International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 02 , February - 2026

# Comparative Study of Scaling Monolithic and Microservices

Vedant Dinesh Mhamankar
Student, Department of Information Technology, Chikitsak Samuha's SS & LS Patkar College of Science and art & VP Varde College of Commerce, Mumbai

Aamina Imran Piyarji
Student, Department of Information Technology, Chikitsak Samuha's SS & LS Patkar College of Science and art & VP Varde College of Commerce, Mumbai

**Abstract - This paper presents a comparative study of scaling monolithic and microservices architectures. The primary objective is to analyze how modern applications can be made scalable and reliable under increasing workloads. The study first explains the foundational concepts of monolithic and microservices architectures, highlighting how each structure handles scalability, performance, and system complexity. It then explores essential system design principles such as load balancing, database scaling, caching, containerization, and distributed communication that impact scalability in both architectures. This research provides a clear understanding of how each architecture can be optimized for high-performance applications. The findings aim to guide developers and organizations in selecting the right architectural approach based on scalability requirements and system growth.**

**Index Terms - System Design, Microservices and Monolithic Architecture.**

## I. INTRODUCTION

In today's digital world, the online presence of businesses such as providing services and selling products through web and mobile applications is growing rapidly. As more users rely on digital platforms, it has become essential for businesses to develop robust web or mobile applications supported by a strong system design. A poorly designed system can lead to performance degradation, service disruptions, and loss of customer trust.

To support business growth and ensure uninterrupted services, systems must be designed to scale efficiently. **Scalability** refers to the ability of a system to handle an increasing workload by adding resources. For example, a business that currently serves 100,000 users may grow to serve 1,000,000 or more users in the future. In such scenarios, the application must be capable of managing increased traffic, data volume, and user requests without compromising performance.

System scalability is largely influenced by the underlying software architecture and the scaling strategies adopted during system design. Two widely used architectural approaches are **Monolithic Architecture** and **Microservices Architecture**. In a monolithic architecture, the entire application is developed as a single, tightly coupled unit in which all components share the same codebase and resources. Although this approach is simpler to develop and deploy initially, it often encounters limitations related to scalability and maintainability as the application grows.

In contrast, **Microservices Architecture** structures the system as a collection of small, independent services, each responsible for a specific business function and typically maintaining its own database. This architectural style enhances flexibility, fault isolation, and scalability, making it suitable for modern large-scale applications.

Scalability can be achieved using two primary techniques: **vertical scaling** and **horizontal scaling**. Vertical scaling involves increasing the capacity of existing hardware resources, while horizontal scaling focuses on distributing the workload across multiple service instances. In the initial stages of system growth, investing heavily in vertical scaling can be risky due to high costs, hardware limitations, and the introduction of single points of failure.

To address increasing user demand in a cost-effective and reliable manner, modern systems increasingly adopt **horizontal scaling**, which improves fault tolerance and supports incremental growth without excessive upfront investment. Consequently, architectural choices such as monolithic and microservices-based systems must be evaluated based on how effectively they support horizontal scaling under increasing workloads.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 02 , February - 2026

This research paper presents a **comparative study of horizontal scaling in monolithic and microservices architectures**, analysing their scalability characteristics, advantages, and limitations to determine their suitability for growing applications.

## II. LITERATURE SURVEY

The ongoing evolution of software architecture from monolithic systems to microservices is primarily driven by the need for increased scalability, flexibility, and maintainability in modern digital environments. Monolithic architectures, characterized by a single codebase and unified process, are often praised for their simplicity in deployment and initial development but face significant challenges as applications grow in complexity and traffic. Research by Elgheriani and Ahmed (2022) highlights that while monoliths are easier to test and have lower initial hosting costs, they suffer from tight coupling, making them vulnerable to single points of failure that can crash the entire system. Conversely, microservices decompose applications into autonomous, loosely coupled services that can be developed and scaled independently using technology heterogeneity. However, comparative studies by Al Debagy and Martinek (2018) reveal performance trade-offs, such as a 6% better throughput in monolithic systems during concurrency testing due to the lack of inter-service communication overhead. While previous literature has largely focused on general performance comparisons or AI-driven migration strategies, your research, "Comparative Study of Scaling Monolithic and Microservices," uniquely bridges these gaps by providing a deep technical analysis of specific infrastructure components. By exploring essential system design principles like NGINX Ingress, Kubernetes pod management, and Redis caching, your study offers practical insights into optimizing both architectures for high-performance, real-world scaling scenarios.

