JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 37 (2025) 405–409
Received: 17 June 2025
Accepted: 6 July 2025

Performance and scalability analysis of monolithic and microservice architectures in social networks

Viacheslav Chernohor[a,*]

[a] *Department of Computer Science, Kharkiv National University of Radio Electronics, Nauky Ave, 14, 61166 Kharkiv, Ukraine*

Abstract

The article is focused on the research of the efficiency of using monolithic and microservice architectures in web applications. A comparative analysis of seven architectural configurations is made according to the following metrics: response time, resource consumption, number of processed requests, and cost of deployment in AWS. Docker, Postman, Prometheus, and Grafana were used to collect metrics. The results of the experiment allowed us to determine the optimal architectures for different load levels and formulate recommendations for choosing architectural solutions in modern systems.

Keywords: microservice architecture; monolith architecture; performance; cloud deployment costs

*Corresponding author

Email address: **viacheslav.chernohor@nure.ua** (V. Chernohor)

1. Introduction

In recent years, the choice of software architecture for web applications has become one of the key issues in information systems development. Monolithic architecture has traditionally been used due to its simplicity, fast development, and convenient centralized management, but it has significant limitations in terms of scalability and flexibility. At the same time, microservice architecture is gaining popularity as an approach that allows dividing a system into independent services, each of which can be scaled, updated, and maintained independently.

This work aims to conduct an experimental analysis of the performance and efficiency of various architectural approaches, including a monolithic model and several variants of microservice architecture, based on system load, resource consumption, and deployment costs.

The research includes an analysis of seven architectures, with subsequent performance testing using Postman, Prometheus, and Grafana. The main hypothesis of the study is that microservices demonstrate higher efficiency in medium-load and high-load environments, while monolithic architecture is still appropriate for simple and lightly loaded systems.

2. Materials and methods

2.1. Research object

The research object is a social-type web application that implements the basic functionality of interaction between users: authentication, creating and viewing posts, messaging, and receiving system notifications. To implement the server side, we developed a set of REST API controllers divided into logical areas of responsibility. Based on a single business logic, seven different architectural configurations were implemented, which differ in the structure of interaction between components, approaches to request processing and data storage.

The following architectures were considered in the research:

1. Monolithic architecture [1] - all components are implemented within a single application with a single database (Figure 1).

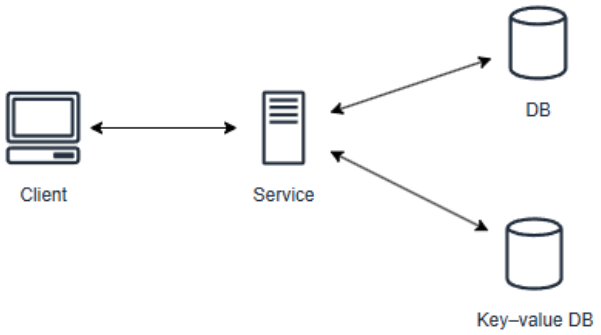

Figure 1: Monolith architecture.

2. Microservice architecture [2] - the application is divided into three services; the services share a single database (Figure 2).

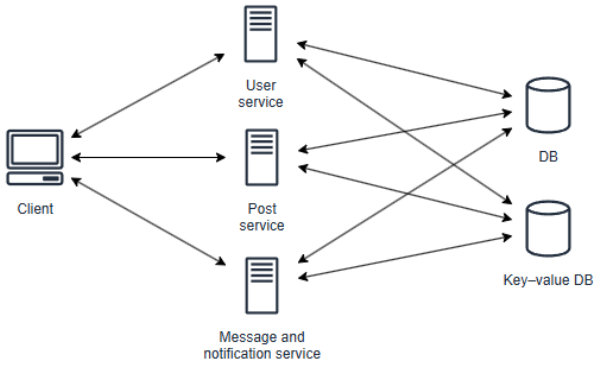

Figure 2: Microservice architecture.

3. Microservices with API Gateway [3] - a basic microservice architecture with the additions of API Gateway - a service that is a centralized gateway that routes requests to individual services (Figure 3).

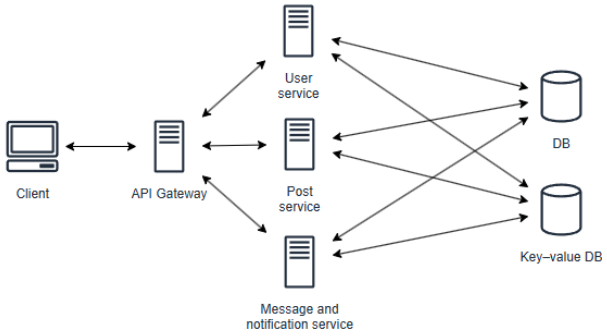

Figure 3: Microservice architecture with API Gateway.

