On the Impact of Message Brokers Implementations in the Choreography of Microservices

Ahmed Gamal Ibrahim(✉), Rui Pedro Lopes, José Rufino, and Paulo Leitão

Research Center in Digitalization and Intelligent Robotics (CeDRI), Laboratório Associado para a Sustentabilidade e Tecnologia em Regiões de Montanha (SusTEC), Instituto Politécnico de Bragança, Campus de Santa Apolónia, 5300-253 Bragança, Portugal
ahmed@ipb.pt

Abstract. Communication brokers are essential in modern software development to enable efficient, reliable, and scalable message passing within microservices architectures. However, flawed or delayed communication could be a massive setback that prevents achieving real-time analytics. This paper compares four prominent brokers: Apache Kafka, ActiveMQ Artemis, RabbitMQ, and NATS. Their performance is evaluated in terms of latency, throughput, scalability, and reliability, particularly in the clients implemented in the Java (SpringBoot) and Python languages. Experiments that were conducted in a standardized environment showed that Kafka offers great performance in real-time data processing with its low latency and high reliability. ActiveMQ Artemis provides reliable performance but not without drawbacks as it shows much higher latency. RabbitMQ showed competitive latency but faced some issues in cases of network disruptions. NATS, designed for low-latency and high-throughput scenarios, showed excellent scalability and throughput in all the different scenarios.

Keywords: Microservices · Message brokers · Choreography · Performance

1 Introduction

Communication brokers are vital in modern software development as they enable efficient, reliable, and scalable message passing in distributed systems. As applications transition from monolithic to modular microservices architectures, the selection of an appropriate broker becomes critical for ensuring robust inter-service communication.

Microservices architectures emphasize flexibility and scalability based on multilanguage development and independent scaling of services. They make use of asynchronous communication to control data flow and enhance fault tolerance.

A. I. Pereira et al. (Eds.): OL2A 2025, CCIS 2617, pp. 3–17, 2026.
https://doi.org/10.1007/978-3-032-00137-5_1

The brokers reliably transmit messages even under network disruptions, and support patterns like point-to-point, publish-subscribe, and request-reply, which influence the system's performance and scalability.

This paper provides a comparative analysis of four renowned brokers—Apache Kafka, ActiveMQ Artemis, RabbitMQ, and NATS—assessing their performance, scalability, and reliability under conditions that mimic real-world usage. The study benchmarks these brokers in Java-to-Java, Java-to-Python, Python-to-Java, and Python-to-Python scenarios, reflecting the complexity of today's distributed systems. Moreover, it explores their scalability in different workloads to provide insights in order to select the right broker for each specific application.

The evaluations are conducted using standardized virtual machines and Docker containers to ensure consistency and replicability. Performance metrics, including latency and throughput, are measured across different number of messages and communication patterns, providing actionable insights for software architects and engineers.

The structure of this paper is as follows: Sect. 2 reviews the state-of-the-art in communication brokers. Section 3 discusses the experimental setup and methodology. Section 4 presents the findings, followed by a discussion. Finally, Sect. 5 summarizes key points and insights and outlines future research directions.

2 Communication in Microservices Architecture

Microservices architecture has now become a central component of modern software development as it is characterized by its modular approach in which complex applications are composed of small, independent processes communicating with each other using language-agnostic APIs. This section discusses communication in microservices-based architectures.

2.1 Core Concepts in Messaging Systems

Modern communication systems are designed to allow producers and consumers to interact asynchronously, making sure that information is shared reliably and that it stays consistent [12].

At the center of these systems are messages—think of them like digital envelopes. Each message has two main parts: the body, which carries the actual information or data, and the headers, which act like labels containing extra details such as where the message needs to go or how it should be handled. These messages can be represented in different formats like JSON, XML, or even compact binary formats, depending on what the system or application requires.

When it comes to how messages are delivered, most systems offer two main styles: queues and topics. Queues are like passing a note directly to one person, only the intended recipient gets it. Topics, on the other hand, are more like a group announcement, where the same message can be sent to many people at

once. This flexibility makes messaging systems highly adaptable, fitting a wide range of needs and scenarios.

Expanding beyond queues and topics, advanced communication models combine aspects of both types to handle complex consumer requirements. Hybrid models enable independent consumer scaling, and optimize high-demand scenarios by balancing load across large consumer groups. In such cases, Complex Event Processing (CEP) comes into play, allowing brokers to filter, transform, and aggregate data for insights in real time. CEP is quite important in use cases like fraud detection and telemetry monitoring as in such cases rapid data processing is crucial [6].

