STOCK INSIGHTS PREP

📊 Stock Insights Dashboard: A Data-Driven Analysis Tool

---

1. Project Architecture: The Data Pipeline

Our system is structured into five cohesive components, ensuring robust data integrity and fast performance.

Component	Files	Role
Data Source & Setup	schema.sql, db_config.py, insert_companies.py	Defines the database schema, establishes connections using **Psycopg2**, and seeds the **companies** table with the initial list of stock tickers.
Data Acquisition (ETL)	data_fetcher.py	Fetches historical data from external APIs (yfinance is the primary source). It includes logic to check the database for the last update date (MAX(trade_date)), ensuring only **new data is fetched and inserted**.
Data Storage	PostgreSQL Database	Acts as the persistent **single source of truth**, storing company metadata and all historical price data in the **stock_prices** table.
Business Logic	analysis.py	Contains all the core mathematical and statistical functions (e.g., SMA, Volatility, Correlation). It queries the database using **Pandas' read_sql** and processes the data into actionable insights.
Presentation Layer	app.py (Streamlit)	The interactive web dashboard that handles user input, orchestrates the analysis calls, and visualizes the results.

---

2. Database Core: PostgreSQL Integrity

Our database design is optimized for financial time-series data, prioritizing speed and integrity.

Data Insertion Mechanism

The system uses the most efficient way to manage time-series data updates:

1. **Unique Constraint:** The stock_prices table uses a **Compound Unique Key** on (company_id, trade_date).
2. **Upsert Logic:** The data_fetcher.py script uses the PostgreSQL command:

SQL

INSERT ... ON CONFLICT (company_id, trade_date) DO UPDATE SET ...

- This command ensures that if a date's price data already exists, the row is **updated** instead of causing an error, guaranteeing **data integrity** and preventing duplicates.

Performance and Reliability

- **Indexing:** An index is created on (company_id, trade_date). This allows for **rapid retrieval** of price data for any specific ticker and date range, speeding up all charts and analyses.
  - *Example:* When fetching data for a chart, the database uses this index to find the data instantly, avoiding a slow full table scan.
- **Foreign Key (ON DELETE CASCADE):** The stock_prices table links to companies using ON DELETE CASCADE. If a company is removed, all its historical prices are automatically deleted, preventing **orphaned data**.
  - *Example:* If "HDFC Bank" is deleted from the companies table, all its thousands of price records are deleted automatically.

---

3. Dashboard Functionality: Actionable Insights

The Streamlit dashboard (app.py) manages user input and calls functions based on the active tab.

Tab 1: 📊 Price & Trends

This tab provides foundational **Technical Analysis**.

- **Data:** Historical Close Price (from DB).
- **Analysis:** Computes and displays the **Simple Moving Average (SMA)** and **Exponential Moving Average (EMA)**. SMA tracks long-term trends, while EMA is more responsive to recent price changes.
- **Feature:** Displays the latest price using st.metric.

Tab 2: 🔍 Abrupt Changes

This tab focuses on filtering significant daily movements.

- **User Input:** A slider controls the **Threshold (%)**.
- **Analysis:** Calculates the daily **percentage change (pct_change)** and filters the data to show only days where the movement exceeded the user-defined threshold.

Tab 3: 🤖 ML & Volatility

This tab quantifies **Risk**.

- **User Input:** A slider defines the rolling **Window (days)** (e.g., 20 days).
- **Key Output:** It calculates and plots **Volatility** (measured as the rolling Standard Deviation) and a simple **Risk** metric. This helps users assess the instability and potential risk of a stock over time.

Tab 4: 🧮 Compare & Correlate

This tab is crucial for **Portfolio Management**.

- **Process:** The compare_companies() function fetches and merges price data for all selected stocks, ensuring data alignment.
- **Key Output: Correlation Matrix:** The correlation_analysis() function calculates the pairwise correlation coefficients between the stocks using Pandas' .corr() method.
    - This helps investors assess **diversification**: a low or negative correlation suggests the stocks do not move together, which generally lowers overall portfolio risk.

