Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2021, 8(8):123-128

Research Article ISSN: 2394 - 658X

Monolith to Microservices: Challenges, Best Practices, and Future Perspectives

Venkata Baladari

Software Developer, Newark, Delaware, USA
vrssp.baladari@gmail.com

ABSTRACT

The adoption of microservices architecture has significantly impacted software development, offering benefits such as enhanced scalability, increased flexibility, and accelerated deployment, although it also brings about issues including intricate communication complexities, security vulnerabilities, and elevated operational burdens. This study examines the shift from monolithic to microservices architecture, focusing on significant obstacles and effective methods including API gateways, containerization, Continuous Integration and Continuous Deployment (CI/CD) pipelines, and event-driven architectures. Technologies like AI-driven automation, serverless computing, and edge computing are anticipated to boost performance, reduce costs, and facilitate real-time processing. Research in self-healing systems, sustainable cloud computing, and multi-cloud approaches will enhance microservices in the future. To maximize the advantages of microservices, organizations should transition their systems gradually, implement automation, and prioritize innovation in order to develop robust, secure, and forward-thinking applications.

Keywords: Monolithic, Microservices, Monitoring, Architecture, Kubernetes

---

INTRODUCTION

The underlying structure of software architecture determines how applications are created, implemented, released, and updated. Historically, software systems utilized a monolithic framework, where all components, including the user interface, business rules, and data management, were closely integrated within a single source code and deployed as a single unit. Monolithic architectures, although offering simplicity in development and deployment, present substantial difficulties as applications grow in size.

- Limited scalability – Scaling monoliths can be challenging due to the need to replicate the entire application rather than individual components when dealing with high workloads.
- Slow development cycles – A solitary alteration to the system necessitates comprehensive testing and deployment, thereby hindering the pace of feature rollouts.
- Tightly coupled components – Modifications to a single component of a system can have a ripple effect on other interconnected modules, thereby heightening the likelihood of a comprehensive system collapse.
- Deployment inefficiencies – Making a minor adjustment could necessitate re-deploying the application in its entirety, resulting in system downtime and elevated operational expenses.

A modular and decentralized approach to application design has become increasingly popular as a way to address the limitations of traditional microservices architecture. In a microservices architecture, a large application is divided into separate, self-contained services that exchange data through application programming interfaces (APIs). Each service functions independently, enabling teams to build, release, and expand individual components on their own. The architectural progression offers a multitude of advantages, such as:

- Improved scalability – Services can be scaled independently based on varying levels of demand.
- Faster development cycles – Multiple teams can collaborate on distinct services, thereby speeding up the deployment of new features.
- Resilience and fault isolation – Incidents in a single service do not necessarily have a cascading effect on the entire system.

- Enhanced DevOps and Continuous Integration and Continuous Deployment (CI/CD) integration – Microservices facilitate smooth automation throughout the process of software development and deployment pipelines [1].

UNDERSTANDING MONOLITHIC AND MICROSERVICES ARCHITECTURES

Definition and Characteristics of Monolithic Architecture

Traditionally, a software design approach known as monolithic architecture involves developing, deploying, and maintaining an application as a solitary entity. This model integrates user interface, business logic, and database components into a unified, single executable unit. Initial development and deployment are streamlined in monolithic systems, but they become increasingly challenging to manage as applications expand in complexity. Modifying or scaling one component of the system frequently necessitates adjustments across the entire application, resulting in extended development periods, greater technical debt, and elevated maintenance expenditures [2].

A defining feature of a monolithic architecture is the centralized management of data, in which all components communicate with a solitary database. One of the drawbacks is that simplifying data management introduces a single vulnerability point, resulting in the loss of functionality if any component fails. Moreover, deployment inefficiencies are prevalent in monolithic applications due to the necessity of redeploying the entire system for even minor updates, thereby causing service disruptions and downtime. Despite the drawbacks, monolithic architectures continue to be beneficial for smaller to medium-sized applications that do not demand high scalability or regular standalone upgrades [2].

