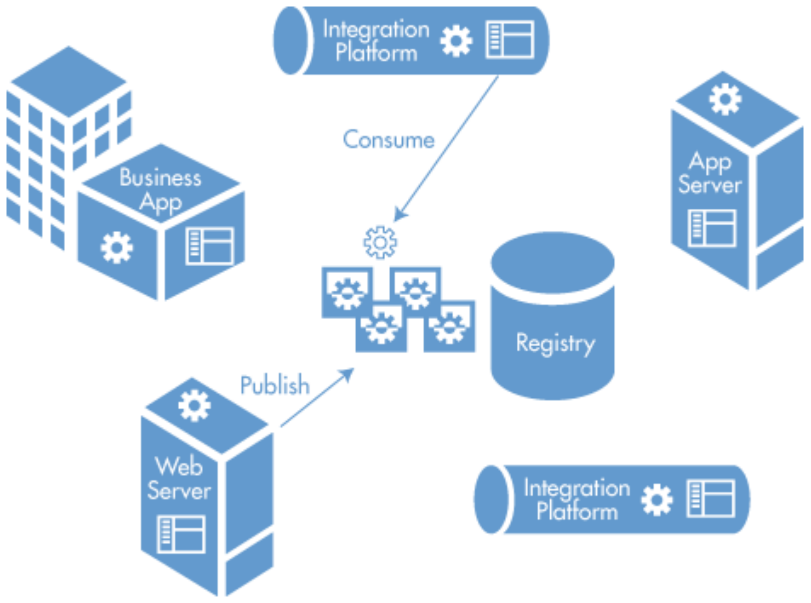
UNIVERSITÀ
DI PARMA

Service-Oriented Architectures

Microservices



Tecnologie Internet
a.a. 2022/2023

Outline

- The Monolith
- What are Microservices?
- How to Model Microservices
- Microservice Communication styles
- Implementing Microservice Communication
- Workflow
- Build
- Deployment

The Monolith

When all functionality in a system must be deployed together, we consider it a monolith.

In a **single-process monolith**, all code is packaged into a single process.

The **modular monolith** is a variation in which the single process consists of separate modules. Each can be worked on independently, but all still need to be combined together for deployment.

A **distributed monolith** is a system that consists of multiple services, but for whatever reason, the entire system must be deployed together.

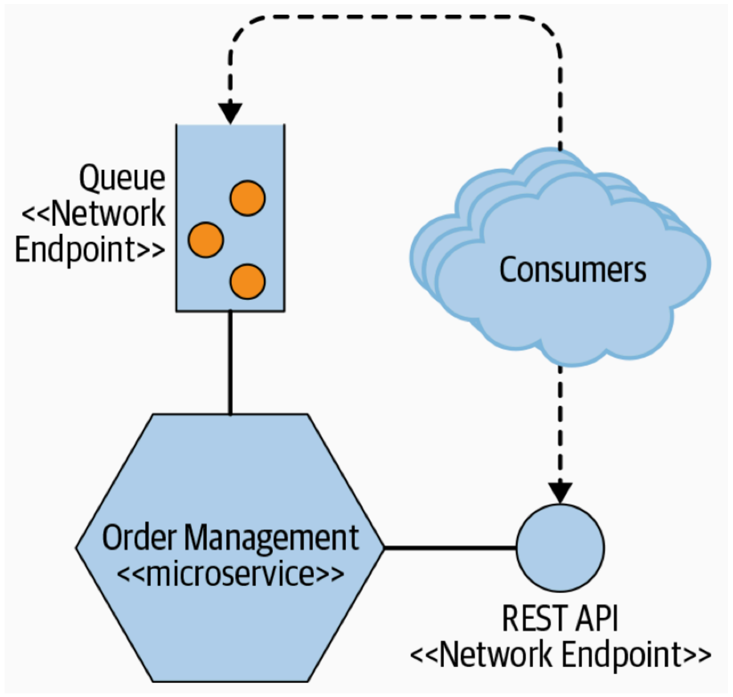
A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.

-- Leslie Lamport

What are Microservices?

Microservices are independently releasable services that are modelled around a business domain.

A service encapsulates functionality and makes it accessible to other services by exposing one ore more network endpoints (e.g., a queue or a REST API).



What are Microservices?

"The microservice approach has emerged from real-world use, taking our better understanding of systems and architecture to do SOA well. You should think of microservices as a specific approach for SOA in the same way that Extreme Programming (XP) or Scrum are specific approaches for Agile software development."

Excerpt From: Sam Newman. "Building Microservices, 2nd Edition".

What are Microservices?

From the outside, a single microservice is treated as a **black box**, hiding as much as possible and exposing as little as possible via external interfaces.

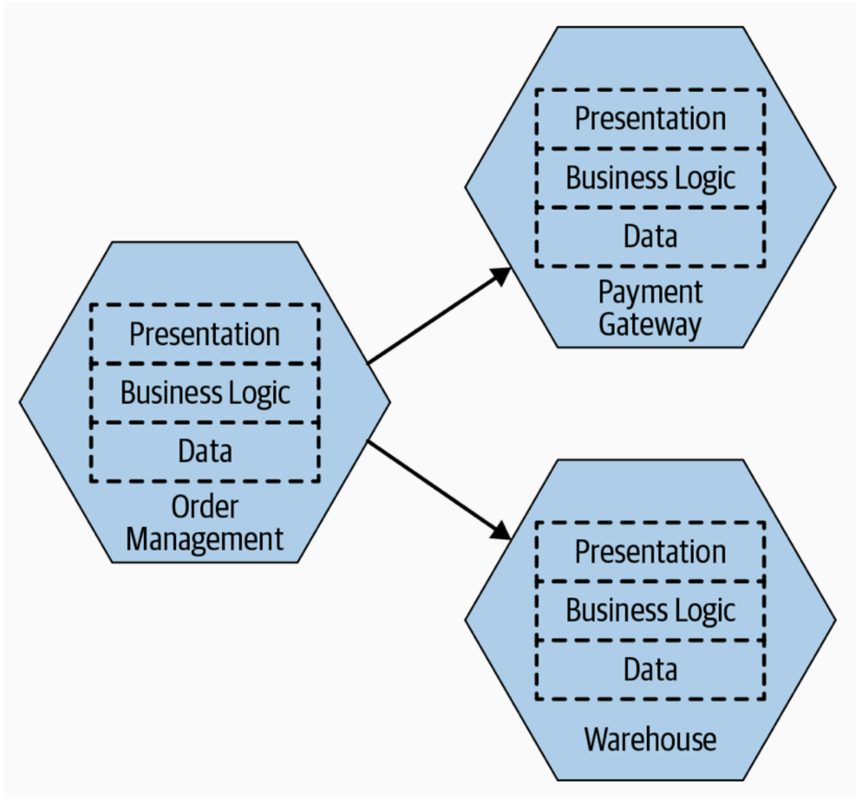
Internal implementation details are entirely hidden from the outside world. This means microservice architectures avoid the use of shared databases in most circumstances; instead, each microservice encapsulates its own database where required.

Having clear, **stable service boundaries** that do not change when the internal implementation changes results in systems that have **looser coupling** and **stronger cohesion**.

What are Microservices?

