

# Microservices





Somkiat Puisungnoen

Search

Somkiat | Home

Update Info 1 View Activity Log 10+ ...

Timeline About Friends 3,138 Photos More

When did you work at Opendream? X

... 22 Pending Items

Post Photo/Video Live Video Life Event

What's on your mind?

Public Post

Intro

Software Craftsmanship

Software Practitioner at สยามชานาญกิจ พ.ศ. 2556

Agile Practitioner and Technical at SPRINT3r

Somkiat Puisungnoen 15 mins · Bangkok · ...

Java and Bigdata

When did you work at Opendream? X

... 22 Pending Items

Post Photo/Video Live Video Life Event

What's on your mind?

Public Post

Intro

Software Craftsmanship

Software Practitioner at สยามชานาญกิจ พ.ศ. 2556

Agile Practitioner and Technical at SPRINT3r

Somkiat Puisungnoen 15 mins · Bangkok · ...

Java and Bigdata



Facebook somkiat.cc

Page Messages Notifications 3 Insights Publishing Tools Settings Help ▾

somkiat.cc  
@somkiat.cc

Home Posts Videos Photos

Liked Following Share ... + Add a Button

Help people take action on this Page. X



**[https://github.com/up1/  
course\\_microservices-3-days](https://github.com/up1/course_microservices-3-days)**



# Module 1 : Design

Cloud Native Application

Evolution of architecture

Microservice architecture

How to decompose app to Microservice

Communication between service

Data consistency

Workshop



# Module 2 : Develop

Recap Microservice

Properties of Microservice

Microservice 1.0 - 4.0

How to develop Microservice ?

How to test Microservice ?

12-factors app

Workshop



# Module 3 : Deploy

How to deploy Microservice ?  
Continuous Integration and Delivery  
Practices of Continuous Integration  
Deployment strategies  
Working with containerization (Docker)  
Workshop



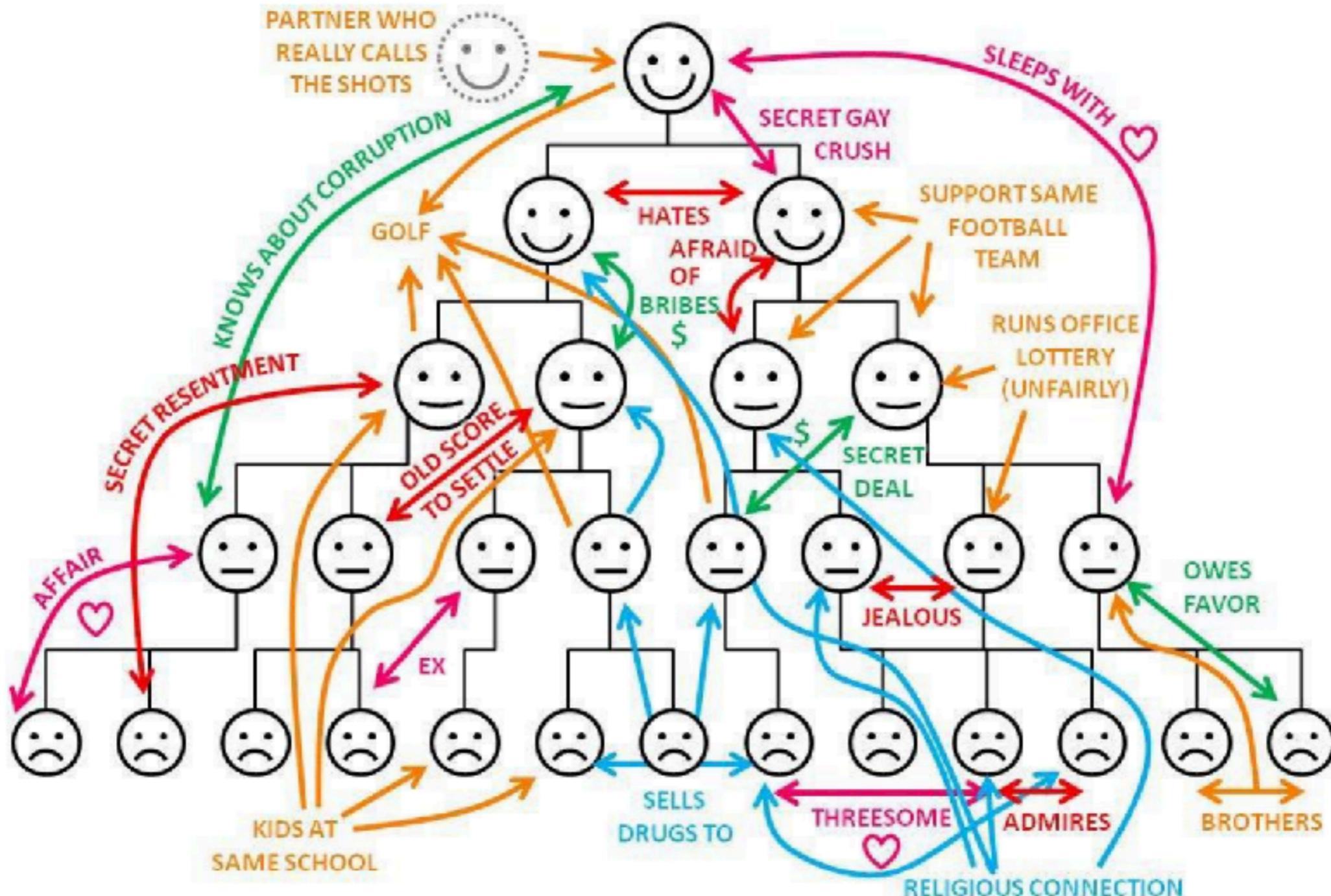
# Module 1 : Design

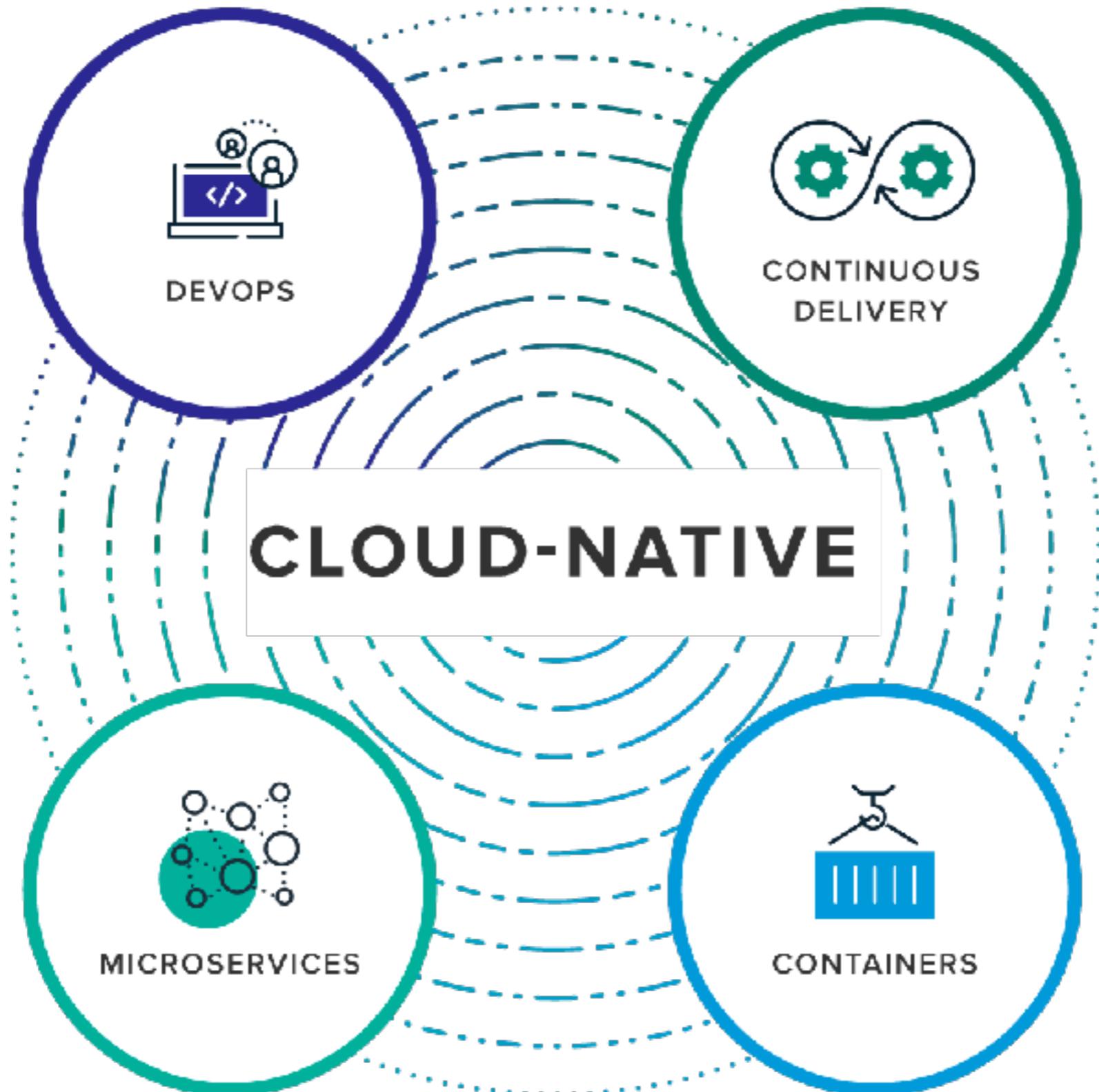


“organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations”

- *Melvin Conwey, 1968* -







<https://pivotal.io/cloud-native>



# Evolution of Architecture

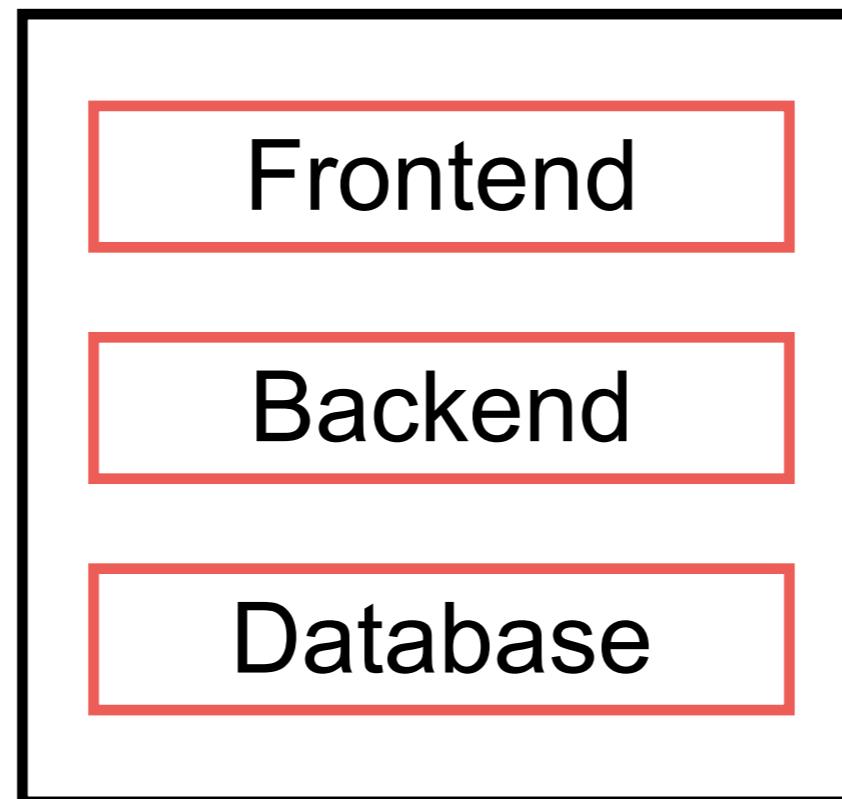


# Monolith

App



# Layer



# Layer

Frontend

Backend

Database

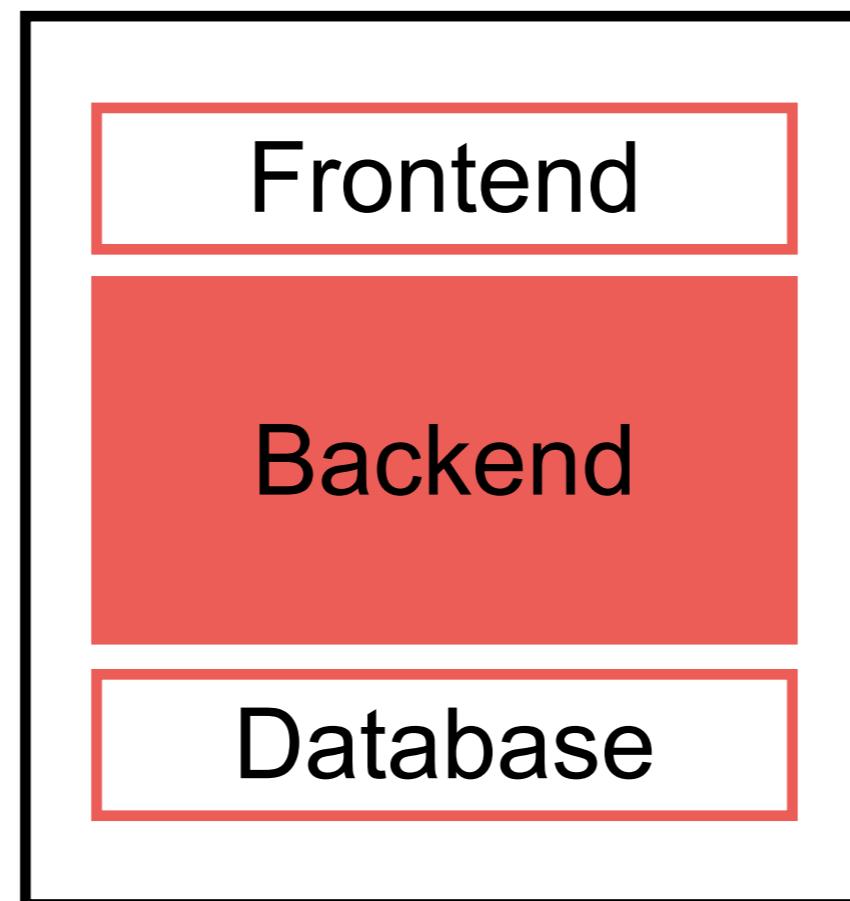


# Monolith

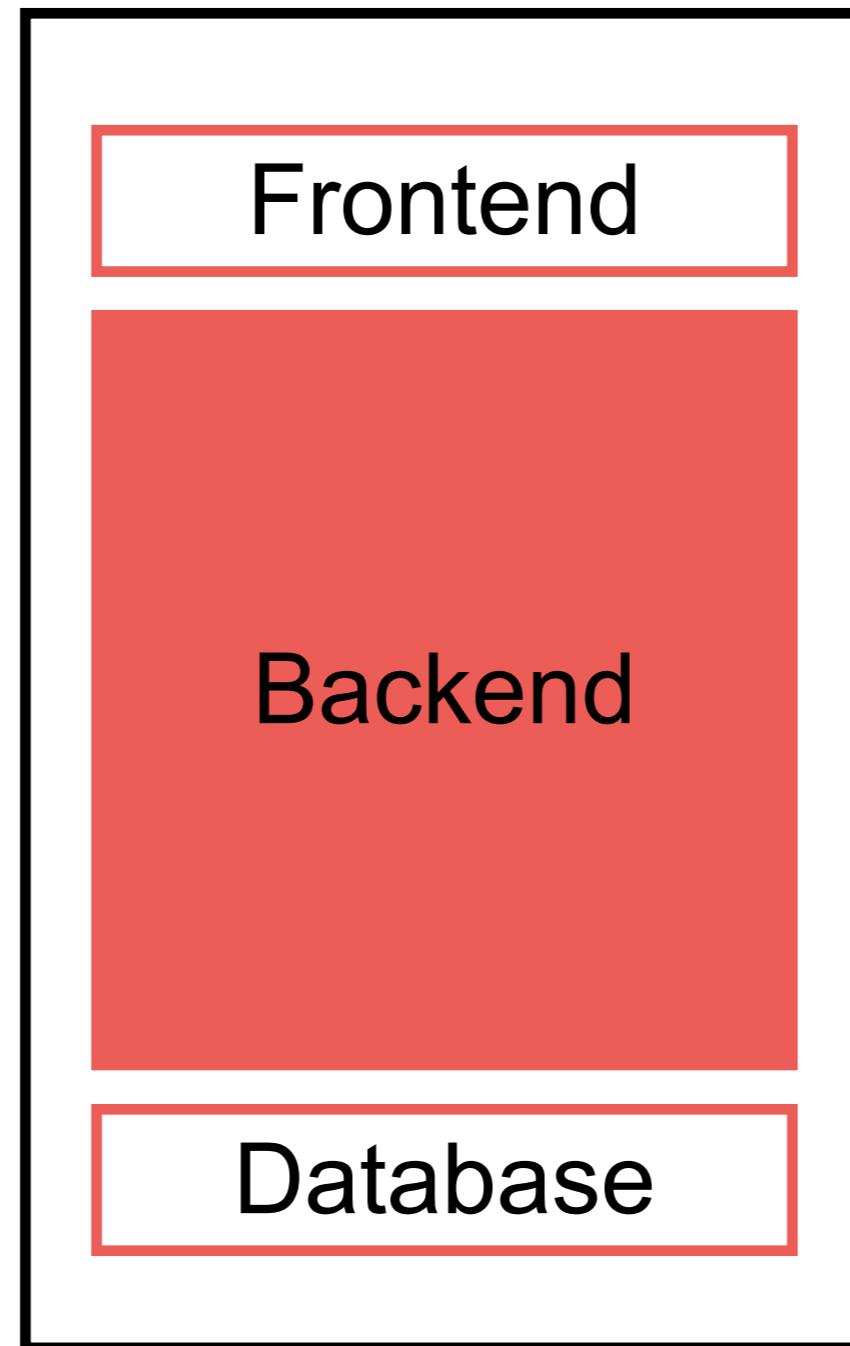
Easy to develop  
Easy to change  
Easy to testing  
Easy to deploy  
Easy to scale



# More features ...



# More features ...



# More features ...

Frontend



In spaghetti code, the relations between the pieces of code are so tangled that it is nearly impossible to add or change something without unpredictably breaking something somewhere else.

Database

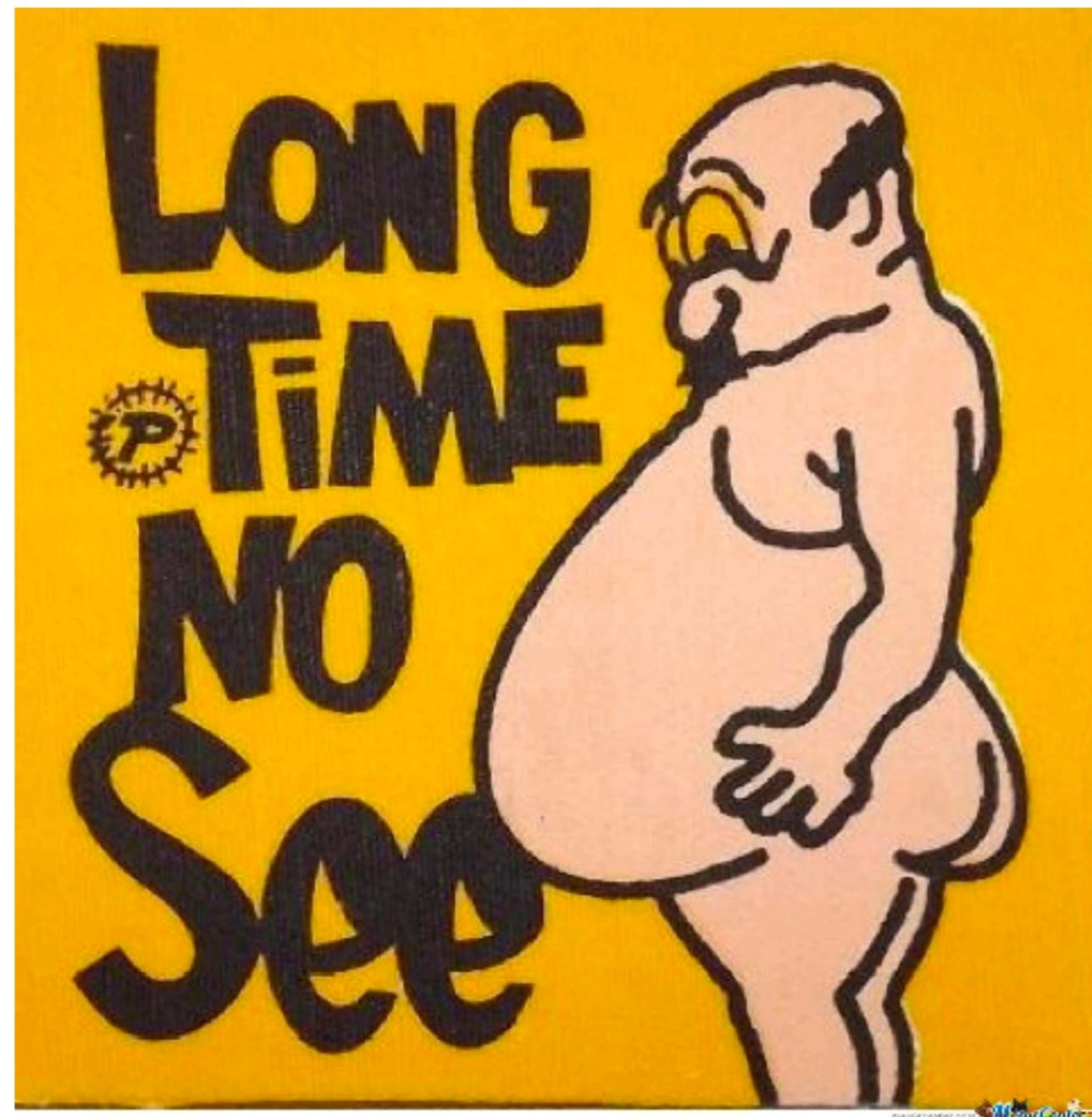


Microservices

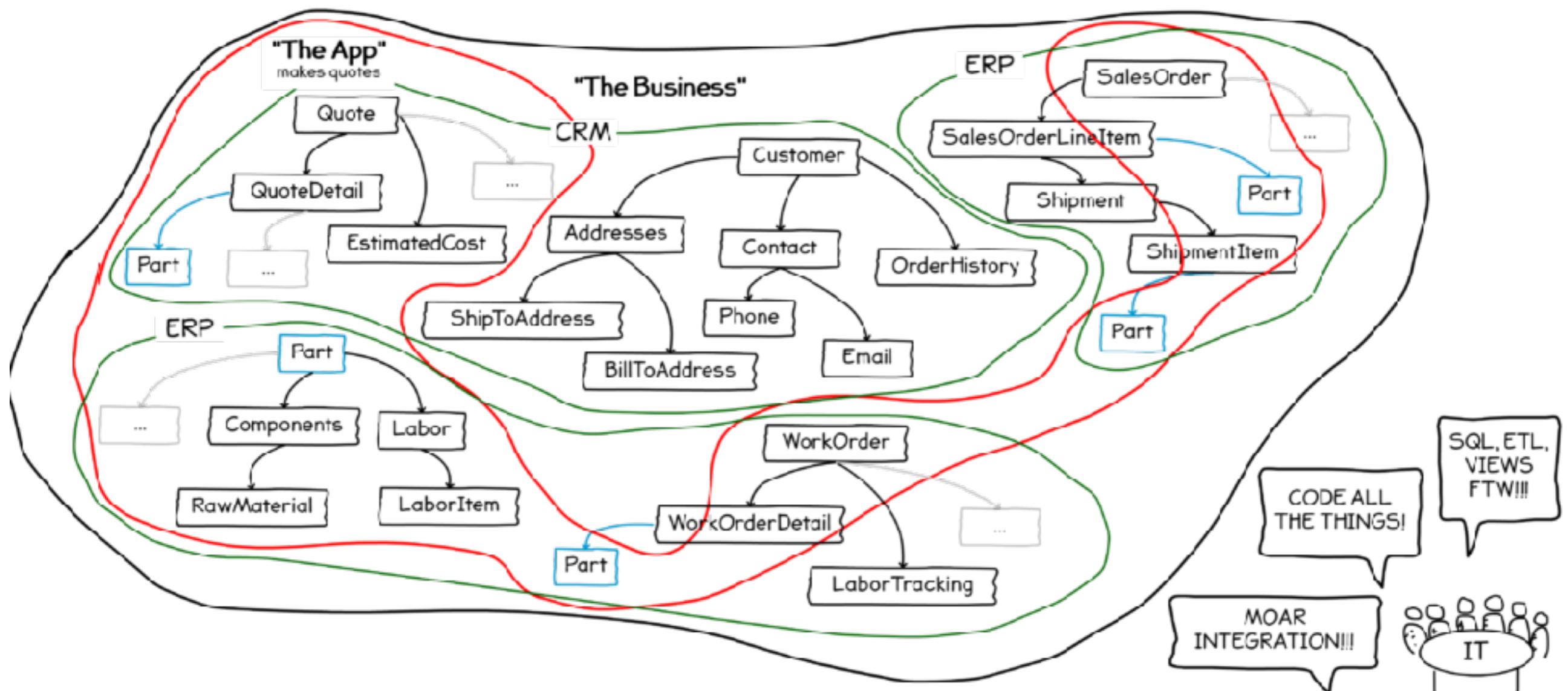
© 2017 - 2018 Siam Chamnankit Company Limited. All rights reserved.

# More features ...





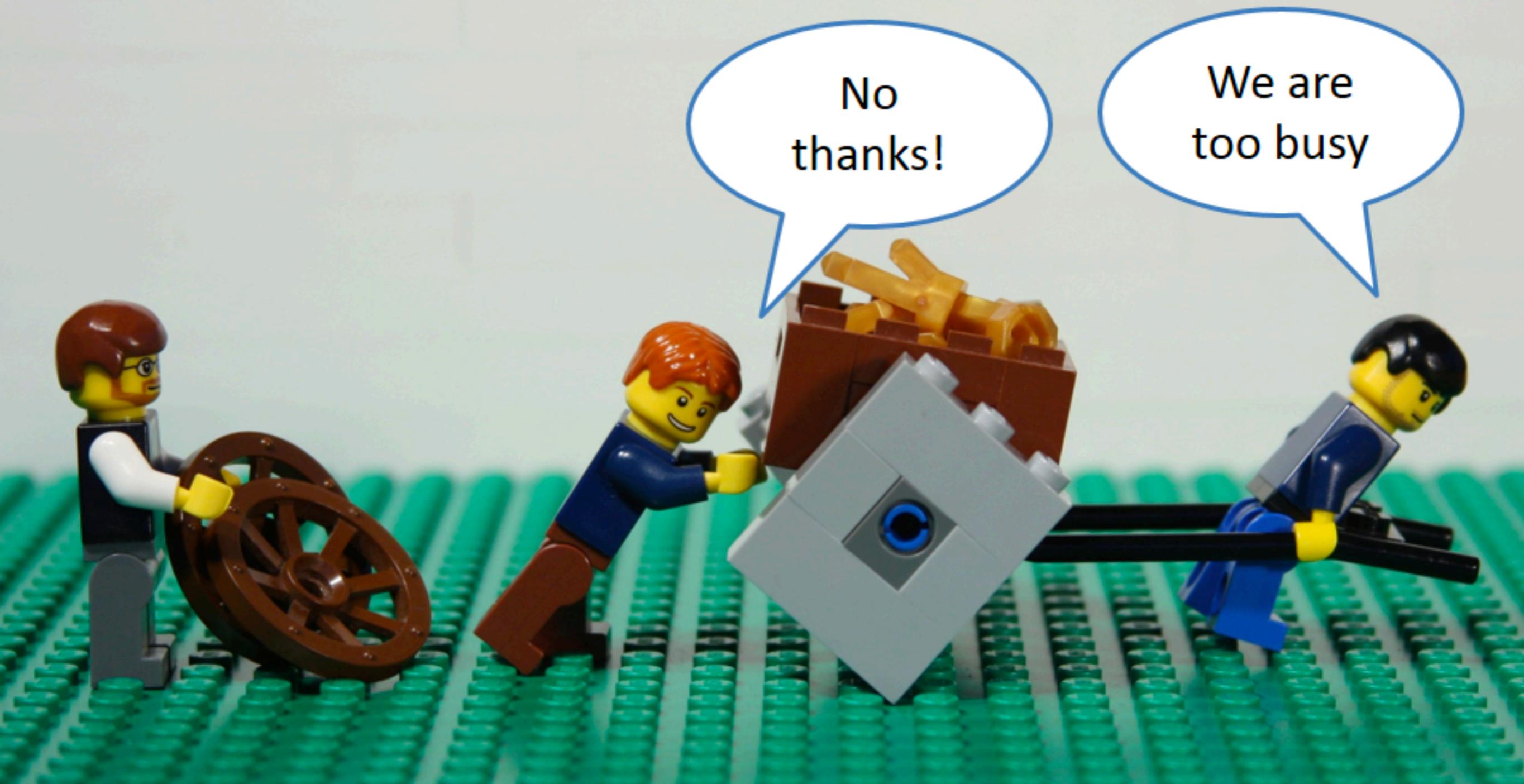
# Monolith Hell



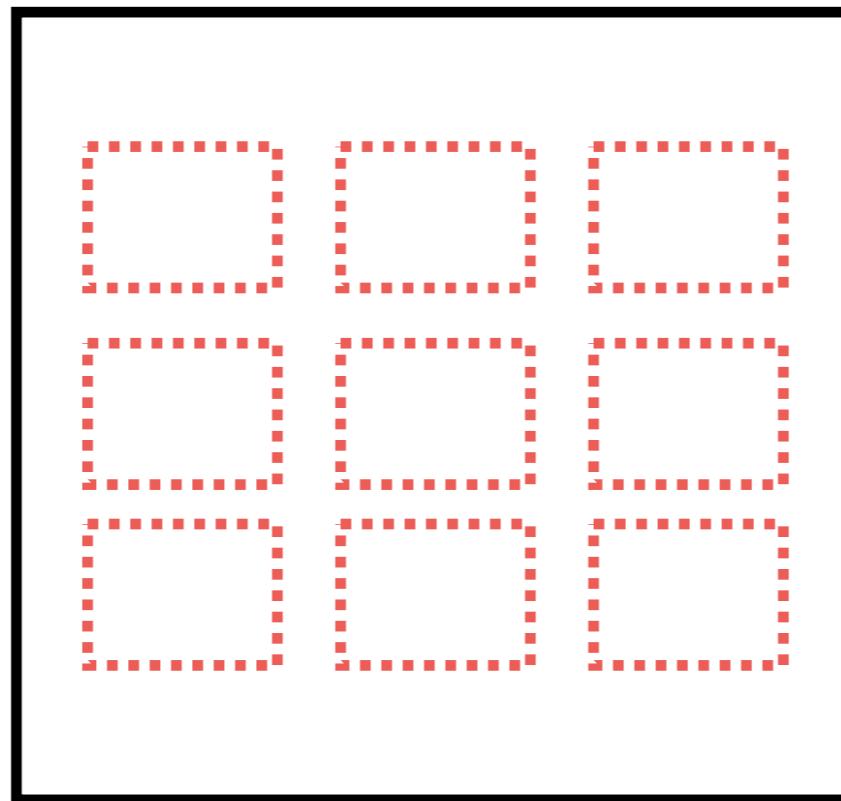
**We need to improve**

**We need to get better**

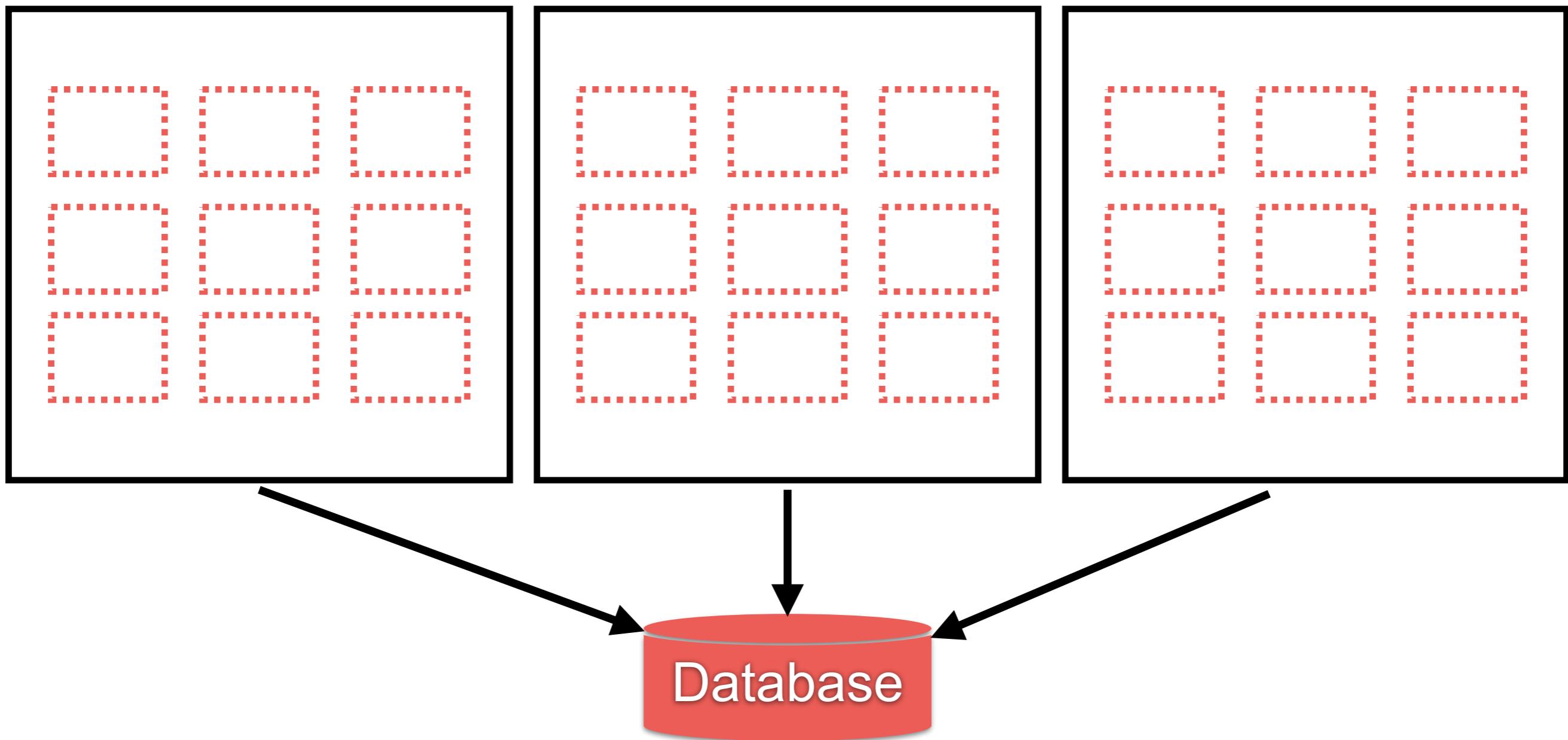




# Modules



# How to scale ?



# **What happens when we need more servers ?**



# **What if we don't use all modules equally ?**



# How can we update individual models/database ?



**Do all modules need to use the  
same Database, Language and  
Runtime ... ?**



**We need to improve**

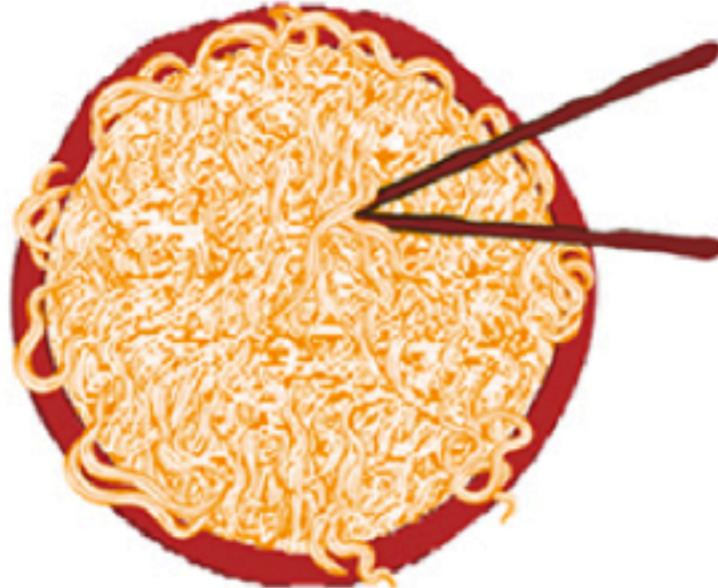
**We need to get better**



# SOA

## 1990s and earlier

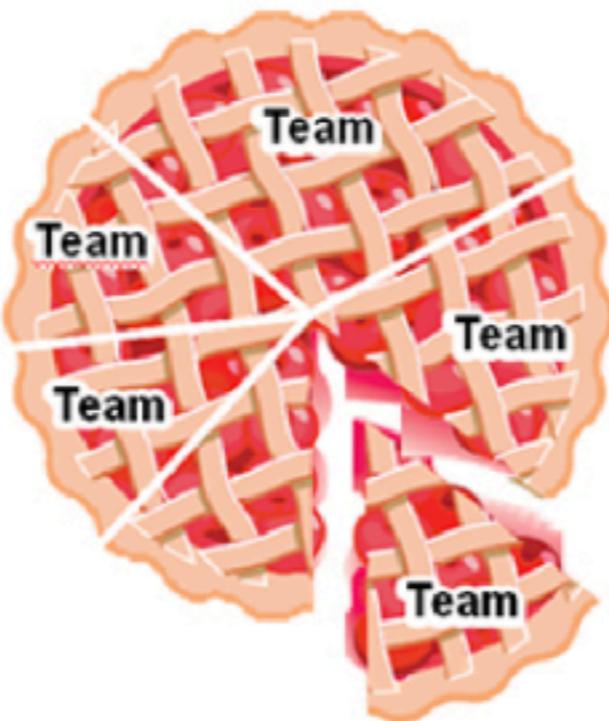
Pre-SOA (monolithic)  
Tight coupling



For a monolith to change, all must agree on each change. Each change has unanticipated effects requiring careful testing beforehand.

## 2000s

Traditional SOA  
Looser coupling



Elements in SOA are developed more autonomously but must be coordinated with others to fit into the overall design.

## 2010s

Microservices  
Decoupled



Developers can create and activate new microservices without prior coordination with others. Their adherence to MSA principles makes continuous delivery of new or modified services possible.



# Service-Oriented Architecture



# What is service ?

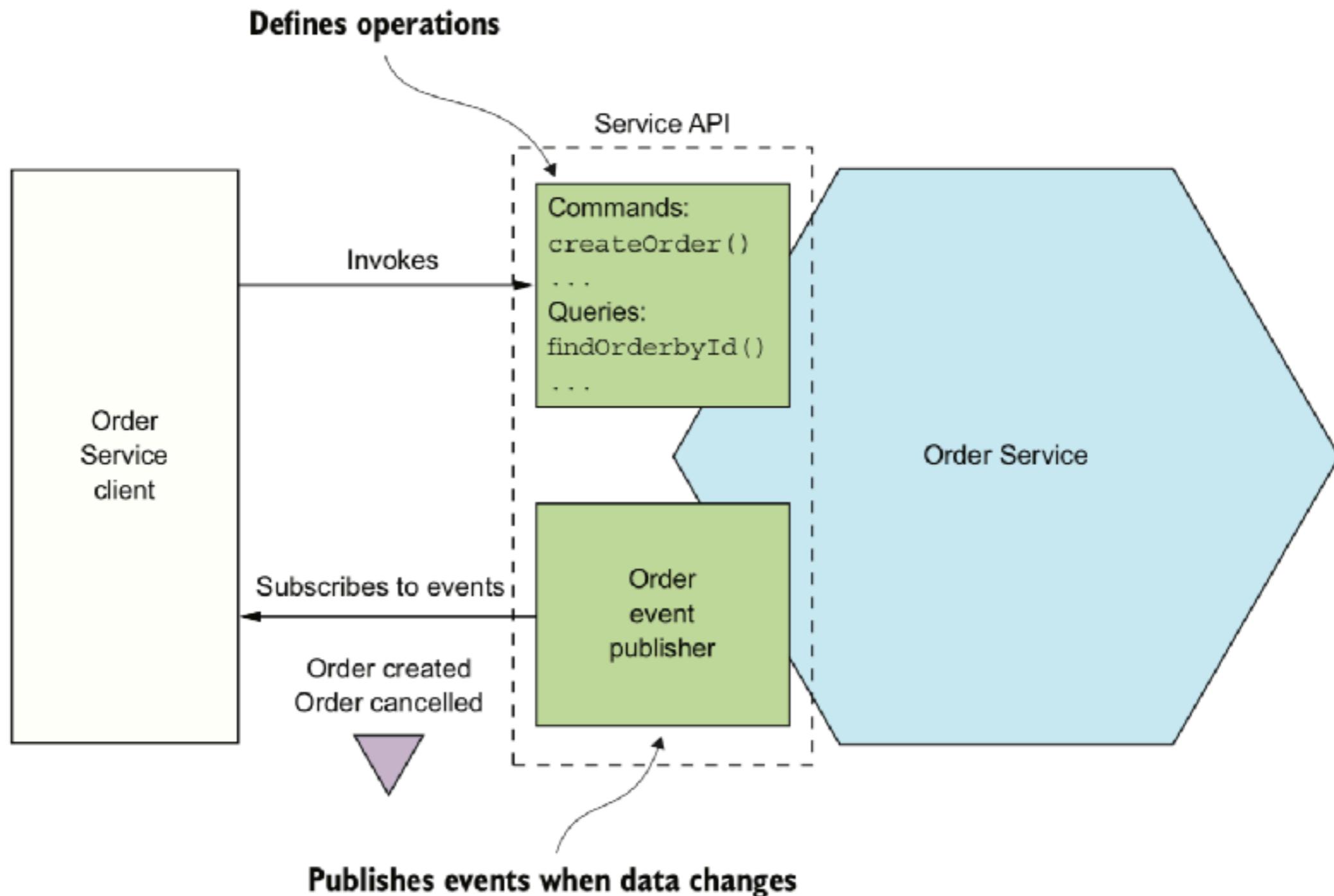
Standalone and loosely couple

Independently deployable software component

Implement some useful functionality



# Example of service



# Service-Oriented Architecture

## SOA Manifesto

Service orientation is a paradigm that frames what you do.  
Service-oriented architecture (SOA) is a type of architecture  
that results from applying service orientation.

We have been applying service orientation to help organizations  
consistently deliver sustainable business value, with increased agility  
and cost effectiveness, in line with changing business needs.

<http://www.soa-manifesto.org/>



Microservices

© 2017 - 2018 Siam Chamnankit Company Limited. All rights reserved.

# Service-Oriented Architecture

Through our work we have come to prioritize:

**Business value** over technical strategy

**Strategic goals** over project-specific benefits

**Intrinsic interoperability** over custom integration

**Shared services** over specific-purpose implementations

**Flexibility** over optimization

**Evolutionary refinement** over pursuit of initial perfection

<http://www.soa-manifesto.org/>



# Service-Oriented Architecture

Through our work we have come to prioritize:

**Business value** over technical strategy

**Strategic goals** over project-specific benefits

**Intrinsic interoperability** over custom integration

**Shared services** over specific-purpose implementations

**Flexibility** over optimization

**Evolutionary refinement** over pursuit of initial perfection

<http://www.soa-manifesto.org/>



Microservices

© 2017 - 2018 Siam Chamnankit Company Limited. All rights reserved.

# Service-Oriented Architecture

Through our work we have come to prioritize:

**Business value** over technical strategy

**Strategic goals** over project-specific benefits

**Intrinsic interoperability** over custom integration

**Shared services** over specific-purpose implementations

**Flexibility** over optimization

**Evolutionary refinement** over pursuit of initial perfection

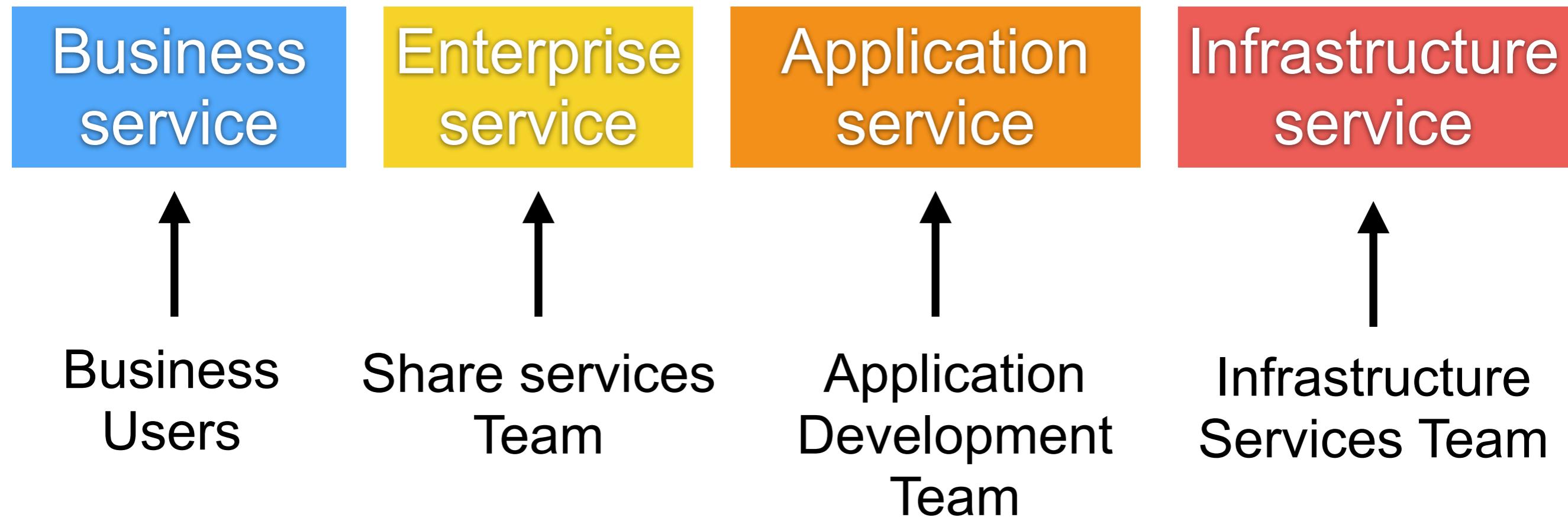
<http://www.soa-manifesto.org/>



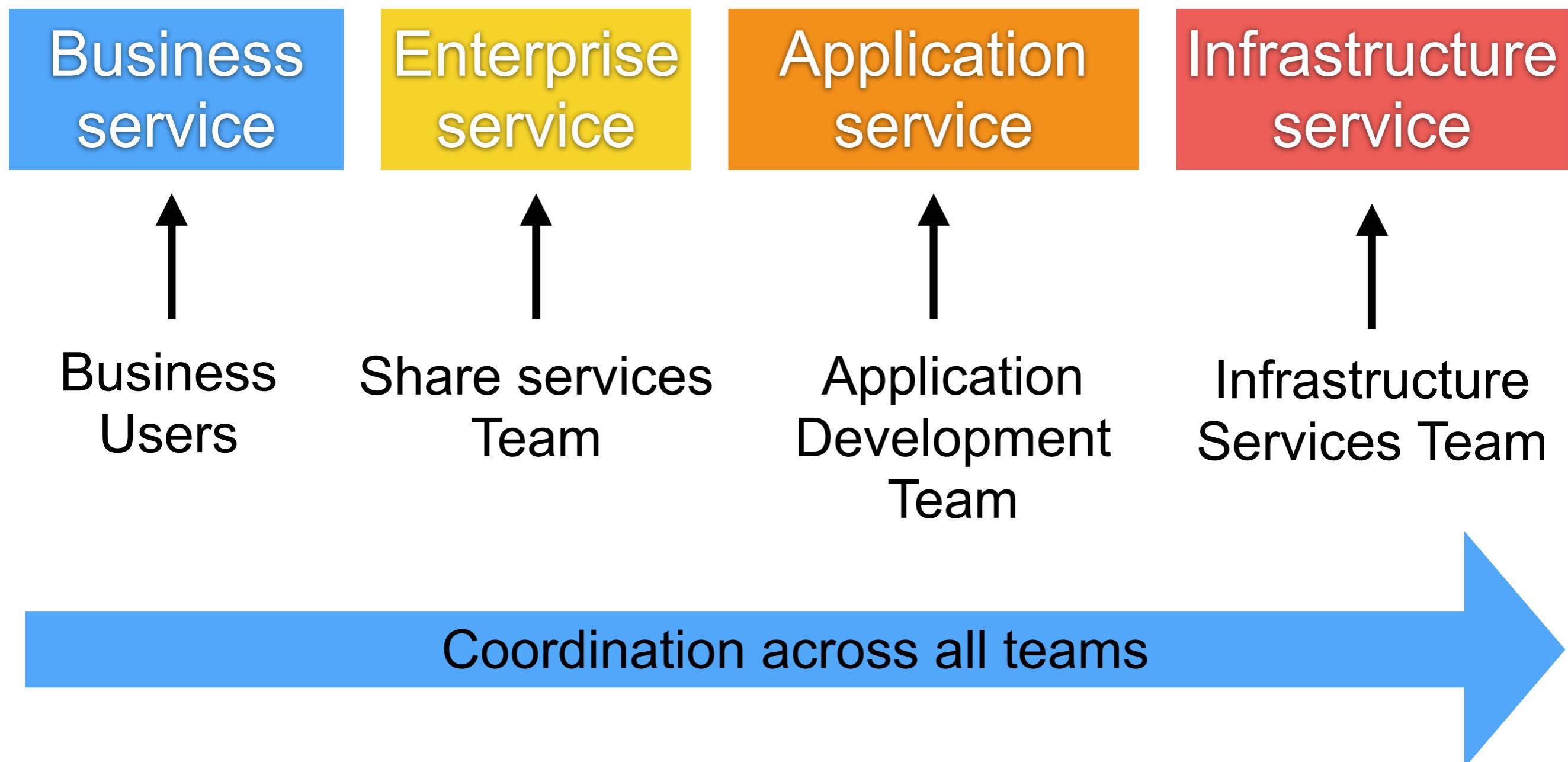
Microservices

© 2017 - 2018 Siam Chamnankit Company Limited. All rights reserved.

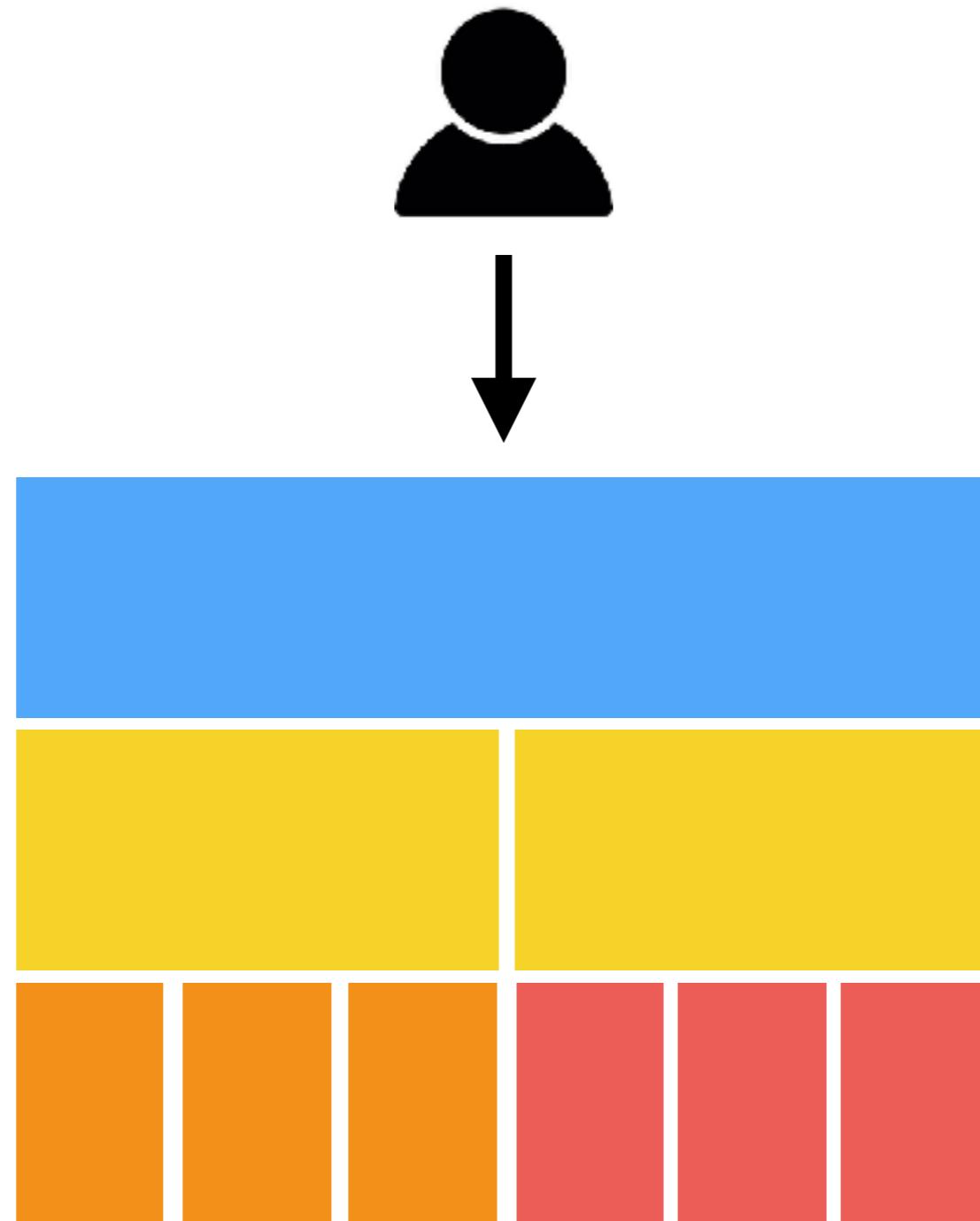
# Service-Oriented Architecture



# Service-Oriented Architecture



# Service-Oriented Architecture



**We need to improve**

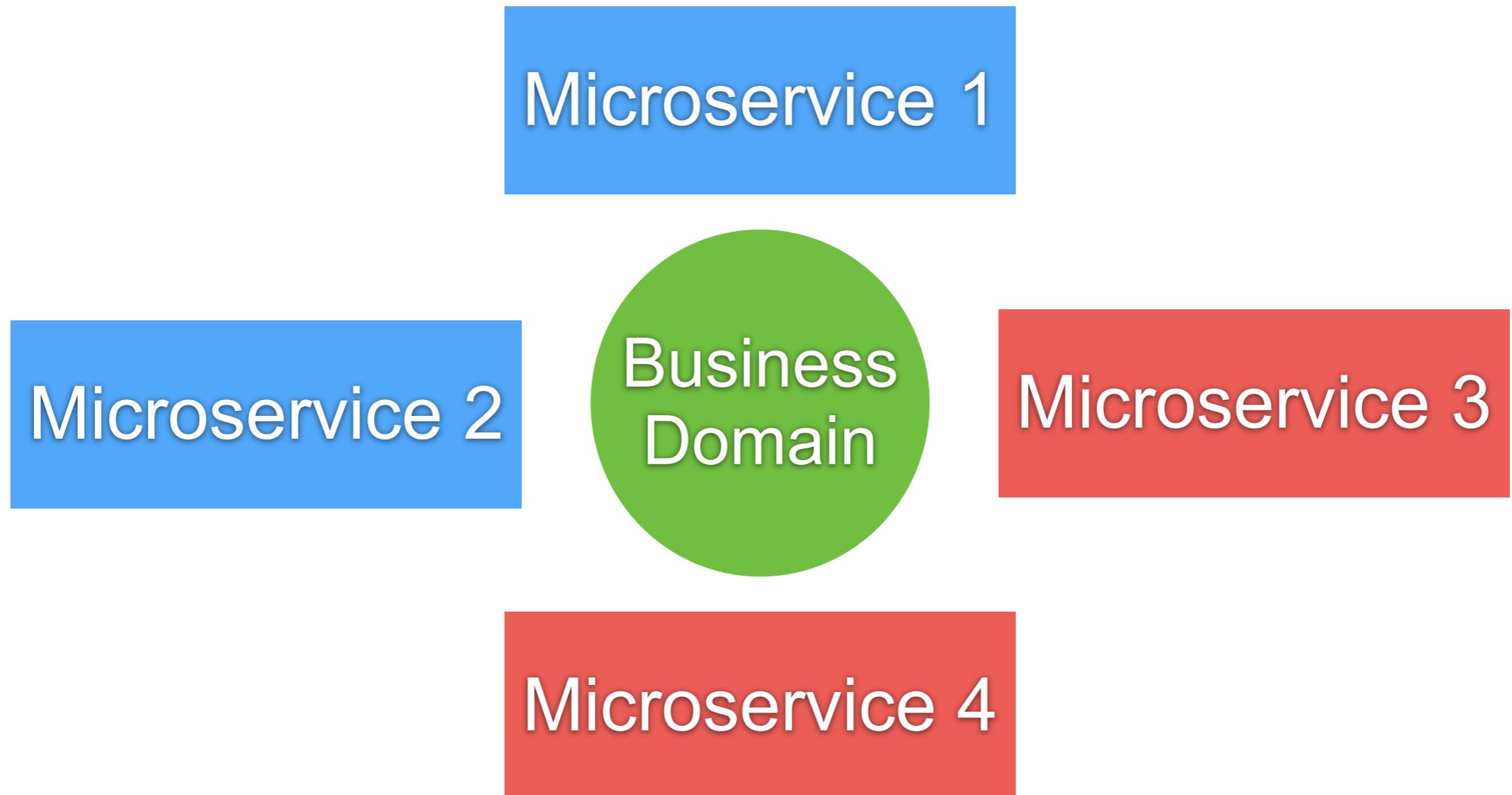
**We need to get better**



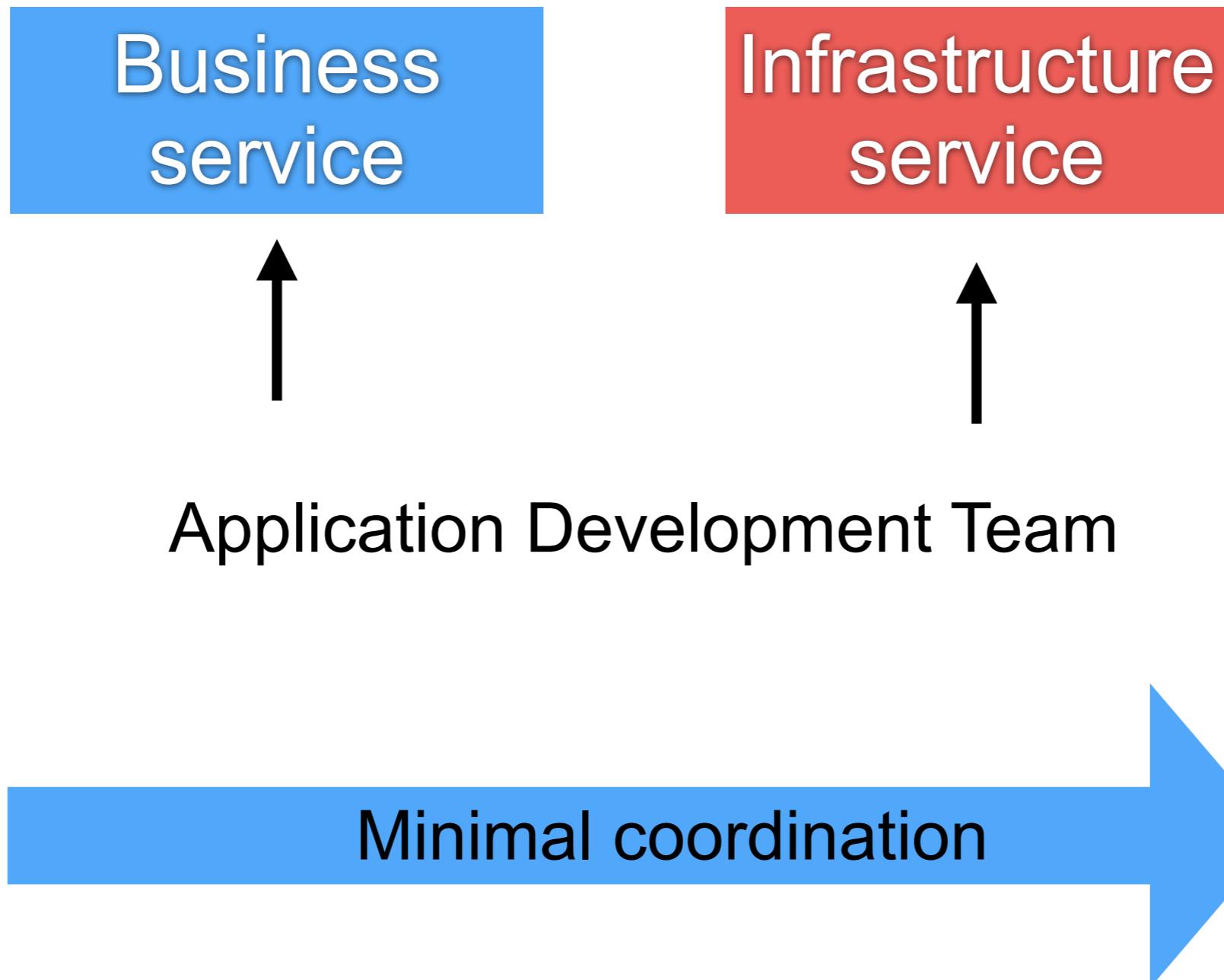
# Microservice



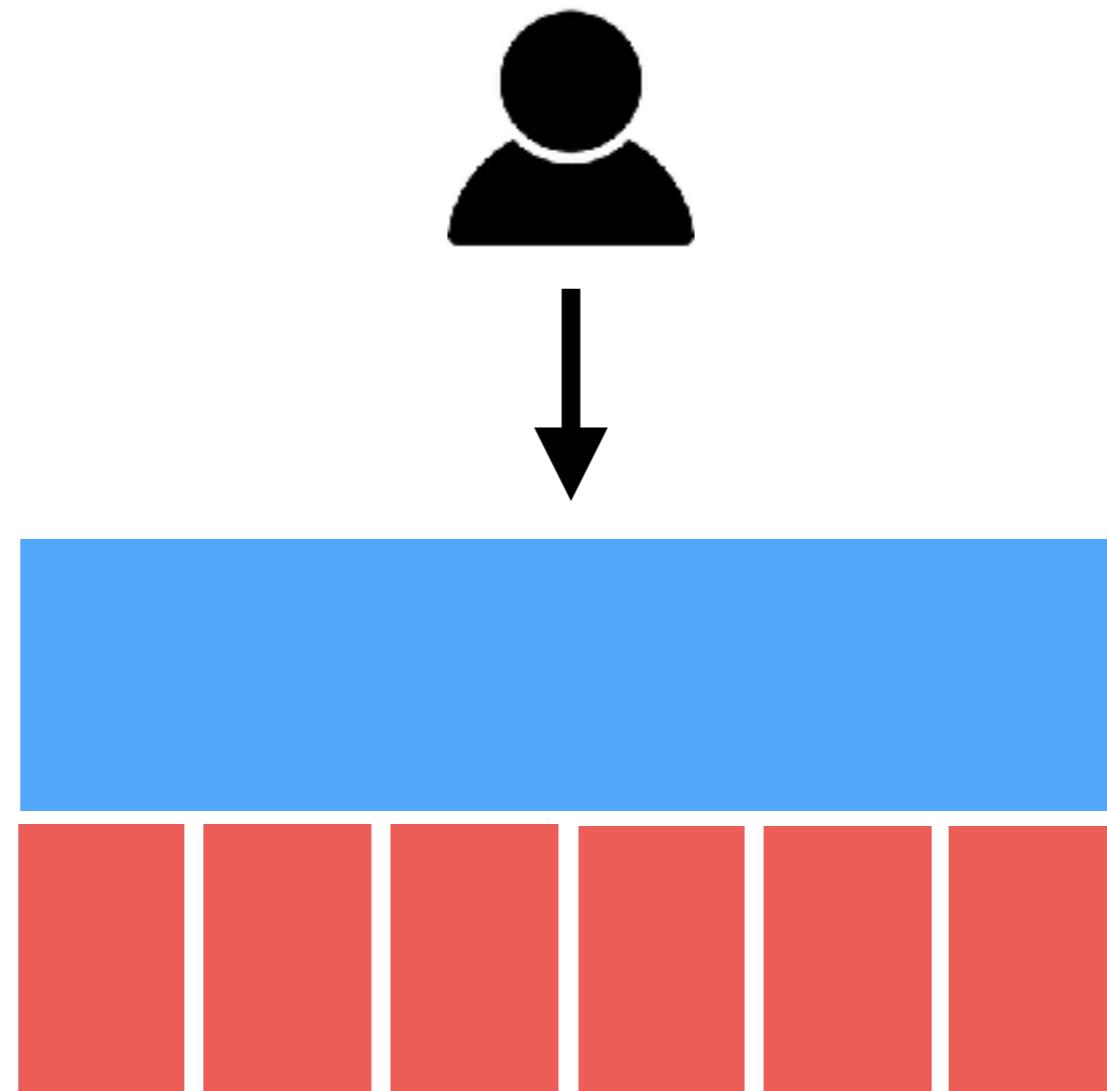
# Microservice



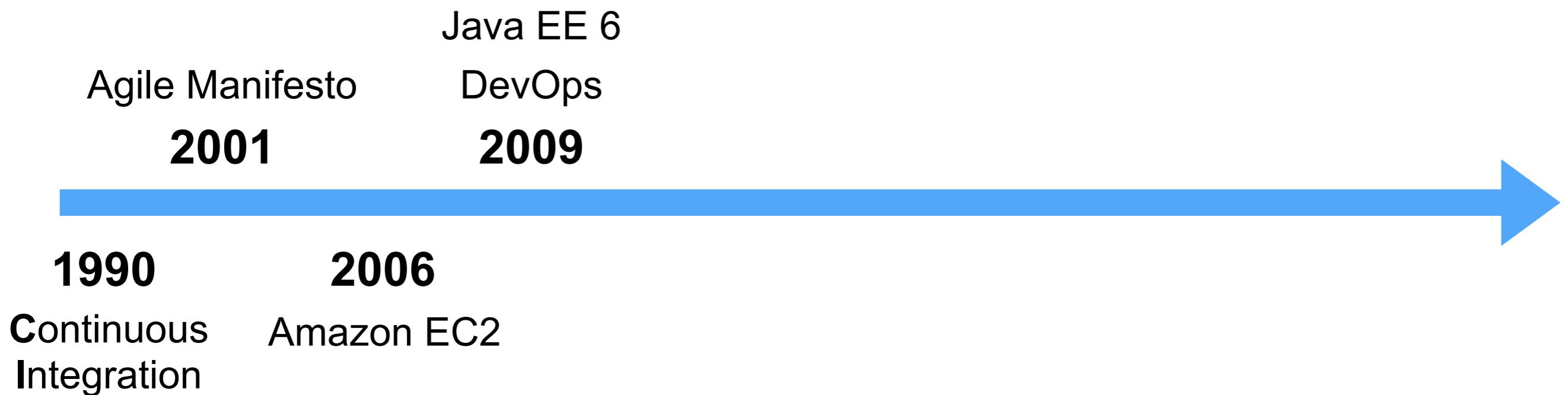
# Microservice



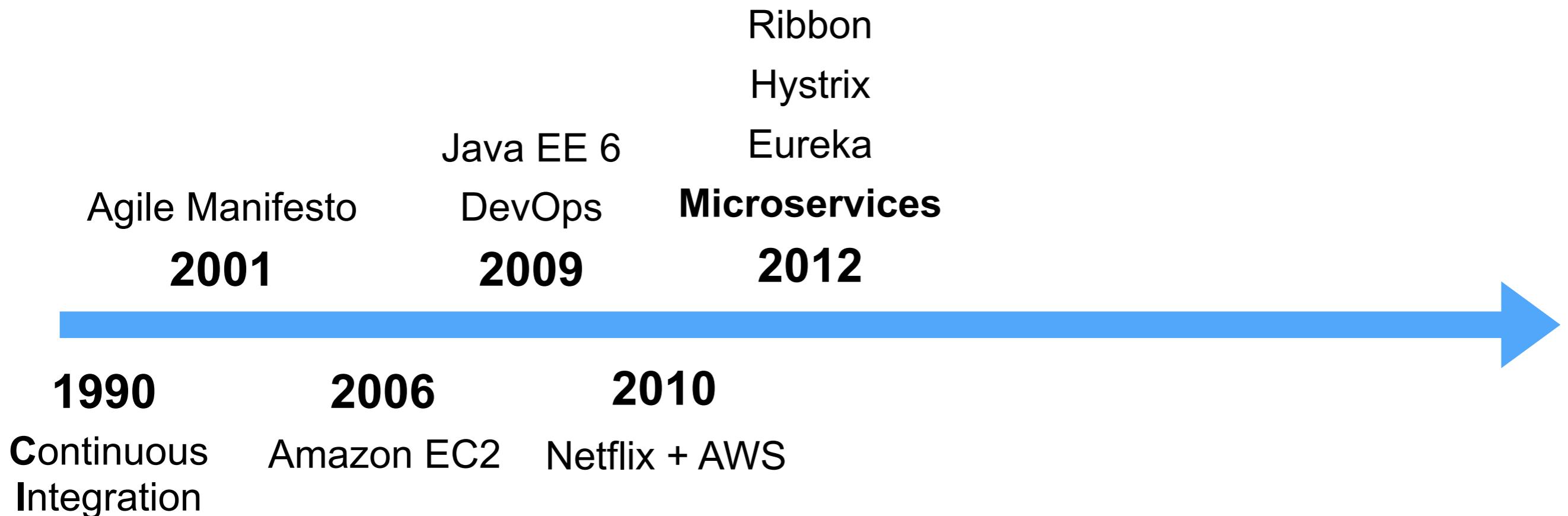
# Microservice



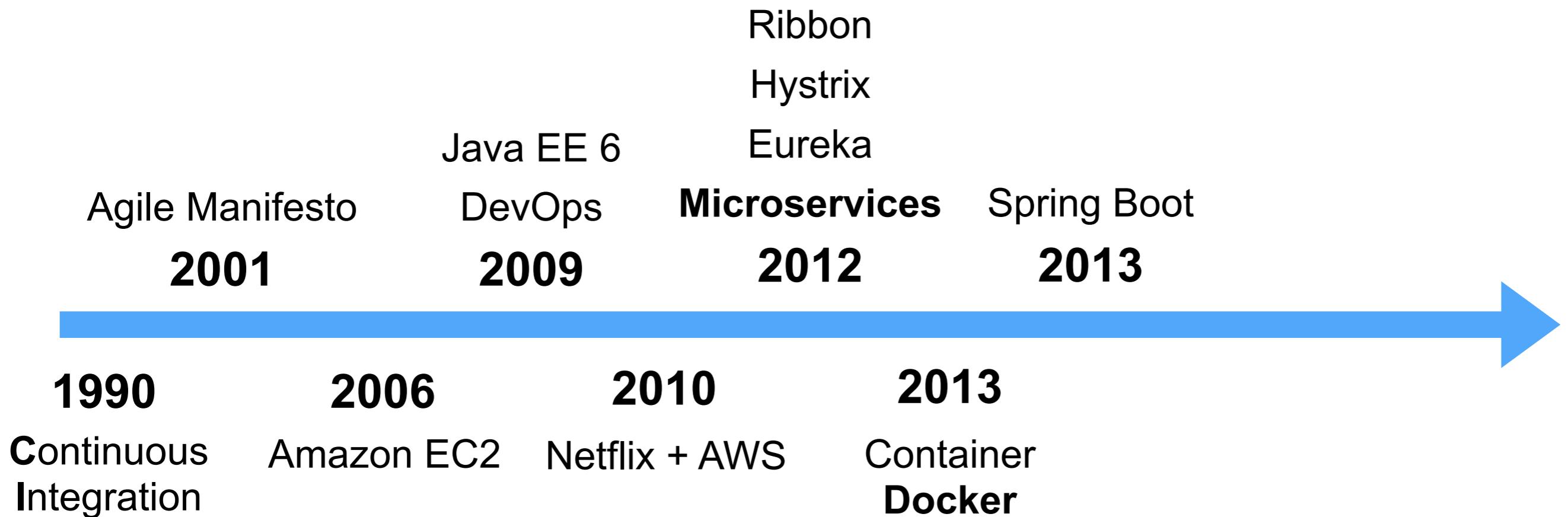
# Microservice history



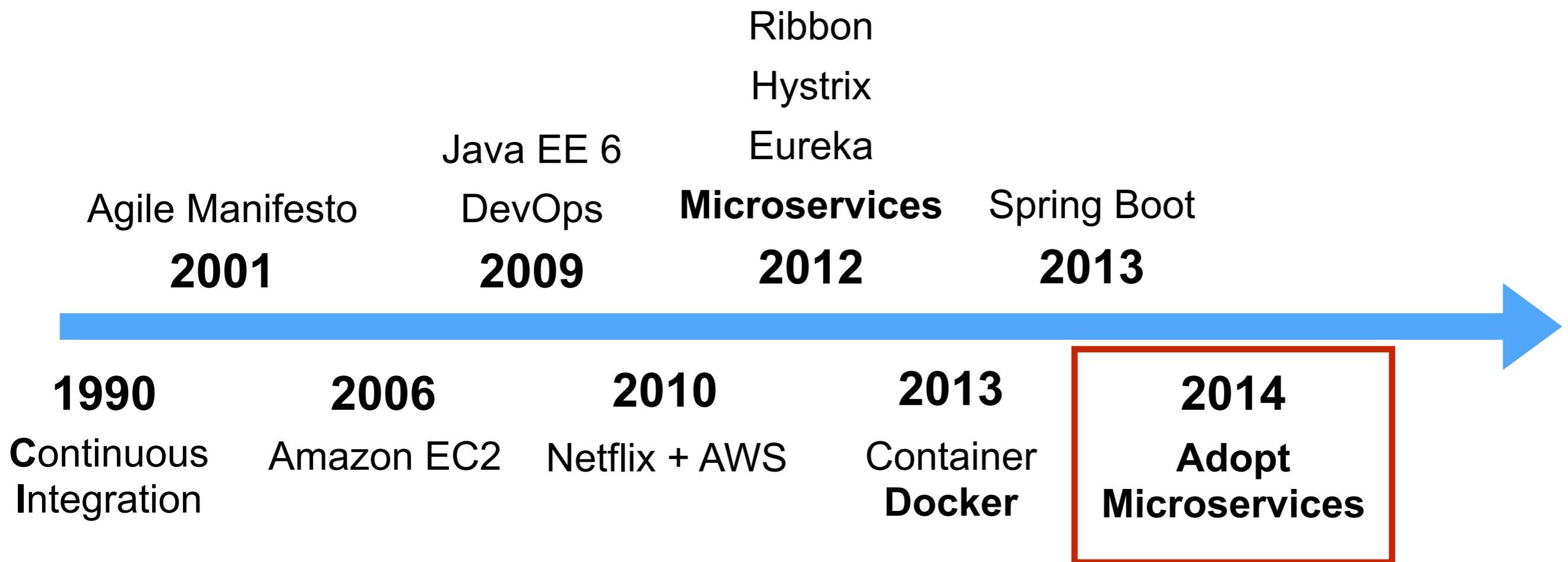
# Microservice history



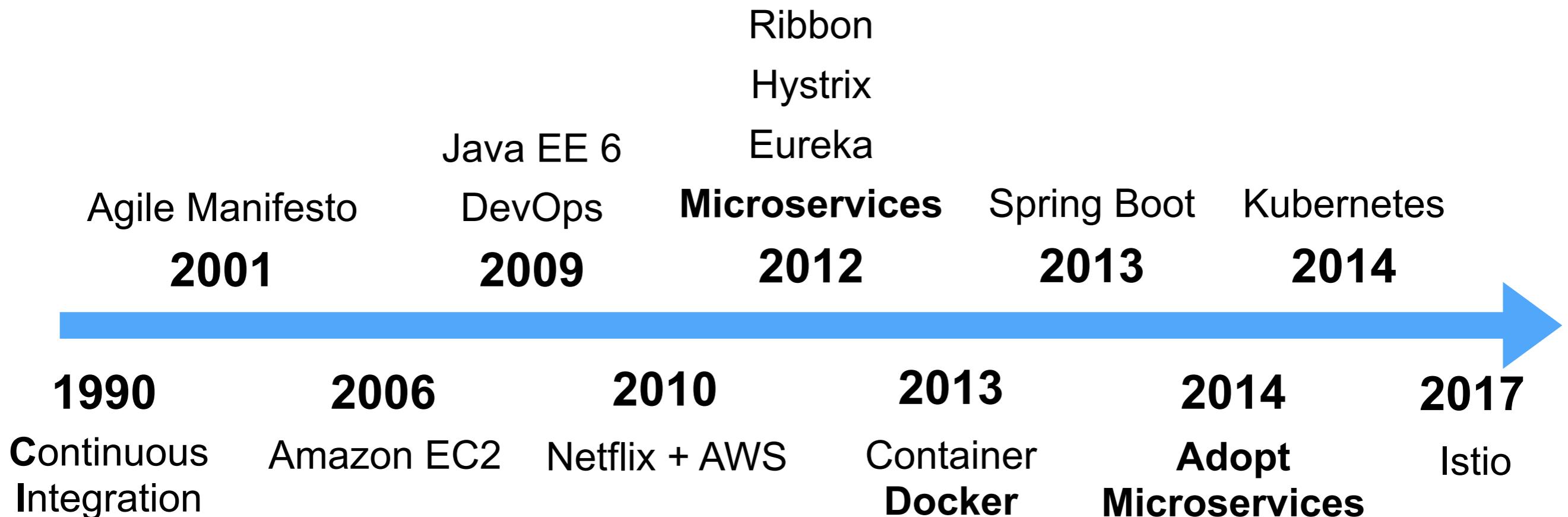
# Microservice history



# Microservice history



# Microservice history



# **Microservice**

**Small, Do one thing (Single Responsibility)**

**Modular**

**Easy to understand**

**Easy to develop**

**Easy to deploy**

**Easy to maintain**

**Scale independently**



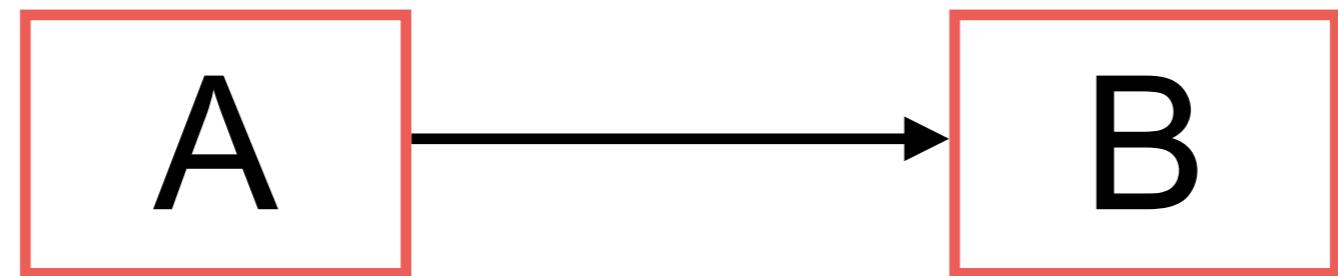
# Goals

Increase the **velocity** of application release  
By decomposing app to **small services**  
**Autonomous services**  
**Deploy independently**



