Stream Native

Benchmarking Pulsar and Kafka – A More Accurate Perspective on Pulsar's Performance

Sijie Guo

Penghui Li

Table of Contents

Benchmarking Pulsar and Kafka - A More Accurate Perspective on Pulsar's Performance

Executive Summary

Today, many companies are looking at real-time data streaming applications to develop new products and services. Organizations must first understand the advantages and differentiators of the different event streaming systems before they can select the technology best-suited to meet their business needs.

Benchmarks are one method organizations use to compare and measure the performance of different technologies. In order for these benchmarks to be meaningful, they must be done correctly and provide accurate information. Unfortunately, it is all too easy for benchmarks to fail to provide accurate insights due to any number of issues.

***Note**: This post presents StreamNative's response to Confluent's recent article "Benchmarking Apache Kafka, Apache Pulsar, and RabbitMQ: Which is the Fastest?"* For a brief overview of these systems, see our review of Pulsar vs. Kafka (part 1, part 2).

Confluent recently ran a benchmark to evaluate how Kafka, Pulsar, and RabbitMQ compare in terms of throughput and latency. According to Confluent's blog, Kakfa was able to achieve the "best throughput" with "low latency" and RabbitMQ was able to provide "low latency" at "lower throughputs". Overall, their benchmark declared Kafka the clear winner in terms of "speed".

While Kafka is an established technology, Pulsar is the top streaming technology of choice for many companies today, from global corporations to innovative start-ups. In fact, at the recent Splunk summit, conf20,

Sendur Sellakumar, Splunk's Chief Product Officer, discussed their decision to adopt Pulsar over Kafka:

"... we've shifted to Apache Pulsar as our underlying streaming. It is our bet on the long term architecture for enterprise-grade multi-tenant streaming." - Sendur Sellakumar, CPO, Splunk

This is just one of many examples of companies adopting Pulsar. These companies choose Pulsar because it provides the ability to horizontally and cost effectively scale to massive data volumes, with no single point of failure, in modern elastic cloud environments, like Kubernetes. At the same time, built-in features like automatic data rebalancing, multi-tenancy, geo-replication, and tiered storage with infinite retention, simplify operations and make it easier for teams to focus on business goals.

Ultimately, developers are adopting Pulsar for its features, performance, and because all of the unique aspects of Pulsar, mentioned above, make it well suited to be the backbone for streaming data.

Knowing what we know, we had to take a closer look at Confluent's benchmark to try to understand their results. We found two issues that were highly problematic. First, and the largest source of inaccuracy, is Confluent's limited knowledge of Pulsar. Without understanding the technology, they were not able to set-up the test in a way that could accurately measure Pulsar's performance.

Second, their performance measurements were based on a narrow set of test parameters. This limited the applicability of the results and failed to provide readers with an accurate picture of the technologies' capabilities across different workloads and real-world use cases.

In order to provide the community a more accurate picture, we decided to address these issues and repeat the test. Key updates included:

1. We updated the benchmark setup to include all of the durability levels supported by Pulsar and Kafka. This allowed us to compare throughput and latency at the same level of durability.
2. We fixed the OpenMessaging Benchmark (OMB) framework to eliminate the variants introduced by using different instances, and corrected configuration errors in their OMB Pulsar driver.
3. Finally, we measured additional performance factors and conditions, such as varying numbers of partitions and mixed workloads that contain writes, tailing-reads, and catch-up reads to provide a more comprehensive view of performance.

With these updates made, we repeated the test. The result - Pulsar significantly outperformed Kafka in scenarios that more closely resembled real-world workloads and matched Kafka's performance in the basic scenario Confluent used.

The following section highlights the most important findings. A more comprehensive performance report in the section StreamNative's Benchmark Results also gives detail of our test setup and additional commentary.

StreamNative Benchmark Result Highlights

#1 With the same durability guarantee as Kafka, Pulsar achieves 605 MB/s publish and end-to-end throughput (same as Kafka), and 3.5 GB/s catch-up read throughput (3.5 times higher than Kafka). Increasing the number of partitions and changing durability levels have no impact on Pulsar's throughput. However, the Kafka's throughput was severely impacted when changing the number of partitions or changing durability levels.

Table 1-1: Throughput differences between Pulsar and Kafka under different workloads with different durability guarantees

	Durability Levels	Partitions	Pulsar	Kafka
		1	300 MB/s	160 MB/s

Peak Publish + Tailing Reads Throughput (MB/s)	Level-1 Durability[1]	100	300 MB/s	420 MS/s
		2000	300 MB/s	300 MB/s
	Level-2 Durability	1	300 MB/s	180 MB/s
		100	605 MB/s	605 MB/s
		2000	605 MB/s	300 MB/s
Peak Catch-up Reads Throughput (MB/s)	Level-1 Durability	100	1.7 GB/s	1 GB/s
	Level-2 Durability	100	3.5 GB/s	1 GB/s

#2: Pulsar delivers significantly better latency than Kafka in each of the different test cases (including different number of subscriptions, different number of topics, and different durability guarantees).

Pulsar's 99th percentile latency is within the range of 5 and 15 milliseconds. Kafka's 99th percentile latency can go up to seconds and is hugely impacted by the number of topics, subscriptions and different durability guarantees.

Table 1-2: End-to-End P99 Latency between Pulsar and Kafka of different number of subscriptions with different durability guarantees

	Partitions & Subscriptions	Local Durability	Replication Durability	Pulsar	Kafka
End-to-End P99 Latency (ms) (Publish + Tailing Reads)	100 Partitions, 1 Subscription	Sync	Ack-1	5.86	18.75
			Ack-2	11.64	64.62
		Async	Ack-1	5.33	6.94
			Ack-2	5.55	10.43
	100 Partitions, 10	Sync	Ack-1	7.12	145.10
			Ack-2	14.65	1599.79

[1] See Section "An Overview of Durability in Distributed Systems" for detailed discussion on durability differences between Pulsar and Kafka.

	Subscriptions	Async	Ack-1	6.84	89.80
			Ack-2	6.94	1295.78

Table 1-3: End-to-End P99 Latency between Pulsar and Kafka of different number of topics with different durability guarantees

	Local Durability	Replication Durability	Partitions	Pulsar	Kafka
End-to-End P99 Latency (ms) (Publish + Tailing Reads)	Sync	Ack-1	100	5.86	18.75
			5000	6.26	79236
			10000	6.67	187840
		Ack-2	100	11.64	64.62
			5000	14.38	157960
			10000	15.78	197140
	Async	Ack-1	100	5.33	6.94
			5000	5.75	86641
			10000	6.64	184513
		Ack-2	100	5.55	10.43
			5000	6.20	116028
			10000	7.50	200793

#3: Pulsar provides significantly better I/O isolation than Kafka. Pulsar's 99th percentile publish latency remains around 5 milliseconds when there are consumers catching up on reading historic data. In contrast, Kafka's latency is severely impacted by catchup reads. Kafka's 99th percentile publish latency can increase from milliseconds to multiple seconds.

Table 1-4: Publish P99 Latency between Pulsar and Kafka with catching up reads

	Local Durability	Replication Durability	Pulsar	Kafka
Publish P99 Latency (ms)	Sync	Ack-1	5.89	13.48

(Mixed Workload)		Ack-2	15.39	2091.31
	Async	Ack-1	10.44	9.51
		Ack-2	35.51	1014.95

All of our benchmarks are open source, so curious readers can repeat the test for themselves. Additionally, you can dig deeper into the results and metrics, which are available in the repository.

Although our benchmark is more accurate and more comprehensive than Confluent's, it doesn't cover every scenario. Ultimately, no benchmark can replace testing done on your own hardware with your own workloads. We encourage you to evaluate additional variables and scenarios and to test using your own setups and environments.

A Deeper Look at Confluent's Benchmark

Confluent used the OpenMessaging Benchmark (OMB) Framework as the basis for their benchmark with a few modifications. In this section, we describe the issues we found in Confluent's benchmark and explain how they impacted Confluent's test results and led to erroneous conclusions.

Issues with Confluent Setup

A fundamental problem with Confluent's benchmark is that Pulsar was not set up properly. (We will talk more about this in the section on the StreamNative Benchmark). In addition to the issues tuning Pulsar, Pulsar and Kafka were set up with different durability guarantees. Because the level of durability impacts performance, the durability settings on both systems must be equivalent for a comparison to be meaningful.

Confluent's engineers used the default durability guarantee for Pulsar, which is a much stronger guarantee than the configuration which was used

for Kafka. Because increasing durability negatively impacts latency and throughput, Confluent's test placed a much higher demand on Pulsar than it did on Kafka. In the version of Pulsar used by Confluent, there was not yet support for reducing the durability down to a level to match Kafka, but such a future will be released as part of Pulsar in an upcoming release and was used in this test. Had their engineers used this new equivalent durability setting on both systems, the test results would have allowed for an accurate comparison. We certainly don't fault Confluent's engineers for not using a not-yet-released feature, however, the writeup failed to provide the necessary context for these results and treated that as though they were equivalent, that additional context which will be provided here.

Issues with OMB Framework

Confluent's benchmark followed OMB Framework guidelines, which recommend using the same instance type across multiple event streaming systems. However, in our testing we found large amounts of variance among different instances of the same type, particularly when it came to disk IO. In order to minimize this variance, we used the same instances from run to run for both Pulsar and Kafka, which we found significantly helped to increase the accuracy of the results, as even small variations in disk IO performance can result in much larger variance in overall system performance.. We would suggest the OMB framework guidelines be updated to include this recommendation in the future.

Issues with Confluent Methodology

Confluent's benchmark measured only a few, limited scenarios. For example, real-world workloads consist of writes, tailing reads, and catch-up reads. Tailing-reads occur when a consumer is reading recent messages near the "tail" of the log, which was the only scenario tested by Confluent. In contrast, a catch-up read is when a consumer has a large amount of backlog it must consume to "catch-up" to the tail of the log,

which is a common and critical task in a real-world system. Catch-up reads, when not taken into account, can severely impact the latency of writes and tailing reads. As Confluent's benchmark focused only on throughput and end-to-end latency, it fails to give a complete picture of expected behavior across a variety of workloads. Likewise to further give a result closer to real-world use cases, , we also considered it important to run the benchmark with varying numbers of subscriptions and partitions. Very few organizations only care about a few topics with a handful of partitions and consumers, they need the ability to have large numbers of different consumers with a large number of distinct topics/partitions to map to their business use cases

To summarize, we have outlined specific issues with Confluent's methodology in the following table:

Table 1-5: Issues with Confluent's benchmark methodology

Parameters Tested	**Exclusions**	**Limitations**
Writes and tailing reads	Catch-up reads	While maximum throughput and end-to-end latency are useful to illustrate the basic performance characteristics of an event streaming system, limiting the study to two parameters provides only a partial view of system performance.
1 subscription	Varying numbers of subscriptions / consumer groups	Did not show how the number of subscriptions impacts throughput and latency.
100 partitions	Varying numbers of partitions	Did not show how the number of partitions impacts throughput and latency.

Many of the issues with Confluent's benchmark stem from a limited understanding of Pulsar. To help others avoid these problems when running benchmarks in the future, we'll provide some insights on the technology.

Understanding Pulsar's durability guarantees is required in order to run an accurate benchmark, so we will begin our discussion there. We'll start with a general overview of durability in distributed systems, and then explain the differences between the durability guarantees offered by Pulsar and Kafka.

An Overview of Durability in Distributed Systems

Durability refers to maintaining system consistency and availability in the face of external problems, such as a hardware or operating system failure. Single-node storage systems (such as RDBMS) will "fsync" writes to disk to ensure maximum durability. Operating systems will typically buffer writes, which can be lost in the event of failure, but an fsync will ensure these data is written to physical storage. In distributed systems, durability typically comes from replication, with multiple copies of the data being distributed to different nodes that can fail independently. However, it is important not to conflate local durability (fsyncing data) with replication durability, as they both have a distinct purpose. In the following sections, we explain some key differences between these features and why both are important.

Replication durability and local durability

Distributed systems typically provide both replication durability and local durability. Separate mechanisms control each type of durability. You can use these mechanisms in various combinations to set the desired level of durability.

Replication durability is achieved by using an algorithm to create multiple copies of data so the same data can be stored in several locations to improve availability and accessibility. The number of replicas N, determines the system's failure tolerance, with many systems requiring a "quorum", or N/2 + 1 nodes, to acknowledge a write. Some systems offer the ability to continue to serve existing data with any single replica still available. This

replication mechanism is key to handling the total loss of an instance, with a new instance able to re-replicate data from the existing replicas, while also being critical to availability and consensus (which is beyond the scope of this discussion).

By contrast, local durability determines what an acknowledgement means at the individual node level. This requires fsyncing data to a persistent medium to ensure no data is lost, even if a power failure or hardware failure occurs. An fsync of data ensures that in the event of transient failure, where the machine can recover, the node has all the data for it's previous acknowledgements.

Durability Modes: Sync vs. Async

Different types of systems offer varying levels of durability guarantees. In general, the level of *overall* durability any given system can provide depends on the following:

- Whether the system fsyncs data to local disks
- Whether the system replicates data to multiple locations
- When the system acknowledges replication to a peer
- When the system acknowledges writes to the client

Among different systems, these choices vary widely and not all systems give users the option to control these values, but in general, systems that lack some of these mechanisms (such as replication in non-distributed systems) offer less durability.

To discuss this more concretely, we can define two durability modes which control when a system acknowledges writes, both internally for replication and to the client. These are "sync" and "async". These modes operate as described below.

- **Sync Durability**: The system returns a write response to the peer/client ONLY AFTER the data has been successfully f-synced to local disks (local durability) or replicated to multiple locations (replication durability).
- **Async Durability**: The system returns a write response to the peer/client BEFORE the data has been successfully f-synced to local disks (local durability) or replicated to multiple locations (replication durability).

Durability Levels: Measuring durability guarantees

Durability guarantees can take many forms and depend on the following variables:

- Whether data is stored locally, replicated in multiple locations, or both
- When writes are acknowledged (sync vs. async)

Like with durability mode, we define some durability "levels" which we can use to differentiate between different distributed systems. We define four levels. Table 6 describes each, from the highest level of durability to the lowest.

Table 1-6: Durability Levels of Distributed Systems

Level	Replication	Local	Operation
1	Sync	Sync	The system returns a write response to the client ONLY AFTER the data has been replicated to multiple (at least the majority of) locations AND each replica has been successfully fsync-ed to local disks.
2	Sync	Async	The system returns a write response to the client ONLY AFTER the data has been replicated to multiple (at least the majority of) locations.There is no guarantee that each replica has successfully fsync-ed to local disks.

3	Async	Sync	The system returns a write response to the client when one replica has been successfully fsync-ed to a local disk. There is no guarantee that data is replicated to the other locations.
4	Async	Async	The system returns a write response to the client immediately after the data has been replicated to multiple locations. There are no replication or local durability guarantees.

Most distributed relational database management systems (such as NewSQL databases) guarantee the highest level of durability; therefore, they would be categorized as Level 1.

Much like a database, Pulsar is a Level 1 system that provides the highest level of durability by default. In addition, Pulsar allows the option to customize the desired durability level for each application individually. By contrast, Kafka can be configured to operate as either a Level 2 or Level 4 system.

We'll look at the durability capabilities of each in detail, beginning with Pulsar.

Durability in Pulsar

Pulsar offers durability guarantees at all levels. It can replicate data to multiple locations and fsync data to local disks. Pulsar has two durability modes (sync and async described earlier). Each option is individually configurable. You can use them in various combinations to customize settings for individual use cases.

Pulsar controls replication durability using a raft-equivalent, quorum-based replication protocol. You can tune the durability mode for replication durability by adjusting the `ack-quorum-size` and `write-quorum-size`

parameters. The settings for these parameters are described in Table 7 below. The durability levels supported by Kafka are described in Table 8 below. (A detailed discussion of Pulsar's replication protocol and consensus algorithm are beyond the scope of this article; however, we will explore these areas in depth in a future blog post.)

Table 1-7: Durability Configuration Settings in Pulsar

Location	**Configuration Settings**	**Durability Mode**
Replication	ackQuorumSize = 1	Async
	ackQuorumSize ≥ writeQuorumSize / 2 + 1	Sync
Local	(default) journalWriteData = true journalSyncData = true	Sync
	journalWriteData = true journalSyncData = false	Async
	journalWriteData = false journalSyncData = false	Async

Table 1-8: Durability Levels in Pulsar

Durability Level	**Replication Durability**	**Local Durability**
Level 1	Sync: ackQuorumSize ≥ writeQuorumSize / 2 + 1	Sync: journalWriteData = true journalSyncData = true
Level 3	Async: ackQuorumSize = 1	Sync: journalWriteData = true journalSyncData = true
Level 2	Sync: ackQuorumSize ≥ writeQuorumSize / 2 + 1	Async: journalWriteData = true journalSyncData = false
Level 4	Async: ackQuorumSize = 1	Async: journalWriteData = true journalSyncData = false
Level 2	Sync:	Async:

	ackQuorumSize ≥ writeQuorumSize / 2 + 1	journalWriteData = false journalSyncData = false
Level 4	Async: ackQuorumSize = 1	Async: journalWriteData = false journalSyncData = false

Pulsar controls local durability by writing and/or fsyncing data to a journal disk(s). Pulsar also provides options for tuning the local durability mode using the configuration parameters in Table 9:

Table 1-9: Pulsar's Local Durability Mode Parameters

Parameter	Description	Values
journalWriteData	Controls whether a bookie writes data to its journal disks before persisting data to the ledger disks	true = enable journaling false = disable journaling
journalSyncData	Controls whether a bookie fsyncs data to journal disks before returning a write acknowledgement to brokers	true = enable fsync false = disable fsync

Durability in Kafka

Kafka offers two durability levels: Level 2 and Level 4. Kafka can provide replication durability at Level 2, but offers no durability guarantees at Level 4 because it lacks the ability to fsync data to disks before acknowledging writes.

Kafka's ISR replication protocol controls replication durability. You can tune Kafka's replication durability mode by adjusting the acks and min.insync.replicas parameters associated with this protocol. The settings

for these parameters are described in Table 10 below. The durability levels supported by Kafka are described in Table 11 below. (A detailed explanation of Kafka's replication protocol is beyond the scope of this article; however, we will explore how Kafka's protocol differs from Pulsar's in a future blog post.)

Table 1-10: Durability Configuration Settings in Kafka

	Configuration Settings	**Durability Mode**
Replication Durability	acks = 1	Async-durable
	acks = all	Sync-durable
Local Durability	Default fsync settings	Async-durable
	flush.messages = 1 flush.ms = 0	Async-durable

Table 1-11: Durability Levels in Kafka

Durability Level	**Replication Durability**	**Local Durability**
Level 2	Sync: acks = all	Async: Default fsync settings
Level 4	Async: acks = 1	Async: Default fsync settings
Level 2	Sync: acks = all	Async: flush.messages = 1 flush.ms = 0
Level 4	Async: acks = 1	Async: flush.messages = 1 flush.ms = 0

Unlike Pulsar, Kafka does not write data to a separate journal disk(s). Instead, Kafka acknowledges writes before fsyncing data to disks. This operation minimizes I/O contention between writes and reads, and prevents performance degradation.

Kafka's does offer the ability to fsync after every message, with the above flush.messages = 1 and flush.ms = 0, and while this can be used to greatly reduce the likelihood of message loss, it still does not wait for the fsync to acknowledgement to the client, which ultimately means it is still possible to encounter some data loss, though unlikely.

Kafka's inability to journal data makes it vulnerable to data loss in the event of a machine failure or power outage. This is a significant weakness, and one of the main reasons why Tencent chose Pulsar for their new billing system.

Durability Differences Between Pulsar and Kafka

Pulsar's durability settings are highly configurable and allow users to optimize durability settings to meet the requirements of an individual application, use case, or hardware configuration.

Because Kafka offers less flexibility, depending on the scenario, it is not always possible to establish equivalent durability settings in both systems. This makes benchmarking difficult. To address this, the OMB Framework recommends using the closest settings available.

With this background, we can now describe the gaps in Confluent's benchmark. Confluent attempted to simulate Pulsar's fsyncing behavior. In Kafka, the settings Confluent chose provide async durability. However, the settings they chose for Pulsar provide sync durability. This discrepancy produced flawed test results that inaccurately portrayed Pulsar's performance as inferior. As you will see when we review the results of our own benchmark later, Pulsar performs as well as or better than Kafka, while offering stronger durability guarantees.

StreamNative Benchmark

To get a more accurate picture of Pulsar's performance, we needed to

address the issues with the Confluent benchmark. We focused on tuning Pulsar's configuration, ensuring the durability settings on both systems were equivalent, and including additional performance factors and conditions, such as varying numbers of partitions and mixed workloads, to enable us to measure performance across different use cases. The following sections explain the changes we made in detail.

StreamNative Setup

Our benchmarking setup included all the durability levels supported by Pulsar and Kafka. This allowed us to compare throughput and latency at the same level of durability. The durability settings we used are described below.

Replication Durability Setup

Our replication durability setup was identical to Confluent's. Although we made no changes, we are sharing the specific settings we used in Table 12 for completeness.

Table 1-12: Replication Durability Setup Settings

	Durability Mode	Configurations
Pulsar	Sync	ensemble-size=3 write-quorum-size=3 ack-quorum-size=2
	Async	ensemble-size=3 write-quorum-size=3 ack-quorum-size=1
Kafka	Sync	replicas=3 acks=all min.insync.replicas=2
	Async	replicas=3 acks=1 min.insync.replicas=2

Local Durability Set-Up

A new Pulsar feature gives applications the option to skip journaling, which relaxes the local durability guarantee, avoids write amplification, and improves write throughput. (This feature will be available in the next release of Apache Bookkeeper). However, this feature will not be made the default, nor do we recommend it for most scenarios, as it still introduces the potential for message loss.

We used this feature in our benchmark to ensure an accurate performance comparison between the two systems. Bypassing journaling on Pulsar provides the same local durability guarantee as Kafka's default fsync settings.

Pulsar's new feature includes a new local durability mode (Async - Bypass journal). We used this mode to configure Pulsar to match Kafka's default level of local durability. Table 13 shows the specific settings for our benchmark.

Table 1-13: Local Durability Setup Settings for StreamNative's Benchmark

	Durability Mode	**Configuration Values**
Kafka	Async	flush.messages=1 flush.ms=0
	Async (default)	flush.messages=10000 (default) flush.ms=1000 (default)
Pulsar	Sync (default)	journalWriteData=true journalSyncData=true journalMaxGroupWaitMSec=1
	Async (write to journal)	journalWriteData=true journalSyncData=false journalMaxGroupWaitMSec=1 journalPageCacheFlushIntervalMSec=1000
	Async	journalWriteData=false

	(bypass journaling)	journalSyncData=false

StreamNative Framework

We fixed some issues in Confluent's OMB Framework fork and corrected configuration errors in their OMB Pulsar driver. The new benchmarking code we developed, including the fixes described below, is available as open source.

Fixes in the OMB Framework

Confluent followed the OMB Framework's recommendation to use two sets of instances—one for Kafka and another for Pulsar. For our benchmark, we allocated one set of three instances to eliminate variations. In our first test, we deployed all three instances on Pulsar. Then, we repeated the test on Kafka using the same set of instances.

Because we used the same machines for benchmarking different systems, we cleared the filesystem pagecache before each run. This ensured the current test would not be impacted by previous activity.

Fixes in the OMB Pulsar Driver Configuration

We fixed a number of errors in Confluent's OMB Pulsar driver configuration. The following sections explain the specific changes we made to the broker, bookie, producer, consumer, and Pulsar image.

Broker Changes

Pulsar brokers use the `managedLedgerNewEntriesCheckDelayInMillis` parameter to determine the length of time (in milliseconds) a caught-up subscription must wait before dispatching messages to its consumers. In the OMB Framework, the value for this parameter was set to `10`. This was

the main reason why Confluent's benchmark inaccurately showed Pulsar to have higher latency than Kafka. We changed the value to `0` to emulate Kafka's latency behavior on Pulsar. After making this change, Pulsar showed significantly better latency than Kafka in all test cases.

