Aalto University
School of Electrical
Engineering

Master's programme in Computer, Communication and Information Sciences

Performance Evaluation of Apache Kafka and Redpanda

Simon Helenius

Master's Thesis
2024

Aalto University
School of Electrical Engineering

Author Simon Helenius

Title Performance Evaluation of Apache Kafka and Redpanda

Degree programme Computer, Communication and Information Sciences

Major Signal Processing and Data Science

Supervisor Prof. Risto Wichman

Advisor Mr Simo Aaltonen (MSc)

Collaborative partner Nokia

Date 21 July 2024 **Number of pages** 54 **Language** English

Abstract

The aim of this thesis was to measure the performance of three event streaming platforms in three different test cases. The thesis went through the most important information about streaming infrastructure, including the Kappa architecture, event streaming tools, stream processing tools and the key performance indicators and metrics used to measure the performance of the platforms. The thesis also showed the Python implementations of the producers and consumers used to generate and read the data. Additionally the thesis shows the Docker implementations of the brokers, including the tools used to measure the broker performances.

The three event streaming platforms inspected were using Apache Kafka with ZooKeeper, Apache Kafka with KRaft and Redpanda. The first test case was done with one producer, consumer, broker, topic and partition. The second test case increased the number of producers to four. The third test case had four producers, but also now three brokers and partitions. Overall, the platform using Redpanda had slightly slower throughput degradation than the other platform, but also had significantly worse latency distribution. Both platforms using Kafka had essentially the same latency distributions in each test case, but Kafka with KRaft had slightly slower throughput degradation.

Keywords Event Streaming Platforms, Stream Processing, Streaming Data, Kappa Architecture

Aalto-yliopisto
Sähkötekniikan korkeakoulu

Tekijä Simon Helenius

Työn nimi Apache Kafkan ja Redpandan suorituskyvyn mittaaminen

Koulutusohjelma Tietokone-, kommunikaatio- ja informaatiotieteet

Pääaine Signaalinkäsittely ja datatiede

Työn valvoja Prof. Risto Wichman

Työn ohjaaja DI Simo Aaltonen

Yhteistyötaho Nokia

Päivämäärä 21.7.2024 **Sivumäärä** 54 **Kieli** englanti

Tiivistelmä

Tämän työn tavoitteena oli mitata kolmen viestivirta-alustan suorituskykyä kolmessa eri testitapauksessa. Tutkielmassa käytiin läpi datavirtainfrastruktuurin tärkeimmät periaatteet, mukaan lukien Kappa-arkkitehtuurin, viestivirta-alustat, datavirtaytyökalut sekä keskeiset suorituskykyindikaattorit ja parametrit, joiden avulla alustojen suorituskykyä mitataan. Tutkielma esitteli myös Python-toteutukset tuottajille ja kuluttajille, joita käytettiin datan luomiseen ja lukemiseen. Lisäksi tutkielmassa näytettiin alustojen Docker-toteutukset, sisältäen myös työkalut, joilla mitattiin näiden alustojen suorituskykyä.

Tutkittavana oli kolme viestivirta-alustaa, jotka käyttivät Apache Kafkaa ZooKeeperin kanssa, Apache Kafkaa KRaftin kanssa ja Redpandaa. Ensimmäinen testitapaus suoritettiin yhdellä tuottajalla, kuluttajalla, alustalla ja osioinnilla. Toisessa testitapauksessa tuottajien määrä nostettiin neljään. Kolmannessa testitapauksessa oli neljä tuottajaa, mutta myös kolme alustaa ja osiontia. Kokonaisuudessaan Redpandaa käyttävä alusta koki hieman hitaampaa suoritustehon heikkenemistä kuin muut alustat, mutta sillä oli myös merkittävästi huonompi viivejakauma. Molemmilla Kafkaa käyttävillä alustoilla oli käytännössä samat viivejakaumat jokaisessa testitapauksessa, mutta Kafka KRaftin kanssa koki hieman hitaampaa suoritustehon heikkenemistä.

Avainsanat Viestivirta-Alusta, Datavirran Käsittely, Virtaava Data, Kappa Arkkitehtuuri

Contents

1 Introduction

As the requirement to handle ever growing data efficiently continues, companies must adopt new solutions to meet the demand. This has popularized adoption of streaming architectures across fields to provide real-time handling of the data over the already existing methods, such as handling it in chunks with batch processing. Stream processing is mainly associated with unbounded streaming data, which is continuously generated new data thought to have no end point. However, it has also recently been adopted to process bounded data as well, which has a clear end point. The main advantage of stream processing over existing methods, such as batch processing, is lower latency while still having high throughput. Importantly, stream processing provides (near) real-time analysis of the data allowing new use cases and improving existing ones. Some examples of these use cases are more detailed business analysis, low-latency application error monitoring and alerting, more responsive wait times and dynamic pricing, as well as improved fraud detection and human mobility analytics [1–3].

Streaming infrastructure provides a pipeline from the data source to an application layer flowing through the streaming layer. The streaming layer ingests the incoming data from a data source, prepares the ingested data (e.g., filtering, transforming and aggregating), and often forwards the data to a data store for low-latency access. The application layer uses this data to provide services such as analysis or reporting.

However, these streaming infrastructures must be designed to accommodate specific use cases. The initial issue in the design of streaming infrastructure is choosing the tools. Studies have compared which stream processing tools provide the best results through benchmarking in different use cases. References [4–6] compare popular stream processing tools, including Apache Flink, Apache Spark, Apache Storm and Apache Samza. [4] shows that Spark or Flink is the optimal choice depending on the use case, whereas Storm was the optimal choice on none of them. On the other hand, Storm performed over Samza and Spark in [5]. While Storm and Spark have similar throughput in [6], Storm is the optimal choice due to its lower latency.

While comparisons between popular stream processing tools, such as Apache Spark, Apache Flink, Apache Storm and Apache Samza exist, there's no academic research on comparative benchmarking of different event streaming platforms. The most popular event streaming platform is Apache Kafka, but Redpanda has gained traction by promoting to be an evolved version of Kafka with better performance.

The main goal of this thesis is to compare event streaming platforms in a few separate test cases. The results could provide the optimal event streaming platform for the streaming architecture needs at Nokia. Comparing the performance of the event streaming platform will support designing the architecture for the needs of 5G and 6G R&D data. The test cases and metrics/KPIs (Key Performance Indicators) are prioritized for what is relevant for Nokia.

The thesis will test three event streaming platforms: Apache Kafka with ZooKeeper, Apache Kafka with KRaft and Redpanda. This thesis will also provide the criteria and method for benchmarking these different tools. This benchmarking is important for making general conclusions about the performance of the tools. However, different tools should be benchmarked for critical use cases each time as the results may differ depending on the use case or the used system and its configurations. Furthermore, tools evolve, and new ones get introduced, which could also affect the results in the future. As shown by [4–6], the optimal choice may depend on the use case and other factors.

The remainder of this thesis is organized as follows. Chapter 2 provides the necessary background information for understanding the stream processing architecture, the event streaming platforms, the stream processing tools and the metrics used to compare the event streaming platforms. Chapter 3 explains the test cases, shows the Python and Docker implementations for each part of the event streaming platform, including the metric monitoring. In Chapter 4 the results from these test cases are evaluated. Finally, Chapter 5 concludes the thesis by summarizing the key concepts and findings, and providing potential future research topics.

2 Streaming Infrastructure

In order to understand the basics of the streaming infrastructure, this Chapter goes through the most important parts. Section 2.1 provides the required architectures to handle streaming data, answers why Kappa architecture was chosen in this thesis, and illustrates this chosen architecture. Section 2.2 explains how event streaming platforms work, with also going a bit deeper on each of the platforms. Section 2.3 provides the most important theory on stream processing tools and the existing research on their performance. Section 2.4 explains the measured Key Performance Indicators (KPIs) and metrics in this thesis. Note that this thesis uses the terms message, event and record interchangeable to describe the data handled by event streaming platform, while this might not be the case in some of the other sources.

2.1 Kappa Architecture

Streaming data is constantly generated and generally big data. Generally it's important to manage this data as soon and fast as possible. Batch processing is a method where the data is processed in certain size or timed chunks. Stream processing on the other hand provides real-time or near real-time handling of the data with minimal latency requirements. Importantly, this provides (near) real-time analysis of the data.

Kappa architecture is a streaming architecture introduced first in 2014 by Jay Kreps in [7] to replace the existing Lambda architecture. The difference between the two is that Lambda architecture has its own layer for batch processing. On both architectures the new streaming data is processed by the streaming layer. In Lambda architecture the retained historical data is processed by the batch layer, as changes are made to streaming infrastructure code due to new requirements, features or bugs. Kappa architecture on the other hand allows historical data to also go through the streaming layer rather than using another layer for it.

The thought behind Lambda architecture is that stream layer is used only when the lower latency is needed and batch processing otherwise as the streaming platform was seen as inherently approximate and more lossy, although in [7] it's argued that this might just be due to the maturity of batch processing systems compared to stream processing systems. However, the main problem of Lambda architecture is maintaining two complex systems that provide the same results for the data. Essentially Lambda architecture has two systems, a scalable high-latency batch processing layer and a low-latency stream processing layer that can't reprocess results, merged into a low-latency processing system that could be provided by a single system in Kappa architecture.

Two drawbacks of Kappa architecture compared to the Lambda architecture are pointed out on [7]. First, it requires a database that supports high-volume writes. Second, the database requires two times the storage temporarily whenever a change is made in the code with the implementation in the tech blog. A new output table is made in the database and while the retained historical data is sent to this new table, the old one is kept running until the new table has caught up the original table. Afterwards the original table can be deleted.

In a 2019 study [8] it was shown that while Lambda architecture had a 9% better accuracy, it also had 2.2 times longer processing time. Furthermore, Lambda architecture had 10-20% higher CPU usage and 0.5 GB RAM usage. In the study the architectures were tested by counting the number of each different word in a document. The study uses Apache NiFi to ingest the data to Apache Kafka. From Kafka it continues to the stream processing that uses Apache Spark. Additionally, the Lambda architecture uses Apache Flume and HDFS in the batch layer.

In this thesis only Kappa architecture is considered due to its simplicity. In this case we only have to develop and manage the streaming layer without the batch layer. Furthermore, given the possible size of data that needs to be processed, the much lower processing time seems critical to provide low-latency solutions and may even be required to prevent the data processing falling behind the produced data. If the data processing had constantly lower throughput than the data producing, the latency would keep increasing without bound.

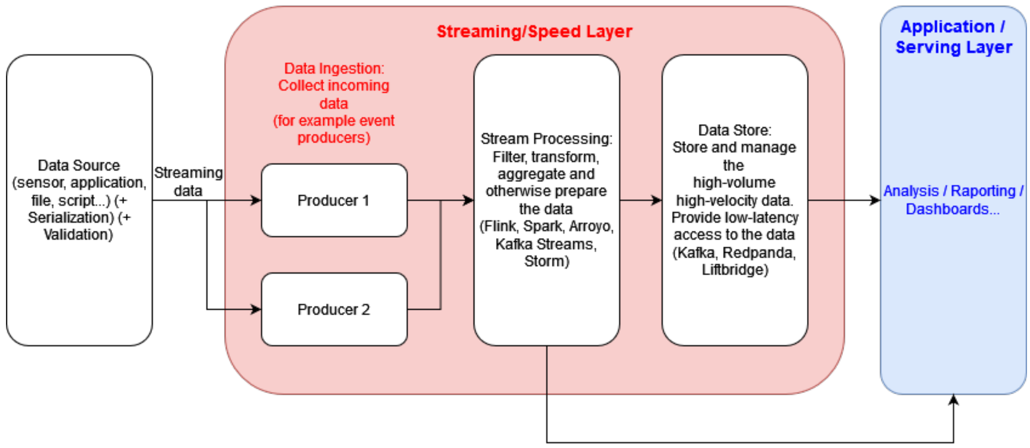


Figure 1: Kappa architecture

Figure (1) shows a graphical representation of the Kappa architecture. One of the benefits of Kappa architecture is its simplicity in concept. Streaming data originates from a data source, such as a sensor, and is handled by the streaming layer. Afterwards the data is ingested and forwarded to stream processing where the data is filtered, transformed, aggregated and otherwise prepared. The data can then be sent to a data store to store, manage and provide low-latency access to the data. Finally, application layer provides analysis, reporting or whatever the data is used for from the data store or straight from the stream processing.

2.2 Event Streaming Platforms

This section contains general concepts of event streaming platforms that can be found mostly on both of their documentations [9, 10]. The subsections further divide the information that's relevant for each of the used tools. That information can be found on the corresponding tool documentations. Also a section on an even more novel event streaming platform called Liftbridge that was not tested in this thesis is included. [9] is the Kafka documentation, [10] is the Redpanda documentation and [11] is the Liftbridge documentation.

Event streaming platforms utilize topics to organize and store messages. The producers write the data in certain topics so consumers can read messages only from the needed topics. With event streaming platforms, messages are not deleted after being read so the messages can be read as often as needed, which enables separate consumer groups to read the same messages and the messages can be reread in case of failure, new feature or other code change. Since multiple consumers can be attached to the same topic, multiple consumers can use the same data for their own purposes. Furthermore, Event streaming platforms like Kafka, Redpanda and Liftbridge tend to be linearly scalable by simply adding new brokers, producers or consumers.

Event streaming platforms have an important concept of partitioning. With partitioning, producers spread their load to partitions within separate brokers. This is done to increase performance as the producers can write to and consumers can read from multiple brokers simultaneously.

For fault-tolerance and high availability the topics should be replicated to multiple brokers in case of broker failure or maintenance. One of the brokers will have a partition leader, where all of the new messages will be produced and a chosen number of different brokers will have a partition follower that replicates the messages from the leader. If the partition leader is down, one of the other partitions will be elected as the new leader.

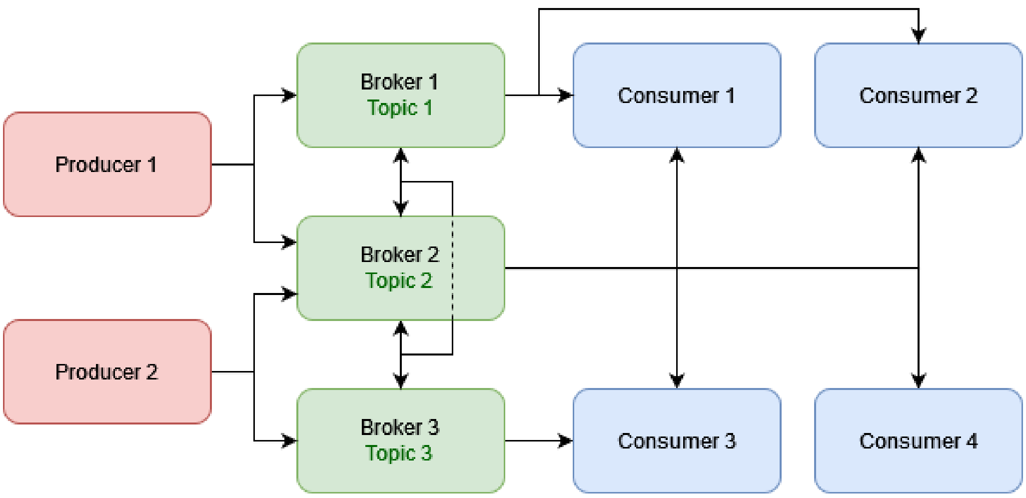


Figure 2: Event streaming platform architecture

Figure (2) portraits the architecture of event streaming platforms. In this example there are two producers, three brokers, three topics and four consumers. For simplicity only the partition leaders of the topics are shown in the brokers, but the arrows between the brokers imply the connection between brokers for replication. Note how each partition leader of a topic is on a different broker, as it's the desirable implementation to maximize performance, since it minimizes the number of writes to each broker as messages are only written to the partition leader and now the leaders are split between each broker. Producer 1 publishes to Topics 1 and 2, whereas Producer 2 publishes to Topic 3 and also to Topic 2. If each topic had three partitions, all of the producers and consumer would be connected to each broker. Consumers 1 and 2 have subscribed to Topic 1, Consumer 3 is the only consumer to have subscribed to Topic 3, and all of the consumers have subscribed to Topic 2.

Let's now consider the Kappa architecture with only a event streaming platform. In this case the producers handle the data ingestion from a data source. The brokers work as the low-latency data store for the high-volume high-velocity data. Consumers apply as the connector from the data store to the application layer. There's no stream processing without including a stream processing tool like Kafka Streams, Flink or Spark.

2.2.1 Apache Kafka

Apache Kafka is an event streaming platform, that provides three key capabilities. Kafka provides a publish-subscribe system, where event producers publish (write) events to certain topics and consumers subscribe (read) only to needed topics.

Kafka uses TCP network protocol for its high-performance communication between brokers and clients. In the case of a broker failure the other brokers can pick up its work to ensure continuous operations without data loss. The brokers can be deployed on hardware, virtual machines and containers in native and cloud implementations. The client applications are either producers that write messages to the brokers or consumers that read them. Client applications are provided in most of the popular programming languages such as C/C++, Python, Go, Java, Rust and HTTP Rest API by the Kafka community.

