JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 37 (2025) 457–462
Received: 24 June 2025
Accepted: 20 September 2025

Comparative Performance Analysis of RabbitMQ and Kafka Message Queue Systems in Spring Boot and ASP.NET Environments

Analiza porównawcza wydajności systemów kolejkowych RabbitMQ i Kafka w środowiskach Spring Boot i ASP.NET

Filip Kamiński*, Radosław Kłonica, Beata Pańczyk

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract
The article analyzes and compares the performance of Kafka 4.0 and RabbitMQ 4.1 in applications built with Spring (Kotlin) and .NET. Given the growing importance of microservices and event-driven architectures, the research examines message throughput, resource consumption, and stability under different loads. Two applications were developed to measure performance in terms of processing speed, CPU, and memory usage. The study also explores architectural considerations and factors affecting performance. The findings offer insights into when each system is most suitable, helping developers make informed decisions based on project requirements. The results show that Kafka performs better in .NET environments with up to 38% higher throughput and 40% lower latency while RabbitMQ is more efficient in Spring Boot setups, using nearly 29% less memory and delivering responses 25% faster.

Keywords: Apache Kafka; RabbitMQ; .NET; Spring Boot

Streszczenie
Artykuł naukowy analizuje i porównuje wydajność systemów kolejkowych Kafka i RabbitMQ w aplikacjach Spring Boot (Kotlin) oraz .NET (C#). Celem jest ocena szybkości przetwarzania wiadomości, zużycia zasobów i stabilności w różnych scenariuszach obciążenia. Badania przeprowadzono za pomocą dwóch aplikacji, każda korzystała z obu systemów kolejkowych. Praca uwzględnia również aspekty architektoniczne i czynniki wpływające na wydajność. Wyniki dostarczają rekomendacji dotyczących wyboru odpowiedniego narzędzia w zależności od środowiska I wymagań projektu. Wyniki pokazują, że Kafka działa lepiej w środowiskach .NET, podczas gdy RabbitMQ jest bardziej efektywny w środowiskach Spring Boot o ograniczonych zasobach. Wyniki pokazują, że Kafka działa lepiej w środowisku .NET osiągając do 38% wyższą przepustowość i 40% niższe opóźnienie, natomiast RabbitMQ jest bardziej efektywny w środowisku Spring Boot, zużywając prawie 29% mniej pamięci i dostarczając odpowiedzi o 25% szybciej.

Słowa kluczowe: Apache Kafka; RabbitMQ; .NET; Spring Boot

*Corresponding author
Email address: s95249@pollub.edu.pl (F. Kamiński)

1. Introduction

In the era of rapid technological development and the growing popularity of distributed and microservices architectures, efficient information exchange between system components is crucial. Message queue systems, enabling asynchronous communication while ensuring consistency, scalability, and flexibility, play a key role in such architectures. Kafka [1] and RabbitMQ [2], two of the most popular message queue systems, differ in architecture and adaptability to various application requirements, making them interesting subjects for performance analysis and comparison across different environments.

The primary goal of this article is to evaluate the performance of Kafka and RabbitMQ using two applications one developed in Spring with Kotlin [3] and the other in .NET [4]. The study analyzes message processing speed, system resource efficiency, and stability under various load scenarios. Selecting the right message queue system often challenges developers and engineers to align the tool with specific project and operational requirements. This research seeks to determine which system performs better under certain conditions and provides valuable insights for future projects utilizing message queue architectures.

Through detailed comparative analysis and evaluation of experimental results, the article offers a comprehensive overview of Kafka and RabbitMQ performance across different technological environments and configurations. It also provides recommendations for their practical application in production systems. Through detailed comparative analysis and evaluation of experimental results, the article offers a comprehensive overview of Kafka and RabbitMQ performance across different technological environments and configurations. It also provides recommendations for their practical application in production systems. The originality of this study lies in the parallel analysis of both message brokers across two distinct platforms, offering unique cross-environment insights not commonly addressed in existing literature.

2. Related works

Henning and Hasselbring (2024) [5] conducted an extensive benchmarking study focused on the scalability of modern stream processing frameworks deployed in

microservices architectures in the cloud. Their research, spanning over 740 hours of experiments, evaluated five popular frameworks - Apache Flink, Kafka Streams, Apache Samza, Hazelcast Jet, and Apache Beam - by analyzing their efficiency in processing up to one million messages per second in cloud environments. The findings confirmed that all tested frameworks exhibited linear scalability; however, their resource consumption varied significantly under increased loads. This article provides valuable context for our comparative analysis of RabbitMQ and Apache Kafka, especially highlighting Kafka's advantages as a foundational element in scalable microservices and high-throughput streaming systems.

