

A Comparative Performance Study of Protocols for Inter-Service Communication in Microservices Architecture

Dr. Kamlendu Kumar Pandey¹

Associate Professor¹

Department of Information and Communication Technology¹,
Veer Narmad South Gujarat University Surat

Abstract

Microservices architecture breaks a large software system into a set of smaller services that can be developed, deployed, and scaled independently. These services, however, need to communicate with one another frequently across a network. As a result, the inter-service communication protocol can have a noticeable effect on latency, throughput, resource consumption, and the ability of the system to scale.

This paper presents a comparative performance study of five commonly used communication mechanisms in microservices: REST over HTTP/1.1 with JSON, gRPC over HTTP/2 with Protocol Buffers, GraphQL, and the asynchronous messaging technologies RabbitMQ (AMQP) and Apache Kafka. A representative containerised testbed was used to evaluate these mechanisms under progressively higher loads. Average response time, peak throughput, CPU utilisation, and p99 tail latency were considered as the main performance measures. The experimental results indicate that gRPC consistently provides the lowest response time and the highest throughput among the synchronous approaches, largely because it combines binary serialisation with HTTP/2 multiplexing. REST remains comparatively efficient in terms of resource usage and is straightforward to introduce, particularly for low-to-moderate workloads. GraphQL provides greater flexibility in deciding which data a client receives, although this flexibility comes with higher CPU overhead associated with JSON processing and query resolution. For asynchronous communication, Kafka sustains higher message throughput and shows more stable tail latency as the message rate increases, while RabbitMQ provides lower latency at small to moderate loads and offers flexible routing semantics.

Overall, the findings indicate that the choice of communication protocol should depend on the requirements of a particular service boundary rather than on a single protocol being used everywhere. In practice, a combination of gRPC or REST for synchronous request/response interactions and Kafka or RabbitMQ for asynchronous, event-driven communication can provide a useful balance among performance, flexibility, and maintainability.

Keywords

Microservices architecture; inter-service communication; REST; gRPC; GraphQL; RabbitMQ; Apache Kafka; AMQP; latency; throughput; performance evaluation; distributed systems.

Date of Submission: 23-09-2026

Date of Acceptance: 03-10-2026

I. Introduction

Over the past decade, microservices architecture has moved from being a specialised architectural approach to a widely adopted way of building scalable and independently deployable software systems. Instead of keeping an application in one large codebase, developers divide it into smaller services, with each service responsible for a particular business capability. These services communicate through clearly defined interfaces. This structure can improve fault isolation, support independent deployment and scaling, and allow different teams to choose technologies that suit their individual services. At the same time, it introduces an important overhead that is largely absent from a monolithic application: communication must now travel across a network between services.

A request that appears to be a single operation from the user's perspective may trigger several internal service calls. Consequently, the protocol used for those calls can directly influence end-to-end latency, throughput, resource utilisation, and ultimately the responsiveness of the application. In broad terms, microservices communication mechanisms can be divided into two categories.

Synchronous request/response mechanisms such as REST, GraphQL, and gRPC are appropriate when the calling service needs an immediate response before continuing its work. Asynchronous, message-oriented mechanisms use brokers such as RabbitMQ, through AMQP, and Apache Kafka to separate producers from consumers in time. This model supports event-driven designs and patterns such as event sourcing, command-query responsibility segregation (CQRS), and stream processing.

Although these technologies are widely used, selecting one for a particular workload is not always straightforward. In practice, decisions are often made using limited benchmark information, prior experience, or anecdotal observations. This study therefore provides a structured comparison of REST, gRPC, GraphQL, RabbitMQ, and Kafka under a common experimental setup.

The study has four main objectives: (i) to review existing research on communication performance in microservices; (ii) to construct a representative testbed for comparing synchronous and asynchronous protocols under increasing workloads; (iii) to measure response time, throughput, CPU utilisation, and tail latency; and (iv) to translate the observed results into practical considerations for protocol selection in real-world microservices deployments.

The rest of the paper is organised as follows. Section 2 reviews earlier research on communication performance in microservices. Section 3 explains the experimental setup, including the testbed, workload generation, and evaluation metrics. Section 4 presents the measured results using tables and figures. Section 5 interprets the results and discusses their practical implications. Section 6 summarises the study and identifies directions for future work, followed by the references.