Security in messaging systems is reinforced through multiple layers, including end-to-end encryption, authentication, and authorization. Some brokers, such as RabbitMQ, use TLS/SSL to encrypt messages in transit, while others, like Kafka, incorporate pluggable security protocols like OAuth and SASL for flexible authentication. These protocols protect sensitive data streams in sectors like finance and healthcare, which ensures compliance with industry standards like the Health Insurance Portability, Accountability Act (HIPAA), and the General Data Protection Regulation (GDPR). Role-Based Access Control (RBAC) adds an authorization layer, securing access at a granular level and helping prevent unauthorized data flow within the microservices ecosystem.

2.2 Overview of Key Communication Brokers

Communication brokers act as the backbone of microservices architectures, acting as intermediaries that enable efficient and reliable communication between different microservices. Their importance is shown in facilitating scalability, load balancing, and resilience within distributed systems. Batista et al. (2024) [3] extend this discussion by exploring efficient strategies for managing asynchronous workloads in a multi-tenant microservice architecture, thereby illustrating the critical role of communication brokers in guaranteeing operational efficiency and reliability in distributed systems.

Apache Kafka, as documented by various studies such as those conducted by Chy et al. [5] and Maharjan et al. [13], has proved to show high throughput and durability in handling large volumes of messages. It plays a vital role in real-time data processing and integration with advanced machine learning frameworks and big data management tools. Furthermore, Ataei et al. (2023) [2] demonstrate Kafka's effectiveness in a publish/subscribe model designed for real-time data processing in Massive IoT (MIoT) environments. This model improves the scalability and security of MIoT ecosystems by integrating blockchain technology for data storage, showcasing Kafka's abilities to support advanced data processing and storage solutions. T and K (2019) [14] also praise Kafka's scalability and fault tolerance as they are important factors for ensuring data availability and robustness in distributed systems in general. Additional studies underline Kafka's high throughput and low latency, which are essential qualities for real-time applications [9,11]. Kafka's architecture, which includes data partitioning and replication, ensures high availability and fault tolerance, making it

a reliable choice for big data streaming applications [10,15]. Artemis is the modern successor to the original ActiveMQ, and it distinguishes itself with improved performance and versatility, supporting a wide range of messaging protocols and communication patterns. Insights from Chy et al. (2023) [5] and Fu et al. (2021) [8] point out Artemis's efficient resource utilization, and that makes it ideal for resource-constrained microservices architectures. The flexibility in protocol support and communication patterns puts Artemis as a reliable option for a number of different application scenarios. Maharjan et al. (2023) [13] note its balanced performance in latency and throughput, offering a stable solution for enterprise-level applications.

Renowned for its efficient performance and protocol versatility, RabbitMQ supports deployment in both on-premise and cloud settings, providing support for multiple messaging protocols. Research by T and K (2019) [14] suggests that RabbitMQ's complex routing capabilities and protocol support make it a candidate for applications that prioritize reliability and guaranteed message delivery, such as financial transactions. Despite its versatile features, Maharjan et al. (2023) [13] found RabbitMQ to lag behind Kafka when it comes to throughput, but it offers competitive latency metrics, making it a viable choice for various messaging needs. RabbitMQ's abilities to handle complex routing and to ensure message delivery have been well-documented, underscoring its reliability for critical applications [7].

NATS is characterized by its high-performance and lightweight messaging capabilities, making it particularly suited for microservices, IoT, and cloud-native applications. NATS offers multiple communication models, including publish-subscribe, request-reply, and queue groups, all within a single platform, which makes it a valid option for many different settings. For instance, it makes NATS an excellent choice for dynamic network environments where low latency and secure communication are essential [1]. Additionally, according to T and K (2019) [14], NATS provides a setup process and message handling speed that are competitive, making it fit well in modern, cloud-based high-performance applications.

Recent studies provide insights into the performance and scalability of different message queuing systems, highlighting differences between various brokers' capabilities. These analyses provide important information for selecting the most appropriate communication broker based on specific use-case requirements. Ataei et al. (2023) [2] expand on this by introducing a publish/subscribe model that uses Apache Kafka for real-time data processing in Massive IoT (MIoT) environments. Their approach, which employs blockchain technology for secure data storage, showcases the potential for communication brokers to support efficient, scalable, and secure frameworks within IoT ecosystems. This demonstrates the growing role of message brokers in enabling modern efficient solutions for data processing challenges in distributed systems.

