The Goobers

Conor O'Mahony 19462894

Thomas Pollock 17483996

Vincentiu Ciuraru-Cucu 21208899

Synopsis:

TickerTrek is a real-time, distributed, trading data provision system, created to help traders make informed buy/sell decisions on Apple stock. This distributed system polls three different API providers for financial information (e.g. stock open/close price), processes the data into a standard format, predicts price movement using machine learning, serves the predictions, and aggregates historical price data for access through a simple and intuitive web application UI. Ultimately, this system is a simplified version of a standard high-frequency trading (HFT) system, minus the automated trade execution, which is instead left to the trader to carry out independently.

Technology Stack

- *Apache Kafka (with KRaft): This is a distributed message broker which enables the real-time asynchronous communication necessary between the nodes in the architecture. KRaft uses a Raft-based consensus mechanism to manage metadata and ensure high availability and fault tolerance of the nodes.*
- *Spring Boot: This provides the backend framework for the project's REST APIs for retrieving historical data, as well as WebSocket support.*
- *WebSockets: Necessary for displaying real-time price updates and up/down price movement signals on the frontend.*
- *Vite, React, Axios, Redux & Material UI: JavaScript libraries used for the creation of the user interface, consisting of several intuitive components, and featuring global state management and efficient data loading.*
- *MongoDB: This is a NoSQL database which is used to store historical stock data and the real-time predictions of the machine learning model.*
- *Sci-kit learn: Python library used to generate the machine learning model for price movement predictions and feature engineering.*
- *Docker: For containerisation of the individual services, which enables simple deployment and scalability.*

Kafka was chosen as it is the industry standard for real-time distributed stream processing of data [1]. Considering we were creating a trading system with real-time updates, it was necessary to have as little latency as possible when passing price update messages between the nodes in the architecture. As such, Kafka's reliable and fault-tolerant real-time pub-sub message delivery system was an obvious choice for us. Kafka relies on metadata management for leader elections in partitions and ensuring fault tolerance which it doesn't do itself. KRaft was chosen for this over the standard choice of Zookeeper as it integrates metadata management directly into Kafka using the Raft consensus mechanism. This simplifies the architecture and enables a much simpler setup and improved stability. [2]

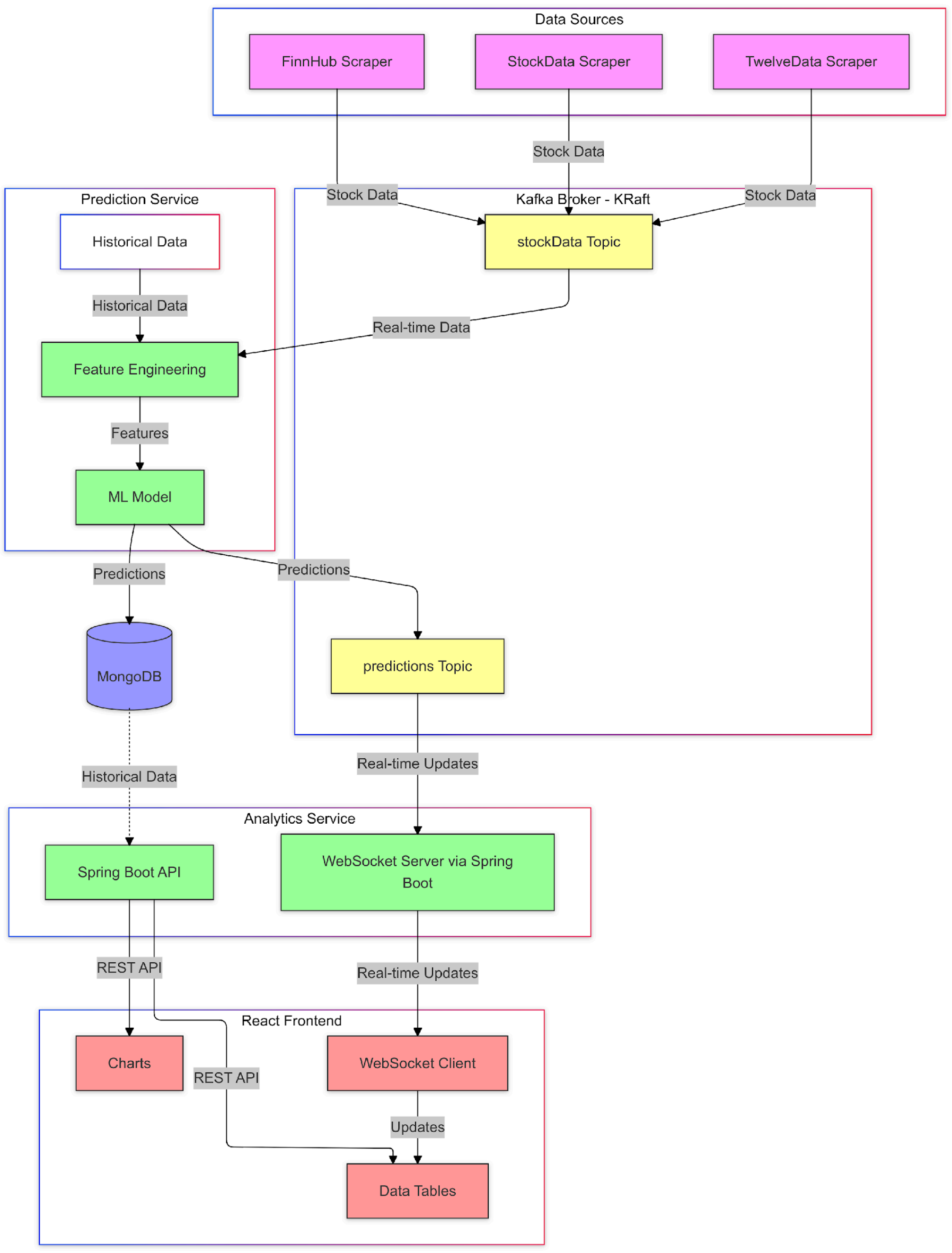
Spring boot was chosen as it allowed us to easily implement backend microservices. It enables simple REST API and WebSocket creation ideal for serving endpoints for historical, real-time and aggregated stock price data.

