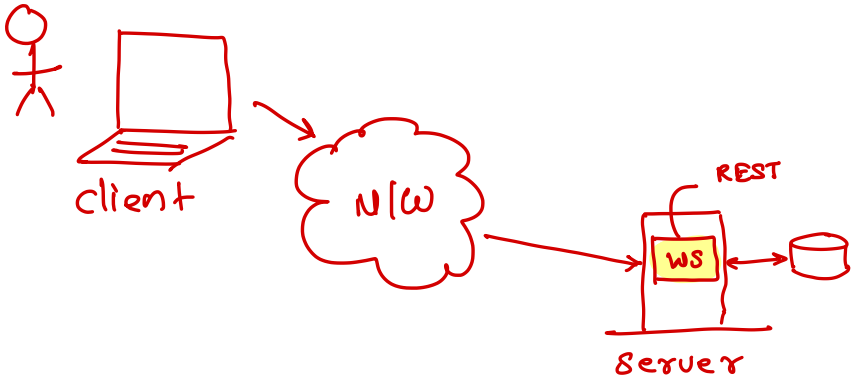
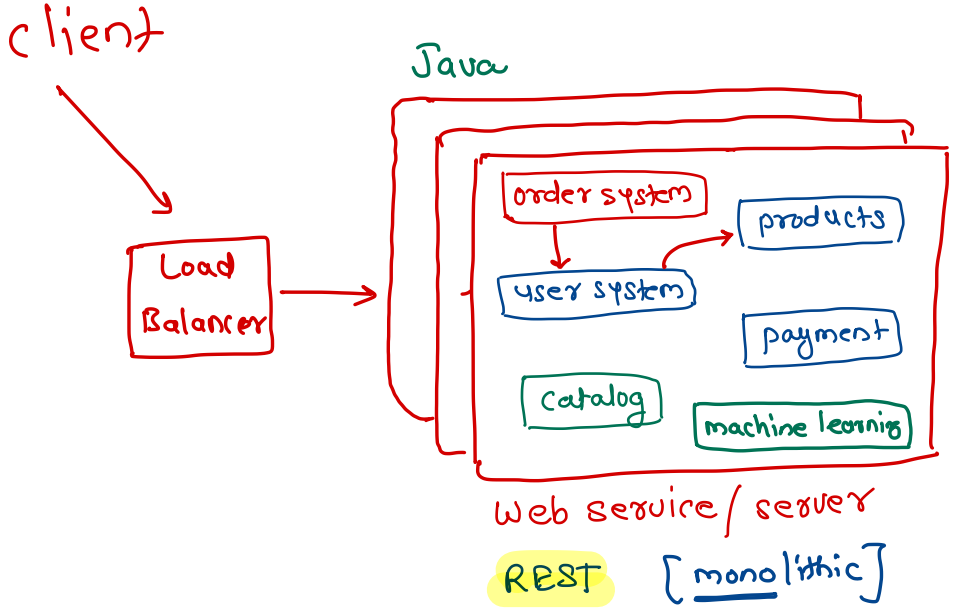
# Monolithic

# Monolithic Architecture

- Monolithic applications are designed to handle multiple related tasks
- They're typically complex applications that encompass several tightly coupled functions
- For example, consider a monolithic ecommerce application. It might contain a web server, a load balancer, a catalog service that services up product images, an ordering system, a payment function, and a shipping component
- As you can imagine, given their broad scope, monolithic tools tend to have huge code bases. Making a small change in a single function can require compiling and testing the entire platform, which goes against the agile approach today's developers favor.



# Advantages

- Simple to develop
  - The goal of current development tools and IDEs is to support the development of monolithic applications
- Simple to deploy
  - You simply need to deploy the WAR/JAR file (or directory hierarchy) on the appropriate runtime
- Simple to scale
  - You can scale the application by running multiple copies of the application behind a load balancer

fast : speed

# Disadvantages

- The large monolithic code base intimidates developers, especially ones who are new to the team. The application can be difficult to understand and modify. As a result, development typically slows down. Also, because there are not hard module boundaries, modularity breaks down over time. Moreover, because it can be difficult to understand how to correctly implement a change the quality of the code declines over time. It's a downwards spiral.
- Overloaded IDE - the larger the code base the slower the IDE and the less productive developers are
- Overloaded web container - the larger the application the longer it takes to start up. This had have a huge impact on developer productivity because of time wasted waiting for the container to start. It also impacts deployment too.
- Continuous deployment is difficult - a large monolithic application is also an obstacle to frequent deployments. In order to update one component you have to redeploy the entire application.
- Scaling the application can be difficult - a monolithic architecture is that it can only scale in one dimension. On the one hand, it can scale with an increasing transaction volume by running more copies of the application.

