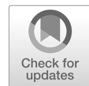
Leveraging Cloud-Native Microservices Architecture for High Performance Real-Time Intra-Day Trading: A Tutorial



Mousumi Hota, Ahmed M. Abdelmoniem, Minxian Xu, and Sukhpal Singh Gill

Abstract Day trading has been gaining attention from prospective investors over the past decades, even more so in the last decade due to a plethora of factors such as instantaneous availability and accessibility to information such as social media, news, Internet of Things (IoT), availability of market's sentiment data associated with them, and increased broker discounts. This tutorial aims at providing a framework for intra-day trading that supports scalability, easily maintainable by creating a low coupling, high cohesion, and stateless architecture between client and server, considering the time-sensitive nature of transactions involved. This provides the benefits of additional business value for Software as a Service (SaaS) providers based on high productivity as well as enhanced end-user experience. To achieve these objectives, a combination of cloud-native architectural components, such as microservices and event streaming using Kafka, is used in this tutorial to provide a near real-time experience to end users. Additionally, to ensure security, robust authentication management is used in the proposed solution to control the access of read and write operations on the Firebase cloud database.

Keywords Cloud computing · Cloud-native architecture · Distributed systems · Microservices · Real-time stock monitoring

1 Introduction

Over the last decades, intra-day trading has evolved from a topic of interest within the regular trader's community to pre-investors, such as friends and close contacts,

---

M. Hota · A. M. Abdelmoniem · S. S. Gill (✉)
School of Electronic Engineering and Computer Science, Queen Mary University of London, London, UK
e-mail: ahmed.sayed@qmul.ac.uk; s.s.gill@qmul.ac.uk

M. Xu
Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China
e-mail: mx.xu@siat.ac.cn

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

M. Kumar et al. (eds.), *6G Enabled Fog Computing in IoT*,
https://doi.org/10.1007/978-3-031-30101-8_5

who are not actively involved but started taking an interest in trading. Due to the availability of Software as a Service (SaaS) start-ups, intra-day trading has further piqued their interest. Since the art of intra-day trading can only be learned through observation and experience [1], when it comes to day trading, prospective investors need to focus their attention on a magnitude of trading metrics all at once in order to take holistic and pragmatic decisions regarding their investment. However, users are apprehensive about using such platforms due to concerns such as security, overall web application performance, and ease of use. On the other hand, intra-day trading SaaS providers face challenges such as scalability, cost-effectiveness, and high availability because of the high frequency of transactions involved.

The tutorial aims at presenting high-performance real-time stock data monitoring on a dashboard to support end users who are interested in day trading [21]. Momentum-based strategies were adopted as stock price changes(or volatility) could be magnified and hence could help users make better stock order decisions [1]. For enabling end users to have a 360-degree view of 5 leading and 3 lagging indicators, to assist them in making the assessment of risks and profits associated over an interval of 1 min (as long as the connection persists). The implementation of the project considers criteria such as Kafka event streaming, which offers low latency, flexibility, high throughput and failover (by considering multiple Kafka brokers) [2]. On the server side, it considers server-sent events to provide end users with the best experience of the consolidated stock chart view, updated every minute without any delay (with the exception of the delay on intervals mentioned).

1.1 Main Contributions

The main contributions of this work are:

- Proposed a framework cloud-native architectural framework for web applications using techniques such as microservices, Firebase for the backend, Kafka for middleware, and Websocket server for establishing communication in a publish, subscribe manner.
- Designed microservices based approach to fetch stock data from Finhubb stock API using Asynchronous Promises.
- Built highly secure user management using Firebase authentication service.
- Optimised Enabled event streaming Kafka consumer by availing multi-broker support.
- Optimized user's order processing using Firestore for easier, fast access to backend user's data.
- Implemented the proposed architecture using web application run time environment as nodejs on both client and Server side.
- Presented detailed analysis of momentum trading strategy using leading indicators.
- Proposed future directions that could bolster the cloud offerings SaaS providers.

1.2 Article Organisation

The rest of the tutorial is structured as follows: Section 2 presents related work. Section 3 gives details about the technologies used. Section 4 provides broader insight into application architecture. Section 5 comprises implementation considerations. Section 6 concludes the proposed solution and highlights future directions.

2 Related Work

We have identified only one similar work in the literature. In [24] authors has adopted Kafka for event streaming by providing supporting evidence that increasing large event streaming leads to high processing time [24]. The authors have further provided evidence on how Kafka could make event streaming fault-tolerant, fast and secure. However, the author's work differs from the proposed solution in this paper, as it focuses on data analysis aspects of tweet data. In [24], authors have delineated how we can control access to stock data, that is, whether to share or hide required services using the Microservices approach. This could be very useful for scaling applications in future. Literature on cloud-native architecture reported [4–7] that this work is not been applied to investigate intra-day trading.

3 Technologies Used

This section introduces the details about the technologies used in this tutorial.

3.1 Kafka

Kafka provides continuous processing of events sequentially. Since consumer producers are decoupled, both can evolve and fail independently of the topic/Stream. Ordered records are immutable and can only add at the end. Records numbering usually starts from 0 and monotonically increases.

