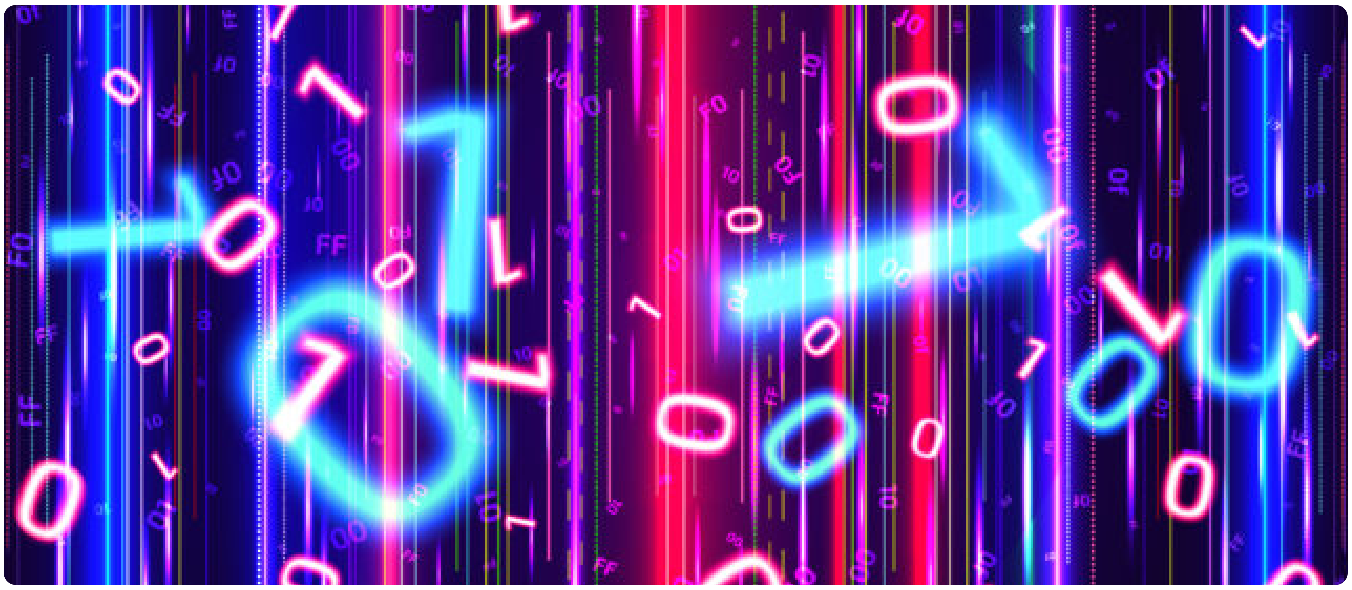
GIGAOM

Benchmarking Enterprise Streaming Data and Message Queuing Platforms



By William McKnight, and Jake Dolezal

Sponsored by streamlio

GIGAOM

Benchmarking Enterprise Streaming Data and Message Queuing Platforms

03/02/2018

Table of Contents

1 Summary

Historical paradigms around data are being shifted due to exponential data growth, hybrid cloud architectures, real-time data needs, massive computing at scale, globalization, 24×7 uptime, and the Internet of Things. The list could go on. Organizations are either seizing opportunities, scrambling to keep up, or coping with the changes in their industry verticals due to technological shifts. Many want or need to adopt these newer technologies, but sometimes struggle with how to evaluate or select them based on their current and future needs.

The needs and uses of data have evolved to a new level. In today's climate, businesses must be able to ingest, analyze, and react to data immediately. With artificial intelligence and machine learning progressing by leaps and bounds, we see increasing numbers of technologies emerging as data-driven, autonomous, decision-making applications—where data is produced, consumed, analyzed, and reacted to (or not) in real-time. In this way, the technology becomes sentient of what's going on inside and around it—making pragmatic, tactical decisions on its own. We see this being played out in the highly publicized development of self-driving cars, but beyond transportation, we see it in telephony, health, security and law enforcement, finance, manufacturing, and in most sectors of every industry.

Prior to this evolution, the analytical data was derived by humans long after the event that produced or created the data had passed. We labored under the assumption that by analyzing the past, we could set the course for the future—in our businesses, in our systems, in our world. Now, however, technology has emerged that allows us to capture and analyze what is going on right now. Much data value is time-sensitive. If the essence of the data is not captured within a certain window of time, its value is lost and the decision or action that needs to take place as a result never occurs.

This category of data is known by several names: streaming, messaging, live feeds, real-time, event-driven, and so on. This type of data needs special attention, because delayed processing can and will negatively affect its value—a sudden price change, a critical threshold met, an anomaly detected, a sensor reading changing rapidly, an outlier in a log file—all can be of immense value to a decision maker, but only if he or she is alerted in time to affect the outcome.

Much has already been said and written about artificial intelligence and autonomous decision making. The focus of this paper is the gap between the production and consumption of real-time data and how these autonomous, sentient systems can be fed at scale and with reliable performance. There are a number of big data technologies designed to handle large volumes of time-sensitive, streaming data.

We will introduce and demonstrate a method for an organization to assess and benchmark—for their own current and future uses and workloads—the technologies currently available. We will begin by reviewing the landscape of streaming data and message queueing technology. They are alike in purpose—process massive amounts of streaming data generated from social media, logging systems, clickstreams, Internet-of-Things devices, and so forth. However, they also have a few distinctions, strengths, and weaknesses.

2 The Current Landscape for Streaming Data and Message Queuing

Recent data trends are driving a dramatic change in the old-world thinking and batch Extract-Transform-Load (ETL) architecture. These include data platforms operating at an enterprise-wide scale today, a high variety of data sources, and ubiquitous real-time streaming data. Enterprise Application Integration (EAI) and Enterprise Service Buses (ESB) were early attempts at real-time data integration, but the technologies used were difficult to scale. This left data integration in the old world with a dilemma—either use real-time loading but not be scalable, or be scalable but use batch loading.

While the uses and applications are numerous, one needs to examine the information need that drives the development of streaming data. The need to handle, analyze, and react to data in real-time is clear. Thus, early pioneering organizations saw the need for tools, strategies, and architectures that were powerful, resilient, and efficient enough to consume and analyze a roaring stream of data. It was clear that a central, monolithic architecture of a central database where all this data could be stored in a single database was not the best route. Multiplicity was required. Data, produced from numerous sources, was a torrent of flowing information, needing to be timestamped, dispatched, and even duplicated (to protect against data loss). A postman was needed to distribute data from message senders to receivers at the right place at the right time. This was the only way to build a robust, sustainable big data architecture.

Enter message-oriented middleware to manage and distribute streaming data. A message-oriented middleware (which can be thought of as a form of a data bus) is an architectural component that deals with messages. Messages can be any kind of data wrapped in a neat package with a very simple header as a bow on top. Messages are sent by "producers"—systems, sensors, or devices that generate the messages—toward a "broker." A broker does not process the messages, but instead routes them into queues according to the information enclosed in the message header or its own routing process. Then "consumers" retrieve the messages from the queues to which they subscribe (although sometimes messages are pushed to consumers rather than pulled). The consumers open the messages and perform some kind of action on them. Often, messages are then discarded once processed, and consumers carry on with the next set of messages. However, some use cases require message storage and retention of history.

Evolution in this space extends to the variety of use cases with new ones emerging all the time. These uses span several industries and sectors. Some uses are very exciting. For example, stream processing is beginning to be used in medical and healthcare settings with both systems and health professionals analyzing privacy-protected streams of medical device data to detect early signs of disease, identify correlations among multiple conditions, and assess efficacy of treatments. Also, streaming and message queuing supports applications for equipment problem detection and predictive maintenance, and can provide data for machine learning algorithms to prevent the occurrence of outages and breakdowns. These are just a few of current and emerging use cases.

