ChatGPT

AI-Powered Stock Analysis and Prediction Platform

Project: Multi-Modal Approach to Stock Analysis
Course: Bachelor of Technology in Computer Science and Engineering (AI & ML) – Capstone Project Phase III (UE22CS320A)
Date: October 2025

Acknowledgements

We would like to thank our project advisor and all faculty members for their guidance and support throughout this project. We also appreciate the contributions of our peers and family for their encouragement and feedback. This work was developed as part of the Capstone project at [Your Institution Name], and it benefited greatly from discussions with fellow students and mentors.

Abstract

This report presents an **AI-powered stock market analysis and prediction platform**. The platform integrates traditional financial analysis with advanced machine learning and natural language processing techniques to assist investors. Key features include fetching real-time and historical stock data, computing technical indicators, performing news sentiment analysis (using TextBlob and GPT-4o), and predicting future prices with multiple models (Random Forest, ARIMA, BiLSTM, CNN). It also supports portfolio optimization and discovery of related stocks. A user-friendly Streamlit dashboard and a Flask REST API enable interaction with the system. This report covers the system architecture, design details, modules, usage instructions, sample results, and future work. The stock market is “very complex and volatile” with many hidden trends [1]. By combining technical analysis with news sentiment, our platform aims to provide comprehensive insights for better investment decisions.

Table of Contents

List of Figures and Tables

1. Introduction / Project Overview

Stock market analysis is challenging because the market is "very complex and volatile" and influenced by many factors [1]. This project delivers a comprehensive, AI-powered platform to analyze stock data and help users make investment decisions. The system fetches real-time and historical stock prices, computes a variety of technical indicators, analyzes market sentiment from news articles, and applies machine learning and deep learning models to predict future stock prices. Key features include:

- **Real-time & Historical Data:** Connects to Yahoo Finance (via `yfinance`) to download up-to-date price, volume, and fundamentals for selected stocks.
- **Technical Analysis:** Automatically computes dozens of indicators (Moving Averages, RSI, MACD, Bollinger Bands, etc.) on the price data for trend and momentum signals.
- **Sentiment Analysis:** Retrieves news related to a stock (from Financial Modeling Prep API or Yahoo Finance) and computes sentiment scores. It uses **TextBlob** for quick polarity and **OpenAI GPT-4o** for more nuanced analysis. The platform can also generate AI-powered summaries of news articles. Such sentiment signals serve as a "voice of the market" and are known to correlate with price movements [4].
- **Price Prediction Models:** Offers multiple models for forecasting stock prices:
- **Random Forest Regression:** A feature-based ML model using engineered indicators.
- **ARIMA:** A classical time-series forecasting model (often combined with GARCH) for baseline predictions [5].
- **BiLSTM & CNN (Deep Learning):** Sequential neural networks (bidirectional LSTM and 1D CNN) to capture complex temporal patterns.
- **Portfolio Optimization:** Tools to compute optimal allocations in a portfolio for a given risk level, based on mean-variance optimization principles.
- **Related Stocks Discovery:** Graph-based analysis to suggest stocks that are related by sector/industry or price correlation, helping users find similar investment opportunities.
- **Interactive Interface:** A web dashboard built with **Streamlit** for selecting stocks, visualizing data, and viewing model outputs.
- **REST API:** A Flask-based API (`api.py`) exposing the core functionality programmatically.

By combining these components, the platform provides a one-stop solution for stock data analysis. The workflow begins with user input (a stock ticker), then data is collected and processed, models make predictions, and results are visualized for the user. Throughout, the system emphasizes explainability (e.g., indicator charts and sentiment explanations) and performance. In the following sections, we detail the design, implementation, and usage of this platform.

