Crucial capabilities: Selecting the right messaging and event streaming platform.

Who is this report for?

This report is aimed at architects, technology leaders, and software engineers who are building event driven architectures or have some need for messaging and/or event streaming requirements in the system(s) you are building. This document reflects years of experience working with companies, mostly enterprises, who have adopted the platforms compared in this report. It tries to contextualize various platform features around the situations you are likely to find yourself in and tries to explain what capabilities will be most useful to help you solve the challenges you are likely to encounter. When platforms take different approaches, this report tries to give you perspective in terms of situations where each approach is better suited.

If you are embarking on a project that requires event driven microservices, asynchronous communication between components, event based logging, or real time data pipelines, this report will give you a deep technical comparison between three popular platforms: Apache Kafka, Apache Pulsar and RabbitMQ.

Messaging Platform Scorecards

Scoring Methodology

Using the requirements above, we will assign each open source messaging platform a score of 1, 3, or 5 with the following point assignment scale:

5 points - the platform offers superior capabilities in this criteria, it is fully able to deliver the architectural benefits that we want and it does so with an approach that makes it one of the best choices for delivering these capabilities in the open source market today.

3 points - the platform can only partially satisfy our requirements or that there is considerable technical baggage which you must accept in order to achieve the results you want.

1 point - the platform is either missing these capabilities entirely or that the features in this area are very limited and will likely limit your architecture's ability to deliver the benefits you need in the requirement area.

This report does not declare a winner because depending on your situation and requirements, each of these platforms may be best suited to address your specific set of requirements. Instead it opts to break down the crucial capability areas you should at least consider when making a product decision and allows you to see how we score each platform along with a description of its capabilities, and enough detail for you to walk away with a clear picture of the differences between the products evaluated in this report.

Scorecard Comparison

Messaging Requirement	Pulsar	Kafka	RabbitMQ
Topic / Queue Scalability			
Theoretically infinite topic capacity	5	5	3
Independent scaling of compute and storage	5	1	1
Instant topic scaling	5	3	3
Elastic, bidirectional scaling	5	1	1
Consumer Scalability			
Elastic consumer scalability	5	1	5
Zero degradation consumer scaling	5	1	5
Messaging/Queuing			
Fan out support	5	5	3
Work queue support	5	1	3
Negative message acknowledgement & redelivery	5	3	5
Delayed message delivery	5	1	3
Dead letter routing	5	5	5
Guaranteed ordering	3	3	3
Event Streaming			
Event persistence and replay	5	5	3
Cost-effective, theoretically infinite retention	5	1	1
Connector support	3	5	1

Messaging Requirement	Pulsar	Kafka	RabbitMQ
Event processing	3	5	1
Delivery Processing Guarantees			
Message deduplication	5	5	3
Transactional support	5	5	1
Unacknowledged message retention	5	1	1
Failover and DR			
Local failover	5	5	5
Georeplication	5	5	3
Message Publishing Performance			
Durable write performance	5	1	1
Non-durable write performance	3	5	1
Interoperability			
Schema support	5	1	1
Administration			
Multi tenancy	5	1	3
Upgrades / compatibility of clients	5	3	3
Observability tools / dashboards	5	5	3
Governance			
Role based access control (RBAC)	5	3	3
Content encryption	5	1	1
Policy-based data management	5	3	3
Topic compaction	5	3	1
Community & Non-functionals			
Open source ecosystem	3	5	3
Availability of skillset	3	5	5

Open Source Messaging Platforms Overview

Before we dive into the requirements and scoring, it will be helpful to provide a high level overview of the platforms we will compare in this report. This gives a description of the key concepts you must understand when working with each platform along with a summary of the important architectural choices each platform is built upon.

It is important to note that we are considering open source products only. We do not consider proprietary vendor features or open-core products which are built on top of the open source projects.

Apache Kafka

Apache Kafka has emerged as one of the leading open source event streaming platforms. It is known for its high performance, real-time data streaming capabilities. Kafka has become the backbone of many modern data pipelines and event-streaming architectures.

Key Concepts

Apache Kafka's architecture is centered around topics, producers, consumers, and the broker-based publish-subscribe model. Let's explore each component and understand how they interact:

Topics: Topics serve as the central entities in Kafka, representing categories or feeds of messages. Producers publish messages to specific topics, and consumers subscribe to one or more topics to consume messages.

Topic Partition: A topic partition is a physically ordered and immutable sequence of messages within a topic. It enables horizontal scalability by allowing multiple partitions to be distributed across different brokers, enabling parallel processing and fault tolerance within a Kafka cluster. Physical storage for a topic partition is tightly coupled to the specific brokers where the topic partition resides.

Producers: Producers are responsible for generating and publishing messages to Kafka topics. They send messages to Kafka brokers, which are then distributed to the appropriate topics and partitions.

Consumers: Consumers subscribe to topics and consume messages in real-time. They can consume messages from one or more partitions within a topic, allowing for parallel processing and scalability.

Consumer Groups: Consumer groups in Kafka allow more than one consumer to work together to process events from a topic. The size of a consumer group is limited to the number of partitions associated with a topic.

Brokers: Kafka brokers form the core infrastructure of a Kafka cluster. They are responsible for receiving messages from producers, storing them on disk in a distributed and fault-tolerant manner, and serving them to consumers upon request.

Architecture Overview

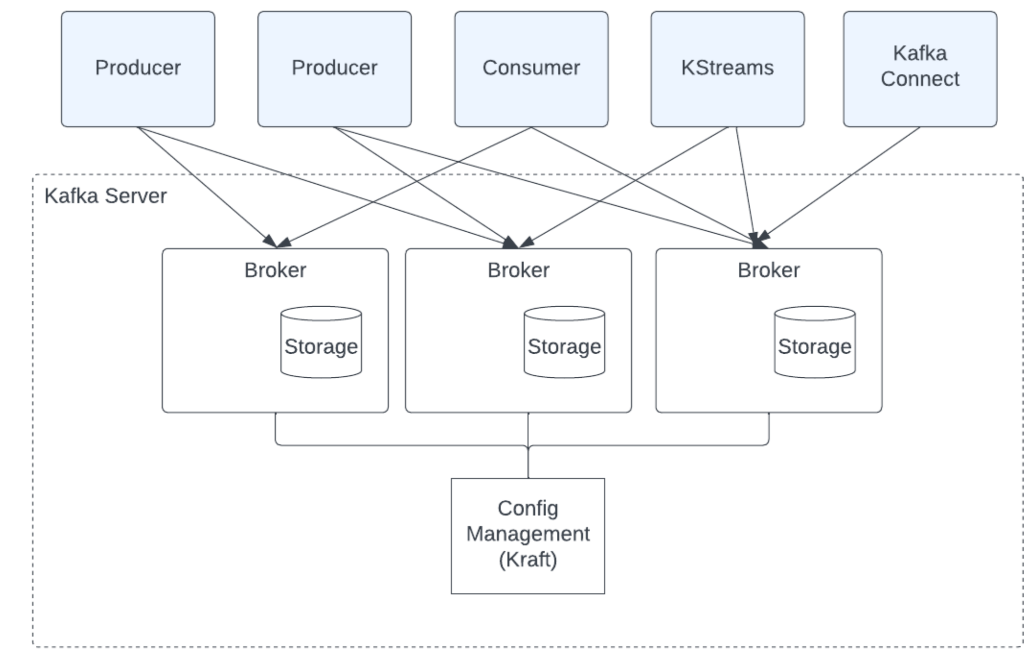


Figure 1: Architectural overview of a typically Kafka deployment

Kafka Brokers:

Kafka brokers are the heart of a Kafka cluster. They form the core infrastructure responsible for handling the storage, distribution, and replication of messages. Each broker is a separate server that manages one or more Kafka topic partitions, handling the read and write requests from producers and consumers. Brokers store messages in durable storage and replicate them across multiple brokers for fault tolerance.

Distributed Configuration Management

In older versions of Kafka, ZooKeeper was a critical component used for coordination, metadata management, and leader election among brokers. However, with recent versions, Kafka has replaced ZooKeeper with Kraft for internal metadata management.

Kafka Connect

Kafka Connect is a scalable and fault-tolerant framework for integrating Kafka with external systems. It provides connectors that allow seamless data ingestion and egress from various sources and sinks, such as databases, message queues, and data lakes. Kafka Connect simplifies the process of building and managing data pipelines between Kafka and external systems.

Kafka Streams

Kafka Streams is a stream processing library that enables real-time processing of data streams within the Kafka ecosystem. It allows developers to build scalable and fault-tolerant stream processing applications using a high-level DSL or the Streams API. Kafka Streams facilitates tasks such as data transformations, aggregations, filtering, and windowing, making it a versatile tool for real-time analytics and data enrichment.

Apache Pulsar

Apache Pulsar is a distributed pub-sub messaging platform designed to provide scalable, durable, and flexible messaging capabilities for modern data-intensive applications. With its unique architecture and rich feature set, Pulsar has gained popularity as a reliable messaging system used by numerous organizations for real-time event streaming and data processing.

Key Concepts

Apache Pulsar's architecture is built around the concepts of topics, producers, consumers, and clusters. Let's delve into each component and explore how they work together:

Topics and Subscriptions: Topics are the fundamental units of data distribution in Pulsar. They represent a stream of messages that can be produced and consumed. Subscriptions define the consumption behavior that consumers want when subscribing to a topic (queue vs stream vs topic behavior)

Producers and Consumers: Producers publish messages to topics, while consumers subscribe to topics and receive messages. Pulsar supports both exclusive and shared consumer models. Exclusive consumers receive messages from a specific topic exclusively, whereas shared consumers allow multiple consumers to share the load of consuming messages from a topic.

Clusters and Brokers: Pulsar clusters consist of multiple brokers, called Pulsar brokers. Brokers handle message storage, distribution, and coordination within the cluster. Each broker manages a subset of topics and maintains metadata about the topics it handles. Clusters provide high availability, fault tolerance, and scalability by distributing the load across multiple brokers.

Architectural Overview

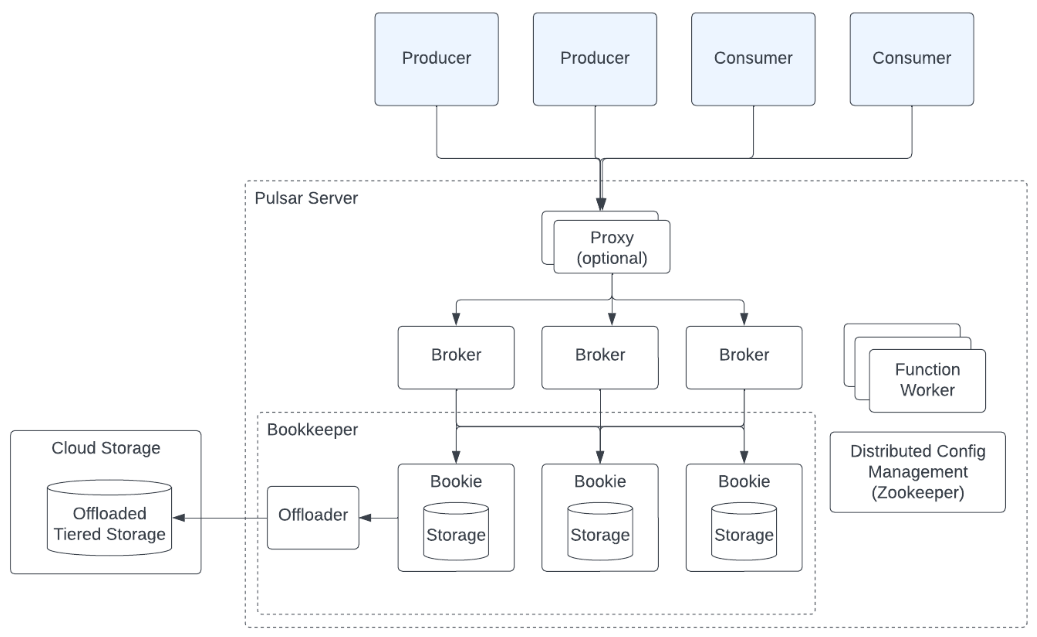


Figure 2: Architectural overview of a typically Pulsar deployment

Storage Layer - Apache BookKeeper

Pulsar relies on Apache BookKeeper, a scalable and fault-tolerant distributed storage system, as its underlying storage layer. BookKeeper ensures durability by storing messages in a write-ahead log called a ledger. BookKeeper achieves high throughput and fault tolerance by replicating the ledgers across multiple BookKeeper servers (called bookies) in a cluster.

Serving Layer - Brokers

The serving layer in Pulsar is responsible for handling message publishing, distribution, and delivery. When a producer publishes a message to a topic, the message is initially written to a BookKeeper ledger. The ledger provides durability guarantees even before the message is acknowledged to the producer. Pulsar brokers maintain metadata about topics, subscriptions, and consumers, facilitating efficient message routing.

Distributed Configuration Management

Pulsar uses a pluggable mechanism to support various implementations of distributed configuration management including Zookeeper, Etcd and others. Zookeeper is currently the default implementation used in Apache Pulsar.

Offloaders

Pulsar employs offloaders to manage the storage and retrieval of messages in durable storage, such cloud object stores like AWS S3 or Google GCS. Offloaders help optimize the storage usage in Pulsar by moving older or less frequently accessed messages from broker storage to cheaper, longer term storage such as AWS S3, Azure Blob Storage or Google Cloud Storage. This approach allows Pulsar to keep storage costs very low even when storing massive amounts of historical data.