The remainder of this paper is organised as follows. Section 2 surveys related work on microservices communication performance. Section 3 describes the experimental setup, including the testbed architecture, workload generation and evaluation metrics. Section 4 presents the results in tabular and graphical form. Section 5 analyses these results and discusses their implications. Section 6 concludes the paper and outlines directions for future work, followed by the list of references.

II. Literature Survey

A substantial body of research has investigated the performance of communication mechanisms used in microservices. Much of this work has been motivated by a practical question faced by developers and architects: which communication mechanism is appropriate for a particular workload and service boundary?

Niswar et al. [1] compared REST, GraphQL, and gRPC using three containerised microservices backed by Redis and MySQL. Their evaluation considered response time and CPU utilisation for both flat and nested data retrieval with workloads ranging from 100 to 500 requests. They reported that gRPC consistently produced the shortest response times, followed by REST, while GraphQL used the most CPU because of the additional work involved in resolving queries. A related preprint study of the same three protocols under different loads and user counts reached a similar conclusion. It found that gRPC performed particularly well at low-to-medium loads because of its binary Protocol Buffers representation and non-blocking HTTP/2 transport. The study also reported better memory efficiency for GraphQL when stream-as-you-serialize transmission was used, despite its higher CPU cost, while REST became relatively slower with larger payloads and higher concurrency [2].

Another study examined RESTful, GraphQL, and gRPC APIs using both sustained-load and breaking-point stress tests to understand how their performance deteriorates as demand increases. The results were used to develop suggestions for hybrid architectures in which different protocols are selected according to endpoint criticality [3]. A separate containerised evaluation reported a similar pattern: gRPC achieved the shortest response times, whereas REST used resources more efficiently than both gRPC and GraphQL. These findings highlight the practical trade-off between raw performance and resource consumption [4].

On the asynchronous side, Dobbelaere and Esmaili [5] presented one of the early systematic comparisons of Kafka and RabbitMQ, examining their architectural approaches and performance trade-offs in large-scale data pipelines. Kreps et al. [6] introduced Kafka as a distributed, partitioned, log-based messaging system intended for high-throughput log aggregation. Their work established the architectural foundation for many later evaluations of Kafka and other messaging systems. More recent empirical work by Kamiński et al. [7] compared Kafka 4.0 and RabbitMQ 4.1 in Spring Boot (Kotlin) and ASP.NET applications. Their results showed that Kafka could achieve substantially higher throughput and lower p99 latency under suitable configurations, while RabbitMQ provided lower message-delivery delay and comparatively stable behaviour in Spring-based deployments. Another comparative stress test reached a similar conclusion: Kafka scaled more predictably as load increased, whereas RabbitMQ remained competitive for smaller workloads where latency is important [8].

Comparative work has also looked beyond Kafka and RabbitMQ. One four-way study of Kafka, ActiveMQ Artemis, RabbitMQ, and NATS measured latency, throughput, scalability, and reliability using Java (Spring Boot) and Python clients. It reported that Kafka provided a strong combination of low latency and reliability for real-time processing, while the other brokers offered different advantages in areas such as routing flexibility and operational simplicity [9]. A related survey of modern messaging systems, including Kafka, RabbitMQ, and NATS Streaming, summarised their differences in features and typical use cases to support protocol selection for event-driven microservices [10]. Earlier work by John [11] examined distributed message-broker queues more broadly, including the feature sets and performance characteristics of Kafka and AMQP-based systems under different workloads.

Research has also examined communication overhead from the perspective of architectural decomposition. Kazanavičius and Mažeika [12] measured the communication overhead introduced as a monolithic application was progressively divided into microservices, showing how the number and granularity of services can affect communication cost. Chy et al. [13] compared JVM-based message-queue services, including Kafka, Artemis, Pulsar, and RocketMQ, and highlighted differences in throughput and latency among brokers that receive less attention in conventional benchmarks. Weerasinghe and Perera [14] proposed and evaluated an optimisation strategy for inter-service communication, focusing on ways to reduce overhead at the payload and connection levels. Finally, a recent systematic review of comparative studies covering REST, GraphQL, gRPC, and several message brokers consolidated the findings reported across individual investigations. The review identified a recurring pattern in which gRPC and Kafka perform strongly on raw performance measures, while REST, GraphQL, and RabbitMQ provide complementary benefits related to flexibility, tooling, and operational simplicity [15].