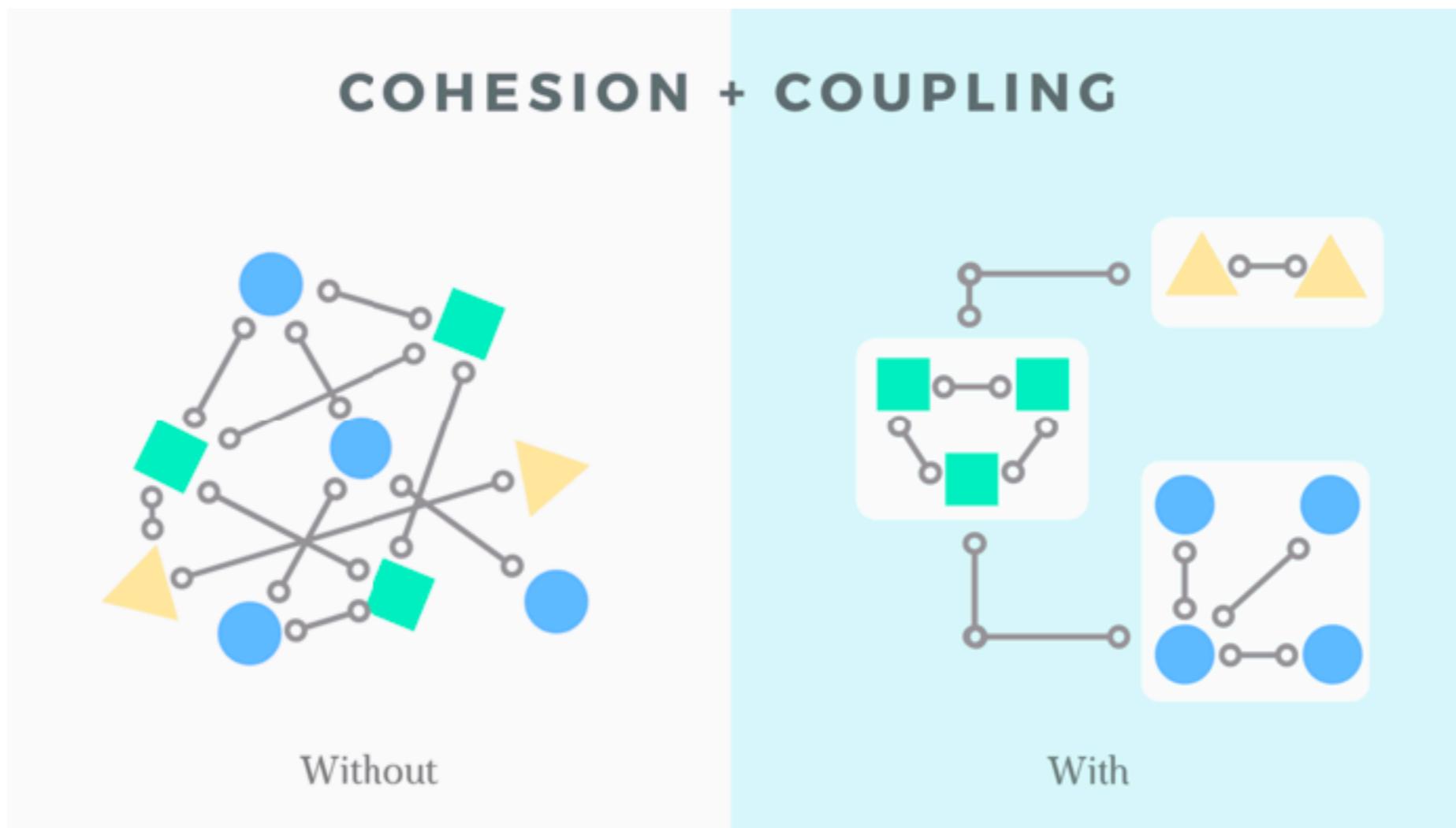
# Independent

## Replicable Updatable Scalable Deployable

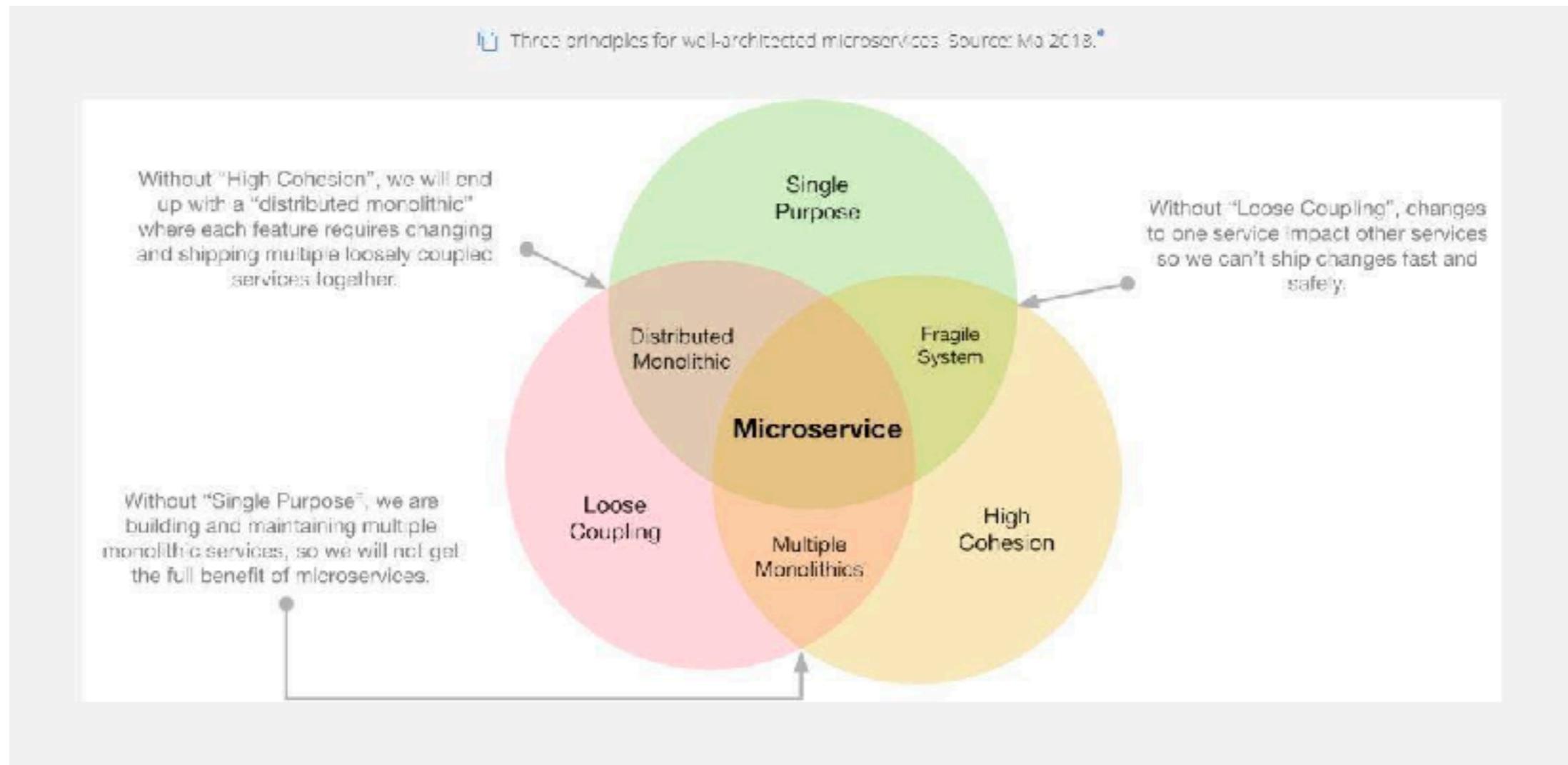


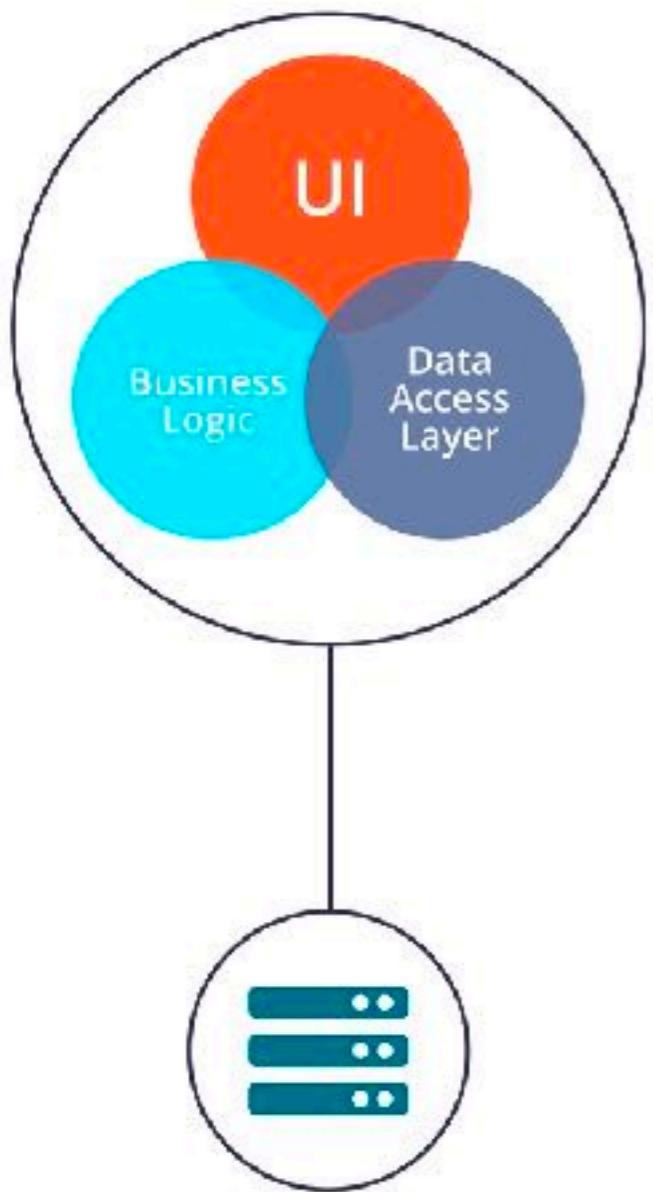
# Better Architecture

Loose coupling  
High cohesion

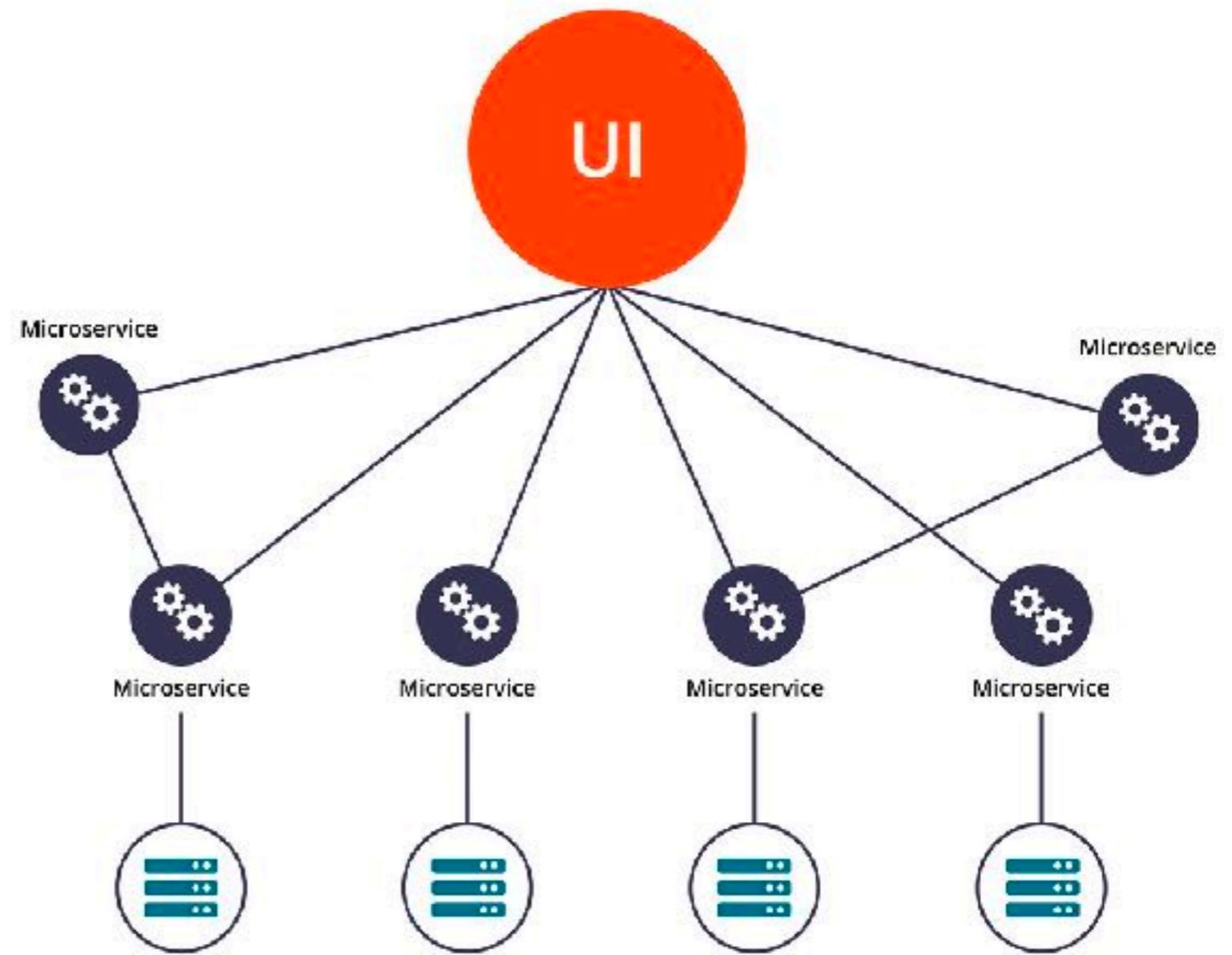


# Better Architecture





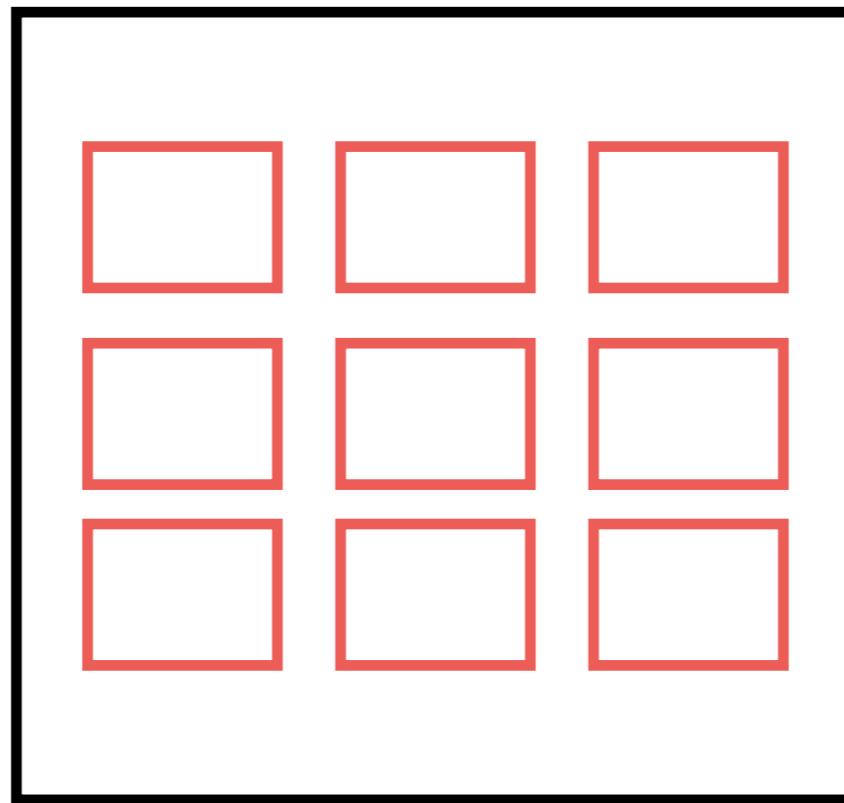
Monolithic Architecture



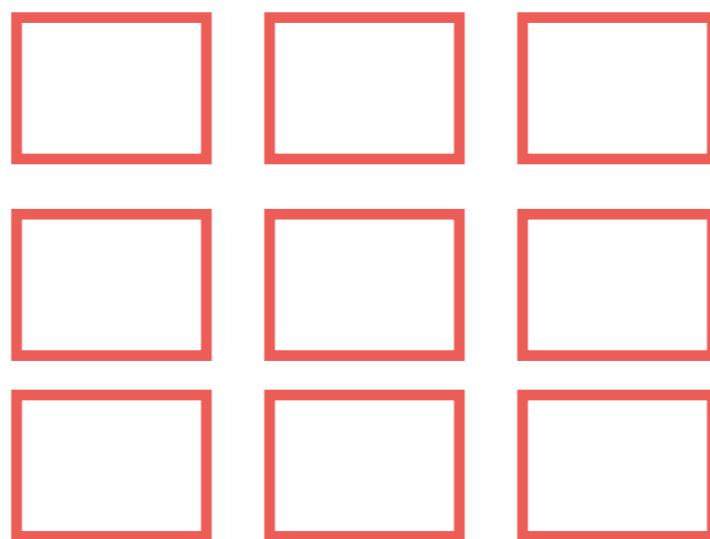
Microservice Architecture



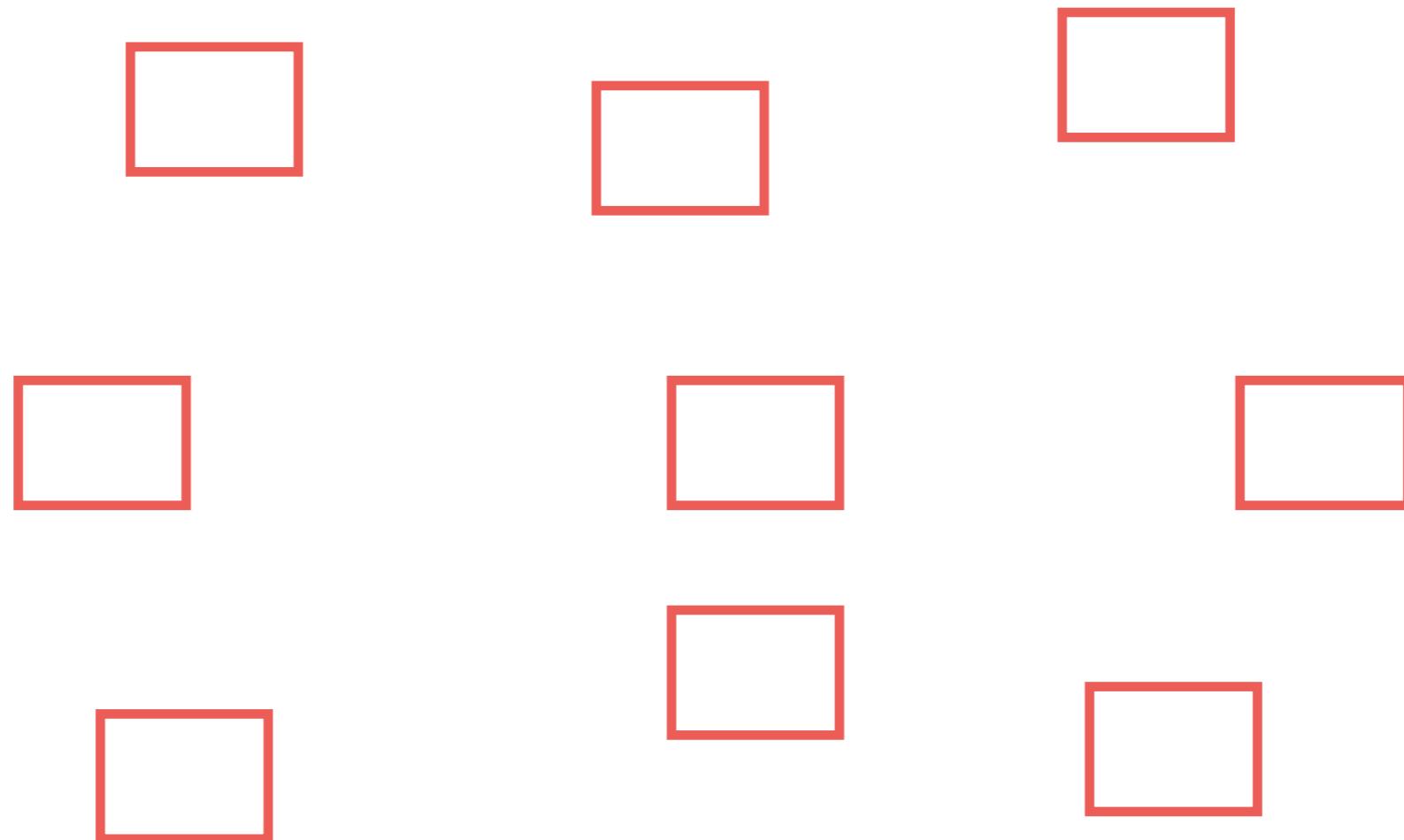
# Microservice



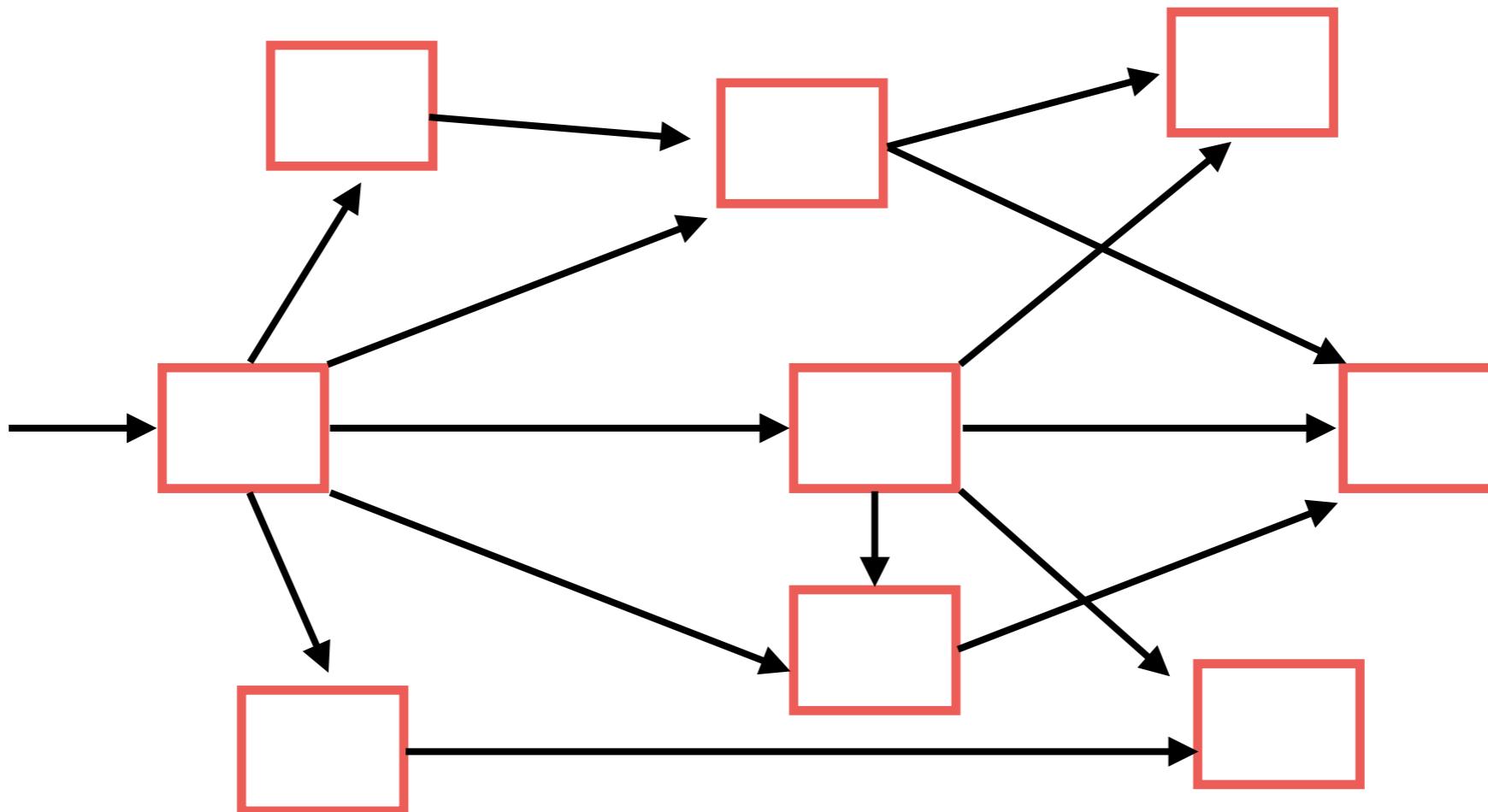
# Microservice



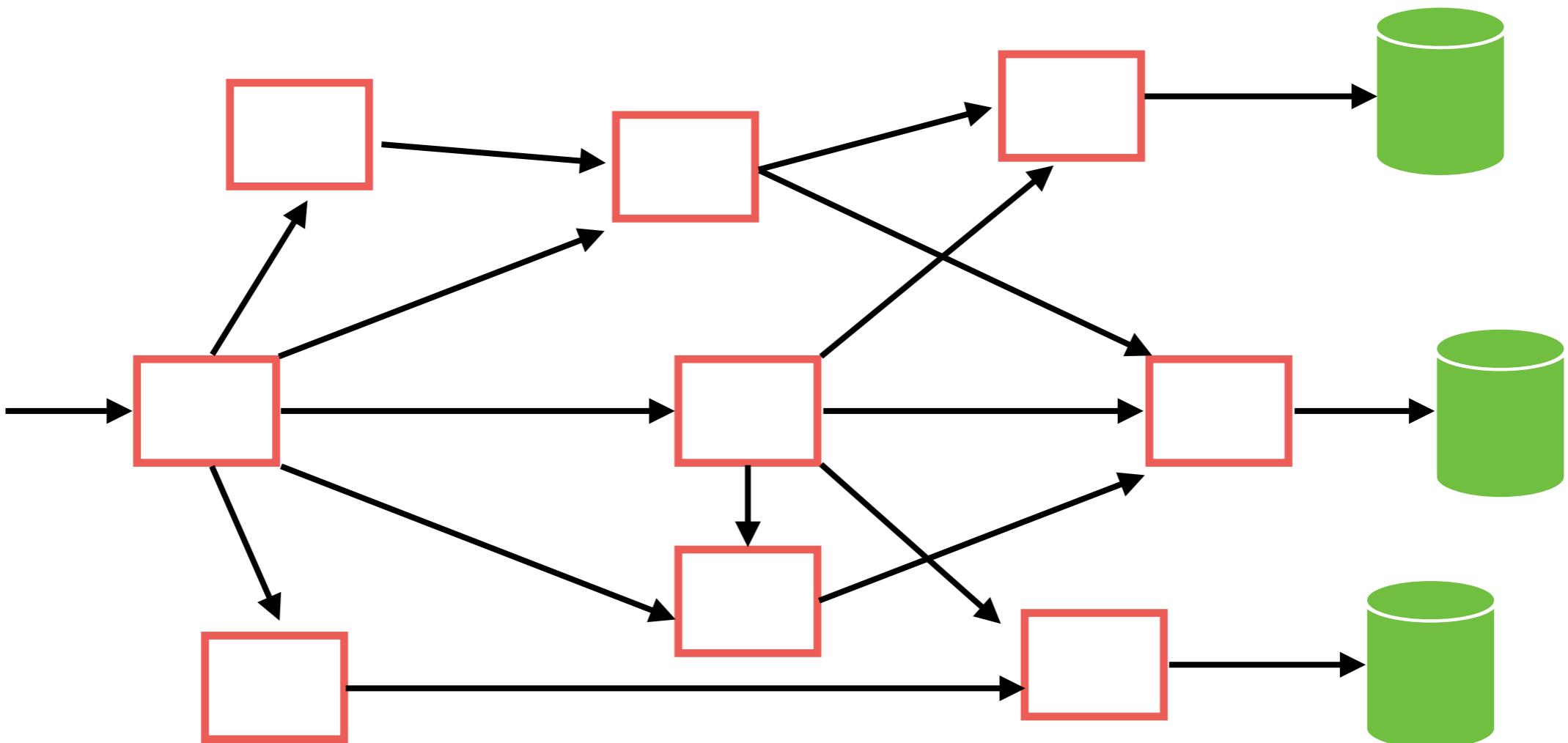
# Microservice



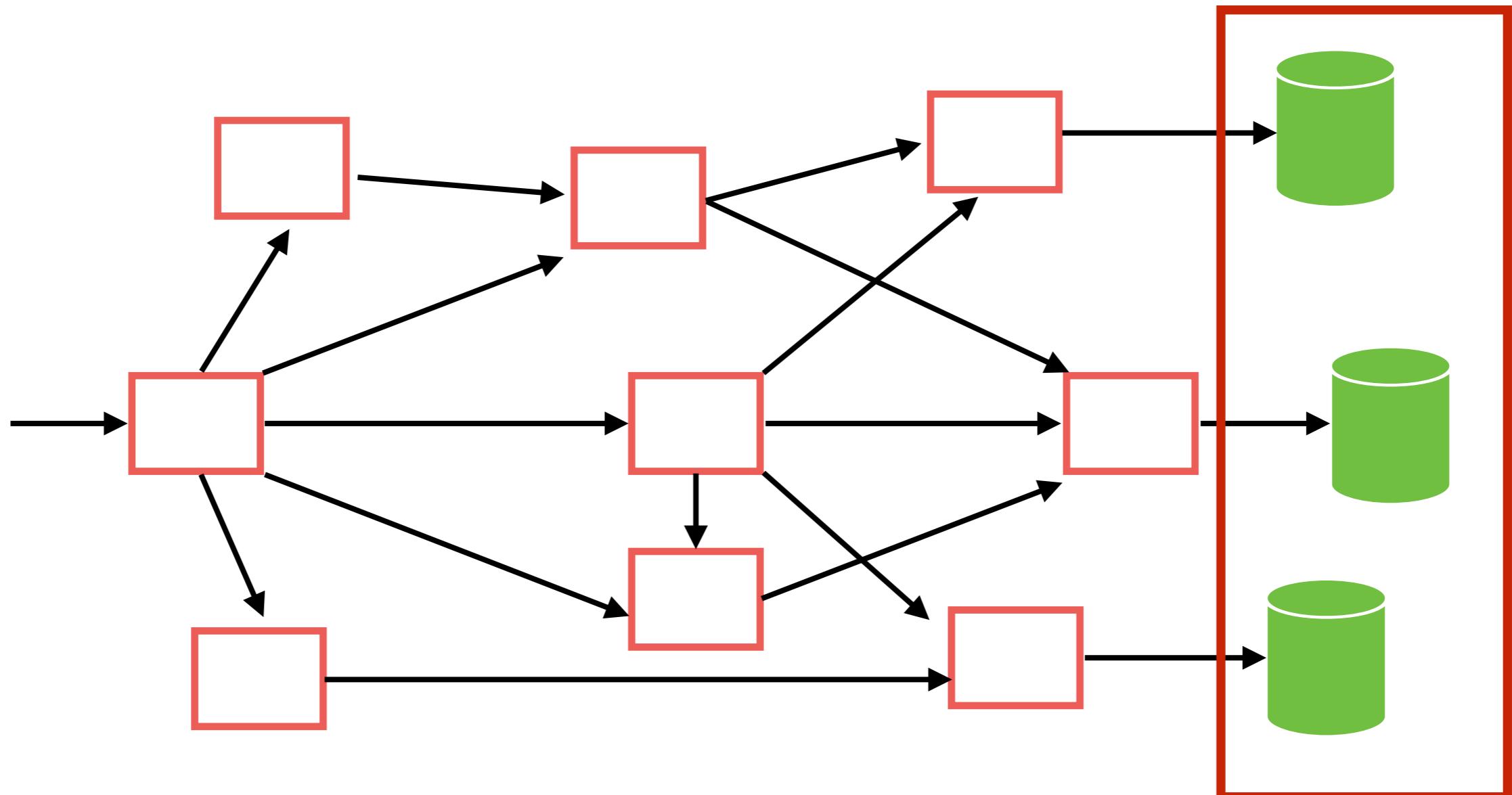
# Microservice == Distributed



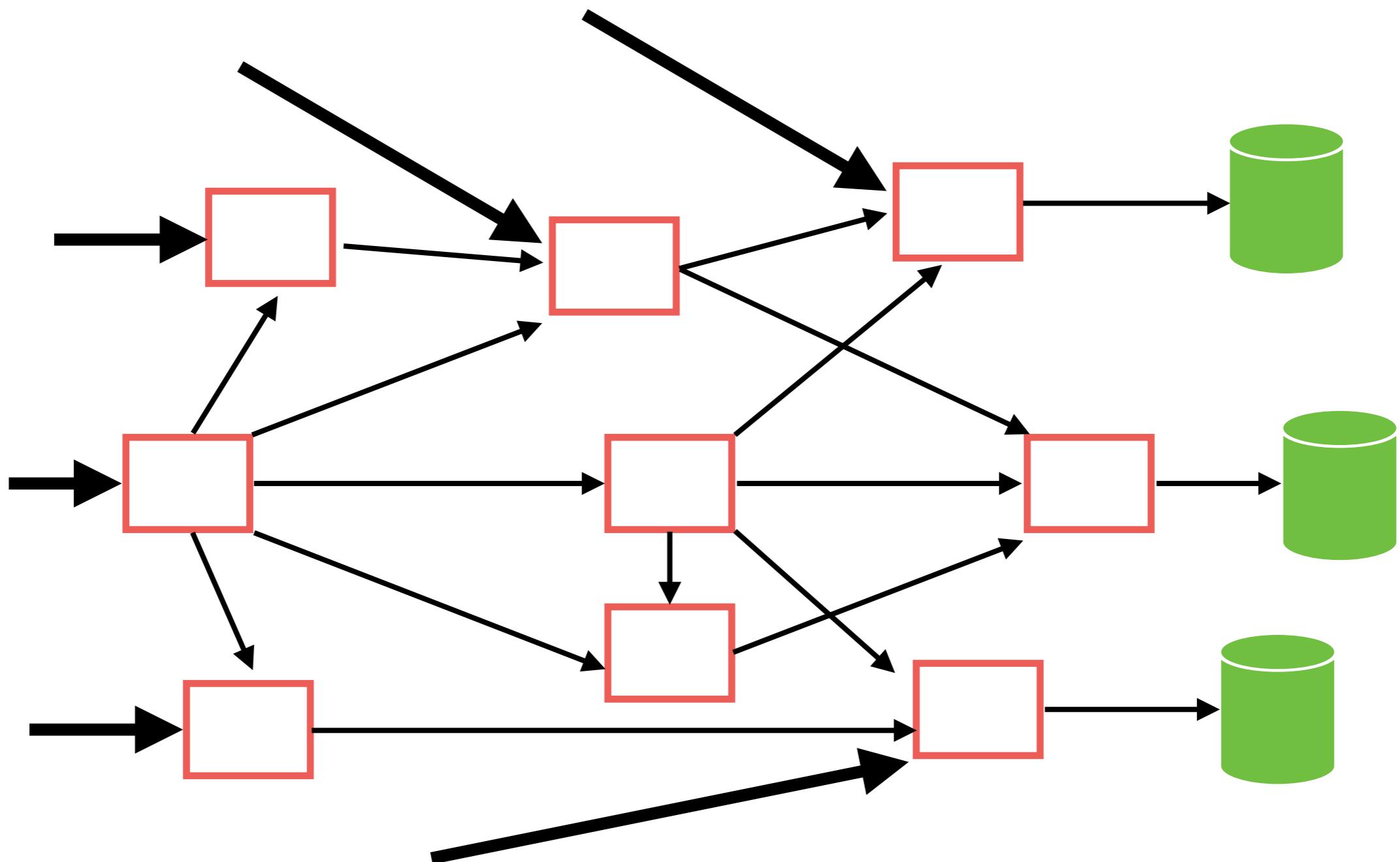
# Own data



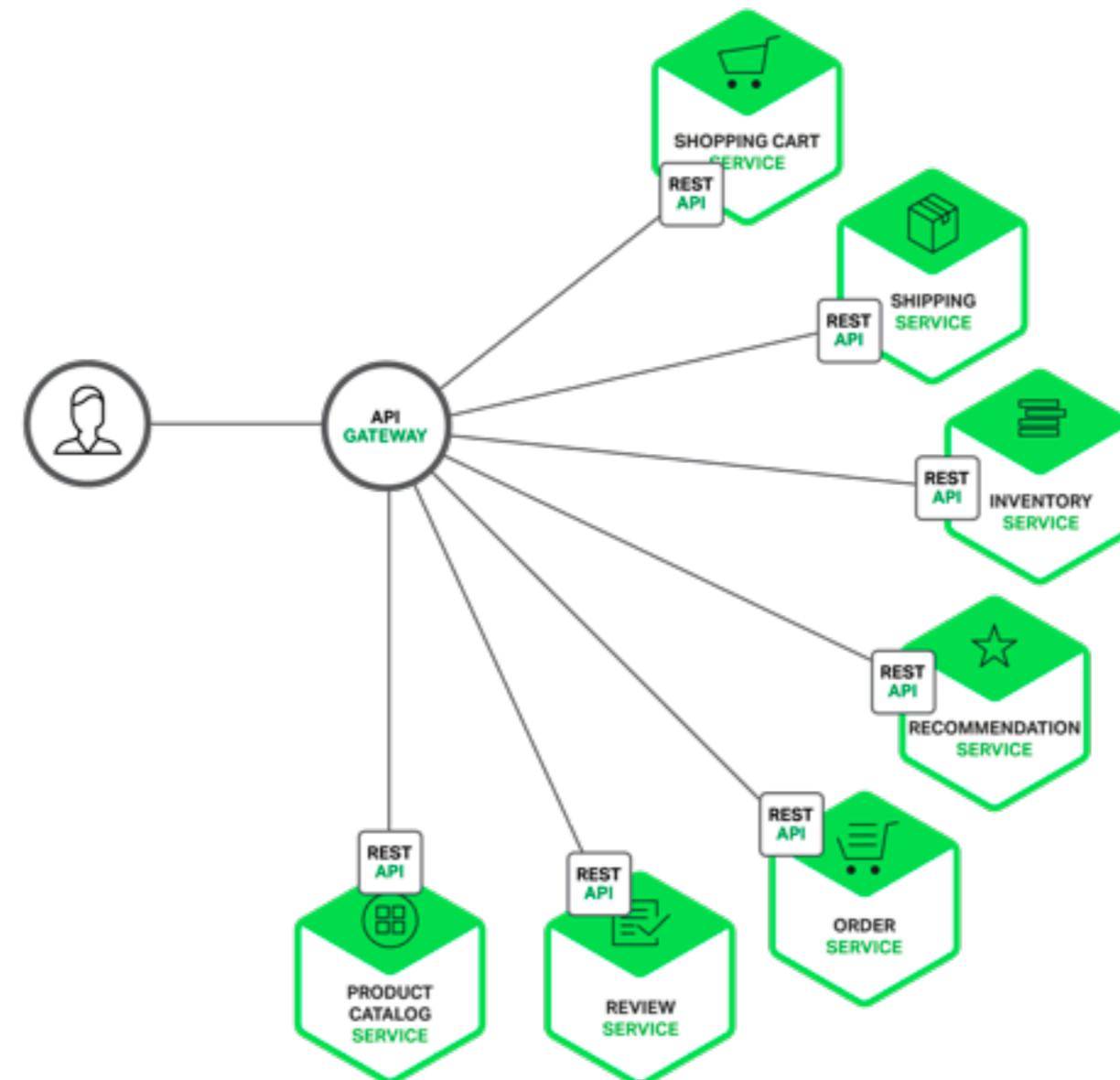
# Data consistency !!



# Multiple entry points



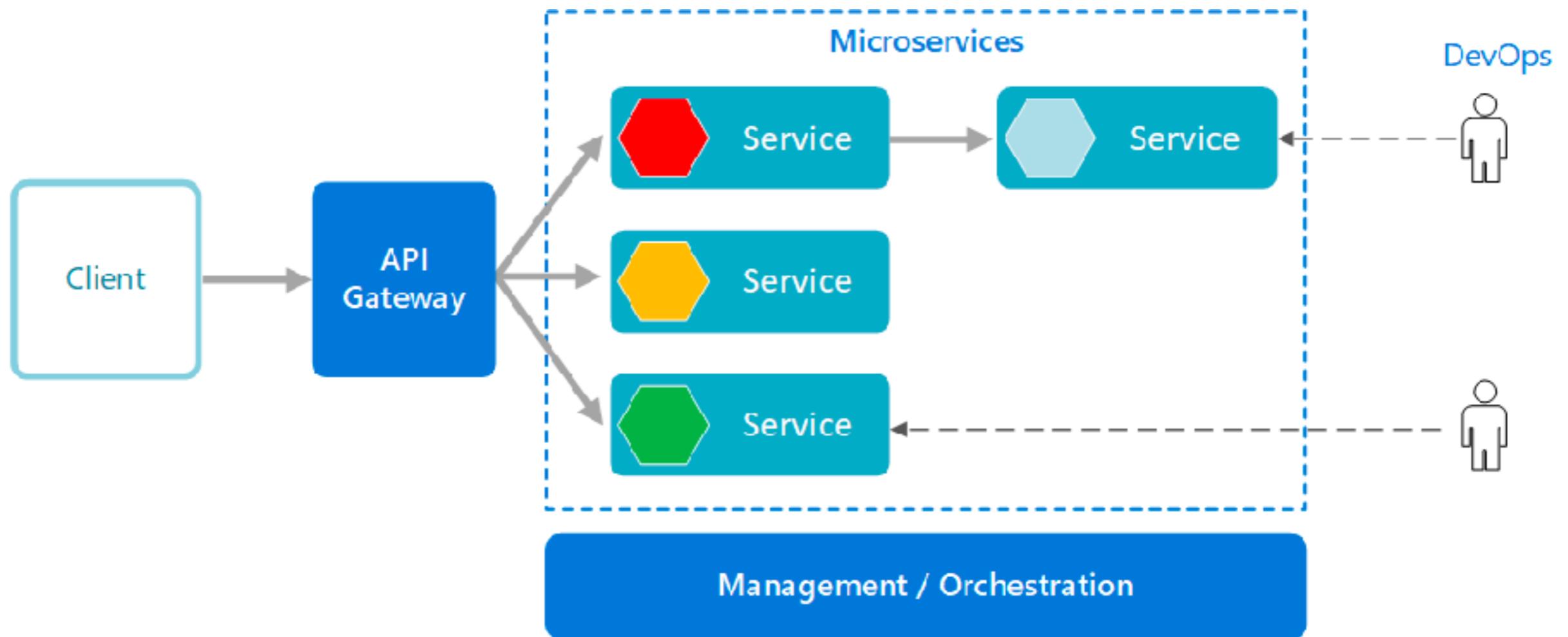
# Working with API gateway



<https://www.nginx.com>



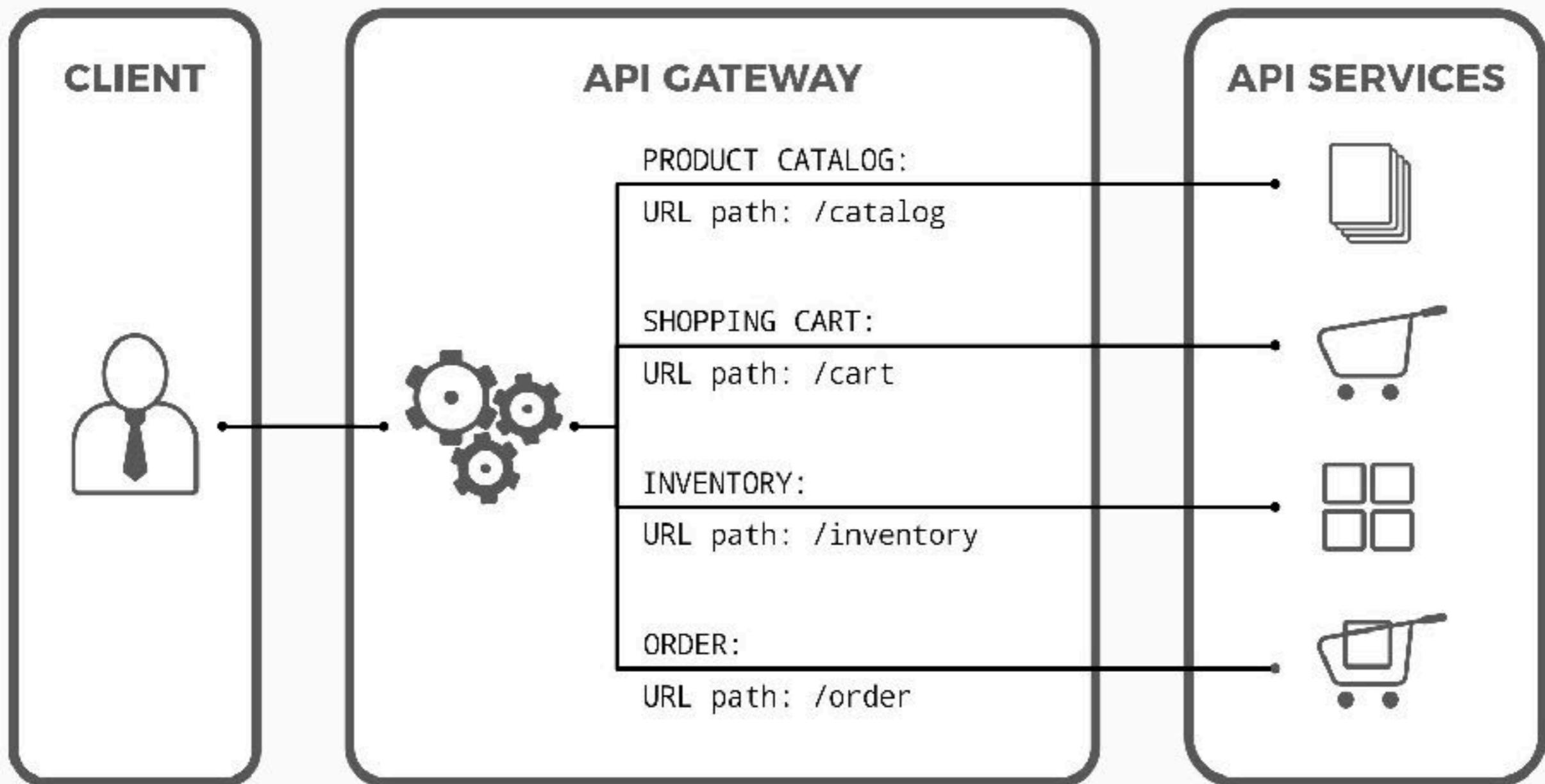
# Working with API gateway



<https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices>



# Working with API gateway



# Functions of API gateway

Authentication

Authorization

Rate limiting

Caching

Metrics collection

Request logging

Load balancing

Circuit breaking



# Benefits

Encapsulation interface structure  
Client talk to service via gateway  
Reduce round trip between service

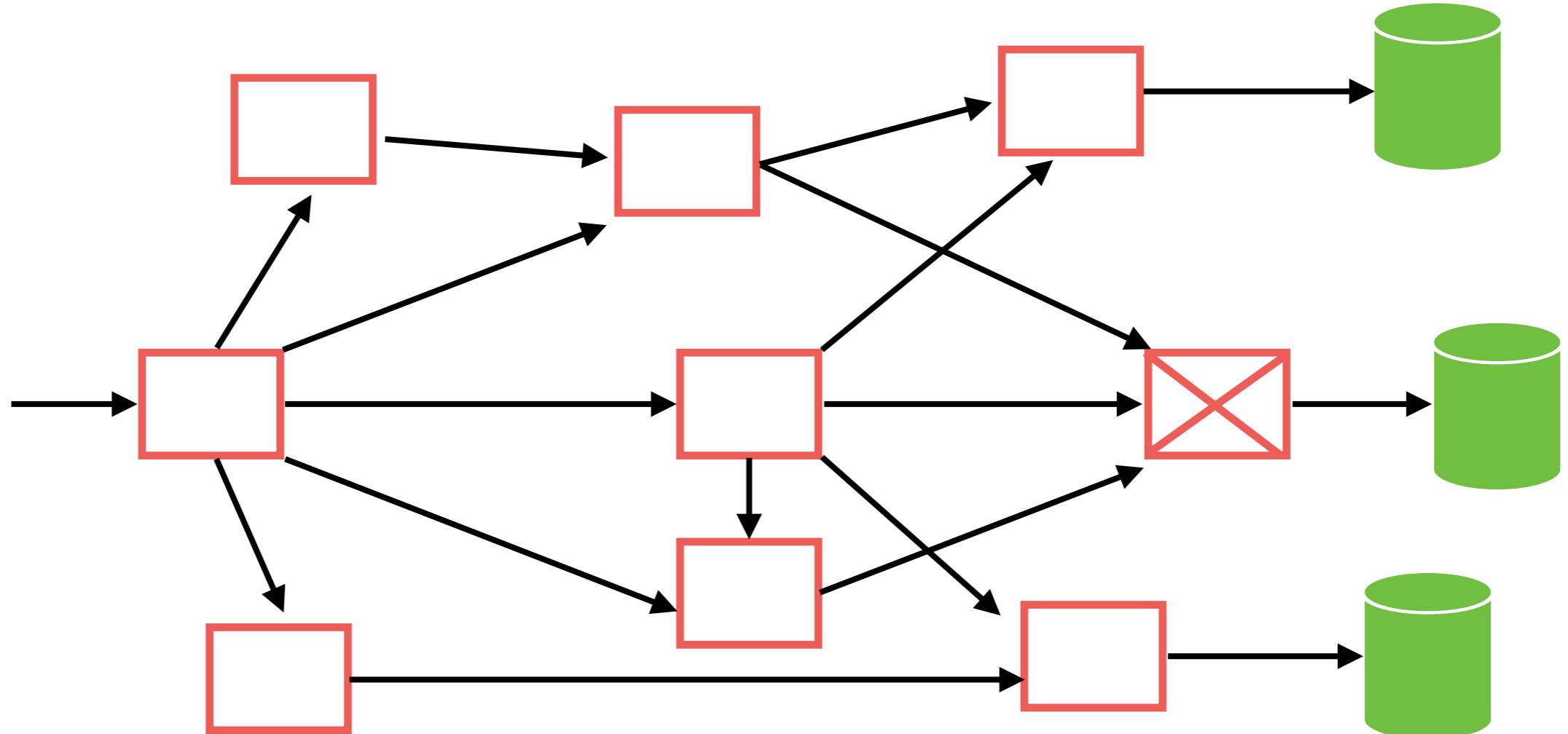


# Drawbacks

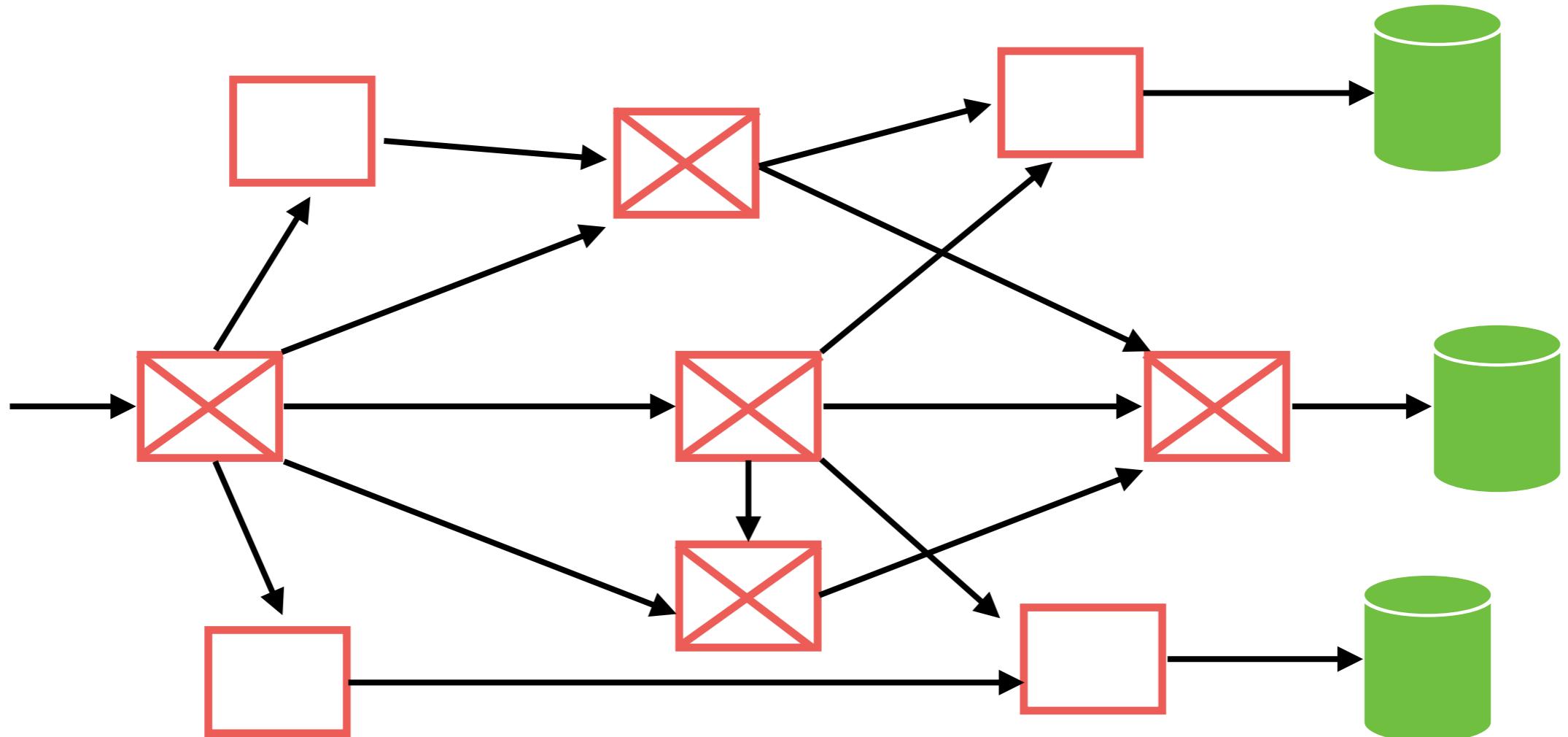
Development/Performance bottleneck  
Single point of failure  
Waiting for update data in gateway



# Failure !!



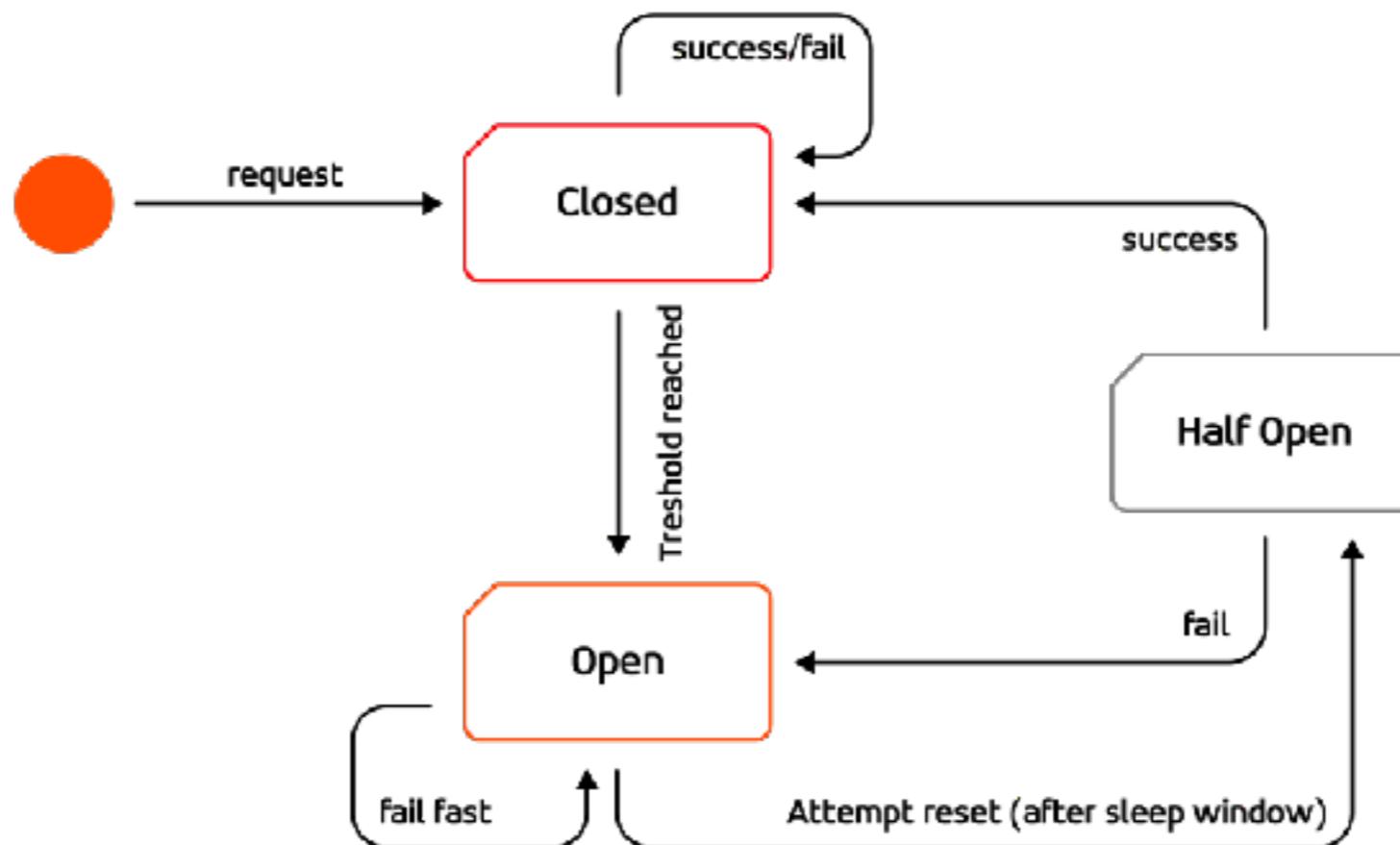
# Cascading Failure !!



# Circuit Breaker pattern

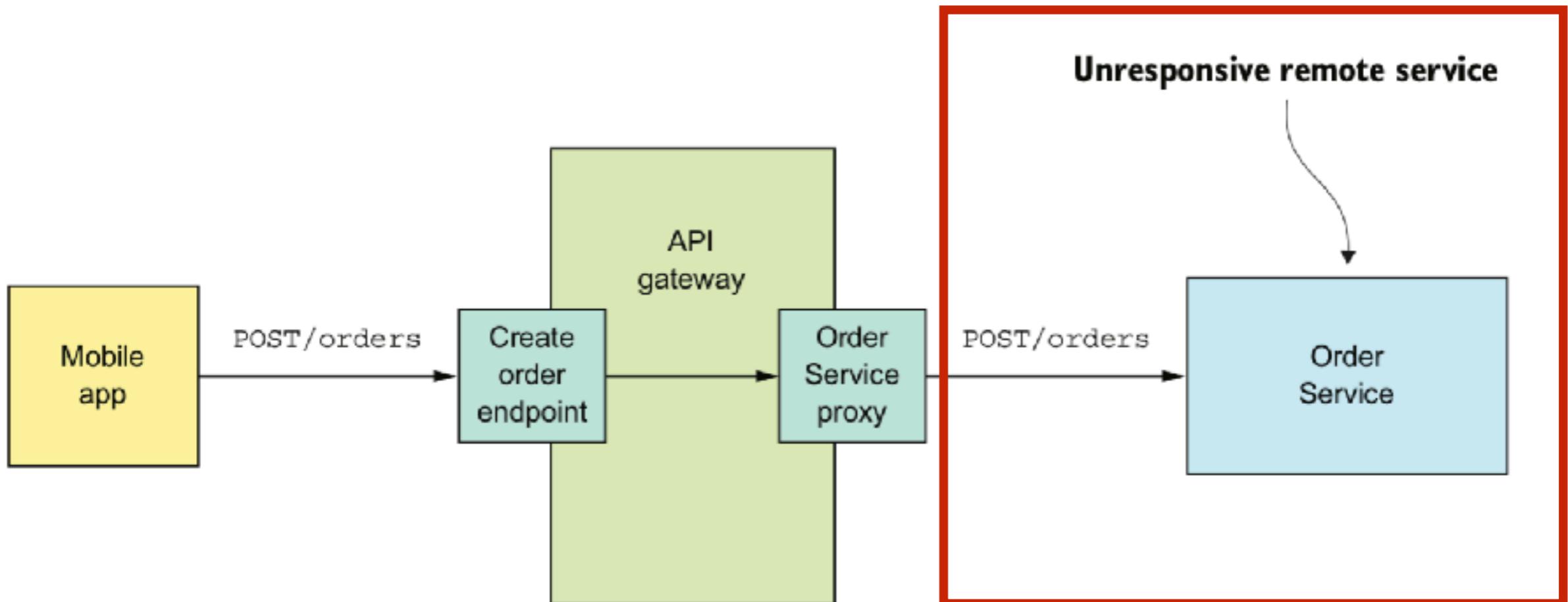
Track the number of success and failure

***If error rate exceed some threshold  
then enable circuits breaker***



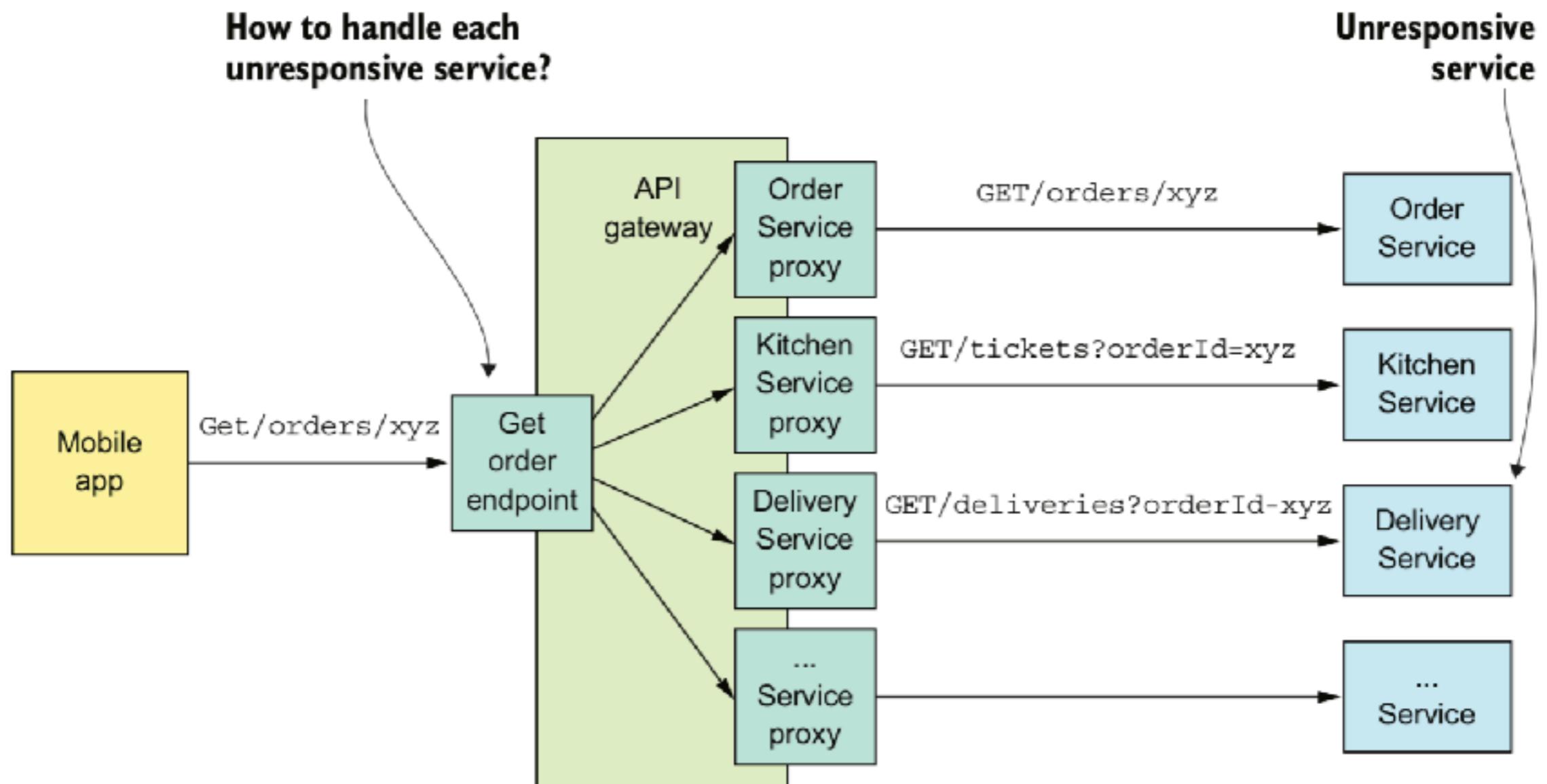
# Circuit Breaker pattern

How to handling partial failure ?

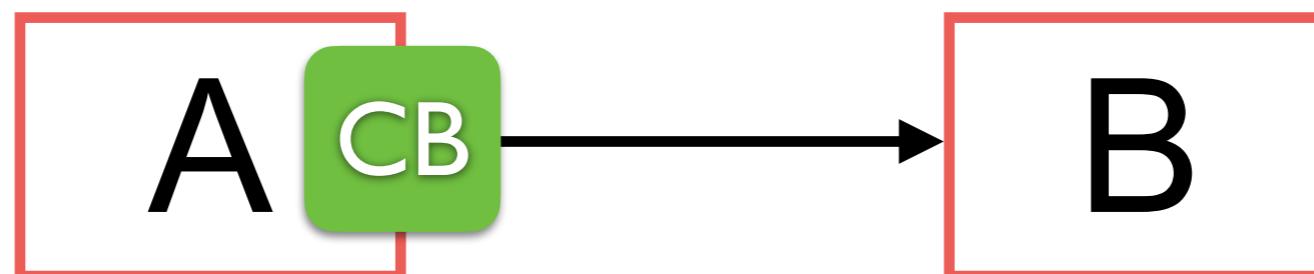
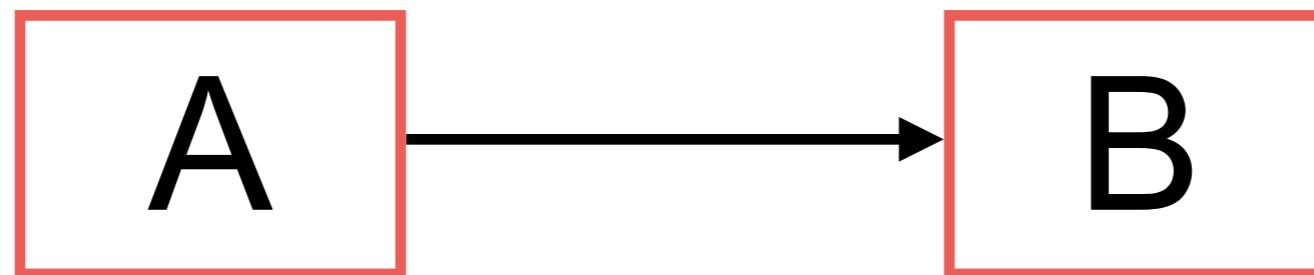


# Circuit Breaker pattern

## How to handling partial failure ?



# Circuit Breaker pattern

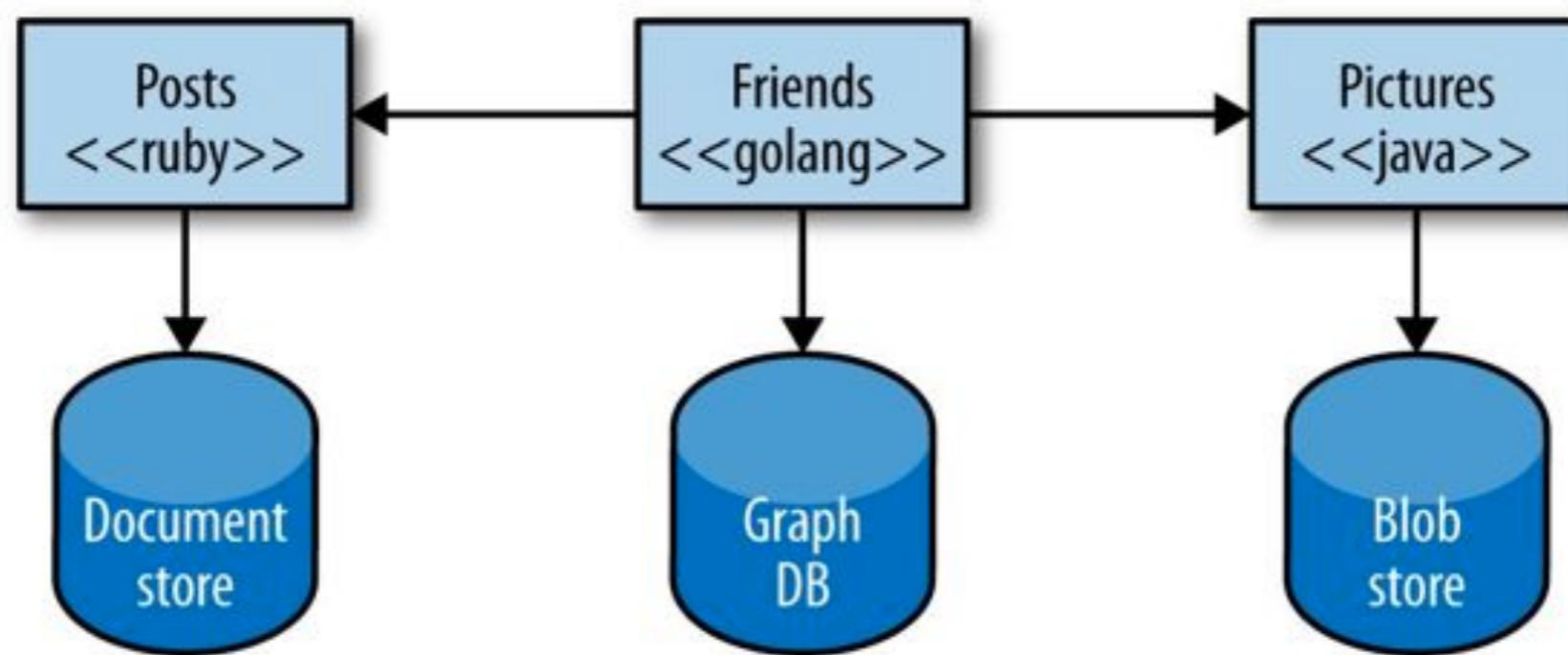


# Key Benefits

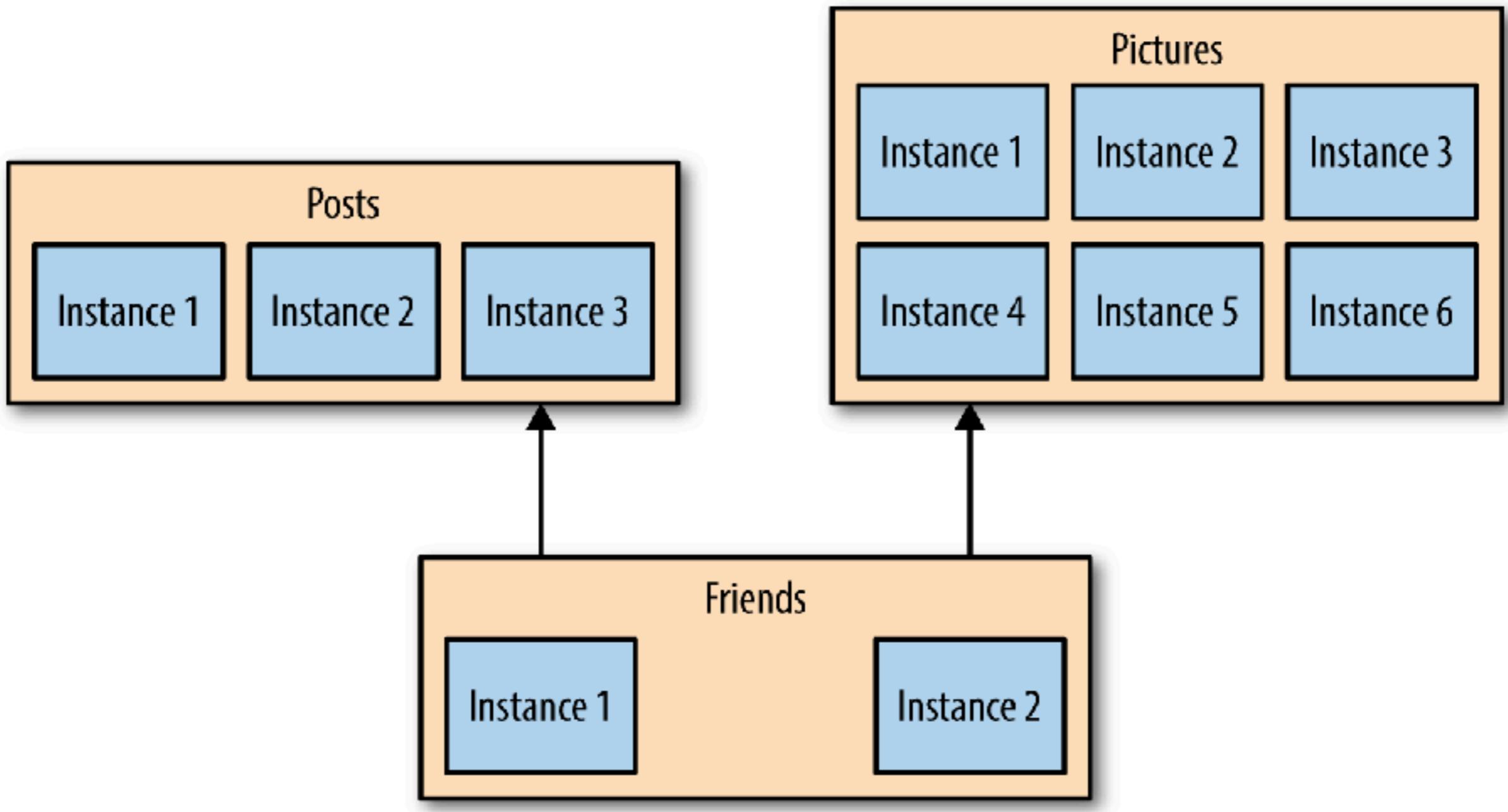


# 1. Technology heterogeneous

The right tool for each job  
Allow easy experimenting and adoption of new technology

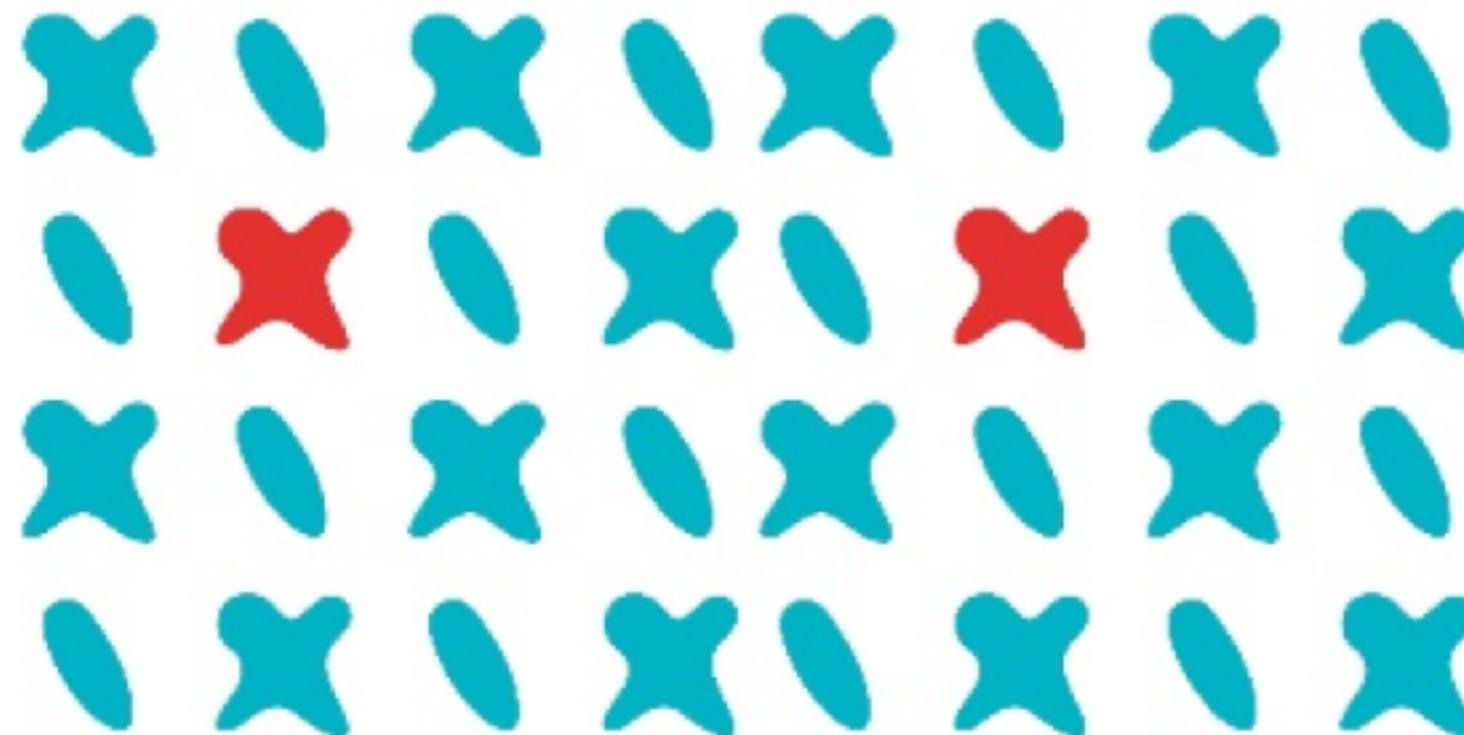


# 2. Scaling

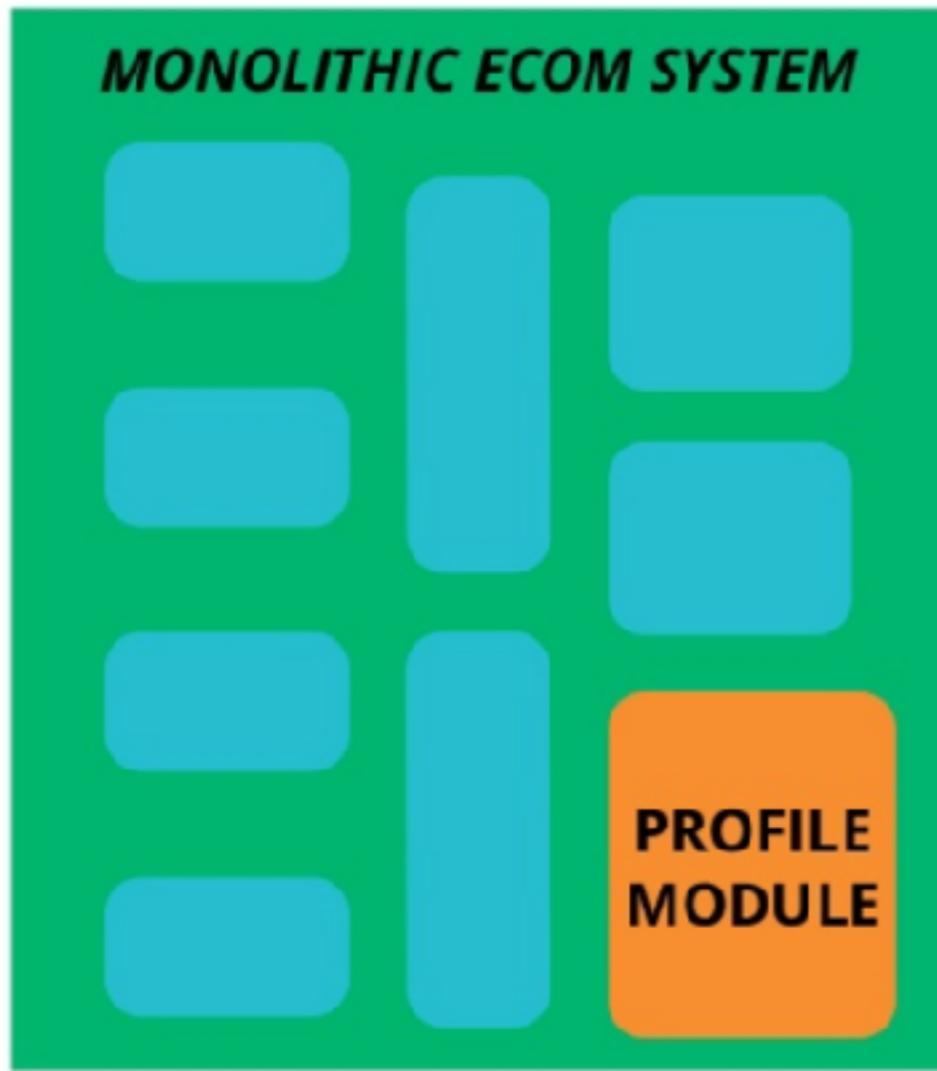


# 3. Ease of deployment

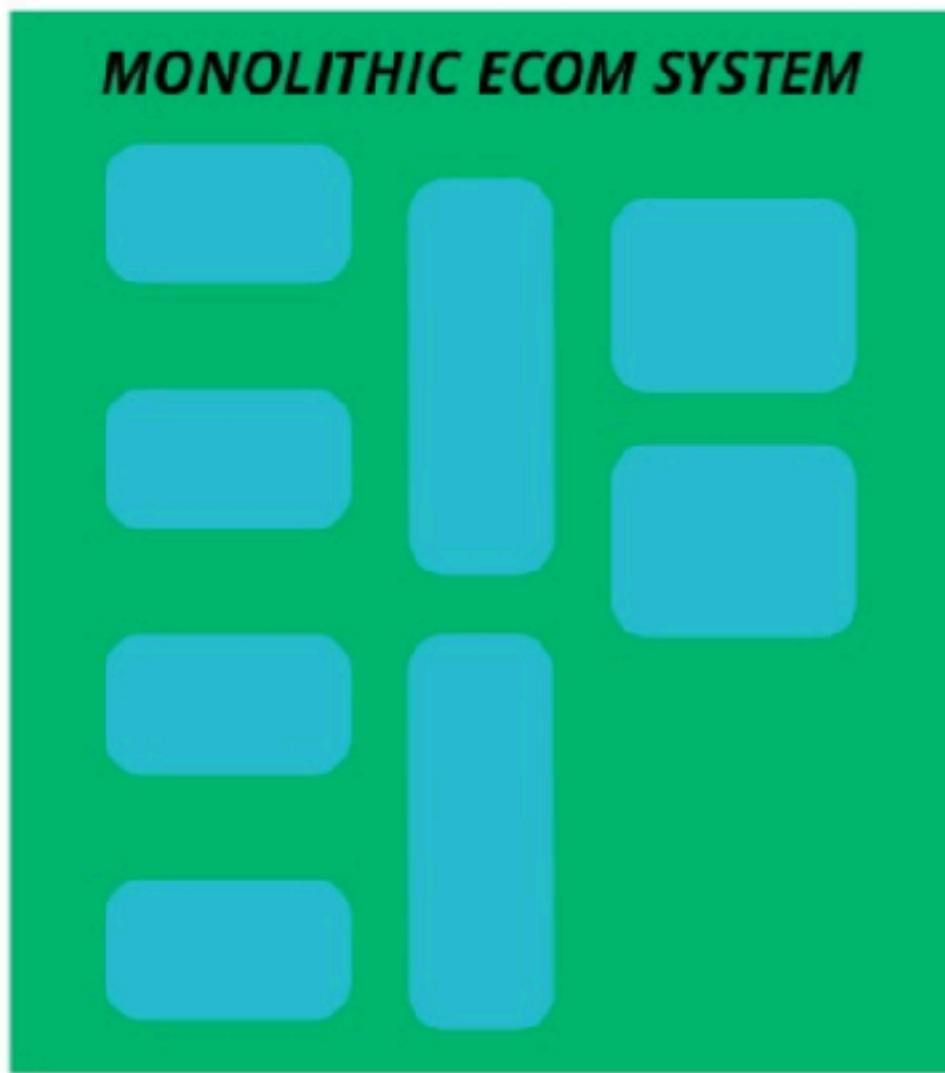
Deploys are faster, independent and problems can be isolated more easily



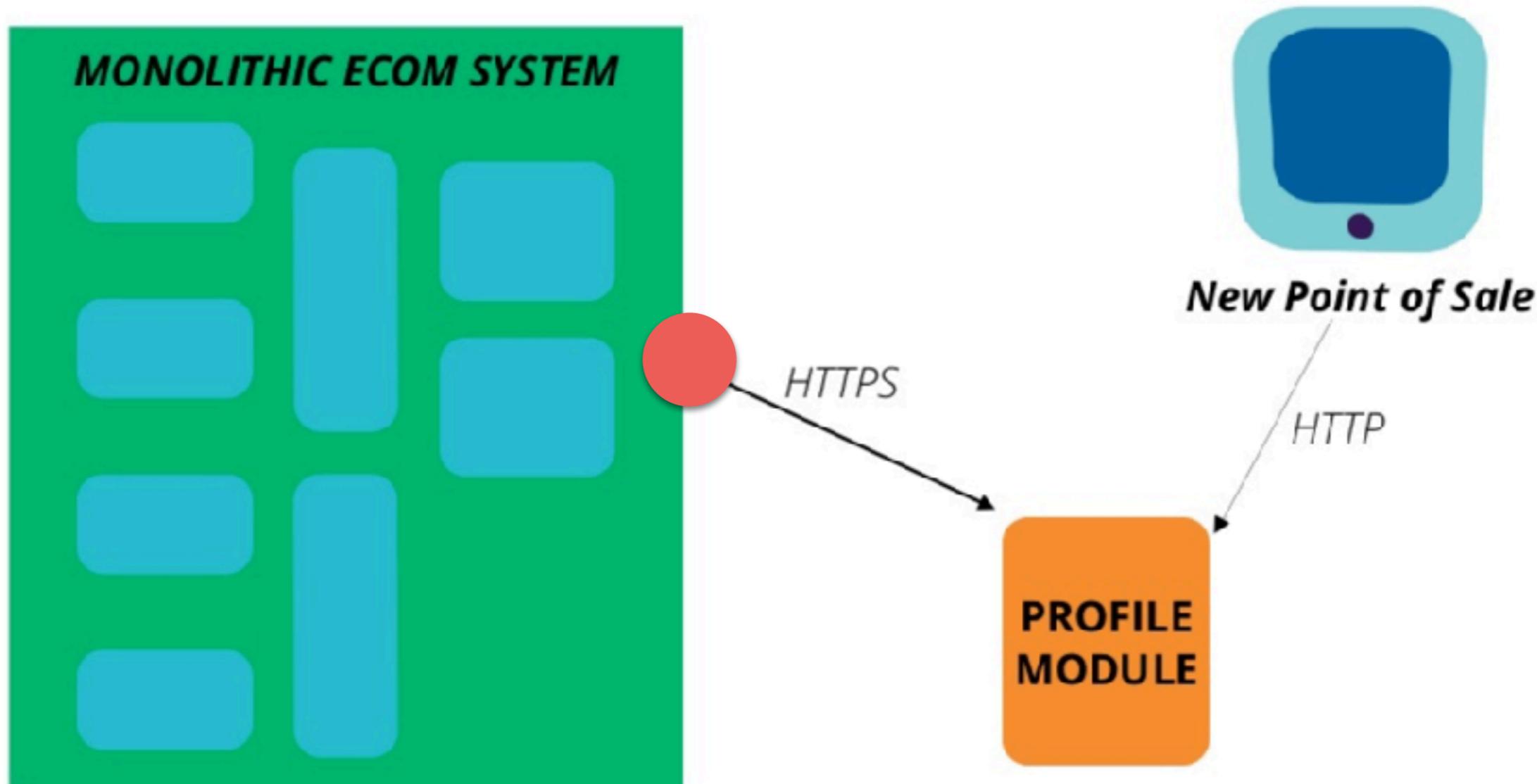
# 4. Composability and replaceability



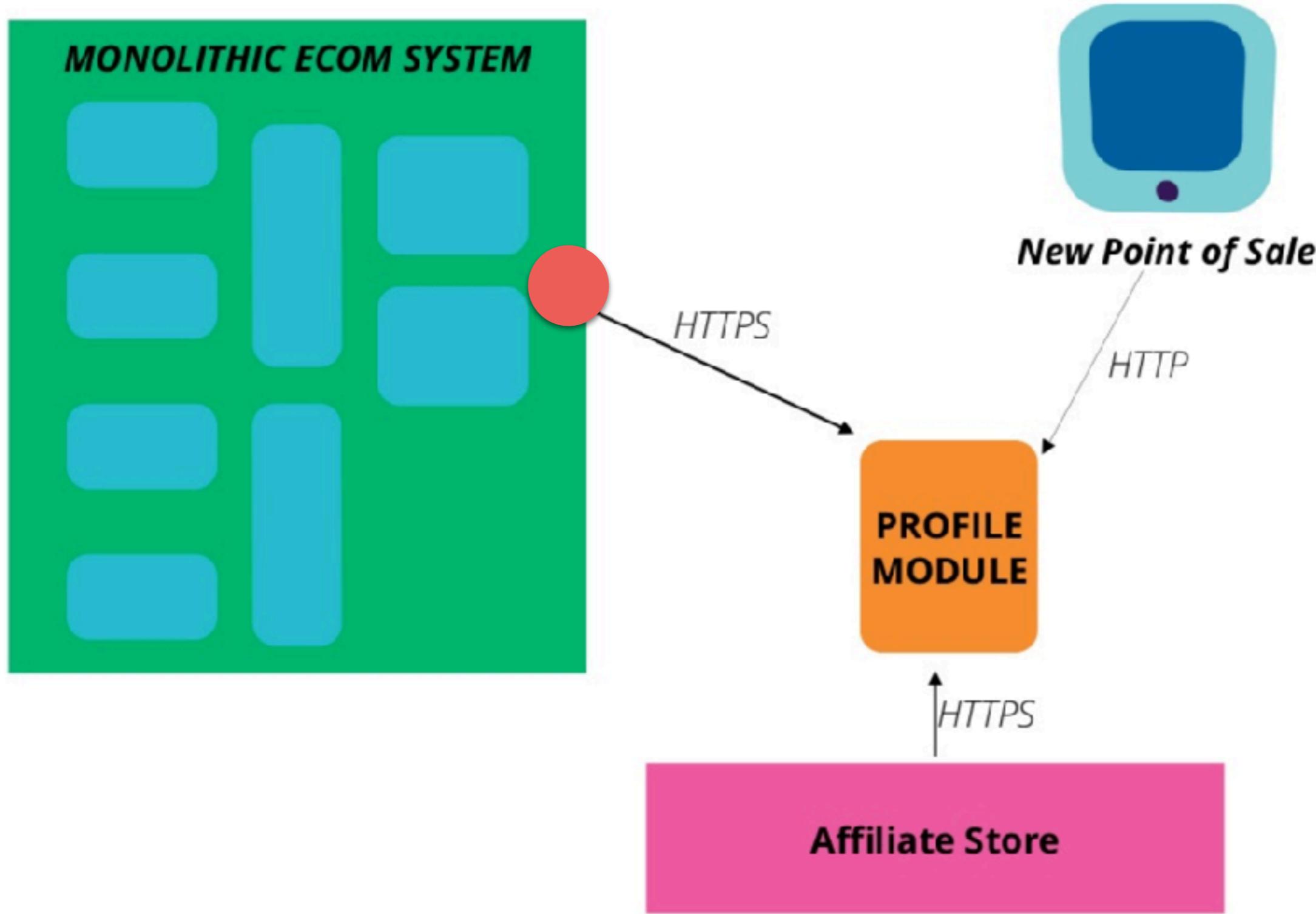
# 4. Composability and replaceability



# 4. Composability and replaceability



# 4. Composability and replaceability



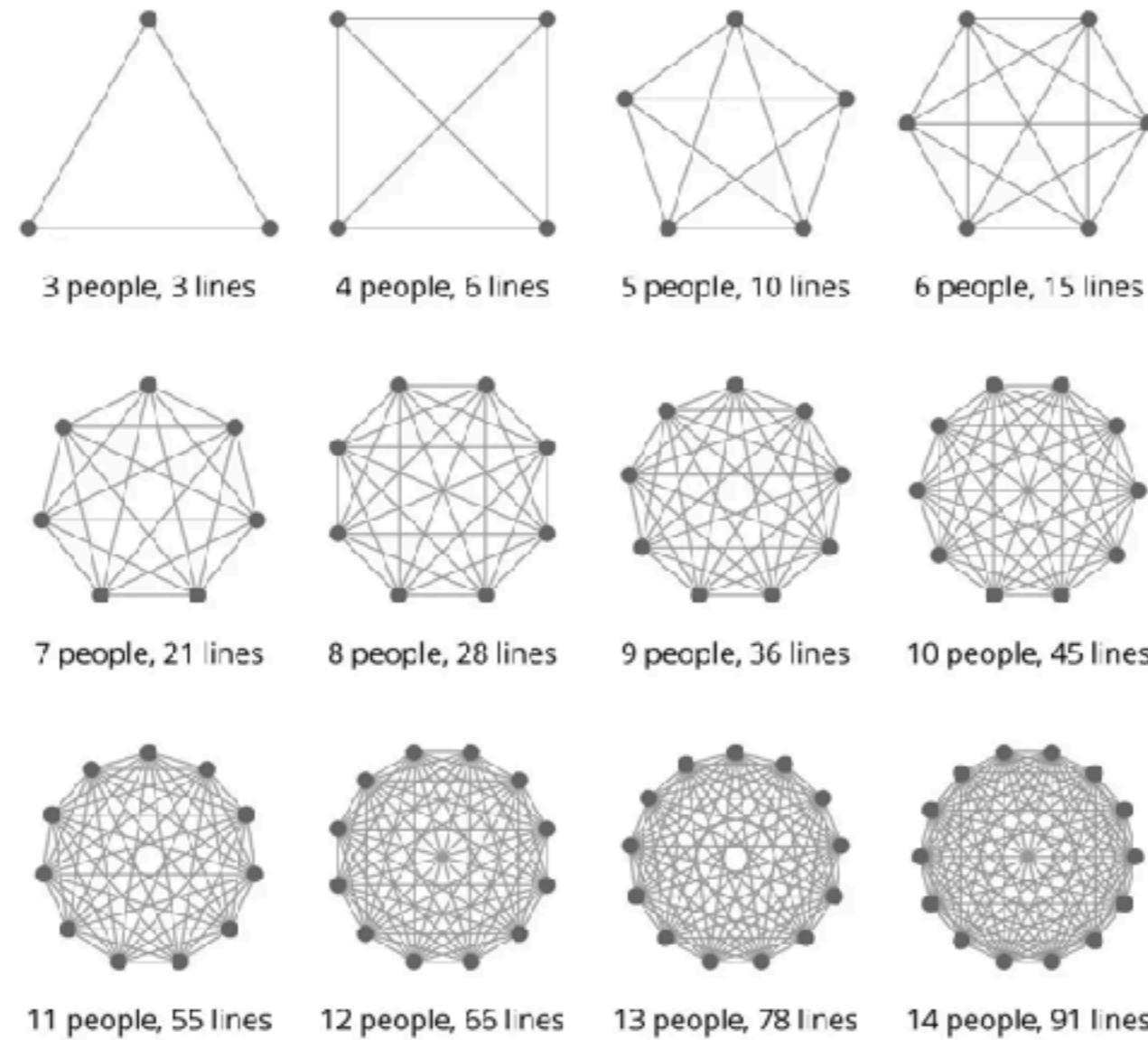
# 5. Organization alignment

Small teams and smaller codebases



# Small team !!

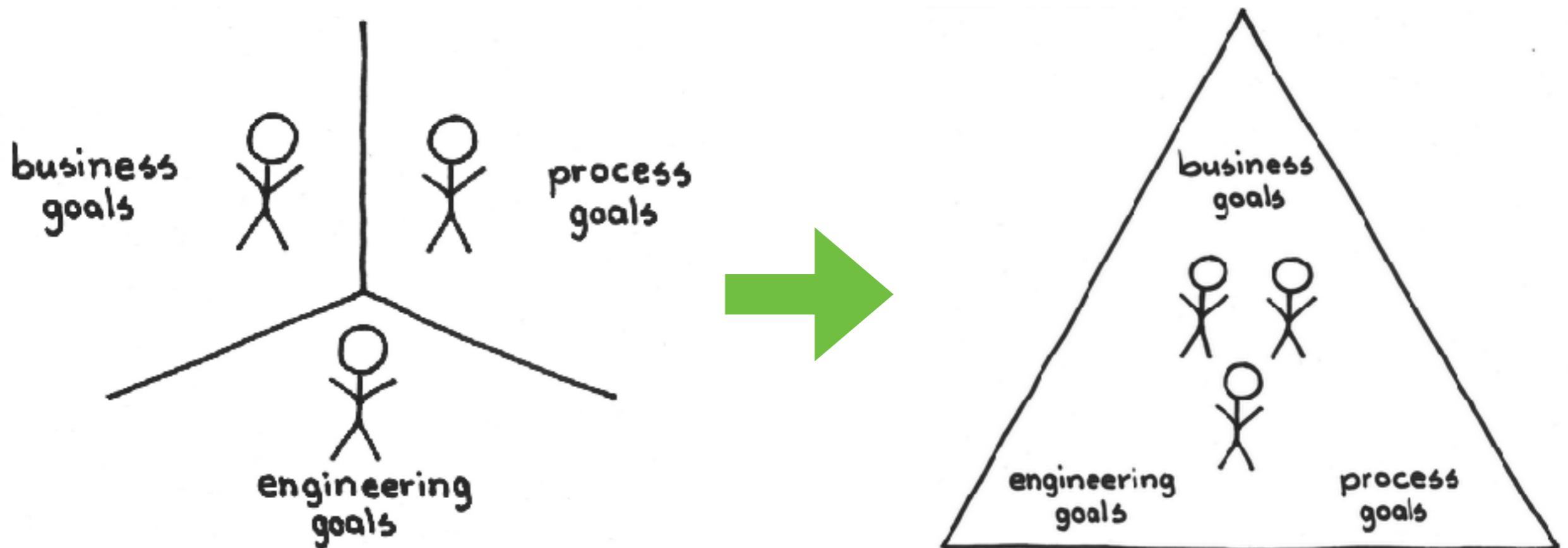
## Line of communication and team size



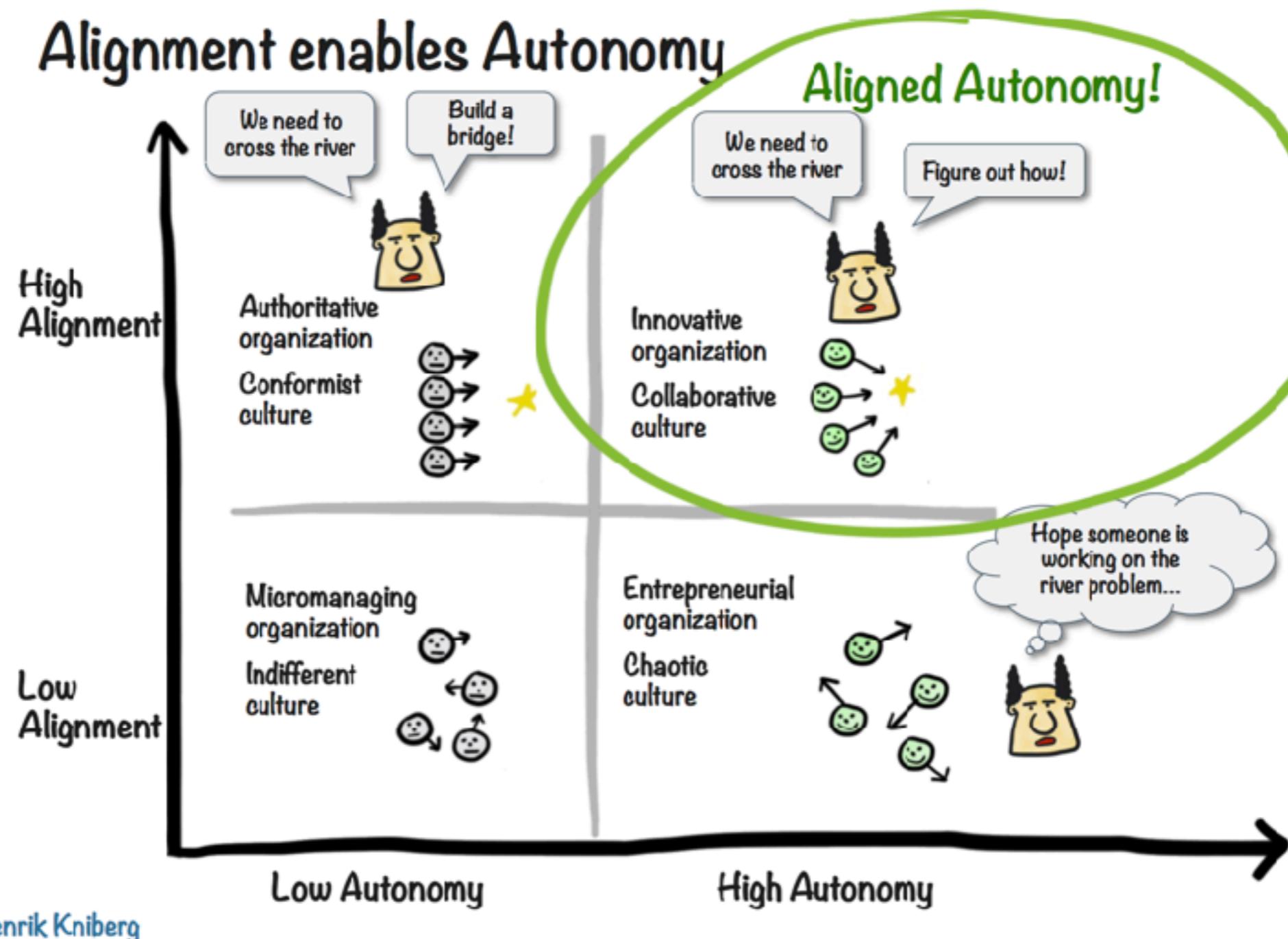
<https://www.leadingagile.com/2018/02/lines-of-communication-team-size-applying-brooks-law>