Function Compute

Pulsar provides a built-in Function Compute capability that allows developers to process messages with serverless functions. Functions can be written in popular languages like Java, Python, and Go. With Function Compute, you can perform transformations, filtering, and enrichment on messages in real-time, making Pulsar a powerful stream processing platform.

Connectors

Pulsar offers a rich ecosystem of connectors that enable seamless integration with external systems and data sources. Connectors facilitate easy ingestion and egress of data, enabling Pulsar to interact with databases, messaging systems, data lakes, and more. These connectors simplify data integration and enable building end-to-end data pipelines.

RabbitMQ

RabbitMQ is a popular and versatile message broker known for its flexibility, reliability, and support for multiple messaging patterns. With its robust architecture and rich feature set, RabbitMQ has become a go-to solution for many companies building scalable and decoupled messaging systems.

Key Concepts

Exchanges: Exchanges receive messages from publishers and route them to appropriate queues based on predefined rules and bindings. RabbitMQ supports different exchange types, including direct, fanout, topic, and headers exchanges, enabling flexible message routing strategies.

Queues: Queues store messages until they are consumed by consumers. Messages are delivered to queues based on bindings with exchanges. RabbitMQ offers durable queues that survive broker restarts and transient queues that are automatically deleted when no longer in use.

Bindings: A binding is a relationship between an exchange and a queue. It defines the rules for how messages should be routed from the exchange to the queue. Bindings are crucial for message routing and act as a link between the producing side (exchange) and the consuming side (queue) within RabbitMQ's messaging system.

Publishers: Publishers, or producers, send messages to exchanges for further distribution. They define the routing key that determines how messages are routed to the appropriate queues. Publishers initiate the flow of messages within the messaging system.

Consumers: Consumers subscribe to queues and consume messages from RabbitMQ. They process messages based on their specific business logic. RabbitMQ supports multiple consumer patterns, including the competing consumers pattern and the publish-subscribe pattern.

Virtual Hosts: Virtual hosts provide logical grouping and isolation of resources within a RabbitMQ server. Each virtual host acts as an isolated messaging environment, enabling separate messaging spaces for different applications or teams. Virtual hosts manage access control and resource allocation within RabbitMQ.

Architecture Overview

Message Broker

RabbitMQ has a very simple architecture in which each message broker is meant to operate independently.

Cluster

While RabbitMQ brokers are built to operate independently, clustering brokers does provide some limited capabilities to scale exchanges and queues horizontally across multiple brokers.

Messaging Infrastructure Requirement Areas

In this section we will consider the general patterns and capabilities we need from our messaging system and take a closer look at how well each of the open source messaging platforms we considered handles that requirement. It’s important to note that this is a point in time analysis and as products evolve their ability to satisfy requirements may improve.

Topic/Queue Scalability

Theoretically Infinite Topic Capacity

The last thing you want when building out event-driven services is for your messaging infrastructure to become a bottleneck, suddenly becoming an obstacle to resilience instead of a facilitator. Of course, nothing in software is truly infinite, but we would like our messaging system to take an approach to topic scalability that recognizes the need for unbounded scaling.

There are two scenarios where this requirement will become important. The first case is where you have a topic subscriber that goes down:

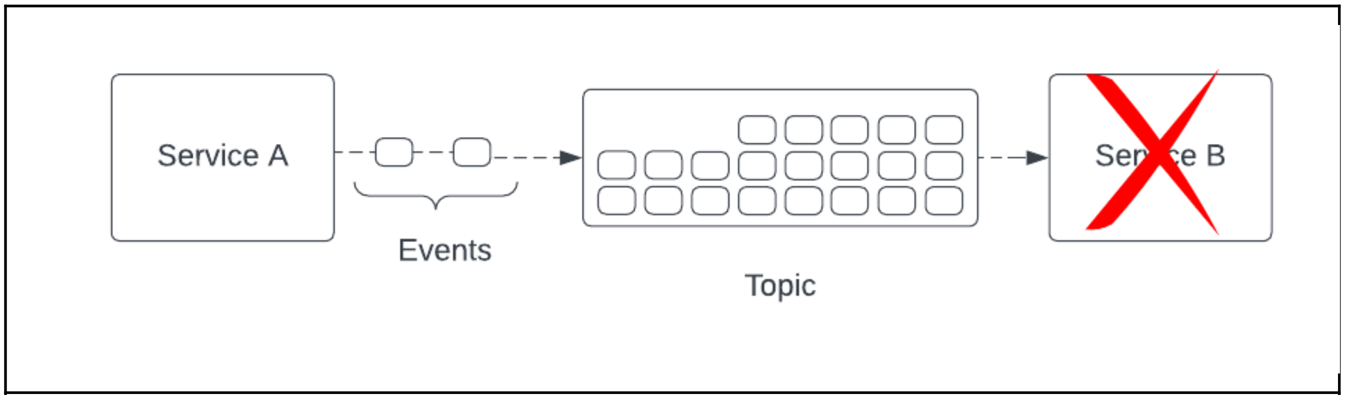


Figure 3: When consumers fail, topics will back up, potentially causing problems if your messaging system isn't prepared to handle them.

Of course, our intention is not to allow the consumer to linger in a failed state longer than it needs to. That doesn't mean that we want our messaging system to start dictating terms to us around how long a consumer can be down before it starts losing messages. The messaging platform's job is to ensure delivery of any messages that it acknowledges. It doesn't matter if the consumer is down for 10 seconds or 10 days; as far as the messaging infrastructure is concerned, it should have the capability to scale the topic as needed.

The second scenario we need to consider is what happens when the message publisher increases the volumes of messages it is publishing to such an extent that it exhausts the available capacity dedicated to our topic. You may remember earlier in this paper we touched on the relationship between a message broker and a message queue/topic. Just like a stock broker connects buyers and sellers together to complete the purchase/sale of shares of stock, a message broker connects publishers and subscribers together to ensure messages are delivered from the publisher to any subscribers who need to receive them. Message brokers are software programs and like all software programs they require some hardware infrastructure to run. Because there are hard limits on the amount of compute, memory, network and storage we can associate with a single virtual/physical machine, in order to achieve a theoretically infinite scale of our topic, this practically means that we need our topic to stretch across multiple brokers and we want to be able to horizontally scale those brokers as needed to support traffic levels as they increase.

The most widely used terminology that describes this approach is *topic partitioning.* This describes the fact that one logical topic is partitioned across multiple brokers, with each broker taking responsibility for delivering messages in the partitions it is responsible for managing. You can imagine the logical architecture as looking something like this:

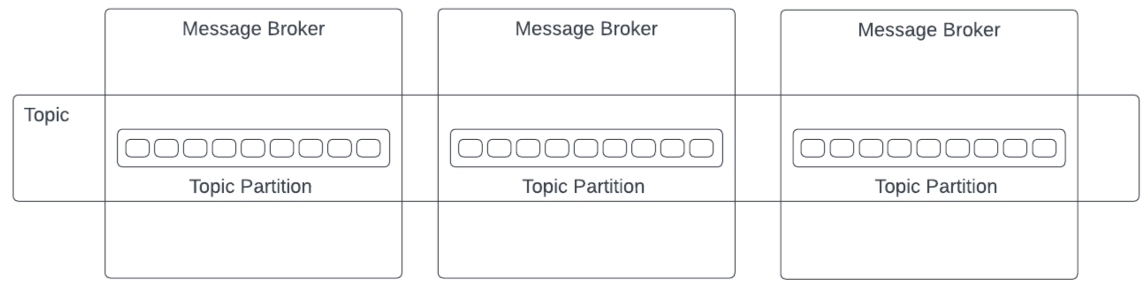


Figure 4: Logical view of a partitioned topic

Kafka

Score: ***5***

Kafka was one of the earliest products to take the approach of partitioning topics to achieve horizontal scalability. In Kafka, message data for each topic partition is stored on the Kafka broker. You can continue to add additional partitions to your topic, and brokers to your Kafka cluster (to accommodate more partitions) as needed to scale your topic to an unbounded size.

Pulsar

Score: ***5***

Pulsar fully supports partitioned topics. The serving layer for a partition in Pulsar is associated with a broker, but the storage for the partition is distributed using a distributed ledger which is provided by Apache Bookkeeper. In Pulsar, you can continue adding additional brokers and/or bookies (Bookkeeper nodes) as required to scale your topic to an unbounded size.

RabbitMQ

Score: ***3***

RabbitMQ relies on independent brokers rather than the distributed model used in Kafka and Pulsar. RabbitMQ doesn't support the concept of a partitioned topic out of the box. However, with recent enhancements, RabbitMQ now has a stream plugin which can be configured to provide some topic partitioning capabilities.

Independently Scalable Compute and Storage

As mentioned in the previous section, we need to consider scalability in cases when a consumer fails and the topic backs up. In this case, we don't necessarily need to scale the messaging compute resources since we aren't scaling the number of messages we're processing. We do however need to scale the storage since accumulating unacknowledged messages will take up space and need to be persisted.

Likewise, there are times when we have a surge of consumers that want to subscribe to or replay events from a topic or consume events in real time. In these cases, our storage doesn't need to scale, but we may need to increase the amount of compute resources that are available to serve the increased number of requests.

Kafka

Score: ***1***

A foundational aspect of Kafka's architecture is that compute and storage are tightly coupled onto a Kafka broker:

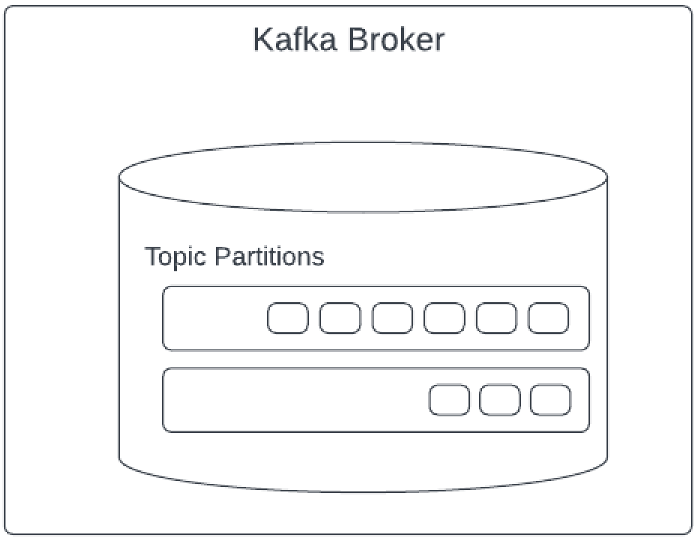


Figure 5: Simplified representation of a Kafka broker which tightly couples event storage and serving

The unfortunate consequence of this is that if you only need to scale compute to serve an influx of consumers, you are forced to add additional compute and storage since they are tightly coupled into a monolithic broker architecture.

With Kafka, the closest you can come to independently scaling storage separately from compute is by adding additional disks to the VM/machine where your brokers are running. Even then, the process requires adding additional partitions to your topic which get assigned to the new storage volumes. This has the downside that it both requires significant involvement from the operator and it also results in downtime since you'll need to generally restart the brokers for the new disk to be available and once you scale partitions the topic will halt while rebalancing consumers occurs.

Pulsar

Score: ***5***

Pulsar's architecture separates compute and storage into two independently scalable layers. Both brokers and bookies can be added and removed as needed in response to increasing and decreasing load. While brokers can be

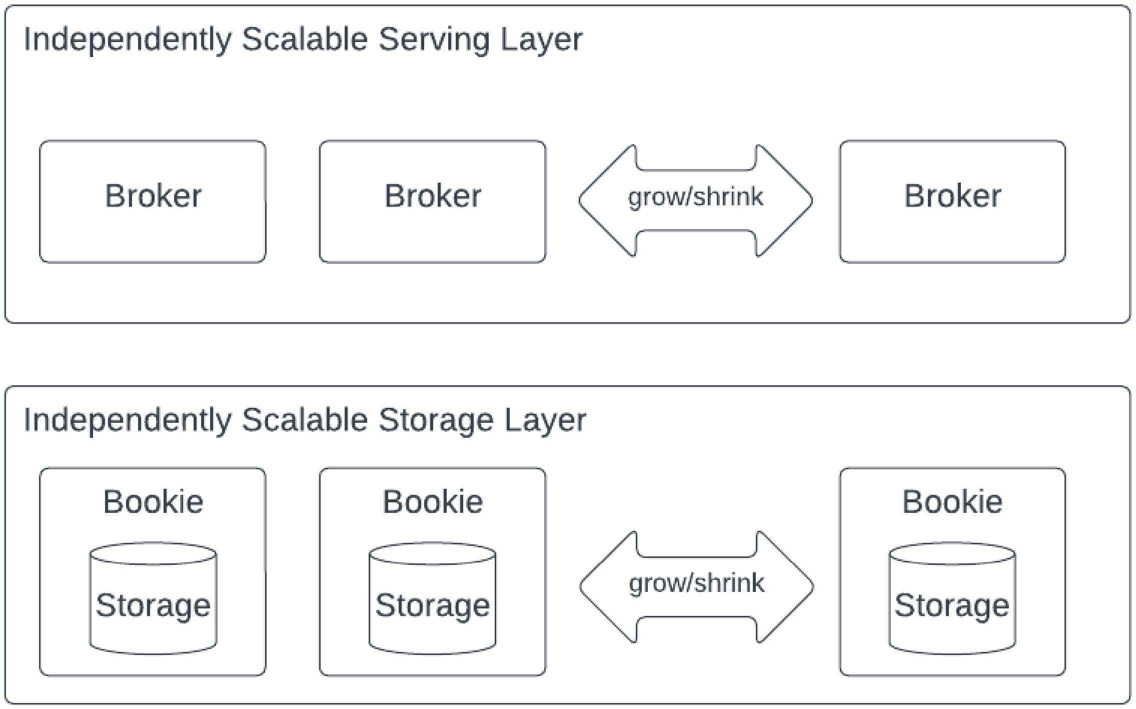


Figure 6: Representation of Pulsar which has independently scalable, decoupled event storage and serving layers

RabbitMQ

Score: ***1***

RabbitMQ uses a monolithic broker architecture to both serve messages as well as store message data using the stream plugin.