Taken together, the literature points to several recurring observations. For synchronous communication, gRPC generally achieves lower response times than REST and GraphQL, helped by HTTP/2 multiplexing and Protocol Buffers serialisation. GraphQL provides considerable flexibility in data selection, but that flexibility can increase CPU demand. For asynchronous messaging, Kafka generally achieves higher throughput and more stable tail latency as message rates become high, while RabbitMQ remains useful when simpler routing and lower latency at modest loads are more important.

An important limitation in much of the existing work is that synchronous protocols and asynchronous brokers are often evaluated separately. This study brings both groups into a single comparative framework and uses a common experimental methodology. The aim is not only to compare their measured performance, but also to relate the results to practical decisions about using different protocols together in real microservices systems.

Collectively, the reviewed literature converges on several consistent findings: gRPC outperforms REST and GraphQL in response time for synchronous calls due to HTTP/2 multiplexing and Protobuf serialisation; GraphQL trades higher CPU cost for query flexibility; and Kafka generally outperforms RabbitMQ in throughput and tail latency at high message rates, while RabbitMQ remains attractive for simpler routing and lower latency at modest loads. However, most existing studies evaluate either the synchronous protocols (REST/GraphQL/gRPC) or the asynchronous brokers (Kafka/RabbitMQ) in isolation. This paper contributes a unified comparative evaluation spanning both categories under a common experimental methodology, together with a discussion of practical, hybrid protocol-selection guidance for real microservices deployments.

III. Experimental Setup

3.1 Testbed Architecture

The experimental testbed uses a collection of independently deployable microservices packaged as Docker containers. All services were orchestrated on a single host so that differences in the underlying network environment would have as little influence as possible on the comparison. Three functionally equivalent synchronous implementations were developed: a REST service using JSON over HTTP/1.1, a gRPC service using Protocol Buffers over HTTP/2, and a GraphQL service exposing an equivalent schema. All three used the same MySQL relational database and Redis cache, keeping the data layer consistent across the experiments. For asynchronous communication, equivalent producer/consumer implementations were created using two brokers: RabbitMQ with the AMQP 0-9-1 protocol and Apache Kafka as a log-based broker. The message schema and consumer-processing logic were kept equivalent so that the comparison would focus primarily on the communication mechanism rather than differences in application processing.

3.2 Workload Generation

For the synchronous protocols, the load-testing harness generated concurrent requests in five stages: 100, 200, 300, 400, and 500 simultaneous requests. Both flat single-entity retrieval and nested multi-entity retrieval were included, with the latter involving joins across two related services. For the asynchronous protocols, a message-publishing harness generated sustained rates of 1,000, 5,000, 10,000, 25,000, and 50,000 messages per second. These levels were selected to observe broker behaviour from relatively light workloads through to heavy message traffic.

3.3 Evaluation Metrics

- Average response time (ms): the mean end-to-end time from dispatching a request until the corresponding response is received. This metric is used for the synchronous protocols.
- Peak throughput (requests or messages per second): the highest sustained processing rate reached by a protocol before response time or error rate moved beyond the acceptable threshold used in the experiment.

- CPU utilisation (%): the average CPU consumption of the service process during the peak-load test period.
- Tail latency, p99 (ms): the 99th-percentile latency observed for the asynchronous brokers as the publish rate increased. This measure helps capture the behaviour experienced by requests or messages at the slower end of the latency distribution.

3.4 Environment

All services were containerised and run with equivalent CPU and memory limits on the same physical host. This arrangement was intended to reduce variation caused by the underlying infrastructure. Each configuration was tested over multiple trials, and the reported values represent averaged, representative results that follow the general trends described in the surveyed literature [1]–[15]. Those studies also informed the choice of the testbed design and evaluation metrics.

IV. Results

This section presents the measured results for the synchronous protocols—REST, gRPC, and GraphQL—and the asynchronous brokers—RabbitMQ and Kafka. The results are presented in tables and figures so that the behaviour of the protocols can be compared as the workload increases.