# Autonomous team



# Autonomous team



# Autonomous team



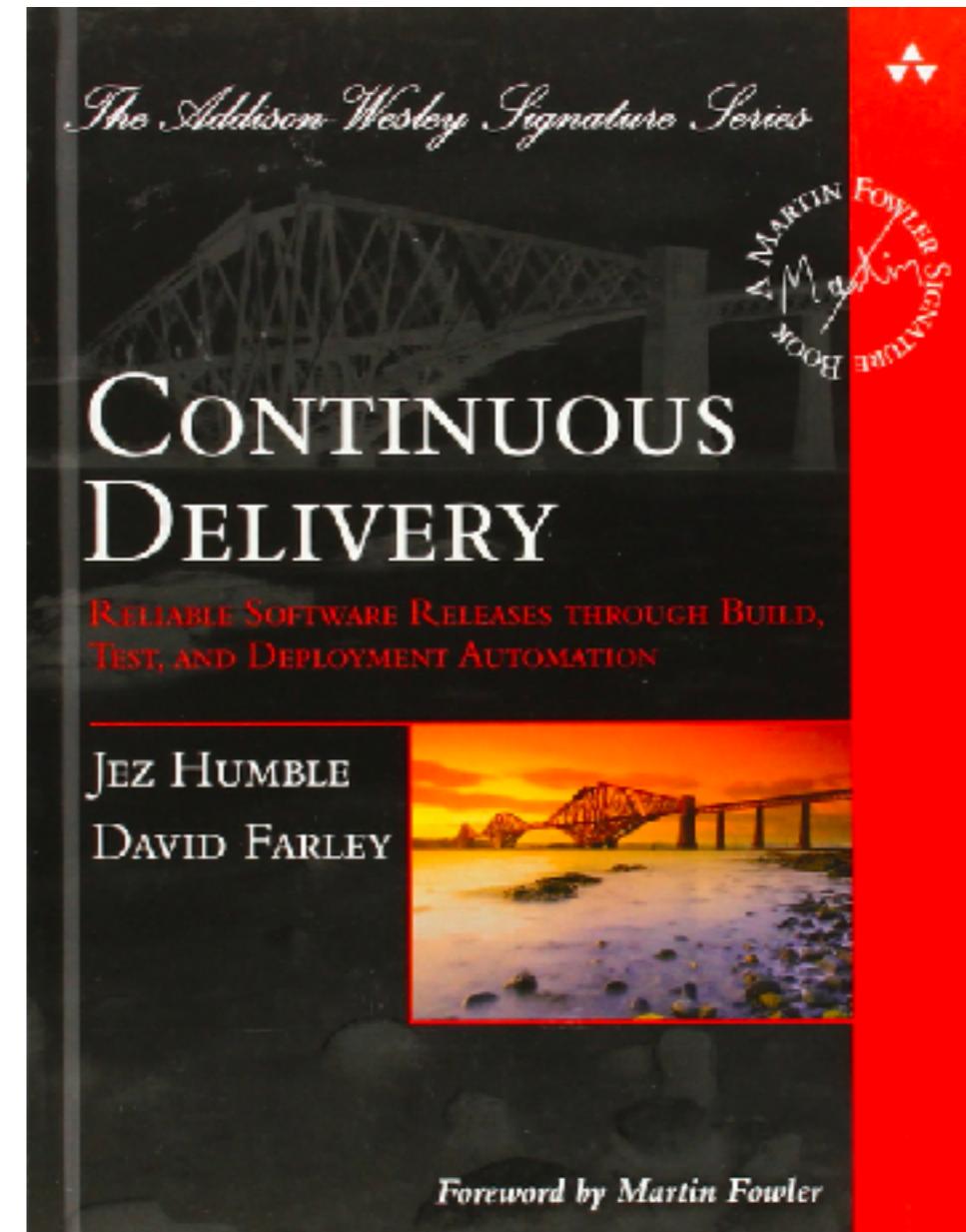
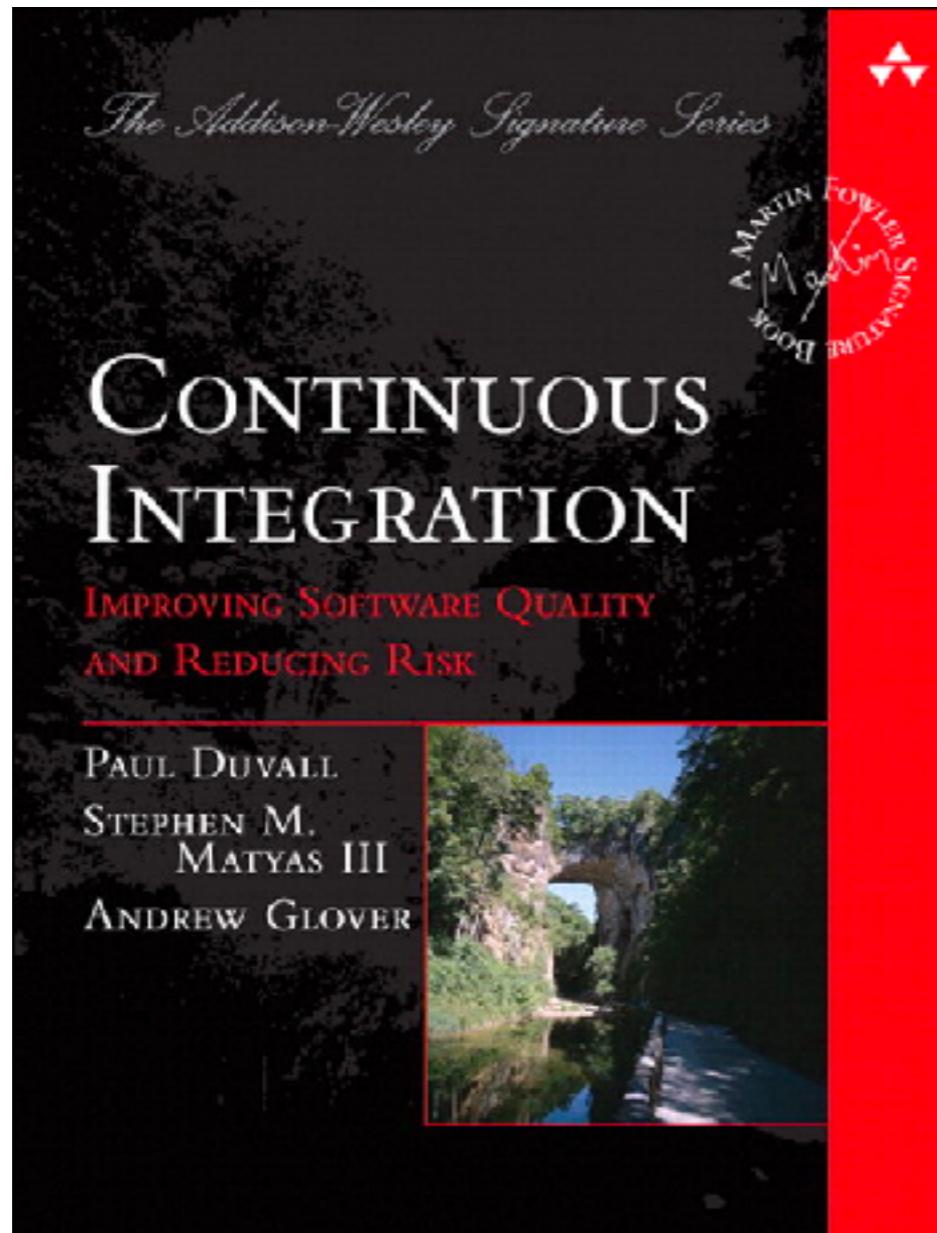
# 6. Enable Continuous Delivery

A part of DevOps

Set of practices for the **rapid, frequent and reliable delivery** software



# Continuous Delivery/Deployment



# Continuous Delivery/Deployment

Testability  
Deployability

Autonomous team and loose coupling



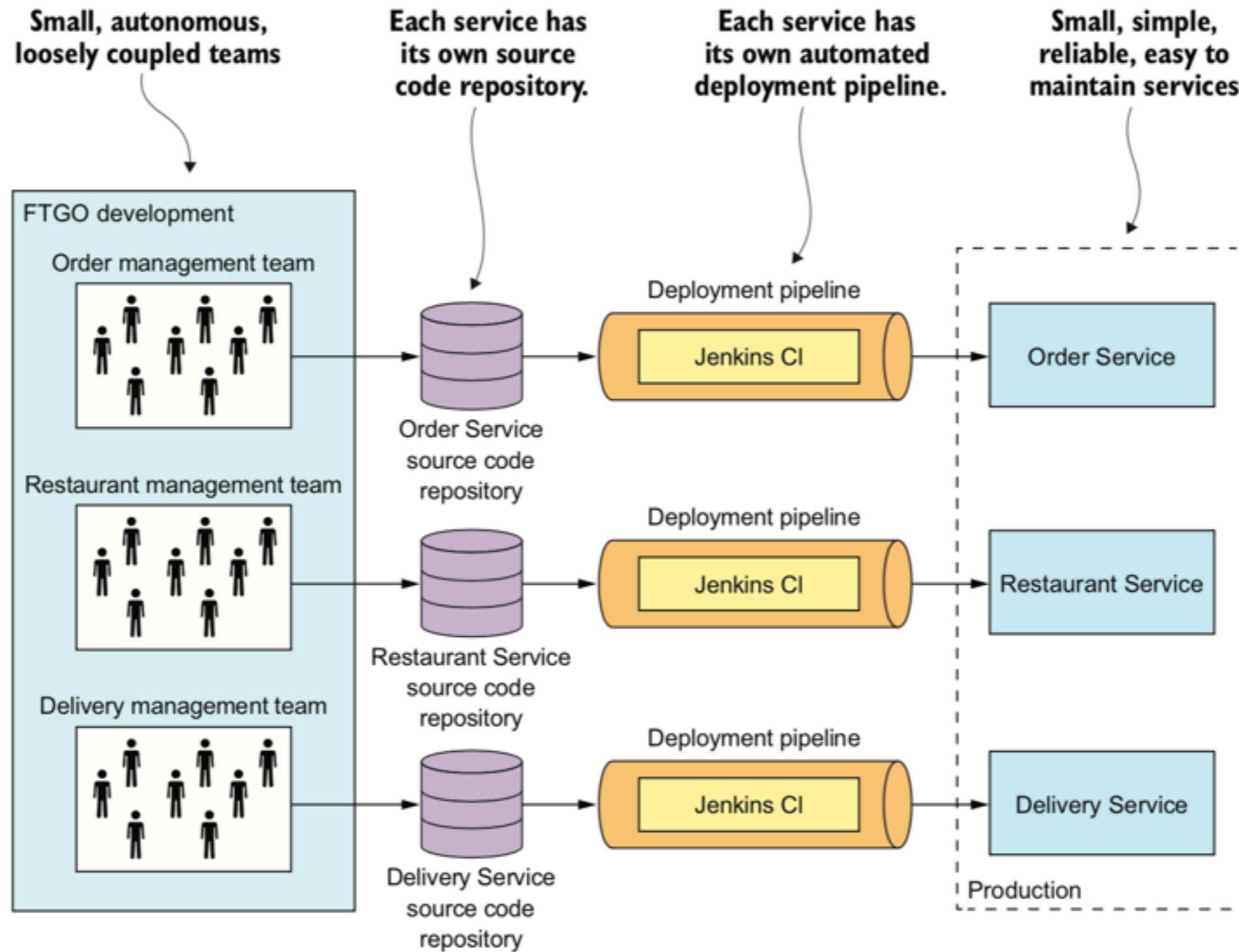
# Benefits of Continuous Delivery

Reduce time to market

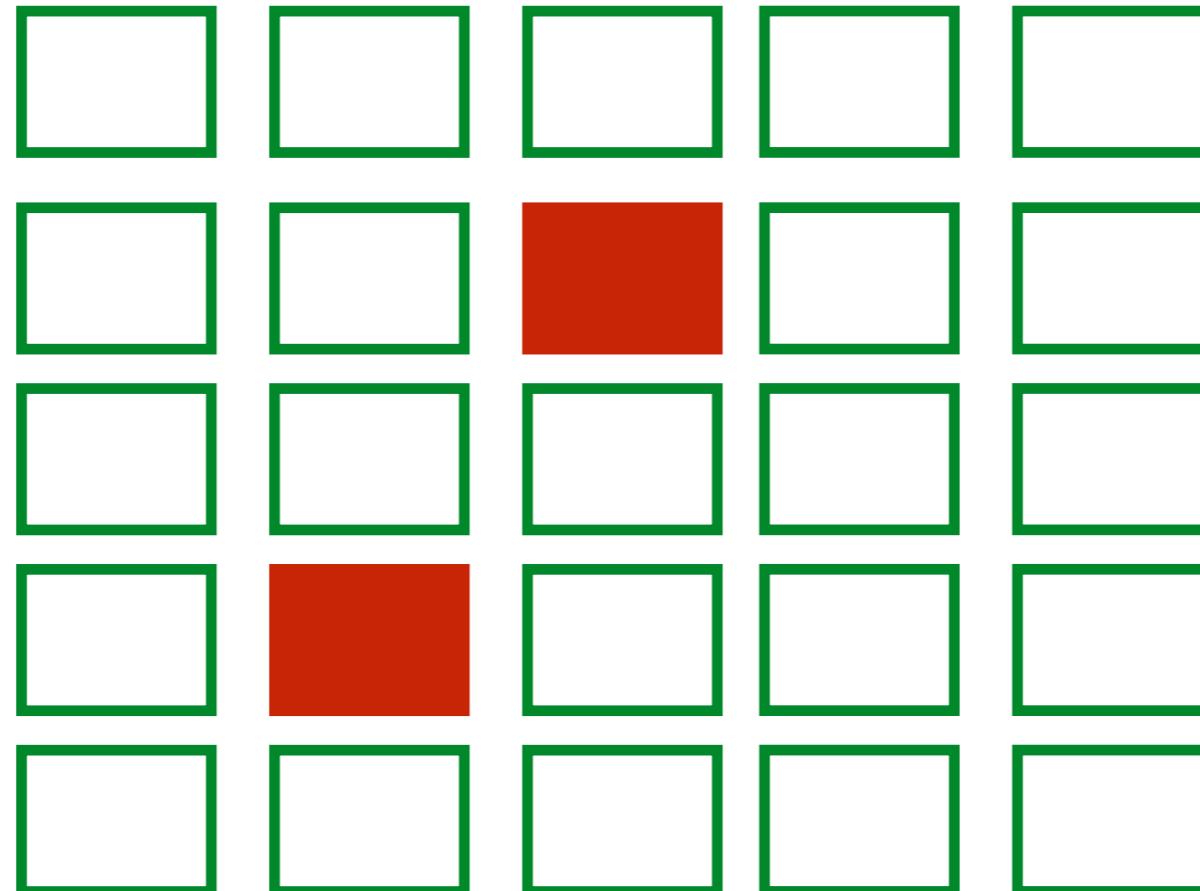
Enable **business** to provide reliable service  
**Employee** satisfaction is higher



# Continuous Delivery for service



# 7. Better Fault isolation



# Drawbacks of Microservice



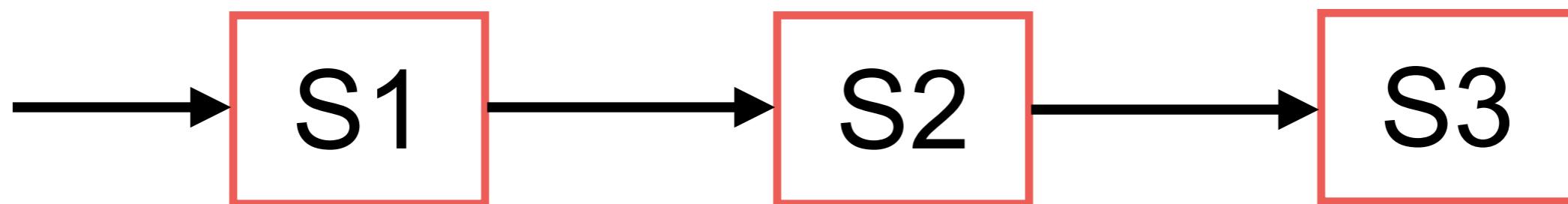
# Drawbacks of Microservice

Find the right set of services  
Distributed systems are complex  
How to develop, testing and deploy ?



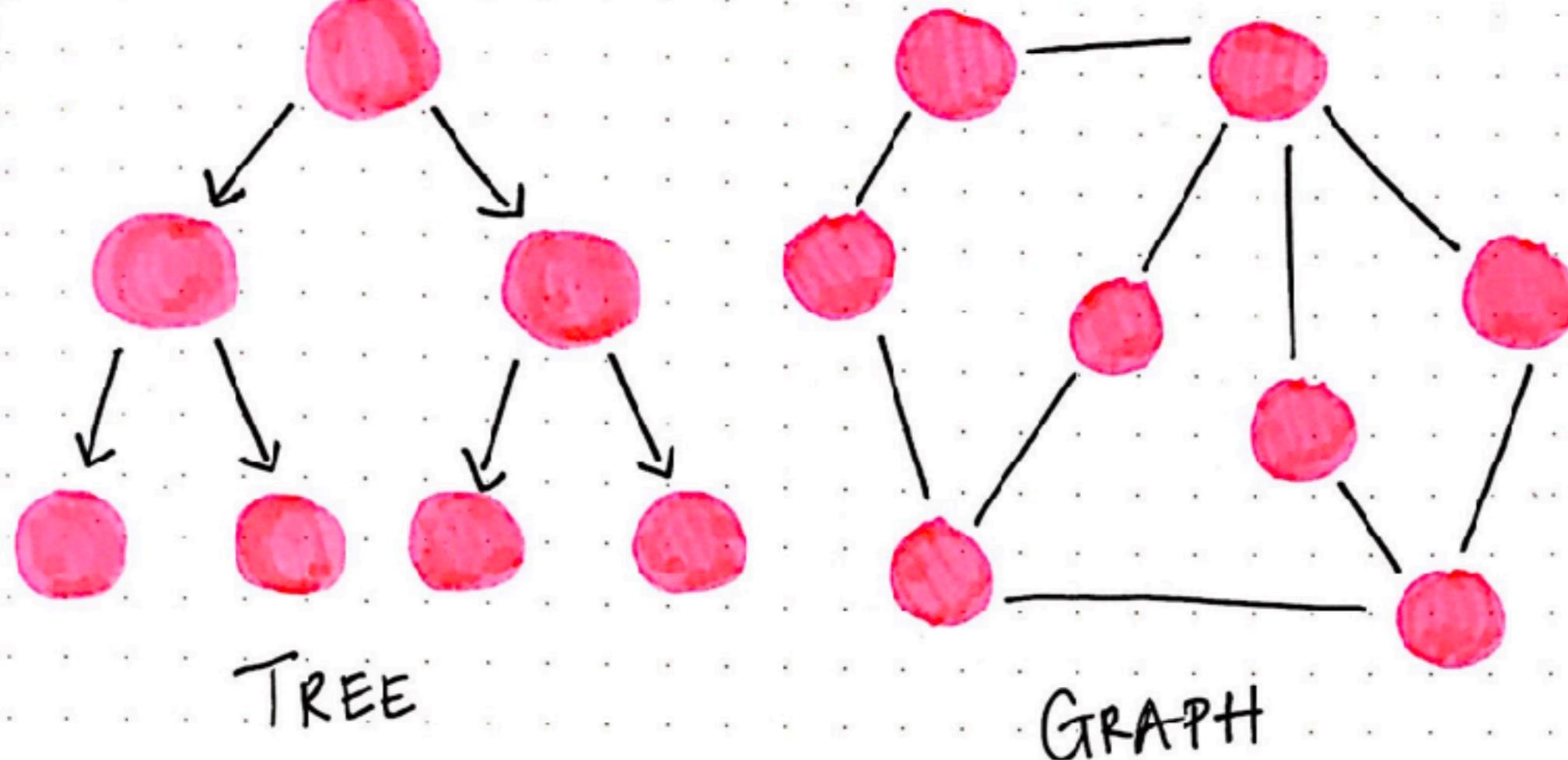
# Drawbacks of Microservice

Deploy feature that required multiple service ?



# Service Principles

Minimize the depth of the service call-graph



<https://github.com/Yelp/service-principles>



# Service Principles

Minimize the number of services owned by your team

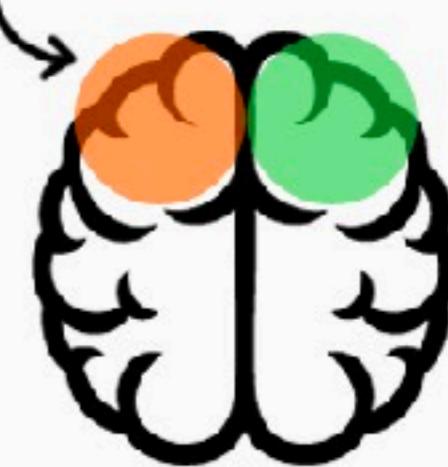


SINGLE TASK



TWO TASKS

THE 3RD TASK IS REPLACING  
ONE OF THE 3 TASKS



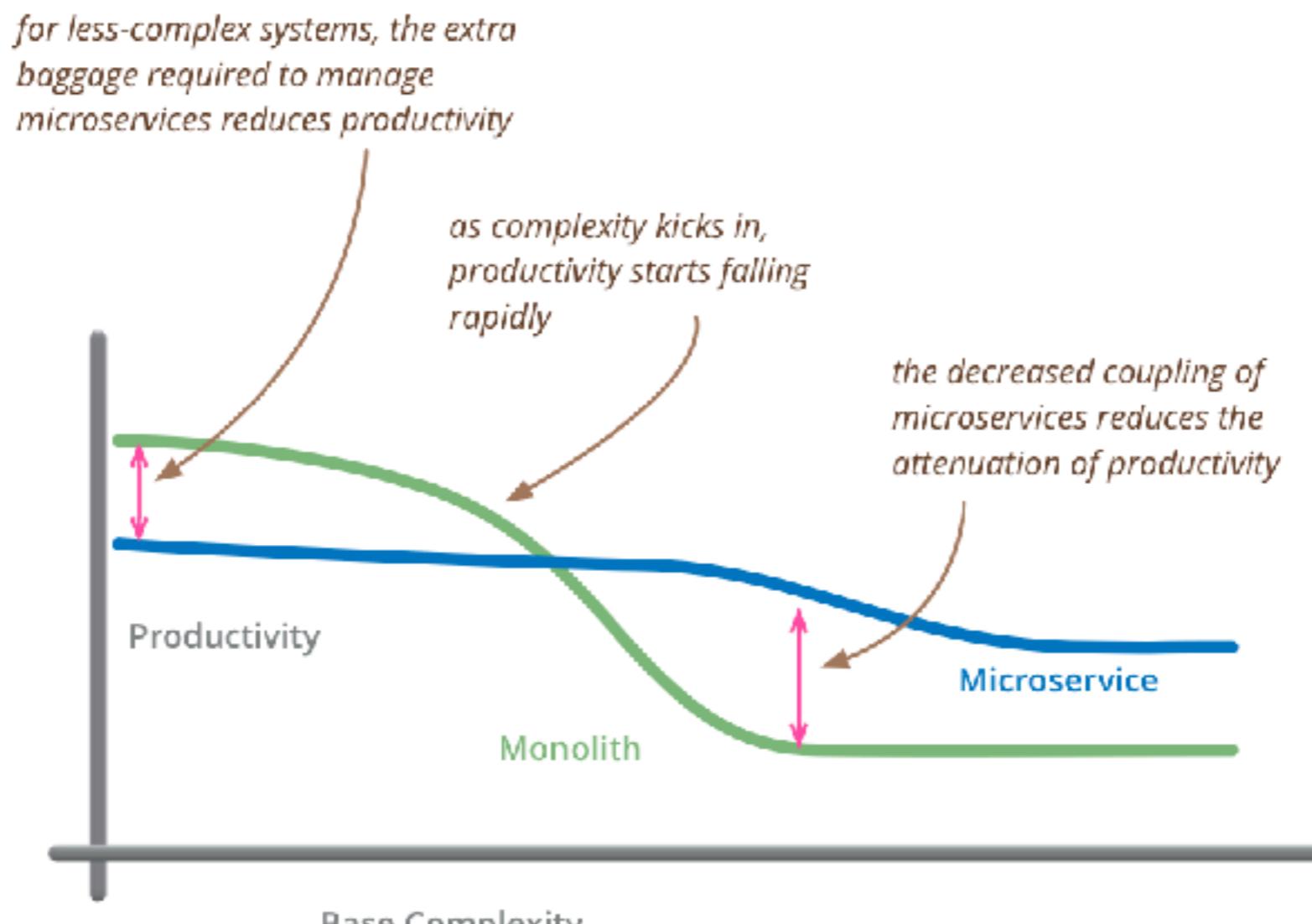
THREE TASKS

<https://github.com/Yelp/service-principles>



# Drawbacks of Microservice

Decide to use when adopt is difficult !!



# Microservice architecture patterns

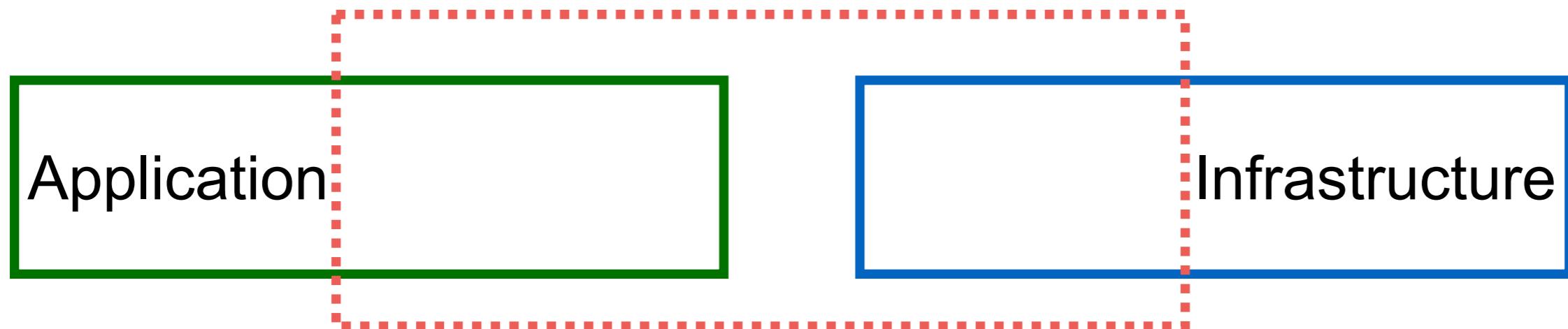


# 3 Layers of service patterns

Application patterns

Application infrastructure pattern

Infrastructure pattern



<https://microservices.io/>



# Decompose application into services



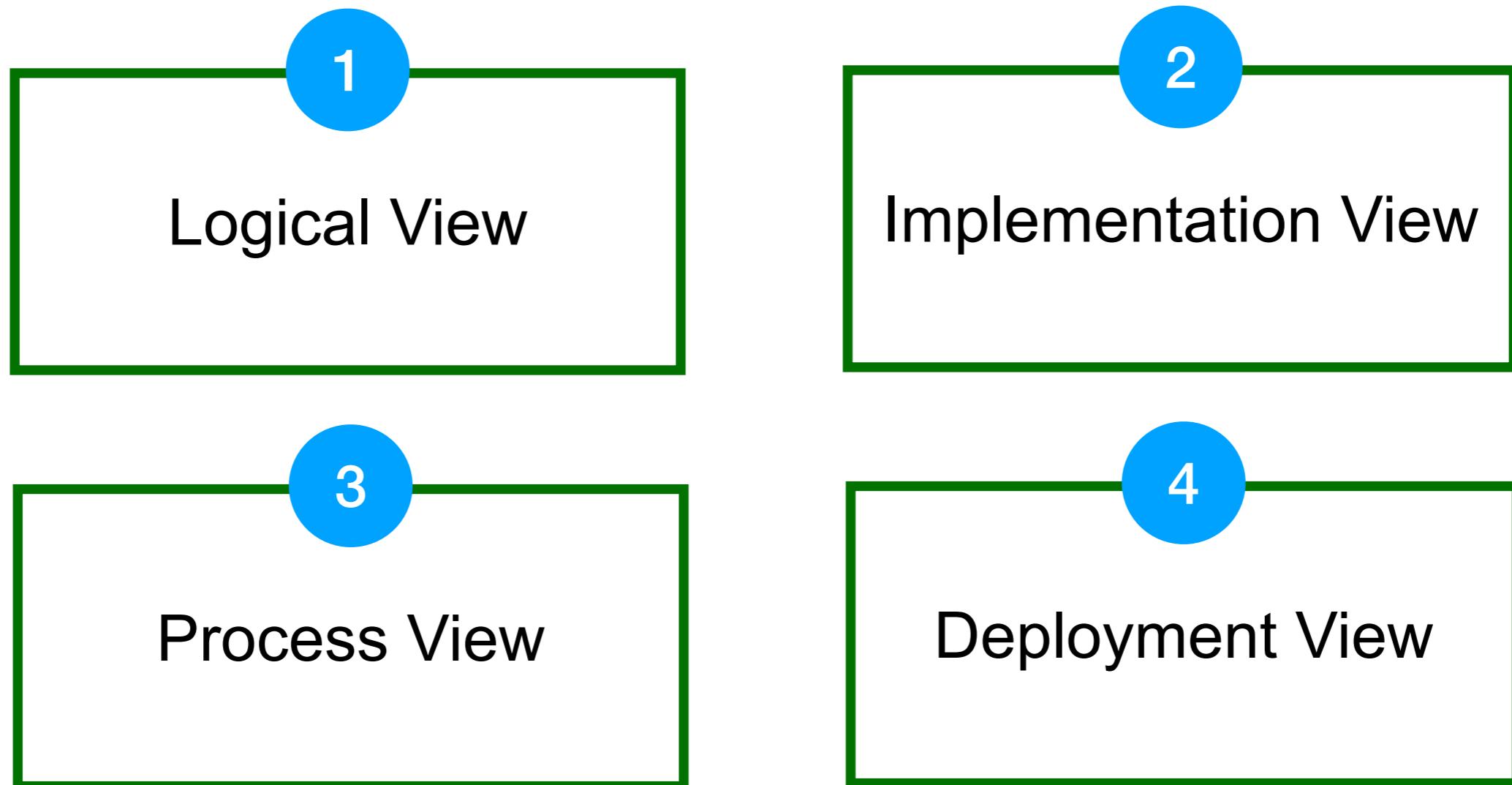
Premature splitting is the root of  
all evil.



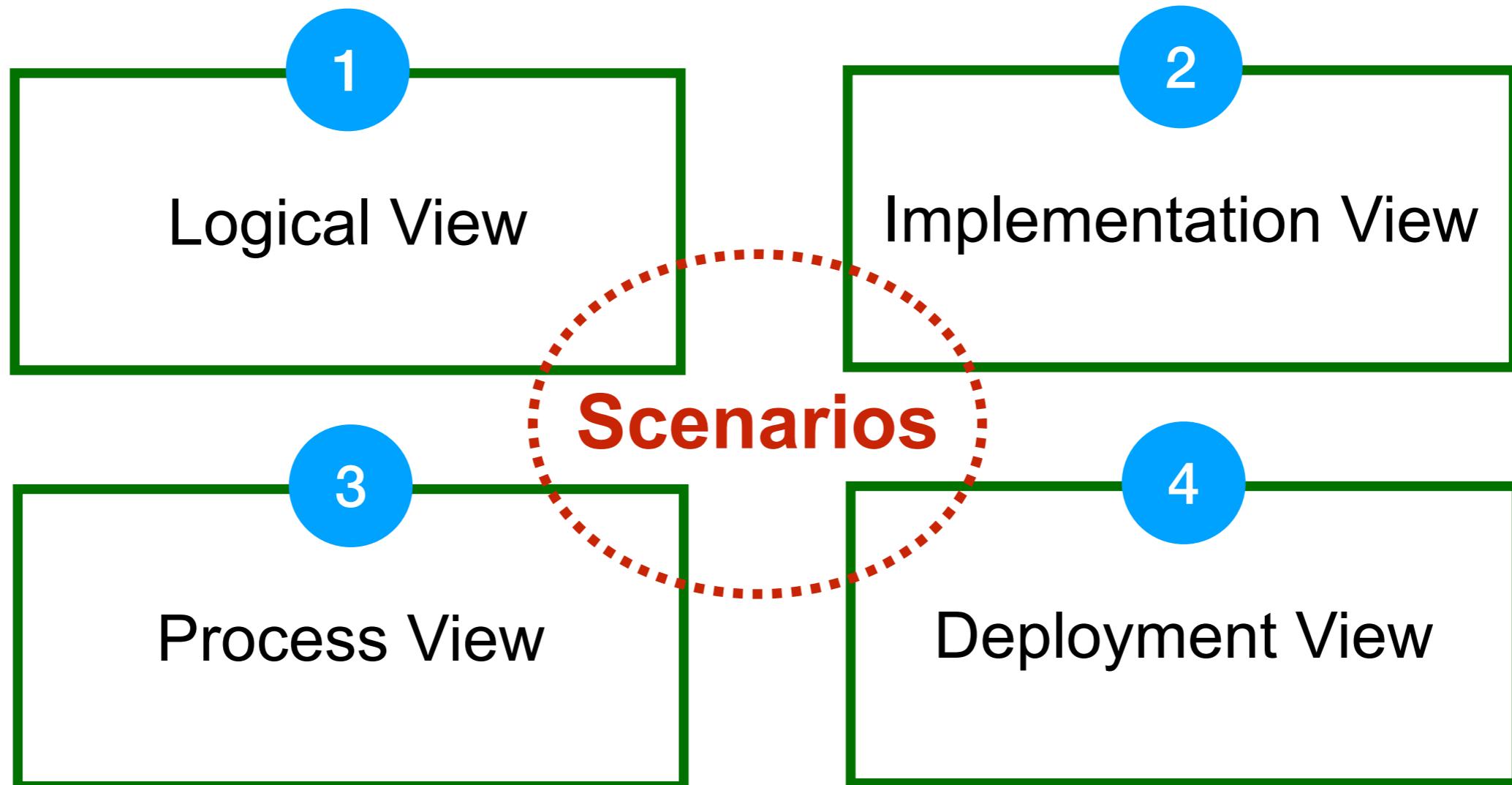
Every time you make the decision  
to split out a new microservice,  
there's a **risk** of ending up with a  
bloated app.



# 4 View model of Software Architecture



# 4 View model of Software Architecture



# Decompose application into services

By business capability ?  
By subdomain/technical ?



# Step to define services

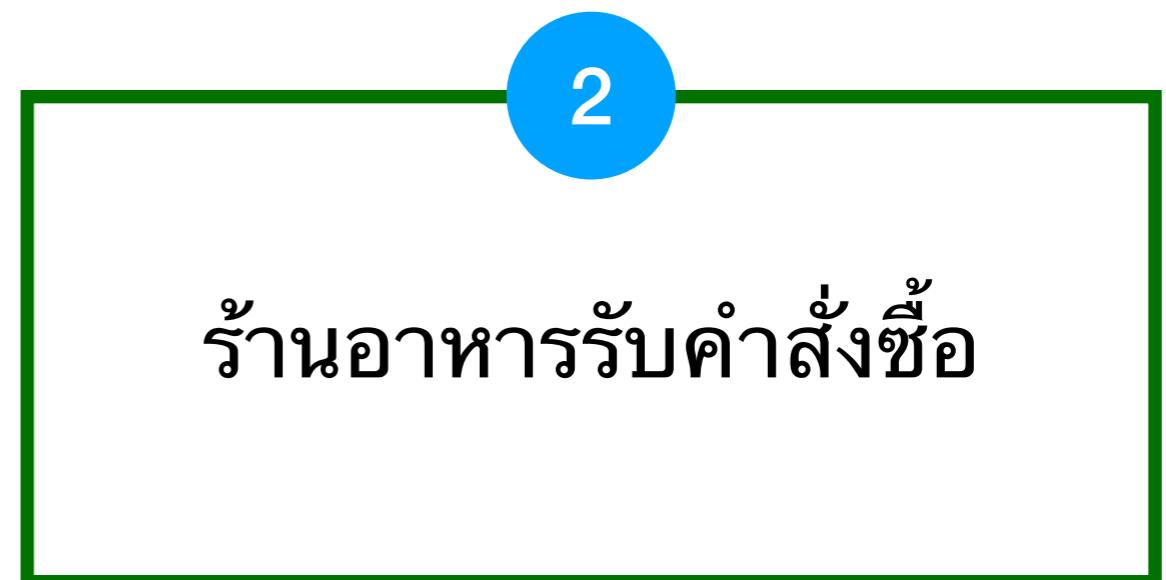
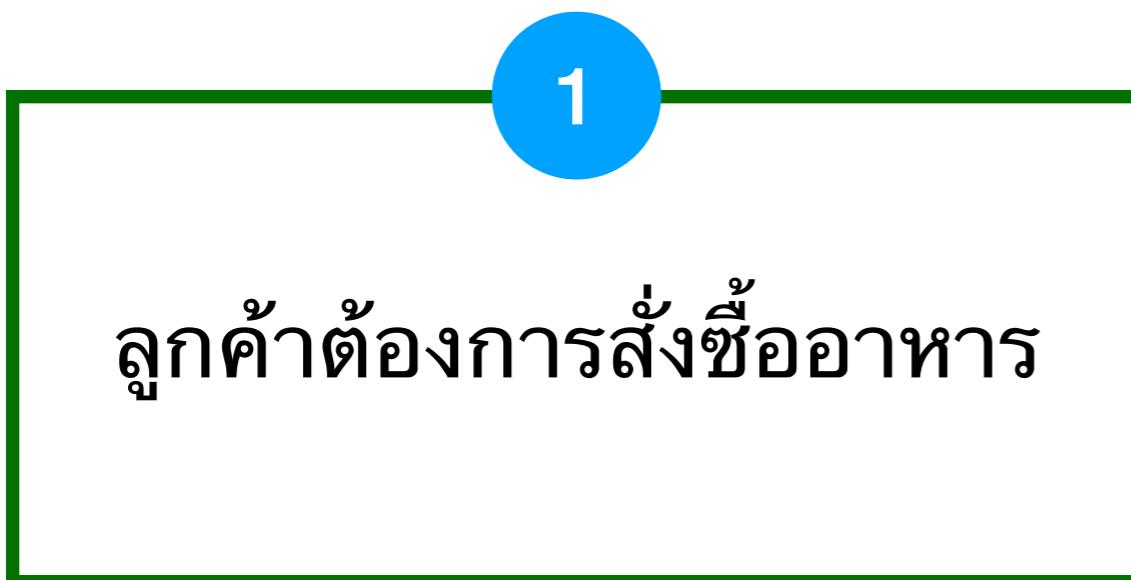
1. Identify system operations
2. Identify services
3. Define service APIs and collaboration



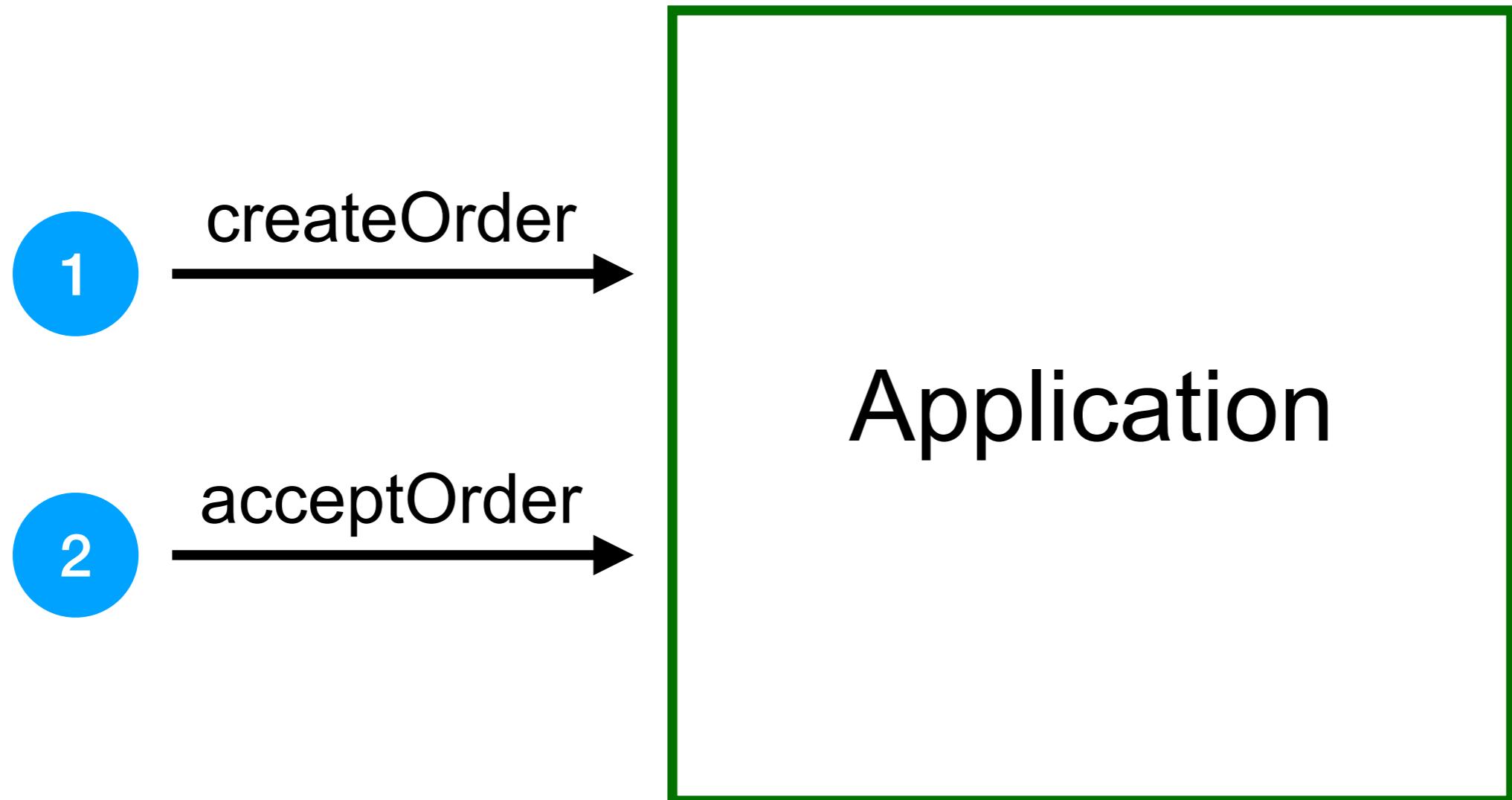
# 1. Identify system operations

Start with functional requirements

User story



# 1. Identify system operations



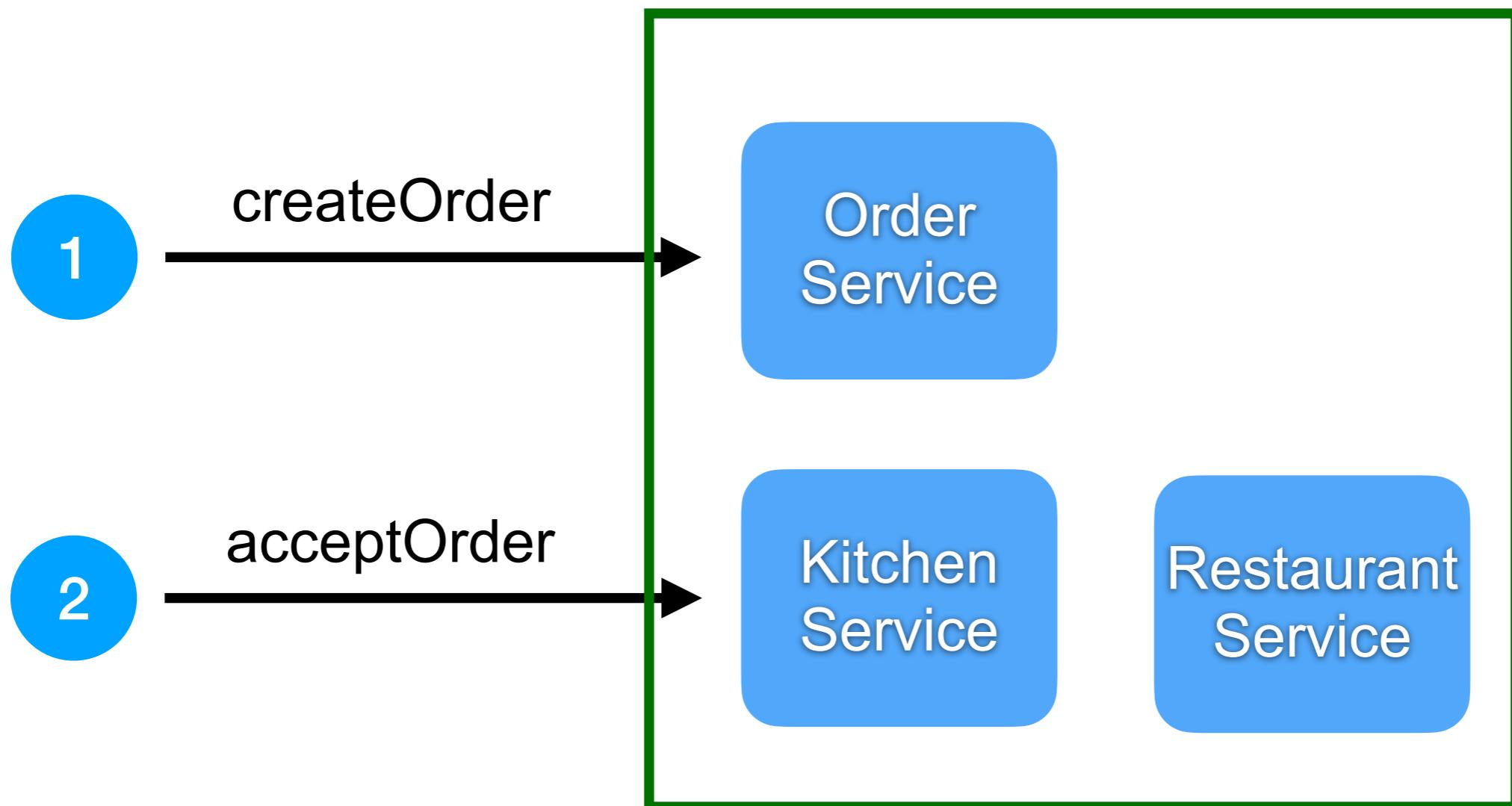
## 2. Identify services

Try to decomposition into services

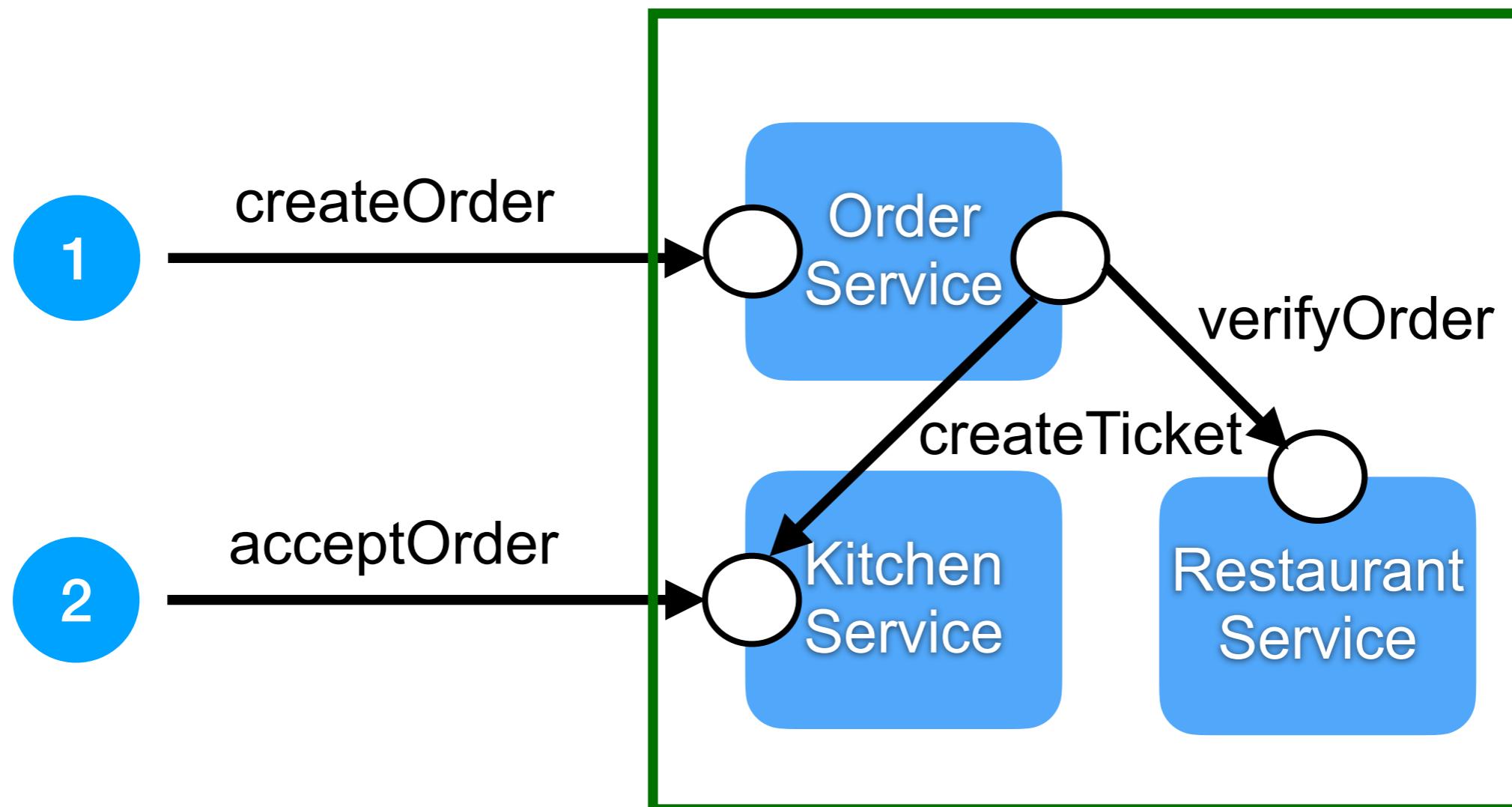
**Business > Technical concept**  
**What > How**



## 2. Identify services



# 3. Define service APIs and collaborations



# Problems when decompose services ?

Network latency

Reliability of communication (sync)

Maintain data consistency

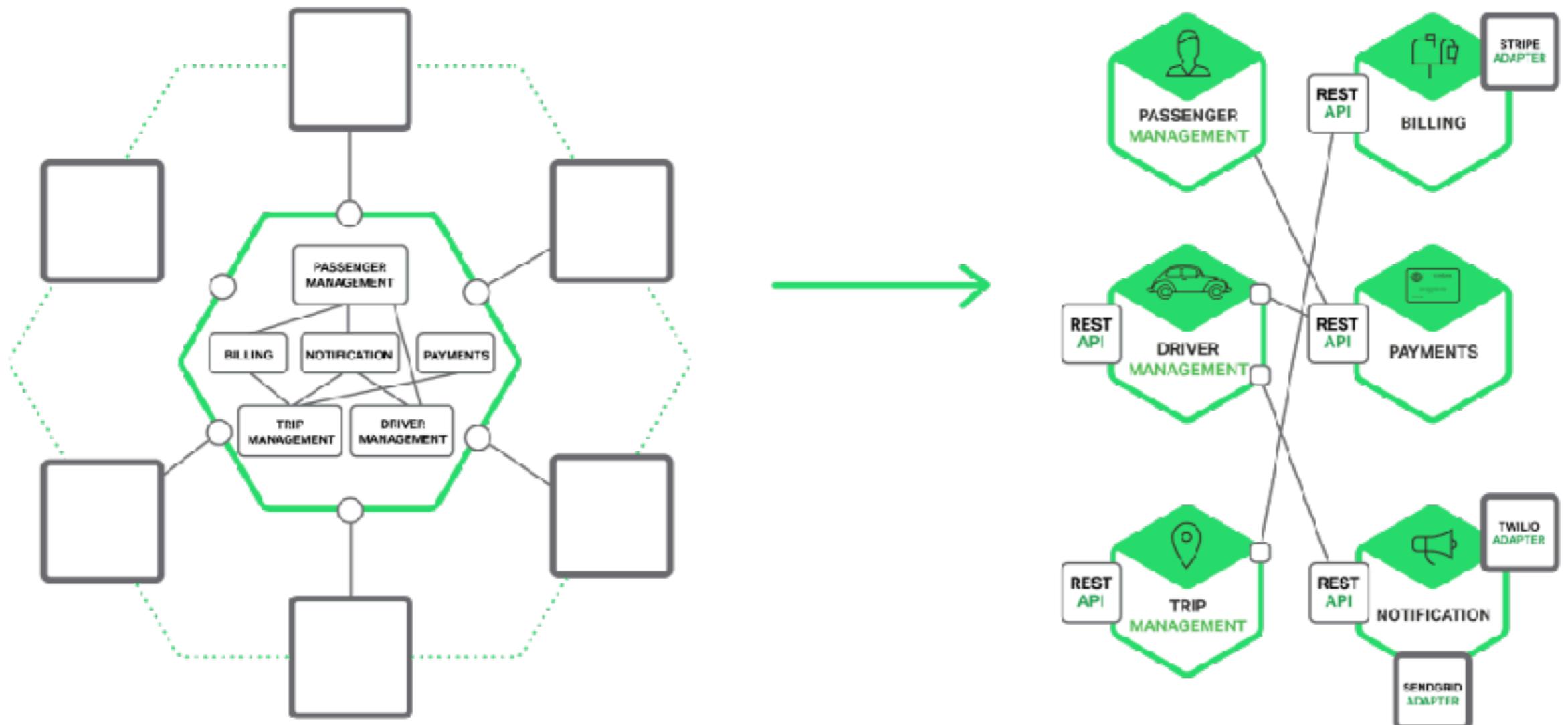
God classes



# Communication between services



# Inter Process Communication (IPC)



<https://www.nginx.com>



# Interaction Styles

	<b>One-to-one</b>	<b>One-to-many</b>
<b>Synchronous</b>	Request/response	-
<b>Asynchronous</b>	Async request/response	Publish/subscribe
	Notification	Publish/async response



# Synchronous communication



# Synchronous Communication

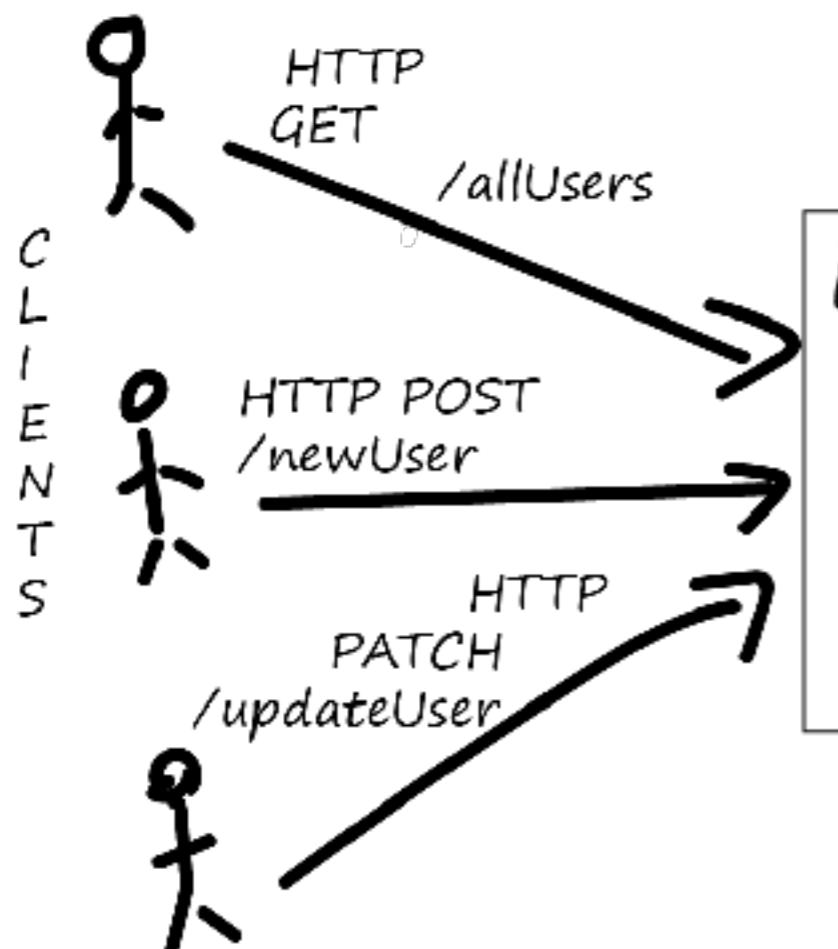
REST  
gRPC

Handling failure with Circuit breaker  
Service discovery



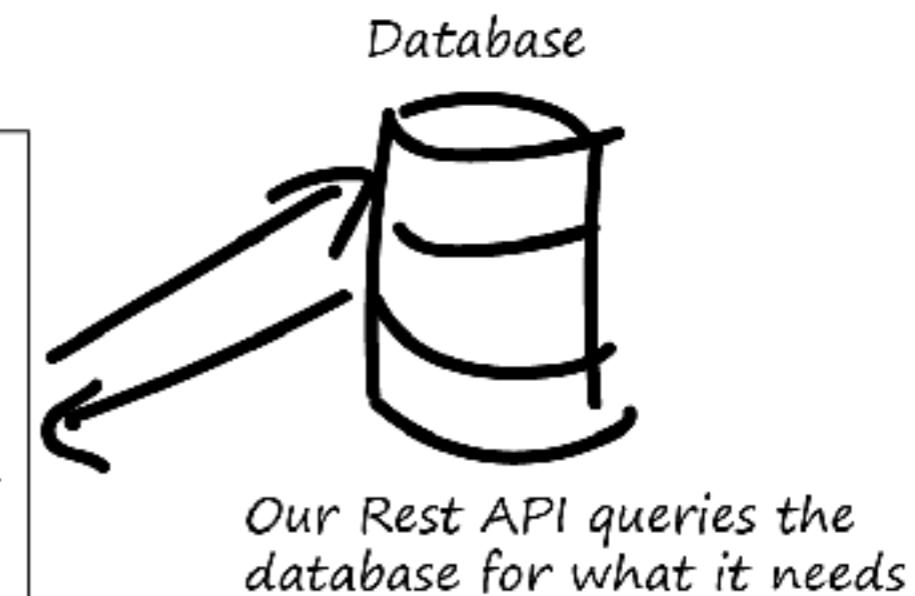
# REpresentational State Transfer

## Rest API Basics



Our Clients, send HTTP Requests and wait for responses

Typical HTTP Verbs:  
GET → Read from Database  
PUT → Update/Replace row in Database  
PATCH → Update/Modify row in Database  
POST → Create a new record in the database  
DELETE → Delete from the database



Response: When the Rest API has what it needs, it sends back a response to the clients. This would typically be in JSON or XML format.



# REST :: mapping with HTTP verbs

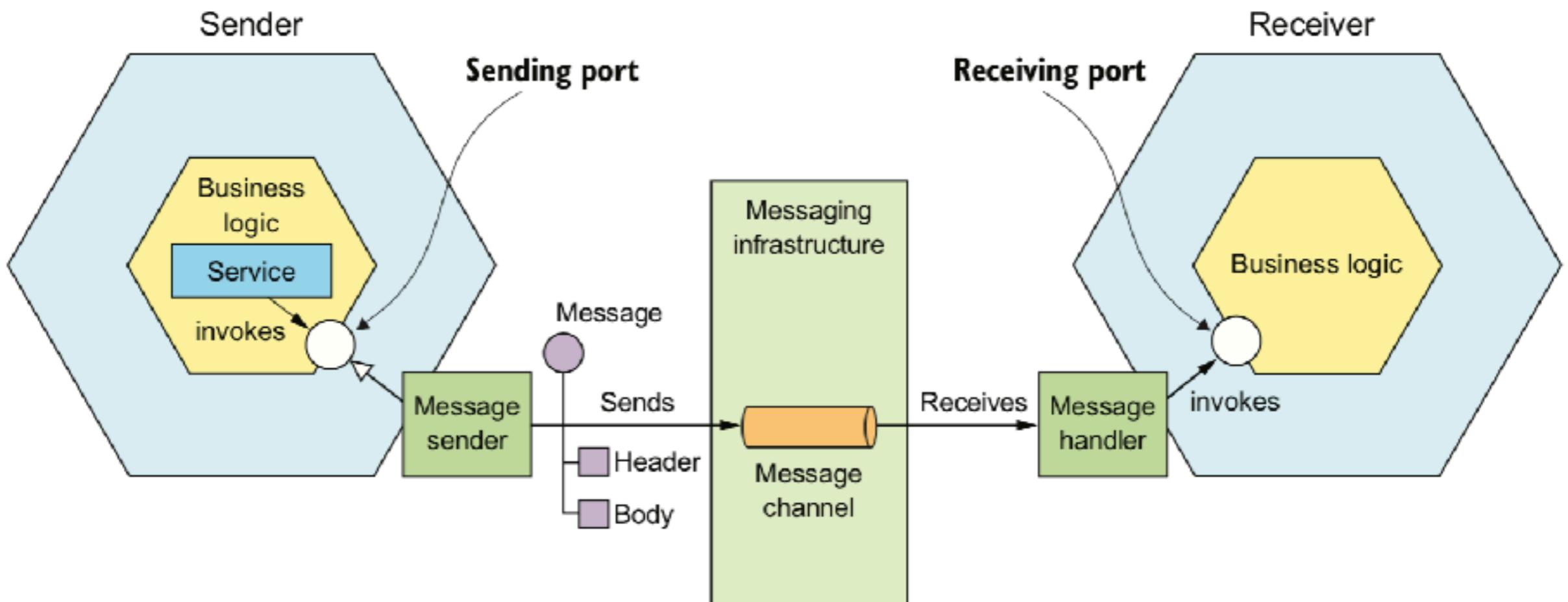
Activity	HTTP Method
Retrieve data	GET
Create new data	POST
Update data	PUT
Delete data	DELETE



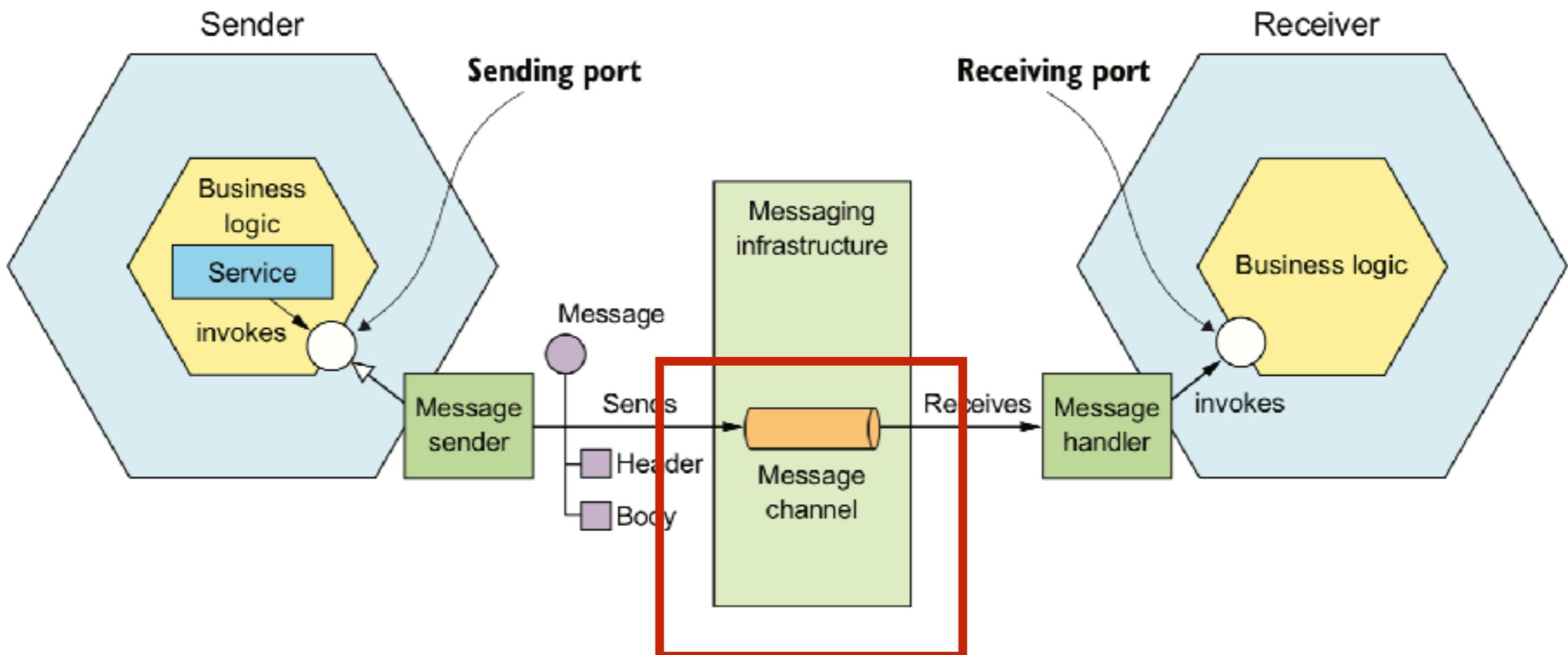
# Asynchronous communication



# Asynchronous Communication

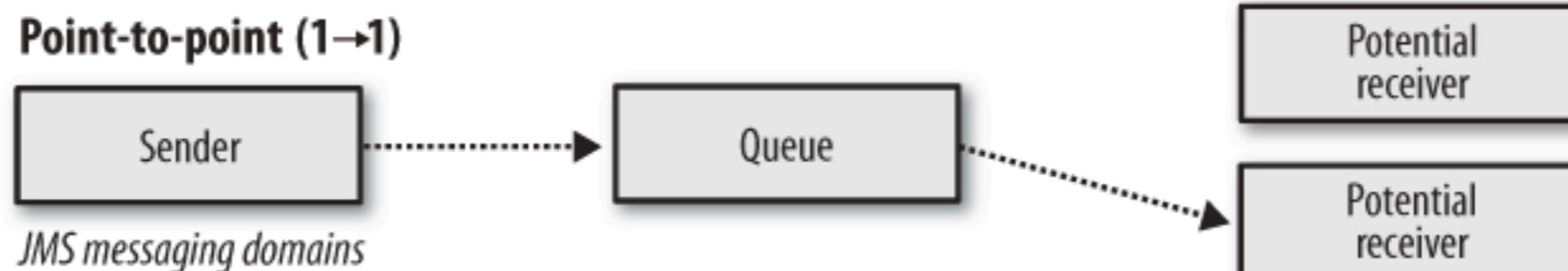
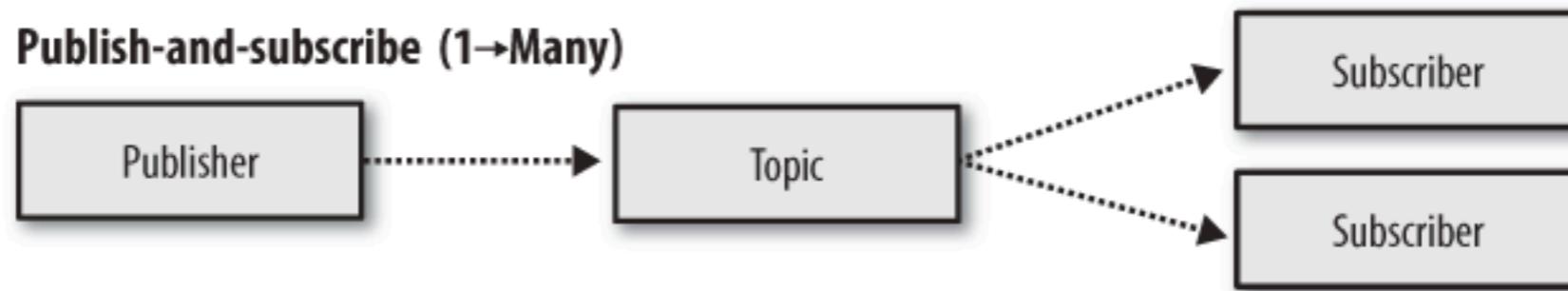


# Asynchronous messaging pattern



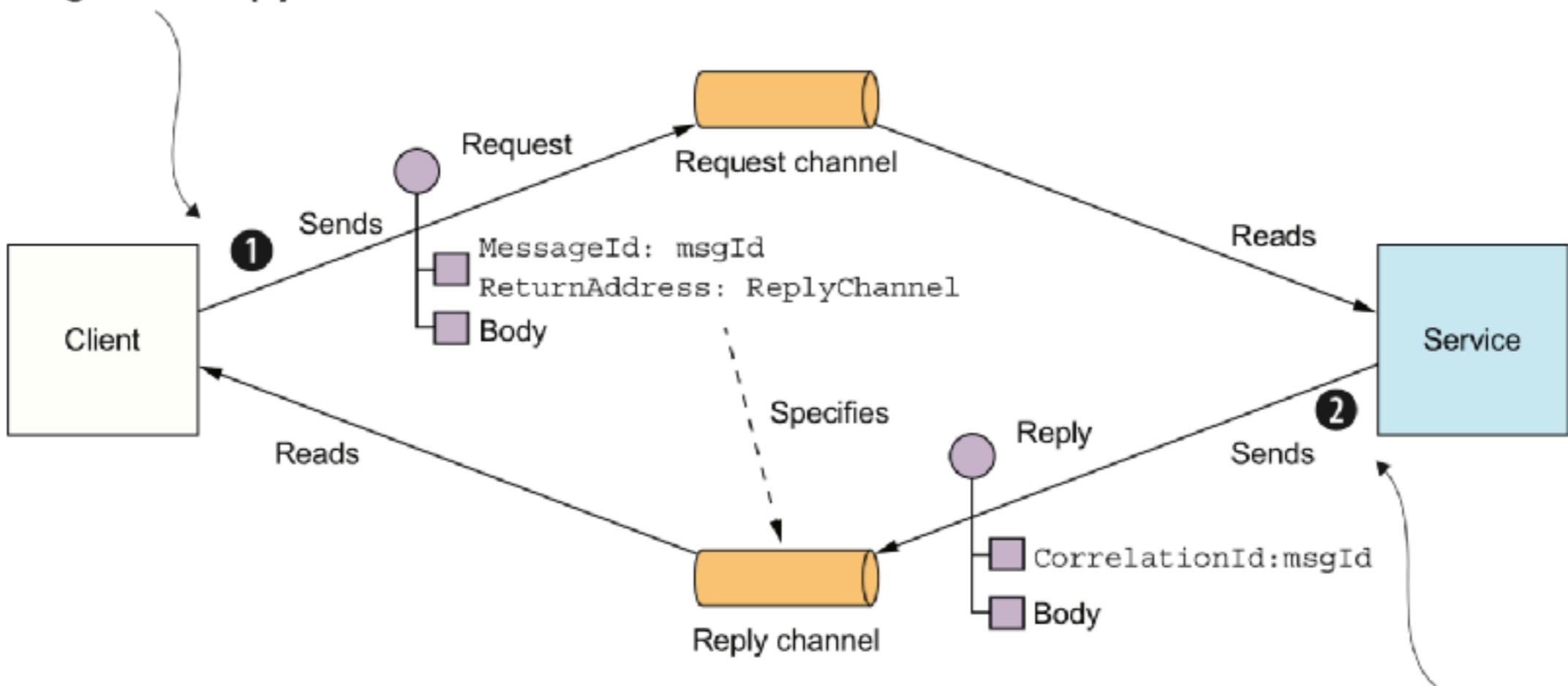
# Messaging channels

## Point-to-point Publish-subscribe



# Async request/response (1)

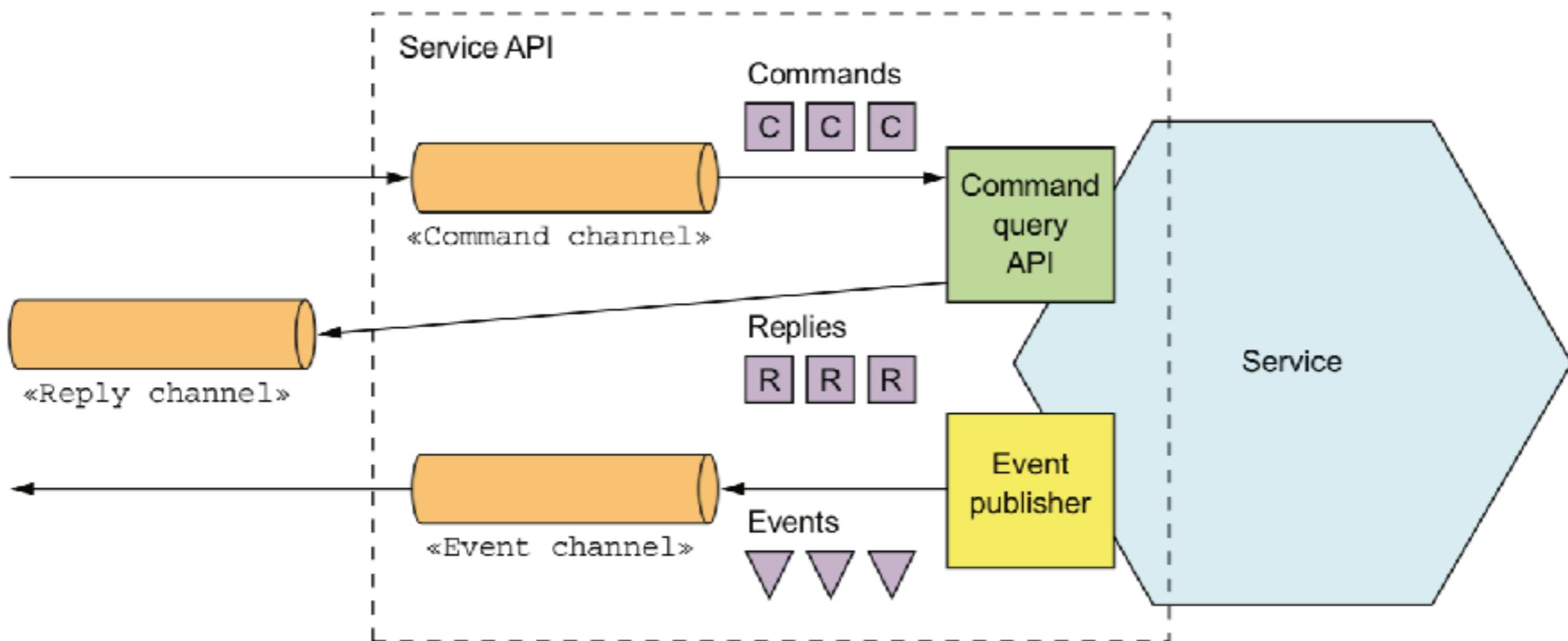
**Client sends message containing msgId and a reply channel.**



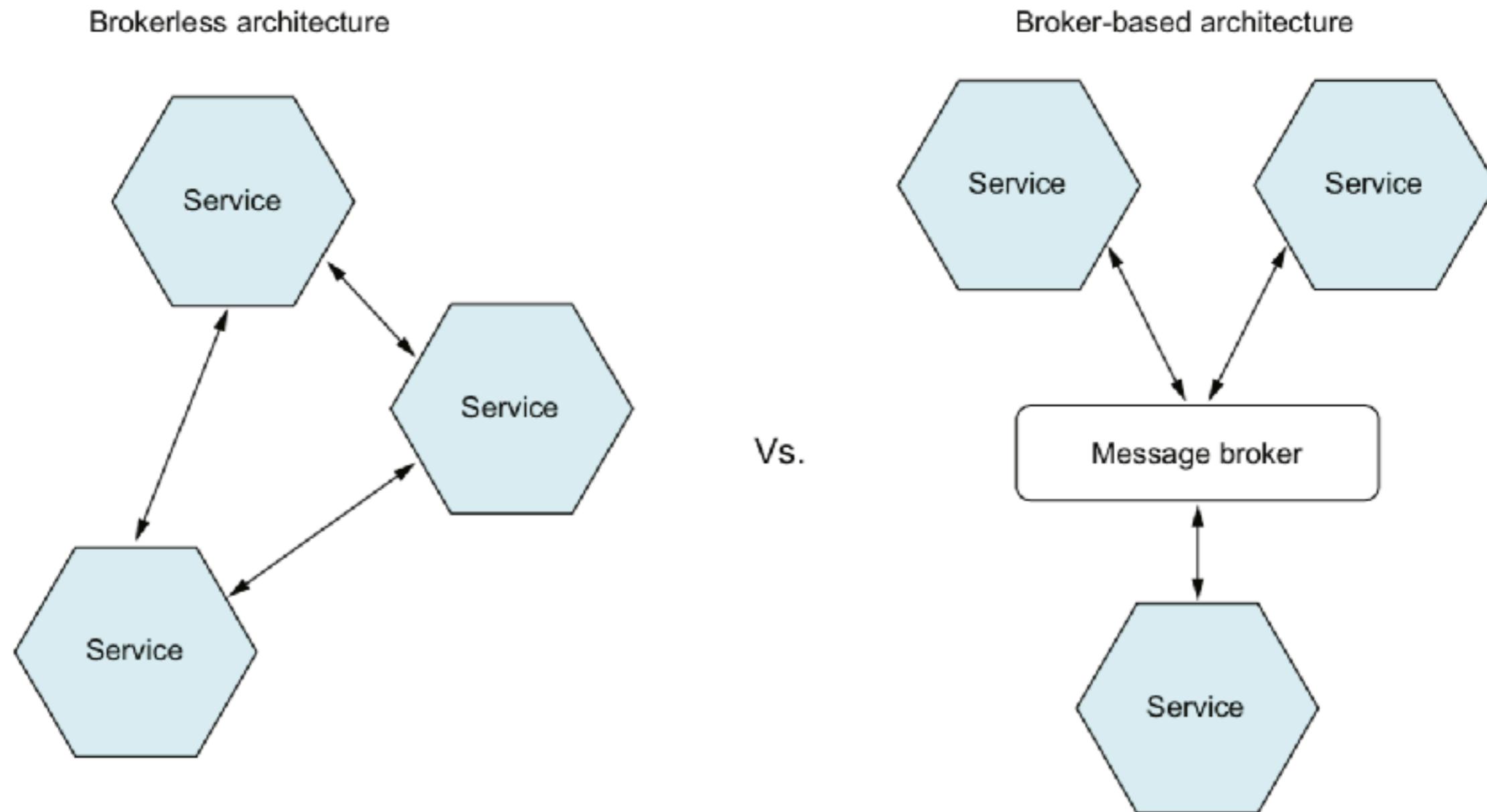
**Service sends reply to the specified reply channel. The reply contains a correlationId, which is the request's msgId.**



# Async request/response (2)



# Messaging-based use message broker



# Message brokers



# Benefits

Loose-coupling  
Message buffer  
Flexible communication



# Drawbacks

Performance bottleneck  
Single point of failure  
Operational complexity

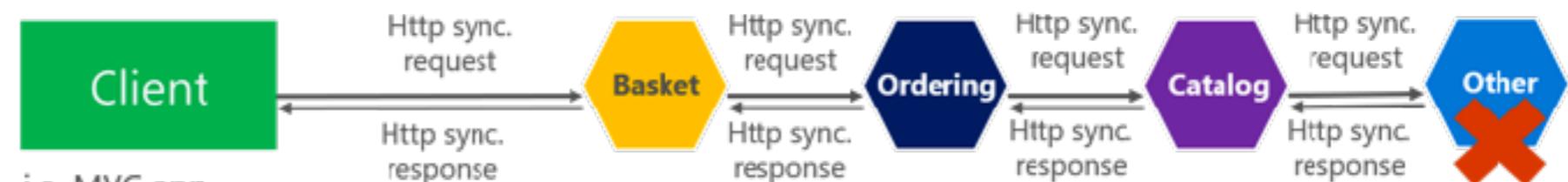


# Communication !!

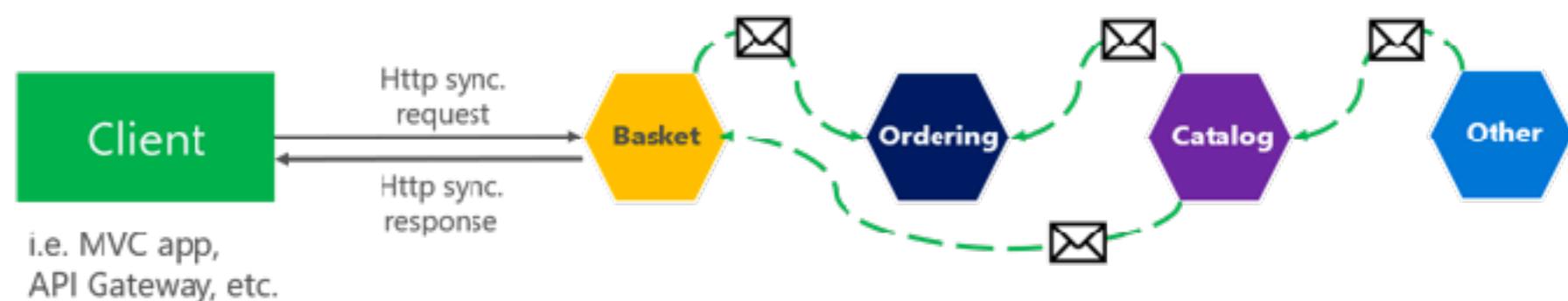
Synchronous vs. async communication across microservices

## Anti-pattern

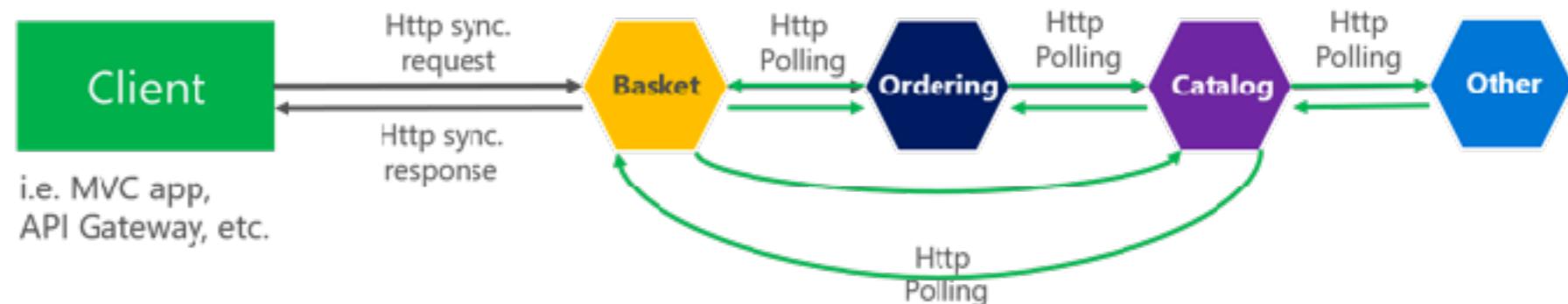
**Synchronous**  
all req./resp. cycle



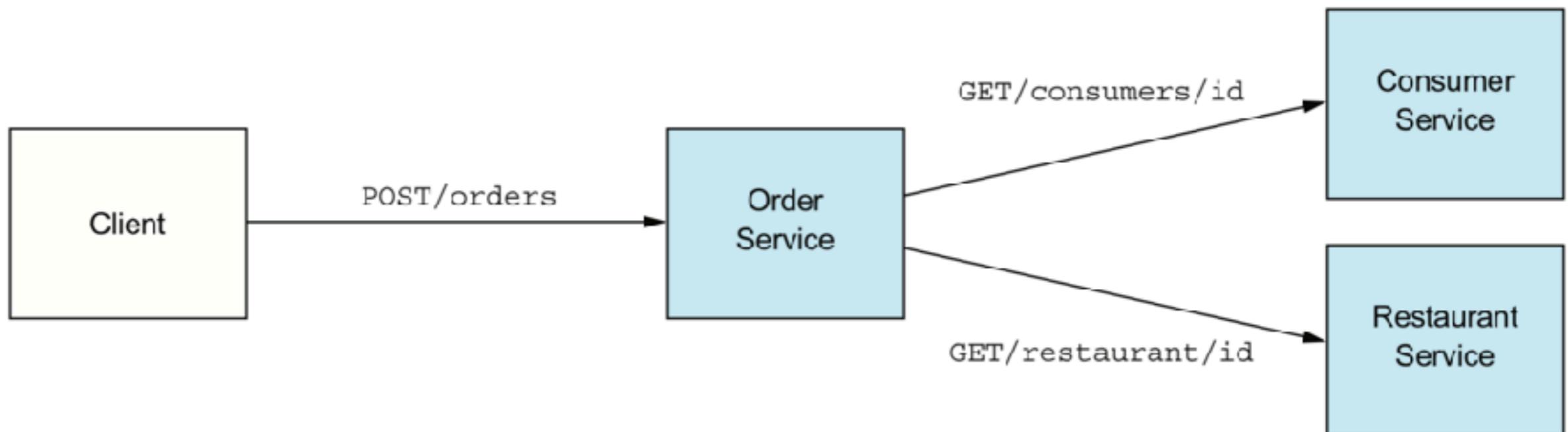
**Asynchronous**  
Comm. across  
internal microservices  
(EventBus: i.e. **AMQP**)