3.2 Kafka Message

Every event is a key (domain object that you can serialise and store there, key is integer/string), value pair. Every message has a timestamp, is automatically provided, and can be set explicitly.

3.3 Broker

It manages partitions or takes requests from updates from producers, and update them on partitions. Take requests from a consumer and writes them out. Storage and pub-sub Schema of the topic-retention period, compaction properties. Brokers need to agree on a consensus, zookeeper ensures:

- Leader Election
- Access control list
- Replication organization
- Failover

Broker ensures:

- Partition Rebalancing: The Kafka consumer group performs rebalancing of workload when a new partition is added
- Parallelization: With parallelization, multiple consumers can access data across multiple systems
- Durability: It is ensured by Kafka replication [8]

3.4 Partition

Each partition has a considerable amount of replicas. Generally, the replication factor is 3; one of them is a leader, and the others are followers. When you produce a partition, we actually produce it for the leader. The producer is connecting to the broker that has the lead partition there. The job of the brokers with follower partitions is to reach out to leaders and scrape the new messages and keep them up to date with them ASAP. Language support:

- Java native library
- Kotlin, Closure, Groovy
- Failover
- Supportive languages: Python, C, C++ (library Kafka)

3.5 Producer

Producer applies round-robin scheduling for writing to a topic. In reality, it actually writes to a partition. Partitions are even; a lot of orders in not in place in the round-robin mechanism. If the order is important, they need to be ordered by a key; the producer can hash that key and mod that key with the number of partitions, given the partition number that the producer is going to write to. The same key is always going to write to the same partition as long as the number of partitions is constant (usually the case).

3.6 Leading Indicators

Leading indicators are used to predict a trend before the phenomenon has happened and hence help traders/prospective traders make assessments of risks and benefits of their investment in advance. We discuss the key leading stock data indicators used in our application as below:

OHLCV—Open, High, Low, Close, Volume Represents Open, High, Low, and Close, Volume of a ticker every 1 min interval from Finnhub API.

RSI—"Relative Strength Index" [8]

- A technical indicator that shows traders the magnitude in which the price of stock changes. In general, conception, when the value is above 70%, shows that the stock is oversold, vs when the value is below 30, the stock is undersold.
- This momentum indicator is considered useful as the rising market(bull market) persists more than the bear market.

Stoch—"Stochastic Oscillator" [9]

1. This momentum indicator is based on the movement of the close price of a stock with a range of prices.
2. It relies on the price history
3. Its sensitivity can be smoothed using the moving average

ADX—"Average Directional Strength"

1. It is used to indicate the strength price trending, whether the price is going up, upward trend or moving down, downward trend.
2. Its trend strength by ADX can be measured as: if between 0 and 25 then weak, then the trend is strong, 50–75 shows very strong, and above 75 extremely strong.

OBV—"On-Balance Volume" [10]

1. This momentum indicator is based on changes in volume.
2. It demonstrates crowd sentiment for predicting whether the result is rising or falling.
3. The calculation is based on the principle that if consecutive close prices are the same, then obv is 0, if the current close is more than the previous close, then the volume is considered positive else, negative.

CCI—"Commodity Channel Index" [22]

1. This momentum indicator indicates the trend of when a stock is about be oversold or stock is over-brought
2. If overbought the price is corrected by market soon, indicating user it is not the right time to sell the stock. If oversold there is potential for the stock price to rebound soon, hence user can wait for that time to buy the stock.
3. If CCI rises from negative value to a value above 100, it shows uptrend.

3.7 Lagging Indicators

Lagging indicators are based on historical data and are used to identify patterns in data. Below are examples of Lagging indicators.

SMA—Simple Moving Average [23]

1. SMA calculation is based on average of closing prices(generally) from a selected range of closing prices.
2. Indicates if price would rise in future or not.
3. It smooths out price volatility to make it easier for traders to observe and analyse the price trend.

EMA—Exponential Moving Average [11]

1. It is based on selected range of prices, where more weightage is given to price data that is more recent.
2. It indicates points in chart where intersection price of security and EMA (in this case) occurs to signal buy, sell.
3. It is more efficient indicator than SMA.

TEMA—Triple Exponential Moving Average [12]

1. It is a technical indicator.
2. It follows same calculation as EMA with an exception that lags are subtracted, to capture quick changes in the price.
3. Its trend direction can be identified using TEMA.
4. Is more efficient indicator than EMA.

In order to have a full view of a company, other microservices were included:

```
get_FINNHUB_Quote ,
get_FINNHUB_Company_Profile ,
get_FINNHUB_Income_Details ,
get_FINNHUB_performance ,
get_FINNHUB_peers ,
get_FINNHUB_topnews ,
get_FINNHUB_recommedation ,
```

3.8 Microservices

With the growing size of systems, scalability becomes an issue. Using "Cloud-native architecture" such as microservices helps overcome this issue as they could be easily understood and scaled without dependency on each other [13]. In terms of software

development, it supports continuous delivery [14]. It supports scalability, reusability, resiliency, and cost-effectiveness. This can be supporting by evidence author in [3].