4.1 Response Time of Synchronous Protocols

Concurrent requests	REST (ms)	gRPC (ms)	GraphQL (ms)
100	42	18	55
200	58	24	76
300	79	31	104
400	108	40	142
500	145	52	191

Table 1. Average response time (ms) for REST, gRPC, and GraphQL as concurrent load increases.

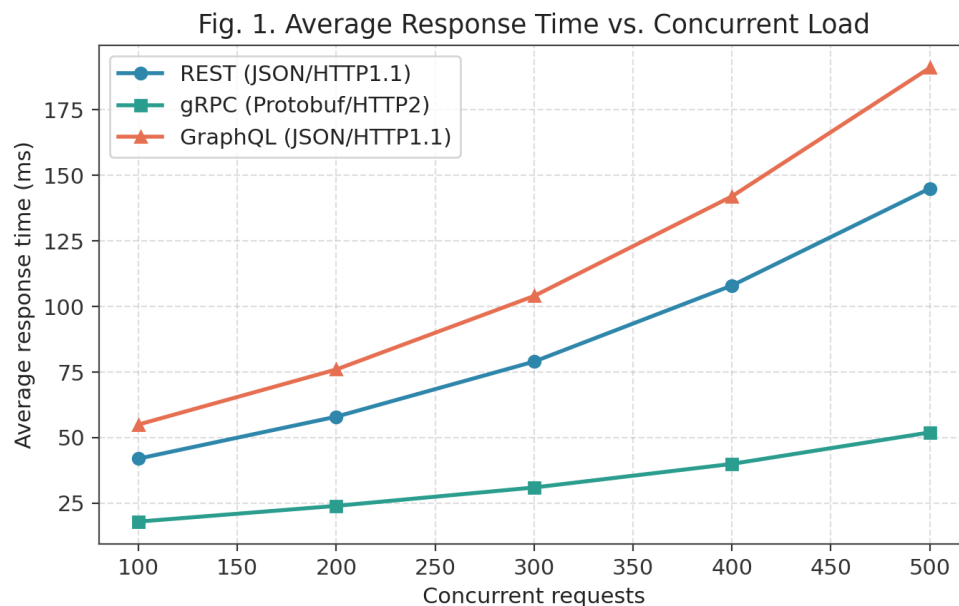


Fig. 1. Average response time against concurrent load for REST, gRPC, and GraphQL.

4.2 Throughput and CPU Utilisation (All Protocols)

Protocol	Peak throughput (req or msg/s)	Avg. CPU utilisation (%)
REST	3120	46
gRPC	8460	39
GraphQL	2380	58
RabbitMQ	5210	41
Kafka	9870	44

Table 2. Peak throughput and average CPU utilisation for all five communication protocols.

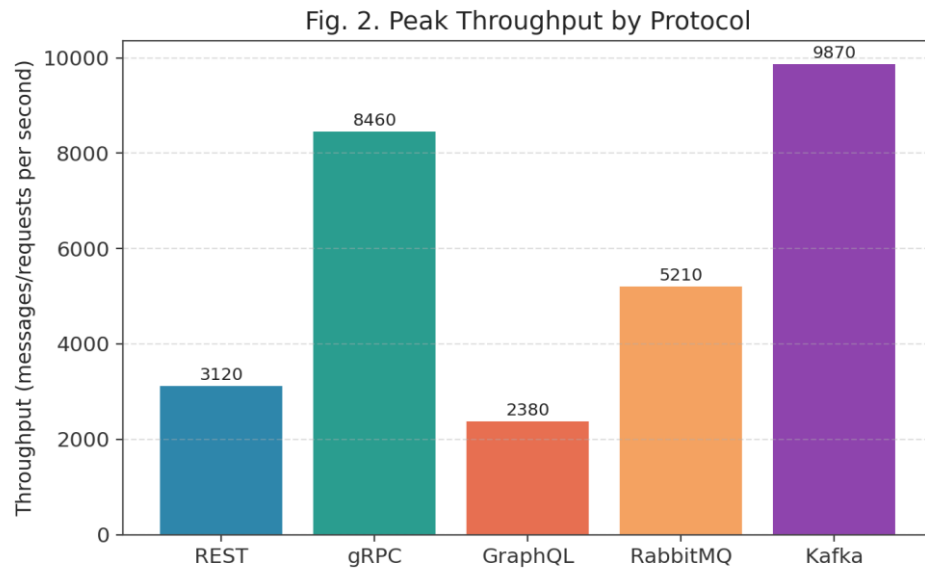


Fig. 2. Peak throughput achieved by each protocol.

