Scalable Microservices Architectures: Building Resilient Distributed Systems for the Modern Web

Article · August 2024

DOI: 10.56472/25838628/IJACT-V2I3P109

2 authors:

Vijaya Pothuri
Indian Institute of Technology Bombay
14 PUBLICATIONS **7** CITATIONS

SEE PROFILE

Rama Koppula
Salesforce.com
9 PUBLICATIONS **8** CITATIONS

SEE PROFILE

ESP-IJACT
ESP International Journal of Advancements in Computational Technology
ISSN: 2583-8628 / Volume 2 Issue 3 August 2024 / Page No: 101-109
Paper Id: IJACT-V2I3P109 / Doi: 10.56472/25838628/IJACT-V2I3P109

Original Article

Scalable Microservices Architectures: Building Resilient Distributed Systems for the Modern Web

Vijaya R Varma Pothuri[1], Rama C Koppula[2]
[1,2]*Salesforce Inc., 415 Mission St, San Francisco, CA, USA.*

Received Date: 14 June 2024 Revised Date: 16 July 2024 Accepted Date: 13 August 2024

Abstract: *The advancement of technologies on the web and the constant need for modern applications in today's world made the architects go with the microservices architecture. This architectural style is about decomposing monumental applications into a set of microservices that can be developed, deployed, and scaled separately. Microservices architecture is very beneficial for different SOAs, as it provides better scalability, fault tolerance, and maintainability. However, it is also associated with problems like inter-service communication, consistently replicated data and service discovery. The objective of this paper is to offer a detailed handbook on achieving horizontally scalable and fault-tolerant microservices systems. We start by presenting the main ideas of microservices, including decomposition, data ownership, and fault tolerance. Having established the architectural layers in the cloud foundry platform, we expand into the architectural components: the service registry, the API gateway, the load balancer and the service mesh, and we look at some of the tools that enable the implementation of these components; Docker, Kubernetes, Istio and Jenkins, among others.*

Keywords: *Microservices, Scalability, Resilience, Distributed Systems, API Gateway, Service Mesh, Container Orchestration, Observability.*

I. INTRODUCTION

The advancement in technology, particularly in the use of the internet has played a major factor in changing the business environment. Since users demand speed, reliability, and scalability of services, modern complex applications and solutions cannot be based on monolithic architectures. [1,2] Monolithic applications are linked and are usually closely coupled, which present the following problems: scalability issues, poor modularity and maintainability, and failure of a single component can halt the entire project.

Static websites and monolith implementations have thus turned out to be hard to deal with, mostly because of these challenges. This architectural style focuses on breaking down a large application into a set of numerous and relatively independent components that are loosely connected, each of which performs only a small part of the whole application's functionality. These services are based on well-defined APIs, and it is possible to develop, deploy and scale the service differently.

A. Problem Statement:

However, just like any other architectural style, there are several challenges associated with microservices architecture. There can be many issues, such as complicated inter-service communication, data consistency across services, and having proper service discovery mechanisms. Also, avoiding over doctrines or failures and expanding the services for survivors efficiently are some important issues.

B. What Are Microservices:

Microservices architecture is a mode of designing the application in which a large application is developed as a sum of multiple small applications that can be deployed independently of each other. [3] Services are delivered as individual business functions and utilize clear interfaces for communicating with other services, and each service can be created, deployed, and extended separately.

a) Benefits of Microservices:

- Scalability: Microservices also help in scaling the individual components differently based on their requirements to make efficient and performant use of resources.
- Resilience: It should be noted that through the division of failures within certain services, microservices make the overall system more actively reliable.
- Flexibility: The update of services can be done by teams with little intervention from other parts, and services can be deployed and developed faster.

- Technology Diversity: With the help of microservices, it becomes possible to employ the technologies and programming languages of the teams' choice for the development of a particular service.

b) Challenges of Microservices:

- Complexity: Microservices' distribution raises the architectural concern level and puts a significant demand on management and coordination.
- Data Management: The existence of distributed data storage is the main reason there is often some difficulty in guaranteeing the principle of data consistency and data integrity across services.
- Network Latency: Inter-service communication creates network delay across the system and predictably creates overall latency in the given system.

