

Streaming Pipeline Using Apache Kafka and GraphQL for Dynamic Trade Surveillance

Natalia Robledo Díaz



4th Year Project Report
Computer Science and Management Science
School of Informatics
University of Edinburgh
2023

Abstract

Continuous sequences of data are referred to as data streams. Computing running aggregations on the inbound stream has shown to have multiple applications in the industry, from real-time stock value fluctuations or even IoT data pipelines. Previous to the adoption of (open-source) real-time event streaming platforms, data analytics could often only be produced in long-running batch jobs. By combining Kafka and GraphQL, this project leverages the strengths of both technologies and successfully attains the main goal of efficiently addressing current industry issues.

With a context in trade surveillance, the project builds an end-to-end optimised new data pipeline architecture that addresses two main legacy issues in the field: shifting from static to dynamic thresholds and from batch processing to (near) real-time monitoring. Given the scarce literature focusing on this topic, the completion of the project serves to bridge the existing application gap between industry interest and academia research in anomaly detection in finance, and proposes a combined architecture which has yet remained unexplored in the literature. The project is developed following a two-staged approach. The first stage builds a live data pipeline that would allow for dynamic threshold settings and classify fraudulent activity by performing windowed continuous aggregations using Apache Kafka, and the second stage develops a GraphQL interface to recreate the compliance post-trade analytics.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Natalia Robledo Díaz)

Acknowledgements

I would like to thank my project supervisor, Dr. Michael Glienecke, for consistently providing time and weekly guidance throughout the development of this project.

I would also like to thank my family for all their encouragement and support given throughout my degree.

Lastly, I want to thank my friends, a constant source of support, inspiration and joy.

Table of Contents

1	Introduction	1
1.1	Motivation	3
1.2	Goals	4
1.3	Project Contributions - Critical Evaluation of Previous Work	5
1.4	Structure of the Dissertation Report	6
2	Background	7
2.1	Event streams	7
2.2	From Batch Workload to Stream Processing	7
2.3	Technology used for Streaming Live Data – Messaging Systems	8
2.4	Distributed Messaging Systems - Why Are They Useful for the Project?	9
2.5	From REST API to GraphQL - Web API Query Standards	11
2.6	Modelling through Graphs in GraphQL	12
2.7	Research Landscape	14
3	System Design and Architecture	15
3.1	System Specifications and Requirements	15
3.2	Streaming Data Processing Architecture	17
3.2.1	Challenges on Streaming Application	17
3.2.2	Possible Streaming Architectures	18
3.3	High Level Overview of the Prototype’s Architecture	18
4	Implementation	22
4.1	Getting Started - Setting up Docker	22
4.2	Data Collection	23
4.2.1	Dataset	23
4.3	Kafka Producer	24
4.3.1	Data Manipulation: Generating Outliers and Price values	24
4.3.2	Data Serialization	24
4.4	Data Processing	25
4.4.1	Spring Boot Server	25
4.4.2	Consuming Records from Kafka Topic	25
4.5	Data Aggregation – Dynamic Thresholds	26
4.5.1	Why Dynamic Thresholds	26
4.5.2	Methodology to Implement Dynamic Threshold using Kafka Streams	28

4.6	Query Language: GraphQL Interface	31
4.6.1	Queries	32
4.6.2	Subscriptions	33
5	Evaluation	35
5.0.1	Evaluating Kafka: Functionality Validation - Handling Errors	35
5.0.2	Evaluating GraphQL: Functionality Validation	35
5.0.3	Evaluating Overall System Performance	35
6	Conclusion	39
6.0.1	Reflection	39
6.0.2	Future Work	40
	Bibliography	41
A	Unit Tests	46

Chapter 1

Introduction

Data streams are a continuous sequence of data that are being generated over a period of time. Fraud detection, real-time stock value fluctuations or even IoT data pipelines are examples of industry applications, where the consumption and analysis of data streams is altering industry dynamics [51]. Through the use of a data streaming platform, a company can perform analysis of every incoming data record ¹ applying techniques such as sampling, filtering or aggregation, and gain new business insights to increase its level of control over its data processes [2].

Previous to the adoption of (open-source) real-time event streaming platforms, data analytics could often only be produced in long-running batch jobs. This is because the most common previous architectures (which often employed an ETL² batch workload processing) were not designed to be event-driven [53]. These batches present major problems, including that calculations (such as running analytics) could not be started until the data extraction process was completed or batched [8], possibly affecting the following day's operations. Legacy application architectures often consist of many data silos [7] and multiple data systems that require manual workarounds [53], preventing companies from using the full potential of their data.

The second motivation for real-time event streaming platforms is that the data analytics process is further fractured and delayed by the use of inflexible REST APIs³ to fetch the data, resulting in multiple additional endpoint requests [55]. It is argued that having multiple endpoints to fetch data in a pipeline introduces latency and further complexity to event-driven architectures [2].

Rebuilding legacy systems into new state-of-the art technology has drawn significant research interest, which greatly focuses on improving and optimizing the three stages of a data pipeline architecture which are: data collection, data aggregation⁴ and data analysis⁵ [2]. There is a widespread interest both in academia and industry for building efficient streaming architectures; these views are often focused solely on parts of the

¹Data unit that is processed and transmitted through the pipeline.

²known as Extract, Transform and Load, ETL architectures process data in large volumes at intervals.

³A naïve, and traditional, approach default to using HTTP requests.

⁴Data Aggregation refers to processing data from different sources.

⁵Data Analysis refers to drawing insights from data in order to help make data-driven decisions.

data pipeline, like enhancing data collection and aggregations [20][32], or improving data fetching for analytics [55][34].

Accessing data from different sources, and therefore integrating silos within an organization to link existing data, has incredible potential to yield insights. But many organizations have struggled with creating the right structure for that synthesis to take place [2].

Apache Kafka is a distributed event streaming platform for high performance data pipelines that was firstly developed by LinkedIn as a way of tackling the issue of developing a scalable fully stream-based architecture [33]. Ever since LinkedIn released the first open-source version of Apache Kafka, over 80% of the Fortune 500 nowadays have adopted it to rebuild legacy systems into faster and reliable data pipelines [46].

Apache Kafka allows to receive and send high volume of data with low latency, and the data is stored in-order. This technology has thus sparked great interest within the academic and research area, many of which evaluate and compare Kafka applications to current streaming solutions [35], or even how to best connect ML/AI frameworks with streaming processes [38].

This has driven Kafka to become a very popular and available framework to work with, allowing companies to develop the right structure for combining and integrating large stores of data [51]. Using Kafka as a main messaging system for streaming data allows tackling both data collection and aggregating information from different sources in the data pipeline.

Having the right distributed system in place will allow for an efficient data streaming pipeline, but using data for analysis and decision making is the following critical step to ensure the third part of the streaming data pipeline process (data analysis) is optimized. The most common software architectural style defaults to using HTTP requests⁶, which is often used as the protocol for implementing REST services, to render the data needed [6]; using RESTful web services often leads to under-fetching⁷ or over-fetching⁸, resulting in multiple additional endpoint requests [52].

GraphQL is a novel query language for implementing service-based software architectures and is as an efficient alternative to REST-based applications, allowing clients to define exactly the data they require from service providers [31]. Initially open-sourced by Facebook, it is now supported by major Web APIs, including the ones provided by GitHub, Airbnb, Netflix, and Twitter [31].

Combining both Apache Kafka's and GraphQL's synergies will allow to develop an end-to-end dynamic data streaming architecture and exploit their main features to propose solutions to current industry issues. There is little academia research as to how an end-to-end data pipeline combining Apache Kafka and GraphQL would greatly benefit the streaming data domain. There is considerable application gap between industry and

⁶HTTP requests are endpoints that provide a simple way to define the base URL.

⁷Underfetching takes place when an API call doesn't have enough data sent to an endpoint, necessitating another call.

⁸Overfetching takes place when when an API request returns too much data that you're not going to use.

academia research for anomaly detection streaming architectures in finance (referred to as trade surveillance). The motivation for this proof-of-concept is supported by the fact that existing state-of-the-art streaming services in the trade surveillance domain still rely on low-latency RESTful applications.

The process of monitoring financial transactions for potentially illegal or abusive trading (known as trade surveillance) has drawn interest from industry leaders towards near real-time monitoring [27] [19]. The industry is focusing on moving towards a flexible and dynamic approach to detect anomalous activity (trading activity) [19]. Current solutions rely on batch workload processing, meaning that suspicious activity cannot be detected before data processing has finished [27]. Furthermore, there has been a considerable interest in the industry towards dynamic threshold settings [21][37], moving away from fixed and static values that heavily rely on historical activity, failing to capture real-time patterns.

However, there is scarce literature focusing on this topic. Most trade surveillance research literature focuses on ‘high-performance architectures’ [3], ‘low latency using in-memory data grids’ [4], but there is no recent literature that provides an end-to-end architecture for dynamically monitoring trading activity. Therefore, this project aims at introducing a PoC (proof-of-concept) involving cross-sector applications of Apache Kafka and GraphQL, and how their application in larger technical projects would transfer to the trade surveillance domain.

1.1 Motivation

Current state-of-the-art trade surveillance monitoring systems operate following a threshold-based approach to detect outliers in the stock market orderbook⁹ activity [42]. Threshold-based alerts have one or a combination of predefined limits (thresholds) and will generate an alert when any incoming executed order¹⁰ exceeds them as seen in Figure 1.1.

This approach presents two major issues. On the one hand, current surveillance solutions rely on a “too late infrastructure”, in which aggregations and on-demand analytics are only available after illegal trading activity has finished. This approach, called batch workload processing, implies that suspicious activity cannot be detected before data processing has finished [27]. Secondly, monitoring for illegal activity operates on fixed threshold values, heavily relying on historical activity, and therefore failing to capture real-time patterns. Additionally, these threshold values are only updated after, and if, threshold calibration is deemed appropriate after manual review [41].

Although there has been a huge emphasis on implementing machine learning for trade surveillance monitoring [42], it rather shifts the focus to using predicted benchmark values [21]. This would mean that detecting anomalies would still be based on historical values, and threshold values would still remain fixed (current industry solutions operate under fixed thresholds as explained in [1]). This poses a serious risk, as many suspicious

⁹Live record of all buy and sell orders for a given stock.

¹⁰This is an order to buy or sell a security at the prevailing market price.

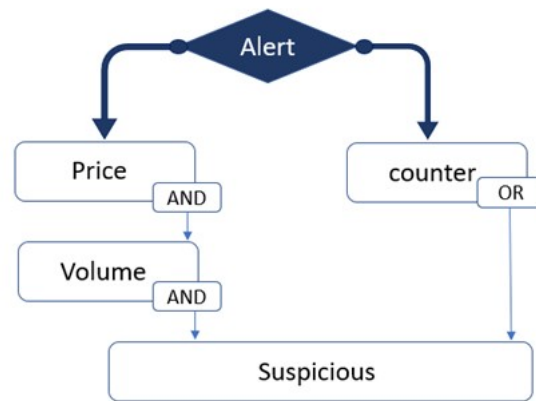


Figure 1.1: Example of rule-based anomaly detection alert. This diagram was developed by me, with inspiration from Nasdaq SMARTS surveillance system.

trading activities would be ‘flying under the radar’.

There is scarce literature that focuses on tackling the former issues. There is no recent academic paper that presents a proof-of-concept for an end-to-end streaming data pipeline addressing an dynamic aggregations and a dynamic interface to query data.

There is little academic research as to how an end-to-end data pipeline combining Apache Kafka and GraphQL would greatly benefit the streaming data domain. There is considerable application gap between industry and academia research for anomaly detection streaming architectures in finance. The motivation for this proof-of-concept is supported by the fact that existing state-of-the art streaming services in the trade surveillance domain still often rely on RESTful applications.

1.2 Goals

The project description allowed for multiple possible approaches, but the main takeaway was to take the opportunity to frame the proposal into a possible current industry solution. The key goal of this project is to leverage the combination of both technologies (Kafka and GraphQL), building upon the existing literature, and provide an end-to-end streaming pipeline novel solution to address current industry limitations with a context on trade surveillance. The project built upon the required completion criteria while developing a two-staged approach. The first one consisted of building a live data pipeline that would allow for dynamic threshold settings and classify fraudulent activity by performing windowed continuous aggregations using Apache Kafka, and a second stage in which a GraphQL interface is developed to recreate the compliance post-trade analytics.

1.3 Project Contributions - Critical Evaluation of Previous Work

The main goal is to build a reliable, scalable and ultimately maintainable end-to-end data pipeline from scratch, addressing dynamic aggregations and a dynamic GraphQL interface to query and subscribe to data. By using modern technologies, mainly Kafka and GraphQL, the project provides a real-time monitoring approach using dynamic thresholds, a seamless integration of data sources, and a scalable GraphQL API for handling large volumes of data. In this dissertation, with a context on trade surveillance, a prototype of such system is proposed as a proof of concept. Later chapters will evaluate how well the project requirements (detailed in Chapter 3) are met.

The proof-of-concept targets existing gaps between research papers and industry focus and provide an end-to-end data pipeline solution by leveraging modern technologies. These include, among others, Kafka and GraphQL, while also using supporting technologies throughout, including Docker, Springboot, Javascript, Python, Kafka Streams and Apollo Server.

The proof-of-concept builds upon previous research in trade surveillance and contributes to the literature (explained in-depth in Chapter 2) by demonstrating the use of state-of-the-art technologies to address some of the limitations of traditional surveillance systems.

Research papers such as [18] by M. Fevzi Esen et al., further investigate leveraging streaming data to detect illegal trading activities using techniques such as data mining or machine learning. However, their main focus is only enhancing the suspicious detection algorithm component rather than building an end-to-end pipeline architecture that uses modern technologies to address current issues. Using machine learning would mean heavily relying on historical activity, which would hence fail to capture real-time patterns.

On top of this, an important finding from the literature review (Chapter 2) is that most academic/research papers, either on Kafka or GraphQL, mostly focus on improving one part of the data pipeline: data collection, data aggregation or data analysis. For example, [38] and [54] focus on how to best connect to ML/AI frameworks with Kafka, [47] focuses on performance comparisons to improve data analysis or even papers such as [9] to improve displaying analytics. As a result, the literature review remains fractured as there is little emphasis made on overall streaming data pipeline solutions.

Currently, there is no proof-of-concept for an end-to-end streaming pipeline using Kafka and GraphQL for trade surveillance. Hence, this work is able to provide new insights contributing to:

- **Dynamic Thresholds:** The use of dynamic thresholds is a key feature of the proof-of-concept using Kafka and GraphQL. This approach allows for more accurate alerts and reduces false positives compared to traditional surveillance systems, which often use static thresholds and rule-based systems.