Additionally, to optimize performance, we increased the value of the `bookkeeperNumberOfChannelsPerBookie` parameter from `16` to `64` to prevent any single Netty channel between a broker and a bookie from becoming a bottleneck. Such bottlenecks cause high latency when large volumes of messages accumulate in a Netty IO queue.

We intend on providing this guidance more clearly in the Pulsar documentation to help users who are looking to optimize entirely for end-to-end latency.

Bookie Changes

We added a new bookie configuration to test Pulsar's performance when bypassing journaling. See the Durability section for a discussion on this and recall that will this feature, we more closely match Kafka's durability guarantees

To test the performance of this feature, we built a customized image based on the official Pulsar 2.6.1 release to include this change. (For more details, see Pulsar Image.)

We configured the following settings manually to bypass journaling in Pulsar.

```
journalWriteData=false
journalSyncData=false
```

Additionally, we changed the value of the `journalPageCacheFlushIntervalMSec` parameter from `1` to `1000` to

benchmark async local durability (journalSyncData=false) in Pulsar. Increasing the value enabled Pulsar to simulate Kafka's flushing behavior as described below.

Kafka ensures local durability by flushing dirty pages in the filesystem page cache to disks. Data is flushed by a set of background threads called pdflush. Pdflush is configurable and the wait time between flushes is typically set to 5 seconds. Setting Pulsar's journalPageCacheFlushIntervalMSec parameter to 1000 is equivalent to a 5-second pdflush interval on Kafka. Making this change enabled us to benchmark async local durability more precisely and achieve a more accurate comparison between Pulsar and Kafka.

Producer Changes

Our batching configuration was identical to Confluent's with one exception: We increased the switch interval to make it longer than the batch interval. Specifically, we changed the value of the batchingPartitionSwitchFrequencyByPublishDelay parameter from 1 to 2. This change ensured Pulsar's producer would focus on only one partition during each batching period.

Setting the switch interval and the batch interval to the same value can cause Pulsar to switch partitions more often than necessary, which generates too many small batches and can potentially impact throughput. Making the switch interval larger than the batch interval minimizes this risk.

Consumer Changes

Pulsar clients use receiver queues to apply back pressure when applications are unable to process incoming messages fast enough. The size of the consumer receiver queue can affect end-to-end latency. A larger queue can pre-fetch and buffer more messages than a smaller one.

Two parameters determine the size of the receiver queue: `receiverQueueSize` and `maxTotalReceiverQueueSizeAcrossPartitions`. Pulsar calculates the receiver queue size as follows:

Math.min(receiverQueueSize, maxTotalReceiverQueueSizeAcrossPartitions / number of partitions)

For example, if `maxTotalReceiverQueueSizeAcrossPartitions` is set to 50000 and you have 100 partitions, the Pulsar client sets the consumer's receiver queue size to 500 on each partition.
For our benchmark, we increased `maxTotalReceiverQueueSizeAcrossPartitions` from `50000` to `5000000`. This tuning optimization ensured consumers would not apply back pressure.

Pulsar Image

We built a customized Pulsar release (v. 2.6.1-sn-16) to include the Pulsar and BookKeeper fixes described above. Version 2.6.1-sn-16 is based on the official Pulsar 2.6.1 release and available to download at https://github.com/streamnative/pulsar/releases/download/v2.6.1-sn-16/apache-pulsar-2.6.1-sn-16-bin.tar.gz .

StreamNative Methodology

We updated Confluent's benchmarking methodology to get a more comprehensive view of performance using real-world workloads. Specifically, we made the following changes for our test:

- Added catch-up reads to evaluate the following:
  - The maximum level of throughput each system can achieve when processing catch-up reads
  - How reads impact publish and end-to-end latency

- Varied the number of partitions to see how each change impacted throughput and latency
- Varied the number of subscriptions to see how each change impacted throughput and latency

Our benchmark scenarios measured the following types of workloads:

- **Maximum Throughput**: What is the maximum throughput each system can achieve?
- **Publish & Tailing Read Latency**: What are the minimum publish and end-to-end tailing latency levels each system can achieve for a given throughput?
- **Catch-up Reads**: What is the maximum throughput each system can achieve when reading messages from a large backlog?
- **Mixed Workload**: What are the minimum publish and end-to-end tailing latency levels each system can achieve while consumers are catching
  up? How do catch-up reads impact publish latency and end-to-end tailing latency?

Testbed

The OMB Framework recommends specific testbed definitions (for instance types and JVM configurations) and workload driver configurations (for the producer, consumer, and server side). Our benchmark used the same testbed definitions as Confluent's. These testbed definitions can be found in our fork within Confluent's OMB repository.

Below, we highlight the disk throughput and disk fsync latency we observed. These hardware metrics are important to consider when interpreting benchmark results.

Disk Throughput

Our benchmark used the same instance type as Confluent's—specifically, `i3en.2xlarge` (with 8 vCores, 64 GB RAM, 2 x 2, 500 GB NVMe SSDs). We confirmed that `i3en.2xlarge` instances can support up to ~655 MB/s of write throughput across two disks. See the `dd` result below.

```
Disk 1
dd if=/dev/zero of=/mnt/data-1/test bs=1M count=65536
oflag=direct
65536+0 records in
65536+0 records out
68719476736 bytes (69 GB) copied, 210.08 s, 327 MB/s

Disk 2
dd if=/dev/zero of=/mnt/data-2/test bs=1M count=65536
oflag=direct
65536+0 records in
65536+0 records out
68719476736 bytes (69 GB) copied, 209.635 s, 328 MB/s
```

Disk data sync latency

It is critical to capture the fsync latency on NVMe SSDs when running latency-related tests. We observed that the 99th percentile fsync latency on these 3 instances varies from 1 millisecond to 6 milliseconds as shown in the following diagram. As was mentioned earlier, we saw large amounts of variance of disks from different instances. That was primarily manifested in this latency and we found a set of instances that exhibited consistent latency.

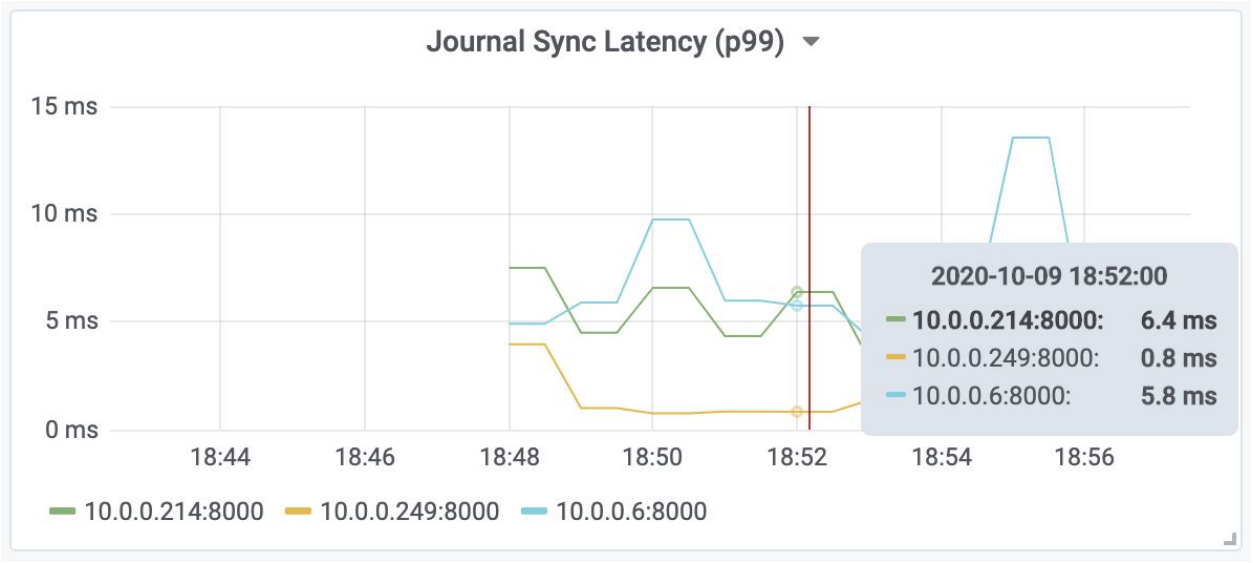

Figure 1-1: 99th percentile fsync latency on 3 different instances

StreamNative Benchmark Results

We have summarized our benchmark results below. You can find our complete benchmark report in the next section.

Maximum Throughput Test

Note: See the full report of "Maximum Throughput Test" here.

The Maximum Throughput Test was designed to determine the maximum throughput each system can achieve when processing workloads that include publish and tailing-reads under different durability guarantees. We also varied the number of topic partitions to see how each change impacted the maximum throughput.

We found that:

1. When configured to provide level-1 durability guarantees (sync replication durability, sync local durability), Pulsar achieved a maximum throughput of ~300 MB/s, which reached the physical limit of the journal disk's bandwidth. Kafka was able to achieve ~420 MB/s

with 100 partitions. It should be noted that when providing level-1 durability, Pulsar was configured to use one disk as journal disk for writes and the other disk as ledger disk for reads, comparing to Kafka use both disks for writes and reads. While Pulsar's setup is able to provide better I/O isolation, its throughput was also limited by the maximum bandwidth of a single disk (~300 MB/s). Alternative disk configurations can be beneficial to Pulsar and allow for more cost effective operation, which will be discussed in a later blog post.

2. When configured to provide level-2 durability (sync replication durability and async local durability), Pulsar and Kafka each achieved a max
   throughput of ~600 MB/s. Both systems reached the physical limit of disk bandwidth.
3. The maximum throughput of Kafka on one partition is only ½ of the max throughput of Pulsar.
4. Varying the number of partitions had no effect on Pulsar's throughput, but it did affect Kafka's.
   a. Pulsar sustained maximum throughput (~300 MB/s under a level-1 durability guarantee and ~600 MB/s under a level-2 durability guarantee) as the number of partitions was increased from 100 to 2000.
   b. Kafka's throughput decreased by half as the number of partitions was increased from 100 to 2000.

Publish & End-to-End Latency Test

Note: See the full report of "Maximum Throughput Test" here.

The Publish & End-to-End Latency Test was designed to determine the lowest latency each system can achieve when processing workloads that consist of publish and tailing-reads under different durability guarantees. We varied the number of subscriptions and the number of partitions to see how each change impacted both publish and end-to-end latency.

We found that

1. Pulsar's publish and end-to-end latency were significantly (up to hundreds of times) lower than Kafka's in all test cases, which evaluated various durability guarantees and varying numbers of partitions and subscriptions. Pulsar's 99th percentile publish latency and end-to-end latency stayed within 10 milliseconds, even as the number of partitions
   was increased from 100 to 10000 or as the number of subscriptions was
   increased from 1 to 10.
2. Kafka's publish and end-to-end latency was greatly affected by variations in the numbers of subscriptions and partitions.
   a. Both publish and end-to-end latency increased from -5 milliseconds to ~13 seconds as the number of subscriptions was
      increased from 1 to 10.
   b. Both publish and end-to-end latency increased from ~5 milliseconds to ~200 seconds as the number of topic partitions was increased from 100 to 10000.

Catch-up Read Test

Note: See the full report of "Maximum Throughput Test" here.

The Catch-up Read Test was designed to determine the maximum throughput each system can achieve when processing workloads that contain catch-up reads only. At the beginning of the test, a producer sent messages at a fixed rate of 200K per second. When the producer had sent 512GB of data, consumers began to read the messages that had been received. The consumers processed the accumulated messages and had no difficulty keeping up with the producer, which continued to send new messages at the same speed.

When processing catch-up reads, Pulsar's maximum throughput was 3.5 times faster than Kafka's. Pulsar achieved a maximum throughput of 3.5 GB/s (3.5 million messages/second) while Kafka achieved a throughput of only 1 GB/s (1 million messages/second).

Mixed Workload Test

Note: See the full report of "Maximum Throughput Test" here.
This Mixed Workload Test was designed to determine the impact of catch-up reads on publish and tailing reads in mixed workloads. At the beginning of the test, producers sent messages at a fixed rate of 200K per second and consumers consume messages in tailing mode. After the producer produces 512GB of messages, it will start a new set of catch-up consumers to read all the messages from the beginning. At the same time, producers and existing tailing-read consumers continued to publish and consume messages at the same speed.

We tested Kafka and Pulsar using different durability settings and found that catch-up reads seriously affected Kafka's publish latency, but had little impact on Pulsar. Kafka's 99th percentile publish latency increased from 5 milliseconds to 1-3 seconds. However, Pulsar maintained a 99th percentile publish latency ranging from several milliseconds to tens of milliseconds.

Conclusion

A tricky aspect of benchmarks is that they often represent only a narrow combination of business logic and configuration options, which may or may not reflect real-world use cases or best practices. Benchmarks can further be compromised by issues in their framework, set-up, and methodology. We noted all of these issues in the recent Confluent benchmark.

At the community's request, the team at StreamNative set out to run this benchmark in order to provide knowledge, insights, and transparency into Pulsar's true performance capabilities. In order to run a more accurate benchmark, we identified and fixed the issues with the Confluent benchmark, and also added new test parameters that would provide insights into how the technologies compared in more real-world use cases.

The results to our benchmark showed that, with the same durability guarantee as Kafka, Pulsar is able to outperform Kafka in workloads resembling real-world use cases and to achieve the same end-to-end through as Kafka in Confluent's limited use case. Furthermore, Pulsar delivers significantly better latency than Kafka in each of the different test cases, including varying subscriptions, topics, and durability guarantees, and better I/O isolation than Kafka.

As noted, no benchmark can replace testing done on your own hardware with your own workloads. We encourage you to test Pulsar and Kafka using your own setups and workloads in order to understand how each system performs in your particular production environment. If you have any questions on Pulsar best practices as you go through, please reach out to us directly, or feel free to join the Pulsar Slack channel.

In the next few months we will publish a series of blog posts to help the community better understand and leverage Pulsar to meet their business needs. Specifically, we will show the performance of Pulsar in different workloads and setups, how to select and size your hardware across different cloud providers and on-prem environments, and show how you can use Pulsar to build the most cost effective streaming platform.

Benchmarking Pulsar and Kafka - The Full Benchmark Report

Having identified multiple issues in Confluent's approach to evaluating various performance factors, we decided to repeat their benchmark on Pulsar and Kafka with some adjustments. We wanted to improve the accuracy of the test to facilitate more meaningful comparisons between the two systems. We also wanted to get a more comprehensive view, so we broadened the scope of our test to include additional performance measures and simulated real-world scenarios.

Our benchmark repeated Confluent's original tests with the appropriate corrections and included all the durability levels supported by Pulsar and Kafka. As a result, we were able to compare throughput and latency at equivalent levels of durability. In addition, we benchmarked new performance factors and conditions, such as varying numbers of partitions, subscriptions, and clients. We also emulated real-world use cases by testing mixed workloads containing writes, tailing-reads, and catch-up reads.

In this section, we describe the tests we performed in detail and share our results and conclusions.

Maximum Throughput Test

The following is the test setup.

We designed this test to determine the maximum throughput Pulsar and Kafka can achieve when processing workloads that consist of publish and tailing-reads. We varied the number of partitions to see how each change impacted throughput. Our test strategy included the following principles and expected guarantees:

- Each message was replicated three times to ensure fault tolerance.

- We varied the number of acknowledgements to determine the maximum throughput of each system under various replication durability guarantees.
- We enabled batching for Kafka and Pulsar, batching up to 1 MB of data for a maximum of 10 ms.
- We tested varying numbers of partitions—specifically, 1, 100, and 2000—to measure the maximum throughput for each condition.
- When benchmarking the maximum throughput for 100 and 2000 partitions, we ran 2 producers and 2 consumers.
- When benchmarking the maximum throughput for a single partition, we varied the number of producers and consumers to measure changes in throughput under different conditions.
- We used a message size of 1 KB.
- For each scenario, we tested the maximum throughput under various durability levels.

The following is the result for each test.

#1 100 partitions, 1 subscription, 2 producers / 2 consumers

Our first test benchmarked maximum throughput on Pulsar and Kafka with 100 partitions under two different durability guarantees. We used one subscription, two producers, and two consumers for each system. Our test results are described below.

- When configured to provide Level-1 durability guarantees (sync replication durability, sync local durability), Pulsar achieved a maximum throughput of ~300 MB/s, which reached the physical limit of the journal disk's bandwidth. Kafka was able to achieve ~420 MB/s with 100 partitions. It should be noted that when providing level-1 durability, Pulsar was configured to use one disk as journal disk for writes and the other disk as ledger disk for reads, comparing to Kafka use both disks for writes and reads. While Pulsar's setup is able to provide better I/O isolation, its throughput was also limited by the

maximum bandwidth of a single disk (~300 MB/s). Alternative disk configurations can be beneficial to Pulsar and allow for more cost effective operation, which will be discussed in a later blog post.

- When configured to provide Level-2 durability guarantees (sync replication durability, async local durability), Pulsar and Kafka each achieved a maximum throughput of ~600 MB/s. Both systems reached the physical limit of disk bandwidth.

Figure 1 shows the maximum throughput on Pulsar and Kafka with 100 partitions under sync local durability.

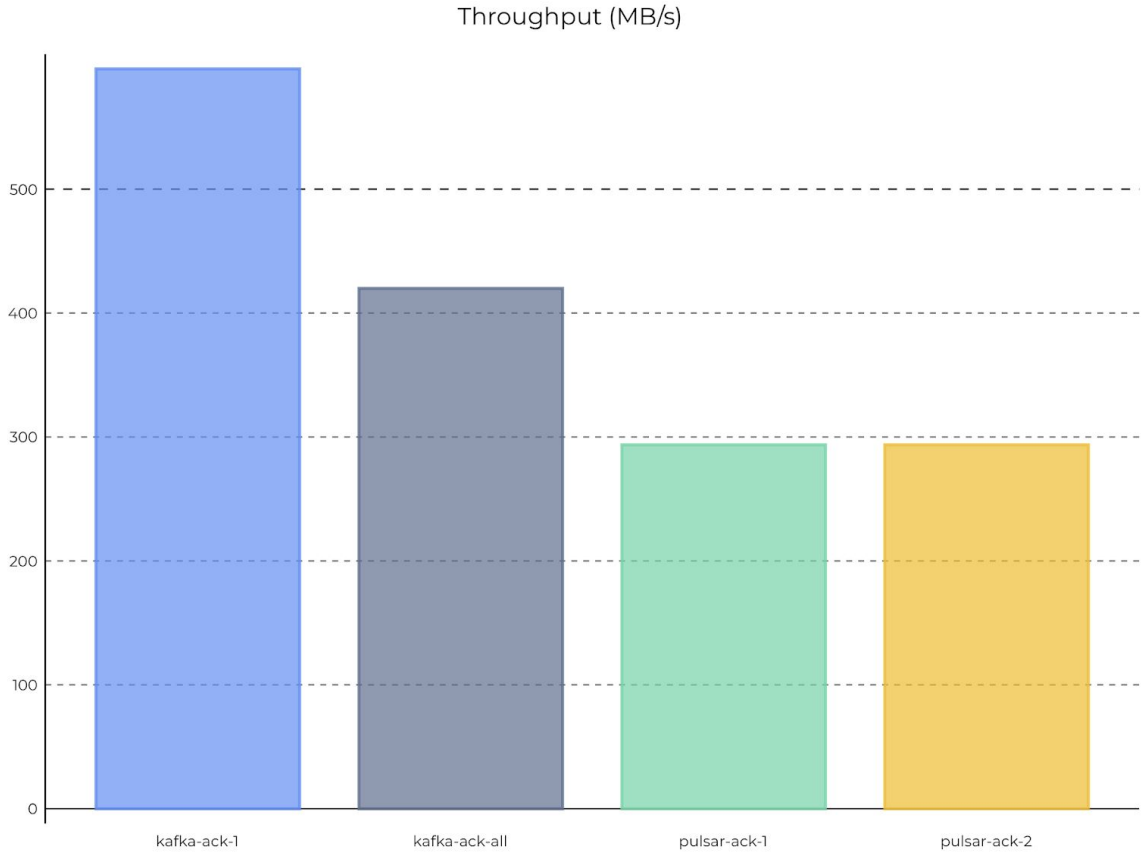

Figure 1 Maximum throughput with 100 partitions on Pulsar and Kafka (sync local durability)

Figure 2 shows the maximum throughput on Pulsar and Kafka with 100 partitions under async local durability.

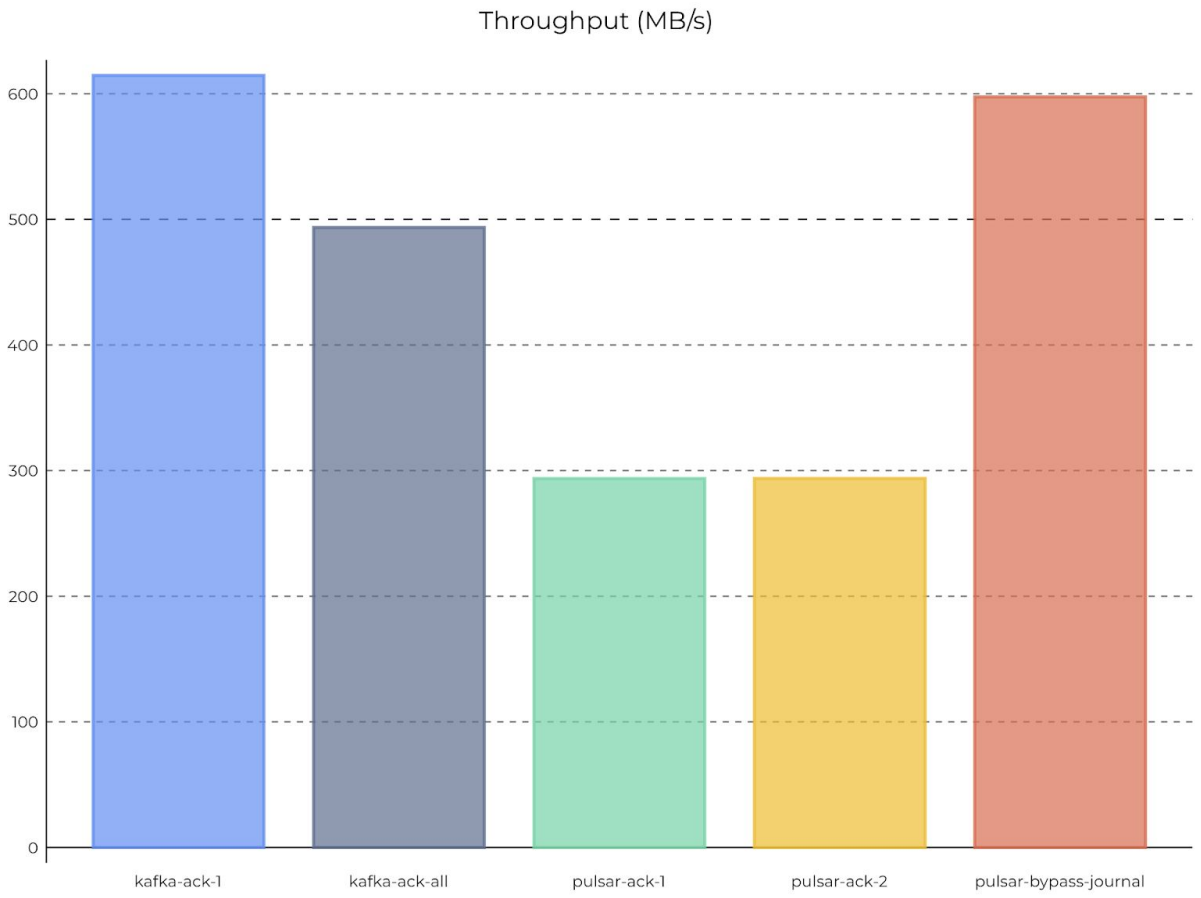

Figure 2 Maximum throughput with 100 partitions on Pulsar and Kafka (async local durability)

#2 2000 partitions, 1 subscription, 2 producers / 2 consumers

Our second test benchmarked maximum throughput using the same durability guarantees (acks = 2) on Pulsar and Kafka. However, we increased the number of partitions from 100 to 2000. We used one subscription, two producers, and two consumers. Our test results are described below.

