Analysis of queuing services

A Project Report
Presented to
The Faculty of the College of
Engineering

San Jose State University
In Partial Fulfillment
Of the Requirements for the Degree
Master of Science in Software Engineering

By
Sanjay Nag Bangalore Ravishankar, Atish Maitreya, Daniel Sampreeth Reddy
Eadara and Viraj Upadhyay

May 2020

APPROVED

Prof. Saldamli Gokay, Project Advisor

ABSTRACT

Distributed architecture is where multiple nodes or machines work together as a unit. High availability, resiliency, and scalability are some of the top reasons for using a distributed architecture. In this architecture, the project is decoupled into smaller parts such that they are easier to develop and maintain. One big challenge in this architecture is the way these systems communicate and distribute data within one another. Messaging queues provide an excellent solution to this problem as they allow asynchronous sending and receiving of data. They also simplify decoupling because they serve as message-oriented middleware between applications, and are easily scalable. Choosing a particular message queuing service for a system depends on many factors. For example, Amazon SQS is a good option for an easy, managed service. Apache Kafka, on the other hand, is a highly reliable, scalable and high throughput system. Choosing a wrong service could decrease the performance of the whole system. In this project, we compare the aforementioned message queuing services and test their performance in the exact same architecture.

Acknowledgments

The authors are deeply indebted to Professor Xiao Su for her invaluable comments and assistance in the preparation of this study.

Analysis of Queuing systems in a distributed architecture

Atish Maitreya, Daniel Sampreeth Reddy Eadara, Sanjay Nag Bangalore Ravishankar, Viraj Upadhyay
Computer Engineering Department
San José State University (SJSU)
San José, CA, USA
Email: {atish.maitreya, danielsampreethreddy.eadara, sanjaynag.bangaloreravishankar, viraj.upadhyay}@sjsu.edu

***Abstract*—Distributed architecture is where multiple nodes or machines work together as a unit. High availability, resiliency, and scalability are some of the top reasons for using a distributed architecture. In this architecture, the project is decoupled into smaller parts such that they are easier to develop and maintain. One big challenge in this architecture is the way these systems communicate and distribute data within one another. Messaging queues provide an excellent solution to this problem as they allow asynchronous sending and receiving of data. They also simplify decoupling because they serve as message-oriented middleware between applications, and are easily scalable. Choosing a particular message queuing service for a system depends on many factors. For example, Amazon SQS is a good option for an easy, managed service. Apache Kafka, on the other hand, is a highly reliable, scalable and high throughput system. Choosing a wrong service could decrease the performance of the whole system. In this project, we compare the aforementioned message queuing services and test their performance in the exact same architecture.**

***Index Terms*—Distributed architecture, Message Queues, Amazon SQS, Apache Kafka**

I. Introduction

Distributed systems - a group of computers working together as a single unit, are extremely important in this ever-growing technological world. Although they share states, they are independent of each other. They support failures much more efficiently, implying that the failure of a single system does not always affect the system as a whole. Vertical scaling means increasing the capacity and power of a single piece of hardware, whereas, horizontal scaling is increasing the number of machines and dividing the load. Distributed systems enable us to scale horizontally. For large-scale applications, horizontal scalability is crucial. Message queues are a mode of asynchronous communication between the distributed systems. They offer temporary storage services, processing depth, are reliable and scalable. There are many message queues available currently such as Kafka, Amazon SQS, RabbitMQ etc. Each message queue has its own set of advantages. A user has to decide on using a particular message queue based on the requirements. This paper shows results from the comparison of Amazon SQS and Apache Kafka and discusses their differences.

II. Project Architecture

1) **Technology Stack**

a) **Middleware** : NodeJS servers will be acting like Middleware. These contain the application's APIs. The load balancer will auto-scale the middleware containers based on the number of requests. When a request is made to an endpoint, it hits the load balancer. This request from the load balancer is passed to one of the middleware servers.

b) **Database** : The requests from the messaging queues are consumed by the data source. MongoDB Atlas is used to serve as database. The database setup will remain consistent for testing both Kafka and SQS. This ensures the performance test is isolated to the message queues, hence providing accurate results.