Instant Topic Scaling

When we need to scale our topics, we want to instantly alleviate any degradation in the overall stability and performance of our messaging infrastructure that results from stress the system is currently under.

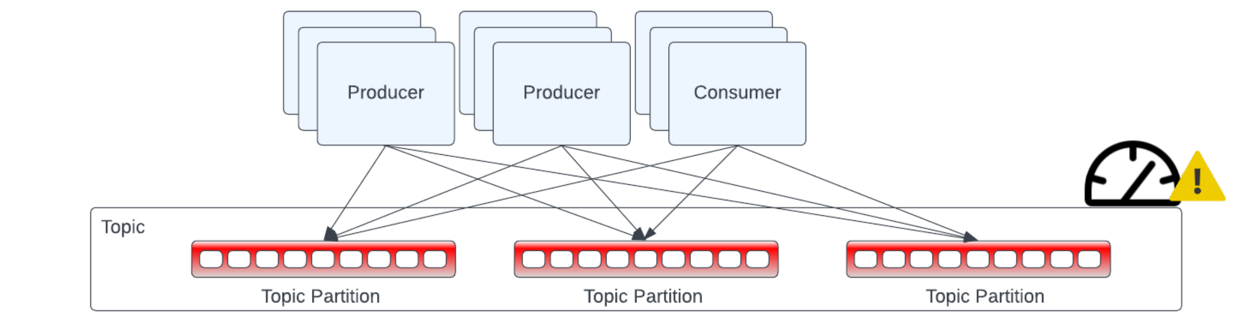


Figure 7: Representation of a partitioned topic overloaded and needing to scale

In order to accomplish this, we need to be able to add new capacity to the topic which is able to immediately start serving traffic and alleviating the pressure on the system.

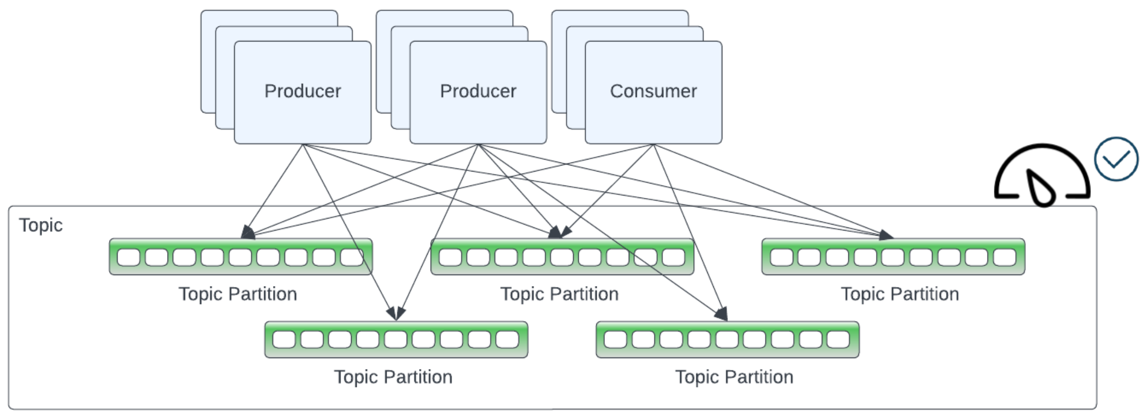


Figure 8: Ideal state of a previously overloaded topic once additional capacity is added to relieve system pressure

Kafka

Score: ***3***

In Kafka, events are mapped onto specific topic partitions by key ranges. This means that for any given key value, that key will map onto one specific partition in the Kafka topic.

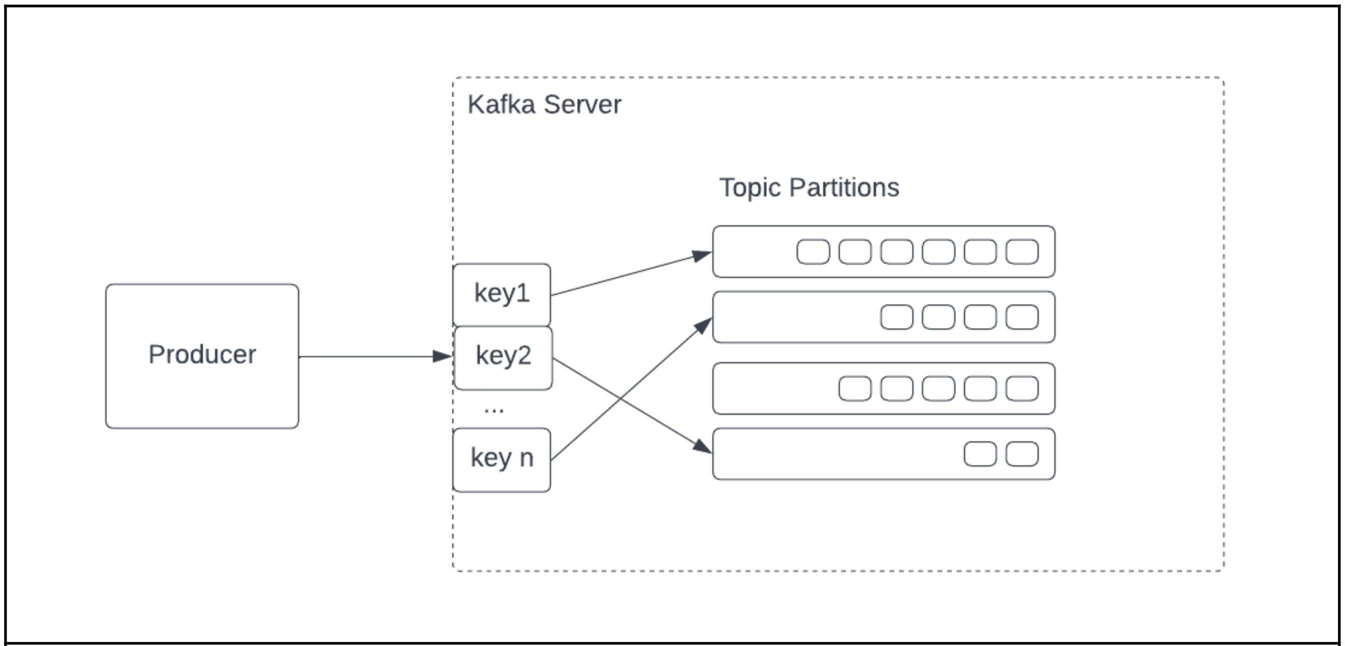


Figure 9: Kafka maps events onto partitions based on the key for the event

As long as the set of partitions remains static, the mapping of keys onto partitions will remain static as well and once a message is published to a partition, it will remain on that partition for the remainder of its lifecycle (until it reaches the retention period and is purged). Once you add additional partitions, Kafka will update this mapping and start to route new messages to partitions, but the existing message data will remain where it is. That means that partitions/brokers which were under heavy load will not immediately see any relief by adding new capacity. A telltale sign of broker stress is that you'll notice delayed messages due to growing backlogs of undelivered events. When you add new partitions in Kafka, the events remain in the partitions where they were originally published. This means that new capacity cannot serve the existing backlogs, which means that the new capacity does nothing to alleviate the pressure the system is currently under until the backlogs clear and the incoming messages are more evenly balanced across the new partitions. For those that are trying to achieve a cloud native architecture which can take advantage of auto scaling in response to changing load, this is a very unfortunate aspect of Kafka's architecture.
Additionally, since a producer in Kafka is entirely responsible for determining which partitions receive its messages, and since consumers can't change which partitions they consume from, it's not actually possible for Kafka to automatically redistribute incoming messages to different consumers to resolve a traffic imbalance, so preventing the backlogs from continuing to grow in these situations (even as new partitions are added) may require changes to the producer logic. In emergency situations like these that put SLAs at risk, on-call operators may (1) feel pressure to roll emergency patches directly into production, which can introduce new problems that can

be hard to remedy, or (2) have no direct way to resolve the increasing imbalance without assistance from a team with access to the codebase.

As we'll see coming up in consumer scalability, adding additional partitions also triggers a rebalancing on the consumer side which actually brings processing to a halt temporarily, making scaling in Kafka even more challenging.

Pulsar

*Score: **5***

Pulsar can map events to partitions based on the event key; however, with Pulsar, the topic storage is not tightly coupled to the broker. This is one of the fundamental attributes of Pulsar's architecture that we described earlier in the section on independent scalability of storage and compute. This architecture also provides benefits in terms of immediate pressure relief to the overall system when additional capacity is needed to scale the topic.

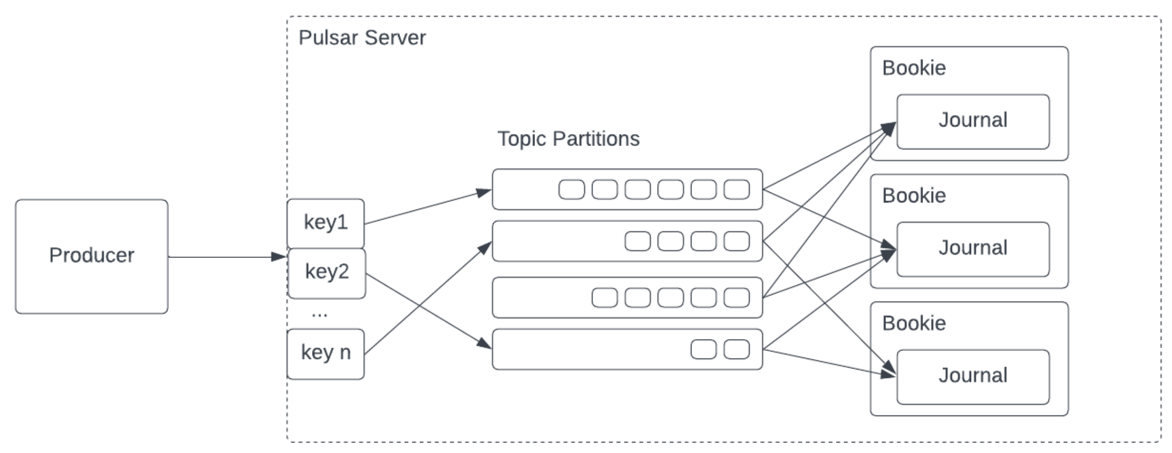


Figure 10: Pulsar's storage layer prevents any one broker/partition from being overloaded

Because messages are accepted by the broker and stored in a distributed ledger, those messages can be retrieved from the ledger by any broker. When the overall system is under stress, Pulsar allows you to add new brokers and/or topic partitions to increase parallelism from both the producer and consumer perspective. This increased capacity immediately starts handling load to quickly bring down the overall strain on the system.

RabbitMQ

*Score: **3***

With the stream plugin for RabbitMQ, you will get behavior similar to Kafka with the need to rebalance consumers in the event of a scaling event.

Elastic, Bidirectional Scaling

It’s often the case that peak traffic volumes are temporary. This may be the result of predictable seasonal patterns, such as the case of Black Friday for retailers, or it could occur unexpectedly due to press coverage or a successful marketing campaign that temporarily increases traffic before eventually receding back to steady state levels. In both cases, we don't want our messaging infrastructure to force us into a situation where we need to size for peak load and waste money on resources which aren’t being utilized the vast majority of the time. Ideally, we can respond rapidly to load increases when they occur and then scale back down once that load returns to normal levels.

Kafka

*Score: **1***

Kafka’s tight coupling of compute and storage makes it impossible to scale down a Kafka topic once you scale it up. As a consequence of this, coupled with the challenges of scaling Kafka partitions described in previous sections, the general recommendation from the Kafka community is to size your Kafka topics to anticipate and support the peak load you expect your Kafka cluster to require.

Pulsar

*Score: **5***

In Pulsar, brokers are cheap and expendable. You can add new brokers when you need to, and you can remove them if the load on your system decreases. Likewise, you can add additional bookkeeper storage nodes (bookies) if your write volumes jump up. Bookies can also be scaled down if your write volumes drop to levels which can be handled by fewer storage nodes. Because journals maintained by the bookies cannot be offloaded until they are closed (closing a journal indicates that the journal is complete and cannot have new data written to it), there will generally be some lag between the time when your write throughput drops and the time when you can reduce the number of bookies.

RabbitMQ

*Score: **1***

RabbitMQ only supports horizontal scaling of brokers for increased parallelization via the streaming plugin. You can of course vertically scale your broker, but this requires a restart of the broker which will cause interruptions to both your producers and consumers, certainly not the elastic scaling approach we expect for a modern, cloud native architecture.