Microservices are created to read data from **Finnhub stock API** and were processed to useful JSON format for making it suitable for data consumption on the client side. Microservices were implemented using **Nodejs** using respective npm packages for the APIs indicated earlier. The following are details on how to use the microservices:

- No state or session variables are shared between the client and server.
- Each microservice is implemented as promise calls to the stock API endpoints within an asynchronous function using nodejs in two javascript files. The data returned was then processed in a required format and exposed to Kafka producer. In this approach caller (producer) starts producing messages when the data has been received from the microservices modules, that is, when the API end point promise calls were resolved. In the event of errors, for instance, if API call limit was reached per minute, error handled was also implemented in the microservices module.
- For real-time event streaming, from a performance perspective, Kafka was found to be a suitable technology for middleware.
- Having multiple brokers ensures durability; hence based on the requirements and scope of the project, the number of Kafka brokers chosen was three. Further, a centralized service was required to manage Kafka brokers within such distributed system setup to ensure robust synchronization, maintenance of metadata, and list of topics and messages produced and consumed. This was facilitated by **zookeeper**.
- The simplest way to install the aforementioned for setting up the environment for using Kafka was using a docker-compose command, file extension, and .yml. as the installation required multiple containers. A docker-compose YAML file was created with all the required configuration details including port numbers for respective Kafka, and zookeeper servers. As per the config file used, the Kafka broker used was named kafka_broker_1, kafka_broker_2, kafka_broker_3, and the zookeeper container was named zookeeper.

3.9 Producer-Consumer

- Kafka producer-consumer were implemented using kafkajs: kafkajs producer, consumer: kafkajs consumer kafkajs npm package link
- Kafka producer was used for writing to a topic on a specific partition so that in case of partition failure, topics written on other partitions were still available. Using Kafka subscribers, the written topics were subscribed to and consumed by Kafka consumers.

Observation When partitions were increased and set to 10 from 2, during testing, the consumer hanged and so took more time to start consuming topics.

3.10 Earlier Considerations

The chosen interval for the topic message creation is 1 min. Hence the producer needs to push the messages to Kafka topics and then be consumed by Kafka consumers on the server side. As soon as the client requests for the stock data to be rendered on the front end. Hence **sse**, server-sent events server-side events. On the client end, **EventSource API** methods are used to handle the message received from the consumer, which acts as a server in the given context.

4 Application Architecture

The application architecture developed in this work is shown in Fig. 1.

The architecture consists of Kafka service that manages the producers and consumers. The zookeeper service which manages the coordination among the various components in the application. The database service which stores and maintains the data. The APIs that are used to communicate with the client and apply the trade instructions. In the following, we discuss the details of the application architecture which is based on the Client-Server Communication paradigm. Specifically, we

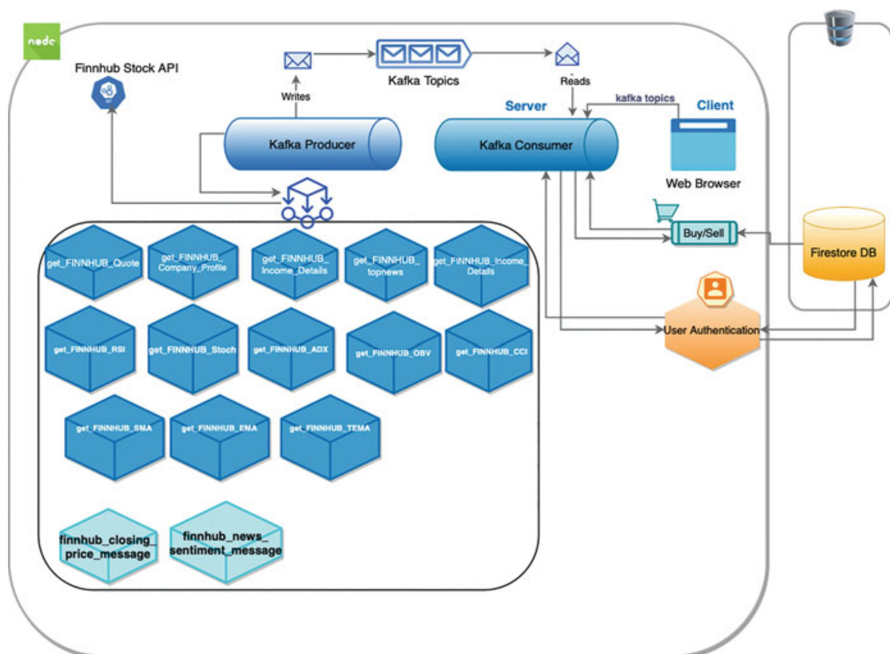


Fig. 1 Application architecture

analyze and select which method is most suitable for client Server Communication. We depict the client server communication options in this architecture from existing work [15]. It shows short polling, long polling and server-based events types of communication that are possible.

5 Implementation Highlights

- For any given day, the Finnhub stock API gives access to stock data available from 12:00 am to 9:00 am, weekends (Saturday, Sunday), hence date input is adjusted to the previous date for uninterrupted user experience.
- As Finnhub stock API used UNIX-based timestamps to fetch data from the API. Hence current date has been converted to human readable form (with suffixes AM or PM) and from Unix timestamps.

