Microservices

...

Calin-Ioan Nicoara
Senior Software Developer and Trainer

esolutions Academy

Let’s get to know each other

Outline

- Microservice architecture
- Intercommunication
- Security
- Testing strategies
- Deployment
- Monitoring

What are Microservices?

A microservice application is a collection of independent services, each doing one thing well, that work together to perform more complex operations.

Before Microservices

Monolithic
Tight coupling

All elements must agree on each change

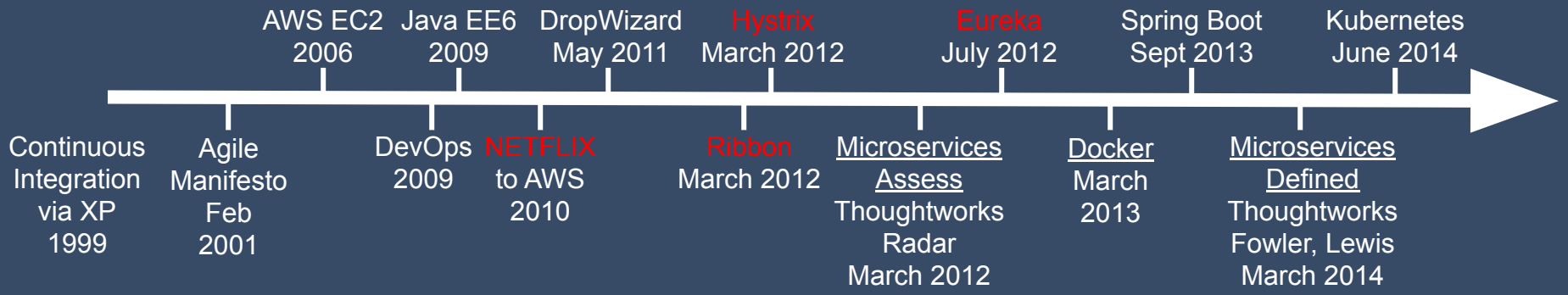
Traditional SOA
Looser coupling

Elements must be coordinated each other

Microservice
Decoupled

Elements can be changed or added without prior coordination with others

Short history of microservices



Microservice benefits

Autonomy

Autonomous team with isolated implementation

Speed of change

Build and deploy small independent services

Scale

Functional and Team Scaling

Tech Diversity

Allows try-out of new technologies

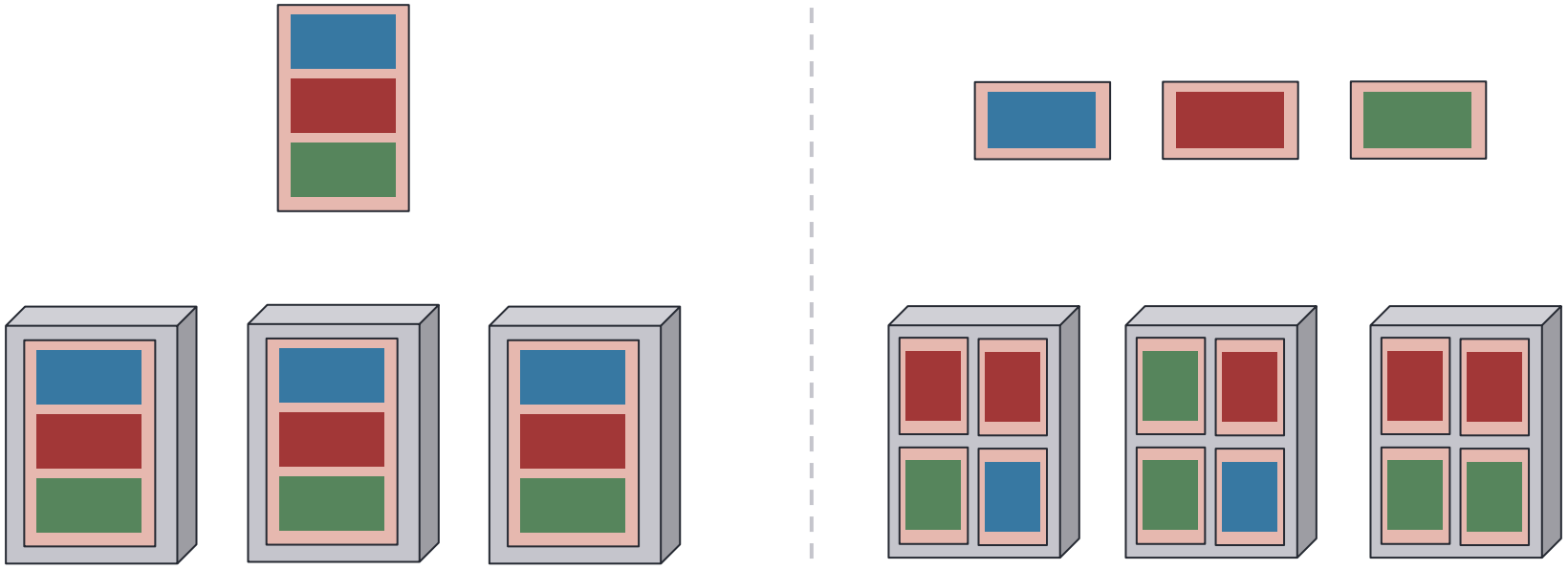
Intelligible

Lower cognitive load

Microservices

the main characteristics

Scalability & High-availability



Organized around business capabilities

UI

Server

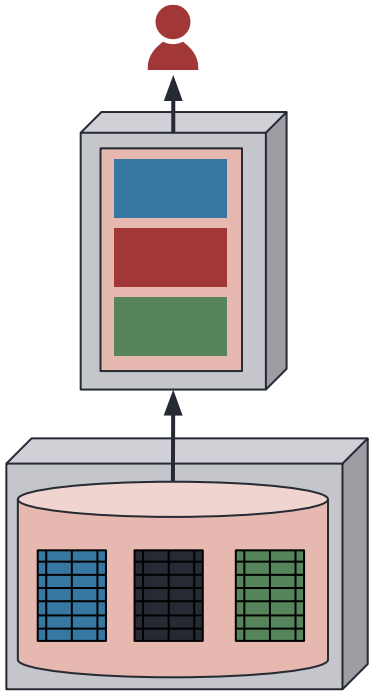
DBA

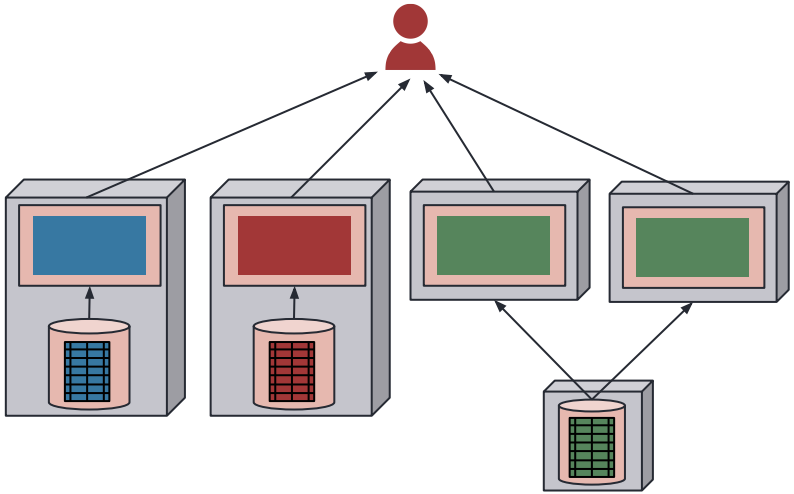
Shipping

Orders

Catalog

Autonomous (from design to deployment)





Disadvantages
of distributed computing

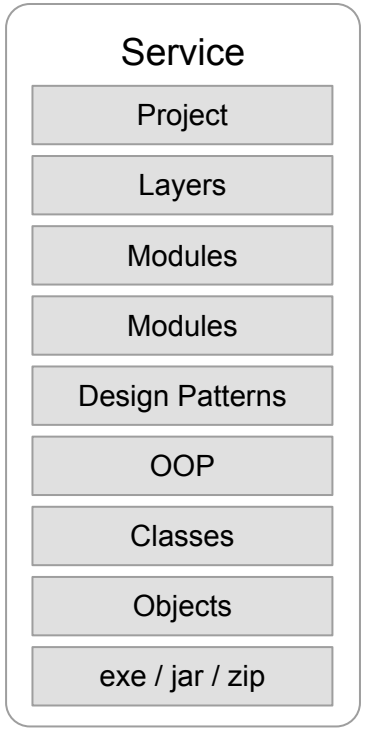
- Latency
- Partial failure
- Infrastructure

Designing Microservices

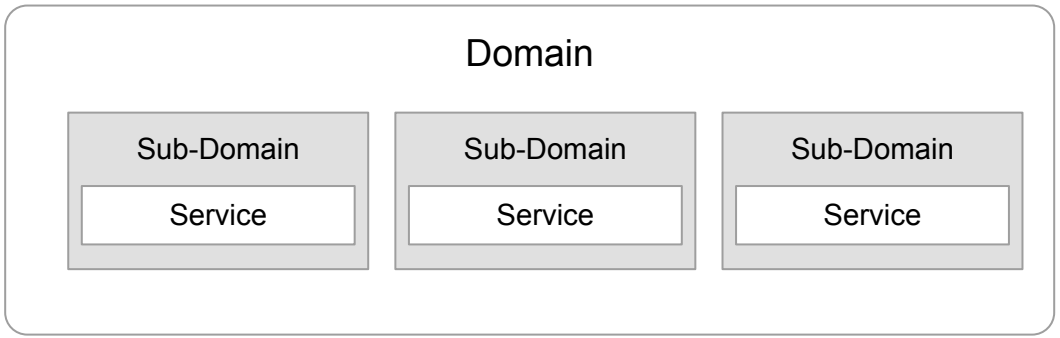
- **Size**
  Single Responsibility Principle
- **Scope**
  Agile / Independent development and deployment
- **Capabilities**
  Bounded Context in Domain-Driven-Design

Domain Driven Design

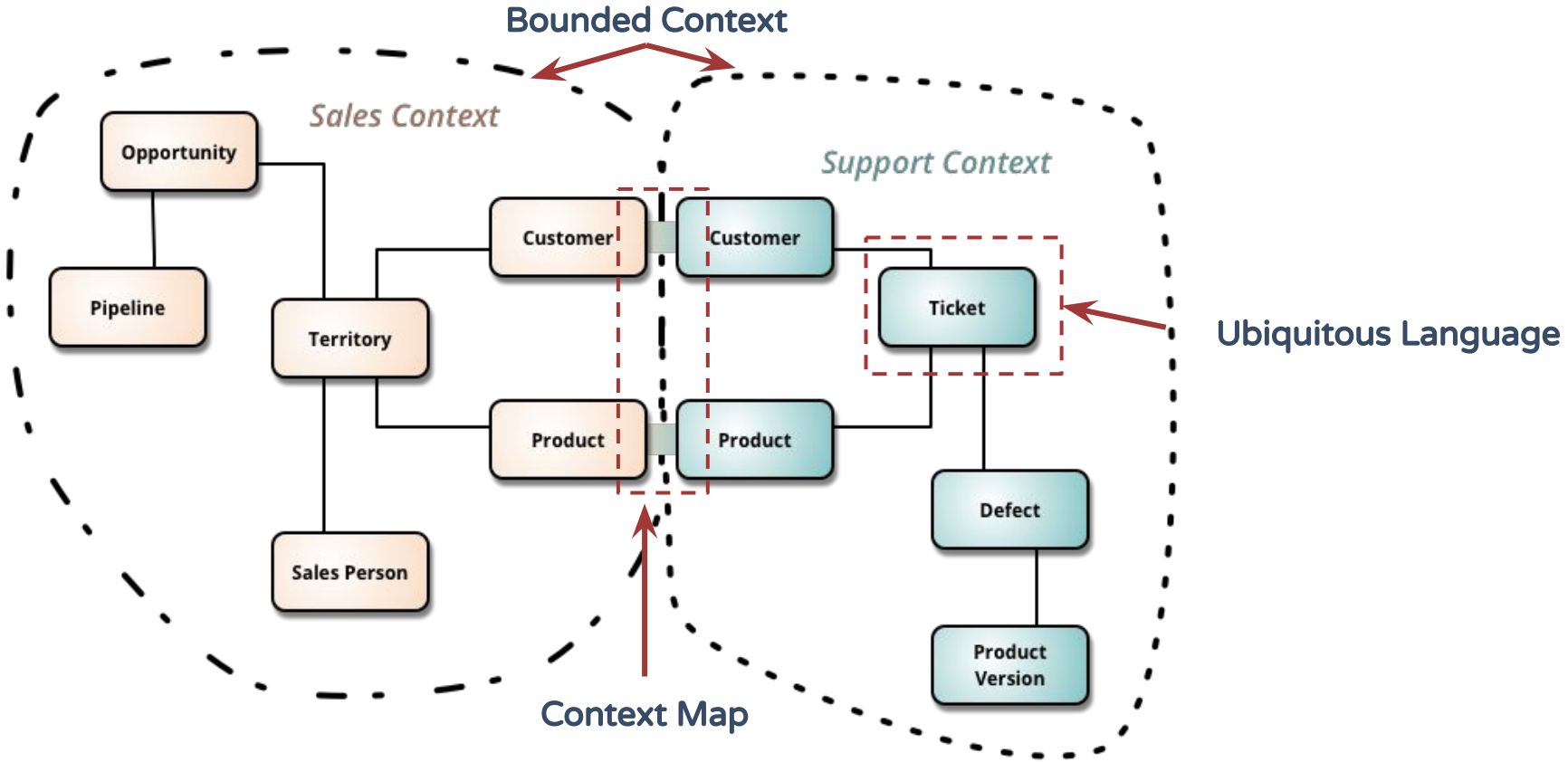
Tactical Design



Strategic Design



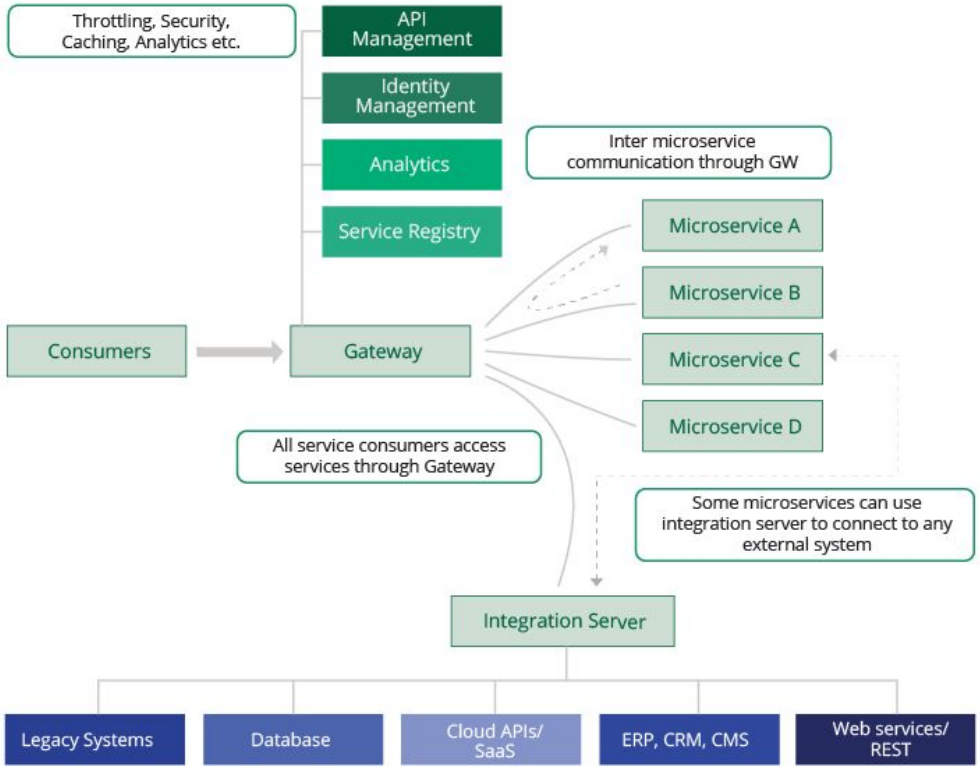
DDD - Strategic Design



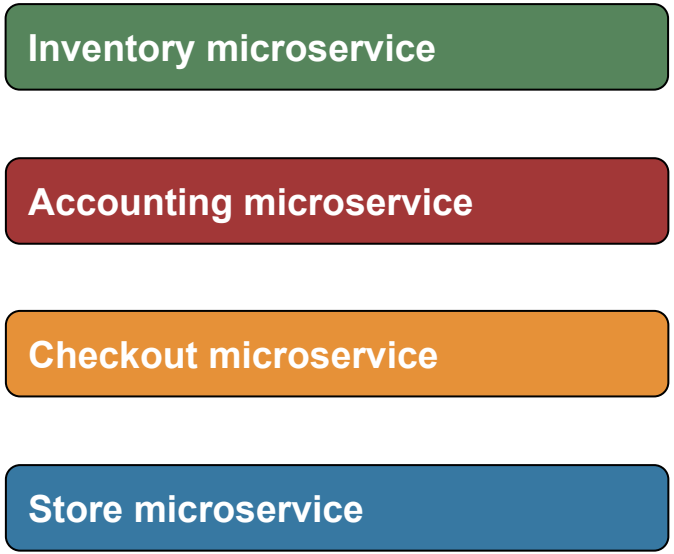
Microservices in Practice

eCommerce

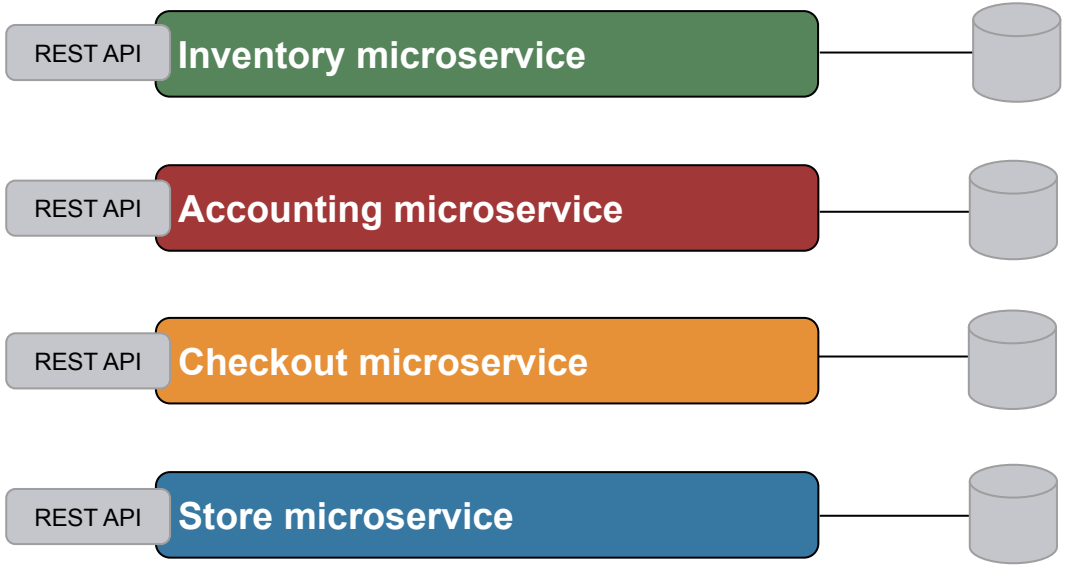
Microservices Architecture



eCommerce Microservices



eCommerce Microservices

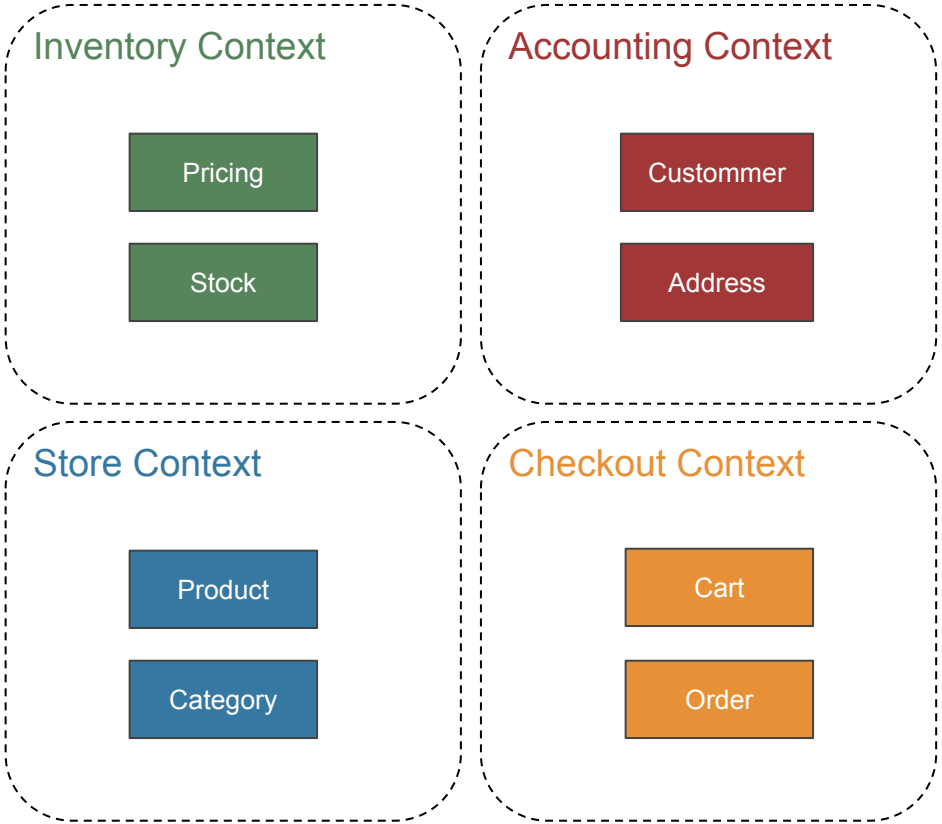


REST

HTTP verbs

- Get
- Post
- Put
- Delete
- Patch

Business Capabilities

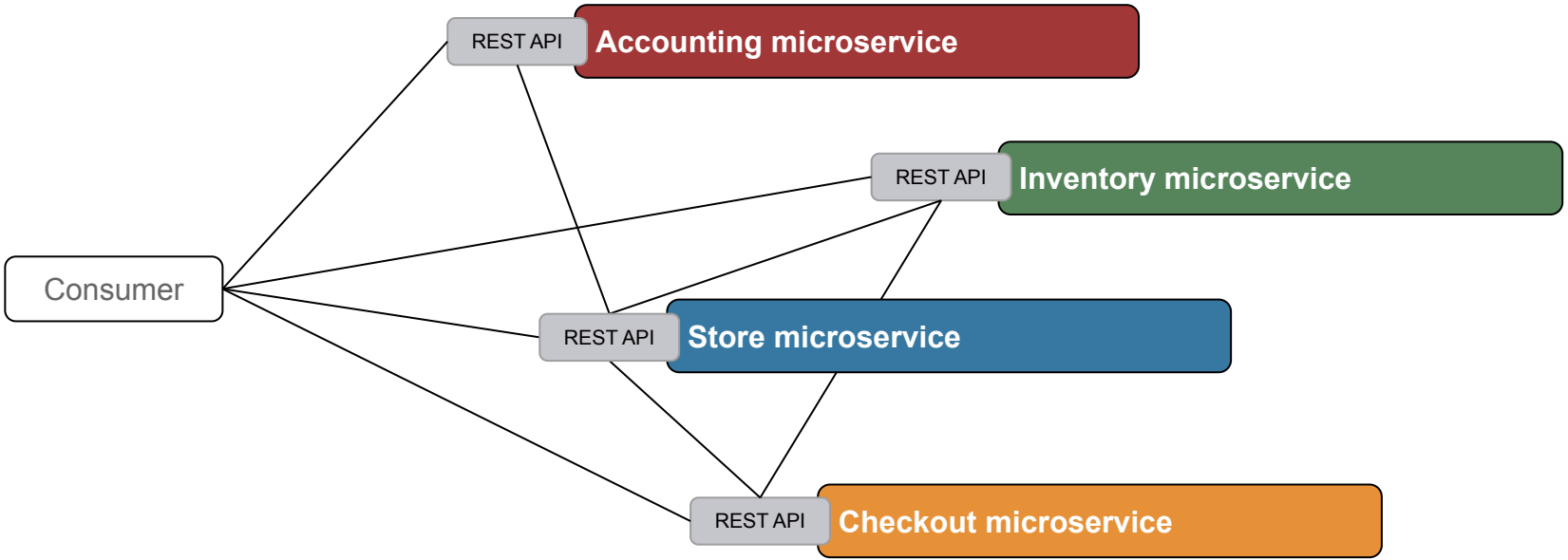


Communication

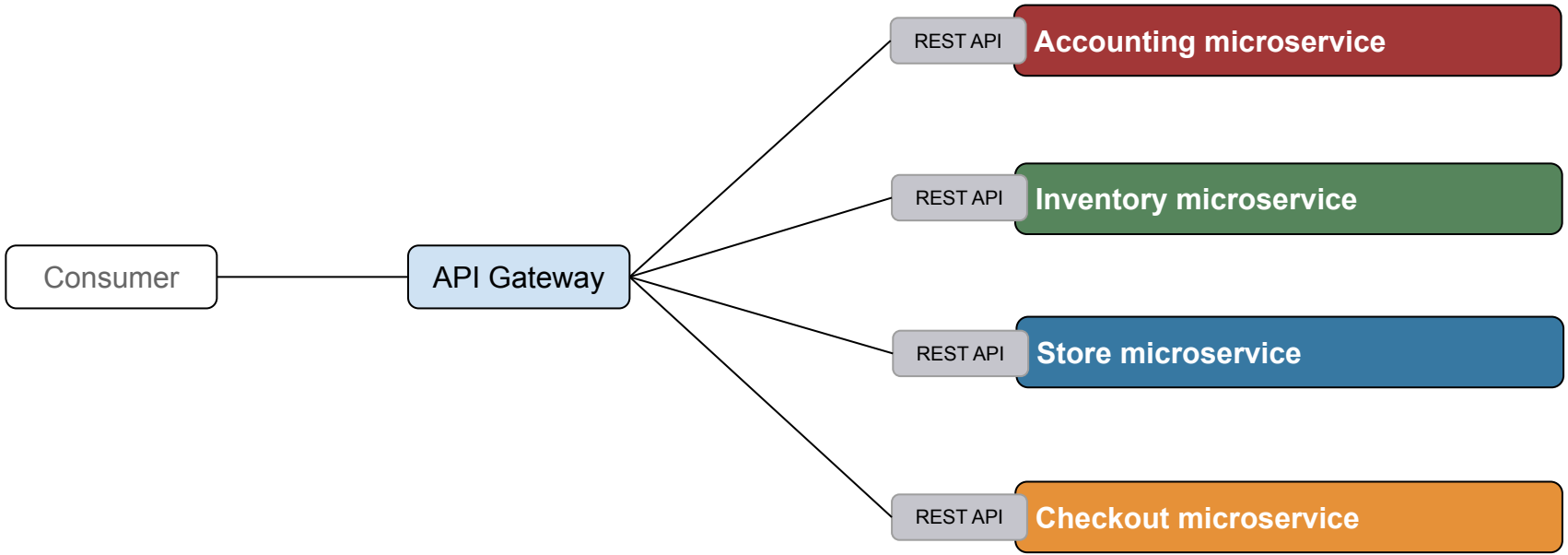
Interaction Styles

	One-to-One	One-to-Many
Synchronous	Request / Response	--
Asynchronous	Notification	Publish / Subscribe
	Request / Async response	Publish / Async response