Lercher et al. (2024) [6] focused on real-world strategies and challenges related to API evolution in microservices-based architectures. Based on 17 interviews with practitioners from 11 companies, the authors identified six major strategies and six key challenges associated with evolving REST APIs and event-driven communication through message brokers such as RabbitMQ and Kafka. Strategies included backward compatibility maintenance, API versioning, and tight inter-team collaboration. Key challenges were difficulties in impact analysis, inefficient team communication, and client dependency on outdated API versions. The article emphasizes the importance of automated impact analysis and communication efficiency as directions for future research. In the context of our study, it confirms the growing significance of message brokers in modern software architectures and their influence on the design and evolution of APIs.

Pathak and Kalaiarasan (2021) [7] presented an in-depth analysis of RabbitMQ's queuing mechanisms in publish-subscribe models, with particular emphasis on applications in the Internet of Things (IoT). The paper discusses RabbitMQ's strengths in scalability, reliability, and availability in distributed systems. It explores various communication models (request-response, push-pull, exclusive pair) and internal architecture components such as exchanges and queues. Special attention is given to queue overload issues, message loss, time-to-live (TTL) mechanisms, and the impact of message size and consumer count on system performance. Experimental results showed that while RabbitMQ is effective for message communication, high-load environments may require additional optimizations such as sub-exchanges to improve throughput and reduce congestion. This study contributes critical insights into the internal behavior of RabbitMQ and its performance under varying queuing configurations.

The reviewed studies highlight the increasing importance of message queue systems such as Apache Kafka and RabbitMQ in the context of modern, distributed, and event-driven architectures. They collectively emphasize the need for scalability, resilience, and efficient API evolution when designing microservice-based systems. Kafka emerges as a highly scalable and resource-efficient platform suitable for high-throughput scenarios, particularly in data-intensive cloud environments. RabbitMQ, in contrast, offers strong reliability and flexibility, especially in IoT and real-time communication scenarios where control over delivery and queuing mechanisms is essential. Moreover, the integration of both brokers with enterprise applications requires careful consideration of architectural patterns, system load characteristics, and developer tooling. These insights reinforce the relevance of conducting performance comparisons in diverse runtime environments, such as Spring Boot and ASP.NET, to inform practical design decisions.

3. Description of the tested applications

This section presents the implementation of the analyzed system using two popular technology stacks. The first is C# with the ASP.NET framework, which is designed for building web applications and services. The second is Kotlin with Spring Boot, which offers a secure and efficient environment for the Java Virtual Machine (JVM).

3.1. RabbitMQ in ASP.NET

In the ASP.NET implementation of RabbitMQ 4.1, the system consists of several key components. Controllers manage incoming HTTP requests, with examples including *AuctionsController.cs*, *CityController.cs*, and *ImageController.cs*. The Data layer defines the database context using Entity Framework Core, while Dtos (Data Transfer Objects) simplify data transfer between layers. Repositories and Interfaces handle data access, enabling dependency injection for greater modularity. Services, such as *AuctionProducerService.cs* for publishing messages and *AuctionConsumerService.cs* for consuming them, are responsible for RabbitMQ 4.1 integration. The system's workflow begins when a controller receives a request and invokes the *AuctionProducerService*, which publishes serialized auction data to RabbitMQ. Simultaneously, the *AuctionConsumerService* listens for incoming messages, processes them such as saving data to the database and manages acknowledgments to ensure message reliability [7]. This architecture supports asynchronous, scalable, and resilient communication.

3.2. Apache Kafka in ASP.NET

The Kafka 4.0 implementation in ASP.NET follows a similar structure but is divided into two primary components: *Producer*, which publishes auction-related messages, and *Consumer*, which processes inventory updates. Controllers manage auction and inventory operations, while the data layer handles database contexts. *Dtos*, *Repositories*, and *Interfaces* organize data flow and access logic. The services layer includes *ProducerService.cs* for publishing and *ConsumerService.cs* for consuming Kafka messages. The system flow starts when auctions are published to Kafka topics via the *ProducerService*. The *ConsumerService* subscribes to these topics, deserializes incoming messages, and processes the data accordingly. Kafka's architecture ensures asynchronous, scalable, and reliable communication across the system.

