**Monoliths vs. Microservices**

The **monolith versus microservices** architecture debate has been around for years. Some claim that a monolith architecture is the best option if we're working from scratch. Others believe that the advantages of microservices outweigh the initial hassle of creating multiple servers.

In this tutorial, we'll be discussing these architectures in detail, and see how we can decide between them for developing our software application.

To understand the primary differences between these architectures, it is imperative that we have at least a basic understanding of coupling and cohesion. We should also get to know how these concepts should be incorporated to improve software. Understanding the balance between the two ensures stability in our systems.

Coupling:

This refers to the inter-dependency between various modules of a single application. It deals with the inter-modular dependencies.

Cohesion:

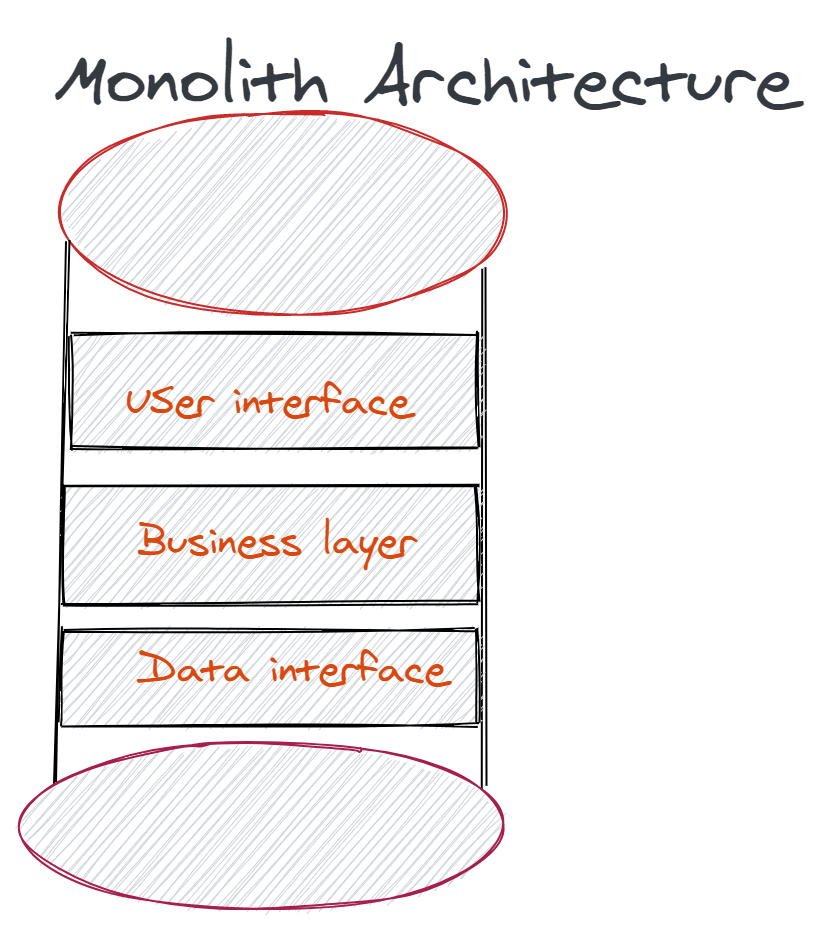
Cohesion is the measure of the degree to which the elements of a single module are *functionally related*. This is the intra-module inter-dependency.

