”ALEXANDRU-IOAN CUZA” UNIVERSITY OF IASI

FACULTY OF INFORMATICS

MASTER’S THESIS

A Comparative Study of Event-Driven Implementations for Financial Market Data Analysis

proposed by

Robert Frentz

Session: july, 2023

Coordinator

Conf. dr. Onica Emanuel

″ALEXANDRU-IOAN CUZA″ UNIVERSITY OF IASI

FACULTY OF INFORMATICS

A Comparative Study of Event-Driven Implementations for Financial Market Data Analysis

Robert Frentz

Session: july, 2023

Coordinator

Conf. dr. Onica Emanuel

Avizat,

Îndrumător lucrare de licență,

Conf. dr. Onica Emanuel.

Data: .... Semnătura: ....

Declarație privind originalitatea conținutului lucrării de licență

Subsemnatul **Frentz Robert** domiciliat în **România, jud. Iași, mun. Iași**, născut la data de **29 decembrie 2018**, identificat prin CNP **1981229226708**, absolvent al Facultății de informatică, **Faculty of Informatics** specializarea **Informatics**, promoția 2023, declar pe propria răspundere cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art. 143 al. 4 și 5 referitoare la plagiat, că lucrarea de licență cu titlul **A Comparative Study of Event-Driven Implementations for Financial Market Data Analysis** elaborată sub îndrumarea domnului **Conf. dr. Onica Emanuel**, pe care urmează să o susțin în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului ei într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice în vederea facilitării falsificării de către cumpărător a calității de autor al unei lucrări de licență, de diplomă sau de disertație și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Data: .... Semnătura: ....

Declarație de consimțământ

Prin prezenta declar că sunt de acord ca lucrarea de licență cu titlul **A Comparative Study of Event-Driven Implementations for Financial Market Data Analysis**, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test, etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de informatică.

De asemenea, sunt de acord ca Faculty of Informatics de la "Alexandru-Ioan Cuza" University of Iasi, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Absolvent **Robert Frentz**

Data: .... Semnătura: ....

Table of Contents

Chapter 1

Introduction

1.1 Background

Stock analysis is the process of evaluating a particular trading instrument, investment sector, or the overall market to make informed buying and selling decisions [6]. It involves studying and analyzing past and current data to gain insights and forecast future activity.

There are two main types of stock analysis: fundamental analysis and technical analysis. Fundamental analysis focuses on examining financial records, economic reports, company assets, market share, and other factors to determine the intrinsic value of a stock. This analysis involves studying financial statements such as balance sheets, income statements, and cash flow statements to assess a company's profitability, liquidity, solvency, growth trajectory, and leverage. Comparative analysis can also be conducted by comparing a company's financial performance to that of its peers in the same industry.

Technical analysis, on the other hand, relies on the study of past and present price action, volume, supply and demand factors to predict future price movements. Technical analysts use charts and graphical illustrations to identify trends, support and resistance levels, and potential turning points in the market. This analysis is based on the belief that historical price patterns and market behavior can provide insights into future price movements.

Market data refers to real-time event data and background information about financial instruments like equities, indices, or funds. It includes information about demand, supply, trades, and other metadata [9]. The amount of event notifications processed by technical solution providers, such as Infront Financial Technology[1], is increasing daily. In 2019, Infront processed an average of 18 billion event notifications per day, which increased to 24 billion in 2021.

Market data is provided to users at different levels of quality of information based on their subscription. Quality attributes include granularity, timeliness, and completeness, ranging from fine-granular tick data to end-of-day aggregations.

Traders, analysts, and stakeholders use market data terminals to analyze market data and make investment decisions. These terminals combine market data with metadata, news, analytics, visualization, and trading functionality.

In addition to fundamental analysis, traders also use quantitative indicators to identify price trends in financial instruments. Early identification of upward (price increase) or downward (price decrease) trends is crucial for making profitable trades.

1.2 Event-Based Systems in Finance Market

Algorithmic trading relies on timely information about global trades to make informed decisions. Many services provide vast volumes of transaction records from different markets, which need to be filtered and processed [1] . Handling large event volumes is a significant challenge, so scalability is crucial for the event processing systems. Algorithmic trading aims to take advantage of real-time information from multiple markets, filtered from the event stream. Trading policies based on value-added analytics, like the Volume-Weighted Average Price, are applied to obtain favorable positions. Calculating these value-added metrics involves defining time windows and tracking transactions within those windows. Additionally, algorithmic trading requires persisting event streams and the ability to replay them, often used for testing new trading strategies on historical data before implementation. The aforementioned algorithmic requirements find substantial support in event-based systems, which serves as a strong justification for their suitability in this particular domain.

Real-time profit and loss (P&L) analysis is a growing application of events processing systems. Market volatility and sudden bursts in activity can expose companies to unexpected risks and may confuse algorithmic trading systems. Daily P&L calculations are insufficient in such circumstances. To calculate real-time P&L, it is required to make use of live feeds from multiple markets and assets. This includes direct feeds

[1]https://www.infront.co/global/en.html

from stock exchanges and trade processing workflows. These different streams need to be standardized, cleansed, and adjusted for currencies, while establishing common reporting bases. Based on the combination of trades and market conditions, indicators are computed and aggregated to determine profit and loss. These calculations can be rolled up to different levels of aggregation, such as book or portfolio level, triggering alerts for traders accordingly.

1.3 Research Problem and Context

The need for a comparative analysis of event-based system implementations for market data analysis is becoming a significant matter. With the ever growing number of stock events produced each day, it is important to identify the right technology that can effectively handle this growing volume of data. By selecting the most suitable technology from the beginning, the probability of avoiding costly changes in later phases is increased.

These types of comparative analyses are not commonly conducted. They involve evaluating multiple technologies based on various metrics such as latency, throughput, scalability, and fault tolerance. Before proceeding with a comparative study, it is necessary to demonstrate the effectiveness of each technology within the proposed testing context. Conducting a comparative analysis can be time-consuming and challenging due to the variations in programming languages, syntax, setup, integration requirements, and dependencies associated with each technology. These differences require significant preparation time and highlights the difficulties involved in performing a thorough comparative analysis.

The context of this thesis is mainly inspired from the Grand Challenge proposed during the DEBS conference in 2022 [2]. The challenge revolves around the development of a trading strategy that aims to identify price movement trends by aggregating events within 5-minute tumbling windows and triggering buy/sell advices using event processing upon detecting specific patterns. Specifically, it involves the computation of two queries: exponential moving average and breakout patterns via crossovers.

The exponential moving average (EMA) [3] is a variation of the moving average which focuses on the importance of recent data points by assigning them a higher

[2]https://2022.debs.org/call-for-grand-challenge-solutions/

[3]https://www.investopedia.com/terms/e/ema.asp

weight. This weighting scheme makes the EMA a valuable tool for analyzing trends and capturing short-term market movements. The calculation of the EMA is done by assigning a weight to the most recent price observed in the current window in relationship with the last price observed in the previous window [9].

$$EMA^{j}_{w_i} = \left[Close_{w_i} \cdot \left(\frac{2}{1+j}\right)\right] + EMA^{j}_{w_{i-1}} \cdot \left[1 - \left(\frac{2}{1+j}\right)\right]$$

with

$|w|$: window duration in minues

$w_i : w_i = (t_a, ..., t_{a+|w|})$

j : smoothing factor for EMA with $j \in \{38, 100\}$

$Close_{w_i}$: last price event observed within window w_i

Breakout patterns serve as indications of significant changes in price development, exposing trends, even it they may be temporary. When the price begins to rise and surpasses the support area from below, it is referred to as a bullish breakout. A bearish breakout occurs when the price moderately declines and breaks through the resistance area from above. These patterns are identified by examining the computation results of the EMA indicator.

A bullish breakout pattern can be observed if and only if

$EMA^{38}_{w_i} > EMA^{100}_{w_i}$ and

$EMA^{38}_{w_{i-1}} \leq EMA^{100}_{w_{i-1}}$

A bearish breakout pattern can be observed if and only if

$EMA^{38}_{w_i} < EMA^{100}_{w_i}$ and

$EMA^{38}_{w_{i-1}} \geq EMA^{100}_{w_{i-1}}$

An additional computed query is the simple moving average (SMA2) [4] with the total periods of two. This is an arithmetic moving average calculated by adding recent

[4]https://www.investopedia.com/terms/s/sma.asp

prices and then dividing that result by the number of time periods (two). This indicator is used together with EMA by traders to increase the probability of identifying advantageous buy/sell scenarios.

$$SMA^2_{w_i} = \left(\frac{Close_{w_i} + Close_{w_{i-1}}}{2} \right)$$

1.4 Contribution

The contributions of the thesis are as follows:

- A comparative analyses of Apache Storm and Apache Flink frameworks with the intentions of providing a clear overview of the advantages/disadvantages specific to these technologies in the context of market data analysis (described in Chapter 4).
- A flexible event-based system that provides an easy way of plugging different event processing frameworks with the intentions of reducing the necessary time dealing with integration requirements and dependencies in order to conduct comparative analyses. Furthermore, the system offers a controlled and isolated testing environment that allows for the customization of allocated resources for a particular technology, such as the number of processor cores and amount of RAM (described in Chapter 3 and Chapter 4).
- A monitoring solution created with the intentions of providing visualization of various metrics and statistical results through the use of time series graphs for a comparative analysis (described in Chapter 3 and Chapter 4).
- A web server developed with the purpose of abstracting the system's specific consumption of events using web sockets, thus eliminating the requirement for web clients to implement specific communication contracts in order to request associated topics results. (described in Chapter 3).
- An overview of the current state of the art in market events processing done by analyzing current approaches and methodologies (described in Chapter 2).

1.5 Dissertation Outline

The thesis is structured as follows: Chapter 2 describes a literature review for event-based systems in market data, Chapter 3 describes the architecture of the developed system together with details about the implementations, Chapter 4 presents the evaluation results and observations of the comparative analysis, and conclusions are given in Chapter 5.

Chapter 2

Literature Review

Chapter 2 of this dissertation provides a comprehensive review of the existing literature on event-based systems for analyzing financial market data and aims to establish a contextual understanding of the subject. Additionally, this chapter delves into existing comparative studies that have been conducted between Apache Flink and Apache Storm.

2.1 Existing Approaches for Stock Analysis

One approach of stock analysis was implemented by a system called Noir [3], a specifically designed distributed data processing platform for financial computation based on events. It leverages the data flow model to provide efficient processing with minimal overhead.

The core of Noir is a data flow graph composed of operators that process input batches. These operators, including *flat map*, *group by* and *rich filter map* enable the manipulation and analysis of events and subscriptions related to stock data. This approach focuses on efficient stream parallel processing, state accumulation, and indicator computation to generate financial indicators like EMA and SMA in order to detect breakout patterns.

During the evaluation phase, the performance of this approach was assessed, with a specific focus on measuring latency and scalability. It was found that this approach that focuses on process-level parallelism played an important role in achieving impressive data processing times. Additionally, the evaluation highlighted that serialization and network related processes are the most time-consuming operations during data stream processing.

Another approach to analyze market data involves utilizing sentiment analysis [2]. This approach involves constructing a classifier that examines the connection between significant market events and Twitter posts occurring within a day, resulting in predictions based on the content of the tweets. The underlying motivation for this approach is the observation that the aggregated sentiment score reflects a notable correlation with market performance.

The algorithm involves selecting the significant event criteria, which serve as the basis for further analysis. Subsequently, appropriate pre/post/contemporaneous tweets are chosen based on the selected event criteria. These tweets are then labeled with an appropriate sentiment, such as positive for a rise or negative for a fall. Next, a classifier is trained on the labeled tweets to learn patterns and associations between tweet content and sentiment. Using the trained classifier, the algorithm predicts the sentiment of new and future tweets. The sentiments of the tweets are aggregated to determine the overall sentiment trend. Finally, based on the net aggregated sentiment, a decision is made to take a long position (positive sentiment) or a short position (negative sentiment).

For the predictions evaluation part, the algorithm focuses on testing the aggregated sentiment as a predictor of the S&P 500 index. The daily prices of the S&P 500 are obtained, and the daily returns are calculated by taking the difference between the close and open prices divided by the open price. The alignment of tweets with the next trading day's returns is done to predict the next day's performance.

The evaluation demonstrated that sentiment analysis, particularly aggregated sentiment, can predict stock market performance, especially in extreme sentiment regions. However, it is important to note that this approach does not provide protection against market volatility, and relying solely on tweets may lead to reduced accuracy in stock predictions.

Another approach to predicting stocks involves usage of data mining techniques and the auto-regressive integrated moving average (ARIMA) model [4]. The ARIMA model [6]is a popular time series forecasting model used in statistics and econometrics. It is designed to capture the temporal dependencies and patterns in a time series data set and make predictions about future values.

The algorithm involves an initial data collection step from public services that provides market data events like Google Finance. Afterwards, data processing is performed to prepare the final data set from the raw data. This involves tasks such as selecting useful subsets of data, cleaning the data, and transforming it into combined value vectors or differentiated value vectors. Finally, model estimation is performed using pre-processed historical data then used to make forecasts for future values of the time series resulting in stock predictions.

The approach presented has a significant advantage in that it utilizes the ARIMA model, which is a well-established and widely used approach in time series analysis. The ARIMA model has been extensively studied and has demonstrated its effectiveness in capturing temporal dependencies and patterns in data. However, the ARIMA model is generally better suited for short-term forecasting rather than long-term predictions. As the forecast horizon increases, the accuracy of the model may decline, and the predictions may become less reliable.

2.2 Existing comparative analyses

A similar comparative study was undertaken involving Apache Storm, Apache Flink, and Apache Storm for assessing their respective capabilities [5]. The study focused on evaluating the latency and throughput of each framework using a custom-defined benchmark system. The benchmark workload was derived from an online video game application at Rovio. The data set is specifically centered around analyzing advertisements for gem packs, utilizing two data streams: the purchases stream and the ads stream. The benchmark included aggregation queries to calculate revenue from each gem pack and correlating advertisements with revenue by joining the ads stream with the purchases stream.

It is important to note that, compared to this thesis, the aforementioned study utilized older versions of Apache Flink and Apache Storm (1.1.3 and 1.0.2, respectively). Specifically, in this thesis, most recent versions (1.16.1 and 2.4.0) were used, resulting in significant performance improvements.

Apart from the study mentioned earlier, there have been no recent comparative studies that showcased the performance difference between Apache Flink and Apache Storm.

Chapter 3

Stock Trading System

This chapter provides a examination of the system architecture that has been developed, beginning with a detailed overview of the architecture, followed by a description of the events that are propagated throughout the system, and finally going through the implementation details.

3.1 Architecture

The developed system is composed of the following parts: a messaging cluster, a supervisor server, a processing cluster, a metrics storage server, a metric processing server, a web server and a single page application. A visual representation of the aforementioned modules can be seen in Figure 3.1 below.

The messaging cluster is responsible for managing the flow of events within a system by keeping track of entities that produce events and entities that consume events. This messaging functionality is based on the popular publish-subscribe pattern relying on brokers as the primary coordinators of the messages. Apache Kafka is a distributed event streaming platform used to implement the messaging cluster [11]. Kafka's extensive support for stream processing frameworks, including Apache Flink, Apache Spark, and Apache Storm, showcases its technical suitability for implementing the messaging cluster. Additionally, Kafka's straightforward deployment process further justifies its technical suitability.

