Application Architectures

In this module we will be talking about **Application Architecture**.

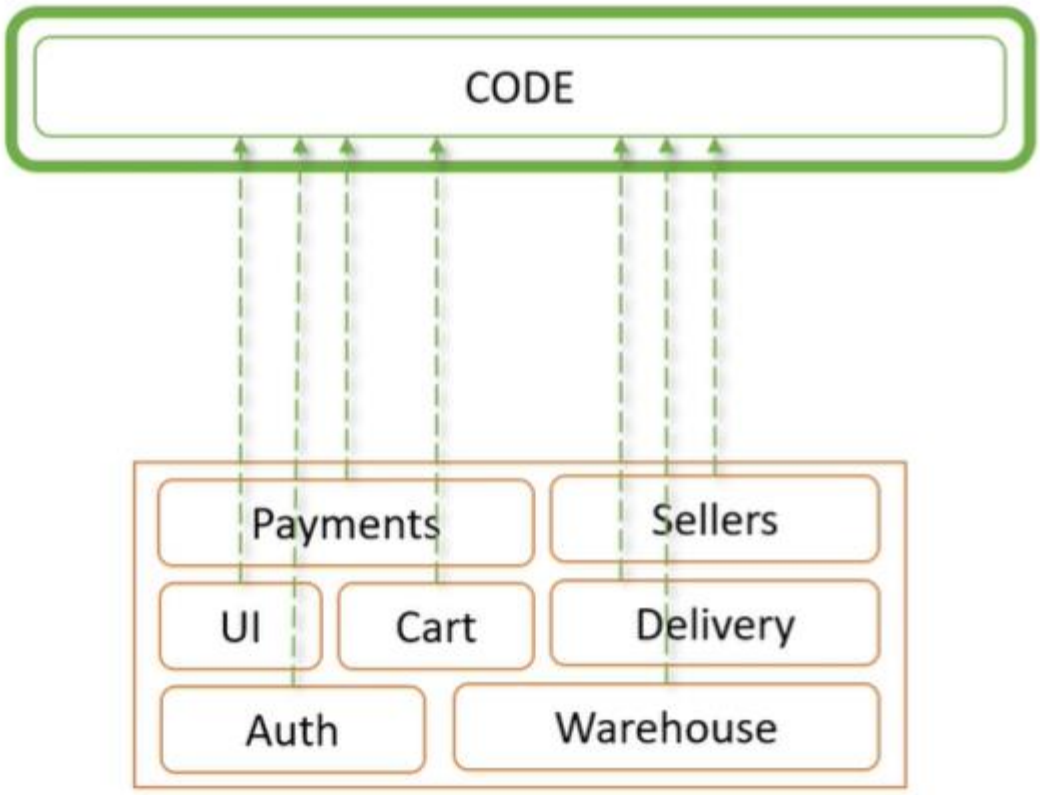
Some of the things that we'll cover here are:

1. What are the different architecture types
2. When should you prefer which architecture
3. What are the tradeoffs associated with each architecture