- Pulsar's maximum throughput remained at ~300 MB/s under a Level-1 durability guarantee and increased to ~600 MB/s under Level-2 durability.

- Kafka's maximum throughput decreased from 600 MB/s (at 100 partitions) to ~300 MB/s when flushing data for each message individually (kafka-ack-all-sync).
- Kafka's maximum throughput decreased from ~500 MB/s (at 100 partitions) to ~300 MB/s when using the system's default durability settings (kafka-ack-all-nosync).

To understand why Kafka's throughput dropped, we plotted the average publish latency for each system under each durability guarantee tested. As you can see in Figure 3, when the number of partitions increased to 2000, Kafka's average publish latency increased to 200 ms and its 99th percentile publish latency increased to 1200 ms.

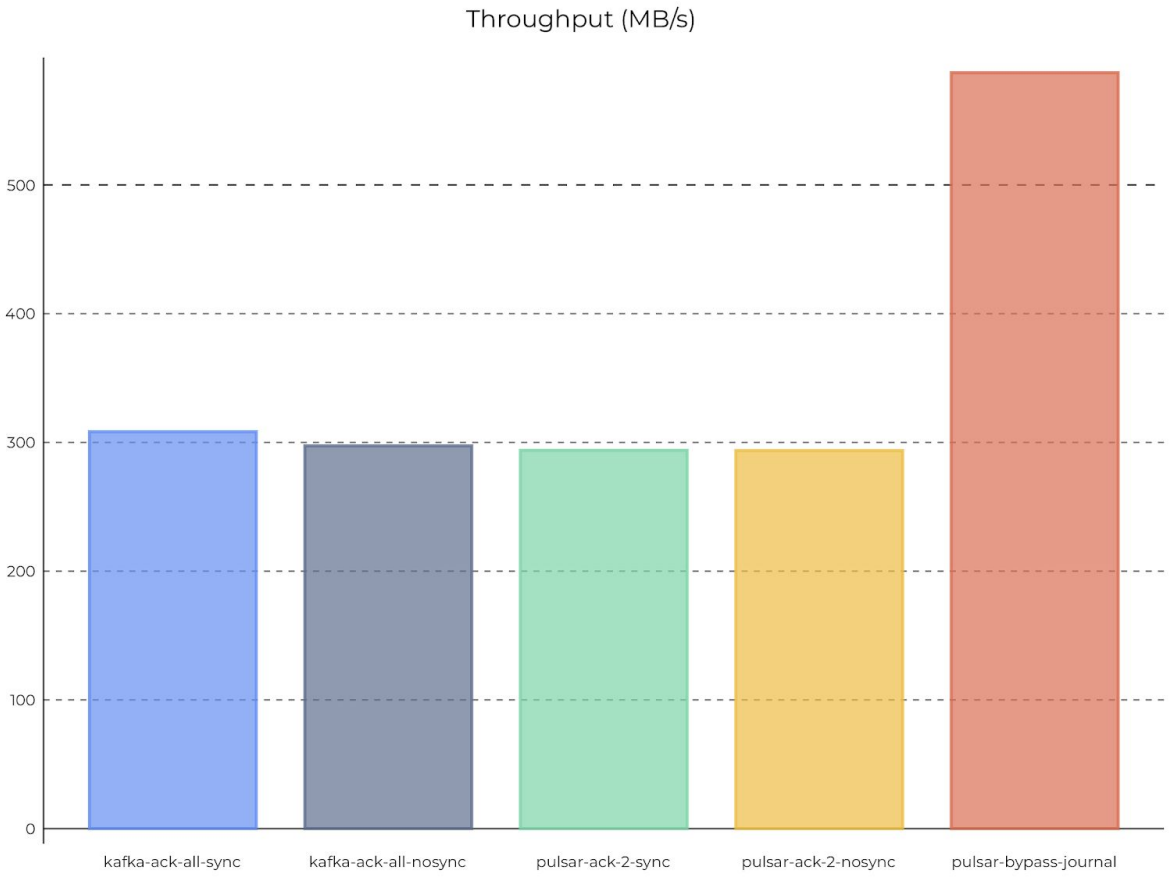

Figure 3 Maximum throughput with 2000 partitions on Pulsar and Kafka

Increased publish latency can significantly impact throughput. Latency did not affect throughput on Pulsar because Pulsar clients leverage Netty's

powerful asynchronous networking framework. However, latency did impact throughput on Kafka because Kafka clients use synchronous implementation. We were able to improve throughput on Kafka by doubling the number of producers. When we increased the number of producers to four, Kafka achieved a throughput of ~600 MB/s.

Figure 4 shows the publish latency for Pulsar and Kafka with 2000 partitions.

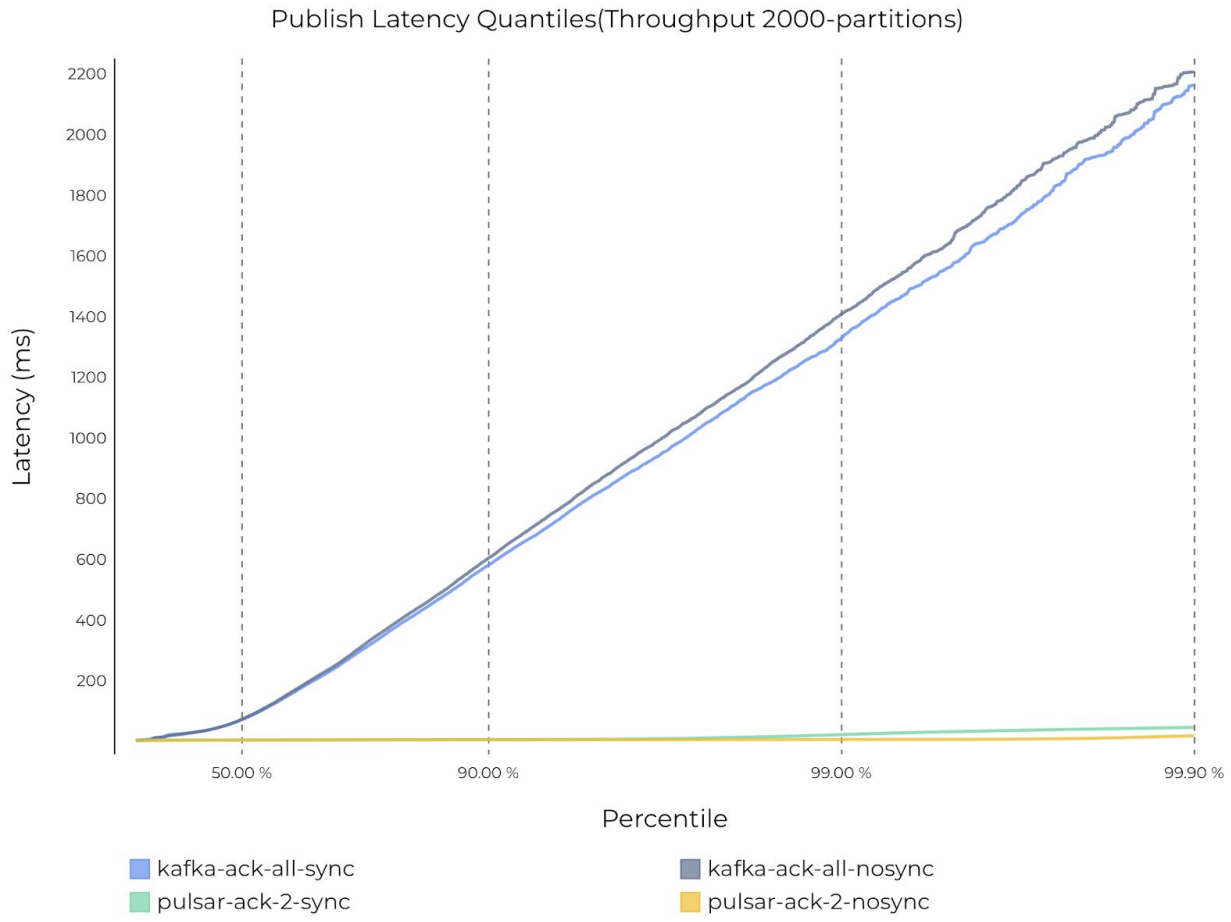

Figure 4 Publish latency with 2000 partitions on Pulsar and Kafka

#3 1 partition, 1 subscription, 2 producers / 2 consumers

Adding more brokers and partitions helps increase throughput on both Pulsar

and Kafka. To gain a better understanding of each system's efficiency, we benchmarked maximum throughput using only one partition. For this test, we used one subscription, two producers, and two consumers.

We observed the following:

- Pulsar achieved a maximum throughput of ~300 MB/s at all levels of durability.
- Kafka achieved a maximum throughput of ~300 MB/s under async replication durability, but only ~160 MB/s under sync replication durability.

Figure 5 shows the maximum throughput on Pulsar and Kafka with one partition under sync local durability.

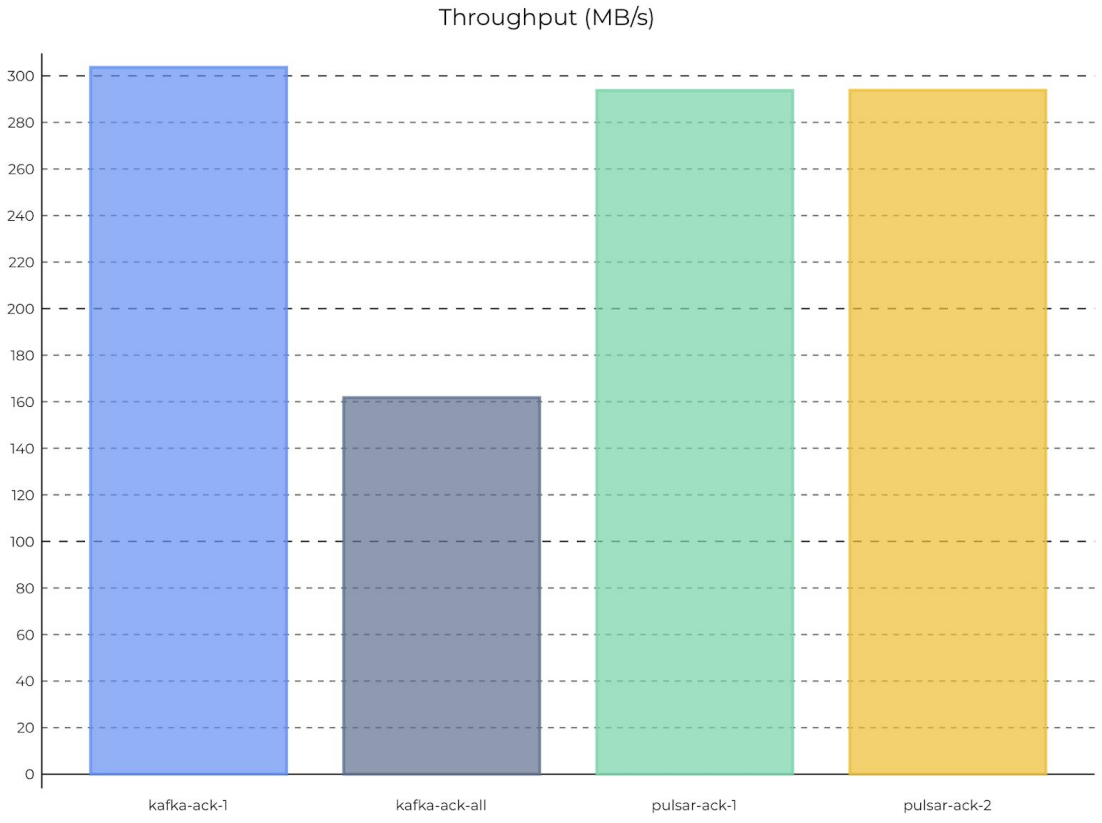

Figure 5Maximum throughput with 1 partition on Pulsar and Kafka (sync local durability)

Figure 6 shows the maximum throughput on Pulsar and Kafka with one partition under async local durability.

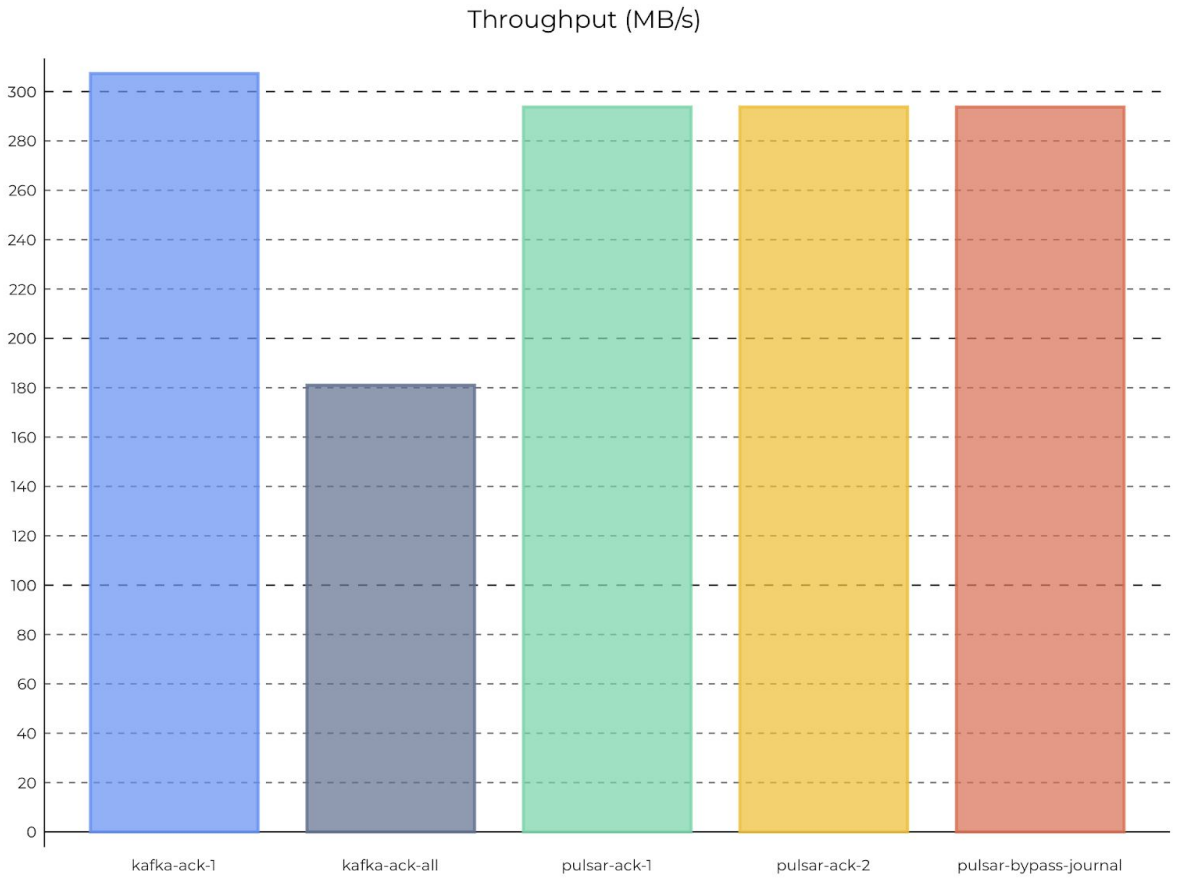

Figure 6 Maximum throughput with 1 partition on Pulsar and Kafka (async local durability)

#4 1 partition, 1 subscription, 1 producer / 1 consumer

We benchmarked maximum throughput on Pulsar and Kafka using only one partition and one subscription, as in the previous test. However, for this test, we used only one producer and one consumer (instead of two of each).

We observed the following:

- Pulsar sustained a maximum throughput of ~300 MB/s at all durability levels.
- Kafka's maximum throughput decreased from ~300 MB/s (in Test #3) to ~230 MB/s under async replication durability.
- Kafka's throughput was dropped from ~160 MB/s (in Test #3) to ~100 MB/s under sync replication durability.

Figure 7 shows the maximum throughput Pulsar and Kafka achieved with one partition, one producer, and one consumer under sync local durability.

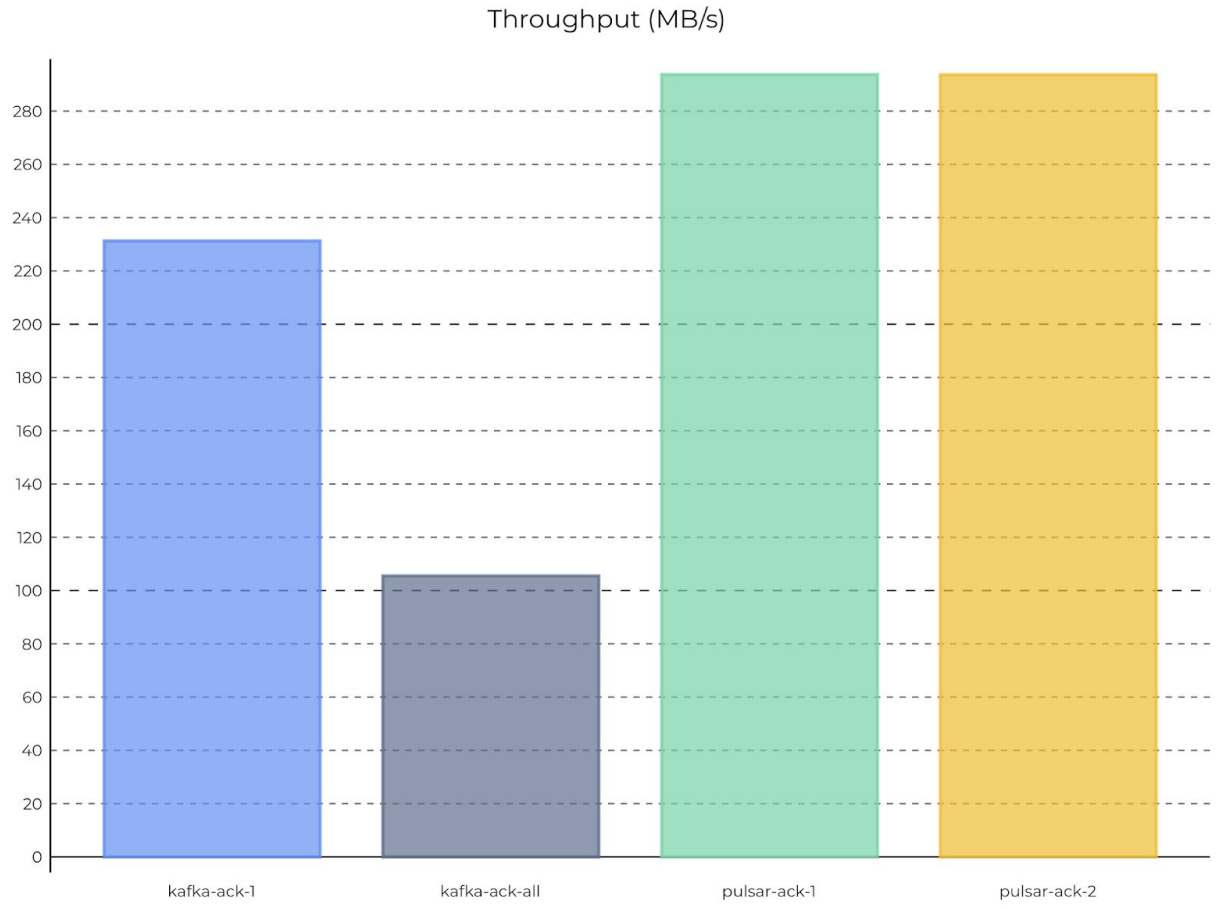

Figure 7 Maximum throughput with 1 partition, 1 producer, 1 consumer on Pulsar and Kafka (sync local durability)

To understand why Kafka's throughput dropped, we plotted the average publish latency (see Figure 8) and end-to-end latency (see Figure 9) for each system under different durability guarantees. As you can see from the graphics below, even with just one partition, Kafka's publish and end-to-end latency increased from single-digit values to multiple hundreds of milliseconds. Reducing the number of producers and consumers greatly impacted Kafka's throughput. In contrast, Pulsar consistently offered predictable low single-digit latency.

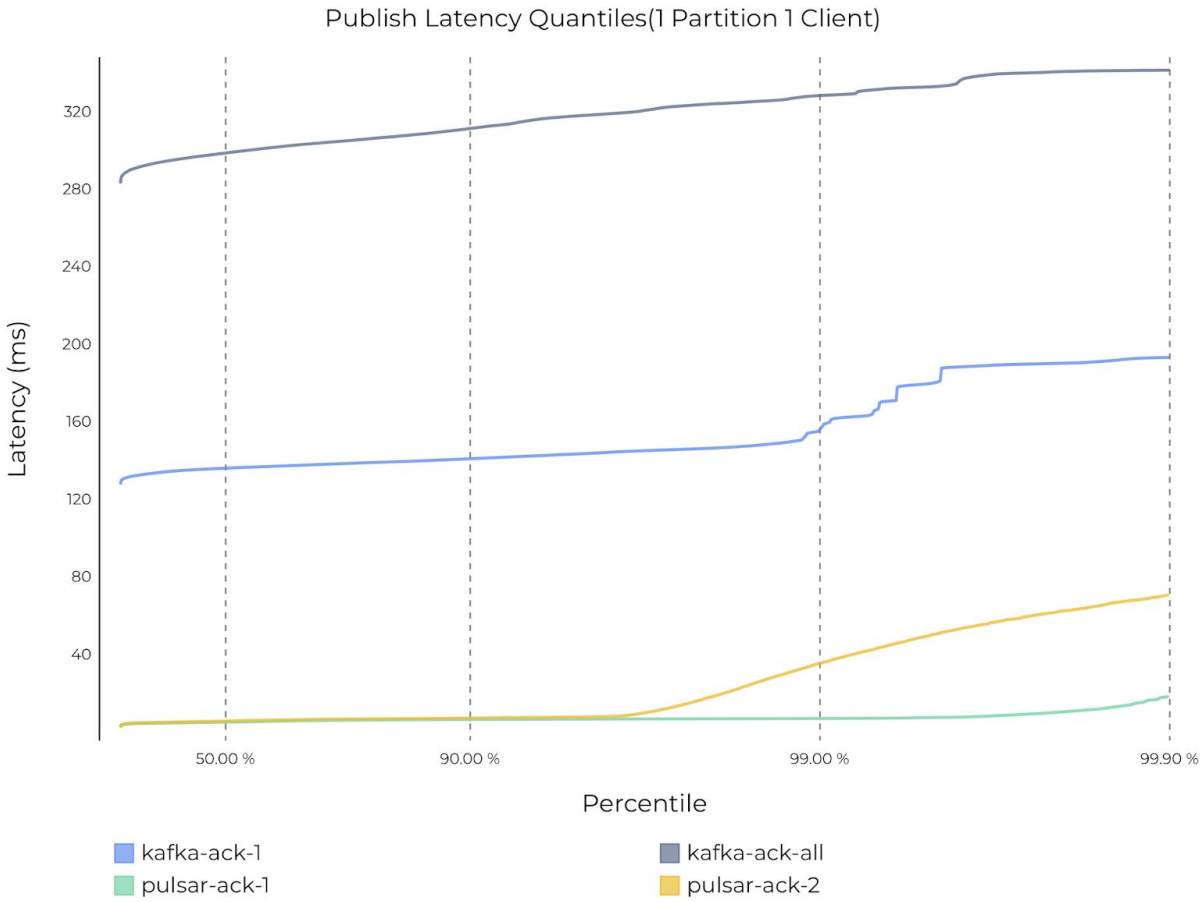

Figure 8 Publish latency with 1 partition and 1 producer and 1 consumer on Pulsar and Kafka (sync durability)

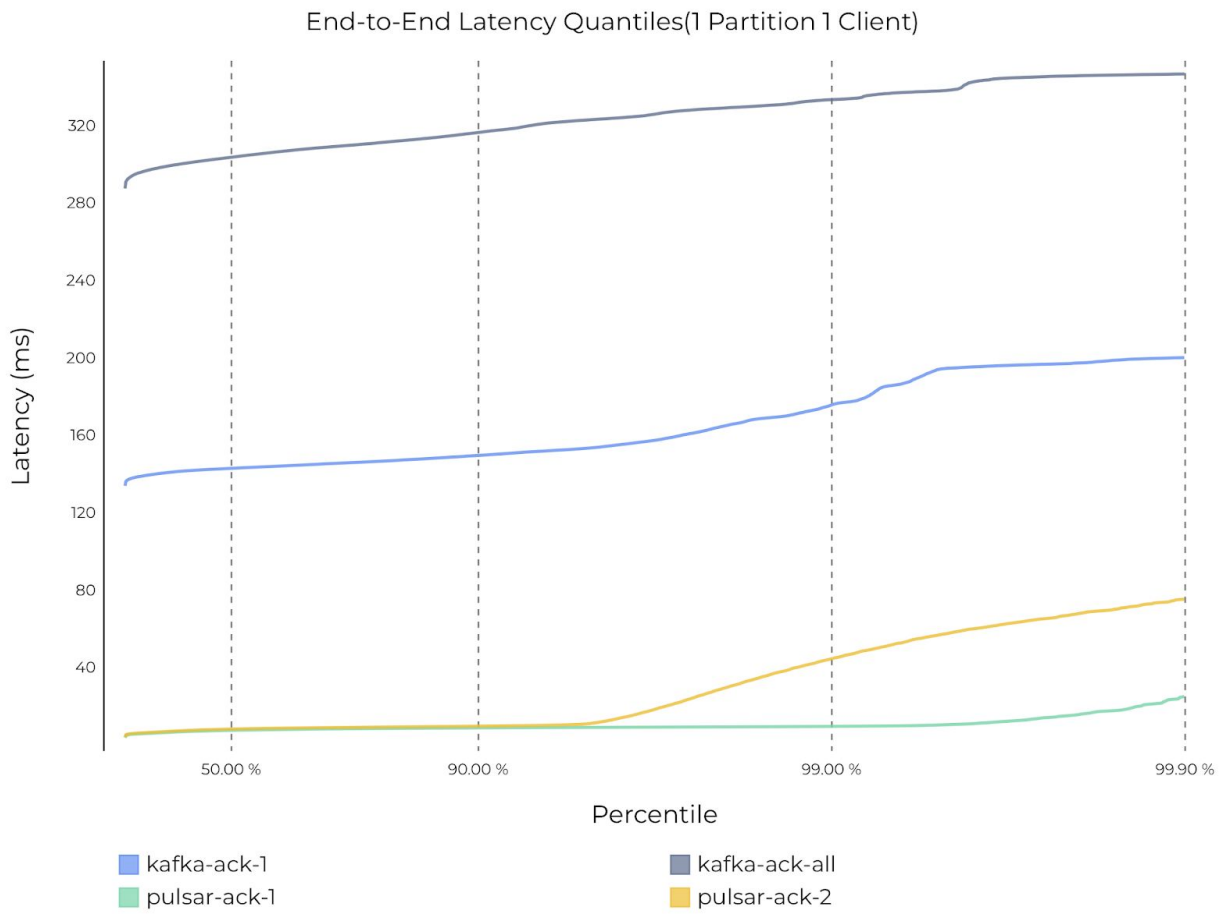

Figure 9 End-to-end latency with 1 partition, 1 producer, and 1 consumer on Pulsar and Kafka (sync durability)

Publish & End-to-End Latency Test

The following is the test setup.

