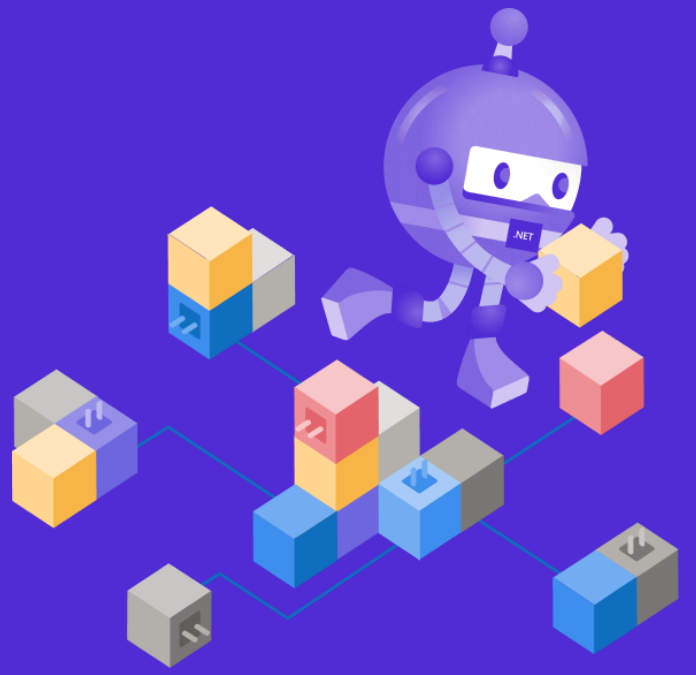
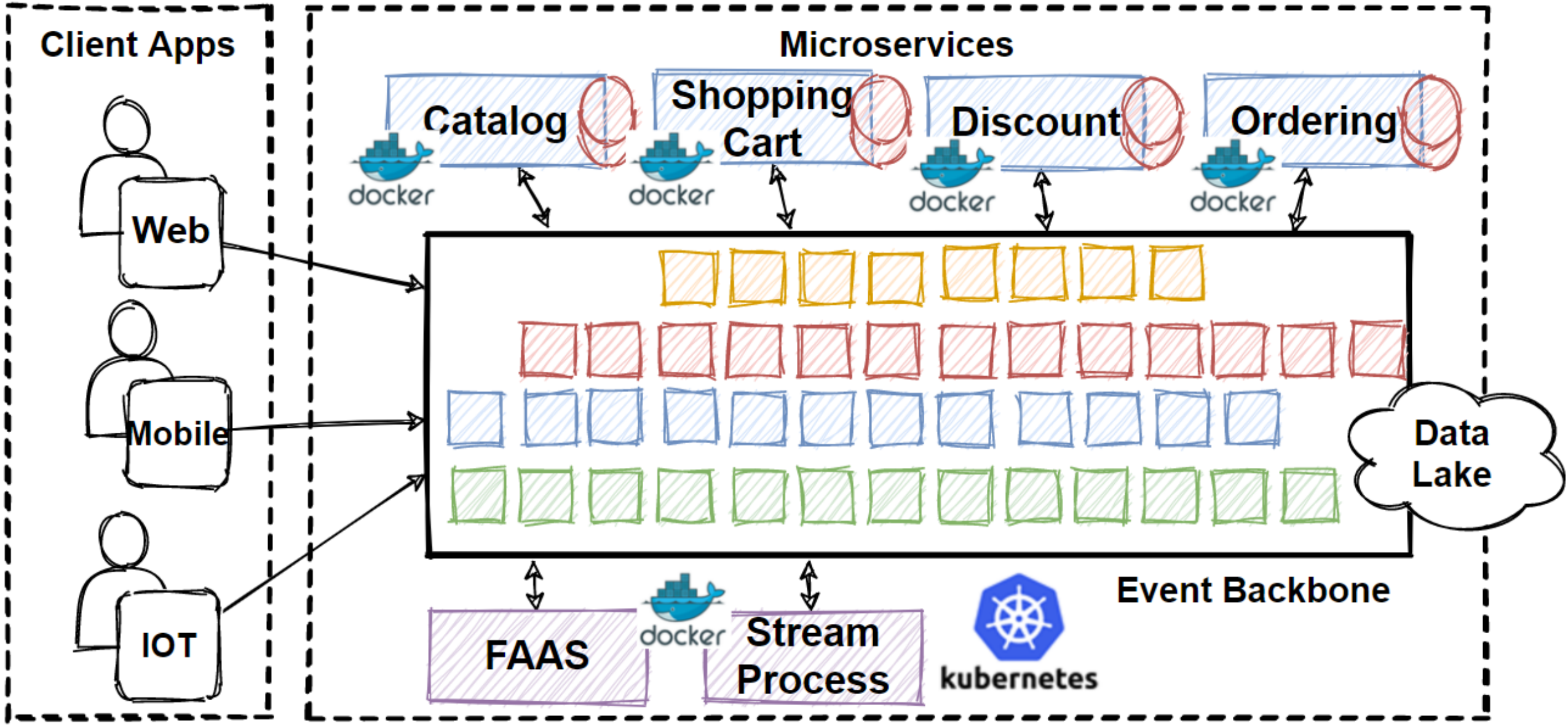
Design Microservices Architecture with Patterns & Practices

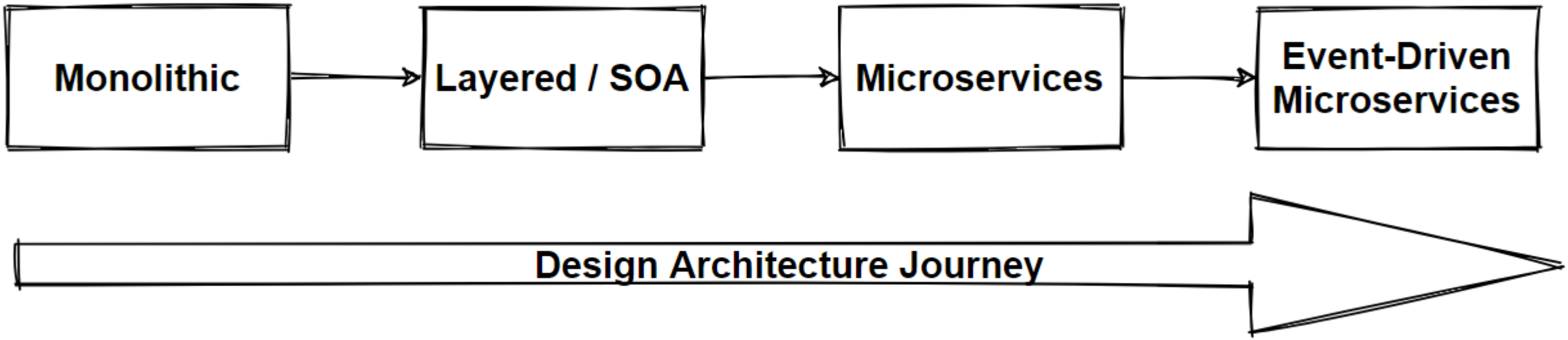


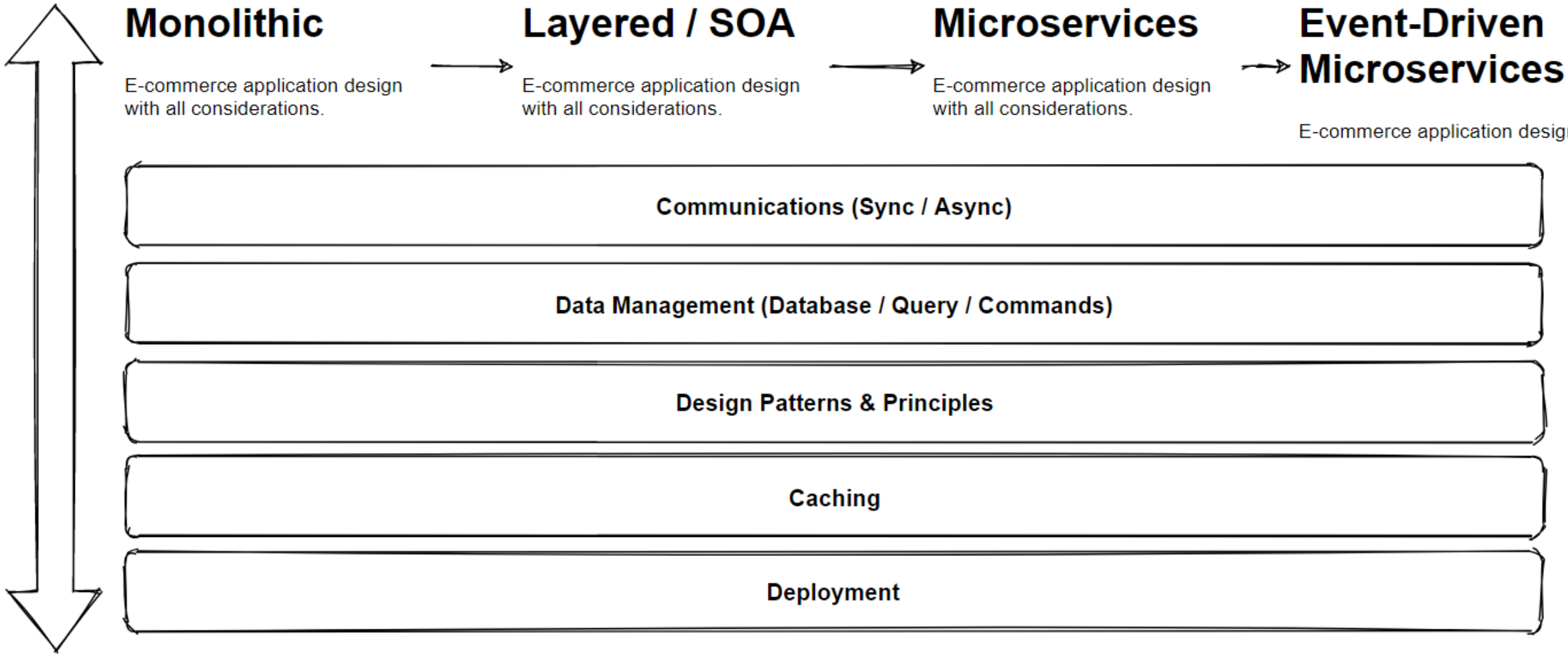
Mehmet Ozkaya
Software Architect, .NET
@ezozkme

Event-Driven Microservices Architecture

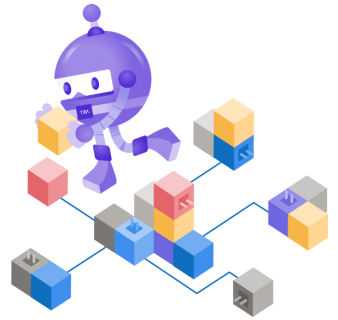


Design Architecture Journey



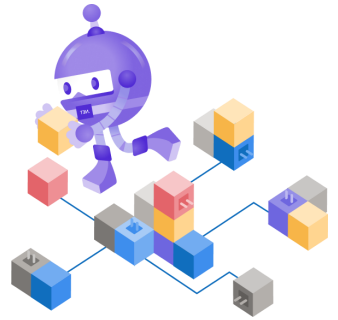

Monolithic
E-commerce application design with all considerations.
Layered / SOA
E-commerce application design with all considerations.
Microservices
E-commerce application design with all considerations.
Event-Driven Microservices
E-commerce application design
Communications (Sync / Async)
Data Management (Database / Query / Commands)
Design Patterns & Principles
Caching
Deployment

Course Target



- Hands-on Design Activities
- Iterate Design Architecture from On-Premises to Cloud Serverless
- Evolves architecture Monolithic to Event-driven Microservices
- Refactoring System Design for handling million of requests
- Apply best practices with microservices design patterns and principles
- Examine microservices patterns with all aspects like Communications, Data Management, Caching and Deployments
- Prepare for **Software Architecture Interviews**
- Prepare for **System Design Architecture Interview** exams

Architecture Position Interviews



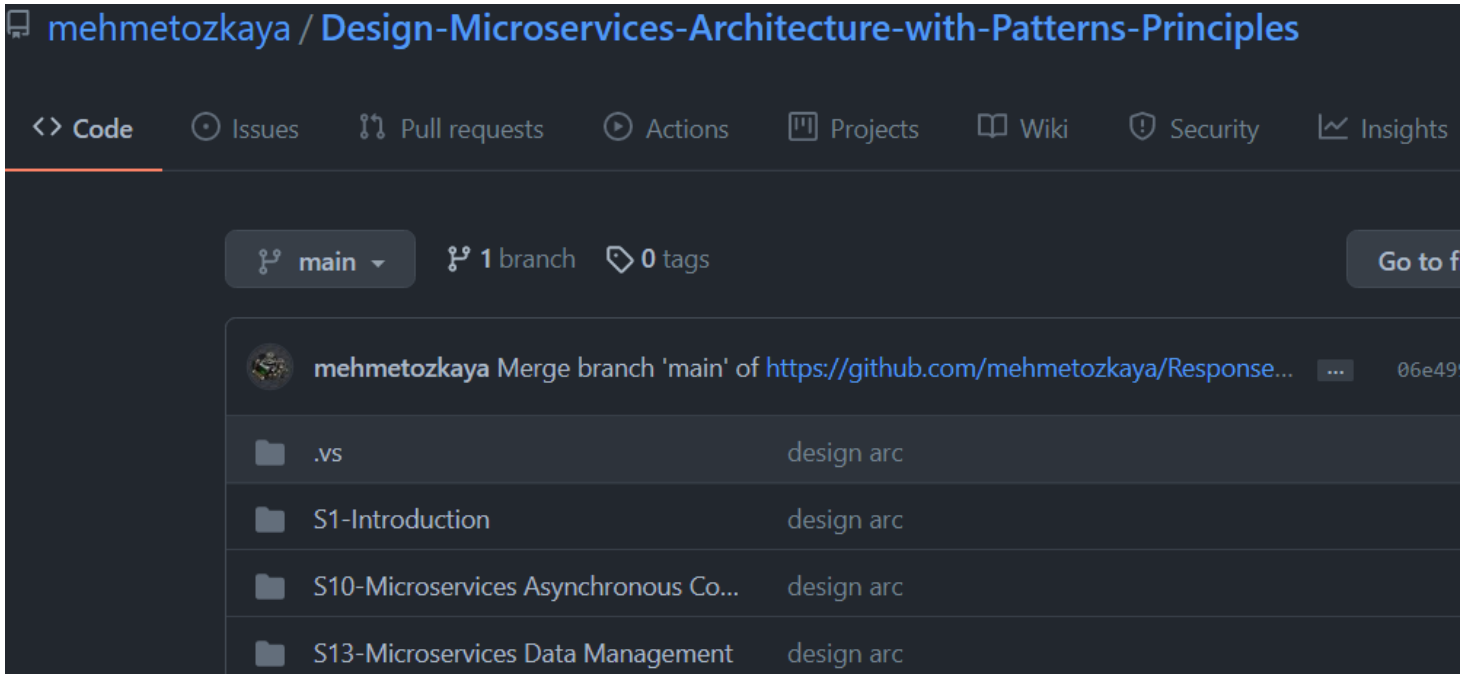
- Prepare for **Software Architecture Interviews**
- Prepare for **System Design Architecture Interview** exams
- Joined to FAANG company interviews
- Involved software architect positions assessment process more than 50+ interviews
- Collect the architecture requirements for software industry positions
- Real-world experience about all architecture positions

Prerequisites

- Basics knowledge Software Design
- Servers, Applications, APIs, Communications

Drawings on Github

- Drawings on Github
- Fork the repository
- Open issues
- Send Pull Requests

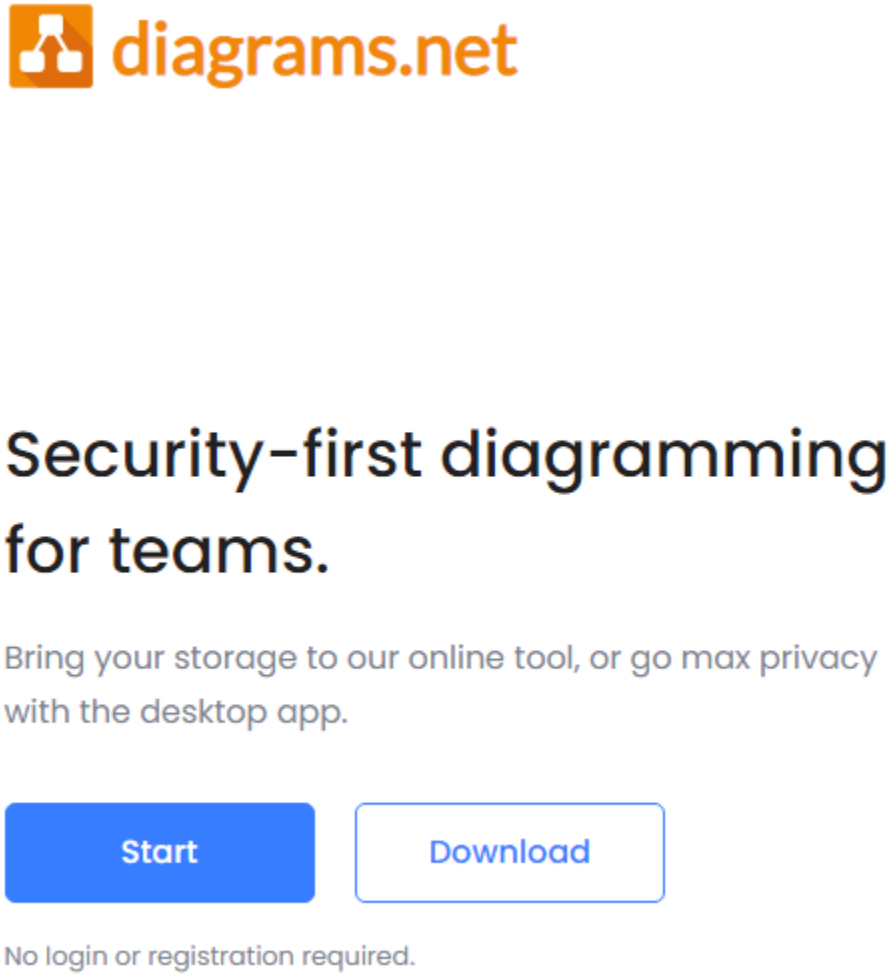


https://github.com/mehmetozkaya/Design-Microservices-Architecture-with-Patterns-Principles

Tools that we will use

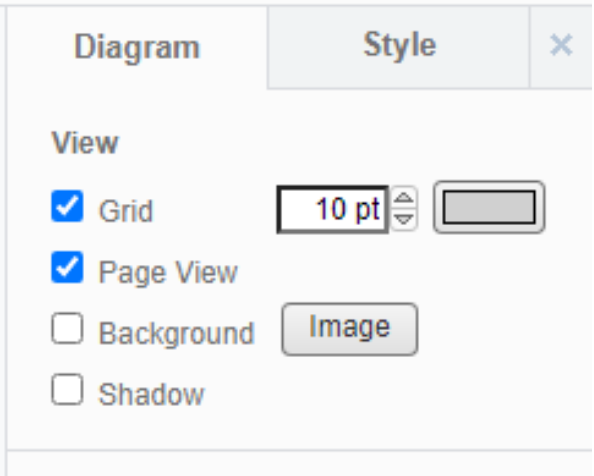
- Use diagrams.net (draw.io)
- Go to https://www.diagrams.net/
- Click to download
- Download draw.io for your os

https://github.com/jgraph/drawio-desktop/releases/tag/v14.6.13

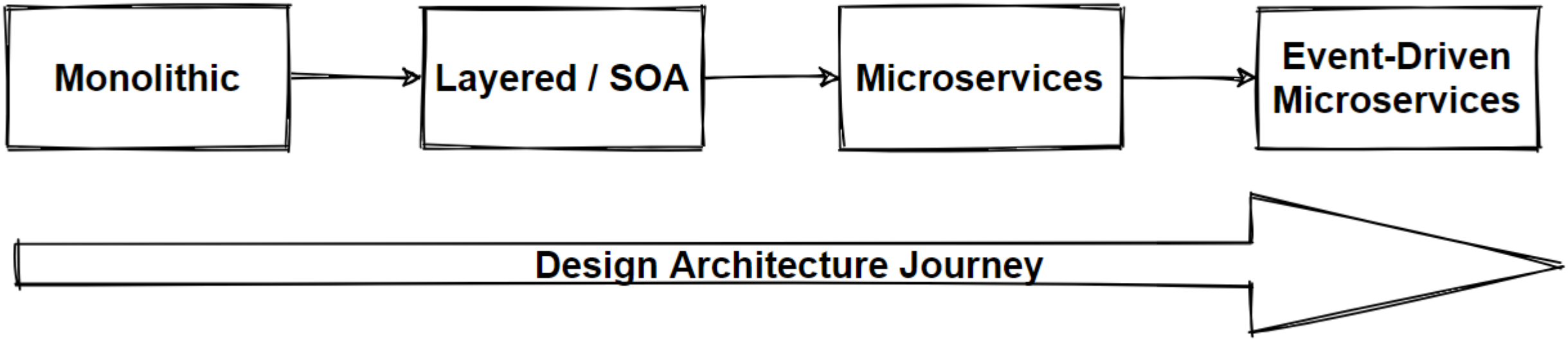


Setting Up Design Environment

- Open draw.io
- Create new Diagram
- Select Blank Diagram
- Set Configurations

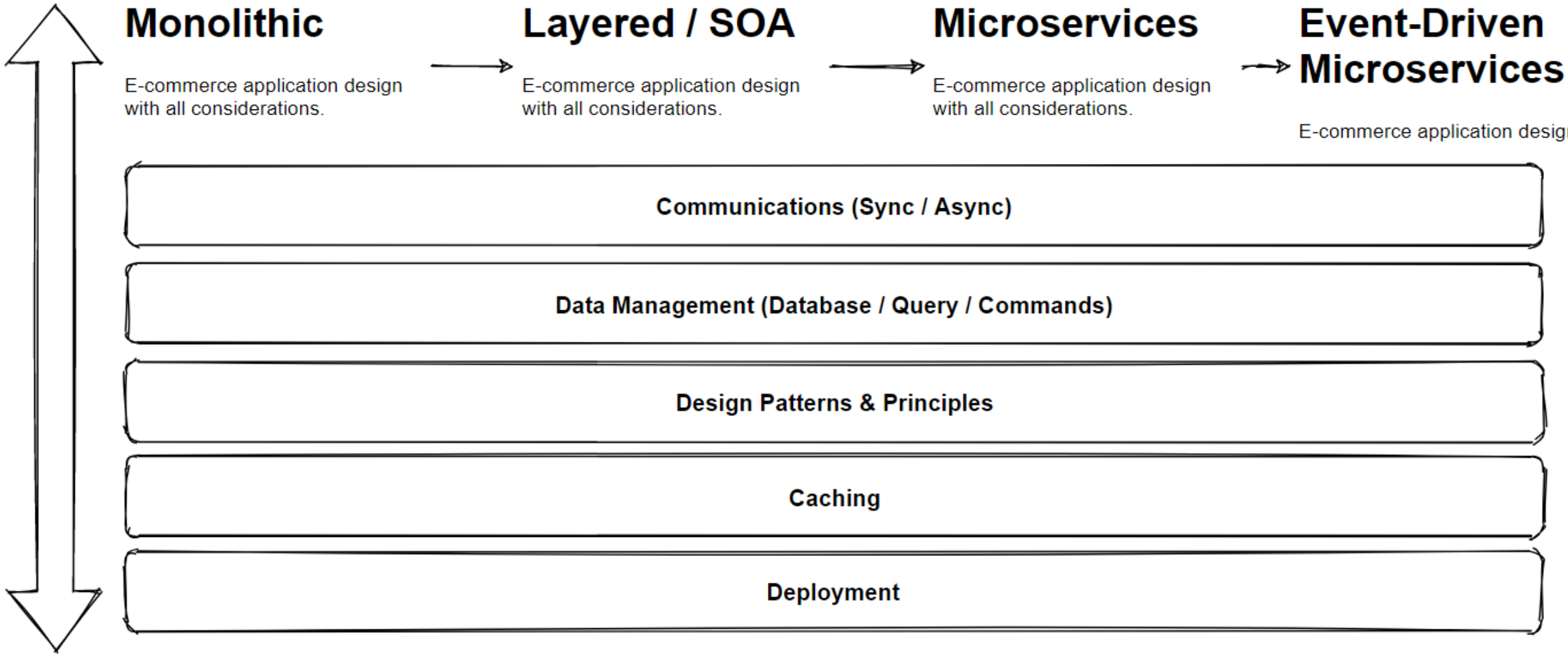


Design Architecture Journey



Subcomponents of Architectures

- MS Communications (Sync / Async)
- MS Data Management (Database / Query / Commands)
- MS Design Patterns & Principles
- MS Caching
- MS Deployment

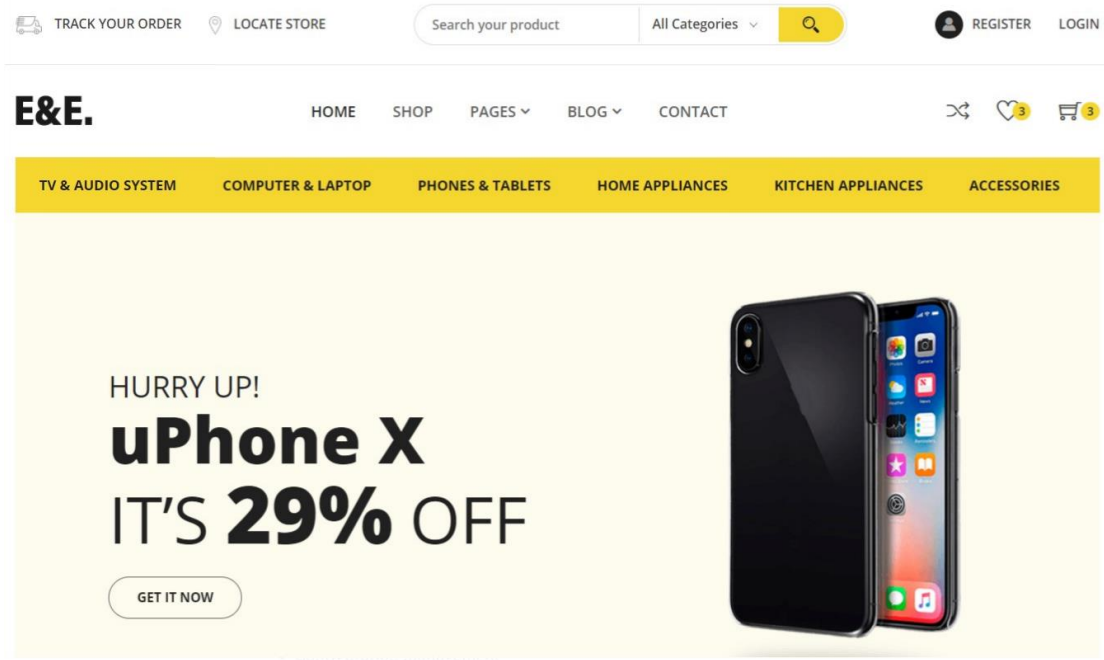

Monolithic
E-commerce application design with all considerations.
Layered / SOA
E-commerce application design with all considerations.
Microservices
E-commerce application design with all considerations.
Event-Driven Microservices
E-commerce application design
Communications (Sync / Async)
Data Management (Database / Query / Commands)
Design Patterns & Principles
Caching
Deployment

Evolve Architecture

- How can we scale the application ?
- How many request that we need to handle in our application ?
- How many second Latency is acceptable for our arch ?

Understand E-Commerce Domain

- Use Cases
- Functional Requirements



Follow steps;

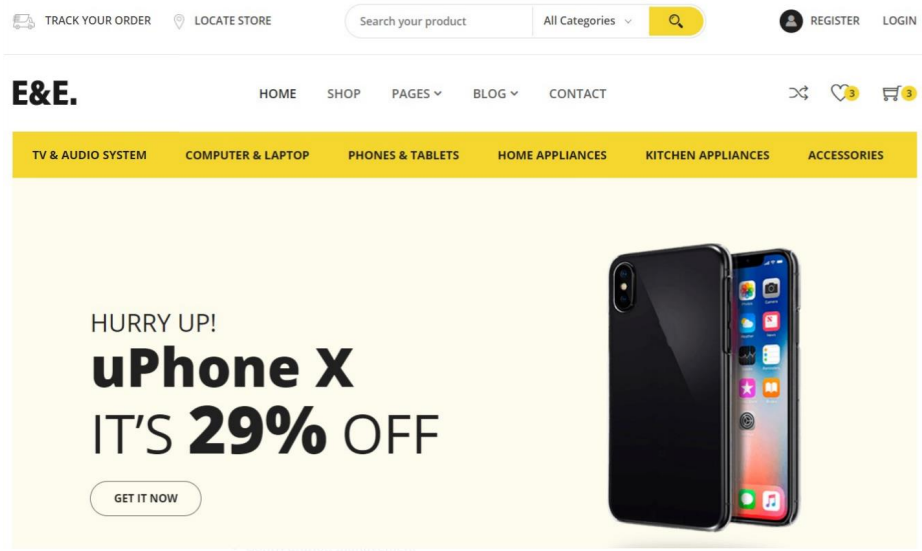
- Requirements and Modelling
- Identify User Stories
- Identify the Nouns in the user stories
- Identify the Verbs in the user stories

E-Commerce Functional Requirements

- List products
- Filter products as per brand and categories
- Put products into the shopping cart
- Apply coupon for discounts and see

the total cost all for all of the items

in shopping cart



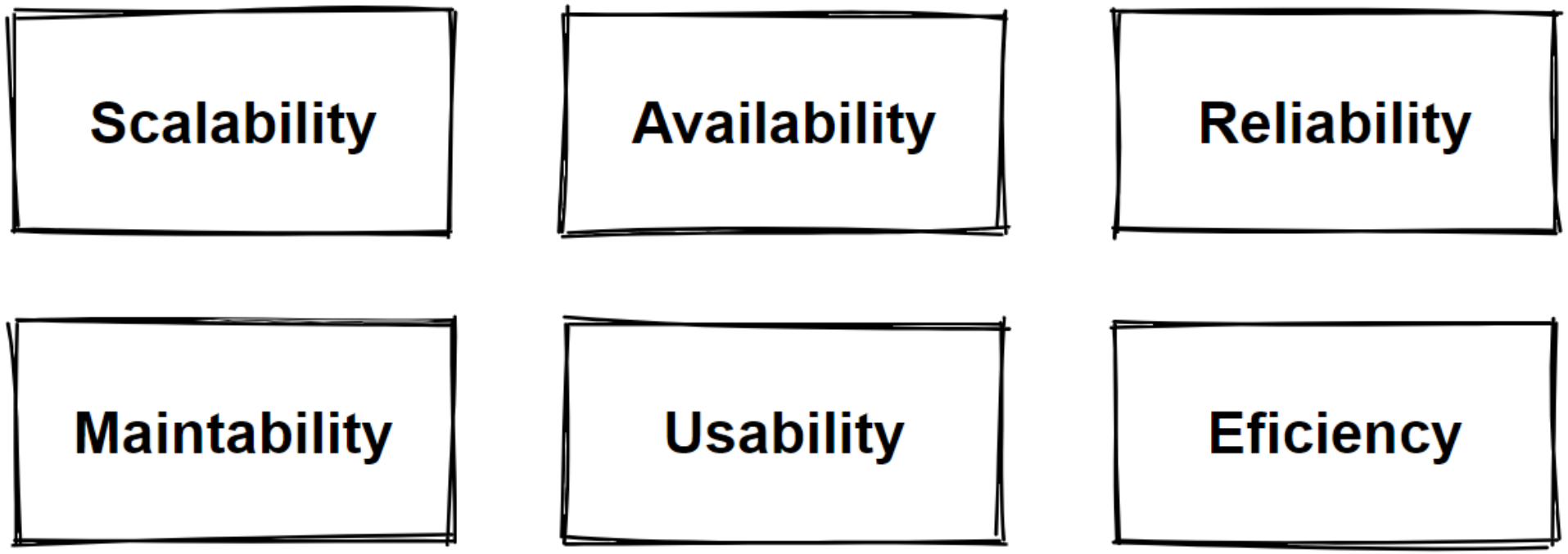
- Checkout the shopping cart and create an order
- List my old orders and order items history

E-Commerce User Stories

- As a user I want to list products
- As a user I want to filter products as per brand and categories
- As a user I want to put products into the shopping cart so that I can check out quickly later
- As a user I want to apply coupon for discounts and see the total cost all for all of the items that are in my cart
- As a user I want to checkout the shopping cart and create an order
- As a user I want to list my old orders and order items history
- As a user I want to login the system as a user and the system should remember my shopping cart items

E-Commerce Non-Functional Requirements

- **-ilities**



Request per Second and Acceptable Latency

Concurrent Users	Requests/second	Latency (Expected)
2K	0.5K	
20K	12K	
100K	80K	<= 2 sec
500K	300K	?

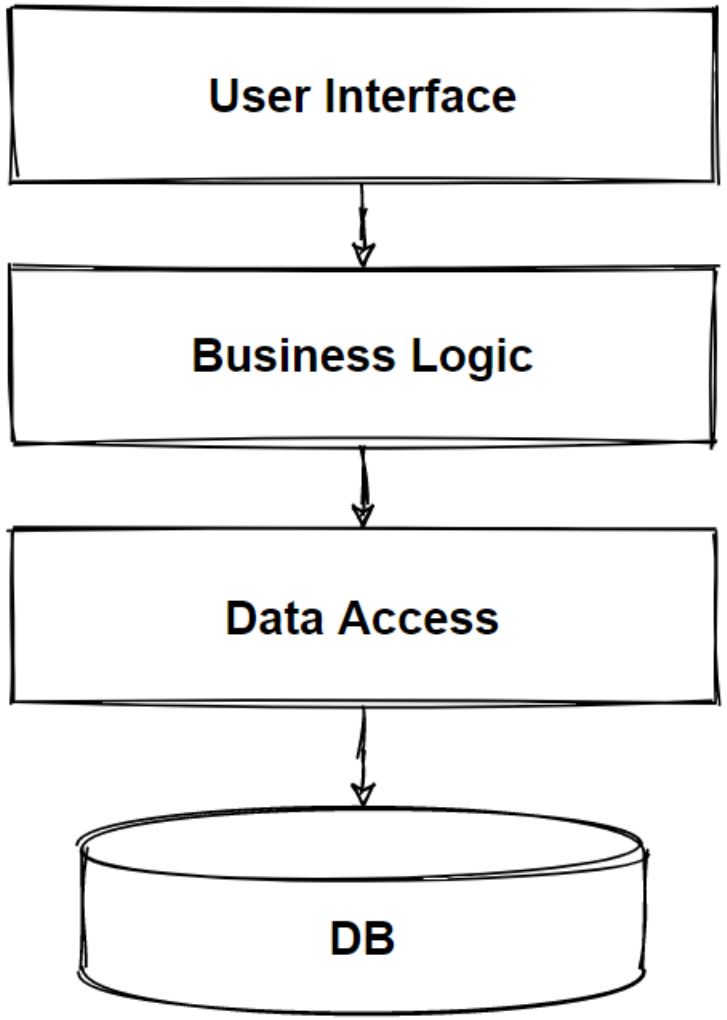
Section 2
Monolithic Architecture

Benefits and Challenges of Monolithic Architecture

Monolithic Architecture

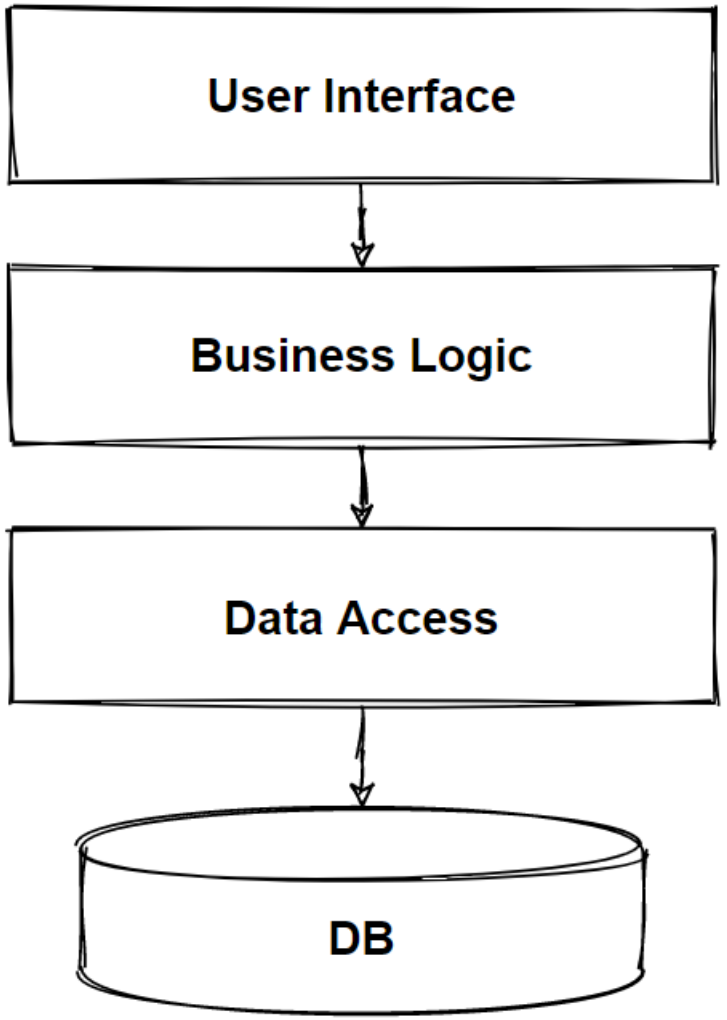
- Single codebase
- Big deployment
- Single jar/war file
- Difficult to manage
- Hard to implement

new feautures



When to use Monolithic Architecture

- They're straightforward to:
- Build
- Test
- Deploy
- Troubleshoot
- Scale vertically (scale up)

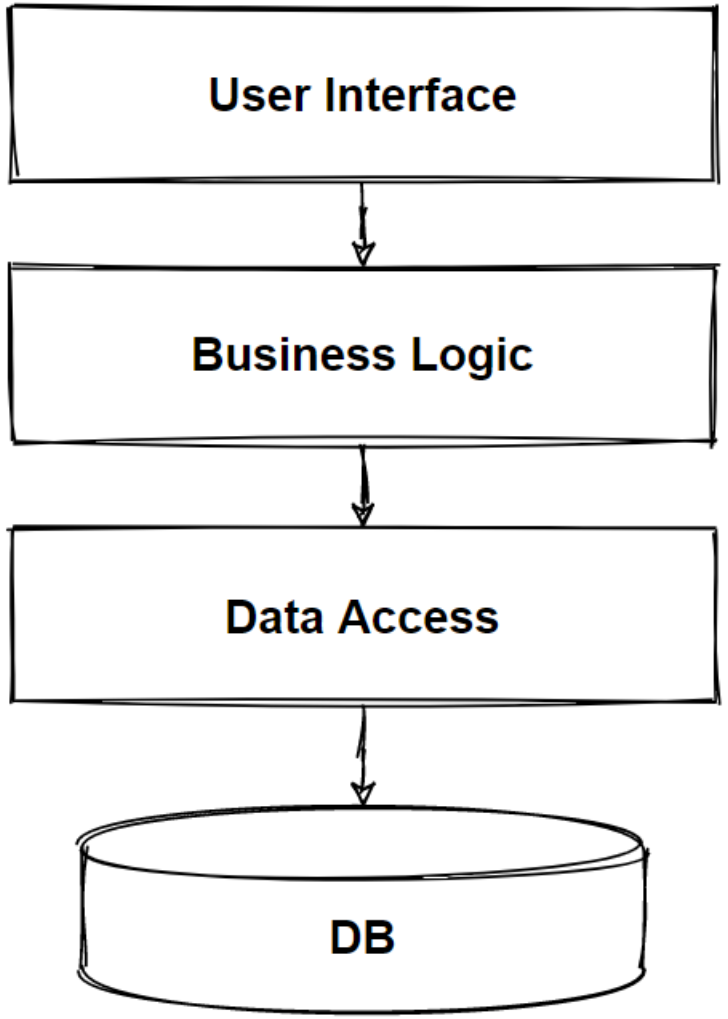


Request per Second and Acceptable Latency

Concurrent Users	Requests/second	Latency (Expected)
2K	0.5K	
20K	12K	
100K	80K	<= 2 sec
500K	300K	?

Benefits of Monolithic Architecture

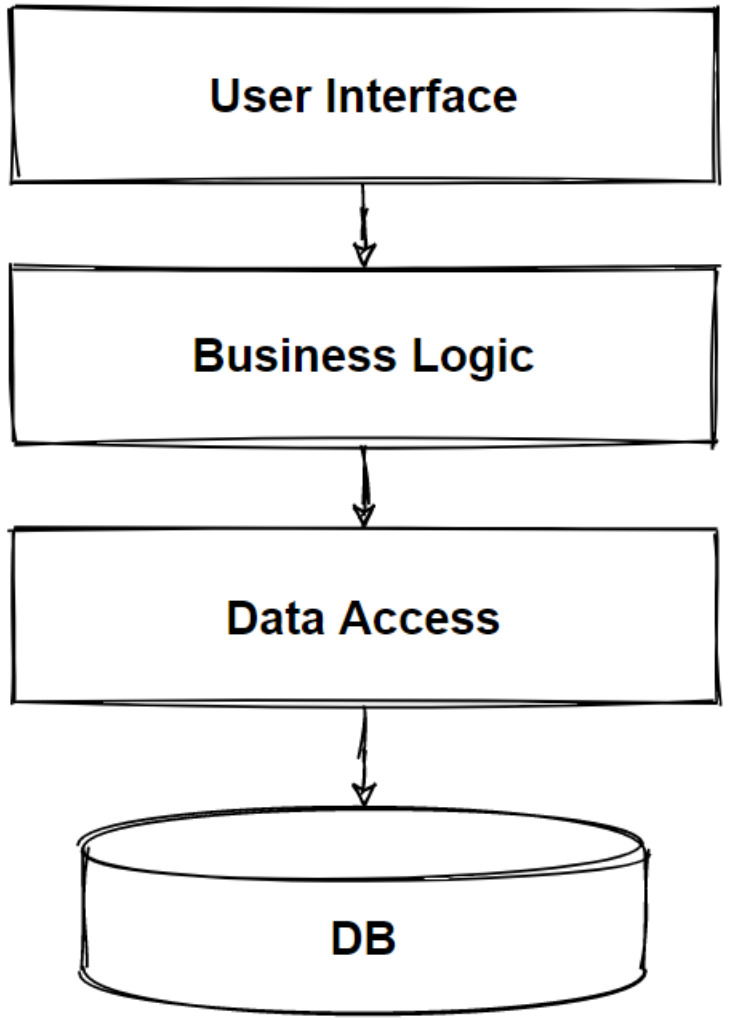
- Simple to develop
- Easier debugging and testing.
- Simple to deploy.



Challenges of Monolithic Architecture

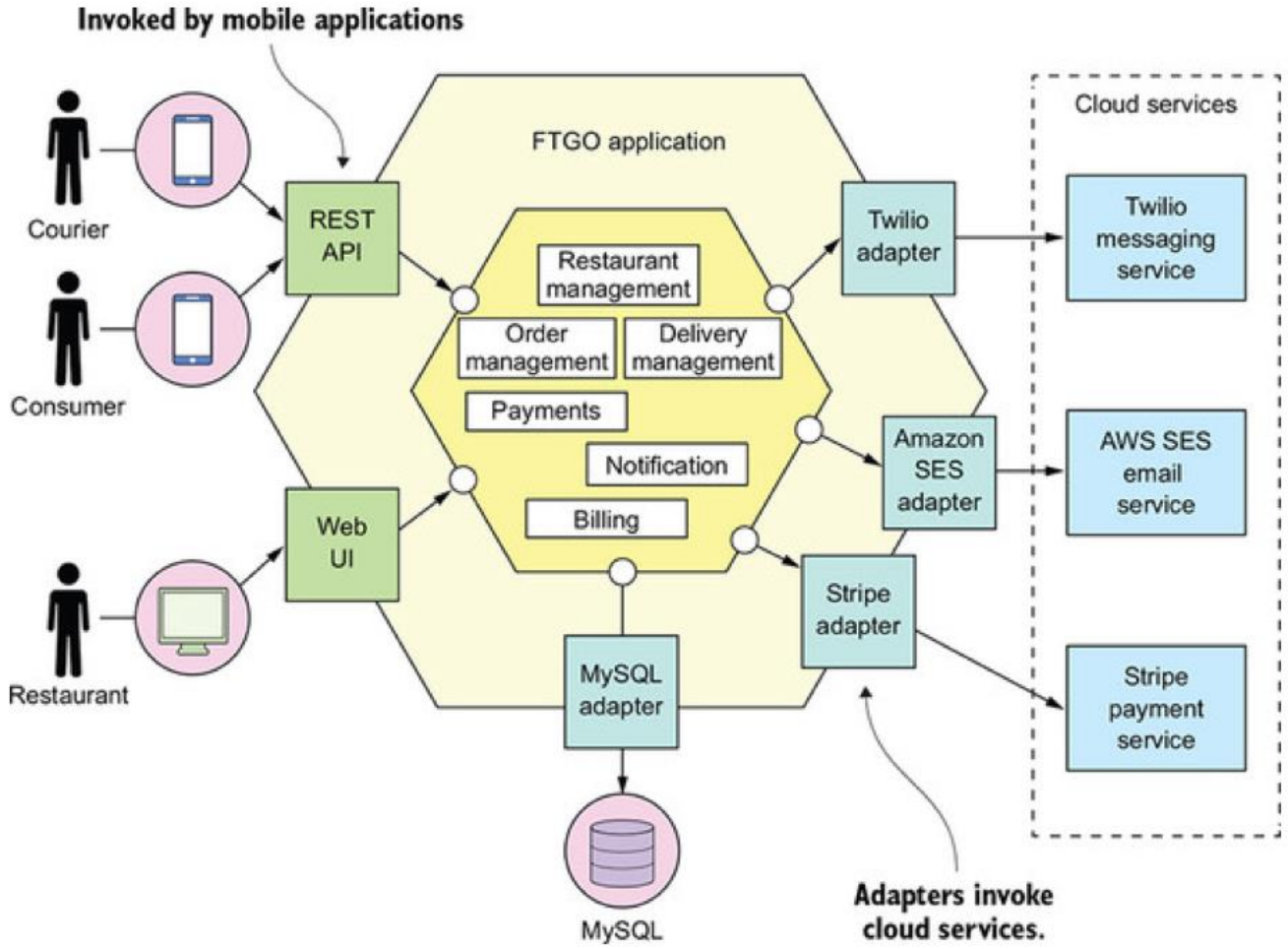
- Complicated to understanding
- Making New changes
- New technology barriers
- Scalability

Check the video ☺
https://www.linkedin.com/posts/bahunov_whats-the-best-definition-of-legacy-code-ugcPost-6814480044996485120-5dQc



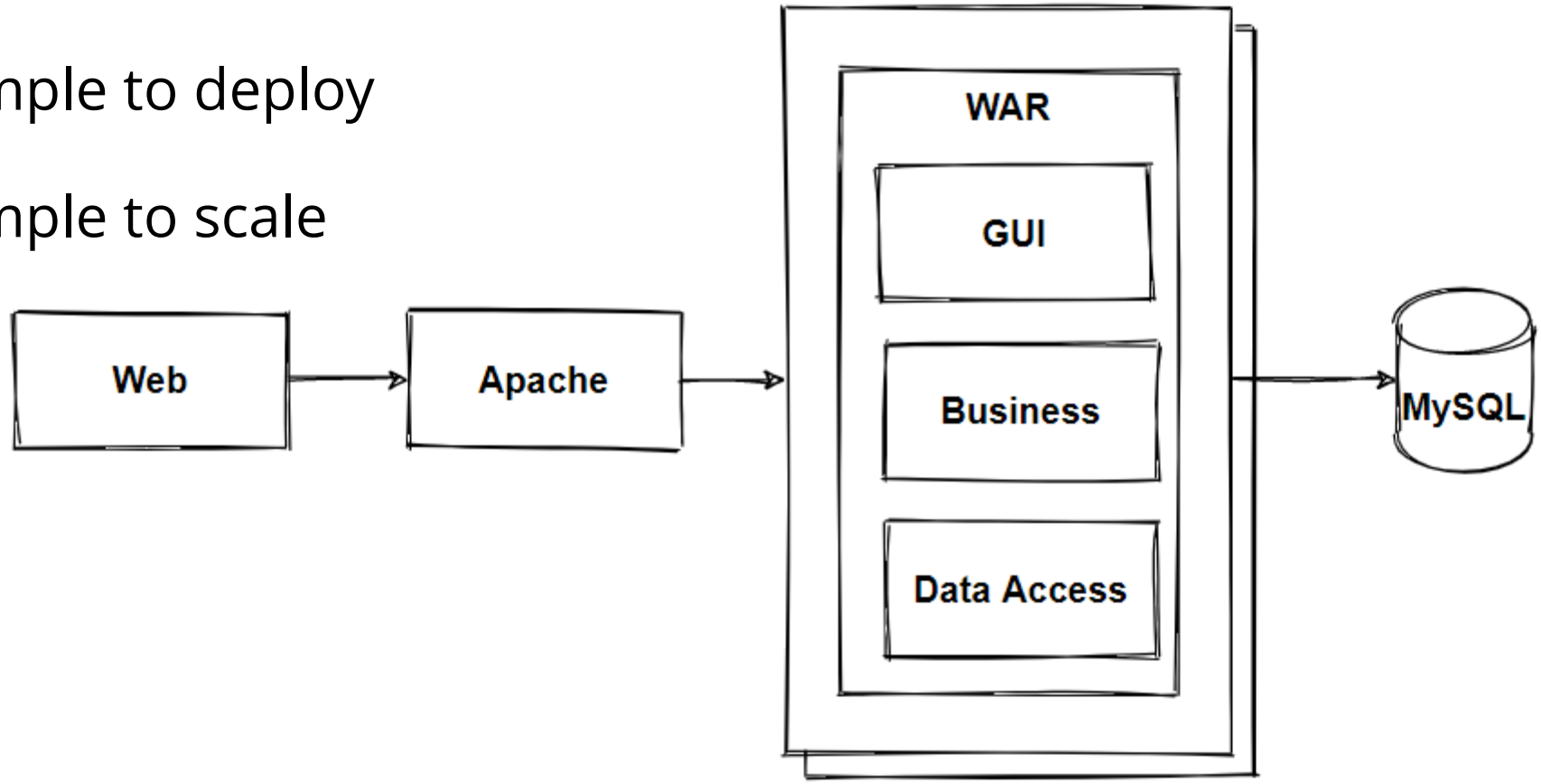
Monolithic Architecture Pros-Cons

- Single codebase
- Easy to develop, debug and deployments
- Complexity
- Hard to maintenance
- Challenge of making changes
- Inability to apply new technology

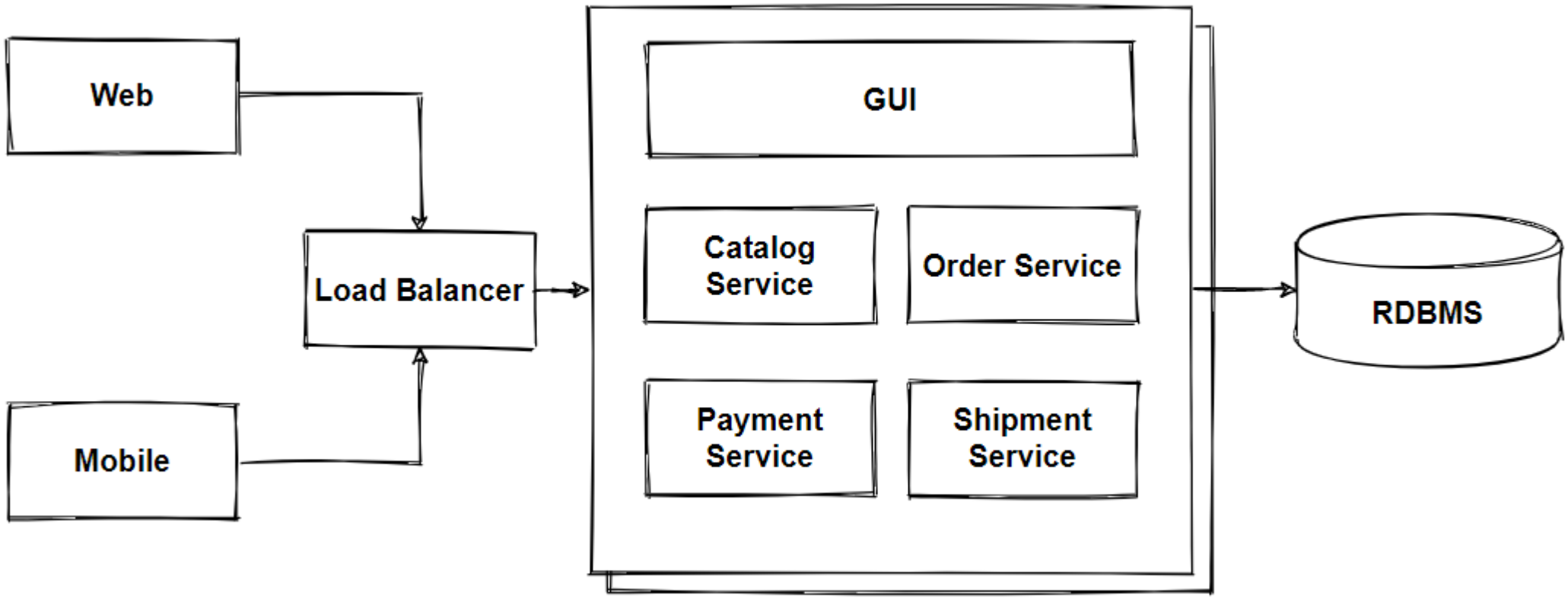


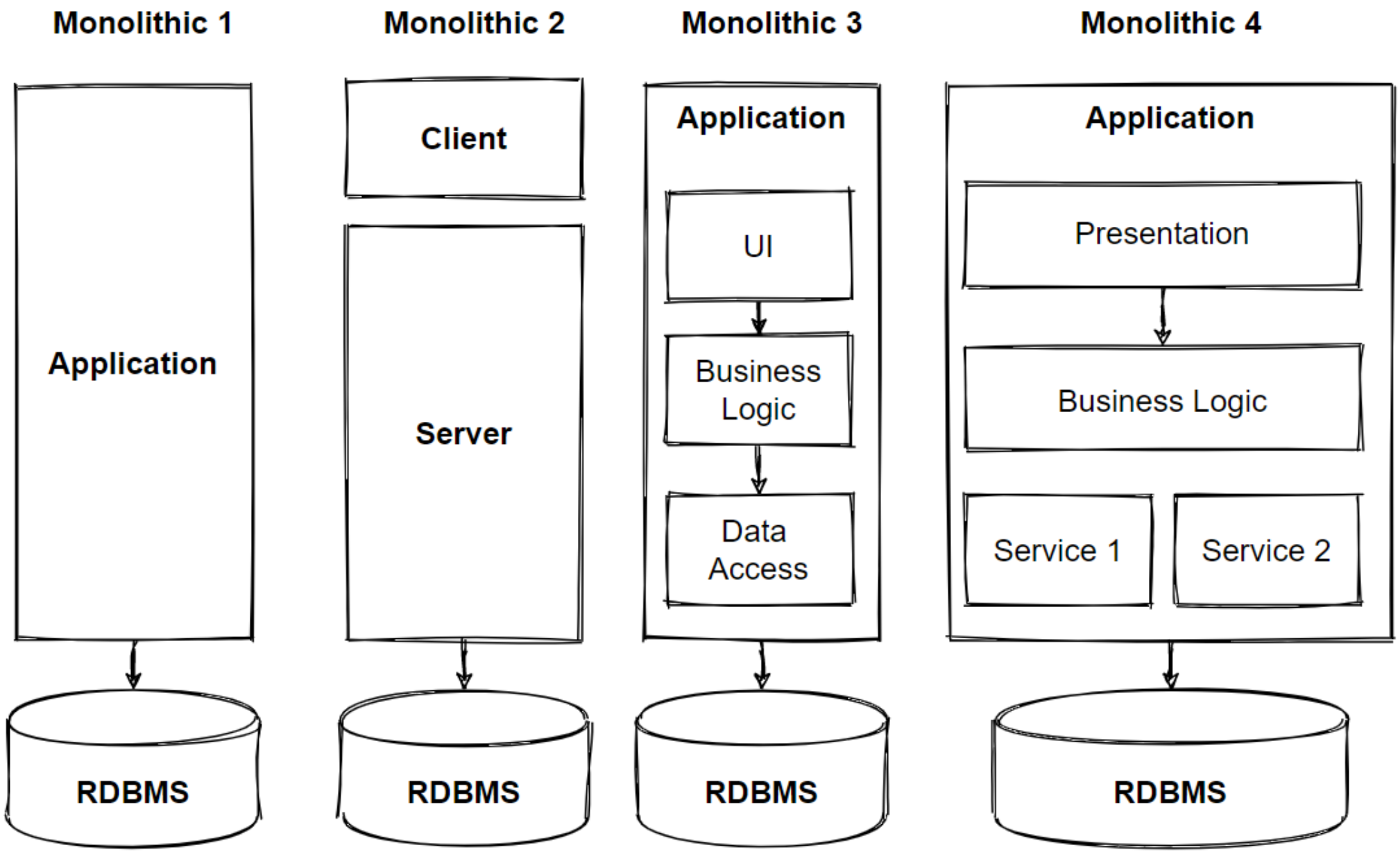
Reference Architectures of Monolithic 1

- Simple to develop
- Simple to deploy
- Simple to scale



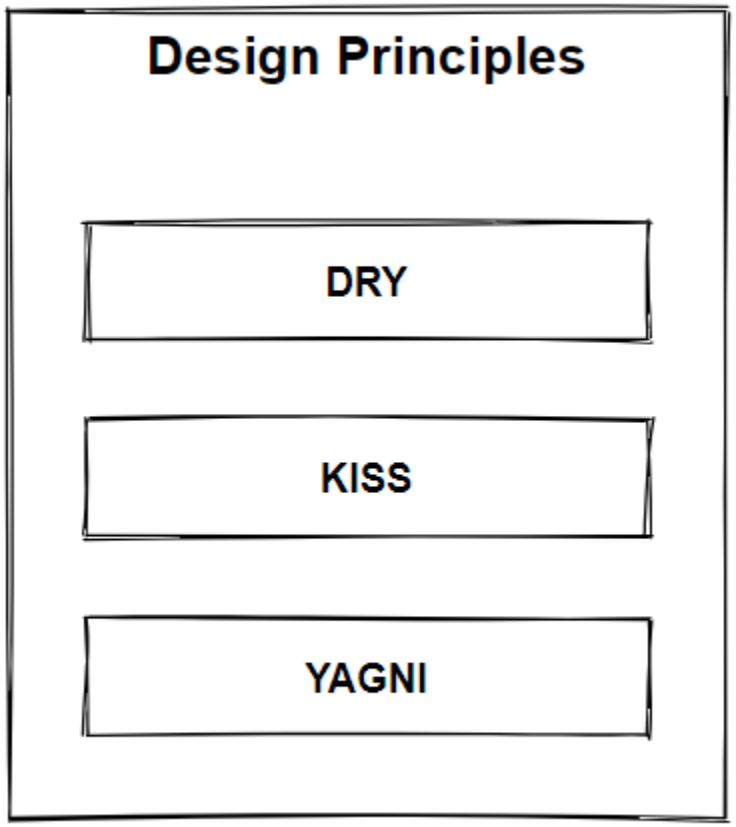
Reference Architectures of Monolithic 2




Monolithic 1
Application
RDBMS
Monolithic 2
Client
Server
RDBMS
Monolithic 3
Application
UI
Business Logic
Data Access
RDBMS
Monolithic 4
Application
Presentation
Business Logic
Service 1
Service 2
RDBMS

Design principles - KISS, YAGNI, DRY

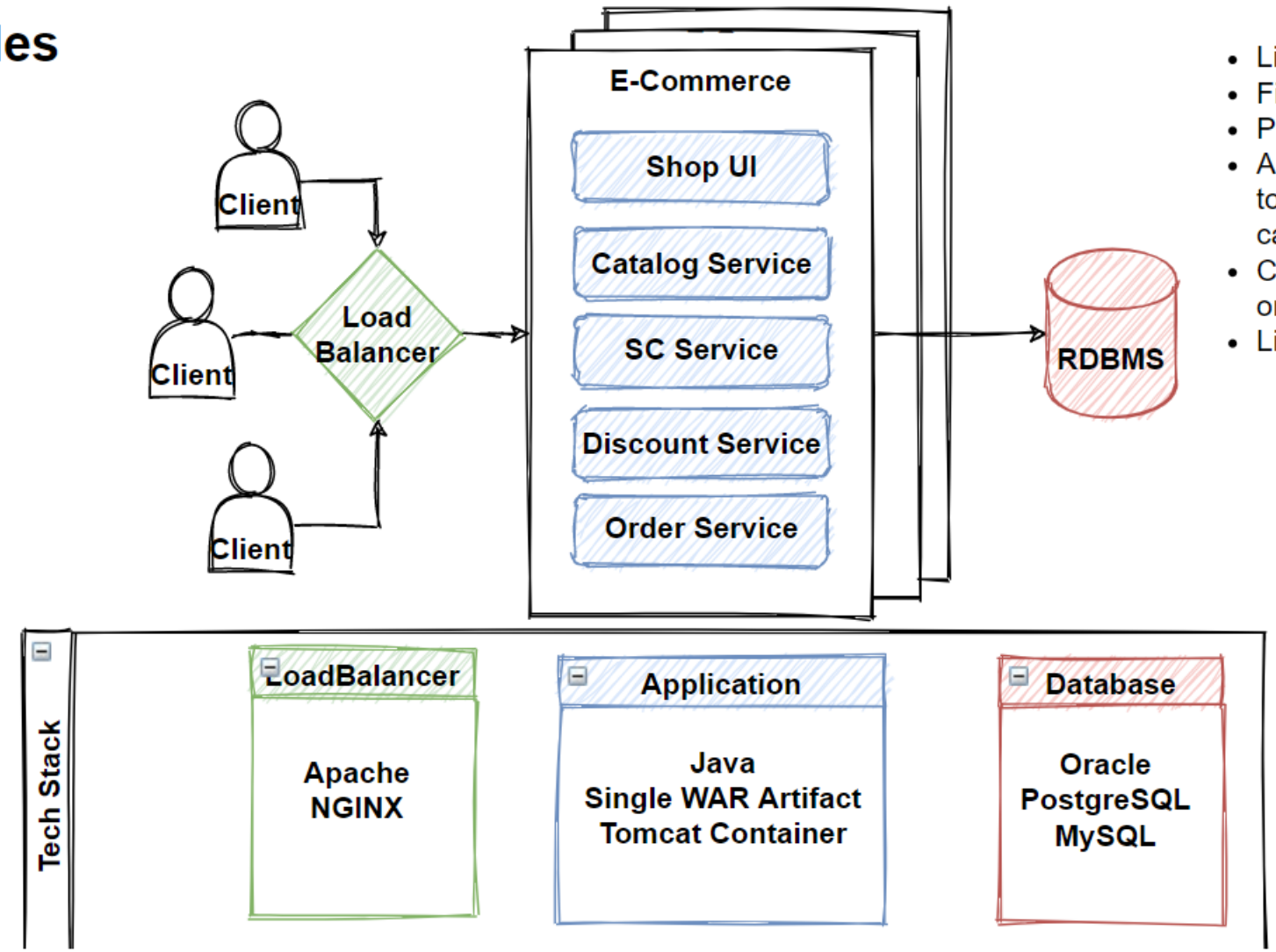
- DRY - Don’t Repeat Yourself
- KISS - Keep It Simple, Stupid
- YAGNI - You Ain’t Gonna Need It



Monolithic Architecture

Principles

- KISS
- YAGNI



Functional Requirements

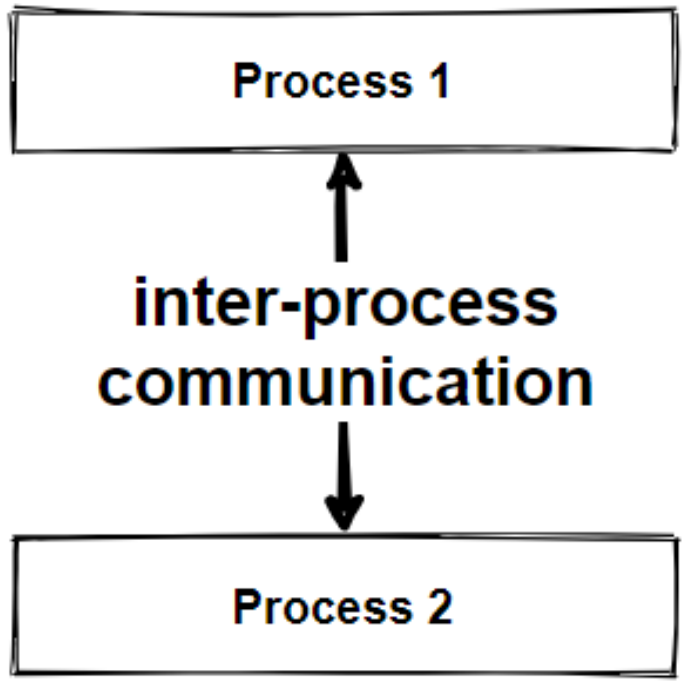
- List products
- Filter products as per brand and categories
- Put products into the shopping cart
- Apply coupon for discounts and see the total cost all for all of the items in shopping cart
- Checkout the shopping cart and create an order
- List my old orders and order items history

Non-Functional Requirements

- Scalibility
- Increase Concurrent User

Communication of Components - E-Commerce App - Monolithic Application

- Inter-Process Communication
- Same server with all modules
- Processes to communicate with each other by method calls into the code

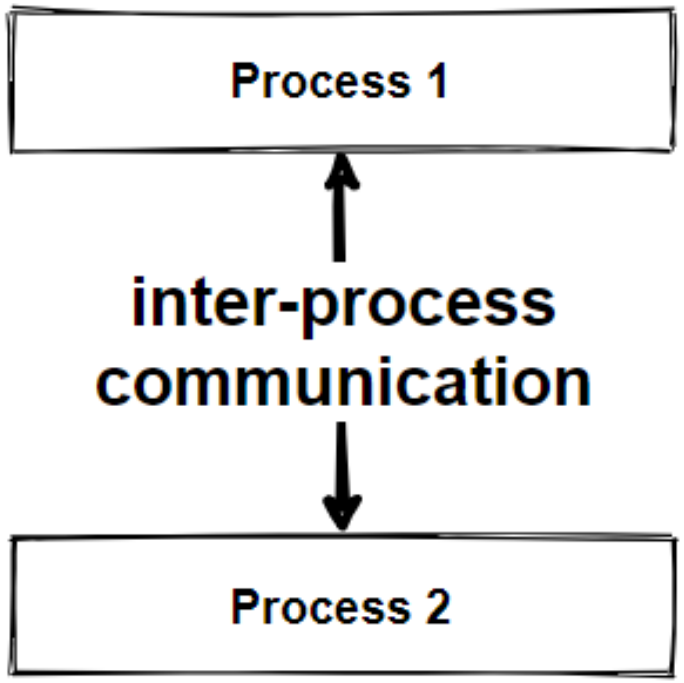


Inter-Process communication - Transaction Management in Monolithic Architectures

- Single database of the whole application

- Simply commit and rollback operations

```
function place_order()
    do_payment
    decrease_stock
    send_shipment
    generate_bill
    update_order
```

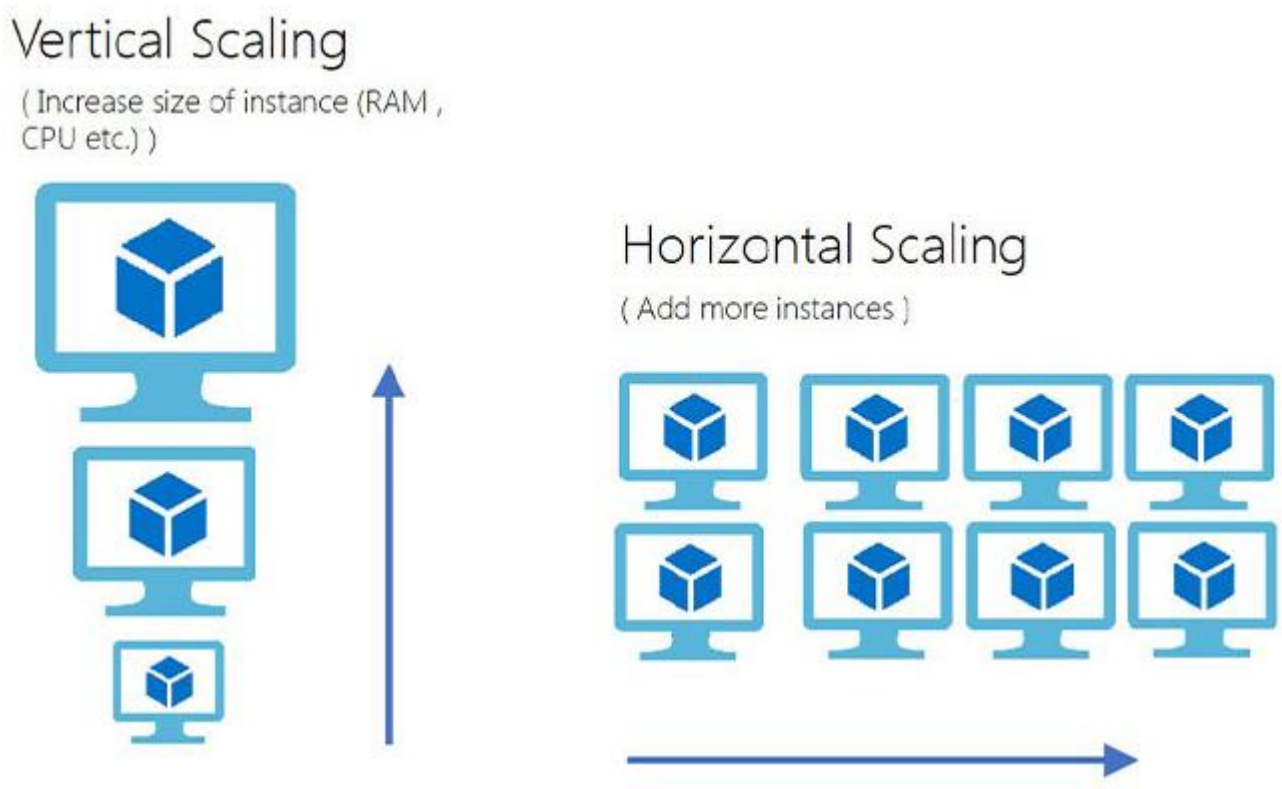


Section 3
Scalability

Vertical Scaling - Horizontal Scaling

Scalability - Vertical Scaling - Horizontal Scaling

- Vertical Scaling = scaling up
- Horizontal scaling = scaling out
- The number of requests an application can handle
- To prevent downtime, and reduce latency, you must scale
- Horizontal scaling by adding more machines
- Vertical scaling by adding more power



https://www.webairy.com/horizontal-and-vertical-scaling/

Scalability - How many concurrent request can accommodate our design ?

Concurrent Users	Requests/second	Latency (Expected)
2K	0.5K	
20K	12K	
100K	80K	<= 2 sec
500K	300K	?

Vertical Scaling - Scale up

- Makes the nodes stronger
- Adding more computing power
- Same code on machines

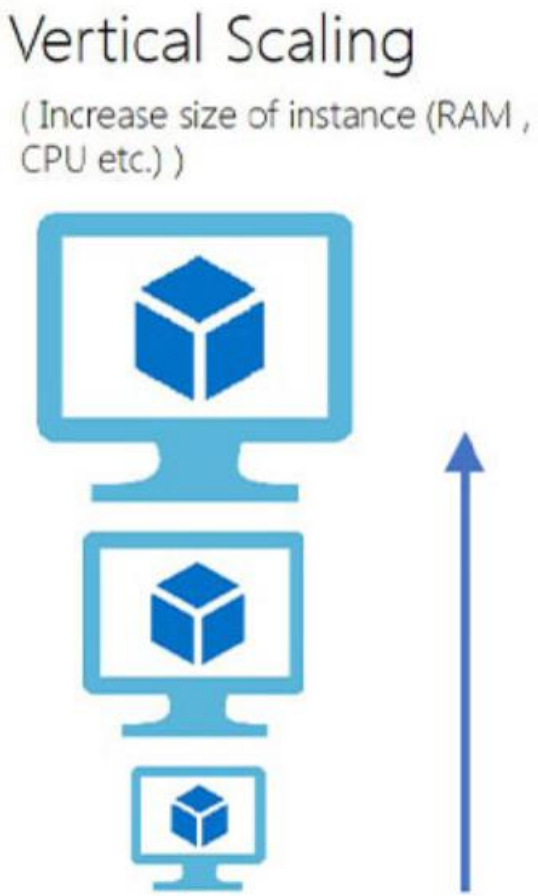
with better specs

- Adding additional CPU, RAM,

and DISK

- Increase power since to **hardware**

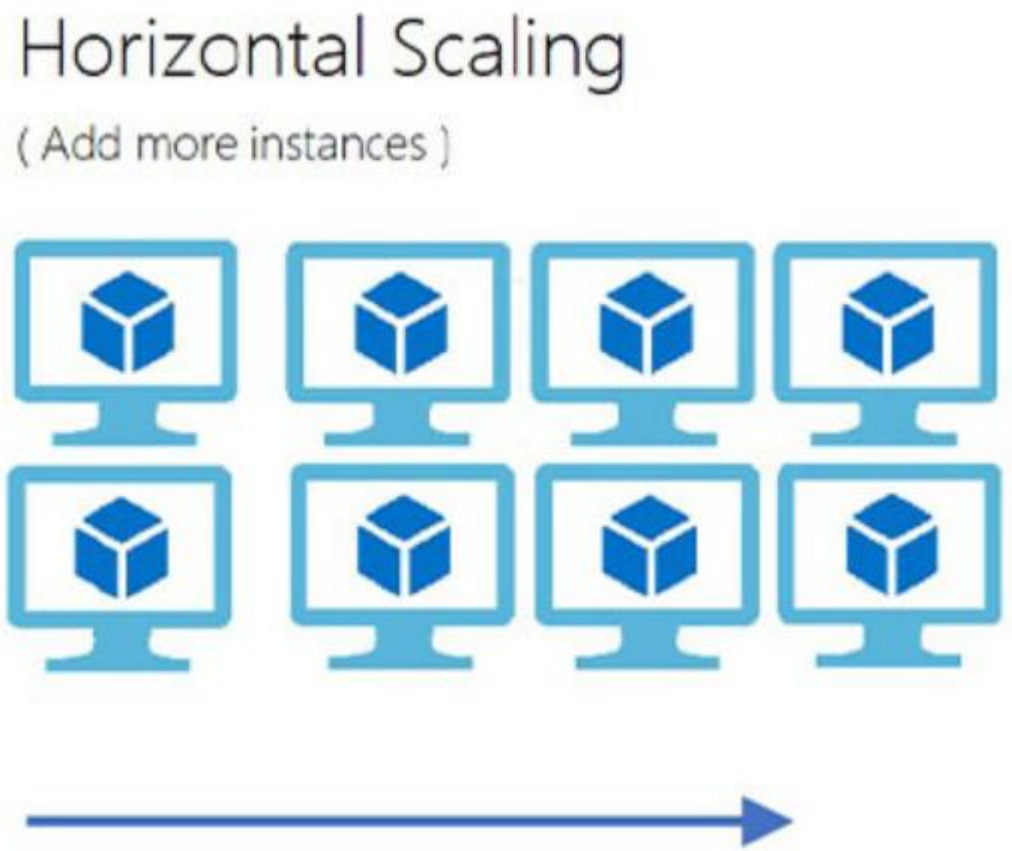
limitations when you reach the maximum capacity



Horizontal Scaling - Scale out

- Splitting the load between different servers
- Adds more instances of machines
- Share the processing power
- Gives you scalability but also reliability

- **Stateful or Stateless**
- **CAP Theorem**

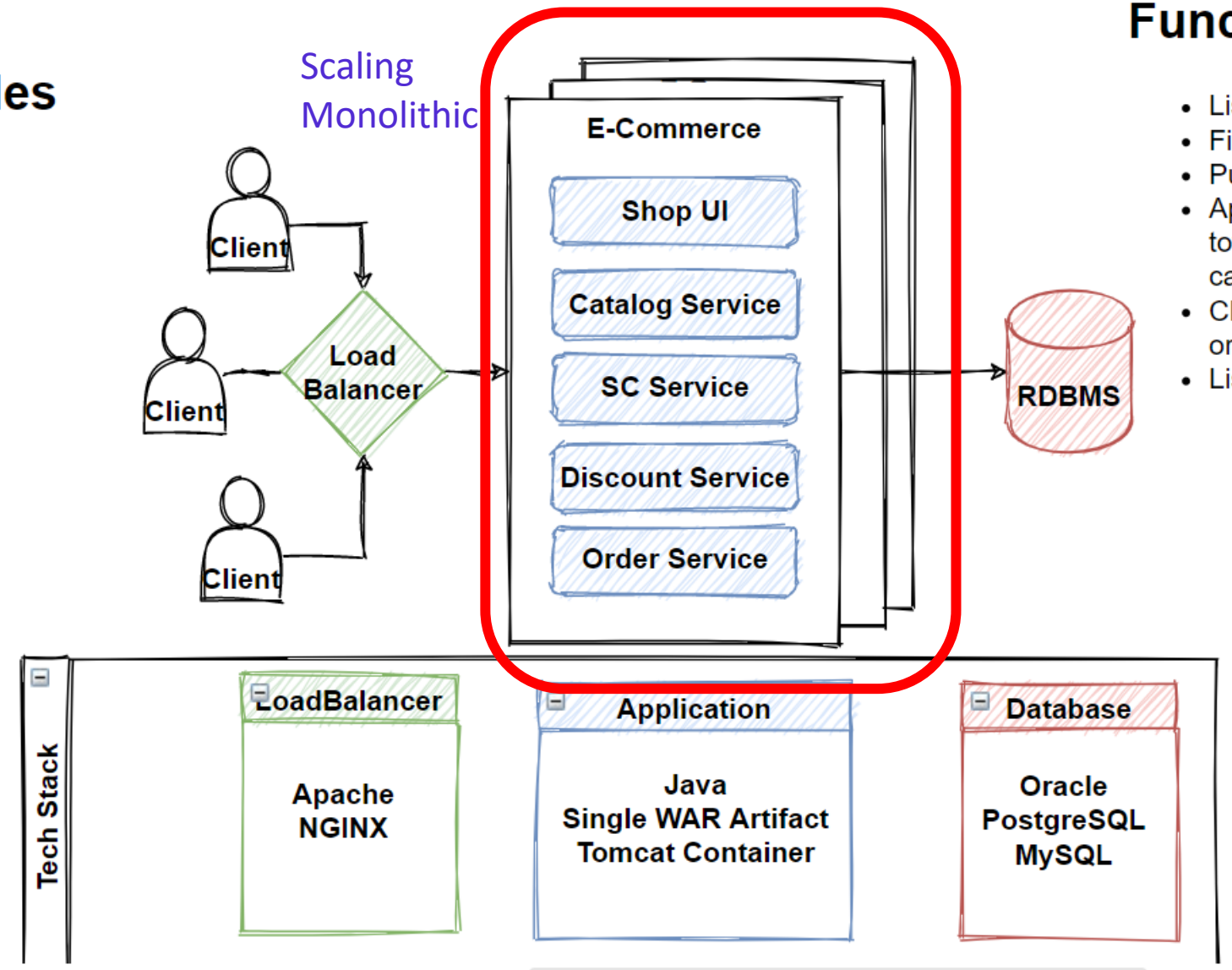


Monolithic Architecture

Principles

- KISS
- YAGNI

Scaling Monolithic



Functional Requirements

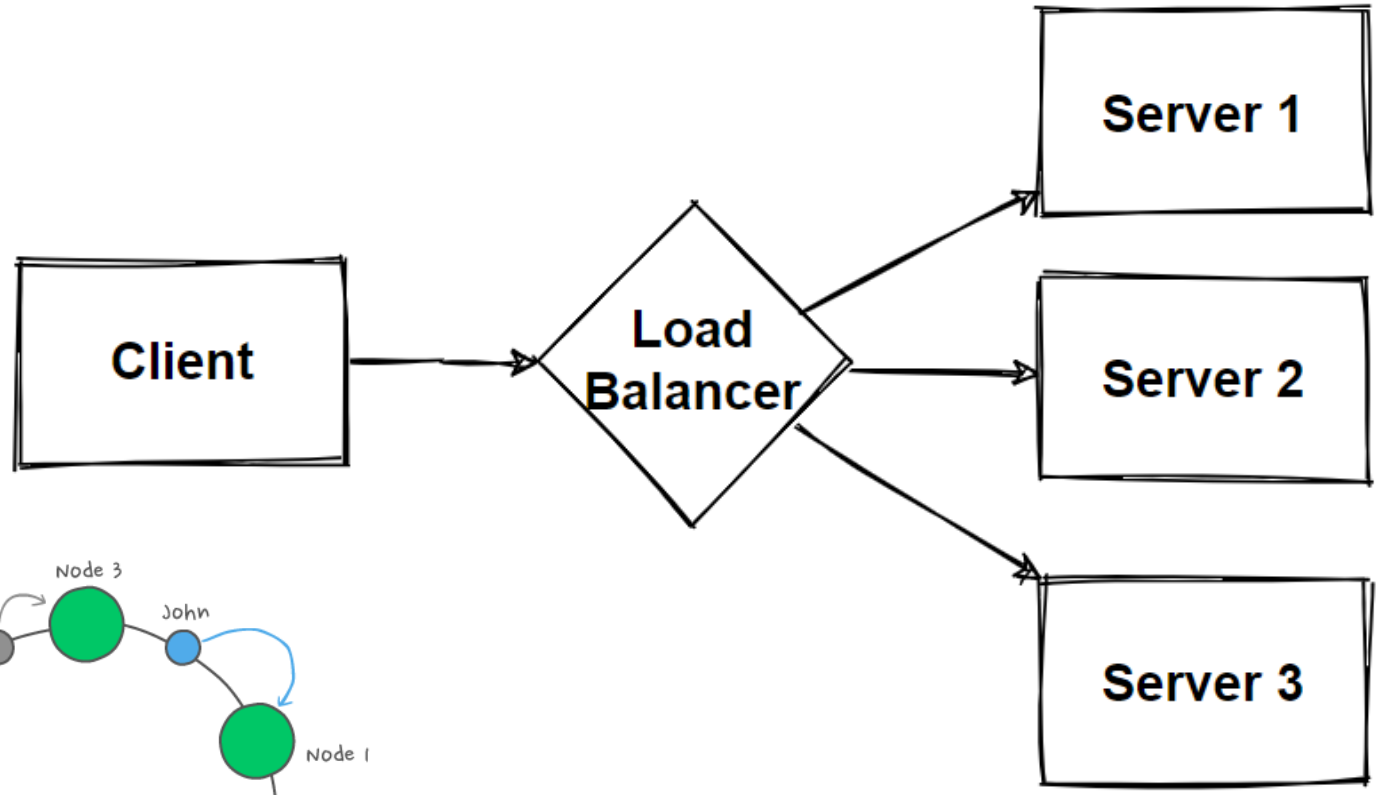
- List products
- Filter products as per brand and categories
- Put products into the shopping cart
- Apply coupon for discounts and see the total cost all for all of the items in shopping cart
- Checkout the shopping cart and create an order
- List my old orders and order items history

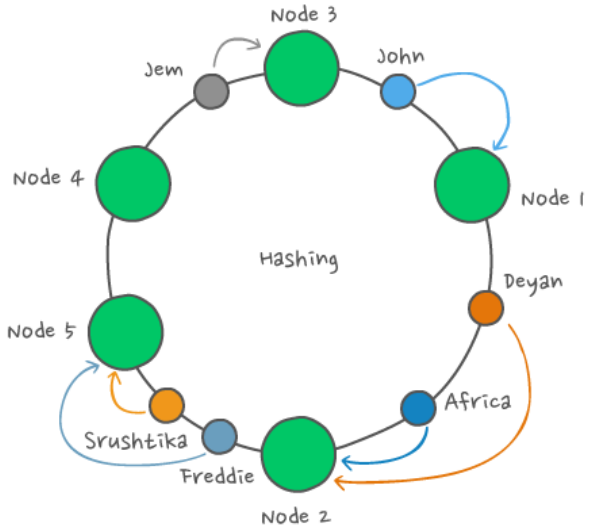
Non-Functional Requirements

- Scalibility
- Increase Concurrent User

Load Balancer

- Balance the traffic
- Spread the traffic across a cluster
- Consistent hashing algorithms