- Dynamic query API¹¹: The proof-of-concept using GraphQL demonstrates the system can integrate data sources to provide flexible query APIs. It optimizes the query response by allowing the server/client to specify the data needed and dynamically generating a response that only includes the requested data.
- Scalability: The use of Kafka enables (near) real-time monitoring¹² for today's high-speed trading systems (tested in Chapter 5).
- Real-time Monitoring: The approach of using Kafka and GraphQL enables real-time monitoring of trading activities. This means that the system can detect and respond to market events as they happen, rather than relying on delayed data.

1.4 Structure of the Dissertation Report

The structure of this dissertation is divided into 6 chapters.

Chapter 1 states the motivation behind the main idea, highlighting the project's goals, the main gap in the literature it aims to bridge, and highlights the project's contributions. This provides a comprehensive clear overview of the problem and how it aims to stand out from previous work in the field.

Chapter 2 will give a background on the technologies used to deliver the project, as well as a thorough literature review on both Kafka and GraphQL best practices which can thereafter be applied on the system. This sets the foundation that allows to critically address previous work, and support the design decisions made in the next chapter.

Chapter 3 will showcase and reason the design choices made throughout, as well as give an overall overview of the system's implementation. It thus provides both functional and non-functional requirements the design needs to comply with, which will be tested in Chapter 5.

Chapter 4 explains the technical process of developing the project from inception to completion, highlighting challenges faced and their proposed solutions.

Chapter 5 Evaluates the system and the requirements exposed in Chapter 2 through unit testing and use case testing through different scenarios.

Chapter 6 presents the final reflection, along with further work proposal.

¹¹A dynamic API using GraphQL means it is flexible and adaptable to changes

¹²This project refers to real-time as the continuous monitoring of incoming data as soon as it is received in the system (Kafka). The use of term *near* accounts for possible bottlenecks for higher loads as shown in Chapter 5.

Chapter 2

Background

This chapter will outline preliminary concepts which form the starting point of the project. It will provide a more in-depth perspective on Apache Kafka and GraphQL, which are the main technologies developed throughout. Given the broad range of open-source technologies available to develop and implement the project's architecture, understanding their key features and capabilities is key to partake a comparative analysis of them. The project's implementation is structured in two main components: a streaming platform and an interface for querying and subscribing data (based on GraphQL). As such, the background will first focus on streaming data, followed by further details on web API query standards (mainly GraphQL).

2.1 Event streams

The data injected into a streaming pipeline comes in the form of records, which are often referred to as *events*. They usually contain a timestamp, that allows to maintain a chronological order by indicating the exact moment they were sent. The data that sensors send regarding room temperature/humidity or, in the case of trading, each trading order executed by the trading floor, are all possible examples of event streams.

These events are described as immutable objects which include information regarding a specific event that happened at some point in time [30].

With batch processing, a file can be written only once and then queried or read multiple times. On the contrary, stream processing allows to generate an event by a producer (also known as publisher or sender), and then be potentially consumed (subscribed) by multiple consumers. A collection of associated records is frequently referred to as a *topic*.

2.2 From Batch Workload to Stream Processing

Data that is being continuously generated by different sources (like server log files or trading floors) and is available to use without having to initially download it [12],

is often referred to as *streaming data*. The main challenge lies in processing data sequentially (or sliding-window) to perform analytics on the stream [12].

Previous to the adoption of open-source real-time event streaming platforms, data analytics often could only be produced in long-running batch jobs as common architectures (which often employed an ETL batch workload processing) were not designed to be event-driven [53]. Batch data processing is the traditional method consisting of collecting, processing and analysing data in batches or scheduled jobs.

These batches presented major problems, including that calculations (such as running analytics) could not be started until the data extraction process was completed [8], possibly impacting the next day's operations. Since batch data integration involved storing all data in one batch, this approach was only beneficial to those who could wait to receive all data and analyse it periodically (Table 2.1 shows the comparison table). Hence, it has only proven to be successful for scenarios where time is not an issue.

However, a study carried out by IBM showed that more than 80% of participants [25] claimed to have noticed a considerable increase in both the volume and the speed of data handled. Digitalization has shifted the way companies receive and process data, as data is rarely static, and there are now major disparities in performance between those companies which harness streaming data and the those who still rely on legacy systems, creating "winner-take-most dynamics" [2].

	Batch Processing	Stream Processing
Data Scope	Processing most of the data	Rolling window
Data size	Large batches of data	Micro batches
Performance	Latency in hours/minutes	Seconds/milliseconds
Analysis	Complex	Simple aggregations

Table 2.1: Batch vs Stream Processing

2.3 Technology used for Streaming Live Data – Messaging Systems

The different technologies used on a streaming pipeline need to be able to communicate effectively between them. Although a database can act as the primary connection for messages exchanged between producers¹ and consumers, it does not allow for any notification mechanism and polling for updates is often considered to be expensive and inefficient.

A solution to address the latter polling inefficiency is using a *messaging system*. This allows for a producer to send messages that contain events for a particular topic, which is then published to consumers. It is an efficient way of transferring data between applications in a reliable, fast and scalable way. Therefore, its main objective is to safeguard the message between two parties.

¹It is usually an application that generates and sends messages to a topic.

This is an important aspect for the proof-of-concept in this project, as there is a need for transferring variable loads of data, expecting the consumer to receive such messages in a reliable way. With a context in trade surveillance, a key consideration is that the messaging system can indeed become a bottleneck itself as the system scales up and increases its throughput or messages [56]. Since outlier detection is performed as soon as messages are received, there is a risk that the message broker will queue up the messages if the message rate increases. This will promote a high-latency data pipeline which will lead to considerable delays in the consumption of messages

The project requires consumers, which in our case is the Java application, to receive data through a message system, in order to process this incoming stream and detect outliers. There are three different types of communication patterns that a message system can use:

- **Point-to-Point** : Only one consumer can receive the message from a producer [15].
- **Publish/Subscribe**: The producer sends messages to a general message channel (or topic), from which various consumers can retrieve messages. It is important to highlight that the producer sending the messages does not require any knowledge on the number of receivers subscribed, as well as the receivers do not need to keep track on how many senders there are (they can process the messages whenever preferred).
- **Request-response**: In this communication pattern, the producer sends a request to a consumer, and the latter one replies back with a response.

When developers of both applications have a common interface, this type of communications allow for a lower coupling between them. This is because they are independent of each other. Even if programming languages or system requirements/details change over time, as long as the shared interface is intact, the two systems will be able to communicate between them. Having publishers and consumers loosely coupled encourages evolutionary systems (systems that can adapt and evolve over time).

2.4 Distributed Messaging Systems - Why Are They Useful for the Project?

Distributed messaging systems usually implement a publish/subscribe pattern, which as mentioned before, allow to decouple the communication of data from the processing of data. A system has a data producer, a data consumer and a storage system.

Typically, a monitoring trade surveillance system would receive data from different trading desks that are constantly executing orders at an institution (an investment bank, for example). A distributed messaging system would allow to have independent applications (for instance, trading desks) send their orders to a place-holder previously referred to as a topic or message channel (and have multiple interested subscribers consume such incoming data).

This decoupling becomes crucial for the project's future scalability. In fact, research on

monitoring trade systems allows to get further insight into how this flexibility requirement is important, as a systems should enable the possibility of whether monitoring or not (and therefore subscribe to such stream of data or not), allowing to accommodate business decisions.

Kafka

Apache Kafka is a distributed event streaming platform and works as a messaging system [33], as it is used for processing streamed data. Kafka allows to send and receive high throughput of data with low latency, where data is stored sequentially and can be used for long-term storage (records are stored as long as it is needed).

Kafka encompasses various concepts including topics, which are a stream of messages of a particular type (Figure 2.1). Topics provide a way to store and persist data, since producers² send data which is then stored into a topic, from which consumers³ can then read.

Messages are sent into Kafka topics, and act as place-holders (channels) by storing data before being consumed by subscribers (also referred to as consumers). These topics can themselves be further divided into what is known as partitions. All messages in a particular partition contain a unique sequential id (which allows to read messages from lowest index to highest).

The chronological ordering of a stream of messages is only kept within each partition in a topic. Every partition can be allocated on different servers, which allows for horizontal scalability⁴.

These messages are consumed by subscribers (or consumers) which keep an *offset* in their metadata should a consumer stop and wish to start reading later on (this is implemented in Chapter 5 for testing purposes).

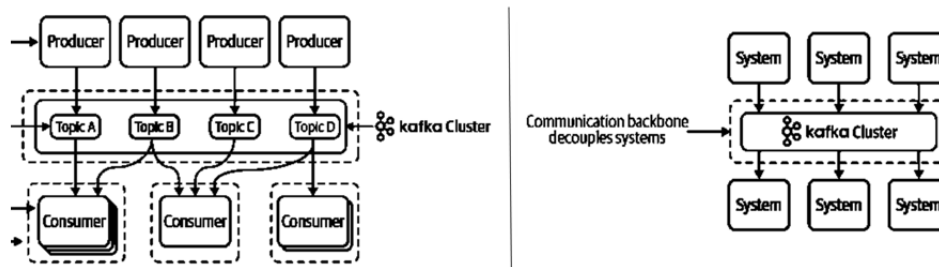


Figure 2.1: Apache Kafka Cluster. Sourced and adapted from [48].

A Kafka server, also referred to as a broker, ensures that consumer's requests for partition messages are addressed, saves messages and provides offset measurements

²client application that publishes or writes data to one or more topics in a Kafka cluster

³A consumer can be an application that subscribes to a topic(s) and consumes/reads data from them

⁴Allows the system to handle increased workload by adding more resources, and thus distribute it

for producers. When different brokers collaborate, they form a *cluster* (employs a leadership architecture), in which each server manages its partition (assigned by a controller). To enhance durability, several brokers can be designated for a partition.

Zookeeper

Apache Kafka employs Zookeeper which allows to manage a cluster of brokers. This is done by tracking the cluster's metadata and subscriber's information. Zookeeper helps with failure detection, crash detection, and assists with Kafka's synchronization (functions as a key-value manager).

Zookeeper and Kafka mutually benefit each other, since Zookeeper helps manage and control possible issues in the system that might arise with multiple clusters.

2.5 From REST API to GraphQL - Web API Query Standards

The execution of a request for information, known as a *query*, provides a response which meets the specified query criteria. To execute queries, there needs to be a set of instructions to allow the queried system (can be a database) to understand them. Hence, a query language is therefore a language which uses formal programming commands to retrieve data.

As requirements grow more complex [2], having in place an optimal query language to extract, and thereafter perform data analysis, has become critical to help make fetching the results of data analyses digestible.

REST has long been the most common architecture for web-APIs and applications since its introduction in Fieldings dissertation in 2000 [16]. The way a REST API work is by communicating via HTTP requests which perform CRUD (Create, Read, Update, Delete) operations on the queried system. As a result, using a REST API would mean that gathering data to perform data analysis/aggregations would possibly entail either making several different calls to multiple endpoints, or developing an additional back-end server that makes these calls to provide a single endpoint to the user. This often becomes too inflexible and expensive.

In the past years, a new architecture, GraphQL, has been introduced in the API query system domain, offering a dynamic alternative to RESTful APIs. GraphQL is a dynamic query language that offers flexibility in its queries, enhancing usability. This query language allows clients to state exactly the information they require (the structure of the data needed) in a single endpoint, and the response will only include exactly what the client needs.

Performing data analytics on a data pipeline requires fetching information from multiple sources to perform the required aggregations. Compared to a classic REST architecture, which often either requires calling various endpoints to retrieve specific pieces of information or an additional back-end server, a GraphQL architecture can fetch everything

in a single request as seen in Figure 2.2. This dynamic feature can improve application performance by minimizing the time spent on back-end queries, and allows to reduce the number of round trips required to fill a web page (Figure 2.2).

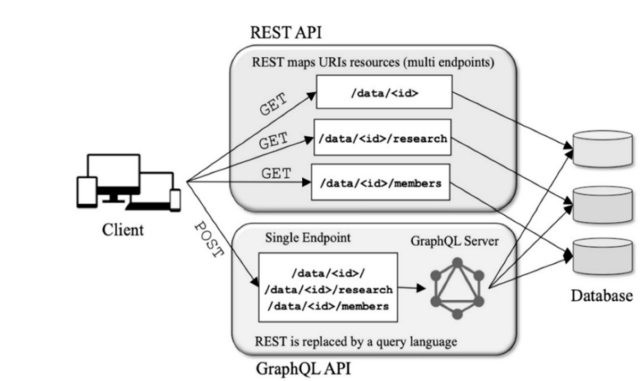


Figure 5. REST API vs GraphQL

Figure 2.2: REST API vs GraphQL. Sourced from [34].

2.6 Modelling through Graphs in GraphQL

GraphQL is a query language for APIs, and a server-side runtime for executing flexible queries release by Facebook in 2015 [23]. Contrary to REST calls, GraphQL only has a single endpoint for each query (although can have multiple ports too) and allows the client to define the structure of the data needed for such query. This, in turn, avoids transferring unnecessary/unused data or relying on making multiple endpoint requests to fetch all data.

GraphQL allows framing business logic as a graph, which closely resembles our mental models. In graphs, relationships between represented entities are as relevant as the entities themselves. A graph is built of abstract objects known as nodes, connected by edges. Following these edges allows to recursively traverse the graph.

By defining a schema, GraphQL allows to establish the structure of the different nodes and how one relate to each other (via the edges). The schema itself is a graph composed of types representing different entities (Figure 2.3). Most importantly, since GraphQL only defines the interface, there is a freedom to choose whichever backend is deemed appropriate.

How GraphQL Works

Essentially, queries will be sent to the GraphQL server, which will then reply with a JSON object containing the specified information. It is considered more efficient than previous web query standards (such as REST) because the client sends customized POST request to fetch only the needed data.

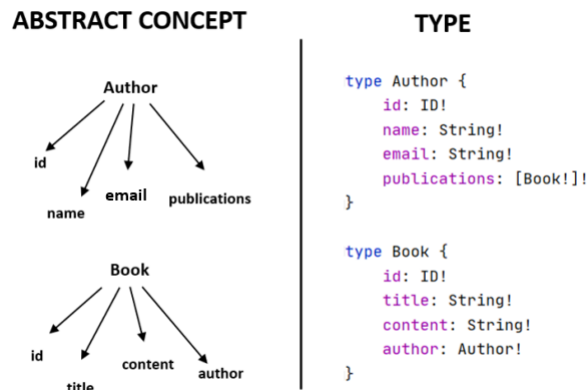


Figure 2.3: Example GraphQL Graph

The two fundamental aspects of GraphQL are :

- **Schema:** The schema functions as a translation of the app’s foundational data model into GraphQL schema language.
- **Resolver:** Resolvers allow the system to retrieve the required data. These are functions written in the service’s programming language that explain how to populate the data returned by a particular query (defined using a schema). Without resolvers, the system cannot execute GraphQL requests.