The test was designed to determine the lowest latency each system can achieve when processing workloads that consist of publish and tailing-reads. We varied the number of subscriptions and the number of partitions to see how each change impacted both publish and end-to-end latency. Our test strategy included the following principles and expected guarantees:

- Each message was replicated three times to ensure fault tolerance.
- We varied the number of acknowledgments to measure variances in

throughput using different replication durability guarantees.
- We varied the number of subscriptions (from 1 to 10) to measure latency for each.
- We varied the number of partitions (from 100 to 10000) to measure latency for each.
- We used a message size of 1KB.
- The producer sent messages at a fixed rate of 200000/s (~200 MB/s) and the tailing-read consumers processed the messages while the producer continued to send them.

The following is the result for each test.

#1 100 partitions, 1 subscription

We started with 100 partitions and 1 subscription to benchmark the lowest latency Pulsar and Kafka can achieve under all different durability guarantees.

Our test showed Pulsar's publish and end-to-end latency to be two to five times lower than Kafka's at all levels of durability. You can see the actual test results in Table 1.

Table 2-1 Actual publish and end-to-end latency test results for Pulsar and Kafka by durability level

	Publish Latency	**End-to-End Latency**
Sync Local Durability	Results	Results
Async Local Durability	Results	Results

Publish Latency - Sync Local Durability

Figure 10 shows the differences in publish latency between Pulsar and Kafka using two replication durability settings (ack-1 and ack-2, respectively) and sync local durability. Table 2 shows the exact latency numbers for each case. As you can see, Pulsar's 99th percentile latency is

three times lower than Kafka's under async replication durability (ack-1) and five times lower under sync replication durability (ack-2).

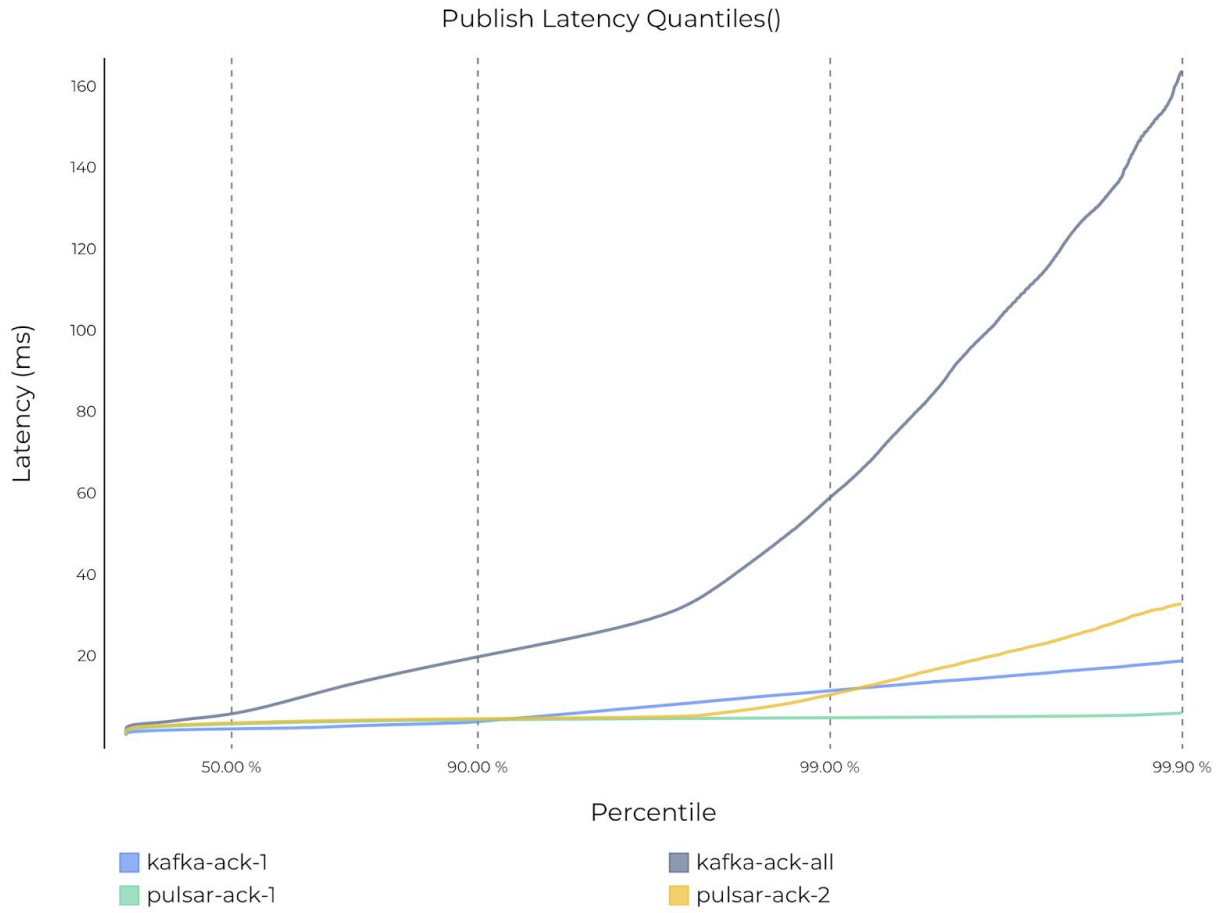

Figure 10 Publish latency on Pulsar and Kafka (with data sync)

Table 2-2 Actual publish latency test results on Pulsar and Kafka (with data sync)

	Average	**P50**	**P90**	**P99**	**P999**
pulsar-ack-1	*3.23*	*3.21*	*4.23*	*4.73*	*5.89*
pulsar-ack-2	*3.64*	*3.42*	*4.47*	*10.37*	*32.74*
kafka-ack-1	*2.54*	*1.99*	*3.78*	*11.37*	*18.71*
kafka-ack-all	*9.84*	*5.71*	*19.7*	*58.83*	*164.20*

To gain a better understanding of how latency changes over the time, we plotted the 99th percentile publish latency for Pulsar and Kafka using various replication durability settings. As you can see in Figure 11, Pulsar's

latency stayed consistent (~5 ms) but Kafka's latency was spiky. Stable and consistently low latency is crucial to mission-critical services.

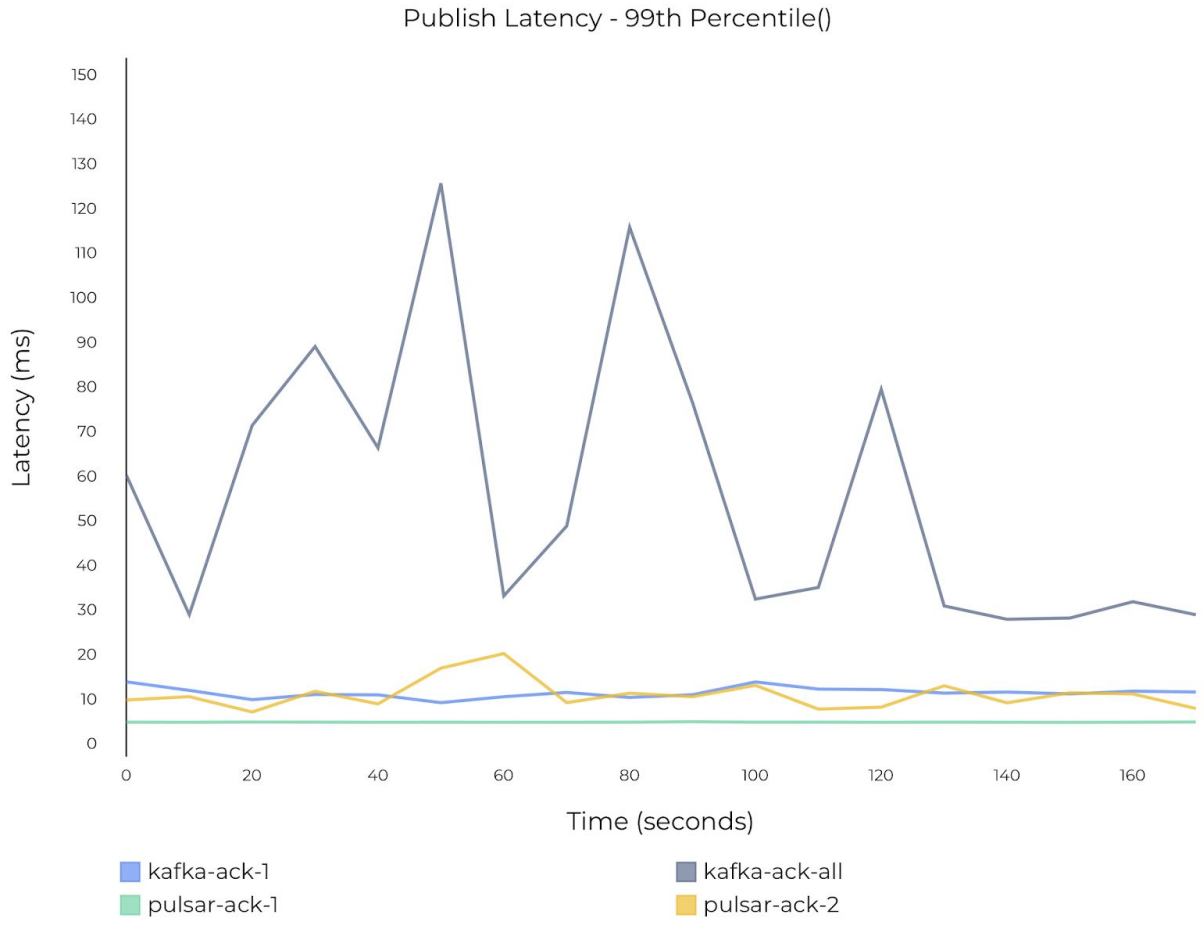

Figure 11 99th percentile publish latency on Pulsar and Kafka

End-to-End Latency - Sync Local Durability

Figure 12 shows the differences in end-to-end latency between Pulsar and Kafka using two replication durability settings (ack-1 and ack-2, respectively) and sync local durability. Table 3 shows the exact latency numbers for each case. As you can see, Pulsar's 99th percentile end-to-end latency was three
times lower than Kafka's under async replication durability (ack-1) and five times lower under sync replication durability (ack-2).

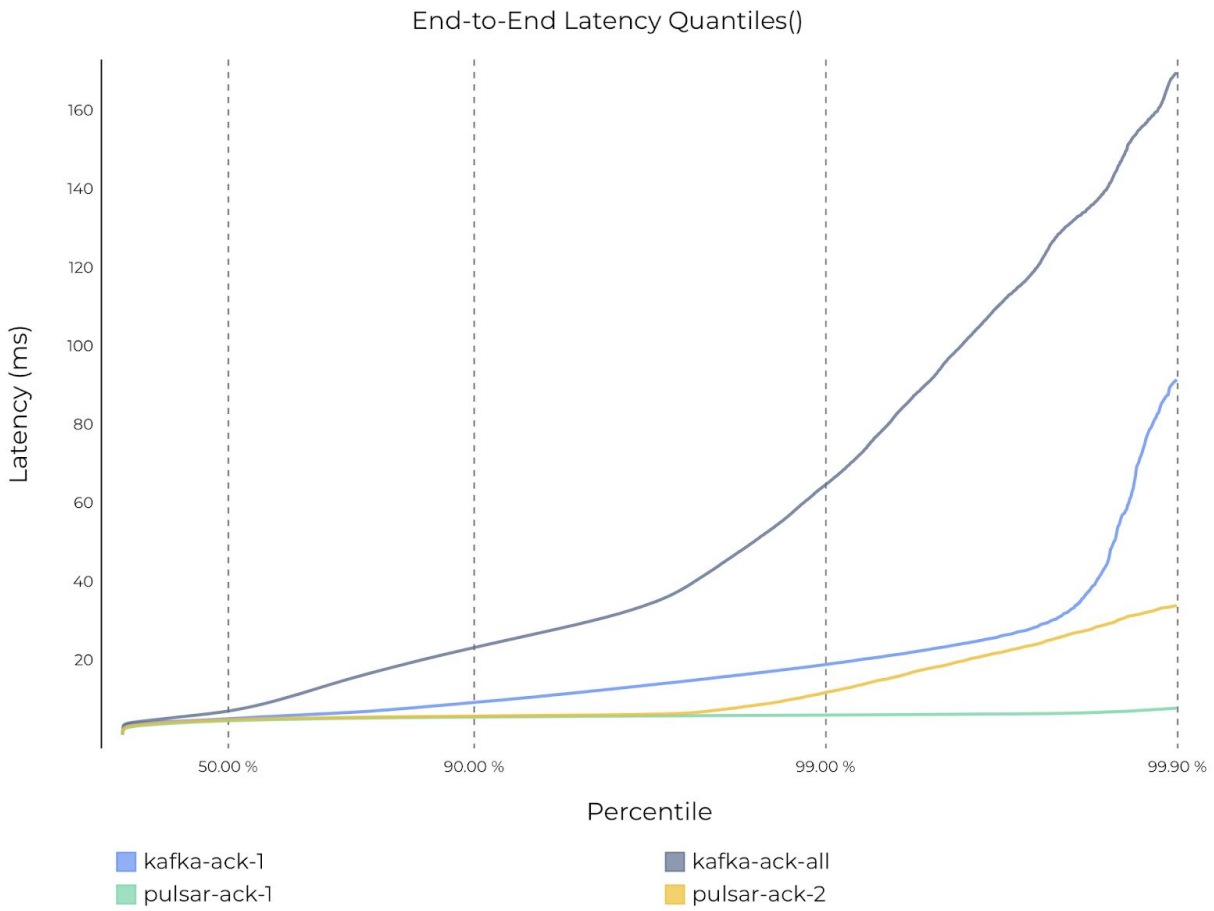

Figure 12 End-to-end latency with 1 subscription on Pulsar and Kafka (with data sync)

Table 2-3 Actual end-to-end latency test results with 1 subscription on Pulsar and Kafka (with data sync)

	Average	**P50**	**P90**	**P99**	**P999**
pulsar-ack-1	*4.32*	*4.41*	*5.35*	*5.86*	*7.65*
pulsar-ack-2	*4.72*	*4.65*	*5.60*	*11.64*	*33.90*
kafka-ack-1	*6.23*	*4.91*	*9.08*	*18.75*	*91.74*
kafka-ack-all	*12.89*	*7.53*	*23.07*	*64.62*	*169.83*

Publish Latency - Async Local Durability

Figure 13 shows the differences in publishing latency between Pulsar and Kafka using two replication durability settings (ack-1 and ack-2, respectively) and async local durability. Table 4 shows the exact latency

numbers for each case. As you can see, Kafka performed better in the async replication durability (ack-1) case. But Pulsar's 99th percentile publish latency stayed consistent (below 5 ms) and increasing the replication durability guarantee (from ack-1 to ack-2) did not impact latency. However, Kafka's 99th percentile publish latency with sync replication durability (ack-2) was three times higher than Pulsar's.

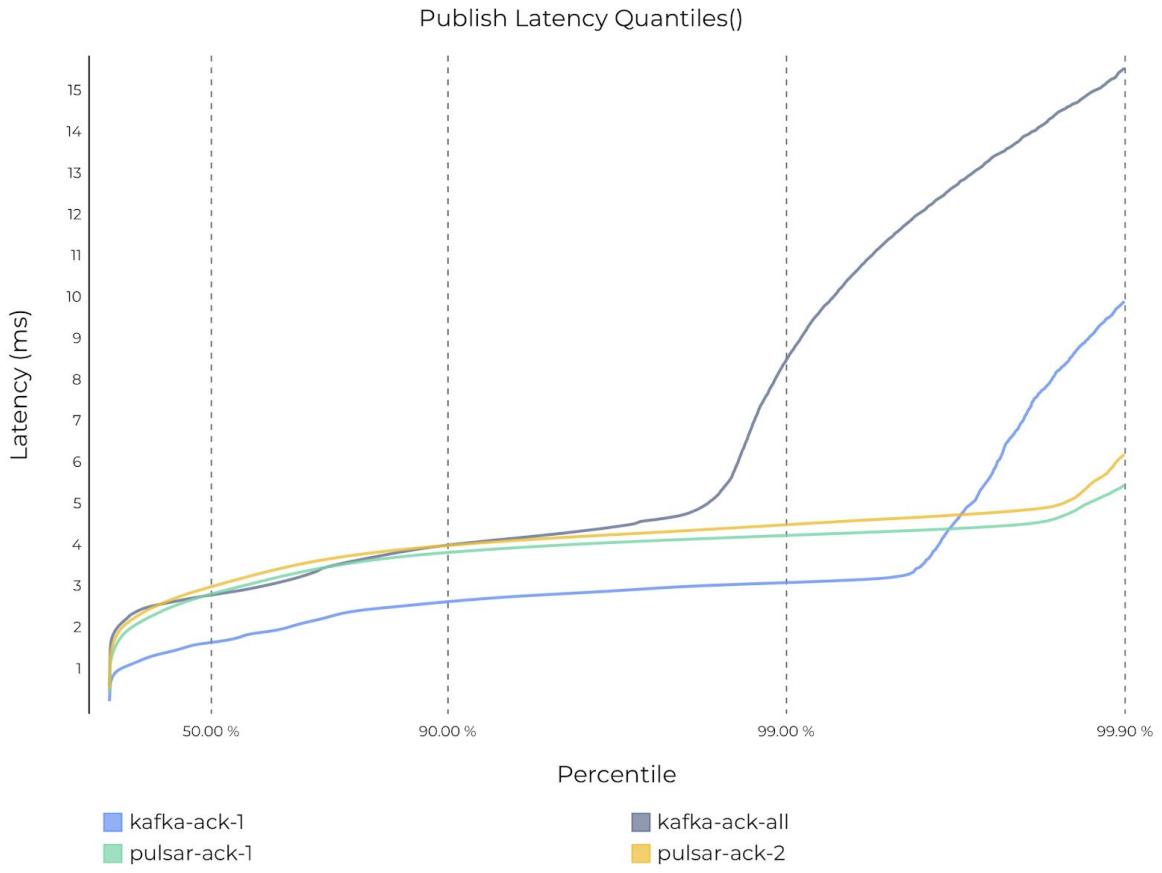

Figure 13 Publish latency on Pulsar and Kafka (without data sync)

Table 2-4 Actual publish latency test results on Pulsar and Kafka (without data sync)

	Average	**P50**	**P90**	**P99**	**P999**
pulsar-ack-1	*2.81*	*2.79*	*3.8*	*4.21*	*5.45*
pulsar-ack-2	*2.99*	*2.97*	*3.96*	*4.47*	*6.19*
kafka-ack-1	*1.74*	*1.62*	*2.60*	*3.06*	*9.91*
kafka-ack-all	*3.01*	*2.77*	*3.97*	*8.47*	*15.57*

To gain a better understanding of how publish latency changes over the time, we plotted the 99th percentile publish latency for Pulsar and Kafka under various replication durability settings. As you can see in Figure 14, Pulsar's latency stayed consistently low (below 5 ms) and Kafka's was consistently three times higher than Pulsar's.

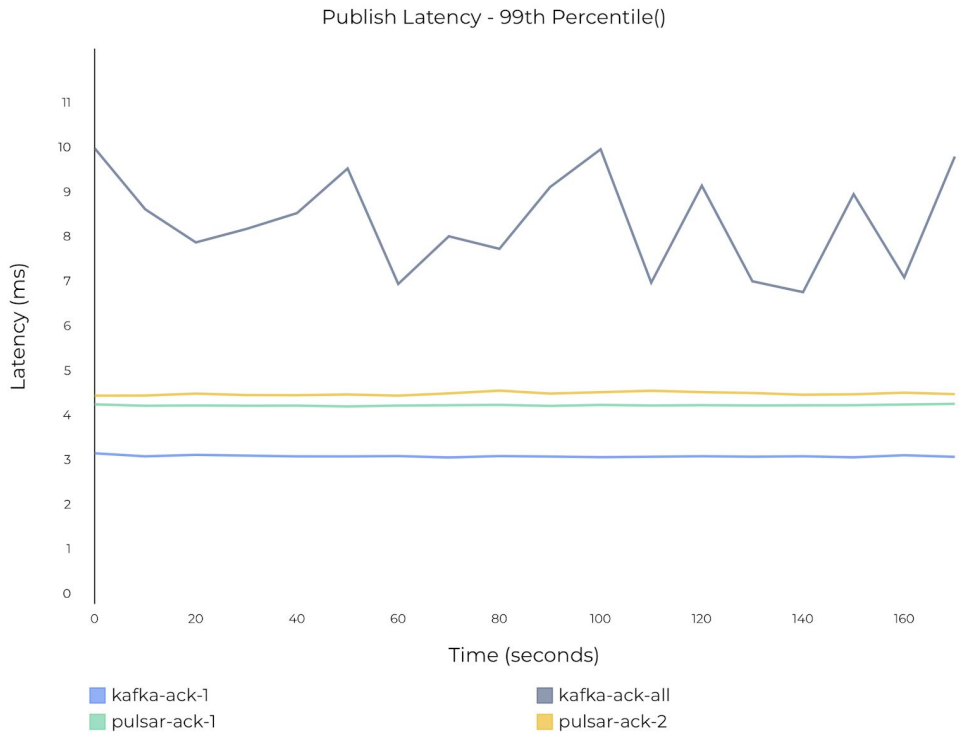

Figure 14 99th percentile publish latency on Pulsar and Kafka under various replication durability settings

End-to-End Latency - Async Local Durability

Figure 15 shows the differences in end-to-end latency between Pulsar and Kafka under two replication durability settings (ack-1 and ack-2, respectively) and async local durability. Table 5 shows the exact latency numbers for each case. As you can see, Pulsar performed consistently better than Kafka in all cases. Pulsar's 99th percentile end-to-end latency stayed consistent (~ 5 ms) and varying the replication durability setting had no impact. Kafka's 99th percentile end-to-end latency was 1.5 times higher than Pulsar's for ack-1 and 2 times higher for ack-2.