Independent deployability is the idea that we can make a change to a microservice, deploy it, and release that change to the users, without having to deploy any other services.

By **modelling services around business domains**, we can make it easier to roll out new functionality, and make it easier to recombine microservices in different ways to deliver new functionality to the users.



What are Microservices?

How big should a microservice be?

There is no ultimate answer.

Microservices should have as small an interface as possible.

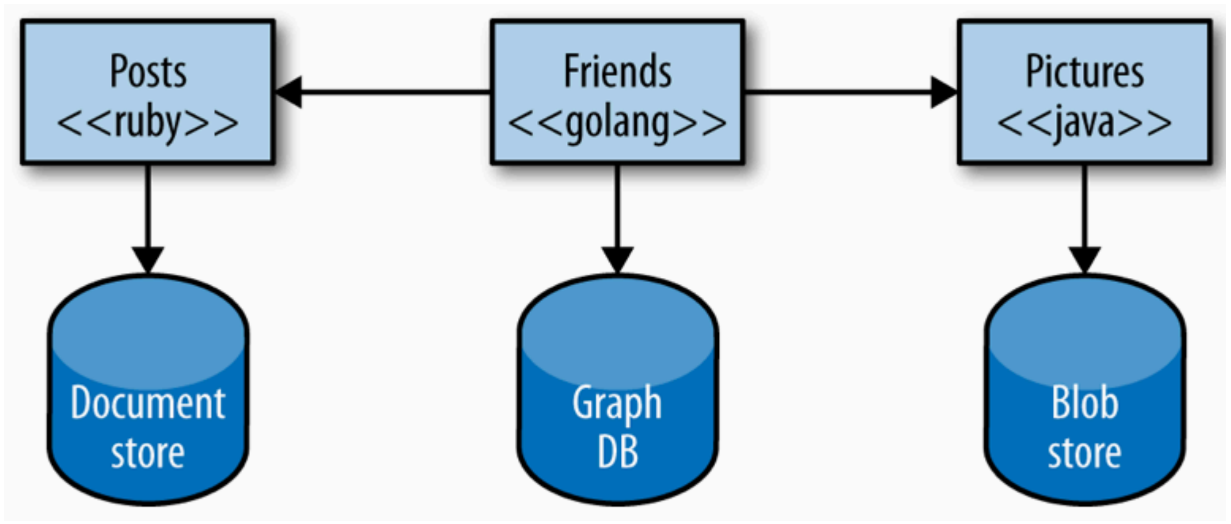
However, having too many "small" microservices may result in increased system complexity, requiring new skills and technologies.

What are Microservices?

A major advantage of microservices is that they support **technology heterogeneity**.

We can pick the right tool for each service implemention.

We can easily change technology of one microservice, without affecting the other microservices.



How to Model Microservices

Using the **domain** as the primary mechanism for identifying boundaries for microservices makes the most sense in general.

Cohesion

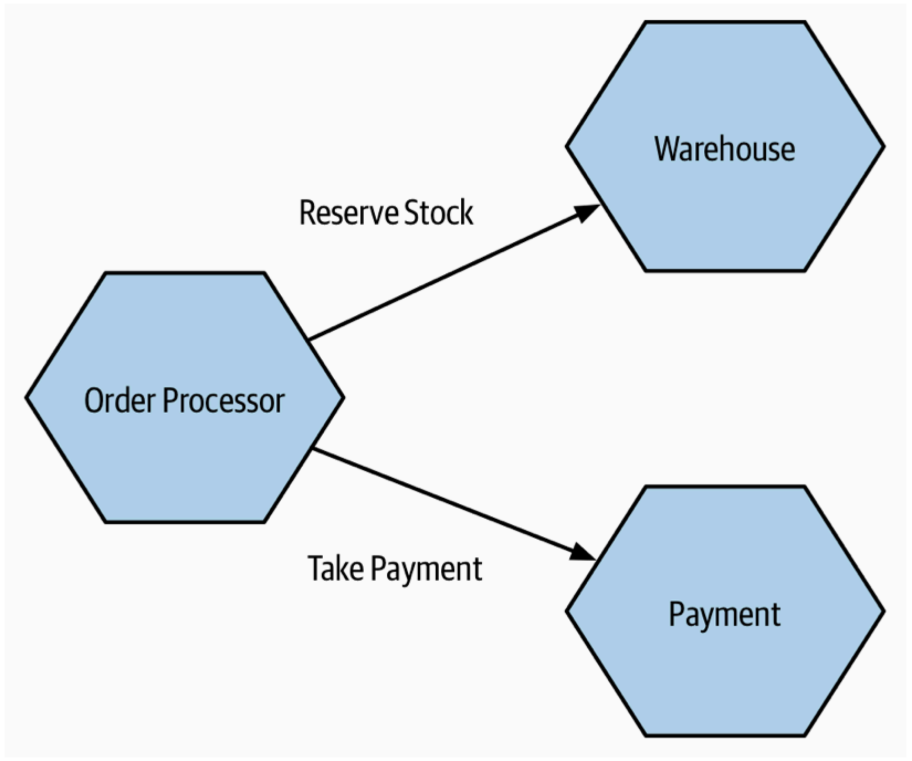
We want the functionality grouped in such a way that we can make changes in as few places as possible.

Loose Coupling

The whole point of a microservice is being able to make a change to one service and deploy it, without needing to change any other part of the system.

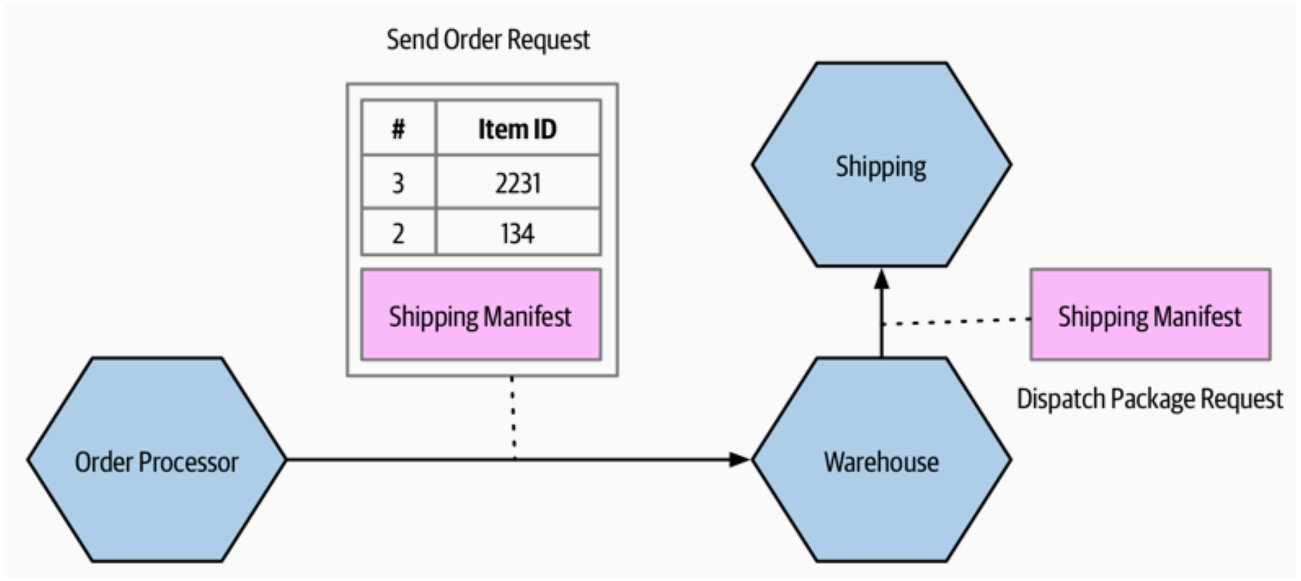
How to Model Microservices

Domain coupling describes the situation where one microservice needs to interact with another microservice, because it needs to make use of the functionality that the other microservice provides.