Definition and Core Principles of Microservices Architecture

This modern software design pattern decomposes complex applications into individual, loosely connected services that exchange data through Application Programming Interfaces (API). In contrast to monolithic systems, which integrate all functionality into a single component, microservices break down an application into multiple, independent modules, each serving a distinct business purpose. These services can be developed, deployed, and scaled separately, empowering organizations to increase agility, enhance fault resilience, and maximize performance [2][3].

These principles form the foundation upon which microservices operate. According to the Single Responsibility Principle (SRP), each microservice is designed to handle a distinct task, for instance, authentication, payment processing, or order management. This modularity enables various development teams to work on distinct services concurrently, thereby decreasing bottlenecks and facilitating quicker development cycles. A core concept is standalone deployment, which enables updates or bug corrections to be implemented on a single service without impacting the entire application. Microservices foster decentralized data management by enabling each service to utilize either an independent database or a jointly accessible data layer, thereby decreasing the likelihood of widespread system failures [3].

Microservices utilize DevOps and Continuous Integration and Continuous Deployment (CI/CD) practices in conjunction with API-driven communication to simplify and expedite development and deployment processes. Organizations that adopt microservices typically leverage technologies such as Docker and Kubernetes to containerize and orchestrate their systems, thereby facilitating seamless scalability and fault isolation. Microservices are commonly implemented in complex, cloud-based applications, such as e-commerce platforms, streaming services, and financial systems, for which high availability and performance are essential [1],[2].

Key Differences Between Monolithic and Microservices Models

The key distinctions between monolithic and microservices architectures are rooted in structural design, scalability, deployment methods, and operational adaptability. Monolithic architectures operate as a single, cohesive application, whereas microservices architectures comprise numerous, independently functioning services that collaborate to achieve a common goal. In a monolithic system, each component is heavily interdependent, so even a small update necessitates comprehensive testing and redeployment of the entire software program. In contrast, microservices offer loose connections, enabling separate updates and alterations without impacting other services.

Scalability: Typically, monolithic applications rely on vertical scaling, which entails augmenting a single server with additional resources, such as CPU and RAM, to manage heightened traffic. This approach has inherent limitations that hinder its ability to expand beyond a certain capacity. In contrast, microservices facilitate horizontal scaling, allowing companies to scale each service independently according to demand. This enables microservices to operate efficiently when dealing with dynamic workloads and to optimize resource utilization. [2],[3]

Flexibility: Monolithic applications necessitate a complete system overhaul for every update, resulting in extended deployment periods and possible system downtime. In contrast, microservices enable incremental deployments, allowing developers to introduce new features or resolve issues for single services without impacting the entire application. Organizations utilizing continuous delivery and agile development approaches can significantly benefit from this.

Data Management: Monolithic designs rely on a solitary, centralized database, resulting in streamlined data coherence but inherent performance constraints. In contrast to traditional architectures, microservices often employ

a decentralized data approach, where each service independently controls its own data storage and management. Reducing data dependencies, improving fault isolation, and increasing query efficiency is particularly beneficial in distributed systems. Implementing multiple databases can lead to data consistency issues, necessitating the careful adoption of event-driven architecture and synchronization methods for databases.

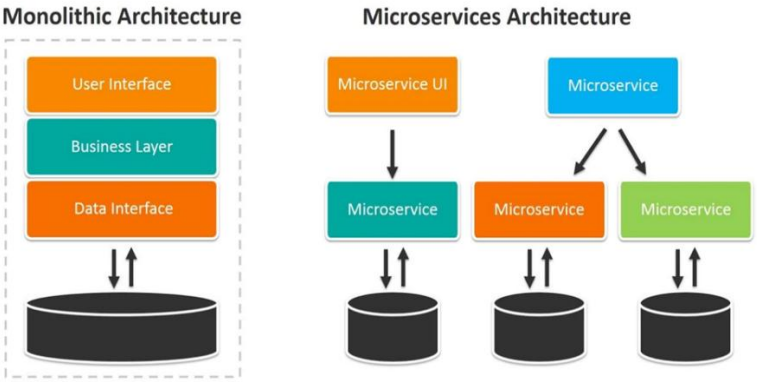

Figure 1: Monolithic vs Microservice Architecture
(Accessed from https://www.suse.com/c/rancher_blog/microservices-vs-monolithic-architectures/)