While we use the term "message-oriented middleware," we feel like the term itself is outdated, because it incorrectly frames the technology as a gap-fix. Instead, we prefer streaming and message queuing technology, because it better encapsulates how the technology has evolved.

The Familiar Players

There are a number of popular technologies in the streaming data and message queuing technology space.

Kafka

One of the better known streaming data processors is the **Apache Kafka** project. Kafka was first created by LinkedIn and was open-sourced and graduated from the Apache Incubator in late 2012. Kafka is noteworthy because of its use by big names in the industry. Data driven technology companies like LinkedIn, Netflix, PayPal, Spotify, and Uber all have or are currently using Kafka. In short, Kafka is a distributed publish-subscribe messaging system that maintains feeds of messages in groups, known as topics. Publishers write data to topics and subscribers read from topics. Kafka is a distributed system; thus, topics are partitioned and replicated across multiple nodes in the cluster.

Within Kafka, messages are simple byte-long arrays that can store objects in virtually any format—most often strings or JSON. Kafka makes it possible to attach a key to each message, so that all messages with the same key will arrive together within the same partition or be delivered to the same subscriber.

Kafka treats each topic partition as a log file where the messages are ordered by a unique offset. To be efficient and scalable, Kafka relies on subscribers to track their location within each log so that it spends its processing resources supporting a large number of users and retaining huge amounts of data with little overhead.

Kafka does have enterprise-supported deployments with Confluent. Cloudera and Hortonworks offer support as part of their larger Hadoop ecosystems.

RabbitMQ

RabbitMQ is an open-source message broker platform originally created in 2007 and now managed by Pivotal Software. RabbitMQ uses an exchange to receive messages from brokers and pushes them to the registered consumers. The broker pushes messages—which are queued in random order—toward the consumers. Brokers are persistently connected to consumer, and they know which ones are subscribed to which queues. Consumers cannot fetch specific messages but can receive them unordered. They are completely unaware of the queue state. RabbitMQ is not "disk-oriented," i.e. most of the messaging operations are performed in memory. Messages are paged out to disk only if there is no more memory available, or if they are explicitly told to be stored.

A difference between RabbitMQ and other technologies like Kafka is that messages, queues, and exchanges do not persist unless otherwise instructed. This is important because if a broker is restarted or fails, the messages are lost. However, RabbitMQ offers settings to make both queues and messages durable. Moreover, non-critical messages can be tagged by the producer to not be sent to a durable queue.

In addition to in-memory storage and optional message durability, RabbitMQ can also require acknowledgements of message delivery from consumers. The acknowledgement can also inform the broker if a message was not able to be processed so the broker can requeue it. Also, RabbitMQ allows producers' and consumers' code to declare new queues and exchanges. Finally, RabbitMQ has several replication and load balancing alternatives.

ActiveMQ Artemis

Apache ActiveMQ Artermis[1] is another open-source message-oriented broker-client written in Java utilizing Java Message Service (JMS). It was originally created by LogicBlaze in 2004 but was later donated to the Apache Software Foundation in 2007. ActiveMQ touts high availability through a shared backend file-system replication and database row-level locks, and coordination via ZooKeeper. It can be scaled horizontally by creating what is called a mesh or network of brokers. Within the mesh, the queue location is not known to the producer or consumer, but the broker does know location and routs requests to the right nodes. ActiveMQ is used in enterprise service bus implementations and is often paired with another technology like Apache Camel for service-oriented architecture infrastructure. One downside is a broker keeps messages for the consumers. However, one big drawback to ActiveMQ Artemis is the fact that it does not have an enterprise-grade supported platform on the market. There are a number of partners that offer consulting and training services.

Amazon Kinesis

Amazon Web Services offers **Amazon Kinesis** as streaming data platform. Kafka and Kinesis share much of the same functionality. The main difference is while Kafka is very fast and free, it requires the developer to turn it into an enterprise-class solution via installation, management, and configuration. Amazon rolled out Kinesis as a managed, pre-configured, nearly out-of-the-box ready tool with the speed and scale of Kafka, but without the administrative overhead. Another difference is nomenclature. Due to the similarity of Kinesis to Kafka, and the proprietary, black-box nature of the technology, it was not considered for this report.

Google Cloud Pub/Sub

Also, strictly in the cloud, **Google Cloud Pub/Sub** is another scalable message queuing platform of interest. Like the others, Pub/Sub provides many-to-many, asynchronous messaging and decouples senders and receivers; it allows for secure and highly available communication between independently written applications. Google Cloud Pub/Sub has low-latency, durable messaging for systems already on the Google Cloud Platform, but also allows developers to integrate systems hosted externally. Like Amazon Kinesis, those enterprises with a significant Google Cloud presence may consider this to expand their footprint.

Other players, such as, Tibco, IBM, and Microsoft, also have solutions worth exploring for those who already are invested in data infrastructures with these particular vendors.

The Need for a Unified, Enterprise Grade Solution

As one can see in the evolution of the current landscape of streaming and message queuing technology, a gap emerged. Either a platform is more streaming data oriented (like Kafka) or more message queuing oriented (like RabbitMQ). Deciding on one or the other presents tradeoffs for an enterprise. For example, if one chooses to go more into streaming data with a solution built on Kafka, they will compromise on giving up capabilities that are more message queuing oriented, such as consumer and producer queue definition, conditional message routing, batch fetch and delivery, broker push, and message rejection and resending.

On the other hand, if one chooses to build their solution on a more message queue-oriented technology, they will lose capabilities, such as ordered storage and delivery, message persistence and durability options, queue data compression, publisher/subscriber methods, and so forth. It makes for a tough decision because in an enterprise-wide information ecosystem, some uses are more streaming-oriented and others are more queuing-oriented.

Despite the presence and expansion of the aforementioned technologies, what has been missing is an enterprise-grade solution ready for deployment that covers both streaming and queuing use cases extremely well—without significant effort by an organization to string together a slew of disparate technologies or write complex custom code to achieve the desired functionality and scale their business and applications demand.

Also missing is a rigorous, scalable, adaptable, easily deployable, and repeatable method for evaluating these platforms across a variety of scale and use case requirements—an issue we will address later on in this paper.

Apache Pulsar and Streamlio – Emergence of Unified Queuing and Streaming

Apache Pulsar

The relatively new kid on the block in message queuing and streaming big data technology is Apache Pulsar. Originally developed at Yahoo, it began its incubation at Apache in late 2016. Pulsar had been in production at Yahoo for 3 years prior—utilized in popular services and applications like Yahoo! Mail, Finance, Sports, Flickr, Gemini Ads, and Sherpa.

Pulsar follows the publisher-subscriber model (pub-sub), and has the same producers, topics, and consumers as some of the aforementioned technologies. Pulsar uses built-in multi-datacenter replication. Pulsar was architected for multi-tenancy and uses concepts of properties and namespaces—such that there is a hierarchy of a Pulsar cluster containing multiple properties, which contain different namespaces, which contain any number of topics. A property could represent all the messaging for a particular team, application, product vertical, et cetera. Namespaces is the administrative unit where security, replication, configurations, and subscriptions are managed. At the topic level, messages are partitioned and managed by brokers using a user-defined routing policy—such as single, round robin, hash, or custom—thus granting further transparency and control in the process.

Additionally, Pulsar has a Kafka API compatibility interface—making porting existing Kafka applications easier.