Consumer Scalability

Elastic Consumer Scalability

Just like scalability with topics, we want to extend similar capabilities down to the consumer level for services that are subscribing to our topics. When load is high and our topic backlog starts to grow, we want to be able to scale the consumers up to avoid delays in processing the messages that are waiting in the topic. As the consumer starts to catch up and the topic backlog starts to recede, we no longer need as many consumers, and we would like to scale them back down following the flow here:

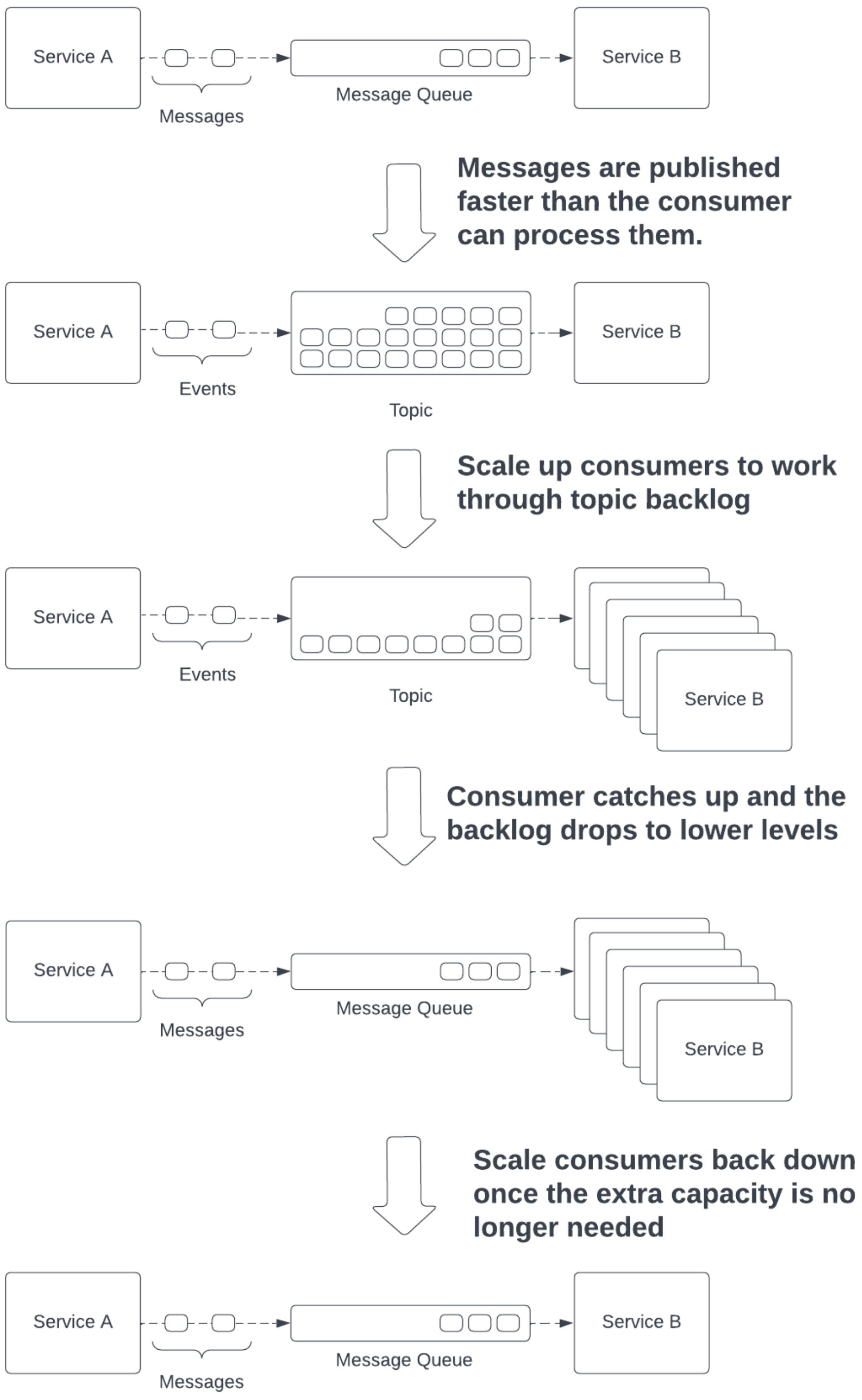

Service A
Messages
Message Queue
Service B
Messages are published faster than the consumer can process them.
Service A
Events
Topic
Service B
Scale up consumers to work through topic backlog
Service A
Events
Topic
Service B
Consumer catches up and the backlog drops to lower levels
Service A
Messages
Message Queue
Service B
Scale consumers back down once the extra capacity is no longer needed
Service A
Messages
Message Queue
Service B

Figure 11: Desired auto scaling behavior or topic consumers

Kafka

*Score: **1***

In Kafka, the mechanism that allows multiple consumers to work together to process events from a topic is called a consumer group. Each consumer in a consumer group can be responsible for processing messages from one or more partitions. Because each partition can have *at most* one consumer that processes messages from it, the maximum number of active consumers that can be assigned to a consumer group for a topic is limited to the number of partitions. This puts an upper bound on the parallelism that can be achieved in Kafka. Additionally, during unexpected traffic distributions that put greater stress on specific partitions, Kafka provides no direct solution.

Another consequence of this approach is that Kafka must carefully track which consumers are associated with a given partition, and at any given point, every partition will have a specific consumer in the consumer group that is responsible for processing events for that partition. When you add consumers to a consumer group, you will trigger something known as a consumer rebalancing. During this process, Kafka will make updates in its configuration management store to remap consumers to producers. While this process is happening and while the changes are propagating across each of the components in your Kafka deployment, no processing can occur.

This rebalancing process not only occurs when you add or remove consumers from a consumer group, but also when you attempt to scale up the number of partitions on the topic. If you try to scale both at the same time in response to some degradation of system performance in Kafka, you will actually cause the health of the system to further deteriorate as the rebalancing occurs and processing stops temporarily at a time when the system is already struggling to keep up.

For this reason, the Kafka community generally recommends that you overprovision both partitions for your topic as well as consumers in your consumer group. This ensures you have the capacity to support peak loads, but with the tradeoff that you will be paying to run underutilized consumers most of the time. Since the consumers need to be running continuously, this also creates a significant barrier to achieving an elastically scalable, cloud native architecture.

This problem not only manifests itself when scaling consumers, but also in more routine situations such as performing a rolling upgrade on consumers. As you take each consumer out of the consumer group to perform an upgrade, you will trigger a consumer rebalance causing processing to stop. If you are dealing with a large number of consumers for a large topic with a lot of partitions, this can significantly impact the overall health of your Kafka cluster during this time.

Pulsar

*Score: **5***

Pulsar provides an abstraction layer between topics and consumers called subscriptions. In Pulsar, subscriptions are used to indicate the consumption pattern which the consumer will use when consuming from the topic. Subscriptions can indicate to Pulsar that the consumer wants to receive its own copy of every message which is published to a topic; or it can indicate to Pulsar that multiple consumers will collaborate to work through the topic, with each consumer processing a subset of the messages. This latter approach is called a *Shared Subscription* in Pulsar.

With a shared subscription, you can dynamically add and remove consumers which are actively working through the messages on the topic. As soon as a new consumer is added to the subscription, it immediately starts to process messages from the topic. As the consumers catch up and the topic backlog empties out, you can just as easily remove those consumers, releasing the resources and ensuring an efficient cost structure where you only utilize resources when you need them.

Pulsar will also emit observability data about topic backlog depths via Prometheus. This gives you a convenient hook to control the number of Pulsar consumer instances using your autoscaler in response to growing and shrinking load.

RabbitMQ

*Score: **5***

When a consumer in RabbitMQ opens a channel to the broker, it can specify an exclusivity property. If the consumer is not exclusive, then multiple consumers can be created on the same channel or across different channels with the same connection. Multiple consumers can compete for messages from the same queue, allowing for load balancing and parallel processing of messages. To scale up additional consumers on the same queue you can simply start additional instances of the consumer and the broker will begin delivering messages to those consumers.

Zero Degradation Consumer Scaling

When we do need to scale consumers, we want it to be a non-event. We don’t want there to be any pauses in processing while the messaging system attempts to figure out which consumer to deliver a given message to. We simply want to add new consumers and those consumers should immediately start helping to process messages from the topic’s backlog.

Kafka

*Score: **1***

As mentioned in the previous section, when adding consumers to a consumer group in Kafka, you will trigger a consumer rebalance. While this process is in flight, processing of the Kafka topic will stop until the rebalance operation is complete.

Pulsar

Score: ***5***
Using Pulsar's shared subscription model, you can add and remove workers to/from a topic dynamically. Workers that are added immediately start processing from the topic and workers which are removed are taken out of the processing pool without any interruption to the overall processing of the topic.

RabbitMQ

Score: ***5***
Using non-exclusive consumers in RabbitMQ, you can add and remove workers to/from a queue dynamically. Consumers that are added immediately start processing from the queue and consumers which are removed are taken out of the processing pool without any interruption to the overall processing of the queue.

Queuing and Streaming Capabilities

Differences Between Queuing and Streaming

One major misconception that you'll commonly find is the idea that message queueing and event streaming are the same thing. This is understandable given that modern messaging and streaming platforms both use pub/sub as the primary interface for interacting with them. However, there are some important differences that you should be aware of. It's fairly common to assume that because event streaming is a newer technology, it must be the successor to message queuing and that event streaming solutions eliminate the need for message queuing capabilities. It turns out this is not the case, and by failing to understand the distinction, many architecture teams have selected a streaming platform as the foundation of their event driven architecture only to find that it's a poor fit for many of the requirements covered in this section. Let's start by explaining what each capability area is and when you should reach for that tool out of your toolbox.

Message queuing is a communication pattern where messages are sent to a queue and consumed by one or more recipients. It focuses on reliable delivery and can support cases where ordering of messages needs to be preserved as well as fan-out situations where you want to distribute messages across a pool of workers for efficient parallel processing. Message queuing systems provide features like message durability, acknowledgment, redelivery and the ability to scale horizontally. They are well-suited for asynchronous, reliable, and decoupled communication between components or services, ensuring message persistence and guaranteed delivery. In the world of event-driven microservices, the vast majority of your requirements are queuing requirements rather than streaming requirements.

Event streaming, on the other hand, is primarily concerned with capturing, storing, and processing streams of events in real-time. You can think of an event stream as an immutable append-only log of events. The append-only nature of this data structure makes it useful for situations where the ordering of events is important. For example, event streaming is incredibly useful if you have a service whose job is to record all of the actions a particular user takes on your application’s front end (i.e. a clickstream). Event streaming is also particularly well suited for data engineering use cases where you need to capture a stream of changes from one database and replicate that data to various other systems and data stores as they occur. Event streaming is primarily a *data* technology where the type of data you want to store can be represented as a log, like the two use cases we just mentioned. The ability to persist event streams to disk and replay those streams are two of the hallmarks of event streaming platforms. You will definitely encounter use cases when building out event driven microservices where you need the persistence and replay capabilities of event streaming so these are important, but generally speaking they will represent a tiny minority of the requirements for your overall event-driven architecture.

Message Queuing Capabilities

Fan out support

As we described in an earlier use case, fan-out support refers to a messaging pattern where a message is distributed or broadcasted from a single sender to multiple subscribers. It allows for the dissemination of a message to multiple consumer endpoints simultaneously. In this model, the publisher publishes messages to a topic. Subscribers interested in receiving these messages can subscribe to the topic. When a message is published to the topic, it is delivered to all the subscribers that have subscribed to that topic.

At a small scale, it is possible to have a single consumer receive a copy of every message that is published on a topic. As topics grow large, it becomes impossible for a single broker to handle all of that topic’s events. In turn, as you partition topics across multiple brokers, it becomes less practical for a single consumer to process every message for that topic. Therefore, when considering fan out support in a modern messaging platform, you want to consider both how your messaging platform handles cases where the topic is small enough to fit on a single broker where a single consumer can process every message, as well as how the system behaves when dealing with partitioned topics.

Kafka

*Score: **5***

As previously discussed, Kafka uses partitioned topics to represent data and consumer groups to assign consumers to process the data from any given partition. In Kafka, if you want a consumer to receive its own copy of every message for a topic, you can have a consumer group with only one consumer in it. If you want guaranteed ordering across that topic, you must have a topic with only a single partition. Once you have a topic with multiple partitions and multiple consumers in the consumer group, each consumer will receive every message for the partitions which it is responsible for processing.

Multiple consumer groups can listen to the same topic in Kafka, so in order to achieve fan out support, you simply need to create multiple consumer groups and point them at the same topic.

Pulsar

Score: ***5***

Pulsar supports the concept of both partitioned and non-partitioned topics. Non-partitioned topics are essentially just a single partition topic. Pulsar also supports more fine grained subscription models to indicate what behavior you'd like Pulsar to use when delivering messages to a consumer. When using a subscription type of *Exclusive,* you're telling Pulsar that you'd like for a single consumer to receive a copy of every message associated with a topic. If you're using a non-partitioned topic, Pulsar will deliver every message in order to that exclusive consumer. If you're using a partitioned topic, then the consumer will receive a copy of every message for the entire topic, but the ordering will only be guaranteed within a partition. Pulsar also offers a subscription type of *Key-Shared* which allows you to spread the processing of the topic messages across multiple consumers with guarantees that messages for the same message key will be processed by the same consumer in the order those messages were received. Pulsar also offers a pluggable hashing mechanism

In Pulsar, multiple subscriptions can be associated with the same topic. To achieve fan out support in Pulsar you need to decide on the behavior that you want - either *Exclusive* (one consumer gets a copy of every message with guaranteed global ordering for non-partitioned topics) or *Key-Shared* (multiple consumers work against the same topic, but one consumer will be responsible for ordered processing of all messages having a given message key).

