Tampere University

Juuso Nuikka

# Comparison of Cloud Native messaging technologies

Faculty of Information Technology and Communication Sciences (ITC)
Master's thesis
April 2021

# Abstract

Juuso Nuikka: Comparison of Cloud Native messaging technologies
Master's thesis
Tampere University
Master's Degree Programme in Software Development
April 2021

In a large distributed system, managing outgoing and incoming communications is a complex task due to a large amount of connections. The data flowing inside the system is growing along with the system and manually managing them is not efficient after a certain point. Thus, there is a need for special software to simplify the connections in the system. The special software developed for this purpose is commonly referred to as message-oriented-middleware (MOM).

This paper will compare three different MOMs in the form of a literature review, Apache Kafka, Apache Pulsar, and RabbitMQ. These MOMs are compared based on predefined characteristics. These characteristics are important for a network management system running in a Cloud Native environment. These characteristics are consumer-producer patterns, scalability, throughput, reliability, security, and backward compatibility.

This paper's result is that Apache Kafka remains the primary choice as a MOM for systems demanding high throughput for its wide community adoption and mature technology. Apache Pulsar is Apache Kafka's most significant competitor in this area because it can outperform Apache Kafka in many performance-related characteristics. Apache Pulsar's community is not near Apache Kafka's, and its technology is less mature. RabbitMQ is found to be the best choice when data safety and reliability are a primary requirement.

**Keywords:** Cloud Native, Message-Oriented-Middleware, Message broker, Event broker, Apache Kafka, Apache Pulsar, RabbitMQ, comparison, consumer-producer pattern, scalability, throughput, reliability, security, backward compatibility

# Tiivistelmä

Juuso Nuikka: Viestinvälitysteknologioiden vertailu Cloud Native-ympäristössä
Diplomityö
Tampereen yliopisto
Tietotekniikka, DI
Huhtikuu 2021

Suuren hajautetun järjestelmän sisäisen ja ulkoisen viestinnän hallinta on monimutkaista yhteyksien suuren lukumäärän vuoksi. Systeemissä liikkuva datan määrä lisääntyy järjestelmän kasvaessa, eikä näiden yhteyksien hallinta ole enää tietyn pisteen jälkeen manuaalisesti tehokasta. Siksi tarvitaan tähän tarkoitukseen tehtyjä erityisiä sovelluksia yksinkertaistamaan systeemin kommunikointia. Tähän tarkoitukseen erikoistuneita sovelluksia kutsutaan yleisesti termillä viestinvälitykseen suuntautuneet väliohjelmistot.

Tässä työssä tutkitaan kirjallisuuskatsauksen muodossa kolmea eri viestinvälitykseen suuntautunutta väliohjelmistoa: Apache Kafka, Apache Pulsar ja RabbitMQ. Tässä työssä väliohjelmistoja vertaillaan etukäteen määriteltyjen ominaisuuksien osalta. Nämä vertailtavat ominaisuudet ovat Cloud Native-ympäristössä toimivan verkonhallintajärjestelmän kannalta tärkeitä ominaisuuksia. Nämä ominaisuudet ovat kuluttaja-tuottajamallit, skaalautuvuus, läpimenonopeus, luotettavuus, turvallisuus ja taaksepäin yhteensopivuus.

Työn lopputuloksena on, että Apache Kafka pysyy jatkossakin pääasiallisena valintana korkeaan suoritustehoon tähtäävänä ratkaisuna sen suuren käyttäjäyhteisön ja luotettavan teknologiansa vuoksi. Apache Pulsar on Apache Kafkan kovin kilpailija, sillä se kykenee voittamaan Apache Kafkan monissa suoritustehoon liittyvissä ominaisuuksissa. Apache Pulsarin käyttäjäyhteisö ei ole kuitenkaan Apache Kafkan tasoa, sekä teknologia on paljon uudempi. RabbitMQ:n todetaan olevan paras valinta silloin, kun suositaan datan turvallisuutta ja luotettavuutta.

**Avainsanat:** Cloud Native, viestinvälitykseen suuntautunut väliohjelmisto, viestivälittäjä, tapahtumavälittäjä, Apache Kafka, Apache Pulsar, RabbitMQ, vertailu, kuluttaja-tuottajamalli, skaalautuvuus, läpimenonopeus, luotettavuus, turvallisuus, taaksepäin yhteensopivuus

# Preface

First of all, I want to thank Nokia for giving the topic and helping me to achieve graduation in this spring semester by allowing me to write this partly during work hours. Also, I want to thank University of Tampere and my supervisors for guiding me during this project. I want also to thank my parents for supporting me.

Urjala/Tampere, 4.4.2021

Juuso Nuikka

# Contents

# 1 Introduction

Message-oriented middleware (MOM) is software that handles exchanging messages between different parts of a distributed system. The message-oriented middleware term includes event brokers and message brokers. A message broker is a software, that manages communications between senders and receivers in the system. An event broker is an evolved version of a message broker. The event broker maintains all the data in its storage in an immutable replayable data structure.

There are many possible use cases for a MOM. The best choice as a MOM always depends on the use case. Possible use cases are, for example, financial transactions, website order tracking, and event sourcing. The first thing is to consider whether there is a need for a message-driven message broker or an event-driven event broker. This decision is depending on overall system complexity. If the system is a message-driven system with simple consumer-producer relationships, then a traditional message broker or message queue can be enough. In case the relationship is not clear due to architectural complexity, then an event broker is preferred.

This work aims to make a comparison of MOMs in the form of a literature review. The comparison focuses on the essential characteristics needed in a cloud native network management system (NMS). These characteristics are: high throughput, implemented consumer-producer patterns, and supported security, reliability, scalability mechanisms, and backward compatibility. The chosen MOMs are Apache Kafka, Apache Pulsar, and RabbitMQ. A commonly used messaging solution in the industry is Kafka, but also aforementioned MOMs are capable of implementing the same kind of functionality. These MOMs were chosen for the research based on knowledge gathered from the industry of the best Kafka alternatives. Other alternatives also were considered, such as RocketMQ and NATS Streaming, but they were excluded because there was very little research. Also, including them would have enlarged the scope of this research too broad.

This paper aims not to find an alternative for Kafka but rather to gain knowledge of MOMs for cloud native environment. The aim of the research and the essential characteristics of the NMS leads to the following research questions:

- *Which consumer-producer patterns do Kafka, Pulsar, and RabbitMQ implement?* ($RQ_1$).
- *How do scalability mechanisms differ in Kafka, Pulsar, and RabbitMQ?* ($RQ_2$).
- *How does throughput differ in Kafka, Pulsar, and RabbitMQ?* ($RQ_3$).
- *How do reliability mechanisms differ in Kafka, Pulsar, and RabbitMQ?* ($RQ_4$).

- *Which security mechanisms do Kafka, Pulsar, and RabbitMQ support?* ($\mathrm{RQ}_5$).
- *How does backward compatibility differ in Kafka, Pulsar, and RabbitMQ?* ($\mathrm{RQ}_6$).

This paper is structured as follows: Section 2 background is described starting from distributed computing to event and messaging brokers' characteristics. Section 3 describes the methodologies used in the research. Section 4 describes the MOMs under research, and Section 5 presents the results. Discussion is in Section 6 and conclusion in Section 7.

# 2 Background

## 2.1 Distributed computing

A distributed system is a collection of interconnected computers over a network and appears to its user as a single coherent system. The benefit of the distributed system comes from the ability to exploit parallel processing on separate computers. These computers are called nodes, and they operate independently from each other. Even though they operate independently, they aim to solve a common goal, and this involves message exchange between computers. Node listens for incoming messages and processes them, which in turn leads to further exchange of messages. The nodes in a distributed system can run on different hardware and software components by different vendors. The nodes communicate with each other using messages through Application Programming Interface (API). The API abstracts the implementation details. Therefore the communicating nodes do not have to know anything about each other except for their API.

The main benefits of a distributed system compared to a monolithic system are increased portability, adaptability, and scalability. Increased availability of cloud computing enables smaller networked devices to run more computationally demanding applications. Therefore scalability has become one of the leading design goals of a distributed system. Typically, problems with a distributed system's scalability are related to servers' limited computational capacity and networking speed. [1]

Scalability in a distributed system has three different dimensions. Size scalability means, that adding more components (i.e. more capacity) to the system will not decrease the performance. Size scalability can face problems related to CPUs' computational capacity, storage capacities of disks, and network capacity between the user and the system. Geographical scalability means that different system components can lie geographically apart, but the user should not notice the delay between components. Geographical scalability is hard to implement with synchronous communication, and hence asynchronous technologies are typically used. Administrative scalability means that the system needs a safe way to handle multiple independent administrative domains that govern the system. Administrative scalability faces problems related to policies of resource usage, management, and security. Typical pitfalls and false assumptions when implementing a distributed system for the first time are related to networking, latency, bandwidth, and administration. [1]

A common way to solve scalability problems is to improve the capacity of the components in the system. This approach is called vertical scaling or *scaling up* and can include increasing memory size, network bandwidth, and upgrading CPUs.

Another way to solve these problems is by deploying more machines into the system. This approach is called horizontal scaling or *scaling out* using three techniques: hiding communication latency, distributing work, and using replication. Hiding communication latency is done using asynchronous communication technologies that enable the system to process other tasks while waiting for the previous to complete. Distribution of work is done by splitting components and spreading them across the system to avoid a single part becoming a bottleneck for the system. Replication spreads the data across the distributed system to increase availability and balance workload. Geographical scaling replicates components and data so that they are closer to the user, decreasing the latency and enabling disaster avoidance. When implementing replication, the system's data consistency is hard to maintain and requires a global synchronization mechanism. [1]

In distributed systems, there are two subtypes of high-performance computing; cluster computing and grid computing. In cluster computing, the aim is to aggregate resources, locating all nodes physically in the same location. In cluster computing, the nodes are homogeneous by their hardware and software and interconnected with a high-speed local area network. Cluster computing aims to solve high-performance tasks where multiple jobs are run concurrently on multiple machines, exploiting parallelism benefits. Typically in cluster computing, a slave-master architecture is used, where a master node allocates jobs to a slave node to perform. One widely applied example of a cluster computer is a Beowolf cluster, in which a single master node schedules tasks to slave nodes governed by the master node. The master node needs to run the software needed to execute sub-tasks on slave nodes and manage the cluster, for example, Apache Hadoop. [1]

Grid computing aims to segregate available resources using networking to run tasks concurrently. In grid computing, the nodes are a collection of computers with heterogeneous software, hardware, location, and networking technologies. The main difference in grid computing from cluster computing is that in grid computing, the nodes are connected using the internet, and the unused resources in a node are available to any job from the grid. Because the nodes contribute to the grid from different geographical locations, the main problem is administrating the shared resources between the sub-jobs. The problem of heterogeneous hardware and software is solved by focusing on the system's architecture; therefore, a layer architecture is used. The architecture of grid computing consists of four layers: fabric layer, connectivity layer, resource layer, and application layer. Typically the collective, connectivity, and resource layers form what is called a grid middleware layer. [1]

Problems of giving access to the resources for customers in grid computing systems lead to the formation of utility computing, in which customers pay per-resource basis. This utility computing concept initiated the basis for cloud computing. Vast amount cloud computing solutions nowadays are based on grid middleware implementations. These solutions include elastic provisioning of computing resources and homogenization of heterogeneous resources of nodes through virtualization. Cloud computing should still be seen as an evolved next step away from the grid computing model rather than its subclass. [1]

## 2.2 Cloud computing

Cloud computing is defined as easily usable and accessible computer system resources. [1] Probably the most significant achievement for cloud computing is the globalization of computing resources. Cloud vendors offer these services as business offerings to the customer through various interfaces like command-line tools, programming interfaces, and web interfaces. These interfaces enable cloud usage without special knowledge, expertise, or control over how the infrastructure supporting those services works. The cloud seems a tempting possibility for companies to increase their software resources compared to managing their infrastructure. Cloud computing is usually billed on a pay-per-use model described with Utility computing, usually defined in Service Level Agreement (SLA). Cloud computing resources are easily scalable since more resources can be acquired automatically when more computing power is needed. This automatic scaling is called *elastic provisioning* or *rapid elasticity*. Using elastic provisioning, customers do not have to engineer their applications for peak times. They simply acquire more computing resources from the cloud provider. Most cloud services run in highly reliable and time-tested data centers, which usually define higher than 99,99 percentage up time in their SLAs. Sharing a data center's computing resources and costs across a large pool of users is called *multitenancy*. [2]

There are four different cloud deployment models: private cloud, community cloud, public cloud, and hybrid cloud. Private cloud means that the cloud infrastructure is available and maintained only for a particular organization based on its needs. Community cloud means that the cloud infrastructure is designed and available only for a particular user group or organization group that shares a specific concern or a mission. Public cloud means that the cloud infrastructure is available for general public and a cloud vendor maintains the infrastructure. A hybrid cloud is a deployment model where the infrastructure contains two or more clouds that are designed to interoperate together. [3] Figure 1 shows the layered architecture of a typical cloud.

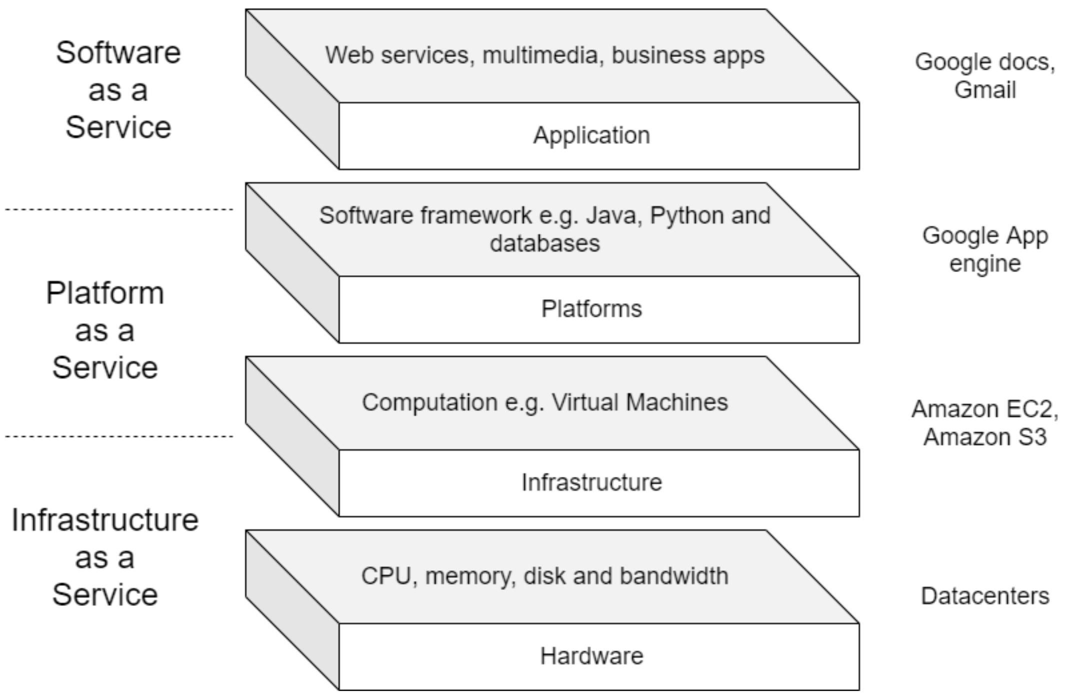
Software as a Service
Web services, multimedia, business apps
Application
Google docs, Gmail
Software framework e.g. Java, Python and databases
Platforms
Google App engine
Platform as a Service
Computation e.g. Virtual Machines
Infrastructure
Amazon EC2, Amazon S3
Infrastructure as a Service
CPU, memory, disk and bandwidth
Hardware
Datacenters