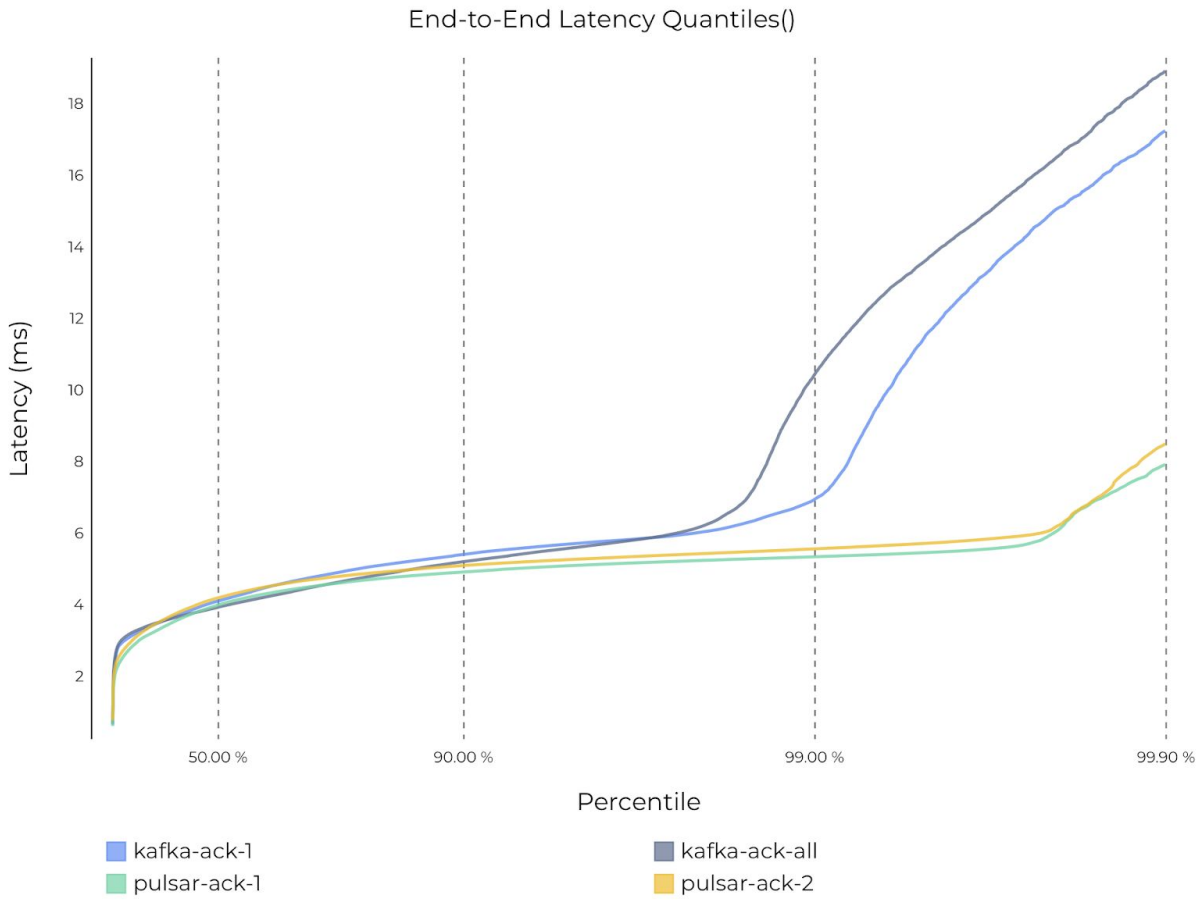

Figure 15 End-to-end latency with 1 subscription on Pulsar and Kafka (without data sync)

Table 2-5 Actual end-to-end latency test results with 1 subscription on Pulsar and Kafka (without data sync)

	Average	**P50**	**P90**	**P99**	**P999**
pulsar-ack-1	*3.96*	*3.99*	*4.90*	*5.33*	*7.93*
pulsar-ack-2	*4.06*	*4.17*	*5.08*	*5.55*	*8.52*
kafka-ack-1	*4.26*	*4.10*	*5.39*	*6.94*	*17.24*
kafka-ack-all	*4.22*	*3.96*	*5.19*	*10.43*	*18.95*

#2 100 partitions, 10 subscriptions

Once we understood how Pulsar and Kafka performed with just one subscription, we wanted to see how varying the number of subscriptions affected publish and end-to-end latency. So, we increased the number of subscriptions from 1 to 10 and assigned 2 consumers to each subscription.

As you can see from the details in Table 6, our test results showed the following:

- Pulsar's 99th percentile publish and end-to-end latency stayed between 5 and 10 ms.
- Kafka's 99th percentile publish and end-to-end latency were greatly impacted by increasing the number of subscriptions and went up to multiple seconds.

Table 2-6 Publish and end-to-end latency test results with 10 subscriptions

	Publish Latency	**End-to-End Latency**
Sync Local Durability	Results	Results
Async Local Durability	Results	Results

Publish Latency - Sync Local Durability

Figure 16 shows the differences in publish latency between Pulsar and Kafka under two replication durability settings (ack-1 and ack-2, respectively) and sync local durability. Table 7 shows the exact latency numbers for each case. As you can see, Pulsar's 99th percentile publish latency was still three times lower than Kafka's under async replication durability (ack-1). But under sync replication durability (ack-2), Pulsar's publish latency was 160 times lower than Kafka's (as compared to 5 times lower with only one subscription).

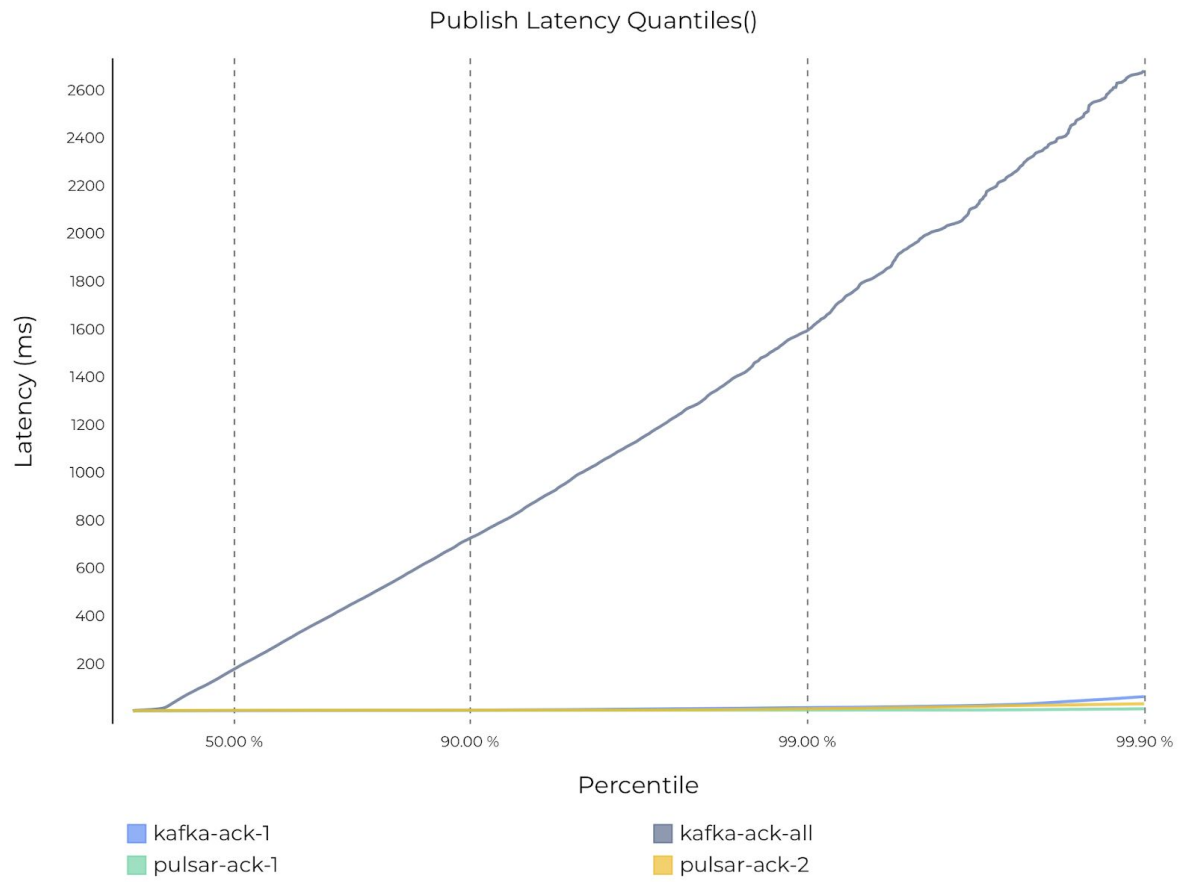

Figure 16 Publish latency with 10 subscriptions on Pulsar and Kafka (with data sync)

Table 2-7 Actual publish latency test results with 10 subscriptions on Pulsar and Kafka (with data sync)

	Average	**P50**	**P90**	**P99**	**P999**
pulsar-ack-1	*3.24*	*3.20*	*4.26*	*4.89*	*10.31*
pulsar-ack-2	*3.67*	*3.47*	*4.56*	*9.94*	*31.31*
kafka-ack-1	*3.14*	*2.39*	*4.39*	*15.07*	*61.29*
kafka-ack-all	*290.51*	*176.82*	*724.26*	*1593.46*	*2686.41*

End-to-End Latency - Sync Local Durability

Figure 17 shows the differences in end-to-end latency between Pulsar and Kafka under two replication durability settings (ack-1 and ack-2, respectively) and sync local durability. Table 8 shows the exact latency

numbers for each case. As you can see, Pulsar's 99th percentile latency was 20 times lower than Kafka's under async replication durability (ack-1) and 110 times lower under sync replication durability (ack-2).

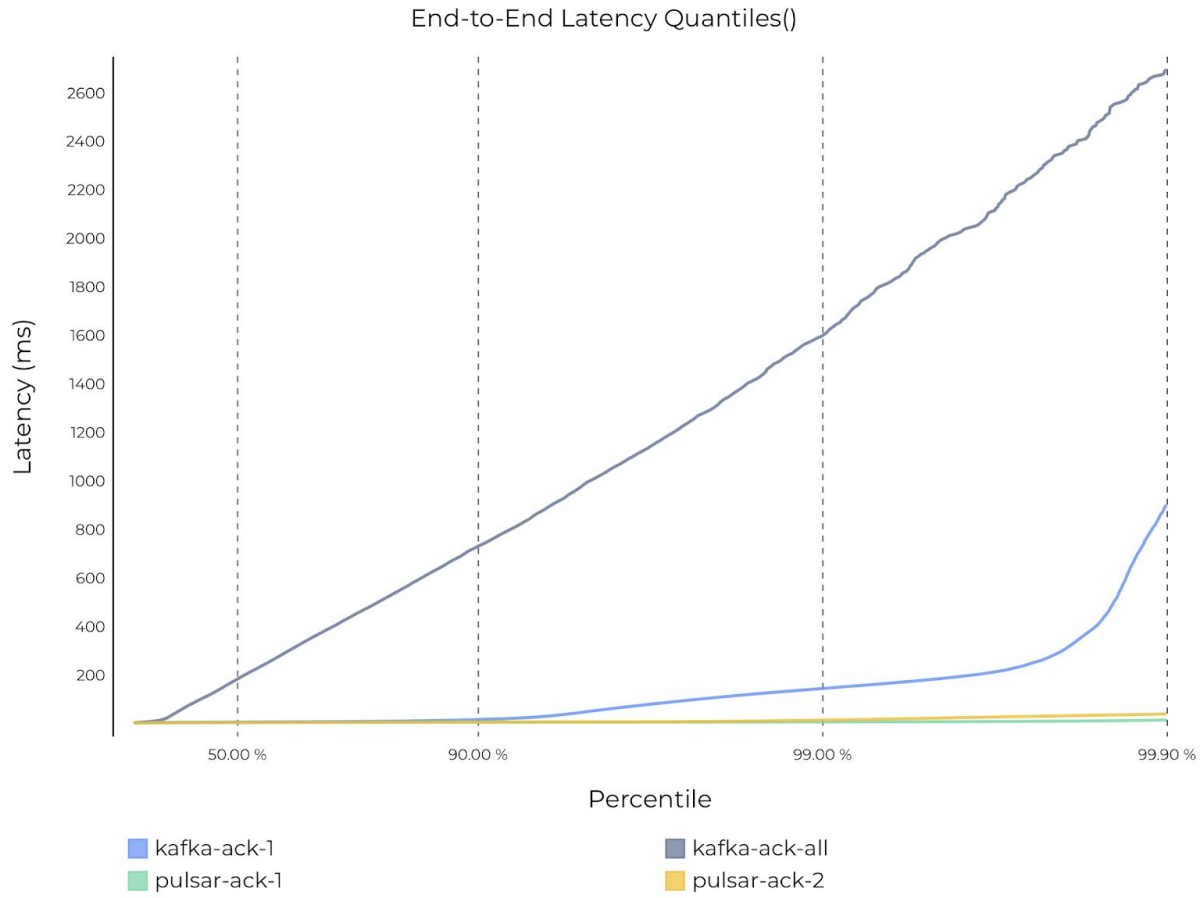

Figure 17 End-to-end latency with 10 subscriptions on Pulsar and Kafka (with data sync)

Table 2-8 Actual end-to-end latency test results with 10 subscriptions on Pulsar and Kafka (with data sync)

	Average	**P50**	**P90**	**P99**	**P999**
pulsar-ack-1	*4.79*	*4.83*	*6.03*	*7.12*	*15.36*
pulsar-ack-2	*5.34*	*5.12*	*6.43*	*14.65*	*39.90*
kafka-ack-1	*11.36*	*6.65*	*17.12*	*145.10*	*914.19*
kafka-ack-all	*296.45*	*171.32*	*731.67*	*1599.79*	*2696.63*

Publish Latency - Async Local Durability

Figure 18 shows the differences in publish latency between Pulsar and Kafka under two replication durability settings (ack-1 and ack-2, respectively) and async local durability. Table 9 shows the exact latency numbers for each case. As you can see, Pulsar outperformed Kafka significantly. Pulsar's average publish latency was ~3 ms and its 99th percentile latency was within 5 ms. Kafka's performance was satisfactory under async replication durability (ack-1), but significantly worse under sync replication durability (ack-2). Kafka's's 99th percentile publish latency under sync replication durability was 270 times higher than Pulsar's.

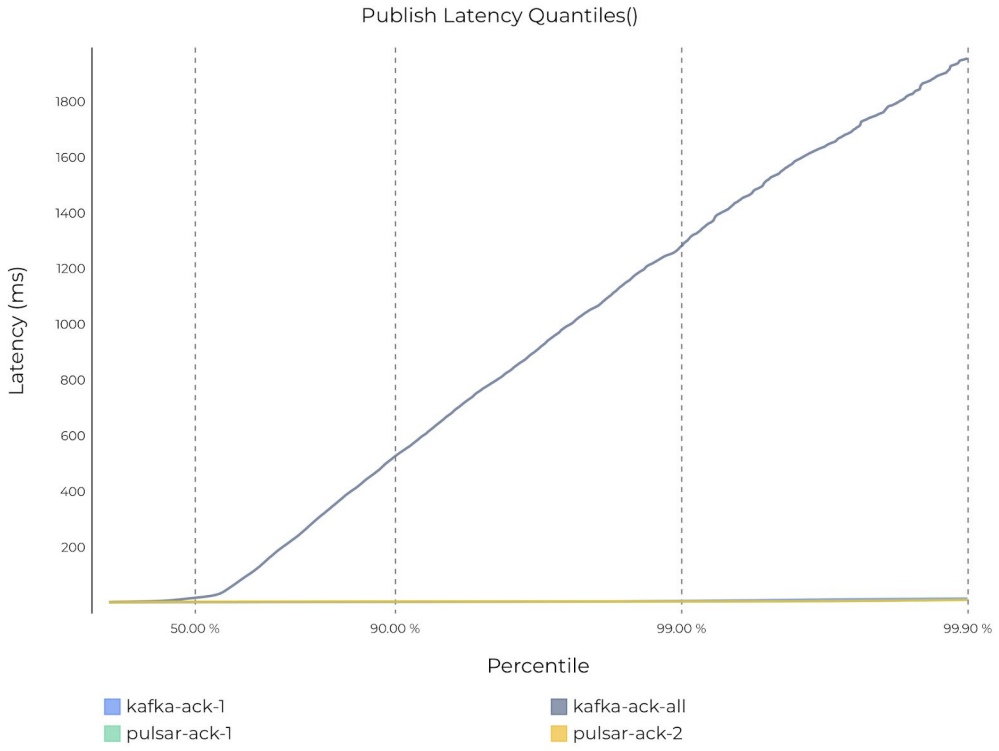

Figure 18 Publish latency with 10 subscriptions on Pulsar and Kafka (without data sync)

Table 2-9 Actual publish latency test results with 10 subscriptions on Pulsar and Kafka (without data sync)

	Average	**P50**	**P90**	**P99**	**P999**
pulsar-ack-1	*2.86*	*2.82*	*3.86*	*4.46*	*11.18*
pulsar-ack-2	*3.05*	*3.00*	*4.03*	*4.73*	*10.39*

kafka-ack-1	*2.11*	*1.89*	*3.02*	*6.35*	*14.74*
kafka-ack-all	*158.04*	*17.63*	*526.91*	*1281.25*	*1956.71*

End-to-End Latency - Async Local Durability

Figure 19 shows the differences in end-to-end latency between Pulsar and Kafka under two replication durability settings (ack-1 and ack-2, respectively) and async local durability. Table 10 shows the exact latency numbers for different cases. As you can see, Pulsar performed consistently better than Kafka in all cases. Pulsar's end-to-end latency consistently stayed between 4 and 7 ms and varying the replication durability setting had no impact. Kafka's 99th percentile end-to-end latency was 13 times higher than Pulsar's for ack-1 and 187 times higher for ack-2.

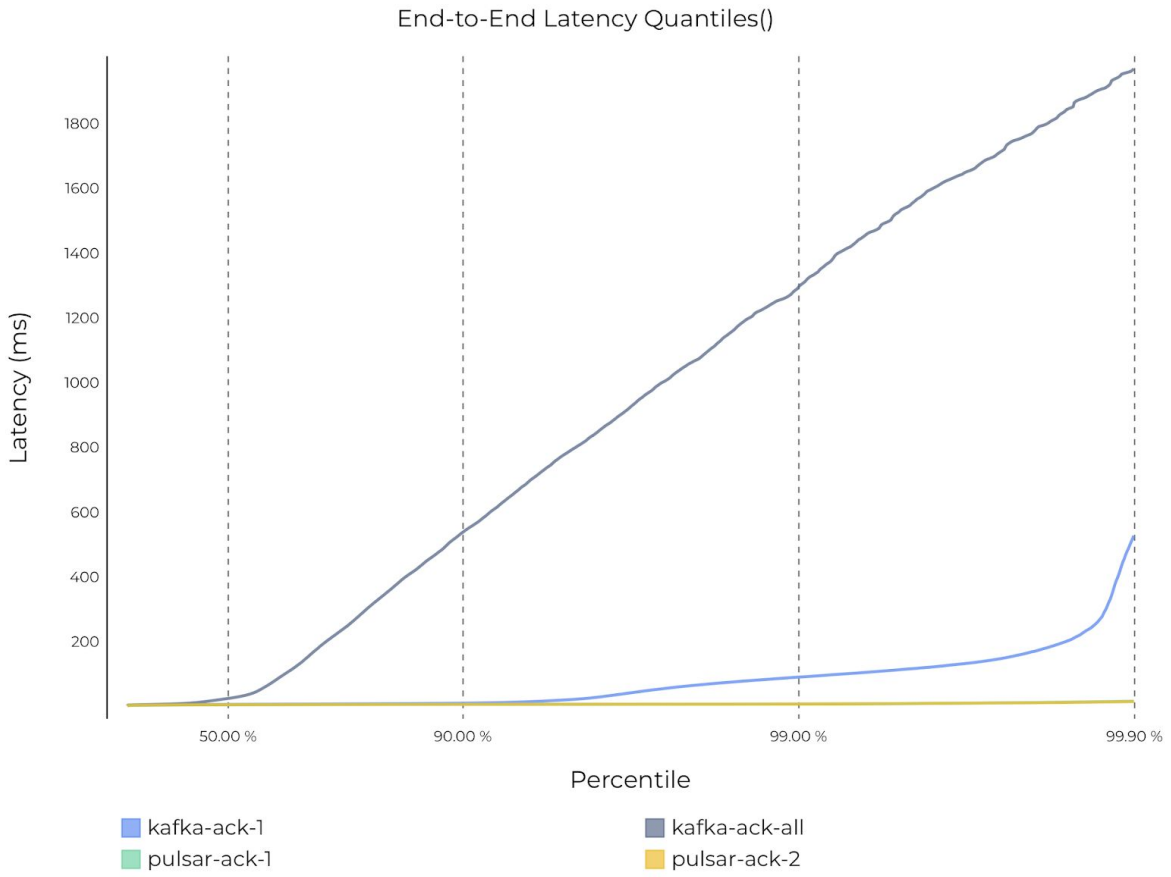

Figure 19 End-to-end latency with 10 subscriptions on Pulsar and Kafka (without data sync)

Table 2-10 Actual end-to-end latency test results with 10 subscriptions (without data sync)

	Average	P50	P90	P99	P999
pulsar-ack-1	*4.51*	*4.47*	*5.60*	*6.84*	*15.77*
pulsar-ack-2	*4.61*	*4.61*	*5.76*	*6.94*	*14.21*
kafka-ack-1	*8.01*	*5.90*	*9.38*	*89.80*	*532.68*
kafka-ack-all	*212.77*	*87.72*	*537.85*	*1295.78*	*1971.03*

#3 100, 5000, 8000, 10000 partitions

Having learned how varying the number of subscriptions affects publish latency in both Pulsar and Kafka, we wanted to vary the number of partitions
and observe the effects. So, we increased the number of partitions in increments from 100 to 10000 and looked for changes.

As you can see from the details in Table 11, our test results showed the following:

- Pulsar's 99th percentile publish latency remained stable at ~5 ms when the number of partitions increased.
- Kafka's 99th percentile publish latency was greatly impacted by incremental increases in the number of partitions and went up to multiple seconds.
- When the number of partitions exceeded 5000, Kafka's consumer was unable to keep up with the publish throughput.

Table 2-11 Actual publish latency test results with varying acks and durability

	Ack = 1	Ack = 2
Sync Local Durability	Results	Results
Async Local Durability	Results	Results

Ack = 1, Sync local durability

Figure 20 and Figure 21 show the differences in publish and end-to-end latency, respectively, between Pulsar and Kafka when varying the number of partitions under sync local durability and async replication durability (ack = 1).

