Comparative Analysis of Message Queuing Systems

Aditya Ravikant Jadhav
Department of Computer Science
North Carolina State University
Email: ajadhav3@ncsu.edu

Radhika Toravi
Department of Computer Science
North Carolina State University
Email: rtoravi@ncsu.edu

Leanne Serrao
Department of Computer Science
North Carolina State University, Raleigh
Email: lserrao@ncsu.edu

***Abstract*—Message queue (MQ) allows IT systems to communicate data amongst themselves. Asynchronous protocols are provided by the queue, allowing senders and receivers to communicate over long distances and at various times. Messages might be requests, responses, or alerts, depending on the needs of the sender. By storing, processing, and removing activities as they are finished, the queue facilitates service-to-service communications. The publisher/subscriber pattern, which is commonly employed in large, message-oriented middleware systems, is used in some message queue applications. IT workers, system administrators, and software developers are the most common users of MQ tools. Message queue software is used by businesses to coordinate remote applications, simplify building diverse apps, boost performance, and automate communication-related operations. MQ solutions include a buffer that allows users on different systems to send messages to be temporarily stored until action is taken. In today's massive data processing environments, message queuing systems with high throughput and low latency are critical. Message queuing systems are being used by a variety of businesses and systems. Because of the differences in design and architecture, these systems can be used in a variety of contexts.**

I. Introduction

For many years, message queuing has been utilized in data processing. Because of the large volume of streaming data and high speed requests to the applications from various users, the production environments face various data issues like data loss, data inconsistency, delay in responses, etc. Message queuing systems with high throughput and low latency are crucial in today's big data processing environments. Nowadays, various companies and systems are built with popular message queuing systems. These systems differ in their design and architecture which leads them to be utilized in different scenarios.

Some of the other benefits of message queuing are: Applications can be designed using small programs that can be shared between many applications. By reusing these building elements, you can quickly create new applications. Changes in the way queue managers work have no effect on applications that use message queuing techniques. The queue manager is responsible for dealing with all aspects of communications and doesn't require use of any communication protocols. Programs that receive messages need not be running at the time that messages are sent to them. The messages are retained on queues.

However, it is difficult for a novice in technology design with no prior knowledge of message queues, to select a suitable system to meet the requirements optimally. With this thought in mind, through this paper, we would like to bring up a fair comparison between different message queuing systems with few fixed metrics to give a better view towards their usage and their efficiency in respective scenarios. For this comparison, we have chosen the following popular message queuing systems based on their architecture difference (Kafka, Amazon SQS and ActiveMQ) and they will be evaluated on the basis of qualitative use case scenarios and quantitative analysis. This paper will also enlighten us with the appropriate use-case fit for the respective system and provide a few use-case scenarios according to the results obtained.

II. Previous Work

[11]Our inspiration is the work done on Fair Comparison of Message Queuing Systems presented by Guo Fu, Yanfeng Zhang and Ge Yu. They evaluated Five popular MQ systems namely Kafka, ActiveMQ, RabbitMQ, RocketMQ and Pulsar on basis of Quantitative analysis as well as Qualitative Analysis. They have implemented these systems in java as java is the most widely used language for such integrations. The Results captured in their work concluded that the above Message Queuing Systems had similar performance metrics results except Kafka which outperforms all other MQ Systems in Throughput as well as Latency.

Our main focus is an extension to this research considering a set of some new MQ technologies and expanding our scope from on-prem systems to cloud-based systems like Amazon SQS. In addition to the evaluated metrics, we will be introducing few more metrics to compare these systems.

III. Apache Kafka

[1]Apache Kafka is an open-source distributed publish/subscribe messaging system that can process trillions of records each day. Kafka was originally designed as a communications queue, but it is now an abstraction of a distributed event log.