These use cases have been provided by the documentation of Kafka: messaging, website activity tracking, metrics, log aggregation, stream processing, event sourcing and commit log. Kafka can work as a large scale message processing application to for example decouple processing from event producers or to buffer unprocessed messages. Kafka can be used to track site activity such as page views or searches, to collect logs from servers for processing and to log state changes of a system. Furthermore, Kafka can be used for the whole streaming process, including ingestion through event producers, preparing the data in stream processing through Kafka Streams, storing the data through brokers and consumers allow handling the data in the application layer.

With Kafka, messages are deleted after a chosen data retention period, or alternatively the oldest messages are deleted after the size of the storage taken by the messages crosses a chosen byte limit. Additionally, Kafka supports log compaction. In this case publishers should add keys to most messages, since with log compaction only the last messages for each unique key are retained. Furthermore, Kafka enables having different retention rules for separate topics.

2.2.2 Redpanda

Redpanda claims four aspects that differentiates it from other event streaming platforms. Redpanda is implemented on C++ and claims due to it to have a greater throughput and up to ten times lower 99 percentile latency. Redpanda is a single binary and doesn't rely on any external systems, unlike Kafka that is built on Java virtual machine (JVM) and used to require Apache ZooKeeper. Furthermore, while Liftbridge is a single binary, it's build on top of a publish-subscribe messaging system NATS. Redpanda also claims that their usage of Raft consensus algorithm allows it even at high loads to tolerate partial environmental failures and provide predictable performance. Finally, Redpanda claims to provide data sovereignty even with their cloud version.

With Redpanda, messages are deleted after a chosen data retention period or either the oldest messages are deleted after the size of the storage taken by the messages crosses a chosen byte limit. Like Kafka, Redpanda supports different message-retention rules per topic. However unlike Kafka and Liftbridge, Redpanda doesn't seem to support log compaction.

2.2.3 Liftbridge

Liftbridge is a Go-based event streaming platform that's implemented on top of a publish-subscribe messaging system NATS. The difference between NATS and event streaming platforms like Kafka and Redpanda is the lack of message durability, which Liftbridge provides. This is done by adding partitioning that native NATS doesn't have.

In NATS and Liftbridge the terms topic and consumer of Kafka and Redpanda are called subject and subscriber instead. Essentially, Liftbridge acts as a consumer that writes messages received from NATS subjects to a message log from which the subscribers can read. Along Go, client libraries have been done for Java, Rust, Node.js and Python.

Like Kafka and Redpanda, Liftbridge supports age-based and size-based message-retention rules. Additionally, Liftbridge has an option for message-based rule, that deletes messages after the number of messages go over a chosen limit. Like Kafka, Liftbridge supports log compaction. Liftbridge doesn't however support different message-retention rules per subject that Kafka and Redpanda provide.

2.3 Stream Processing Tools

Given the time based nature of streaming data, stream processing tools provide many time based operations. Perhaps the most important time based operation is windowing. Windowing aggregates the unbounded streaming data into windows. Windowing allows to for example calculate the moving average, sum or count of a certain variable during that window.

Stream processing tools, such as Apache Flink and Apache Spark generally provide tumbling, sliding and session windows. Tumbling windows aggregate the data in certain size time chunks that come one after the other. Sliding windows also aggregate data over certain size time chunks, but now the windows overlap given a wanted slide time period, where the window starts the slide time period before the previous window ends. Unlike the other two, session windows don't use specific sized time chunks. Instead, the session window lasts until the next event doesn't arrive before a given gap period. Whenever new events don't appear during the gap period, a new window is started when the next event comes in. Furthermore, Flink also provides global windows. In this case, each key will have it's own global window and a custom trigger for when to end a window needs to be configured.

Stream processing tools, such as Apache Flink and Apache Spark, provide timely stream processing. Timely stream processing uses the event time, such as Kafka message timestamp, rather than the processing time itself. Depending on the use case, windowing can be done either with processing time or event time. The advantage of using processing time is that it provides better throughput and lower latency. Using the event time allows the stream processing tools to utilize time dependant operations such as windowing. Effectively, this means the same input data provides basically the same output.

Timely stream processing uses watermarks to track the progress of the stream processing by marking timestamps over time at the times that the tool doesn't except to receive any data from before the watermark. This is done, since with streaming data the events may arrive in any order and some of the events may be delayed. Although the streaming tools don't except any events with timestamp before the watermark, if still some event with earlier timestamp than the current watermark is received, the event is dropped.

Applications in stream processing tools compose of streaming dataflows. These dataflows form directed graphs starting from sources and ending in sinks. Between the sources and sinks can be many different operations, such as windowing or joining one streaming data with other streaming data from different sources or operations. The applications may consume both real-time unbounded streaming data, as well as bounded historic data. Often the ability of re-processing historical data is important for the streaming applications for producing deterministic results.

Each node/operation in the dataflow may maintain a state to enable stateful operations. Stateful operations like joins and windowing use the information from multiple events. States are checkpointed with limited effect on performance, so in cases of machine, network or software failure the state can be restored to the checkpoint. The records after the checkpoint can be replayed, since the checkpoint includes for example the Kafka offset.

Studies [4–6] compare the performance of multiple stream processing tools. [4] shows that the optimal choice is either Spark or Flink depending on the use case, whereas Storm was not the optimal choice for any of them. [5] shows conversely, that Storm performed over Samza and Spark. [6] also preferred Storm over Samza due to its lower latency, although they had similar throughputs. Furthermore, it is shown that the performances between Docker and native implementations is not significantly different, with the Docker implementation having only 2-10% worse performance.

2.4 Key Performance Indicators and Metrics

Key Performance Indicators (KPIs) describe how well a system is operating. KPIs are often used to find an optimal solution to a problem, and in the context of this thesis they are used to find the optimal initial streaming platform for the use cases at Nokia. Other metrics will help explain these results and may be useful in finding the bottlenecks of the different tools.

Here's a list for the investigated metrics of the event streaming platforms [9, 10]:

Byte Rate How many total bytes the producer sends / consumer receives per unit of time.

Throughput How many total bytes of payload, the actual relevant data that is required in the application layer, the producer sends / consumer receives per unit of time. In other words, any overhead included in the byte rate will be omitted.

Average Record Size On average, how many bytes a record is.

CPU/Memory Usage Measures how much the tool uses system resources. Important to identify potential bottlenecks of the tools.

Latency Measures how long it takes from message being produced to the message being consumed.

Lag Measures how many messages behind the consumed messages are from the produced messages.

The main KPIs used to measure the performance of event streaming platforms are throughput, message produce/consume rate and latency, while the other metrics help to find the potential bottlenecks.

3 Implementation

This Chapter shows the implementation of the streaming infrastructure regarding this thesis. Section 3.1 goes through the test cases used to measure the perfromance of the event streaming platforms. Sections **??** shows and explains the Python implementations of the producers and consumers. Sections 3.4 shows and explains the Docker implementations of the brokers and the tools that measure their performance.

3.1 Test Cases for Event Streaming Platforms

For measuring the performance of Kafka, this thesis will use Python producers and consumers provided by Confluent Kafka. Some preliminary testing and existing knowledge at Nokia concluded that the Confluent Kafka performed better than the native version. Since Redpanda doesn't have its own producers and consumers, but rather its brokers work with the Kafka API, this thesis will also use Confluent Kafka Python producers and consumers to measure the performance of Redpanda. The brokers will run on docker containers along Prometheus, Grafana and cAdvisor, which are used to handle the performance data of the brokers. With Redpanda its own docker implementation of the broker will be used, whereas with Kafka the docker implementation is also provided by Confluent.

Test Case 1 The first test case is the simplest with one producer, one consumer, one topic and one broker. This will give the baseline of how much data can be send per producer to one topic assuming that one consumer, the network, CPU and memory can deal with the data of one producer.

Test Case 2 To make sure the limiting factor in the first test case is the producer as expected, the second test case will increase number of producers to four. The number of consumers, topics and brokers will be kept at one. If the data sent per producer is now noticeable less than with only one producer, that would indicate that the limiting factor has been shifted to some other factor.

Test Case 3 In the last test case of the event streaming platforms the number of brokers is increased from one to three to see how that impacts the results. The number of producers is kept as four and the number of consumers and topics are kept as one.

3.2 Producer Implementation

Here is a Python implementation of the producer used to send data and gather metrics:

```
import json
import datetime
from confluent_kafka import Producer
from confluent_kafka.admin import AdminClient, NewTopic
import pandas as pd

```

```python
class Stats:
    def __init__(self):
        self.producer = Producer({'bootstrap.servers': "server1:9092,server2:9093,server3:9094", 'statistics.interval.ms': 30*1000, 'stats_cb': self.stats_cb}) #, Producer({'bootstrap.servers': "server1:9092", 'statistics.interval.ms': 30*1000, 'stats_cb': self.stats_cb})
        self.timestamps = []
        self.bytes = []

    def stats_cb(self, stats_json_str):
        json_dict = json.loads(stats_json_str)
        self.timestamps.append(datetime.datetime.fromtimestamp(json_dict['time']))
        self.bytes.append(json_dict['tx_bytes'])

def kafka_delivery_error(err, msg):
    if err is not None:
        print(f'Message delivery failed: {err}')

stats = Stats()
timestamps = []
msg_sizes = []
t_start = datetime.datetime(2023,12,29,10,49,0,0)
t_end = t_start + datetime.timedelta(0,30*60 + 60)
tool = 'kafka_zk'#,'kafka_kraft','redpanda'
topic = f'{tool}_{t_start}'.replace(' ','_').replace(':','.')

admin_client = AdminClient({"bootstrap.servers": "server1:9092,server2:9093,server3:9094"})
if topic not in admin_client.list_topics().topics:
    admin_client.create_topics([NewTopic(topic, 3, 1)]) #, admin_client.create_topics([NewTopic(topic, 1, 1)])
while topic not in admin_client.list_topics().topics:
    pass

print(f"Topic: {topic}")
while datetime.datetime.now() <= t_start:
    stream = generate_record()
    stats.producer.produce(topic, stream, callback=kafka_delivery_error)
    stats.producer.poll(0)
print("Benchmark starting...")
while datetime.datetime.now() <= t_end:
    stream = generate_record()
    try:
        stats.producer.produce(topic, stream, callback=kafka_delivery_error)
        stats.producer.poll(0)
        timestamps.append(datetime.datetime.now())
        msg_sizes.append(len(stream))
    except:
        pass

```

```
52 df = pd.Series(msg_sizes, index=timestamps)
53 try:
54     df.to_csv(f"data/{tool}/producer_msg_sizes.csv", mode='a')
55 except:
56     df.to_csv(f"data/{tool}/producer_msg_sizes.csv")
57 
58 df = pd.Series(stats.bytes, index=stats.timestamps)
59 try:
60     df.to_csv(f"data/{tool}/producer_stats_bytes.csv", mode='a')
61 except:
62     df.to_csv(f"data/{tool}/producer_stats_bytes.csv")
```

Listing 1: producer.py

First a Stats-named class is created on line 9, that initiates the producer with an ability to gather producer statistics. The bootstrap.servers are the broker addresses to send the data into, while the statistics.interval.ms sets that the statistics are gathered every 30 seconds and with stats_cb the statistics to be gathered can be accessed through a function. This function is defined between lines 13 and 16. The stats_json_str parameter in the function call is given by the underlying C-implementation of the producer. For the producer only the total sent bytes and the timestamp of when that value was taken are gathered. Between lines 25 and 28, the start of the benchmarking time is set, the end of it is set for 31 minutes after the start, the event streaming platform used is set and a new topic name is generated with the start time to generate an unique topic name so previous runs won't have an effect. Between lines 31 and 34, the new topic is created with one replication factor and either with three partitions in Test Case 3 (one for each broker) or with one partition in Test Case 1 and Test Case 2 , and then the program will wait until the topic is made. Then between lines 37 and 40 a record is generated and sent with the producer for a few minutes, so the star of producing records won't affect the results. Next as the benchmark is started new records will be produced, but now also the sent time and size of each record is collected for plotting the throughput results. At the end of the program the measured values are saved to a .csv file. In Test Case 2 and Test Case 3 the producer.py program is simply run four times separately. The generate_record function is not shown, but it generates a single record. The record generation contains a bit of randomness, with the record size ranging between 1 565 and 1 602 bytes averaging at around 1587.80. Given how many records are sent during the time frame of the tests, this randomness shouldn't have an effect. The average record sizes between test cases, tools and whether measured at producer or consumer are all equal within one 100th of a byte.

3.3 Consumer Implementation

Here is a Python implementation of the consumer used to read the sent data and gather metrics:

```
from confluent_kafka import Consumer
import pandas as pd
import numpy as np
import datetime
```

```python
import time
import json

class Stats:
    def __init__(self):
        self.consumer = Consumer({'bootstrap.servers': "server1:9092,server2:9093,server3:9094", 'group.id': 'thesis', 'statistics.interval.ms': 30*1000, 'stats_cb': self.stats_cb}) #, Consumer({'bootstrap.servers': "server1:9092", 'group.id': 'thesis', 'statistics.interval.ms': 30*1000, 'stats_cb': self.stats_cb})
        self.timestamps = []
        self.bytes = []
        self.lag = {}

    def stats_cb(self, stats_json_str):
        json_dict = json.loads(stats_json_str)
        self.timestamps.append(datetime.datetime.fromtimestamp(json_dict['time']))
        self.bytes.append(json_dict['rx_bytes'])
        for topic in json_dict["topics"]:
            for partition in json_dict["topics"][topic]["partitions"]:
                try:
                    self.lag[json_dict["topics"][topic]["topic"] + '_' + str(json_dict["topics"][topic]["partitions"][partition]["partition"])].append(json_dict["topics"][topic]["partitions"][partition]["consumer_lag"])
                except:
                    self.lag[json_dict["topics"][topic]["topic"] + '_' + str(json_dict["topics"][topic]["partitions"][partition]["partition"])] = [json_dict["topics"][topic]["partitions"][partition]["consumer_lag"]]

stats = Stats()
timestamps = []
msg_sizes = []
latencies = []
t_start = datetime.datetime(2023,12,29,10,49,0,0)
t_end = t_start + datetime.timedelta(0,30*60 + 60)
tool = 'kafka_zk'#,'kafka_kraft','redpanda'
topic = f'{tool}_{t_start}'.replace(' ','_').replace(':','.')
print(f"Topic: {topic}")
stats.consumer.subscribe([topic])
while datetime.datetime.now() <= t_start:
    msg = stats.consumer.poll(0)
print("Benchmark starting...")
while datetime.datetime.now() <= t_end:
    msg = stats.consumer.poll(0)
    if msg:
        msg_timestamp = msg.timestamp()[1]
        if msg_timestamp >= 0:
            timestamps.append(datetime.datetime.now())
```

```python
            latencies.append(np.floor(time.time()*1000) - 
    msg_timestamp)
            msg_sizes.append(len(msg.value()))

df = pd.Series(msg_sizes, index=timestamps)
df.to_csv(f"data/{tool}/consumer_msg_sizes.csv")

df = pd.Series(stats.bytes, index=stats.timestamps)
df.to_csv(f"data/{tool}/consumer_stats_bytes.csv")

df = pd.Series(latencies, index=timestamps)
df.to_csv(f"data/{tool}/latencies.csv")

df = pd.DataFrame()
for partition in stats.lag:
    df[partition] = pd.Series(stats.lag[partition], index=stats.
    timestamps[-len(stats.lag[partition]):])
    df[partition] = df[partition][df.index >= t_start + datetime.
    timedelta(0,60)]
df.to_csv(f"data/{tool}/lag.csv")
```

Listing 2: consumer.py

Again, first a Stats-called class is created, that initiates the consumer similarly to as how the producer is initiated. Now however, along the timestamp and the byterate, also the lags are gathered from the internal C-implementation. The lags are separated per topic, which are further separated per partition. Also similarly, as with the producer, then the consumer is run until the benchmarking has begun, and after the benchmark has ended, the results are written into .csv files. One difference here is, that along with the timestamps and message sizes, also the latency of each message is gathered in milliseconds.