CHALLENGES IN TRANSITIONING FROM MONOLITHIC TO MICROSERVICES

Architectural Complexity and Design Considerations

Transitioning to microservices presents one of the most substantial challenges, specifically the heightened architectural complexity. In a monolithic system, all components are housed within a single codebase, which simplifies development, debugging, and deployment processes. In a microservices setup, an application is fragmented into several standalone services, each handling a distinct task and necessitating its own database, deployment process, and communication protocol. Service decomposition can be a challenge for organizations which can result in interconnected dependencies that undermine the benefits of microservices architecture.

Efficient inter-service communication is vital, as thoughtless interface design can lead to high network latency, performance limitations, and a plethora of inter-service connections. Selecting the appropriate communication protocols, such as REST APIs, GraphQL, or messaging platforms like Kafka and RabbitMQ, is crucial for ensuring optimal efficiency. The risk of developing a distributed monolith, where individual microservices are overly reliant on one another, must be minimized through the implementation of proper domain-driven design (DDD) and event-driven architectures [4],[5],[6],[7].

Data Management and Database Decentralization

Unlike traditional monolithic applications, which rely on a single, centralized database to manage all components, microservices employ a decentralized data strategy, where each service maintains its own data storage. The introduction of this system poses difficulties with maintaining consistent data, synchronizing it, and managing distributed transactions. Maintaining data consistency across various services can be challenging because conventional ACID (Atomicity, Consistency, Isolation, Durability) transactions do not function efficiently in distributed settings. To manage intricate business transactions, organizations must rely on event-driven data synchronization, eventual consistency or adopt strategies such as the Saga pattern [8].

Maintaining multiple standalone databases can result in redundant data, duplicate entries, and higher storage expenses. Businesses must decide on the most effective approach for designing data models, ensuring that data is accessible quickly and securely across multiple services. Choosing the suitable database technology, whether it's SQL, NoSQL, or a combination of both, is equally important, as various services may necessitate distinct database models to achieve peak performance [9].

Figure 2: SQL vs NoSQL
(Accessed from https://www.michalbialecki.com/2018/03/16/relational-vs-non-relational-databases/)

Deployment, Monitoring, and Debugging Difficulties

Building a monolithic application is relatively uncomplicated, as all its components are packaged together within a single, self-contained executable or container. In microservices, each service is deployed separately, necessitating more intricate CI/CD pipelines and deployment methods [1]. Deploying applications across various environments, including development, staging, and production settings, can be complicated, particularly when it comes to maintaining compatibility between interconnected services.

Monitoring and debugging a distributed system presents a range of additional difficulties. In a single, unified application, logging and error monitoring are consolidated, thereby simplifying problem resolution. Unlike traditional systems, microservices produce dispersed log files across various services, which complicates efforts to track and resolve issues. Integrating centralized logging and monitoring tools for ensuring system observability.

Cost and Resource Implications

Microservices architecture offers improved scalability and resource efficiency, but it also results in increased operational expenses relative to monolithic systems. A monolithic application usually operates on one or a limited number of servers, simplifying the process of cost estimation. Implementing microservices necessitates the use of multiple containers, virtual machines, or cloud instances, thereby significantly raising infrastructure costs.

In addition, network costs escalate due to inter-service communication, and observability tools for logging, monitoring, and security necessitate further investments. Operating multiple continuous integration/continuous deployment pipelines necessitates greater expertise in DevOps, resulting in increased personnel expenses.