GraphQL allows for three main processes:

- **Query:** Queries allow clients to retrieve data from the server (usually reads from database). Although, as seen in Figure 2.4, the Author class has ”email” and ”publications” as attributes too, the query described by GraphQL in Figure 2.4 will only return exactly what is requested.



Figure 2.4: GraphQL Example Query

- **Mutation:** In GraphQL, mutations are a type of operation that allows the client to modify data on the server. Mutations are used when the client wants to create, update, or delete data on the server.
- **Subscription:** Subscriptions allow to establish and maintain a real-time connection between the client and the server (most commonly via WebSocket). This essentially allows the server to push updates to the subscription’s result. This becomes useful to avoid repeatedly polling for large JSON objects or ensure clients receive updates as soon as changes occur. Whenever a client executes a subscription, unlike with queries, it establishes a connection with the server.

However, the server will only return a response when there is an update available (receives information).

2.7 Research Landscape

Understanding the research landscape on how Apache Kafka and GraphQL are currently being used in other domains was relevant to further exploring how to best develop the end-to-end data pipeline for trade surveillance. Academic literature regarding streaming data applications has drawn attention long before Apache Kafka open-sourced in 2011 by academic in papers such as [36] [11]. However, since then, academic interest has grown exponentially, shifting the focus from building efficient pipelines for non-continuous data processing [26] [39] to researching how to best connect ML/AI frameworks with streaming processes [38] to performance comparisons under tight latency constraints for streaming data [9].

There is an influx of research into several industries, most of which benefit from analysing real-time incoming data, racing to bring real-time synchronization of their apps and operations [2]. With medical applications and sensor stream frameworks [13] becoming more prominent in the streaming data pipeline research landscape, academic focus has moved from building basic infrastructures to optimizing high-throughput low latency systems.

Consequently, ever since Apache Kafka was released as an alternative for data streaming, it has been prominently cited in recent research papers. Papers such as [44] highlight Kafka's suitability to deal with large volumes of data, and proposes a high level data pipeline architecture to classify tweets in real time using machine learning frameworks for applications such as sentiment analysis. Other research trends [50] propose using Kafka to act as a distributed streaming platform, for instance used to monitor real-time traffic light violations using machine learning models.

But Kafka also has interesting stream processing frameworks, such as Kafka Streams, that allows to perform data aggregations to the inbound stream. To this end, the proposed solution for trade surveillance shifts fixed threshold values to dynamic ones, using Kafka Streams to perform windowed operations on the stream. Hence, understanding how Kafka and its libraries were applicable to this pressing issue was key. Papers such as [28], although applied to the cloud domain, use Kafka Streams to develop benchmarking methods, allowing researchers and practitioners to conduct empirical scalability evaluations of cloud-native applications.

On top of this, GraphQL has also sparked academic attention given its dynamic features. A considerable number of highly cited papers for GraphQL focus on comparing it with REST under controlled experiments [6] as well as practical guidance to best migrate to GraphQL [5]. There is a considerable emphasis on the literature that GraphQL outperforms RESTful web services and presents an efficient way of connecting to different databases as this paper states [9]. Since the project aims at providing and displaying analytics to the front-end, academic literature has thus emphasized how GraphQL's powerful frameworks is an efficient and suitable way of overcoming data fetching issues.

Chapter 3

System Design and Architecture

This chapter focuses on showcasing and reasoning the design choices made throughout, as well as gives an overall overview of the system's implementation. It thus provides both functional and non-functional requirements that the design needs to comply with, which will be later on tested in Chapter 5.

3.1 System Specifications and Requirements

The main goal is to build a reliable, scalable and ultimately maintainable end-to-end data pipeline from scratch, addressing dynamic aggregations¹ and a dynamic interface to query data. In this dissertation, with a context on trade surveillance, a prototype of such a system is implemented which serves as a proof-of-concept. In Chapter 5, we will evaluate how well the requirements, described in this chapter, are met.

The three stages that comprise an end-to-end data pipeline are data collection, data aggregation and data analysis. As such, the system design should allow to collect and receive trading data, process it and perform running calculations to unlock new insights. These insights can thereafter be queried by users for query purposes and by the server, which the project implements, to run subscriptions and display real-time charts (GraphQL).

As mentioned in Chapter 1, trade surveillance relies heavily on triggering anomaly detection alerts (suspicious activity) from trading activity. Hence, by shifting from static to calculating dynamic thresholds, the pipeline aims to capture fluctuations in the stream's price values across short periods of time (from now on referred to as *windows*). Fixed thresholds can quickly become outdated as market conditions evolve, as they fail to capture real-time activity patterns. This makes the newest incoming data valuable compared to historic data which were used to detect outliers.

¹Dynamic aggregations will allow to calculate dynamic thresholds which will be updated every timed-window using Kafka Streams (more on Implementation Chapter)

Functional Requirements

The system should therefore comply with the following functional requirements:

- **FR1 - Query datasets at any point in their history**

The client should be able to query data regardless of the time range, and retrieve these back. Therefore, incoming data should be stored and remain available while the system is running. In a trade surveillance system, when thresholds are exceeded, an alert is generated which appropriate parties will query for further investigation.

- **FR2 - Message preservation**

The system should be able to preserve messages even if the network connection between the producer and consumer is lost temporarily. In the event of adding new producers to the system (scaling up), they should be able to recover older messages.

- **FR3 - Process data in real-time**

The system should consider possible bottlenecks when triggering suspicious activity as these will be triggered on the fly. The system should efficiently trigger and classify activity based on pre-defined logic. This means the system should be able to have fast access to historic dynamic thresholds while receiving incoming data and avoid possible bottlenecks [17].

- **FR4 - User-initiated Queries support (Queries)**

The system enables convenient retrieval of data. Queries are useful for investigating specific trading activity that is suspected of being fraudulent.

- **FR5 - Server-initiated Queries support (Subscriptions)**

The system enables to execute subscriptions to provide current real-time values for visual monitoring (real-time charts).

- **FR6 - Addable data sources and formats**

The system should be able to allow data enrichment and flexibility to add additional new business logic, for instance. The system should store data for the server to query but should allow to incorporate additional sources of input data without impacting existing data and processing pipelines.

Non-Functional Requirements

The system should then comply with the following non-functional requirements:

- **NFR1- The system should be scalable**

By adding system resources, the system can sustain its performance even under increased load. Trade surveillance heavily relies on receiving trade information, and as such, increased loads should be handled without compromising the system's performance. This is because overnight activity greatly differs from peak trading,

where current data patterns can experience sudden price changes in response to market conditions, news events, or other factors (price swings that are difficult to predict).

- **NFR2 – The system should be fault tolerant**

The system should be able to retrieve missed information should any subscriber face temporary errors.

- **NFR3 – Evolutionary abilities: Generalization and Extensibility**

The system should remain efficient in case of changing requirements (for example changing data sources or programming languages used). On top of that, the system should be able to adjust to evolving requirements or unanticipated use cases.

- **NFR5 – Low Latency**

Should the system's incoming orders per second increase (scaling up throughput), it should address latency issues (keeping it relatively low) when classifying orders.

3.2 Streaming Data Processing Architecture

Since the proof-of-concept tackles the main stages in a data pipeline (data collection, aggregation and analysis), the implementation is structured in two main components: a streaming platform (which will use Apache Kafka) and an interface for querying, subscribing and mutating data (which will use GraphQL).

After having laid out the main requirements, the project represents an end-to-end data pipeline of stream processing on a real trading data to capture potential illegal activity on the fly as a first step. Therefore, researching cross-domain systems which dealt with large volumes of data and stream processing were investigated. Thinking about the system's architecture and how each of the components were arranged and communicated with each other was essential to avoid unnecessarily complex layouts.

3.2.1 Challenges on Streaming Application

The main challenge faced on the initial stage concerned both the data collection and data aggregation process. It involved understanding how to move away from current batch processing to handle real-time and historical analytics. This would allow to compute and classify incoming trading activity as genuine/suspicious in near real-time while performing windowed-aggregations to dynamically shift the threshold value to detect outliers on data that is subject to rapid and unpredictable changes.

These latter challenges have been addressed by different proposed streaming architectures in the literature, namely the Lambda architecture discussed in papers [22][10] and the Kappa Architecture discussed in papers such as [57]. These two architectures are widely accepted when working with streaming data, which will be discussed below as the two main possible choices for the project.

3.2.2 Possible Streaming Architectures

While both the Kappa and the Lambda architectures are possible approaches to scalable and fault-tolerant data streaming systems, the main difference lies in the way they handle data processing.

The Lambda architecture (Figure 3.1) allows to process real-time and batch data by using both a batch processing layer and a speed processing layer. On the contrary, Kappa architecture processes only real-time data, which allows to eliminate the batch processing layer entirely.

The Lambda architecture uses two separate data stores for batch and real-time data processing while the Kappa architecture uses a single immutable data store for both. This essentially means that data is never deleted or updated once it is written.

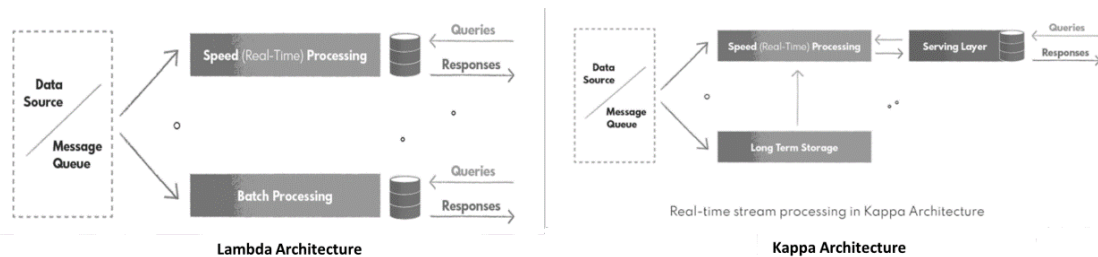


Figure 3.1: Lambda vs Kappa Architecture. Sourced from [45].

Since the system has to compare the incoming order's price with a threshold (which is continuously being updated), the lambda architecture deemed a better approach than a Kappa architecture. It was first thought that two main layers were needed; one to compute and store windowed aggregations that would allow to dynamically update thresholds, and another real-time layer that would allow to classify trading activity by comparing it to the calculated windowed thresholds.

However, the need for the system to provide processed data with a low latency highlighted the need for a single stream processing pipeline (in depth discussion in Chapter 4), as constantly querying and fetching dynamic thresholds proved to encourage bottlenecks.

3.3 High Level Overview of the Prototype's Architecture

After reasoning the overall architecture chosen to build the data pipeline, this section is dedicated to providing an overview of the technologies selected for the project and the rationale behind their choice.

Main Technologies Used

The system uses several technologies as shown in Figure 3.2. These are the following:

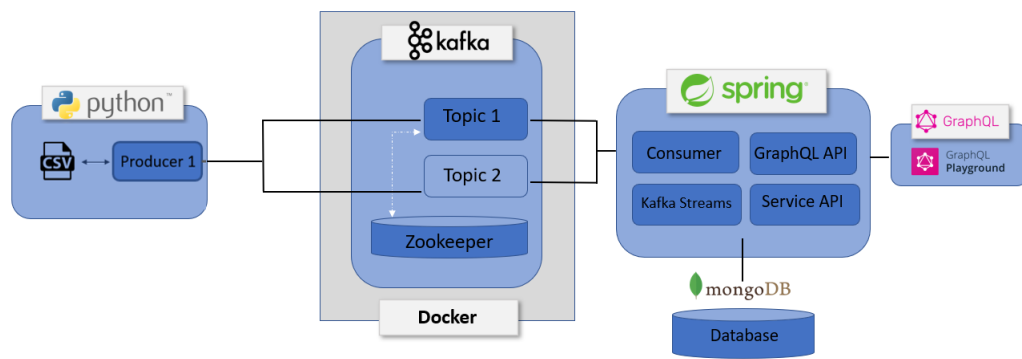


Figure 3.2: Technologies used in the System

- Message System – Apache Kafka :** Kafka is a distributed messaging system able to handle large volumes of real-time data. It acts as a main distributed messaging system, transporting messages between the python script that sends orders and the server in Spring Boot that analyses them (Java). Kafka is not the only open-source messaging system available to use, with RabbitMQ and ActiveMQ providing similar functionalities. However, previous research papers [14] highlight the suitability of Kafka for processing and analysing large amounts of real-time data. RabbitMQ, on the other hand, may struggle to handle high data volumes and may not ensure message persistence [49]. Kafka is designed to provide high throughput and low latency, making it ideal for real-time data streaming use cases. RabbitMQ, while still performant, may not provide the same level of performance as Kafka.
- Query Language – GraphQL:** GraphQL is used as the main query language to provide a flexible and dynamic API to retrieve data. The second part of the implementation involves developing an interface for querying and subscribing data (which will use GraphQL). The main underlying reason for choosing a dynamic GraphQL API over the traditional RESTful API is its flexibility; clients can specify exactly what data they need and get only that data in response, allowing for efficient data retrieval improving the overall investigation on each suspicious order. This decision is supported by the literature papers showing how GraphQL APIs outperform traditional RESTful APIs [6].
- Database – MongoDB:** MongoDB is chosen as the main database, used to store both genuine and suspicious orders after being classified. Data (orders) is stored in the form of *documents*, which is a data structure that the Non-SQL DB uses for each data unit; a collection of documents form a *collection*. The database is queried through the Spring Boot Service API to resolve GraphQL requests. Since FR6 (Addable data sources and formats) requires the system to accommodate for future data enrichment, choosing a Non-SQL database would allow to add new data without needing a pre-defined schema for it; this ensures future flexibility.
- Programming Languages-** Java (SpringBoot), JavaScript and Python

Overall System Overview

Figure 3.3 showcases the overall system overview used to implement the above technologies, and develop the end-to-end data pipeline.