A key distinction of Pulsar over the others in this space is how it handles message durability. Pulsar ensures that message data is never lost under any circumstances. Pulsar achieves this with Apache BookKeeper to provide low-latency persistent storage. When a Pulsar broker receives messages, it sends the message data to several BookKeeper nodes, which push the data into a write-ahead log and into memory even before an acknowledgement is sent. In the event of a hardware failure, power outage, network problems, or whatever arises, Pulsar messages are safe-kept in permanent storage.

Streamlio

Pulsar is available in an enterprise-ready deployment with Streamlio. Streamlio builds on the Pulsar base to leverage the strengths and distinctions of the platform. One particular distinction emphasized by Streamlio is the emphasis of the unified messaging model—where the strengths of both streaming and message queuing technologies are brought to bear. The unified messaging model has three components:

- Consumption – how are messages dispatched and consumed?
- Acknowledgement – how are messages acknowledged?
- Retention – how long are messages retained, what triggers their removal, and how are they removed?

Streamlio has extensively published information online about this model, so we will not attempt to explain it in detail here. However, an example or two will serve the reader well.

First, Streamlio uses Pulsar's three modes of subscription: exclusive, failover, and shared. Exclusive subscription is only one consumer at a time in a single subscription digesting a topic partition. This method is a perfect application for streaming. Failover subscriptions have multiple consumers, but one is elected the master. This is also a great way to leverage streaming. Third is a shard subscription where you can add as many consumers as you like without adding addition partitions. This method is an excellent use for queuing.

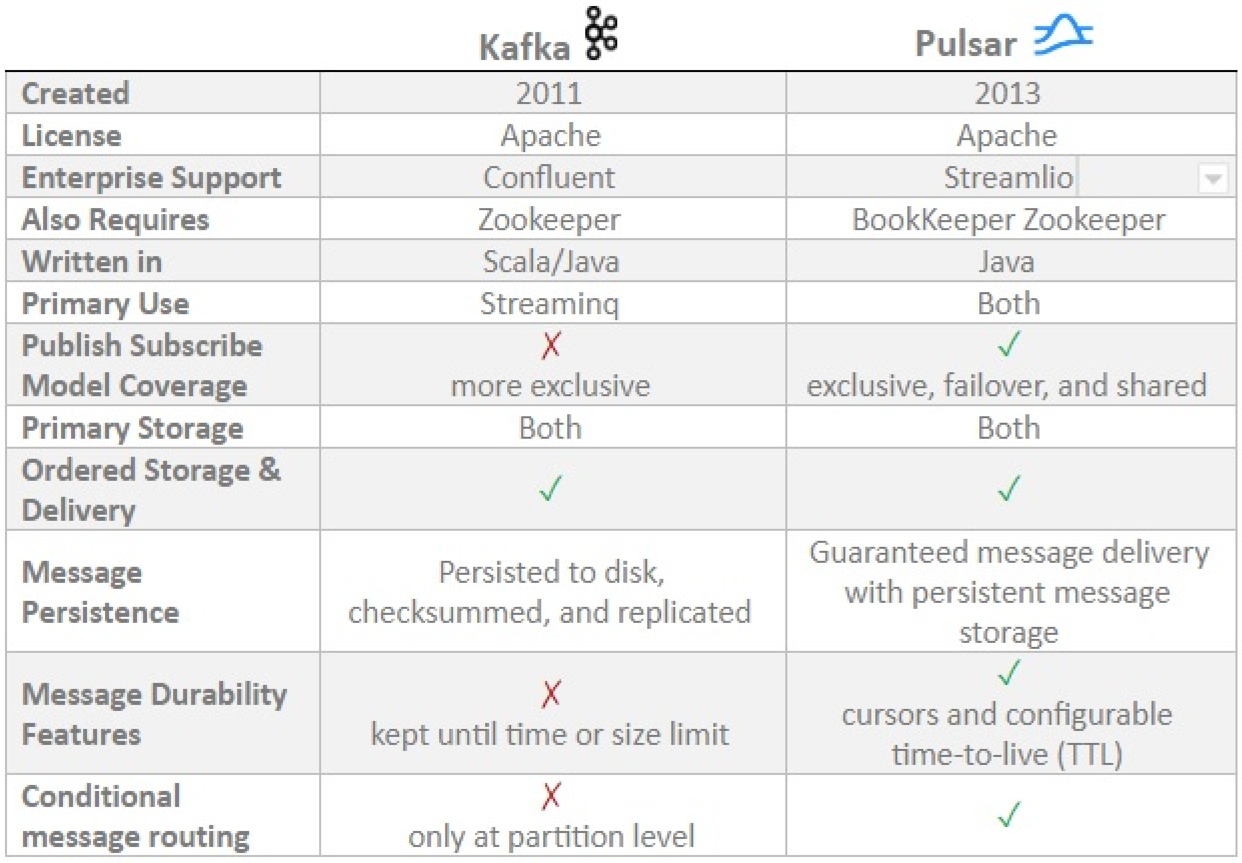
Another example is the use of selective acknowledgements and cursors to manage which messages get sent, resent, and where a consumer left off receiving messages in a stream or queue. It may be very important that some consumption applications never get the same message resent to them. Pulsar can store a committing offset, or set a cursor, on the acknowledged messages.

A key distinction is that Pulsar supports both persistent and non-persistent states. In persistent topics, all messages are durably persisted on disk (multiple disks in a broker cluster). Pulsar manages the cursor to ensure messages are not removed until all subscribers have acknowledged receipt. It also has a configurable time-to-live (TTL) feature than can be set to handle messages that have not been consumed. Again, Pulsar leverages the strength of a robust message queuing system for these needs in shared consumption patterns.

A unified platform gives enterprises the best of both the streaming and message queuing worlds.

Summary of Platform Distinctions, Strengths, and Weaknesses

In summary, the following table provides a side-by-side comparison of the two players in message queuing and streaming data platforms tested in this benchmark.



	Kafka	Pulsar
Created	2011	2013
License	Apache	Apache
Enterprise Support	Confluent	Streamlio
Also Requires	Zookeeper	BookKeeper Zookeeper
Written in	Scala/Java	Java
Primary Use	Streaming	Both
Publish Subscribe Model Coverage	✗ more exclusive	✓ exclusive, failover, and shared
Primary Storage	Both	Both
Ordered Storage & Delivery	✓	✓
Message Persistence	Persisted to disk, checksummed, and replicated	Guaranteed message delivery with persistent message storage
Message Durability Features	✗ kept until time or size limit	✓ cursors and configurable time-to-live (TTL)
Conditional message routing	✗ only at partition level	✓

[1] Note that ActiveMQ was rewritten into a separate product called Apache ActiveMQ Artemis to take advantage of the HornetQ code base from Red Hat and brought the broker's JMS implementation up to the 2.0 specification.

GIGAOM

3 Evaluating Potential Solutions

To this point, we have discussed the functionality distinctions of each streaming data and message-oriented middleware. Overlaying their respective strengths and weaknesses on your own use cases is certainly an important step in short-listing the platforms capable of delivering on the use cases within your own enterprise.

It does not stop there, however. There is still the issue of performance. If a platform cannot perform at the scale your enterprise needs, it really does not matter if it meets all your other checkbox criteria. Beyond the requirement baseline, higher performance is not always the number one selection criterion. It may rank with varying importance in juxtaposition with the other features and capabilities mentioned in the preceding section.

The OpenMessaging Benchmark

In order to evaluate and measure streaming and message queuing technologies, a Linux Foundation Project called OpenMessaging has developed an open standard for evaluating distributed messaging, streaming, and queuing. OpenMessaging is vendor neutral and language independent—it is a multi-organization effort with contributions from Alibaba, Yahoo, Streamlio, DiDi, and the Linux Foundation. The project's intent is to provide industry guidelines across domains such as finance, e-commerce, big data, data science, and the Internet of Things (IoT). The OpenMessaging project encompasses standards-based connectivity, APIs, language support, and benchmarking.