3.4 Broker Implementation

With cAdvisor resource usage statistics can be gathered of Docker containers, specifically here used to gather the CPU usage, memory usage, and sent and received network rates of the brokers. Prometheus is used to store the values and Grafana is used to plot these results in real-time. The following docker-compose file contains the creation of these three monitoring tools:

```yaml
version: '3.8'
services:
  grafana:
    image: docker-registry/grafana/grafana
    container_name: simon_grafana
    ports:
      - "24000:3000"
    volumes:
      - ./volumes/grafana:/var/lib/grafana
  prometheus:
    image: docker-registry/prom/prometheus
    container_name: simon_prometheus
```

```
13     ports:
14       - "24090:9090"
15     volumes:
16       - ./prometheus.yml:/etc/prometheus/prometheus.yml
17       - ./volumes/prometheus:/prometheus
18     command: "--config.file=/etc/prometheus/prometheus.yml"
19   cadvisor:
20     image: gcr.io/cadvisor/cadvisor
21     container_name: simon_cadvisor
22     ports:
23     - 24808:8080
24     volumes:
25     - /:/rootfs:ro
26     - /var/run:/var/run:rw
27     - /sys:/sys:ro
28     - /var/lib/docker/:/var/lib/docker:ro
29     command:
30       - '-housekeeping_interval=15s'
31       - '-docker_only'
32       - '-disable_metrics=percpu,process,sched,tcp,udp,diskIO,disk'
33       - '-store_container_labels=false'
```

Listing 3: docker-compose-monitoring.yml

With the image parameter, the image copies the default docker file from a docker registry and with the other parameters default settings can be modified. With the ports parameter, the open ports on the host and on the container can be defined. The first number is the open port on the host, while the second is the same port on the container, but can possible be mapped in the container as a different number. In this file for example, Grafana is open on the 24000 port for the host, but in the container it is mapped to the default port of 3000. The reason to have different values as ports, is so the container can have default ports of the tools open for simpler implementation, but on the host the ports may already be used for different applications. Here the advantage is that Grafana and cAdvisor are by design created to interact with Prometheus. With the volumes parameter, files and folders accessible in the container can be set up. The extra commands for the cAdvisor on rows between 29 and 33 are added, as cAdvisor by default has a relatively high CPU and memory usage.

The following docker-compose-kafka-zk-3.yml file shows the docker-compose file used to start up ZooKeeper and three Kafka brokers in Test Case 3 :

```
version: '3.8'
services:
  zookeeper:
    image: docker-registry/confluentinc/cp-zookeeper
    container_name: simon_zookeeper
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
    ports:
      - "24181:2181"
    volumes:
      - ./volumes/zookeeper/data:/var/lib/zookeeper/data
      - ./volumes/zookeeper/log:/var/lib/zookeeper/log
```

```
      - ./volumes/zookeeper_secrets:/etc/zookeeper/secrets
  kafka-1:
    build: .
    container_name: simon_kafka-1
    ports:
      - "24092:9092"
    environment:
      KAFKA_BROKER_ID: 1
      CLUSTER_ID: 's8lpgdm2RtGl82StuC0SRg'
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_LISTENERS: PLAINTEXT://kafka-1:19092, PLAINTEXT_HOST://0.0.0.0:9092
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka-1:19092, PLAINTEXT_HOST://host:24092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT, PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_OPTS: -javaagent:/usr/app/jmx_prometheus_javaagent.jar=7071:/usr/app/prom-jmx-agent-config.yml
    volumes:
      - ./volumes/kafka-1_data:/var/lib/kafka/data
      - ./volumes/kafka-1_secrets:/etc/kafka/secrets
    deploy:
      resources:
        limits:
          cpus: '6'
          memory: 2.67G
  kafka-2:
    build: .
    container_name: simon_kafka-2
    ports:
      - "24093:9092"
    environment:
      KAFKA_BROKER_ID: 2
      CLUSTER_ID: 's8lpgdm2RtGl82StuC0SRg'
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_LISTENERS: PLAINTEXT://kafka-2:19092, PLAINTEXT_HOST://0.0.0.0:9092
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka-2:19092, PLAINTEXT_HOST://host:24093
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT, PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_OPTS: -javaagent:/usr/app/jmx_prometheus_javaagent.jar=7071:/usr/app/prom-jmx-agent-config.yml
    volumes:
      - ./volumes/kafka-2_data:/var/lib/kafka/data
      - ./volumes/kafka-2_secrets:/etc/kafka/secrets
    deploy:
      resources:
        limits:
          cpus: '6'
```

```
59            memory: 2.67G
60    kafka-3:
61      build: .
62      container_name: simon_kafka-3
63      ports:
64        - "24094:9092"
65      environment:
66        KAFKA_BROKER_ID: 3
67        CLUSTER_ID: 's8lpgdm2RtGl82StuC0SRg'
68        KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
69        KAFKA_LISTENERS: PLAINTEXT://kafka-3:19092, PLAINTEXT_HOST
     ://0.0.0.0:9092
70        KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka-3:19092,
     PLAINTEXT_HOST://host:24094
71        KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,
     PLAINTEXT_HOST:PLAINTEXT
72        KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
73        KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
74        KAFKA_OPTS: -javaagent:/usr/app/jmx_prometheus_javaagent.jar
     =7071:/usr/app/prom-jmx-agent-config.yml
75      volumes:
76        - ./volumes/kafka-3_data:/var/lib/kafka/data
77        - ./volumes/kafka-3_secrets:/etc/kafka/secrets
78      deploy:
79        resources:
80          limits:
81            cpus: '6'
82            memory: 2.67G
```

Listing 4: docker-compose-kafka-zk-3.yml

Both ZooKeeper and Kafka brokers used are provided by Confluent. Lines 32-36, 55-59 and 78-82 show how each broker has been given 6 CPUs (18 total) and 2.67 GB of memory (8 GB total). For Test Case 1 and Test Case 2 the one broker is defined similarly, but the single broker was given the total 18 CPUs and 8 GB of memory. Each broker is given its own broker ID, but each broker is put on the same cluster. The broker addresses are set with the KAFKA_ADVERTISED_LISTENERS-parameter. The first is the address on the container and the second is the address on the host. The KAFKA_OFFSET_TOPIC_REPLICATION_FACTOR-parameter is the replication factor for an underlying topic that keeps track of topic offsets and is simply set to 1 from the default 3. This is in fact required on Test Case 1 and Test Case 2 , when there is only one broker.

The following docker-compose-kafka-kraft-3 shows the docker-compose file used to start up Kafka with KRaft in Test Case 3 :

```
\begin{lstlisting}[caption=docker-compose-kafka-zk-3.yml]
version: '3.8'
services:
  kafka-1:
    build: .
    container_name: simon_kafka-1
    hostname: kafka-1
    ports:
```

```
      - "24092:9092"
    environment:
      KAFKA_NODE_ID: 1
      CLUSTER_ID: '8FbS82vSS3mjE8xPa2AAag'
      KAFKA_CONTROLLER_LISTENER_NAMES: 'CONTROLLER'
      KAFKA_LISTENERS: PLAINTEXT://kafka-1:19092,CONTROLLER://kafka-1:19093,PLAINTEXT_HOST://kafka-1:9092
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka-1:19092,PLAINTEXT_HOST://host:24092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,CONTROLLER:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_CONTROLLER_QUORUM_VOTERS: '1@kafka-1:19093,2@kafka-2:19093,3@kafka-3:19093'
      KAFKA_PROCESS_ROLES: 'broker,controller'
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_OPTS: -javaagent:/usr/app/jmx_prometheus_javaagent.jar=7071:/usr/app/prom-jmx-agent-config.yml
    volumes:
      - ./volumes/kafka-1_data:/var/lib/kafka/data
      - ./volumes/kafka-1_secrets:/etc/kafka/secrets
    deploy:
      resources:
        limits:
          cpus: '6'
          memory: 2.67G
  kafka-2:
    build: .
    container_name: simon_kafka-2
    hostname: kafka-2
    ports:
      - "24093:9092"
    environment:
      KAFKA_NODE_ID: 2
      CLUSTER_ID: '8FbS82vSS3mjE8xPa2AAag'
      KAFKA_CONTROLLER_LISTENER_NAMES: 'CONTROLLER'
      KAFKA_LISTENERS: PLAINTEXT://kafka-2:19092,CONTROLLER://kafka-2:19093,PLAINTEXT_HOST://kafka-2:9092
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka-2:19092,PLAINTEXT_HOST://host:24093
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,CONTROLLER:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_CONTROLLER_QUORUM_VOTERS: '1@kafka-1:19093,2@kafka-2:19093,3@kafka-3:19093'
      KAFKA_PROCESS_ROLES: 'broker,controller'
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_OPTS: -javaagent:/usr/app/jmx_prometheus_javaagent.jar=7071:/usr/app/prom-jmx-agent-config.yml
    volumes:
      - ./volumes/kafka-2_data:/var/lib/kafka/data
      - ./volumes/kafka-2_secrets:/etc/kafka/secrets
    deploy:
      resources:
```

```
53         limits:
54           cpus: '6'
55           memory: 2.67G
56   kafka-3:
57     build: .
58     container_name: simon_kafka-3
59     hostname: kafka-3
60     ports:
61       - "24094:9092"
62     environment:
63       KAFKA_NODE_ID: 3
64       CLUSTER_ID: '8FbS82vSS3mjE8xPa2AAag'
65       KAFKA_CONTROLLER_LISTENER_NAMES: 'CONTROLLER'
66       KAFKA_LISTENERS: PLAINTEXT://kafka-3:19092,CONTROLLER://kafka-3:19093,PLAINTEXT_HOST://kafka-3:9092
67       KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka-3:19092,PLAINTEXT_HOST://host:24094
68       KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,CONTROLLER:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
69       KAFKA_CONTROLLER_QUORUM_VOTERS: '1@kafka-1:19093,2@kafka-2:19093,3@kafka-3:19093'
70       KAFKA_PROCESS_ROLES: 'broker,controller'
71       KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
72       KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
73       KAFKA_OPTS: -javaagent:/usr/app/jmx_prometheus_javaagent.jar=7071:/usr/app/prom-jmx-agent-config.yml
74     volumes:
75       - ./volumes/kafka-3_data:/var/lib/kafka/data
76       - ./volumes/kafka-3_secrets:/etc/kafka/secrets
77     deploy:
78       resources:
79         limits:
80           cpus: '6'
81           memory: 2.67G
```

Listing 5: docker-compose-kafka-kraft-3.yml

The difference here is that KRaft doesn't need ZooKeeper and KRaft isn't even something that needs to be deployed separately. To make the Kafka broker run on KRaft, each broker is also set up as a controller in lines 18, 44 and 70. Additionally, each broker is opened also as a controller on port 19093 on lines 14, 40 and 66, and each controller is connected to each other via quorum voters on lines 17, 43 and 69. Generally an odd number of controllers is wanted, so there won't be ties. Three controllers here allow for one controller failure. (https://docs.confluent.io/platform/current/kafka-metadata/config-kraft.html)

Lastly, the following docker-compose-redpanda.yml shows the docker-compose file that starts up Redpanda broker for Test Case 1 and Test Case 2 :

```
version: "3.8"
services:
  redpanda:
    image: docker-registry/redpandadata/redpanda
    container_name: simon_redpanda
```

```
    ports:
      - "24092:9092"
      - "24644:9644"
    command:
      - redpanda start
      - --kafka-addr internal://redpanda:19092,external://0.0.0.0:9092
      - --advertise-kafka-addr internal://redpanda:19092,external://host:24092
      - --smp 18
      - --memory 8G
    volumes:
      - ./volumes/redpanda:/var/lib/redpanda/data
    deploy:
      resources:
        limits:
          cpus: '18'
          memory: 8G
  console:
    container_name: simon_redpanda_console
    image: docker-registry/redpandadata/console
    entrypoint: /bin/sh
    command: -c 'echo "$$CONSOLE_CONFIG_FILE" > /tmp/config.yml; /app/console'
    environment:
      CONFIG_FILEPATH: /tmp/config.yml
      CONSOLE_CONFIG_FILE: |
        kafka:
          brokers: ["redpanda:19092"]
        redpanda:
          adminApi:
            enabled: true
            urls: ["http://redpanda:9644"]
```

Listing 6: docker-compose-redpanda.yml

The deployment of the Redpanda broker is very similar to the Kafka broker. Along the Redpanda broker, a Redpanda console needs to be started up. The adminApi needs to be enabled in the console for the ability for Prometheus to gather statistics on the broker. Redpanda console also provides a web application to manage and debug streaming data workloads, although it isn't utilized in the test cases of this thesis.

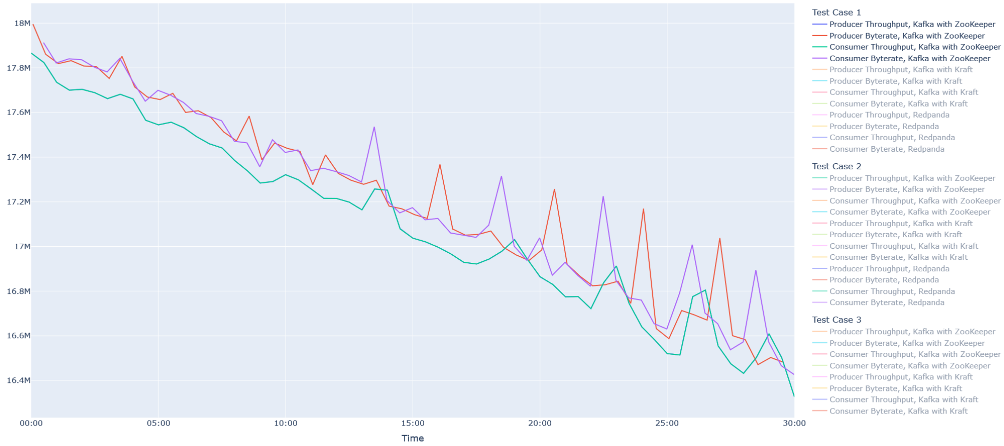
4 Results

This Chapter shows the results gathered from the test cases defined in 3.1. The three Sections of this Chapter will go through the results of each test case. Only the resource usage of the inspected tools will be shown, as the effects of the auxiliary tools (ZooKeeper, Prometheus, Grafana, cAdvisor) on resource usage are negligible in each of the test cases.

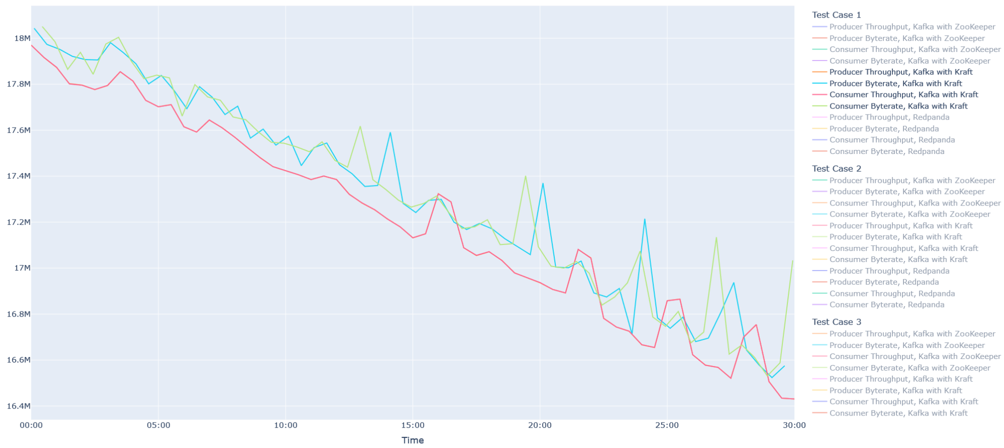
4.1 Evaluation of Event Streaming Platforms in Test Case 1

Figure 3 shows the producer and consumer throughput and byterate comparisons of each of the tools in Test Case 1 . The producer and consumer throughputs are so similar they can't be distinguished in this resolution. This is optimal, as it indicates that the latencies are low and stable. As expected, for each tool both the producer byterate and consumer byterate are slightly higher than the producer and consumer throughputs, and all the byterates and the throughputs are clearly correlated between each other. Each of the platforms have a clear downward trend in both byterate and throughput over time, and given that the tests were ran for essentially an arbitrary amount of time before starting to measure the KPIs and metrics, the rate of change should be considered rather than the absolute values, especially since the absolute values are so close.

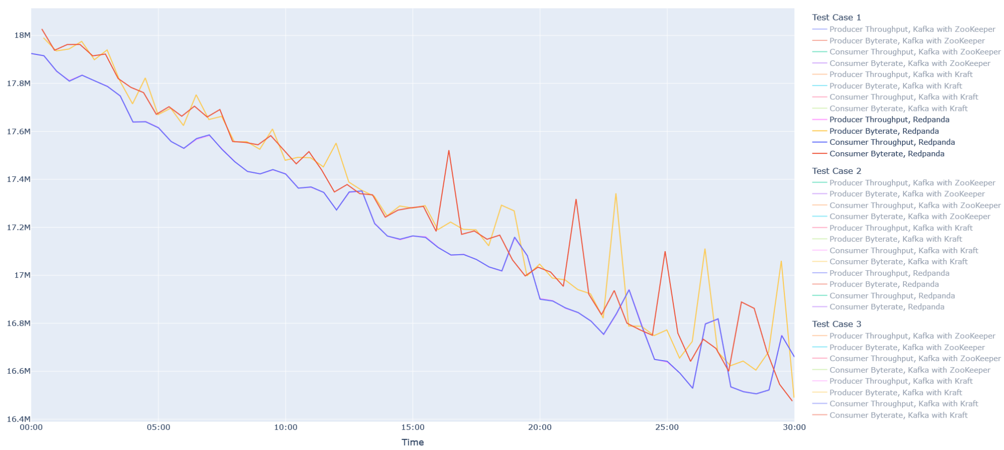
Figure 4 shows the rate of changes in throughput and byterate for each of the tools inspected in Test Case 1 . The throughputs for the platforms using Kafka essentially degrade at the same rate. The throughput of the platform using Redpanda however degrades over the time period somewhat slower, being about 0.12 MB/s higher at minimum (-1.42 MB/s) compared to the other platforms (-1.54 MB/s). For the platform using Kafka with ZooKeeper, the byterate in the producer seems to drop somewhat faster than the byterate in the consumer, having the minimum at about -1.525 MB/s compared to -1.486 MB/s respectively. The opposite is the case for the platform using Redpanda, as the byterate for the producer decreases slightly slower, having the minimum rate of change at -1.500 MB/s compared to 1.551 MB/s. For the platform using Kafka with KRaft the minimum of the rate of change between the producer and consumer on the other hand is essentially equal at -1.520 MB/s.