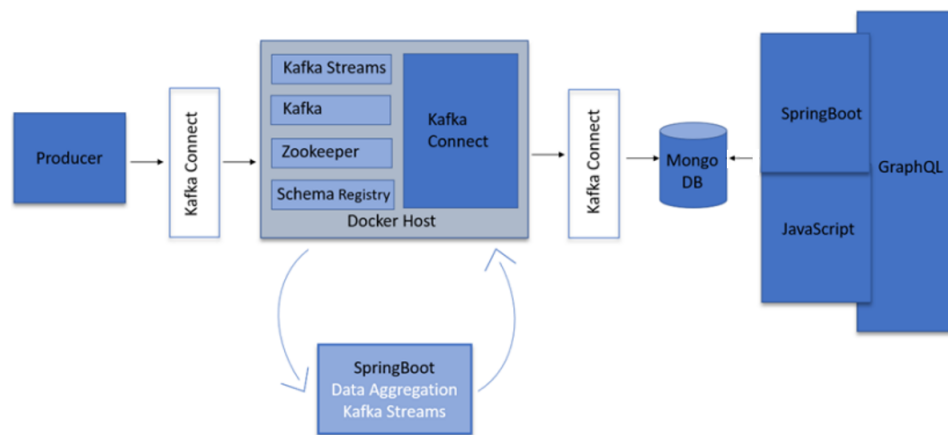


Figure 3.3: Overall System Overview

The proposed high level architecture thus follows the Kappa Architecture, which allows to perform both real-time (order classification) and (near real-time) batch processing (windowed-aggregations) with a single technology stack. It is based on a streaming architecture in which an incoming series of data is first stored in the messaging system Apache Kafka. Figure 3.4 shows the overall data workflow.

The system first has a producer (developed in python), which reads the `csv` file simulating incoming orders from a trading floor. This producer is connected to Apache Kafka, which is running in docker. Running Kafka in docker allows to easily set up a Kafka environment on a local machine without installing and configuring locally.

Kafka allows to store data locally in channels known as *topics*. The system makes use of two different topics to store data; the first one is used to store the incoming stream from the `csv` python producer and the second topic is used to store the latest updated value for the dynamic threshold (updated every timed-window).

Data is ingested into the messaging system Apache Kafka (stored in topic 1), which acts as the input stream for the pipeline. From there, a subscriber (which is developed in Java in Spring Boot) will read the data by subscribing to this topic through a consumer, and transform it into a readable format (serialized to JSON). As soon as data is received in the implementing class with the running logic in SpringBoot, a series of computations (described in Chapter 4) will be performed, used in order to classify the order as genuine or trigger an alert and classify the order as suspicious.

In parallel, the system will continuously keep track of dynamic threshold values to determine how to benchmark illegal/suspicious activity by keeping track of windowed aggregations that will determine the threshold values. This is done in Spring Boot using kafka Streams library, which are stored back into a kafka topic 2 for caching purposes (described in Chapter 4) and ensure threshold values are updated automatically.

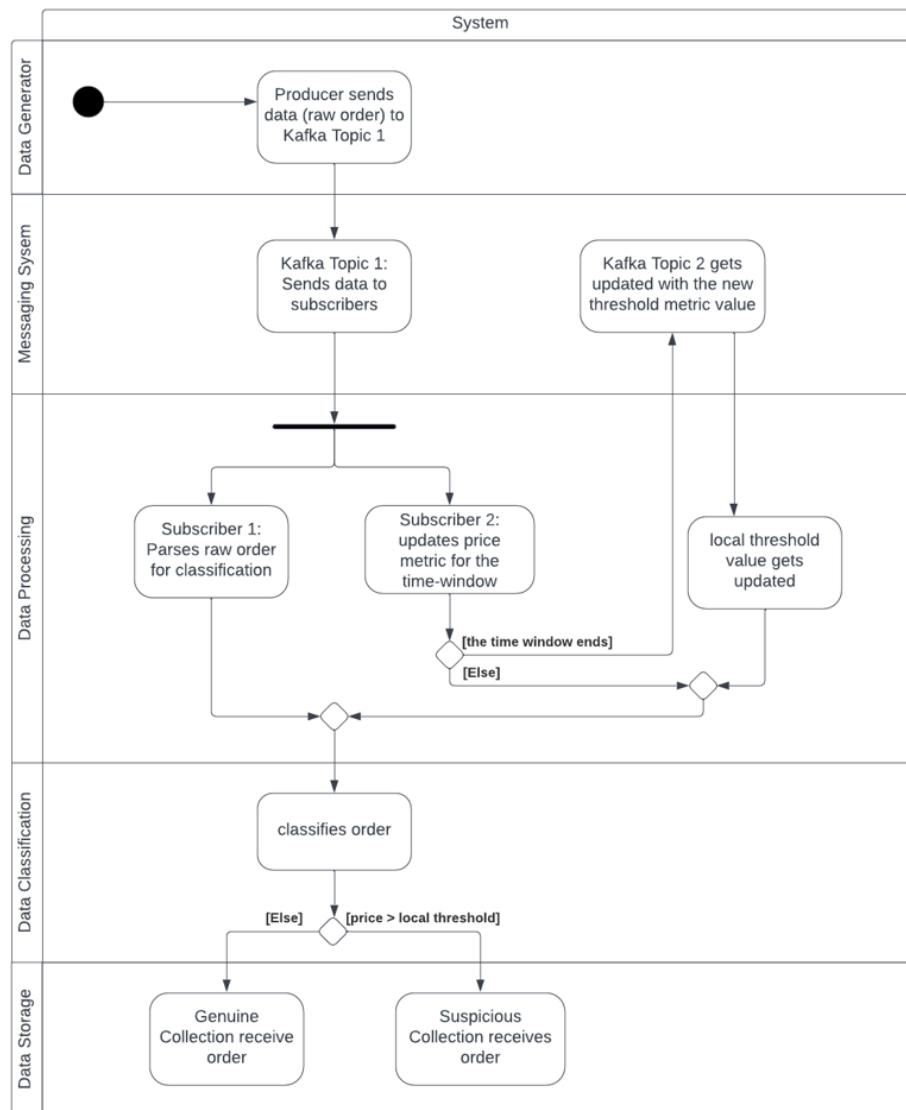


Figure 3.4: UML Activity Diagram Showing Data Workflow

These dynamic threshold values are stored in Kafka while classified orders are then stored into a non-SQL database. A query interface is then developed to connect GraphQL with the database and perform queries. On top of this, the data pipeline also allows to run a JavaScript server to showcase real-time charts (further explained in Chapter 4).

Why do we not query Kafka directly (and query the database instead)? When it comes to querying data using GraphQL, there were a few challenges to consider. Initially, the system design handled queries from GraphQL by fetching data directly from a Kafka topic. Thus, it was using the information that producers sent into a topic as the main source for data fetching. But there are several issues, namely related to query flexibility. Querying data stored in a Kafka topic meant that GraphQL query's flexibility would be greatly affected. Kafka topics do not provide native support for performing aggregations on the data. This meant that only simple queries could be carried out, mostly regarding retrieving the last logs for a given timestamp.

Chapter 4

Implementation

4.1 Getting Started - Setting up Docker

Docker allows to containerize apps. This essentially allows to package, deploy, and run apps virtually instead of installing all dependencies locally. Using docker allows to create docker images that contain the needed applications and all their dependencies, and then use those images to create and run containers. These containers are known to be lightweight, portable, and self-contained environments for running these applications, allowing therefore to isolate environments.

The proof-of-concept needs to setup an isolated environment to run Apache Kafka, and although both Linux VMs and Docker can be used, Docker provides a more lightweight and efficient containerized environment. This approach offers a more efficient use of resources, as containers are smaller and faster to start up than VMs.

With Docker, the most common way to run Kafka is by setting up a YAML file¹ (as seen in Figure 4.1.) that allows to define the components needed.

Docker Images: Components Needed

- **Zookeeper:** Zookeeper is used namely for cluster coordination. Zookeeper handles kafka's cluster synchronization and configuration, allowing developers to focus on the core application logic. It does so by ensuring it manages broker connections to particular clusters [108]. For the project's yml file, we ensured that zookeeper was kept alive in port 2181.
- **Kafka:** Apache Kafka was set up in the Kafka container. The default `Broker_ID` was set to "1", and the external connection to Kafka topics (for subscribers and producers) was set to be port 9092. The `Broker_ID` is used as a single identifier assigned to each broker in a cluster.

¹Used to define and configure a container image and its associated properties.


```

5 services:
6
7   zookeeper:
8     image: wurstmeister/zookeeper
9     hostname: zookeeper
10    container_name: zookeeper
11    ports:
12      - "2181:2181"
13    environment:
14      ZOOKEEPER_CLIENT_PORT: 2181
15      ZOOKEEPER_TICK_TIME: 2000
16      mem_limit: 200m
17    restart: always
18
19
20   kafka:
21     image: wurstmeister/kafka
22     hostname: localhost
23     container_name: kafka
24     ports:
25       - "9092:9092"
26     depends_on:
27       - zookeeper
28     environment:
29       KAFKA_BROKER_ID: 1
30       KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
31       KAFKA_ADVERTISED_LISTENERS: INSIDE://kafka:9092,OUTSIDE://:9092
32       KAFKA_LISTENERS: INSIDE://kafka:9092,OUTSIDE://:9092
33       KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: INSIDE:PLAINTEXT,OUTSIDE:PLAINTEXT
34       KAFKA_INTER_BROKER_LISTENER_NAME: INSIDE
35       KAFKA_CONFLUENT_SCHEMA_REGISTRY_URL: http://schema-registry:8081
36       CONFLUENT_SUPPORT_METRICS_ENABLE: "false"
37     mem_limit: 1500m
38     restart: always

```

Figure 4.1: Docker-Compose yml file

4.2 Data Collection

4.2.1 Dataset

The context used to develop this prototype is based on trading activities in financial markets. As such, the system needs to ingest data related to trading activities, including Trade Data and Market Data. For Trade Data, it refers specifically to information about individual trades, such as volume, instrument, and timestamp. Market data includes information about market conditions, specifically maximum and lowest price at trade time.

Institutions which carry out trading activity usually consider their trading activity highly confidential, and is not publicly available. This poses constraints, as the main goal ingest data that simulates to real trading activity patterns, and inject outliers to showcase the system's capabilities (shown in Figure 4.2).

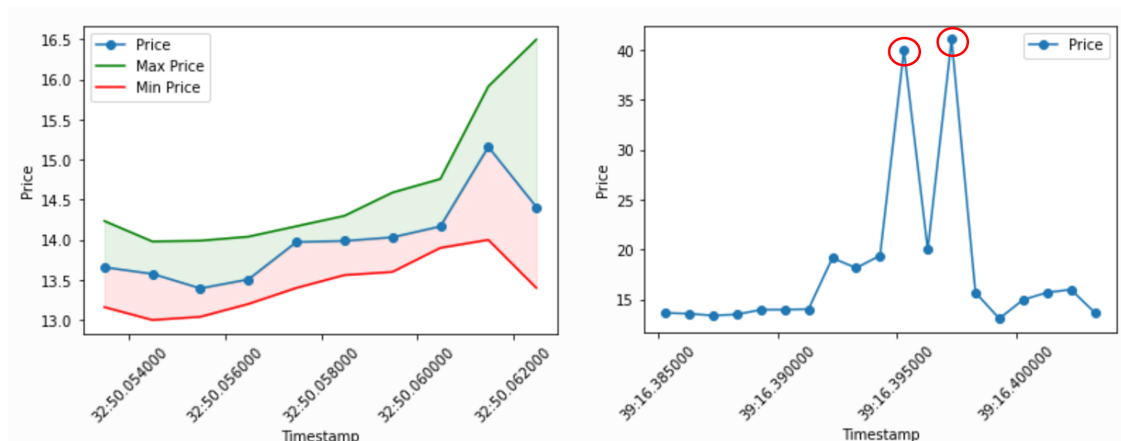


Figure 4.2: Dataset used

With this being said, there are some cases where banks or other financial institutions may make their data available to researchers or third-party vendors. This is the case with Kaggle, which hosts a large number of public datasets related to financial markets, including historical market data.

Kaggle hosts “Huge Stock Market Dataset” (shown in Figure 4.2), which provides the complete historical data of daily prices and trading volumes for every stock based in the US. The projects ingests data from the `amj.us` , which visually inspecting it proved to be highly tradable, and experience fluctuations in its price values across time.

4.3 Kafka Producer

In Kafka, a producer is responsible for sending the data records into a Kafka topic. In the context of Trade Surveillance, our producer reads trades from the dataset mentioned in 2.1, simulating a trading floor by generating and publishing trade data to a Kafka topic.

To carry out this data ingestion, a *data_generator* Python script was created. This python script makes use of the built-in `csv` module to read the latter `csv` file, in which each row represents a given trade (Figure 4.2). Besides reading the `csv` data, this python script establishes a Kafka connection to send data into a topic. To interface with Kafka, the ‘`kafka-python`’ library was utilized as a client.

4.3.1 Data Manipulation: Generating Outliers and Price values

The dataset mentioned did not pertain to a particular Institution, but rather represents trading activity for each stock. As such, only highest and lowest prices are shown (public market data) but not the actually price that traded for each row. As such, the python script makes use of ‘`random.uniform`’, to generate a random floating-point number that lies between the maximum price and lowest price in each row. This number is set to be the price at which the trade was executed.

Since the first part of the project aims at also shifting from current static thresholds to a dynamic approach to monitor activity, there should be several abnormally large or frequent trades to test the approach.

Hence, the python script randomly generates abnormal trades that fall outside of the expected range of trading activity. This simulates a given trading desk manipulating the market.

The rows in the `csv` file are injected every second, simulating real-time activity.

4.3.2 Data Serialization

Sending data to a Kafka topic requires serialization. This process requires converting the data structure from the `csv` file into a format that can be transferred over a network. In Kafka, these messages need to be encoded into a byte array or binary format before sending them to the Kafka cluster. Although Kafka supports different serialization

formats², the project makes use of JSON format as it best suits our use case. Hence the data will have the following format seen in Figure 4.3 :

```
{
  "type": Message,
  "Order_id": ID,
  "Field": [
    {
      "timeStamp": Time
    },
    {
      "Order_id": ID
    },
    {
      "Price": Float,
      "High": Float,
      "Low": Float
    }
  ]
}
```

Figure 4.3: Schema Definition of the Dataset Used.

4.4 Data Processing

4.4.1 Spring Boot Server

The system uses Spring Boot as the main layer for data processing and perform aggregations on it before sending it to the database.

The role of the spring boot app is to serve as the main connection layer to consume records from Kafka topics. On top of this, the project uses Spring Boot to perform aggregations and compute dynamic thresholds using Kafka Streams in order to classify incoming activity and inject it into MongoDB. Lastly, the Spring Boot server also provides the necessary GraphQL Resolvers to provide a GraphQL-based interface for the client application.