The continuous development and refinement of messaging systems will continue to shape their adoption and effectiveness in modern distributed systems.

2.3 Comparative Analysis

To evaluate the effectiveness of communication brokers within microservices architectures, we reference key findings from Maharjan et al. [13] and the comparison conducted by Fu et al. [8]. The studies benchmark Kafka, RabbitMQ, RocketMQ, ActiveMQ, and Pulsar across various criteria, including throughput, latency, scalability, and feature set.

Evaluation Criteria. The comparative tests were conducted using the following evaluation criteria:

- *Throughput:* Measured as the number of messages processed per second under various workloads and message sizes.
- *Latency:* The time taken for a message to be transmitted from producer to consumer.
- *Scalability:* Assessed by varying the number of producers, consumers, and partitions to determine the system's ability to handle increased workloads.
- *Reliability:* Evaluated based on the system's fault tolerance, message delivery guarantees (at-most-once, at-least-once, and exactly-once), and recovery mechanisms.

Throughput and Latency. Kafka consistently outperforms other brokers in throughput, as demonstrated in controlled experiments. The high throughput is attributed to optimizations such as zero-copy technology and efficient disk I/O operations [8]. However, RocketMQ excels in latency-sensitive applications, achieving latency below 10ms in most scenarios, which is made possible by optimizations like reduced JVM pauses and page cache latency.

Scalability and Reliability. Pulsar's broker-bookie architecture offers excellent scalability by decoupling storage from messaging. This design enables smooth horizontal scaling and quick recovery from broker failures whenever needed. In contrast, Kafka relies on partition replication for high availability, ensuring robustness in distributed environments.

Use Cases and Best-Suited Scenarios. Different brokers are best suited for specific application scenarios:

- *Kafka:* Ideal for high-throughput applications, such as log aggregation, stream processing, and big data pipelines [13].
- *RabbitMQ:* Preferred for enterprise applications requiring complex routing and protocol versatility, though it lags behind Kafka and RocketMQ in throughput [8].
- *ActiveMQ:* Offers robust JMS support, making it suitable for legacy systems and point-to-point messaging.
- *NATS:* Ideal for scenarios requiring simplicity, low latency, and lightweight deployment, such as microservices architectures, real-time communications, IoT, and edge computing [4].

3 Experimental Methodology

This research undertakes a comparative analysis of four principal communication brokers widely employed in microservices architectures: Kafka, ActiveMQ Artemis, RabbitMQ, and NATS.

The study focuses on evaluating the default messaging protocols employed by these brokers and their efficacy in supporting reliable communication between microservices developed using Java and Python. The different client implementations may also contribute to the results, and, as such, multi-language implementations are combined. The main purpose is to assess the performance metrics and compatibility across four different interaction scenarios: Java-to-Java, Java-to-Python, and Python-to-Python communications.

3.1 Environment Setup

Development of Java applications is done using JDK 11 coupled with the Spring-Boot framework, whereas Python applications are developed employing Python 3.8 with the FastAPI framework. To ensure environmental consistency and replicability of results, all applications are containerized using Docker.

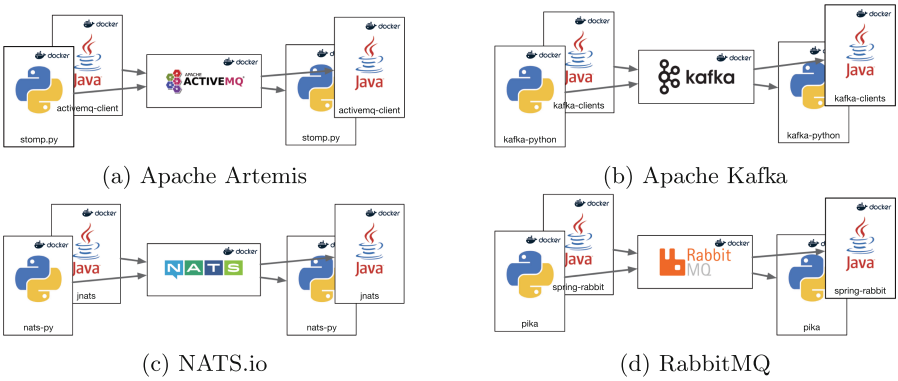
Each broker is scrutinized under its default configuration with particular attention to its supported messaging protocols:

- *Kafka:* Operates primarily on a custom TCP-based protocol optimized for high-throughput scenarios.
- *ActiveMQ Artemis:* Implements Advanced Message Queuing Protocol (AMQP) by default, which supports a wide range of cross-language clients.
- *RabbitMQ:* Uses AMQP as its main protocol, making it possible to deal with complex routing and allowing reliable message delivery.
- *NATS:* Employs a straightforward TCP-based publish-subscribe protocol focusing on high performance and scalability in lightweight environments.