In late 2017 and early 2018, OpenMessaging developed a performance benchmark to evaluate several of the streaming and message queuing platforms discussed in the paper. As of the time of this writing, there are benchmark deployment modules[2] publically available on GitHub[3] for Pulsar and Kafka.

The benchmark arose from the need to go beyond features and capabilities assessments for message-oriented middleware platform selections by providing a method to measure the performance of these platforms under various workloads. The value of the benchmark is that it allows organizations to apply the benchmark to their own uses and pick from the available workloads or design their own.

This is an open benchmark—it is intended to evolve over time (more platforms, workloads, software, etc.) This is a different approach of benchmarking from other uses in the past, where vendors would pit their solutions against their competitors, apply the same workloads, measure the execution time and other metrics, and declare a winner. In the OpenMessaging benchmark, ultimately the intent is to make the organization the winner, by steering them to workloads that are meaningful to their use cases and requirements. Contributions to the benchmark are welcomed.

---

[2] At the time of this writing, benchmark modules for **RabbitMQ** and **ActiveMQ Artemis** were in development but not ready for execution within the OpenMessaging benchmark framework.

[3] The OpenMessaging benchmark is available at https://github.com/openmessaging/openmessaging-benchmark

Goals

The goal of the OpenMessaging Benchmark Framework is to provide benchmarking suites that have the following characteristics:

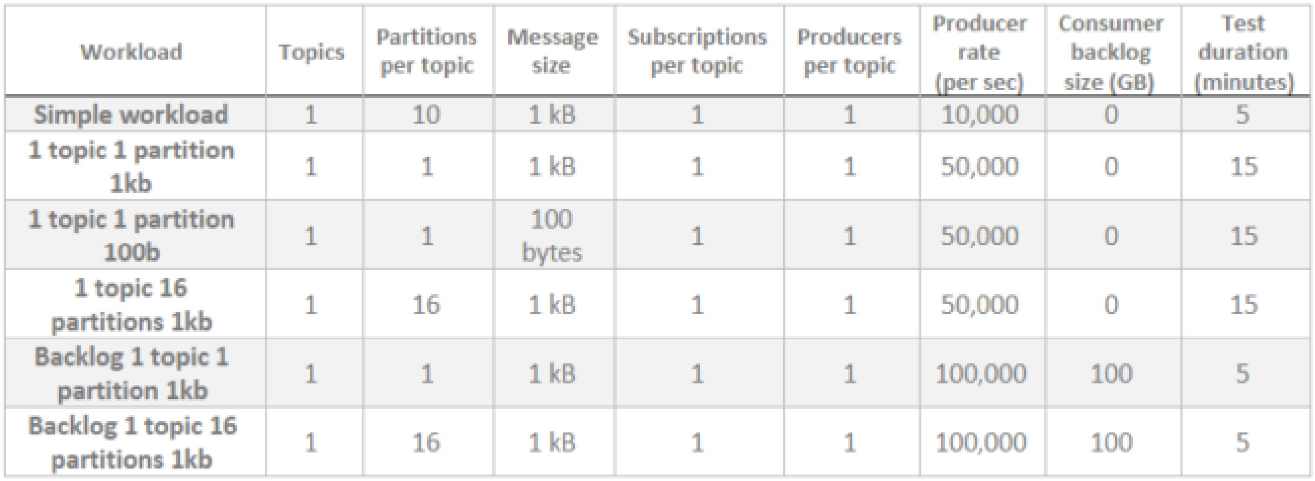
- Deployed in the Cloud – benchmarks are run on cloud infrastructure (but could be deployed on a local machine)
- Easy to use – only limited knowledge of Linux shell commands is required and a few commands will download the benchmark, launch the necessary cloud infrastructure, deploy resources, and run the tests
- Transparent – the benchmarking code is open source and publically available on GitHub
- Realistic – the benchmarks are oriented toward standard, real-life use cases rather than extreme edge cases

Workloads

Again, the benchmarking workloads were designed with real-life, common use cases in mind. The workloads are distinguished by the following factors:

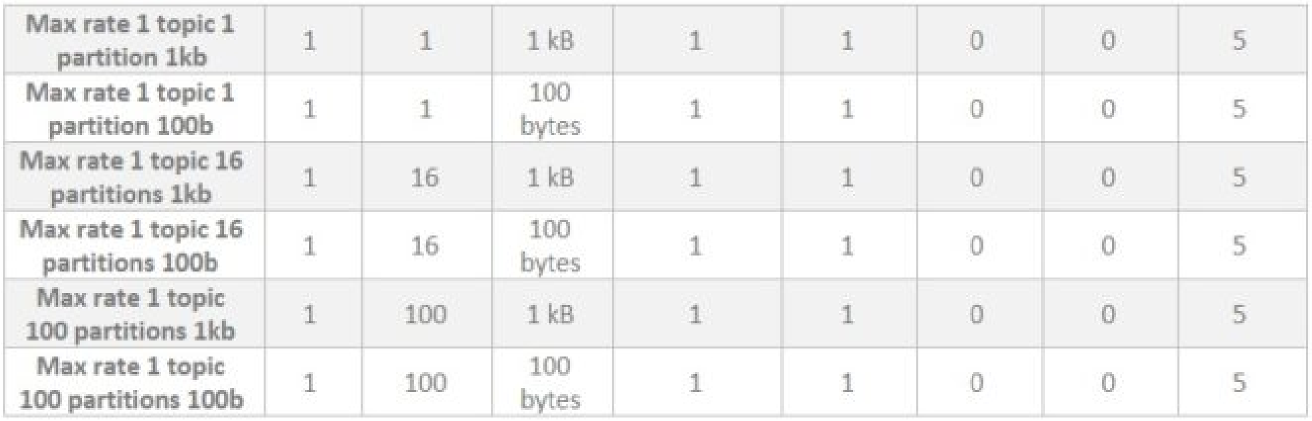
- The number of topics
- The size of the messages being produced and consumed
- The number of subscriptions per topic
- The number of producers per topic

- The rate at which producers produce messages (per second)[4]
- The size of the consumer's backlog (in gigabytes)
- The total duration of the test (in minutes)



Workload	Topics	Partitions per topic	Message size	Subscriptions per topic	Producers per topic	Producer rate (per sec)	Consumer backlog size (GB)	Test duration (minutes)
Simple workload	1	10	1 kB	1	1	10,000	0	5
1 topic 1 partition 1kb	1	1	1 kB	1	1	50,000	0	15
1 topic 1 partition 100b	1	1	100 bytes	1	1	50,000	0	15
1 topic 16 partitions 1kb	1	16	1 kB	1	1	50,000	0	15
Backlog 1 topic 1 partition 1kb	1	1	1 kB	1	1	100,000	100	5
Backlog 1 topic 16 partitions 1kb	1	16	1 kB	1	1	100,000	100	5

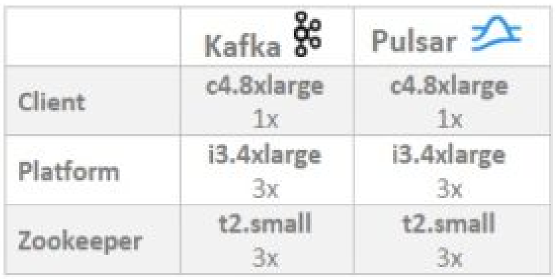
[4] A value of 0 for message production rate means that messages are produced as quickly as possible, with no rate limiting.



