



# Architectural Design Considerations

## Monolith vs Microservices

### Monolith

- What is a Monolithic application? Essentially, it's a single, unified codebase that performs all of your application's functions.
- Advantages? It's straightforward to develop, test, and deploy. Atomic transactions are easier.
- But the downsides? As your app grows, the monolith becomes increasingly complex, hard to understand, and hard to scale.
  - As the monolith becomes more complex, design becomes really important.

### Microservices

- On the other side, we have Microservices, which break down your application into small, focused components.
- The benefits here include better scalability and a clear separation of concerns, which also allows for independent deployments. Also:
  - You can use different programming languages, different hardware environments (for example: use a fast server to host the backend and serve the frontend, but use a slower server for handling offline internal reporting)
- The downsides? You might encounter network latency and challenges in maintaining data consistency across services.

### Distributed Monolith

- Now, what's worse than either of these? The Distributed Monolith. Imagine a set of microservices so tightly coupled that they act as a monolith.
- How do you know you're creating one? If you find that you need to deploy multiple services together to avoid breaking changes, that's a big red flag.
- How to avoid it? Let's take a look at some best practices for microservices.

# Best practices

- **Start with a Monolith**
  - Initially build a monolithic application to understand business domain and requirements better.
  - Break into microservices once a stable version is achieved and domain boundaries are understood.
- **API First Approach**
  - Design APIs before implementing the services.
  - Ensure APIs are versioned to handle changes without breaking clients.
- **Data Isolation**
  - Each service should own its data and its database schema, if possible.
- **Domain-Driven Design (DDD)**
  - Use Domain-Driven Design to identify the boundaries between microservices.
  - DDD is a major software design approach. You can dive really deep into it, which - especially for smaller projects - I wouldn't recommend. The key thing to remember though is that you organize things according to the concepts in the domain, whether that's microservices, entities, classes, methods.
- **Loose Coupling and High Cohesion**
  - Services should be as decoupled as possible.
  - Keep related functionalities together for high cohesion.
- **Single Responsibility Principle**
  - Each microservice should do one thing and do it well. That should map to a context in the business domain.
- For example if you have a train ticket selling system, I could imagine there being microservices for payment handling, ticket management and logistics of mapping the number of tickets sold to train planning.
- Another example: building a trading system. Microservice that operates as layer between broker and your trading bot, microservice for the trading bot itself, microservice for data collection and analysis, microservice for admin user interface, one for connection with the accounting system, etc.

## Stateful vs Stateless Services

### Stateless Services

- Stateless services are those that don't store any client state between requests.
- Why are these great? Because they are easy to scale. Just spin up more instances as demand grows.
- Typical use-cases include RESTful APIs and Web Servers.

## Stateful Services

- Stateful services, in contrast, maintain state between different requests or sessions.
- Scaling these services needs more planning. You need to think about session affinity, distributed caches, and so on. (show example on tablet)
- Common examples are databases and caching services like Redis.

## Decomposing to Stateless Components

- If possible, consider decomposing stateful services into stateless ones, using techniques like external databases or session stores.
- This allows you to enjoy the scalability benefits of stateless services even when dealing with stateful operations.

## Choreography vs Orchestration

### Choreography

- Choreography involves decentralized, autonomous interactions between services.
- This is good for system flexibility and fault tolerance. If one service fails, the others can often continue.
- The downside? It can make debugging and tracing more complicated, and you might miss centralized control.

### Orchestration

- Orchestration means a centralized service coordinates the interactions between services. This is what we would also call a “broker architecture”.
- It's easier to monitor and control but it also introduces a single point of failure.
- It also may not scale as elegantly as a choreographed system.

## Frameworks vs PaaS Solutions

### Using Frameworks

- Examples
  - installing and using Kafka for message queues
  - self-hosted Wordpress
  - Airflow or Prefect for data orchestration
- You get more control and customizability but at the cost of complexity. You have to manage it all.
- There are also concerns over versioning and updates.

## Using PaaS Solutions

- Alternatively, you could go with Platform as a Service solutions like MongoDB Atlas, Redis as a service, Elasticsearch.
- Cloud providers also have lots of tools that you can use for messaging, events, scheduling, etc. I'll talk more about these things later in the course.
- The advantages? Much of the management is taken care of, and you get a simplified, streamlined interface.
- The downside primarily is the cost and the potential for vendor lock-in. Once you use all the stuff from Google Cloud, it becomes harder to switch from Google Cloud to something else. Which is of course what they want.
- Also, in some cases actually using these solutions can be hard. They may have complex processes for authenticating, not having exactly the features you want, or way too many options. (show cloud task example)

## Summary and Takeaways

- We explored key considerations when architecting software.
- Remember, there's no one-size-fits-all approach. It's crucial to weigh the pros and cons based on your specific needs and constraints.