RabbitMQ

Score: ***3***

In RabbitMQ, topics are limited to a single broker unless you rely on additional plugins, so it's possible to achieve global ordering at a relatively small scale. In RabbitMQ, a producer publishes messages via an *Exchange*. Exchanges are typed, and the type of exchange you select will impact how RabbitMQ behaves when a message comes in. One of the exchange types that RabbitMQ supports is called a *Fanout Exchange.* This tells RabbitMQ to deliver any published messages to every queue that is bound to that exchange. Likewise, when creating a consumer you can specify an exclusivity property which indicates to RabbitMQ that the consumer should receive its own copy of every message in the topic. This pattern allows you to both fan out the delivery of messages to multiple consumers and to give you a mechanism to say that you want a consumer to receive its own copy of every message in the exchange.

Work queue support

The work queue pattern, also called the task queue pattern, is a messaging pattern used to distribute and process tasks or work items asynchronously across a pool of workers that each subscribe to a queue and process a subset of the messages:

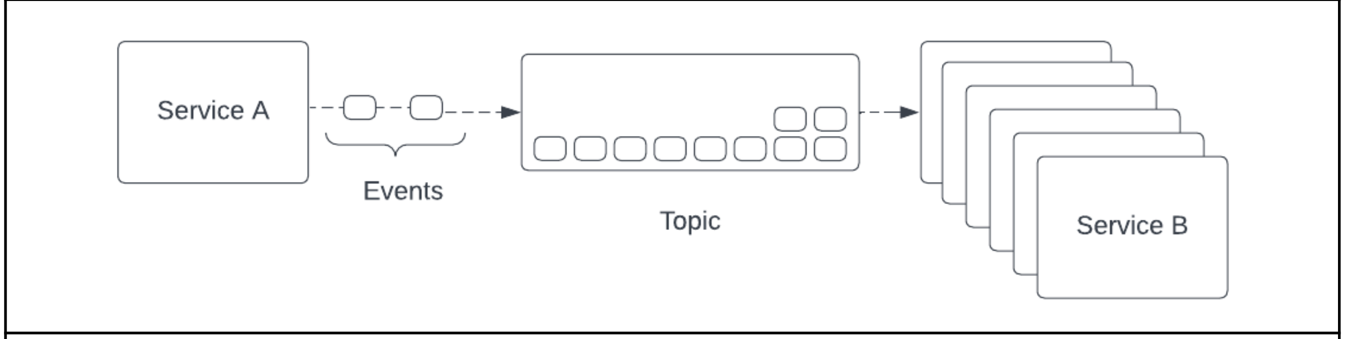


Figure 12: Representation of the work queue pattern where a pool of workers processes messages from a queue

Work queues rely heavily on two main capabilities. The first is the ability to grow and shrink the pool of workers in response to the queue depth to process the backlog of messages in parallel with minimal delay. The second key element of a work queue is guaranteed message delivery and processing. In order to achieve both of these, you need to have individual acknowledgement of messages. This is because with a pool of workers, you cannot guarantee that the messages in a queue will be processed in order. If you have 5 workers processing messages in parallel, it will often be the case that a message that arrived earlier won't finish being processed by a consumer until some number of later messages were processed by a different consumer. If the messaging platform uses a watermarking approach to acknowledgement - that is, once a message is successfully processed you assume that all earlier messages were also successfully processed - then parallel processing introduces the likelihood of message loss.

Let's imagine a situation where two workers are processing messages in parallel. The first worker starts working on the next message in the queue, but for whatever reason processing takes a little while and something goes wrong such that the processing of that message fails. While the first worker is trying to process its message, the second worker picks up a later message and processes it successfully right away, reporting back to the broker that the message is successful. A watermarking strategy would indicate to the broker that it can forget about all messages received before the one that has just been acknowledged. In this case, because the first message didn't fail processing until after the second message was acknowledged, a watermarking strategy would essentially lose this message.

On the other hand, if our messaging platform is capable of handling individual message acknowledgement, this becomes a non-issue. In this same example, the broker will receive an acknowledgement from the second worker noting that it processed its message successfully. It will then receive a failure notification from the first client indicating that the earlier message failed and the broker can then simply attempt to redeliver the message to avoid any message loss.

Kafka

Score: ***1***

As we saw in previous sections, Kafka tightly couples the number of consumers in a consumer group to the topic partitions which make up a topic. This severely limits Kafka's ability to dynamically grow and shrink the number of consumers to accommodate the work queue pattern. Scaling consumers requires a consumer rebalance and halts processing of the topic until that rebalancing process completes.

Likewise, within a partition, Kafka relies on a watermarking approach to acknowledge messages. This design is at the heart of why Kafka is only able to have a single consumer associated with a topic partition at any given time. When the consumer acknowledges a message in Kafka, an offset value is used to tell Kafka the latest message which was successfully processed. When you have a single consumer thread working against a partition, that single consumer is responsible for every message so you avoid the problems that arise when you have multiple consumers working in parallel. Unfortunately, the trade off is that you cannot dynamically add new consumers to the work queue and you have strict upper bounds on the level or parallelism you can achieve without introducing message loss.

Pulsar

Score: ***5***

Pulsar is made to support both a dynamic pool of consumers as well as individual message acknowledgement. With Pulsar, the *Shared Subscription* type achieves the behavior needed for a work queue pattern. You can dynamically assign any number of consumers to a shared subscription and those consumers will work across all of the topic partitions that make up a topic.

At the same time, Pulsar provides support for individual message acknowledgement. This means that regardless of the order in which the messages for a topic are processed by the pool of consumers, any failed message can be redelivered at a later time for reprocessing. To increase throughput, individual messages can optionally be batched. When batching is enabled, when a consumer failure requires a message within a batch to be redelivered, Pulsar supports a feature called batch indexing that allows the specific failed message within the batch to be redelivered without requiring redelivery of the other messages from within the batch that were successfully processed.

Because the pool of consumers can grow and shrink, Pulsar is able to solve the work queue use case in a way that is efficient from a resource utilization standpoint and fits well into a modern cloud native architecture.

RabbitMQ

*Score: **3***

RabbitMQ is able to support a dynamic pool of consumers that work in parallel to process the message backlog in a queue. This is accomplished by setting the exclusivity parameter when the consumer connects to the queue. Likewise, RabbitMQ has configurable acknowledgement modes, one of which allows the equivalent of individual message acknowledgement.

The challenge with RabbitMQ stems from the fact that there is a limit to the size of the queue that RabbitMQ can process. When using the streaming plugin in RabbitMQ to work around these limitations, you end up with similar constraints as Kafka where only a single consumer can process messages from a partition.

Negative message acknowledgement & redelivery

If a subscriber receives a message, but for some reason that message cannot be processed successfully, we want some mechanism by which the consumer can indicate to the message broker that the message processing failed and that the message needs to be re-delivered. For example, in our previous example of a user registering with our site and a notification service subscribing to a registration topic, you can imagine a case where the notification attempts to send an email by establishing a connection with an SMTP (email) server. If for some reason that server is unavailable, the notification service needs some way of telling the message broker that it was unable to process the event and to try sending it again later. We call this ability *negative message acknowledgement* since the consumer is acknowledging to the broker that it received the message, but that processing could not be completed. It probably goes without saying, but when this happens we only want to reprocess the event that failed.

Kafka

*Score: **3***

Negative acknowledgements are a fairly recent feature in Kafka (as of the time of this writing). Negative acknowledgements in Kafka work within Kafka's offset watermark approach to message acknowledgements.

When using the Java KafkaConsumer class, you can indicate that you want to negative acknowledge a message using the nack method:

```java
consumer.nack(offset, 1000);
```

This method takes two parameters. The first is an optional offset, this tells Kafka that you would like to successfully acknowledge messages up to the specified offset. The second mandatory parameter is a delay, in this case 1,000 milliseconds, until you'd like to retrieve the next batch of messages and re-attempt processing.

While this feature is useful, it also requires that all processing for the partition stops for the delayed period before attempting to resume processing.

Pulsar

Score: ***5***

Pulsar supports individual message acknowledgements and provides you fine grained control to negatively acknowledge specific messages without introducing any processing delays or impacting your ability to process other messages in the same topic.

Pulsar provides the negativeAcknowledge method to negatively acknowledge a message:

```java
consumer.negativeAcknowledge(message);
```

By default, Pulsar will use an exponential backoff when attempting to redeliver messages giving the system progressively more time to resolve any transient instability issues before making the next attempt. Eventually, the maximum number of retries will occur and dead letter behavior will take over as covered in the upcoming section on this topic.
As mentioned previously, Puslar also supports batch indexing to allow individual messages to be negatively acknowledged even when Pulsar's built-in batching functionality is utilized to increase throughput.

RabbitMQ

Score: ***5***

RabbitMQ gives consumers the ability to negatively acknowledge either individual messages or batches of messages. It also provides controls to specify whether rejected messages should be re-queued or deleted.

Delayed message delivery

Following on from the ability to negatively acknowledge messages when they can't be processed, it can be useful to indicate that you want some delay to take place when these failure conditions do arise. It's not particularly useful to continuously reprocess messages in many cases. One capability that can be handy in these situations is the ability to tell the broker that you want to delay re-delivery of that message to give the system time to recover and improve the chances that subsequent attempts to process that message will be successful.

Kafka

*Score: **1***
Kafka has no support for delayed message delivery.

Pulsar

*Score: **5***
Pulsar has built-in support for delayed message delivery at scale. Pulsar's implementation allows messages to be delivered after a particular duration of time has passed or once a particular time is reached. When the bucket-based delayed message index tracker is enabled, Pulsar's delayed message delivery can be used for large numbers of messages.

RabbitMQ

*Score: **3***
RabbitMQ supports delayed message delivery through a separate plugin, however there are lower durability and delivery guarantees when using this plugin compared to using the core messaging capabilities in RabbitMQ.

Dead letter routing

There will be cases when even after numerous attempts to process a message, you are still not able to process the message. This could be because the message contains bad data or any number of other things that can go wrong during the normal course of trying to operate software in the real world. When this happens, we want some way to capture those failed messages to either attempt more advanced corrective action or to at least have visibility into which messages could not be processed.

Kafka

*Score: **5***
Kafka has full support for dead letter routing. In Kafka you can specify the dead letter topic where you want failed messages to be delivered along with the number of retry attempts that should be made before giving up and routing the message to the dead letter topic.

Pulsar

*Score: **5***
Pulsar has full support for dead letter routing. In Pulsar, this is configured on the client using a dead letter policy that specifies the dead letter topic to use along with the maximum number of retries that should be attempted. For additional flexibility, Pulsar also allows failed messages to first be written to a retry topic and retried for a specified number of times (with a configurable delay between attempts) before the messages are sent to the dead letter topic.

RabbitMQ

Score: ***5***
RabbitMQ fully supports dead letter routing and allows you to specify a maximum number of attempts before giving up and delivering the message to a dead letter queue.

Guaranteed Ordering

Global ordering of all messages in a topic would be a handy feature; unfortunately, the processing overhead required to offer this capability quickly makes it impractically slow for use cases which require significant scale. This capability is included in our criteria in this document because many people investigating messaging platforms investigate these capabilities, but to-date no platform is able to offer capabilities that significantly stand out in this area. All three of the platforms we're considering are able to support global ordering guarantees only in the case when you're working with a topic that is not partitioned.

Kafka

Score: ***3***
Kafka is able to offer global guaranteed ordering for single-partitioned topics only. Otherise, Kafka can only guarantee that messages within a single partition will be delivered in order.

Pulsar

Score: ***3***
Pulsar is able to offer global guaranteed ordering for non-partitioned topics only. Otherwise, Pulsar can only guarantee that messages within a single partition will be delivered in order.

RabbitMQ

Score: ***3***
RabbitMQ is able to offer global guaranteed ordering by virtue of the fact that topics do not span multiple brokers, creating capabilities similar to both Kafka and Pulsar in this regard. When using the stream plugin, the capabilities fall right in line with Kafka and Pulsar.

Broker-Side Filtering

Broker-side filtering is very useful in use cases where there is a need to control delivery so that only specified messages are sent to consumers, such as when a topic has a large variety of messages and consumers want to consume only messages with specific tags from the topic. This practice became very common for JMS implementations, and without server-side filtering, filtering must be performed on the consumer-side, which can create significant cost implications when many consumers need to read (and then perform post-consumption filtering) from a high throughput topic – thus multiplying the number of reads and filtering-steps for consumers that might only be interested in a small subset of messages. Broker-side filtering addresses this scale problem by allowing messages to be filtered prior to dispatch so that only the messages a consumer is interested in will be delivered to it.

Kafka

Score: **1**
Kafka does not currently support broker-side filtering.

Pulsar

Score: **3**
Pulsar supports broker-side filtering for individual messages. Pulsar's implementation allows a consumer to specify a list of properties, and based on a pluggable entry filter in the dispatcher, filter out messages as desired.

RabbitMQ

Score: **1**
Server-side filtering is not possible with RabbitMQ, and selective consumers are actually an anti-pattern in its current design.

Event Streaming Capabilities

Event persistence and replay

While queueing systems typically prioritize in-memory data storage, streaming systems offer the capability to persist event data on disk for long-term retention. In queueing systems, messages are often evicted from memory or disk once all subscribers have acknowledged them. In contrast, streaming systems anticipate that new consumers may join later and require access to historical event stream data, necessitating the retention of such data for replay purposes.

When evaluating a platform for event streaming capabilities, two very basic features that must be available are event persistence and the ability to replay event streams.

Kafka

Score: ***5***
Kafka was really the pioneer in this area and the entire platform is built to support use cases around event streaming. Kafka uses an approach of an append-only log which is persisted to disk on the event brokers. Kafka was also the first mainstream platform to solve the limitations of previous generation technologies by introducing the concept of partitioned topics that span across multiple broker instances. With these two innovations, Kafka was able to capture and retain large volumes of event stream data and enable playback of event streams.

