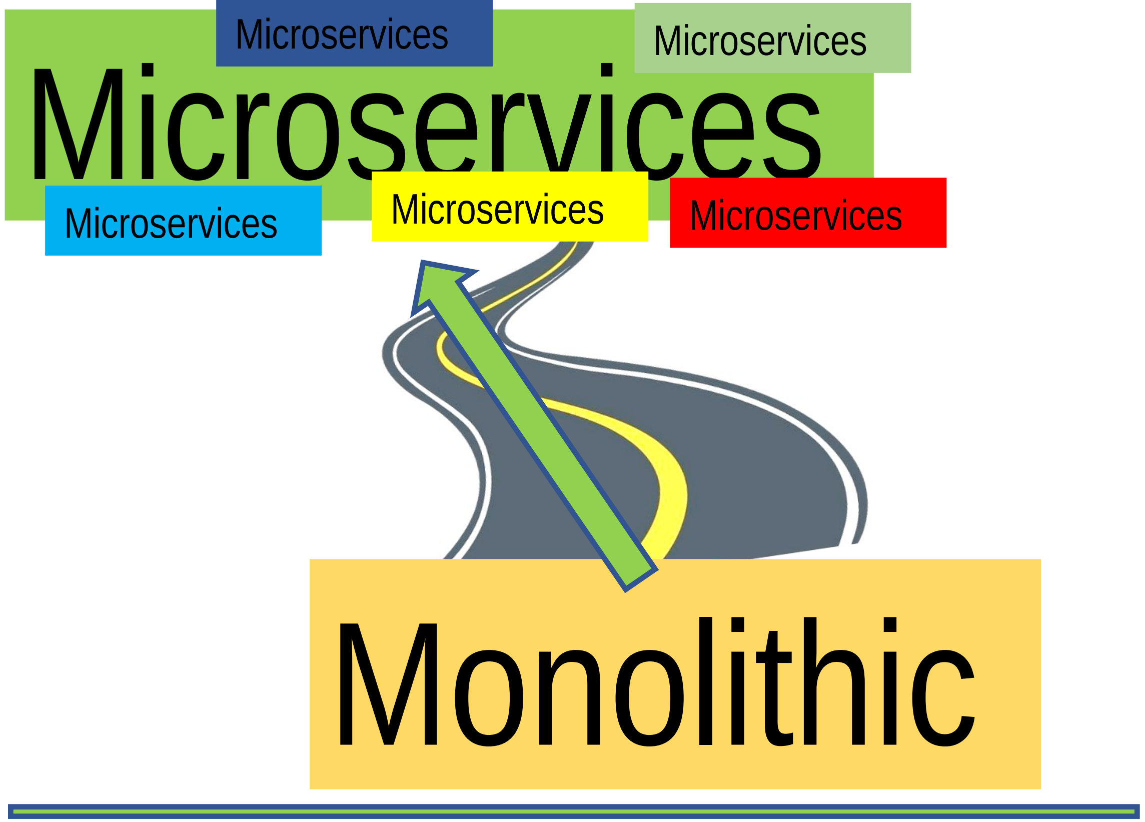
Microservices Architecture

Rajeev Gupta
Java Trainer & Consultant

Microservices Architecture



Microservices

Microservices

Microservices

Microservices

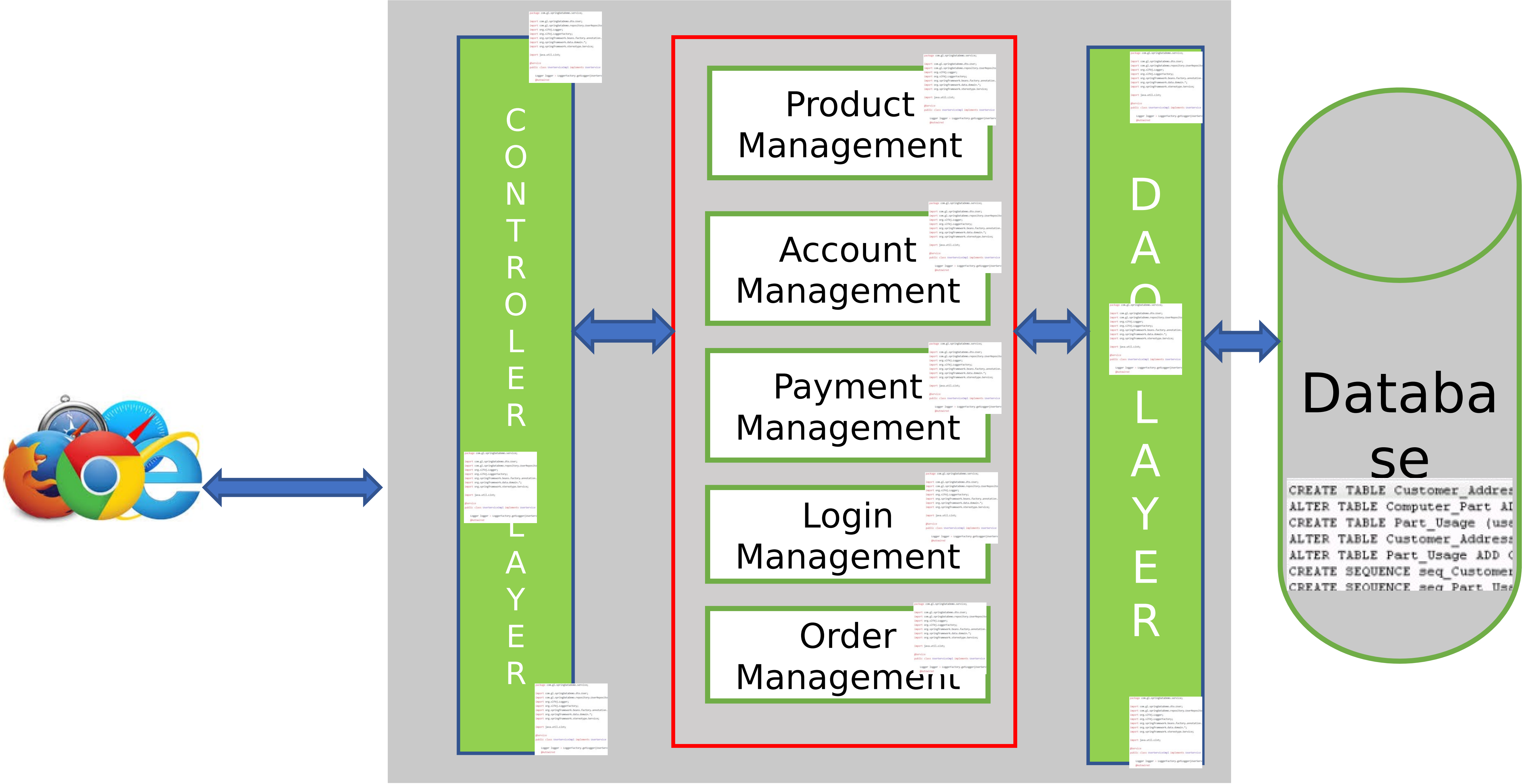
Microservices

Microservices

Monolithic

Monolith Applications

Online Shopping Portal

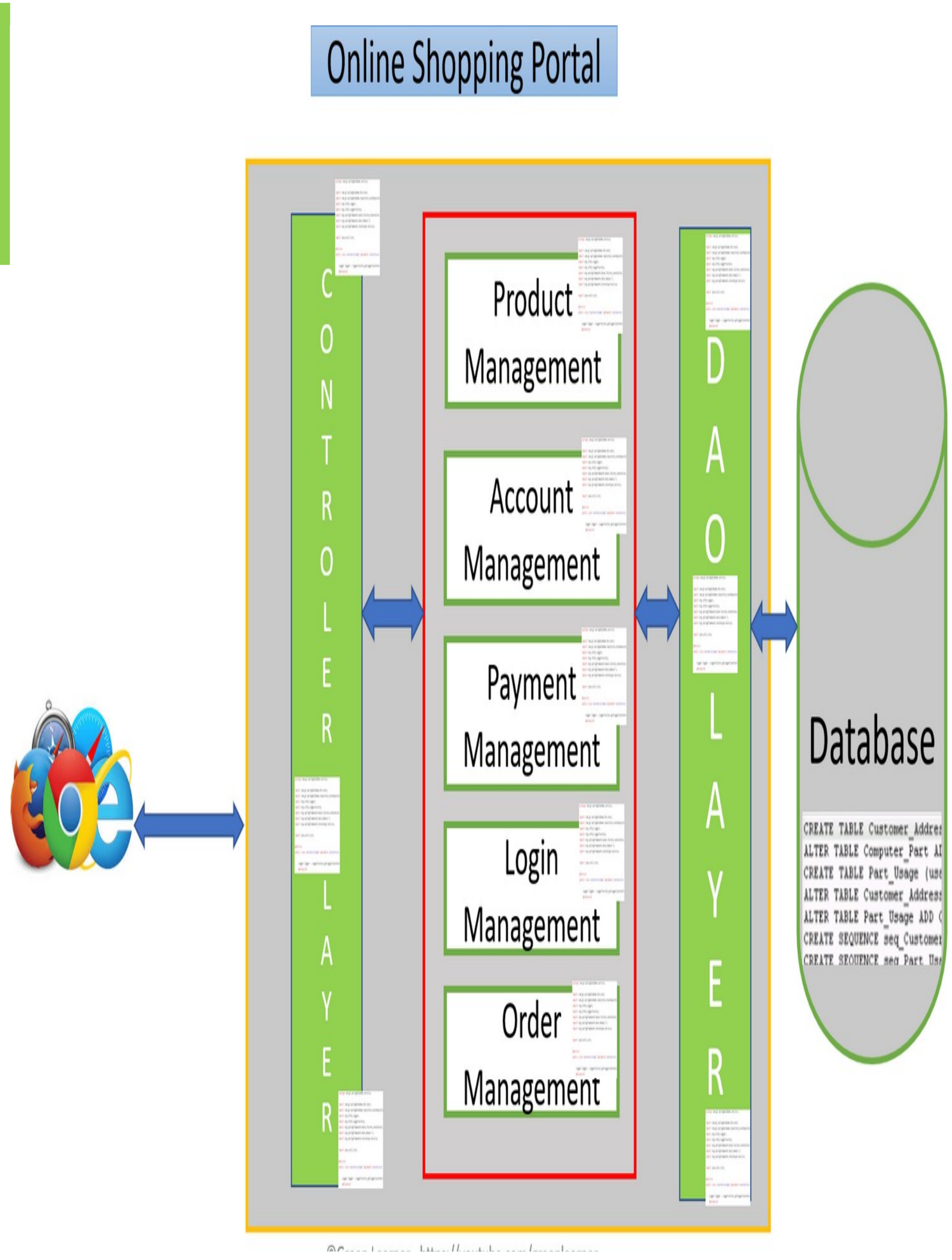


```
package com.gl.springDataDemo.service;

import com.gl.springDataDemo.dto.User;
import com.gl.springDataDemo.repository.UserReposito
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.
import org.springframework.data.domain.*;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class UserServiceImpl implements UserService

    Logger logger = LoggerFactory.getLogger(UserServ
    @Autowired
```

```
CREATE TABLE Customer_Addres
ALTER TABLE Computer_Part AD
CREATE TABLE Part_Usage (use
ALTER TABLE Customer_Address
ALTER TABLE Part_Usage ADD C
CREATE SEQUENCE seq_Customer
CREATE SEQUENCE seq_Part_Usa
```

Monolith – positive aspects

- Easy to develop
- Simple testing
- Quick deployment

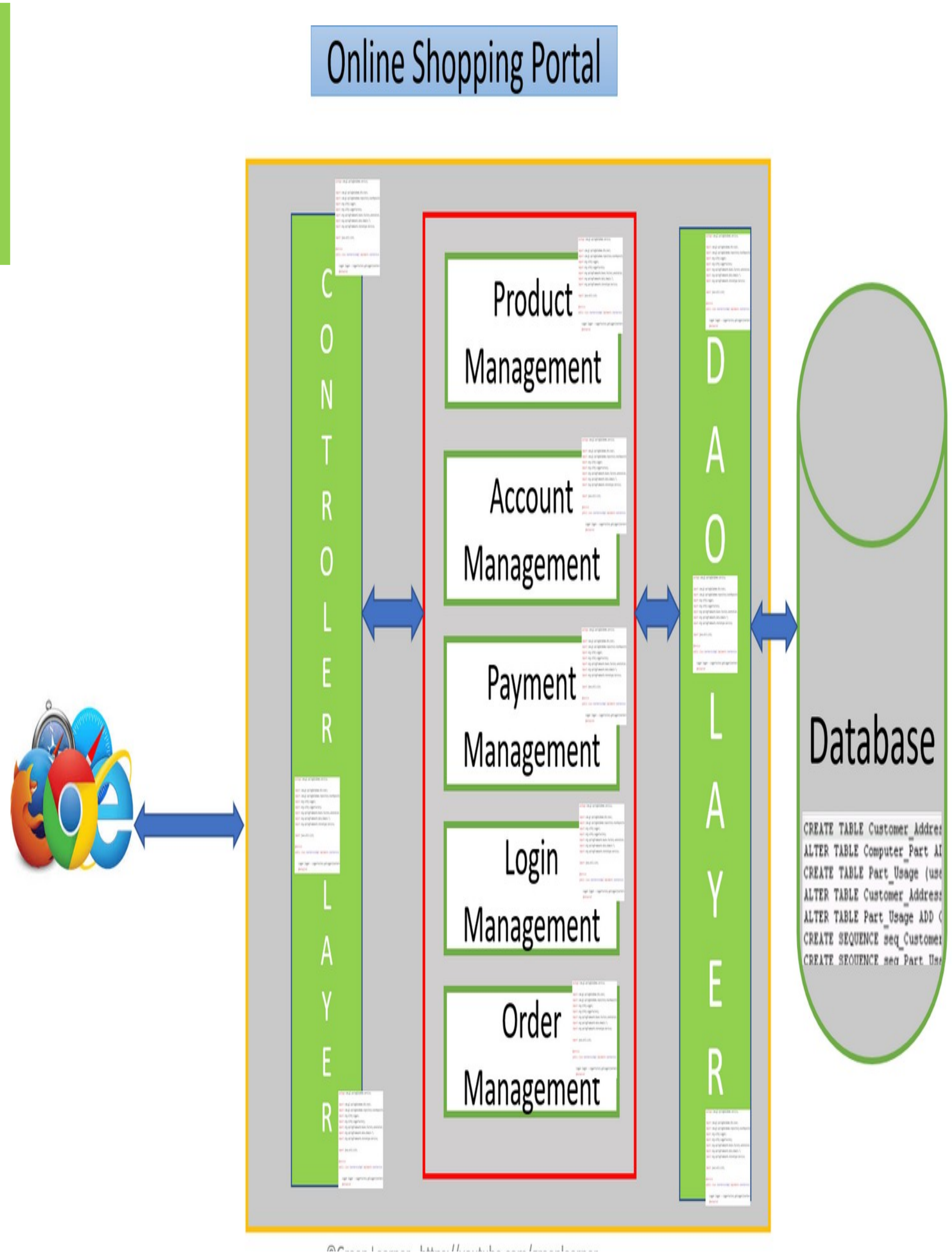


```
CREATE TABLE Customer_Addres
ALTER TABLE Computer_Part AD
CREATE TABLE Part_Usage (use
ALTER TABLE Customer_Address
ALTER TABLE Part_Usage ADD (
CREATE SEQUENCE seq_Customer
CREATE SEQUENCE seq_Part_Us
```

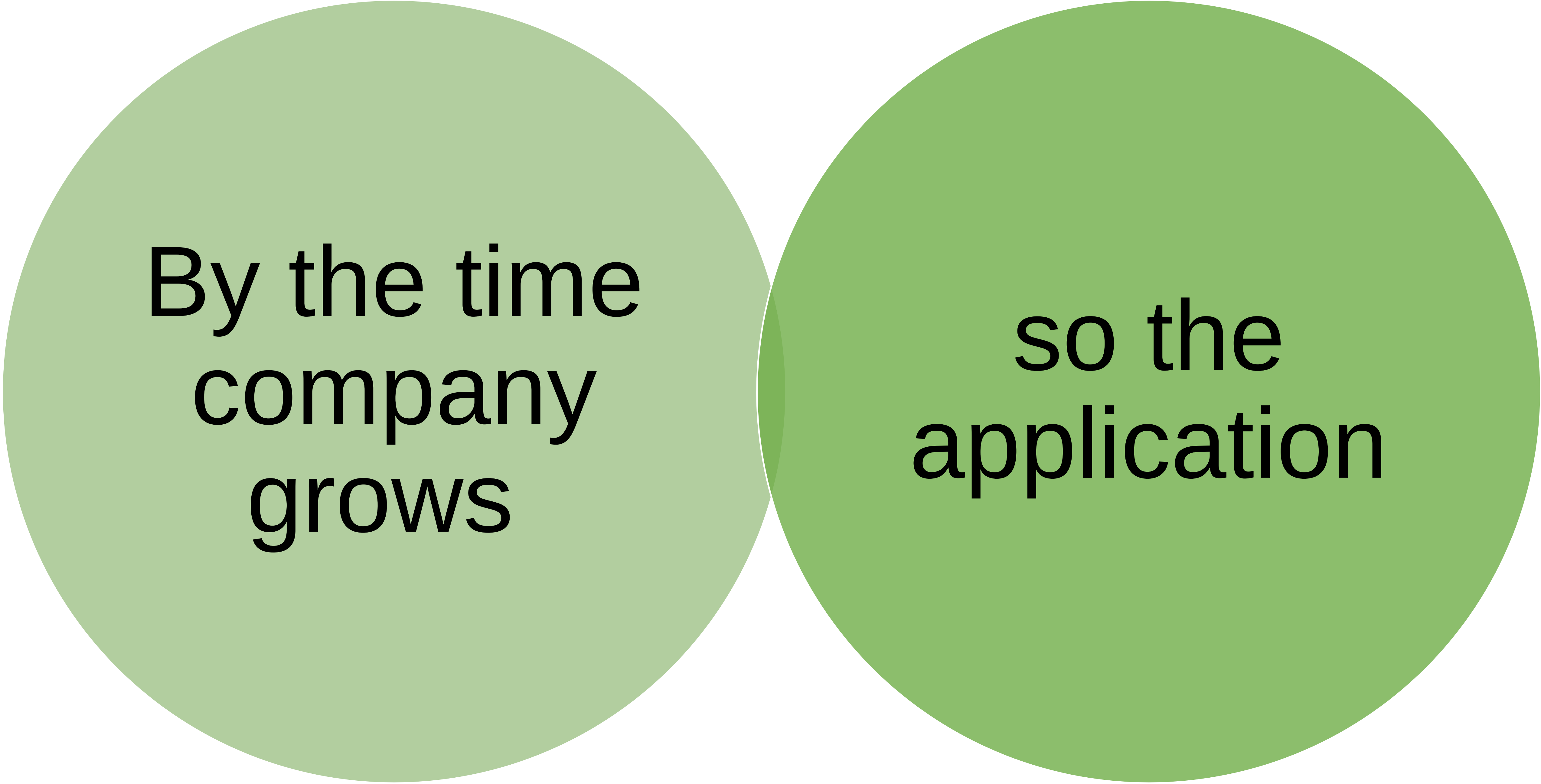
©Green Learner https://youtube.com/greenlearner

Monolith – positive aspects

- Easy to scale by having multiple copy of same application(horizontal scaling)
- Less technicality
- Better for small scale apps



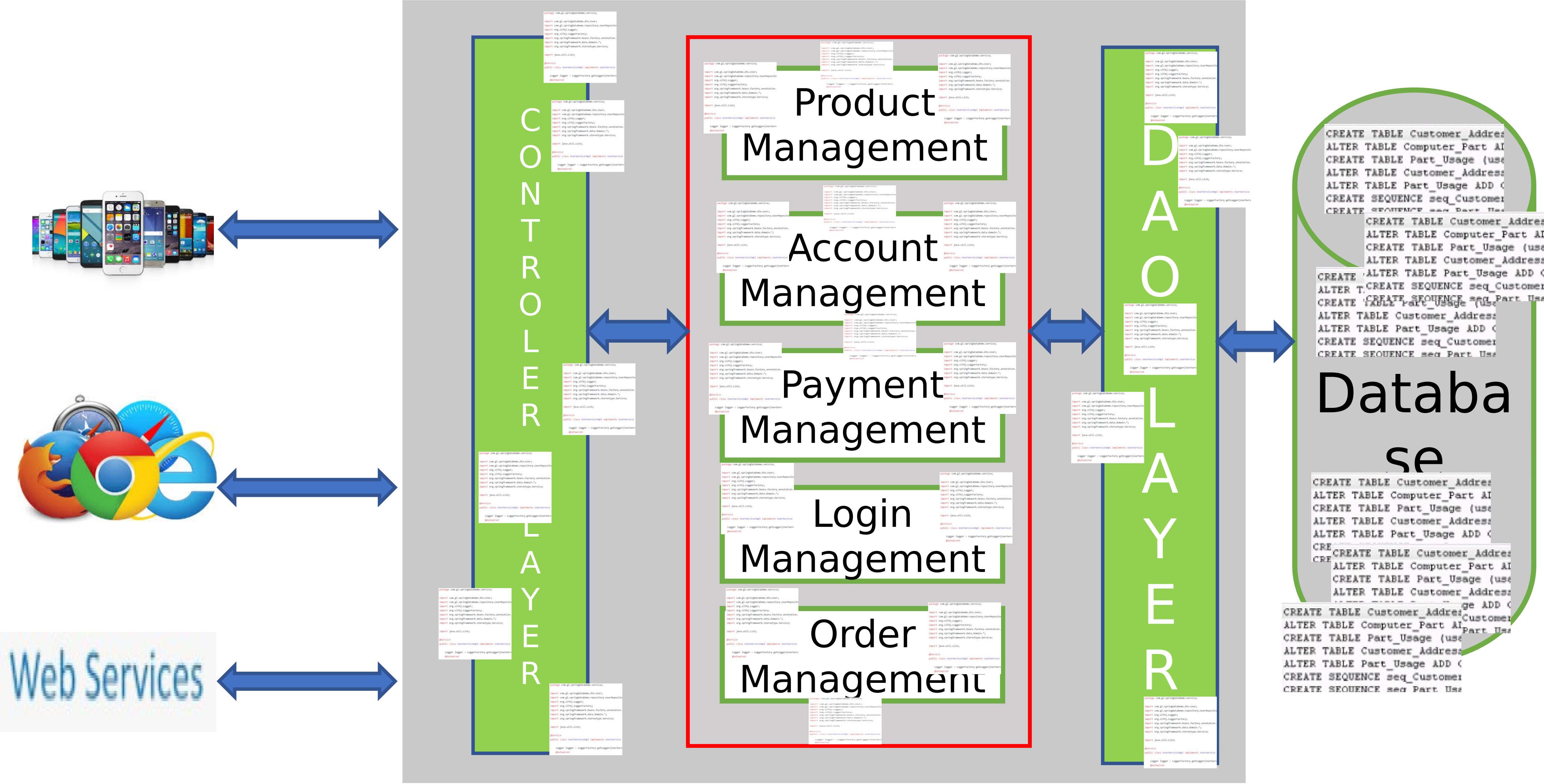
Online Shopping Portal
CONTROLLER LAYER
Product Management
Account Management
Payment Management
Login Management
Order Management
DAO LAYER
Database
CREATE TABLE Customer_Addres
ALTER TABLE Computer_Part AD
CREATE TABLE Part_Usage (use
ALTER TABLE Customer_Address
ALTER TABLE Part_Usage ADD (
CREATE SEQUENCE seq_Customer
CREATE SEQUENCE seq_Part_Us
©Green Learner https://youtube.com/greenlearner



By the time company grows

so the application

Online Shopping Portal



CONTROLER LAYER

Product Management

Account Management

Payment Management

Login Management

Order Management

DAO LAYER

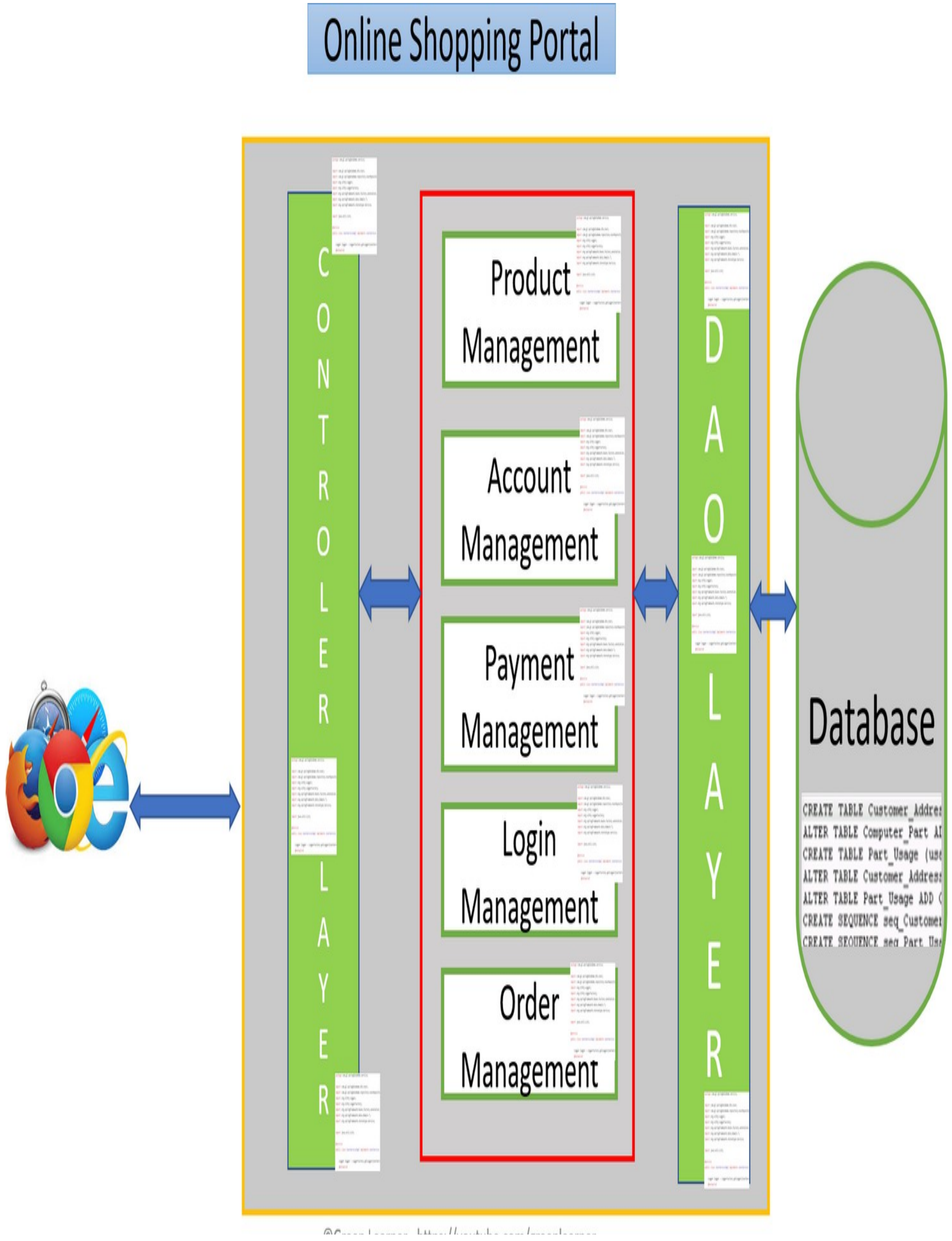
Database

Web Services

```
CREATE TABLE Customer_Addre
ALTER TABLE Computer_Part AI
CREATE TABLE Part_Usage (use
ALTER TABLE Customer_Address
ALTER TABLE Part_Usage ADD C
CREATE SEQUENCE seq_Customer
CREATE SEQUENCE seq_Part_Usa
```

Monolith – Challenges

- Limitation in size
- Complexity grows with time
- Long time to release new features
- More time to send the fix for production bug
- Even small change in one module needs redeployment of whole application


Online Shopping Portal
CONTROLLER LAYER
Product Management
Account Management
Payment Management
Login Management
Order Management
DAO LAYER
Database
CREATE TABLE Customer_Addre
ALTER TABLE Computer_Part AD
CREATE TABLE Part_Usage (use
ALTER TABLE Customer_Address
ALTER TABLE Part_Usage ADD C
CREATE SEQUENCE seq_Customer
CREATE SEQUENCE seq_Part_Usa

©Green Learner https://youtube.com/greenlearner

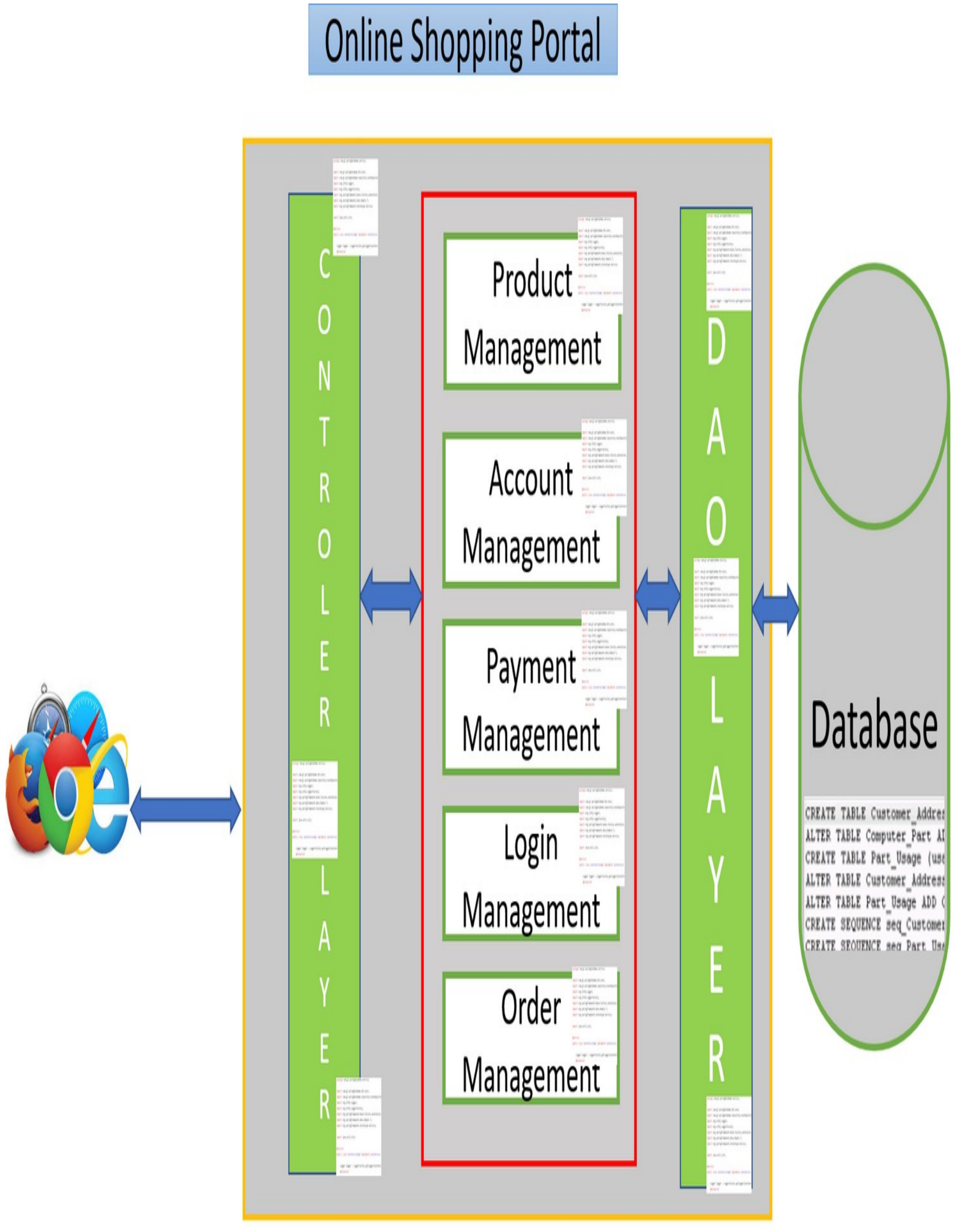
Monolith – Challenges

High Dependency on few human resources

Hiring new team and making them understand whole application is tough

Stuck in one technology

Single point failure

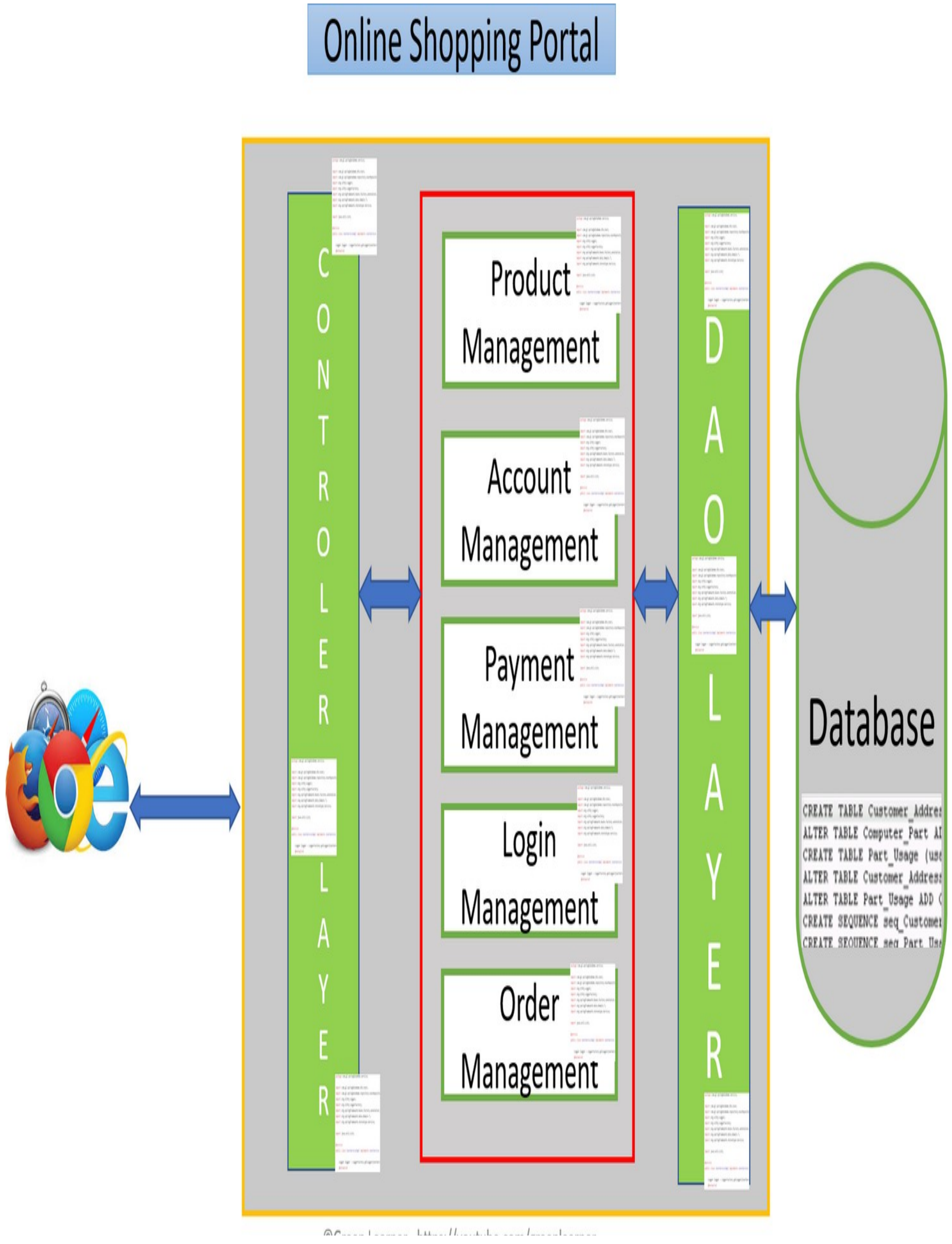


```
CREATE TABLE Customer_Addres
ALTER TABLE Computer_Part AD
CREATE TABLE Part_Usage (use
ALTER TABLE Customer_Address
ALTER TABLE Part_Usage ADD (
CREATE SEQUENCE seq_Customer
CREATE SEQUENCE seq_Part_Use
```

©Green Learner https://youtube.com/greenlearner

Monolith – Challenges

- Continuous deployment is difficult
- Difficult to scale when we have large code base
- High coupling between modules
- Reliability and availability problem



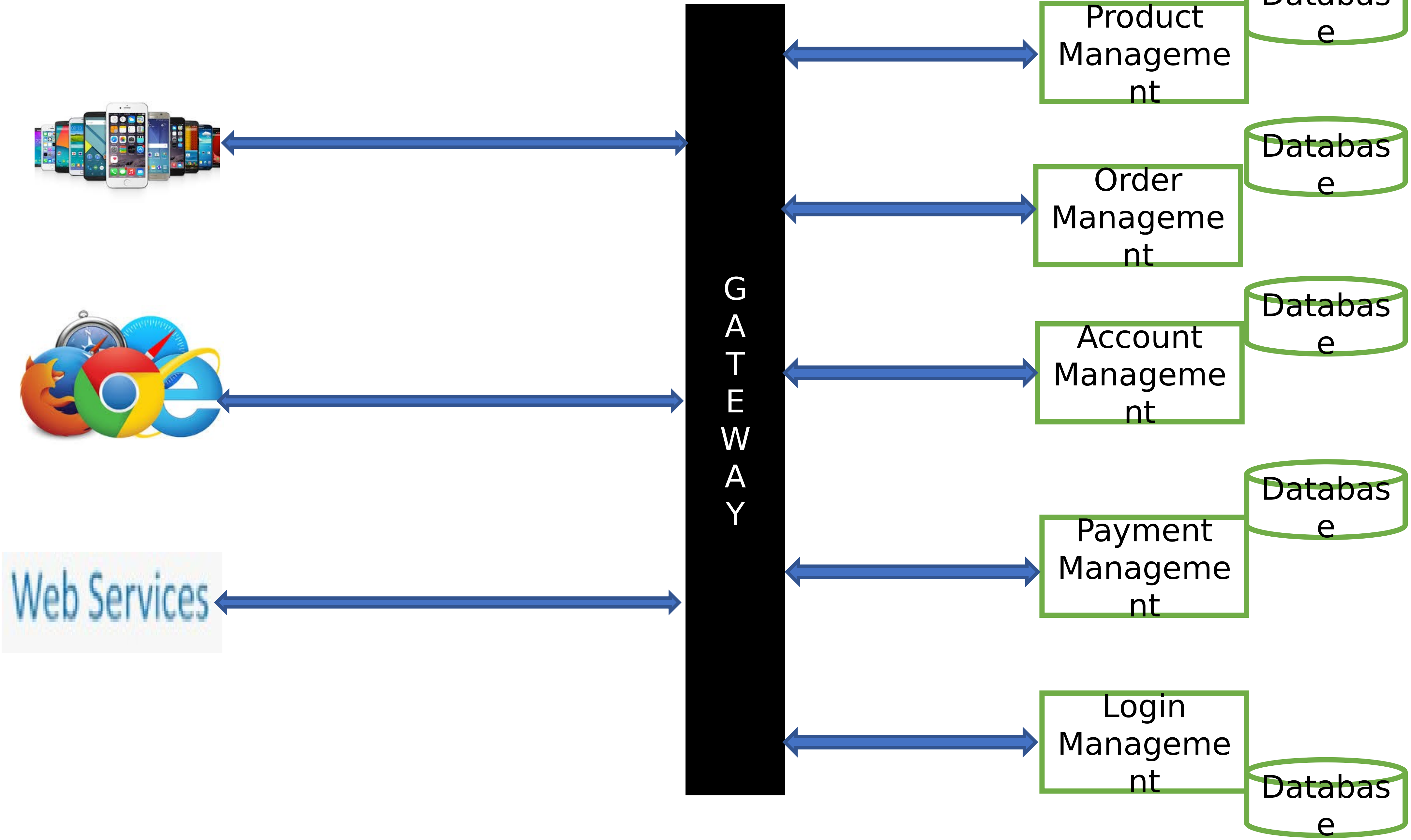
```
CREATE TABLE Customer_Addres
ALTER TABLE Computer_Part AD
CREATE TABLE Part_Usage (use
ALTER TABLE Customer_Address
ALTER TABLE Part_Usage ADD (
CREATE SEQUENCE seq_Customer
CREATE SEQUENCE seq_Part_Use
```

©Green Learner https://youtube.com/greenlearner

What’s the solution then??