A. History

Kafka has quickly developed from a messaging queue to a full-fledged event streaming platform since it was invented and open sourced by LinkedIn in 2011. Confluent Platform, which was founded by the original Apache Kafka developers, provides the most comprehensive Kafka distribution. often described as a distributed event log where all the new records are immutable and appended to the end of the log

Fig. 1. Apache Kafka Architecture

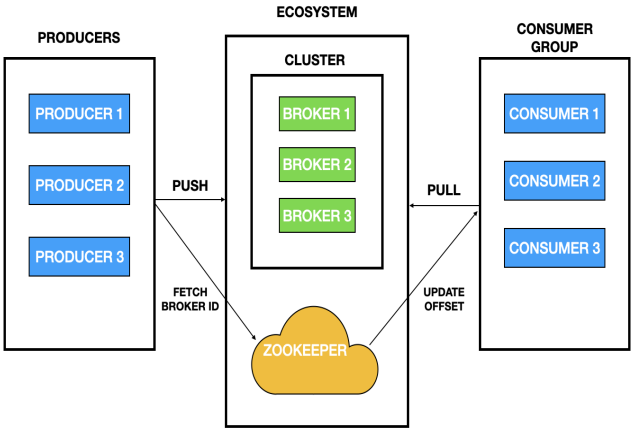
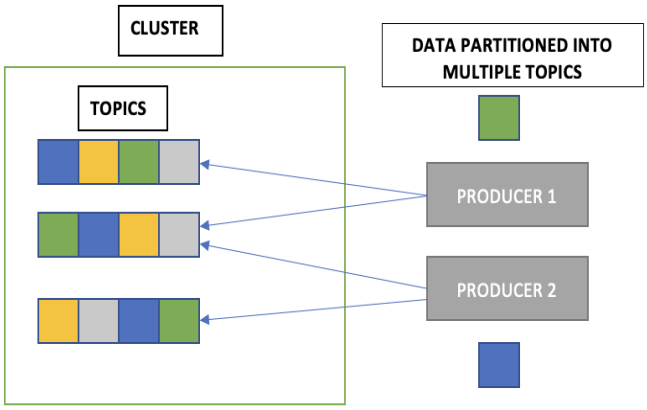


Fig. 2. Partition in Topics



B. Architecture

Apache Kafka follows a Client-Server model where multiple servers and clients communicate via a high-performance TCP network protocol. The architecture consists of a Kafka Cluster which would comprise of multiple server nodes within the same network. (Refer to Fig. 1) The nodes in the Kafka Cluster are called as **Kafka Brokers.** Each Kafka broker has a unique id which represents it. A producer or a consumer client connecting to one broker automatically connects it to the entire Kafka Cluster. There is no limit on the number of brokers allowed in a Kafka Cluster considering cost as one of the factors in scaling up the brokers. Each node in the Kafka Cluster would process a stream of events.

An **Event** in Kafka is nothing but a record which signifies a action taken place. For example, a User opening a website is an event stored as a record in Kafka. When the data is read from or written into Kafka, it is taken place in form of events. Events are organized into topics. **Topic** (Refer to Fig. 2) can be considered as a storage unit in the system and events are nothing but chunks stored in topics. In order to interact with the topics in the kafka cluster, we have two major components- Producers and Consumers. **Producer/Publisher** is essentially an application that is the source of the data stream. The Apache Kafka Producer is used to generate tokens or messages and then publish them to one or more topics in the Kafka cluster. **Consumer/Subscriber** is again a piece of client code which reads the produced records from the topics in the Kafka cluster.

[9]These topics can be partitioned into multiple Kafka brokers. One major component of the entire Kafka Architecture is the **Zookeeper**. It is an open-source system whose primary purpose is to monitor the Kafka Brokers, Kafka Topics and Kafka Messages. It manages kafka to work as a distributed system.