3.3. RabbitMQ in Spring Boot

In the Spring Boot implementation of RabbitMQ 4.1, the system architecture is based on a clear separation of concerns across application layers**.** REST controllers, such as *AuctionController*, *CityController*, and *ImageController*, handle incoming HTTP requests and delegate business logic to corresponding facades. The domain layer uses models and service classes responsible for data processing and RabbitMQ communication. Asynchronous messaging relies on two main components: *AuctionProducerService*, which publishes messages to the RabbitMQ queue, and *AuctionConsumerService*, which listens for incoming messages, interprets them, and performs operations such as saving data to a MySQL database. Data transfer between layers is handled via Data Transfer Objects (DTOs), which help simplify and organize the structure of the transmitted information. The implementation leverages Spring annotations such as *@Service*, *@Async*, and *@RestController*, allowing the system to remain modular, scalable, and resilient to communication failures.

3.4. Apache Kafka in Spring Boot

In the Spring Boot implementation utilizing Apache Kafka 4.0, the architecture is organized around modular components that ensure scalability and asynchronous data flow. Controllers such as *AuctionController*, *CityController*, and *ImageController* serve as entry points for HTTP requests and delegate logic to domain-level facades. Kafka integration is managed through services like *ProducerService*, which serializes and sends messages to designated Kafka topics, and *ConsumerService*, which subscribes to those topics and processes incoming messages often resulting in operations such as persisting data to a MySQL database [8]. Data is encapsulated using DTOs to maintain clarity and separation between internal logic and external interfaces. Spring's support for Kafka via annotations like *@KafkaListener* simplifies consumer configuration and promotes clean message handling. This approach allows the application to operate reliably in distributed environments, supporting event-driven communication with high throughput and resilience.

3.5. MySQL database

The database was implemented in MySQL 8.0 to support the online auction platform used in the experiments and follows a relational model. It consists of four main tables: *Auctions*, *Images*, *Categories*, and *Cities*. The *Auctions* table stores information about individual auction listings, including name, description, price, expiration date, product condition, contact phone number, and current status, with each auction linked to a specific category and city, and optionally associated with a thumbnail image. The *Images* table contains binary data for auction related images along with metadata such as type and file size, each tied to a single auction. The *Categories* table defines item types available for auction, while the Cities table stores location data, enabling auctions to be filtered or grouped geographically. Relationships between tables are enforced via foreign keys to maintain data integrity. In the context of performance testing, database interactions were limited to operations directly triggered by message broker consumers. These included INSERT operations when saving new auctions or images received via message queues, and SELECT queries when retrieving auction, city, or category details in response to API requests. Write-heavy tests (e.g., binary image uploads) primarily measured the time to persist records in the *Auctions* and *Images* tables, while read-heavy tests measured retrieval times from *Auctions*, *Cities*, and *Categories*. No complex joins or additional business logic outside these core operations were executed during benchmarking, ensuring that measured times reflected message broker integration and I/O performance rather than application side processing.

4. Methods and conduct of research

The research was conducted according to the research scenario described below, as well as executed in the test environment and runtime environment described below.

4.1. Research hypotheses

This article investigates and compares two widely used message queue systems Apache Kafka and RabbitMQ in the context of modern distributed applications. The goal is to evaluate their performance and integration capabilities across different technological platforms, namely Spring Boot and ASP.NET, under varying message throughput conditions. Based on a literature review and preliminary analysis, the following research hypotheses were formulated and tested through controlled experiments:

- Kafka achieves higher throughput than RabbitMQ in high-volume data processing scenarios, regardless of the platform.
- RabbitMQ provides lower latency for individual message delivery, particularly under light load.
- In Spring Boot environments, RabbitMQ integrates more efficiently in terms of configuration and runtime stability.
- In ASP.NET environments, Kafka performs better in stream processing due to stronger support for data-driven architectures.

A series of benchmark tests was conducted to validate these hypotheses under different system loads. The results provide insights into the optimal use of Kafka and RabbitMQ depending on the application's characteristics and platform, supporting more informed architectural decisions.

