# CS-E4780 Scalable Systems and Data Management Course Project Detecting Trading Trends in Financial Tick Data

Linh Ngo
Aalto University
Espoo, Finland
linh.l.ngo@aalto.fi

Doğa Türkseven
Aalto University
Espoo, Finland
doga.turkseven@aalto.fi

Oben Yozgyur
Aalto University
Espoo, Finland
oben.yozgyur@aalto.fi

## Abstract

This report outlines the design, implementation, and performance evaluation of our solution to the course project, which focuses on real-time complex event processing of high-volume financial tick data. The challenge involves efficiently computing trend indicators and detecting patterns used by traders to guide buy/sell decisions in financial markets. Our solution incorporates a custom windowing mechanism that leverages event semantics to optimize the processing of streaming data. We assess the system's performance by evaluating its scalability, resource utilization, and the accuracy of trend indicators. The results highlight the effectiveness of our approach in processing large-scale data and accurately identifying financial patterns in real-time.

## Keywords

High-frequency Trading System, Apache Kafka, Apache Flink, InfluxDB, Grafana

**ACM Reference Format:**
Linh Ngo, Doğa Türkseven, and Oben Yozgyur. 2024. CS-E4780 Scalable Systems and Data Management Course Project Detecting Trading Trends in Financial Tick Data. In . ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

The ability to make real-time decisions from continuous streams of financial data is crucial in today's fast-paced stock market environment. Stock market analytics must process a vast amount of data, including price changes and transaction volumes, within moments to identify trends and opportunities. This fast-moving landscape highlights the critical need for efficient data processing systems that can effectively navigate market fluctuations and provide timely insights.

In light of these challenges, the main goal of this project is to design and implement an efficient trading system capable of processing and analyzing high-frequency market data to support trading decisions. The trading process consists of ingesting a large volume of trading events, performing streaming processing, and producing outputs that mimic real-time trading actions such as buy, sell, or hold decisions.

The data provided for this project is real-world market data provided by Infront Financial Technology GmbH in 2021 which is publicly available [9]. The data set includes one week of trading events from November 8th to 14th, 2021 on three major exchanges in Europe, with events record the tick data event of 5504 financial instruments.

This project aims to develop a software/data platform solution that addresses two primary tasks using a provided financial dataset:

**Query 1: Exponential Moving Average (EMA) as quantitative indicators**

The first query focuses on EMA calculation, which is a crucial financial indicator used to identify trends in stock prices over time. The EMA is defined as follows:

$$\mathrm{EMA}^{j}_{s,w_i} = [\mathrm{Close}_{s,w_i} \cdot \frac{2}{(1+j)}] + \mathrm{EMA}^{j}_{s,w_{i-1}} \cdot [1 - \frac{2}{(1+j)}] \quad (1)$$

with: $|w|$ : window duration in minutes,
$j$ : smoothing factor for EMA with $j \in \{38, 100\}$,
$s$ : symbol where $s \in S = \{s_1, s_2, \ldots, s_n\}$,
$\mathrm{Close}_{s,w_i}$ : last price event for $s$ observed in window $w_i$,
$\mathrm{EMA}^{j}_{s,w_0} = 0$ : initial EMA value.

The EMA calculation requires the grouping of incoming events into 5-minute, non-overlapping windows, both by symbol and time, to ensure accurate and timely computation of the metrics. This approach ensures that trends and breakout patterns, such as bullish or bearish crossovers, are detected in real-time for each symbol.

**Query 2: Breakout Patterns: Crossovers**

The second query builds upon the first query, which aims to detect breakout patterns by analyzing the per-symbol EMA computed at every 5-minute, non-overlapping intervals. A bullish breakout occurs when the price rises steadily, while a bearish breakout occurs when the price falls steadily. Identifying these trends timely allows traders to act quickly such as buying during bullish breakouts or selling during bearish ones to maximize profits.