**The general recommendation is to keep coupling low, while cohesion should be high to ensure stability.**

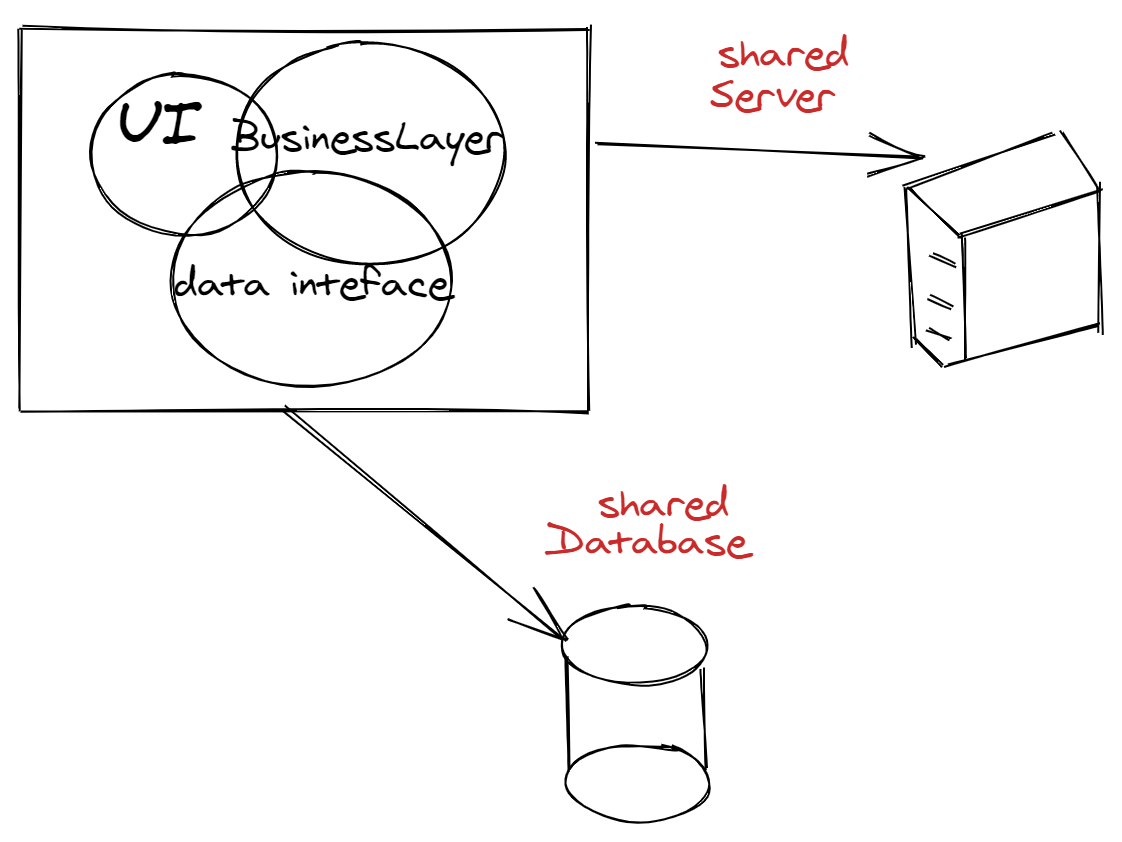
Now that we've reviewed these concepts, let's dive into the actual debate between proprietors of monolith and microservices architectures.

Monolithic Architectures:

Generally speaking, a "monolithic" architecture is regarded as the "traditional way" of developing a software application. It's a scenario where we have everything-- the UI, the Business Layer, and the Data interface-- packed within a single application.



This means that one large application encapsulates each and every single module and process of that application. It will typically have one universal server, a shared database and, a single file system-- which will be used by business units within that application.