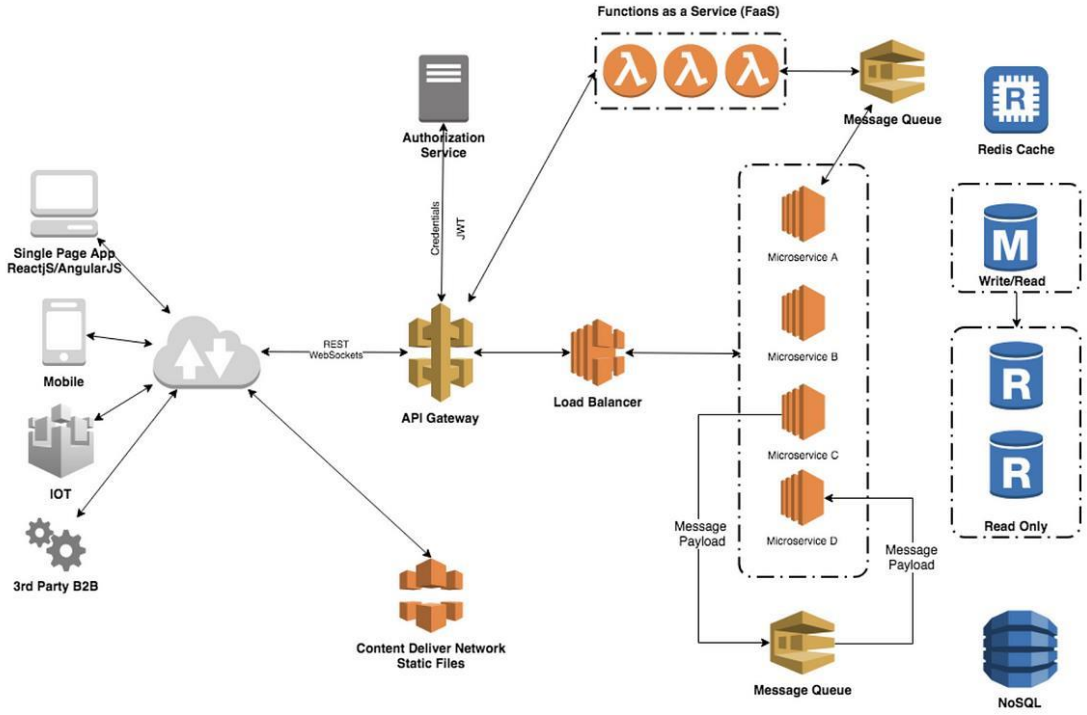

Figure 1: Example of a Scalable Web App Design

This Example of a scalable web app design depicts a contemporary cloud-based architecture suitable for dealing with different kinds of client applications and services employing the microservices paradigm. [6,7] the system comprises several key components:

Numerous clients, such as Single-Page Applications built with ReactJS or AngularJS frameworks or iOS-Android applications, IoT appliances, and B2B third-party providers, are accommodated by the architecture. Such clients have a regular interaction with the system through REST or WebSocket protocols thus providing flexibility in interaction. Various clients guarantee that the architecture can support specific kinds of clients and their interaction with the system, as well as with each other.

As one of the scopes of the application, the API Gateway plays the role of the initial contact point for the requests from the client. It handles the issues of authentication and authorization through the authorization service and validates credentials with JWTs. The API Gateway authenticates the requests and then forwards the calls to the right services, thus acting as a traffic cop in making sure that all the requests are delivered to the right locations. This setup not only protects the system but also classes and improves the client's interfaces with the backend services.

Following the requests, the Load Balancer comes into the picture to direct the traffic to other microservices availed through the API Gateway. This distribution guarantees that one service does not become overloaded and that the overall system is highly available. This way, the load is distributed equally, and the system remains fast and responsive even during the most popular durations.