Tab 5: ⚙️ Export & Info

This tab handles utility and data accessibility.

- **Key Feature:** Uses st.download_button() to allow the user to instantly generate and download the stock's full historical data as a **CSV file** using the Pandas df.to_csv() method.

---

4. Performance & Logging

- **Caching:** The @st.cache_data decorator ensures the data fetching process is not repeated unnecessarily, keeping the app fast and efficient.
- **Logging:** The data_fetcher.py script uses the Python logging module to record warnings (e.g., API failures) and errors, providing a clear audit trail and aiding in debugging.

This project provides a comprehensive, structured, and performant platform for data-driven stock analysis.

1.Schema.Sql :

Command	Explanation	Example/Detail
DROP TABLE IF EXISTS stock_prices CASCADE;	This command safely attempts to **delete the stock_prices table** if it exists, preventing an error if the script is run multiple times. The **CASCADE** clause ensures that any objects that depend on stock_prices (though none are defined here) would also be deleted.	If another table, portfolio_holdings, had a foreign key referencing stock_prices, CASCADE would drop portfolio_holdings as well.
-- DROP COMPANIES	This is a **SQL comment** and has no effect on the database. It is used for clarity, indicating the next command's intent.	*No applicable example.*
DROP TABLE IF EXISTS companies CASCADE;	This command safely attempts to **delete the companies table** if it exists. The **CASCADE** clause is critical here: because the stock_prices table references companies via a foreign key, running this command deletes	If the companies table contained 'RELIANCE.NS' and you dropped the table with CASCADE, the database automatically removes all

Command	Explanation	Example/Detail
	the companies table **and automatically deletes the dependent stock_prices table** as well.	'RELIANCE.NS' price data from stock_prices to maintain integrity.

The importance of the **ON DELETE CASCADE** and the CASCADE keyword used in your schema is to ensure **referential integrity** and **automatic cleanup** when a company is deleted.

Here is a clear explanation using your tables as an example:

Importance of ON DELETE CASCADE

The ON DELETE CASCADE rule is defined in your stock_prices table definition:

SQL:

company_id INTEGER NOT NULL REFERENCES companies(company_id) ON DELETE CASCADE

1. Referential Integrity (Why it's needed)

- The stock_prices table has a foreign key (company_id) that links every single price record to its parent company in the companies table.
- The system *must* ensure that every price record points to an existing company. If a company were deleted, its price records would become **orphaned data** (records pointing to a non-existent parent), which violates data integrity.

2. Automatic Cleanup (How it works with your data)

ON DELETE CASCADE tells the database: "If a row in the parent table (companies) is deleted, automatically delete all corresponding child rows in the stock_prices table."

Action (Deletion on companies table)	Result (Automatic Deletion on stock_prices table)
Example Action: You decide to delete **HDFC Bank** (company_id = 2) from the companies table.	**Result:** The database automatically scans the stock_prices table and immediately deletes every single row where company_id = 2.
Benefit: This prevents hundreds or thousands of HDFC Bank's historical price records from becoming junk data, saving you from writing extra code to perform the manual cleanup.	

Importance of CASCADE in the DROP TABLE Command

The schema uses CASCADE in the initial cleanup commands:

SQL:

DROP TABLE IF EXISTS companies CASCADE;

This is used for **script execution safety and cleanup during schema setup**.

1. Dependency Resolution

- When you run DROP TABLE IF EXISTS companies, the database checks if any other tables rely on it. The stock_prices table *does* rely on it because of the foreign key constraint.
- Without CASCADE, the DROP TABLE companies command would **fail** because stock_prices depends on it.

2. Full Schema Reset

- The **CASCADE** keyword forces the deletion of both the companies table **and** any objects that depend on it, which includes the stock_prices table.
- **Result:** This single command ensures the complete removal of both tables, allowing the rest of your script to successfully recreate a fresh, empty schema.