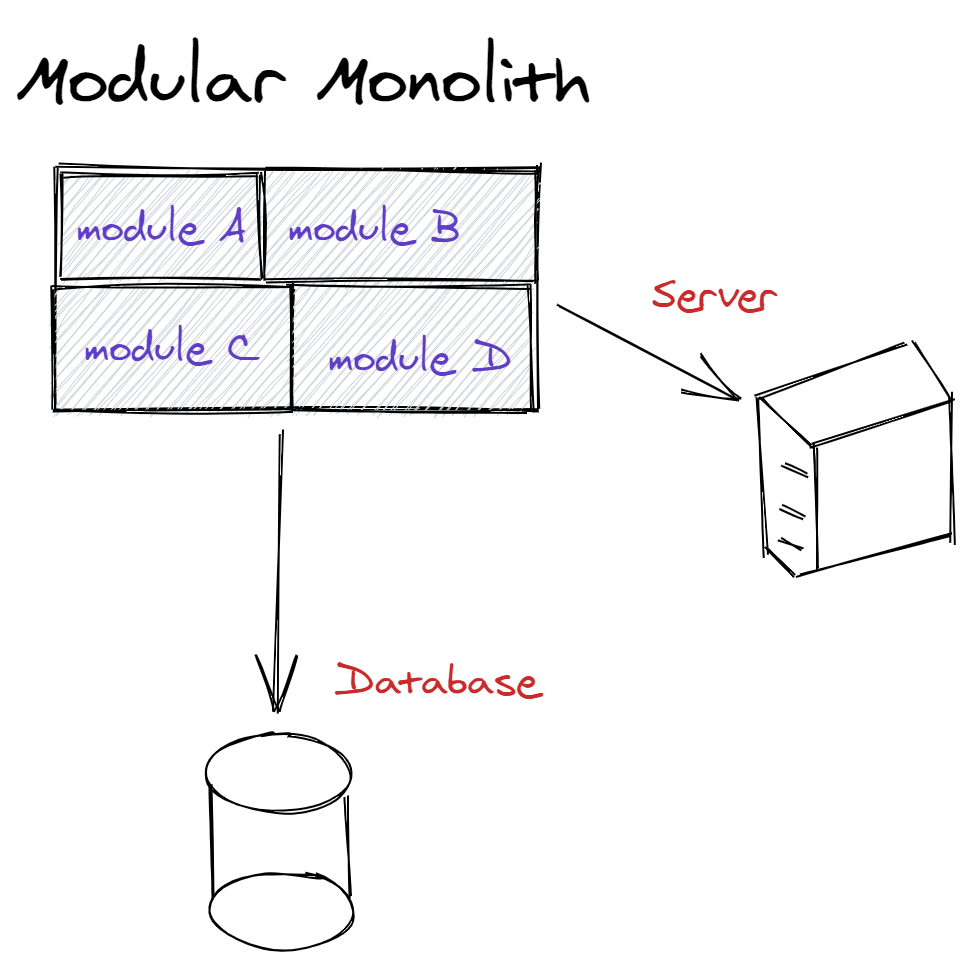
It's generally impossible for a single developer to know an entire application's codebase. Although people working in such architectures have a high-level understanding of each piece's specific functionality, no single person can go deep in all functionalities.

Let's discuss the various types of monoliths in order to boost our understanding.

Types of Monoliths:

The type of monolithic architecture we discussed previously is called the Single Process Monolith. We have another type called the Modular Monolith, which is a variation of the Single Process Monolith. In a modular monolith, we have separate modules for each functionality of our single process monolith and these modules are functionally independent.

A Single Process Monolith largely doesn’t support parallel programming. However, if we have well-defined boundaries of our project, then a modular monolithic architecture may support it. Additionally, the database that we use with modular monoliths typically doesn’t have the level of decomposition that is supported at the code level, creating several challenges.



There is yet another type of monolith called the Distributed Monolith, which, although consists of multiple services, needs to be deployed as one single application. Despite distribution into smaller services, such monoliths are highly coupled. The changes which you considered to be of local scope, may well impact other parts of the system.

The Truth Behind the Veil:

Having a single server or a single database or a single file system isn’t the defining characteristic of a monolithic architecture. This will become clear when we analyze it from the deployment perspective. Therefore, we can understand monolithic architectures to be those in which the whole system has to be deployed at once. In other words, **every single feature, module, or business unit, when deployed together, is as a monolithic architecture.**

Advantages of a Monolithic Architecture:

Since this is the traditional way of setting up software, there's several advantages and disadvantages.

It's been said that when we're developing software from scratch, a *monolithic architecture is the best choice*. This is because most of the time, we are not sure that how soon we'll need to scale up the software application. A monolithic architecture provides us with an efficient way of deploying the application earlier, by focusing first on features, and then scaling vertically as the app grows. Amazon, Netflix, and eBay all started with monolithic architectures.

Since everything is packed into one single unit, the applications tend to be faster. This is because they usually don’t use network calls to communicate within the application. It offers **simpler workflows** and mitigates the complexities involving things like end-to-end testing. The monolithic approach also offers easy **reusability of code** within the program. The advantages can be summarized by the following:

1. They are easy to scale
2. Simple to develop and test
3. Easy to deploy initially

Limitations:

Often, within an application, there may be multiple business domains that bind several business contexts inside. These business contexts may have their own dependencies or business requirements. Therefore, it becomes cumbersome to deal with *a huge number of dependencies* simultaneously. Moreover, if such applications aren’t written well, then owing to high coupling, changing part of the code may have a domino effect. As such, monolithic architectures are **not fault tolerant**. If one part of the system faces a breakdown, it will bring down the whole application with it.

The decisions made in the early stage of planning and development haunts engineers even after deployment. For instance, if we developed the application in Java, the straightest path forward is to **complete the entire application in the same programming language**. Newer features would need to be coded in Java, restricting us from using other programming languages.