The supervisor server's responsibility is coordinating the messaging cluster and the processing cluster. It leverages Apache ZooKeeper, as both Apache Kafka and Apache Storm rely on this technology. Apache ZooKeeper is a distributed coordination service that provides a highly reliable and fault-tolerant infrastructure for distributed systems [12]. This necessary technology is required by the messaging cluster for internally coordinating various information like topics, partitions, consumer groups, and producer. On the other hand, the processing cluster, implemented using Apache Storm (which will be described in the following paragraph), depends on the supervisor server for coordinating topology metadata, fault tolerance mechanisms, and overall cluster supervision.

The processing cluster is another important component of the system and is responsible for processing market data that is introduced into the system by external entities. It relies on the messaging cluster to subscribe and consume market events followed by the producing of financial results. This thesis presents two implementations for the processing cluster: one with Apache Flink and another with Apache Storm. Apache Flink is a stream and batch processing framework designed for distributed, high-throughput, and fault-tolerant data processing [9]. It provides a unified programming model for building real-time streaming applications and large scale batch processing jobs. Similar to Apache Flink, Apache Storm is a distributed real-time computation framework designed for processing streaming data wh[10]. It provides a fault-tolerant and highly scalable platform for processing continuous streams of data in real-time. Both technologies were selected because of their suitability in event processing, their widespread applications in various real-life scenarios and the limited availability of extensive comparative analysis between Apache Flink and Apache Storm.

The metrics storage server fulfils the role of centralizing the metrics provided by the processing cluster. Graphite is used as the underlying technology, a metric storage and visualization system, offering the functionality to collect and store time-series data, such as gauges and histograms [13]. The choice of Graphite was motivated by its capability to easily integrate with Apache Flink and Apache Storm due to their already existing support for metrics reporting to Graphite.

The metric processing server has the role of retrieving data from the metrics storage server and presenting it in various graphical formats for comparative analysis. This functionality is achieved through Grafana, a data visualization and monitoring tool, specifically designed to offer a flexible and comprehensive platform for visualizing time-series data sourced from databases, monitoring systems, and cloud platforms [14]. The decision to adopt Grafana was driven by its functionality to generate interactive time-series graphs, facilitating real-time visualization, analysis, and providing

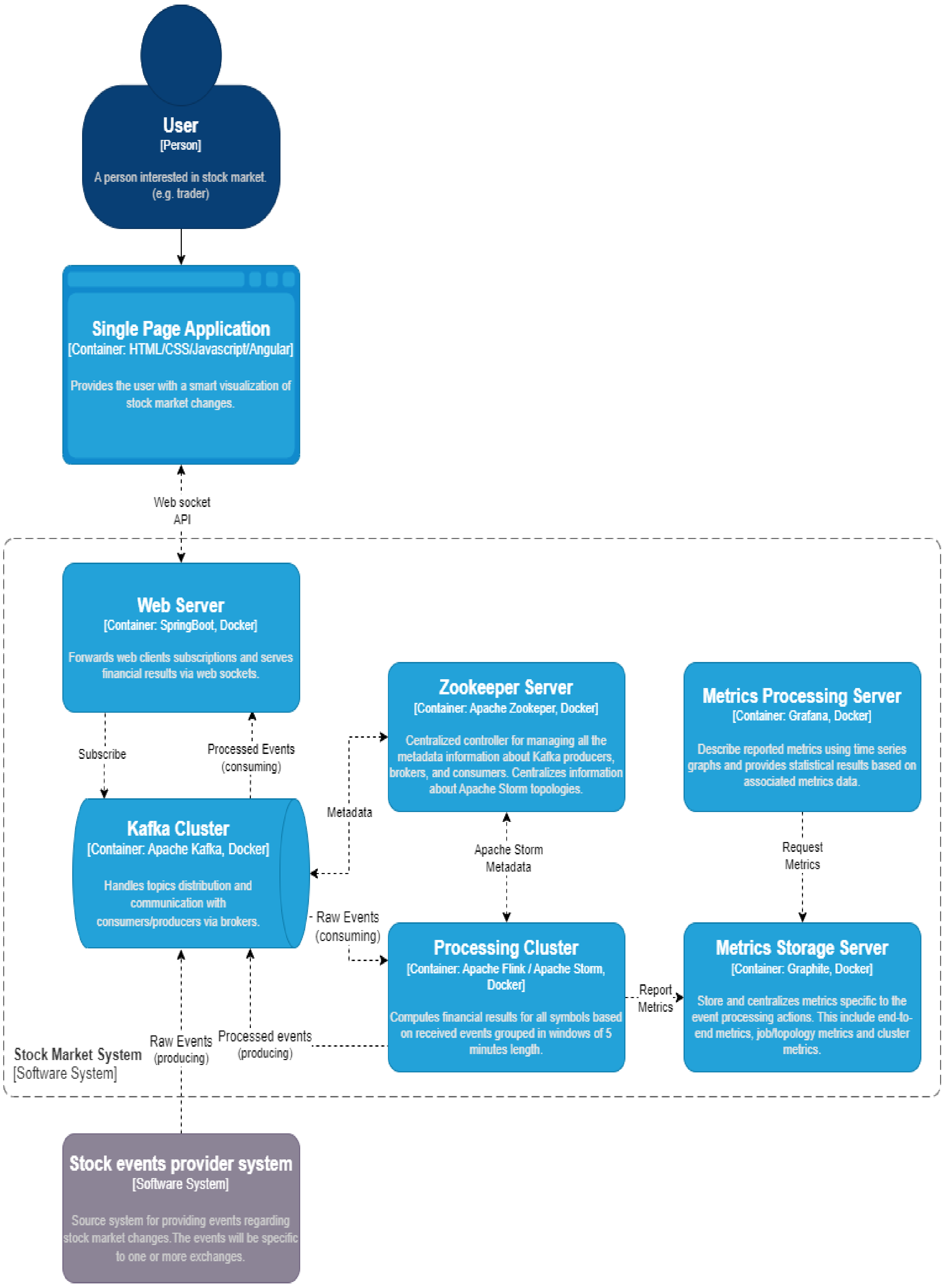

User
[Person]
A person interested in stock market. (e.g. trader)
Single Page Application
[Container: HTML/CSS/Javascript/Angular]
Provides the user with a smart visualization of stock market changes.
Web socket API
Web Server
[Container: SpringBoot, Docker]
Forwards web clients subscriptions and serves financial results via web sockets.
Subscribe
Processed Events (consuming)
Kafka Cluster
[Container: Apache Kafka, Docker]
Handles topics distribution and communication with consumers/producers via brokers.
Metadata
Zookeeper Server
[Container: Apache Zookeper, Docker]
Centralized controller for managing all the metadata information about Kafka producers, brokers, and consumers. Centralizes information about Apache Storm topologies.
Apache Storm Metadata
Metrics Processing Server
[Container: Grafana, Docker]
Describe reported metrics using time series graphs and provides statistical results based on associated metrics data.
Request Metrics
Raw Events (consuming)
Processing Cluster
[Container: Apache Flink / Apache Storm, Docker]
Computes financial results for all symbols based on received events grouped in windows of 5 minutes length.
Report Metrics
Metrics Storage Server
[Container: Graphite, Docker]
Store and centralizes metrics specific to the event processing actions. This include end-to-end metrics, job/topology metrics and cluster metrics.
Raw Events (producing)
Processed events (producing)
Stock Market System
[Software System]
Stock events provider system
[Software System]
Source system for providing events regarding stock market changes. The events will be specific to one or more exchanges.

Figure 3.1: C4 Container Diagram

statistical insights.

The web server role is in establishing an abstract communication channel between various web clients and the stock trading system using web sockets. This component is implemented using Spring Boot due to its straightforward and efficient deployment process, as well as it's popularity in handling such scenarios [15].

The single page application has the purpose of providing the end user with a friendly way of viewing the computed stock events produced by the processing cluster. The framework used for implementing this interface is Angular, and for the graphical display the library is utilized [16] [17].

3.2 Events

As briefly mentioned in the previous chapters, the data transmitted within the system is classified as events. These events are grouped under specific Kafka topics which producers utilize in order to append additional events and consumers in order to access the associated stream of events.

The system encompasses two types of events: input events (or raw market data) and output events (or processed market data). An input event refers to market data that is generated outside the system's boundaries, produced by a trusted external entity. This data is assumed to be accurate and reliable in terms of its correctness and integrity. This type of event has the following structure:

- a symbol which incorporates a unique identifier, either an equity identifier (e.g. Royal Dutch Shell, Siemens Healthineers etc.) or an index identifier (e.g. DAX) together with an associated exchange, such as Paris (FR), Amsterdam (NL) or Frankfurt (ETR). For instance, a possible symbol would be *RDS.NL*.
- a security type which specifies if the unique identifier found in the symbol is an equity identifier (E) or an index identifier (I).
- a price specific for this symbol (e.g. 121.940000). The currency associated with the price is considered irrelevant in the comparative analysis context.
- a trading time which represents time of last update with the *HH:mm:ss.SSS* format (e.g. 07:00:00.024). The update can be any sort of bid, ask or trade.

- a trading date which represents the date of the last update with the *yyyy-MM-dd* format (e.g. 2021-11-08).

This type of event is transmitted through the designated topic ”trade-data”, which is specifically utilized for managing input events received from producers. In order for the processing cluster to access the corresponding events within the topic, a subscribe action is required which involves registering and distributing a subscription across the Kafka brokers. Once the subscription is recorded, the brokers supply the processing cluster with any incoming input events.

The output event contains financial results obtained by the processing of input events. This type of data is generated by the processing cluster, as a result of the stream operations applied on the supplied events. An output event has the following structure:

- a symbol which has the same representation as the symbol encountered in input events.
- a security for which the type has the same representation as the security type encountered in input events.
- a price which has the same representation as the price encountered in input events.
- a EMA38 indicator which represents the computation of the exponential moving average with a smoothing factor of 38, within a specific window.
- a EMA100 indicator which represents the computation of the exponential moving average with a smoothing factor of 100, within a specific window.
- a SMA2 indicator which represents the computation of the simple moving average with a total periods of 2, within a specific window.
- a breakout pattern indicator which determines the existing pattern (bullish or bearish) deduced from the aforementioned indicators.
- a trading time which represents time of last update with the *HH:mm:ss.SSS* format (e.g. 07:04:45.024). It reflects the time of the last update within a specific window.

- a trading date which represents the date of the last update with the *yyyy-MM-dd* format (e.g. 2021-11-08). It reflects the date of the last update within a specific window.

With the varying interest of each output event produced, their designated topics are established based on the symbol of the event. For instance, when processing a stream of input events with the symbol *IUIFL.FR*, the resulting output event will be grouped under the topic corresponding to that symbol (IUIFL.FR). The method described offers flexibility by allowing consumers with specific requirements to avoid receiving irrelevant information.

Both input and output events have similar structure and both are serialized into JSON format, which is the format use for propagating the events within the system.

3.3 Implementation details

This section provides a description of the implementations in Apache Flink and Apache Storm, as well as their interaction with the events discussed in the previous section. Both implementations share a high degree of conceptual similarity, while the key difference lies in the execution manner of their respective code.

3.3.1 Apache Flink

In the Apache Flink implementation, the workload is structured and managed as a job, which consist of tasks distributed among the nodes in the processing cluster. The Job Manager node is responsible for submitting the job, while the Task Manager nodes controls the execution and resource allocation of the job.

The job designed to process the input events, which are consumed through the brokers, consists of the following operators: a Kafka source operator, a window operator, a reduce operator, a processing operator, and a Kafka sink operator. Figure Figure 3.2 provides a visual representation of the flow of events through the operators, as well as the dependencies among them.

The Kafka source operator holds the responsibility of establishing the connections between the processing framework and the Kafka cluster in order to retrieve input events whenever they are registered in the system. The retrieval process involves establishing a connection with a bootstrap broker, followed by subscribing to the ”trade-data" topic, and finally the assigning of a partition and offset for the framework by the brokers. Once this initial setup is complete, a data stream is generated which groups events by their event symbol (e.g RDS.NL) and is regularly updated whenever a new event is consumed. The source operator remaining actions are represented by the continuous events grouping and deserialization processes executed whenever a new input event is received. Periodic heartbeats signals are transmitted to the brokers in order to verify the status and health of the events consuming connection.

The window operator has the main responsibility of grouping input events in 5-minute tumbling windows. This grouping strategy involves extracting the timestamp from each event, calculated by converting the associated date and time of the last update to the number of milliseconds elapsed since the Unix epoch time. The obtained timestamps are utilized to determine the time boundaries of windows.

To ensure effective event processing, extracting the timestamp is insufficient due to the lack of a mechanism to advance the event based time. As such, a watermarking strategy is necessary to establish a reference point for the progression of time. When a new event is emitted, the watermark is updated based on the event's timestamp, assuming that no events with timestamps earlier than the watermark will arrive in the future. This assumption in real-world scenarios rarely holds true due to factors like network delays or occurrence of downtime in the events source systems. To address this, the implemented watermarking strategy incorporates a bounded out-of-order approach. This approach takes into consideration the possibility of events arriving out of order, but with a known maximum delay or bound. There is no universal guideline for determining the optimal maximum delay due to the diverse range of external factors affecting the arrival time of events. As a starting point, it is recommended to set a conventional value, which can be adjusted based on later observations and insights. For this implementation, the chosen delay value is 5 seconds.

The reduce operator purpose is to optimize memory usage and processing time by selecting only relevant events. Within a window, the event with the most recent timestamp is picked for computing per symbol indicators. Therefore, the reduce function compares the date and time of events and selects the most recent one among them within a window. Due to the group by key function applied in the Kafka source operator, the reducing is guaranteed to be conducted between events with the same symbol.

The processing operator is responsible for computing the per symbol indicators EMA38, EMA100, SMA2, and identifying breakout patterns within a window as described in Chapter 1. This computation occurs after the aggregation operator, guaranteeing that only relevant events are used as input for the processing operator. To effectively calculate these indicators, previous window results are necessary. Thus, the operator utilizes the in-memory state of the framework to persist indicator results per symbol, enabling their storage for later use.

The Kafka sink operator is responsible for serializing and publishing the acquired output events within the Kafka cluster. The operator dynamically selects the topic based on the symbol associated with each event. For example, an output event with the symbol ”RDS.NL” would be published within the messaging cluster under a topic with the same name.