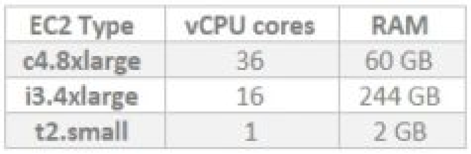
Max rate 1 topic 1 partition 1kb	1	1	1 kB	1	1	0	0	5
Max rate 1 topic 1 partition 100b	1	1	100 bytes	1	1	0	0	5
Max rate 1 topic 16 partitions 1kb	1	16	1 kB	1	1	0	0	5
Max rate 1 topic 16 partitions 100b	1	16	100 bytes	1	1	0	0	5
Max rate 1 topic 100 partitions 1kb	1	100	1 kB	1	1	0	0	5
Max rate 1 topic 100 partitions 100b	1	100	100 bytes	1	1	0	0	5

Demonstration of the Benchmark

To demonstrate this benchmark, we executed the benchmark on the provided workloads for Kafka and Pulsar. We utilized Amazon Web Services EC2 instances for our virtual machine clusters. The following table represents the EC2 instance types for the benchmark tests.



	Kafka	Pulsar
Client	c4.8xlarge 1x	c4.8xlarge 1x
Platform	i3.4xlarge 3x	i3.4xlarge 3x
Zookeeper	t2.small 3x	t2.small 3x



EC2 Type	vCPU cores	RAM
c4.8xlarge	36	60 GB
i3.4xlarge	16	244 GB
t2.small	1	2 GB

These instance types were selected, because we wanted adequate processing and memory resources available to execute the larger workloads without having any hardware constraints.

The benchmark modules were deployed on these clusters and the workloads for each platform were executed. The following sections offer some insights and discoveries we made.

Results comparison

In this section, we will compare results and present results by overlaying the performance of one platform over another.

Latency

We uncovered some interesting findings by comparing the benchmark results of Kafka and Pulsar.

Pulsar has message permanent durability turned on, i.e., write message log immediately to disk, as a default. To match this behavior, Kafka was configured to also immediately flush its log to disk, by configuring the benchmark topics with the following parameters[5]:

```
flush.messages=1

flush.ms=0
```

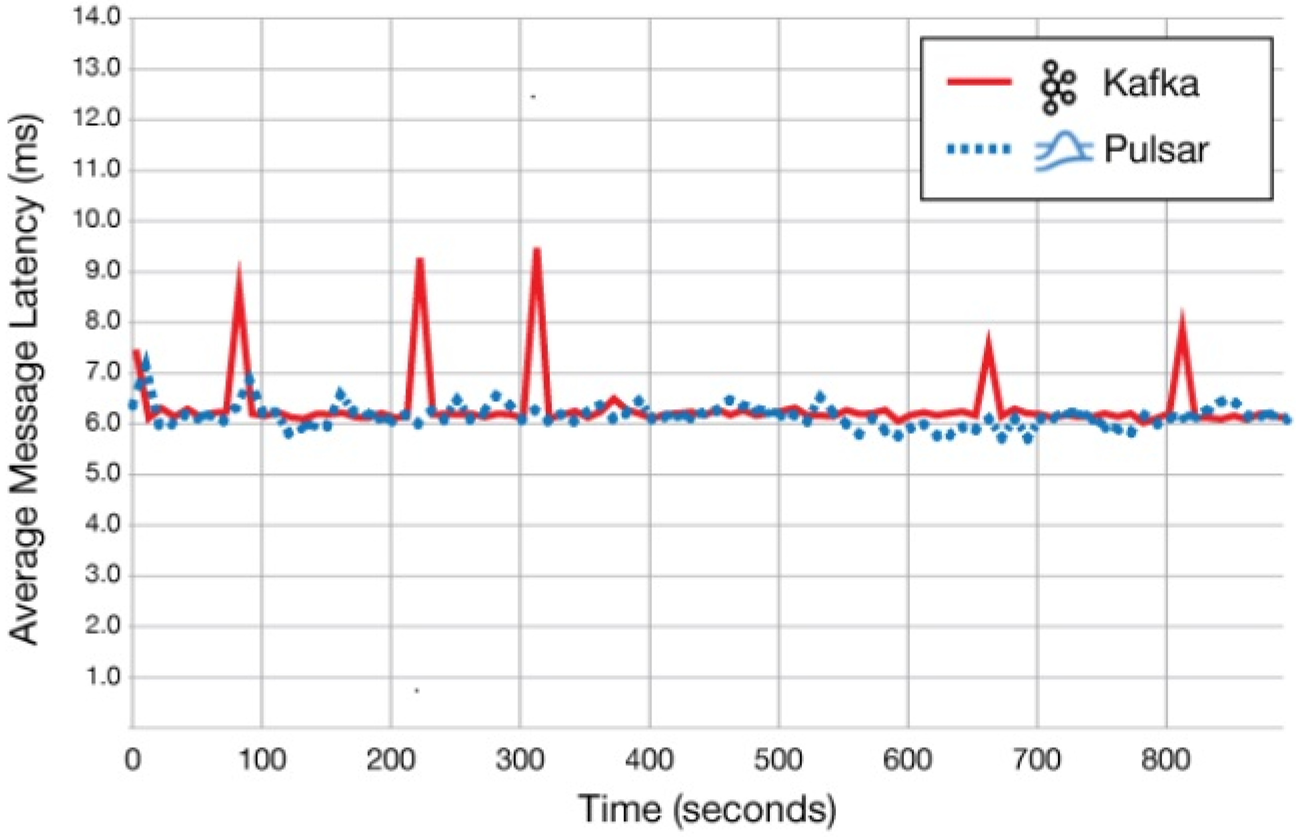
We wanted more of an apples-to-apples comparison.

First, we found differences in the average message latency of the two platforms—even with the workloads at the smaller end. Latency is the response time of a message queue. One way to describe latency is to use the real world example of a fast food restaurant's drive-through lane. Let us say as cars pull up and order, the restaurant is able to serve the customer in under 2 minutes 90% of the time, but 10% of the time it takes 5 minutes. That sounds really good for the average customer to have their meal served in 2 minutes. However, their total response time or latency also consists of the time the customer waited in line. What if a customer gets their order in 2 minutes, but they had to wait in line behind someone whom it took 5 minutes? Their total response time was 7 minutes. Thus, in message queues, latency is a function of individual message response times and the rate of arrival of new messages. When we measure latency, we attempting to understand overall response time—the lower, the better.