How to Model Microservices

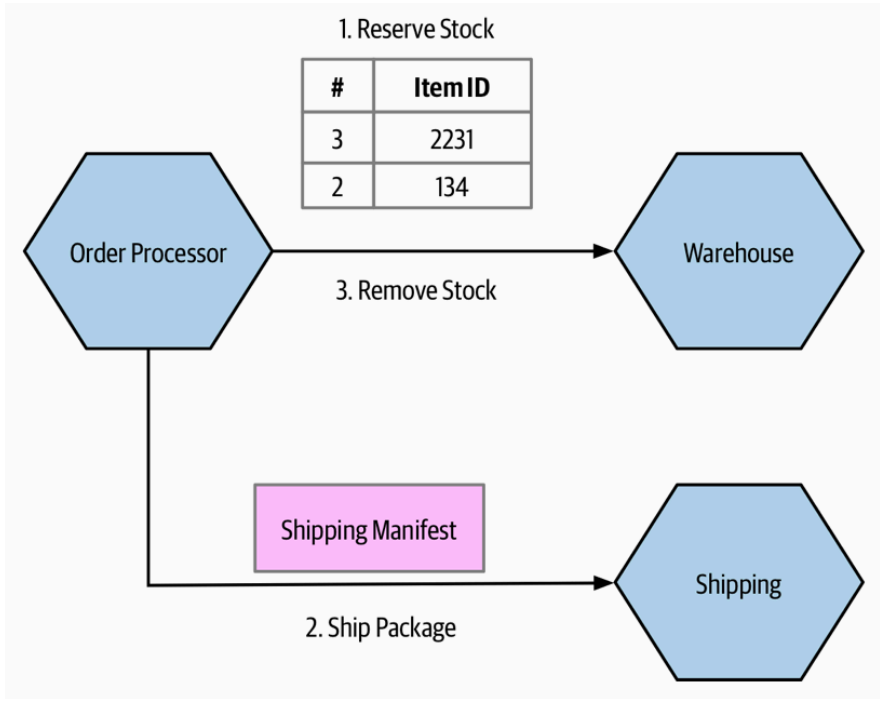
Pass through coupling describes a situation where one microservice passes data to another microservice purely because it is needed by some other further downstream microservice.



How to Model Microservices

The major issue with pass through coupling is that a change to the required data downstream can cause a more significant upstream change.

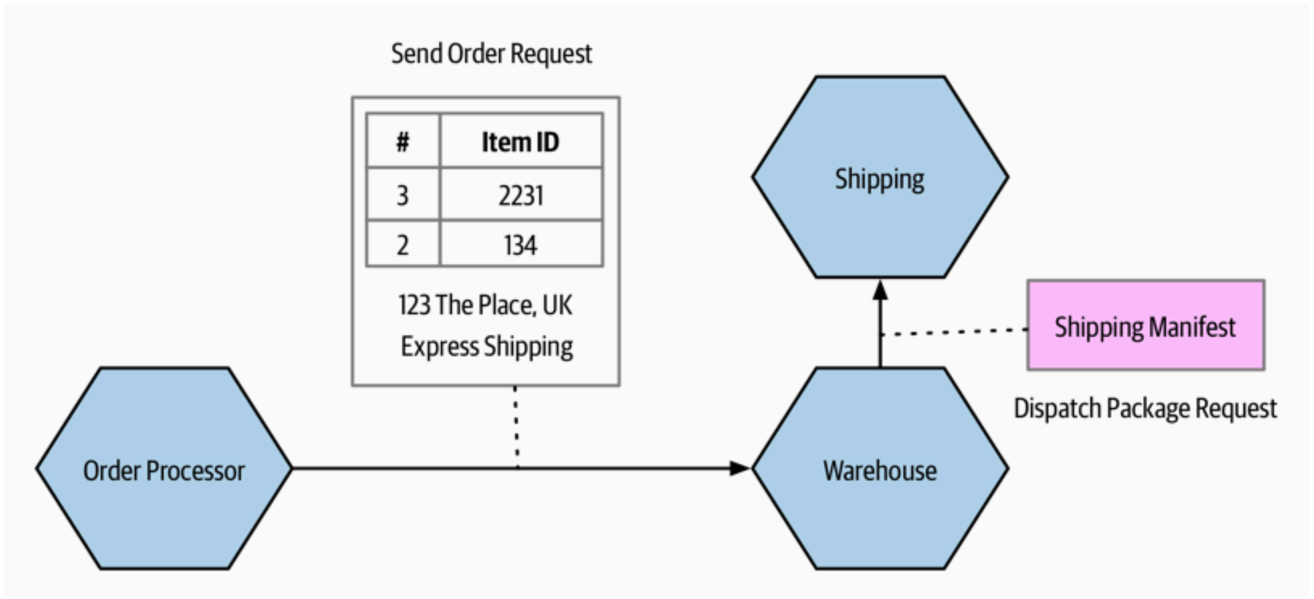
Solution 1: communicate directly with the downstream service



How to Model Microservices

The major issue with pass through coupling is that a change to the required data downstream can cause a more significant upstream change.

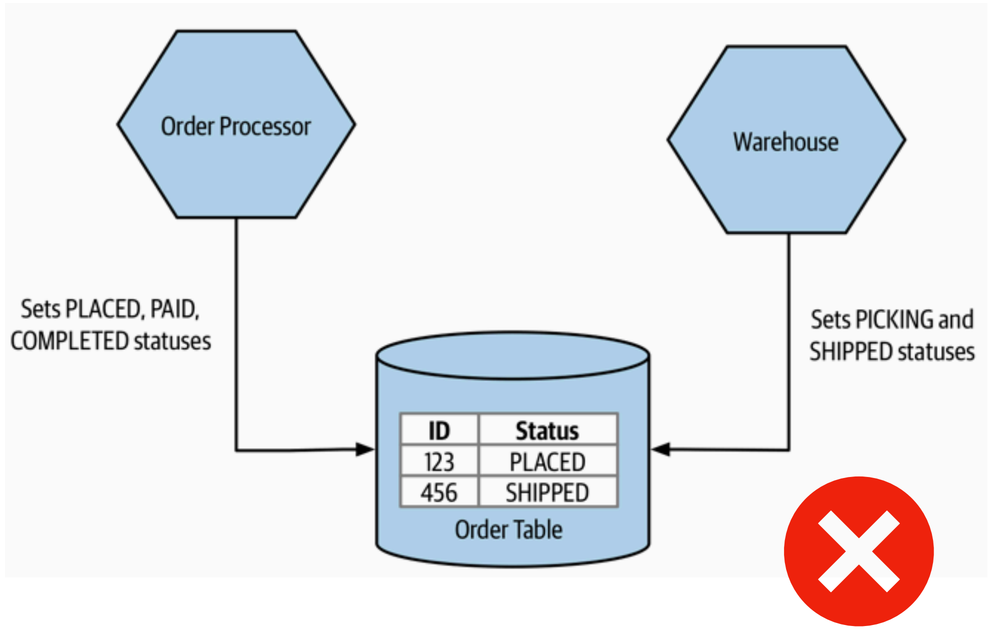
Solution 2: avoid unnecessary message propagations