Organizations should implement cost-reduction strategies to minimize expenses, including the use of autoscaling, serverless computing for non-permanent tasks, and dynamic adjustments of cloud resources according to real-time requirements. Implementing infrastructure automation through Infrastructure-as-Code (IaC) can also facilitate the optimization of resource allocation and decrease operational overhead [10],[11].

BEST PRACTICES FOR MICROSERVICES ADOPTION

Defining a Clear Migration Strategy

A successful shift to microservices necessitates a clearly outlined plan for migration to mitigate potential risks and guarantee a seamless transition from a single unified system. Organizations should begin by examining their current infrastructure, pinpointing areas of congestion, capacity constraints, and interdependent modules that must be decomposed into standalone services. The migration process should be implemented incrementally, with services being phased in over time, rather than undergoing a full-scale transformation all at once. The Strangler Fig Pattern is a successful approach that entails creating new microservices in parallel with the current monolithic system and systematically replacing the monolith's functionalities until the entire transition is finished.

Defining the service boundaries with Domain-Driven Design (DDD) is key to guaranteeing that microservices are loosely linked and independently expandable [7]. Companies should also invest in automation for deployment, testing, and scaling to simplify the migration process. A standardized governance model needs to be put in place to ensure consistency in API development, security protocols, and monitoring techniques. Organizations can prevent disruptions and achieve a successful switch by implementing a gradual migration approach to a microservices environment, thereby minimizing dependencies.

Database per Microservice and Event-Driven Architecture

Unlike monolithic applications that rely on a single, centralized database, microservices necessitate a decentralized data framework, wherein each service keeps its own database to guarantee operational independence. The Database Per Service methodology prevents data congestion, enhances fault compartmentalization, and enables independent scalability. This system also poses challenges in terms of maintaining data consistency and synchronizing it.

For effective management of distributed data, organizations should establish an event-driven architecture, in which microservices exchange information asynchronously via events, rather than through direct requests. Apache Kafka, RabbitMQ, and Amazon SQS can act as message brokers to enable real-time event-driven data exchange [4],[6],[12]. The Saga Pattern can also be employed to synchronize distributed transactions across multiple services while ensuring the integrity of the data. The CQRS (Command Query Responsibility Segregation) model can enhance performance by isolating read and write operations, thus minimizing database contention. Businesses can boost data adaptability and ensure data accuracy by implementing decentralized databases and event-driven messaging systems, thereby increasing the reliability of their microservices [13].

CI/CD Pipeline Implementation for Efficient Deployment

A comprehensive Continuous Integration and Continuous Deployment (CI/CD) pipeline is crucial for facilitating quick, trustworthy, and automated releases in microservices settings. Unlike traditional, single-unit applications, where updates are rolled out as a single entity, microservices necessitate separate deployments for each individual service, making automated Continuous Integration/Continuous Deployment processes a requirement [1].

Implementing continuous integration and deployment pipelines with Jenkins, GitHub Actions, GitLab CI/CD, or CircleCI is recommended to automate code integration, testing, and deployment processes [14]. Incorporating automated testing methods, including unit tests, integration tests, and contract tests, is crucial to preventing failures

when making updates. Strategies for deployment such as blue-green deployments and canary releases enable gradual updates, thereby decreasing the duration of outages and mitigating potential hazards. Feature flags can be used to dynamically turn features on or off without requiring redeployments of services. By setting up continuous integration and continuous delivery pipelines, companies can experience shorter release times, increase software reliability, and boost system dependability.