Experiments are conducted on standardized virtual machines, each configured with identical CPU, memory, and network resources, to reduce any inconsistencies that could show up from hardware variability. The specific configurations of the VMs used in the experimental setup are as follows: CPU - 8 vCores (4 real cores, 2 threads per core) of a physical AMD EPYC 7351 16-Core CPU; RAM: 32 GB; Network - bridged vNIC on top of a physical 100 Gbps Ethernet NIC; Secondary Storage: 64 GB virtual disk on a PCIe 4.0 NVMe SSD.

These configurations were provided to ensure enough resources are used to handle the workloads and messaging scenarios under test. Co-locating the VMs on the same virtualization server minimized external network variability, allowing for a more controlled environment to benchmark broker performance. Identical configurations were maintained across all experiments to ensure consistency and fairness in the results.

The benchmarking architecture defined four scenarios for each broker, combining Java and Python libraries as both the producer and the consumer,



(a) Apache Artemis (b) Apache Kafka

(c) NATS.io (d) RabbitMQ

Fig. 1. The sixteen configurations for benchmarking

although keeping the platform and network conditions unchanged, in a total of 16 scenarios (Fig. 1).

To implement and test various communication brokers, both Java and Python environments used multiple key libraries. The latest supported versions of each library were used during testing to ensure optimal compatibility and performance (Table 1).

Table 1. Libraries used in each language

Broker	Language	Library	Version
Artemis	Java	activemq-client	6.1.4
Artemis	Python	stomp.py	8.2.0
Kafka	Java	kafka-clients	3.9.0
Kafka	Python	kafka-python	2.0.2
NATS	Java	jnats	2.20.2
NATS	Python	nats-py	2.9.0
RabbitMQ	Java	spring-rabbit	3.2.0
RabbitMQ	Python	pika	1.3.2

Regarding ActiveMQ Artemis, the org.springframework.boot:spring-boot-starter-artemis dependency simplifies the integration of Artemis within Spring Boot applications. It provides built-in configuration and support for the Java Message Service (JMS), enabling easy setup and messaging functionalities with minimal code. The stomp.py library offers a client interface for interacting with brokers that use the STOMP protocol, including ActiveMQ Artemis. It supports subscribing to queues and topics, enabling message exchange and handling. It is worth mentioning that STOMP was used instead of AMQP here due to compatibility issues.

For Apache Kafka, the spring-kafka library was used to produce and consume messages from Kafka topics within Java applications. It provides high-level abstractions over Kafka's native APIs. The confluent-kafka library is a Python client for Apache Kafka. It enables interaction with Kafka topics, providing message production and consumption capabilities.

NATS provides libraries for multiple languages and, the Java io.nats:jnats dependency provides a lightweight client for communicating with NATS servers. For asynchronous communication with NATS, Python relies nats-py library. This library supports an event-driven architecture, enabling the efficient handling of messages through asynchronous tasks.

Finally, RabbitMQ integration for Java is done with the com.rabbitmq:amqp-clien which provides an interface for connecting to RabbitMQ brokers using the AMQP protocol. It supports reliable message routing, sending, and receiving. The pika library serves as a client for RabbitMQ communication using the AMQP protocol. It provides simple message publishing and consumption, enabling smooth interactions with RabbitMQ brokers.

All libraries were tested using their latest supported versions during the experimental phase to ensure compatibility, performance, and access to the most recent features and security patches.

3.2 Testing Criteria

Key performance indicators such as latency, throughput, and scalability are carefully recorded. The study also evaluates the compatibility and integration of each broker's protocols with Java and Python applications, identifying any potential compatibility issues in these mixed-language setups. Moreover, the reliability and fault tolerance of each broker are also tested, particularly their resilience to network disruptions and their ability to recover messages.

3.3 Testing Procedure

The experimental framework is structured to include diverse communication scenarios: – Java-to-Java, Java-to-Python, Python-to-Java, and Python-to-Python – each tested across different number of messages.

Data collection for performance metrics is automated using logging facilities within the applications. This ensures that the records reflect the accurate results of the conducted tests.

3.4 Limitations

The scope of this study is limited to evaluating the default configurations of each broker without any consideration for the potential enhancements to the configurations through custom optimizations. It is also important to acknowledge that the selected hardware and software configurations, while standardized, may still influence the experimental outcomes.