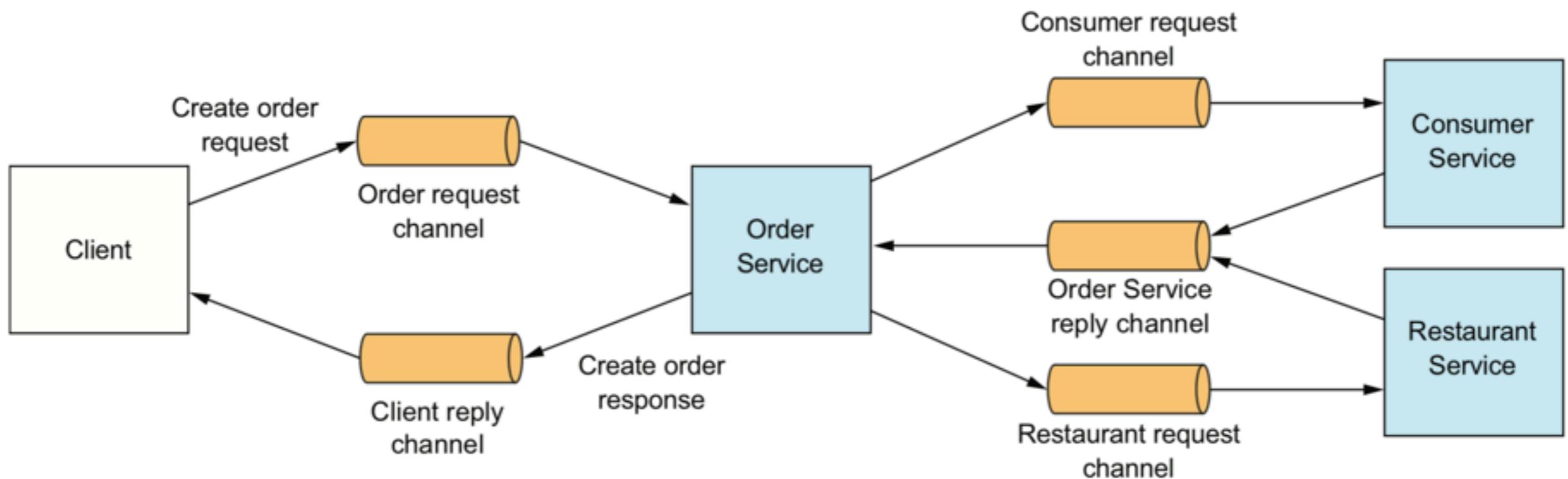
**"Asynchronous"**  
Comm. across  
internal microservices  
(Polling: **Http**)



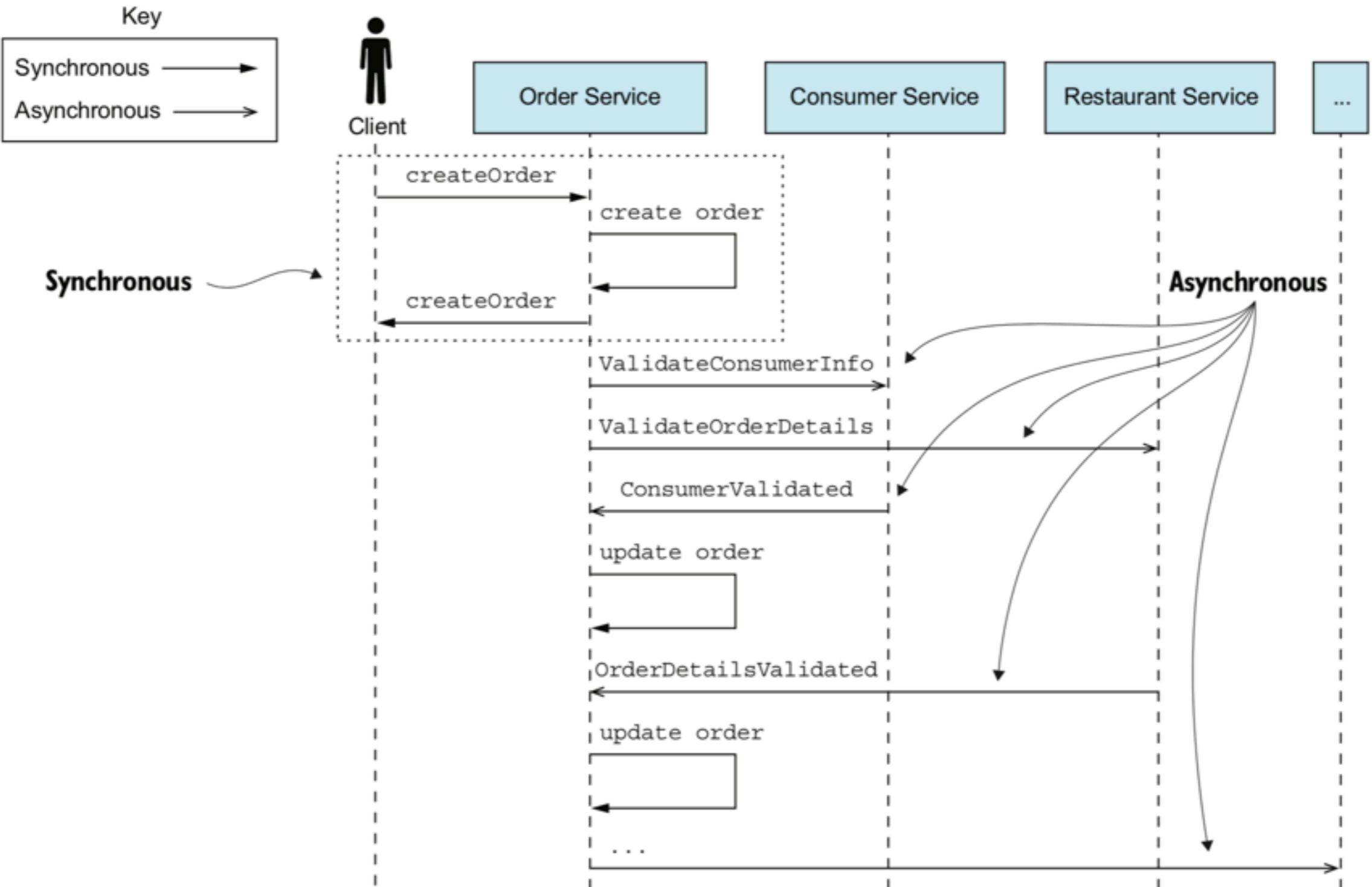
# Synchronous reduce availability



# Replace with asynchronous (1)



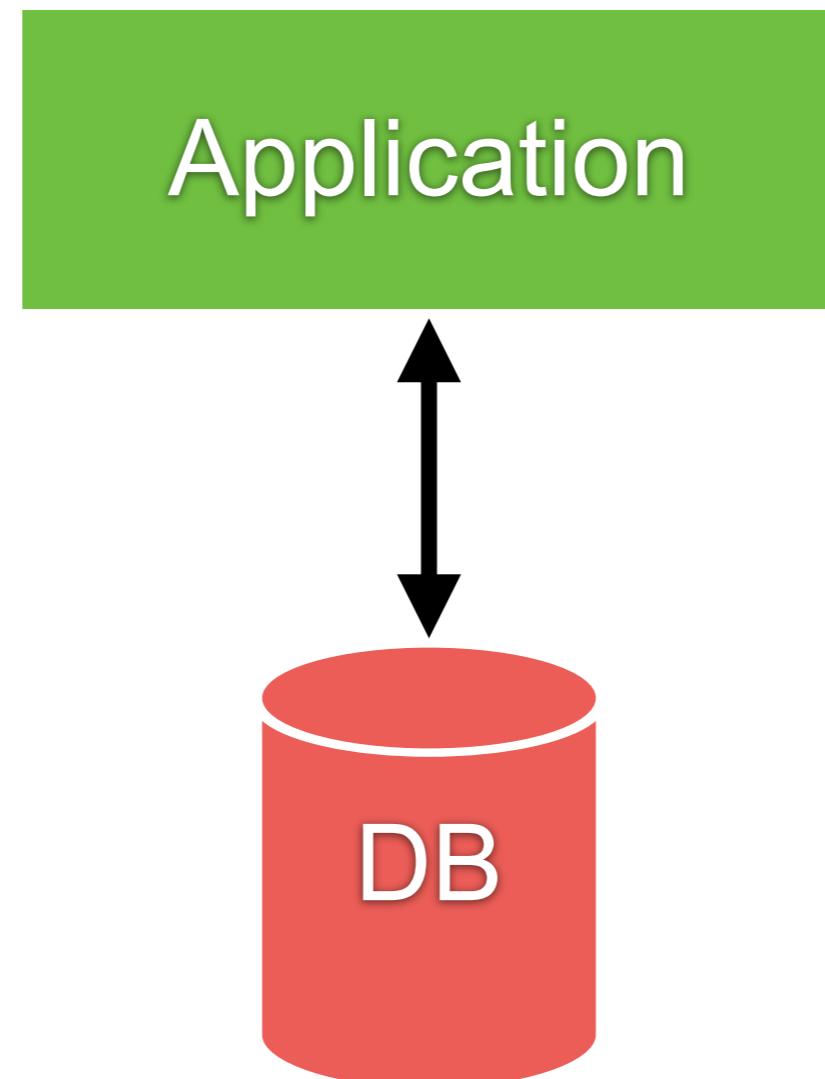
# Replace with asynchronous (2)



# Manage data consistency

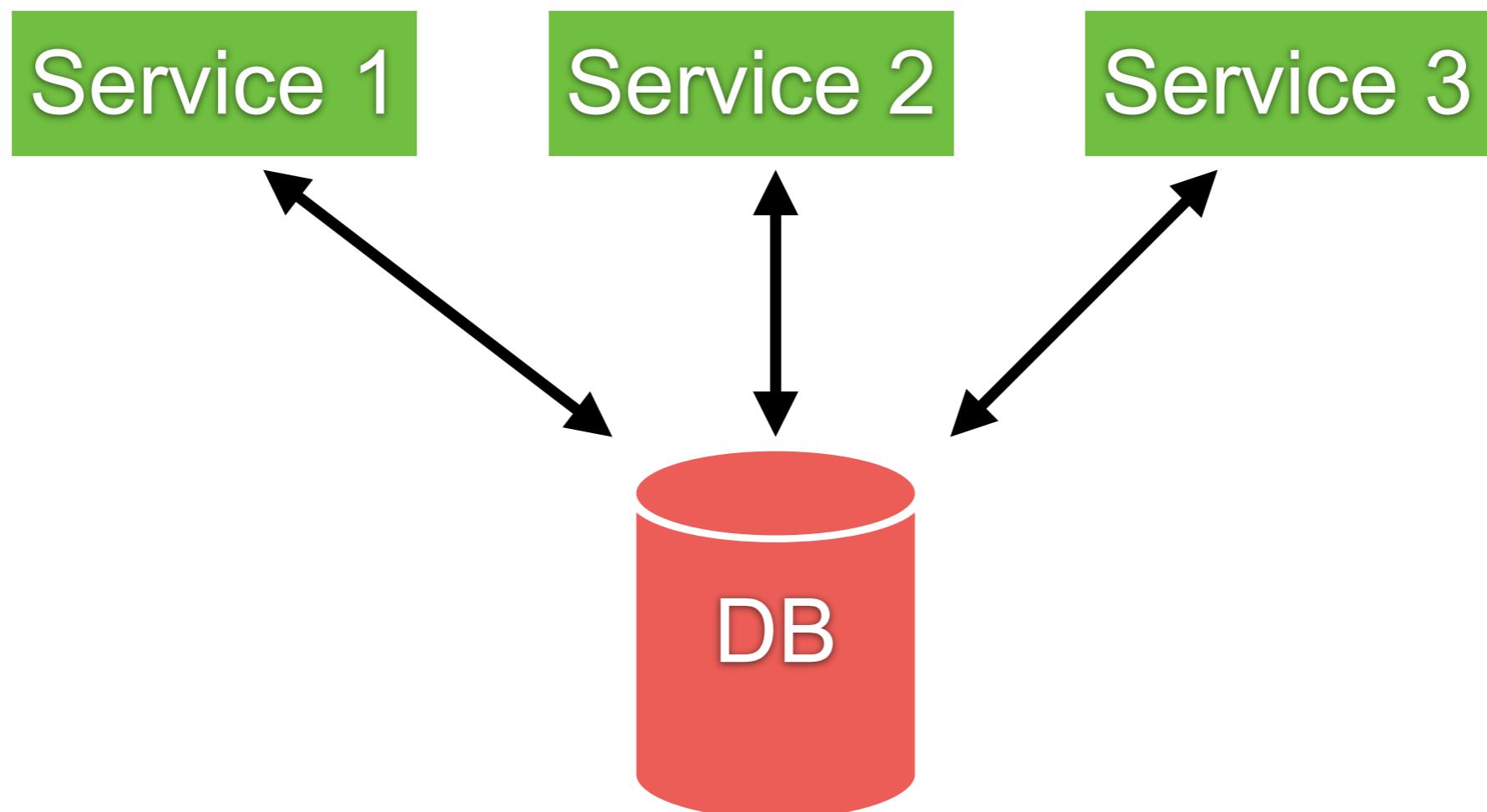


# Monolithic



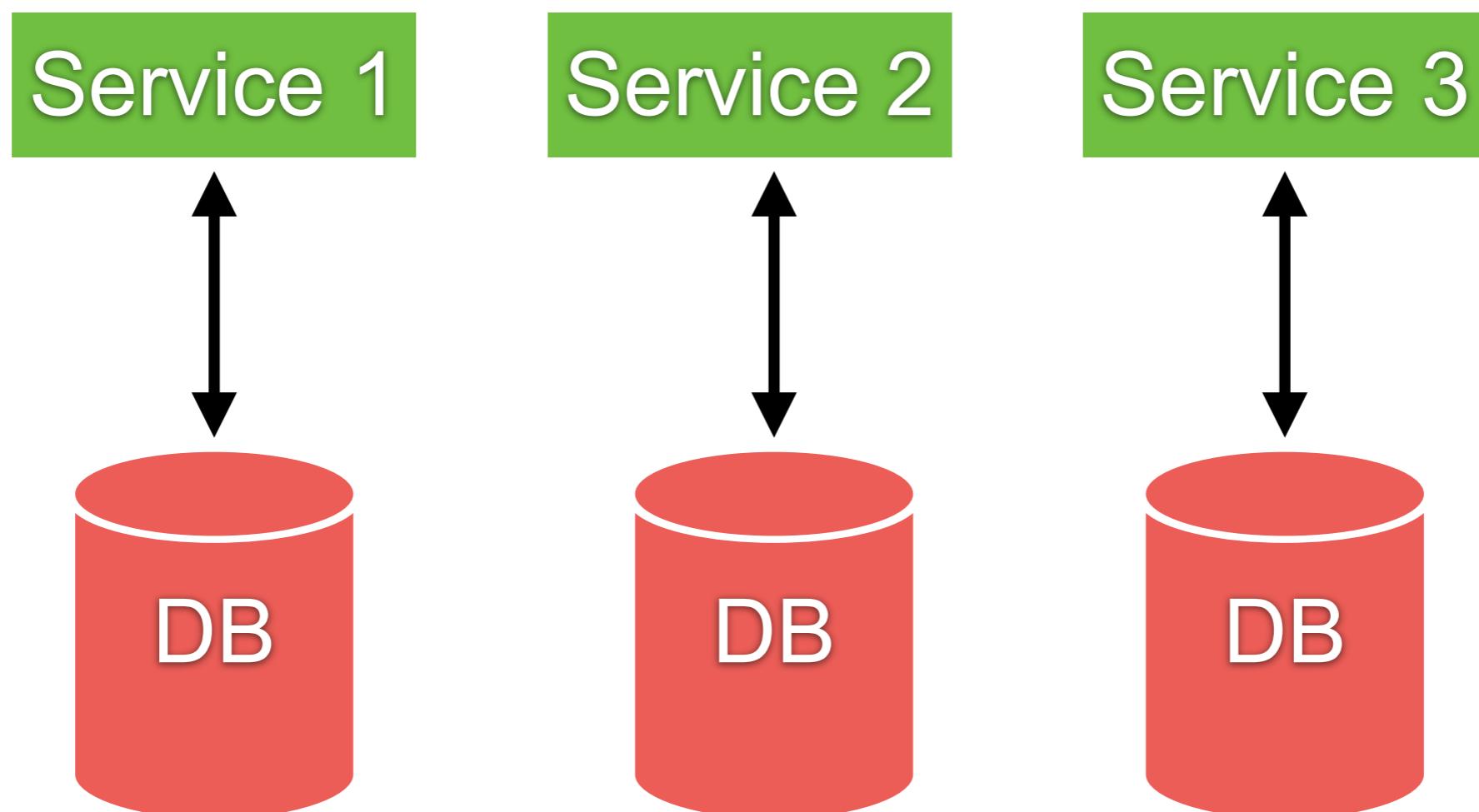
# Microservices

Shared database

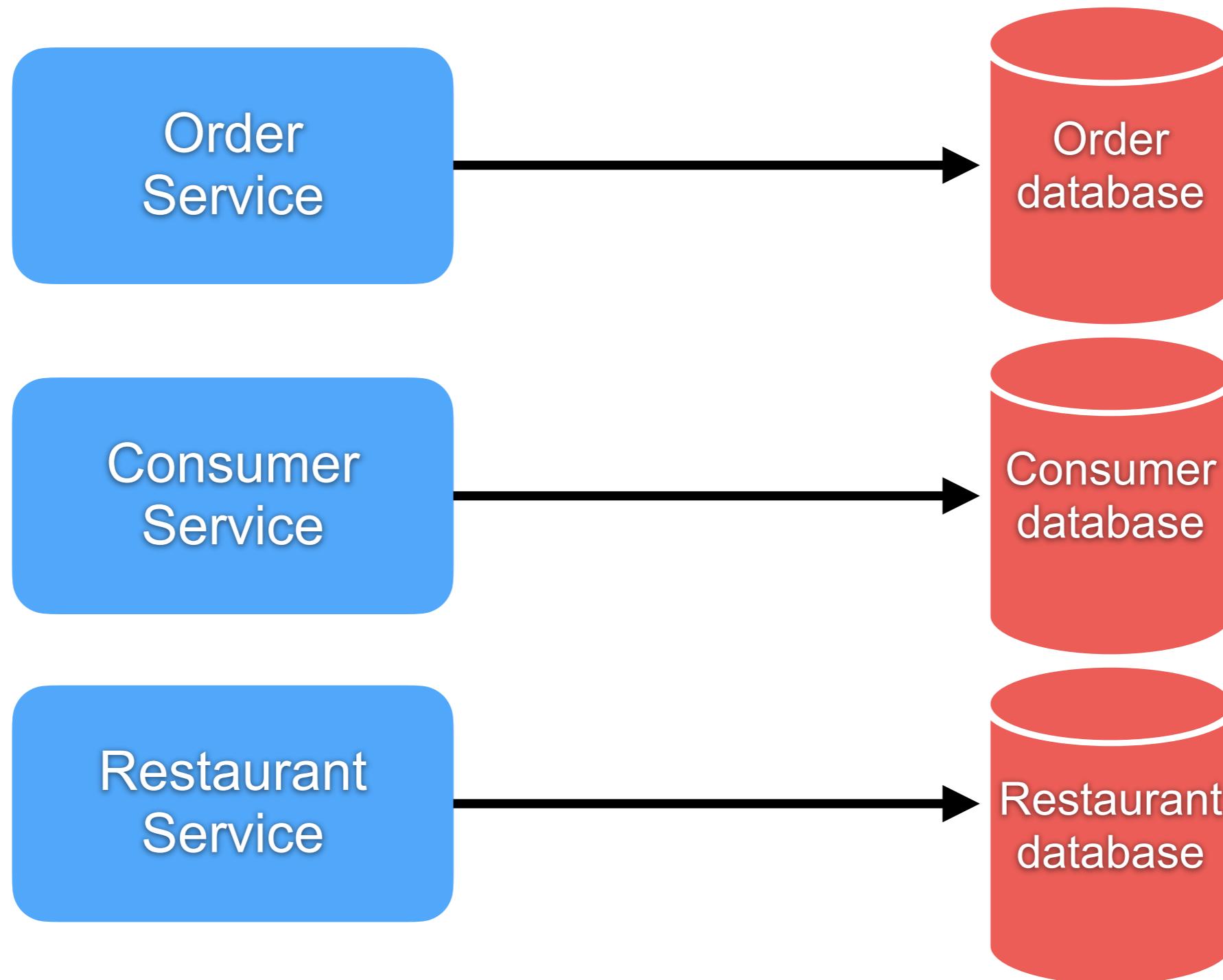


# Microservices

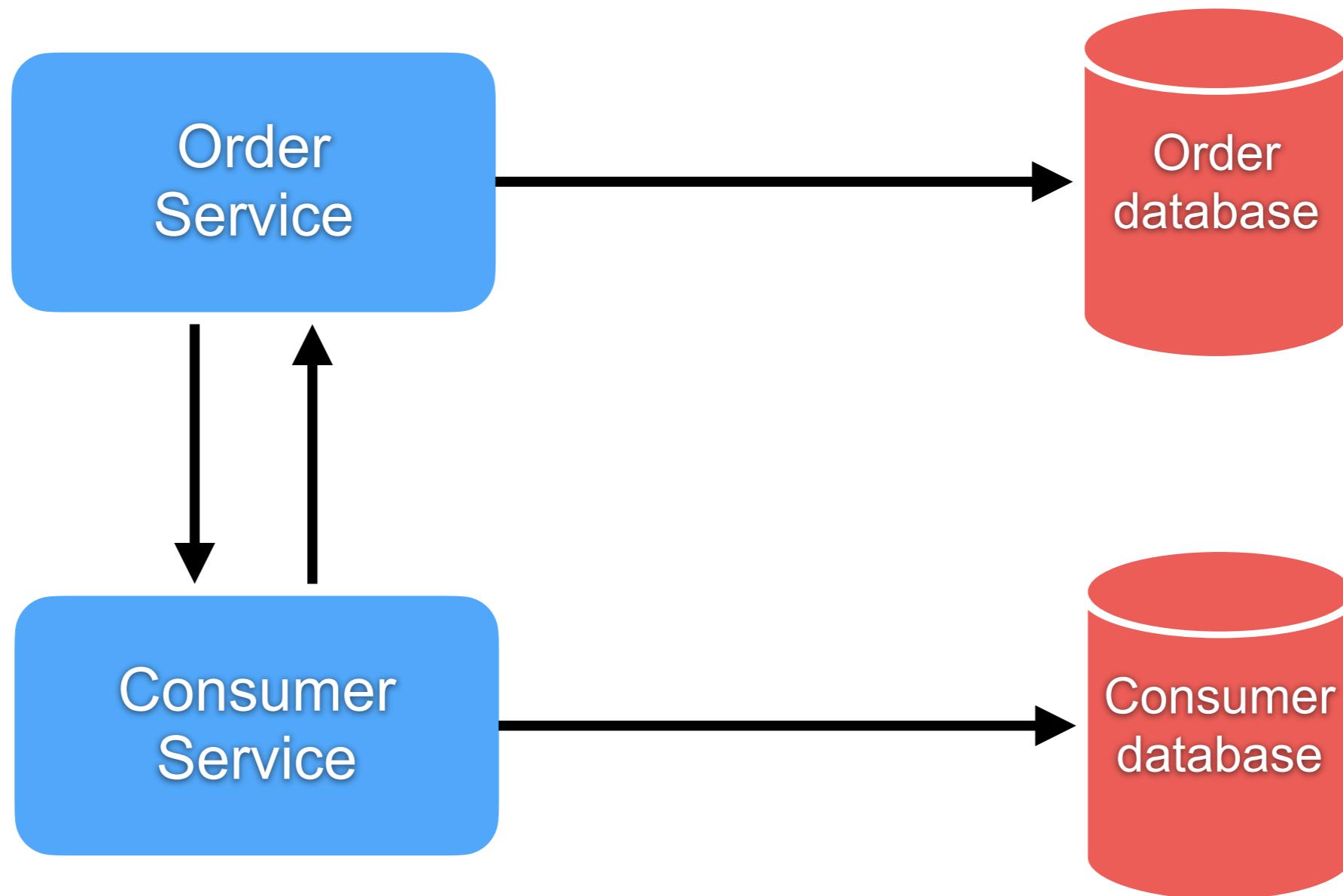
Database per service

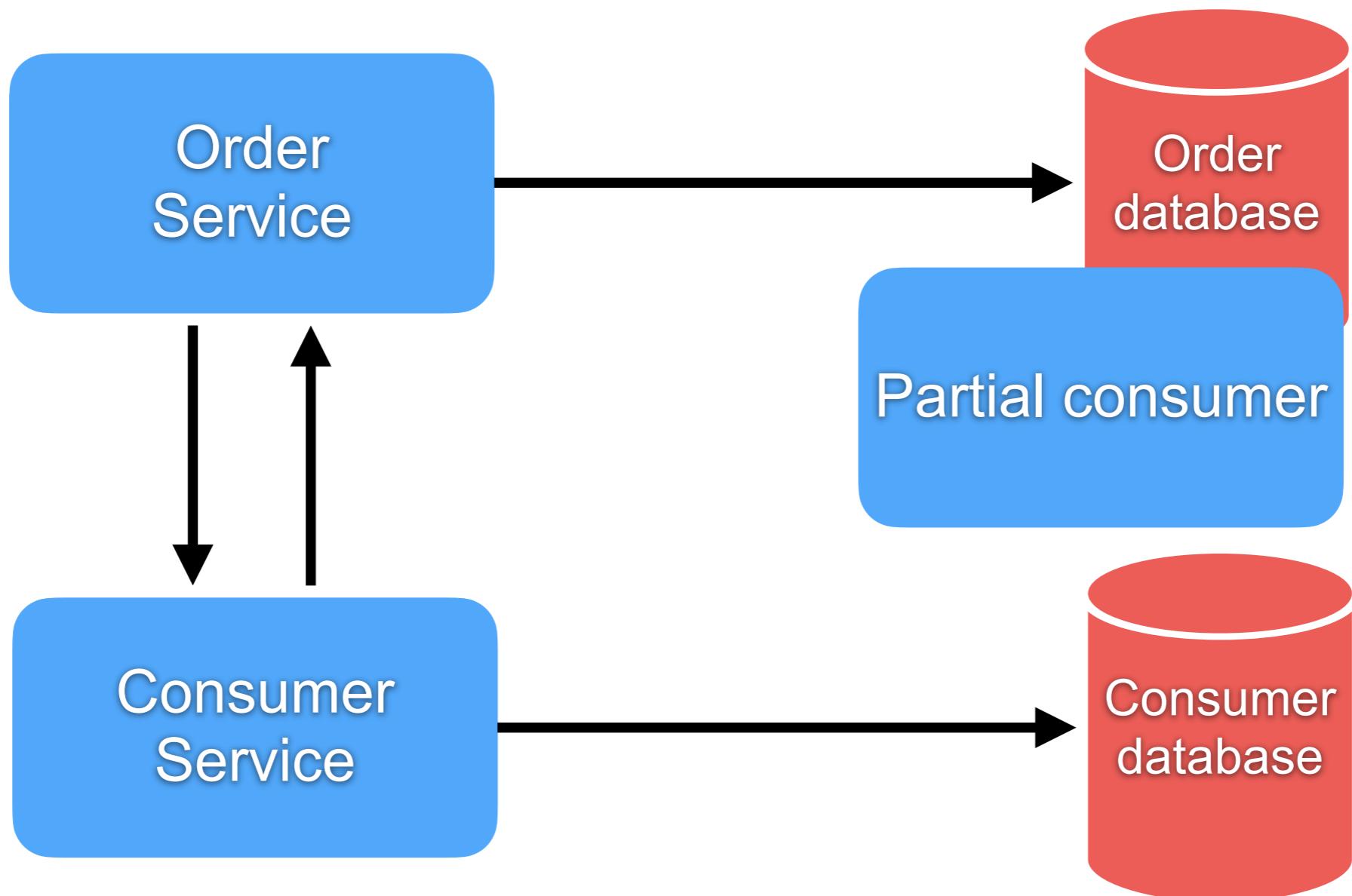


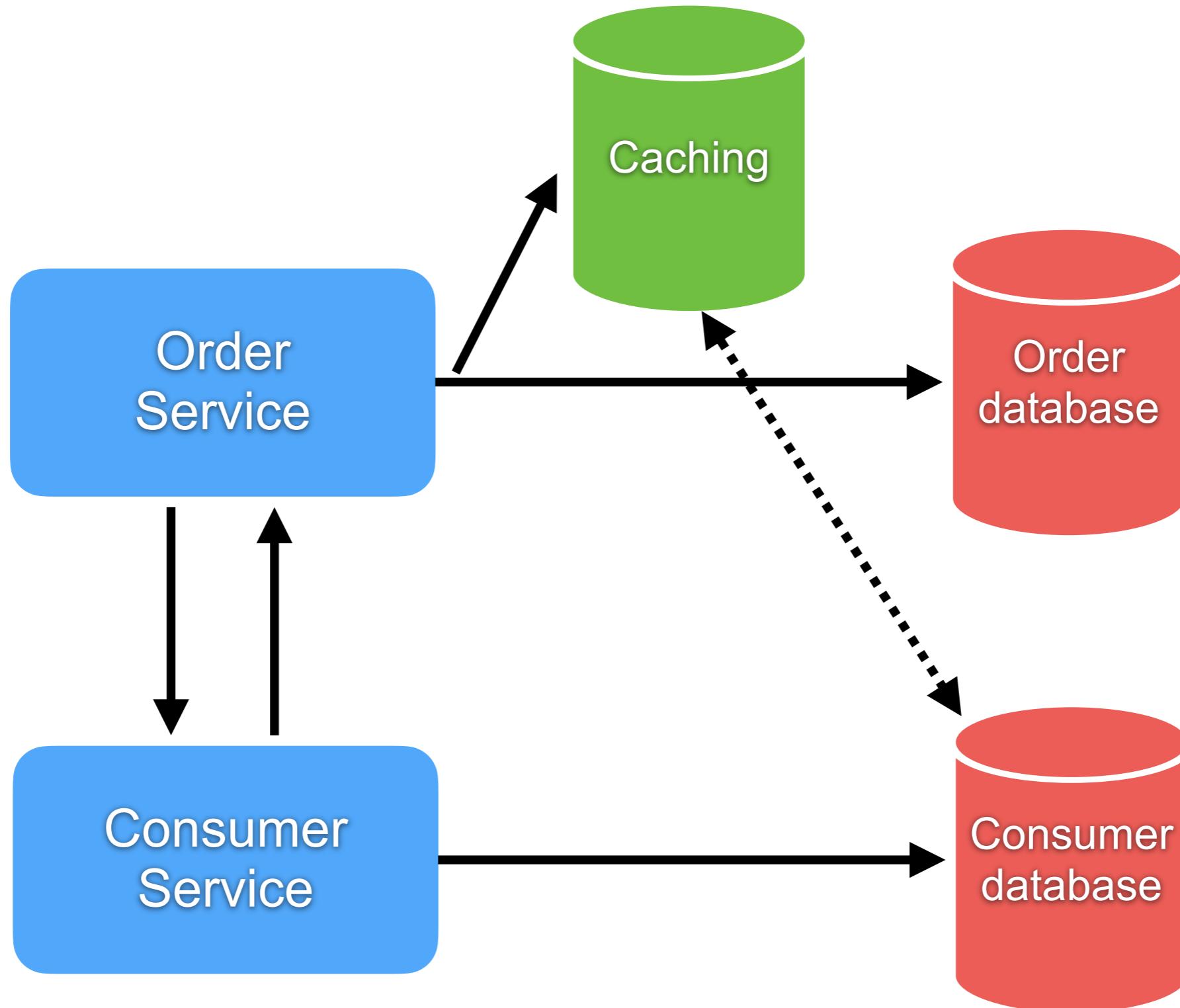
# Database per service



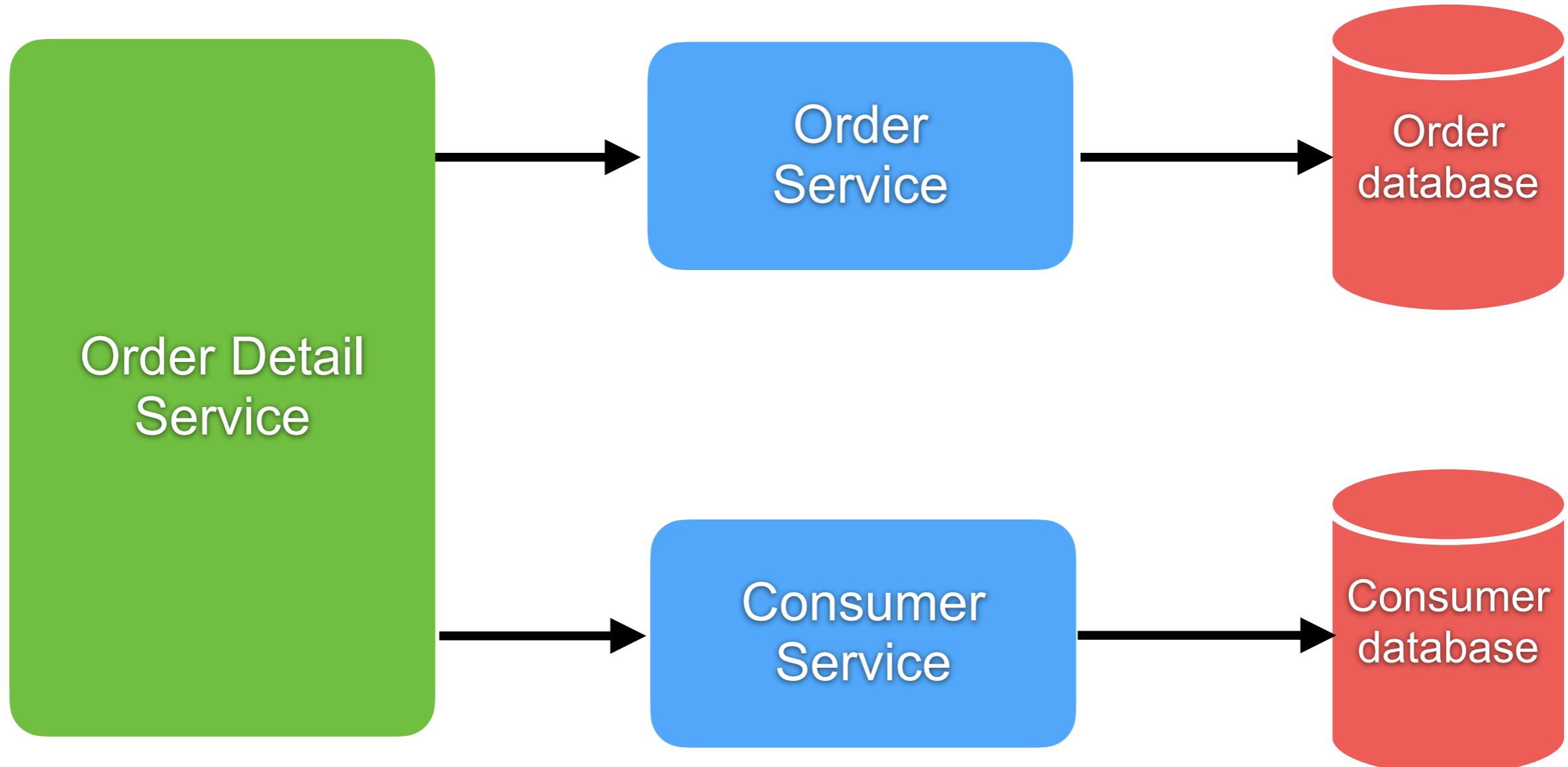
# Loose coupling = encapsulation data



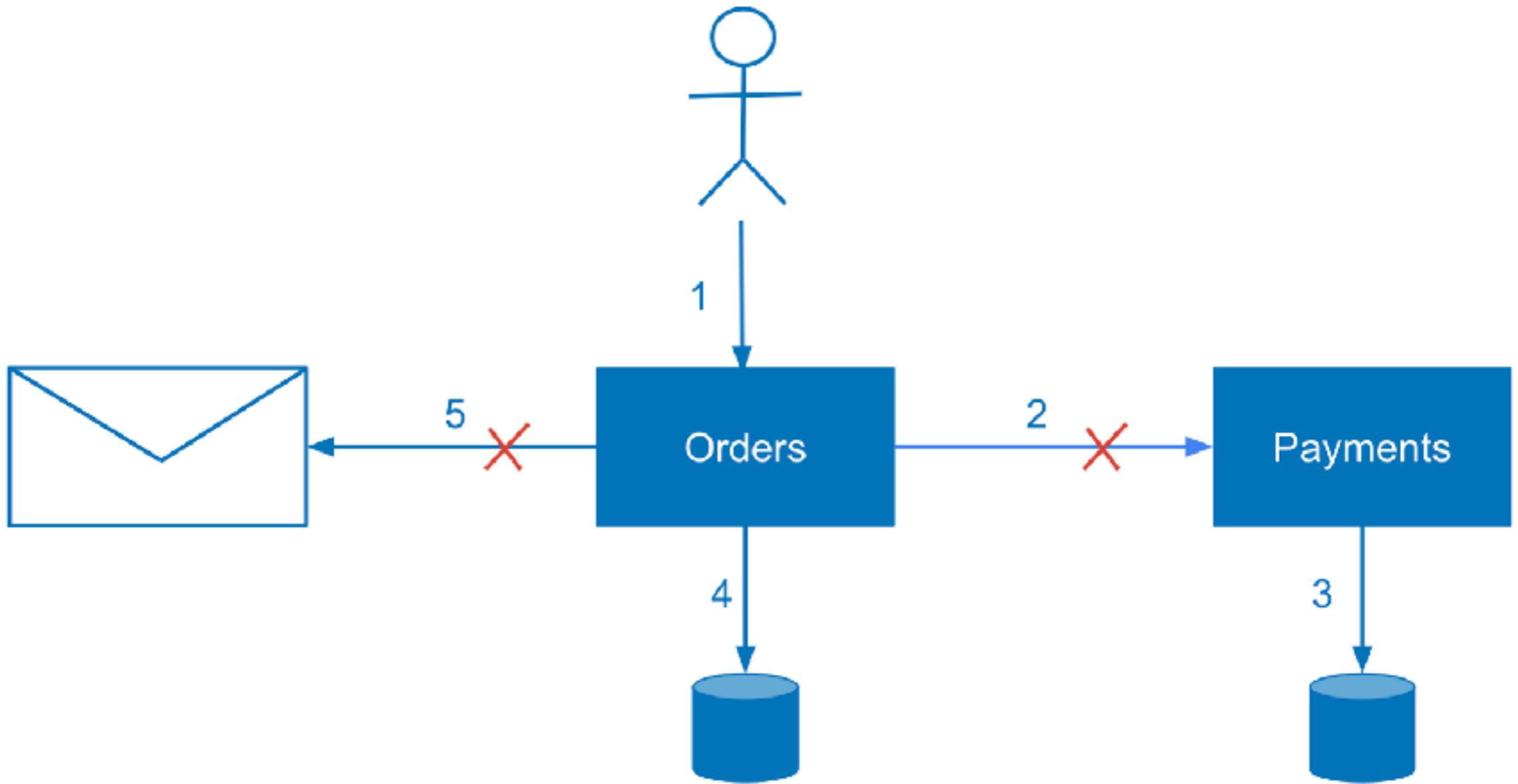




# Order detail orchestration service

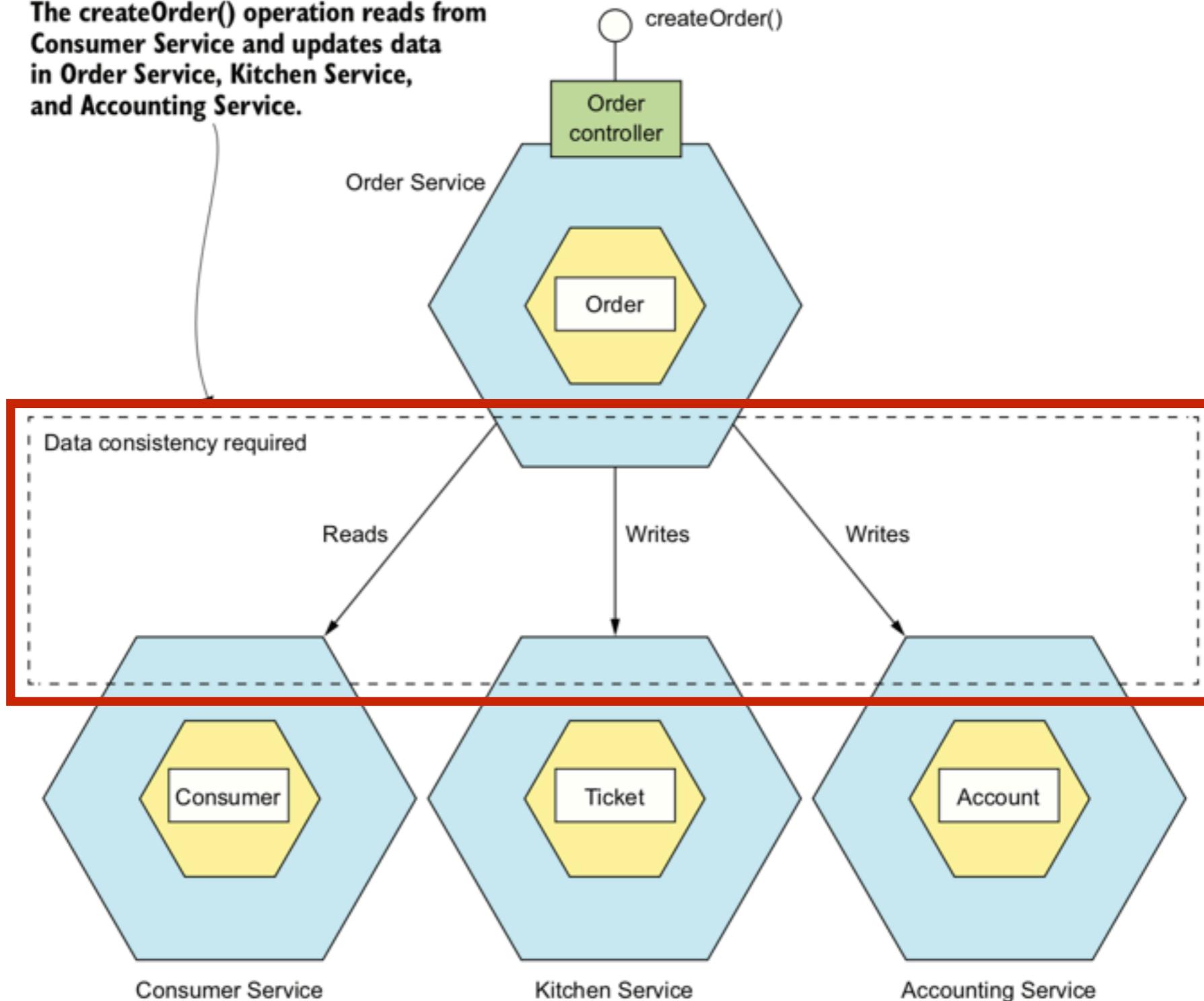


# Distributed process failure !!



# Problem ?

The **createOrder()** operation reads from Consumer Service and updates data in Order Service, Kitchen Service, and Accounting Service.



# Can't use **ACID** transaction !!



A - Atomicity

All or Nothing Transactions

C - Consistency

Guarantees Committed Transaction State

I - Isolation

Transactions are Independent

D – Durability

Committed Data is Never Lost

(c) <http://blog.sqlauthority.com>

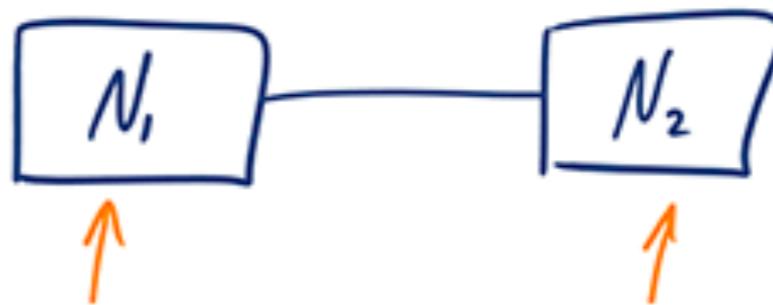


# CAP Theorem

**Consistency**



**Availability**



**Partition Tolerance**



<http://robertgreiner.com/2014/08/cap-theorem-revisited/>



# Database per service

High complexity

Query data across multiple databases

Challenge “How to join data ?”

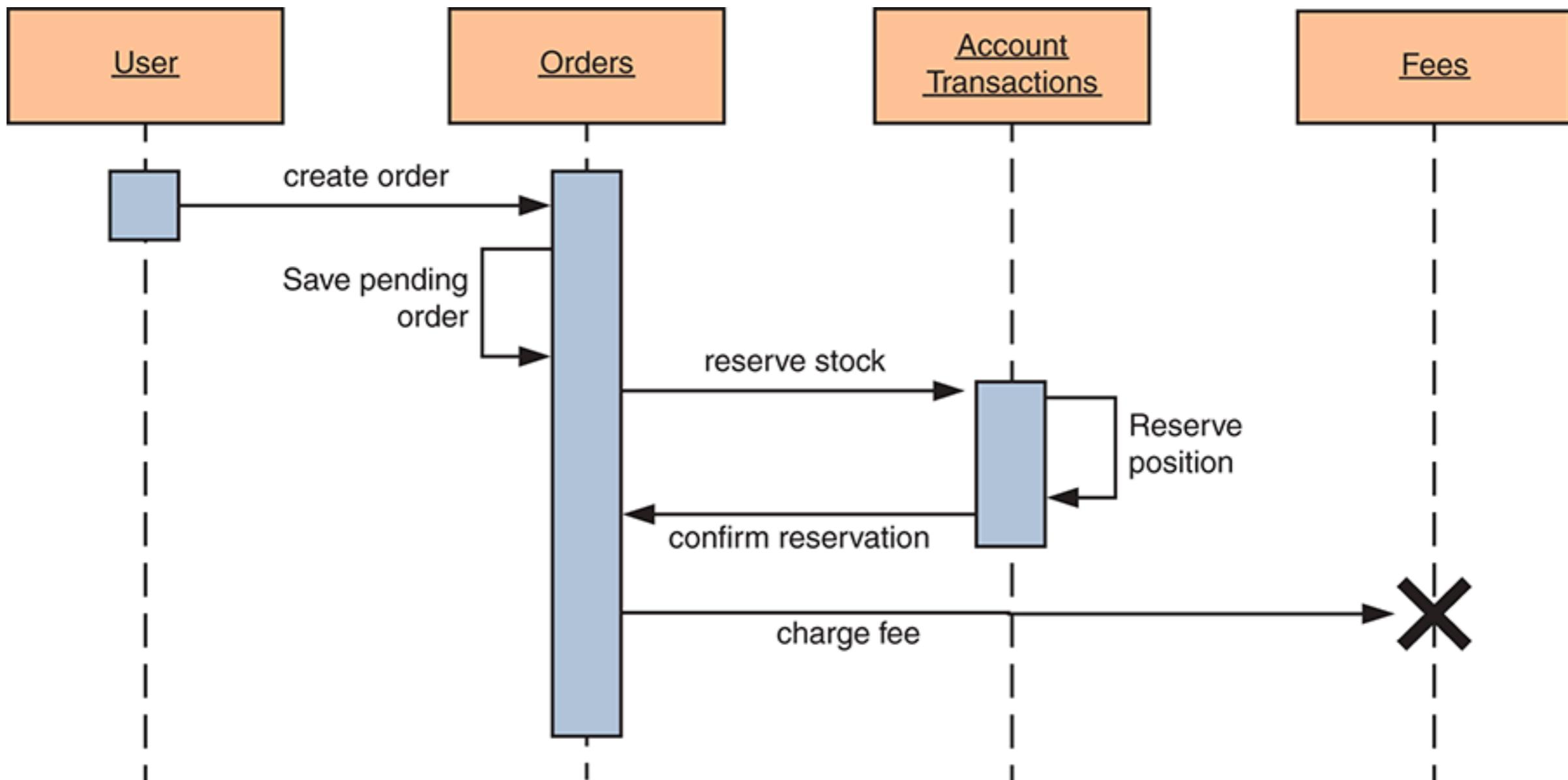
Maintain database consistency



# **How to maintain data consistency across services ?**



# Problem



# Distributed transaction

Common approach is Two-phase commit(2PC)

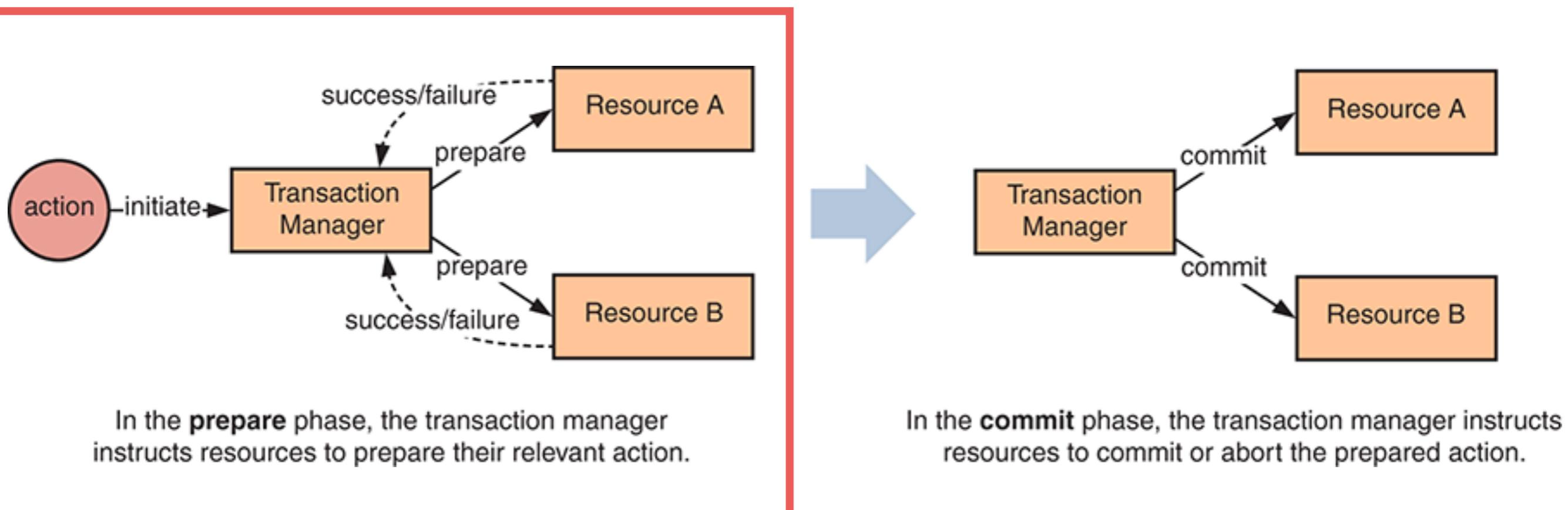
Use transaction manager to split operations across multiple resources in 2 phases

1. *Prepare*
2. *Commit*



# Two-phase commit

## Phase 1: prepare

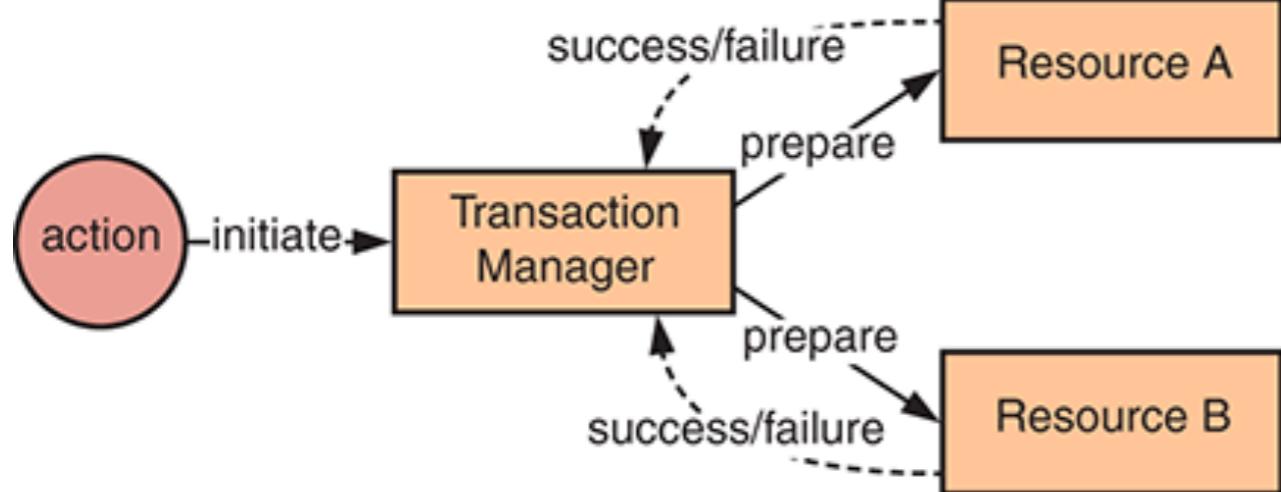


[https://en.wikipedia.org/wiki/Two-phase\\_commit\\_protocol](https://en.wikipedia.org/wiki/Two-phase_commit_protocol)

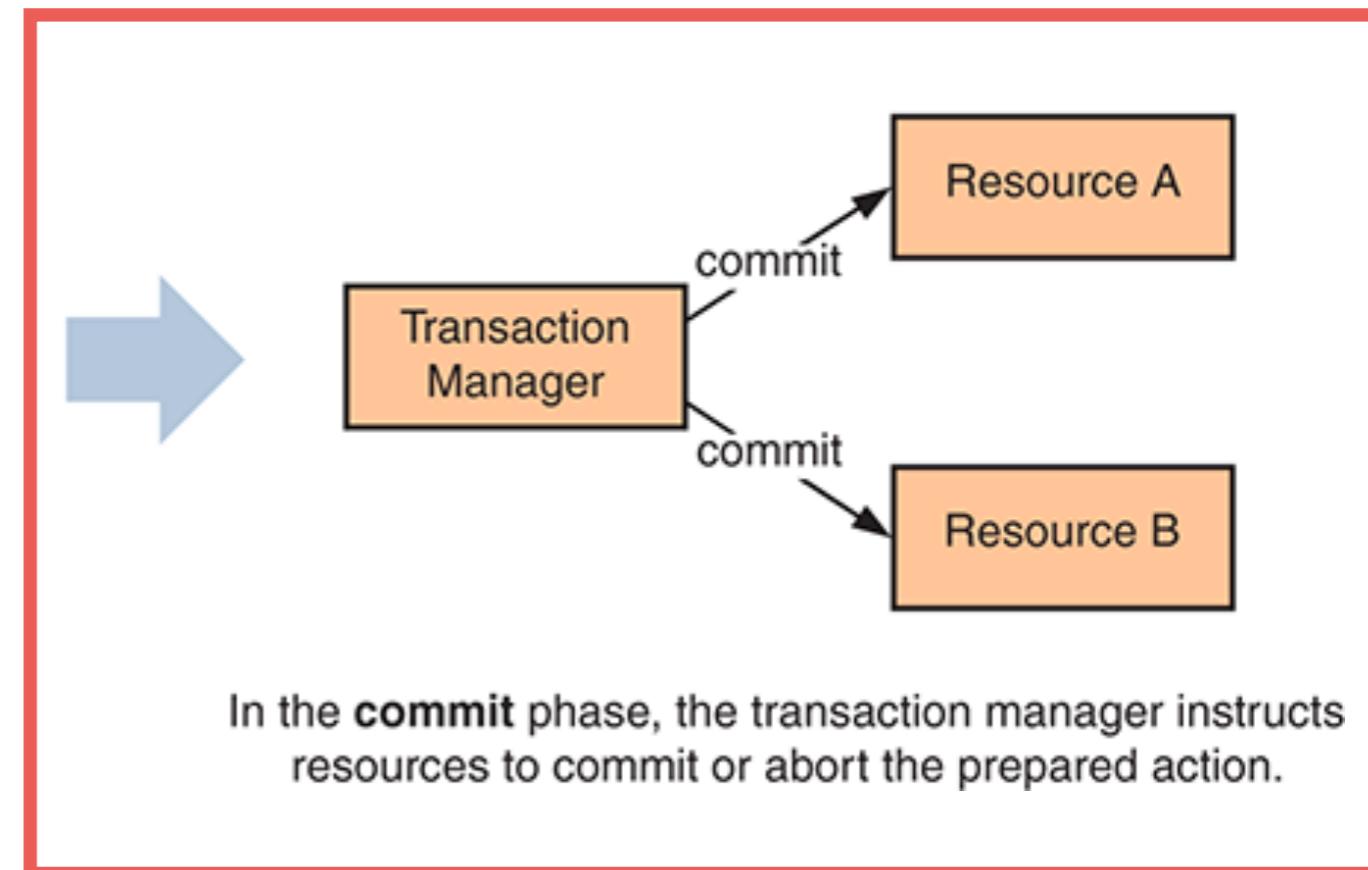


# Two-phase commit

## Phase 2: commit



In the **prepare** phase, the transaction manager instructs resources to prepare their relevant action.



In the **commit** phase, the transaction manager instructs resources to commit or abort the prepared action.

[https://en.wikipedia.org/wiki/Two-phase\\_commit\\_protocol](https://en.wikipedia.org/wiki/Two-phase_commit_protocol)



**Distributed transaction places  
a lock on resources under  
transaction to ensure isolation**



# Inappropriate for long-running operations



# Increase risk of contention and deadlock

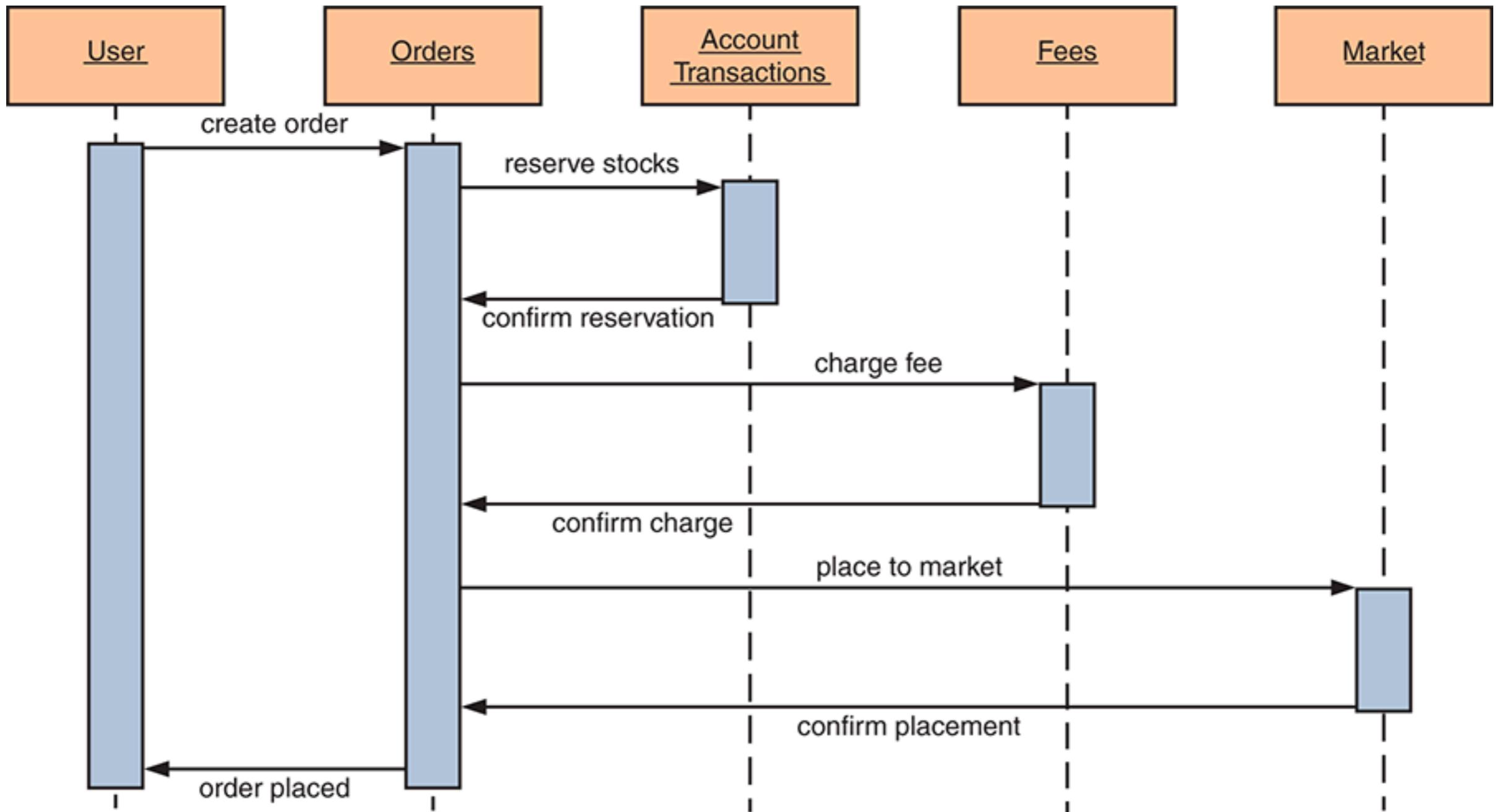


# We need ...

Don't need distributed transaction  
Reduce complexity of code  
Reliable



# Synchronous process

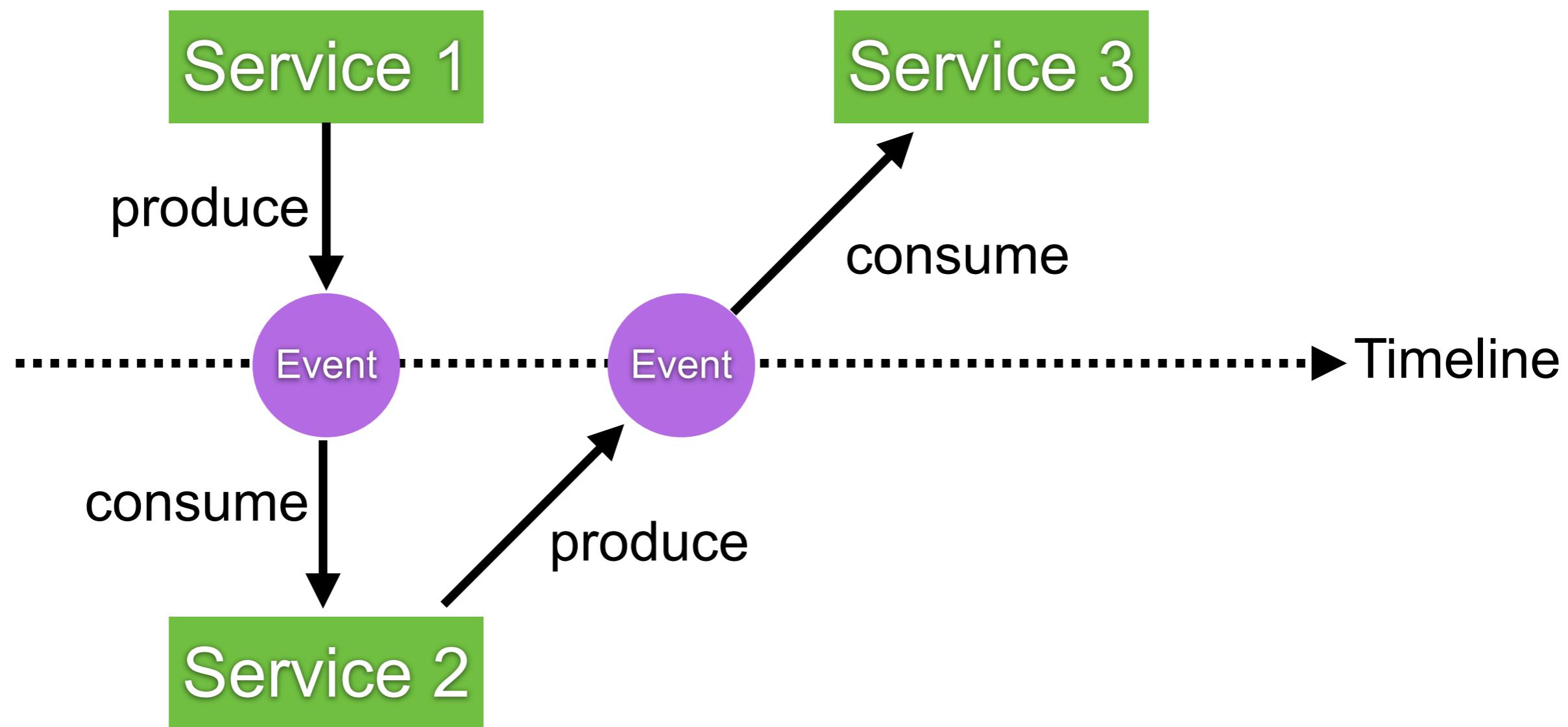


# Event-based communication



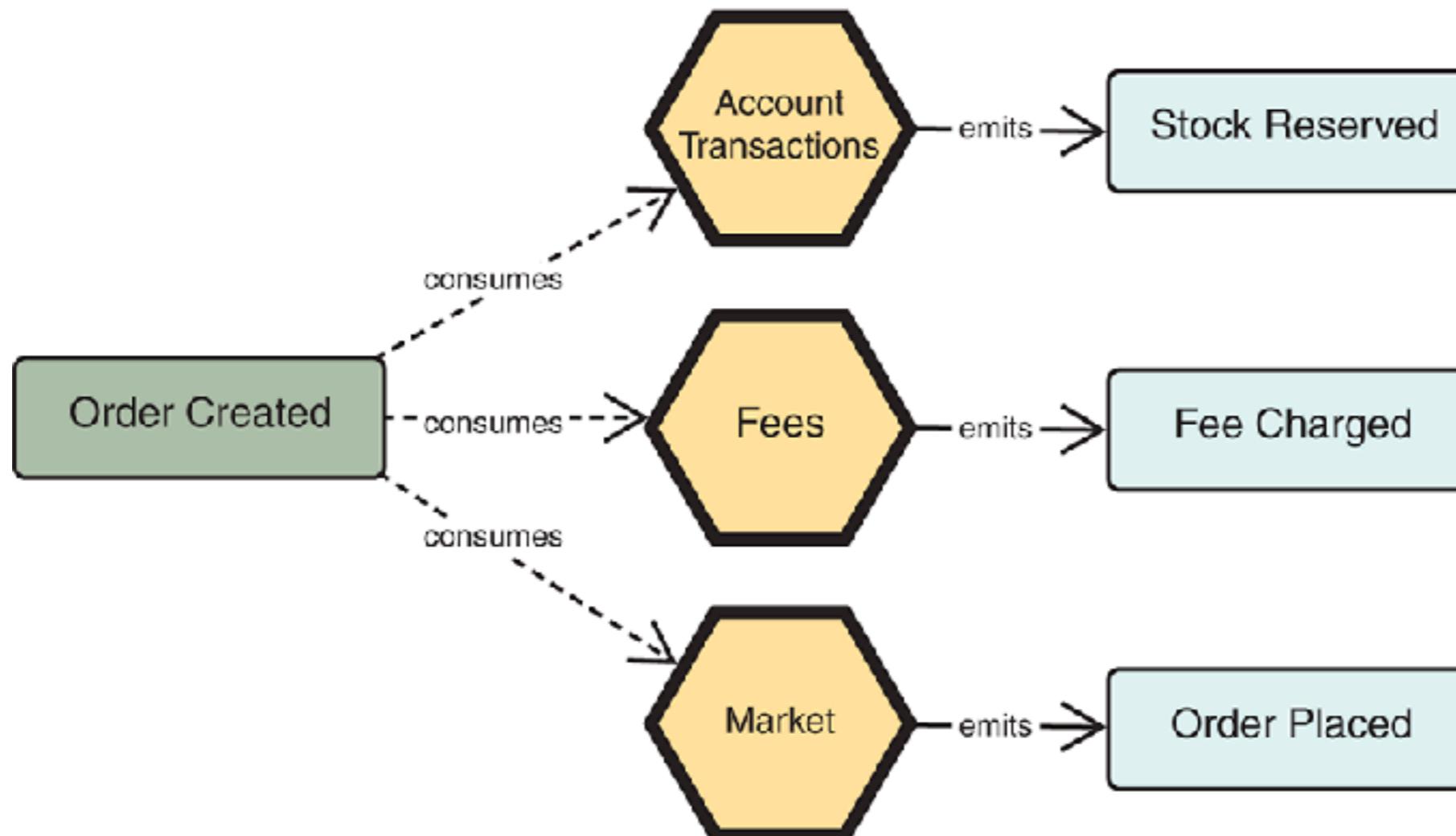
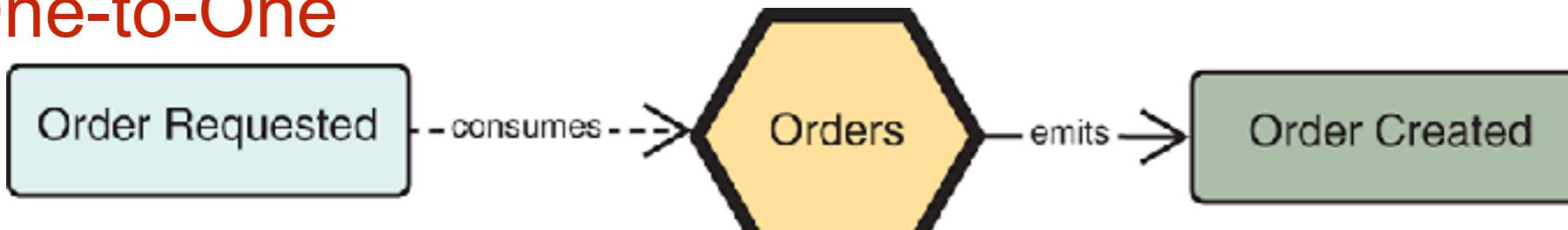
# Choreography pattern

Sequence of Tx and emit **event** or **message** that trigger the next process in Tx

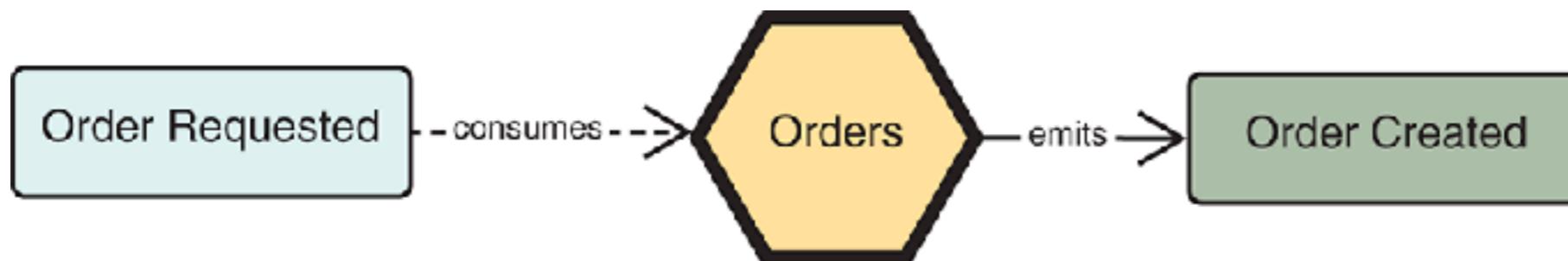


# Service consume and emit event

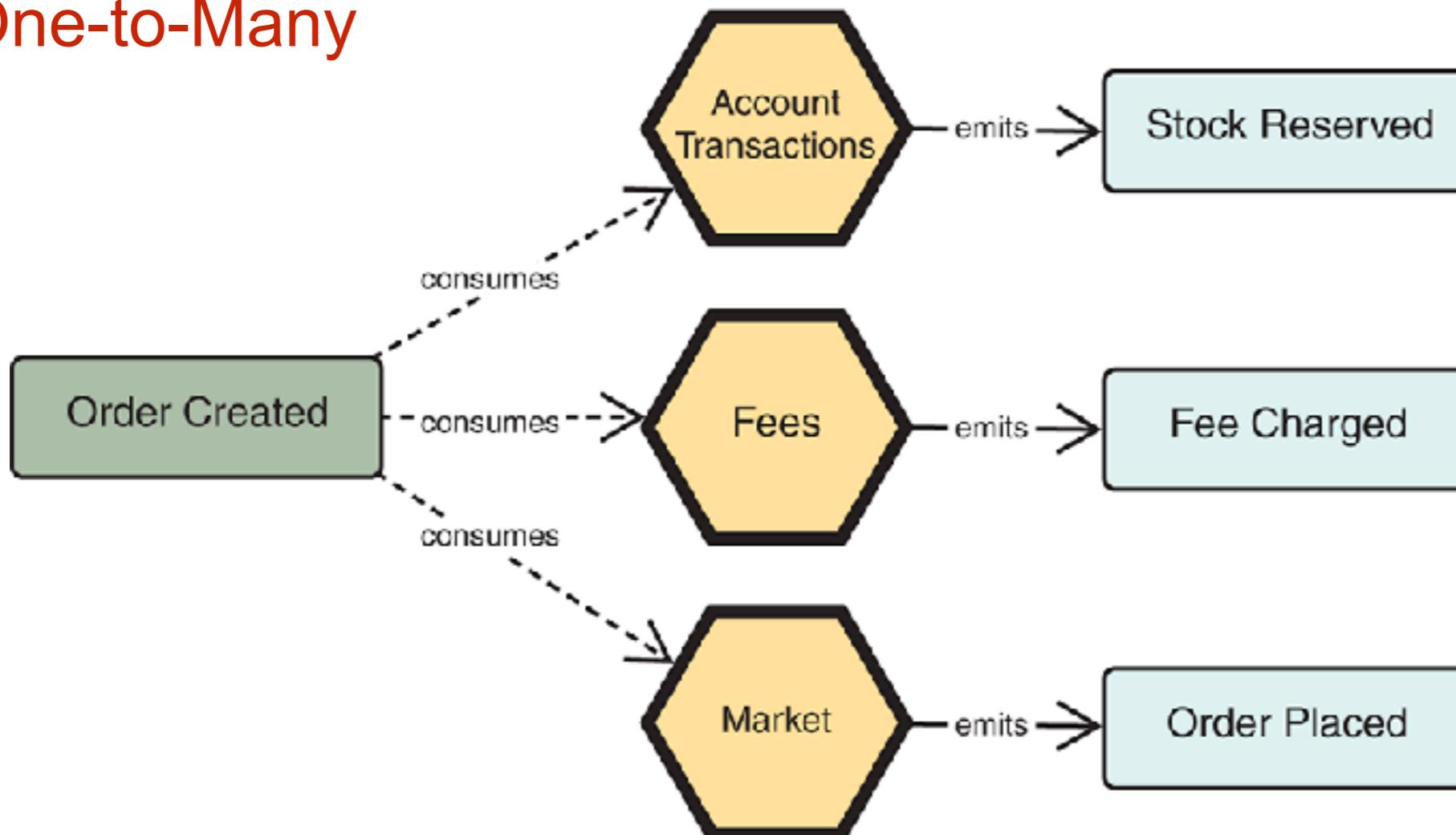
## One-to-One



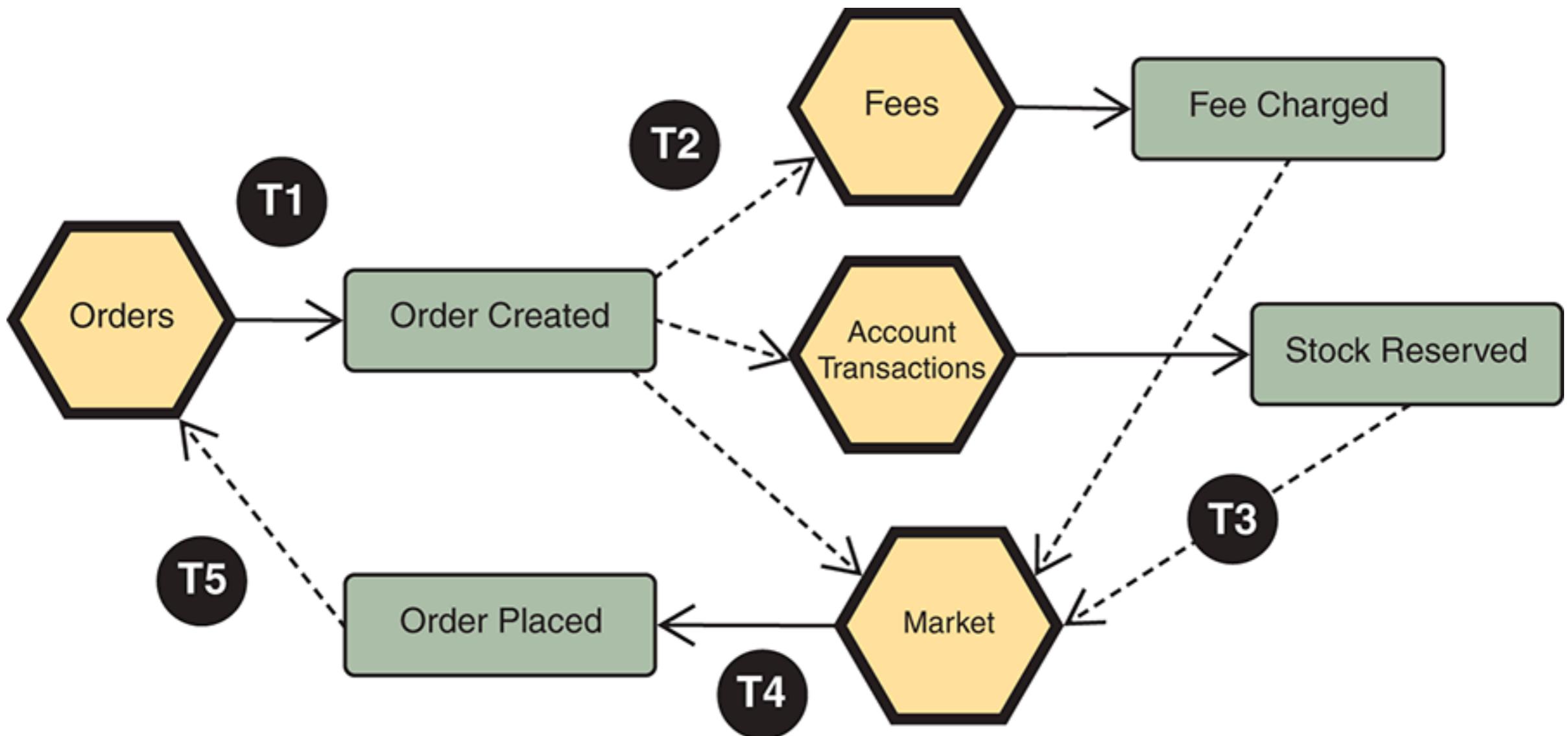
# Service consume and emit event



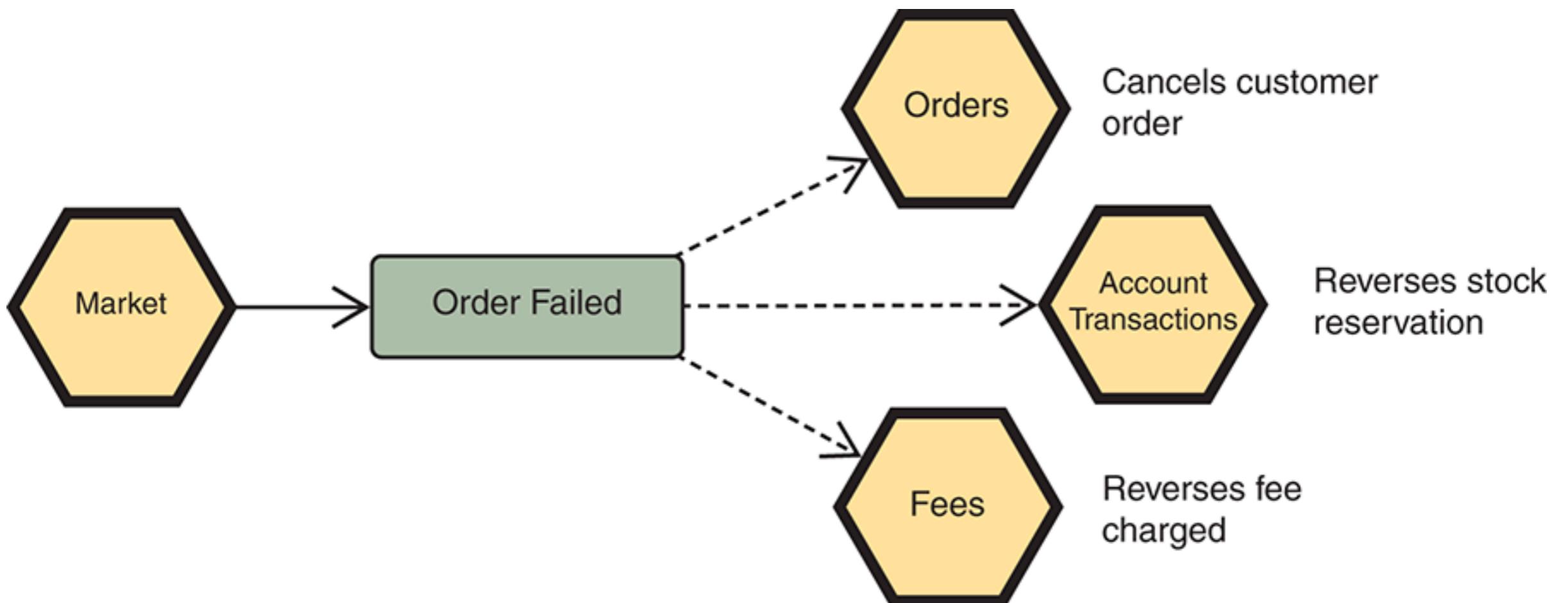
One-to-Many



# Example (Success)

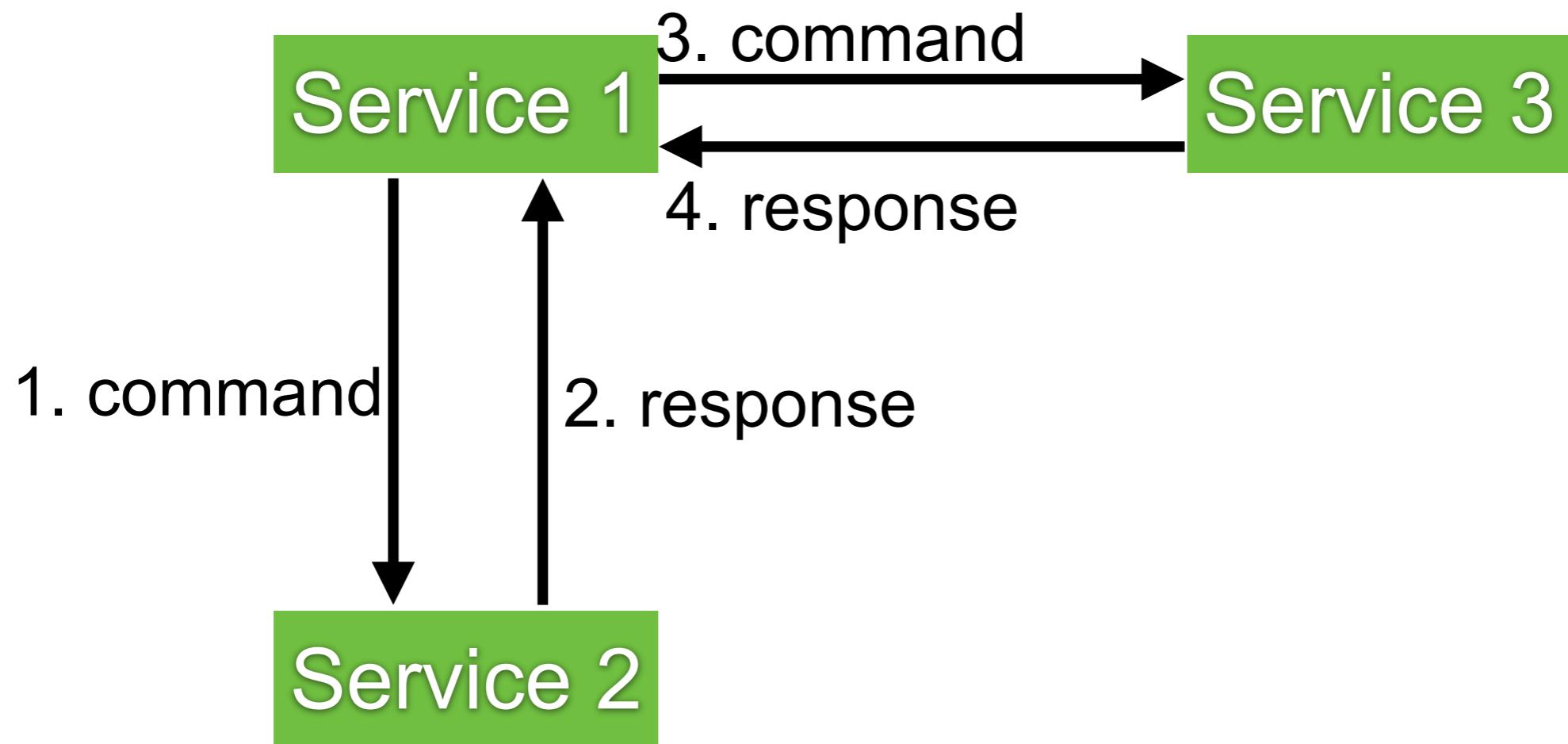


# Example (Fail)

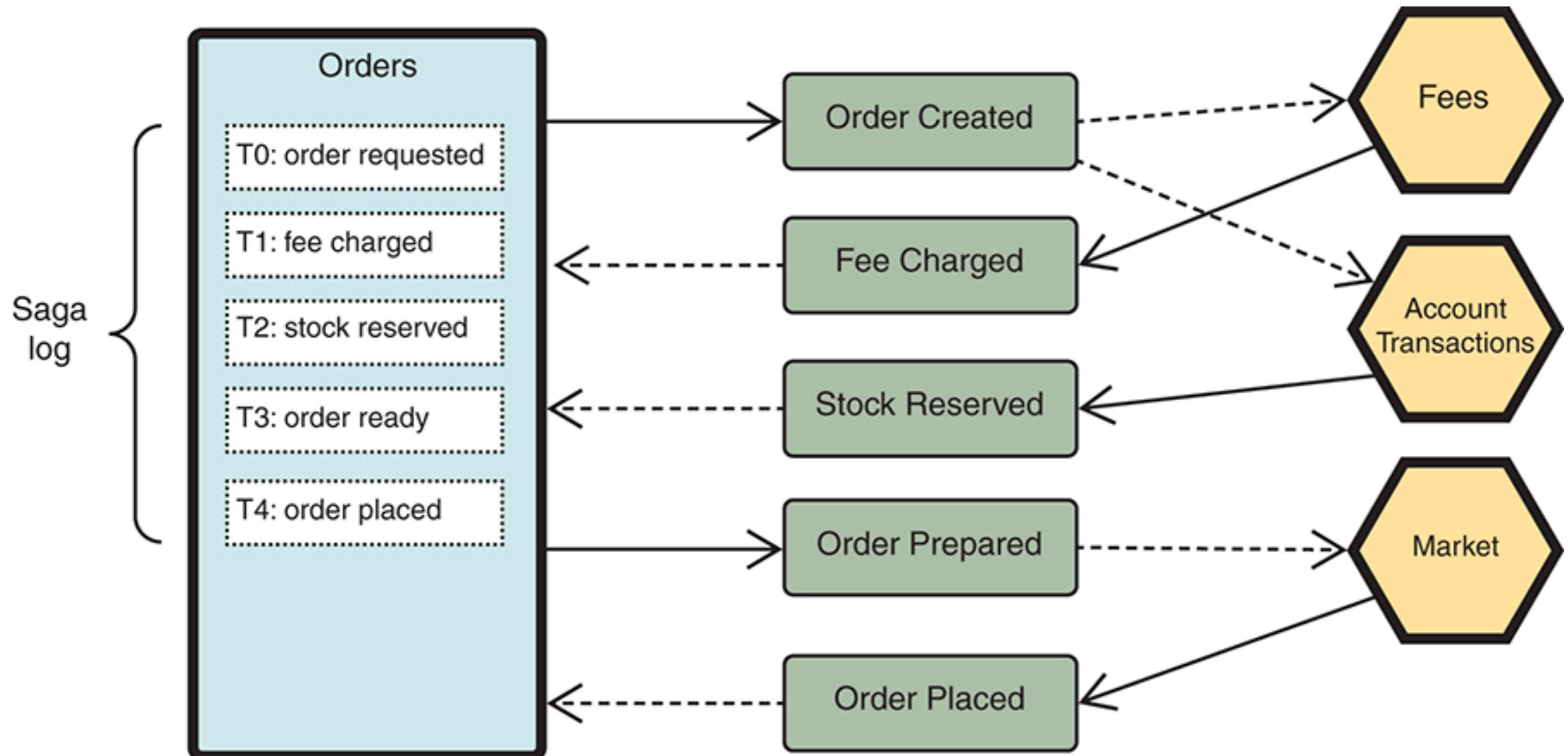


# Orchestrate pattern

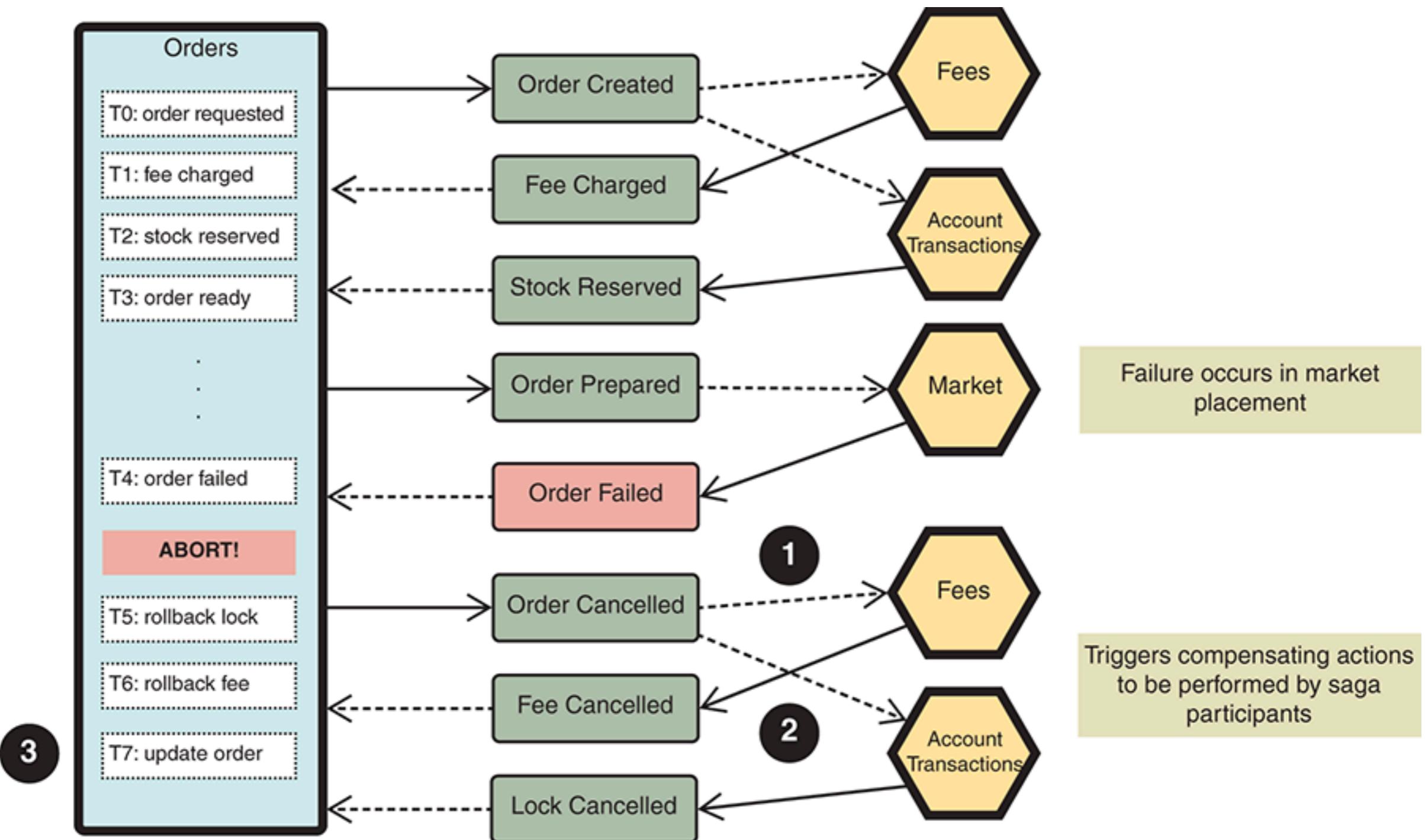
Sequence of Tx and emit **event or message** that trigger the next process in Tx