4. Microservices with API Gateway and RabbitMQ [4] - interaction between services occurs asynchronously through the RabbitMQ message broker, requests are transmitted through the Gateway API (Figure 4).

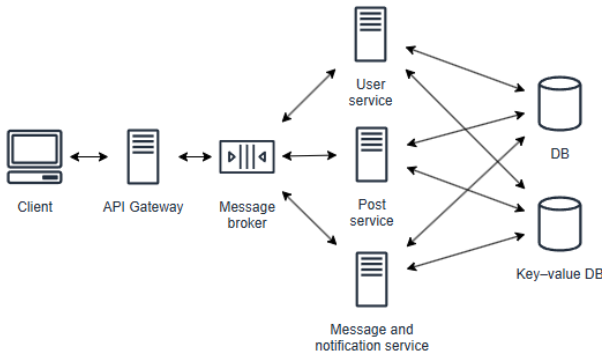

Figure 4: Microservice architecture with API Gateway and message broker.

5. Microservices with the Database-per-Service pattern [5] - each microservice has its own separate database, which provides logical and technical data isolation (Figure 5).

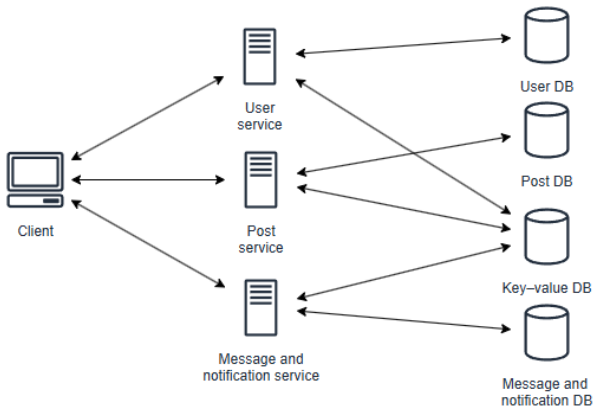

Figure 5: Microservice architecture with split database.

6. Microservices with the Database-per-Service and API Gateway pattern - each microservice has its own separate database, with the addition of a centralized gateway that routes requests to individual services (Figure 6).

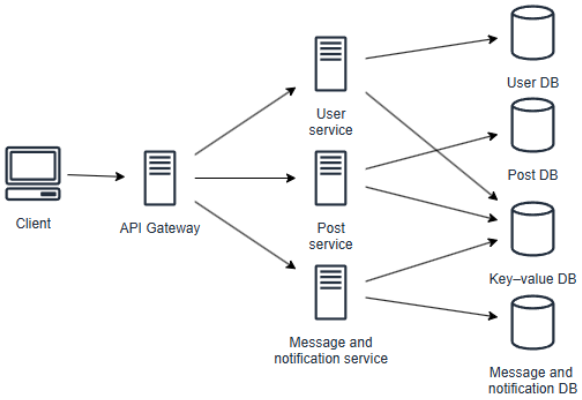

Figure 6: Microservice architecture with split database and API Gateway.

7. Microservices with the Database-per-Service pattern, API Gateway, and RabbitMQ - an architecture that combines the API Gateway, asynchronous interaction through the RabbitMQ broker, and complete database isolation (Figure 7).

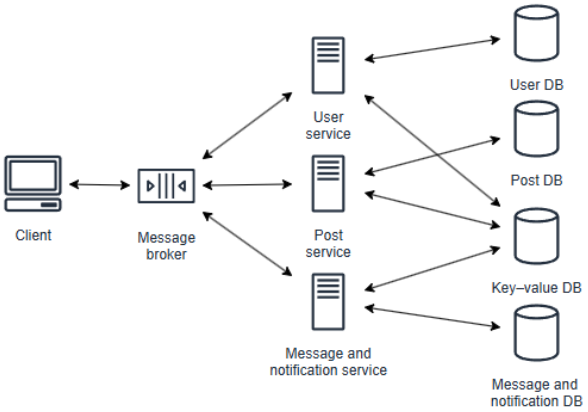

Figure 7: Microservice architecture with split database, API Gateway and message broker.

2.2. Description of the test method

The research method is experimental and comparative, involving the implementation of each architecture as an isolated environment based on Docker containers. The goal was to evaluate the performance, resource efficiency, and overall economic feasibility of each configuration.