***Figure 1*** *Layer architecture of a typical cloud.*

Typical cloud computing architecture contains four separate layers: hardware, infrastructure, platform, and application layer. The hardware layer is the bottom layer that includes the hardware's necessary resources in data centers like CPUs, memory, routers, power, and cooling system. The infrastructure layer includes mostly the virtualization technologies, including virtual storage and computing resources like virtual machines. The platform layer is equal for a customer as an operating system for a programmer. It gives the customer an interface to upload and manage files to be executed and stored in the cloud. The application layer includes the actual running applications, for example, some of the popular office tools used by office workers like Microsoft Office. Offering these layers to the customers leads to three main types of services: Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS). [1]

SaaS is an application that runs in a vendor's cloud with multiuser architecture that works within a web browser and does not need an installation or management by the user. Running the cloud application using multiuser architecture addresses no up-front investment costs to the client for maintaining infrastructure. Service provider costs are relatively low since only a single application is maintained. [2] The customer does not have access to managing the underlying infrastructure, such as network, server, operating system, or storage. [3]

PaaS is a variation of SaaS. It delivers a platform where the applications are deployed. PaaS environment offers a possibility to manage the whole lifecycle from the development of the application to deliver the application to the user. PaaS can be viewed as an execution environment for programmers that can be used over an API. When using PaaS, web developers are only concerned with web development and do not manage the underlying operating system and infrastructure. The main drawback with PaaS is that the development and execution of the application is limited by the vendor's cloud design and its capabilities. [2] The developers do not have access to modify the cloud infrastructure but have control of the application and limited control of the execution environment. [3]

IaaS is a service offering model that offers the computing infrastructure as a service, usually in a platform virtualization environment. IaaS business model aims to offer a standardized infrastructure to the customer, which the customer then can specialize based on their needs. By outsourcing the infrastructure maintenance work to the cloud provider, customers only maintain their applications and the execution environment. In IaaS, developers can decide the operating system, storage, and networking level, which technologies and configurations to use for running the application. IaaS is an attractive choice because the customer will be offered the service in secure "sandboxes" maintained with the newest security updates and infrastructure equipment. [2]

Among SaaS, PaaS, and IaaS, cloud vendors offer multiple other services using as-a-Service model. These services are called in general XaaS. Cloud vendors are aiming to provide these services at lower cost through new virtualization technologies. [2][3]

Cloud computing still has some problematic topics, including provider lock-in, security, and privacy issues. [1] Vendor lock-in means that changing the cloud provider could be problematic due to differences in handling and saving data. When deciding the cloud provider, also data sensitivity needs to be taken into account. [2]

Developers try to avoid vendor lock-in by adopting a multi-cloud architecture, similar to the hybrid cloud architecture. Multi-cloud architecture has been increasing for a couple of years, but some common problems still need to be solved. For example, a common API needs to be developed so that using multiple clouds from multiple vendors can be simplified. [4]

Typically a cloud application needs to be run in a Virtual Machine (VM) or a container. Using containers leads to a situation where the application runs, even when no request to the container is made and therefore it is running idle. Another cloud computing model is developed to avoid paying for idle running containers, called serverless computing. Instead of running the whole application in the container, the serverless model utilizes Function-as-a-Service functionality (FaaS). In FaaS, only a corresponding function is being run based on the request received. In serverless computing, consumers are mainly billed based on a request count and a memory size rather than a CPU's computational power. [4]

## 2.3 Microservices

Microservice architecture has gained popularity to build a distributed system utilizing cloud infrastructure. Microservice architecture is service-oriented architecture, in which components of the application are split into smaller units to be run in a loose-coupled fashion based on business capabilities. Every service is typically maintained by a single team, and each service communicates with each other using API. These services are run in separate processes and communicate with either by synchronous or asynchronous messaging technologies. Every service can be seen as a separate application with its data, tests, builds, and deployment. Microservice architecture increases development agility by enabling making quick changes only to a single part of the product. The ability to make quick changes makes it possible not to require a redeployment of the whole application when a change is made. Microservices enable polyglot development, meaning each service can be written with programming languages and frameworks best fit for the purpose. Microservices are maintained by small, focused teams with an in-depth knowledge of the service's functionality. Hence new team members can easier join the team and start working. By loose coupling and isolation, fault tolerance increases since faulty service will not affect the whole system. Scaling out of the system can be done simply by adding more replicas of the needed service. System monitoring and traceability increases since metrics from a service can be inspected per container, and for example, memory usage and process utilization can be easily tied to a specific container. The microservice architecture enables the concept of DevOps, which targets to automate everything within the deployment process to enable Continuous Integration and Continuous Deployment (CI/CD). DevOps also aims to merge traditionally separated research and development teams with the operations team that manages the production environment. [5]

Typically microservice architecture is implemented using containers or functions. Containers are the basis of microservices. They are typically based on Linux kernel features called namespaces and control groups. Docker made containers available for the public majority, even though containers are not a new technology. Containers make it possible to isolate components in a distributed system as individual entities that contain their dependencies. Containers have many technological benefits over Virtual Machines (VM), the main ones being a fast startup and stop delays. Containers share the same kernel when running on a single host. VMs also provide hardware-level isolation, hence requiring more computational power. [5] Figure 2 shows how containers and virtual machines locate on a host computer.

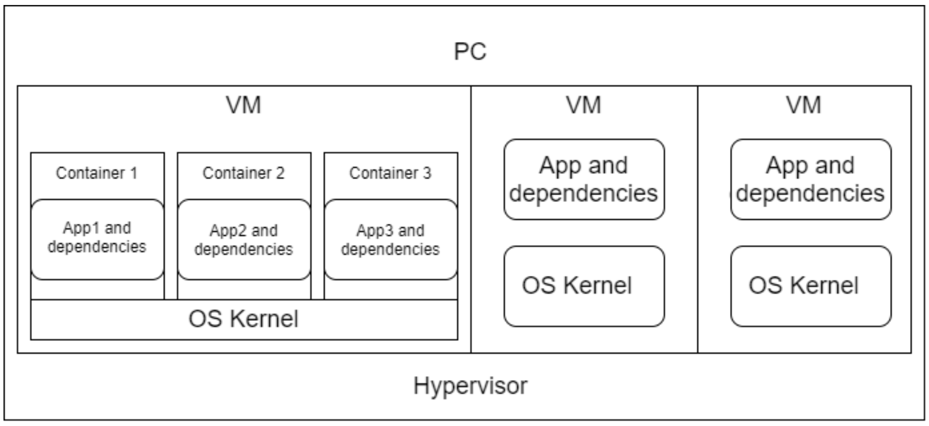

***Figure 2*** *Virtual Machines and Containers on a single host.*

Networking is an important topic in distributed systems because the services communicate over a network using messages. Thereby problems in networking have a significant impact on overall system performance. There exist mainly two different kinds of communication in the distributed system. Internal communication means service-to-service communication in the system, and external communication means communication to external systems outside the system. Internal communication is usually referred to as East-West communication, and external communication is referred to as North-South communication. Hypertext Transfer Protocol (HTTP) is the most used communication protocol in the system in internal communication and external communication. There are also other internal communication technologies to achieve improved performance, such as Websockets and Remote Procedure Call (RPC). Typically communication between internal and external services is done asynchronously so that the running thread will not be blocked when waiting for a response. [5]

A modern distributed system consists of a large number of containers that can not be handled manually. For this purpose, a container orchestrator is used, the most popular being Kubernetes. The container orchestrator takes care of deploying the containers to the nodes and managing their resources. The orchestrator also monitors the resource limits of a node and the overall health of a container. To maintain a container's healthiness, it can restart, reschedule, load balance, and scale the containers based on the configuration. [5] The most important terms and concepts of Kubernetes are shown in Figure 3. They include pods, services, ReplicaSets, and deployments.

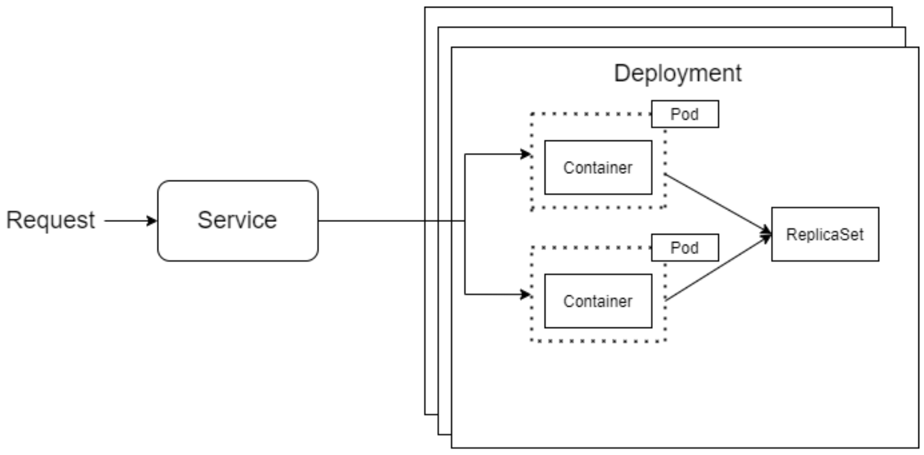

***Figure 3*** *Main concepts of a Kubernetes.*

The pod is an entity of one or more containers, the pod governs the life cycle of those containers, and the containers have a shared network and storage resources in the pod. Service is an entrypoint for a group of pods to create a network service that Kubernetes uses for load balancing between the included pods. ReplicaSet defines how many running instances of the pods are needed to be running at any given time. Deployment is an entity to manage ReplicaSets and pods. Management is based on the desired state defined in the deployment, for example, to rollout and scale ReplicaSets or change pod state. [5]

## 2.4 Cloud Native

There is no unified definition of the term cloud native. The definition depends on the context in which the term is being used. However, the Twelve Factor App methodology presented on the next page is the most commonly referred one. Nevertheless, there is a relative agreement among developers over why cloud native development is important. Cloud native breaks old monolithic concepts and replaces them with a more agile way to deliver innovation with confidence. Cloud native promises increasing speed, safety, and scalability over traditional development models. Cloud native helps companies delivering products faster to customers by leveraging CI/CD deployment model. Cloud native also increases the feedback loop's speed from the customer, hence enabling fast experimental changes of the product. By following cloud native best practices, the application can be expected to have nearly zero downtime. By embracing a disposable environment, flexible architecture, isolated components, and leveraging scaling, the system can stay responsible even when facing failures. Cloud native also drives cultural change inside an organization to cloud native driven lean thinking to accelerate innovation and experimentation. [6] Cloud Native Computing Foundation (CNCF) is a Linux Foundation project to build a sustainable ecosystem for cloud native technologies. CNCF hosts a list of projects on their site that are considered best suited for Cloud Native development.

It is challenging to design bounded isolated components that are not too big or too small by functional or technical perpective. The goal is to have each component designed for a single responsibility to maintain logical structure and simplicity in the system. A shared understanding has to be created to bound the components to maintain consistency across the teams. For that purpose, the concept of Domain Driven Design (DDD) was invented. DDD embraces bound components based on the business needs of the software. Different areas of the system are divided into bounded contexts that have internally consistent naming. The messaging between these contexts and their components can be modeled based on domain events. Components publish the events to downstream components and subscribe for events from upstream components. [6]

When cloud computing started gaining popularity, scalability problems were usually solved by scaling up, meaning adding more resources to computing nodes. In the cloud, a more beneficial way to scale is by scaling out, which means adding more machines to the system to share the computing load. Scaling out requires the application to be stateless. Hence new methodologies for application development had to be invented. As an outcome, engineers at Heroku developed a methodology called the twelve Factor App. These twelve factors describe the best practices for stateless application development. The Twelve Factor App methodology can be considered as the foundation of cloud native application development. [5]

The twelve factors are:

**Codebase** In order to have a successful CI/CD pipeline, the application should have a single codebase. Typically, this is implemented using a single repository with the help of a Version Control System (VCS) such as Git. The codebase can have multiple deployment stages, for example, development, testing, and production stages. [5]

**Dependencies** The dependencies of the application should be isolated and declared separately from the application code. The dependency declarations should be shipped with the code. Hence it is highly advised to use dependency management tools such as npm or Maven. Also, proper usage of containerization tools will reduce the management complexity of the dependencies. Dependencies can be declared with tools such as Dockerfile. [5]

**Configuration** Developers should separate configuration code from the source code to tackle differences in the configuration in different deployment environments. By separating configuration from code, the configuration can be changed dynamically per environment. It is a good practice to use configuration management tools, such as Helm. [5]

**Backing Services** A backing service can be anything that the application uses over a network, for example, a database or a message broker. These backing services should be treated as any other component in the system, and they should be loosely coupled so that they can be replaced easily. [5] From the application perspective, there should not be a difference if the service is local or external. [7]

**Build, Release, Run** Build, release and run stages should be separated from each other in the development pipeline to establish continuous integration, delivery, and deployment models. Properly attaching the required configurations and dependencies to the application is critical in every stage. [5]

**Processes** The application should be executed in one or more stateless processes. Meaning all the data should be retained outside the process to another stateful backing service, like into a database. [5]

**Port Binding** Data in every service should be isolated from other services. No service should be able to access data directly maintained by another service. The service managing its data can be accessed only through an exposed port, thereby securing its internal data from other services. [5] Every service should be exported using port binding, which means listening to incoming requests from the exported port and responding accordingly. [7]

**Concurrency** The processes are first-class citizens. Their implementation is based on the Unix process model, and they leverage different types of processes for different tasks. [7] Using the Unix process model, the processes are independent and can be scaled horizontally to achieve better resource usage. [5]

**Disposability** The robustness and scalability of the application are designed by leveraging fast process startup and stopping time. [7] The containers already implement such functionality and therefore are recommended as virtualization tools in cloud native development. [5] The application should be designed to be robust against sudden death, which is called a crash-only design. [7]

**Dev/Prod Parity** Development, staging, and production environments should be as similar as possible to avoid configuration errors between the environments. Having similar environments is a critical part of establishing a functional CI/CD pipeline successfully. [7] Containers will help to package the dependencies within the application to avoid differences in the environment. [5]

**Logs** The distributed system needs a good logging strategy because of its distributed nature. Without a proper logging strategy, finding bugs and their reason can be complicated from the system afterward. Therefore it is a good practice to treat logs as event streams and use a proper logging framework. [5]

**Admin Processes** Administrative and management tasks such as database migrations should be done using on-off scripts. These scripts should be shipped within the application to avoid synchronization issues. [7]

## 2.5 Event-Driven Architecture

In a complex distributed system, a single action can trigger a long chain of actions. At some point, the action flow is no longer clearly connected to the original action, leading to the creation of an event-driven architecture. The event-driven architecture is a popular pattern used in an asynchronous distributed system. It can be applied to large complex systems as well as small systems to provide high scalability. The architecture is based on decoupled event processing components that consume and produce events. The service, which consumes and processes events from the event stream, is called a consumer, and the service which produces an event to an event stream is called a producer. There exist two different main topologies to implementing Event-Drive Architecture, mediator topology and broker topology. The mediator topology is useful when there is a need to trigger multiple actions parallel on a single event and broker topology when there is a need to chain actions. The mediator topology is typically more complex to implement than broker topology since the mediator component is more tightly coupled with event processors. [8] Figure 4 shows an example of the broker topology.