# Example (Success)



# Example (Fail)



# Consistency patterns

Name	Strategy
Compensating action	Perform action that undo prior action(s)
Retry	Retry until success or timeout
Ignore	Do nothing in the event of errors
Restart	Reset to the original state and start again
Tentative operation	Perform a tentative operation and confirm (or cancel) later



# “Saga”



# **Message vs Event**

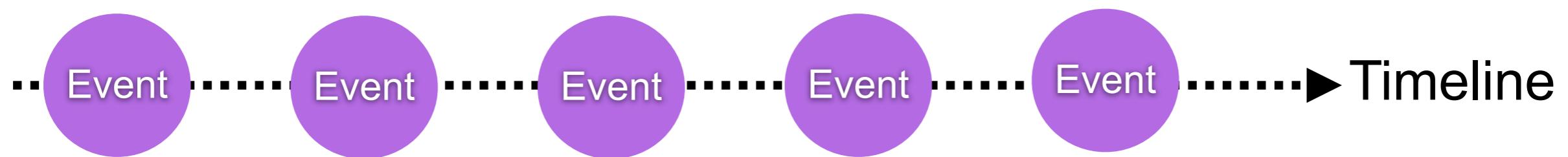
**Message** is addressed to someone

**Event** is something that happen and someone can react to that



# Event sourcing

Maintain source of truth of business  
Immutable sequence of events

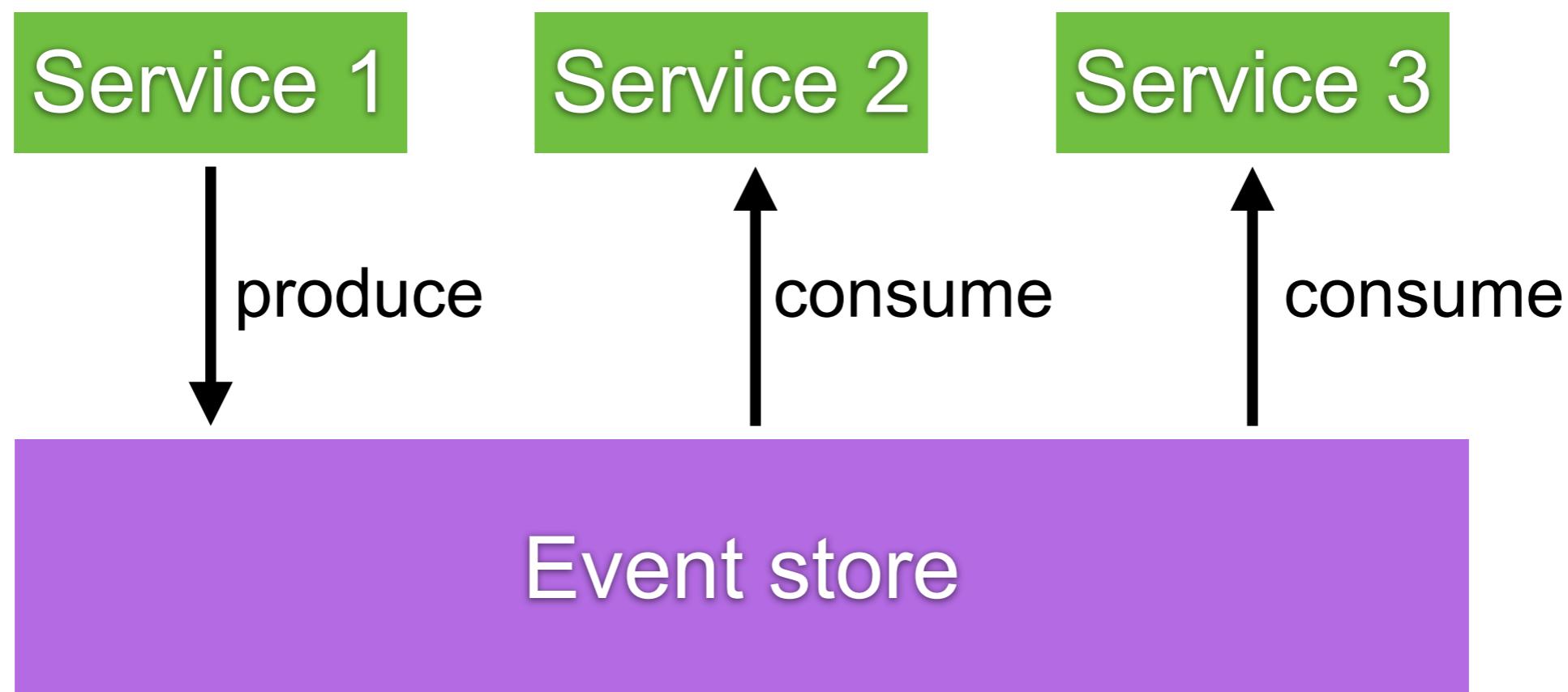


Sequence of event is keep in **Event store**



# Event store

Keep events in order  
Broadcast new event

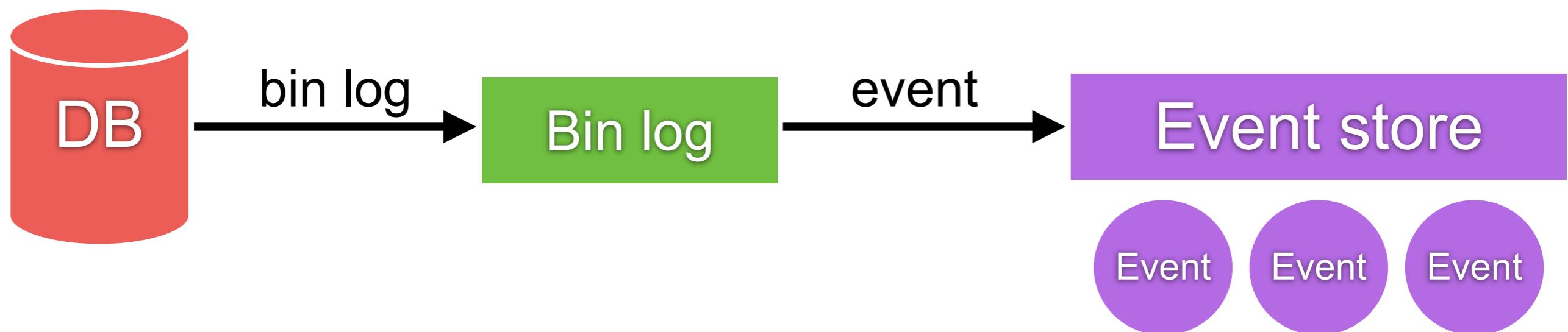


**With Event sourcing, we can  
decouple services (query and  
command)**



# Event sourcing with database

## Working with database



Binlog or Binary log event is information about data modification made to database



# Event sourcing

Duplication data (denormalize database)  
Complexity (separate query and command)  
Difficult to maintain



# Event sourcing can't solve all problems



# Start with **understand** your purpose



Start with understand your  
**problem** to resolve



# How to implement queries ?



# **Query data from multiple services**

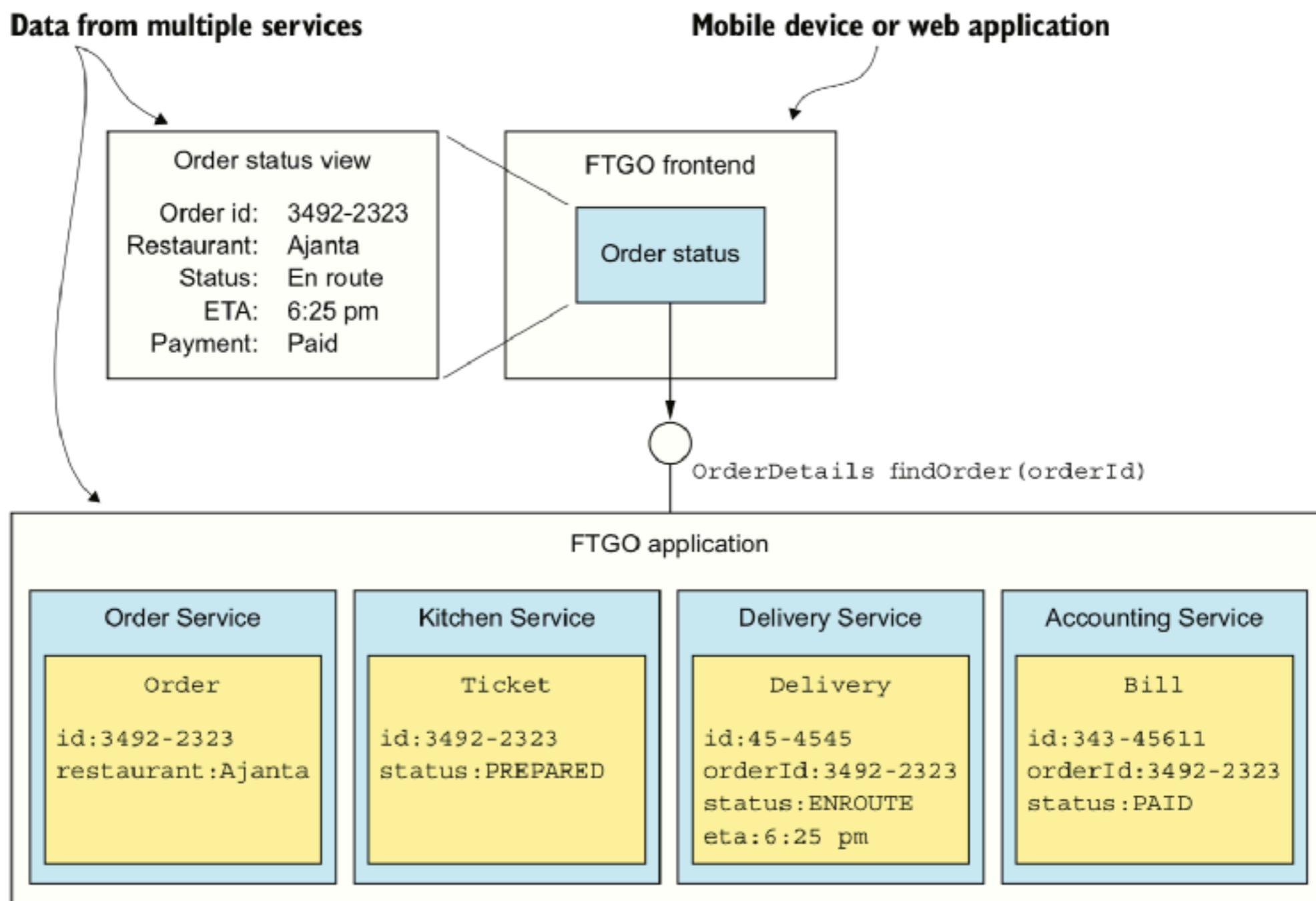
**API composition**

**Cold data from services**

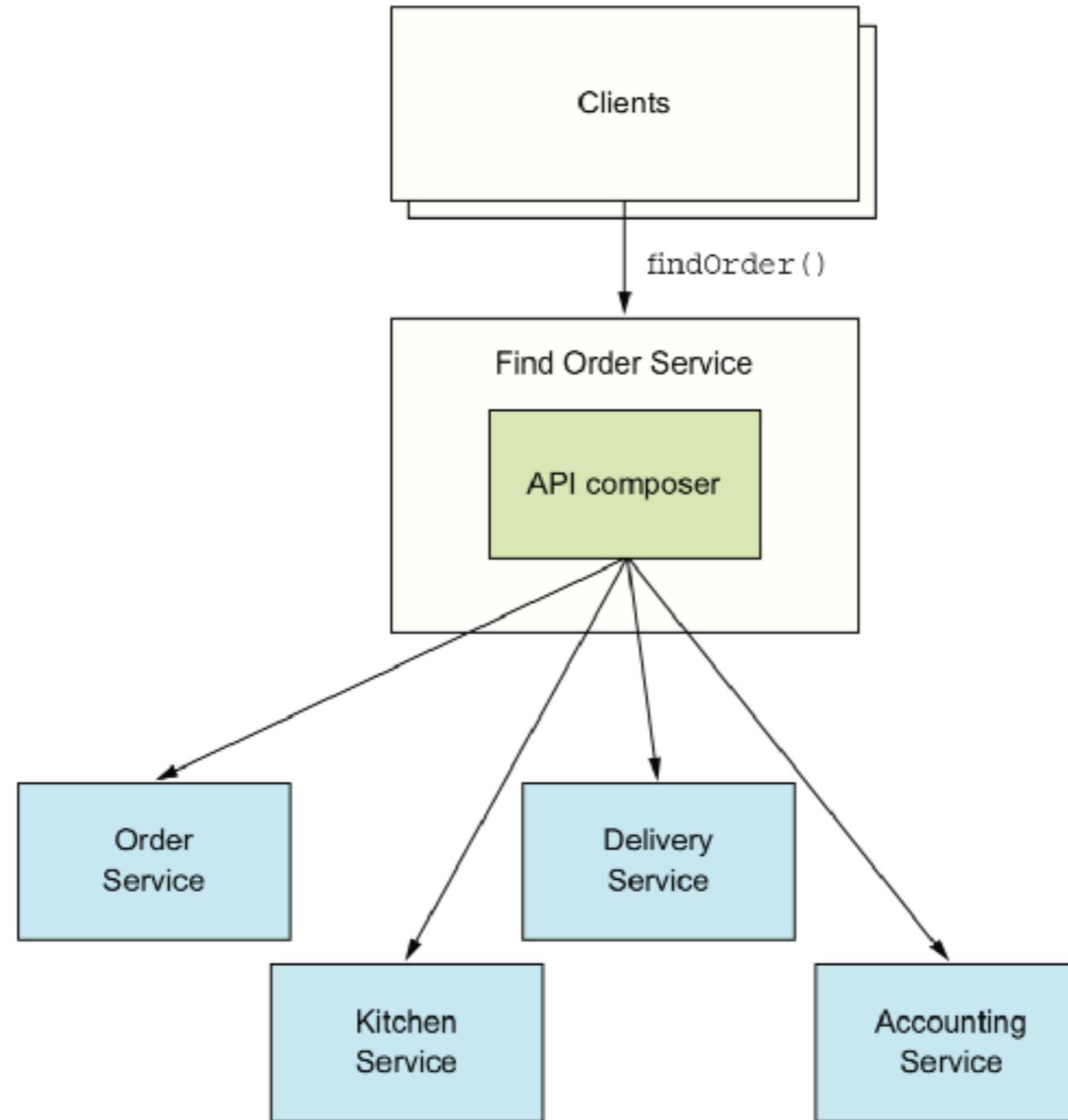
**CQRS (Command Query Responsibility Segregation)**



# Problem ?



# API composition pattern



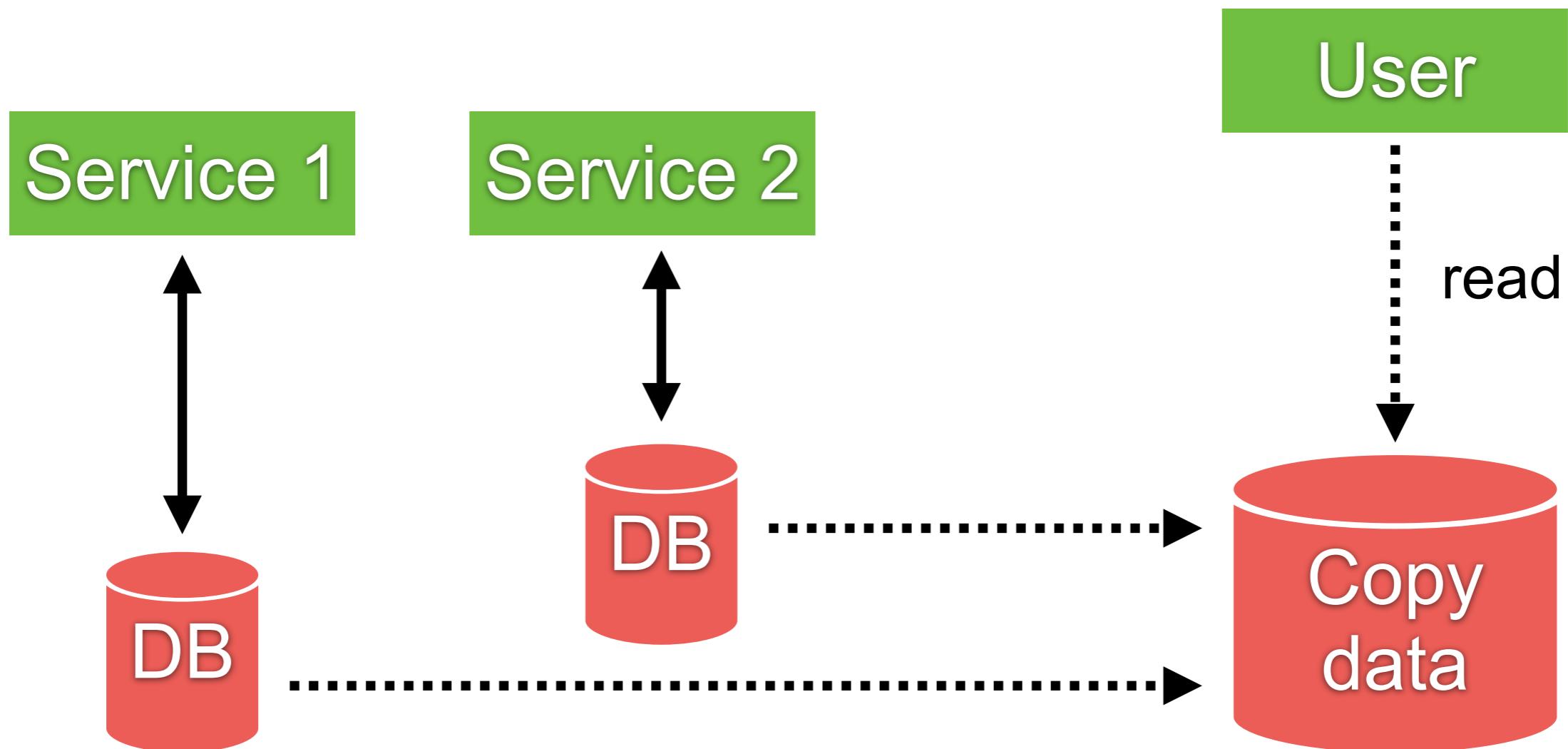
# Drawbacks

Increase overhead  
Lack of transactional data consistency  
Reduce availability



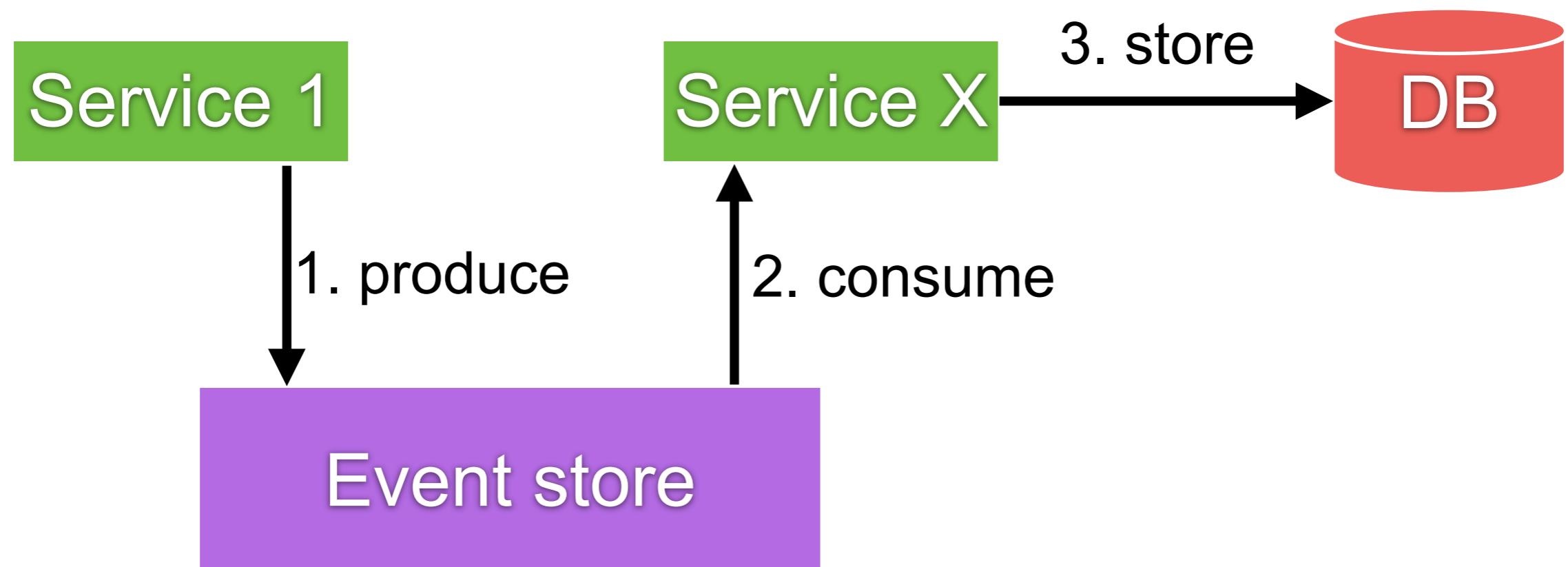
# Storing copy data

Copy or caching data to other database



# Working with Event-based

Publish event to store copy data

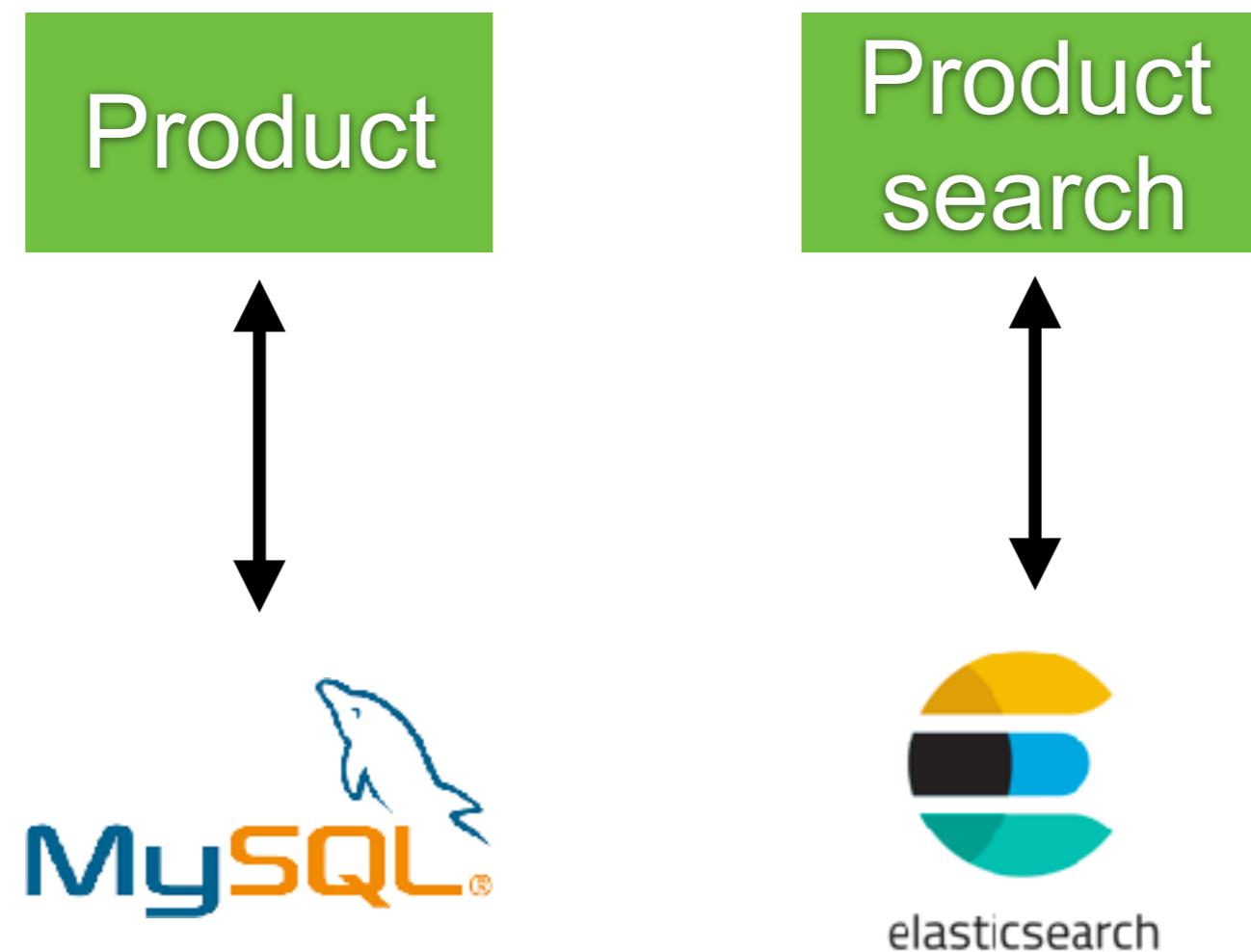


# Separate query and command



# Separate data for read and write

For example MySQL to write, Elasticsearch to search

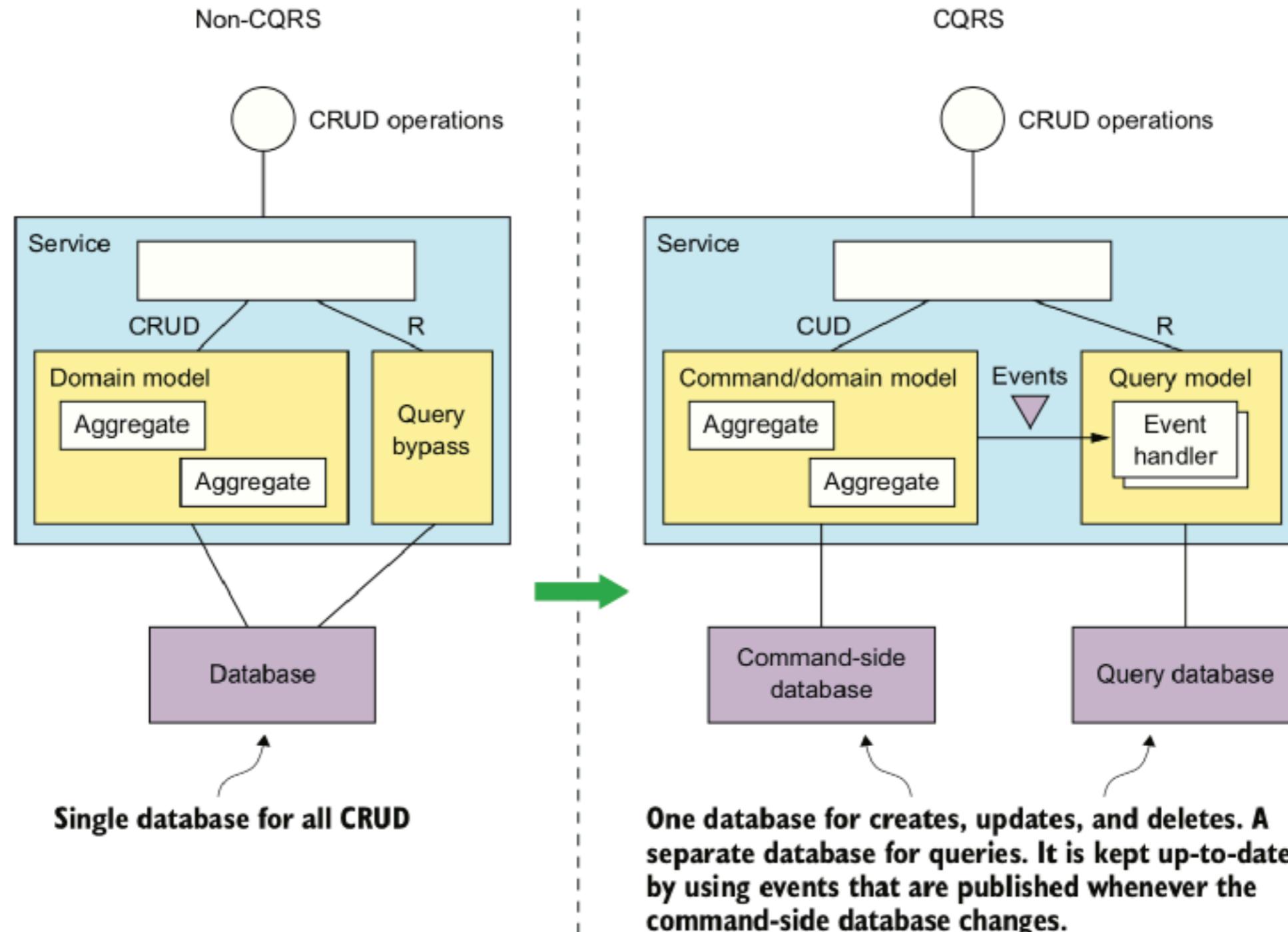


# CQRS

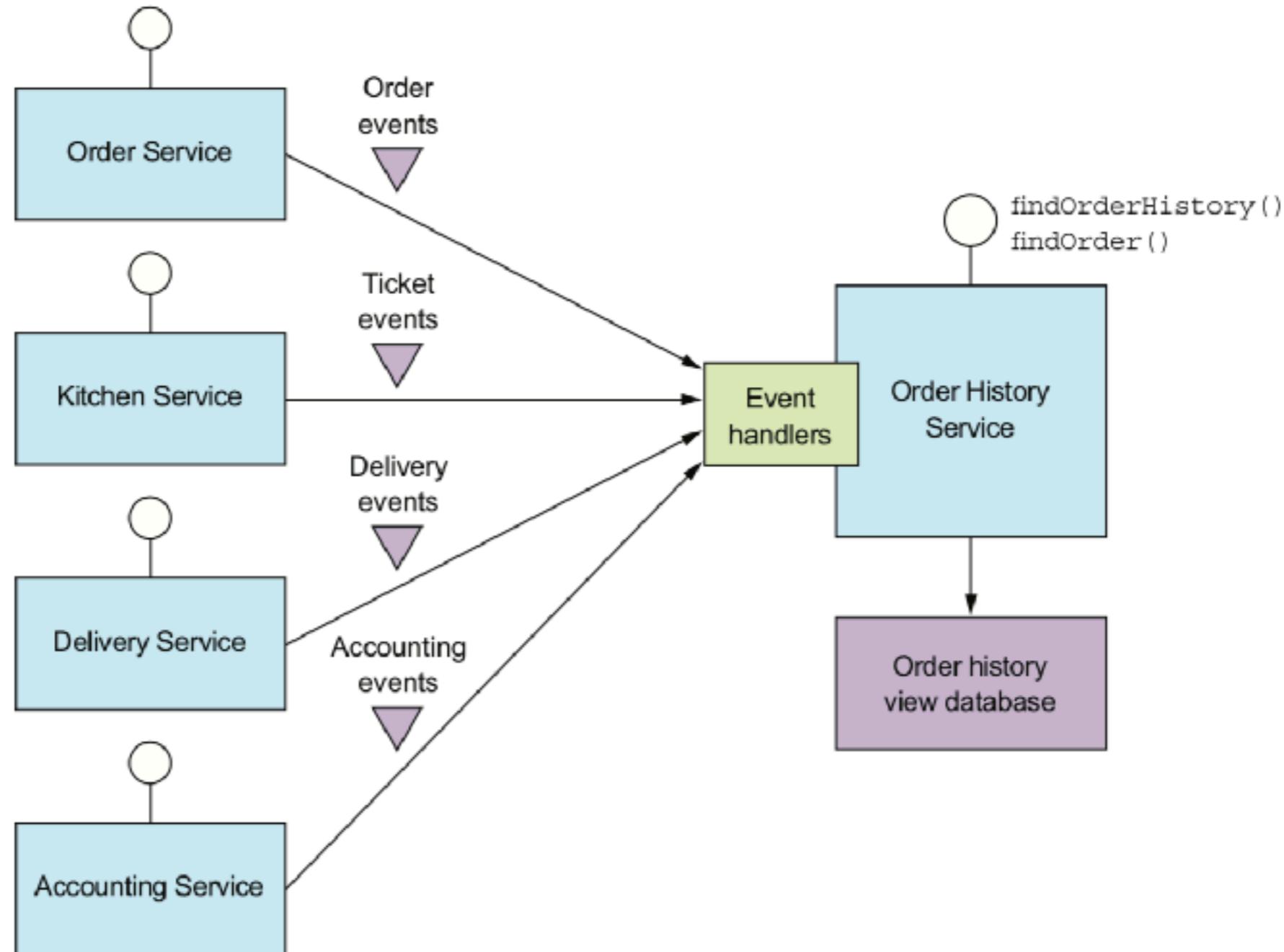
## Command Query Responsibility Segregation



# CQRS = Separate command from query



# CQRS = Query-side service



# Benefits

Improve separation of concern  
Efficiency query



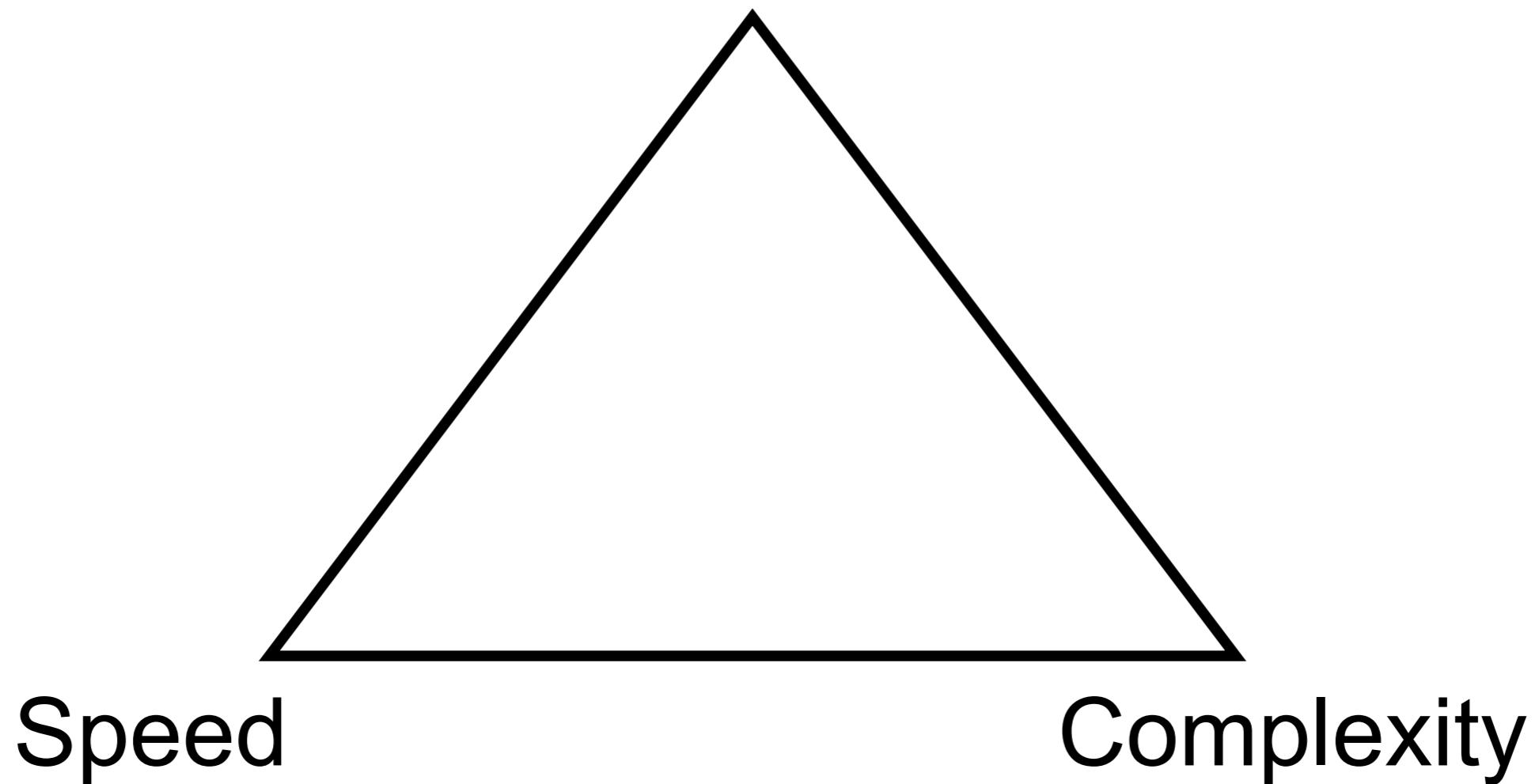
# Drawbacks

More complex

Lag between command and query side



# Customer value

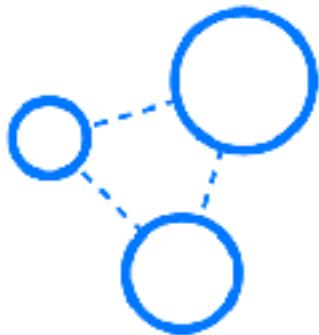


# More ...



# Beyond Microservice

## Process and Organization



### Flexible organizational structure

With clear roles and accountabilities



### Efficient meeting formats

Geared toward action and eliminating over-analysis



### More autonomy to teams and individuals

Individuals solve issues directly without bureaucracy



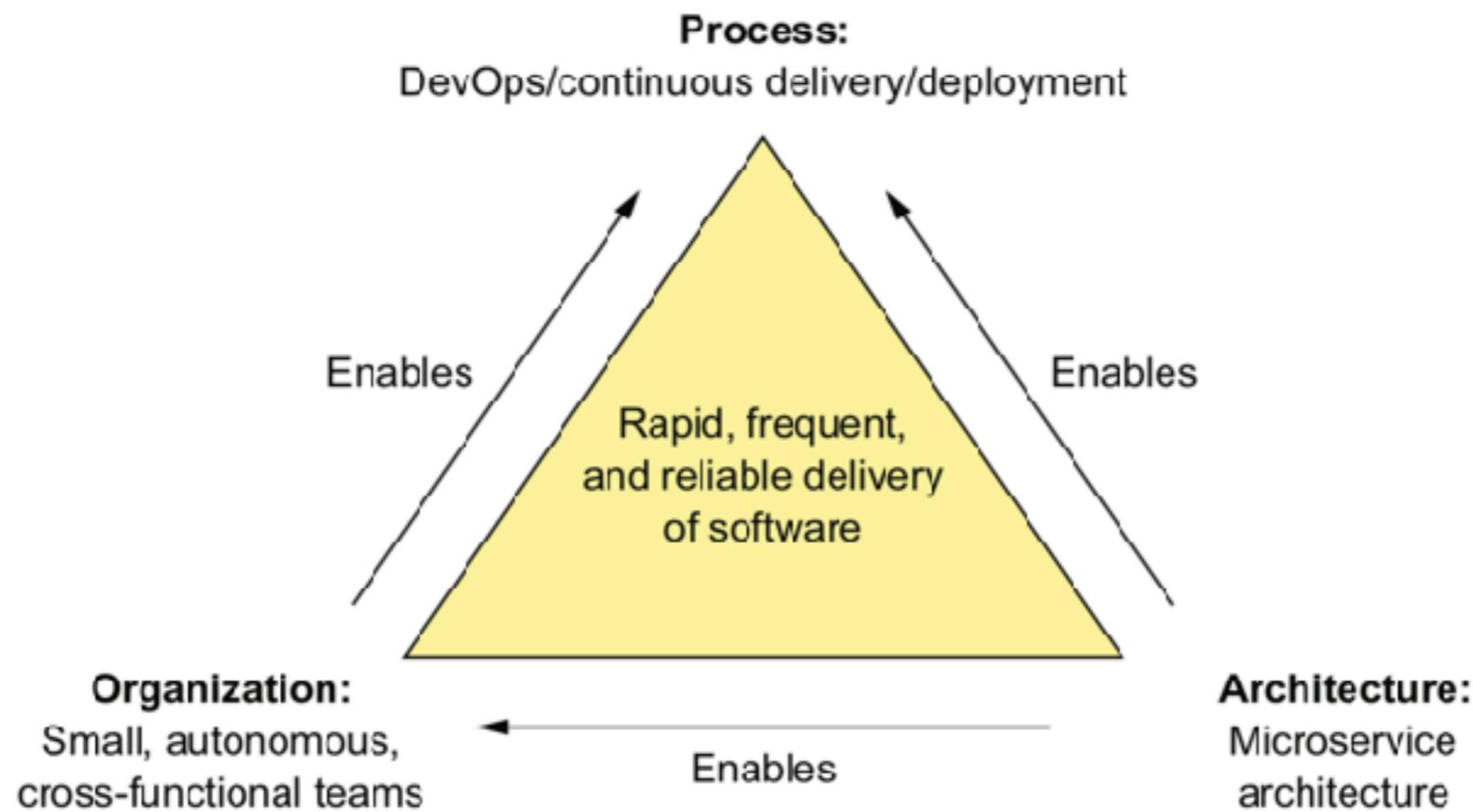
### Unique decision-making process

To continuously evolve the organization's structure.



# Success project needs ...

Organization process  
Development process  
Delivery process



# Don't forget about the human side when adopt Microservice



“organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations”

- *Melvin Conwey, 1968* -



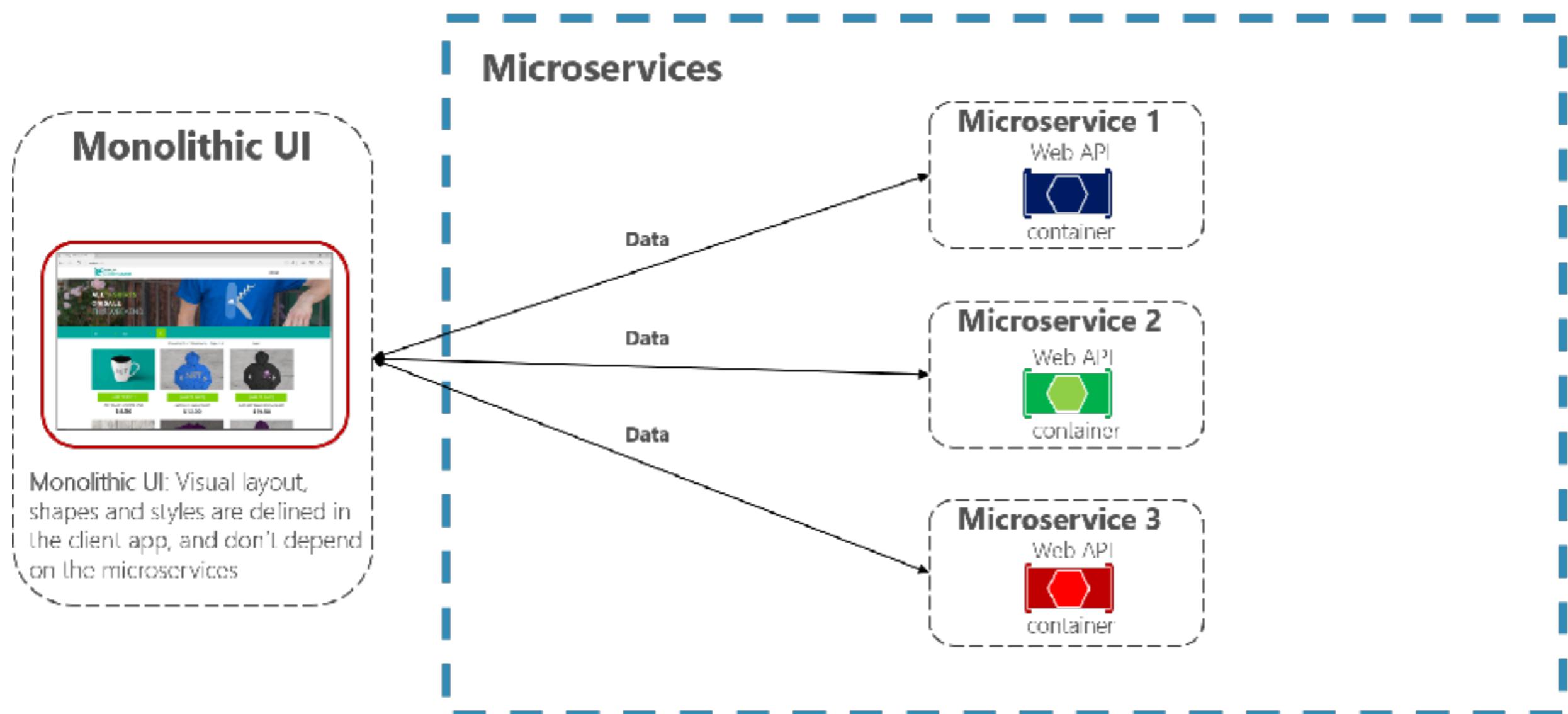
# More ...



# **Integrate services with User Interface**



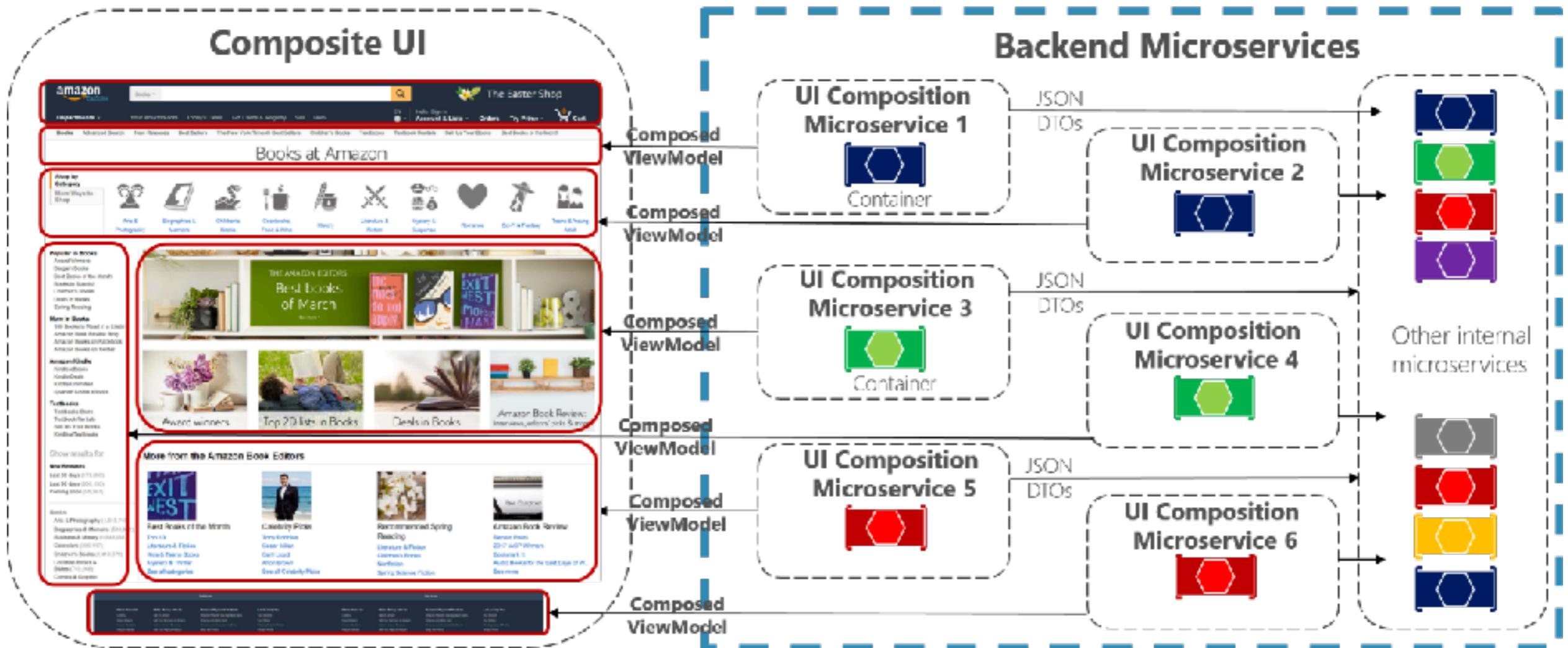
# Monolithic UI consuming microservices



<https://docs.microsoft.com>



# Composite UI generated by microservices



<https://docs.microsoft.com>

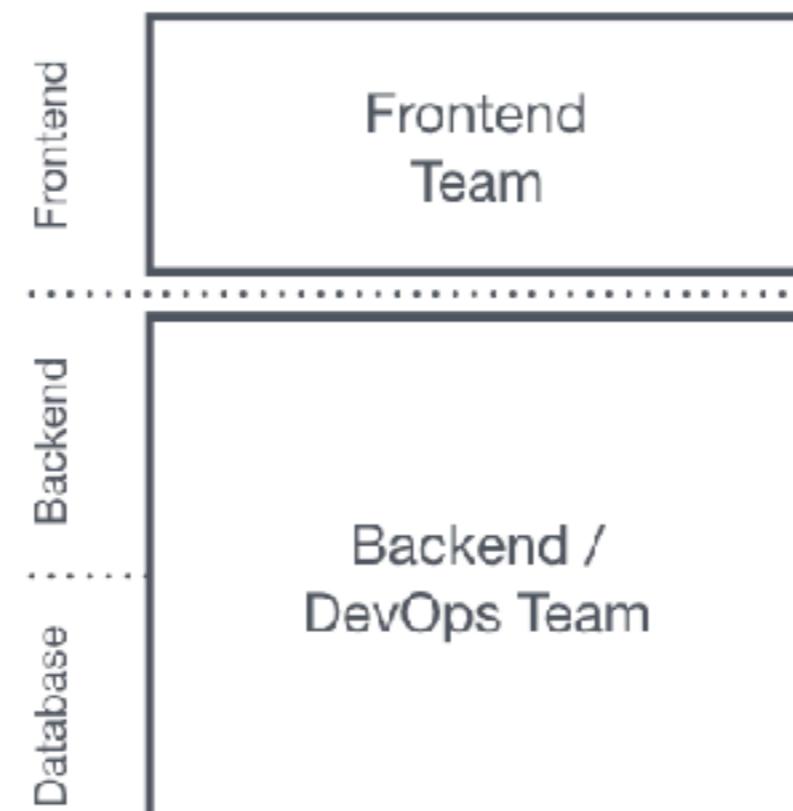


# Monolith frontend

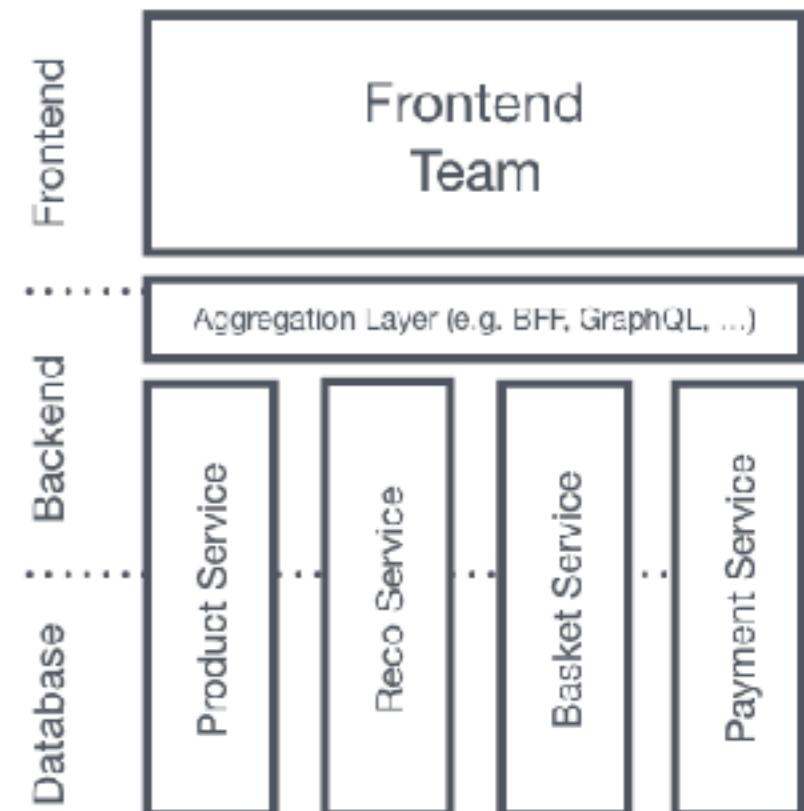
*The Monolith*



*Front & Back*



*Microservices*

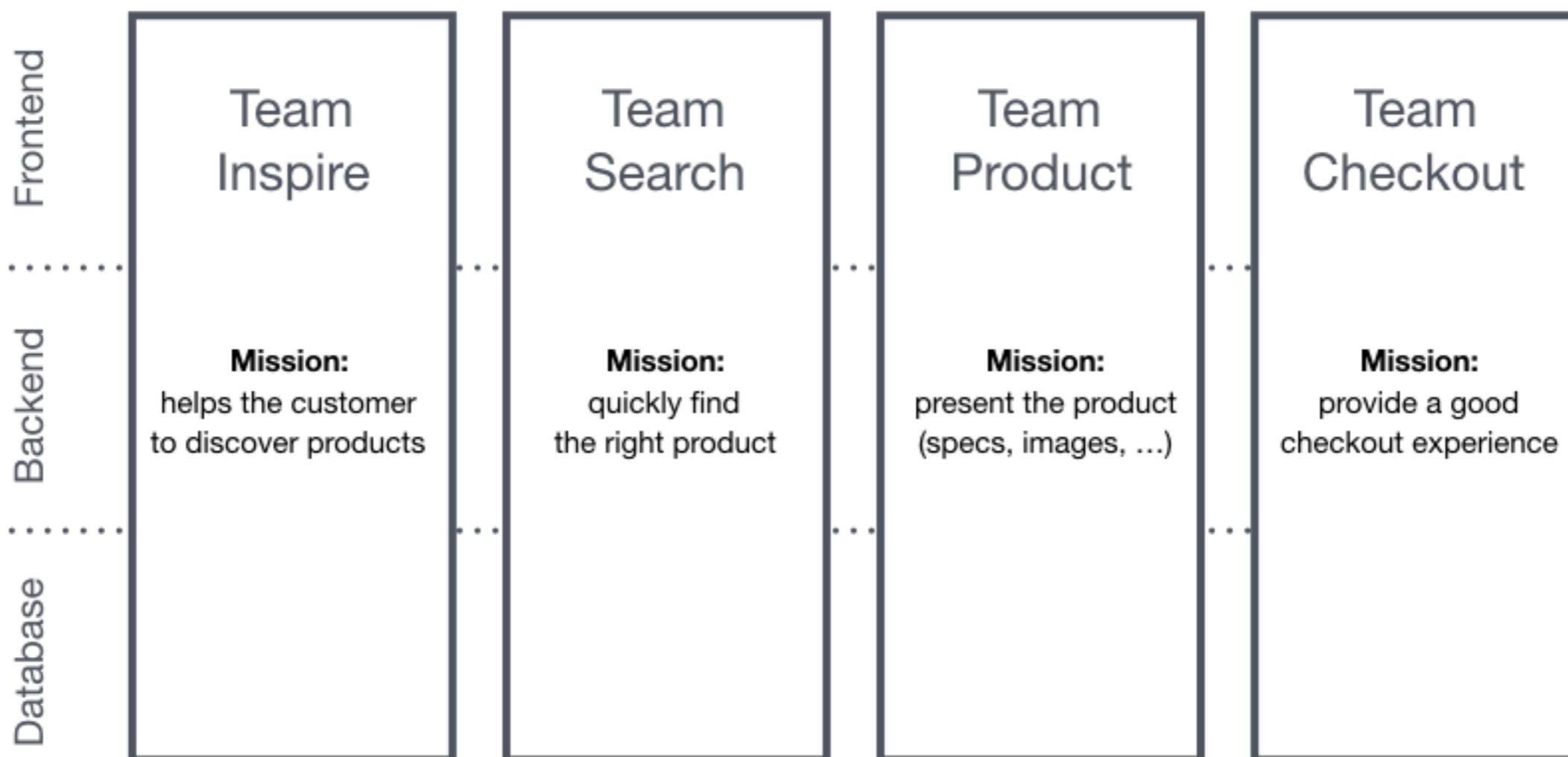


<https://micro-frontends.org/>



# Micro frontend

*End-to-End Teams with Micro Frontends*



<https://micro-frontends.org/>



# Summary



# Let's start with good monolith



Module 1

Module 2

Module 3

Module 4

Module 5

Module 6



# Find your problem



Module 1

Module 2

Module 3

Module 4

Module 5

Module 6



Module 1

Module 2

Module 3

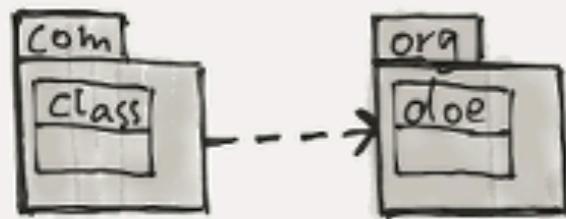
Module 5

Module 6

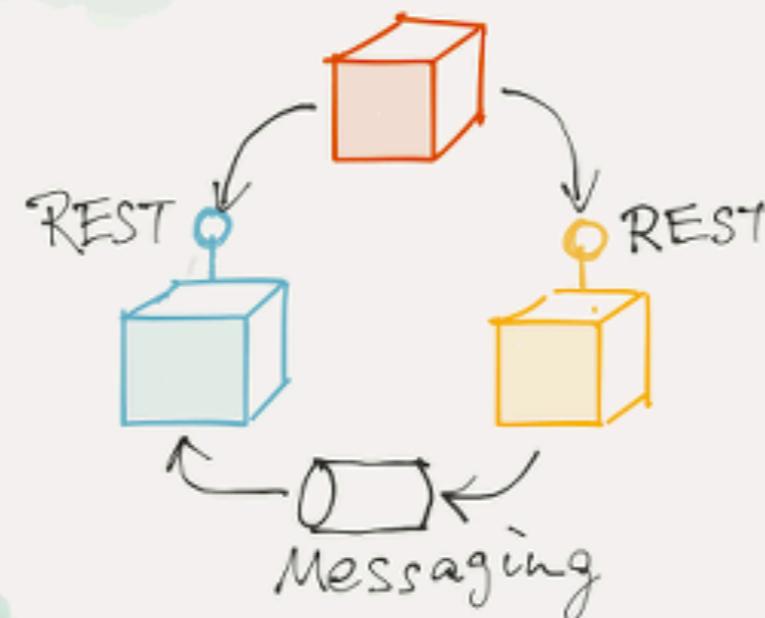
Module 4



# Architecture



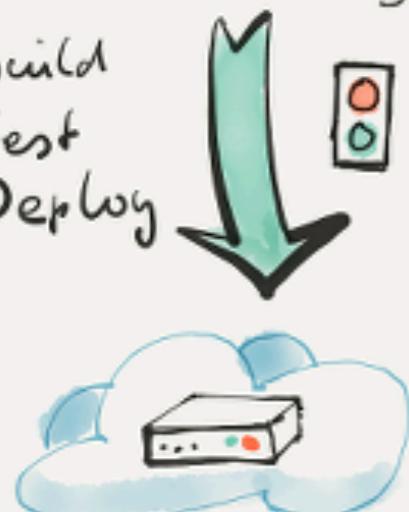
# Microservices



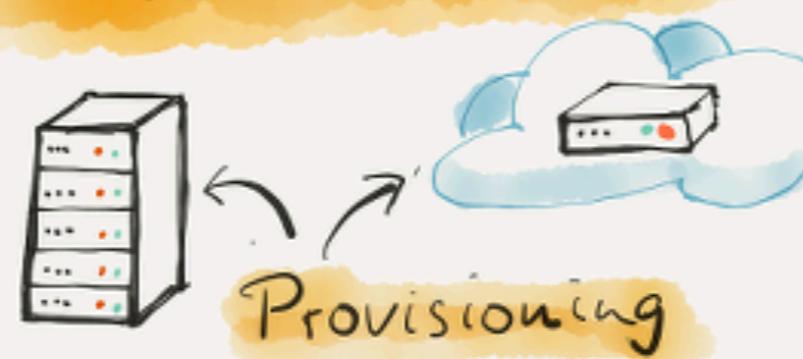
# Deployment

## Continuous Delivery

`{ var i=1; }`  
Build  
Test  
Deploy



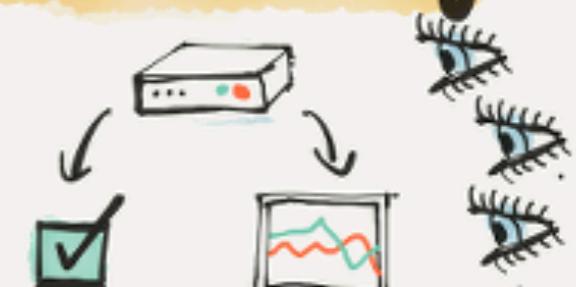
# Infrastructure



# People & Teams



# Monitoring

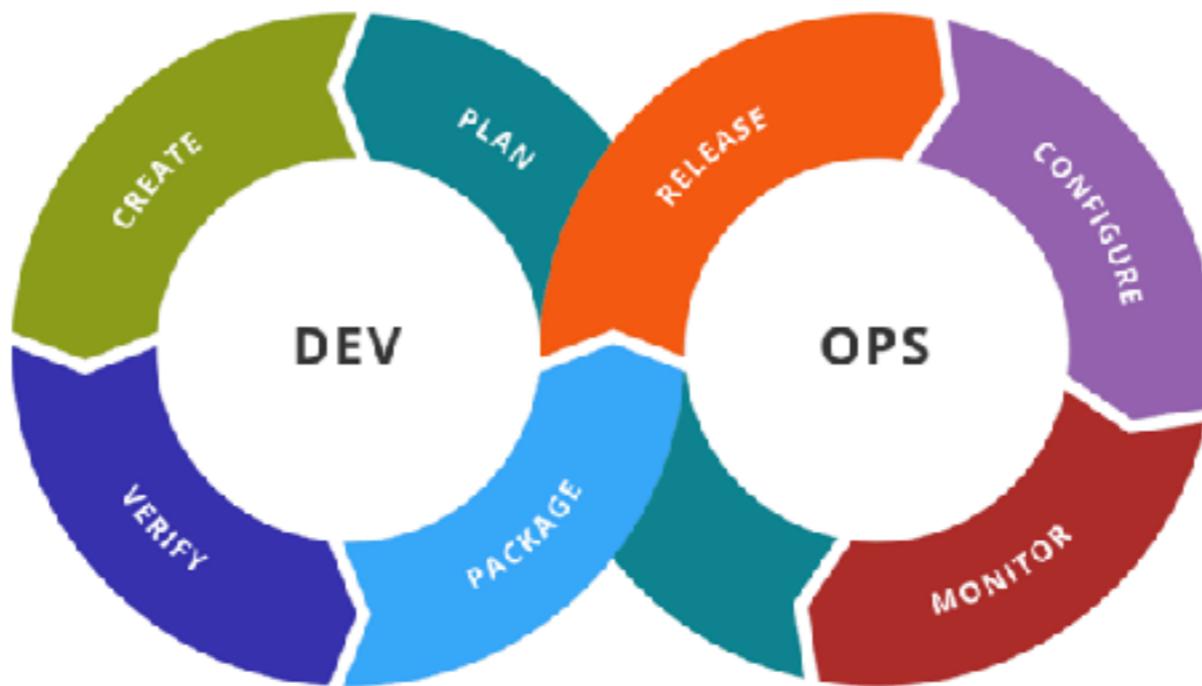


## Features & Technology



# Microservice prerequisites

Rapid provisioning  
Basic monitoring  
Rapid Application Deployment  
DevOps culture