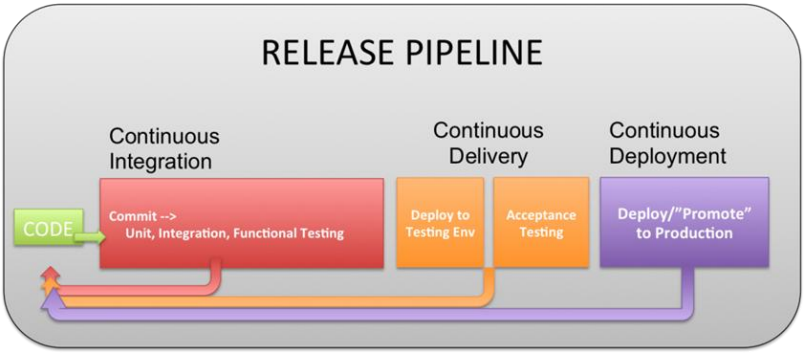

Figure 3: CI/CD Pipeline
(Accessed from https://devops.com/i-want-to-do-continuous-deployment/)

FUTURE PERSPECTIVES OF MICROSERVICES ARCHITECTURE

Integration with Serverless Computing

A pivotal advancement in microservices architecture is the incorporation of serverless computing, enabling applications to operate without the requirement for manual infrastructure administration. Typical microservices deployments necessitate containerized environments like Docker and Kubernetes to manage workloads, scale resources, and facilitate networking. With the emergence of serverless platforms like AWS Lambda, Google Cloud Functions, and Azure Functions, organizations are now able to run microservices functions as needed, thereby obviating the necessity of having to set up and manage servers [12],[15].

The main benefit of serverless microservices is lower cost due to the dynamic allocation of resources in response to actual demand rather than pre-allocated computing capacity. This method also streamlines deployment and expansion, with serverless functions automatically adapting to variations in workload. Future progress in this area is expected to concentrate on lowering cold start latency, augmenting containerized microservices integration, and bolstering multi-cloud functionality. Serverless computing is not a suitable approach for every situation, but its combination with microservices can transform the way cloud-native applications are developed, enabling companies to construct more streamlined, expandable, and budget-friendly systems.

AI and Machine Learning in Microservices Optimization

Artificial intelligence and machine learning are increasingly vital components in enhancing the efficiency of microservices-based systems. As microservices architectures continue to grow in complexity, the manual management of performance, scaling, security, and fault tolerance becomes increasingly difficult. AI-driven solutions can alleviate these challenges by automating scaling decisions, forecasting potential failures, and optimizing resource usage in real-time [16].

AI is being integrated into microservices primarily through auto-scaling optimization. Conventional auto-scaling techniques rely on fixed benchmark values, including CPU or memory consumption. AI-powered models can foresee traffic surges and scale microservices ahead of time to prevent performance bottlenecks. Furthermore, AI-powered anomaly detection systems can examine system logs, identify uncharacteristic patterns, and trigger preventive measures to avert system failures [16].

CONCLUSION

The shift from monolithic to microservices architecture has improved software's scalability, flexibility, and efficiency, but it also introduces difficulties such as intricate communication, heightened security risks, and increased operational burdens. To facilitate a seamless transition, companies should decompose large, complex systems into smaller, standalone components over time. Effective practices utilizes API gateways and service meshes for intercommunication, containerizing using Docker and Kubernetes for scalability purposes, and adopting an event-driven architecture to enhance data management capabilities. Implementing Continuous Integration and Continuous Deployment pipelines automates the deployment process, leading to increased efficiency and productivity. Companies should also consider serverless computing to decrease expenses and invest in DevOps training to boost team skills.

Future advancements in AI, serverless computing, and edge computing are expected to enhance the intelligence and automation of microservices. Artificial intelligence can aid in forecasting failures, enhancing performance and

concurrently edge computing will diminish latency for real-time applications. Research into self-healing systems, sustainable cloud computing, and multi-cloud deployment will continue to enhance the capabilities of microservices. To remain competitive, companies must continually innovate, adopt automation, and adjust to emerging technologies. In today's software development landscape transformed by microservices, careful planning and adherence to best practices enable companies to develop strong, scalable, and forward-thinking applications.

REFERENCES

[1]. A. Agarwal, S. Gupta, and T. Choudhury, "Continuous and Integrated Software Development using DevOps," 2018 International Conference on Advances in Computing and Communication Engineering (ICACCE), Paris, France, 2018, pp. 290-293, doi: 10.1109/ICACCE.2018.8458052.