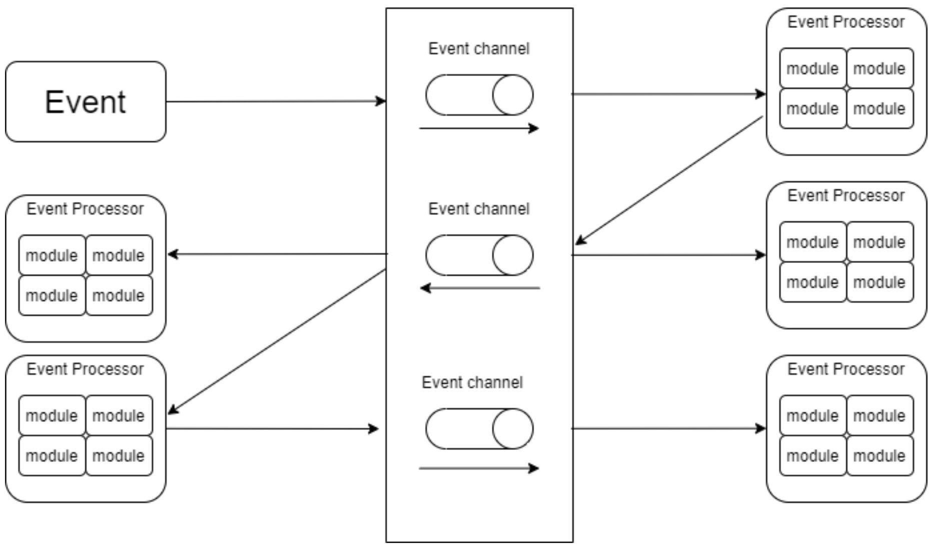

***Figure 4*** *Example of a broker topology.*

In the broker topology, the messages are distributed across the components using a message broker. The broker topology simplifies event processing flow and does not require an event orchestrator. The broker topology consists of two main components, a broker component and an event processor component. The broker component includes all the event channels of event flow, and those channels can be implementations of message queues or message topics, or both. In broker topology, the event processors receive the event from a broker and possibly publish a new event to the broker after processing the event. Since microservices exchange data with each other using messages, a standard message format must be chosen that is being respected from the very start of implementation. Among all the other event-driven architecture patterns, the broker topology pattern still faces the typical issues common when designing distributed systems using asynchronous messaging. One of the main issues is broker failure. Therefore actions for reconnecting the broker must be implemented. Since event processors are decoupled and distributed, and each microservice typically has its database, atomic transactions are hard to implement. [8]

## 2.6 Message

The most used message format for internal and external communication on the web is JavaScript Object Notation (JSON). [5] It is a flexible and readable data format, but serialization and deserialization require a large memory footprint with large amounts of data. When performance is a critical requirement in the system, alternative formats that convert messages into binary can be used, such as Avro. [5]

## 2.7 Message Queue

A message queue is a component in messaging systems that implements a first-in-first-out (FIFO) functionality to order and store messages. The message queue can store received messages in the memory for later use if there are no active consumers at receiving time. Into the message queue can be sent any data. It does not validate the format by default. Since messages can be saved in the queue for a certain amount of time, the messages are sent asynchronously, increasing decoupling and performance. Message brokers typically implement message queue functionality. [9]

## 2.8 Message Broker

A message broker is a communication technology for microservice architecture in cloud native environment. It keeps track of publishers and subscribers for defined topics, which over the services then exchange the data. The message broker is an important abstraction layer in inter-service communication to isolate services and simplify communication. It enables the microservices to be written in a polyglot manner because the message broker can transfer the messages using common technologies, such as HTTP. Using message brokers, the communication can be abstracted into this communication layer, and developers can concentrate on the service business logic. Message brokers can validate, store, route, and deliver internal and external messaging based on the setup and configuration. [9] Message brokers are designed for saving the messages for a short while. The messages are deleted when acknowledged or shortly after. [10]

## 2.9 Event

An event can be seen as a notification of an action that just happened or a mechanism to transfer a state inside the system. An event flows through an event channel and gets sent to interested consumers, and makes them do a state change. [11] An event in event-driven architecture is the data for which services react and that they process. An event stream forms a channel for communication between services and data storage in which the events are stored afterward. The events in the event stream act as a single source of truth for the system. Hence the events must accurately describe what happened. Not all the data must be published into the event stream, but if even a single service is interested in the event, it must be sent into the event stream. Events are typically formed in a structure of key-value pairs. The key includes a unique id so that the event can be identified later on. The event structure is typically defined in an event definition, and a single event stream should include only one type of event and ensure that different kinds of events are not mixed up. When creating the event definitions, most narrow data types should be used to assure proper data boundaries. [10] It is best to save exactly the whole received event stream to the event storage and not aggregate them. [11]

## 2.10 Event broker

An event broker includes all the functionalities of a message broker and can be used as a replacement for the system's message broker. The event broker is designed to retain an ordered log of events, so they keep track of all the events that have been sent into the stream instead of deleting them like message brokers. Since message brokers are designed usually based on message queue logic, the messages are delivered to consumers per-queue basis. All the consumers that share that particular queue for consumption will only receive a subset of all the messages sent into the queue. By sharing the queue between consumers, it is impossible to communicate state changes to all the consumers properly since all the consumers can not obtain all of the data. The main benefit with event brokers compared to message brokers comes from retaining and replaying the messages. The event brokers typically implement this by saving the event stream into append-only immutable storage. The consumers can read the whole event stream from start to end in the correct order. Since event brokers retain the immutable stream in its storage, the stream works as a single source of truth for the application. Typically all the event brokers and message brokers are referred to with a hypernym known as message-oriented-middleware. [9] Figure 5 illustrates the difference between a queue and a topic.

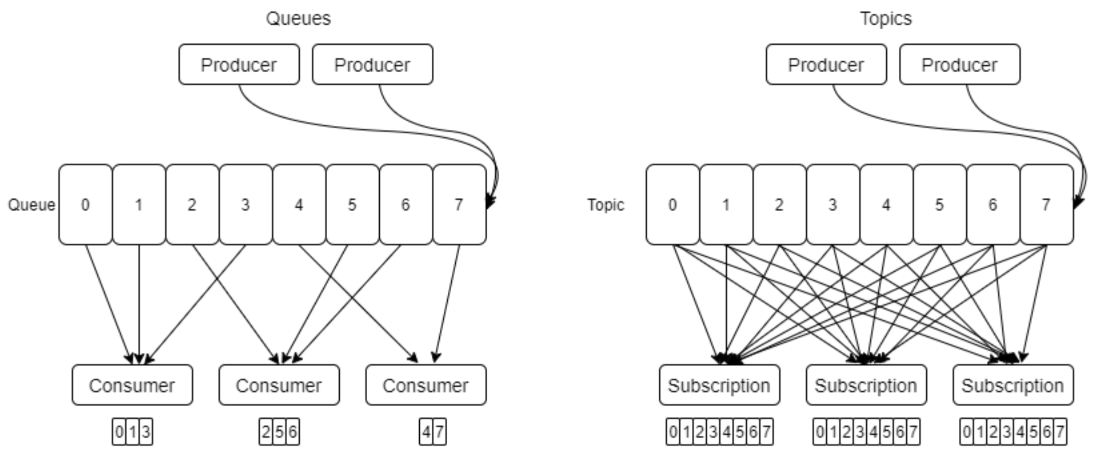

***Figure 5*** *Queue and topic based pub/sub implementation.*

Event brokers can be distributed into multiple instances to increase performance, scalability, durability, high availability, and fault tolerance. In distributed event broker, the data needs to be replicated between nodes to keep the system functioning in the broker fault case. The event broker is in charge of storing the events for further processing. Hence it has many requirements considering its functionality. The broker's storage must be able to partition the event stream into substreams for parallel processing for multiple instances of a consumer. The ordering of events must not be changed in the event stream, and they must be delivered to the consumer in the order they were saved into the partition. The events in the event stream must be immutable. No service should be able to modify the content of the event stream once published. Indexing in the event stream is vital for consumers since they keep track of the offset to know where it is reading the stream. The difference between the offset in which the consumer is reading currently and the newest item in the stream is consumer lag. The system can be scaled to retain the consumer lag in desired limits.

When choosing an event broker for the system, many things must also be taken into consideration. Supportive tooling is vital, such as browsing capabilities of the event stream, monitoring, topic management, and access control. Many hosted services by the cloud vendors for event brokers have a rich feature set for a meaningful cost. The client library support varies between brokers, and hence it must be taken into account that the programming languages used are compatible with the client libraries. [9]

Consumers can be grouped into consumer groups to implement horizontal scaling. Every consumer in a consumer group is assigned a partition from the stream which the consumer group reads. The number of consuming consumers is limited to the number of partitions in the event stream. Even though the event broker stores all the event stream data in its storage, the consumers can also store the received events in their own storage, such as a database. [10]

## 2.11 Persistence and Durability

Durability in messaging systems means that the messages sent into the queue or topic will be retained by the MOM if there are no active subscribers on the queue or topic at the moment. Durability is strictly connected with a quality of service in MOMS. Without durability, messages will be lost if there are no active subscribers at the time of message arrival. Persistence in messaging systems refers to saving the received message into persistent storage, such as a log or a database. MOMs can be durable or persistent, depending on the used technology and configuration. Both are essential functionalities in case of broker failure that requires a broker restart. In case if the broker is not durable and messages not persistent, data will be lost. [12]

## 2.12 Event streaming

The difference between messaging and streaming is that event stream is a constant replayable immutable dataflow. Instead of containing all the information as a payload for further processing, like with messages, they contain only the needed information to make a state change in the system. Messages contain the intention or action in them along with the payload, making them not lightweight. Stream data is unbounded, meaning it has no real start or endpoint. The system state can hence be recreated by replaying the stream again until the wanted point.

Event streaming is an efficient way to implement asynchronous inter-component communication in the microservices architecture. To achieve responsive, resilient, and loosely coupled systems, stream processing has become de facto standard for cloud native systems. Event streaming simplifies service discovery because data change events are delivered to all consumers via loosely coupled topic-based subscriptions defined in the event broker. Event streams are implemented using event brokers that stream the events to all interested participants in the system. [6] Event streaming does not guarantee the availability of all the data at the same time, hence buffering would be needed. It is a good practice to implement services as stateless by using a database per service model. This approach makes the service stateless and the database stateful, but this can become problematic because most services are not fully stateless. A service can become easily stateful, for example, when it needs to do any filtering, joining, or aggregation on its input data. When a service is stateful, a stateful streaming approach will help to manage the state. A stateful streaming approach can be applied by leveraging the service's database to solve stateless streaming problems. [11] Figure 6 illustrates stateful streaming.

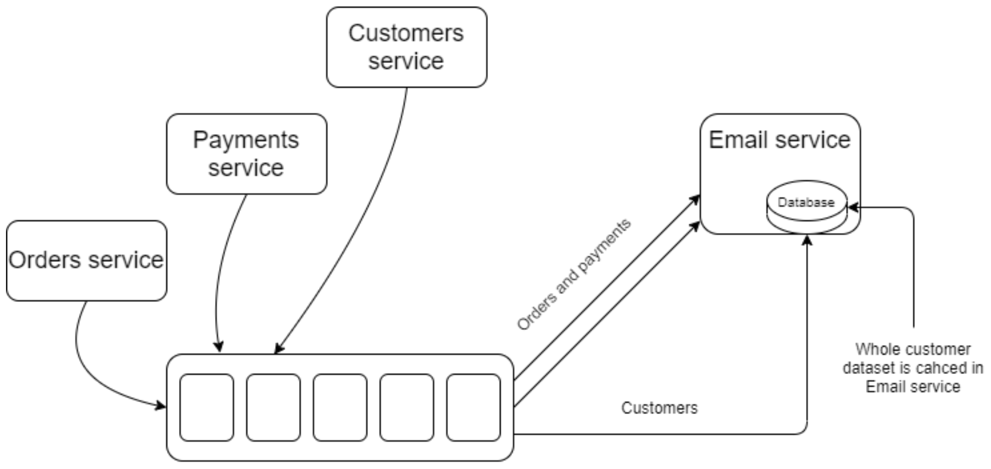

***Figure 6*** *Example of a stateful streaming using Kafka Streams.*

In stateful streaming, event streams can be pushed to all the interested services, converted and saved into database tables locally, and kept in sync by Kafka. A great example of a stateful streaming solution is Kafka Streams. [11]

## 2.13 Message Bus

Message Bus is an architecture, which in the core is a centralized software platform to translate internal system communication into a common language for all the connected services to understand. Meaning, it translates different kinds of data formats, such as XML, for example, into JSON so that all components can communicate together. When deploying a message bus, all the services connected to it must share common data types, communication commands, and communication protocols. The underlying software also operates internal orchestration logic such as connectivity, routing, and request processing. Message bus was a common technology used in the 90s with service-oriented architecture. Since microservices architecture has gained popularity, message bus does not fit well anymore with microservices, and it is hard to scale and maintain. For these reasons, the message bus has dropped out of favor in modern distributed cloud systems. A message broker is in microservices architecture corresponding component for message bus in the service-oriented architecture. [9]

## 2.14 Patterns

### 2.14.1 Point-to-Point

A point-to-point pattern uses a message queue in the middle of two services that deliver the data from sender to receiver. The queue can be set to follow the first-in-first-out (FIFO) principle to deliver the messages in the receiving order. Only one receiver will receive the message from the queue, but the messages can typically be distributed in a round-robin fashion in case of multiple receivers to increase throughput. No response from the receiving service is expected by default in a point-to-point pattern. [12] Figure 7 illustrates the point-to-point messaging pattern.

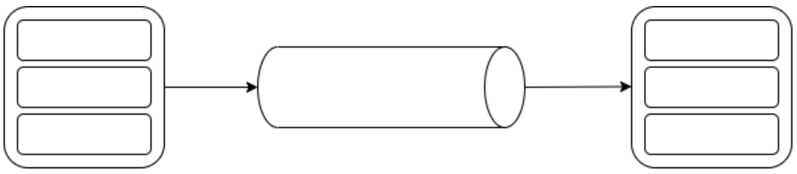

***Figure 7*** *Example of a point-to-point messaging pattern.*

## 2.14.2 Request/Response

Request/Response is a messaging pattern that differs from point-to-point so that the receiver sends a response to the request. The two communicating services change data by sender sending a message through a queue and waiting until the receiving service sends a response. The queues in this pattern are not required. Request/response pattern can also be implemented without queues. When services typically engage in multiple communications, some messages can be lost or delayed. In this situation, the message queue can be used to assure message delivery.[5]

In MOMs, implementing request/response pattern is typically complicated, and it comes with a downside of decreased throughput and increased complexity. Figure 8 illustrates the request/reply pattern.