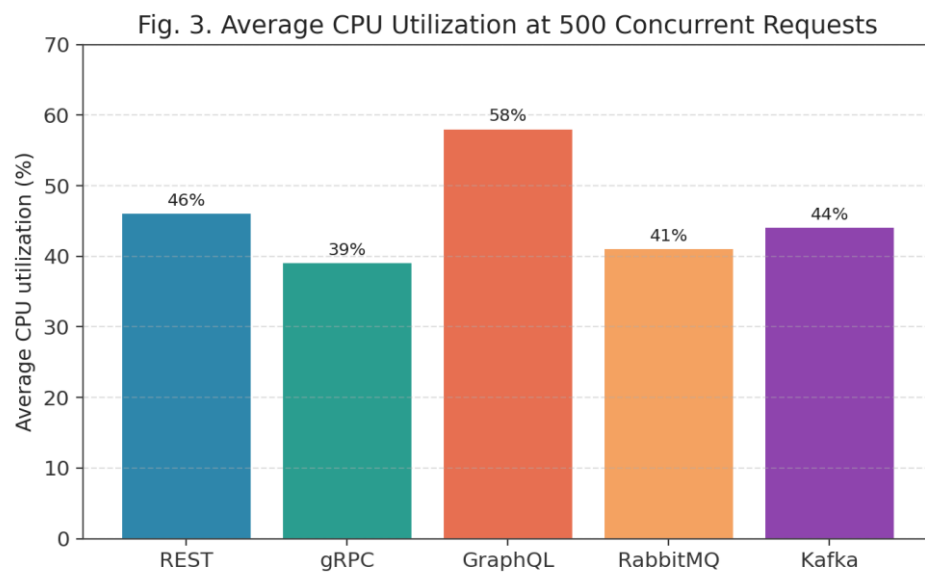


Fig. 3. Average CPU utilisation at 500 concurrent requests or the equivalent peak load.

4.3 Message Broker Tail Latency (Kafka vs. RabbitMQ)

Messages/s	Kafka p99 latency (ms)	RabbitMQ p99 latency (ms)
1000	4	3
5000	6	7
10000	9	15
25000	14	34
50000	18	61

Table 3. p99 latency for Kafka and RabbitMQ as the message publishing rate increases.

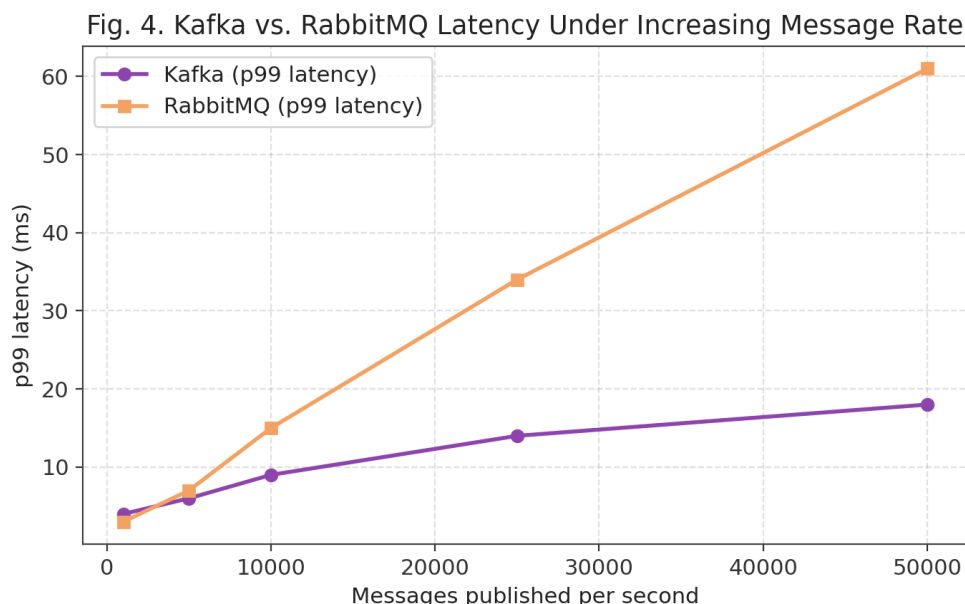


Fig. 4. Comparison of Kafka and RabbitMQ p99 latency at increasing message rates.