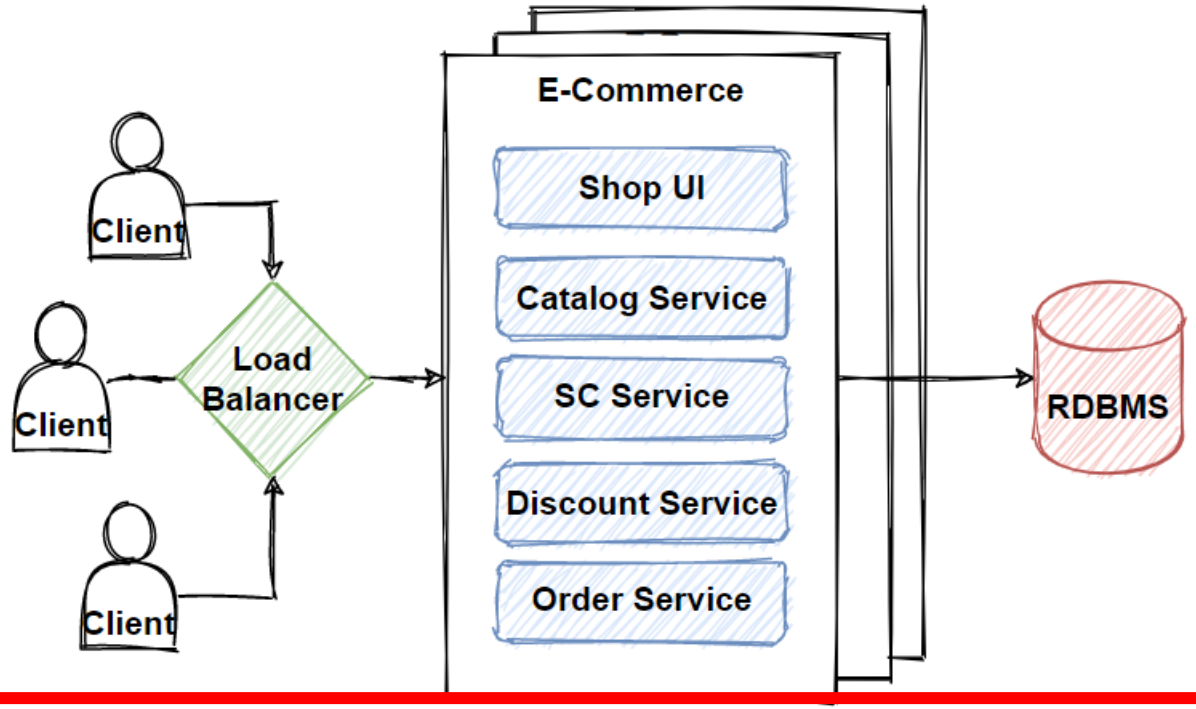




Monolithic Architecture

Principles

- KISS
- YAGNI



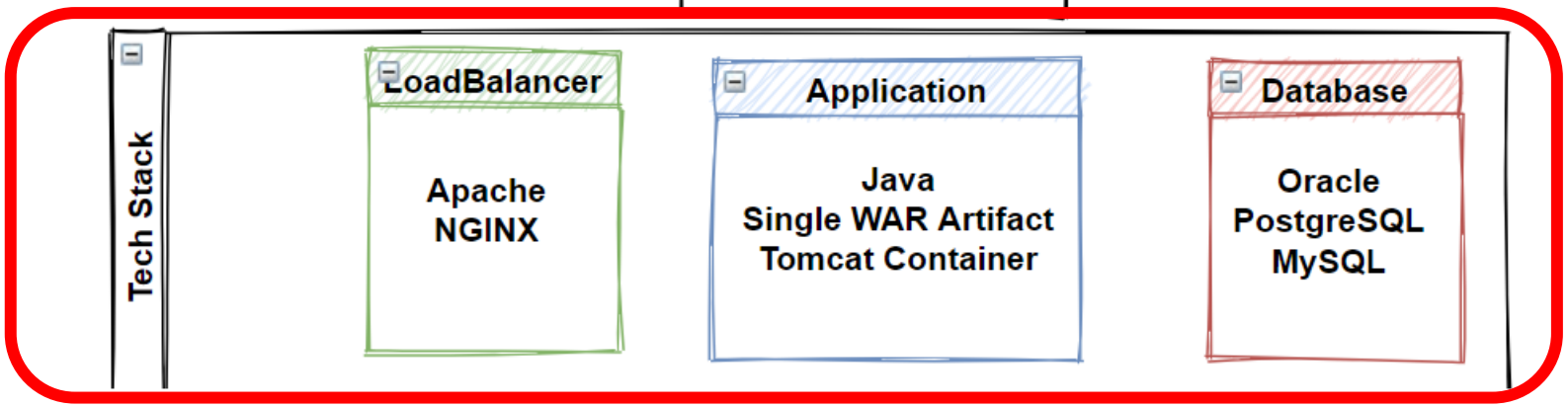
Functional Requirements

- List products
- Filter products as per brand and categories
- Put products into the shopping cart
- Apply coupon for discounts and see the total cost all for all of the items in shopping cart
- Checkout the shopping cart and create an order
- List my old orders and order items history

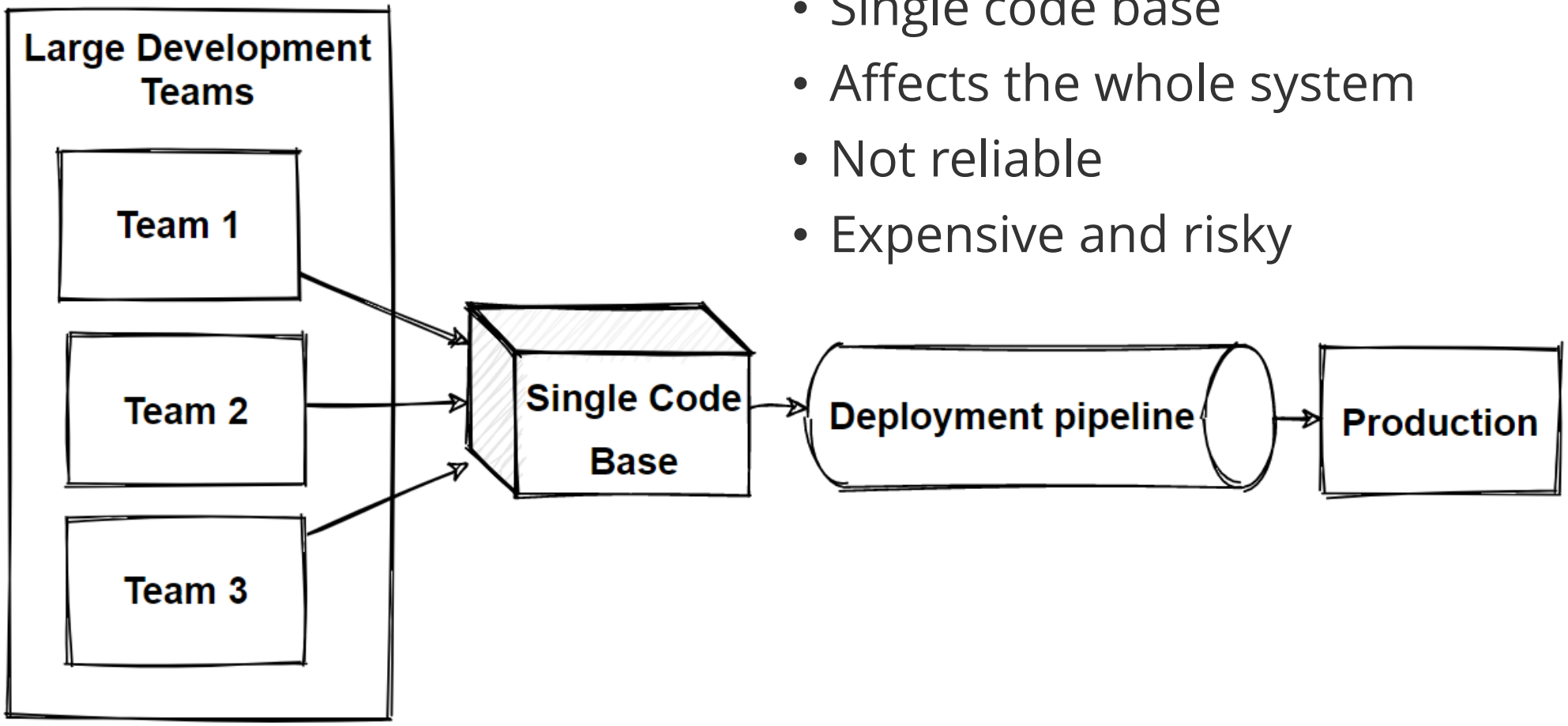
Non-Functional Requirements

- Scalibility
- Increase Concurrent User

Technology Choices



Deployments for Monolithic Architecture

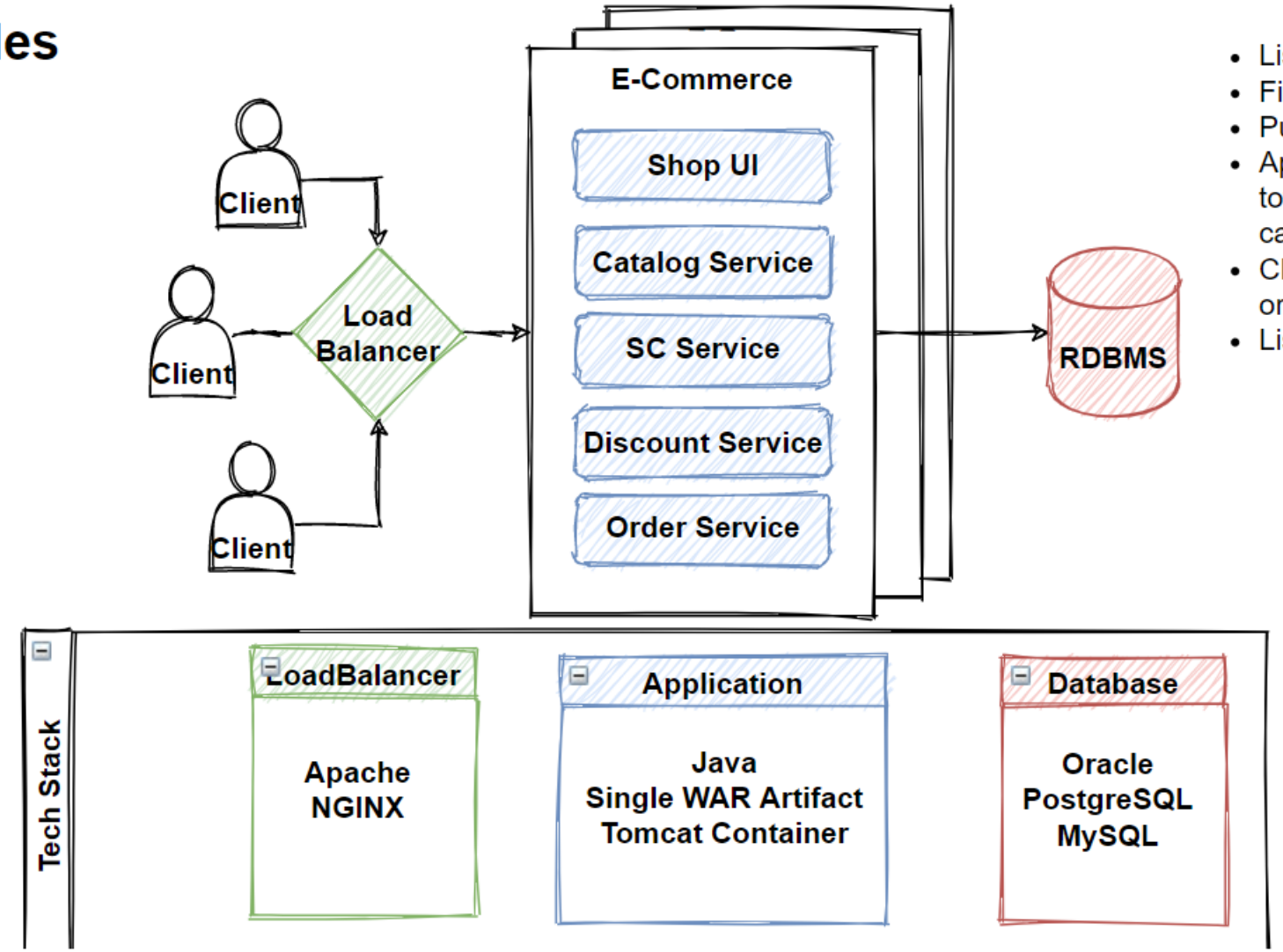


- Single code base
- Affects the whole system
- Not reliable
- Expensive and risky

Monolithic Architecture

Principles

- KISS
- YAGNI



Technology Choices

Functional Requirements

- List products
- Filter products as per brand and categories
- Put products into the shopping cart
- Apply coupon for discounts and see the total cost all for all of the items in shopping cart
- Checkout the shopping cart and create an order
- List my old orders and order items history

Non-Functional Requirements

- Scalibility
- Increase Concurrent User

Section 4
Layered (N-Layer) Architecture

Benefits and Challenges of Layered Architecture

Layered (N-Layer) Architecture

- Presentation tier,

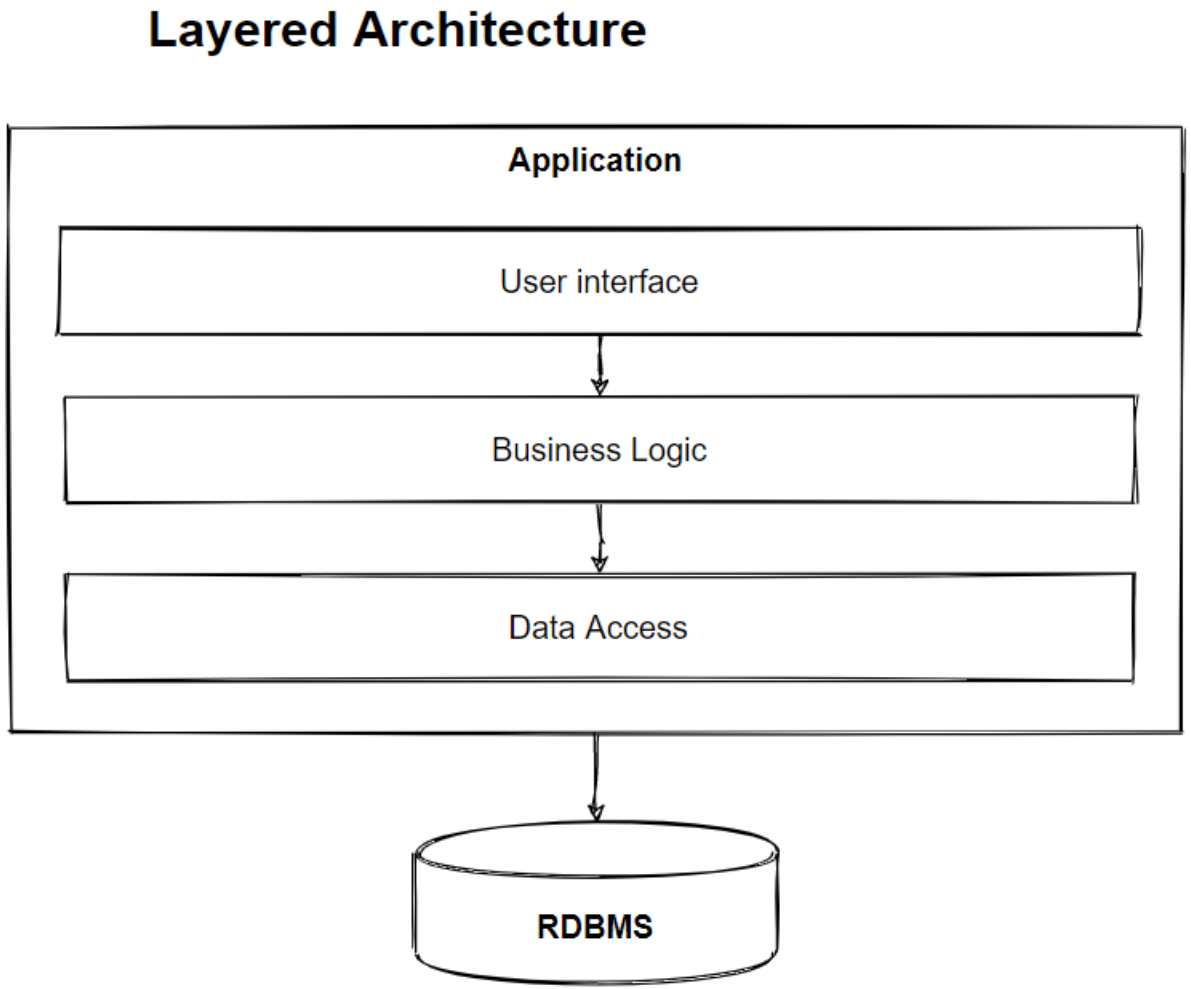
for example, a web app.

- Business tier,

including use case implementations

- A data tier, such as

a SQL database.

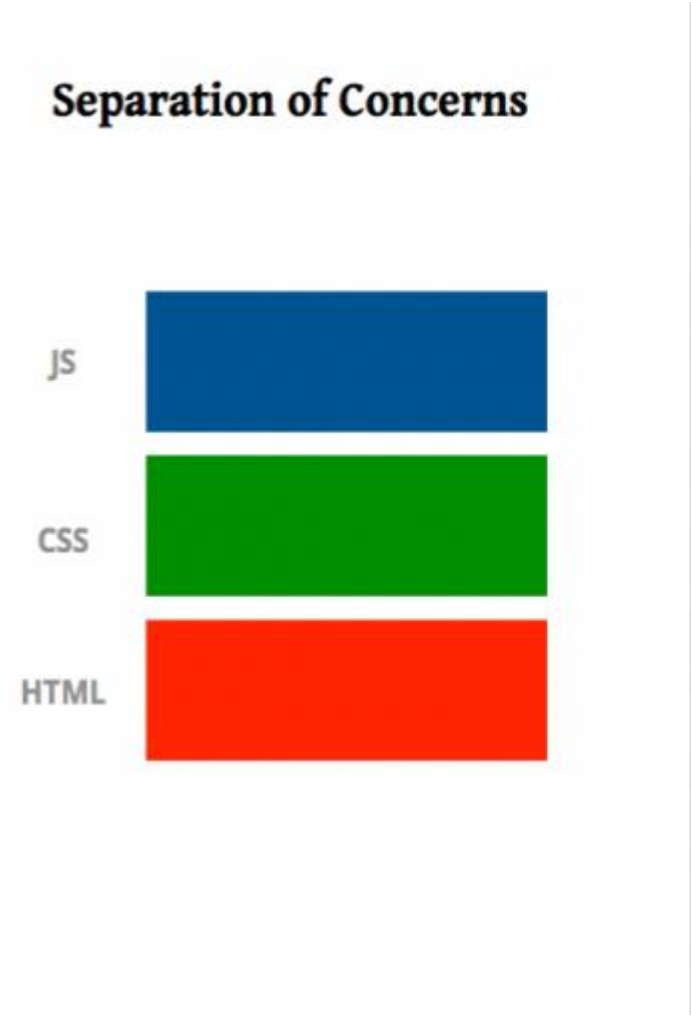


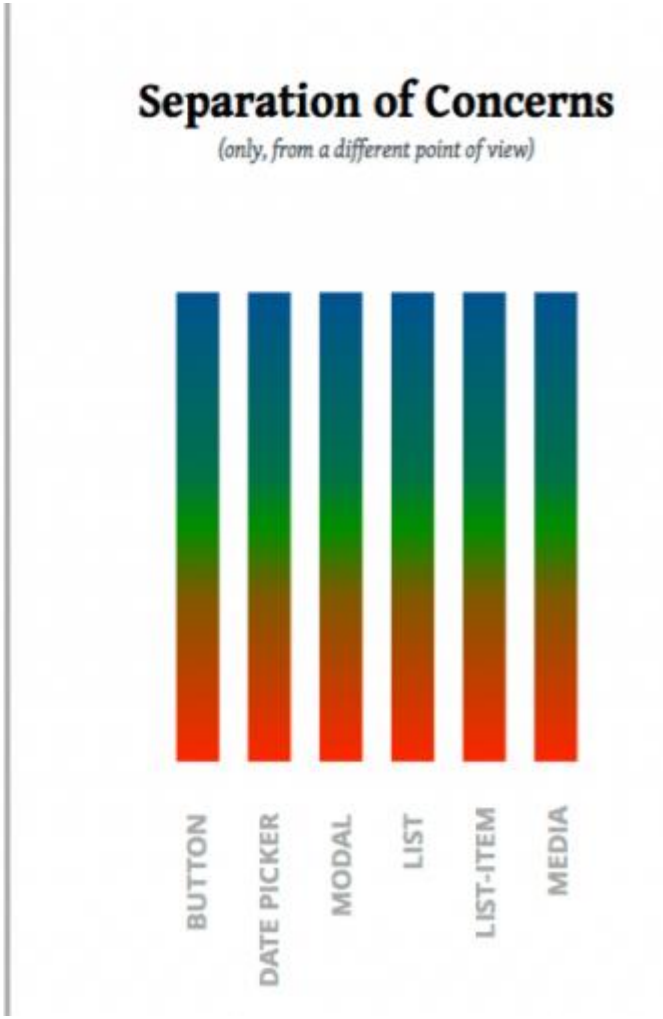
Design principles - Separation of Concerns (SoC)

- Elements in the software should be unique
- Separate responsibilities

- Limits to allocate Responsibilities

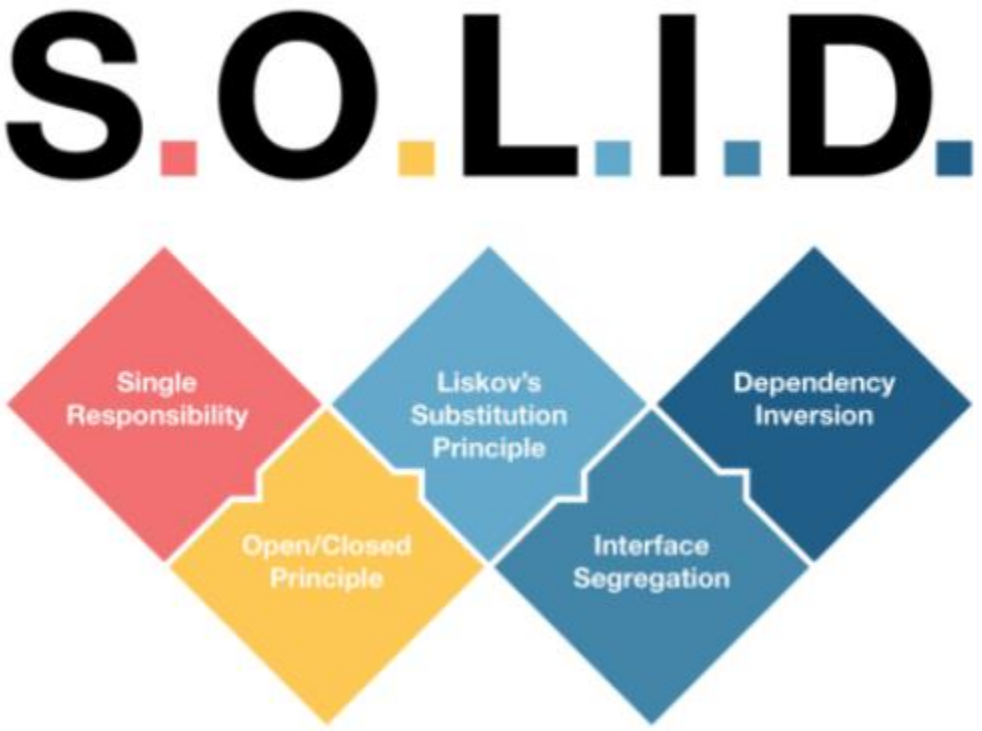
- Low-coupling, high-cohesion





Design principles - SOLID

- Single Responsibility Principle
- Open-Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

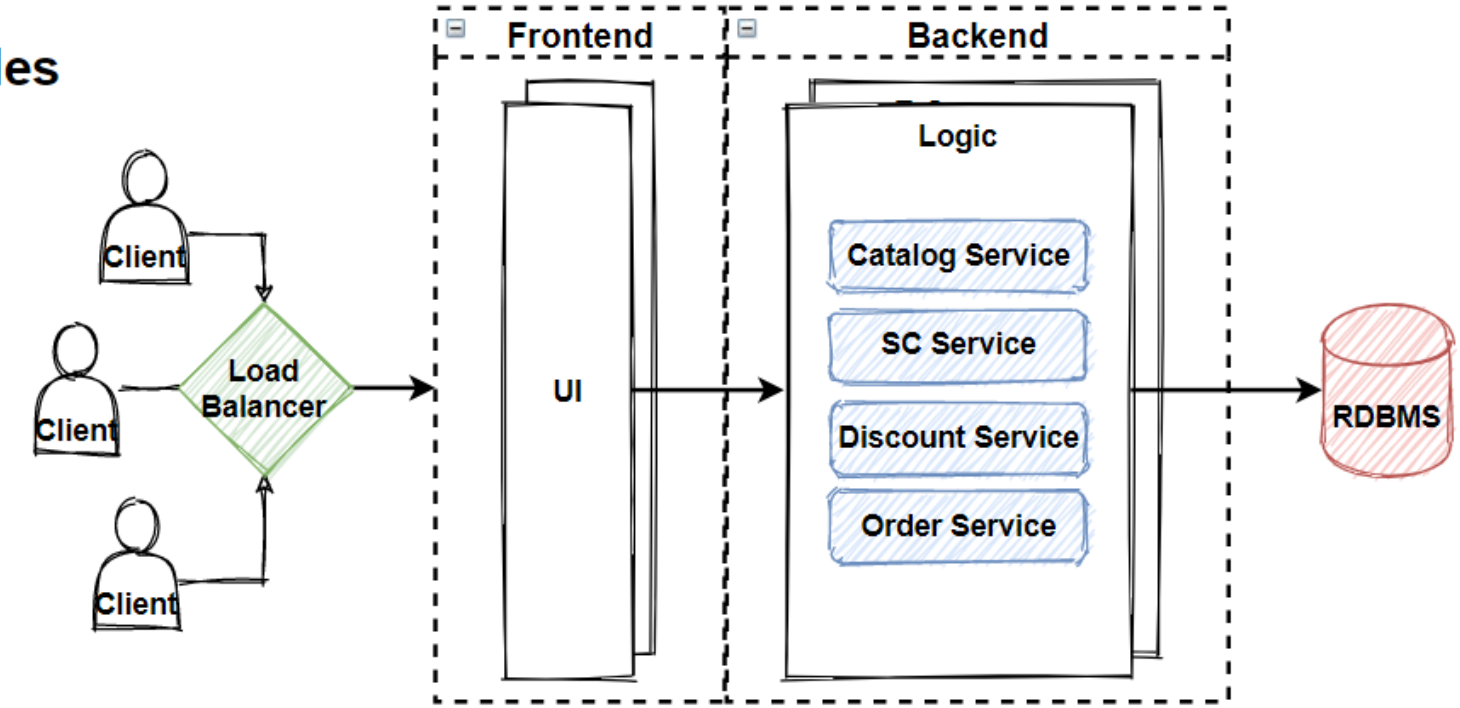


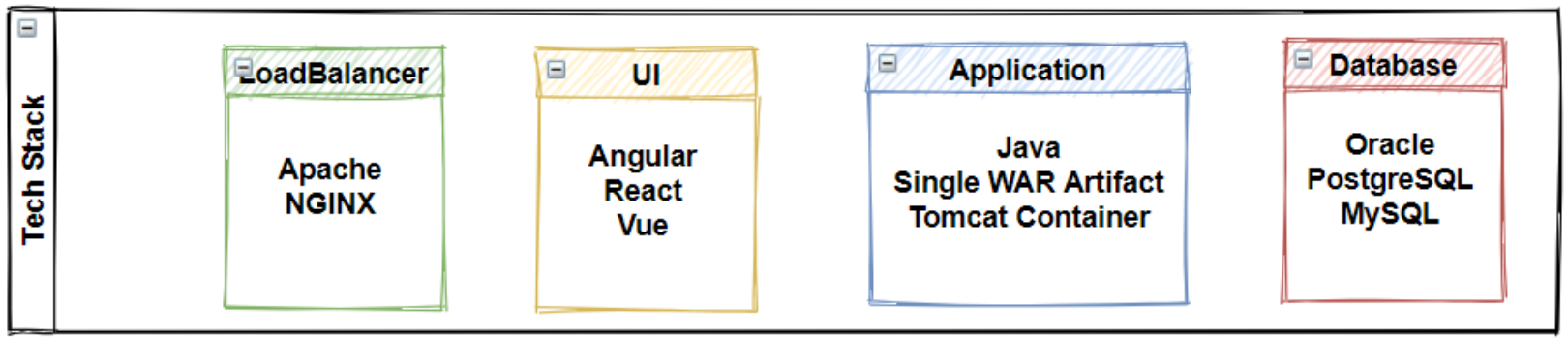
https://medium.com/bgl-tech/what-are-the-solid-design-principles-c61feff33685

Layered Architecture

Principles

- KISS
- YAGNI
- SoC
- SOLID





Functional Requirements

- List products
- Filter products as per brand and categories
- Put products into the shopping cart
- Apply coupon for discounts and see the total cost all for all of the items in shopping cart
- Checkout the shopping cart and create an order
- List my old orders and order items history

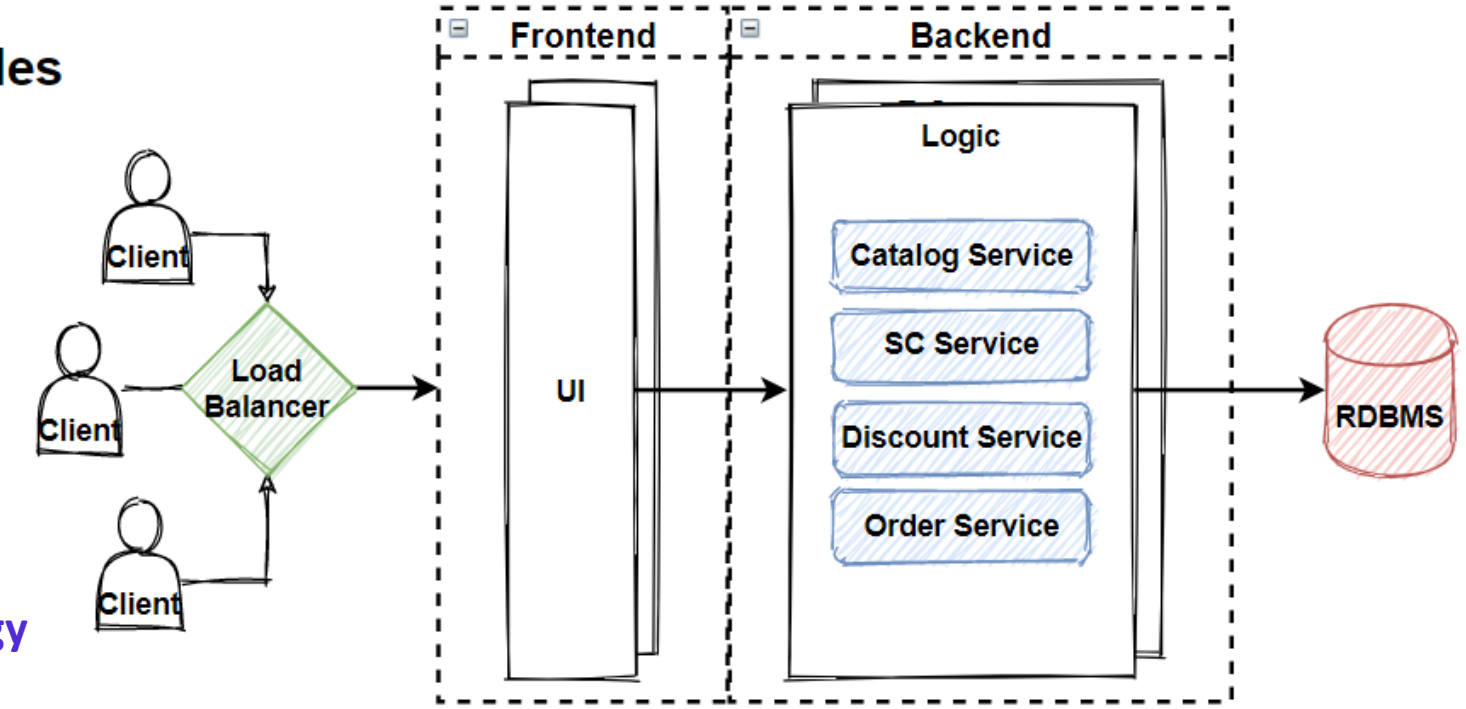
Non-Functional Requirements

- Scalibility
- Increase Concurrent User
- Maintainability

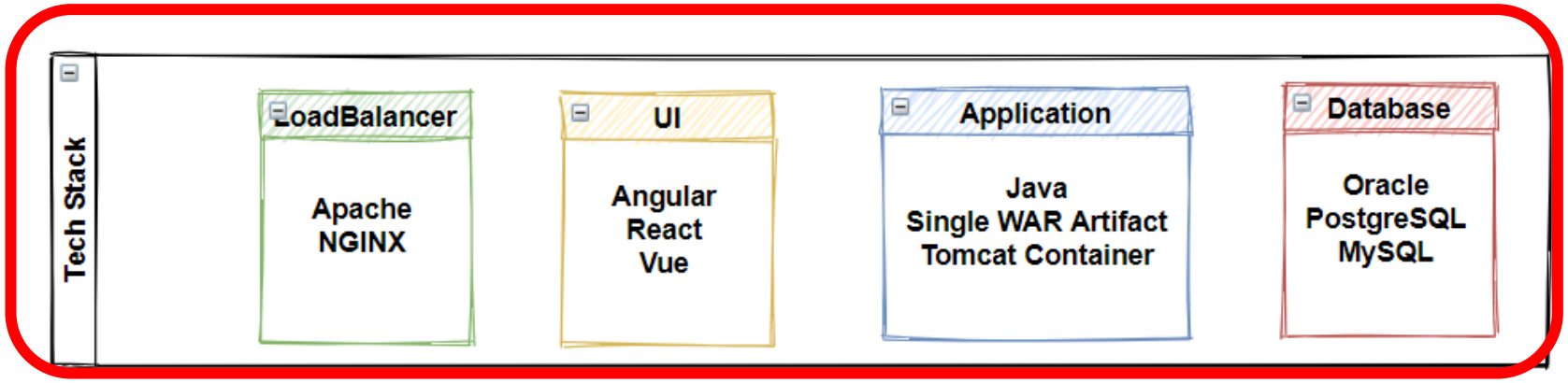
Layered Architecture

Principles

- KISS
- YAGNI
- SoC
- SOLID



Technology Choices



Functional Requirements

- List products
- Filter products as per brand and categories
- Put products into the shopping cart
- Apply coupon for discounts and see the total cost all for all of the items in shopping cart
- Checkout the shopping cart and create an order
- List my old orders and order items history

Non-Functional Requirements

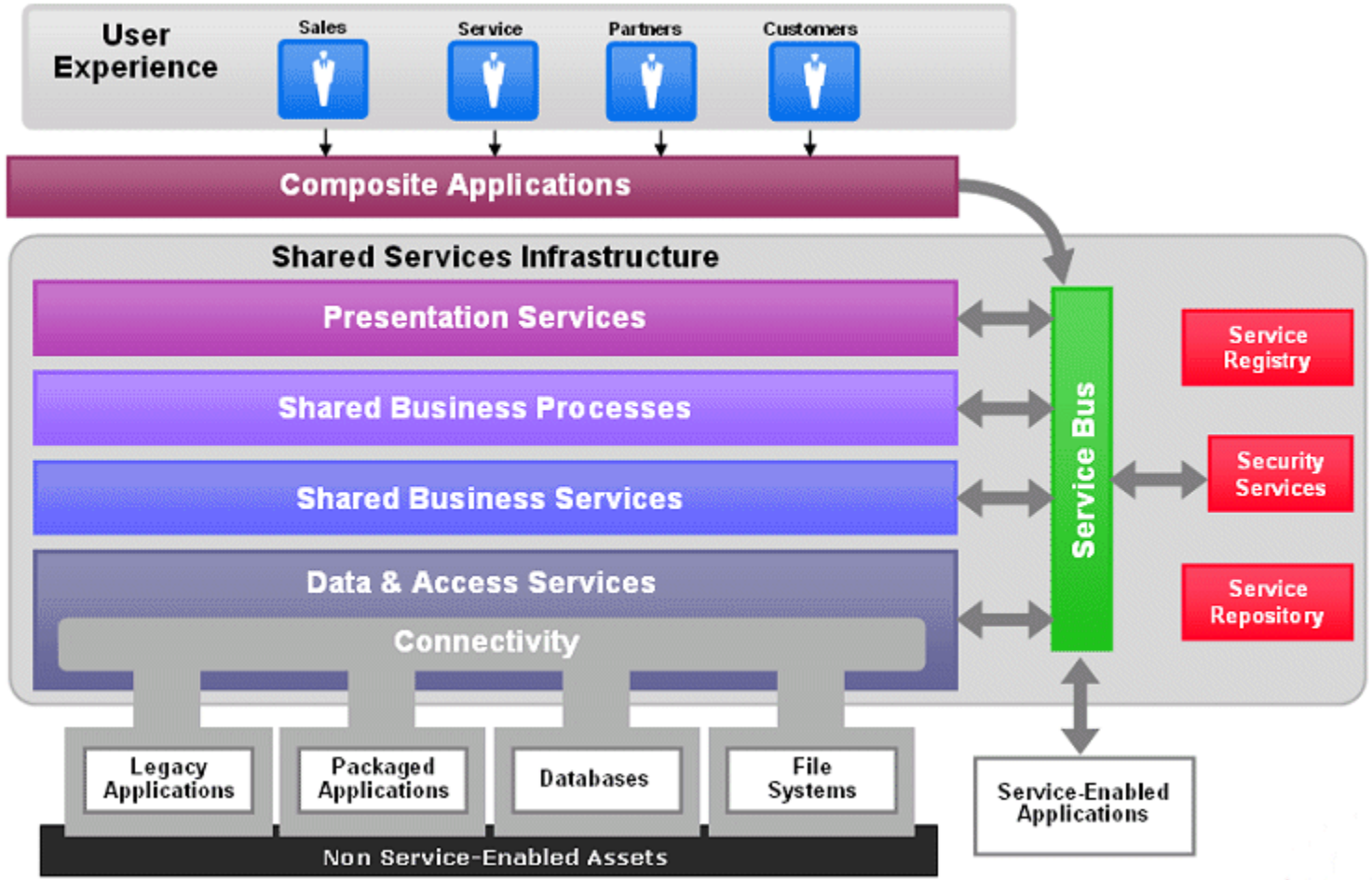
- Scalibility
- Increase Concurrent User
- Maintainability

Section 5
Service-Oriented Architecture

Benefits and Challenges of SOA Architecture

Service-Oriented Architecture

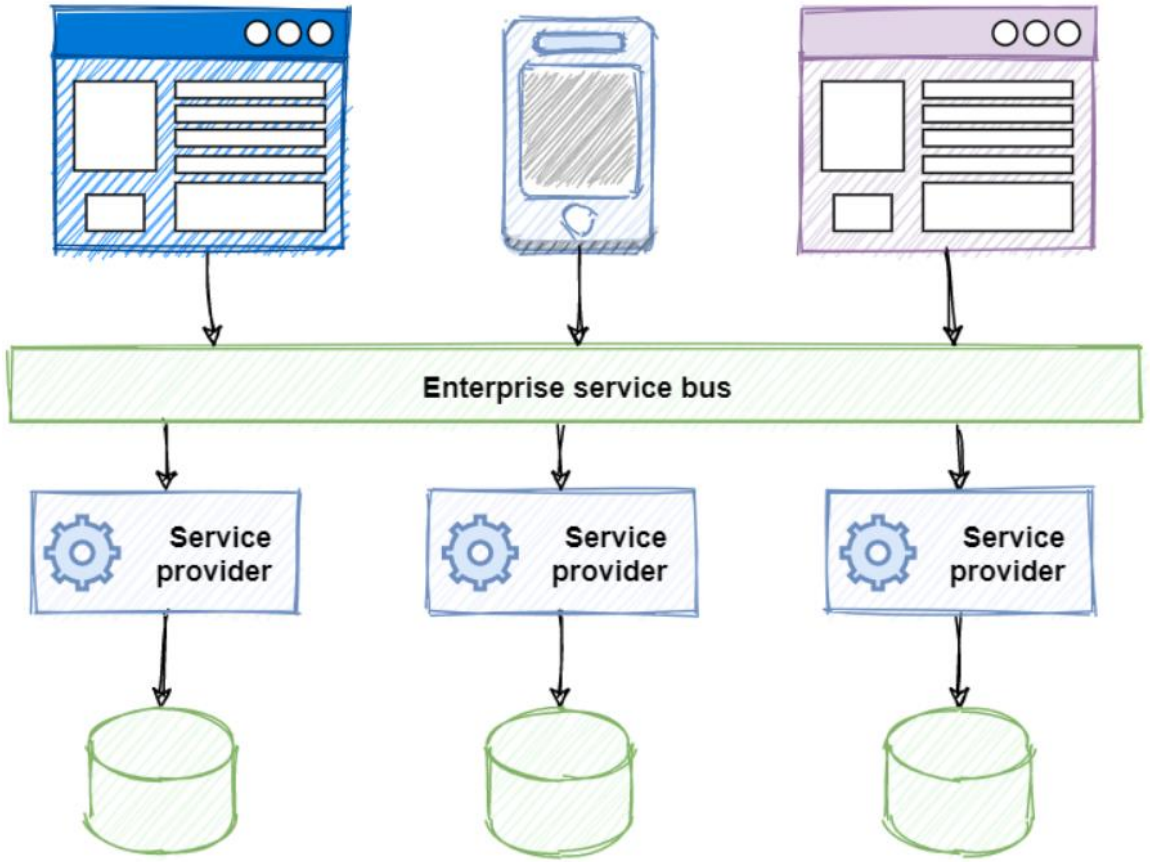
- Service components
- Communicates over the network
- Converged service Infrastructure
- Enterprise applications



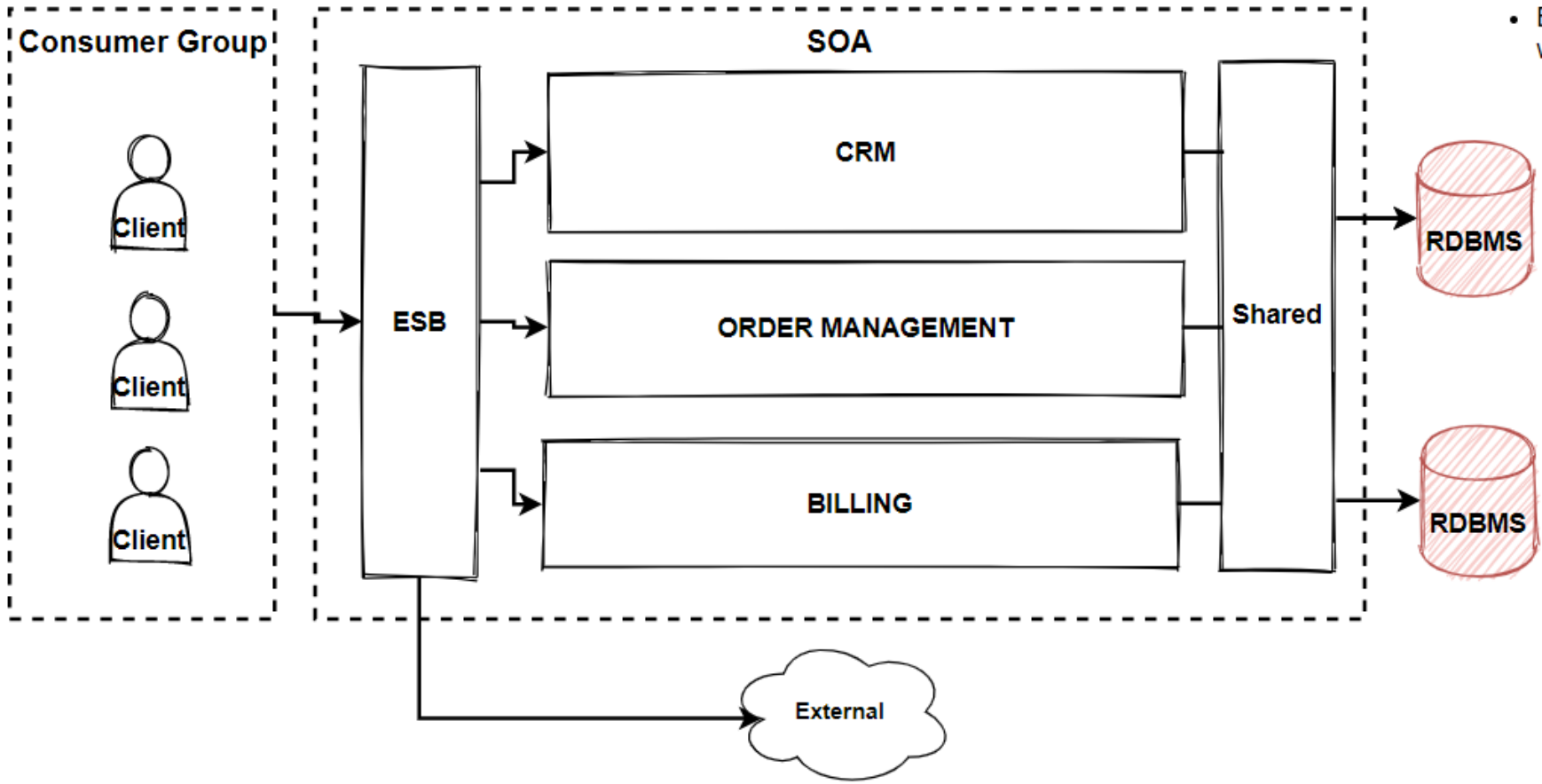
https://docs.oracle.com/cd/E13171_01/alsb/docs30/concepts/introduction.html

Architectural Design patterns - Enterprise Service Bus (ESB)

- Integrations between applications
- Transformations of data models
- Middleware messaging components
- Service orchestration
- Increased complexity and introduced bottlenecks



Service-Oriented Architecture

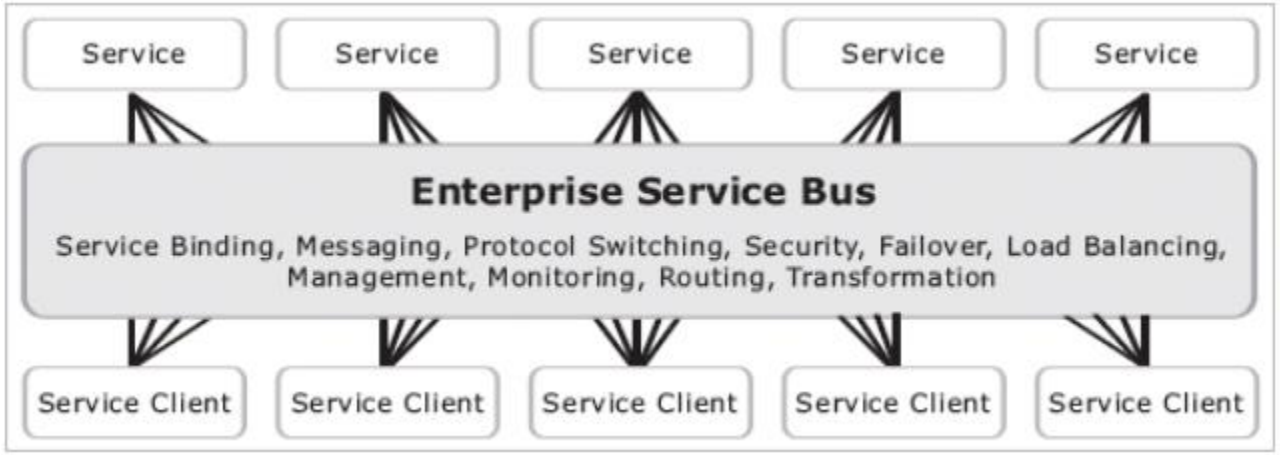


Key Features

- SOAP WS Integrations
- Big Monolithic Applications with using Shared Services

Communication in SOA

- Enterprise service bus - ESB systems.
- Handles connectivity and messaging, performs routing
- Inter-service communication
- SOAP-based web services
- SOAP, WSDL, and XSD



https://docs.oracle.com/cd/E13171_01/alsb/docs30/concepts/introduction.html

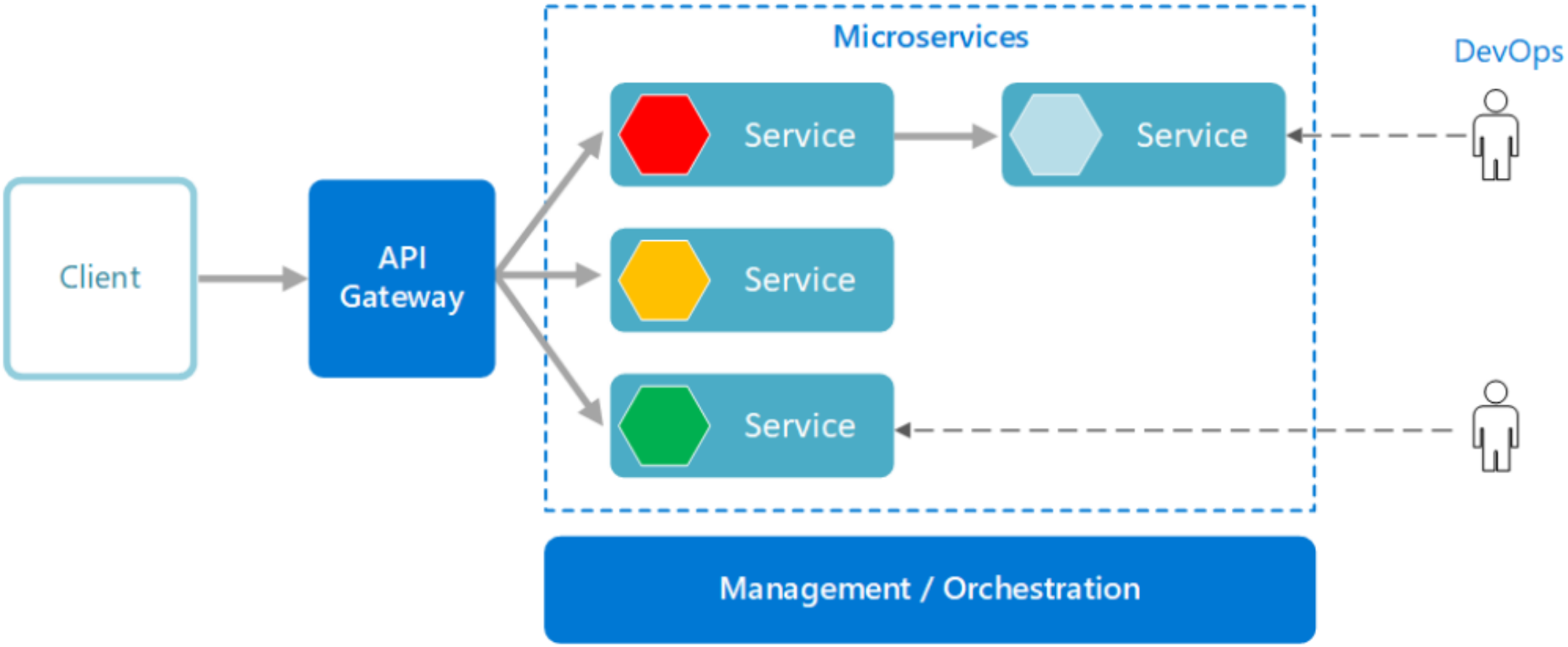
Comparing SOA and Microservices

- Inter-service communication due to distributed services
- SOA, using Enterprise Service Bus
- Microservices, using message brokers
- SOA using Global data model
- Microservices has polyglot databases
- Size of the services different
- SOA increased complexity and introduced bottlenecks
- ESB middleware expensive

Section 6
Microservices Architecture

Benefits and Challenges of Microservices Architecture

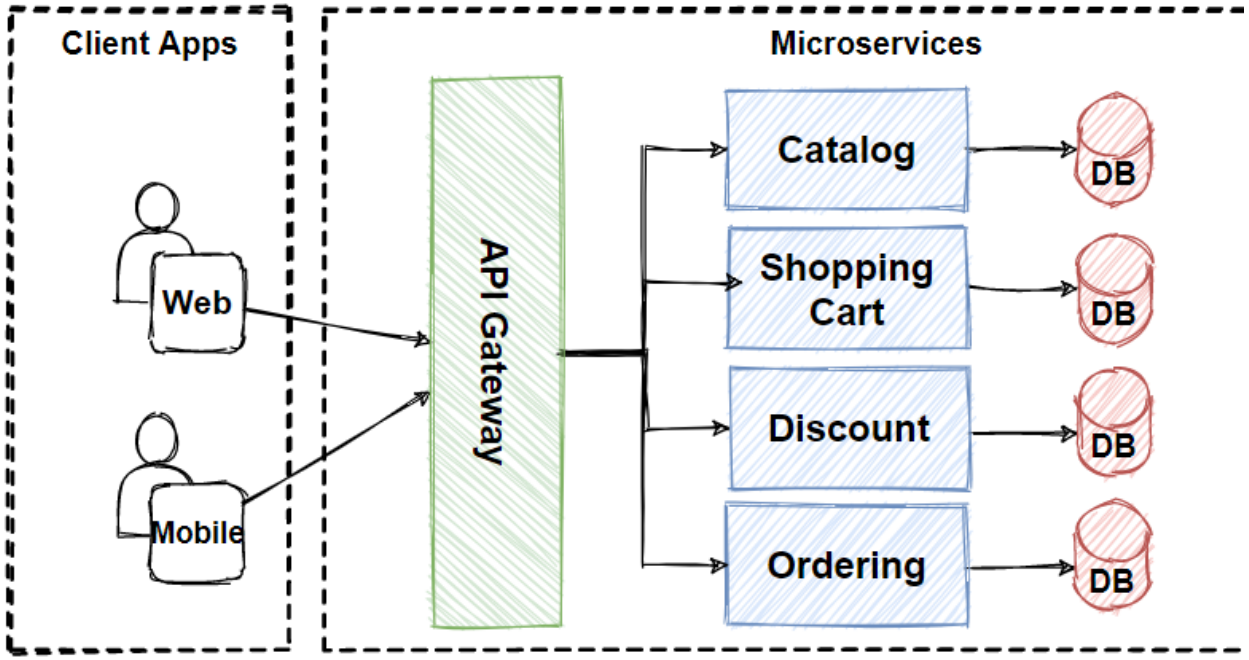
What are Microservices ?



What are Microservices ? - 2

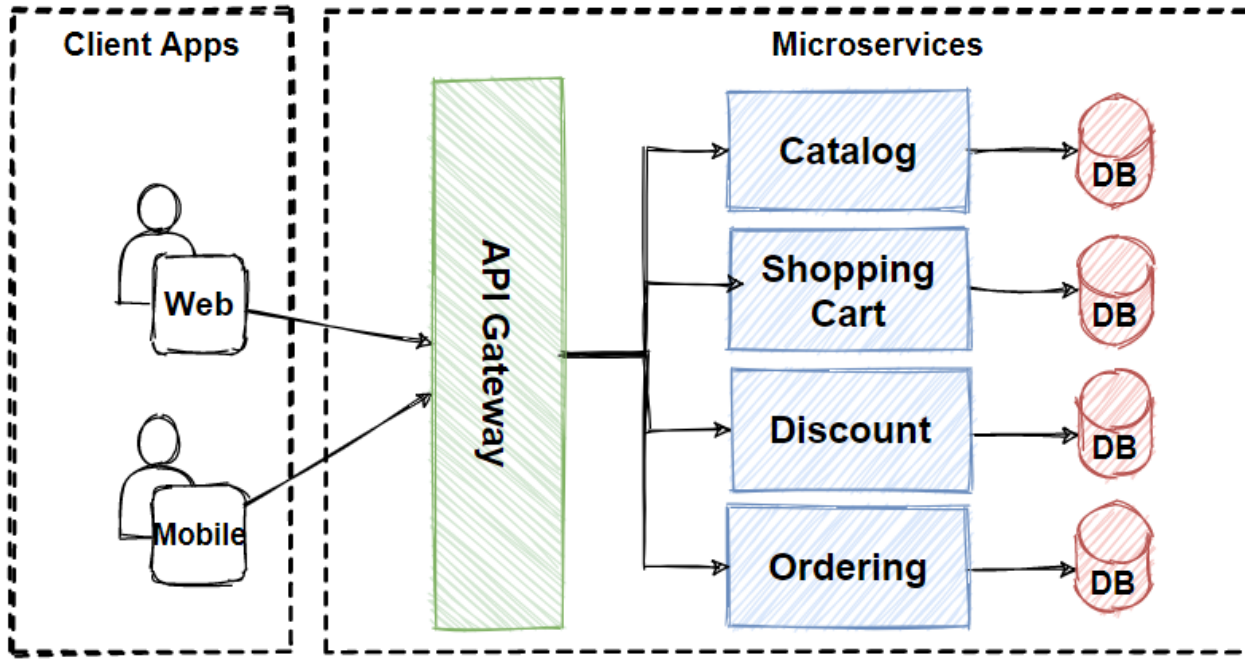
- Small services
- Running in its own process
- Communicating with APIs
- Independently deployable
- Different programming languages
- Own technology stack
- Decouple microservices

with bounded context



Microservices Characteristics

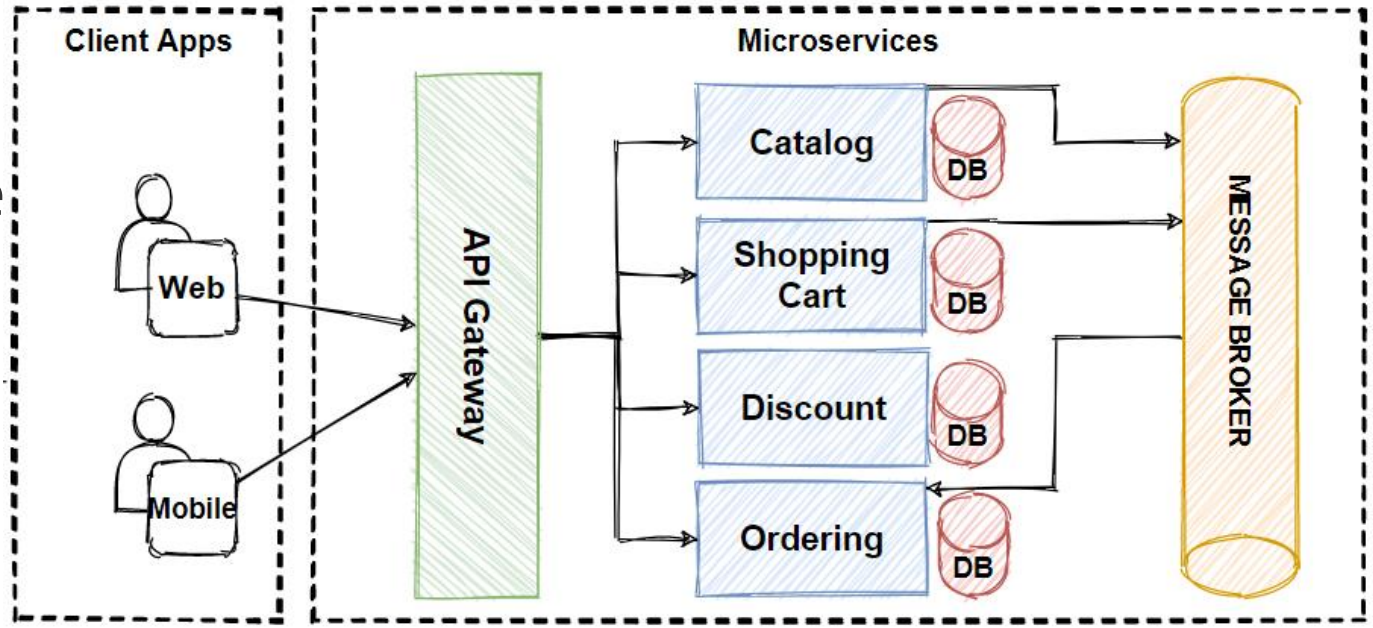
- Componentization via Services
- Organized by Business Capabilities
- Products not Projects
- Smart endpoints and dumb pipes
- Decentralized Governance
- Decentralized Data Management
- Infrastructure Automation
- Design for failure



https://martinfowler.com/articles/microservices.html

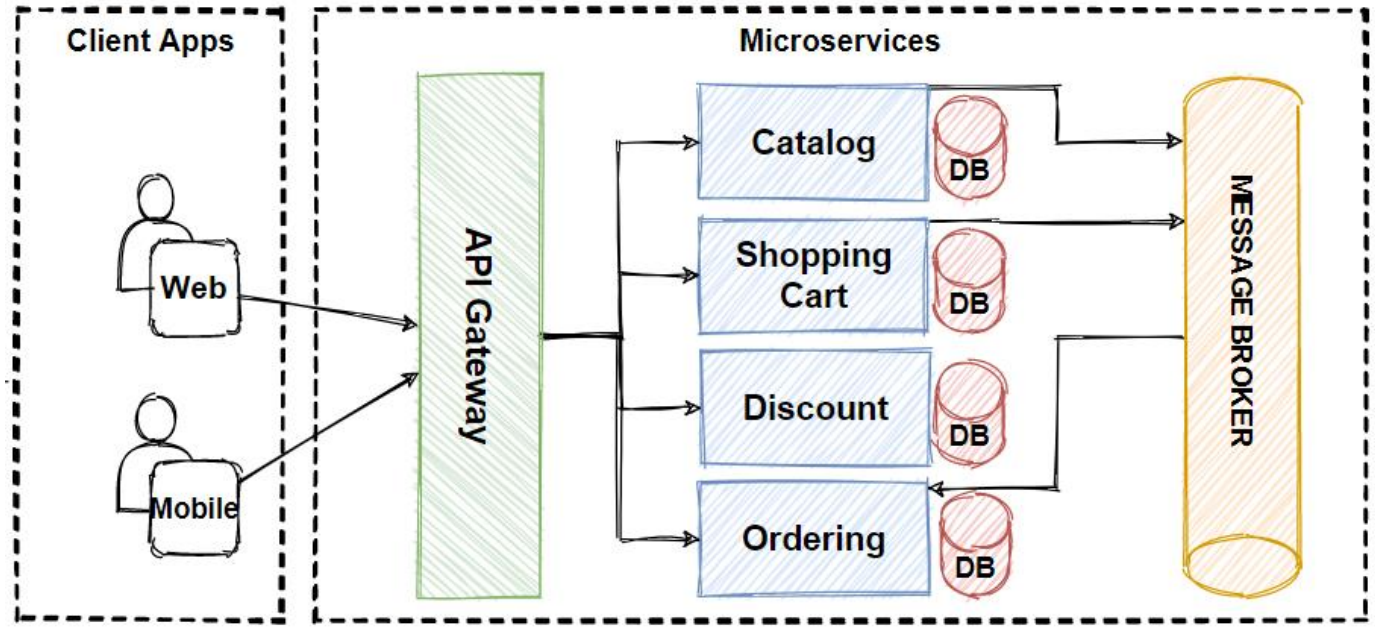
Benefits of Microservices Architecture

- Agility
- Small, focused teams
- Small and separated code base
- Right tool for the job
- Adapting Technology changes
- Fault isolation
- Scalability
- Data isolation



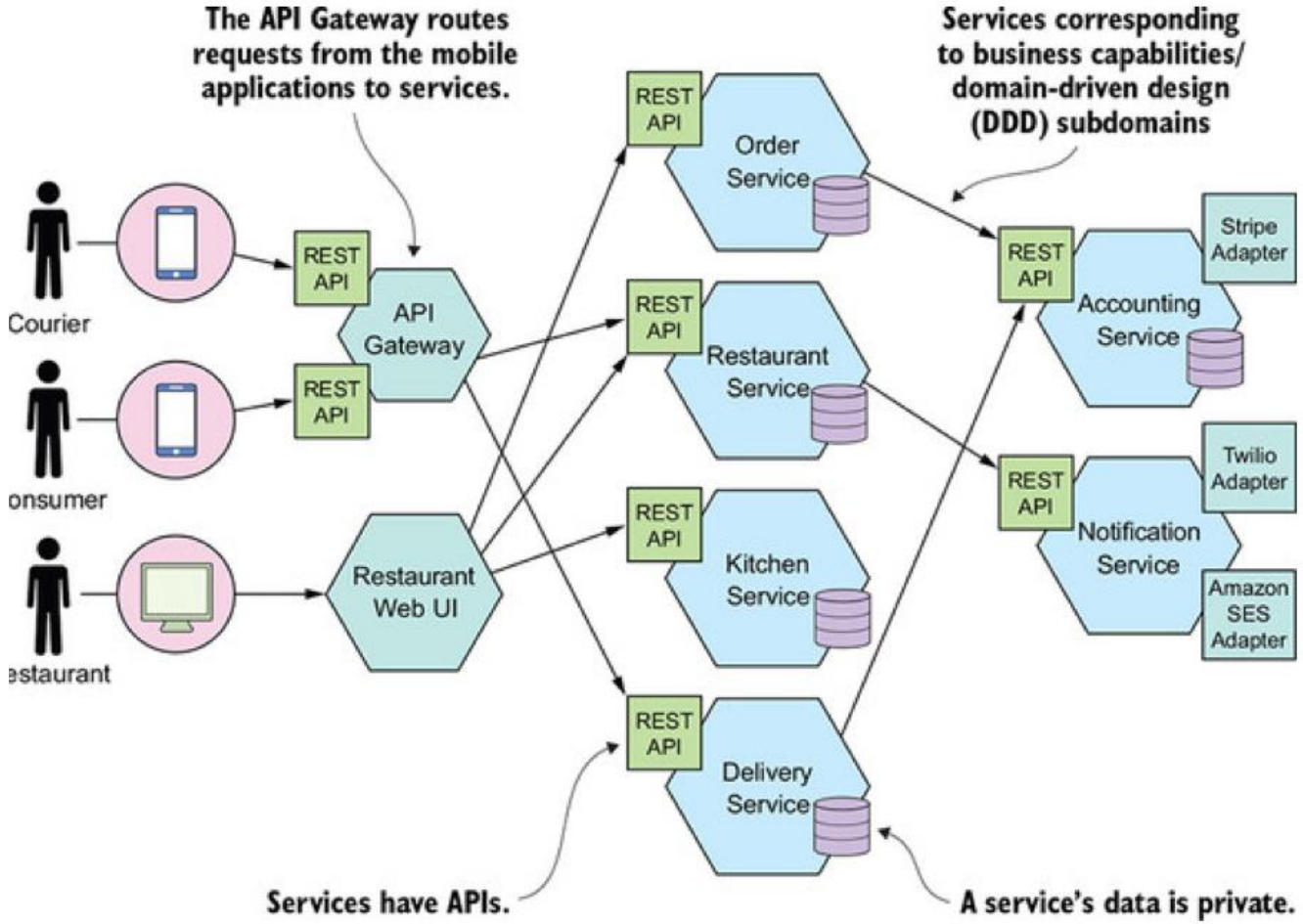
Challenges of Microservices Architecture

- Complexity
- Network problems and Latency
- Development and testing
- Data integrity

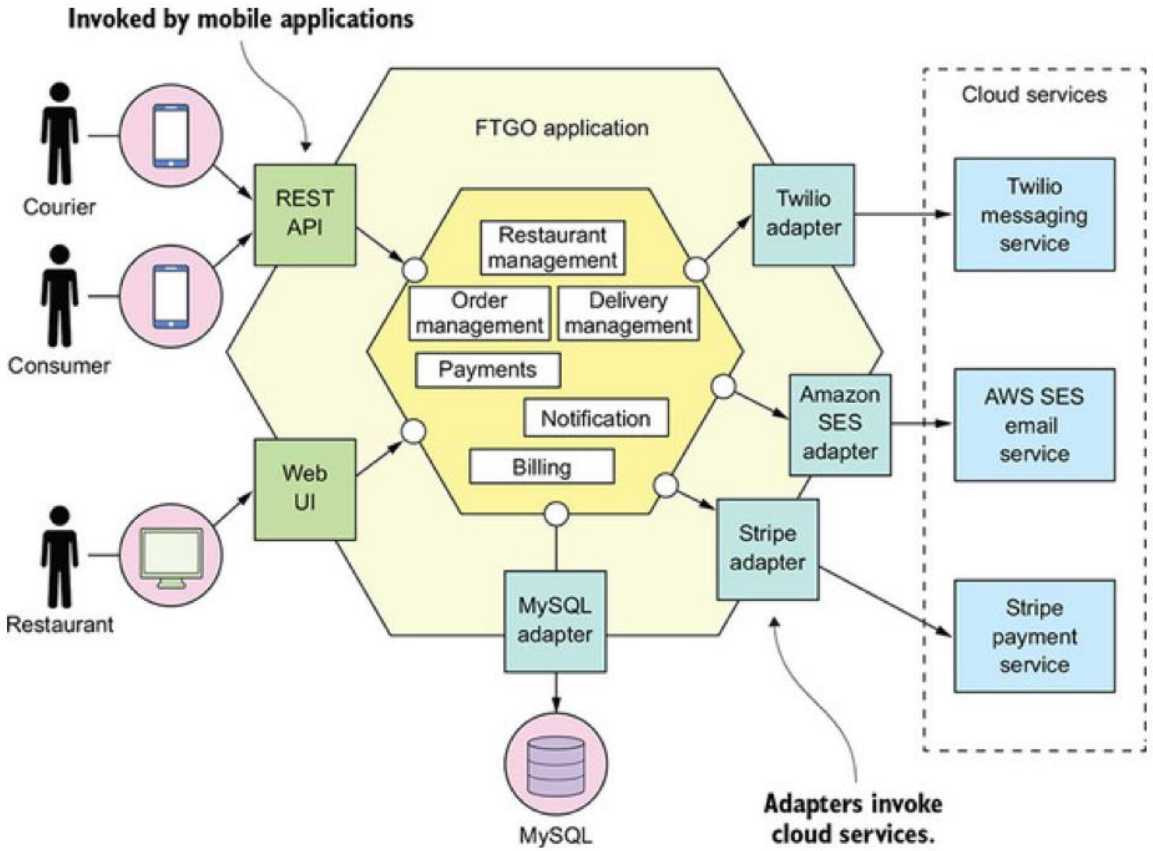


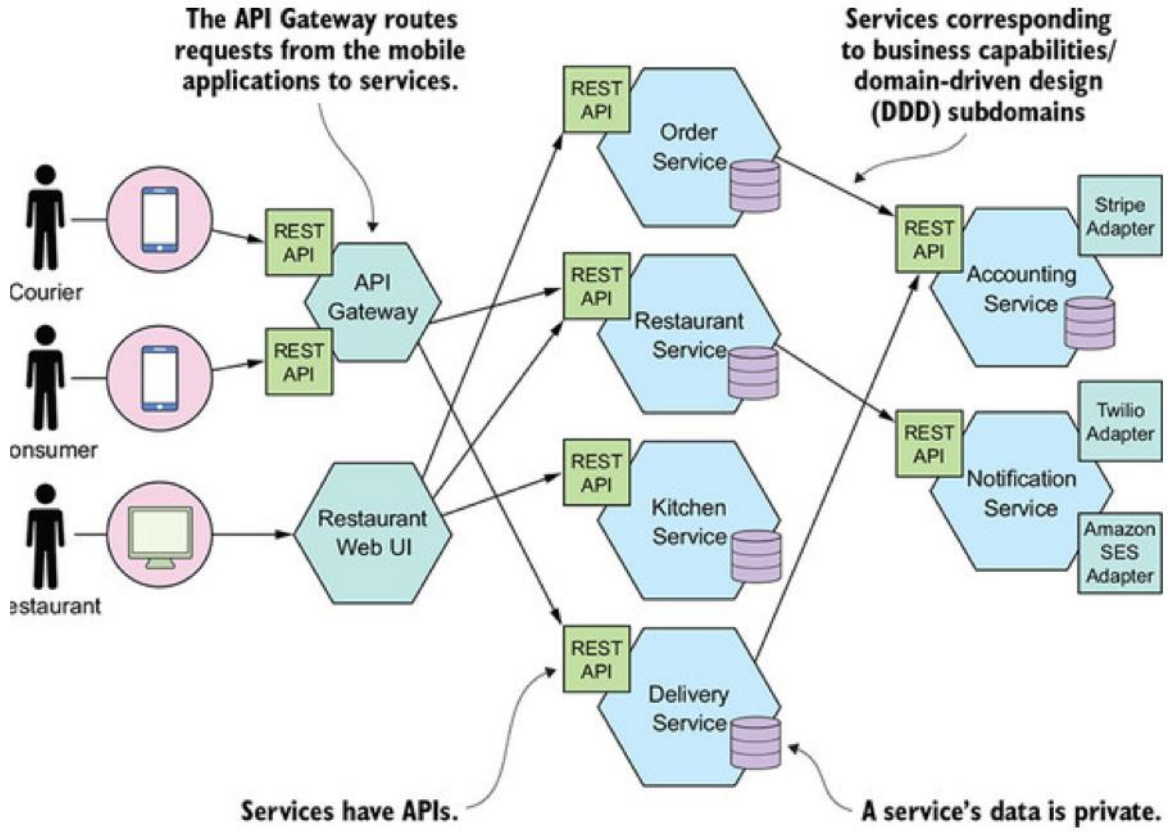
Microservices Architecture Pros-Cons

- Independent Services
- Better scalability
- Technology Diversity
- Agility
- Small, focused teams
- Challenge of management and traceability

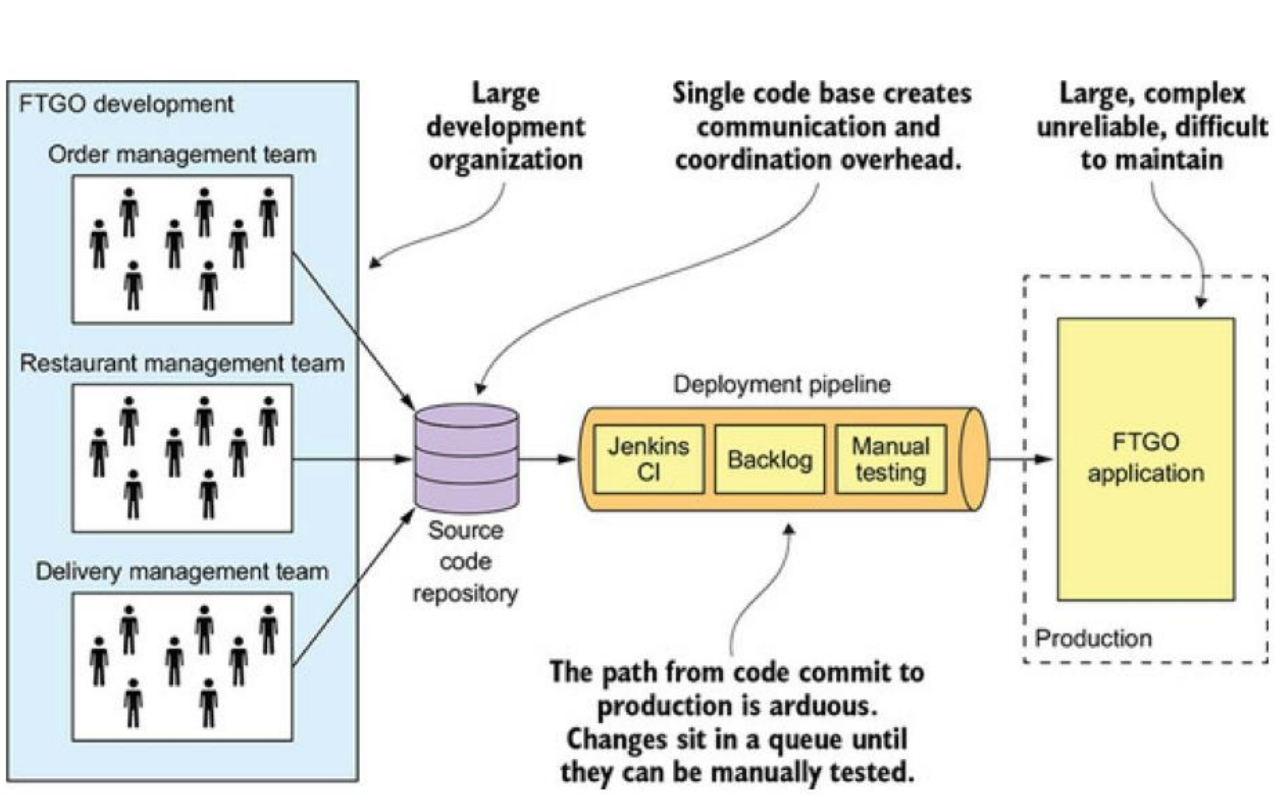


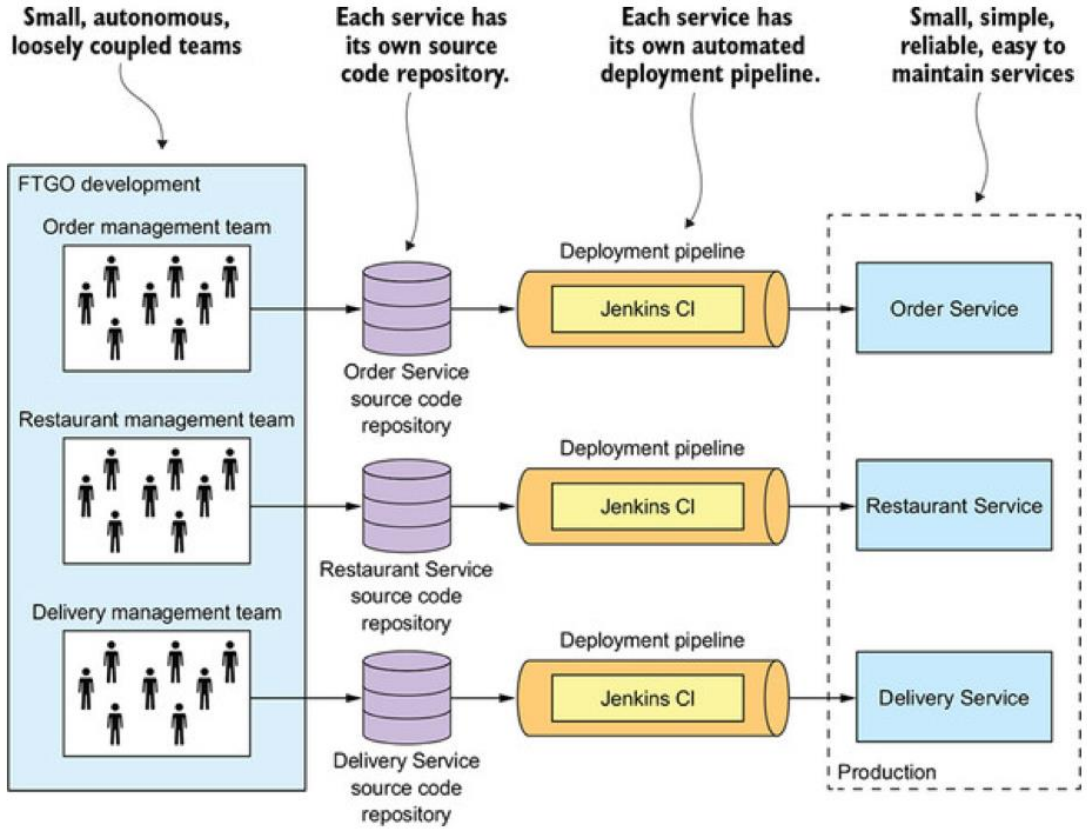
Architecture Comparison





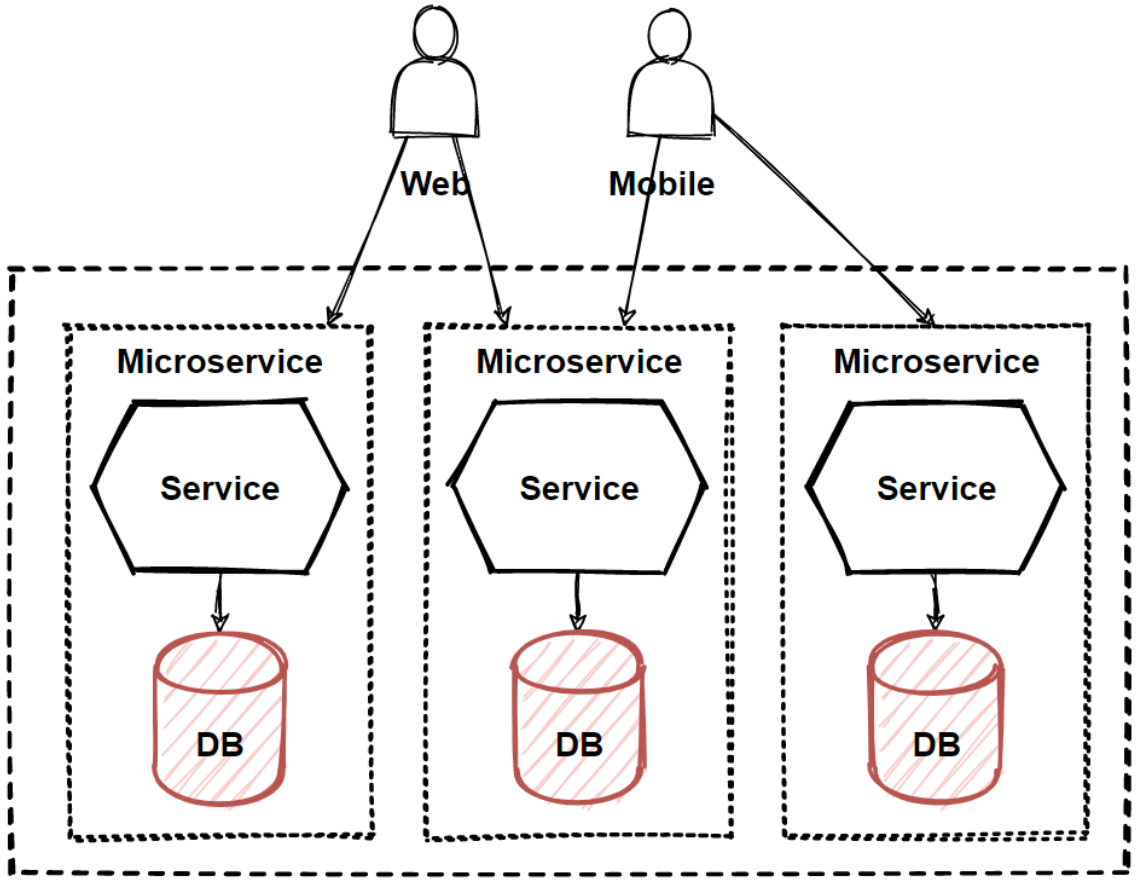
Deployment Comparison



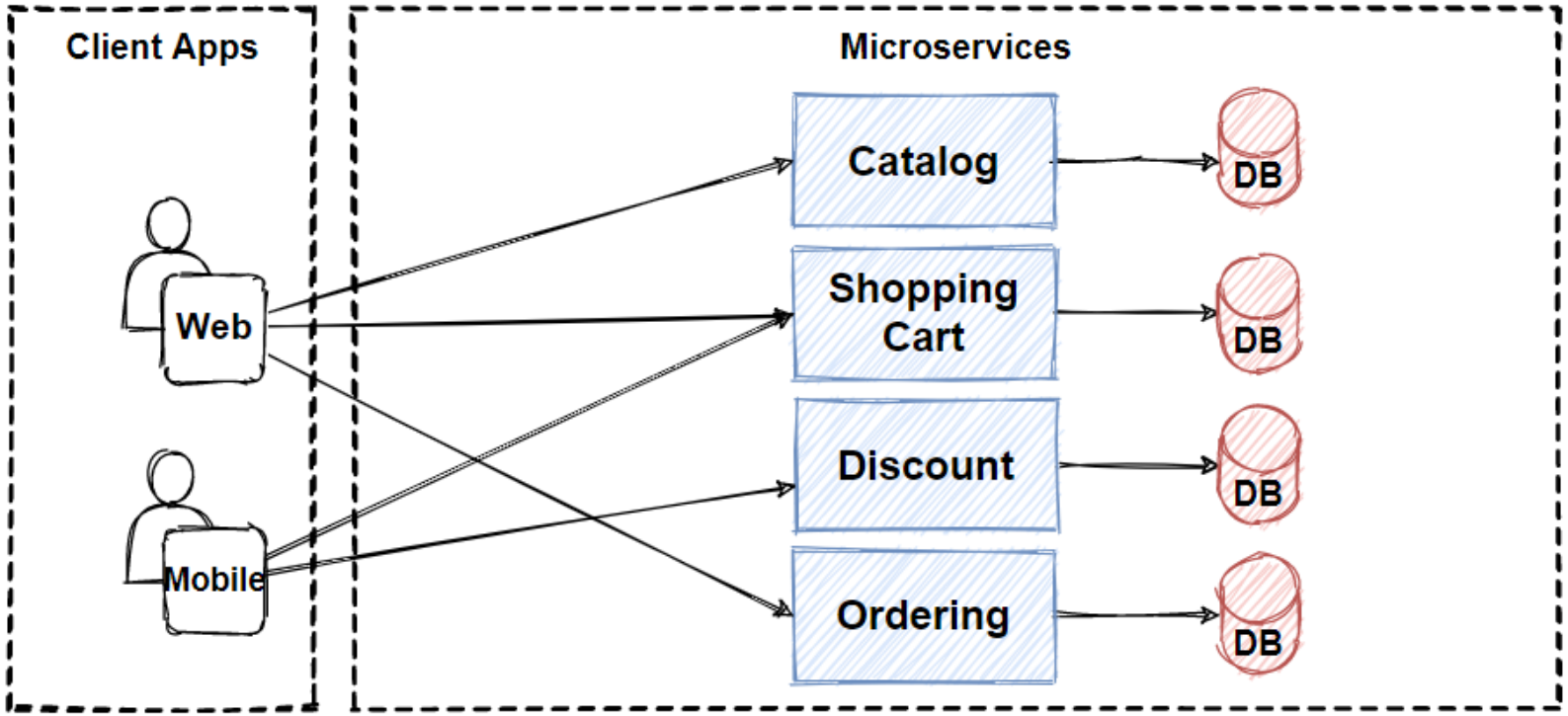


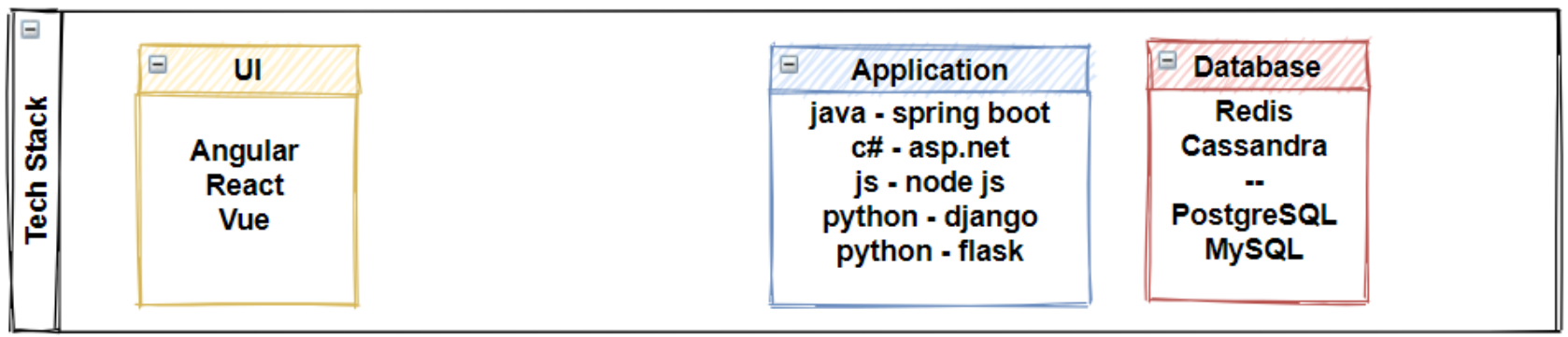
The Database-per-Service Pattern

- Loose coupling of services
- Own databases
- Polygot persistence
- Can’t be accessed directly
- Scale independently
- Data encapsulated within the service
- Not affect to other services