4.2. Research scenario and procedure

The aim of this study is to evaluate the performance of Kafka and RabbitMQ under load [9], focusing on query execution times, resource usage, and overall system performance. Performance testing was conducted using Apache JMeter 5.6.3 [10], which simulated concurrent HTTP/HTTPS requests to the tested applications. Each application implemented both producer and consumer

logic for the respective broker, ensuring that messages were published to the broker, consumed by the application, and then persisted in a MySQL database. Three categories of scenarios were defined, each corresponding to a different use pattern:

- Binary Data Upload Scenarios (H1-1B, H1-1KB, H1-10KB): Measured the time required to process binary image uploads of sizes 1 byte, 1 kilobyte, and 10 kilobytes via POST requests. Each scenario ran with 500 concurrent threads for 60 seconds, simulating heavy upload traffic. Metrics captured included the time from request submission to the completion of database write operations in the *Images* and *Auctions* tables, throughput in requests per second, and the error rate.
- Low-Traffic Retrieval Scenarios (H2-Low, H2-Mid, H2-High): simulated typical application use with 5 concurrent threads issuing GET requests to retrieve data for a single city, multiple cities, or multiple auctions. The time measurement began at the moment the HTTP request reached the application and ended when the corresponding database SELECT query returned data to the client. These tests provided insight into broker latency under minimal load.
- High-Stress Mixed Scenarios (H3, H4): Combined POST, GET, and PUT requests under extreme load conditions with 5,000 concurrent threads over a 90-second test window. The requests included binary uploads, auction updates, and retrieval operations. The measured execution time covered the complete cycle: API request receipt, message publication, broker delivery, message consumption, database write or read completion.

To guarantee the reliability and comparability of results, each test scenario was conducted in a distinct, dedicated Docker environment version 28.3.3 [11]. Each scenario was executed three times for each broker framework combination (.NET and RabbitMQ, .NET and Kafka, Spring Boot and RabbitMQ, Spring Boot and Kafka). The mean values from the three runs were used in the final analysis to minimize the influence of transient system fluctuations. For each run, JMeter's aggregate report was exported, containing average latency, median latency, 95th percentile latency, throughput, and error rate. In parallel, container-level CPU and memory usage were recorded with the docker stats command [12] at one-second intervals, allowing correlation between resource consumption and observed performance.

4.3. Testing environment

For the purpose of the research scenario, a computer with the following technical parameters was used, as listed in Table 1.

Table 1: Testing bench components

Component	Parameter
RAM	16GB DDR4 3600MHz
CPU	AMD Ryzen 5 5600 3.5GHz 6 cores, 12 threads
Disk	ADATA 512GB M.2 NVMe SX8200 Pro
Motherboard	B450 Gaming Plus Max
Software	IntelliJ IDEA 2023.1.2
Operating system	Microsoft Windows 10 Pro 10.0.19045 Compilation 19045

In order to reliably and accurately replicate benchmarking performance testing procedures on systems built on top of Apache Kafka and RabbitMQ, an isolated testing environment was built using Docker and Docker Compose. This method allowed for the swift setup of intricate service matrices that included message brokers, databases, and monitoring systems.

For each RabbitMQ and Kafka for .NET application, and each Spring Boot counterpart, a corresponding test environment was configured.

RabbitMQ + .NET

The environment contains:

- RabbitMQ with a web interface running on port :15672 and a default broker listening on port :5672.
- rabbitmq-exporter for metrics available on port :9419.
- MySQL version 8.0 as the application's database.
- Prometheus and Grafana for monitoring.

Kafka + .NET

The environment contains:

- Kafka running on port :9092 and Zookeeper on :2181, both version 7.5.0.
- Kafka UI for browsing topics and messages running on port :8080.
- MySQL version 8.0 as the application's database.
- Prometheus and Grafana for monitoring.

Rabbit + Spring Boot

The environment contains:

- RabbitMQ with a web interface running on port :15672 and a default broker listening on port :5672.
- rabbitmq-exporter for metrics available on port :9419.
- MySQL version 8.0 as the application's database.
- Prometheus and Grafana for monitoring.

Kafka + Spring Boot

The environment contains:

- Kafka running on port :9092 and Zookeeper on :2181, both version 7.5.0.
- Kafka UI for browsing topics and messages running on port :8080.
- MySQL version 8.0 as the application's database.
- Prometheus and Grafana for monitoring.

General Docker Compose Settings

All containers in the environments were set the following container limits:

- CPU: 2 vCPUs.
- RAM: 2 GB.