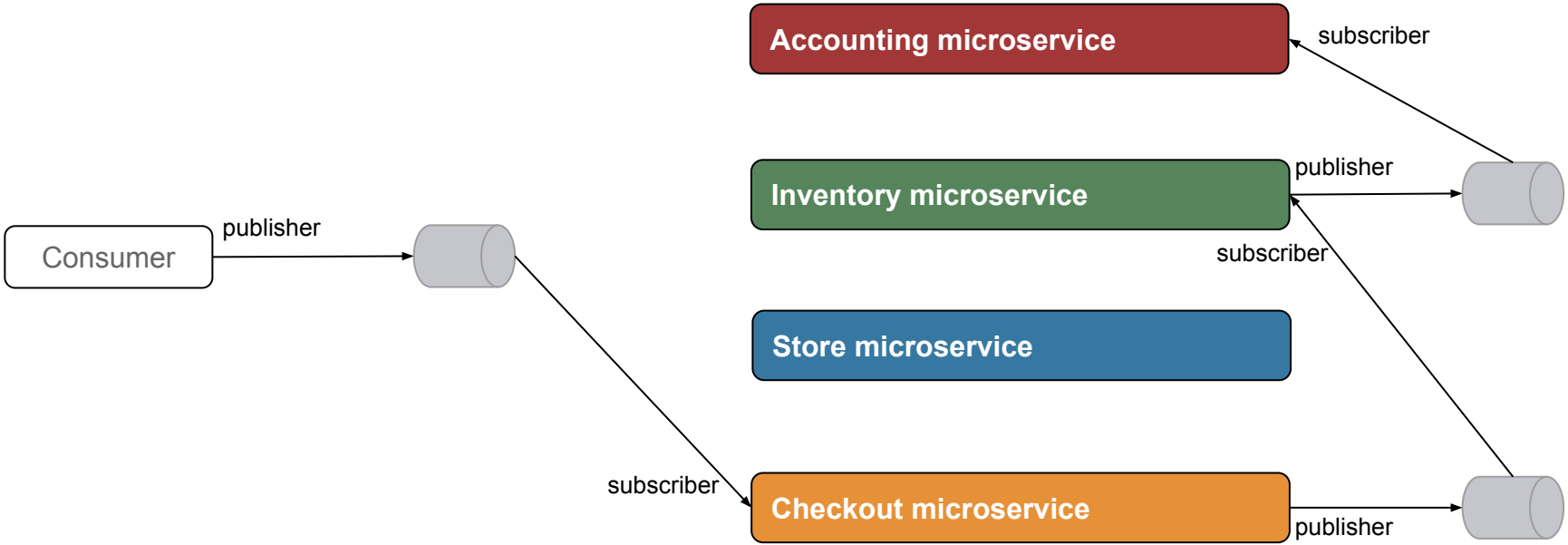
Point-to-point style



Api-Gateway style

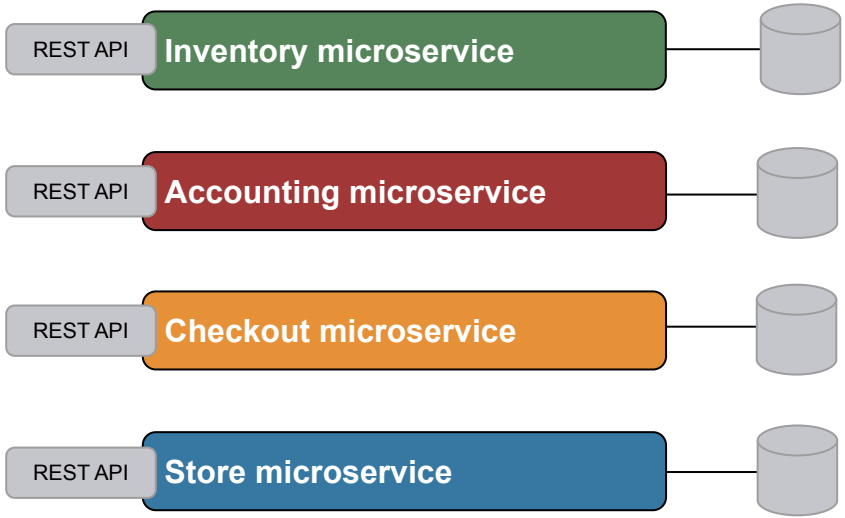


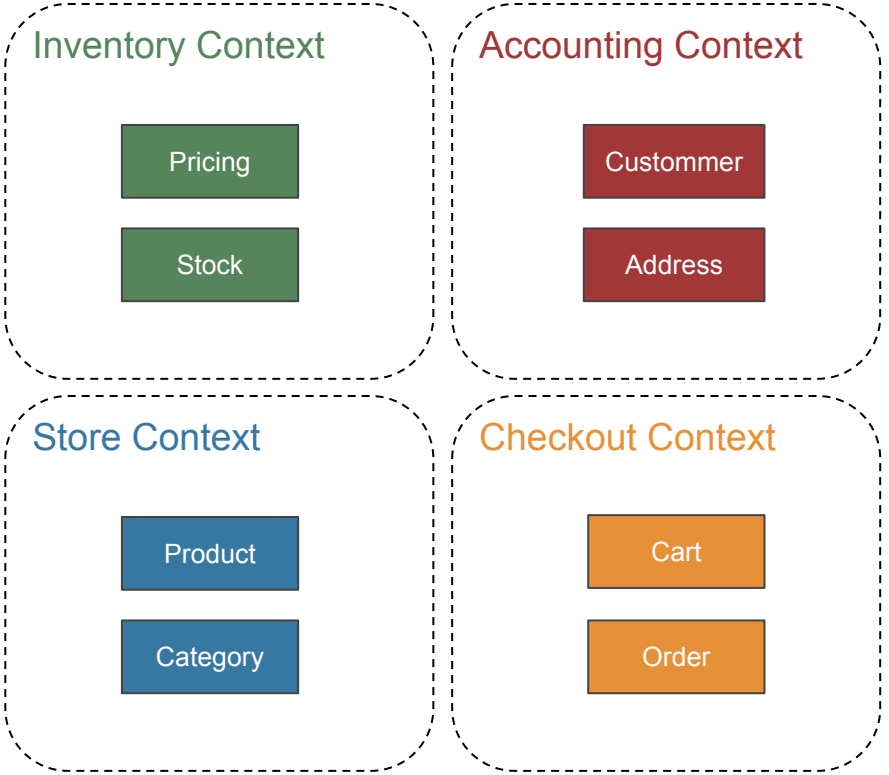
Message-Broker style



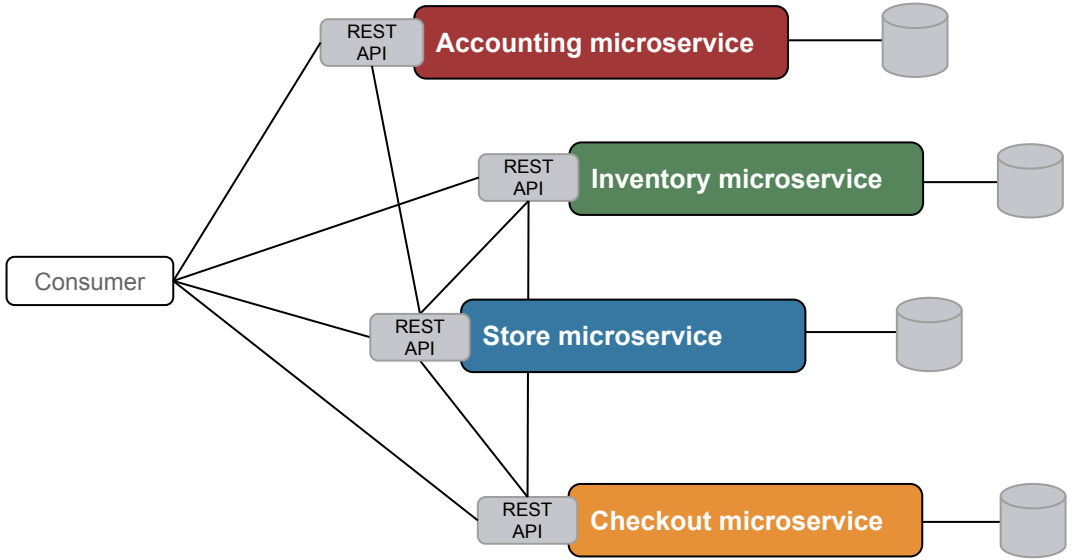
Coding time

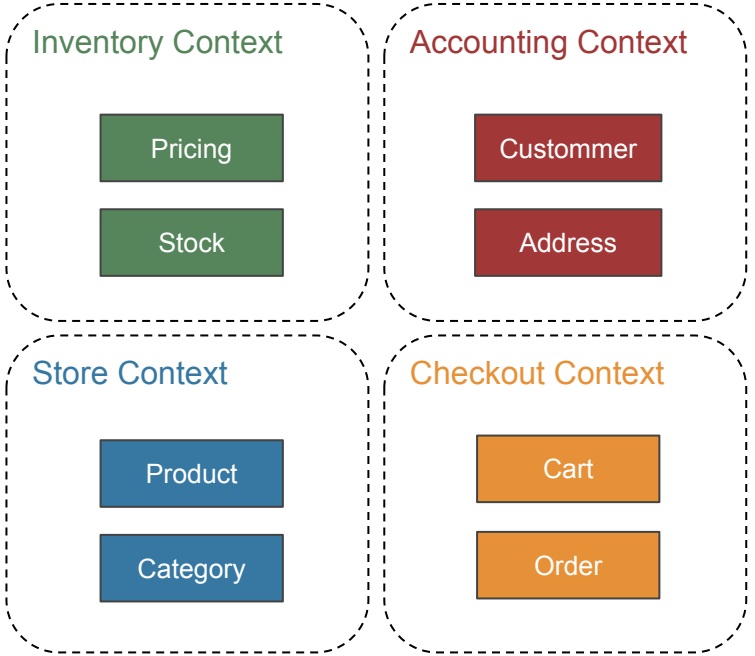
Coding time


REST API
Inventory microservice
REST API
Accounting microservice
REST API
Checkout microservice
REST API
Store microservice


Inventory Context
Pricing
Stock
Accounting Context
Custommer
Address
Store Context
Product
Category
Checkout Context
Cart
Order

Coding time

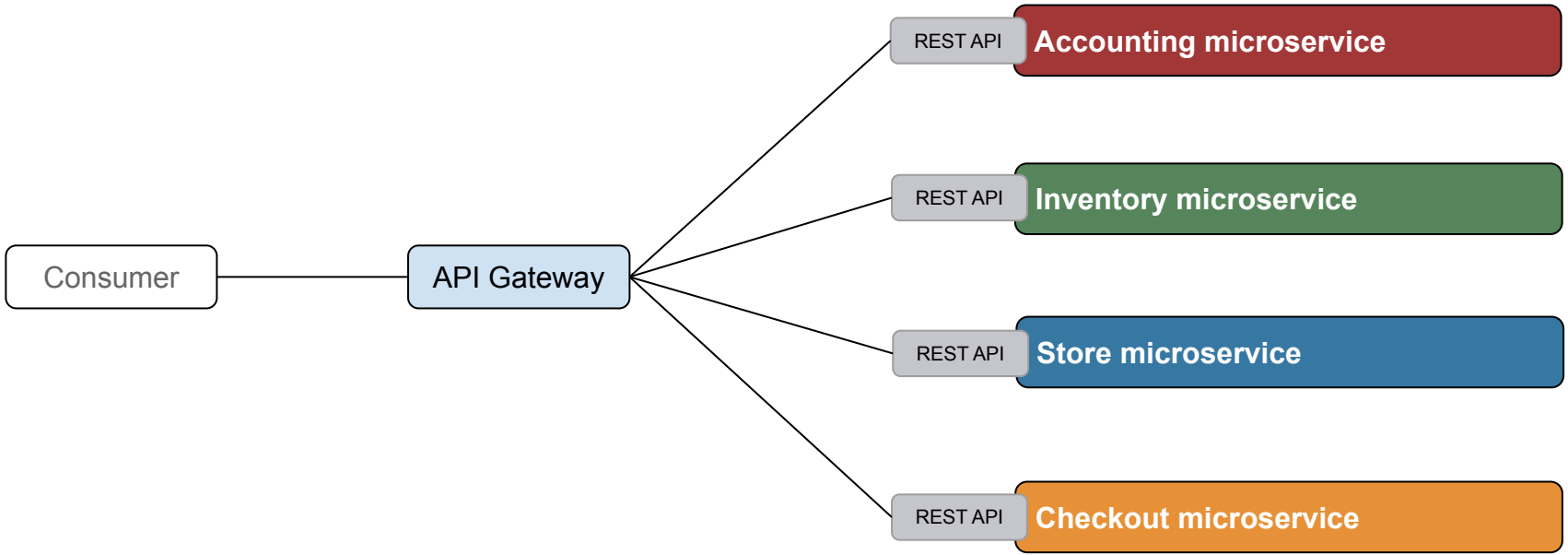

REST API
Accounting microservice
REST API
Inventory microservice
Consumer
REST API
Store microservice
REST API
Checkout microservice


Inventory Context
Pricing
Stock
Accounting Context
Custommer
Address
Store Context
Product
Category
Checkout Context
Cart
Order

Spring Cloud Gateway

- Front door
- Edge service
- Routing and Filtering
- Monitoring
- Security

Coding time


Consumer
API Gateway
REST API
Accounting microservice
REST API
Inventory microservice
REST API
Store microservice
REST API
Checkout microservice

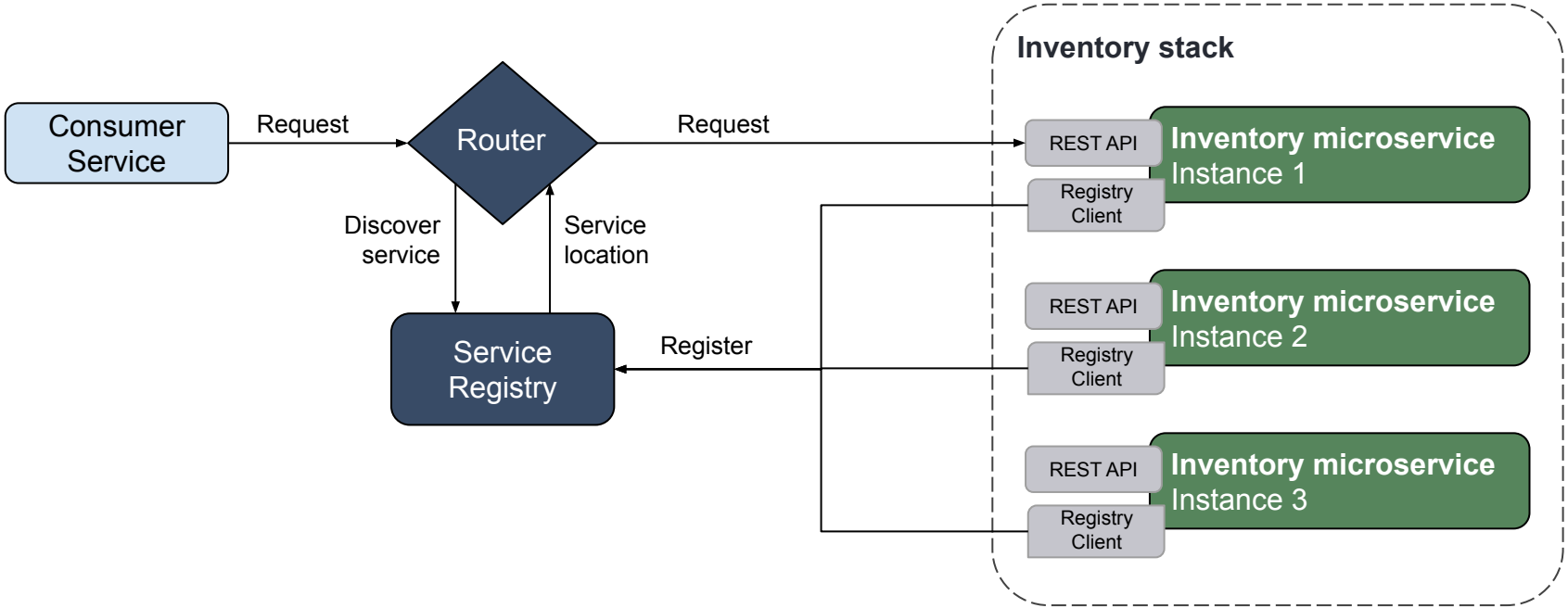
Service Discovery

Server-side and Client-side

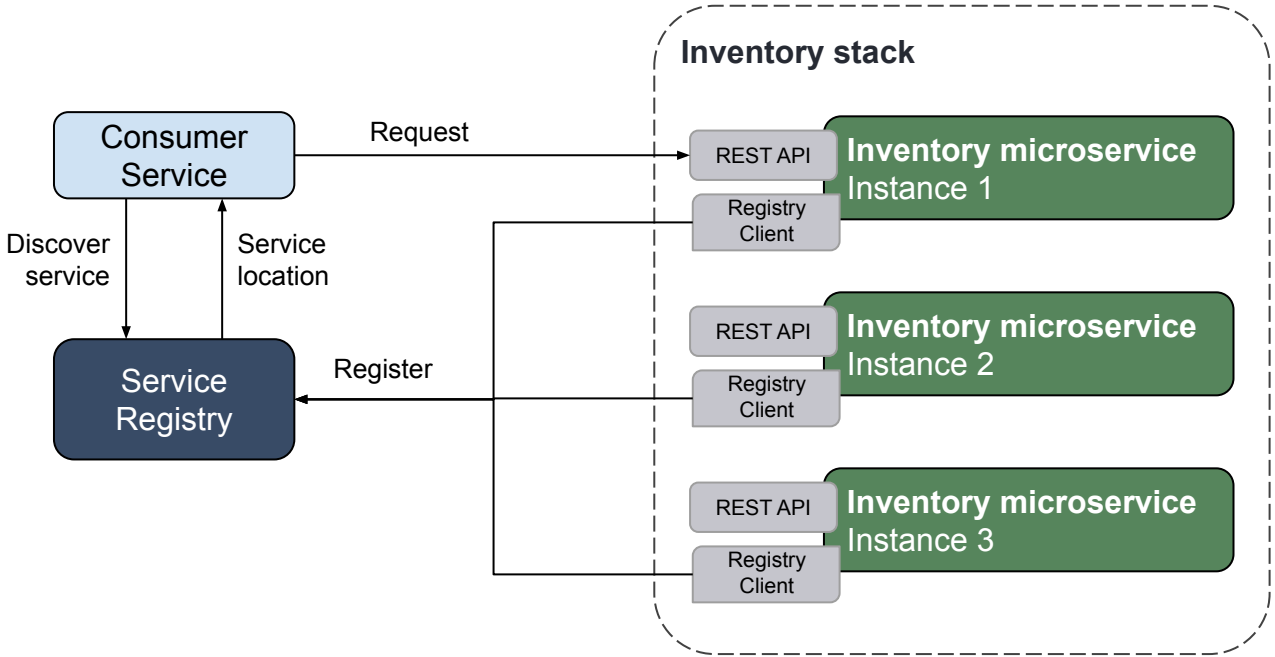
Why use
Service Discovery

- Dynamically network location
- Autoscaling
- Failures
- Upgrades

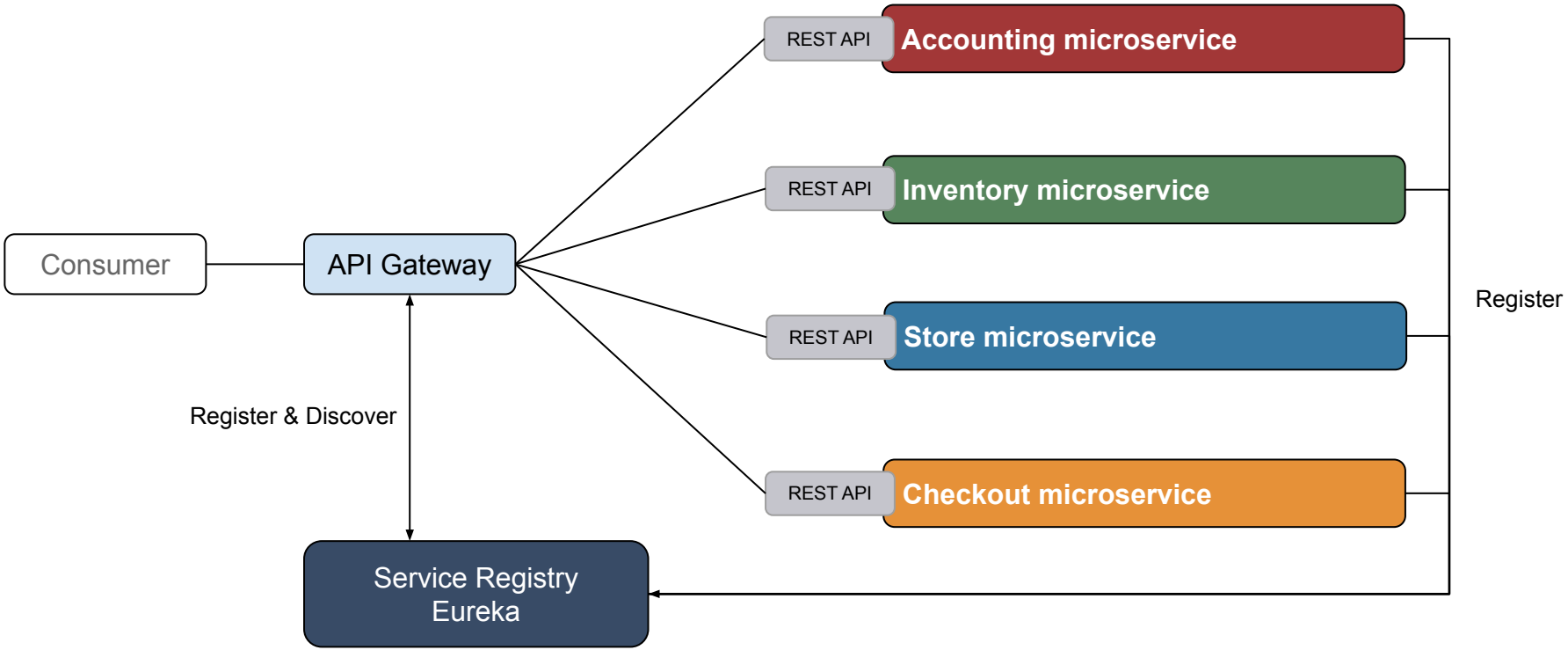
Server-Side Discovery



Client-Side Discovery



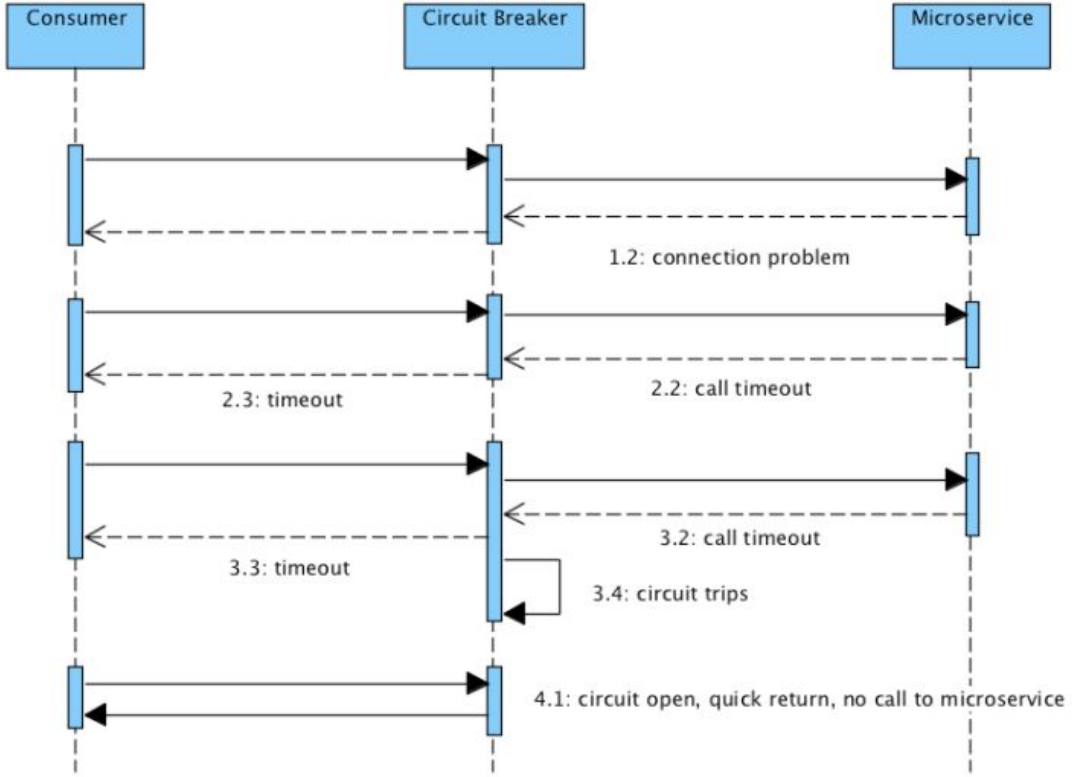
Coding time


REST API
Accounting microservice
REST API
Inventory microservice
REST API
Store microservice
REST API
Checkout microservice
Consumer
API Gateway
Register & Discover
Service Registry
Eureka
Register