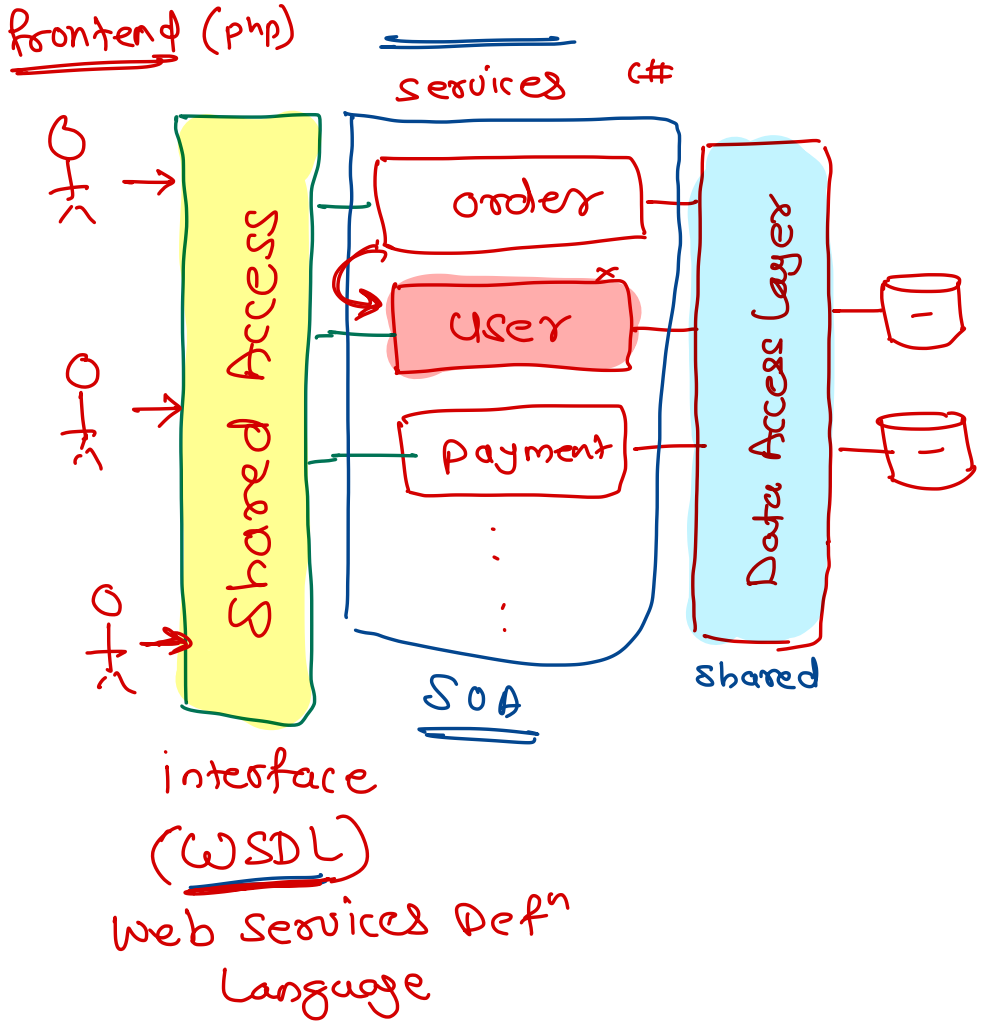
# When to choose ?

- **Small team**
  - If you are a startup and your team is small, you may not need to deal with the complexity of the microservices architecture
  - A monolith can meet all your business needs so there is no emergency to follow the hype and start with microservices
- **A simple application**
  - Small applications which do not demand much business logic, superior scalability, and flexibility work better with monolithic architectures
- **No microservices expertise**
  - Microservices require profound expertise to work well and bring business value
  - If you want to start a microservices application from scratch with no technical expertise in it, most probably, it will not pay off
- Quick launch (POC)
  - If you want to develop your application and launch it as soon as possible, a monolithic model is the best choice. It works well when you aim to spend less initially and validate your business idea.

# Service Oriented Architecture (SOA)

# What is SOA ? : SOAP : Simple object Access Protocol

- Service-oriented architecture (SOA) is a type of software design that makes software components reusable using service interfaces that use a common communication language over a network
- A service is a self-contained unit of software functionality, or set of functionalities, designed to complete a specific task such as retrieving specified information or executing an operation
- It contains the code and data integrations necessary to carry out a complete, discrete business function and can be accessed remotely and interacted with or updated independently
- In other words, SOA integrates software components that have been separately deployed and maintained and allows them to communicate and work together to form software applications across different systems



# Benefits

- Editing and updating any service is easy
- Services have the same directory structure, which allows consumers to access the service data from the same directory every time
- Services communicate with other applications using a common language which means it is independent of the platform
- Services are usually small size as compared to the full-fledged application. Therefore, it is easier to debug and test the independent services
- SOA allows reusing the service of an existing system, alternately building the new system
- It offers to plug in new services or to upgrade existing facilities to place the new business requirements
- You can enhance the performance, functionality of a service, and easily makes the system upgrade
- SOA can adjust or modify the different external environments
- The companies can develop applications without replacing existing applications