Pulsar

Score: ***5***
Pulsar had the benefit that it followed in the footsteps of Kafka and was able to improve upon many of the good ideas that Kafka had and to avoid the challenges that early adopters struggled with when they tried to use Kafka at scale. Pulsar also relies on the concept of partitioned topics, but instead of tightly coupling event stream persistence at the broker level, Pulsar opted to use a distributed ledger to persist historical stream data to an elastic storage layer which enabled many of the benefits that are covered throughout this report. Additionally, Pulsar added a subscription type abstraction between the consumer and the broker to support more fine grained message exchange behavior, in essence acting like a queuing platform (when queuing is the desired behavior) and acting like an event streaming platform (when streaming is the desired behavior).

RabbitMQ

Score: ***3***
RabbitMQ was never intended to support event streaming use cases and therefore does not support the idea of playback or long-term persistence of stream data out of the box. In the most recent releases of RabbitMQ, a streaming plugin was added to increase the capabilities in this area.

Cost-effective, theoretically infinite retention

Following on from the ability to persist event data, we would like our event streaming system to support unbounded storage of event data. Ideally, this retention would not require us to use server attached storage and instead to offload historical data onto a blog storage provider or data lake such as AWS S3, Google Cloud Storage, or Azure Blob Storage so that this retention is cost effective.

Kafka

Score: ***1***
One of the unfortunate consequences of Kafka's architecture is that in order to scale storage, which is typically very inexpensive in a cloud architecture, you must also scale your broker nodes which involves much more expensive compute resources. While Kafka theoretically supports infinite retention of stream data, the fact that you need to keep adding additional VMs as you extend your retention periods makes unbounded retention impractical in Kafka.

Likewise, the fact that event data is persisted on a given partition permanently (or at least for the life of the topic) means that if you want to add additional brokers to enable additional history you will need to rebalance the key to partition mapping that occurs. This means that the history for certain keys may be spread across multiple partitions. If you are storing something like clickstream data where your message key is some unique identifier for a user and you would like to replay the clickstream history to process all of that user's actions, it's not straightforward to implement. You may have one consumer attached to the partition with the older history for that key and another consumer attached to the partition with more recent data that was published after the rebalancing occurred.

Pulsar

Score: ***5***
Pulsar's architecture uses a distributed ledger to store historical event data in an elastically scalable storage layer. One of the more interesting features of this storage layer is the ability to offload historical data onto what is referred to as *tiered storage*. When historical data is offloaded, it no longer resides on the server attached storage of the storage nodes. Instead, it can be moved to a low cost blog storage solution like AWS S3 or GCS on Google. Even though the historical data is offloaded, it can still be accessed by the same replay mechanisms provided within Pulsar. In this way, the location of the persisted stream data is transparent from the consumer's point of view.

At the same time, given the low cost of storage on these cloud solutions, you can retain as much history as you would ever want for only a couple cents per GB per month.

Because of this separation, you also avoid any of the challenges around shifting location of historical data for a given message key and more recent data that you can encounter with a platform like Kafka. From Pulsar's perspective, it is happy to deliver (at high throughput) the entire history of a given message key to one or more consumers irrespective of where its history is stored on server attached disks on a storage node or in long term cloud storage.

RabbitMQ

Score: ***1***
RabbitMQ does not support retention of message data without a separate streaming plugin. When using that plugin, you get similar capabilities to Kafka in this requirement area.

Connector support

As you ingest large volumes of event stream data you will often find that you need to push it to a downstream system for additional processing. For example, if you have a clickstream which is capturing every click across your website and mobile app, you may want to push that data into a data warehouse like Google Big Query to enable your reporting team to generate analytics and reports about that data. When considering an event streaming platform, you should ensure that the platform includes a connector framework to both source event streams from upstream systems (for example CDC from a relational database) and sink event stream data to downstream systems (like the just-mentioned BigQuery example).

Kafka

Score: ***5***

Kafka includes a framework called Kafka Connect. Kafka Connect provides a mechanism to create both source and sink connectors. One key architectural feature of Kafka Connect is that the connectors run separate from the Kafka server and are essentially a client framework.

Kafka's connector ecosystem is very large and most products that integrate with streaming/messaging platforms will include a connector for Kafka.

Pulsar

*Score: **3***
Pulsar includes a framework called PulsarIO. PulsarIO provides a mechanism to create both source and sink connectors. One key architectural feature of PulsarIO is that the connectors are meant to run on the Pulsar server. Pulsar's architecture allows for you to elastically scale the number of workers for a given connector as needed to support the volume of events being ingested by a source connector or written to a downstream system by a sink connector. Connectors can be run in Kubernetes mode or in process mode (for bare metal or VM deployments).

While Pulsar's core connector capabilities are complete and capable, the ecosystem of Pulsar connectors is not as widely supported as platforms such as Kafka.

RabbitMQ

*Score: **1***
RabbitMQ does not support a connector architecture. However, both Kafka and Pulsar include mechanisms to ingest data from RabbitMQ via a source connector and write data into RabbitMQ via a sink connector.

Event processing

Event *stream* processing is a large and complex subject and it is beyond the scope of this document to go into detail around the many facets of that topic. Purpose-built platforms like Apache Flink, Google DataFlow and Hazelcast provide highly sophisticated robust platforms just to deal with the intricacies of event stream processing. However, you will sometimes have lightweight processing requirements that your event streaming platform can provide. In this paper, we refer to this capability as *event processing* and treat it separately from event *stream* processing. Event processing allows you to modify events in-stream without relying on an external platform. Examples of use cases where event processing is useful include things like removing PII or other sensitive data from an event before pushing it into a downstream data store or making an API call to enrich the event payload, for example calling the Google Maps API to geocode a lat/lon into an address.

Kafka

*Score: **5***
Kafka includes a framework called Kafka Streams or KStreams. KStreams listen to a topic and provide lightweight event processing capabilities. KStreams also includes sophisticated stateful

capabilities such as KTables which give you database-like views of your topic data. Generally with KStreams you will have an input topic your KStreams application will listen on and an output topic your KStreams application will write the transformed stream to.

Pulsar

*Score: **3***

Pulsar includes the Pulsar Functions framework. Pulsar functions are lightweight compute instances that Pulsar manages to perform simple processing tasks. They're designed for low-latency and support Java, Python, and Go implementations. Because they are managed by Pulsar, unlike KStreams, they don't require you to manage an external environment for hosting the processing; Pulsar handles the processing for you and can relocate functions across function workers as needed to balance resource consumption without downtime. Pulsar functions include some basic state management, but does not provide the more advanced capabilities such as those in Kafka's KTables.

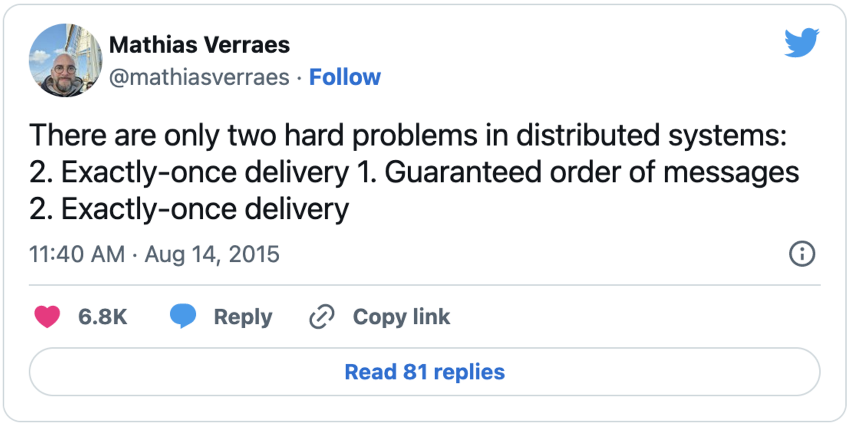
RabbitMQ

*Score: **1***

RabbitMQ does not provide an analog to either Kafka Streams or Pulsar Functions.

Delivery/Processing Guarantees

In an ideal world, messaging platforms would provide exactly-once processing capabilities. Unfortunately, what sounds like a reasonable goal turns out to be an unsolvable problem in distributed systems. To quote Mathias Verraes:



Mathias Verraes
@mathiasverraes · **Follow**

There are only two hard problems in distributed systems:
2. Exactly-once delivery 1. Guaranteed order of messages
2. Exactly-once delivery

11:40 AM · Aug 14, 2015

To understand why exactly-once processing is unsolvable, you can dig into the well known [Two Generals Problem]() which is covered in almost every introductory computer science class. This exact problem manifests itself in messaging systems. For instance, message publishers can lose connectivity to the broker after they sent a message, but before they received confirmation from the broker, thus needing to decide whether to resend the message or not. Likewise, brokers could fail to get a message acknowledgement from the consumer and be in a similar predicament needing to decide whether to re-deliver a message that the consumer might have already processed. In “happy path” processing, which represents the normal steady state of a message system, these issues don’t arise and messages get processed exactly once. However, when failure conditions arise you need to make sure your messaging system is able to guarantee delivery no matter what happens.

Message Deduplication

Situations can occur where your message producer could generate duplicate messages for the same event. Say that some exception occurs which leaves the message producer in a situation where they are unsure if the broker accepted a message they attempted to publish. In these cases, it’s typically preferable for the producer to send a possible duplicate message. However, we don’t necessarily want to process the same message multiple times just because the producer sent two copies of it. If your messaging platform is able to provide deduplication capabilities, it can eliminate the need for your message consumers to worry about handling these situations explicitly.

Kafka

Score: ***5***
Kafka relies on an approach they call idempotent producers to solve for deduplication. Kafka will add a producer ID, sequence number, and generation to each message. The producer ID is a unique identifier for the producer, and the sequence number and generation are used to track the order of messages. When a producer sends a message, the broker stores the message in a map keyed by the producer ID and sequence number. If the broker receives the same message again, it will check the map to see if the message has already been stored. If it has, the broker will discard the message. This ensures that each message is only stored once, even if it is sent multiple times by the producer.

Pulsar

Score: ***5***
Message deduplication works very similar in Pulsar to the way it works in Kafka. Pulsar uses a combination of the producer name and a sequence number to uniquely identify a message in a topic. If some failure scenario occurs which results in the producer republishing a message due to it not receiving an acknowledgement from the broker (due to network partitioning or another failure condition), the broker will detect that a duplicate message has been received and take steps to prevent the duplicate message from being written to the topic, achieving idempotency from the producer’s point of view. Additionally, when message batching is enabled to increase throughput, Pulsar can utilize batch indexing to prevent duplicate delivery of acknowledged messages when a failure requires a message within a batch to be reprocessed.

RabbitMQ

Score: ***3***

RabbitMQ supports deduplication via either the Stream Plugin or RabbitMQ Streams API to create a deduplication queue. Without leveraging a separate plugin with RabbitMQ, you will generally need to implement either idempotent consumers or account for deduplication on the consumer side.

Transactional Support

Transaction capabilities in messaging platforms aim to achieve the same atomicity, consistency, isolation and durability guarantees as transaction capabilities in databases. One common place where you will see transactions used is when a message is processed and the result of that processing results in a new message which is published to a different topic. In this case, you may need the acknowledgement of the original message and the publishing of the new message to be either all committed, or all rolled back so processing can be retried. Without a transaction coordinating the state of these related operations, you could end up in situations where the original message is considered acknowledged, but publishing the new message failed and you effectively lose a message. Or the reverse could happen where the acknowledgement failed, but the publish was accepted – causing duplicate processing and duplicate copies of the new message.

Transaction managers are not immune to the two generals problem mentioned above, but they protect against several edge cases that can occur, and for situations where you need better processing guarantees, they can offer you major improvements. Of course, the penalty for using transactions is exactly what you would expect - increased latency, increased resource utilization and lower performance. For those reasons, you generally want to be very selective about the use cases where you choose to leverage transactional support.

Kafka

Score: ***5***
Kafka's transaction support allows producers to send multiple messages to multiple topics, and guarantee that all of the messages are either committed or aborted as a single unit of work. This is done by grouping multiple calls to send() into a transaction. Once a transaction is started, the producer can call commitTransaction() or abortTransaction() to complete it. Consumers configured with isolation.level=read_committed will not receive messages from aborted transactions.

If the producer fails before committing the transaction, the transaction is aborted automatically. If the producer commits the transaction, all of the messages in the transaction are guaranteed to be delivered to all of the consumers that are subscribed to the topics.

Using transactions in Kafka, you can get close to exactly-once message delivery.

Pulsar

Score: ***5***

Apache Pulsar's transaction support allows producers and consumers to send and receive multiple messages to multiple topics, and guarantee that all of the messages are either committed or aborted as a single unit of work. This is done by grouping multiple calls to send() and receive() into a transaction. Once a transaction is started, the producer and consumer can call commit() or abort() to complete it.

If the producer or consumer fails before committing the transaction, the transaction is aborted automatically. If the producer or consumer commits the transaction, all of the messages in the transaction are guaranteed to be delivered to all of the consumers that are subscribed to the topics.

RabbitMQ

Score: ***1***

Currently, RabbitMQ does not have transactional capabilities that span across multiple queues.

Unacknowledged Message Retention

Especially for topics with a relatively short retention time (no retention, retain for minutes, or even an hour) you should consider what will happen if a consumer fails for a period of time that exceeds that retention period. Most often, you want to ensure that no data loss occurs in these cases so it is important that you can configure your messaging platform to only purge *unacknowledged* messages from the topic when the retention period is met. In order to do this safely, you need many of the architectural requirements we already covered such as independently scalable compute and storage and theoretically infinite message retention.

