
7	2006-08-31	1.2763	1.2938	1.2695	1.2806	0.3369	1.9141	0.4568	1.8975	0.177			
8	2006-09-29	1.2805	1.2874	1.2632	1.2674	-1.023	1.9158	0.1736	1.9094	-0.5413			
9	2006-10-31	1.268	1.2782	1.2483	1.2762	0.6467	2.3953	0.9331	2.3429	0.2742			
10	2006-11-30	1.2761	1.3274	1.2682	1.3241	3.7615	4.668	0.9443	4.471	0.8108			

## 4. Conclusion on the Alpha Vantage

The decision to use Alpha Vantage for data extraction was driven by its ability to provide historical Forex exchange rates, which are not readily available through Yahoo Finance. Given the limitations of the free API key, I opted to retrieve monthly EUR/USD exchange rates over a 230-month period to balance data availability and request constraints. The extraction methodology involved constructing a well-defined API request with specific parameters to ensure consistency in data retrieval.

For the transformation phase, I applied data cleaning techniques, including the conversion of JSON data into a structured Pandas DataFrame, renaming columns for SQL compatibility, and performing quality checks for missing values and outliers. These steps were necessary to ensure the reliability and accuracy of the dataset before storing it in PostgreSQL.

A key area for improvement would be automating data updates by implementing a scheduled ETL pipeline, allowing real-time or periodic updates instead of relying on static CSV exports. Additionally, enhancing the outlier detection methodology by incorporating statistical approaches such as the Interquartile Range (IQR) or Z-score analysis could improve the accuracy of data validation. Finally, integrating additional economic indicators from Alpha Vantage, such as interest rates or macroeconomic data, could provide deeper insights into currency movements.

## Yahoo Finance model

### 1. Extraction Process

Unlike Alpha Vantage, Yahoo Finance does not require an API key, making it a more accessible and straightforward option for retrieving historical stock price data. The extraction process was designed to be flexible, allowing the user to define both the list of stock tickers and the date range for analysis. To achieve this, I implemented a function that fetches stock price data using the `yfinance` library. This function takes three parameters: the stock ticker, the start date, and the end date. By calling `yf.download()`, it retrieves historical price data, which is then processed for further analysis.

To enhance usability, I incorporated an automated selection of CAC 40 stock symbols rather than requiring manual input. The date range was also predefined but remained adjustable if needed. A validation check was added to ensure that the start date was always earlier than the end date, preventing potential errors in data retrieval.

```
# Prompt the user to enter the stock tickers to analyze
# tickers = input("Enter the stock tickers separated by spaces (e.g., MC.PA AIR.PA OR.PA SAN.PA BNP.PA): ").split()
tickers = [
    "AC.PA", "AI.PA", "AIR.PA", "MT.AS", "CS.PA", "BNP.PA", "EN.PA", "CAP.PA",
    "CA.PA", "ACA.PA", "BN.PA", "DSY.PA", "EDEN.PA", "ENGI.PA", "EL.PA", "ERF.PA",
    "RMS.PA", "KER.PA", "OR.PA", "LR.PA", "MC.PA", "ML.PA", "ORA.PA", "RI.PA",
    "PUB.PA", "RNO.PA", "SAF.PA", "SGO.PA", "SAN.PA", "SU.PA", "GLE.PA", "STLAP.PA",
    "STMPA.PA", "TEP.PA", "HO.PA", "TTE.PA", "URW.PA", "VIE.PA", "DG.PA", "VIV.PA"
]

# Choose the time period
# start_date = input("Enter the start date (YYYY-MM-DD, e.g., 2010-01-01) : ").strip()
# end_date = input("Enter the end date (YYYY-MM-DD, e.g., 2024-01-01) : ").strip()
start_date = '2010-01-01'
end_date = '2025-01-01'

# Validate the entered dates
if start_date >= end_date:
    raise ValueError("The start date must be earlier than the end date.")
```

✓ 0.0s

Figure 4

This code defines a list of CAC 40 stock tickers and a fixed date range for historical data extraction, while ensuring that the start date is earlier than the end date to prevent errors.