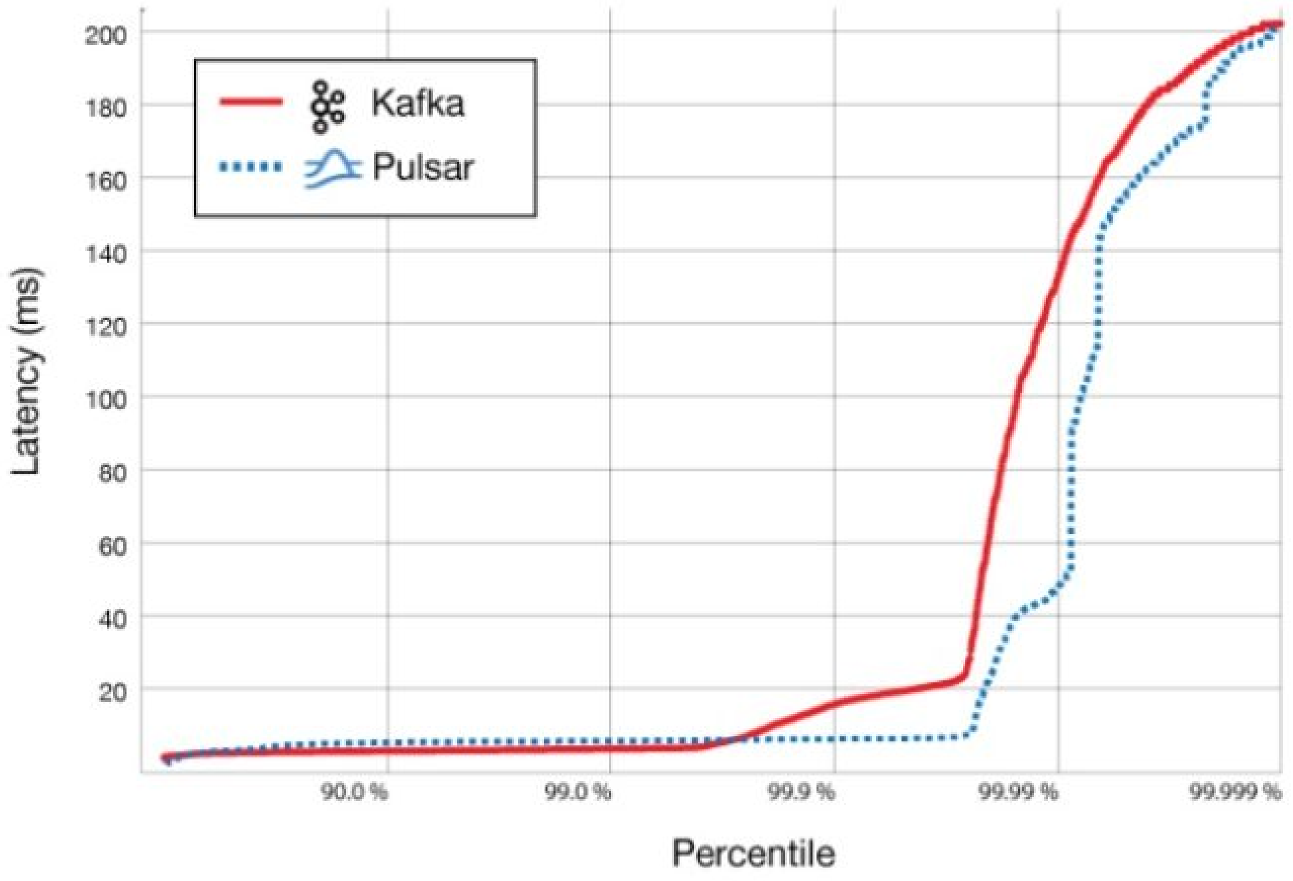
A good example of this was our results of the 1 topic, 1 partition, 1Kb message size workload for both Kafka and Pulsar. The following chart shows some spikes that occurred during the Kafka run. Somewhere after 300 seconds into the benchmark run, Kafka had some average latency responses well above its average of roughly 6 milliseconds for particular time intervals. Pulsar did not miss a beat.

---

[5] This configuration is available in the OpenMessaging framework by invoking kafka-sync.yaml when performing the benchmark runs.



Since the OpenMessaging benchmark issues a fixed rate of requests per second and measures the latency of each request issued synchronously, it forms a latency distribution. The distribution of latencies over the entire benchmark can be viewed using a high dynamic range (HDR) histogram. This steps back and views the total latency distribution from the “six nines” perspective, i.e., 90%, 99%, 99.9%, 99.99%, 99.999%, and 99.9999%. The lower the latency at higher percentages, the better. The following chart shows how the Kafka message queue was more “backed up” than the Pulsar queue during our benchmark runs.

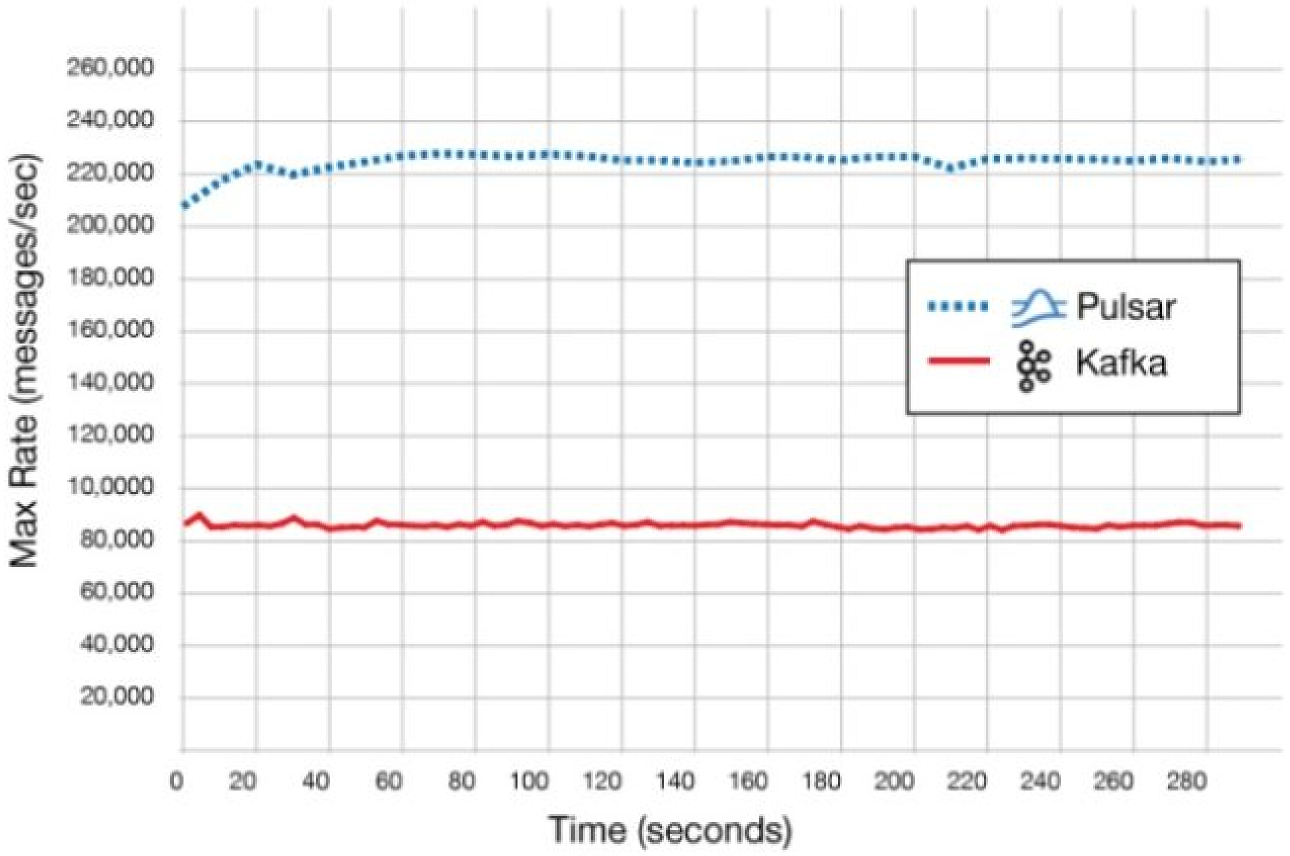


Overall latency in response time is a critical performance factor for most all use cases because it is a measure of overall experience—much like the fast food drive-through line from our example.

Maximum message rate

Perhaps more useful to sizing most use cases is message rate. Many times, we already know approximately the number of messages we need flowing through our queues or streams in a given time frame. Then the question becomes, can the platform handle the throughput we require? For this, you can measure maximum message rate. Basically, it means opening up the throttle, producing messages as fast as possible and seeing how the platform responds. The OpenMessaging benchmark has several workloads for these tests.