5.1 Brief Introduction to the API

- Finhubb stock API that sources its data from multiple stock exchanges.
- Keys are obtained for gaining authorized access to the API endpoints.

5.2 Technology Stacks

Next, we describe the main technologies used in the development of the application.

Nodejs Nodejs is a cross-platform and open-source server environment that can run on Windows, Linux, Unix, macOS as well as other operating systems. It is used as a back-end JavaScript runtime environment to run on the JavaScript Engine, and execute JavaScript code outside a web browser.

Backend As the data are obtained and some amount of latency is expected, the below mentioned methods are implemented which promises better error handling as compared to its contemporary approaches.

Cloud Database Firebase provides us with the needed hosting services. It can host any application type (Android, iOS, Javascript, Node.js, Java, Unity, PHP, C++, etc.). The benefits is that it can run NoSQL databases and services like a real-time communication server and provide real-time hosting of content, social interactions (e.g., Twitter, Google, Facebook, and Github), and notifications.

Middleware Docker was used for building, deploying and managing containers. Docker containers are used to dockerize the various services used in the architecture

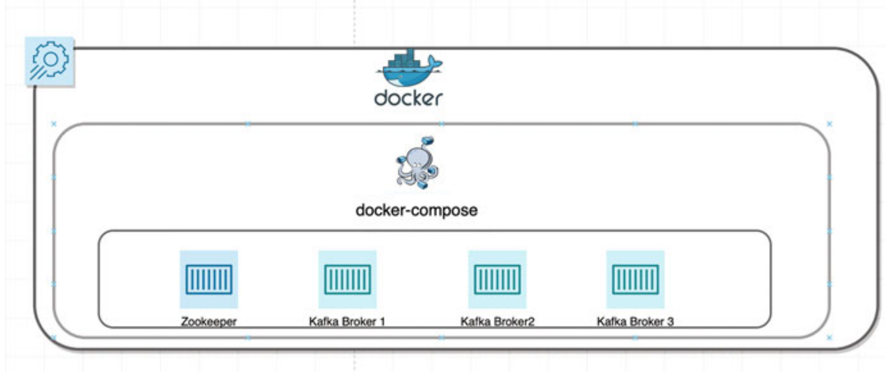


Fig. 2 Dockerizing Zookeeper, Kafka, and KSQLDB servers

such as Zookeeper, Kafka and KSQLDB servers as shown in Fig. 2. The containers required in the application implementation are as follows:

– Three **Kafka brokers** to enhance resiliency in case of any failovers, one container was build and deployed for each broker Note: There containers were configured to enable kafka producer and consumer to be able connect and write to, read from an available, elected leader broker from the pool of deployed brokers.
– One container was configured **zookeeper** for managing kafka brokers.

Since multiple containers were required to be implemented, **docker-compose** tool is selected as it is aptly suited for the purpose of defining and sharing such multi-container applications.

– **Kafkajs** ("Apache Kafka client for Nodejs")
– **Producer Script** Kafka producer is implemented to make asynchronous promise calls to stock-micro microservice

Monitoring Docker Containers Docker containers were monitored using docker desktop, using the log option under any specific container. For usage monitoring stats, stats option was used to view the statistics.

This could also be achieved using:

– **docker stats**: this command is used to view statistics such as memory usage, network usage for all the containers.
– **docker ps**: this command os used to list the containers to view statistics for those containers.
– **docker ps -all**: this command is used to list for all the containers.
– **docker container ps –filter "status=exited"**: this command is used to check the containers that have exited or stopped

- **docker container prune**: this command is used to remove container that exited, though they could be removed individually or at once using the bin symbol besides the container listing inside docker desktop, to bin symbol on the top right of all containers listing.
- **docker logs –since=30m 29ef5d137b2b**: this is a sample command that was used to get logs since past 30 min of container **29ef5d137b2b**, which is a kafka container id.
- Alternatively **docker logs –since=30m kafka6** could be used using the container name directly to retrieve the log where the**kafka6** is the container name.

Environment details for any specific container was used to view the setup details such server port number for a container, using the **inspect** option.

Observation Earlier KSQLDB was adopted to run queries on kafka stream, however as per application requirement, it was only needed to store user's relevant order transactions. Implementation details, a container for building and deploying ksqldb server for real time stream processing of intended kafka topic(s).

5.3 The Key Microservices Used in the Application

Below are the details of the microservers used in the application and their invocation API.

- **Quote Service**: gets stock quote for a given day.
- **Profile Message**: gives an overview of company name, industry.
- **Candles Service**: provides high, low, open, close prices for a stock based on the interval earlier selected by user.
- **News Service**: provides top company news, source and URL to give insights into current market sentiments associated with company. In the application implementation top 10 news are displayed.
- **Income Service**: provides company liquidity over last years in terms of **o**ne of current ratios metrics, P/E: profit to earning ratios over last years. It could serve as important criteria for a user to find for analysing how risk or "future growth" associated when buying stock of that company. Low P/E ratio indicates high risk, can also be interpreted as low growth. Low P/E ratio indicates high risk, can also be interpreted as low growth.
- **RSI Service**: shows the relative stock index value for a stock based on the interval earlier selected by user up to the current time on any given day.
- **OBV Service**: shows the balance stock index value for a stock based on the interval earlier selected by user up to the current time on any given day.
- **STOCH Service**: shows the balance stock index value for a stock based on the interval earlier selected by user up to the current time on any given day.
- **ADX Service**: gives back average directional index value for a stock based on the interval earlier selected by user up to the current time on any given day.