Because there may be situations where you wish to purge the unacknowledged messages from the topic, that capability should be available to you as well.

Kafka

Score: ***1***

In Kafka, there is no distinction made between unacknowledged and acknowledged messages from a retention standpoint. This means that if a consumer is unavailable for longer than the configured retention period, you can end up losing messages.

Pulsar

Score: ***5***

By default, Pulsar will only purge unacknowledged messages when the retention period is met. If you have a consumer which fails for longer than the retention period, once it recovers, Pulsar will deliver all of the unacknowledged messages to the consumer first, then purge the events which are outside the bounds of the retention period. In this way, Pulsar is able to offer very strong guarantees against data loss in these conditions. Pulsar also offers policy-based TTL options to automatically acknowledge messages after a period of time if desired for a topic or namespace of topics.

RabbitMQ

Score: ***1***
RabbitMQ provides a TTL (time to live) setting which can be used to determine how long messages will stay on the queue, but once that TTL is reached the messages will be removed if they have not been processed. Processed messages are deleted almost immediately which is understandable given it's core functional purpose is to serve as an ephemeral message bus.

Failover & DR

For any sort of mission critical use cases you may have, you want to prepare for the eventuality where you lose an entire data center, cloud region, or availability zone. In these cases there are three important capabilities that you need to ensure that processing will continue with minimal interruptions.

Local Failover

Within a data center or cloud region, your architecture needs to provide some level of redundancy at a component level. For example, if a broker fails or becomes partitioned from the network, you want another broker in your deployment to take on the responsibilities of fulfilling requests from publishers and subscribers until that broker becomes available again. This type of failure should be a non-event for your messaging platform to handle. You should expect that your messaging platform is able to both continue processing in these situations as well as recover in these situations without any major consequences for your producers and consumers.

Kafka

Score: ***5***
Kafka is built to be resilient in the event of a component failure. Kafka provides a configurable acknowledgement and write quorum - that is, you can configure how many brokers need to acknowledge that an event has been received and stored in memory before telling the producer it was successfully received. You can also configure how many brokers need to write the event data to physical disk (fsync) before responding to the producer to confirm that the message has been received.

Generally speaking, you want to configure ack quorum to at least 2 brokers to ensure you can withstand a machine failure without losing data. For higher durability guarantees you also need to configure the write sync to 2 or more brokers.

Pulsar

Score: ***5***

(Same as Kafka) Pulsar is built to be resilient in the event of a component failure. Pulsar provides a configurable acknowledgement and write quorum - that is, you can configure how many brokers need to acknowledge that an event has been received and stored in memory before telling the producer it was successfully received. You can also configure how many brokers need to write the event data to physical disk (fsync) before responding to the producer to confirm that the message has been received.

Generally speaking, you want to configure ack quorum to at least 2 brokers to ensure you can withstand a machine failure without losing data. For higher durability guarantees you also need to configure the write sync to 2 or more brokers.

Pulsar also offers a proxy component that simplifies negotiation of the connection for producers and consumers when broker failover occurs.

RabbitMQ

*Score: **5***

RabbitMQ supports the concept of Quorum Queues which provides similar resiliency to failovers compared to Pulsar and Kafka.

Georeplication

In order to build an architecture that is resilient to losing an entire cloud region or data center, you must not only anticipate individual component-level failures, but also cases where you need to shift processing of your entire deployment from one location to another. This can only happen if messages that are published and accepted by a broker get replicated to your failover region. If you don't have these capabilities, you'll find that these failure situations will leave you in a highly inconsistent state where messages that were in flight remain in an unknown processing state until the region is repaired and resumes normal operation.

Kafka

*Score: **5***

Kafka has built-in support for a tool called Mirror Maker 2. Mirror Maker allows you to replicate cluster configuration, topic event data, consumer offset information, between two or more Kafka clusters. Replicated event data will not have guaranteed ordering of partition data across multiple clusters.

Pulsar

*Score: **5***

Pulsar has built-in georeplication that can be configured at a namespace level, giving you slightly more streamlined control over which topics to replicate than with Mirror Maker. Pulsar also has the concept of a *replicated subscription* which ensures that the individual message acknowledgement data for a given subscription is also replicated across clusters. Just like

Kafka, Pulsar supports the ability to replicate across two or more clusters and doesn't make guarantees about event ordering between clusters.

RabbitMQ

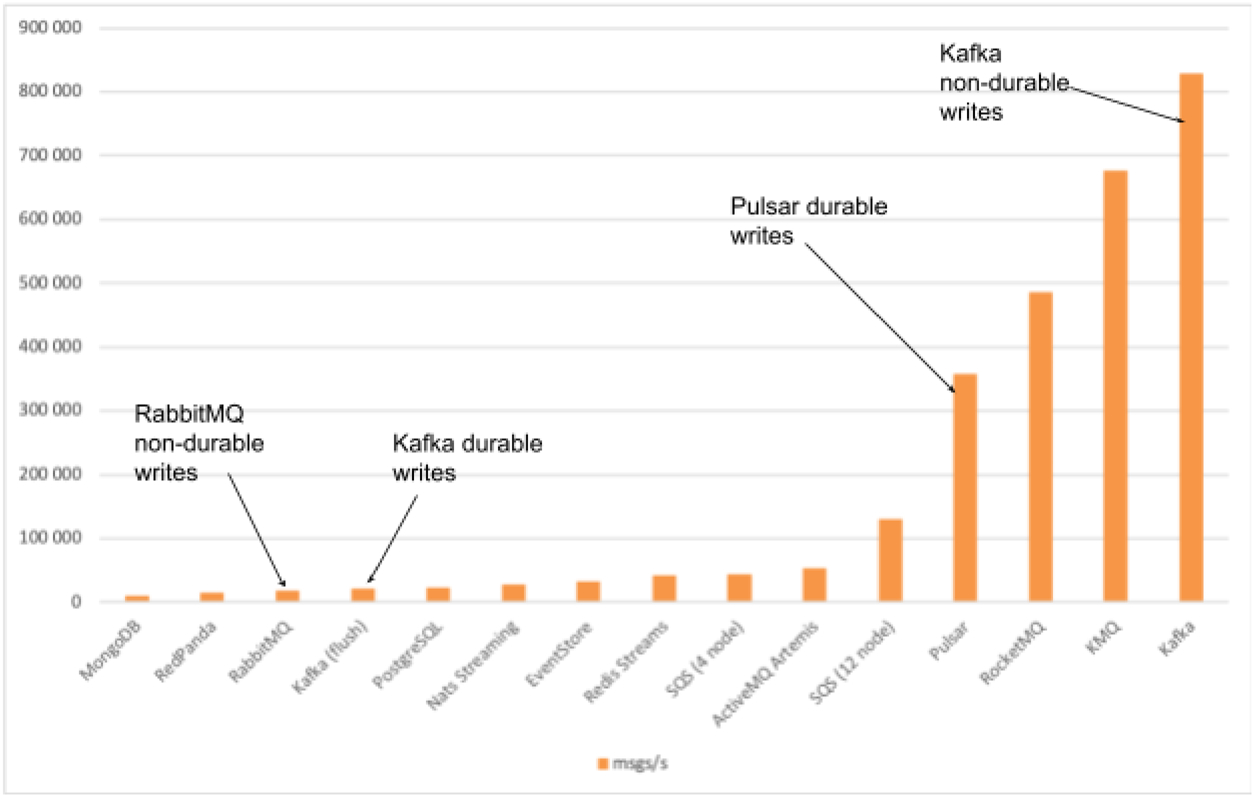
*Score: **3***

RabbitMQ supports a federated deployment of multiple RabbitMQ brokers/clusters. In this configuration, message information can be delivered from an exchange on one broker and copied to another. In general you could characterize these capabilities as "better than nothing" but incomplete compared to the capabilities of Pulsar's georeplication and Mirror Maker.

Message Publishing Performance

An inherent challenge you will face when adopting a microservices architecture is performance. Compared to a monolithic architecture where your application shares memory and communicates via function calls on the same machine, the additional overhead of network calls can be orders of magnitude slower. Therefore, when we're using an event driven approach, it is critical that our messaging infrastructure add as little additional latency as possible. When tuning performance, you will generally have a trade off between durability guarantees and raw throughput. Therefore, we will consider performance from two perspectives.

Unbiased benchmark data is hard to come by. Vendors almost always stack the deck in their favor when showing how they compare to competitive products. Use cases are cherry picked to align with best-case scenarios, highly optimized and tuned configurations are compared against stock configs for competitors, and the unrealistic nature of the tests is downplayed. For that reason, this section uses benchmark data from Software Mill, a software engineering company with no products in this space who conducted rigorous performance testing to simulate real world conditions and requirements as much as possible. The following graph shows their results with labels and arrows added for this report:



Durable write performance

When you need to build an architecture that's optimized to prevent data loss, you want to ensure that your messaging platform is highly performant using a configuration that ensures each message that is published is written to disk before communicating back to the publishers that each message has been accepted. Since the earliest days of messaging system benchmarks, unscrupulous products and open source projects disable durability guarantees in order to show how performant their products are, so it's important to always ask the question when looking at benchmarks if the messages are written to disk or simply left in memory.

Kafka

Score: ***1***

When configured for durable writes and guaranteed prevention for data loss, Kafka saw a roughly 90% decline in performance in the Software Mill benchmark tests. Generally speaking, this puts Kafka on par with RabbitMQ and most other messaging platforms.

Pulsar

Score: ***5***

Pulsar's architecture is optimized for situations where you want extremely high throughput with zero risk of data loss. In this area, Pulsar stands above every other option in the market.

RabbitMQ

Score: ***1***
RabbitMQ does have the ability now to perform durable writes, but only at a very small scale compared to other messaging platforms.

Non-durable write performance

There are certainly some cases where you're willing to accept some risk of data loss in exchange for raw performance on a per-broker basis. In these cases, it's good to know that your messaging platform is able to provide you with tunable trade offs when the durability of messages isn't important to you.

Kafka

Score: ***5***
There is no doubt about it, if you need extremely high throughput and are willing to make tradeoffs around durability and risk some message loss, Kafka is able to offer performance second to none for this use case.

Pulsar

Score: ***3***
Pulsar supports non-durable topics using non-persistent topics. These topics can be partitioned just like persistent topics and will generally perform with lower latency and higher throughput than persistent topics. However, since this use case was not tested in the 3rd party, independent benchmark reference above, we're assigning Pulsar a score of 3 in this category.

RabbitMQ

Score: ***1***
RabbitMQ non-durable write performance is significantly behind other competitive products.

Interoperability

Schema Support

In order for messaging platforms to support multiple programming languages, you'll usually find that under the covers the message data is represented internally by the messaging platform as an array of bytes. From a consumer's point of view, this is not always convenient. We need to know how to serialize those bytes into meaningful data structures that can be more easily processed using code.

The standard way to support this requirement is using a schema. Schemas describe the structure of the data being published to a messaging system and allow publishers to serialize objects or other complex (and simple) data structures into a byte array representation and for consumers to deserialize those byte arrays into meaningful data structures that can be used during message processing.

Kafka

Score: ***1***
Kakfa stores messages as a raw byte array with no in-built support for coordinating serialization and deserialization of those bytes between the producer and consumer. This leaves Kafka users with a challenge of figuring out how to serialize/deserialize messages from each topic. As the number of topics in use throughout your Kafka deployment grows large, this challenge becomes more pressing and more complex to solve.

Pulsar

Score: ***5***
Pulsar stores messages as a raw byte array and includes a schema catalog as a core part of the platform. With Pulsar, both schema mapping for a given topic as well as schema versioning is supported. When clients connect to Pulsar, they can also retrieve a schema in Avro, JSON or primitive formats to automate the process of deserializing the raw byte array. As your deployment grows, it is simple for producers and consumers to ensure interoperability between producers and consumers by using the schema registry to validate that messages adhere to the stated schema and for consumers to quickly turn raw bytes into meaningful data structures. This design, for example, enables producers and consumers across multiple languages to standardize on Avro schemas, use code generation to generate type definitions, and evolve schemas to support the desired level of compatibility for your messaging flows.

RabbitMQ

Score: ***1***
Similar to Kafka, RabbitMQ does not have any support for schema management. You must create your own solution for serializing and deserializing raw message bytes into a useful format.

Administration

Multi Tenancy

Multi-tenancy is a software architecture in which a single instance of a software application serves multiple tenants. Each tenant provides logical, and sometimes physical, separation of the entities and artifacts used by the platform. In the context of this document, those entities would be things like topics in Kafka, or Pulsar Functions in Pulsar, or exchanges and queues in RabbitMQ. Tenants typically have their own sets of users who can perform various operations within them and provide a logical container to separate work by team, business unit, customer and so on.

Kafka

*Score: **1***
Kafka lacks multi-tenancy support. The Kafka documentation attempts to portray access control lists as equivalent to multi-tenancy, but functionally fails to provide anything resembling true multi-tenancy. For example, there is no operation in Kafka to support creating a new tenant. There is no mechanism to apply configurations to a tenant or to associate physical resources like brokers to a specific tenant for resource isolation purposes. One common problem this leads to with Kafka is cluster sprawl. Since there is no multi-tenant support, many organizations that use Kafka default to a cluster-per team or a cluster-per project approach. This can quickly become massively expensive and complex to manage.