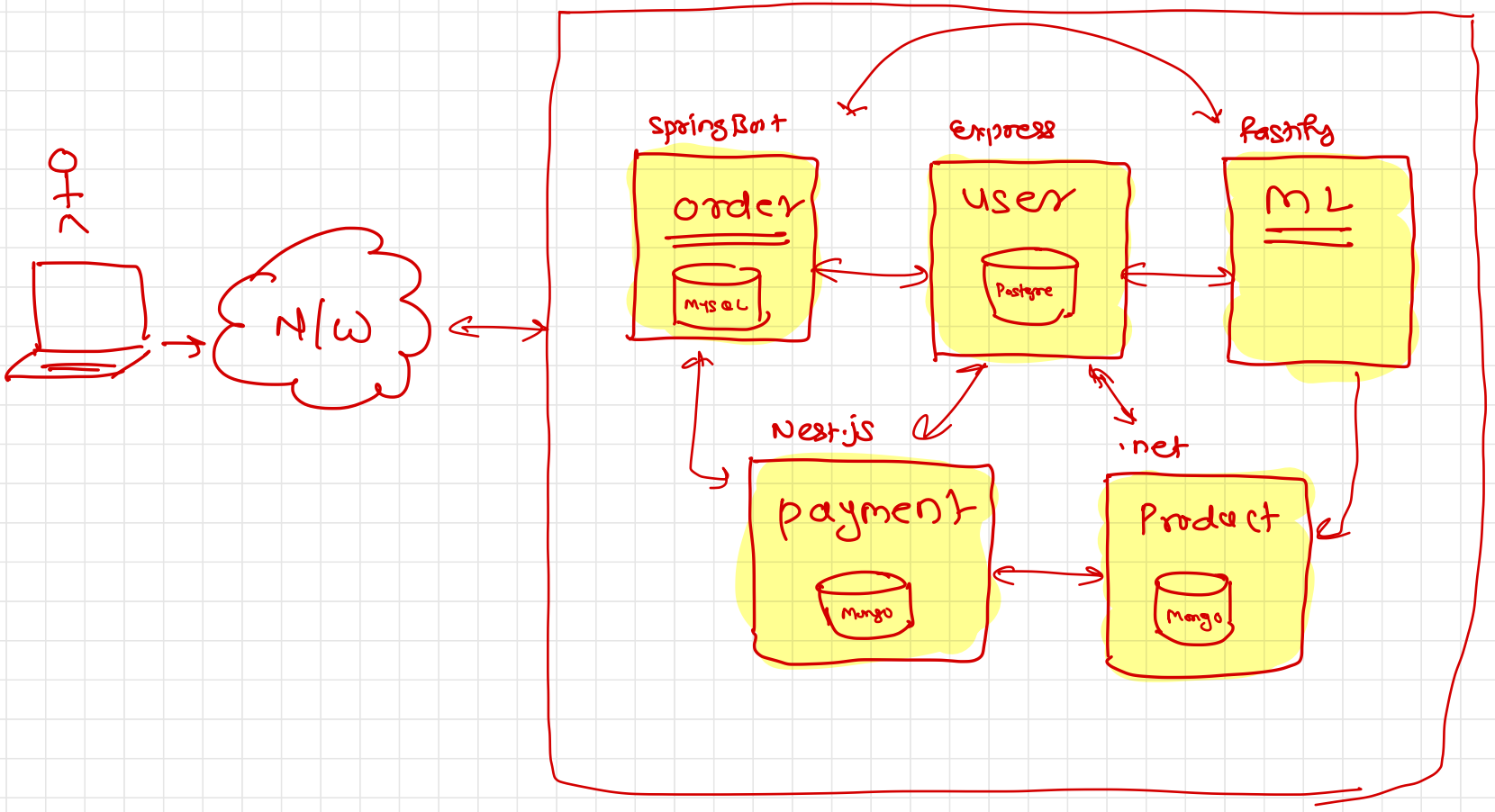
# Disadvantages

- All inputs should be validated before it is sent to the service
- SOA is a costly service in terms of human resources, development, and technology
- Some web service needs to send and receive messages and information frequently, so it reaches a million requests per day easily
- SOA requires high investment cost
- There is bigger overhead when a service interacts with another service this will increases the response time
- SOA service is not suitable for GUI (graphical user interface) applications that so it will become more complicated when the SOA needs heavy data exchange
  - ↳ mostly used for backend [server]

# Microservices

# Microservice Architecture

- Microservices are small, independent, and loosely coupled. A single small team of developers can write and maintain a service.
- Each service is a separate codebase, which can be managed by a small development team
- Services can be deployed independently. A team can update an existing service without rebuilding and redeploying the entire application.
- Services are responsible for persisting their own data or external state. This differs from the traditional model, where a separate data layer handles data persistence.
- Services communicate with each other by using well-defined APIs. Internal implementation details of each service are hidden from other services. REST / GraphQL
- Supports polyglot programming. For example, services don't need to share the same technology stack, libraries, or frameworks.


N/W
SpringBoot
order
MySQL
Express
user
Postgre
FastAPI
ML
Nest.js
payment
Mongo
.net
Product
Mongo

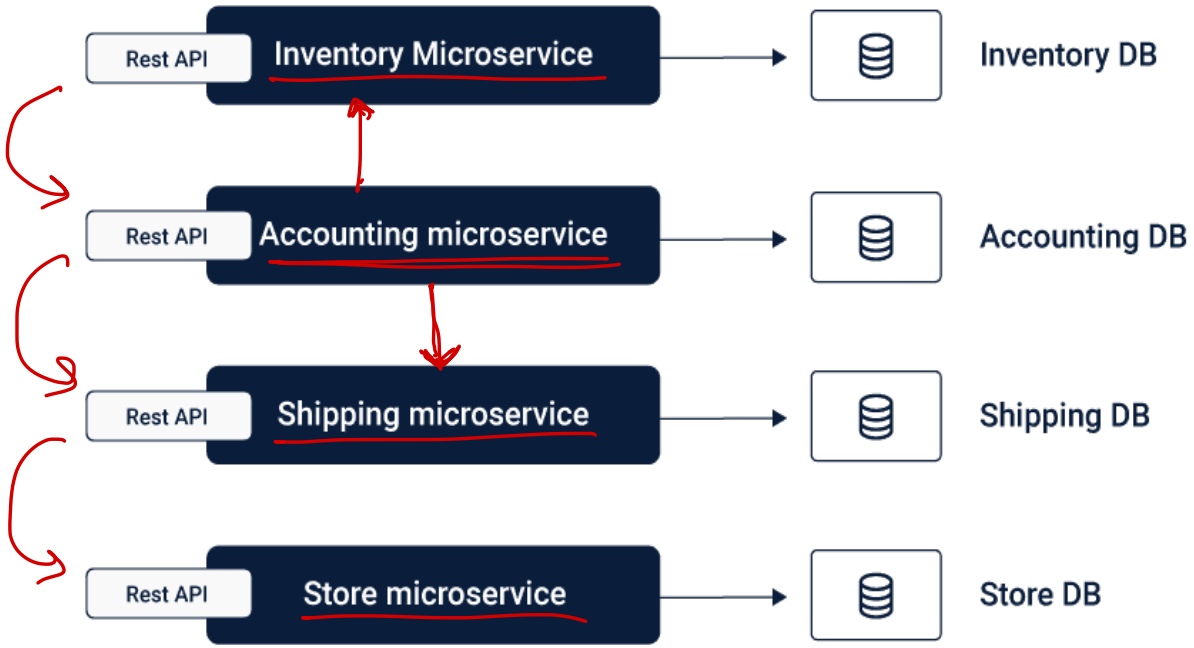
# Communication in Microservices

- In Microservices Architecture, the Microservices communicate with each other using messaging, using a lightweight and straightforward mechanism
- There are Synchronous and Asynchronous messaging techniques and several message formats that the Microservices could follow to communicate depending upon the purpose and requirements