- **CCI Service**: brings back on commodity channel index value for a stock based on the interval earlier selected by user up to the current time on any given day.
- **SMA Service**: gives back simple moving average value for a stock with interval of 5 min based the current time on any given day.
- **EMA Service**: presents the exponential moving average value for a stock with interval of 5 min based the current time on any given day.
- **TEMA Service**: provides the triple exponential moving average value for a stock with interval of 5 min based the current time on any given day.

Below is the name of the API (or function) as implemented in the application:

```
Quote Service: finnhub_quote_message();
Profile Service: finnhub_profile_message();
Income Service: finnhub_income_message();
News Message: finnhub_news_message();
Candles Service: finnhub_candles_message();
RSI Service: finhubb_rel_index_message();
OBV Service: finnhub_obv_message();
STOCH Service: finnhub_stoch_message();
ADX Service: finnhub_adx_message();
CCI Service: finnhub_CCI_message()
SMA Service: finnhub_SMA_message();
EMA Service: finnhub_EMA_message();
TEMA Service: finnhub_TEMA_message();
```

5.4 The Prediction Services Used in the Application

Closing Price Prediction This is Closing Price Prediction for user's selected ticker was obtained by analysing the closing price data obtained from OHLC data using deep learning approach. This microservice was used to predict the subsequent 10 min closing prices. The Closing Price Prediction is implemented as a Nodejs module and the API is listed below:

```
get_ClosingPrice_Prediction
```

Observations The following are the main observations about the closing price prediction service.

- Two methods were implemented and compared for price prediction: Training accuracy and **LSTM timestep neural network** approach lead to better results as compared to **Recurrent neural network**;
- **minmaxscaler** module was as feature scaling method in the data preprocessing step to normalize to ensure that the neural network model's prediction the different in magnitude of closing price values

News Sentiment Prediction Gives the users valuable insights on top news obtained based of ticker user selected on the client side form, sentiment analysis was performed on news summary data obtained from all top news category To achieve this it is required to have **Tokenization function** which is performed as data preprocessing steps before applying the analyzer on each news summary.

Observations The following are the main observations about the news sentiment prediction service.

1. Three stemming were implemented and compared such as porter **porterStemmer**, **Lancaster Stemmer**, **Snowball Stemmer** were implemented and compared for price prediction:
2. Training accuracy and **LSTM timestep neural network** approach lead to better results as compared to Recurrent neural network;

5.5 The Client Side of the Application

Next, we discuss how the client side of the application is implemented.

1. On the client side stocks charts were rendered using to provide end users with instant insight into variation in stock prices, based on ticker and interval criteria set by a currently logged in user.
2. Stock fundamental data for basic company overview we stock quote, p/e ratios, top 10 news and associated market sentiment classified as bearish, bullish, neutral. Bearish category indicated pessimistic sentiment associated with a stock news summary indicating a stock price would fall in future, where a or bullish category indicated optimistic sentiment indicating a stock price would fall in future.
3. User could hover over any chart on the dashboard page and click on data points on any chart to initiate a buy or sell transaction. In this context, user clicks on a canvas element on HTML5 rendered using express handlebars. where charts are rendered using javascript functions. Threshold values were indicated for buy and sell, or to indicate a trend on in the technical indicator charts (OBV, ADX, SMA, EMA, TEMA) for providing baseline functionality support in user's decision making while placing order.
4. Due to asynchronous nature of promises, kafkajs library supports for consumer configuration on server side, kafka message topics could be sent using websocket only using a interval based approach, where 1 min is least interval limit. However on client side it appeared that end users would need to wait. Hence producer script was designed to sending stock data as kafka topics over the desired interval (1, 5, 10 min). On the other had consumer was designed to send the consumed message more frequently, sending the same consumed message obtained for say 1 min every 20 s, until producer would have write to the topics in next 1 min. We show a sample of this script in Fig. 3.

```
const {lineCh } = require("./create_charts");

const kafka = new Kafka({
  clientId: "kafka_consumer",
  groupId: "kafka_consumer_group",
  brokers: ["localhost:9092", "localhost:9093", "localhost:9094"],

});
var ticker;
const consumer = kafka.consumer({
  groupId: "kafka_consumer_group",
  allowAutoTopicCreation: false,
});
const topics = [
  "alpha_intraday_topic"
]; /*["alpha_intraday_topic", "alpha_profile_topic", "alpha_quote_topic", "alpha_income_topic",
"finnhub_profile_topic","finnhub_quote_topic", "finnhub_candles_topic","finnhub_income_topic","alpha_rsi_topic"
,"alpha_stoch_topic","alpha_adx_topic","alpha_obv_topic", "finhubb_rsi_topic","finhubb_stoch_topic","finhubb_adx_topic", "finhubb_
"alpha_sma_topic","alpha_ema_topic" ,"alpha_tema_topic", "finnhub_sma_topic", "finnhub_ema_topic", "finnhub_tema_topic"]

*/
```

Fig. 3 Kafka producers and consumer

5. To keep client side light weight, most of the stock data processing was done on the nodejs microservices modules.

5.6 Firebase Cloud Database

1. **User Authentication**: As security remains to be one of majors concern for cloud service provider, in the proposed web application firebase backend authentication is used. Firebase backend authentication services were used to create user account, verify sign in or sign out. At any given point in the web application only user's display name was shared between the front end HTML pages. The user has to first login to the platform using the assigned credentials as shown in Fig. 4.
2. textbfOrder Processing: When User would click on a data point on any of the charts, it would lead to subsequent order page requesting user to submit, and then confirm the details of order. Two types of transactions are allowed either Buy or Sell.