The following metrics were collected for each architecture:

- Average response time - measured while executing queries using the Postman [6] tool in performance testing mode.
- CPU usage - collected automatically from containers through the cAdvisor [7] agent that integrates with Prometheus [8], this metric is measured as a percentage of the total available computing power of one processor core, i.e. it can exceed 100%.
- RAM usage was also collected through Prometheus and cAdvisor.

- The number of messages processed in the queue - was recorded only for architectures using RabbitMQ using the built-in RabbitMQ monitoring tools.
- Estimated monthly cost of deployment - calculated manually in the AWS Pricing Calculator [9] based on the infrastructure parameters selected for each architecture.

2.3. Conducted research

The experiment was conducted in a controlled environment by sequentially running each of the seven architectures in Docker [10] containers. For each architectural configuration, separate containers were run with services, databases, Redis, a message broker (if applicable), and additional components (e.g., API Gateway). All architectures were tested on the same hardware configurations:

- Processor: AMD Ryzen™ 5 3550H quad-core processor (4 cores, 8 threads, 6MB cache, 3.7GHz max.).
- RAM: 16 GB DDR4-2400 SO-DIMM.
- Storage: 512 GB SSD (PCIe® 3.0 NVMe™ M.2).
- Operating system: Pop!_OS (based on Linux).

For each architecture, three series of performance tests were conducted: at low, medium, and high load levels. The load was generated using Postman, which had pre-created collections of requests that emulated typical user actions: authorization, viewing posts, creating messages, and updating a profile. For each load level, a certain number of requests were executed with the same distribution by action.

All performance metrics (average response time, number of requests per second, CPU usage, memory usage) were recorded during the testing process. To do this, we used the stack of Prometheus and cAdvisor, which automatically polled Docker containers and saved the collected metrics. Visualization was performed in Grafana [11], where unified dashboards were set up for each configuration. For architectures with message brokers, the number of processed messages in the queue was also tracked using the built-in tools of the RabbitMQ message broker. Figure 8 shows a diagram of the test environment for the microservice architecture.

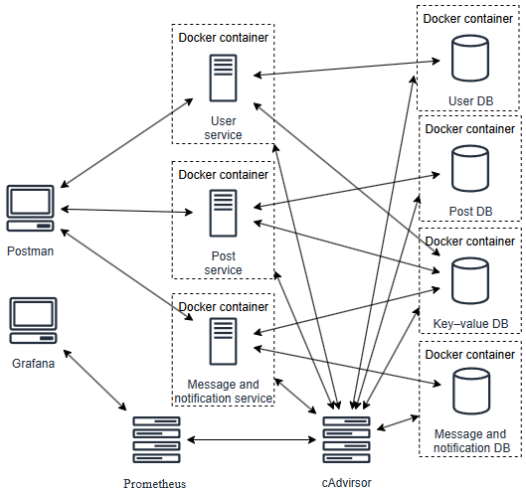

Figure 8: Test environment for microservice architecture.

After each experiment, the cost of cloud deployment for the corresponding configuration was calculated using the AWS Pricing Calculator, taking into account EC2 instance types, amount of RAM, number of services, usage of message brokers, Redis, API Gateway, and other components. The configurations were designed to support a system with two hundred concurrent users.

3. Presentation of results

As a result of the experiment, numerous quantitative indicators were obtained that allowed comparing the architectures with each other. Table 1 shows the metrics collected during the testing of the systems of each system implementation at low load.

The abbreviations are used in the results:

A1 – Monolithic architecture

A2 – Microservice architecture

A3 – Microservices with API Gateway

A4 – Microservices with API Gateway and RabbitMQ

A5 – Microservices with the Database-per-Service pattern

A6 – Microservices with API Gateway and Database-per-Service pattern

A7 – Microservices with the Database-per-Service pattern, API Gateway, and RabbitMQ

Table 1: Results of testing at low load

Arch. type	Average response time(ms)	CPU usage(%)	Memory usage (MiB)	Number of mes. (mps)
A1	102	294.705	627.15	-
A2	104	293.316	881.7	-
A3	132	292.682	1089.17	-
A4	159	304.204	1204.57	200
A5	94	287.346	889.34	-
A6	173	299.801	1116.13	-
A7	123	290.824	1229.85	240

Table 2 shows the metrics collected during the testing of the systems of each system implementation at medium load.

