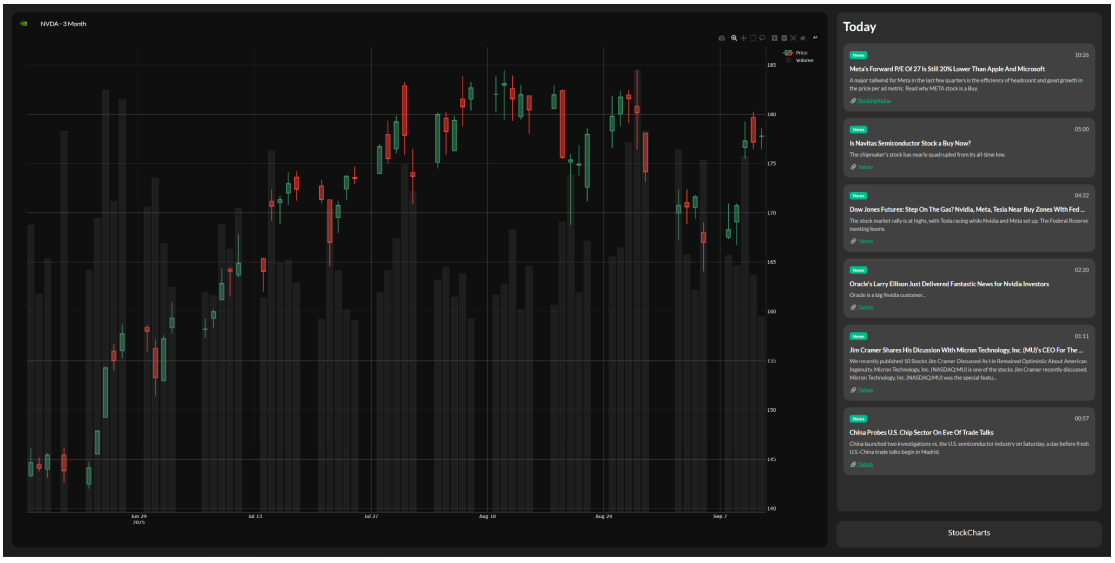
StockChart

Python Project



Ryan Lim Ding Xuan

Contents

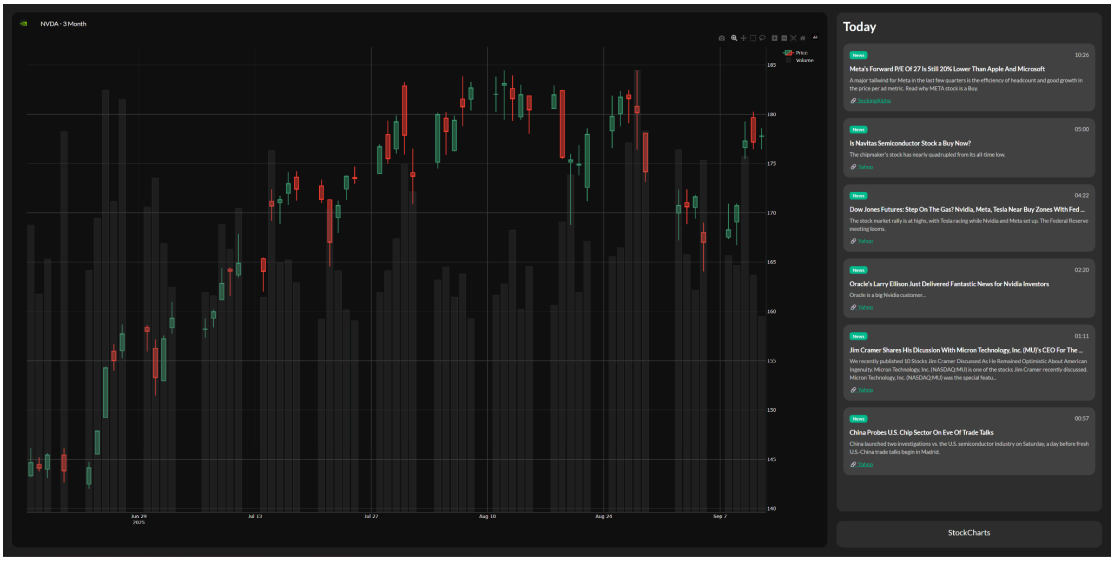
---

1. Project Overview

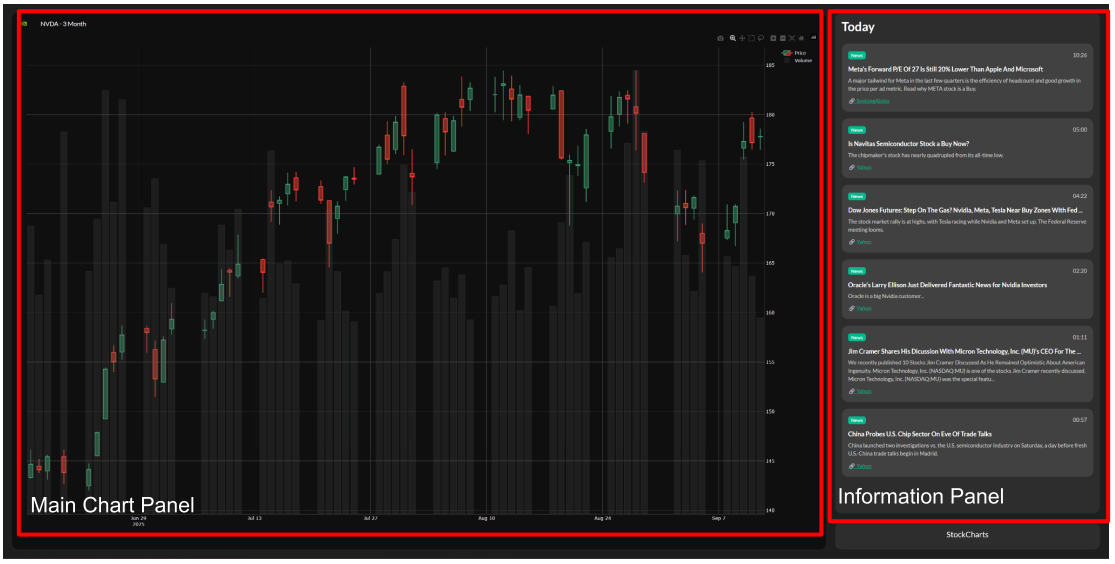
The primary objective of StockChart is to extract the last three months of daily price data for a given ticker (e.g., $NVDA) and correlate price