2) **Message Queues**

Message queues are between the backend and the database of architecture. They enable applications to exchange information. They have the ability to store messages (packets of data) as well as process the data. They can rather be said as the means of transferring streams of data from the sender to the receiver. The entity where the items are held in a stream, holding until they are processed is called a queue. The message queue here is the entity that holds messages until they are processed when requested by the receiver. Message queues can be reliable, provide high throughput and scalability depending on the requirement. They avoid data loss with their ability to store messages and allows the continuation of processes even when the system fails.

3) **Kafka**

Kafka is a distributed streaming platform. Kafka is developed by LinkedIn. LinkedIn has a requirement for collecting user engagement data. This is fulfilled by Kafka. Kafka has the ability to process offline. This is fundamentally used for the log processing of data. Kafka is used for applications with a requirement in processing a high amount of data. The downside here is an occasional loss while handling huge loads of data. Kafka's data is called topics which are streams of data. These streams of data are made up of partitions that are of the same size. Partitions are owned by the users and these users can own more than a single partition.

Each record of data consists of a timestamp, a value, and a key. The sender publishes the topics while the receiver consumes them. Kafka is generally applicable for building a real-time streaming data pipeline to transfer data reliably between applications and for building real-time streaming applications that respond to the streams of data. Receivers store the data using the transfer of bytes which are called iterators. Kafka uses the pull models for streamlining the messages. This enables the data to be stored and transferred depending on the request of receivers. Kafka uses zookeeper to keep track of its topics and partitions. Zookeeper acts as a centralized service for providing synchronization within the distributed system. It triggers the synchronization process in case a change is detected. Kafka consists of four API's namely Producer API, Connector API, Stream API, and Consumer API. [1]

a) Producer API: This API enables the producer to publish the data to topics. One or more topics can be published at a time. [1]
b) Connector API: The connector API is used for connecting multiple systems. This connectivity is done by passing the data through an output or connecting the input data into a system. This enables the reuse of existing data. [1]
c) Streams API: The streams API acts as a mode of connection between the incoming and outgoing topics. It provisions for processing streams of data from more than one incoming topic and delivers it to one or more outgoing topics. [1]
d) Streams Consumer API: This API enables the receiver to subscribe to a particular topic already published by a producer. A receiver can subscribe to one topic or more. [1]

4) **Amazon SQS**
Amazon SQS allows for scaling microservices and decoupling them along with distributed systems and applications. This service is based majorly on providing scaling service. Amazon makes things easier by providing the server to its users, unlike other messaging queues. Amazon SQS works asynchronously meaning, senders and recipients need not work in correlation with each other. They can publish and subscribe to messages at their own rate.
[2] Amazon SQS offers two types of message queues:

a) Standard message queue: These type of message queues do not maintain order but rather correlate to the swift and efficient delivery of messages. Duplication of messages can be a downside. This is majorly used for cases requiring high throughput.

b) FIFO:This type of message queues offers delivery of messages in a first come first serve order. This offers low throughput. There will not be any duplication of messages. The priority of messages is maintained, and each message is processed only once.
The major benefits offered by Amazon SQS are the elimination of administrative overhead, reliable delivery of messages, securing sensitive data, scale data elastically and cost-effectively. The Amazon SQS works in a way where it makes use of a visibility timeout clock while sending a message to the queue. The recipient component also makes use of the timeout clock while accessing the message. The timeout clock helps in processing the message in a particular timeframe. A standard message retention period is enabled and when a message has to wait for more than the retention period, it's deleted. [2]

III. System Design

A) Setup
Based on various factors, such as, throughput, requests per second, response time, latency, and failures, the performance of the system is tested. The application is deployed on Amazon Web Services (AWS) for benchmarking. The message queuing services used are Amazon Managed Streaming for Apache Kafka (MSK) and Amazon Simple Queue Service (SQS). The database service used is MongoDB Atlas. Lastly, the automated load testing service used is Locust. [4]

B) Implementation
For the back-end of the application, two NodeJS servers are hosted on AWS m4.xlarge instances which provides high network performance and has the hardware configuration of 4 vCPUs and 16 GB RAM. [6] These servers sit behind a classic load balancer on which, requests are served. The database service uses the cluster tier MongoDB Atlas M40, which runs on 16 GB RAM and 4 vCPUs with 1500 IOPS. [5] Locust is deployed on AWS m4.xlarge instance as well. For Amazon SQS, the queue type is standard queue and a lambda worker consumes the message from the queue. Apache Kafka version 2.2.1 [1] is used with 6 brokers across three availability zones and the broker instance type kafka.m5.large. [3] The encryption is not enabled within the cluster and the topic partitions are 5. The Kafka consumer is also hosted on AWS m4.xlarge instance.