## III. METHODS

To scale microservices architecture there are multiple methods with different combinations of tech stack. Below common methods used to scale microservices and monolithic architecture.

1. **Load balancing :** Load balancer is network device or software that distributes incoming traffic load to multiple backend server to improve performance, reliability and uptime.
2. **Caching :** Caching refers to storing frequently used data or dealt-up data in temporary high-speed storage to reduce latency of system.
3. **Docker and containerization :** Docker and containerization enable application to be packaged with all their dependencies into light weight, portable containers. Containers ensure consistency across development, testing, and production environments. Docker simplifies building, deploying, and scaling applications by isolating services. This approach improves resource utilization, portability and deployment speed.
4. **Message Queue :** Message queues enable Asynchronous communication between system components.
5. **API gateway :** API gateway is a single entry point for all client requests in a microservice architecture.
6. **Database Sharding and Replication :** Database sharding is a database scaling technique where data is partitioned across multiple servers or databases. Each database is known as shard.
   Database replication is the process of copying and maintain database data across multiple servers or locations. It ensures there are multiple replicas of the same data.

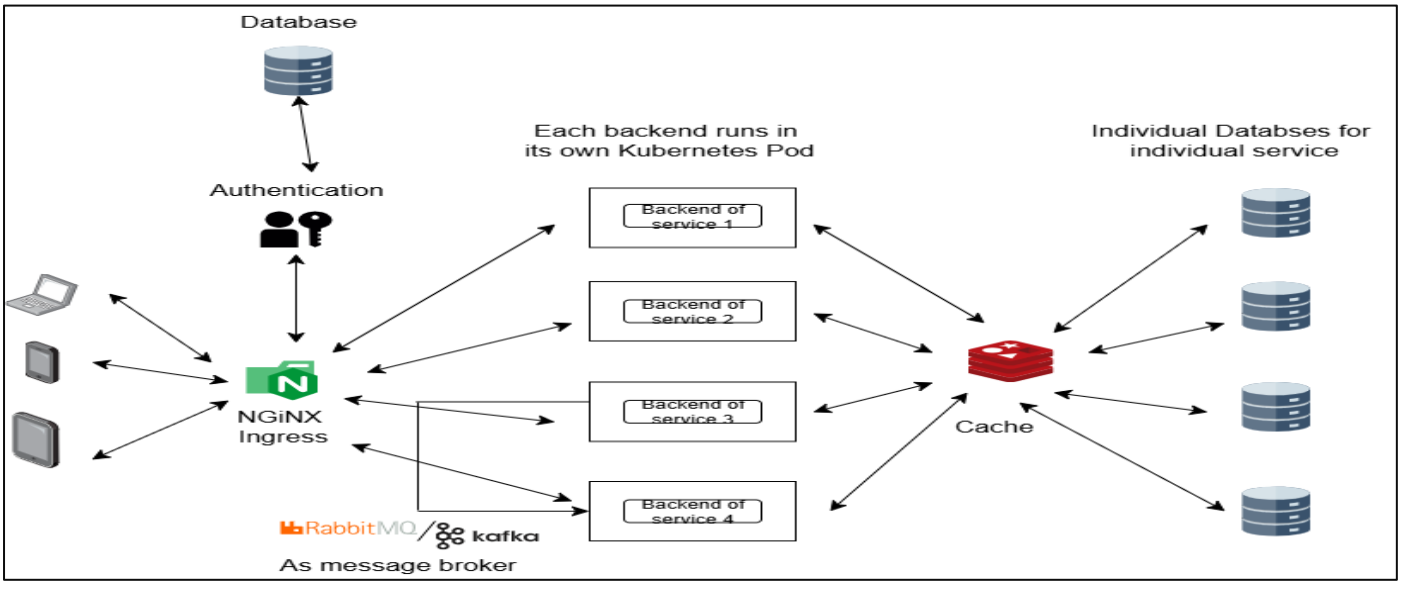


*Figure 1 : Microservices Architecture*

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 02 , February - 2026

The system works by first receiving all client requests from devices such as mobile phones, tablets, or computers through **NGINX Ingress**, which acts as a single entry point to the application. NGINX handles request routing, load balancing, and security concerns such as SSL termination. Before a request is forwarded to any backend service, it is validated by the **Authentication component**, which verifies user credentials or tokens using its own database. This centralized authentication mechanism improves security and avoids duplication of authentication logic across services.