Table 2: Results of testing at medium load

Arch. type	Average response time(ms)	CPU usage(%)	Memory usage (MiB)	Number of mes. (mps)
A1	820	292.286	595.4	-
A2	758	305.387	1476.35	-
A3	921	197.383	1 378.2	-
A4	1106	290.267	1 177.18	320
A5	726	281.714	908.12	-
A6	1190	246.37	1267.06	-
A7	875	288.282	2 473.04	350

Table 3 shows the metrics collected during the testing of the systems of each system implementation at high load.

Figure 9 shows a comparison of the average response time of each architecture implementation when tested under different loads.

Table 3: Results of testing at high load

Arch. type	Average response time(ms)	CPU usage(%)	Memory usage (MiB)	Number of mes. (mps)
A1	2433	257.164	398.13	-
A2	2188	241.8	847.75	-
A3	2735	221.856	953.21	-
A4	2963	225.079	1042.99	300
A5	2056	213.924	706.26	-
A6	2986	217.689	942.61	-
A7	2378	273.093	1122.72	320

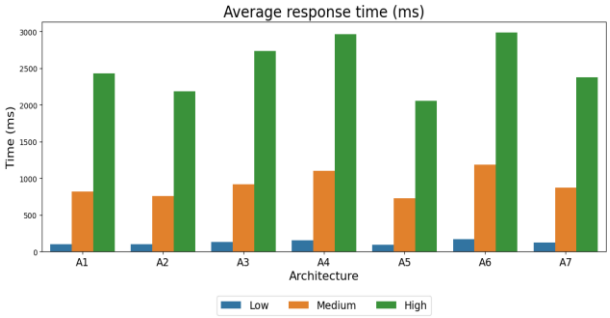

Figure 9: Average response time(ms).

Figure 10 shows a comparison of the CPU resource usage of each architecture implementation variant when tested under different loads.

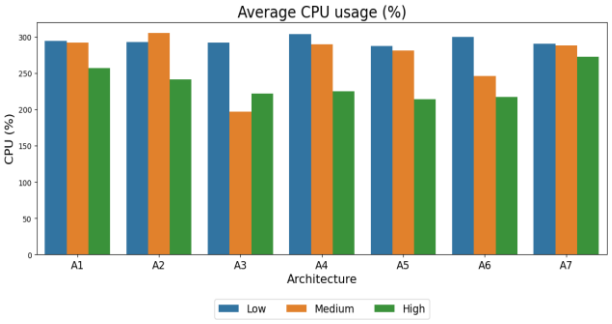

Figure 10: Average CPU usage.

Figure 11 shows a comparison of the memory usage of each architecture implementation when tested under different loads.

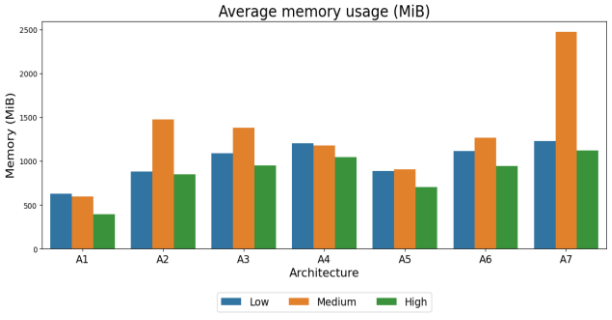

Figure 11: Average memory usage.

Figure 12 shows the number of messages received by the message broker for the architecture implementation variants where the message broker is used during testing under different loads.

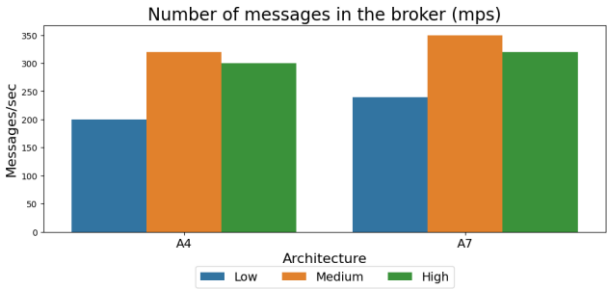

Figure 12: Number of messages in the message broker.

The following components were used to calculate the estimated monthly cost of deployment:

- An ElastiCache service of the type cache.t3.small with an average memory usage of 1 GB was used in the calculations for each system type.
- The EC2 service of the t3.large type was used to calculate the system with a monolithic architecture, and the t3a.medium type was used for calculations for the microservice architecture for each service.
- To calculate the cost of using the API Gateway, additional EC2 service of the t3a.medium type was used.
- An RDS service of the db.t3.medium type with 50GB of memory was used for architectures with a single database.
- Three RDS services of the db.t3.small type with 20GB of memory were used for architectures with a split database by a database.
- Amazon RabbitMQ Broker service of the mq.t3.micro type with 20GB of memory was used was used in systems where a message broker is required.