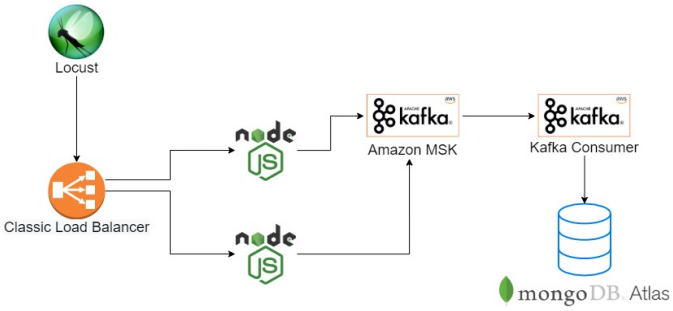

Fig. 1. Architecture - Kafka.

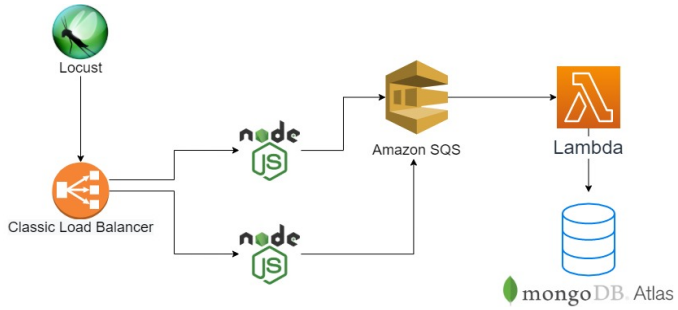

Fig. 2. Architecture - SQS.

IV. Experiment Results

a) Amazon SQS

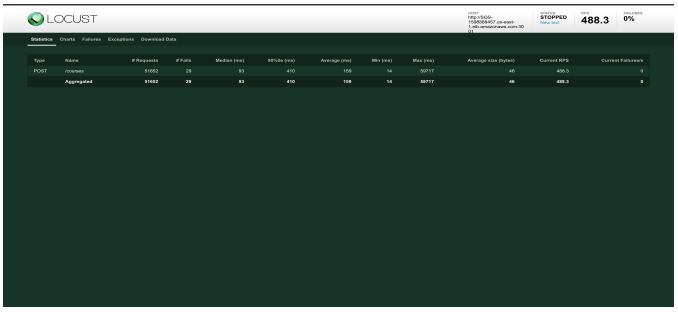

Fig. 3. Locust result.

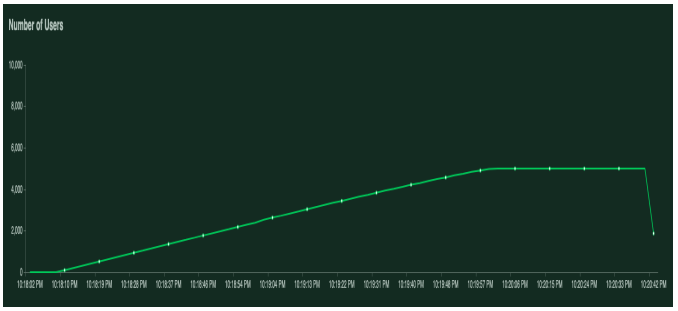

Fig. 4. Number of users.

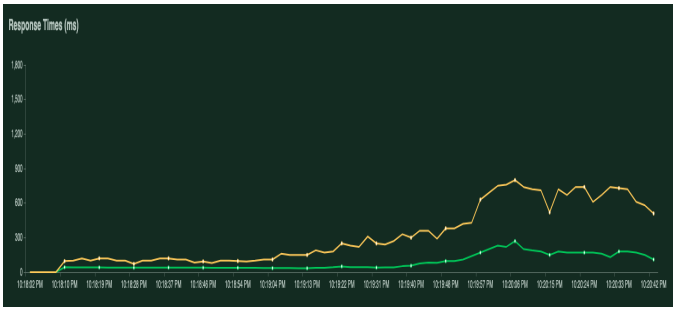

Fig. 5. Response times.