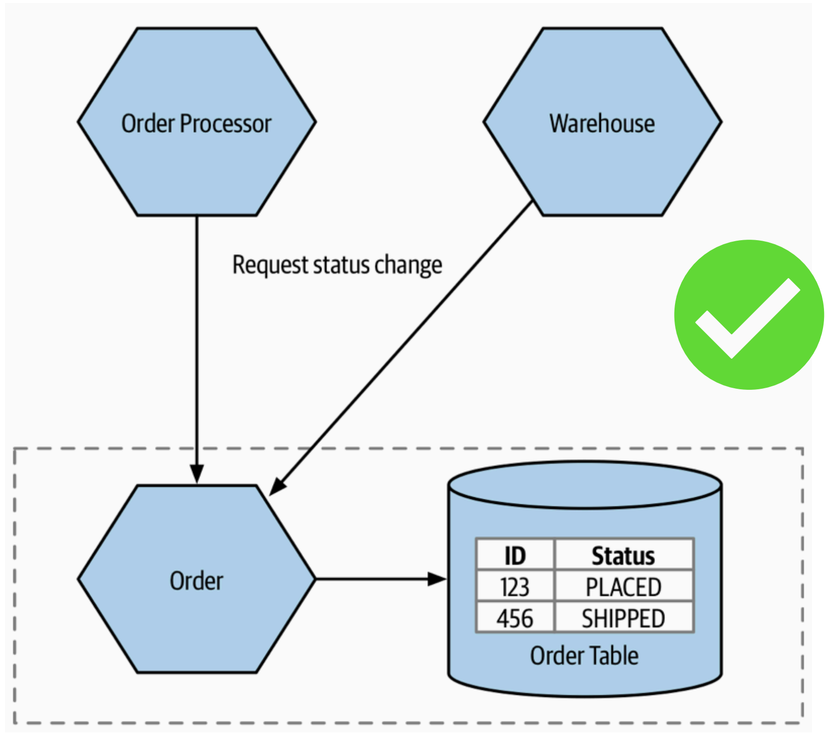


How to Model Microservices

Common coupling occurs when two or more microservices make use of a common set of data.

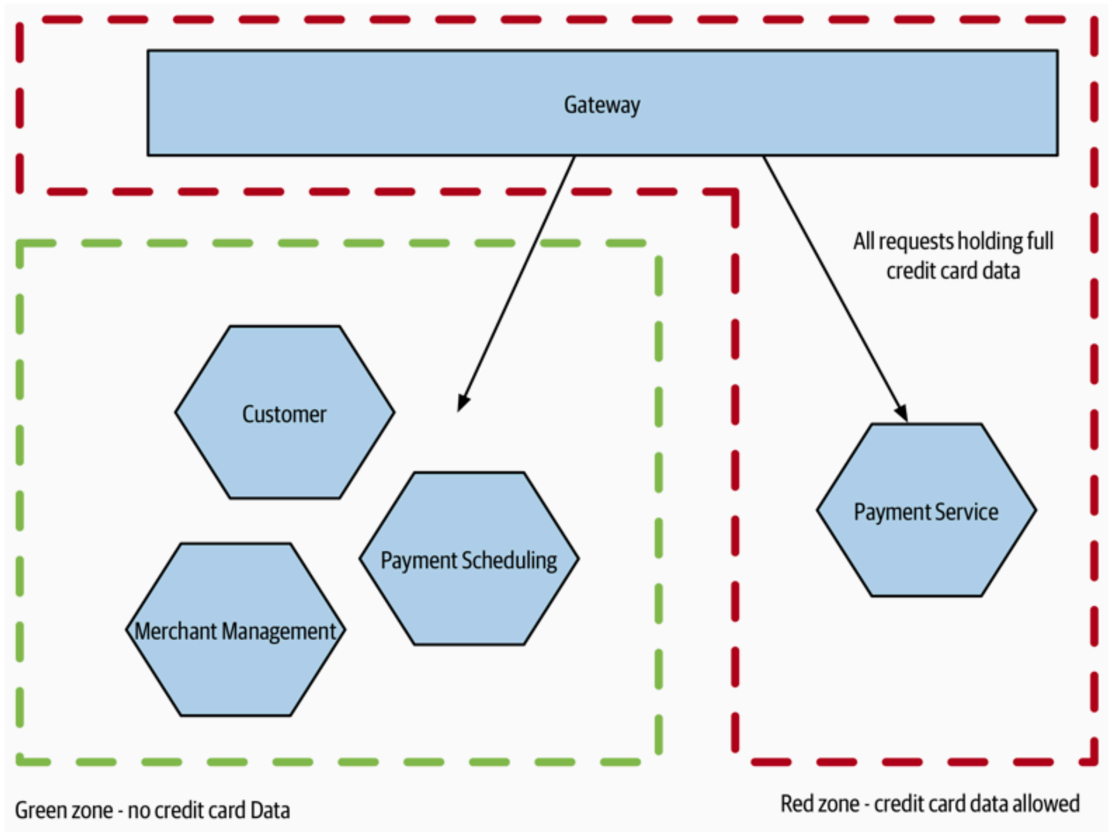
We need to avoid using the same database!