At the core of architecture there are several microservices which serve as the carriers of the different business logic operations. These microservices interact with each other via message queues, and this means that the processing can be

done independently and as such, the various services can be easily changed in a way that does not affect the other one. This microservices-based approach also has the advantage that any maintenance, update, or scaling can be completed on the individual service.

Also, some functions are delivered in the form of serverless computing or Microservices as Functions as a Service (FaaS). These serve-fewer functions are event-driven and communicate with the microservices using message brokers. Thus, the use of serverless functions through this approach is advantageous since the functions are only initiated when necessary, and the design is kept relatively simple, hence lowering the costs of operations.

The two types of solutions used for data management in the architecture are caching and database solutions. Redis Cache is used for Read/Write operations which hold frequently fetched data for fast retrieval of data and boosts performance. For persistent storage, a NoSQL database is used, and some of the instances they set up are read-only to be more efficient in handling large numbers of reads. Thus, caching and DB solutions guarantee fast data access and, at the same time, provide reliable data storage.

Last but not least, the static files are deployed using a Content Delivery Network so that end-users can get their content quickly. On the same note, through the reduction of the number of requests, the CDN improves the performance and response time thus providing optimum content delivery to the user irrespective of his geographical region. This broad architecture incorporates the use of several technologies and techniques that help in offering a flexible, secure and effective system that is capable of fulfilling several needs of clients.

II. LITERATURE SURVEY

A. Microservices Architecture:

A comprehensive literature review identifies the existing condition of microservices architecture, fault tolerance in distributed systems, and scalability. [8-10] This survey focuses on the advantages and issues of the microservices architecture and discusses the newest developments and approaches to it. The architecture of microservices is that there is the use of individual small services that function as one complete application. Services are individually targeted at certain business capabilities and can be built, used, and tuned separately. This mode of decoupling of services increases the flexibility of the system because the development teams who are faced with the implementation of these services can use the most suitable technologies and development paradigms.

B. Key Characteristics of Microservices:

- **Autonomy:** All the services are modular in that they can be deployed and managed independently as well as when updates are required, they can be done individually. This makes it possible to make changes in the iterations and modify the deployment process based on high levels of detail. Different services are delivered independently and do not affect the whole application, making it possible to provide continuous deliveries by updating teams for services.
- **Isolation:** Perhaps there is no cascading effect for one service to cause issues for other services. They must achieve isolation so that the other services in the large system will not be affected if some of them fail. Since every microservice contains both the logic and data of a specific service, architectures in this Mold reduce the effects of such failures and make the debugging and overall maintenance a much easier task.
- **Scalability:** It is easy to scale services as a way of dealing with increased load as compared to other approaches. High availability can also be achieved through scaling, while in monolithic architectures, scaling means copying the application completely; microservices architecture makes it possible to scale only the parts that require scaling. This results in effective utilization of infrastructure resources and logistic ability to perform well when many clients put pressure on the system.

C. Resilience in Distributed Systems:

Basically, resilience is the process which revolves around the capacity of a system to face and overcome failures. In distributed systems, reliability is obtained through replication, backup, and techniques like circuit braking, retrying and fallbacks. [11,12] These techniques make it possible for a system to remain working even if some parts of the system are not functioning well or even if they are non-functional still the system will be able to work.

a) Techniques for Achieving Resilience:

- Circuit Breakers: Circuit breakers stop the cascade by stopping calls to a failing service. If the service fails, the circuit breaker is tripped; this, in turn, does not try to call the service until it is back online. This mechanism guards the system from being flooded with failure notifications and carries on with the other services even when there is a failure in a couple of services.

- Retries and Timeouts: Logically, each failed operation should attempt the operation again while limiting the time one waits for an operation to complete is also crucial in the architecture for resilience. Retries enable the system to redial in cases of temporary problems, while timeouts ensure that services do not wait for a response for a long and exhaust the system's resources, thus lowering the performance.
- Fallbacks: Fallbacks work as an option in case a service does not work or cannot be used. For instance, if the primary service is not accessible, the system can reply from the cache or utilize the backup service. This guarantees that the user does not have a feel of the other parts of the system that might be troubled.