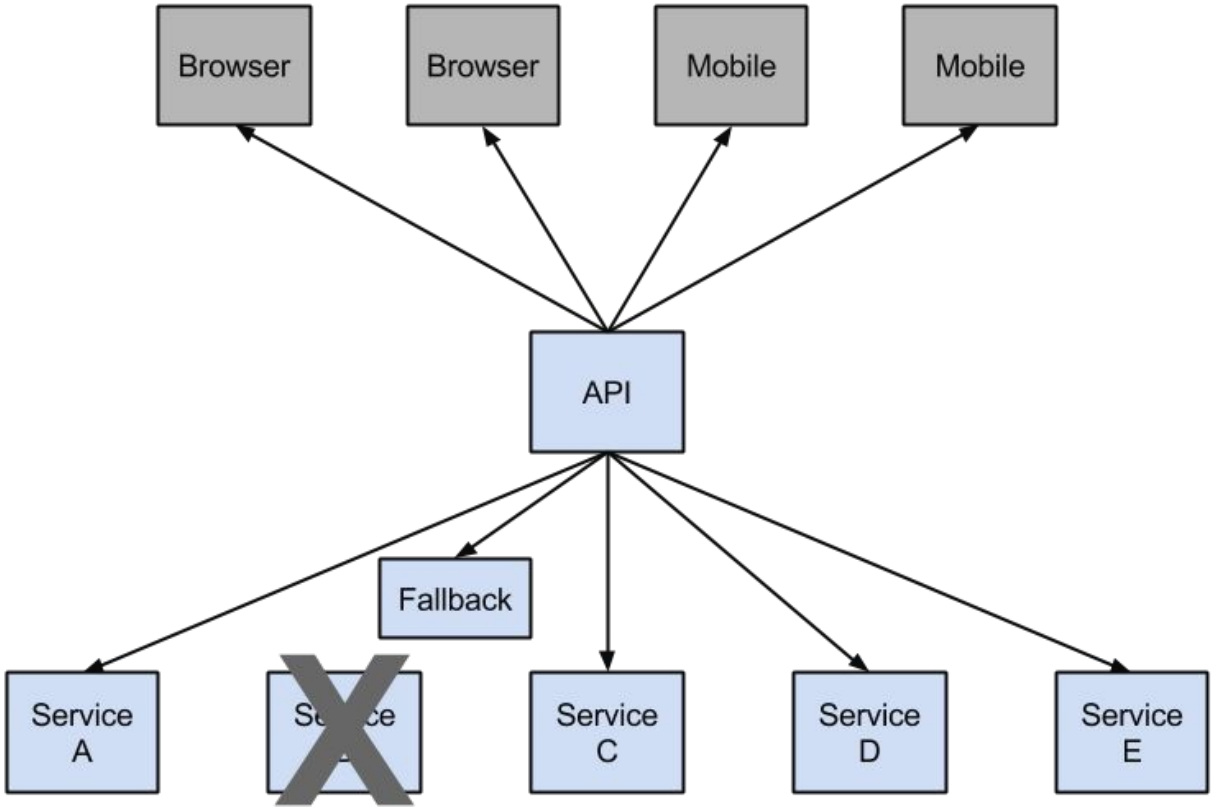
Fault-Tolerance

Distributed Systems

Circuit Breaker



Hystrix Fallback



Feign

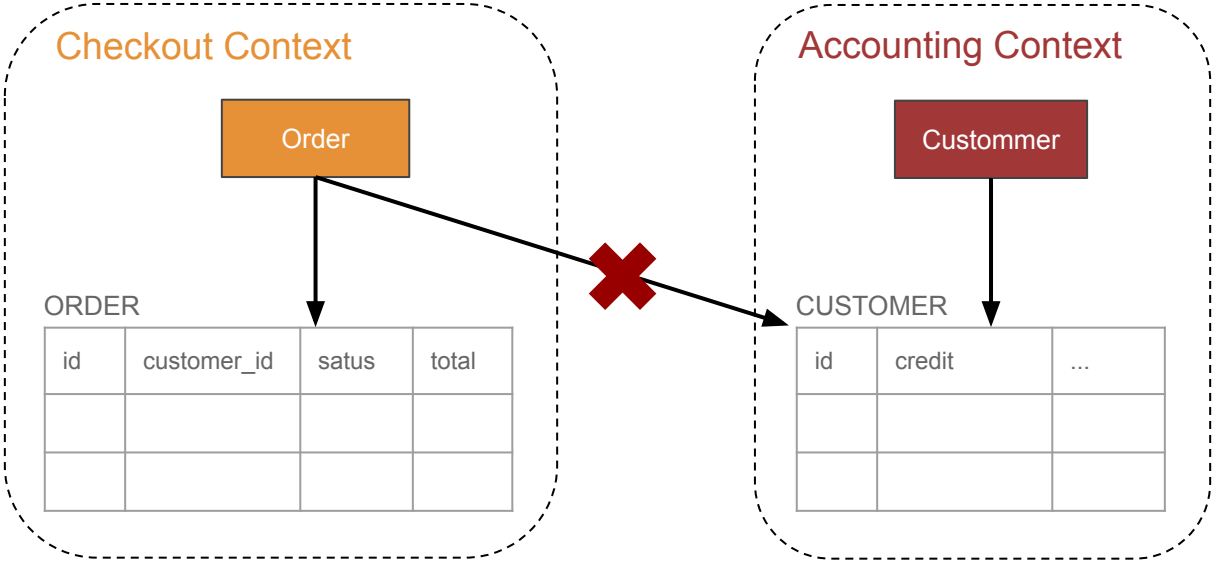
Declarative REST Client

- Discovery Service - Eureka
- Fallbacks - Hystrix
- Request / Response compression

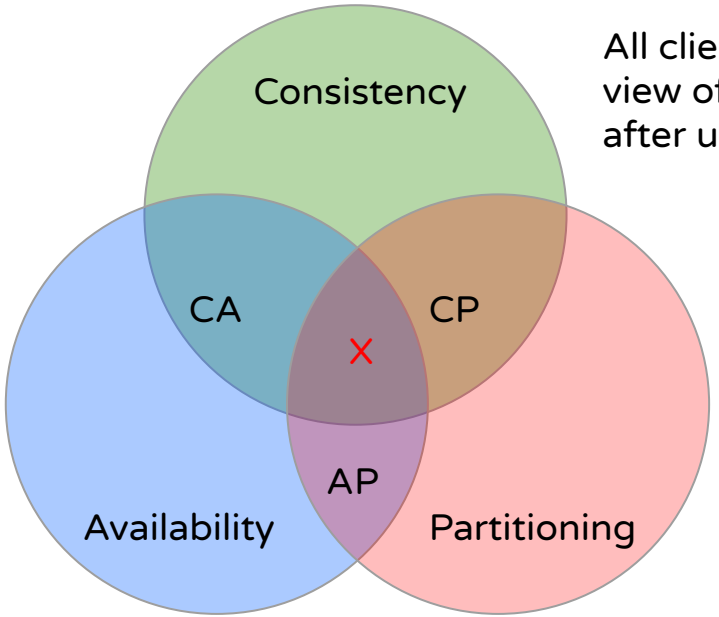
Event-Driven

data management

Problem of Distributed Data



CAP theorem

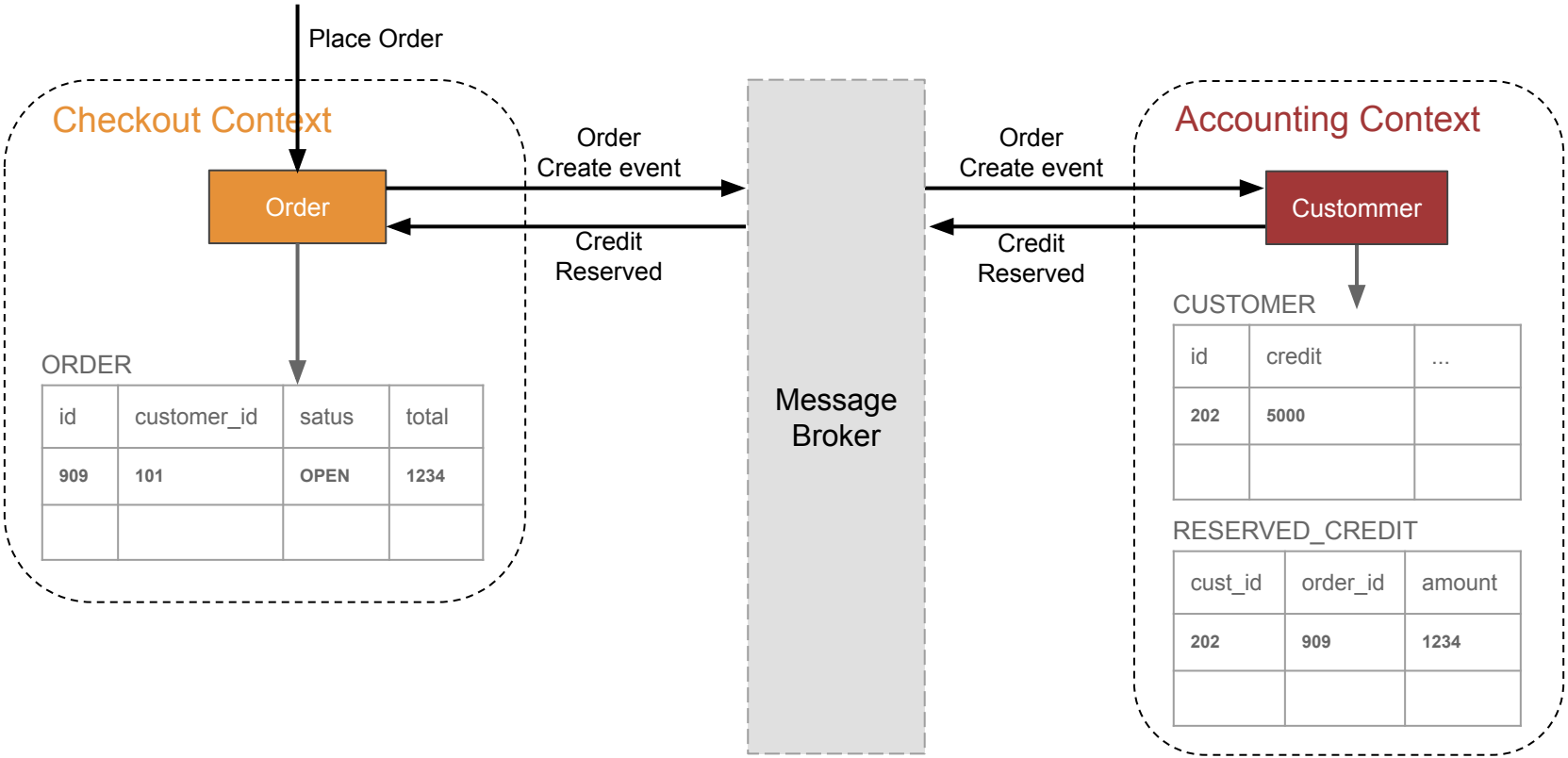


All clients see the same view of data, even right after update or delete

All clients can find a replica of data, even in case of partial node failures

The system continues to work as expected, even in presence of partial network failure

Event-Driven Architecture

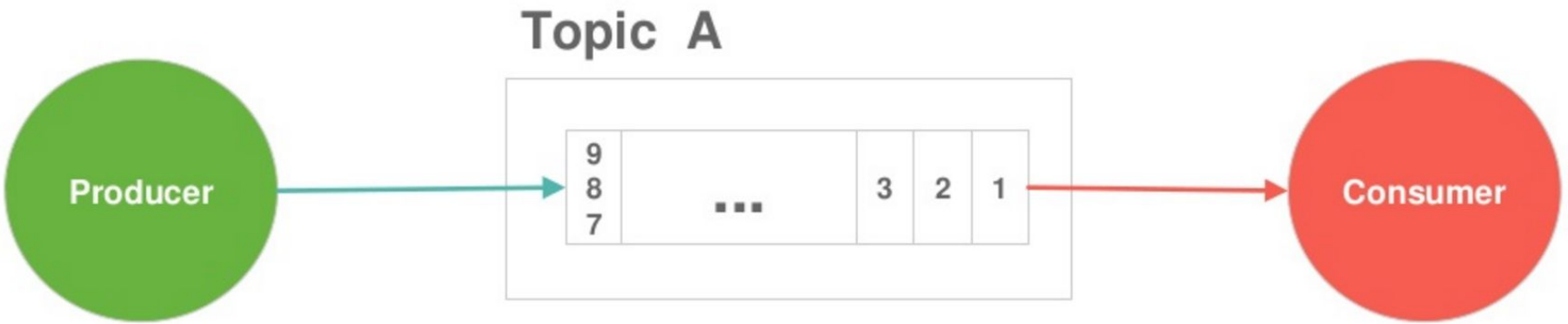


Kafka

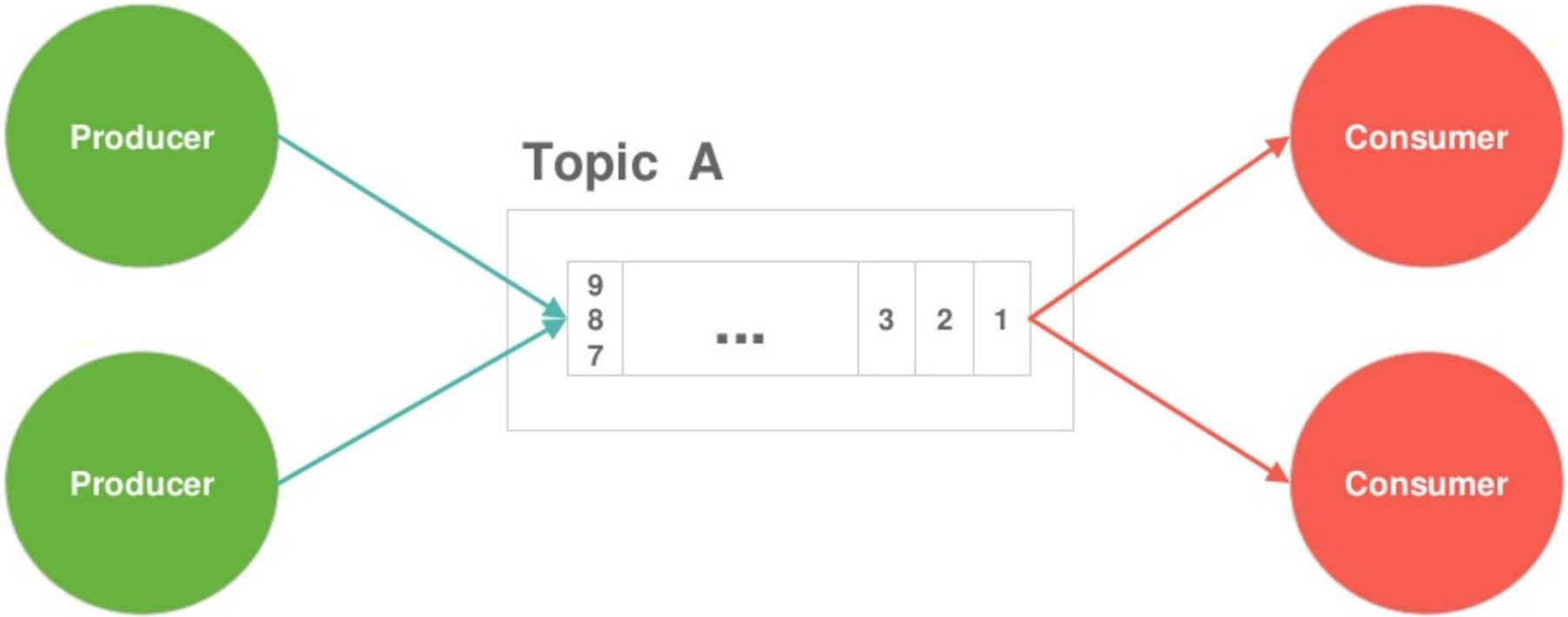
Message Broker

- **Topic** - category or feed name in which messages are published
- **Broker** - server process. The topics live in the broker processes
- **Producer** - publishes data to topics
- **Consumer** - processes subscribed to topic and process the feed of published messages
- **ZooKeeper** - coordinator between the brokers and consumers