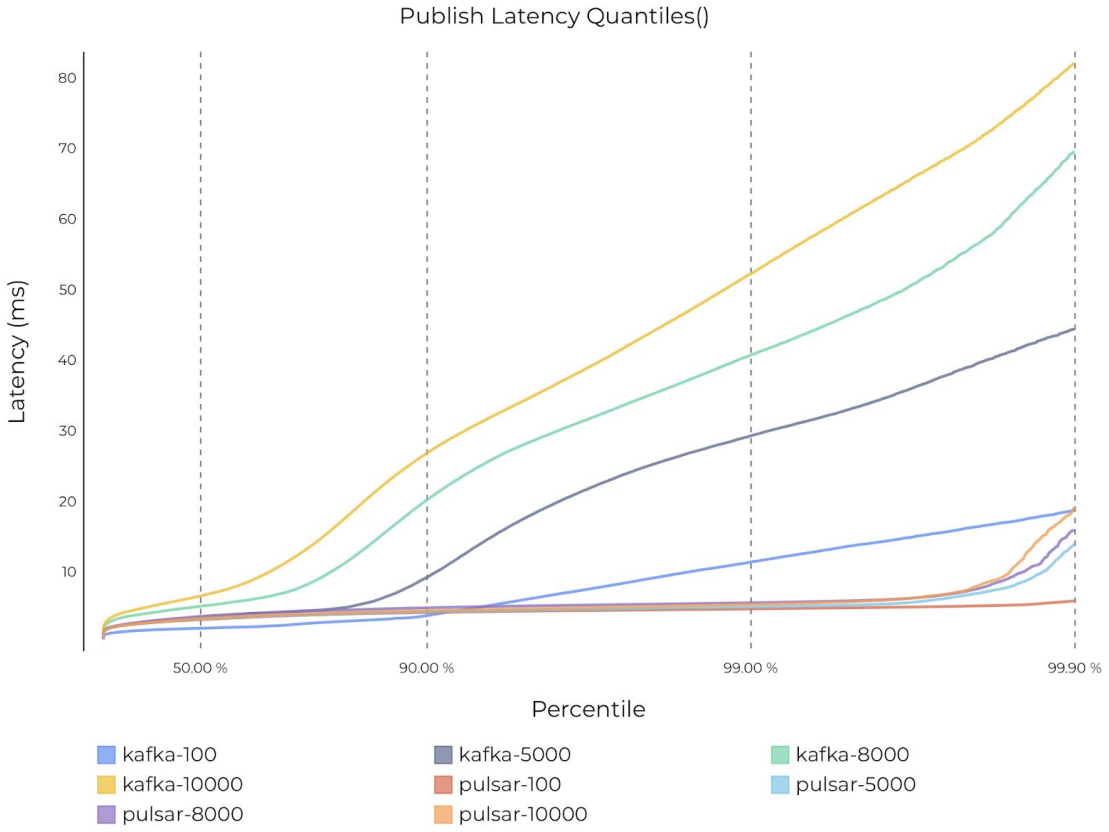

Figure 20 Publish latency with varying numbers of partitions and 1 ack (with data sync)

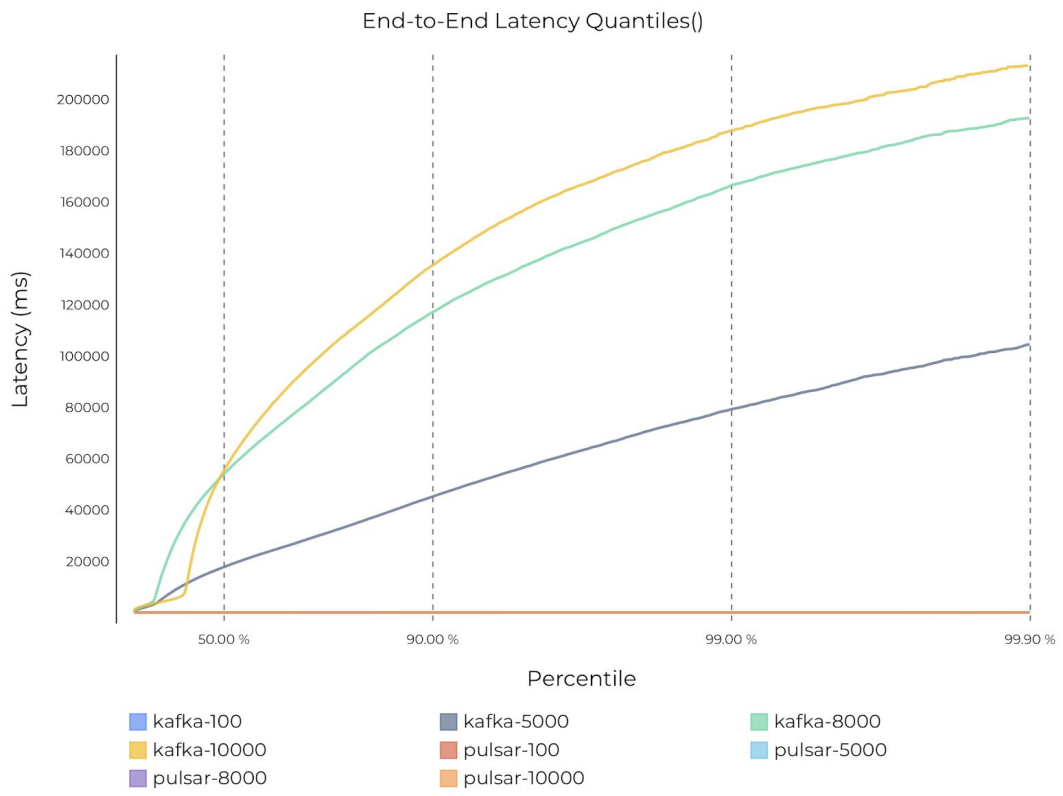

Figure 21 End-to-end latency with varying numbers of partitions and 1 ack (with data sync)

Table 12 shows our actual publish latency test results with varying numbers of partitions and one acknowledgement. Table 13 shows our actual end-to-end latency test results with varying numbers of partitions and one acknowledgement.

Table 2-12 Actual publish latency test results with varying numbers of partitions and 1 ack (with data sync)

	Average	**P50**	**P90**	**P99**	**P999**
kafka-100	*2.54*	*1.99*	*3.78*	*11.37*	*18.71*
kafka-5000	*3.50*	*4.41*	*9.21*	*29.26*	*44.39*
kafka-8000	*8.37*	*5.11*	*20.18*	*40.70*	*69.72*
kafka-10000	*11.14*	*6.57*	*26.81*	*52.24*	*82.07*
pulsar-100	*3.23*	*3.21*	*4.23*	*4.73*	*5.89*
pulsar-5000	*3.35*	*3.30*	*4.34*	*5.03*	*13.96*
pulsar-8000	*3.67*	*3.67*	*4.89*	*5.61*	*16.07*

pulsar-10000	*3.42*	*3.37*	*4.48*	*5.36*	*19.20*

Table 2-13 Actual end-to-end latency test results with varying numbers of partitions and 1 ack (with data sync)

	Average	**P50**	**P90**	**P99**	**P999**
kafka-100	*6.23*	*4.91*	*9.08*	*18.75*	*91.74*
pulsar-100	*4.32*	*4.41*	*5.35*	*5.86*	*7.65*
pulsar-5000	*4.52*	*4.53*	*5.55*	*6.26*	*17.78*
pulsar-8000	*4.89*	*4.99*	*6.11*	*6.86*	*23.83*
pulsar-10000	*4.49*	*4.62*	*5.70*	*6.67*	*27.25*

Figure 22 shows end-to-end latency with varying numbers of partitions and one acknowledgment on Pulsar. Figure 23 shows end-to-end latency with varying numbers of partitions and one acknowledgement on Kafka.

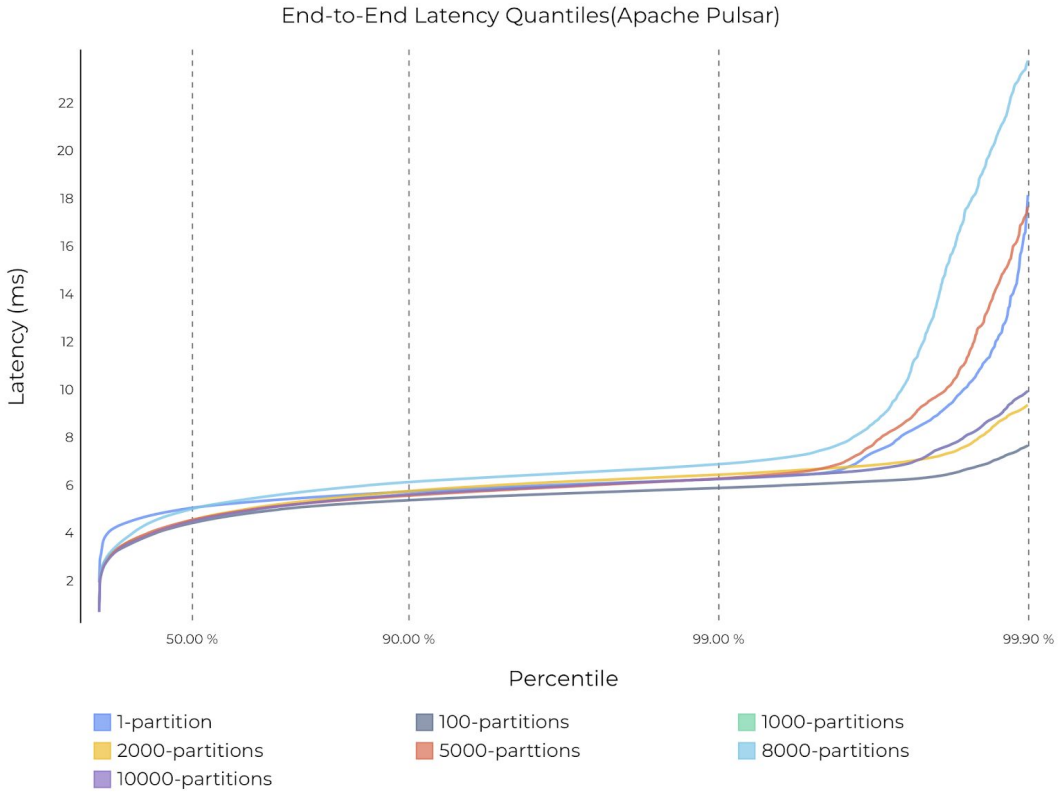

Figure 22 End-to-end latency with varying numbers of partitions and 1 ack on Pulsar (with data sync)

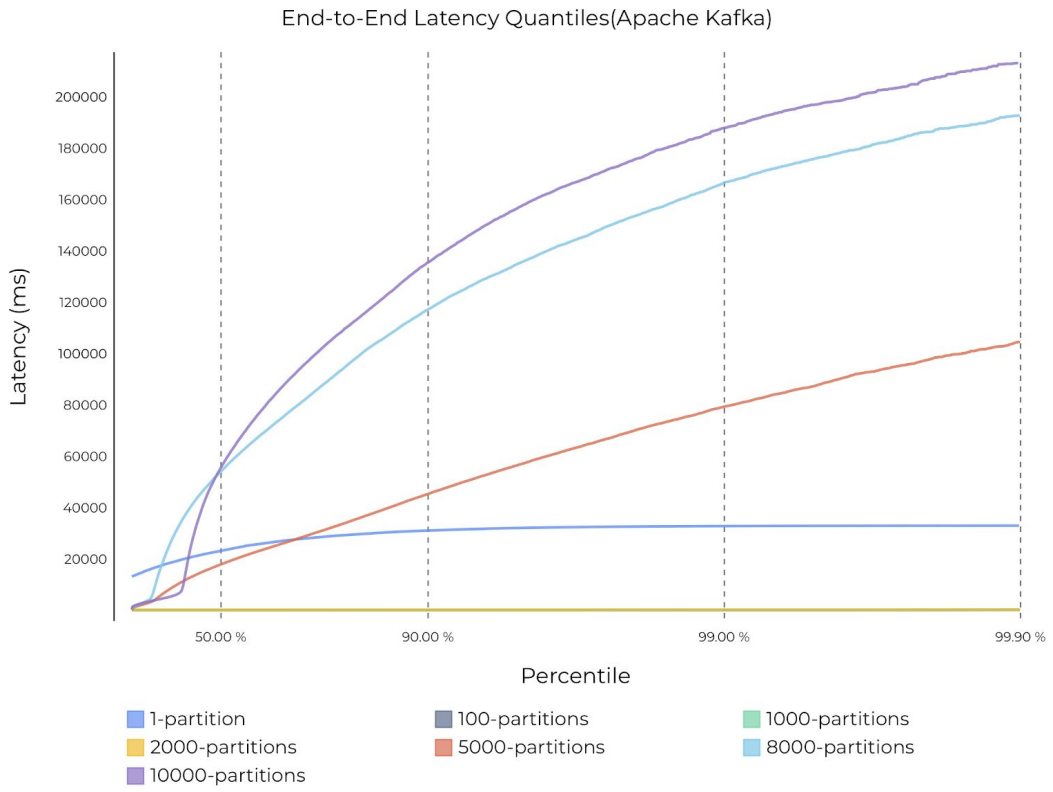

Figure 23 End-to-end latency with varying numbers of partitions and 1 ack on Kafka (with data sync)

As you can see from the figures and tables above,

- Pulsar's 99th percentile publish latency remained stable at ~5 ms. Varying the number of partitions had no effect.
- Pulsar's 99th percentile end-to-end latency remained stable at ~6 ms. Varying the number of partitions had no effect.
- Kafka's 99th percentile publish latency degraded incrementally as the number of partitions increased and was 5 times higher at 10000 partitions (as compared to 100). This was 10 times higher than Pulsar's.
- Kafka's 99th percentile end-to-end latency degraded incrementally as the number of partitions increased and was 10000 times higher at 10000 partitions (as compared to 100). Kafka's 99th percentile end-to-end latency at 10000 partitions increased to 180 s and was 280000 times higher than Pulsar's.

Ack = 2, Sync local durability

Figure 24 shows the differences in publish latency between Pulsar and Kafka when varying the number of partitions under sync local durability and sync replication durability (ack = 2). Table 14 shows the exact latency numbers for each case.

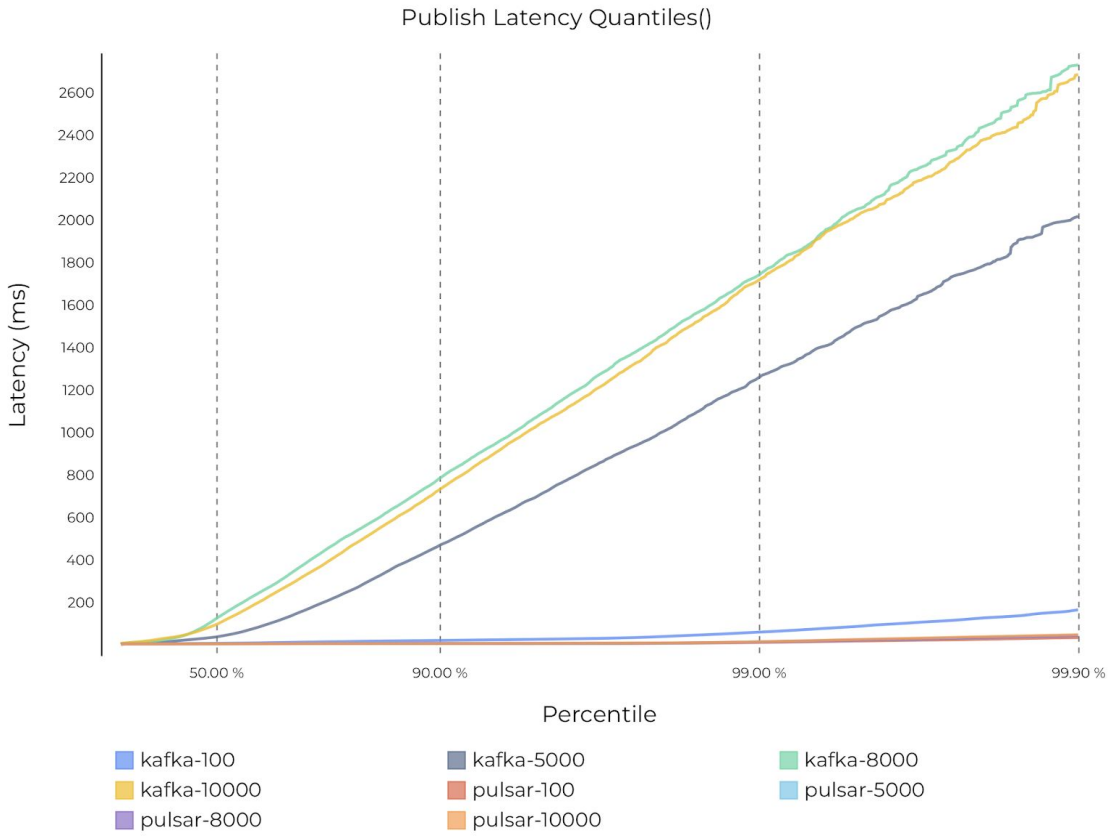

Figure 24 Publish latency with varying numbers of partitions and all/2 ack (with data sync)

Table 2-14 Actual publish latency test results with varying numbers of partitions and all/2 ack (with data sync)

	Average	**P50**	**P90**	**P99**	**P999**
kafka-100	*9.84*	*5.71*	*19.7*	*58.83*	*164.20*
kafka-5000	*154.50*	*36.86*	*468.75*	*1259.82*	*2017.15*
kafka-8000	*283.50*	*124.82*	*784.69*	*1742.27*	*2729.79*
kafka-10000	*259.50*	*96.15*	*731.85*	*1718.09*	*2684.28*
pulsar-100	*3.64*	*3.42*	*4.47*	*10.37*	*32.74*

pulsar-5000	*3.84*	*3.57*	*4.69*	*12.81*	*42.83*
pulsar-8000	*4.15*	*3.97*	*5.23*	*11.90*	*42.19*
pulsar-10000	*4.04*	*3.71*	*5.03*	*13.54*	*46.73*

Figure 25 and Figure 26 show how varying the number of partitions affects end-to-end latency in Pulsar and Kafka, respectively.

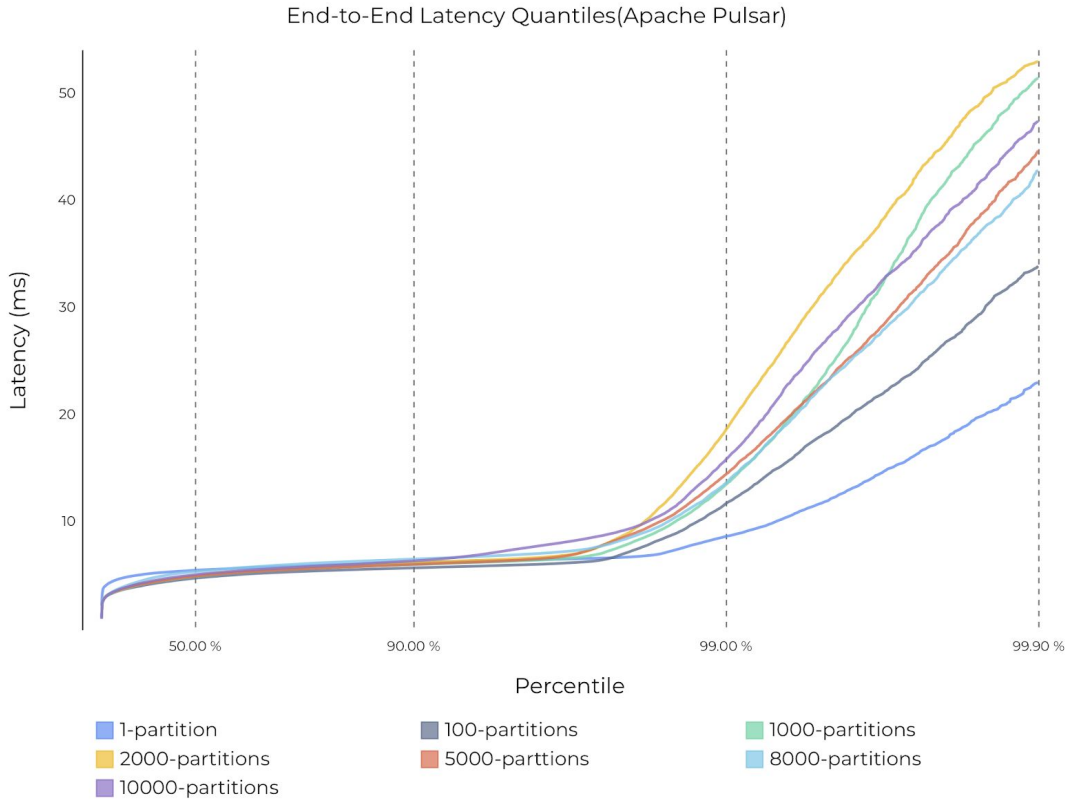

Figure 25 End-to-end latency with varying numbers of partitions and 2 acks on Pulsar (with data sync)

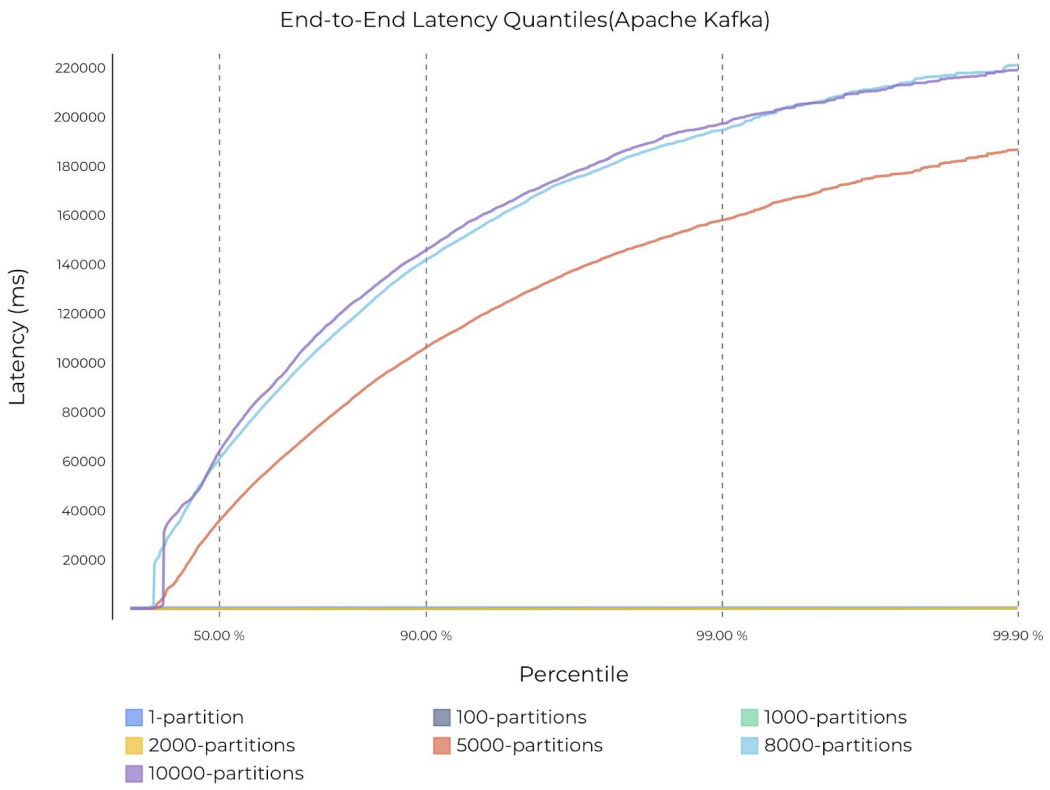

Figure 26 End-to-end latency with varying numbers of partitions and 2 acks on Kafka (with data sync)

As you can see from the figures and tables above,

- Pulsar's 99th percentile publish latency remained stable at ~10 ms. Increasing the number of partitions had no effect.
- Kafka's 99th percentile publish latency degraded incrementally as the number of partitions increased and was 30 times higher at 10000 partitions (as compared to 100). Kafka's 99th percentile publish latency at 10000 partitions increased to 1.7 s and was 126 times higher than Pulsar's.
- Pulsar's 99th percentile end-to-end latency remained stable at ~10 ms. Increasing the number of partitions had only a slight impact on Pulsar's 99th percentile end-to-end latency. But even at 10000 partitions, it remained relatively low at ~50 ms.
- Kafka's 99th percentile end-to-end latency degraded incrementally as the number of partitions increased. At 10000 partitions, Kafka's 99th percentile end-to-end latency increased to 200 s and was 14771 times higher than Pulsar's.