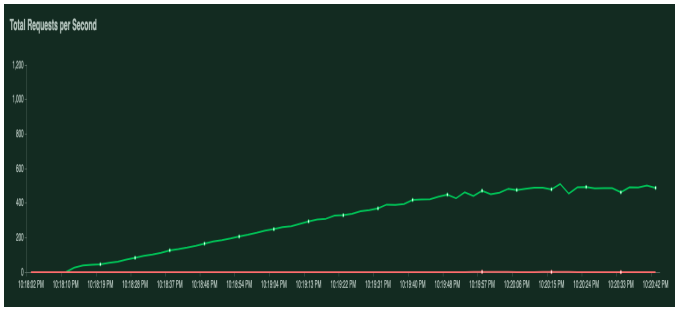

Fig. 6. Total requests per second.

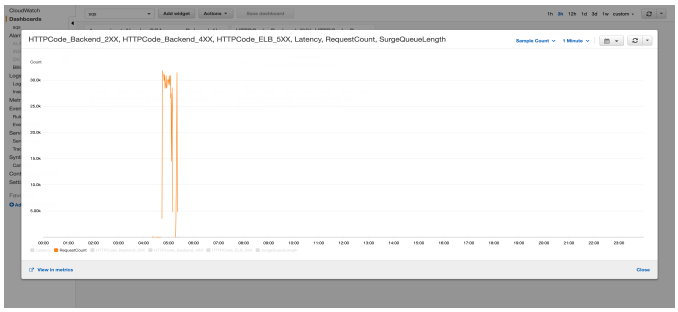

Fig. 7. Request count.

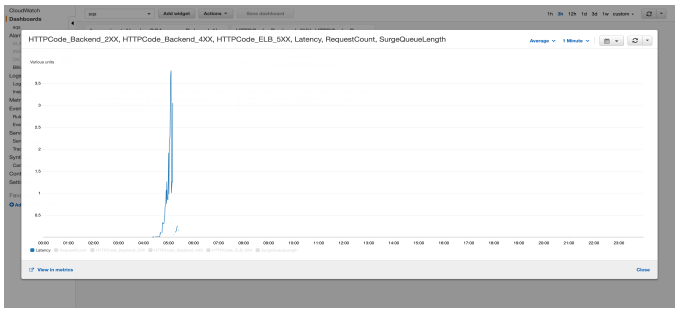

Fig. 8. Latency.

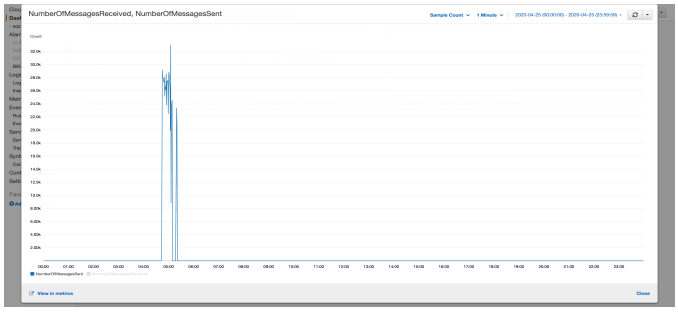

Fig. 9. Total messages sent.

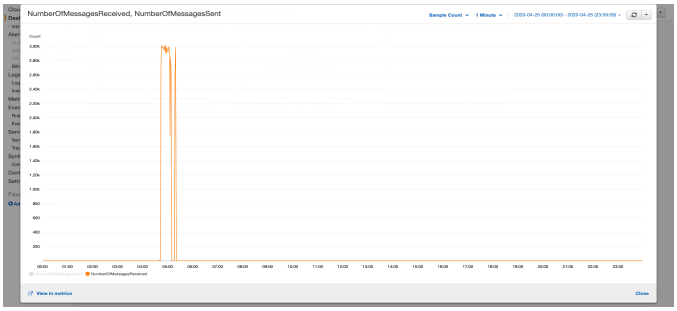

Fig. 10. Total messages received.