2. Literature Review (Related Work)

Machine learning and sentiment analysis for stock prediction have been widely explored in recent research. For example, Gite *et al.* (2021) highlight that stock prices are influenced by technical fundamentals and sentiment in media releases [1] [4]. Classical time-series methods like ARIMA and GARCH have been used to model trends and volatility [5], but deep learning models (LSTM, CNN) have shown promising improvements due to their ability to capture nonlinear dependencies. Sentiment analysis of financial news is known to correlate with price movements [4], and newer systems leverage large language models (LLMs) to better interpret textual data. Our platform builds on these insights by integrating multiple approaches: it uses both traditional models and deep learning, and incorporates sentiment analysis powered by GPT-4o, aiming to improve prediction accuracy over single-method systems.

3. System Architecture / High-Level Design (HLD)

The high-level architecture of the platform consists of several interconnected components: the **user interface**, **data sources**, **processing modules**, and **output services**.

Figure 1: Context-level Data Flow Diagram for a securities trading system, showing external entities and the main process [2]. In this illustrative diagram, external entities (e.g., *Customers*, *Brokers*) interact with the central trading platform process. Similarly, our stock analysis platform interacts with users and external data sources.

- **Web Front-End (Streamlit):** The user interacts via a Streamlit app (`run_streamlit_app.py`), selecting stocks, setting parameters, and viewing results. Streamlit handles UI rendering of charts, tables, and controls in the browser.
- **REST API (Flask):** A Flask-based API (`api.py`) allows programmatic access to the functionality (e.g., fetching data, triggering predictions). It defines endpoints for stock data retrieval, sentiment scoring, and predictions.
- **Data Collection Layer:** The `stock_data.py` module acts as the data ingestion layer. It fetches stock prices, company info, and economic indicators from Yahoo Finance (`yfinance` library). It also retrieves market data (e.g., indices, VIX) to gauge overall market sentiment.
- **News Analysis Module:** The `news_analysis.py` fetches news articles for the chosen stock (via Financial Modeling Prep API or Yahoo Finance RSS). It then processes these texts for sentiment using TextBlob and GPT-4o. The integration of GPT-4o provides rich analysis (e.g., sentiment polarity and summaries) beyond simple lexical methods.
- **Modeling Engine:** The `model.py` module implements various prediction models. Based on the data and user choice, it selects the appropriate algorithm (Random Forest, ARIMA, BiLSTM, or CNN) to forecast future prices. It also handles feature engineering and data normalization for the models.
- **Portfolio & Graph Modules:** For advanced features, `portfolio_optimization.py` computes efficient frontiers and optimal allocations (using historical return data). The

`generate_related_stocks_graph.py` analyzes a graph of stocks (using industry and price correlations) to find peers or alternatives to a given stock.
- **Visualization:** The `visualization.py` module contains plotting functions (built on Plotly and Matplotlib) to generate candlestick charts, line charts, scatter plots, etc. These functions are used by both the Streamlit front-end and for saving image outputs (e.g., if using the API).

Overall, the architecture follows a **modular pipeline**: user request → data retrieval → analysis (technical + sentiment) → prediction → visualization. The system can be deployed on a single server or scaled (for example, using Docker containers or microservices) by running the Streamlit app on one service, the Flask API on another, etc. We did not implement a real-time streaming engine, but the design is compatible with such extensions (e.g., incorporating Kafka or Spark Streaming as suggested by literature on stock exchange system design).

4. Low-Level Design (LLD)

At the code level, the project is organized into Python modules, each encapsulating specific functionality. Key classes/functions include:

- `StockDataCollector` **(in** `stock_data.py`**):** Functions such as `get_stock_data()` fetch historical prices and compute technical indicators (over 20 indicators like SMA, EMA, RSI, MACD). Others (`get_company_info()`, `get_historical_earnings()`, etc.) retrieve fundamental data (PE ratio, revenue, industry, etc.). This module abstracts the Yahoo Finance API and ensures data is cleaned (e.g., handling missing values).
- `NewsAnalyzer` **(in** `news_analysis.py`**):** Functions `fetch_news()` and `analyze_sentiment()` work together to gather news snippets and assign sentiment scores. If an OpenAI key is present, `gpt_sentiment_score()` uses the GPT-4o model; otherwise, `textblob_score()` provides a quick estimate. The module returns both the raw articles and summary insights.
- **Model Classes (in** `model.py`**):** There are classes or functions for each model type. For example, `train_volatility_model()` (Random Forest), `predict_with_arima()`, `BiLSTMModel`, and `CNNModel`. Each class follows a standard interface: fit on training data and predict future values. The method `predict_prices()` chooses the model based on data size or user choice. Feature engineering (lagged returns, moving averages, etc.) is handled in `create_features()`.
- `PortfolioOptimizer`**:** Implements Markowitz optimization. It takes a list of tickers and historical data, then uses convex optimization (via libraries like `cvxpy` or custom math) to output the optimal weight for each asset, given a target return or risk.
- `RelatedStocksGraph`**:** Uses network analysis (e.g., NetworkX) on the stock-industry dataset (`*.csv`) and cached JSON graphs. It computes similarity scores (based on industry, sector, and return correlations) to suggest related stocks.
- `Visualizer`**:** Contains functions like `plot_candlestick()`, `plot_price_with_indicators()`, and `plot_prediction()`. These generate static or interactive figures. The Streamlit app calls these functions to embed graphs.
- **Utilities (**`utils.py`**):** Helper functions for tasks like data caching, formatting dates, logging, and common preprocessing steps.

The low-level design ensures separation of concerns: data I/O, analysis, modeling, and interface code are in different modules. For example, the Streamlit app (`run_streamlit_app.py`) mostly orchestrates user input and calls functions from the modules above, without embedding complex logic. Similarly, the Flask `api.py` defines routes that call these modules. The code is structured to allow unit testing (`test_*.py` files are present for key components).

5. Workflow / Use Case / Sequence

The typical **user workflow** is as follows:

1. **User Input:** The user enters a stock ticker symbol (e.g., AAPL) and selects options (date range, prediction horizon, model choice) on the Streamlit dashboard.
2. **Data Fetching:** The app calls `stock_data.get_stock_data()`, which uses `yfinance` to download historical price and volume data for the selected range. It computes technical indicators on this data.
3. **News Fetching:** Simultaneously, `news_analysis.fetch_news()` is called to get recent news articles about the company. These are passed to `analyze_sentiment()` to compute a sentiment score for each article and an overall sentiment indicator. If GPT is enabled, it also generates a text summary.
4. **Data Processing:** The raw data (prices, indicators, sentiment features) are assembled into a feature set. `create_features()` may compute additional lagged variables or normalized values.
5. **Model Prediction:** Based on user choice or data amount, `model.predict_prices()` selects one of the model methods:
6. If using **Random Forest**, it trains `RandomForestRegressor` on past data (features vs. next-day price) and predicts future prices.
7. If **ARIMA** is chosen, `predict_with_arima()` fits an ARIMA time-series model to the closing price history and forecasts the next values. (Per literature [5], ARIMA often models long-term trend well, and further combining ARIMA with GARCH can improve forecasts.)
8. For **BiLSTM/CNN**, the data is reshaped into sequences, the neural network is trained (or a pretrained model is loaded), and the network outputs the price predictions.
9. **Visualization:** The platform then prepares output charts:
10. A candlestick chart of historical prices.
11. Overlays of chosen technical indicators.
12. A line chart showing the model's predicted prices vs. actual past prices.
13. A bar or line chart of sentiment scores over time. These charts are generated by functions in `visualization.py`.
14. **Output Display:** The Streamlit UI displays the charts, along with key numeric results: predicted price values, next-day volatility estimate, sentiment summary, etc. If the user requested portfolio or related-stock analysis:
15. **Portfolio:** The user inputs multiple tickers. The app calls `portfolio_optimization.optimize_portfolio()`, which returns optimal weights. The UI shows these weights and an efficient frontier graph.
16. **Related Stocks:** Given one ticker, `generate_related_stocks_graph.find_related()` returns a list of similar tickers. These might be shown in a table or network graph.