Ack = 1, Async local durability

Figure 27 shows the publish latency difference between Pulsar and Kafka when varying the number of partitions under async local durability and async replication durability (ack = 1). Table 15 shows the exact latency numbers for different cases.

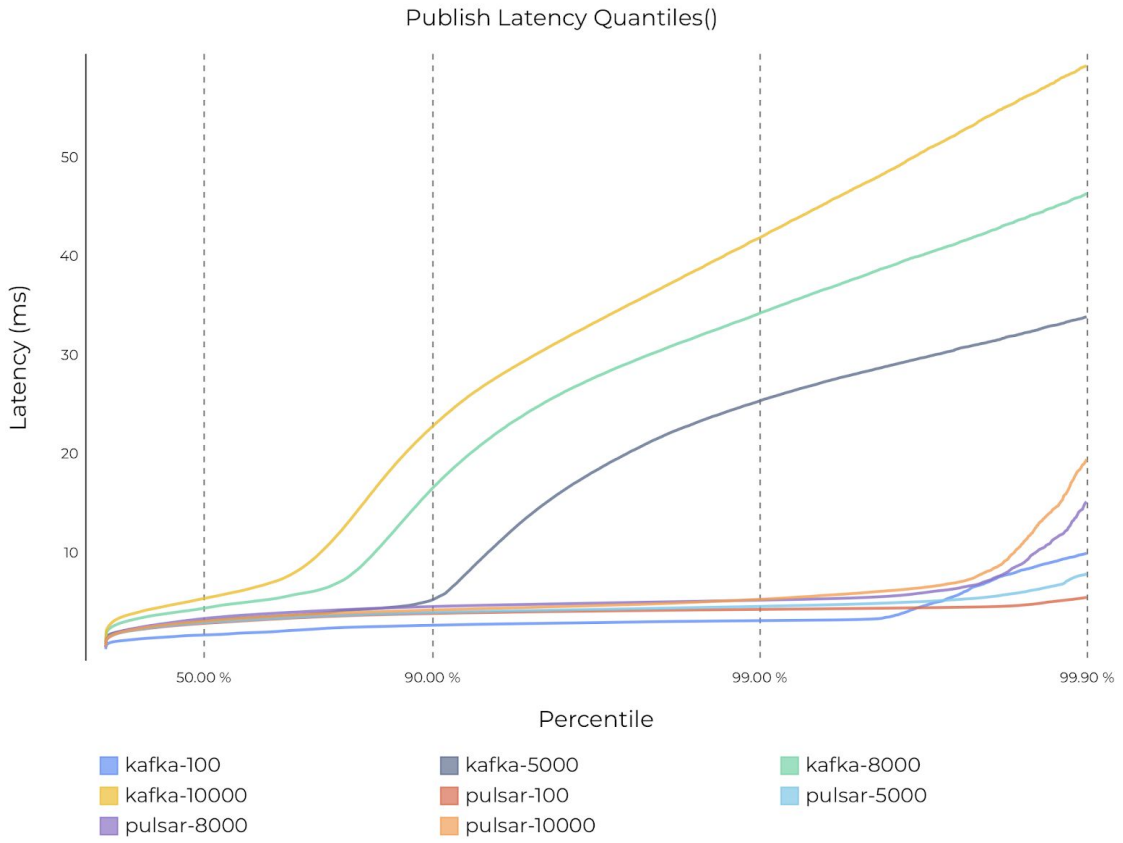

Figure 27 Publish latency with varying numbers of partitions and 1 ack (without data sync)

Table 2-15 Actual publish latency test results with varying numbers of partitions and 1 ack (without data sync)

	Average	**P50**	**P90**	**P99**	**P999**
kafka-100	*1.74*	*1.62*	*2.60*	*3.06*	*9.91*
kafka-5000	*4.13*	*3.05*	*5.20*	*25.32*	*33.86*
kafka-8000	*6.84*	*4.32*	*16.51*	*34.19*	*46.34*

kafka-10000	*8.95*	*5.32*	*22.75*	*41.83*	*59.32*
pulsar-100	*2.86*	*2.79*	*3.8*	*4.21*	*5.45*
pulsar-5000	*2.98*	*2.95*	*3.89*	*4.51*	*7.85*
pulsar-8000	*3.26*	*3.27*	*4.50*	*5.14*	*15.21*
pulsar-10000	*3.06*	*2.97*	*4.15*	*5.23*	*19.47*

Figure 28 and Figure 29 show how varying the number of partitions affected end-to-end latency in Pulsar and Kafka, respectively.

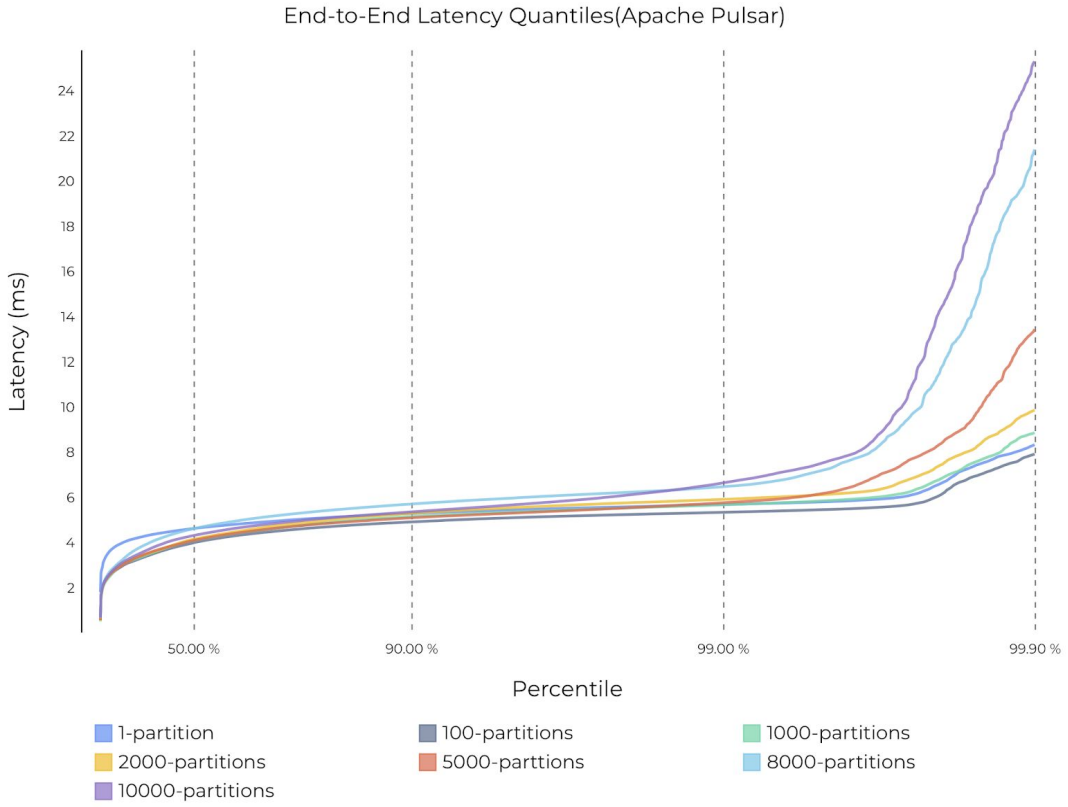

Figure 28 End-to-end latency with varying numbers of partitions and 1 ack on Pulsar (without data sync)

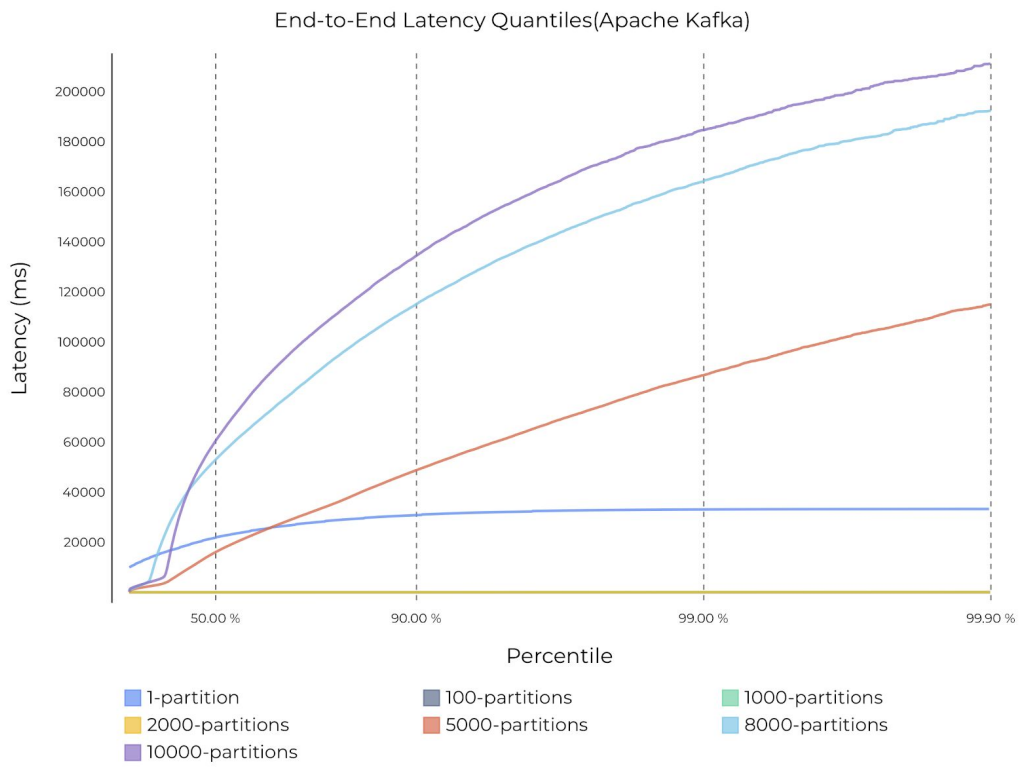

Figure 29 End-to-end latency with varying numbers of partitions and 1 ack on Kafka (without data sync)

As you can see from the above figures and tables,

- Pulsar's 99th percentile publish latency remained stable at ~4 to ~5 ms. Increasing the number of partitions had no impact.
- Kafka's 99th percentile publish latency degraded incrementally as the number of partitions increased and was 13 times higher at 10000 partitions (as compared to 100). Kafka's 99th percentile publish latency at 10000 partitions increased to 41 ms and was 8 times higher than Pulsar's.
- Pulsar's 99th percentile end-to-end latency remained stable at ~4 to ~6 ms. Increasing the number of partitions had a slight impact on Pulsar's 99.9th percentile end-to-end latency but, even at 10000 partitions, it stayed relatively low (within 24 ms).
- Kafka's 99th percentile end-to-end latency degraded incrementally as the number of partitions increased. At 10000 partitions, Kafka's 99th percentile end-to-end latency went up to 180 s and was 34416 times higher than Pulsar's.

Ack = 2, Async local durability

Figure 30 shows the differences in publish latency between Pulsar and Kafka when varying the number of partitions under async local durability and sync replication durability (ack = 2). Table 16 shows the exact latency numbers for each case.

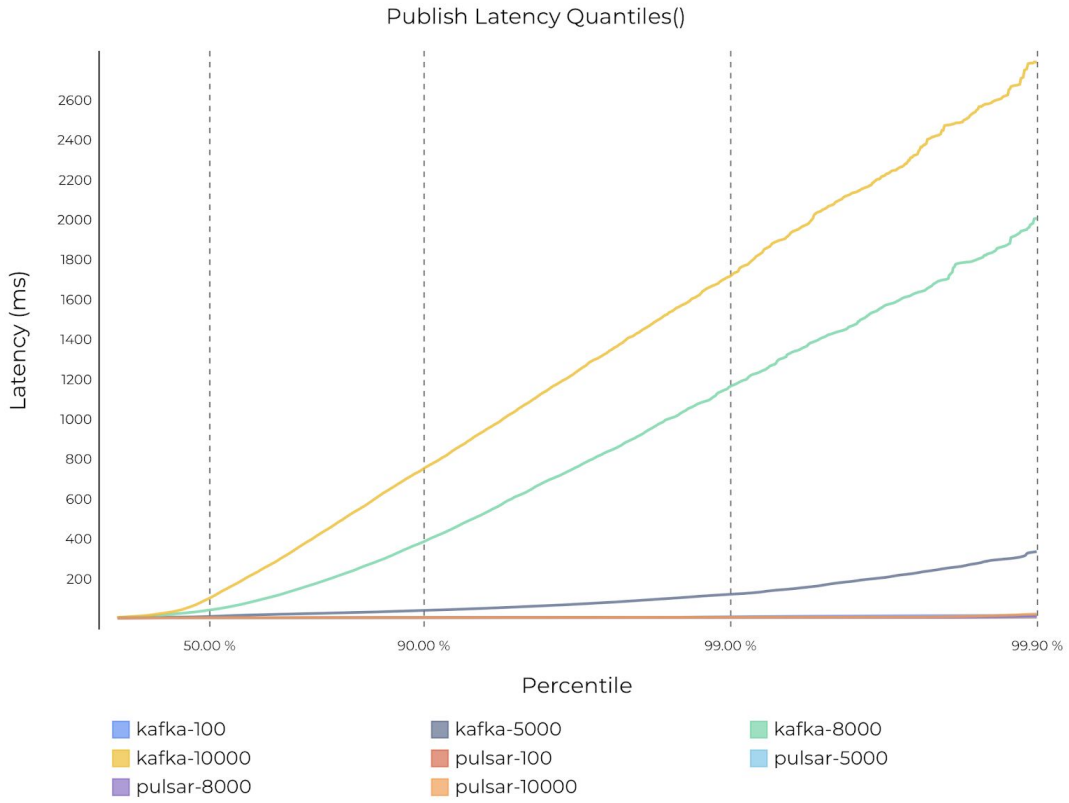

Figure 30 Publish latency with varying numbers of partitions and all/2 ack (without data sync)

Table 2-16 Actual publish latency test results with varying numbers of partitions and all/2 ack (without data sync)

	Average	**P50**	**P90**	**P99**	**P999**
kafka-100	*3.01*	*2.77*	*3.97*	*8.47*	*15.57*
kafka-5000	*19.33*	*10.16*	*40.40*	*121.40*	*336.21*
kafka-8000	*138.19*	*42.52*	*385.86*	*1164.90*	*2008.28*
kafka-10000	*266.66*	*102.55*	*752.95*	*1717.83*	*2797.51*
pulsar-100	*2.99*	*2.97*	*3.96*	*4.47*	*6.19*

pulsar-5000	*3.13*	*3.10*	*4.17*	*4.98*	*9.45*
pulsar-8000	*3.44*	*3.44*	*4.64*	*5.36*	*12.92*
pulsar-10000	*3.32*	*3.24*	*4.39*	*6.18*	*23.10*

Figure 31 and Figure 32 show how varying the number of partitions affected end-to-end latency on Pulsar and Kafka, respectively.

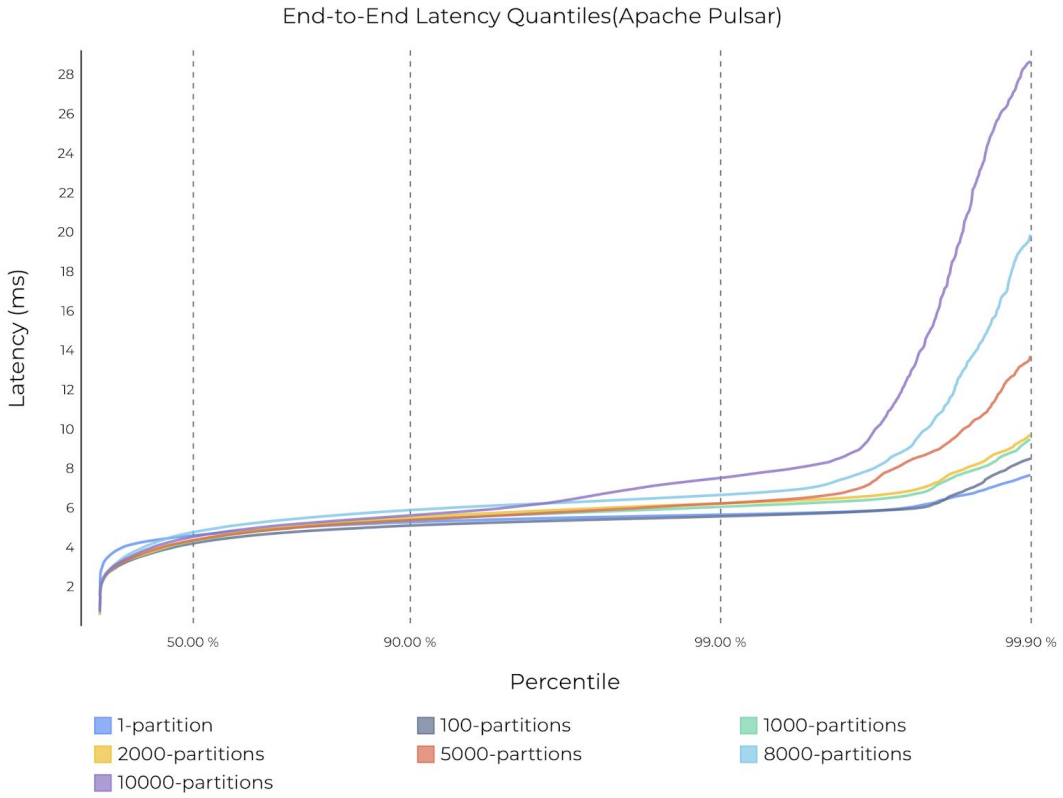

Figure 31 End-to-end latency with varying numbers of partitions and 2 acks on Pulsar (without data sync)

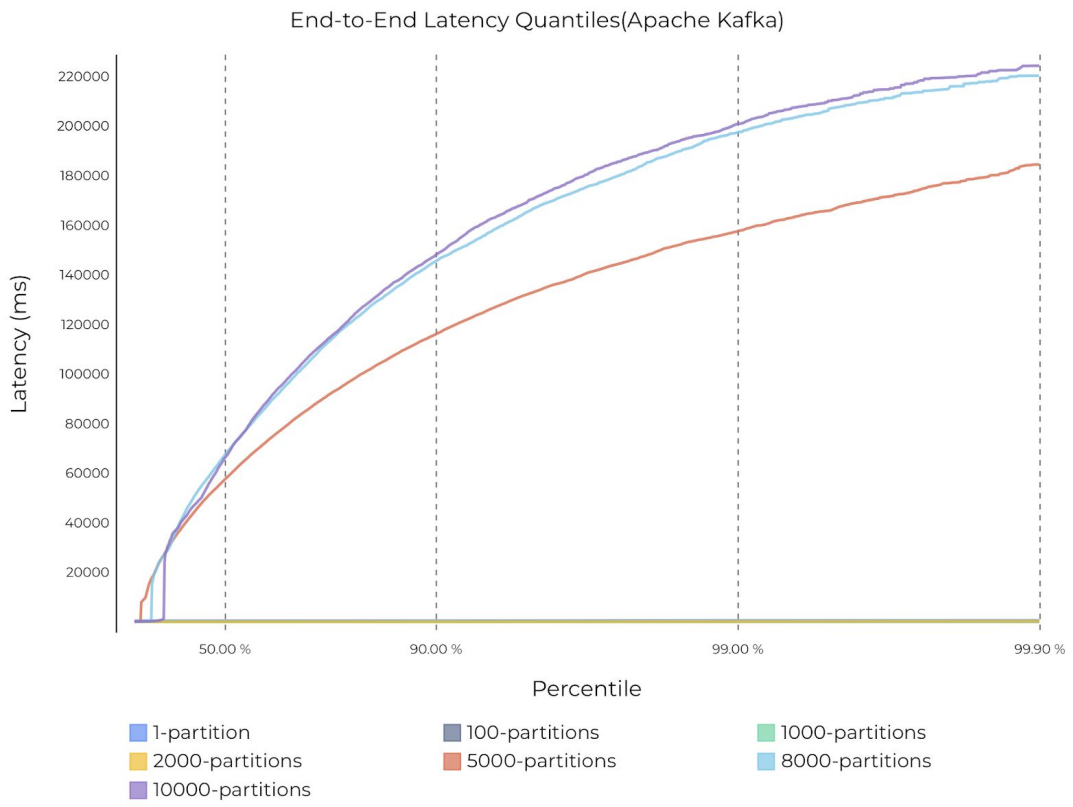

Figure 32 End-to-end latency with varying numbers of partitions with 2 acks on Kafka (without data sync)

As you can see:

- Pulsar's 99th percentile publish latency remained stable at ~4 to ~5 ms. Increasing the number of partitions had no impact.
- Kafka's 99th percentile publish latency degraded incrementally as the number of partitions increased and was 202 times higher at 10000 partitions (as compared to 100). At 10000 partitions, Kafka's 99th percentile publish latency increased to 1.7 s and was 278 times higher than Pulsar's.
- Pulsar's 99th percentile end-to-end latency remained stable at ~4 to ~6 ms. Increasing the number of partitions had a slight impact on Pulsar's
  99.9th percentile end-to-end latency, but it remained relatively low within 28 ms.
- Kafka's 99th percentile end-to-end latency degraded incrementally as the number of partitions increased. At 10000 partitions, Kafka's

99th percentile end-to-end latency increased to 200 s and was 32362 times higher than Pulsar's.

Catch-up Read Test

The following is the test setup.

The test was designed to determine the maximum throughput Pulsar and Kafka can achieve when processing workloads that contain catch-up reads only. Our test strategy included the following principles and expected guarantees:

- Each message was replicated three times to ensure fault tolerance.
- We varied the number of acknowledgements to measure changes in throughput under various replication durability guarantees.
- We enabled batching on Kafka and Pulsar, batching up to 1 MB of data for a maximum of 10 ms.
- We benchmarked both systems with 100 partitions.
- We ran a total of four clients—two producers and two consumers.
- We used a message size of 1 KB.

At the beginning of the test, the producer started sending messages at a fixed rate of 200 K/s. When 512 GB of data had accumulated in a queue, the consumers began processing. The consumers first read the accumulated data from beginning to end, and then went on to process incoming data as it arrived. The producer continued to send messages at the same rate for the duration of the test.

We evaluated how quickly each system was able to read the 512 GB of backlog data. We compared Kafka and Pulsar under different durability settings.

The following is the result for each test.

#1 Async local durability with Pulsar's journal bypassing feature enabled

In this test, we used equivalent async local durability guarantees on Pulsar and Kafka. We enabled Pulsar's new journal bypassing feature to match the local durability guarantee provided by Kafka's default fsync settings.

As you can see from our results shown in Figure 33 below,

- Pulsar's maximum throughput reached 3.7 million messages/s (3.5 GB/s) when processing catch-up reads only.
- Kafka's only reached a maximum throughput of 1 million messages/s (1 GB/s).
- Pulsar processed catch-up reads 75% faster than Kafka.