movements with firm-specific information like news, SEC filings, and insider transactions. This provides users with the context needed to understand *why* a price change may have occurred on a specific day (and subsequent days).

2. Application Demonstration

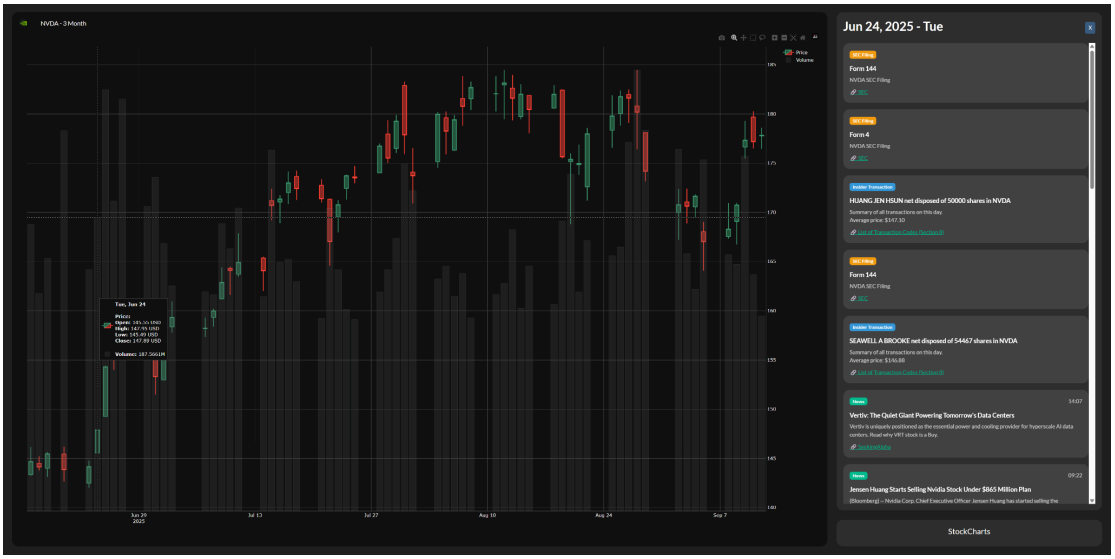
The application presents a clean, two-panel user interface:



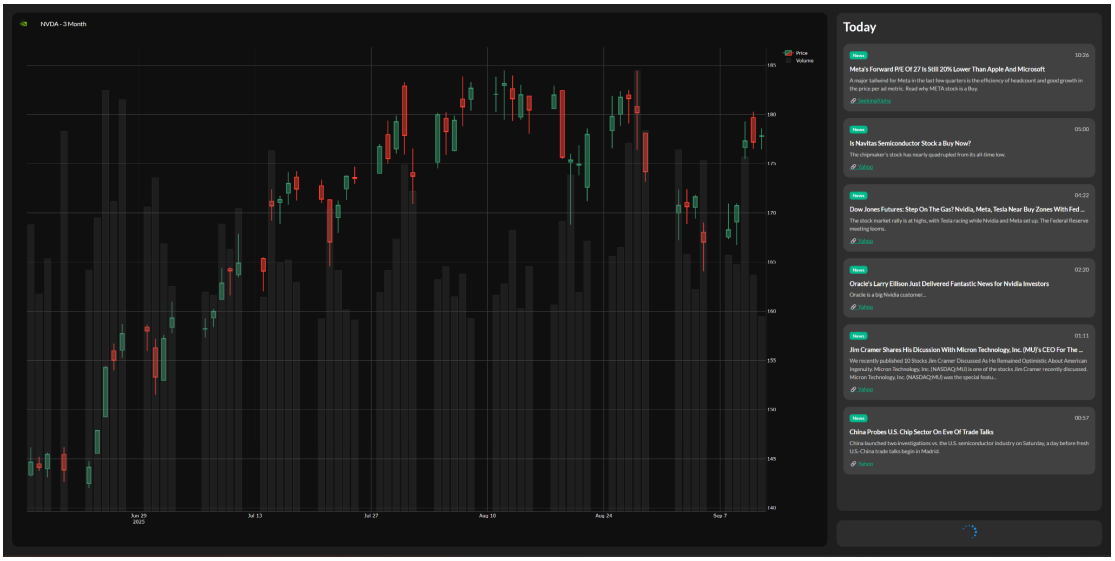
The default view



- **Main Chart Panel:** An interactive Plotly candlestick chart displays the stock's price (Open, High, Low, Close - OHLCV) and trading volume over the last 90 days. Users can zoom, pan, and hover over the chart to inspect data.
- **Information Panel:** This panel dynamically displays a timeline of key events. Clicking any date on the main chart updates this panel to show the top (20) relevant news and corporate events for that specific day. By default, it shows events for the current day.



- **Interactivity:**
    - **Hover** over the chart for a tooltip with OHLCV data.
    - **Click** on a candlestick to bring up the relevant news on that day.
    - **Click** on the close button of the information panel to show Today's events.
    - **Click** on the link in each card to access the source.



- **Updating the data:**
    - **Click** on the button with the app name (StockChart) to refresh the data.
      (Note: refreshing the browser would not re-fetch the data)

3. System Architecture & Design Choices

Several decisions were made with a focus on interactivity and data handling (fetching and processing).

3.1. Frontend Framework: Why Dash?

While alternatives like PySimpleGUI (with `pywebview`) were considered, Dash was selected due to:

1. **Native Web Integration:** Dash creates interactive, browser-based data applications. This avoids the complexity of embedding a webview component within a desktop GUI framework and managing the communication bridge between Python and JavaScript.
2. **Rich Components:** `dash-bootstrap-components` provides a wide array of pre-built, customisable UI components (like Cards, Rows, and Loading spinners) that accelerate development and ensure a polished look.
3. **Callback-Driven Interactivity:** Dash's callback system is a powerful and pythonic way to define the application's interactive logic. Linking chart clicks directly to updates in the information panel is a core feature of the framework.