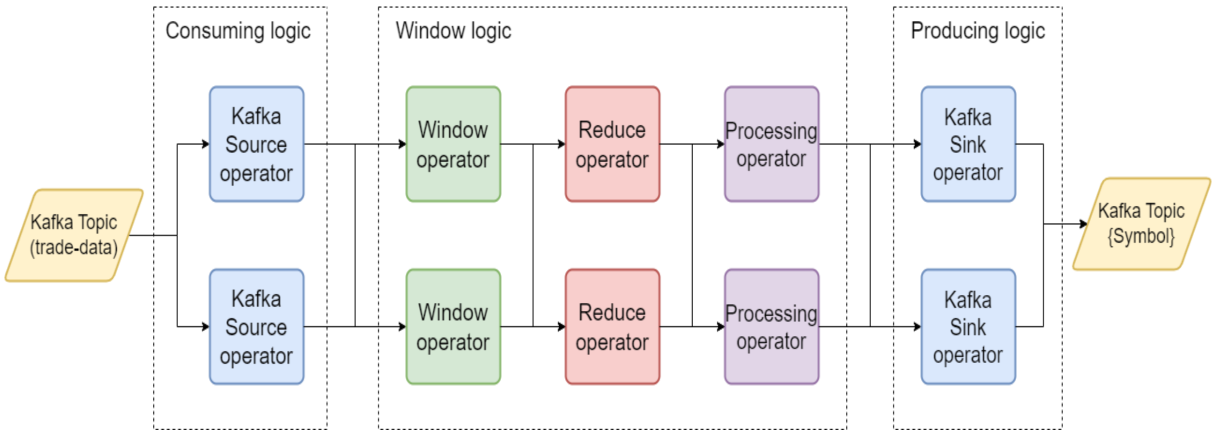


Figure 3.2: Apace Flink Dataflow Diagram

3.3.2 Apache Storm

The Apache Storm implementation functions by partitioning the workload into a topology, which consists of spouts and bolts distributed across the nodes in the processing cluster. The Nimbus node in Apache Storm is responsible for topology submission, whereas the Supervisor nodes manages task execution and resource allocation for the topology.

The topology designed to process the input events, which are consumed through the brokers, consists of the following operators: a Kafka spout, a window bolt, a processing bolt, and a Kafka bolt. Compared to the more granular components presented in the Apache Flink implementation, the Apache Storm implementation include a simpler set of components. Figure 3.3 provides a visual representation of the flow of events through the operators, as well as the dependencies among these operators.

Both the Kafka spout and bolt have a high degree of similarity with the Kafka source and sink discussed previously. They share a common abstraction and are implemented using the same Kafka library (kafka-clients). A notable difference between them is the Kafka sink's ability to automatically create topics if they do not already exist, a feature that is not supported in the Kafka bolt. In order to compensate this disadvantage, the existence of the topic is verified, and if needed, it is generated during the window bolt process.

Unlike the Flink implementation, in Storm there is only a single window bolt to handle all window related logic, due to Storm missing support of aggregation operations on events until the specified timer for the window ends. Therefore, the role of the window bolt is to eliminate irrelevant events (a task performed by the reduce operator in Flink), and subsequently transmit the remaining events to the processing bolt responsible for conducting indicator calculations. Both implementations share similar approaches in terms of the logic for extracting timestamps, the strategy for watermarking, and the in-memory approach for storing previous window results.

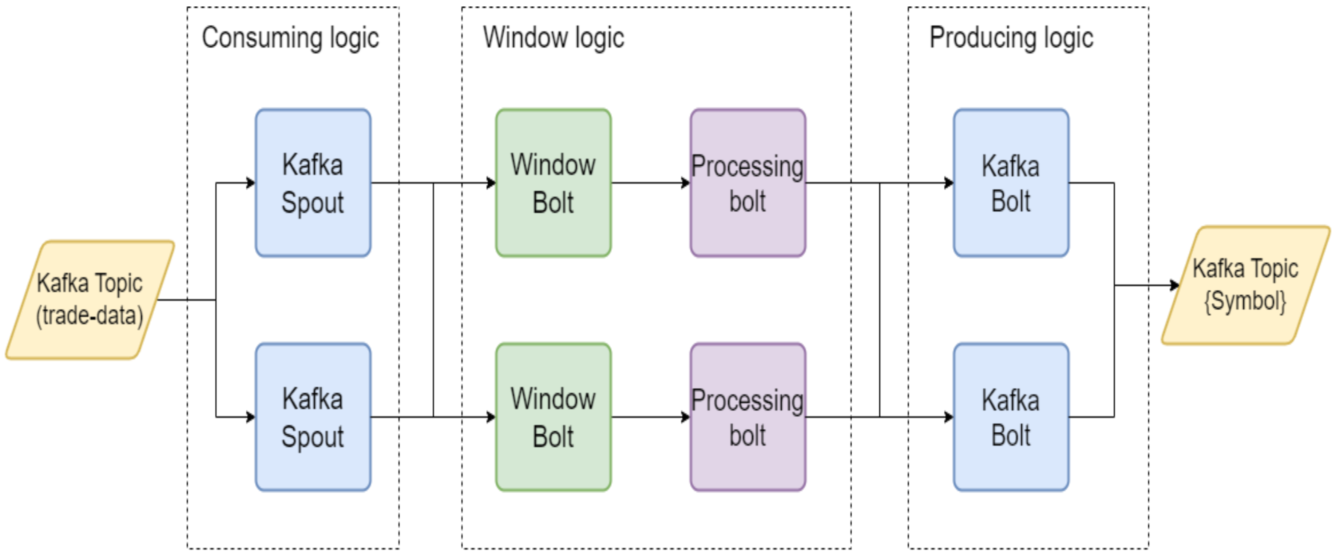


Figure 3.3: Apace Storm Dataflow Diagram

3.3.3 Web Server

As described in the architecture section, the web server role is in establishing an abstract communication channel between various web clients and the stock trading system using web sockets. Figure 3.4 provides a visual representation of the specific web server flow.

The application utilizes a multi-threaded approach and is categorized into two logical threads: one for handling web sockets and another for performing specific actions related to communicating with the Kafka cluster.

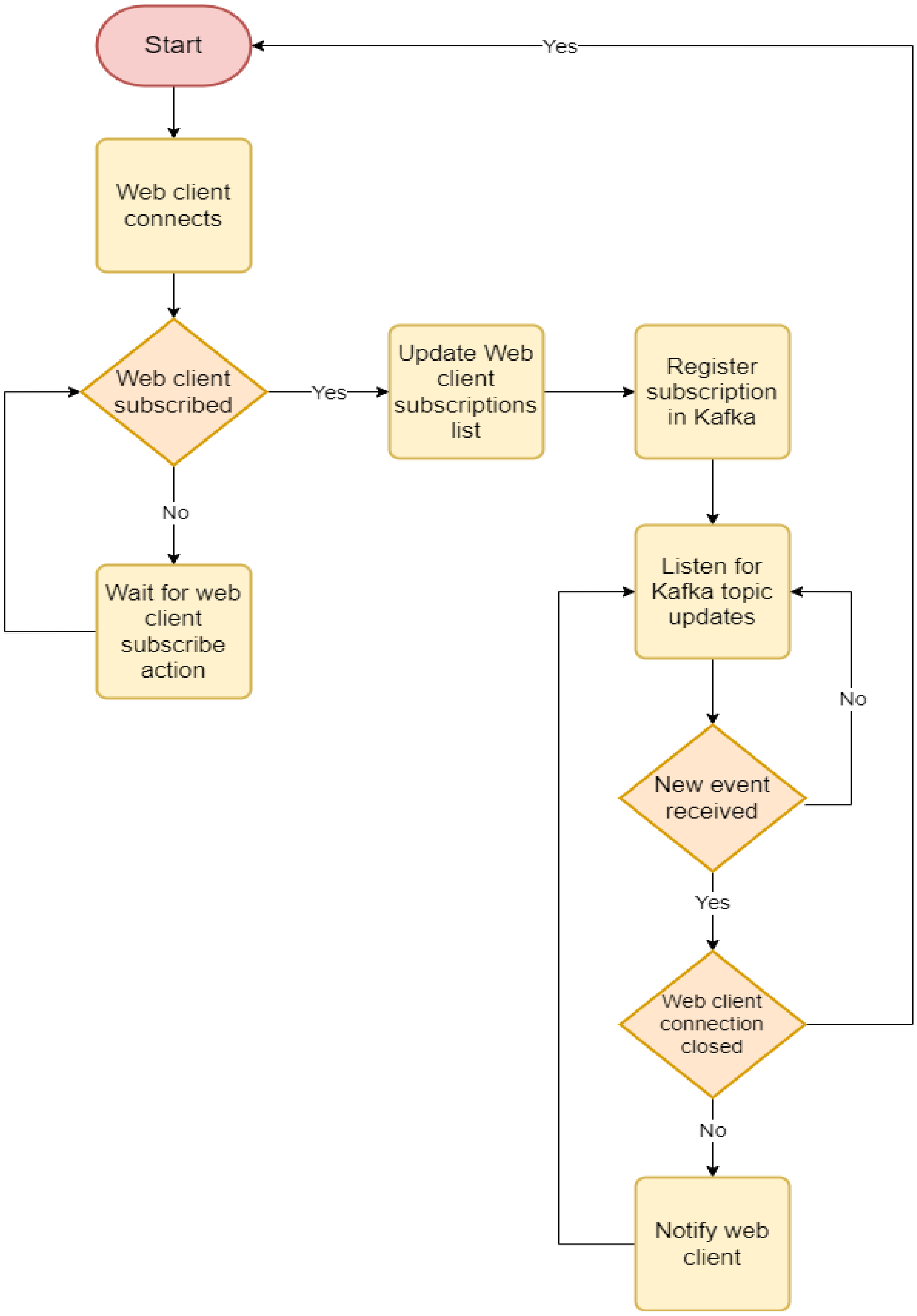
Once a new web socket session is established, the web socket handling mechanism creates a specific list to store the subscriptions for each session. Whenever a subscribe action is received from the web client, the web socket handling mechanism updates the subscription list and subscribes to the specified topic. When a web socket session is closed, its corresponding list is deleted, and any irrelevant Kafka subscriptions are removed.

On the other hand, the second thread is responsible for consuming and notifying the web client about newly received events. The Kafka cluster is queried every second in order to to fetch any updates related to the subscribed topics.

3.3.4 Single page application

As described in the architecture section, the purpose of the single page application is to present computed stock indicators to the end user in a friendly manner. It uses linear graphs to provide a clear overview of the indicators.

The application utilizes web sockets to establish a connection with the web server, and upon entering a symbol submitted by the user, a graph is displayed initially without any results. As soon as the web server sends computed stock indicators for the specified symbol, the graph is continuously updated in real time. Additionally, the user has the option to view results for other symbols, and a history of the symbols they have submitted along with relevant information is stored in a cookie for later sessions.



Start
Yes
Web client connects
Web client subscribed
Yes
Update Web client subscriptions list
Register subscription in Kafka
No
Wait for web client subscribe action
Listen for Kafka topic updates
No
New event received
Yes
Web client connection closed
No
Notify web client

Figure 3.4: Web server flowchart per client connection

Chapter 4

Comparative Analysis

This chapter presents a extensive description of the metrics analysed during the comparative analysis, along with the associated test scenarios, followed by the description of the benchmark setup used to conduct the aforementioned test scenarios and ending with a comprehensive description of the obtained results, including observations and justifications associated to these results.

4.1 Analysed metrics

The main goal of the comparative analysis was to assess the performance of Apache Flink and Apache Storm within the specified context described in the previous chapters. While this remained the main objective, examinations were also conducted on the fault tolerance mechanisms of each framework, as well as on the development and maintenance costs associated with their implementations. The evaluation of the latter was based on the challenges encountered during the system's implementation phase. In a detailed manner, the following metrics were analyzed:

- The average, minimum, and maximum latency of each technology and if there are any bottlenecks that could impact the latency.
- The average and maximum throughput that each technology can handle and if there are any bottlenecks that could impact the throughput.
- The tolerance of each technology to failures or errors.
- Ease of developing and maintaining applications using each technology by analyzing the challenges encountered during programming, integration, monitoring

and researching phases.

The analysis of latency and throughput was performed in three different types of testing scenarios.

4.1.1 End-to-end testing

The first scenario focused on end-to-end testing, where the evaluation began with the production of an input event in the Kafka cluster and concluded when the corresponding output event was produced into the cluster. This scenario was utilized due to its similarity with real-world situations, taking into account integration challenges and unexpected events that may appear. For the analysis of metrics in this scenario, the same solution utilized for producing the input events into the Kafka cluster was also used for consuming the corresponding output events. For both Flink and Storm, latency and throughput were analyzed in a similar way.

Latency was assessed by measuring the time difference between the production of an input event and the consumption of the corresponding output event. This evaluation was performed using a histogram metric type that is specially designed for analyzing latency.

Throughput was determined by querying the count of records consumed per second. This measurement was accomplished by utilizing a meter metric type that is specifically designed for analyzing throughput, which registers the number of records generated from the Kafka consumer polling mechanism.

4.1.2 Framework performance testing

The second scenario revolved around end-to-end testing of the frameworks. It involved the process of ingesting an input event into the framework and concluded when the corresponding output event was serialized and ready to be produced into the Kafka cluster. This specific scenario was chosen due to it's isolation from external factors that could potentially influence the performance testing of the framework.

In the context of Flink, latency was computed by subtracting the time at which the input event was deserialized by the Kafka source operator from the time at which the corresponding output event was processed by the Kafka Sink operator. In the case of Storm, latency was calculated by subtracting the time at which a tuple corresponding to the input event was created by the Kafka spout operator from the time at which the associated output event was processed by the Kafka bolt operator.

Flink throughput was determined by counting the number of events generated by the Kafka sink operator, while Storm's throughput was measured by counting the number of events produced by the Kafka bolt operator.

4.1.3 Window performance testing

In the third scenario, the focus was on the primary logic shared by both frameworks, specifically the window logic. Testing in this scenario started when the window's initialization process commenced and concluded when the processing was completed. Regarding Flink, window performance refers to the duration a record remains within the window, reduce, and process operators. On the other hand, in the context of Storm, window performance refers to the duration a record remains within the window and process bolts. By referencing it with the second scenario, an analyze of the proportion of performance attributed to this specific component was conducted relative to the overall framework performance.

In both Flink and Storm, latency was determined by calculating the difference between window's initialization time and completeness time. Specifically, it was determined by computing the average of the time spent by all the records within the window.

Both Flink and Storm compute throughput by counting the number of events emitted by the window operator.

4.2 Benchmark Setup

In order to have a complete testing environment with minimum external factors that may influence the results, every utilized technology was hosted in a Docker container. Containers are self-contained units that packages everything needed to run an application, such as source code, runtime environment, system tools, and libraries [18]. There are several advantages for this approach, including:

- A controlled environment for comparison provided by the isolation of technologies.
- A testing platform that is independent of any specific operating system.

- A flexible resource management environment, due to the possibility of assigning each container a different quantity of memory and CPU resources.
- A feature (Docker Compose) that enables the creation of multi-container environments, making the testing environment portable across various machines for testing with different specifications.

The containers communicated with each other using docker custom created internal network called ”stock-network”.

4.2.1 Parameters

The testing scenarios described in the previous section were conducted using a 64x Windows machine equipped with an Intel(R) Core(TM)i5-8250U CPU. This machine has 4 cores and a base speed of 1.60GHz, along with 16 GB of RAM. No resource limitations were imposed on the Docker containers or the Docker engine.

Different values were picked for the parallelism degree and buffer size parameters in both frameworks. Given the presence of a 4-core CPU, the parallelism degree values were ranging from 1 to 4. The importance of experimenting with different level of parallelism comes from the fact that a framework may achieve sufficient processing power with a parallelism hint of 1. In this case, increasing the parallelism degree would introduce unnecessary complexity to the system by having more instances of an operator than needed. Another consideration is that once a high enough level of parallelism is chosen that effectively handles the processing load, the parallelism can be easily increased as the system encounters periods of high data traffic.