- **Bullish pattern** can be detected and *buy event* must be generated if and only if:

$$EMA38_{s,w_i} > EMA100_{s,w_i} \quad \text{and} \quad EMA38_{s,w_{i-1}} \leq EMA100_{s,w_{i-1}}$$

- **Bearish pattern** can be detected and sell event must be generated if and only if:

$$EMA38_{s,w_i} < EMA100_{s,w_i} \quad \text{and} \quad EMA38_{s,w_{i-1}} \geq EMA100_{s,w_{i-1}}$$

In this paper, we propose a solution to both queries and provide a detailed explanation of the architecture and implementation of our


, ,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

trading system. The outline of the paper is as follows. Section 2 outlines the system architecture and presents the key logic implementation. Section 3 elaborates more on the practical implementation, including the methodology for the hardware, programming tools, and GUI tools used. Section 4 presents a performance evaluation of the system, where different indicators such as its efficiency and scalability are accessed. Finally, Section 5 offers a conclusion about the findings and discusses potential improvements.

## 2 System Architecture and Logic Implementation

This section outlines the system design and architecture of a high-frequency trading system aimed at processing and analyzing trade data in real-time. The system is composed of four main layers: (1) Data Ingestion and message distribution layer that utilizes Apache Kafka, which efficiently ingests and distributes financial tick data from external market feeds; (2) Data Streaming and Processing layer that utilizes Apache Flink to perform real-time stream processing, including complex event processing, aggregation, and stateful computations; (3) Data Storage layer that utilizes InfluxDB, a time-series database designed to store the processed data and metrics for efficient querying and historical analysis; and (4) Data Visualization or Frontend layer that utilizes Grafana to visualize the data stored in InfluxDB to provide real-time dashboards for monitoring market conditions, trading strategies, and system health. Together, these components enable the system to handle high-frequency data, perform rapid analytics, and support real-time decision-making in a trading environment.

Our system is designed as a data-streaming application for real-time trade data processing. We leverage the concept of data streams in Apache Flink to process a continuous flow of financial market data as it arrives to allow for real-time decision-making.

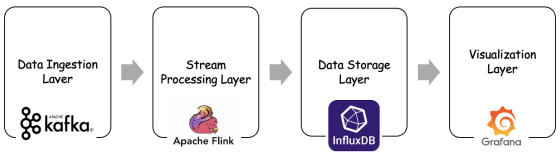


**Figure 1: Architecture of trading system layers**

### 2.1 Data Ingestion and Message Distribution

In the data ingestion process, the system begins by establishing a connection to the Kafka broker through the `create_producer` function, which initializes a SerializingProducer in Python. This producer is responsible for sending data to Kafka. To ensure a successful connection to the broker, the `create_producer` function retries up to five times. It configures the producer with essential settings, including the broker's address and the necessary serializers for both the key and value. The key is serialized as a UTF-8 encoded string, while the value is serialized as a JSON object, which represents the actual trade data to be transmitted.

After that, trade data is fetched from provided remote CSV files which are retrieved via HTTP requests. Each CSV file contains financial tick data and is filtered to only retain relevant attributes that are crucial for downstream processing and analysis. The core attributes of interest and their descriptions are in Table 1. Furthermore, rows with missing or null values for critical fields such as `Trading time` and `Trading price` are excluded to ensure that only actionable data is transmitted for downstream analysis.

**Table 1: Descriptions of Useful Attributes from the Dataset**

| Attribute | Description |
|---|---|
| **ID.[Exchange]** | Unique identifier for the symbol, including the trading exchange: Paris (FR), Amsterdam (NL), or Frankfurt (ETR). |
| **SecType** | Security type, represented as either [E]quity or [I]ndex. |
| **Last** | The last trade price for the symbol. |
| **Trading Date** | System date for the last received update. |
| **Trading Time** | Time of the last update (bid/ask/trade). |

For every incoming event, the filtered data is ingested into the Kafka queue. Kafka organizes data into topics that represent distinct streams of messages. In our system, all trade data is stored in a single topic, as all information comes from one continuous streaming source. To support parallel processing and enhance throughput, the topic is partitioned to allow multiple consumers to process different subsets of data concurrently. Partitioning in our system is based on the instrument ID to ensure that events related to the same instrument are processed within the same partition. This approach maintains the order of events for each instrument and supports horizontal scaling, which allows the system to efficiently process large volumes of data.

To ensure messages are sent reliably and no data is lost, the Kafka producer calls `poll()` function after sending each message. This step allows Kafka to process any pending messages and address issues such as a full queue to ensure that the system runs smoothly by managing delays and delivering messages reliably.