4 Experimental Results and Discussion

This section details the tests conducted on the four communication brokers – Kafka, ActiveMQ Artemis, RabbitMQ, and NATS – and presents the results obtained. The tests were designed to evaluate key performance metrics in different microservice communication scenarios.

The key performance metrics evaluated were:

- *Latency*: the time taken for a message to travel from the producer to the consumer.
- *Throughput*: the number of messages successfully delivered per unit of time.
- *Scalability*: the ability to handle increasing load by adding more resources.
- *Reliability*: consistency in message delivery without loss, even under network disruptions.

The tests were conducted by varying the number of messages and the combination of producer, consumer and broker. The size of the messages was kept unchanged in all situations at 100 KB. Messages were sent in batches of 100, 1,000, 10,000, and 100,000 messages.

4.1 Results

Latency was measured for each broker under various number of messages. Results were represented in heatmaps to better visualize the broker performance, where darker colors indicate slower sending and receiving, while clear colors represent faster exchange of messages.

Both the producer and the consumer were measured. For 100 messages (Table 2), the best results were obtained with the combination of NATS and Python as both consumer and producer.

Table 2. 100 messages

Broker	Pt-Pt	Pt-Jv	Jv-Pt	Jv-Jv
RabbitMQ	0.1	0.11	0.13	0.04
NATS	0.02	0.02	0.09	0.09
Artemis	0.13	0.6	2.1	1.28
Kafka	0.05	0.45	0.5	0.5

(a) Producer Performance

Pt-Pt	Pt-Jv	Jv-Pt	Jv-Jv
0.08	0.1	0.07	0.04
0.03	0.04	0.03	0.04
5.2	1.52	1.8	1.21
0.03	0.1	0.1	0.12

(b) Consumer Performance

For 1000 messages (Table 3), the best results were obtained with the combination of NATS and Python as both consumer and producer, just like in the previous case.

Table 3. 1000 messages

Broker	Pt-Pt	Pt-Jv	Jv-Pt	Jv-Jv
RabbitMQ	0.85	0.9	0.25	0.3
NATS	0.17	0.19	0.22	0.22
Artemis	1.2	1.2	11.87	11.61
Kafka	0.28	0.47	0.99	1

(a) Producer Performance

Pt-Pt	Pt-Jv	Jv-Pt	Jv-Jv
0.85	0.85	0.54	0.29
0.21	0.22	0.22	0.25
7.8	3.21	10.5	11.24
0.29	0.91	0.69	0.81

(b) Consumer Performance

For 10,000 messages (Table 4), the best results were obtained with the combination of RabbitMQ and Java as producer and Python as consumer. NATS continued to show low latency, recording 2.03 s in Python-to-Python, reflecting its suitability for high-throughput applications.

Table 4. 10,000 messages

Broker	Pt-Pt	Pt-Jv	Jv-Pt	Jv-Jv
RabbitMQ	8	8.01	1.44	1.6
NATS	2.03	2.04	1.19	1.25
Artemis	11.49	11.51	98.77	98.24
Kafka	5.02	4.39	6.22	6.1

(a) Producer Performance

Pt-Pt	Pt-Jv	Jv-Pt	Jv-Jv
8	8	5.26	1.67
2.1	2.06	2.31	2.2
14.37	29.6	92.8	97.85
3.65	6.06	5.92	5.76

(b) Consumer Performance

Finally, for 100,000 messages (Table 5), the best results were obtained with NATS and Java as both consumer and producer, close to the Python implementation. RabbitMQ maintained its performance at 14 s for Java-to-Java, suitable for high-load Java applications.

Table 5. 100,000 messages

Broker	Pt-Pt	Pt-Jv	Jv-Pt	Jv-Jv
RabbitMQ	77.38	78.22	66.66	14.66
NATS	21.85	20.03	23.71	19.94
Artemis	116.06	124.97	949.13	949.66
Kafka	42.71	42.73	57.03	55.39

(a) Producer Performance

Pt-Pt	Pt-Jv	Jv-Pt	Jv-Jv
77.37	78.55	55.54	14.69
21.9	20.04	25	20.87
122.56	292	944.69	949.26
41.16	51.79	55.22	54.9

(b) Consumer Performance

The impact of the increase in the number of messages on the brokers is similar in all situations. As the number of messages increases, the time taken to send and receive the total amount also increases (Fig. 2). Only the NATS situation is shown, although the other brokers behave the same way.