Pulsar

*Score: **5***
Pulsar has multi-tenancy support as a core part of its architecture. You can configure Pulsar to share physical resources like bookies and brokers across multiple tenants, or you can dedicate specific nodes to only certain tenants. This prevents noisy-neighbor problems and allows you to more efficiently utilize resources. Pulsar also has an additional container called a *Namespace* which enables you to organize resources within a tenant. With Pulsar you are able to set access control rules to limit access at a tenant, namespace or topic level.

Between support for multi-tenancy and the elastically scalable architecture of Pulsar, organizations that adopt Pulsar are generally able to support an enterprise wide deployment with fewer clusters and an overall smaller footprint compared to other platforms like Kafka.

RabbitMQ

*Score: **3***

RabbitMQ has no support for multi-tenancy, but it does satisfy some of the requirements of multi-tenancy through the use of its virtual host feature which allows for grouping and separation of resources.

Upgrades / Compatibility Of Clients

Product and version management is another key item to consider when choosing a messaging platform. This includes upgrades to client-libs and compatibility with existing clients.

Some items for considerations:

- Need for downtime for upgrades
- Support of cloud-native upgrade patterns like “blue-green” environments and switch
- Ability to down-upgrade

Kafka

Score: **3**

Kafka supports rolling-upgrades but lacks a well-documented procedure or tasks for an upgrade. Additionally, it does not support the “blue-green” pattern for upgrades.

Pulsar

Score: **5**
Pulsar has a well documented upgrade process, including down-grade options. These steps and tasks are detailed on the Apache Pulsar website. Options include “rolling upgrades” for near-zero downtime/disruption of your clients.

Additionally, the latest Pulsar version (3.0.0) and Pulsar clients support the “blue-green” upgrade pattern of upgrading and fall-back options.

RabbitMQ

Score: **3**
RabbitMQ upgrades are well documented and include steps and tasks considerations for upgrading. It supports rolling upgrades via “node maintenance” CLI commands. It doesn’t support the “blue-green” upgrade pattern.

Observability Tools / Dashboards

Messaging systems and platforms can be complex. Monitoring, metrics, and observability are key items for considerations. Real-time metrics and dashboards are critical components to ensuring the health of any messaging platform.

Kafka

Score: **5**
Kafka also has built-in metrics which are exported to external tools, like Prometheus and Grafana for monitoring and alerting. It also has several GUIs and CLI available for administrations and visibility of the Kafka cluster.

Pulsar

Score: **5**
Pulsar has built-in metrics which are easily exported to monitoring and alerting tools, like Prometheus and Grafana. Pulsar has many available web-based GUI for administrations, monitoring, and visibility into Pulsar. Some GUIs and CLIs are Pulsar Console, Pulsar Manager, and CLIs like Pulsar-Shell, and Pulsar Admin.

Additionally, Pulsar has a tool, Pulsar Heartbeat, that can create synthetic messages/transactions and observe and report end-to-end latency. This is a very powerful tool for observability and monitoring of Pulsar infrastructure, with full metrics that can be exported to Prometheus/Grafana tools

RabbitMQ

Score: **3**
RabbitMQ (version 3.8 or higher) has built-in support for Prometheus and available Grafana dashboards for basic visibility and monitoring. Additional “plug-ins” must be installed and set up for management and enhanced observability of more complex RabbitMQ topologies.

Governance

Role Based Access Control (RBAC)

Kafka

Score: **3**
Kafka does not support multi-tenancy, so RBAC in Kafka is limited to cluster-level and topic-level controls. Companies that desire greater isolation or management control must create and manage separate clusters per each domain they wish to isolate (e.g. per project, per app, or per team), increasing the operational burden and complexity. Additionally, permissions must also be managed for consumer groups, which are used to read specific partitions across topics. So, for a given flow, permissions must be governed not only for the specific consumers but also for any consumer groups they are members of.

Pulsar

Score: **5**
Pulsar enables role-based access control in a way that leverages the built-in multi-tenancy features. Access control can be implemented at the tenant, namespace, or topic levels of granularity, and you can define roles using the conventions that work best in your organization. Permissions can be isolated to specific domains or shared between domains, and domains can represent one or more topics and/or namespaces.

Pulsar additionally provides a pluggable authorization interface that enables greater flexibility for use cases that require custom authorization strategies.

Administrative controls can also be governed at the tenant and cluster levels.

RabbitMQ

Score: **3**
Rabbit’s access control is similar to Kafka’s but also provides built-in support for LDAP.

Content encryption

Content encryption enhances the guarantees provided by transport encryption (SSL/TLS) and hardware encryption (e.g. using encrypted hard drives) by fully encrypting the message content before a message is produced so that if intercepted, it will be impossible to decipher for any consumer that does not have the required key (symmetric or asymmetric, depending on how it's configured). Content encryption is required for the highest security guarantees and is particularly important when handling Personally Identifiable Information (PII) and for enabling cryptographic-shredding for GDPR compliance.

Kafka

Score: **1**
Kafka provides no built-in support for content encryption, so in order to implement it, you must manage content encryption on your own.

Pulsar

Score: **5**
Pulsar provides several options to implement content encryption.

1. Pulsar provides a cryptographic interface that enables built-in symmetric and asymmetric encryption strategies to provide full content encryption between producers and consumers.
2. Pulsar has a pluggable client interception layer that can be leveraged to provide additional control over how content is encrypted and decrypted between producers and consumers, such as for implementing field-level encryption.
3. Pulsar functions also support a pluggable interface for secret management, enabling Pulsar functions to integrate with a 3rd party KMS of your choice for secure access to secrets within functions, such as those used to encrypt content before sending to consumers.

RabbitMQ

Score: **1**
RabbitMQ does not provide built-in support for content encryption.

Policy-Based Data Management

From an operational standpoint, it's important to allow operations to scale to meet your requirements. If the number of Full Time Employees (FTE's) required to manage your cluster size increases with every new workload, then your operational costs and complexity will not scale. Policy-based data management enables rules to be applied to partially automate governance of your data platform.

Kafka

Score: **3**
As Kafka lacks built-in multi-tenancy, all configuration must happen either per individual topics or through cluster-wide settings. An unfortunate consequence of this design is that this feature gap

contributes to cluster sprawl because organizations may need to create separate clusters to satisfy specific governance requirements, such as around security and compliance.

Pulsar

Score: **5**
Policy-based management is where Pulsar's multi-tenancy really shines. Policies can be applied to different parts of the topic hierarchy, such as at the namespace or topic levels, that govern quotas, backlog behavior, retention, message expiration (TTLs), replication, security, and other parameters. When applied to a namespace, these policies are automatically applied to any new topics created within the namespace, allowing administrators to configure top-level rules and let users do what they need without requiring additional intervention.

RabbitMQ

Score: **3**
RabbitMQ supports regular expression based policy mapping. Although this approach does allow policies to be applied to multiple queues, queues must follow careful naming conventions to ensure policies are applied correctly. Additionally, regular expressions can become hard to read, depending on how they are used.

Topic Compaction

Topic compaction is used to prune older entries from a topic based on your specific use case. In some use cases, consumers want to only consume the latest messages for a given key or partition (without needing to read through a large amount of retained historical data). Topic compaction enables consumers to accomplish this task by consolidating messages on the server-side to only the latest messages.

Kafka

Score: **3**
Kafka supports compaction as a destructive operation on a per-partition basis. As partitions are tightly coupled to brokers, an advantage of compaction is that it frees disk space from the broker. However, a consequence of this design is that compacted data is non-recoverable.

Pulsar

Score: **5**
Pulsar supports compaction on a per-key basis. In Pulsar, compacted messages are stored on a separate topic, enabling consumers to either consume the full retained history of a topic or only the compacted, latest values. As Pulsar decouples partitions from message storage, users don't need to worry about where the messages are physically located when managing compaction, nor do they need to worry about how the consumers are grouped together to support the desired throughput. In Pulsar, compaction is a non-destructive operation, so you can enable compaction to improve read performance without losing retained message history.

RabbitMQ

Score: **1**
RabbitMQ doesn’t support this type of message compaction.

Community & Non-functionals

Open Source Ecosystem

The ecosystem surrounding an open source technology is important when making a decision about which product to use because it can provide a number of benefits, such as community support, training and resources, integration with other products, and security and stability. A strong ecosystem can make the product more valuable and easier to use. When evaluating an open source technology, it is important to consider the size and activity of the community, the availability of documentation and resources, the integration with other products, and the security and stability. By considering these factors, you can make an informed decision about which open source technology is right for you.

Kafka

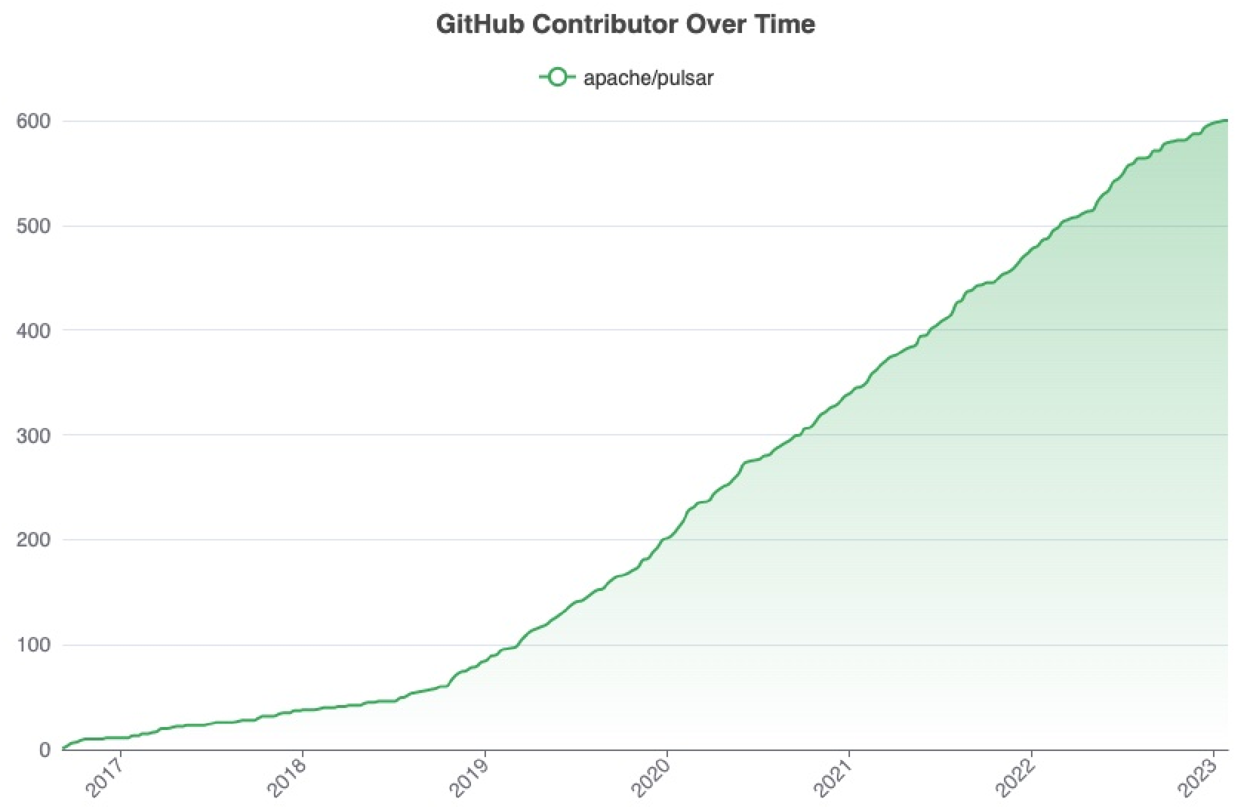
Score: **5**

Kafka is an extremely popular event streaming platform. On GitHub, the main Apache Kafka repository has over 25,000 stars and 12,500 forks. According to GitHub, there are just over 1,000 developers who have contributed to the Apache Kafka project. These are all significant indicators of the engagement that the community of developers and users has with the project. Additionally, there is a huge ecosystem of connectors available for Kafka as well which makes it easy to integrate Kafka with many other products and tool you may be using.

Pulsar

Score: **3**

Pulsar is growing in popularity, but still lags behind platforms like Kafka and RabbitMQ in total ecosystem. On GitHub, the main Apache Pulsar repository has over 12,000 stars and 2,500 forks. There are just over 600 developers who have contributed to the Apache Pulsar project. While the community size is smaller, it has been growing rapidly in recent years:



RabbitMQ

Score: **3**

RabbitMQ has a respectable sized community although it is also smaller than Kafka. On GitHub, the main RabbitMQ server repository has almost 11,000 stars and just under 4,000 forks. While RabbitMQ does not have a connector ecosystem like Kafka and Pulsar, RabbitMQ is widely supported via integrations in other platforms. Both Kafka and Pulsar support connectors to get data from and write data to RabbitMQ for instance.

Availability of Skillset

Kafka

Score: **5**
On LinkedIn Recruiter, there are over 330k people who claim to have Apache Kafka expertise.

Pulsar

Score: **3**

On LinkedIn Recruiter, there are thousands of individuals who state that they have experience with Apache Pulsar. The fundamental concepts and APIs of Apache Pulsar are quite similar to other streaming platforms such as Kafka, or messaging platforms like RabbitMQ. This makes it relatively easy to find individuals with a general understanding who can quickly adapt to Pulsar. However, locating candidates with extensive experience in utilizing and managing Pulsar might prove to be challenging.

RabbitMQ

Score: **5**
There are over 100k people on LinkedIn Recruiter that claim to have RabbitMQ experience.