   Another reason is that it is free to use, making it good to experiment with!

3.2. Charting Library: Why Plotly?

Due to its interactivity. The requirement for capturing click events are native features of Plotly, requiring no extra libraries or complex workarounds, and it has other features such as zooming and panning. Matplotlib, while excellent for static plots, would have required significant additional effort to achieve a comparable level of interactivity in a web context.

3.3. API & Data Source Strategy

A multi-API approach was chosen to gather a comprehensive set of macro and firm-specific information, while minimising dependence on one API for news events.

- **yFinance:** Used for fetching historical daily price and volume data. It is reliable and provides clean, structured OHLCV data.
- **Finnhub:** The primary source for event-based data, including company news, SEC filings, and insider transactions. Its generous rate limit (30 calls/second) makes it suitable for fetching multiple data types concurrently.
- **Alpha Vantage:** Serves as the source for macroeconomic news. Its rate limit is much stricter (25 requests/day), but its `ticker_sentiment` data is invaluable for the event scoring algorithm.

  This strict limit was a primary driver for implementing a robust caching system.

4. Data Processing Pipeline & Optimisations

Significant effort was made for an efficient data pipeline, to ensure the application is relevant, fast, scalable, and memory-efficient.

4.1. Data Selection

To provide intuition behind share price movements.

- **Scope**: A 90-day historical window was chosen as it provides sufficient medium-term context for trend analysis without overwhelming the user with excessive data.
- **Data types** (to cover different spheres of influence on a stock's price):
    - Macro News: Provides context on broader market and economic trends that can affect all stocks (e.g., interest rate changes, inflation data).
    - Company News: Includes press releases and specific news articles that offer direct insight into the company's operations, products, and performance.
    - SEC Filings: Official documents that provide factual, unfiltered information about a company's financial health and significant corporate events.
    - Insider Transactions: Offer a powerful signal of sentiment from a company's own executives and major shareholders. While an insider might sell shares for many reasons unrelated to company performance, an open-market purchase is typically made for one reason: they believe the stock price is going to rise. Therefore, it is a strong indicator of internal confidence in the company's future prospects

4.2. Data Fetching: Concurrency with `ThreadPoolExecutor` & Batching

Concurrency: To avoid a slow user experience where the app waits for each API call to complete sequentially, a concurrent fetching model was implemented in `data_fetcher.py`.

- **Mechanism:** `concurrent.futures.ThreadPoolExecutor` is used to dispatch all API requests (for price, news, filings, etc.) into separate threads.
- **Rationale:** Network I/O operations (like waiting for an API response) release Python's Global Interpreter Lock (GIL). This makes a thread-based approach ideal, as multiple API calls can be in-flight simultaneously, reducing the total data-loading time compared to a sequential approach. The UI gets all the data it needs in the time it takes the *slowest single API call* to complete, not the sum of all of them. Timing this can be done with decorators.

Batching: Finnhub endpoints restrict a single API call to a specific date range or a maximum number of returned items (250 documents, for example). To fetch the full 90 days of history, a single call is insufficient.

- **Mechanism:** The `_fetch_all_from_finnhub_endpoint` function was created as a generic orchestrator. It uses the `date_utils.get_dates_in_range` helper to split the total 90-day period into smaller, sequential chunks (e.g., 7 days each). It then iterates through these chunks, making a separate API call for each one and aggregating the results.
    - Strategic Ordering: The get_dates_in_range function was intentionally designed to generate these date ranges in reverse chronological order. This means the first API call fetches the most recent week of data, the second call fetches the week before that, and so on. This is to prioritise freshness: in the unlikely event the application hits an API rate limit mid-fetch, it will have already successfully retrieved the most recent and relevant data for the user.

4.3. Memory-Efficient Data Handling: Generators

Throughout the data processing pipeline, Python **generators** are used extensively.

- **Pipeline:** There are 3 key stages in the pipeline.
    - Preprocessing: Yields an event after cleaning (drop data with missing details) and transforming the data (such as giving a score) for each endpoint.
    - Processing: Yields an event after processing the data (which may involve aggregation for calculating net insider transactions for the day, for example, from observations of the dataset) for each endpoint.
    - Post processing: Yields an event after processing events in context with events from across other endpoints.
- **Mechanism:** The various `_*process_*` methods yield one event at a time instead of building a complete list in memory.
- **Rationale:** This approach is highly memory-efficient. The application processes data as a stream, reducing the need to load the entire raw dataset from all APIs into memory at once (unless context is needed).