Above shown are the results of bench marking the system using SQS. The total number of users are capped at 5000 and the hatch rate is set at 50 users/sec.

b) Kafka

a) Non-optimized

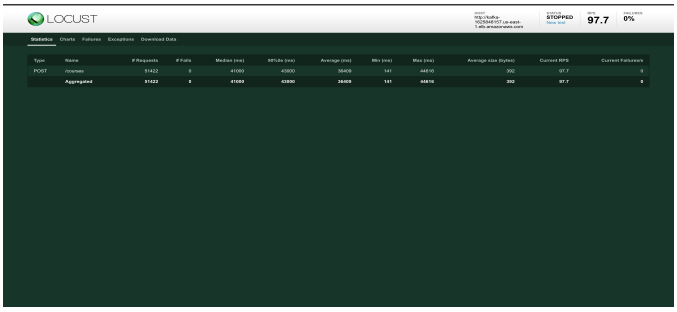

Fig. 11. Locust result.

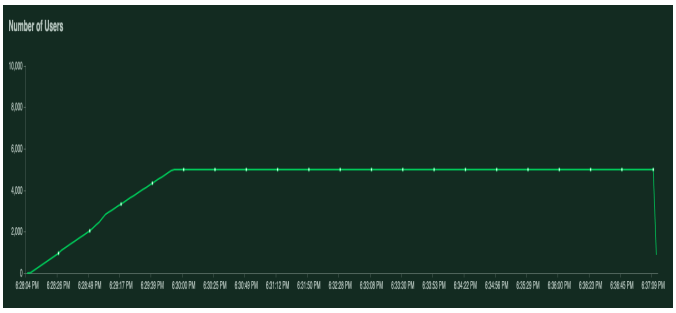

Fig. 12. Number of users.

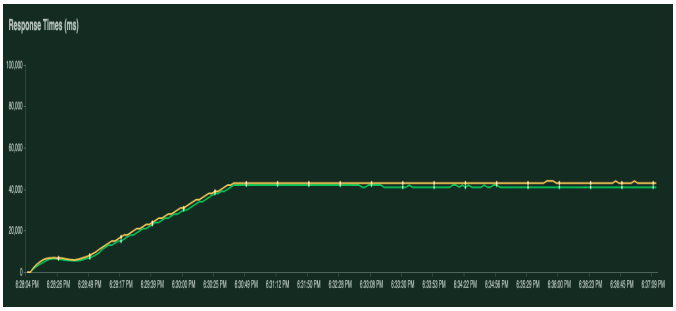

Fig. 13. Response times in ms.

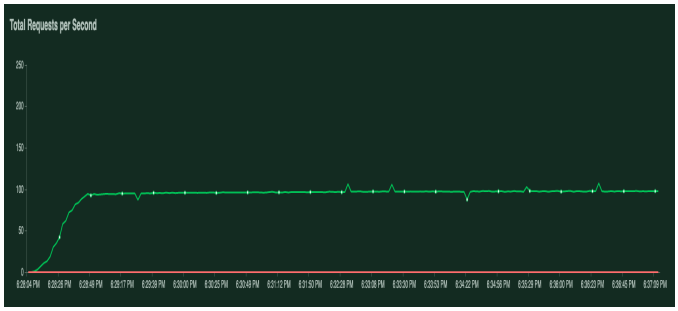

Fig. 14. Total requests per second.

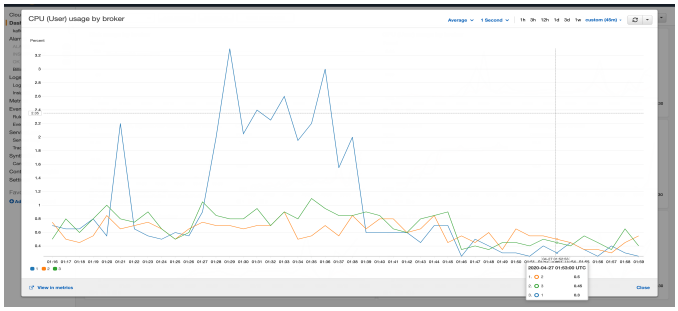

Fig. 15. CPU.