Given that I had previously worked on a similar project for my GitHub portfolio, aimed at preparing for interviews in data and finance, I was able to build upon my existing model. While the core structure remained similar, I updated and optimized certain aspects to improve efficiency and compatibility with Tableau. The extracted dataset was reshaped into a long format, making it more suitable for visualization and analysis. Additionally, column names were dynamically adjusted to prevent errors when integrating the data into PostgreSQL. This approach ensured a scalable and structured data retrieval process, aligning with both analytical and database storage requirements.

```
# Download historical stock price data
stock_prices = get_stock_data(tickers, start_date, end_date)

# Reshape the data for Tableau compatibility
stock_prices = stock_prices.stack(future_stack=True).reset_index()
stock_prices.columns = ["Date", "Ticker"] + list(stock_prices.columns[2:]) # Dynamically rename to avoid errors
```

✓ 1.6s

[\*\*\*\*\*100%\*\*\*\*\*] 40 of 40 completed

Figure 5

This code downloads historical stock price data for the specified tickers and date range, then reshapes the dataset into a long format for better compatibility with Tableau, dynamically renaming columns to prevent errors.

## 2. Transformation Process

The transformation phase follows the same structured approach as in the Alpha Vantage model, ensuring that the dataset is cleaned, enriched, and optimized for analysis and SQL storage. Several key financial indicators are computed to evaluate stock performance, momentum, and volatility.

To assess stock price variations, the Daily Return is calculated using the percentage change between consecutive days, while the Cumulative Return tracks the compounded return over time. To analyze short-term and long-term price trends, 50-day and 200-day Simple Moving Averages (SMA) are computed, allowing for a more refined trend analysis. The Sharpe Ratio, which measures risk-adjusted returns, is also integrated, using an assumed annual risk-free rate of 2%, converted into a daily rate for accuracy. Handling missing values is a crucial step; backfilling (bfill()) is applied where possible, and any remaining NaN values are replaced with 0 to ensure data consistency.

```

# Calculate daily return
stock_prices["Daily Return"] = stock_prices.groupby("Ticker")["Close"].pct_change(fill_method=None)

# Calculate cumulative return
stock_prices["Cumulative Return"] = (1 + stock_prices["Daily Return"]).groupby(stock_prices["Ticker"]).cumprod()

# Compute 50-day and 200-day simple moving averages (SMA)
stock_prices["SMA_50"] = stock_prices.groupby("Ticker")["Close"].rolling(window=50).mean().reset_index(level=0, drop=True)
stock_prices["SMA_200"] = stock_prices.groupby("Ticker")["Close"].rolling(window=200).mean().reset_index(level=0, drop=True)

# Define the annual risk-free rate (e.g., 2%)
risk_free_rate_annual = 0.02 # 2%

# Convert to daily risk-free rate
risk_free_rate_daily = (1 + risk_free_rate_annual) ** (1/252) - 1

# Compute the adjusted Sharpe Ratio
stock_prices["Sharpe Ratio"] = stock_prices.groupby("Ticker")["Daily Return"].transform(
    lambda x: (x.mean() - risk_free_rate_daily) / (x.std() + 1e-8)
)

# Handle missing values
stock_prices.bfill(inplace=True) # Backfill to prevent data loss
stock_prices.fillna(0, inplace=True) # Final safeguard against NaN values

```

✓ 0.1s

**Figure 6**

This code calculates key financial indicators for stock analysis, including daily returns, cumulative returns, simple moving averages (50-day and 200-day), and the Sharpe Ratio, while also handling missing values to ensure data integrity.

