IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 3, March 2026
Impact Factor: 8.2

# Application of Message Brokers (RabbitMQ, Kafka): Performance Analysis and Use Cases

**Andrey Berezhnoy**
Bachelor's Degree
Peter the Great St. Petersburg Polytechnic University, St. Petersburg, Russia

**Abstract:** *This article analyzes the capabilities of modern message brokers in the context of event-driven and scalable architectures of distributed systems. It examines the theoretical foundations of message-exchange systems, including pub/sub models, queues, and stream processing, as well as their role in ensuring reliability and flexibility of inter-service communication. Particular attention is given to the comparative characteristics of two widely used brokers: RabbitMQ and Apache Kafka. It analyzes their architectural principles, message delivery and storage mechanisms, scalability features, fault tolerance, and performance metrics. The applicability of these tools under different workload types and architectural requirements is also investigated.*

**Keywords**: Message Brokers, RabbitMQ, Kafka, Distributed Systems, Messaging, Performance, Scalability

## I. INTRODUCTION

Modern distributed information systems are characterized by high dynamism, scalability, and the need to process large volumes of data in real time. As the number of microservice-based architectures, IoT devices, streaming analytics platforms, and event-driven solutions continues to grow, the demand for efficient mechanisms of message exchange between system components becomes increasingly critical. Message brokers have become a key element of such architectures, providing reliable data exchange, flexible routing capabilities, and isolation of interacting subsystems.

In numerous available solutions, RabbitMQ and Apache Kafka occupy one of the leading positions in terms of popularity as message brokers with fundamentally different architectures and use cases. The development of high-loaded services and increasing complexity of interaction models imply a detailed understanding of their pros and cons, as well as their characteristics in terms of performance. Even for such accepted solutions as RabbitMQ and Apache Kafka, it is not always easy to make the correct selection in terms of engineering considerations. The aim of this study is to conduct a comparative analysis of RabbitMQ and Apache Kafka, evaluate their performance, and assess their applicability under various architectural and workload conditions.

## II. MAIN PART. THEORETICAL FOUNDATIONS AND ARCHITECTURAL PRINCIPLES OF MESSAGE BROKERS

Modern distributed software systems are becoming increasingly complex. Under these conditions, message brokers serve as a key architectural component, enabling asynchronous communication between services and reducing the coupling of system components.

A message broker is a piece of middleware that facilitates data exchange between distributed systems by implementing a publish/subscribe (pub/sub) or message-queuing pattern. In a pub/sub model, the broker routes messages to subscribers according to their declared interests after publishers send messages to it without naming specific recipients (fig. 1).


DOI: 10.48175/IJARSCT-31621

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 3, March 2026
Impact Factor: 8.2

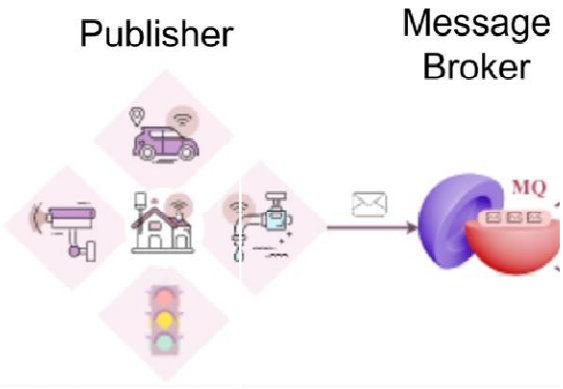


Fig. 1. The Publish/Subscribe paradigm [1]

This decouples senders from receivers: neither side needs knowledge of the other's identity or state. The broker further provides an intermediary layer that manages, filters, and routes messages between clients. It secures that data is delivered accurately, efficiently, and in a timely fashion. More importantly, message queues within brokers can temporarily buffer messages, protecting systems from overloads or downstream outages.

By requiring acknowledgments and persisting messages, message brokers frequently ensure dependable delivery. Because mechanisms like durable queues and acknowledgments guarantee that messages are not lost even if customers disconnect or temporarily fail, broker-based systems are particularly reliable. To put it another way, brokers have the ability to store messages in memory or on disk until customers successfully acknowledge them. They also commonly support multiple communication protocols, multiple data formats, and security features like authentication and encryption [2].