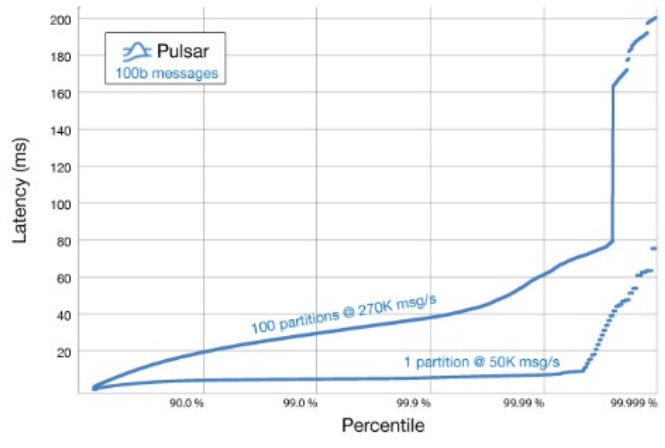
A quick comparison between Pulsar and Kafka reveals a striking difference. The following chart represents the maximum message rate on 1 topic over 1 partition with 100-byte messages. Clearly, Pulsar was outperforming Kafka by averaging 220,000+ messages per second compared to the 80,000+ messages per second on our Kafka cluster.

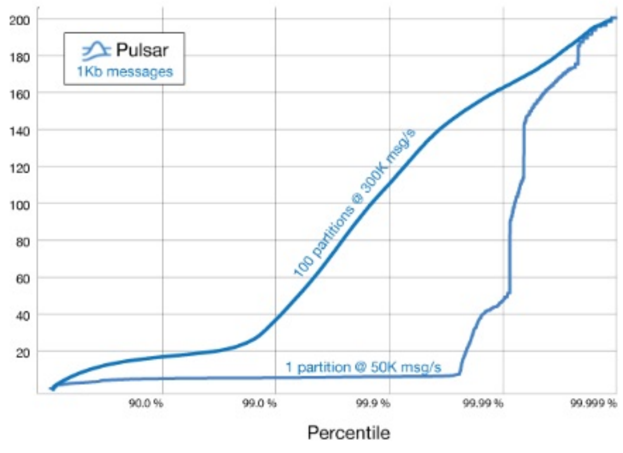


Results at scale

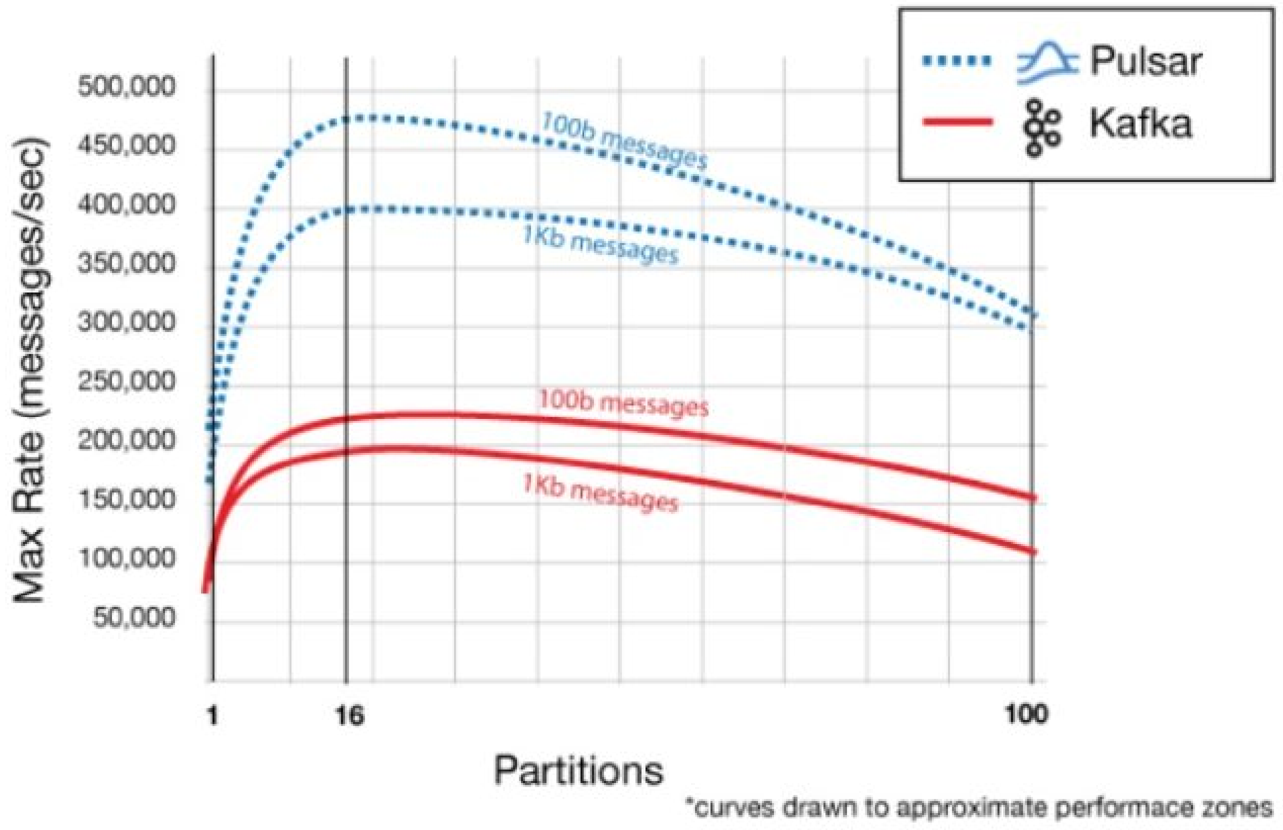
To aide you in meeting your specific requirements, it is useful to present a more cumulative, big picture view of our benchmark results, so that you can see performance at scale and create a range where your own requirements might fit.

First, it might be useful to see latency scaled. In these following Pulsar charts, you can see how the high dynamic range chart curves change as a function of message size, number of partitions, and the message rate. The other platforms also have similar scale characteristics. Kafka does not demonstrate low latency as consistently as Pulsar.





Second, let us examine how the platforms' maximum message rates scaled. This shows how the maximum rates compared across the platforms we benchmarked. It also is informative to understand this trend as you decide how to partition data in your own use case. Note that the curves were drawn to approximate the performance zones we might see at other partition sizes (other than the 1, 16, and 100 provided by the benchmark out of the box).



Summary

From a pure, raw performance perspective, Apache Pulsar outpaced Kafka in terms of maximum message rate and had less latency across all the workloads we attempted. As stated earlier, we believe performance to be a key selection criterion for those considering a streaming or message queuing solution. We viewed these benchmark results with a critical analytical point of view—particularly in key measures. For example, we paid close attention to latency at or above the 99th percentile. Remember that latency is total response time, not individual message processing time. If latency spikes occur (as we saw in the Kafka run shown above), and they start to happen regularly, spikes could snowball and performance could get really bad quickly in an avalanche. While Pulsar did not experience extreme spikes like Kafka, we did note that the variance of latency at the 99th percentile increased—we saw values of the 1KB messages and 100 partitions workload waffle back and forth between 25 and 70 milliseconds with one reading as high as 100.

Maximum message rate was also drawn out of this benchmark as a measure of just raw throughput. This is important because we believe the requirements and demand placed on these types of systems will only increase. Clearly, Pulsar outperformed Kafka, which makes a case for the platform for those enterprises who see rapid scaling needs in the near future. (What forward thinking enterprise would ***not*** see such growth coming in years to come?) Better performance scalability as partition count grows means that a system can support more data and users with a lower investment and cost.