To enhance the dataset, sector and industry information for each stock is retrieved using the yfinance API. This data is stored in a dictionary and converted into a Pandas DataFrame before being merged with the stock price dataset. A left join ensures that stock price data remains intact while adding sector and industry details, allowing for sector-based and industry-based analysis of stock performance.

```

# Dictionary to store sector and industry data
sector_industry_dict = {}

for ticker in tickers:
    try:
        stock = yf.Ticker(ticker)
        info = stock.info
        sector = info.get("sector", "Unknown") # Retrieve sector
        industry = info.get("industry", "Unknown") # Retrieve industry
        sector_industry_dict[ticker] = {"Sector": sector, "Industry": industry}
    except Exception as e:
        print(f"Error fetching data for {ticker}: {e}")

# Convert the dictionary into a DataFrame
sector_industry_df = pd.DataFrame.from_dict(sector_industry_dict, orient="index")
sector_industry_df.reset_index(inplace=True)
sector_industry_df.columns = ["Ticker", "Sector", "Industry"]

# Merge with stock prices dataset
stock_prices = stock_prices.merge(sector_industry_df, on="Ticker", how="left")

```

✓ 10.2s

**Figure 7**

This code retrieves sector and industry information for each stock ticker using the Yahoo Finance API, stores the data in a dictionary, converts it into a DataFrame, and merges it with the stock prices dataset for sector-based analysis.

Before exporting the dataset to PostgreSQL, several formatting steps are applied. The "Date" column is converted into datetime64 format for SQL compatibility, and all numerical values are stored as float, rounded to four decimal places. Missing values are replaced with NULL, ensuring proper storage in the database. Column names are renamed to be SQL-friendly, replacing spaces with underscores and standardizing lowercase formatting to

prevent case sensitivity issues. A final verification step is conducted to confirm that all data types are correctly formatted before insertion into PostgreSQL.

```
# Convert the Date column to datetime format (if it exists)
if "Date" in stock_prices.columns:
    stock_prices["Date"] = pd.to_datetime(stock_prices["Date"])

# Convert all numerical columns to float and round to 4 decimal places
numerical_cols = stock_prices.select_dtypes(include=['number']).columns
stock_prices[numerical_cols] = stock_prices[numerical_cols].astype(float).round(4)

# Replace NaN values with np.nan (which will be converted to NULL in SQL)
stock_prices.fillna(value=np.nan, inplace=True)

# Rename columns to be SQL-friendly (no spaces, lowercase)
stock_prices.columns = stock_prices.columns.str.replace(" ", "_").str.lower()

# Verify after cleaning
print("\n Data is ready for SQL:")
print(stock_prices.dtypes)
```

✓ 0.0s

```
Data is ready for SQL:
date                datetime64[ns]
ticker              object
close              float64
high              float64
low               float64
open              float64
volume            float64
daily_return       float64
cumulative_return  float64
sma_50            float64
sma_200           float64
sharpe_ratio       float64
sector             object
industry           object
dtype: object
```