# Synchronous Messaging Techniques

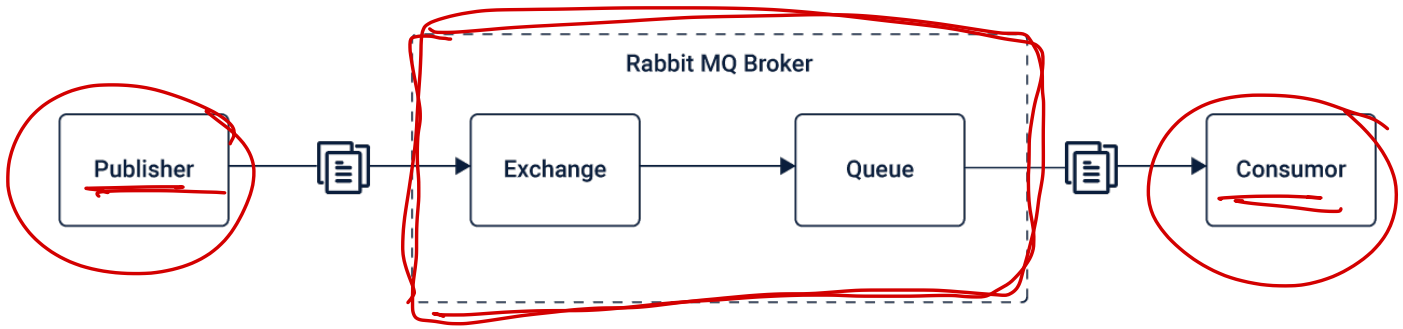
GraphQL

- REST is one of the highly preferred synchronous messaging techniques used by the Microservices, where the HTTP request-response that defines a set of constraints based on the resource API
- In the REST communication, services communicate over HTTP in the Request-Response set, without any additional infrastructure requirements.
- As REST allows a point to point communication, it can be used to connect the microservices directly for the discussion, this can surface the issue of coupling, causing several dependencies within the Microservices.



# Asynchronous Messaging Techniques

- In cases where an immediate response is not required Asynchronous messaging techniques such as AMQP, STOMP, or MQTT can be used.
- AMQP is an open standard application layer protocol used for asynchronous messaging that is highly reliable and secure. AMQP is a binary, flow-controlled communication protocol that has encryption. [IoT]
- RabbitMQ uses AMQP protocol to communicate by queuing the message and siphoning it off to single or multiple subscriber programs that listen to the RabbitMQ server. As RabbitMQ is written in Erlang, you must have Erlang installed in your system before you download the RabbitMQ.



① place order

② db update

→ ③ customer's SMS

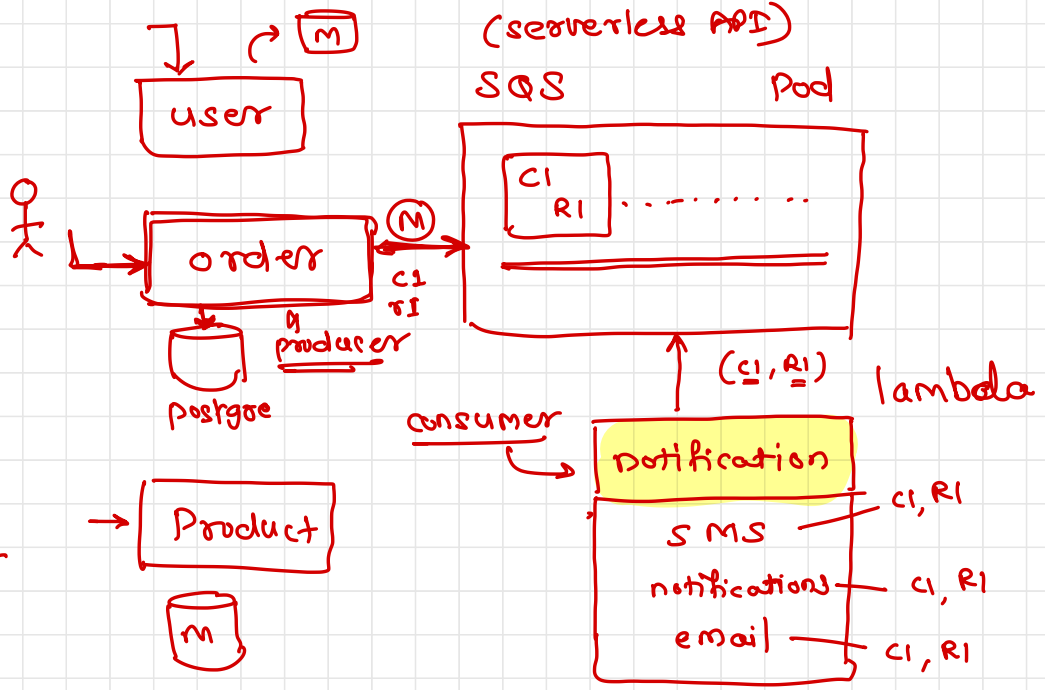
→ ④ push notification

→ ⑤ email to customer

→ ⑥ retailer SMS

→ ⑦ retailer's PN.

