# Introduction to Microservices

## Introduction

Microservices are an architectural style that structures an application as a collection of loosely coupled and independently deployable services that implement business capabilities. The loosely coupled design helps in the minimization of design-time coupling and the minimization of runtime coupling. The coupling designs reduce the likelihood of two or more software elements changing together and minimize the availability of one service being dependent on the availability of another service, which in return makes an application easier to scale and faster to develop for any new features.

With Monolithic architectures, all the processes are tightly coupled, and they run as a single service. These are simple to develop, test, deploy, and scale horizontally. However, they only tend to work well in the preliminary stages of the project. Over time, when the application becomes large, it becomes overwhelmingly complex to maintain and identify the bug because of the multi-million line of code application. These raise complexities like slower start-up time and difficulty in the adoption of new frameworks and languages. Even scaling horizontally is costlier and cannot be afforded with large applications.

Microservices evolved as a solution to the scalability and reliability challenges with monolithic architectures. The term microservices itself was first coined by Dr. Peter Rodgers in 2005 and the main driver was to break up single large monolithic designs into multiple independent components/processes, thereby making the codebase more granular and manageable.

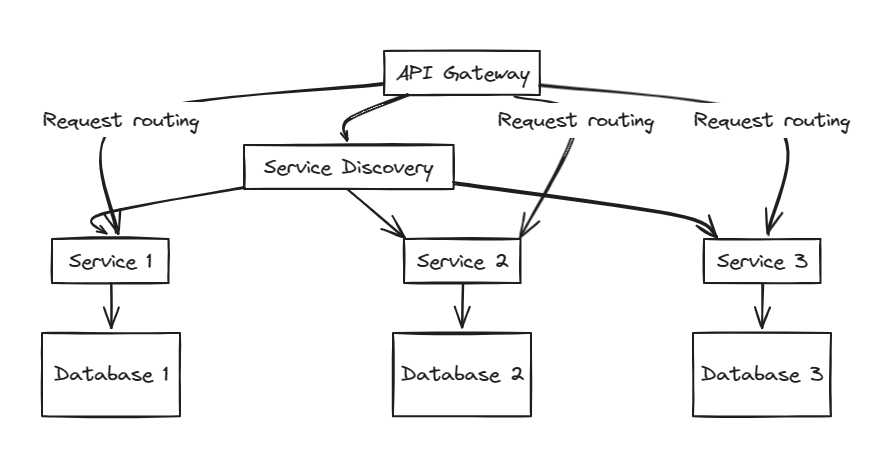


Fig: Microservices Architecture

The diagram above is a simple reference model for a microservice architecture. It illustrates a system comprised of several granular services that either communicate synchronously – via internal API calls or asynchronously – via messaging.

Unlike monolithic architecture, the application here is built as independent components and runs each of these application processes as a service. This architecture pattern enforces a level of modularity that is difficult to achieve with monolithic architectures. Since each service is independent in microservice architecture, redeployment of the entire application on each update is not necessary and fixes the slowed down start up time. Microservices allow for granular scalability, where specific services of the applications can be scaled based on demand which overcomes the horizontal scaling in monolithic architectures. Overall, microservices have gained popularity due to their ability to address the limitations of monolithic architectures, especially in large and complex systems where scalability, resilience, and agility are critical.

## Principles of Microservices

1. Decentralization:

Decentralization is a fundamental principle of microservices architecture that enables the distributing of responsibilities, decision-making authority, and control across individual services within a system. Since services are developed, deployed and managed independently, each service is responsible for its own data management, business logic, and persistence which benefits the system abundantly. It gives the team the freedom to choose the most appropriate technology stack, development methodologies and deployment for their services that boosts performance of each service. Independent services/ decentralized services also help in isolating failures to an individual service, which prevents them from affecting the entire system. Isolation of these failures helps the system by improving overall resilience and fault tolerance.

Decentralization in microservice architecture empowers the team to flexibility and scalability by distributing responsibilities and decision-making authorities within the services.