In terms of architecture, brokers may be organized in either a cluster configuration or distributed configuration to provide scalability and high availability. Conventional message brokering systems come with a central queue where all messages to be delivered wait. In such systems, typically, all messages would be delivered from the message broker to their consumers in a 'push mode.' In log-based systems, there would be a commit log architecture in which all messages are appended to their designated partitions across nodes of message brokers.

Overall, the theoretical basis of message brokers hinges on the pub/sub paradigm, which decouples senders and receivers and enables scalable, asynchronous communication. Brokers centralize control and monitoring of message flows, providing a reliable, queue-based substrate that can persist messages and protect against data loss. This architecture has evolved over decades from simple Message-Oriented Middleware (MOM) to modern streaming platforms, but the core principle remains: act as an intermediary to ensure messages reach their destinations according to system rules (ordering, routing, delivery guarantees, etc.) [3].

Thus, the theoretical foundations of message exchange form the basis for building event-driven and scalable distributed systems. The examined pub/sub, queue-based, and stream-processing models demonstrate how message brokers provide reliability, resilience, and flexibility in interactions between system components.

## III. COMPARATIVE ANALYSIS OF RABBITMQ AND APACHE KAFKA

RabbitMQ and Apache Kafka are two sought-after open-source message broker systems with different design philosophies. RabbitMQ was introduced as an Erlang-based broker that is compatible with the Advanced Message Queuing Protocol (AMQP). Clients send messages to exchanges using this all-purpose queueing system.

Many protocols and features, such as broker clustering and fault-tolerant mirroring, are supported by RabbitMQ. It is helpful to describe how a message passes through the broker before it reaches a customer in order to shed light on RabbitMQ's internal structure and the functions of its essential parts (fig. 2).

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 3, March 2026
Impact Factor: 8.2

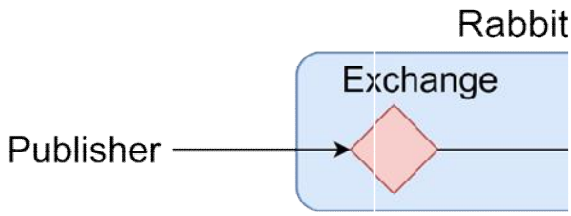


Fig. 2. Working principle of RabbitMQ intermediary [4]

RabbitMQ takes on a push-based delivery model by default. In this case, the broker actively pushes messages to customers when they're ready, consumers can explicitly acknowledge messages to remove them from the queue, and messages can be marked persistent to withstand a broker restart. RabbitMQ is adaptable to complex topologies because of its queueing model and exchange routing, which allow for complex patterns (direct, fanout, topic routing, etc.).

LinkedIn created Kafka in 2011 as a distributed streaming platform with high throughput. Data is divided into topics by Kafka and distributed among a group of broker nodes. The write-ahead log data can be reclaimed differently by the consumers, while the producers append their messages to the end of each partition. It provides best-notch tools and services for data processing and integration, which includes Kafka Connect, Kafka Streams. Additionally, this also provides the ability to develop the libraries of the clients using Java, Python, Go. The disk I/O in order and replication represent large components of the Kafka system because each message can be committed to the partition and has to be replicated afterwards to the followers. Kafka concentrates on high-throughput publishing/subscriptions; it provides its own binary protocol instead of RabbitMQ's multiple ones.

These architectural differences yield distinct performance trade-offs. Kafka's log-centric design allows linear scalability: adding brokers and partitions increases throughput with minimal coordination overhead.

Kafka maintained roughly 45,000 messages/sec for 1 MB payloads (in a secure test), which was about 41% higher than RabbitMQ under the same conditions, according to a recent comparative study on message broker performance. RabbitMQ dominated small-payload throughput because it was designed for smaller message routing. It outperformed Kafka in that regime, achieving over 35,000 messages/sec for 10-byte messages.

The same study defined four security levels, each introducing additional protection mechanisms: Level 0 – no encryption or authentication; Level 1 (Transport Layer Security, TLS) – traffic encryption enabled; Level 2 (TLS+Auth) – client and broker authentication added; Level 3 (TLS+Auth+ACL) – a full security configuration applied, including Access Control Lists (ACL) (table 1).