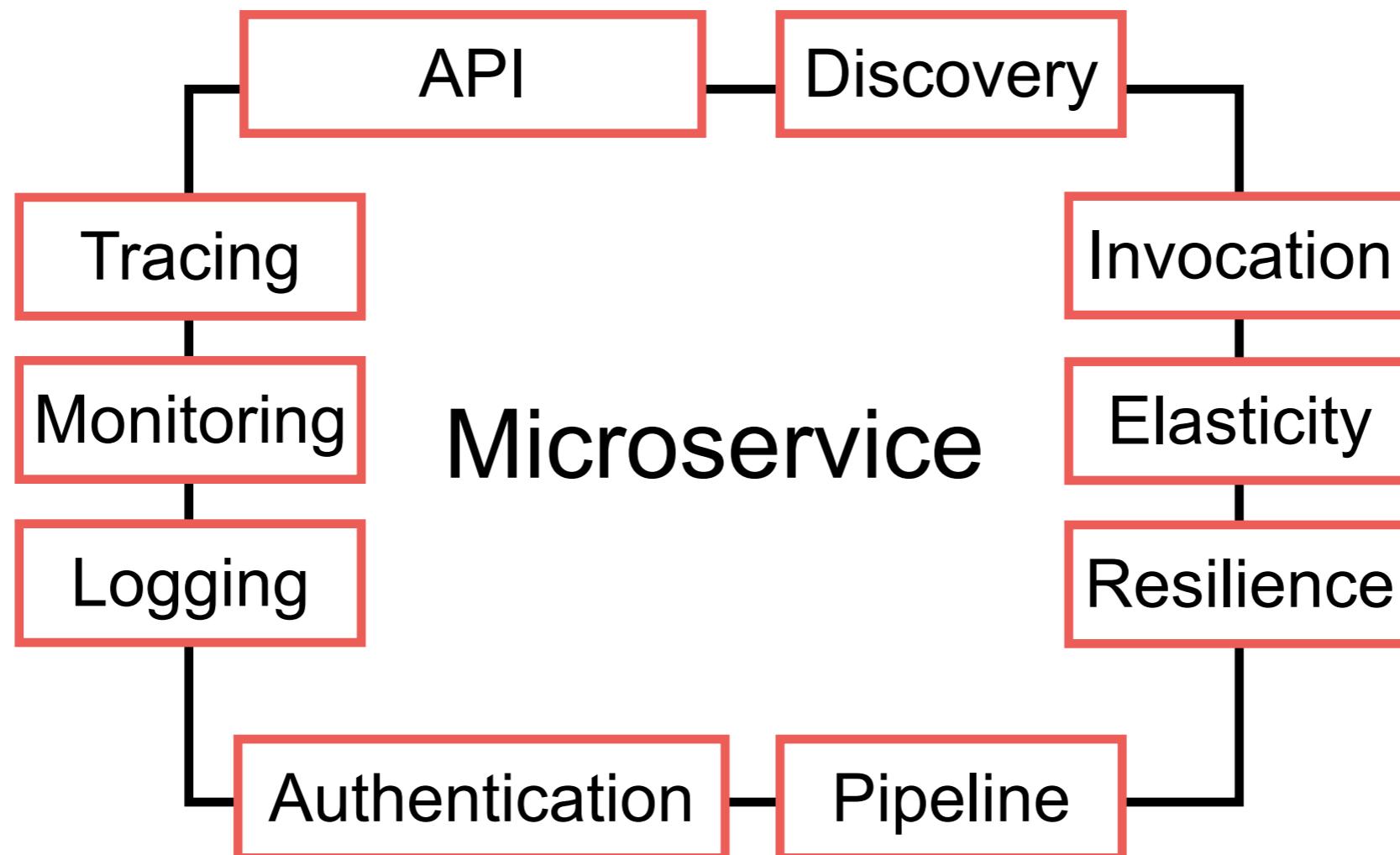
# Design Workshop



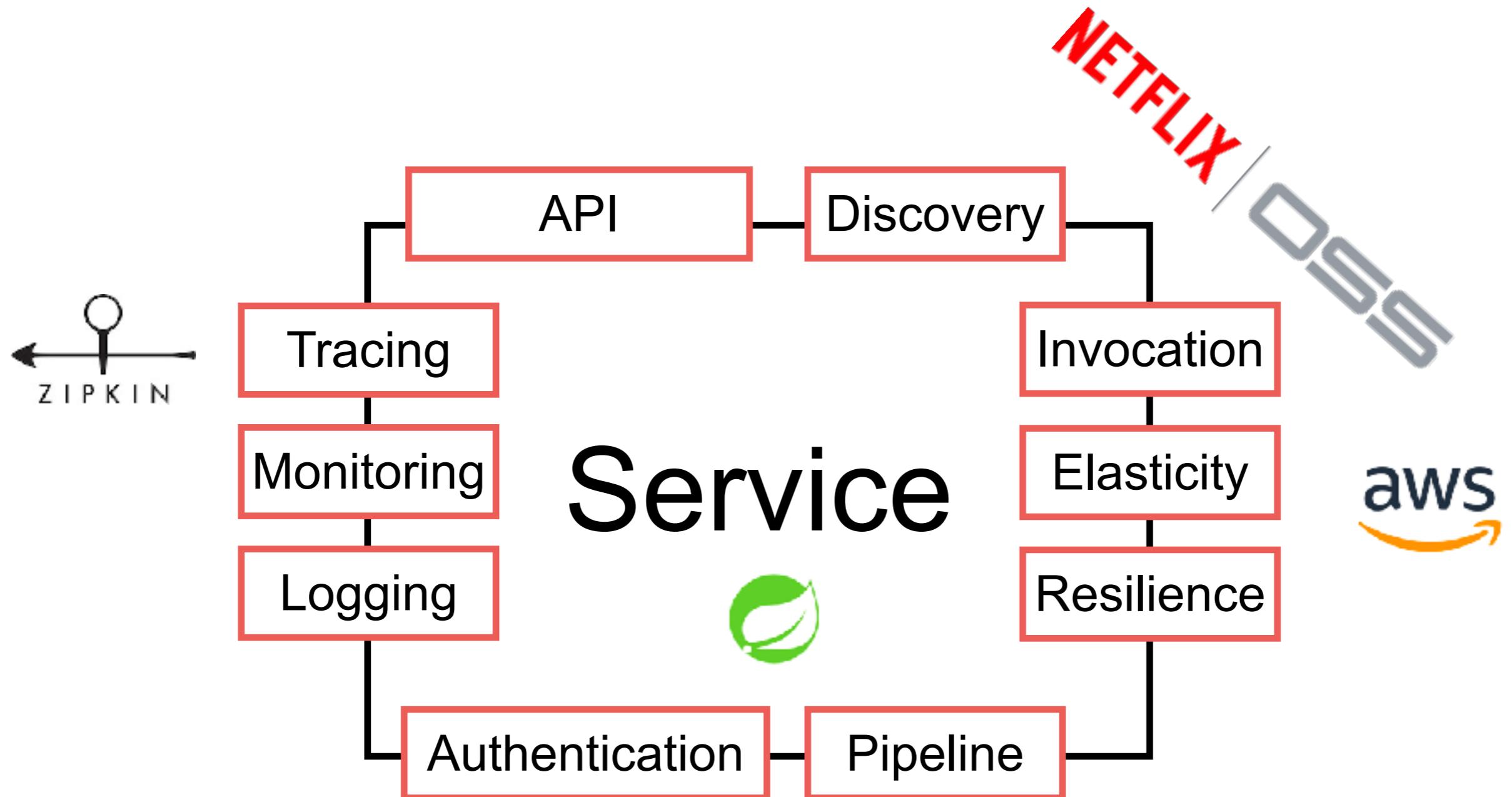
# Module 2 : Develop



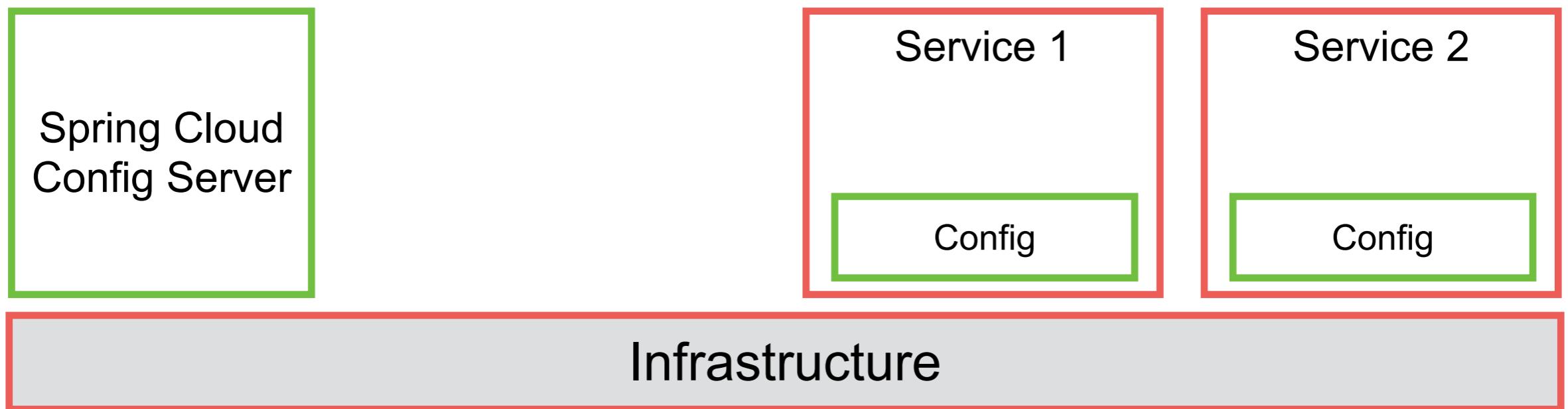
# Properties of Microservice



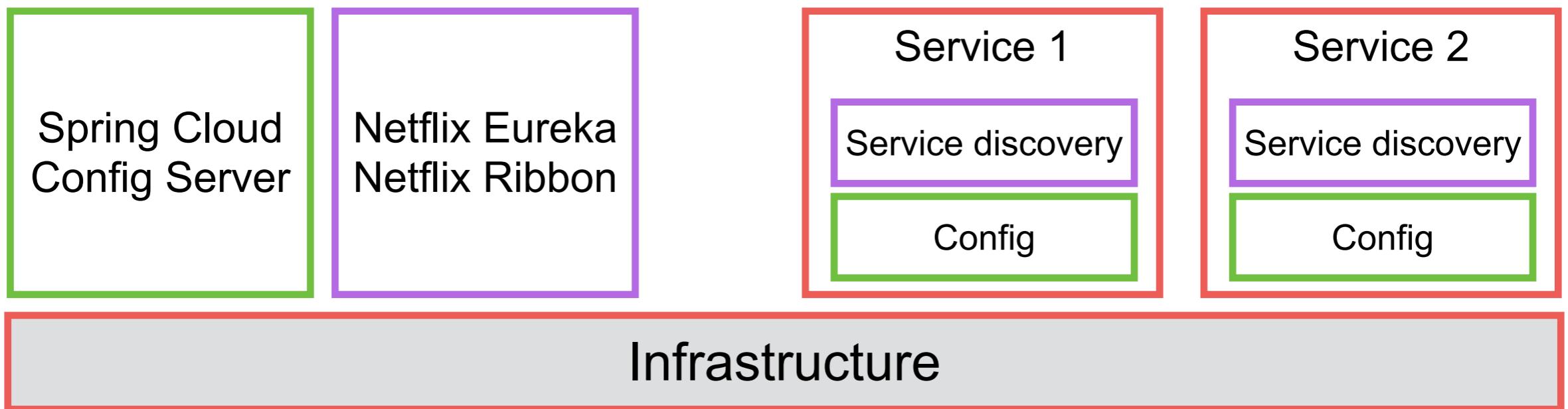
# Microservice 1.0



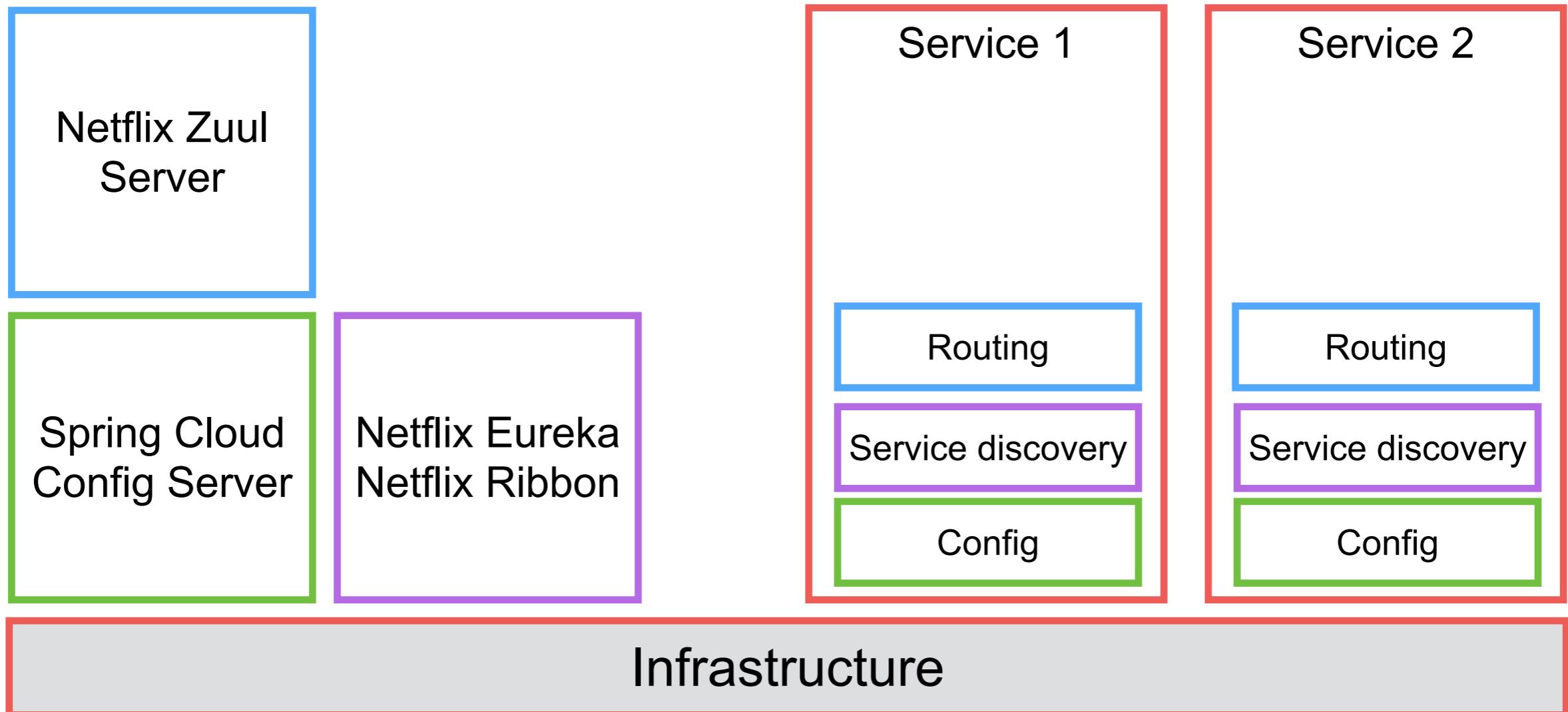
# Configuration



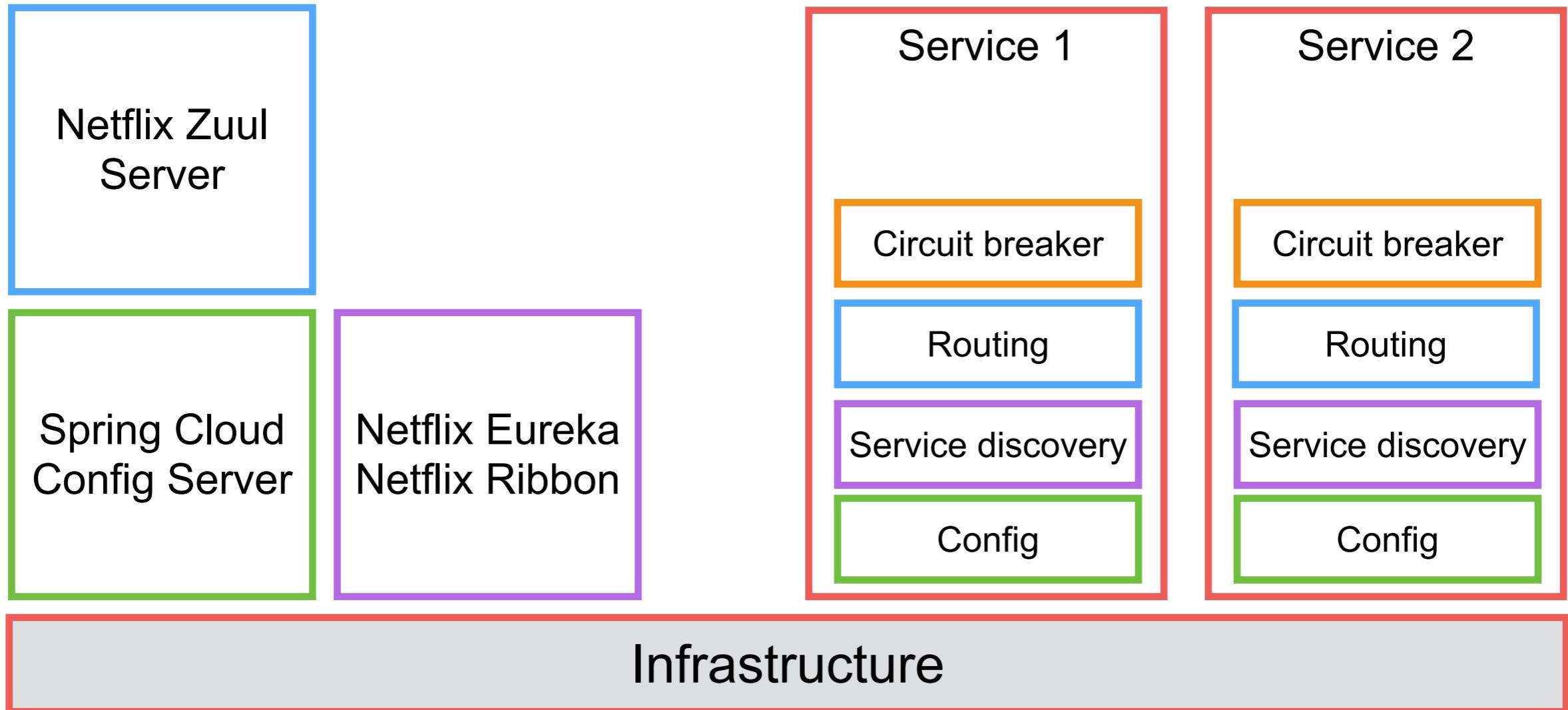
# Service Discovery



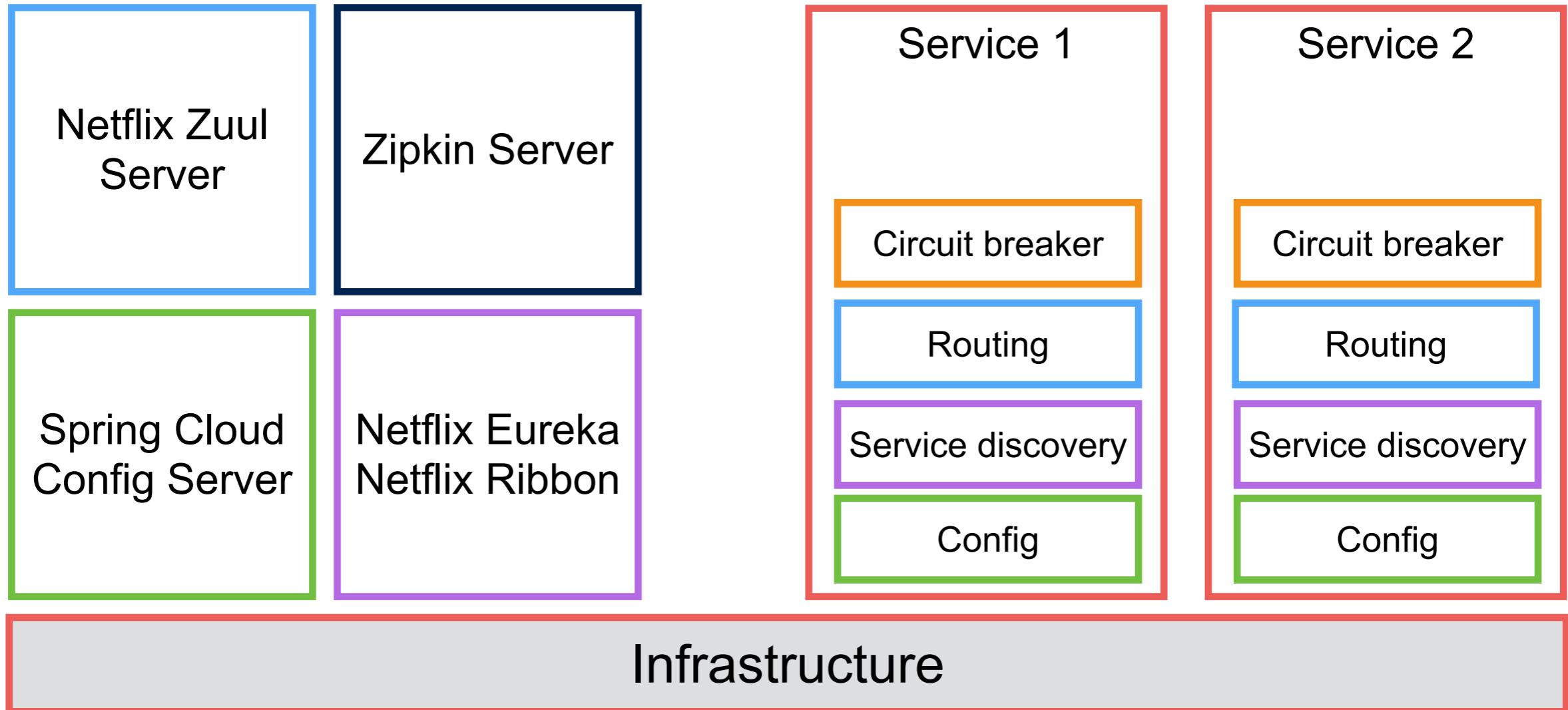
# Dynamic Routing



# Fault Tolerance



# Tracing and Visibility



# Microservice 1.0

JVM only  
Add libraries to your code/service

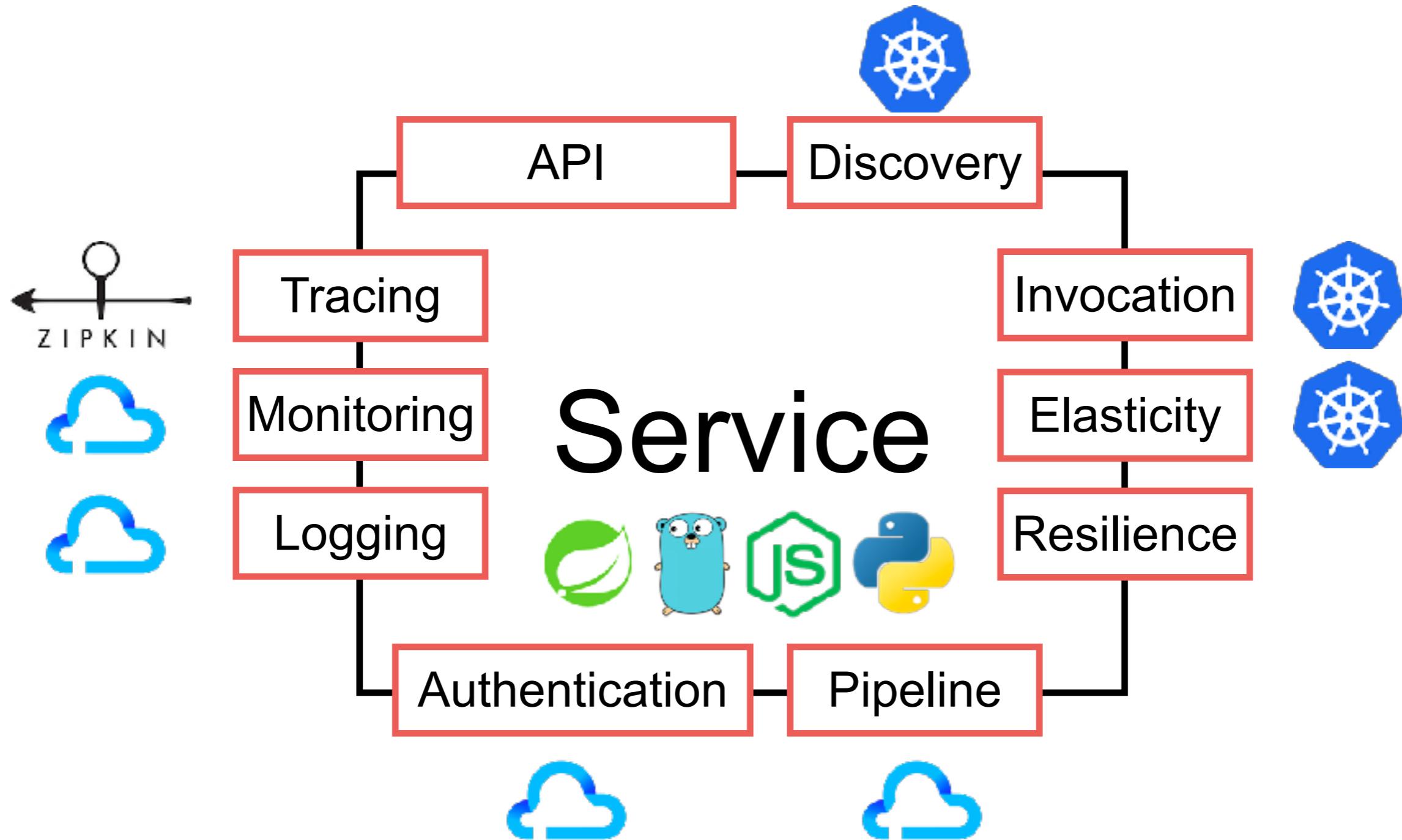


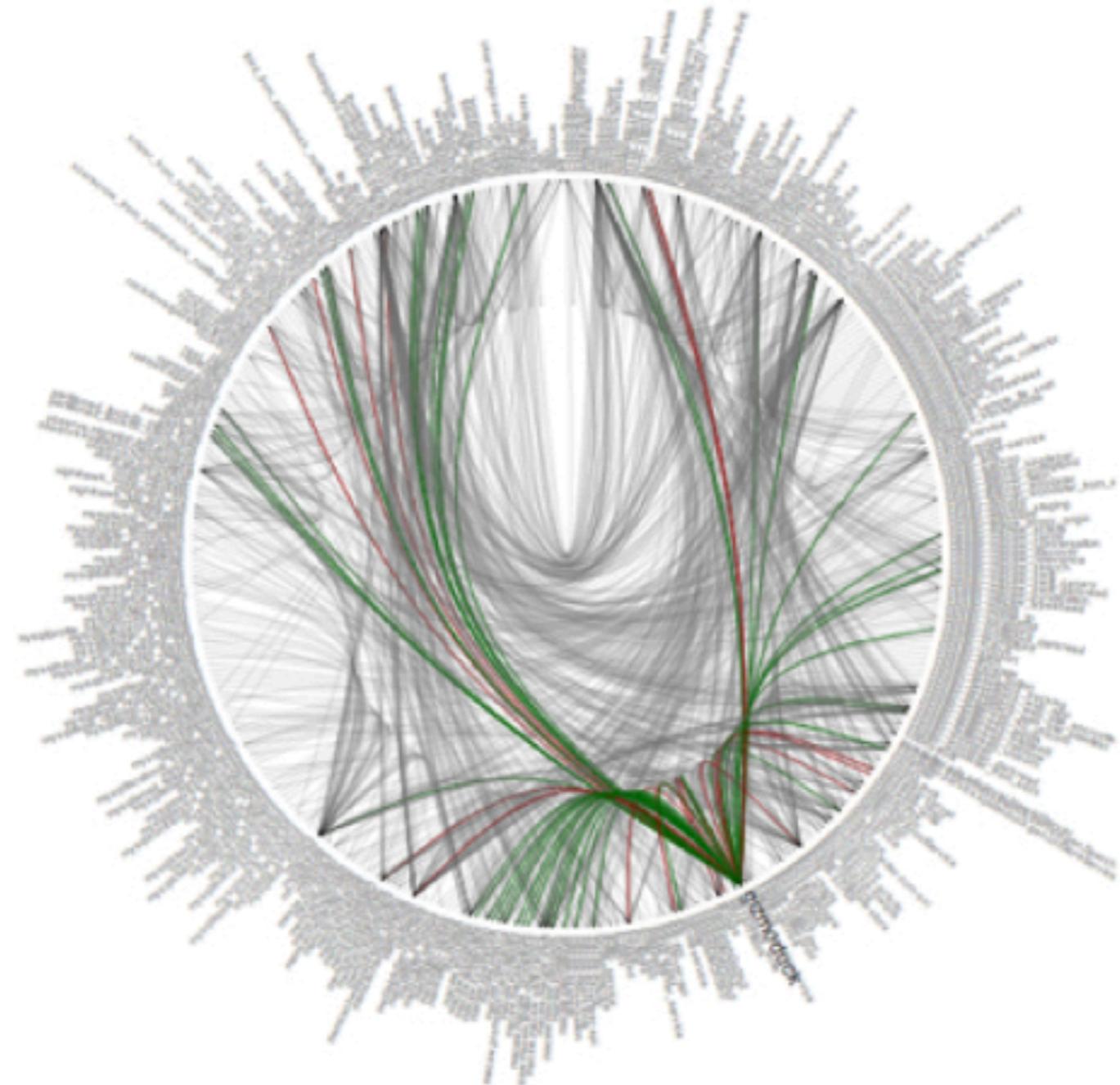


# kubernetes



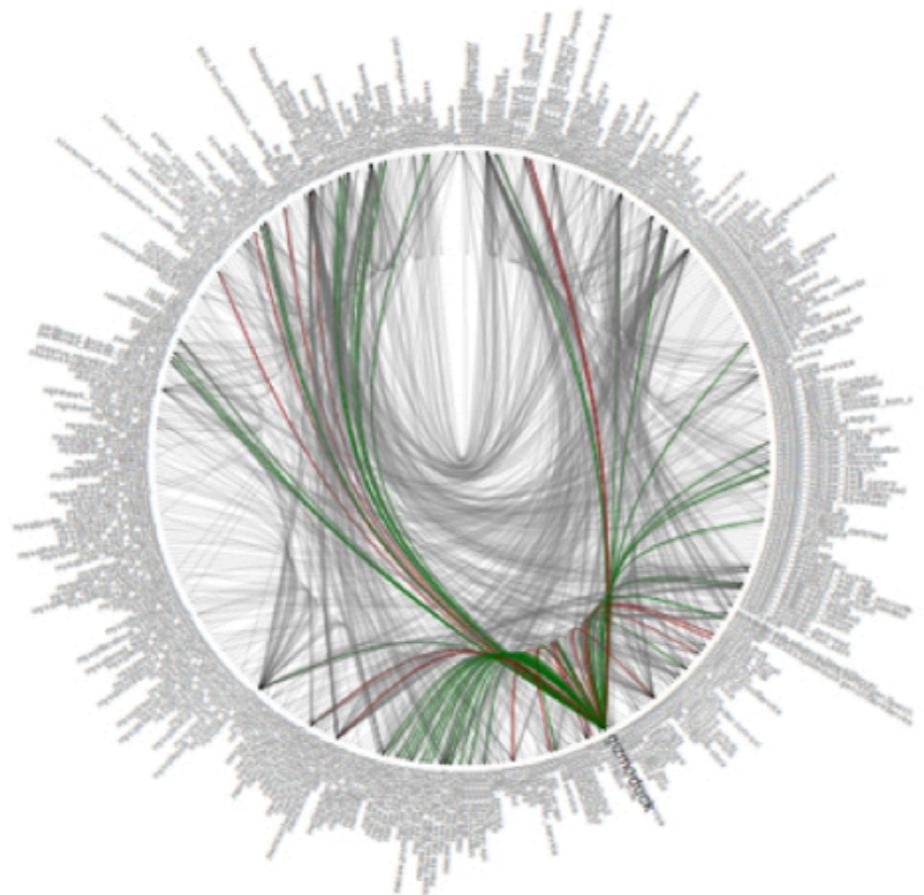
# Microservice 2.0





# Service Mesh

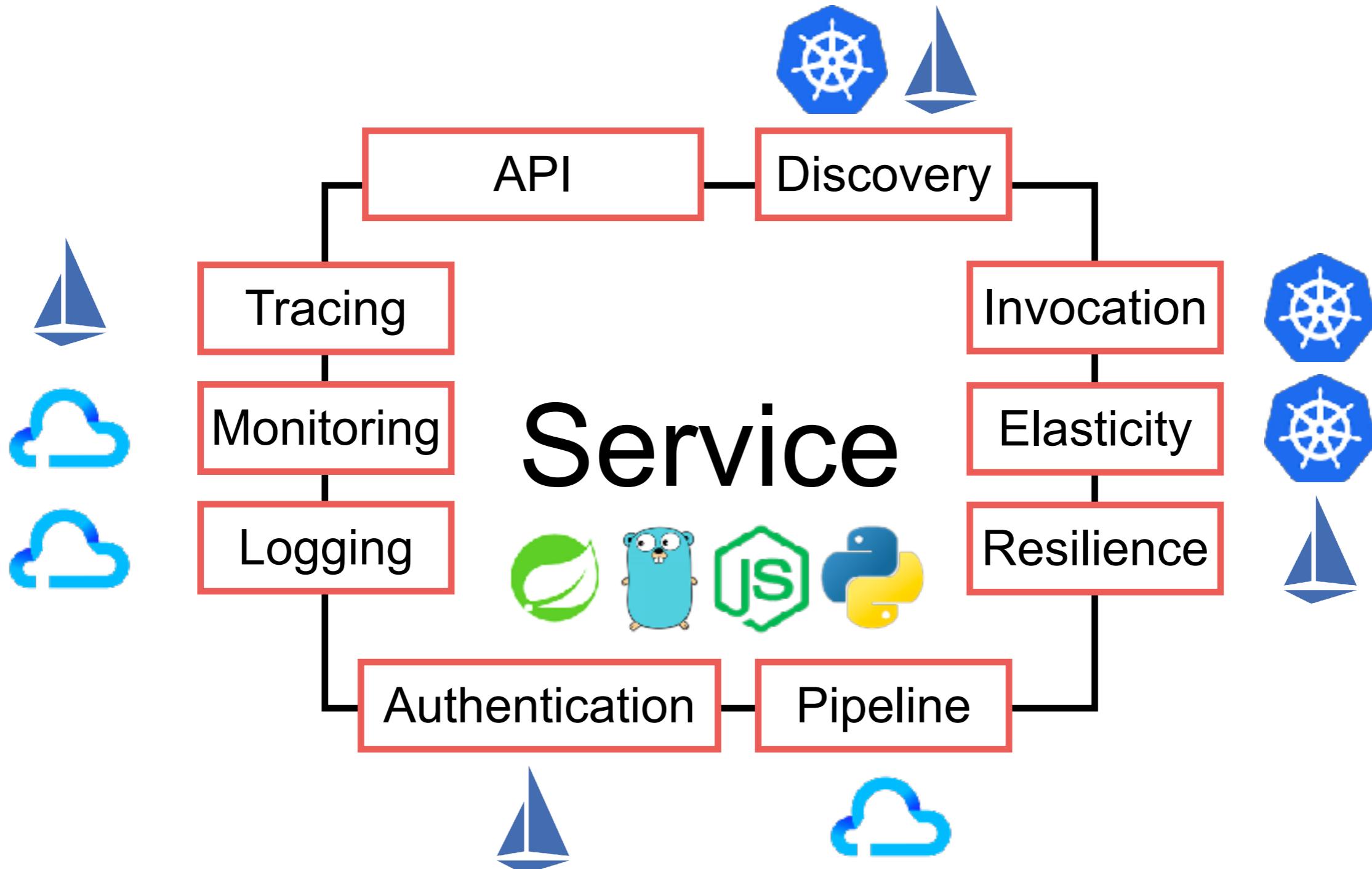




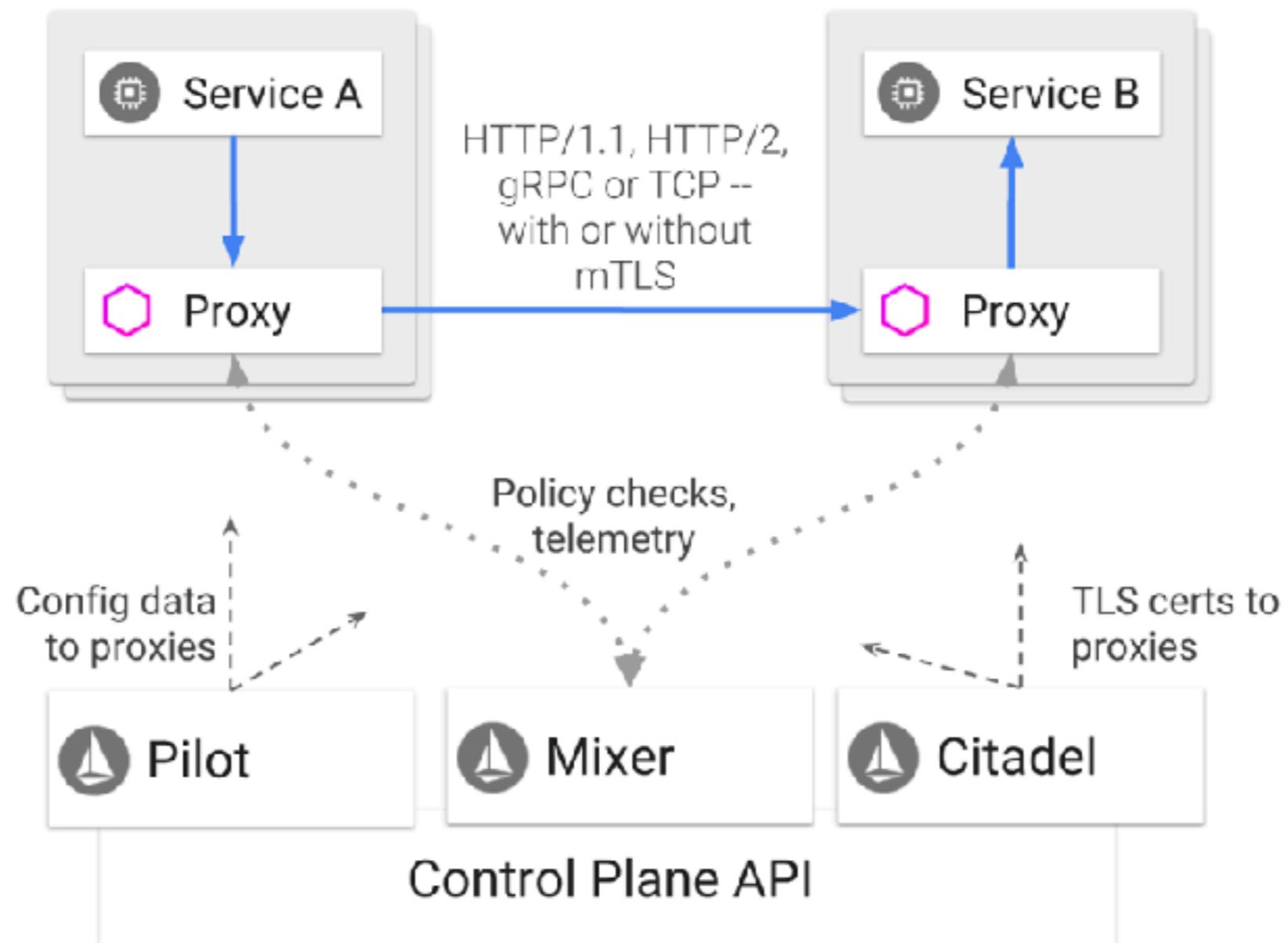
# Service Mesh with Istio



# Microservice 3.0



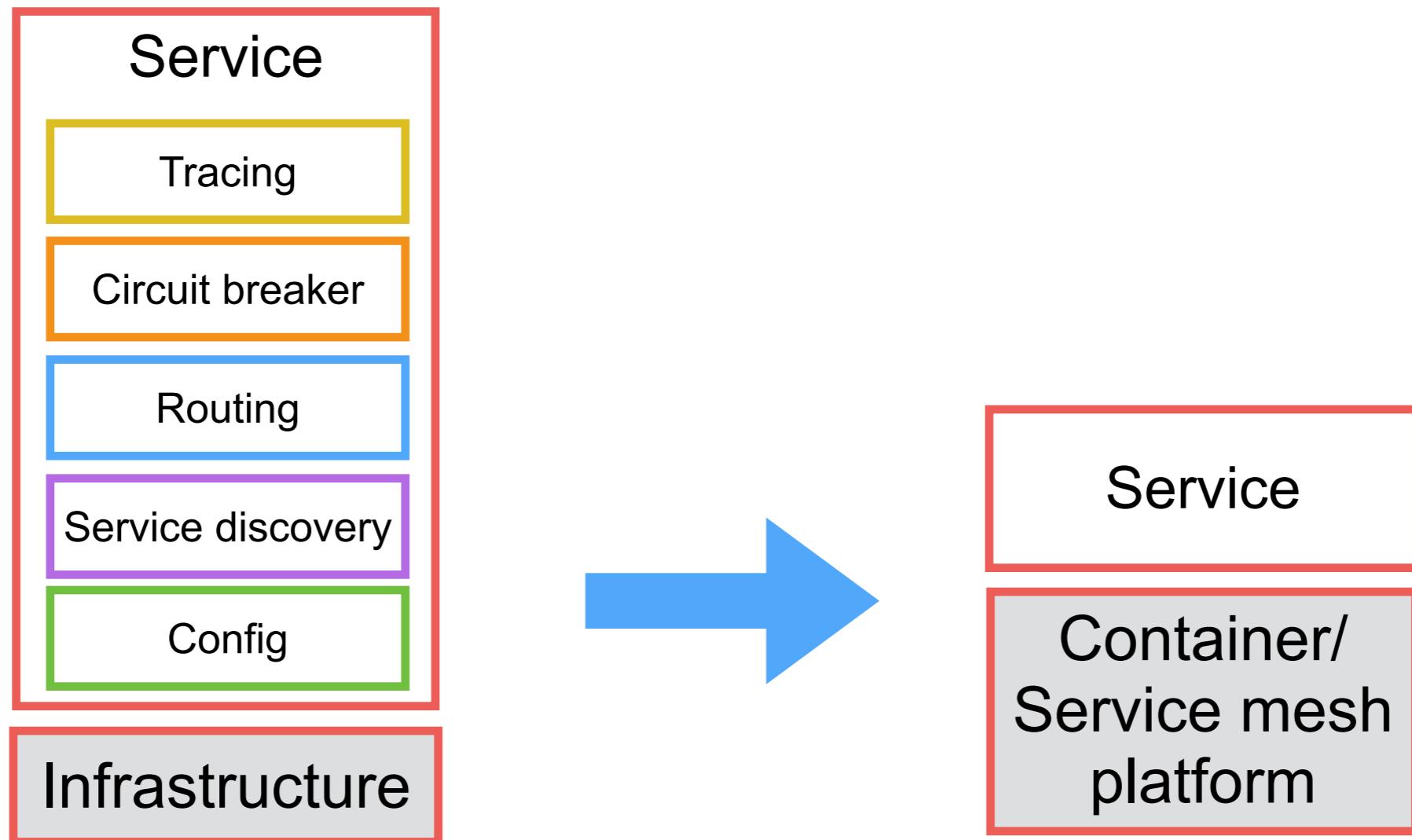
# Istio



<https://istio.io/docs/concepts/what-is-istio/>



# Microservice Evolution



# Function-as-a-Service (FaaS)



# Microservice 4.0 == FaaS



OPEN FAAS



APACHE  
OpenWhisk™



# **Workshop**

# **Microservice 1.5**



# Twelve-Factor App



# Twelve-Factor App

<https://12factor.net/>

Initial to build app for Heroku

Principles to cloud and container native app



# 1. Codebase

Codebase must be tracked in version control  
and will have many deploy



# 2. Dependencies

Dependencies are explicitly declared and isolated

Use dependency management tools to shared  
libraries



# 3. Configuration

Store configuration in the environment

Add configuration in environment variables or  
config files



# 4. Backing services

Treat backing services as an attached resource

Should easy to deploy and change



# 5. Build, Release, Run

Always have a build and deploy strategy

Build strategies for repeated builds, versioning of running system and rollback



# 6. Processes

Execute the application as a stateless process



# 7. Port Binding

Expose services via port bindings



# 8. Concurrency

Scale out with the process model



# 9. Disposability

Quick application startup and shutdown times  
Graceful shutdown



# 10. Dev/prod parity

Application is treated the same way in dev, staging  
and production

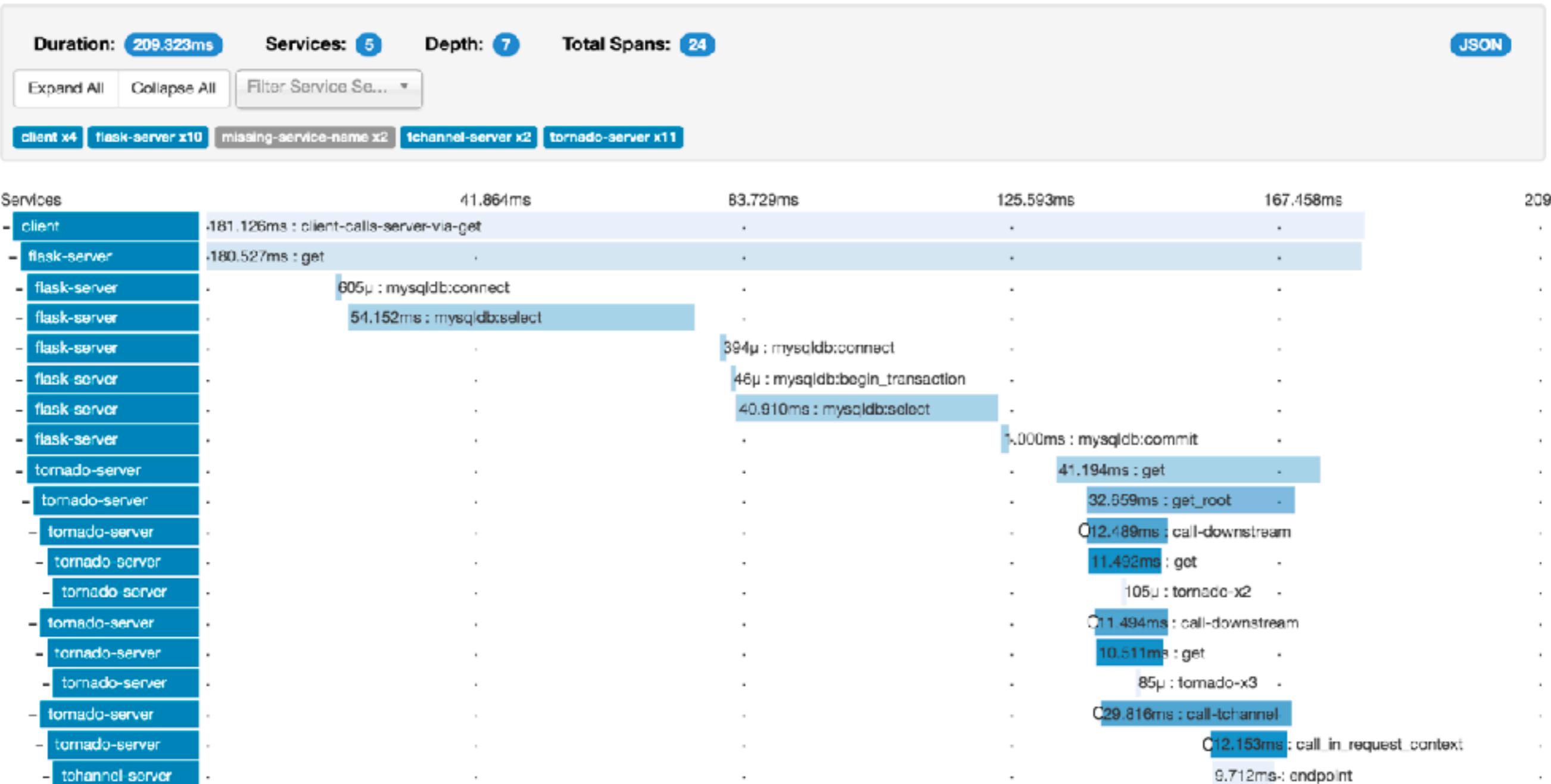


# 11. Log management

Treated as an event stream



# Logging/Tracing



<https://zipkin.io/>



# 12. Admin tasks

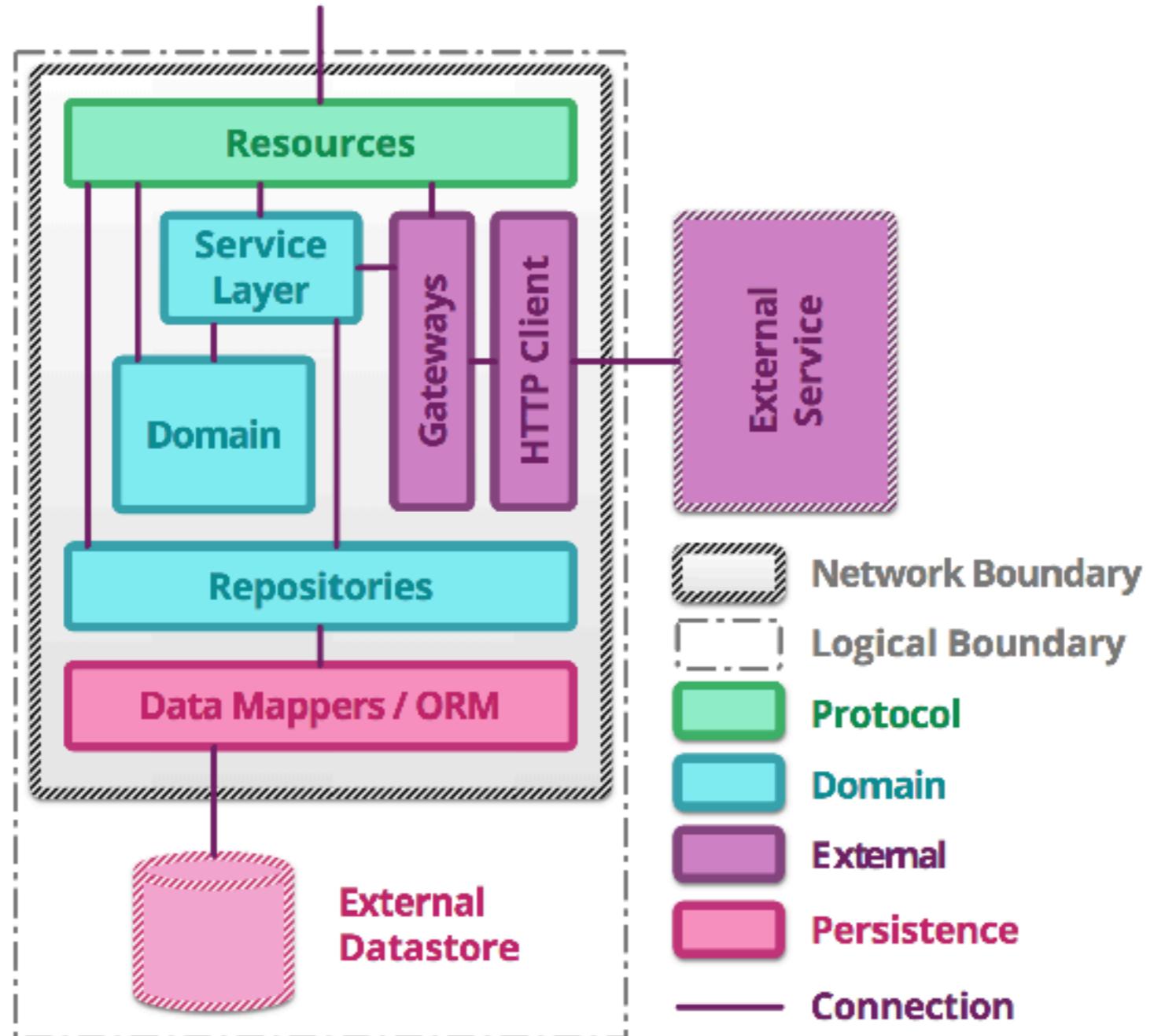
Treated the same way like the rest of the application



# Microservice Testing



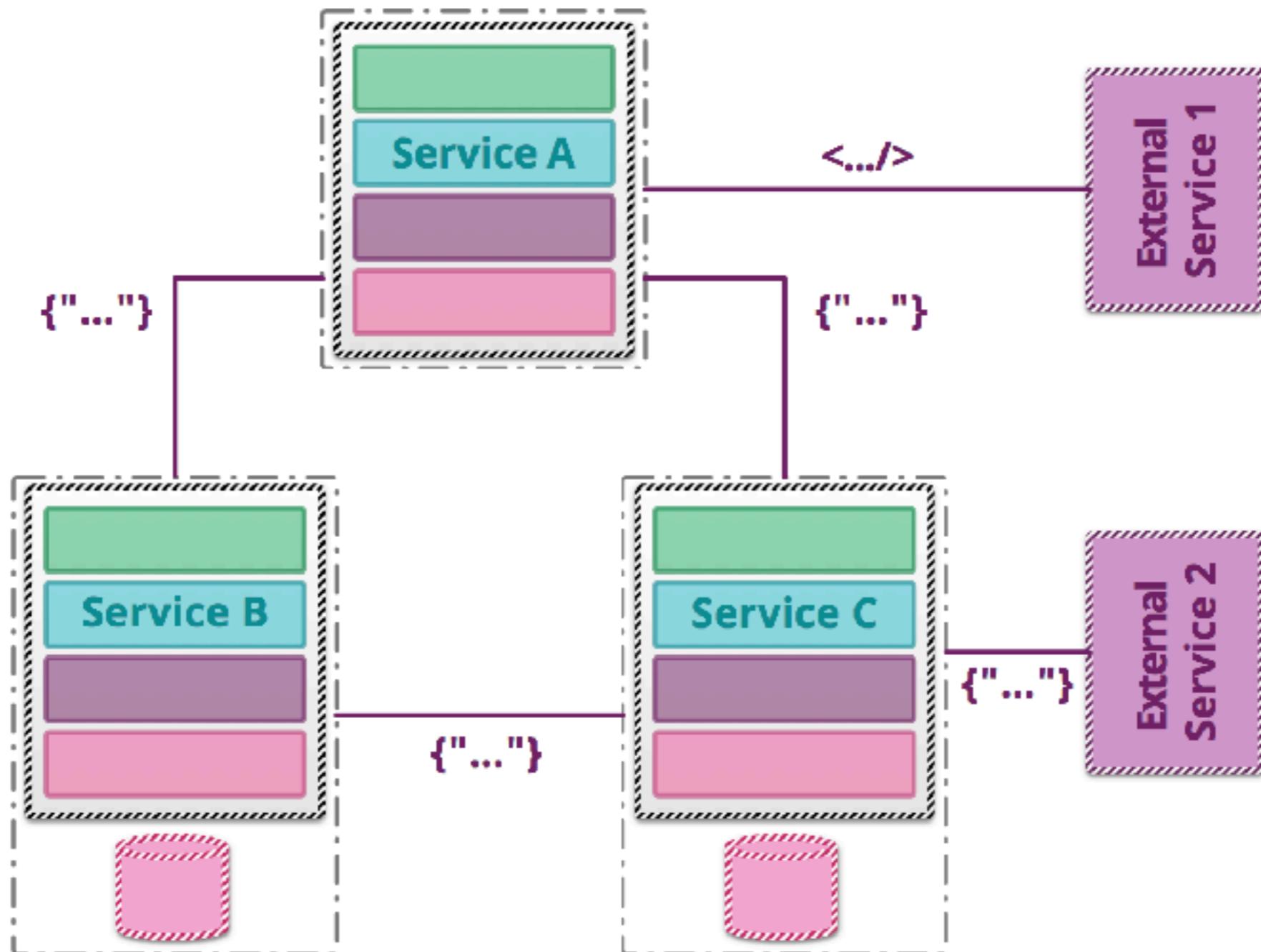
# Service structure



<https://martinfowler.com/articles/microservice-testing>



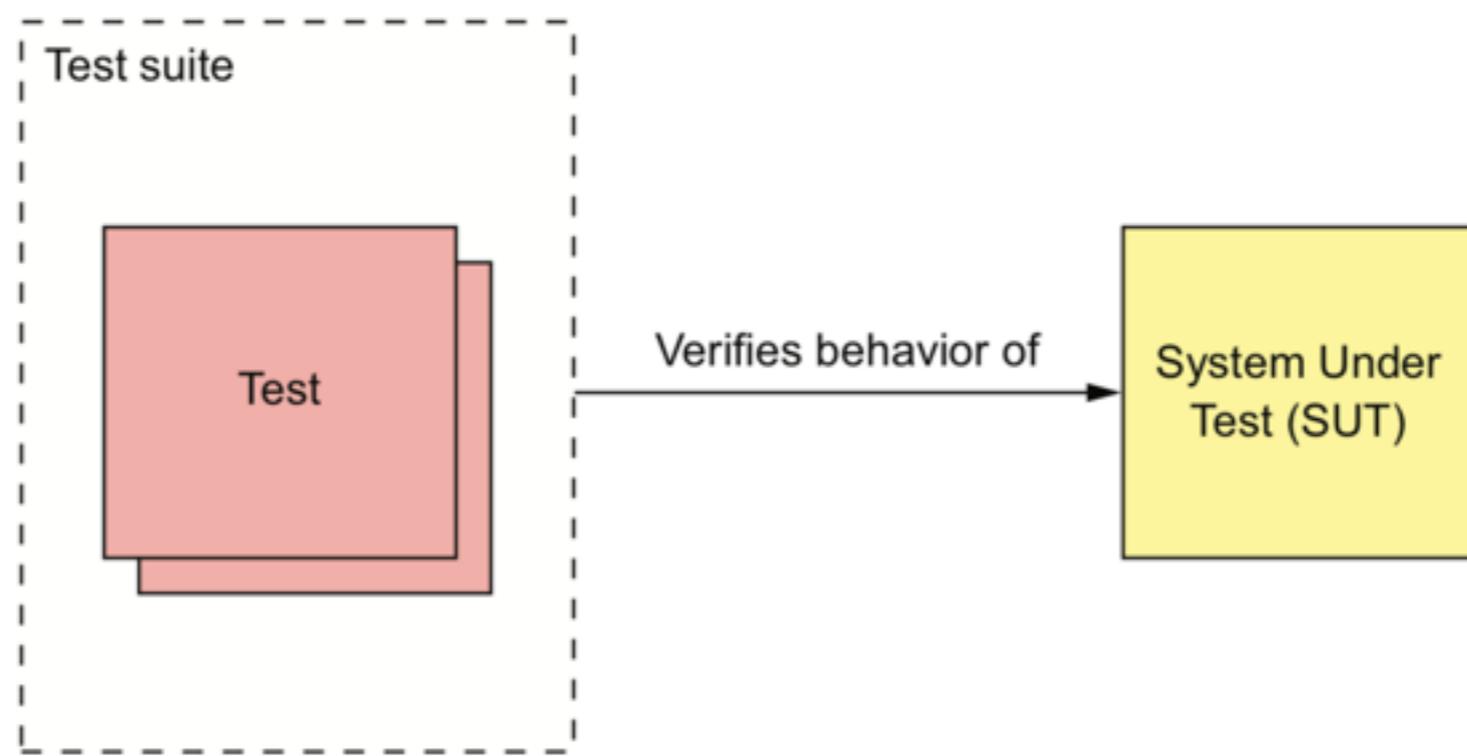
# Multiple services



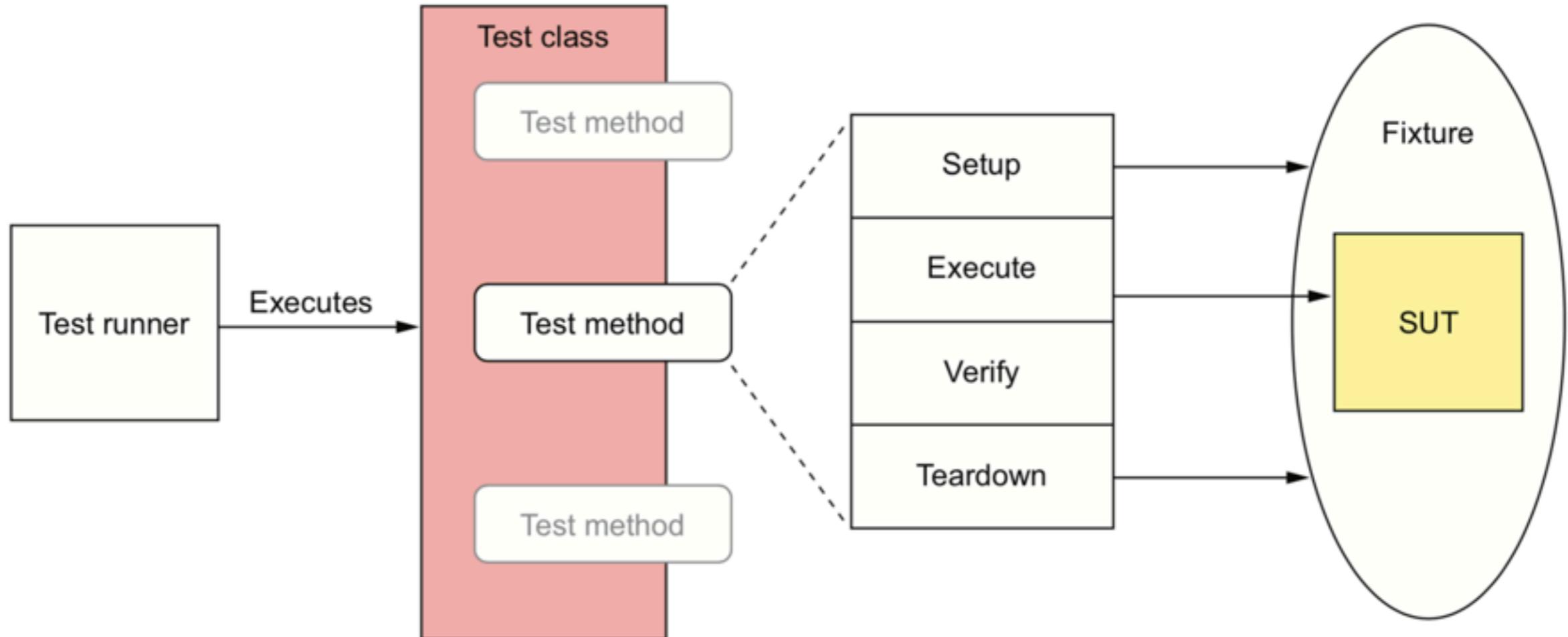
<https://martinfowler.com/articles/microservice-testing>



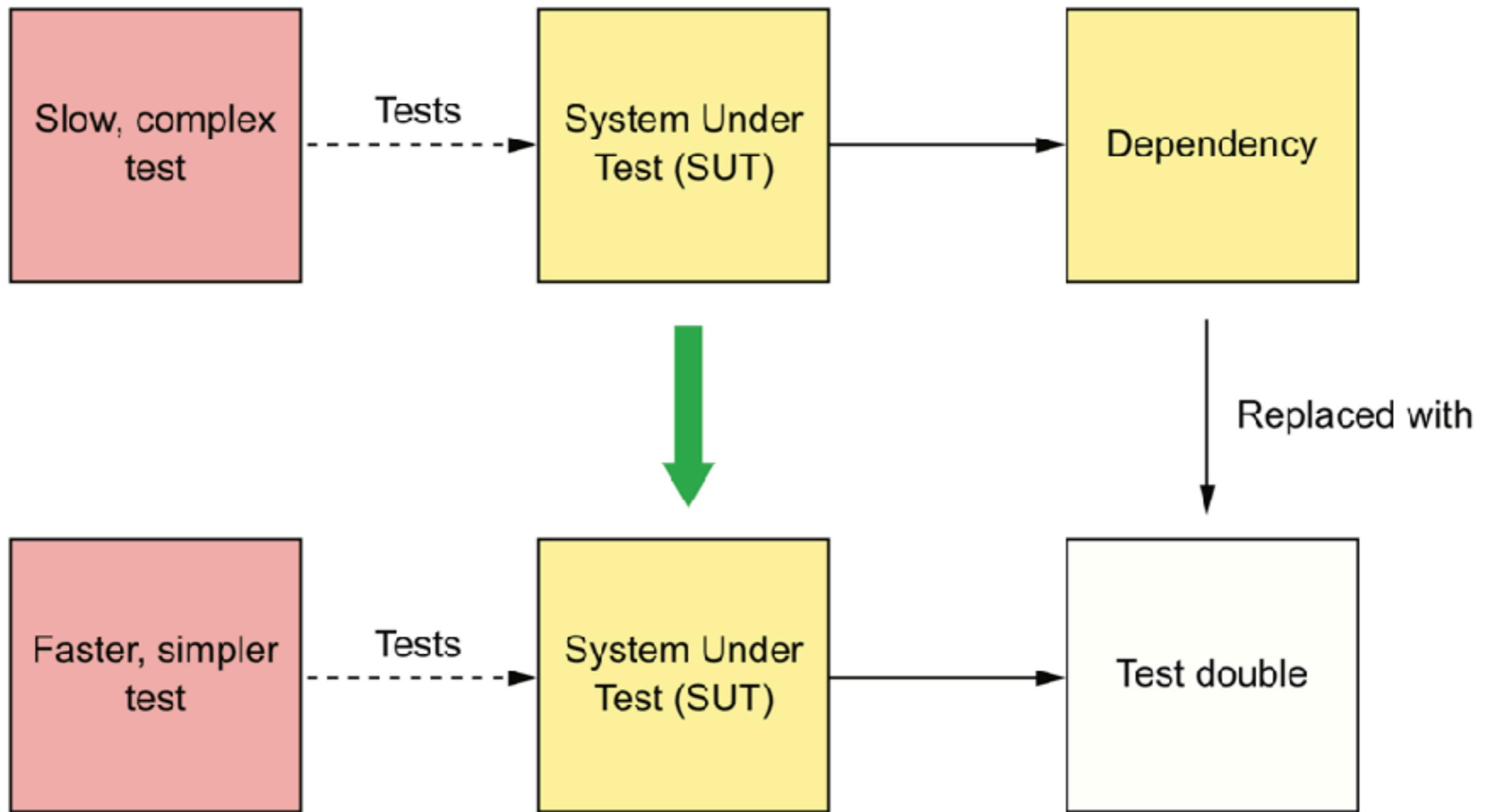
# Testing

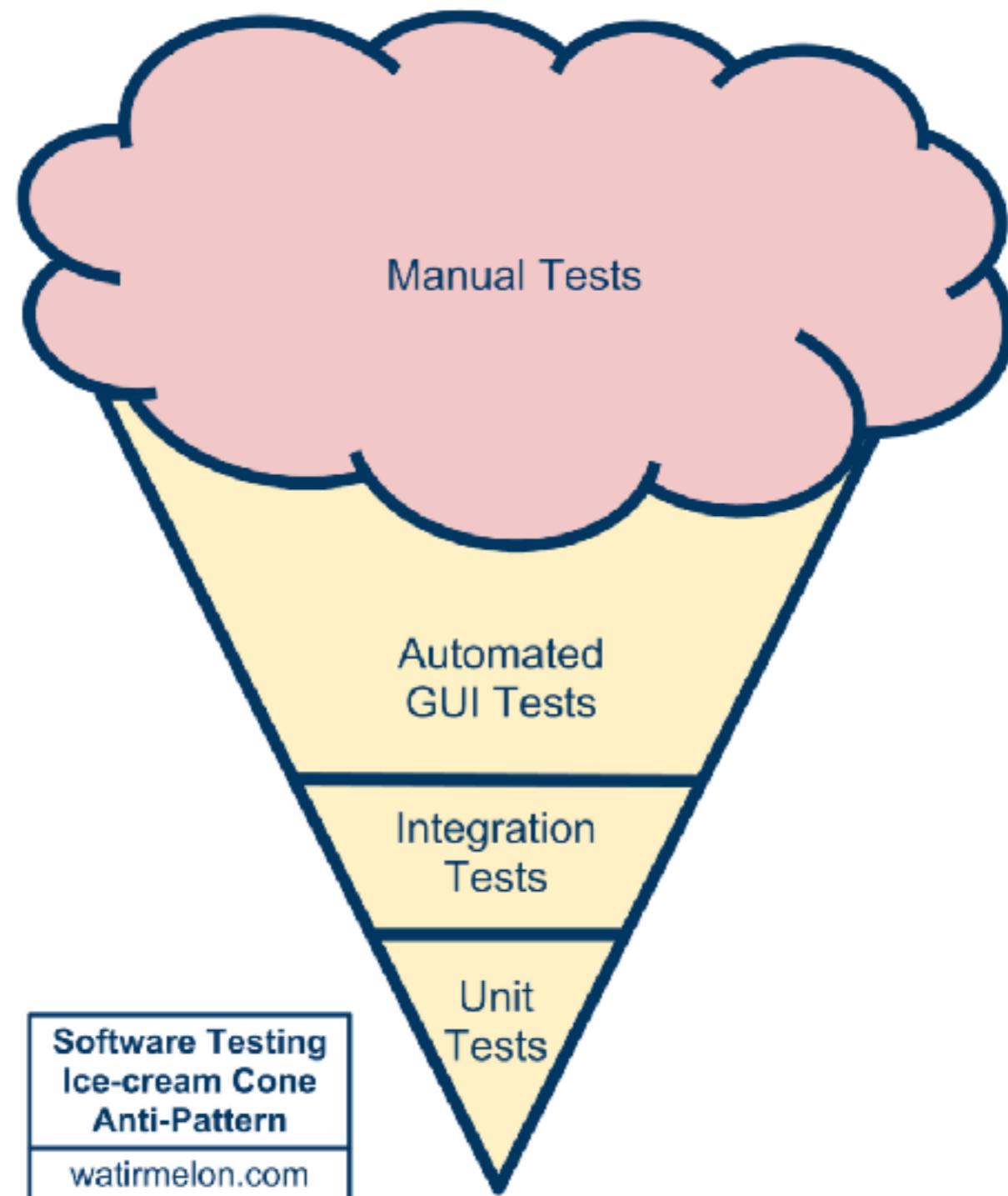


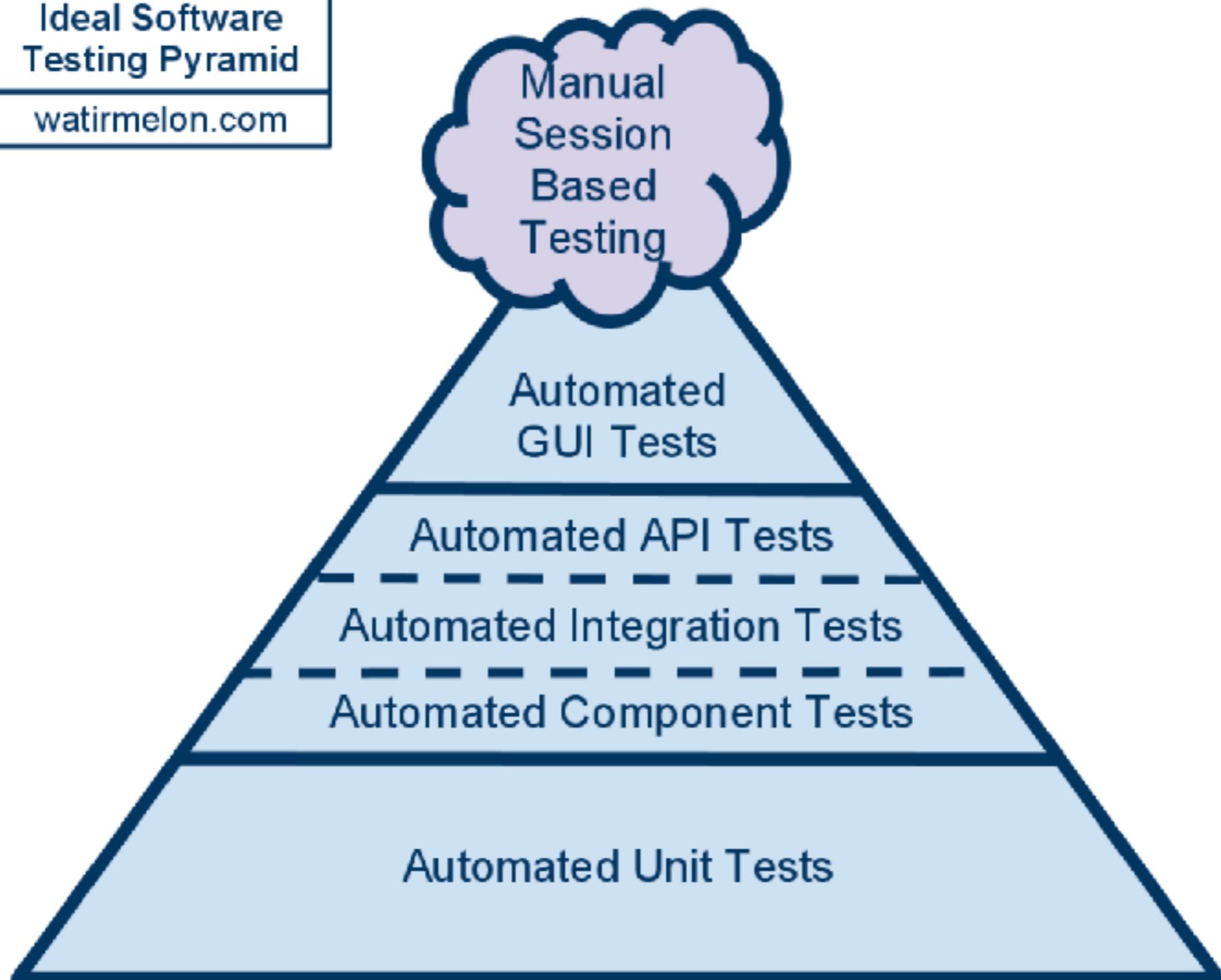
# Structure of Automated tests

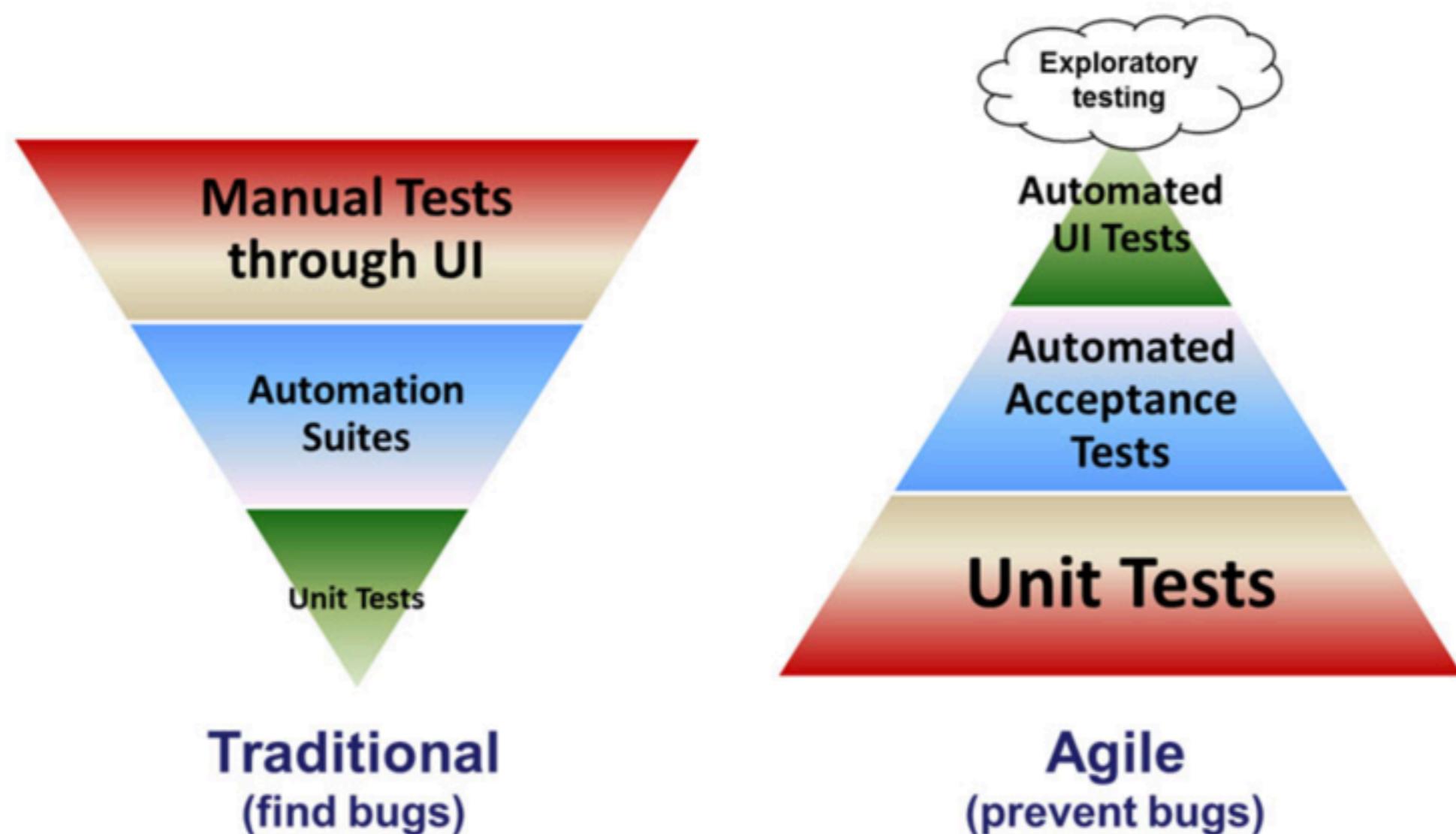


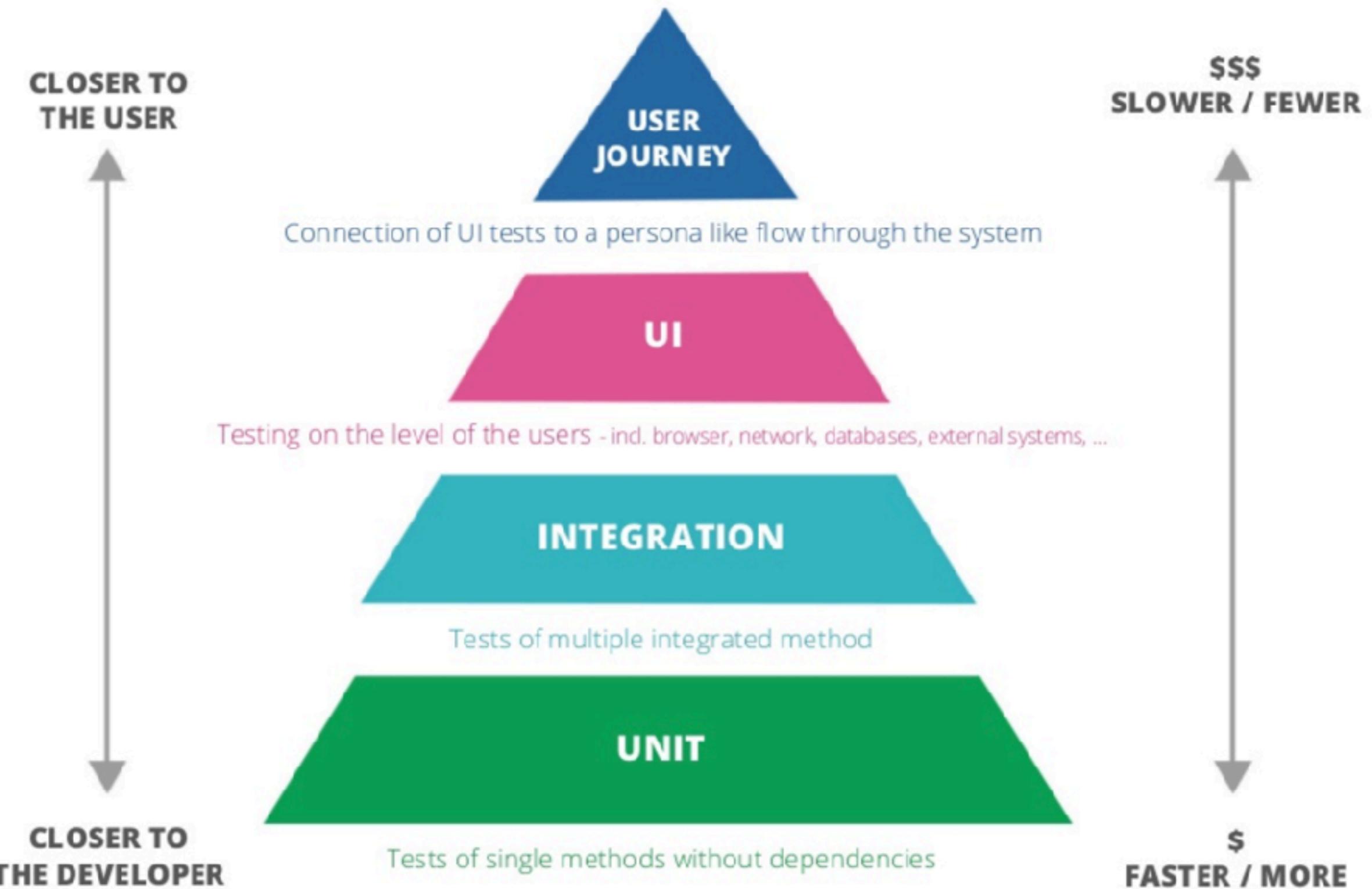
# Working with Test double



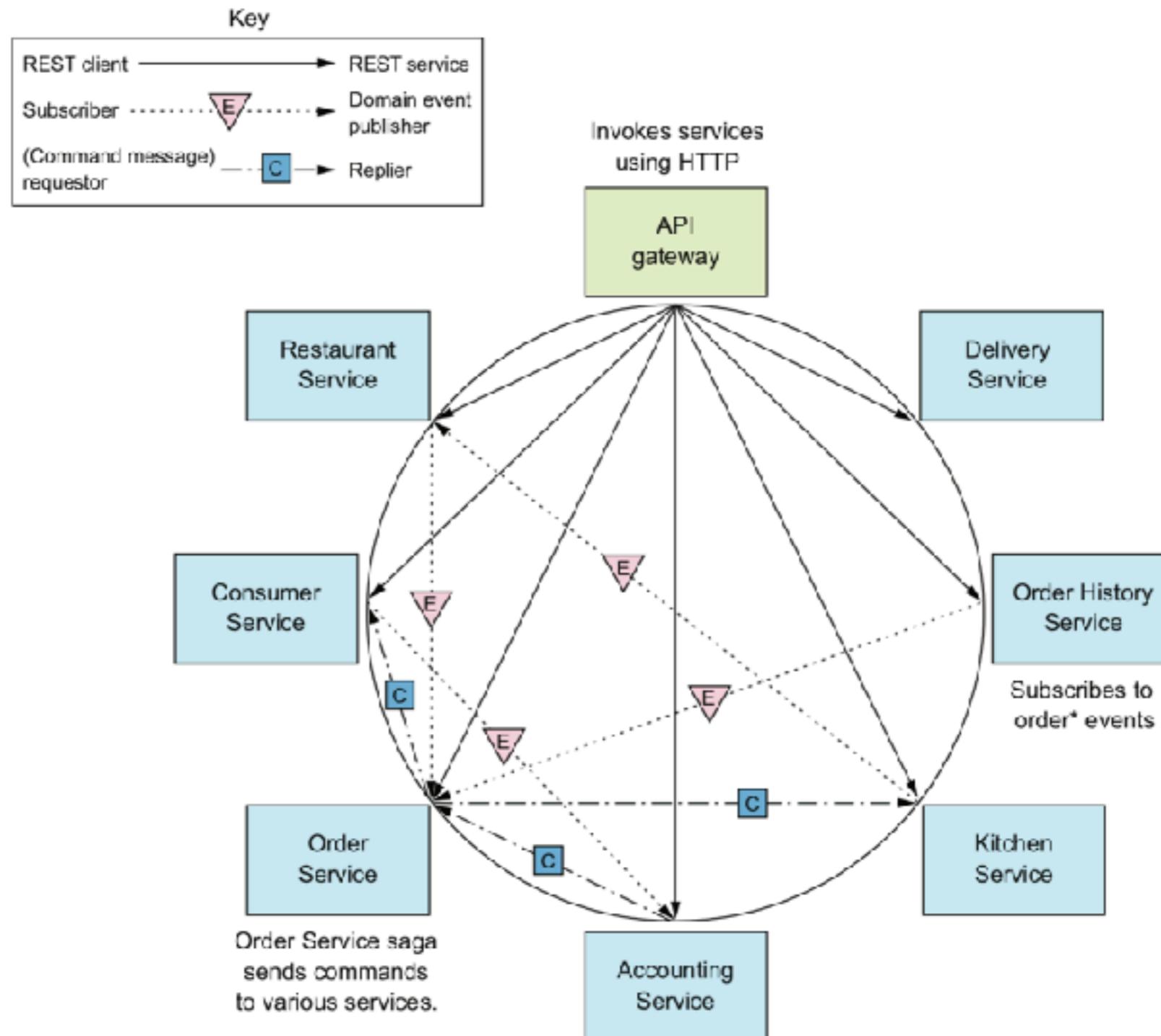






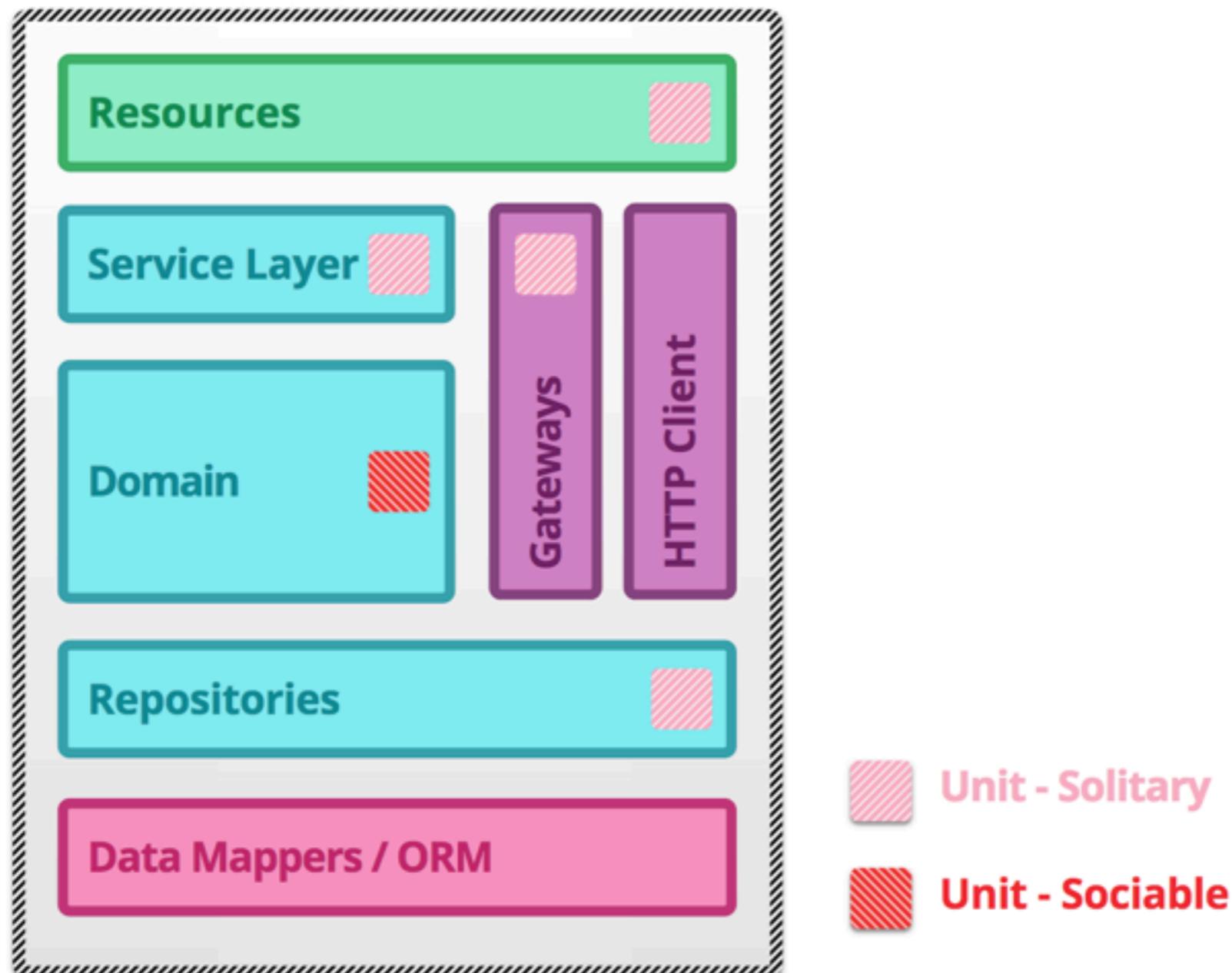


# Testing Microservices ?

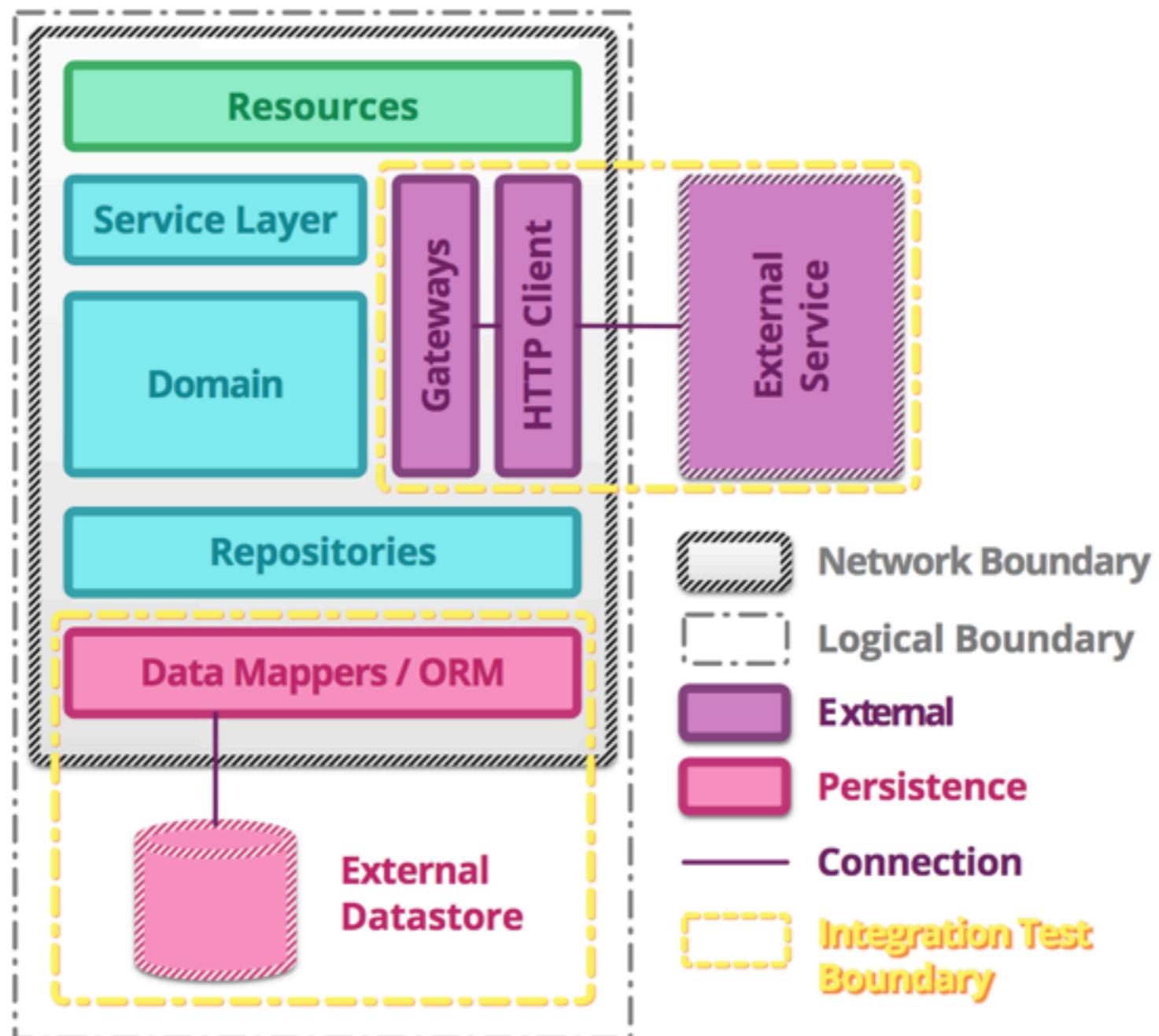




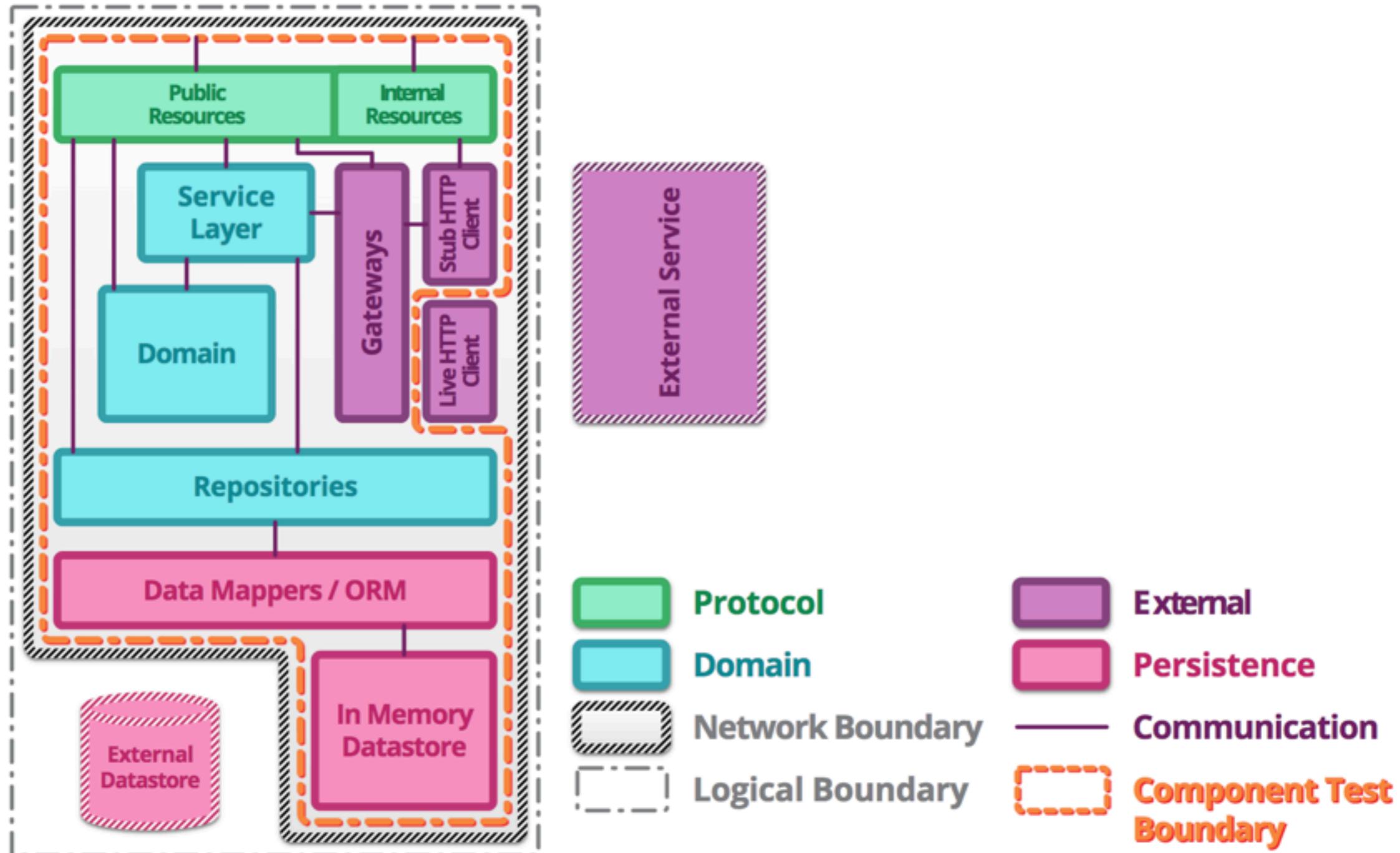
# Unit testing



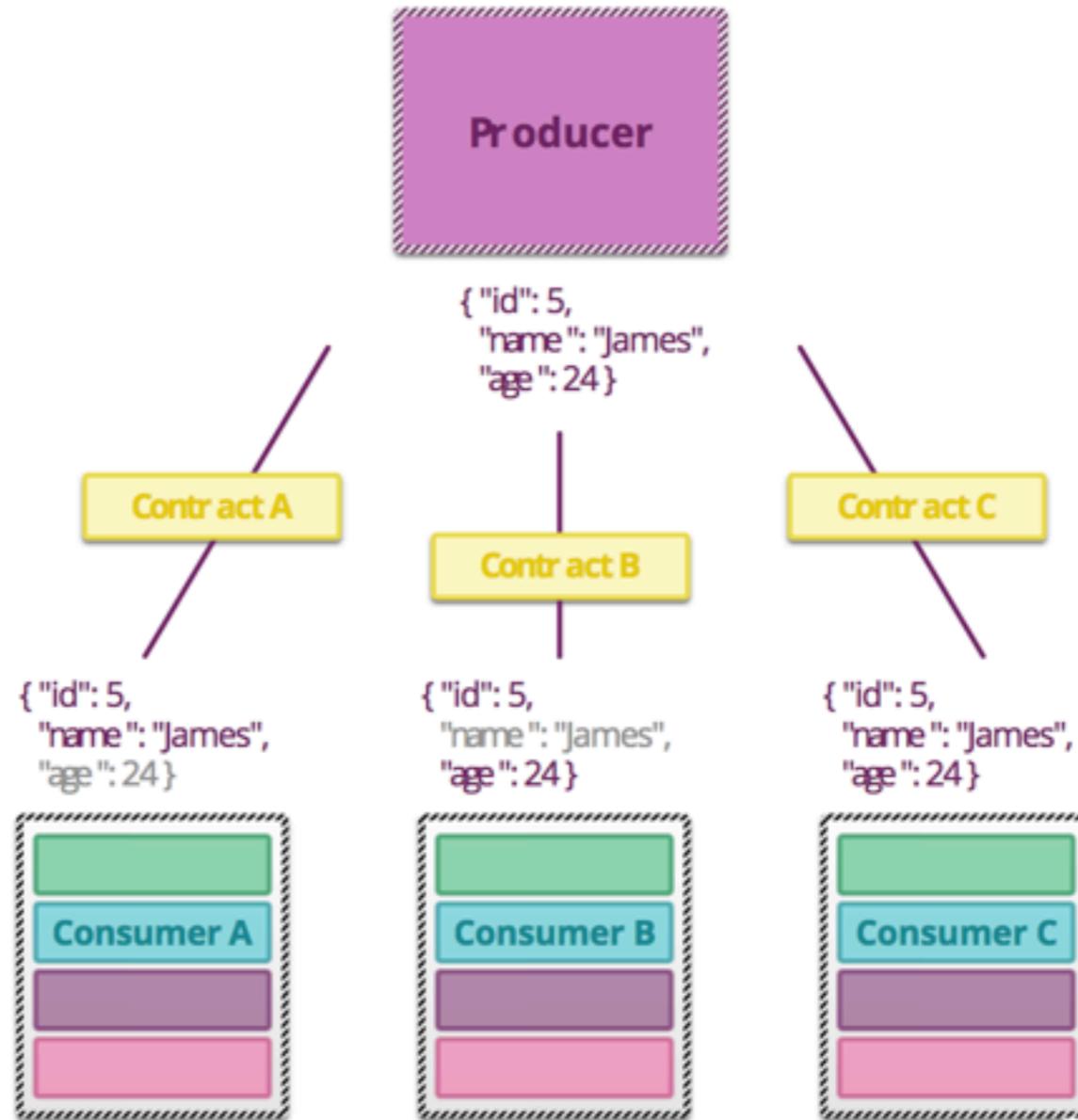
# Integration testing



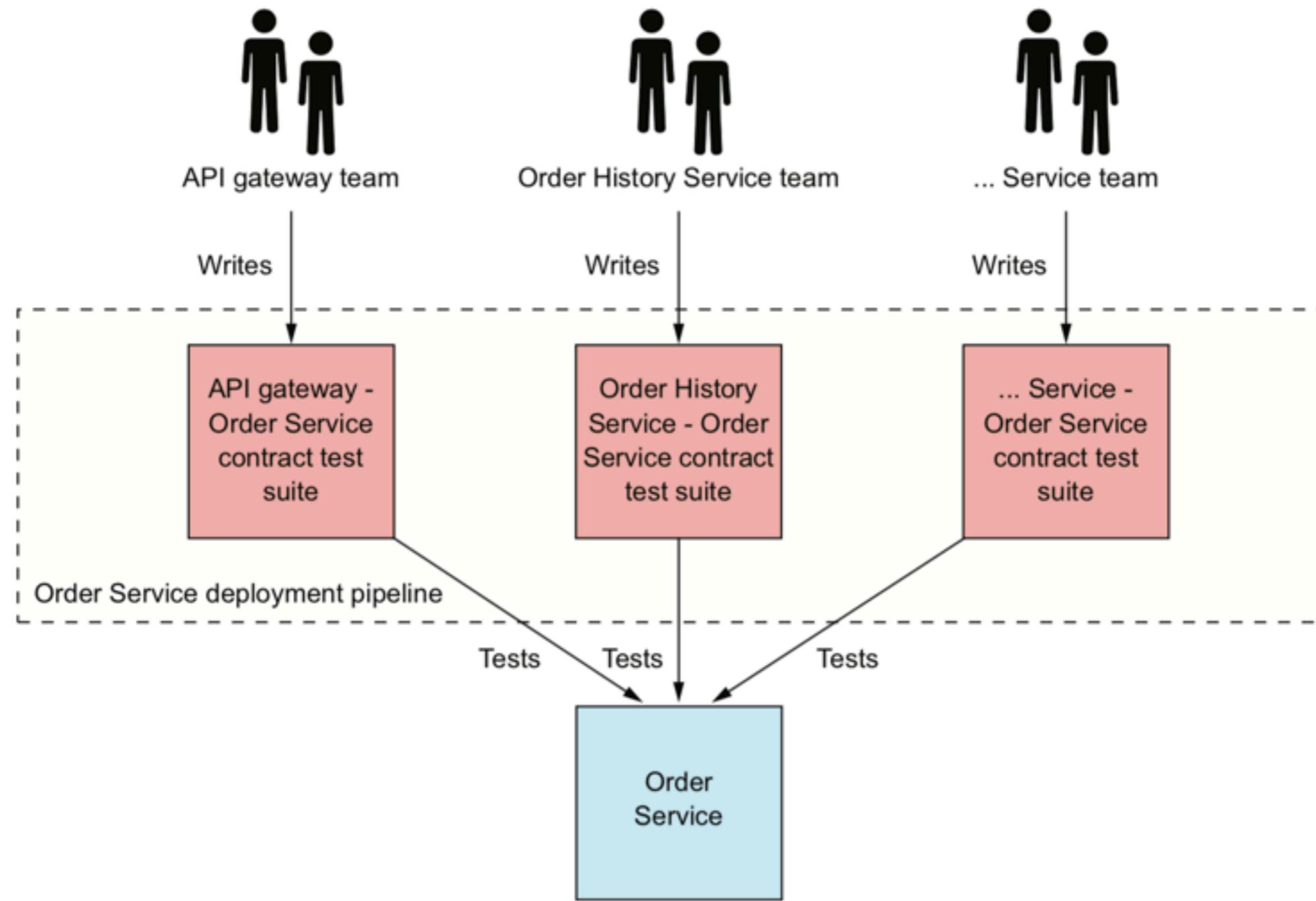
# Component testing



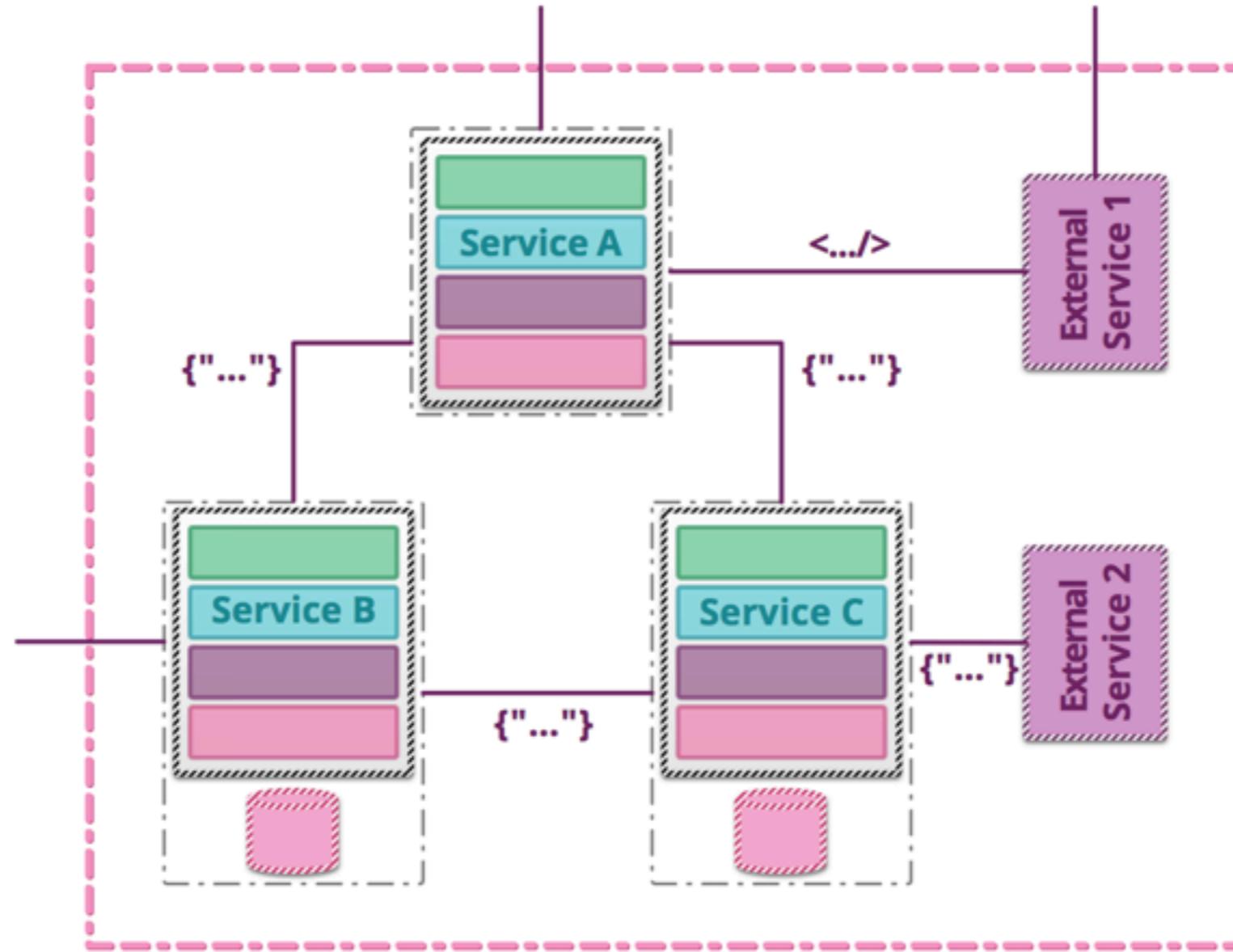
# Contract testing



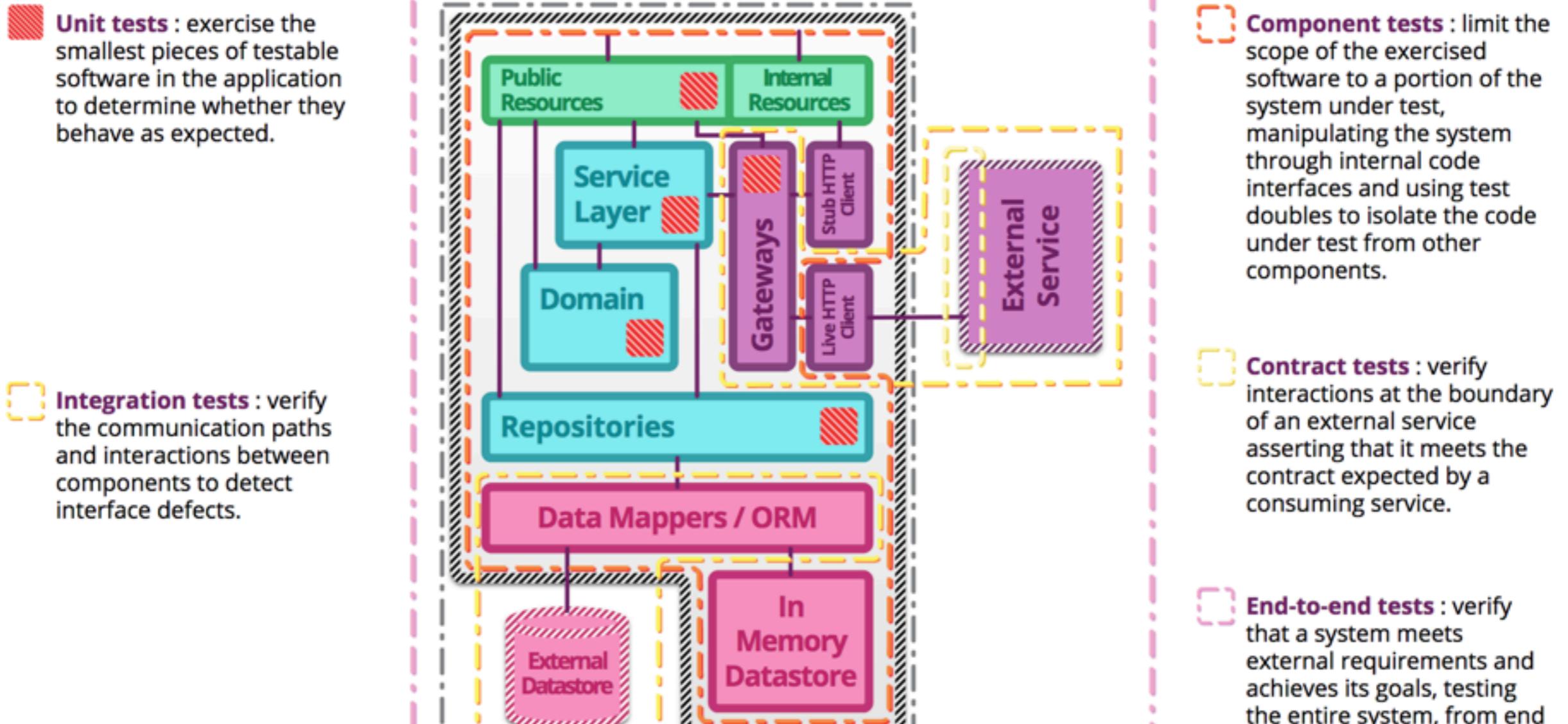
# Consumer Contract testing



# End-to-End testing



# Summary



**Unit tests** : exercise the smallest pieces of testable software in the application to determine whether they behave as expected.