Microservices

Online Shopping Portal



GATEWAY

Product Management

Database

Order Management

Database

Account Management

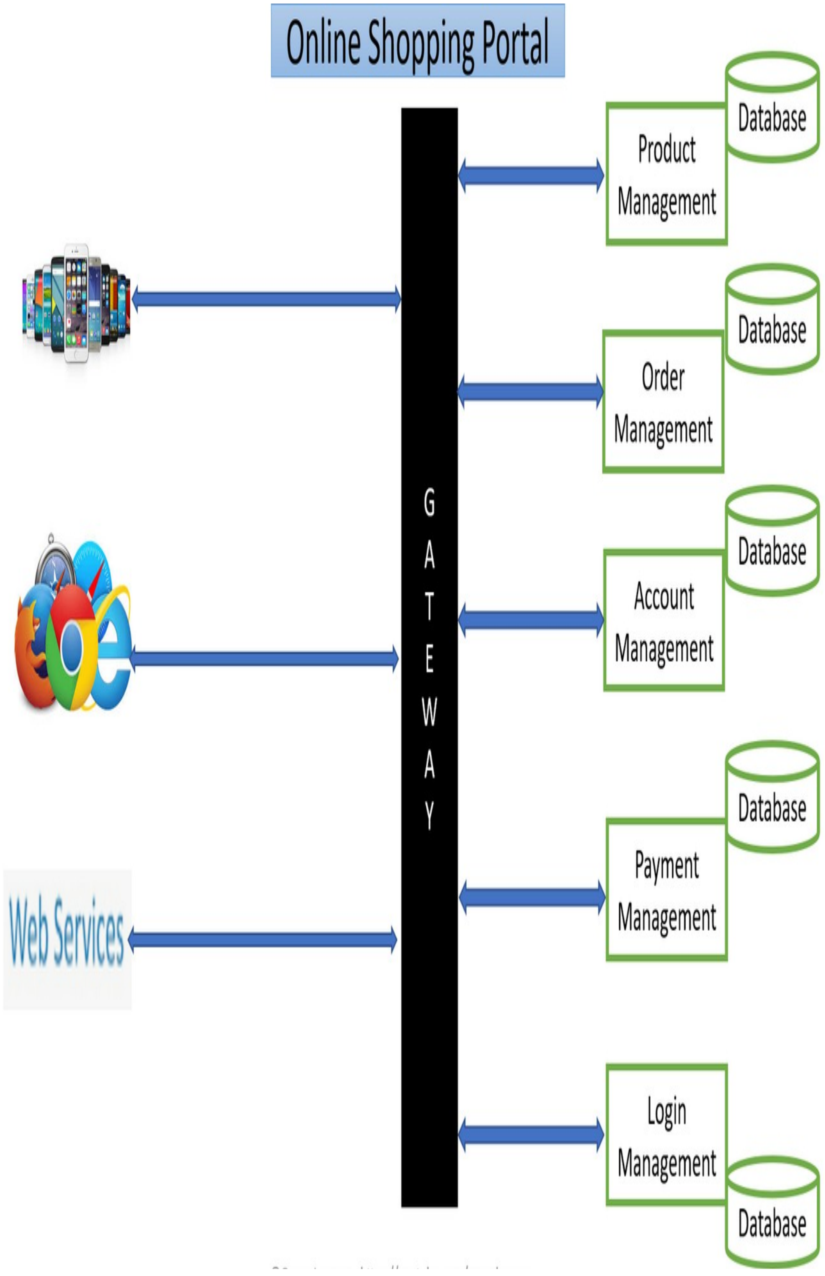
Database

Payment Management

Database

Login Management

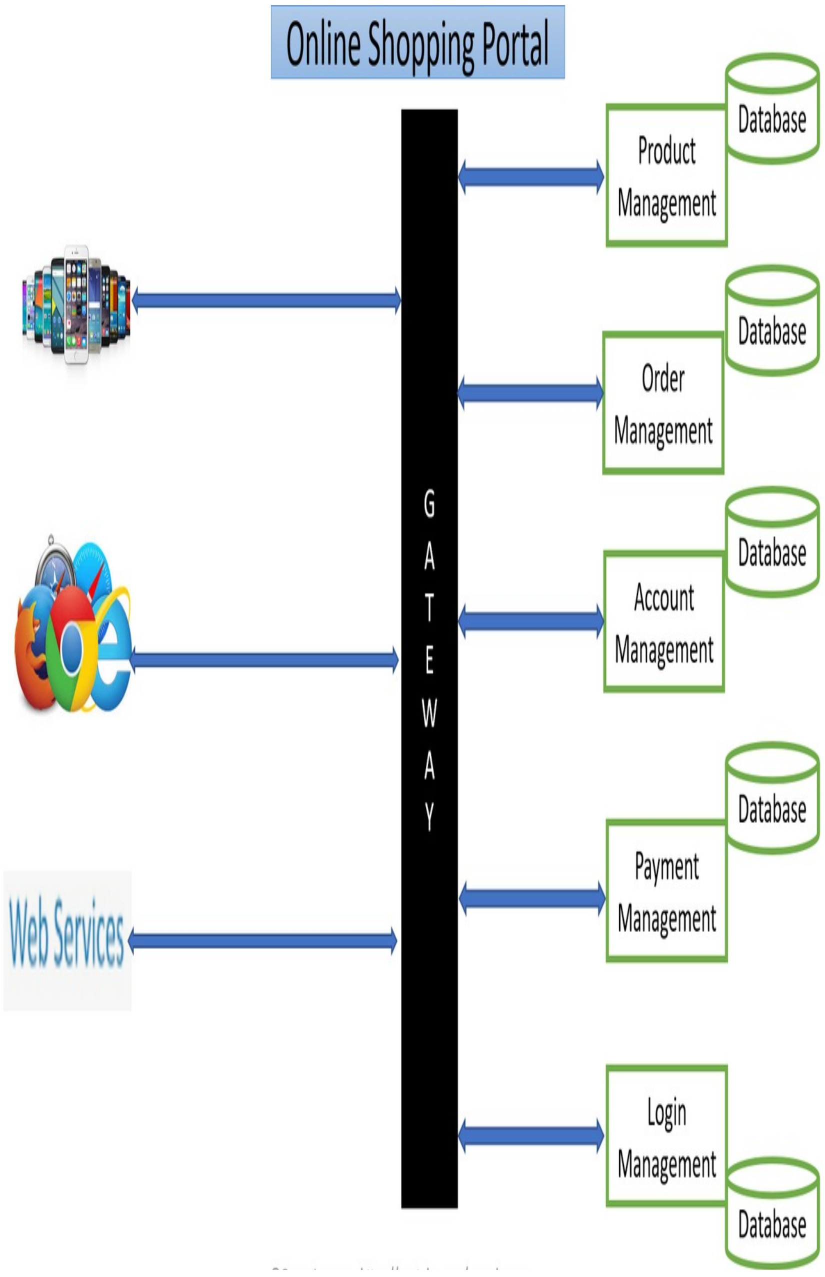
Database



Microservices : Positive aspects

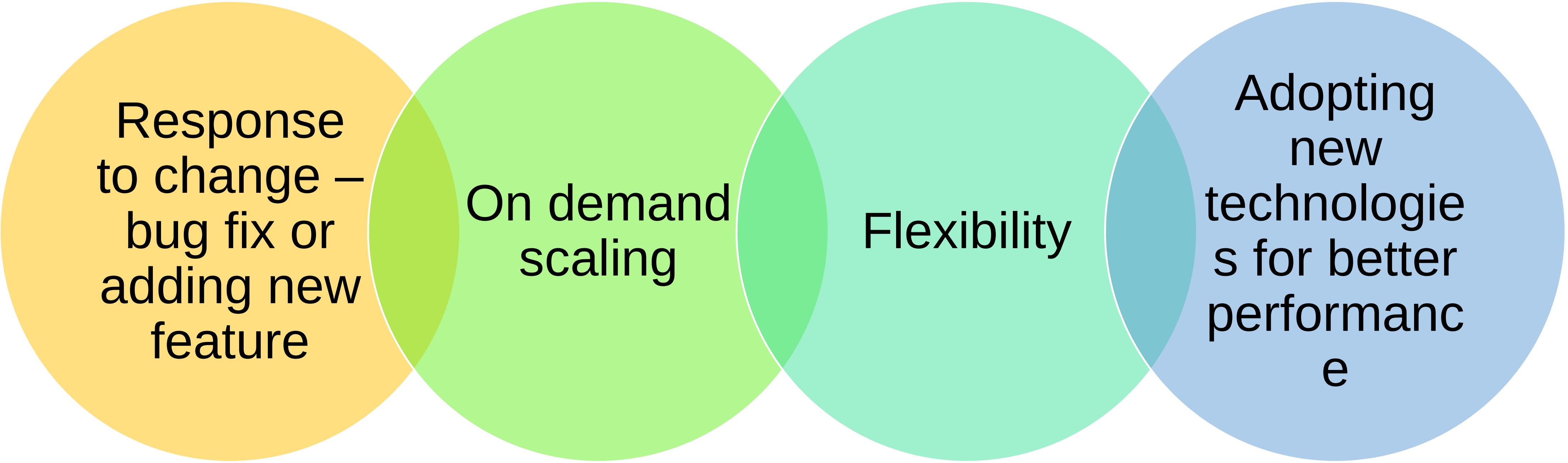
- Domain expertise
- Easy and quick to scale – on demand
- Isolated decision making
- Self Organisation
- Quick response to change
- Increase uptime
- Can experiment with any tech
- Loose coupling
- Service reusability
- Agile, SCRUM
- Best for large scale apps

Microservices :Challenges



- Additional complexity with distributed systems
- Deployment complexity
- Monitoring complexity
- Increased resource consumption
- Communication among services is challenging
- Testing each service is also a challenge
- Maintaining transnationality among services

Fundamental problems with Monoliths which can’t be solved without changing the architecture??



Response to change – bug fix or adding new feature

On demand scaling

Flexibility

Adopting new technologies for better performance

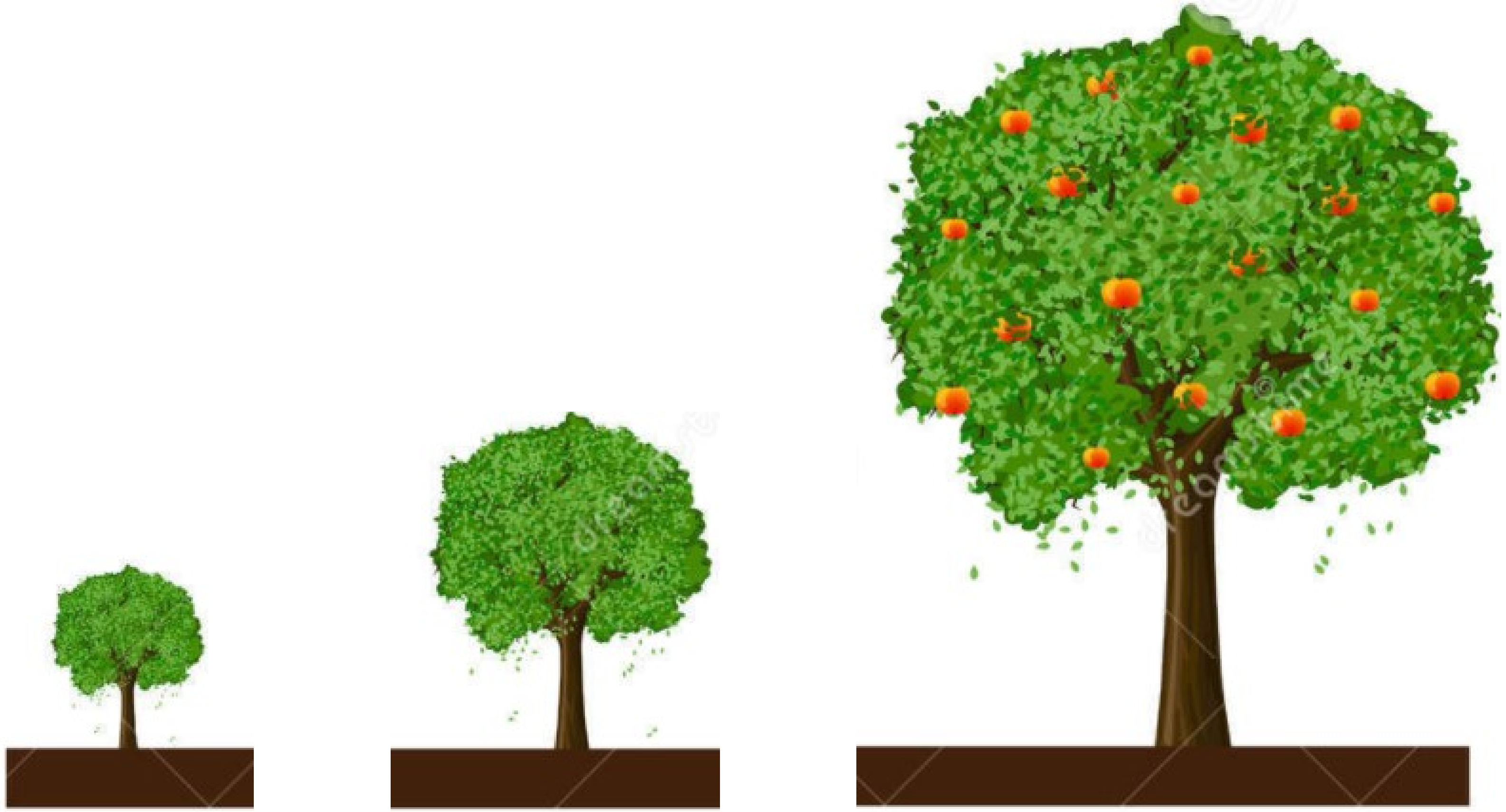
What next??

Microservices : Design principles

Microservices Architecture

Microservices: design principles

Why??
Is it mandatory to go through these?

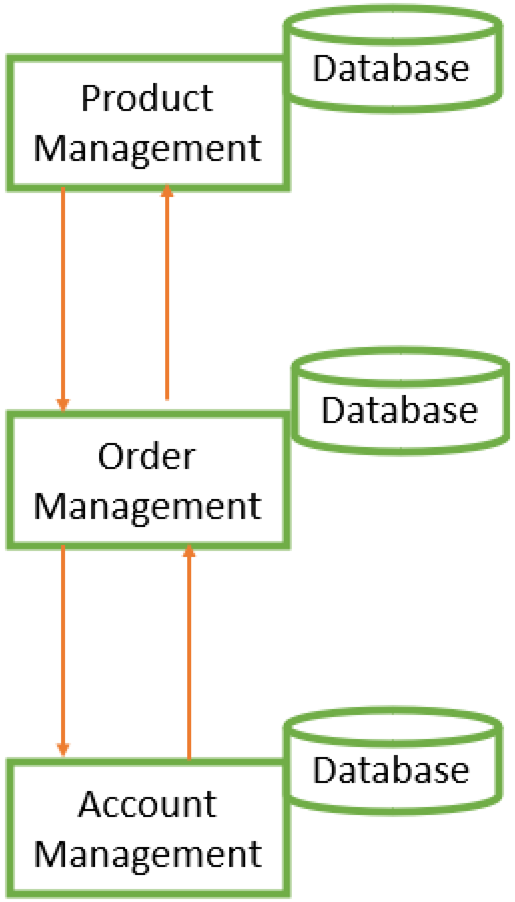


What are Design Principles??

- Independent/ Autonomous
- Resilient/ Fault Tolerant/ Design For Failure
- Observable
- Discoverable
- Domain Driven
- Decentralization
- High Cohesion
- Single Source Of Truth

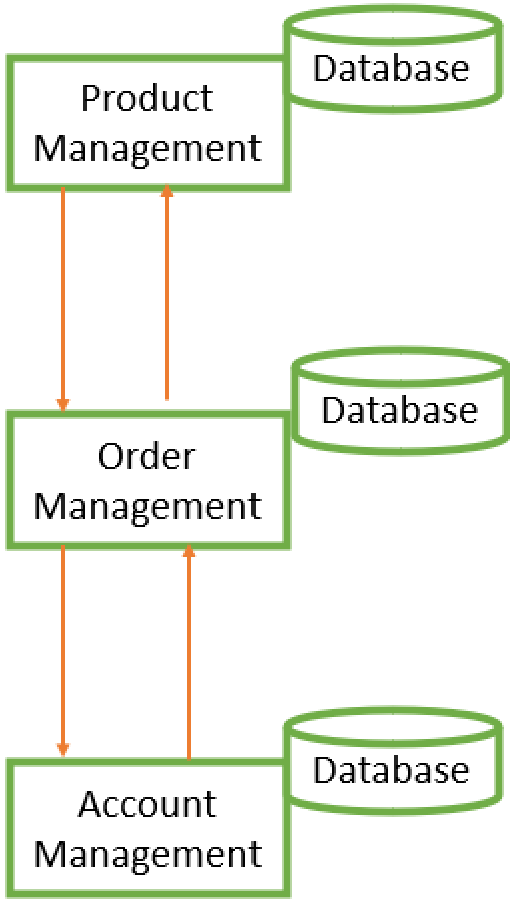
Independent/ Autonomous

- Small team size
- Parallel development
- Clear contracts
-



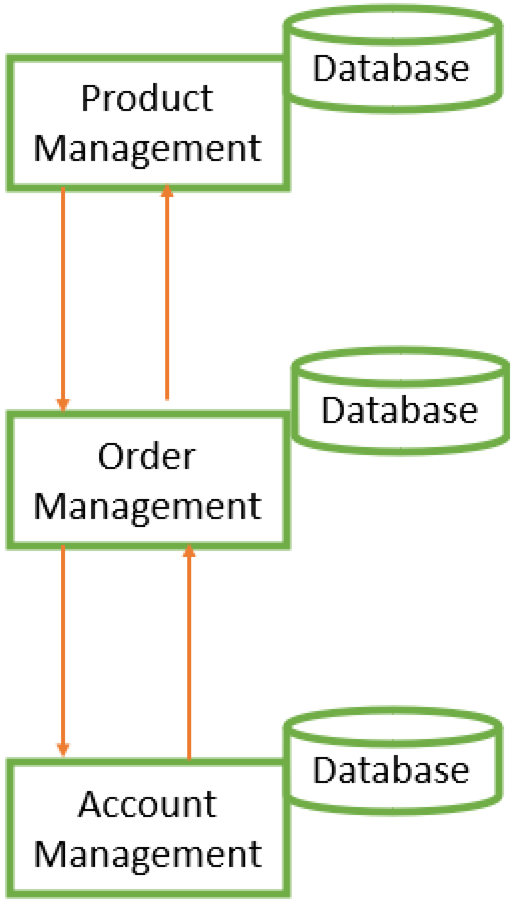
Resilient / Fault Tolerant / Design For Failure

- Avoid single point of failure
- Avoid cascading failure
-



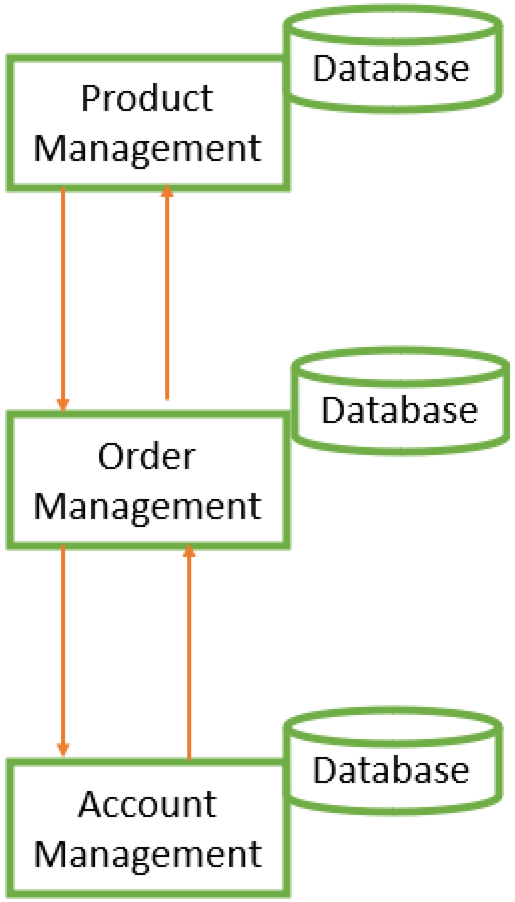
Observable

- Centralized monitoring
- Centralized logging
-



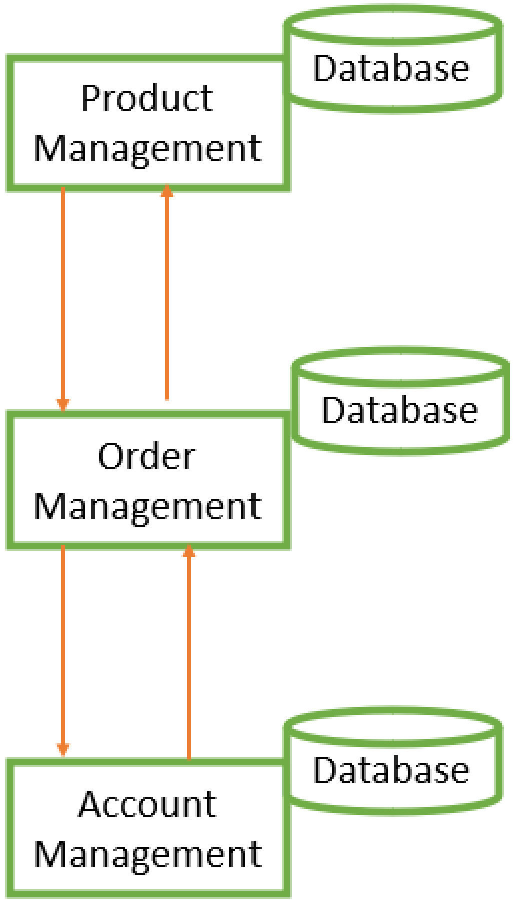
Discoverable

- All services should be registered at one place
- It makes client’s life easy when looking for specific service



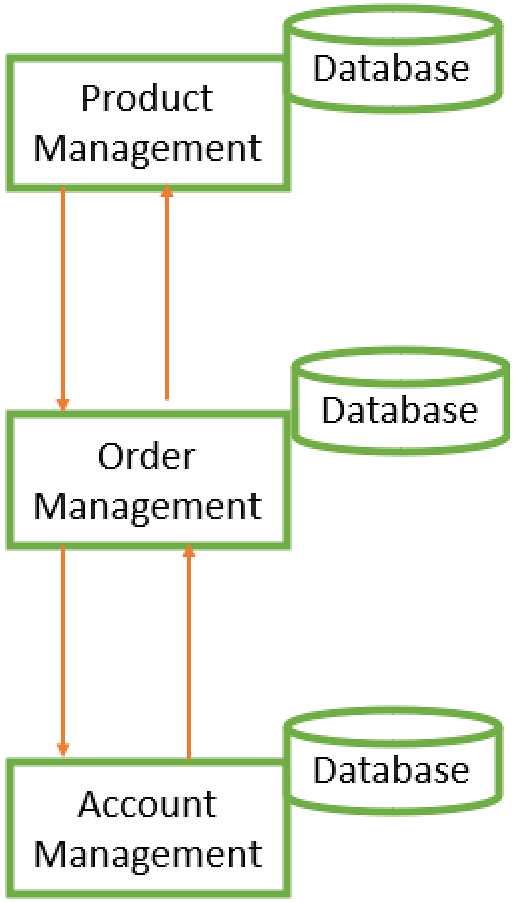
Domain Driven

- Focussed on business
- Focussed on core domain
-



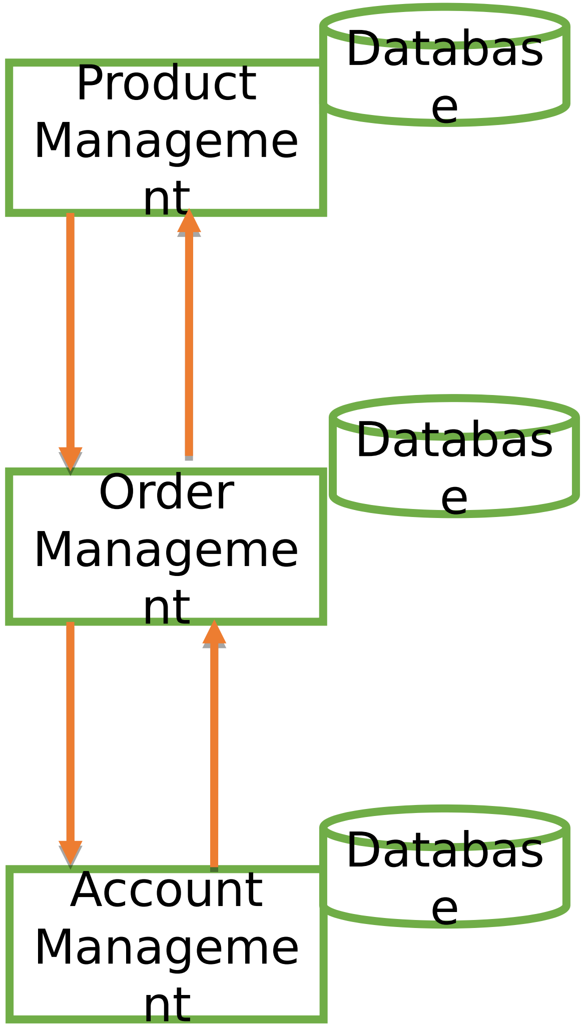
Decentralization

- Database for each service
- Choice of database depends on the nature of particular service



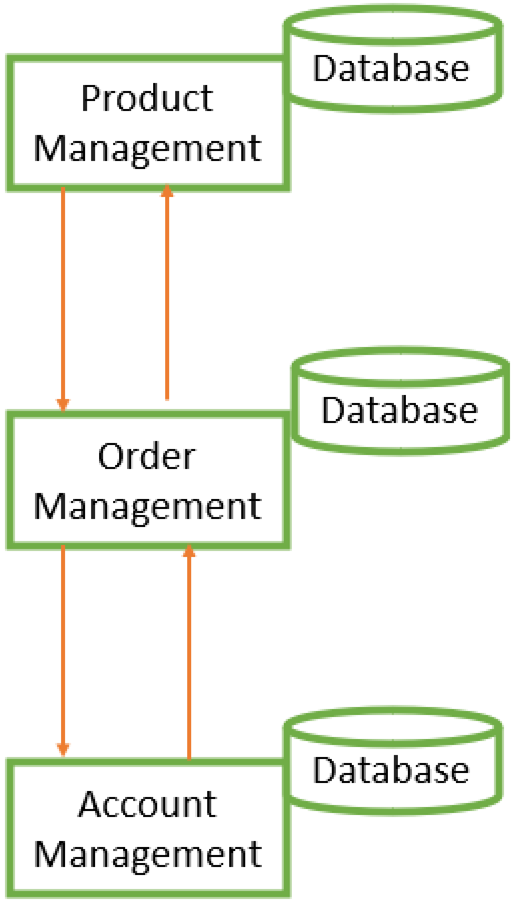
High Cohesion

- Do one thing only
- SRP
- A business function
- A business domain
- Easy to take new similar feature
- Why
  - Scalability
  - Availability



Single Source Of Truth

- There should be only one source to get the complete information
- This helps in avoiding the duplicity



What next??

Microservices : Design Patterns

Microservices: Design Patterns

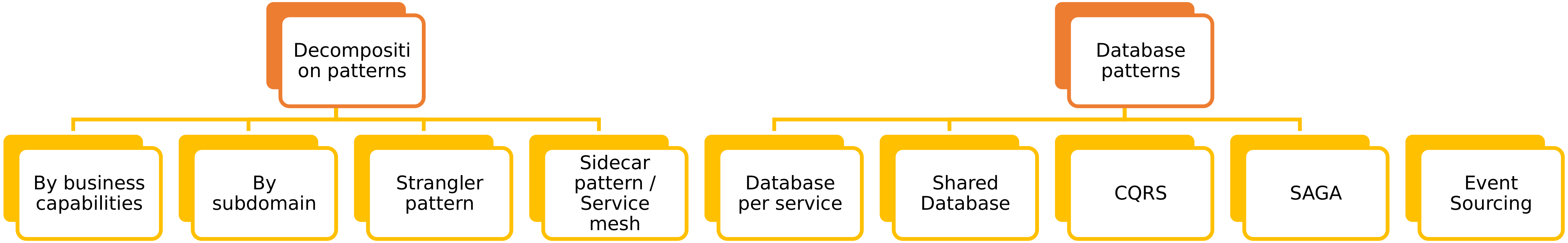
Microservices Architecture

Why??

- We need our services to be highly
  - **Available**
  - **Scalable**
  - **Resilient to failures**
  - **Efficient**
- Design patterns help in solving the specific microservice architecture challenge
-

What are those patterns??

- Decomposition
- Database
- Communication Among services
- Integration
- Deployment
- Observability
- Cross-cutting concern

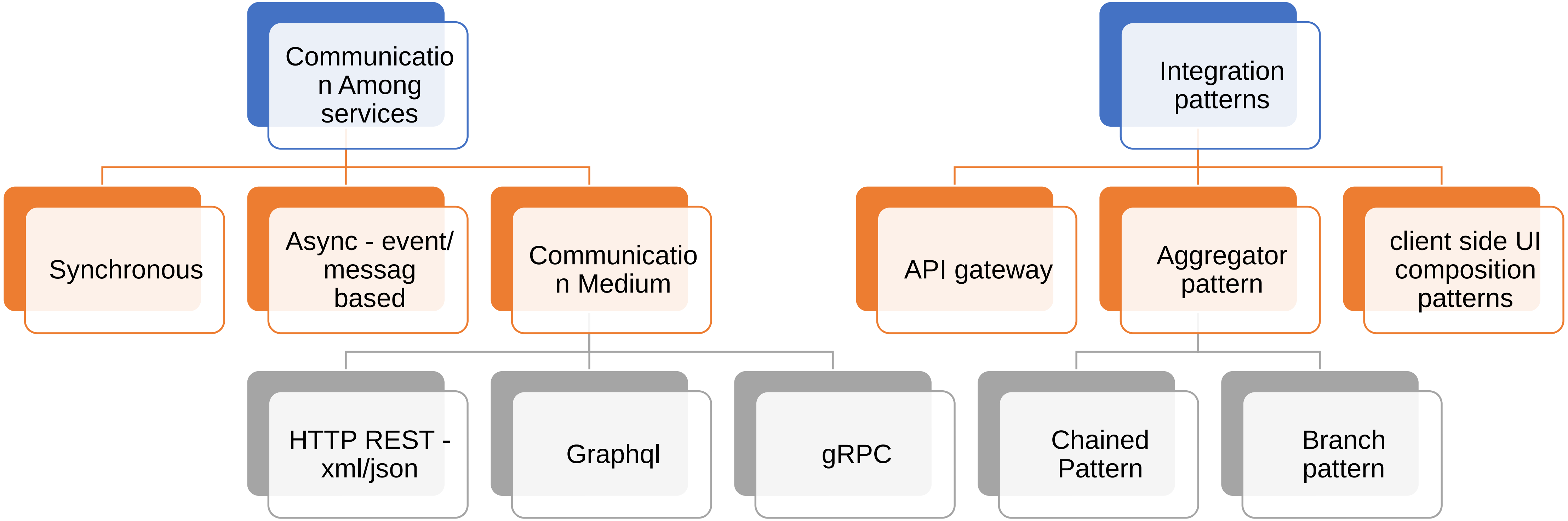


Decomposition patterns

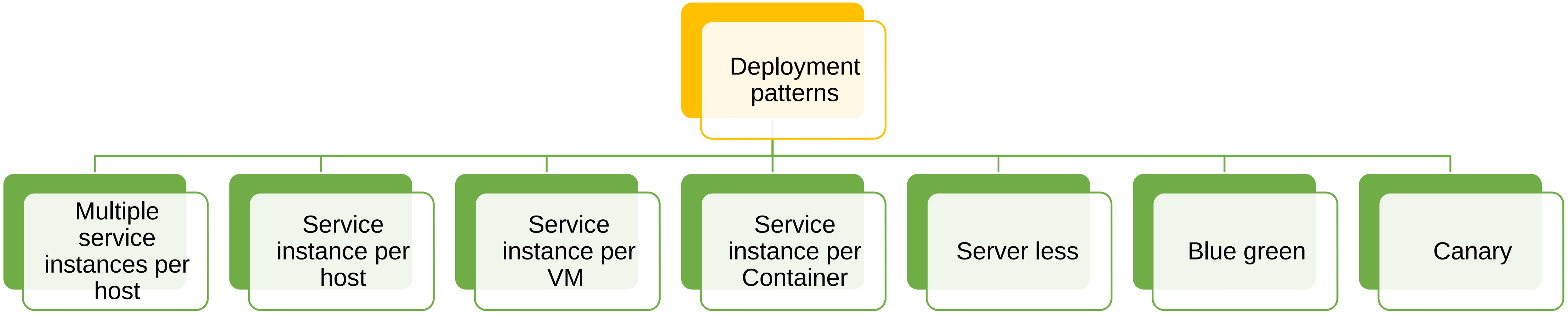
- By business capabilities
- By subdomain
- Strangler pattern
- Sidecar pattern / Service mesh

Database patterns

- Database per service
- Shared Database
- CQRS
- SAGA
- Event Sourcing



- Communication Among services
  - Synchronous
  - Async - event/ messag based
  - Communication Medium
    - HTTP REST - xml/json
    - Graphql
    - gRPC
- Integration patterns
  - API gateway
  - Aggregator pattern
    - Chained Pattern
    - Branch pattern
  - client side UI composition patterns



Deployment patterns

Multiple service instances per host

Service instance per host

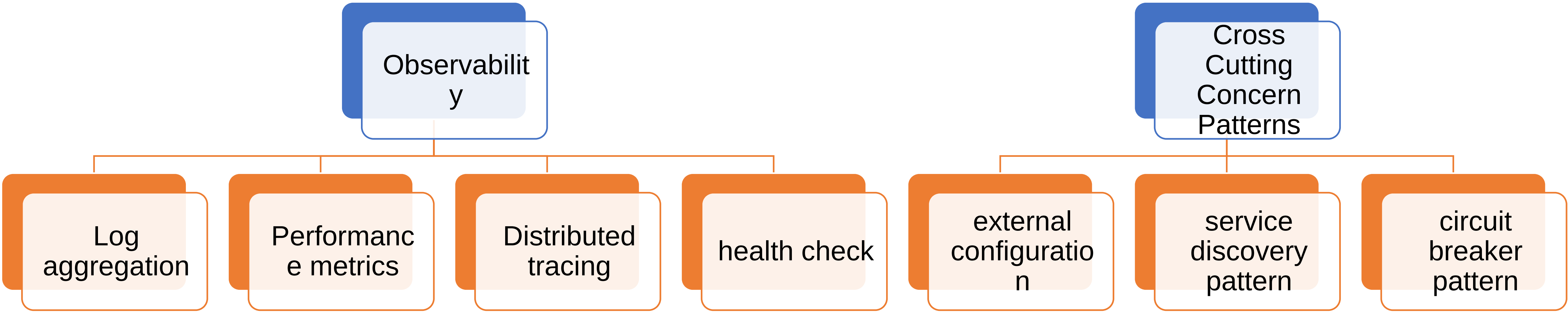
Service instance per VM

Service instance per Container

Server less

Blue green

Canary


Observability
Log aggregation
Performance metrics
Distributed tracing
health check
Cross Cutting Concern Patterns
external configuration
service discovery pattern
circuit breaker pattern

What next??

Decomposition Pattern : By Business Domain and subdomain

Decomposition Pattern : By Business Domain & Sub Domain

Microservices Architecture

There are 2 kinds of project under microservices

Monolithic to Microservices – Brown Field Projects

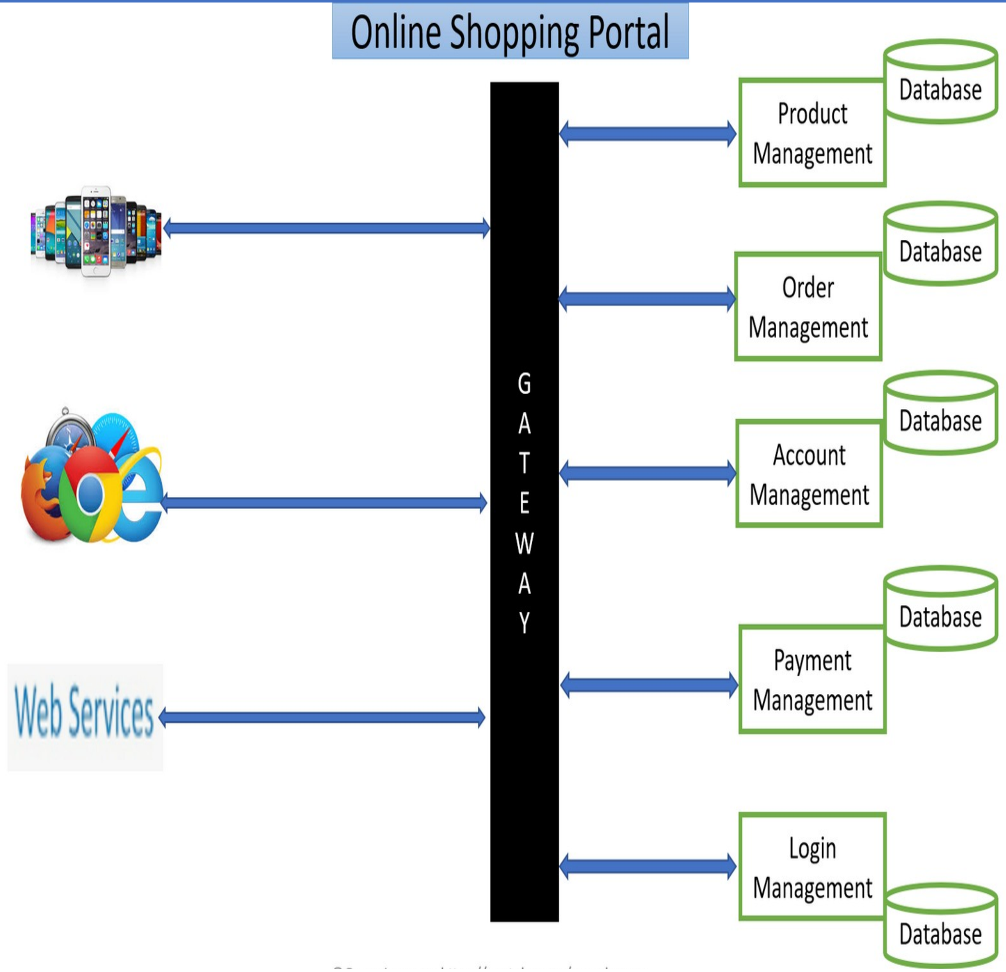
Microservices in nature from scratch – Green Field Projects

Microservice – Micro + Service

How Micro? How Small?

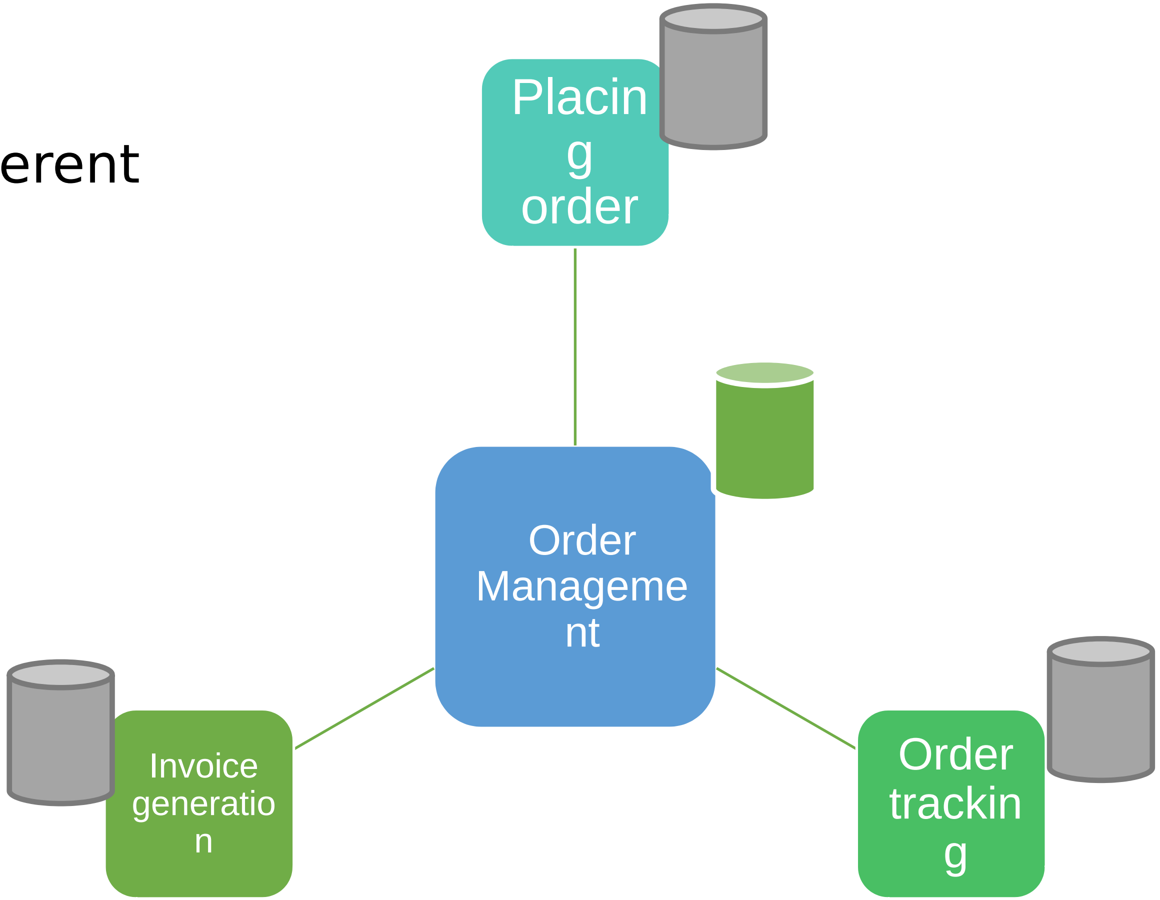
How to decide the size of microservice??

- Business functiona



How to decide the size of microservice??

- God Classes
  - Shared among different modules
- Sub domain
  - Domain Driven Design
  -



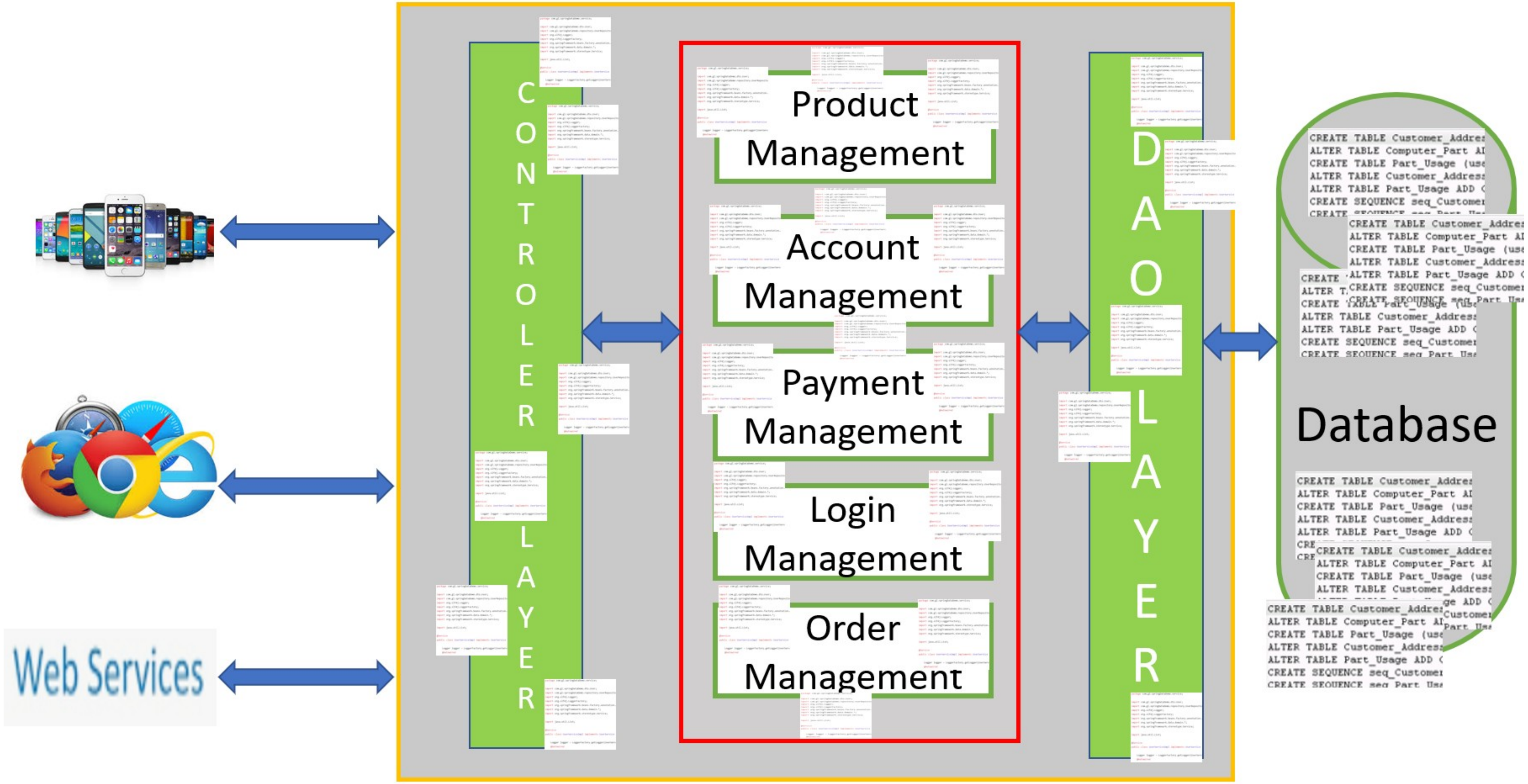
What next??