Companies-Table:

Command	**Explanation**
CREATE TABLE companies (	Starts the definition of the companies table.
company_id SERIAL PRIMARY KEY,	Defines the company_id column as a unique, auto-incrementing integer that serves as the **primary key** for this table.
company_name VARCHAR(100) NOT NULL ,	Defines the company name as a string up to 100 characters, which **cannot be left empty** (NOT NULL).
ticker_symbol VARCHAR(200) UNIQUE NOT NULL	Defines the stock ticker as a string up to 200 characters. It must be present (NOT NULL) and **must be unique** across all rows in the table.

stock_prices Table :

Command	**Explanation**
CREATE TABLE stock_prices (	Starts the definition of the stock_prices table.
id SERIAL PRIMARY KEY,	Defines the id column as the unique, auto-incrementing **primary key** for this table.
company_id INTEGER NOT NULL REFERENCES	Defines the **Foreign Key**. It links this price record to a company ID in the companies table. NOT NULL ensures a price record must be tied to a company. The **ON DELETE CASCADE** rule means if a

Command	Explanation
companies(company_id) ON DELETE CASCADE,	company is deleted from the companies table, all its corresponding price records in stock_prices are deleted too.
trade_date DATE NOT NULL,	Defines the trading date, which **must be provided**.
open_price NUMERIC(12, 4),	Defines the opening price as a number with a total of 12 digits, with 4 digits after the decimal point.
high_price NUMERIC(12, 4),	Defines the high price using the same numeric precision.
low_price NUMERIC(12, 4),	Defines the low price using the same numeric precision.
close_price NUMERIC(12, 4),	Defines the closing price using the same numeric precision.
volume BIGINT,	Defines the trading volume as a large integer.
UNIQUE(company_id, trade_date)	Defines a **compound unique key**. This constraint ensures that the same company cannot have more than one price entry for the exact same date, preventing duplicate data.

The Index Creation command :

CREATE INDEX idx_stockprices_company_date ON stock_prices(company_id, trade_date);,

builds a separate, sorted lookup table that acts like a book index 📚 for the stock_prices data. Its purpose is purely to boost performance. Without it, the database would have to search every row to find a price, but with the index, it can rapidly locate data by first identifying the company_id and then jumping directly to the requested trade_date, ensuring queries are fast even with millions of records.

Example in Website Query Context: When a user opens the Price & Trends tab and selects Infosys (company_id is, for instance, 7) and a date range of January 1 to March 31, 2024, the database executes a query like SELECT * FROM stock_prices WHERE company_id = 7 AND trade_date BETWEEN '2024-01-01' AND '2024-03-31'. The index uses the combination of company_id=7 and the date range to fetch only the 60 or so relevant rows almost instantly, preventing a slow full table scan and allowing the chart to load quickly.

2.db_config.py:

Psycopg2 is used because it is the **most popular and reliable PostgreSQL database adapter for the Python programming language**.

1. **Store Connection Parameters:** It securely holds all necessary credentials (host, port, database name, user, and password) required to access the PostgreSQL database.

2. **Centralize Connection Logic:** It defines the single function, create_connection(), which wraps the psycopg2.connect() call, making it the standardized way for all other scripts to establish a database session.
3. **Ensure Modularity:** It isolates database-specific settings and logic from the application's core functions (like data fetching and analysis), ensuring a clean, modular code structure.

3.data_fetcher.py:

data_fetcher.py is the project's data pipeline responsible for fetching raw stock data from external APIs (Alpha Vantage/yfinance) and then **inserting or updating that data into the PostgreSQL database**.

1. Setup and Imports

Code Block	**Purpose & Detail**
from db_config import create_connection	Imports the function needed to establish a connection to the PostgreSQL database. **This is the critical link to the database system.**
import yfinance as yf, import pandas as pd, etc.	Imports external libraries for API access (yfinance), data manipulation (pandas), and utility functions.
logging.basicConfig(...)	Sets up logging to track the fetching process, writing status messages to both the console and a file (data_fetch_log.txt).
ALPHA_VANTAGE_API_KEY = 'H13FNA09HVUDWWKU' INEFFICIENT!	Stores the API key needed for the Alpha Vantage service.