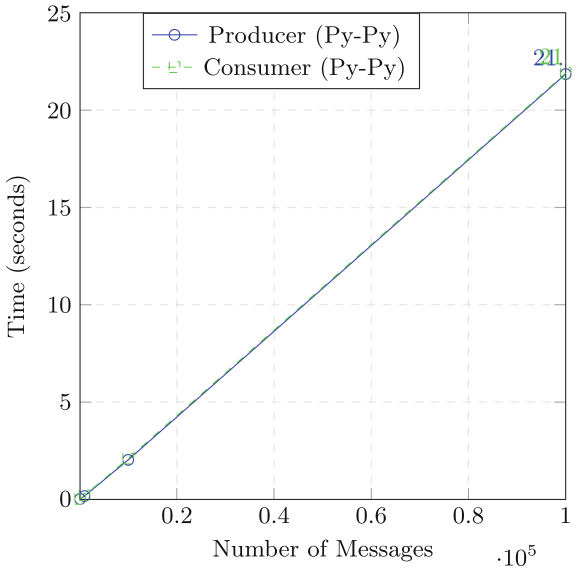


Fig. 2. NATS Producer and Consumer Performance (Python-Python)

Throughput and Latency. RabbitMQ demonstrated strong throughput and low latency in Java-to-Java and Python-to-Python scenarios at lower message volumes, maintaining latencies around 0.04 to 0.1 s for both producers and consumers at 100 messages. However, latency increased significantly with higher message volumes, particularly in cross-language scenarios such as Python-to-Java. By 100,000 messages, Java-Java remained relatively efficient, but Python-Python and cross-language setups showed substantial latency increases for both producers and consumers, highlighting RabbitMQ's limitations in handling large-scale, mixed-language communication efficiently.

NATS consistently showed low latency and high throughput across all scenarios and message volumes, making it the fastest broker overall. For both producers and consumers, latency remained minimal at 100 messages, mostly under 0.1 s for all scenarios, including cross-language communication. Even at 100,000 messages, latency for both producers and consumers stayed under 25 s, proving NATS's ability to scale efficiently while maintaining excellent performance. These results confirm NATS as the optimal choice for latency-sensitive, high-throughput applications.

ActiveMQ Artemis showed significantly higher latency in comparison to the other brokers, even at smaller message volumes. For both producers and consumers, latency across different scenarios started relatively high at 100 messages and increased dramatically as the message volume grew. By 100,000 messages, scenarios with Java producer exhibited extreme latency, with both producers and consumers exceeding several hundred seconds. This pattern suggests Artemis is unsuitable for applications requiring low latency or scalability (Table 5).

Kafka provided consistently low latency and high throughput across all scenarios and message sizes. At 100 messages, both producers and consumers exhibited latencies under 0.2 s for all scenarios and. As message volumes increased, latency increased steadily but remained manageable, with values under 60 s for both producers and consumers at 100,000 messages. These results put Kafka as a reliable broker for high-throughput, large-scale applications, particularly where latency requirements are moderate (Table 5).

In summary, RabbitMQ and Kafka have shown strong performance in language-matched scenarios, with Kafka excelling at higher message volumes. NATS consistently outperformed all the other brokers in both throughput and latency, maintaining exceptional performance across all scenarios and message sizes, proving to be a strong high-performance broker. ActiveMQ Artemis, however, faced significant challenges in maintaining low latency as its results were worse than the other competitors that were tested, limiting its suitability for high-volume or latency-critical applications.

Scalability. RabbitMQ showed moderate scalability overall. In the Python-to-Python scenario, latency increased steadily from 0.1 s at 100 messages to over 77 s at 100,000 messages for both producers and consumers. In the Java-to-Java test, RabbitMQ scaled better, with latency increasing from 0.042 s at 100 messages to just under 15 s at 100,000 messages, proving its suitability for high-load Java applications. However, in mixed-language scenarios such as Python-to-Java and Java-to-Python, latency increased, exceeding 78 s in some cases at 100,000 messages. These results suggest that RabbitMQ is a viable option for language-matched communication in moderate-load environments but may have some struggles and difficulties under heavy, mixed-language traffic.