**Integration tests** : verify the communication paths and interactions between components to detect interface defects.

**Component tests** : limit the scope of the exercised software to a portion of the system under test, manipulating the system through internal code interfaces and using test doubles to isolate the code under test from other components.

**Contract tests** : verify interactions at the boundary of an external service asserting that it meets the contract expected by a consuming service.

**End-to-end tests** : verify that a system meets external requirements and achieves its goals, testing the entire system, from end to end.



# What is your testing strategy ?



# **Performance testing ?**

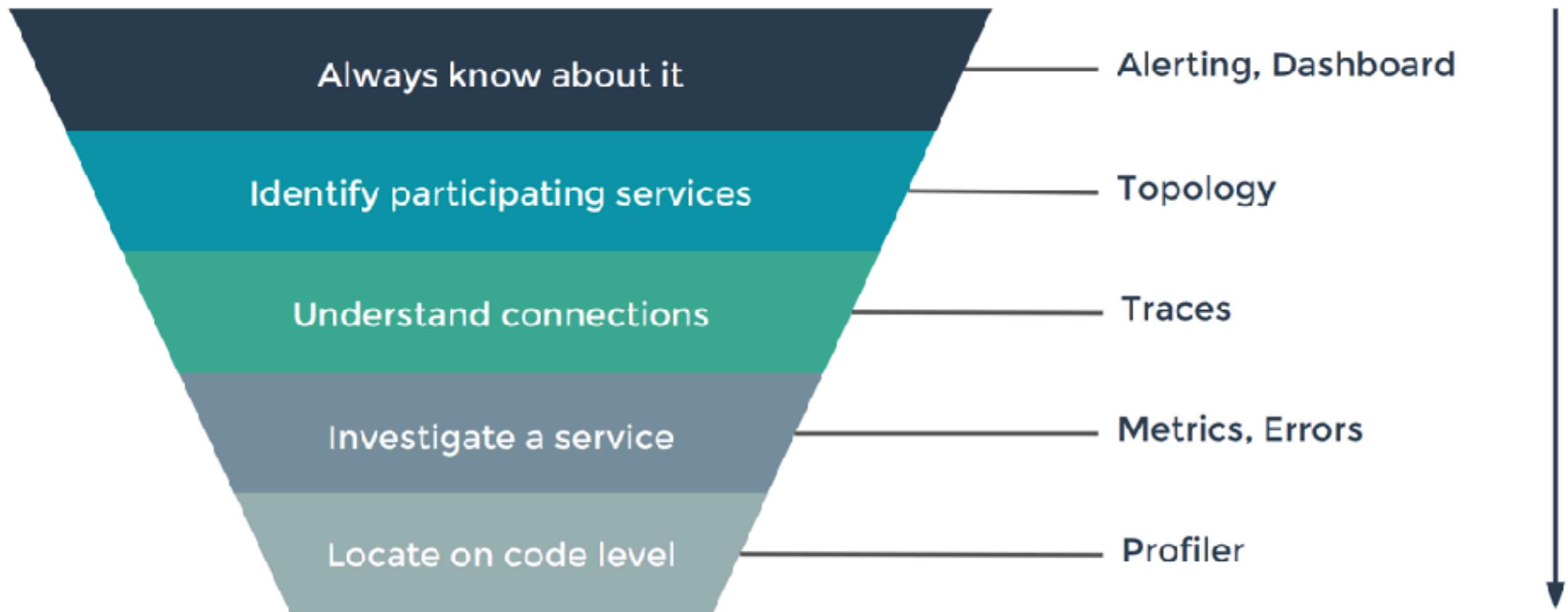
# **Security testing ?**



# How to find an issue ?



# How to find an issue ?



# Develop production-ready services



# Important quality attributes

Security  
Configurability  
Observability



# Secure services



# Develop secure services

Authentication

Authorization

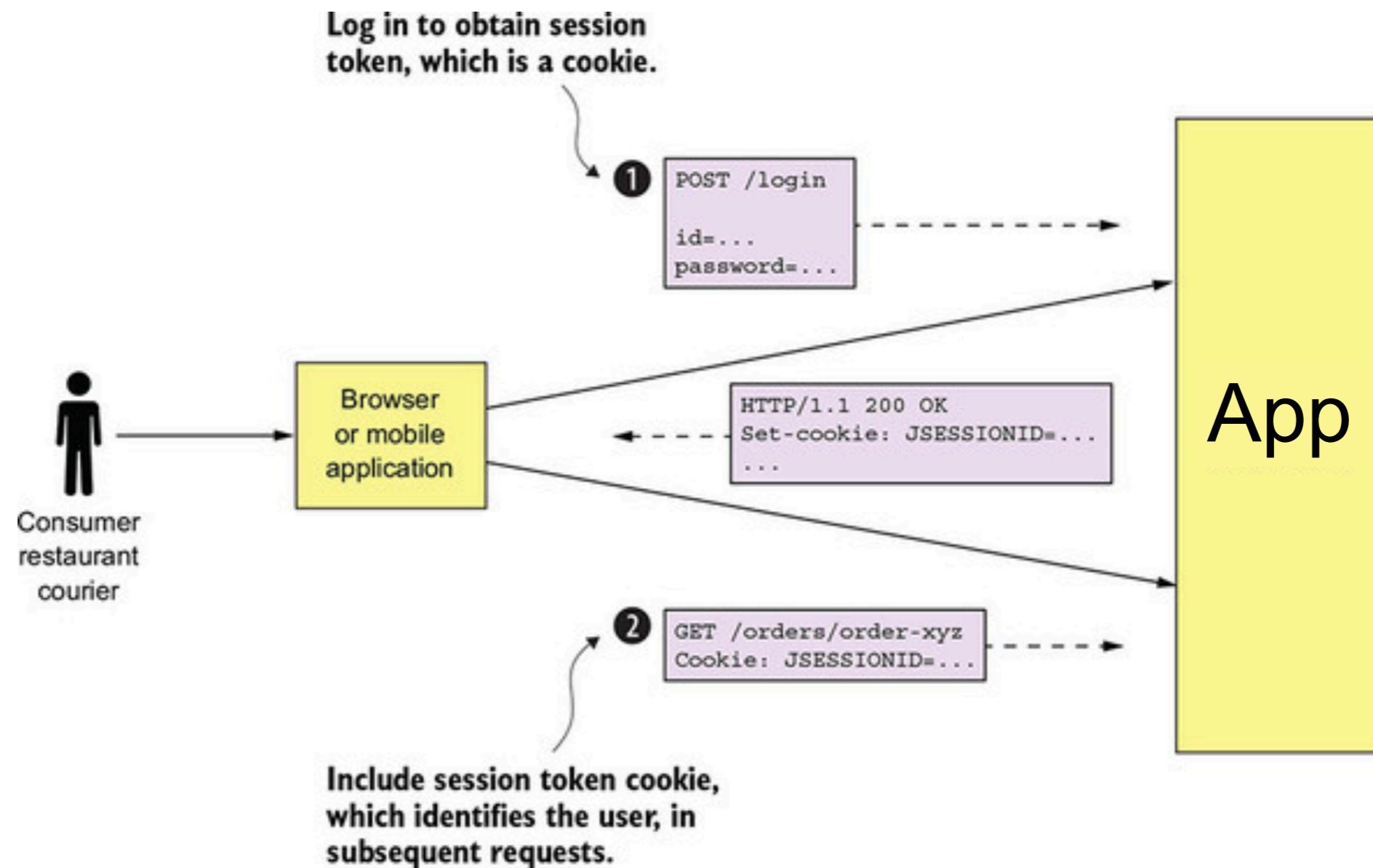
Auditing

Secure interprocess communication



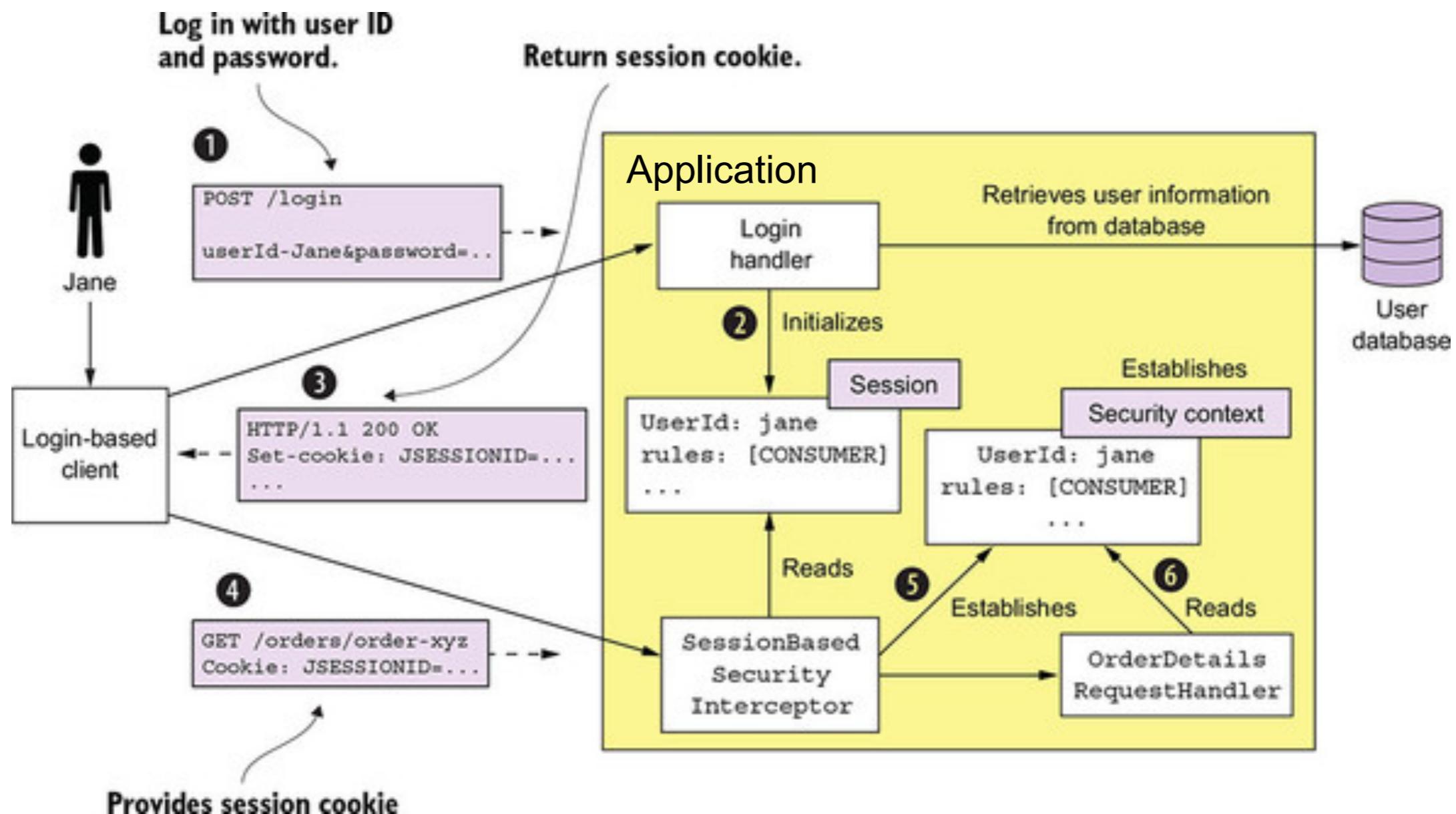
# Security in traditional application

Keep security information in browser's cookie



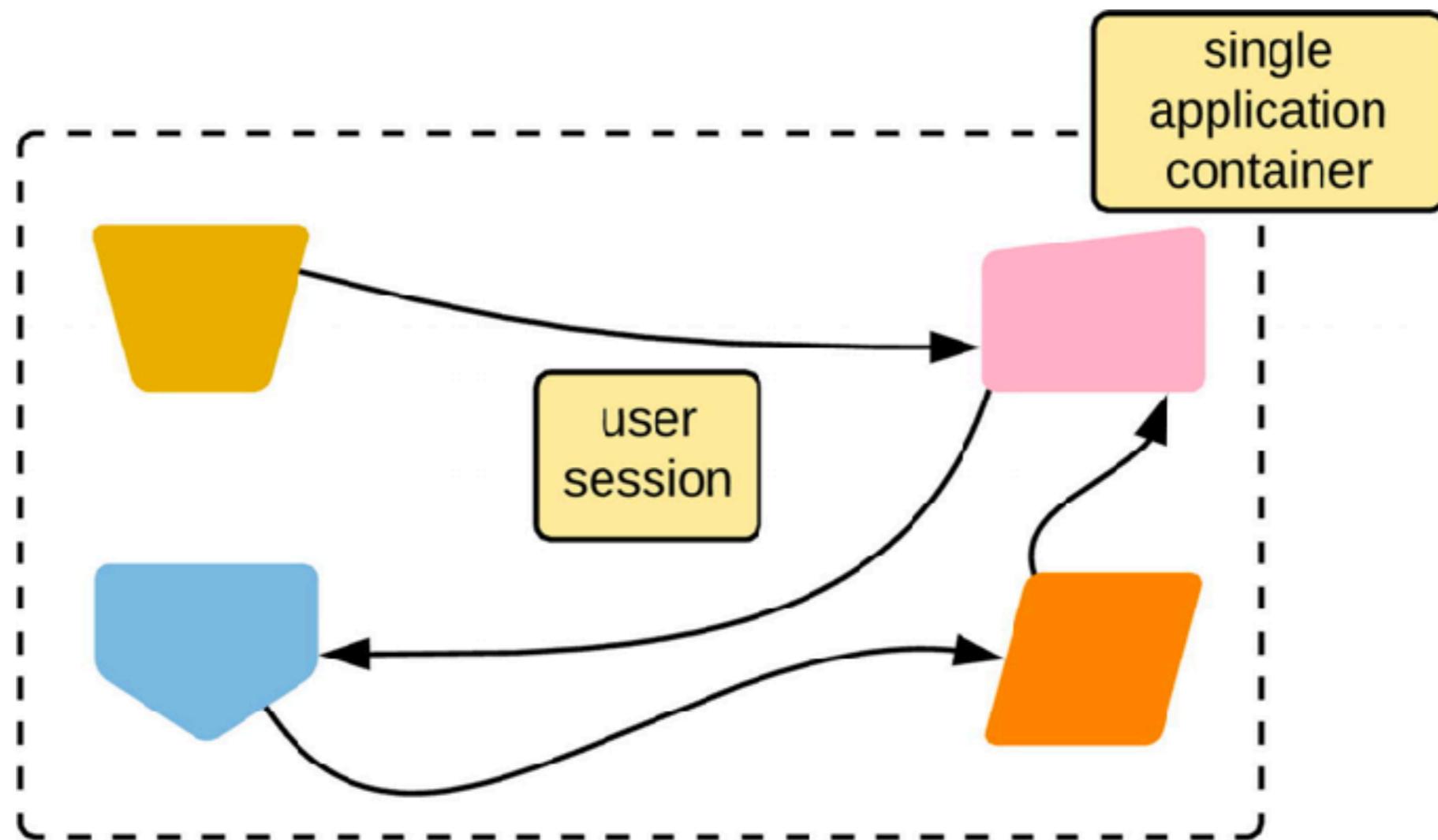
# Security in traditional application

## Implement security process in application



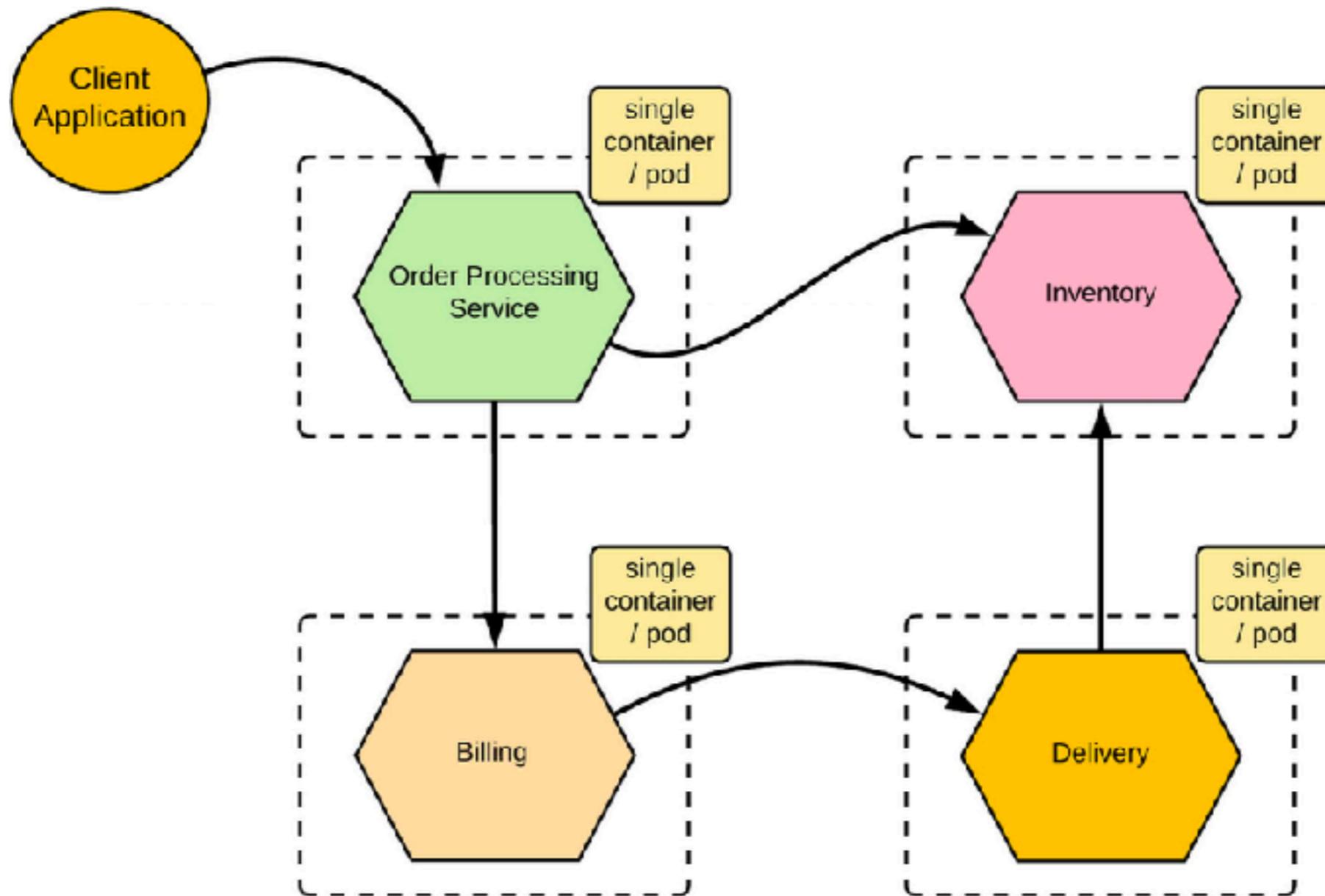
# Security in traditional application

## Sharing a user session in application



# Security in multiple services ?

## Interaction between multiple services



# Secure service-to-service communication

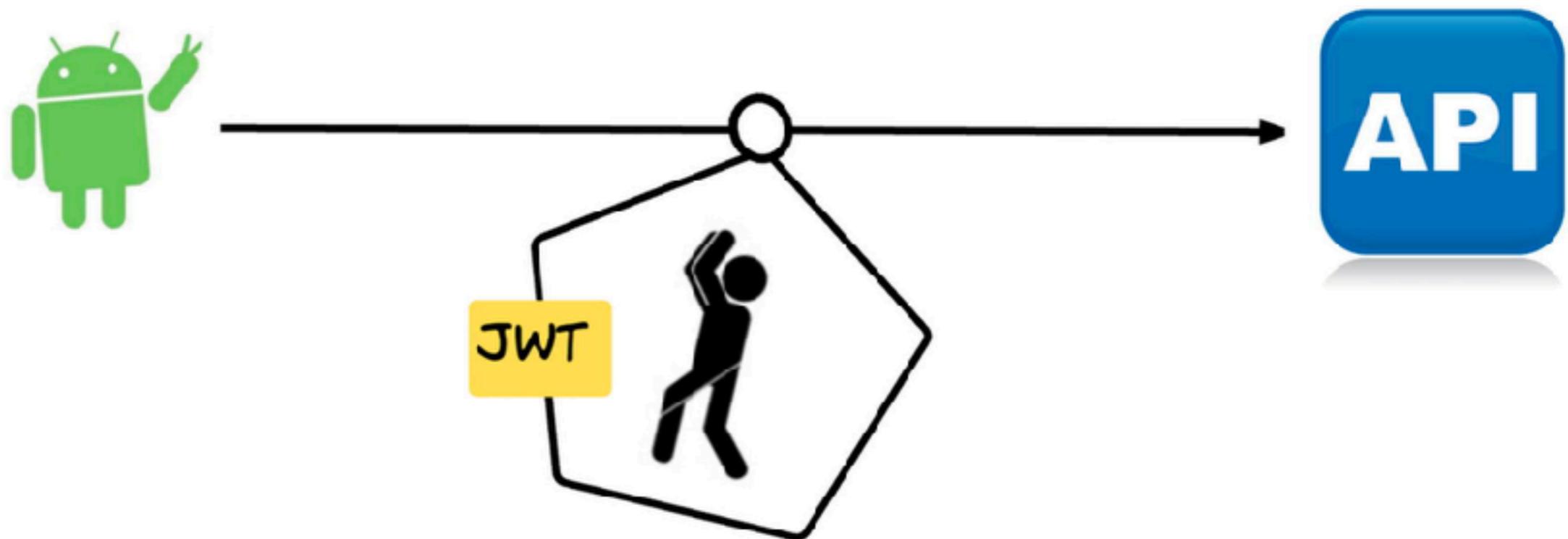
JSON Web Token (JWT)

Transport Layer Security (TLS) mutual authentication

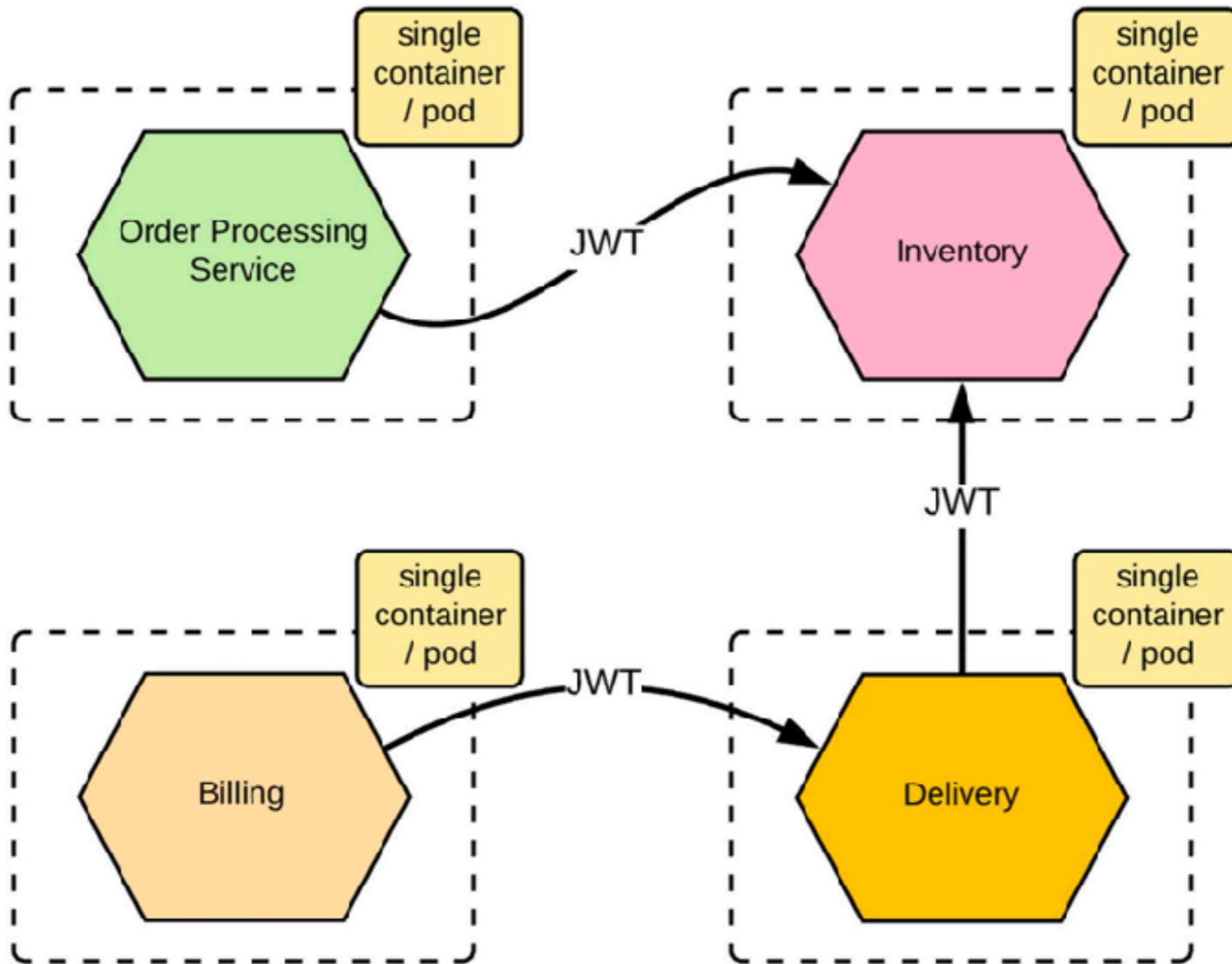


# JSON Web Token (JWT)

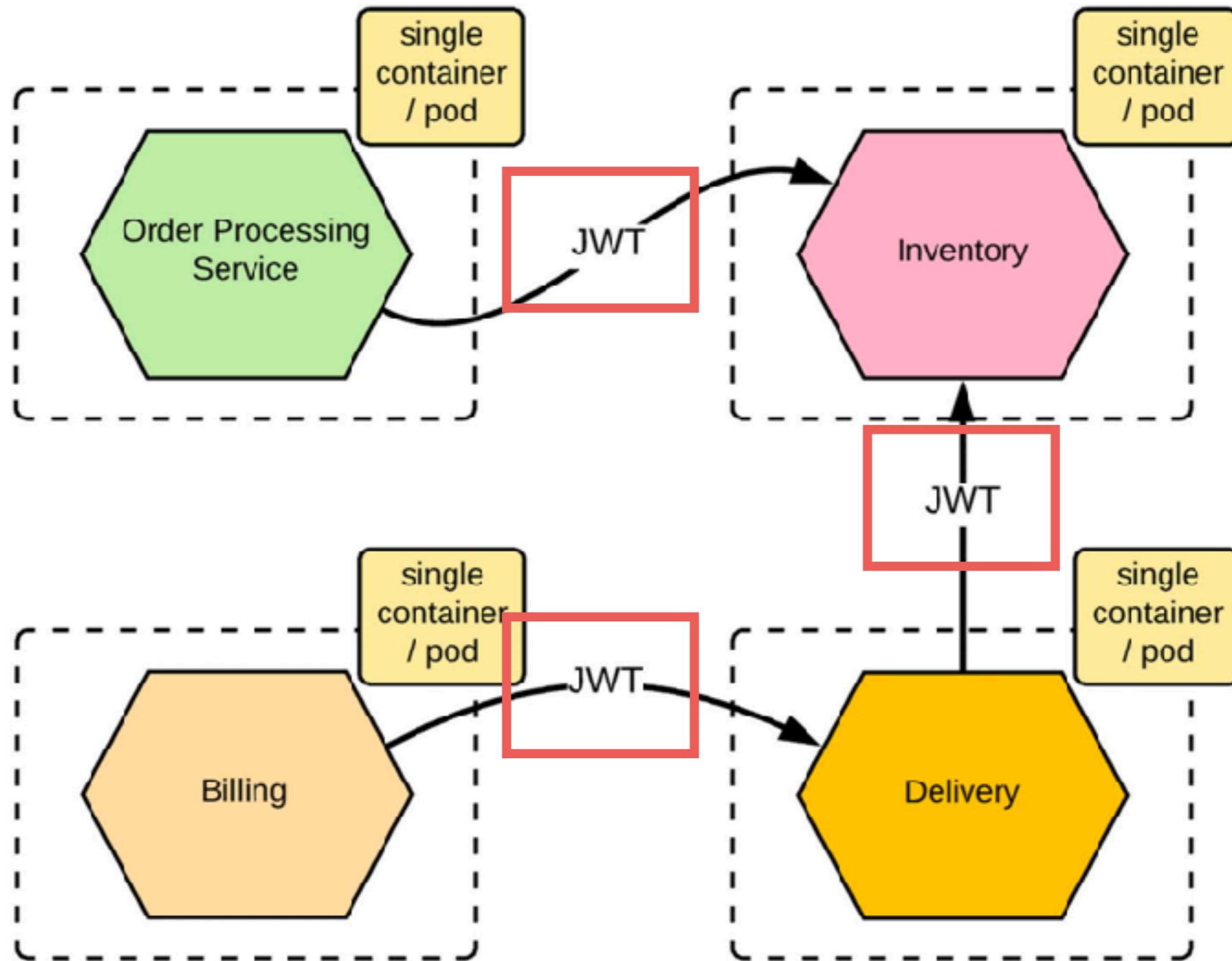
Define a container to transport data between interested parties



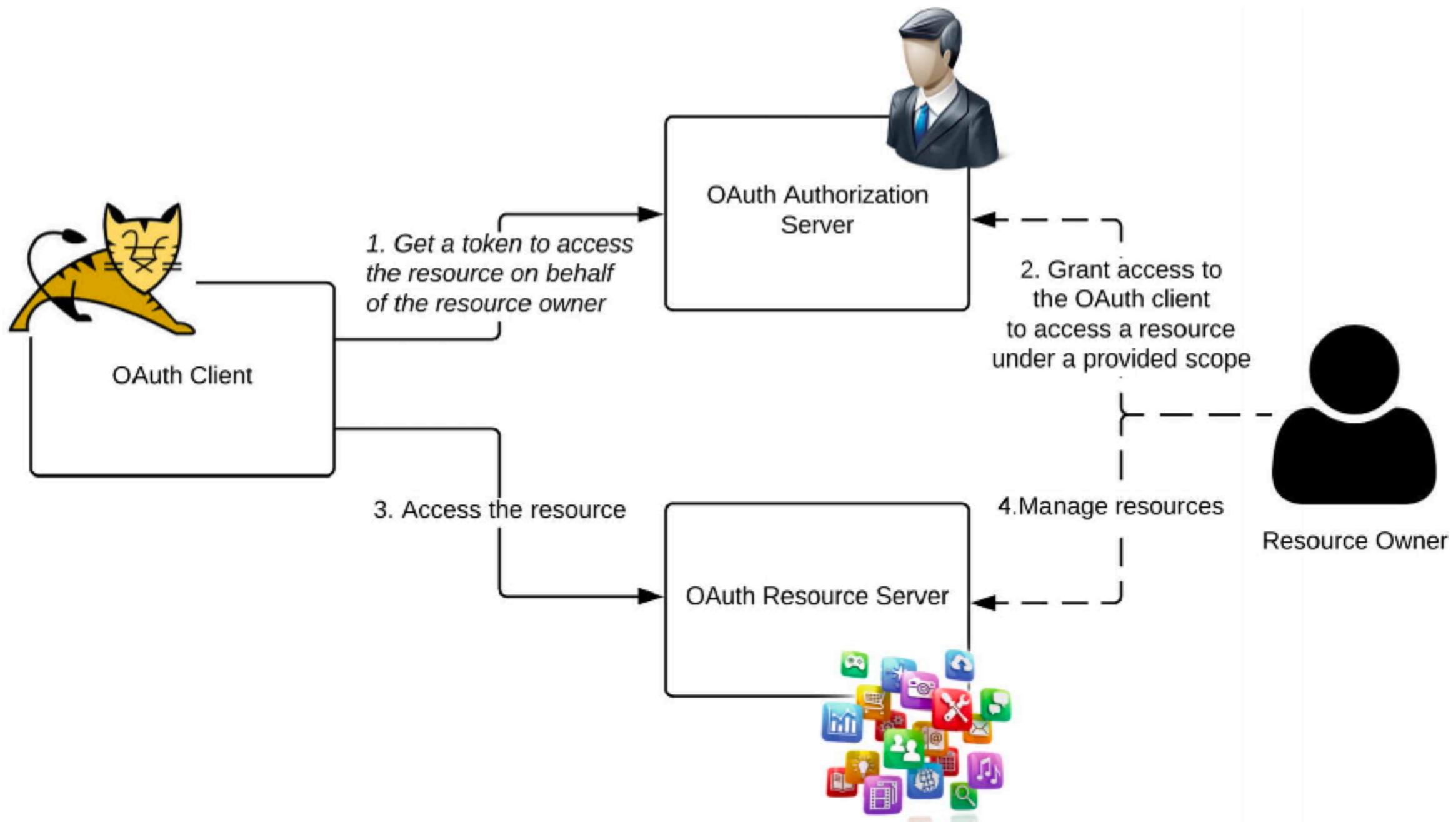
# Passing user context as JWT



# Problem ?

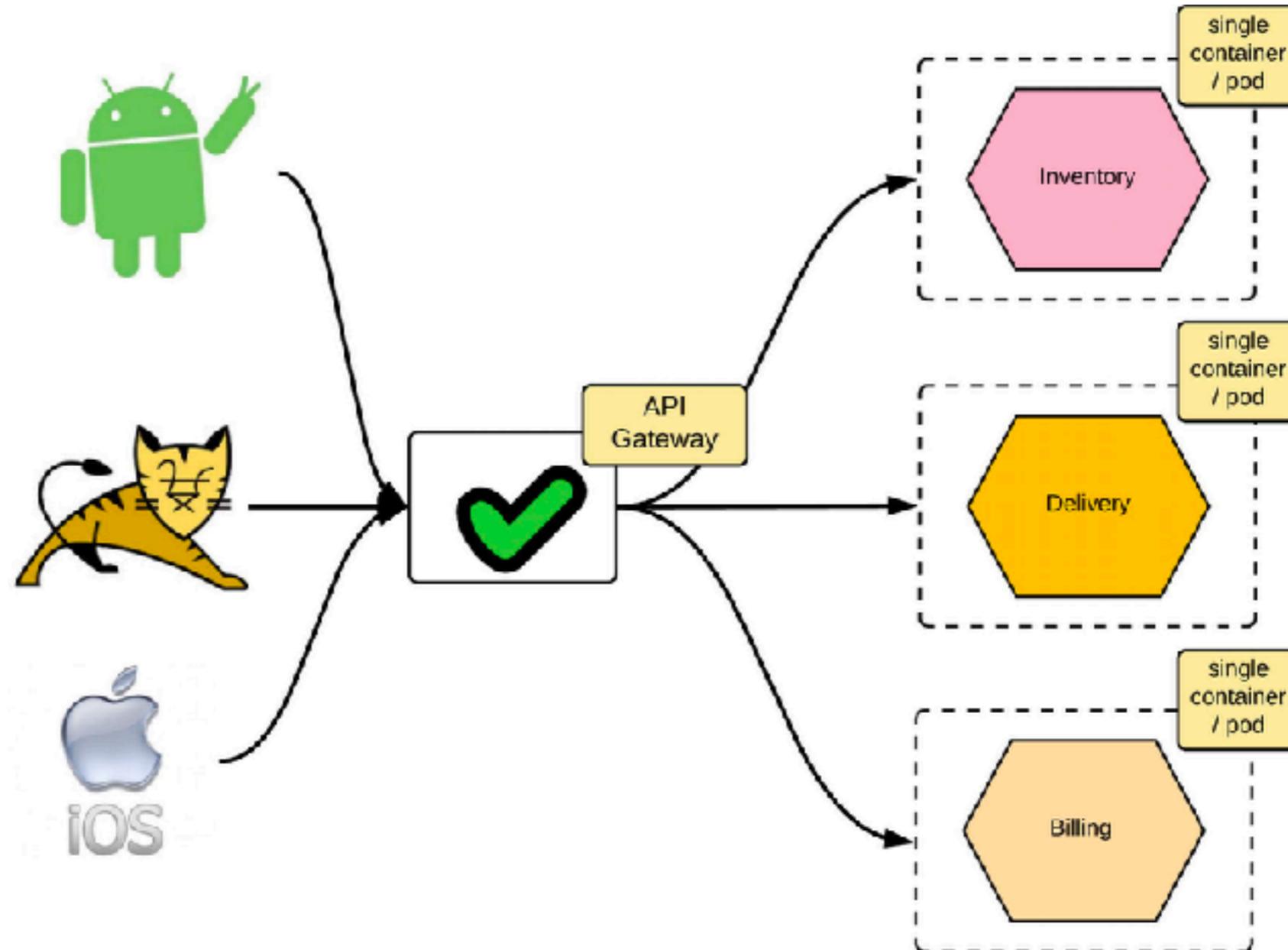


# OAuth 2.0



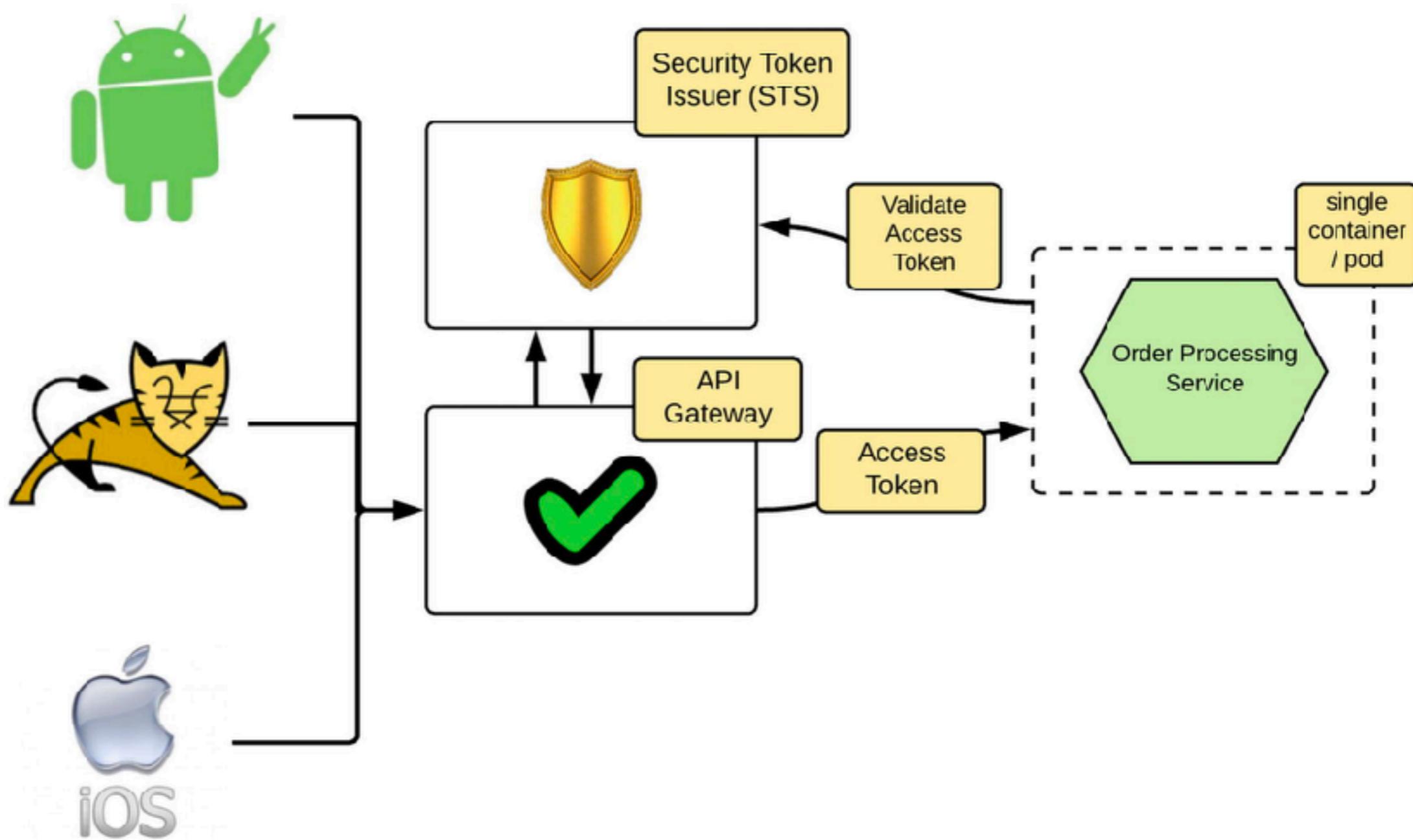
# Centralize pattern

## Using API gateway pattern

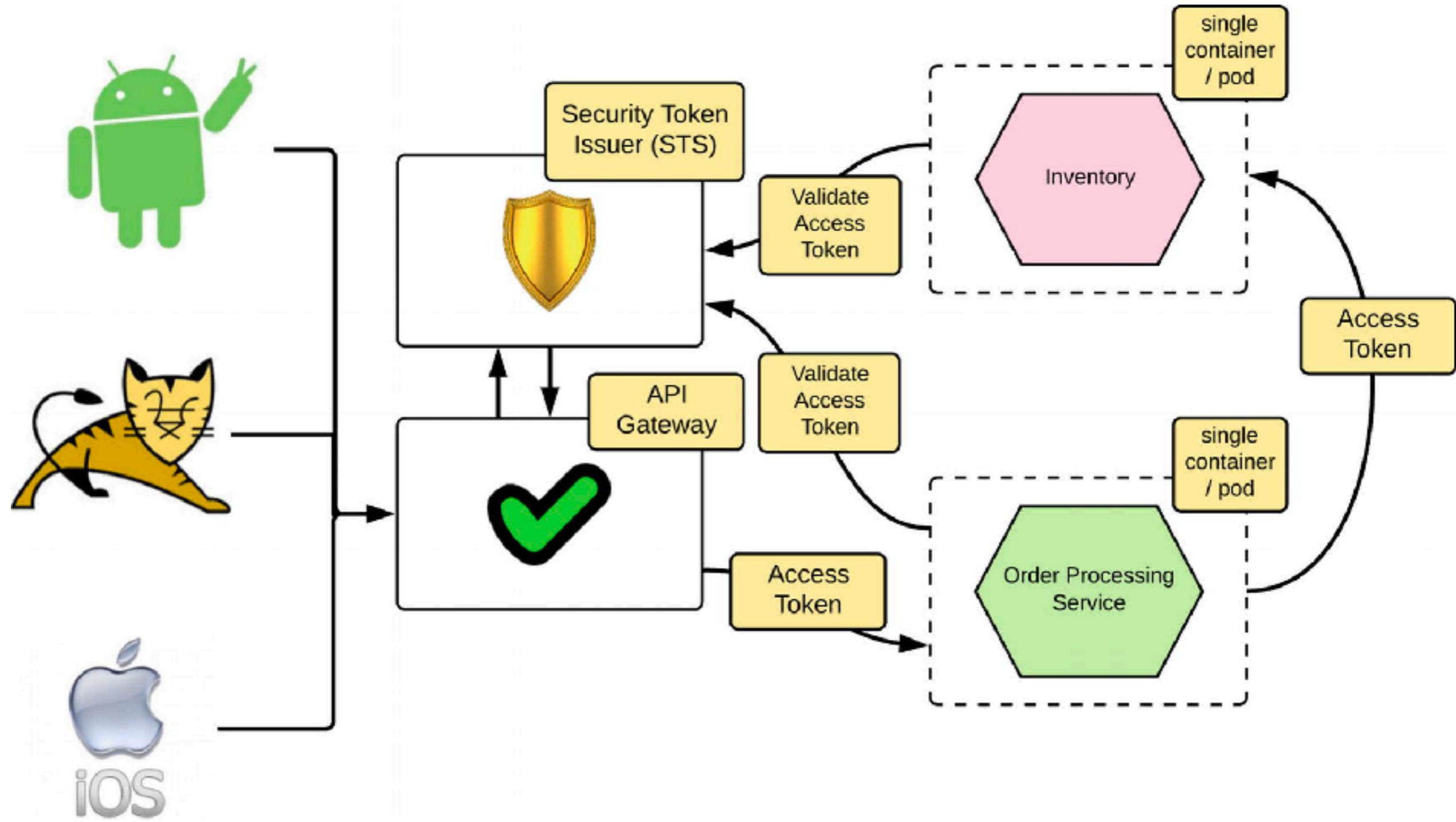


# Centralize pattern

## Using API gateway pattern

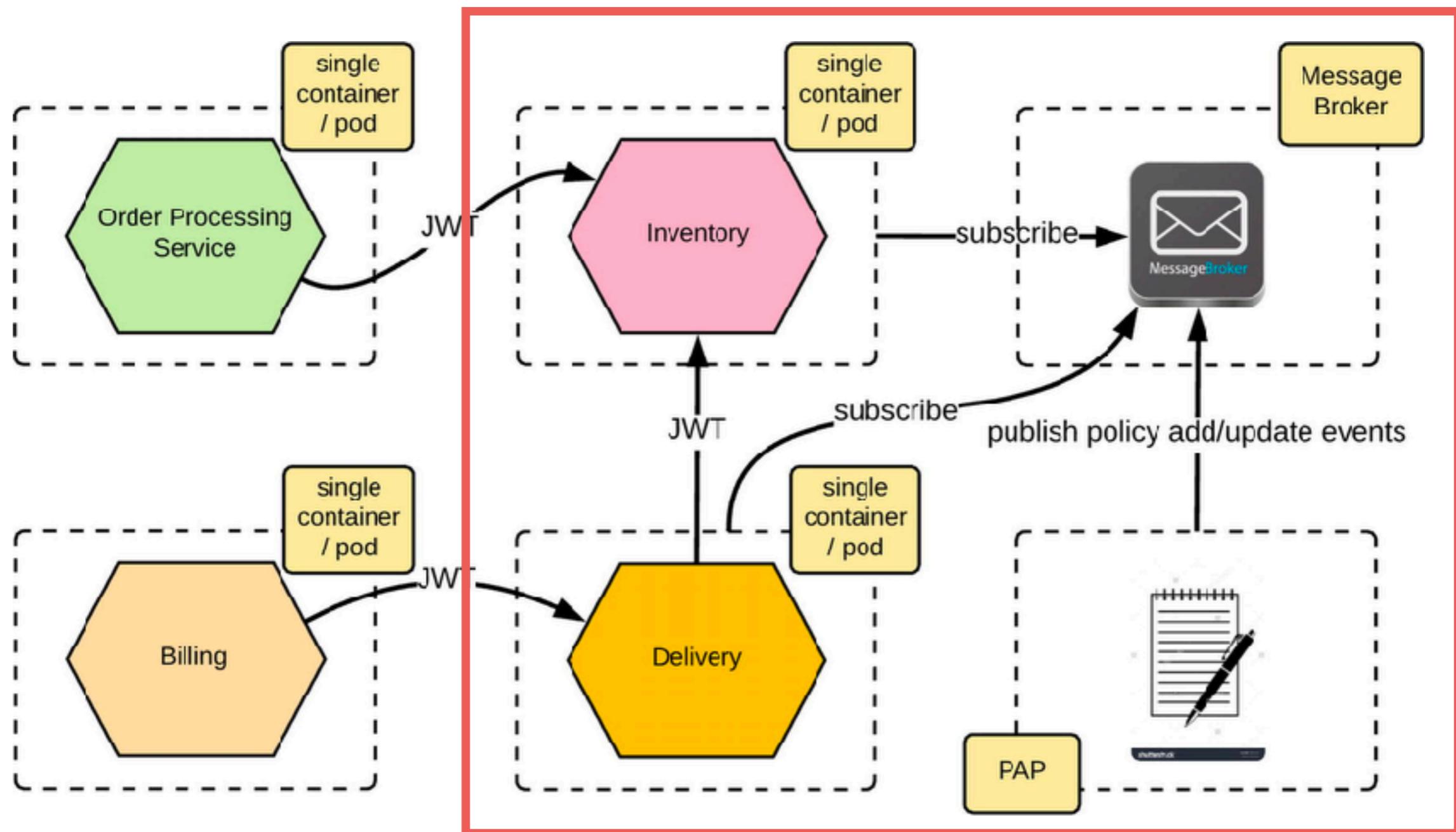


# API gateway with OAuth 2.0



# Access control of services

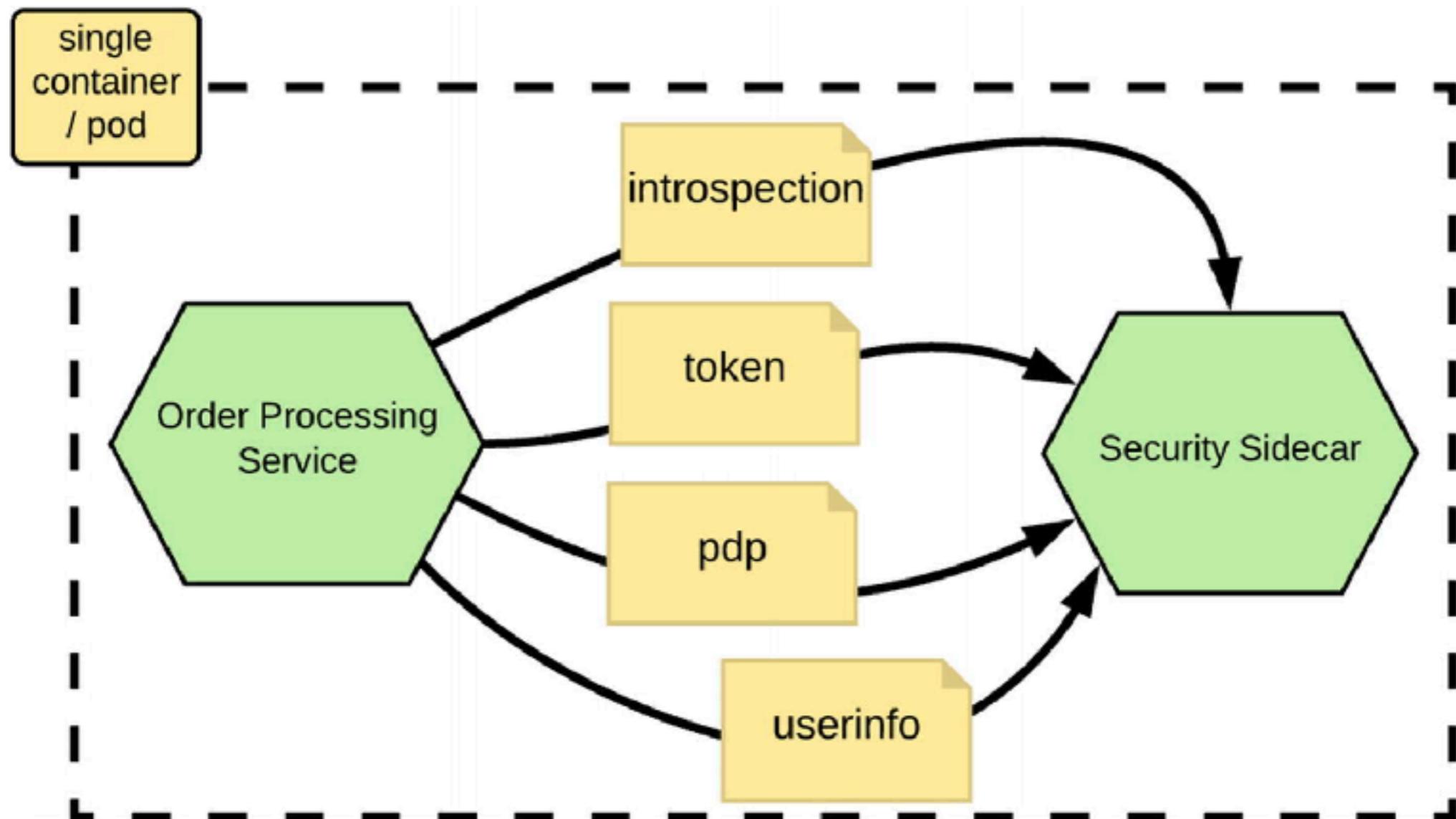
## Policy Administration Point



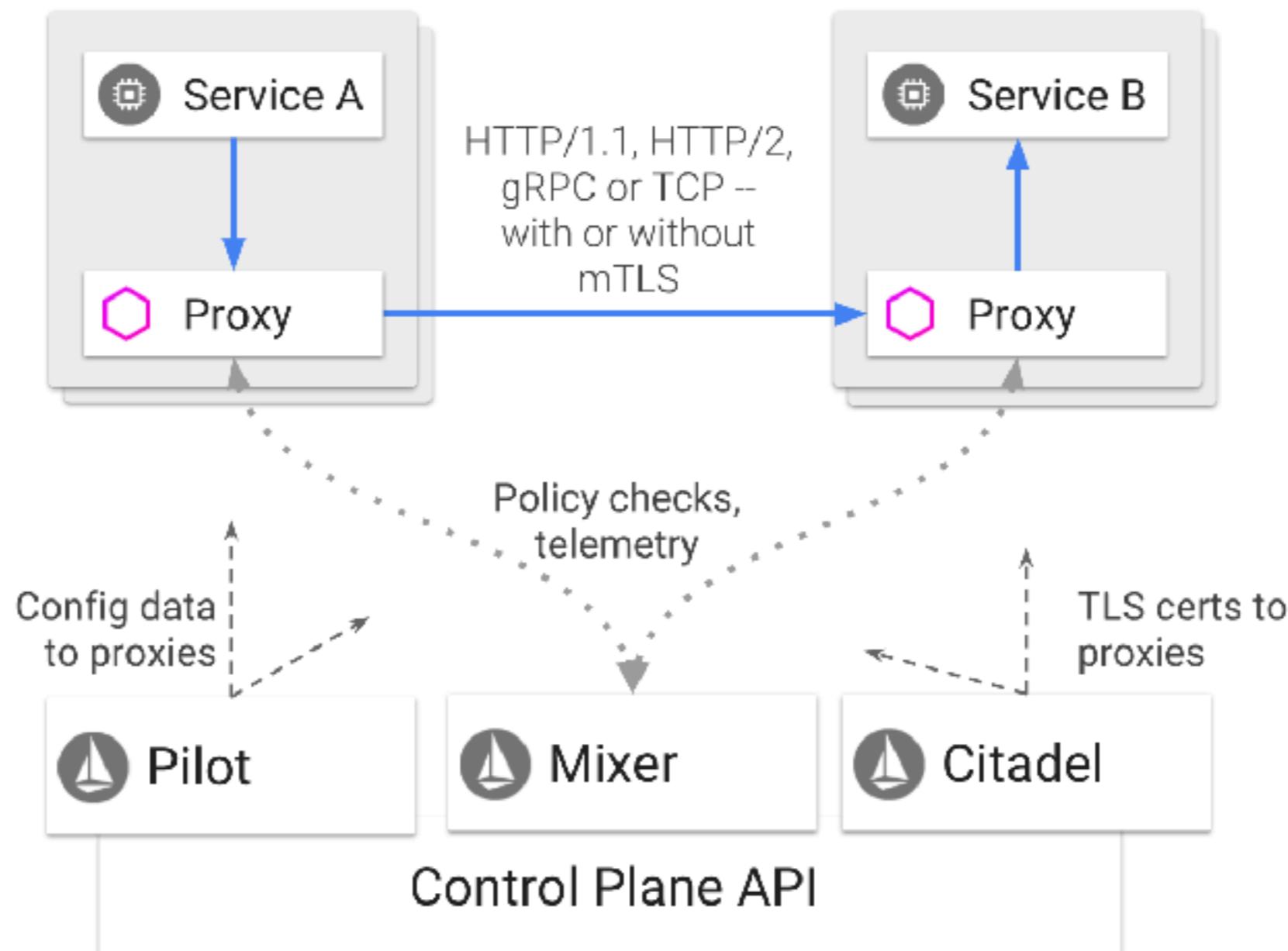
# Working sidecar



# Security sidecar



# Security sidecar



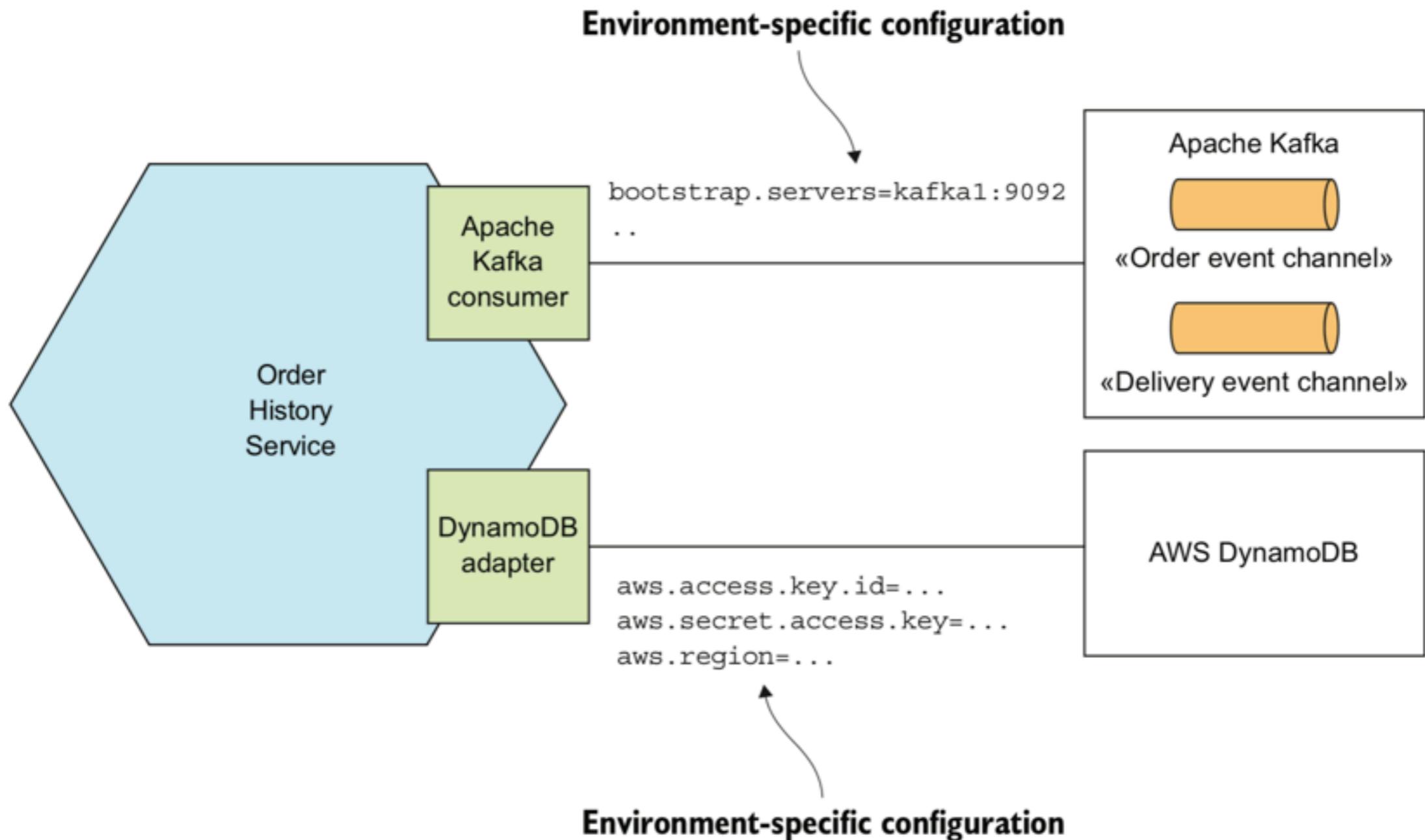
<https://istio.io/docs/concepts/what-is-istio/>



# Configurable services



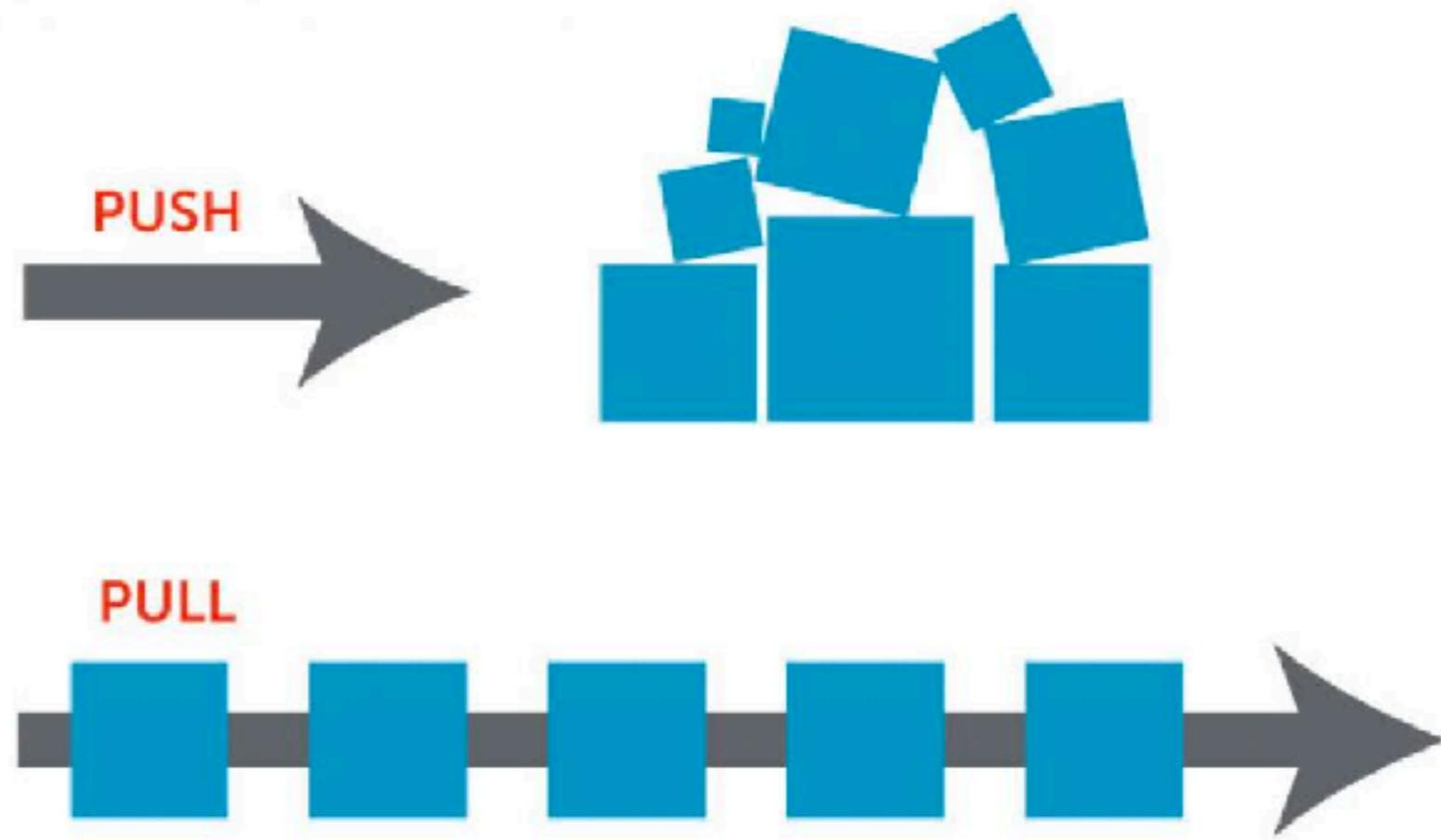
# Design configurable services



# External configuration models

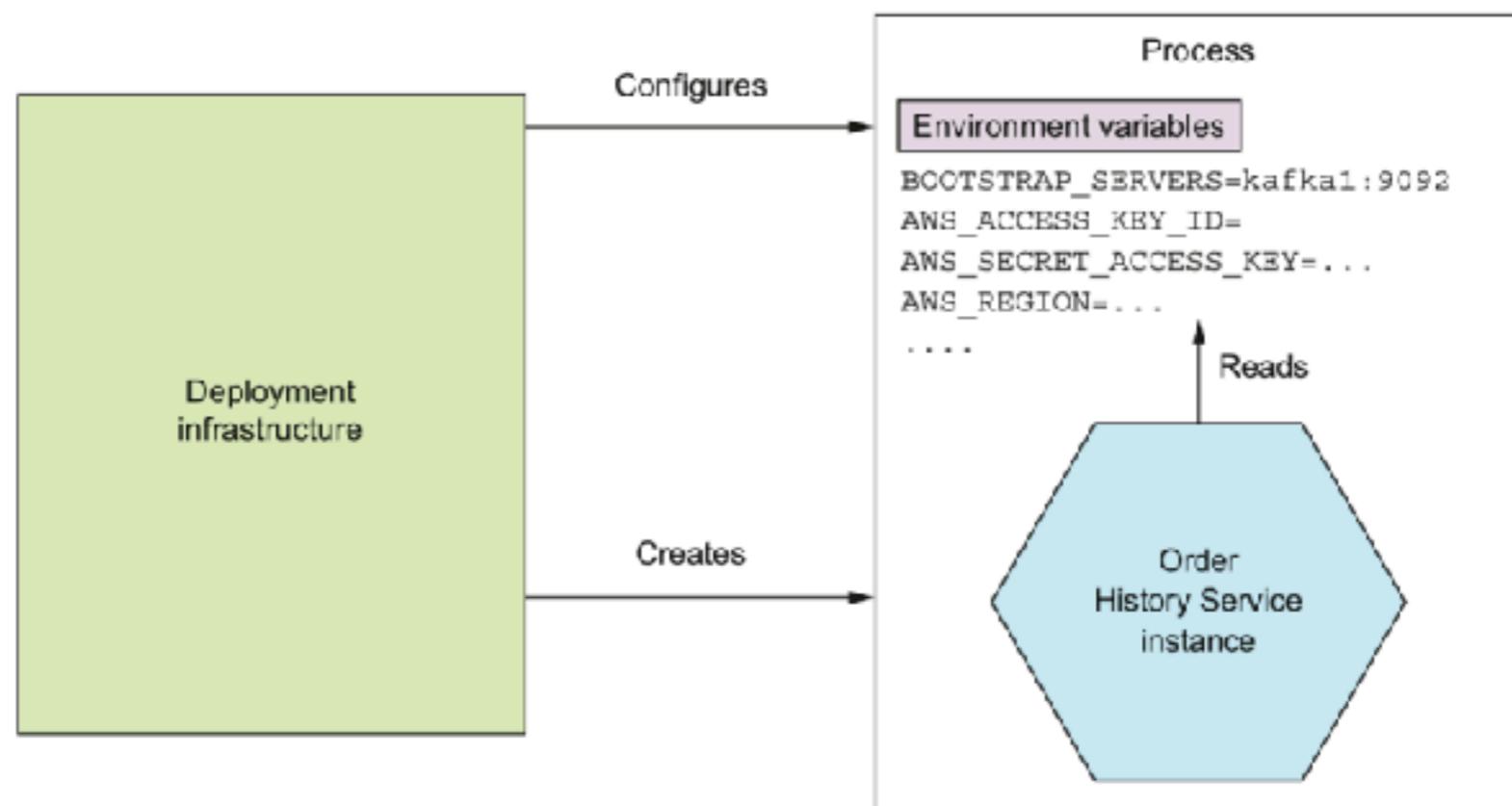
Push model

Pull model



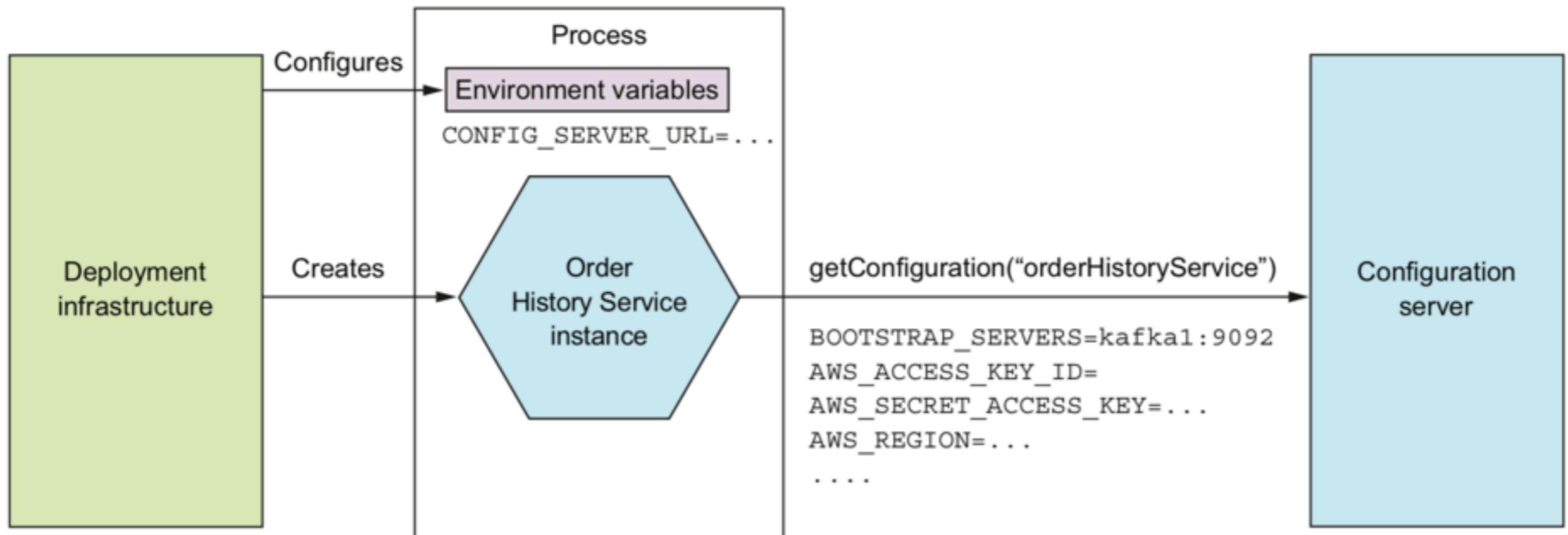
# Push model

Pass the configuration to service  
OS environment variables  
Configuration files



# Pull model

Service read configuration from configuration server



# Benefits of configuration server

Centralized configuration

Transparent decryption of sensitive data

Dynamic reconfiguration



# Drawbacks of configuration server

Need setup and maintain !!



# Design observable services

Health check API

Log aggregation

Distributed tracing

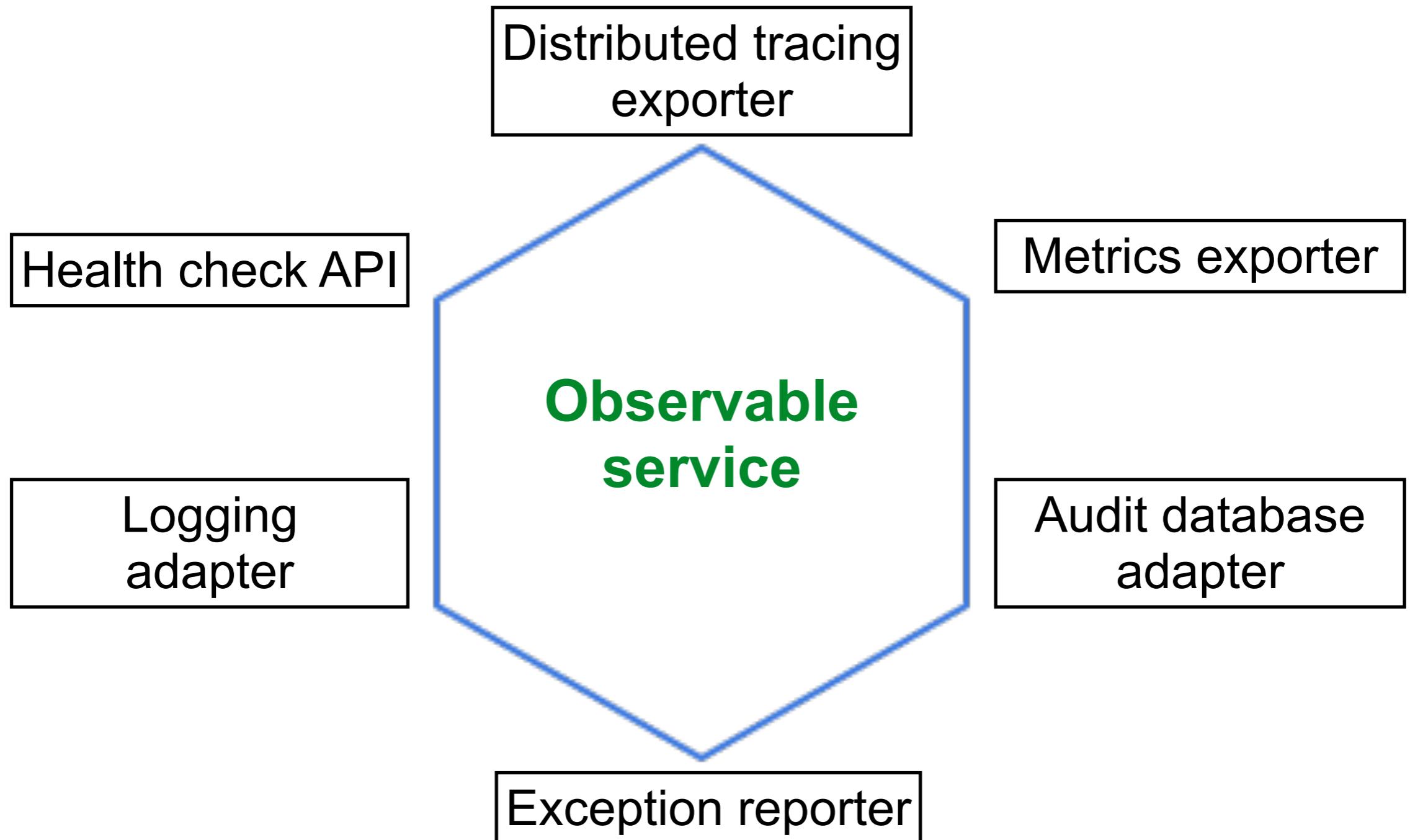
Exception tracking

Application metrics

Audit logging

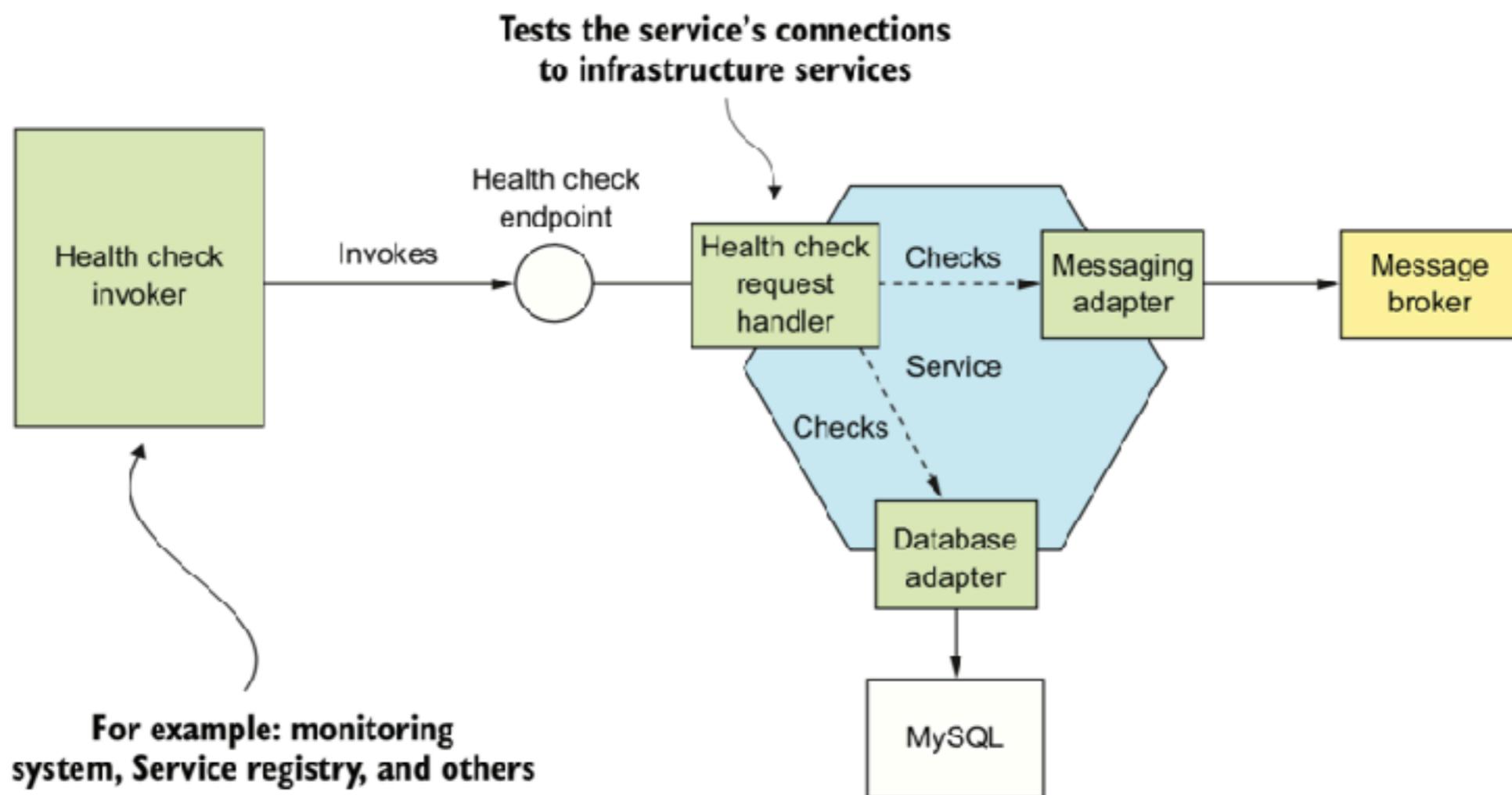


# Observable services



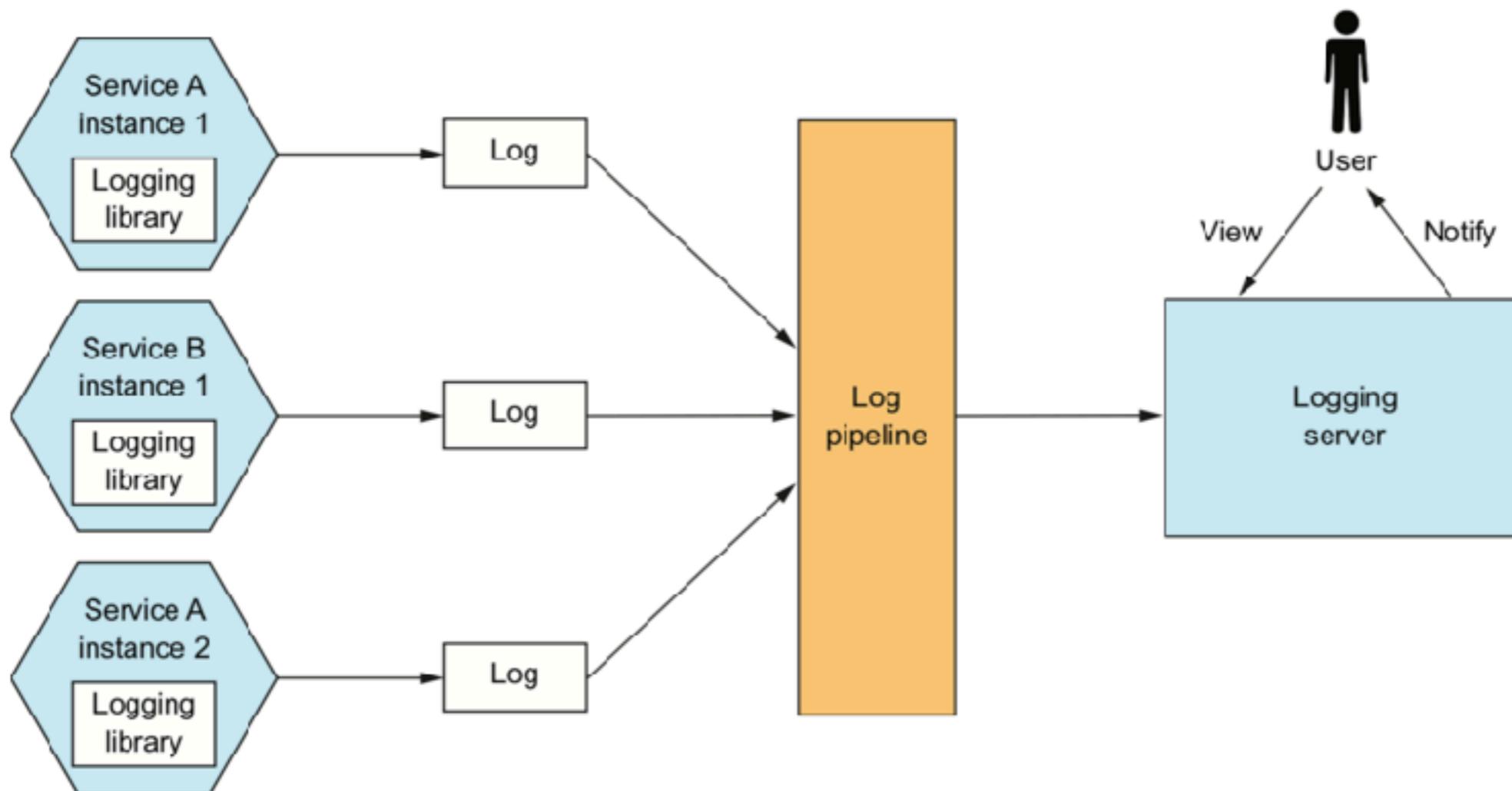
# Health check API

Expose an endpoint that return the health of service



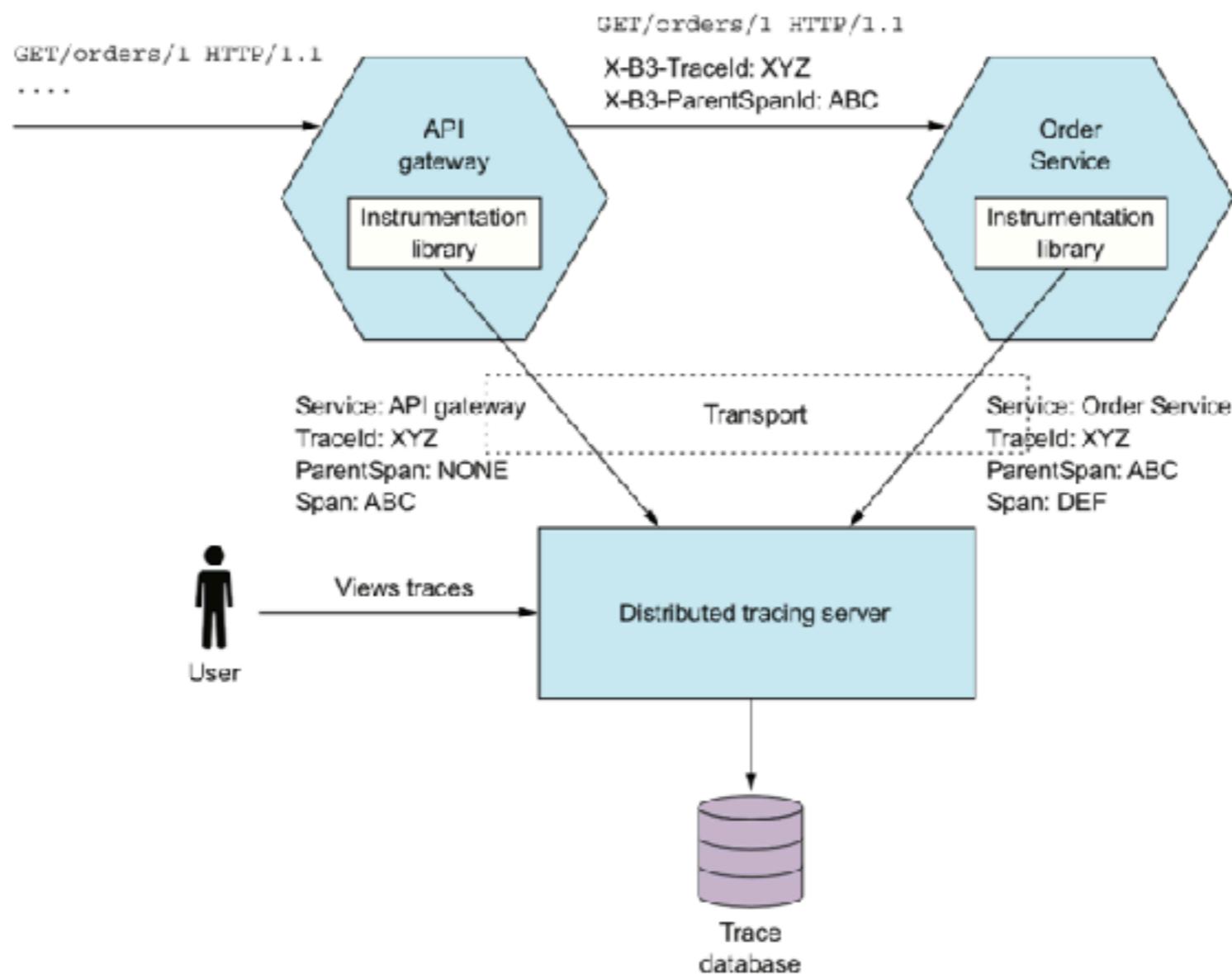
# Log aggregation

Log service activity and write logs into a centralized logging server. (searching, alerting)