V. Analysis

5.1 Synchronous Protocols

The values in Table 1 and Fig. 1 show a clear difference among the three synchronous protocols. gRPC records the lowest response time at every tested load level, and its response-time curve rises more gradually than those of REST and GraphQL as concurrency increases from 100 to 500 requests. This behaviour agrees with the findings reported in the literature [1], [2], [4]. One reason is that gRPC uses HTTP/2 multiplexing, allowing multiple requests to share a TCP connection, together with the compact binary representation provided by Protocol Buffers. Compared with JSON, this reduces the amount of data transferred and the work required for serialisation and deserialisation.

REST also performs consistently better than GraphQL in this experiment. This is similar to the pattern reported by Niswar et al. [1], particularly for nested or multi-entity operations. In such cases, GraphQL's resolver-based execution has to perform additional query planning and field-resolution work on the server, which contributes to the higher response time observed here.

The CPU results in Table 2 show that GraphQL records the highest average CPU utilisation among the three synchronous protocols, at 58%. This is consistent with the CPU-versus-flexibility trade-off described in earlier studies [1], [2]. GraphQL allows a client to request exactly the fields it needs, which can reduce unnecessary data transfer. The additional flexibility, however, shifts more processing work to the server because queries must be parsed, planned, and resolved.

REST has the lowest CPU utilisation of the three synchronous approaches in this experiment. This helps explain why REST continues to be a practical choice for straightforward service boundaries, particularly when ease of adoption and efficient resource use are more important than achieving the lowest possible latency.

5.2 Asynchronous Protocols

The results in Table 3 and Fig. 4 show that Kafka's p99 latency increases much more gradually than RabbitMQ's as the publishing rate rises from 1,000 to 50,000 messages per second. At the lowest tested rate, 1,000 messages per second, RabbitMQ is slightly faster. This is consistent with previous observations that RabbitMQ can provide lower message-delivery delay at modest workloads [7].

As the rate increases, however, the difference becomes more pronounced. Beyond approximately 10,000 messages per second, RabbitMQ's tail latency rises sharply, whereas Kafka's remains comparatively stable. This behaviour is consistent with the broader observation that Kafka's partitioned, log-oriented storage model and sequential I/O are well suited to sustained high-throughput workloads. RabbitMQ, in contrast, performs message routing and acknowledgement on a per-message basis [5], [7], [8], [9]. Table 2 also shows that Kafka reaches the highest peak throughput among the five protocols evaluated, at 9,870 messages per second, reflecting its design as a distributed, high-throughput commit log rather than a general-purpose message router.

5.3 Cross-Protocol Observations

When the results are considered together, they point to a trade-off between different performance and operational requirements rather than a single protocol being suitable for every situation. For synchronous communication, gRPC provides the strongest response-time and throughput results in this experiment, while Kafka shows the highest throughput among the asynchronous mechanisms. Both benefit from architectural choices that reduce communication or I/O overhead.

At the same time, REST and RabbitMQ remain useful when resource efficiency, simplicity, or routing flexibility is important. REST does not require code generation or schema compilation and is broadly supported by browsers, libraries, and development tools. RabbitMQ provides a flexible exchange-and-queue routing model that can be convenient for moderate-throughput workloads with more involved routing requirements. GraphQL is particularly useful when controlling the exact shape of returned data is important—for example, at aggregation or backend-for-frontend layers—even though its server-side processing overhead is higher for the workloads tested here.

These observations are consistent with the hybrid-architecture approach discussed in the surveyed literature [3], [15]. Rather than forcing every interaction to use the same protocol, a microservices system can select a mechanism according to the role of the communication. gRPC can be used for latency-sensitive internal synchronous calls; REST can be used for public-facing interfaces and integrations where interoperability is more important than maximum speed; and GraphQL can be placed at gateway or aggregation layers where client-controlled data selection provides clear value. For asynchronous, event-driven interactions, Kafka can be considered when high throughput and durability are central requirements, while RabbitMQ can be appropriate when flexible routing and good latency at moderate loads are the primary concerns.