NATS consistently demonstrated exceptional scalability across all scenarios. In the Python-to-Python scenario, latency grew only slightly from 0.02 s at 100 messages to around 22 s at 100,000 messages. Similarly, the Java-to-Java scenario showed minimal latency growth, increasing from 0.09 s at 100 messages to just under 20 s at 100,000 messages. Cross-language scenarios, like Python-to-Java and Java-to-Python, have also shown similar trends, with latency increasing from less than 0.1 s at 100 messages to around 23 s at 100,000 messages for both producers and consumers. This consistent performance highlights NATS's ability to handle high message volumes efficiently and smoothly, making it the most scalable broker for both language-matched and mixed-language scenarios.

ActiveMQ Artemis faced major challenges with scalability in all the different scenarios. For Python-to-Python communication, latency started at 0.13 s for 100 messages and rose to over 116 s at 100,000 messages. In the Java-to-Java scenario, latency was higher, starting at 1.28 s and reaching nearly 950 s at 100,000 messages. Mixed-language scenarios didn't perform well either, with latency exceeding 940 s for both producer and consumer at high message volumes in the Java-to-Python scenario. These results indicate that Artemis is not well suited for high-load or latency-sensitive applications, as it heavily struggles to scale effectively under increasing traffic (Table 5).

Kafka displayed strong scalability, in the Java-to-Java test: latency increased from 0.5 s at 100 messages to around 55 s at 100,000 messages. In the Python-to-Python scenario, latency was even lower, starting at 0.048 s and growing to about 43 s at 100,000 messages. Mixed-language scenarios, such as Python-to-Java and Java-to-Python, showed decent scalability as well, with similar results for both producers and consumers. These results prove Kafka's ability to handle high-throughput applications efficiently without any major issues while maintaining strong scalability across diverse communication setups.

In summary, while RabbitMQ and Kafka have shown reasonable and manageable scalability in high-load scenarios, NATS consistently outperformed them in this aspect, maintaining minimal latency growth across all message volumes and different scenarios. In contrast, ActiveMQ Artemis struggled heavily, making it unsuitable for highly scalable systems or latency-critical applications.

Reliability. RabbitMQ demonstrated moderate reliability in default settings, showing consistent message delivery across all the test scenarios under normal conditions but experiencing occasional message loss during network disruptions.

NATS had problems when it comes to reliability in its default setup due to the lack of persistence, with messages in transit being lost during network disruptions.

ActiveMQ Artemis delivered high reliability by default due to persistent messaging, retaining and delivering messages despite multiple network disruptions.

Kafka showed great reliability in default configuration as well, with no message loss during network issues, proving to be a robust and reliable broker.

4.2 Discussion

The experimental results highlight multiple major performance differences across brokers, mainly in terms of latency, throughput, scalability, and reliability.

RabbitMQ performed well in Python-to-Python communication, maintaining low latency and high throughput under moderate loads (Table 5). However, its latency saw a major increase in cross-language scenarios like Python-to-Java and Java-to-Python as message volumes grew. While highly scalable for Java-to-Java communication, occasional message loss during network disruptions could be an issue for fault-tolerant applications.

NATS excelled in both latency and throughput across all tested scenarios, including cross-language setups. It maintained low latency at high message volumes, as shown in Table 5, making it highly appropriate for latency-sensitive applications. Its outstanding scalability guaranteed minimal performance degradation under high loads. However, its default lack of persistence caused message loss during network disruptions, requiring additional configuration to improve its reliability for critical applications.

ActiveMQ Artemis showed higher latency and notable performance degradation under high message volumes. Its high latency and scalability problems make it unsuitable for low-latency or high-load environments. However, its persistent

messaging ensures high reliability, as it allows it to deliver messages even after multiple network disruptions.

Kafka provided strong performance overall, maintaining low latency and high throughput across all scenarios (Table 5). Its scalability was reasonable, handling high loads efficiently with manageable latency growth. Kafka's reliability is dependable in default configuration as well, making it a robust and viable option for high-throughput, language-agnostic microservices.

In conclusion, each broker's suitability depends on the application requirements. NATS is optimal for low-latency, high-throughput scenarios but requires configuration for critical reliability. RabbitMQ and Kafka perform well in language-matched setups, with Kafka excelling under high loads. Artemis is best suited for applications where fault tolerance and message persistence are critical.

5 Conclusion

This study compared Apache Kafka, ActiveMQ Artemis, RabbitMQ, and NATS as communication systems for microservices architectures, evaluating latency, throughput, scalability, and reliability across language-matched and cross-language setups.

RabbitMQ performed well under moderate loads in language-matched scenarios but showed increased latency and scalability issues at higher volumes, with occasional message loss during disruptions. NATS excelled in latency, throughput, and scalability but requires additional configuration for reliability in order to deal with network disruptions. ActiveMQ Artemis prioritized reliability with persistent messaging but showed high latency and performance decrease under heavy loads. Kafka demonstrated strong scalability, low latency, and robust reliability, making it ideal for high-throughput microservices.