The capacity of Apache Kafka to scale would be severely limited if a topic was forced to live exclusively on a single system. It might handle multiple topics across multiple machines—after all, Kafka is a distributed system—but no single topic could ever grow too large or aspire to handle too many reads and writes. Kafka, fortunately, does not leave us with no options: it allows us to split topics. Partitioning divides a single topic log into numerous logs, each of which can be stored on a separate Kafka cluster node. The job of storing messages, writing new messages, and processing current messages can be distributed over a large number of nodes in the cluster in this fashion. (Refer to Fig 2) Partitions can have copies to increase durability and availability and enable Kafka to failover to a broker with a replica of the partition if the broker with the leader partition fails.

IV. Apache ActiveMQ

Apache ActiveMQ is a Java-based message broker that also includes a full **Java Message Service** (JMS) client. It offers ”Enterprise Features,” which in this case refers to facilitating communication between several clients or servers. Java via JMS 1.1, as well as a number of other ”cross language” clients, are supported. Computer clustering and the ability to use any database as a JMS persistence provider, in addition to virtual memory, cache, and journal persistency, are used to manage communication.

Artemis is the name of another broker under the ActiveMQ umbrella. It is based on the HornetQ code base, which was provided to the Apache ActiveMQ community by the JBoss community in 2015. Artemis is ActiveMQ’s ”next generation” broker, and it will eventually become the next major version.

A. History

The ActiveMQ project began in 2004 as an open source message broker hosted at CodeHaus by its developers from LogicBlaze [5]. In 2007, the code and the ActiveMQ trademark were transferred to the Apache Software Foundation, where the founders continued to work on the codebase with the Apache community at large.

B. Architecture

[4]Client applications use ActiveMQ to send messages between producers and consumers. Producers create messages

Fig. 3. ActiveMQ Classic Architecture

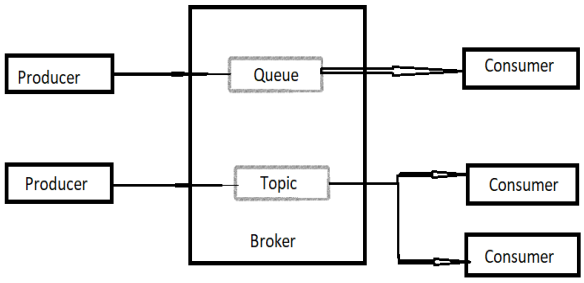
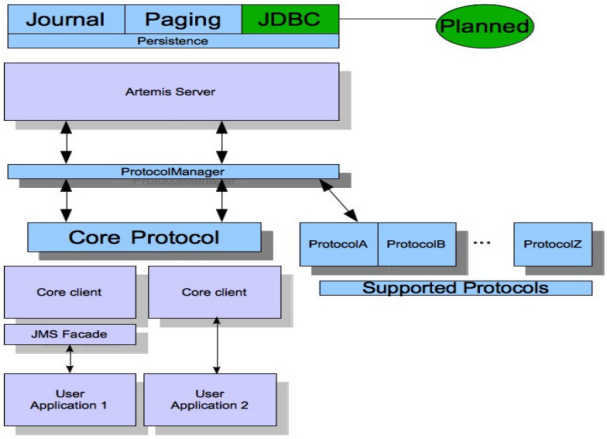


Fig. 4. ActiveMQ Artemis Architecture



and submit them for delivery, while consumers receive and process them. Each message is routed by the **ActiveMQ broker** through a messaging endpoint known as a destination (in ActiveMQ Classic) or an address (in Artemis). **Point-to-point** messaging (in which the broker routes each message to one of the available consumers in a round-robin pattern) and **publish/subscribe** (or "pub/sub") messaging (in which the broker delivers each message to every consumer who is subscribed to the topic (in ActiveMQ Classic) or address (in ActiveMQ) are both supported by both ActiveMQ versions (in ActiveMQ Artemis).