In the Flink cluster setup, a single job manager and one task manager were deployed. The number of tasks allocated to the task manager was determined by equally matching the parallelism hint used for the job. This configuration was primarily chosen based on observations obtained from the test scenarios and recommendations found in the documentation, which focuses on optimal configurations for a single machine. Multiple task managers or more allocated tasks than parallelism hint led to a decrease of performance due to the unjustified addition of complexity to the cluster. The testing scenarios were conducted using Flink version 1.16.1.

The Storm cluster configuration involved deploying a single nimbus node and one supervisor node. For each topology, only one worker was allocated on the supervisor node. Similar to the Flink setup, this configuration is considered optimal for a single machine deployment, based on observed performance and documentation recommendations. The testing scenarios were conducted using Storm version 2.4.0.

In all testing scenarios, both the Kafka source and Kafka bolt are initialized with a parallelism hint of 1. This decision was made based on the observation that there were no significant difference between the numbers of ingested events within the framework when using more instances.

As detailed in Chapter 3, the recorded metrics were sent to the Graphite server. Both frameworks established the connection with the metrics storage server through the internal configuration yaml file. The metrics were reported every 30 seconds. There is an exception for Storm cluster metrics, which were stored in a csv file instead, as there is no support for reporting these specific metrics to Graphite.

4.2.2 Data set

The testing scenarios were conducted using a data set consisting of real stock market data collected by Infront Financial Technology from 08.11.2021 to 14.11.2021. This data set, obtained from Zenodo[1], has a size of approximately 25GB. A pre-processing step was performed to eliminate irrelevant events that lacked price, time, or date values. As a result of this pre-processing, the data set was reduced to approximately 4GB.

On the testing machine, the maximum achievable data production rate was approximately 65000 events/sec with a total approximation of 2574000000 events produced for each trading day.

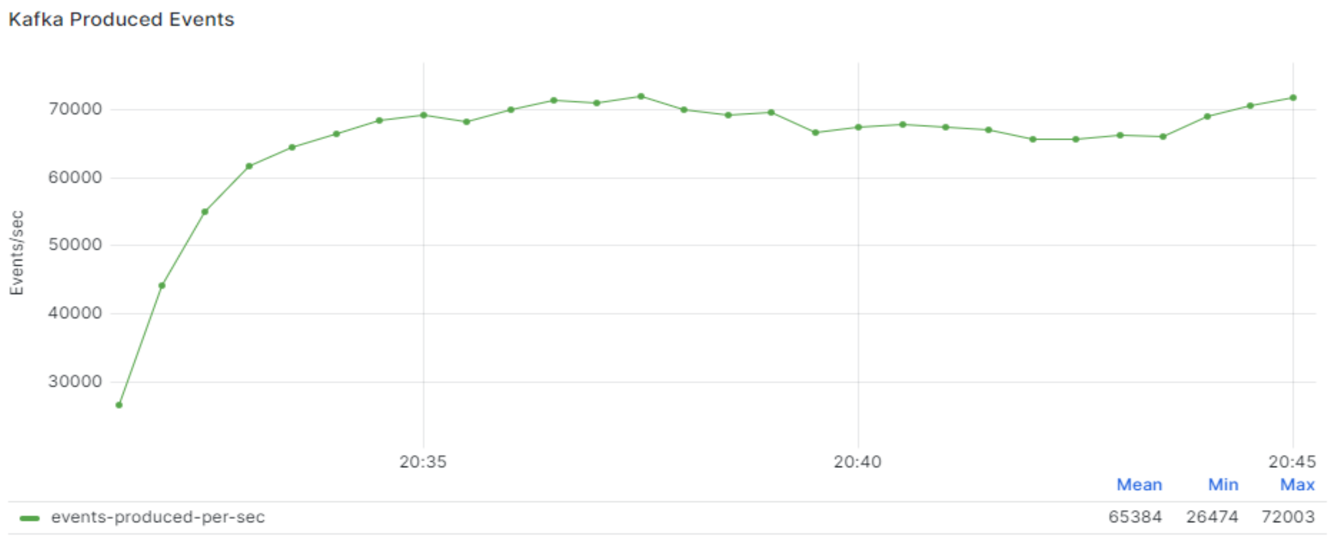


Figure 4.1: Kafka Produced Events

[1]https://zenodo.org/record/6382482

4.3 Comparison results

As described in the previous section, testing scenarios were conducted with different framework configurations in order to evaluate latency, throughput and fault tolerance. The development and maintenance analysis was based on the encountered challenges.

4.3.1 Latency

For the end-to-end latency tests, both frameworks were evaluated using a parallelism hint of 1, meaning that all operators were limited to utilizing only one core of the CPU. The results indicated that the Flink framework had a significantly lower latency compared to its counterpart, Storm. Storm initially had an acceptable latency, but its lower processing efficiency prevented it from serving output events as quickly as input events were being produced. These observations were derived through the analysis presented below in Figure 4.2 and Figure 4.4.

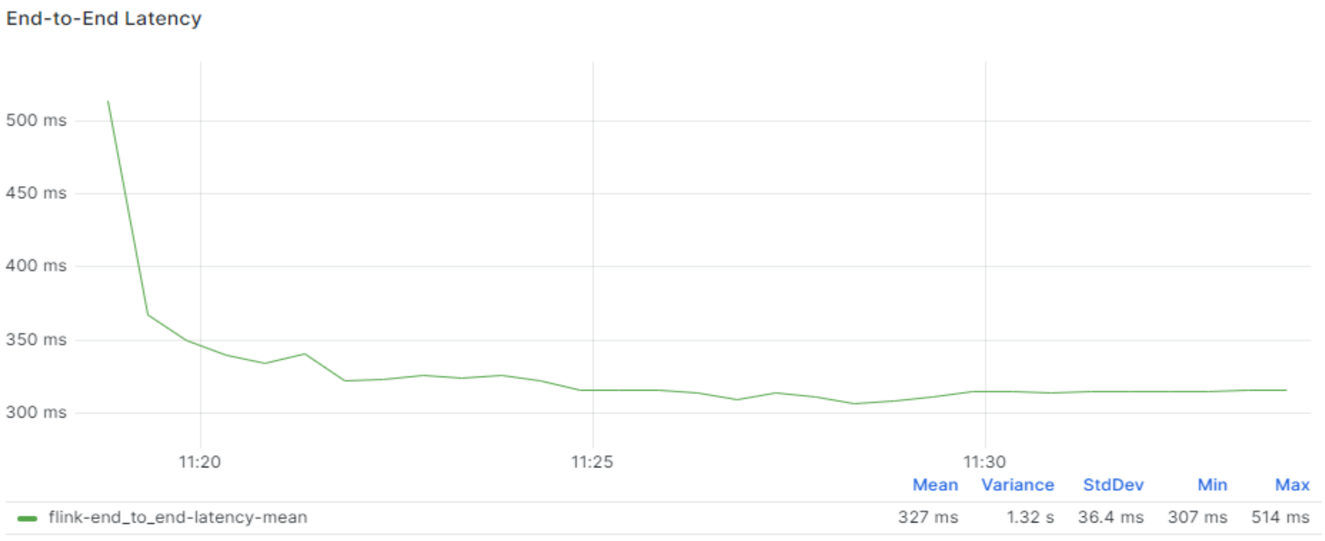
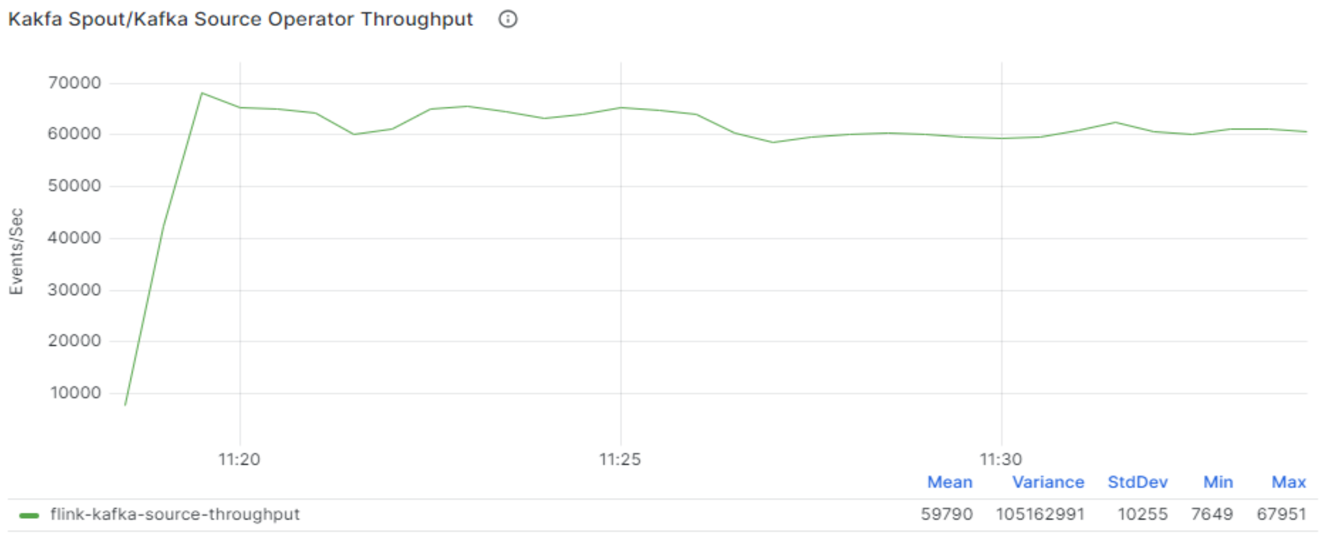


Figure 4.2: Flink End-to-End latency - parallelism hint 1

In the case of Flink, the average latency obtained was 327ms, with a minimum of 307ms and a maximum of 514ms. The initial higher latency in Flink was due to the JVM warm-up process for the operators. Once this warm-up process was completed, the average latency stabilized over time. Another significant observation from the tests was that the processing rate of events in Flink surpassed the rate at which events are produced. As an implication, the window operator buffer rarely filled up. This observation was deduced by analyzing Figure 4.3, which illustrates an average of around

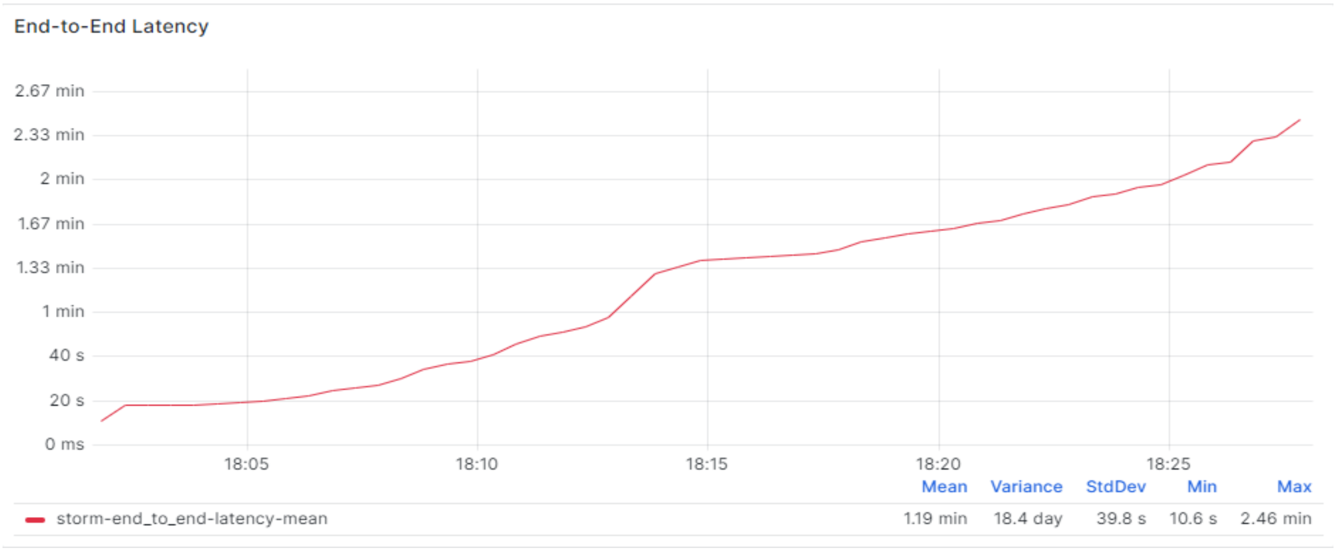
60000 events per second. Although this appears lower than the production rate of 65000 events per second, it can be attributed to the lower values observed during the initial stages of processing. By excluding the initial part, the average processing rate reached approximately 65000 events per second.



	Mean	Variance	StdDev	Min	Max
🟩 flink-kafka-source-throughput	59790	105162991	10255	7649	67951

Figure 4.3: Flink Kafka source - parallelism hint 1

In the case of Storm, the average latency obtained was 1.19min, with a minimum of 10.6s and a maximum of 2.46min. Upon examining Figure 4.4, it becomes evident that Storm is unable to maintain an acceptable processing rate relative to the production rate.



	Mean	Variance	StdDev	Min	Max
🟥 storm-end_to_end-latency-mean	1.19 min	18.4 day	39.8 s	10.6 s	2.46 min

Figure 4.4: Storm End-to-End latency - parallelism hint 1

For Flink specific framework testing and window testing, it was observed that the window processing operators accounted for the majority of the latency, while the consuming and producing operators contributed to a smaller portion of the overall latency. These findings are based on the analysis presented below in Figure 4.5 and Figure 4.6.

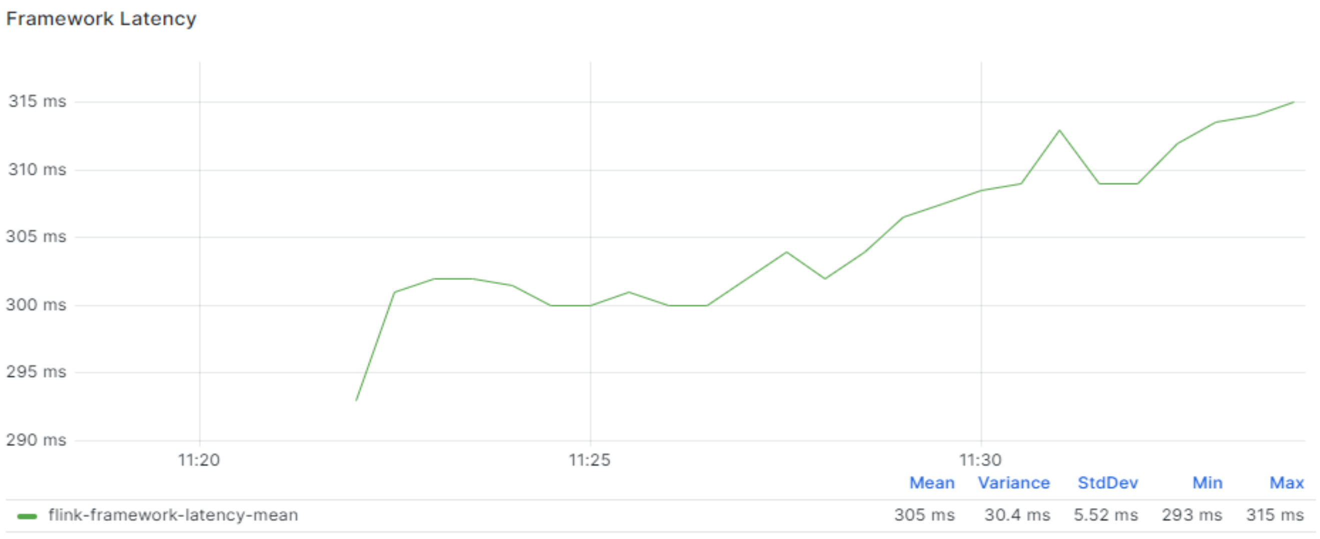


Figure 4.5: Flink Framework latency - parallelism hint 1

In the case of Storm framework testing and window testing, similar observations can be made as in the case of Flink. The window processing operators were identified as the primary source of latency, while the consuming and producing operators contributed to a smaller portion of the overall latency.