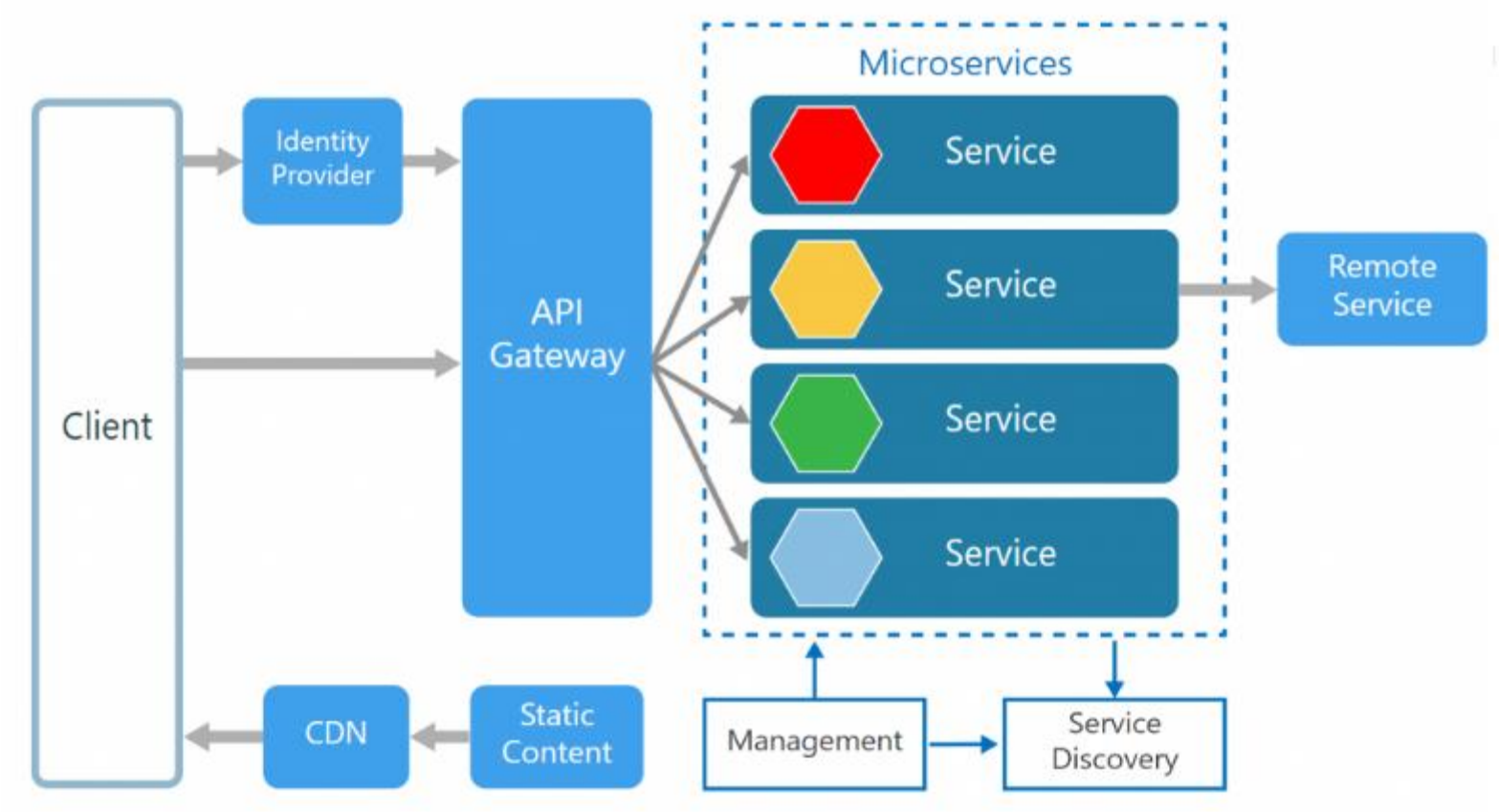


Microservices Architecture



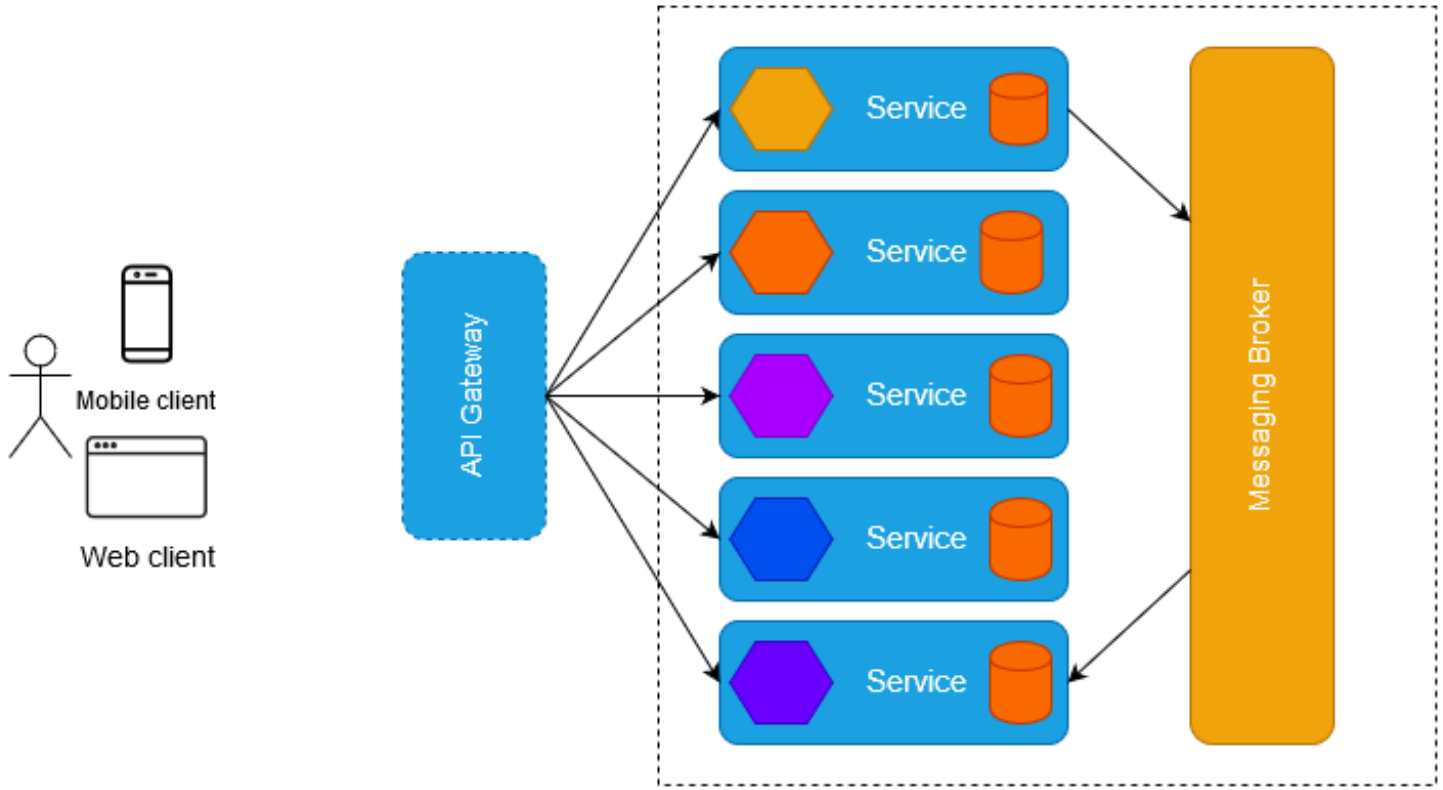


Reference Microservice Architectures



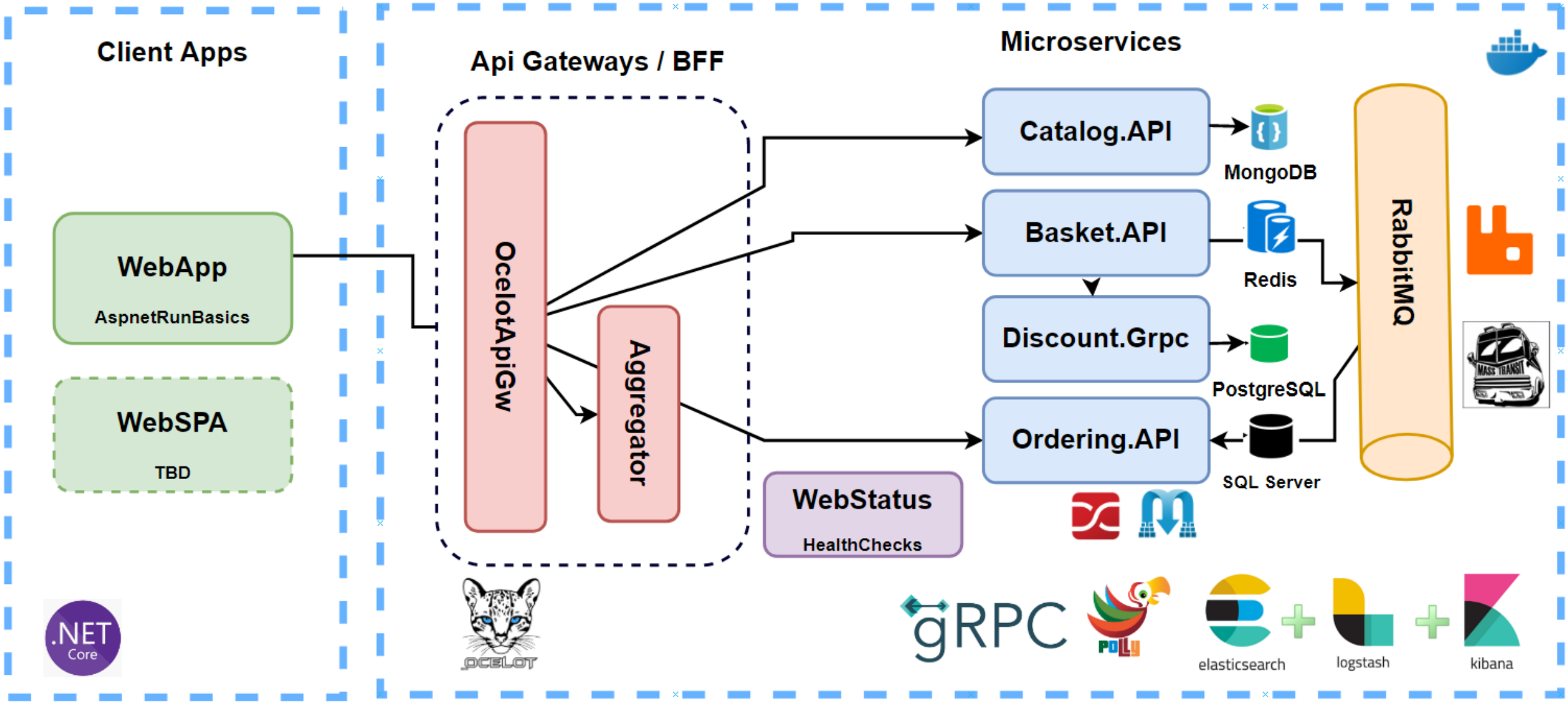
https://www.futurefundamentals.com/what-is-microservices-architecture/

Reference Microservice Architectures 2

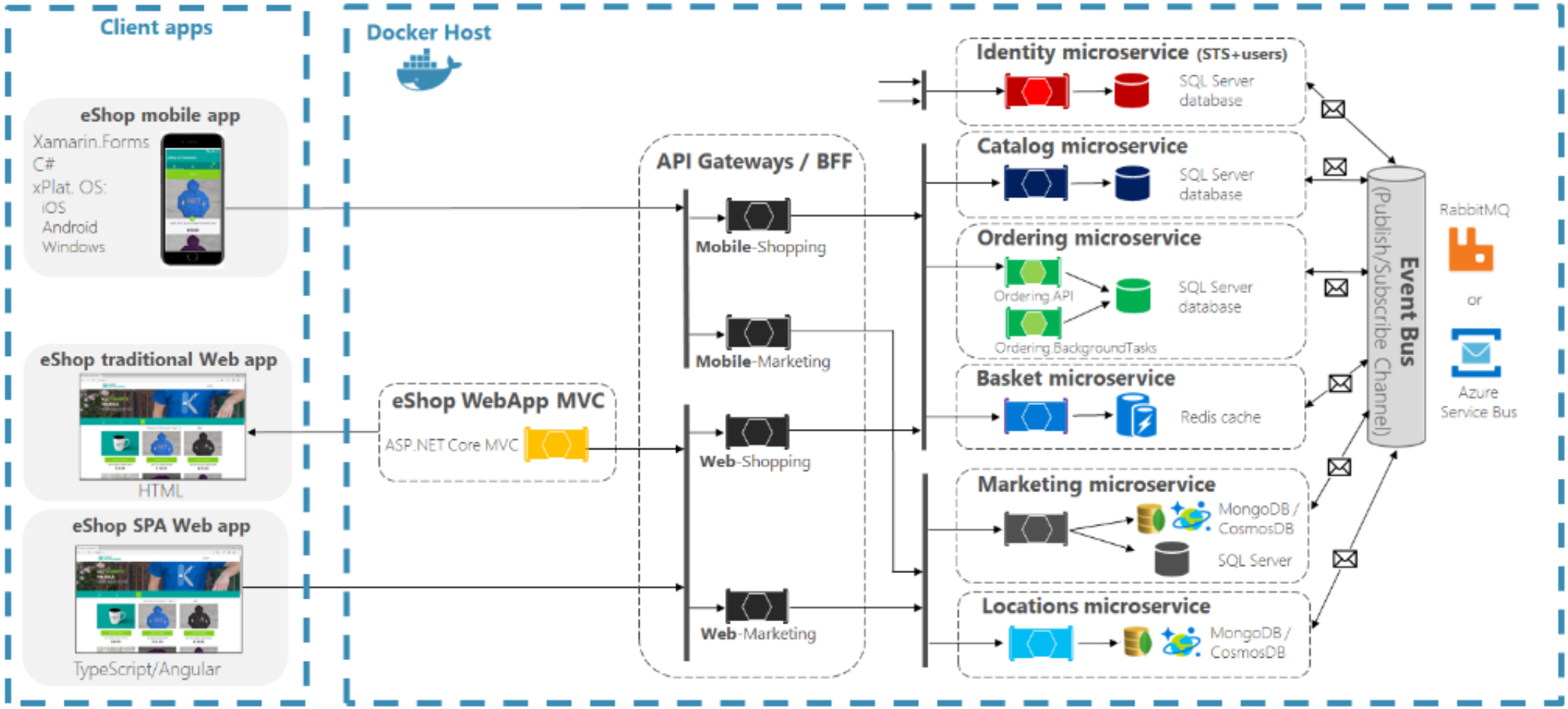


https://feras.blog/microservices-architecture-to-be-or-not-to-be/

Reference Microservice Architectures 3



Reference Microservice Architectures 4



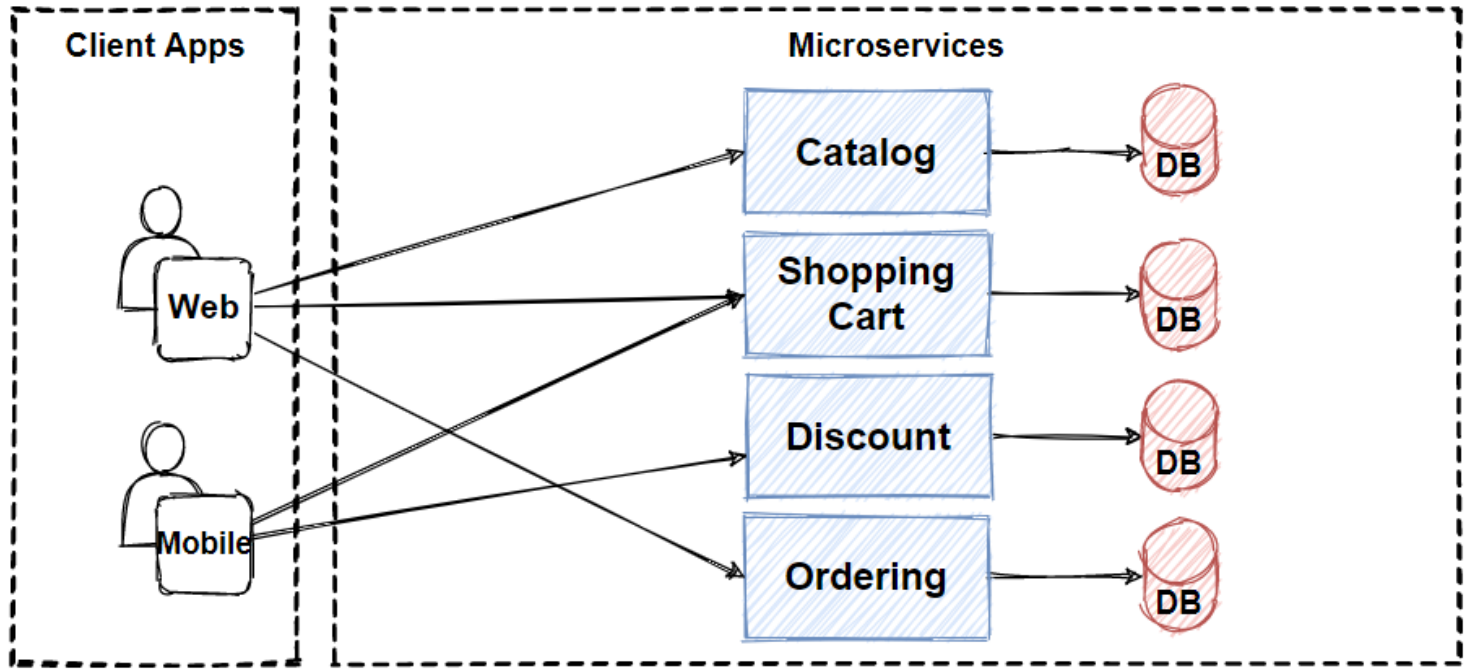
https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/introduce-eshoponcontainers-reference-app

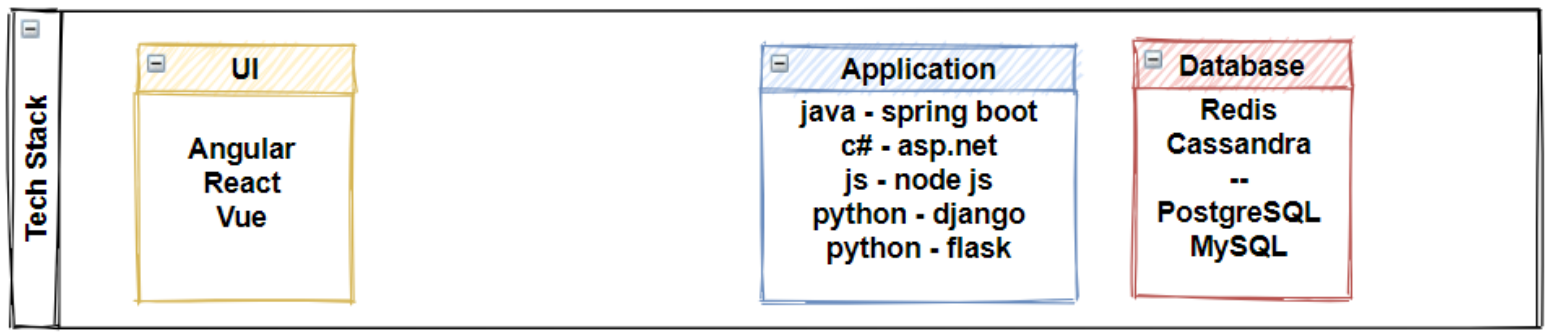
Microservices Architecture

Principles

- KISS
- YAGNI
- SoC
- SOLID

- Api Gw
- Database per Microservices





Functional Requirements

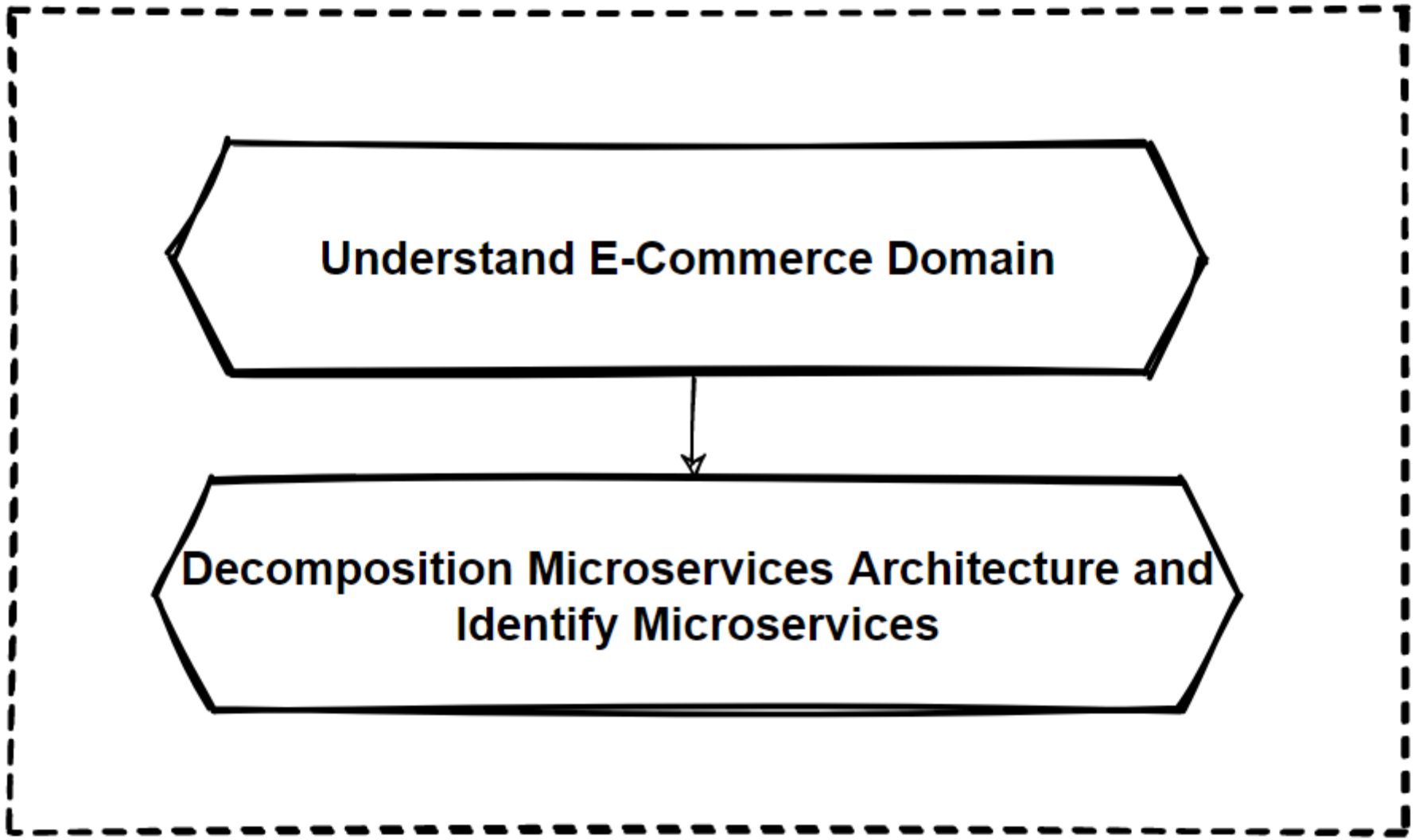
- List products
- Filter products as per brand and categories
- Put products into the shopping cart
- Apply coupon for discounts and see the total cost all for all of the items in shopping cart
- Checkout the shopping cart and create an order
- List my old orders and order items history

Non-Functional Requirements

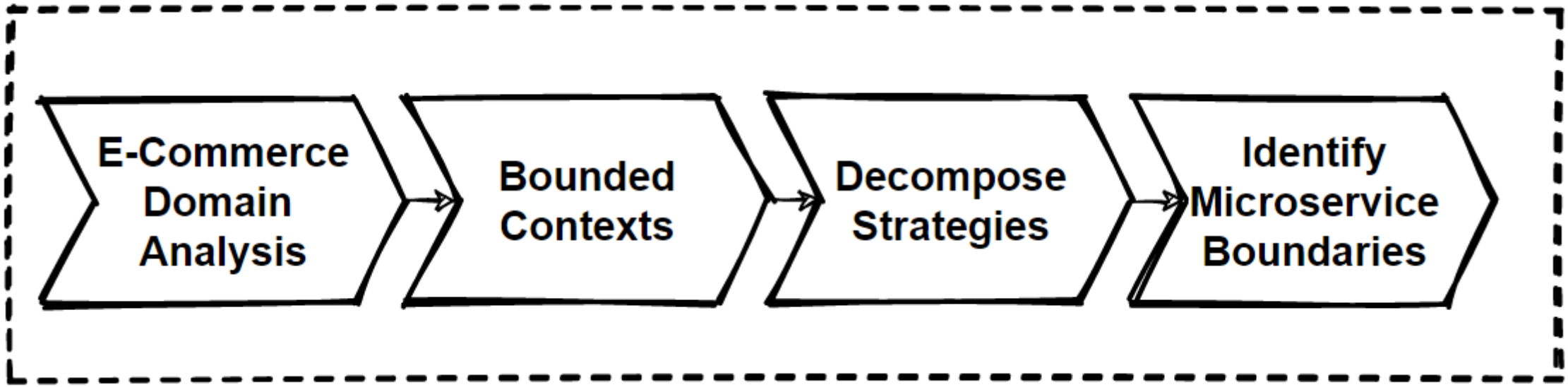
- Scalibility
- Increase Concurrent User
- Maintainability

Section 7
Decomposition Microservices Architecture

Understanding E-Commerce Domain

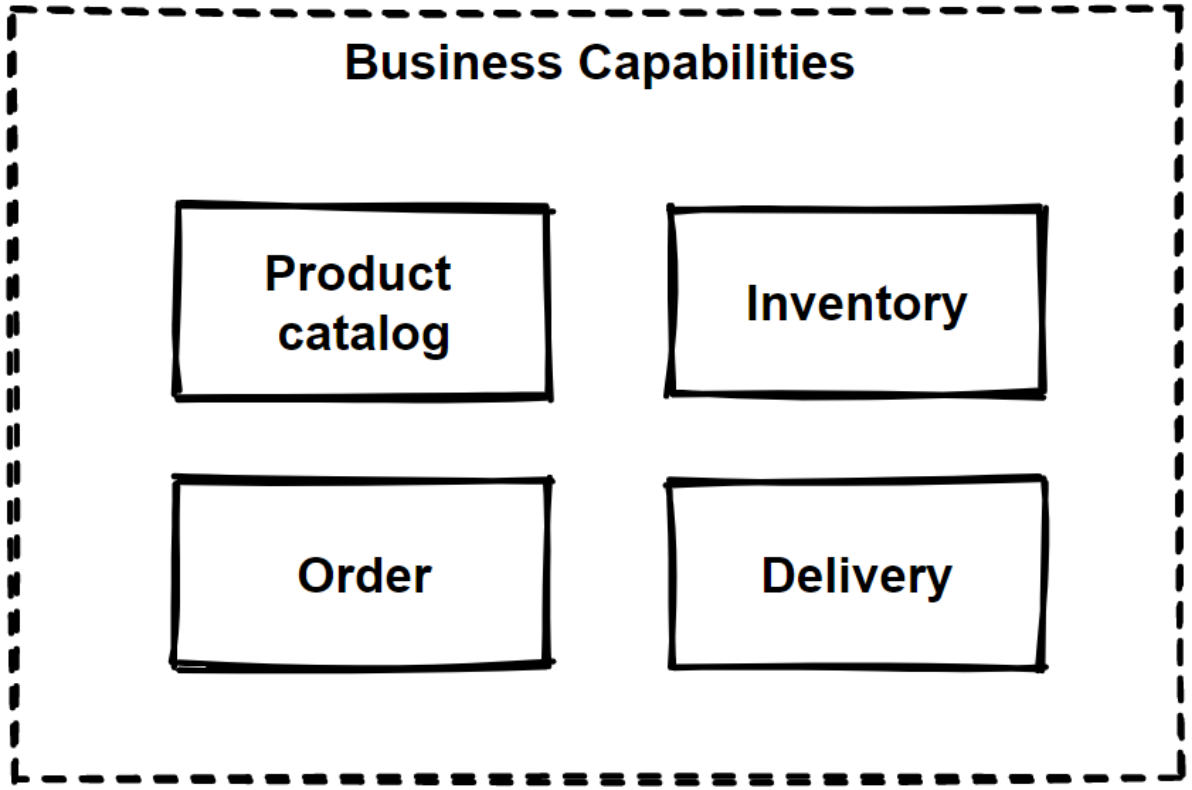

Understand E-Commerce Domain
Decomposition Microservices Architecture and Identify Microservices

Decomposition Microservices Architecture Path



Microservices Decomposition Pattern - Decompose by Business Capability

- Services must be Cohesive
- Services must be Loosely Coupled
- Corresponding to Business Capabilities
- Business Capability

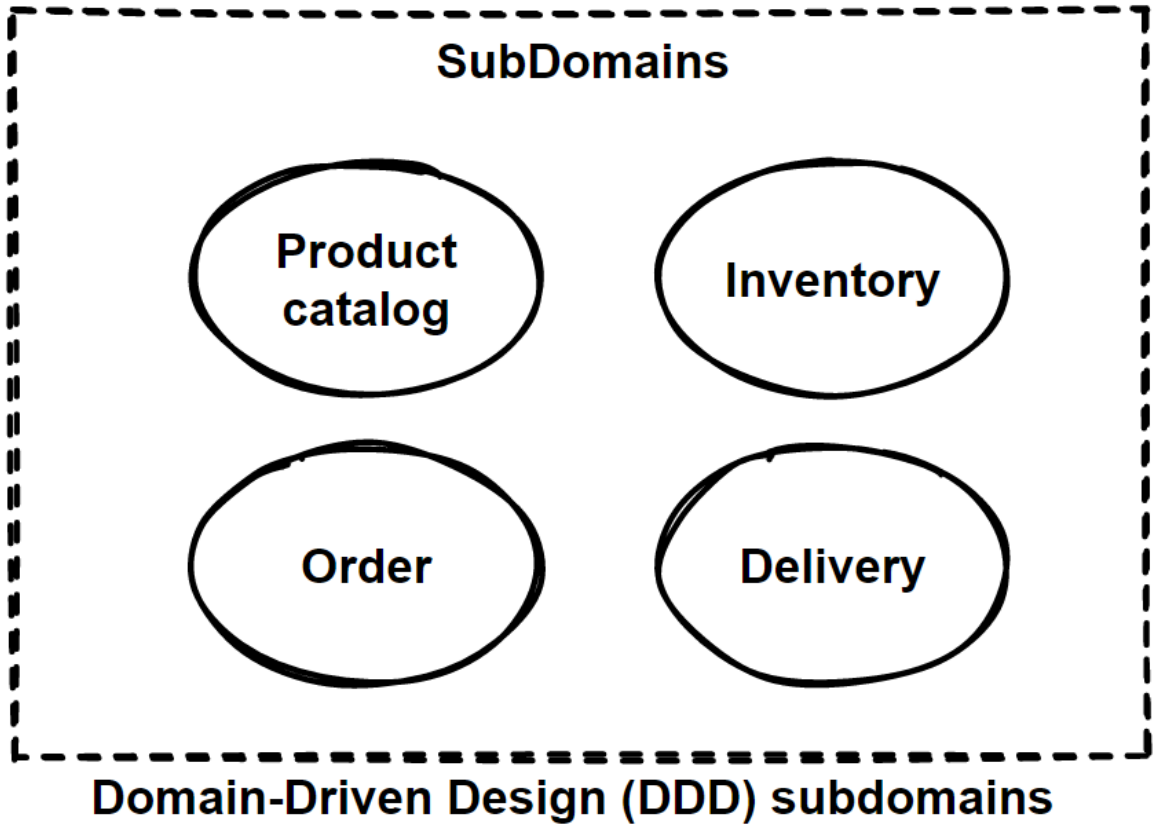


Microservices Decomposition Pattern - Decompose by Subdomain

- Services must be Cohesive
- Services must be Loosely Coupled
- Domain-Driven Design (DDD)

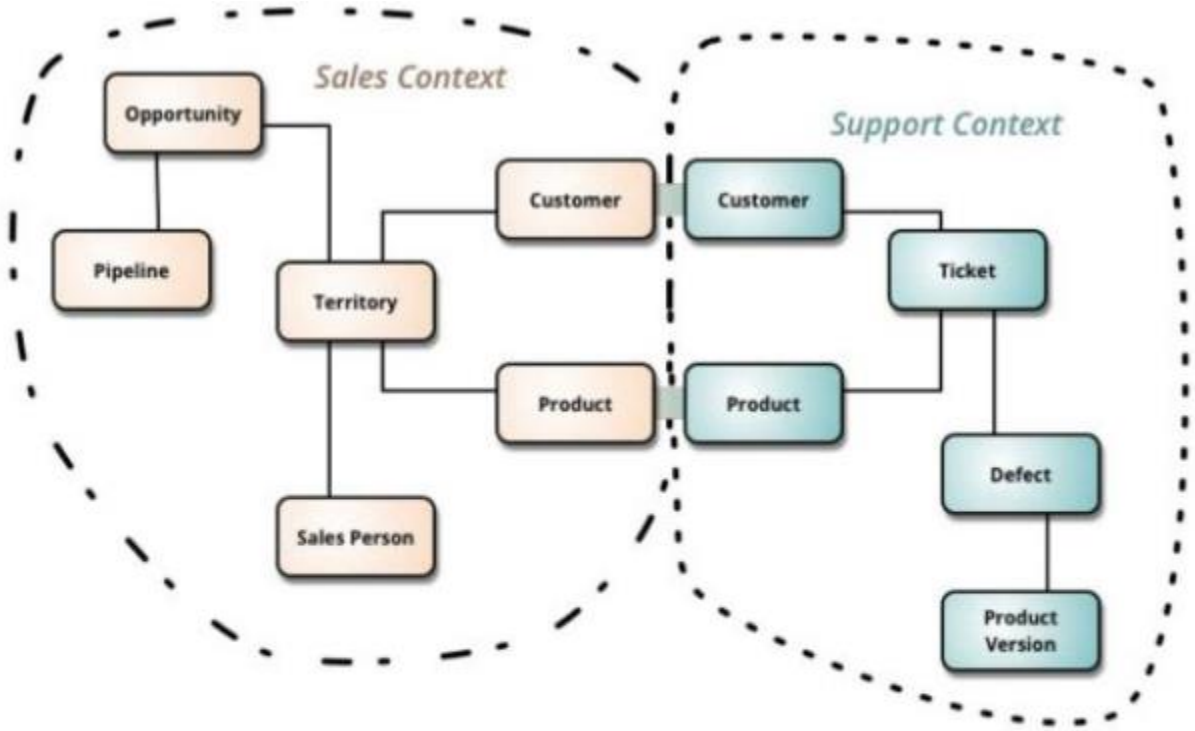
Subdomains

- The business - as the domain



Bounded Context Pattern (Domain-Driven Design - DDD)

- DDD-Bounded Context
- Domains are required high cooperation
- Strategic and Tactical DDD
- Grouping of closely related Scopes
- Logical boundaries have common business rules

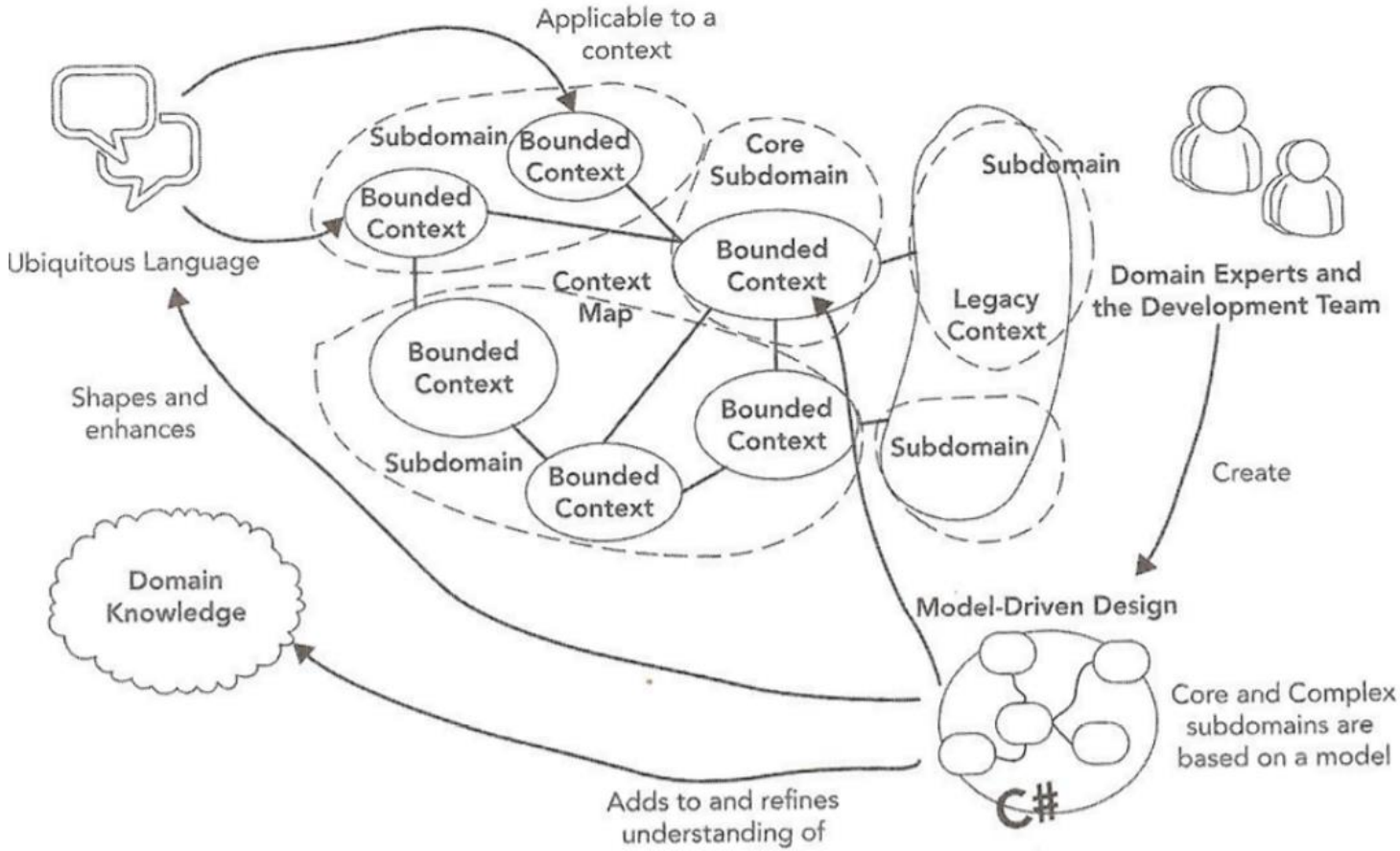


DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.

http://martinfowler.com/bliki/BoundedContext.html

Identify Bounded Context Boundaries for Each Microservices

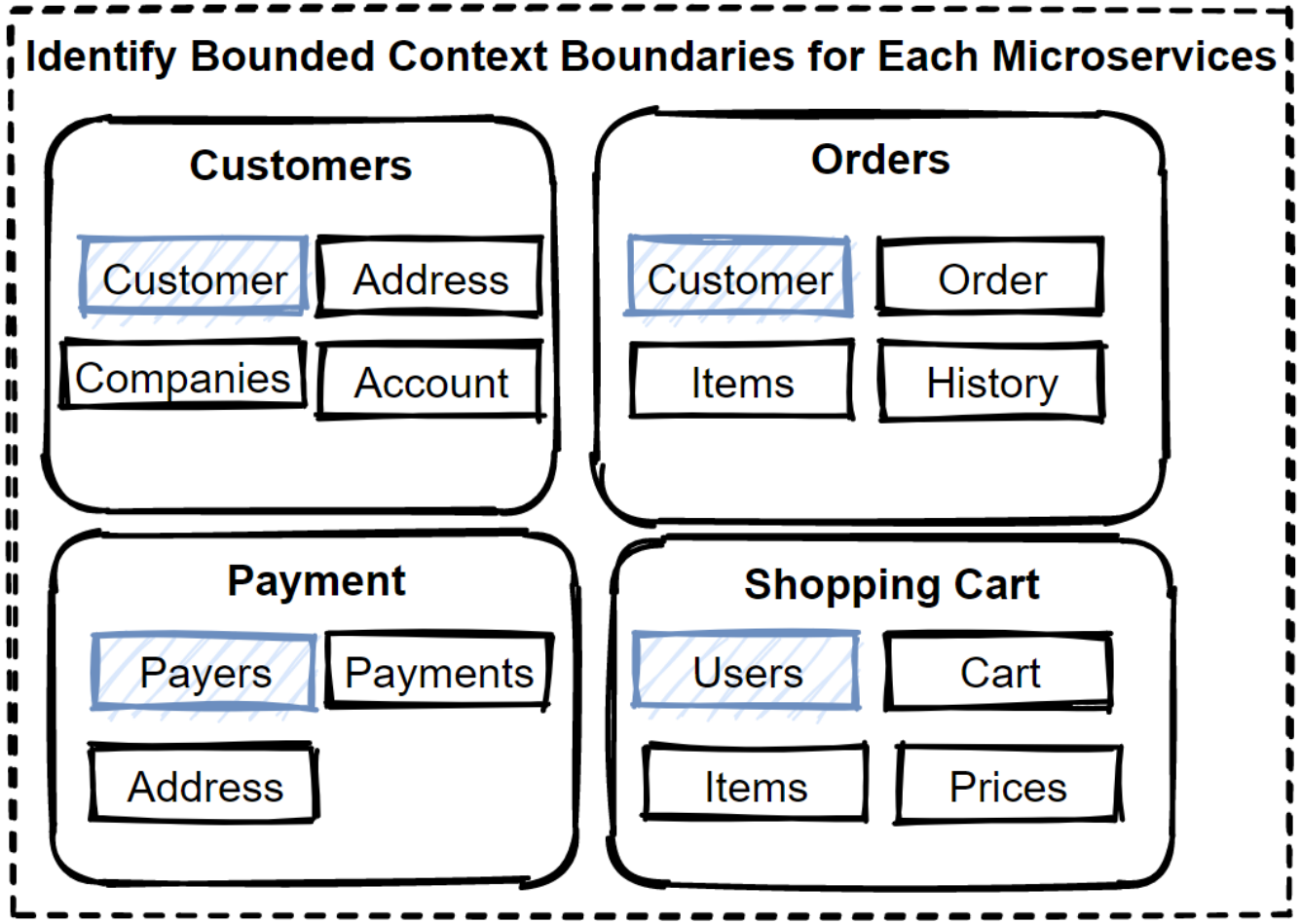
- DDD-Bounded Context
- Context Mapping pattern
- Context Mapping
- Define logical boundaries between domains



Identify Bounded Context Boundaries for Each Microservices 2

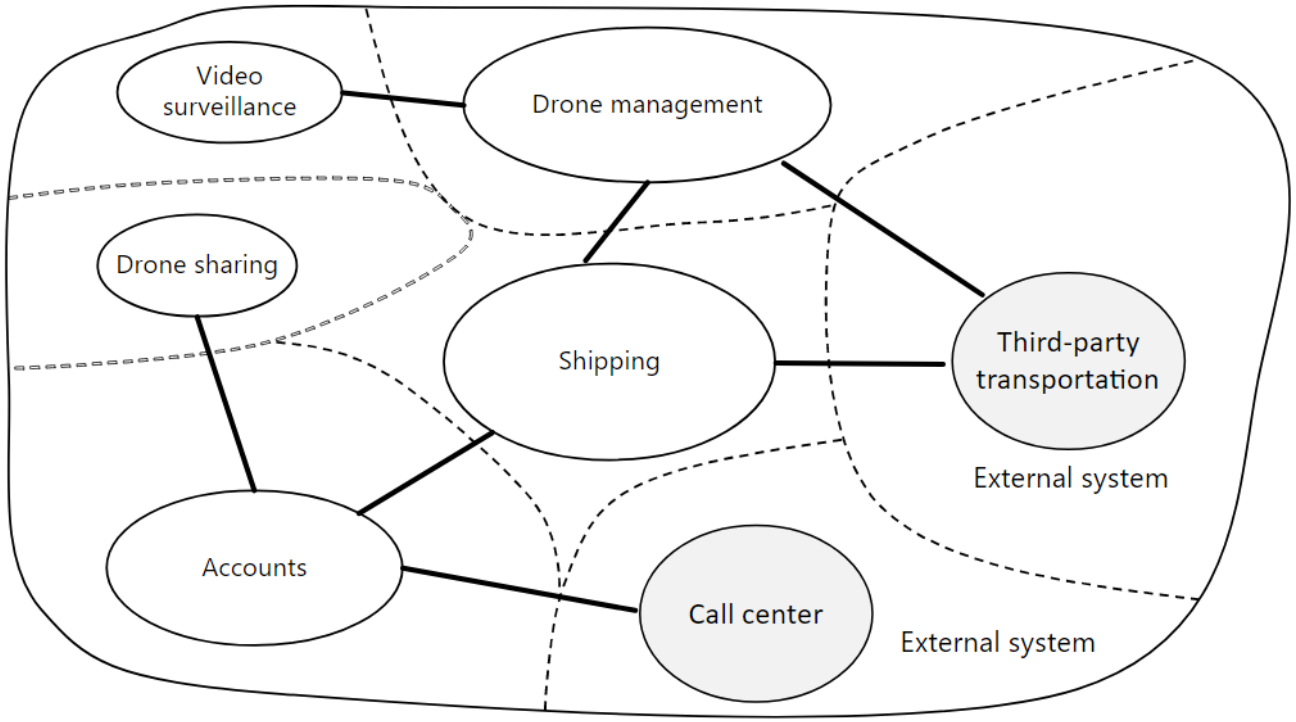
- Talking to the domain experts
- Changes to the boundaries
- Reshape your Bounded Contexts
- Consider Refactorings

- **A Bounded Context == A Microservice ?**

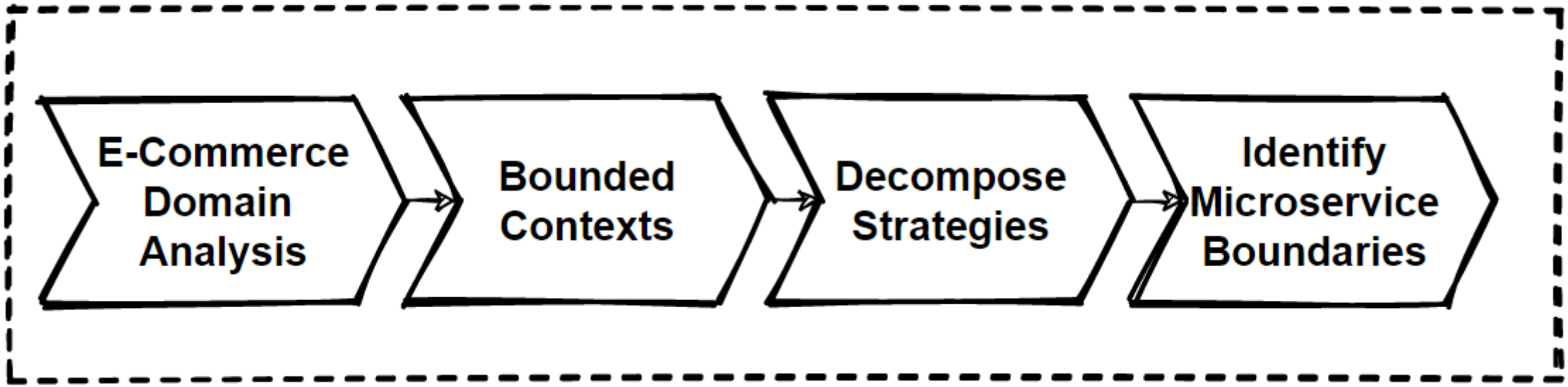


Using Domain Analysis to Model Microservices

- Domain Analysis
- Designed by business capabilities
- Loose coupling and autonomous services
- **DDD-Bounded Context** following **Context Mapping Pattern** and **decompose by sub domain** models patterns

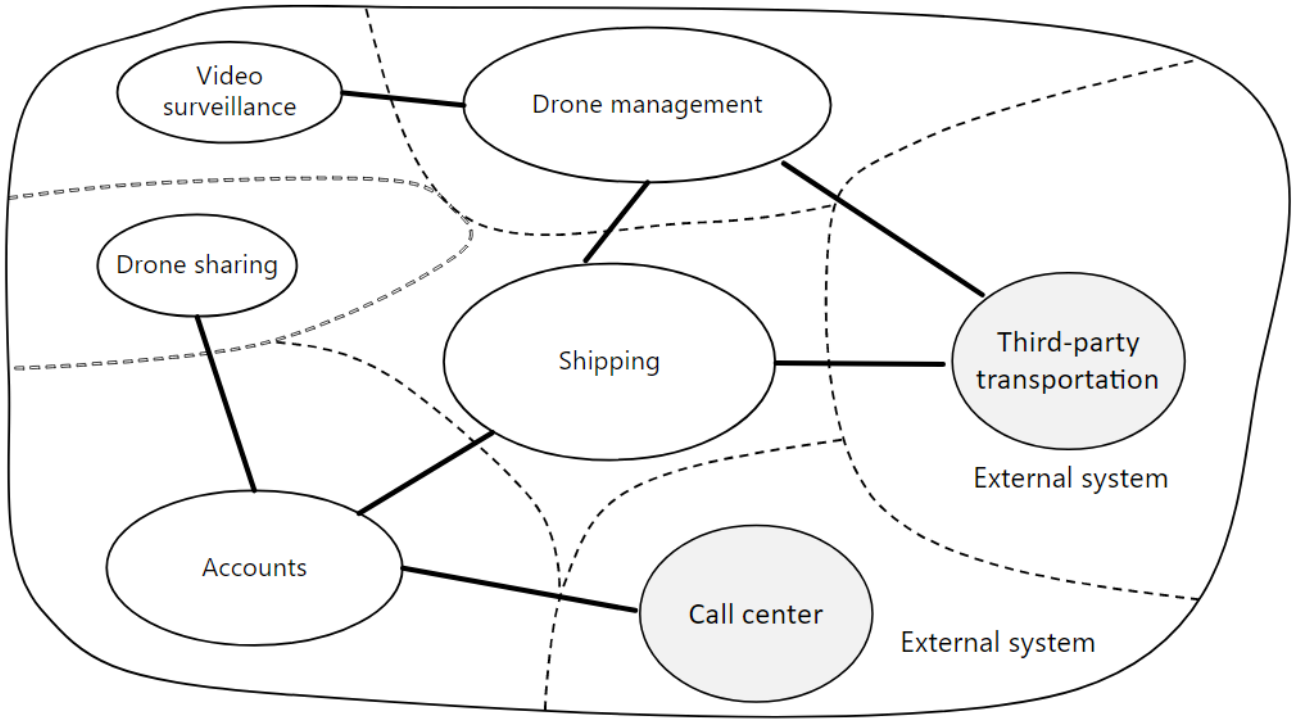


Decomposition Microservices Architecture Path



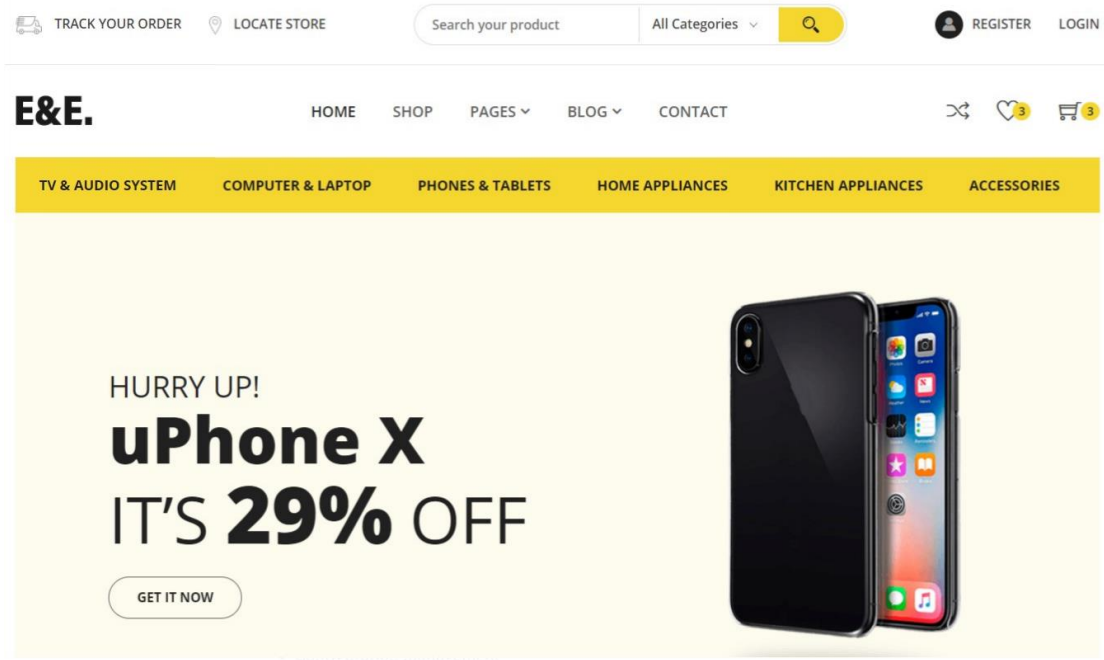
Using Domain Analysis to Model Microservices

- Domain Analysis
- Designed by business capabilities
- Loose coupling and autonomous services
- **DDD-Bounded Context** following **Context Mapping Pattern** and **decompose by sub domain** models patterns



Understand E-Commerce Domain

- Use Cases
- Functional Requirements



Follow steps;

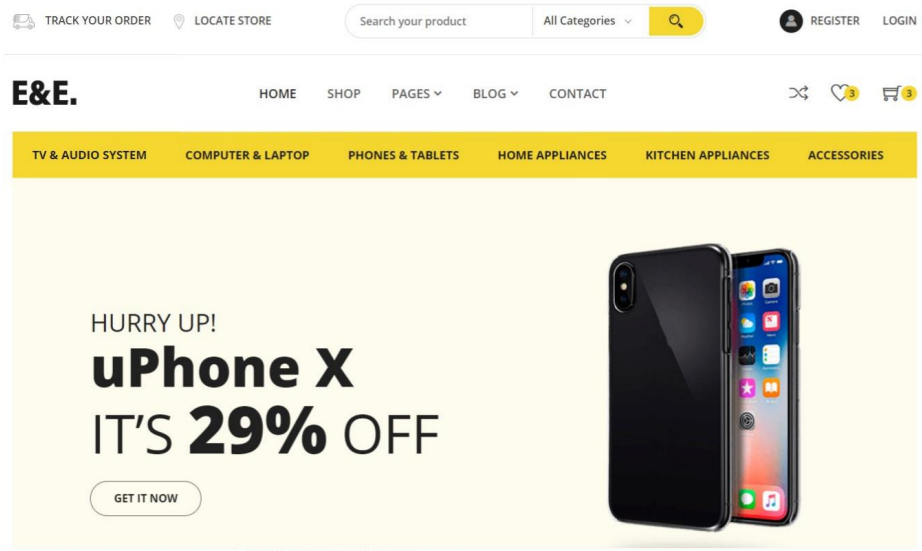
- Requirements and Modelling
- Identify User Stories
- Identify the Nouns in the user stories
- Identify the Verbs in the user stories

E-Commerce Functional Requirements

- List products
- Filter products as per brand and categories
- Put products into the shopping cart
- Apply coupon for discounts and see

the total cost all for all of the items

in shopping cart



- Checkout the shopping cart and create an order
- List my old orders and order items history

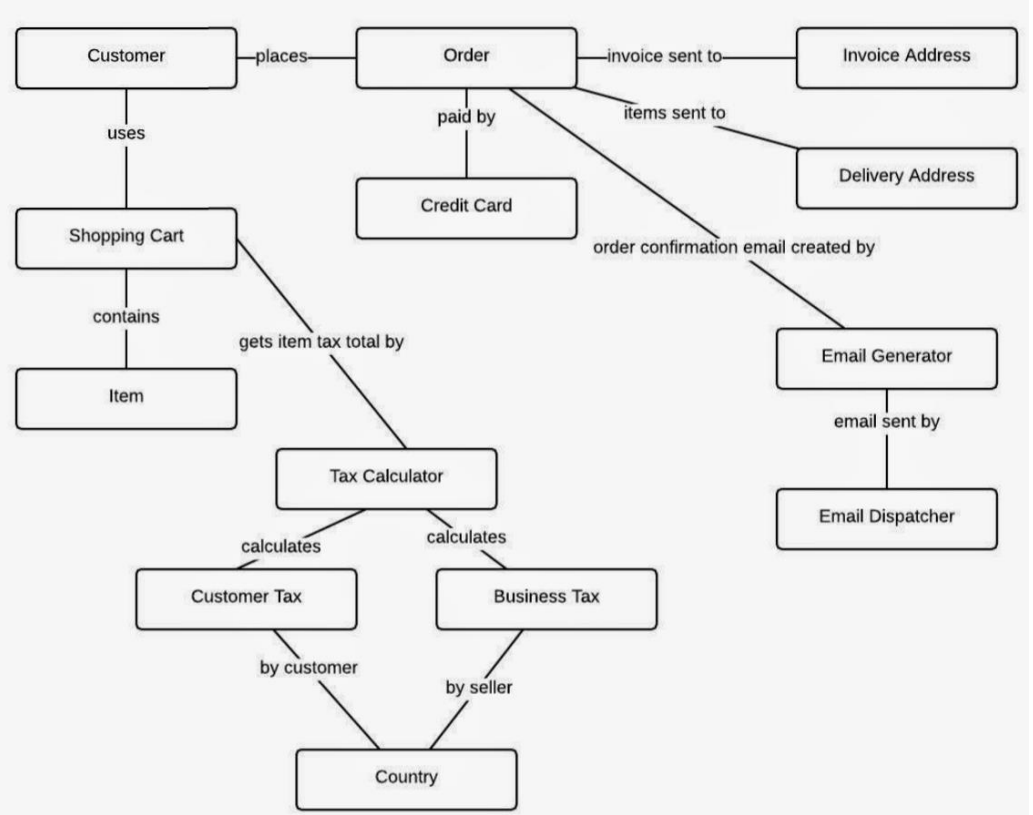
Analysis E-Commerce Domain - Nouns and Verbs

- List products
- Filter products as per brand and categories
- Put products into the shopping cart
- Apply coupon for discounts and see

the total cost all for all of the items

in shopping cart

- Checkout the shopping cart and create an
- List my old orders and order items history



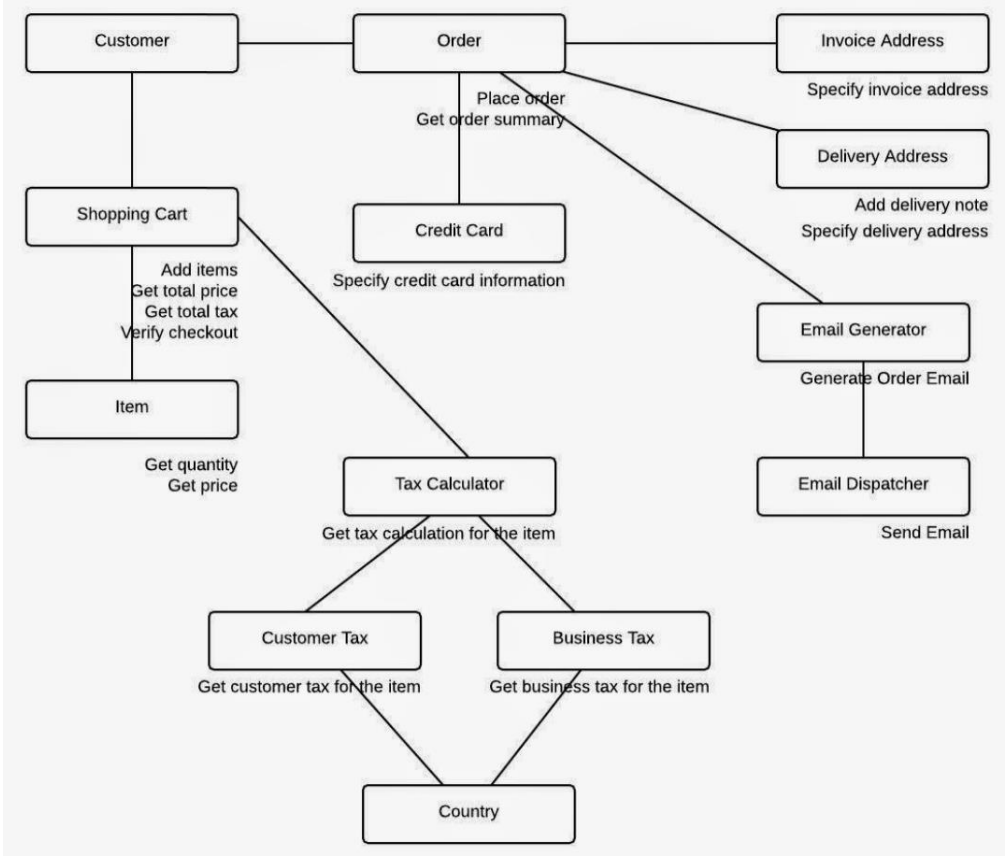
https://medium.com/aspnetrun/build-layered-architecture-with-asp-net-core-e framework-core-in-a-real-word-example-aa54a7ed7bef

Analysis E-Commerce Domain - Nouns and Verbs 2

- As a user I want to **list** **products**
- As a user I want to be able to **filter** **products** as per **brand** and **categories**
- As a user I want to see the supplier of **product** in the product detail screen with all characteristics of product
- As a user I want to be able to **put** **products** that I want to **purchase** in to the shopping cart so that I can check out quickly later on
- As a user I want to see the total cost all for all of the items that are in my **cart** so that I see if I can afford to buy everything
- As a user I want to **see** the total cost of each **item** in the **shopping cart** so that I can re-check the price for items
- As a user I want to be able to specify the address of where all of the products are going to be sent to
- As a user I want to be able to add a note to the delivery **address** so that I can provide special instructions to the postman
- As a user I want to be able to specify my credit card information during **check out** so that I can pay for the items
- As a user I want system to tell me how many items are in **stock** so that I know how many items I can purchase
- As a user I want to **receive** **order** confirmation email with order number so that I have proof of purchase
- As a user I want to **list** my old **orders** and **order items history**
- As a user I want to **login** the system as a **user** and the system should remember my shopping cart items

Analysis E-Commerce Domain - Nouns and Verbs 3

- Customer
- Order
- Order Details
- Product
- Shopping Cart
- Shopping Cart Items
- Supplier
- User
- Address
- Brand
- Category

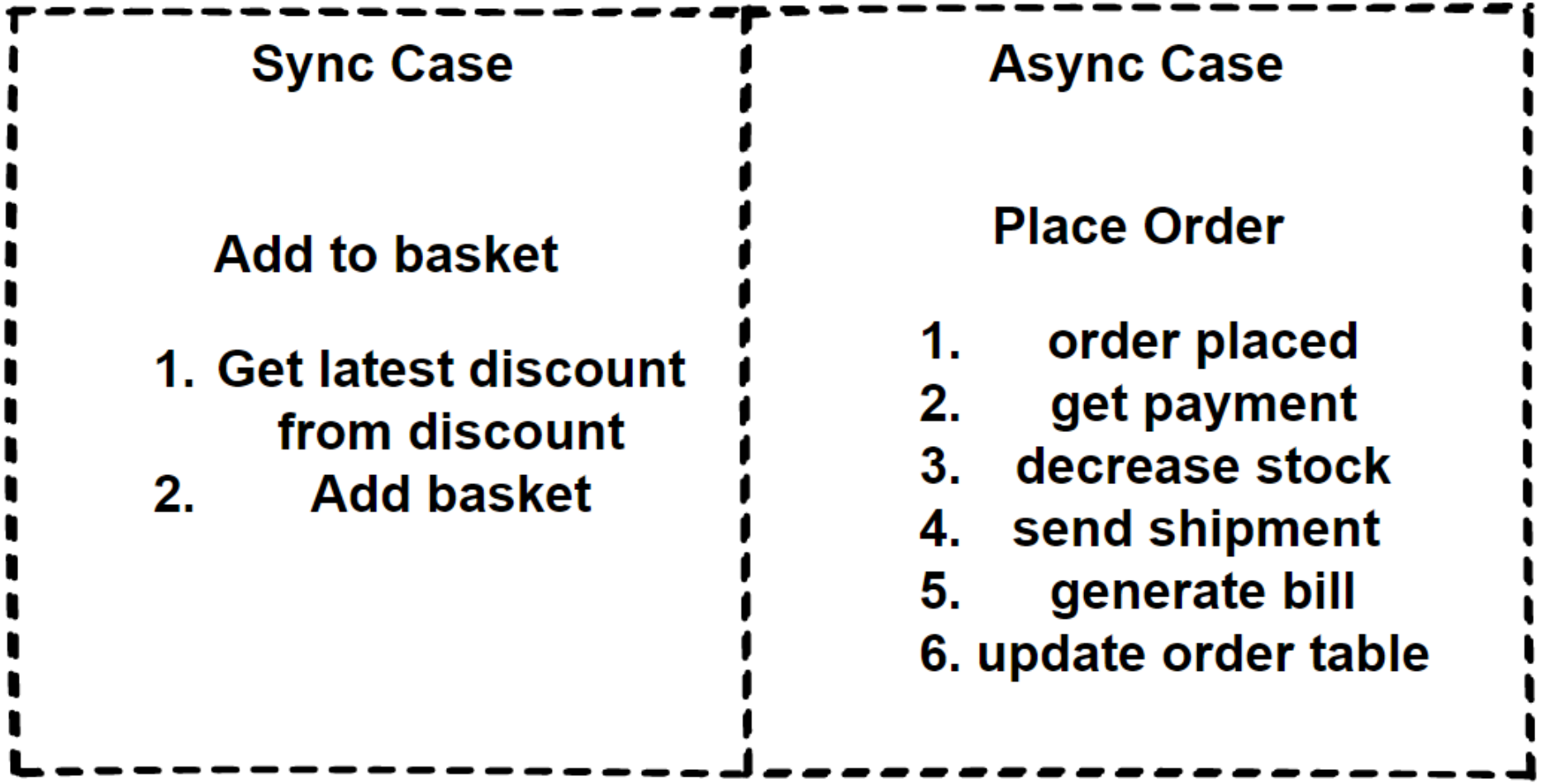


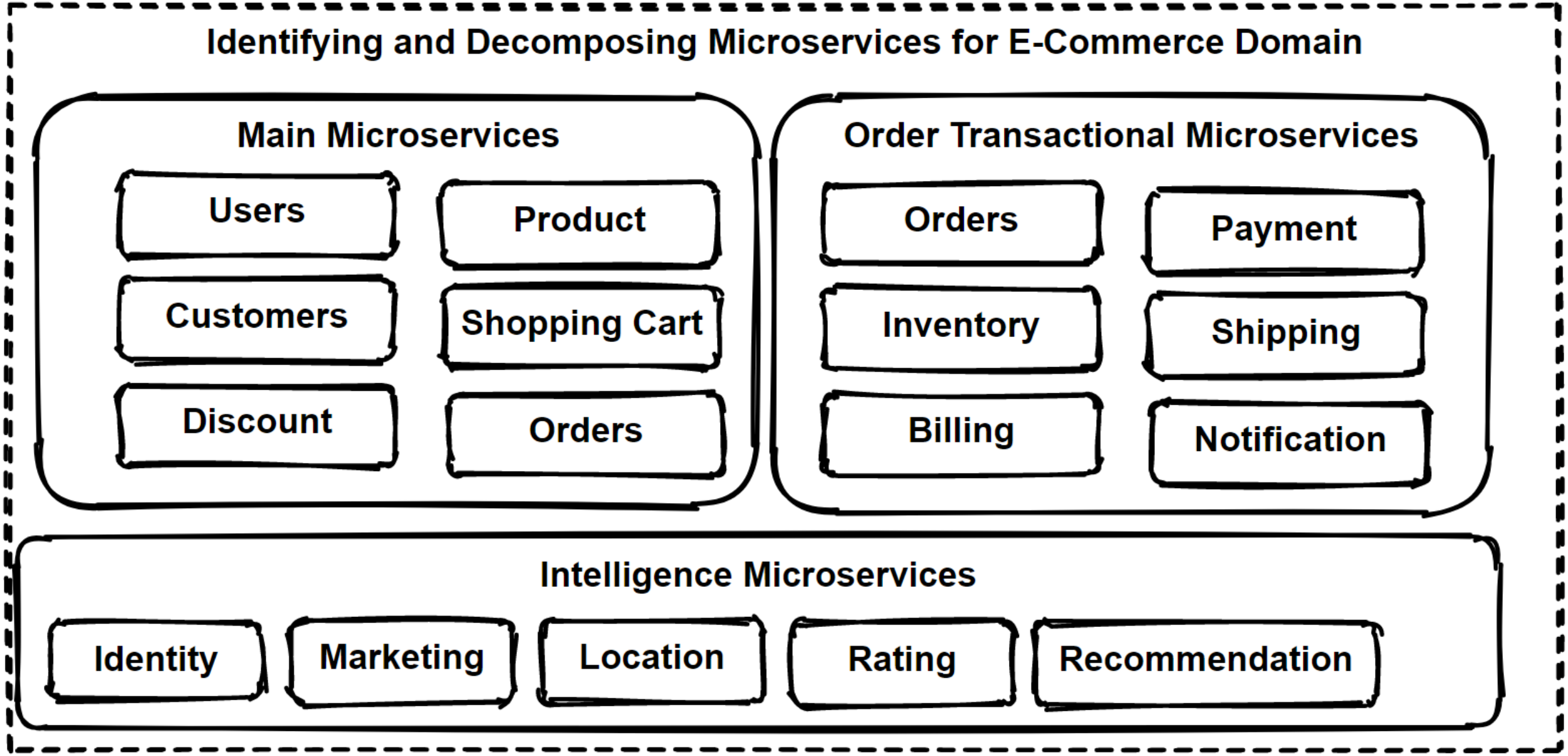
https://medium.com/aspnetrun/build-layered-architecture-with-asp-net-core-e framework-core-in-a-real-word-example-aa54a7ed7bef

Analysis E-Commerce Domain - Nouns and Verbs 4

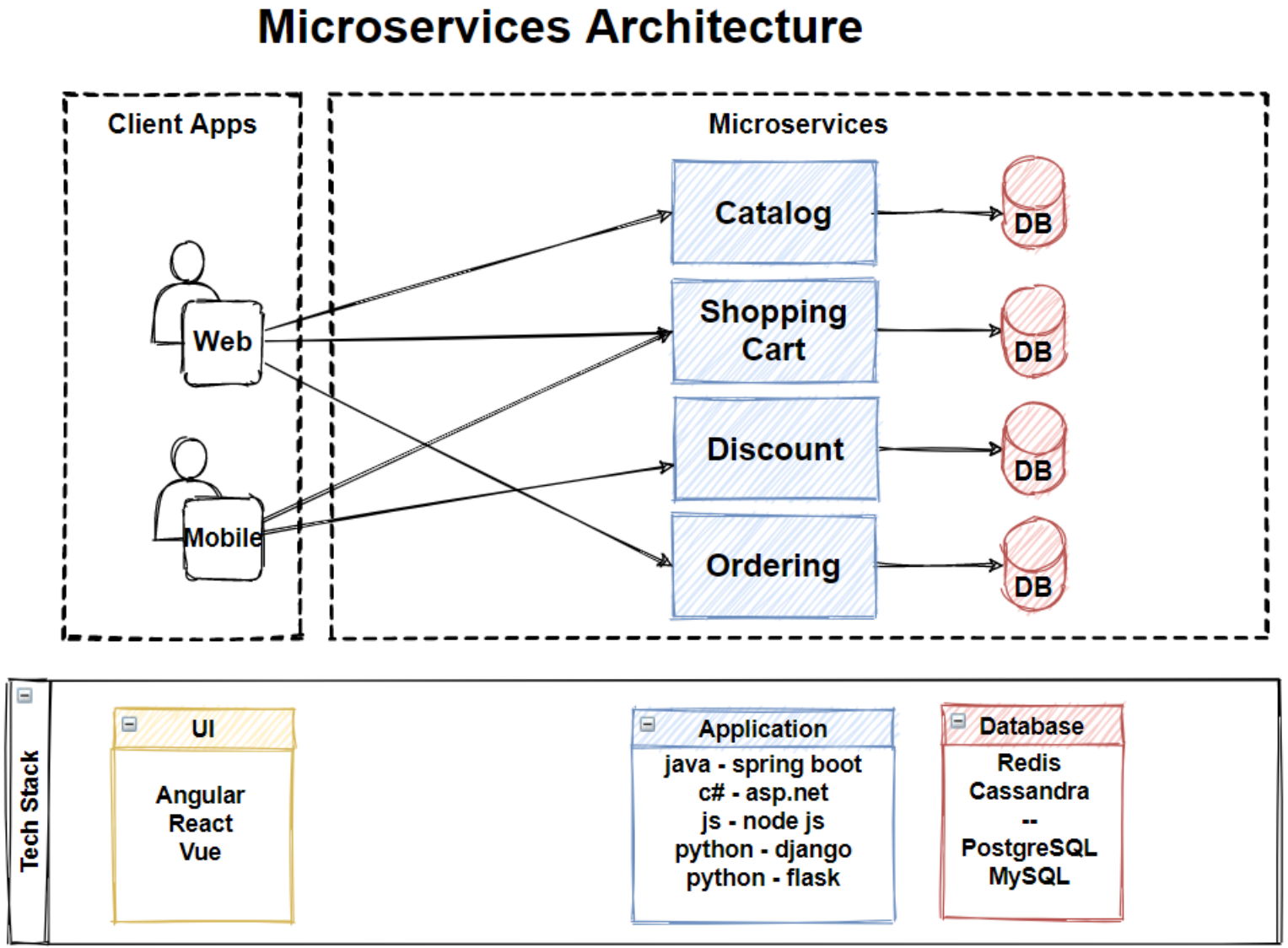
- List products applying to paging
- Filter products by brand, category and supplier
- See product all information in the details screen
- Put products into the shopping cart
- See total cost for all of the items
- See total cost for each item
- Checkout order with purchase steps
- Specify delivery address
- Specify delivery note for delivery address
- Specify credit card information
- Pay for the items
- Tell me how many items are in stock
- Receive order confirmation email
- List the order and details history
- Login the system and remember the shopping cart items

Identifying and Decomposing Microservices for E-Commerce Domain




Identifying and Decomposing Microservices for E-Commerce Domain
Main Microservices
Users
Product
Customers
Shopping Cart
Discount
Orders
Order Transactional Microservices
Orders
Payment
Inventory
Shipping
Billing
Notification
Intelligence Microservices
Identity
Marketing
Location
Rating
Recommendation

Where were we ?