### 2.2 Stream Processing Layer

The Stream Processing Layer is a crucial part of the trading system and is responsible for the real-time processing of data streams and the execution of complex event processing on trading data. This layer utilizes Apache Flink, a distributed stream processing framework known for its low-latency and high-throughput capabilities, which makes it ideal for handling large volumes of financial data in real-time. Flink uses operators which are modular units of computation that sequentially apply various processing tasks on incoming data streams. These operators are designed to support parallel execution and allow the system to scale efficiently across multiple nodes.

The pipeline in our system is designed around Flink's execution model, which operates as a directed acyclic graph (DAG), where each node represents a specific operation. The pipeline begins by ingesting data from data sources and passes down to consumers for downstream processing. Then, core Flink operators perform critical tasks, such as consuming incoming data, applying time-based windowing for aggregation, calculating EMA, and detecting

significant market events such as crossovers. Finally, the processed data is written to sinks, such as time-series databases, where it is stored for further analysis and visualization.

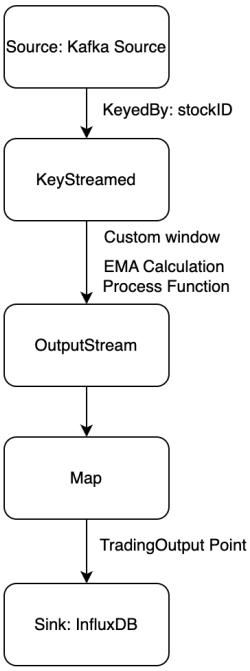


**Figure 2: Stream Processing Jobs**

**Source operator**. In our system, the KafkaSource connector [3] is used to ingest real-time trading data from Kafka into Flink. The system ingests data by subscribing to Kafka topic where trade data is published, serving as the first layer of data ingestion. The source is configured to start consuming data from the earliest available offset to ensure that no trade events are missed, even if there is a delay in the arrival of messages. During ingestion, the source operator uses a custom deserializer `JSONValueDeserializationSchema` to convert the incoming JSON-encoded trade data into `TradeData` objects. To ensure accurate event-time processing, we utilize the watermark strategy in Flink to assign event timestamps to trade data as it is ingested from Kafka. This approach ensures that exvents are processed based on their actual occurrence of trading time, even if they arrive out of order.

**Listing 1: TradeData Class Definition**

```
class TradeData {
    String id;
    String secType;
    String lastTradePrice;
    String tradingTime;
    String tradingDate;
}
```

**Window operator**. In this system, we implement the logic for our own 5-minute, non-overlapping window instead of using the defined window operator in Flink. As we prioritize the correctness, event-time semantics is leveraged instead of system-time or processing time to reflect the actual occurrence of events to prevent slight temporal discrepancies which can skew analytical insights. This approach prevents minor temporal discrepancies from distorting the analysis. To achieve this, we utilize a monotonous event timestamp watermark strategy, where each incoming event has a timestamp that is always greater than or equal to the previous event, as seen in the data stream source.

The windowing mechanism leverages Flink's event-time timer service to effectively manage the lifecycle of each window. When a new event arrives, the system compares its timestamp with the start time of a current window. For example, if an event's timestamp is 00:03:00, it is assigned to the window from 00:00:00 to 00:05:00. Upon detecting the arrival of a new event and determining that it falls outside the current time window, the system registers an event-time timer to fire at the window's closing time (e.g., 00:05:00 for a 5-minute window). When this timer triggers, it indicates that the window has reached its end, and it is time to calculate EMAs and generate output for that window.

One of the key strengths of this design is its ability to handle late and out-of-order events, which are common in distributed systems due to network delays or unsynchronized sources. To ensure completeness, the window remains open until an event with a timestamp beyond the window's closing time is encountered, preventing premature closure. While this approach improves analytical accuracy and data integrity by ensuring that all relevant events are included in the window, it can also introduce processing latency and increased resource consumption, especially under high data volumes. This represents a trade-off between prioritizing accuracy and completeness of the data over real-time performance, a crucial consideration in financial applications.

**EMA calculator and crossover detections operator**. The system is designed to process real-time financial data streams using stateful stream processing within the Apache Flink framework. Specifically, the EMA is calculated for each stock symbol uses prior EMA values to ensure accurate and continuous computation. To achieve this, the system employs Flink's `KeyBy` [6] and `KeyedProcessFunction` [7].