Decomposition Pattern : Strangler

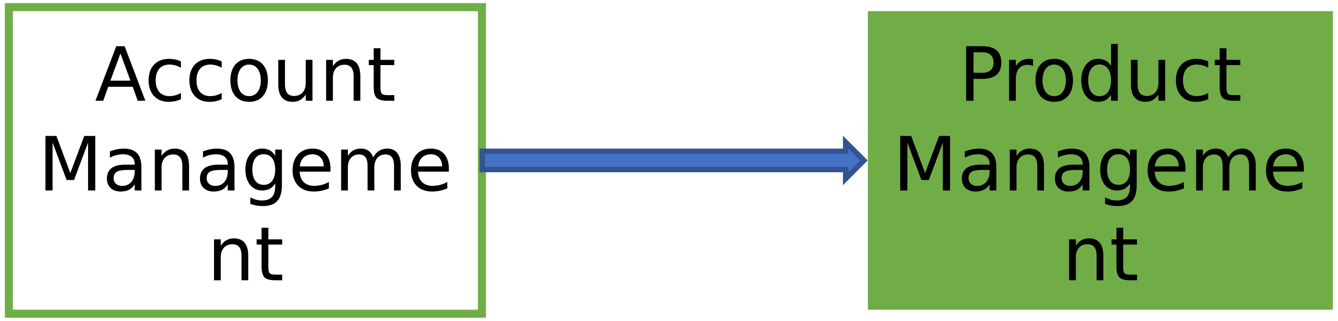
Decomposition Pattern : Strangler

Microservices Architecture

Online Shopping Portal

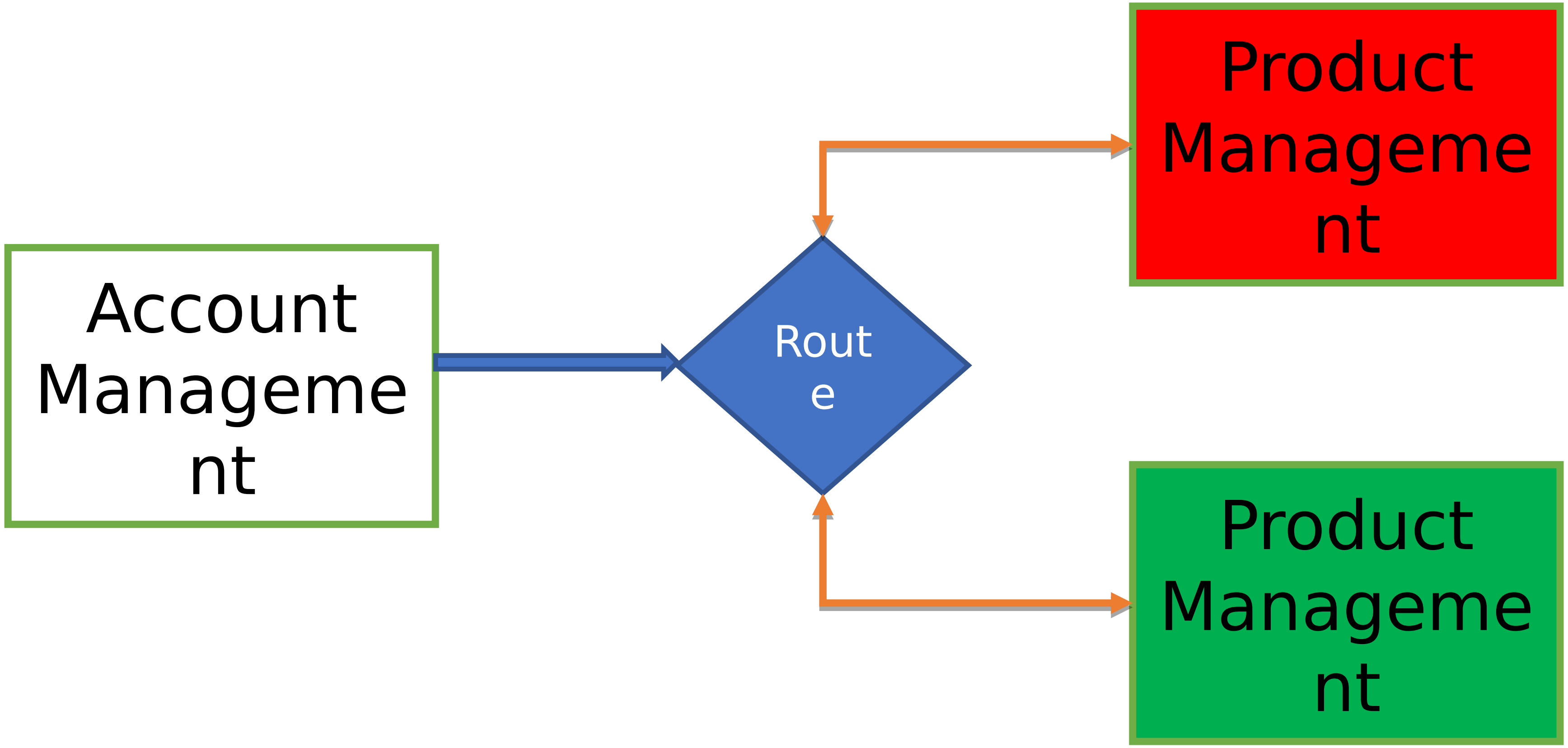

Web Services
CONTROLER LAYER
Product Management
Account Management
Payment Management
Login Management
Order Management
DAO LAYER
Database
CREATE TABLE Customer_Addres
ALTER TABLE Computer_Part AI
CREATE TABLE Part_Usage (use
ALTER TABLE Customer_Address
ALTER TABLE Part_Usage ADD C
CREATE SEQUENCE seq_Customer
CREATE SEQUENCE seq_Part_Usa

©Green Learner - https://youtube.com/greenlearner



Account Management

Product Management

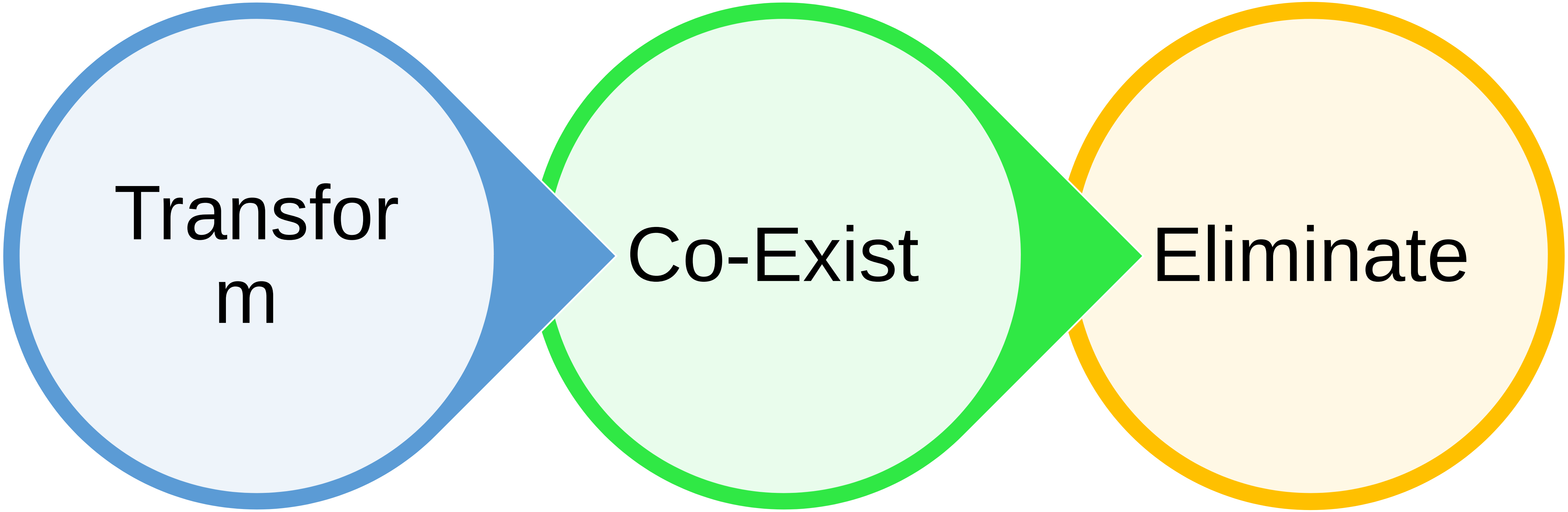


Account Management

Route

Product Management

Product Management

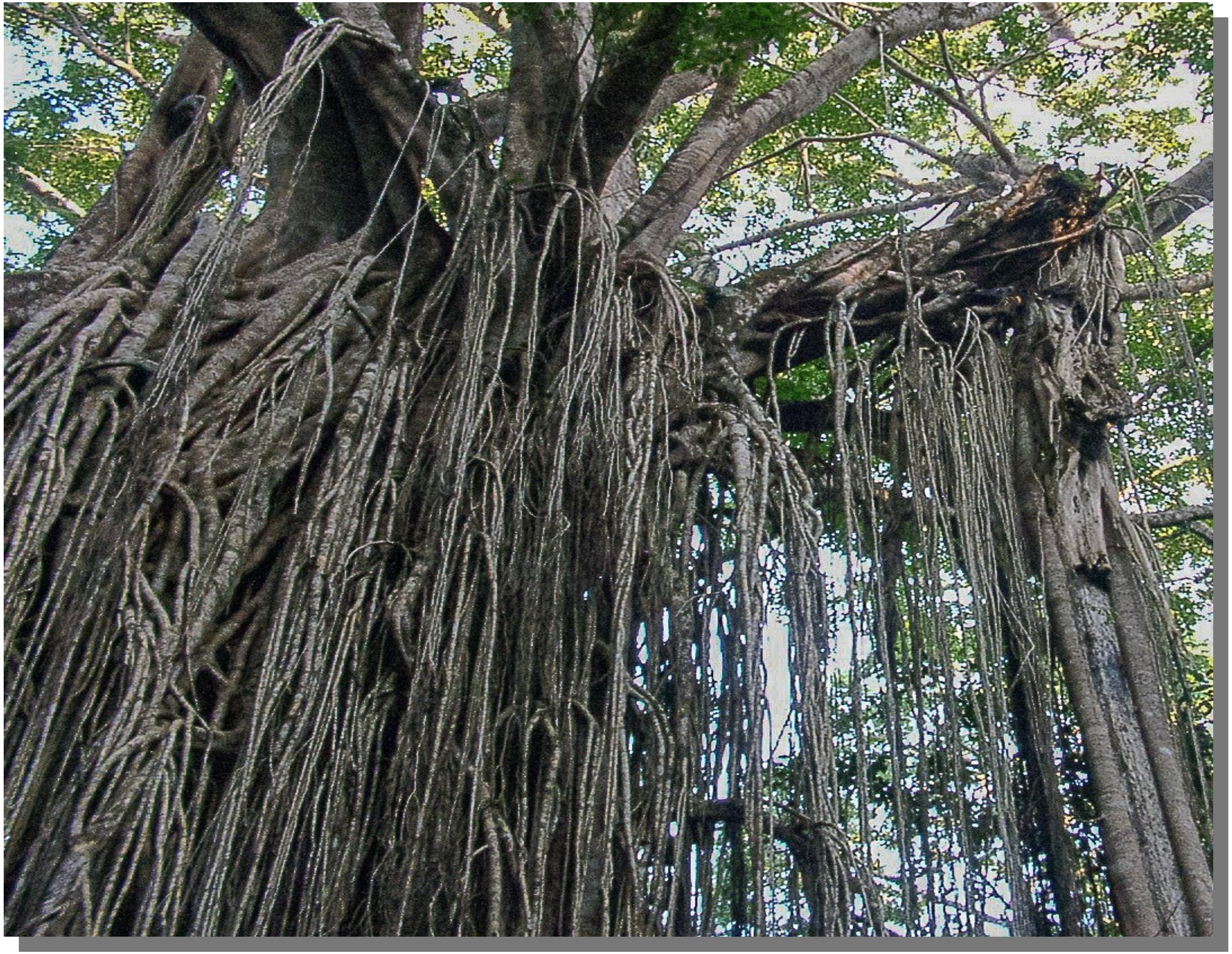


Transfor m

Co-Exist

Eliminate

Strangler



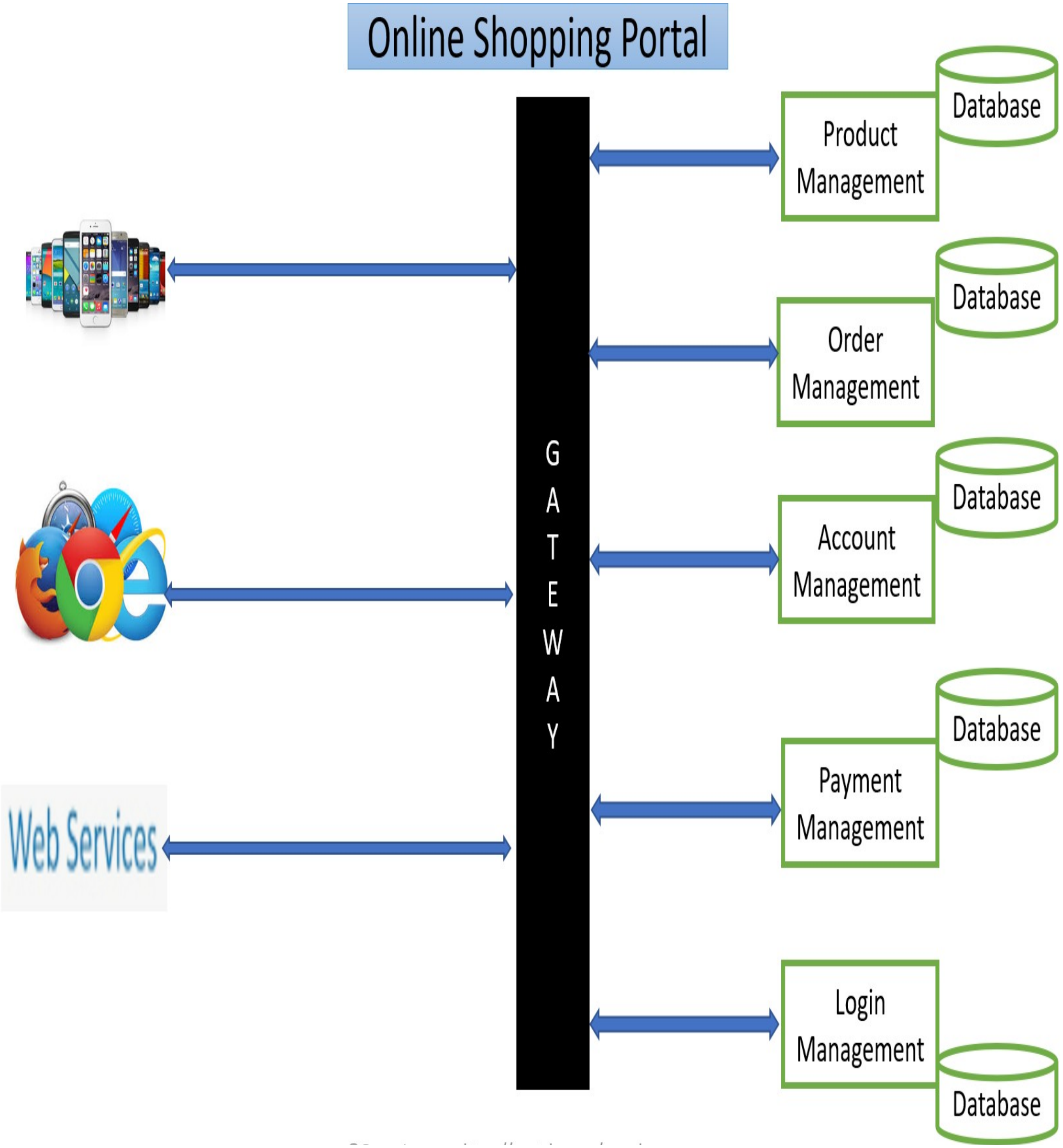
What next??

Decomposition Pattern : Sidecar/ Service-Mesh

MICROSERVICES DESIGN PATTERNS

Decomposition Pattern : Sidecar/Sidekick Pattern

Microservices Architecture



Online Shopping Portal

Web Services

GATEWAY

Product Management

Order Management

Account Management

Payment Management

Login Management

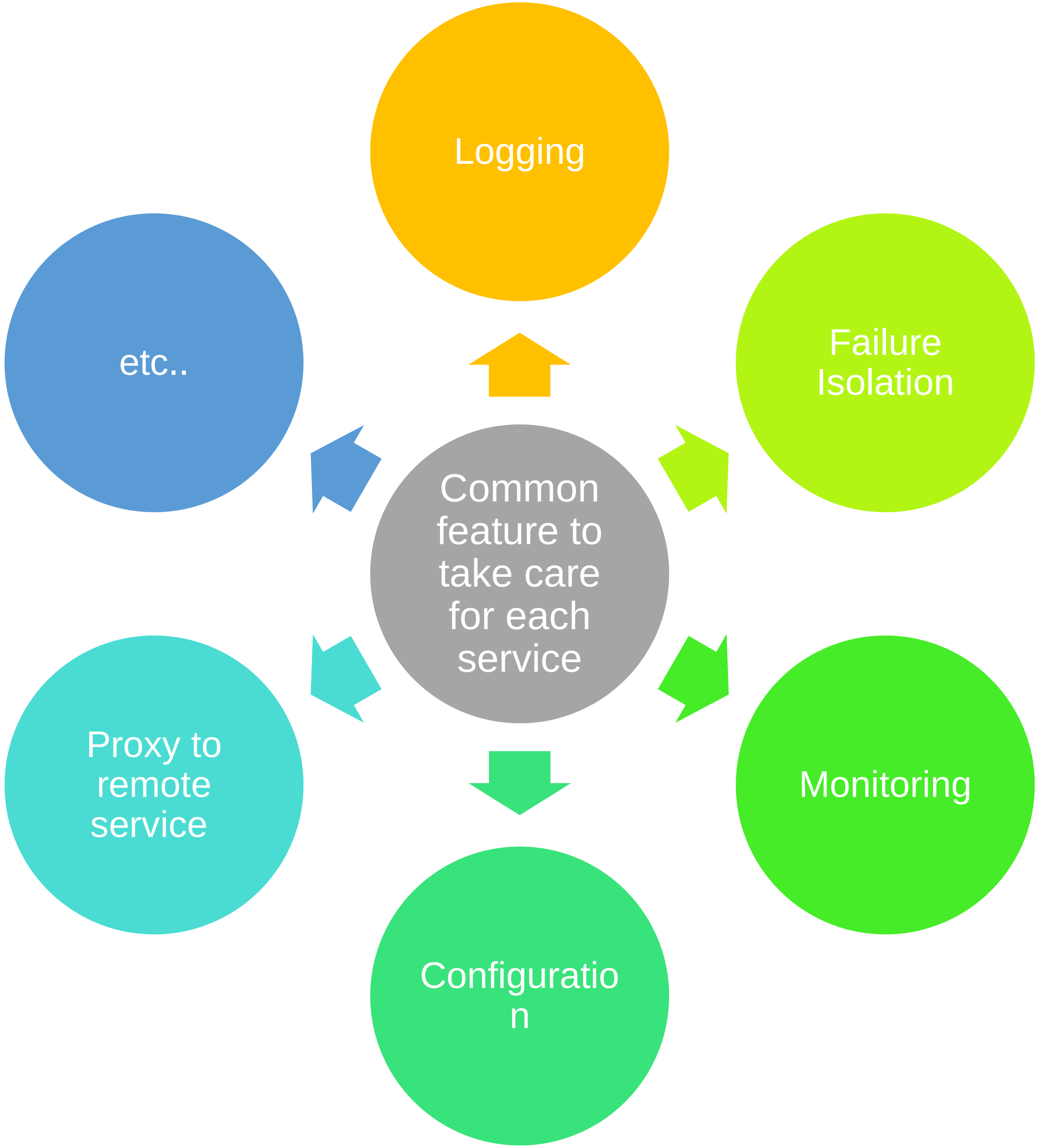
Database

Database

Database

Database

Database



Common feature to take care for each service

Logging

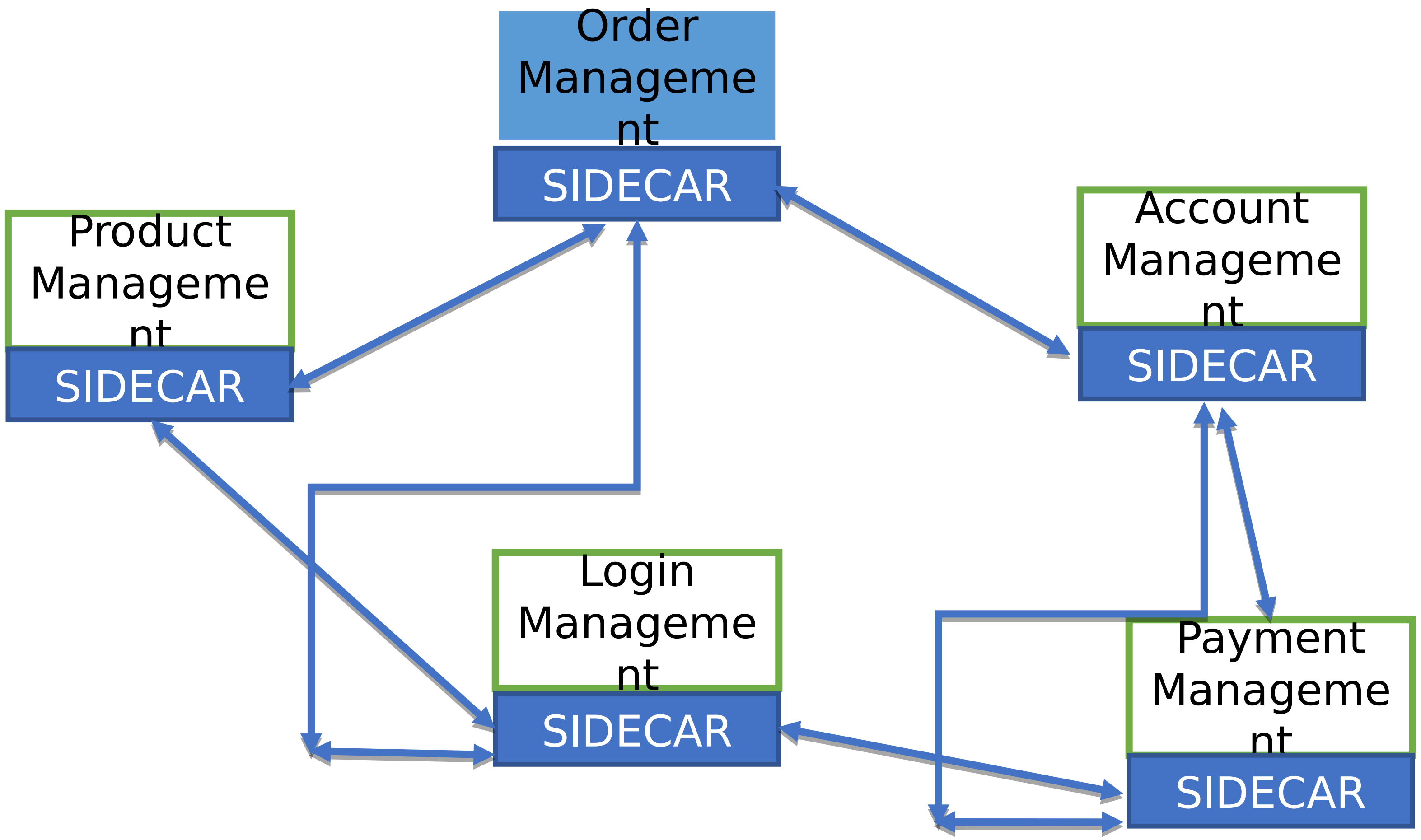
Failure Isolation

Monitoring

Configuratio n

Proxy to remote service

etc..



Order Management

SIDECAR

Product Management

SIDECAR

Account Management

SIDECAR

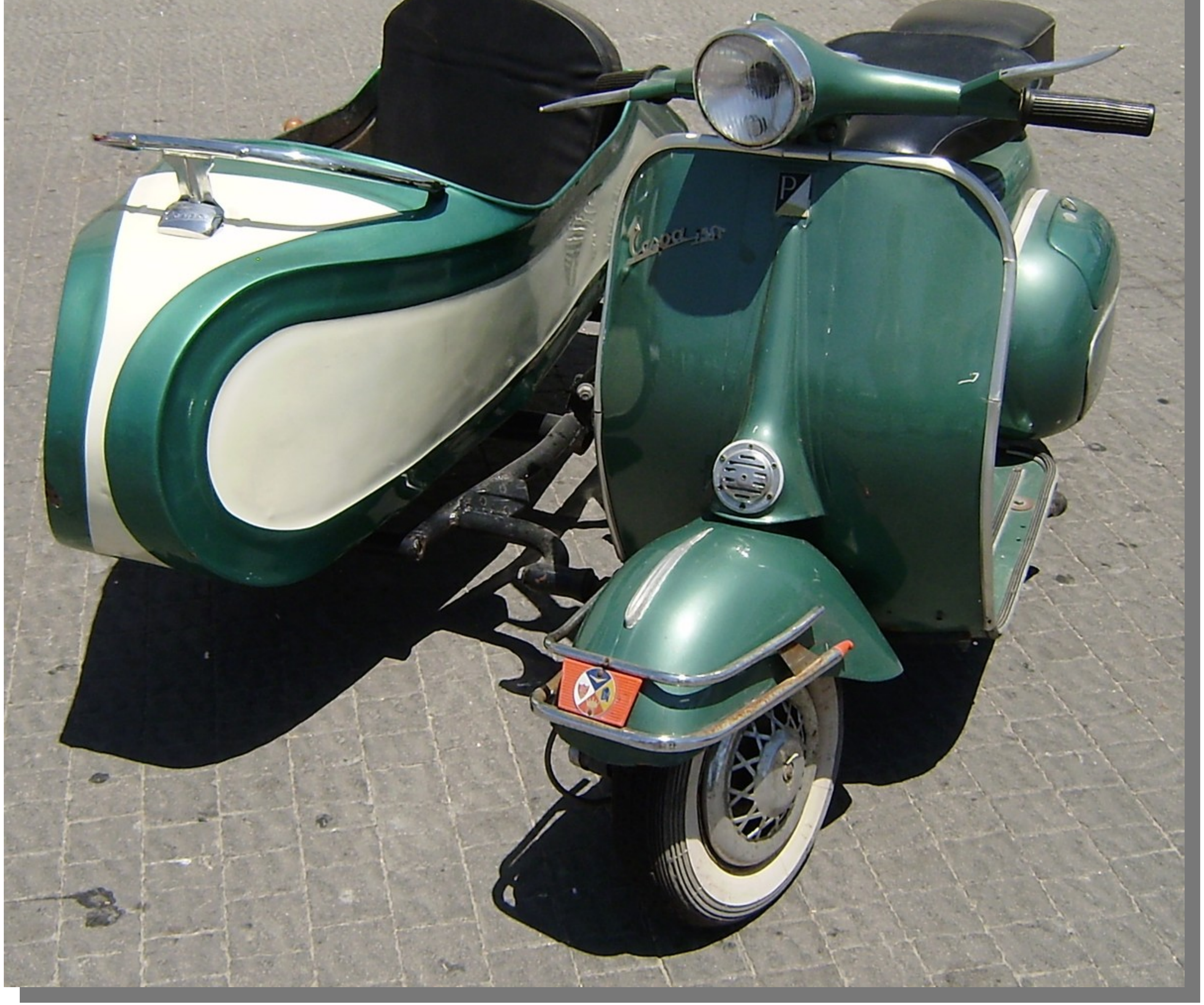
Login Management

SIDECAR

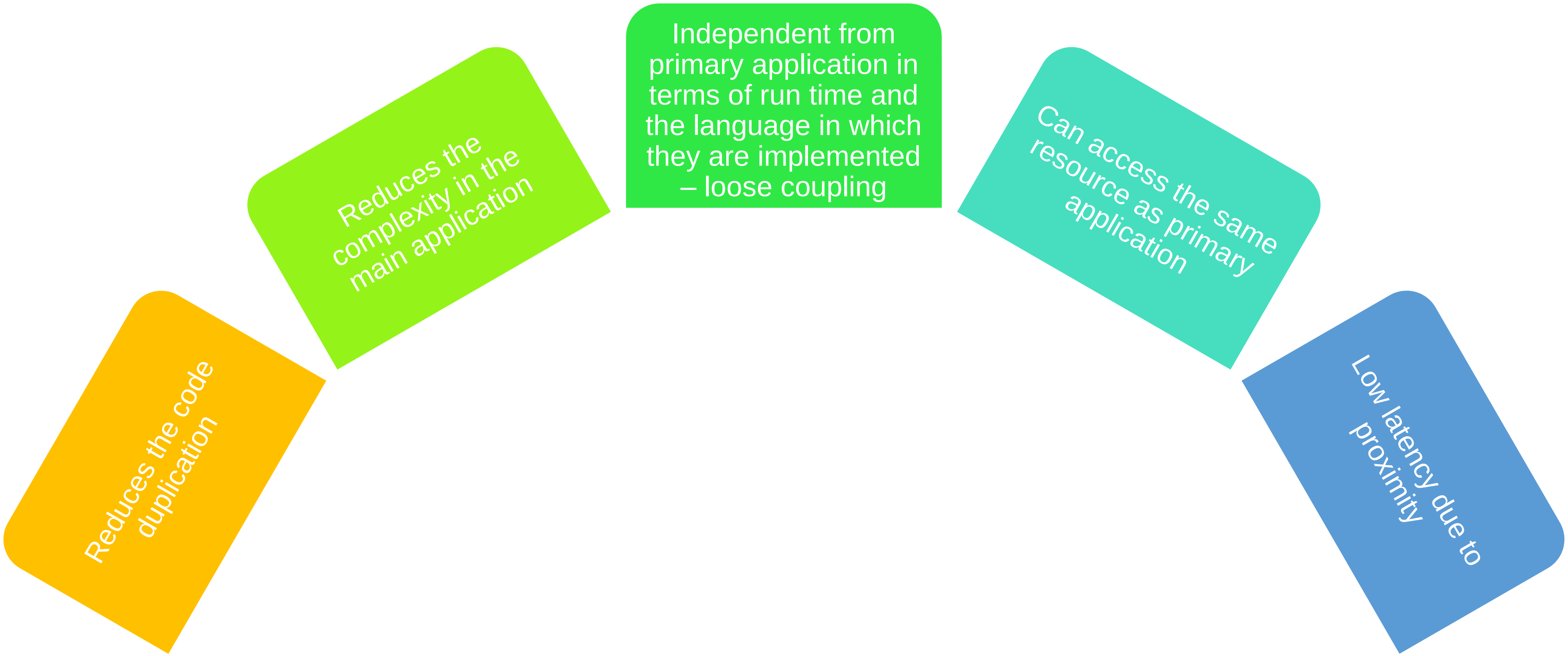
Payment Management

SIDECAR

Sidecar



Advantages



Reduces the code duplication

Reduces the complexity in the main application

Independent from primary application in terms of run time and the language in which they are implemented – loose coupling

Can access the same resource as primary application

Low latency due to proximity

Issues & Concerns

- Try to use language- or framework-agnostic technologies

- Before putting functionality into a sidecar, consider whether it would work
  - better as a separate service or a more traditional daemon.
  - the functionality could be implemented as a library.

- Containers are particularly well suited to the sidecar pattern.

What next??

Decomposition Pattern : Service-Mesh

MICROSERVICES DESIGN PATTERNS

Decomposition Pattern : Service Mesh

Microservices Architecture

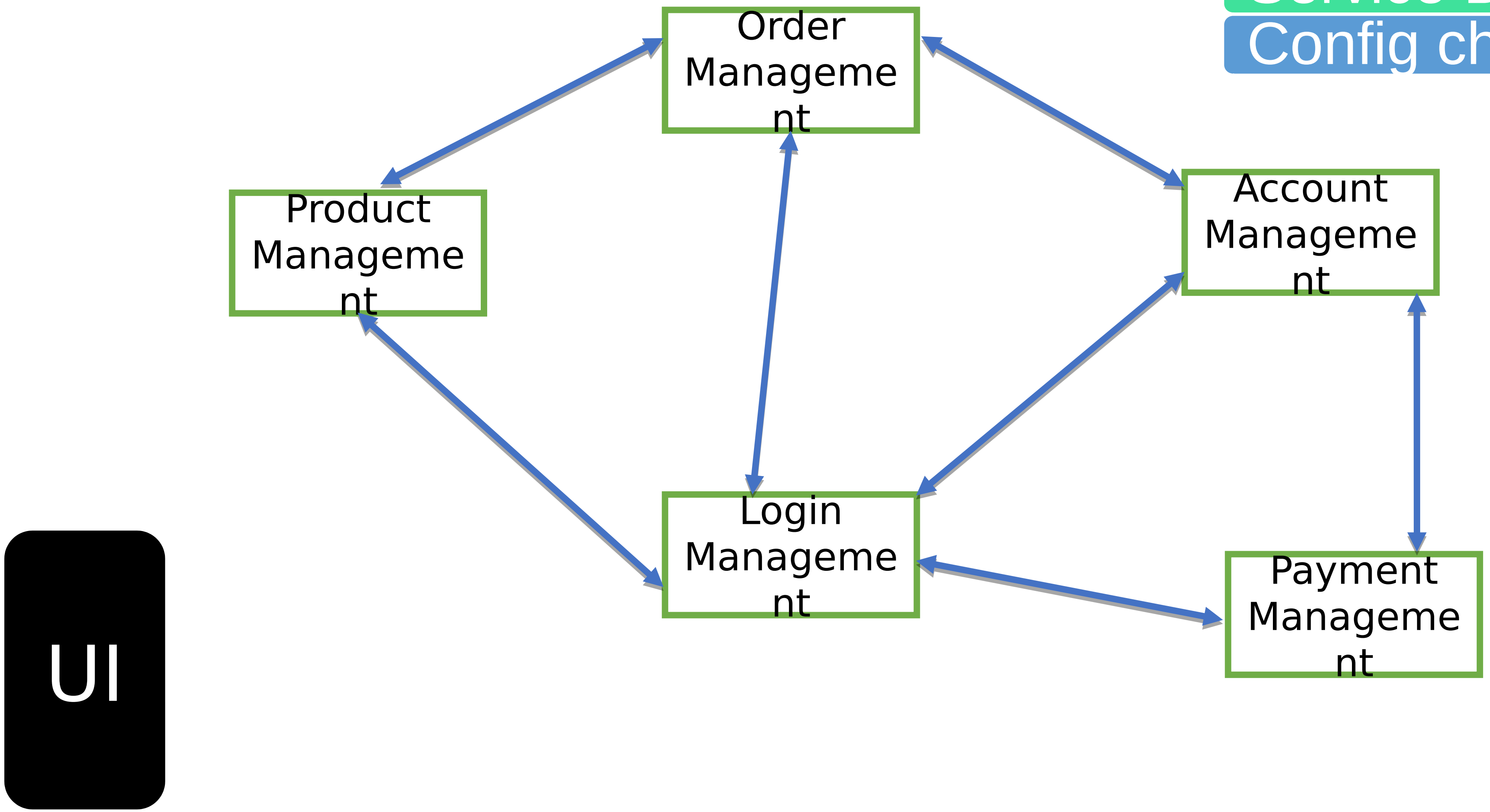
Use Case (Problem Statement)

Communication Complexity

Failure Isolation

Service Discovery

Config changes



What is service mesh?

It’s dedicated infrastructure layer for service-to-service communication

- focusing on managing all service-to-service communication within a distributed software system.
- This makes communication optimization easier

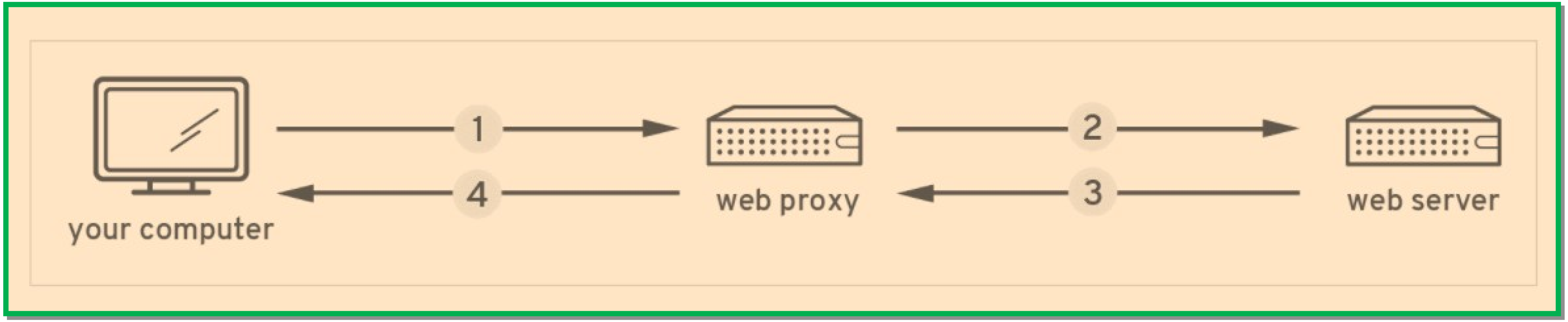
It’s a way to control how different parts of an application share the data among themselves

Typically array of network proxies

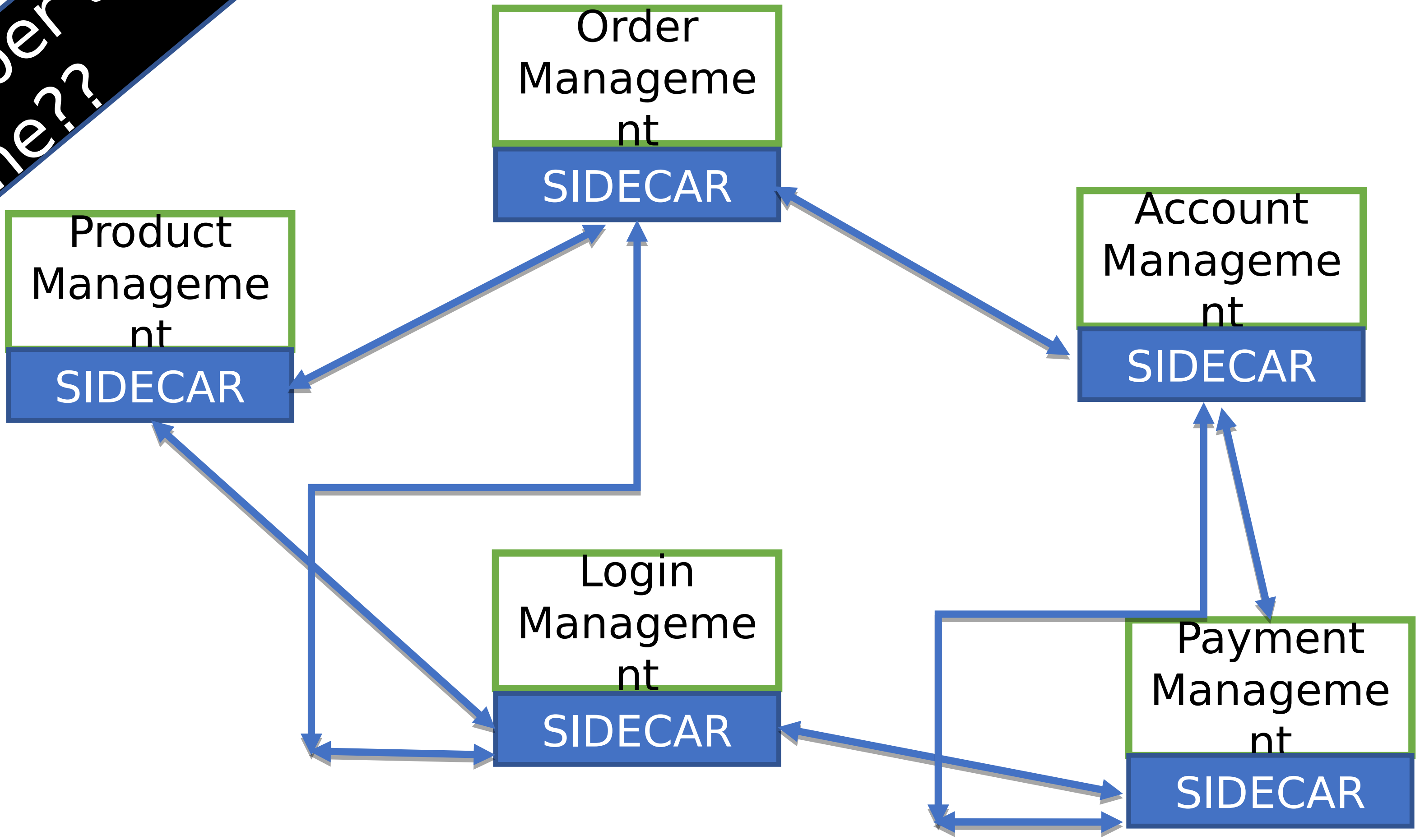
- Deployed alongside main service
- Main service need not to be aware of this proxy

For cloud native application

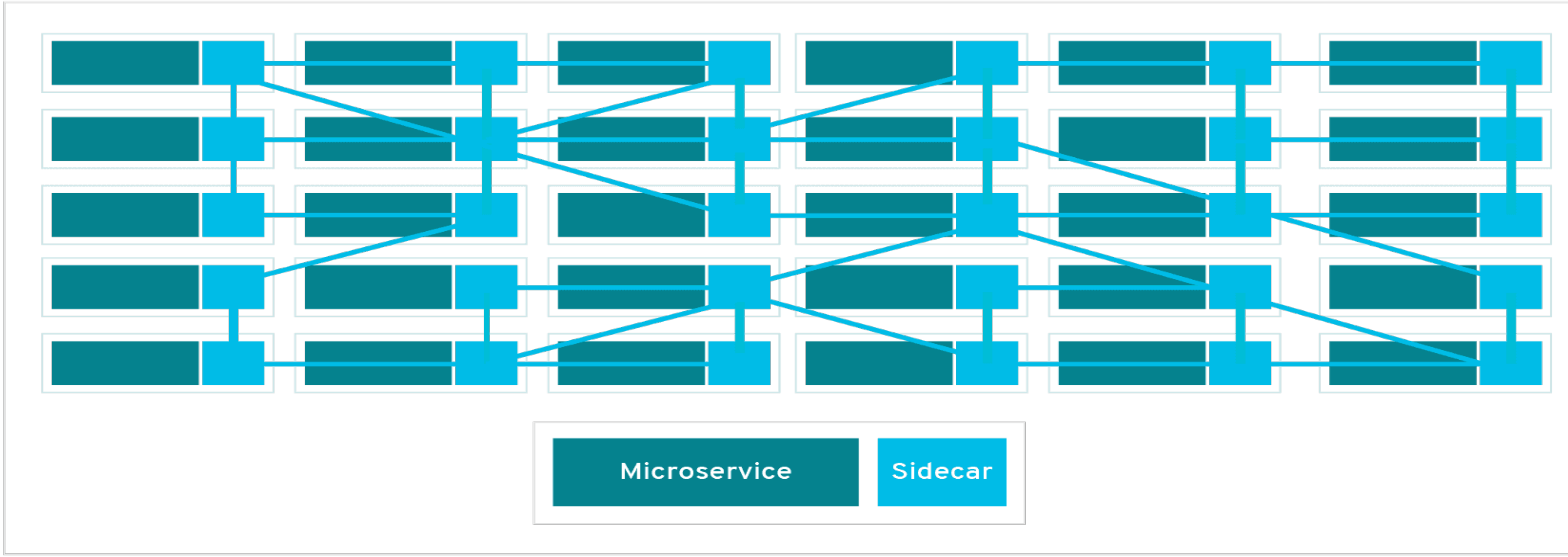
How does it work??



Remember this one??