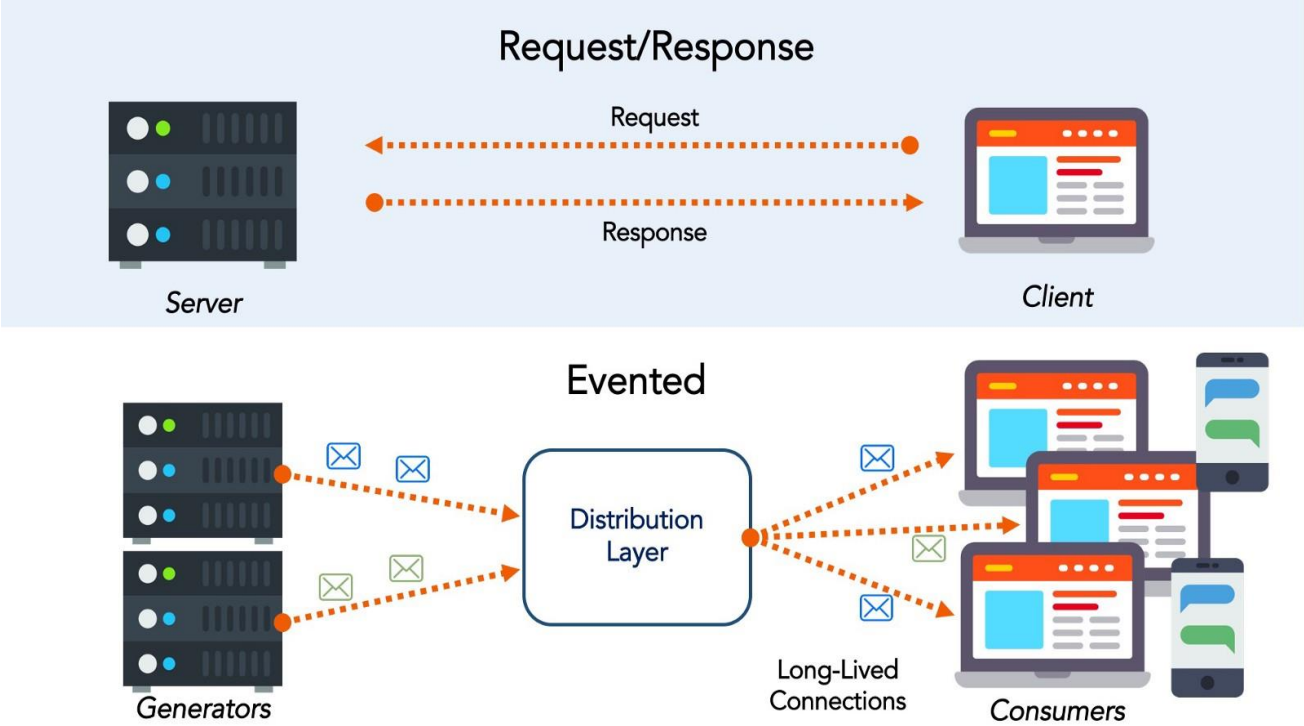


Section 8
Microservices Communications

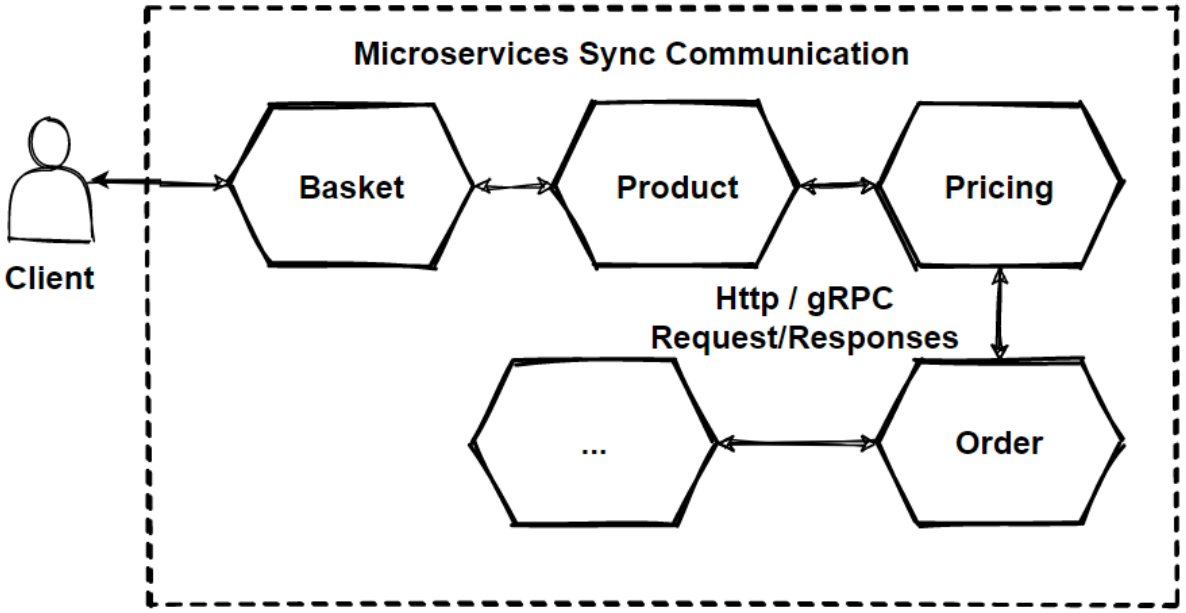
Microservices Communication Types - Sync or Async Communication

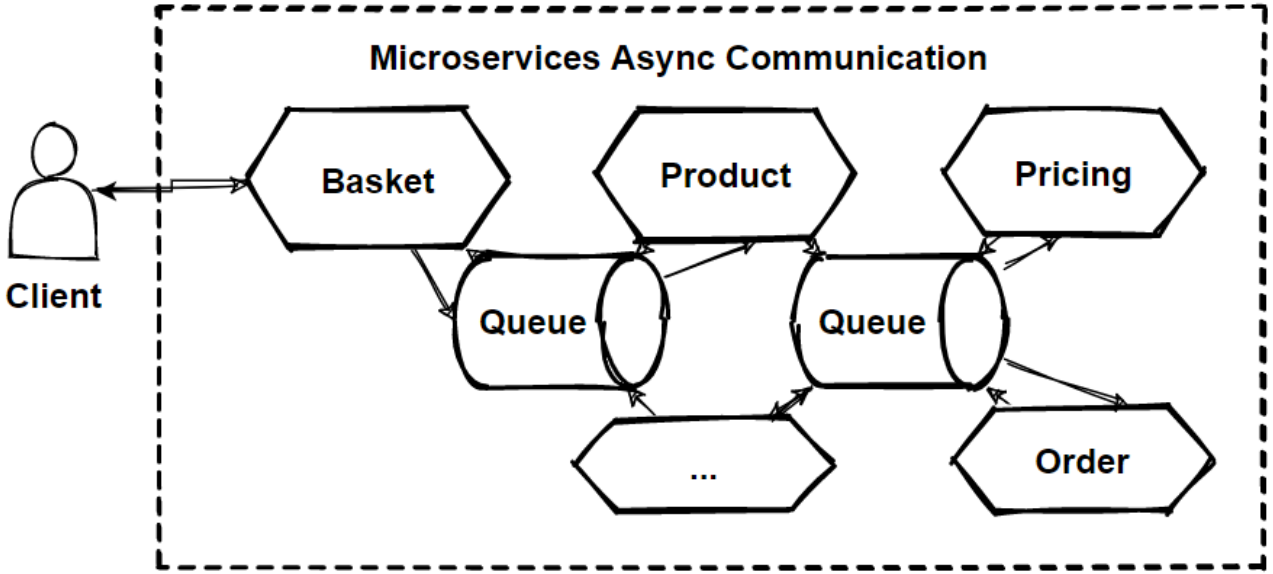
Microservices Communications

- Monolithic = Inter-Process
- Microservices = Inter-Service communication
- Distributed network calls
- HTTP, gRPC or message brokers AMQP protocol
- Synchronous or Asynchronous



Microservices Communications



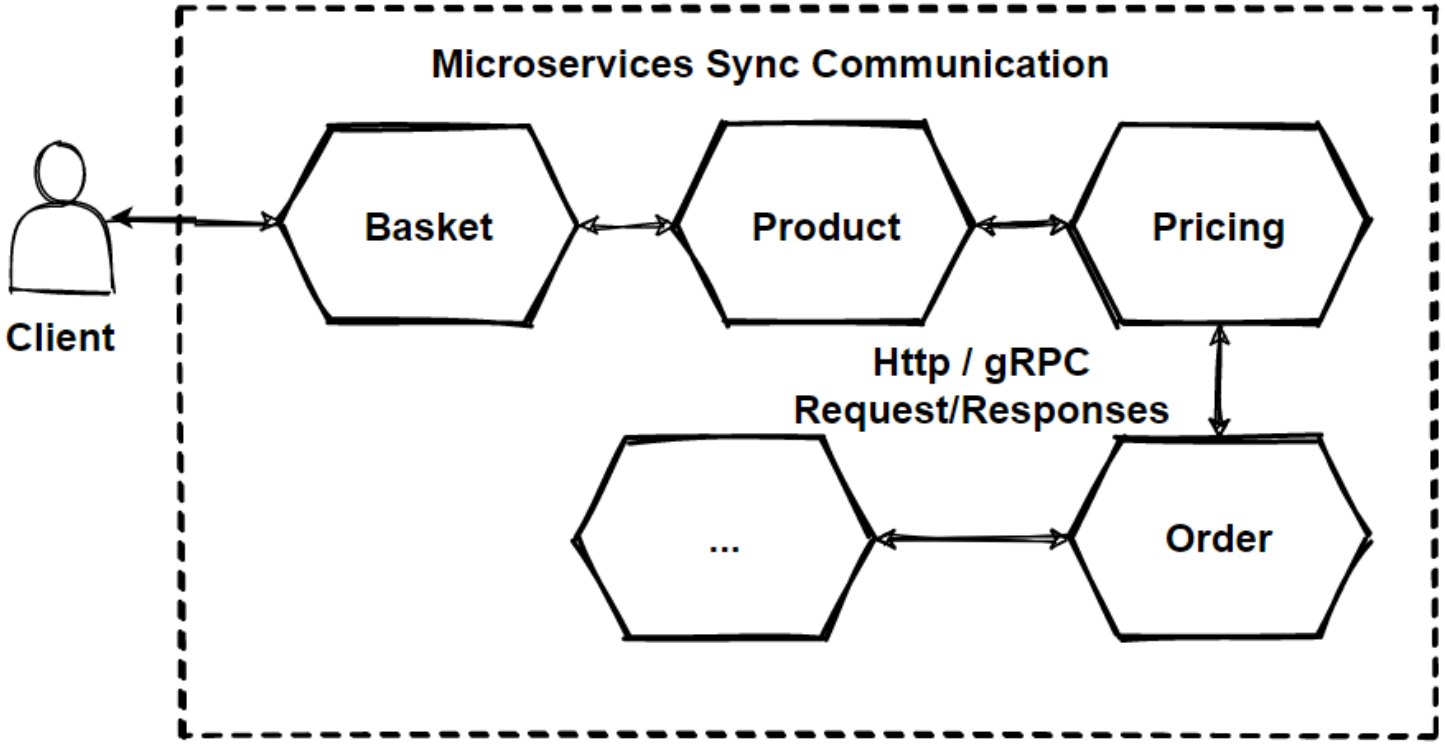


Microservices Synchronous Communication

- Inter-Service communication
- Distributed network calls
- HTTP, gRPC protocols
- Synchronous

Request/Response

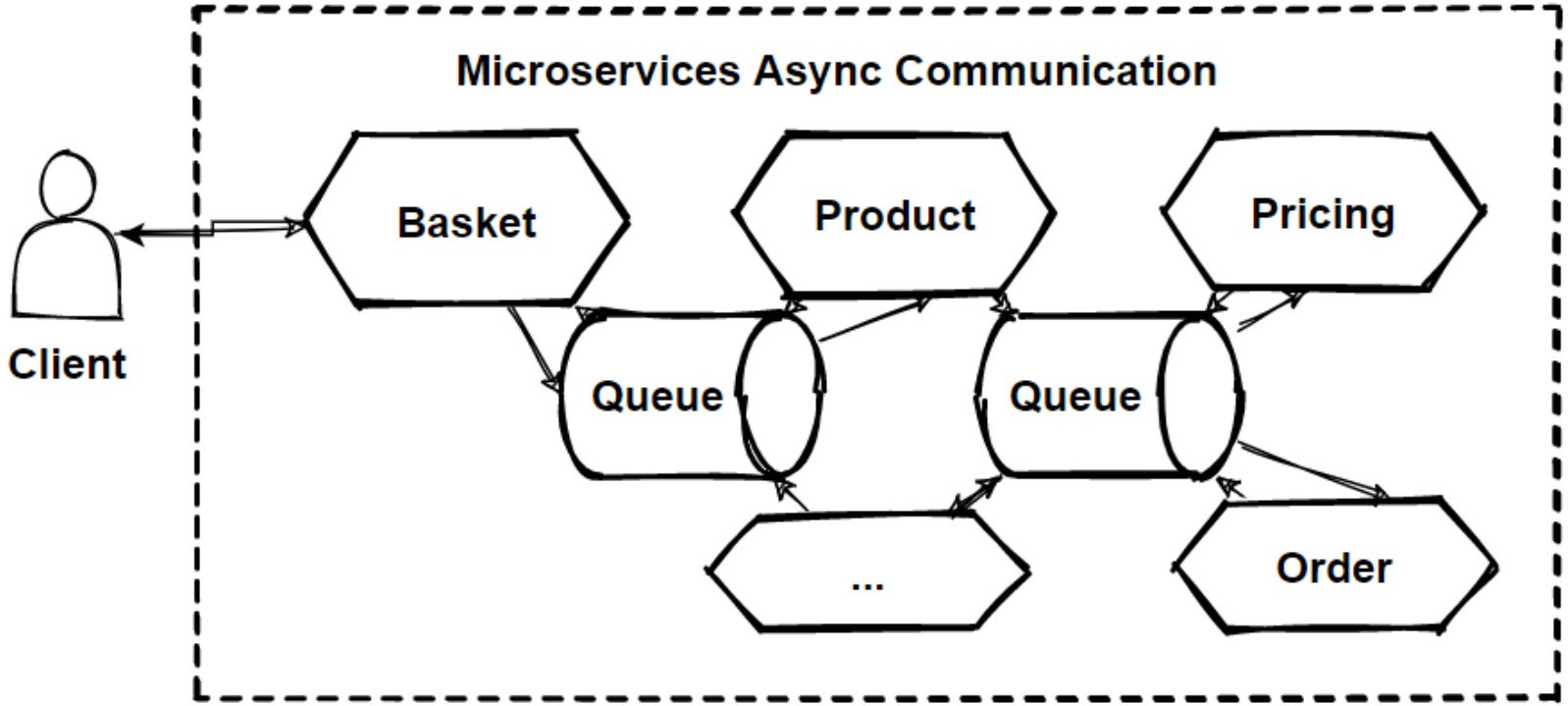
- REST APIs



Microservices Asynchronous Communication

- Not wait Response and not have blocked a thread
- AMQP (Advanced Message QueuingProtocol)
- one-to-one(queue)
- one-to-many (topic)
- publish/subscribe
- Event-driven

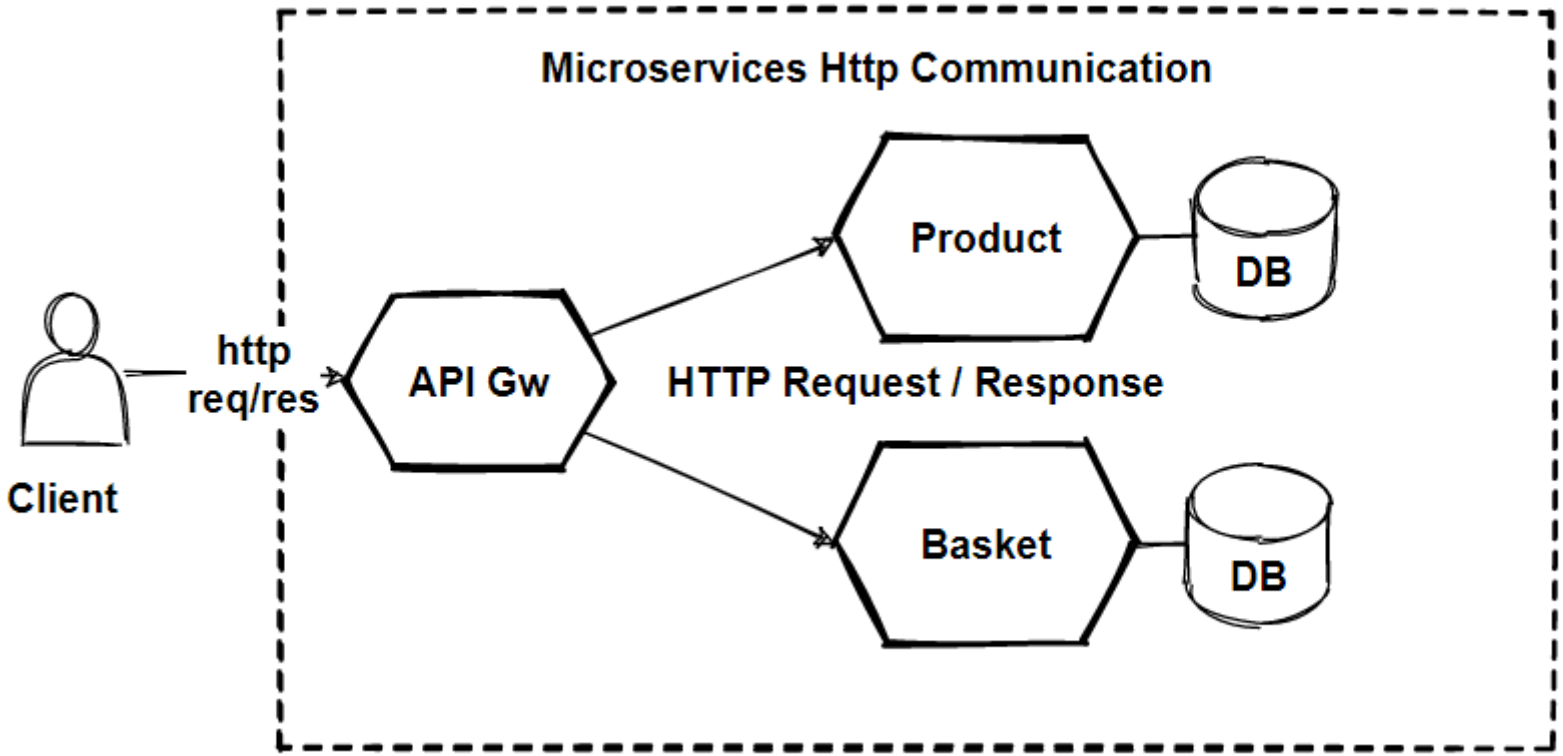
microservices architecture



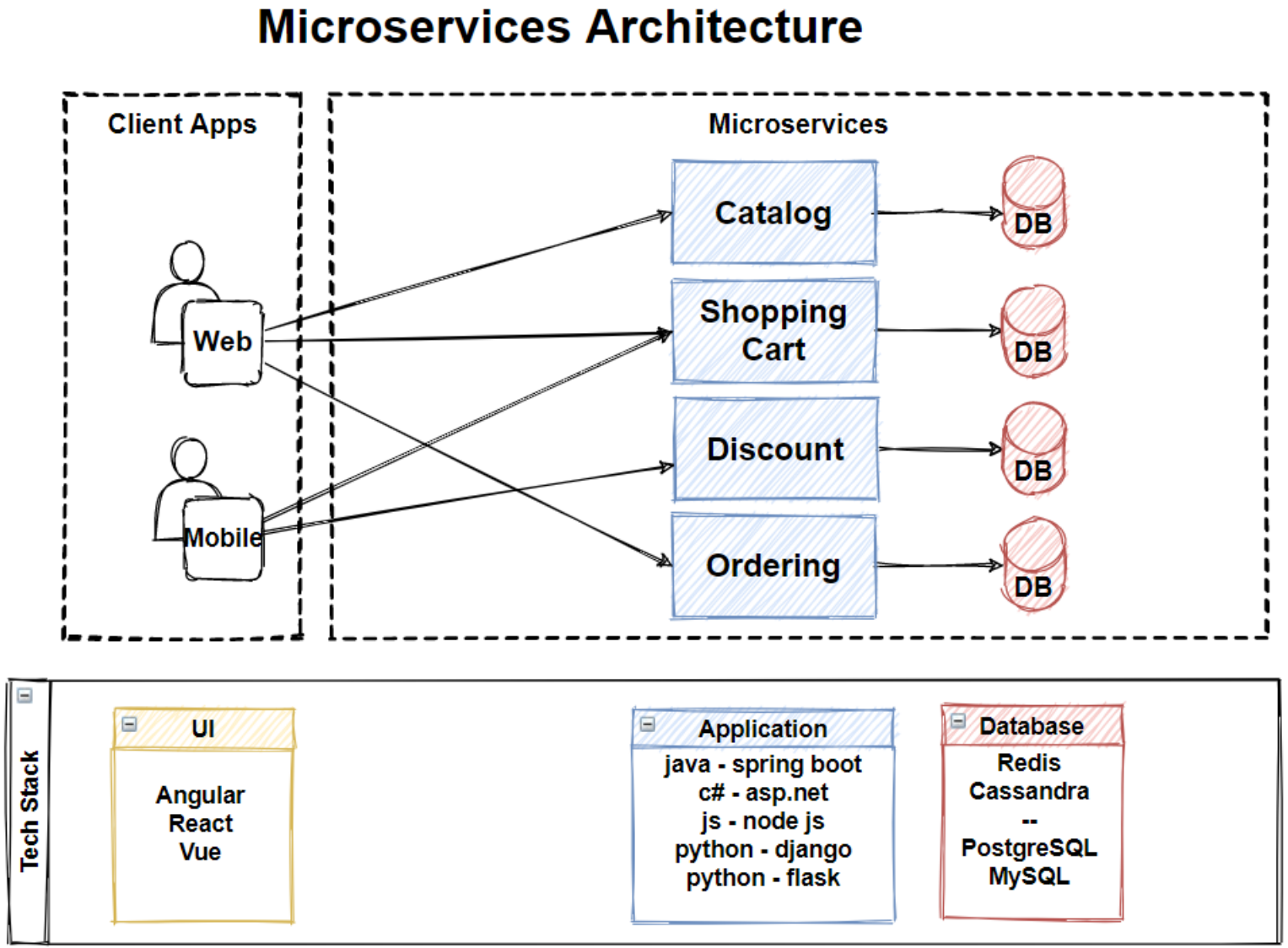
Microservices Synchronous Communication and Practices

- Not wait Response and not have blocked a thread
- AMQP (Advanced Message QueuingProtocol)
- one-to-one(queue)
- one-to-many (topic)
- publish/subscribe
- Event-driven

microservices architecture

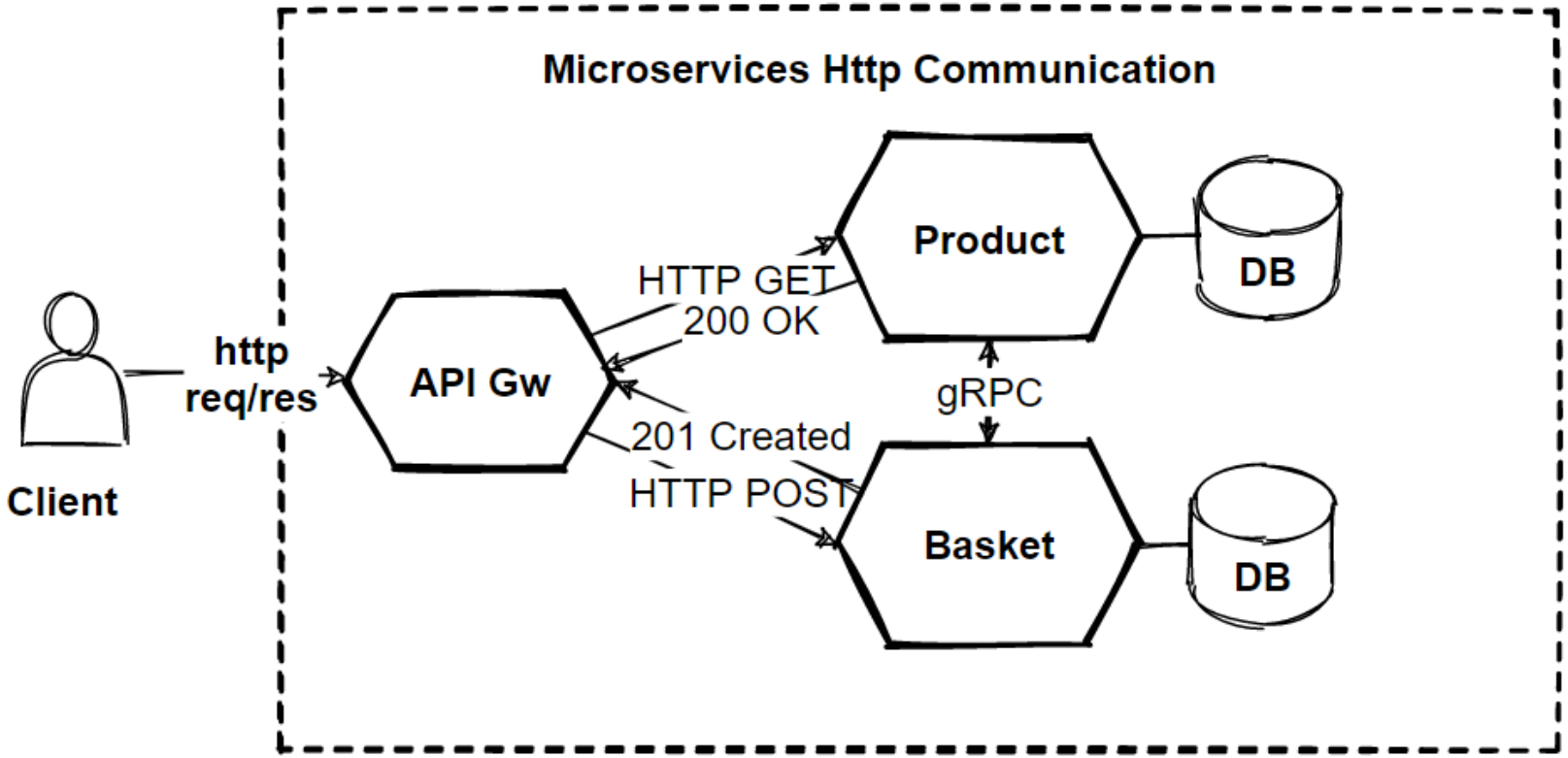


E-Commerce Service Communications



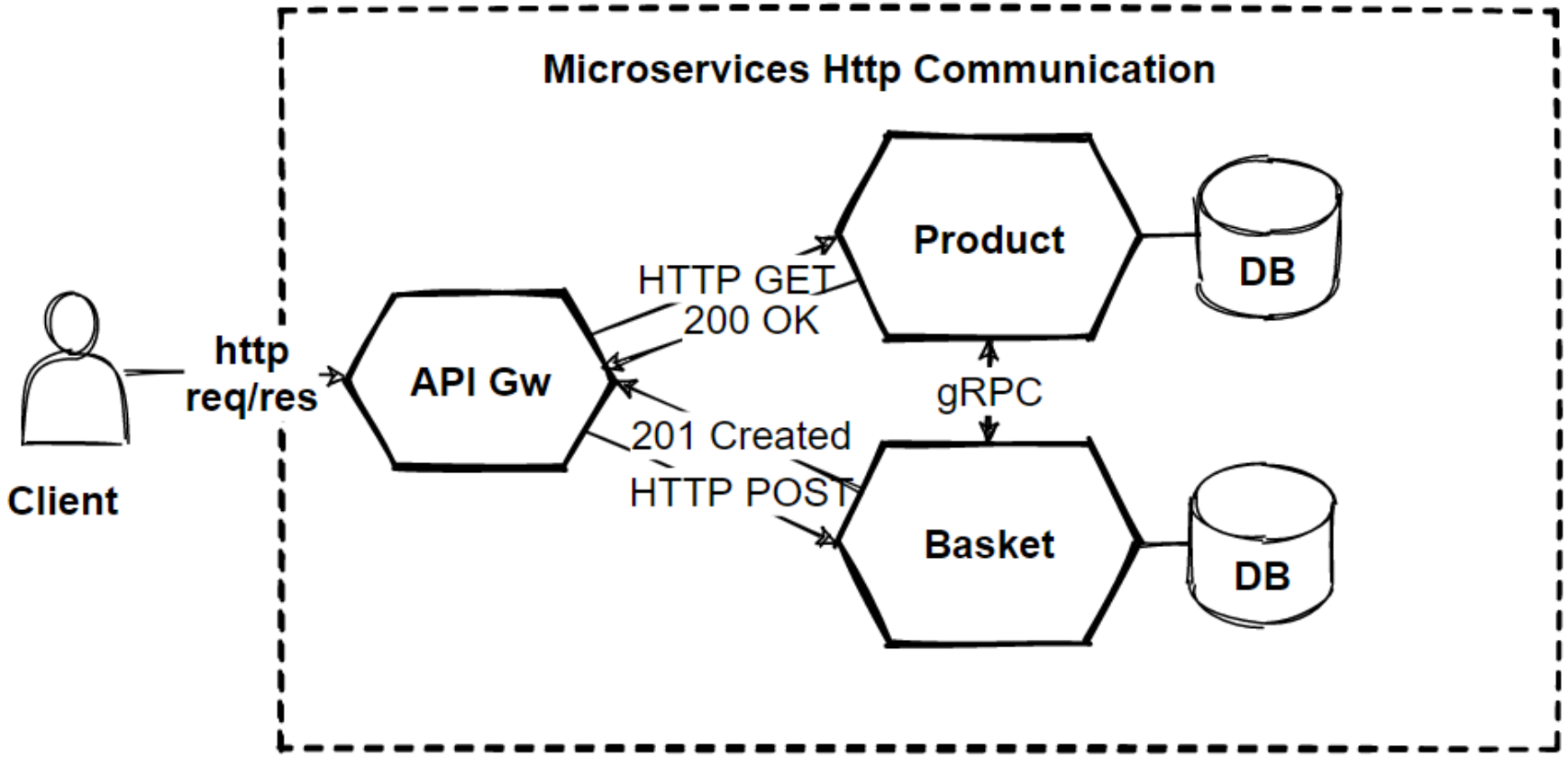
Designing HTTP based RESTful APIs for Microservices

- REST APIs for HTTP verbs like GET, POST, and PUT
- Creating APIs for our microservices
- Design Microservice APIs



Designing HTTP based RESTful APIs for Microservices – HTTP GET POST PUT

- REST APIs for HTTP verbs like GET, POST, and PUT
- Avoid chatty calls
- Public APIs
- Backend APIs
- REST vs gRPC protocols

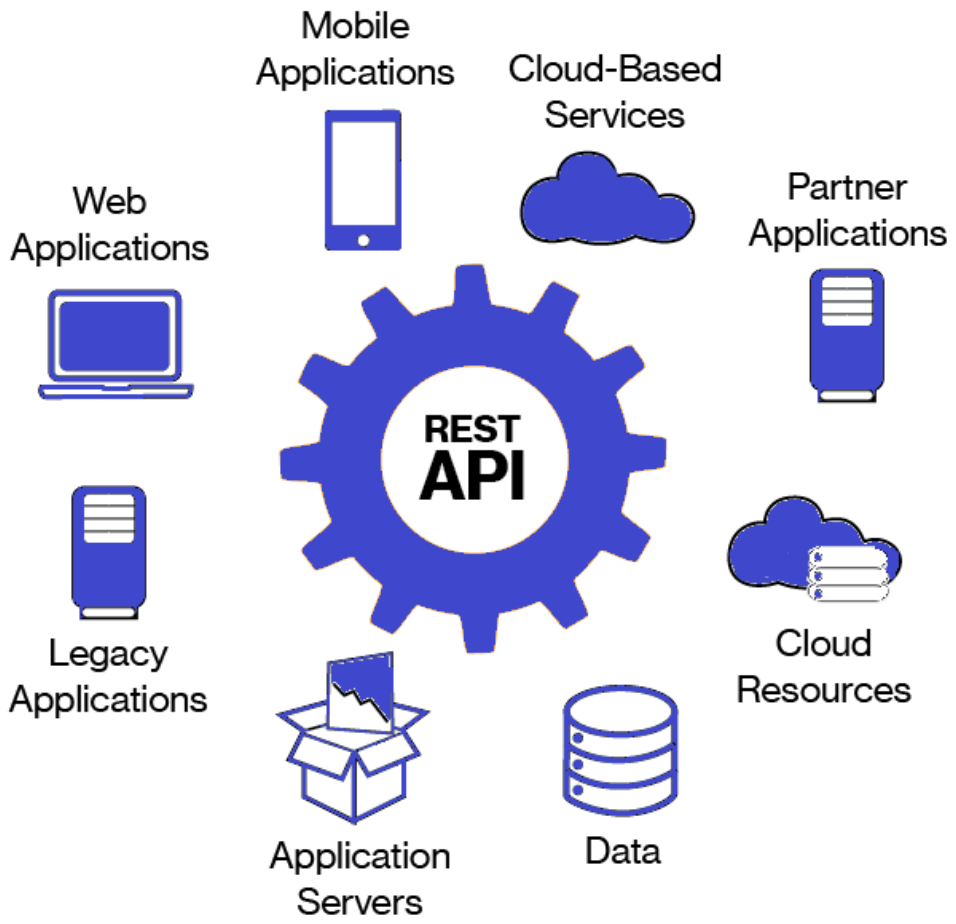


RESTful API design for Microservices

- Request/Response communication
- REST HTTP protocol, and implementing

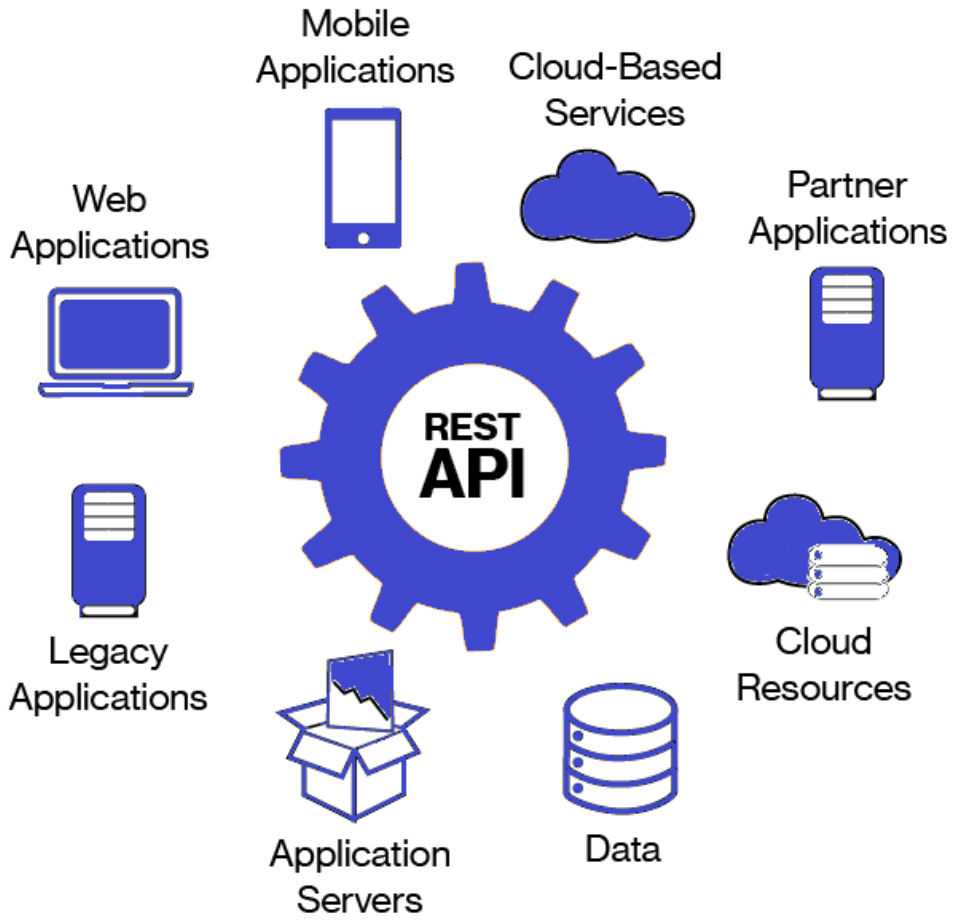
HTTP verbs like GET, POST, and PUT.

- REST (Representational State Transfer)
- Stateless
- Uniform Interface
- Cacheable
- Client-Server
- Layered System
- Code on Demand



What is RESTFul APIs?

- HTTP protocol with HTTP methods (GET, POST, PUT, DELETE)
- Richardson Maturity Model
- Level 0: Define one URI
- Level 1: Create separate URIs
- Level 2: Use HTTP methods
- Level 3: Use hypermedia



HTTP Methods

GET	retrieve information
HEAD	retrieve resource headers
POST	submit data to the server.
PUT	save an object at the location
DELETE	delete the object at the location

RESTful API design for Microservices - Part 2

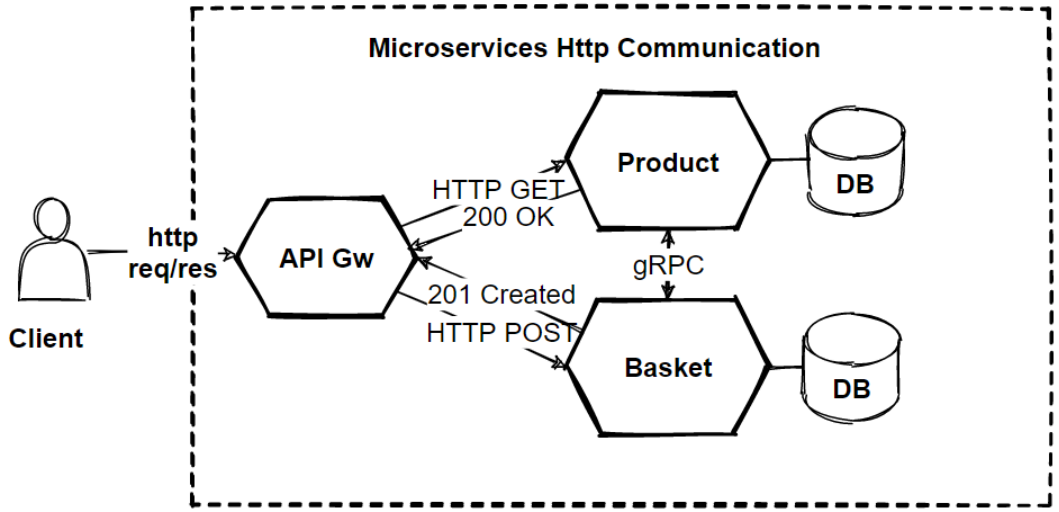
- Focus on the business entities that we expose APIs
- E-commerce entities might be customers and orders
- Design Microservice

APIs

For example :

https://eshop.com/orders // Correct

https://eshop.com/create-order // Wrong



RESTful API design for Customer Entity

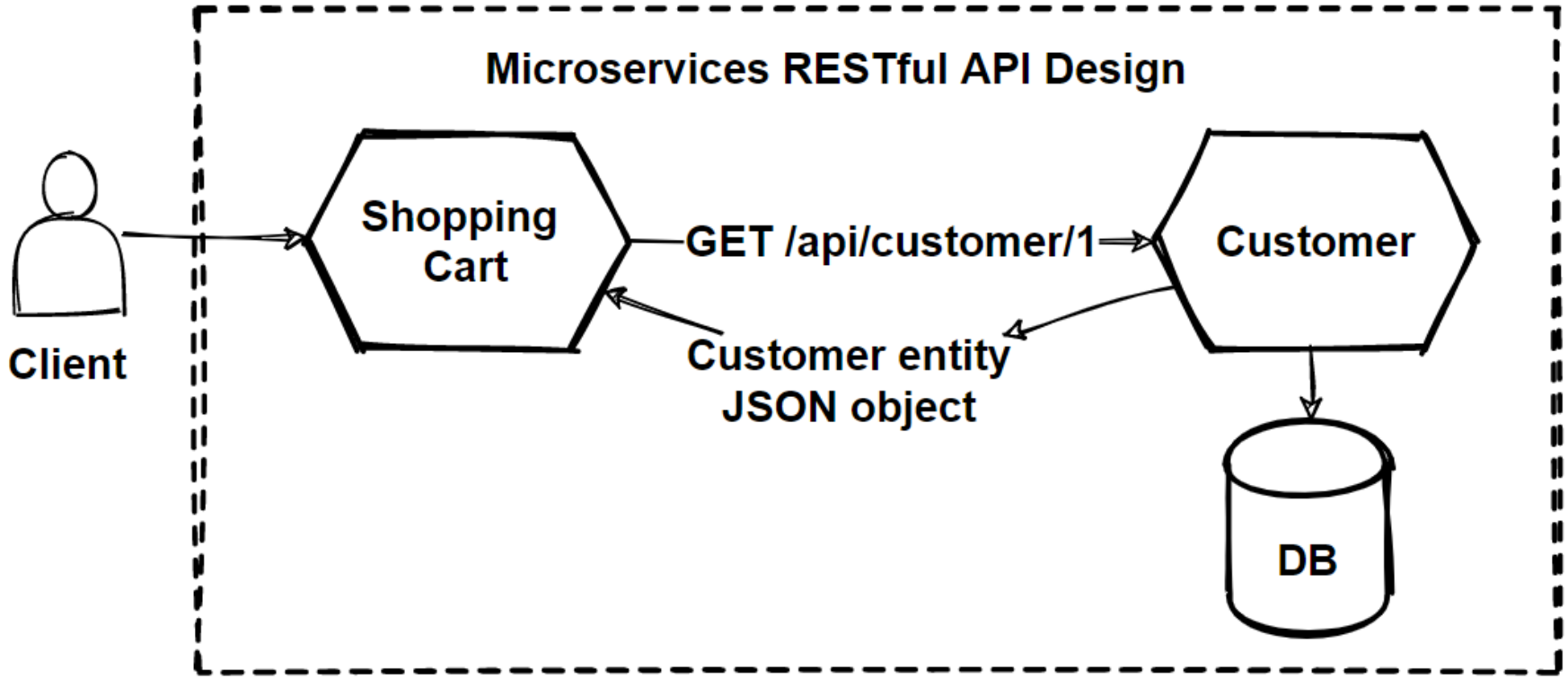
- Design Customer and Order entities and resources
- Different Resource Urls with http verbs like get-put-post and delete

For example :

Resource	POST	GET	PUT	DELETE
/customers	Create a new customer	Retrieve all customers	Bulk update of customers	Remove all customers
/customers/1	Error	Retrieve the details for customer 1	Update the details of customer 1 if it exists	Remove customer 1
/customers/1/orders	Create a new order for customer 1	Retrieve all orders for customer 1	Bulk update of orders for customer 1	Remove all orders for customer 1

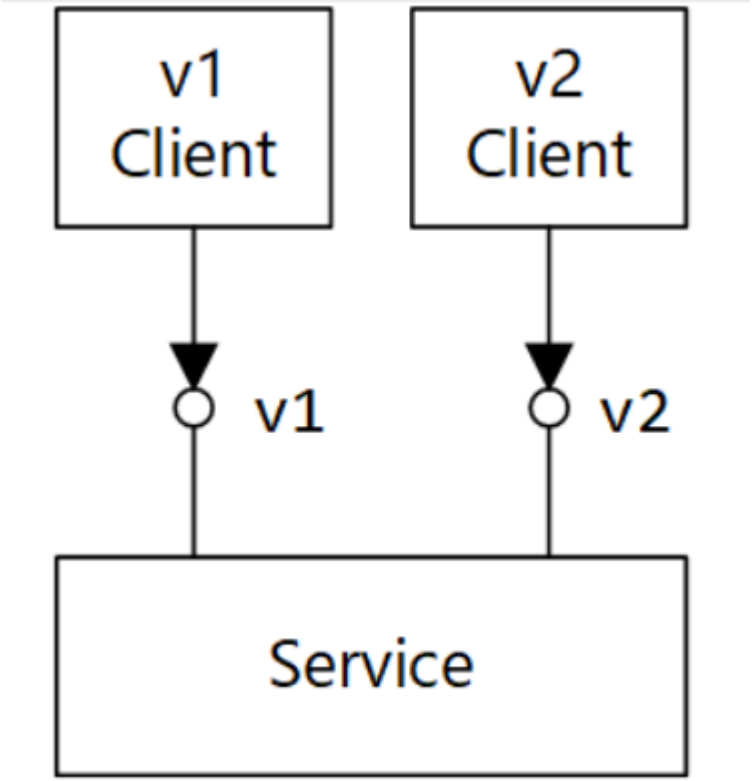
RESTful API design for Microservices

- Microservices don't share the same code base
- Customer entity model as a JSON object



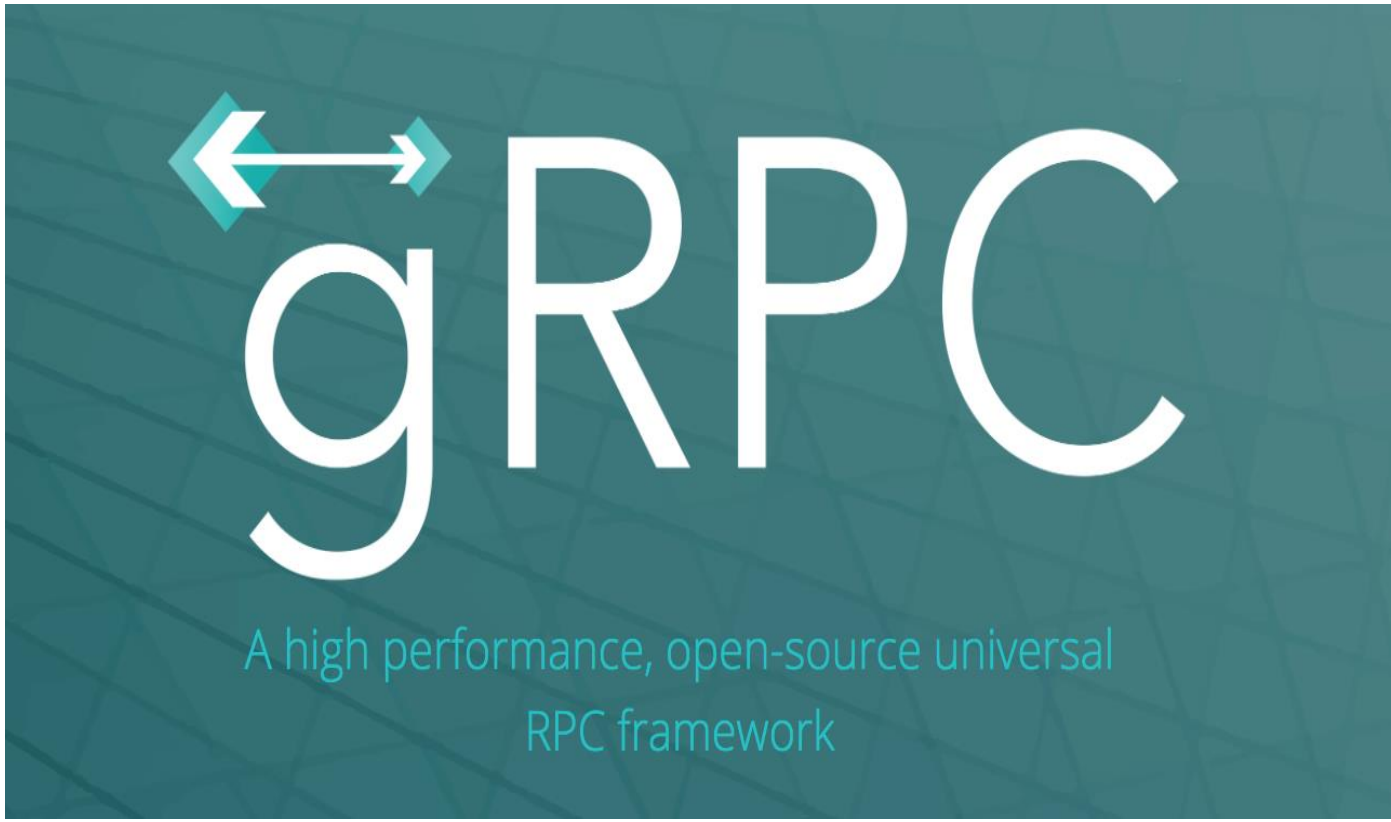
API versioning

- API changes may break some communications
- Deploy your services - required to add new features or bug fixes
- Backward compatible and not break any communications
- Best practice for the roll-out or canary deployments on Kubernetes



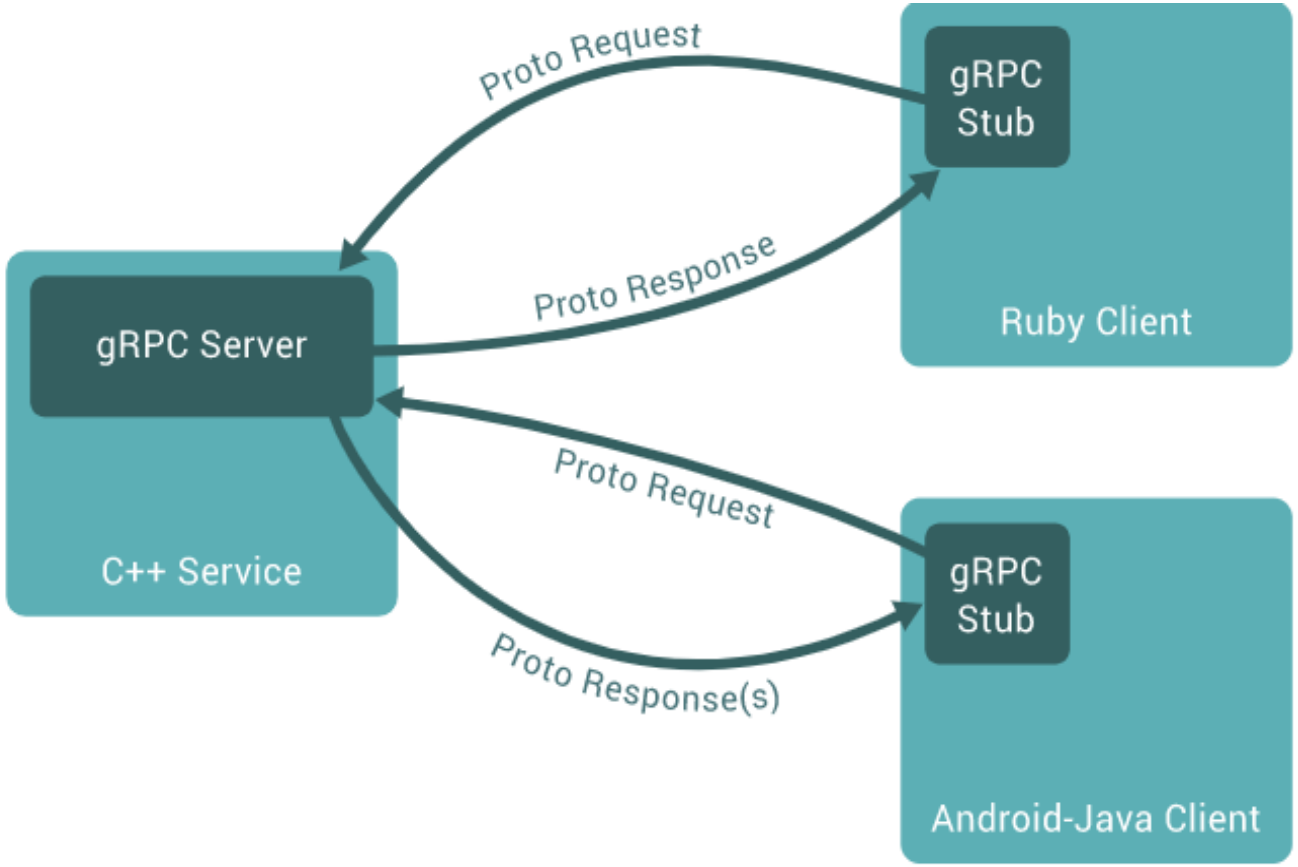
What is gRPC ?

- gRPC (gRPC Remote Procedure Calls)
- HTTP/2 protocol to transport binary messages
- Protocol Buffers, also known as Protobuf files
- Cross-platform client and server bindings



How gRPC works ?

- gRPC directly call a method on a server
- Build distributed applications and services
- Defining a service that specifies methods that can be called remotely
- Can be written in any language that gRPC supports



Advantages of gRPC

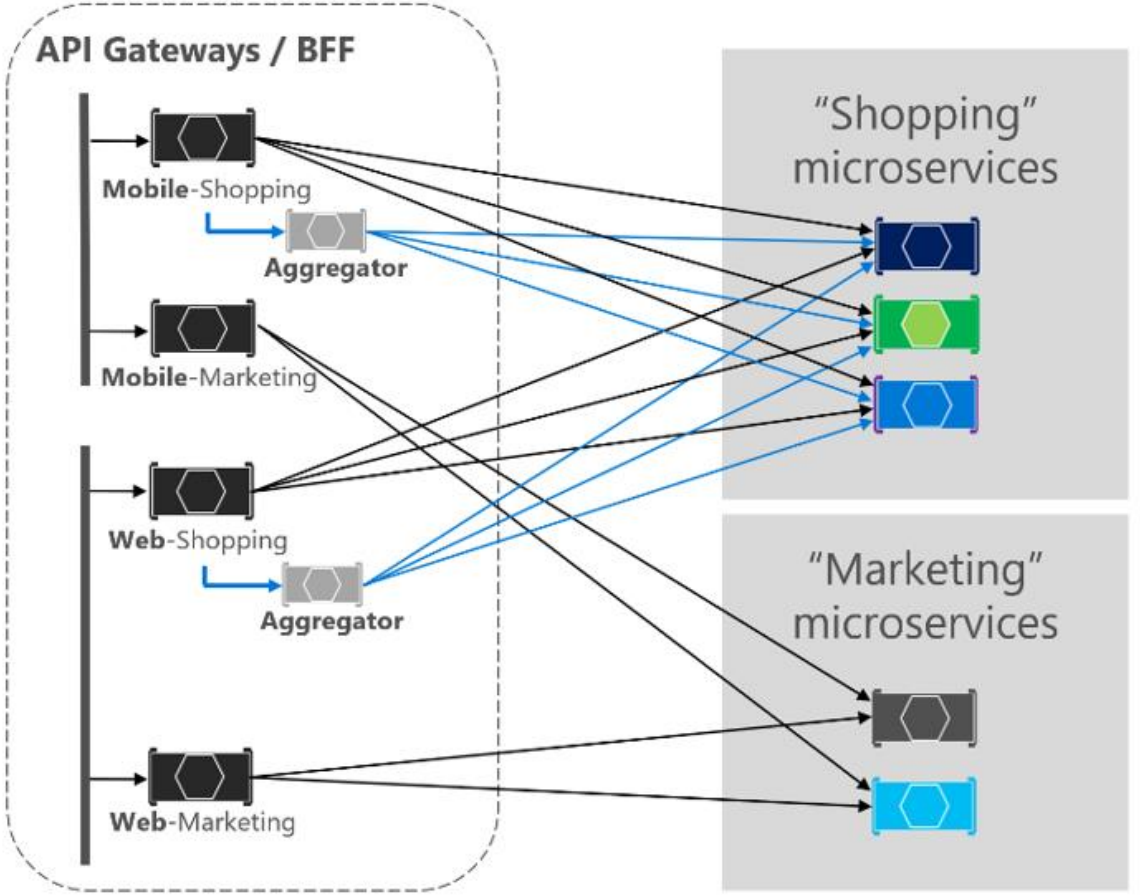
- Using HTTP / 2
- Binary serialization
- Supporting a wide audience with multi-language / platform support
- Open Source and the powerful community behind it
- Supports Bi-directional Streaming operations

gRPC vs REST

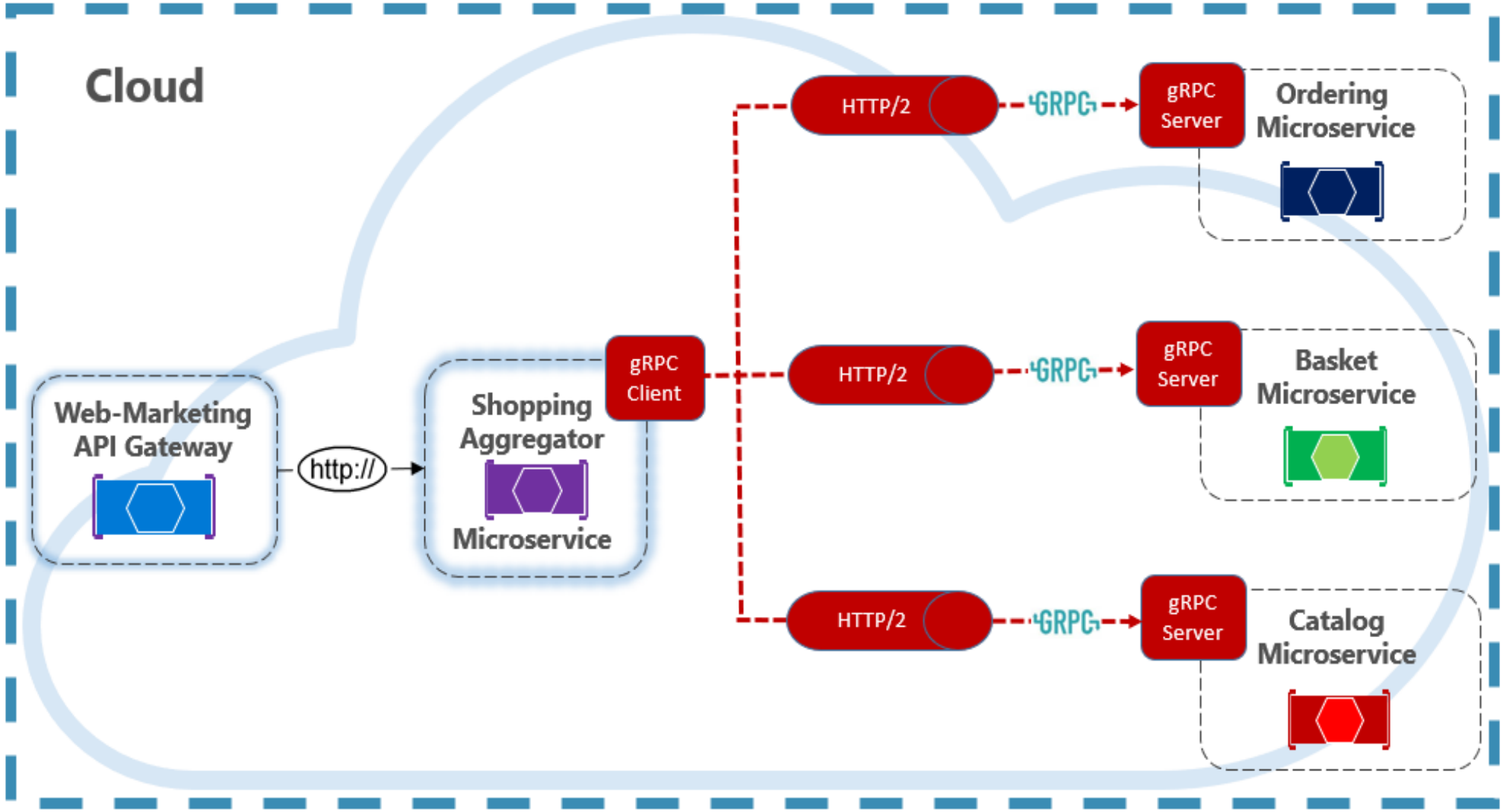
	Goal	REST (HTTP/JSON)	gRPC
COMPATIBILITY	Single source of truth	✗	✔
	Multi-platform + languages built in	✗	✔
	Handle non-breaking changes.	✗	✔
PERFORMANCE	Network: connection handling	Manual, 1 per call ✗	Built-in, Multi per conn ✔
	Speed: Transmission of data	Human-readable Text ✗	Binary ✔
	CPU: Improved resource usage	✗	✔
MAINTENANCE	Tracing	Manual ✗	Easy to plug in ✔
	Logging	Manual ✗	Easy to plug in ✔
	Monitoring	Manual ✗	Easy to plug in ✔

gRPC usage of Microservices Communication

- Synchronous backend microservice-to-microservice communication
- Polyglot environments
- Low latency and high throughput communication
- Point-to-point real-time communication
- Network constrained environments



Example of gRPC in Microservices



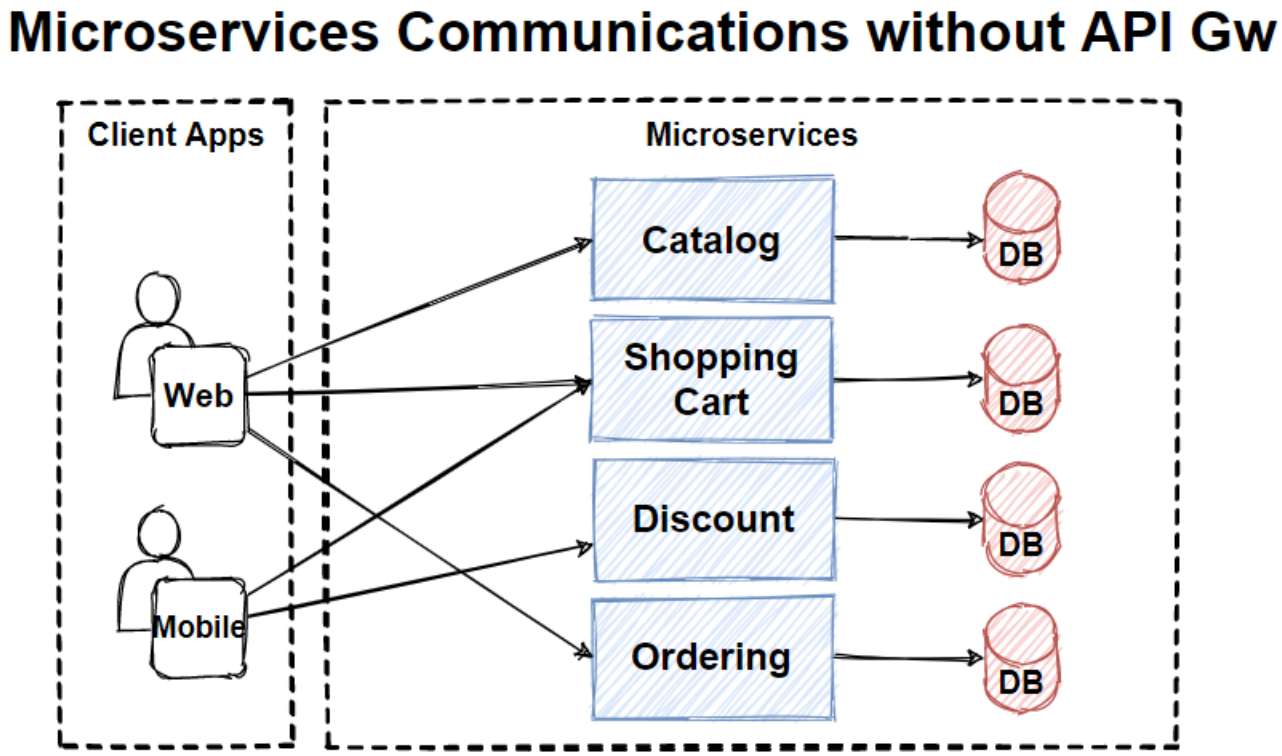
Section 9
Microservices Communication Patterns - API Gateways

Microservices Communication Design Patterns - API Gateways and Direct client-to-microservice communication.

Drawbacks of the direct client-to-microservices communication

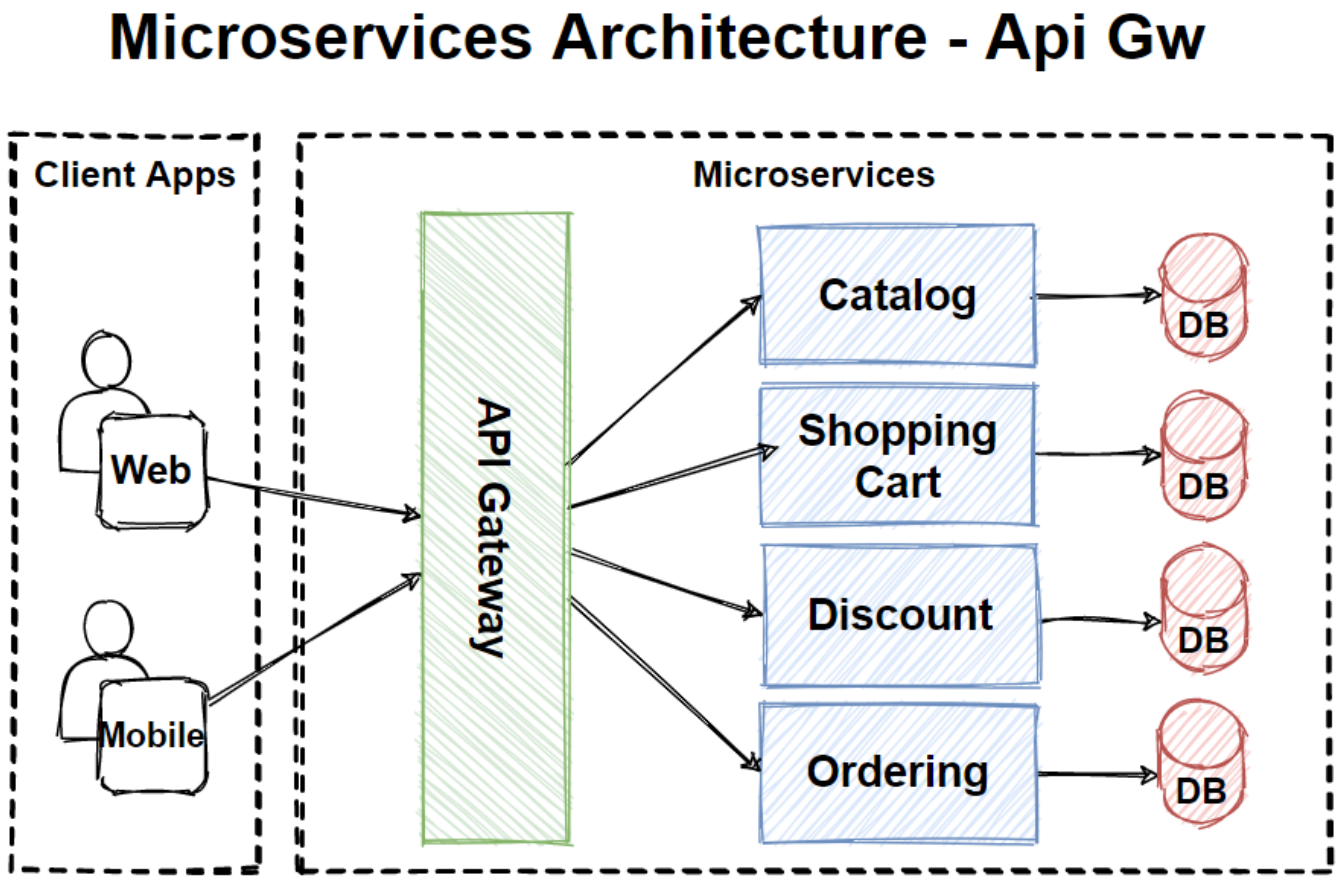
- Direct client-to-microservice communication problems
- Expose fine-grained endpoints
- Client try to handle multiple call
- Chatty communications
- Increases latency and

Complexity on the UI side



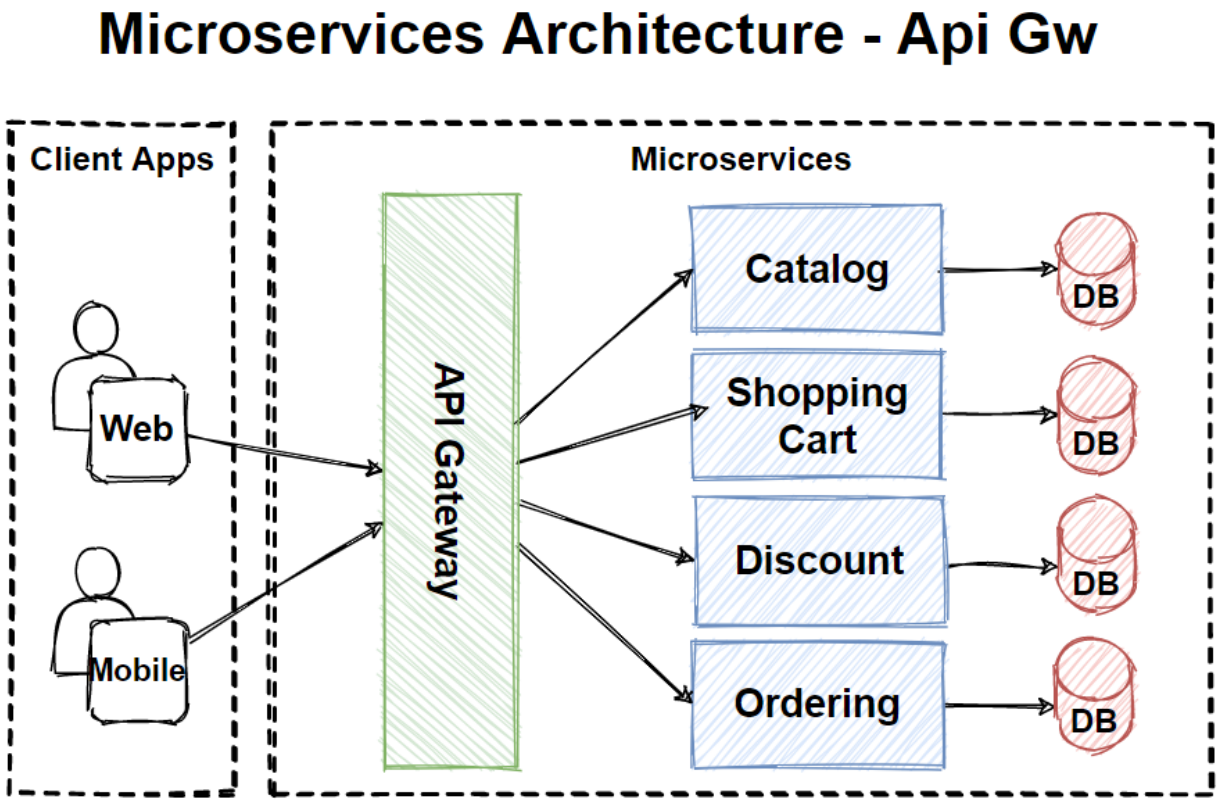
Microservices Communication Design Patterns - API Gateways

- API gateway pattern
- Cross-cutting concerns
- Routing to internal microservices
- Aggregate several microservice



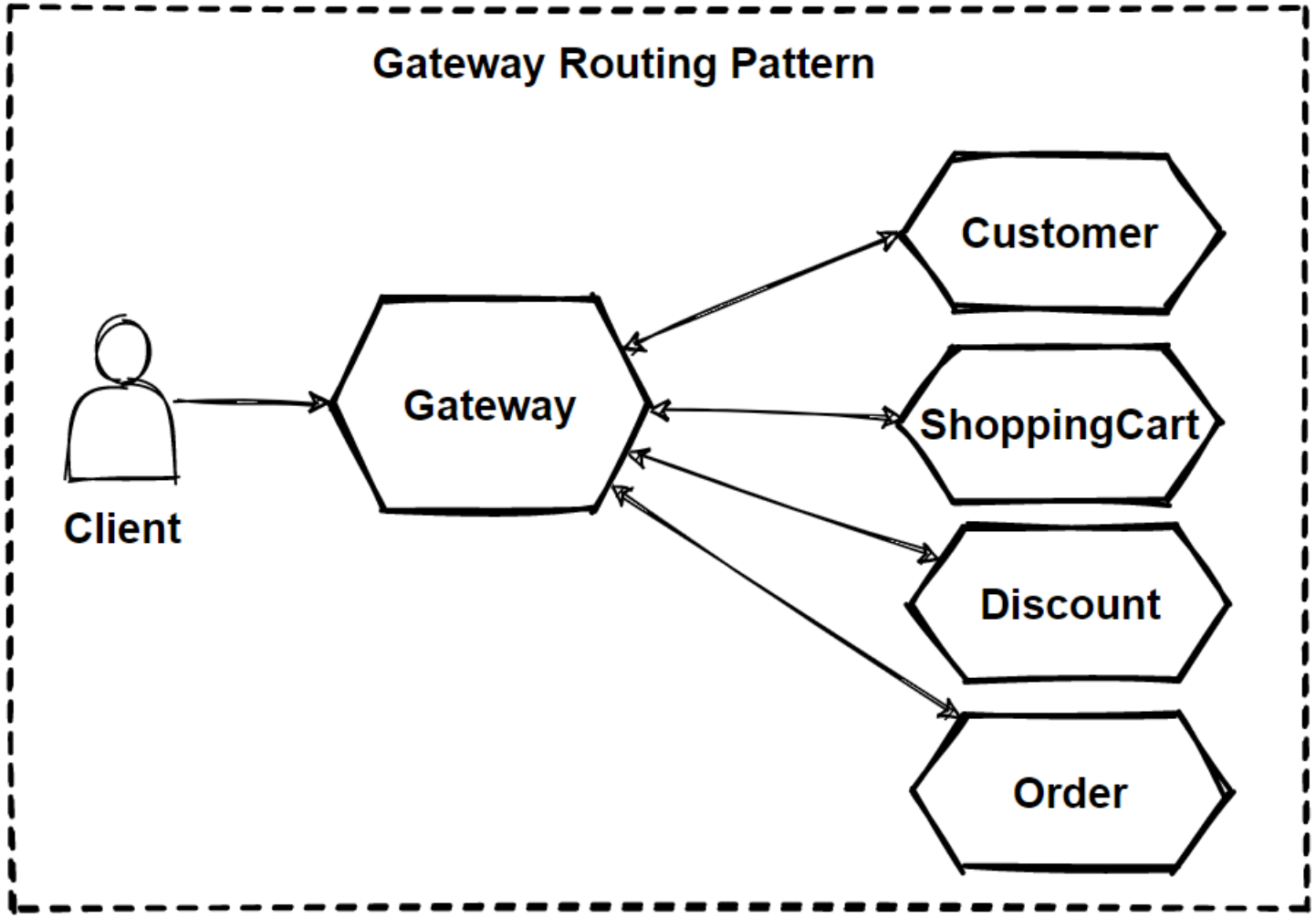
3 Main Pattern = API Gateways

- Gateway Routing pattern
- Gateway Aggregation pattern
- Gateway Offloading pattern



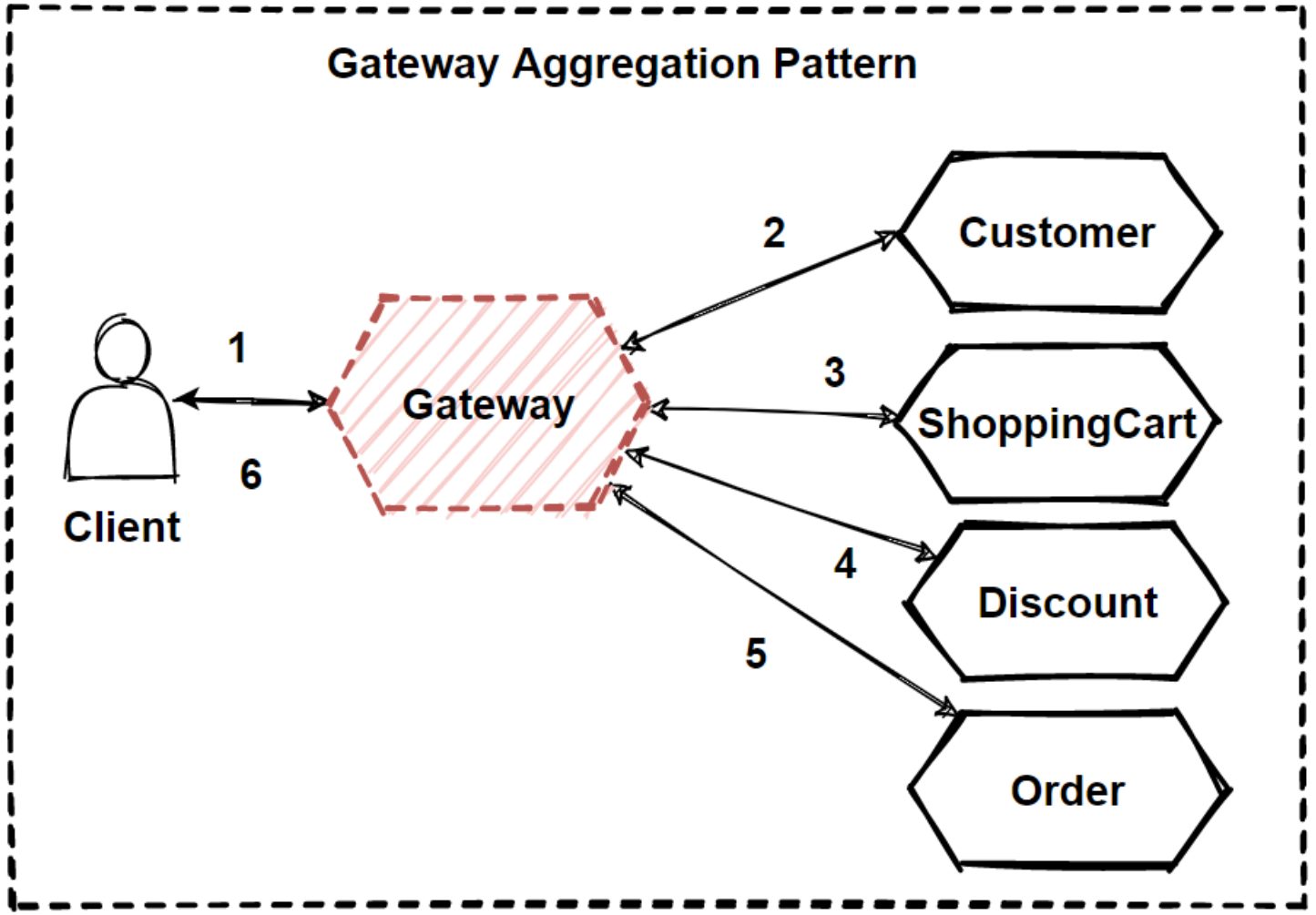
Gateway Routing pattern

- Route requests to multiple microservices
- Exposing a single endpoint
- Layer 7 routing
- Protocol Abstraction
- Centralized Error Management



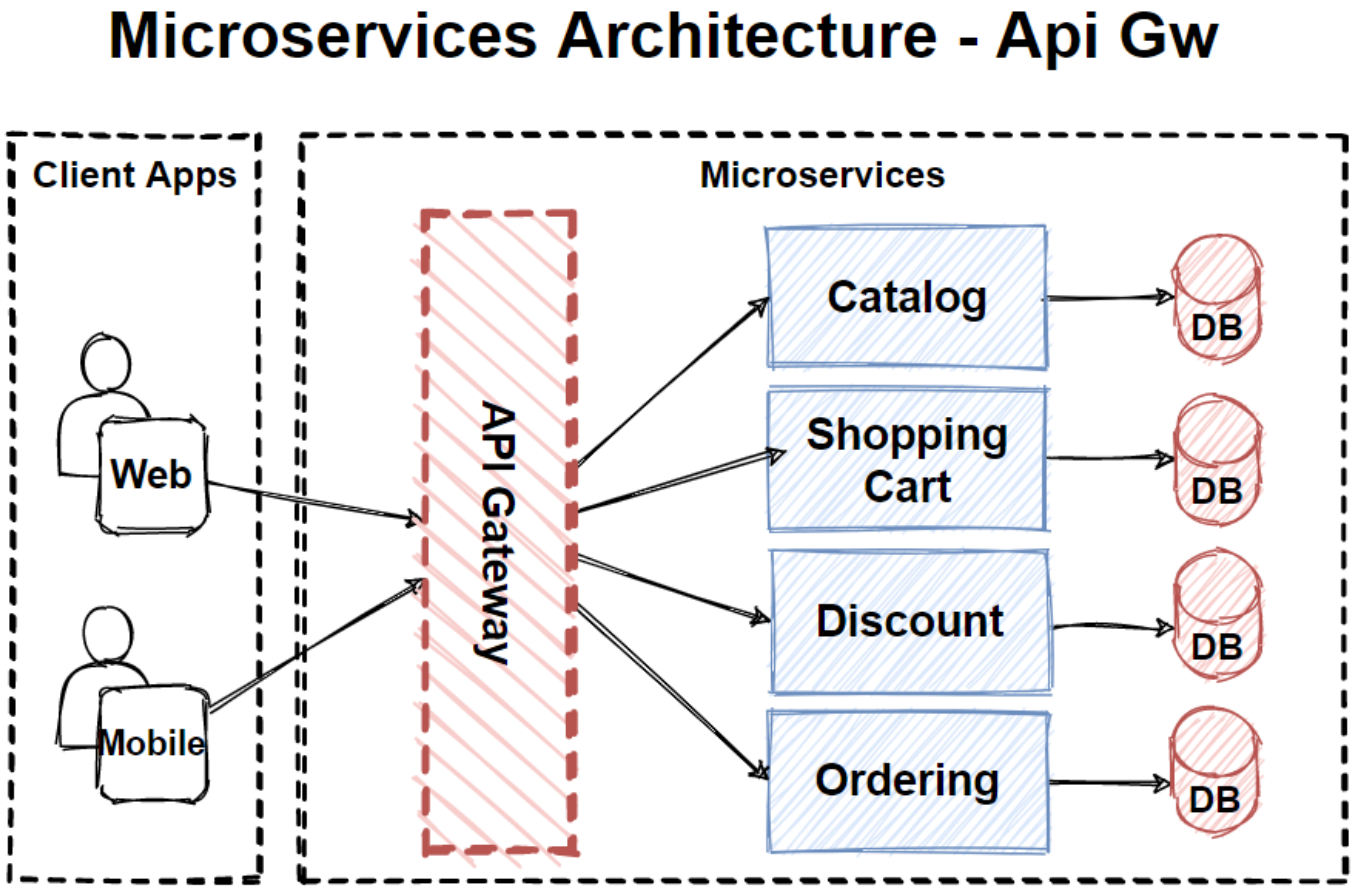
Gateway Aggregation pattern

- Aggregate multiple internal requests
- Different backend microservices
- Manage direct access services
- Reduce chattiness communication
- Dispatches requests



API Gateways Pattern

- Complex large microservices based applications
- Facade pattern
- Reverse proxy or gateway routing
- Routing requests from client backend services
- Single-point-of-failure risk



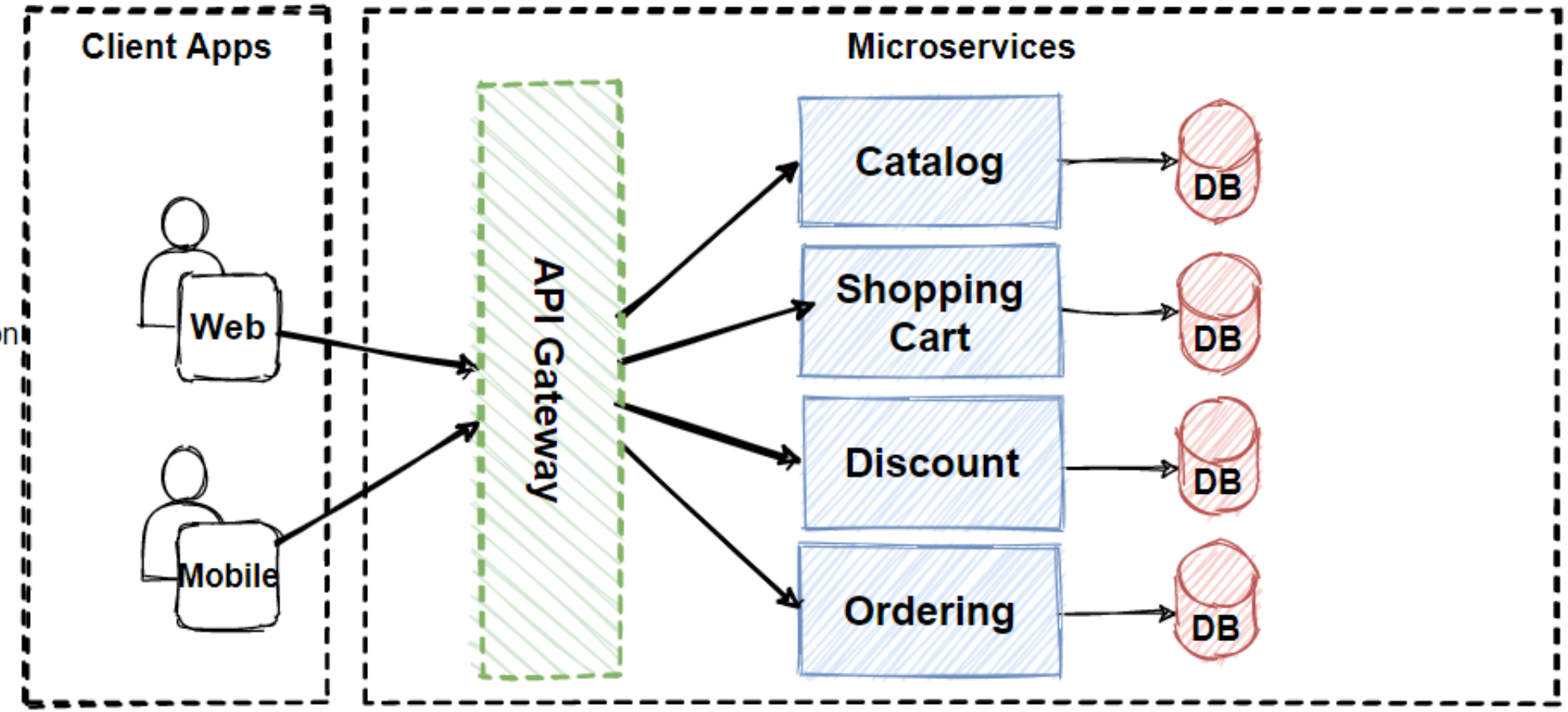
Main features in the API Gateway pattern

Ocelot Features	
Routing	Authentication
Request Aggregation	Authorization
Service Discovery with Consul & Eureka	Throttling
Load Balancing	Logging, Tracing
Correlation Pass-Through	Headers/Query String Transformation
Quality of Service	Custom Middleware

Microservices Architecture - Api Gw

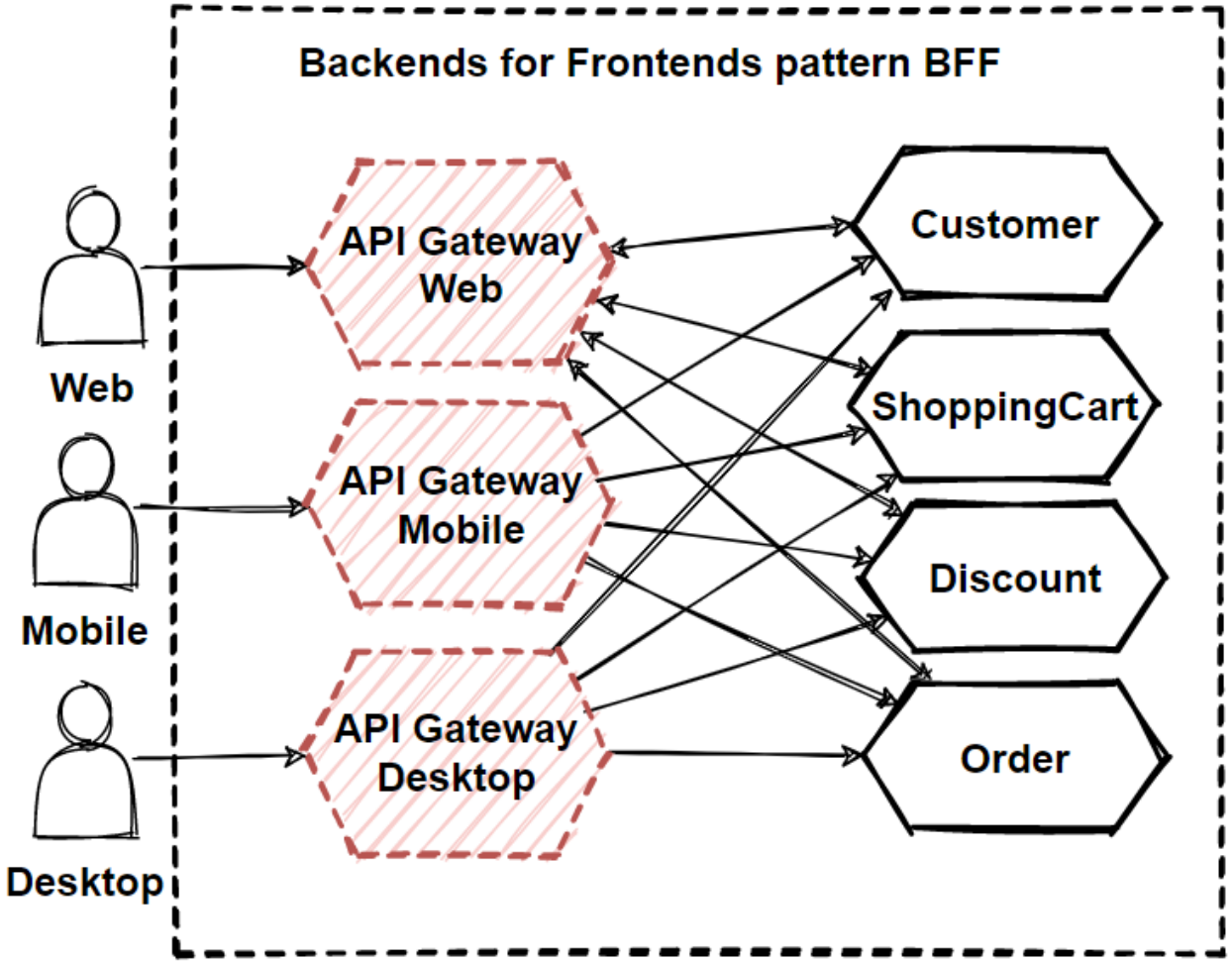
Principles

- KISS
- YAGNI
- SoC
- SOLID
- Gateway Routing
- Gateway Aggregation
- Gateway Offloading
- Api Gw
- Database per Microservices



Backends for Frontends pattern BFF

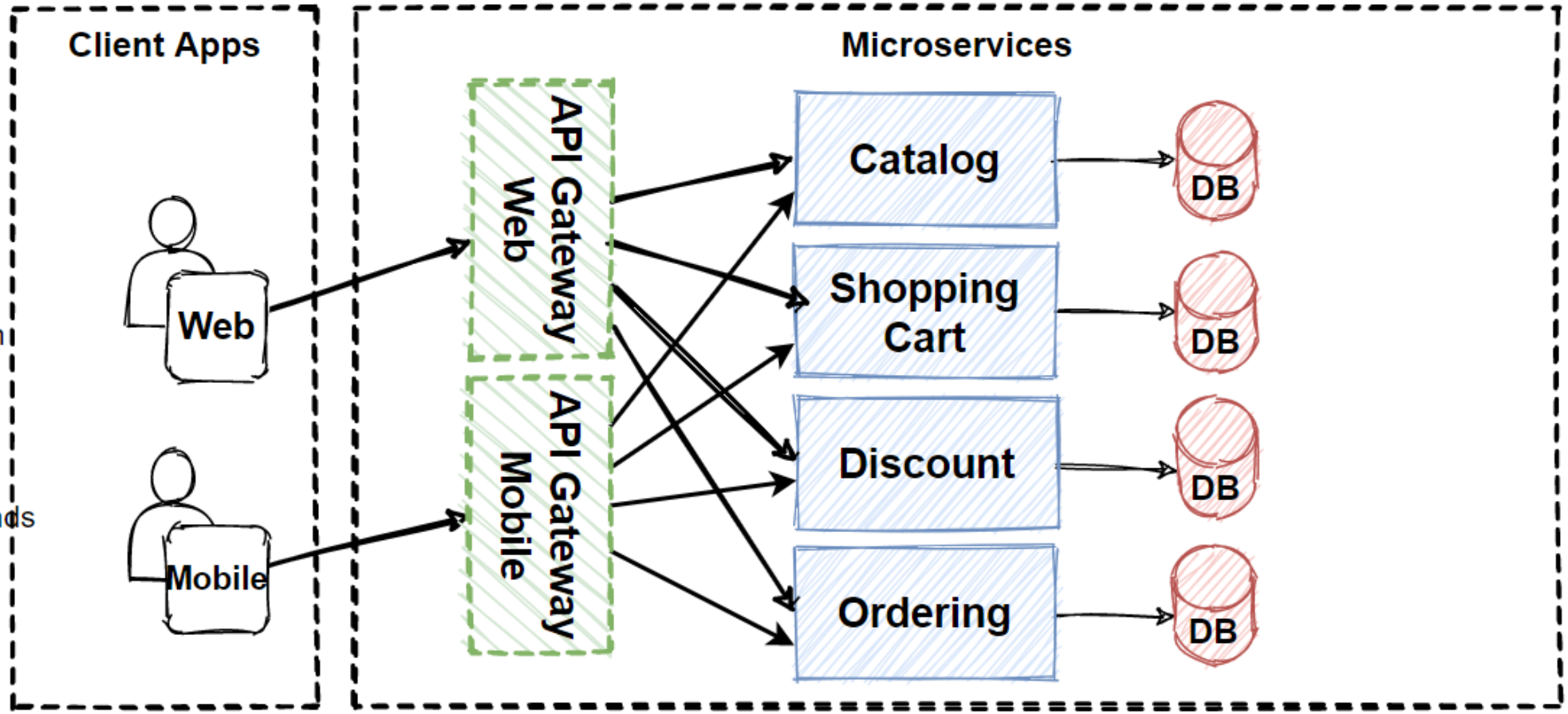
- Separate api gateways
- API Gateway for handling to routing and aggregate operations
- Grouping the client applications
- Avoid bottleneck of 1 API Gw
- Several api gateways as per user interfaces



Microservices Architecture - Api Gw - BFF

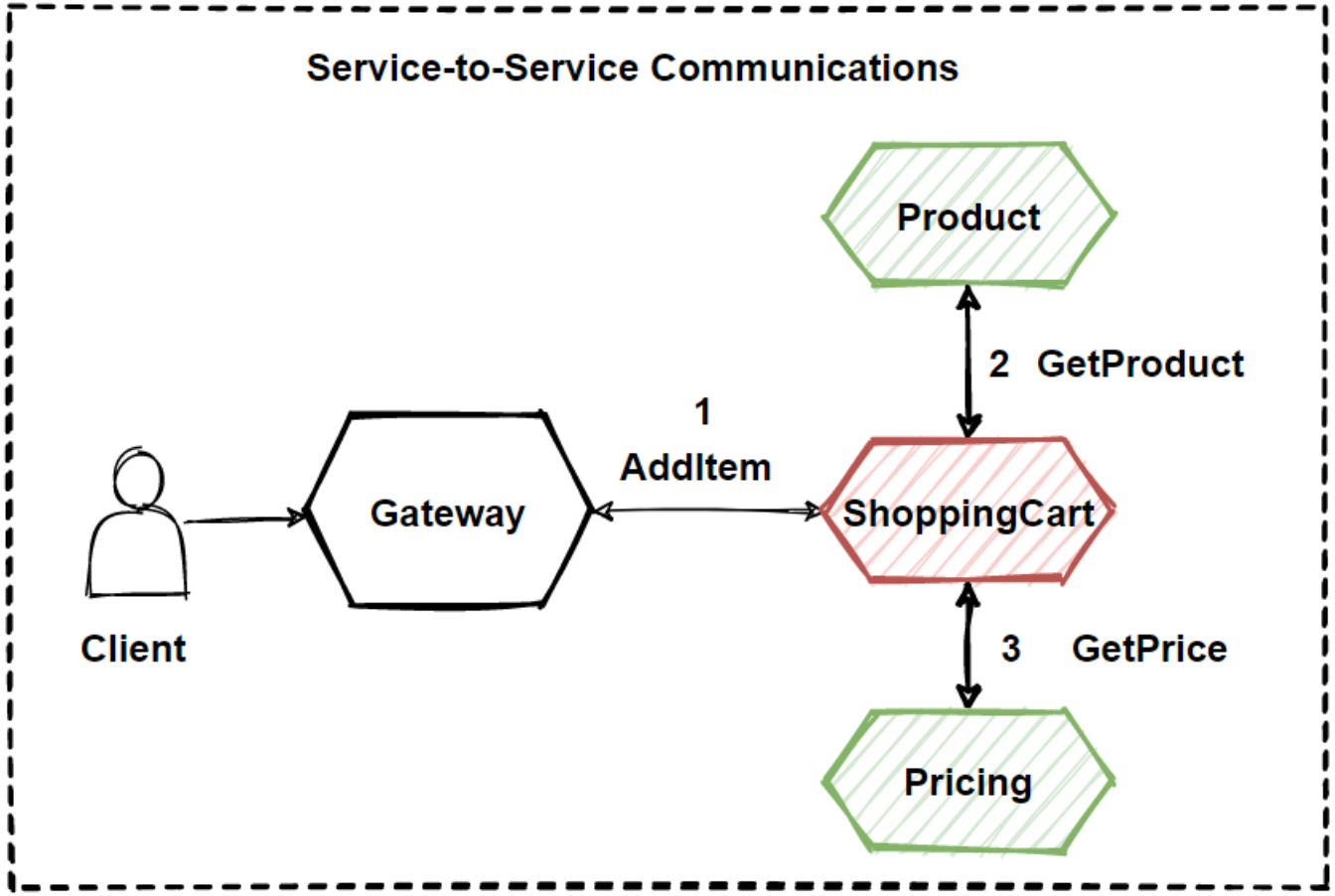
Principles

- KISS
- YAGNI
- SoC
- SOLID
- Gateway Routing
- Gateway Aggregation
- Gateway Offloading
- Api Gw
- Database per Microservices
- Backends for Frontends pattern BFF