Each of these steps can be considered a use-case sequence. For example, the sequence of calling the stock data, sentiment analysis, and model modules constitutes the core "predict stock price" use case. Similarly, "find portfolio allocation" and "suggest related stocks" have their own flows. The interactions between modules are linear and stateless (each request recomputes or fetches fresh data), which simplifies the sequence.

6. Project Structure

The codebase is organized into modules as shown below. Table 1 summarizes the key modules and their purposes:

Module / File	Description
`run_streamlit_app.py`	Launches the Streamlit web application (main user interface).
`api.py`	Implements the Flask REST API; defines endpoints for data retrieval, sentiment analysis, and predictions.
`stock_data.py`	Fetches stock market data via `yfinance` (historical prices, financials) and computes technical indicators.
`news_analysis.py`	Retrieves news articles (Financial Modeling Prep / Yahoo Finance) and performs sentiment analysis (TextBlob and GPT-4o summaries).
`model.py`	Contains functions and classes for price prediction models (Random Forest, ARIMA, BiLSTM, CNN) and feature engineering (`create_features`).
`portfolio_optimization.py`	Implements portfolio optimization algorithms (e.g., mean-variance optimization for asset allocation).
`generate_related_stocks_graph.py`	Analyzes stock relationships (using industry/sector data and historical correlations) to suggest related stocks.
`visualization.py`	Functions for generating plots and charts (candlesticks, line charts, scatter plots) using Matplotlib and Plotly.
`utils.py`	Utility functions (data processing helpers, caching, formatting) used by various modules.
`stock_industry_sector_dataset.csv`	Dataset of stocks with their industry and sector information (used by related-stocks analysis).
`requirements.txt`	List of Python dependencies for the core project.
`requirements_advanced.txt`, `requirements_torch.txt`	Additional requirements for advanced features (e.g., deep learning, PyTorch).

Table 1: Overview of key project modules and files.

The repository also contains example datasets (`*.csv`), JSON files for caching related-stock graphs (`related_stocks_*.json`), and test scripts (e.g., `test_installation.py`, `test_related_stocks.py`). The main entry points are `run_streamlit_app.py` (for the dashboard) and `main.py` (for alternative CLI or testing usage). All modules are designed to be reusable and testable.

7. Module Descriptions

Data Collection (`stock_data.py`)

This module is responsible for gathering and pre-processing all stock market data. It uses the `yfinance` library to interface with Yahoo Finance. Key functions include:

- `get_stock_data(symbol, start, end)`: Downloads historical OHLCV (open-high-low-close-volume) data for `symbol` between the given dates. It also **calculates over 20 technical indicators** (moving averages, RSI, MACD, Bollinger Bands, etc.) from this price series. These indicators are added as new columns to the data frame.
- `get_company_info(symbol)`: Retrieves fundamental company information (industry, sector, business summary) and key ratios (PE ratio, EPS, dividend yield, etc.) from Yahoo Finance.
- `get_historical_earnings(symbol)`, `get_quarterly_earnings(symbol)`: Fetches past earnings reports (annual or quarterly) to assess financial performance.
- `get_market_sentiment()`: Optionally computes a broad market sentiment index by analyzing indices (e.g., S&P 500) and volatility index (VIX) trends.

This module ensures the application always has the latest data. It also centralizes feature engineering (indicator calculations) so other modules (like the models) can rely on these precomputed features. Any missing data (e.g., holidays or blank days) are forward-filled or handled gracefully.

Sentiment Analysis (`news_analysis.py`)

This module brings in the qualitative “voice” of financial news. It works in two stages:

- **News Retrieval:** The function `fetch_news(symbol)` uses either the Financial Modeling Prep (FMP) API or Yahoo Finance news to gather recent articles related to `symbol`. It returns a list of article titles and snippets.
- **Sentiment Scoring:** For each news snippet, the module computes a sentiment score. By default, it uses **TextBlob**, which gives a polarity score (–1 for negative, +1 for positive). If an OpenAI API key is provided, the platform also queries **GPT-4o** to analyze the text. GPT-4o can capture nuance in language, sarcasm, and context better than a simple lexical method. The result is a numerical sentiment (and sometimes a short text summary) for each article.