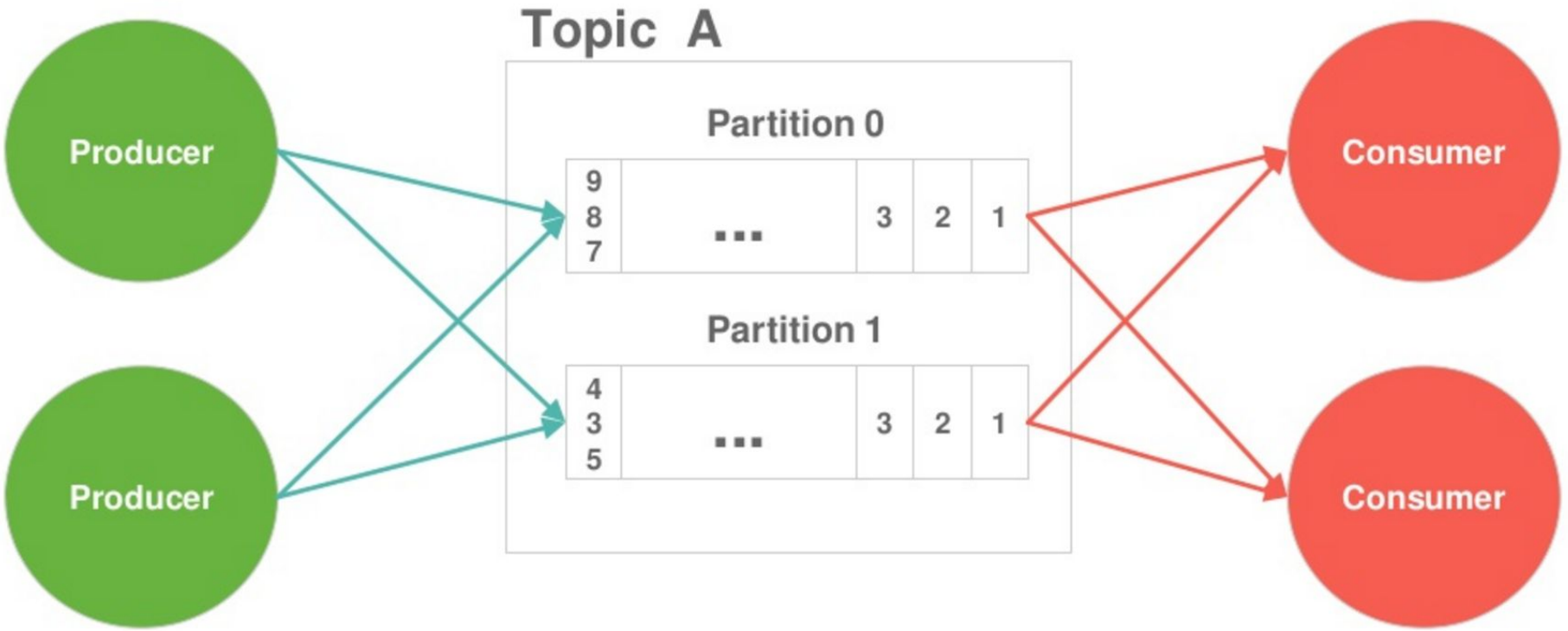
Connection the Microservices to Kafka



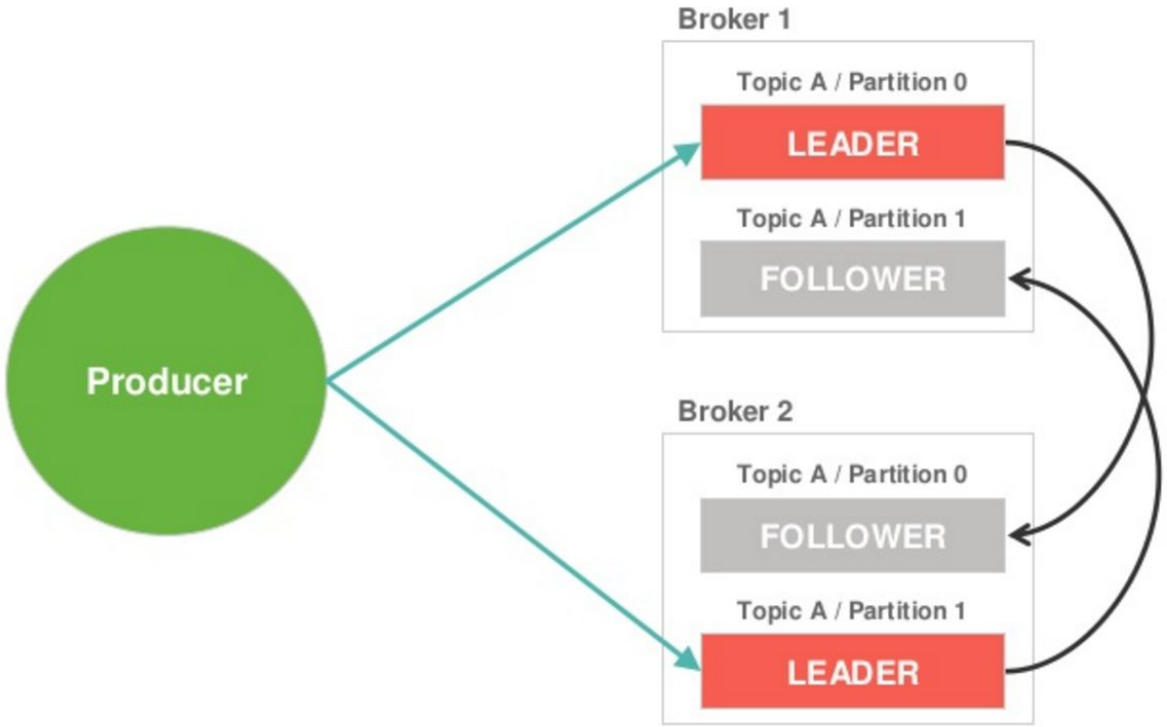
Connection the Microservices to Kafka



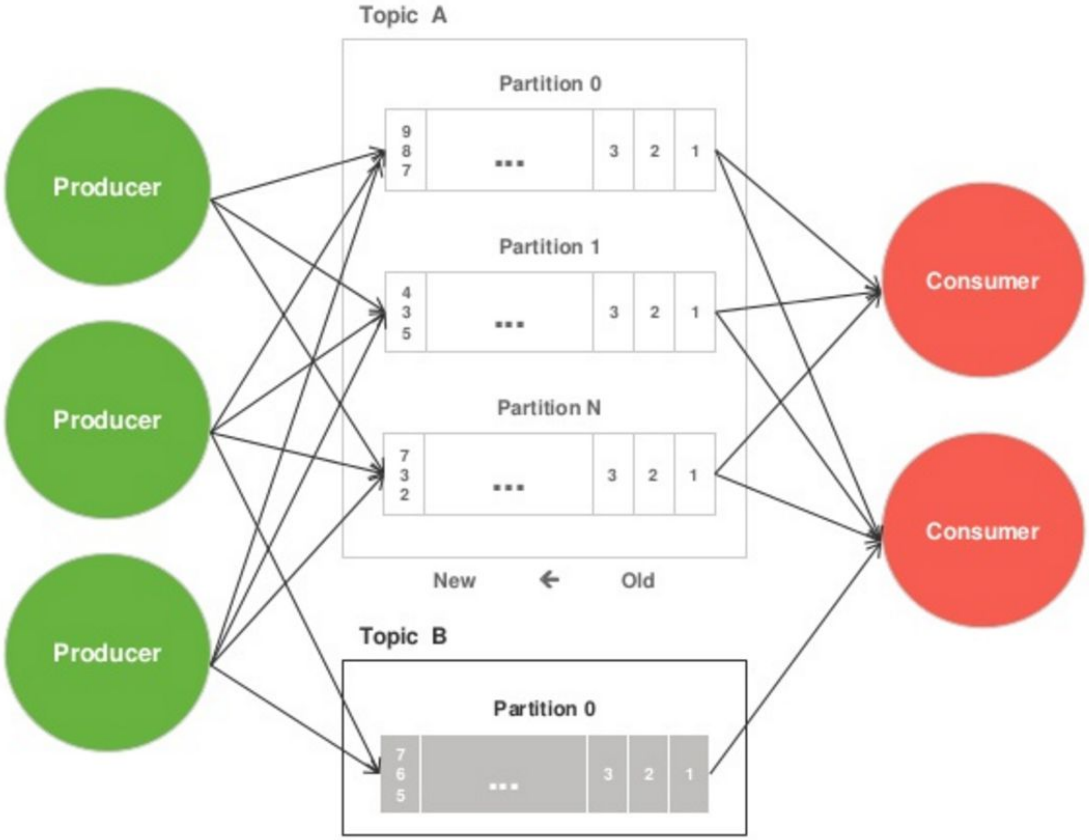
Connection the Microservices to Kafka



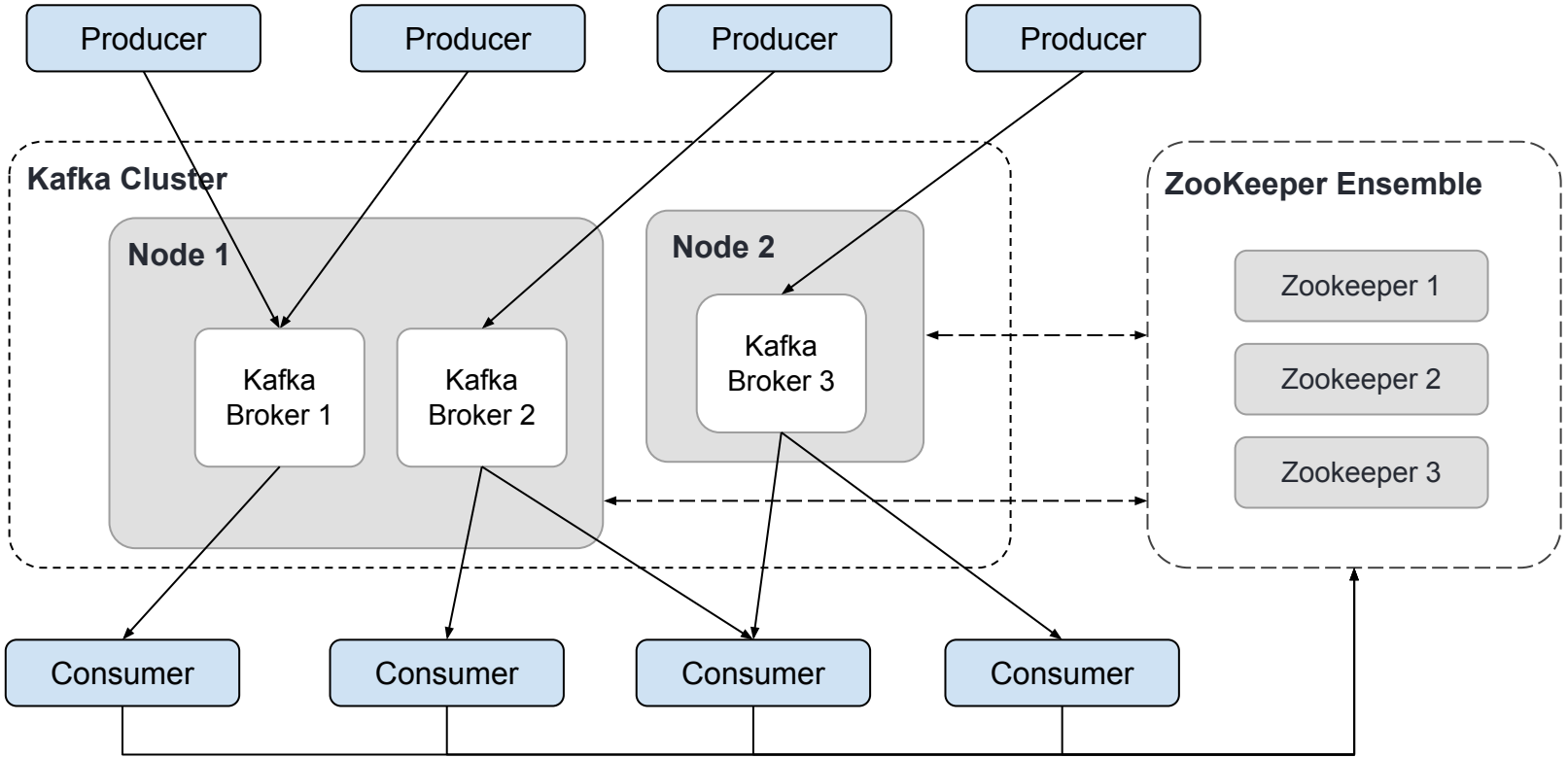
Connection the Microservices to Kafka



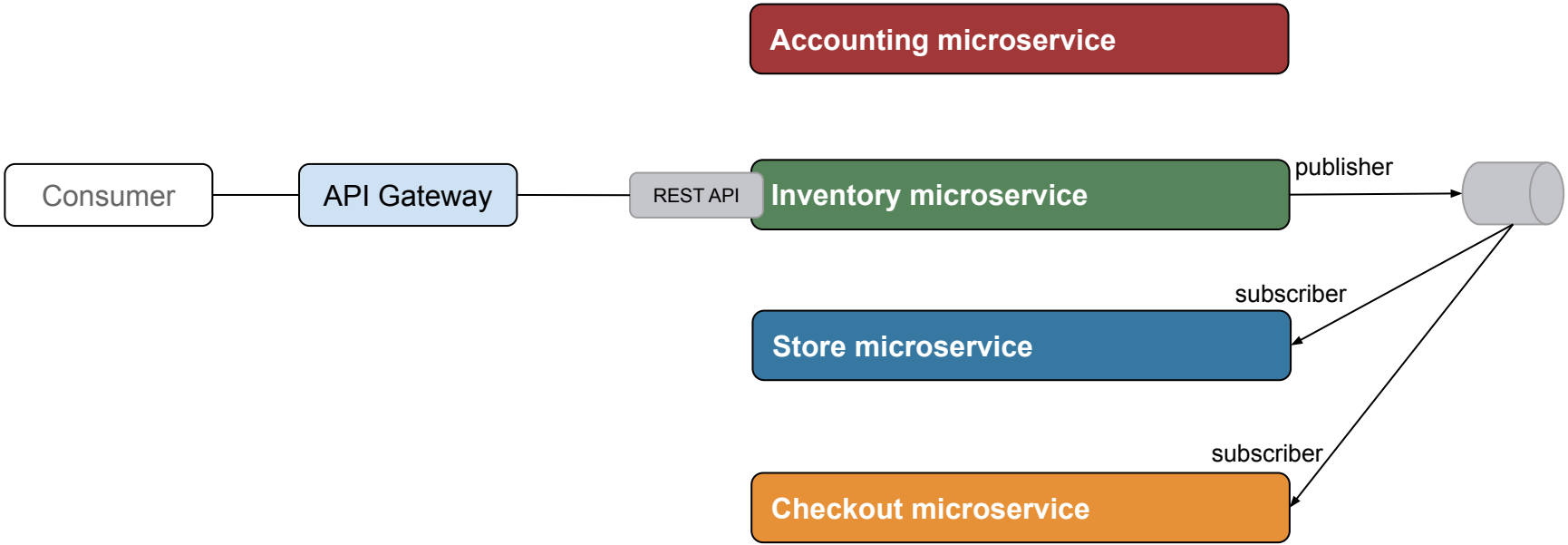
Connection the Microservices to Kafka



Kafka Cluster



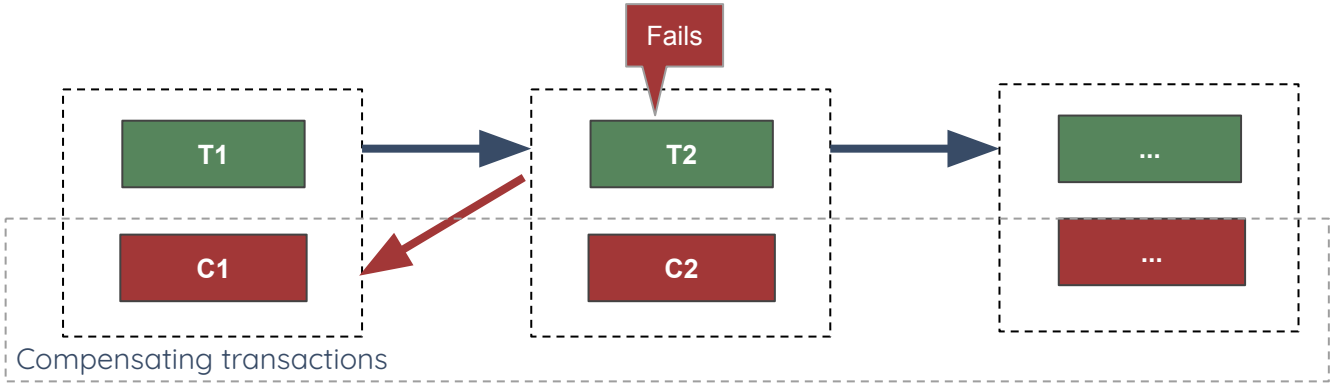
Coding time


Accounting microservice
Consumer
API Gateway
REST API
Inventory microservice
publisher
subscriber
Store microservice
subscriber
Checkout microservice

Saga

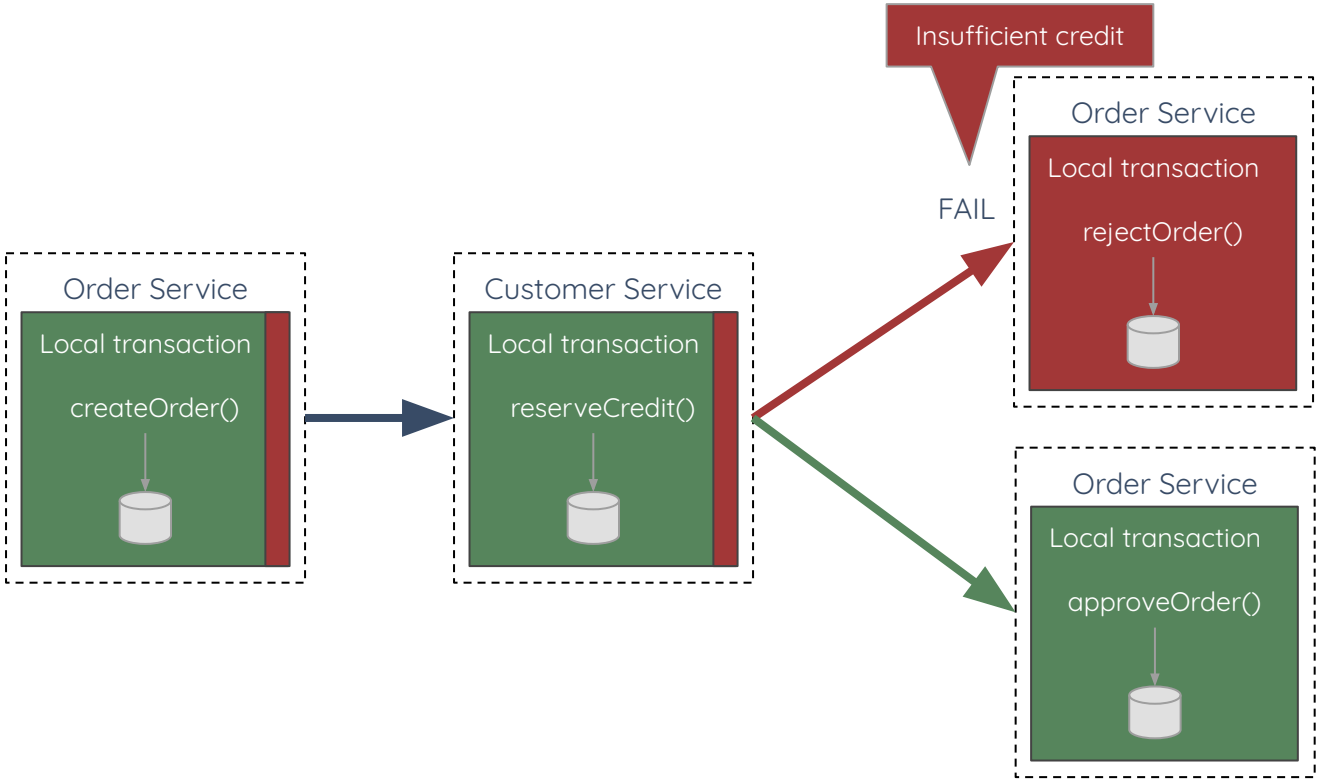
Transaction model

Transaction and Compensation

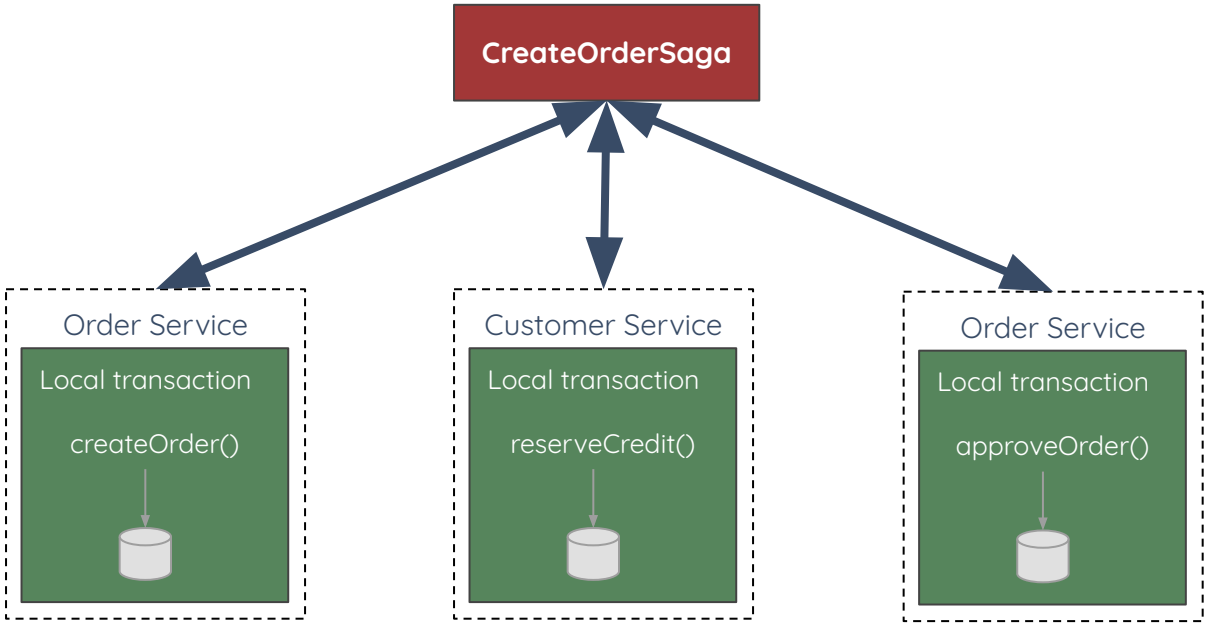


T1 -> T2 -> C1

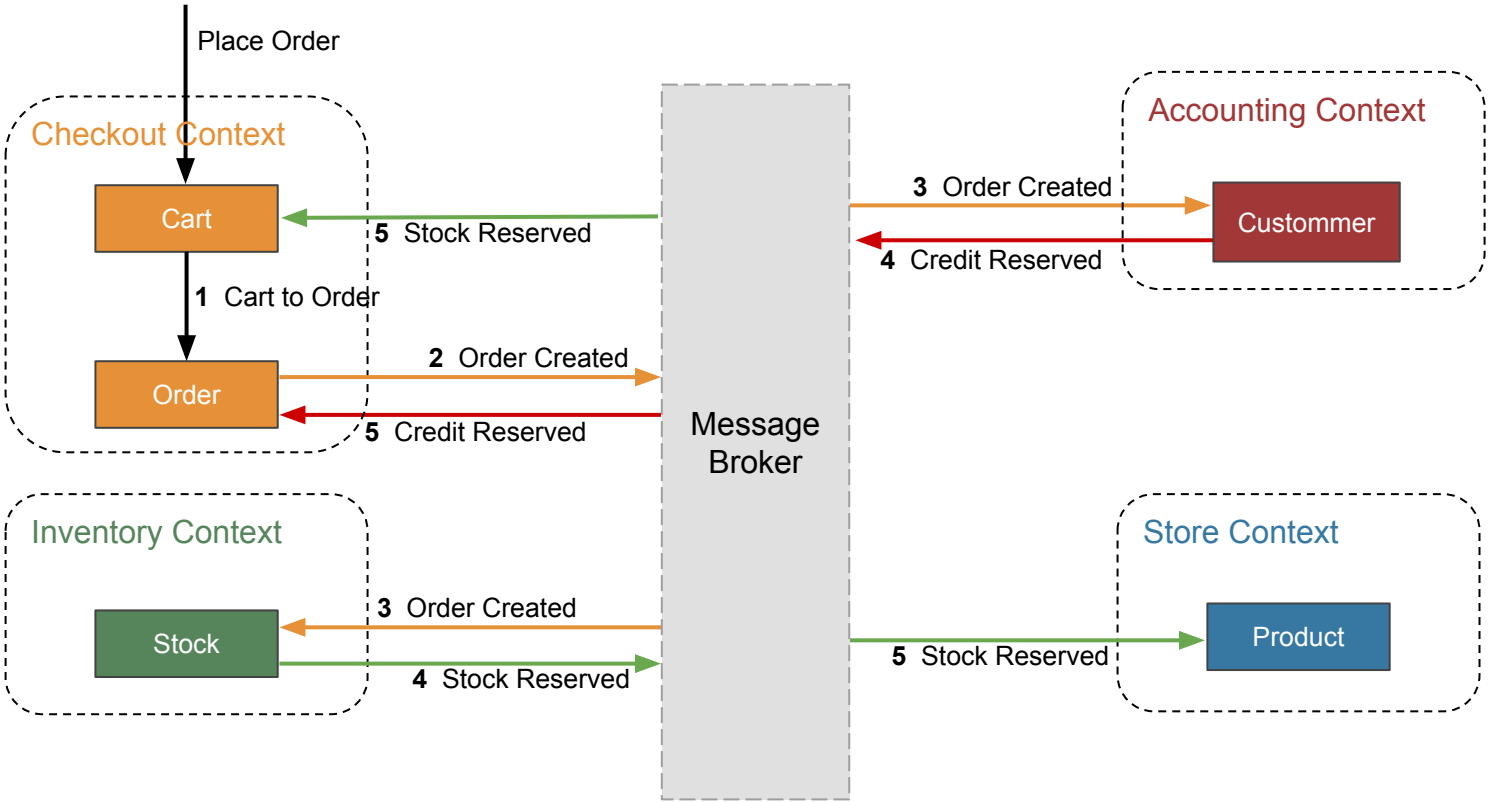
Choreography Coordination



Orchestration Coordination



Coding time


Place Order
Checkout Context
Cart
1 Cart to Order
Order
5 Stock Reserved
2 Order Created
5 Credit Reserved
Message Broker
Accounting Context
3 Order Created
Custommer
4 Credit Reserved
Inventory Context
Stock
3 Order Created
4 Stock Reserved
Store Context
Product
5 Stock Reserved

Security

Patterns and Best Practices

Security Model

Availability is guarantee of reliable access to the information by authorized people.

Integrity is the assurance that the information is trustworthy and accurate.

Confidentiality is set of rules that limits access to information.

API Gateway / Perimeter security

- Requests are authenticated and authorized by the gateway
- The public LB cannot send request to apps directly
- Apps trust all traffic they receive by assumption

Pros

- Network setup can virtually guarantee assumptions
- Apps have stateless security

Cons

- Does noting for internal threats

Basic + Central Auth DB

- All apps get to do authentication and authorization themselves
- Credentials are verified against a central DB
- Basic credentials are passed along in every request

Pros

- Stateless - authenticate every time
- Central user store

Cons

- Auth DB is hit every request
- DB lookup logic needs to be implemented everywhere
- User's credentials can unlock all functionality

Sessions everywhere

- Same as before but each app gets to maintain a session with the client device

Pros

- Auth DB is hit once per session

Cons

- Hard to manage all the sessions
- No single sign on
- DB lookup logic needs to be implemented everywhere
- User’s credentials can unlock all functionality

API Tokens

- Username and password is exchanged for a token at a centralized auth server
- Apps validate the token for each request by hitting the auth server

Pros

- App don’t see user credentials

Cons

- Auth server bottleneck
- Token provides all or nothing access

Microservices

Security Concerns

- Central user store bottleneck
- Single Sign On
- Statelessness
- User credentials
- Fine grained authorization
- Interoperability with non browser

JWT Token


```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
1c2VyX25hbWUiOiJyZWFkZXIiLCJzY29wZSI6WyJ
yZWFkIiwid3JpdGUiXSwicm9sZXMiOlt7ImF1dGh
vcml0eSI6IlJPTEVfVVNFUiJ9XSwiZnVsbG5hbWU
iOiJDYWxpbiBOaWNvYXJhIiwiZXhwIjoxNTg5OTI
1OTA3LCJhdXRob3JpdGllcyI6WyJST0xFX1VTRVI
iXSwianRpIjoiYzA1YzI4YjctODBmOC00ZTY2LTh
kMzktNzFjYmIyODY1ZWE0IiwiY2xpZW50X2lkIjo
iY2xpZW50SWQifQ.pahqfOyYUD8BFBumvULGGEgu
99wMWG4UWHlkgrpT0zCEpVnjTKHKcE_2oGtkjrtx
KPgFL8ESM5ur8JB_kCL0pWB1BLQ3ZE2t8-
rJOGNvPbxfLbCh8CxrYbdTe75hmDJb3g-
BUxdxxg64lfqDObGC0eILfJflDjoe1BY3uCg1gs7
dt6lthT4GhTb4pmq6j7Oz70ztIHqn6bwnHFjJwwn
xyZ43uwnc5Er1snKkUmgg3xWq_9R82iHQO6eSOIW
bdRtPWhbRSKi-70aG8wSoun-
0bVm9aSOaYq1CZmnHDNrihRdgXHrnavPBHBX9Ye1
Tv2_rILcyHHFYnDilGlPISlenjQ
```


HEADER: ALGORITHM & TOKEN TYPE


```
{
  "alg": "RS256",
  "typ": "JWT"
}
```

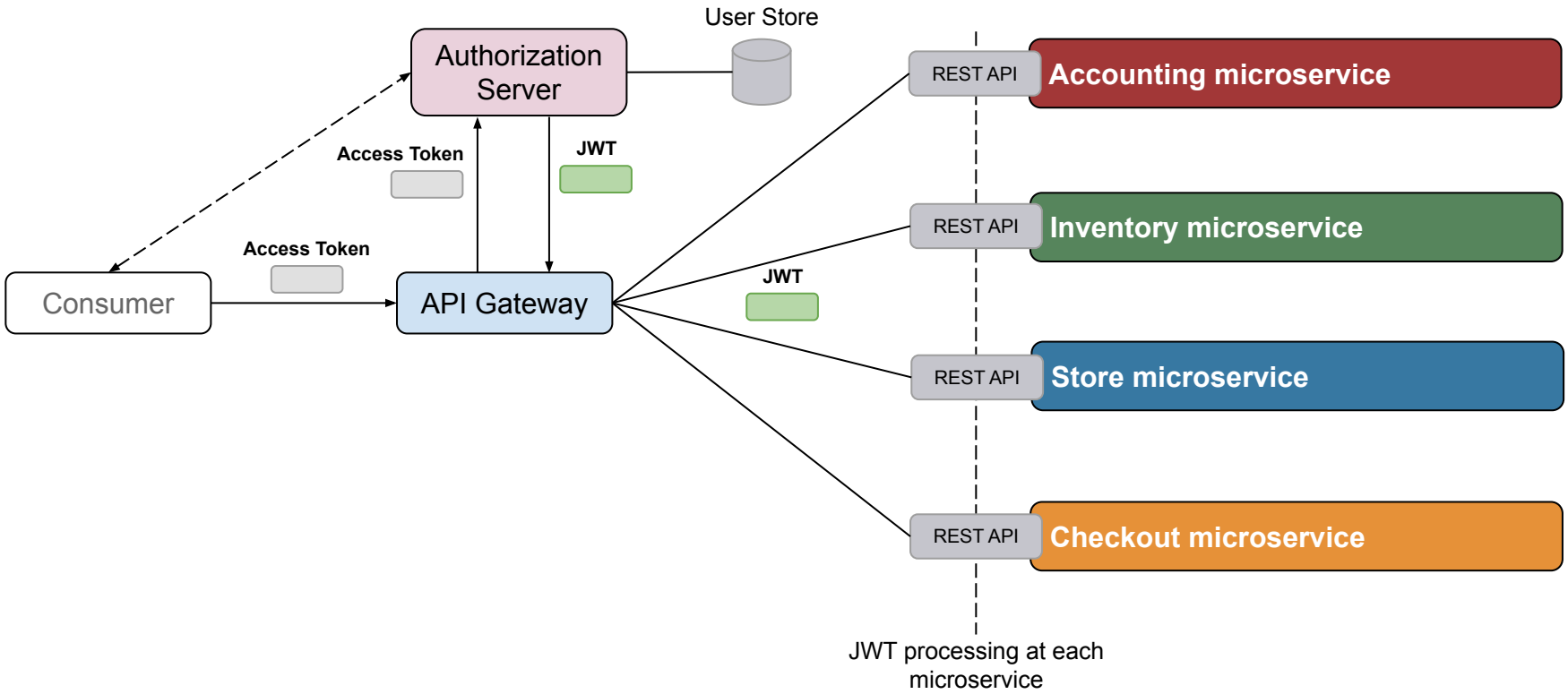

PAYLOAD: DATA


```
{
  "user_name": "reader",
  "scope": [
    "read",
    "write"
  ],
  "roles": [
    {
      "authority": "ROLE_USER"
    }
  ],
  "fullname": "Calin Nicoara",
  "exp": 1589925907,
  "authorities": [
    "ROLE_USER"
  ],
  "jti": "c05c28b7-80f8-4e66-8d39-71cbb2865ea4",
  "client_id": "clientId"
}
```

Oauth2 Concepts

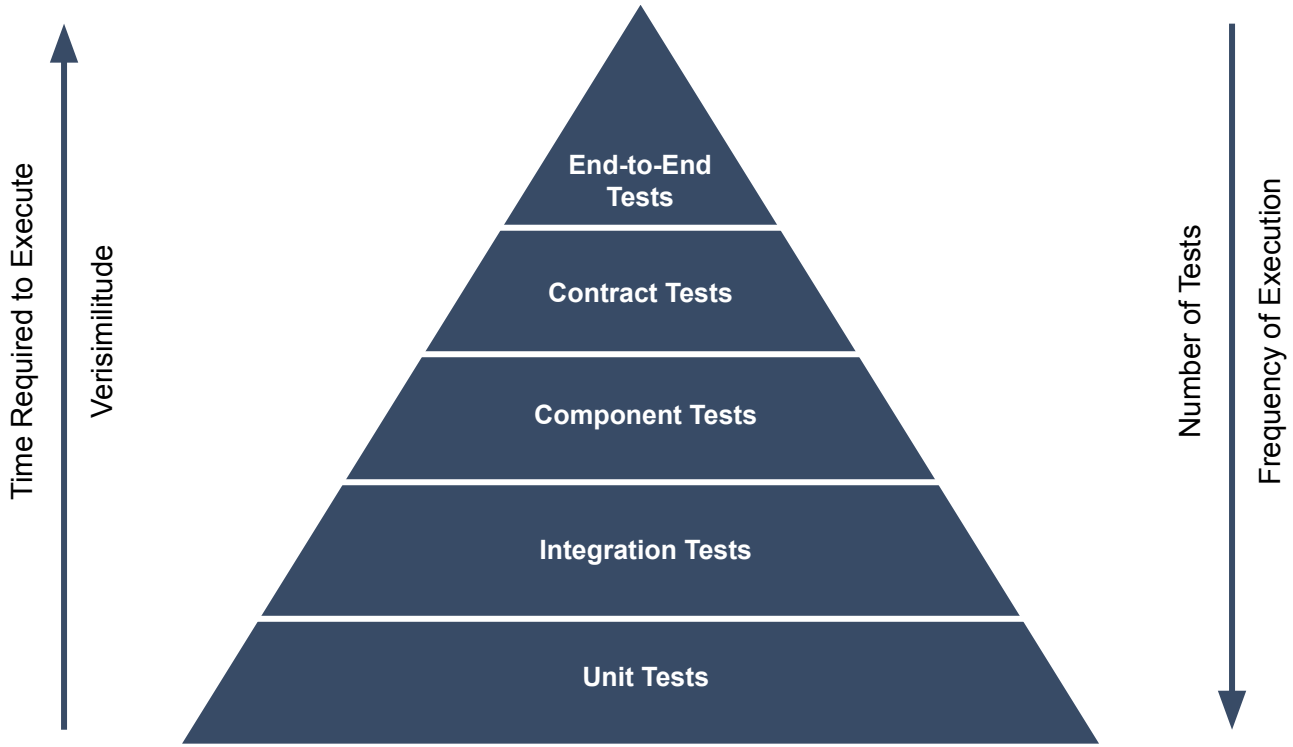
- **Resource Owner -** The user who authorizes an application to access his account. The access is limited to the scope.
- **Resource Server** - A server that handles authenticated requests after the **client** has obtained an **access token**
- **Client -** An application that access protected resources on behalf of the resource owner.
- **Authorization Server -** A server which issues access tokens after successfully authenticating a **client** and **resource owner** and authorizing the request.
- **Access Token -** A unique token used to access protected resources
- **Scope -** A Permission (Example: READ, WRITE)
- **JWT -** JSON Web Token is a method for representing claims securely between two parties
- **Grant type -** a method of acquiring an access token.