In both the options user were only required to mention the quantity, and select suitable option from buy or sell from drop down in the UI as the price for data point is auto-filled in the form. Upon submission it would lead to a conformation page showing the total order value, current portfolio balance. The ticker selection choice is shown in Fig. 5 and the user ticker fundamental page is shown in Fig. 6.

5.7 Challenges

The challenges faced during the implementation of the trading application are:

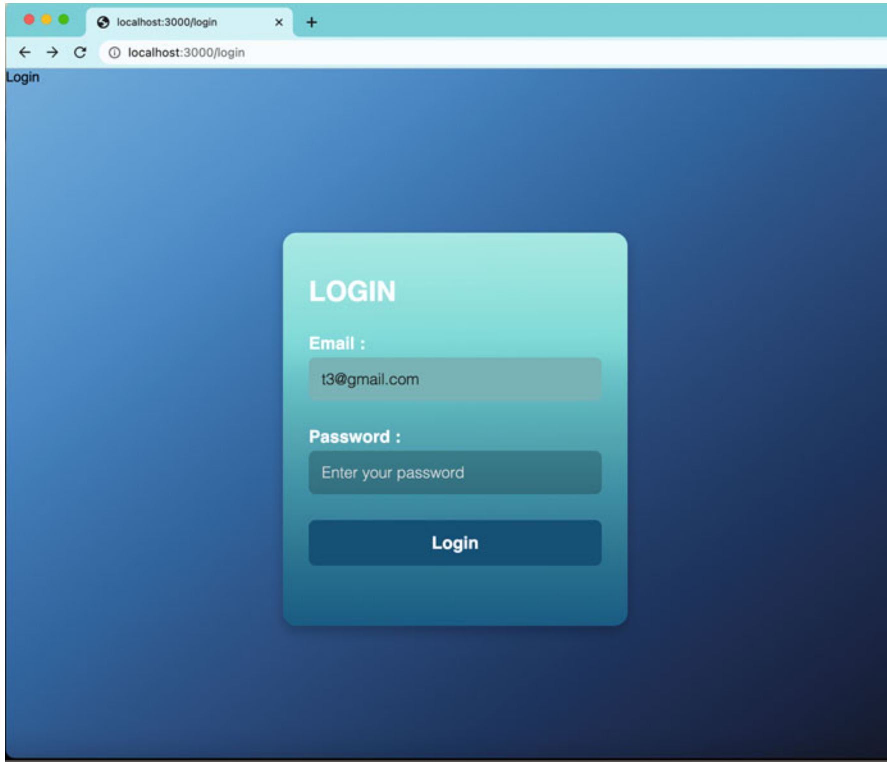


Fig. 4 Web application UI login webpage

1. It was observed that due to the unavailability of Kafka broker, it was required to redeploy the kafka, zookeeper containers. Hence, the number of Kafka brokers, also referred as storage layer, was increased to 3 brokers. Usually at least one of the brokers is available at any given time, sometimes the leader election (managed by Kafka) takes more time.
2. Lack of proper documentation support, examples for some of nodejs libraries.

6 Conclusions and Future Work

A light weight, cost effective, secure and scalable web application was created as a cloud service, emphasising on cloud native infrastructure considerations such as on backend, cloud database, Firebase was used for scalability with no downtime. In the middleware, kafka broker containers were fully managed by docker, that were easily, quickly deployed on cloud. Kafka producer consumer were implemented for real-time streaming of stock data as Kafka topics. Kafka was preferred in the