The output of this module is a time series of sentiment scores (and key phrases) associated with the stock. Studies have shown that news sentiment often correlates with and can **lead** stock price changes [4]. Our platform displays a plot of news sentiment over time alongside price charts. This helps users see if the market mood (based on news) is optimistic or pessimistic. The integration of GPT-4o is a notable enhancement: it allows the system to understand financial language (e.g., "beats expectations" vs. "misses target") with greater accuracy.

Machine Learning / Deep Learning Models (`model.py`)

This module implements the predictive algorithms—the “brain” of the platform. It supports multiple model types:

- **Feature Engineering:** The function `create_features(data)` takes the raw price data (with indicators) and prepares it for modeling. It adds lagged returns, volatility measures, and any other engineered features needed (e.g., a 5-day moving average of RSI).
- **Random Forest (train_volatility_model):** This is a fast, tree-based regression model. It trains on historical feature data to predict future volatility or price. Random Forests are interpretable: we can extract feature importance to see which indicators matter most.
- **ARIMA (predict_with_arima):** A classical time-series model. We fit an ARIMA model to the closing price series to forecast the next value(s). ARIMA is a good baseline; for example, recent research shows that combining an ARIMA(2,1,0) model with a GARCH(1,1) volatility model improves accuracy [5]. In our system, ARIMA is especially useful when data is limited or when a quick forecast is needed.
- **BiLSTM (`BiLSTMModel` class):** A bidirectional long short-term memory network. This deep learning model can capture long-range temporal patterns in sequential data. We reshape the price series into sequences (e.g., look-back windows of 60 days) and train the BiLSTM to predict the next day’s price. BiLSTMs learn both forward and backward dependencies, which can capture more context.
- **CNN (`CNNModel` class):** A 1D convolutional neural network applied to the time series. CNNs are good at extracting local patterns (like repeated shapes or cycles) in the data. We apply convolutional filters across the price/indicator sequence to detect such patterns for prediction.
- **Dynamic Model Selection (predict_prices):** Based on user choice and data size, the platform automatically picks the model. For example, if the user selects “Deep Learning” and enough data is present, it runs the BiLSTM/CNN. If “Time Series” is chosen, it runs ARIMA. The function returns a uniform output (predicted price series) regardless of model type.

All models can be trained on demand or loaded from saved weights (for speed). The module also evaluates performance metrics (e.g., RMSE, MAE) if historical future data is available for validation.

Visualization (`visualization.py`)

To make the results interpretable, this module provides plotting utilities:

- **Candlestick Charts:** Using Matplotlib or Plotly, it draws interactive candlestick charts of price data with volume. The user can hover over points to see exact values.
- **Indicator Overlays:** Functions can overlay moving averages, Bollinger Bands, or RSI lines on the price chart, so users can visually assess trends and momentum.
- **Prediction Plots:** When a model makes a prediction, the module plots the historical price line and the predicted future line (often as a continuation of the same chart). This directly shows how the forecast extends the known data.
- **Sentiment Charts:** A bar or line chart showing news sentiment (e.g., 7-day moving average of positive news) alongside price.

- **Portfolio Charts:** If optimizing a portfolio, it can plot the efficient frontier (risk vs. return) and mark the selected point.
- **Network Graphs:** For related stocks, it may use a network visualization (nodes = stocks, edges = similarity) to illustrate the graph structure.

These visualizations ensure the output is not just numbers but intuitive charts. (For example, Figure 2 below is an example plot showing historical prices of multiple tech stocks.) Many of these functions are used by Streamlit with minimal modification, taking advantage of Streamlit's support for Plotly and Matplotlib figures.

Portfolio Optimization (`portfolio_optimization.py`)

This component implements classical portfolio theory. Given a list of stock tickers and historical returns, it computes the **optimal asset allocation** for a specified risk tolerance. Internally, it may use the following steps:

1. Compute expected returns and covariance matrix of the assets.
2. Solve a convex optimization problem to maximize return for a given variance (or vice versa). This yields the weights of each stock in the portfolio.
3. Optionally perform Monte Carlo simulations of random portfolios to visualize the efficient frontier.