4.4.2 Consuming Records from Kafka Topic

One of the main goals is to move away from batch processing and classify orders as suspicious or genuine (this is done through dynamic thresholds explain in section 4.5.1).

Hence, consuming data from Kafka topics can allow you to move from batch processing to near real-time processing by receiving data as soon as it is publish into the topic (which acts as a channel). To do so, Java allows to use Kafka Listener framework (Figure 4.4) which subscribes to a given topic (*raw_topics*) and initializes an instance of an *Order* for each message the producer sends into the topic.

²This includes JSON, Avro, BSON, Apache Thrift, Protocol Buffers or MessagePack.


```
@Component
public class Consumer {

    @KafkaListener(topics = "raw_orders", groupId = "groupId")
    public void listenRawRecords(String order) {
```

Figure 4.4: Kafka Listener for Incoming Data.

”Order” Java Class

Since we serialized data to input into Kafka, the project has an *Order* Class with four main attributes, the same ones the producer reads from the `csv` file. These are: ID, High, Low, Price, Volume and Timestamp.

Hence, the main usage is to parse the JSON input into an instance of an *Order* class (for later usage in classification). Hence, the class has getter methods to later call its attributes when classifying the order/activity.

4.5 Data Aggregation – Dynamic Thresholds

Once data has been consumed by our Spring Boot Listener, the transition to (near) real-time Trade Surveillance would only occur if potential illegal activity is captured as soon as it takes place. To do so, every incoming order is checked upon specific threshold values (more on next section).

4.5.1 Why Dynamic Thresholds

Monitoring financial transactions most frequently involves using fixed thresholds which that could indicate possible suspicious or fraudulent behaviour. These thresholds are part of metric type alerts known as threshold-based, which will trigger by checking if a particular value of the incoming order crosses or exceeds this value. These thresholds may be based on factors such as trade volume, trade frequency, trade price, or other relevant variables. When these thresholds are exceeded, an alert is generated and sent to the appropriate parties for further investigation.

The main underlying issue that has sparked interest among the industry is that although these static thresholds can be useful for detecting some types of suspicious activity, there are potential drawbacks to this. Since they are fixed, they can quickly become outdated as market conditions evolve, as they fail to capture real-time activity patterns [27]. They are prone to generate a high number of false positives, as legitimate transactions may trigger alerts, leading to wasted time and resources for investigation [27]. Additionally, these threshold values are only updated after, and if, threshold calibration is deemed appropriate after periodic manual review [43].

Originally, having fixed values provided an approachable way of monitoring transactions for companies with limited resources. However, the release of Kafka, among other

streaming platforms, has certainly made it easier to process and analyse real-time data streams.

As such, the project aims at employing dynamic threshold-based alerts, and develop a simple algorithm that would allow to recognize anomalies within trading activity. This essentially would allow to prevent noisy (low precision) or wide (low recall³) thresholds that don't have an expected pattern. Dynamic threshold would help configure up metric alerts using high-level concepts without extensive domain knowledge about the metric.

Conceptual Challenge: How to implement Dynamic Thresholds and which Streaming Framework to use

The main goal is shifting from static thresholds to dynamic ones, and employ an appropriate architecture to do so. Dynamic thresholds are designed to adjust and update in real-time as new trading data comes in, to enable more accurate and effective monitoring, adapting to changes in market conditions and individual trading patterns.

This poses a different perspective to current approaches in static threshold calculations, as these values could be calculated once all data has been fully ingested (and hence follow a batch workload approach). The main challenge therefore is understanding which streaming framework to use, which would allow to simultaneously receive, calculate and update new values according to current trading patterns.

In order to understand which framework fit best our use case, it involved carrying out an overall understanding of open-source streaming frameworks available to use.

MongoDB recently release MongoDB Change Streams, which was originally launched in 2017 [40] to deal with a real-time stream of changes in a database, collection, or deployment [40], and would notify our current application of any changes to the data. Since our system makes use of MongoDB to store trading activity, it deemed appropriate to make use of MongoDB Streams *watch()* method to compute and update the overall price value mean every time an incoming order was received.

If a trade price is significantly outside the expected range of prices for a particular security, this may indicate that the trade was executed under unusual circumstances and may require further investigation.

However, there were certain subtleties which lead to migrate to the current framework (Kafka Streams). While MongoDB certainly allows to watch streams of data and perform aggregations, it does not have native support for windowed operations. MongoDB's query language allows to efficiently retrieve individual or collections of documents, but is not ultimately designed to perform running calculations across fixed-length windows. This completely dismisses the original purpose, which is to use dynamic thresholds to adapt to current patterns. Hence, these patterns could only be captured if only the most recent orders were averaged (and not the entire collection of documents): a phenomenon known as *windowing*. Since it's a document-oriented database, a possible way to perform windowed operations in MongoDB is by writing custom code, which proved to be very impractical;

³A low recall means that threshold values are set too high, and suspicious activity is not being flagged.

Using Kafka Streams to Perform Window Operations

Kafka offers a client library known as Kafka Streams, which is part of the Kafka ecosystem. Kafka Streams' library allows to build processing applications that perform operations such as filtering, aggregating or joining data in real-time.

As such, the project makes use of it to develop a near real-time aggregation (which will compute the dynamic threshold) to process and analyse streaming data from our incoming Kafka topic (Figure 4.5).

In Kafka Streams, windowing aggregations are performed using the concept of a "window," which is a fixed-size or time-based segment of a stream. Previous work on computing running aggregations highlighted Kafka Streams Windowing operations [17][48], proving to be an efficient way of performing running aggregations with streaming data. Kafka supports four main types of windows, namely Tumbling, Sliding, Hopping and Session windows. The project makes use of Tumbling windows, employing the important characteristic of having non-overlapping windows to calculate new averages every few seconds.

The project uses dynamic thresholds based on a statistical measure of price values for every timed window in Kafka. This was accomplished by using a combination of Kafka Streams and a streaming aggregation function that computes the statistical measure $q3 + 1.5 * IQR^4$ over each incoming time window.

This statistic will be computed every window, and will be the threshold used. The use of $q3 + 1.5 * iqr$ as a threshold for outlier detection is a commonly used method in statistical analysis. Q3 discriminates the top 25% of the data and IQR is calculated by subtracting the first quartile (Q1) from the third quartile (Q3).

4.5.2 Methodology to Implement Dynamic Threshold using Kafka Streams

- Using the previous kafka listener for incoming data, Kafka Streams consumes trades from the Kafka topic using a KStream. A KStream represents an unbounded, continuously updating stream of data records in Kafka.
- The Spring Boot application has a TradeStats Class, which holds three main methods used in the running aggregations, namely *ComputeMetric()*, *ComputeSum()*, and *add()*.
- The Kstream aggregation is constantly running under the main() Spring Boot method (figure 4.5), as it is iniliased as soon as the server starts running. It uses the *map* operator to transform the KStream into a stream of trade prices. It then uses the *windowedBy* method to create a tumbling window every second. By using the *aggregate* operator, it allows to initialise the TradeStats class for every inbound order.

⁴IQR is Interquartile Range: measure of the spread of a dataset. Q3 is the median of the upper half of the dataset (third quartile). This measure is a standard outlier detection metric for upper limit.

A time windowed serde⁵ is responsible for serializing and deserializing data within a time window. Every time window will have to be serialized using a supporting method named *TradeStatsSerde()* which is a customised serializer developed to create time windowed serdes for the TradeStats class.

```
KStream<Windowed<String>,String> stats = source
    .groupByKey() KGroupedStream<String, String>
    .windowedBy(TimeWindows.of(Duration.ofSeconds(2)))
    .advanceBy(Duration.ofSeconds(1))) TimeWindowedKStream<String, String>
    .<~>aggregate(TradeStats::new, (k, v, stats1) -> stats1.add(v),
        Materialized.<~>as("trade-aggregates")
        .withValueSerde(new TradeStatsSerde())) KTable<Windowed<String>, TradeStats>
    .toStream() KStream<Windowed<String>, TradeStats>
    .mapValues(TradeStats::computeMetricForThresholdValue);
```

Figure 4.5: Computing running Price Thresholds using Kstreams. Developed by me, adapted from Apache Tutorial Documentation [29].

Caching Challenge and Solution Testing

After calculating running averages to be used as dynamic thresholds every timed-window, the system should be able to retrieve an updated threshold to compare against new incoming orders.

The challenge lies in caching (store temporarily) this threshold such that the system would not need to constantly query the value to check if every incoming order exceeds it. Constantly fetching the threshold would allow to ensure that the threshold used for comparison is updated. However, a message system can indeed become a bottleneck itself when carrying out outlier detection in Trade Surveillance if the rate of messages sent increases (and queue them up). Constantly querying a threshold numerous times every second/millisecond will eventually become a bottleneck as seen in Figure 4.6.

Messages/Second	Non-Caching Mean Latency (seconds)	Caching Mean latency (seconds)
100	0.206765	0.019676
500	2.876642	0.165723
5000	4.24	1.43
10000	6.510334	2.88

Figure 4.6: Caching vs Non-Caching Latency Tests Undertaken

The project thus injects every new calculated threshold into Kafka Topic 2 (named *Kafka.Threshold*), and sets up a new listener which is subscribe to this topic. The system has a local variable name *current_threshold*, which is updated automatically every time

⁵Serde is a Rust library used for serializing and deserializing Rust data structures to formats including JSON

the listener receives an updated value (Figure 4.7, meaning that the time window has finished and a new value has been injected into the Kafka Topic. This way, the system ensures there is no constant querying to an outside database (as the value is stored locally), and it ensure it is continuously updated by subscribing to a new Kafka topic which receives a new value every time the time window ends.

Tests were run by varying the load frequency received per second, and tested the mean average latency (measures the time delay between when a trade is executed until it is processed by the surveillance system to trigger an alert). Results show that this caching solution, although not fully optimal, it does reduce the linear latency by a factor of more than x5 for same input rate streams as showcased in Figure 4.6 (more on Evaluation Chapter 5).

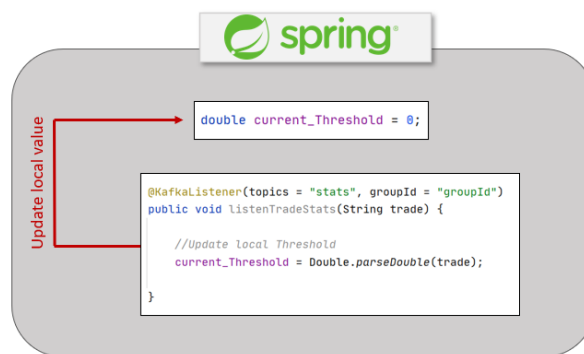


Figure 4.7: Caching Solution.

Running Alerts and Injecting Data into MongoDB

After setting up the streaming architecture and the additional layer to compute dynamic thresholds, the algorithm checks the incoming data against the these latter ones through *threshold-based* alerts.

For this prototype, a simple threshold-based alerts on the price value was setup, as shown in figure 4.8.

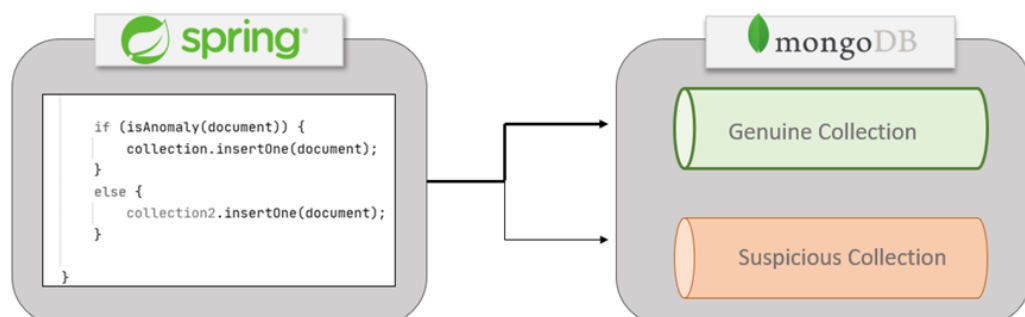


Figure 4.8: Data Flow from Alert to MongoDB.

If the price value for the incoming order exceeds the threshold value, an alert is triggered. This means that the order is classified as suspicious, and sent into the *Suspicious* collection in MongoDB. Otherwise, the order is considered genuine, and sent into the genuine collection.

4.6 Query Language: GraphQL Interface

The second component of the project involved developing an interface using GraphQL language to retrieve information from MongoDB. The main underlying reason is that data analysis is the last component in a data pipeline; When these thresholds are exceeded, an alert is generated and sent to MongoDB where, should the system be deployed, appropriate parties will query this data for further investigation. Hence, the GraphQL interface will serve as an API, to provide a flexible and efficient approach for it. Spring Boot is used as the backend server to implement the GraphQL Queries and Javascript for GraphQL subscriptions.

GraphQL allows for three main types of processes, namely queries, subscriptions and mutations. This section will involve describing and showcasing these processes, and how they have been implemented.

Code-First vs Schema-First Approach Decision

There are two main approaches to develop a GraphQL API: Schema-first, in which resolvers are generated after a schema has been defined, and Code-first, which involves writing the code first and then generating the schema from it.

Both approaches have their own advantages and disadvantages, and choosing between them greatly depends on the use case. A code-first approach offers more flexibility, enabling more dynamic and iterative development. Any changes to either the schema or the code is done directly on the code, without having to modify a separate schema definition.

A Schema-first requires a clear blueprint for the API, needing therefore a strict Schema Validation. This allows to pinpoint errors earlier, useful for projects in which defining the schema first can help to establish a clear structure and hierarchy for the data.

The project implements both GraphQL Queries using a Code-first approach and Subscriptions using Schema-first.

The underlying reason for this is given by the type of methodologies employed when developing the system. GraphQL queries were developed iteratively, as the data format changed while the Kafka environment was being finalised in parallel. The code for query resolvers was constantly modified to accommodate new changes, and therefore a Code-first approach allowed for rapid changes to both the data model and the GraphQL schema in Java.

On the other hand, subscriptions were used to generate real-time charts in JavaScript, which required a clear well-defined structure and format. As such, these were generated using a Schema-first approach.

4.6.1 Queries

The project development followed a number of steps to develop Code-first GraphQL queries by fetching data from MongoDB into Spring Boot.

4.6.1.1 Defining data model using Java classes

Even though a Code-first approach does not initially require a schema definition, the resolvers, which are in charge of retrieving the data, need to be able to map structure of the data that is stored in the database to a Java class.

As such, the Spring Boot application counts with a class called `DocumentScalar`, which represents a document matching those stored in a MongoDB collection.