OAuth2 and OpenID Connect



Testing
Strategies

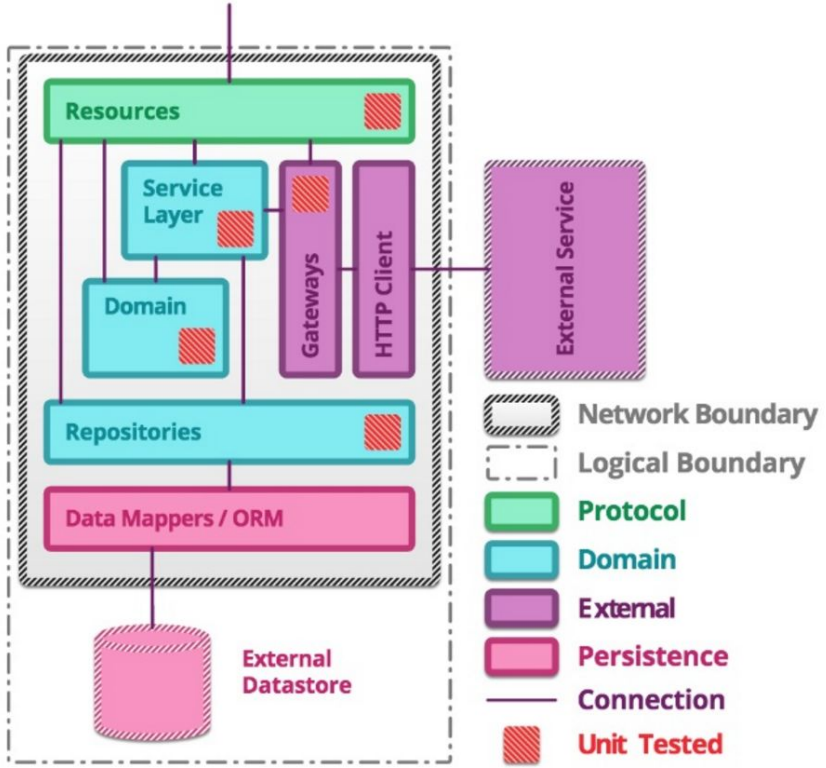
Testing Pyramid



Unit Testing

Make sure that the each component behave as expect.

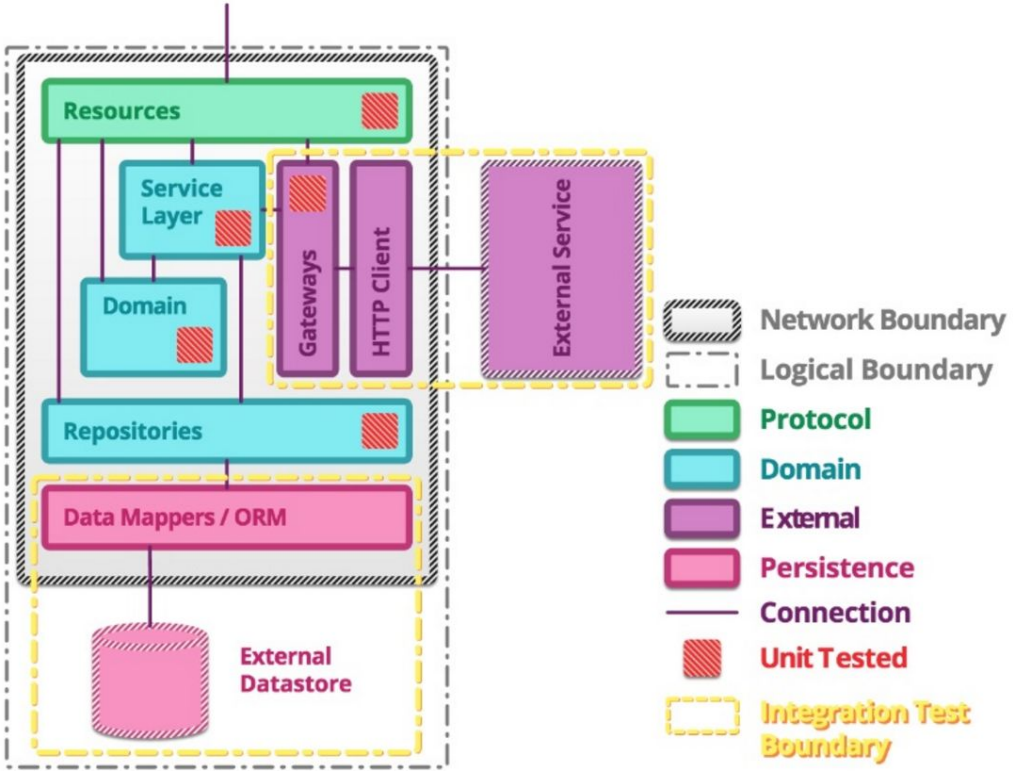
- Domain
- Gateways
- Resources
- Persistence



Integration Testing

Verifying the communications and interactions between components

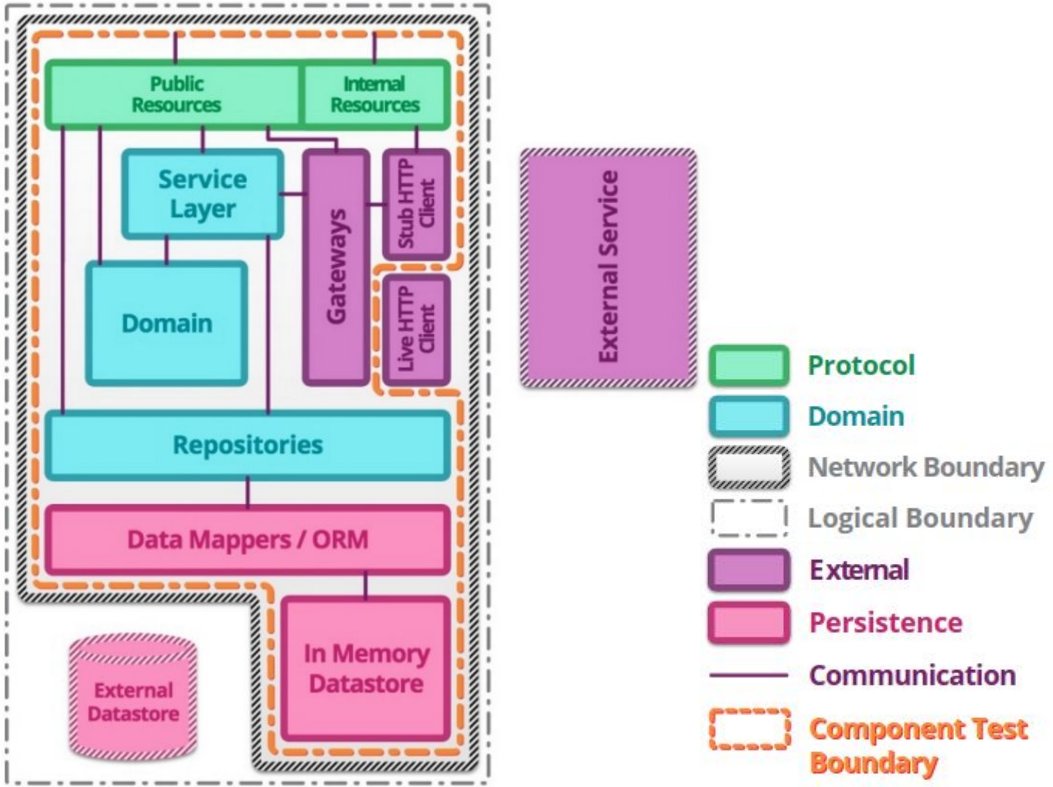
- Persistence
- Gateways



Component Testing

Testing the service in isolation

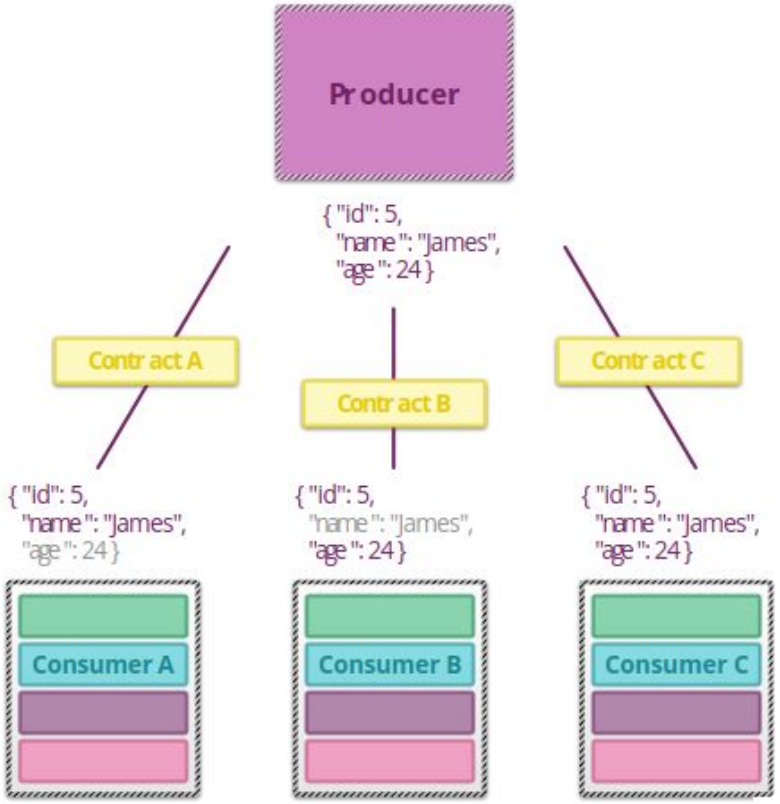
- In Memory DB
- Stub HTTP Client
- Simulate external service



Contract Testing

Verifying the agreements between producer and consumers services

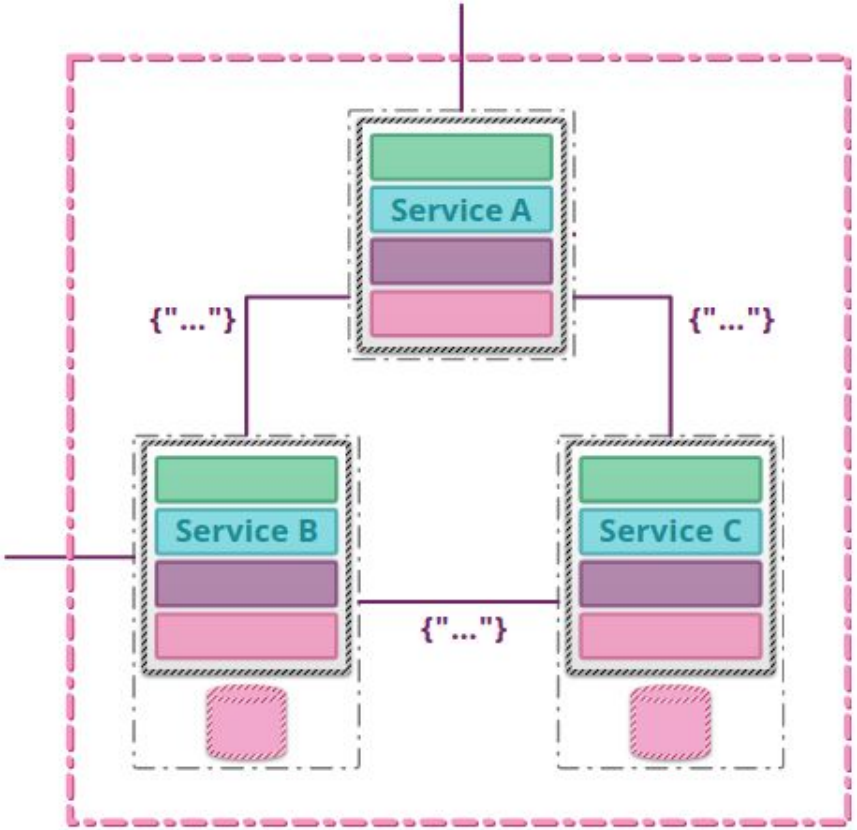
- Input & Output data structure
- Calls attributes



End-to-End Testing

Testing the behaviour of the entire system

- Business flow
- Proxies
- Load Balancers



Deployment
Strategy

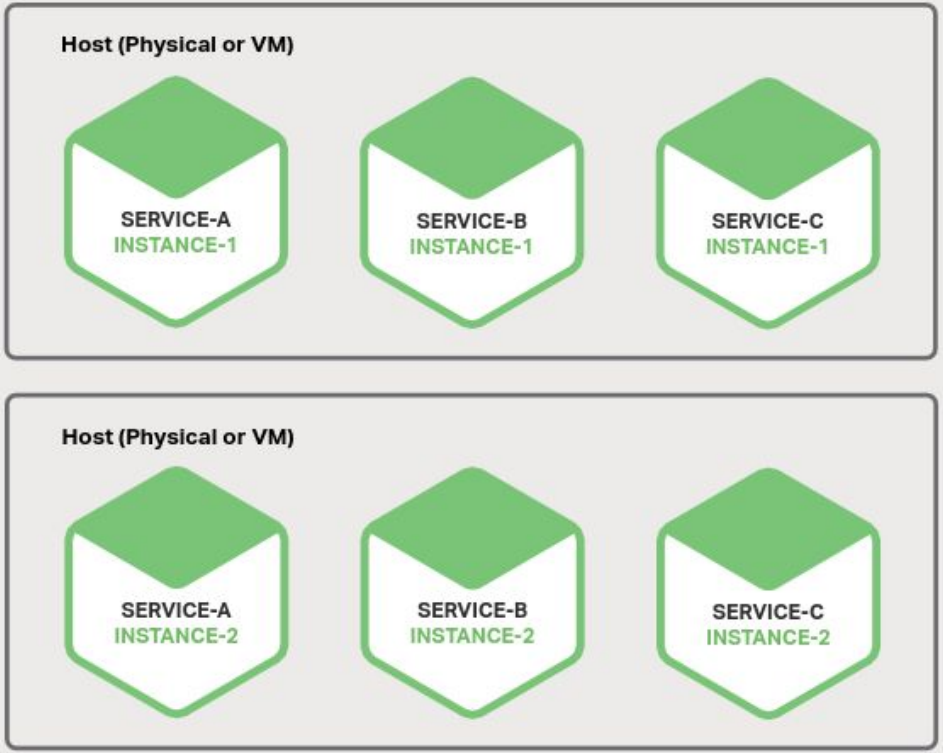
Multiple Service Instances per Host

Pros

- Efficient resource utilization
- Fast deployment

Cons

- Poor isolation
- Difficult to limit resource utilization
- Risk to dependency version conflicts
- Poor encapsulation of implementation technology



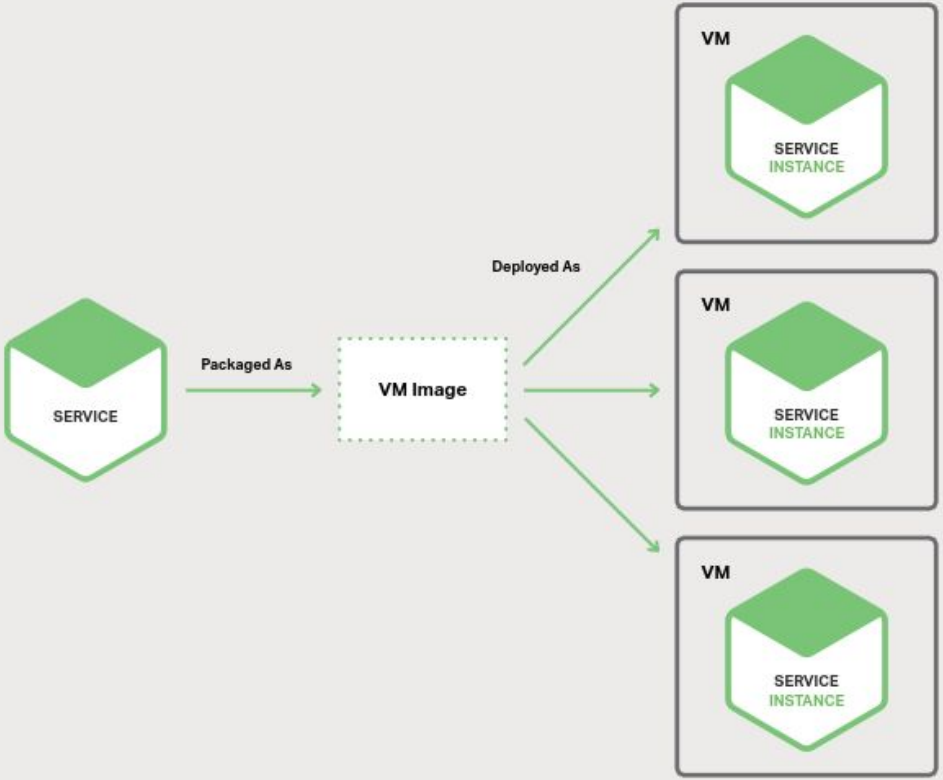
Service Instance per Virtual Machine

Pros

- Great isolation
- Great manageability
- VM encapsulates implementation technology

Cons

- Less efficient resource utilization
- Slow deployment



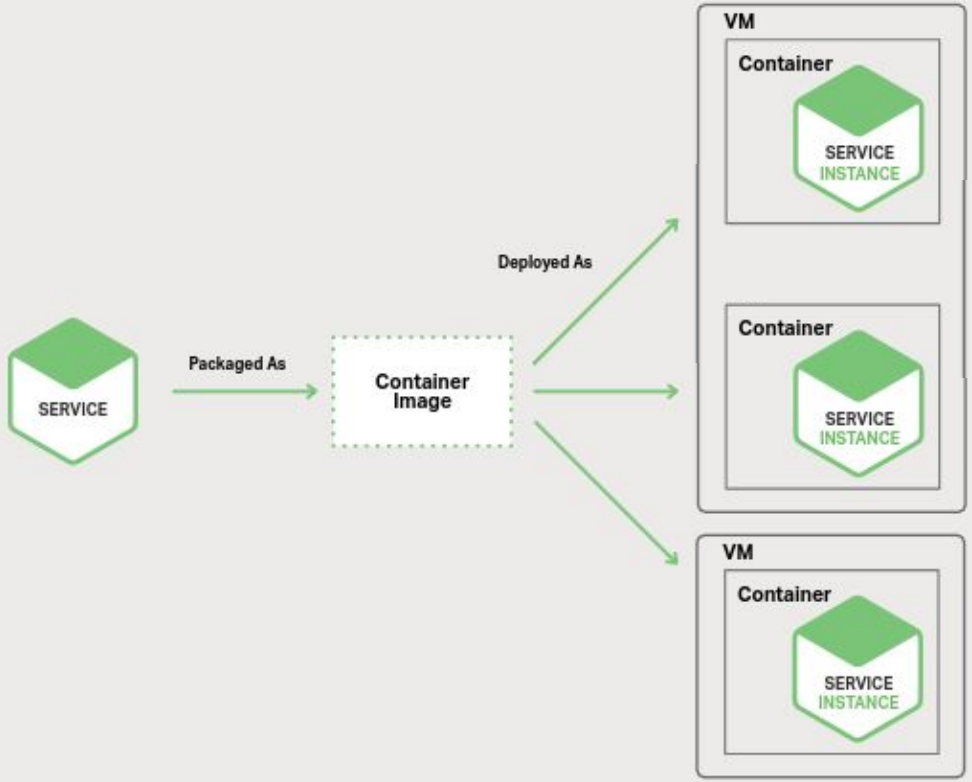
Service Instance per Container

Pros

- Great isolation
- Great manageability
- Container encapsulates implementation technology
- Efficient resource utilization
- Fast deployment

Cons

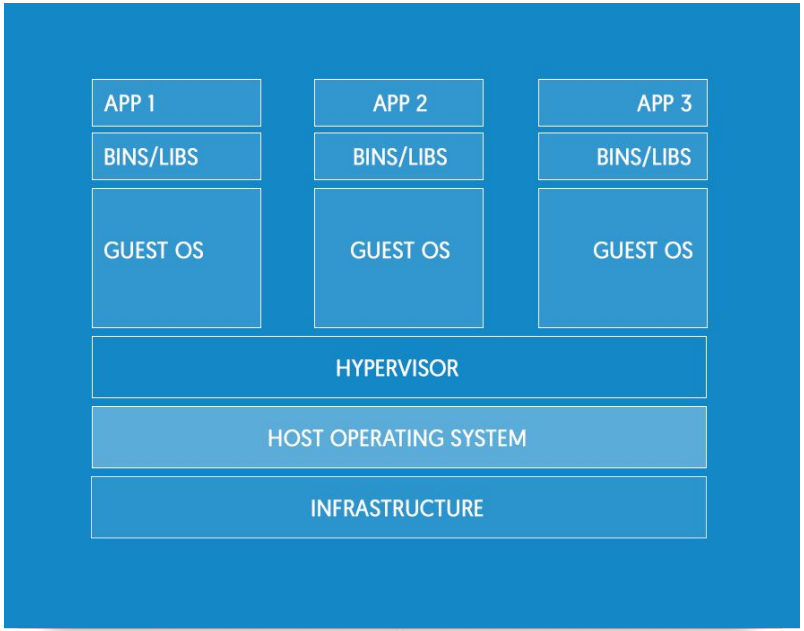
- Immature infrastructure for deploying containers