Next Steps

Of course, this is our interpretation of these benchmark results, which ran pretty much "out of the box," without one particular use case in mind. The next steps for anyone reading this paper and evaluating these systems would be to perform your own tests. One of the purposes of this paper was not to necessarily declare a "winner." Instead, we set out to demonstrate this benchmark methodology with the expectation that you consider your own uses and expected load in production, take this benchmark framework, modify its configurations around your needs and required service levels, and then iterate through the workloads, platforms, and provision resources until you're within those limits. Use that as the starting point, and then scale out a plan to adequately prepare for growth.

If you elect to run the benchmarks, we have captured some of our experience in this paper and attempted to distill it for you in a way that makes it more transparent and easier to deploy. Please see the appendix for instructions and helpful hints to help you get off the ground running and avoid issues.

Remember, this benchmark is only a part of your overall assessment of streaming and message queuing technologies. The features, support, and interweaving services of the enterprise implementations of these platforms (such as Streamilo) will go a long way to filling all the gaps and unknowns around your use cases and needs.

Finally, you should also understand streaming and message queuing technology's place as a part of your overarching information management strategy. If you do not have such a strategy in place, we also recommend you seek out professional services and consulting firms, such as McKnight Consulting Group, to help you conduct a strategic assessment and develop an action plan to assess, select, and implement technologies such as these in a way that fits your information ecosystem and business goals.

4 Conclusion and takeaways

Streaming and message queuing have lasting value to organizations. We predict they will be **as** prevalent as ETL was and is in the world of data warehousing and integration. Streaming and messaging will be able to meet the data volume, variety, and timing requirements of the coming years. Moreover, the technology is still new or not yet a part of many organizations. Looking to the future, data driven organizations will benefit handsomely from these technologies because it will allow them to ingest data and operate at a scale that would have been practically impossible just a few years ago.

Additionally, this paper has shown there is a quality benchmark framework in place for your own evaluation purposes. There have been a lot of scattered benchmarking done as evidenced by Internet searches on the topic. The OpenMessaging benchmark is scalable, configurable, adaptable, easily deployable, and repeatable. It also highlights the key measures and metrics one needs to pay attention to in evaluating technologies against requirements and desired capabilities.

Finally, we do not hide the fact that Pulsar performed very well in our benchmark runs. This does offer a compelling case for the platform—the latest and greatest among the other tried and true platforms that have been around a number of years. As we have seen in this industry, it is often the newcomer that disrupts the status quo and forces an industry or discipline to evolve and adapt. We think Pulsar has that potential, and we expect to see it in wide-scale use in the near future.

5 Appendix – Deploying and Executing the Benchmark

Deploying and executing the prescribed OpenMessaging benchmarks is a simple procedure. Only a limited amount of experience with basic Linux commands is needed to complete the steps provided in the benchmark documentation on GitHub.

Setup and deployment

The following is an overview of the setup and deployment process:

First, you have to make sure you have an AWS account with Amazon[6]. Your local machine also needs a command line interface tool installed, so it can communicate with the instances you will create. You also have to generate a public/private key pair, so you can securely log into your instances. Instructions for that step is in the Git repository.

Second, you need to clone the Git repository (the link is a footnote on a previous page) to your local machine. You can download it from there or use git clone command from your terminal prompt (provided it is installed). You also should install Apache Maven. See the comments on that in the next section.

Next, an application called Terraform (that you install and execute locally) will launch clusters for each of the platforms mentioned above as Amazon Web Services (AWS) EC2 instances. This will include a client node from which you will execute the benchmarks, a multi-node cluster (3 by default) of the given platform (i.e., Kafka, Pulsar, RabbitMQ, or Artemis), and sometimes a cluster for other required supporting cast (e.g., Zookeeper, BookKeeper, et cetera.) For example, for Pulsar with the default settings, the Terraform cloud formation template will launch 7 instances into a Virtual Private Cloud:

- 1 Client node
- 3 Pulsar nodes
- 3 BookKeeper nodes

Of course, the defaults (cluster size, EC2 instance type, et cetera) can be adjusted by modifying the deployment YAML file per the instructions in the Git repository.

An application called Ansible (which you also install and execute locally) will also deploy instructions to your cluster members to setup the instances into a ready state, including mounting and formatting disks, installing software, launching service daemons, et cetera. This is done through what is called a playbook.

Once setup is complete, one only needs to connect to the client instance via secure shell (SSH) and execute the benchmarks using the instructions in the Git repository. If everything goes well, you should see a flurry of activity as the benchmark gives you updates on progress, for example:

```
19:23:05.057 [main] INFO - Pub rate 303510.6 msg/s / 296.4 Mb/s | Cons rate 30
```

Once a benchmark finishes, there is a Python script included that will generate informative charts in SVG format.

---

6 The benchmark was not created to be runnable only on AWS. The deployment scripts could be easily leveraged for other cloud platforms as well.

Helpful hints

There are a few requirements for the installation and helpful hints we offer for novice users:

- You will need a POSIX-type shell environment (MacOS, any Linux distribution, etc.) to perform the setup and installations, i.e., you cannot deploy the benchmark from a Microsoft Windows environment.
- If you only have a Windows machine locally, you can spin up a free-tier AWS EC2 instance (such as t2.micro) using Amazon Linux, RedHat, or any of the other popular Linux distributions. Then SSH into this instance and proceed with the steps listed in GitHub.
- You may also need to install some packages for your OS as prerequisites to what is already listed in the benchmark instructions, such as git and wget, using the package manager on your OS (Homebrew on MacOS, Yum on Fedora/RedHat, APT on Debian, etc.)
- Installing Apache Maven, Terraform, and Ansible are not as simple as using a package manager, but the instructions[7] are fairly straightforward.
- Once the benchmark completes, you can execute the Python directly from the client node you are logged into, and then secure copy the SVG files down to your local machine for analysis, or you can download the Python script and execute it locally. Note: you will need to install the Pygal package into Python using PIP. If you do not have PIP, you must install it first.

- Beware of the AWS charges! While not overly expensive instance by the hour, there is no need to leave the instances running after your benchmarks are complete. If you use the same EC2 types as we did (c4.8xlarge, i3.4xlarge x3, and t2.small x3), it costs approximately $130 per 24-hour period the instances are running, and that does not include any EBS storage costs.

---

[7] Instructions for installing Apache Maven can be found at https://maven.apache.org/install.html

6 About the Authors: William McKnight & Jake Dolezal

William is President of (McKnight Consulting Group Global Services). He is an internationally recognized authority in information management. His consulting work has included many of the Global 2000 and numerous mid-market companies. His teams have won several best practice competitions for their implementations and many of his clients have gone public with their success stories. His strategies form the information management plan for leading companies in various industries.

William is author of the books *Integrating Hadoop* and *Management: Strategies for Gaining a Competitive Advantage with Data*. William is a popular speaker worldwide and a prolific writer with hundreds of published articles and white papers. William is a distinguished entrepreneur, and a former Fortune 50 technology executive and software engineer. He provides clients with strategies, architectures, platform and tool selection, and complete programs to manage information.

Jake Dolezal has two decades of experience in the Information Management field with expertise in business intelligence, analytics, data warehousing, statistics, data modeling and integration, data visualization, master data management, and data quality. Jake has experience across a broad array of industries, including: healthcare, education, government, manufacturing, engineering, hospitality, and restaurant. He has a doctorate in information management from Syracuse University.

7 About Gigaom Research

Gigaom Research gives you insider access to expert industry insights on emerging markets. Focused on delivering highly relevant and timely research to the people who need it most, our analysis, reports, and original research come from the most respected voices in the industry. Whether you're beginning to learn about a new market or are an industry insider, Gigaom Research addresses the need for relevant, illuminating insights into the industry's most dynamic markets.

Visit us at: Gigaom.com/reports.