D. Scalability:

Scalability is the third characteristic of a portable system, which states the way in which a system can increase its capacity when the load is increased. [13] In a microservices architecture, scalability is done through horizontally, where extra instances of services are added as loads increase. This is made possible through container orchestration tools such as Kubernetes, which assist in extending and organizing apps in the containers.

a) Types of Scalabilities:

- Vertical Scaling: Vertical scaling entails the increase in the quantity of resources in the same tier, for example, the increase of CPU or memory in a server. There are pros and advantages to this, and it can also have cons and disadvantages that restrict the scalability of such applications.
- Horizontal Scaling: Horizontal scaling deals with the addition of more instances of a service to help reduce the load. This approach is more conducive to attaining high scalability because the system can accommodate a larger throughput by instantiating more services in other servers. Tools like Kubernetes manage containers and their instances for their deployment, scaling and load balancing which helps in optimum resource usage and high availability.

E. Tools and Technologies:

Various tools and technologies facilitate the implementation of microservices architecture. Some of the key tools include:

- Containers: Docker
- Orchestration: Kubernetes
- Service Mesh: Istio
- API Gateway: Kong
- CI/CD: Jenkins
- Monitoring and Logging: Prometheus, Grafana, ELK Stack

Table 1: Comparison of Tools and Technologies

Tool	Purpose	Key Features
Docker	Containerization	Lightweight, portable, consistent environment
Kubernetes	Orchestration	Automated deployment, scaling, and management
Istio	Service Mesh	Traffic management, security, observability
Kong	API Gateway	Load balancing, authentication, rate limiting
Jenkins	CI/CD	Automated build, test, and deployment
Prometheus	Monitoring	Metrics collection, alerting
Grafana	Visualization	Dashboards, data visualization
ELK Stack	Logging and Monitoring	Log collection, analysis, and visualization

F. Challenges in Microservices Architecture:

Despite its benefits, microservices architecture presents several challenges:

- Inter-Service Communication: Coordination of the communication between services can become challenging particularly where there is the use of both synchronous and asynchronous communications.
- Data Consistency: This type of transaction implementation must be carefully designed and conducted in services to maintain data consistency.
- Service Discovery: To support services' self-organization, these services should be able to find and inform other services on their availability, thus requiring reliable mechanisms of service discovery.
- Security: Ensuring service communication and managing the authentication and authorization of services are important.

III. BUILDING RESILIENT DISTRIBUTED SYSTEMS

The Building Resilient Distributed Systems diagram [14,15] shows that chaos engineering is a sub-discipline of software engineering where faults are injected into a system deliberately to understand the system's behavior. Thus, the general aim is to find potential vulnerabilities that have not emerged in the production environment yet. Here is a detailed explanation of the steps involved:

The term 'steady state', when used, discusses system reliability and robustness and simply depicts the steady state where everything is in order. This is the state that is most often met, which corresponds to the leading, or benchmark, level of performance and steadiness.