Initially, the `KeyBy` operator is applied to the input data stream to partition the stream based on unique stock symbols. This ensures that the events for each stock symbol are processed independently. Subsequently, the EMA values are calculated separately for each stock symbol using the `EMACalculationProcessFunction`, a custom implementation of the `KeyedProcessFunction` in Flink.

Within the `EMACalculationProcessFunction`, `ValueState` in Flink [8] is utilized to maintain key components, which includes the last observed trade price, the previous EMA values, and the current window state. These states are essential for recursive EMA calculations, which use the EMA values from previous window combined with the latest trade price from the current window. The `ValueState` also tracks the start time of each time window to ensure only the relevant data contributes to EMA computations.

This function leverages Flink's design for efficient state updates as new events arrive. Timers are employed to manage time windows

and trigger actions at the end of each window. When a timer fires, it initiates the EMA computation and evaluates crossover patterns simultaneously. Based on these patterns, buy or sell advisories are generated. After this process is done, `TradingOutput` objects for each stock symbol is produced, which encapsulates the calculated EMA values and any detected breakout patterns, such as bullish or bearish crossovers, along with the corresponding advisories for trading actions.

**Listing 2: TradingOutput Class Definition**

```
class TradingOutput {
    private String symbol;
    private String tradingTime;
    private String tradingDate;
    private double lastTradePrice;
    private double ema38;
    private double ema100;
    private double prevEma38;
    private double prevEma100;
    private Integer eventCode;
    private String advisoryReason;
    //Bullish Crossover, Bearish Crossover
}
```

**Sink operator**. In Flink, a data sink serves as the final destination for a data stream, where the processed data is stored or forwarded. Sinks are commonly configured to connect with external systems like databases or streaming platforms. Flink utilizes its connectors to efficiently transfer data from the stream to these target systems to complete the data processing pipeline [4]. In our system, the `TradingOutput` stream is directed to InfluxDB, a time-series database, as the sink. This integration facilitates high-performance storage and enables downstream processes like analytics, visualization, and alert generation using tools such as Grafana.

## 3 Data Storage and Visualization Layer

In our system, the measurement or table name is `trading_db`, where the trading outputs for all symbols are stored as Points. Each Point consists of fields representing various trade metrics, such as the last trade price, EMA values, and advisory reason. The tags is the stock symbol identifier, which allows for efficient filtering and querying of data based on the stock symbol.

Once the data is stored in InfluxDB, we integrate with Grafana, an open-source platform for visualizing time series analytics and monitoring, to retrieve and display the trade data [10]. Grafana is chosen as the visualization tool because it easily connects to InfluxDB through data connectors, offers real-time, customizable dashboards, interactive features, and powerful filtering options to focus on specific metrics or trades for efficient monitoring and analysis. Additionally, we implement alerts in Grafana to notify users of buy and sell advisories when a crossover is detected. These alerts are sent to Discord to provide real-time updates and enhance the responsiveness of the trading system. The Discord channel can be joined via this link [1].

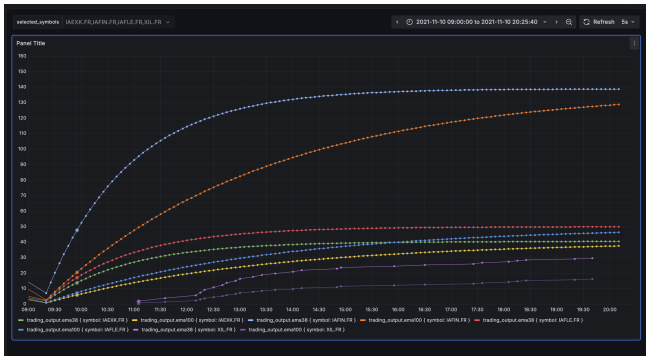


**Figure 3: Snapshot of selected stock symbols on 10/11/2021 in Grafana dashboard**

## 4 Implementation

The details of the logic implementation of the code are already included in Section 2. This section focuses on the practical aspects of the system implementation, which covers the setup, hardware/cloud services, programming tools, and GUI tools utilized in the project.

The implementation utilizes Python on the Kafka producer side and Java (OpenJDK11) for Flink processing. On the producer side, Python is used to interact with the Kafka messaging system, ingesting financial tick data, and publishing it to Kafka topic. Python is chosen due to its simplicity and robust library ecosystem for managing data ingestion and preprocessing. Additionally, Zookeeper is utilized for managing Kafka's distributed system to ensure reliable coordination and synchronization of Kafka brokers.

