Key Considerations While Shifting to Microservices

* **Degree of Independence** – The first thing to consider is the level of independence you want for the services in your microservices architecture. In the first approach, each service is completely independent with its own UI and database. This would be an instance of an extreme microservices architecture where services are entirely decoupled and share nothing. The difficulty arises in ensuring all datastores stay in sync and updated at all times. To rectify this, in the second approach, you can choose to share some components between the services, such as the database. This would make it easier for you to ensure data consistency and enforce data standards, regulations, and compliances within the software.
* **Technology Stack** – It’s hard enough to decide a tech stack for a monolithic application. Now, imagine doing that for each service within a microservices architecture. If your services are heterogeneous, this could create an issue with standardization. Moreover, it becomes harder for your people to move between teams if every team uses a different tech stack. A recommended approach is to take a balanced approach in deciding the desirable tech stack across the application. If a team wants to override the default choice, they will have to support their decision with the pros and cons of the change that compelled their decision. The ideal tech stack may include the cloud provider, infrastructure, storage, monitoring, programming language, and a testing and logging framework.
* **Complexity** – Microservices impacts the operational complexity of the underlying application. You need to consider aspects such as the infrastructure which needs to be scaled up and down for a sophisticated level of automation. Load balancing and scaling, which will happen either for all services or only a few, will be a concern. Service discovery as services in a microservices world changes dynamically due to upgrades and scaling and that should be part of the consideration set. As should be monitoring, which needs to be configured for each service individually. And, think of the capability to handle scenarios when a subset of services are scaled up or down.
* **Decouple Capabilities** – There is a definite cost associated with moving to microservices from monolithic systems. Therefore, you’d want to consider what capability your systems have that can be decoupled and how you can migrate incrementally. Assess this in your system by taking a look at the operational readiness for creating services or migrating them. The primary idea is to start with capabilities that are fairly decoupled right in the monolith. These could be services that don’t require changes to several client-facing applications that use monolith and don’t use a data store. After decoupling simple edge services, consider those deeply embedded in the monolithic architecture.
* **Continuous Delivery** – As Martin Fowler points out in his [**article**](http://martinfowler.com/articles/microservice-trade-offs.html) on microservices tradeoffs, easily deploying small independent units is a blessing for development, but operations get more complicated as a few applications become hundreds of microservices. This reinforces the vital role of continuous delivery. While CD is a valuable skill for monoliths, it’s absolutely essential for microservices. Organizations such as Netflix and Amazon have spent their energy in building homegrown custom continuous delivery pipelines for microservices. As an alternative, organizations can choose a CD automation platform for a less intensive choice.
* **Data Services** – Refactoring the underlying data structures is one of the most complex issues of migrating to microservices. There are several models that you can follow. Use reference data to populate drop-downs in GUIs, Master Data Management to eliminate several views of an entity such as the customer within a database, flat object structure to store documents such as feedback surveys, independent tables to support data retrieval with SQL, and blob storage for storing a structured Java object, for instance.
* **Team Organization** – Lastly, you must reorganize your teams to ensure that all services are developed, deployed, and maintained independently. You may need an independent team to work on each microservice because when engineers work on multiple microservices, they might make optimization decisions that are not in the best interest of all associated services. Each team may need to have capabilities such as development, testing, Ops, database administration, UX, and product management. The central idea is to organize teams for maximum optimization of each microservice, without dependence on other teams.

# The benefits and challenges of using microservices architecture in cloud-native applications

Microservices architectures are becoming the go-to choice for cloud-native applications due to their abundance of benefits. Through domain-driven designs that break apart monolithic applications into individual, autonomous services, scalability and flexibility can be achieved in a way that was previously impossible.

1. **Scalability:** One of the biggest advantages of microservices is that each service can be scaled independently. This means that if one service experiences an increase in traffic or demand, it can be scaled up without affecting the rest of the application. This can help you avoid costly downtime and ensure that your customers have a seamless experience.
2. **Flexibility:** With microservices, you can use different languages and technologies for each service. This allows you to choose the best tool for the job and enables your development team to work with what they are most comfortable and proficient with. This can lead to faster development times, and better-quality code.
3. **Resilience:** Microservices are designed to be small and independent, so if one service goes down, it doesn't take the whole application with it. This can help you achieve higher availability and uptime, which is critical for any business.
4. **Faster Deployment:** Smaller services can be deployed and updated more quickly and frequently. This means you can get new features and bug fixes to your customers faster, which can help you stay ahead of the competition.

**Here are some challenges that come with using a microservices architecture…**

1. **Complexity:** Breaking an application into smaller services can increase complexity, making it harder to manage and understand the overall system. This can be especially challenging for businesses that are new to microservices.
2. **Coordination:** Microservices need to communicate with each other, and this can be a challenge to implement and maintain. This can lead to increased development time and costs.
3. **Security:** With more services, there are more potential attack surfaces, and it can be harder to secure the entire system. This is a critical consideration for any business, as a security breach can have devastating consequences.
4. **Monitoring:** With more services, it can be harder to monitor and troubleshoot issues across the entire system. This can lead to increased downtime and decreased productivity.

# 10 Challenges in Adopting and Implementing Microservices

We always hear great things about Microservices. But today let's talk about the top 10 challenges that come with adopting Microservices.

## Managing Microservices

As the number of microservices increases, managing them becomes tough. If there is no plan or accountability then we might end up with a lot of tiny microservices or with a huge macro-service.

## Extensive Monitoring and Logging

Monitoring what happens across the entire infra is critical. Along with this, we would also need an ability to trace end-user request path spanning services - also called Distributed Tracing.

## Service Discovery

It does not take much time for our services to grow beyond 100 and at that scale, discovering a service becomes a pain requiring us to put Service Discovery.

3 ways to do it are

* a central service registry
* load balancer-based discovery
* pure service mesh implementation

## Authentication and Authorization

Inter-service communication should be secure to ensure that a service does not abuse others; hence we need to put auth in place that allows authorized services to talk to each other.

## Config Management

Every microservice has a set of configs, like DB passwords, and API Keys. Committing them to the repository is an unacceptable practice, and we would not want every service to have its own config server.

Hence we need to have a central config management system that is fault-tolerant, robust, and scales well.

## No going back

It is extremely difficult to move back to monolith after the teams have tasted microservices. A few reasons would be

* services are written in various languages
* teams used to being autonomous
* teams have adopted new tools and processes

## Fault Tolerance

Outages are inevitable and as engineers, we always try to minimize them. A way to achieve this is to keep services loosely coupled that keep outage isolated ensuring no cascading failures.

## Internal and External Testing

End-to-end testing becomes complex as it is hard to spin up environments with all services running fine.

## Design with Failures in mind

Robust microservices require a counter-intuitive approach, and we need to assume everything would collapse after every line of code. Then we amend the code and architecture to handle it and re-iterate.