[2]. M. Villamizar et al., "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," 2015 10th Computing Colombian Conference (10CCC), Bogota, Colombia, 2015, pp. 583-590, doi: 10.1109/ColumbianCC.2015.7333476.

[3]. M. Kalske, N. Mäkitalo, and T. Mikkonen, "Challenges When Moving from Monolith to Microservice Architecture," in Current Trends in Web Engineering. ICWE 2017, I. Garrigós and M. Wimmer, Eds. Cham: Springer, 2018, vol. 10544, Lecture Notes in Computer Science. doi: 10.1007/978-3-319-74433-9_3.

[4]. X. J. Hong, H. Sik Yang and Y. H. Kim, "Performance Analysis of RESTful API and RabbitMQ for Microservice Web Application," 2018 International Conference on Information and Communication Technology Convergence (ICTC), Jeju, Korea (South), 2018, pp. 257-259, doi: 10.1109/ICTC.2018.8539409.

[5]. S. K. Mukhiya, F. Rabbi, V. K. I. Pun, A. Rutle, and Y. Lamo, "A GraphQL approach to Healthcare Information Exchange with HL7 FHIR," Procedia Computer Science, vol. 160, pp. 338-345, 2019. doi: 10.1016/j.procs.2019.11.082.

[6]. B. Leang, S. Ean, G.-A. Ryu, and K.-H. Yoo, "Improvement of Kafka streaming using partition and multi-threading in big data environment," Sensors, vol. 19, no. 1, p. 134, 2019. doi: 10.3390/s19010134.

[7]. R. Steinegger, P. Giessler, B. Hippchen, and S. Abeck, "Overview of a domain-driven design approach to build microservice-based applications," in Proc. 3rd Int. Conf. Adv. Trends Softw. Eng. (SOFTENG), Venice, Italy, Apr. 2017.

[8]. Q. Abbas, H. Shafiq, I. Ahmad and S. Tharanidharan, "Concurrency control in distributed database system," 2016 International Conference on Computer Communication and Informatics (ICCCI), Coimbatore, India, 2016, pp. 1-4, doi: 10.1109/ICCCI.2016.7479987.

[9]. W. Ajayi, "SQL '+' NoSQL, A Merger with Great Capabilities," Int. J. Comput. Trends Technol. (IJCTT), vol. 62, no. 1, pp. 14, Aug. 2018. [Online]. Available: http://www.ijcttjournal.org.

[10]. Kim D, Muhammad H, Kim E, Helal S, Lee C. TOSCA-Based and Federation-Aware Cloud Orchestration for Kubernetes Container Platform. Applied Sciences. 2019; 9(1):191. https://doi.org/10.3390/app9010191

[11]. M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero and D. A. Tamburri, "DevOps: Introducing Infrastructure-as-Code," 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), Buenos Aires, Argentina, 2017, pp. 497-498, doi: 10.1109/ICSE-C.2017.162.

[12]. R. Buyya, C. Vecchiola, and S. T. Selvi, "Cloud platforms in industry," in Mastering Cloud Computing, R. Buyya, C. Vecchiola, and S. T. Selvi, Eds. Morgan Kaufmann, 2013, pp. 315–351. doi: 10.1016/B978-0-12-411454-8.00009-7.

[13]. M. Overeem, M. Spoor and S. Jansen, "The dark side of event sourcing: Managing data conversion," 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Klagenfurt, Austria, 2017, pp. 193-204, doi: 10.1109/SANER.2017.7884621.

[14]. V. Ivanov, Implementation of DevOps Pipeline for Serverless Applications, Master's thesis, School of Science, Aalto University, Espoo, Finland, May 27, 2018.

[15]. L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," in Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18), Boston, MA, USA, Jul. 2018.

[16]. A. Smiti, "When machine learning meets medical world: Current status and future challenges," Comput. Sci. Rev., vol. 37, p. 100280, 2020. doi: 10.1016/j.cosrev.2020.100280.