→ ⑧ email to retailer

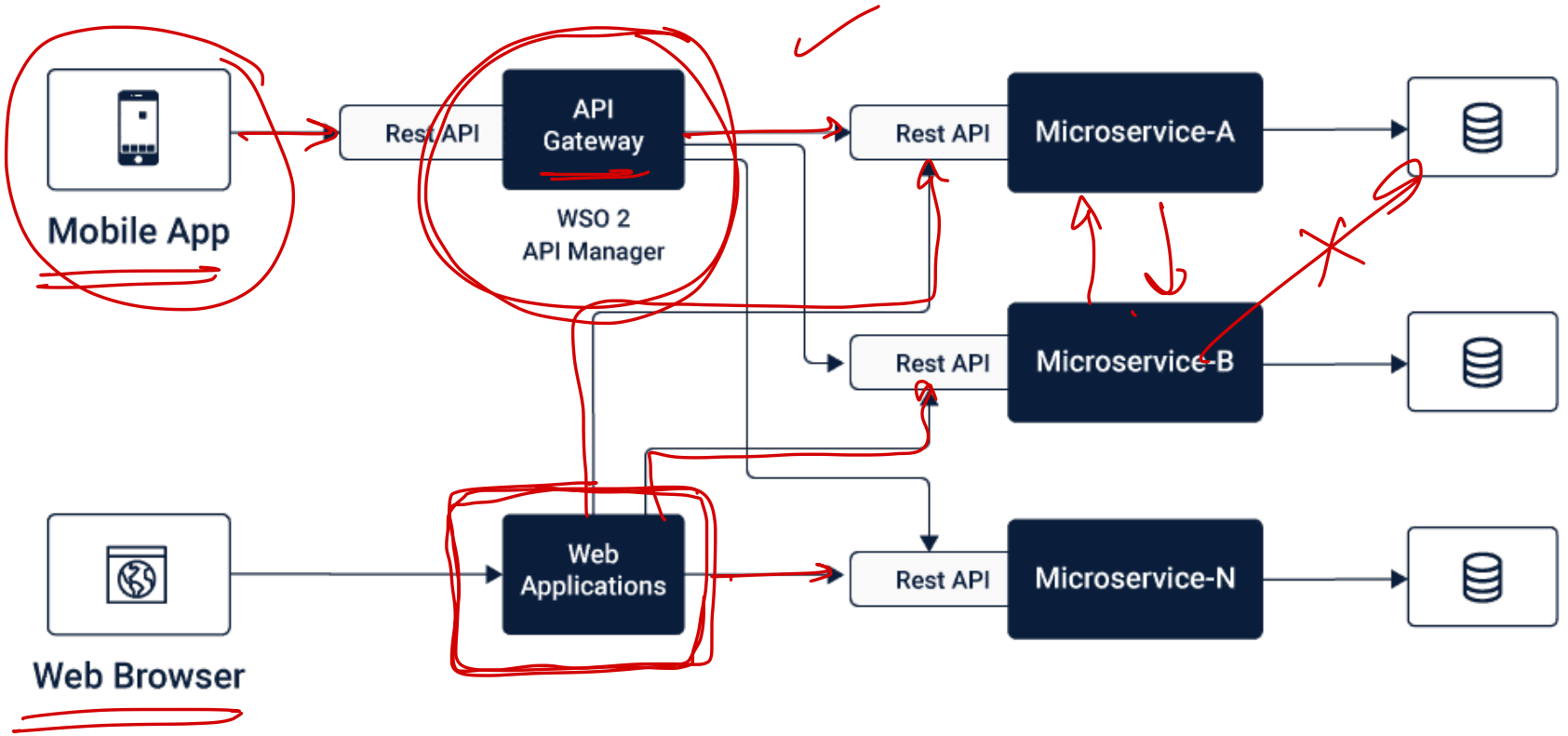


# Integration of Microservices

- As the individual Microservices serve different tasks that come within their scope, to realize a business use case, several Microservices have to coordinate and deliver the desired result in a combined effort
- Thus, there has to be inter-service communication using a lightweight message bus or gateway that involves minimal routing without any business logic to avoid complexity within the architecture
- There are different ways in which the inter-services communication takes places, depending upon the requirements and frequency of communication