Apache ActiveMQ Classic sends point-to-point messages via **queues** and pub/sub messages via **topics**, as shown in Fig 3. Artemis, on the other hand, supports both types of messaging with queues and imposes the proper behavior using routing types. In the instance of point-to-point messaging, the broker transmits a message to an address specified with the anycast routing type, which is then queued for retrieval by a single consumer. (An anycast address usually has only one queue, but it can have multiple queues if needed, such as to support a cluster of ActiveMQ servers.) In the case of pub/sub messaging, the address contains a queue for each topic subscription, and the broker sends a copy of each message to each subscription queue using the multicast routing type.

The Java Message Service (JMS) API defines a standard for creating, sending, and receiving messages, and ActiveMQ implements the above functionality. The JMS API can be used to send and receive messages in ActiveMQ client applications (producers and consumers) written in Java. Non-JMS clients written in Node.js, Ruby, PHP, Python, and other languages can connect to the ActiveMQ broker using the AMQP, MQTT, and STOMP protocols, which both Classic and Artemis support. The ubiquitous AMQP protocol can be used to connect with multi-platform apps. STOMP via websockets allows to send messages between different web apps. Using MQTT, IoT devices can be managed. Any messaging use-case may be supported with ActiveMQ's power and flexibility.

As ActiveMQ sends messages asynchronously, consumers may not receive messages right away. The task of composing and sending a message is separated from the task of retrieving it by the producer. Producers and consumers are independent (and even oblivious) of each other since ActiveMQ employs a broker as an intermediary. Regardless of whether or not a consumer hears the message, a producer's job is done once it delivers a message to a broker. When a consumer receives a communication from a broker, on the other hand, it does so without knowing who originated the message.

C. Apache ActiveMQ Artemis

[8]The core of Apache ActiveMQ Artemis is a collection of plain old Java objects (POJOs). Each Apache ActiveMQ Artemis server has its own ultra-fast persistent journal, which it utilizes to store messages and other data. Clients of Apache ActiveMQ Artemis communicate with the Apache ActiveMQ Artemis broker, which may be on different physical machines. At the moment, Apache ActiveMQ Artemis ships three client-side messaging API implementations:

1. Core client API- This is a straightforward Java API that is compatible with Artemis' underlying core. Increasing the control over broker objects (e.g direct creation of addresses and queues). The Core API also provides a complete set of messaging capabilities without the complexities of JMS.

2. JMS 2.0 client API- On the client side, the usual JMS API is available. The Jakarta Messaging 2.0 specification is also met by this client.

3. Jakarta Messaging 3.0 client API- This is nearly identical to the JMS 2.0 API. The only difference is that instead of javax, the package names use jakarta. This distinction arose as a result of the transition from Oracle's Java EE to Eclipse's Jakarta EE.

Fig. 4 shows two user applications interacting with an Apache ActiveMQ Artemis server. User Application 1 is using the JMS API, while User Application 2 is using the core client API directly.

V. Amazon SQS

[7]Amazon Simple Queue Service (SQS) is a fully managed

message queuing service that enables you to decouple and scale microservices, distributed systems, and serverless applications. SQS eliminates the complexity and overhead associated with managing and operating message-oriented middleware, and empowers developers to focus on differentiating work. Using SQS, you can send, store, and receive messages between software components at any volume, without losing messages or requiring other services to be available.

A. History

The Amazon Simple Queue Service (Amazon SQS) is a distributed message queuing service that was first released as a test in late 2004 and became commercially available in mid 2006. It allows users to communicate over the Internet by sending messages programmatically via web service applications. SQS is designed to provide a highly scalable hosted message queue that addresses challenges emerging from the common producer–consumer problem, as well as connectivity issues between producer and consumer. Amazon provides SDKs in several programming languages including Java, Ruby, Python, .NET, PHP, Go and JavaScript. A Java Message Service (JMS) 1.1 client for Amazon SQS was released in December 2014.

B. Architecture

As shown in the Fig. 5 [7], there are three main parts in a distributed messaging system:

1. Components of the distributed system- Components of a distributed system usually refer to producer and consumer. Producers are components that send messages to the queue and consumers are components that recieve messages from the queue. There can be any number of producers and consumers.