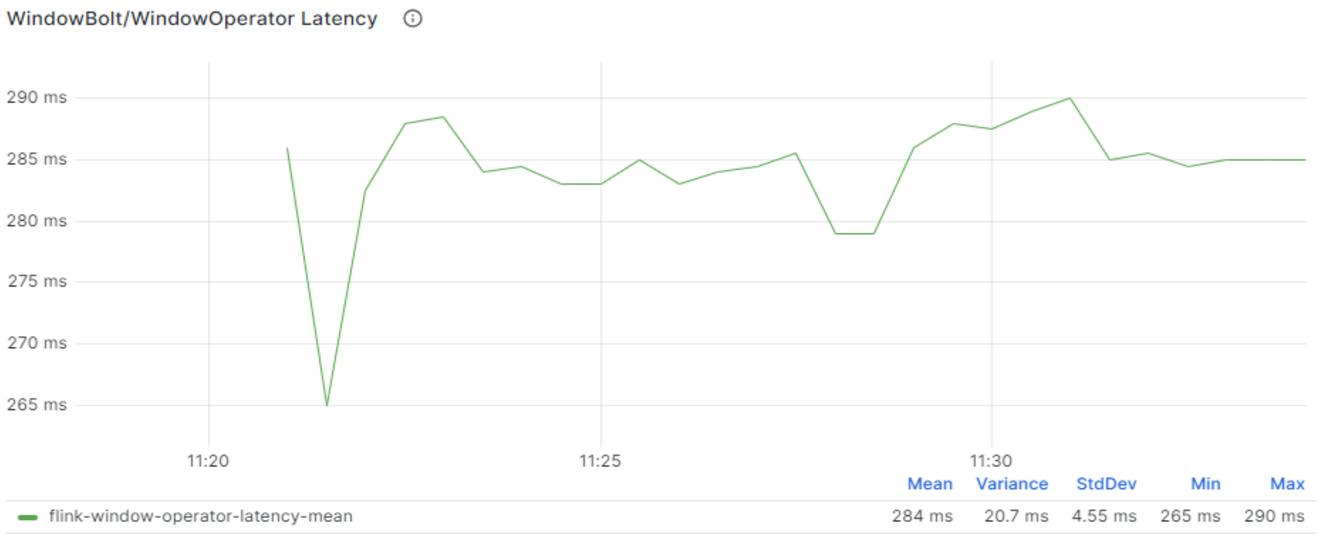


Figure 4.6: Flink Window latency - parallelism hint 1

The latency tests conducted on the optimal configurations for both frameworks with a parallelism hint of 4 had varying impacts. In the case of Flink, the performance

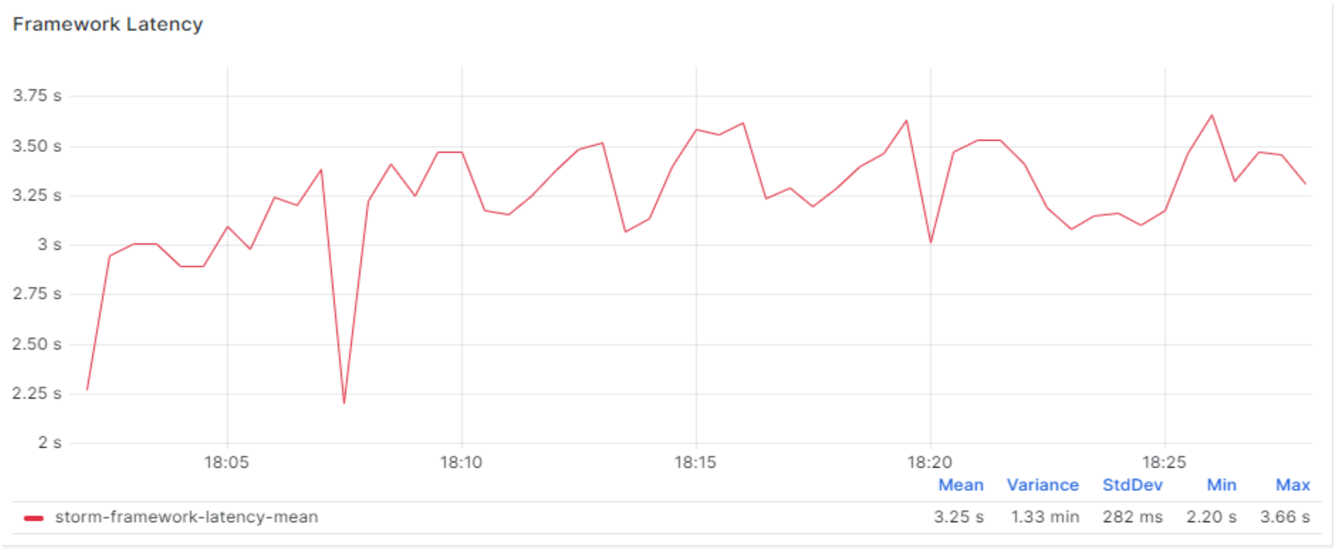


Figure 4.7: Storm Framework latency - parallelism hint 1

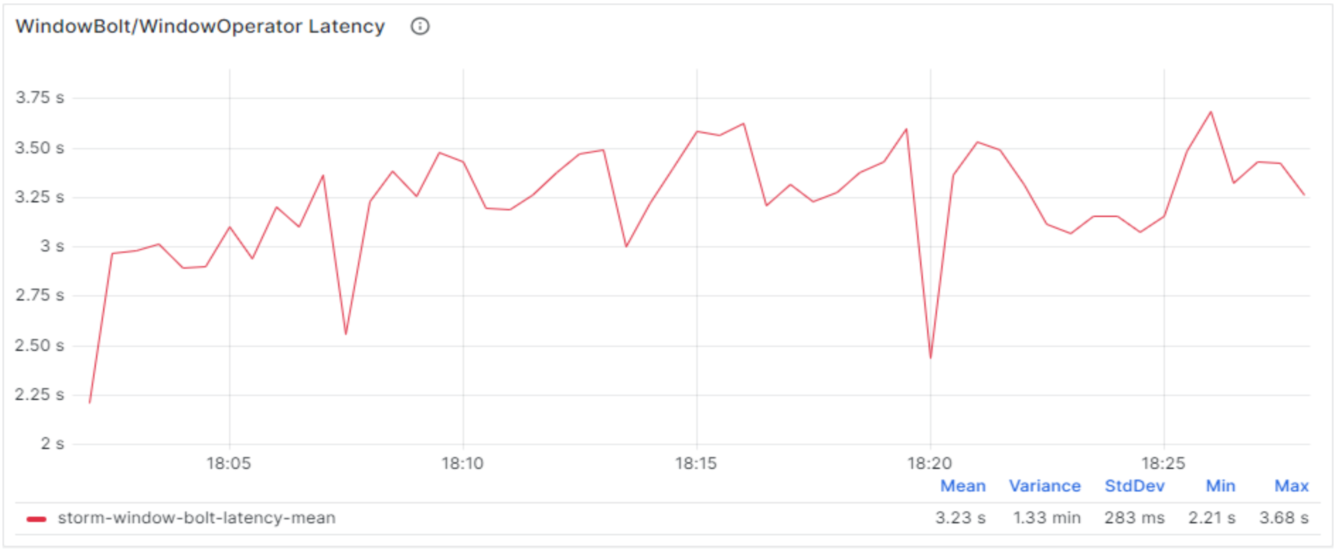


Figure 4.8: Storm Window latency - parallelism hint 1

results were not significantly different due to the low latency already achieved with a parallelism hint of 1. In contrast, the latency test results for Storm were notably different and showed significant increase in performance. The end-to-end latency for both frameworks can be seen in below Figure 4.9 and Figure 4.10.

In the case of Flink, the average latency achieved was 301ms, with a minimum of 285ms and a maximum of 309ms. A notable observation is that the source of the initial high latency values differs from the scenario with a parallelism hint of 1. This difference arose due to the distribution of load to multiple operator instances resulting in a faster processing time of events.

In the case of Storm, a notable observation is that while the latency initially in-

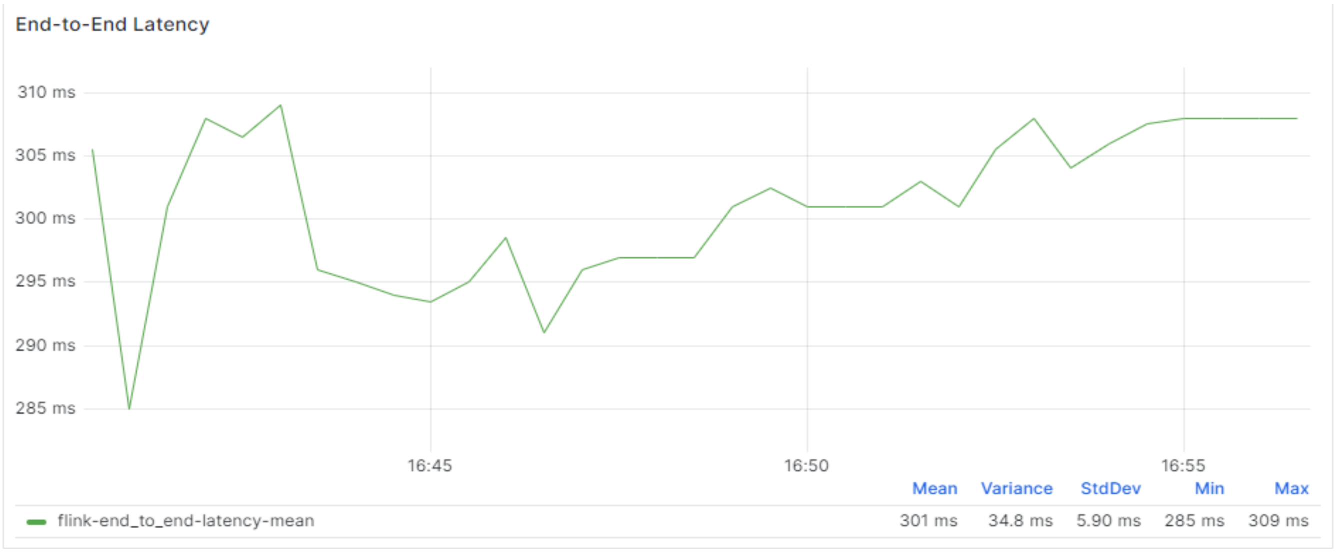


Figure 4.9: Flink End-to-End latency - parallelism hint 4

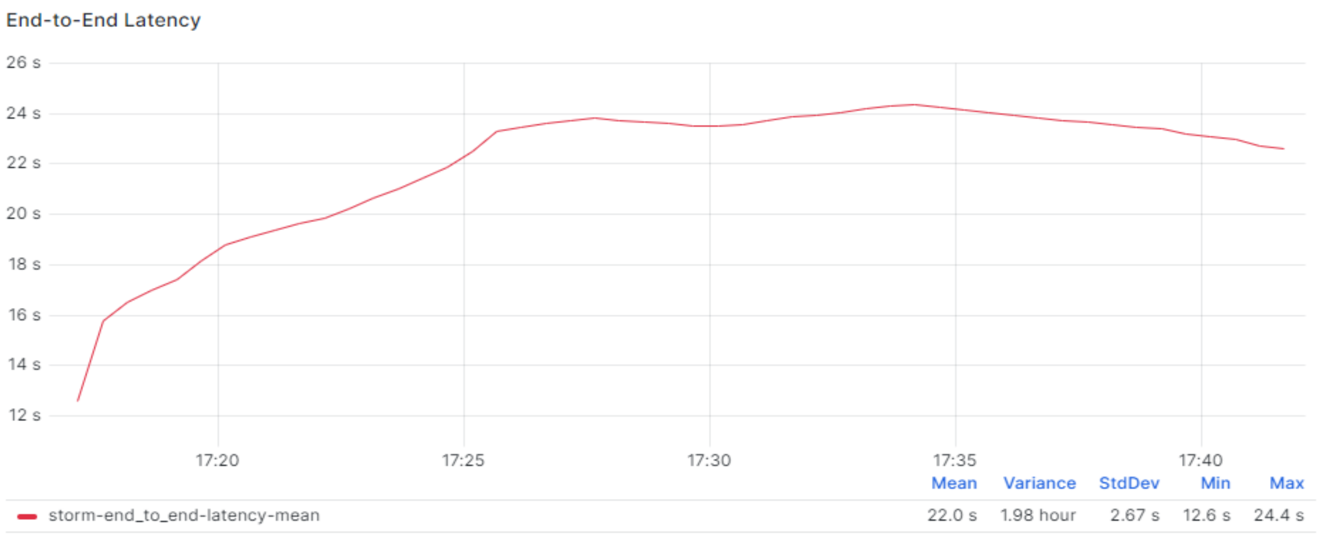


Figure 4.10: Storm End-to-End latency - parallelism hint 4

creased during the initial stages of processing, it gradually stabilized over time. Consequently, with a parallelism hint of 4, Storm achieved a processing rate that was sufficiently high to keep up with the production rate. Additionally, as shown in Figure 4.10, a downward trend can be observed towards the later stages of processing. This trend can be attributed to the effective load distribution across the four instances of the processing bolt, which proves to be vital in establishing Storm as a viable solution for these processing tasks. At the conclusion of the testing phase, Storm achieved a significant improvement in performance with an average latency of 22.0s, with a minimum of 12.6ms and a maximum of 24.4s.

In the case of framework testing, Storm achieved significant increase in performance with an average latency of 2.65s, with a minimum of 2.26s and a maximum of 3.21s. The same performance increase was also for window latency, achieving an average latency of 2.63s, with a minimum of 2.19s and a maximum of 3.18s. These results can be visualized in Figure 4.11 and Figure 4.12.

These figures also highlight four scenarios where both window and framework testing recorded minimum latency values. The significant difference in these values can be attributed to the nature of the data being analyzed. Specifically, the majority of stock market events were registered between 07:00:00 and 18:00:00 each trading day. In contrast, the quantity of registered events between 18:00:00 and 07:00:00 the following day is significantly lower. The decreased window resources required for buffering and processing events during the periods with fewer events led to a lower latency. Both of the aforementioned figures illustrate that the processed events correspond to a span of four days of financial events.