Table 1 shows the estimated cost of deploying a system with a specific architecture per month on AWS.

Table 4: Estimated cost of system deployment per month on AWS

Architecture type	Price per month($/month)
A1	256.48
A2	294.73
A3	333.12
A4	358.28
A5	326.76
A6	365.15
A7	390.31

4. Conclusions

The experimental research compared seven architectural models of a web application based on microservice and monolithic approaches under three load conditions: low, medium, and high. The research confirmed the hypothesis that the microservice architecture, especially using the Database-per-Service design pattern, demonstrates better performance with increasing load, while the monolithic model remains efficient in terms of cost and resource usage at low traffic rates.

According to the results, Microservices with DpS showed the lowest average response time in all three scenarios: 94 ms (low load), 726 ms (medium load), and 2056 ms (high load). This result is caused by the isolation

of data access, which reduces competition between services. At the same time, the monolithic architecture consistently remained the least expensive to deploy (approximately $256/month) and the most economical in terms of memory usage.

The Gateway and RabbitMQ APIs, despite their advantages in centralization and asynchronous processing, significantly increased response times, which was especially noticeable under high load (up to 2986 ms). This confirms that such components should be used only in cases where there are specific requirements, such as security, integration with external systems, or a distributed environment with many clients.

Important results of the research:

- It is confirmed that using the Database-per-Service design pattern significantly improves performance in high-load systems.
- It was found that the use of AG significantly increases the response time, even when data is isolated.
- The results demonstrate that the monolithic architecture is still appropriate for systems with a limited budget and a small number of users.

Possible sources of error are the load emulation, i.e. user emulation through Postman and simplified configuration during containerization, which do not fully reproduce the behavior of a cloud environment with distributed nodes.

Main conclusions:

1. Microservices using the Database-per-Service design pattern are the best option for medium to high load performance.
2. The monolith remains the best solution for small projects or MVPs due to low infrastructure costs.
3. Adding API Gateway and message brokers is only advisable if there are clear non-functional requirements, as it reduces performance and increases cost.

Areas for further research include testing the scaling of services separately, studying the impact of the choice of DBMS when using the Database-per-Service design pattern, and extending the experiment to deploy test systems in cloud environments and scenarios with real users.

References

[1] G. Blinowski, A. Ojdowska, A. Przybyłek, Monolithic vs. microservice architecture: A performance and scalability evaluation, IEEE Access 10 (2022) 20357–20374, https://doi.org/10.1109/ACCESS.2022.3152803.

[2] I. Shabani, E. Mëziu, B. Berisha, T. Biba, Design of modern distributed systems based on microservices architecture, International Journal of Advanced Computer Science and Applications 12(2) (2021) 153–159, http://dx.doi.org/10.14569/IJACSA.2021.0120220.

[3] D. Oktaria, J. A. M. Ginting, M. Abdurohman, R. Yasirandi, Design of API Gateway as middleware on Platform as a Service, Indonesia Journal on Computing (Indo-JC) 6(3) (2021) 47–62. https://doi.org/10.34818/INDOJC.2021.6.3.597.

[4] A. Ćatović, N. Buzađija, S. Lemes, Microservice development using RabbitMQ message broker, Science, Engineering and Technology 2(1) (2022) 30–37, https://doi.org/10.54327/set2022/v2.i1.19.

[5] A. Messina, R. Rizzo, P. Storniolo, M. Tripiciano, A. Urso, The database-is-the-service pattern for microservice architectures, Information Technology in Bio-and Medical Informatics: 7th International Conference, ITBAM 2016, Porto, Portugal, September 5-8, 2016, Proceedings 7 (2016) 223–233, Springer International Publishing, https://doi.org/10.1007/978-3-319-43949-5_18.

[6] Postman documentation, https://learning.postman.com/docs, [17.06.2025]

[7] cAdvisor documentation, https://github.com/google/cadvisor, [17.06.2025]

[8] Prometheus documentation, https://prometheus.io/docs, [17.06.2025]

[9] AWS Pricing Calculator, https://calculator.aws, [17.06.2025]

[10] Docker documentation, https://docs.docker.com, [17.06.2025]

[11] Grafana documentation, https://grafana.com/docs, [17.06.2025]