2. Queue- The queue that holds messages itself is distributed over several amazon SQS servers. The queue redundantly stores the messages across multiple Amazon SQS. There are two types of Amazon SQS queues:

a. Standard queues - They offer Unlimited Throughput as standard queues support a nearly unlimited number of API calls per second, per API action (SendMessage, ReceiveMessage, or DeleteMessage). Standard queues ensure that a message is delivered at least once, but occasionally more than one copy of a message is delivered. In case of standard queues, messages are delivered in an order different from which they were sent.They are generally used to send data between applications when the throughput is important.

b. FIFO queues- They process a message exactly once i.e a message is delivered once and remains available until a consumer processes and deletes it. Duplicates aren't introduced into the queue. The order in which messages are sent and received is strictly maintained. They offer high throughput. FIFO queue also supports message groups. It allows multiple ordered message groups within a single queue. By using batching, FIFO queues support up to 3,000 messages per second, per API method (SendMessageBatch, ReceiveMessage, or DeleteMessageBatch). Without batching, FIFO queues support up to 300 API calls per second, per API method (SendMessage, ReceiveMessage, or DeleteMessage). They are generally used to send data between applications when the order of events is important.

Fig. 5. Amazon SQS Architecture

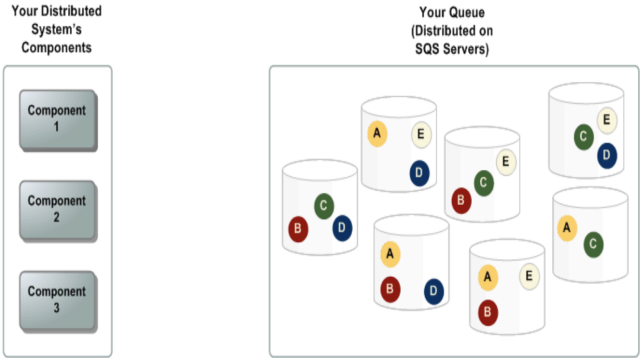
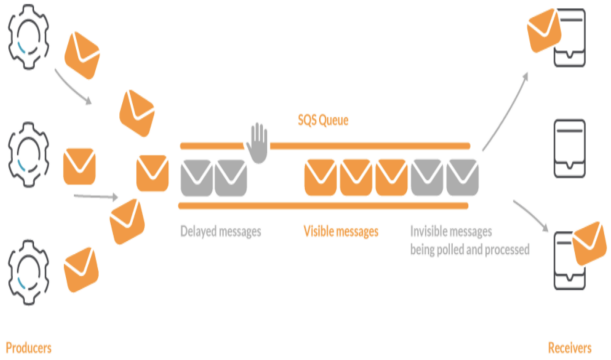


Fig. 6. Amazon SQS Architecture



3. Messages in the queue- Messages can be of any type, and the data contained within is not restricted. Message bodies are limited to 256KB. For larger messages, the user has a few options to get around this limitation. A large message can be split into multiple segments that are sent separately, or the message data can be stored using Amazon Simple Storage Service (Amazon S3) or Amazon DynamoDB with just a pointer to the data transmitted in the SQS message.

C. Working

Fig. 6 [9] represents the workflow of a message in a SQS queue.

The message is created by a producer service and sent to the SQS queue.The message is visible to all prospective receivers in the queue. This stage does not have to be completed right away. If there is a 'delay' specified in the message, it will remain in a delayed state in the queue and will not be available to receivers until the delay expires. After a configurable delay, the message is marked again as visible

so other receivers can get the message and process it. One of the possible receivers does a polling of the SQS queue's messages i.e the visible messages in the queue are retrieved and switched to an invisible state, but they are not deleted. This prevents other receivers from receiving those messages if they start polling again. When the receiver ends to process the message, explicitly removes it from the queue. Amazon SQS alsp provides a another queue called as a dead-letter queue.A dead-letter queue is where the messages end after being polled a number of times.