2. API Fetching Functions (No Direct DB Interaction)

These functions retrieve the raw data into a Pandas DataFrame format.

Code Block	**Purpose & Detail**
def fetch_alpha_vantage(ticker):	Attempts to fetch full daily historical data using the Alpha Vantage API. (The comment #NOT-WORKING suggests this is the less-reliable source).
def fetch_yfinance(ticker):	Fetches daily historical data for the ticker using the yfinance library, setting the period to "max" to get all available data. **This is the primary data source.**
def safe_scalar(val):	A utility function to ensure values extracted from the DataFrame (which might sometimes be Pandas Series objects) are converted into simple scalar types for database insertion.

3. Database Query Functions (Retrieval/Read Operations)

These functions interact with the database to get necessary metadata for the fetching process.

Code Block	Purpose & Detail	SQL Query Executed
def get_company_list():	**Retrieves the master list of all tickers** that need updating from the companies table.	***SELECT ticker_symbol FROM companies ORDER BY company_name;***
def get_company_id(conn, ticker_symbol):	**Gets the internal Foreign Key** (company_id) for a given ticker. This ID is mandatory for linking price data to the correct company in the stock_prices table.	***SELECT company_id FROM companies WHERE ticker_symbol = %s;***
def get_latest_date(conn, company_id):	**Determines the update starting point**. It queries the stock_prices table to find the most recent trade_date already stored. **This is a crucial optimization step.**	***SELECT MAX(trade_date) FROM stock_prices WHERE company_id = %s;***

4. insert_prices() Function (Write/Update Operation)

This function manages the data cleaning, filtering, and final insertion into the stock_prices table.

Code Block	Purpose & Detail	SQL Query Executed
df = df[df.index >= '2015-01-01']	Filters the downloaded data	*(None - Pandas operation)*
company_id = get_company_id(...)	Retrieves the primary key ID. (If not found, it logs a warning and exits).	*(Calls SQL from get_company_id)*
latest_date = get_latest_date(...)	Retrieves the last date saved for this company.	*(Calls SQL from get_latest_date)*
if latest_date: df = df[df.index > pd.to_datetime(latest_date)]	**Filters out old data.** If a latest date exists, it truncates the downloaded DataFrame to contain only newer records, minimizing the amount of data to be processed and inserted.	*(None - Pandas operation)*
for idx, row in df.iterrows():	Loops through each remaining new row (trade date) in the DataFrame.	*(See next block)*
cur.execute(...)	**Executes the actual database write operation**. This robust statement performs either an INSERT or an UPDATE:	INSERT INTO stock_prices (company_id, trade_date, open_price, high_price, low_price, close_price, volume) VALUES (%s, %s, %s, %s, %s, %s, %s) ON CONFLICT (company_id, trade_date) DO UPDATE SET

Code Block	Purpose & Detail	SQL Query Executed
		open_price=EXCLUDED.open_price, high_price=EXCLUDED.high_price, low_price=EXCLUDED.low_price, close_price=EXCLUDED.close_price, volume=EXCLUDED.volume;
conn.commit() and conn.close()	**Finalizes the transaction.** Makes all the insertions permanent in the database and then closes the connection, releasing resources.	*(None - Transaction management)*

5. run_fetching() Function (Orchestration)

The main function, run_fetching(), executes the entire data pipeline, and for each company it processes, it prints and accumulates the **number of new or updated rows inserted** into the stock_prices table.