MongoDB was chosen as the database to store historical stock data and predictions as it supports real-time updates much better than a relational database like SQL, as well as distributed clusters which ensures scalability and fault tolerance of the database. [3]

Both React and scikit-learn were chosen because they were familiar to us and they would allow us to achieve associated goals effectively.

Finally, Docker with compose was crucial as it greatly simplified deploying the system on other machines. This is especially important considering there are many interacting components in the architecture which would otherwise be overly complicated to set up on different machines.

System Overview



Components:

1. Data Scraper Nodes (Finnhub, Twelvedata, Stockdata): Each of these nodes implements an AbstractAPIScraper class to fetch AAPL stock data from their respective API by constantly polling them as fast as their free tier limits will allow. These particular API's were chosen as they provide the best free tier polling rates currently available. Optimally, a websocket would be used which would retrieve price updates faster than polling, although this is generally not offered by companies for free, limiting us to polling. The respective scapers then transform the data into a standardised Stock class format, stringify it and publish it to the Kafka "stockData" topic. Multiple API's were used to combat polling rate limits and reduce the bias associated with data coming from one single source.

2. Prediction Node: The prediction node is a Kafka consumer, subscribed to the "stockData" topic. This node uses Python and its scikit-learn library to make price movement predictions. On system startup, the node updates the historical data of the database using an Alpha Vantage stock API call. It takes in the real time price messages from the "stockData" topic and combines this with historical price data pulled from the

MongoDB to engineer new features, such as 3/5/10 minute rolling averages of price. It then runs a logistic regression model to predict whether the price will go up or down next. The input data and its output prediction are saved in the MongoDB as well as published to a Kafka topic called "predictions" for consumption by the analytics service.

3. Analytics Node: This node consists of a Spring Boot application, incorporating a Kafka consumer and WebSocket setup, a custom MongoDB model and interface, and a REST API. It consumes the stream of predictions from the "predictions" Kafka topic and enables its transmission to the client via WebSocket. It uses Spring Boot to create a backend framework for the data, supplying REST API endpoints to retrieve historical data, as well as price signal aggregations, which are retrieved from MongoDB upon request.
4. Client Node: The client implements a React-based frontend and Material UI components to provide a clean and simple user interface to display the real-time price updates, historical price action, as well as a graph of historical price movement. It also displays the aggregated number of real-time and historical price up/down signals. Under the hood, Redux enables effective application state management, Axios facilitates the passing of promise-based HTTP requests between client and server, and Vite enables faster and leaner development and deployment.

This system has been designed with both scalability and fault tolerance in mind. Apache Kafka is highly scalable due to its distributed architecture. The topics used are divided into partitions, handled by different brokers [1]. This allows parallel processing of the messages, enabling high message stream throughput. Brokers can easily be added to the Kafka cluster to scale horizontally as the rate of data flow grows. The Spring Boot Microservice can easily be replicated to handle increasing load on the API/WebSocket endpoints. This is the same for the machine learning node. Likewise, MongoDB supports sharding, which allows distribution of the data over multiple servers. [4] This ensures that as the amount of historical data grows, the database can horizontally scale such that time to query stays low. All of the services incorporate fault tolerance quite well, especially considering they can be restarted by Docker on failure in the worst case. Kafka is made fault tolerant by its replication mechanism. Partitions can have multiple replicas and they can be stored across multiple brokers. On failure, Kafka elects a new leader for the affected partition ensuring data is not lost. [1] KRaft simplifies metadata management which will reduce the chance of inconsistency during recovery from failure. Spring boot could theoretically be decoupled into prediction service and data scraping service for enhanced failure tolerance, but in the event of failure, Docker can quickly restart this service, ensuring minimal downtime. Likewise, the frontend can swiftly be restarted in the event of failure, although this is highly unlikely to fail. As such, it made sense to decouple it from the backend. MongoDB has replica sets which ensure very high availability so even if the primary replica set fails, a secondary one is automatically promoted to the primary, ensuring great fault tolerance. [4] The ML node uses a Kafka queue so if it fails and goes offline, only a maximum of 1 message can be lost. Once restarted, the node can continue to consume messages where it left off.