VI. Methodology

The above three message queuing systems have many overlapping advantages and disadvantages to discuss. This made it difficult to identify the differences between them. Hence, we based the comparison between them through the below performance metrics,

1. Throughput : Measure of how many units of information(records/messages) a system can process in a given amount of time.

2. Latency : Time interval between the start of a request at the client end to deliver the result back to the client by the server.

A. Experiment Setup

We replicated the Message Queue Servers locally in order to run the experiment. For Apache Kafka, we set up 3 brokers with one topic. Each topic was split into 3 partitions, each replicated in a different broker. For ActiveMQ, we implemented ActiveMQ Classic in our local system which would work as a server with a topic created. We created a SQS topic/queue in Amazon SQS service provided by AWS, setting some default parameters provided by AWS. We even set some basic security parameters in order to access SQS.

We created one Publisher to send messages to the topic and Subscriber who will receive messages from the topic for all the above systems.

B. Tools Used

There are various existing tools used to measure performance metrics of software systems. In our project, we have used a few amongst them to compare the the three Message Queuing Systems, mentioned as follows.

Apache JMeter can measure performance and load test static and dynamic web applications. It is built on Java and is completely portable. It even supports multi-threading i.e, allows concurrent sampling by many threads and simultaneous sampling of different functions by separate thread groups. It can be used to simulate a heavy load on a server, group of servers, network or object to test its strength or to analyze overall performance under different load types.

AWS Meter is a JMeter plugin to execute tests over AWS services like Kinesis, SQS and SNS. This plugin has a set of Java sampler per AWS service that are using AWS SDK to integrate with AWS and communicate with each service. When you use AWS services you need set up those to process normal and peak of load. To make sure the service configuration is right you can execute load testing with awsmeter to see the behaviour of the system under variant or specific load.

Performance Scripts are bash Scripts which we created to measure Kafka Performance measures. These include the Kafka Configurations and some parameters in order to effectively capture the metrics.

C. Implementation

For **Apache Kafka**, we configured a local kafka cluster. Initially, we started the Zookeeper Server to manage the Cluster. Then, we configured a broker with their own server properties. The broker will be managed by the Zookeeper Server. This was followed by creating a Kafka Topic.

The broker was started and the topic was configured to only one broker as that will be considered as the primary node and zookeeper would convey the topic details to other nodes in the cluster. We created two bash scripts which are Producer and Consumer scripts. They were created to publish and subscribe 10K messages at once to the topic respectively. These scripts also captures performance metrics like Throughput and Latency which were needed for our analysis.

Apart from the Comparison, we also did an in-depth analysis of Apache Kafka to understand the benefits of the cluster architecture. For this, we configured two more brokers with different server properties i.e, different ports to access them. The topic created was configured with some more parameters like the partition count was set to 3, distributed into the configured brokers each. And the replication factor was set to 2 i.e, the record pushed into the topic will be replicated into 2 brokers at once. Later we ran the same performance scripts in order to measure the Apache Kafka Throughput and Latency with different count of brokers active. This time, the message count configured was 100000. We tested Apache Kafka with 1, 2 and 3 active brokers and captured the metrics at each stage.

Amazon SQS AWS Meter is a JMeter plugin to execute tests over AWS services like Kinesis, SQS SNS. This plugin has a set of Java sampler per AWS service that are using AWS SDK to integrate with AWS and communicate with each service. We first created a SQS queue on AWS console with desired configuration like selecting queue type, setting visibility timeout i.e time that AWS will wait to receive delete message action before enabling the message to process again, setting up delivery delay i.e the amount of time the message will be hidden for consumers, AWS control the time and deliver the message to consumer when the time is up. We then installed awsmeter in JMeter to connect and send messages to SQS queues. We used awsmeter to produce and publish messages in Standard queue and used AWS Management Console to receive the messages. We then used JMeter Test Plan to execute our tests after configuring different parameters like the queue name, message body and fields to connect our AWS account.