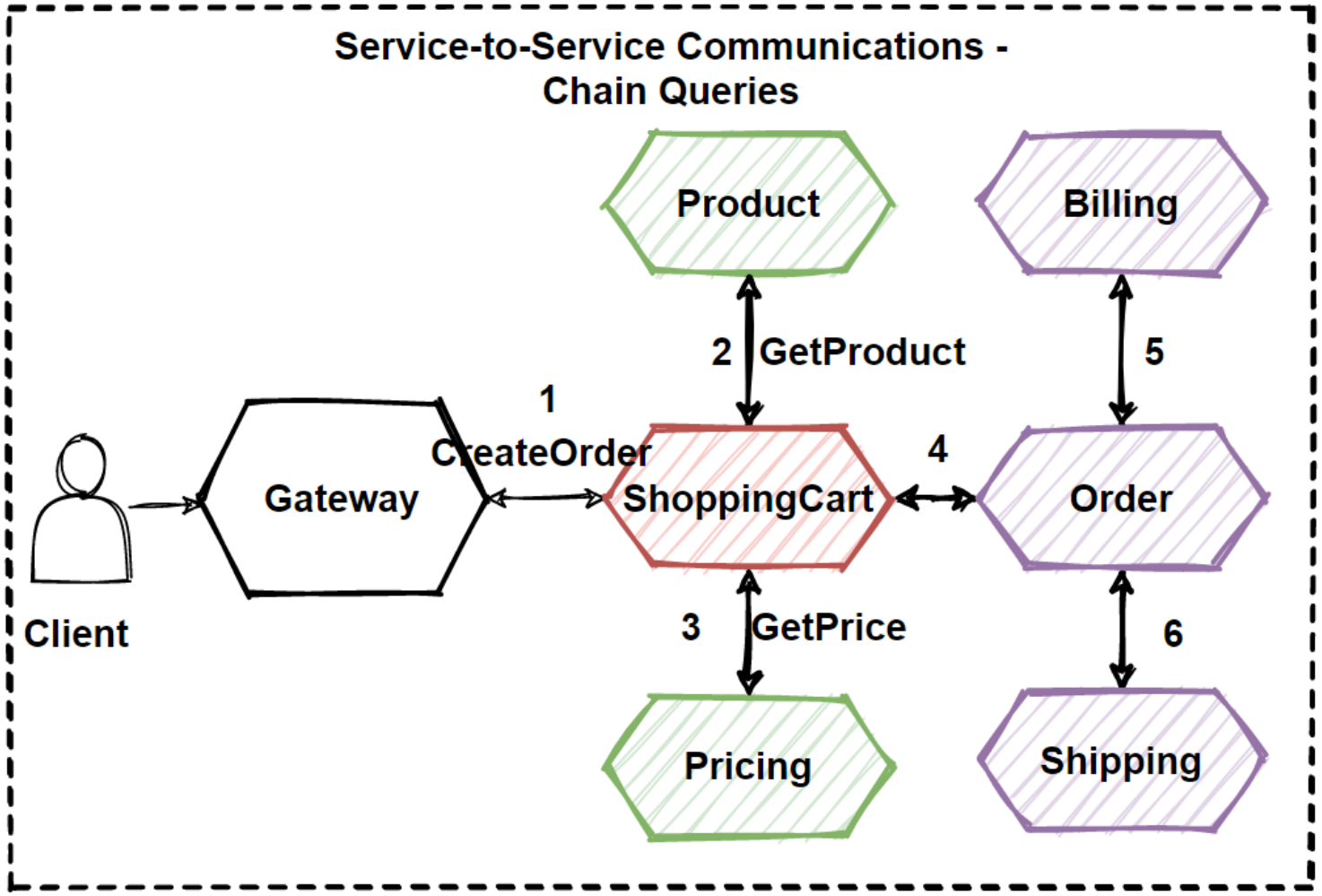
Service-to-Service Communications between Backend Internal Microservices

- Sync request comes from the clients
- Client requests are required tc visit more than one internal microservices
- Reducing inter-service communication
- Query request to internal microservices
- Dependent and coupling service



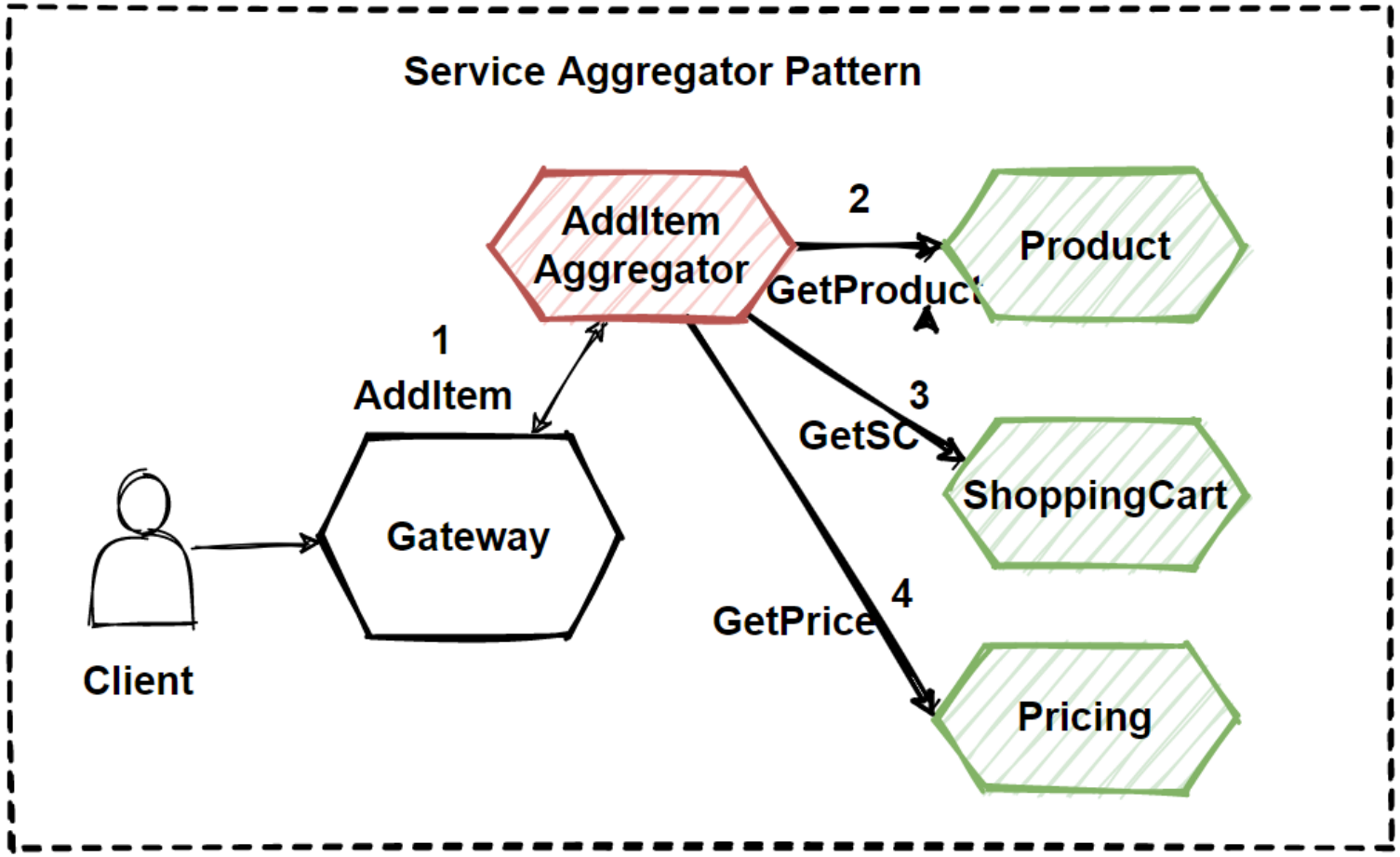
Service-to-Service Communications – Chain Queries

- Much chain calls
- Create order use case
- Request/Response sync Messaging pattern
- Increase latency and negatively impact the performance



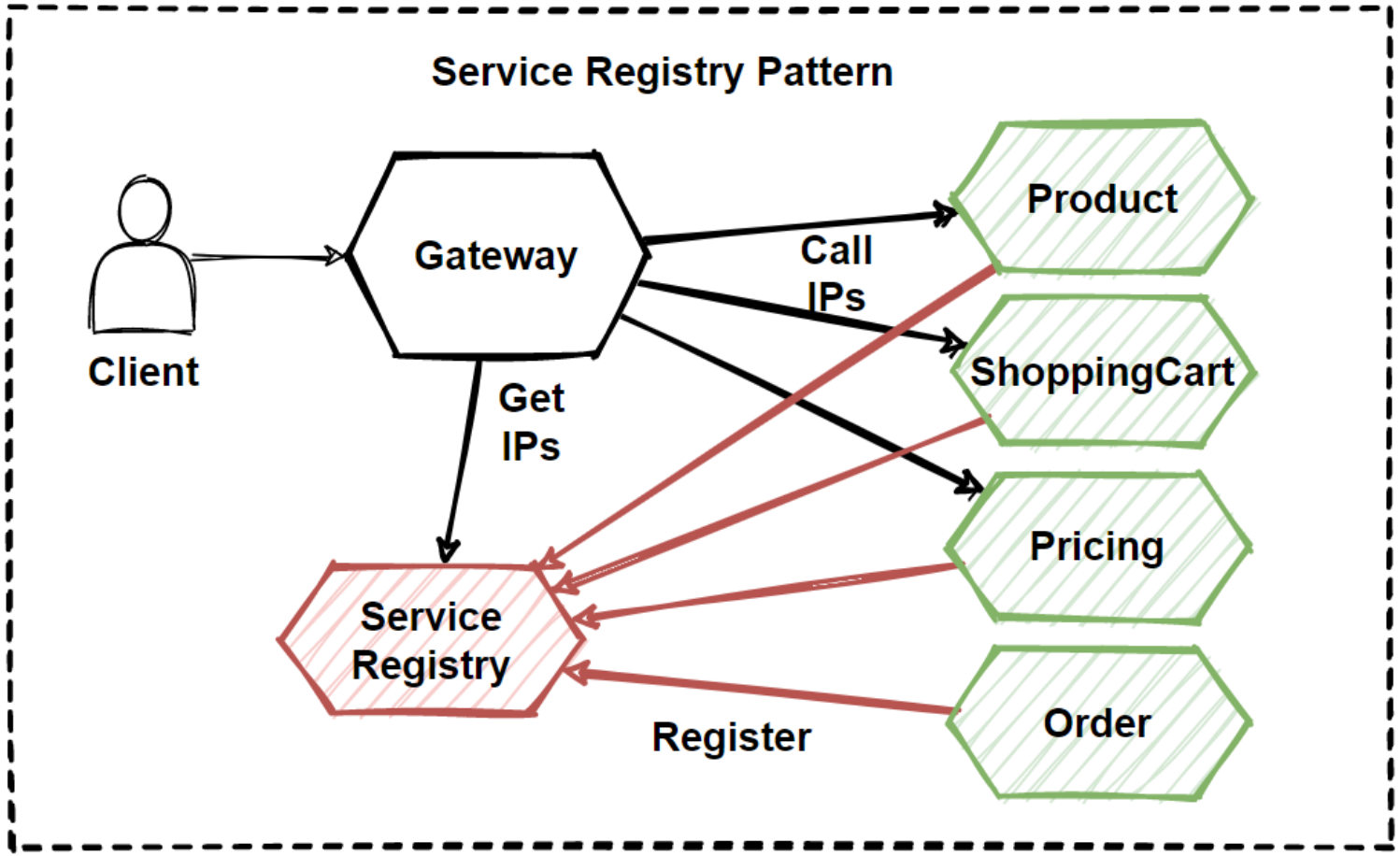
Service Aggregator Pattern

- Minimize service-to-service communications
- Receives a request from api gw
- Dispatchs requests of multiple internal backend microservices
- Combines the results



Service Registry Pattern

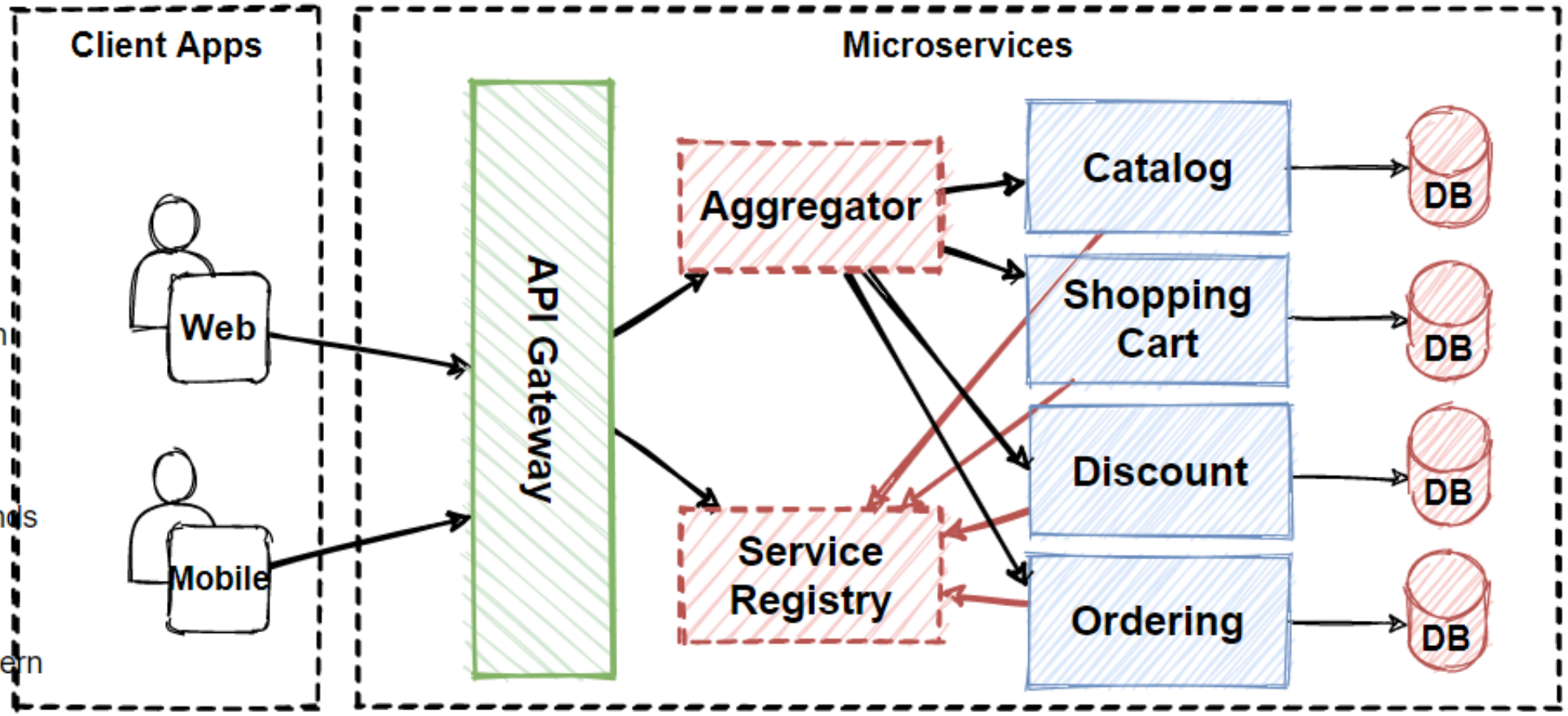
- Microservice Discovery Patterns
- Register and discover microservices
- Microservices should register to Service Registry
- Finds the location of microservice



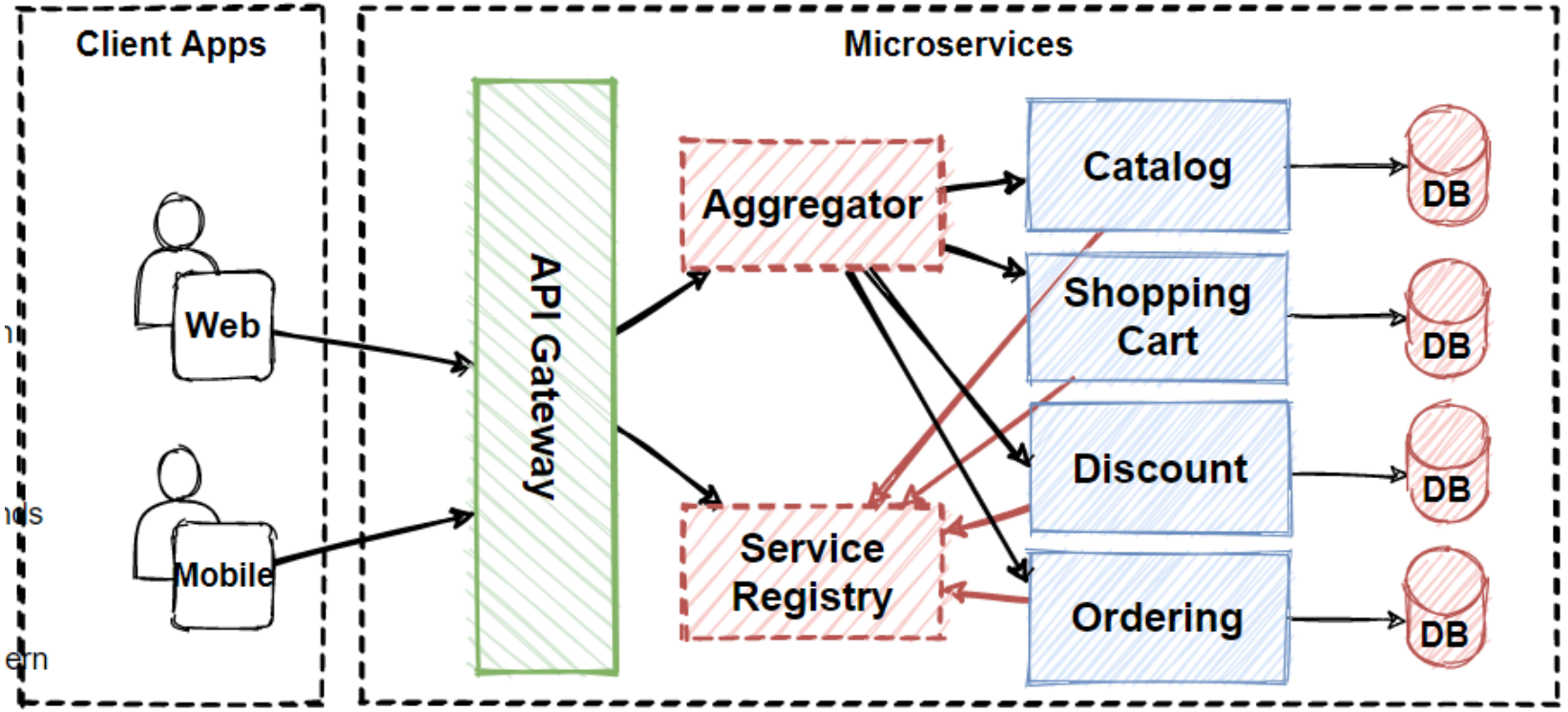
Microservices Architecture - Service Aggregator / Registry Patterns

Principles

- KISS
- YAGNI
- SoC
- SOLID
- Gateway Routing
- Gateway Aggregation
- Gateway Offloading
- Api Gw
- Database per Microservices
- Backends for Frontends pattern BFF
- Service Aggregator Pattern
- Service Registry Pattern



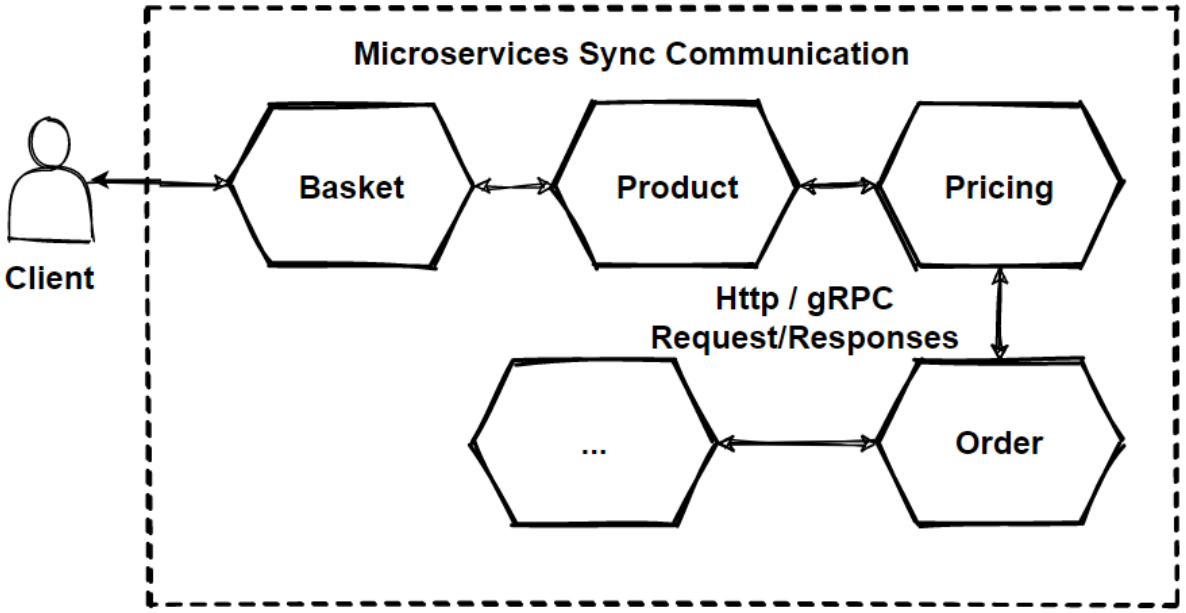
Adapting Technology Stack

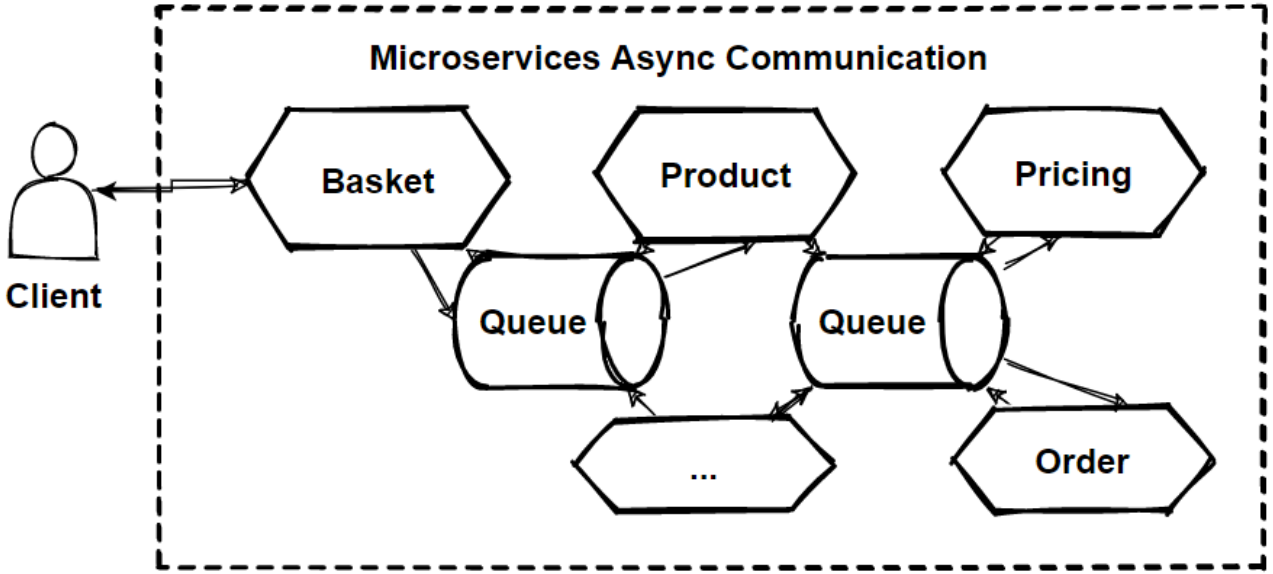


Section 10
Microservices Asynchronous Message-Based Communication

Asynchronous Message-Based Communication in Microservices Architecture

Microservices Communications

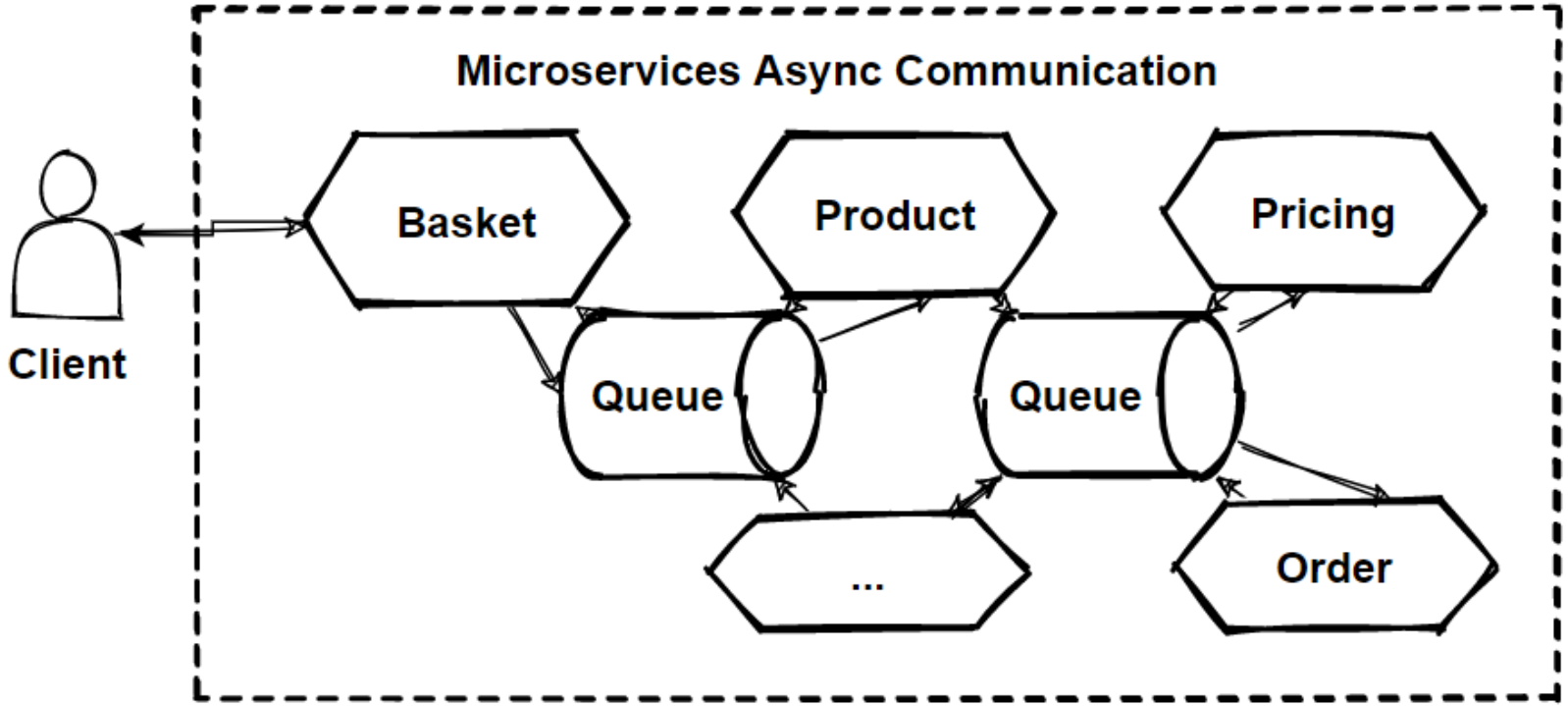




Microservices Asynchronous Communication

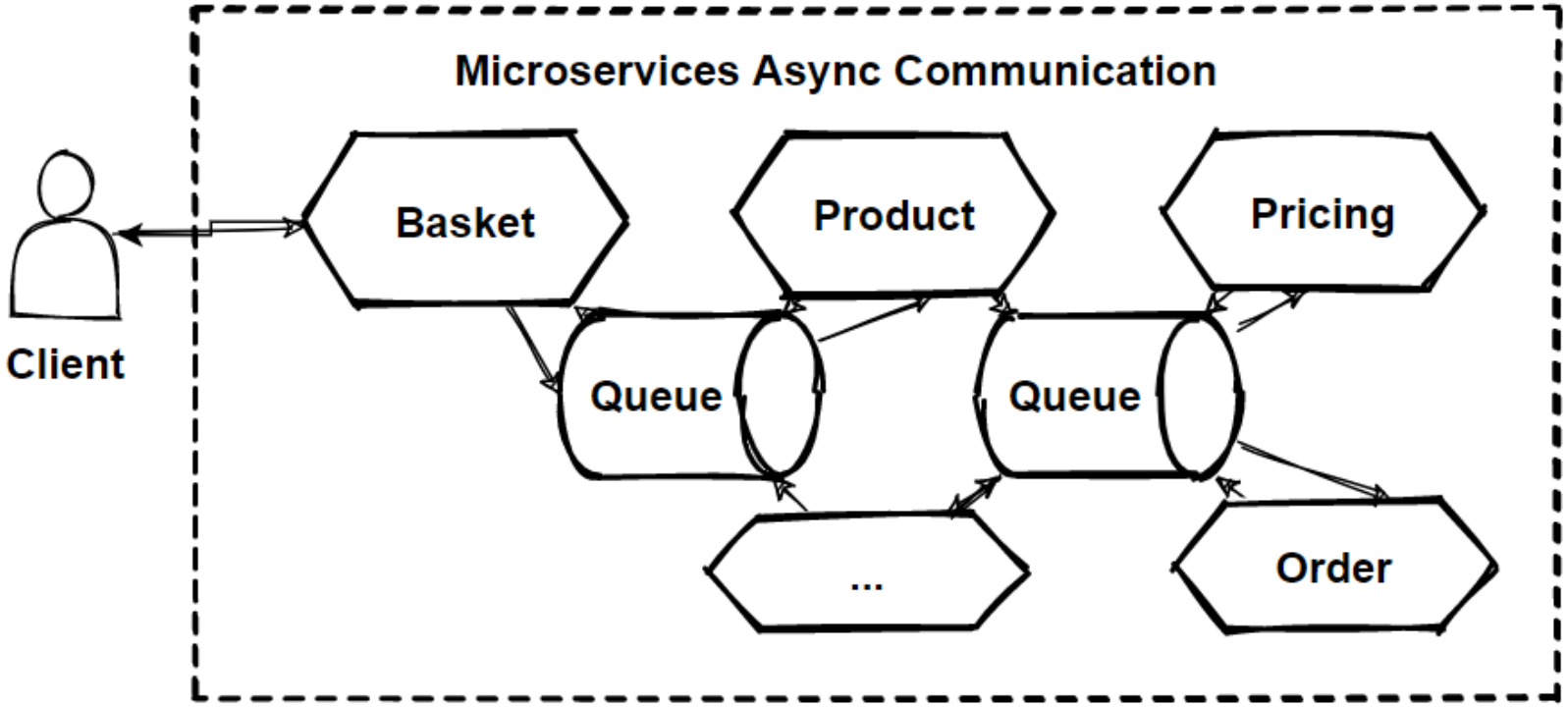
- Not wait Response and not have blocked a thread
- AMQP (Advanced Message QueuingProtocol)
- one-to-one(queue)
- one-to-many (topic)
- publish/subscribe
- Event-driven

microservices architecture

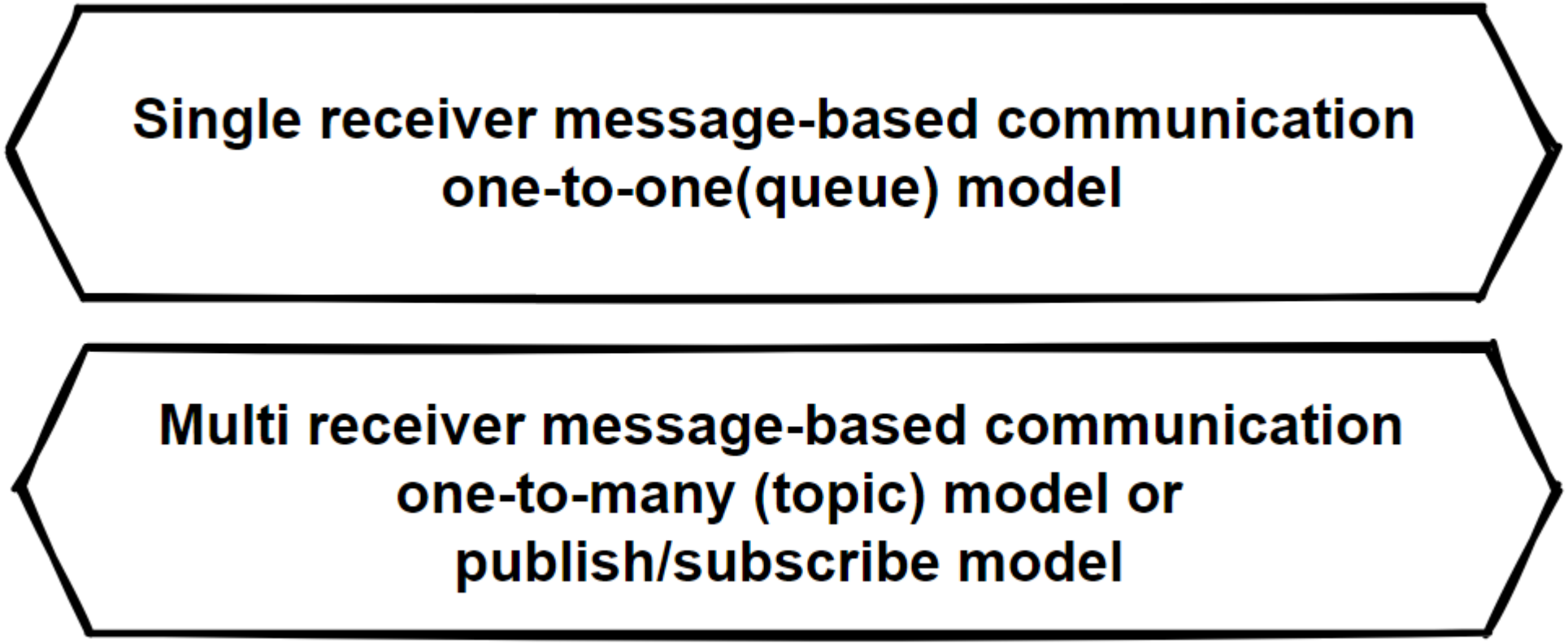


Asynchronous Message-Based Communication

- Multiple microservices are required to interact each other
- Without any dependency or make loosely coupled
- Message-based communication
- Eventual consistency
- Message broker systems
- AMQP protocol
- Kafka and Rabbitmq



2 Type of Asynchronous Messaging Communication

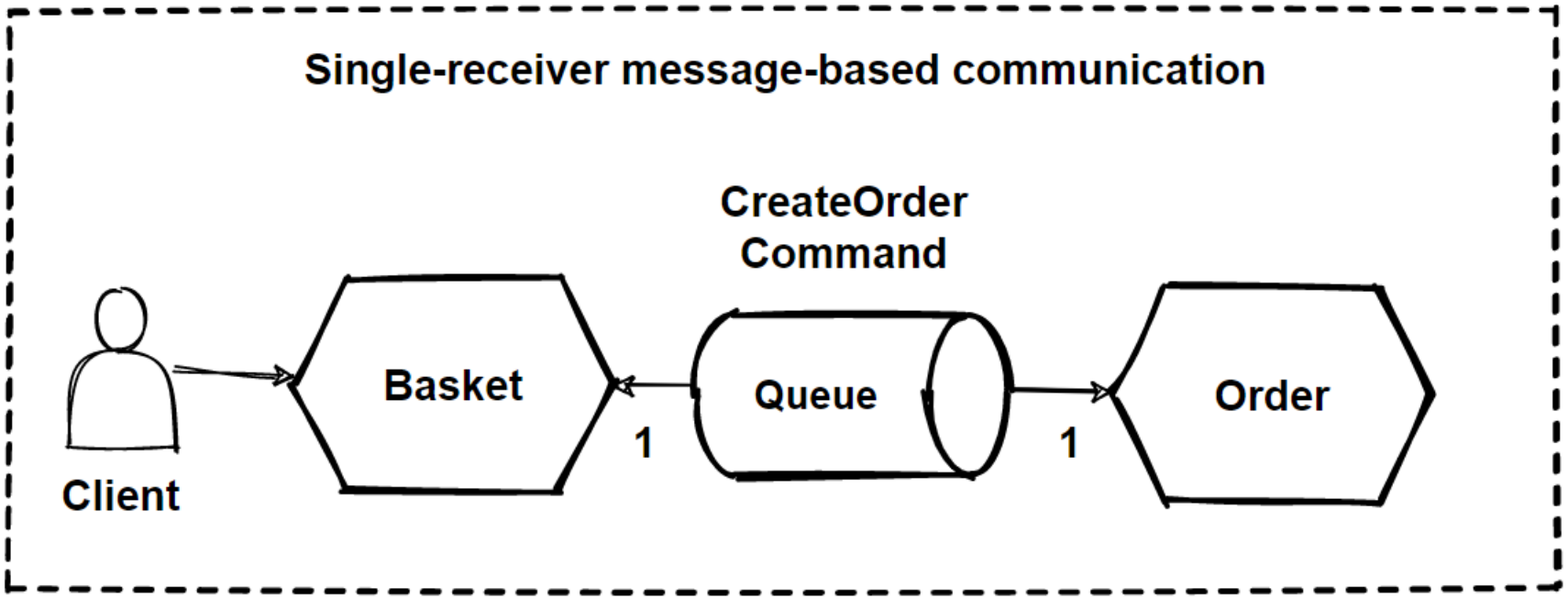


Single-receiver message-based communication

- One-to-one or point-to-point communications
- Without any dependency or make loosely coupled
- Message-based

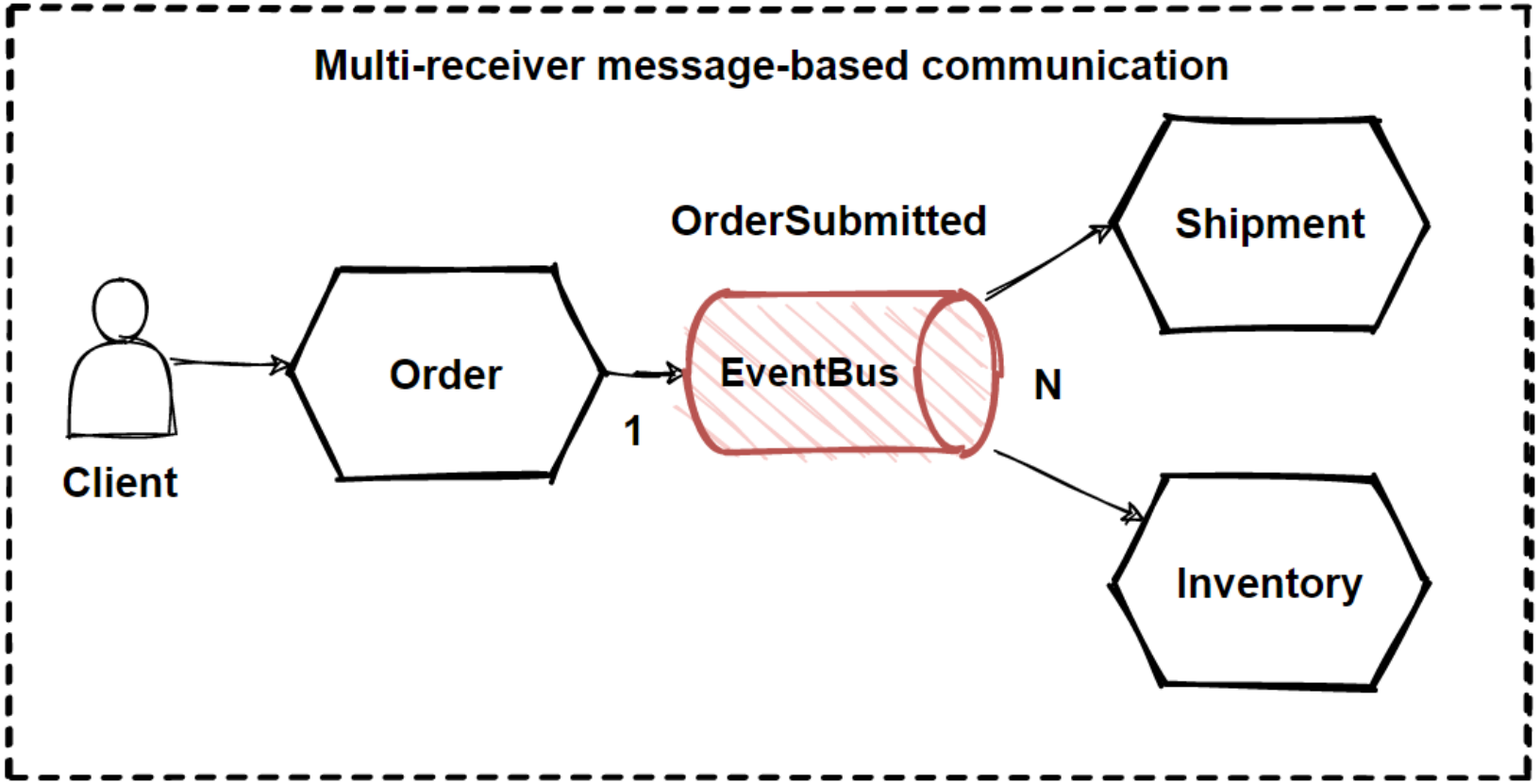
Communication

- CreateOrder event

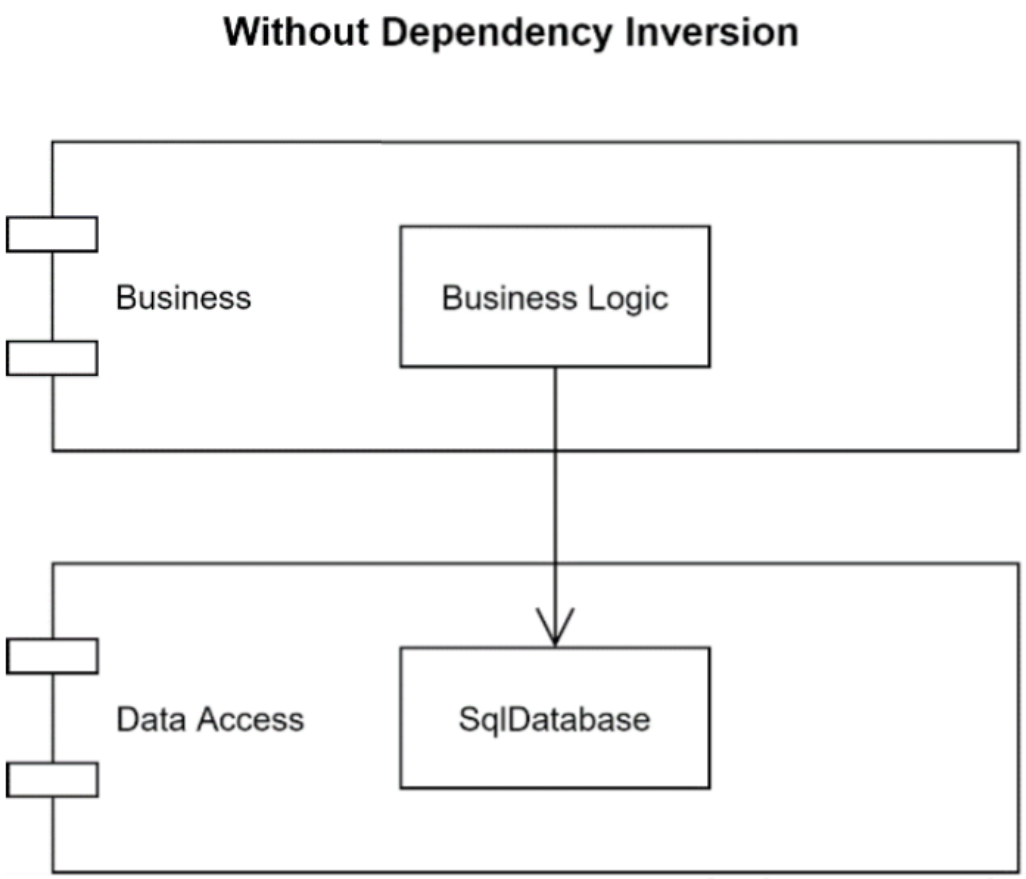


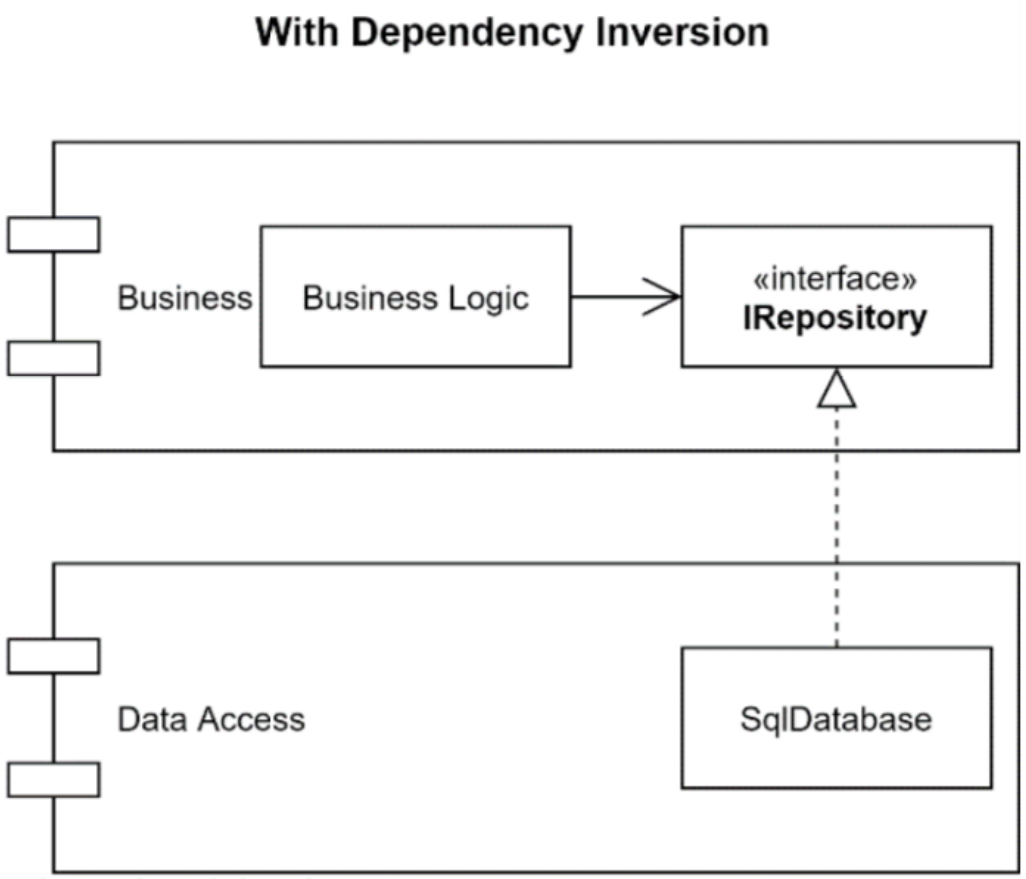
Multiple-receiver message-based communication

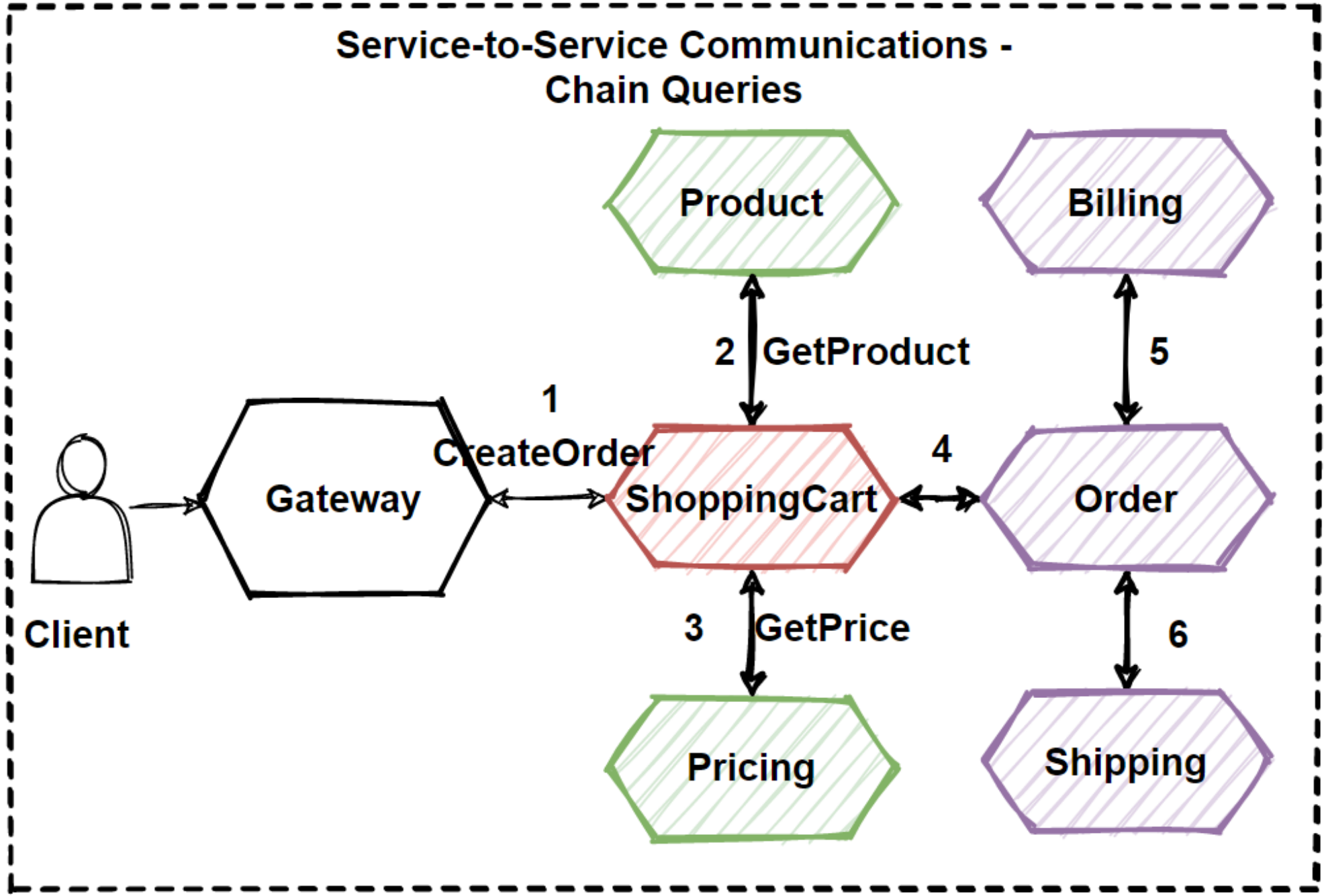
- One-to-many or publish/subscribe communications
- Service publish a message and it consumes from several microservices
- Event bus interface
- OrderSubmitted Event
- Event-driven Architecture, CQRS pattern, event storing, eventual consistency principles

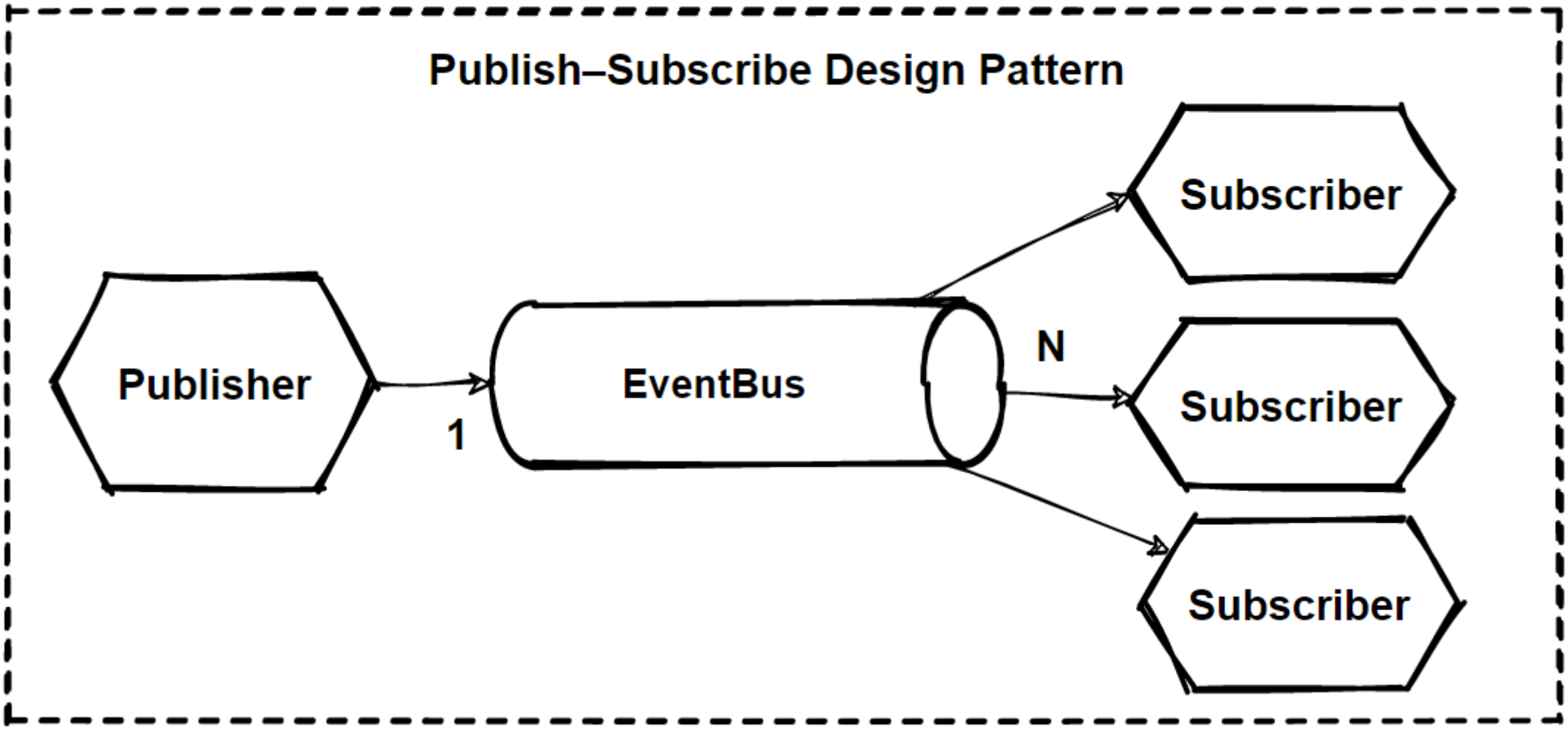


Dependency Inversion Principles (DIP)





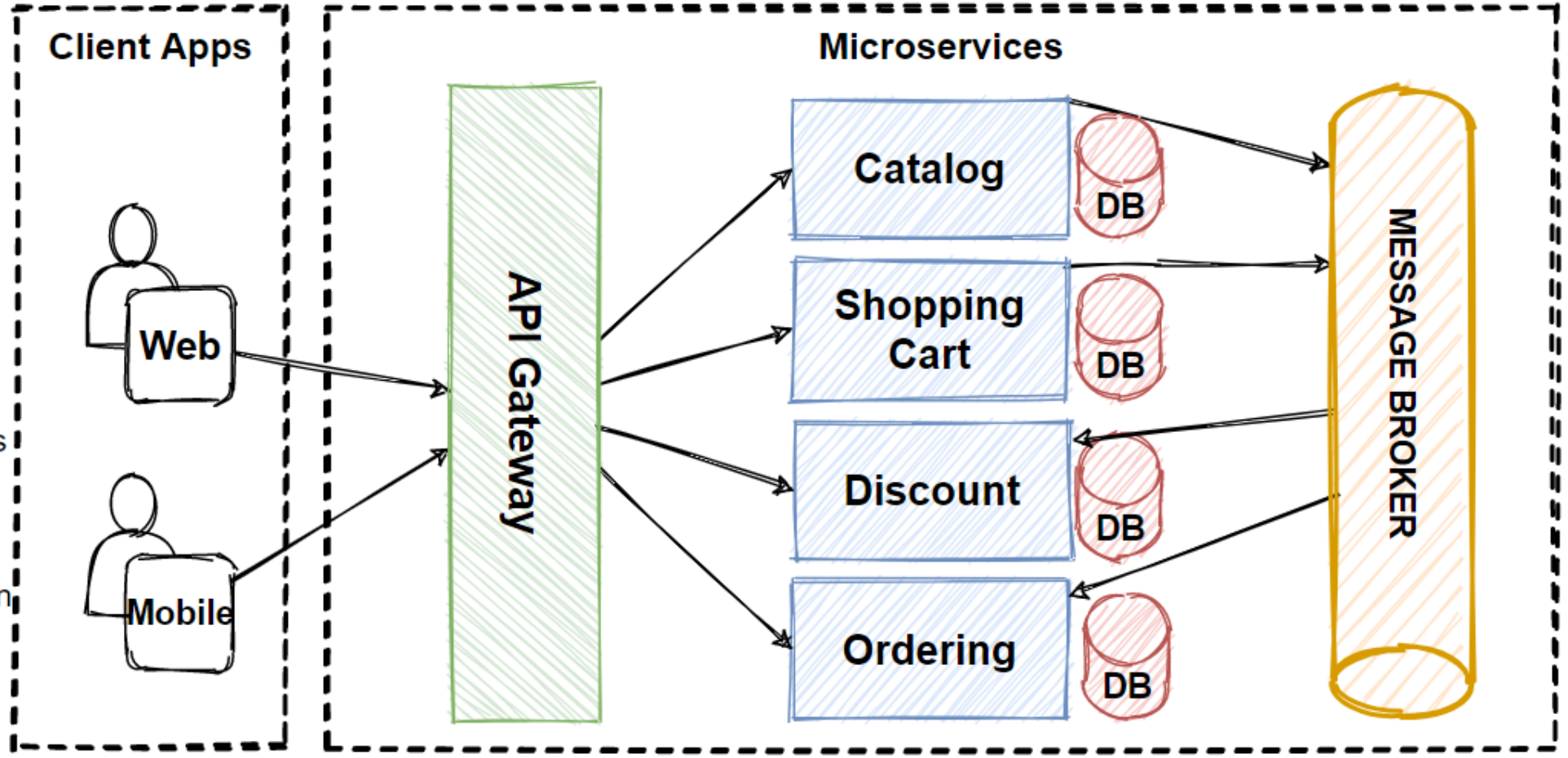

Service-to-Service Communications -
Chain Queries
Product
Billing
2
GetProduct
5
1
CreateOrder
4
Client
Gateway
ShoppingCart
Order
3
GetPrice
6
Pricing
Shipping


Publish–Subscribe Design Pattern
Publisher
1
EventBus
N
Subscriber
Subscriber
Subscriber

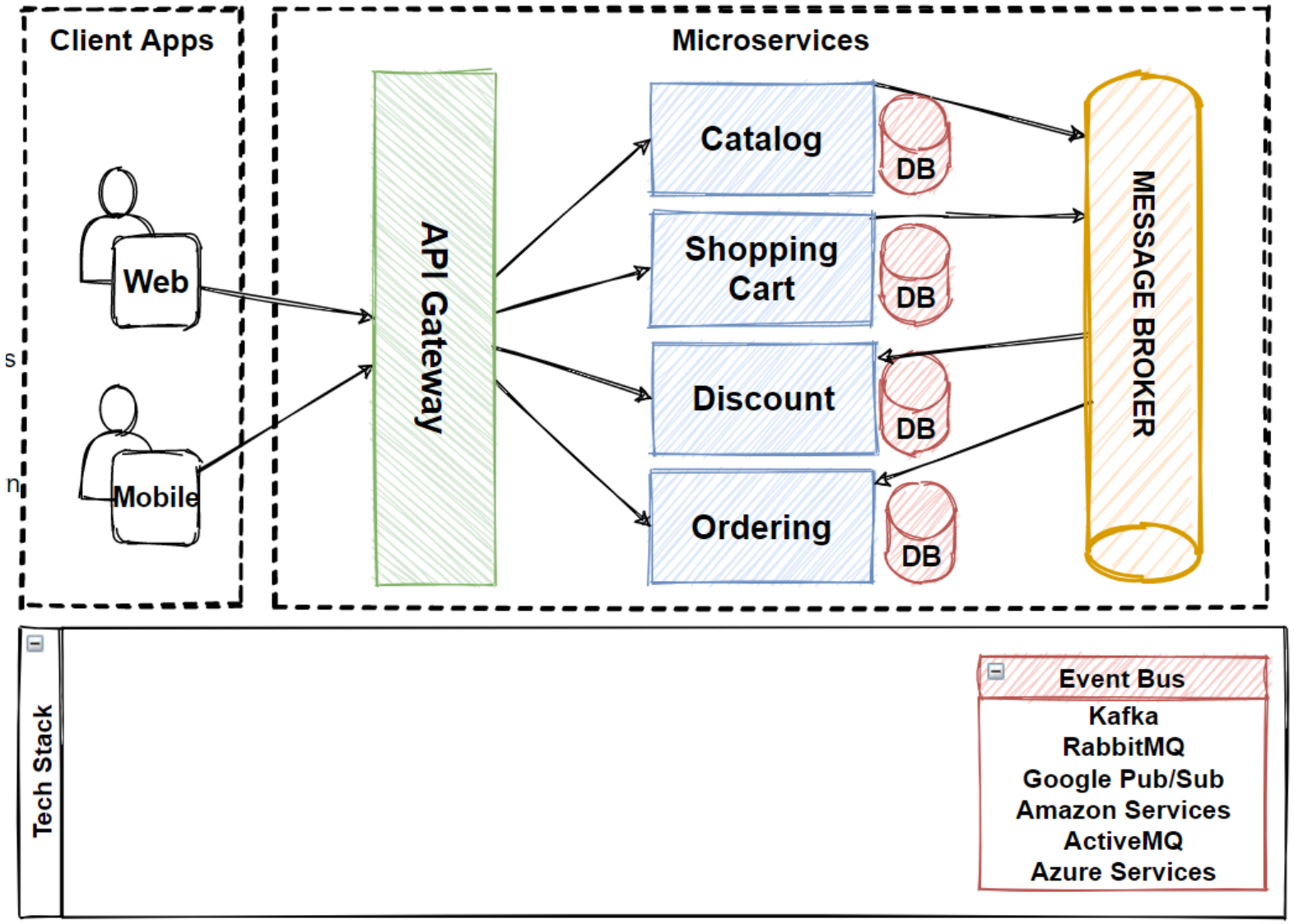
Microservices Architecture - Message Broker

Principles

- KISS
- YAGNI
- SoC
- SOLID
- Gateway Routing
- Gateway Aggregation
- Gateway Offloading
- Api Gw
- Database per Microservices
- Backends for Frontends pattern BFF
- Service Aggregator Pattern
- Service Registry Pattern
- Dependency Inversion Principles (DIP)
- Publish–Subscribe Design Pattern



Adapting Technology Stack



Section 11
Kafka and RabbitMQ Architecture

Asynchronous Message-Based Kafka Communication in Microservices Architecture

What is Apache Kafka ?

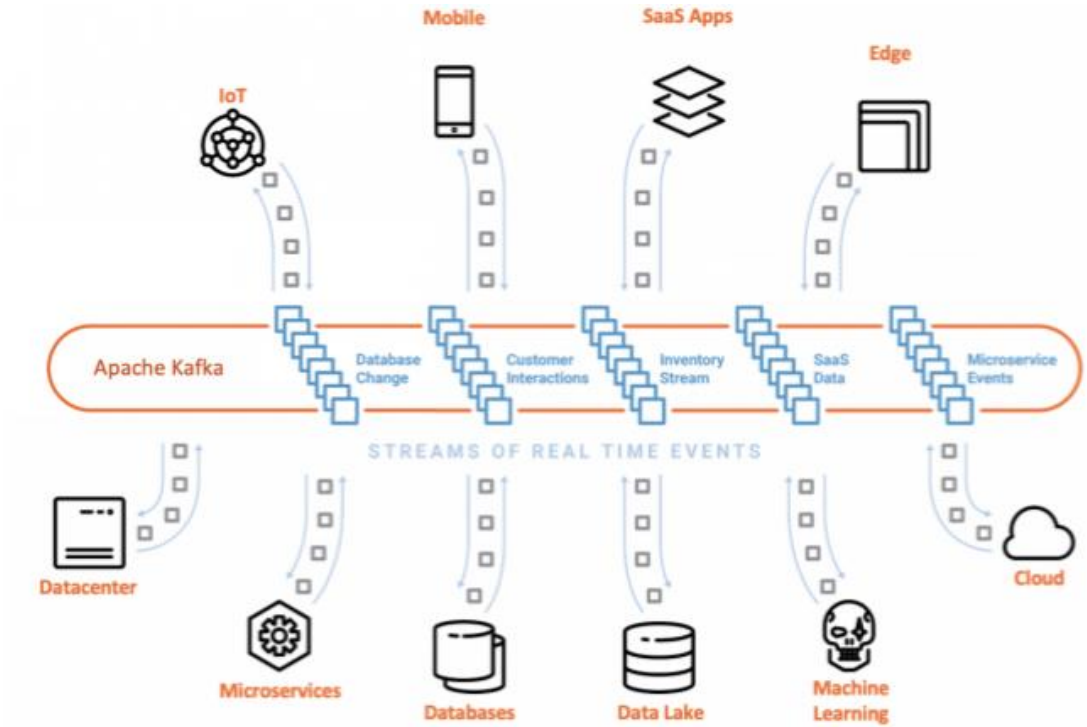
- Open-source event streaming platforms
- Horizontally scalable, distributed, and fault-tolerant
- Distributed publish-subscribe
- Event-driven Architecture
- Topic Messages
- ZooKeeper sync
- Topics, Partitions, Brokers, Producer, Consumer, Zookeeper

Apache Kafka Benefits and Use Cases

- Reliability, Scalability, Durability, Performance
- Built-in partitioning, replication and fault -tolerance

Use Cases

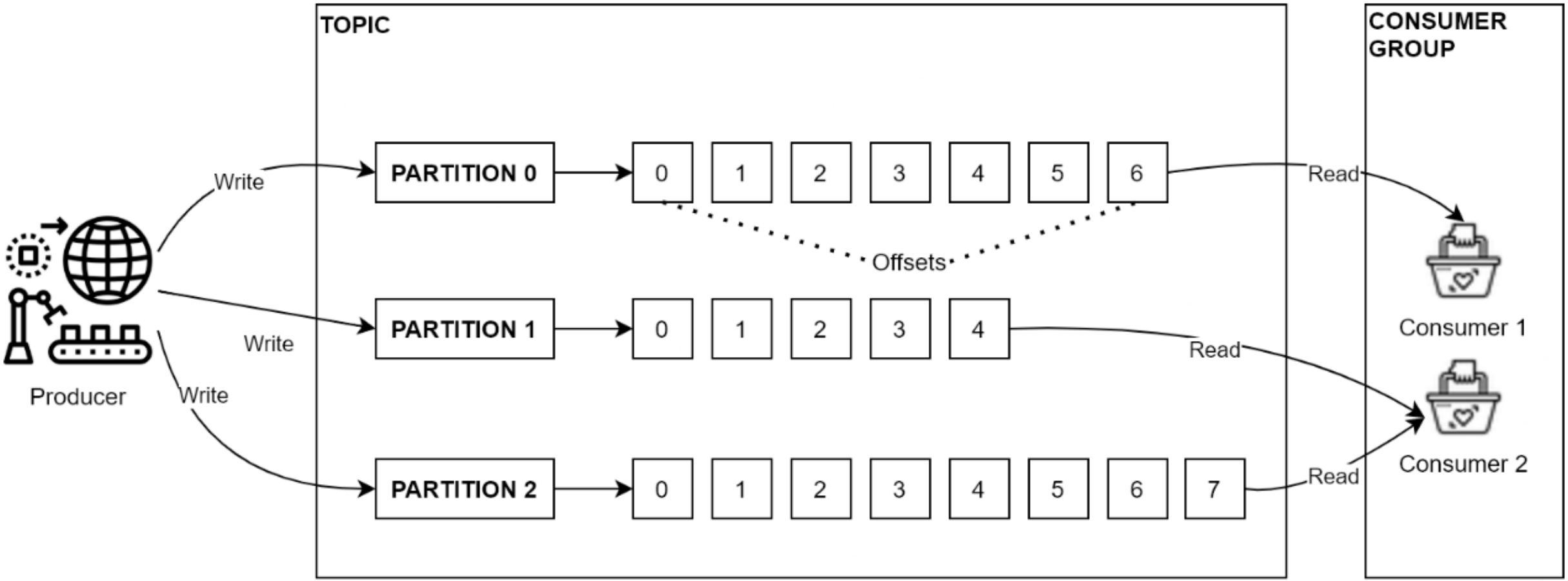
- Messaging
- Metrics
- Log Aggregation
- Stream Processing
- Activity Tracking
- Event Sourcing



- **Global-scale**
- **Real-time**
- **Persistent Storage**
- **Stream Processing**

APACHE kafka

Kafka Components - Topic, Partitions, Offset and Replication Factor



Apache Kafka Cluster Architecture

Kafka Brokers - Kafka Cluster

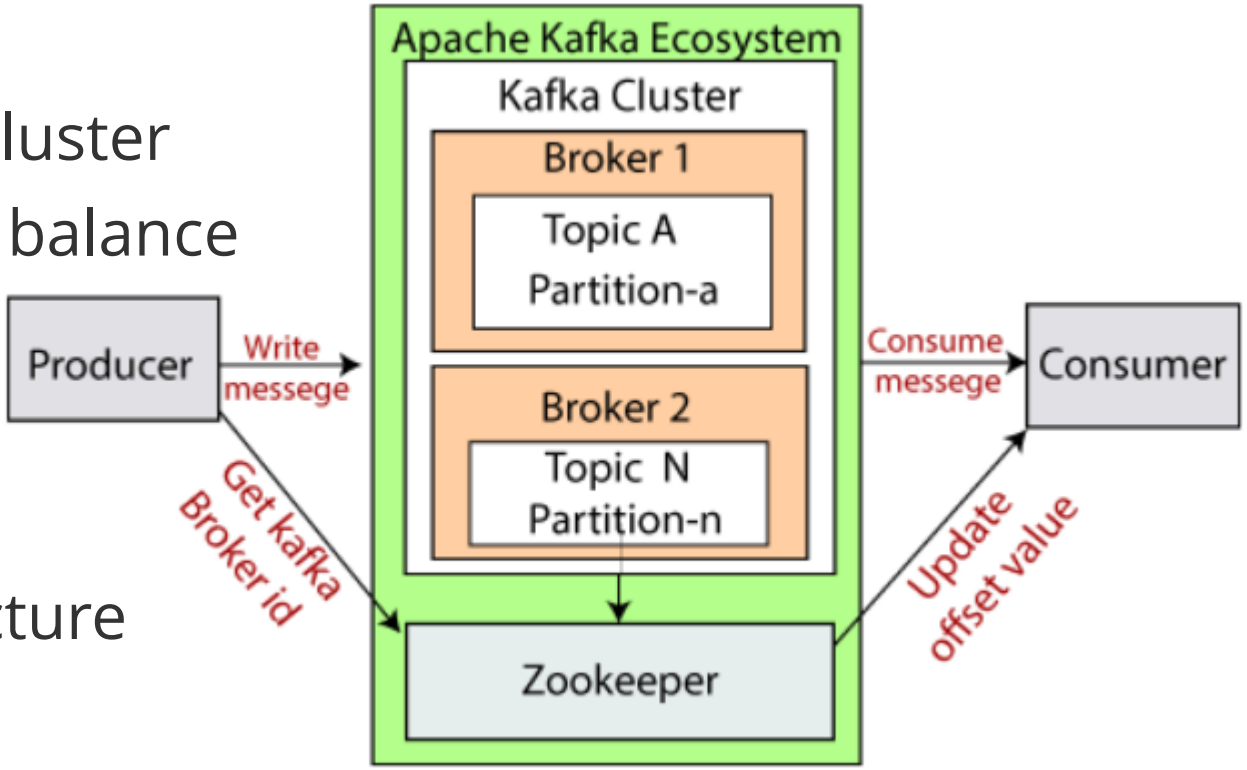
- Data replication, fault tolerance,

and high availability of your Kafka cluster

- Multiple brokers to maintain load balance

Zookeeper

- Manage and maintenances
- Coordinating Kafka broker
- Kafka Cluster master-less architecture
- Follows the brokers in the cluster
- Takes care about broker lifecycle



Apache Kafka Core APIs - Producer, Consumer, Streams and Connect API

Producers API

- Producers push data to brokers

Consumer API

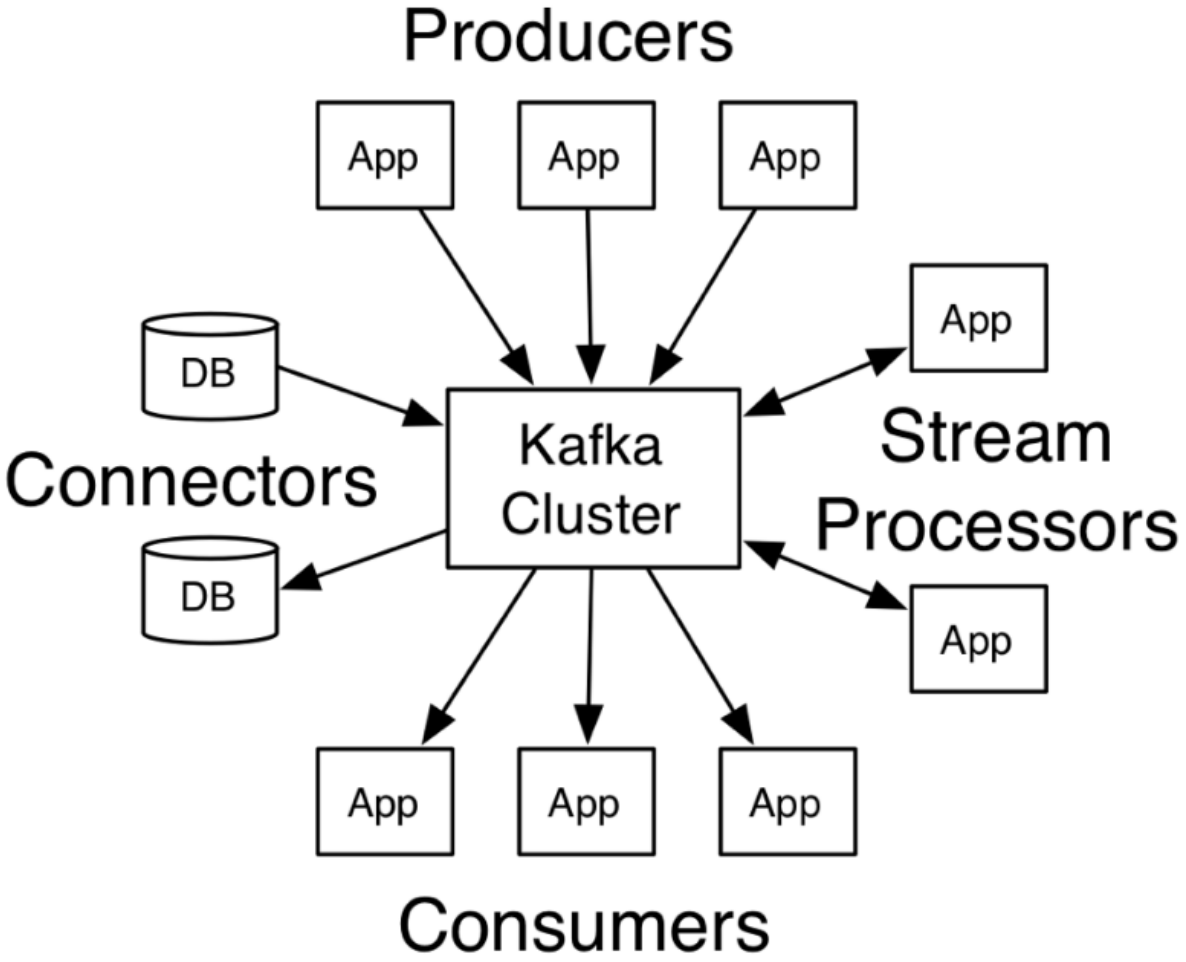
- Subscribe to topics and process the streams of data

Streams API

- Transforming the input streams to output streams

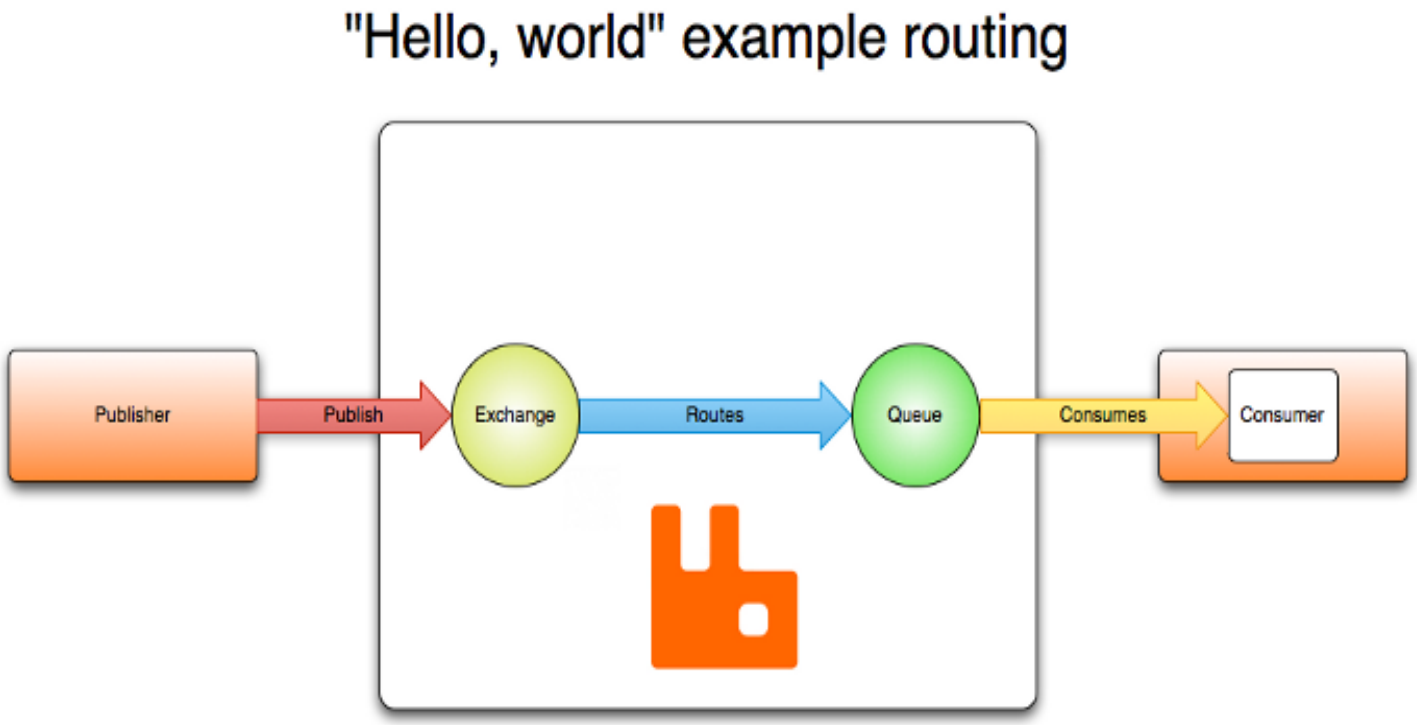
Connect API

- Configure connectors to move data into Kafka or from Kafka

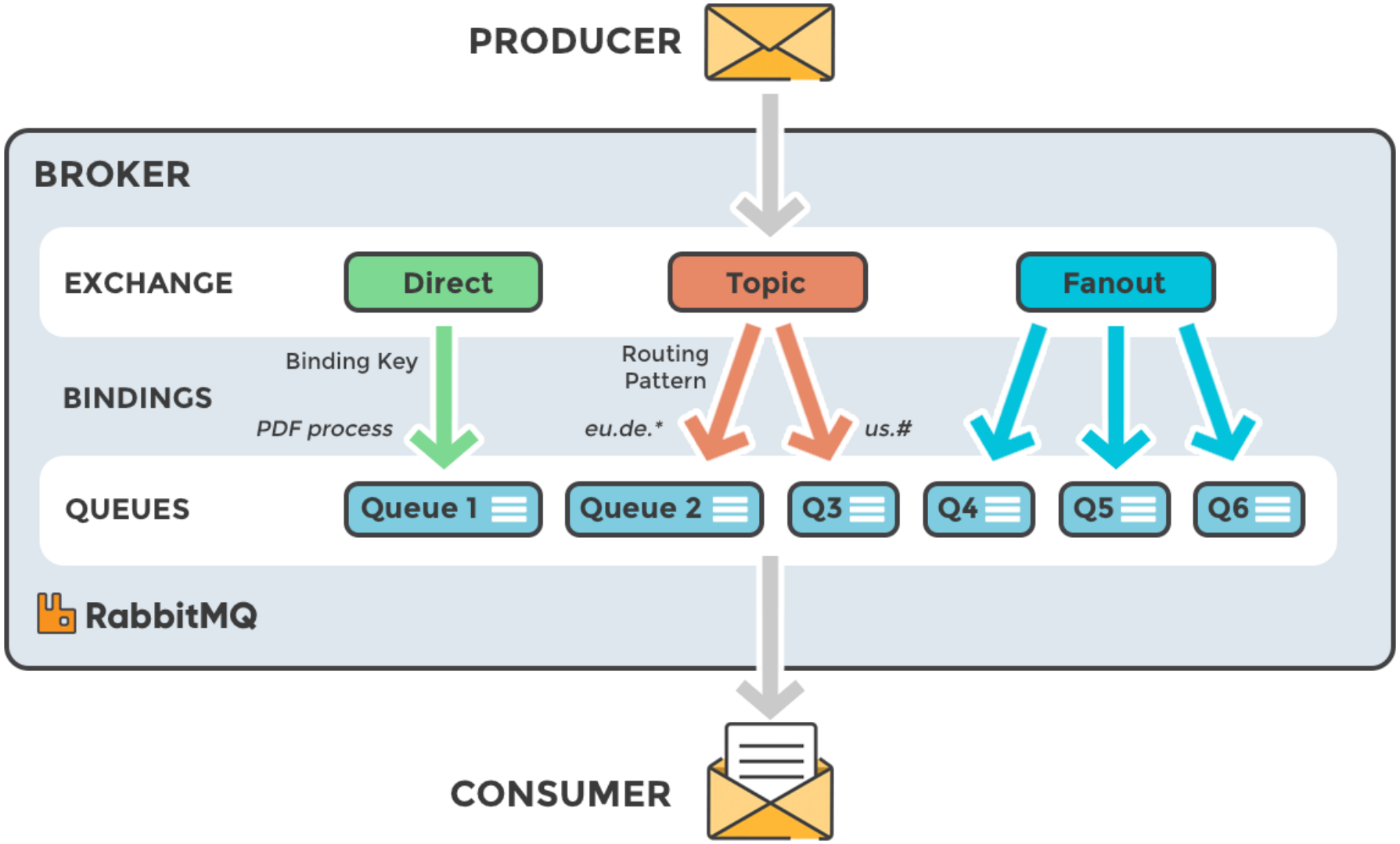


What is RabbitMQ ?