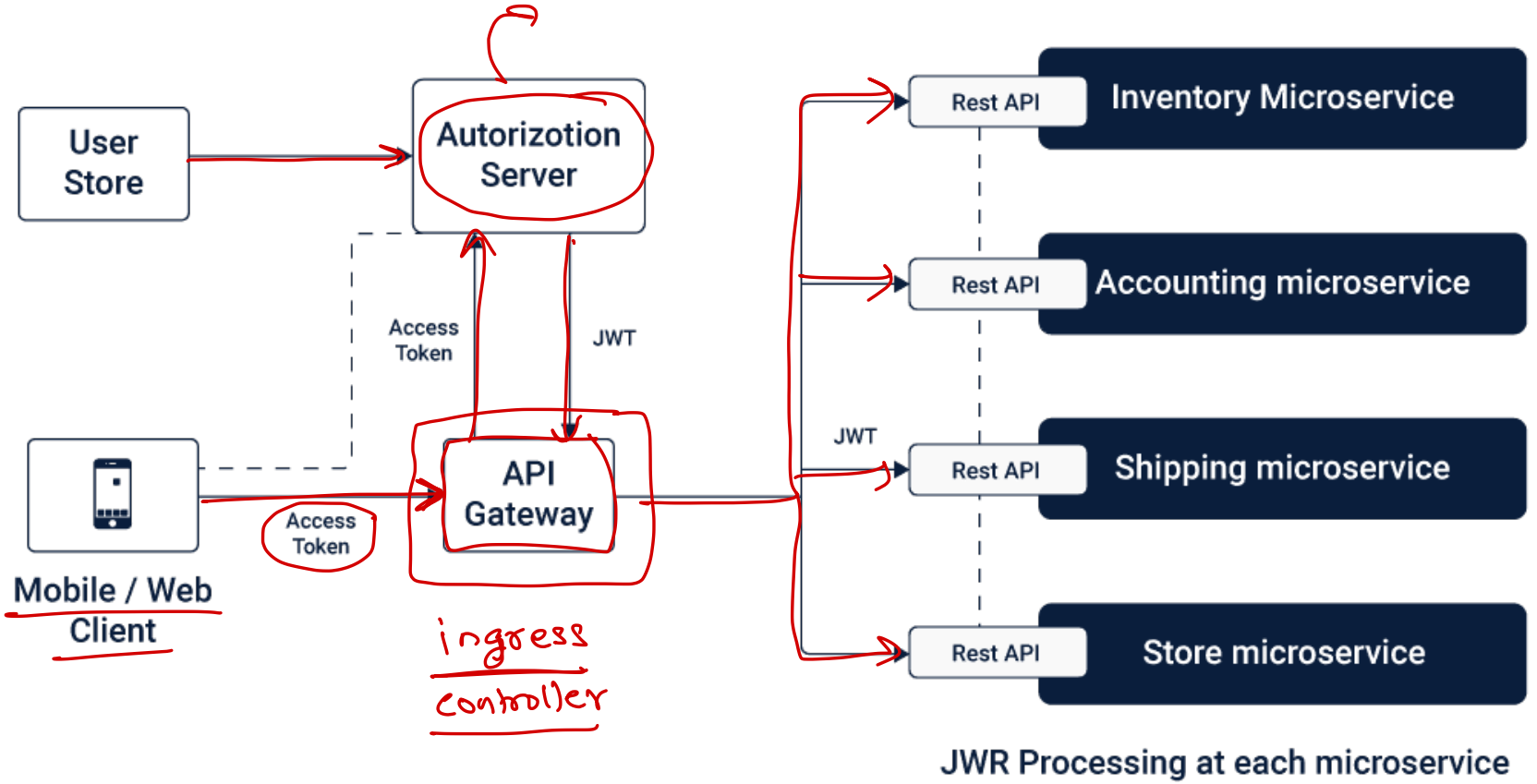
# Point to Point Inter-services Communication

- In this method of inter-services communication, the routing logic depends entirely on the endpoints, letting the Microservices to communicate directly



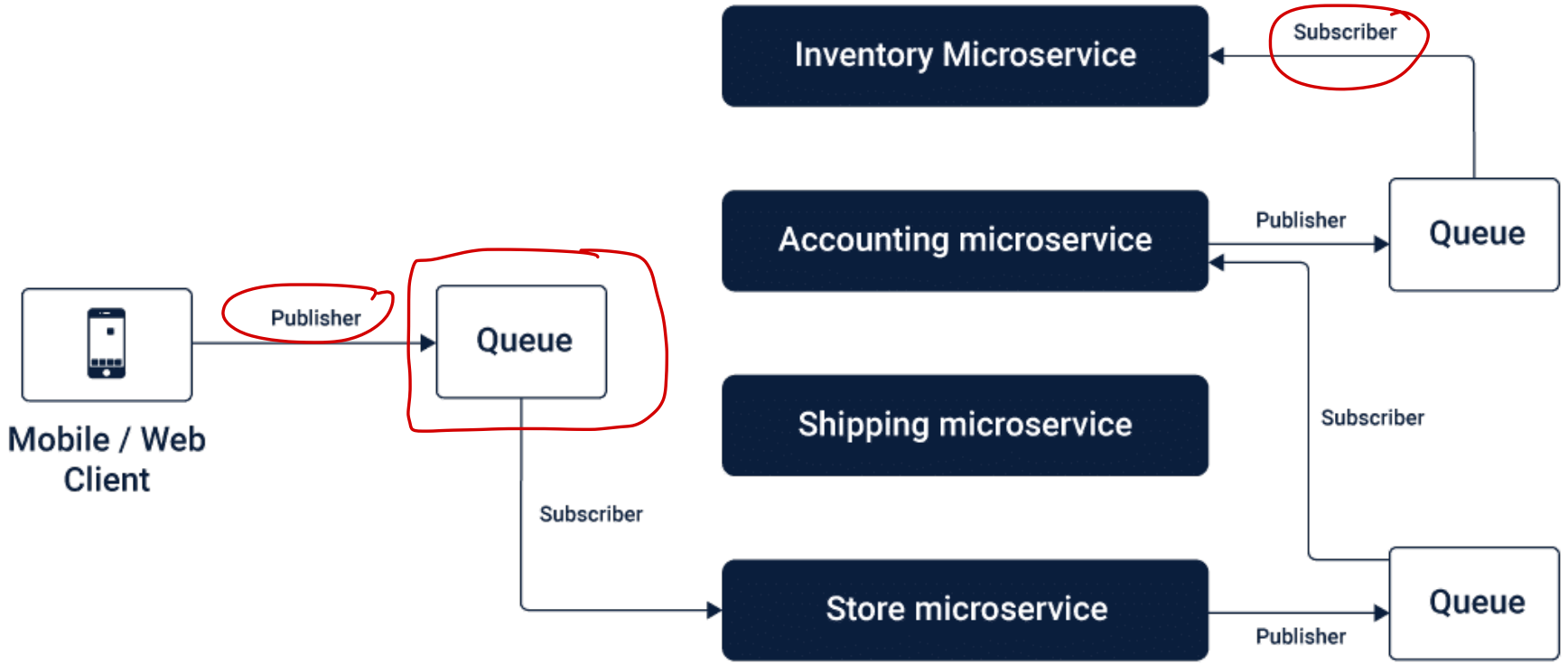
# API Gateway Inter-services Communication

- In this method of inter-services communication, a lightweight message gateway acts as the main entry point for all the client or consumers, where the standard and non-functional requirements are addressed at the portal itself
- This helps in optimizing the architecture by reducing unnecessary couplings between the Microservices



# Message Broker Inter-services Communication

- In this method of inter-services communication, asynchronous messages with one-way requests and publish-subscribe approach are queued to connect the different Microservices as per the business logic.