1. Componentization:

Microservices are treated as independent components that can be easily replaced or upgraded without affecting the rest of the systems. Componentization emphasizes breaking down an application into smaller, reusable, and independently deployable components. It involves designing software applications as a collection of loosely coupled components which helps in reduction of two or more software elements/services changing together due to same reason. Loosely coupled components also result in minimization of the availability of one service being dependent on the availability of any other service which decreases the response time. Components in a microservice are designed to be reusable across different parts of the application or even in other applications. Hence, it promotes code reuse, reduces duplication and simplifies maintenance.

In summary, componentization in microservices involves breaking down of application into smaller, reusable, and independently deployable components which promotes reusability, independent deployment and fosters flexibility and agility in software development.

1. Autonomy:

A key function of microservices is their autonomy and ability to operate independently of each other. Since they are loosely coupled, the service has complete control over its runtime environment and database schema. The teams have autonomy to work independently on each service, making decisions about technology stack, development methodologies, and deployment strategies separately. Since each microservice operates autonomously it allows them to enhance the performance and reliability of the service by making the right decisions according to the service requirements and needs. As the teams work independently on each service, it reduces the coordination overhead and contributes to the overall availability and scalability of the service. Each service can be developed, deployed, operated, and scaled without affecting the functionality of other services as services do not need to share any of their code and resources with other services. The communications between individuals happen via well-defined APIs which can be messaging or Rest or SOAP according to the requirement.

Due to autonomy behavior of microservice, the system is less prone to experiencing failure and performance issues due to another service. This boosts the flexibility, scalability and performance time of the service abruptly.

1. Technology Diversity:

Microservices architecture doesn’t follow “one size fits all” approach. This means that the teams have freedom to choose the best tool to solve their specific needs and don’t have to stick with one technology unlike in a monolith architecture. Hence, allowing the team to build microservices choosing the best tool for each job. Hence, diversity fosters innovation and experimentation to a better problem-solving approach. Managing the right technology diversity can lead to better performance, scalability, and maintainability of individual services. However, managing a diverse technology stack requires expertise in multiple technologies and can introduce complexities in terms of integration, monitoring, and maintenance.

## Advantages of Microservices:

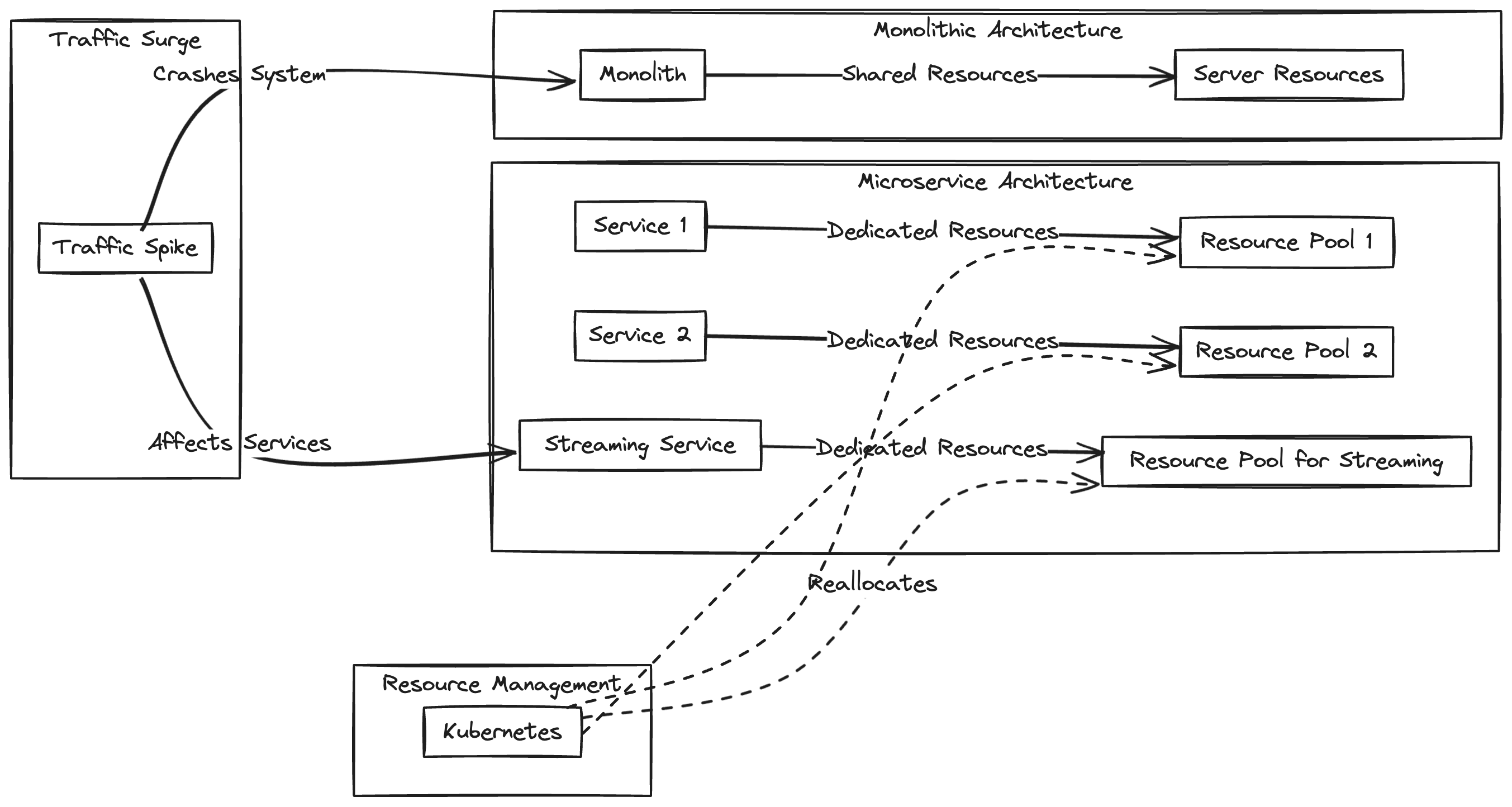
* Scalability: Services can be scaled independently, allowing for more efficient use of resources.
* Resilience: Faults in one service do not impact others, improving overall system robustness.
* Technological Agility: Allows the adoption of new technologies and processes without overhauling the entire system.

