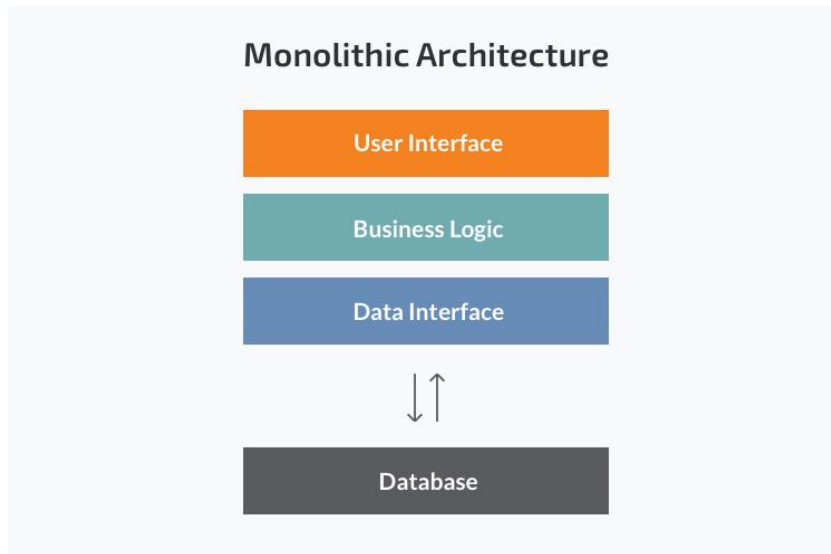


Monolith

The monolithic architecture is a traditional way of building applications. A monolithic application is built as a single and indivisible unit.



Strengths:

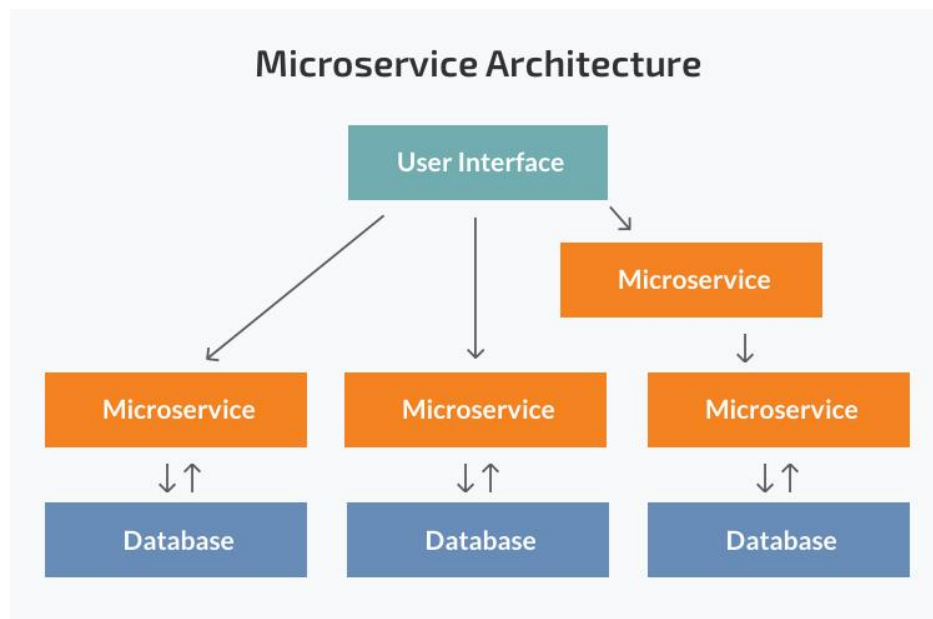
1. Easier debugging and testing
A monolithic app is a single indivisible unit, you can run end-to-end testing much faster.
2. Simple to deploy
you do not have to handle many deployments – just one file or directory.
3. Simple to develop
As long as the monolithic approach is a standard way of building applications.

Weakness:

- This simple approach has a limitation in size and complexity.
- Application is too large and complex to fully understand and made changes fast and correctly.
- The size of the application can slow down the start-up time.
- You must redeploy the entire application on each update.
- Impact of a change is usually not very well understood which leads to do extensive manual testing.
- Continuous deployment is difficult.
- Monolithic applications can also be difficult to scale when different modules have conflicting resource requirements.
- Another problem with monolithic applications is reliability. Bug in any module (e.g. memory leak) can potentially bring down the entire process.
- Monolithic applications has a barrier to adopting new technologies.

Microservices

A microservices architecture breaks it down into a collection of smaller independent units. These units carry out every application process as a separate service. So all the services have their own logic and the database as well as perform the specific functions.