On the consumer side, Apache Flink is implemented using Java to perform real-time stream processing. Java is chosen for its tight integration with Flink, which provides high performance and the ability to leverage Flink's extensive APIs for stateful processing. Flink is configured with 4 task slots to enable parallel execution of tasks to maximize throughput and efficiency. In this setup, Flink consumes data from Kafka, computes EMA indicators, detects breakout patterns, and generates trading advisories. This combination of Python for data ingestion and Java for stream processing ensures a flexible, efficient, and scalable real-time trading system. To enable rapid deployment and scalability of our applications, we utilize Docker Compose [2] to orchestrate Kafka, Zookeeper, Flink, InfluxDB, and Grafana within containers. Our repository can be retrieved here [11].

## 5 Performance Evaluation

For this system, correctness was verified through a structured testing approach using a toy dataset. This dataset was constructed with carefully selected trade events, including timestamps, trade prices, and predefined expected values for EMA38 and EMA100 calculations. The `EMACalculationProcessFunction` was applied to the toy dataset, and its output was systematically compared against the known correct values.

In our experiment to assess the scalability of Flink, we compared the CPU and memory usage between two configurations: 1 task slot and 4 task slots in TaskManager for each job in Flink. With 1 task slot, Flink could only execute one task at a time and leads to

underutilization of CPU resources. Since only a single task—such as the Kafka source or one downstream operator—was running at any given moment, the CPU load remained relatively low and less consistent, with idle time between tasks. In contrast, with 4 task slots, Flink could execute four tasks in parallel, which significantly increased CPU usage. By allowing multiple tasks to run concurrently, more CPU resources were required to handle the higher throughput and parallelism. However, it resulted in better overall performance, with faster data processing and reduced latency. Therefore, while CPU usage was higher with 4 task slots, the trade-off was improved resource utilization and better scalability.

## 6 Conclusion

In conclusion, we have presented the architecture and implementation of a high-frequency trading system designed for real-time financial data processing and decision-making. By leveraging Apache Kafka, Apache Flink, InfluxDB, and Grafana, we have built a robust, scalable, and efficient solution capable of handling large volumes of financial data, performing complex calculations like EMA, and detecting breakout patterns to generate buy and sell advisory events.

For further improvement, if we had more time, we would explore the integration of additional metric reporters [5] from FLink with InfluxDB to gain deeper insights into the system's performance. By capturing detailed metrics related to task execution, resource utilization, and processing times, we could identify potential bottlenecks and areas for optimization.

## References

[1] [n. d.]. Discord Notification Linh. https://discord.com/invite/36MZ4fsa
[2] Docker. 2024. Docker. https://www.docker.com Accessed: 2024-12-06.
[3] Apache Flink. 2024. Apache Kafka Connector. https://nightlies.apache.org/flink/flink-docs-master/docs/connectors/datastream/kafka/#kafka-source Accessed: 2024-12-06.
[4] Apache Flink. 2024. Flink DataStream API Programming Guide. https://nightlies.apache.org/flink/flink-docs-release-1.20/docs/dev/datastream/overview/#data-sinks Accessed: 2024-12-06.
[5] Apache Flink. 2024. Metric Reporters. https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/metric_reporters/ Accessed: 2024-12-06.
[6] Apache Flink. 2024. Operators. https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/operators/overview/#keyby Accessed: 2024-12-06.
[7] Apache Flink. 2024. Process Function. https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/operators/process_function/#the-keyedprocessfunction Accessed: 2024-12-06.
[8] Apache Flink. 2024. Working with State. https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/fault-tolerance/state/#using-keyed-state Accessed: 2024-12-06.
[9] Sebastian Frischbier, Jawad Tahir, Christoph Doblander, Arne Hormann, Ruben Mayer, and Hans-Arno Jacobsen. 2022. *DEBS 2022 Grand Challenge Data Set: Trading Data.* https://doi.org/10.5281/zenodo.6382482
[10] Grafana. 2024. Grafana Gui. https://www.influxdata.com/grafana/ Accessed: 2024-12-06.
[11] Ozgur Turkseven, Ngo. 2024. Github. https://github.com/dogaturkseven/CS-E4780_course_project Accessed: 2024-12-06.