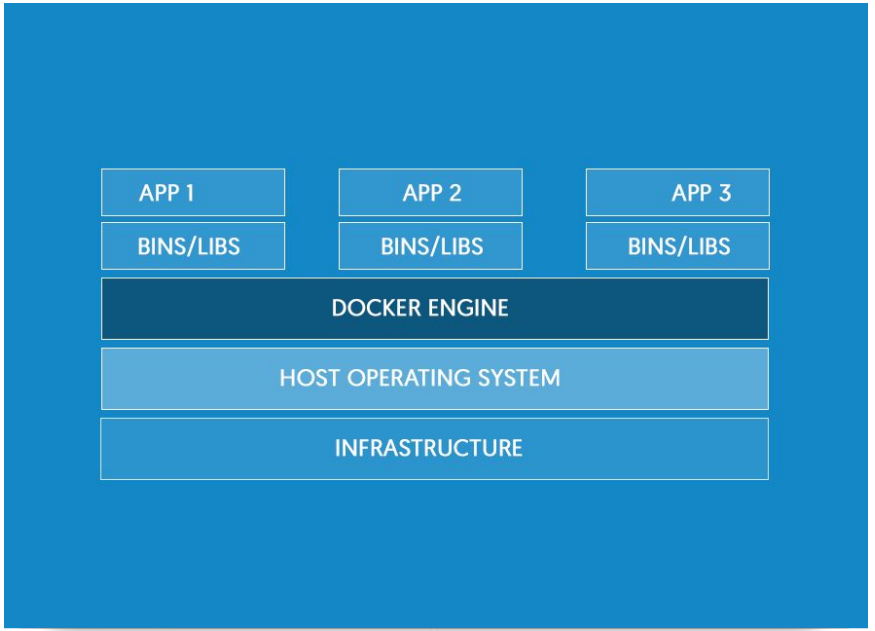


Deployment

With containers

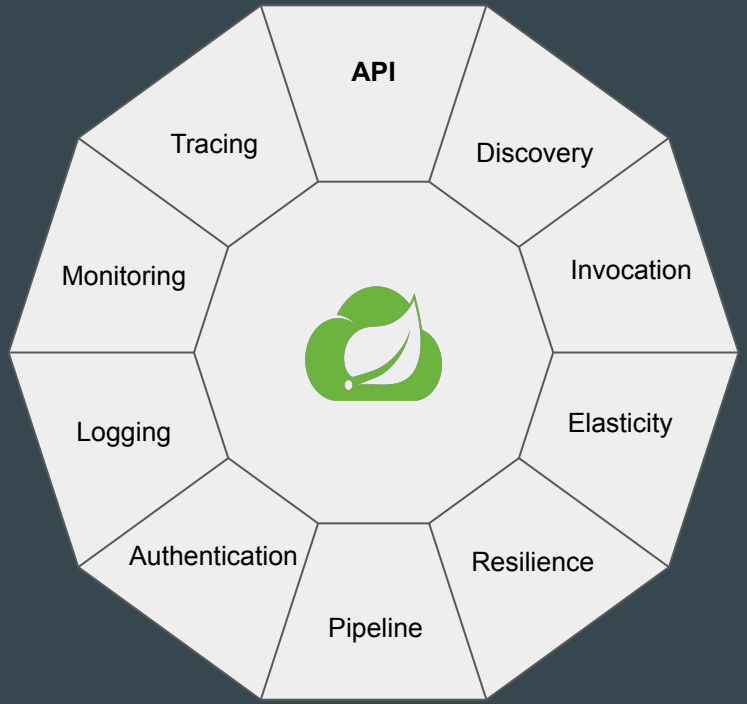
Docker





Source: https://www.docker.com/what-docker

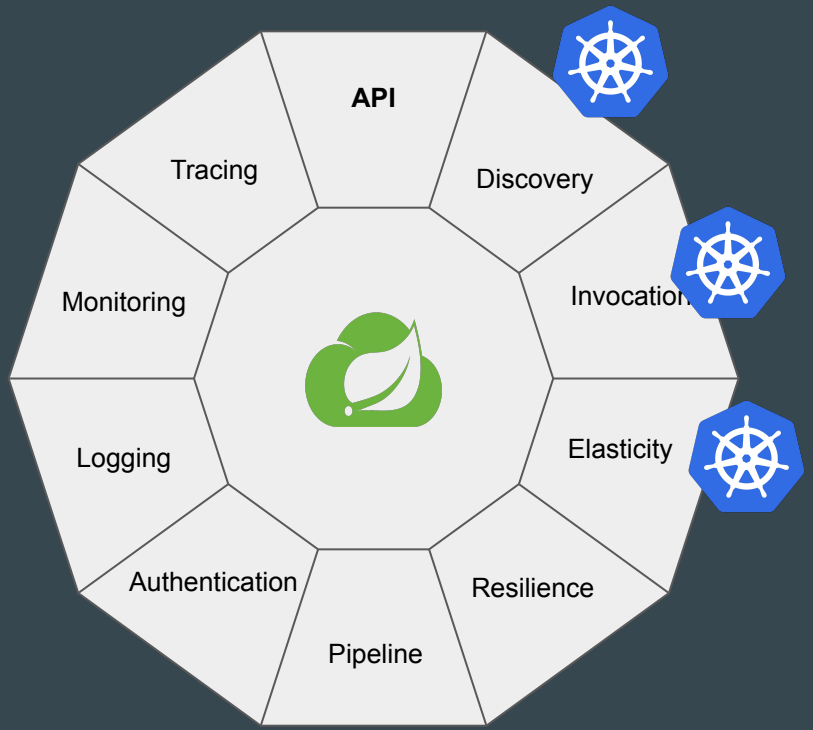
Microservices requirements



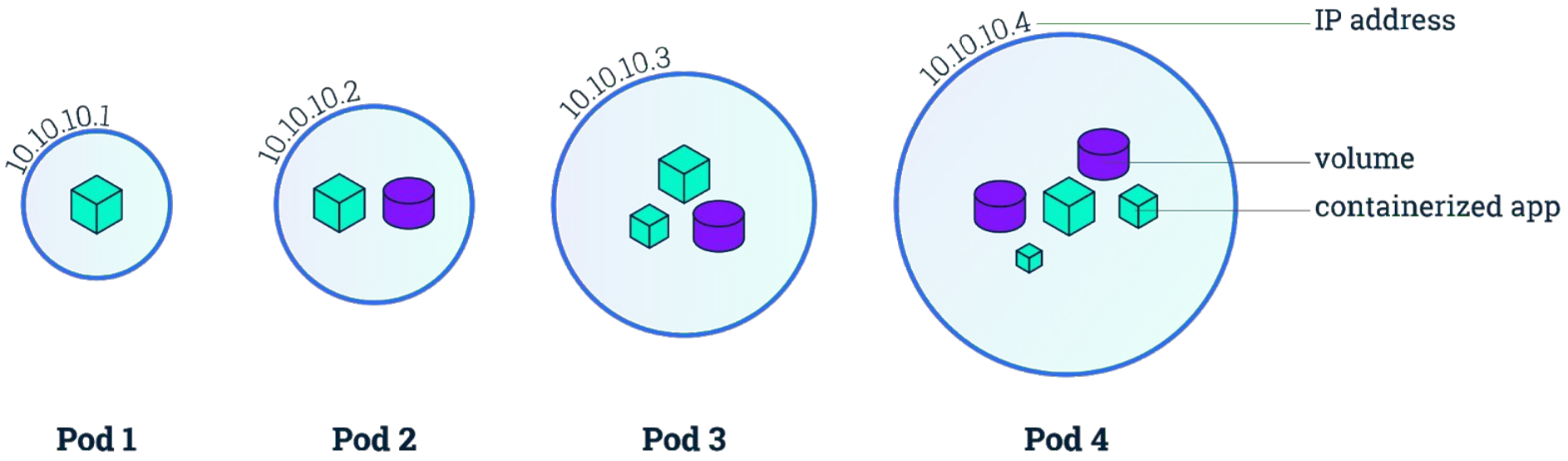
Kubernetes

Kubernetes coordinates a highly available cluster of computers that are connected to work as a single unit.