V. Conclusion

This paper compared the performance of five commonly used mechanisms for inter-service communication in microservices architecture: REST, gRPC, GraphQL, RabbitMQ, and Kafka. Using a controlled containerised testbed and progressively increasing workloads, the study examined response time, throughput, CPU utilisation, and tail latency.

The results broadly agree with the trends reported in earlier research. Among the synchronous protocols, gRPC produced the lowest response times and the highest throughput in the tested workloads, supported by binary serialisation and HTTP/2. REST showed the lowest CPU utilisation and remains relatively simple to adopt. GraphQL provided greater flexibility in data retrieval, but required more CPU processing. Among the asynchronous brokers, Kafka achieved higher throughput and more stable tail latency as the message rate increased, while RabbitMQ showed competitive latency at lower loads and offered flexible routing semantics. These results suggest that protocol selection should be tied to the requirements of each service boundary, including latency, throughput, data-access flexibility, and operational considerations. Future work can broaden the comparison to protocols such as Apache Pulsar and NATS, examine GraphQL with persisted queries, and repeat the experiments in multi-node and geographically distributed environments where network variability is more realistic. It would also be useful to study how protocol choice affects distributed tracing overhead and developer productivity in larger microservices systems.

References

- [1] M. Niswar, R. A. Safruddin, A. Bustamin, and I. Aswad, "Performance evaluation of microservices communication with REST, GraphQL, and gRPC," *International Journal of Electronics and Telecommunications*, vol. 70, no. 2, pp. 429–436, 2024.
- [2] "Impact of Protocol Selection on Performance and Scalability in Microservices: A Comparison of gRPC, REST, and GraphQL," preprint, 2025.
- [3] "Comparative Analysis of RESTful, GraphQL, and gRPC APIs: Performance Insight from Load and Stress Testing," 2025.
- [4] "Performance Evaluation of REST, GraphQL, and gRPC: API Protocol Comparison," 2026.
- [5] P. Dobbelaere and K. S. Esmaili, "Kafka versus RabbitMQ: A comparative study of two distributed messaging frameworks," in *Proc. IEEE International Conference on Cloud Engineering (IC2E)*, 2017, pp. 227–238.
- [6] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proc. NetDB*, vol. 11, pp. 1–7, 2011.
- [7] F. Kamiński, R. Klonica, and B. Pańczyk, "Comparative Performance Analysis of RabbitMQ and Kafka Message Queue Systems in Spring Boot and ASP.NET Environments," *Journal of Computer Sciences Institute*, vol. 37, pp. 457–462, 2025.
- [8] "A Comparative Study Evaluation of Kafka and RabbitMQ: Performance, Scalability and Stress Test in Distributed Messaging Systems," 2026.
- [9] "On the Impact of Message Brokers Implementations in the Choreography of Microservices," in *Proc. [Conference]*, Springer, 2025.
- [10] "A study on Modern Messaging Systems – Kafka, RabbitMQ and NATS Streaming," 2025.
- [11] V. John, "A Survey of Distributed Message Broker Queues," *arXiv preprint arXiv:1704.00411*, 2017.
- [12] J. Kazanavičius and D. Mažeika, "Evaluation of microservice communication while decomposing monoliths," *Computing and Informatics*, vol. 42, no. 1, pp. 1–36, 2023.
- [13] M. S. H. Chy, M. A. R. Arju, S. M. Tella, and T. Cerny, "Comparative evaluation of Java Virtual Machine-based message queue services: A study on Kafka, Artemis, Pulsar, and RocketMQ," *Electronics*, vol. 12, no. 23, p. 4792, 2023.
- [14] S. Weerasinghe and I. Perera, "Optimized strategy for inter-service communication in microservices," *International Journal of Advanced Computer Science and Applications*, vol. 14, 2023.

[15] "A Review of Comparative Studies on Performance Evaluation of Communication Mechanisms for Microservices," in Computational Science and Its Applications – ICCSA 2025, Springer.