(a) Throughput and Byterate with Platform Using Kafka with ZooKeeper

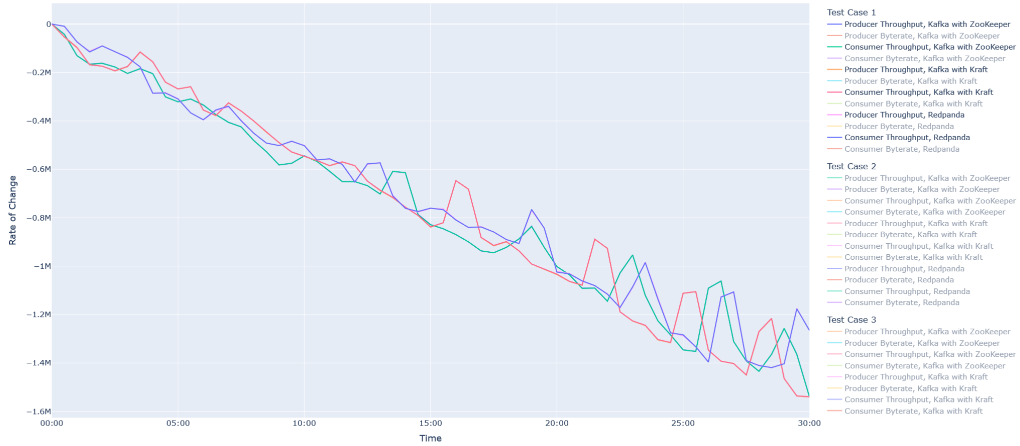


(b) Throughput and Byterate with Platform Using Kafka with KRaft

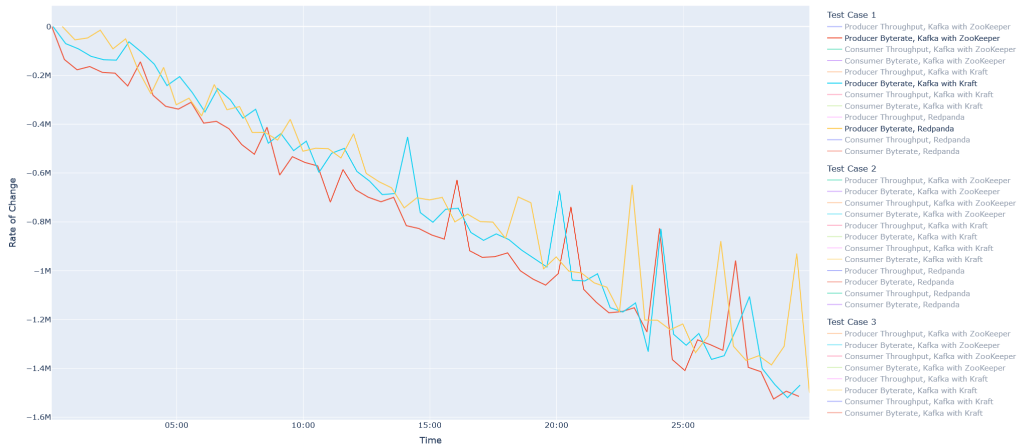


(c) Throughput and Byterate with Platform Using Redpanda

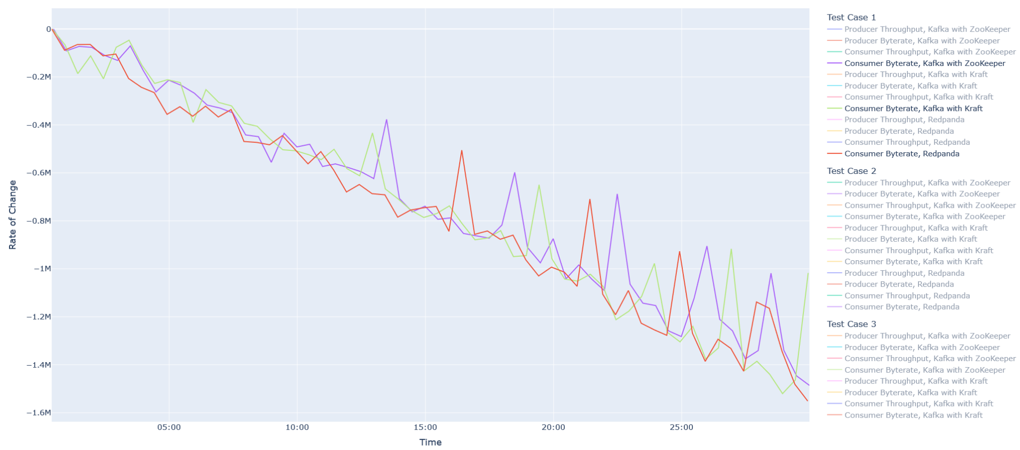
Figure 3: Throughput and Byterate Results for Test Case 1



(a) Producer and Consumer Throughput



(b) Producer Byterate



(c) Consumer Byterate

Figure 4: Rate of Changes in Throughput and Byterate for the Tools Inspected in Test Case 1

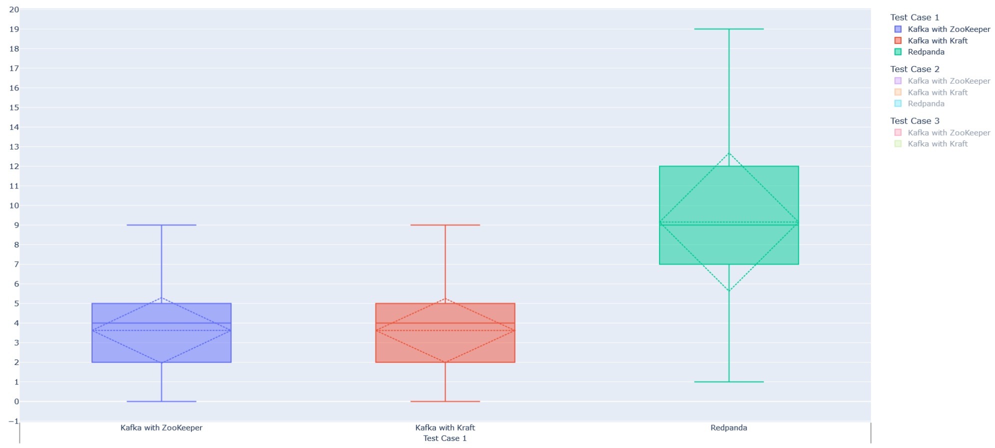


Figure 5: Box Plot of the Latency Distribution in Test Case 1

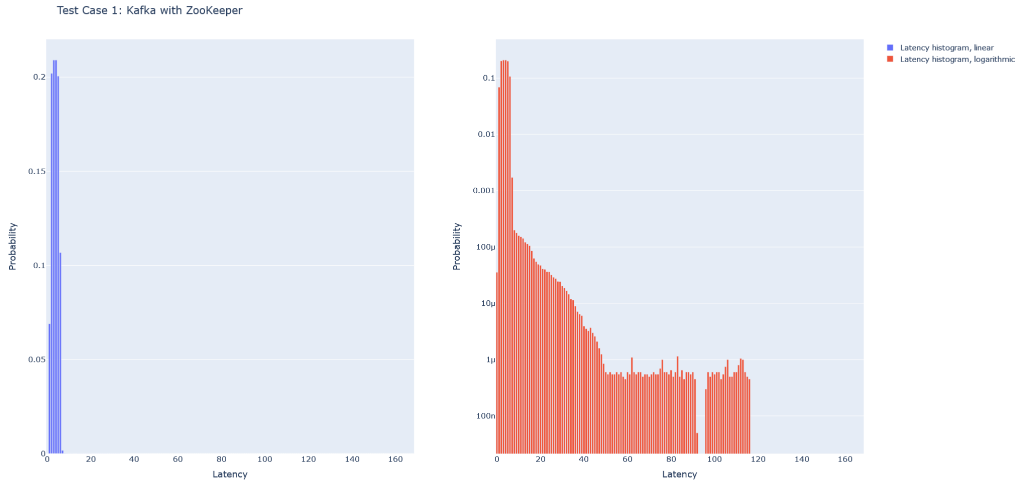
Figure 5 shows a box plot of the latency distributions for each of the tools in Test Case 1 . Often box plots show the outliers, however in this case there are way too many outliers as the size of the data is huge and the highest latencies would mess with the scale too much. The first thing to notice is that the box plot with Kafka using ZooKeeper and Kafka using KRaft has the exact same values other than the ever so slightly higher mean and standard deviation for Kafka using ZooKeeper. The platform using Redpanda however has over twice the higher median (filled line inside the box), mean (straight dashed line), standard deviation (dashed triangles from the mean, indicating the range around the mean and one standard deviation both ways), first and third quartile (bottom and top edge of the box), and lower and upper fence (bottom and top whisker). The box plot shows that at least 50% of the latencies are lower than or equal to 4 ms (median) and 50% are higher than or equal to 4 ms for platforms using either Kafka with ZooKeeper or Kafka with KRaft. The platform using Redpanda on the other hand has 50% below or equal to 9 ms and 50% above or equal to 9 ms. The plot also illustrates that at least 50% of the latencies are between 2 ms (first quartile) and 5 ms (third quartile) and at least 25% are below or equal to 2 ms while at least 25% are above or equal to 5 ms for the platforms using Kafka with ZooKeeper or Kafka with KRaft. With Redpanda, at least 50% of the date is between 7 ms and 12 ms and at least 25% are below or equal to 7 ms and at least 25% are above or equal to 12 ms. The values of the upper and lower whiskers are based on the interquartile range (IQR). The value of IQR is the range between the first quartile and the third quartile and is illustrated in the plot as the bottom and top of the box. The lower whisker is $1.5 * IQR$ lower than the first quartile, while the upper whisker is $1.5 * IQR$ higher than the third quartile. Since the latencies are represented here as integers, in the case that the IQR is odd the lower whisker is essentially rounded up and the upper whisker is rounded down. The IQR of the platform with Kafka is 3 ms, so the lower whisker is at $2ms - 1.5 * IQR = 2ms - 4.5ms = -2.5ms$, but since the lowest possible value is at 0 ms, the lower whisker value is 0 ms. The upper whisker value on the other hand is at $5ms + 1.5 * IQR = 5ms + 1.5 * 3ms = 9.5ms$, which get rounded down to 9 ms. Similarly the lower whisker value of the platform using Redpanda would be 0 ms, but is 1 ms due to it being the minimum of the distribution, while the upper whisker is 19 ms.

Figure 6 shows the histogram for the latencies in Test Case 1 in both linear and logarithmic scale. From this can be seen that over 80% of the latencies when using Kafka with ZooKeeper are between 2 ms and 5 ms, over 90% are between 2 ms and 6 ms and over 95% are between 1 ms and 6 ms. The platform with Kafka using KRaft has the same case, but the maximum value is less than two times that of the maximum value when using Kafka with ZooKeeper. When using Redpanda the latencies are more spread out. Over 60% of the latencies are between 6 ms and 12 ms, and over 85% are between 3 ms and 15 ms.

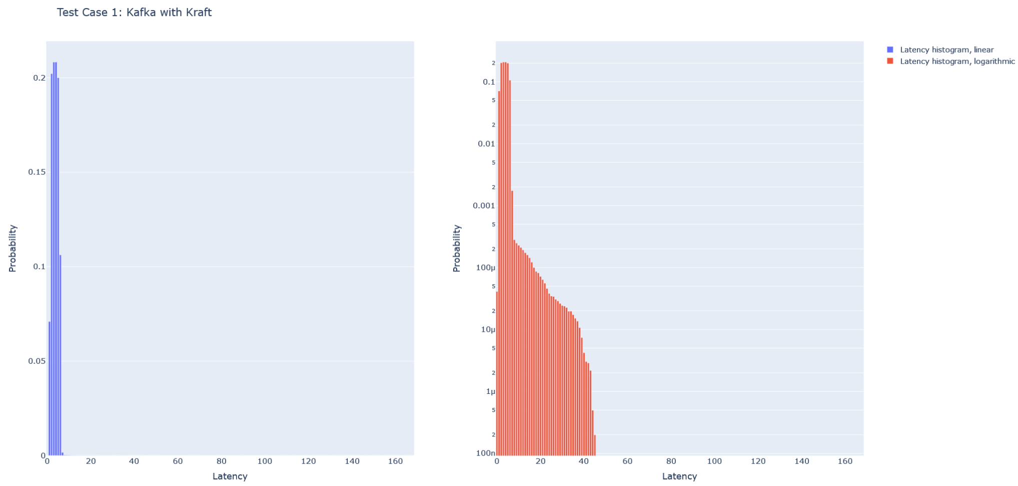
Figure 7 shows the lag over time for each platform inspected in Test Case 1 . Each platform has similar values and a similar shape where it follows an upper value but drop occasionally and each time lower. With Redpanda the lag is on average the highest, ranging between 46 643 and 57 020 messages with a mean of around 54 480 messages. The lowest lag on average is on the platform using Kafka with KRaft with a mean of around 53 759 messages and ranging between 46 831 and 57 146 messages. Lastly, Kafka with ZooKeeper has a mean of around 54 043 messages, ranging between 46 690 and 56 676 messages.

Figure 8 shows the CPU usage of the different event streaming tools used in Test Case 1 . The CPU usage is quite similar when using Kafka whether it's with ZooKeeper or with KRaft. The average is 1.3% lower and the minimum is 0.3% lower when using KRaft compared to using ZooKeeper. The main difference between Kafka with KRaft and Kafka with ZooKeeper here is that occasionally the CPU usage reduces to zero with KRaft while the minimum usage is 9.04% with ZooKeeper. On the other hand, using Redpanda rather than Kafka almost doubles the average and maximum CPU usage with them being 29.6% and 39.7%. The minimum usage here is also 0%, although it only dropped to zero once in the time period compared to the three times with Kafka using KRaft.

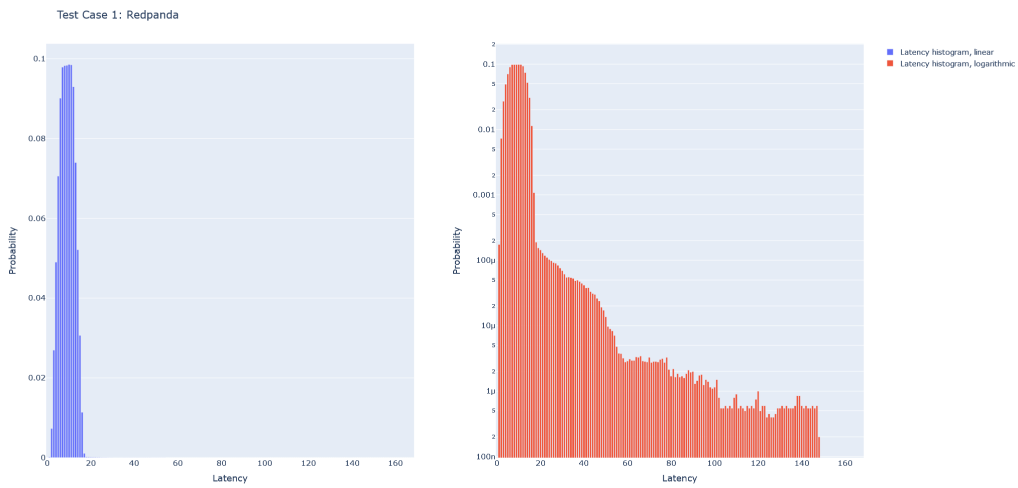
Figure 9 shows the network usage of each of the event streaming tools for Test Case 1 . The sent and received network rates are pretty much equal for each of the platform options. There's almost no difference between the network usage of the platform with Kafka using ZooKeeper and the platform with Redpanda, as the platform using Redpanda has 0.1 MB/s lower average received network rate and 0.2 MB/s lower sent network rate. The platform using Kafka with KRaft on the other hand has a noticeable larger difference with having 1.1 MB/s lower sent and received network rate than the platform using Kafka with ZooKeeper. The maximum on all the platform however is the same 18.1 MB/s for both received and sent network rate. The figure also shows an expected correlation when the network rate drops to zero at same times as the CPU usage drops to zero as shown in Figure 8. The mean differences between the producer and consumer byterates and the sent and received network using Kafka with ZooKeeper are around 1.0 MB/s. With Redpanda this is slightly higher with the mean differences being around 1.3 MB/s for the producer and 1.2 MB/s for the consumer. Using Kafka with KRaft these differences are noticeable higher at around 2.2 MB/s.