Hence, it is clear that by appropriately utilising our chosen distributed technologies, our system is both highly scalable and fault-tolerant, capable of adapting to increasing workloads and recovering in the face of failures.

Contributions

Conor:

- Set up the project structure;
- Developed the data scraper modules (finnhub, stockdata, twelvedata)
- Set up Apache Kafka with KRaft and the base Docker compose for the project
- Engaged in troubleshooting problems across the application
- Wrote the README file, and co-wrote the report.

Thomas:

- Developed the prediction module;
- Trained the logistic regression prediction model;
- Set up the cloud-based MongoDB shard for storage purposes;

- Co-wrote the report.

Vincentiu:

- Developed the analytics and client modules;
- Co-wrote the report;
- Prepared the project video.

Reflections

One of the key challenges we faced in this project is that we were using a significant amount of new technologies that we had little to no experience in. None of us had experience in Kafka and we had very limited experience with MongoDB, React and WebSockets. As a result, we had to learn while we developed, which significantly slowed our development time and led to us having to cut features. We were initially going to give the user the option to choose from a variety of stocks. Due to time constraints, we selected one of them (AAPL) as a proof of concept and to demonstrate the system. The setup is such, however, that expanding the system would be straightforward.

Another significant challenge was finding market data API providers which offered a high enough number of API requests per minute such that we could simulate real time conditions. The Finnhub API allows 60 polls per minute, which is the best we could attain for free, whereas the other two only allow 800 and 100 polls per day respectively. Ideally, all 3 would have WebSocket connections to supply actual real time updates to the ML node, however this is impossible without incurring expenses beyond what is reasonable for the module. Another problem was storage: to have a centralised hub for the historical data, we had to set up a MongoDB shard in the cloud. The alternative to this would have been passing the database around to every person who wants to use the service. Obviously, this is infeasible.

If we could re-start the project, we would explore the technologies we used deeper before trying to implement them for our specific use case. This would help mitigate the steep learning curve we encountered during implementation. A good idea would be to implement more rigorous automated testing early in the development process. This would've helped identify and resolve bugs more efficiently. This is especially true considering we spent much time debugging problems that the prediction service had when it encountered NaN values. We could better de-couple our architecture. The analytics service could be split into smaller microservices to enhance its fault tolerance. Finally, implementing container orchestration from the start also would've given us the best scalability possible.

Through this project we were exposed to many new technologies and their associated limitations and benefits. We learned that Kafka is incredibly powerful for handling real-time, low latency, high throughput data streams. The only problem with this is that limited documentation exists for setting up Kafka with KRaft, though we chose to pursue this anyway for a more simplistic end result. Spring boot was excellent for setting up REST APIs and WebSockets for our service with ease, however we found that the service was slower to start in comparison to the other nodes. If it were to encounter a failure, it would take a non-negligible amount of time to recover and hence it would have benefitted from having a standby replica node that could be swapped in in the event of failure. MongoDB was highly effective for our use case however its syntax was somewhat unintuitive, as we are mostly familiar with SQL syntax. React allowed us to create a very aesthetically pleasing dynamic frontend, however it came with some difficulties, taking more development time than expected. Docker was highly effective for us in transferring the project between computers and we saw no downsides for our use case.

Some further improvements that could be implemented include backtesting the machine learning model and regularly re-training it (ideally after the stock market closes) to ensure the model is as accurate as possible. From a distributed systems standpoint, we could introduce gRPC for efficient inter-service communication between the modules. Finally, it would be a good idea to explore Kubernetes for container orchestration for this project which would ensure it has the best scalability possible.

References:

[1] Narkhede, N., Shapira, G., & Palino, T. (2017). *Kafka: the definitive guide: real-time data and stream processing at scale*. " O'Reilly Media, Inc.".

[2] *Kraft - Apache Kafka Without Zookeeper* (2020) *Confluent*. Available at: https://developer.confluent.io/learn/kraft/ (Accessed: 03 January 2025).

[3] Khedkar, S., Thube, S., Estate, W. I., & Naka, C. (2017). Real time databases for applications. *International Research Journal of Engineering and Technology (IRJET)*, *4*(06), 2078-2082.

[4] Bradshaw, S., Brazil, E., & Chodorow, K. (2019). *MongoDB: the definitive guide: powerful and scalable data storage*. O'Reilly Media.