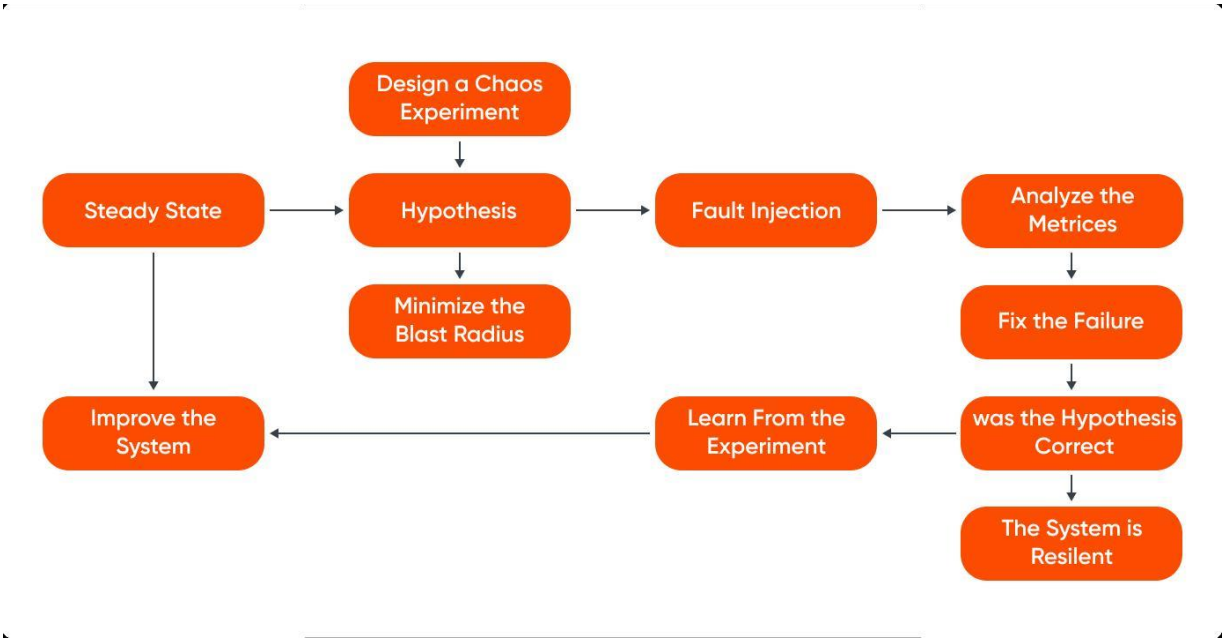

Figure 2: Building Resilient Distributed Systems

A chaos experiment, therefore, focuses on the design of working models whose performance is deliberately brought down to conditions close to failure or stress. The objective of this experiment is to assess the performance of the system in compromised circumstances. Prior to implementing the experiment, a hypothesis is made to estimate the system's behavior in the presence of faults. Regarding the notion of hypothesis, this would form the basis of checking whether the effects produced by the introduced faults tally with the expected response of the designed system.

In order to not harm the experiences of the vast majority of users, it is necessary to keep the blast radius of the chaos experiment as small as possible. This means controlling the expansion of the experiment to the system in a proportion that minimally involves the failing conditions. Fault injection or mock failure involves the act of deliberately creating faults in the system, and this may entail shutting down servers or slowing them down, partitioning a network or even causing other problems that may be considered to be faulty.

This means that after the faults are introduced, one has to look into the metrics and record the corresponding values together with any data that will have to be used when evaluating the performance of the system. This includes measures like error, response times, and system behavior in relation to how well the model is performing in terms of the real application. If gaps or loss-making activities are identified in the process of the experiment, then certain actions have to be taken to eliminate these failures or strengthen their indicators. This could mean alteration of the code or configuration in some manner to correct the issues that were pinpointed.

Lastly, it is assessed if the predicted effects were achieved to conclude whether the hypothesis from the experiment tallied with the hypothesis preceding the implementation of the experiment. This step is important in order to either accept or reject the hypothesis that had been established concerning the system. It is advisable to collect thorough information and documentation when getting to the experiment in order to understand the stability of the system in detail. Thus, if the system operates correctly in the presence of the above-defined faults with minor problems, then the system is a resilient system.

Last but not least, in any case with the hypothesis, the results obtained during the experiment should be utilized to enhance the established system. This is done to adapt to newer faults since fault tolerance is the key aim of the system; making changes leads to an increase in the reliability of the system.

IV. METHODOLOGY

A. System Design:

The objective of microservices for organizing a large volume of applications is to initiate a large volume of services, build a resilient system, and properly manage a large volume of data and networks, [16] new architectures need a scalable service architecture, data management strategies, an API gateway, service mesh integration, and container orchestration. Thus, logging, monitoring, and tracing become the main approaches to ensure the health and performance of the developed system.