# Deployment of Microservices

- Deployment is very crucial for the functioning of Microservices. Here are the points to keep in mind while deploying the Microservices:
  - Modularize the self-contained Microservices making them as standalone components that can be reused across the application using automation [Jenkins]
  - Connect the microservices using bindings that can be manipulated easily
  - Deploy the Microservices independently to ensure agility and lower the impact on the application
  - You can use Docker to deploy the Microservices efficiently. Each of the Microservices can be further broken down in processes, that can be run in separate Docker containers. These can be specified with Dockerfiles and Docker Compose configuration files
  - You can use provisioning tool such as Kubernetes to manage and run a cluster of Docker Containers across multiple hosts with co-location, service discovery and replication control feature making the deployment powerful and efficient in case of large scale deployments.

Kubernetes / Docker swarm

# Benefits

- **Agility.** Because microservices are deployed independently, it's easier to manage bug fixes and feature releases. You can update a service without redeploying the entire application, and roll back an update if something goes wrong. In many traditional applications, if a bug is found in one part of the application, it can block the entire release process. New features may be held up waiting for a bug fix to be integrated, tested, and published.
- **Small, focused teams**. A microservice should be small enough that a single feature team can build, test, and deploy it. Small team sizes promote greater agility. Large teams tend be less productive, because communication is slower, management overhead goes up, and agility diminishes.
- **Small code base**. In a monolithic application, there is a tendency over time for code dependencies to become tangled. Adding a new feature requires touching code in a lot of places. By not sharing code or data stores, a microservices architecture minimizes dependencies, and that makes it easier to add new features.
- **Mix of technologies**. Teams can pick the technology that best fits their service, using a mix of technology stacks as appropriate.

# Benefits

- **Fault isolation**. If an individual microservice becomes unavailable, it won't disrupt the entire application, as long as any upstream microservices are designed to handle faults correctly (for example, by implementing circuit breaking).
- **Scalability**. Services can be scaled independently, letting you scale out subsystems that require more resources, without scaling out the entire application. Using an orchestrator such as Kubernetes or Service Fabric, you can pack a higher density of services onto a single host, which allows for more efficient utilization of resources.
- **Data isolation**. It is much easier to perform schema updates, because only a single microservice is affected. In a monolithic application, schema updates can become very challenging, because different parts of the application may all touch the same data, making any alterations to the schema risky.

# Challenges

- **Complexity**. A microservices application has more moving parts than the equivalent monolithic application. Each service is simpler, but the entire system as a whole is more complex.
- **Development and testing**. Writing a small service that relies on other dependent services requires a different approach than a writing a traditional monolithic or layered application. Existing tools are not always designed to work with service dependencies. Refactoring across service boundaries can be difficult. It is also challenging to test service dependencies, especially when the application is evolving quickly.
- **Lack of governance**. The decentralized approach to building microservices has advantages, but it can also lead to problems. You may end up with so many different languages and frameworks that the application becomes hard to maintain. It may be useful to put some project-wide standards in place, without overly restricting teams' flexibility. This especially applies to cross-cutting functionality such as logging.
- **Data integrity**. With each microservice responsible for its own data persistence. As a result, data consistency can be a challenge. Embrace eventual consistency where possible.

# Challenges

- **Network congestion and latency**. The use of many small, granular services can result in more interservice communication. Also, if the chain of service dependencies gets too long (service A calls B, which calls C...), the additional latency can become a problem. You will need to design APIs carefully. Avoid overly chatty APIs, think about serialization formats, and look for places to use asynchronous communication patterns like queue-based load levelling.
- **Management**. To be successful with microservices requires a mature DevOps culture. Correlated logging across services can be challenging. Typically, logging must correlate multiple service calls for a single user operation.
- **Versioning**. Updates to a service must not break services that depend on it. Multiple services could be updated at any given time, so without careful design, you might have problems with backward or forward compatibility.
- **Skill set**. Microservices are highly distributed systems. Carefully evaluate whether the team has the skills and experience to be successful.

# Best practices

- Model services around the business domain.
- Decentralize everything. Individual teams are responsible for designing and building services. Avoid sharing code or data schemas.
- Data storage should be private to the service that owns the data. Use the best storage for each service and data type.
- Services communicate through well-designed APIs. Avoid leaking implementation details. APIs should model the domain, not the internal implementation of the service.
- Avoid coupling between services. Causes of coupling include shared database schemas and rigid communication protocols.
- Offload cross-cutting concerns, such as authentication and SSL termination, to the gateway.
- Keep domain knowledge out of the gateway. The gateway should handle and route client requests without any knowledge of the business rules or domain logic. Otherwise, the gateway becomes a dependency and can cause coupling between services.