Monolithic Architecture

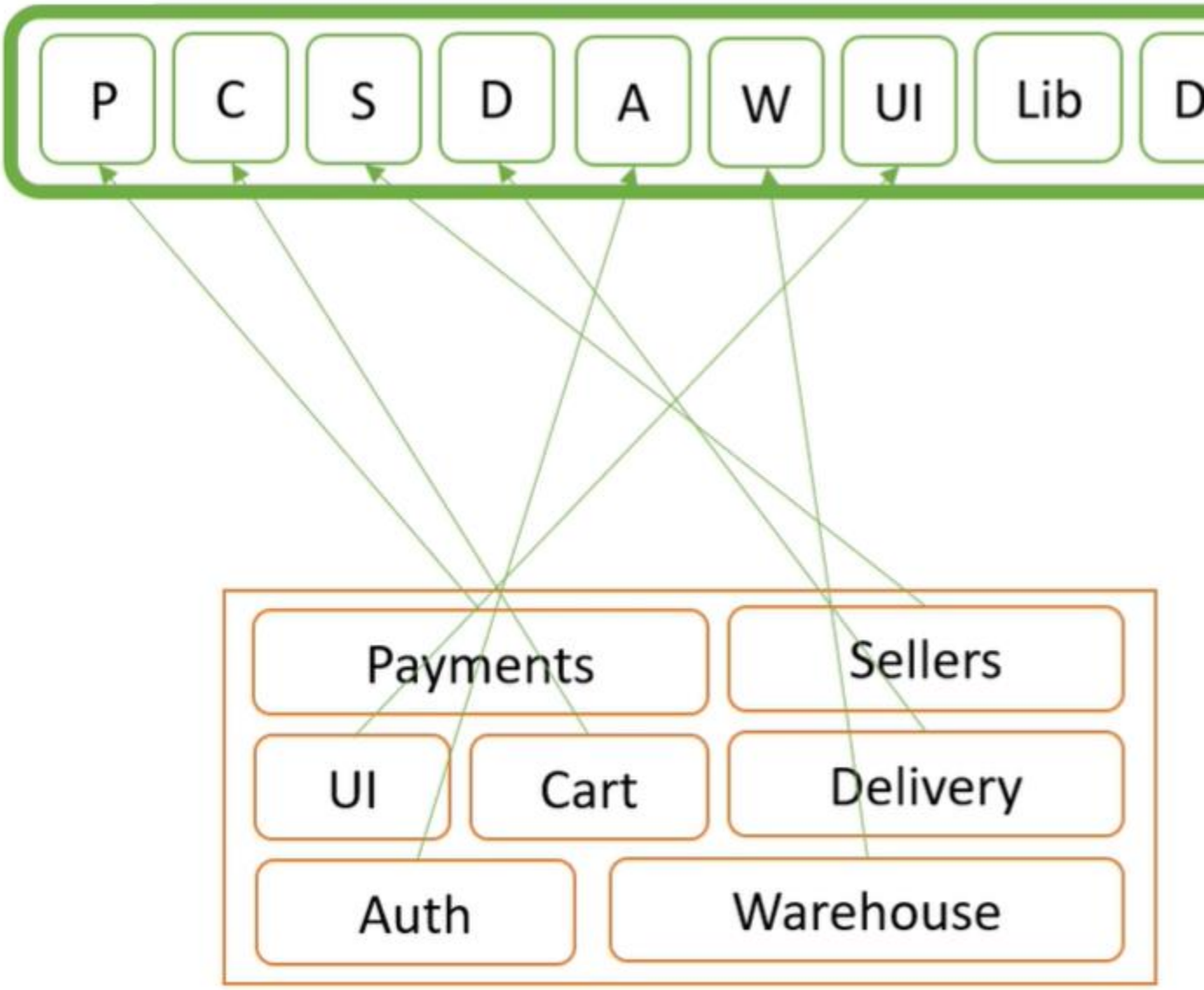
Back when the internet was just starting to gain popularity, websites used to serve mostly static content. There wasn't a lot of user interaction like we see now. The applications were much less complex, and so was their architecture. A single application used to take care of the entire user journey, everything from UI rendering to backend business logic to fetching the data from DBs or File Systems. This was the world of Web 1.0.

Then came Web 2.0 with social networks, e-commerce, and online gaming and things became a lot more interactive. By this time everything was still maintained in a single huge codebase. If you consider an e-commerce system, back then everything from UI to business logic for payments, carts, orders, etc. was maintained in a single codebase. This is known as **Monolithic Architecture**.



The problem with this approach was that the code was very complex, difficult to maintain, and hard to iterate and improve. On top of that, multiple people were working on the same codebase; it was a recipe for disaster.

To solve this, we started using more efficient design patterns, various OOPS concepts to streamline the code and tried to make it more modular. For this we broke down the code into various modules which took care of a specific feature group. So all the payment related stuff was handled by a Payments module, cart related features were taken care of by a Cart module, orders related code in an Order module, if there was any shared code we moved it out to a shared module, and so on. Now instead of everyone working on the same code, each team could work on the module meant for their own features. This is how the modern day monoliths are built.



What are the Downsides?

As the team size grows, some problems start coming up with this approach as well.

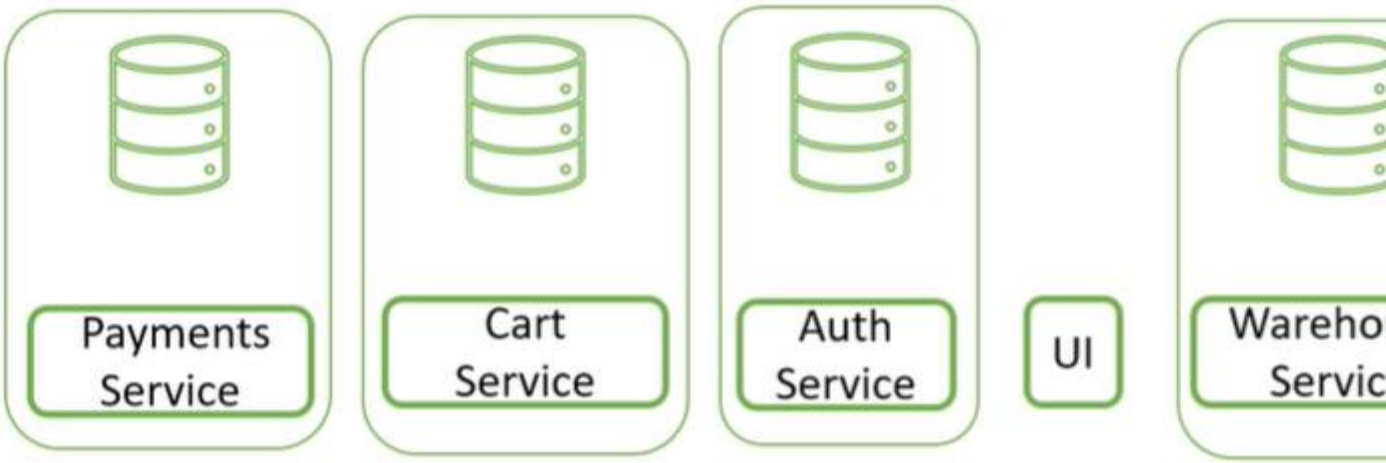
- One of the most common problems with monoliths is that you are bound to a single technology stack. Suppose you have a monolith built on Java with Spring framework. Now you need to add some Machine Learning logic, and you want to use Python for it. That is nearly impossible in monolithic architecture. You either need to figure out a way to do this with Java, or you need a standalone