This setup guarantees uniform measurement conditions across different tested technologies. MySQL databases were attached to the applications to emulate transactional write activity.

Base technology versions:

- Apache Kafka: version 4.0 (running with Zookeeper 7.5.0).
- RabbitMQ: version 4.1 (with rabbitmq_management plugin enabled).
- .NET Runtime: .NET 8.0 (ASP.NET Core).
- Kotlin: version 2.1.21 running on Spring Boot 3.3.x (JVM 21).
- MySQL Database: version 8.0.

Each tested stack RabbitMQ with .NET, Kafka with .NET, RabbitMQ with Spring Boot, and Kafka with Spring Boot was deployed in a dedicated Docker Compose network to prevent cross-interference. Applications were built either in ASP.NET Core 8.0 (C#) or Spring Boot 3.3.x (Kotlin 2.1.21, JVM 21) and implemented both producer and consumer endpoints, communicating with a MySQL 8.0 database for transactional persistence. RabbitMQ 4.1 brokers listened on port 5672, provided a web UI on port 15672, and exposed metrics via rabbitmq-exporter on port 9419. Apache Kafka 4.0 brokers operated on port 9092 with Zookeeper 7.5.0 on port 2181 and included a Kafka UI on port 8080. Each setup also ran Prometheus for metric collection and Grafana for real-time visualization. All containers were limited to 2 vCPUs and 2 GB RAM to ensure identical measurement conditions, and broker states were reset before each benchmark to avoid caching effects. Each environment also included a MySQL 8.0 container for handling transactional writes and reads, ensuring the database interactions resembled real-world production systems.

5. Research results

The brokers were evaluated on response time, throughput, memory consumption, and error rate. Results were obtained using Apache JMeter 5.6.3, which generated the test workloads and exported aggregate reports with latency, throughput, and error statistics. Container-level CPU and RAM usage were recorded with docker stats and cross-checked in Prometheus/Grafana. Each scenario was executed three times in an isolated Docker Compose environment, and the mean values were used for the tables and charts presented below.

5.1. .NET Performance

In the .NET ecosystem, both Kafka and RabbitMQ were evaluated in terms of average response times, throughput, and memory usage, which is summarized in the Table 2 below.

Table 2: Performance comparison .NET

Metric	RabbitMQ (.NET)	Kafka (.NET)
Avg. response time (ms)	2,130	2,081
Median response time (ms)	1,876	1,754
95th percentile (ms)	4,596	4,108
Throughput (req/sec)	1,272.7	1,279.1
Error rate (%)	16.98%	1.82%
RAM usage (avg.)	~146 MB	~400 MB

Response times were superior for Kafka, alongside an improved value in the error rate; however, the error rate for RabbitMQ was considerably higher.

5.2. Spring Boot performance

There were differences with regards to the performance trends in the Spring Boot environment. Summarized results are provided in Table 3.

Table 3: Performance comparison Spring Boot

Metric	RabbitMQ (Spring)	Kafka (Spring)
Avg. response time (ms)	1,347	1,593
Median response time (ms)	1,153	1,510
95th percentile (ms)	3,081	2,869
Throughput (req/sec)	1,248.6	1,232.1
Error rate (%)	1.61%	6.63%
RAM usage (avg.)	~130 MB	~450 MB

Compared to the .NET results, with Spring Boot, RabbitMQ showed better average and median response times along with lower memory consumption, while in Kafka, higher error rates were noted.

5.3. Visual comparition

This section presents a visual comparison of the performance metrics collected during the experiments. Charts and graphs are used to highlight the key differences between RabbitMQ and Apache Kafka across different load scenarios and technology platforms. Likewise, Kafka consumes much more memory in both environments, as illustrated in Figure 1.

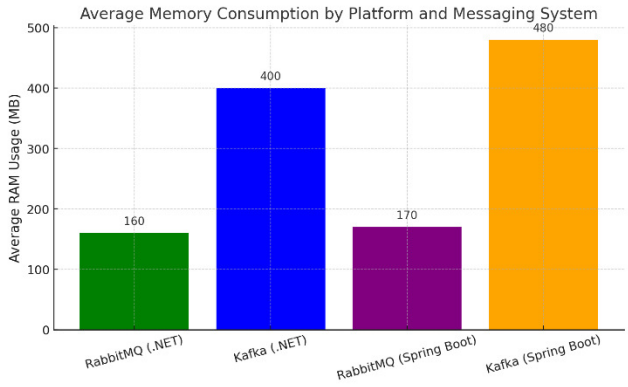

Figure 1: Average memory usage.