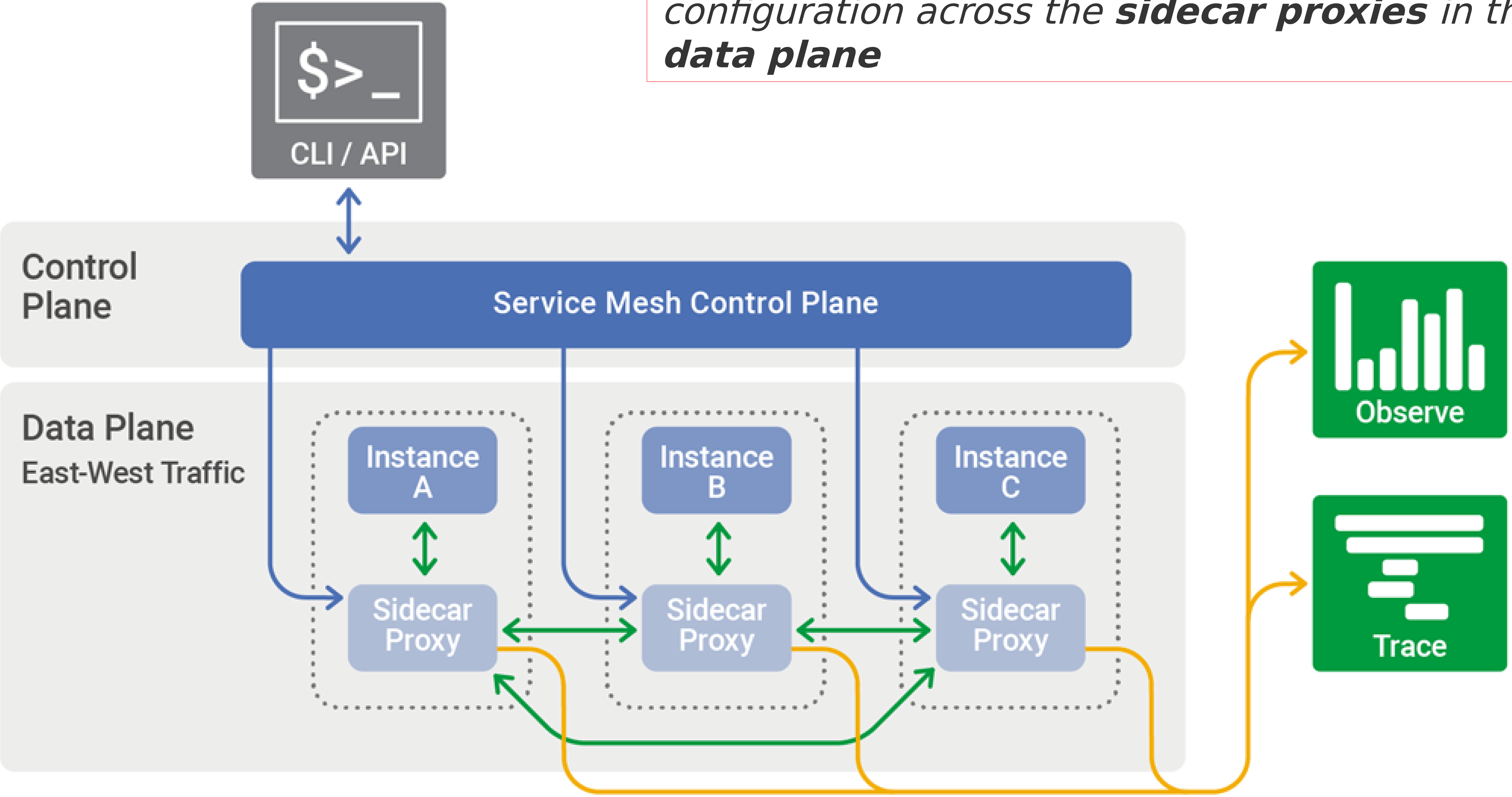


Now it’s like real Mesh



https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh

*The **control plane** in a service mesh distributes configuration across the **sidecar proxies** in the **data plane***



https://www.nginx.com/blog/what-is-a-service-mesh/

Role of Service Mesh

- Service Discovery
- Fault Tolerance
- Routing
- Observability
  - Logging
  - Monitoring
- Security
- Access Control
- Deployment

Pros & Cons

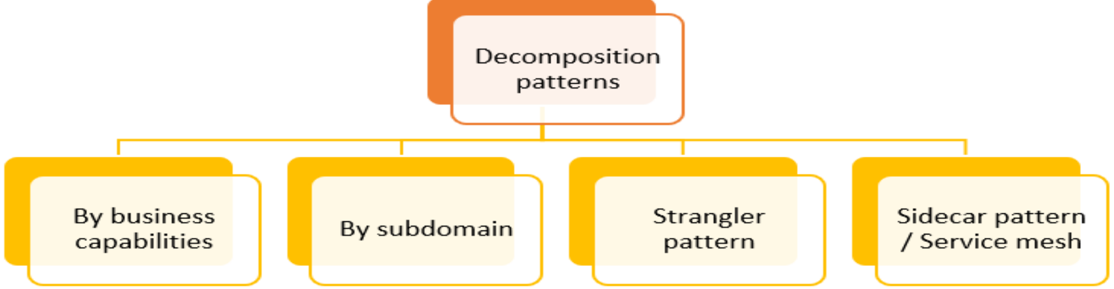
- Pros
  - Centralized solution for logging, distributed tracing, logging security, access control
    - All of these are reusable for different microservices
  - Language agnostic for microservices
- Cons
  - Complexity
  - Extra network hop
  - New and Immature
- FAQ - https://www.infoq.com/articles/service-mesh-ultimate-guide/

How to Implement Service Mesh

- https://linkerd.io/
- https://istio.io/
- https://www.envoyproxy.io/
- https://www.consul.io/

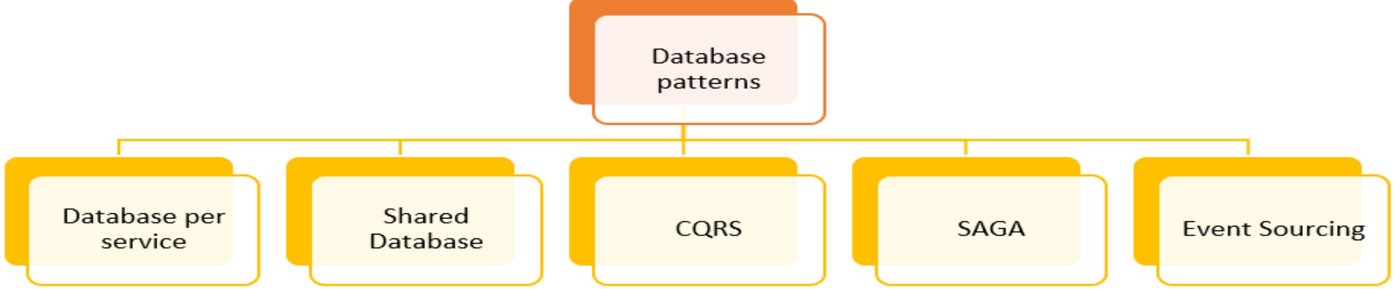
What next??

Decomposition Pattern : Summary



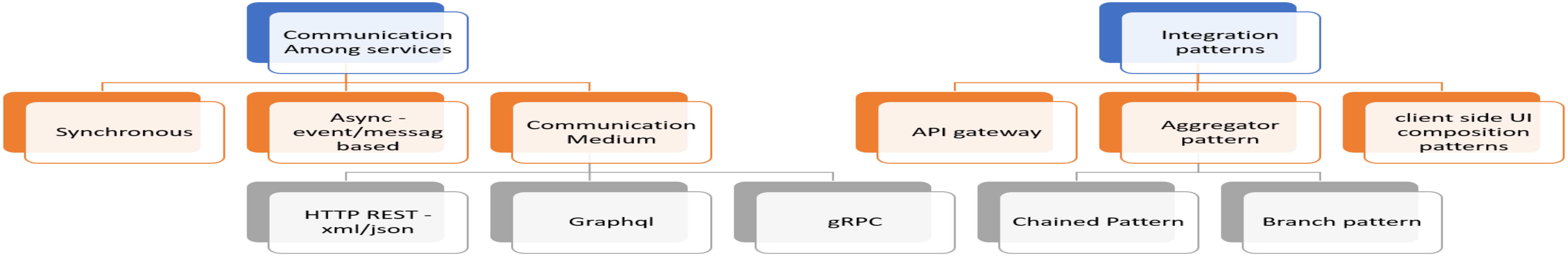
Decomposition patterns

- By business capabilities
- By subdomain
- Strangler pattern
- Sidecar pattern / Service mesh



Database patterns

- Database per service
- Shared Database
- CQRS
- SAGA
- Event Sourcing

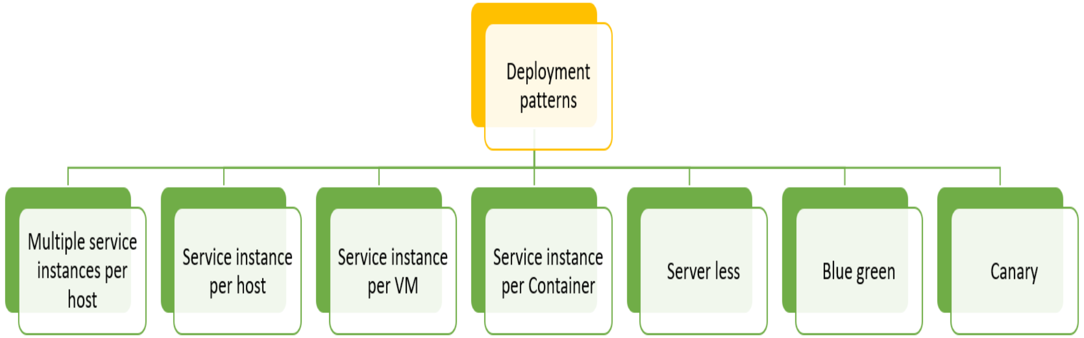


Communication Among services

- Synchronous
- Async - event/messag based
- Communication Medium
  - HTTP REST - xml/json
  - Graphql
  - gRPC

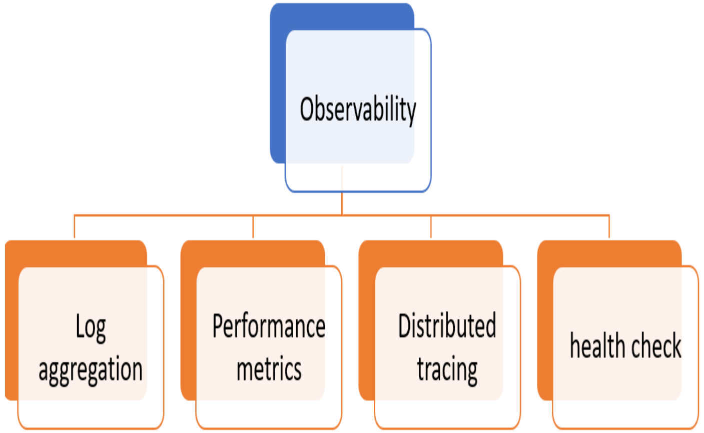
Integration patterns

- API gateway
- Aggregator pattern
  - Chained Pattern
  - Branch pattern
- client side UI composition patterns



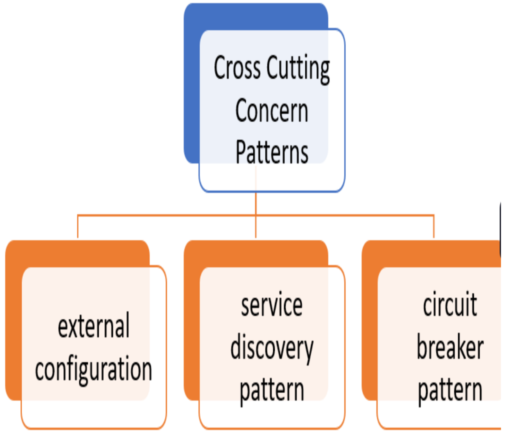
Deployment patterns

- Multiple service instances per host
- Service instance per host
- Service instance per VM
- Service instance per Container
- Server less
- Blue green
- Canary



Observability

- Log aggregation
- Performance metrics
- Distributed tracing
- health check



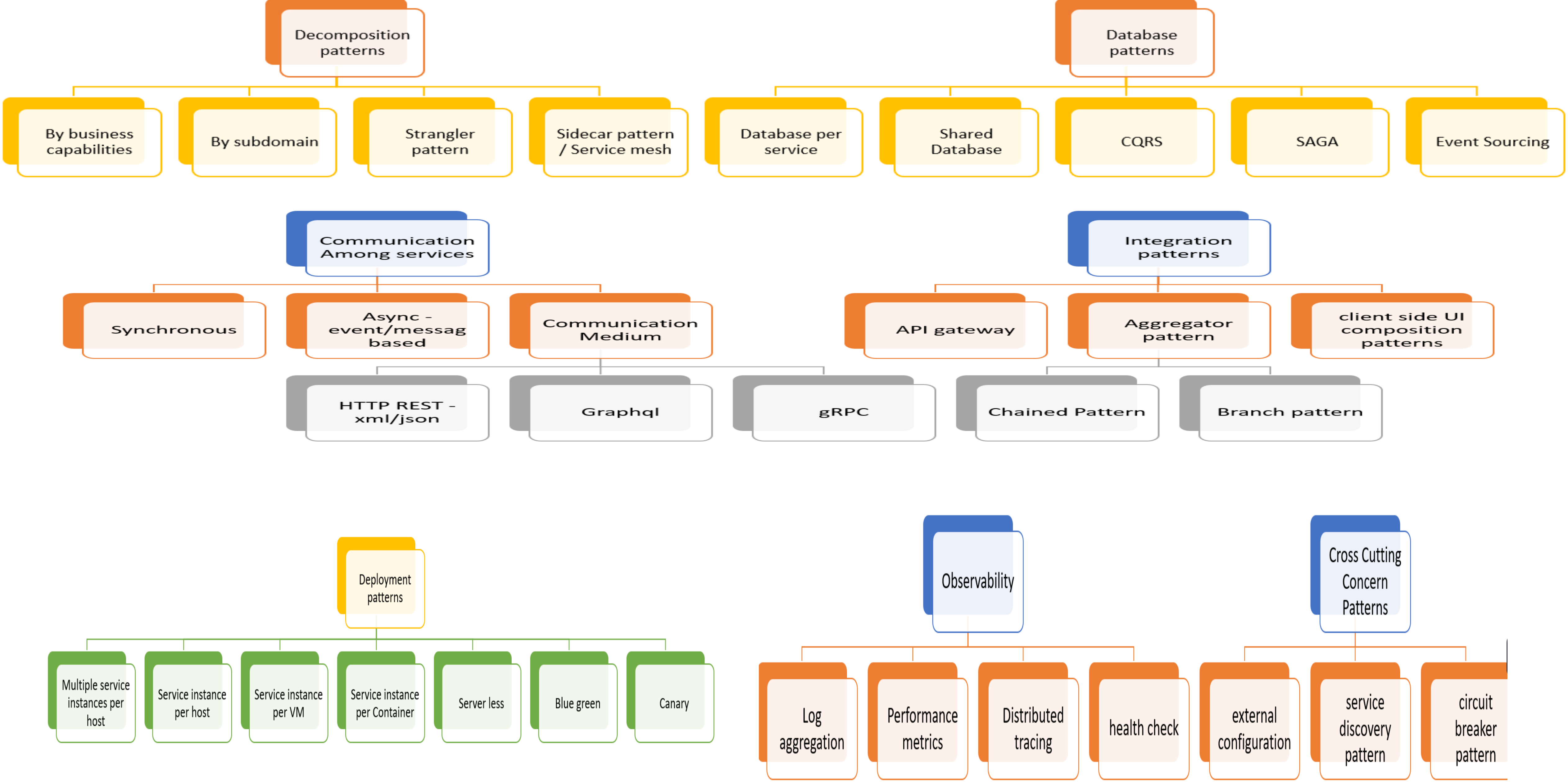
Cross Cutting Concern Patterns

- external configuration
- service discovery pattern
- circuit breaker pattern

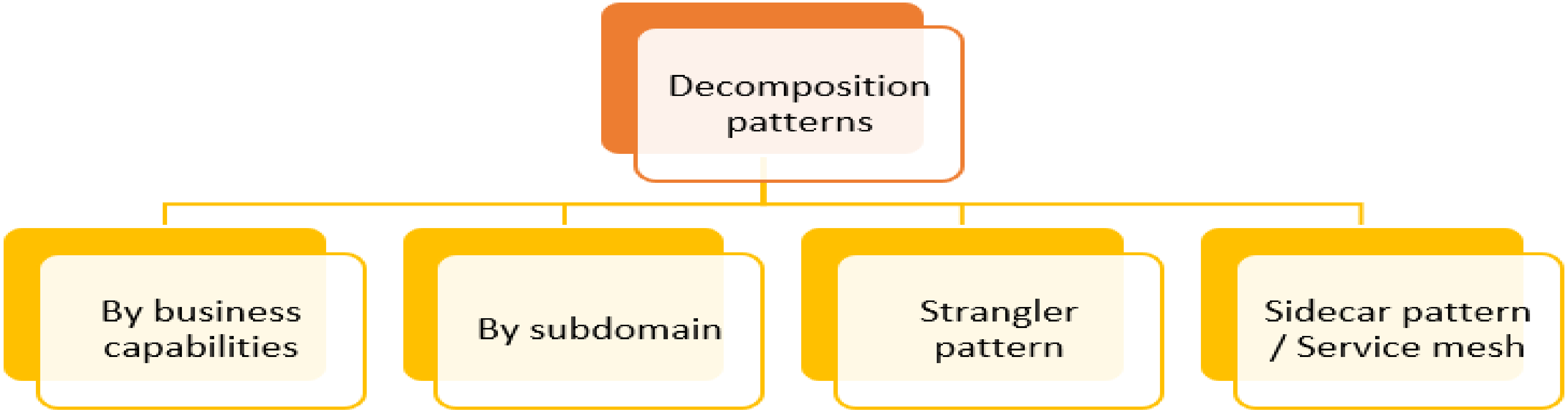
MICROSERVICES DESIGN PATTERNS

Decomposition Pattern : Summary

Microservices Architecture



- Decomposition patterns
  - By business capabilities
  - By subdomain
  - Strangler pattern
  - Sidecar pattern / Service mesh
- Database patterns
  - Database per service
  - Shared Database
  - CQRS
  - SAGA
  - Event Sourcing
- Communication Among services
  - Synchronous
  - Async - event/messag based
  - Communication Medium
    - HTTP REST - xml/json
    - Graphql
    - gRPC
- Integration patterns
  - API gateway
  - Aggregator pattern
    - Chained Pattern
    - Branch pattern
  - client side UI composition patterns
- Deployment patterns
  - Multiple service instances per host
  - Service instance per host
  - Service instance per VM
  - Service instance per Container
  - Server less
  - Blue green
  - Canary
- Observability
  - Log aggregation
  - Performance metrics
  - Distributed tracing
  - health check
- Cross Cutting Concern Patterns
  - external configuration
  - service discovery pattern
  - circuit breaker pattern



Decomposition patterns

By business capabilities

By subdomain

Strangler pattern

Sidecar pattern / Service mesh

/hat next??

Database Pattern : DB / Shared-DB per service

MICROSERVICES DESIGN PATTERNS

Database Patterns :

DB / Shared-DB per service

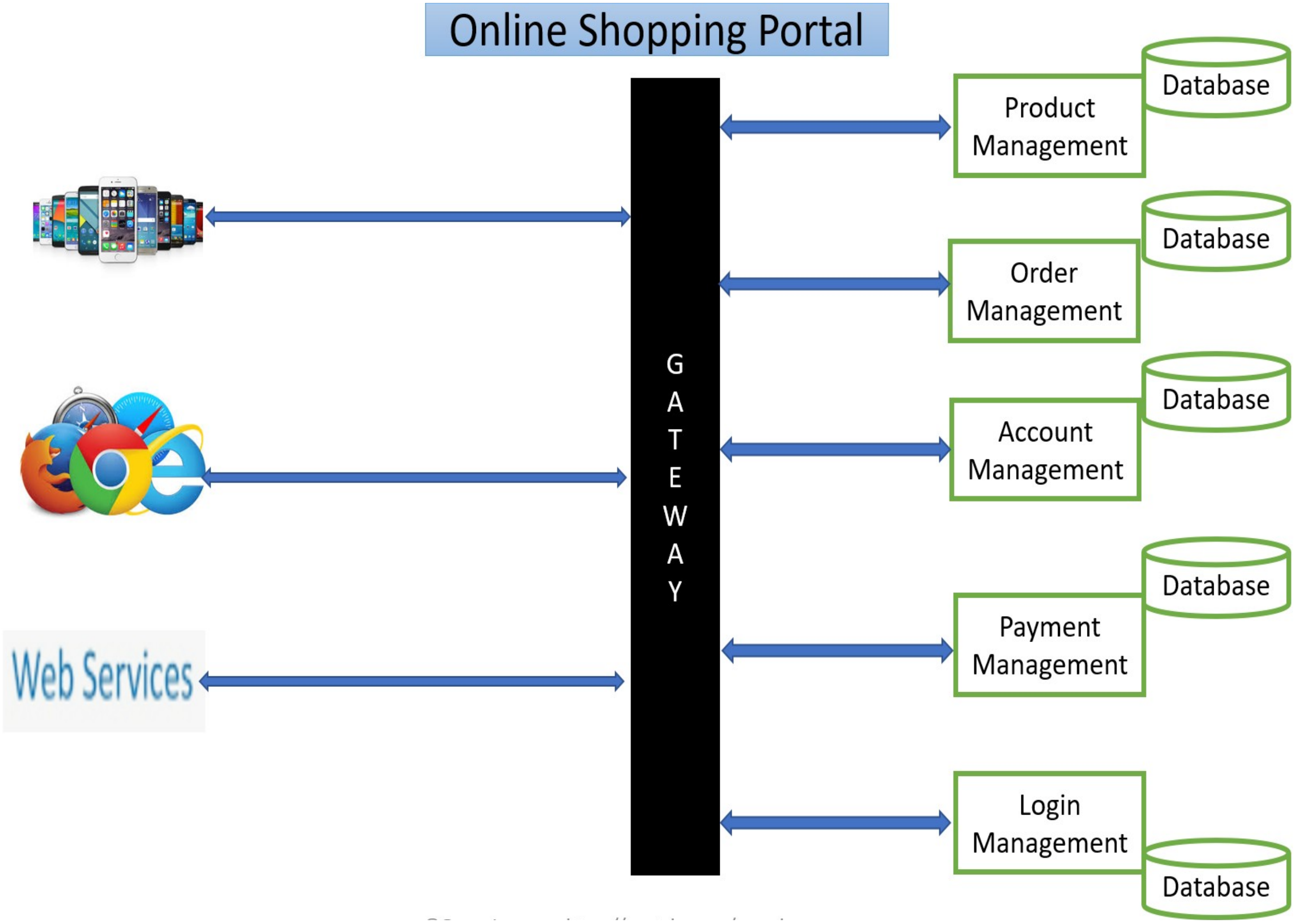
Microservices Architecture

Challenges with microservices architecture

- Services must be loosely couples so that they can be
  - Developed independently
  - Deployed independently
  - Scaled independently

- Unique requirement for each service
  - Different data
  - Different storage type

Database Per service



Database Per service

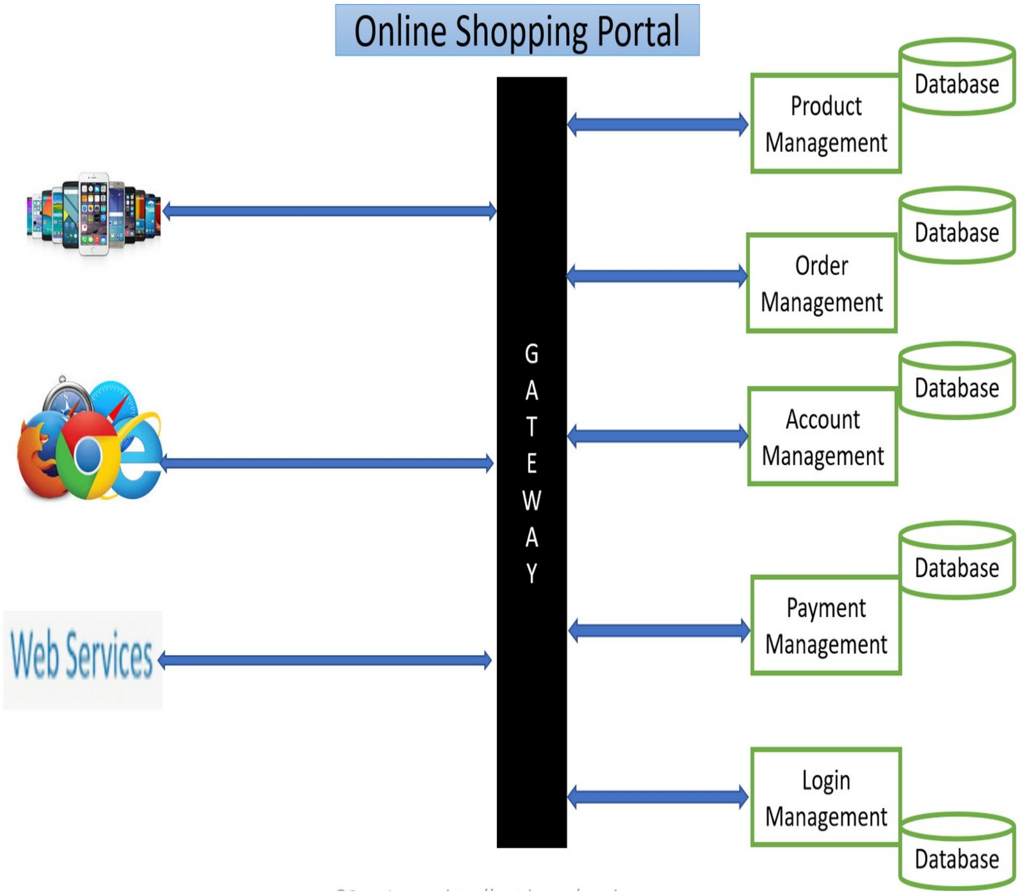
- Benefits
    - Loosely coupled
    - Free to choose database type e.g. RDBS, mongo Cassandra etc.

- if you are using a relational database then the options are:
    - Private-tables-per-service –
        - each service owns a set of tables that must only be accessed by that service
    - Schema-per-service –
        - each service has a database schema that’s private to that service
    - Database-server-per-service –
        - each service has it’s own database server.

Database Per service

- Challenges
  - Queries that needs join over multiple database
  - Transactions across multiple databases

- Solutions
  - Queries that needs join over multiple database
    - **CQRS**
    - **Event Sourcing**
    - **API Compositions**
  - Transactions across multiple databases
    - **Saga Pattern**



Shared DB per service

Brownfield Projects

Shared DB per service

- Brownfield Projects

ORDER table

Id	Cust_id	Status	sum	..
12345	9876	Complete	1000	

Customer table

Id	name	address	Address_type	
9876	xyz	Noida	home	
1111	abc	Delhi	office	

Shared DB per service

- Benefits
  - Familiar
  - Simpler to operate
- Drawbacks
  - Runtime coupling
  - Development time coupling
  -

What next??

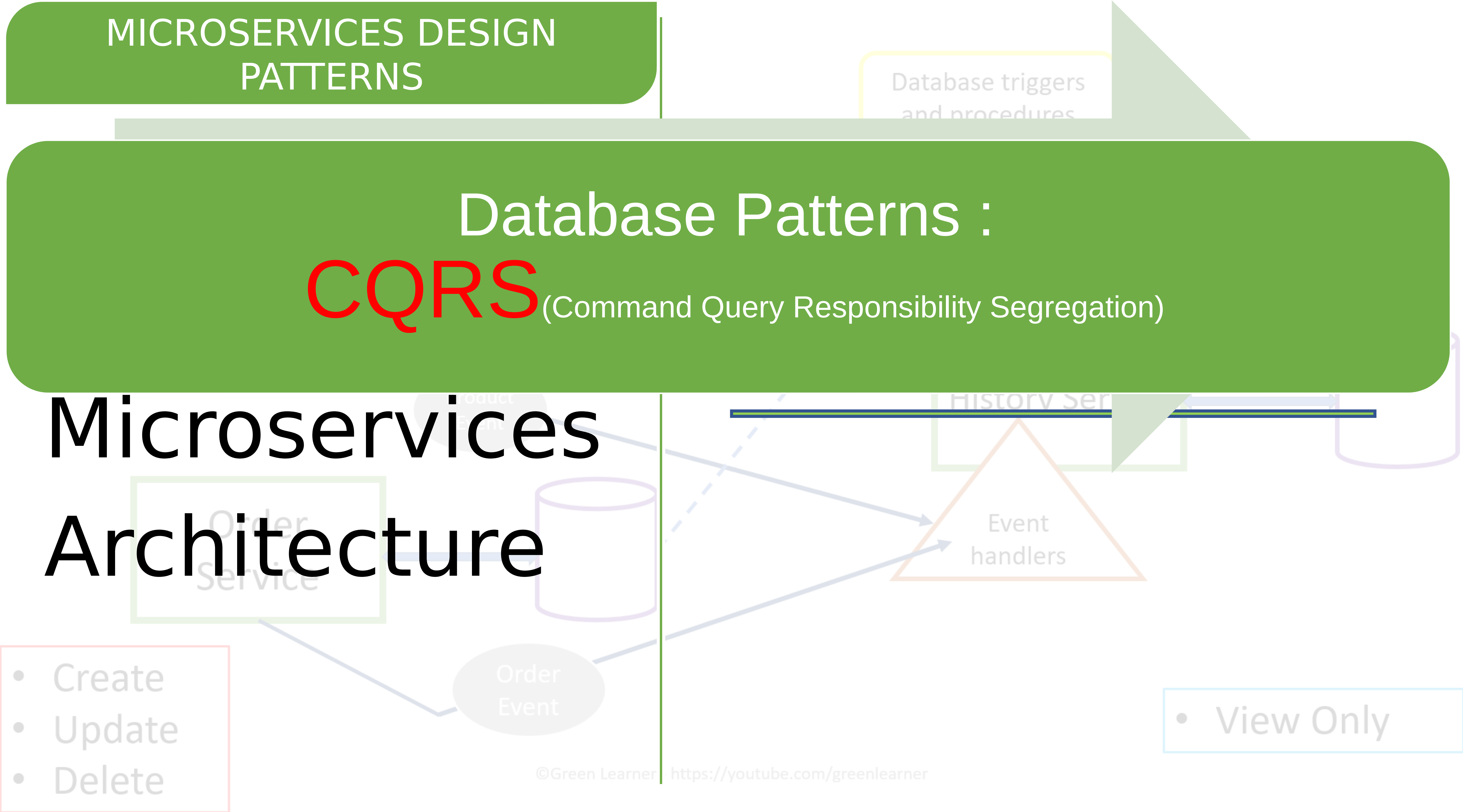
Database Pattern : CQRS

MICROSERVICES DESIGN PATTERNS

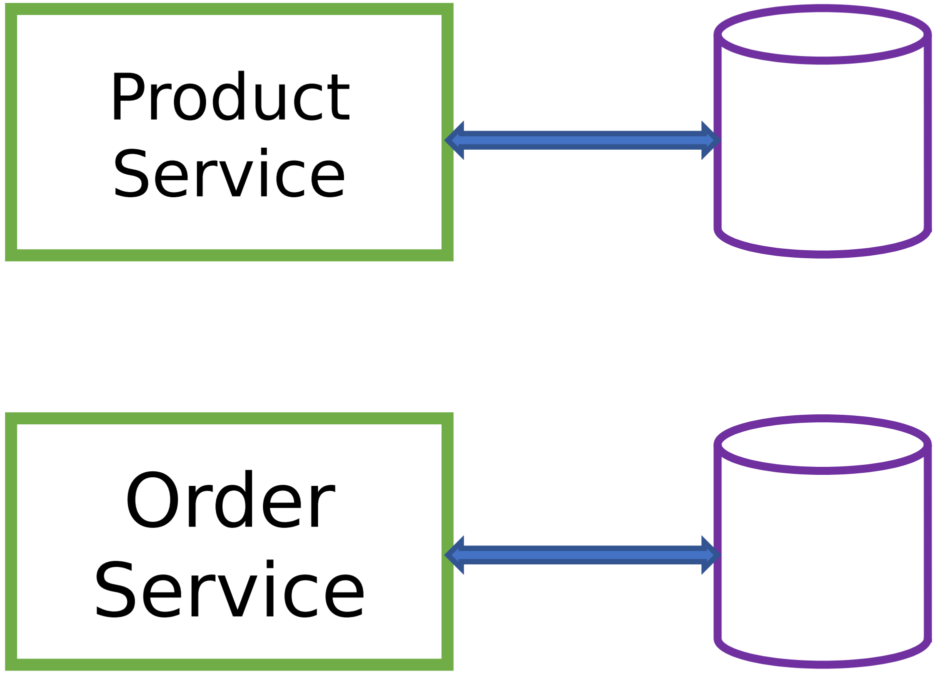
Database Patterns :

CQRS (Command Query Responsibility Segregation)

Microservices Architecture



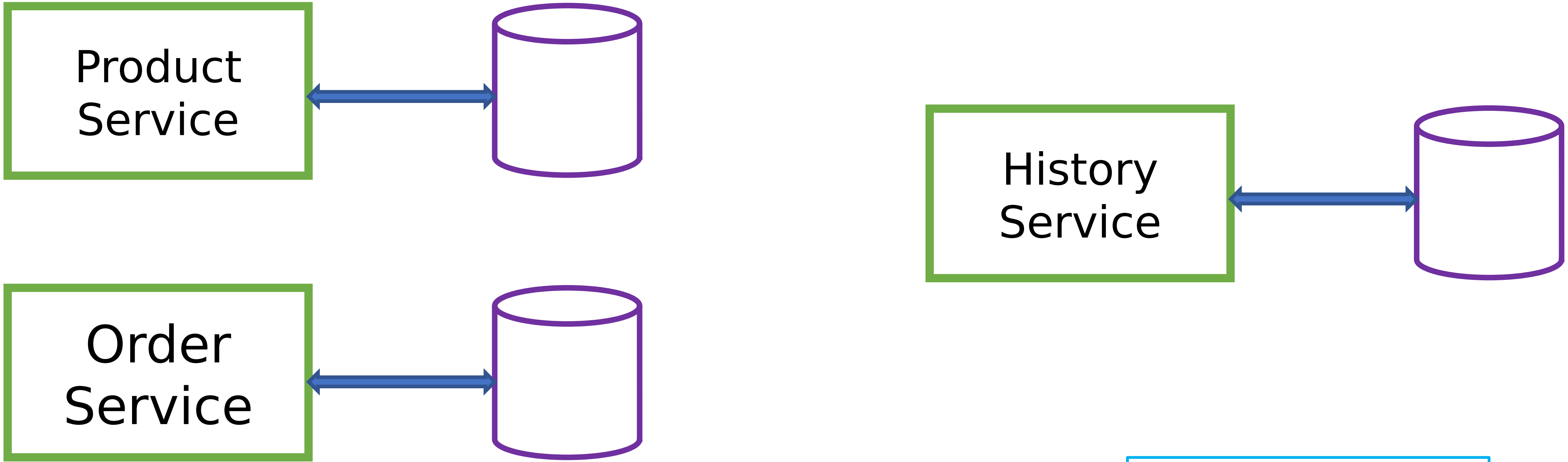
Problem Statement



- Create
- Update
- Delete

- View Only

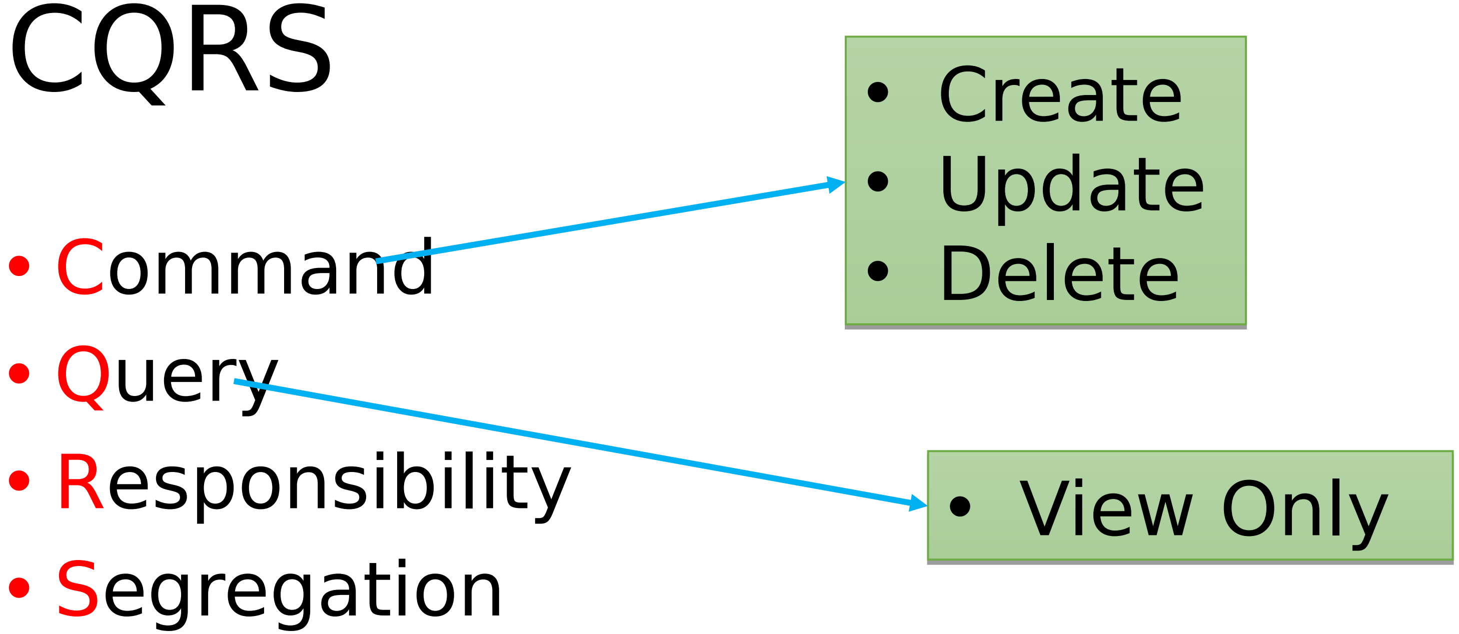
The Way out



- Create
- Update
- Delete

- View Only

CQRS

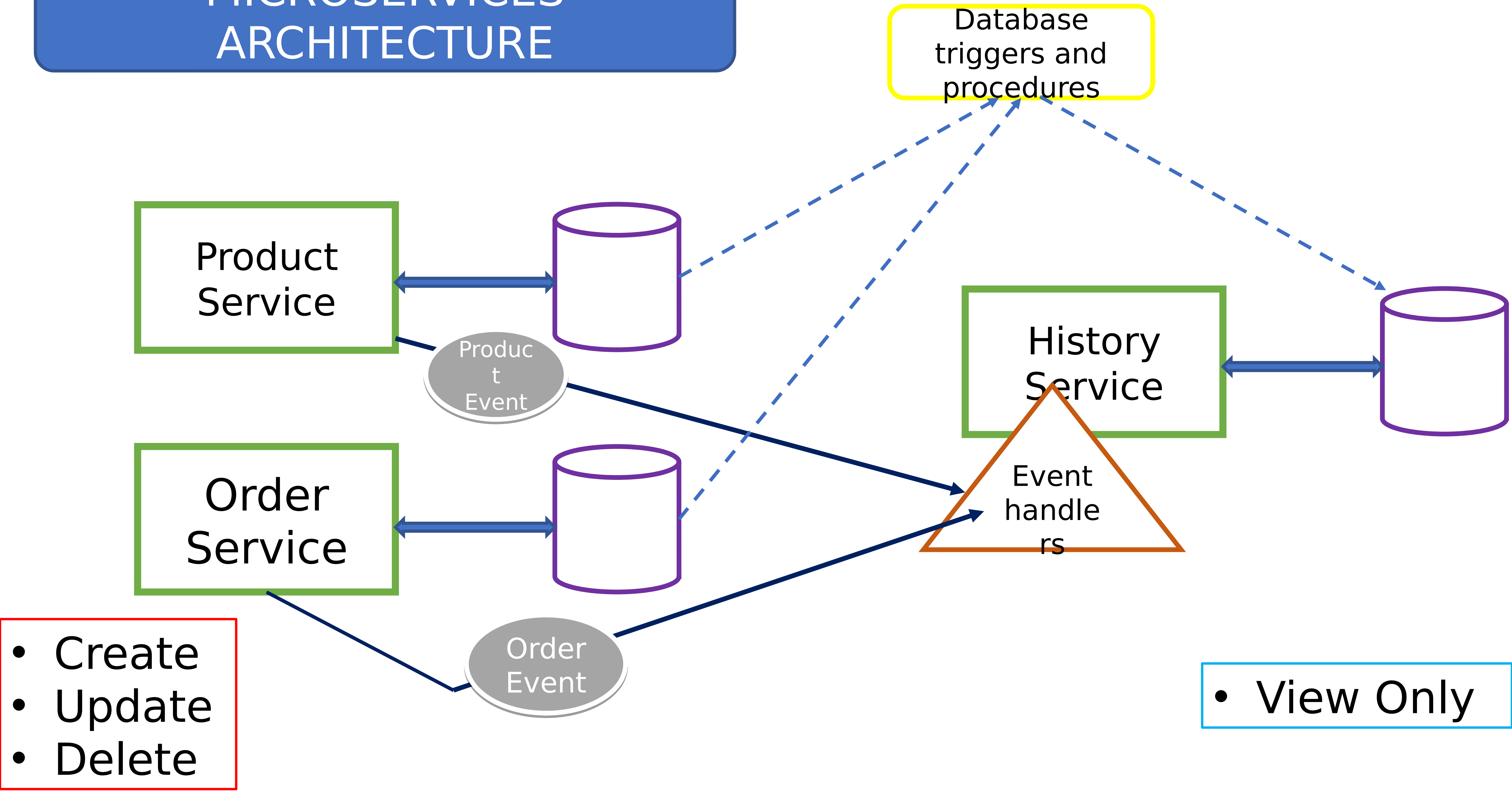


- Command
- Query
- Responsibility
- Segregation

- Create
- Update
- Delete

- View Only

MICROSERVICES ARCHITECTURE



Database triggers and procedures

Product Service

Product Event

Order Service

Order Event

History Service

Event handlers

- Create
- Update
- Delete

- View Only

CQRS : Challenges

- How the data will come to History Service Database?
  - Events
  - Database replication methods
- Replication delay
- Extra complexity
- Code Duplication

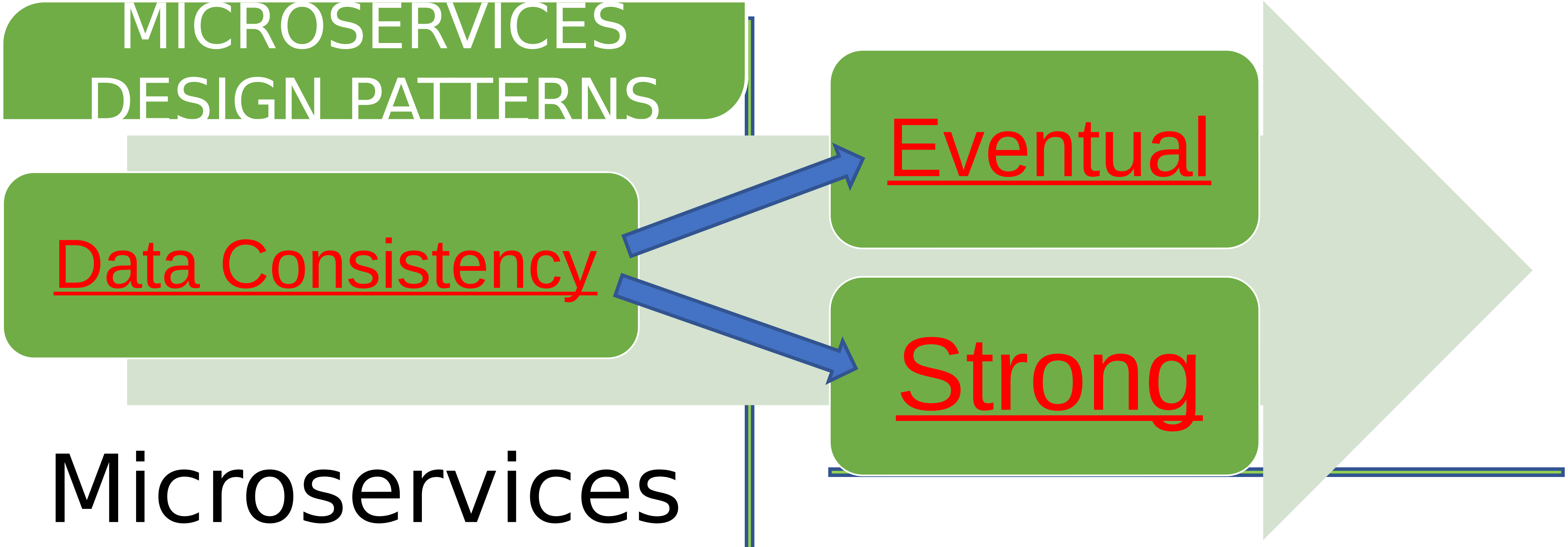
CQRS : Benefits

- Responsibility segregation which is the core principle of microservices/distributed systems
- Simpler command and query models
- Flexibility to choose database for view
-

What next??