Here is how the row count is returned and tracked:

1. **insert_prices(data, ticker) Return Value:** The insert_prices function is designed to return the inserted_count, which is incremented every time the cur.execute() command successfully runs inside the database transaction loop. This count represents the number of days of price data that were either newly added or updated for that specific ticker.
2. **run_fetching() Accumulation:** Inside the main loop of run_fetching(), the returned value is stored in the inserted variable.
    - If inserted > 0, the script prints the number of new rows and adds that count to the total_new_rows variable.
    - If inserted is 0, it means the company's data was already up to date, and the script prints "-> up to date".
3. **Final Summary:** When the script finishes, it prints a summary including the grand total of new_rows inserted across all companies.

The usage of the logging module in the data_fetcher.py file is to **record events and status messages** during the execution of the data fetching script.

This serves several vital purposes in a background or data-processing script:

1. **Debugging and Error Tracking:** It records specific errors (logging.error) that occur during the API calls or database insertions, such as a "YFinance error" or an "Error inserting row". This helps developers diagnose exactly when and why the script failed.
2. **Monitoring and Auditing:** It tracks key events (logging.info), such as the start and end times of the fetch process and a summary of how many new rows were successfully inserted for each company. This provides an audit trail of the data updates.

3. **Warning Flagging:** It captures potential issues (logging.warning) that don't stop the script but are worth noting, such as a "Company not found for ticker" or a failure to get data from the primary Alpha Vantage API.
4. The script is configured to use two handlers: one sends messages to the console (StreamHandler), and the other writes messages to a file (data_fetch_log.txt), ensuring persistent record-keeping.

4.insert_companies.py:

Script Breakdown

1. Imports and Function Definition

1. from db_config import create_connection: Imports the function necessary to establish a connection to the PostgreSQL database.
2. def insert_companies(companies):: Defines the main function that takes a list of company tuples (name, ticker) and handles the database insertion.

2. Database Connection and Execution

1. conn = create_connection(): Establishes the connection to the database.
2. with conn.cursor() as cur:: Creates a cursor object, which is used to execute SQL commands. The with statement ensures the cursor is closed automatically.
3. for name, ticker in companies:: Loops through the provided list of companies (e.g., ("Reliance Industries", "RELIANCE.NS")).
4. cur.execute(...): Executes the SQL insertion command for the current company.

3. The SQL Logic (ON CONFLICT)

The critical part of the script is the SQL statement:

SQL

"INSERT INTO companies (company_name, ticker_symbol) VALUES (%s, %s) ON CONFLICT (ticker_symbol) DO NOTHING;"

1. **INSERT INTO companies ... VALUES (%s, %s)**: This is the standard command to insert the company name and ticker into the companies table.
2. **ON CONFLICT (ticker_symbol) DO NOTHING**: This is a PostgreSQL feature that manages conflicts. Since the ticker_symbol column in the companies table is defined as **UNIQUE** (in schema.sql), attempting to insert the same ticker twice would normally cause an error. This clause tells the database: **"If you try to insert a ticker that already exists, simply skip the operation and do nothing."**

You are inserting only the company name and ticker symbol, but the **company_id (the Primary Key) is inserted automatically by the PostgreSQL database itself.**

Company_id is SERIAL

5.analysis.py:

1. Data Fetching and Utility Functions

Code/Function	Purpose	Database Interaction/Pandas Usage	Example/Detail
Imports	Loads external libraries necessary for DB access, data analysis, and plotting.	Imports matplotlib.pyplot (plt), pandas (pd), and the database connection function create_connection.	*(None)*
fetch_prices(...)	Primary Data Retriever. Retrieves all historical price (OHLCV) data for a single stock ticker within an optional date range.	SQL Execution: Uses create_connection() and pd.read_sql() to execute a SELECT query that joins stock_prices and companies tables. Pandas Prep: Converts the 'trade_date' column to the proper datetime format and sorts the data.	SQL Example: SELECT ... FROM stock_prices sp JOIN companies c ON ... WHERE c.ticker_symbol = 'TCS.NS' AND sp.trade_date BETWEEN '2023-01-01' AND '2023-12-31'.
fetch_current_price(...)	Retrieves the most recent closing price and trade date for a ticker.	Executes an SQL query that orders data by trade_date DESC and uses LIMIT 1. Returns the single result row as a dictionary.	SQL Logic: Retrieves the latest price without having to load the entire history.
fetch_company_info(...)	Retrieves the full company name based on the ticker symbol.	Executes a simple SQL query on the companies table. Returns the company name and ticker as a dictionary.	SQL Logic: SELECT company_name, ticker_symbol FROM companies WHERE ticker_symbol = %s.