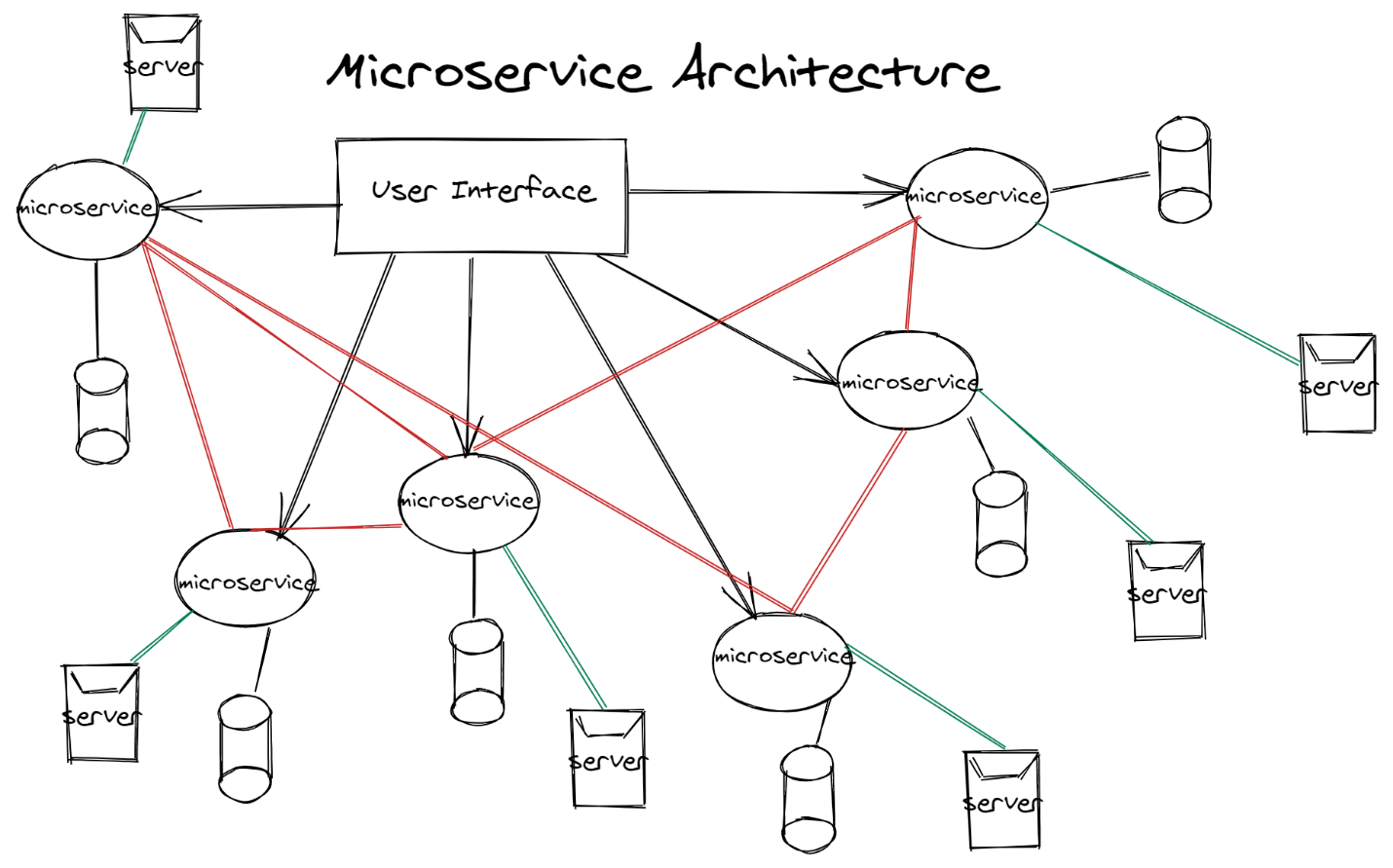
Additionally, in a monolithic architecture, there are high chances of running into situations where multiple developers may edit the same piece of code for different reasons. Similarly, different teams may want to deploy a single functionality at different times or in different ways. Studies have indicated that such practices lead to the issue of delivery contention, which means that the team is not sure as to who owns what and who makes the decisions. This may adversely affect the morale of the development team.