1. **Scalability**:

Scalability stands out as a cornerstone feature within microservice architecture. Let's examine the drawbacks of monolithic architecture for a moment. Imagine a scenario with a video streaming application; when a surge in traffic occurs, the entire system crashes. This issue stems from all components within the monolith sharing the same server resources, leading to an overload affecting all services.

However, in microservice setups, each service operates with its own dedicated resources. Consequently, during a spike in streaming service traffic, resources can be promptly augmented exclusively for streaming services, without any adverse impact on other services. This refined scalability not only averts outages but also ensures users consistently enjoy a seamless experience.

The implementation of orchestration tools like Kubernetes further enhances scalability by dynamically reallocating resources to services based on demand. Moreover, scalability streamlines the deployment of new features and updates without necessitating system-wide shutdowns. This streamlined deployment process significantly reduces both time and costs, while concurrently mitigating the risks associated with errors and downtime.



1. **Resilience**:

Microservices offer resilience by isolating issues within individual services, preventing cascading failures. Each service operates independently with its own configurations and resources, ensuring uninterrupted system operation even during peak usage. Features like Circuit Breakers mitigate the risk of server resource depletion due to failed services, maintaining system stability. Moreover, microservices' modular structure allows for easier fault isolation and faster recovery, minimizing service disruptions and enhancing overall system reliability.

1. **Technological Agility**:

Microservices adapt easily to evolving technologies, offering flexibility in programming languages and tools. Unlike monolithic architectures, microservices accommodate diverse project needs and team skill sets, facilitating quick adoption of new technologies. For instance, integrating an AI feature into a Java-based banking system using Python can be seamlessly achieved by creating a separate microservice component. This agility optimizes return on investment by leveraging existing resources and swiftly embracing new technologies. Moreover, microservices' compatibility with multiple languages simplifies integration with legacy systems. This adaptability ensures future-proofing and enables organizations to stay ahead in a rapidly changing technological landscape.