Eventual Consistency

MICROSERVICES DESIGN PATTERNS



Data Consistency

Eventual

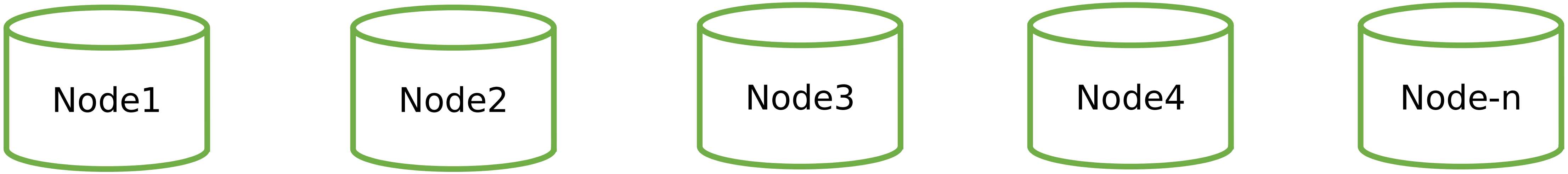
Strong

Microservices Architecture

Data Consistency in Distributes Systems

Eventual Consistency

Strong/Strict Consistency



Node1

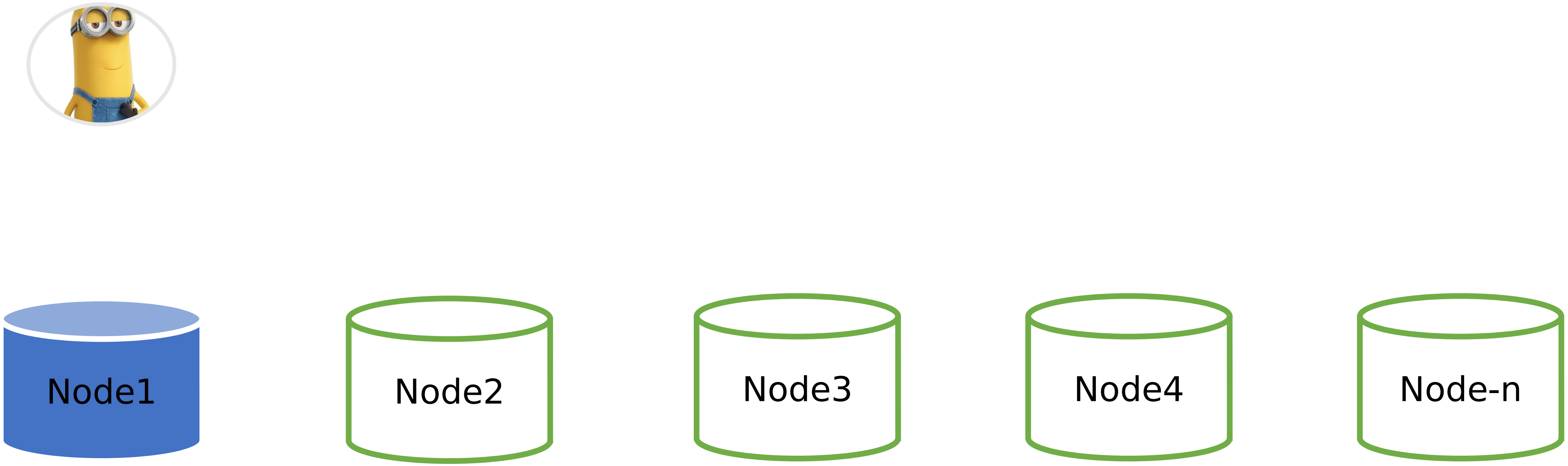
Node2

Node3

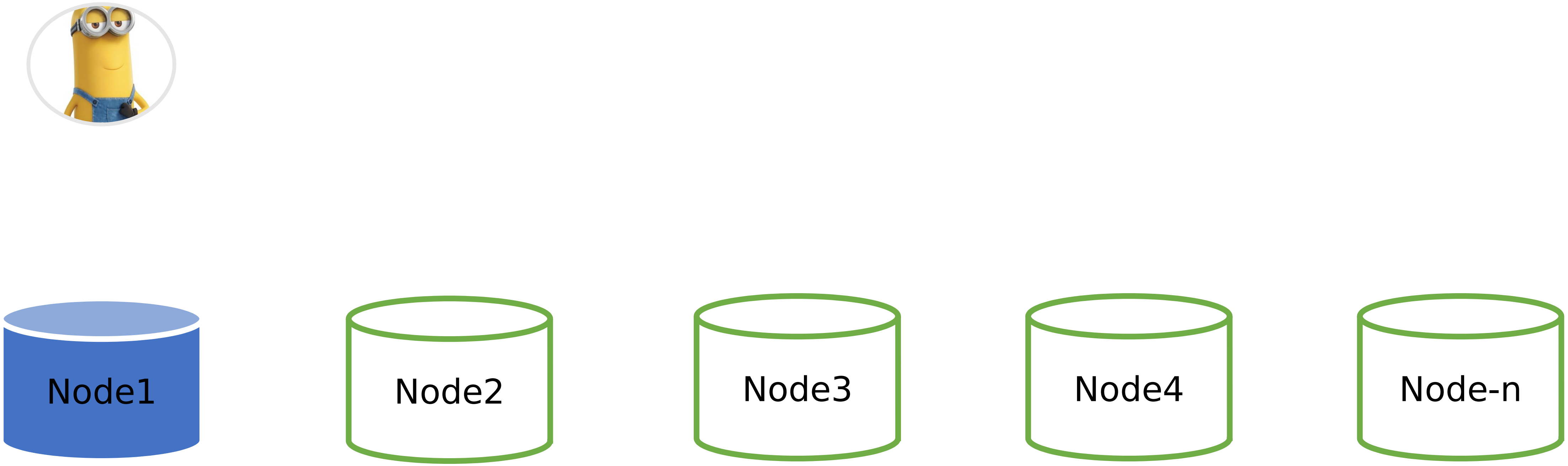
Node4

Node-n

Eventual Consistency



Strong/Strict Consistency



Eventual vs Strong Consistency

- Eventual consistency guarantees low latency with some stale data

- Strong consistency guarantees updated/latest data with some higher latency

What next??

Database Pattern : Event Driven

MICROSERVICES DESIGN PATTERNS

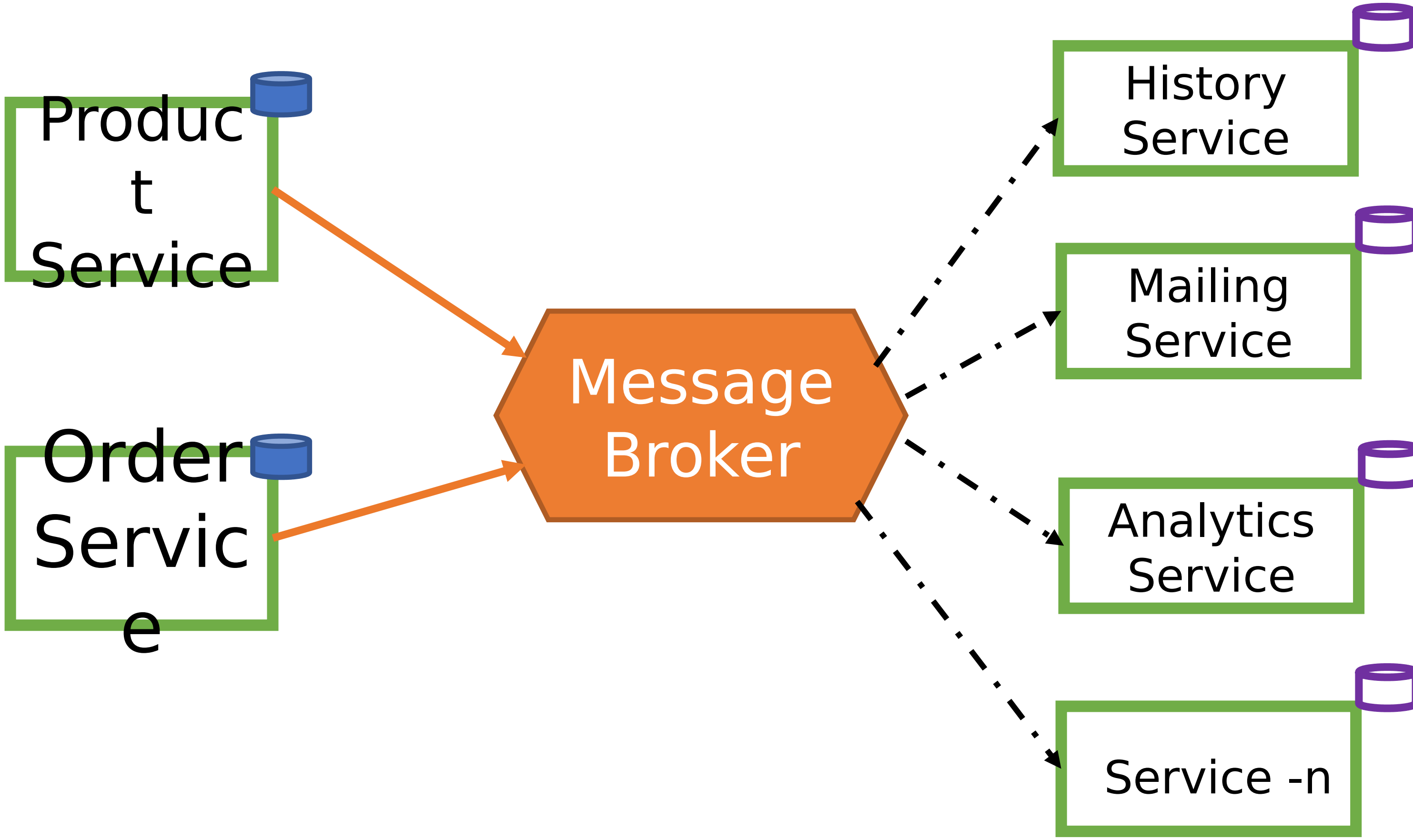
Database Pattern :

Event Driven

Microservices Architecture

Event-Driven System

- Use of Message Brokers
- Decoupled architecture
- Asynchronous communication



Event Notification
Event Carried State Transfer

https://martinfowler.com/articles/201701-event-driven.html

What next??

Database Pattern : Event Sourcing

MICROSERVICES DESIGN PATTERNS

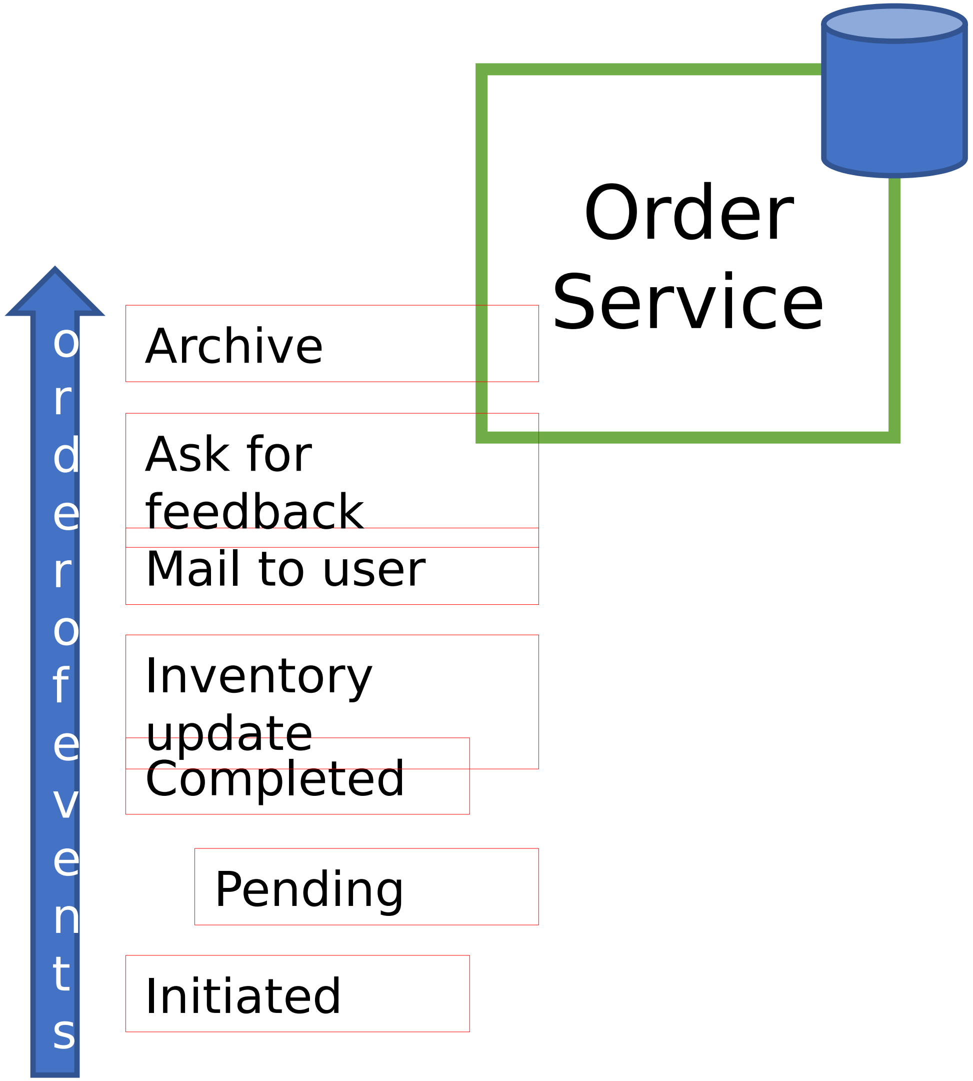
Database Pattern :

Event Sourcing

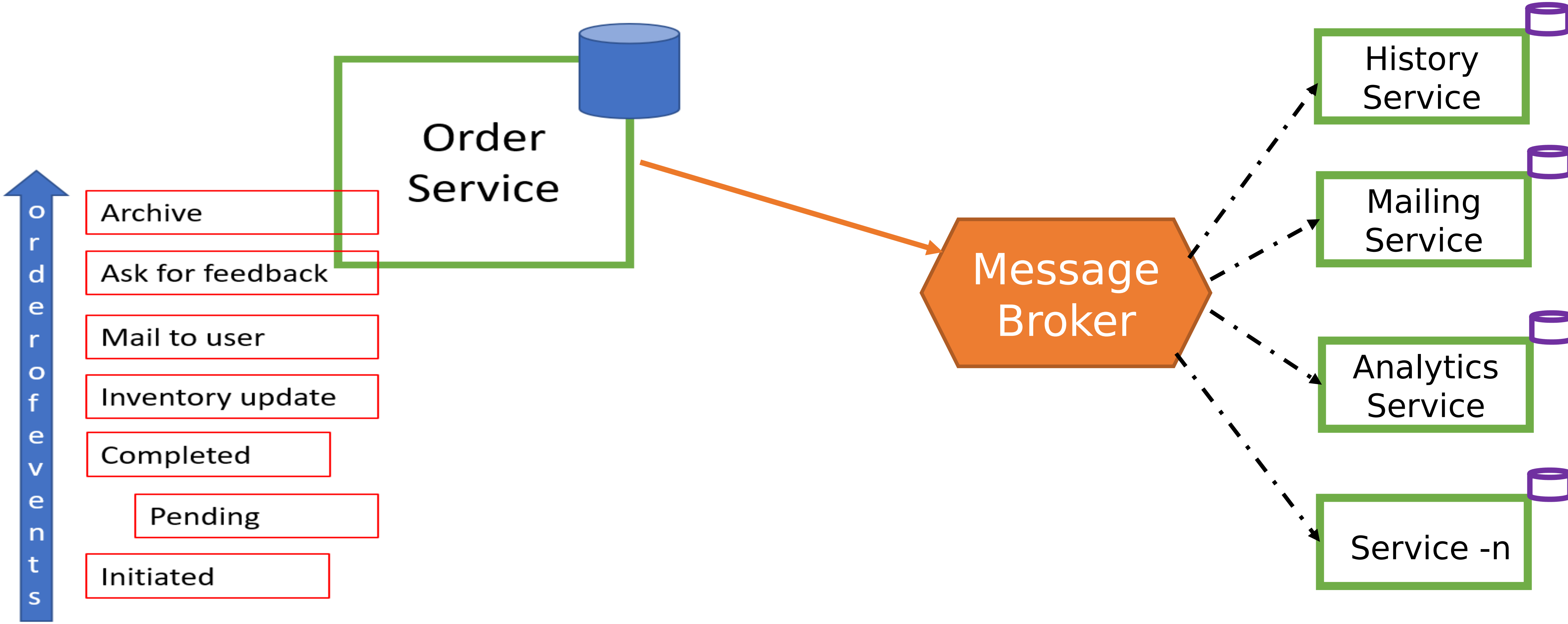
Microservices Architecture

Event-Sourcing System

- State is stored as a series of events
- Any record is about the state change from previous one
- We can always replay events to get the state at any point of time
  - E.g. Git Commit
- Efficient
- Asynchronous communication
- Storage vs performance
  - Regular snapshots



Event-Sourcing System



©Green Learner - https://youtube.com/greenlearner

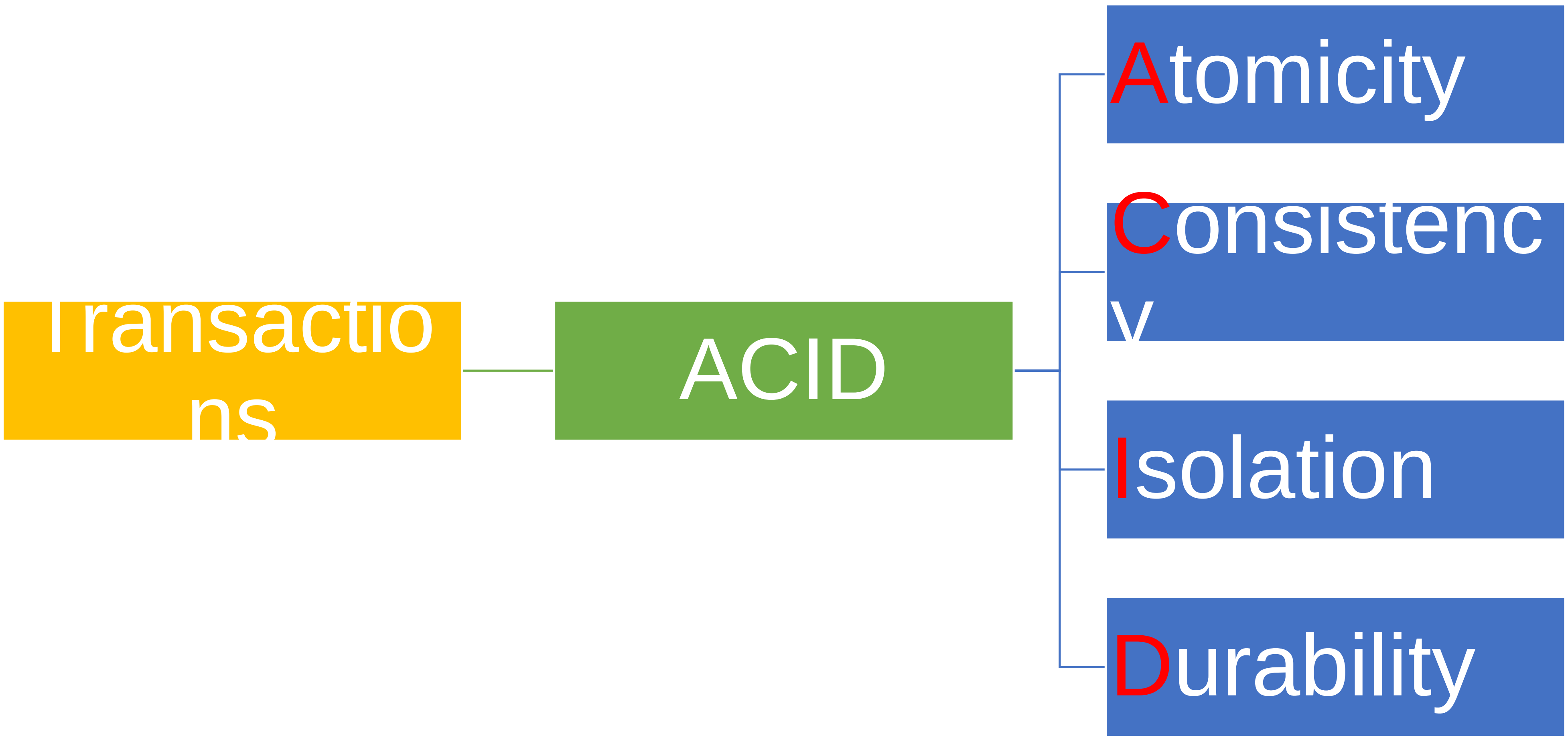
What next??

Database Pattern : 2 Phase Commit

MICROSERVICES DESIGN PATTERNS

Database Pattern : 2 Phase Commit

Microservices Architecture



Transactions

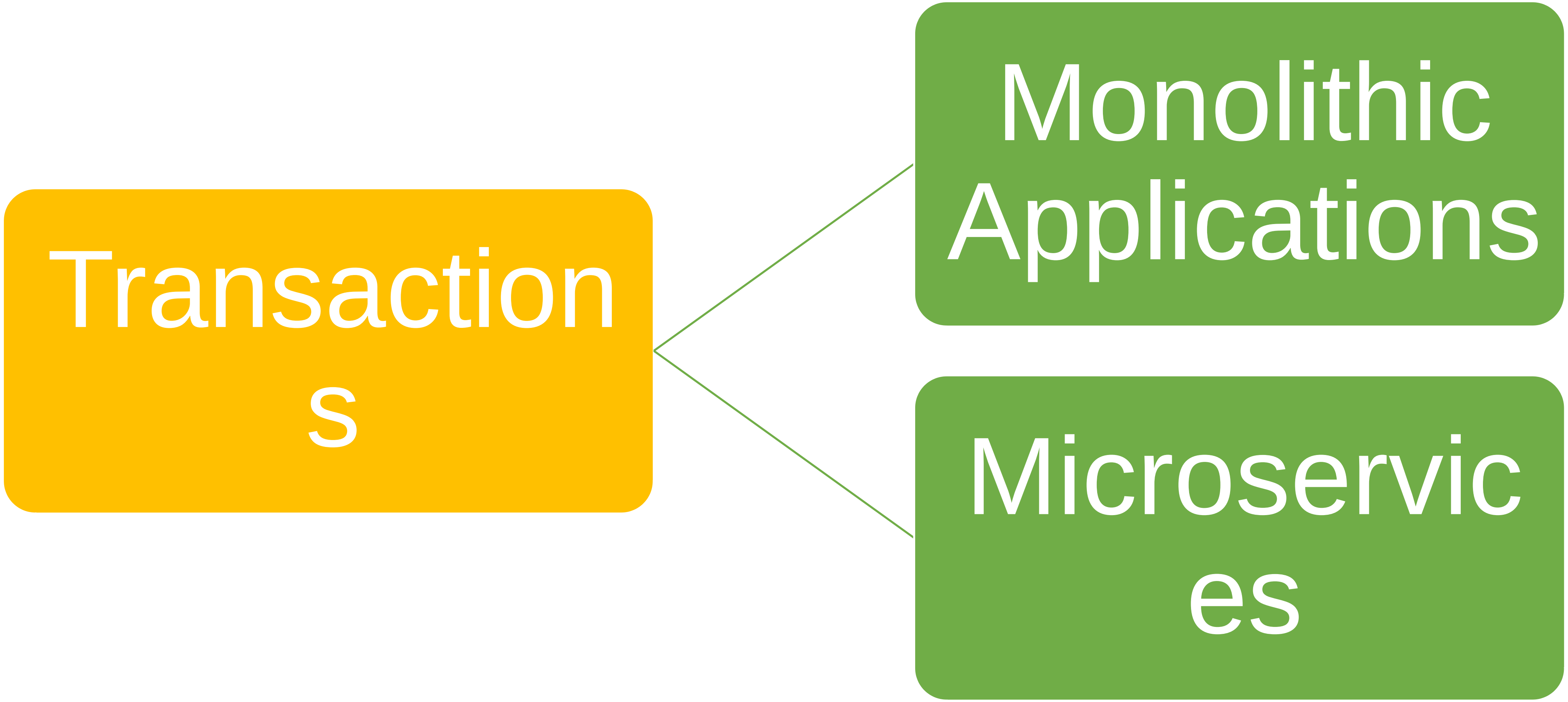
ACID

Atomicity

Consistency

Isolation

Durability



Transactions

Monolithic Applications

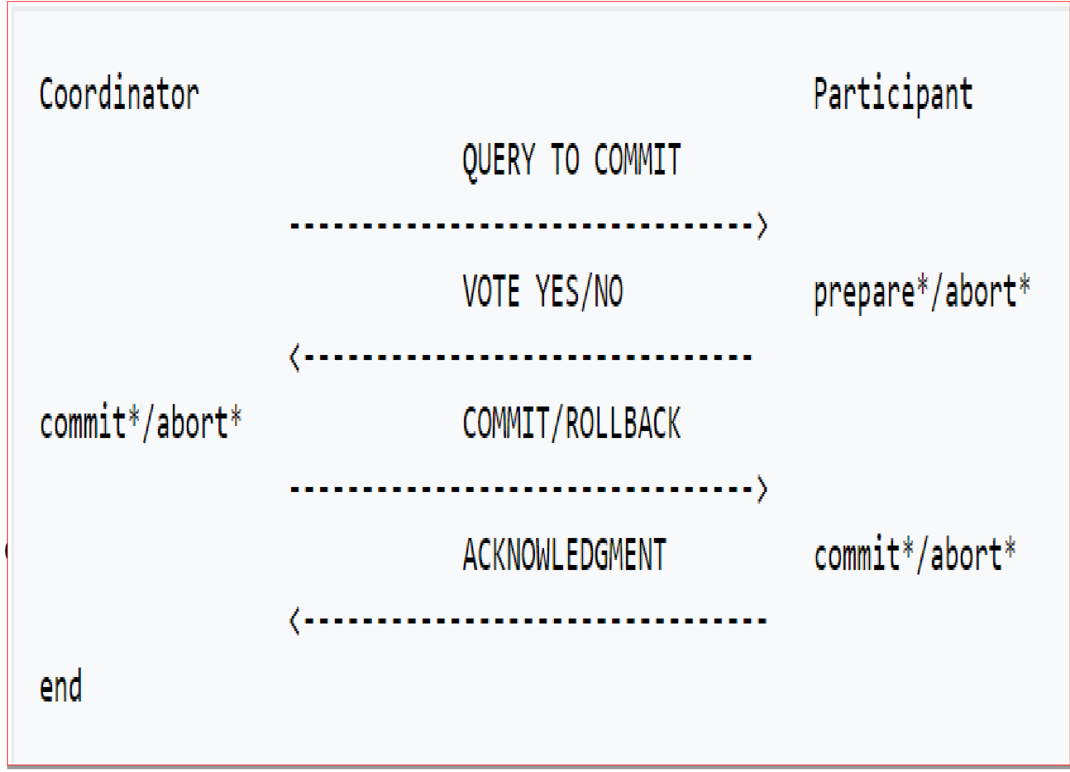
Microservices

2 Phase Commit Protocol

- XA(Extended Architecture)
- It’s a pattern for distributed transactions
- ACID like properties for global/distributed transaction processing
- Transaction manager manages the transactions
  - Preparation for commit or abort
  -

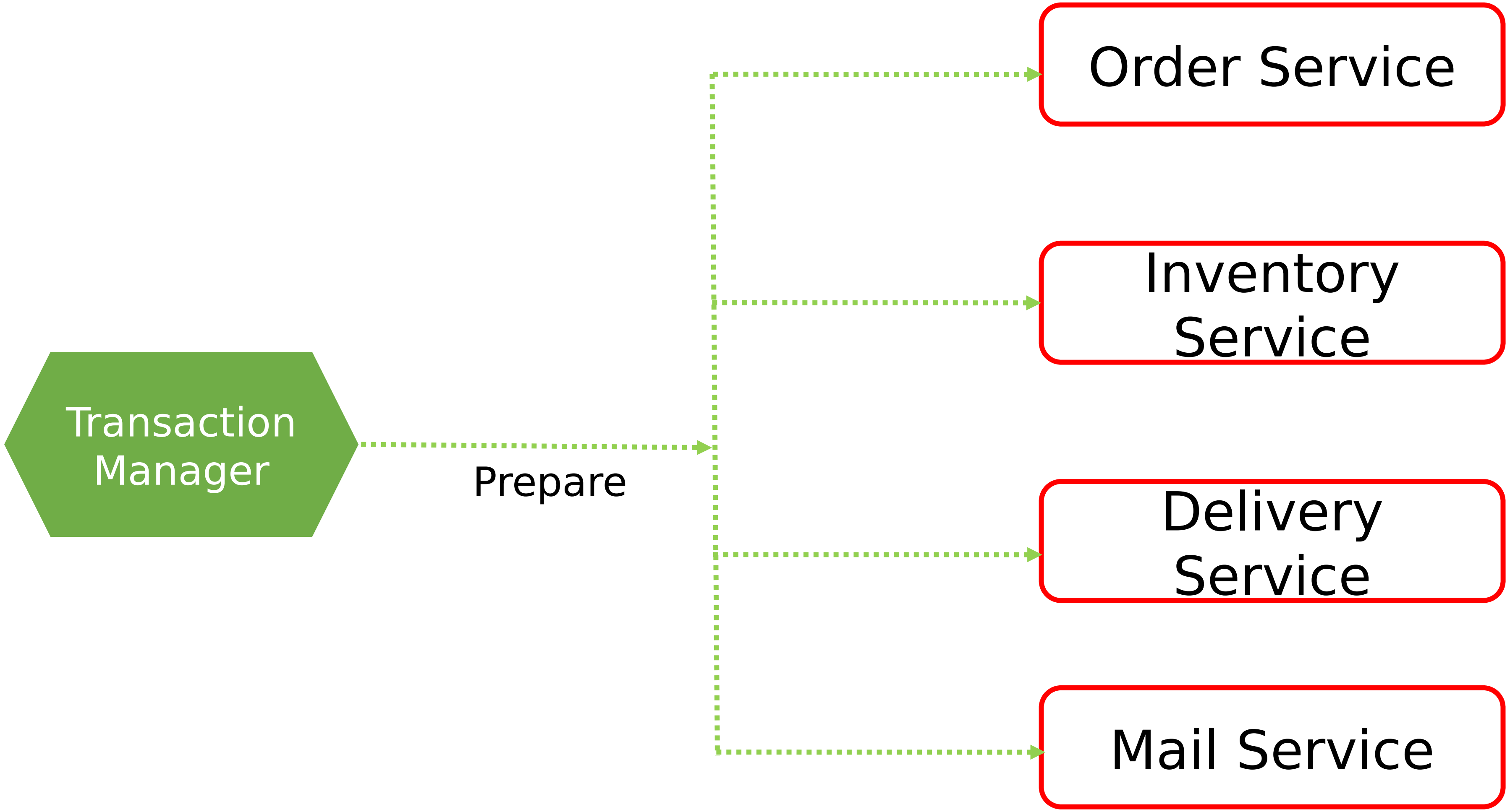
Basic Algorithm

- Phase 1 – Preparation/Voting
  - TM sends query to commit to all participants
  - Participants reply with appropriate message(Yes/N
    - Also makes an entry in undo and redo logs
- Phase 2 - Action
  - Commit
    - TM Sends ***commit*** message to all participants
    - Each participants completes the operation and releases the lock
      - Sends acknowledgement to TM
    - TM marks the transaction complete with success
  - Abort
    - TM Sends ***abort*** message to all participants
    - Each participants aborts the operation and releases the lock
      - Sends acknowledgement to TM
    - TM marks the transaction complete with failure

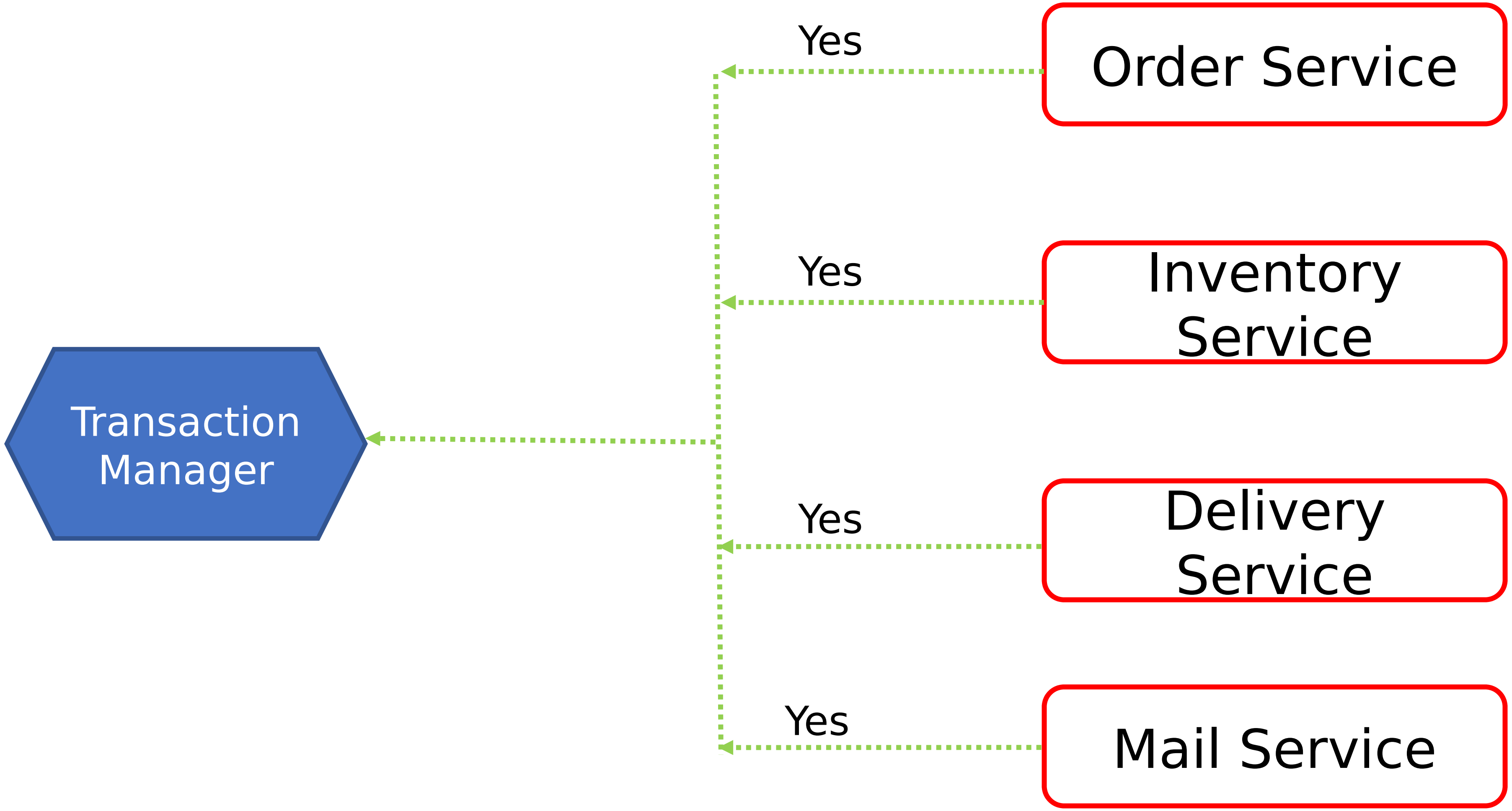


```
Coordinator                                         Participant
                          QUERY TO COMMIT
                -------------------------------->
                          VOTE YES/NO               prepare*/abort*
                <-------------------------------
commit*/abort*            COMMIT/ROLLBACK
                -------------------------------->
                          ACKNOWLEDGMENT            commit*/abort*
                <--------------------------------
end
```

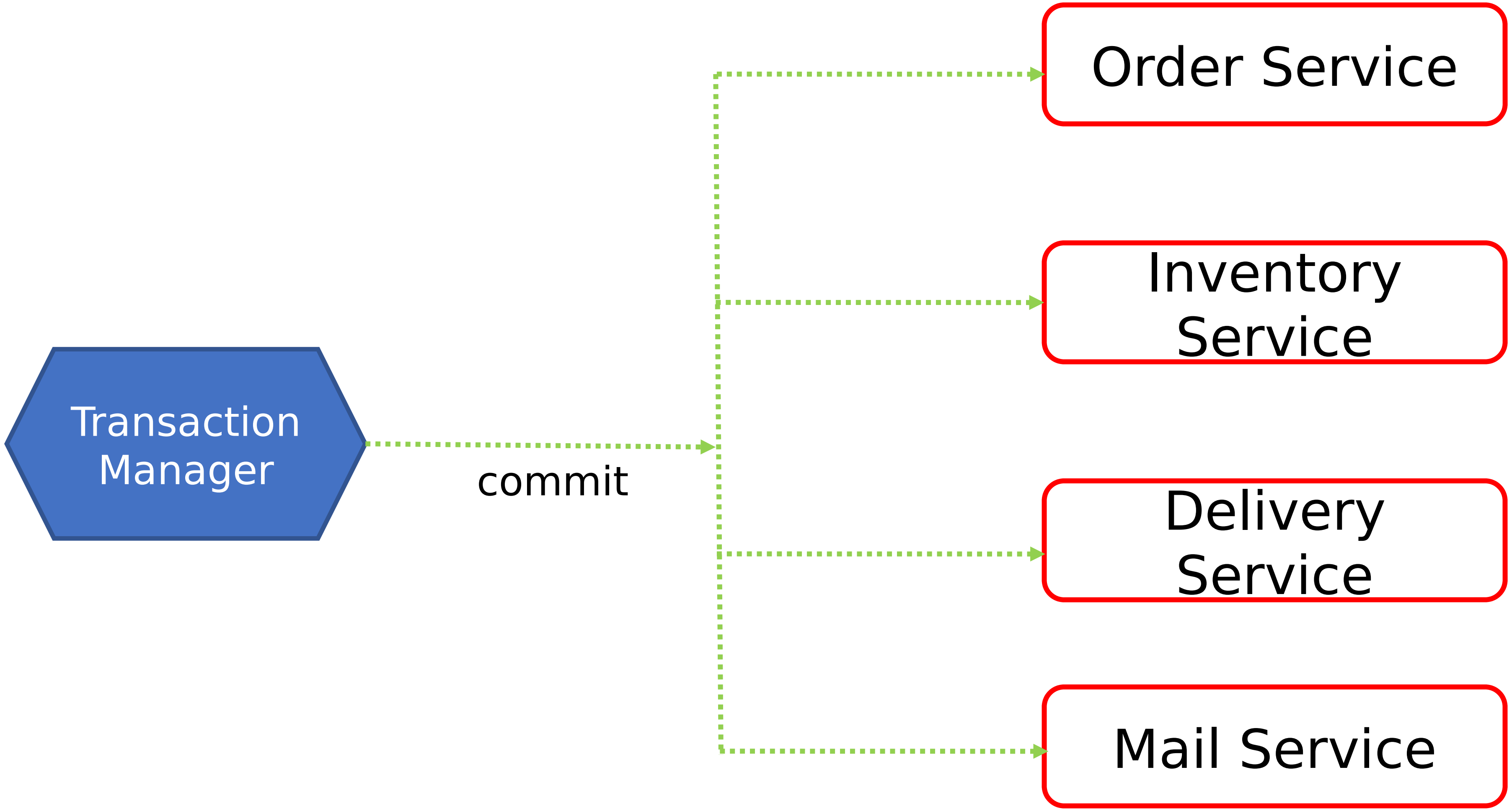
Preparation Phase



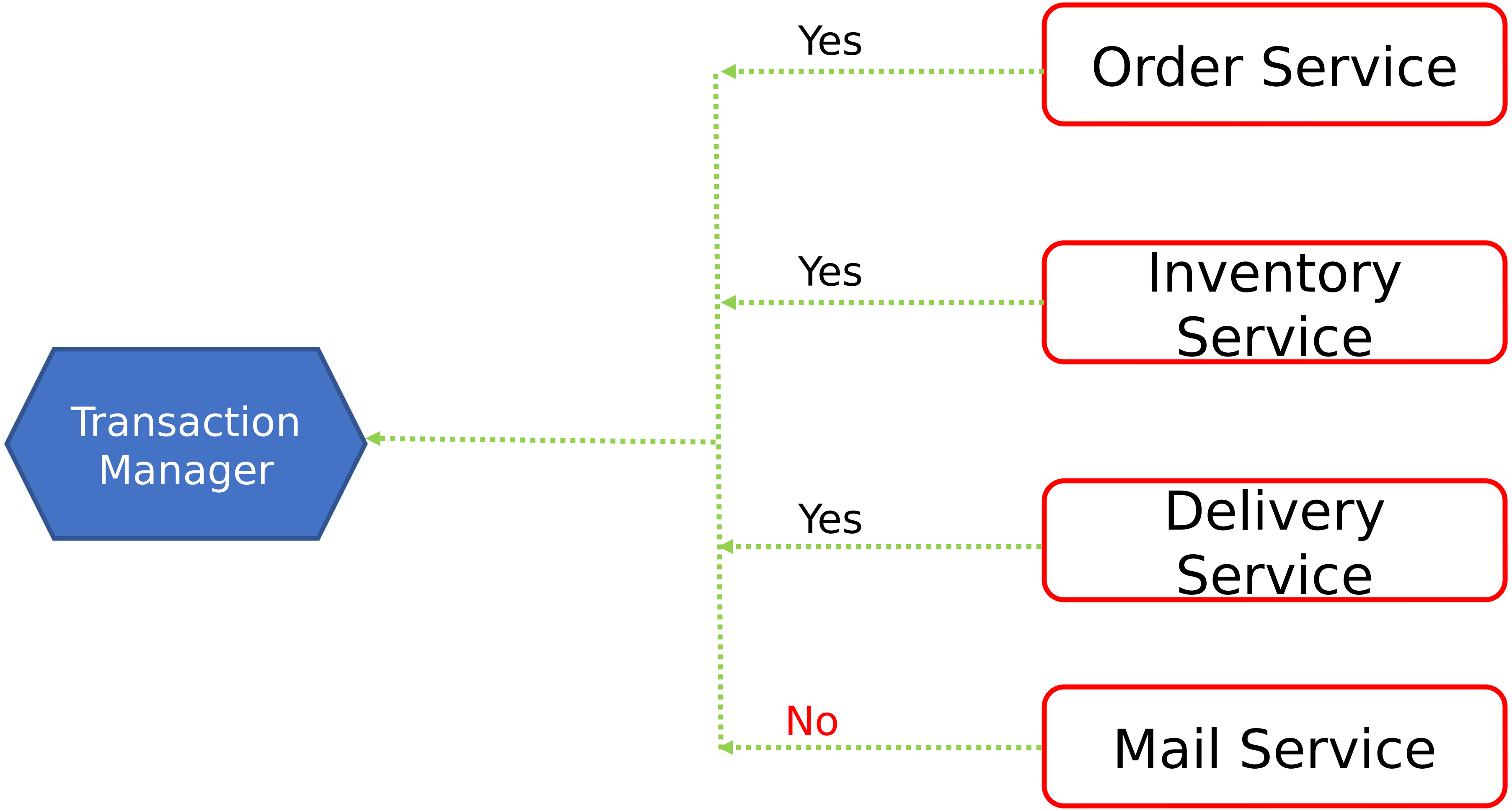
Voting Phase – All Yes



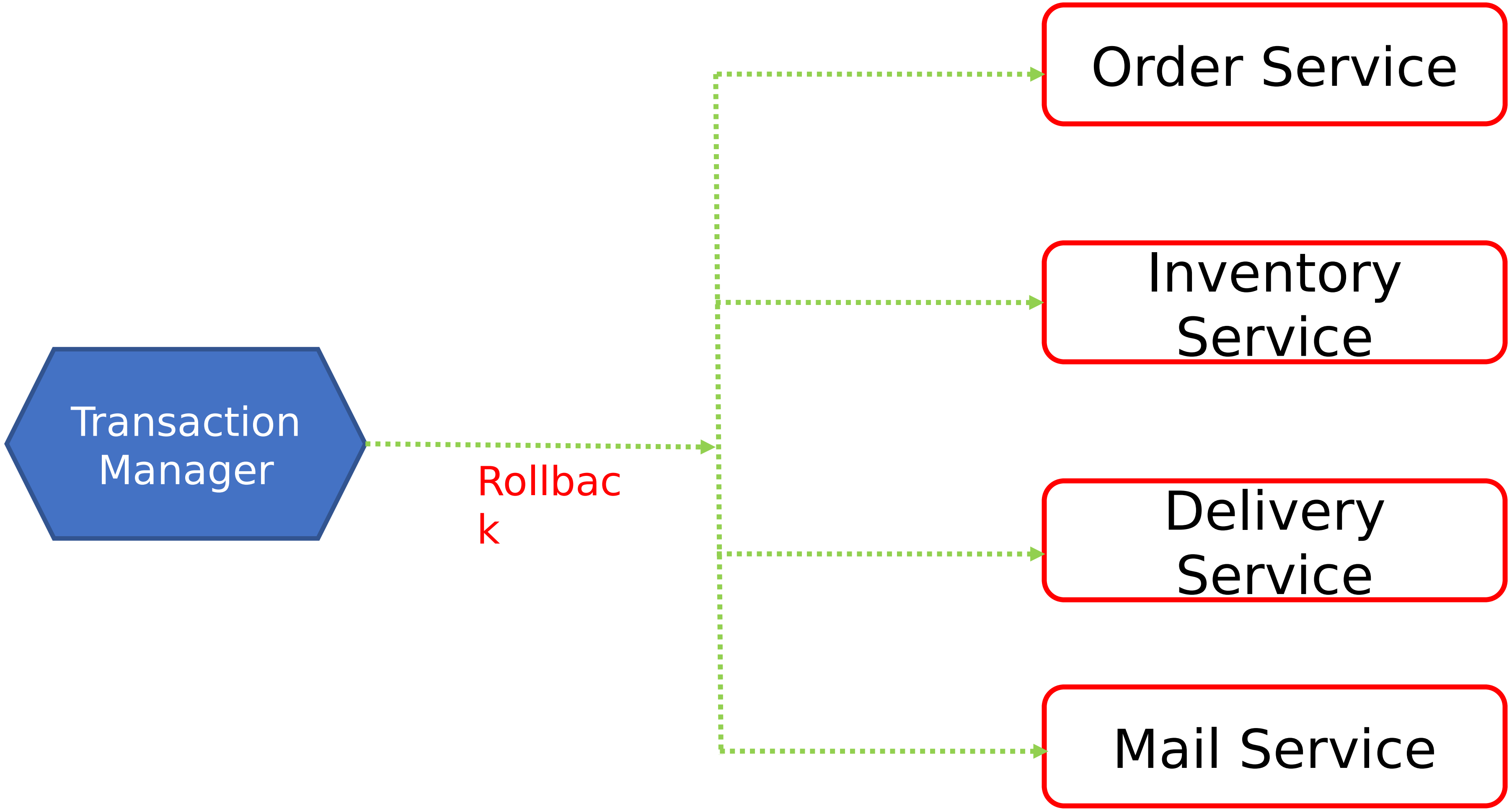
Action - commit



Voting Phase – No by at least on service



Action - Rollback



Challenges with 2 Phase Commit

- Complete reliance on transaction manager
  - One point failure
  - Scaling issues
  - Reduced throughput
- No response from participant services
  - Then resources is locked until timeout
- Commit failure after successful vote
- Locks resources due to pending transactions

What next??