The output is a set of weights (percent allocations) that maximize return at the chosen risk level. This allows users to enter, for example, 5 favorite stocks and get an "efficient" mix of those stocks. The Streamlit dashboard then displays these weights in a table and plots the efficient frontier curve. This helps users diversify their holdings mathematically.

Related Stocks Discovery (`generate_related_stocks_graph.py`)

This module helps users find **alternative investment opportunities**. It constructs a graph where nodes are stocks and edges represent similarity (e.g., same industry or high return correlation). The related CSV datasets (`indian_stocks_industry_sector.csv`, etc.) provide industry/sector grouping. The process is:

- Build a graph where stocks in the same industry/sector are connected, possibly weighted by their historical price correlation.
- Given a target stock, find its neighbors in this graph (either direct connections or using graph algorithms like PageRank).
- Return a list of similar stocks (e.g., if you pick TCS.NS, it might suggest Infosys.NS, Wipro.NS, etc.).

The related stocks feature is useful for portfolio diversification (instead of Apple, try Microsoft) or for sector analysis. The output is typically shown as a table of tickers with similarity scores. An example related-stock graph file (`related_stocks_AAPL.json`) is precomputed for quick response. This module can be extended with Graph Neural Networks as hinted by the presence of `graph_neural_networks.py`.

8. How to Run the Project

To set up and run the platform, follow these steps:

```
# Clone the repository
git clone <repository-url>
cd <repository-directory>

# Create and activate a Python virtual environment
python3 -m venv venv
source venv/bin/activate    # On Windows: venv\Scripts\activate

# Install required packages
pip install -r requirements.txt
# For advanced features (deep learning, GPT-4o), also install:
pip install -r requirements_advanced.txt
pip install -r requirements_torch.txt
```

Next, **set up API keys** (optional but recommended for full functionality):

- **Financial Modeling Prep API key:** (for premium news data)
- **OpenAI API key:** (for GPT-4o sentiment analysis)

Export these as environment variables or put them in a `.env` file:

```
export FMP_API_KEY="your_fmp_api_key"
export OPENAI_API_KEY="your_openai_api_key"
```

Finally, launch the Streamlit application:

```
streamlit run run_streamlit_app.py
```

After Streamlit starts (by default on `http://localhost:8501`), open your web browser to that URL. You will see the dashboard interface. From there, you can search for a stock ticker, view its data, and use the prediction and optimization tools.

Alternatively, you can run the Flask API server if you need programmatic access:

```
python api.py
```

This will start a local HTTP server (default port 5000) with endpoints like `/get_stock_data`, `/predict`, etc., which can be called from other applications or scripts.

No special hardware is required, but for training deep models (BiLSTM/CNN) having a GPU and installing PyTorch (as in `requirements_torch.txt`) will speed up the process. Otherwise, the system can run on any standard PC.

9. Results / Screenshots

Here we describe sample outputs of the platform. Figure 2 below shows an example chart of historical stock prices generated by the system (using Matplotlib). In the Streamlit dashboard, a similar multi-line chart is displayed interactively.

Figure 2: Example chart of historical stock prices for several technology companies (1990–2022) [3] .

In the actual application, selecting a stock (e.g., "AAPL") would display a **candlestick chart** of its recent price, with moving averages overlaid. The sentiment analysis panel would list recent news headlines along with a positivity/negativity score from TextBlob and GPT. For example, the system might note that sentiment turned negative prior to a price drop, reflecting findings from prior studies [6] [4] .

Below the charts, the **prediction panel** would show the next 5 days of forecasted prices (with confidence intervals) produced by the chosen model. If the Random Forest is selected, it will be labeled accordingly; if ARIMA is used, the ARIMA parameters will be noted. For instance, one run might predict that Apple stock will rise 2% over the next week, which the dashboard would display as a line chart extension beyond the current data.