Once authenticated, the request is routed by NGINX to the appropriate **backend service**, where each service runs independently in its own **Kubernetes pod**. This design ensures fault isolation and allows each service to be deployed and scaled independently based on workload. Backend services interact with a **shared cache (Redis)** to quickly retrieve frequently accessed data, reducing database load and improving response time. Each service also maintains its own **individual database**, which enforces loose coupling, enables independent schema changes, and prevents direct data dependency between services.

For operations that do not require immediate responses, services communicate asynchronously using a **message broker such as RabbitMQ or Kafka**. This event-driven communication model improves system scalability and resilience by decoupling services and preventing cascading failures. Kubernetes continuously monitors all running pods, automatically restarting failed services and scaling them horizontally when traffic increases. Together, these components create a scalable, resilient, and maintainable microservices-based system architecture.

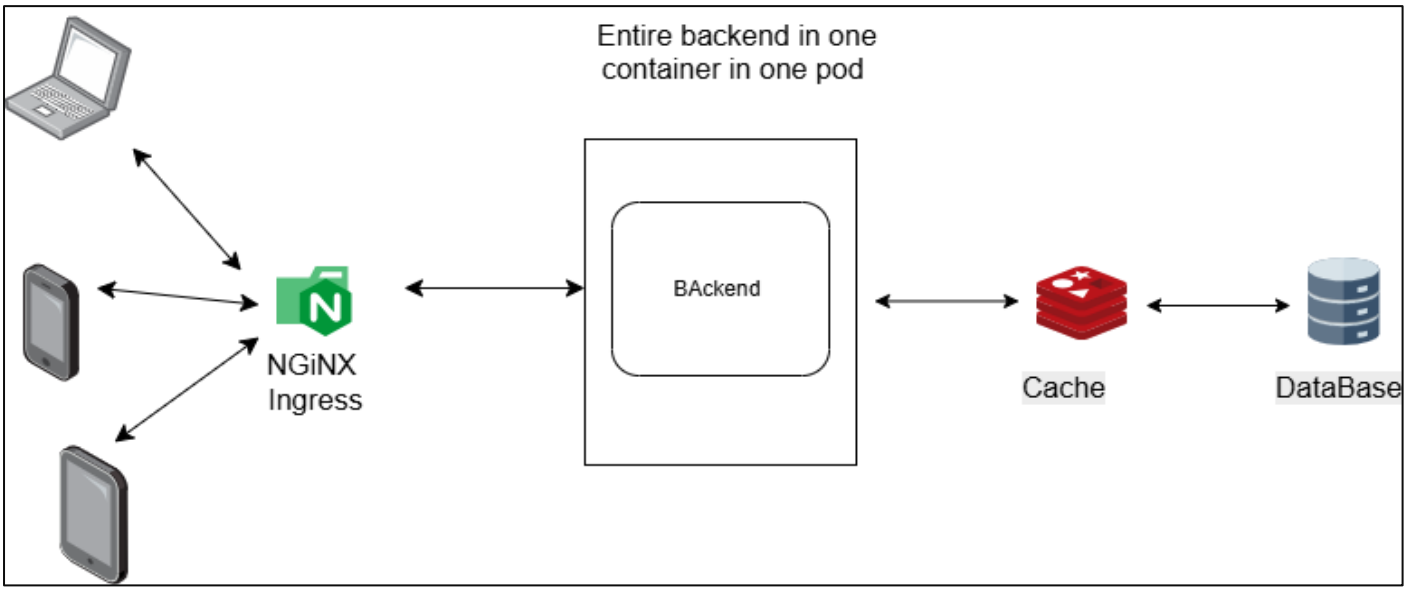


*Figure 2 : Monolithic Architecture*

In this monolithic system architecture, all client requests originating from devices such as laptops, mobile phones, and tablets are first directed to **NGINX Ingress**, which acts as the single entry point responsible for request routing, basic load balancing, and security functions such as SSL termination. NGINX forwards these requests to a **single backend application** that contains all business logic, including authentication, user management, and other functional modules, bundled together in one codebase and deployed as **one container inside a single Kubernetes pod**. Within the backend, internal modules communicate through in-process function calls rather than network calls, which simplifies development and reduces latency. To improve performance and minimize direct database access, the backend interacts with a **cache layer (such as Redis)** to store frequently accessed or session-related data, while all persistent data is stored in a **single shared database** accessed by the entire application. Although this architecture is easier to develop, test, and deploy due to its simplicity and centralized design, it also means that any change, failure, or scaling requirement affects the entire system, as the whole application must be redeployed or scaled together rather than individual components independently.

These are two example of both architecture how it could be design. Each one had its own Advantages and Disadvantages.

**Microservices Architecture**

**Advantages**

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 02 , February - 2026

- Independent scaling of each service improves resource efficiency.
- Failure in one service does not affect the entire system.
- Services can be developed and deployed independently.
- Supports use of different technologies for different services.
- Better suited for large, high-traffic applications.

**Disadvantages**

- System design and management are complex.
- Requires advanced infrastructure and DevOps expertise.
- Network communication adds latency.
- Data consistency across services is difficult to manage.
- Higher initial setup and operational cost.