For **ActiveMQ**, we first installed the server locally. We then implemented a simple Java code to test the sending and receiving of messages in the queue as well as topic. After starting the server, the working of ActiveMQ can be monitored at: http://localhost:8161/admin/queues.jsp. This is an interactive web console of ActiveMQ where server details can be viewed and monitored. Other details like number of queues, number of messages sent, active producers, consumers can also be seen here. We then used JMeter to measure the performance measures like throughput, latency. For this we first created a Test Plan. In the test plan, we created separate thread groups for producer and consumer and in the thread groups we created a sampler for producer and consumer respectively. In the thread group, we can set the number of threads, loop count etc. The default server url is: tcp://localhost:61616. In the sampler, we set the number of consumer and producers respectively. We then added a Listener to monitor the progress of the messages sent and to proctor the performance metrics. We then executed the test plan to send upto 10k messages to make it easier for comparison with the other message queuing systems.

VII. Results

Our Results is based on two different evaluations. First, we compared all three MQ systems i.e, Apache Kafka, ActiveMQ and Amazon SQS with around 10,000 messages with 1KB message size. As per our results shown in Fig. 5, Apache Kafka has higher throughput and lower latency then ActiveMQ and Amazon SQS. Apache Kafka throughput comes around 15K msgs/secs for around 10K messages whereas for Amazon SQS, it comes to 5.5k msgs/secs and for AciveMQ it is 1k msgs/secs. For latency, Kafka had the lowest of around 500secs, followed by Amazon SQS with 700secs and ActiveMQ had the largest with around 1000secs. Thus from our analysis, we can see that in terms of both the performance metrics throughput and latency, Kafka performs the best followed by Amazon SQS and then ActiveMQ.

The second evaluation was in-depth performance monitoring of Apache Kafka as a cluster. As shown in Fig. 6, we captured Throughput at different active brokers count i.e, 1, 2 and 3. We can observe that the throughput slightly increases with the increase in the number of active brokers. Similarly, as shown in Fig. 7, we see that the Latency reduces as we increase the number of brokers. One vital observation was noticed that the Producer and Consumer Process is not disturbed when one of the three brokers goes down, but there is a slight increase in the latency.

VIII. Conclusion

In this paper, we examined the three popular message queuing systems Kafka, ActiveMQ, and Amazon SQS on a variety of factors, including performance and functionality. The above Results section summarizes the results of the comparison analysis of these systems. We create a test framework with a unified tool to imitate producer and consumer in order to assess throughput/latency performance equally. In comparison to the other two systems, our tests reveal that Kafka offers a higher