Strenghts:

- Due to small – small modules it's easier to develop, understand and maintain the application.
- It enables each service to be developed independently by a team that is focused on that service.
- It reduces barrier of adopting new technologies since the developers are free to choose whatever technologies make sense for their service.

- Microservice architecture enables each microservice to be deployed independently. As a result, it makes continuous deployment possible for complex applications.
- Microservice architecture enables each service to be scaled independently.

Weakness:

- Microservices architecture is a bit complex compare to monolithic. Microservices application is distributed system. You need to choose and implement an inter-process communication mechanism.
- Microservices has the partitioned database architecture. Business transactions that update multiple business entities in a microservices-based application need to update multiple databases owned by different services.
- Testing a microservice based application is also much more complex compare to a monolithic web application.
- It is more difficult to implement changes that include multiple services.
- Deploying a microservices-based application is also more complex. In contrast, a microservice application typically consists of a large number of services.

Choosing a monolithic architecture

Small team, simple application, no microservice expert, quick launch

Choosing a microservices architecture

A large team, Microservices expertise, A complex and scalable application.

Service Mesh

A service mesh is an inter-service communication infrastructure.

- A given Microservice won't directly communicate with the other microservices.
- Rather all service-to-service communications will take place on-top of a software component called service mesh
- service developers can focus more on the business logic while most of the work related to network communication is offloaded to the service mesh.
- Load balancing
- Authentication
- Security

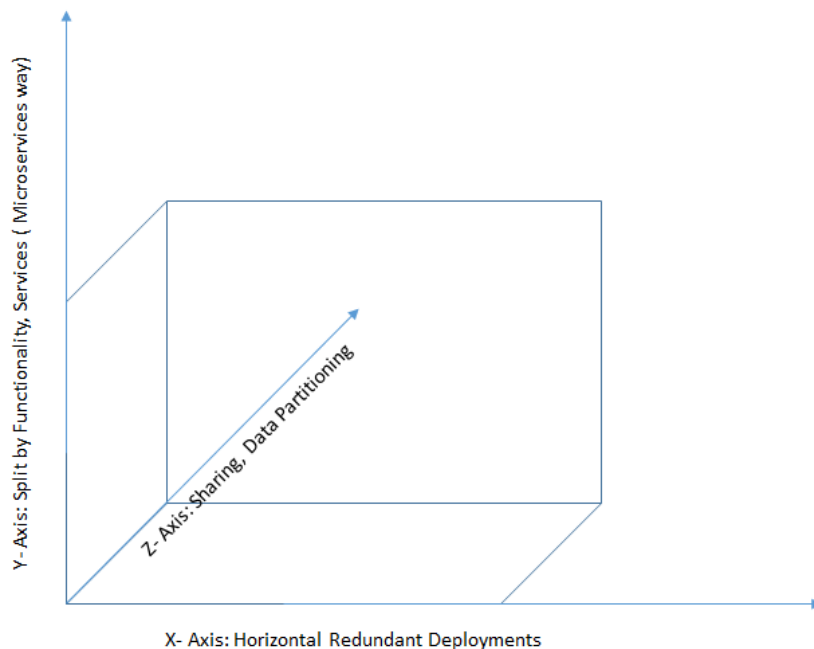
Why Is Service Mesh Necessary?

In a microservice architecture, handling service to service communication is challenging and most of the time we depend upon third-party libraries or components to provide functionalities like Hystrix for circuit breakers, Eureka for service discovery, Ribbon for load balancing which are popular and widely used by organizations.

Linkerd and Istio are two popular open source service mesh implementations. They both follow a similar architecture, but different implementation mechanisms.

Scale Cube

If cube is microservice application then,



X-axis

X-axis scaling consists of running multiple copies of an application behind a load balancer. If there are N copies then each copy handles $1/N$ of the load. This is a simple, commonly used approach of scaling an application.

Y-axis

Decomposing a microservice into two or more microservices based on the functional capabilities.

Z-axis

Z-axis splits are commonly used to scale databases. For an example we can take a user table which contains all the users. Now we are decomposing the table into two or more tables :