(a) Kafka with ZooKeeper



(b) Kafka with KRaft



(c) Producer throughput comparison between tools

Figure 6: Histograms of the Latency Distributions in Test Case 1

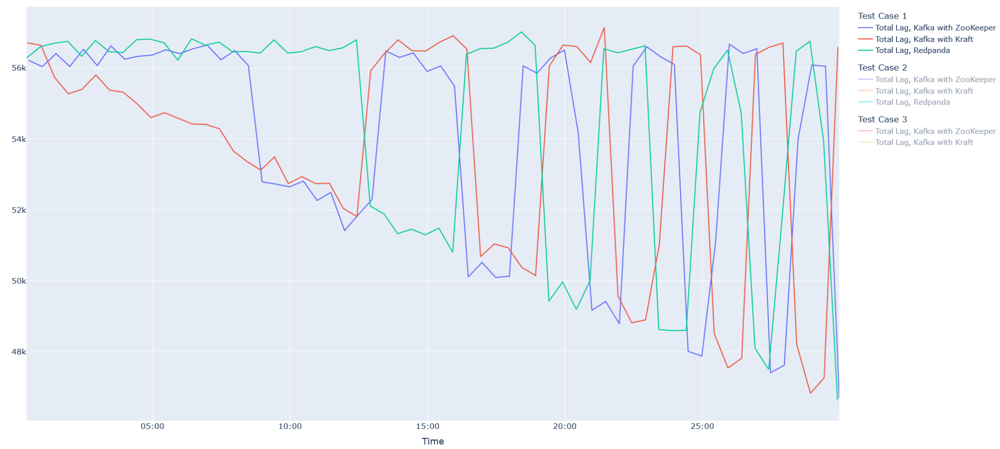
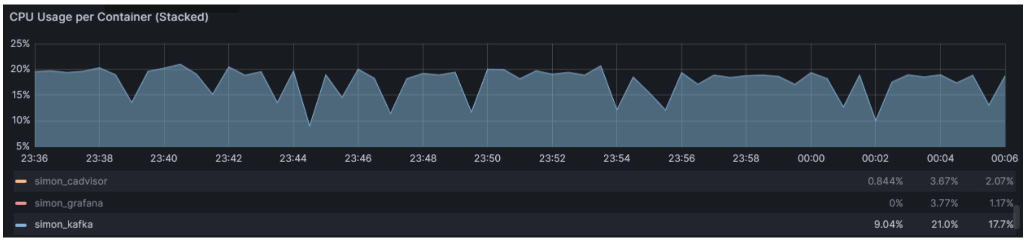
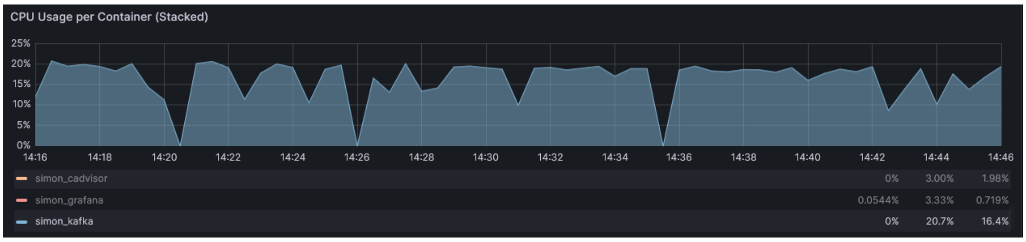


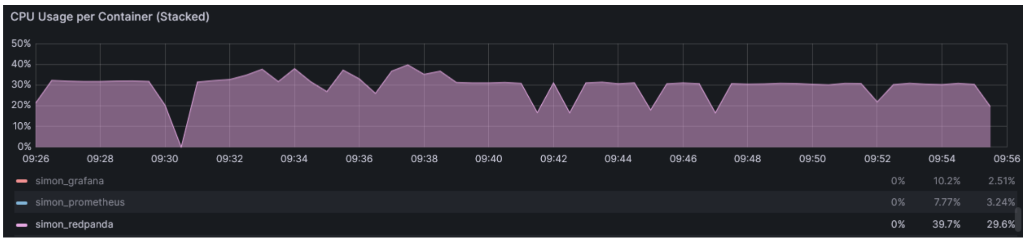
Figure 7: Lag Over Time in Test Case 1



(a) Kafka with ZooKeeper

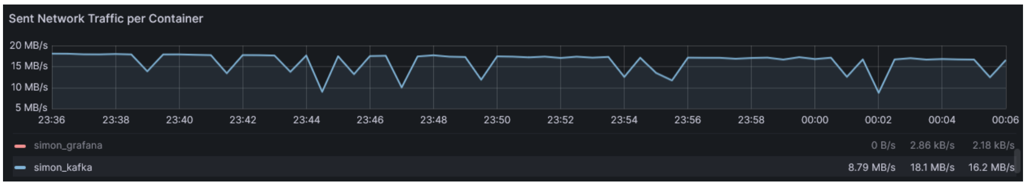


(b) Kafka with KRaft

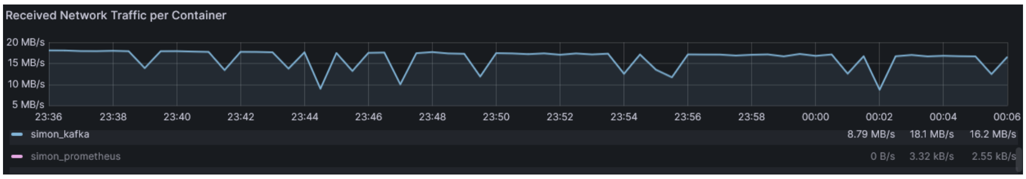


(c) Redpanda

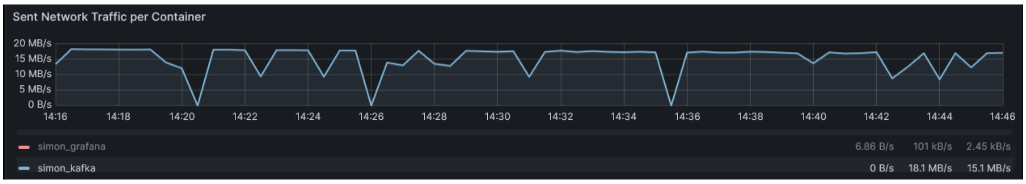
Figure 8: CPU usage in Test Case 1



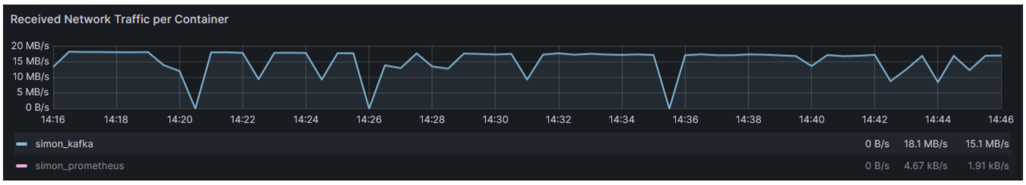
(a) Sent network rate with Kafka using ZooKeeper



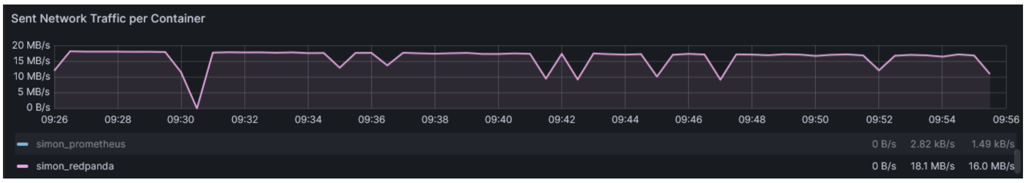
(b) Received network rate with Kafka using ZooKeeper



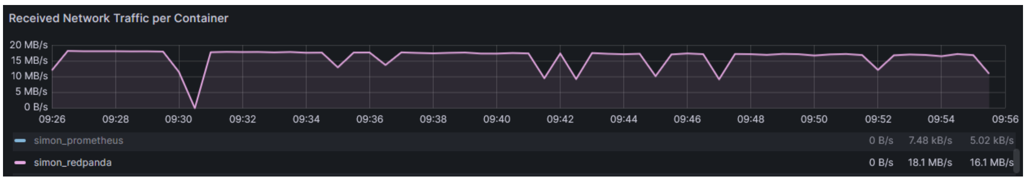
(c) Sent network rate with Kafka using KRaft



(d) Received network rate with Kafka using KRaft

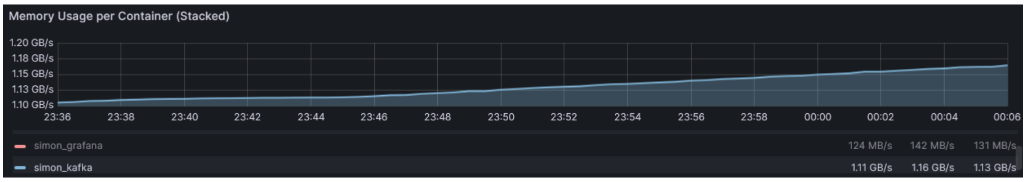


(e) Sent network rate with Redpanda

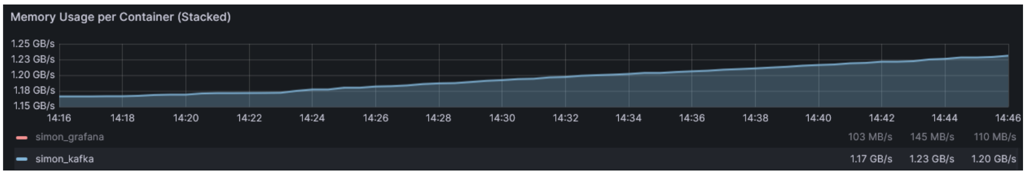


(f) Received network rate with Redpanda

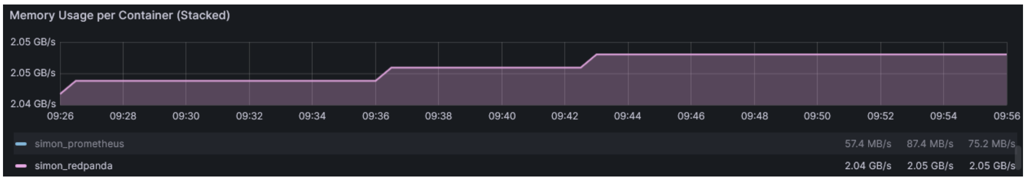
Figure 9: Network usage in Test Case 1



(a) Kafka with ZooKeeper



(b) Kafka with KRaft



(c) Redpanda

Figure 10: Memory usage in Test Case 1

Figure 10 shows the memory usage of the platforms for Test Case 1 . The platform with Kafka using ZooKeeper has the lowest usage during the time frame and the platform with Redpanda has the highest. The memory usage for each of the platforms increases over time. The platform with Kafka using KRaft has the fastest increase from 1.17 GB to 1.23 GB, while Kafka using ZooKeeper has an increase from 1.11 GB to 1.16 GB. Although the platform with Redpanda has the highest memory usage during the time frame, it does have the slowest increase from 2.04 GB to 2.05 GB.

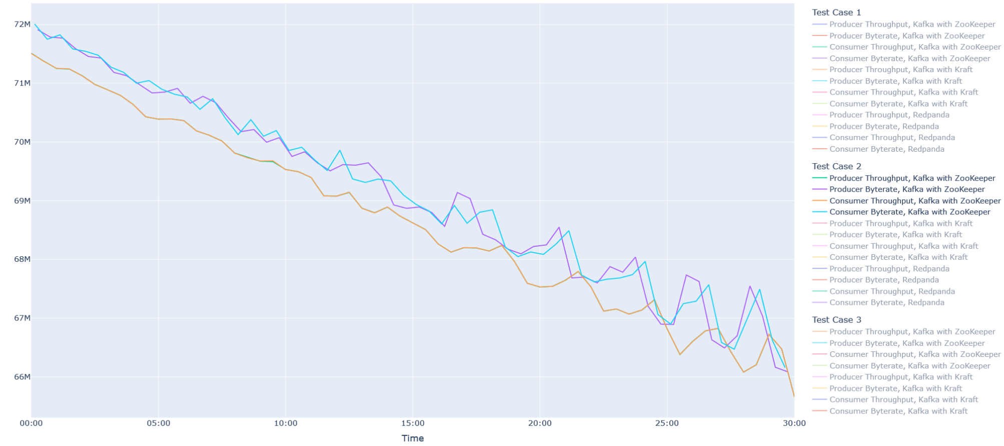
Overall, the platforms with Kafka performed similarly. The only noticeable difference is that the platform using Kafka with KRaft has lower sent and received network rates. Therefore the platform with KRaft should generally be used due to its simpler architecture and requiring lower network rates if that is a bottleneck. The platform with Redpanda on the other hand has some noticeable potential issues compared to the other platform. It has a larger latency distribution spread and over twice higher mean and median, it has an over 10 percentage units higher mean CPU usage and it has almost double mean memory usage. The advantages the platform using Redpanda has over the other platforms is that it has a slightly slower throughput and very slow memory usage degradation, so it is possible that if the platforms were to be run for a very long time, Redpanda could remain stable for longer. It's also possible that the higher CPU and memory usages simply enable Redpanda to have the slightly slower throughput degradation, but it would still scale as well as Kafka even if the load is increased.

4.2 Evaluation of Event Streaming Platforms in Test Case 2

Figure 11 shows the producer and consumer byterates and throughputs for Test Case 2 . Similarly as in Test Case 1 , the producer and consumer throughput differences can barely be seen. The differences are a little bit more noticeable, which indicates either higher or more unstable latencies. Also the byterates and throughputs are all clearly correlated with each other as expected. The downward trend over time is apparent here as well for the byterates and throughputs, so only the rates of change should be considered.

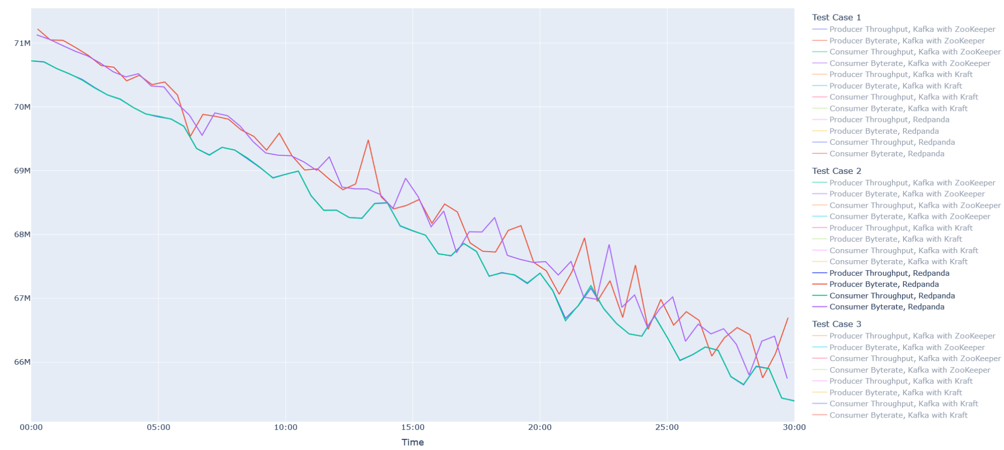
Figure 12 shows the rate of change in throughput and byterate for each of the tools inspected in Test Case 2 . The throughput degrades fastest for the platform using Kafka with ZooKeeper by about 5.849 MB/s. The throughput of the platfrom using Kafka with KRaft also degrades faster than the platform using Redpanda by 5.598 MB/s compared to 5.323 MB/s. For each of the platforms the degradation per producer is lower than in Test Case 1 , with the same degradation as in Test Case 1 the platforms with Kafka would approximately be around 6.16 MB/s and the platform with Redpanda would be around 5.68 MB/s. For this test case the byterate degradation between the producers and consumers are much closer than in Test Case 1 . For the platform using Kafka with ZooKeeper the degradation in byterate is also the fastest, for the producer being around 5.828 MB/s and for the consumer being around 5.851 MB/s. The slowest degradation also in byterate is the platform using Redpanda, with the producer being around 5.466 MB/s and the consumer around 5.384 MB/s, compared to the producer byterate degradation being around 5.503 MB/s and the consumer byterate degradation being around 5.530 MB/s with the platform using Kafka with KRaft. Just as with throughput the byterate degradation is slower per producer than in Test Case 1 . For the platform using Kafka with ZooKeeper the producer byterate degradation with the same degradation rate as in Test Case 1 would be around 6.100 MB/s and the consumer byterate would be around 5.944 MB/s. For the platform using Kafka with KRaft the producer and consumer byterates would be around 6.080 MB/s and for the platform using Redpanda the producer byterate degradation would be around 6.000 MB/s and the consumer byterate degradation around 6.204 MB/s.

Figure 13 shows the box plot of the latency distribution for Test Case 2 . Both platforms using Kafka have the same median, first and third quarter, and lower and upper whisker as they had in Test Case 1 and thus also equal values between each other. This time however the standard deviation is noticeable higher going well out of the IQR range. Also the means are slightly higher, this time the platform using Kafka with KRaft having a slightly higher mean than the platform using Kafka with ZooKeeper. The boxplot of the latency distribution of the platform using Redpanda is also very similar to its counterpart in Test Case 1 . The third quarter is the same 12 ms, while the median and first quarter is one higher at 10 ms and 8 ms. Since the first quarter is higher, the IQR is now smaller and the lower whisker is therefore one higher at 2 ms and the upper whisker is one lower at 18 ms.