Fig. 7. Results of comparison

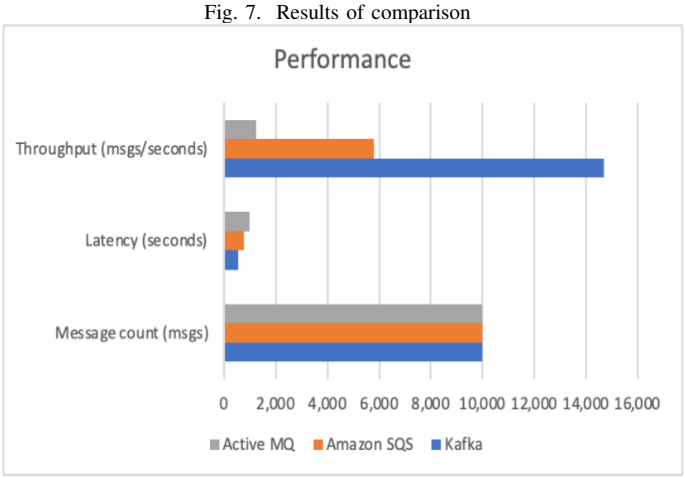


Fig. 8. Kafka Throughput Results

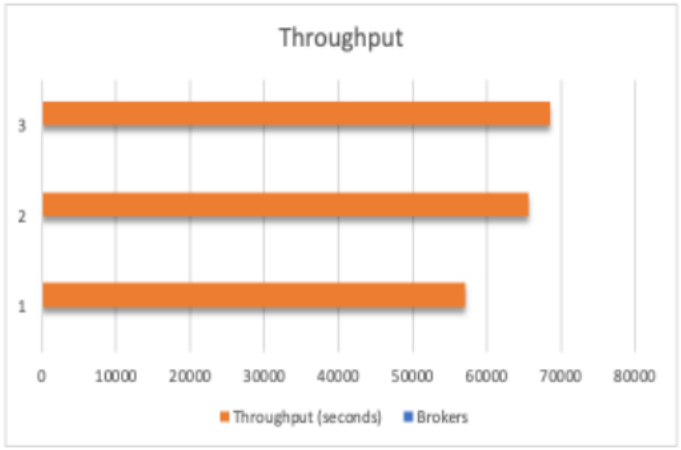
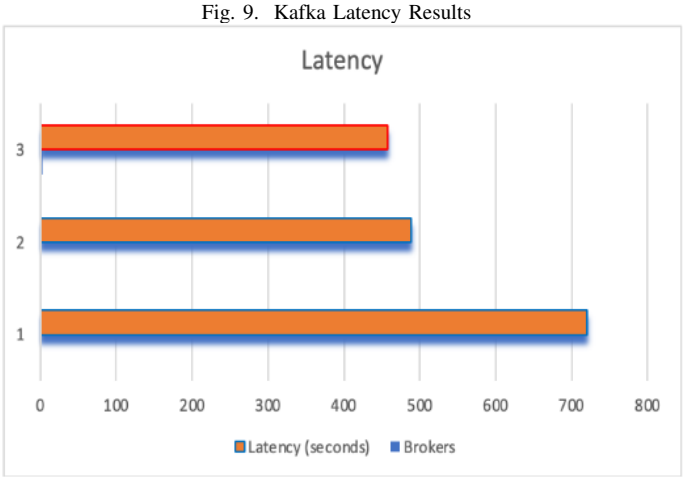


Fig. 9. Kafka Latency Results



throughput and shorter latency. We can also deduce that as the number of brokers in the cluster grows, Kafka's performance improves and it becomes fault-tolerant. Kafka is the ideal solution if the message queue is utilized to process online commerce that demands low latency and high throughput. However, when it comes to cluster administration, Apache Kafka requires manual intervention, whereas AmazonSQS is maintained by AWS which reduces man-work. As a result, AmazonSQS is a good option if you don't want to deal with a lot of manual work or manage a lot of resources. If you need to process a large number of messages, such as log processing, big data analysis, or stream processing, Kafka is the way to go.

REFERENCES

[1] Data-Flair, https://data-flair.training/blogs/kafka-architecture/.
[2] Apache Kafka, https://kafka.apache.org.
[3] Apache ActiveMQ, https://activemq.apache.org/
[4] DataDog ActiveMQ architecture and key metrics," https://www.datadoghq.com/blog/activemq-architecture-and-metrics/
[5] Wikiwand Apache ActiveMQ, https://www.wikiwand.com/en/Apache_ActiveMQ
[6] M. Ishii and S. Kim, "Development of One-group and Two-group Interfacial Area Transport Equation," http://www.ne.anl.gov/pdfs/NEAMS/NEAMSQuarterlyReport10-2014.pdf (2014).
[7] AWS SQS, https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-basic-architecture.html
[8] Apache ActiveMQ Artemis Documentation, https://activemq.apache.org/components/artemis/documentation/latest/architecture.html
[9] AmazonSQS-Working, https://sysdig.com/blog/monitor-amazon-sqs-prometheus/
[10] Understanding Kafka Topic Partitions, https://medium.com/event-driven-utopia/understanding-kafka-topic-partitions-ae40f80552e8
[11] A Fair Comparison of Message Queuing Systems, https://doi.org/10.1109/ACCESS.2020.3046503