The **Portfolio Optimization** page allows input of multiple tickers. For example, entering `AAPL, MSFT, GOOGL` and clicking "Optimize" would compute something like: 50% Apple, 30% Microsoft, 20% Alphabet for maximal return at a given risk. The output includes a pie chart of these weights and an efficient frontier curve.

The **Related Stocks** page lets a user enter a ticker (e.g., `RELIANCE.NS`) and see a list of similar stocks (e.g., `TCS.NS`, `INFY.NS`) ranked by similarity score, based on the underlying industry/correlation graph.

(No actual screenshots are included here, but the descriptions illustrate the expected outputs. In practice, each chart and table is clearly labeled with titles and axes. Users can hover or click to get more details in the interactive interface.)

10. Conclusion and Future Work

This project has developed a versatile stock analysis and prediction platform that integrates data collection, technical analysis, sentiment analysis, and multiple machine learning models. By providing both a graphical interface and an API, it serves various user needs—from retail investors to researchers. Key strengths include the multimodal approach (combining numerical and textual data) and the flexibility of model choice. The system can give users a broad market overview (indices sentiment), detailed stock insights (charts and indicators), and predictive analytics (price forecasts).

Potential future enhancements include:

- **Refactoring Feature Engineering:** Currently, technical indicators are computed in both `stock_data.py` and `model.py`. These could be centralized into a single feature-engineering module to avoid redundancy and ensure consistency.

- **Expanded Data Sources:** Integrate additional information such as social media sentiment (Twitter, Reddit) or alternative data (economic indicators, satellite data for commodity companies) to enrich analysis.
- **Advanced Models:** Explore Graph Neural Networks (already hinted by `graph_neural_networks.py`) for modeling stock relationships, or Reinforcement Learning (see `reinforcement_learning.py`) for developing automated trading strategies.
- **User Personalization:** Add user accounts so individuals can save portfolios, watchlists, and analysis settings. Persist historical user queries for personalized recommendations.
- **Alerts System:** Implement real-time alerts (via email or push notifications) for events like large price swings, threshold breaches, or significant sentiment changes.
- **Performance Optimizations:** Use techniques like caching (memoization) to speed up repeated queries, and consider containerizing the app with Docker for easier deployment.
- **UX Improvements:** Enhance the dashboard with more interactivity (e.g., time-range sliders, dynamic filtering) and explanatory tooltips for technical terms to improve usability for beginners.

Overall, this platform demonstrates how combining multiple AI techniques can yield a powerful tool for stock market analysis. As financial markets evolve, the system can be continuously improved by incorporating new models, data, and user feedback.

References / Bibliography

- Visual Paradigm. (2015). *Data Flow Diagram with Examples - Securities Trading Platform*. Retrieved from visual-paradigm.com [2].
- Hunter, J. D. et al. (2020). *Stock prices over 32 years* (Matplotlib example). Retrieved from the Matplotlib gallery [3].
- Gite, S., Khatavkar, H., Kotecha, K., Pandey, N., et al. (2021). *Explainable stock prices prediction from financial news articles using sentiment analysis*. PeerJ Computer Science. (See excerpts [1] [4] [5].)
- *Note:* Citations [24] correspond to insights from Gite et al. (2021) relevant to market complexity, the role of news sentiment, and ARIMA modeling.

---

[1] [4] [5] System Architecture of Stock Market Prediction using LSTM and XAI Shows... | Download Scientific Diagram
https://www.researchgate.net/figure/System-Architecture-of-Stock-Market-Prediction-using-LSTM-and-XAI-Shows-how-the-data-is_fig2_348847477

[2] Data Flow Diagram with Examples - Securities Trading Platform
https://www.visual-paradigm.com/tutorials/data-flow-diagram-example-securities-trading-platform.jsp

[3] Stock prices over 32 years — Matplotlib 3.10.7 documentation
https://matplotlib.org/stable/gallery/showcase/stock_prices.html

[6] Stock News Sentiment vs Stock Price - StockSnips
https://stocksnips.ai/learn/stock-news-sentiment-vs-stock-price/