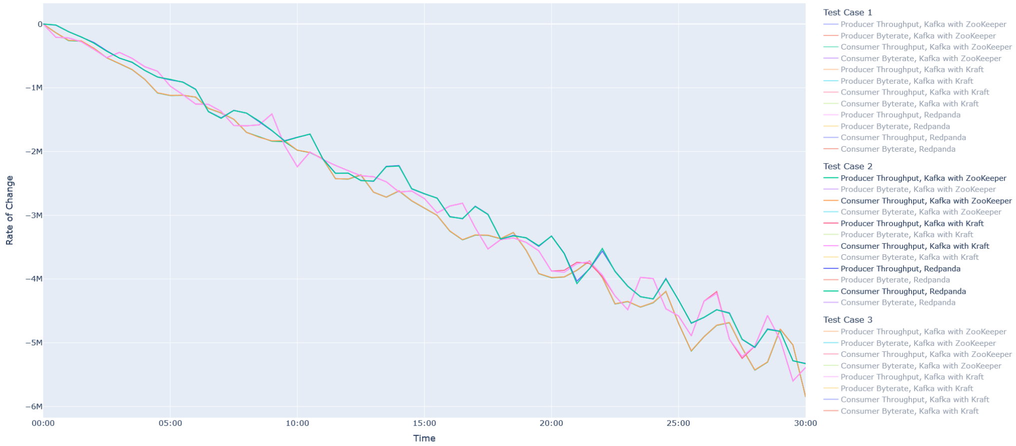
(a) Throughput and Byterate with Platform Using Kafka with ZooKeeper

(b) Throughput and Byterate with Platform Using Kafka with KRaft

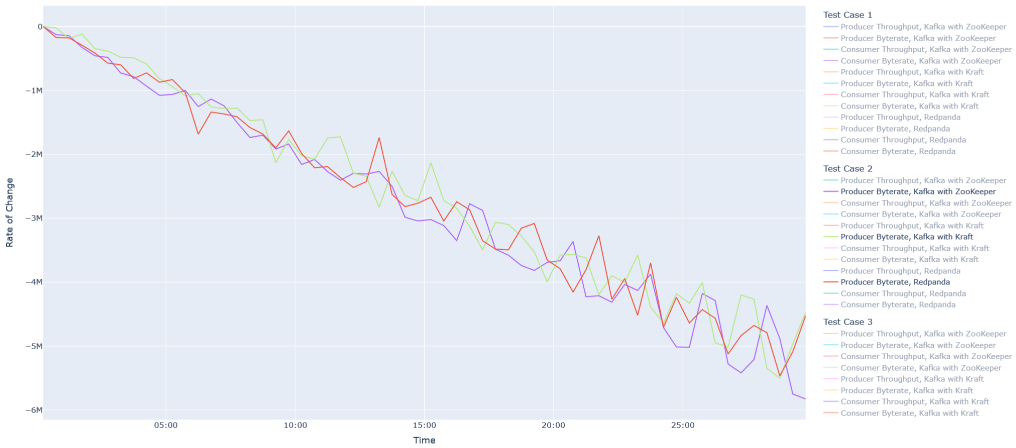


(c) Throughput and Byterate with Platform Using Redpanda

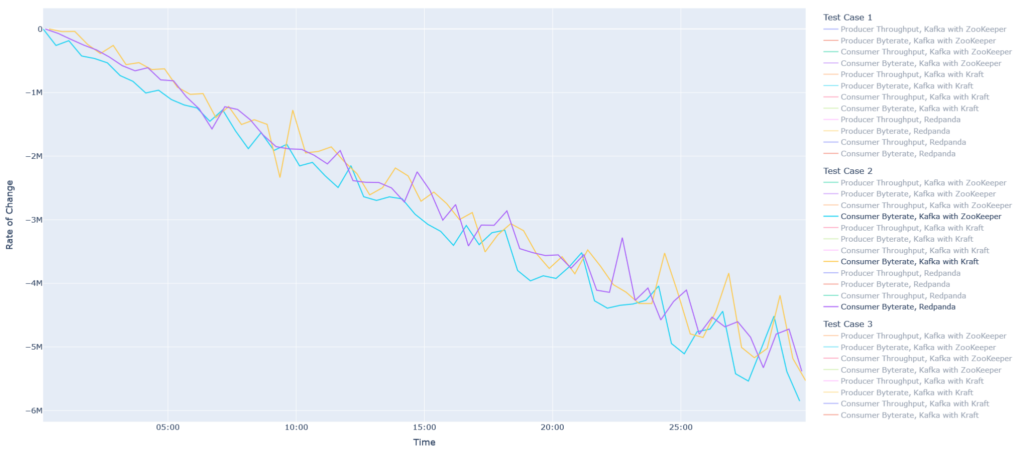
Figure 11: Throughput and Byterate Results for Test Case 2



(a) Producer and Consumer Throughput



(b) Producer Byterate



(c) Consumer Byterate

Figure 12: Rate of Changes in Throughput and Byterate for the Tools Inspected in Test Case 2

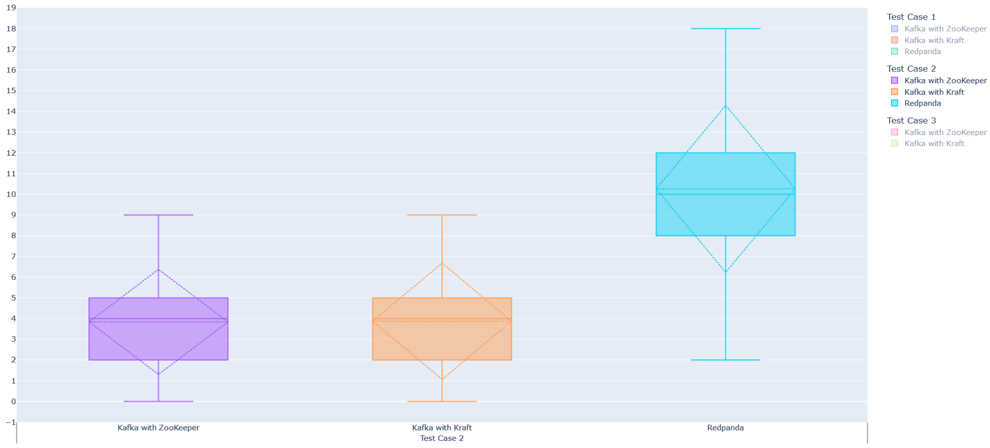
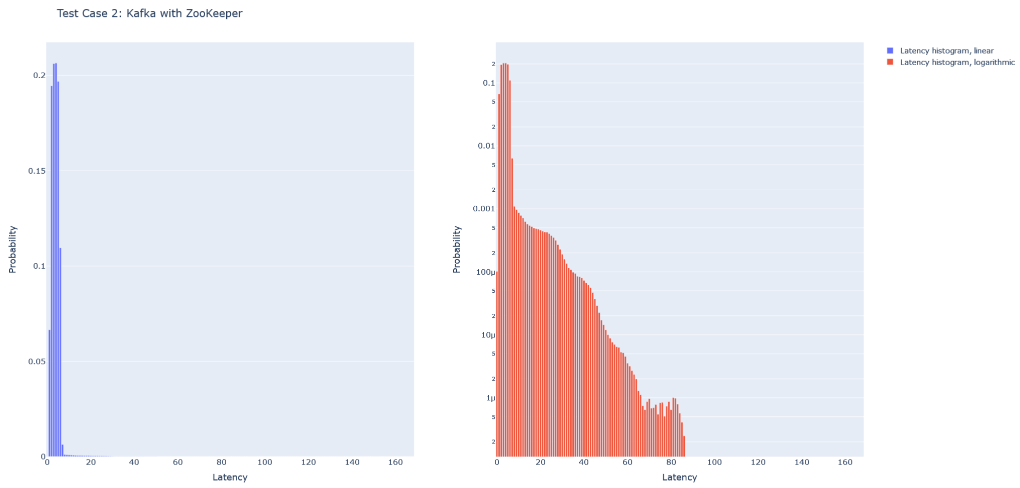


Figure 13: Box Plot of the Latency Distribution in Test Case 2

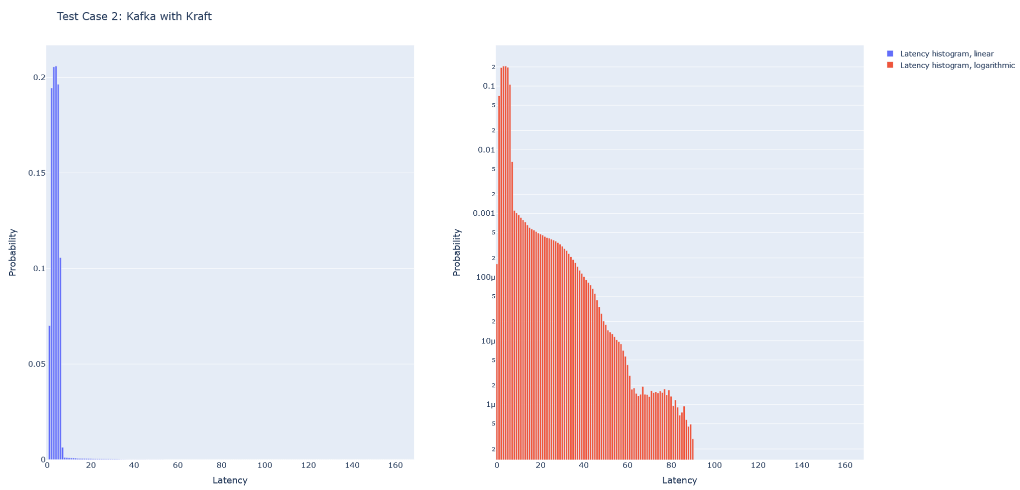
Figure 14 shows the histograms for the latency distributions of the platforms inspected in Test Case 2 . The same conclusions apply to both platforms using Kafka as in Test Case 1 . With the platform using Redpanda there's a slight shift, where over 60% of the latencies are now between 7 ms and 13 ms. One difference for each of the platforms is that now the end tails are visible even in linear scale.

Figure 15 shows how lag over time for each platform inspected in Test Case 2 . The lags are now shown with two plots due to the large difference in scale. The lag on the platforms using Kafka with KRaft and Redpanda is now significantly higher than in Test Case 1 . The highest lag on average is with the platform using Kafka with KRaft with a mean of around 214 382 messages, ranging between 194 657 and 225 295 messages. The platform with Redpanda has a mean of around 214 057 messages, ranging between 197 828 and 223 469 messages. Interestingly, the platform using Kafka with ZooKeeper on the other hand has a much lower lag than in Test Case 1 , even though it has a similar latency distribution compared to the platform using Kafka with KRaft in both test cases. On average it's only around 574 messages, ranging between 228 and 1 260 messages. This likely implies that the platform using Kafka with ZooKeeper could efficiently handle even larger loads with one broker, while the other platforms might start to struggle.

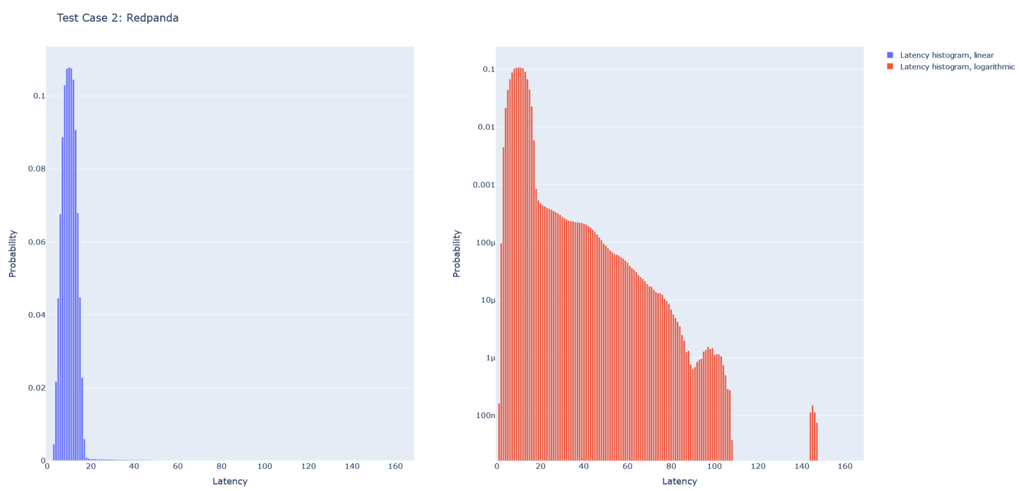
Figure 16 shows the CPU usage of the platforms with different tools for Test Case 2 . Both platforms with Kafka have comparable usage ranging from 29.3% to 62.5% averaging 56.2% when using ZooKeeper and ranging from 34.5% to 63.3% averaging 56.3% when using KRaft. The platform with Redpanda has much larger usage with averaging 76.5% and having the maximum of 92.4%. Similarly to Test Case 1 , there's a single dip to 0%.