This class is crucial to easily interact with the database, since Java inherently receives data from MongoDB as a built-in `Document` class, and the system parses it as a customised `DocumentScalar` class to add the trade order attributes (Price, High value, Low Value, ID, Volume).

4.6.1.2 Defining Interface Service for MongoDB data fetching

The system communicates with MongoDB by using a `UserService` interface. This is a Spring Data MongoDB library which has been further customised (since it only provides a high-level abstraction for interacting with MongoDB) by adding methods that carry out the resolvers' requests, as seen in figure 4.9.

```
public List<DocumentScalar> getOrders() throws ExecutionException, InterruptedException {  
    List<DocumentScalar> listDocu = new ArrayList<>();  
    Gson gson = new Gson();  
    FindIterable<Document> documentCursor = collection.find();  
    for (Document docu : documentCursor) {  
        listDocu.add(gson.fromJson(docu.toJson(), DocumentScalar.class));  
    }  
    return listDocu;  
}
```

Figure 4.9: Example Method on Interface for MongoDB Connection.

4.6.1.3 Resolvers for Queries

The system needs to implement methods known as *resolvers*, which allow to retrieve data from MongoDB and return it in the format specified by the user.

These are methods using the `@GraphQLQuery` annotation to map the resolvers to the corresponding queries.

Simple Code-First Queries

The system defines simple resolvers such as the one shown in Figure 4.10, which allows the user to select all possible attributes (eg. Price, Volume, High, Low values) for the order with the highest price.

Figure 4.10 showcases the resolver which has been implemented in Sping Boot, and the corresponding GraphQL UI client (GraphQL Playground [24]) to test GraphQL queries.



Figure 4.10: Simple Highest Price GraphQL Query.

Code-First Query Variables

GraphQL queries also accept arguments or parameters, which allow the user to further specify dynamic values to be used in the query.

In order to implement it, two main methods were created; one as the main GraphQL resolver (Figure 4.11) and the MongoDB userService method (Figure 4.12)

```

@GraphQLQuery(name = "getDocumentsAboveMaxPriceVolume")
public List<DocumentScalar> getDocumentsAboveMaxPriceVolume(@GraphQLArgument(name = "Price") String price)
    throws ExecutionException, InterruptedException {

    List<DocumentScalar> ourValues = userService.getVolumes(price);
    return ourValues;
}

```

Figure 4.11: GraphQL resolver for Query Parameters.

```

public List<DocumentScalar> getVolumes(String vol) throws ExecutionException, InterruptedException {
    List<DocumentScalar> listDocu = new ArrayList<>();
    Bson filter = Filters.and(
        Filters.gt( fieldName: "Volume", vol));
    ArrayList<Document> documentCursor= collection.find(filter).into(new ArrayList<>());
    Gson gson = new Gson();
    for (Document docu : documentCursor) {
        listDocu.add(gson.fromJson(docu.toJson(), DocumentScalar.class));
    }
    return listDocu;
}

```

Figure 4.12: MongoDB userService method for Query Parameters.

4.6.2 Subscriptions

GraphQL subscriptions allow to maintain a long-lived connection to the server and receive updates whenever the data changes. This makes subscriptions a feasible application for developing a real-time chart that displays real time charts for the incoming trading data.

4.6.2.1 Real time charts using GraphQL subscriptions

This is an extension to the GraphQL section, as it develops GraphQL subscriptions in JavaScript to display real-time charts in Python. The main challenge involved understanding the basics of JavaScript (since I had not coded in JavaScript before) to establish a WebSocket connection to the server and subscribe to the data stream. The project on JavaScript required installing libraries including *websocket-client*, *subscriptions-transport-ws* or *py-graphql-client*.

- **Defining GraphQL schema, Server setup and MongoDB connection**

This GraphQL project was developed schema-first, since the previous implementation was already finished and the data involved has a clear and defined structure. As such, the schema includes a *Subscription* type. The next stage involved setting up an Apollo GraphQL server with WebSocket support for real-time subscriptions, connecting it to a MongoDB database and listening on port 4000 for incoming requests.

- **Create the Resolver**

Now that the websocket connection is established, the GraphQL subscription needs a resolver to retrieve data and publish the message to a subscription topic. This part is required to notify all subscribers who have subscribed to that topic about the new data (our python script in the next step).

- **Implement the GraphQL subscription in Python to create a chart**

After setting up the GraphQL Subscription, we want to use a python script to subscribe to it and continuously plot the real-time chart whenever the database gets updated. Steps portrayed in Figure 4.13

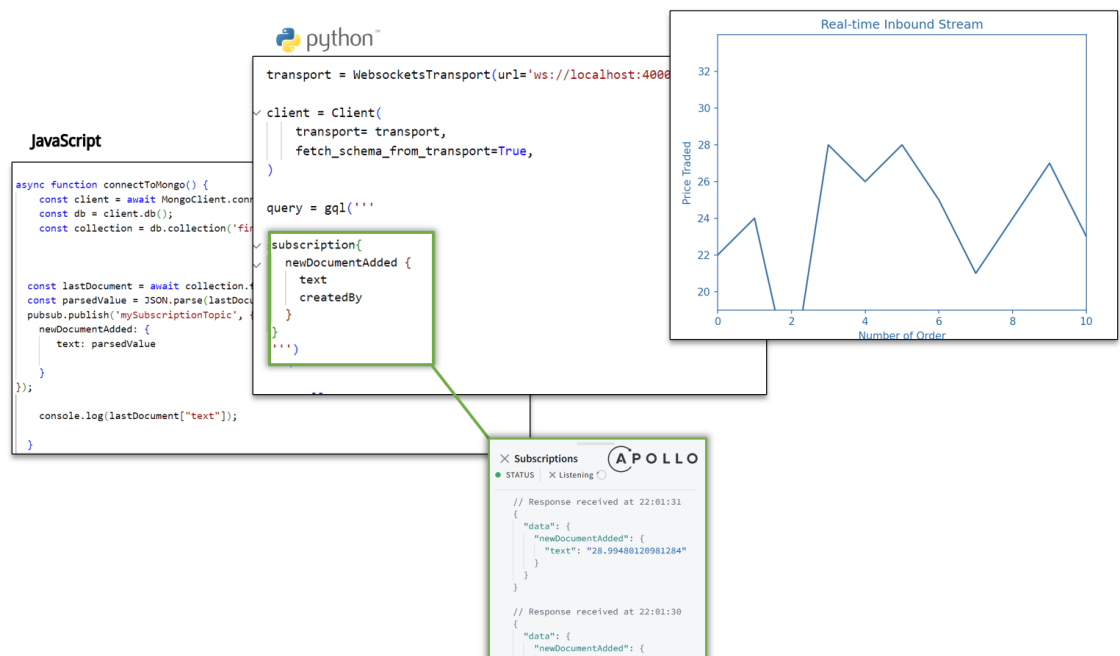


Figure 4.13: Steps Followed to Implement Subscriptions.

Chapter 5

Evaluation

This chapter aims at critically evaluating how the requirements previously laid out in Chapter 3 were met, as well as testing the basic functionality of the system. Several basic Unit Tests were created, targeting both Kafka and GraphQL components. This section will evaluate, and discuss the system's performance and how it aligns with current expectations.

5.0.1 Evaluating Kafka: Functionality Validation - Handling Errors

The project includes several unit tests (see Appendix) to test the basic functionality of the Kafka component in our system. The unit tests run check that the producer is connected to the correct Kafka topic (Unit test 1) . It also tests for possible incorrect serialization of the message data (Unit test 2), and ensures messages are injected and received following the expected ordering (Unit test 3). To test the above cases, a basic producer/consumer was set up, tested using JUnit's *@Test* annotation. Kafka is also tested on handling errors, specifically on Lost Network Connection ¹. Unit test 4 verifies that the consumer has not consumed any messages during the network outage. Unit test 5 tests that once the server is back online, it is capable of retrieving all messages that were sent while the server was down.

5.0.2 Evaluating GraphQL: Functionality Validation

This section ensures Functional requirements 4 & 5 are met. Figure A.6 in the Appendix shows how the system's Queries' resolvers work as intended. Figure A.8 in Appendix A verifies that the GraphQL subscriptions work correctly and returning the expected data.

5.0.3 Evaluating Overall System Performance

The main objective was to build an end-to-end pipeline addressing current gaps between research papers and industry focus (mainly addressing batch workload processing and dynamic thresholding). As such, this project requires meeting basic current functionality

¹ A lost network connection occurs when a consumer no longer receives messages from Kafka due to a network outage.

(tested using unit tests), understanding how the requirements are met, and identifying current strengths and weaknesses from a technical standpoint. After testing the system's basic functionality through unit tests, now the aim is to ensure that it performs as intended. This involves analysing the system's overall performance under different use cases, simulating possible business scenarios on Trade Surveillance.

5.0.3.1 Testing Robustness: Scalability vs Latency

Testing scalability vs latency is key when implementing Kafka. Latency can have a considerable impact on meeting current non-functional requirements (system's performance) when handling increasingly large volumes of streaming data [35].

Kafka was initially chosen given its ability to handle high throughput and a low latency [35]. A key challenge faced when designing the system was ensuring that running alerts would not pose a significant bottleneck to the system. As such, the system aims to optimize performance while minimising the risk of missing potential violations (addressed by using dynamic thresholds).

As mentioned in the Implementation Chapter, the system addresses updating these dynamic thresholds using the proposed caching solution, reducing linear latency by a factor of x5 for same input rate streams, optimising the inbound bottleneck (Figure 5.1). By measuring the latency between the producer and the alert generation, we can determine how long it takes for trade data to be processed by the system and for alerts to be generated. This information can thereafter be used to pinpoint areas for optimisation or improvement and identify any bottlenecks.

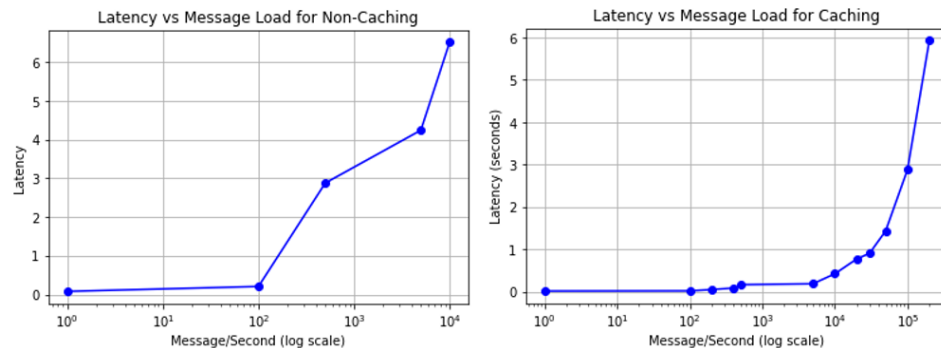


Figure 5.1: Caching vs Non-Caching Latency for different throughput.

In a trade surveillance system, the amount of data being processed can increase rapidly due to changing market conditions or the addition of new trading sources. Testing scalability helps to ensure that the system can handle this increase in data without slowing down or crashing. Hence, different scenarios with different throughput per second were tested (Table 5.1). While it is true that the caching solution helps reduce and maintain a relatively low latency level for considerably large volumes per second, it does delay alert generation by a few seconds for very big throughput. In fact, processing 200k per second has a maximum latency of 5 seconds.

Messages/Second (Throughput)	Max Latency (seconds)
1	0.015676
100	0.020675
200	0.049367
400	0.086347
5k	0.165723
10k	0.187191
30k	0.423977
50k	1.4343
100k	2.8800
200k	5.928531

Table 5.1: Latency Levels Achieved.

Can this be considered real or near-real time performance?

The fact that there is a 5 second delay for throughput of 200k per second does indeed imply that the system will require further optimization should the prototype scale up. The different experiments run on different loads suggests that although local caching proves to successfully keep latency levels relatively low for smaller throughput, it will become a bottleneck for bigger loads. This suggests that in order to meet scalability requirements (for very big throughput), further Kafka configuration is needed. Thus, further investigation on partitioning and distributing data across multiple brokers should help reduce latency by spreading the load across multiple nodes and allowing for parallel processing of message streams.

5.0.3.2 Performance Evaluation: Detecting Suspicious Activity

This section will test a range of values which will allow to reflect the expected variation of use cases.

- **Use Case 1: High Throughput and Big Window Size:** This tests the expected behaviour of detecting a single outlier whose value is higher (in this test, doubled) than the previous injected order. As seen in the first confusion matrix (left Figure 5.2), with 200 trades per second and a 1 minute window that computes the running price threshold (chosen for standard testing purposes), the system efficiently singles out the suspicious transaction while having zero orders as false positive² and correctly discard all other activity as genuine (True negative³) This suggests that detecting outliers using dynamic thresholds works effectively on use cases whose input rate is high and wider threshold windows.
- **Use case 2 - Low Throughput and Small Window Size :** The confusion matrix in Figure 5.2 (right) showcases the system with lower input rate (60 per second) and size window (30 seconds). The high number of misclassified genuine orders suggests that the system is sensitive to the selection of Kafka's window size,

²A false positive occurs when the system generates an alert but that order is found to be genuine.

³The system does not generate an alert and it is not suspicious activity.

which can potentially have an impact on the system's effectiveness. Hence, the system would require further fine-tuning to accommodate to any given specific use case's throughput rate.

	Actual Positive	Actual Negative		Actual Positive	Actual Negative
Classified as Positive	1	0	Classified as Positive	1	238
Classified as Negative	0	23,999	Classified as Negative	0	6,961

Use Case 1: Big Window Size (1 min) and High Trade Rate (200 per second) (measured for 2 minutes)

Use Case 2: Smaller Window Size (30 per second) and lower Trade Rate (60 per second) (measured for 2 minute)

Figure 5.2: System Performance Confusion Matrix.

- **Use case 3 - Detecting Manipulation Patterns:** Trading activity can also be a continuous set of illegal follow-up actions from a trader, that exhibit a certain pattern (e.g. executing multiple consecutive illegal orders to, for example, increase the current price value). As seen in Figure 5.3, the system is able to capture it. However, if these consecutive orders (trading patterns) happen to fall within two different Kafka windows, the threshold will be raised high enough and not detect these last ones.

While the current implementation can easily tackle this issue by excluding these suspicious higher price values when computing the metric each window, this showcases how the system developed serves as a groundwork framework upon which future research can build. Figure 5.3 shows how the dynamic threshold using wider windows are able to capture real time patterns on increasing price values.