TABLE I: Throughput Performance Comparison Across Security Levels (Events/Second) [5]

| Message Size | Broker | Level 0 | Level 1 (TLS) | Level 2 (TLS+Auth) | Level 3 (TLS+Auth+ACL) |
|---|---|---|---|---|---|
| 10 B | RabbitMQ | 35,420 | 31,200 (12% ↓) | 24,500 (31% ↓) | 14,200 (60% ↓) |
| | Kafka | 28,300 | 25,100 (11% ↓) | 19,800 (30% ↓) | 11,300 (60% ↓) |
| 1 MB | RabbitMQ | 32,400 | 28,500 (12% ↓) | 22,600 (30% ↓) | 13,000 (60% ↓) |
| | Kafka | 45,600 | 40,200 (12% ↓) | 31,900 (30% ↓) | 18,200 (60% ↓) |

The behavior of two brokers when processing different volumes of messages can be compared thanks to quantitative data collected in a study looking at message broker performance under various workload conditions. For the producers and consumers of both brokers at three load levels the table displays the measured processing times and throughput values (messages per second) (table 2).


DOI: 10.48175/IJARSCT-31621

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 3, March 2026
Impact Factor: 8.2

TABLE II: PERFORMANCE COMPARISON OF APACHE KAFKA AND RABBITMQ [6]

| Messages processed | System component | Time (s) | Throughput (msg/s) |
|---|---|---|---|
| 100 | Kafka Producer | 0.16 | 644.87 |
| | Kafka Consumer | 2.21 | 45.28 |
| | RabbitMQ Producer | 0.16 | 644.83 |
| | RabbitMQ Consumer | 0.06 | 1614.62 |
| 1000 | Kafka Producer | 1.40 | 714.21 |
| | Kafka Consumer | 0.47 | 2137.99 |
| | RabbitMQ Producer | 1.49 | 671.55 |
| | RabbitMQ Consumer | 0.46 | 2161.49 |
| 10,000 | Kafka Producer | 14.02 | 713.25 |
| | Kafka Consumer | 5.27 | 1895.80 |
| | RabbitMQ Producer | 15.97 | 626.34 |
| | RabbitMQ Consumer | 4.61 | 2169.57 |

Beyond raw speed, the two systems differ in functionality. RabbitMQ's AMQP foundations give it rich routing semantics. It can route messages through multiple exchange types and apply patterns like topic filtering and header matching. RabbitMQ typically deletes messages once acknowledged (though durability options exist). In terms of delivery guarantees, both default to "at-least-once" semantics: RabbitMQ re-delivers unacknowledged messages, and Kafka commits offsets to track consumption. Kafka can additionally provide exactly-once semantics via transactional writes (since v0.11), a feature RabbitMQ lacks natively.

Finally, language and platform support differ: RabbitMQ is implemented in Erlang (noted to some as "unfamiliar" to many developers) and offers client libraries for many languages. Kafka is Java/Scala-based, running on the JVM, and is commonly deployed in Java-centric environments [7].

Both of these technological tools allow clustering, but the methods are different: RabbitMQ replicates state using mirrored or quorum queues, whereas Kafka's cluster is based on partition leaders and followers. These architectural decisions further can affect scaling behavior and fault-tolerance.

## IV. PRACTICAL APPLICABILITY FOR DIFFERENT LOADS AND ARCHITECTURES

The choice between RabbitMQ and Kafka depends on workload characteristics and architectural needs. In general, RabbitMQ is well-suited for traditional messaging tasks and microservice communication patterns. It is ideal when applications require complex routing or guarantee very low latency per message.

RabbitMQ supports work-queue patterns where multiple consumers pull tasks from a queue. Indeed, RabbitMQ has been recommended (alongside Pulsar) for microservices workloads because it offers superior routing flexibility with excellent latency and throughput performance (for small messages) [8]. Common use cases include RPC/message acknowledgments, event notification between services, or any scenario where message order is confined to a single queue and latency must be minimized. Its support for multiple protocols also means it can integrate well with diverse clients and IoT devices that may not natively speak Kafka's protocol.