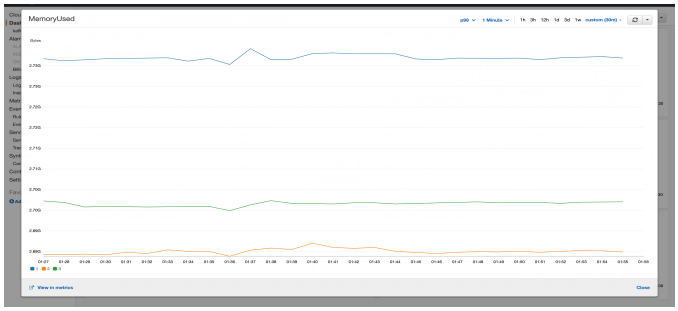

Fig. 16. Memory.

Above shown are the results of bench marking the system using Kafka. The total number of users are capped at 5000 and the hatch rate is set at 50 users/sec. The number of brokers here are 3 and the partition size is also 3. The results on this setup are not efficient. The parallelism and the low database connection limit on the database is the problem resulting in such results.

b) Optimized

The number of brokers are increased from 3 brokers to 6 brokers. The partition size is increased to 5 from 3. Also, the database is moved from us-west to us-east to reduce the latency. The maximum allowed connections are also increased to 1500. The following are the results of the same test on this optimized configuration.

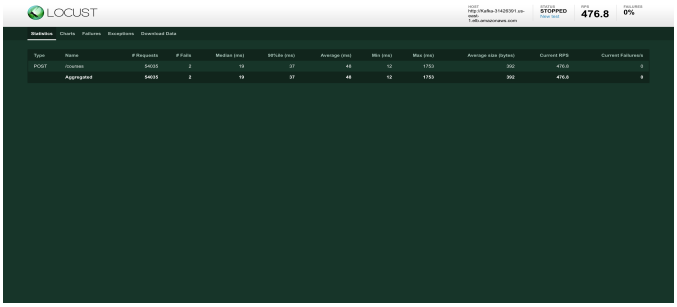

Fig. 17. Locust result.

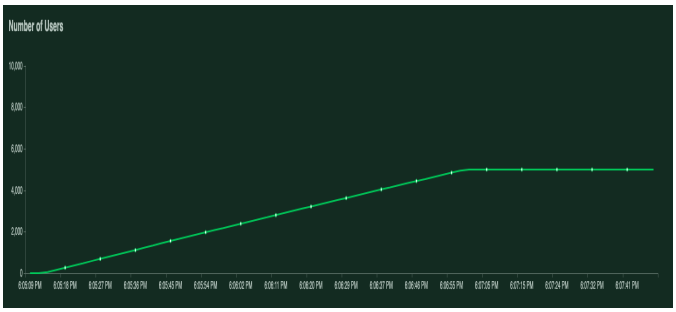

Fig. 18. Number of users.

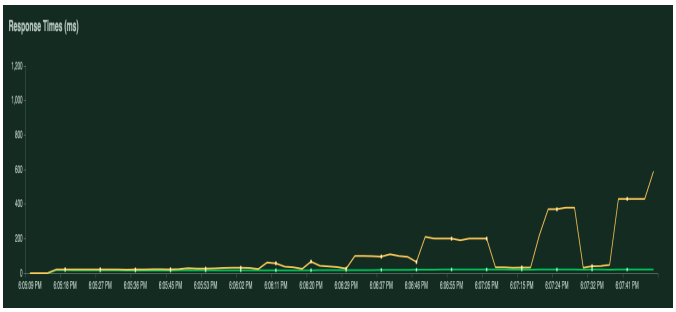

Fig. 19. Response times in ms.

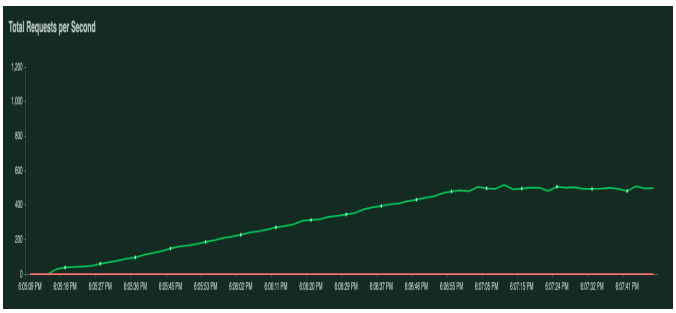

Fig. 20. Total requests per second.