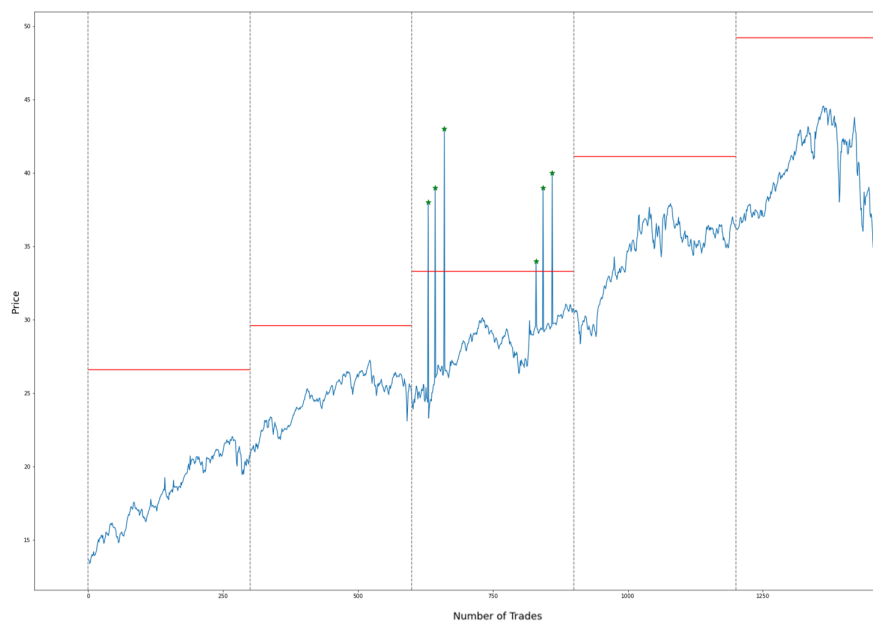


Figure 5.3: Detecting Trading Patterns. Red Lines show threshold value.

Chapter 6

Conclusion

6.0.1 Reflection

By combining Kafka and GraphQL, the system leverages the strengths of both technologies and successfully attains the main goal of efficiently addressing current industry issues. With a context in trade surveillance, the project builds an end-to-end optimised new data pipeline architecture which fulfils the functional and non-functional requirements specified. Given the scarce literature focusing on this topic, the completion of the project serves to bridge the existing application gap between industry and academia research in anomaly detection in finance, and proposes a combined architecture which has yet remained unexplored in the literature.

With the overall goal of exploring the optimal integration of these two technologies, there are two main goals the project successfully tackles. The first one constitutes building a monitoring end-to-end system allowing to shift current industry standards from batch processing to a streaming architecture. Evaluating the system showed that this small-scale solution allowed to shift from batch-processing to near-real time classification with a delay of around 5 seconds on higher throughput. This proof-of-concept hence demonstrates its scalability for future research, while also meeting this required non-functional requirement.

The second goal attained was improving the system's design to compute dynamic thresholds in parallel. While current approaches in the literature use historical values, these fail to capture real-time patterns. There are no recent papers that provide an end-to-end architecture for dynamically monitoring trading activity. Challenges such as understanding how to best dynamically update threshold values in parallel to the system monitoring, and avoid bottlenecks allowed to improve the system's design. The tests performed showcase how the system's choice design reduces the linear latency by a factor of more than x5 for same input rate streams, optimising the inbound bottleneck and ensuring the system remained fault tolerant and with low latency requirement.

This system was developed using an iterative approach, which allowed to accommodate for new perspectives which refined and improved the architecture in accordance to functional and non-functional requirements. The project consisted on a two-staged approach: building a live data pipeline that would allow for dynamic threshold settings

and classify fraudulent activity by performing windowed continuous aggregations using Apache Kafka, and a second stage in which a GraphQL interface is developed to recreate the compliance post-trade analytics. The project was thereafter extended to include GraphQL subscriptions and perform further system evaluation metrics. This project marked my initial experience with using multiple technologies, including JavaScript, GraphQL, Kafka or Docker.

There are several limitations that limit the overall scope of the project. The system's performance evaluation illustrates that both the system's performance and the adherence to non-functional requirements are highly dependent on the parameter settings used during its operation regarding Kafka windows size and input rate (throughput). The data available remains anonymised and unclassified since trading activity datasets remain private to each institution. Therefore, the project incorporates random synthetic outliers, thus the findings may not be fully representative of all potential illegal trading patterns. The system only uses one producer to simulate a trading floor, and while throughput rate was increased to account for variable input loads, further analysis should be performed to test Kafka's cluster configuration.

This project thus serves as a groundwork framework upon which future research can build. While there is no current end-to-end data pipeline on this field using Kafka and GraphQL, the project provides a novel architecture capable of monitoring and flagging suspicious trading activity in (near) real time while calculating dynamic thresholds in parallel, and provides a dynamic query API to inspect fraudulent activity. On top of this, it yields valuable insights as to how a streaming pipeline architecture performs on a trade surveillance context, highlighting the system's sensitivity to kafka window fine-tuning and uncovers further research areas.

6.0.2 Future Work

As mentioned, there are several limiting factors to the project that nonetheless provide further future research opportunities. The system's architecture reduces latency levels compared to an initial design. However, the different experiments run on different loads suggests that although local caching proves to successfully keep latency levels relatively low for smaller throughput, it will become a bottleneck for bigger loads. Hence, further investigation on partitioning and separating incoming orders across different brokers/queues should help reduce latency. This will spread the load across multiple nodes and allowing for parallel processing of message streams. Different criteria could be applied, such as trade type or risk level.

Moreover, the system is a small-scale proof-of-concept which could be further expanded to incorporate multiple different alerts. Given that Java (Spring Boot) was chosen to monitor alerts, its low coupling features will accommodate evolutionary requirements. Further data enrichment could be applied, incorporating account IDs for enhanced personalization and enable tracking and sequencing of orders. This would allow to expand GraphQL queries, and provide a greater comprehensive audit.

Bibliography

- [1] Nice. Markets Surveillance Enterprise Solution Overview. https://www.niceactimize.com/compliance/resource-center/NICE_Actimize_Markets_Surveillance_Brochure.pdf. Accessed: 2022-11-06.
- [2] McKinsey Analytics. The age of analytics: competing in a data-driven world. *McKinsey Global Institute Research*, 2016.
- [3] Rishikesh Bansod, Sanket Kadarkar, Rupinder Virk, Mehul Raval, Rushikesh Rashinkar, and Manoj Nambiar. High performance distributed in-memory architectures for trade surveillance system. In *2018 17th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 101–108. IEEE, 2018.
- [4] Rishikesh Bansod, Rupinder Virk, and Mehul Raval. Low latency, high throughput trade surveillance system using in-memory data grid. In *Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems*, pages 250–253, 2018.
- [5] Gleison Brito, Thais Mombach, and Marco Tulio Valente. Migrating to graphql: A practical assessment. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 140–150. IEEE, 2019.
- [6] Gleison Brito and Marco Tulio Valente. Rest vs graphql: A controlled experiment. In *2020 IEEE international conference on software architecture (ICSA)*, pages 81–91. IEEE, 2020.
- [7] Ruiqing Cao and Marco Iansiti. Digital transformation, data architecture, and legacy systems. *Journal of Digital Economy*, 1(1):1–19, 2022.
- [8] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering*, 38(4), 2015.
- [9] Roman Čerešňák and Michal Kvet. Comparison of query performance in relational a non-relation databases. *Transportation Research Procedia*, 40:170–177, 2019.
- [10] Felipe Cerezo, Carlos E Cuesta, José Carlos Moreno-Herranz, and Belén Vela. Deconstructing the lambda architecture: an experience report. In *2019 IEEE*

- International Conference on Software Architecture Companion (ICSA-C)*, pages 196–201. IEEE, 2019.
- [11] Samarjit Chakraborty, Tulika Mitra, Abhik Roychoudhury, and Lothar Thiele. Cache-aware timing analysis of streaming applications. *Real-Time Systems*, 41:52–85, 2009.
- [12] Confluent. What is Data Streaming? Examples, Benefits, and Use Cases. <https://www.confluent.io/learn/data-streaming/>. Accessed: 2023-02-01.
- [13] Ilias Dimitriadis, Ioannis Mavroudopoulos, Styliani Kyrama, Theodoros Toliopoulos, Anastasios Gounaris, Athena Vakali, Antonis Billis, and Panagiotis Bamidis. Scalable real-time health data sensing and analysis enabling collaborative care delivery. *Social Network Analysis and Mining*, 12(1):63, 2022.
- [14] Philippe Dobbelaere and Kyumars Sheykh Esmaili. Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper. In *Proceedings of the 11th ACM international conference on distributed and event-based systems*, pages 227–238, 2017.
- [15] IBM Documentation. Point-to-point Messaging. <https://www.ibm.com/docs/en/wip-mq/2.0.0?topic=concepts-point-point-messaging>. Accessed: 2023-01-10.
- [16] IBM Documentation. What is a Rest API? <https://www.ibm.com/topics/rest-apis>. Accessed: 2023-03-20.
- [17] Simon Ehrenstein. *Scalability benchmarking of kafka streams applications*. PhD thesis, Kiel University, 2020.
- [18] M Fevzi Esen, Emrah Bilgic, and Ulkem Basdas. How to detect illegal corporate insider trading? a data mining approach for detecting suspicious insider transactions. *Intelligent Systems in Accounting, Finance and Management*, 26(2):60–70, 2019.
- [19] EY. The Future of Trader Surveillance. https://assets.ey.com/content/dam/ey-sites/ey-com/en_gl/topics/emeia-financial-services/ey-trader-surveillance-report.pdf. Accessed: 2022-10-22.
- [20] Ashfaq Farooqui, Kristofer Bengtsson, Petter Falkman, and Martin Fabian. Towards data-driven approaches in manufacturing: an architecture to collect sequences of operations. *International Journal of Production Research*, 58(16):4947–4963, 2020.
- [21] Paul Gibson. Trade Surveillance and how to Improve Accuracy and Detection Rates. <https://ibsintelligence.com/blogs/trade-surveillance-and-how-to-improve-accuracy-and-detection-rates/>. Accessed: 2023-01-09.
- [22] Annabelle Gillet, Éric Leclercq, and Nadine Cullot. Lambda+, the renewal of the lambda architecture: category theory to the rescue. In *Advanced Information Systems Engineering: 33rd International Conference, CAiSE 2021, Melbourne*,

- VIC, Australia, June 28–July 2, 2021, *Proceedings*, pages 381–396. Springer, 2021.
- [23] GraphQL. GraphQL Documentation. <https://spec.graphql.org/draft/>. Accessed: 2023-03-20.
- [24] GraphQL. GraphQL Playground Documentation. <https://www.apollographql.com/docs/apollo-server/v2/testing/graphql-playground/>. Accessed: 2023-02-19.
- [25] Mike Gualtieri, A Rowan Curran, K TaKeaways, and MTBPP To. The forrester wave™: Big data predictive analytics solutions, q1 2013. *Forrester research*, 16, 2013.
- [26] Shenheng Guan, John C Price, Stanley B Prusiner, Sina Ghaemmaghami, and Alma L Burlingame. A data processing pipeline for mammalian proteome dynamics studies using stable isotope metabolic labeling. *Molecular & Cellular Proteomics*, 10(12), 2011.
- [27] Linda Haelsen. Surveillance: Is Real-Time the Right Time? https://www.niceactimize.com/compliance/blog-surveillance_is_real-time_the_right_time.html. Accessed: 2023-01-9.
- [28] Sören Henning and Wilhelm Hasselbring. A configurable method for benchmarking scalability of cloud-native applications. *Empirical Software Engineering*, 27(6):143, 2022.
- [29] Apache Kafka. Apache Kafka Streams Documentation. <https://kafka.apache.org/20/documentation/streams/developer-guide/dsl-api.html>. Accessed: 2023-04-01.
- [30] Martin Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. ” O’Reilly Media, Inc.”, 2017.
- [31] Gerard Klijs. Getting started with graphql and apache kafka.
- [32] Alexey Kovtunenkov, Azat Bilyalov, and Sagit Valeev. Distributed streaming data processing in iot systems using multi-agent software architecture. In *Internet of Things, Smart Spaces, and Next Generation Networks and Systems: 18th International Conference, NEW2AN 2018, and 11th Conference, ruSMART 2018, St. Petersburg, Russia, August 27–29, 2018, Proceedings 18*, pages 572–583. Springer, 2018.
- [33] J. Kreps. Putting Apache Kafka to Use: A Practical Guide to Building an Event Streaming Platform (Part 1). <https://www.confluent.io/en-gb/blog/event-streaming-platform-1/>. Accessed: 2023-02-12.
- [34] Armin Lawi, Benny LE Panggabean, and Takaichi Yoshida. Evaluating graphql and rest api services performance in a massive and intensive accessible information system. *Computers*, 10(11):138, 2021.

- [35] Paul Le Noac'H, Alexandru Costan, and Luc Bougé. A performance evaluation of apache kafka in support of big data streaming applications. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 4803–4806. IEEE, 2017.
- [36] Guo-Liang Lee and Chi-Sheng Shih. Clock free data streams alignment for sensor networks. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, pages 355–362. IEEE, 2007.
- [37] Mr Kasper Lund-Jensen. *Monitoring Systemic Risk Based on Dynamic Thresholds*. International Monetary Fund, 2012.
- [38] Cristian Martín, Peter Langendoerfer, Pouya Soltani Zarrin, Manuel Díaz, and Bartolomé Rubio. Kafka-ml: Connecting the data stream with ml/ai frameworks. *Future Generation Computer Systems*, 126:15–33, 2022.
- [39] Francesco Mazzarotto, Upasana Tayal, Rachel J Buchan, William Midwinter, Alicja Wilk, Nicola Whiffin, Risha Govind, Erica Mazaika, Antonio de Marvao, Timothy JW Dawes, et al. Reevaluating the genetic contribution of monogenic dilated cardiomyopathy. *Circulation*, 141(5):387–398, 2020.
- [40] MongoDB. MongoDB Documentation on Change Streams. <https://www.mongodb.com/docs/manual/changeStreams/>. Accessed: 2023-02-23.
- [41] Chris Montagnino. Running a Trade Surveillance System Without Proper Calibration and Procedures is Like Having a Race Car With No Brakes. <https://www.eventus.com/running-a-trade-surveillance-system-without-proper-calibration-and-procedures/>. Accessed: 2023-01-09.
- [42] Nasdaq. Changing the Game: Artificial Intelligence In Market Surveillance. <https://www.nasdaq.com/articles/changing-game-artificial-intelligence-market-surveillance-2017-04-05>. Accessed: 2023-04-10.
- [43] M. O'Brien. Nasdaq Inside Peek: 2023 Report on FINRA's Examination and 'Risk Monitoring Program. <https://www.nasdaq.com/articles/understanding-trade-surveillance-systems-and-procedures>. Accessed: 2023-04-02.
- [44] Nur Banu Oğur, Mohammed Al-Hubaishi, and Celal Çeken. Iot data analytics architecture for smart healthcare using rfid and wsn. *ETRI Journal*, 44(1):135–146, 2022.
- [45] D. Owczarek. Lambda vs. Kappa. <https://nexocode.com/blog/posts/lambda-vs-kappa-architecture/#:~:text=Lambda%20architecture%20uses%20two%20separate,to%20handle%20complete%20data%20processing>. Accessed: 2023-02-11.
- [46] Hooman Peiro Sajjad, Ying Liu, and Vladimir Vlassov. Optimizing windowed aggregation over geo-distributed data streams. In *2018 IEEE International Conference on Edge Computing (EDGE)*, pages 33–41. IEEE, 2018.