**Monolithic Architecture**

**Advantages**

- Simple architecture that is easy to understand and implement.
- Faster development and testing due to a single codebase.
- Lower infrastructure and maintenance cost.
- Better performance for internal module communication.
- Ideal for small teams and early-stage projects.

**Disadvantages**

- Entire application must be scaled together.
- Failure in one module can crash the whole system.
- Any change requires redeploying the entire application.
- Difficult to maintain as the application grows large.
- Technology flexibility is limited.

## IV. RESULTS

The results of this comparative study show that the microservices architecture offers better scalability, fault isolation, and resource utilization than the monolithic architecture when system load increases. Microservices allow individual services to be scaled independently, which leads to improved performance and reduced response time under high traffic, while the monolithic system requires scaling the entire application, resulting in higher resource consumption. Fault tolerance is also stronger in microservices, as failures are limited to specific services without impacting the whole system, whereas a single failure in the monolithic architecture can cause complete service downtime. Additionally, deployment and maintenance are more flexible in microservices because services can be updated independently with minimal disruption, while monolithic systems require full redeployment even for small changes. However, the results also indicate that monolithic architecture has lower operational complexity and is easier to manage for small-scale or low-traffic applications.

## V. CONCLUSION

The conclusion of this research is that effective system design depends on a clear understanding of application requirements, workload characteristics, and the selected technology stack. While designing a system, it is essential to consider whether the backend

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 02 , February - 2026

services are event-driven or request-driven, and whether the system follows a stateless or stateful architecture, as these factors directly influence scalability, performance, and maintainability. Each system must be designed according to its specific use case, traffic pattern, and available resources, rather than following a single architectural approach. The comparative analysis demonstrates that transitioning between monolithic and microservices architectures is feasible as system requirements evolve. An architectural solution that performs well for one system may not be suitable for another, highlighting that architectural decisions should be context-driven rather than universally applied.

## VI. REFERENCES

[1] Al-Debagy, O., & Martinek, P. (2018). A Comparative Review of Microservices and Monolithic Architectures. 2018 18th IEEE International Symposium on Computational Intelligence and Informatics (CINTI), 149–154.

[2] Elgheriani, N. S., & Ahmed, N. A. S. (2022). Microservices vs. Monolithic Architectures: The Differential Structure Between Two Architectures. Minar International Journal of Applied Sciences and Technology, 4(3), 484–493.

[3] Hassan, H., Abdel-Fattah, M. A., & Mohamed, W. (2024). Migrating from Monolithic to Microservice Architectures: A Systematic Literature Review. International Journal of Advanced Computer Science and Applications (IJACSA), 15(10), 104–112.