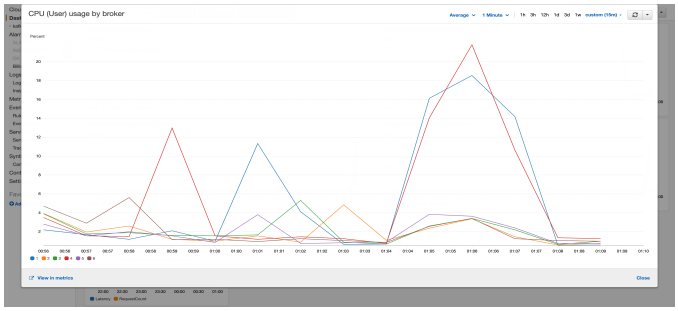

Fig. 21. CPU.

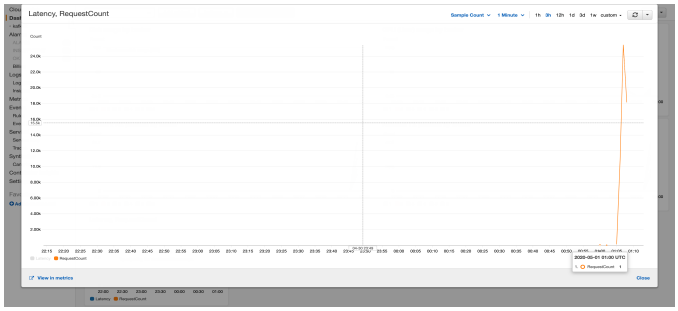

Fig. 22. Request count.

Fig. 23. Bytes in and out.

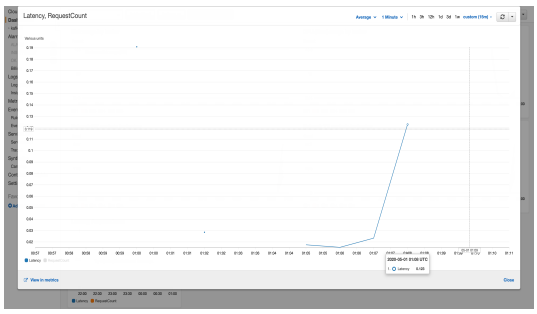

Fig. 24. Latency.

V. Comparison

a) Response time: The response time of Kafka is better than that of SQS. Also, the 90% percentile response time of Kafka is better, which signifies that the worst performing requests of Kafka are faster than SQS.

b) Fails: SQS failed on 29 requests in our tests while Kafka failed only on 2 which suggests that Kafka is more reliable.
c) Throughput: The bytes-in-and-out graph shows that the bytes received and bytes sent out of Kafka increase linearly together which suggest that the throughput is high which is not the case for SQS.
d) Database connections: According to our tests, we found that Kafka was heavier on database connections since the throughput is higher.
e) Ease of implementation: SQS is categorically easier to implement and use. While Kafka needs to be monitored and configured more to get the best out of it.
f) Cost: SQS is priced per request. Hence if the traffic to the queues if very high the cost increases a lot. But, Kafka's pricing depend upon the instances a user uses and the storage.

VI. Conclusion

In conclusion, as seen in the results, it can be stated in this specific setup that Kafka performs better overall if response time, failure rate and throughput is taken into consideration. If the database does not support a high number of connections, SQS is the better choice. In terms of cost, if the traffic is more, Kafka is also more cost efficient as the price is not set per request.

References

[1] Kafka versions, https://kafka.apache.org/22/documentation.html
[2] Amazon SQS Documentation, https://docs.aws.amazon.com/AWSSimpleQueueService/
[3] Amazon Managed Services for Kafka FAQs, https://aws.amazon.com/msk/faqs/
[4] Locust documentation, https://docs.locust.io/en/stable/what-is-locust.html/
[5] AMongoDB Atlast cluster tier, https://docs.atlas.mongodb.com/cluster-tier/
[6] Amazon Web Services EC2 instance types, https://aws.amazon.com/ec2/instance-types/