B. Service Decomposition:

Bounded context services are the services which are deployed microservices and are also known as service decomposition in the art of application evolution. Splitting an application into multiple microservices where each service is developed, deployed, and scaled independently is due to the fact that every microservice deals with a single business capability. The key steps in service decomposition include: the key steps in service decomposition include:

- Identifying Boundaries: Design each microservices based on business capabilities and follow domain-driven design practices.
- Defining Interfaces: Define clear interfaces through which the microservices can communicate with one another.
- Decoupling Services: Separate services to be as independent as possible so as little interdependence as possible exists between them.
- Independent Deployment: Evolve each service independently so that there can be frequent deployment and incorporation.

a) Techniques for Service Decomposition:

- Domain-Driven Design (DDD): Subsequently, it presents an identification of bounded contexts and the subsequent definition of the microservices based on the guidelines of DDD.
- Event Storming: A driving force to engage multiple stakeholders to investigate the domain and help service decomposition.

C. Communication Mechanisms:

There is usually a need for communication between the microservices in the system for efficiency and dependability. [17] Different means of communication can be used based on the needs and limitations of the application.

a) Synchronous Communication:

- REST: A well-known architectural style in designing Network Applications focusing on the Stateless behavior of communication and Resource-oriented interactions.
- gRPC: It is an open-source application layer framework with a focus on high performance. The transport layer is HTTP/2, and the interface description language is Protocol Buffers.

b) Asynchronous Communication:

- Message Queues: Decouple services so they do not depend on each other through the creation of queues for better coping with increasing load.
- Event Streams: Collaborate on real-time integration to assist in the processing of data in the event-driven systems.

D. Orchestration and Deployment:

Orchestration implies the overall life cycle of the microservices, deploying, scaling, and even fixing the problems. Hence, popular instruments like Kubernetes offer highly useful prerogatives to orchestrate microservices.

a) Containerization:

- Docker: A platform for developing, shipping, and running applications in containers, ensuring consistency across environments.
- Kubernetes: An open-source tool that analyzes and schedules the deployment of containerized applications.

E. Data Management:

Within the context of microservices, any given service is usually responsible for its own database, which makes for modularity and scalability. [18] One of the tough cases is to maintain the data in the distributed services to be consistent.

Other practices like the Saga pattern can be adopted in which the transactions are divided into local transactions and actions of compensation.

Table 2: Data Management Strategies

Strategy	Description	Advantages	Disadvantages
Shared Database	All services use a common database	Simple, easy to implement	Tight coupling, scalability issues
Database per Service	Each service has its own database	Loose coupling, independent scaling	Complexity in managing data consistency

F. Implementation:

a) API Gateway:

API Gateway serves as a single point of entry for the client's Application requests and directs them towards the proper microservices. It provides several critical functionalities:

- Load Balancing: Splits incoming requests coming to microservices to multiple instances of the microservice for availability and reliability.
- Authentication and Authorization: Guards the microservices using proper authentication and authorization of users.
- Rate Limiting: Regulates the number of requests in order to avoid exceeding the number of requests that services can handle.

b) Service Mesh:

A service mesh like Istio manages inter-service communication, providing advanced features such as a service mesh like Istio manages inter-service communication, providing advanced features such as:

- Traffic Management: Oversees the movement of traffic between microservices and allows functionality inclusive of feature toggling and feature flagging.
- Security: Comprises and implements end-to-end encryption and Ill- graded access control between the services.
- Observability: Provides useful information regarding service communication such as the metrics, logging and tracing.

c) Container Orchestration:

Kubernetes provides container orchestration which is the process of managing containers into a larger entity and can automate the process of deploying, scaling and managing IT applications that are shipped in containers. Key features include:

- Automated Rollouts and Rollbacks: Enables successful update strategy and rollbacks of the microservices.
- Service Discovery: It is self-sufficient and works to find the other microservices that it needs.
- Load Balancing: Transmits and directs the networking traffic with the intention of high availability.

d) Observability:

Strong observability is, therefore, significant in managing the wellness and reliability of a microservices structure. Key components include:

- Logging: Gathers log data of all the microservices for problem-solving and analyzing the system's activity.
- Monitoring: It employs tools like Prometheus for microservices health and performance checks.
- Tracing: Use of distributed tracing, such as Jaeger, for following the requests as they go through the system.