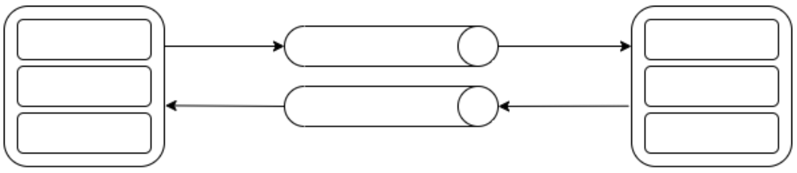

***Figure 8*** *Example of a request/reply messaging pattern.*

## 2.14.3 Publish/Subscribe

Publisher/Subscribe (pub/sub) pattern is one of the most commonly used messaging patterns in cloud applications. Pub/sub pattern is based on topics that define to which subject the sent message is related. Topics reduce the number of connections in the system. Subscribing services need only to know which topic to consume, and publishing services need only to know which topic to publish. Topics enable loose coupling between publishing and consuming services because they do not have to know anything about each other. Pub/sub is the basis for event-driven architecture and design. In pub/sub pattern, a MOM must keep track of subscribing and publishing services to guarantee message delivery. Pub/sub pattern ensures that the published message will be delivered at some point of time, making the data eventually consistent. [5] Figure 9 illustrates the pub/sub messaging pattern.

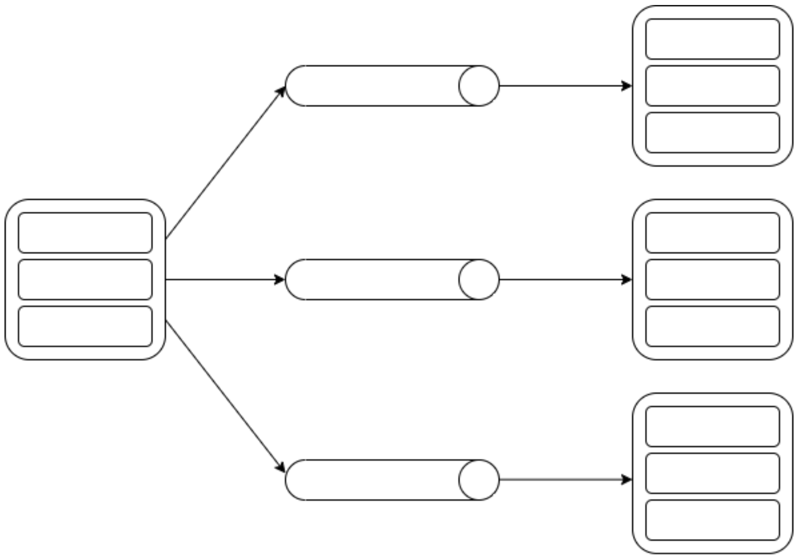

***Figure 9*** *Example of a pub/sub messaging pattern.*

## 2.15 Message Delivery Guarantees

In MOMs, pub/sub topics are typically by default nondurable, hence providing an at-most-once delivery guarantee. That means the message is delivered zero or one times, meaning it can be lost, but it delivers the best performance. Implementations that use durable topics with some reliability mechanism, such as acknowledgments, can give an 'at-least-once' delivery guarantee. At least once delivery means that message is being delivered one or more times. That means, possibly multiple attempts to deliver the message are made, hence possibly ending up with duplicate messages in the receiver. [12][13] Figure 10 illustrates different message delivery guarantees.

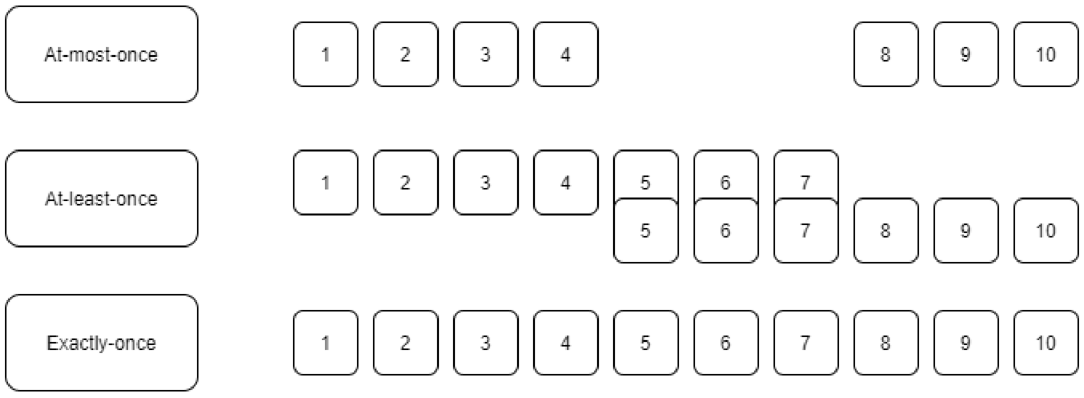

***Figure 10*** *An illustration of different message delivery guarantees.*

Typically, MOMs are incapable by default to deliver an 'exactly-once' delivery. This delivery guarantee is the most expensive because it requires additional acknowledgment logic and additional functionality to filter duplicate messages. Typically most messaging systems deliver all or some of these message delivery guarantees, and the guarantee can be defined when setting up the system depending on the MOM. [12][13]

## 2.16 Network Management System

A network management system (NMS) is designed for Communication Service Providers (CSP) to manage their network operations. NMS operates a large number of network elements (NE) that can not be managed manually on the needed scale. NMS provides intelligent automation tools for, e.g., NE monitoring and configuring so that CSPs can meet the defined service level agreements for their network.

On a high level, the NMS operations can be described with the acronym 'FCAPS', which stands for Fault, Configuration, Accounting (for non-billing use when not dealing with collecting billing data), Administration, Performance, and Security. This is the management task categorization used in the ISO (International Organization for Standardization) model for network management.

Fault management deals with recognizing, collecting, isolating, and correcting faults that occur in the network. Configuration management deals with gathering, storing, and modifying the configurations of NEs. Administration activities cover, e.g., managing system users and their permissions and downloading and activating new SW releases to the NEs. Performance management deals with collecting, processing, and storing measurement data from the NEs and reporting on the collected data (e.g., revealing any adverse trends in some key performance indicators). Security management deals with things like controlling access to NEs, encrypting traffic between NMS and the NEs, and collecting audit logs from the NEs.

NMS plays, if possible, an even more important role in CSPs' future with the advent of 5G technology, which complicates network management even further due to its new functionalities, such as network slicing. NMS should manage thousands of NEs from different telecom vendors with 'zero touch' operations (i.e. with minimum human intervention) to decrease costs for the CSPs and eliminate human errors. Data volumes coming from the managed networks are huge, and the NMS systems must scale to duly process and store all incoming data to ensure no data is lost.

# 3 Methodologies

A systematic literature review was conducted based on guidelines defined in [15] and [16] to gather knowledge about the research topic. Also, a "snowballing" process was conducted based on guidelines in [17].

In this section, research questions are described (Section 3.1), and then in the next section, the implemented search strategy is reported (Section 3.2). After them, a quality assessment of included papers (Section 3.3) is described, and then data extraction methods such as used include and exclude criteria and analysis (Section 3.4).

## 3.1 Research Questions and Goal of the Research

The goal of this research is to compare messaging solutions based on a chosen criteria and to gain knowledge of MOMs for cloud native environment. The chosen criteria for the research is based on NMS's important characteristics in a cloud native environment. These characteristics are: implemented consumer-producer patterns, throughput, and supported security, reliability, scalability mechanisms, and backward compatibility. The chosen MOMs for the research are Apache Kafka, Apache Pulsar, and RabbitMQ.

Based on aforementioned requirements for the research, the research questions are formed:

- *Which consumer-producer patterns do Kafka, Pulsar, and RabbitMQ implement?* ($\mathrm{RQ}_1$).
- *How do scalability mechanisms differ in Kafka, Pulsar, and RabbitMQ?* ($\mathrm{RQ}_2$).
- *How does throughput differ in Kafka, Pulsar, and RabbitMQ?* ($\mathrm{RQ}_3$).
- *How do reliability mechanisms differ in Kafka, Pulsar, and RabbitMQ?* ($\mathrm{RQ}_4$).
- *Which security mechanisms do Kafka, Pulsar, and RabbitMQ support?* ($\mathrm{RQ}_5$).
- *How does backward compatibility differ in Kafka, Pulsar, and RabbitMQ?* ($\mathrm{RQ}_6$).

It is common knowledge in the industry that the most popular MOM in cloud native development is Kafka. Kafka's popularity in cloud native development is well earned. Kafka is by many characteristics outperforming traditional messaging brokers and was one of the first event brokers in the market.

These MOMs were chosen for the research based on knowledge gathered from the industry of the best Kafka alternatives. Other alternatives also were considered, such as RocketMQ and NATS Streaming, but they were excluded because there was very little research. Also, including them would have enlarged the scope of this research too broad.

## 3.2 Search strategy

This section describes the search strategy, used search terms, used bibliographic sources, listing selected inclusion and exclusion criteria, and a selection process of inclusion for the gathered sources. Figure 11 illustrates the process for gathering the sources for this research.

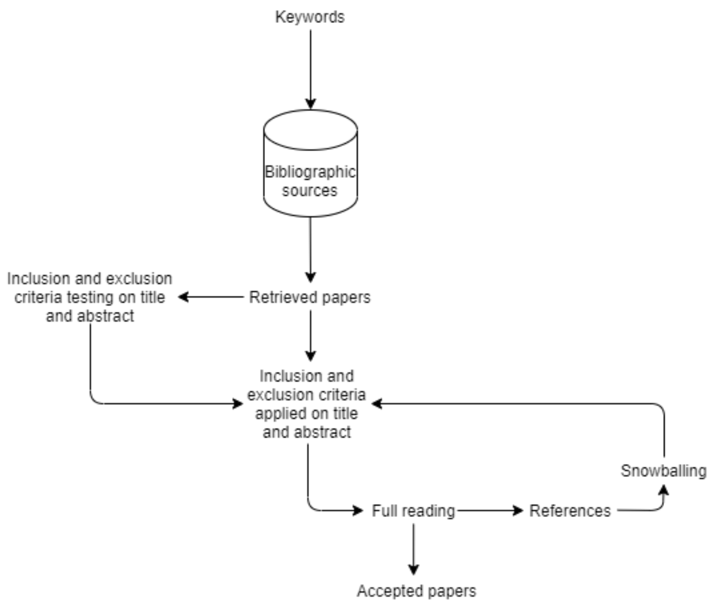
Keywords
Bibliographic sources
Retrieved papers
Inclusion and exclusion criteria testing on title and abstract
Inclusion and exclusion criteria applied on title and abstract
Snowballing
Full reading
References
Accepted papers

***Figure 11*** *A process used for choosing the data included in the research.*

**Search terms.** The selection of used search terms was based on the chosen technologies, including their names and also any wording related to the characteristics for which the comparison was to be done against. The search string contained the following search terms:

**( (”kafka”) OR (”rabbitmq”) OR (”apache pulsar”) ) AND ( (benchmark*) OR (secur*) OR (”reliability”) OR (”scalability”) OR (”performance”) OR (”throughput”) OR (pattern*) )**

The asterisk character (*) was used to match any wording variations, for example, plurals and verb conjugations. The terms were matched against the title, abstract, and keywords of the paper to increase the likelihood of accurate results.

**Bibliographic sources.** The most relevant bibliographic sources were selected based on suggestions in [16]. The sources were selected because they are commonly known as the most relevant sources for the software engineering domain. The list includes Scopus, IEEEXplore Digital Library, Springer, and Google Scholar.

**Inclusion and exclusion criteria.** The inclusion and exclusion criteria were first applied to the title and abstract (T/A). Later it was applied to full text (Full) for included papers after reading the abstract. Table 1 shows how they were implemented based on the criteria.

***Table 1*** *Inclusion and exclusion criteria*

| Criteria | Assessment criteria | Step |
|---|---|---|
| Inclusion | Comparison of the characteristics of chosen technologies | Full |
| | Architectural comparison of chosen technologies | Full |
| | Research on the future of any of chosen technologies | Full |
| Exclusion | Not in final stage | T/A |
| | Not written in English | T/A |
| | Published 2016 or earlier | T/A |
| | Research not applicable for cloud native | Full |
| | Paper done for marketing purposes | T/A |
| | Not peer reviewed | T/A |
| | Duplicate papers | Full |

**Search and selection process**. The search was conducted from September 2020 to January 2021, including all the papers available. The search string returned 241 unique papers.

*Testing the applicability of inclusion and exclusion criteria.* Inclusion and exclusion criteria were tested on ten papers randomly selected from the retrieved papers.

*Applying inclusion and exclusion criteria to title and abstract.* The inclusion and exclusion criteria were applied to all 241 unique papers. From all 241 papers, 19 papers were included after reading their title and abstract.

*Full reading.* The included 19 papers were read fully. From them, 14 papers were excluded based on the excluding criteria, typically because they were not applicable for cloud native environment or had no comparison between two or more technologies. After a full reading, five papers were still included in the research.

*Snowballing.* The snowballing process was performed on all the referenced papers in the included ones after full reading, ending up with one additional paper. Three papers were identified to be potentially included, but only one was selected after a full read.

After the whole search process, six papers were included in the research, as reported in Table 2.

***Table 2** Results of search after applying inclusion and exclusion criteria*

| Step | # Papers |
|---|---|
| Retrieval from bibliographic sources (unique papers) | 241 |
| Reading by title and abstract | 222 rejected |
| Full reading | 14 rejected |
| Backward and forward snowballing | 1 included |
| Papers identified | 6 |
| **Primary studies** | **6** |

## 3.3 Data extraction

Data were extracted from 6 primary studies, and a pie chart was created from the distribution of technologies in the included papers. Since most of the papers included more than one technology typically, the amounts do not add up to 6 papers. Figure 12 illustrates how many times any of the chosen MOMs are under research in the included papers.

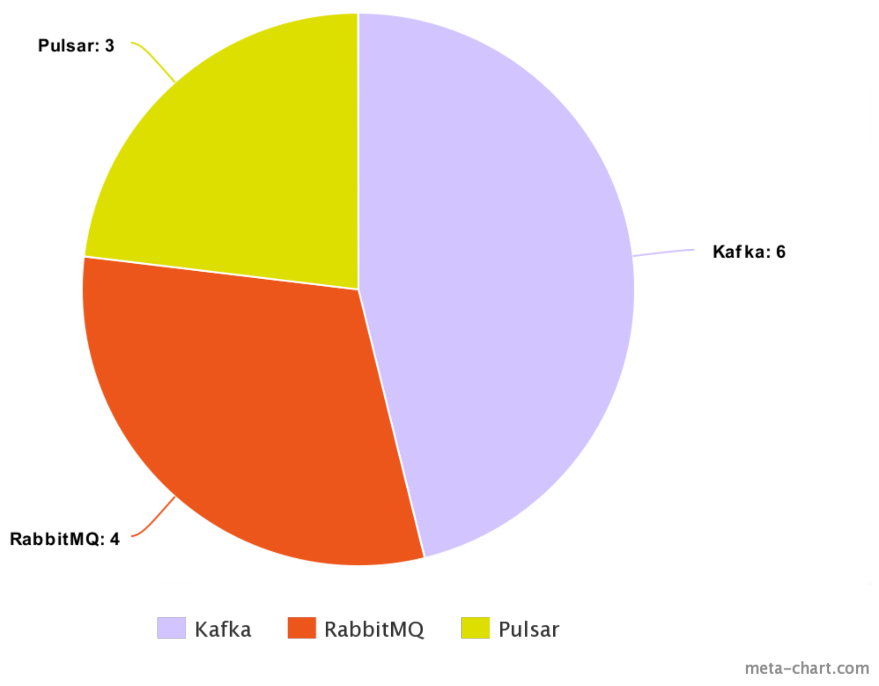

***Figure 12** Distribution of the chosen technologies in the included papers.*

In general, the search process aimed to have every included paper some benchmarking or comparison results between two or more selected technologies. This requirement is the reason why there are only six papers left. This thesis aims to compare the selected technologies one against the another and not compare the features of a single technology.