Database Pattern : Saga

MICROSERVICES DESIGN PATTERNS

Database Pattern : Saga

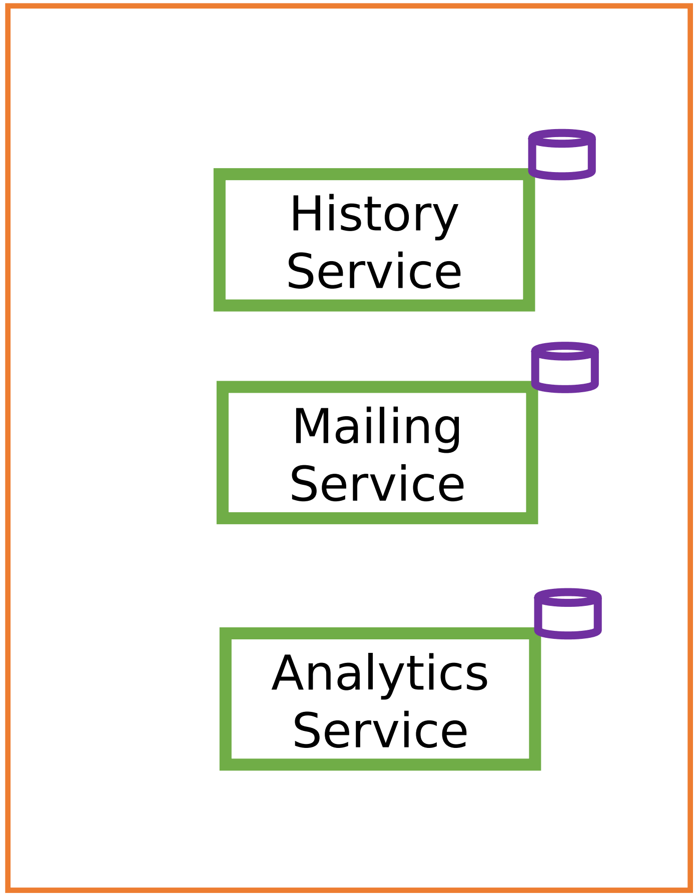
Microservices Architecture

Problem Statement

- Transaction
  - Single unit of logic or work
  -

- Transactions must be
  - Atomic
  - Consistent
  - Isolated
  - Durable

- Within a single service maintaining ACID properties for transactions are easy, but
- Cross-service transaction requires a cross-service transaction management strategy.



What is Saga?

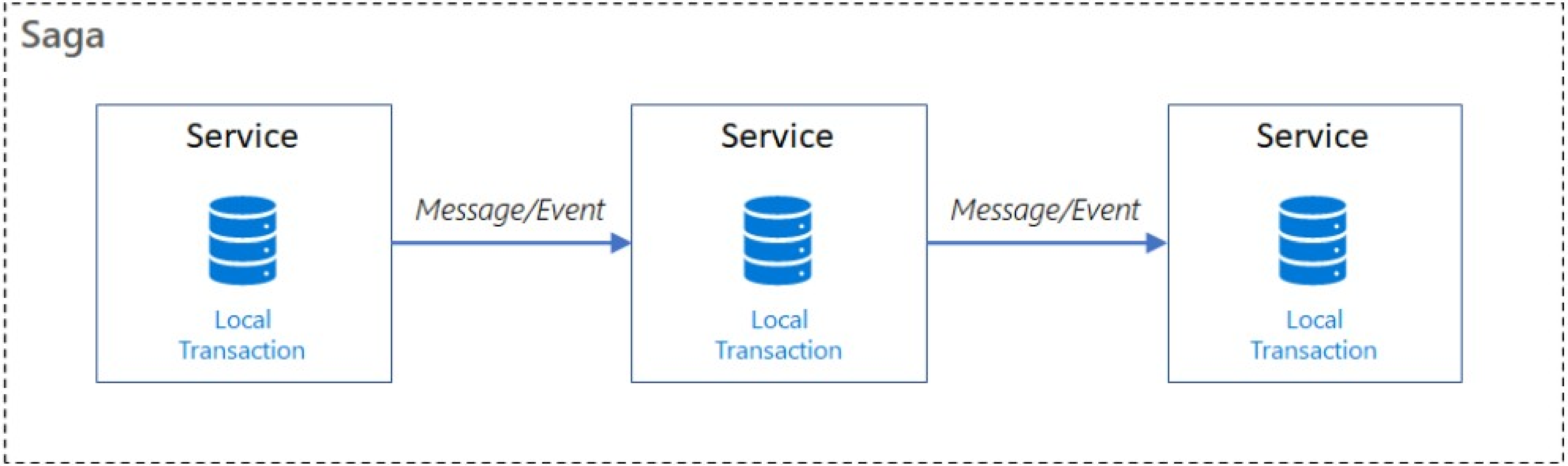
- It’s a way to manage data consistency across microservices in distributed transaction scenarios.

- It’s a sequence of transactions that updates each service and publishes a message or event to trigger the next transaction step.

- If a step fails, the saga executes compensating transactions that counteract the preceding transactions.

Saga

- Sequence of local transactions
- Each local transaction updates the database and published the event for next service
- If any of the local transaction fails then saga publishes COMPENSATING transactions to all the services which have performed commit



Saga patterns have -

Compensable transactions are transactions that can potentially be reversed by processing another transaction with the opposite effect.

A ***pivot transaction*** is the go/no-go point in a saga. If the pivot transaction commits, the saga runs until completion. A pivot transaction can be a transaction that is neither compensable nor retryable, or it can be the last compensable transaction or the first retryable transaction in the saga.

Retryable transactions are transactions that follow the pivot transaction and are guaranteed to succeed.

Saga Implementation

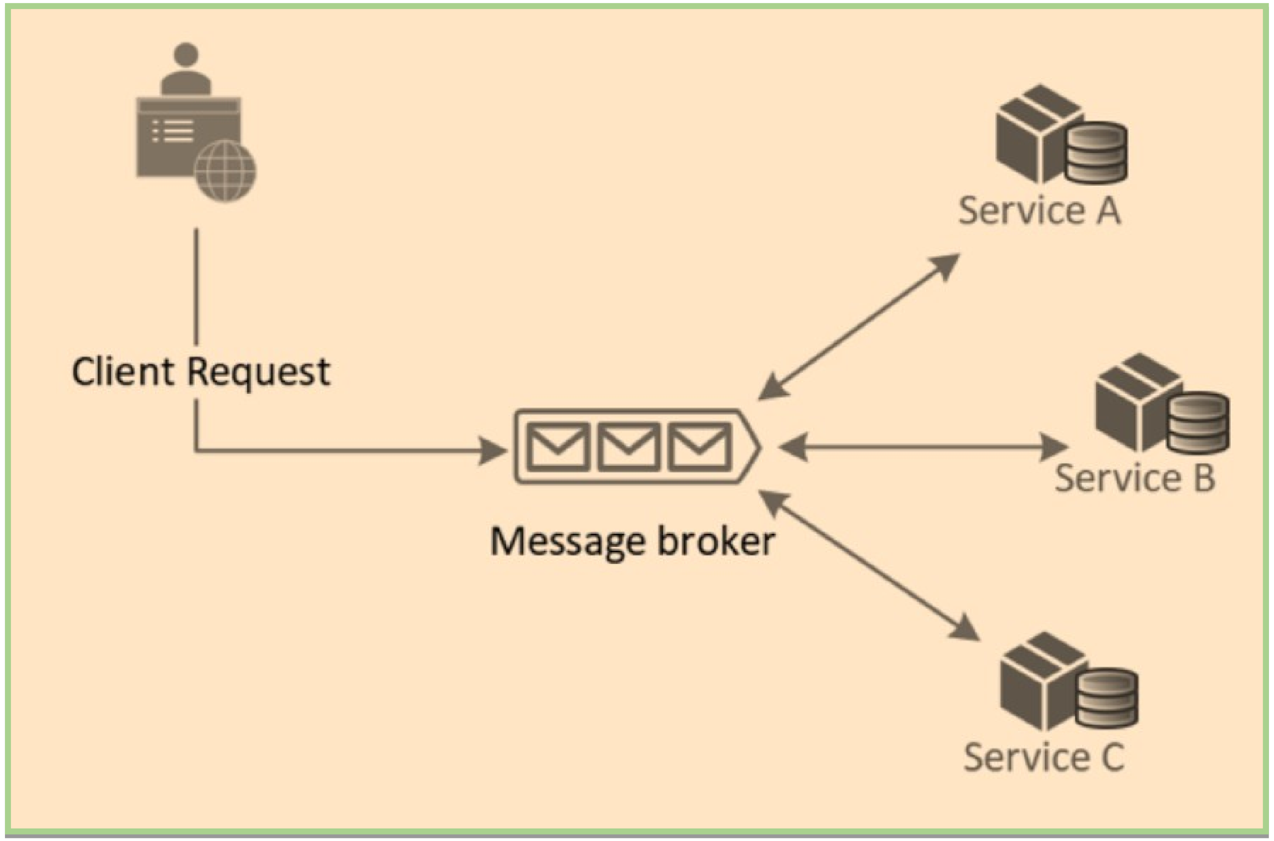
Choreography

Orchestration

Choreography

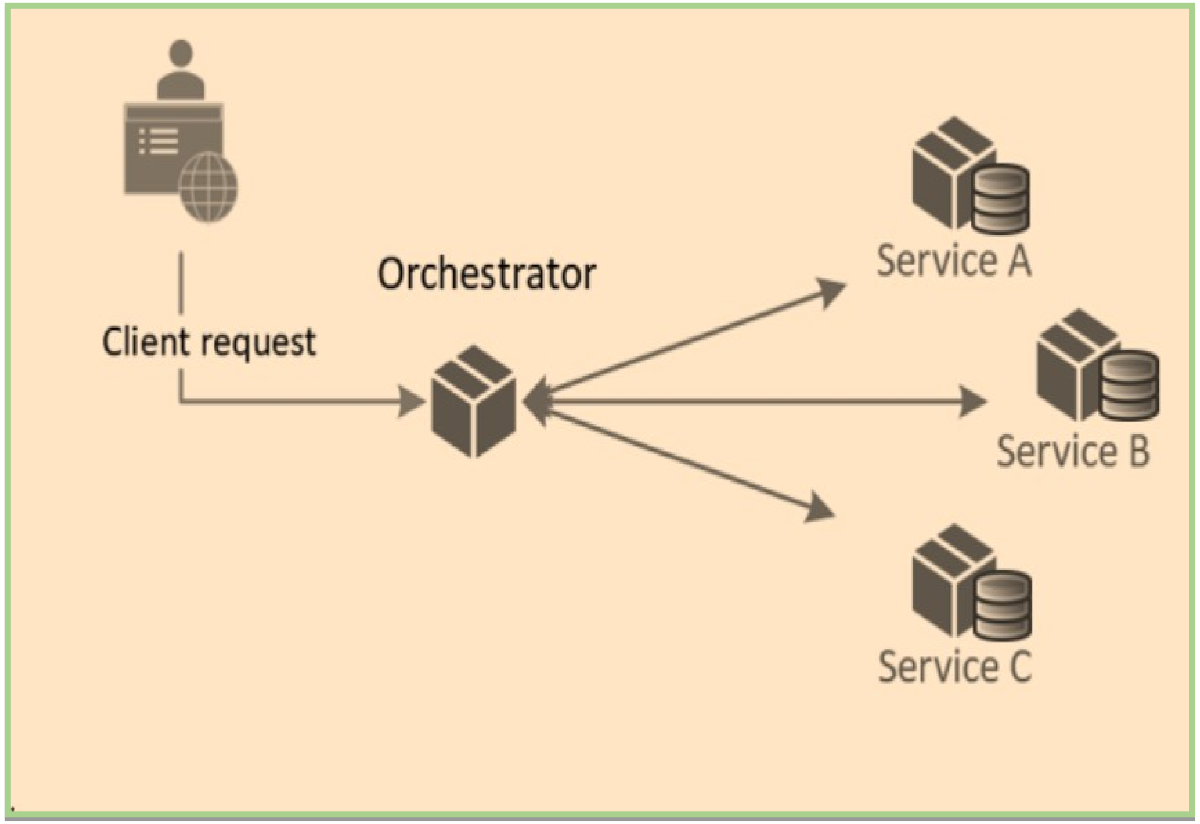
- ***Benefits***
  - ***good for simple workflows which have few services***
  - ***No additional service***
  - ***No single point of failure***

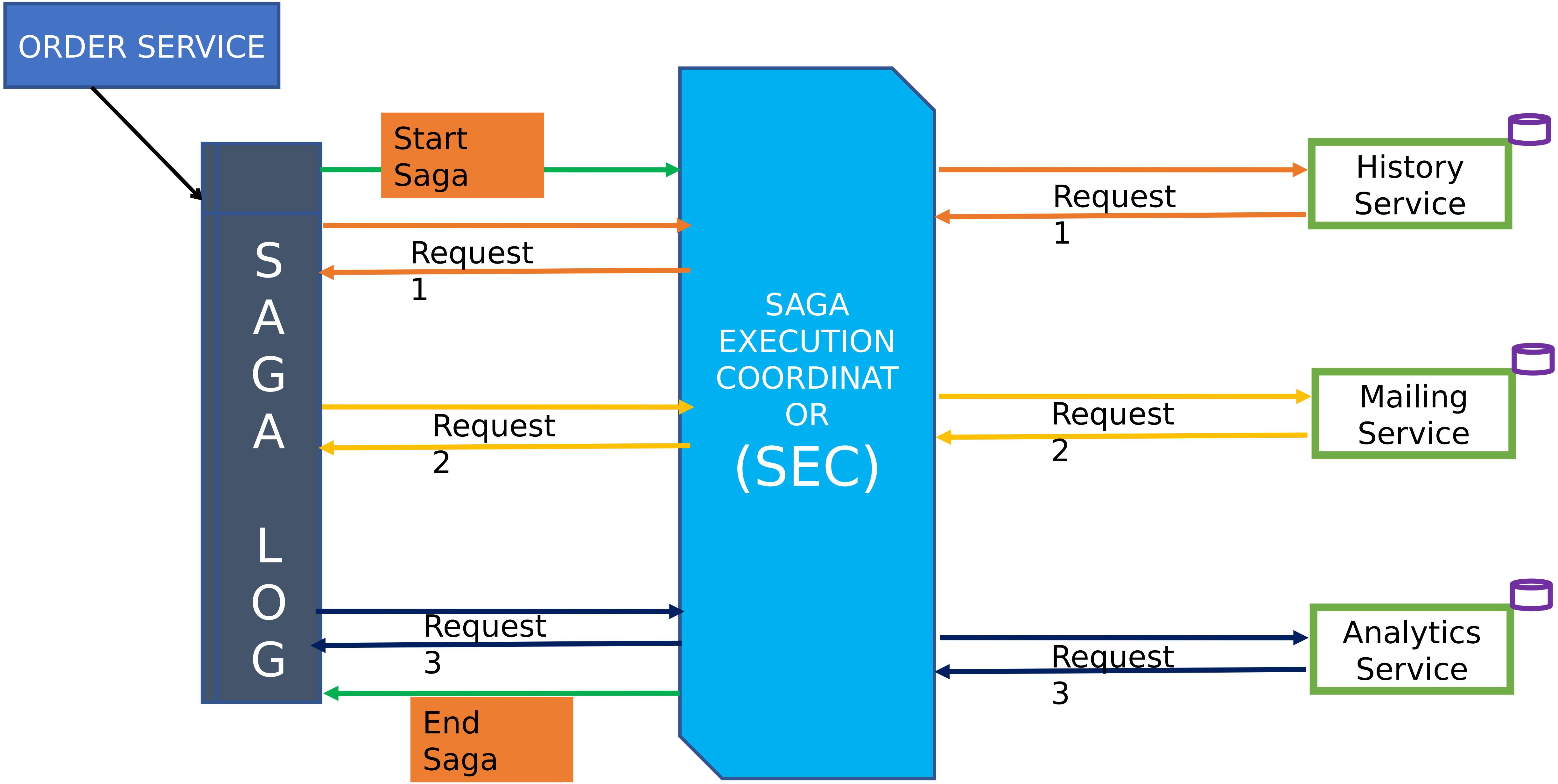
- ***Drawbacks***
  - ***Can be confusing when adding new steps***
  - ***Cyclic dependency risk***
  - ***Integration testing is difficult***

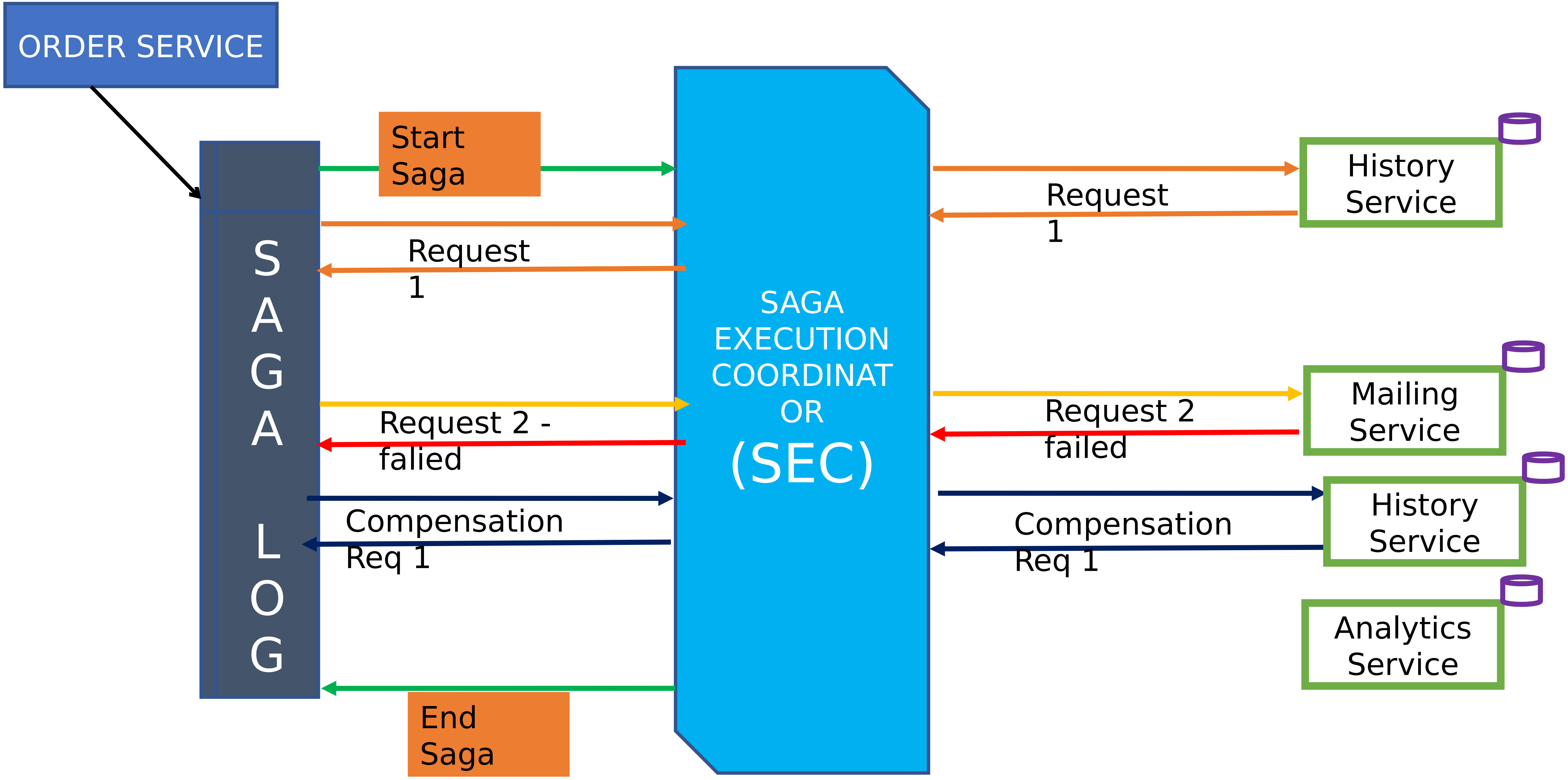


Orchestration

- ***Benefits***
  - ***good for complex workflows which have many services and new service can be added at any time***
  - ***Suitable when there is control over all participants and control over the flow of activities***
  - ***No Cyclic dependency risk***
  - ***Clear separation of concern, separate from business logic***
- ***Drawbacks***
  - ***Additional design complexity***
  - ***Additional point of failure***




ORDER SERVICE
SAGA LOG
Start Saga
Request 1
Request 2
Request 3
End Saga
SAGA EXECUTION COORDINATOR (SEC)
Request 1
Request 2
Request 3
History Service
Mailing Service
Analytics Service



ORDER SERVICE

SAGA LOG

Start Saga

Request 1

Request 2 - falied

Compensation Req 1

End Saga

SAGA EXECUTION COORDINATOR (SEC)

Request 1

Request 2 failed

Compensation Req 1

History Service

Mailing Service

History Service

Analytics Service

When to use saga?

- ***When you need to***
  - Ensure data consistency in a distributed system without tight coupling.
  - Rollback if one of the operations in the sequence fails.

- ***Less suitable for***
  - Tightly coupled transaction
  - Compensating transactions that occur in earlier participants.
  - Cyclic dependencies.

https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga

What next??

Database Pattern : Summary

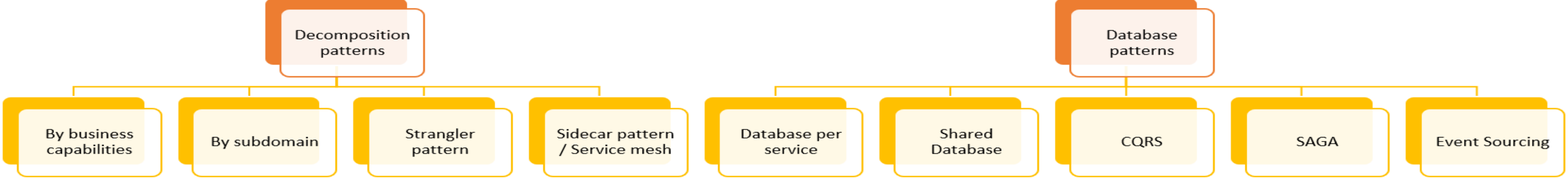
MICROSERVICES DESIGN PATTERNS

Database Patterns : Summary

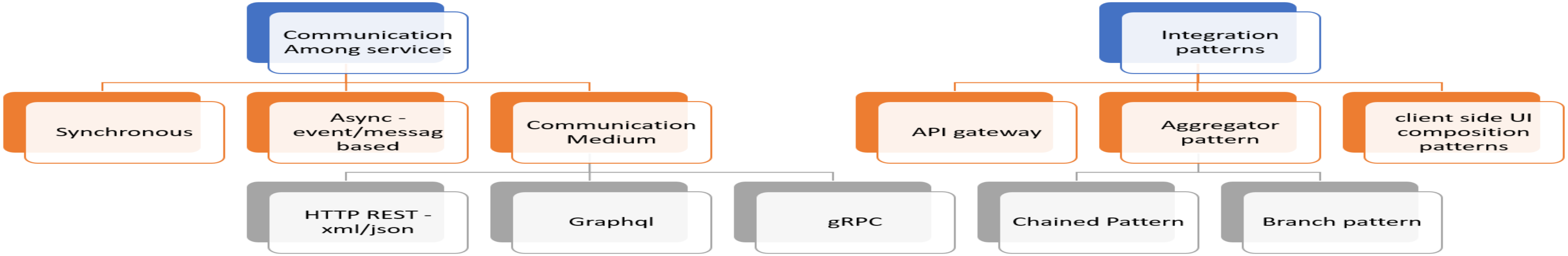
Microservices Architecture

Database Patterns for microservices

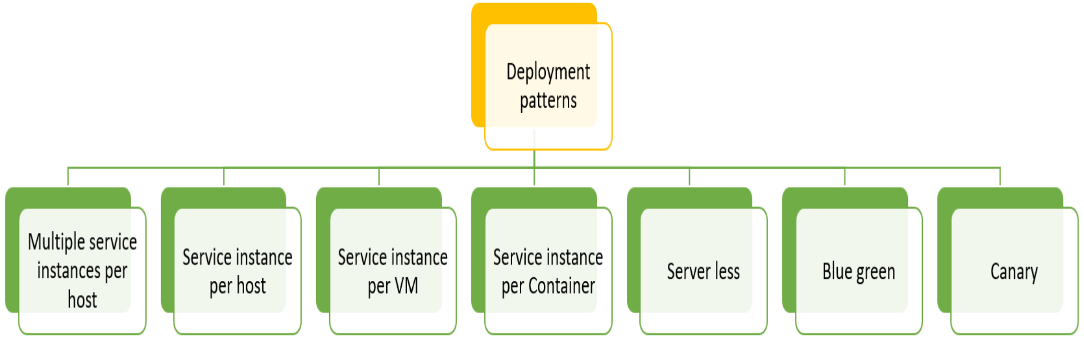
- Database per service
- Shared Database per service
- CQRS
- Data consistency – Eventual vs Strong
- Event Driven
- Event sourcing
- 2 Phase Commit
- Saga



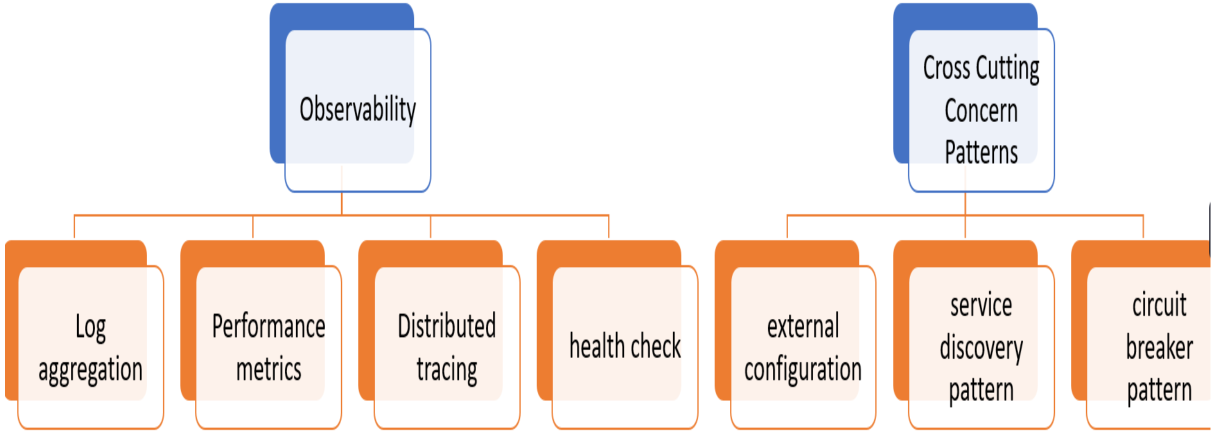
- Decomposition patterns
  - By business capabilities
  - By subdomain
  - Strangler pattern
  - Sidecar pattern / Service mesh
- Database patterns
  - Database per service
  - Shared Database
  - CQRS
  - SAGA
  - Event Sourcing



- Communication Among services
  - Synchronous
  - Async - event/messag based
  - Communication Medium
    - HTTP REST - xml/json
    - Graphql
    - gRPC
- Integration patterns
  - API gateway
  - Aggregator pattern
    - Chained Pattern
    - Branch pattern
  - client side UI composition patterns



- Deployment patterns
  - Multiple service instances per host
  - Service instance per host
  - Service instance per VM
  - Service instance per Container
  - Server less
  - Blue green
  - Canary



- Observability
  - Log aggregation
  - Performance metrics
  - Distributed tracing
  - health check
- Cross Cutting Concern Patterns
  - external configuration
  - service discovery pattern
  - circuit breaker pattern

/hat next??

Communication : Synchronous vs Asynchronous

MICROSERVICES DESIGN PATTERNS

How Microservices **Communicate** with Each Other??

Microservices Architecture

Online Shopping Portal : **Monolithic**

Product Module

listProduct(productId){
//validateUser()
//check For Products
//
notifyMerchant(productId)
//return Products
}

Login Module

validateUser(){
//validate user credentials
//return validation status
}

Payment Module

makePayment(){
//validateUser()
//process the payment
//return response
}

Order Module

placeOrder(cartId){
//validateUser()
//checkProductAvailability()
//makePayment()
//placeOrder
//notifyUser()
//return response
}

Mailing Module

notifyUser() {
//mail/sms to the user
about order
}

Online Shopping Portal : **Microservices**

Product Service

```
listProduct(productId){
  //validateUser()
  //check For Products
  //
  notifyMerchant(productId)
  //return Products
}
```

Payment Service

```
makePayment(){
  //validateUser()
  //process the payment
  //return response
}
```

Login Service

```
validateUser(){
  //validate user credentials
  //return validation status
}
```

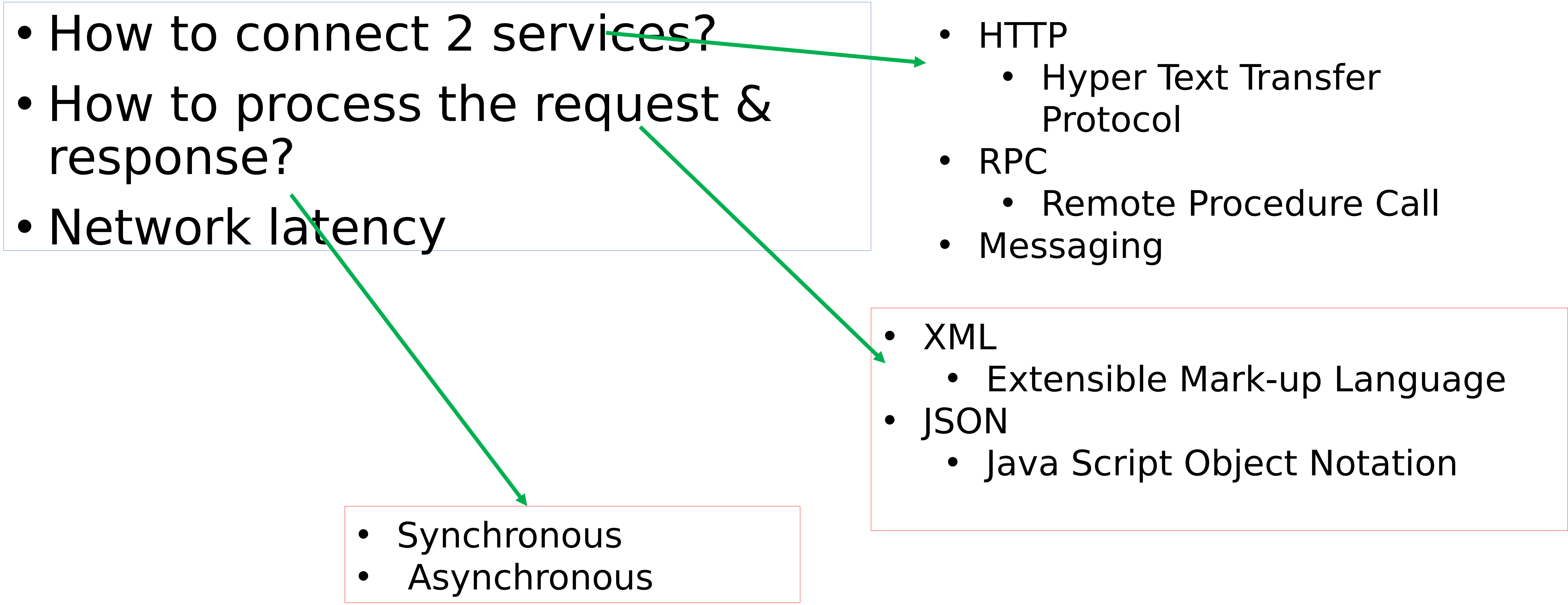
Order Service

```
placeOrder(cartId){
  //validateUser()
  //checkProductAvailability()
  //makePayment()
  //placeOrder
  //notifyUser()
  //return response
}
```

Mailing Service

```
notifyUser() {
  //mail/sms to the user
  about order
}
```

Challenges



- How to connect 2 services?
- How to process the request & response?
- Network latency

- HTTP
  - Hyper Text Transfer Protocol
- RPC
  - Remote Procedure Call
- Messaging

- XML
  - Extensible Mark-up Language
- JSON
  - Java Script Object Notation

- Synchronous
- Asynchronous

/hat next??

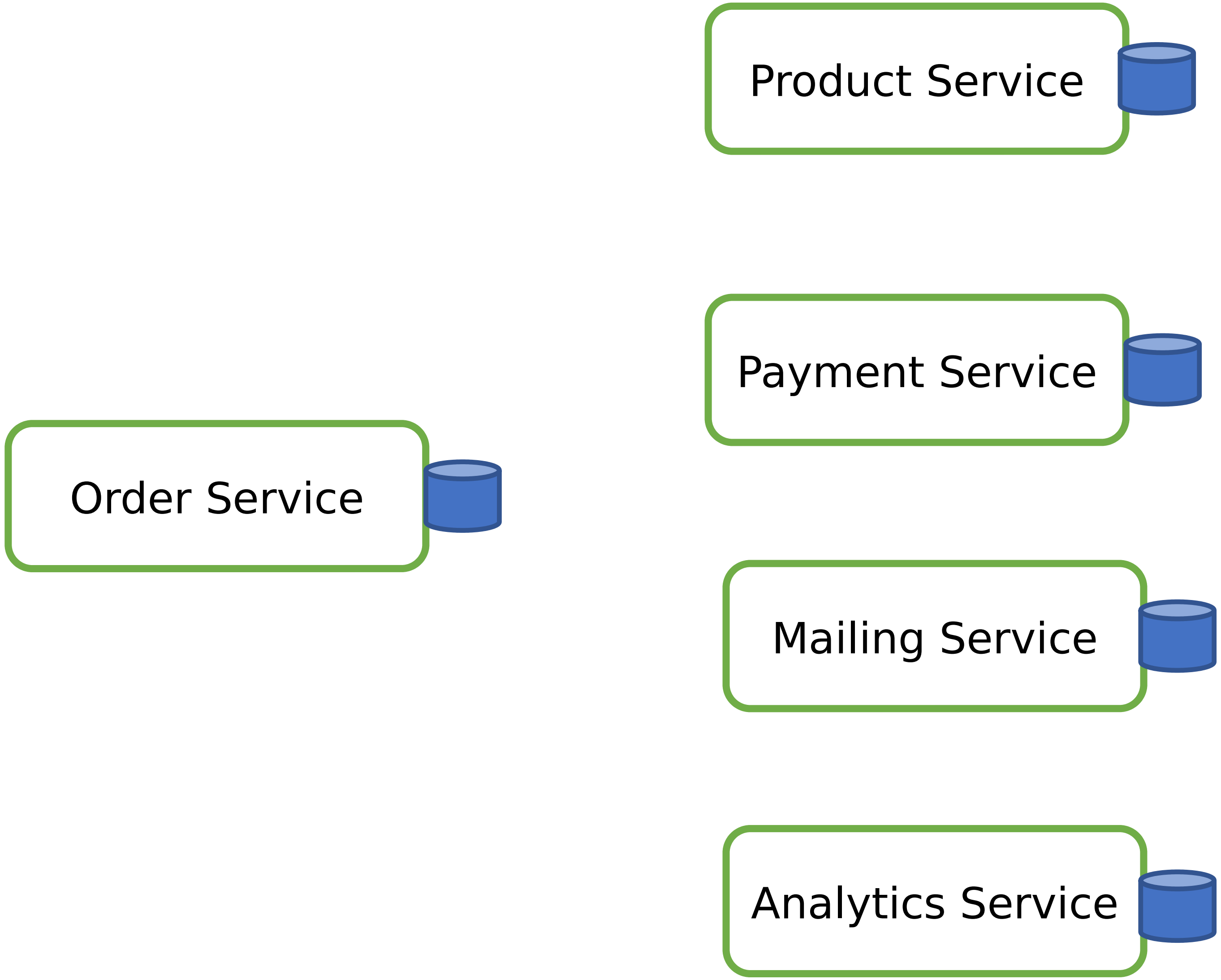
Synchronous vs Asynchronous

MICROSERVICES DESIGN PATTERNS

Synchronous vs Asynchronous Communication

Microservices Architecture

Synchronous Communication



Order Service

Product Service

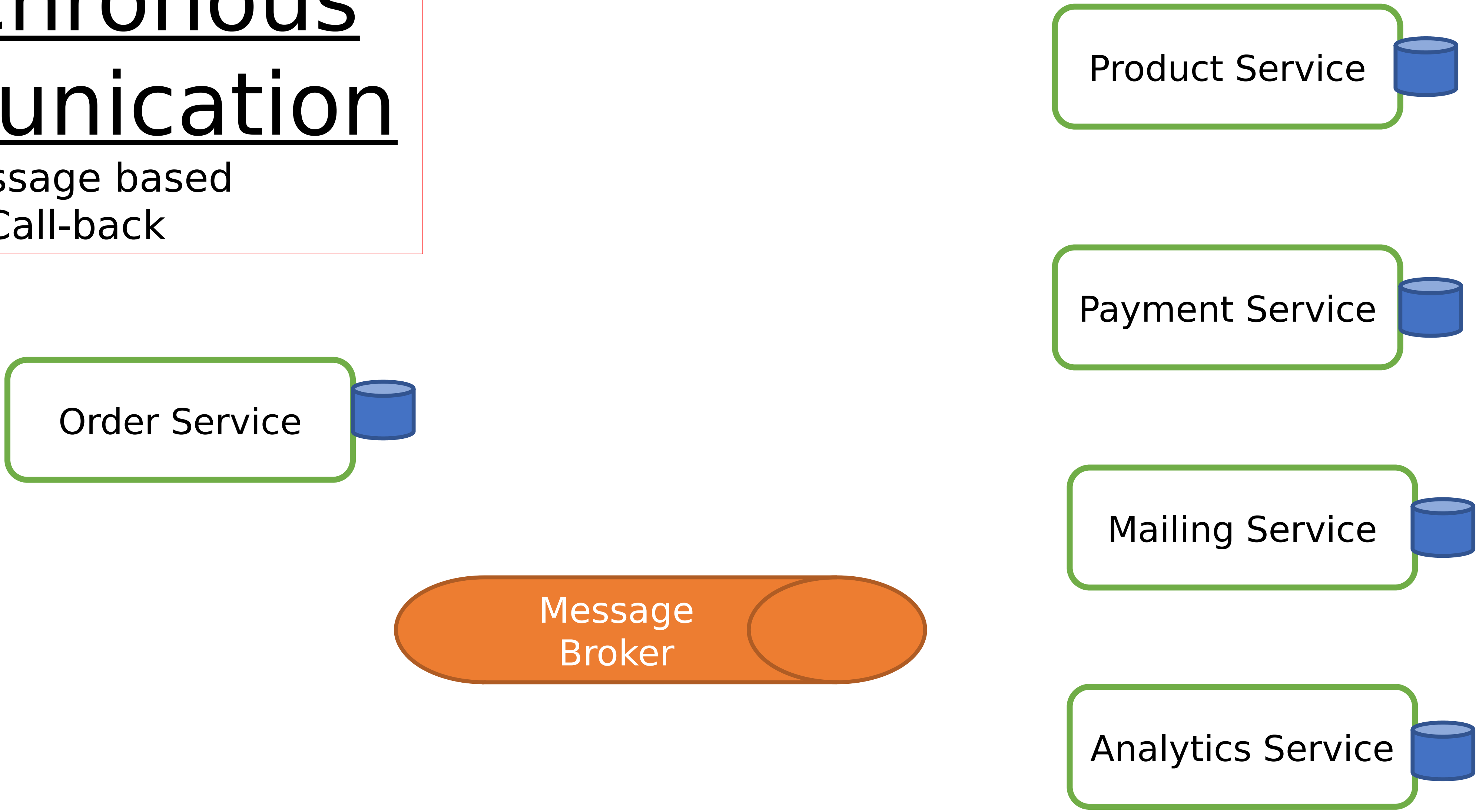
Payment Service

Mailing Service

Analytics Service

Asynchronous Communication

- Message based
  - Call-back



Synchronous

- Easy to implement
- Easy to test
- Easy to debug
- Blocking
- Slow due to waiting
- High coupling
-

Asynchronous

- Difficult due to message broker
- Difficult to test
- Difficult to debug
- Non-blocking
- Fast
- Loose coupling
- Not Reliable due to no-response

Which one should we use??