(a) Kafka with ZooKeeper

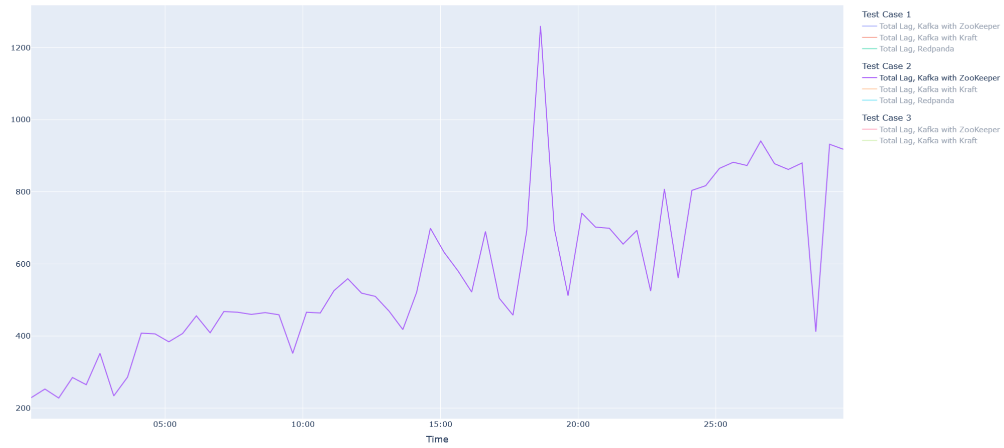


(b) Kafka with KRaft

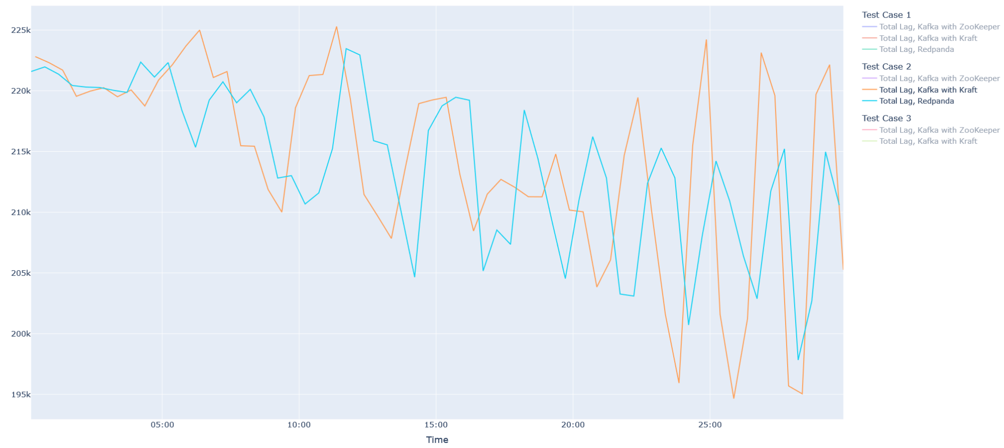


(c) Producer throughput comparison between tools

Figure 14: Histograms of the Latency Distributions in Test Case 2

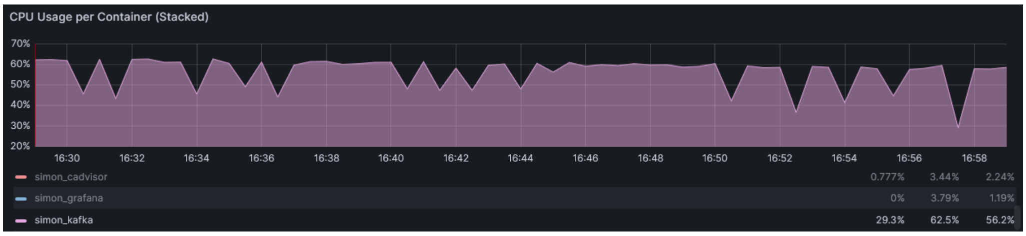


(a) Kafka with ZooKeeper

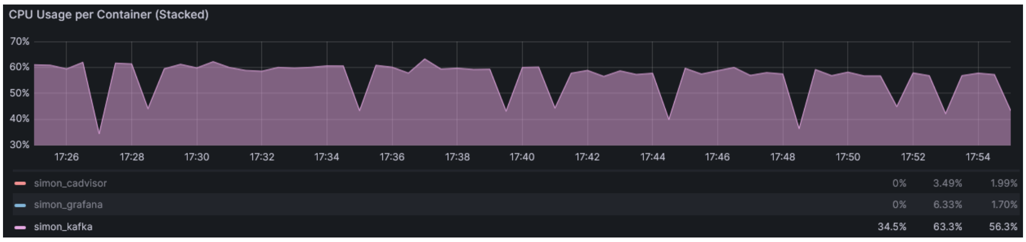


(b) Kafka with KRaft and Redpanda

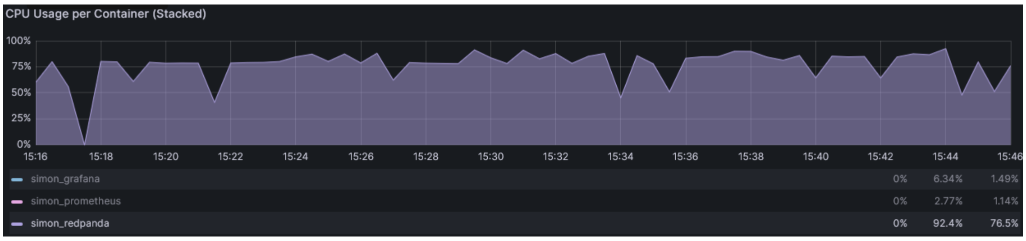
Figure 15: Lag Over Time in Test Case 2



(a) Kafka with ZooKeeper

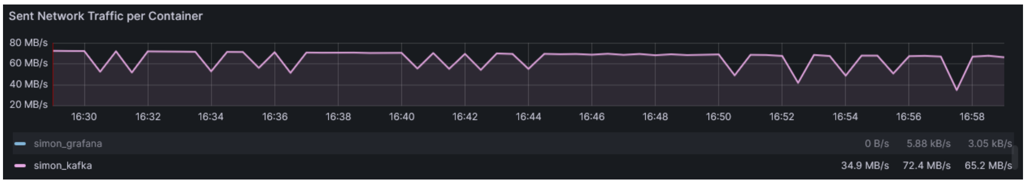


(b) Kafka with KRaft

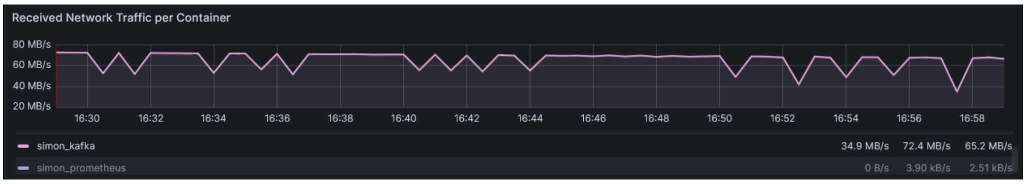


(c) Redpanda

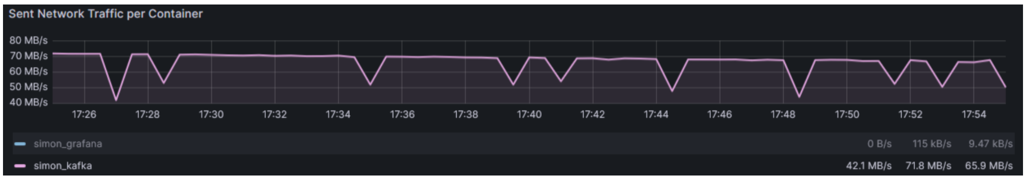
Figure 16: CPU usage in Test Case 2



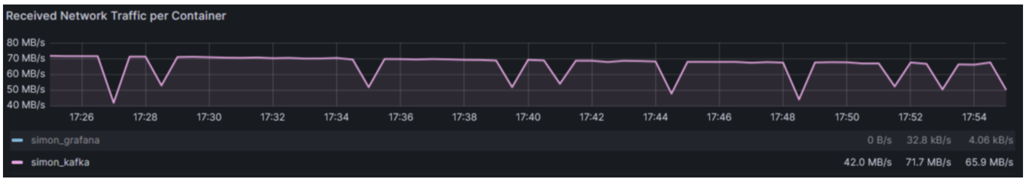
(a) Sent network rate with Kafka using ZooKeeper



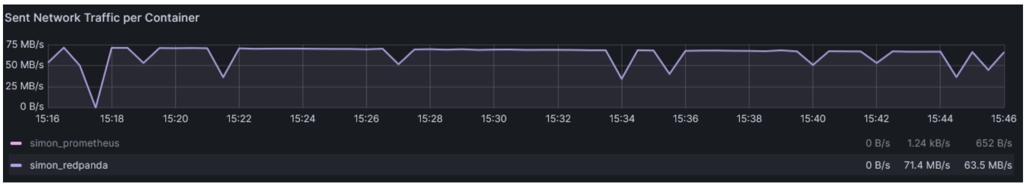
(b) Received network rate with Kafka using ZooKeeper



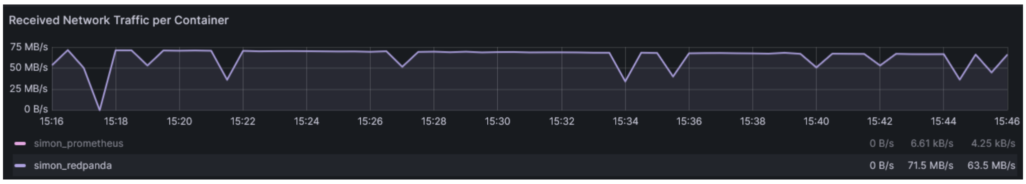
(c) Sent network rate with Kafka using KRaft



(d) Received network with Kafka using KRaft



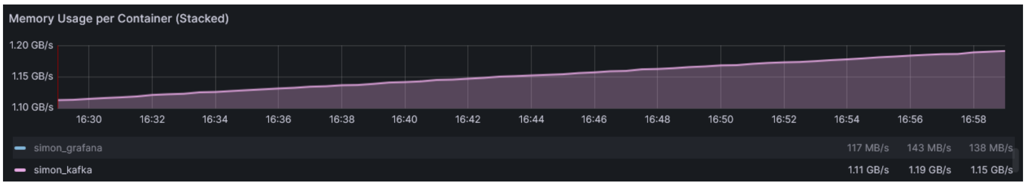
(e) Sent network rate with Redpanda



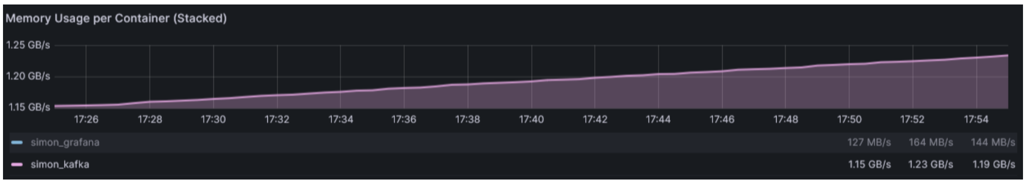
(f) Received network rate with Redpanda

Figure 17: Network usage in Test Case 2

Figure 17 shows the sent and received network rate for each platform inspected in Test Case 2 . While Kafka using KRaft had the lowest network rates in Test Case 1 , it now has the highest with sent rate ranging from 42.1 MB/s to 71.8 MB/s and received rate ranging from 42.0 MB/s to 71.7 MB/s both averaging at 65.9 MB/s. Kafka with ZooKeeper is the platform with second highest network rates with both sent and received network rates ranging from 34.9 MB/s to 72.4 MB/s and averaging at 65.2 MB/s. This time the platform with the lowest network rate is the one using Redpanda with the averages for both sent and received network rates being somewhat lower compared to the other platforms at 63.5 MB/s. The maximum for the sent network rate is 71.4 MB/s and received network rate is 71.5 MB/s. As expected both sent and received network rate has dropped to zero at the same time as the CPU usage has dropped to zero shown in Figure 16 similarly as in Test Case 1 . In this test case the smallest difference between the sent and received network rates and producer and consumer byterates are with the platform using Kafka with KRaft, that had the highest difference in Test Case 1 with it now being 2.7 MB/s. Using Kafka with ZooKeeper the differences are noticeable higher at 3.9 MB/s. With Redpanda the differences are even higher at 5.0 MB/s.



(a) Kafka with ZooKeeper



(b) Kafka with KRaft

(c) Redpanda

Figure 18: Memory usage in Test Case 2

Figure 18 shows the memory usage of each of the platforms using different event streaming tools. The results are similar to how they are in Test Case 1 shown in Figure 10. The platform with Kafka using ZooKeeper has the lowest mean memory usage increasing from 1.11 GB to 1.19 GB, which is only slightly faster than the increase from 1.11 GB to 1.16 GB in Test Case 1 . The platform with Kafka using KRaft again has the second lowest mean memory usage increasing from 1.15 GB to 1.23 GB, which also is slightly faster than from 1.17 GB to 1.23 GB in Test Case 1 . Again, the platform with Redpanda has the most stable usage in the time frame, but the usage is over four times higher than with the other tools with a mean of 4.90 GB, which is over twice higher than in Test Case 1 , that had an usage increase from 2.04 GB to 2.05 GB.

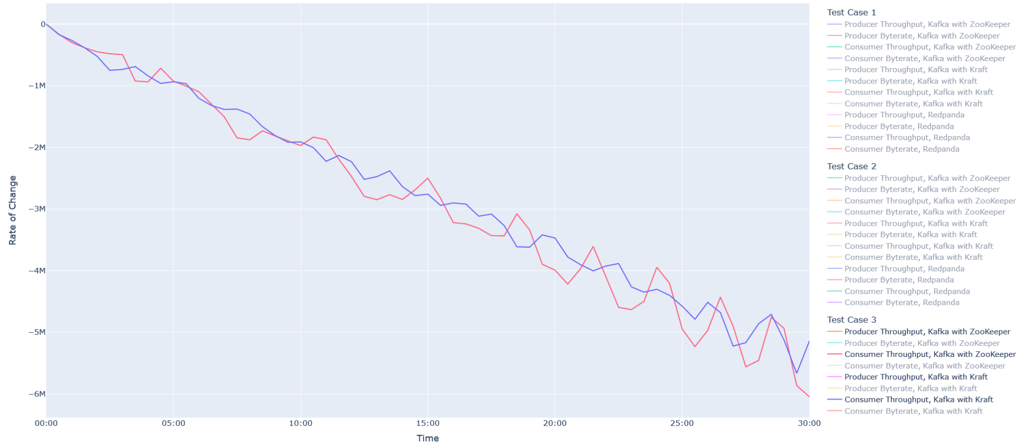
Similarly to Test Case 1 , both platforms using Kafka have similar performances. The main difference now being that the platform using Kafka with ZooKeeper has orders of magnitudes lower lag than the other platforms. Although this huge difference in lag, it doesn't correlate with the latencies of the platforms, as both platforms using Kafka have similar distributions. One smaller difference is that Kafka with KRaft has slightly slower throughput degradation, but again Redpanda has the slowest. As in Test Case 1 , Redpanda again has a larger spread in latency distribution and over twice higher mean and median in it, over 20 percentage points higher CPU usage and over four times higher mean memory usage, but again way slower degradation in it. This time however, the platform with Redpanda has also the lowest sent and received network rates.

4.3 Evaluation of Event Streaming Platforms in Test Case 3

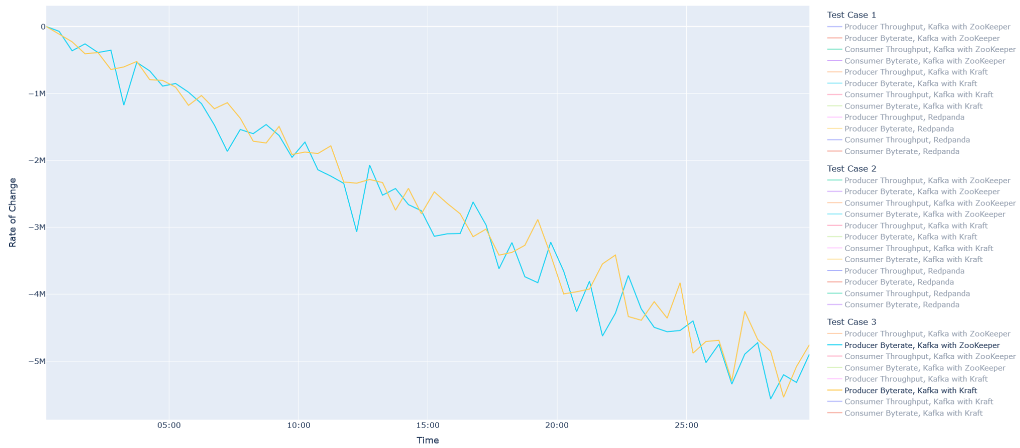
(a) Throughput and Byterate with Platform Using Kafka with ZooKeeper

(b) Throughput and Byterate with Platform Using Kafka with KRaft

Figure 19: Throughput and Byterate Results for Test Case 3



(a) Producer and Consumer Throughput



(b) Producer Byterate

(c) Consumer Byterate

Figure 20: Rate of Changes in Throughput and Byterate for the Tools Inspected in Test Case 3

Figure 19 shows the producer and consumer byterates and throughputs in Test Case 3 . Once again the difference in throughputs between the producer and the consumer cannot be seen with this resolution, as was similarly the case in Test Case 1 and Test Case 2 . Also similarly, there is a clear correlation between each of the throughputs and byterates.

Figure 20 shows the rate that the throughput and byterate changes over time for each of the tools inspected in Test Case 3 . For this test case the degration in throughput for the platform using Kafka with KRaft is somewhat lower being almost 0.4 MB/s compared to the platform using Kafka with ZooKeeper. For the platform using Kafka with KRaft it is around 5.663 MB/s and for the the platform using ZooKeeper it is about 6.046 MB/s. For both platforms these are slightly higher than in Test Case 2 . For the platform using Kafka with ZooKeeper the byterate degradation in producer is significantly lower than in the consumer, being around 5.589 MB/s compared to 5.873 MB/s. For the platform using Kafka with KRaft these are lower, with the producer byterate degradation being around 5.538 MB/s and the consumer byterate degradation being slightly lower at 5.506 MB/s. Compared to the byterate degradation in Test Case 2 , for the platform using Kafka with ZooKeeper the degradation is slower for the producers and only slightly slower for the consumer. For the platform using Kafka with KRaft the byterate degradations are very close together, being slightly faster for the producer and slighlty slower for the consumer.