Challenges of Microservices:

**Complexity**: Increased operational and management complexity.

**Data Integrity**: Ensuring data consistency across services can be challenging.

**Network Issues:** Dependency on network latency and load balancing.

**Skill Set**: Requires a broad set of skills from development teams, including DevOps capabilities.

1. **Complexity**: Increased operational and management complexity.

Creating apps with microservices can get really complicated. One big challenge is setting up each service on its own, especially when you need to deploy them in different places without changing anything. Using centralized config services like Kubernetes configmap can make this whole thing a lot easier by handling all the configurations from one central place.

One important aspect of designing microservices is the thoughtful planning of how services will communicate with each other. This means carefully managing the way requests are handled and prioritizing business capabilities over data-centric perspectives. It can be difficult to determine the boundaries of each service and decide on the appropriate level of detail when faced with complex business needs.

Effective communication and coordination among microservices are essential for smooth business operations. Architects must proactively tackle these obstacles by setting up a clear framework for sharing key information, grasping the interconnections between microservices, and creating efficient boundaries to streamline design hurdles. Failing to establish a solid strategy can lead to complications and delays when breaking down business functions into microservices.

1. **Data Integrity**: Ensuring data consistency across services can be challenging.

When each service in a system runs independently and handles its own data stored in different places, having duplicate data becomes a problem. For example, information about a particular transaction saved in one service might also be copied in other services for tasks like analysis or saving records. This repetition can make it tough to keep data accurate and in sync throughout the system. Old ways of managing data might have trouble making sure all the pieces fit together when working with multiple services. Keeping the system in a stable condition usually needs extra steps, especially when transactions involve multiple microservices. Data changes are usually tracked through recorded business events, rather than directly accessing data stores. This means that to understand the current system state, one must go through the history of business events and determine the current state from there.

Moreover, when multiple microservices are connected to the same database tables, any changes to the database schema can have cascading effects on other microservices. This highlights the importance of each microservice having complete ownership and control over its required data. However, this doesn't necessarily mean each microservice needs its own physical database; rather, it emphasizes the need for each service to have autonomy over the data it manages. Different types of databases, including NoSQL, Graph, and in-memory solutions, may be necessary to effectively support the diverse data needs of microservices, as relying solely on a relational database may not yield optimal results.

1. **Network Issues:** Dependency on network latency and load balancing.

Microservices architecture relies heavily on network communication between services, which can lead to challenges related to network latency and load balancing.

For instance, imagine a shopping application where the product catalog service needs to communicate with the inventory service to check product availability. If there's high network latency between these services, it could slow down the response time for users trying to view products, leading to poor user experience.

Additionally, load balancing becomes crucial to distribute incoming requests evenly across multiple instances of a service to prevent overloading any single instance. Without effective load balancing, some instances may become overwhelmed with requests while others remain underutilized, impacting the overall performance of the system.

Addressing these network issues requires implementing strategies such as caching frequently accessed data, optimizing network communication protocols, and using efficient load balancing algorithms. By mitigating latency and ensuring proper load distribution, microservices can maintain responsiveness and scalability in the face of network challenges.

1. **Skill Set**: Requires a broad set of skills from development teams, including DevOps capabilities.

When it comes to managing microservices, having DevOps skills is crucial. These skills allow development and operations teams to work together smoothly to simplify the process of deploying and managing microservices. Practices like continuous integration (CI), continuous deployment (CD), infrastructure as code (IaC), and automated testing play a vital role in guaranteeing that microservices-based applications are reliable, scalable, and easy to maintain.

In addition, being skilled in containerization tools like Docker and container orchestration systems like Kubernetes is crucial for implementing and overseeing microservices on a large scale. Familiarity with cloud computing platforms like AWS, Azure, or Google Cloud is also advantageous, as they offer the resources and tools needed for creating and operating microservices in an efficient and cost-effective way.Moreover, having proficiency in tracking and observability tools such as Prometheus, Grafana, or ELK stack is crucial for overseeing the well-being and efficiency of microservices and identifying problems instantly. In general, a versatile set of skills covering development, operations, and DevOps techniques is important for effectively creating, executing, and managing applications based on microservices. Collaboration, continuous learning, and adaptability are essential qualities for development teams maneuvering through the intricacies of microservices architecture.