**Figure 8**

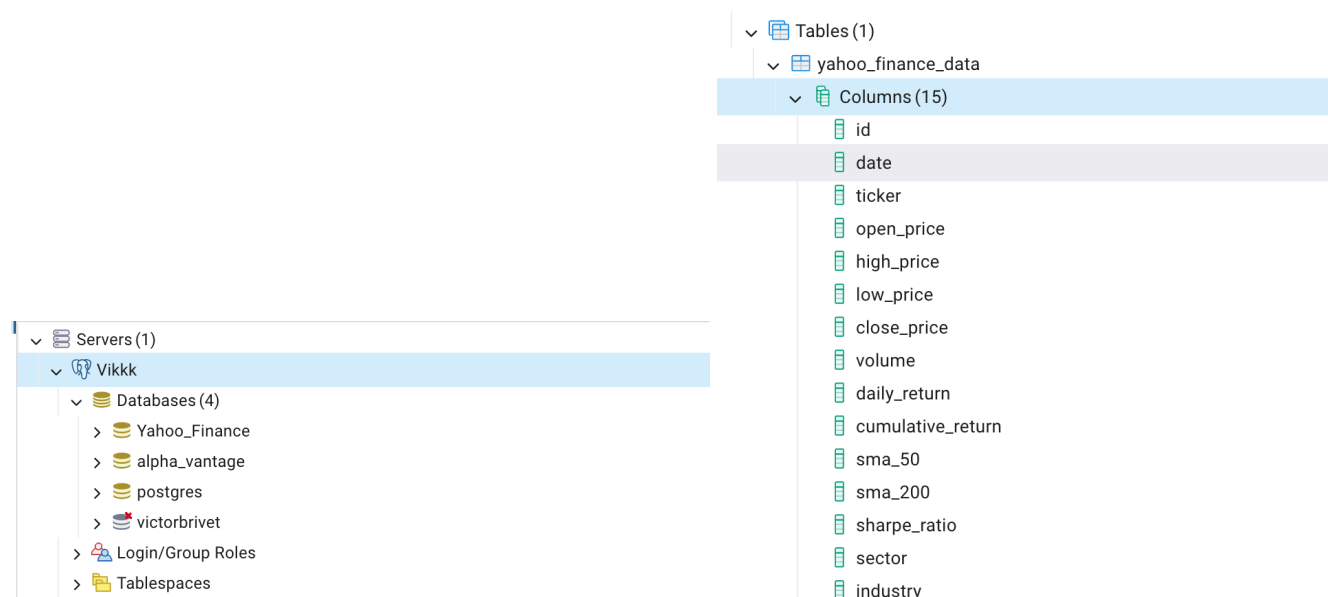
This code prepares the dataset for SQL integration by converting the date column to datetime64, ensuring numerical columns are in float format rounded to four decimal places, replacing missing values with NULL, renaming columns for SQL compatibility, and verifying data types before insertion.

By following the same structured methodology as in the Alpha Vantage model, this approach ensures that the Yahoo Finance dataset is clean, structured, and optimized for both financial analysis and SQL integration.

### 3. Loading Process

The final step of the ETL process involves exporting the cleaned and processed dataset to a CSV file before integrating it into PostgreSQL. This step follows the same approach as in the Alpha Vantage model, ensuring consistency in data handling and database integration.

The dataset is first saved as a CSV file to facilitate database insertion. The export is performed using the UTF-8 encoding to maintain compatibility across different systems, and a comma separator is used to ensure a structured format. Additionally, the DataFrame index is excluded from the export to prevent unwanted columns from being added during SQL integration. Once the CSV file is created, the next step is to load it into PostgreSQL via pgAdmin 4, following the same procedure as in the Alpha Vantage model.



**Figure 9**  
Final database in PostgreSQL

This standardized approach ensures that the data is properly formatted, structured, and efficiently stored in the SQL database, allowing for seamless querying and further analysis.

## Conclusion

This project successfully implemented an ETL pipeline using Python, focusing on extracting financial data from Alpha Vantage and Yahoo Finance, transforming it for analysis, and loading it into a PostgreSQL database. By following a structured approach, the data was cleaned, formatted, and enriched with key financial indicators before being stored for further use.

One of the main challenges encountered was ensuring data quality, as both data sources contained missing values and inconsistencies. To address this, we applied data validation techniques, such as handling missing values, detecting outliers, and computing financial metrics like daily returns, moving averages, and the Sharpe ratio. The final integration into PostgreSQL required formatting adjustments, such as renaming columns and ensuring proper data types for seamless SQL storage and querying.

While the pipeline effectively processes and stores financial data, several improvements could make it more efficient. Using SQLAlchemy instead of CSV exports would allow direct interaction between Python and PostgreSQL, improving performance and flexibility. Additionally, automating the ETL process with a scheduled script would enable regular updates without manual execution. Another possible enhancement would be adding a data validation step to systematically check for anomalies before loading the data into the database.

Overall, this project demonstrated the importance of a well-structured ETL workflow for financial data analysis. By combining Python's processing capabilities with SQL's storage efficiency, the pipeline provides a solid foundation for further expansion and optimization.