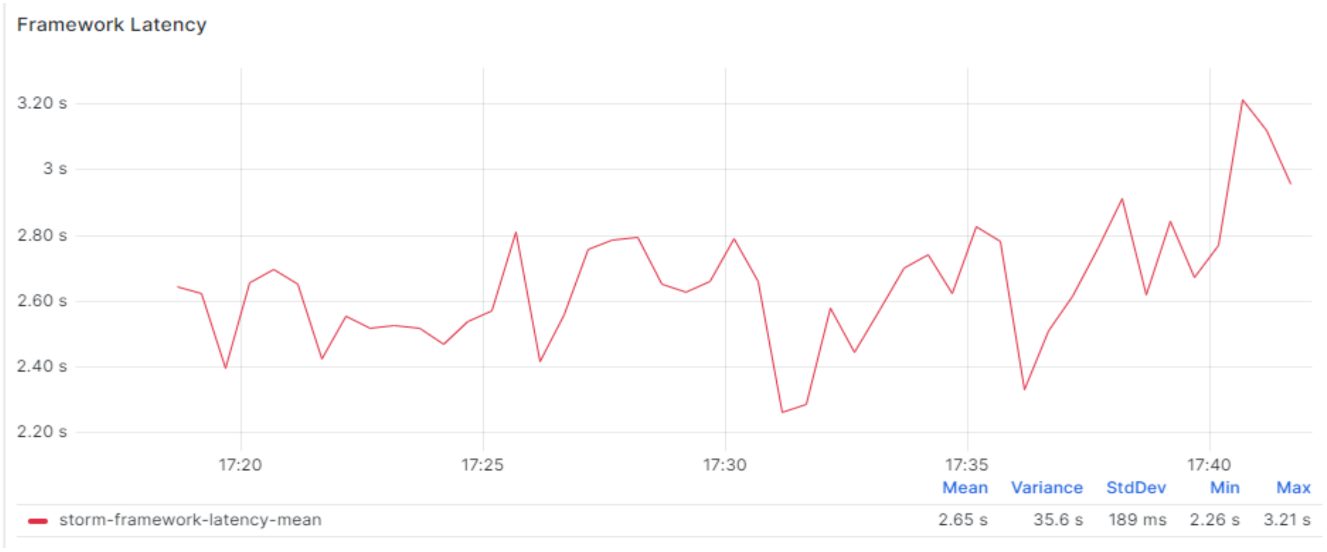


Figure 4.11: Storm Framework latency - parallelism hint 4

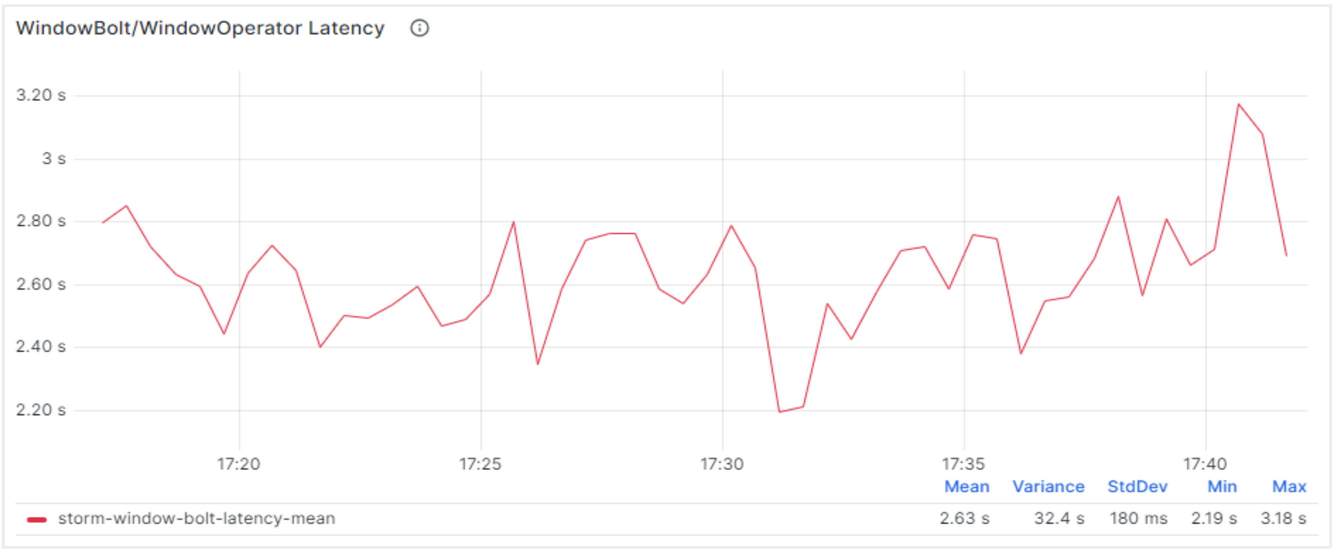


Figure 4.12: Storm Window latency - parallelism hint 4

In order to conduct the analysis of Storm latency, adjustments were made to the Kafka topics retention policy to accommodate the lower event processing rate of Storm. Specifically, the retention size policy was increased from 1GB to 2.4GB. A separate test was conducted using the initial recommended retention size policy, which led to Storm's significantly lower latency. However, this came at the cost of data consistency as some older events were dropped. These observations are based on the analysis of Figure 4.13, which visualizes the four instances where Kafka records were deleted due to exceeding the retention size. As a consequence, Storm experienced four drops in latency.

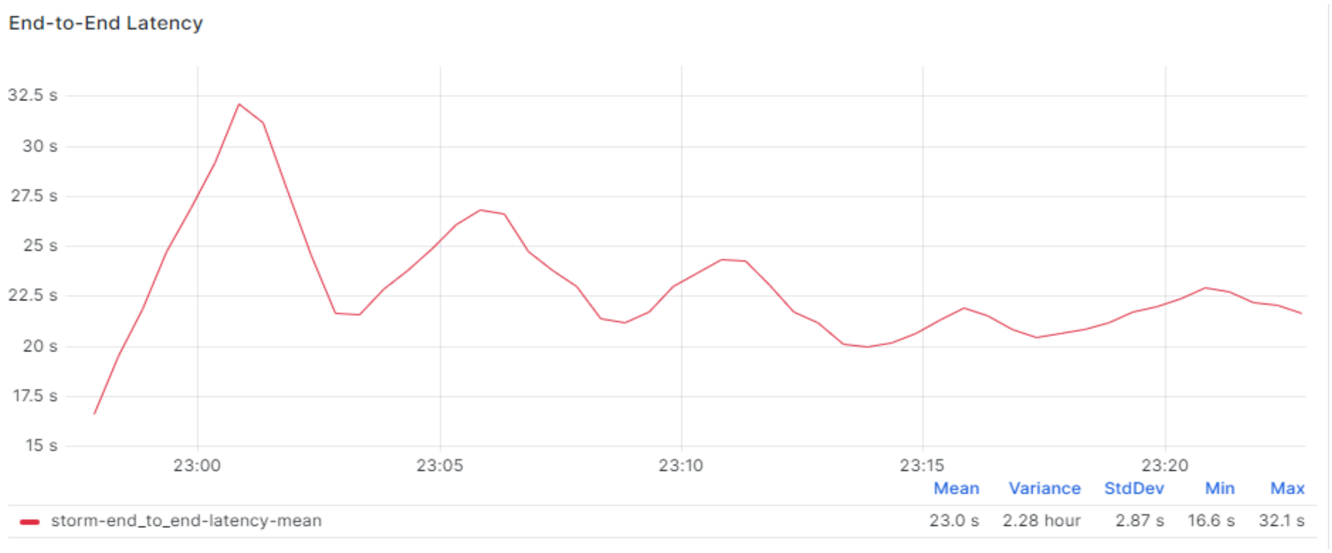


Figure 4.13: Storm End-to-End latency - retention size policy

The latency testing results obtained for different parallelism hints are presented in Table 4.1 and Table 4.2, providing a detailed visualization of the latency measurements.

Paralellism=1	average	min	max
End-to-end	327ms	307ms	514ms
Framework	305ms	293ms	315ms
Window	284ms	265ms	290ms

Paralellism=2	average	min	max
End-to-end	318ms	299ms	559ms
Framework	280ms	270ms	285ms
Window	215ms	207ms	223ms

Paralellism=3	average	min	max
End-to-end	308ms	300ms	384ms
Framework	275ms	268ms	293ms
Window	212ms	203ms	225ms

Paralellism=4	average	min	max
End-to-end	301ms	285ms	309ms
Framework	271ms	267ms	287ms
Window	209ms	200ms	217ms

Table 4.1: Apache Flink latency measurements

Paralellism=1	average	min	max
End-to-end	1.19min	10.6s	2.46min
Framework	3.25s	2.20s	3.66s
Window	3.23s	2.21s	3.68s

Paralellism=2	average	min	max
End-to-end	58.9s	12.6s	1.77min
Framework	2.88s	1.45s	3.53s
Window	2.86s	1.43s	3.34s

Paralellism=3	average	min	max
End-to-end	32.0s	12.67s	34.4s
Framework	2.75s	1.36s	3.31s
Window	2.78s	1.35	3.33s

Paralellism=4	average	min	max
End-to-end	22.0s	12.6s	24.4s
Framework	2.65s	2.26s	3.21s
Window	2.63s	2.19s	3.18s

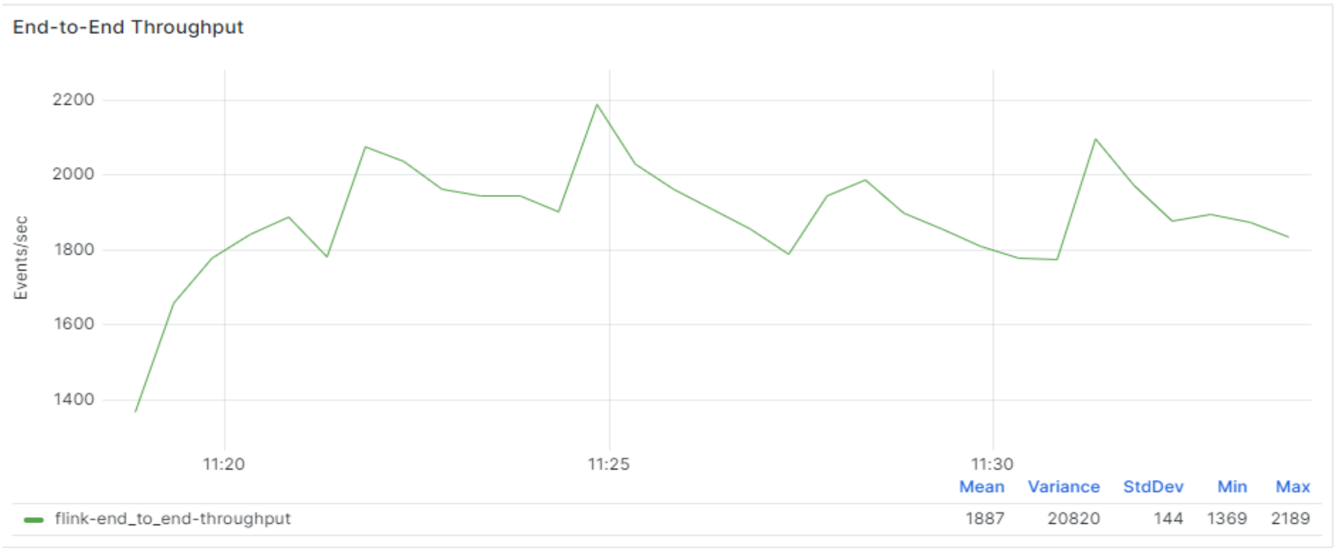
Table 4.2: Apache Storm latency measurements

Based on the examination of the aforementioned tables, it can be deduced that Flink experiences an increase in latency as the parallelism hint increases. In contrast, by analyzing Storm's latency values, it can be inferred that starting from a parallelism hint of 3, Storm demonstrates sufficient processing power to match the production rate and can be considered an acceptable technology for this type of processing. This notable improvement can be attributed to the close values of the average latency and maximum latency values. However, it falls short of matching Flink's performance in terms of processing rate.

In both frameworks, setting the parallelism hint to match the number of CPU cores yields the lowest latency.

4.3.2 Throughput

For the end-to-end throughput tests, similar to the latency testing, both frameworks were evaluated using different values for the parallelism hint. In the case of parallelism hint of 1, Flink had an considerable higher throughput. In the case of Flink, the average throughput achieved was 1887 events/sec, with a minimum of 1369 events/sec and a maximum of 2189 events/sec. Storm achieved an average throughput of 970 events/sec, with a minimum of 219 events/sec and a maximum of 1073 events/sec. These observation were derived by analyzing below Figure 4.14 and Figure 4.15.



	Mean	Variance	StdDev	Min	Max
flink-end_to_end-throughput	1887	20820	144	1369	2189

Figure 4.14: Flink End-to-End Throughput - parallelism hint 1

For Flink specific framework testing, the average throughput achieved was of 1871 events/sec, with a minimum of 1596 events/sec and a maximum of 2269 events/sec. In the case of Storm framework testing, the results showcased an average throughput of 1014 events/sec, with a minimum of 883 events/sec and a maximum of 1167 events/sec. These results were derived by analyzing below Figure 4.16 and Figure 4.17.