  While the current 90-day scope is manageable, this design ensures the application can be scaled to handle much larger date ranges or more verbose APIs in the future without running into memory constraints.
- **Consideration:** In situations where the entire dataset is needed for multiple pipeline steps (for multi-pass processing etc), it is more efficient to collect the data into a list once and pass it around. While generators excel at streaming, repeatedly iterating over one to build a full list would negate its memory-saving benefits. In such cases, a list is more appropriate.

4.4. Event Standardisation & Dynamic Scoring

A critical step is transforming the varied responses from different APIs into a single, standardised format.

- **Standardisation:** Each `_process_*` function is responsible for converting a raw API response into a consistent dictionary structure, including keys like `std_date, type`, `title`, and `importance_rank` (from `_preprocess_*` functions). This decouples the rest of the application from the specific structure of any single API.
- **Dynamic Scoring:** The `importance_rank` is not a static value. For macro news from Alpha Vantage, it is dynamically calculated by factoring in the `relevance_score` specific to the queried ticker. This ensures that macro news directly relevant to the ticker ($NVDA) is ranked higher than general market news.

  The dictionary comprehension (`ticker_sentiments = {s['ticker']: s for s in ticker_sentiment_list}`) is used for a clean and efficient O(1) average-time lookup of the relevant sentiment data; while a traditional for loop would have the same overall asymptotic time complexity of O(n) (since dictionary construction would be O(n)), the difference is in readability, conciseness, and potential for better performance.

4.5. Performance Optimisation I: Server-Side Caching

To manage strict API limits (especially Alpha Vantage) and long load times for events, a file-based caching strategy was implemented. This provides near-instantaneous loads for repeat users.

- **Mechanism:** After successful API fetch and processing, the final list of events is saved to a JSON file in the `/cache` directory, named by ticker (e.g., `NVDA_events.json`), along with a timestamp.
- **Logic:** On subsequent requests, the `DataManager` first checks for a valid cache file. If a cache from the last 6 hours exists, it is served immediately, completely bypassing all API calls and processing. This design dramatically improves performance and respects API usage limits.

  A `force_refresh` option allows the user to manually bypass the cache.
- **Consideration:** Although historical price and ticker icon can also be cached, they load decently fast and hence, are not cached.

4.6. Performance Optimisation II: Efficiently Filtering Top Events

To keep the UI clean, the information panel displays a maximum of the top 20 events for any given day.

- **Initial Approach:** Sort the full list of daily events by `importance_rank` (O(n log n)) and take the top 20.
- **Optimised Solution:** `heapq.nlargest` is used. This is algorithmically more efficient for finding a small number of the largest items from a collection. Its O(n log k) time complexity (where n is the number of events and k is 20) is significantly faster than a full sort O(n log n), especially if a day has a large number of events.

4.7. Potential Optimisation: Removing Duplicated News Events

A significant challenge was handling news where the same story appears from multiple sources with minor title variations.

- **Initial Approach:** A dictionary key of `(date, title)` was used. Subtle differences (e.g., whitespace, punctuation) in titles from different sources may pass.
    - Further optimisation: A possible optimisation is to use a set instead for quick membership lookups.

      However, while perfect for checking for the *presence* of a unique item, sets can't store values to allow for a *comparison* (replace news with one that has a higher `importance_rank`). Furthermore, in Python, both sets and dictionaries are built on hash tables. In both cases, the average time complexity for checking if an item exists is approximately O(1) (constant time). Therefore, a dictionary is more appropriate.

- **Current Approach:** Normalising the title with `.strip().lower()` for more robust comparisons.
- **Possible Solution:** A function (such as `difflib.SequenceMatcher`) can be used to calculate the similarity ratio between two strings. By comparing titles for a similarity score above a certain threshold (e.g., 90%), the application can robustly identify and filter out syndicated duplicates. `importance_rank` can be used as a tie-breaker (ensure the version from the most reputable source is kept, for example, if there is a credibility scoring matrix).