The reason why so few of the papers did include Pulsar is, as we will see later, it is quite new technology. RabbitMQ and Kafka are considerably older, hence having much more papers done research using them.

# 4 Technologies

Comparing MOMs to each other is a non-trivial task. The best choice is whatever fits your use case. Typically, every MOM aims for a certain type of messaging that makes them good at one thing and worse at another. There is no general comparison between the MOMs because there are many different variables that affect the results when making comparison. For example, these variables are the server's underlying hardware and the configuration settings of the MOM. In the next section, typical differences between MOMs are described in which this research is concerned. Some of the features are closely related to each other. For example, data replication is related to reliability and scalability.

## 4.1 Comparing MOMs by defined characteristics

**Patterns**. All the MOMs simplify connections inside the system by implementing different messaging topologies such as point-to-point and pub/sub. There are two types of communication between a MOM and a client, push-based and pull-based communication. Implementations in MOMs that use push method to the consumer, could without any backup logic lead into a consumer becoming overwhelmed by the amount of data it receives, in essence into a type of Denial of Service Attack situation. This research will compare available **messaging patterns** in the MOMs, and **push/pull implementations** between MOM and client.

**Scalability**. MOMs implement load balancing and replication by sending metadata about connected consumers and producers. MOM aims to implement a proper load balancing between the producers and consumers and scale out when needed to avoid bottlenecks. The ability to configure buffering for the messages and streaming is essential functionality for a modern MOM. The broker storage is also closely in connection with the reliability of the MOM. For example, replication can be done on the broker level or data level. By leveraging replication on the broker level by replicating broker instances, broker faults do not endanger the system's whole functionality. Data replication replicates the data across the broker instances to increase fault tolerance so that data will not be lost even if one broker instance is lost. Data and broker replication is also an essential aspect of scalability. In case of an increased message load, increasing the number of broker instances will prevent bottlenecks in the system. This research will take a close look at **data replication mechanism** and **broker replication mechanism** of each MOM.

**Throughput**. Throughput in the context of MOMs is generally defined as the amount of messages flowing through the broker. The factors that affect throughput are typically related to reliability mechanisms and scalability mechanisms. There are also differences between brokers in how they are able to handle different message sizes and how that affects the latency of the system. Also, the filtering of the messages is important, and that implementation can be located in the broker or at the client depending on the broker, which will also affect performance overall. Sending the messages in compressed or decompressed form also has a significant effect on the performance. This research will compare effects of scaling mechanisms, reliability mechanisms, and message size to **latency** and **throughput**. Also available **compression types** are compared for each technology.

**Reliability**. Reliability in MOMs is closely related to message persistence and durability. Traditional MOMs don't always implement durability, meaning that messages can be lost, and the messages can become unordered in case of no active receivers at the receiving moment. Traditional MOMs tend to save received messages into their local memory, and modern MOMs usually have the functionality to save them into persistent storage. [9] Also, message delivery guarantees and message ordering are essential to increase reliability. Delivery guarantees differ from broker to broker, but typically most of them support at-least-once and at-most-once deliveries. Strict message ordering typically has a decreasing performance effect on MOMs. This research will compare **message persistence**, **message durability**, **message delivery guarantees** and **message ordering** of each MOM.

**Security**. The main security features for MOMs are encryption and authentication. Encryption ensures that messages between clients and MOM are secured with an encryption mechanism. Typical encryption mechanisms for MOMs are Secure Sockets Layer (SSL) and Transport Layer Security (TLS), usually mentioned as SSL/TLS. Authentication ensures that only wanted clients can create a connection to the MOM. The connection can be established with a secure and reliable authentication mechanism such as Simple Authentication and Security Layer (SASL). Both security mechanisms can affect the system performance, especially throughput. This research will compare available **encryption** and **authentication** mechanisms for each MOM.

**Backward compatibility**. Typically all the MOMs promise that their systems are backward compatible, depending on the versioning. Anyway, even the smallest changes to the API can require refactoring and bugs. Rolling upgrade is essential for systems promising high availability, meaning an ability to upgrade the broker version without generating downtime in the system. This research will compare implementations of **rolling update** and whether **backward compatibility** is guaranteed between versions in each MOM.

## 4.2 Kafka

Kafka is an event streaming platform written in Scala. Kafka was open sourced in 2011 and is being developed and maintained by Apache Software Foundation. Kafka was initiated at LinkedIn to replace traditional message brokers, which were complicated to scale. Kafka is developed to aim for low latency and high throughput with large volumes of data, meaning it favors performance over reliability features.

Traditional message brokers tend to slow down when adding multiple consumers into a topic and queue. Kafka solves this by a different architectural approach that makes it easier to scale the broker horizontally. Kafka cluster's architecture is two-tiered, which includes a Kafka broker and a ZooKeeper server. [18]

### 4.2.1 Patterns

Kafka supports two messaging patterns by leveraging topics, point-to-point, and publish/subscribe. To utilize parallel processing to increase performance, consumers can be separated into consumer groups. The consumer groups can consume a single topic at the same time. Consumer groups manage their message offsets, telling where it is currently reading the data in the partition. Kafka replicates the topic partitions and tries to assign each partition to one consumer in the consumer group. Since only one consumer in the consumer group can read the messages of a single partition, increasing performance is done by adding more partitions and consumers. By leveraging independent offsets, in case of a single consumer, a topic acts as a queue and with multiple consumers as a pub/sub topic. [12] Figure 13 illustrates the architecture of the Kafka broker.

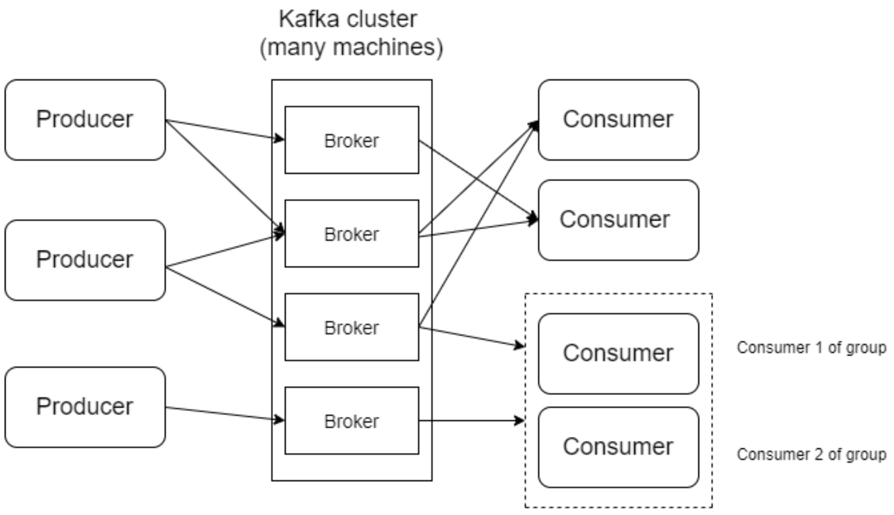

***Figure 13*** *An architecture of a Kafka broker.*

Each consumer group can have multiple instances of a particular type of consumer in the group, but only the lastly joined one into the group will continue consuming messages. The ability to transform a topic into a queue simplifies connections since there is no need to keep track of e.g., which of the consumers should get the next message in a round-robin fashion like in traditional message brokers. In addition, there is no need to keep track of which consumer the earlier messages were delivered, in case of redelivery.

In Kafka, consumers pull data from brokers. When consumer falls behind using pull-based connection, consumer will catch up later, since it manages its own offset in that partition. These all things are an example of transferring responsibility from broker to client. It increases broker performance but needs to be taken into account in client code. As typically with all the pull-based systems, Kafka leverages aggressive batching of data. [12] Kafka offers streaming by Kafka Streams API. [18]

### 4.2.2 Scalability mechanisms

Kafka brokers are connected to ZooKeeper servers that provide high availability by replicating information across the system. ZooKeeper manages the distribution of partitions of a topic between brokers in the cluster. ZooKeeper assigns master and slave roles for broker instances in the cluster. The master broker receives the messages sent into a topic and is responsible for replicating the messages among the slave brokers. These distributed partitions are called 'replicas', and a broker containing all of the messages and partitions also held by the leader is called an *in-sync replica*. There is ongoing work to replace ZooKeeper with a self-managed metadata quorum. [21] Figure 14 illustrates how Kafka brokers are replicated in the cluster.

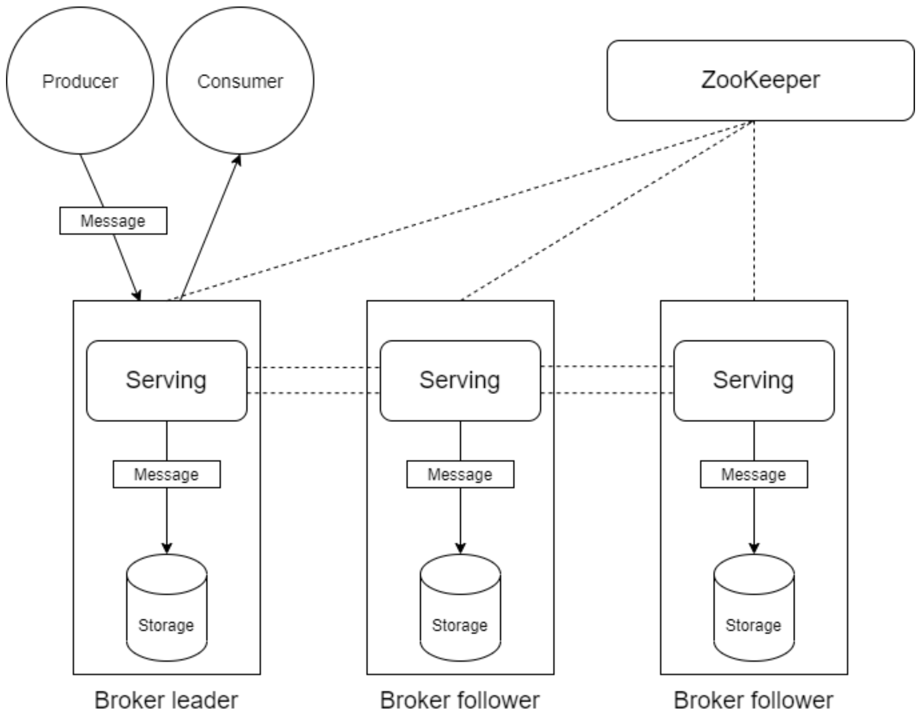

***Figure 14*** *Replication in Kafka brokers.*

By leveraging in-sync replicas, in the case of a master broker going offline, the in-sync replica will take over the master role. Slave brokers can send acknowledgment receipts, which defines how many in-sync replicas must send a receipt about a successful replication before the application thread continues. For example, replication can be three, meaning one master and two slave brokers own replicas. It is suggested to set a minimum amount of in-sync replicas to two. Thereby one in-sync replica broker going offline does not affect the whole system. Using these in-sync replicas, Kafka minimizes the need to write messages from memory to disk to speed up the throughput. [9] All the scalability features in Kafka are closely related to fault tolerance and high availability. [18]

Data replication between multiple Kafka brokers in different geographical locations can be done using a command line tool MirrorMaker, which is shipped by default with Kafka. The keys in a topic need to understandable by MirrorMaker to ensure that ordering is sustained between brokers since partitions could be different between the two brokers per topic when scaling horizontally. [9] MirrorMaker is the tool that will be used when implementing geo-replication.

### 4.2.3 Reliability mechanisms

Kafka enables retaining and replaying the messages from an append-only log file, in which Kafka stores its messages, making them persistent and durable. Kafka does not remove the messages from the storage after they are consumed. Kafka retains by default all the messages for one week. After that, it removes the messages from the storage, regardless they are consumed or not. [9] Kafka leverages zero-copy principle for delivering data from the log file to the network socket, skipping the application layer, saving CPU resources.

When using replication, Kafka will save the messages into replicated partitions of the log. In case of a failure, Kafka will be able to recover from those replicas. [18]

Kafka offers three different message guarantees: at-most-once, at-least-once, and exactly-once. At-most-once does not guarantee message delivery, hence offering the best performance. At-least-once can lead to duplicate messages if acknowledgment is not delivered to the producer back. Exactly-once is implemented using transactions, meaning that messages are sent in batches, and they are all successfully deliver or not at all. [18]

Every topic in Kafka has its own log, which consists of one or more partitions, in which the received messages are stored in order. The ordering is guaranteed within each partition, meaning that ordering is not guaranteed between partitions.

Kafka supports four different compression algorithms: Gzip, Snappy, l4z, and Zstandard. [18]

### 4.2.4 Security mechanisms

Kafka supports mixes between authorized and unauthorized, as well as encrypted and non-encrypted setups. Kafka supports SSL/TLS encryption between broker-client, broker-broker, and broker-cli. Authentication between broker and client is supported using SSL/TLS, SASL, and Kerberos. Also, authentication between Kafka and ZooKeeper is available with SSL/TLS. By leveraging authorization using access control lists (ACL), Kafka can only allow certain users to write to or read from specific topics. Every Kafka broker should have their SSL/TLS certificate generated with an individual SSL/TLS key. [18]

### 4.2.5 Backward compatibility

Kafka is designed to be an always-on system that can be upgraded to a newer version without downtime using rolling update by guaranteeing backward compatibility with earlier versions. After broker upgrade, the Kafka clients need to be updated to newer version, but upgrading the broker first will not break the system. With newer versions of Kafka, it is also possible to upgrade clients before the broker, making them forward compatible. Implementing this needs special effort from a schema management point of view, but according to documentation, it is possible. In the documentation, there exist great step-by-step tutorials for implementing a rolling update. [18]

## 4.3 Pulsar

Pulsar is a high-performance messaging, and streaming platform that originated in Yahoo! in 2014, but its development is nowadays powered mainly by Chinese companies. Pulsar is mostly written Java and has been open sourced in 2016. It is now developed and maintained by Apache Software Foundation. Yahoo! developed Pulsar for the same need as Kafka has been developed, to replace traditional message brokers. Pulsar is also developed for high throughput and low latency with massive amounts of data. This approach always comes with a downside of reduced security and reliability features when comparing to traditional brokers. When the development started for Pulsar, Kafka was already open sourced, so Kafka's best parts were picked and some parts implemented differently that seemed not optimal. Pulsar cluster's architecture is three-layered, which includes Pulsar broker, BookKeeper server, and ZooKeeper server. [19]

### 4.3.1 Patterns

Pulsar supports pub/sub messaging pattern, which is based on topics. Consumers can make different kinds of subscriptions to these topics, which modify how the consumers communicate with the topic. Four different subscriptions are available: exclusive, failover, shared, and key_shared. By combining these subscriptions, different kinds of patterns are achievable, such as point-to-point. By combining these subscriptions, implementing Kafka-like consumer groups is also possible. [19] When a consumer connects to a topic, it specifies the subscription it is using, the subscription keeps track of the point at which the consumer is reading the topic. Figure 15 illustrates these subscriptions and their impact on consumer behavior.