How to Model Microservices

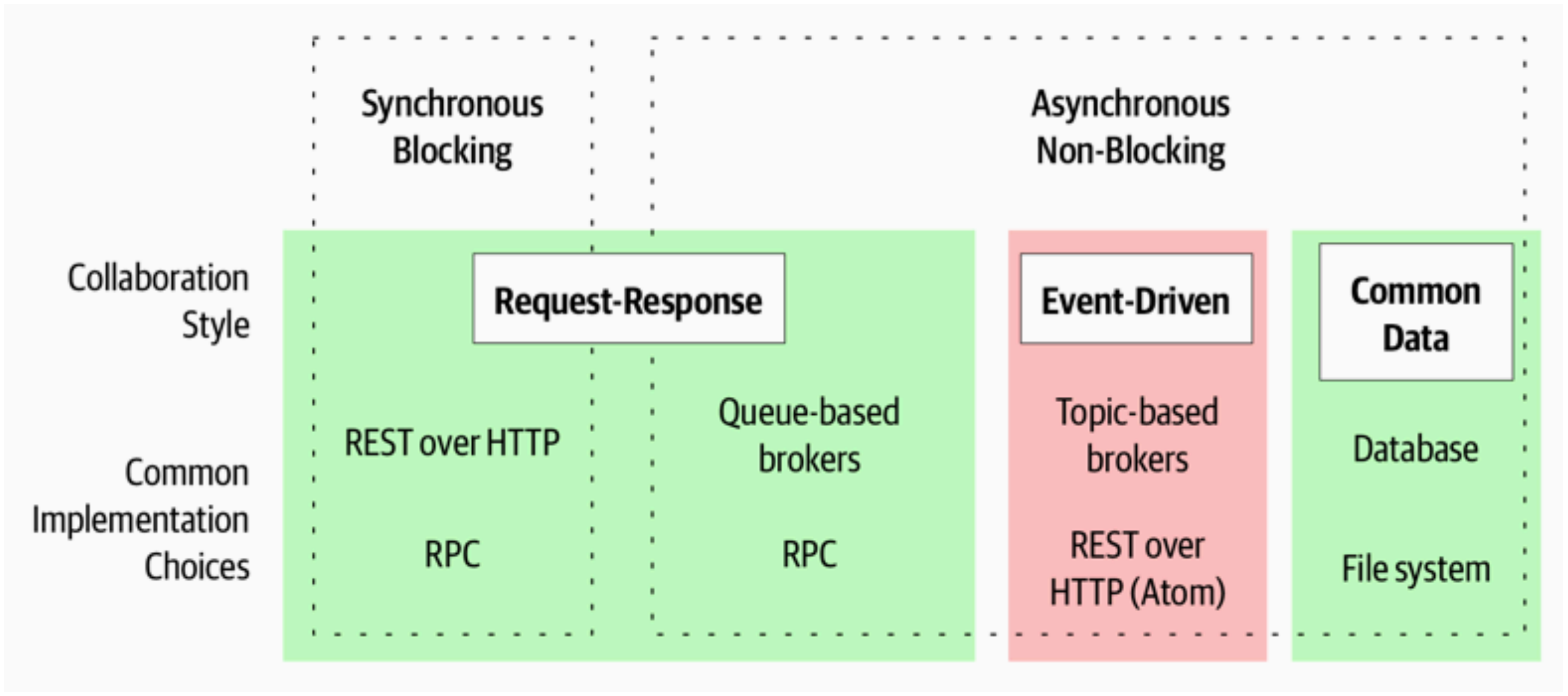
The nature of the data you hold and manage can drive you towards different forms of decomposition.



Gateway
All requests holding full credit card data
Customer
Payment Scheduling
Merchant Management
Payment Service
Green zone - no credit card Data
Red zone - credit card data allowed

UNIVERSITÀ DI PARMA

Microservice Communication Styles



Microservice Communication Styles

Synchronous Blocking: a microservice makes a call to another microservice and blocks operation waiting for the response.

Asynchronous Non-Blocking: the microservice emitting a call is able to carry on processing whether or not the call is received.

Request-Response: a microservice sends a request to another microservice asking for something to be done. It expects to receive a response to the request informing it of the result.

Event-Driven: microservices emit events, which other microservices consume and react to accordingly. The microservice emitting the event is unaware of which microservices, if any, consume the events it emits.

Common Data: miscroservices collaborate via some shared data source.

Implementing Microservice Communication

Your choice of technology should be driven in large part based on the style of communication you want.

Good practices:

- make backward compatibility easy
- make your interface explicit
- keep your APIs technology-agnostic
- make your service simple for consumers
- hide internal implementation detail

Implementing Microservice Communication

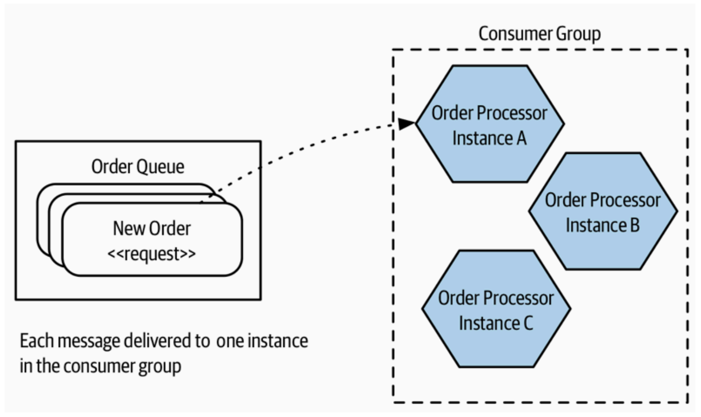
Technologies:

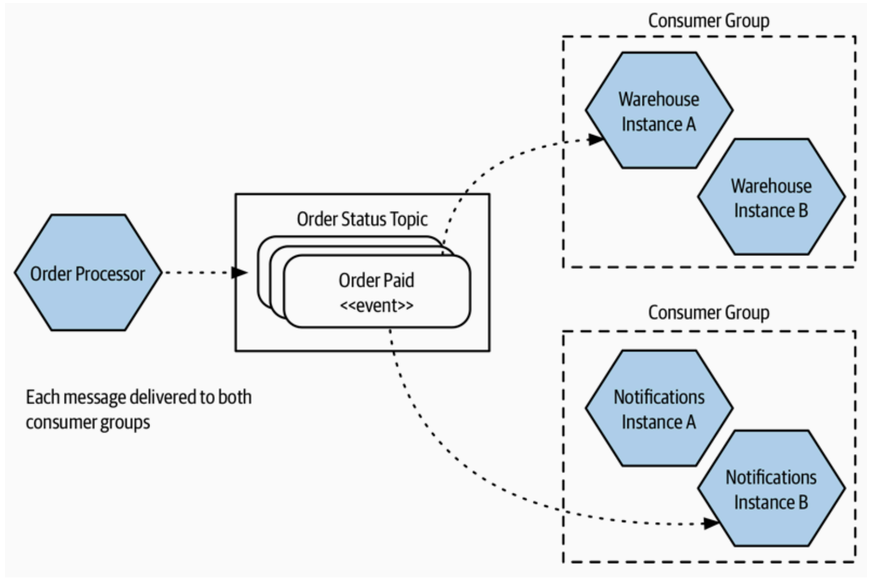
- **Remote Procedure Calls** (local method calls are invoked on a remote process)
  - SOAP https://www.w3.org/TR/soap12-part0/
  - gRPC https://grpc.io/
  - CORBA https://www.corba.org/
  - ..
- **REST** (resources can be accessed through simple methods, e.g. HTTP ones)
- **GraphQL** (custom queries can fetch information from multiple downstream microservices) https://graphql.org/
- **Message Brokers** (middleware for asynchronous communication either via queues or topics)