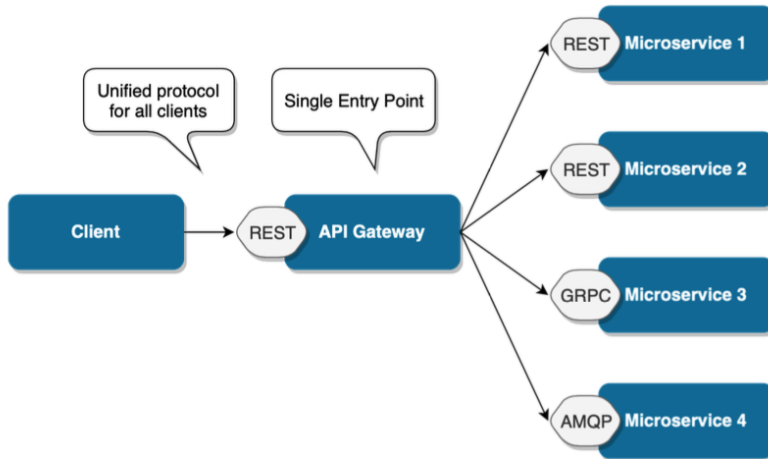
User_1 => A to J

User_2 => K to Q

User_3 => R to Z

API Gateway

Gateway



API Gateway Consists:

HTTPS => 1. HTTP 2. HTTP 3. HTTP etc.

Authentication, SSL termination, Load Balance, Security etc.

Service Discovery/Registry

Service discovery is a pattern to identify the network addresses of all the instances of the microservices.

Service registry contains list all of the service instances of all the microservices and its addresses (IP with port no.).

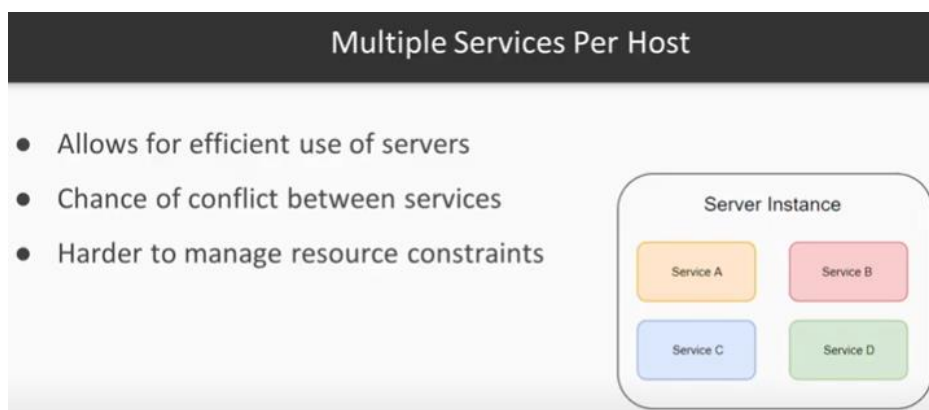
Circuit Breaking

One microservice communicates with other services via RPC call or REST call. There may be a situation that the other service is not responding at that time or hang without a response until some timeout limit is reached.

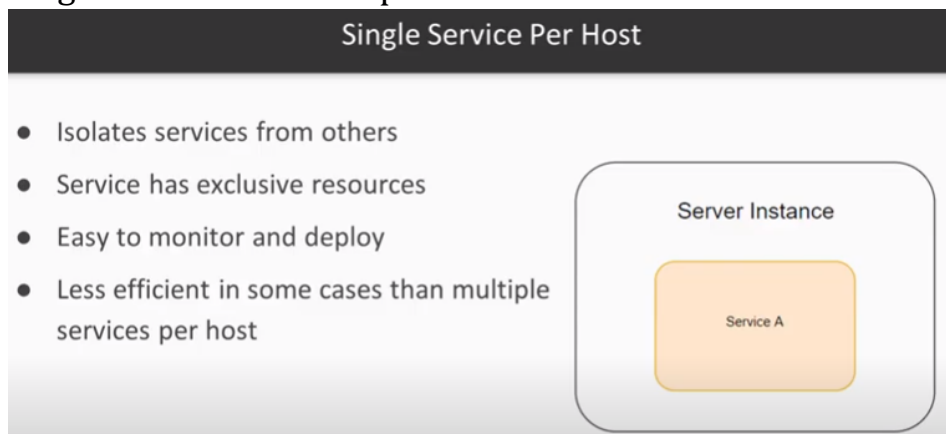
Once the failures reach a certain threshold, the circuit breaker trips, and all further calls to the circuit breaker return with an error or with some alternative service or default message. The circuit breaker has three distinct states: Closed, Open, and Half-Open.

Microservice Deployment Strategies

1. Multiple Service Instances per Host




2. Single Service Instance per Host



3. Service Instance per Container

Service Instance Per Container


- Package the service in a Docker container for deployment
- Scaling is easy by changing number of container instances
- Encapsulates the tech stack
- Isolates each service





4. Serverless Deployment

Serverless Deployment

- Eliminates the need to maintain servers
- Scales according to usage
- These need to be quick and lightweight
- Latency risk
- Pay per request


AWS Lambda


Microsoft Azure

 Google Cloud