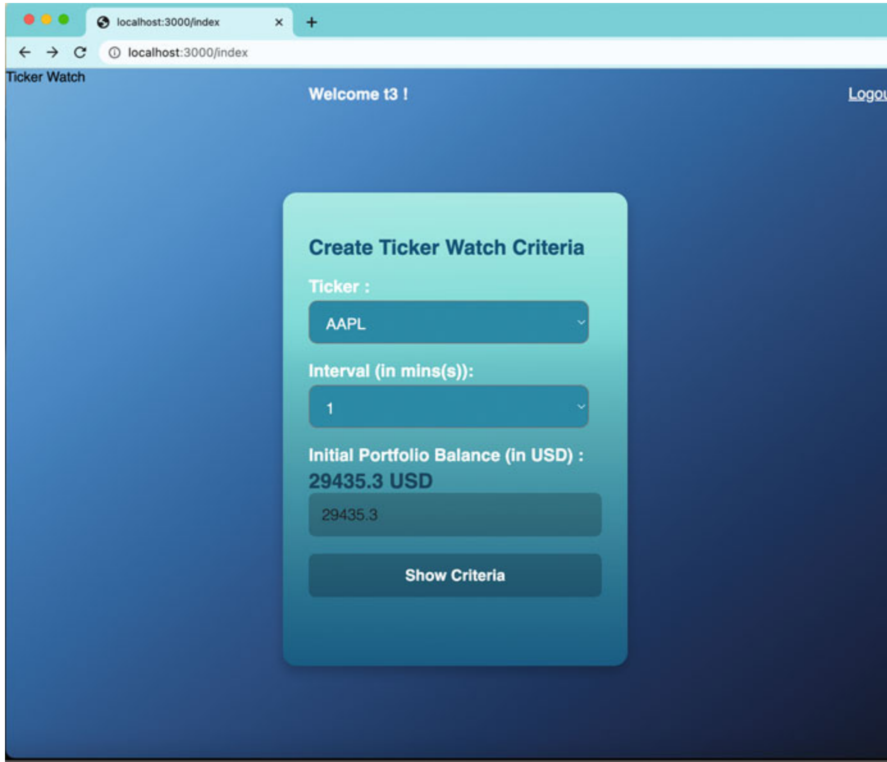


Fig. 5 User ticker selection page

middleware since load balancing, also known as scaling out, is managed internally by kafka based on partitioning and rebalancing. Websocket was used for faster client server communication to HTTP. In contrast to REST API calls, Microservices were created ensure scalability in terms of software development requirement. With low coupling between the microservices, a SaaS provider wishes to scale, can add more adding any modules in future depending on business requirements. For startup SaaS service providers planning to scale, dealing with high volume of sensitive data, these considerations would be helpful for providing high performance, secure and scalable framework, keeping end users experience at heart.

6.1 Future Directions

The proposed architectural framework could be applicable to other time-sensitive business verticals, such as e-commerce, and hospitals, amongst others, in which Kafka could be configured to fetch more data as per the data consumption require-

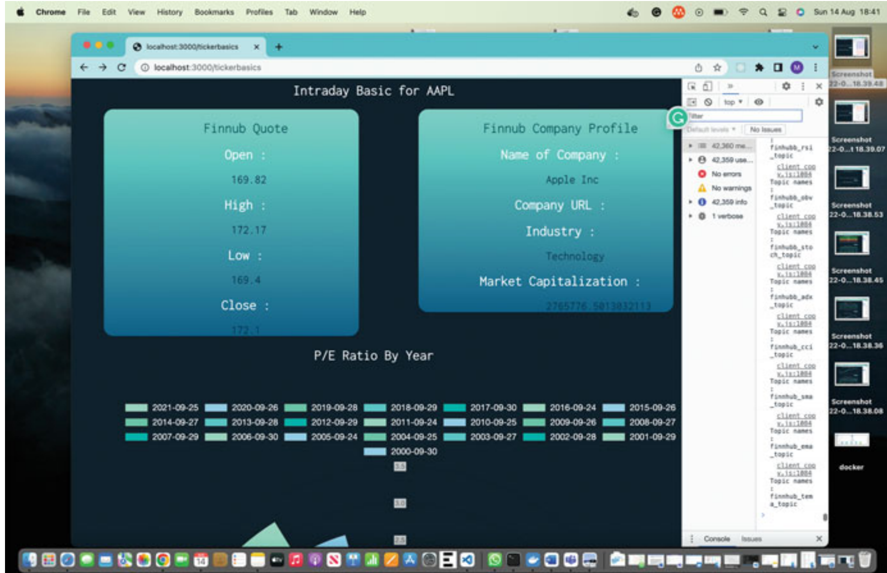


Fig. 6 User ticker fundamental page

ment. In future work, from an end-user perspective, integrations to platforms such as Slack could be built, so that users could get instant notifications on stock prices. From a SaaS provider's perspective, Big Query integration could be built to analyse user google analytics metrics when the user base expands to that of an enterprise to understand user behaviour [16]. On the SaaS provider end, Jira integration could also be built for easier tracking of any user management issues (in the current context of the project). Additional load balancing measures could be taken using the NGINX load balancer or Kubernetes load balancer to manage containers effectively from Kafka's internal load balancing [17]. To enhance the analytics of the application, distributed machine learning could utilized with acceleration techniques to speed the processing [18, 19]. Finally, to speed up the communication between Kafka and Microservices, techniques for improving the TCP performance can be applied [20].

Acknowledgments This work is partially funded by Chinese Academy of Sciences President's International Fellowship Initiative (Grant No. 2023VTC0006), National Natural Science Foundation of China (No. 62102408), Shenzhen Science and Technology Program (Grant No. RCBS20210609104609044), and Shenzhen Industrial Application Projects of undertaking the National key R & D Program of China (No. CJGJZD20210408091600002). We also declare that this work has been submitted as an MSc project dissertation in partial fulfilment of the requirements for the award of degree of Master of Science submitted in School of Electronic Engineering and Computer Science of Queen Mary University of London, UK is an authentic record of research work carried out by Mousumi Hota (first author) under the supervision of Sukhpal Singh Gill (last

author) and refers other researcher's work which are duly listed in the reference section. This MSc project dissertation has been checked using Turnitin at Queen Mary University of London, UK and submitted dissertation has been stored in repository for university record.