In summary, RabbitMQ and Kafka are effective for language-matched setups, with Kafka excelling in scalability. NATS offers the best performance but needs adjustments for critical reliability, while Artemis is best suited for fault-tolerant systems. Future work should explore advanced configurations, hybrid models, and integration with different technologies.

Acknowledgment. This work was partially supported by the HORIZON-CL4-2021-TWIN-TRANSITION-01 openZDM project under Grant Agreement No. 101058673. The authors are grateful to the Foundation for Science and Technology (FCT, Portugal) for support through FCT/MCTES (PIDDAC): CeDRI, UIDB/05757/2020 (DOI: 10.54499/UIDB/05757/2020) and UIDP/05757/2020 (DOI: 10.54499/UIDP/05757/2020); and SusTEC, LA/P/0007/2020 (DOI: 10.54499/LA/P/0007/2020).

References

1. NATS.io – Cloud Native, Open Source, High-performance Messaging. https://nats.io/
2. Ataei, M., Eghmazi, A., Shakerian, A., Landry, Jr., R., Chevrette, G.: Publish/subscribe method for real-time data processing in massive IoT leveraging blockchain for secured storage. Sensors **23**(24) (2023). https://doi.org/10.3390/s23249692
3. Batista, C., Morais, F., Cavalcante, E., Batista, T., Proença, B., Rodrigues Cavalcante, W.B.: Managing asynchronous workloads in a multi-tenant microservice enterprise environment. Softw. Pract. Experience **54**(2), 334–359 (2024). https://doi.org/10.1002/spe.3278. https://onlinelibrary.wiley.com/doi/10.1002/spe.3278
4. Bhandari, D.: NATS: a Kafka alternative for messaging. NashTech Insights (2024). https://blog.nashtechglobal.com/nats-a-kafka-alternative-for-messaging/
5. Chy, M.S.H., Arju, M.A.R., Tella, S.M., Cerny, T.: Comparative evaluation of Java virtual machine-based message queue services: a study on Kafka, Artemis, Pulsar, and RocketMQ. Electronics **12**(23), 4792 (2023). https://doi.org/10.3390/electronics12234792
6. Cugola, G., Margara, A.: Processing flows of information: from data stream to complex event processing. ACM Comput. Surv. **44**(3) (2012). https://doi.org/10.1145/2187671.2187677
7. Dobbelaere, P., Esmaili, K.S.: Kafka versus RabbitMQ: a comparative study of two industry reference publish/subscribe implementations: industry paper. In: Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems, pp. 227–238 (2017)
8. Fu, G., Zhang, Y., Yu, G.: A fair comparison of message queuing systems. IEEE Access **9**, 421–432 (2021). https://doi.org/10.1109/ACCESS.2020.3046503. https://ieeexplore.ieee.org/document/9303425/
9. Hiraman, B.R.: A study of apache Kafka in big data stream processing. In: 2018 International Conference on Information, Communication, Engineering and Technology (ICICET), pp. 1–6. IEEE (2018)
10. Langhi, S., Tommasini, R., Della Valle, E.: Extending Kafka streams for complex event recognition. In: 2020 IEEE International Conference on Big Data (Big Data), pp. 3029–3036. IEEE (2020)
11. Le Noac'h, P., Costan, A., Bougé, L.: A performance evaluation of Apache Kafka in support of big data streaming applications. In: 2017 IEEE International Conference on Big Data (Big Data), pp. 4803–4806. IEEE (2017)
12. Magnoni, L.: Modern messaging for distributed sytems. J. Phys. Conf. Ser. **608**(1), 012038 (2015). https://doi.org/10.1088/1742-6596/608/1/012038
13. Maharjan, R., Chy, M., Arju, M., Cerny, T.: Benchmarking message queues. Telecom **4**(2), 298–312 (2023). https://doi.org/10.3390/telecom4020018
14. Sharvari, T., Sowmya Nag, K.: A study on modern messaging systems- Kafka, RabbitMQ and NATS streaming, December 2019. https://doi.org/10.48550/arXiv.1912.03715. http://arxiv.org/abs/1912.03715 [cs]
15. Wu, H., Shang, Z., Wolter, K.: A reactive batching strategy of Apache Kafka for reliable stream processing in real-time. In: 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE), pp. 294–305. IEEE (2020)