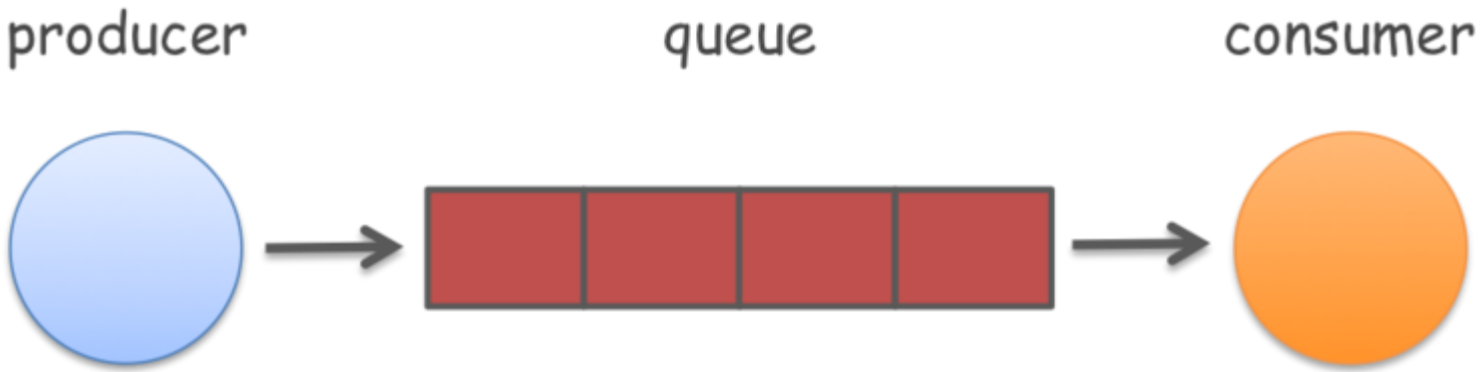
- Message Queue System
- Event-Driven Architecture
- Apache Kafka, Msmq, Microsoft Azure Service Bus, Kestrel, ActiveMQ



Main Components of RabbitMQ

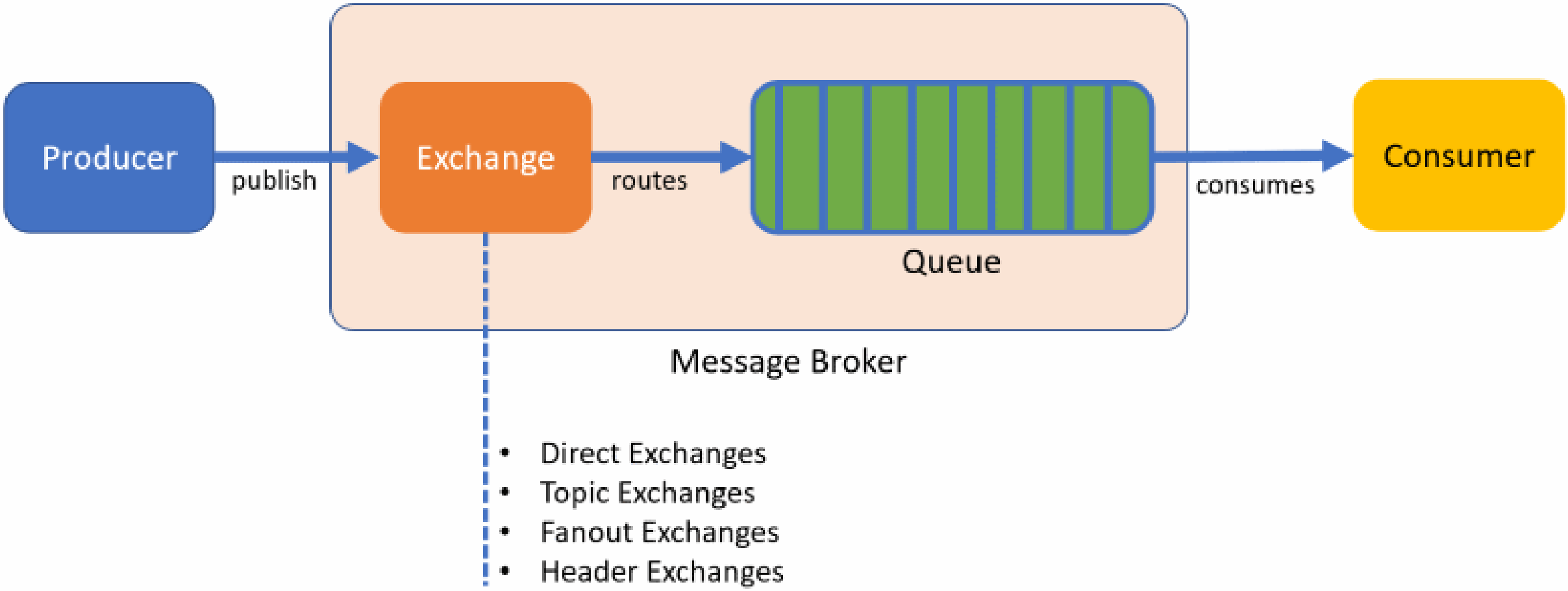


RabbitMQ Queue Properties

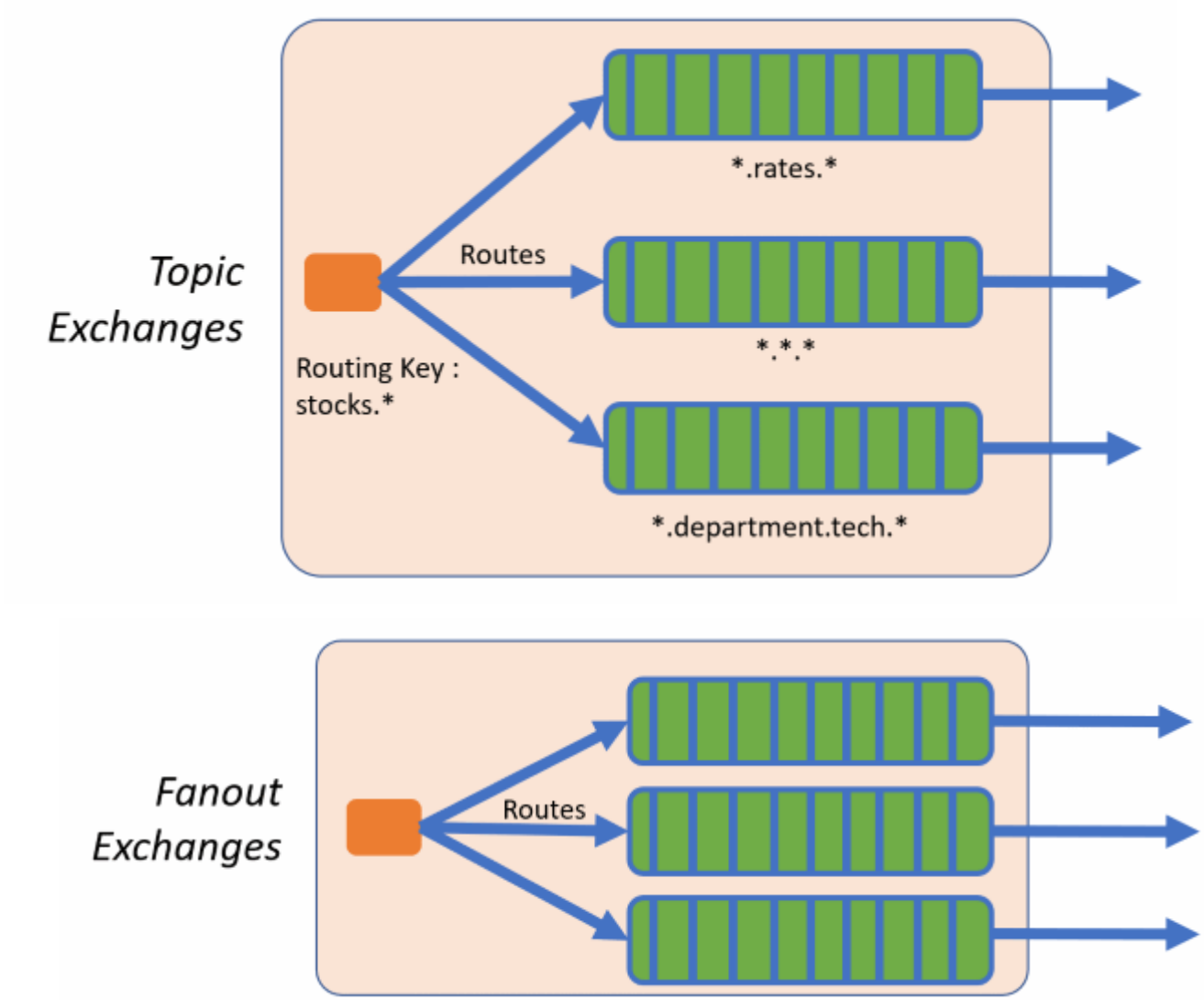


- Name
- Durable
- Exclusive
- AutoDelete

RabbitMQ Exchange Types

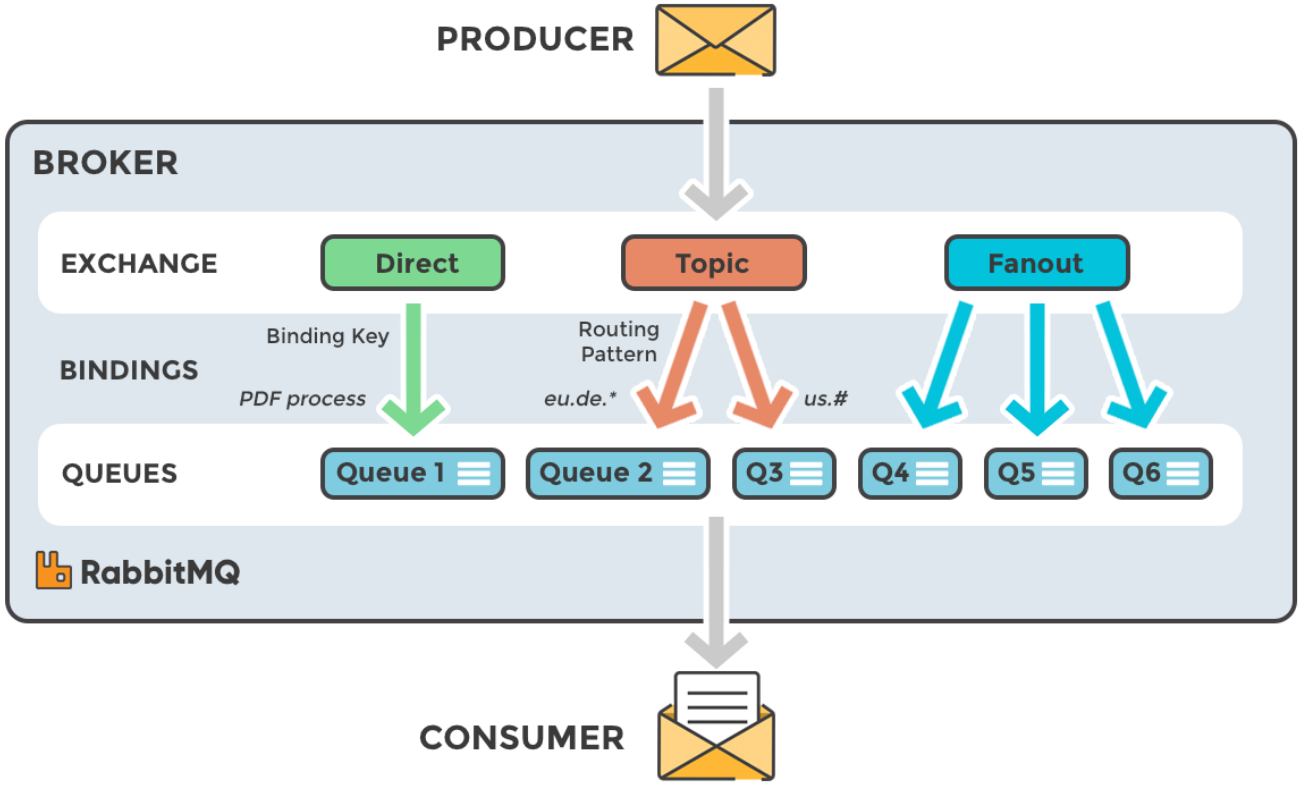


RabbitMQ Topic & Fanout Exchange Types



RabbitMQ Architecture

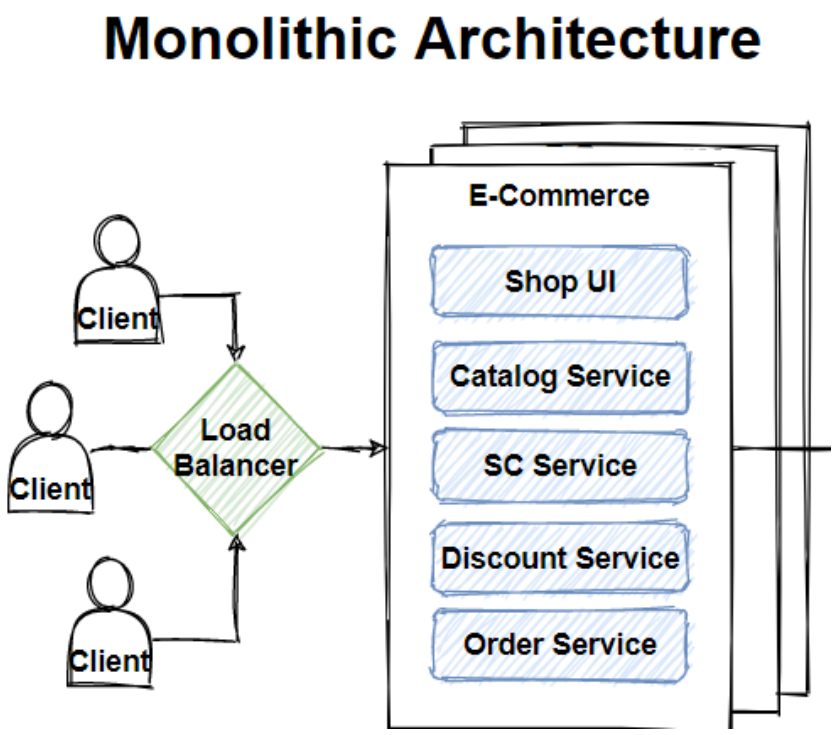
- Producer can only send messages to an exchange
- Exchanges control the routing of messages
- RabbitMQ uses a push model
- Distribute messages individually and quickly

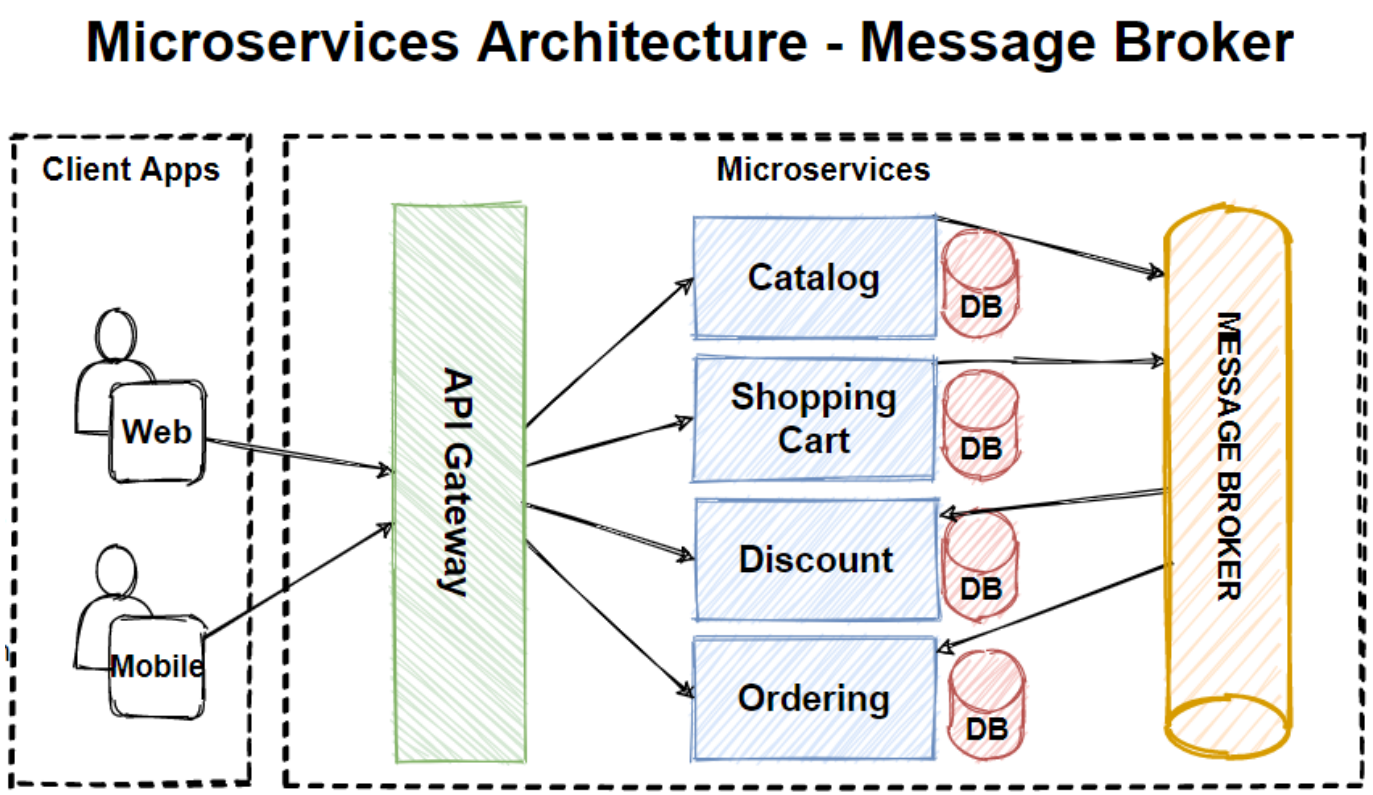


Section 12
Scale the Microservices Architecture Design

Stateless and Stateful Application Horizontal Scaling

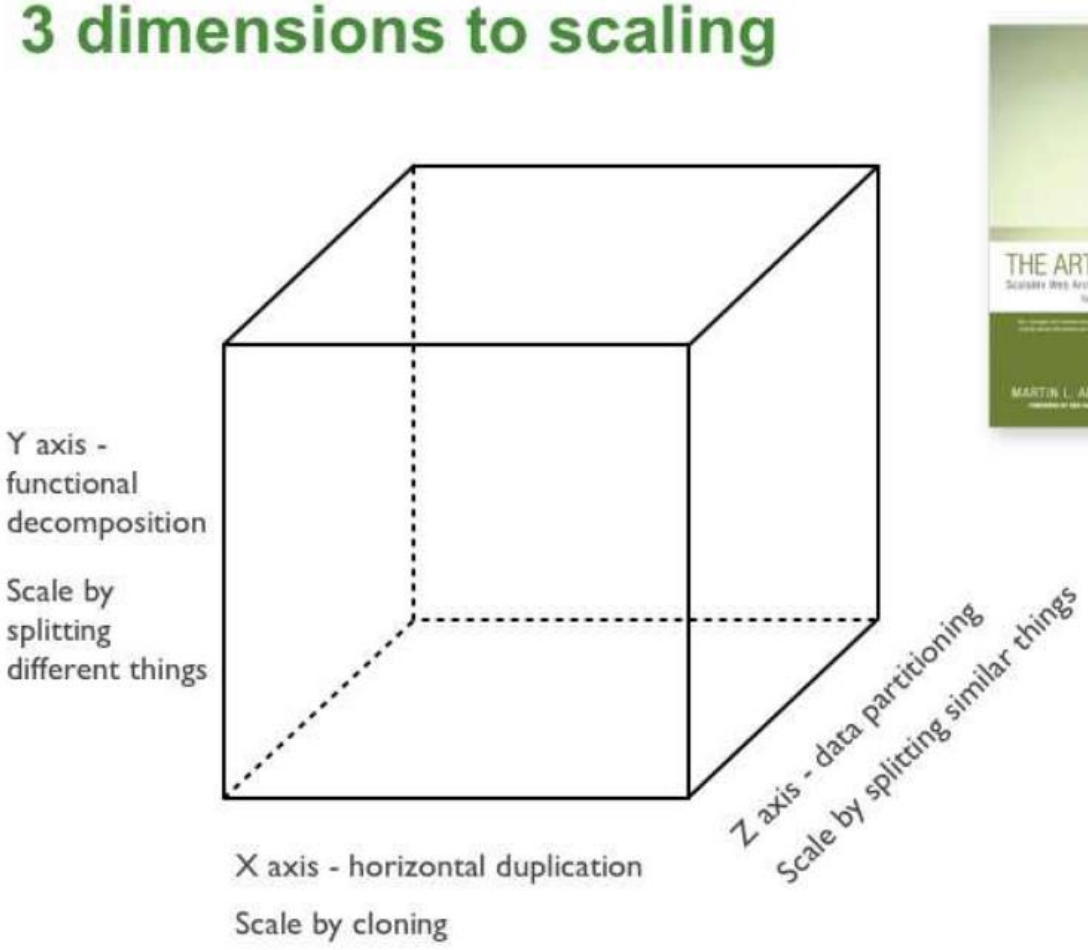
Scale the Microservices Architecture Design

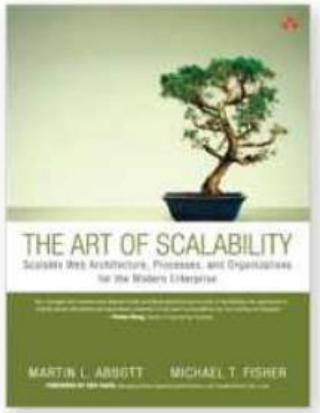




The Scale Cube

- X-Axis: Horizontal Duplication and Cloning of services and data
- Y-Axis: Functional Decomposition and Segmentation - Microservices
- Z-Axis: Service and Data Partitioning along Customer Boundaries - Shards/Pods





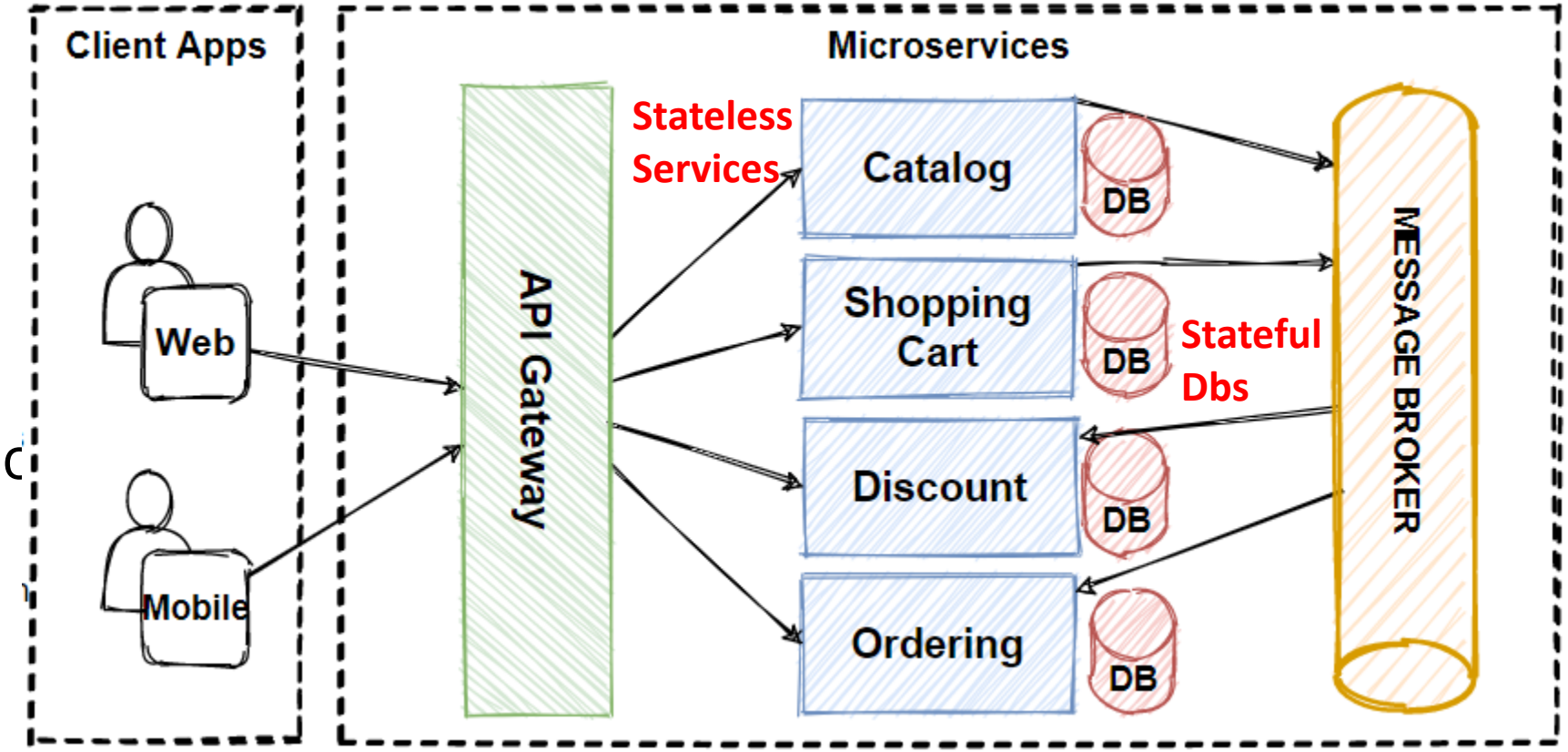
https://microservices.io/articles/scalecube.html

Stateless and Stateful Application Horizontal Scaling

- Scalability of our architecture to handle more number of concurrent requests
- Vertical scaling "scale up" is increase the hardware resources for exiting servers
- Horizontal Scaling "scaling out" is splitting the load between different servers
- **If the server have a state or not ?**

How to Scale Stateful Application Horizontal ?

- CAP Theorem
- Consistency level
- Strict consistency
- Eventual consistenc
- Spliting database servers
- Database Sharding

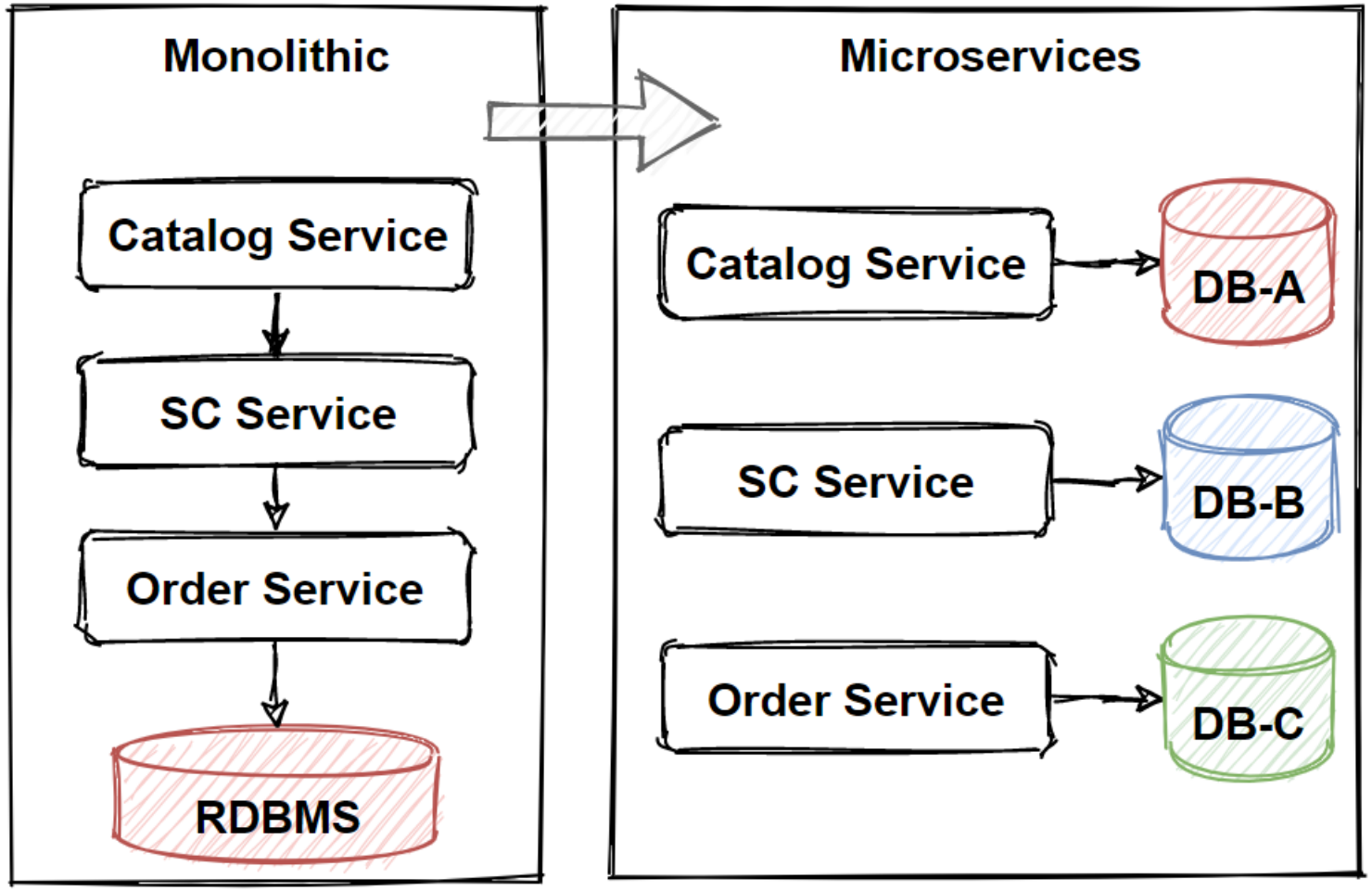


Section 13
Microservices Data Management

Microservices Data Management Pattern and Best Practices

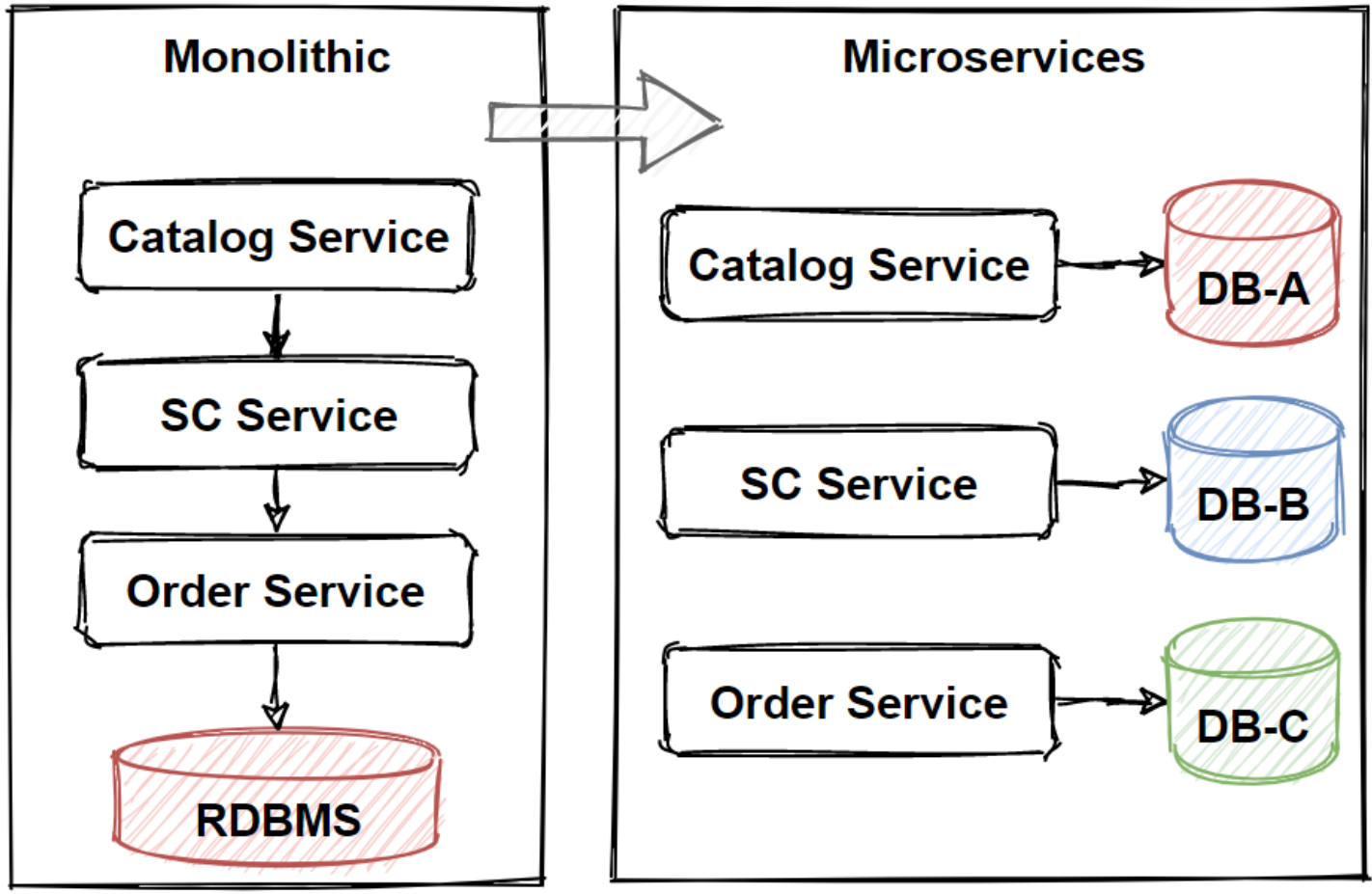
Polyglot Persistence Microservices

- Data integrity and data consistency
- Database-per-service its own data
- Isolating each service's databases
- Polyglot Persistence principle
- Duplicated or partitioned data challenge



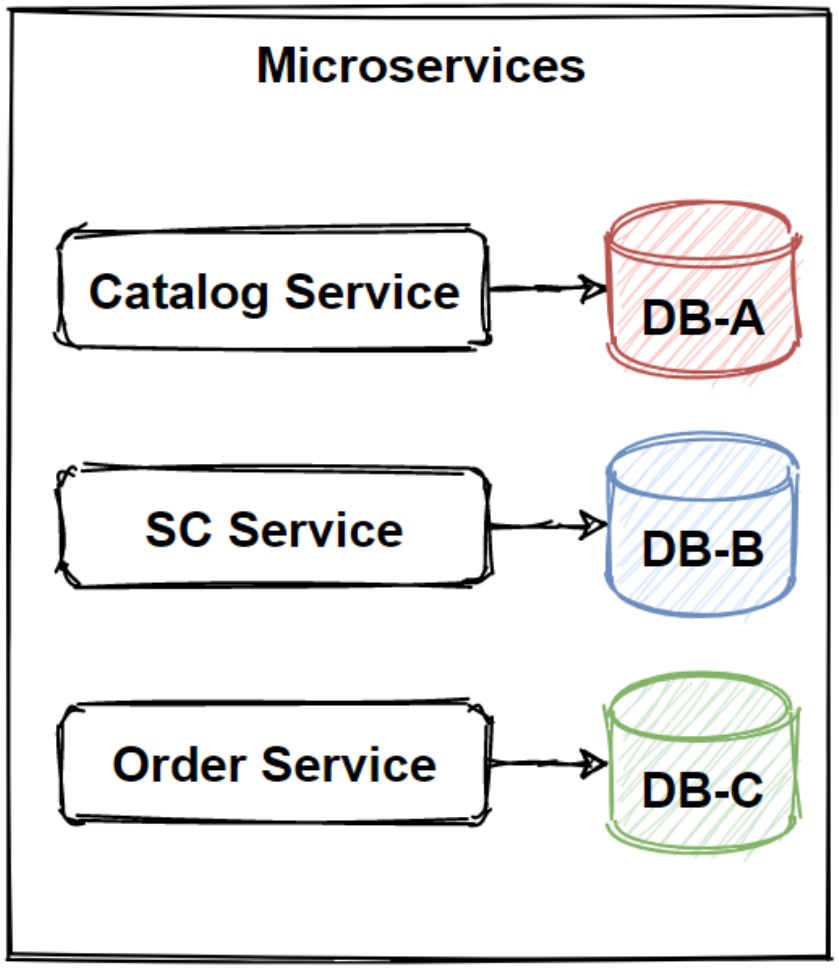
Polyglot Persistence Microservices 2

- Data consistency problems transaction management
- Welcome duplicate datas and eventual consistency
- Accept eventual consistency data
- Ability to scale independently
- Avoid single-point-of-failure o bottleneck databases
- Event driven architecture style



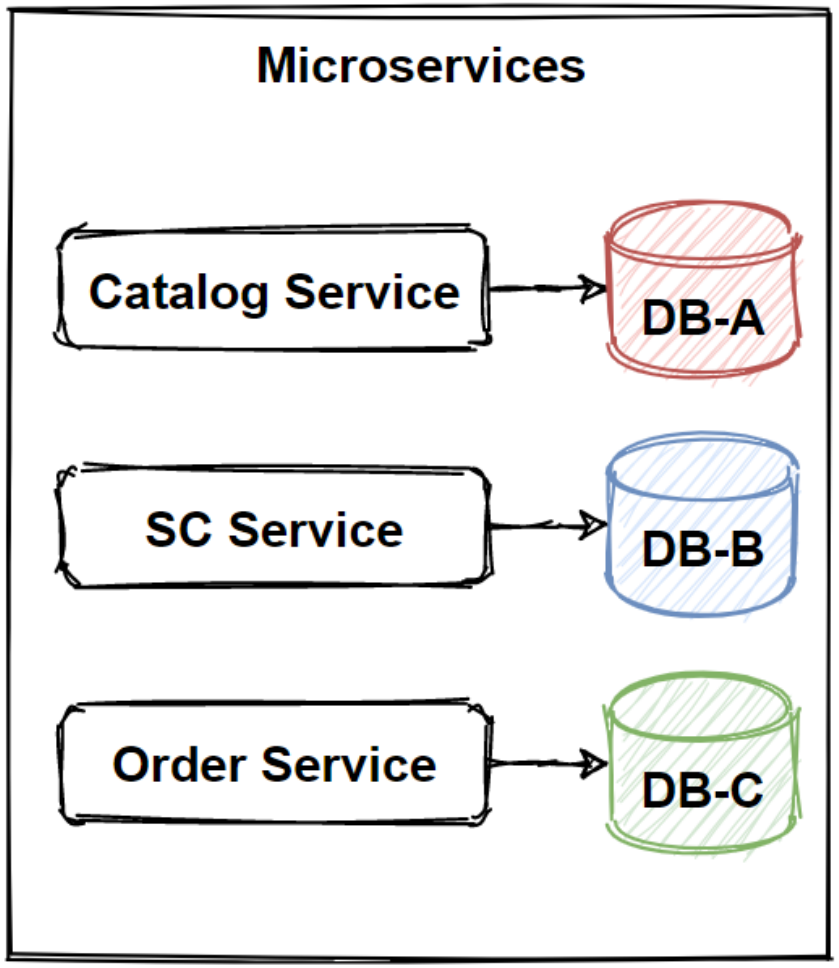
Microservices Database Management Patterns and Principles

- Should have a strategy
- The Database-per-Service pattern
- The API Composition pattern
- The CQRS pattern
- The Event Sourcing pattern
- The Saga pattern
- The Shared Database anti-pattern

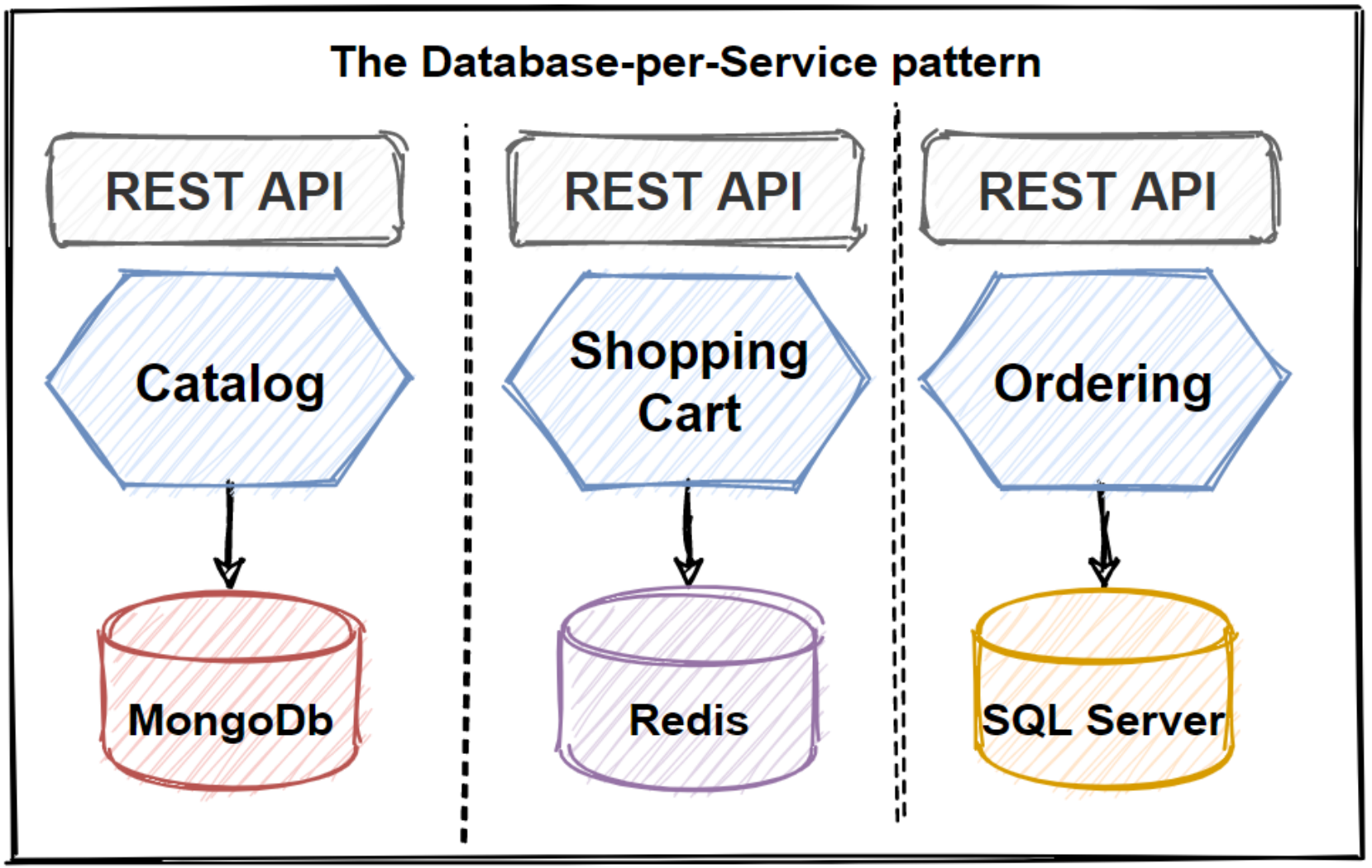


Microservices Database Management Patterns and Principles

- Should have a strategy
- The Database-per-Service pattern
- The API Composition pattern
- The CQRS pattern
- The Event Sourcing pattern
- The Saga pattern
- The Shared Database anti-pattern

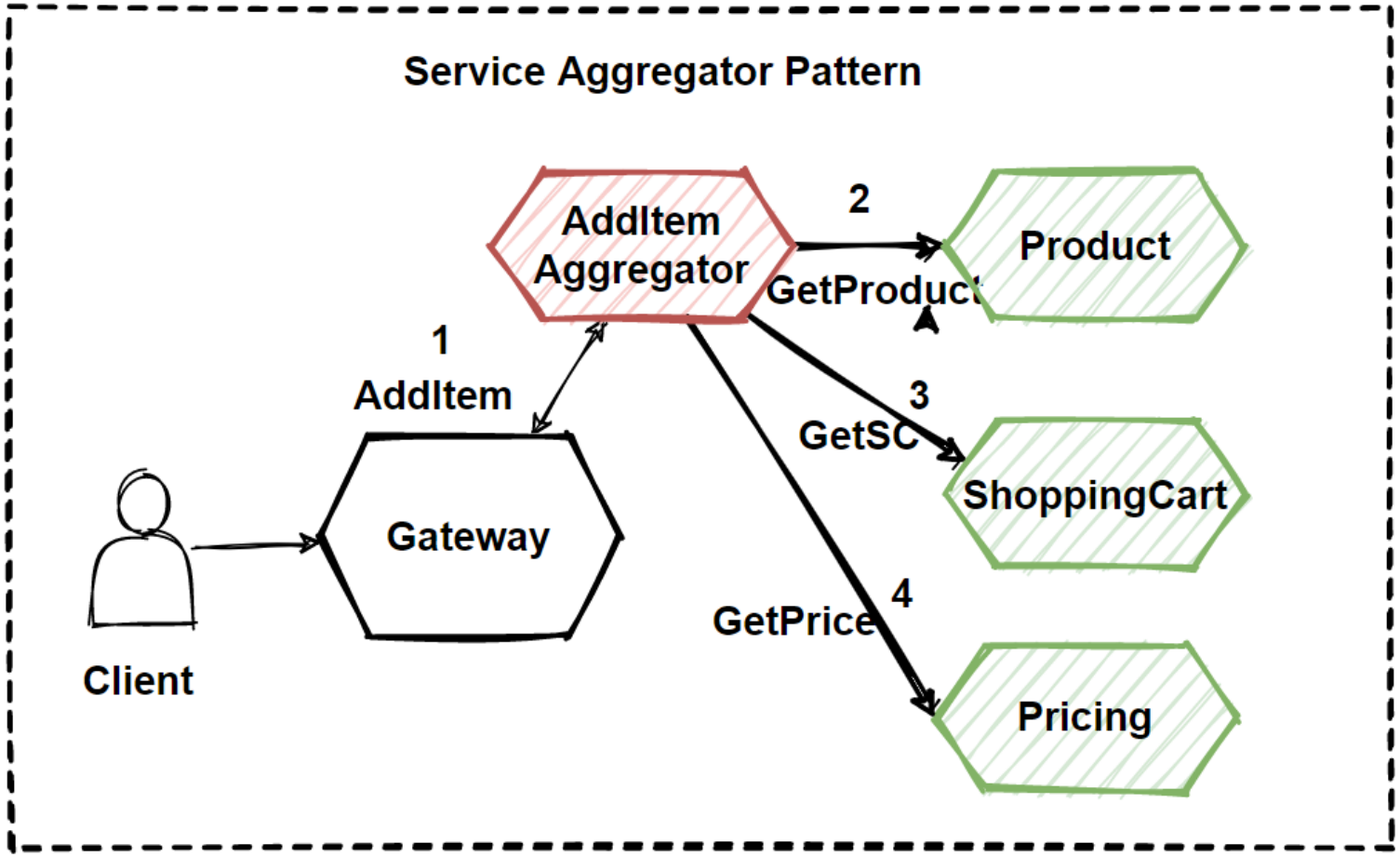


The Database-per-Service pattern



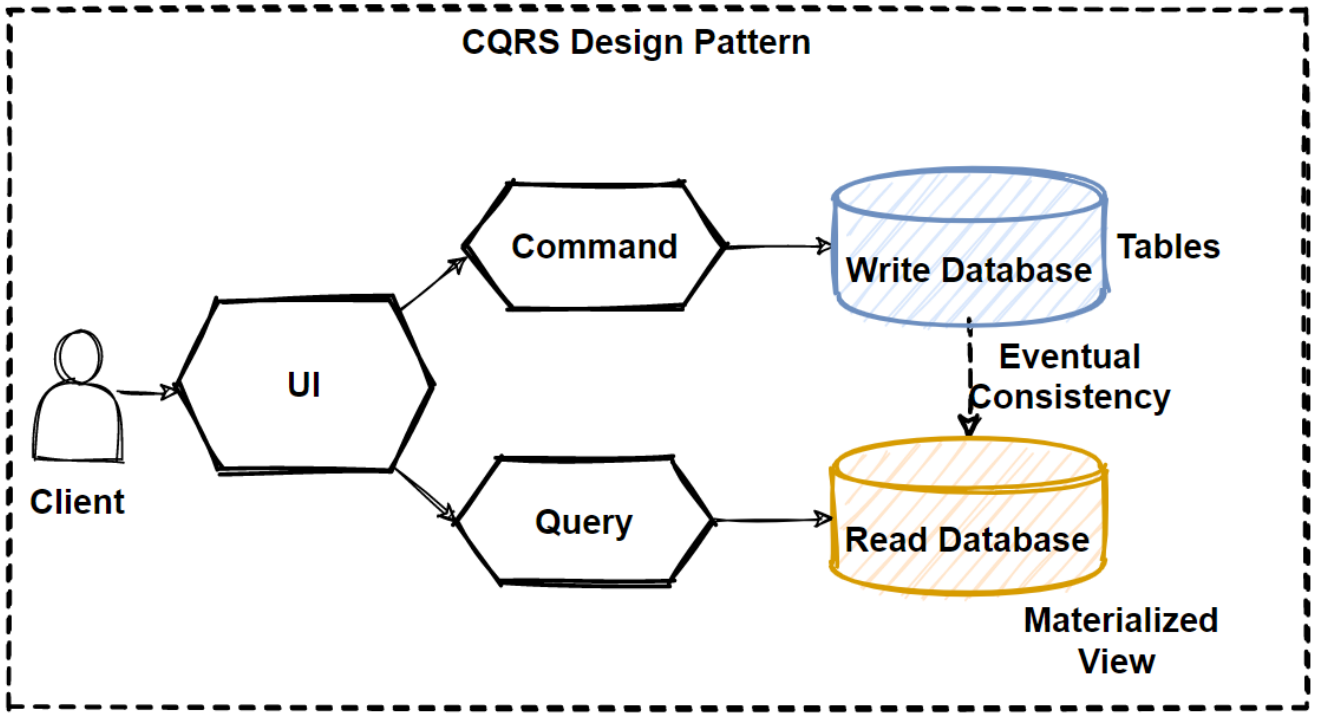
The API Composition patterns

- Dispatches requests of multiple internal backend microservices
- API Gateway Pattern
- Gateway Routing pattern
- Gateway Aggregation pattern
- Gateway Offloading pattern



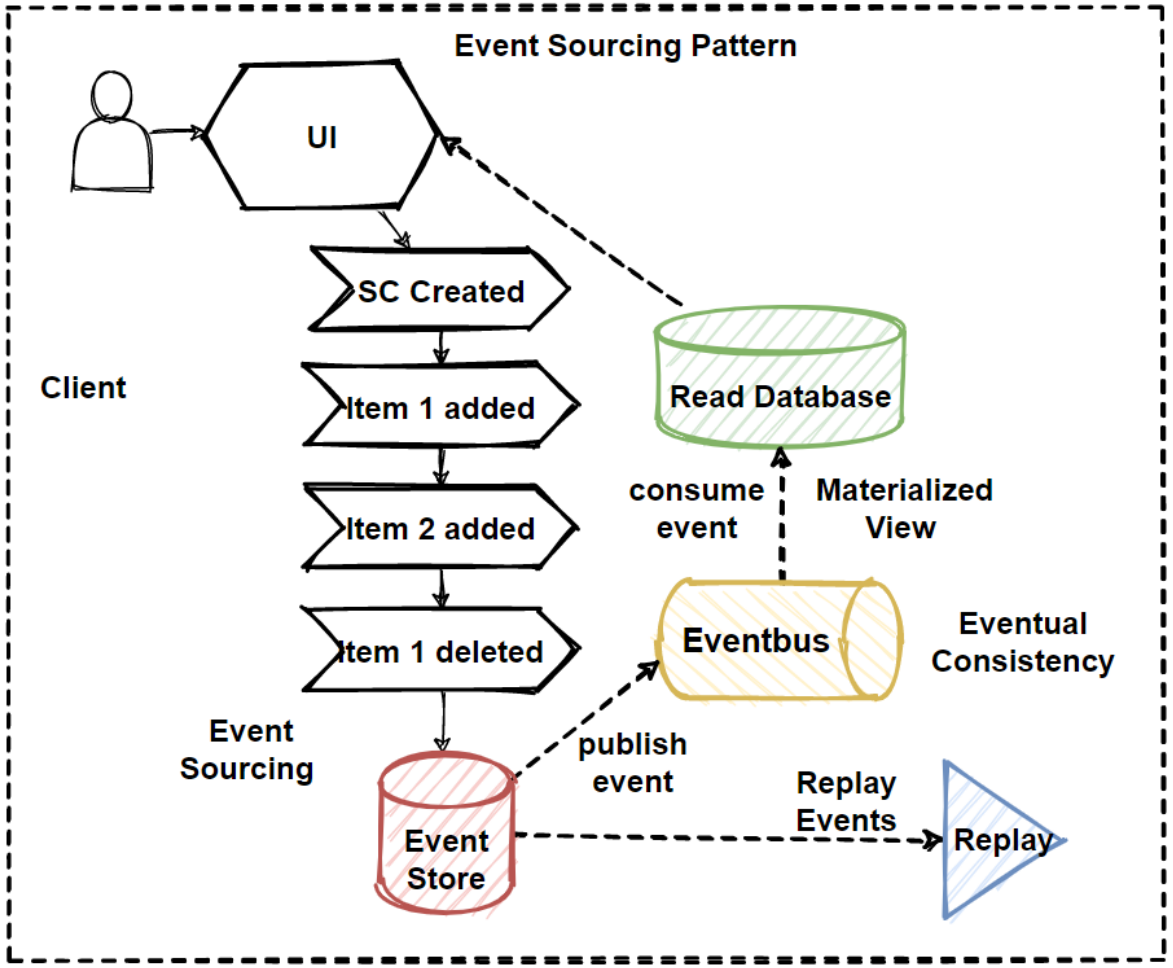
The CQRS pattern

- Command query responsibility segregation (CQRS)
- Separate commands and queries database
- Write-less, read-more approaches



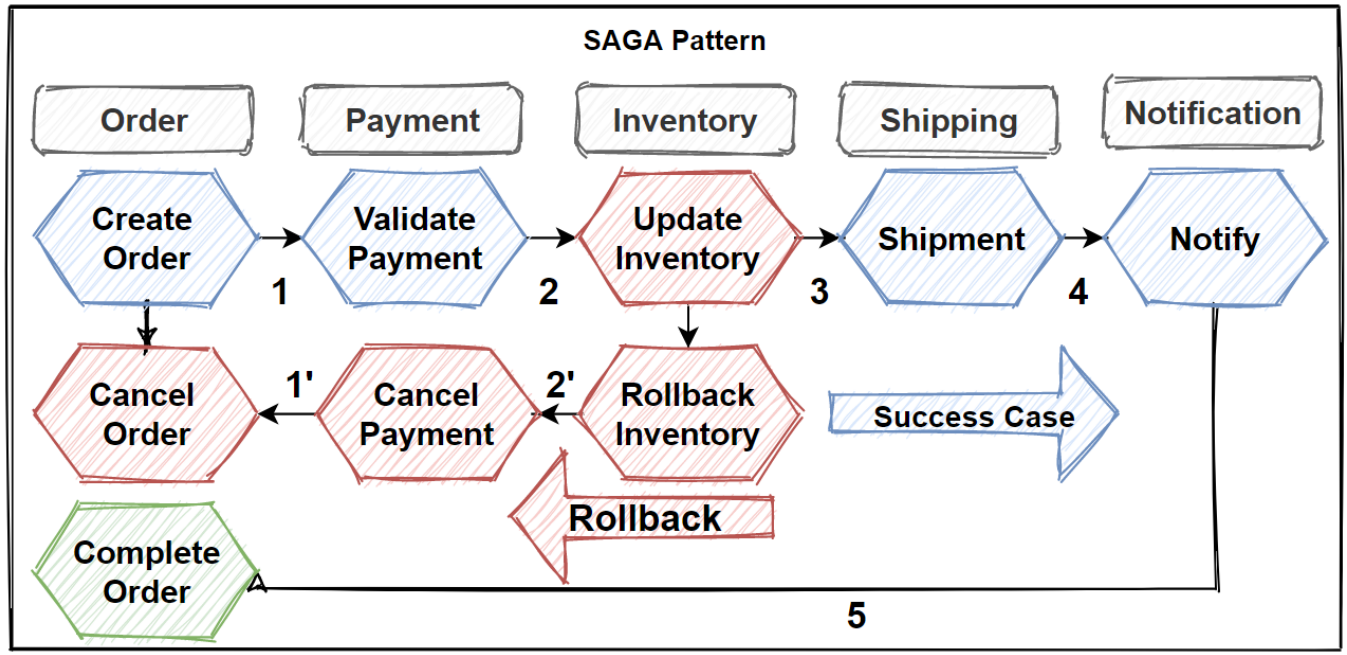
The Event Sourcing pattern

- Accumulate events
- Aggregates them into sequence of events
- Replay at certain point of events



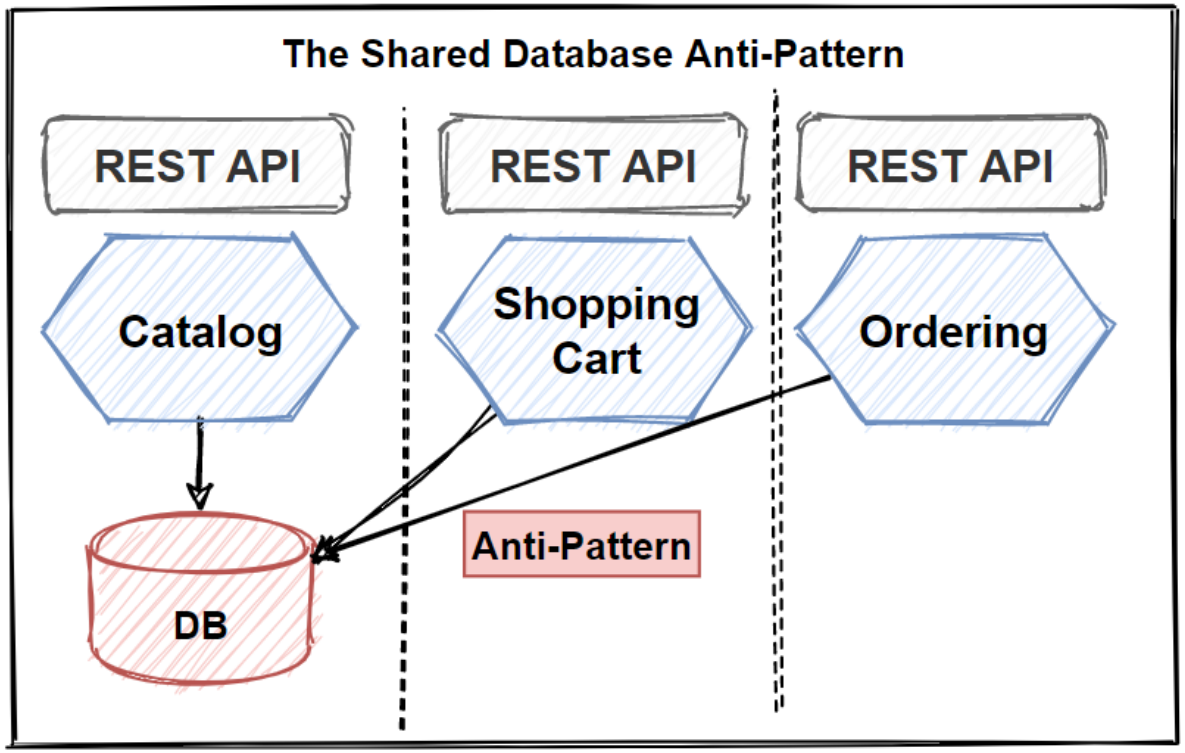
The Saga pattern

- Provide Transaction management
- Maintaining data consistency
- 2 types of SAGA
- Choreography – when exchanging events happens without points of control
- Orchestration – when you have centralized controllers



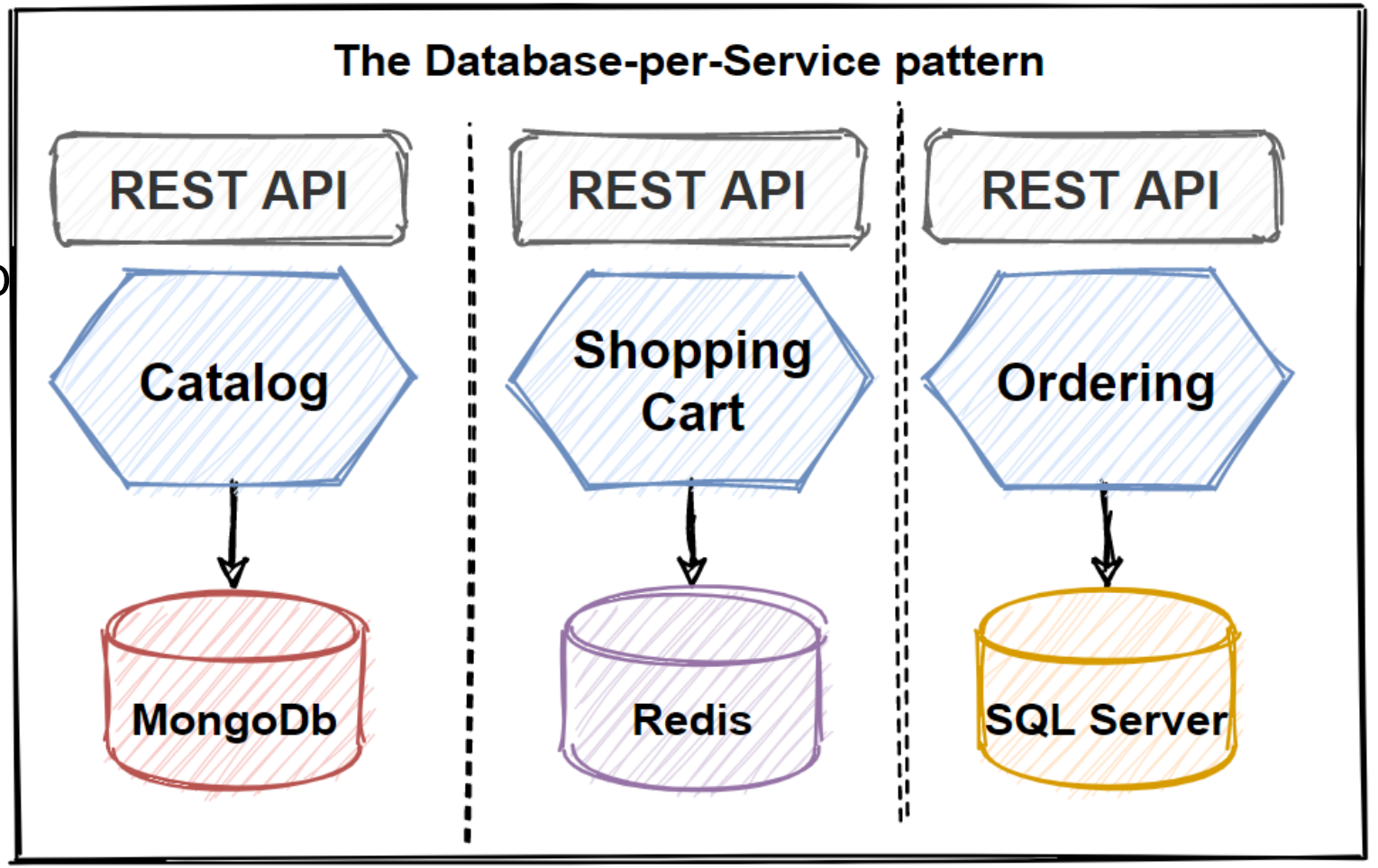
The Shared Database anti-pattern

- Shared Database
- Anti-pattern
- Single shared database with each service accessing data
- Against to microservices nature
- Big a few Monolithic applications

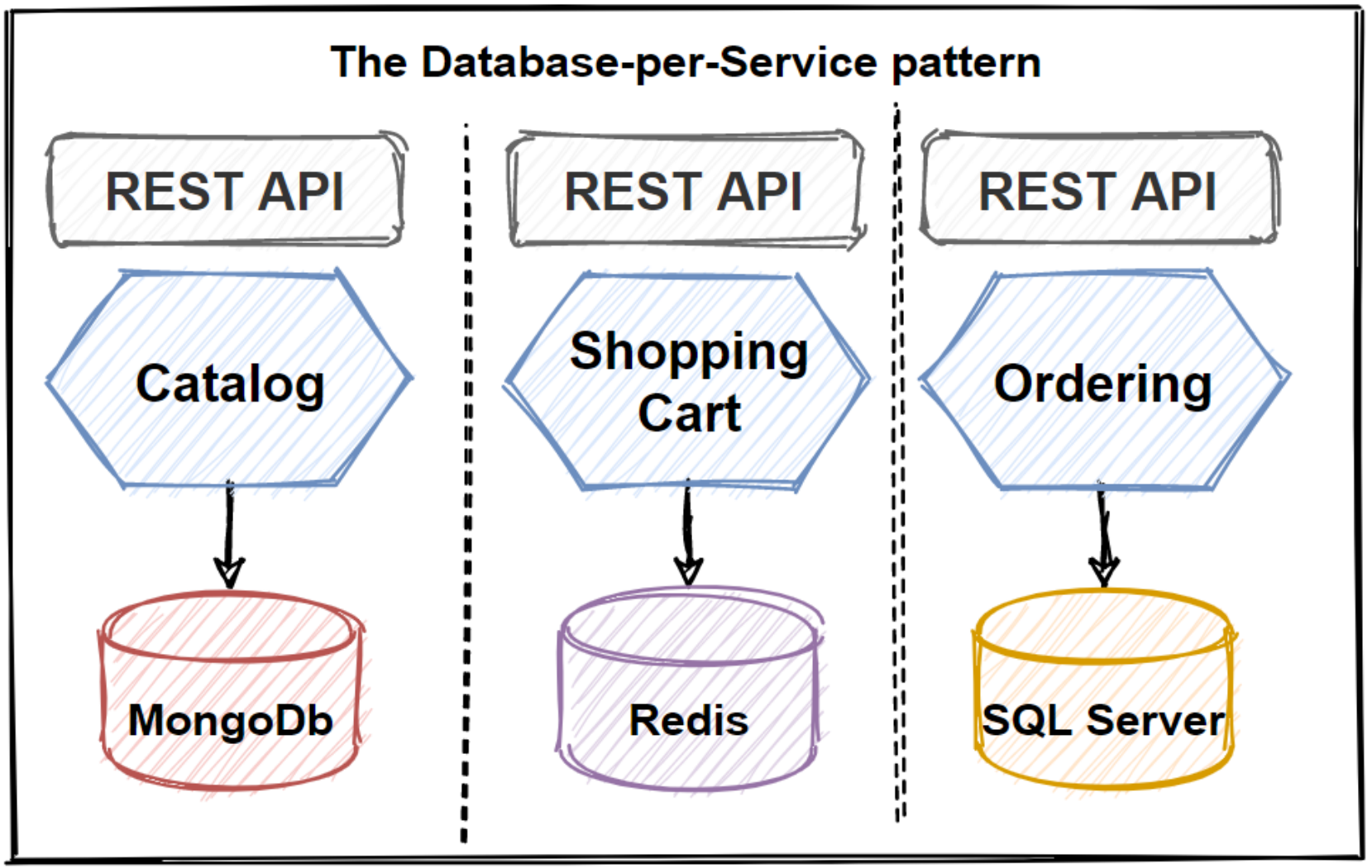


The Database-per-Service pattern

- Shifting to the monolithic architecture to microservices architecture
- Decomposes database into a distributed data model
- Evolve rapidly and easy to scale applications
- Schema changes can perform without any impact
- Scaling independently

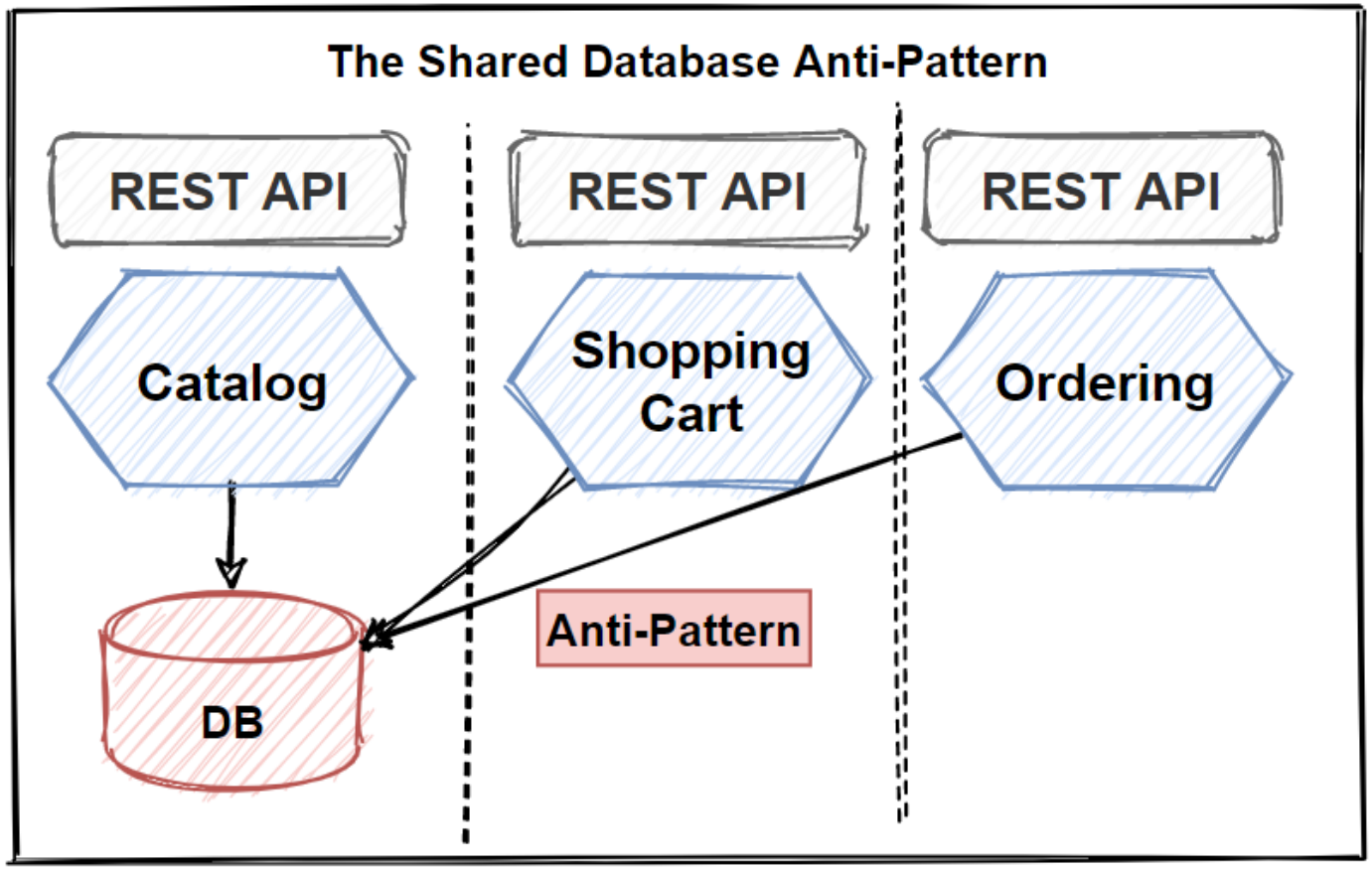


The Database-per-Service pattern 2



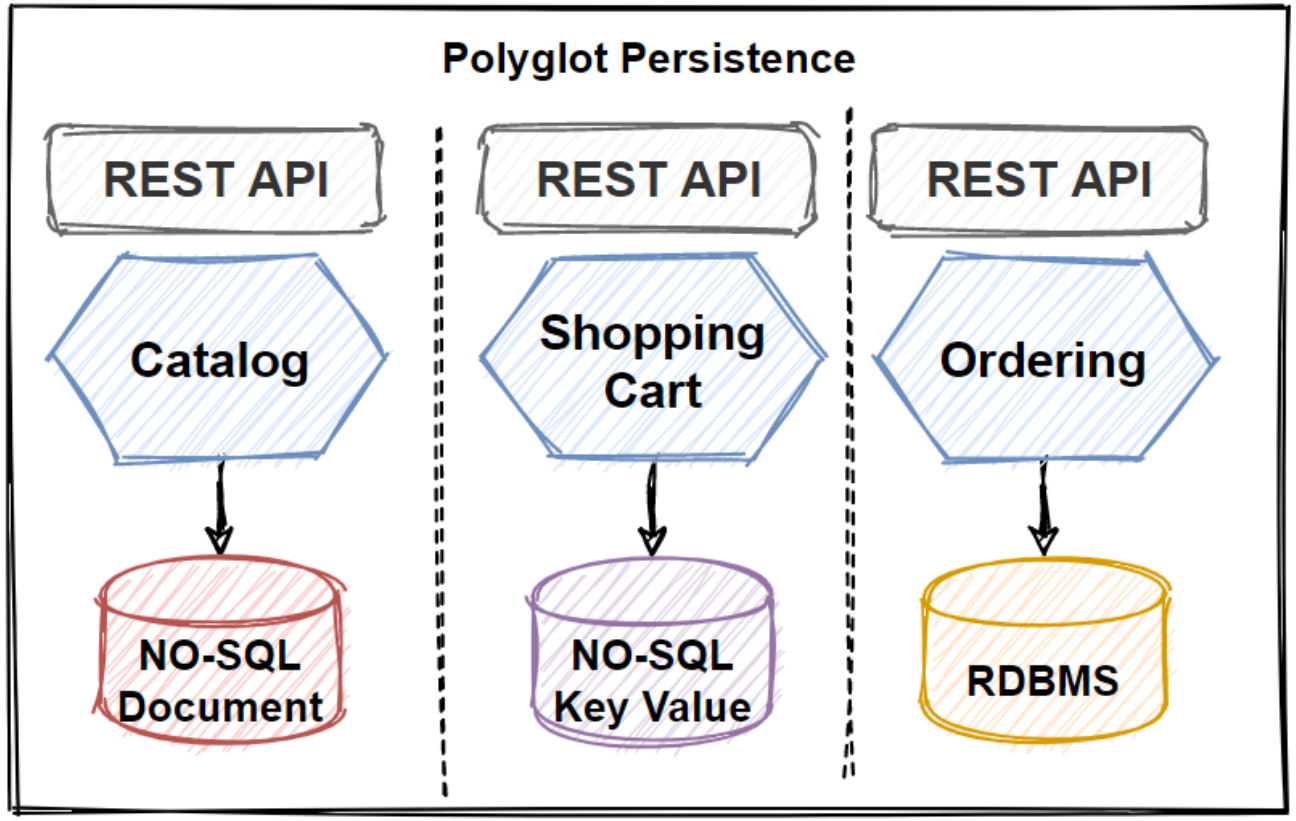
The Shared Database Anti-Pattern

- Shared Database
- Anti-pattern
- Single shared database with each service accessing data
- Against to microservices nature
- Block microservices due to single-point-of-failure



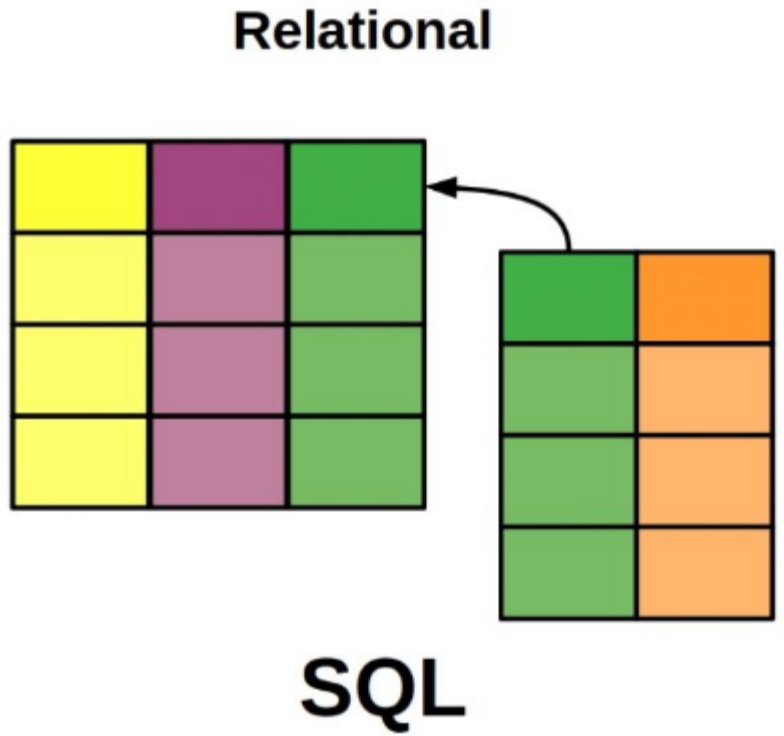
Polyglot Persistence

- Different kinds of data storing technologies
- Come with a cost
- Looked up page elements by ID inappropriately
- Better suited to a key-value no-sql databases
- **How to Choose a Database for Microservices ?**



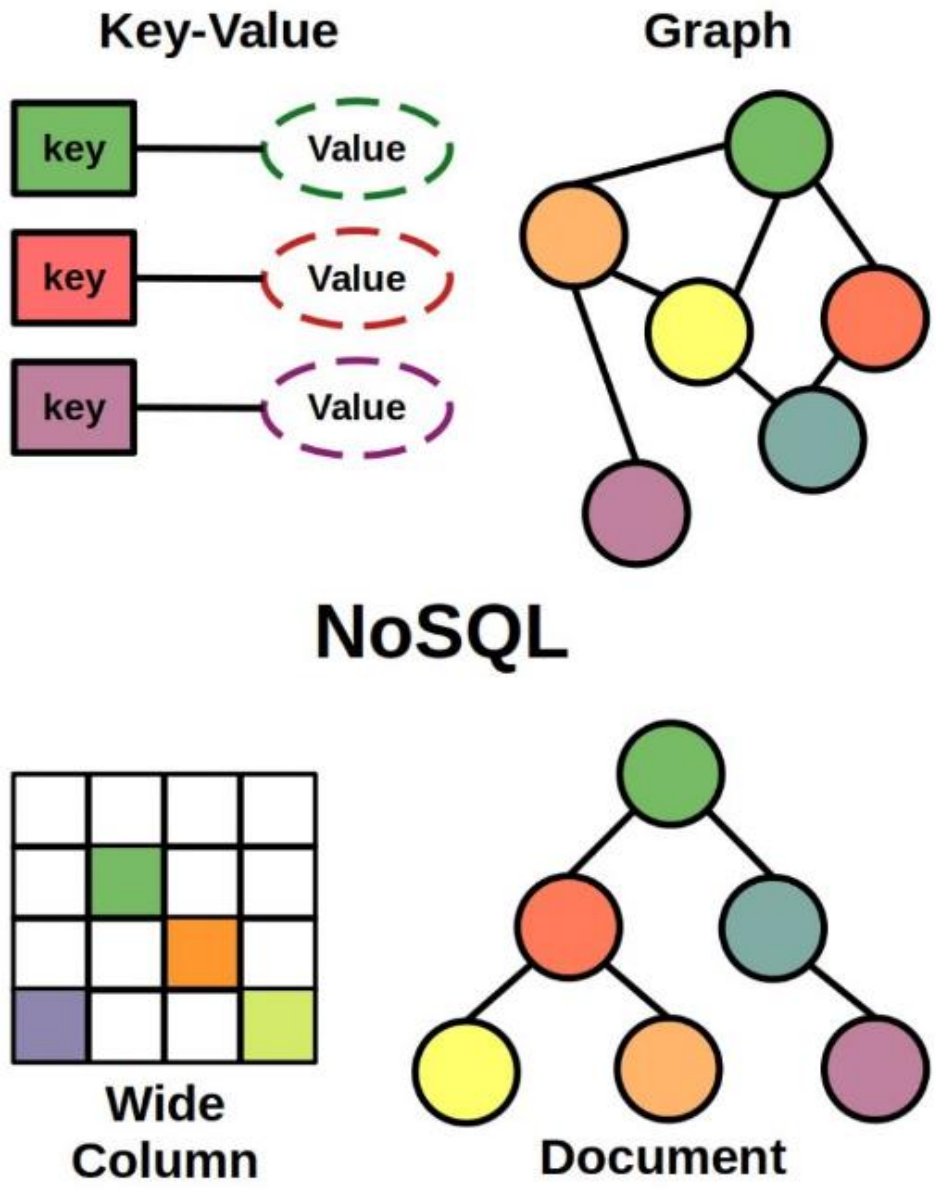
Relational Databases

- Storing data into related data tables
- Fixed schema and use SQL to manage data
- Support transactions with ACID
- Polyglot persistence in microservices
- Oracle, MS SQL Server, MySQL, PostgreSQL



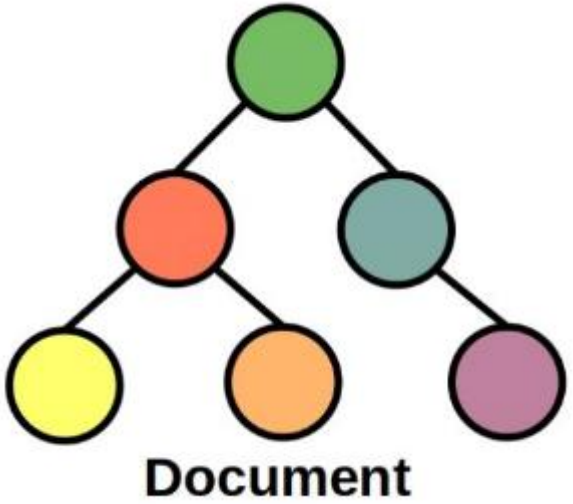
No-SQL Databases

- Different types of stored data
- Ease-of-use, scalability, resilience, and availability
- Stores unstructured data in key-value pairs or JSON documents
- Don't provide ACID guarantees



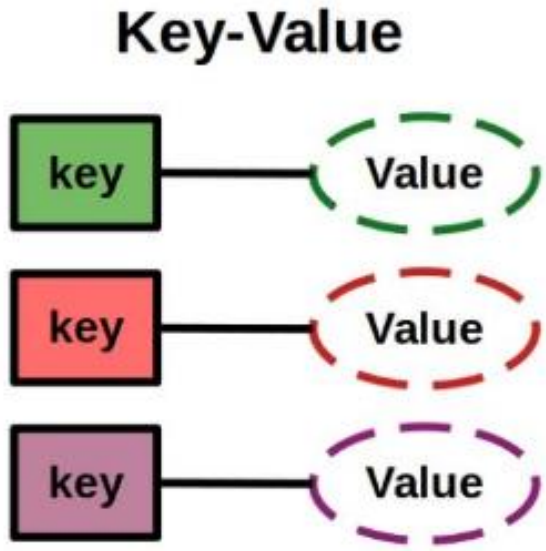
No-SQL Document Databases

- Store and query data in JSON-based documents
- Data and metadata are stored hierarchically
- Objects are mapping to the application code
- Scalability, document databases can distributed very well
- Content management and storing catalogs, MongoDB and Cloudant



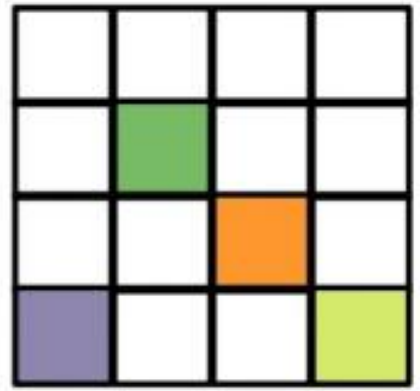
No-SQL Key-Value Databases

- Data is storing as a collection of key-value pairs
- Group of key-value in the database
- Session-oriented applications for example storing customer basket data
- Redis, Amazon DynamoDB, Oracle NoSQL Database



No-SQL Column-Based Databases

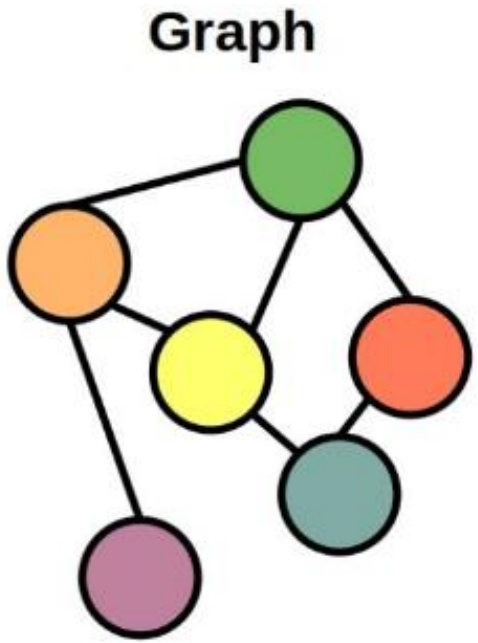
- Wide-Column Databases
- Data is stored in columns
- Access necessary data faster
- Don’t scanning the unnecessary information
- Scale by columns independently
- Data warehouse and Big Data processing
- Apache Cassandra, Apache HBase or Amazon DynamoDB



Wide Column

No-SQL Graph-Based Databases

- Stores data in a graph structure
- Data entities are connected in nodes
- Navigate graph relationships
- Fraud detection, social networks, and recommendation engines
- OrientDB, Neo4j, and Amazon Neptune

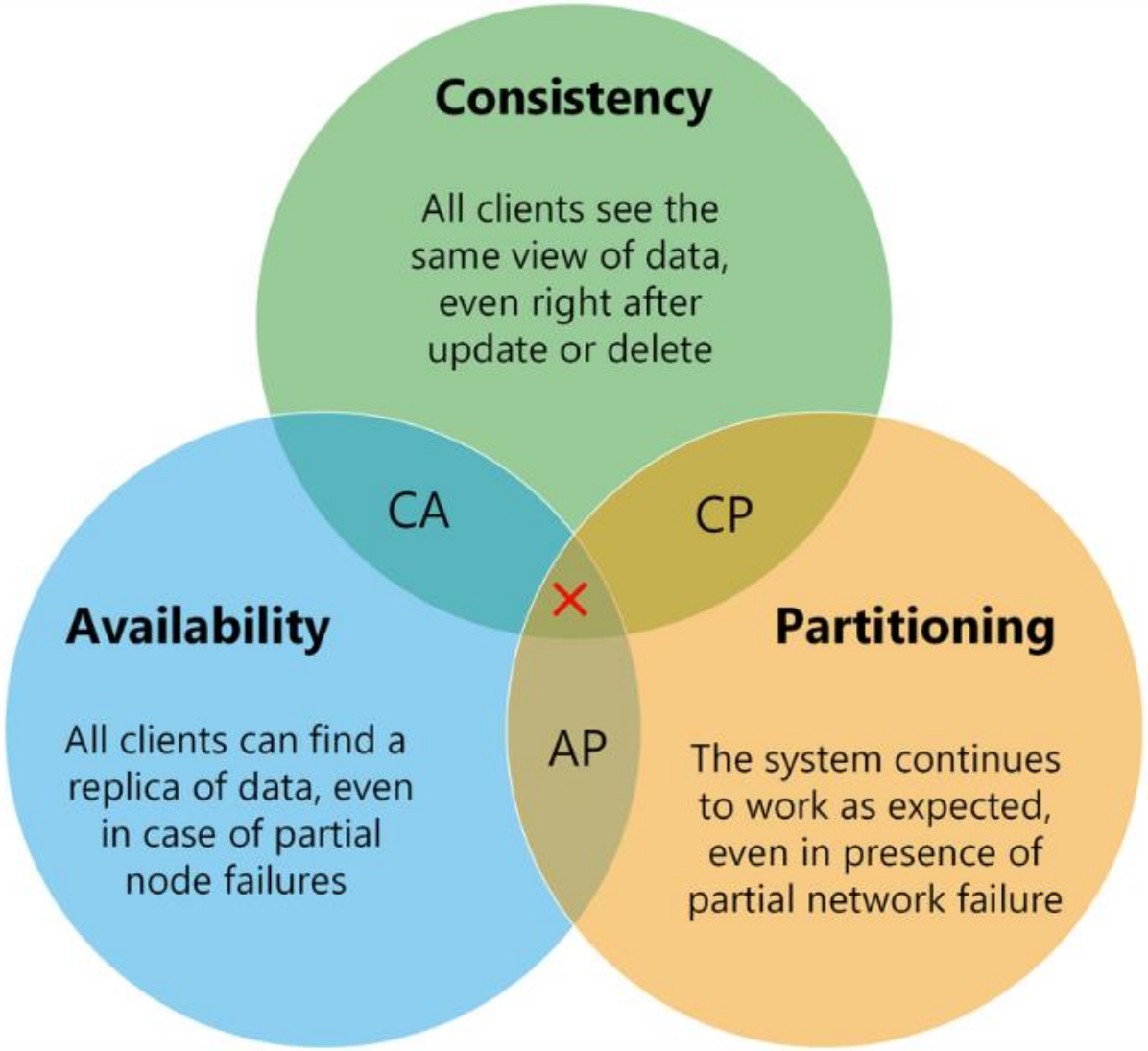


How to Choose a Database for Microservices

- **Key Point 1** - Consider the "consistency level“
- Do we need Strict Consistency or Eventual Consistency ?
- Eventual consistency in microservices architecture in order to gain scalability and high availability
- **Key Point 2** - High Scalability - accommodate millions of request
- **Key Point 3** - High Availability - separate data center
- Before deciding database, we should check the **CAP Theorem**

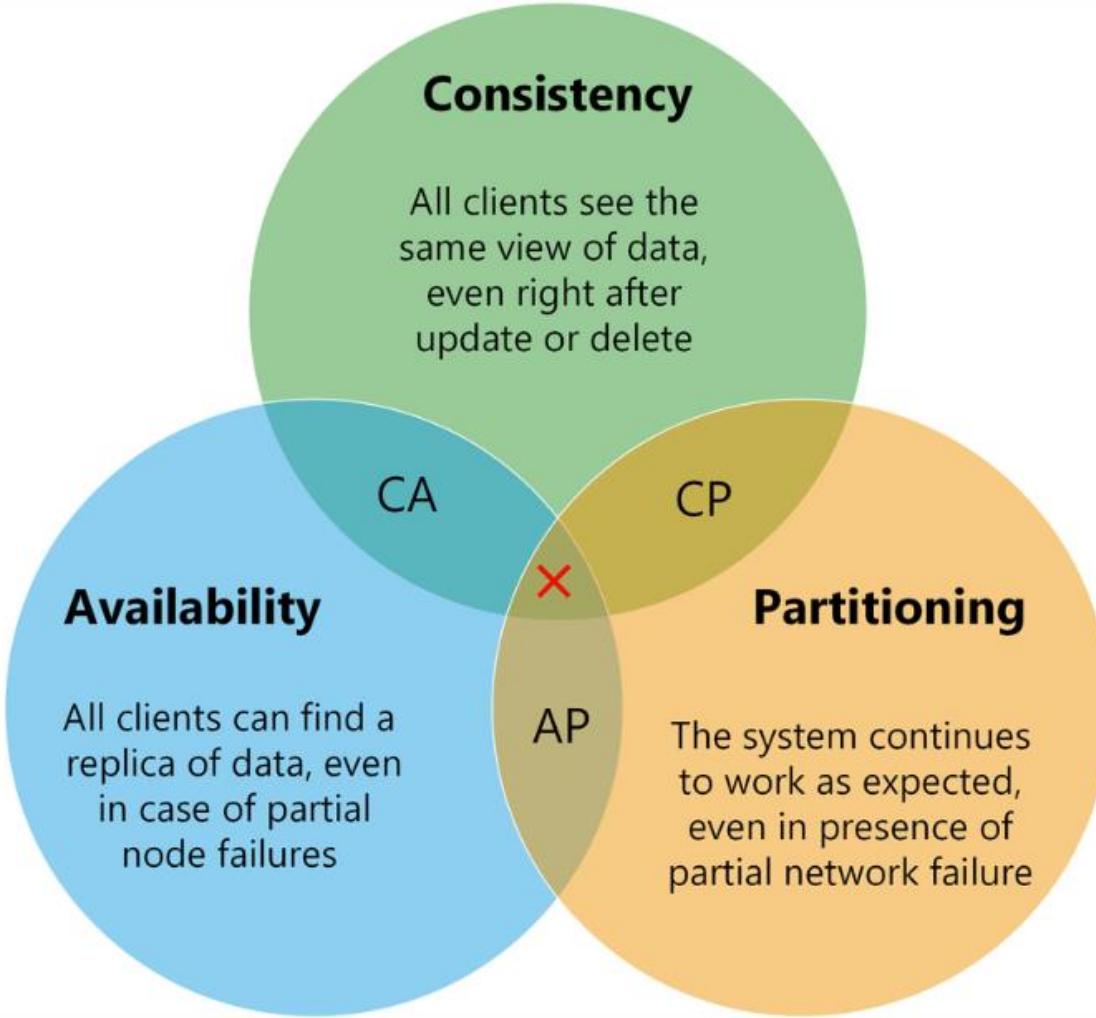
CAP Theorem

- Found in 1998 by a Professor Eric Brewer
- Consistency, Availability, and Partition Tolerance cannot all be achieved at the same time
- Distributed systems should sacrifice between consistency, availability, and partition tolerance.
- Can only guarantee two of the three concepts; consistency, availability, and partition tolerance



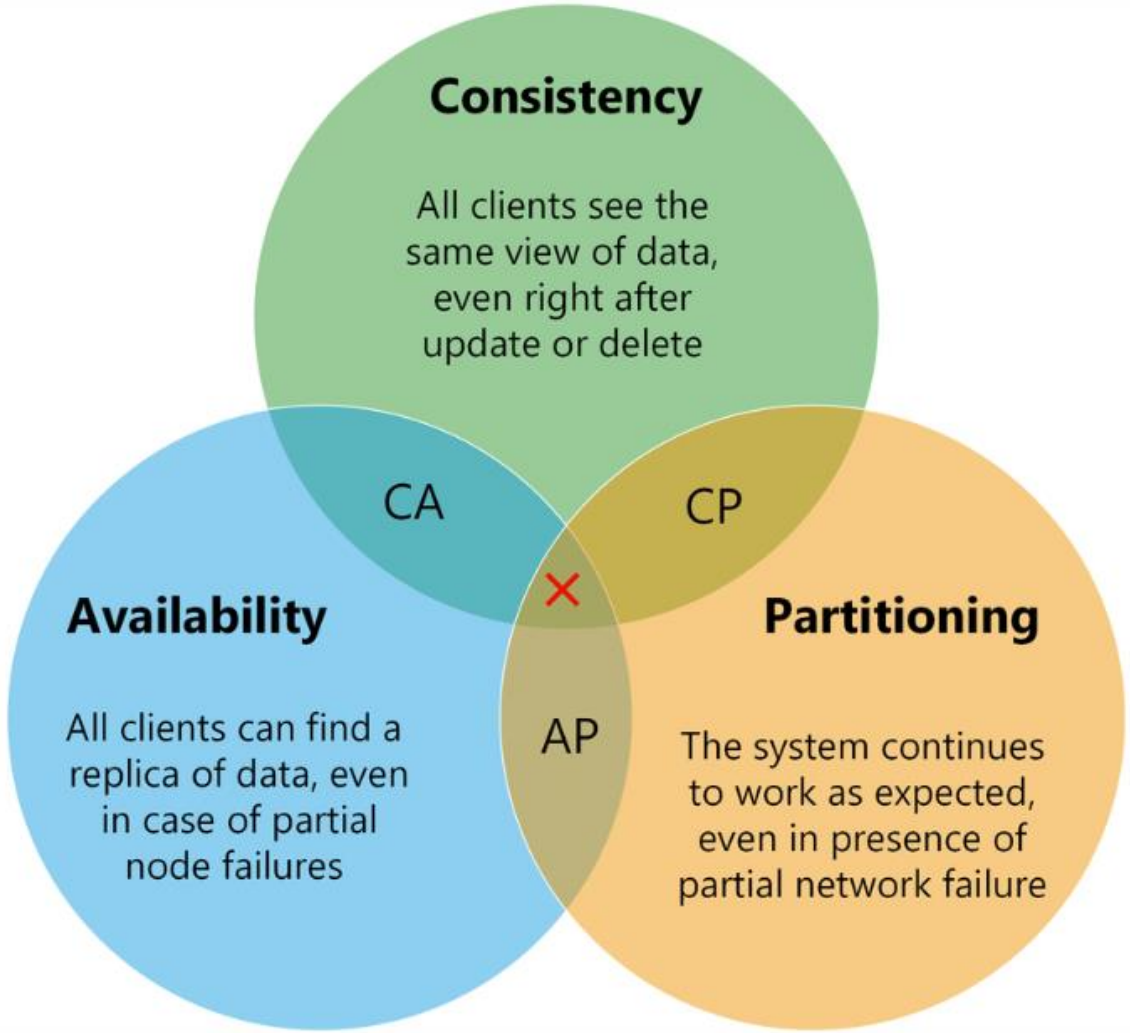
CAP Theorem – Consistency, Availability, Partition Tolerance

- Consistency - get any read request, the data should return last updated value
- Must block the request until all replicas update
- Availability - respond to requests at any time
- fault-tolerance in order to accommodate all requests
- Partition Tolerance - network partitioning, located in different networks



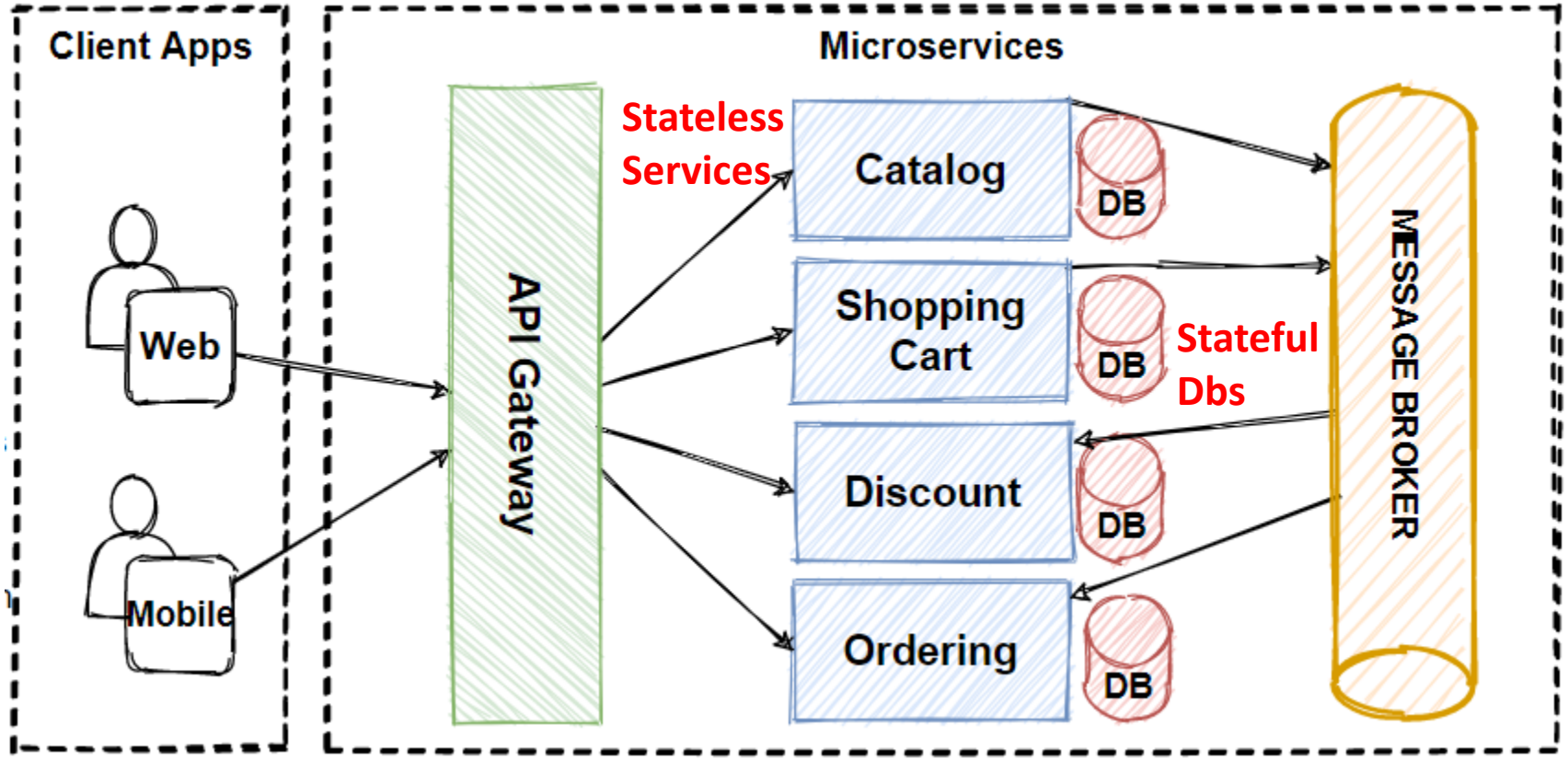
Consistency and Availability at the same time ?