application that handles your machine learning logic, which defeats the purpose of a monolithic app.

- Another problem would be that it is very easy to break things in such an architecture. That is because, when you have such a huge codebase, it is nearly impossible for everyone to know everything about how the entire system works. If you change some logic in the shared code, you might end up breaking someone else's feature. Sure you can have test cases, but even those are not enough sometimes.
- Another major issue is scalability. It is very tricky to scale a monolithic application. Let us look at the example of an e-commerce application. In case of a Black Friday sale, you might need to scale your payments and cart modules, but Warehouse and Notification modules can continue to work at the same pace. This cannot be done in a monolithic app. Since it is the same codebase, you will need to deploy the entire system again and again. This means the capacity meant for the Warehouse module might be sitting idle or that the Payment and Cart modules may choke the rest of the system.
- Deployments are also a very tedious process here. Since the code is huge, it takes much longer to build, package, and deploy the code. That said, if you update the code for Warehousing, even the Payments module must be redeployed since everything is packaged together.
- If there is a performance issue in one part of the system, chances are it will leak to the rest of the system as well. Even a small memory leak in the Warehousing module, if it starts choking the other modules, can cause the entire application to have issues.

So what is the solution for this?

Microservice Architecture



The idea is to break down the application into logical components such that these components become services of their own. Meaning they have the code for that part of the user's journey and the data, basically that this service is now the primary source of truth for that particular feature group. Normally this is also how the teams would be structured, so each team would work on the services that handle their features. These services will now be communicating with each other via a set of API calls like REST APIs or Remote Procedure Calls.

Benefits of Microservice Architecture

- We are not bound to a single technology stack anymore. Different services can use different languages or databases as needed.
- Each system does one thing and does it well, without worrying about breaking another set of features.

- Engineers will be working on and maintaining a smaller codebase.
- Iterating, deploying, and testing becomes a lot easier.
- Unlike in monolith, we can now independently scale up Cart and Payments Services, and the rest of the services can continue working as they are. This ensures optimized use of resources and makes auto-scaling a lot easier.

Monolithic vs Microservices

As we can see, microservice architecture has a lot of benefits over monolithic architecture. Does this mean we should always use microservices?

Not necessarily. To understand why let's look at the cost of microservices.

Latency

- One of the key reasons is latency. Function calls are faster than API calls, so it makes sense that monoliths will have lower latency. Usually, this latency is not high enough to be noticed by the user, but if you are working on something that needs a response in a few microseconds then you should definitely think about using monolithic architecture.
- With network calls comes the possibility of network failures and slightly increases the complexity of error handling and retries.