References

1. Chung, J., Choe, H., & Kho, B.-C. (2008). The impact of day-trading on volatility and liquidity. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.1855759
2. Stopford, B., & Newman, S. (2018). Concepts and patterns for streaming services with Apache Kafka designing event-driven systems. [online] Available at: https://sd.blackball.lv/library/Designing_Event-Driven_Systems_(2018).pdf
3. Salah, T., Jamal Zemerly, M., Yeun, C. Y., Al-Qutayri, M., & Al-Hammadi, Y. (2016). The evolution of distributed systems towards microservices architecture. In *11th international conference for internet technology and secured transactions (ICITST)* (pp. 318–325).
4. Shah, S. D. A., Gregory, M. A., & Li, S. (2021). Cloud-native network slicing using software defined networking based multi-access edge computing: A survey. *IEEE Access, 9*, 10903–10924.
5. Ammi, M., Adedugbe, O., Alharby, F. M., & Benkhelifa, E. (2022). Leveraging a cloud-native architecture to enable semantic interconnectedness of data for cyber threat intelligence. *Cluster Computing, 25*, 3629–3640.
6. Valdez, M. G., & Guervós, J. J. M. (2021). A container-based cloud-native architecture for the reproducible execution of multi-population optimization algorithms. *Future Generation Computer Systems, 116*, 234–252.
7. Saxena, H., & Pound, J. (2020). A cloud-native architecture for replicated data services. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*.
8. Fernando, J. (2019). Relative Strength Index – RSI. [online] Investopedia. Available at: https://www.investopedia.com/terms/r/rsi.asp
9. Hayes, A. (n.d.). Stochastic oscillator. [online] Investopedia. Available at: https://www.investopedia.com/terms/s/stochasticoscillator.asp
10. Hayes, A. (n.d.). On-Balance Volume (OBV). [online] Investopedia. Available at: https://www.investopedia.com/terms/o/onbalancevolume.asp
11. Investopedia. (n.d.). Exponential Moving Average - EMA. [online] Available at: https://www.investopedia.com/terms/e/ema.asp
12. Mitchell, C. (n.d.). Triple Exponential Moving Average (TEMA) definition. [online] Investopedia. Available at: https://www.investopedia.com/terms/t/triple-exponential-moving-average.asp
13. Xu, M., Song, C., Ilager, S., Gill, S. S., Zhao, J., Ye, K., & Xu, C. (2022). CoScal: Multifaceted scaling of microservices with reinforcement learning. *IEEE Transactions on Network and Service Management, 19*, 3995–4009.
14. Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2016). Migrating to cloud-native architectures using microservices: An experience report. In A. Celesti & P. Leitner (Eds.), *Advances in service-oriented and cloud computing. ESOCC 2015*. Communications in computer and information science (Vol. 567). Cham: Springer. https://doi.org/10.1007/978-3-319-33313-7_15
15. Long-Polling vs WebSockets vs Server-Sent Events, Online Link https://systemdesignbasic.wordpress.com/2020/02/01/12-long-polling-vs-websockets-vs-server-sent-events/
16. Iftikhar, S., Gill, S. S., Song, C., Xu, M., Aslanpour, M. S., Toosi, A. N., et al. (2022). AI-based fog and edge computing: A systematic review, taxonomy and future directions. *Internet of Things, 21*, 100674.
17. Gill, S. S., Xu, M., Ottaviani, C., Patros, P., Bahsoon, R., Shaghaghi, A., et al. (2022). AI for next generation computing: Emerging trends and future directions. *Internet of Things, 19*, 100514.

18. Gajjala, R. R., Banchhor, S., Abdelmoniem, A. M., Dutta, A., Canini, M., & Kalnis, P. (2020). Huffman coding based encoding techniques for fast distributed deep learning. In *Proceedings of the 1st Workshop on Distributed Machine Learning*.
19. Abdelmoniem, A. M., & Canini, M. (2021). DC2: Delay-aware compression control for distributed machine learning. In *IEEE Conference on Computer Communications (INFOCOM)*.
20. Abdelmoniem, A. M., & Bensaou, B. (2017). Enforcing transport-agnostic congestion control in SDN-based data centers. In *IEEE 42nd Conference on Local Computer Networks (LCN)*.
21. Kuepper, J. (n.d.). An introduction to day trading. [online] Investopedia. Available at: https://www.investopedia.com/articles/trading/05/011705.asp
22. Mitchell, C. (n.d.). Commodity Channel Index - CCI definition and uses. [online] Investopedia. Available at: https://www.investopedia.com/terms/c/commoditychannelindex.asp
23. Hayes, A. (n.d.). Simple Moving Average - SMA. [online] Investopedia. Available at: https://www.investopedia.com/terms/s/sma.asp
24. Lee, C., & Paik, I. (2017). Stock market analysis from twitter and news based on streaming big data infrastructure. In *IEEE 8th international conference on awareness science and technology* (pp. 312–317).