5. User Experience (UX) Considerations

5.1. Loading State vs. Streaming Data

The possibility of "streaming" events to the UI as they arrive from each API was considered. However, this creates a disorienting user experience. A user might see an initial set of events and assume the data has fully loaded, leading to confusion when more items pop in later or when they are ‘unable’ to click the chart to show the day’s events..

Therefore, a **clear loading state** was chosen. The application waits for all data to be fetched and processed before updating the information panel, showing a loading spinner while waiting. This provides a clear, unambiguous signal to the user that the data they are seeing is complete for that request.

5.2. Prioritising the Price Chart

It was observed that the price chart data from yFinance loads significantly faster than the event data from other APIs. To enhance perceived performance, the data loading was split (hybrid ‘streaming’). The price chart is rendered first, allowing the user to immediately begin analysing price action while the more complex event data loads in the background.

6. Future Work & Potential Improvements

- **Implement Analyst Recommendations:** Integrate the Finnhub `recommendation_trends` endpoint to add another layer of analysis.
- **Advanced Ticker Input:** Implement input validation and a search feature for ticker symbols to improve usability.
- **Secure Cache:** For a production environment with multiple users, the file-based cache could be encrypted, although it is low priority as the current data is not sensitive.
- **Refine UI/UX:** Add more sophisticated loading indicators and potentially allow user customisation of event types.
- **Charting**: Include technical indicators

7. Appendix: Sample API Response

Below are cleaned `JSON` samples of a single item from each endpoint.

- Alpha Vantage `economy_macro` endpoint: `feed`


```
{
    "title": "AI Data Center Investments By Mag 7 Companies… ",
    "url": "https://www.benzinga.com/markets/equities/…",
    "time_published": "20250910T122636",
    "authors": [ "Namrata Sen"],
    "summary": "The race among the \"Magnificent Seven\" tech giants...",
    "source": "Benzinga",
    "topics": [
        {"topic": "Financial Markets", "relevance_score": "0.266143" }
    ],
    "overall_sentiment_score": 0.155546,
    "overall_sentiment_label": "Somewhat-Bullish",
    "ticker_sentiment": [
        {
            "ticker": "NVDA",
            "relevance_score": "0.42",
            "ticker_sentiment_score": "0.22",
            "ticker_sentiment_label": "Somewhat-Bullish"
        }
    ]
}
```


- Finnhub `company_news` endpoint:


```
{
    "category": "company news",
    "datetime": 1569550360,
    "headline": "More sops needed to boost electronic…",
    "id": 25286,
    "image": "https://img.etimg.com/thumb/msid-71321314…",
    "related": "AAPL",
    "source": "The Economic Times India",
    "summary": "NEW DELHI | CHENNAI: India may have to offer…",
    "url": "https://economictimes.indiatimes.com/…"
 }
```

- Finnhub `stock_insider_transactions` endpoint: `data`


```
{
    "name": "Kirkhorn Zachary",
    "share": 57234,
    "change": -1250,
    "filingDate": "2021-03-19",
    "transactionDate": "2021-03-17",
    "transactionCode": "S",
    "transactionPrice": 655.81
}
```


- Finnhub `filings` endpoint:


```
{
    "accessNumber": "0001193125-20-050884",
    "symbol": "AAPL",
    "cik": "320193",
    "form": "8-K",
    "filedDate": "2020-02-27 00:00:00",
    "acceptedDate": "2020-02-27 06:14:21",
    "reportUrl": "https://www.sec.gov/ix?doc=/Ar…",
    "filingUrl": "https://www.sec.gov/Archives/edga…"
}
```


- Finnhub `company_profile2` endpoint:


```
{
  "country": "US",
  "currency": "USD",
  "exchange": "NASDAQ/NMS (GLOBAL MARKET)",
  "ipo": "1980-12-12",
  "marketCapitalization": 1415993,
  "name": "Apple Inc",
  "phone": "14089961010",
  "shareOutstanding": 4375.47998046875,
  "ticker": "AAPL",
  "weburl": "https://www.apple.com/",
  "logo": "https://static.finnhub.io/logo/…",
  "finnhubIndustry":"Technology"
}
```