For Flink specific window testing, the average throughput achieved was of 1866 events/sec, with a minimum of 350 events/sec and a maximum of 2278 events/sec. In the case of Storm window testing, the results showcased an average throughput of 1019

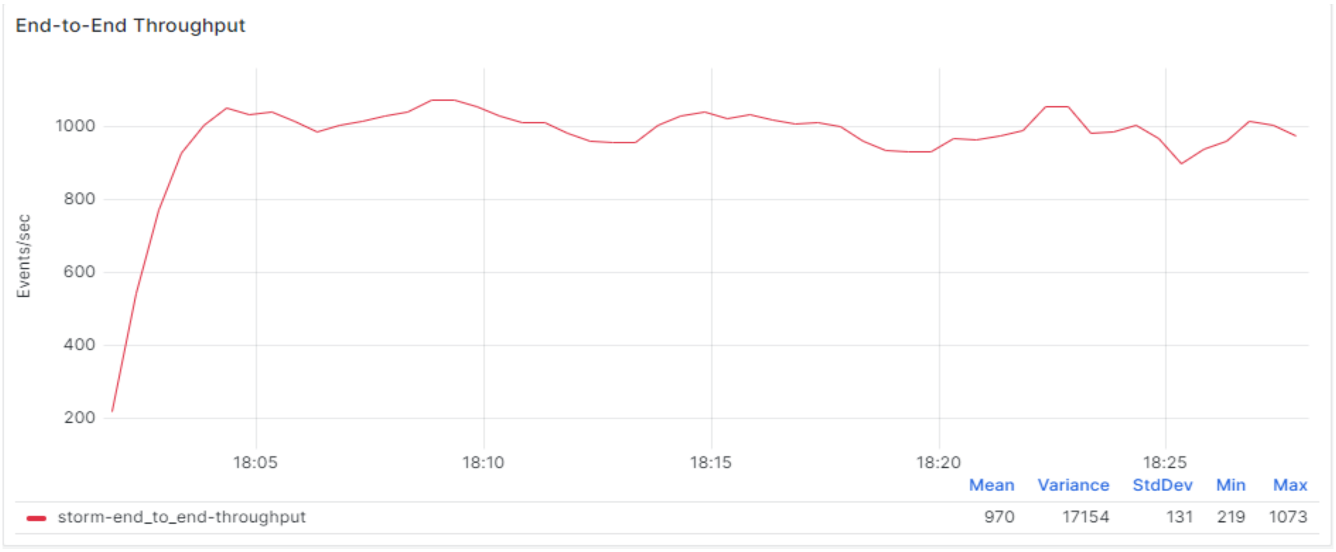


Figure 4.15: Storm End-to-End Throughput - parallelism hint 1

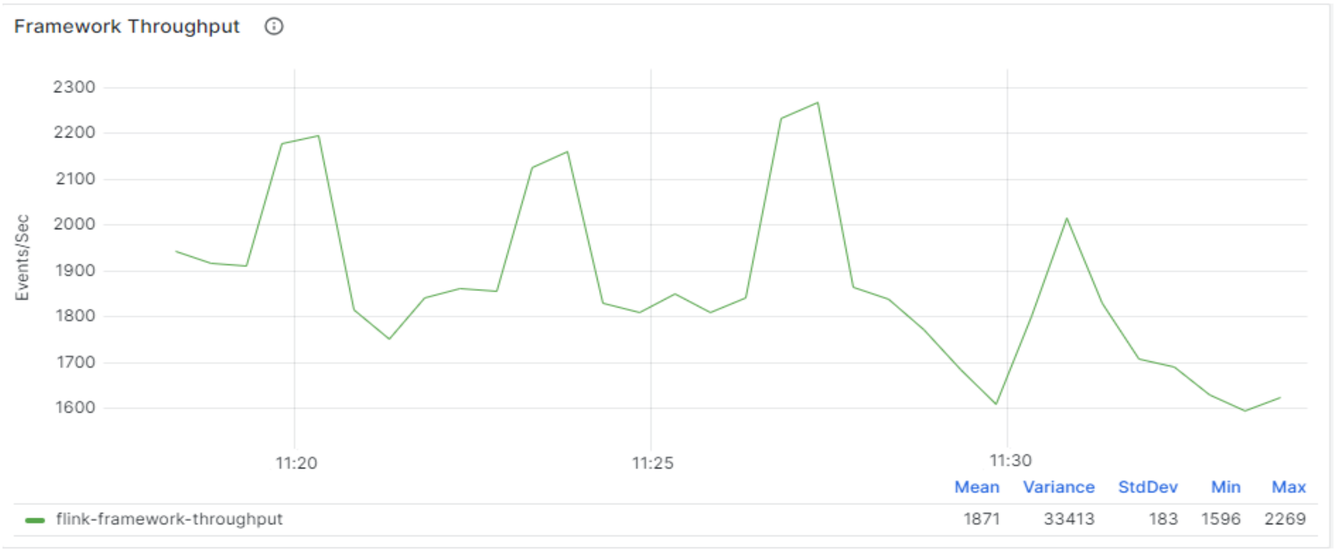


Figure 4.16: Flink Framework Throughput - parallelism hint 1

events/sec, with a minimum of 867 events/sec and a maximum of 1383 events/sec. These results were derived by analyzing below Figure 4.18 and Figure 4.19.

The throughput tests conducted on the optimal configurations for both frameworks with a parallelism hint of 4 had varying impacts. When examining Flink's performance, it can be observed that the results were not notably different when compared to the results obtained with a parallelism hint of 1. This can be attributed to the fact that Flink already achieved a sufficiently high processing power with a parallelism of 1, which was evident in the latency testing as well. On the other hand, Storm achieved a higher throughput by leveraging the improved processing power and employing four instances of the Kafka bolt for generating the output events. The end-to-end through-

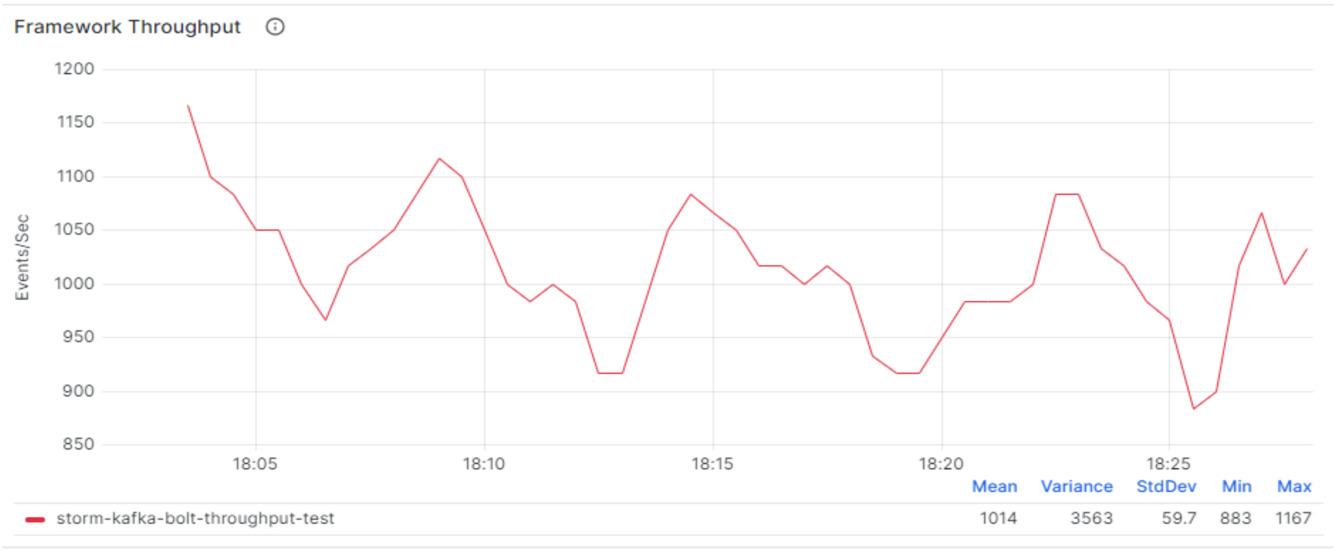


Figure 4.17: Storm Framework Throughput - parallelism hint 1

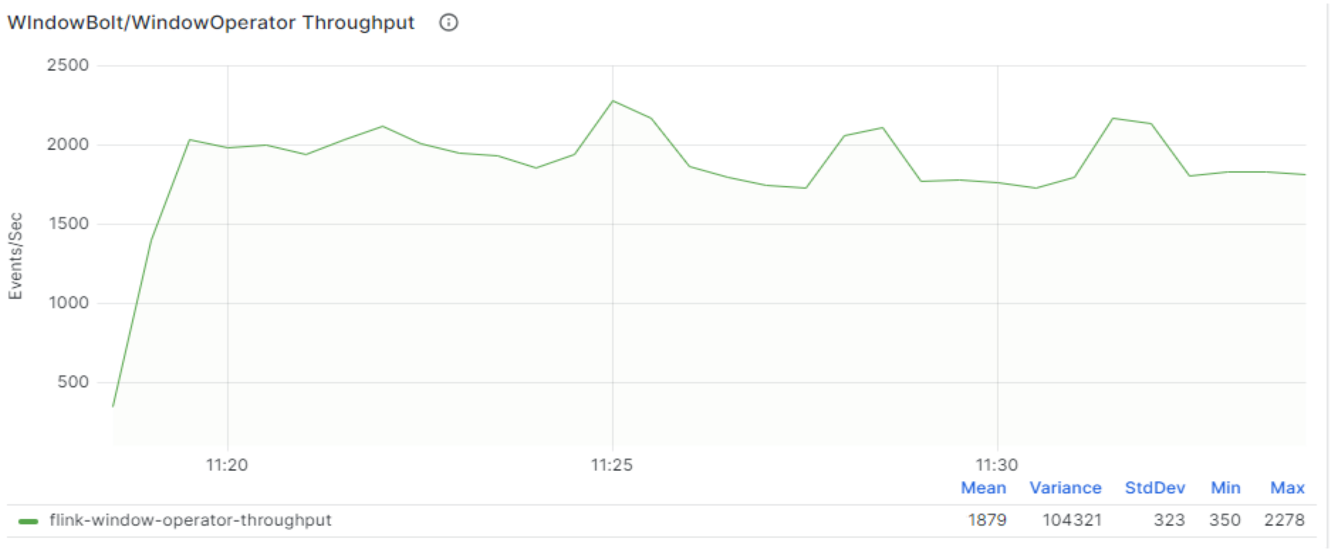


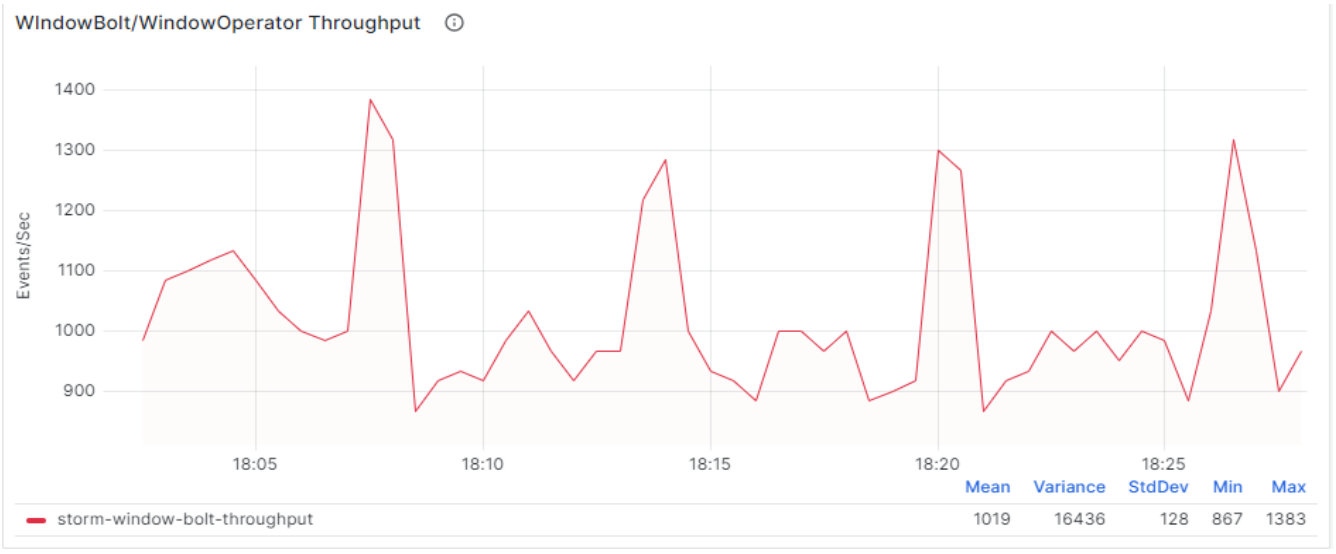
Figure 4.18: Flink Window Throughput - parallelism hint 1

put for both frameworks can be observed in below Figure 4.20 and Figure 4.21.

The four scenarios described earlier in the latency section, which involved extreme shifts of values in latency, can also be observed when considering throughput by examining for each framework the maximum values.

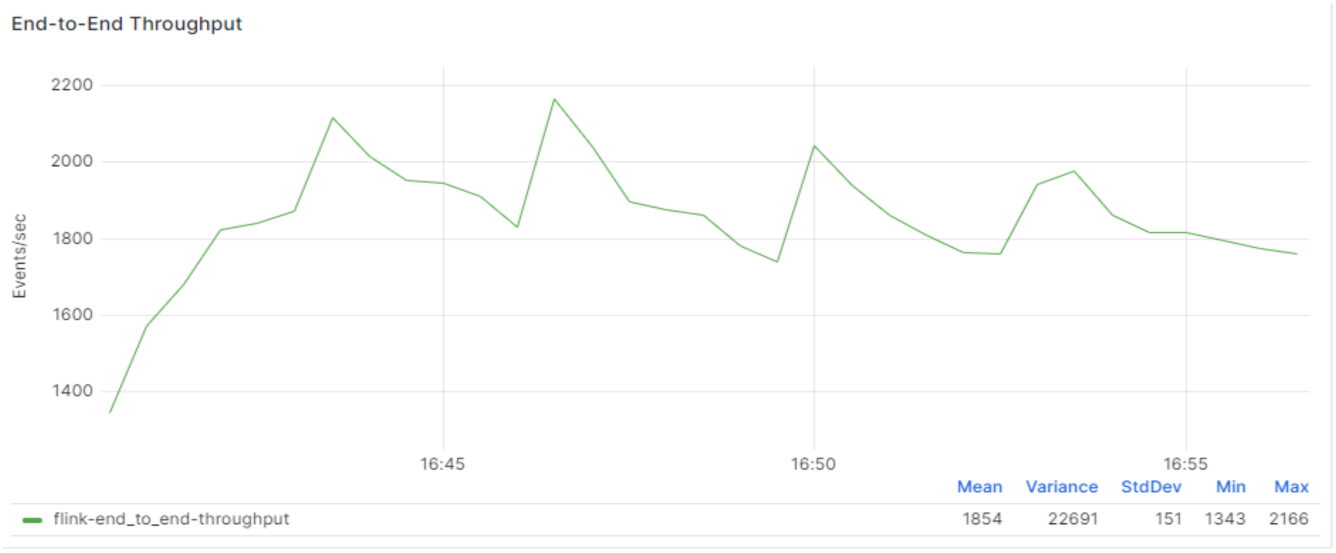
In the case of Flink, the average throughput achieved was 1854 events/sec, with a minimum of 1343 events/sec and a maximum of 2166 events/sec. Storm specific results showcased an average throughput of 1210 events/sec, with a minimum of 570 events/sec and a maximum of 1475 events/sec.

The throughput testing results obtained for different parallelism hints are presented in Table 4.3 and Table 4.4, providing a detailed visualization of the measure-



	Mean	Variance	StdDev	Min	Max
🟥 storm-window-bolt-throughput	1019	16436	128	867	1383

Figure 4.19: Storm Window Throughput - parallelism hint 1

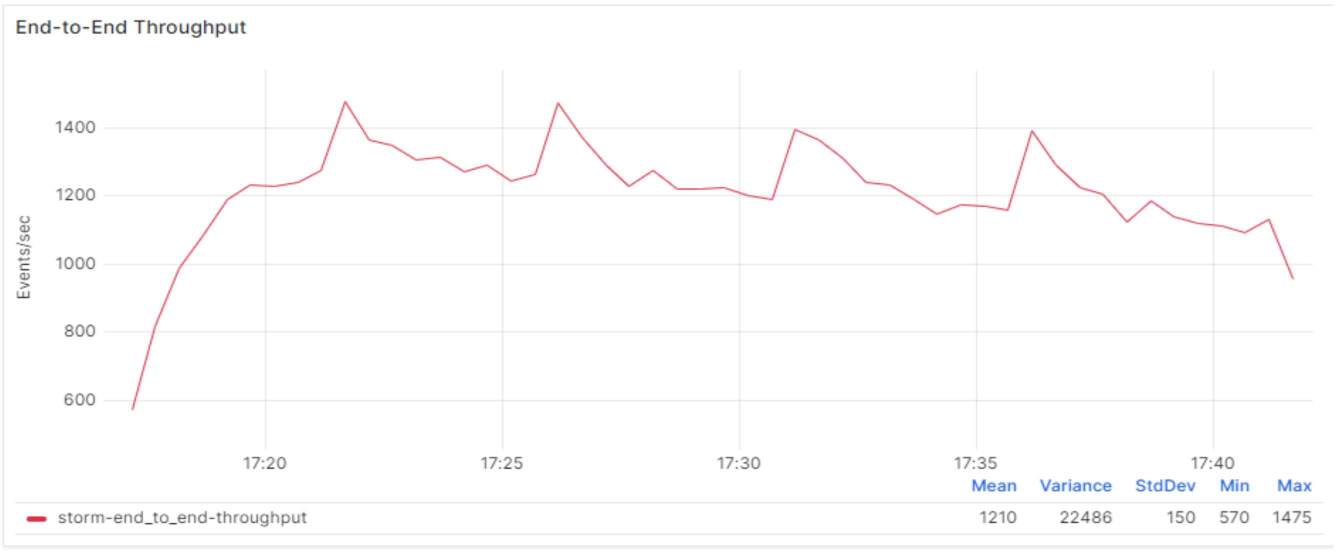


	Mean	Variance	StdDev	Min	Max
🟩 flink-end_to_end-throughput	1854	22691	151	1343	2166

Figure 4.20: Flink End-to-End Throughput - parallelism hint 4

ments. In the case of the window operators, the throughput values are specific to one instance. For Storm, the framework throughput displayed is per Kafka bolt instance, while Flink used only one instance of Kafka sink to produce records due to the insignificant performance obtained of having multiple Kafka sink instances.