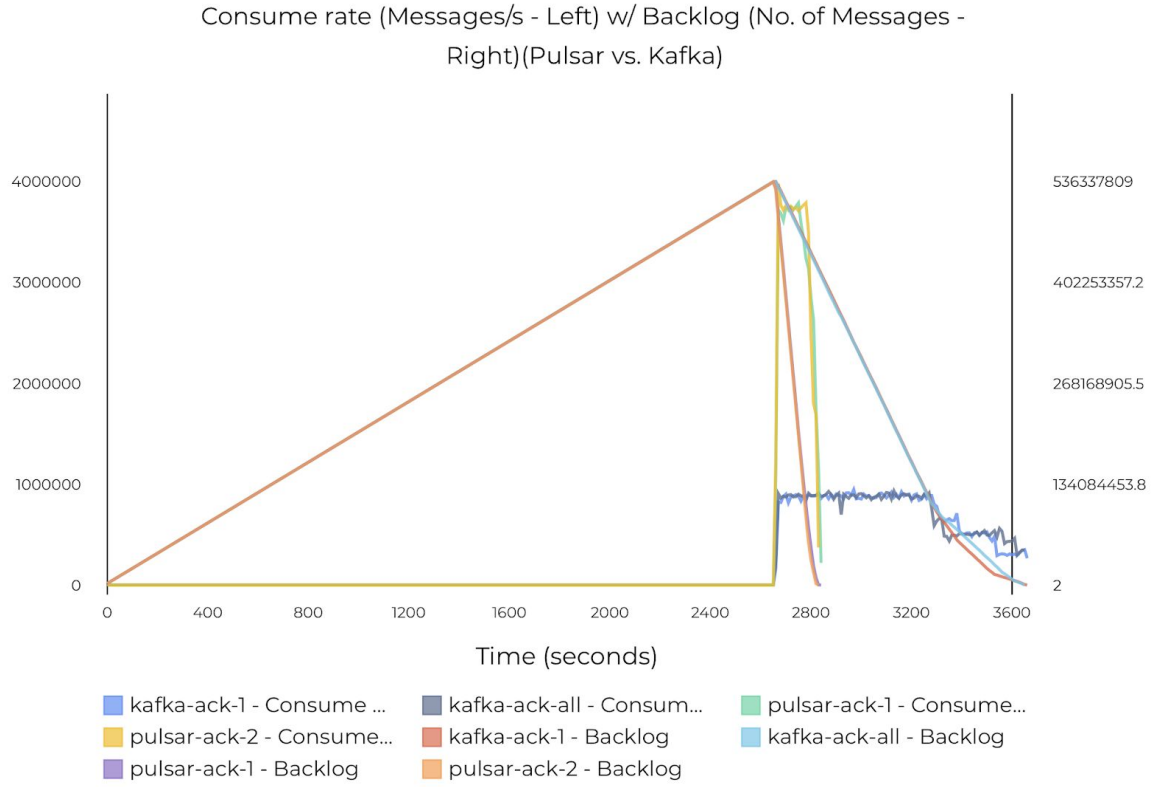

Figure 33 Catch-up read throughput on Pulsar and Kafka (bypass journaling)

#2 Async local durability with Pulsar's journal bypassing feature disabled

In this test, we used async local durability guarantees on Pulsar and Kafka, but disabled Pulsar's journal bypassing feature.

As you can see from our results shown in Figure 34 below,

- Pulsar's maximum throughput when processing catch-up reads reached 1.8 million messages/s (1.7 GB/s).
- Kafka only reached a maximum throughput of 1 million messages/s (1 GB/s).
- Pulsar processed catch-up reads twice as fast as Kafka.

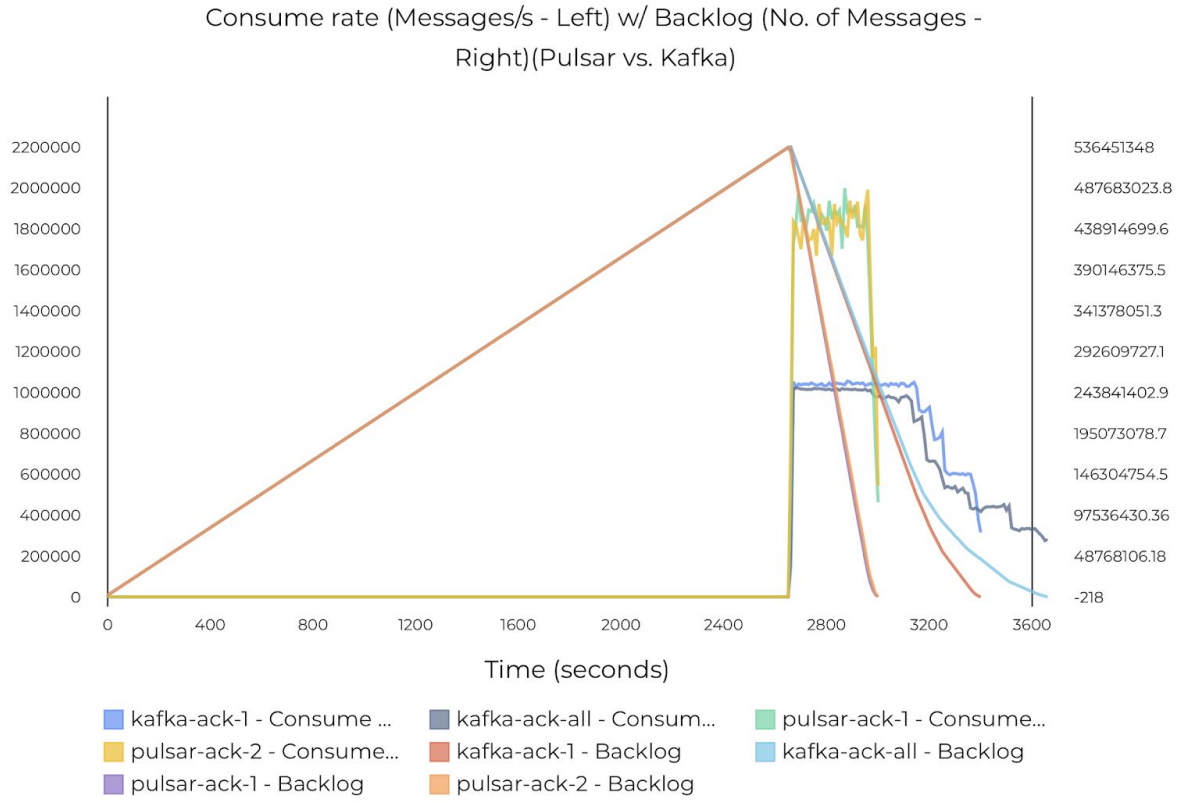

Figure 34 Catch-up read throughput on Pulsar and Kafka (with data sync)

#3 Sync local durability

In this test, we compared Kafka and Pulsar under equivalent sync local durability guarantees.

As you can see from our results shown in Figure 35 below,

- Pulsar's maximum throughput reached 1.8 million messages/s (1.7 GB/s) when processing catch-up reads only.
- Kafka only reached a maximum throughput of 1 million messages/s (1 GB/s).
- Pulsar processed catch-up reads twice as fast as Kafka.

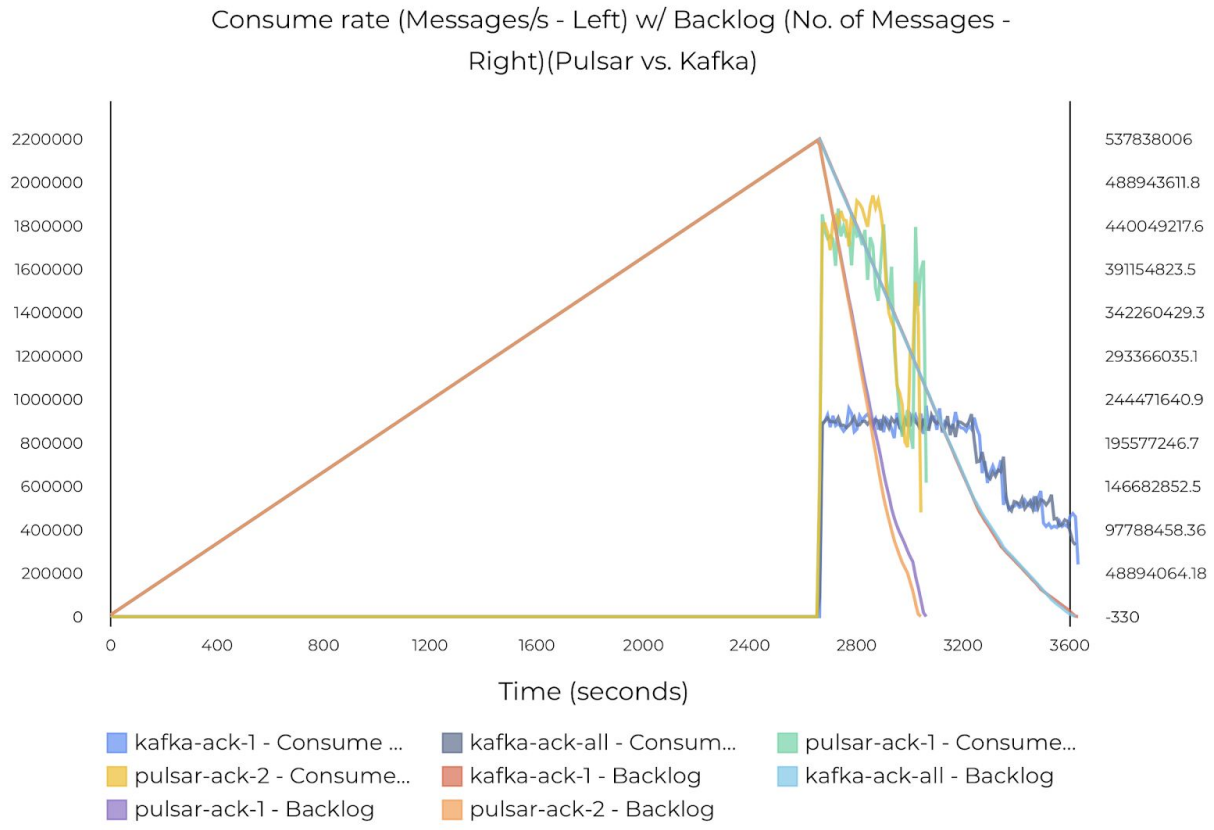

Figure 35 Catch-up read throughput on Pulsar and Kafka (without data sync)

Mixed Workload Test

The following is the test setup.

This test was designed to evaluate how catch-up reads affect publish and

tailing-reads in mixed workloads. Our test strategy included the following principles and expected guarantees:

- Each message was replicated three times to ensure fault tolerance.
- We enabled batching for Kafka and Pulsar, batching up to 1 MB of data for a maximum of 10 ms.
- We benchmarked both systems with 100 partitions.
- We compared Kafka and Pulsar under different durability settings.
- We ran a total of four clients—two producers and two consumers.
- We used a message size of 1 KB.

At the beginning of the test, the both producers started sending data at a fixed rate of 200 K/s and one of the two consumers began processing tailing-reads immediately. When 512 GB of data had accumulated in a queue, the other (catch-up) consumer began reading the accumulated data from beginning to end, and then went on to process incoming data as it arrived. For the duration of the test, both producers continued to publish at the same rate and the tailing-read consumer continued to consume data at the same rate.

The following is the result for each test.

#1 Async local durability with Pulsar enabled bypass-journal feature

In this test, we compared Kafka and Pulsar with equivalent async local durability guarantees. We enabled Pulsar's new journal bypassing feature to match the local durability guarantee provided by Kafka's default fsync settings.

As you can see in Figure 36 below, catch-up reads caused significant write delays in Kafka, but had little impact on Pulsar. Kafka's 99th percentile publishing latency increased to 1-3 seconds while Pulsar's remained steady

at several milliseconds to tens of milliseconds.

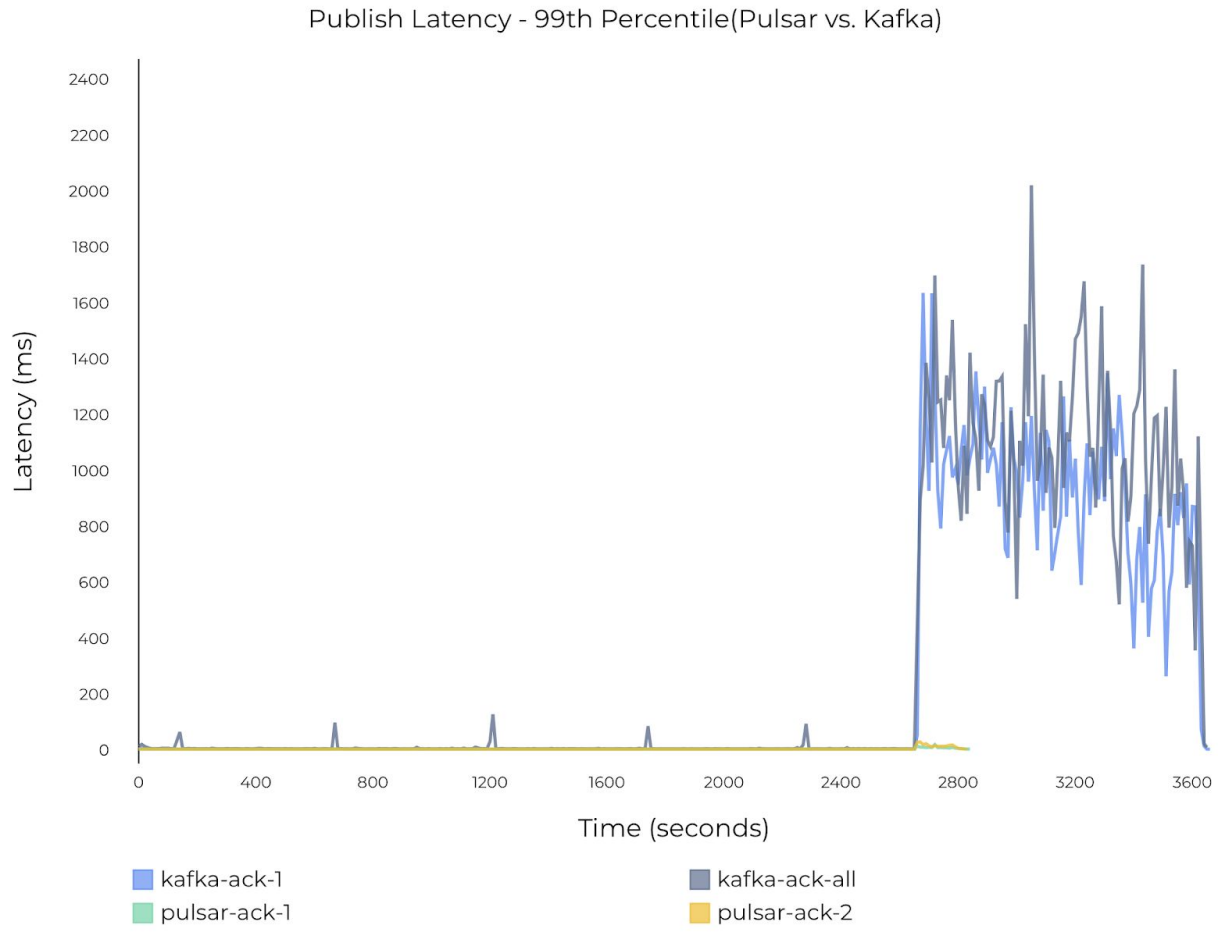

Figure 36 Effect of catch-up reads on publish latency on Pulsar and Kafka (bypass journaling)

#2 Async local durability with Pulsar's journal bypassing feature disabled

In this test, we used async local durability guarantees on Pulsar and Kafka, but disabled Pulsar's journal bypassing feature.

As you can see in Figure 37 below, catch-up reads caused significant write delays on Kafka, but had little impact on Pulsar. Kafka's 99th percentile publishing latency increased to 2-3 seconds while Pulsar's remained steady at several milliseconds to tens of milliseconds.

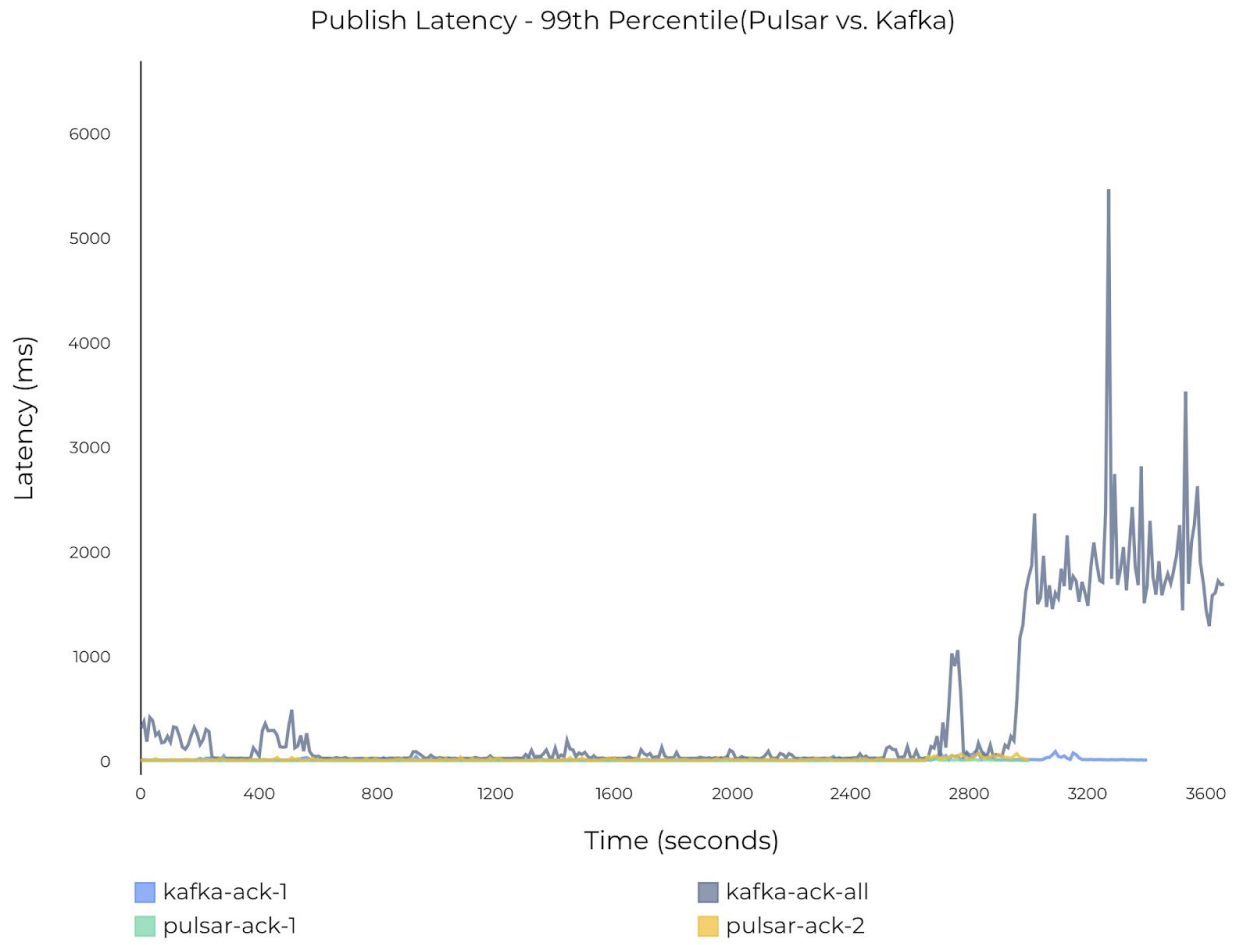

Figure 37 Effect of catch-up reads on publish latency on Pulsar and Kafka (with data sync)

#3 Sync local durability

In this test, we compared Kafka and Pulsar with equivalent sync local durability guarantees.

As you can see in Figure 38 below, catch-up reads caused significant write delays on Kafka, but had little impact on Pulsar. Kafka's 99th percentile publishing latency increased to ~1.2 to ~1.4 seconds while Pulsar's remained steady at several milliseconds to tens of milliseconds.

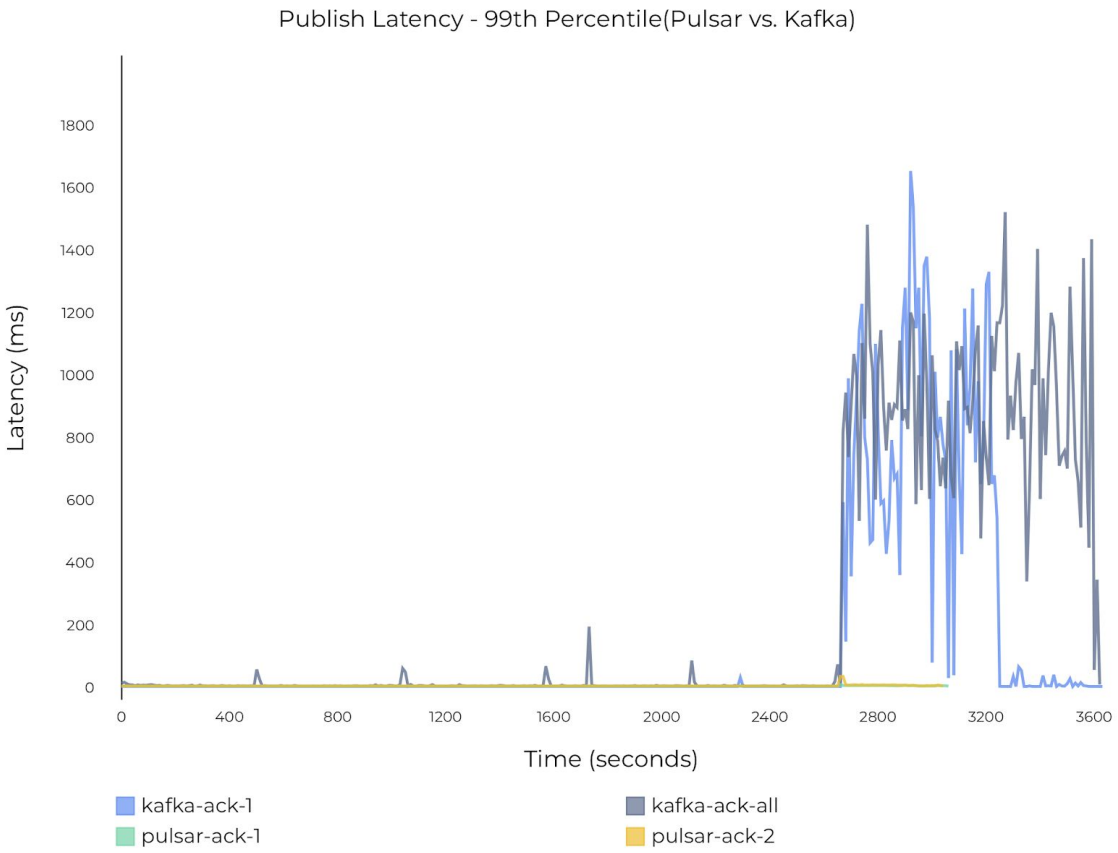

Figure 38 Effect of catch-up reads on publish latency on Pulsar and Kafka (without data sync)

Conclusion

Below is a summary of our findings based on the results of our benchmark.

- After the configuration and tuning errors were corrected, Pulsar matched the end-to-end latency Kafka had achieved in Confluent's limited use case.
- Under equivalent durability guarantees, Pulsar outperformed Kafka in workloads that simulated real-world use cases.
- Pulsar delivered significantly better latency and better I/O isolation than
  Kafka in every test, regardless of the durability guarantee settings used
  or the numbers of subscriptions, partitions, or clients specified.

About the Author

Sijie Guo is the co-founder and CEO of StreamNative, which provides a cloud-native event streaming platform powered by Apache Pulsar. Sijie has worked on messaging and streaming data technologies for more than a decade. Prior to StreamNative, Sijie cofounded Streamlio, a company focused on real-time solutions. At Twitter, Sijie was the tech lead for the messaging infrastructure group, where he co-created DistributedLog and Twitter EventBus. Prior to that, he worked on the push notification infrastructure at Yahoo!, where he was one of the original developers of BookKeeper and Pulsar. He is also the VP of Apache BookKeeper and PMC member of Apache Pulsar. You can follow him on twitter.

Penghui Li is a PMC member of Apache Pulsar and a tech lead in Zhaopin.com, where he serves as the leading promoter to adopt Pulsar. His career has always involved messaging service from the messaging system, through the microservice, and into the current world with Pulsar. You can follow him on twitter.

About StreamNative

Founded by the original developers of Apache Pulsar and Apache BookKeeper, StreamNative provides a cloud-native event streaming platform powered by Apache Pulsar that enables companies to access enterprise data as real-time event streams.

For more information about StreamNative, refer to the following channels.

- StreamNative Website: https://streamnative.io
- StreamNative Twitter: https://twitter.com/streamnativeio