---

2. Analytics and Computation Functions

Code/Function	Purpose	Pandas Usage/Logic	Example/Detail
compute_sma(df, window=20)	Calculates the Simple Moving Average (SMA).	Uses the Pandas .rolling(window=20, min_periods=1).mean() method on the close_price column. min_periods=1 ensures that the SMA column	For a 20-day SMA, the value for any given day is the average of the previous 20 closing prices.

		has values starting from the first day of the data.	
compute_ema(df, window=20)	Calculates the Exponential Moving Average (EMA).	Uses the Pandas .ewm(span=20, adjust=False, min_periods=1).mean() method. The EMA is a weighted average that gives more importance to recent closing prices.	span=20 determines the smoothing factor used in the EMA calculation.
detect_abrupt_changes(...)	Identifies days where the price change (up or down) exceeds a set threshold.	First calculates the daily percentage change using .pct_change() on the close_price column. Then, it filters the DataFrame using boolean indexing: abs(df['pct_change']) > threshold.	If the threshold is set to 0.05 (5%), only days where the price moved more than 5% will be returned.
volatility_and_risk(...)	Calculates rolling price volatility and a risk metric.	Volatility: Calculates the rolling standard deviation (a measure of price dispersion) using .rolling(...).std(). Risk: Calculated by dividing the volatility by the close price: df['volatility'] / df['close_price'].	High volatility (large standard deviation) indicates greater price uncertainty.

3. Comparative and Visualization Functions

Code/Function	Purpose	Pandas/Matplotlib Usage	**Example/Detail**
compare_companies(...)	Fetches and merges the closing prices of multiple stocks for comparison.	Iteration & Renaming: Loops through all selected tickers, calls fetch_prices() for each, and renames the close_price column to the company name. Merging: Uses pd.concat(..., axis=1, join='inner') to combine the individual price series.	**The resulting DataFrame has trade_date as the index and separate columns like 'HDFC Bank' and 'ICICI Bank' for direct plotting.**

		join='inner' ensures the resulting DataFrame only contains dates common to all selected stocks.	
correlation_analysis(...)	Calculates the pairwise correlation matrix for the selected stocks.	Relies on compare_companies logic to create the merged DataFrame. Calculates the matrix using the Pandas .corr() method on the price data.	**The output is a symmetrical matrix where a value of means the stocks move together.**
plot_correlation(...)	Generates a heatmap visualization of the correlation matrix.	Uses matplotlib.pyplot (plt) functions like plt.imshow() to draw the matrix visually. Sets vmin=-1 and vmax=1 to standardize the color scale for correlation coefficients.	***(Visuals)***
export_data(...)	Writes any DataFrame to a specified CSV file.	Uses the Pandas .to_csv(filename, index=False) method. index=False prevents the trade_date column (which is often the index) from being included as a redundant column in the CSV.	***(Export utility)***

WORKFLOW EXAMPLE :

This example traces the entire data flow for two selected stocks, **Infosys (INFY.NS)** and **Wipro (WIPRO.NS)**, from the database to the final display and export on the Streamlit dashboard.

Scenario: Analyzing Infosys and Wipro

Step 1: Initial Data Retrieval (Metadata and Current Price)

When the user selects **Infosys** and **Wipro**, the application performs initial database queries to fetch basic information and the latest price.