RabbitMQ had approximately the same throughput as Kafka; however, Kafka marginally excelled in performance over RabbitMQ in .NET what can be seen in Figure 2.

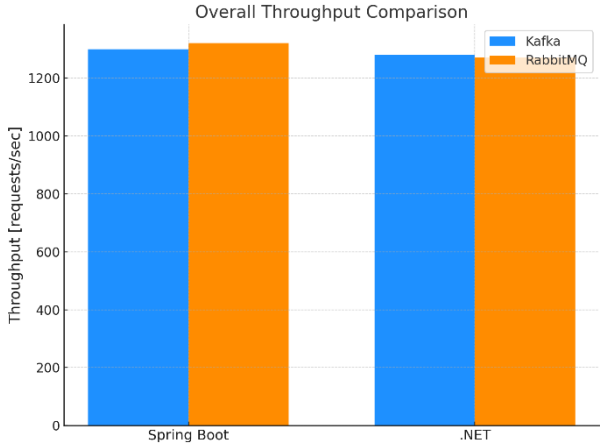

Figure 2: Overall throughput comparison.

6. Conclusions

The present study evaluated the message streaming capabilities of RabbitMQ and Apache Kafka using two software stacks: .NET and Spring Boot. The findings show that Apache Kafka was more consistent in the .NET environment yielding lower latency, higher throughput, and far fewer errors – though higher memory consumption.

RabbitMQ, on the other hand, outperforms in the Spring Boot ecosystem where it achieves lower response times and reduced memory usage, although Kafka still remains competitive in throughput.

The results suggest that a message broker's selection should be highly scoped to the ecosystem.

Kafka is the ideal option for .NET based systems as it provides superior reliability and efficiency in scenarios with high throughput demands and low error tolerances.

For Spring Boot based systems, lightweight and memory-constrained setups could benefit from RabbitMQ.

These findings help system architects and developers in selecting a message broker for streaming architectures targeting systems with strict requirements for latency or resource consumption.

References

[1] Kafka 4.0 Documentation, https://kafka.apache.org/documentation, [16.05.2025].

[2] RabbitMQ 4.1 Documentation, https://www.rabbitmq.com/docs, [16.05.2025].

[3] Kotlin docs Latest stable version: 2.1.21, https://kotlinlang.org/docs/home.html, [16.05.2025].

[4] .NET documentation, https://learn.microsoft.com/en-us/dotnet/, [16.05.2025].

[5] S. Henning, W. Hasselbring, Benchmarking scalability of stream processing frameworks deployed as microservices in the cloud, Journal of Systems and Software 208 (2023) 111879, https://doi.org/10.1016/j.jss.2023.111879.

[6] A. Lercher, J. Glock, C. Macho, M. Pinzger, Microservice API Evolution in Practice: A Study on Strategies and Challenges, Journal of Systems and Software 215 (2024) 112110, https://doi.org/10.1016/j.jss.2024.112110.

[7] A. Pathak, C. Kalaiarasan, RabbitMQ Queuing Mechanism of Publish Subscribe model for better Throughput and Response, Fourth International Conference on Electrical, Computer and Communication Technologies (2021) 1–7, https://doi.org/10.1109/icecct52121.2021.9616722.

[8] T. P. Raptis, C. Cicconetti, A. Passarella, Efficient topic partitioning of Apache Kafka for high-reliability real-time data streaming applications, Future Generation Computer Systems 154 (2024) 173–188, https://doi.org/10.1016/j.future.2023.12.028.

[9] S. Dyjach, M. Plechawska-Wójcik, Efficiency comparison of message brokers, J. Comput. Sci. Inst. 31 (2024) 116–123, https://doi.org/10.35784/jcsi.6084.

[10] Apache JMeter 5.6.3 Documentation, https://jmeter.apache.org/usermanual/index.html, [16.05.2025].

[11] Docker 28.3.3. Documentation, https://docs.docker.com/, [16.05.2025].

[12] S. Ronglong, C. Arpnikanondt, Signal: An open-source cross-platform universal messaging system with feedback support, Journal of Systems and Software 117 (2016) 30–54, https://doi.org/10.1016/j.jss.2016.02.018.