- [47] Jaime Sayago Heredia, Evelin Flores-García, and Andres Recalde Solano. Comparative analysis between standards oriented to web services: Soap, rest and graphql. In *Applied Technologies: First International Conference, ICAT 2019, Quito, Ecuador, December 3–5, 2019, Proceedings, Part I 1*, pages 286–300. Springer, 2020.
- [48] Mitch Seymour. *Mastering Kafka Streams and ksqlDB*. O’Reilly Media, 2021.
- [49] T Sharvari and K Sowmya Nag. A study on modern messaging systems-kafka, rabbitmq and nats streaming. *CoRR abs/1912.03715*, 2019.
- [50] Tinku Singh, Vinarm Rajput, Umesh Prasad, and Manish Kumar. Real-time traffic light violations using distributed streaming. *The Journal of Supercomputing*, pages 1–27, 2022.
- [51] Ralph Steurer. Kafka: Real-time streaming for the finance industry. *The Digital Journey of Banking and Insurance, Volume III: Data Storage, Data Processing and Data Analysis*, pages 73–88, 2021.
- [52] Sashko Stubailo. GraphQL vs. rest. <https://www.apollographql.com/blog/graphql/basics/graphql-vs-rest/>. Accessed: 2022-12-12.
- [53] Brian Swarbrick. Building a quality bi framework solution starts with a quality etl architecture. *Information Management*, 17(7):22, 2007.
- [54] Daniel R Torres, Cristian Martín, Bartolomé Rubio, and Manuel Díaz. An open source framework based on kafka-ml for distributed dnn inference over the cloud-to-things continuum. *Journal of Systems Architecture*, 118:102214, 2021.
- [55] Maximilian Vogel, Sebastian Weber, and Christian Zirpins. Experiences on migrating restful web services to graphql. In *Service-Oriented Computing–ICSOC 2017 Workshops: ASOCA, ISyCC, WESOACS, and Satellite Events, Málaga, Spain, November 13–16, 2017, Revised Selected Papers*, pages 283–295. Springer, 2018.
- [56] Gustav von Heijne and Mattias Mogard. Conceptual dynamics on the trade surveillance market: A study of changes in the swedish trade surveillance market in conjunction with mifid2/miFIR and MAD2/MAR, 2016.
- [57] Theo Zschörnig, Robert Wehlitz, and Bogdan Franczyk. A personal analytics platform for the internet of things-implementing kappa architecture with microservice-based stream processing. In *International Conference on Enterprise Information Systems*, volume 2, pages 733–738. SCITEPRESS, 2017.

Appendix A

Unit Tests

```
@Test
public void testKafkaConsumerNotEmptyTEST1() {

    String topicName = "raw_orders";

    // Configure Kafka consumer properties
    Properties consumerProps = new Properties();
    consumerProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    consumerProps.put(ConsumerConfig.GROUP_ID_CONFIG, "groupId");
    consumerProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
    consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());

    // Create Kafka consumer instance
    KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProps);

    // Subscribe to the topic
    consumer.subscribe(Collections.singleton(topicName));

    ConsumerRecords<String, String> orders = consumer.poll(Duration.ofSeconds(1));

    // Check that at least one message was received
    assertTrue( message: "Topic should not be empty", condition: orders.count() > 0);
}
```

Figure A.1: Unit test 1


```

@Test
public void testKafkaTopicCorrectSerializationTEST2() throws InterruptedException, JsonProcessingException {

    String topicName = "raw_orders";

    // Configure Kafka consumer properties
    Properties consumerProps = new Properties();
    consumerProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    consumerProps.put(ConsumerConfig.GROUP_ID_CONFIG, "groupId");
    consumerProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
    consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());

    // Create Kafka consumer instance
    KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProps);

    // Subscribe to the topic
    consumer.subscribe(Collections.singleton(topicName));

    ConsumerRecords<String, String> orders = consumer.poll(Duration.ofSeconds(1));

    // Deserialize the message payload into a Java object
    ConsumerRecord<String, String> consumedRecord = orders.iterator().next();
    String consumedJson = consumedRecord.value();
    Order deserializedObject = OBJECT_MAPPER.readValue(consumedJson, Order.class);

    // Check that the deserialized object matches the expected object
    Order expectedObject = new Order( s: "3", s1: "20", s2: "10", s3: "1");

    assertEquals( message: "Deserialized object does not match expected object", expectedObject, deserializedObject);

}

```

Figure A.2: Unit test 2

```

@Test
public void testKafkaTopicCorrectOrderingTEST3() throws InterruptedException, JsonProcessingException {

    String topicName = "raw_orders";

    List<String> expectedMessages = Arrays.asList("message1", "message2", "message3");

    // Create Kafka consumer instance
    KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProps);

    consumer.subscribe(Collections.singletonList(topicName));

    // Consume messages and collect actual message order
    List<String> actualMessages = Arrays.asList();
    while (actualMessages.size() < expectedMessages.size()) {
        ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
        for (ConsumerRecord<String, String> record : records) {
            actualMessages.add(record.value());
        }
    }

    // Close Kafka consumer
    consumer.close();

    // Compare actual and expected message order
    assertEquals(expectedMessages, actualMessages);

}

```

Figure A.3: Unit test 3


```

@Test
public void testKafkaOutageTEST4() throws InterruptedException, JsonProcessingException {

    String topicName = "raw_orders";

    List<String> expectedMessages = Arrays.asList("message1", "message2", "message3");

    // Create Kafka consumer instance
    KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProps);

    consumer.subscribe(Collections.singletonList(topicName));

    // Consume messages and collect actual message order
    List<String> actualMessages = new ArrayList<>();
    while (actualMessages.size() < expectedMessages.size()) {
        ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
        for (ConsumerRecord<String, String> record : records) {
            actualMessages.add(record.value());
        }
    }

    //Kafka Producer will be closed in Docker

    // Close Kafka consumer
    consumer.close();

    // Compare that during the outage no more messages other than the three initial ones have been consumed
    assertEquals(expectedMessages, actualMessages);
}

```

Figure A.4: Unit test 4

```

@Test
public void testKafkaOutageTEST5() throws InterruptedException, JsonProcessingException {

    String topicName = "raw_orders";

    List<String> expectedMessages = Arrays.asList("message1", "message2", "message3");
    List<String> expectedMessages2 = Arrays.asList("message1", "message2", "message3", "message4");

    // Create Kafka consumer instance
    KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProps);
    consumer.subscribe(Collections.singletonList(topicName));

    // Consume messages and collect actual message order
    List<String> actualMessages = new ArrayList<>();
    while (actualMessages.size() < expectedMessages.size()) {
        ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
        for (ConsumerRecord<String, String> record : records) {
            actualMessages.add(record.value());
        }
    }

    // Close Kafka consumer
    consumer.close();

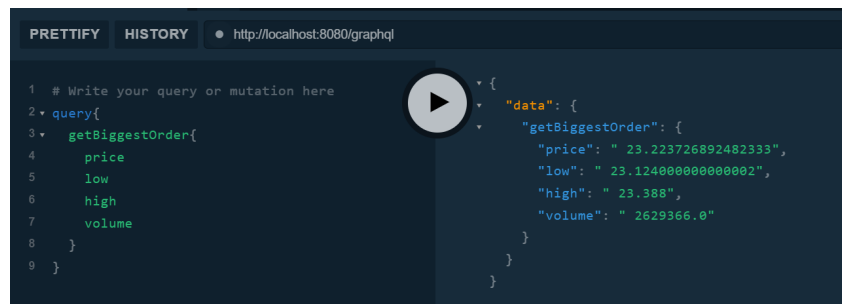
    // Compare that messages have been consumed
    assertEquals(expectedMessages, actualMessages);

    // Create Kafka consumer instance
    KafkaConsumer<String, String> consumer2 = new KafkaConsumer<>(consumerProps);
    ConsumerRecords<String, String> resetRecords = consumer2.poll(timeoutMs: 5000);
    List<String> actualMessages2 = new ArrayList<>();
    while (actualMessages2.size() < expectedMessages2.size()) {
        ConsumerRecords<String, String> records = consumer2.poll(Duration.ofMillis(100));
        for (ConsumerRecord<String, String> record : records) {
            actualMessages2.add(record.value());
        }
    }

    // Compare that messages sent while consumer was down were retrieved
    assertEquals(expectedMessages2, actualMessages2);
}

```

Figure A.5: Unit test 5



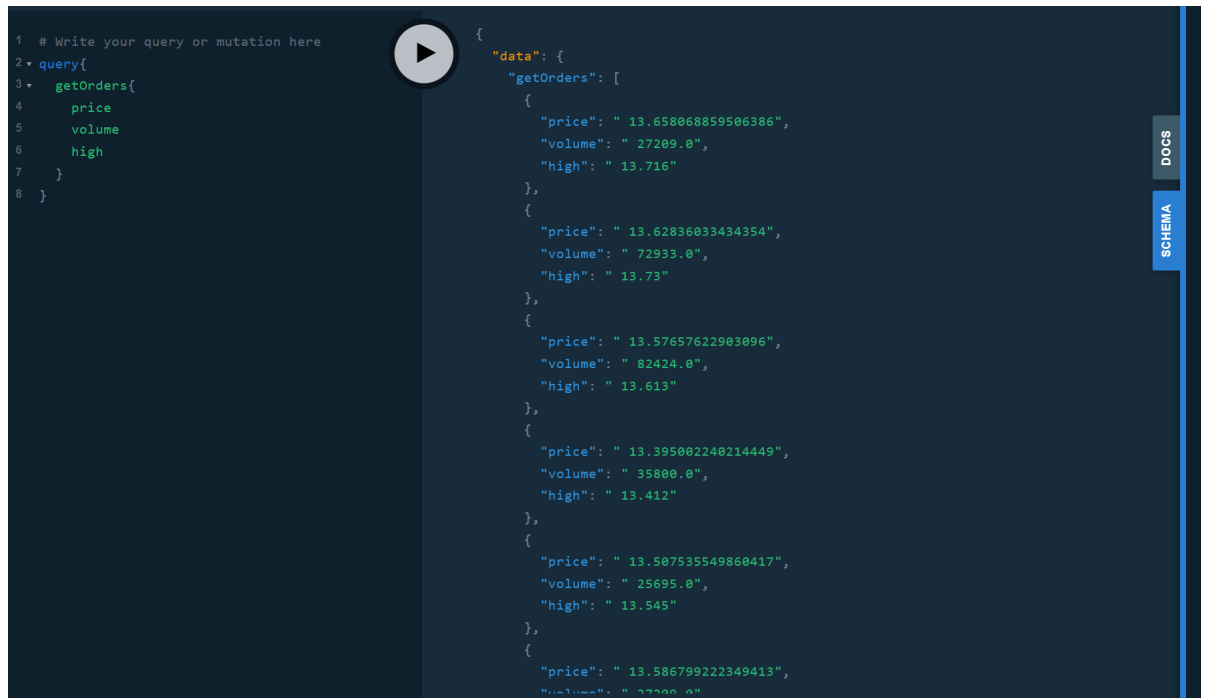
The screenshot shows the GraphQL Playground interface. The URL is `http://localhost:8080/graphql`. The query editor contains the following code:

```
1 # Write your query or mutation here
2 query{
3   getBiggestOrder{
4     price
5     low
6     high
7     volume
8   }
9 }
```

The JSON response is displayed on the right:

```
{
  "data": {
    "getBiggestOrder": {
      "price": " 23.223726892482333",
      "low": " 23.124000000000002",
      "high": " 23.388",
      "volume": " 2629366.0"
    }
  }
}
```

Figure A.6: GraphQL Queries



The screenshot shows the GraphQL Playground interface. The URL is `http://localhost:8080/graphql`. The query editor contains the following code:

```
1 # Write your query or mutation here
2 query{
3   getOrders{
4     price
5     volume
6     high
7   }
8 }
```

The JSON response is displayed on the right:

```
{
  "data": {
    "getOrders": [
      {
        "price": " 13.658068859506386",
        "volume": " 27209.0",
        "high": " 13.716"
      },
      {
        "price": " 13.62836033434354",
        "volume": " 72933.0",
        "high": " 13.73"
      },
      {
        "price": " 13.57657622903096",
        "volume": " 82424.0",
        "high": " 13.613"
      },
      {
        "price": " 13.395002240214449",
        "volume": " 35800.0",
        "high": " 13.412"
      },
      {
        "price": " 13.507535549860417",
        "volume": " 25695.0",
        "high": " 13.545"
      },
      {
        "price": " 13.586799222349413",
        "volume": " 37300.0",
        "high": " 13.545"
      }
    ]
  }
}
```

On the right side of the interface, there are tabs for `SCHEMA` and `DOCS`.

Figure A.7: GraphQL Queries


```
24 transport = WebsocketsTransport(url='ws://localhost:4000/graphql')
25
26 client = Client(
27     transport= transport,
28     fetch_schema_from_transport=True,
29 )
30
31 query = gql('''
32
33 subscription{
34     newDocumentAdded {
35         text
36         createdBy
37     }
38 }
39 ''')
40
41 ans = []
42
43 def test_subscription_value():
44
45     assert ans[0] == 28.99480120981284
46     print("Test passed!")
47
48 for result in client.subscribe(query):
49     ans.append(float(result['newDocumentAdded']['text']))
50     test_subscription_value()
51
52
53
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

PS C:\Users\> graphql> python testSubscriptions.py

Test passed!
Test passed!
Test passed!
Test passed!

node
powershell

Figure A.8: GraphQL Subscriptions test 8. Done in Python since subscriptions were developed in JavaScript and plotted in Python.