What Kubernetes covers

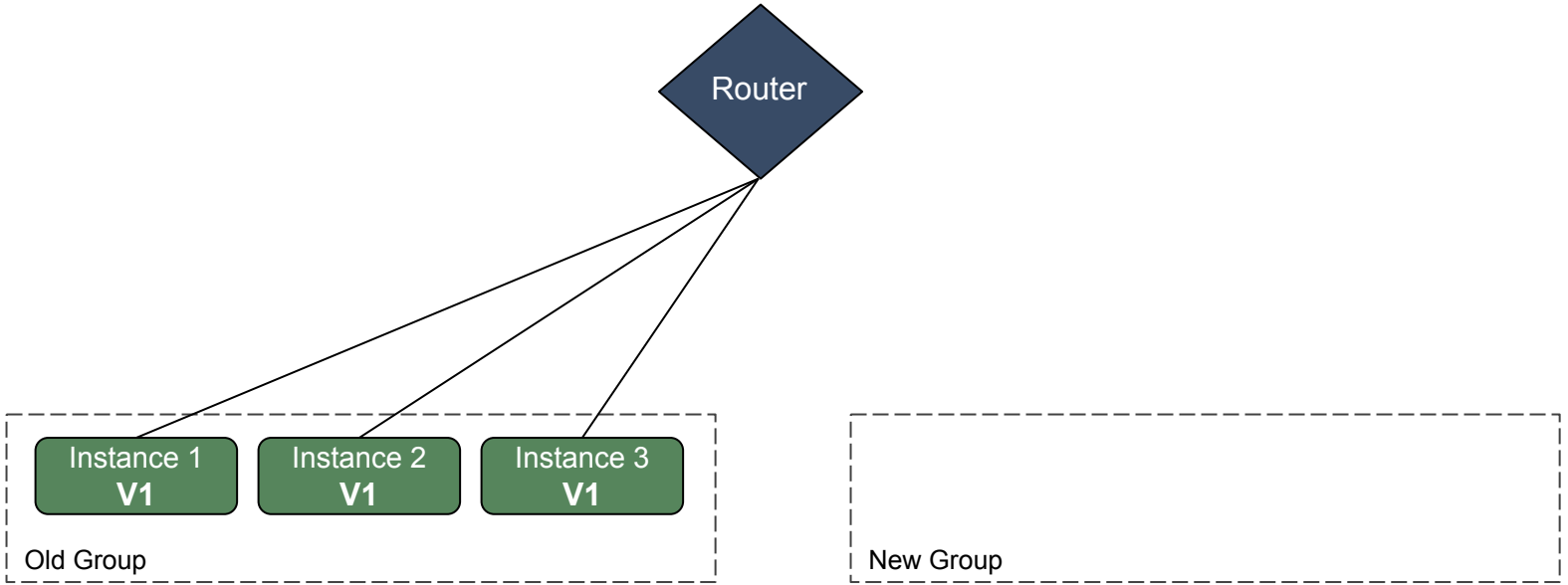


Kubernetes Pods

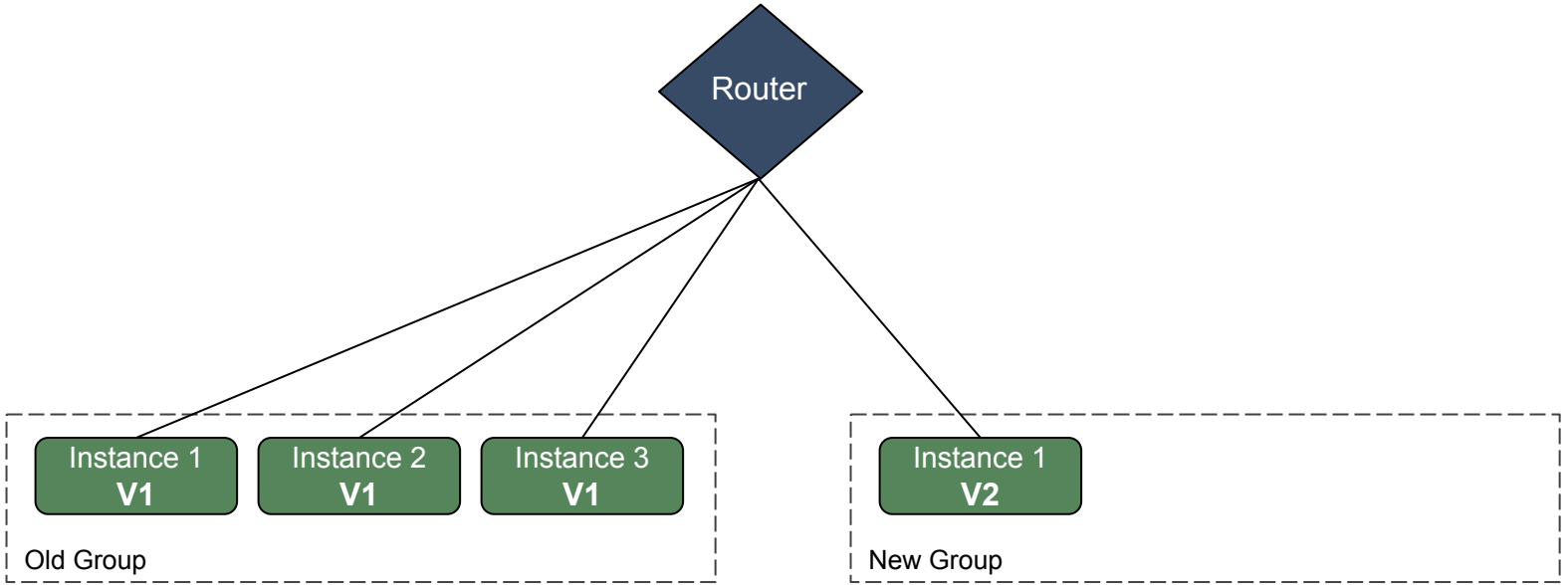


Deployment
Patterns

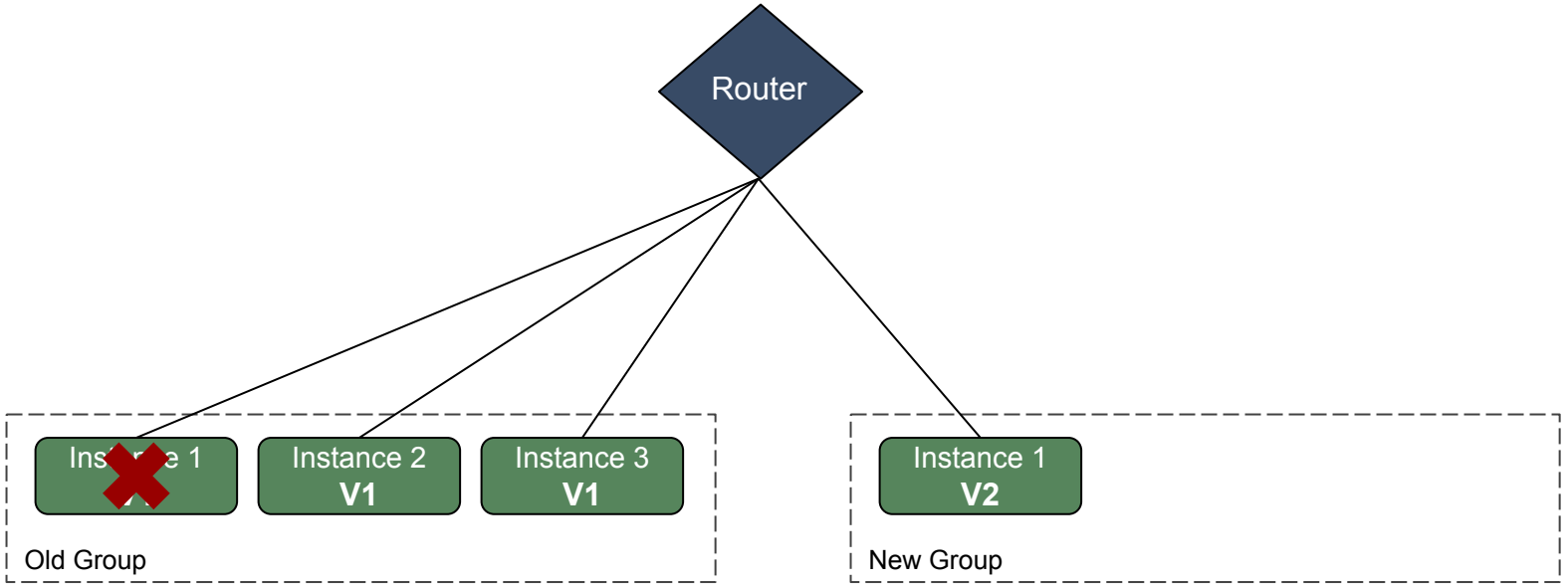
Rolling Update



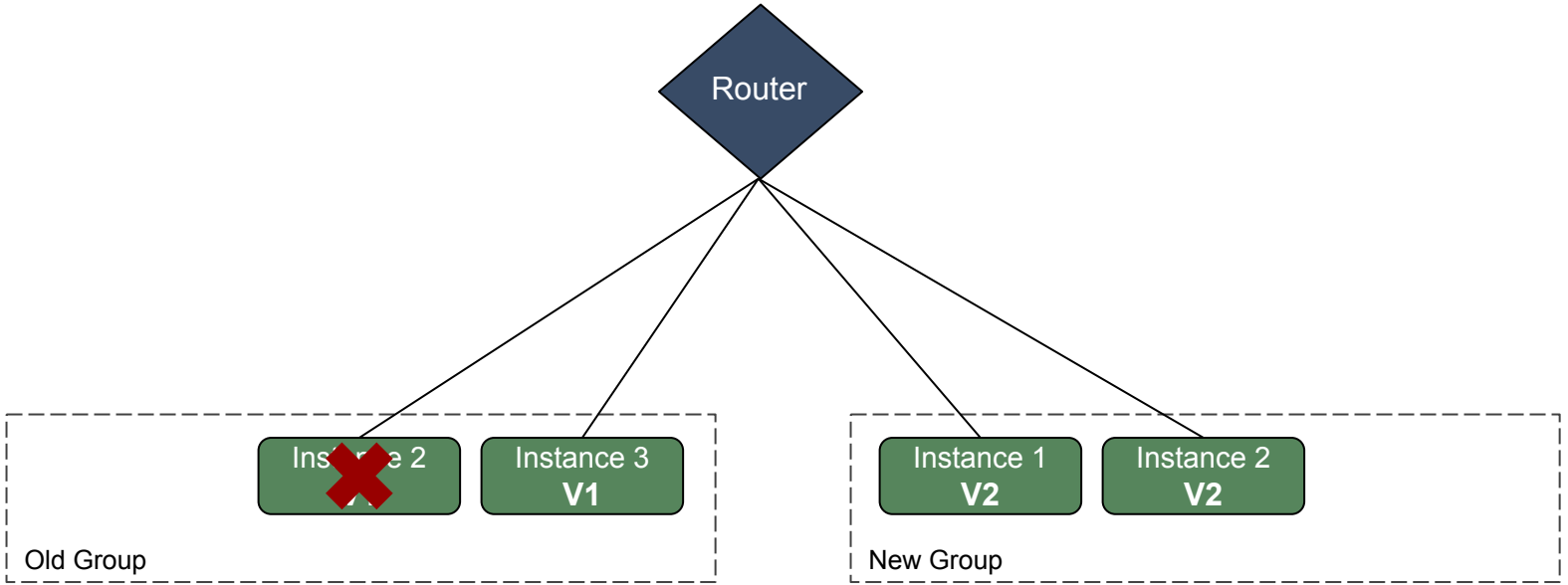
Rolling Update



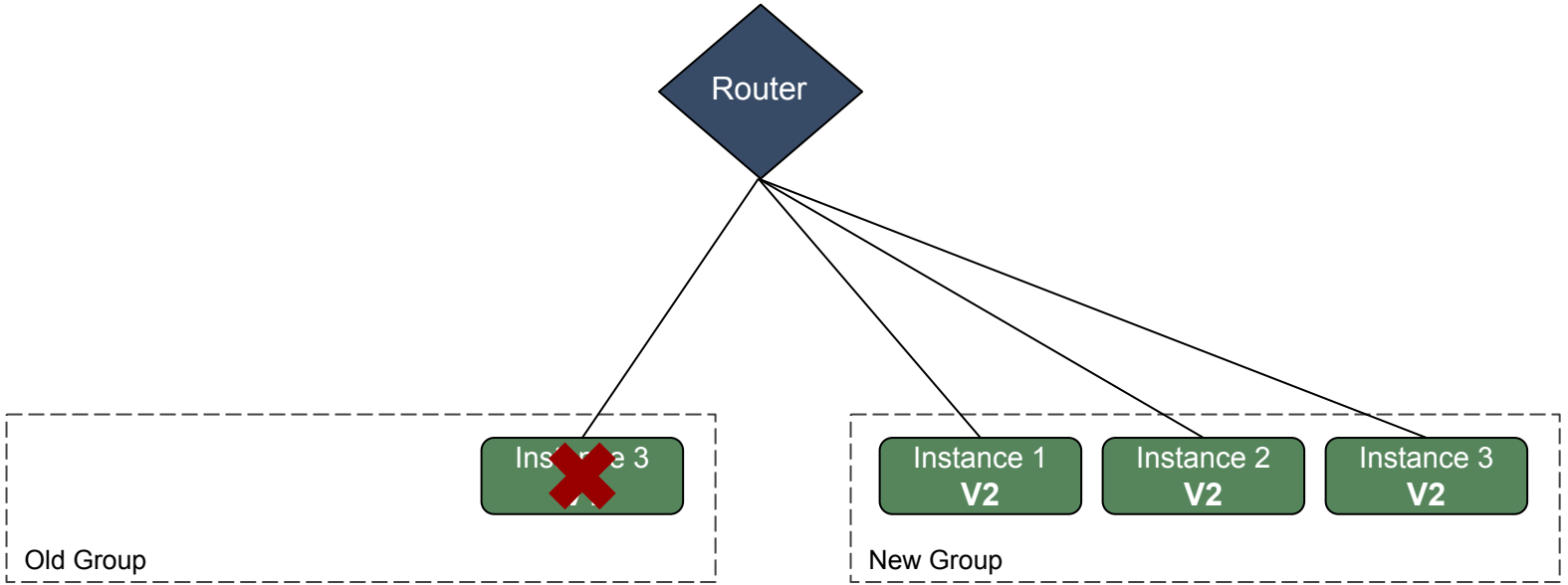
Rolling Update



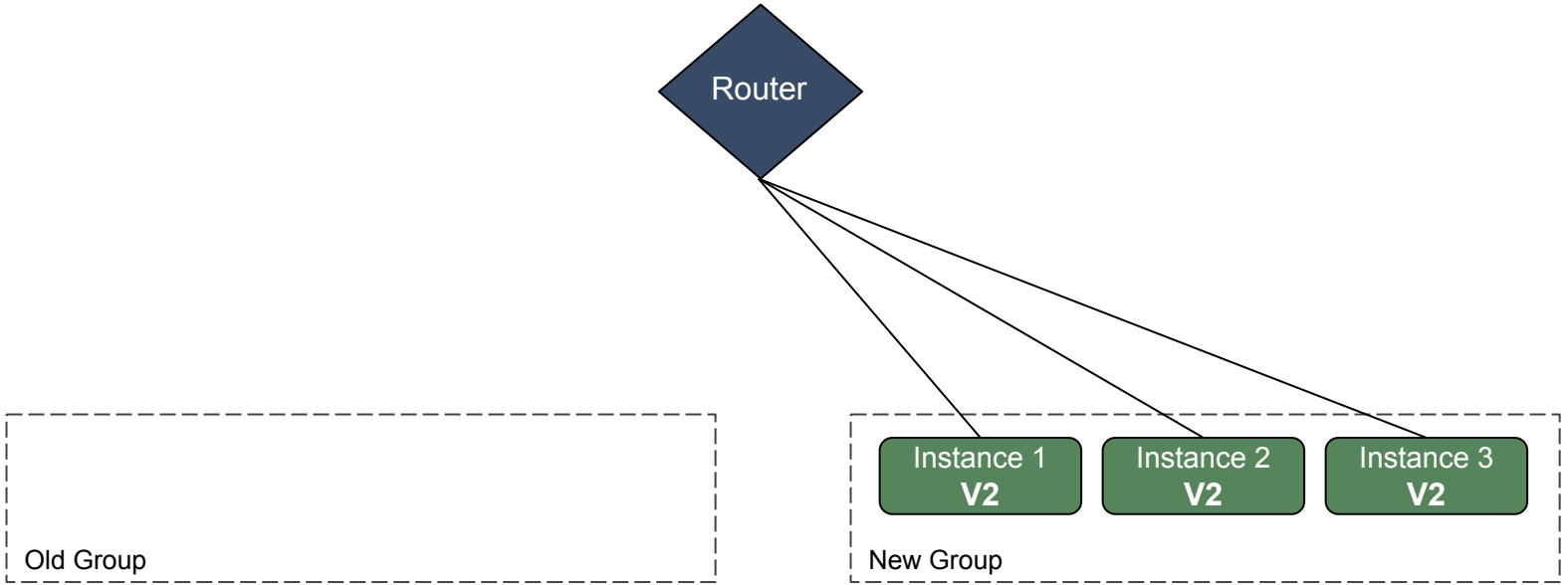
Rolling Update



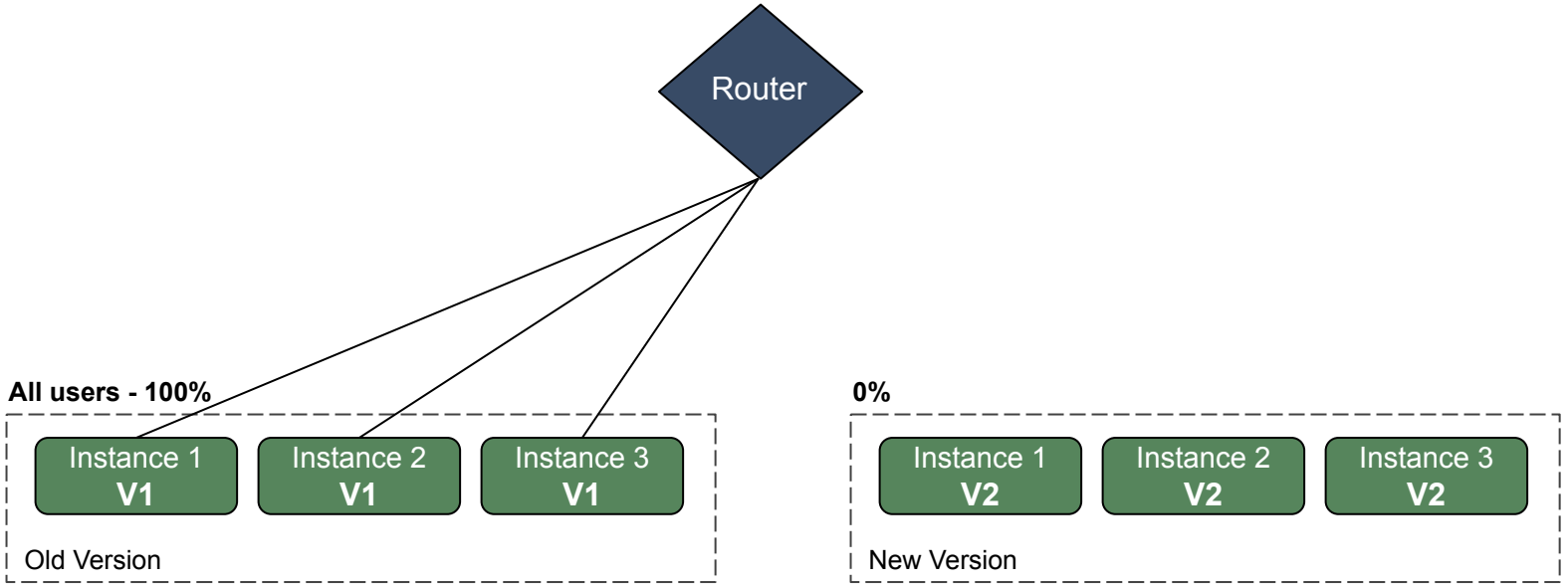
Rolling Update



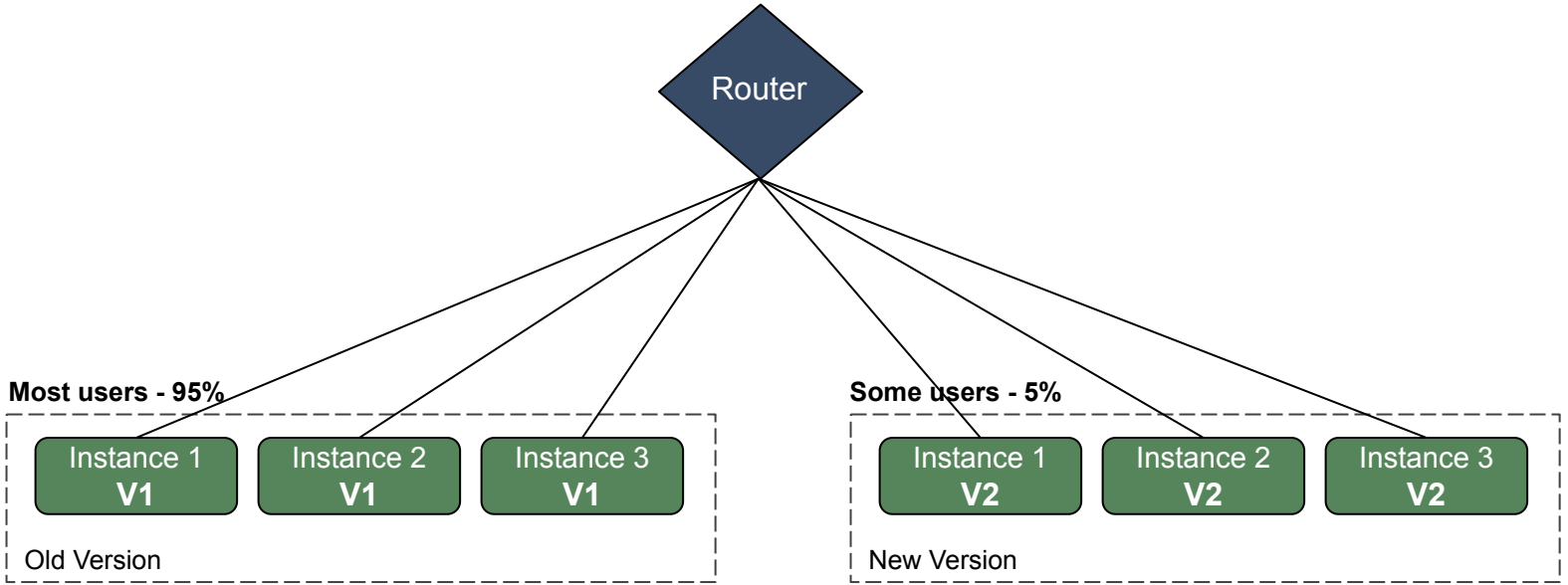
Rolling Update



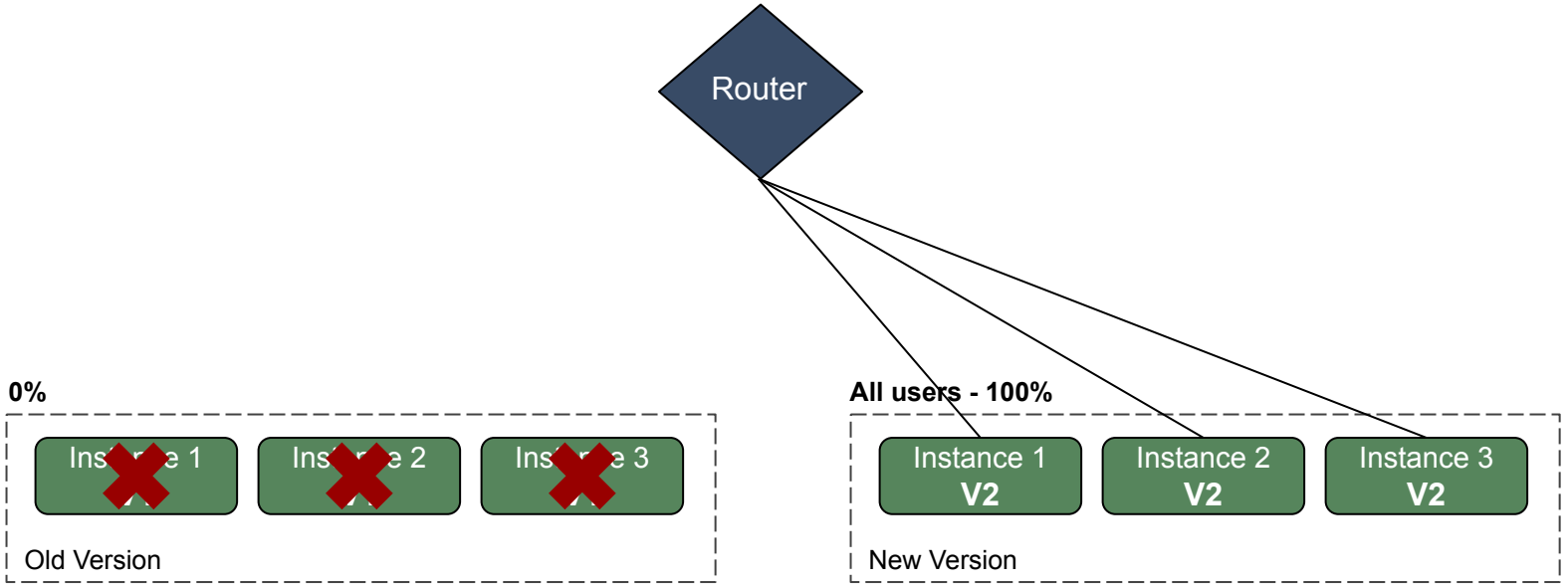
Canary Release



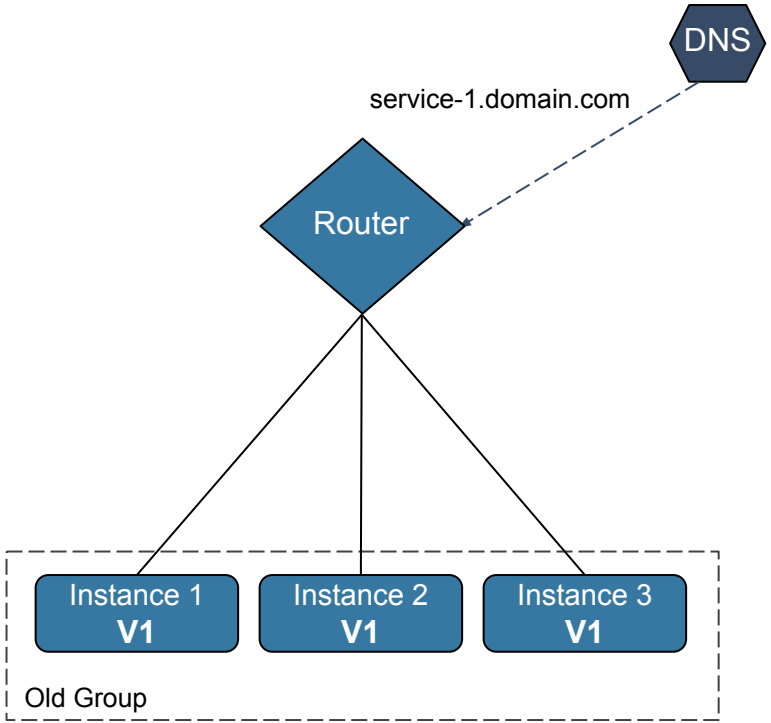
Canary Release



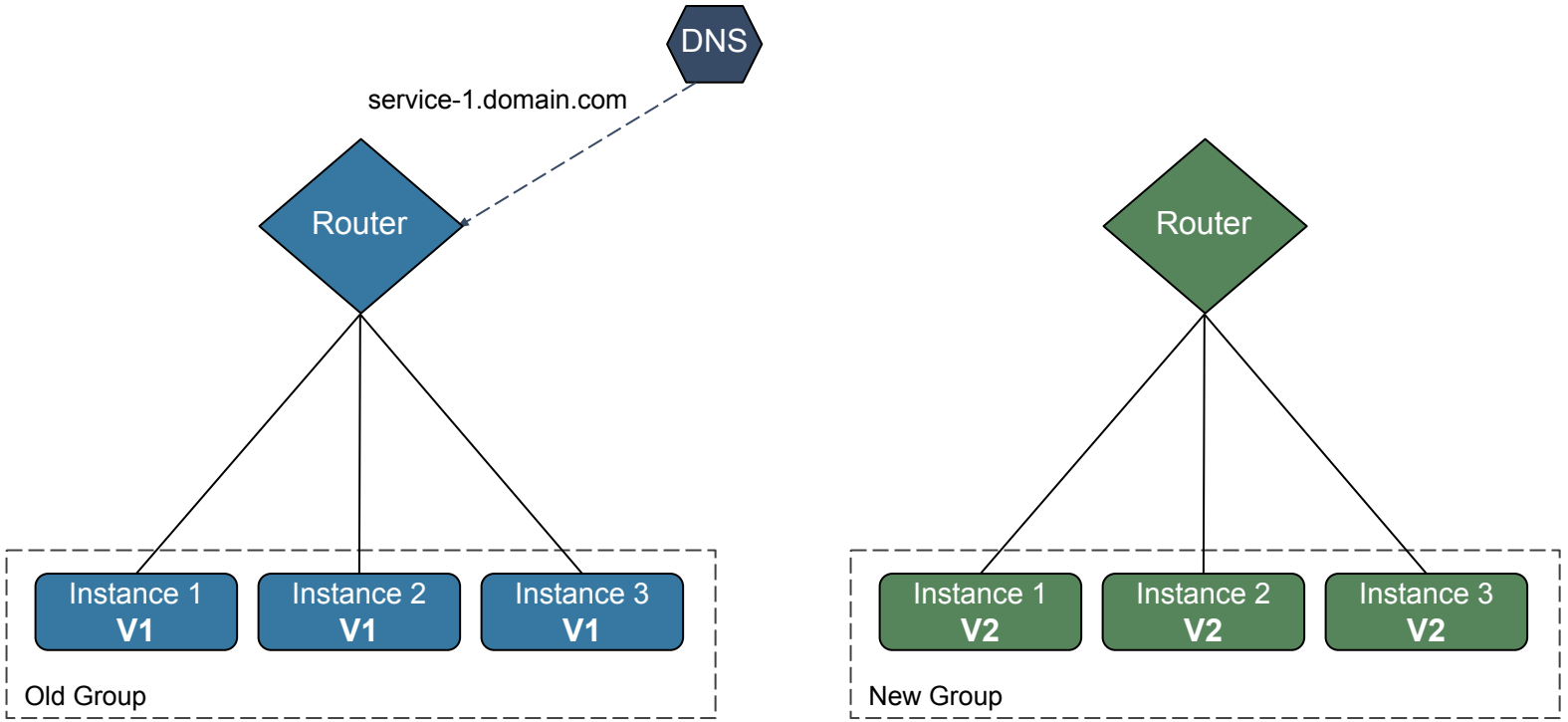
Canary Release



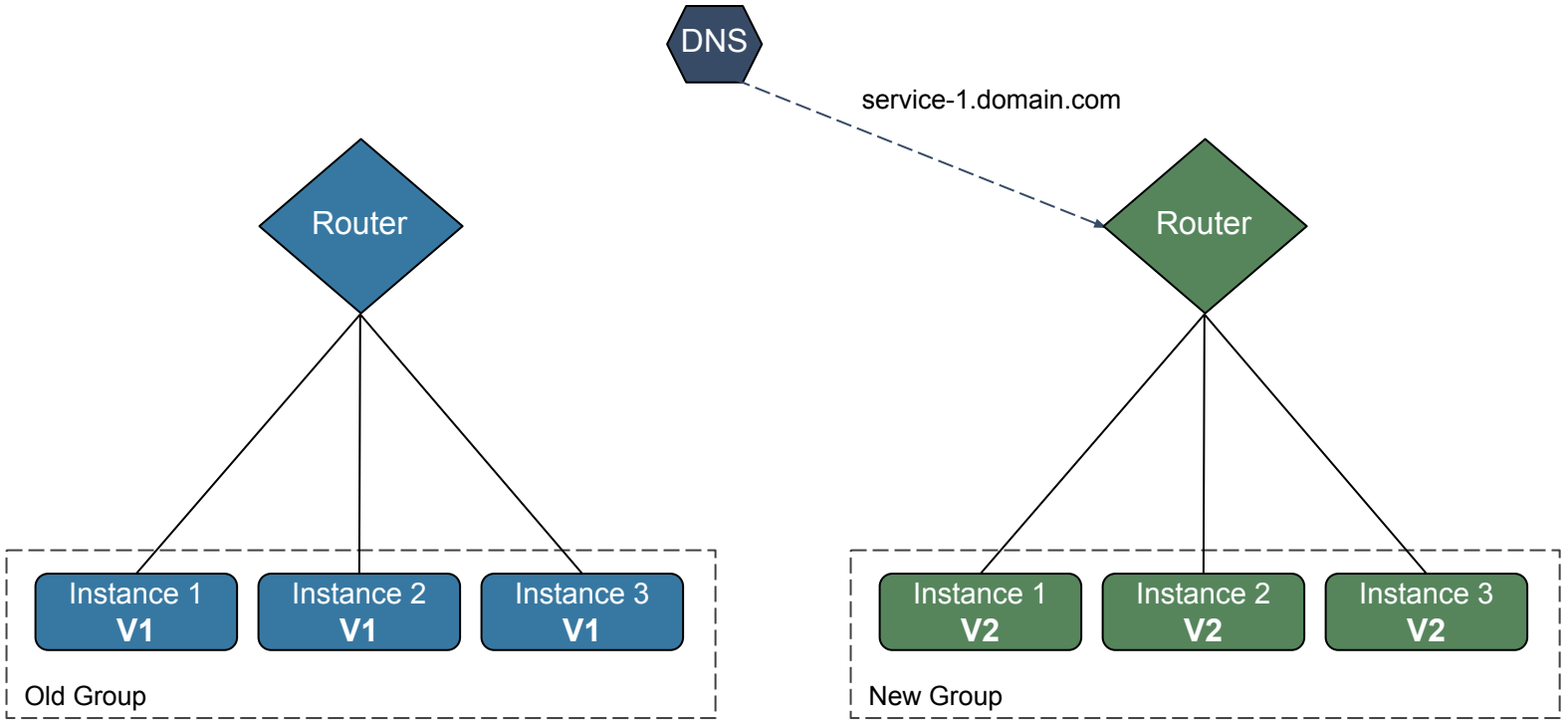
Blue-Green


DNS
service-1.domain.com
Router
Instance 1
V1
Instance 2
V1
Instance 3
V1
Old Group

Blue-Green


DNS
service-1.domain.com
Router
Instance 1
V1
Instance 2
V1
Instance 3
V1
Old Group
Router
Instance 1
V2
Instance 2
V2
Instance 3
V2
New Group

Blue-Green

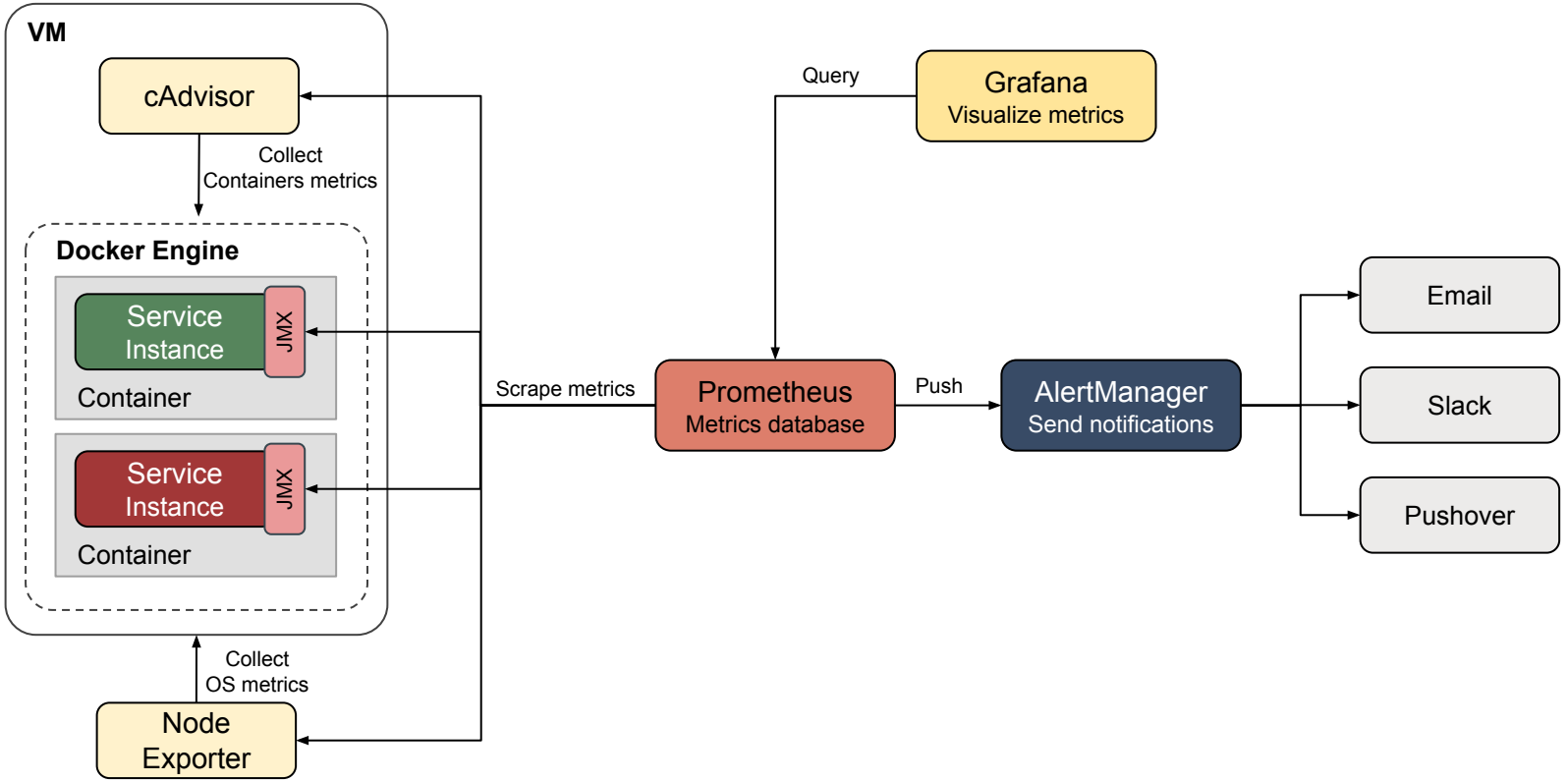

DNS
service-1.domain.com
Router
Router
Instance 1
V1
Instance 2
V1
Instance 3
V1
Old Group
Instance 1
V2
Instance 2
V2
Instance 3
V2
New Group

Monitoring
Microservices

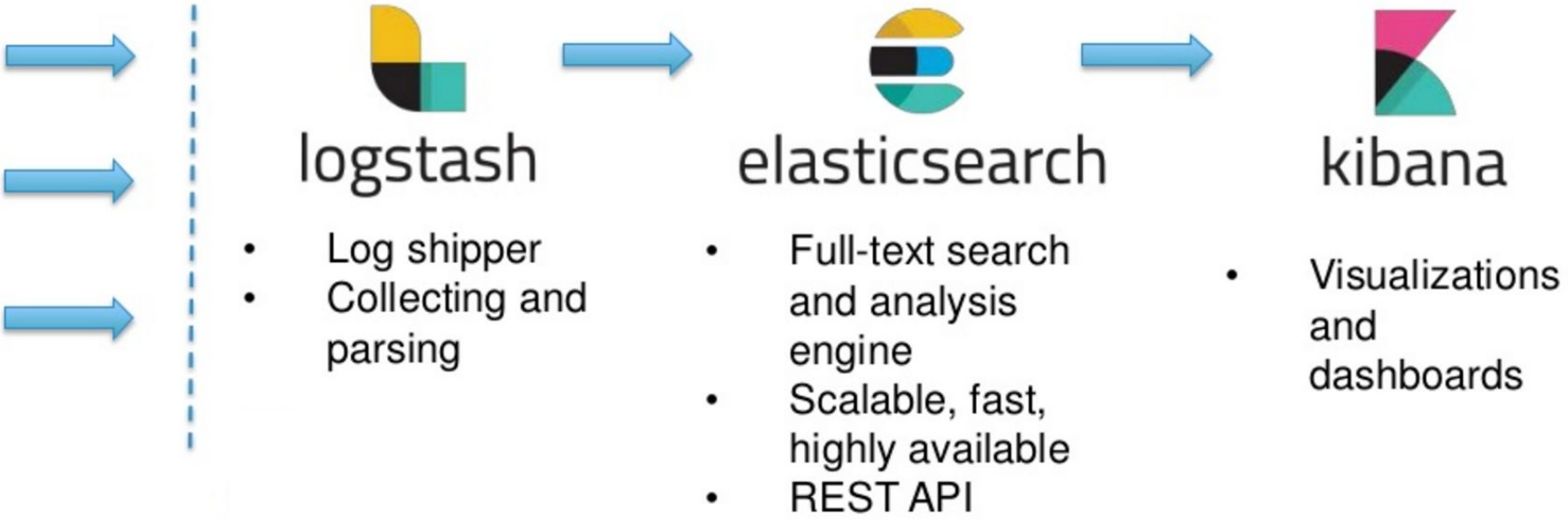
What to monitor?

- **Hosts**
  CPU, Memory, I/O, Network, File system
- **Containers**
  CPU, Memory, I/O, Restarts, Bottleneck
- **Applications**
  Latency, Traffic, Errors

Metrics & Alerts



Logging & Tracing



Forward the foundation!

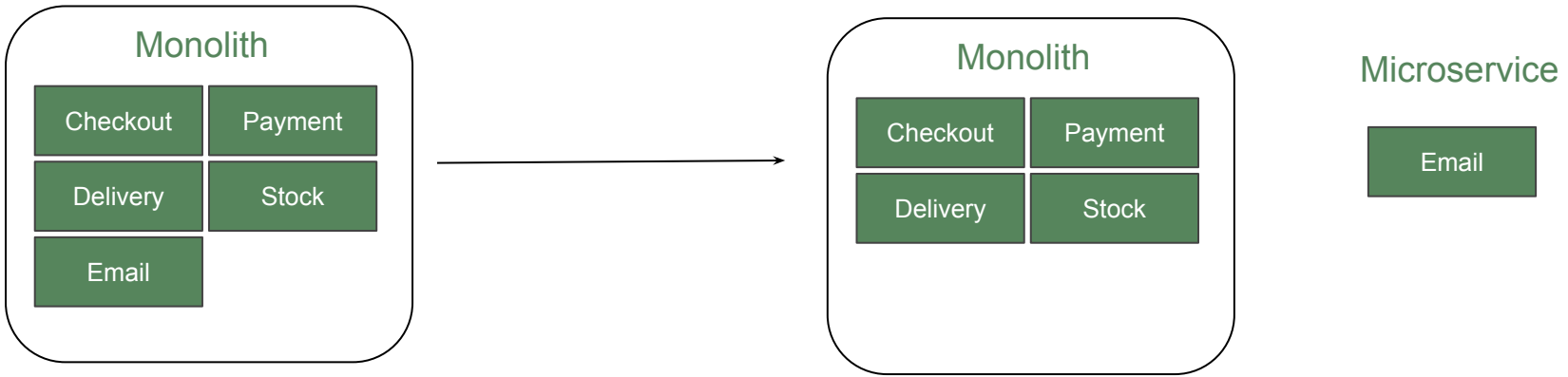
CLOUD NATIVE
COMPUTING FOUNDATION

- Organization that brings together the world's top developers, end users and vendors.
- Hosts critical components of the global technology infrastructure
- Part of the non-profix Linux Foundation

Splitting a Monolith

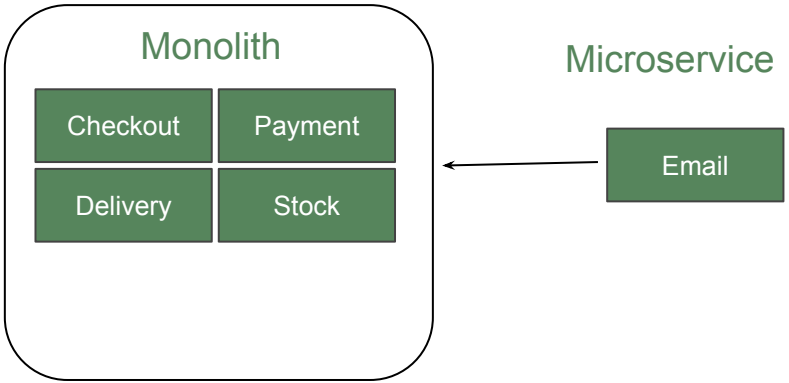
In Microservices

Start with an easy one

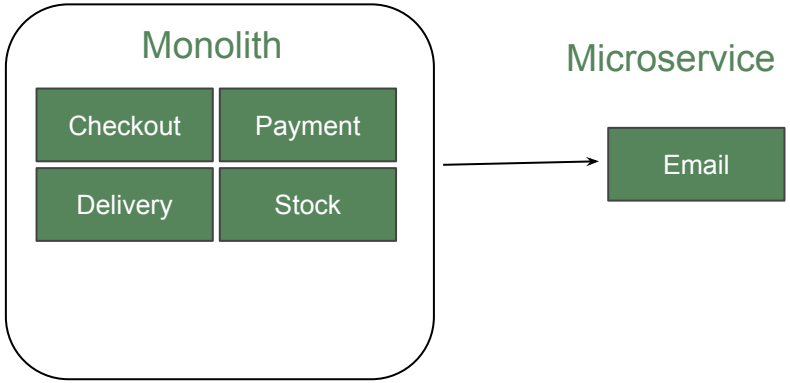


New service should not depend on the monolith

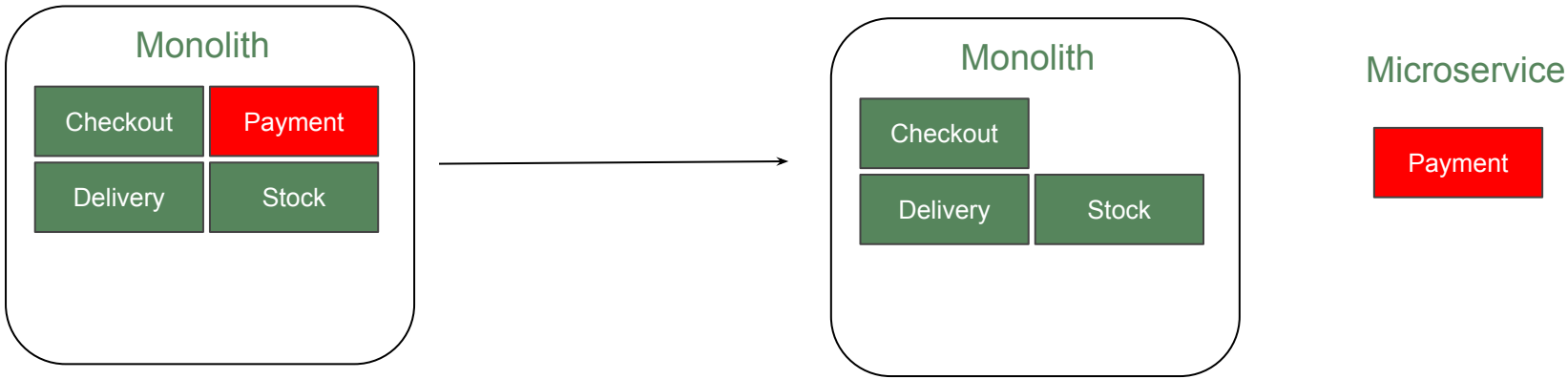
Not okay!