Implementing Microservice Communication

For synchronous request-response:
- gRPC
- REST over HTTP
- GraphQL

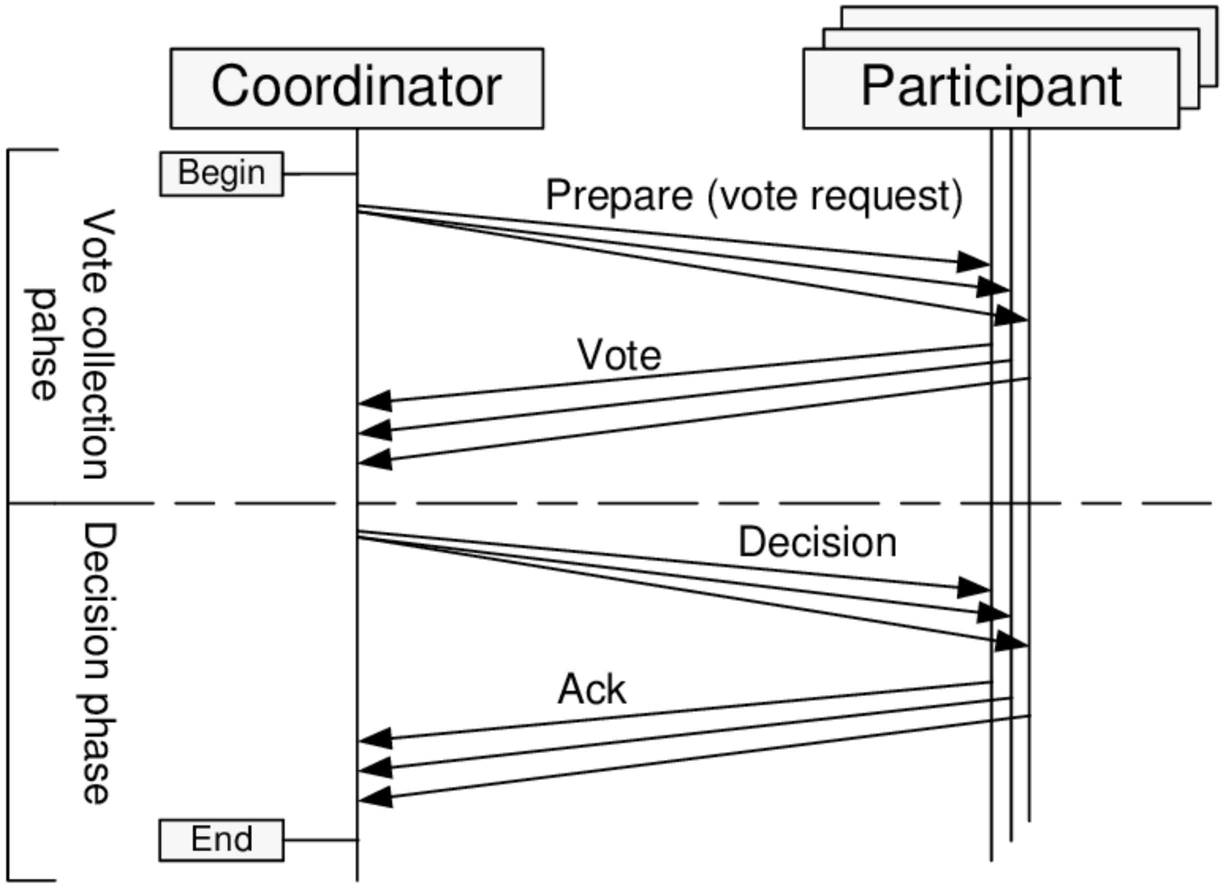
For asynchronous communications:
- Message Brokers (e.g., RabbitMQ, ActiveMQ, Kafka)


Consumer Group
Order Queue
New Order
<<request>>
Order Processor Instance A
Order Processor Instance B
Order Processor Instance C
Each message delivered to one instance in the consumer group



Workflow

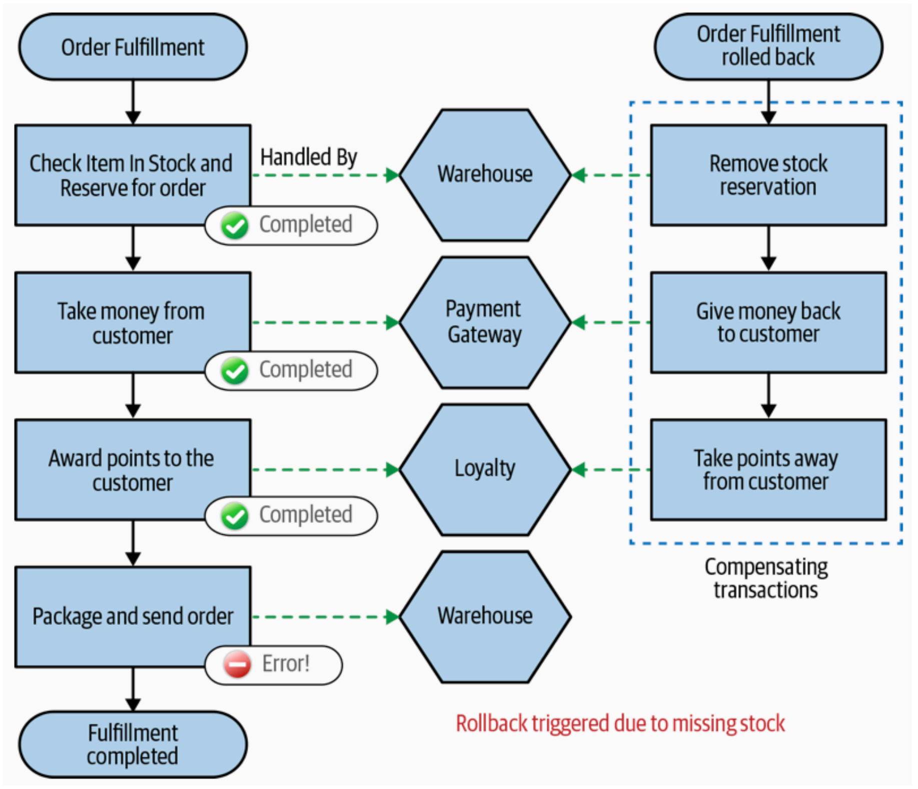
Anti-Pattern: Two-Phase Commit (2PC) Algorithm



Workflow

Two-phase commits (and, in general, distributed transactions) must be avoided, because they can be a quick way to inject huge amounts of latency into the system.

Pattern: Sagas


Order Fulfillment
Check Item In Stock and Reserve for order
Handled By
Completed
Take money from customer
Completed
Award points to the customer
Completed
Package and send order
Error!
Fulfillment completed
Warehouse
Payment Gateway
Loyalty
Warehouse
Order Fulfillment rolled back
Remove stock reservation
Give money back to customer
Take points away from customer
Compensating transactions
Rollback triggered due to missing stock

Build

- **Continuous Integration (CI)**

Make sure that newly checked-in code properly integrates with existing code.

To do this, a CI server detects that the code has been committed, checks it out, and carries out some verification like making sure the code compiles and that tests pass.

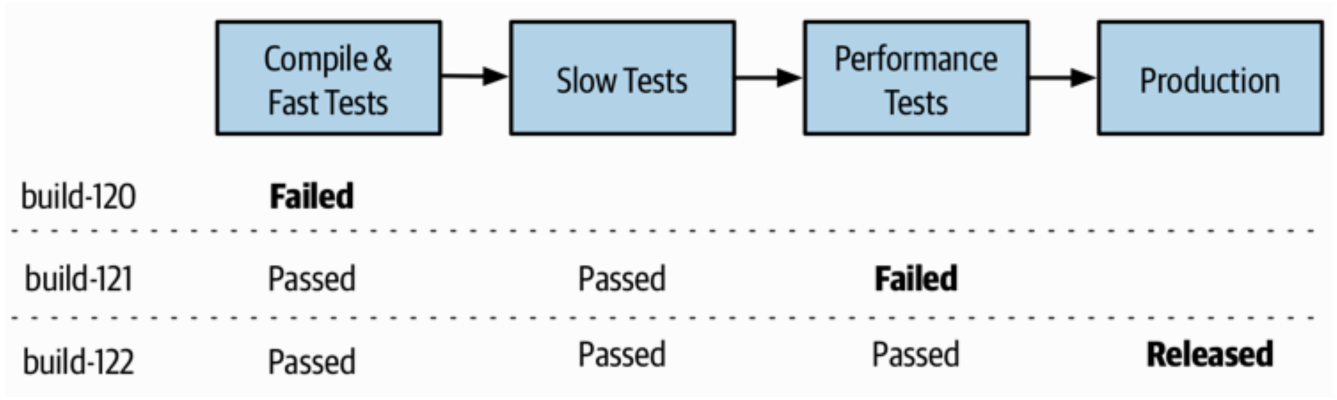
As a bare minimum, we expect this integration to be done on a daily basis.