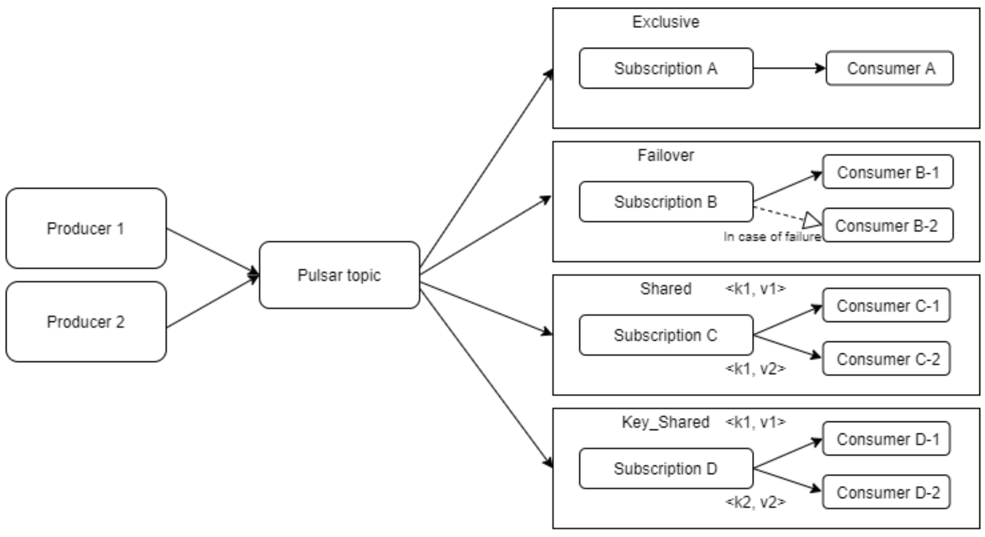

***Figure 15*** *Pulsar topic subscriptions.*

Exclusive subscription means that only one consumer is allowed to create a connection to the topic, rejecting all the other consumers from creating a connection.

Failover subscription means that multiple consumers can create a connection to a topic, but only one of them can consume. In case of the consuming service disconnecting from the topic, the next oldest connected service will continue consuming.

A shared subscription means that multiple consumers can create a connection to a topic, and they all can consume messages from it in a round-robin fashion. By using a shared subscription, the application can scale horizontally since the messages can be processed in parallel by multiple consumers.

Key_shared subscriptions work just like the shared one, but the messages are always delivered to the same consumer, based on the key in the message, rather than using round robin.

Pulsar aggressively batches the data as typical when aiming for high throughput. By default, Pulsar waits for an acknowledgment by the consumer that the message was processed. When using batching, the batch is acknowledged only when the consumer acknowledges all the messages of the batch.

Pulsar also has a feature called chunking that cannot be run simultaneously with batching. Chunking splits the messages when a message exceeds the defined size. Pulsar implements streaming of data using Pulsar Functions. Pulsar implements push-based communication between broker and consumer, but the API is designed to simulate pull-based communication. [19]

### 4.3.2 Scalability mechanisms

Pulsar can effectively replicate broker instances horizontally leveraging ZooKeeper, just like Kafka. ZooKeeper cluster handles the coordination of tasks between Pulsar broker instances when they are replicated. ZooKeeper also plays an important role when implementing geo-replication, which is a default functionality of a Pulsar cluster. Pulsar uses ZooKeeper also as metadata storage for cluster configuration.

In addition to traditional topic-based pub/sub, Pulsar also implements Kafka like partitioned topics. Normal topics in Pulsar can only be used with a single broker instance, meaning partitioned topics must be used when there are multiple brokers in the cluster. [19] Figure 16 illustrates how partitioned topics work in Pulsar.

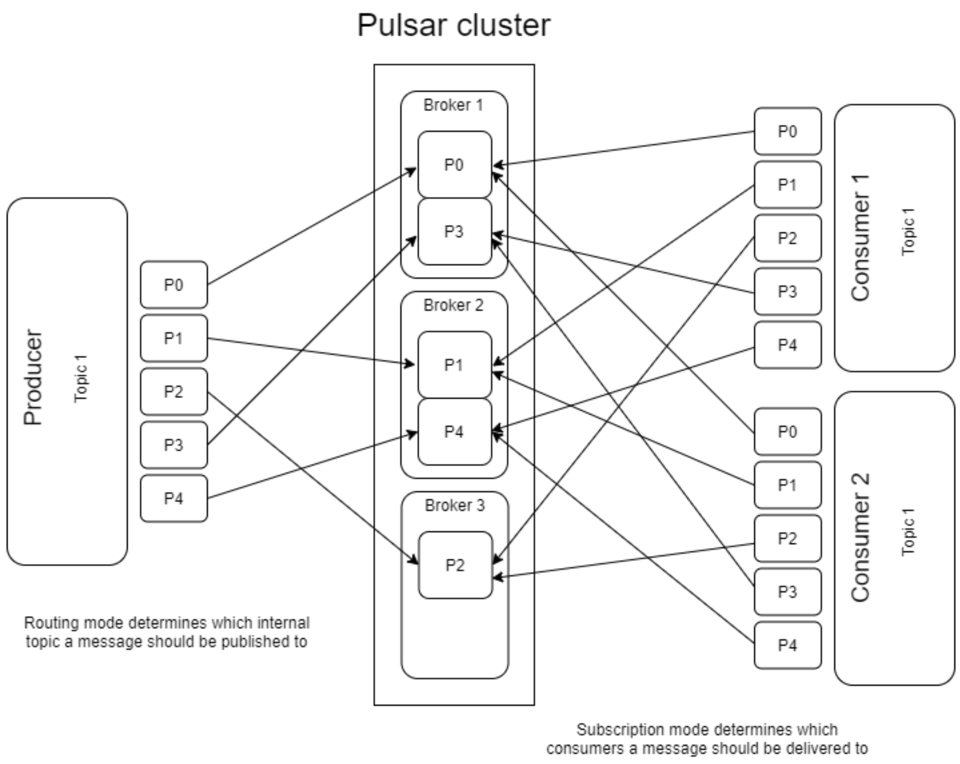
Pulsar cluster
Broker 1
Broker 2
Broker 3
P0
P1
P2
P3
P4
Producer
Topic 1
Consumer 1
Consumer 2
Routing mode determines which internal topic a message should be published to
Subscription mode determines which consumers a message should be delivered to

***Figure 16*** *An example of partitioned topics in a Pulsar cluster.*

These partitions are spread across the replicated broker instances to increase throughput. These partitions also have different kinds of routing modes that configure the way the topic partition is done on brokers. Pulsar automatically handles the partitioning of topics to the brokers. [19] Most of the scalability features are closely related to fault tolerance and high availability. [19]

### 4.3.3 Reliability mechanisms

By default, Pulsar retains the messages if there are no consumers at the receiving time, making the messages durable. Pulsar only stores by default all unacknowledged messages into persistent storage, making the messages persistent. Pulsar deletes by default all the acknowledged messages, but this can be changed from the configuration settings. There is also a possibility to use non-persistent topics, meaning the messages are only saved into the memory, meaning they are not persistent. [19]

In Pulsar cluster, BookKeeper take care of handling the data in the cluster. Pulsar broker handles incoming and outgoing messaging between broker and client. Pulsar uses BookKeeper as storage to save and retain messages to make them persistent. BookKeeper cluster consists of bookies that write to one or more of append-only data structures called ledgers, in which the messages are stored. The ledger is from an architectural point of view somewhat equal to the log file in Kafka. The ledger consists of segments that are somewhat equal to partitions of a log file in Kafka, but the segments are written as an ordered list and not concurrently like in Kafka. This segmentation of a ledger means that Pulsar writes only to the last segment, meaning the earlier segments are immutable. [19] Figure 17 illustrates the architecture of the Pulsar cluster.

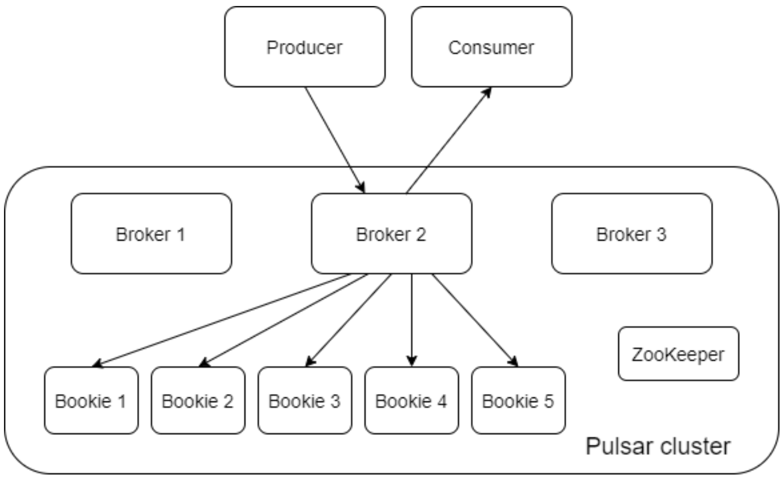

***Figure 17*** *An example of architecture of a Pulsar cluster.*

Pulsar offers a tiered storage functionality that allows older data to be sent from BookKeeper to a long-term and cheaper cloud storage such as Amazon S3. Since segments are written one by one as an ordered list, Pulsar can offload older immutable segments stored in the ledger one by one to the cloud storage. Even though the older segments are offloaded to the cloud storage, Pulsar can query that data using Pulsar SQL. [19]

Pulsar supports three message guarantees: at-most-once, at-least-once, and effectively-once. Effectively-once means that each message that is sent to a function, the function returns an output that is associated with the input message. That means Pulsar does not implement exactly-once guarantee, even though Pulsar supports transactions like Kafka. Transactions in Pulsar are applicable only for streaming and have a little different intend of use. The parallelism of consumption in exclusive and failover subscriptions is limited to the number of partitions in the topic. Parallelism can go beyond the number of partitions when using the shared subscription, but message ordering is not guaranteed.

Pulsar offers a strict ordering guarantee for messages per partition in the topic when using exclusive or failover subscriptions. The ordering guarantee can be implemented per partition or per producer, but typically per partition functionality is used. When a client subscribes to multiple topics, the message ordering is not guaranteed. It is guaranteed only for a single topic subscription. Also, Pulsar has a functionality to guarantee that messages are not saved to the persistent storage multiple times. Meaning that no duplicate messages can be saved no matter which message guarantee is used, but it does not guarantee the consumer will not consume the message multiple times.

Pulsar supports four different compression algorithms: zlib, Snappy, l4z, and Zstandard. [19]

### 4.3.4 Security mechanisms

By default, Pulsar does not use any encryption, authentication, or authorization features. Pulsar supports Advanced Encryption Standard (AES) end-to-end encryption from client to storage level. Encryption for communication between client-broker, broker-broker and broker-cli is supported with SSL/TLS. Like Kafka, by leveraging SSL/TLS keys, SSL/TLS certificates can be assigned to each broker.

Pulsar broker validates credentials when a client tries to connect to the broker. After authentication is done, the broker creates a role token that the connection is valid for a certain time. Pulsar offers four authentication methods: SSL/TLS, SASL, Athenz, Kerberos, and JSON Web Token Authentication.

Pulsar uses access control lists to implement authorization for clients based on the role tokens assigned in the authentication phase. Also, network segmentation can be used. [19]

### 4.3.5 Backward compatibility

According to Pulsar documentation, broker and client versions are backward compatible as well as forward compatible. Pulsar documentation has a great amount of tutorials for implementing canary testing as well as rolling update. Pulsar documentation suggests testing your upgrade with canary tests always first. Typically, there is no need to upgrade ZooKeeper instances, the broker and bookies are enough. When implementing rolling update, there needs to be taken into account, that bookies are stateful and broker is stateless. To implement the rolling update, brokers and bookies need to be upgraded one-by-one to not have any downtime. The upgrade is done by simply stopping the broker or bookie, upgrading it, and then restarting with few commands. [19]

## 4.4 RabbitMQ

RabbitMQ is a feature-rich traditional message broker originally started by Rabbit Technologies in 2007 but is since 2013 developed and maintained by VMware. RabbitMQ's source code is published under Mozilla Public License, and its written in Erlang. RabbitMQ supports multiple protocols to implement messaging: Advanced Messaging Queuing Protocol (AMQP), Streaming Text Oriented Messaging

Protocol (STOMP), Message Queuing Telemetry Transport (MQTT), and Hypertext Transfer Protocol (HTTP) using WebSockets. RabbitMQ was developed using AMQP 0-9-1, which is its core technology still to this day. Other technologies are available only by using a plugin. AMQP 1.0 and its latter versions are also available through a plugin because they share essentially nothing with the RabbitMQ's core version. RabbitMQ has improved the AMQP 0-9-1 by extending it from various functional aspects. [20]

AMQP is a binary protocol with rich features. It is implemented favoring security and reliability features over performance. The most significant benefit of AMQP is its interoperability between different vendors and proven reliability. AMQP is a wire-level protocol, meaning that it describes only the format of data that will be sent, making systems interoperable that are implemented in different languages. [20] The only messaging protocol to be taken under research is the core version of RabbitMQ.

### 4.4.1 Patterns

RabbitMQ supports point-to-point and pub/sub messaging patterns. As its name suggests, RabbitMQ started as a message queue, extended with pub/sub later on. RabbitMQ has four important terms that are used in the broker architecture. Exchanges are endpoints in the RabbitMQ broker, which the clients connect to send messages. Bindings are used to link the exchanges to queues, handling the routing logic depending on the binding key defined in the queue. Queues are components that handle message delivery to receivers and store messages by saving them into persistent storage. Virtual hosts are used to separating exchanges, queues, and users into groups to ease administration and access control. [20]

RabbitMQ supports four different exchange types. Direct exchange delivers the message into a queue defined by the binding key in the message header. Topic exchange delivers the message to the queues based on a routing filter defined between the exchange and the queue. Fanout exchange delivers the message to all bounded queues to the exchange, also known as a broadcast. Also, a headers exchange is supported, which bases the routing logic between the exchange and the queues on other message header attributes than the routing key. The difference between the routing key and the binding key is, that the routing key is defined in the message, whereas the binding key is defined in the queue. [20] Figure 18 illustrates how different exchange types work in RabbitMQ broker.

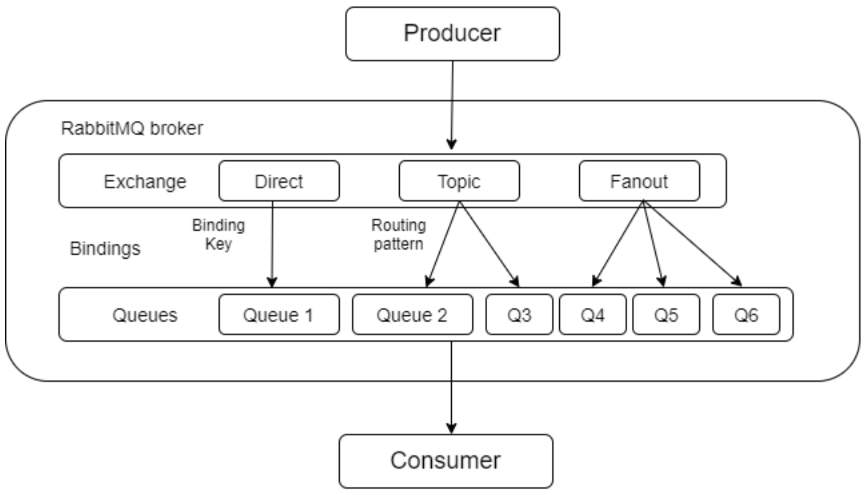

***Figure 18*** *An example of RabbitMQ broker and different exchange types.*

Virtual hosts are used in the RabbitMQ broker to enable hosting multiple isolated environments. Those environments include each their own users, exchanges, queues, and bindings. RabbitMQ cluster contains one or more instances of brokers, which all share exchanges, queues, bindings, virtual hosts, users, and other configurations.