Data Fetching Task	Python Function (analysis.py)	Database Query (SQL Executed)	Result Used For
1. Company Info	fetch_company_info('INFY.NS')	SELECT company_name, ticker_symbol FROM companies WHERE ticker_symbol = 'INFY.NS'	Displaying **"Infosys (INFY.NS)"** as the chart title/heading.
2. Current Price	fetch_current_price('INFY.NS')	SELECT sp.close_price, sp.trade_date FROM stock_prices sp JOIN companies c ON ... WHERE c.ticker_symbol = 'INFY.NS' ORDER BY sp.trade_date DESC LIMIT 1	Populating the **"Current Price" metric** in the Price & Trends tab.

Step 2: Tab 1: Price & Trends (Graph and Tabular Data)

The core historical data is fetched, processed, and displayed as a line graph and a table.

Data Processing Task	Python Function (analysis.py & app.py)	SQL/Pandas Logic	Website Output
1. Historical Data	fetch_prices('INFY.NS', ...)	**SQL:** Retrieves all trade_date, open_price, close_price, etc., for INFY.NS. **Pandas:** Loads the SQL result into a DataFrame.	**Source Data** for the entire tab.
2. Technical Indicators	df = compute_sma(df) and df = compute_ema(df)	**Pandas Logic:** Applies .rolling(window=20).mean() for SMA and .ewm(span=20).mean() for EMA on the close_price column.	**Line Graph:** Displays three lines: Close Price, SMA, and EMA. **Tabular Data:** The full DataFrame, including the newly calculated SMA and EMA columns, is shown in the expandable table (st.dataframe).

Step 3: Tab 4: Compare & Correlate (Comparison Graph and Matrix)

This requires fetching data for both stocks simultaneously and performing statistical analysis.

Data Processing Task	Python Function (analysis.py)	SQL/Pandas Logic	Website Output
1. Comparison Data	merged = compare_companies(['INFY.NS', 'WIPRO.NS'], ...)	**SQL:** Executes two separate fetch_prices queries (one for INFY, one for WIPRO). **Pandas:** Uses pd.concat(..., join='inner') to merge the two price series, ensuring only dates present for **both** stocks are included.	**Comparison Graph:** A single line chart showing the close_price trend of Infosys vs. Wipro side-by-side.
2. Correlation Matrix	corr = correlation_analysis(['INFY.NS', 'WIPRO.NS'])	**Pandas Logic:** Runs the merged.corr() method on the comparison DataFrame to calculate the relationship coefficients.	**Correlation Matrix:** A 2×2 table showing the correlation coefficient (e.g., 0.85) between Infosys and Wipro. **Correlation Plot:** A colored heatmap visualization of the matrix.

Step 4: Tab 5: Export & Info (CSV File)

The user retrieves the processed data as a downloadable file.

Data Processing Task	Python Function (app.py)	Pandas Logic	Website Output
1. Data for Export	df = fetch_prices('INFY.NS', ...)	Executes the same historical SQL query as in Step 2.	Provides the raw, current historical data DataFrame.
2. CSV File Generation	st.download_button(data=df.to_csv(index=False), ...)	Pandas Logic: The .to_csv(index=False) method converts the DataFrame into a CSV string format. index=False omits the date index from the CSV file.	A downloadable INFY.NS_price_data.csv file is created for the user.

6.app.py

1. Setup, Imports, and Configuration

Code Block	Purpose & Detail	Example
Imports	Brings in all the tools needed: streamlit for the UI, functions from analysis.py (like fetch_prices, compute_sma, correlation_analysis), and functions from data_fetcher.py (like get_company_list) [cite: app.py].	If the app needs SMA, it calls compute_sma() from analysis.py.
st.set_page_config(...)	Sets up the initial browser tab, giving the dashboard a title and setting the layout to wide for better screen usage [cite: app.py].	Allows the charts to take up more horizontal space on the page.
@st.cache_data(...)	Caching for Performance. This decorator tells Streamlit to save the result of update_stock_data() for 24 hours (ttl=24*60*60). If the user refreshes the page within that time, the heavy run_fetching() function is not called again [cite: app.py].	Avoids re-downloading and re-inserting all stock data every time the app loads, making the app feel fast.
Theme Selection	Uses a sidebar st.selectbox to allow the user to toggle between Light and Dark visual themes.	Applies CSS styles to change the background color (background-color: #181a1b for Dark) and text color [cite: app.py].