G. Case Study: E-Commerce Platform Transition:

As for the proof of concept for our concept, the case study involves an article-sharing e-commerce platform to show how it adopted the microservices architecture from monolithic architecture. It led to gains in the efficiency, extensibility, and fault tolerance of the system that had to be accomplished during the transition.

a) Transition Process:

- Initial Assessment: Analyzed the current monolithic system to recognize the problematic areas and areas that require changes and improvements.
- Service Decomposition: Initially decomposed the monolith application into smaller microservices in order to achieve the following functionalities, namely, users' management, product services, order services and payment services microservices, respectively.
- Database Management: Applied per-service database approach where each microservice uses its own database, allowing them to be independent and scale according to the need.
- API Gateway Implementation: Implemented an API Gateway to address all the requests coming from the clients, which typically include features such as request balancing and rate limiting.

- Service Mesh Integration: While Dodgester has managed the inter-service communication, Istio has been integrated to ensure the microservices' security, traffic and observability are well maintained.
- Container Orchestration with Kubernetes: Used Kubernetes to deploy microservices which allows for automation of the deployment of applications in containers.
- Observability Enhancements: For monitoring, the solutions included Prometheus, for visualization – Grafana and for logging – the ELK Stack.

b) Performance Metrics:

The transition to a microservices architecture resulted in the following performance improvements:

- Scalability: A remarkable boost in the website's capability, concerning the total of user connections at one and the same time, as well as the total of transactions.
- Resilience: Fault tolerance and minimized downtimes because of the microservices architecture that is precisely decentralized.
- Performance: Better performance and optimized resource utilization of the relevant server by scaling it independently of the rest of the architecture.

c) Architectural Decisions Impact:

- Service Decomposition: Increased efficiency in the development process and also reduced complexity of the codes.
- Database per Service: Lowered contention and provided autonomous scaling; however, it consisted of a set of challenges with the synchronization of data.
- API Gateway: Reduced client interactions thus had crucial elements such as authenticating and rate limiting centralized.
- Service Mesh: Making inter-service communication smooth with added features and depolarizing the management of microservices.
- Container Orchestration: Automation of deployment and scaling of an application with less time and effort, taking up the overhead.

V. RESULTS AND DISCUSSION

A. Results:

In contemporary web systems construction, a pattern of construction with the scale of microservice has become an influential practice of construction of systemic diverse architectures. The chief outcome derived from microservices is the improvement of scalability and flexibility in the applications. Each microservice operates independently, which enables it to be built, released, and adjusted based on the application's requirements, improving resource utilization. This architectural style also encourages an approach to delivering and deploying new and modified software in small installs frequently, and without adversely affecting the whole system. Furthermore, the loose coupling patterns in the services allow failure in one service not to affect other services, thus improving the robustness of the system.

B. Discussion:

The use of microservices architectures averts the problems observed above and greatly enhances the flexibility of the development teams. Breaking down an application into smaller services/sub-services or applications enables different teams within the organization to work on different areas and sub-systems of the application at the same time and hence hasten the development process. Also, microservices do not restrain the utilization of several technologies and languages because the teams decide on it for every service. Having multiple paradigms in play also improves the state and may contribute to generating more efficient solutions. Yet, the modular services and effective componentization also create the issue of service management, monitoring, and communication, especially requiring sound and proven practices and platforms for the implementation and support of the ecosystem in question.