RabbitMQ implements both push and pull models between broker and consumer, but the documentation says that the push model is highly recommended since the pull model is inefficient. [20]

### 4.4.2 Scalability mechanisms

RabbitMQ scales out by replicating broker instances into master/slave topology and replicating data across them. In case of a master broker failure, the messages will be automatically redirected to slave instances. This is implemented using mirrored queues, which means that the queues from the master broker are replicated to slave brokers. Once the message is acknowledged, the message will be removed from master and slave broker instances.

In RabbitMQ cluster, all the users, exchanges, and bindings are by default automatically replicated across the whole cluster. The only exception is the normal queues that only reside only on a single broker instance. In the cluster, which has multiple broker instances, all the queues remain visible and reachable for all broker instances, whether they are located in master or slave brokers. Once the master broker fails, the slave brokers will go through a new master election process and will continue with their new master/slave topology. It is highly advised not to create two-node clusters. Odd numbers should be preferred for master/slave topology to function properly. There is also a concept of shovel and federation to make RabbitMQ brokers distributed.

There is also a new approach to mirrored queues called quorum queues, which offer better throughput and reliability than mirrored queues. Quorum queues retain all the messages in their memory and on disk, meaning they have higher latency than mirrored queues. [20]

With mirrored queues, RabbitMQ saves messages in batches to the disk. The interval is on average about a few hundred milliseconds between disk calls. To increase performance, it is suggested that applications should process acknowledgments asynchronously in an unordered fashion. Queues can be assigned a priority property between one to ten. Higher the priority, the faster the message will be processed by the broker. [20]

### 4.4.3 Reliability mechanisms

In RabbitMQ, one can implement message persistence and durability by leveraging durable queues, exchanges, and persistent messages. In RabbitMQ, durable queues and exchanges mean that they can be recreated automatically after restart. Non-durable queues and exchanges will not be able to be recreated, and persistent messages will have no effect on this type of queues and exchanges. Message persistence mode should be defined in the message by the publisher, simply publishing into a durable queue and exchange will not make the message persistent. [20]

RabbitMQ leverages acknowledgments in broker-to-consumer messaging to ensure data delivery. In the publisher-to-broker messaging, there is a similar functionality, but it is called a confirm. Using confirms, the broker sends back a confirmation upon receiving the message to the producer. It is being called as confirm and not acknowledgement, since RabbitMQ has made there extensions on top of original AMQP acknowledgment logic. The acknowledgment can be configured to be sent at receiving or after processing the message. The confirm will be sent by the broker when the durable queue has persisted the message to the disk. [20]

There is also a concept called Lazy Queues that persist the messages into the storage as soon as possible, minimizing the time they are retained in memory. RabbitMQ cleans the message from the persistent storage periodically using garbage collection after the message is acknowledged. This means that RabbitMQ does not retain the messages after consuming like Kafka and Pulsar. [20]

Message delivery can be guaranteed only by using acknowledgments and confirms, and they can be redelivered if acknowledgment is not received. Acknowledgments do not guarantee that the message will not be lost due, e.g., networking error. Redelivery can lead to duplicated messages, meaning RabbitMQ offers at-least-once and at-most-once delivery guarantees.

RabbitMQ implements a FIFO pattern in its queues, but that is only per queue basis. That means RabbitMQ does not guarantee strict message ordering, and consumers should not rely on the message order. In general, RabbitMQ is very reliable and durable, but there are many ways to screw up.

RabbitMQ does not have any message encoding features supported. Publishers can still encode the messages before sending them using HTTP supported encoding, such as gzip. The open source version of RabbitMQ does not support inter-cluster message compression, which could greatly decrease the amount of data exchanged in the cluster. [20]

### 4.4.4 Security mechanisms

RabbitMQ has built-in support for SSL/TLS. RabbitMQ brokers accept connections from clients, other brokers, and administration tools. SSL/TLS can be used to secure all communications by leveraging certificates.

RabbitMQ supports multiple authentication mechanisms. RabbitMQ supports SAML authentication by default. There is also username/password-based internal mechanism. Also X.509 certificates can be used. The username/password-based mechanism is managed by the internal authentication backend. Users can also be granted limited access to different virtual hosts, and in them, users can have different kinds of permissions. There is also a possibility to use LDAP through a plugin. [20]

### 4.4.5 Backward compatibility

Upgrading RabbitMQ requires checking multiple steps when comparing to Kafka and Pulsar. For some versions, the broker must be first upgraded to an intermediate version to upgrade to the wanted version. RabbitMQ supports rolling update only between specific versions. A rolling update is available in 3.7.18 and latter versions. Earlier versions typically require a full cluster stop.

It is highly recommended to also upgrade the Erlang version together with RabbitMQ, since certain versions of them are not compatible. In addition, since RabbitMQ supports a large amount of plugins, their interoperability must also be verified. Upgrading a RabbitMQ cluster with multiple broker instances that have different versions is supported only for versions 3.8 and latter. RabbitMQ does not support downgrades. [20]

# 5 Comparison results

Benchmarking results are always in relation to the test execution environment and its performance. The included testbeds are described next per paper. The listing of the papers and all the related information can be found in Section 7.2.

## 5.1 Testbeds of each paper

[SP1] A cluster with 5 brokers with a CPU of 12 cores @ 2300.13Mhz, with 16GB of memory, HDD as a persistent storage and 1Gbps network connection. Single producer and consumer tests are run by scaling node amount from 1 to 5 with an amount of 1 million messages of size 50 Bit each. In case of multiple producers and consumers the node amount is kept constant and the amount of consumers and producers are scaled up. Persistent storage was not attached to RabbitMQ.

[SP2] A Linux server with 3.11 kernel, CPU of 24 cores (Intel Xeon X5660 @ 2.80GHz) and 12GB of RAM. The hard disk was WD1003FBYX-01Y7B0 running at 7200 rpm. RabbitMQ version was 3.5.3 and Kafka version 0.10.0.1 with default recommended configurations. New broker instances were always started on the same machine.

[SP3] A single machine with CPU of 12 cores @ 3.6Ghz, 16GB of memory and SSD as persistent storage. Kafka version 2.2.1, RabbitMQ 3.8.1, and Pulsar 2.6.0 with their default recommended configurations.

[SP4] No information about the hardware, but the tests were carried out for Kafka and RabbitMQ with a single broker on a single machine. No replication was allowed, meaning that writing and reading for Kafka were measured on a single topic and a single partition. For RabbitMQ, the tests were carried out in the same fashion, allowing only one queue with one partition.

[SP5] Server was deployed to UpCloud with Debian 10 operating system. CPU of 20 Core Intel(R) Xeon(R) Gold 6136 CPU @ 3.00GHz, memory of 32Gb RAM and 200Gb SSD as persistent storage. The tests were conducted on a single machine. Pulsar and Kafka were both used with their default configuration. Pulsar-client and broker versions 2.5.1 were used. Confluent-kafka was used as a client library with version 1.4.1 and Kafka broker version was 2.5.0.

[SP6] A server was running in openlandscape.cloud with Ubuntu 16.04LTS as an operating system. Each machine in the cluster had a CPU with eight cores, 16Gb RAM and 240Gb SSD. Kafka version 2.2.0 and Pulsar version 2.3.1. With a configuration to allow accepting large messages.

## 5.2 Which consumer-producer patterns do Kafka, Pulsar, and RabbitMQ implement?

According to study, RabbitMQ allows modifying the exchanges through an API, where as Kafka allows only the producer decide which partition to send the message, hence RabbitMQ has better routing functionalities. [SP2]

One study found that in terms of message patterns, RabbitMQ and Kafka are equal, but RabbitMQ does not have support for message batching. [SP4]

A Study compared push/pull implementations for all the compared technologies. In Kafka, producers push the messages to the broker, and consumers pull them from the broker. In RabbitMQ, the producer pushes messages to an exchange, which delivers them to the queue, and then the broker pushes the messages to the consumers. In Pulsar, the producer pushes the messages to the broker, and then the broker pushes them to the consumer. [SP3]

Kafka, Pulsar, and RabbitMQ support point-to-point and publish/subscribe patterns.

## 5.3 How do scalability mechanisms differ in Kafka, Pulsar, and RabbitMQ?

According to one study, Kafka and RabbitMQ support broker replication. Kafka leverages ZooKeeper, and it is built-in in RabbitMQ. RabbitMQ and Kafka both support master-slave architecture in their cluster. Kafka leverages partitions in data replication, and RabbitMQ mirrored queues. The difference is that new brokers added into the RabbitMQ cluster are by default visible to all the consumers. In Kafka, they are not visible because Kafka's architecture leverages consumer groups which can access the replicas of partitions that they are only assigned to. [SP1]

According to another study, Pulsar has the best scalability due to its three-layered architecture (Pulsar broker, ZooKeeper, BookKeeper) when comparing to Kafka and RabbitMQ. The same study argued that Kafka's scalability is better than RabbitMQ's since Kafka has two-layered architecture (Kafka broker and ZooKeeper). RabbitMQ had the worst scalability of all three. [SP3]

According to documentation, the Pulsar broker is implemented being stateless, and BookKeeper is stateful, meaning scaling can be done without re-partitioning existing data between brokers. Kafka brokers require re-partitioning. The Pulsar cluster supports multitenancy, meaning that multiple users can share a cluster using namespaces like in Kubernetes. Kafka does not support multitenancy. Pulsar is also possible to be integrated with RabbitMQ and Kafka, making it easy to integrate with existing systems. [18] [19]

When it comes to geo-replication, it is built-in in Pulsar, whereas MirrorMaker must be used with Kafka. With RabbitMQ, a plugin should be used to implement geo-replication by shoveling. [18][19][20]

## 5.4 How does throughput differ in Kafka, Pulsar, and RabbitMQ?

One study found that Kafka's latency decreased to one-third without any meaningful effect on the producer and consumer throughput when scaling from one to five brokers, one consumer, and producer per broker. The RabbitMQ's throughput also remained high. Kafka's latency dropped to one-third when scaling from one to five brokers. The study did not provide results for RabbitMQ's latency with a single producer and consumer per broker. [SP1] The same study found that latency in Kafka increased more than two orders of magnitude when an increasing amount of consumers and producers from one to five on five separate brokers, hence incoming traffic increasing 5x. The same study found that Kafka's producer throughput dropped about 20% and consumer throughput dropped 96The study found in the same test that the producer throughput about 75% and consumer throughput dropped about 10% in RabbitMQ. RabbitMQ's latency stayed low until it started increasing sharply after adding more than 15 consumers and producers, causing the queues to fill up, increasing latency. [SP1]

One study found that both, RabbitMQ and Kafka can deliver millisecond-level low latencies when using at-most-once when both read only from internal memory. Replication did not drastically affect the results. The same study found that when using at-least-once, RabbitMQ's latency was not really impacted when comparing to at-most-once. This result was argued to be a result of RabbitMQ retaining messages quite long in the memory, meaning that even if messages are written to the disk, they are also available in memory. The results show also that RabbitMQ's latency remains around 10ms in both message delivery guarantees. Kafka's latency remains with at-most-once in below 10 ms but increases to around 20ms when using at-least-once. Also, the results show that when reading from disk, the latency can grow up to 100 ms. [SP2] The study found that when using single node, single producer and consumer, single partition, and no replication, RabbitMQ outperforms Kafka. They also argued that, when replication is allowed, Kafka will outperform RabbitMQ. The same study found that RabbitMQ's throughput decreased by 50% when changing from at-most-once to at-least-once delivery guarantee when replication is allowed. The study also found that Kafka's throughput decreased by 50% to 75% when changing from at-most-once to at-least-once delivery guarantee when replication is allowed. [SP2]

One study argued that Kafka had the best throughput, Pulsar and RabbitMQ had somewhat equal throughput. [SP3]

Another study measured the relation between latency and throughput with varying message size and traffic. The study shows, Kafka overperforms RabbitMQ with 1 KB messages by around twice higher writing speed per second. In reading speed, Kafka outperforms RabbitMQ clearly with 3x to 10x better performance. The average latency was overall higher with Kafka than RabbitMQ. The same study shows that with 100 KB messages, RabbitMQ outperforms Kafka clearly in writing speed with 1000 messages per second, but otherwise, there were no significant differences in throughput between them. The latency with Kafka was still around 10x higher than RabbitMQ. [SP4]

One paper studied Pulsar's and Kafka's maximum throughput with message sizes of 1 KB, 65 KB, and 600 KB. The study found out that Pulsar had the highest maximum throughput with 1 KB messages. Kafka had nearly 35% lower maximum throughput with 1 KB messages. When using 65 KB messages, the study found instead that Kafka had 60% higher maximum throughput than Pulsar. The study argued that the result could be a result of a bottleneck in the Pulsar cluster. With 600 KB messages, Kafka outperformed Pulsar again with three times greater throughput. Kafka handled 870 messages/sec and Pulsar 293 messages/sec. In the test, the Pulsar cluster crashed with out-of-memory error and killed the BookKeeper node. This result shows that with a default configuration, the BookKeeper node is not able to handle big messages. This argument was based on the result that when using a non-persistent configuration, Pulsar was able to handle 600 KB messages with a maximum throughput of 2764 messages/sec. [SP5]

In the same paper, latency was also measured with varying message size and throughput. Messages of 1 KB were used with 1000 and 5000 messages/sec throughput. Messages of 65 KB were used with 100 and 500 messages/sec throughput. Lastly, messages of 600 KB were used with 50 and 200 messages/sec throughput. With 1 KB messages with both throughput rates, Pulsar's latency was around 1/4 of Kafka's. Pulsar managed sub-millisecond latencies with both throughput rates. With 65 KB messages, Pulsar's latency was half of Kafka's with a lower throughput rate, and with a higher throughput rate, Pulsar's latency was about 1/5 of Kafka's latency. Even though Kafka's throughput was significantly greater than Pulsar's, like mentioned earlier with message size of 600 KB, Pulsar's latency remains about half of Kafka's with both throughput rates. Kafka's maximum latency was 4-5x greater than Pulsar's. The study concluded that Kafka is the most reliable in terms of maximum throughput in their research. Still, if the Pulsar cluster would be more scaled out, the results could be significantly different. In latency, Pulsar outperformed Kafka in every aspect. [SP5]

One paper conducted testing with messages of size 1 KB and 1 MB between Kafka and Pulsar in terms of throughput, latency, ram usage, disk usage and bandwidth usage. The study concluded that Pulsar overperforms Kafka in all aspects of performance. With 1 KB messages, Pulsar had over twice greater throughput, and latency was 15 times smaller than Kafka's. In terms of CPU, RAM, and bandwidth usage, Pulsar outperformed Kafka in all aspects, except Kafka slightly won in overall CPU usage of the whole cluster. With 1 MB messages, the gap decreased between them. Pulsar had around 27% higher throughput, but Pulsar's latency was still around 15 times smaller than Kafka's. Also, in terms of CPU, RAM, and bandwidth usage, Pulsar outperformed Kafka in all aspects, except disk writing speed. The study concluded that Kafka still has greater community support and a rich ecosystem, hence making it still a good choice. [SP6]