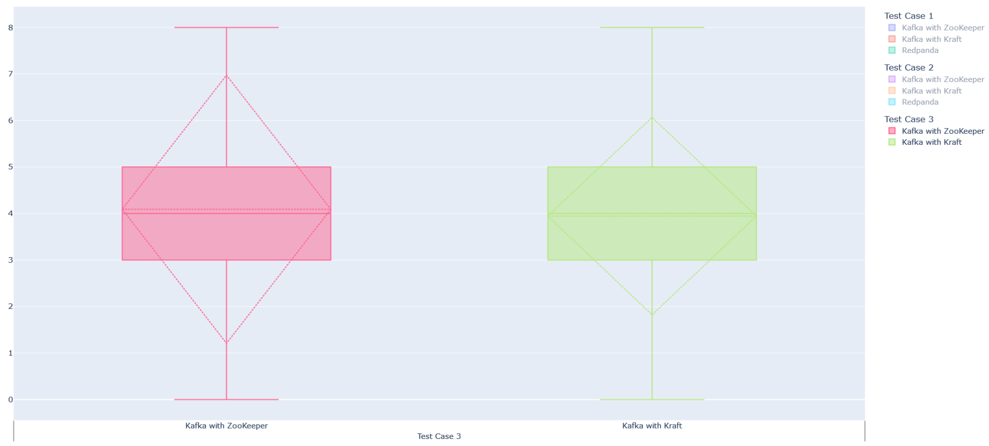
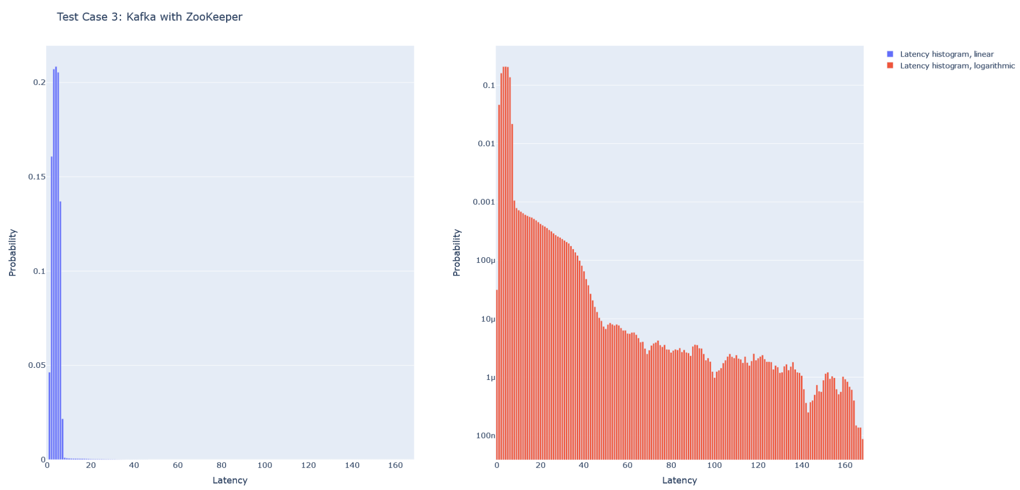


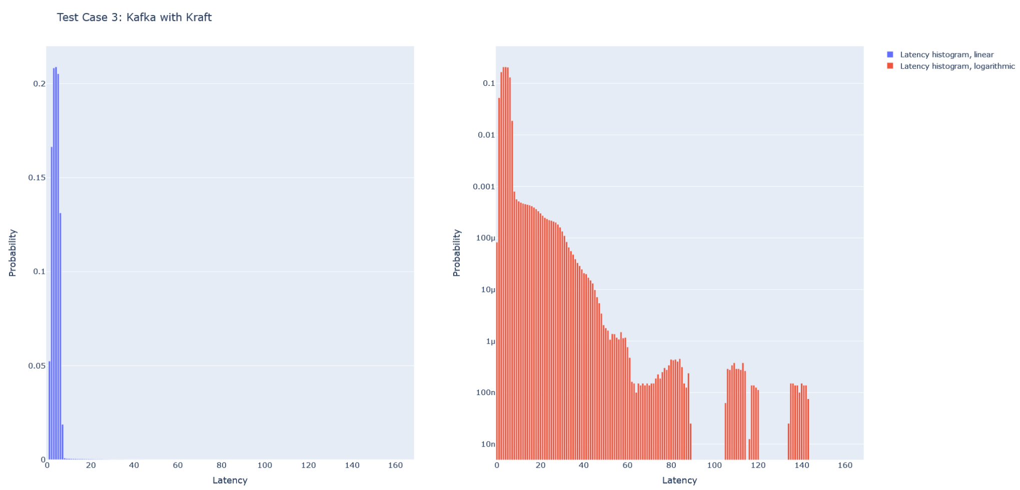
Figure 21: Box Plot of the Latency Distribution in Test Case 3

Figure 21 shows the box plot for the latency distribution of the platforms inspected in Test Case 3 . Both platforms have the same median and third quarter as in Test Case 1 and Test Case 2 . However the first quarter is now one higher at 3 ms, meaning that the IQR is one less at 2 ms and thus the upper whisker is at 8 ms. The lower whisker is however still at 0 ms, since in the previous test cases it was at 0 ms only because it couldn't go lower. This time the platform with Kafka using ZooKeeper has a higher standard deviation than in previous test cases, while the platform with Kafka using KRaft has a lower standard deviation than in Test Case 2 , but still higher than in Test

Case 1 . Both platform however have a higher mean than in the previous test cases.



(a) Kafka with ZooKeeper



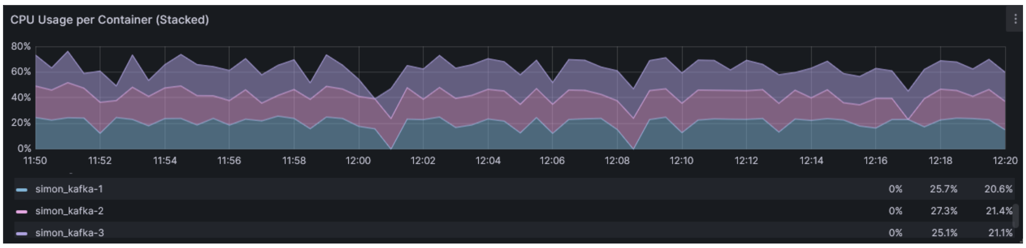
(b) Kafka with KRaft

Figure 22: Histograms of the Latency Distributions in Test Case 3

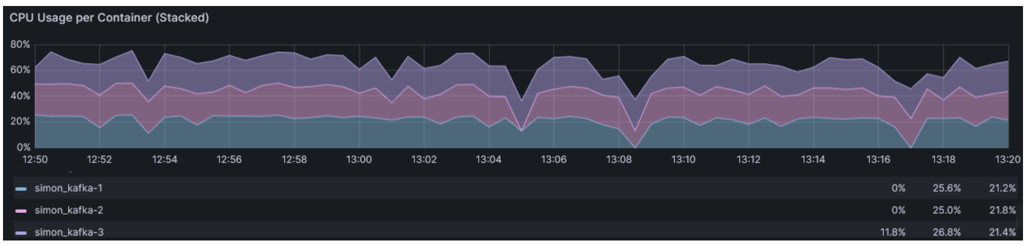
Figure 22 shows the histograms of the latency distributions for platforms inspected in Test Case 3 . Both platform have now slightly more spread out latencies, but still over 75% of the latencies are between 2 ms and 5 ms, while over 85% are between 2 ms and 6 ms, and over 90% are between 1 ms and 6 ms.

Figure 23 shows the total lag over time for the platforms inspected in Test Case 3 . As there are three brokers in this test case, the total lag is calculated as a sum over all brokers. Now the total lags between the platform using Kafka with ZooKeeper and Kafka with KRaft are very close together, compared to the huge difference in Test Case 2 . With the platform using Kafka with ZooKeeper the lag is now slightly higher, but with the platform using Kafka with KRaft the lag is way lower. The average with Kafka using KRaft is slightly lower averaging at around 1647 messages, ranging between 667 and 2799 messages. The average with Kafka using ZooKeeper on the other hand is around 1683 messages, ranging between 622 and 2789 messages. However, here's the issue that the lag seems to increase over time. By the difference between the maximum and minimum, the lag in the platform using Kafka with ZooKeeper increases slightly faster.

Figure 23: Lag Over Time in Test Case 3



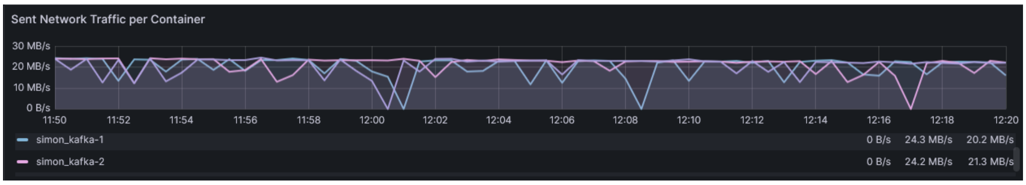
(a) Kafka with ZooKeeper



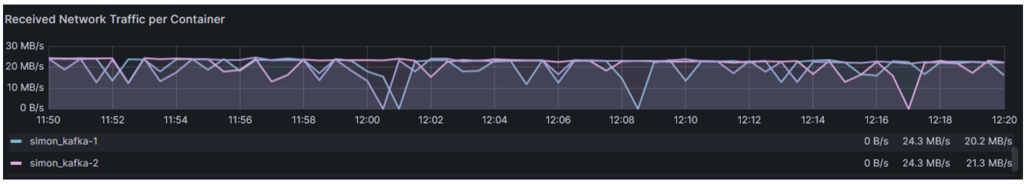
(b) Kafka with KRaft

Figure 24: CPU usage in Test Case 3

Figure 24 shows the CPU usage of each broker in each of the platforms inspected in Test Case 3 . The platform with Kafka using ZooKeeper has slightly lower total average CPU usage compared to the platform with Kafka using KRaft being 63.1% compared to 64.4%. These both are also respectively about 7% and 8% higher than in Test Case 2 , being 56.2% and 56.3%. The averages for each broker in the platform with Kafka using ZooKeeper are 20.6%, 21.4% and 21.1%, while the maximums are 25.7%, 27.3% and 25.1%. The averages for each broker in the platform with Kafka on the other hand are 21.2%, 21.8% and 21.4%, while the maximums are 25.6%, 25.0% and 26.8%. For the platform using ZooKeeper, two of the brokers drop to zero CPU usage once and one of them twice. With KRaft however one of the brokers doesn't drop to zero even once, while similarly one of them drops once and one of them twice.



(a) Sent network rate with Kafka using ZooKeeper



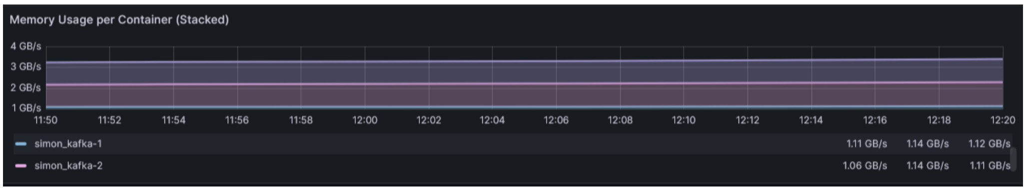
(b) Received network rate with Kafka using ZooKeeper

(c) Sent network rate with Kafka using KRaft

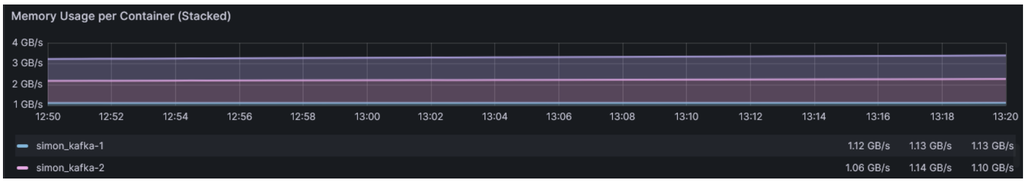
(d) Received network rate with Kafka using KRaft

Figure 25: Network usage in Test Case 3

Figure 25 shows the sent and received network rates for each broker for each of the platforms inspected in Test Case 3 . Overall the platform using Kafka with KRaft has a slightly higher sent and received network than the platform using Kafka with ZooKeeper. With KRaft the total usage averages to about 62.7 MB/s and with ZooKeeper to about 62,25 MB/s, which are less than in Test Case 2 , but each broker has a higher network rate than in Test Case 1 . As in Test Case 1 and Test Case 2 , when the CPU usage drops to zero as shown on 24 on a broker, the network usage also drops to zero for that same broker. Compared to Test Case 2 , the platform using Kafka with KRaft now has a noticeable higher difference between the producer and consumer byterates and the sent and received network rates at around 5.7 MB/s. However, the platform using Kafka with ZooKeeper still has a higher difference at around 6.55 MB/s.



(a) Kafka with ZooKeeper



(b) Kafka with KRaft

Figure 26: Memory usage in Test Case 3

Figure 26 shows the memory usage for each of the brokers in both of the platforms inspected in Test Case 3 . Each broker still uses slightly over 1 GB of memory, so as this test case uses three brokers rather than one as in the other test cases, the total memory usage is also about three times higher compared to the previous test cases. Both platforms average around the same 3.345 GB total memory usage, but with KRaft there's a slower increase from about 3.27 GB to 3.405 GB, compared to from 3.255 GB to 3.42 GB.

As in both Test Case 1 and Test Case 2 , the performances between both platforms using Kafka are very similar. As in Test Case 2 , Kafka with KRaft has a somewhat slower throughput degradation. Opposed to Test Case 2 , there is now no massive difference in lag and now it increases over time.

Overall across each test case and tool, the mean producer and consumer throughputs are over 99% of the mean producer and consumer byterates. For the platform using Kafka with ZooKeeper these ranges between 99.33% and 99.38%. Using Kafka with KRaft, there's a bit more variation with the ratio ranging between 99.30% and 99.42%. The platform using Redpanda has the most efficient ratio, with the mean ranging between 99.35% and 99.47%.

5 Conclusion

The main aim of this thesis was to compare different event streaming platforms in different test cases to make some general conclusions on their performance. The three event streaming platforms to be tested were two separate platforms using Apache Kafka, one with the more mature ZooKeeper to have Kafka brokers connected with each other and the other with a more recent Raft protocol, that is included native to Kafka. The third platform is using Redpanda, which attempts to be an evolved version of Kafka by working with the existing Kafka architecture, including being fully Kafka API-compatible.

In Chapter 2 the thesis explained the most relevant information to understand streaming infrastructure. The first section explained and illustrated the modern architecture to handle streaming data, called the Kappa architecture. The next section explained the event streaming platforms, that handle the data store of the Kappa Architecture, but often also handle the data ingestion through producers and the access to the data store in the application/serving layer through consumers. Another important part of the Kappa architecture is the stream processing, which is handled by stream processing tool explained in section 2.3. Lastly, the measured KPIs and metrics were provided and explained.

The test cases for the event streaming platforms were introduced in 3.1. The first test case was the simplest one, with one producer, consumer, broker, topic and partition. The second test case was the same, but increases the number of producers to four. Similarly the third test case was the same as the second test case other than now the number of brokers and partitions (one per broker) are increased to three.

The performance of these tools are measured with KPIs and metrics. The KPIs were used mainly to compare the performance of the platforms were throughput and latency, while the other metrics were used to consider the limitations and bottlenecks of the platforms were producer and consumer byterates, latency, lag and CPU/memory usage. Also the average record size was measured to make sure there weren't differences in the sizes of the records, as their generation had slight randomness.

There was an issue in measuring the throughputs of the tools, as they degraded over time. The mean throughputs of the tools were however so close to each other, it made sense to only consider the rate of change in the throughput over time. The rates were similar in each of the test cases between the tools, but the platform using Redpanda degraded slightly slower than the other platforms in Test Case 1 and Test Case 2 and the platform using Kafka with KRAft degraded slightly slower than the platform using Kafka with ZooKeeper degraded slightly slower in Test Case 2 and Test Case 3 . Both platforms using Kafka had essentially the same latency distribution in each test case, while with Redpanda the distribution was way more spread and the mean and median were over twice higher. Therefore the choice between the platforms depends highly on the latency requirements, as if the latency is mostly irrelevant it might make sense to use Redpanda for the slight improvement in throughput degradation.

One important future research topic is to measure how the tools will react to even higher loads. There weren't really any issues with one broker and one consumer handling the data of four producers. This would show if the high CPU and memory usage of Redpanda is actually a problem, or if it simply enables the lower throughput degradation of the platform. Similarly, it would show if the orders of magnitudes higher lag seen in Test Case 2 would become an issue for the platforms using Kafka with KRaft or Redpanda. Another potential furute research topic include an even more novel event streaming platform similar to Kafka and Redpanda called Liftbridge [11]. Also one potential future research topic is to deploy way more (even thousands) producers, brokers and consumers with container orchestration system such as Kubernetes or Nomad. Of course one research topic is to also change more of the default settings of the platforms or parameters used in the test cases. For example the number of partitions per broker could be increases or data could be generated to more than one topic.

References

This is the list of references to the sources cited in appendix **??**. The list more or less follows the Vancouver style (IEEE). See appendix **??** for a detailed exposition on cross-referencing and bibliography styles. Follow the description there.

[1] Y. Fu and C. Soman, "Real-time data infrastructure at Uber", *Proceedings of the 2021 International Conference on Management of Data*, New York, USA, 2021, pp. 2503-2516, doi: 10.1145/3448016.3457552.

[2] A. Wang, "Building Scalable Real Time Event Processing with Kafka and Flink®", Doordash Engineering, August 02 2022. [Online]. Available: https://doordash.engineering/2022/08/02/building-scalable-real-time-event-processing-with-kafka-and-flink/. [Accessed: Oct. 5, 2023].

[3] E. Feller, "Data processing architectures – Lambda and Kappa examples", The Ericsson Blog, November 23 2015. [Online]. Available: https://www.ericsson.com/en/blog/2015/11/data-processing-architectures--lambda-and-kappa-examples. [Accessed: Oct. 5, 2023].

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

[5] J. Samosir, M. Indrawan-Santiago, P. D. Haghighi, "An Evaluation of Data Stream Processing Systems for Data Driven Applications", *Procedia Computer Science*, vol. 80, 2016, pp. 439-449, doi: 10.1016/j.procs.2016.05.322.

[6] J. Bang and M. Choi, "Docker environment based Apache Storm and Spark Benchmark Test", *2020 21st Asia-Pacific Network Operations and Management Symposium (APNOMS)*, Daegu, South Korea, 2020, pp. 322-325, doi: 10.23919/APNOMS50412.2020.9237049.

[7] J. Kreps, "Questioning the Lambda Architecture", O'Reilly, July 02, 2014. [Online]. Available: https://www.oreilly.com/radar/questioning-the-lambda-architecture/. [Accessed: Aug. 9, 2023].

[8] A. Sanla and T. Numnonda, "A Comparative Performance of Real-time Big Data Analytic Architectures", *2019 IEEE 9th International Conference on Electronics Information and Emergency Communication (ICEIEC)*, Beijing, China, 2019, pp. 1-5, doi: 10.1109/ICEIEC.2019.8784580.

[9] Apache Kafka, "Kafka 3.6 Documentation". [Online]. Available: https://kafka.apache.org/documentation/. [Accessed: Oct. 13, 2023].

[10] Redpanda, "Redpanda Documentation". [Online]. Available: https://docs.redpanda.com/current/home/. [Accessed: Oct. 13, 2023].

[11] Liftbridge. [Online]. Available: https://liftbridge.io/docs. [Accessed: Oct. 13, 2023].