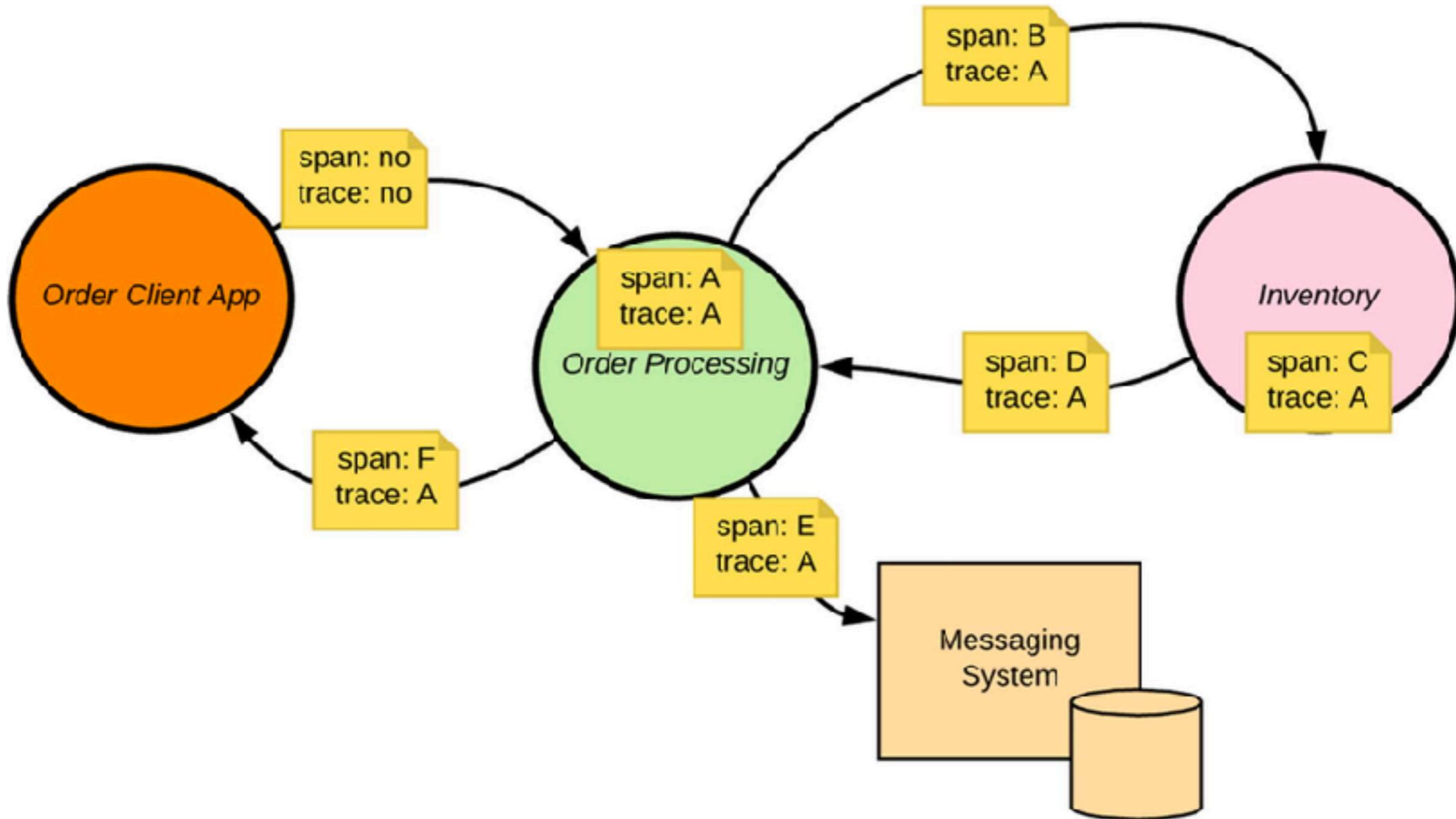


# Distributed tracing

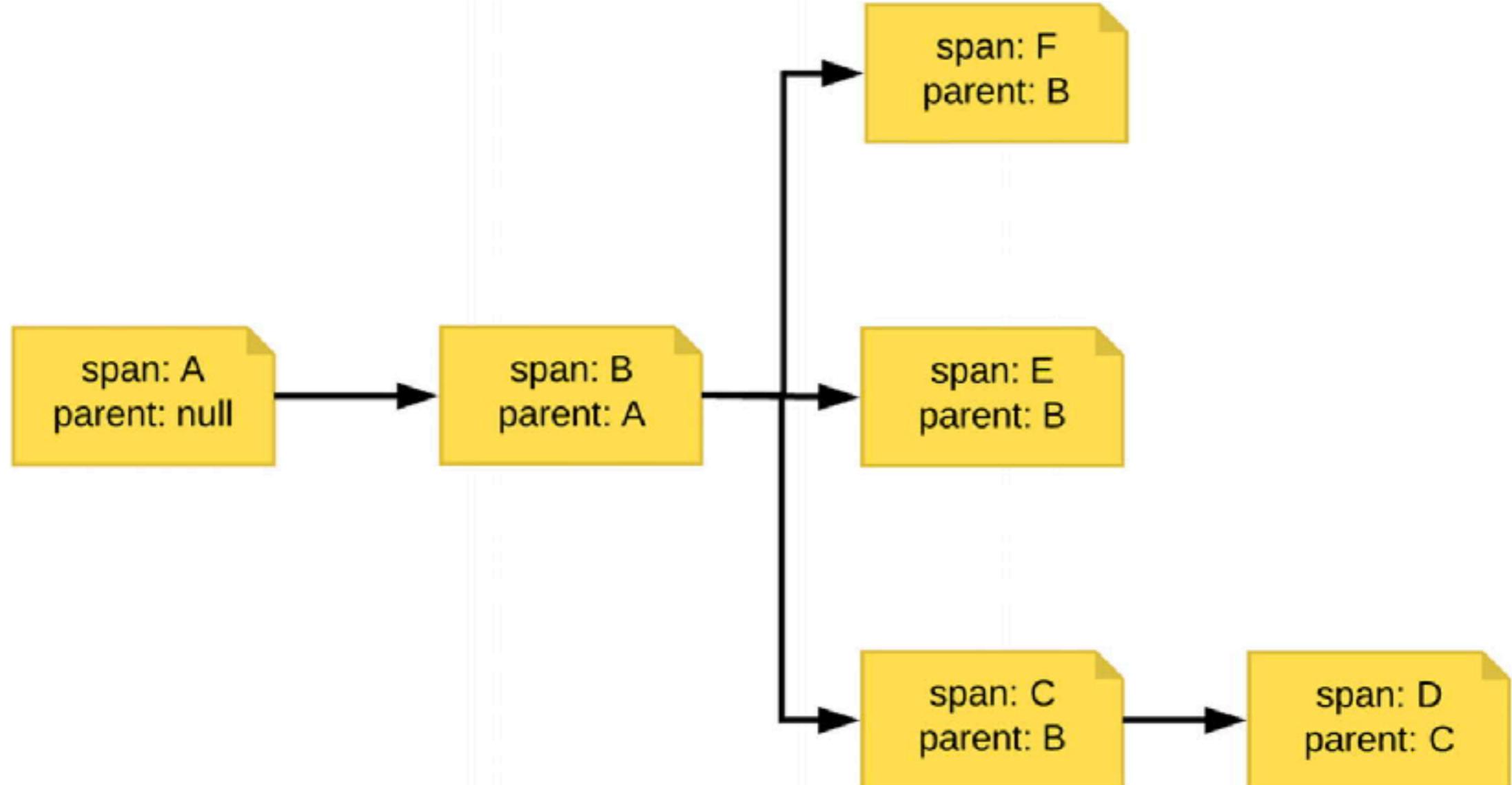
Assign each external request a unique ID and trace requests as flow between services



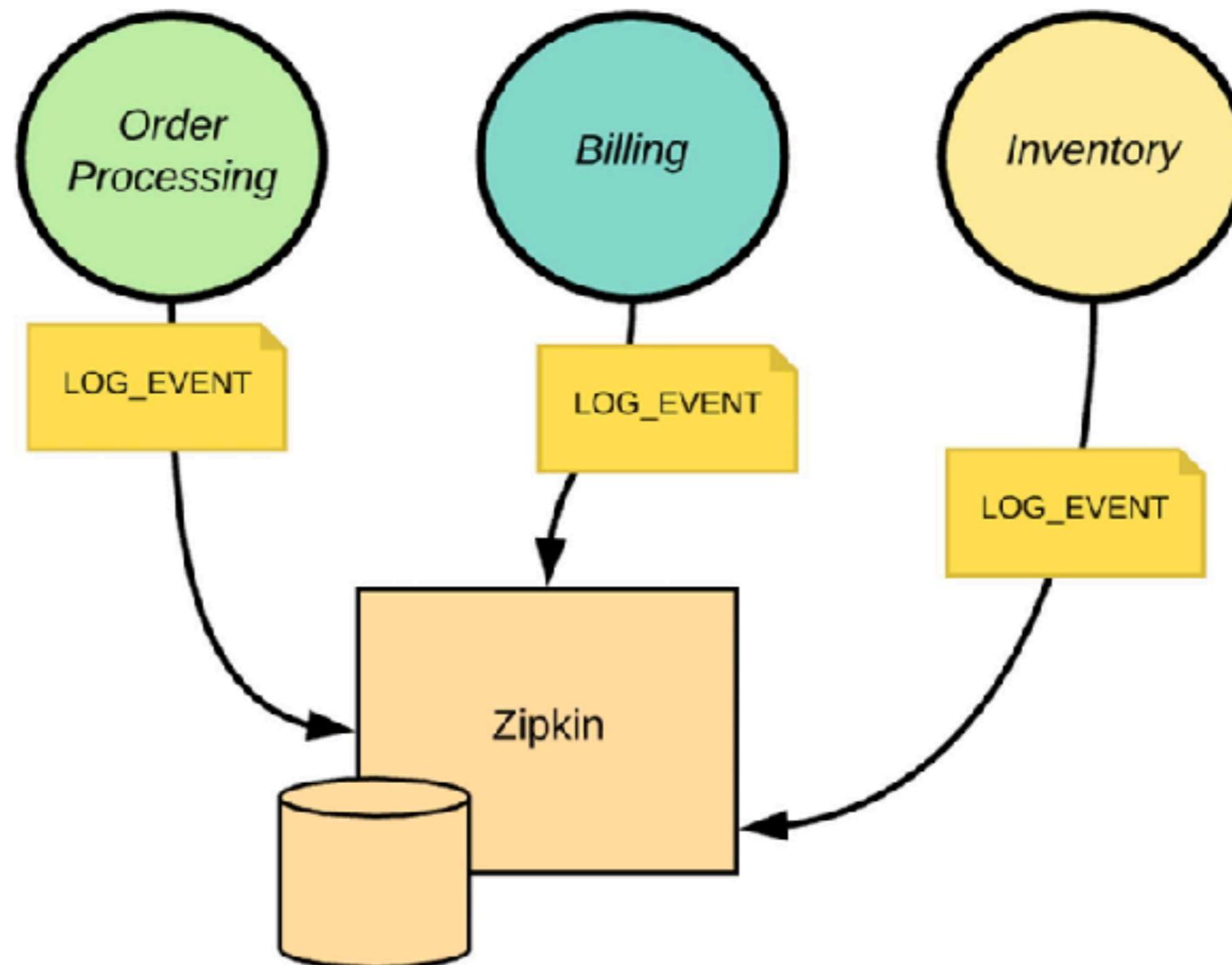
# Distributed tracing



# Distributed tracing



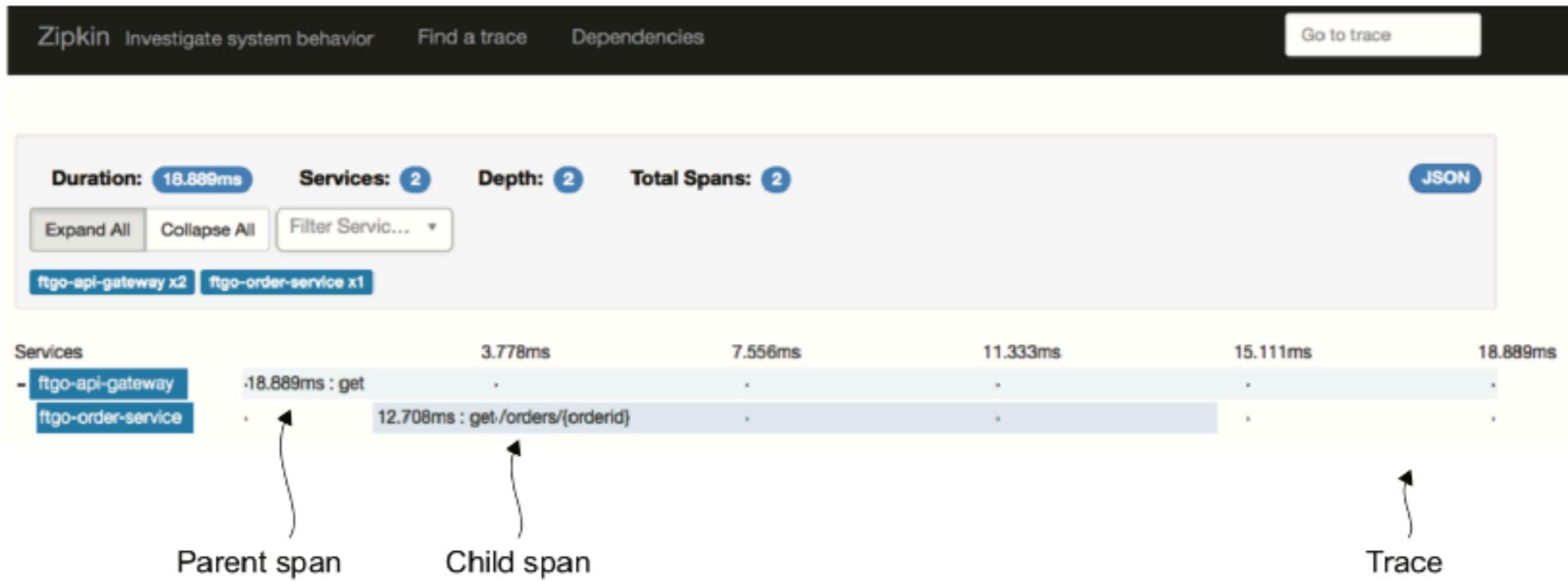
# Distributed tracing with Zipkin



<https://zipkin.io/>



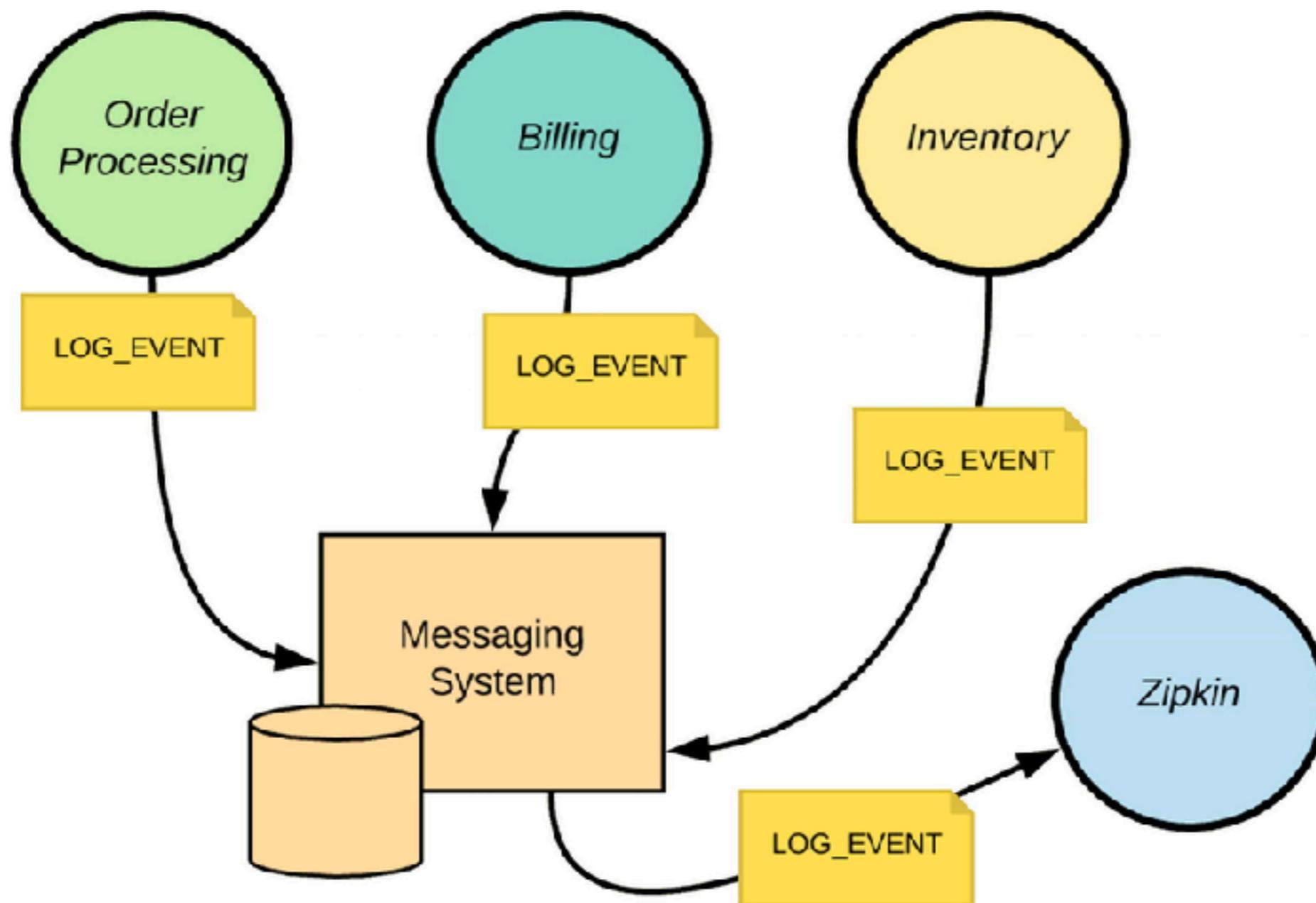
# Distributed tracing with Zipkin



<https://zipkin.io/>

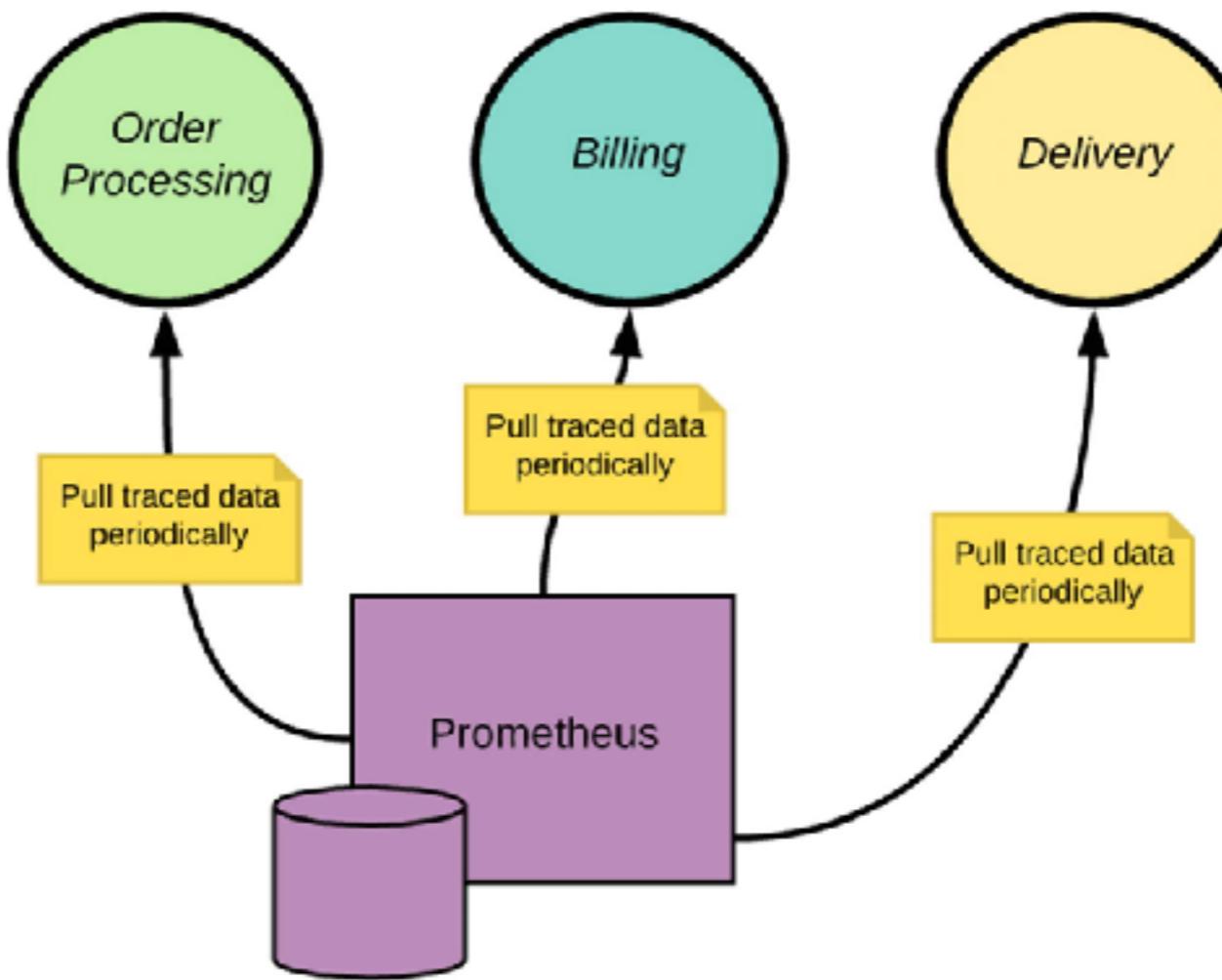


# Distributed tracing with Zipkin



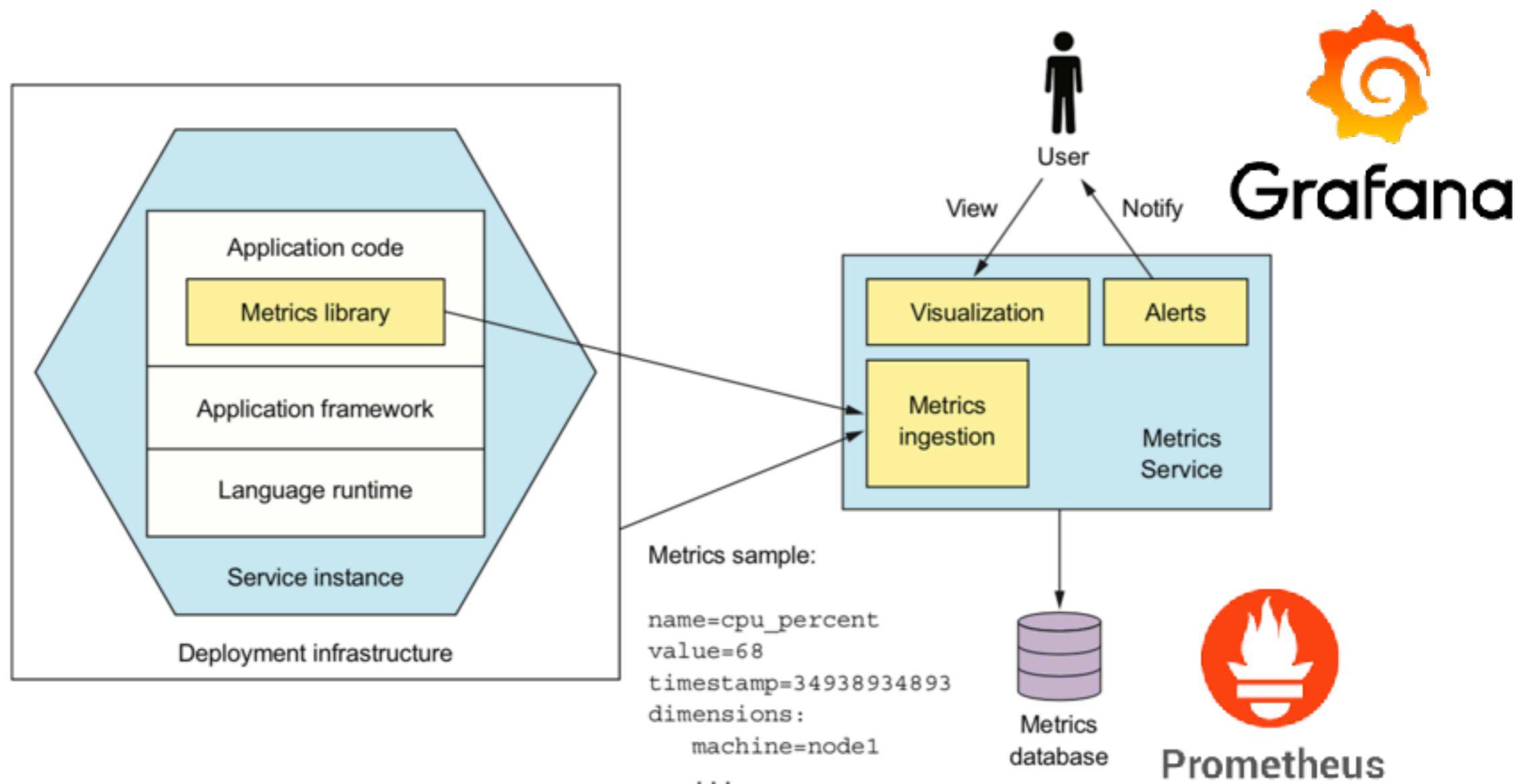
# Application metrics

Services maintain metrics and expose to metric server (counters, gauges)



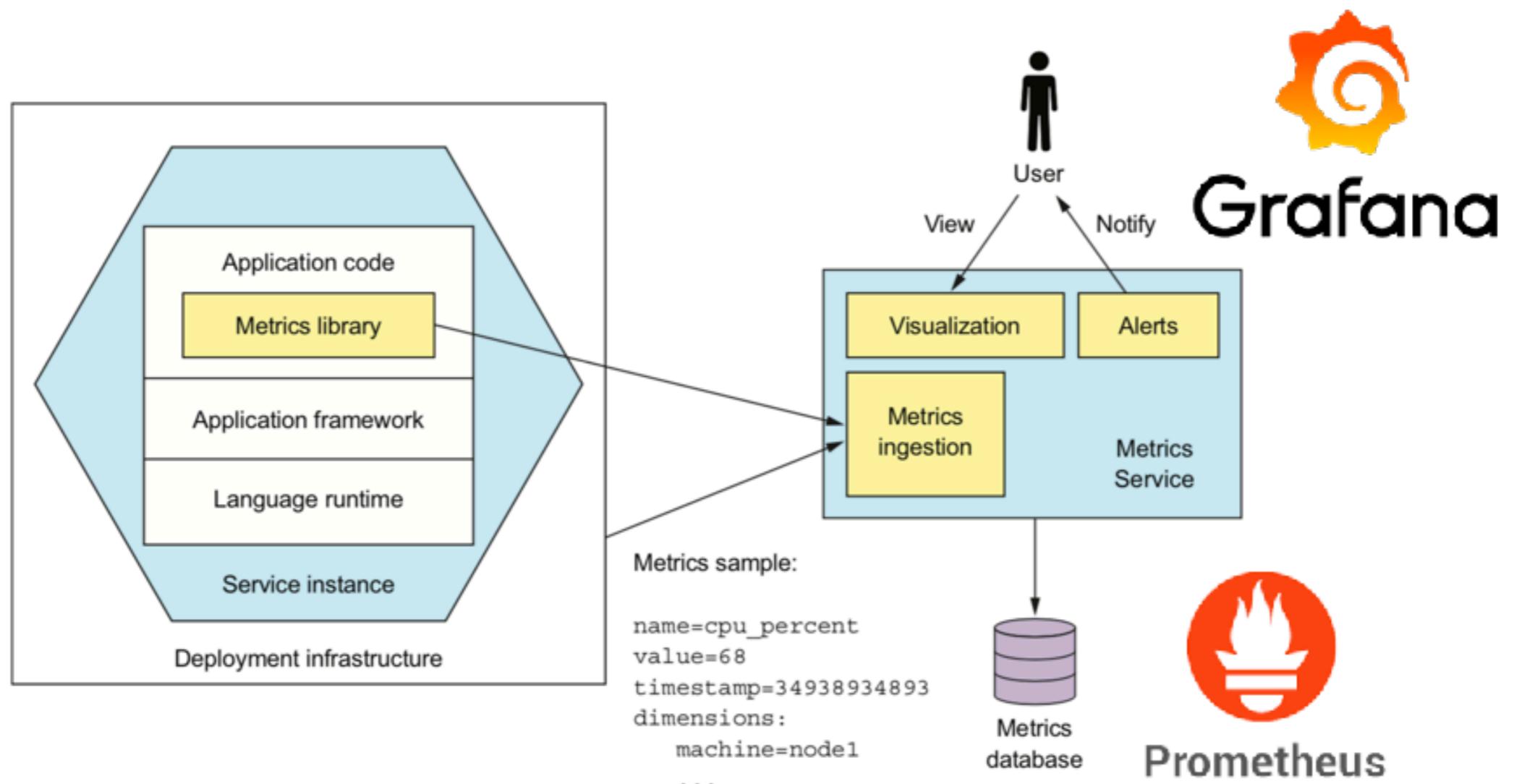
# Application metrics

Services maintain metrics and expose to metric server (counters, gauges)



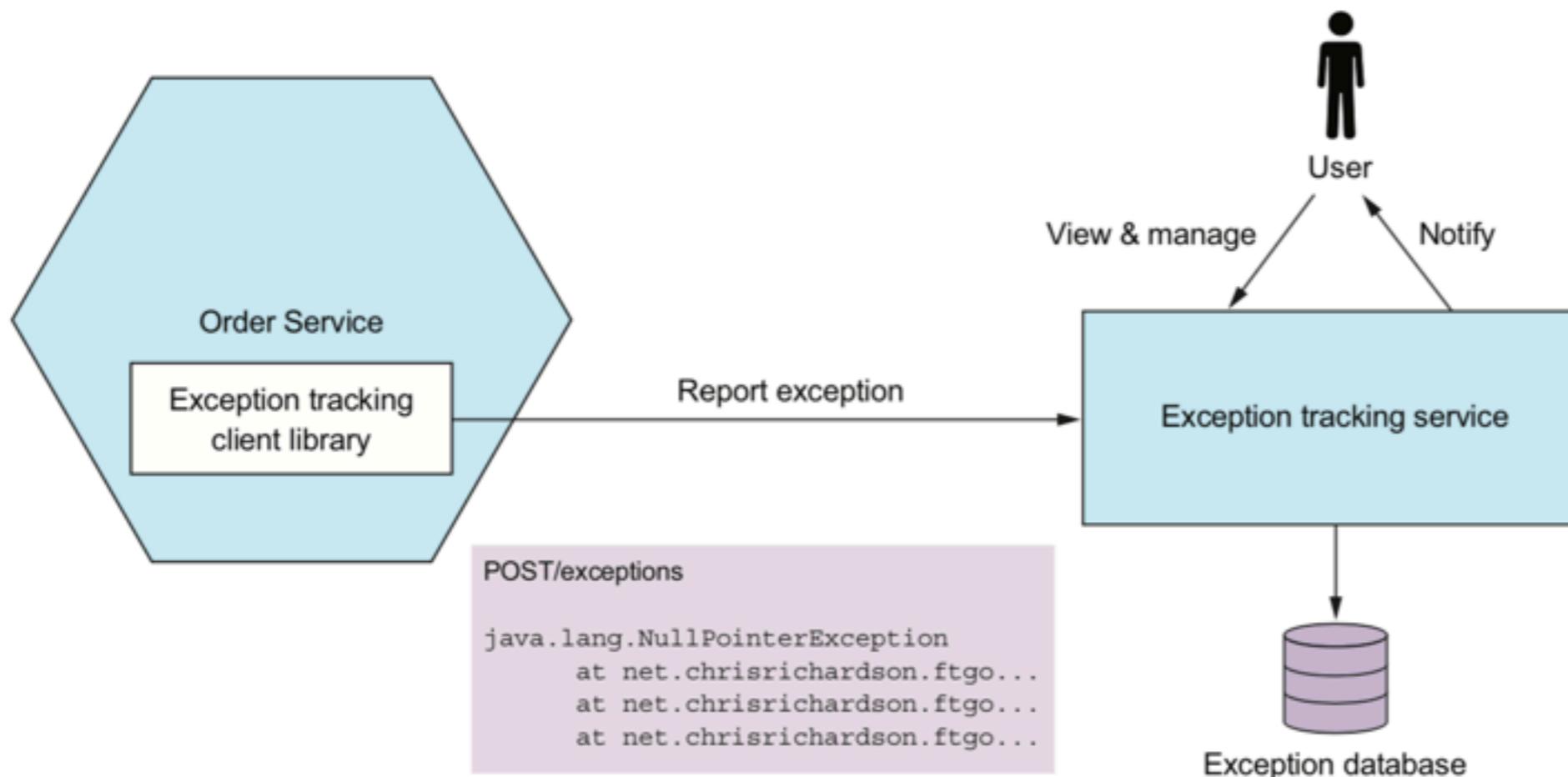
# Application metrics

Metrics database => Prometheus  
Visualization, Alert => Grafana



# Exception tracking

Report exceptions to exception tracking service  
Help to identify the root cause



# Exception tracking services



# Audit logging

Log of user actions

Help customer support

Ensure compliance

Detect suspicious behaviour



# How to implement the audit logging ?

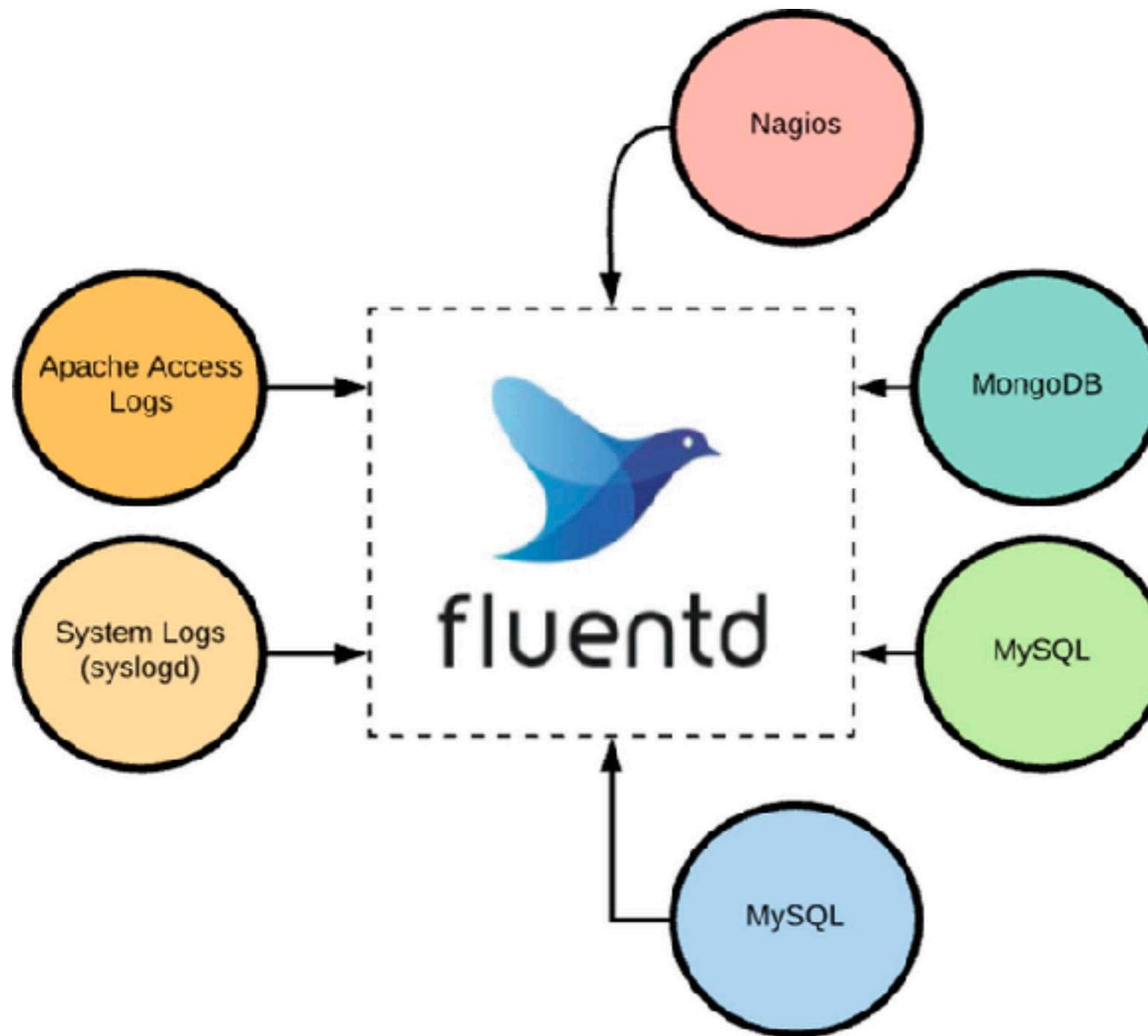
Add logging code in business logic

Use AOP (Aspect-Oriented Programming)

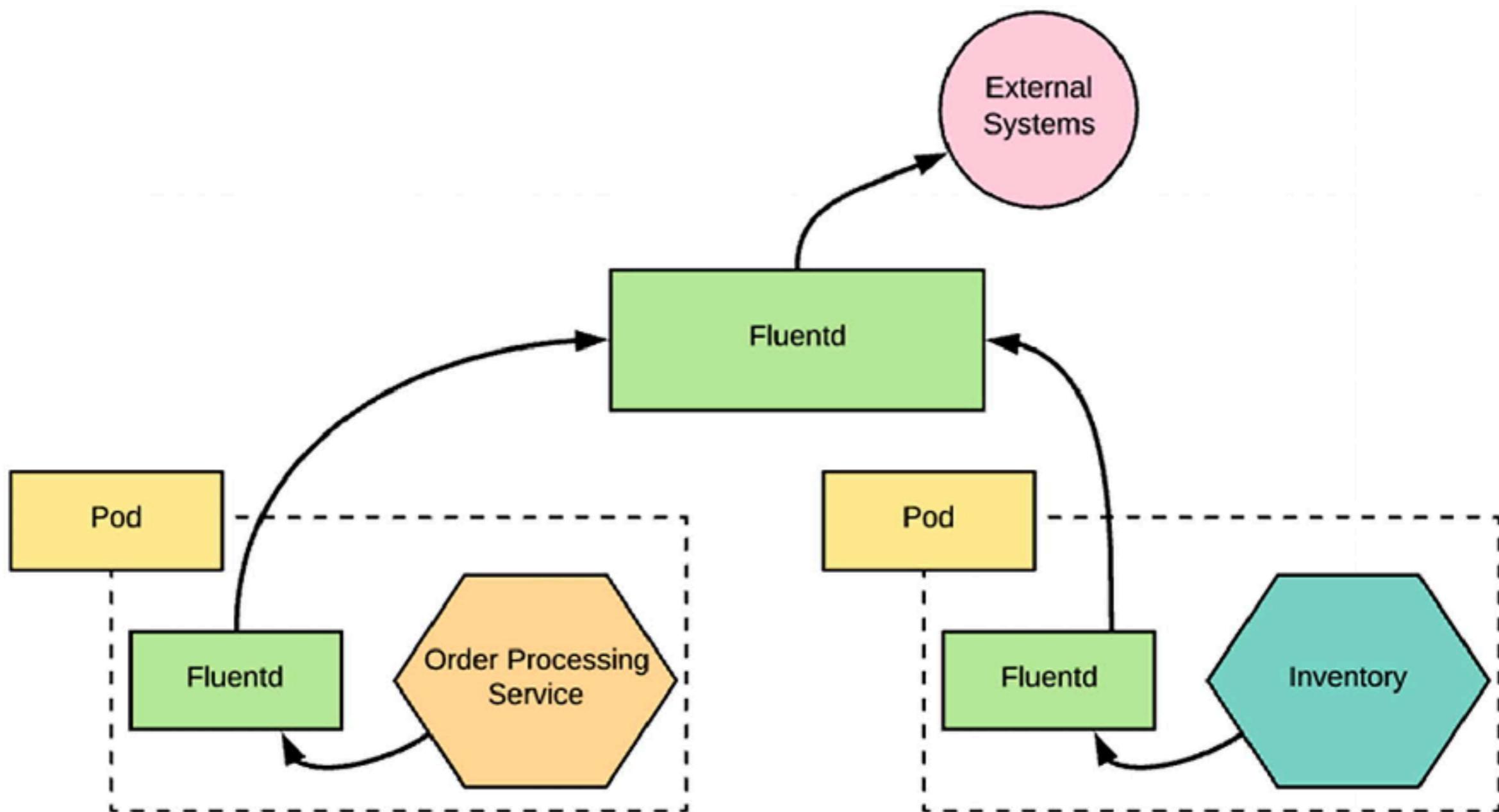
Use event sourcing



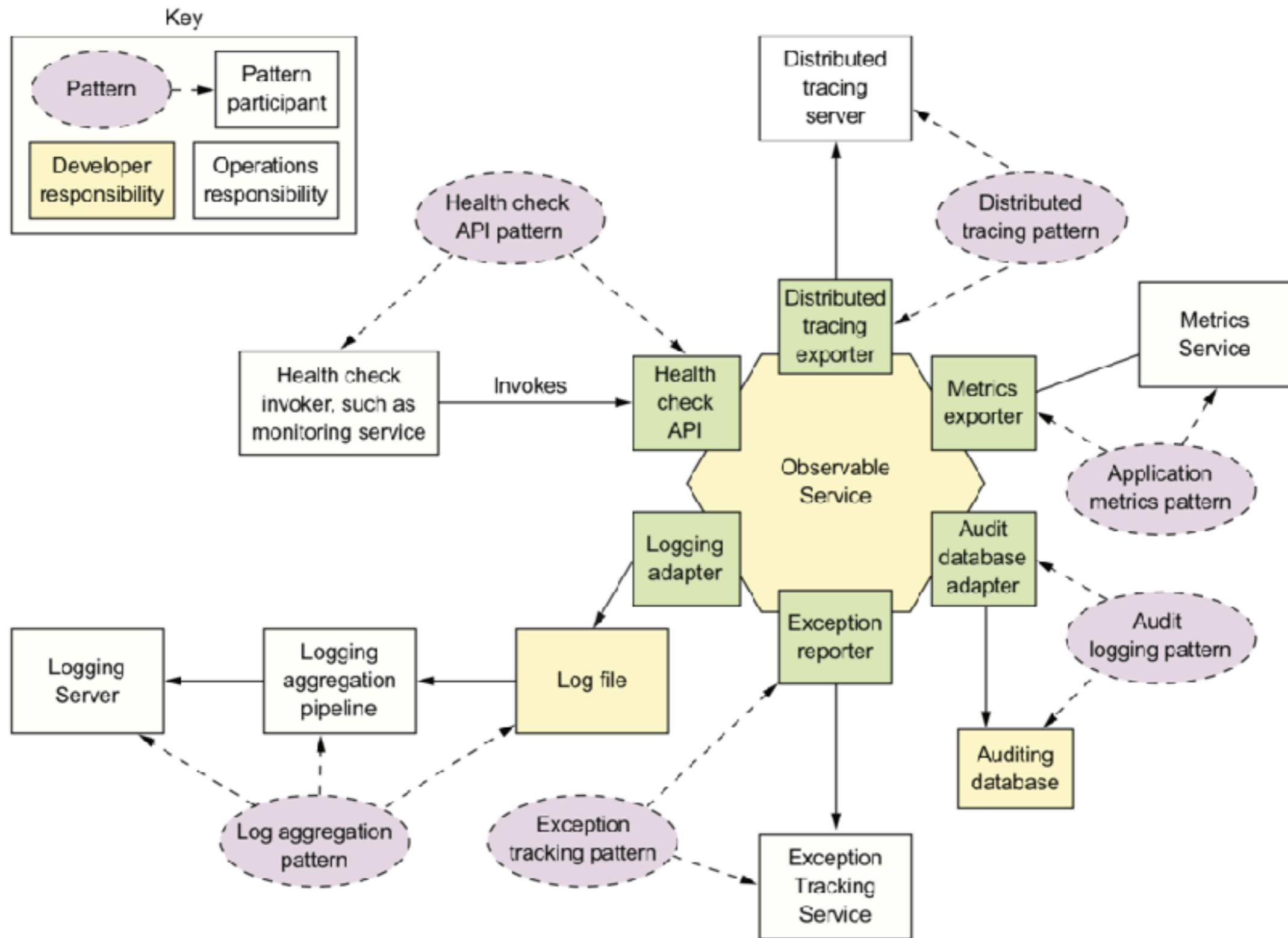
# Centralize logging



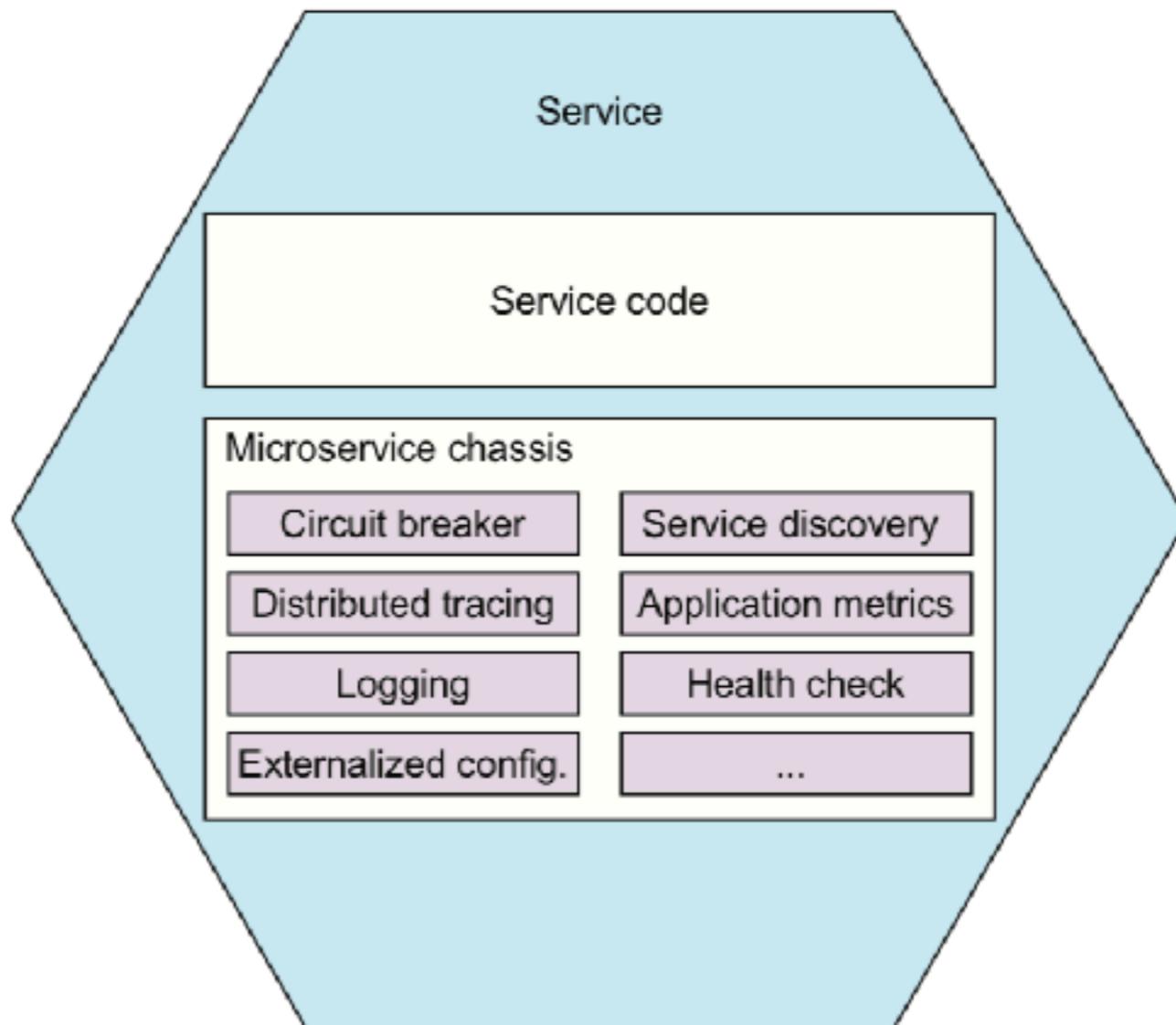
# Centralize logging



# Observable services



# Microservice framework

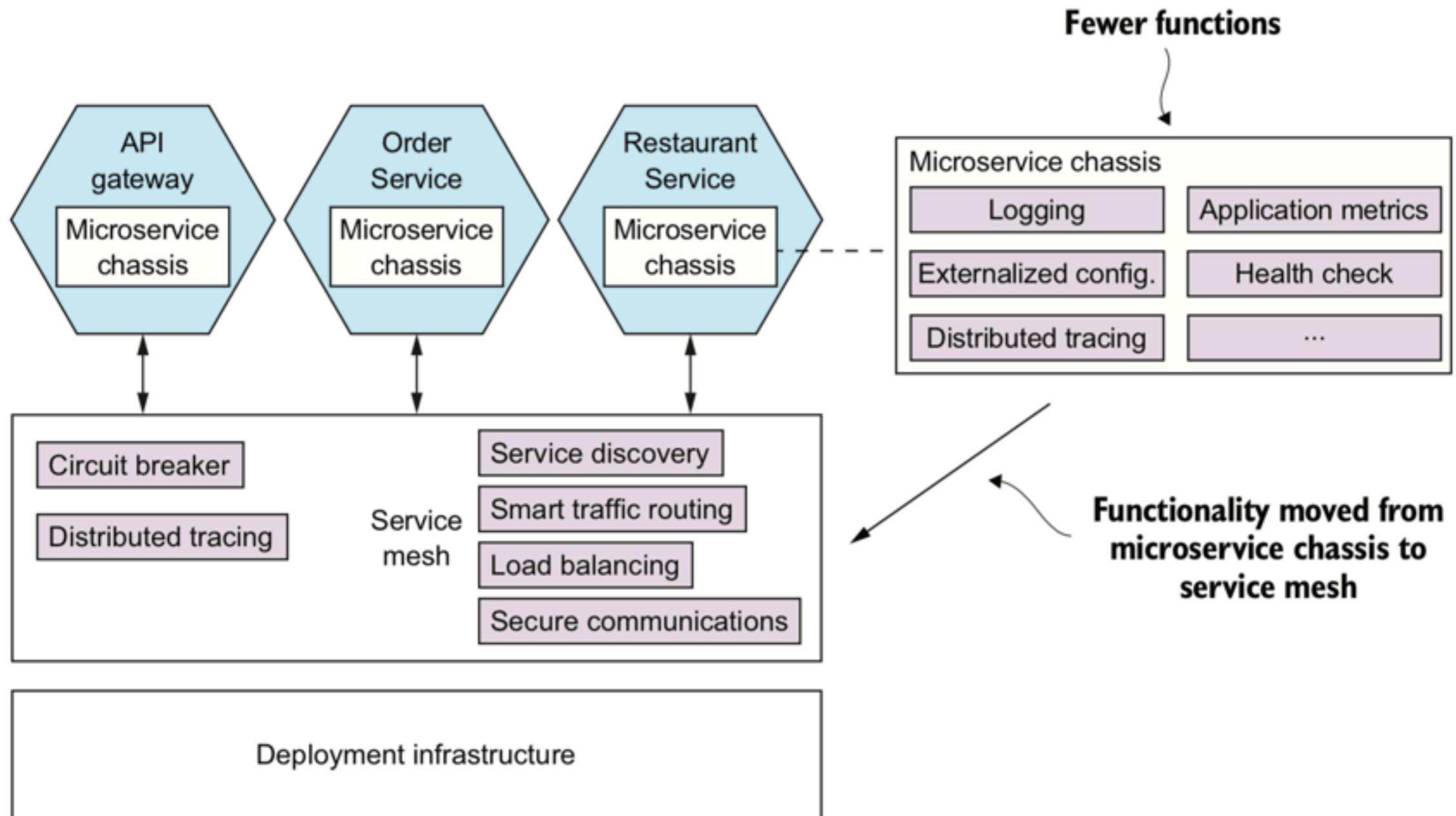


# Microservice framework

Programming language	Framework
Java	Spring boot Spring cloud
Golang	Go-kit Micro



# Service Mesh



# Service Mesh



## Istio

Connect, secure, control, and observe services.



## CONDUIT



## linkerd



# Module 3 : Deploy

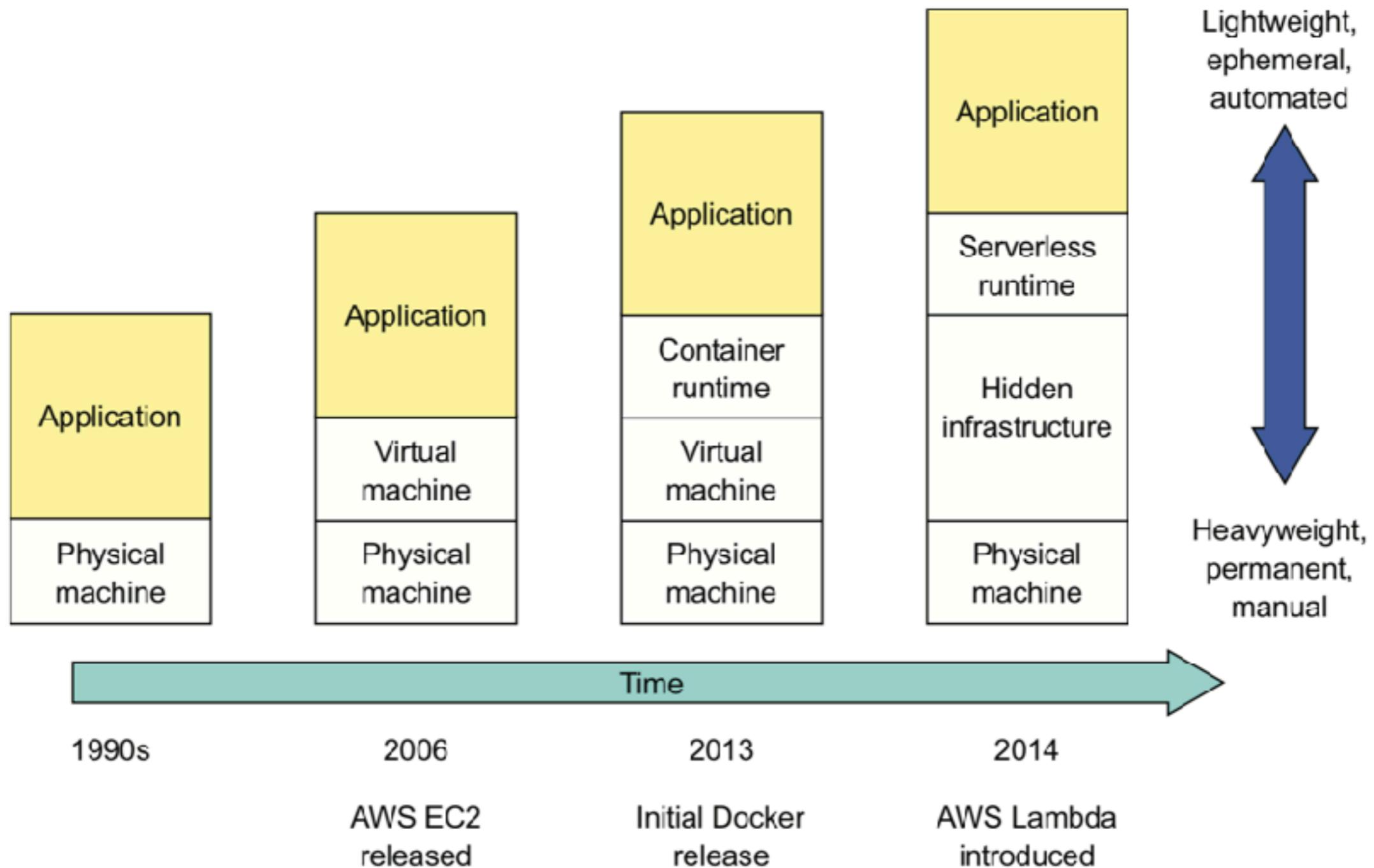




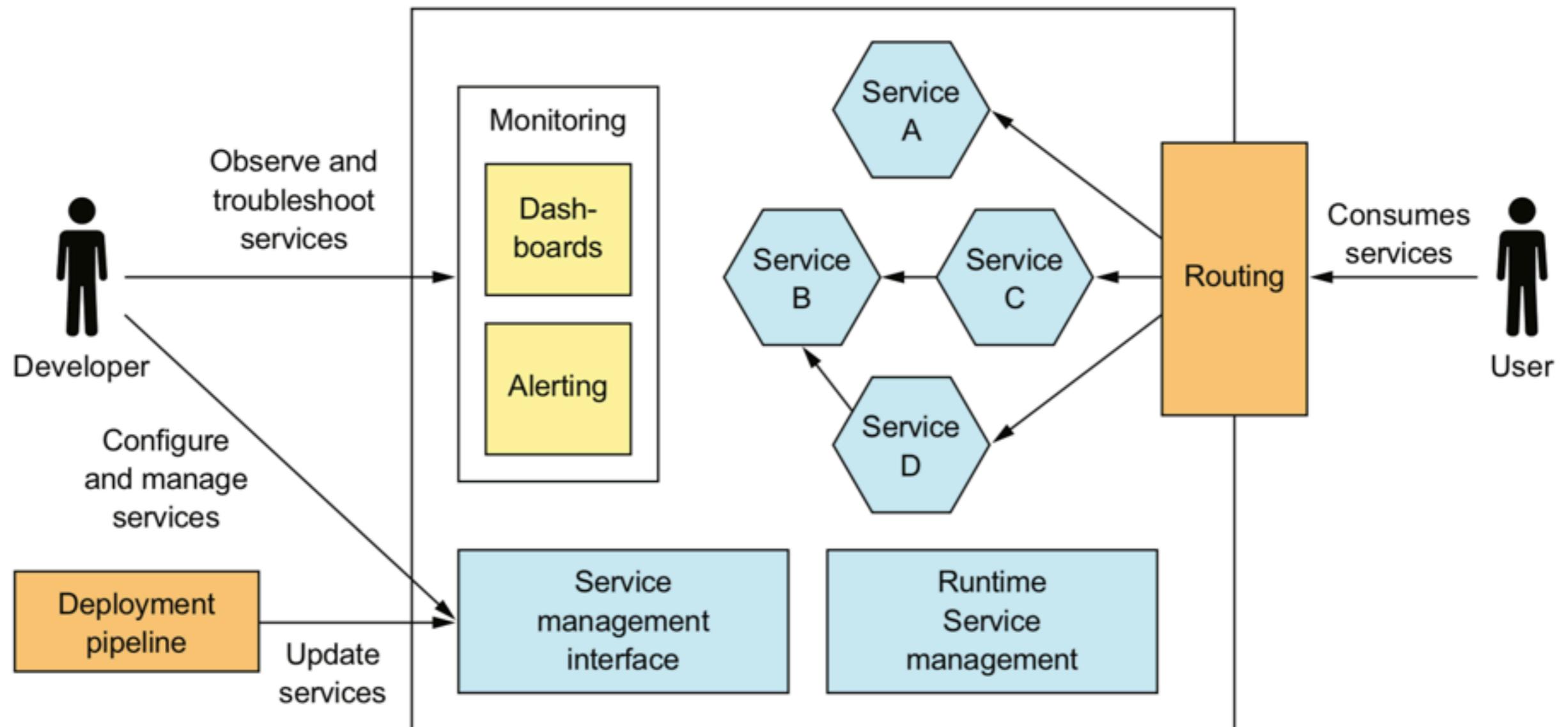
@redbadgerteam



# Infrastructure



# View of production environment



# **Deploy Microservices**

**Language-specific packaging**

**Virtual Machine (VM)**

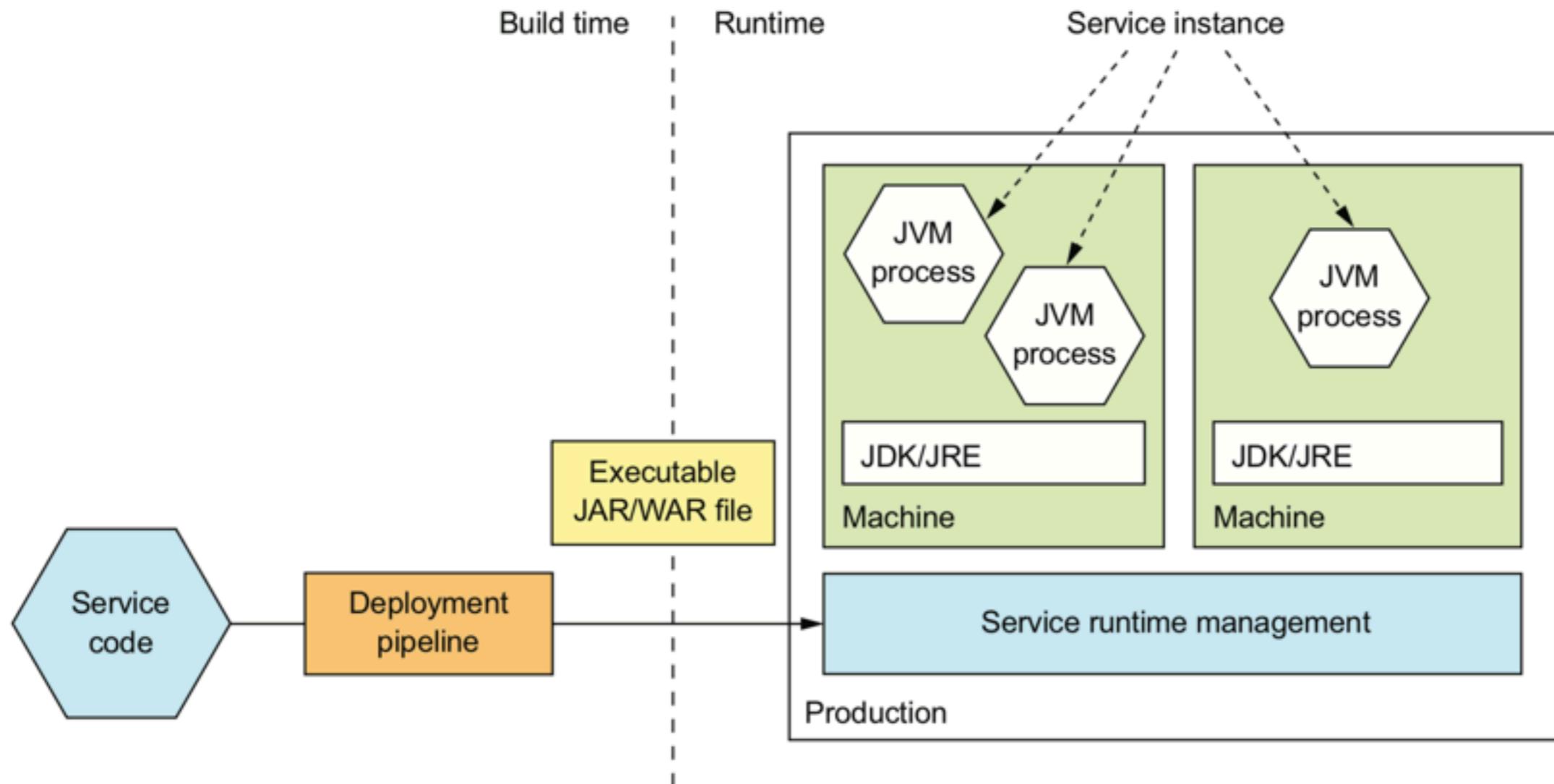
**Container**

**Kubernetes**

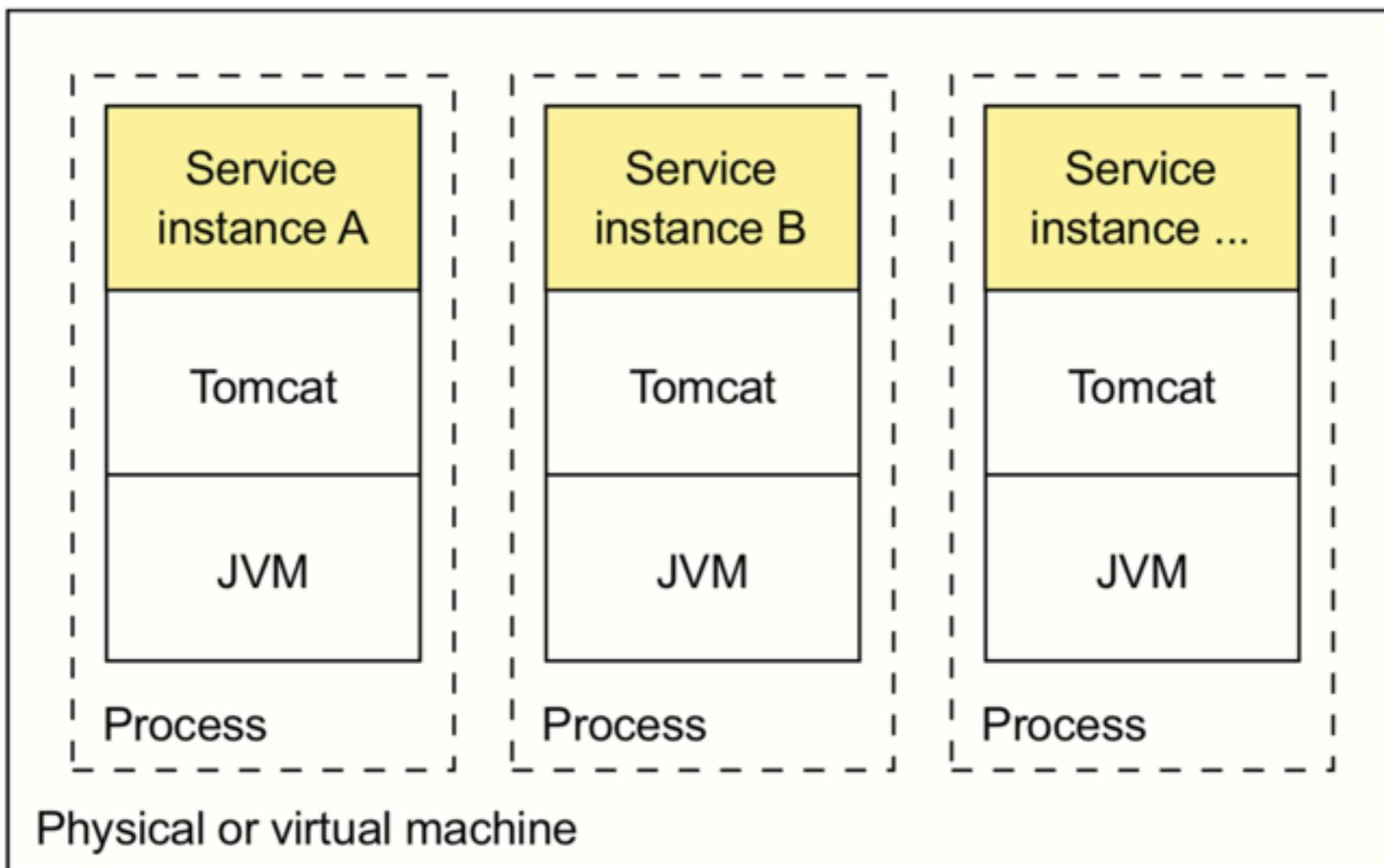
**Serverless/FaaS**



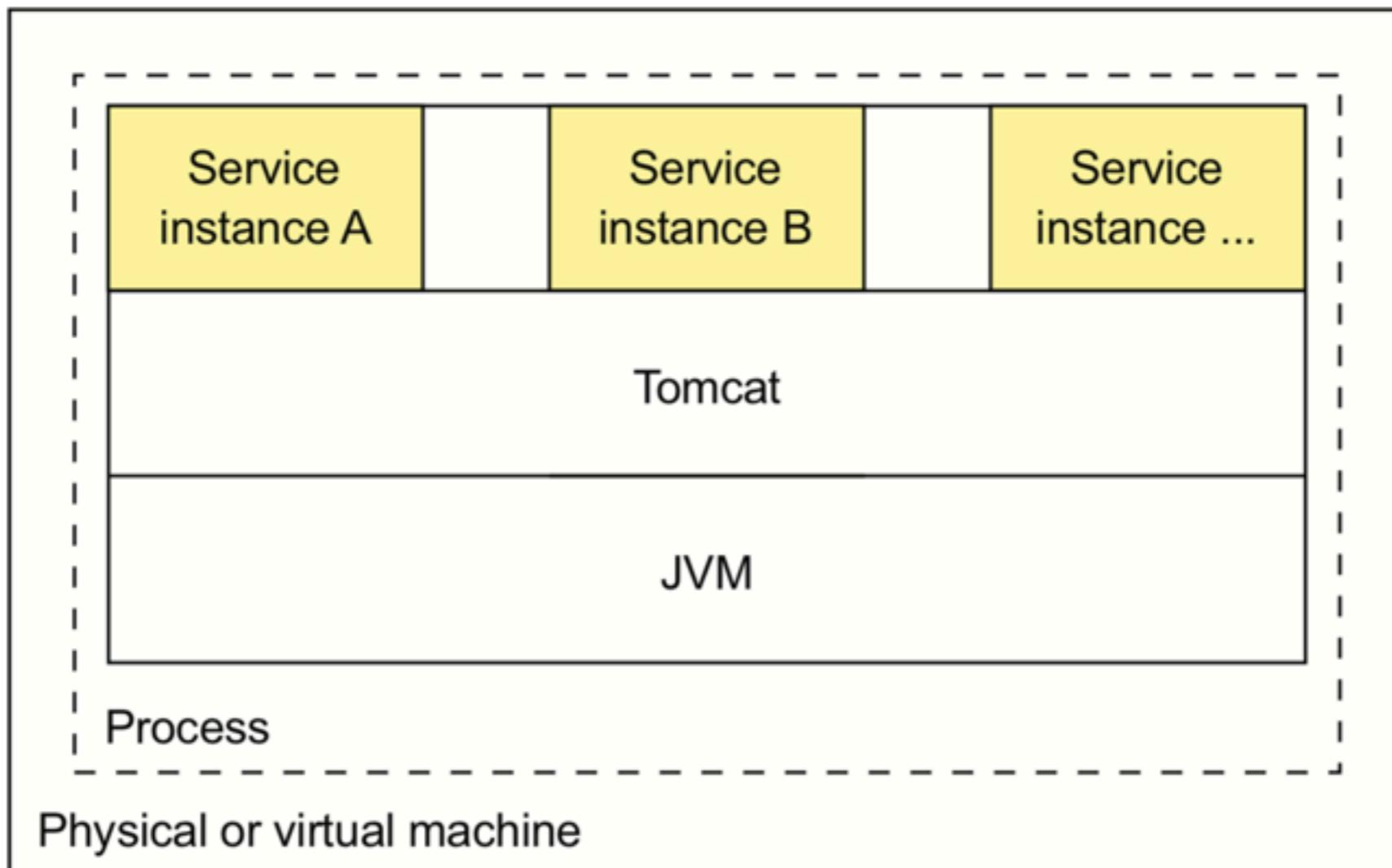
# 1. Language-specific packaging



# Multiple services on same machine



# Multiple services on same process



# Benefits

Fast deployment

Efficient resource utilization

Service instances's resources are constrained



# Drawbacks

Lack of encapsulation of technology stack

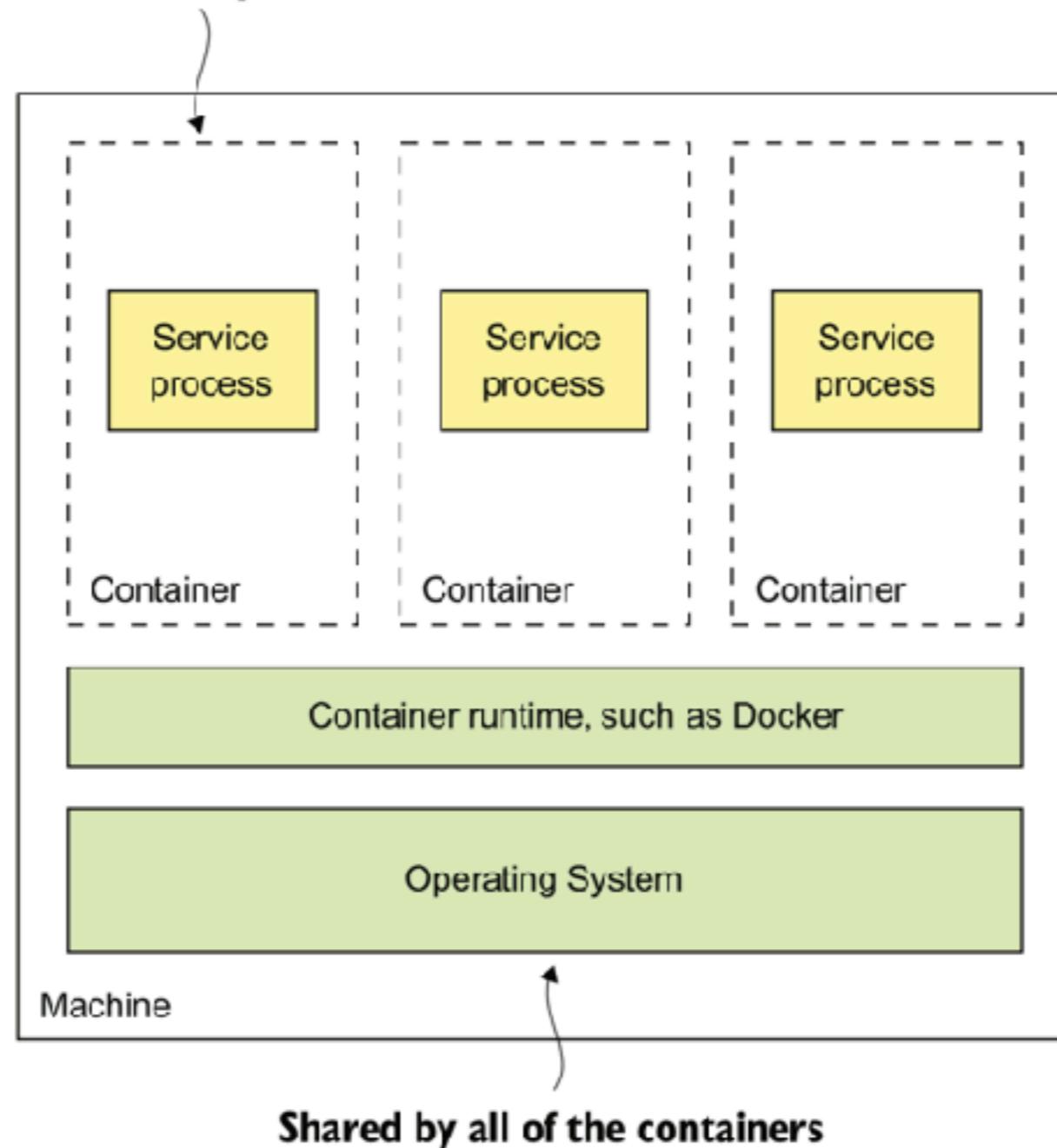
No ability to constrain resources of service

Lack of isolation

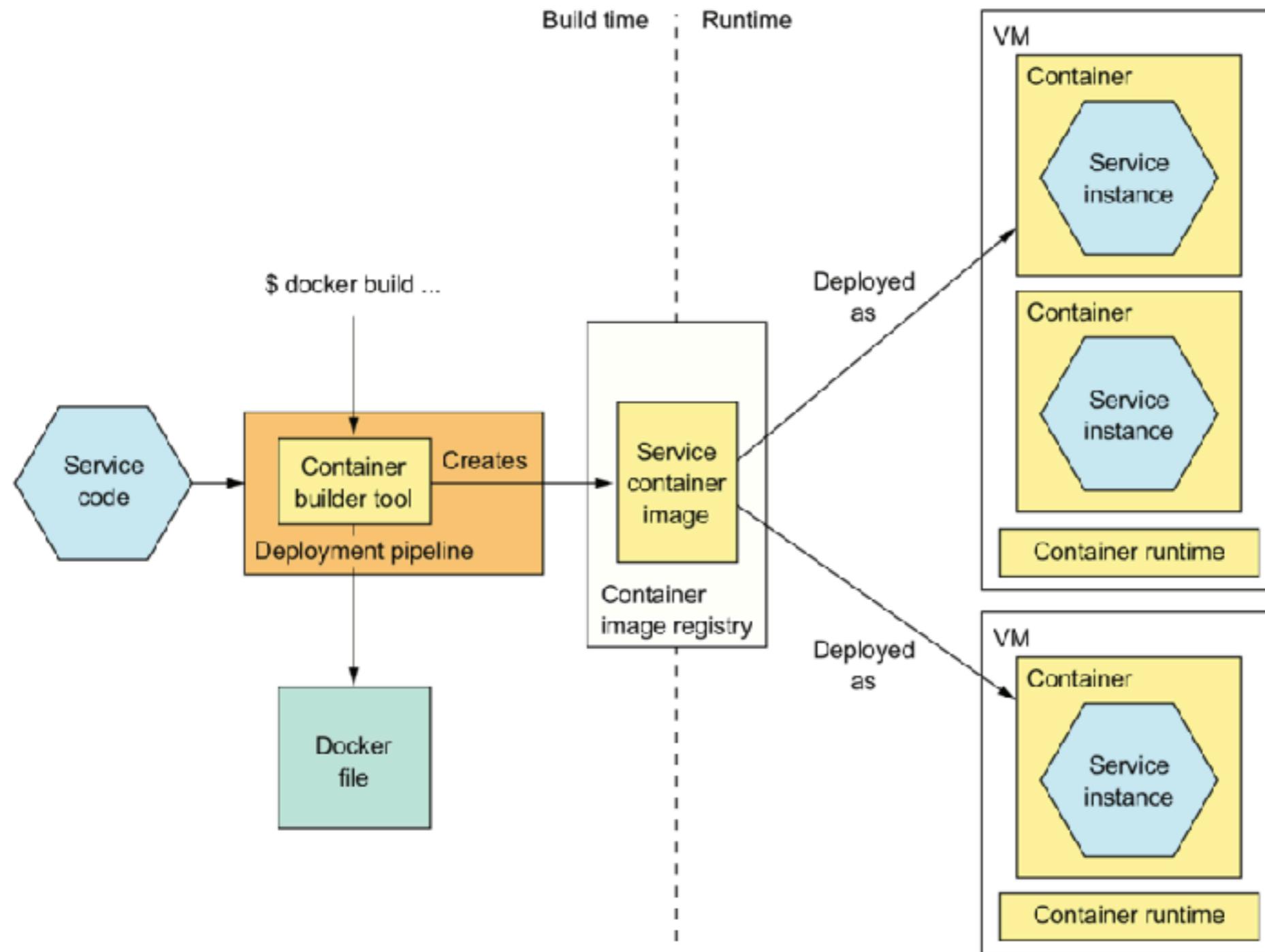


# 2. Working with container

**Each container is a sandbox  
that isolates the processes.**



# Deployment with container



# Deploy services with Docker

Build a docker image

Push docker image to a registry

Run docker container

Working with docker-compose



# Benefits

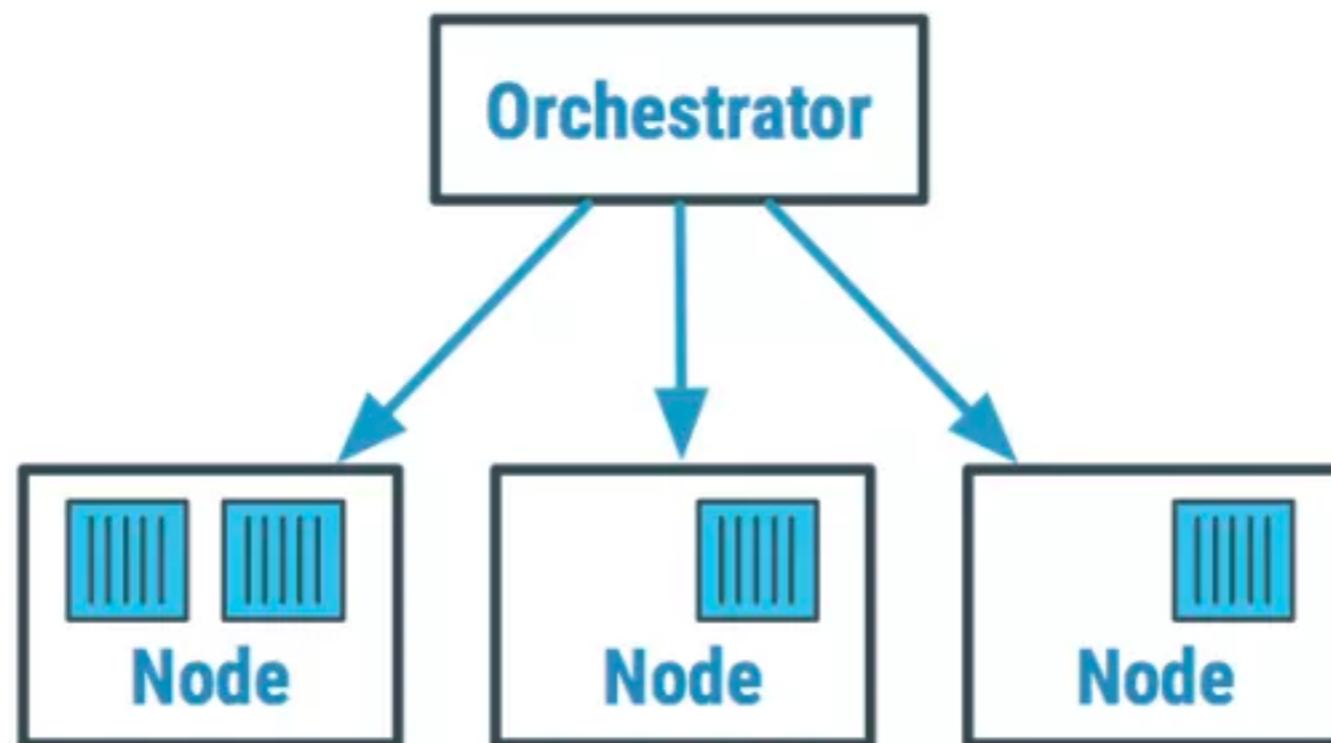
Encapsulate technology stack

Service instances are isolated

Service instances's resources are constrained



# Container Orchestration ?



# Orchestration tools

Configuration Management



CI/CD orchestration



Container orchestration



Cloud-specific orchestration



PaaS orchestration



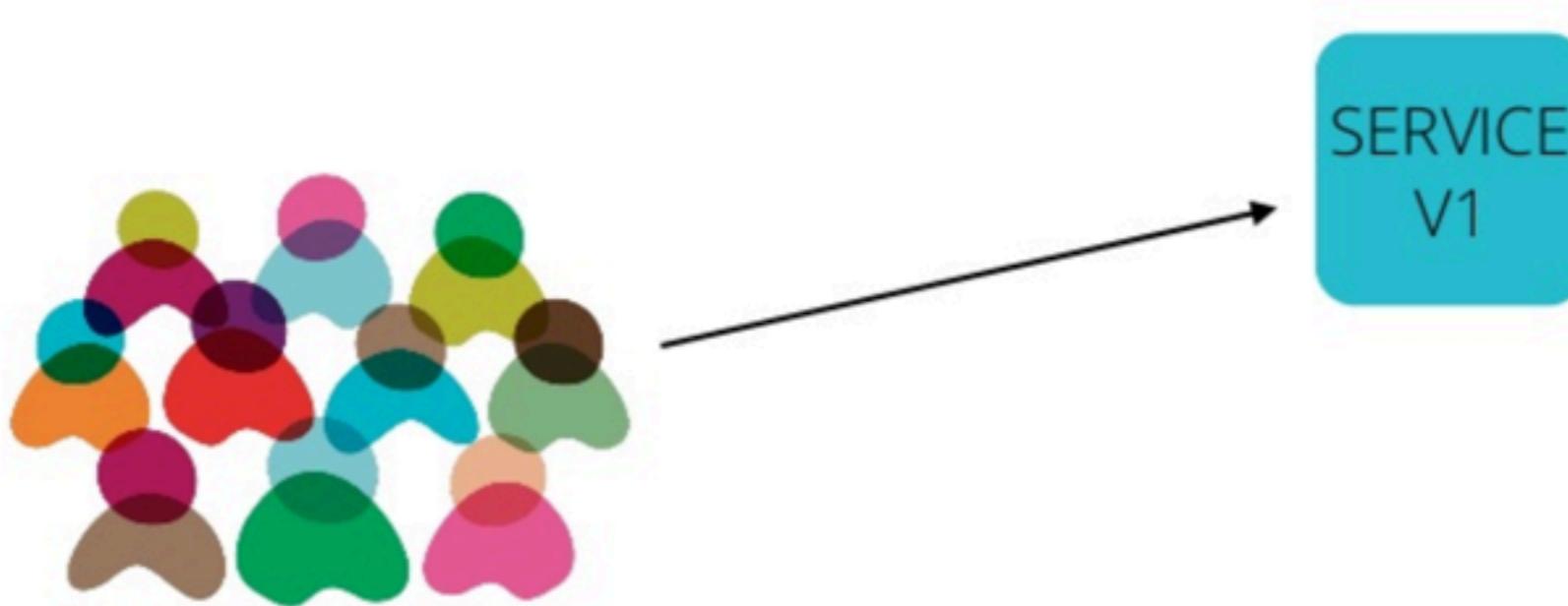
# Deployment strategies



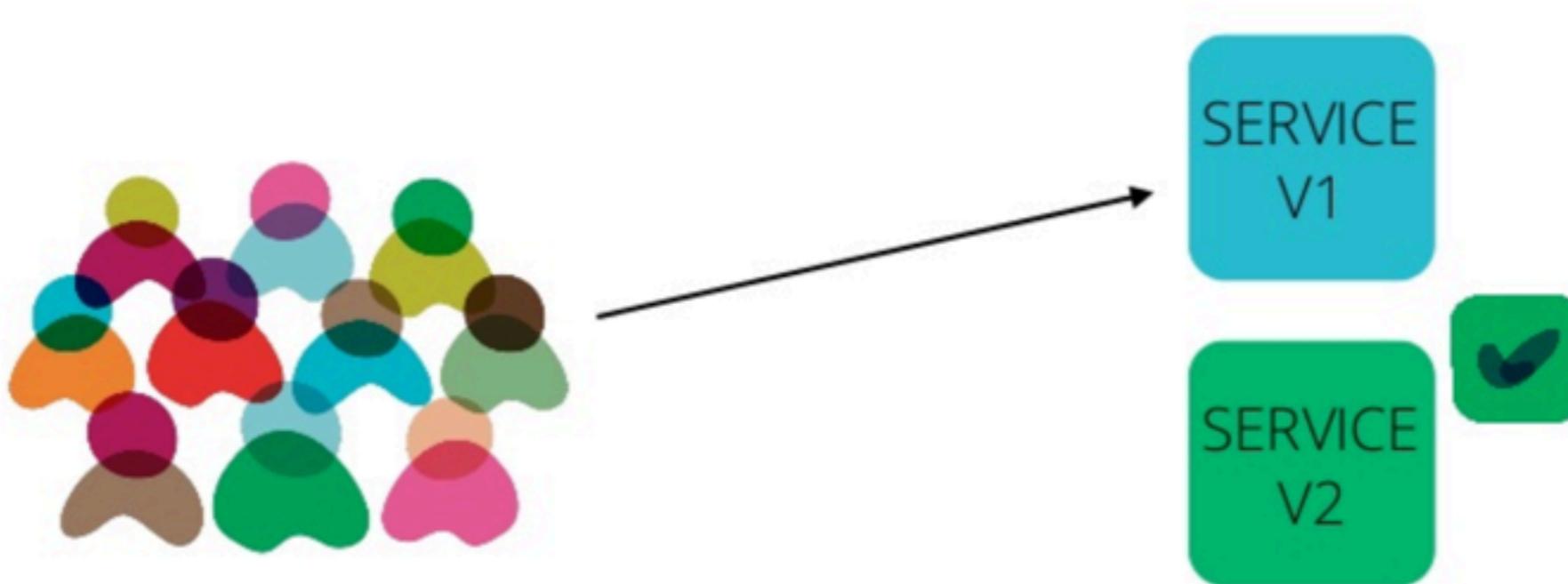
# Blue Green Deployment



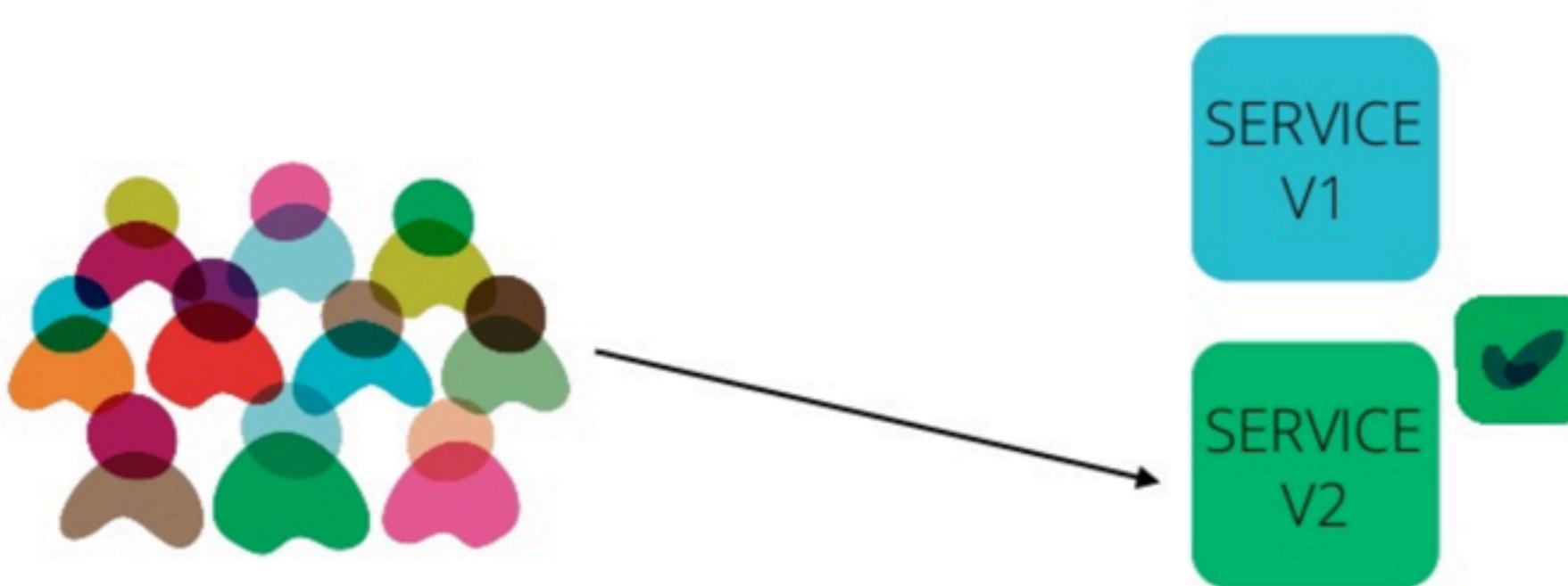
# Blue Green Deployment



# Blue Green Deployment



# Blue Green Deployment



# Canary Release



# Canary Release



# Canary Release



# Canary Release