Kafka supports compression with Gzip, Snappy,l4z, and Zstandard. [18] Pulsar supports zlib, Snappy,l4z, and Zstandard. [19] RabbitMQ does not support any compression algorithms since AMQP does not support them. Still, some compression can be done on the client side using default HTTP supported compression, such as Gzip. RabbitMQ does not support inter-node message compression in its open source version. [20]

## 5.5 How do reliability mechanisms differ in Kafka, Pulsar, and RabbitMQ?

A study found that if reliability and data safety is essential, such that losing a single message can be critical, RabbitMQ is a better choice than Kafka. RabbitMQ and its AMQP protocol have a wider range of security features and still can maintain a good performance when comparing to Kafka. [SP1]

According to a study, RabbitMQ has a feature to sort messages sent into a queue, whereas Kafka can only retain order per partition. Since Kafka leverages heavily batching and aims for high throughput, it is designed such that the whole batch must be resent to correct the ordering in the partition. RabbitMQ can correct that ordering in the queue by leveraging acknowledgments, and since messages are by default retained in memory, there's no need to resend a full batch. [SP2]

A study argued that Pulsar had the best ordering guarantee when compared to RabbitMQ and Kafka, Kafka had the second-best ordering guarantee, and RabbitMQ the worst. The same study argued that there were no significant differences in delivery guarantees between all three systems, even though RabbitMQ does not implement exactly-one guarantee. The same study argued also, that overall reliability was the best in RabbitMQ, and Pulsar and Rabbit just a little bit worse. [SP3]

One study concluded that in terms of reliability, Kafka is more reliable than RabbitMQ since it has more features it supports. The main difference the study found is that RabbitMQ does not support replication synchronization and idempotent messaging. [SP4]

## 5.6 Which security mechanisms do Kafka, Pulsar, and RabbitMQ support?

Kafka supports SSL/TLS encryption in broker-client, broker-broker, broker-ZooKeeper and broker-cli connections. Authentication can be done in Kafka using between them using SSL/TLS, SASL, and Kerberos. Authorization is done in Kafka using ACLs. [18]

Pulsar supports Advanced Encryption Standard (AES) end-to-end encryption from client to storage level. Pulsar also supports SSL/TLS to encrypt messaging between client-broker, broker-broker, and broker-cli. Authentication can be done in Pulsar using SSL/TLS, Athenz, Kerberos, SASL, and JSON WTA. Authorization is done in Pulsar using ACLs or network segmentation. [19]

RabbitMQ supports encrypting messages with SSL/TLS between client-broker, broker-broker, and broker-client. Two primary authentication mechanisms for clients are username/password and X.509 certificates (SSL/TLS), but RabbitMQ also supports SASL. RabbitMQ can implement authorization using separate virtual hosts assigned to different users. Also, ACL-type of permission list can be used in those virtual hosts to grant or deny operations. [20]

## 5.7 How does backward compatibility differ in Kafka, Pulsar, and RabbitMQ?

Kafka broker is backward compatible and supports rolling update. Client versions are forward compatible. Documentation does not describe in detail the actual implementation of the rolling update, what kind of effects that have on partitioning and assigned clients. Downgrading is supported if something goes wrong in the rolling update. [18]

According to Pulsar's documentation, broker and client versions are backward compatible as well as forward compatible. The main benefit that already was mentioned in regards to upgradability is that the Pulsar broker is stateless. By storing data in BookKeeper, the brokers can be upgraded without a need for re-partitioning. Pulsar supports rolling update and downgrade in case of an error. [19]

RabbitMQ supports rolling update with newer versions, but it is only possible between certain versions if update is done from 3.7.18 version or earlier. Sometimes with older versions, updating between the wanted versions requires an update to an intermediate version first. It is recommended also to upgrade the Erlang version with the broker, and on top of that, the certain broker versions and Erlang versions are not compatible. Upgrading a RabbitMQ cluster with multiple versions of broker instances is supported only for versions 3.8 and latter. RabbitMQ does not support downgrades. [20]

# 6 Discussion

Comparing these messaging systems by the defined characteristics, some certain differences can be found. For example, using persistent storage is possible in each technology but not required in RabbitMQ. Out of all three, RabbitMQ implements many things differently since it is a traditional message broker, not aiming for high throughput. Kafka and Pulsar are quite similar in how they handle data. Pulsar and Kafka's main difference is that Pulsar uses a separate data storage system called BookKeeper, and Pulsar has tiered storage. This three-layered architecture with ZooKeeper and BookKeeper increases scalability by enabling the Pulsar broker's statelessness compared to Kafka's two-layered architecture and its stateful broker. Also, some research shows that Pulsar with proper configuration can outperform Kafka in nearly all of the performance-related aspects, including throughput and latency [SP6]. Also, Pulsar's end-to-end encryption with AES, built-in geo-replication, and multi-tenancy is something Kafka does not support by default. Pulsar supports a wider amount of authentication mechanisms and authorization mechanisms when comparing to Kafka.

The best choice as a MOM is always use case specific. Meaning, testing should always be done before jumping into conclusions, simply based on some results that can be read on the site of one solution. Typically, every MOM tries to market their own solution on their site and show test results in use cases in which perform well.

Where Pulsar shines in performance and scalability, it still lacks some community support and technological maturity when comparing to Kafka. The technological maturity will probably come with the time, but the future will show how it gets adapted from the community. In Table 3, some information is gathered from repositories of Kafka, Pulsar, and RabbitMQ on Github.

***Table 3*** *Repository comparison of the chosen technologies.*

| | apache/kafka | apache/pulsar | rabbitmq/rabbitmq-server |
|---|---|---|---|
| Stars | 18189 | 7411 | 8148 |
| Forks | 9676 | 1865 | 3067 |
| Open Issues | 807 | 1163 | 217 |
| Age | 10 years | 5 years | 10 years |
| Language | Java, Scala | Java | Erlang |
| License | Apache-2.0 | Apache-2.0 | Mozilla Public License 2.0 |

There exist many hosted offerings of the chosen technologies. Confluent offers Kafka commercially to users using their Confluent Platform and Confluent Cloud.

RabbitMQ is hosted by Cloud AMQP and Cloud Foundry. Pulsar is currently hosted in Alibaba Cloud, called Pulsar-as-a-service, and StreamNative also has their offering for Pulsar in their cloud.

Confluent currently works on the 'Next-Generation event streaming platform' called Project Metamorphosis, which aims to lower cost and better performance than their current Kafka offering. The future will show whether it will be open sourced or not. [22]

There is also some effort being put into a general benchmarking framework to overcome biased results when benchmarking MOMs. It is called OpenMessaging Benchmark Framework. It is currently a Linux Foundation Collaborative Project and is being supported, for example, by Alibaba. According to README.md in their Github repository, they support benchmarking for Kafka, RocketMQ, RabbitMQ, Pulsar, and NATS Streaming. [23]

# 7 Conclusion

Kafka remains the number one solution for most event streaming applications that demand high throughput due to its rich ecosystem and wide community adoption. RabbitMQ is the choice for reliable and safe messaging. Pulsar is something to keep an eye on. It could be replacing Kafka if it gets adopted by the community. Pulsar is a great tool for a software architect to have as a Kafka alternative. Table 4 in Chapter 7.2 summarizes all three technologies.

When selecting papers for the research, it was hard to find papers that would be comparable. Usually, papers only concerned with a single technology with varying testbeds, meaning they did not provide any value to the comparison. In some of the papers, there is sometimes clear false information to one's eye. That is maybe due to updates to the systems or just general differences in opinions. Meaning that one could, for example, argue that RabbitMQ implements batching using 'consumer prefetch'. It is not actually batching but kind of close, so it is not really a false argument. Or one could argue that Kafka does not implement exactly-once message delivery because using it would 'kill the performance'. Implementing it is technically possible but complicated. These opinion differences allow the firms behind each technology to make some immoral marketing arguments. Also, some chosen papers did not describe the testbed or hardware in detail enough to make the tests fully comparable. Any of the papers did not compare security or backward compatibility related functionality in detail.

For future work, some comparisons should be done about how authentication and encryption mechanisms affect throughput and latency. Also, some research should be done about finding optimal hardware configurations such as CPU speed and memory size. As technologies evolve, general throughput and latency benchmarking continue to be also relevant. These tests should be done using OpenMessaging Benchmark Framework to maintain comparability in results.

## 7.1 Selected Papers

[SP1] Vineet John, Xia Liu, A Survey of Distributed Message Broker Queues, 2017, `https://arxiv.org/pdf/1704.00411.pdf`

[SP2] Dobbelaere P., Esmaili K.S., Industry paper: Kafka versus RabbitMQ: A comparative study of two industry reference publish/-subscribe implementations, DEBS 2017 - Proceedings of the 11th ACM International Conference on Distributed Event-Based Systems, `https://dl.acm.org/doi/10.1145/3093742.3093908`

[SP3] Guo Fu, Yanfeng Zhang , Ge Yu, A Fair Comparison of Message Queuing Systems, December 22, 2020, `https://ieeexplore.ieee.org/document/9303425`

[SP4] Bondarenko A., Zaytsev K., Studying systems of open source messaging, Journal of Theoretical and Applied Information Technology, 15th October 2019, `http://www.jatit.org/volumes/Vol97No19/12Vol97No19.pdf`

[SP5] Sebastian Tallberg, A Comparison Of Data Ingestion Platforms In Real-time Stream Processing Pipelines, June 7, 2020, `https://www.doria.fi/bitstream/handle/10024/177865/tallberg_sebastian.pdf?sequence=2&isAllowed=y`

[SP6] Intorruk S., Numnonda T., A Comparative Study on Performance and Resource Utilization of Real-time Distributed Messaging Systems for Big Data Proceedings - 20th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2019, `https://ieeexplore.ieee.org/document/8935759`

## 7.2 Comparison Table

***Table 4*** *Comparison table of the chosen technologies.*

| | Kafka | Pulsar | RabbitMQ |
|---|---|---|---|
| Developed aiming for | High throughput and low latency | High throughput and low latency | Reliability and data safety |
| Messaging patterns | Point-to-Point, Pub/Sub | Point-to-Point, Pub/Sub | Point-to-Point, Pub/Sub |
| Push or Pull with consumer | Pull | Push, API simulates pull | Both, but push preferred |
| Message durability | Yes | Yes | By using durable queues |
| Message persistence | Yes | Yes, Also tiered storage | Possible, but does not by default have it |
| Message reliability mechanism | Acknowledgements | Acknowledgements | Acknowledgements and confirms |
| Message delivery guarantees | At-least-once, At-most-once, Exactly-once | At-least-once, At-most-once, Effectively-once | At-least-once, At-most-once |
| Message ordering | Per partition | Per partition with partitioned topics | Per queue |
| Data replication mechanism | Partitions in a log | Bookies, partitions and segments | Mirrored Queues, Quorum Queues |
| Broker replication mechanism | ZooKeeper handles replication | ZooKeeper handles replication | Build-in replication, Shoveling, Federation |
| Geo-replication | Using MirrorMaker | Built-in functionality | Hard to implement |
| Message batching | Yes | Yes | Hard to implement |
| Supports streaming | Yes | Yes | No |
| Smart broker, dumb client | No | No | Yes |
| Compression types | Gzip, Snappy, l4z and Zstandard | Zlib, Snappy, l4z and Zstandard | None |
| Encryption | SSL/TLS | SSL/TLS, AES | SSL/TLS |
| Authentication | SSL/TLS, SASL, Kerberos | SSL/TLS, Athenz, Kerberos, SASL, JSON WTA | SSL/TLS, SASL, username and password |
| Authorization | ACL | Permissions, Tokens | vHosts, Permissions |
| Backward compatibility | Good | Excellent | Only between certain versions |
| Rolling udpate | Supported | Supported | Possible but complicated |

# 8 References

[1] Maarten van Steen, Andrew S. Tanenbaum, A brief introduction to distributed systems, Springer, 16 August 2016, pp.968-991, https://link.springer.com/article/10.1007/s00607-016-0508-7/.

[2] James F. Ransome, John W. Rittinghouse, Cloud Computing, CRC Press, 2017, Chapters Introduction-3,6, https://www.oreilly.com/library/view/cloud-computing/9781439806814/.

[3] Usha Divakarla, Geetha Kumari, An Overview of Cloud Computing in Distributed Systems, AIP Conference Proceedings 1324 184, 03 December 2010, pp.1-4, https://doi.org/10.1063/1.3526188.

[4] Blesson Varghese, Rajkumar Buyya, Next generation cloud computing: New trends and research directions Volume 79, Part 3, Elsevier, February 2018, pp.849-861, https://doi.org/10.1016/j.future.2017.09.020.

[5] Boris Scholl, Trent Swanson, Peter Jausovec, Cloud Native, O'Reilly Media Inc., 16 August 2016, Chapters 1-5, https://learning.oreilly.com/library/view/cloud-native/9781492053811/.

[6] John Gilbert, Cloud Native Development Patterns and Best Practices, Packt Publishing, February 2018, Chapters 1-3, https://learning.oreilly.com/library/view/cloud-native-development/9781788473927/.

[7] Adam Wiggins, The Twelve-Factor App, cited November 2020, https://12factor.net/.

[8] Mark Richards, Software Architecture Patterns, O'Reilly Media Inc., February 2015, Chapter 2, https://learning.oreilly.com/library/view/software-architecture-patterns/9781491971437/.

[9] Message Brokers, cited December 2020, https://www.ibm.com/cloud/learn/message-brokers/.

[10] Adam Bellemare, Building Event-Driven Microservices, O'Reilly Media, Inc, July 2020, Chapters 1-3, https://learning.oreilly.com/library/view/building-event-driven-microservices/9781492057888/.

[11] Ben Stopford, Designing Event-Driven Systems, O'Reilly Media, Inc., July 2020, Chapters 5,7, https://learning.oreilly.com/library/view/designing-event-driven-systems/9781492038252/.

[12] Jakub Korab, Understanding Message Brokers, O'Reilly Media, Inc., June 2017, Chapter 1, https://learning.oreilly.com/library/view/understanding-message-brokers/9781492049296/.

[13] Message Delivery Reliability, cited December 2020, https://doc.akka.io/docs/akka/current/general/message-delivery-reliability.html?language=scala.

[14] Kevin Hoffman, Dan Nemeth, Cloud Native Go, Addison-Wesley Professional, December 2016, Chapter 8, https://learning.oreilly.com/library/view/cloud-native-go/9780134505787/.

[15] Barbara Kitchenham, Pearl Brereton, A systematic review of systematic review process research in software engineering, Volume 55, Issue 12, December 2013, Pages 2049-2075, https://doi.org/10.1016/j.infsof.2013.07.010.

[16] B. Kitchenham , S Charters, Guidelines for performing Systematic Literature Reviews in Software Engineering, 2017, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.117.471.

[17] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, Anders Wesslén, Experimentation in Software Engineering, 2012, https://books.google.fi/books?id=QPVsM1_U8nkC.

[18] Kafka Documentation, sited in February 2021 https://kafka.apache.org/documentation/.

[19] Pulsar Documentation, sited in February 2021 https://pulsar.apache.org/docs/en/standalone/.

[20] RabbitMQ Documentation, sited in February 2021 https://www.rabbitmq.com/documentation.html.

[21] Kafka Confluence, https://cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+Quorum.

[22] Unveiling the next-gen event streaming platform, confluence.io, https://www.confluent.io/project-metamorphosis/.

[23] OpenMessaging Benchmark Framework, https://github.com/openmessaging/openmessaging-benchmark/.