# Best Practices

- Services should have loose coupling and high functional cohesion. Functions that are likely to change together should be packaged and deployed together. If they reside in separate services, those services end up being tightly coupled, because a change in one service will require updating the other service. Overly chatty communication between two services may be a symptom of tight coupling and low cohesion.
- Isolate failures. Use resiliency strategies to prevent failures within a service from cascading

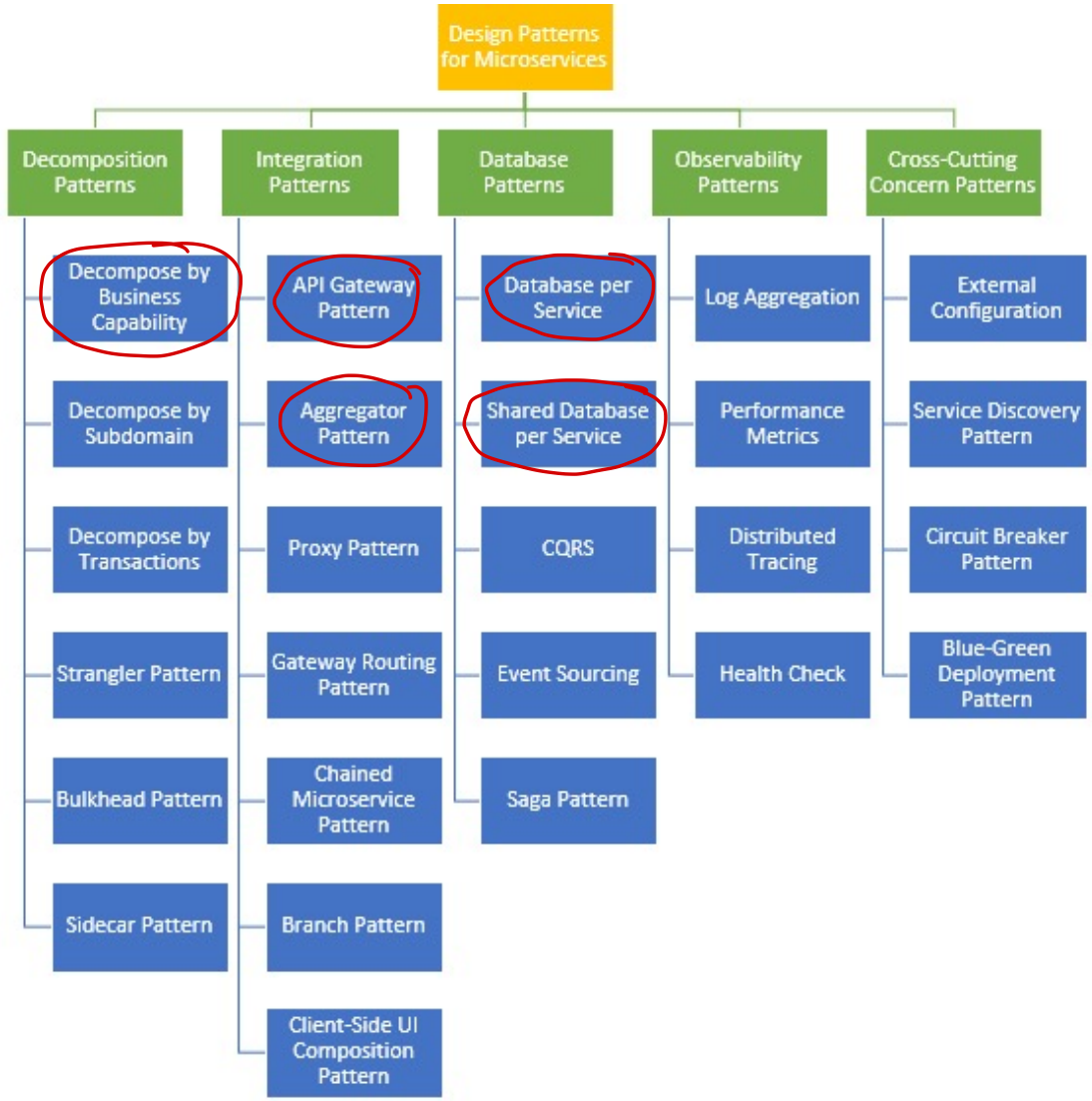
# When to choose ?

- **Microservices expertise**
  - Without proper skills and knowledge, building a microservice application is extremely risky
  - Still, just having the architecture knowledge is not enough. You need to have DevOps and Containers experts since the concepts are tightly coupled with microservices
  - Also, domain modelling expertise is a must. Dealing with microservices means splitting the system into separate functionalities and dividing responsibilities.
- **A complex and scalable application**
  - The microservices architecture will make scaling and adding new capabilities to your application much easier. So if you plan to develop a large application with multiple modules and user journeys, a microservice pattern would be the best way to handle it
- **Enough engineering skills**
  - Since a microservice project comprises multiple teams responsible for multiple services, you need to have enough resources to handle all the processes

# SOA vs Microservices

| | Microservices | SOA |
|---|---|---|
| **Architecture** | Designed to host services which can function independently | Designed to share resources across services |
| **Component sharing** | Typically does not involve component sharing | Frequently involves component sharing |
| **Granularity** | Fine-grained services | Larger, more modular services |
| **Data storage** | Each service can have an independent data storage | Involves sharing data storage between services |
| **Governance** | Requires collaboration between teams | Common governance protocols across teams |
| **Size and scope** | Better for smaller and web-based applications | Better for large scale integrations |
| **Communication** | Communicates through an API layer | Communicates through an ESB |
| **Coupling and cohesion** | Relies on bounded context for coupling | Relies on sharing resources |
| **Remote services** | Uses REST and JMS | Uses protocols like SOAP and AMQP |
| **Deployment** | Quick and easy deployment | Less flexibility in deployment |

# Patterns


Design Patterns for Microservices
Decomposition Patterns
Integration Patterns
Database Patterns
Observability Patterns
Cross-Cutting Concern Patterns
Decompose by Business Capability
Decompose by Subdomain
Decompose by Transactions
Strangler Pattern
Bulkhead Pattern
Sidecar Pattern
API Gateway Pattern
Aggregator Pattern
Proxy Pattern
Gateway Routing Pattern
Chained Microservice Pattern
Branch Pattern
Client-Side UI Composition Pattern
Database per Service
Shared Database per Service
CQRS
Event Sourcing
Saga Pattern
Log Aggregation
Performance Metrics
Distributed Tracing
Health Check
External Configuration
Service Discovery Pattern
Circuit Breaker Pattern
Blue-Green Deployment Pattern