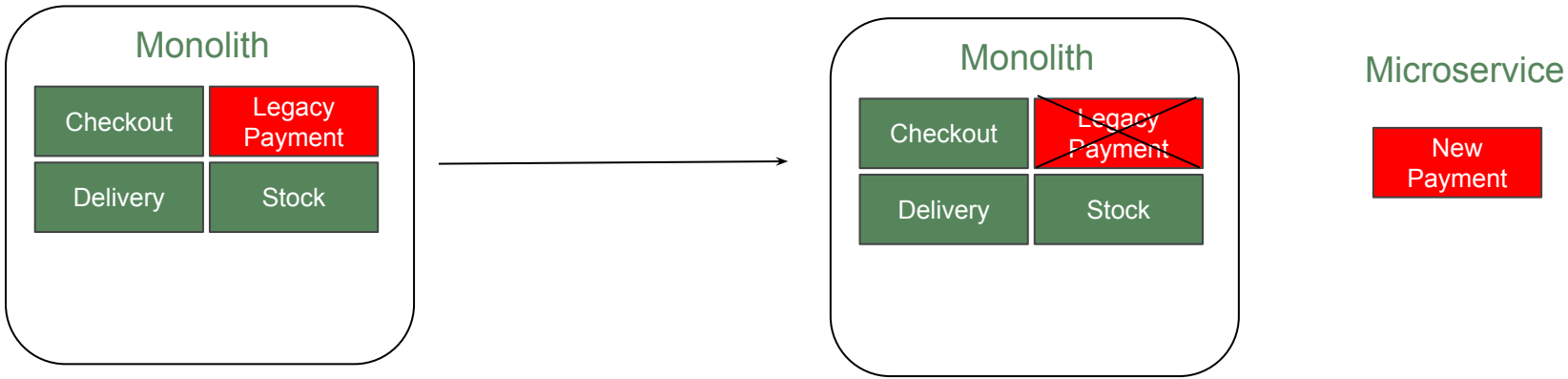
Okay!



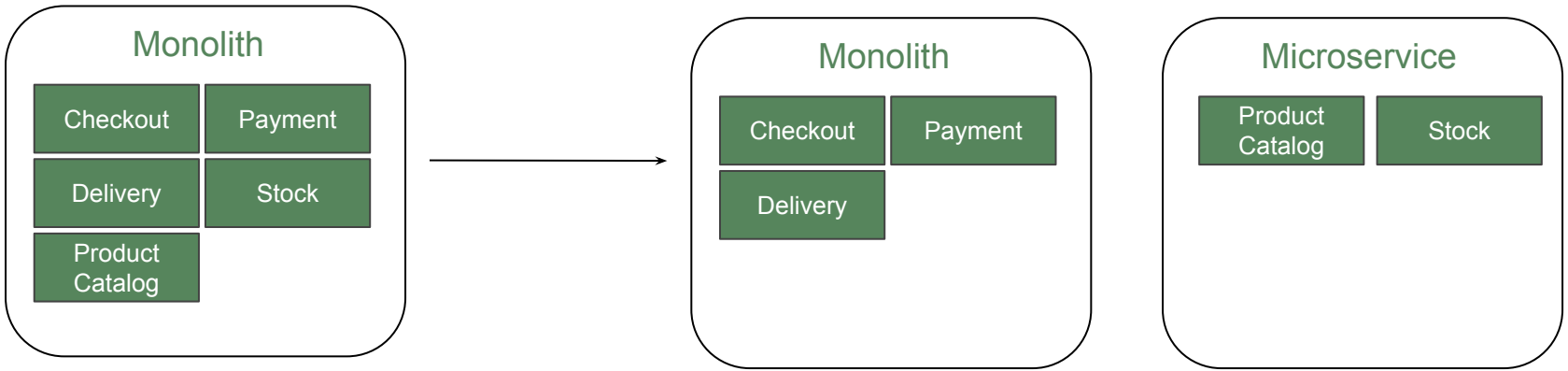
Extract what changes often



Most of times it is better to rewrite and retire the old code



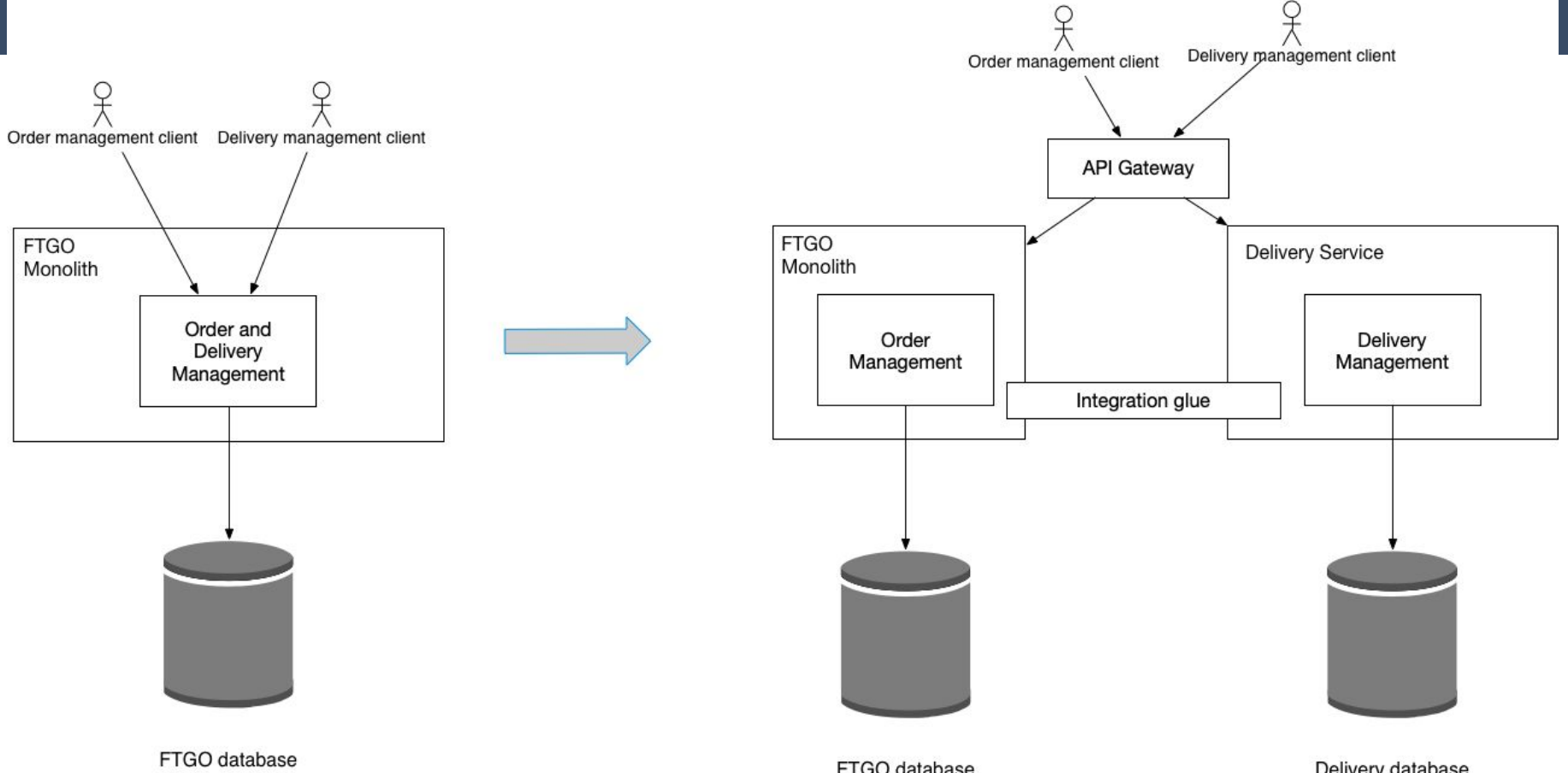
Better to go from big to small



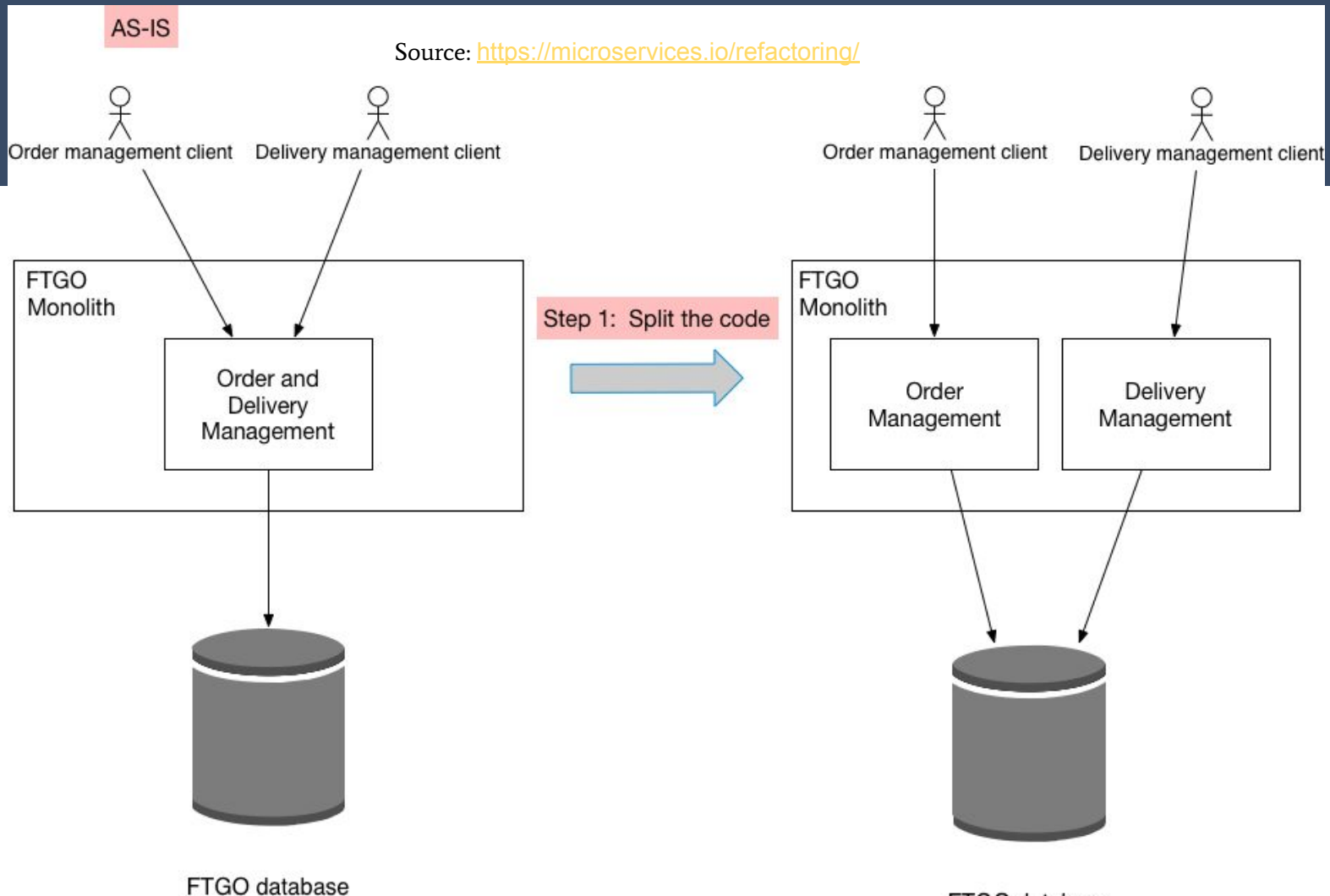
AS-IS

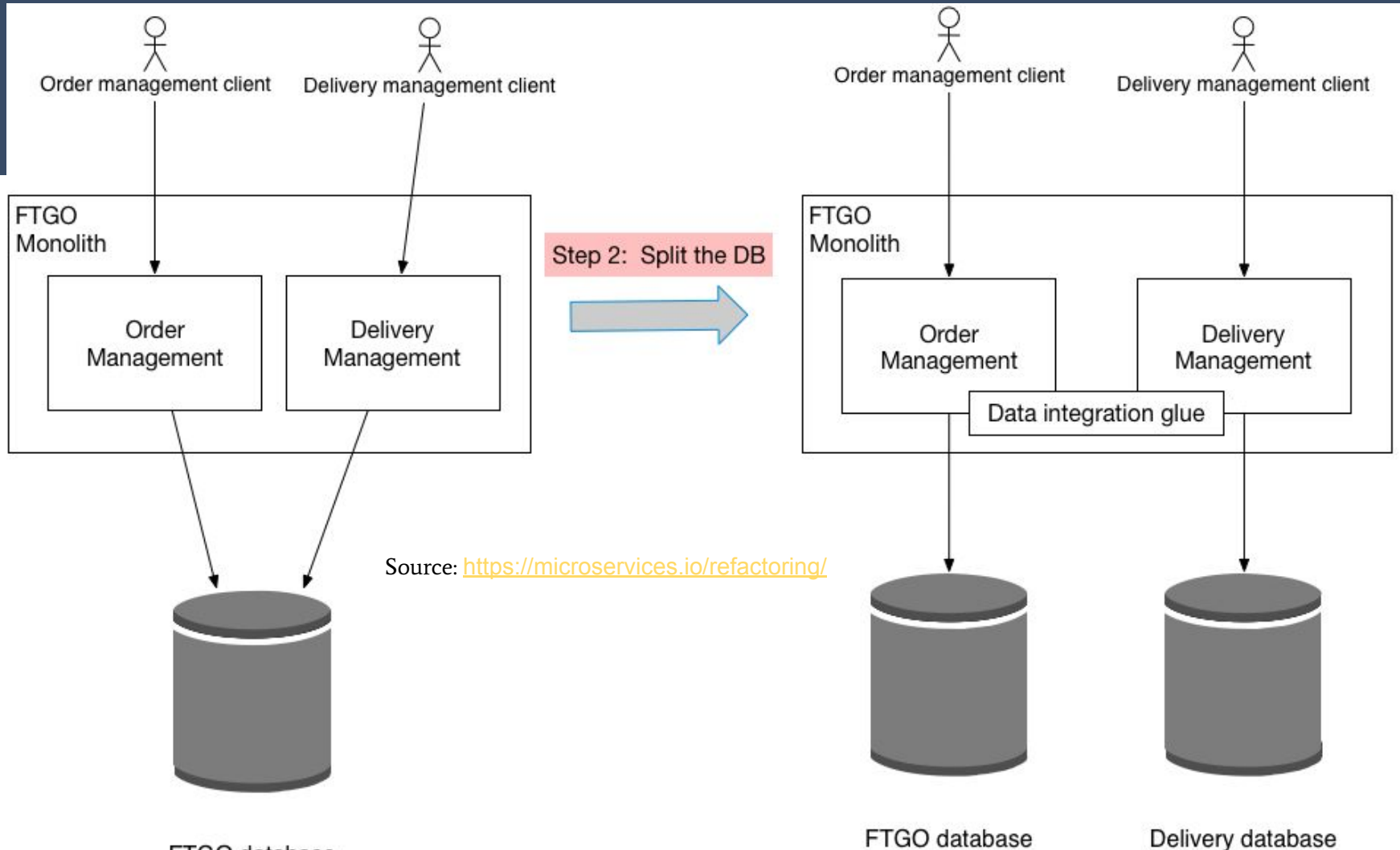
TO-BE

Source: https://microservices.io/refactoring/

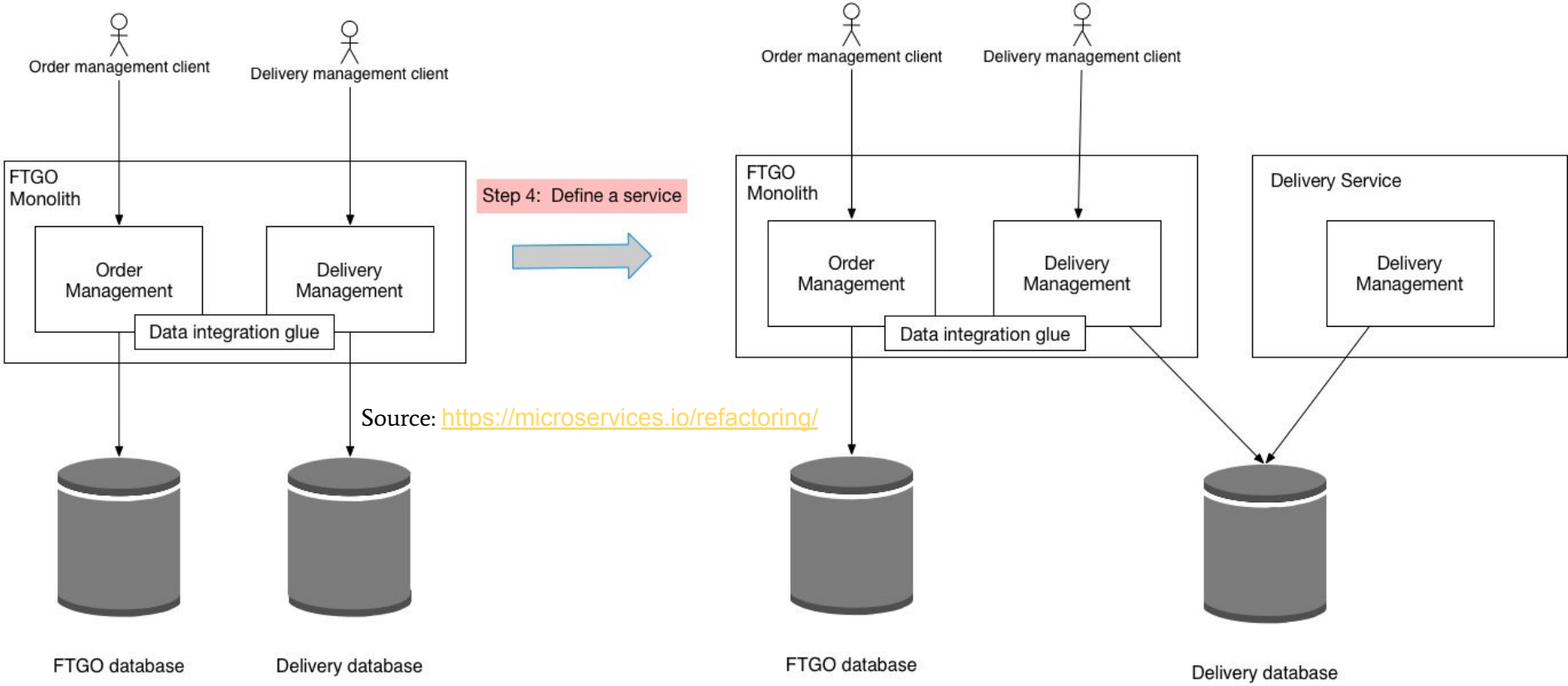


Source: https://microservices.io/refactoring/



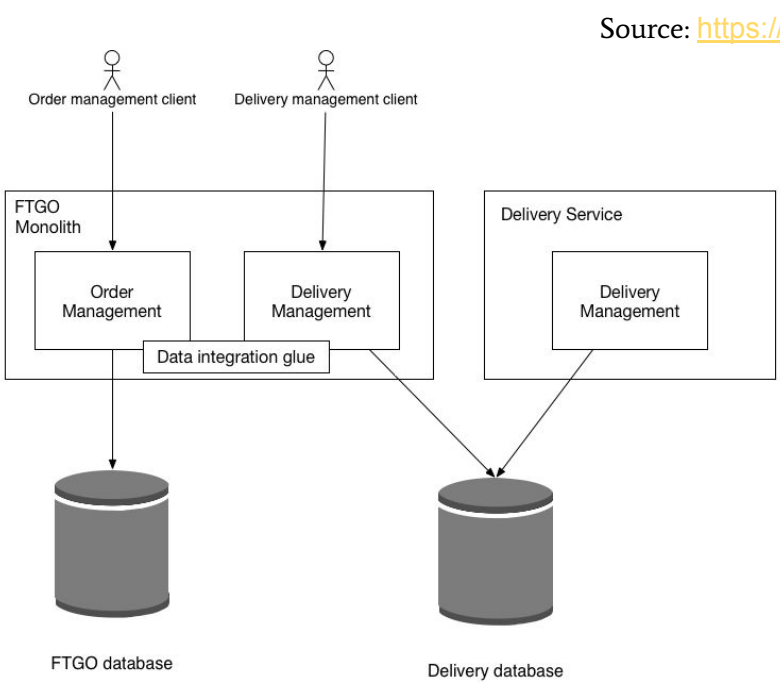


Source: https://microservices.io/refactoring/

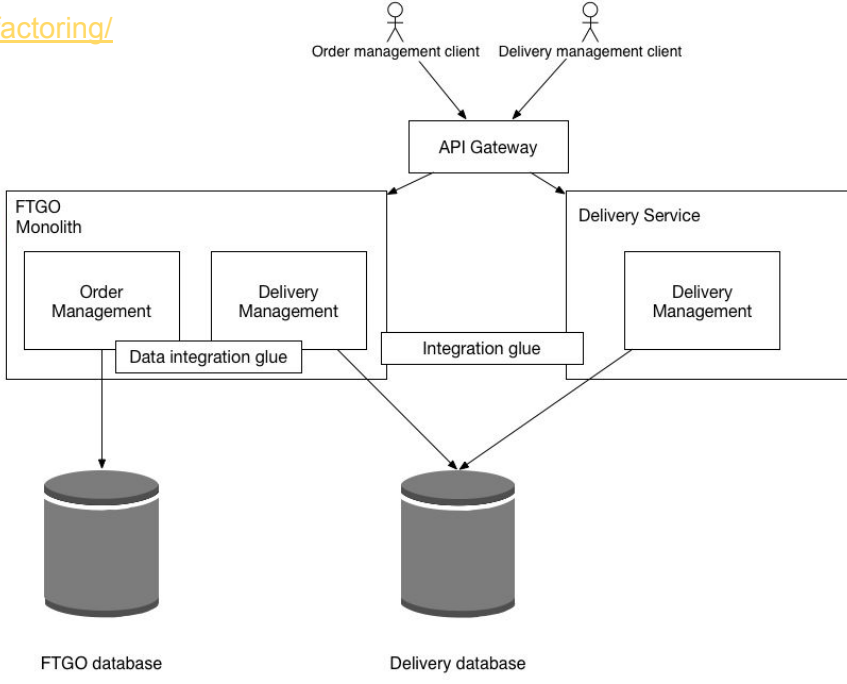


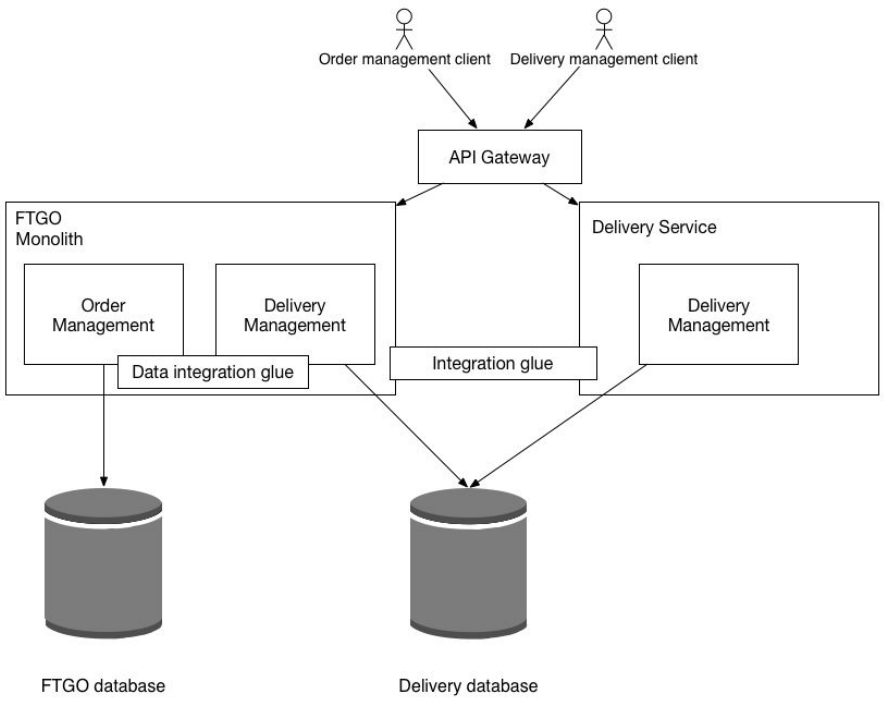
Source: https://microservices.io/refactoring/

Source: https://microservices.io/refactoring/



Step 4: Use Delivery Service





Step 5: Delete old code