2. Sidebar and Input Controls

Code Block	Purpose & Detail	Example
Title & Initial Load	Displays the main title and fetches the list of available stock tickers from the database using get_company_list() [cite: app.py].	Displays " Stocks Insights" [cite: app.py].
Company Selector	Creates the main sidebar control (st.sidebar.multiselect) for the user to choose one or more stocks they want to analyze. The list of options comes directly from the database [cite: app.py].	User selects "Reliance Industries" and "Infosys" from the list.
Date Range Inputs	Creates sidebar controls (st.sidebar.date_input) for the user to optionally filter the data by a start and end date [cite: app.py].	User enters 2024-01-01 as the start date [cite: app.py].
Date Validation	Checks if the user-selected start_date is later than the end_date. If so, it displays an error message [cite: app.py].	If Start date (Aug 1) > End date (July 1), it shows an error: "Start date must be *less* than end date" [cite: app.py].

3. Tab Structure

Code Block	Purpose & Detail	Example
Tab Definition	Creates the main navigation tabs that organize the analysis tools [cite: app.py].	The five tabs are: "Price & Trends," "Abrupt Changes," "ML & Volatility," "Compare & Correlate," and "Export & Info" [cite: app.py].

4. Tab Content Logic (Detailed Breakdown)

Tab	Code Logic	Function Calls & Purpose
Tab 1: Price & Trends	Loops through each selected company [cite: app.py]. Fetches the full price data (fetch_prices) and calculates technical indicators (compute_sma, compute_ema) [cite: app.py].	fetch_prices(): Gets data from DB. fetch_current_price(): Gets the current value for the metric display. st.line_chart(): Plots Close Price, SMA, and EMA [cite: app.py].
Tab 2: Abrupt Changes	Creates a st.slider for the user to set the percentage threshold (e.g., 5%) [cite: app.py]. Calls detect_abrupt_changes(df, threshold) to filter the data [cite: app.py].	detect_abrupt_changes(): Filters the data to show only days where the price change exceeded the slider value [cite: app.py]. st.dataframe(): Shows the resulting table of change events [cite: app.py].
Tab 3: ML & Volatility	Creates a slider (window) to define the period for analysis (e.g., 20 days) [cite: app.py]. Calculates Volatility and Risk using that window [cite: app.py].	volatility_and_risk(): Calculates the rolling standard deviation (volatility) and the risk metric [cite: app.py]. st.line_chart(): Visualizes how the stock's risk has changed over time [cite: app.py].
Tab 4: Compare & Correlate	Only runs if more than one ticker is selected (if len(selected_tickers) > 1) [cite: app.py].	compare_companies(): Fetches and merges data from the DB for all selected stocks into one chart (st.line_chart) [cite: app.py]. correlation_analysis(): Calculates the correlation matrix [cite: app.py]. st.dataframe() & plot_correlation(): Displays the matrix both as a table and as a visual heatmap [cite: app.py].
Tab 5: Export & Info	Provides utility features [cite: app.py].	fetch_company_info(): Displays structured company metadata (st.json). fetch_prices() & st.download_button(): Fetches the latest data and converts it to a CSV file on the fly for the user to download [cite: app.py].

5. Styling (Watermark)

Code Block	Purpose & Detail
Developer Credit CSS	This large block of st.markdown contains custom HTML and CSS to create an elaborate, styled developer credit or "watermark" [cite: app.py].
CSS & Keyframes	It uses custom CSS for gradients, rounded corners, shadows, and animations (@keyframes bounce, slide-in) to make the credit box visually appealing and slightly dynamic [cite: app.py].
Sidebar Footer	Adds a simple message to the sidebar footer [cite: app.py].