In contrast, Kafka is tailored for high-throughput data streaming and event-driven architectures. It is frequently used in analytics, logging, and big data pipelines where very large volumes of events need to be ingested and processed. Kafka's ability to retain a long history of messages (the log) means it can serve as a record of events – multiple independent consumers or consumer groups can rewind and replay data at will [9]. Because of this, Kafka is a good choice for systems that need pub/sub semantics with high fan-out (many independent subscribers), real-time data integration, and change-data capture. All orders may be published to a Kafka topic by an e-commerce platform; downstream systems, such as inventory, billing, and analytics, consume the entire stream from the beginning without

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 3, March 2026
Impact Factor: 8.2

interfering with one another. Similar to this, Kafka can ingest sensor data at scale in a monitoring or telemetry system and feed it into machine learning models or real-time dashboards.

In summary, small-scale and flexible messaging with sophisticated routing and low latency calls for RabbitMQ, forasmuch as large-scale event streaming and high-throughput data pipelines leverage Kafka's distributed log architecture. Performance benchmarks and case studies reinforce this: RabbitMQ dominates in workloads with many tiny messages, while Kafka demonstrated better scalability on large-message streams.

## V. CONCLUSION

RabbitMQ and Apache Kafka are both favoured and general message brokers, but they differ greatly in terms of architecture, data processing principles and performance characteristics. RabbitMQ's flexible routing mechanisms, low delivery latency, and broad protocol support make it a valuable tool for task queues, microservice-based systems, and real-time service-to-service communication. On the other hand, Kafka's high throughput, reliable processing of enormous volumes of events, and long-term data stream storage give it an advantage in streaming analytics, logging, telemetry, and event-sourcing systems.

Specific architectural requirements, workload characteristics, and integration constraints within a distributed system should all be taken into consideration when selecting a message broker. Both solutions exhibit high efficiency within their respective domains, but neither is universal. A well-founded decision between RabbitMQ and Kafka is made possible by a thorough evaluation of performance, scalability, and fault-tolerance factors, guaranteeing the best possible operation of contemporary event-driven applications and high-load computational platforms.

## REFERENCES

[1] A. Saleh, S. Pirttikangas, L. Lovén. (2023) Pub/Sub message brokers for GenAI. [Online]. Available: https://arxiv.org/html/2312.14647v1

[2] R. Garifullin, "Zero Trust models in web development," The Eurasian Union of Scientists. Series: technical and physico-mathematical sciences, vol. 1, no. 2, suppl. 127, pp. 22-25. 2025.

[3] I. Gamal, H. Abdel-Galil, A. Ghalwash, "Osmotic message-oriented middleware for Internet of Things," Computers, vol. 11, no. 4, suppl. 56. 2022.

[4] A. Ćatović, N. Buzađija, S. Lemes, "Microservice development using RabbitMQ message broker," Science, engineering and technology, vol. 2, no. 1, pp. 30-37. 2022.

[5] S.J. Putra, G. Firmansyah, B. Tjahjono, H. Akbar, "Comprehensive Benchmarking of Message Brokers: Evaluating Performance and Security Metrics for Reliable Messaging Systems," Jurnal Locus Penelitian dan Pengabdian, vol. 4, no. 11, pp. 10829-10838.

[6] M. Pacella, A. Papa, G. Papadia, E. Fedeli, "A scalable framework for sensor data ingestion and real-time processing in cloud manufacturing", Algorithms, vol. 18, no. 1, suppl. 22. 2025.

[7] A. Topalidi, "Strategies for improving the performance of Ruby web applications: data caching and SQL query optimization," The scientific heritage, vol. 162, pp. 74-78. 2025.

[8] R. Bux, G.S. Shenoy, "Performance analysis of RESTFUL web services and RABBITMQ for microservices based systems on cloud environment," In2024 3rd International Conference for Innovation in Technology (INOCON), pp. 1-6. 2024.

[9] S. Vyas, R.K. Tyagi, C. Jain, S. Sahu, "Literature review: A comparative study of real time streaming technologies and apache kafka," In2021 Fourth International Conference on Computational Intelligence and Communication Technologies (CCICT), pp. 146-153. 2021.