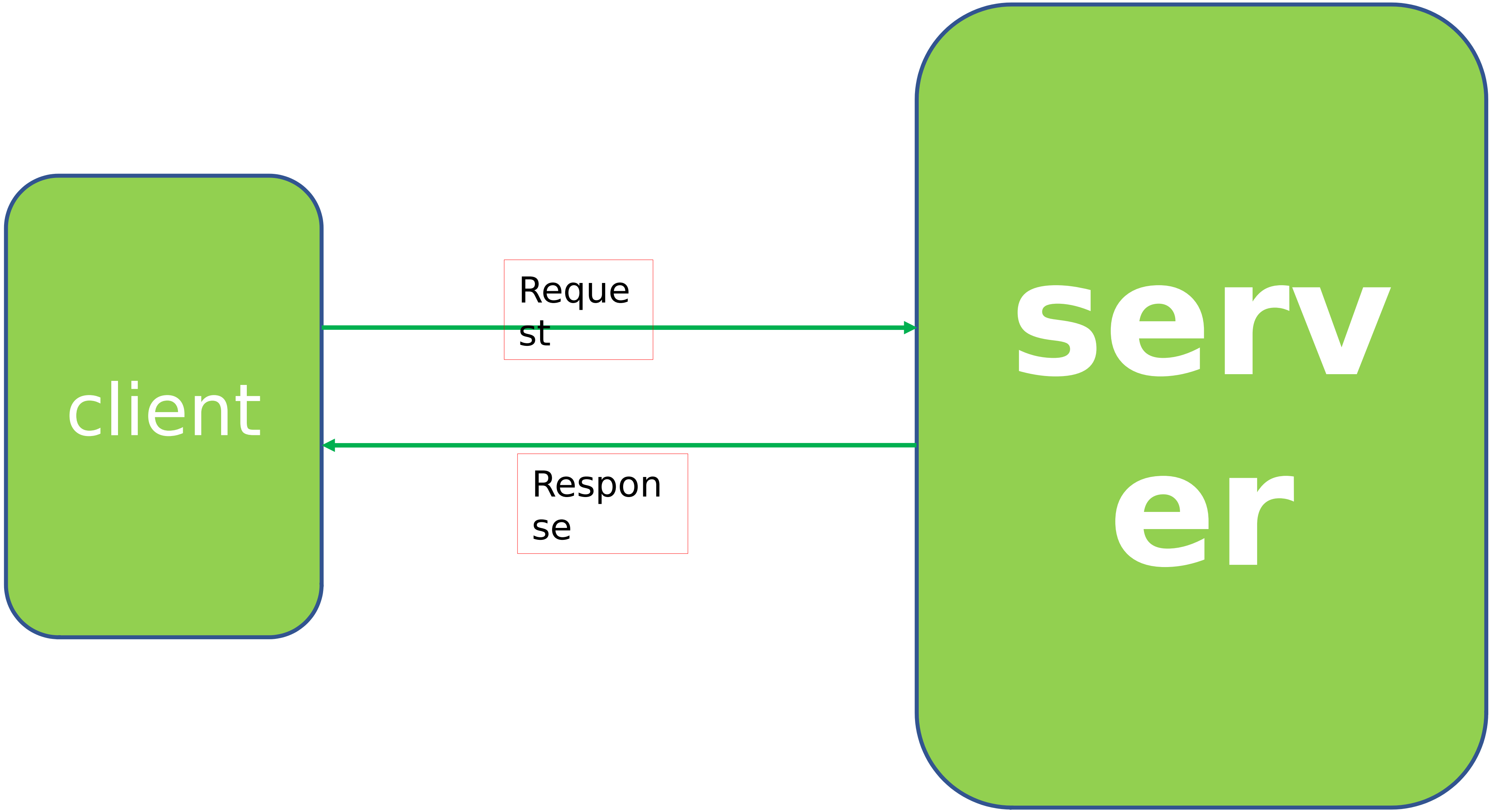
What next??

All about HTTP

MICROSERVICES DESIGN PATTERNS

How to setup Synchronous Communication among microservices? **HTTP**

Microservices Architecture



client

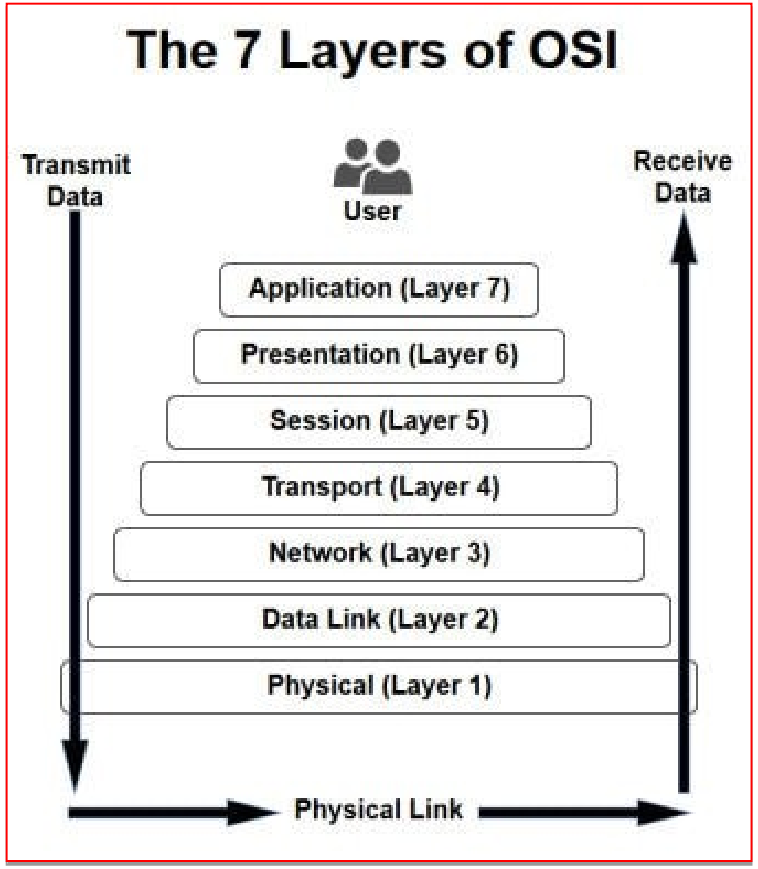
Reque
st

Respon
se

serv
er

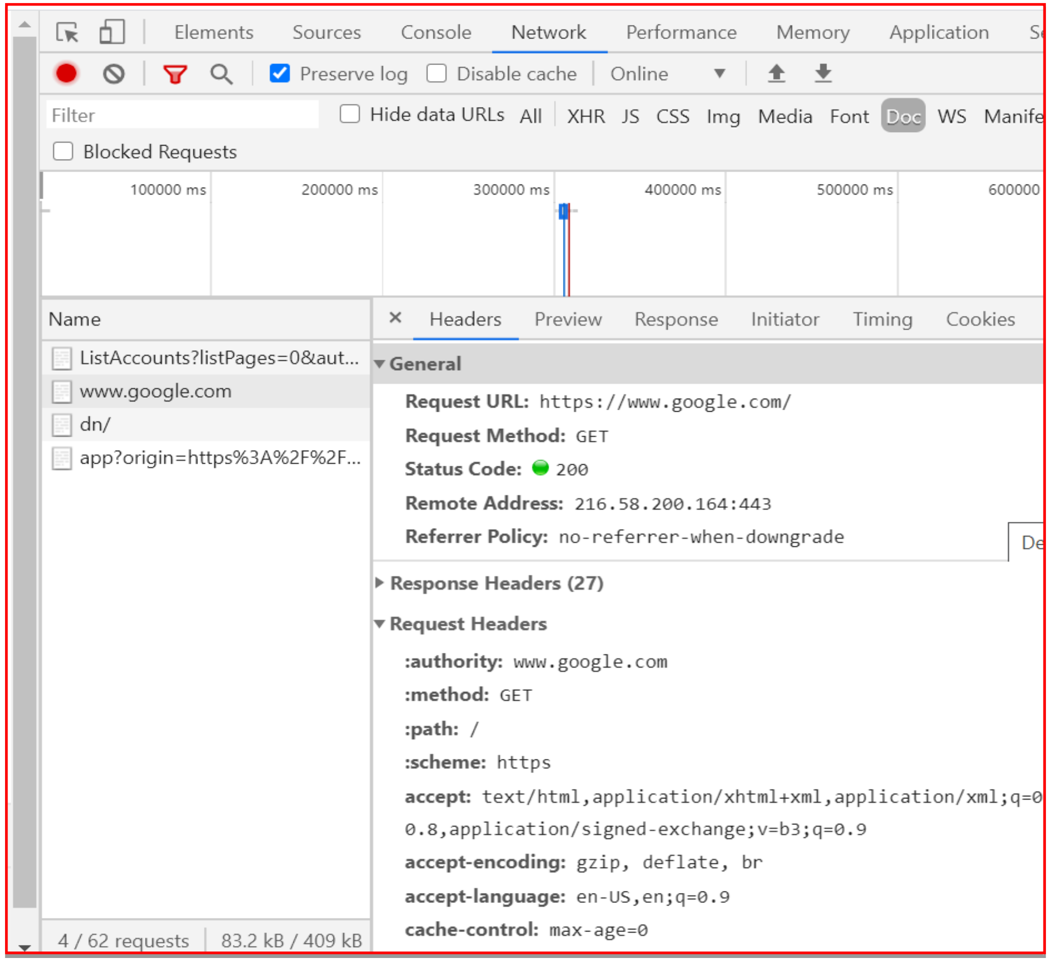
HTTP : Hyper Text Transfer Protocol

- Communication protocol that transports messages from one place to other over a network
- Stateless
  - State is maintained using cookies or session
-



Http Request Format

- URL
- Method
  - GET
  - POST
  - PUT
  - Delete etc..
- Headers
-



▼ **General**

Request URL: `https://www.google.com/`

Request Method: `GET`

Status Code: ● `200`

Remote Address: `216.58.200.164:443`

Referrer Policy: `no-referrer-when-downgrade`

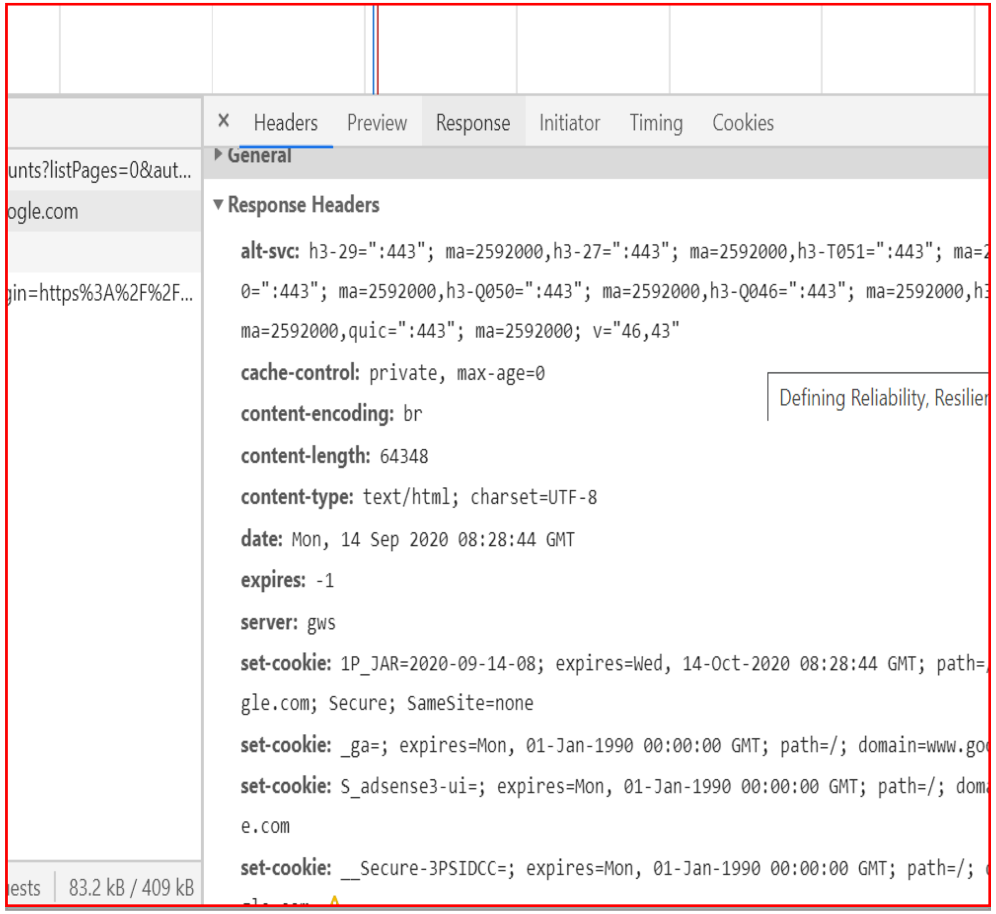
▶ **Response Headers (27)**

▼ **Request Headers**

```
:authority: www.google.com
:method: GET
:path: /
:scheme: https
accept: text/html,application/xhtml+xml,application/xml;q=0
0.8,application/signed-exchange;v=b3;q=0.9
accept-encoding: gzip, deflate, br
accept-language: en-US,en;q=0.9
cache-control: max-age=0
```

Http Response Format

- http status code
  - Informational responses (100–199),
  - Successful responses (200–299),
  - Redirects (300–399),
  - Client errors (400–499),
  - and Server errors (500–599).
- Headers
- Body[optional]



▸ General

▾ Response Headers

```
alt-svc: h3-29=":443"; ma=2592000,h3-27=":443"; ma=2592000,h3-T051=":443"; ma=2
0=":443"; ma=2592000,h3-Q050=":443"; ma=2592000,h3-Q046=":443"; ma=2592000,h3
ma=2592000,quic=":443"; ma=2592000; v="46,43"
cache-control: private, max-age=0
content-encoding: br
content-length: 64348
content-type: text/html; charset=UTF-8
date: Mon, 14 Sep 2020 08:28:44 GMT
expires: -1
server: gws
set-cookie: 1P_JAR=2020-09-14-08; expires=Wed, 14-Oct-2020 08:28:44 GMT; path=/
gle.com; Secure; SameSite=none
set-cookie: _ga=; expires=Mon, 01-Jan-1990 00:00:00 GMT; path=/; domain=www.goo
set-cookie: S_adsense3-ui=; expires=Mon, 01-Jan-1990 00:00:00 GMT; path=/; doma
e.com
set-cookie: __Secure-3PSIDCC=; expires=Mon, 01-Jan-1990 00:00:00 GMT; path=/; d
```

Online Shopping Portal : **Microservices**

Product Service

listProduct(productId){
//validateUser()
//check For Products
//
notifyMerchant(productId)
//return Products
}

Payment Service

makePayment(){
//validateUser()
//process the payment
//return response
}

Login Service

validateUser(){
//validate user credentials
//return validation status
}

Order Service

placeOrder(cartId){
//validateUser()
//checkProductAvailability()
//makePayment()
//placeOrder
//notifyUser()
//return response
}

Mailing Service

notifyUser() {
//mail/sms to the user
about order
}

REST – Representational State Transfer

RESTfull API

Microservice with Spring Boot

What next??

Message based communication

MICROSERVICES DESIGN PATTERNS

How to setup Asynchronous communication :

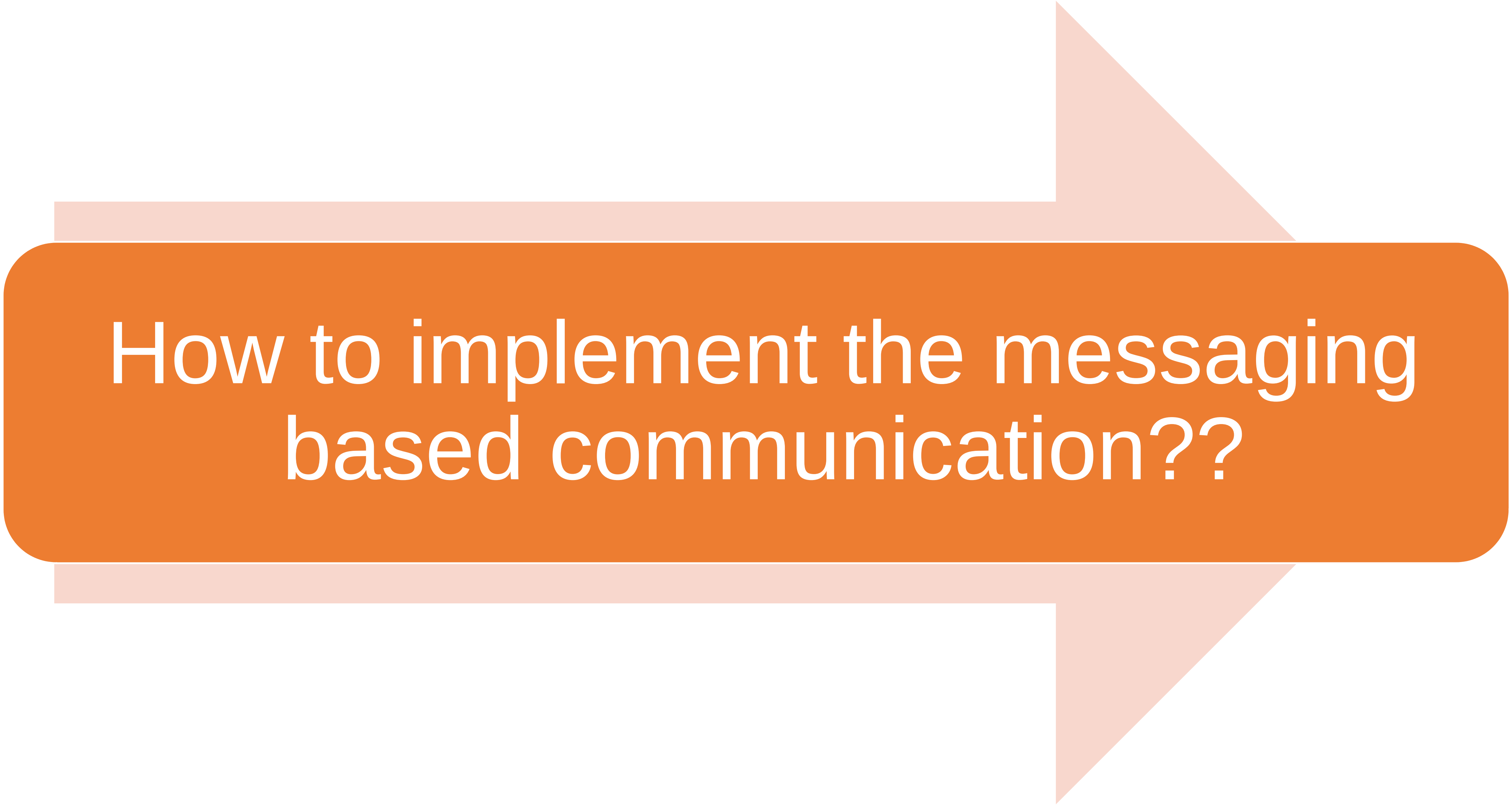
Message Based

Microservices

Architecture

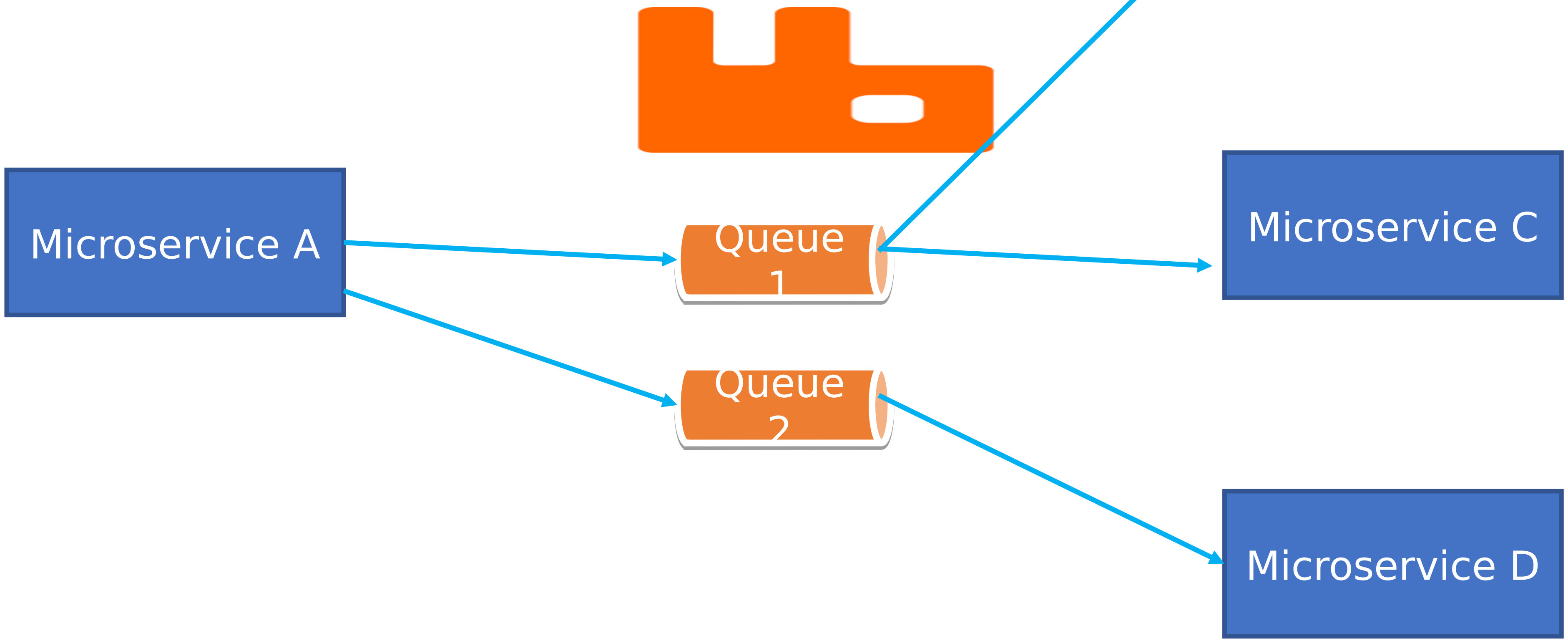
Different approaches for message based communication

- Notifications
- Request/asynchronous response
- Publish/subscribe
-



How to implement the messaging based communication??

RabbitMQ



Microservice A

Queue 1

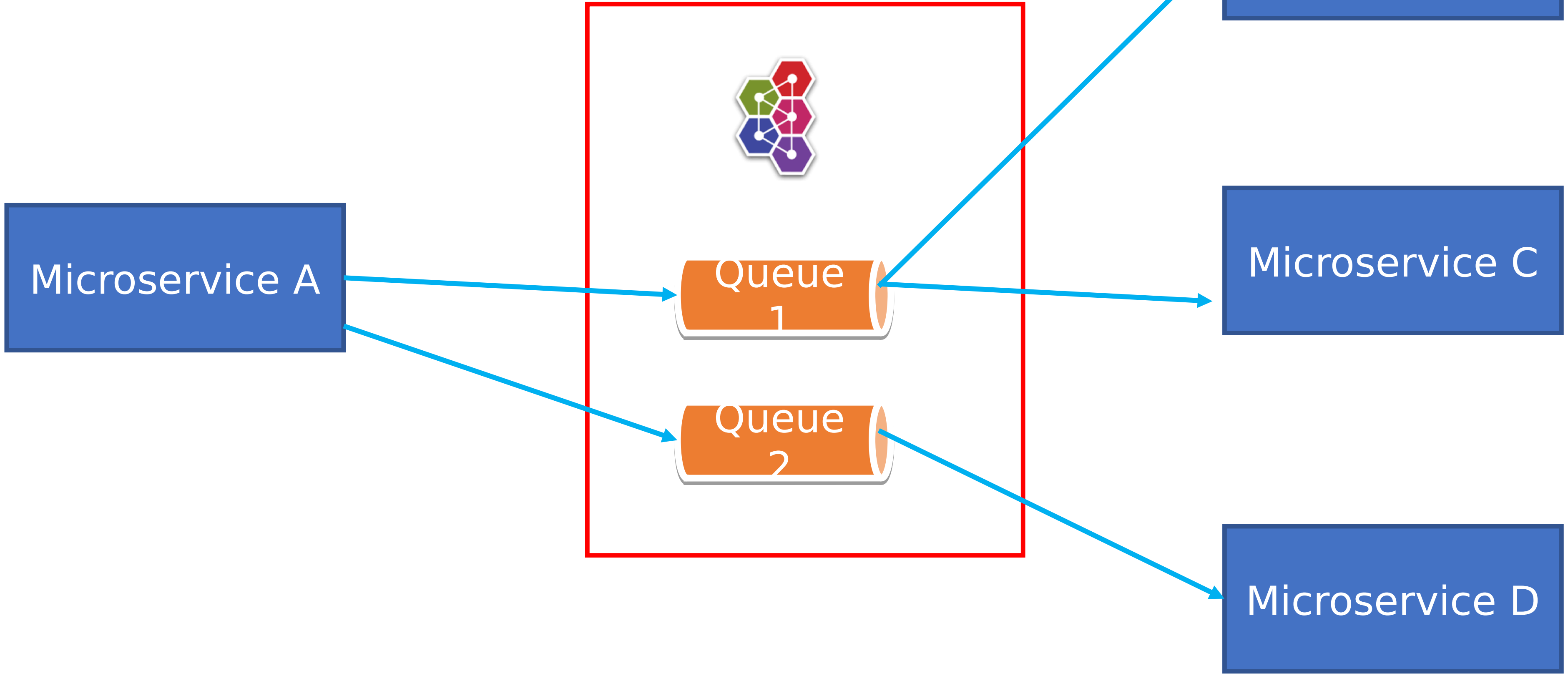
Queue 2

Microservice B

Microservice C

Microservice D

ActiveMQ



Microservice A

Queue 1

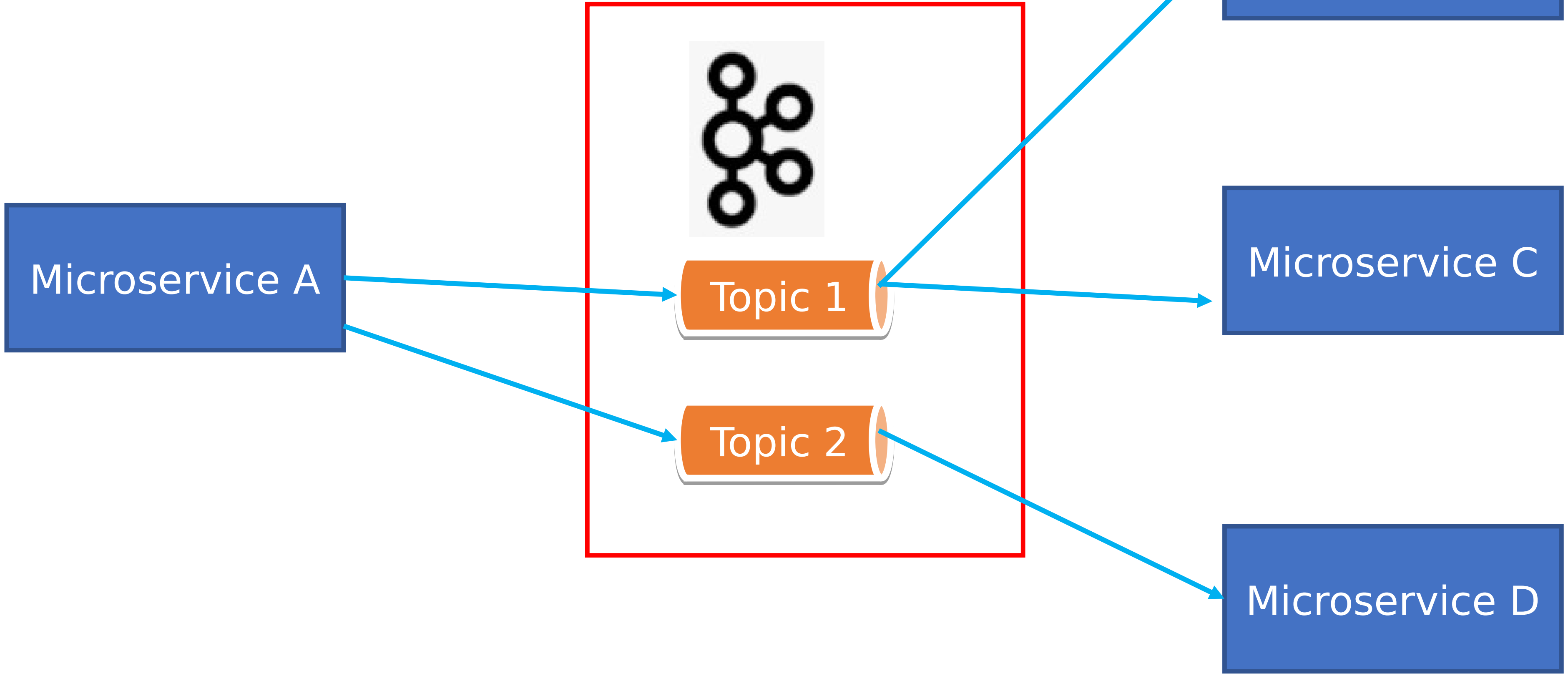
Queue 2

Microservice B

Microservice C

Microservice D

APACHE kafka®



Microservice A

Topic 1

Topic 2

Microservice B

Microservice C

Microservice D

What next??

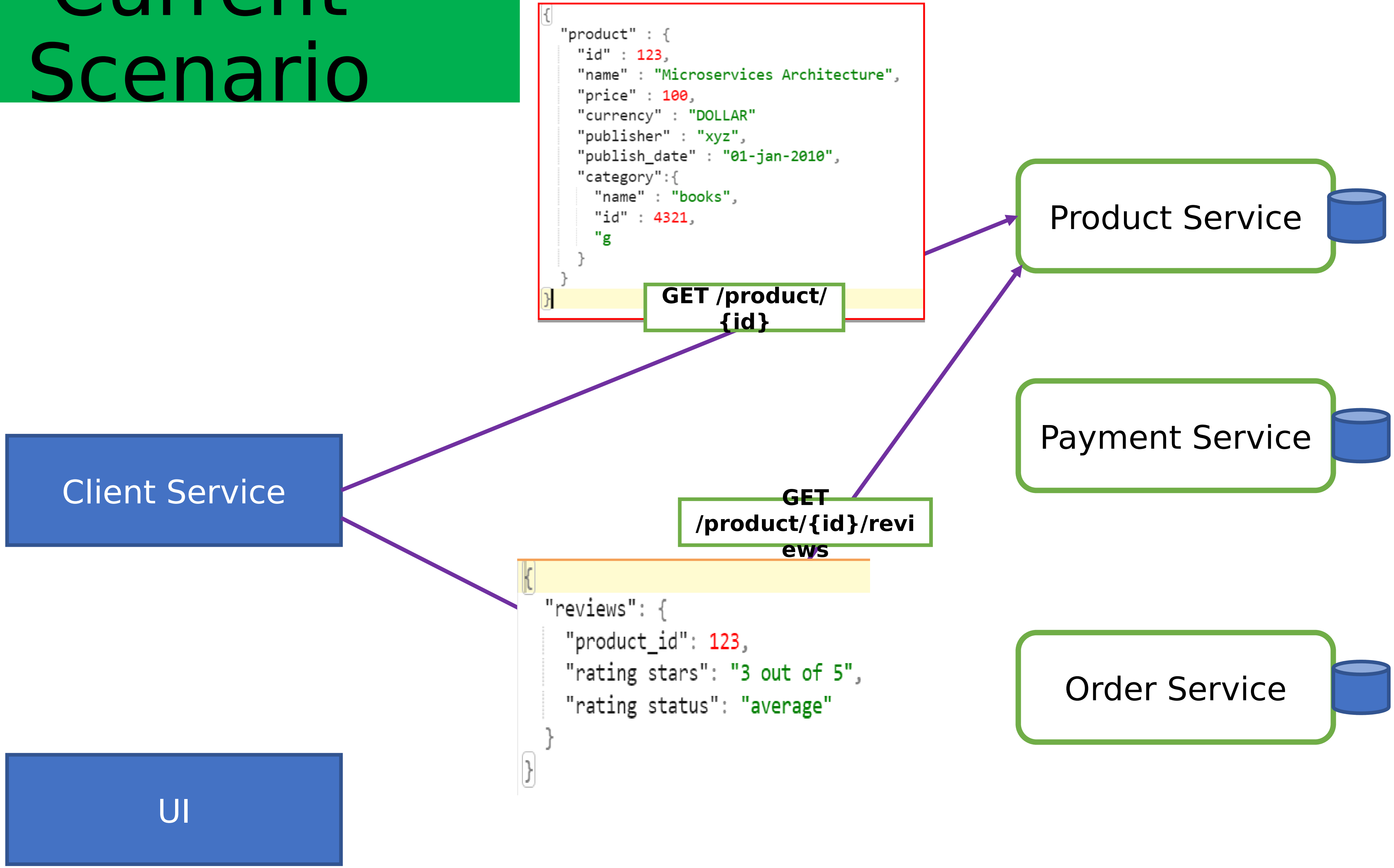
GraphQL

MICROSERVICES DESIGN PATTERNS

What is **GraphQL?**

Microservices Architecture

Current Scenario



```
{
  "product" : {
    "id" : 123,
    "name" : "Microservices Architecture",
    "price" : 100,
    "currency" : "DOLLAR"
    "publisher" : "xyz",
    "publish_date" : "01-jan-2010",
    "category":{
      "name" : "books",
      "id" : 4321,
      "g
    }
  }
}
```

```
{
  "reviews": {
    "product_id": 123,
    "rating stars": "3 out of 5",
    "rating status": "average"
  }
}
```

Problem with current scenario??

- Overfetching
- Underfetching

GraphQL

GraphQL is the better REST

Query language for your API

Strong type system to define the capabilities of the API

An schema serves as contract between client and server

/hat next??

Microserivces Communication patterns: Summary

MICROSERVICES DESIGN PATTERNS

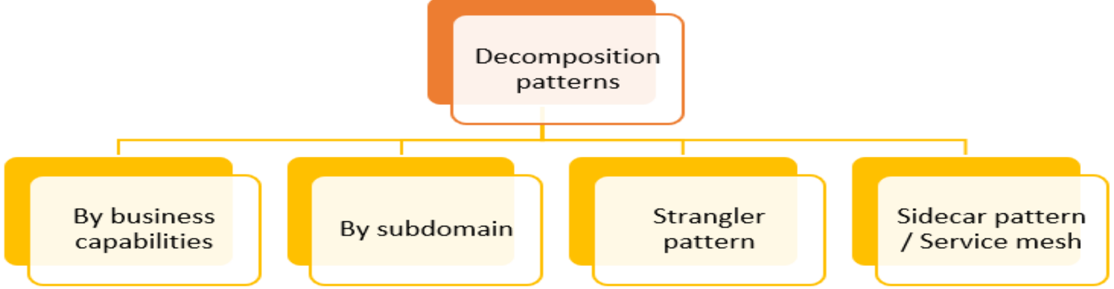
Microserivces Communication Patterns

Summary

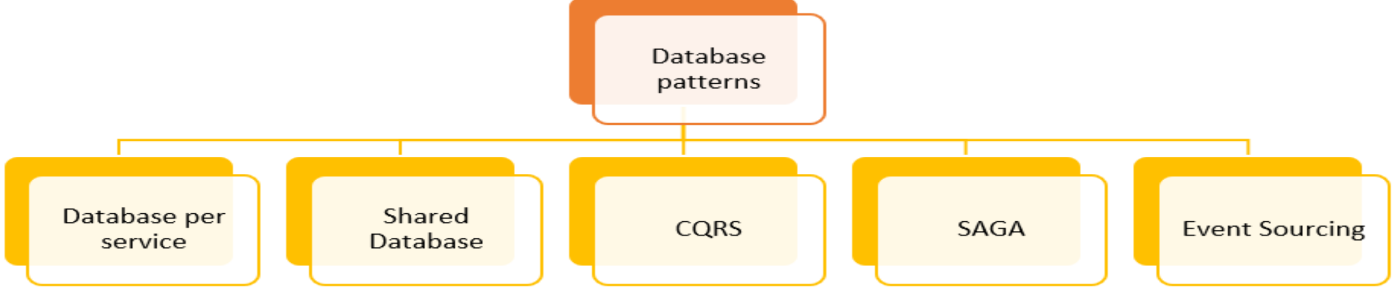
Microservices Architecture

Summary

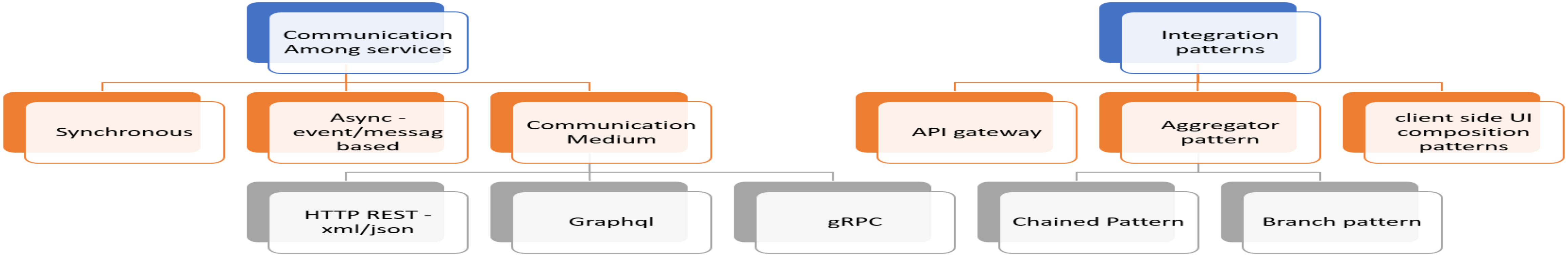
- How microservices talk to each other?
- Synchronous vs Asynchronous communication
- How to setup synchronous communication
- How to setup Asynchronous communication – Message Bases
- REST API
-



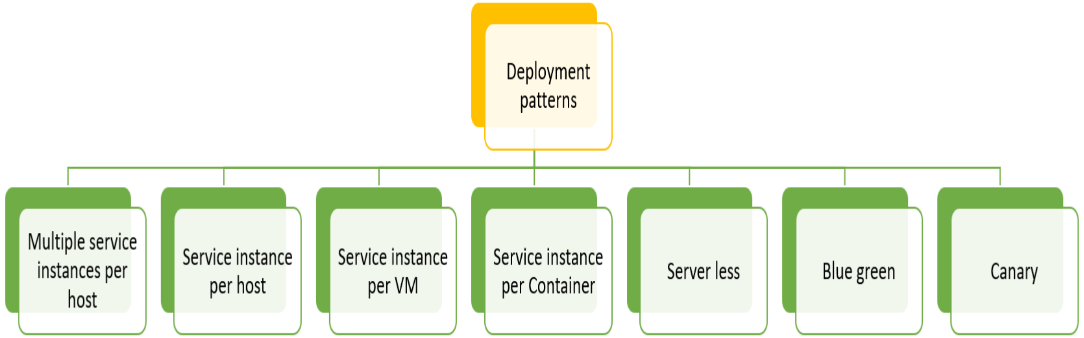
- Decomposition patterns
  - By business capabilities
  - By subdomain
  - Strangler pattern
  - Sidecar pattern / Service mesh



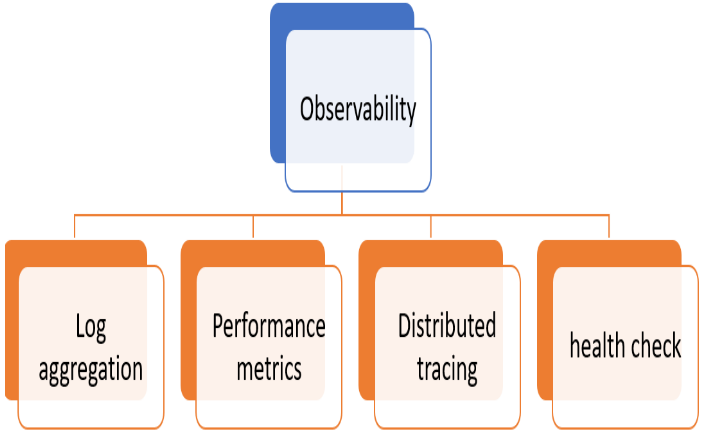
- Database patterns
  - Database per service
  - Shared Database
  - CQRS
  - SAGA
  - Event Sourcing



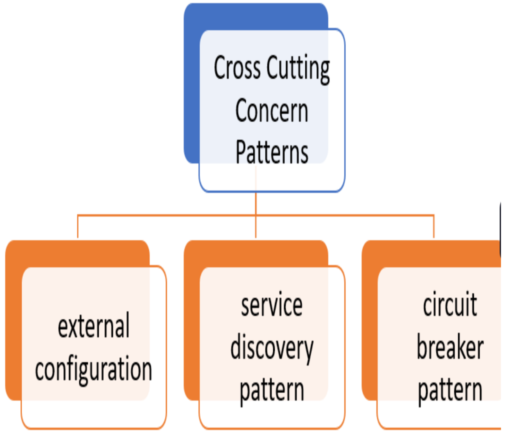
- Communication Among services
  - Synchronous
  - Async - event/messag based
  - Communication Medium
    - HTTP REST - xml/json
    - Graphql
    - gRPC
- Integration patterns
  - API gateway
  - Aggregator pattern
    - Chained Pattern
    - Branch pattern
  - client side UI composition patterns



- Deployment patterns
  - Multiple service instances per host
  - Service instance per host
  - Service instance per VM
  - Service instance per Container
  - Server less
  - Blue green
  - Canary



- Observability
  - Log aggregation
  - Performance metrics
  - Distributed tracing
  - health check



- Cross Cutting Concern Patterns
  - external configuration
  - service discovery pattern
  - circuit breaker pattern

/hat next??