Backward Compatibility

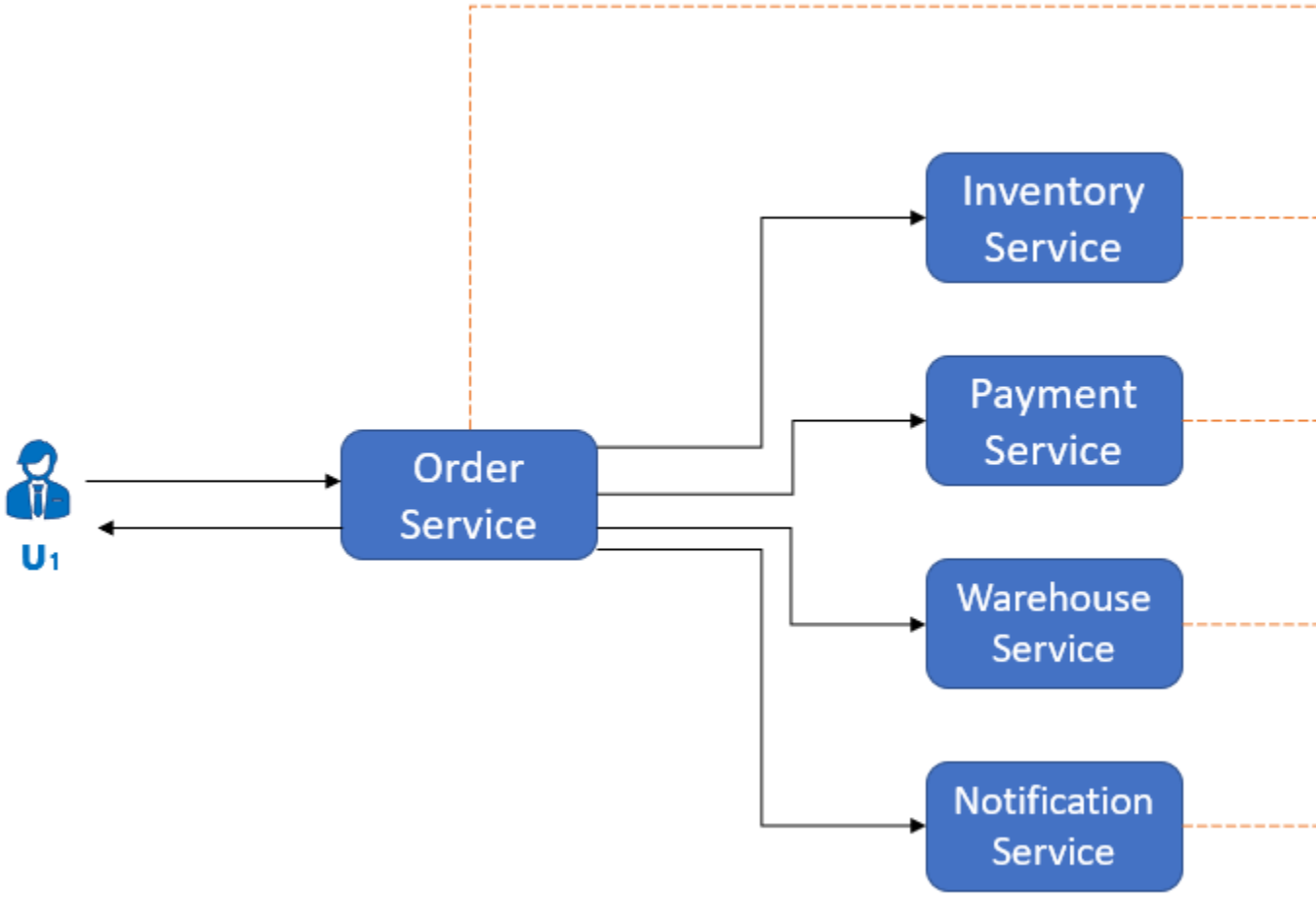
- If a service needs a new mandatory parameter and the other services have not made the change accordingly, certain flows will break.
- In monolith it will be caught in the development phase since the compiler will throw an error. To solve this we need some good automated test cases.

Logging

- If you need to trace logs for a user in monoliths, it is easy to do so as they are all in the same place. But in microservices, for each request from the user there will be multiple service calls.
- In the below example, consider a single request from the user to the order service. The order service is talking to Inventory, Payment, Warehouse, and Notification Services, and thus there will be logs in each of these services. So

tracing exactly what happened becomes very difficult as we need to check logs in each system.

- A better approach would be to store all the logs in a central place where they can be queried. This is a cost since you either need to build a Log Aggregation System or buy a third-party system's license.



Service Discovery

- If we are calling multiple services, we need a way to identify where they are hosted.
- We could hardcode it of course, but that makes it a little difficult to make changes, especially if you have a lot of microservices.

Other Smaller Costs

- We also need a configuration manager system to push configurations into all the nodes of a particular service.
- Even though auto scaling is a benefit of microservices, doing it in a controlled manner needs to be done by a system as well.
- In monoliths, since all the code and data are in one place, access control is straightforward. With multiple microservices, access control, security policies, etc., will need to be done for each service repeatedly.
- There needs to be a managing infrastructure for all the services that need to asynchronously talk to each other, something like a centralized Kafka deployment, which becomes an additional cost.

So, as shown in the diagram above, we have a lot of components running just to support the microservice architecture. If our system was limited to order placement, the cost of the support components would be more than the development cost of the entire application. But when you think of a real-world scenario, there would be hundreds of services working together. In such a case, the cost of supporting components is negligible compared to the development cost, and the benefits would be worth it.

Conclusion

In conclusion, if you are a small business working with a small team and have a limited set of features, monolith might be a better approach for you. However, if you are working on a huge product with hundreds of microservices, the maintenance cost of microservices will be worthwhile.

Also, usually, when you think about HLD interviews, chances are you will be developing an overall architecture for a complex system that might be difficult to design in a monolithic manner. So thinking in terms of microservices will be a good idea.