- If Partition Tolerance, either Availability or Consistency should be selected
- Partition Tolerance is a must for distributed architectures
- Relational databases prevent distribute data from different nodes, NoSQL databases easily scalable.
- Microservices architectures choose Partition Tolerance with High Availability and follow Eventual Consistency



Scale Database in Microservices

- Scaling databases in our microservices architecture
- Splitting database servers
- Database Sharding

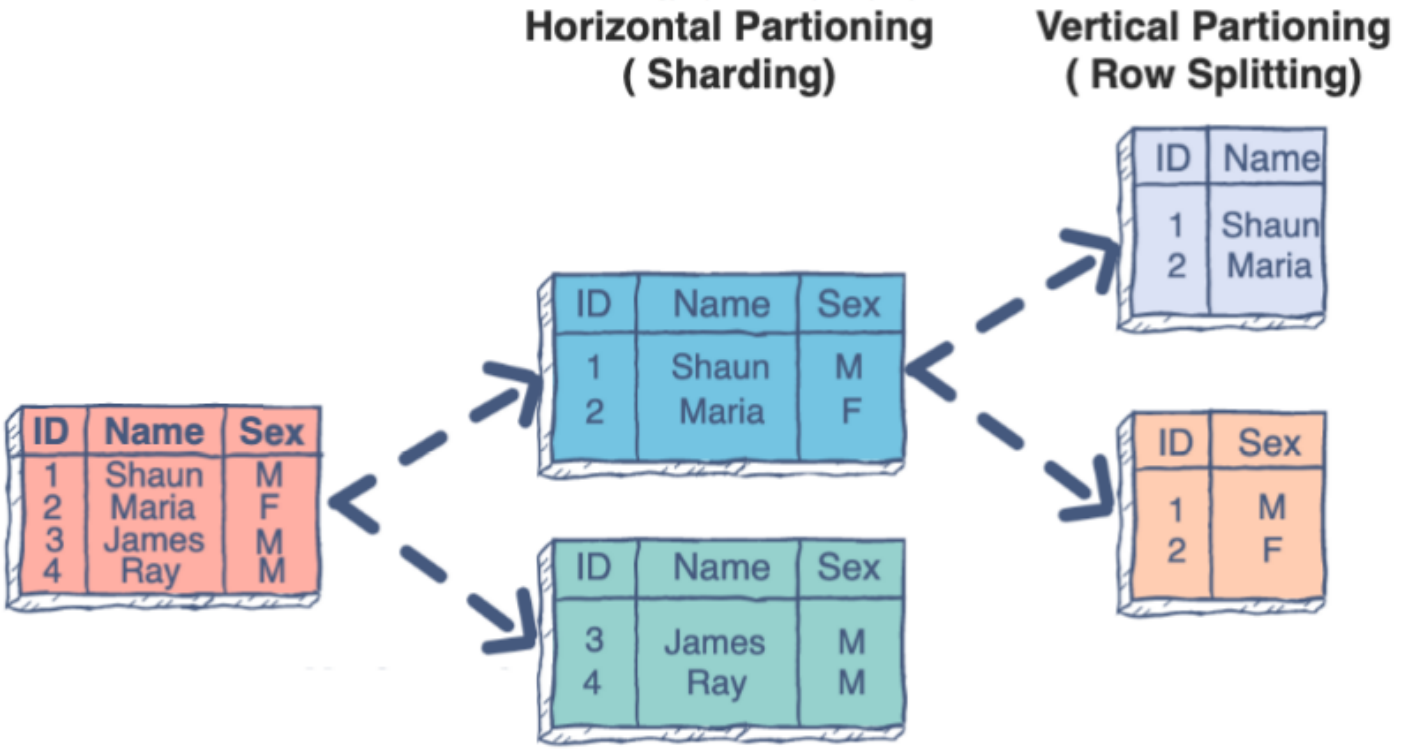


Horizontal, Vertical, and Functional Data Partitioning

- Split databases for scalability

Data Partitioning Types:

- Horizontal Partitioning (often sharding)
- Vertical Partitioning
- Functional Partitioning



Horizontal Partitioning - Sharding

- Each partition is a separate data store
- All partitions have the same schema
- Shards and holds a specific subset of the data
- Sharding keys organized alphabetically
- Sharding separate the load different servers with partition keys

Original Table

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
1	Alice	Anderson	Austin
2	Bob	Best	Boston
3	Carrie	Conway	Chicago
4	David	Doe	Denver

Vertical Shards

VS1

CUSTOMER ID	FIRST NAME	LAST NAME
1	Alice	Anderson
2	Bob	Best
3	Carrie	Conway
4	David	Doe

VS2

CUSTOMER ID	CITY
1	Austin
2	Boston
3	Chicago
4	Denver

Horizontal Shards

HS1

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
1	Alice	Anderson	Austin
2	Bob	Best	Boston

HS2

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
3	Carrie	Conway	Chicago
4	David	Doe	Denver

https://hazelcast.com/glossary/sharding/

Vertical Partitioning

- Row Splitting
- Holds a subset of the columns for table
- Columns are divided according to their pattern
- Frequently accessed columns

Original Table

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
1	Alice	Anderson	Austin
2	Bob	Best	Boston
3	Carrie	Conway	Chicago
4	David	Doe	Denver

Vertical Shards

VS1

CUSTOMER ID	FIRST NAME	LAST NAME
1	Alice	Anderson
2	Bob	Best
3	Carrie	Conway
4	David	Doe

VS2

CUSTOMER ID	CITY
1	Austin
2	Boston
3	Chicago
4	Denver

Horizontal Shards

HS1

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
1	Alice	Anderson	Austin
2	Bob	Best	Boston

HS2

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
3	Carrie	Conway	Chicago
4	David	Doe	Denver

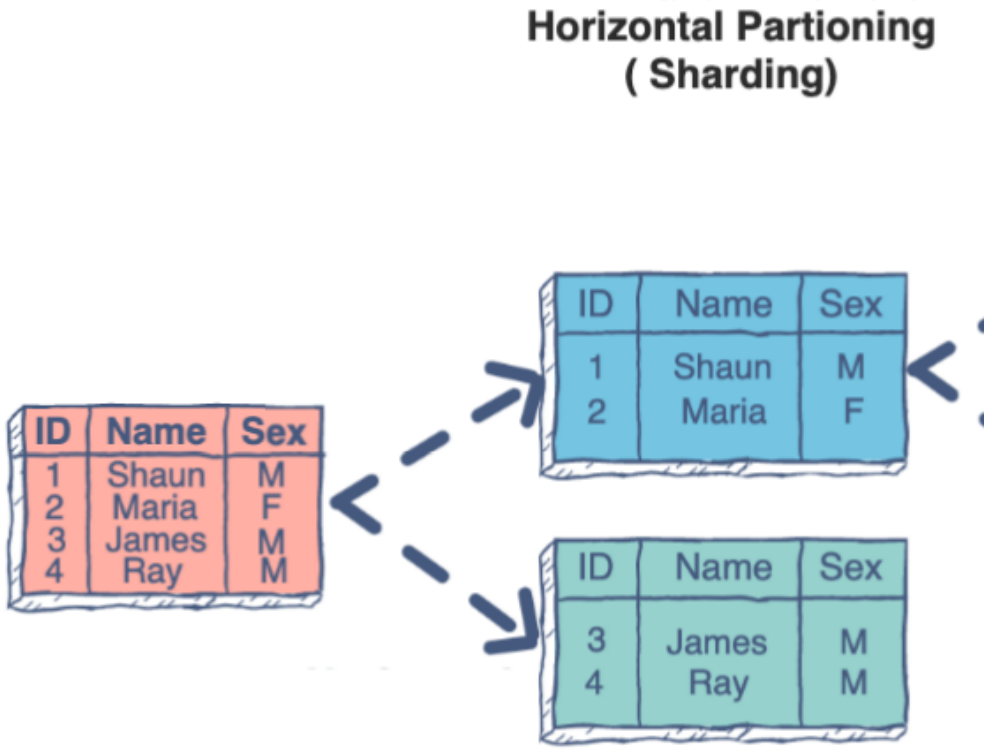
https://hazelcast.com/glossary/sharding/

Functional partitioning

- Functionally partitioning data by following the bounded context or subdomains
- Data is segregated according to usage of bounded contexts
- Like decomposing microservices as per responsibilities

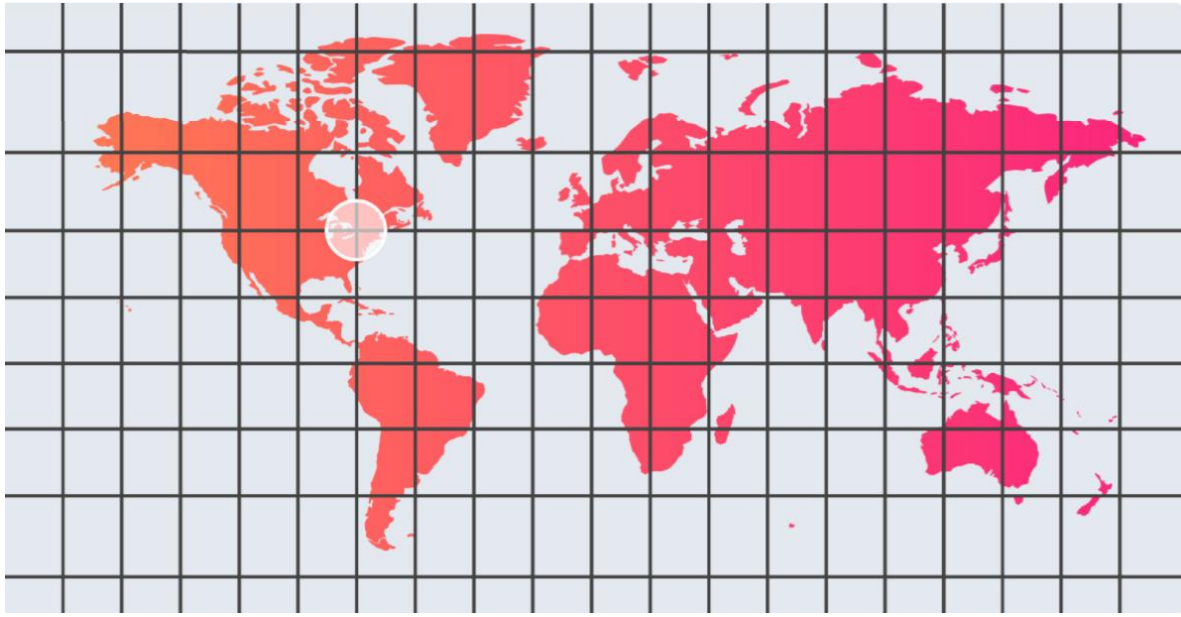
Database Sharding Pattern

- Sharding - "a small piece or part“
- Separation of the data into unique small pieces
- Improve scalability when storing data in microservices
- Each shard has the same schema
- Shardings enable to scale, improve performance by balancing the workload across shards
- Dividing into shards with partition keys



Tinder - Database Sharding Pattern

- Tinder - match and meet other people around 160km based on location
- Find people near you very quickly
- GeoSharding - location-based database sharding
- Dividing the world map into boxes with their locations

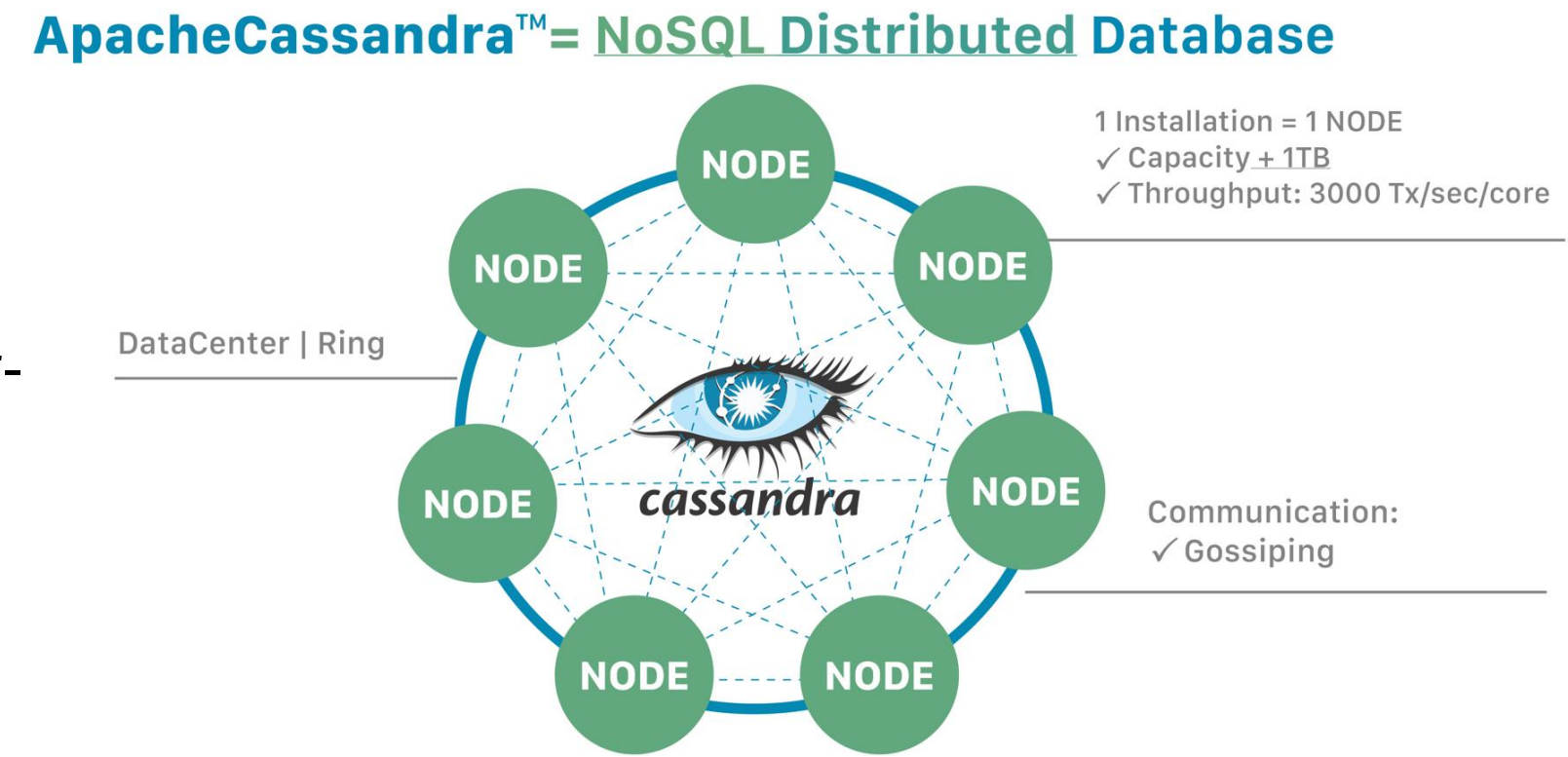


Cassandra No-Sql Database

- Peer-to-Peer Distributed Wide Column Database
- Distributed database from Apache Foundation, highly scalable, high-performance distributed database
- High availability with no single point of failure
- Elastic scalability
- No single point of failure
- Flexible data storage, Easy data distribution

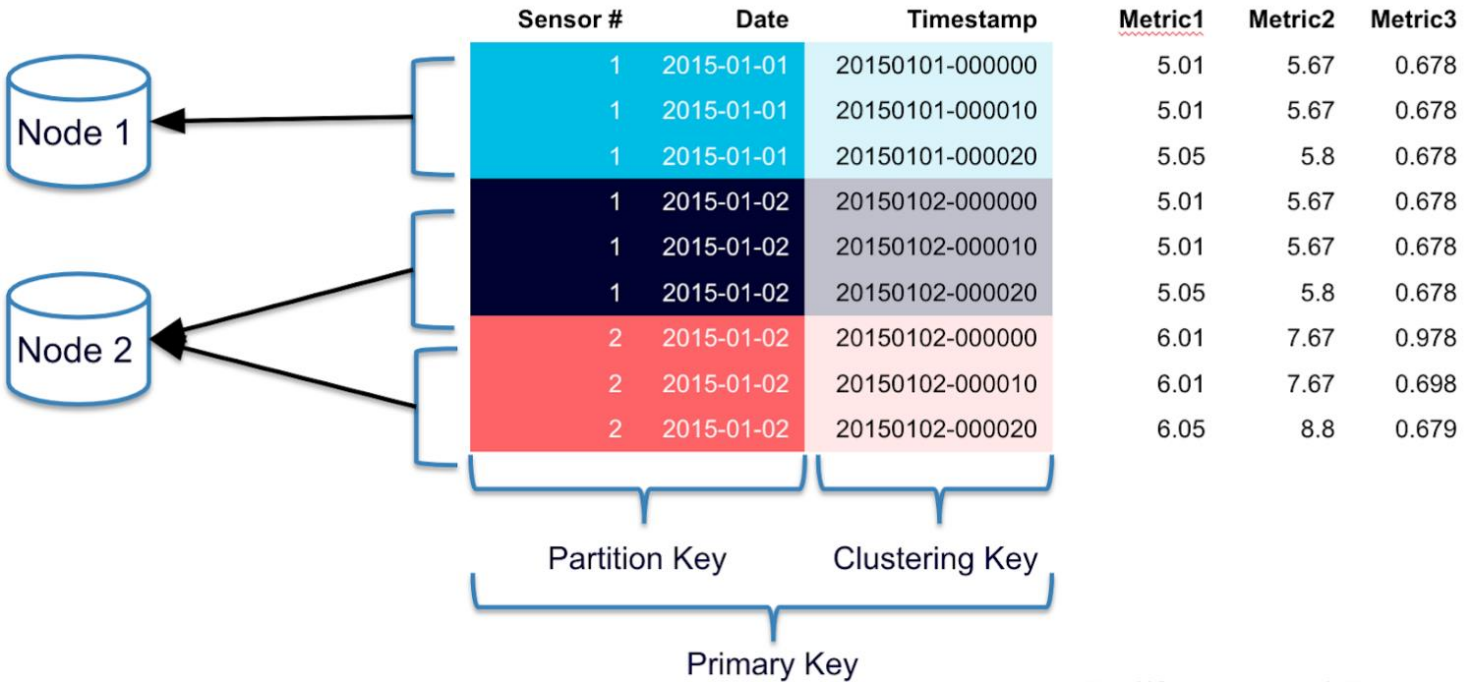
Cassandra Architecture

- Peer-to-peer distributed system across the nodes in a cluster
- Master-Master (Master-Less) architecture
- Every Cassandra Node has the same role



Why Cassandra ?

- Auto-sharding feature
- Data Sharding helps keep data divided among nodes
- Partition Keys set to Sensor# and Date.
- Best choose for microservices database
- CAP Theorem High Availability with Eventual Consistency

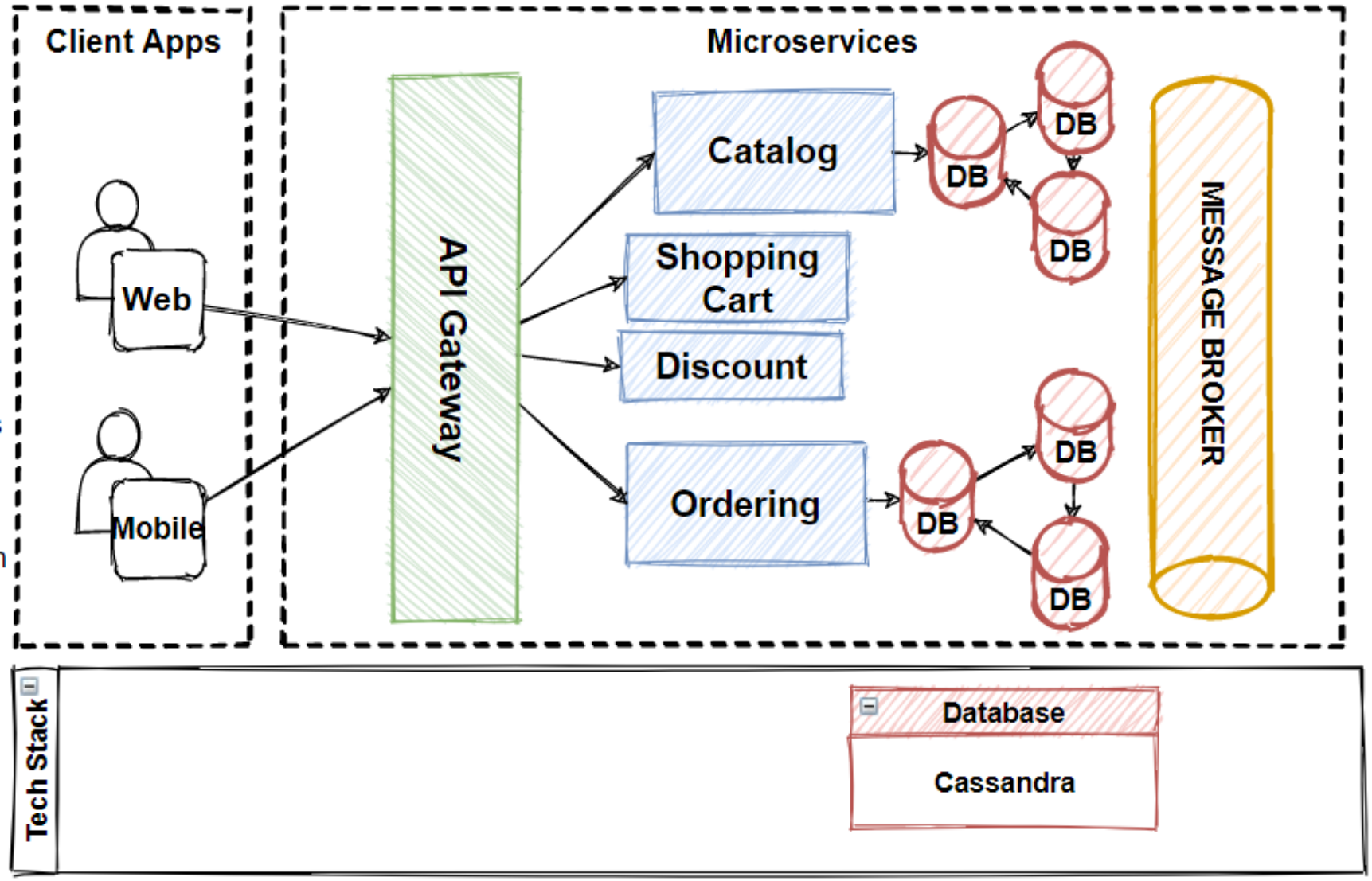


Sensor #	Date	Timestamp	Metric1	Metric2	Metric3
1	2015-01-01	20150101-000000	5.01	5.67	0.678
1	2015-01-01	20150101-000010	5.01	5.67	0.678
1	2015-01-01	20150101-000020	5.05	5.8	0.678
1	2015-01-02	20150102-000000	5.01	5.67	0.678
1	2015-01-02	20150102-000010	5.01	5.67	0.678
1	2015-01-02	20150102-000020	5.05	5.8	0.678
2	2015-01-02	20150102-000000	6.01	7.67	0.978
2	2015-01-02	20150102-000010	6.01	7.67	0.698
2	2015-01-02	20150102-000020	6.05	8.8	0.679

Microservices Architecture - Database Sharding

Principles

- KISS
- YAGNI
- SoC
- SOLID
- Gateway Routing
- Gateway Aggregation
- Gateway Offloading
- Api Gw
- Database per Microservices
- Backends for Frontends pattern BFF
- Service Aggregator Pattern
- Service Registry Pattern
- Dependency Inversion Principles (DIP)
- Publish–Subscribe Design Pattern
- Database Sharding Pattern

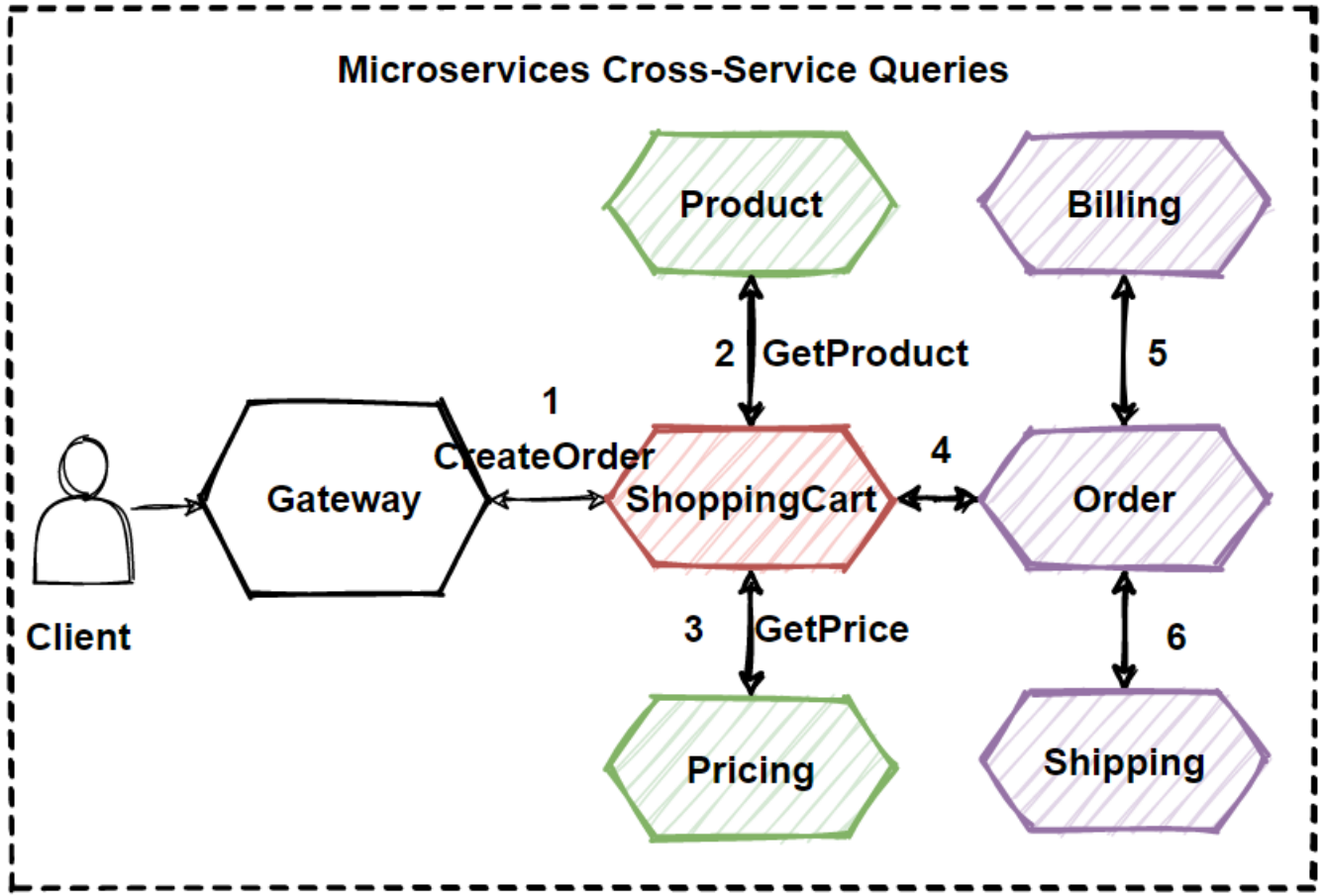


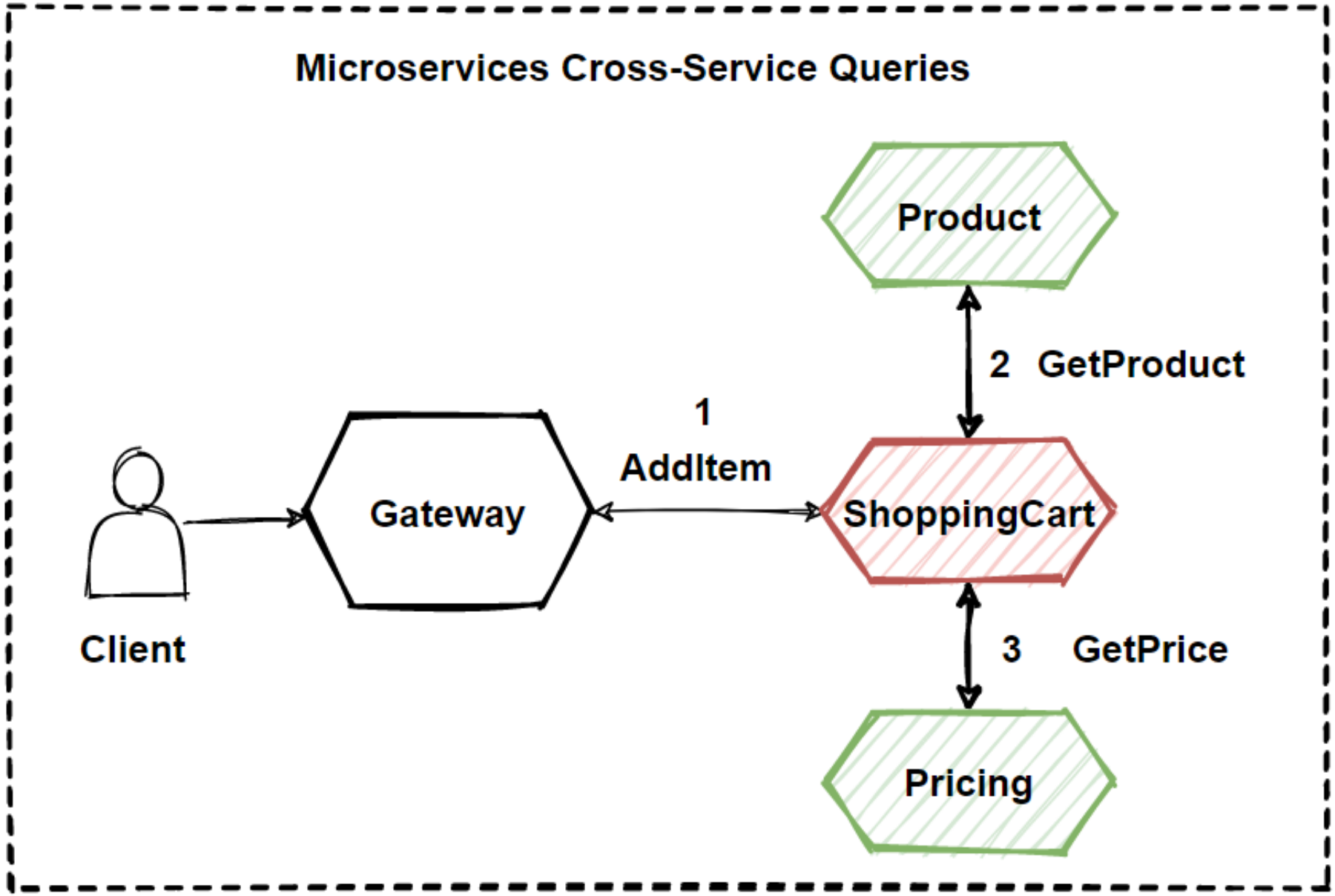
Section 14
Microservices Data Management - Queries

Microservices Data Query Pattern and Best Practices

Microservices Cross-Service Queries

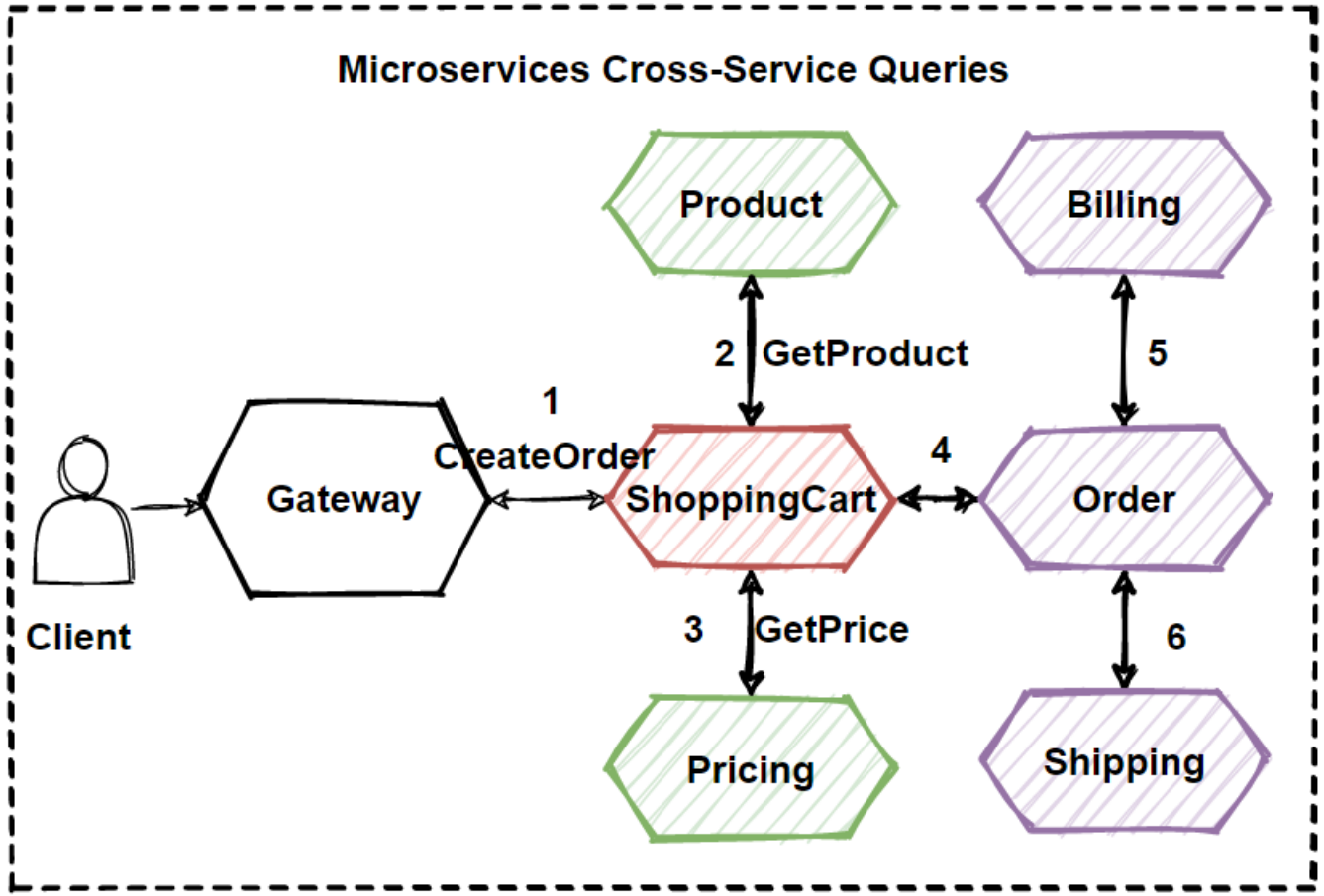
- Perform queries across microservices
- Monolithic easy to query due tc single database
- Microservices polyglot persistence
- Query request to internal microservices
- Microservices are independent




Microservices Cross-Service Queries
Product
2 GetProduct
1
AddItem
Gateway
ShoppingCart
Client
3 GetPrice
Pricing

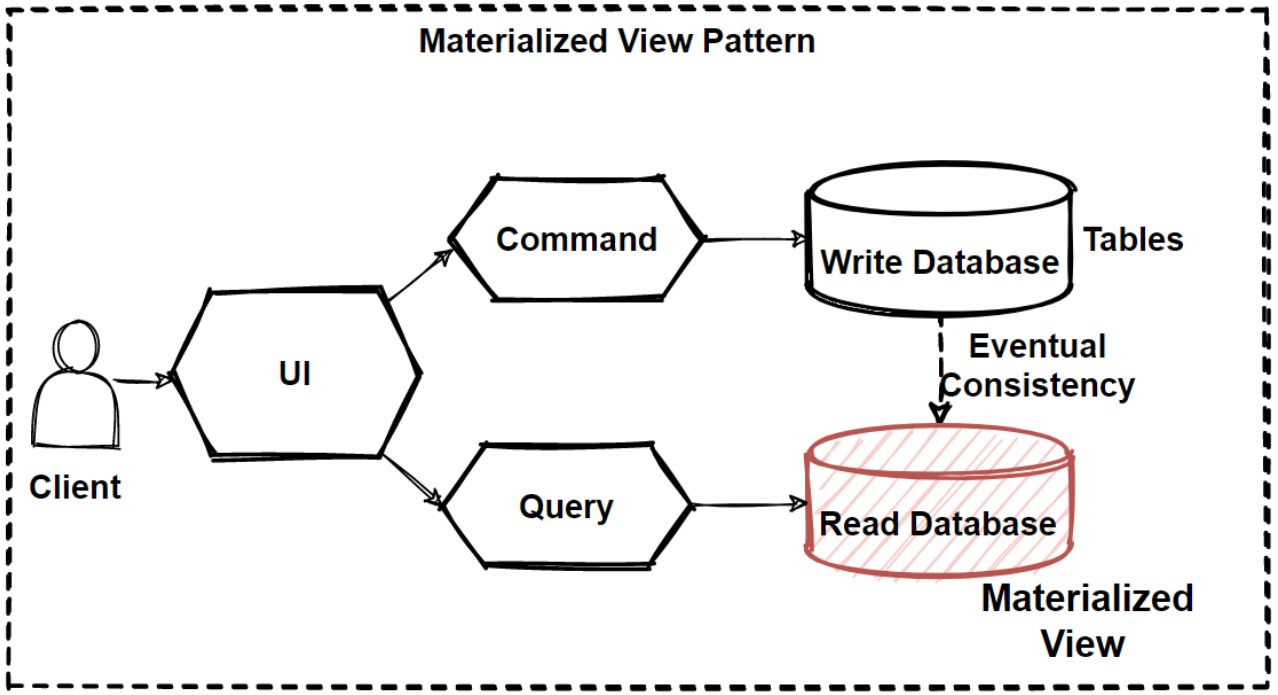
Cross Query Solutions in Microservices

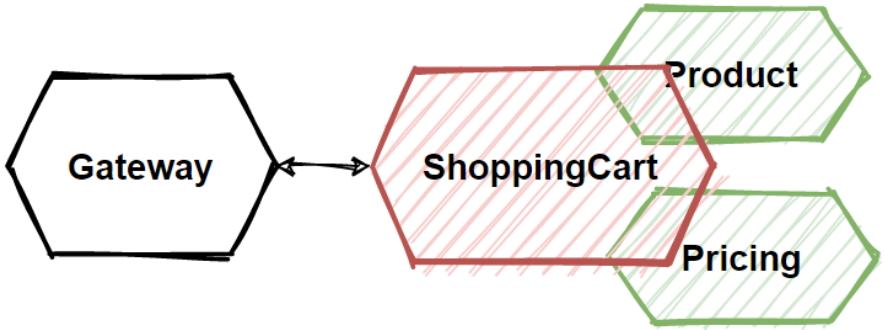
- Perform queries across microservices database level in data stores
- API Gateway patterns, Service Aggregator pattern
- Add a product into the user's shopping cart with get data Catalog and Pricing microservice
- Reducing independency of microservices and makes chatty communications



Materialized View Pattern

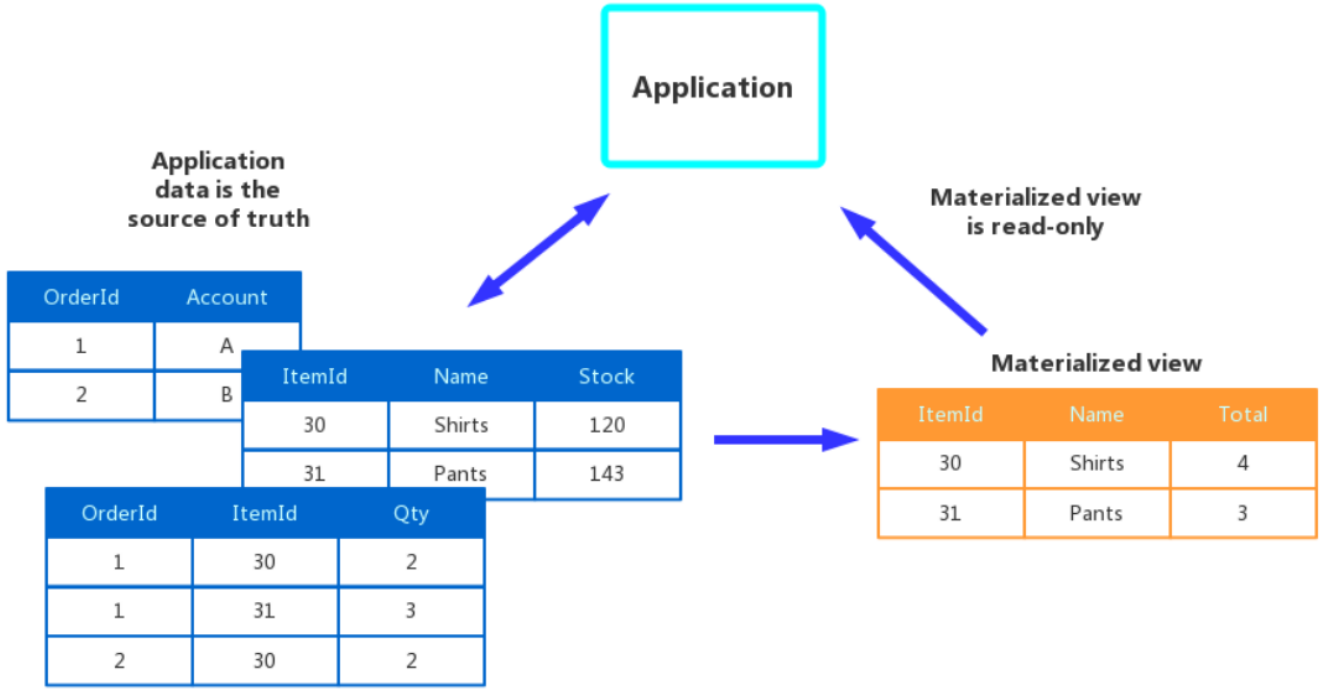
- Store its own local copy of data
- Contains a denormalized copy of the data
- Local copy of data as a Read Model
- Shopping Basket microservice contains a denormalized copy of the data which product and pricing microservices
- Eliminates the synchronous cross-service calls

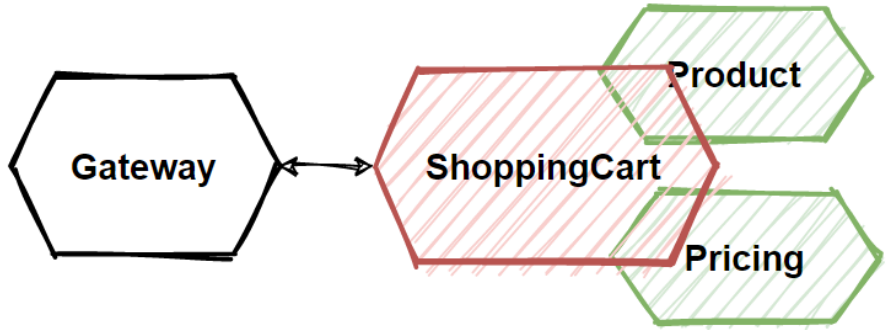




Drawbacks of Materialized View Pattern

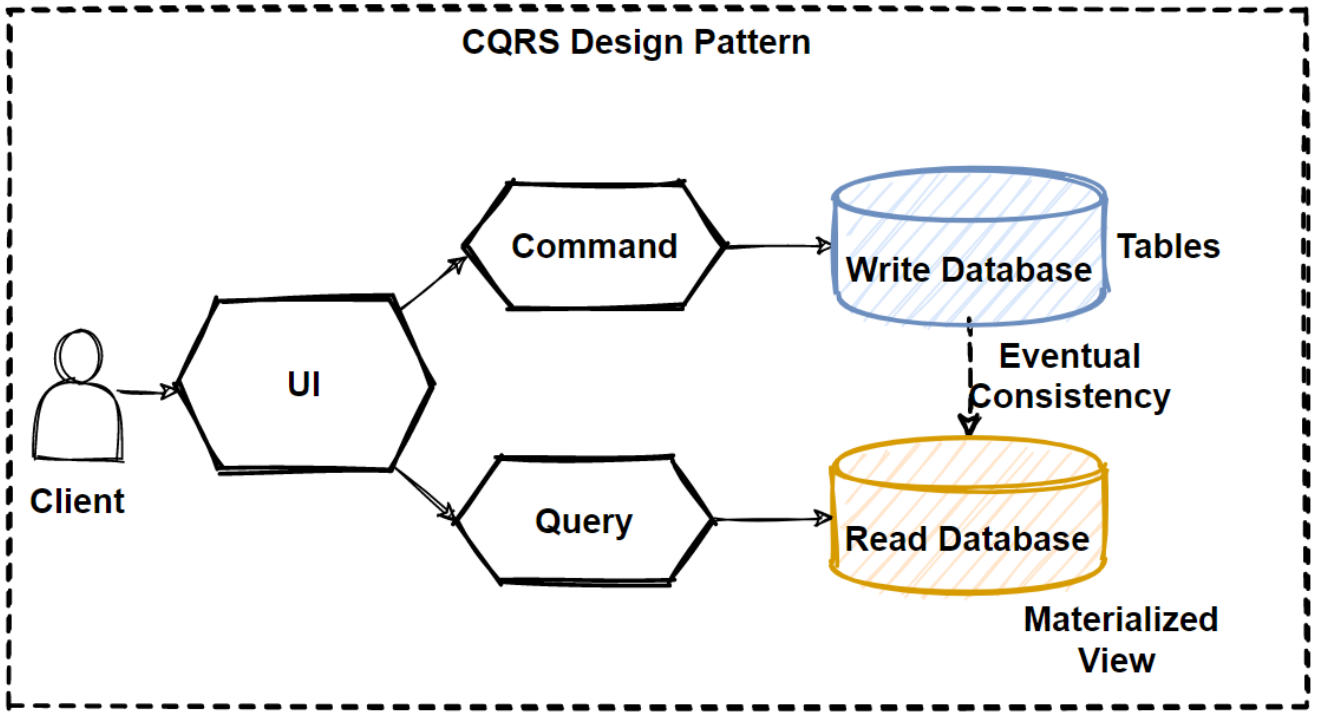
- How and when the denormalized data will be updated ?
- Source of data is other microservices
- When the original data changes it should update into sc microservices
- Publish an event and consumes from the subscriber microservice





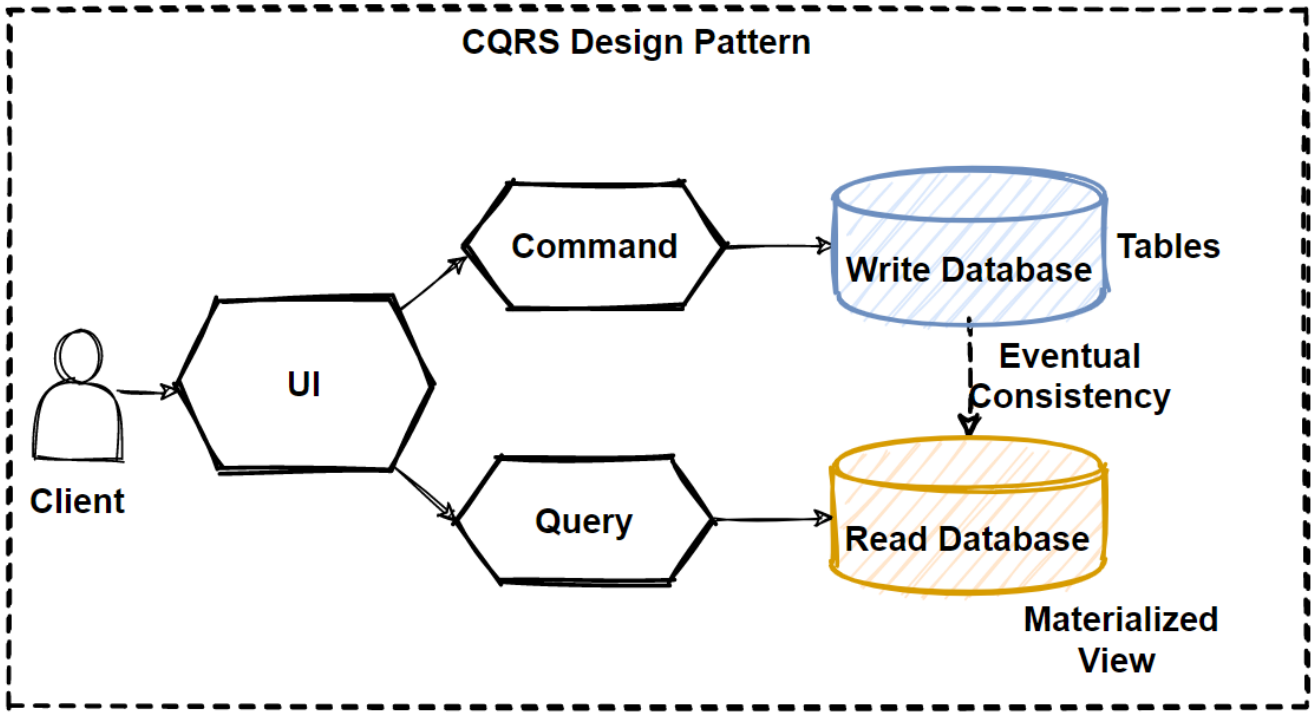
CQRS Design Pattern

- Stands for Command and Query Responsibility Segregation
- Applications need both working for complex join queries and also perform CRUD operations
- Reading and writing database has different approaches
- Using no-sql for reading and using relational database for crud operations
- Read-incentive application



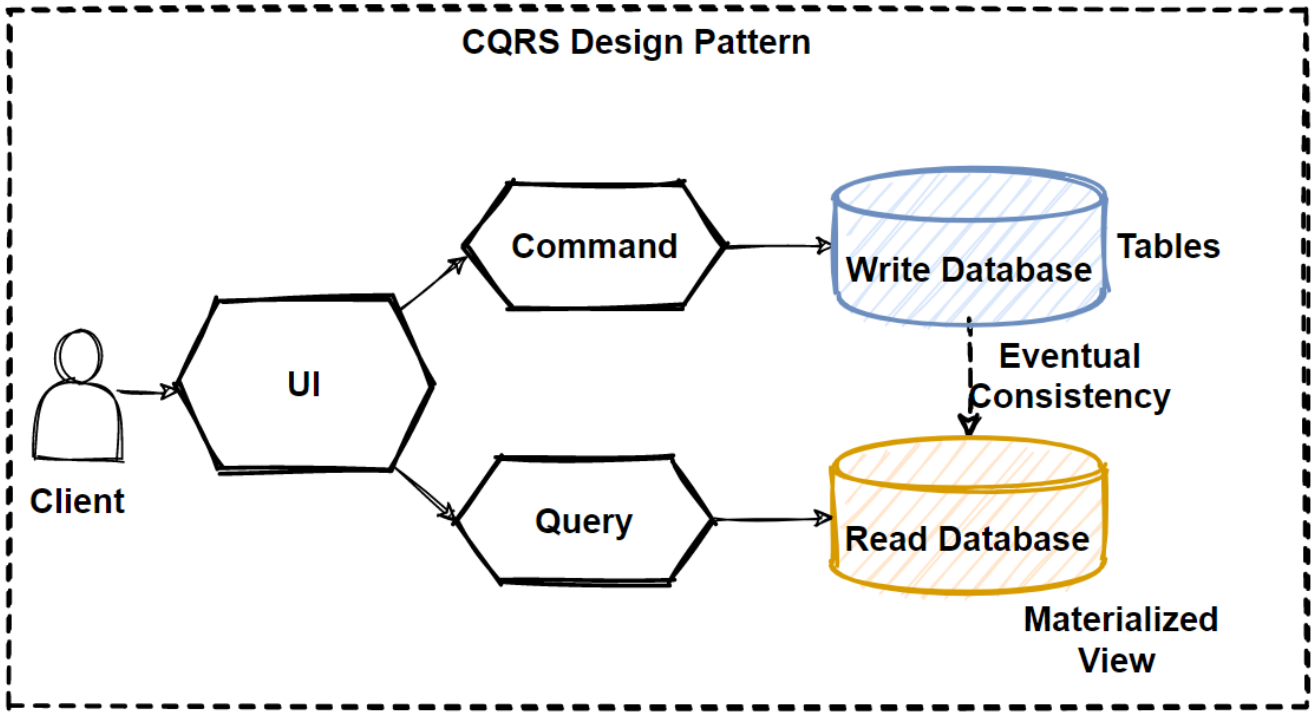
CQRS Design Pattern

- Commands performs update data
- Queries performs read data
- Commands should be actions with task-based operations
- Queries is never modifying the database



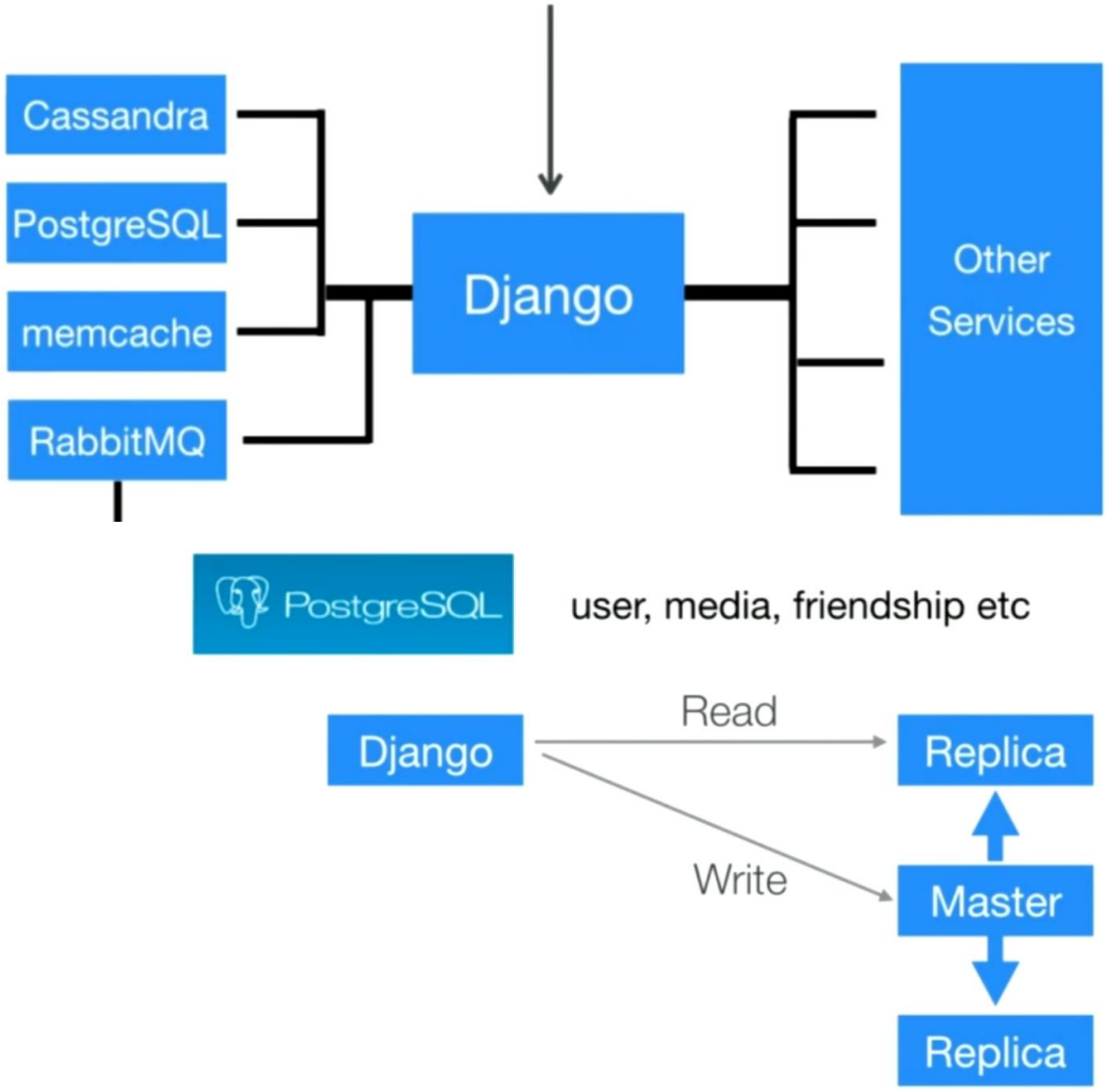
CQRS Design Pattern 2

- Isolate Commands and Queries
- Read and Write database with 2 database
- Materialized view pattern is good example to implement reading databases
- Using no-sql for reading and using relational database for crud operations



Instagram Database Architecture

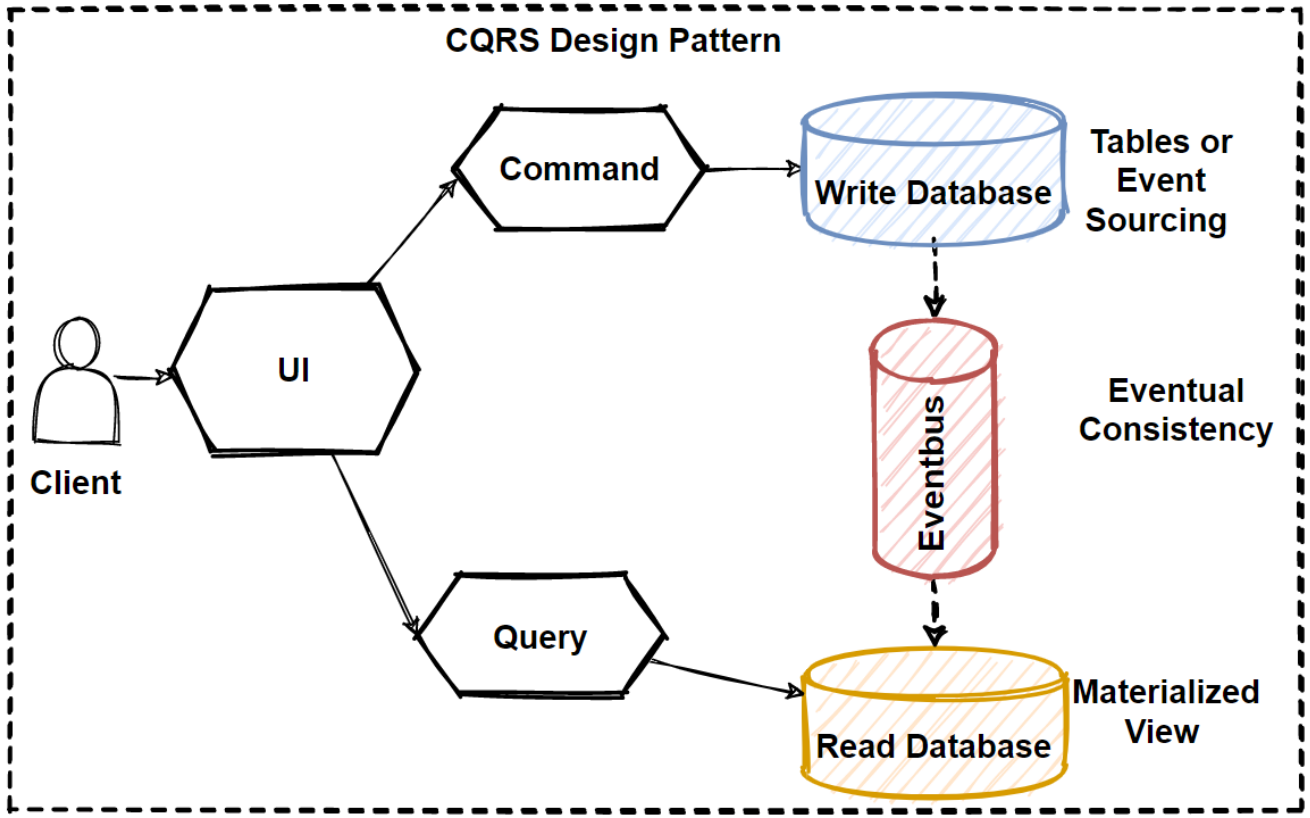
- Uses two database systems for different use cases
- Relational database - PostgreSQL and the other is no-sql database - Cassandra
- Uses no-sql Cassandra database for user stories
- Uses relational PostgreSQL database for User Information bio update



https://arunabhdas.medium.com/infrastructure-scaling-case-study-instagram-5dd46a8d2974

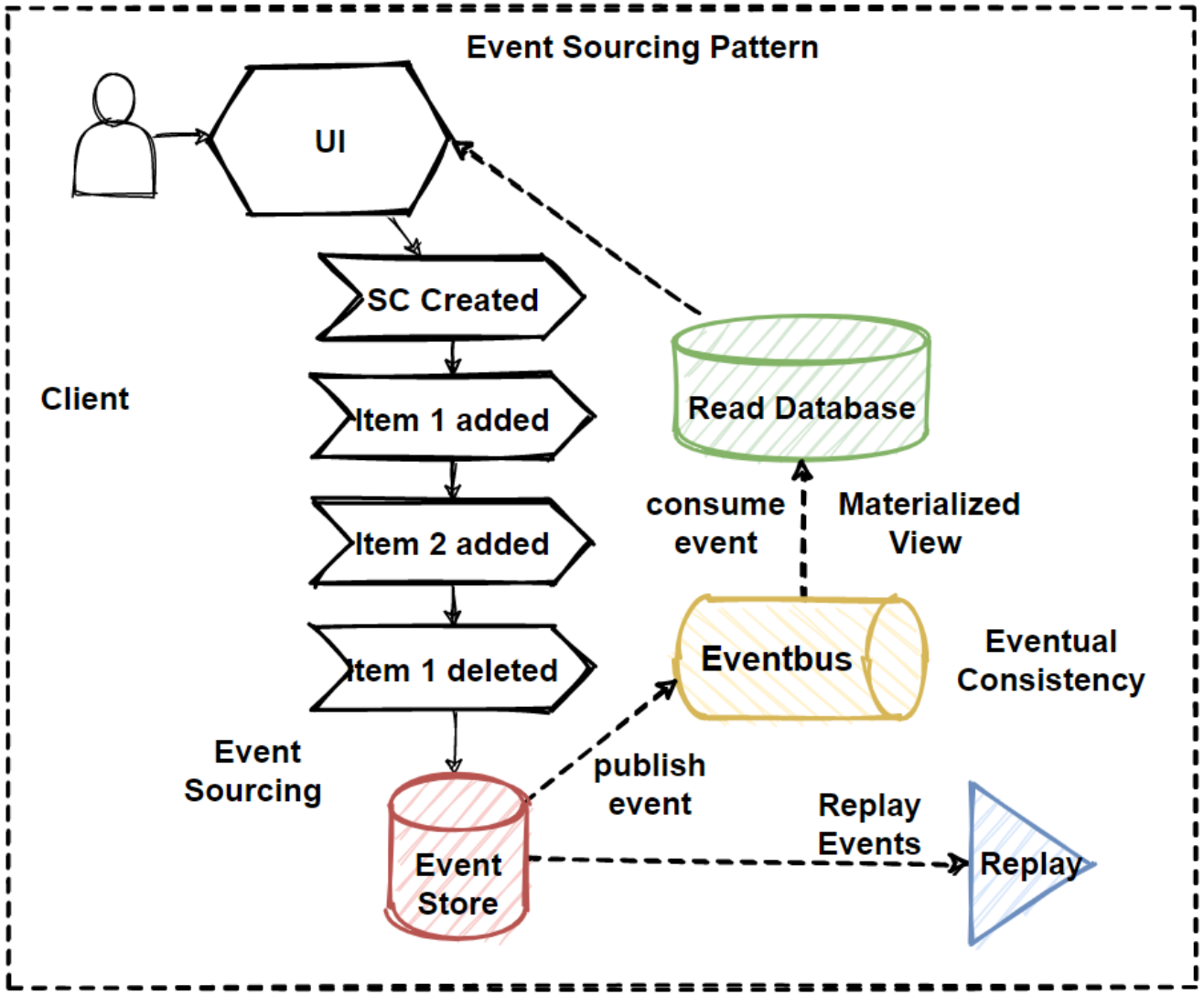
How to Sync Databases with CQRS ?

- Event-Driven Architecture
- Publish an update event with using message broker systems
- Consume by the read database and sync data
- Read database eventually synchronizes with the write database
- Read database from replicas of write database with applying Materialized view pattern



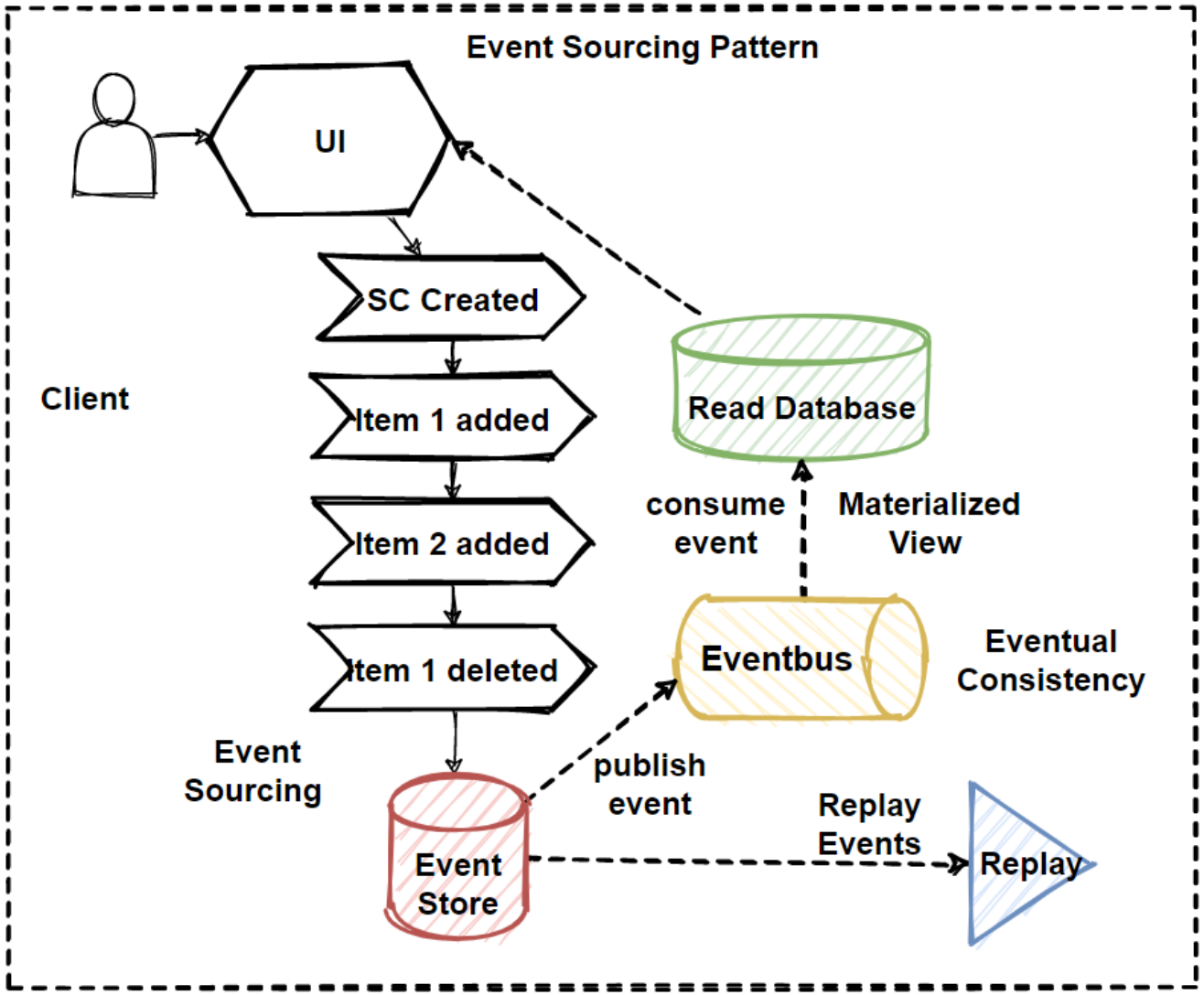
Event Sourcing Pattern

- CQRS with Event Sourcing Pattern
- Source-of-truth events database
- Materialized views of the data with denormalized tables
- Publish an update event with using message broker systems
- Consume by the read database and sync data Materialized view pattern
- Read database eventually synchronizes with the write database



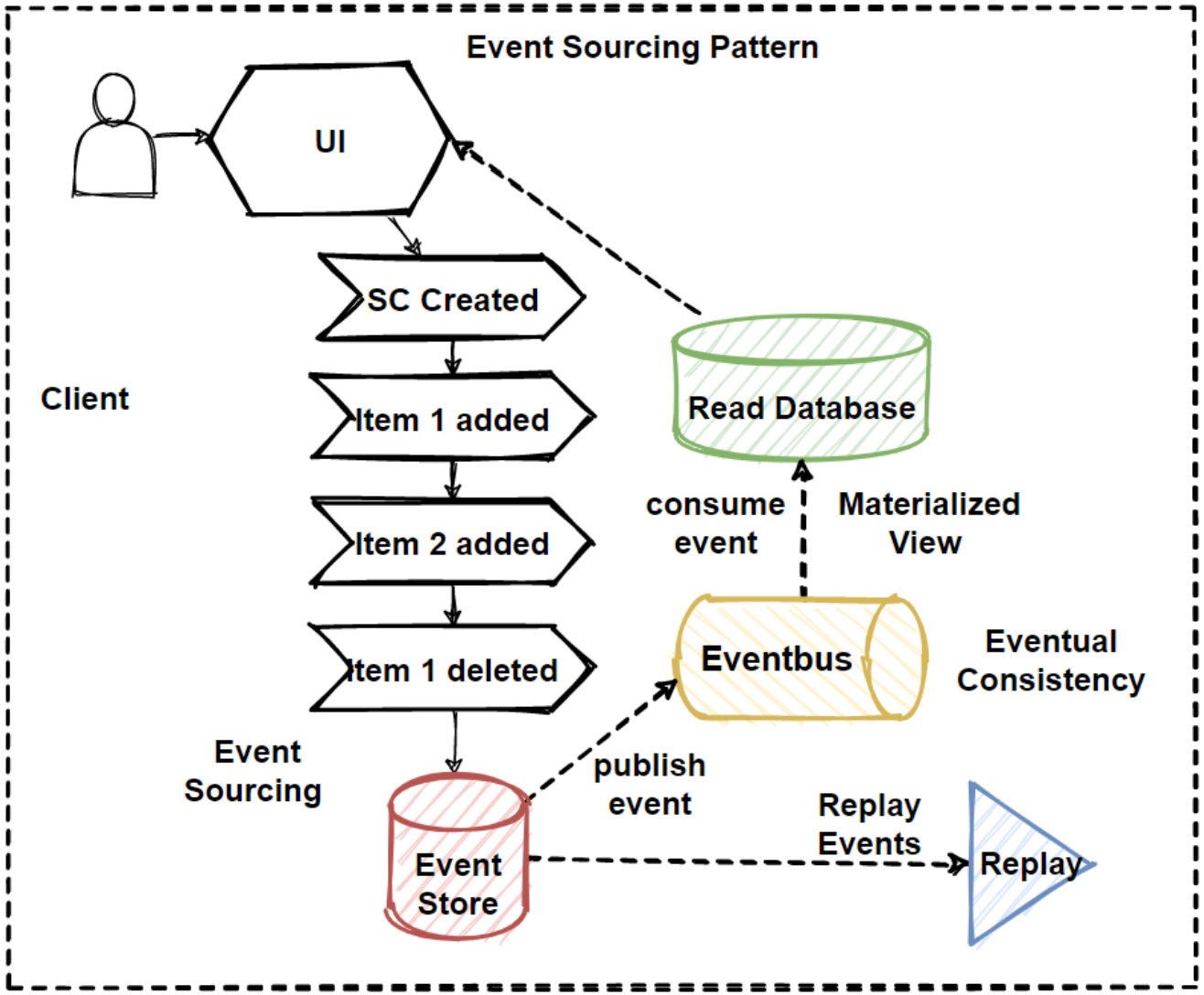
Event Sourcing Pattern 2

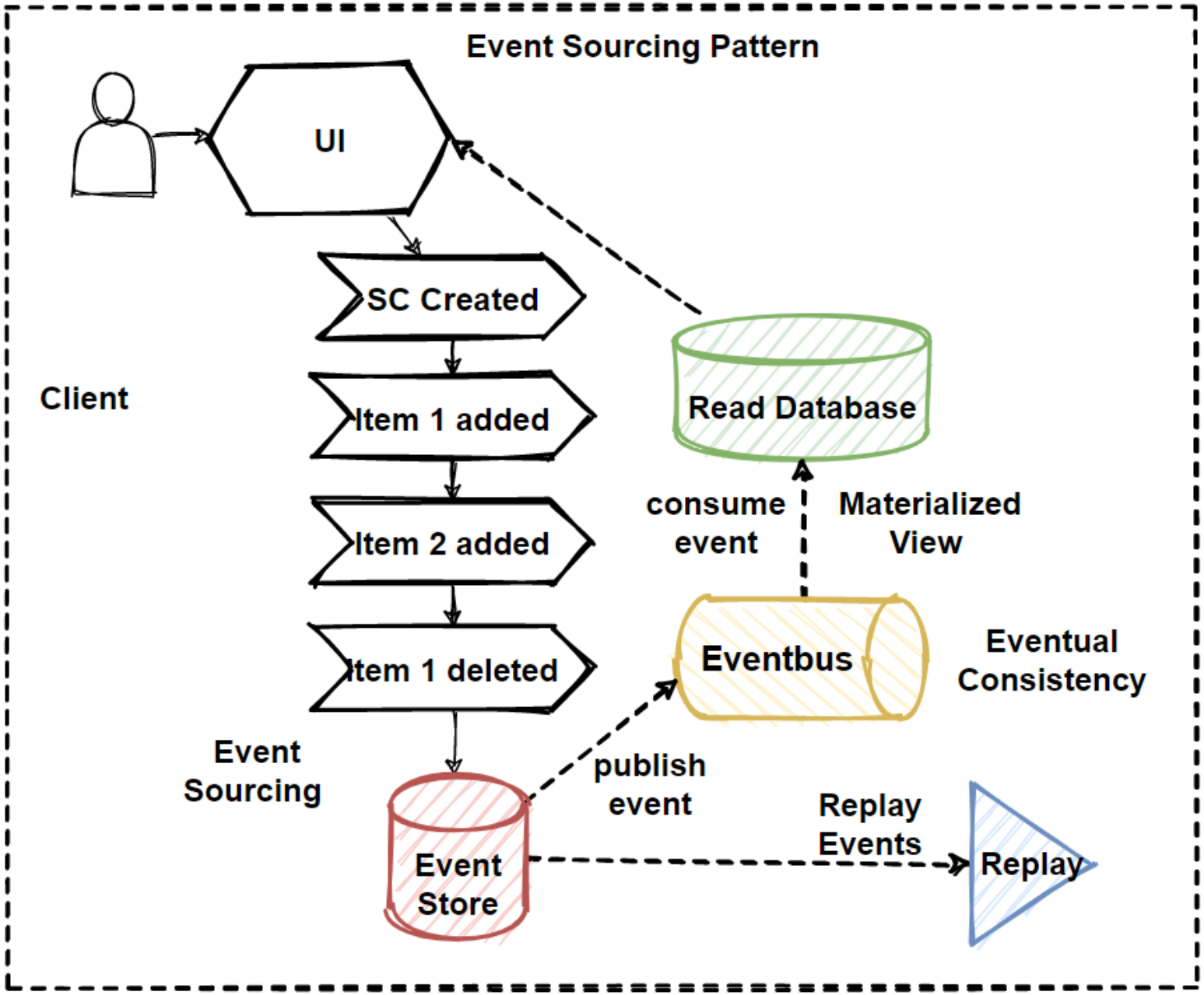
- Changing to data save operations
- Save all events into database with sequential ordered of data events
- Append each change to a sequential list of events
- Event Store becomes the source-of-truth for the data
- Publish/subscribe pattern with publish event
- Replay events to build latest status of data



Eventual Consistency Principle

- CQRS with Event Sourcing Pattern
- For systems that prefer high availability to instant consistency
- Become consistent after a certain time
- Does not guarantee instant consistency
- Consider the "consistency level“
- Strict consistency or Eventual consistency