Avoid the use of long-lived branches for feature development, and consider **Trunk-Based Development** instead.
If you really have to use branches, keep them short!

Build

- **Build Pipelines and Continuous Delivery**

- have different stages in your build
- get constant feedback on the production readiness of each and every check-in
- furthermore treat each and every check-in as a release candidate

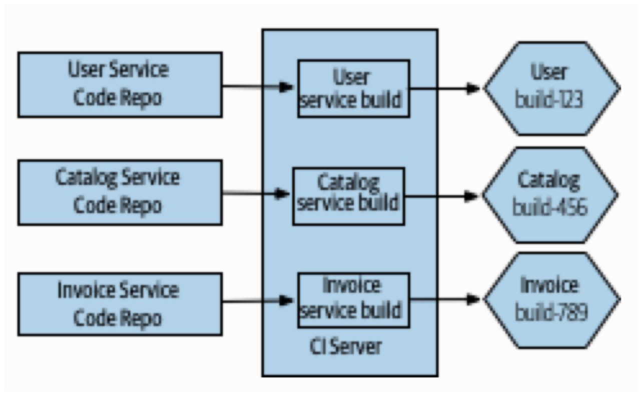


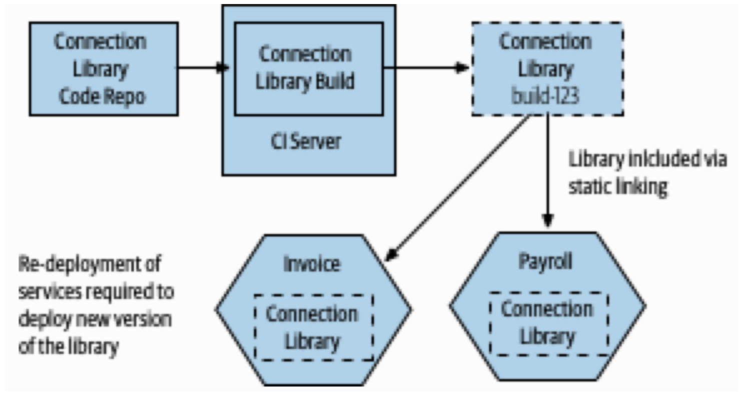
Build

- **One Repository per Microservice (aka Multi-Repo)**

The source code for each microservice is stored in a separate source code repository.

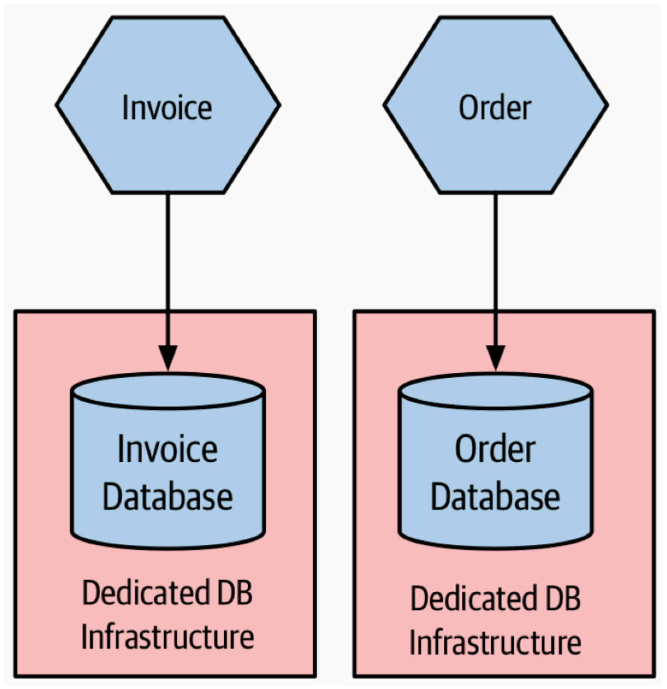
Have the code you want to reuse packaged into a library which then becomes an explicit dependency of the downstream microservices.




Connection Library Code Repo
Connection Library Build
CI Server
Connection Library build-123
Library included via static linking
Re-deployment of services required to deploy new version of the library
Invoice
Connection Library
Payroll
Connection Library