Furthermore, it is known that such applications usually grow, a time comes when *their weaknesses outweigh their advantages*. As the monolith becomes bigger and bigger, it becomes difficult to handle all the complexity that it brings with the increase in size.

Microservices Architecture

From the previous discussion on monoliths, we can see that a monolithic architecture has some inherent flaws. Such issues are bound to emerge once an application is deployed and grows to a certain size. Therefore, to overcome such flaws, the concept of microservices was invented.

Microservices offer a distributed architecture designed to provide a completely decentralized environment. Imagine a world where each business function is an independent application having its own server and a dedicated database. This enables you to model your application around smaller purposes and feature-sets, each one independently deployable. This is accomplished by dividing our monolith into a number of independent services, and making them communicate with each other (usually via RESTful API calls). We can define microservices as a decoupling of the entire application into smaller, independent services.



Advantages:

1. The primary perk of using microservices is the support of independent deployment. This means that if you wanted to make change to a single microservice (out of ten or twenty microservices), you can confidently do it without impacting other parts of the application. Furthermore, the databases typically aren't shared. If service requires data from another service, it would have to request it via API instead of pulling it using a database connection. Therefore, microservices overcome the issue of delivery contention.
2. Microservices allow us to use multiple programming languages when developing the application. You can have one service developed in Java, another in Python, and yet another in NodeJS. However, each service is typically developed in one programming language.
3. Since we have a microservice per single business unit, it encourages ‘parallel programming’. This allows individual teams to manage their own service only while maintaining the incoming and outgoing connections and internal implementation.
4. Each business functionality can have its own dedicated server and database. Therefore, in terms of scalability, we are not required to scale-up the whole system. Rather, we just scale-up the required service, *saving a lot of resources*. Microservices offer loose coupling, which ensures that a change made in one part of the system doesn’t produce a domino effect.
5. Owing to this very loose coupling, microservices are fault-tolerant. This means if one of the services goes down, the other won’t be affected. The application will only experience partial failure. Furthermore, this leads to an efficient way of testing and isolating the problem. You would easily identify which service malfunctioned, and the problem space is narrowed down. This is in contrast to a monolithic architecture where everything is packed into one giant structure.
6. Deploying new features becomes easy, since it will have its own dedicated server and database.

Limitations:

The limitations of microservices lie in the fact that a nontrivial amount of resources is required to build the initial system. First, teams will require more experienced developers familiar with this architecture. Secondly, looser deadlines are needed, as developing each microservice independently is time-consuming. The previous two limitations will require an increase in budget as well. In short, it takes more time and more money to build. As a result, many developers prefer to build the system as a monolith first, and then move towards microservices when they feel the system needs to be split.

Lastly, since microservices talk to each other via API calls, the **latency**, **packet loss**, and damage caused to **network wires** are issues introduced.

Microservice Vs Monolith - A Case study of Amazon

Let's say we're building an e-commerce giant. Consider that we have three different types of people who interact with our application:

1. the buyers
2. the sellers
3. the admin staff

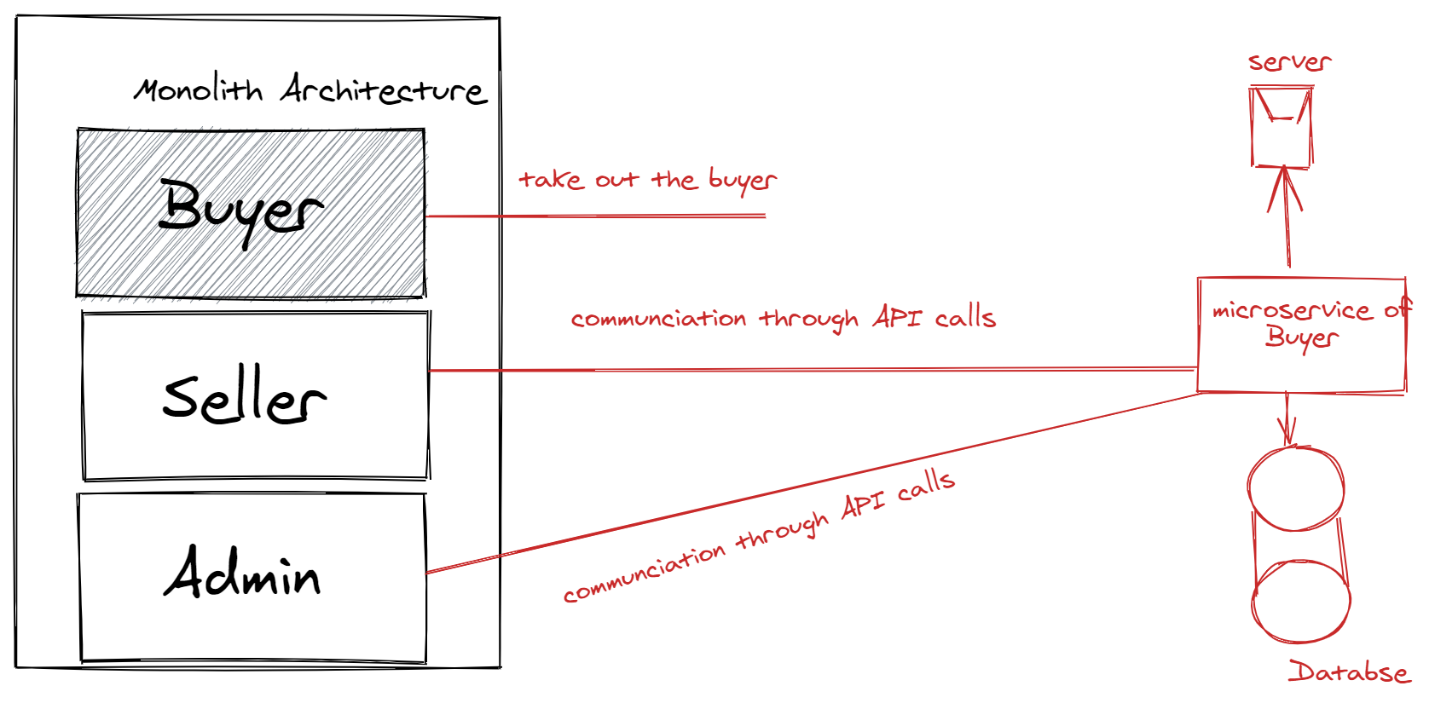
In the case of a monolithic architecture, everything will be under a single umbrella. If you want to offer a certain discount to your buyers, you'd require a change to the purchasing system-- but you'd then need to redeploy the entire application.

This wouldn’t be an issue with microservices-- we could have a separate server for each of the above-mentioned users and their purposes. If we want to edit code in the buyer portal, we don't need to touch the admin CMS.

Similarly, if part of the system broke down, a monolith's entire system could be affected. If it were a microservices architecture, the issue related to sellers could be isolated to that servers. As such, sellers might not be able to log in, but buyers could still view the seller’s products and add items into the cart to place orders.

Going From Monoliths to Microservices

It must be said that there isn’t a simple and quick way of migrating from monoliths to microservices. It's a long-term process maintenance of the monolith, while simultaneously extracting services from that whole application one by one. Each requires separately developing them independently, only to later link it with your monolith application when ready. In the case of an e-commerce app, we may extract just the buyer portal code, and create a separate microservice. After its development, we would remove it from the monolith, and use have our monolith architecture request buyer information via API calls.



After this iscompleted, we would be left with two more microservices to be extracted. They can be extracted and developed on similar patterns and in the end, we would have successfully moved from monolith to microservices.

Should we create a microservice for this feature?

One may face issues extracting the microservices from the monolith application. One of the problems would be scoping the number of microservices to be created. There is always the possibility that the team is creating an unnecessary abstraction. A good indication is to see if it only communicates with just one other service. This might suggest that you can merge these two microservices.

How to Select An Architecture

There's no shortcut to find determining which architecture is best suited for your application. Various factors must be considered. We must identify our **technical resources**, **strengths** of our business, and most importantly, the size and speed of our development team. This can be done by asking:

1. Whether we have people who can work with diverse programming languages.
2. Do we have a mentors experienced in developing separate services?
3. Does our business have the potential of instant growth?
4. How much time do we have to get the first deliverable to code completion?

*No single architecture is best in every condition*. Each architecture has its own pros and cons. It's the job of the engineer to identify if the pros of a certain architecture outweigh its cons. Remember-- it's never prudent for us to migrate from a monolith to microservice architecture just because Amazon or Netflix have done so. It's imperative to choose the architecture based on the observations that are relevant to your requirements.