VI. CONCLUSION

There is a major transition from the monolithic structures and microservices architectures being highly scalable are helpful in many ways as compared to monolithic structures. The modularity of the microservices speaks to the continuous delivery and the cause of the fault, which is important in creating and implementing highly dependable distributed systems. Nevertheless, there is no doubt that flexibility, independent scaling and the use of technologies and programming languages provided by microservices are highly beneficial for modern web applications. The implementation is highly dependent on DevOps principles, and the understanding of how the services are communicated among themselves; the ability to design and implement solid solutions that are maintainable and adaptive makes it worthwhile.

VII. REFERENCES

[1] Hasselbring, W., & Steinacker, G. (2017, April). Microservice architectures for scalability, agility and reliability in e-commerce. In 2017 IEEE International Conference on Software Architecture Workshops (ICSAW) (pp. 243-246). IEEE..

[2] Salah, T., Zemerly, M. J., Yeun, C. Y., Al-Qutayri, M., & Al-Hammadi, Y. (2016, December). The evolution of distributed systems towards microservices architecture. In 2016 11th International Conference for Internet Technology and Secured Transactions (ICITST) (pp. 318-325). IEEE.

[3] Lewis, J., & Fowler, M. (2014). Microservices: A definition of this new architectural term. Martin Fowler website..

[4] Nginx Inc. (2020). Microservices Reference Architecture. Nginx.

[5] Red Hat. (2018). Building microservices with Istio and Kubernetes.

[6] Pahl, C., & Jamshidi, P. (2016). Microservices: A systematic mapping study. International Conference on Cloud Computing and Services Science.

[7] Scalable Web Architectures Concepts & Design, medium, online. https://medium.com/distributed-knowledge/scalable-web-architectures-concepts-design-6fd372ee4541

[8] Richardson, C. (2016). Microservices Patterns: With examples in Java. Manning Publications.

[9] Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media..

[10] This structure provides a comprehensive and detailed exploration of building scalable microservices architectures, incorporating multiple sub-headings, figures, tables, and flowcharts to elucidate key concepts and practices.

[11] Resiliency in Microservices: A Guide to Circuit Breaker Pattern, Medium, online. https://dip-mazumder.medium.com/best-practices-for-error-handling-a-guide-to-circuit-breaker-patterns-41d45ffc02ac

[12] Guide to Microservices Resilience Patterns, JRebel, online. https://www.jrebel.com/blog/microservices-resilience-patterns

[13] What is Scalability in Cloud Computing? Types, Benefits, and Practical Advice, Nops, online. https://www.nops.io/blog/cloud-scalability/

[14] How to Build Resilient Distributed Systems, axelerant, 2024. Online. https://www.axelerant.com/blog/how-to-build-resilient-distributed-systems

[15] Vemasani, P., & Modi, S. Building Resilient Distributed Systems: Fault-Tolerant Design Patterns for Stateful Workflows.

[16] Tyszberowicz, S., Heinrich, R., Liu, B., & Liu, Z. (2018). Identifying microservices using functional decomposition. In Dependable Software Engineering. Theories, Tools, and Applications: 4th International Symposium, SETTA 2018, Beijing, China, September 4-6, 2018, Proceedings 4 (pp. 50-65). Springer International Publishing.

[17] Asynchronous vs Synchronous Communication: Definitions, Differences, and Examples, Getguru, online. Link

[18] Laigner, R., Zhou, Y., Salles, M. A. V., Liu, Y., & Kalinowski, M. (2021). Data management in microservices: State of the practice, challenges, and research directions. arXiv preprint arXiv:2103.00170.

[19] Malavalli, D., & Sathappan, S. (2015, November). Scalable microservice based architecture for enabling DMTF profiles. In 2015 11th International Conference on Network and Service Management (CNSM) (pp. 428-432). IEEE.

[20] Del Esposte, A. M., Kon, F., Costa, F. M., & Lago, N. (2017, April). Interscity: A scalable microservice-based open source platform for smart cities. In International Conference on Smart Cities and Green ICT Systems (Vol. 2, pp. 35-46). SciTePress.