Microservices Integration Patterns:

API gateway

MICROSERVICES DESIGN PATTERNS

Microservices Integration Patterns:

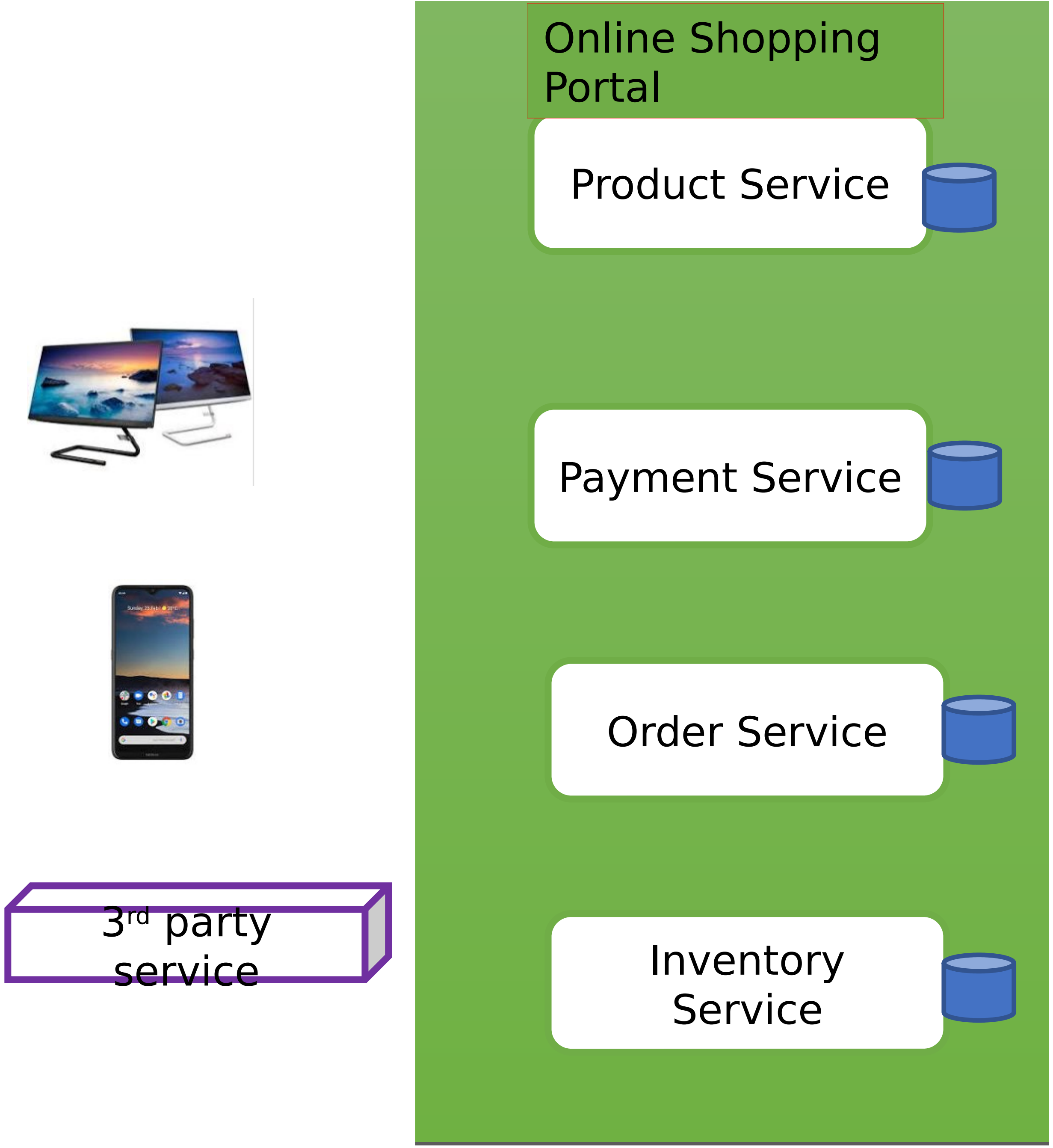
API Gateway

Microservices Architecture

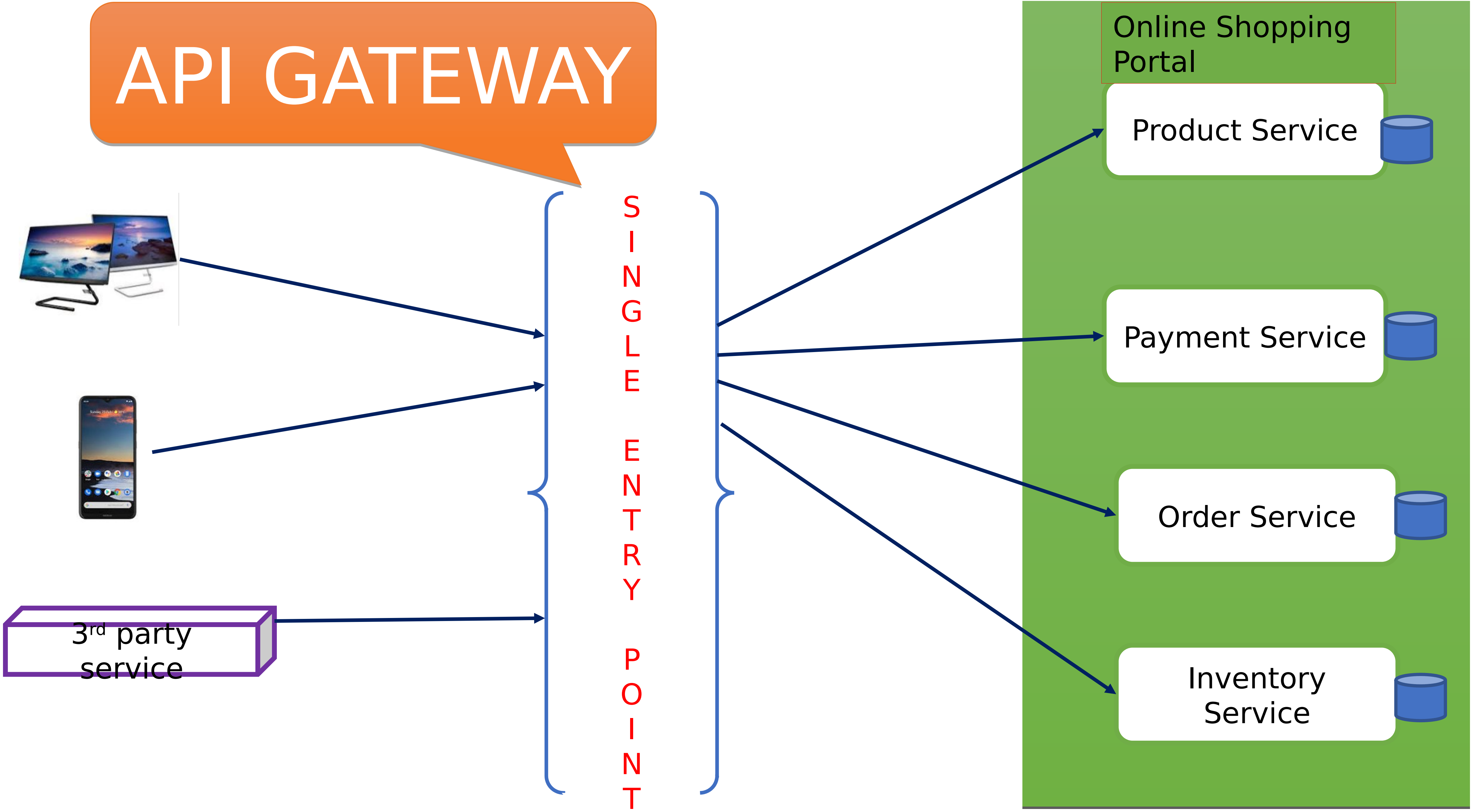
How do the **clients** of microservices based application **access** the **individual microservice**??

Direct Communication

- Aggregating data from multiple services
- Too much chatty behaviour between clients and services
- Non-web friendly protocol -> AMQP
- How to handle the change in location(host + port) of services
- Cross cutting concerns like
  - Security (Authentication & Authorization)
  - Logging, tracing
  - Load balancing
  - Caching
  - IP whitelisting
  - Request/response transformations
  - Failure handling – circuit breaker
- Addressing the needs of different clients like desktop, mobile or any other service
  - High coupling for each client



API GATEWAY



Online Shopping Portal

Product Service

Payment Service

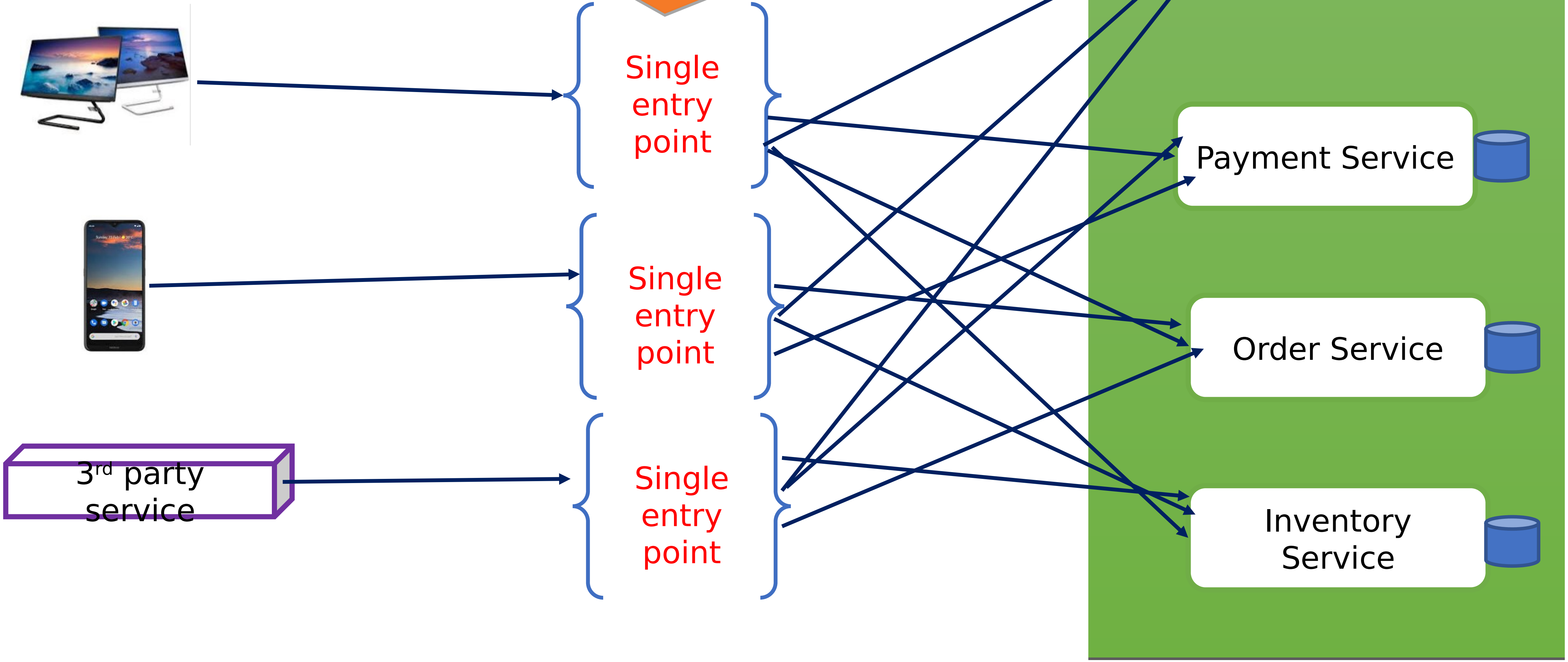
Order Service

Inventory Service

3rd party service

SINGLE ENTRY POINT

API GATEWAY(Backend for frontend-BFF)



3rd party service

Single entry point

Single entry point

Single entry point

Online Shopping Portal

Product Service

Payment Service

Order Service

Inventory Service

API Gateway

- Aggregating data from multiple services
  - Reduces the request roundtrips
- Non-web friendly protocol
  - Client is able to use single standard protocol to communicate to API gateway and independent of what protocol is used in specific service
- How to handle the change in location(host + port) of services
  - Not the headache of client
- Cross cutting concerns like (All of these are now responsibility of API gateway, centralized/clean/standard code)
  - Security(Authentication & Authorization)
  - Logging, tracing
  - Load balancing
  - Caching
  - IP whitelisting
  - Request/response transformations
  - Failure handling – circuit breaker
- Addressing the needs of different clients like desktop, mobile or any other service
  - High coupling for each client
  - Now each client has it’s own optimal API gateway

Few Drawbacks

- Extra application
- Increased complexity of overall application
-

Implementations

- Write your own API gateway
  - Netflix Zuul - https://www.youtube.com/playlist?list=PLq3uEqRnr_2GleAdJYmlBkB_RfbjMGdoH
  - Spring cloud gateway – Coming Soon
- 3rd Party providers
  - Kong - https://konghq.com/kong
  - Apigee – https://cloud.google.com/apigee
  - Amazon API Gateway - https://aws.amazon.com/api-gateway https://aws.amazon.com/api-gateway

What next??

Microservices Integration Patterns: Aggregator

MICROSERVICES DESIGN PATTERNS

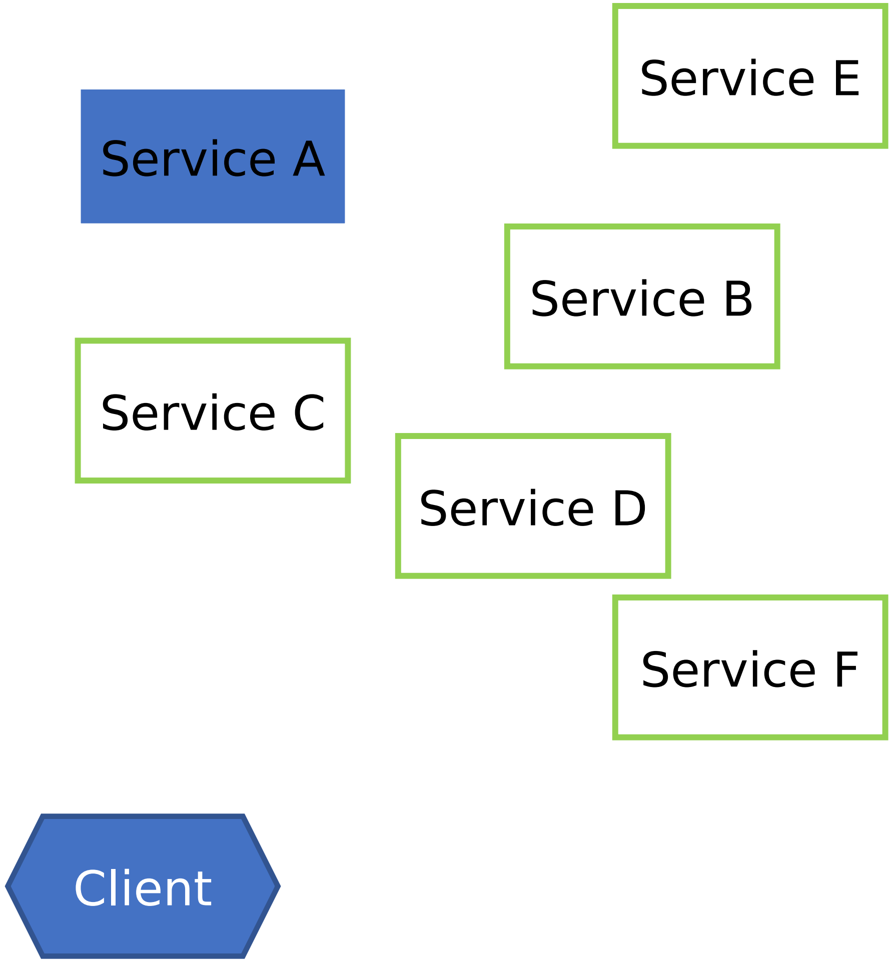
Microservices Integration Patterns:

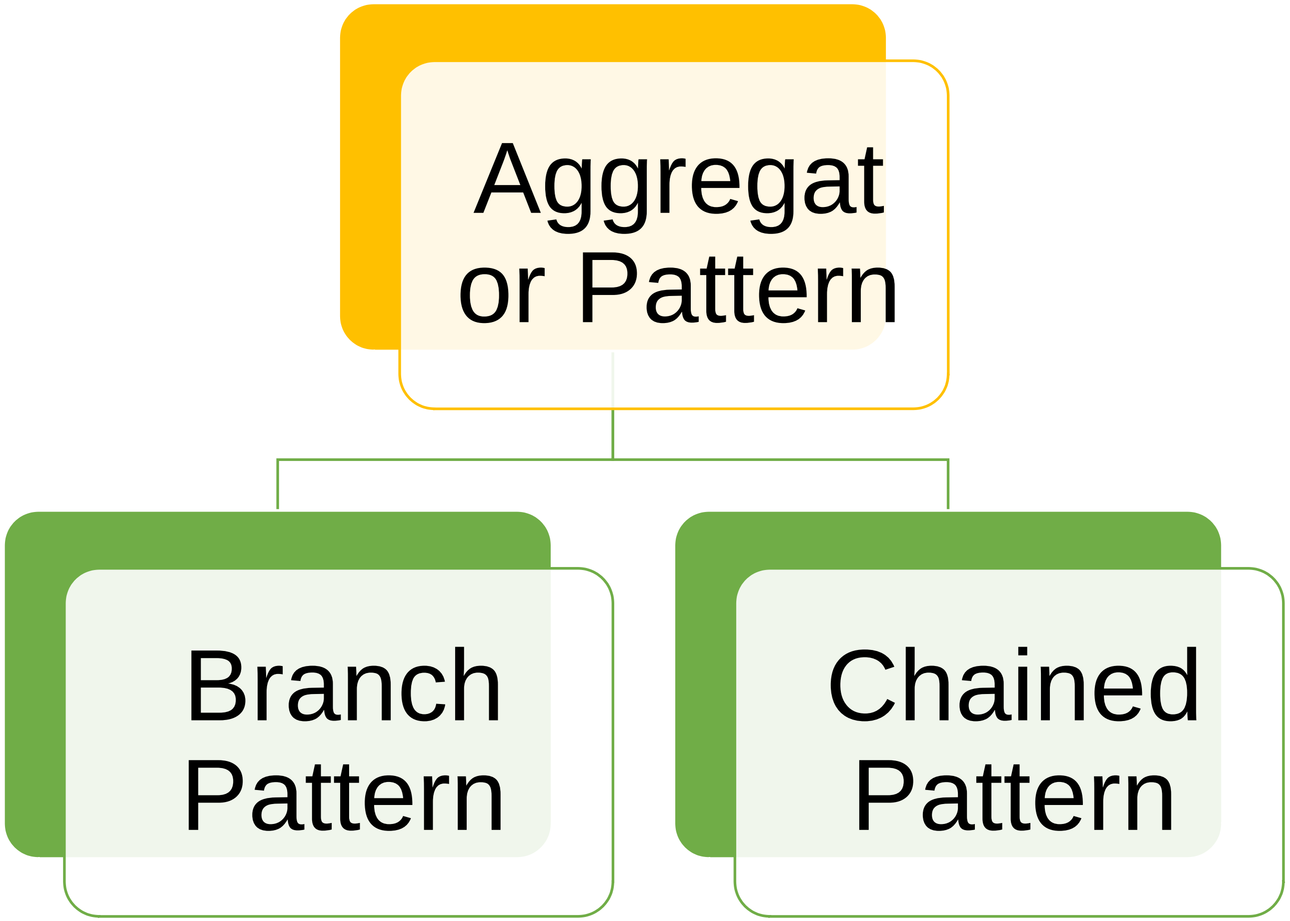
Aggregator

Microservices Architecture

What is aggregator pattern

- Collaborating data returned by each service
- Composite microservice
  - Calling all the services needed to make the response
  - Transform the response as per clients need
  - Return back to client
- API gateway can also do the aggregator job



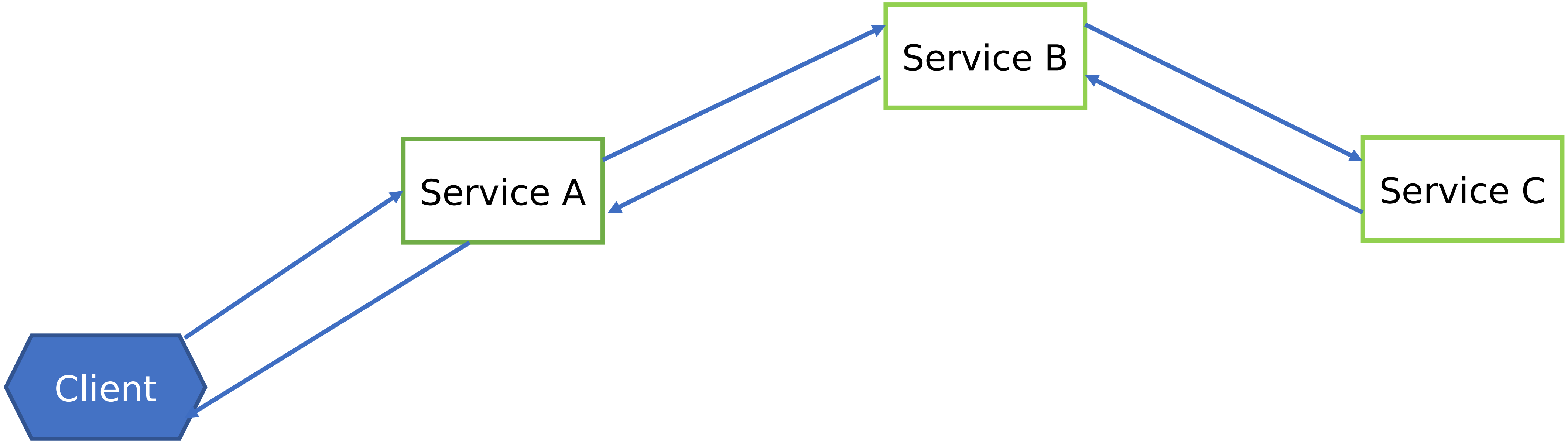


Aggregator Pattern

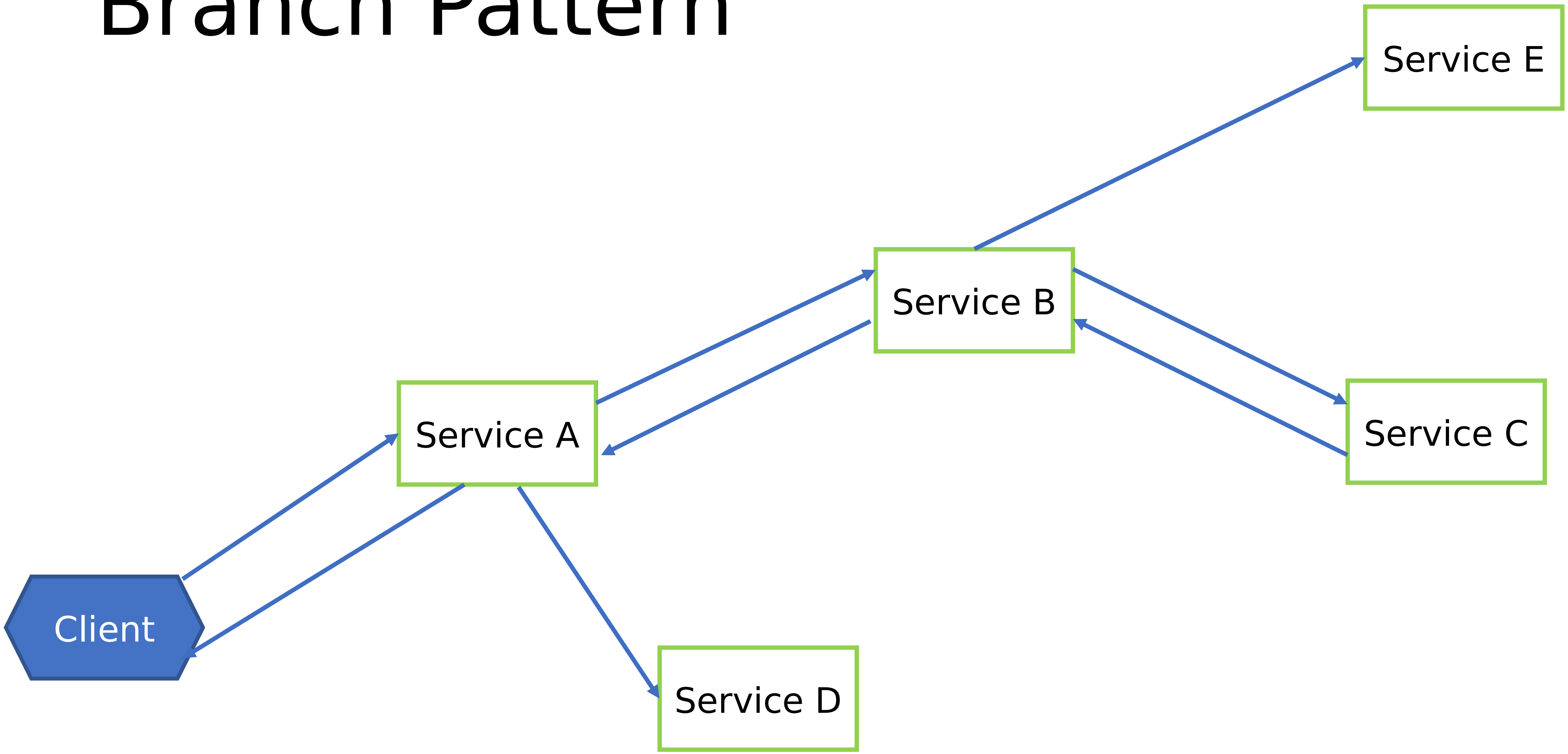
Branch Pattern

Chained Pattern

Chained Pattern



Branch Pattern



What next??

Microservices Integration Patterns: Client Side UI Composition

MICROSERVICES DESIGN PATTERNS

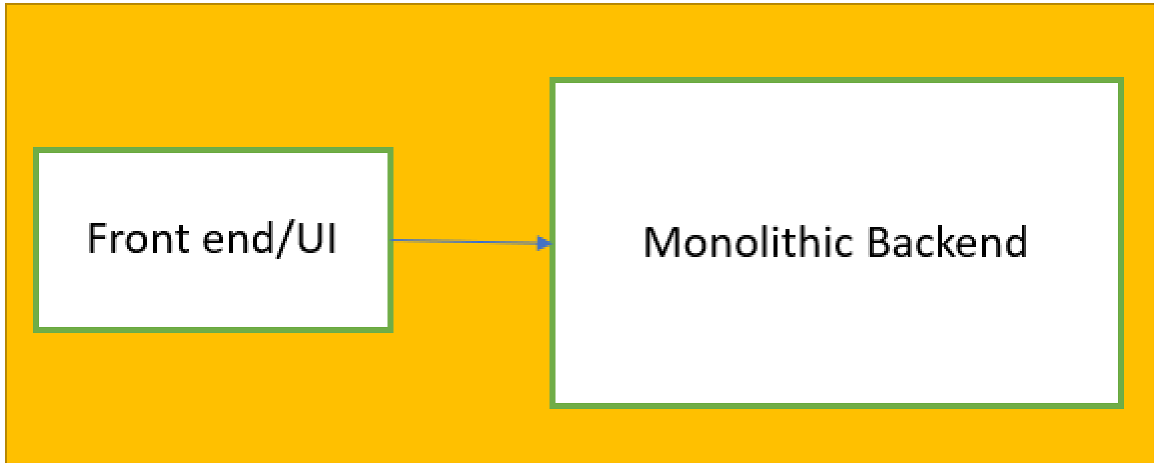
Microservices Integration Patterns:

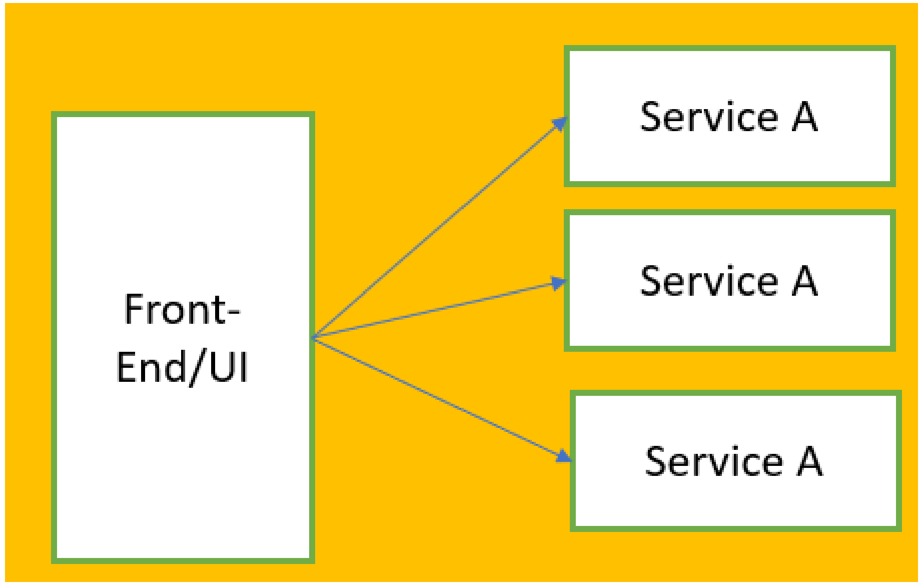
Client Side UI Composition

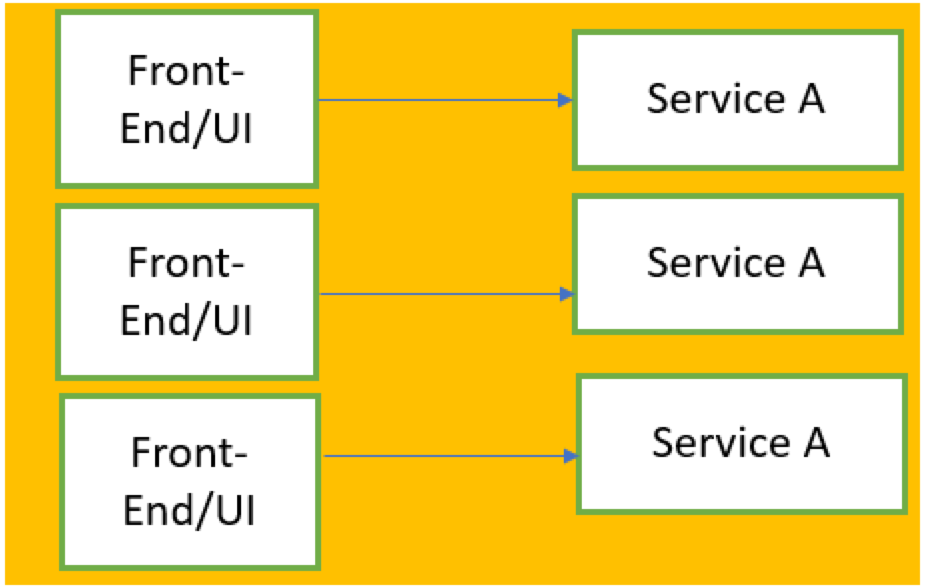
Microservices Architecture

Evolution of UI with Microservices

- Frontend also needed to change
- Micro Front ends
- Single Page Applications
-







/hat next??

Microserivces Integration patterns: Summary

MICROSERVICES DESIGN PATTERNS

Microserivces Integration Patterns

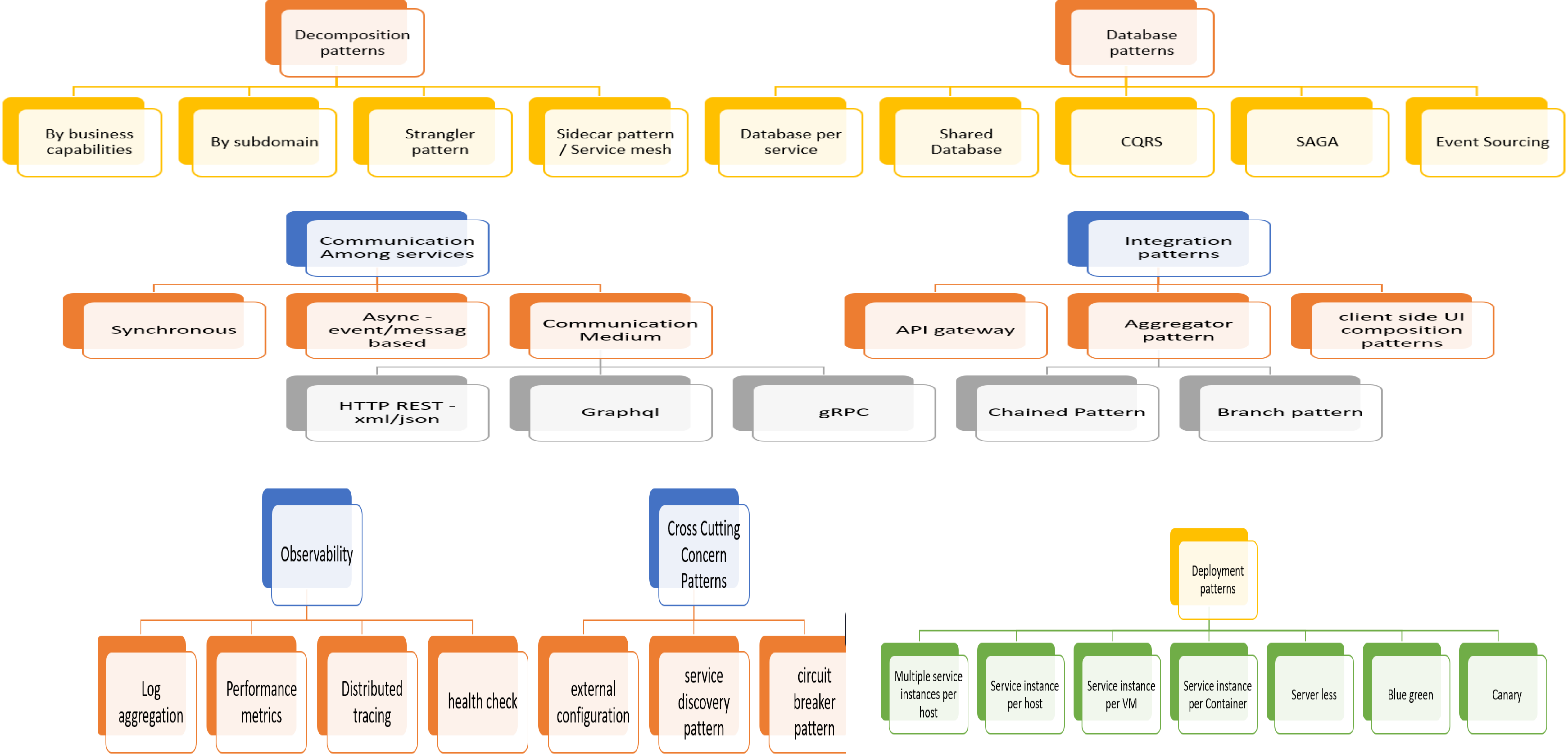
Summary

Microservices

Architecture

Summary

- API Gateway
- Aggregator
  - Branch
  - Chained
-



- Decomposition patterns
  - By business capabilities
  - By subdomain
  - Strangler pattern
  - Sidecar pattern / Service mesh

- Database patterns
  - Database per service
  - Shared Database
  - CQRS
  - SAGA
  - Event Sourcing

- Communication Among services
  - Synchronous
  - Async - event/messag based
  - Communication Medium
    - HTTP REST - xml/json
    - Graphql
    - gRPC

- Integration patterns
  - API gateway
  - Aggregator pattern
    - Chained Pattern
    - Branch pattern
  - client side UI composition patterns

- Observability
  - Log aggregation
  - Performance metrics
  - Distributed tracing
  - health check

- Cross Cutting Concern Patterns
  - external configuration
  - service discovery pattern
  - circuit breaker pattern

- Deployment patterns
  - Multiple service instances per host
  - Service instance per host
  - Service instance per VM
  - Service instance per Container
  - Server less
  - Blue green
  - Canary

What next??

Microservices Observability Patterns:

Log Aggregation & Distributed Tracing

MICROSERVICES DESIGN PATTERNS

Microservices Observability Patterns:

Log Aggregation & Distributed Tracing

Microservices Architecture

/hat next??

croservices Observability

Patterns:

Performance Metrics & Health

MICROSERVICES DESIGN PATTERNS

Microservices Observability Patterns:

Performance Metrics & Health Check

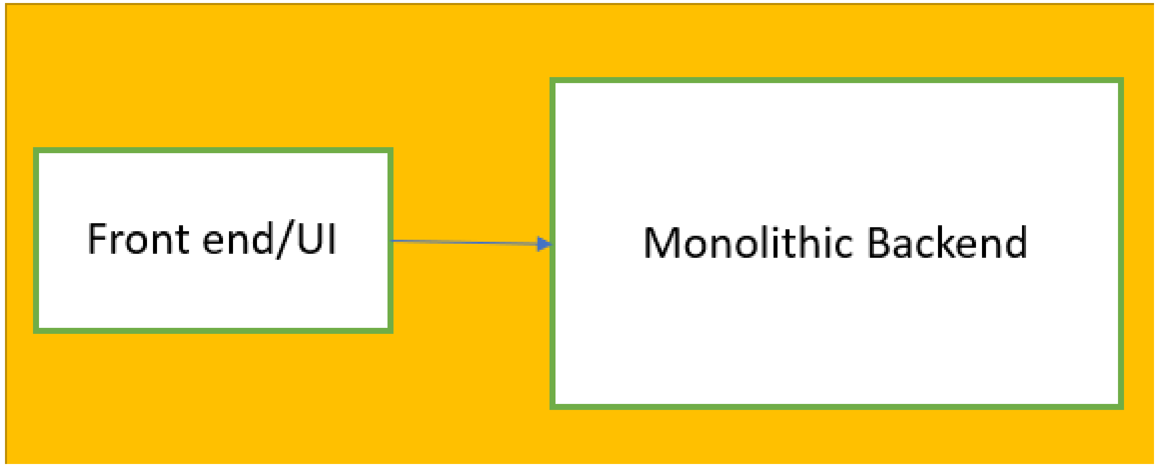
Microservices Architecture

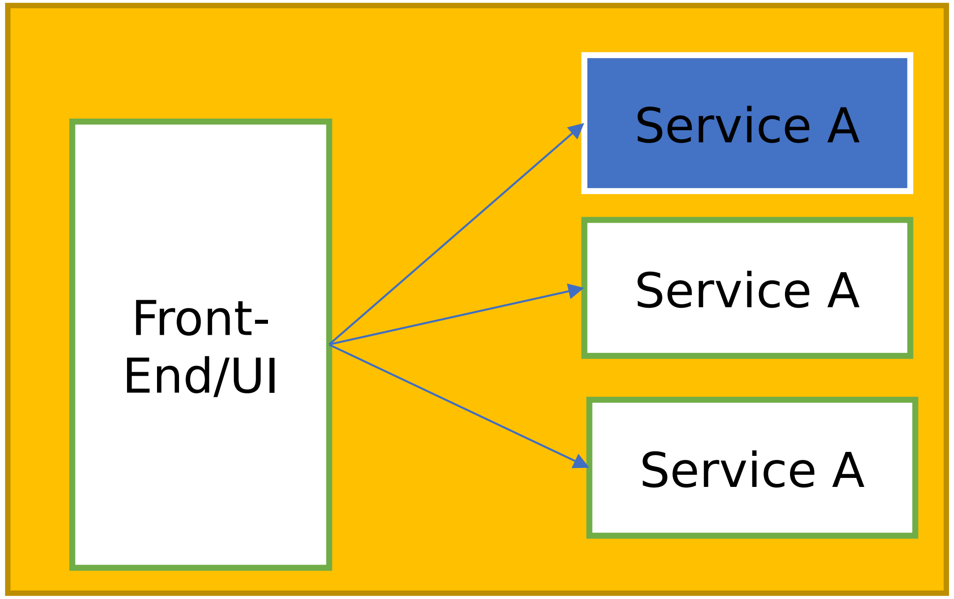
What next??

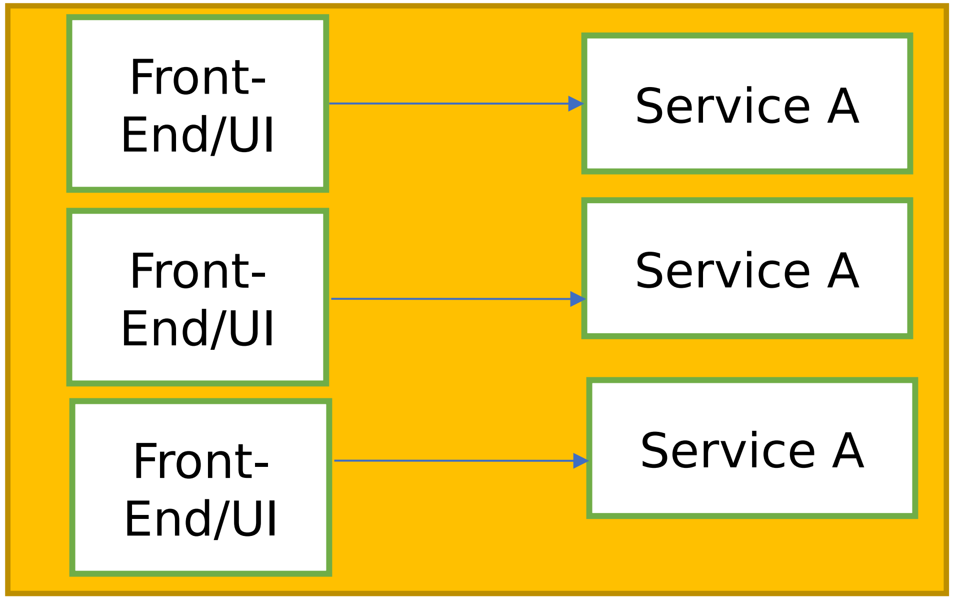
Microservices Observability Patterns: Summary

Evolution of UI with Microservices

- Frontend also needed to change
- Micro Front ends
- Single Page Applications
-







MICROSERVICES DESIGN PATTERNS

Microservices Integration Patterns:

Aggregator

Microservices Architecture

What next??

Microservices Integration Patterns:
Aggregator

Microservices Architecture

Failing badly in microservices

Microservices Architecture

High Availability (HA)

Microservices Architecture

Fault Tolerance
Robustness
Circuit breaker