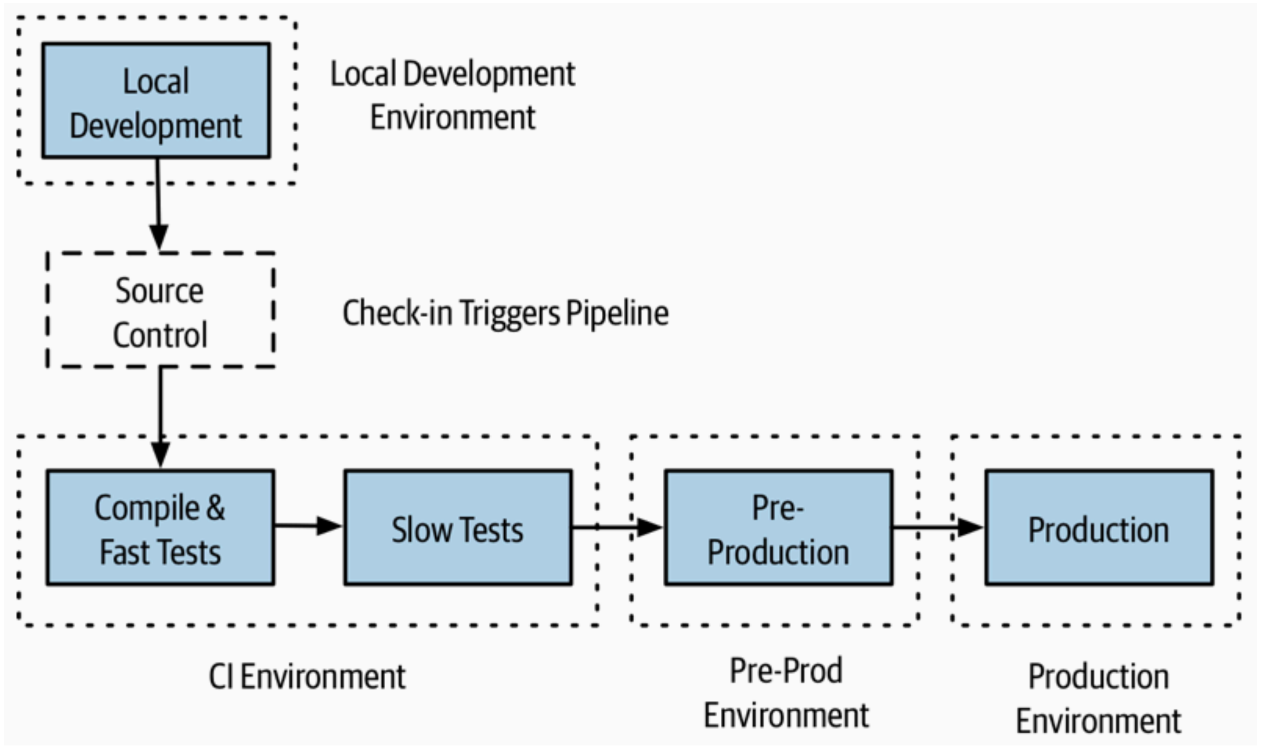
Deployment

Database deployment and scaling



Deployment

Different environments for different parts of the pipeline

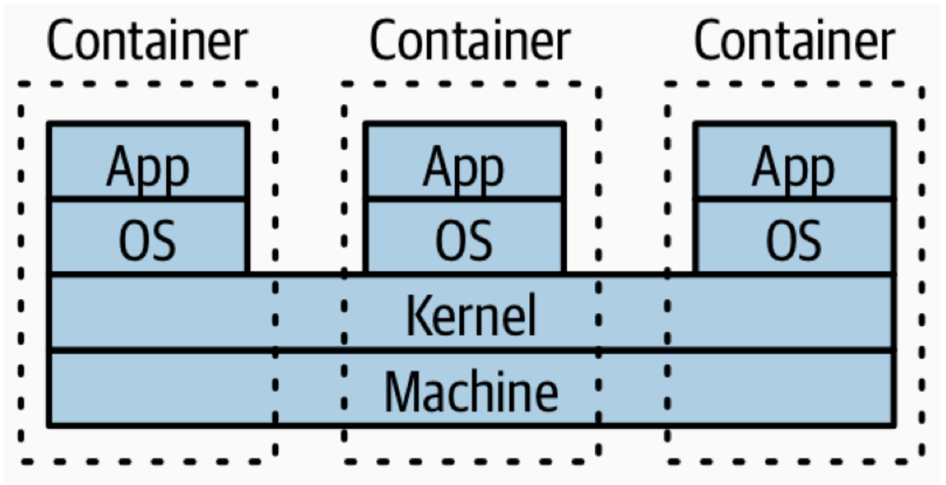


Deployment

Principles of microservice deployment:

- **Isolated Execution**
- **Focus on Automation**
- **Infrastructure as Code**
- **Zero-Downtime Deployment**
- **Desired State Management**

Deployment

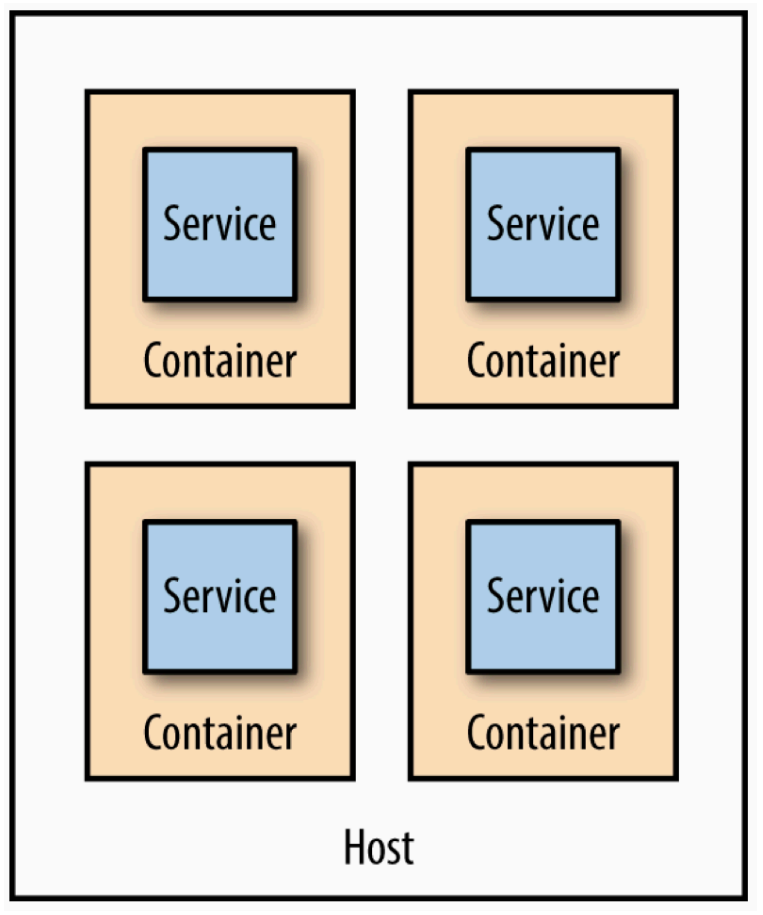


Deployment options:

- **Physical machine**
- **Virtual machine**
- **Container** (Docker + container orchestration tool like Kubernetes)
- **Application container**
- **Platform as a Service** (Heroku, Google App Engine, AWS Beanstalk)
- **Function as a Service** (AWS Lambda, Azure Cloud Functions)
  - serverless approach

Deployment

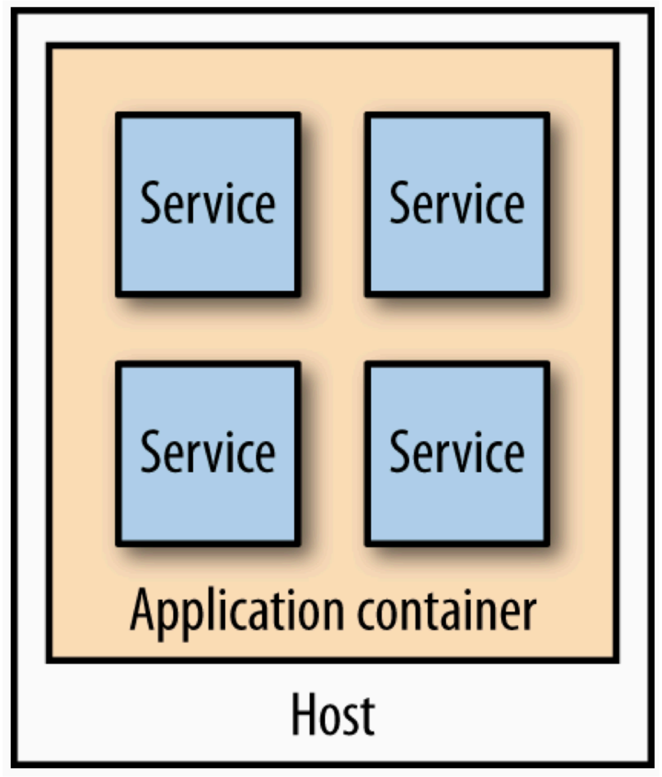
Running microservices in separate containers:



UNIVERSITÀ DI PARMA

Deployment

Multiple microservices per application container:



References

https://samnewman.io/books/building_microservices_2nd_edition/