By observing the values registered in the tables, it becomes apparent that Flink did not experience any increase in throughput as the parallelism hint increased. In contrast, Storm achieved a significant boost in throughput. Consequently, in order to enhance the throughput of Storm, it becomes necessary to decrease latency or make compromises in terms of latency.



	Mean	Variance	StdDev	Min	Max
storm-end_to_end-throughput	1210	22486	150	570	1475

Figure 4.21: Storm End-to-End Throughput - parallelism hint 4

Paralellism=1	average	min	max
End-to-end	1817	1369	2189
Framework	1871	1596	2269
Window	1879	350	2278

Paralellism=2	average	min	max
End-to-end	1847	1321	2141
Framework	1889	1648	2285
Window	941	822	1144

Paralellism=3	average	min	max
End-to-end	1818	1253	2115
Framework	1825	1154	2172
Window	624	384	757

Paralellism=4	average	min	max
End-to-end	1814	1343	2166
Framework	1826	1647	2167
Window	413	410	417

Table 4.3: Apache Flink throughput - events/sec

Paralellism=1	average	min	max
End-to-end	970	219	1073
Framework	1014	883	1167
Window	1019	867	1383

Paralellism=2	average	min	max
End-to-end	1067	278	1353
Framework	556	467	717
Window	562	467	717

Paralellism=3	average	min	max
End-to-end	1110	470	1375
Framework	271	215	348
Window	275	198	348

Paralellism=4	average	min	max
End-to-end	1210	570	1475
Framework	306	250	383
Window	310	233	383

Table 4.4: Apache Storm throughput - events/sec

4.3.3 Fault Tolerance

Both Flink and Storm have different approaches to fault tolerance, which can impact their resilience in the face of failures.

Flink utilizes a distributed mechanism known as checkpointing to capture the state of the entire application at regular intervals [9]. These checkpoints include the states of all operators, their buffers, and any other necessary information for recovery. For instance, the window operator developed for the job uses in-memory state of the associated JVM which will be regularly captured in order to prevent computation of erroneous financial indicators for subsequent windows in case of failure. The snapshots are stored on the Job Manager's heap which will be used for any recovery processes in the case of failures at Task Manager level.

Flink supports exactly-once processing semantics, guaranteeing that each event is processed exactly once, even in the presence of failures. This is achieved by carefully tracking the processing state and using the information from checkpoints to ensure event deduplication during recovery. Another way of avoiding producing duplicated or inconsistent data is to utilize sinks/producers which supports transactions. For example, the Kafka sink operator in the developed job supports transactions. In the event of failures such as the disconnection of the associated job task manager, records pending to be produced are rolled back to the latest checkpoint.

In the conducted testing scenarios, the impact on latency resulting from the checkpointing mechanism has been found to be insignificant.

In contrast to Flink, Storm has a simpler fault tolerance mechanism [10] supported by the stateless and fail-fast design. As a consequence of these characteristics, Storm regularly stores in ZooKeeper metadata about nodes, tasks and tuples in order to facilitate the recovery processes in the event of a failure. For instance, when a worker fails due to a out-of-memory, the nimbus node instantly starts the process of creating and assigning a new worker to the topology. To resume the work from where the previous worker left off, the new worker queries ZooKeeper to obtain information such as the tree map of tuples, the previous number of tasks and the parallelism for each operator. This allows the new worker to continue the processing without data loss or inconsistency.

In opposition to Flink's checkpointing mechanism, Storm's fault tolerance mechanism has been observed to have a greater impact on the overall latency of the process, due to the costly recovery system and topology restarting process.

While Storm's fault tolerance mechanism is simpler and stateless, Flink's recovery mechanism is faster, leading to minimal impact on job execution. Furthermore, Flink does not rely on external technologies like ZooKeeper, whereas Storm's fault tolerance depends on it.

4.3.4 Development and Maintenance

From a development perspective, both Flink and Storm offer high level abstractions for implementing various components such as tumbling windows, Kafka consumers, and Kafka producers. This eliminates the need for developers to create custom components or start from scratch.

Both frameworks provide built-in operators that are designed using the builder pattern. This approach offers several advantages, including a concise development experience and good code readability. The builder pattern allows developers to construct operators and configurations in a structured and intuitive manner, making the code easier to understand and maintain.

Flink offers a significant advantage with its support for processing events before the end of windows by combining aggregation functions with process functions. This capability enables a clear separation of concerns within the application. On the other hand, Storm does not provide this feature, and any aggregation action can only be performed in the execute function, which is called after a window has ended. This proved a challenge in designing the topology for effectively computing the financial indicators due to this limitation imposed on aggregation actions. An important drawback arises from the necessity of keeping the window bolt's parallelism hint at 1 in order to correctly filter out irrelevant events that will not be processed by the downstream bolt. Consequently, the parallelism of the processing bolt is the only aspect that can be modified, as all tuples arriving in the bolt must be processed.

Storm provides various methods to enable processing in a window bolt with multiple instances. One approach involves using the stream "global grouping" pattern, where window tuples are allocated to a single task. However, this approach undermines the purpose of having multiple executors with multiple tasks, as all the load will be concentrated on just one task while the others remain idle. Another approach is to use the stream "all grouping" method to distribute the load across all tasks. However, this approach does not distribute the load evenly. Instead, the load is duplicated, resulting in an increase in the required resources for unnecessary computations. As such, the optimal approach involves having a window bolt with a parallelism of 1 to effectively select the appropriate events for window calculations. Subsequently, a flexible processing bolt can be utilized, allowing the freedom to choose the number of tasks and executors for the bolt while also guaranteeing the effectiveness of the indicators computations.

An additional benefit provided by Flink, as discussed in Chapter 3, is its automatic processing of Kafka sources, which includes checking for the existence of a topic and creating it if it is not already present. In contrast, Storm's Kafka spout does not possess this functionality, placing the responsibility of performing the aforementioned check on the user.

Storm offers a notable advantage over Flink when it comes to enhancing built-in components like Kafka spout and Kafka bolt. This is due to their straightforward implementation, which requires less time to understand and modify. This simplicity is particularly beneficial when implementing user-defined metrics. For example, when analyzing the latency and throughput of the framework, the metrics were defined by extending the implemented Kafka bolt, which in the case of Flink it was challenging to be accomplished.

From maintenance point of view, Flink's dynamic scaling allows for adding or removing resources without stopping or redeploying parts of the job, while Storm's rebalancing process involves restarting parts of the topology, which may result in temporary downtime during the scaling process.

From an integration perspective, both Flink and Storm provide a straightforward procedure for connecting with the Kafka cluster, and in the case of Storm, with ZooKeeper as well. However, Flink surpasses Storm in terms of offering broader support for third-party integration.

Both Apache Storm and Apache Flink offer a standard set of metrics for monitoring and analyzing the performance of jobs or topologies, as well as gathering information about operators and cluster health. One advantage of Storm over Flink is its built-in support for latency metrics at the component level within a topology. This makes it easier to monitor and evaluate latency in Storm-based applications. In contrast, measuring latency in Flink can be challenging since it provides built-in throughput metrics but lacks latency measurements.

Flink provides cluster metrics using the same metric reporting system as job metrics, while Storm offers fewer options for reporters when it comes to cluster metrics compared to topology metrics. For instance, Storm's topology metrics are reported to Graphite, while the cluster metrics are stored in a CSV file. This difference in reporting options make it more challenging to process and analyze the cluster metrics in Storm. Both frameworks provide wrappers for various types of metrics such as gauges, meters, histograms, and counters.

From the community and support point of view, Storm development faces significant challenges due to the lack of resources, documentation, and smaller size of development community compared to Flink.

Chapter 5

Conclusions

The objective of this thesis was to investigate and determine the optimal technology for processing vast quantities of stock market events by comparing two widely used frameworks specifically designed for this purpose, Apache Flink and Apache Storm. This was achieved by conducting extensive testing taking into consideration latency, throughput, fault tolerance and development experience with the primary objective of identifying the most effective and efficient solution. Based on the presented findings, it can be concluded that Apache Flink represents the better solution due to the superior performance results obtained in the conducted tests, extensive integration support, and overall cost-effectiveness in terms of maintenance.

To fulfill the objective of this thesis, the first chapter provided a comprehensive overview of stock analysis, including its methods and applications. It was followed by a brief analysis of the significance of event-based systems in the financial market domain and their applications. Afterwards, the research problem was introduced, emphasizing the significance of conducting a comprehensive comparative study in this field. The focus was on the cost considerations associated with selecting the suitable processing framework during the initial phases of development. Subsequently, a detailed explanation was provided regarding the calculated stock market indicators and their applications in real-world scenarios.

In Chapter 3, the technological context of the developed system used for conducting the testing scenarios was described. The first section focused on outlining the architecture, providing justifications for the selection of each technology employed in implementing each component. Afterwards, a comprehensive description was provided regarding the type and structure of the events transmitted within the system.

The correlation between these events and the calculations of the indicators, as presented in the first chapter, was highlighted. Subsequently, detailed presentations of both Storm and Flink implementations were provided, highlighting their respective characteristics and pointing out the similarities between them. This was followed by comprehensive explanations of the implemented web server and web client components.

Chapter 4 introduced the metrics used in the comparative study, followed by the presentation of the testing scenarios and their justifications. Additionally, the benchmark setup was discussed, along with the justification for selecting Docker as the hosting platform for the testing scenarios. Subsequently, the results of the performed latency and throughput tests were presented, accompanied by visual representations and detailed observations. The conclusion drawn from the analysis indicated that Flink outperformed Storm in both latency and throughput measurements. Additionally, the fault tolerance mechanisms of each framework were described, highlighting their applications in the implemented job and topology. Lastly, the chapter concluded with an analysis of the development and maintenance aspects for each framework, taking into account the challenges faced and observations made during the implementation phases.

In terms of future work, there are a couple of potential directions to consider. Firstly, it is possible to expand the comparative study to incorporate Apache Spark, an extensively used framework for large-scale event processing. By analyzing Apache Spark alongside Apache Flink and Apache Storm, it would provide a concise hierarchical classification of the available technologies in terms of performance and development experience.

Another idea for future work could involve exploring the aforementioned comparative study within the context of high performance machines equipped with state of the art components. This would allow for a more comprehensive assessment of a framework capabilities when operating on advanced hardware configurations.

Bibliography

[1] Hinze, Annika, Kai Sachs, and Alejandro Buchmann. ”Event-based applications and enabling technologies.” Proceedings of the Third ACM International Conference on Distributed Event-Based Systems. 2009.

[2] M. Makrehchi, S. Shah and W. Liao, ”Stock Prediction Using Event-Based Sentiment Analysis,” 2013 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT), Atlanta, GA, USA, 2013, pp. 337-342, doi: 10.1109/WI-IAT.2013.48.

[3] Luca De Martini, Alessandro Margara, and Gianpaolo Cugola. 2022. Analysis of market data with Noir: DEBS grand challenge. In Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems (DEBS ’22). Association for Computing Machinery, New York, NY, USA, 139–144.

[4] Kevin Li, Daniel Fernandez, David Klingler, Yuhan Gao, Jacob Rivera, and Kia Teymourian. 2022. A high-performance processing system for monitoring stock market data stream. In Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems (DEBS ’22). Association for Computing Machinery, New York, NY, USA, 166–170.

[5] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen and V. Markl, ”Benchmarking Distributed Stream Data Processing Systems,” 2018 IEEE 34th International Conference on Data Engineering (ICDE), Paris, France, 2018, pp. 1507-1518, doi: 10.1109/ICDE.2018.00169.

[6] James Chen, March 11, 2023, *Stock Analysis: Different Methods for Evaluating Stocks*, Investopedia, accessed 05 May 2023, ⟨https://www.investopedia.com/terms/s/stock-analysis.asp⟩.

[7] Angadi, Mahantesh C., and Amogh P. Kulkarni. "Time Series Data Analysis for Stock Market Prediction using Data Mining Techniques with R." International Journal of Advanced Research in Computer Science 6.6 (2015).

[8] DEBS 2022, 2022, *CALL FOR GRAND CHALLENGE SOLUTIONS*, DEBS, accessed 14 June 2023, ⟨https://2022.debs.org/call-for-grand-challenge-solutions/⟩.

[9] Apache Software Foundation, 2023, *Apache Flink® — Stateful Computations over Data Streams — Apache Flink*, Apache Software Foundation, accessed 13 June 2023, ⟨https://flink.apache.org/⟩.

[10] Apache Software Foundation 2022, *Apache Storm*, Apache Software Foundation, accessed 16 June 2023, ⟨https://storm.apache.org/⟩.

[11] Apache Software Foundation 2023, *Apache Kafka*, Apache Software Foundation, accessed 20 June 2023, ⟨https://kafka.apache.org/⟩.

[12] Apache Software Foundation 2023, *Apache ZooKeeper*, Apache Software Foundation, accessed 20 June 2023, ⟨https://zookeeper.apache.org/⟩.

[13] Orbitz Worldwide, Inc 2022, *Graphite*, Orbitz Worldwide, Inc, accessed 20 June 2023, ⟨https://graphiteapp.org//⟩.

[14] Grafana Labs 2023, *Grafana: The open observability platform — Grafana Labs*, Grafana Labs, accessed 20 June 2023, ⟨https://grafana.com/⟩.

[15] VMware 2023, *Spring Boot*, VMware, accessed 20 June 2023, ⟨https://spring.io/projects/spring-boot⟩.

[16] Google 2023, *Angular*, Google, accessed 20 June 2023, ⟨https://angular.io/⟩.

[17] Swimlane 2023, *Introduction - ngx-charts*, GitBook, accessed 20 June 2023, ⟨https://swimlane.gitbook.io/ngx-charts/⟩.

[18] Docker Inc. 2023, *Docker: Accelerated, Containerized Application Development*, Docker Inc., accessed 20 June 2023, ⟨https://www.docker.com/⟩.