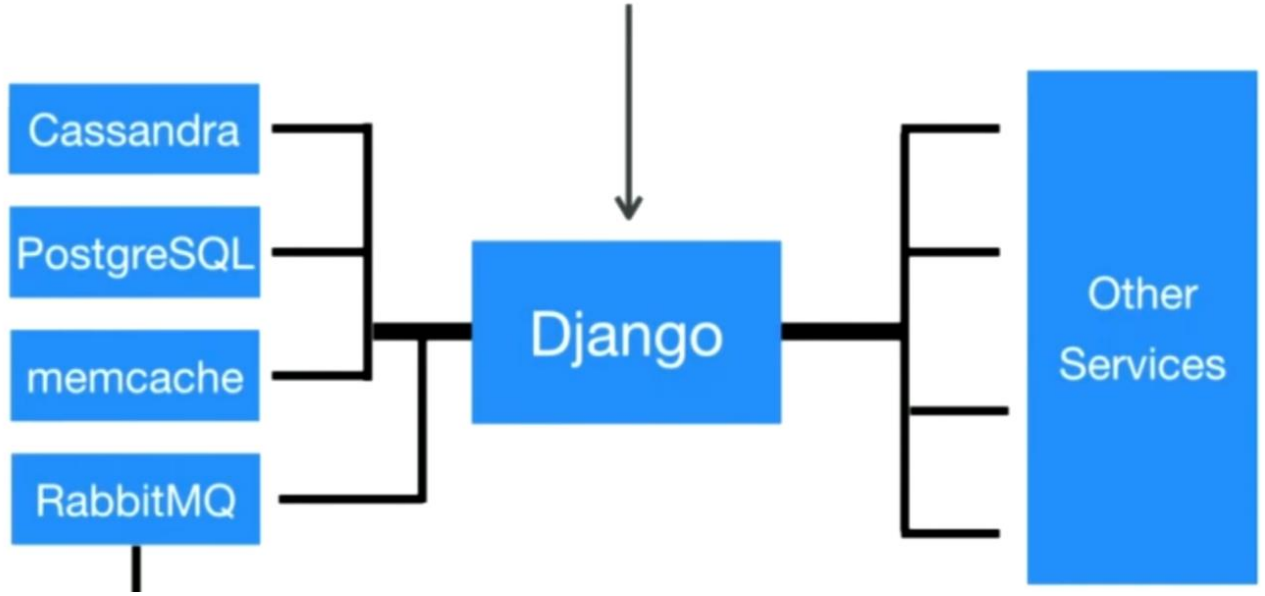




- **Materialized View Pattern**
- **CQRS Design Pattern**
- **Event Sourcing Pattern**
- **Eventual Consistency Principle**

Instagram System Architecture

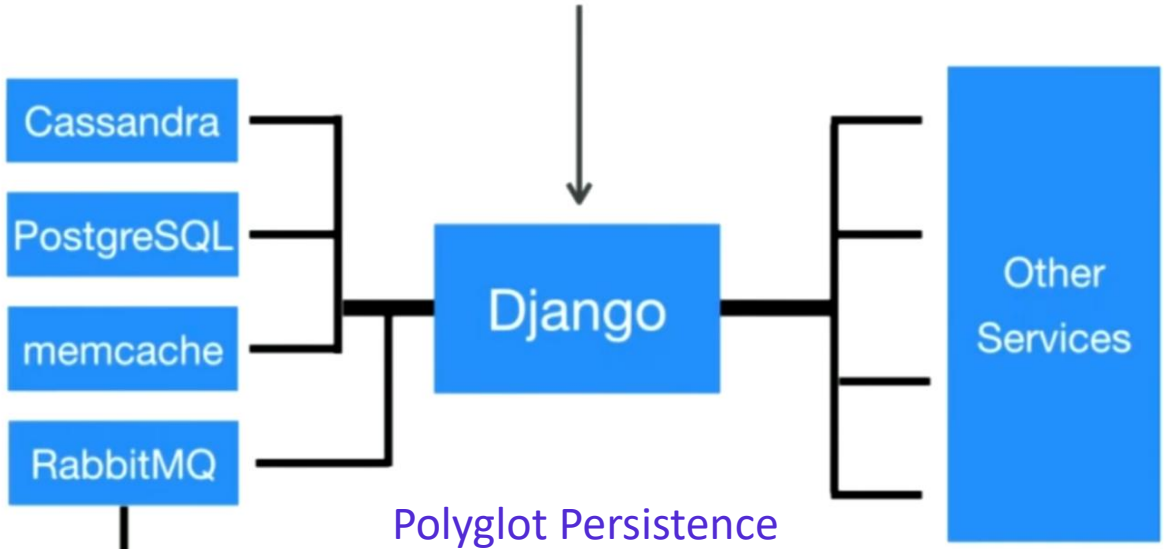
- Visited by 500 million people, Almost 100 million photos are shared, 400 million stories shared
- Huge amount of data and interactions that strong architecture
- Data center to spread to many parts of the world
- Instagram decided that Availability was more important to them and thought that Eventual Consistency would be sufficient



https://arunabhdas.medium.com/infrastructure-scaling-case-study-instagram-5dd46a8d2974

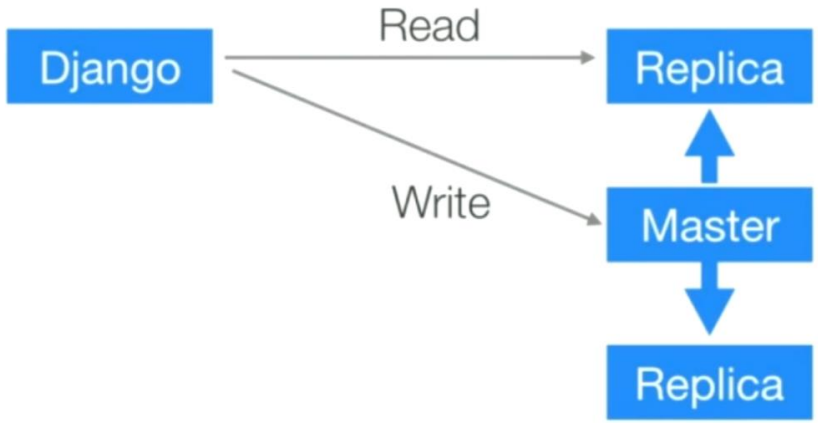
Instagram Database Architecture

- Relational database - **PostgreSQL** and the other is no-sql database – **Cassandra**
- **PostgreSQL** has a **“Master-Slave”** architecture
- Uses relational **PostgreSQL** database for User Information bio update
- Uses no-sql **Cassandra** database for user stories, Counters, messaging
- Cassandra has an **auto-sharding** feature



Polyglot Persistence

user, media, friendship etc

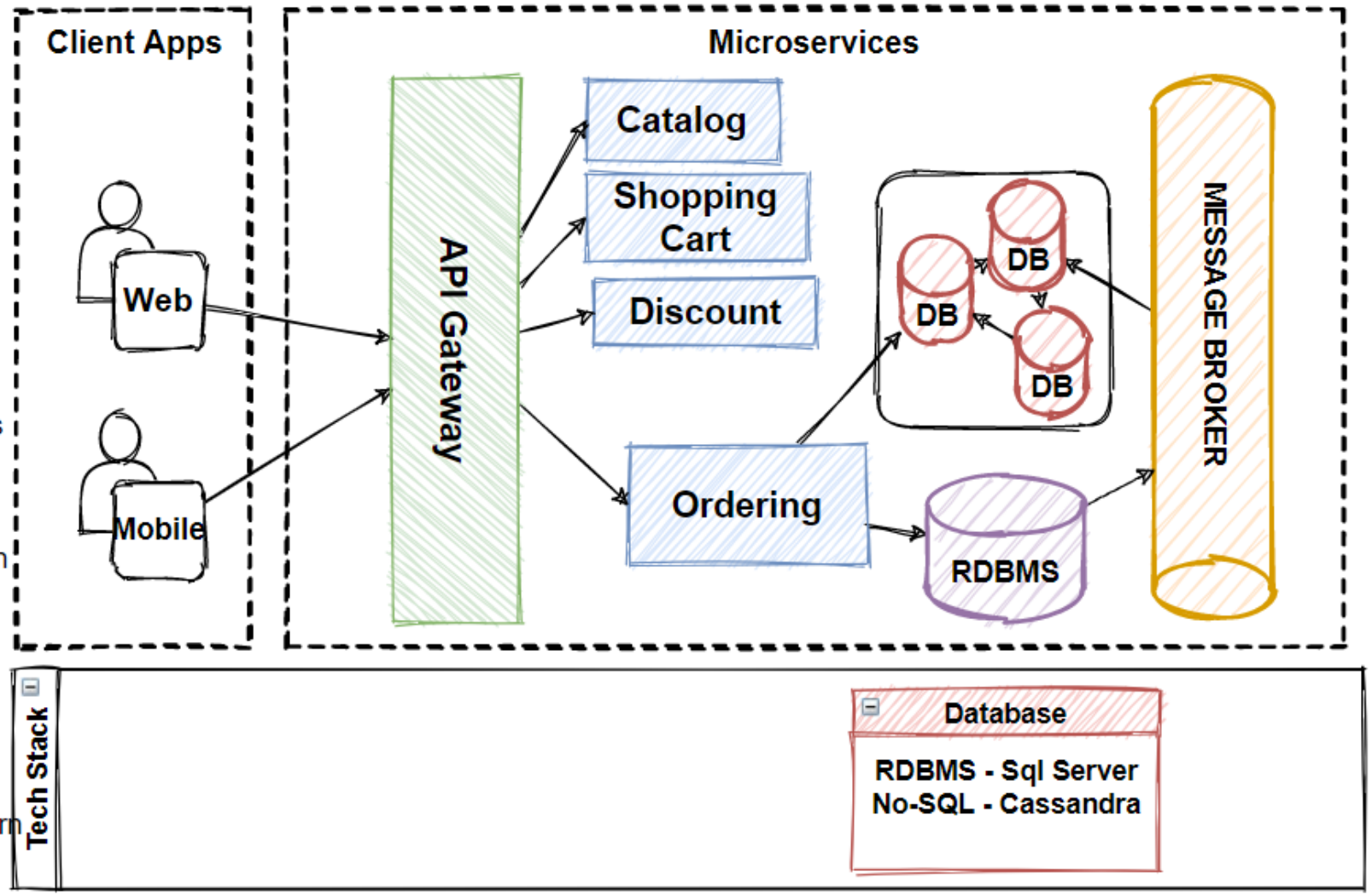


https://arunabhdas.medium.com/infrastructure-scaling-case-study-instagram-5dd46a8d2974

Microservices Architecture - CQRS, Event Sourcing, Eventual Consistency

Principles

- KISS
- YAGNI
- SoC
- SOLID
- Gateway Routing
- Gateway Aggregation
- Gateway Offloading
- Api Gw
- Database per Microservices
- Backends for Frontends pattern BFF
- Service Aggregator Pattern
- Service Registry Pattern
- Dependency Inversion Principles (DIP)
- Publish–Subscribe Design Pattern
- Database Sharding Pattern
- Polygot Persistence
- Materialized View Pattern
- CQRS Design Pattern
- Event Sourcing Pattern
- Eventual Consistency Principle

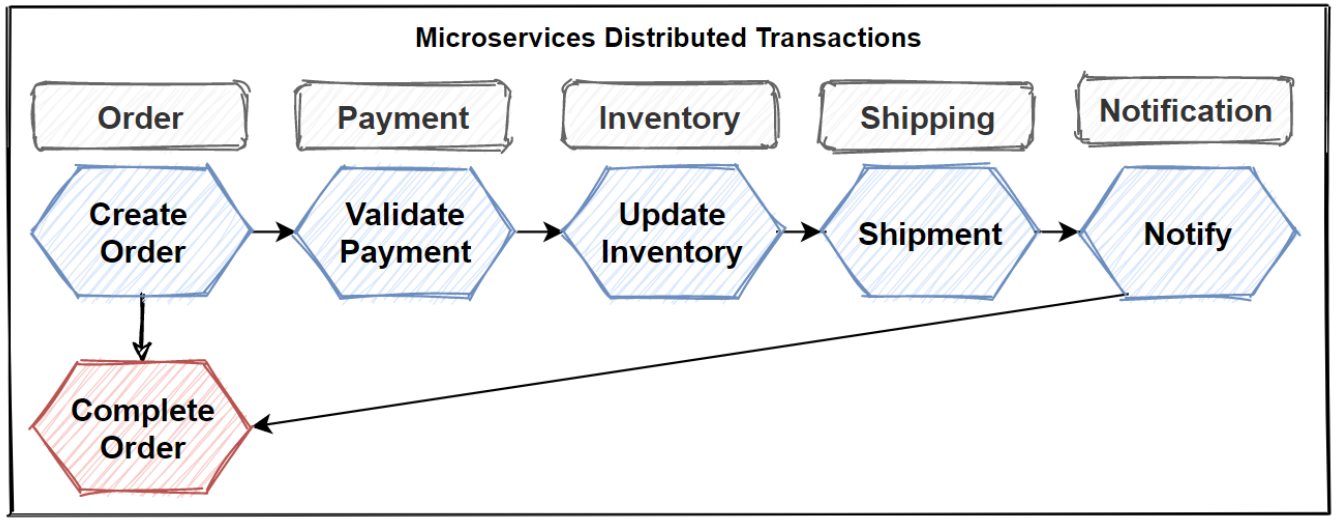


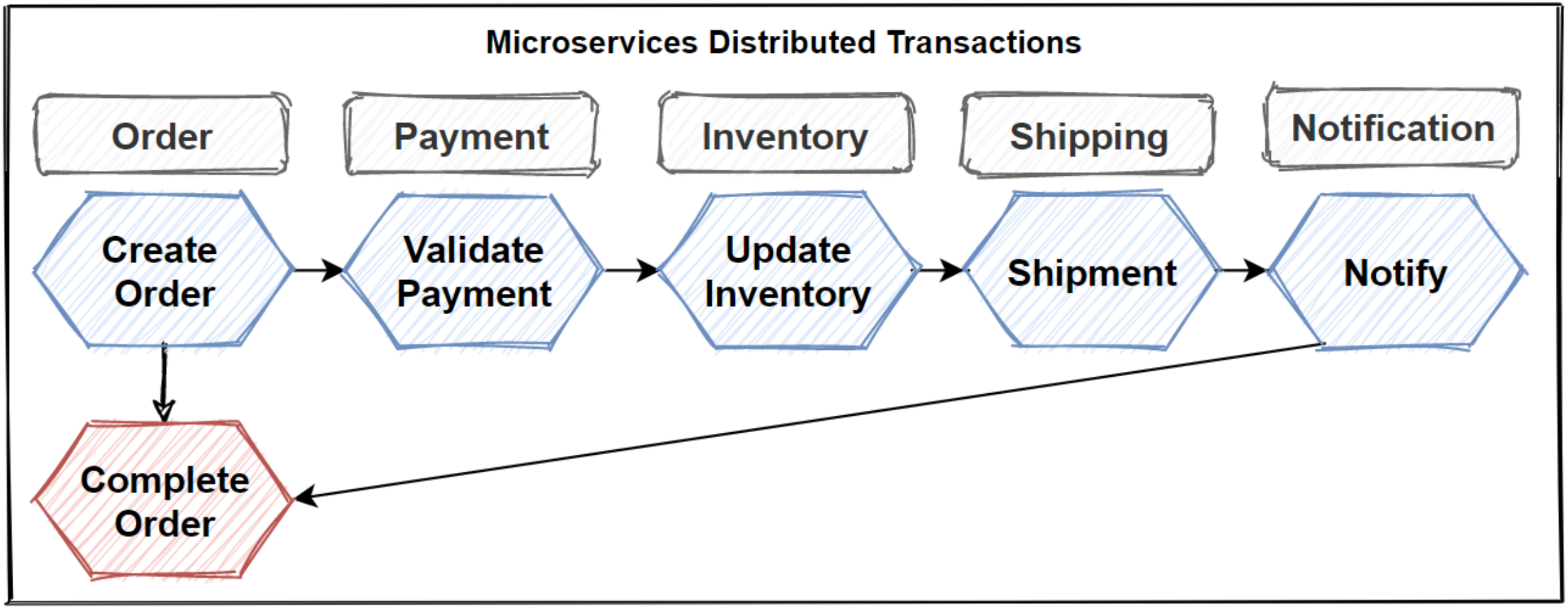
Section 15
Microservices Distributed Transactions

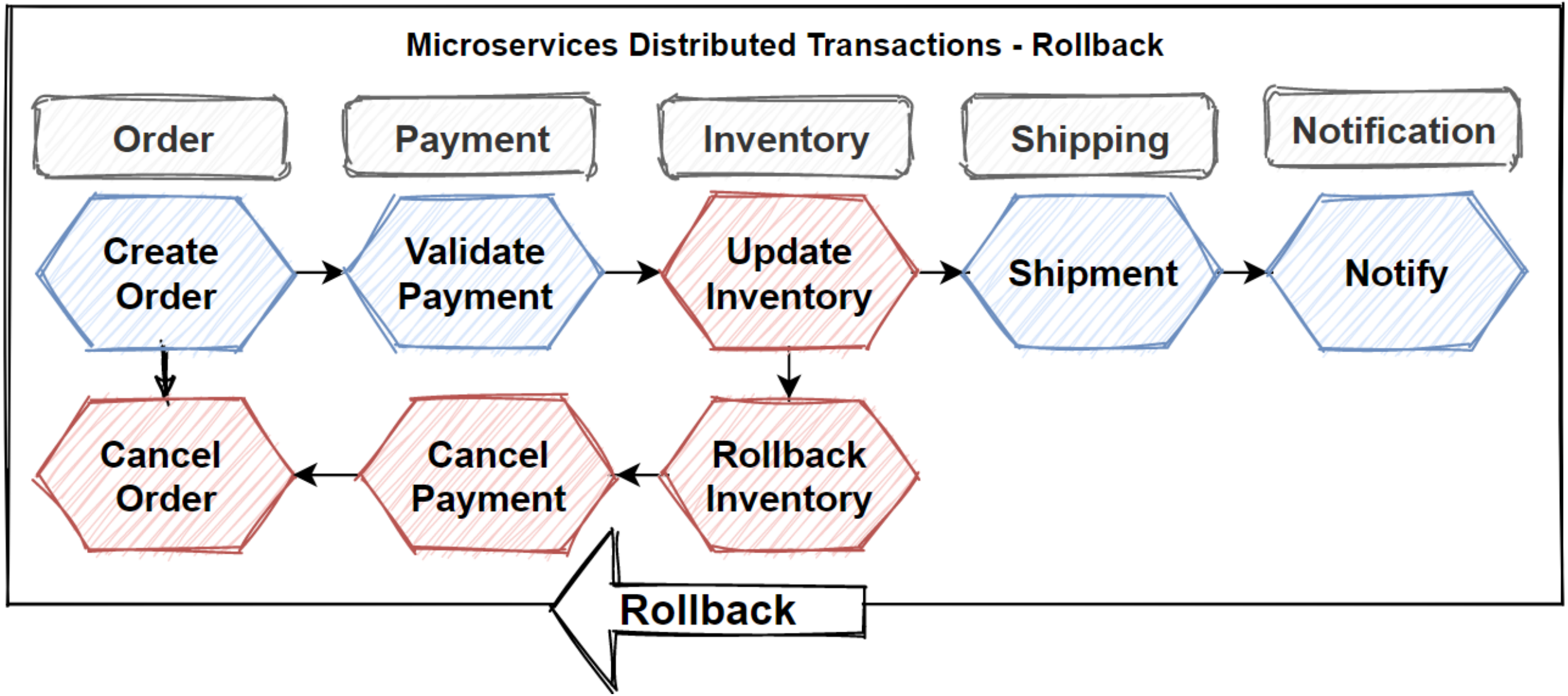
Microservices Saga and Outbox Pattern for Distributed Transactions

Microservices Distributed Transactions

- Implement transactional operations across several microservices
- Complex network concerns with polyglot database on microservices
- Distributed transaction managements on microservices implement manually

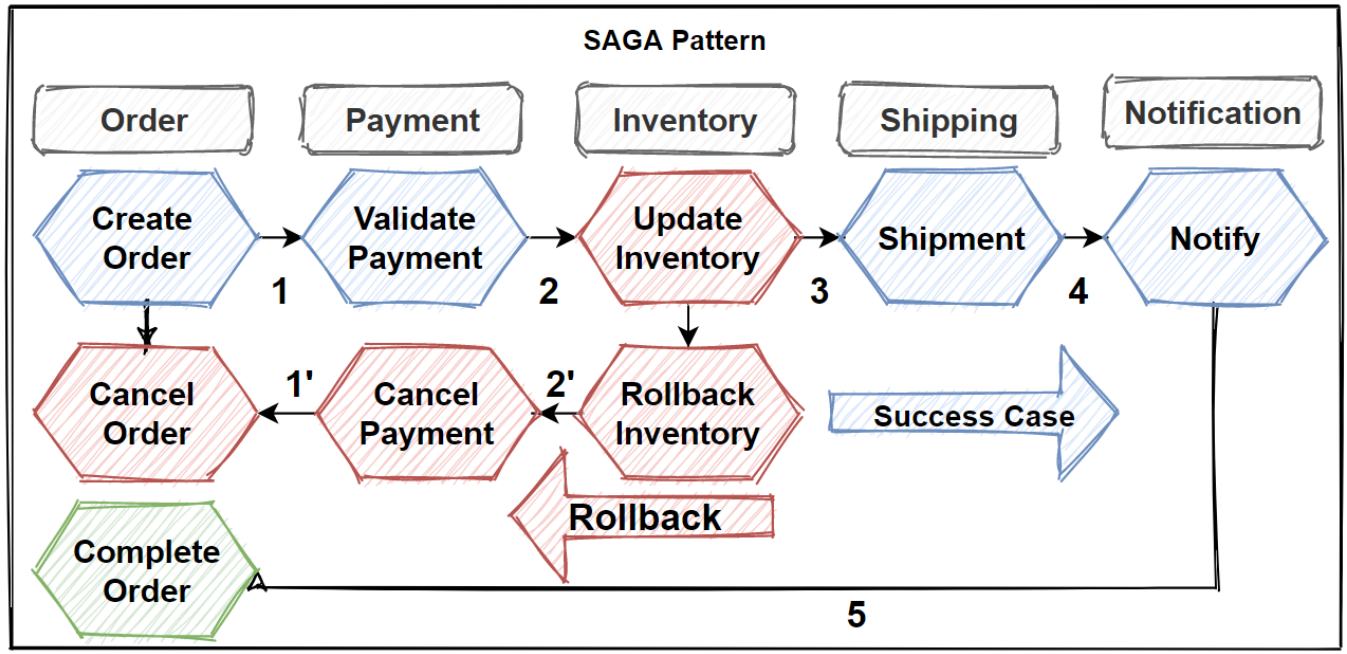



Microservices Distributed Transactions
Order
Payment
Inventory
Shipping
Notification
Create Order
Validate Payment
Update Inventory
Shipment
Notify
Complete Order


Microservices Distributed Transactions - Rollback
Order
Payment
Inventory
Shipping
Notification
Create Order
Validate Payment
Update Inventory
Shipment
Notify
Cancel Order
Cancel Payment
Rollback Inventory
Rollback

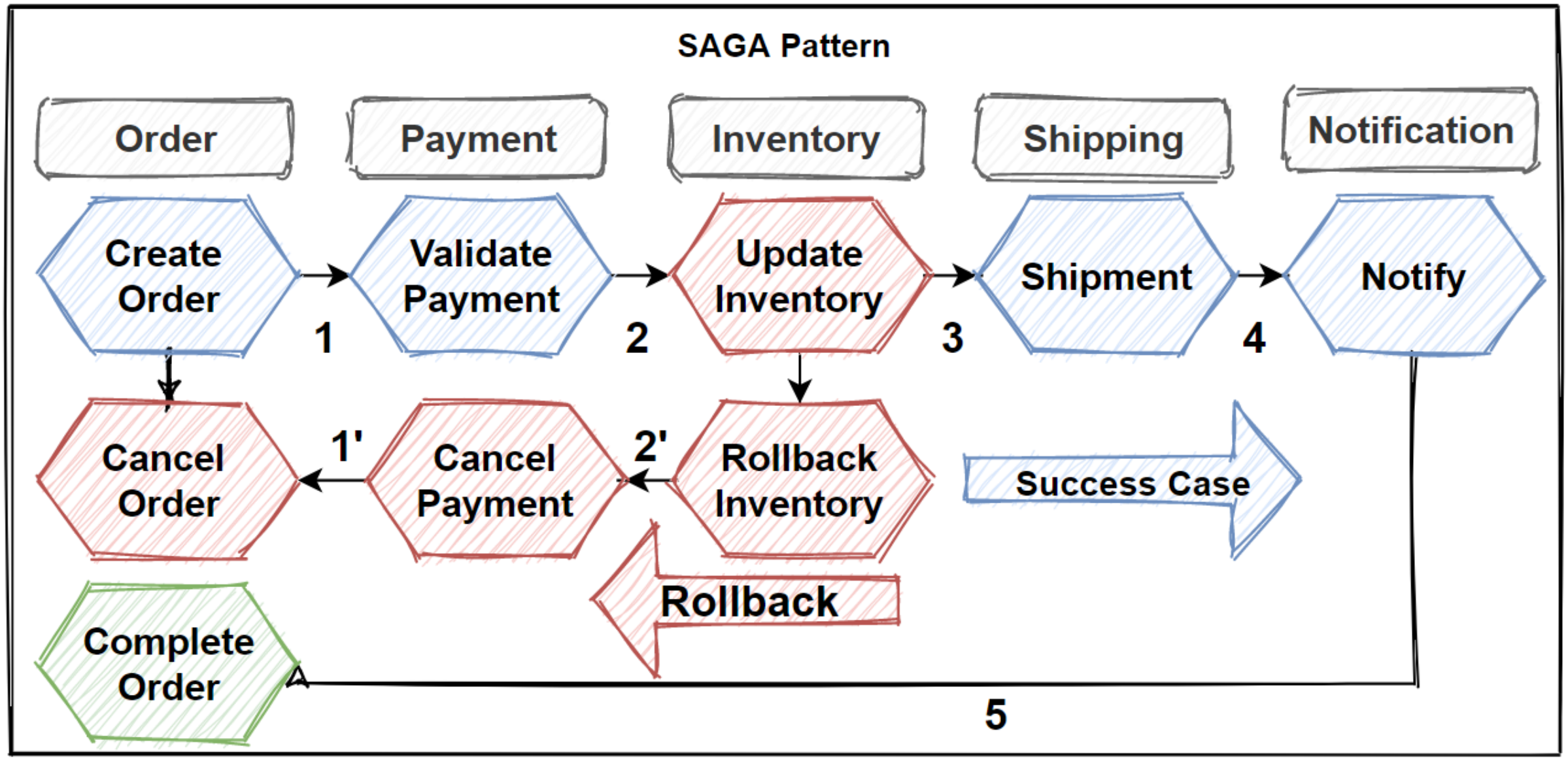
Saga Pattern for Distributed Transactions

- Manage data consistency across microservices in distributed transaction cases
- Create a set of transactions that update microservices sequentially
- Publish events to trigger the next transaction
- If failed, trigger to rollback transactions
- Grouping these local transactions and sequentially invoking one by one



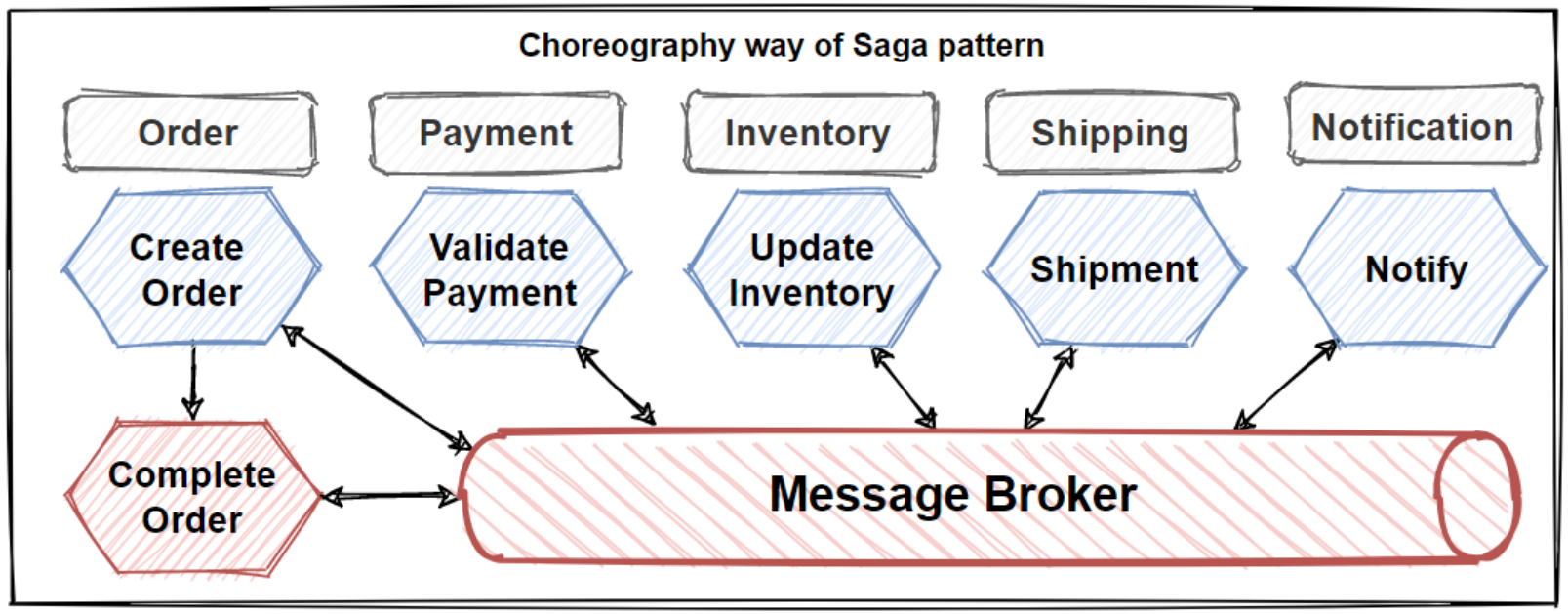
Saga implementation ways

- **Choreography** or **orchestration**


SAGA Pattern
Order
Payment
Inventory
Shipping
Notification
Create Order
Validate Payment
Update Inventory
Shipment
Notify
1
2
3
4
Cancel Order
Cancel Payment
Rollback Inventory
1'
2'
Success Case
Rollback
Complete Order
5

Choreography Saga Pattern

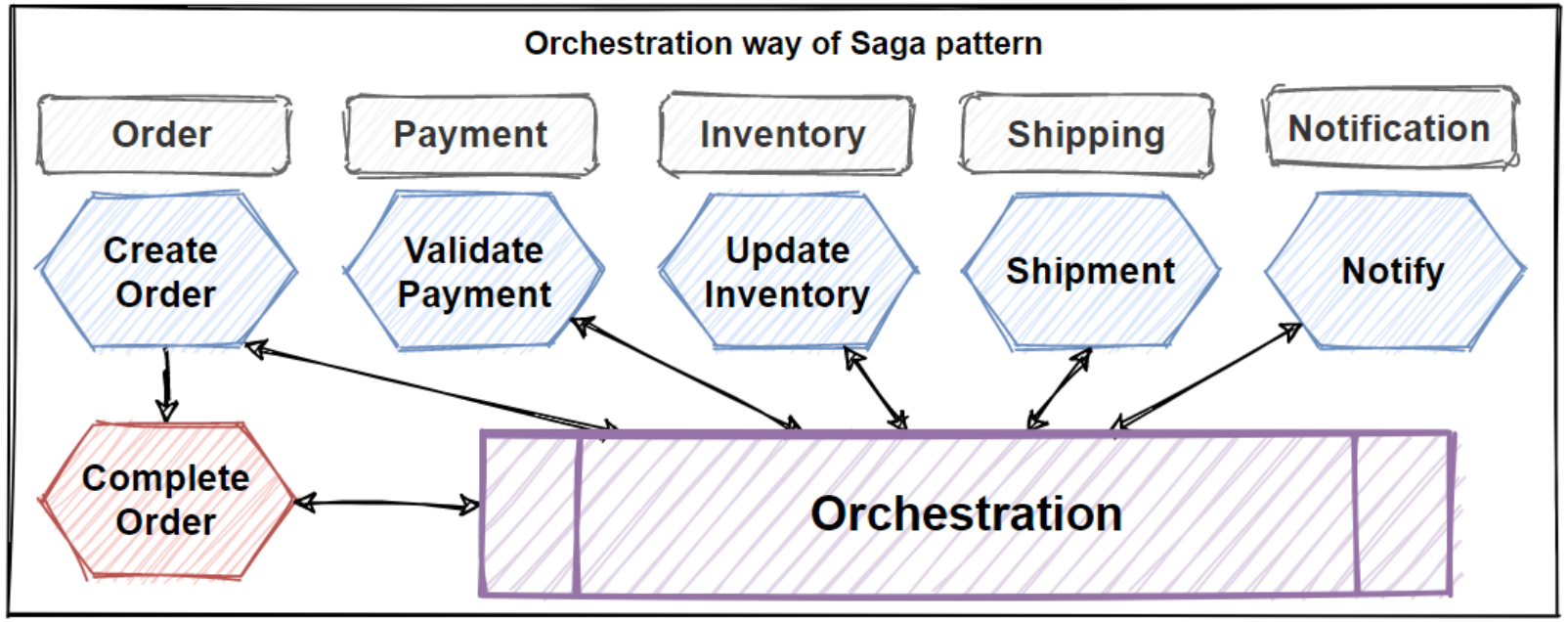
- Coordinate sagas with applying publish-subscribe principles
- Each microservices run its own local transactior
- Publish events to trigge the next transaction



- Workflow steps increase, then it can become confusing and hard to manage transaction
- Decouple direct dependency

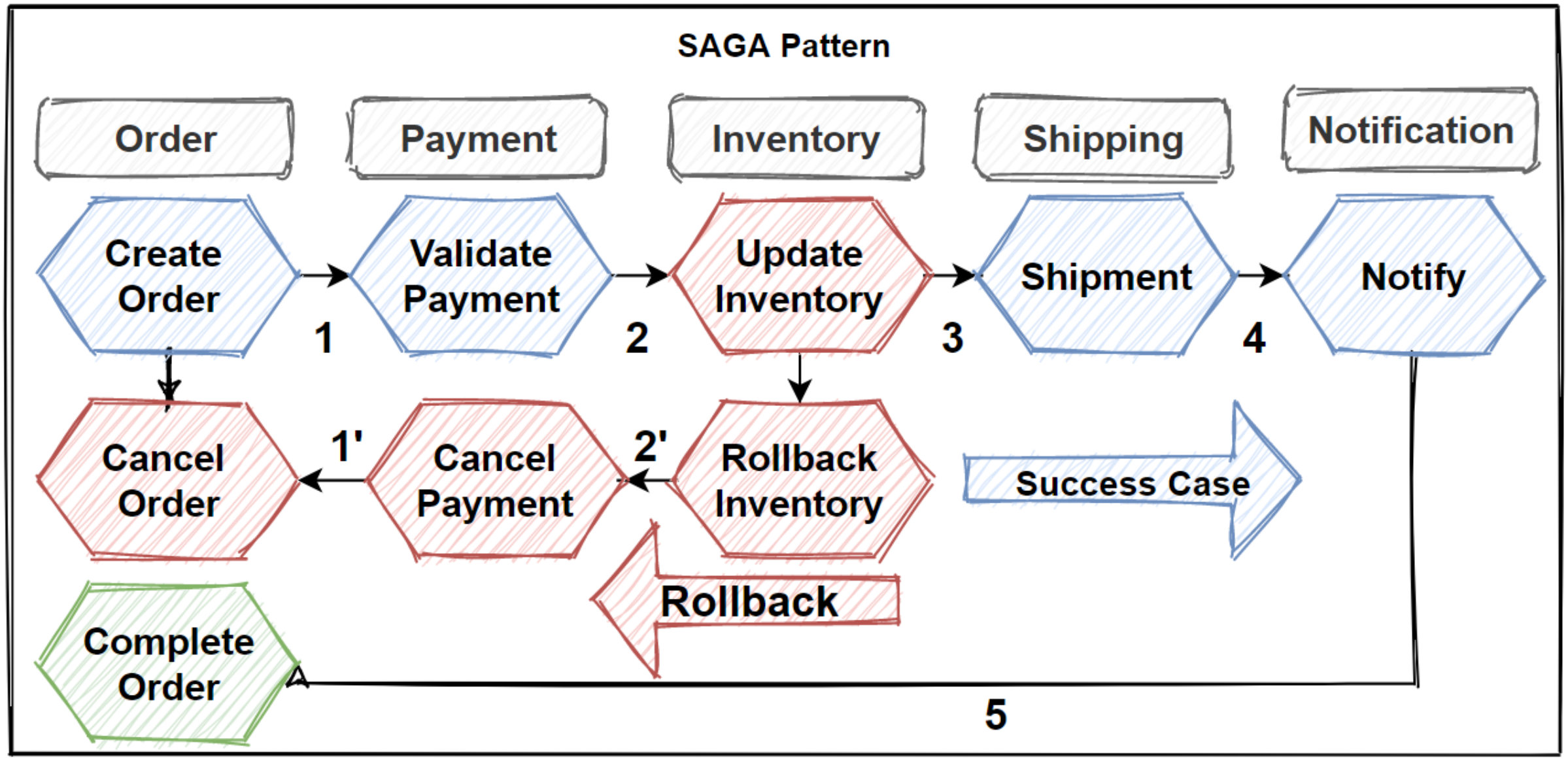
Orchestration Saga Pattern

- Coordinate sagas with a centralized controller microservice
- Invoke to execute local microservices transactions in sequentially



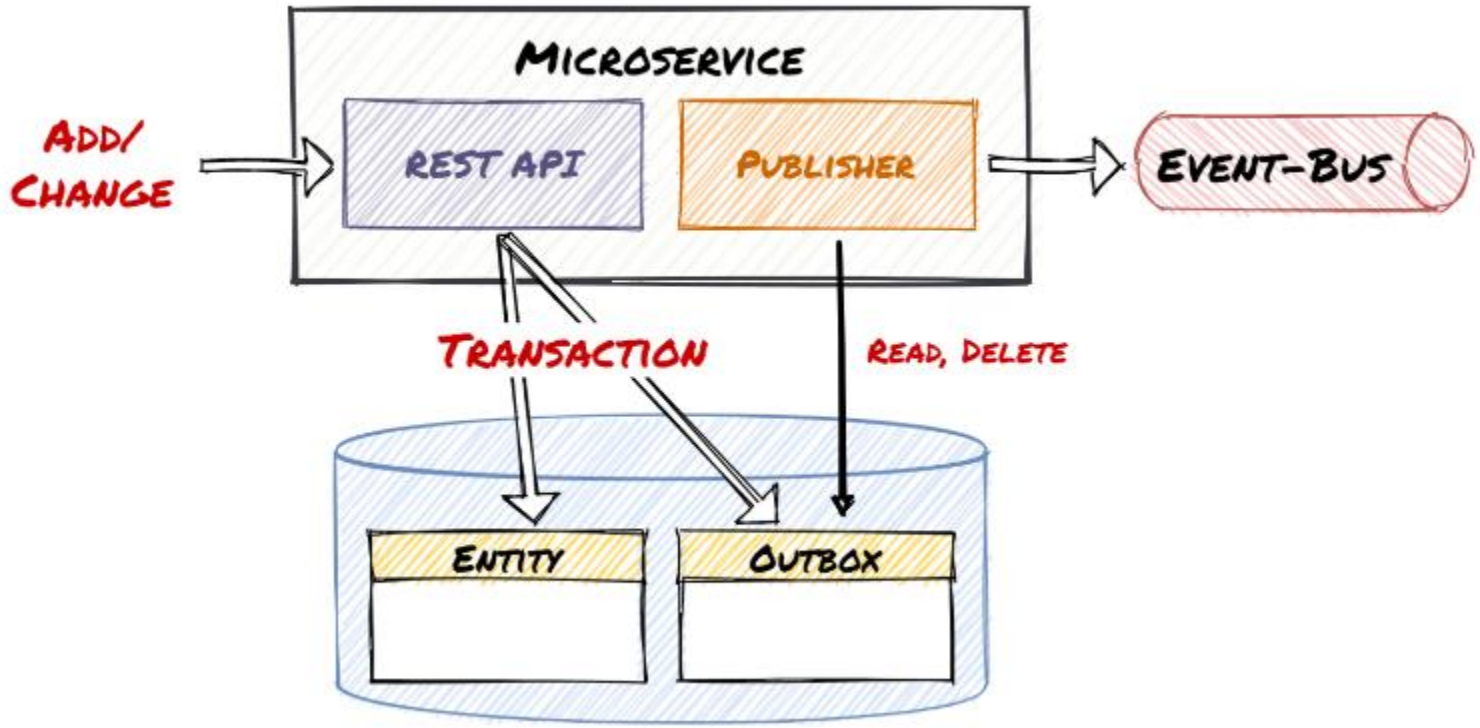
Single point-of-failure

- Execute saga transaction and manage them in centralized way and if one of the step is failed, then executes rollback steps with compensating transactions
- Good for complex workflows which includes lots of steps


SAGA Pattern
Order
Payment
Inventory
Shipping
Notification
Create Order
Validate Payment
Update Inventory
Shipment
Notify
1
2
3
4
Cancel Order
Cancel Payment
Rollback Inventory
1'
2'
Success Case
Rollback
Complete Order
5

The Outbox Pattern

- When your API publishes event messages, it doesn't directly send them
- The messages are persisted in a database table, a job publish events to message broker system
- Provides to publish events reliably with written to a table in the "outbox" role

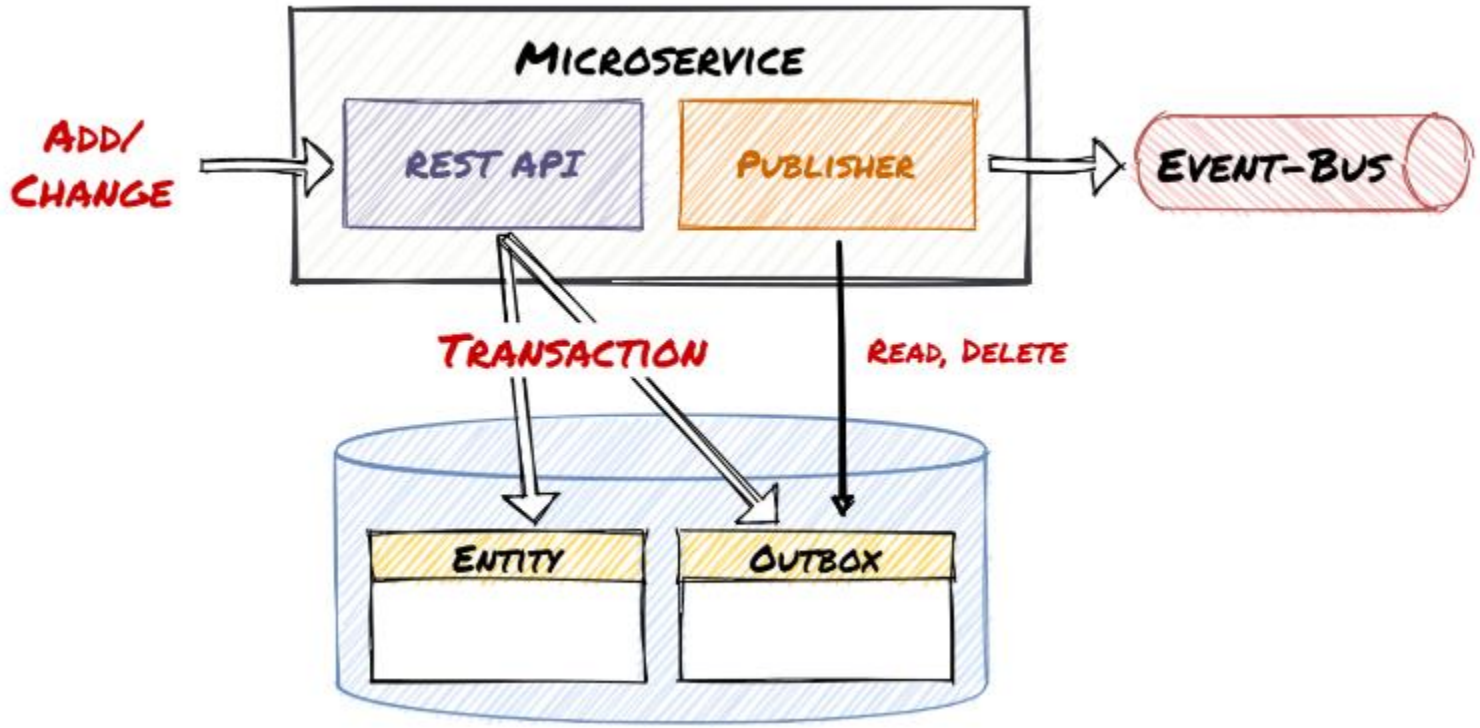


https://itnext.io/the-outbox-pattern-in-event-driven-asp-net-core-microservice-architectures-10b8d9923885

- The event and the event written to the outbox table are part of the same transaction

The Outbox Pattern

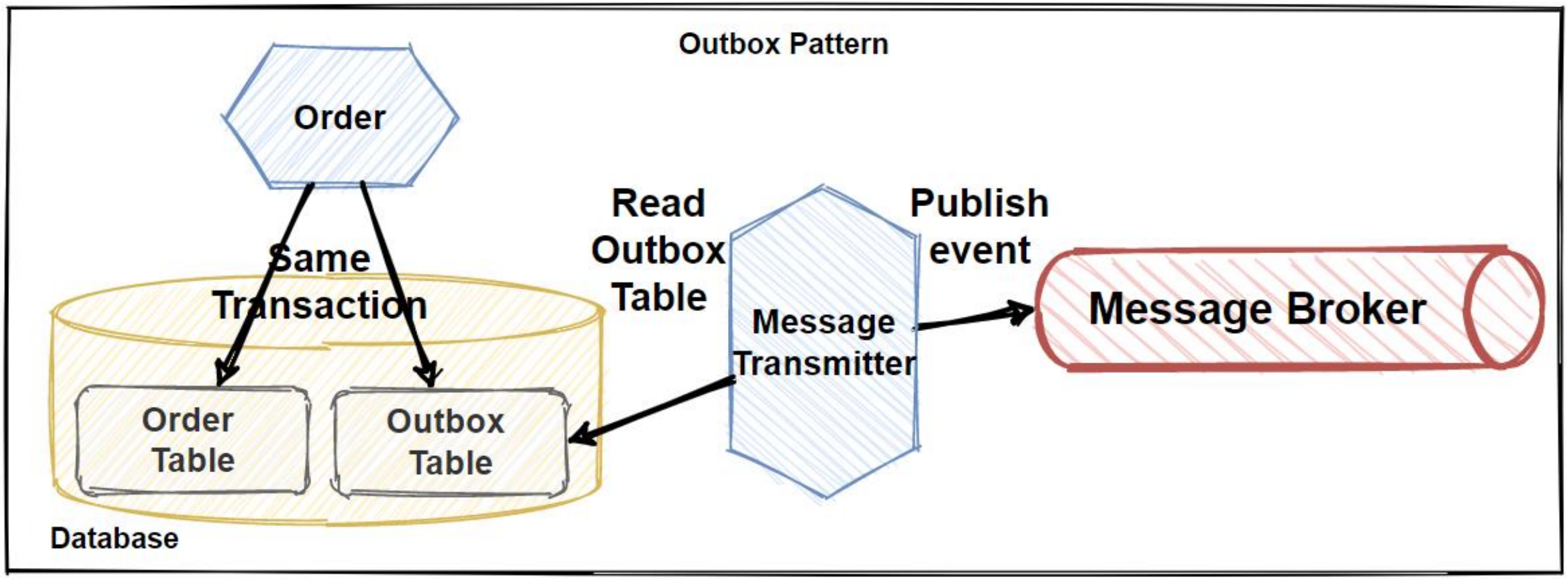
- When your API publishes event messages, it doesn't directly send them
- The messages are persisted in a database table, a job publish events to message broker system
- Provides to publish events reliably with written to a table in the "outbox" role



https://itnext.io/the-outbox-pattern-in-event-driven-asp-net-core-microservice-architectures-10b8d9923885

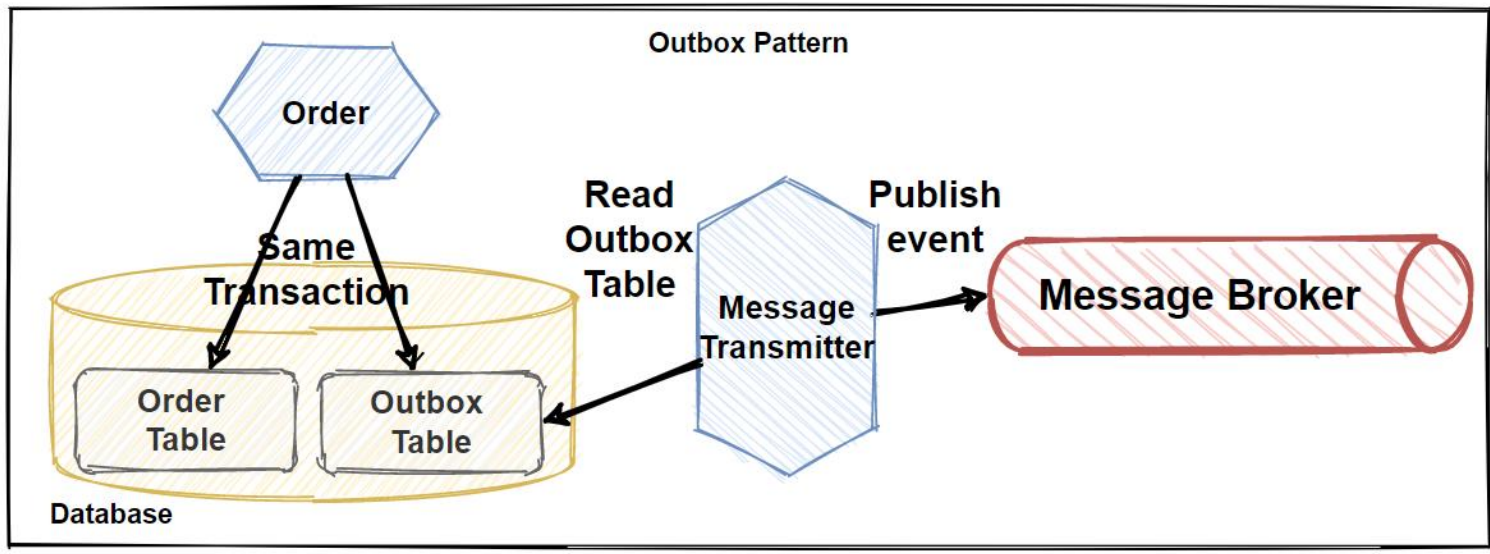
- The event and the event written to the outbox table are part of the same transaction

The Outbox Pattern



Why we use Outbox Pattern ?

- Working with critical data that need to consistent
- Need to accurate to catch all requests
- The database update and sending of the message should be atomic
- Provide data consistency
- Example - Financial business sale transactions

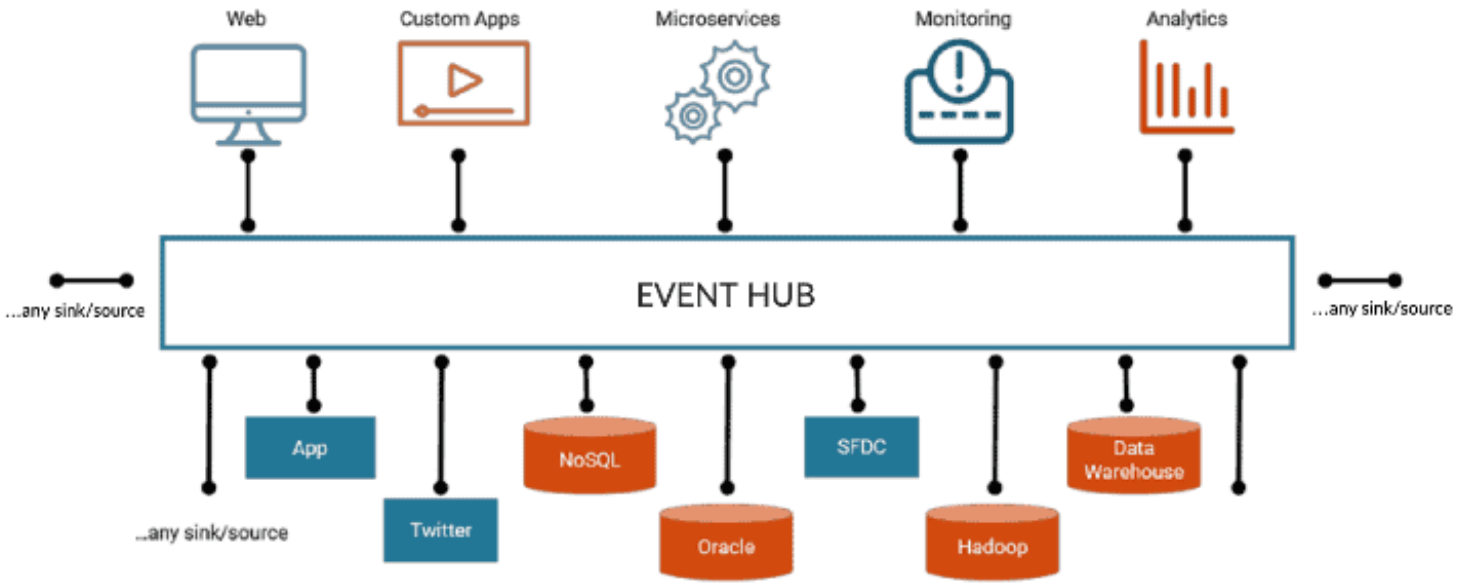


Section 16
Event-Driven Microservices Architecture

Event-Driven Microservices Architecture Patterns and Practices

Event-Driven Microservices Architecture

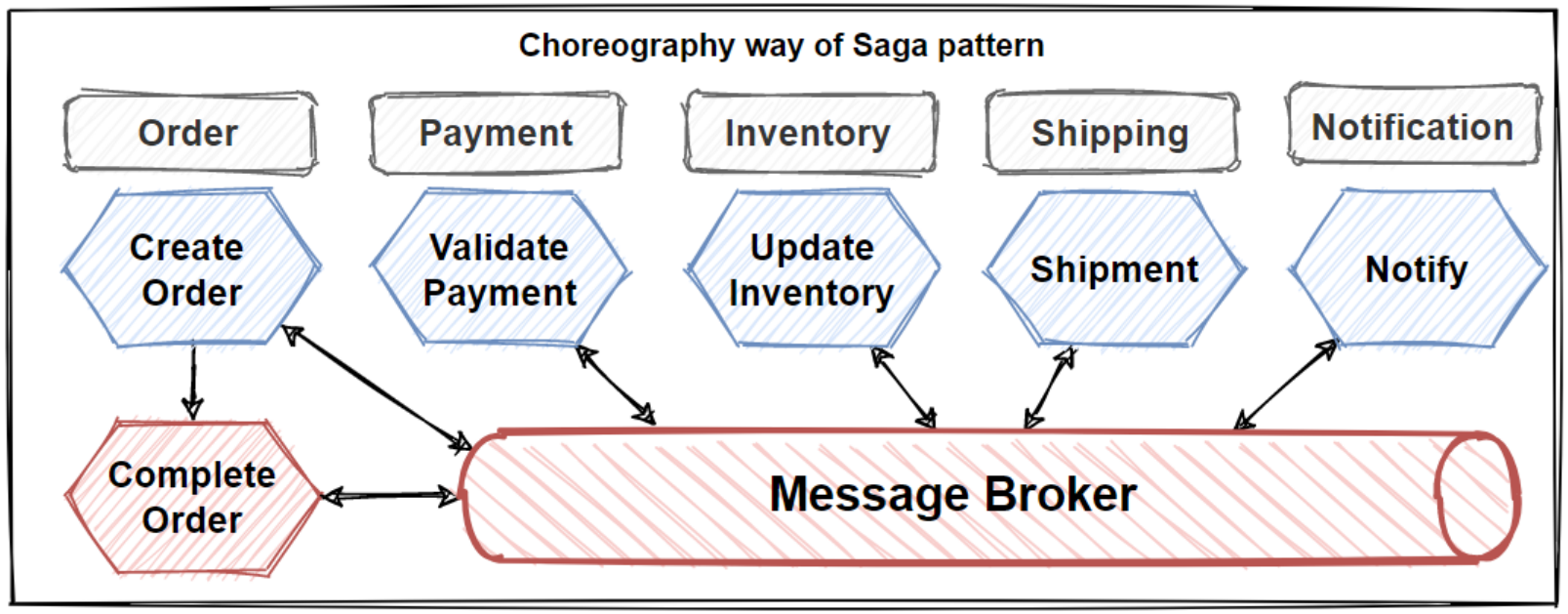
- Communicating with microservices via event messages
- Publish/subscriber pattern and Kafka message broker systems
- Asynchronous behavior and loosely coupled



- Real-time messaging platforms, stream-processing, event hubs, real-time processing, batch processing, data intelligence

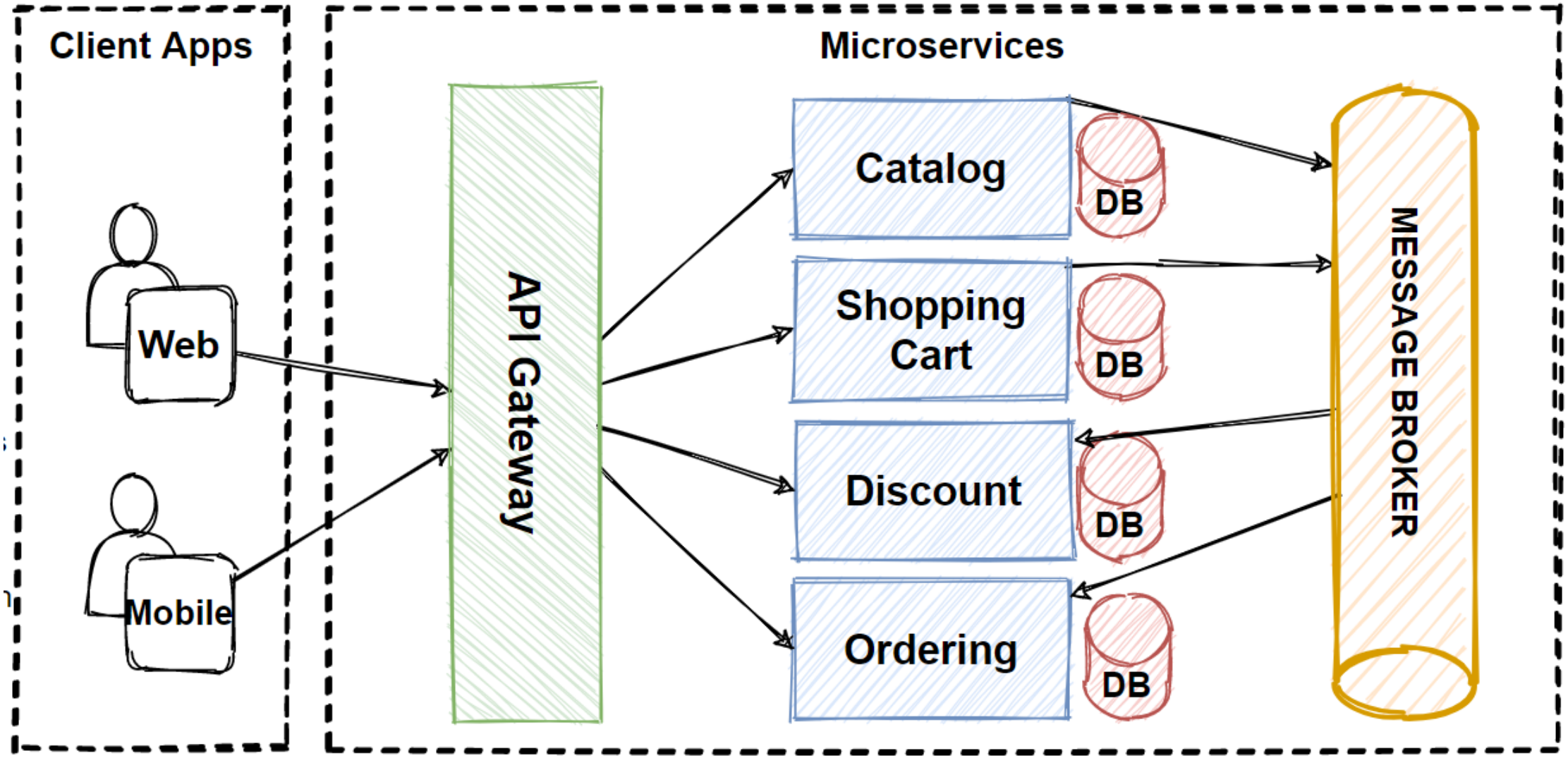
Example of Event-Driven Microservices Architecture

- E-commerce application we have microservices: customer, order, payment and products

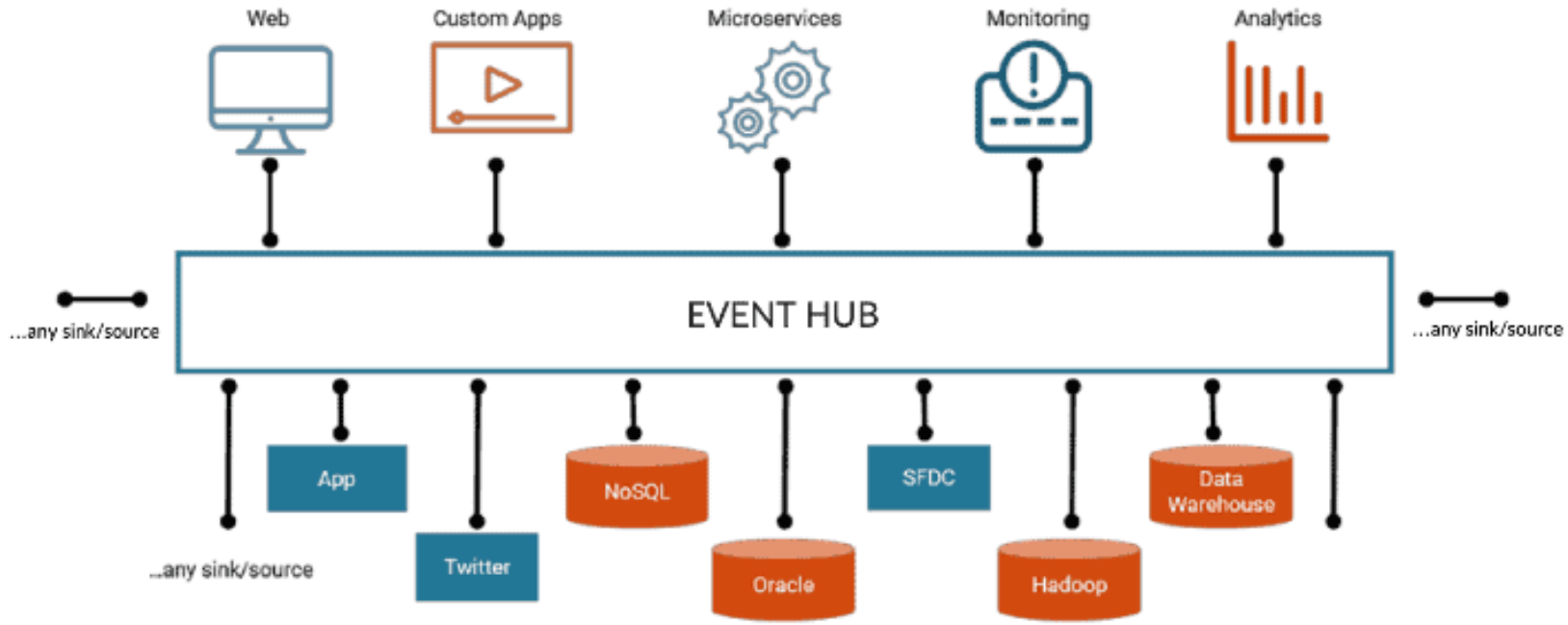


- A customer creates an order the customer and receives a payment request
- If the payment is successful, the stock is updated and the order is delivered
- If the payment is not successful, rollback the order and set order status is not completed.

Traditional Event-Driven Microservices

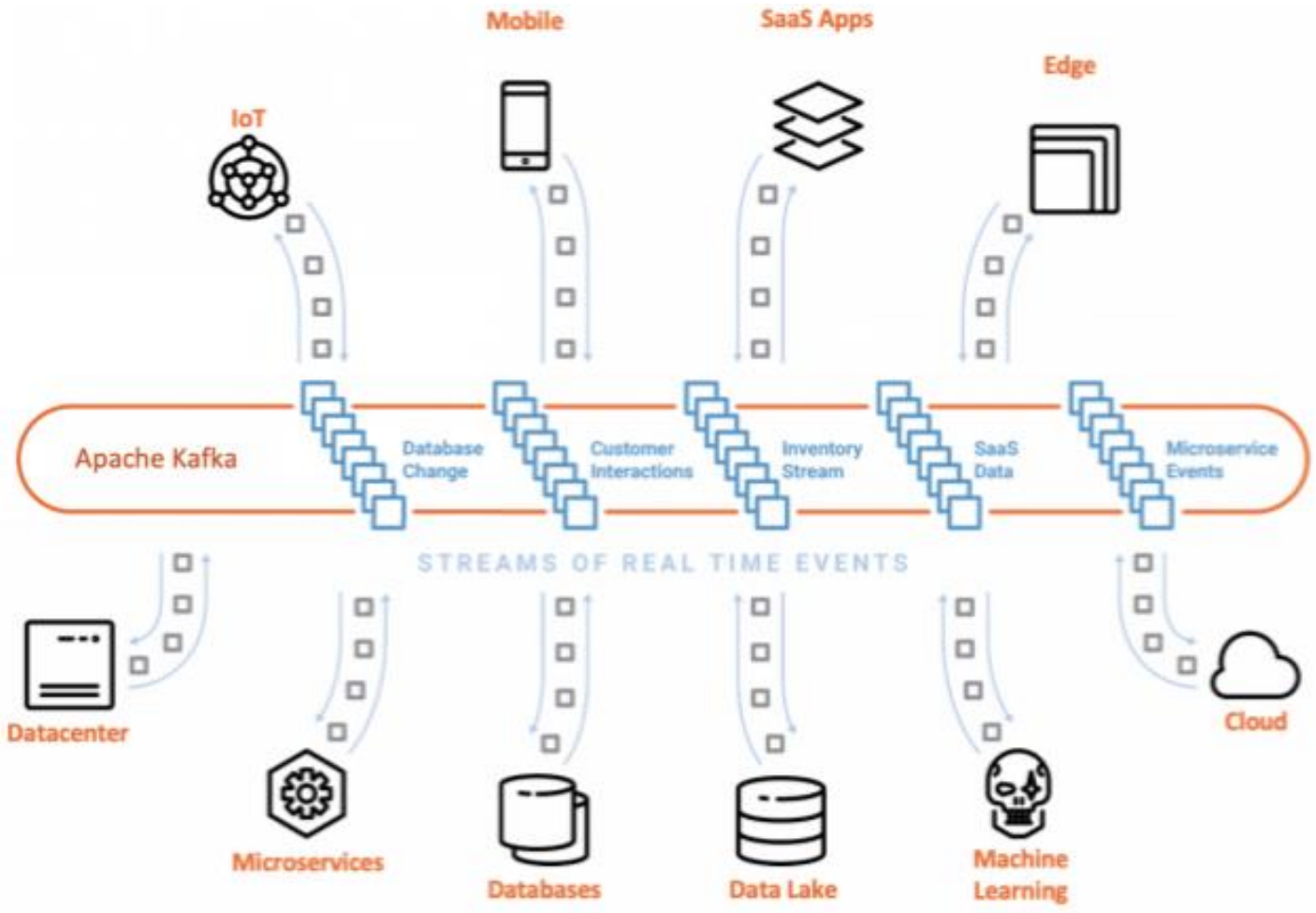


Evolved Event-Driven Microservices



- Real-time messaging platforms, stream-processing, event hubs, real-time processing, batch processing, data intelligence
- Every thing is communication via Event-Hubs
- Huge event store database that can make real-time processing

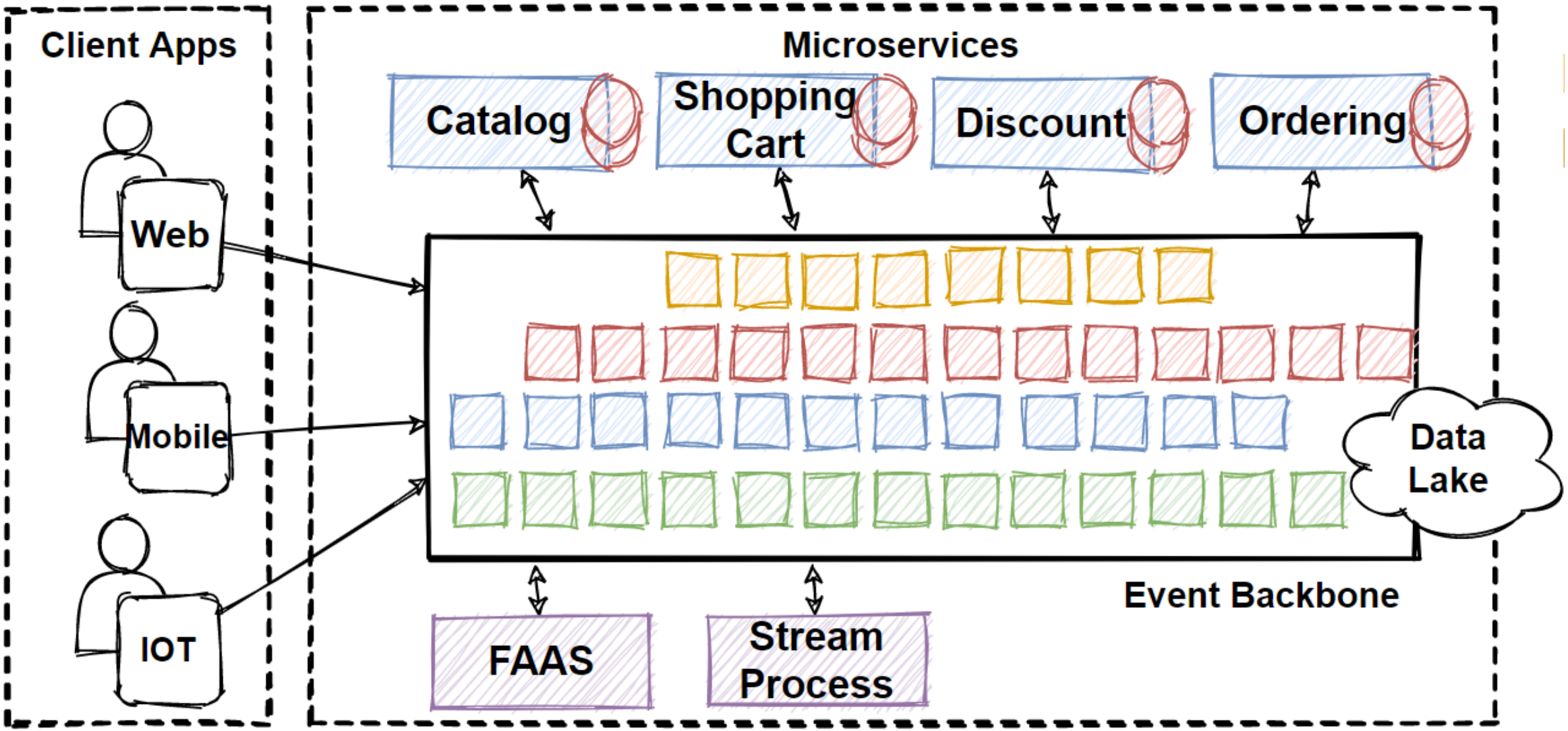
Event-Driven Microservices with Kafka



- Global-scale
- Real-time
- Persistent Storage
- Stream Processing

APACHE kafka

Event-Driven Microservices Architecture

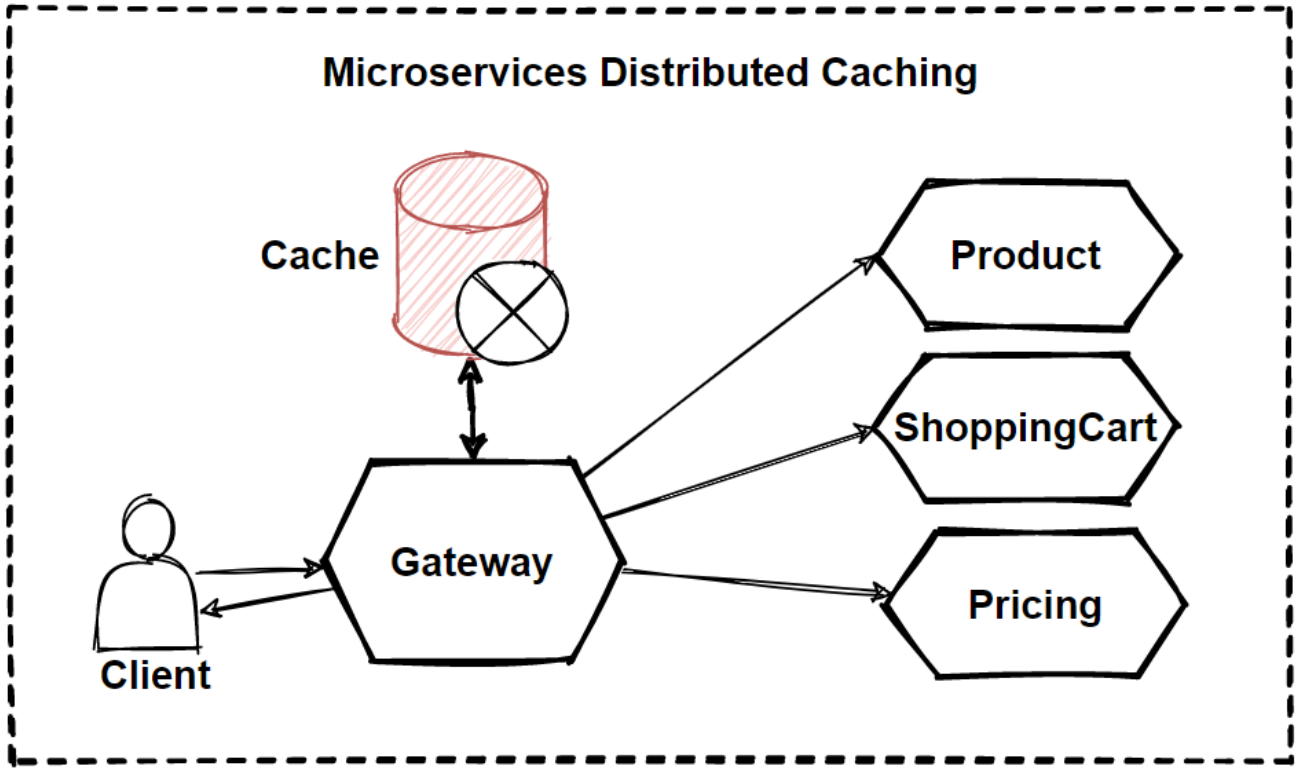


Section 17
Microservices Distributed Caching

Microservices Distributed Caching Patterns and Practices

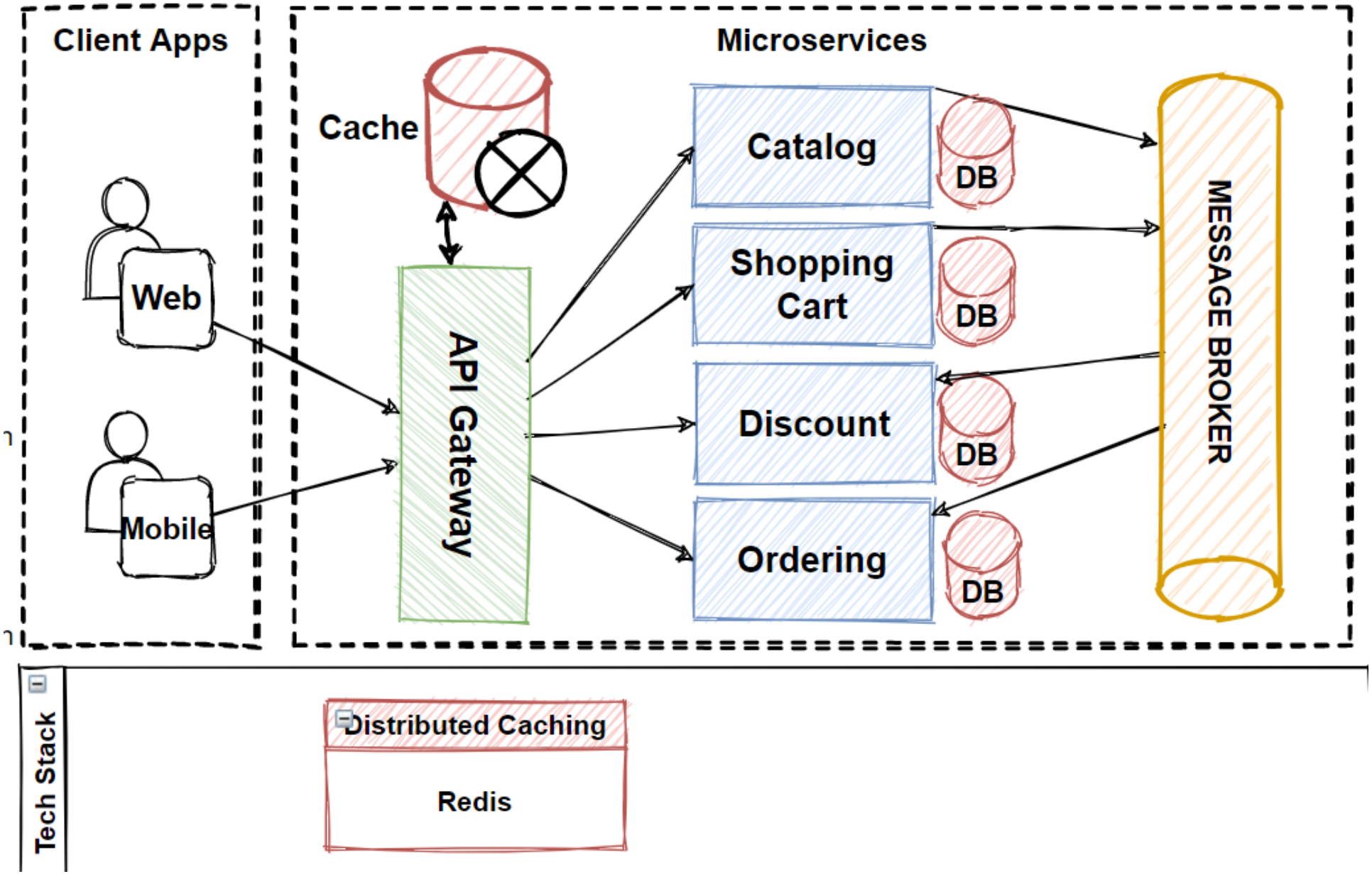
Microservices Distributed Caching

- Caching makes application faster
- Increase performance, scalability, and availability
- Reducing latency with cache when reading data
- Avoid re-calculation processes



- The distributed cache increases system responsiveness by returning cached data
- Separating the cache server scale independently

Microservices Distributed Caching

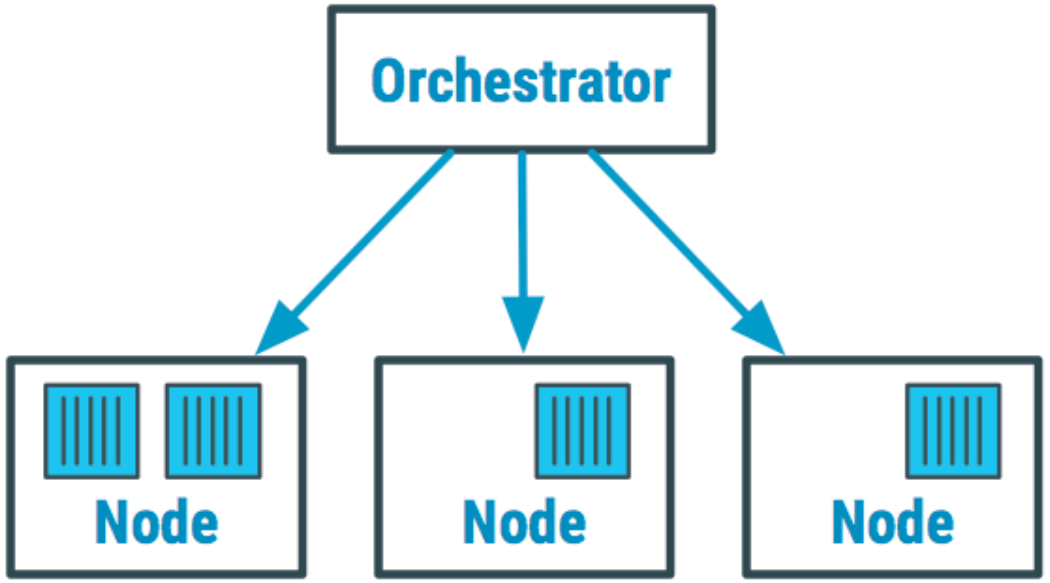


Section 18
Microservices Deployments with Containers and Orchestrators

Microservices Containers and Orchestrators Patterns and Practices

Microservices Containers and Orchestrators

- Monolithic deployment problems; single unit of deploy whole application
- Containers provide to decouple applications with all dependencies and isolate applications
- With containers, microservices can deployed separately in a container

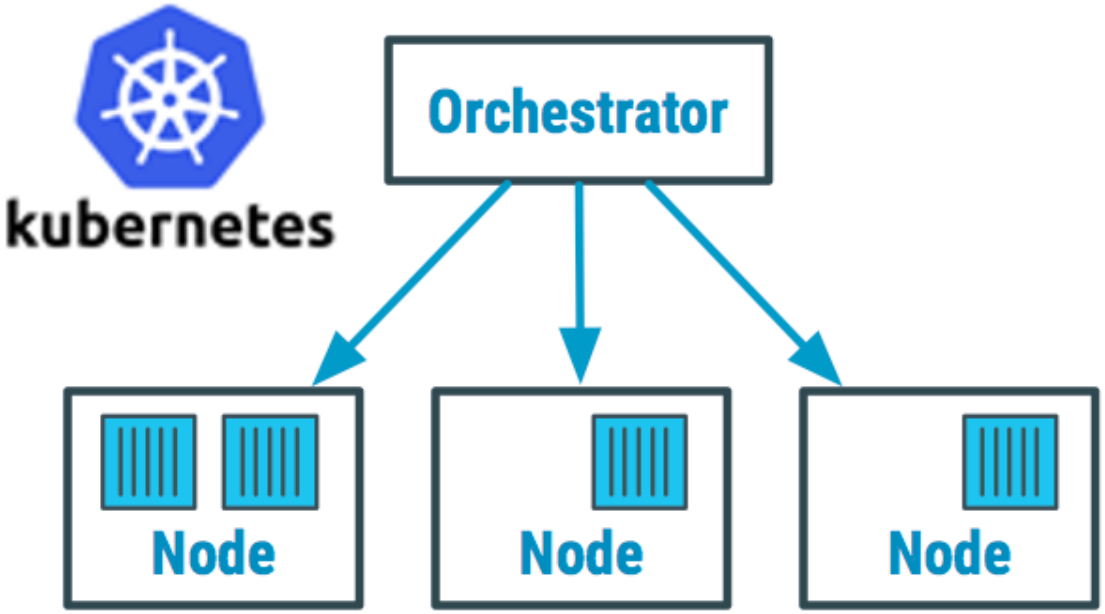


- New features can be applied and rollback very easy with container deployments.

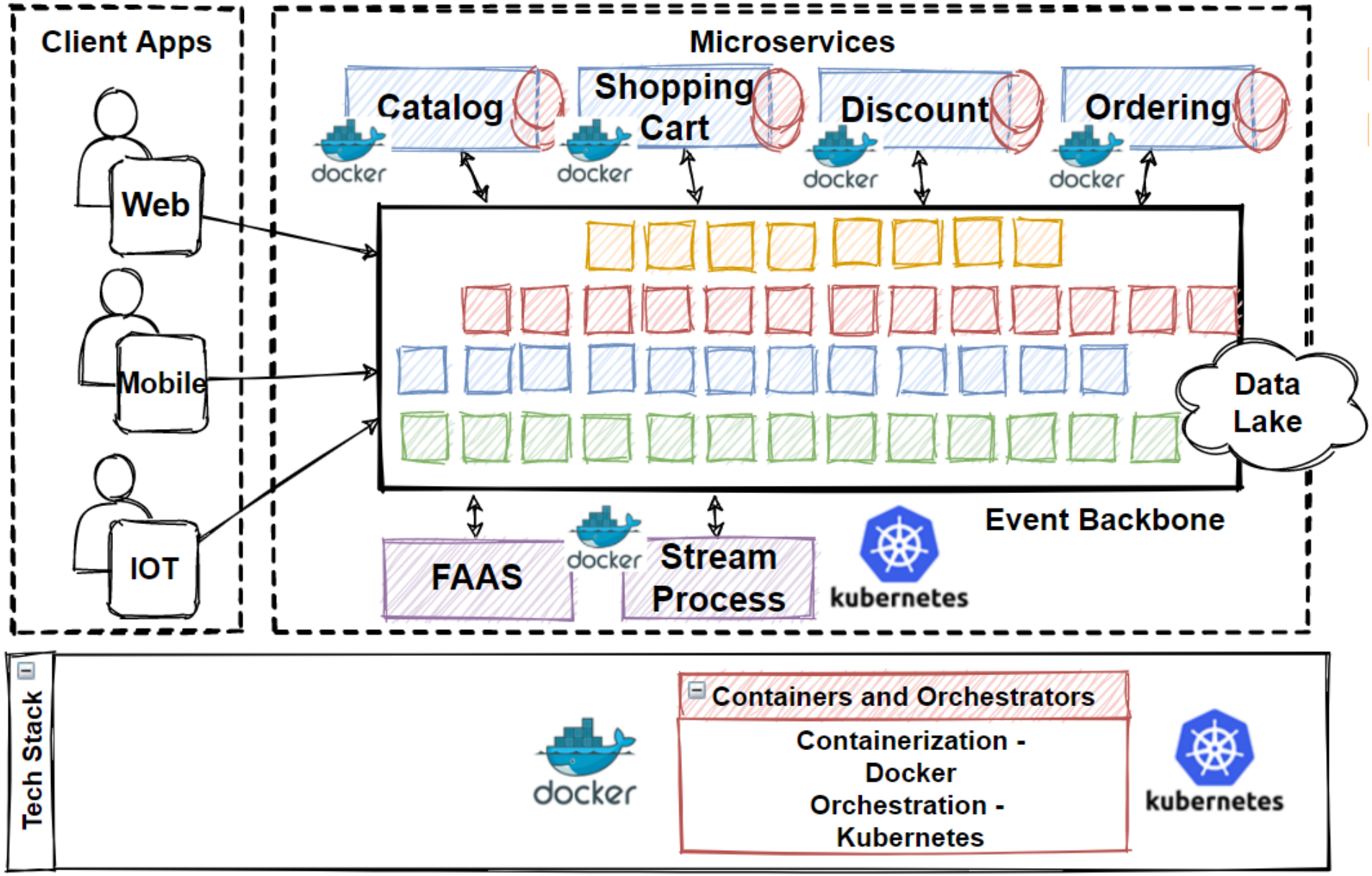
- Docker is defacto standard for containerization of microservices

Microservices Orchestrators

- Containers need to orchestrate in order to manage lots of container in your application cluster
- Orchestrators automates the deployment, scaling, and operational concerns of containerized workloads across clusters
- Kubernetes is defacto standard for orchestration of microservices.



Microservices Containers and Orchestrators



Scalability - How many concurrent request can accommodate our design ?

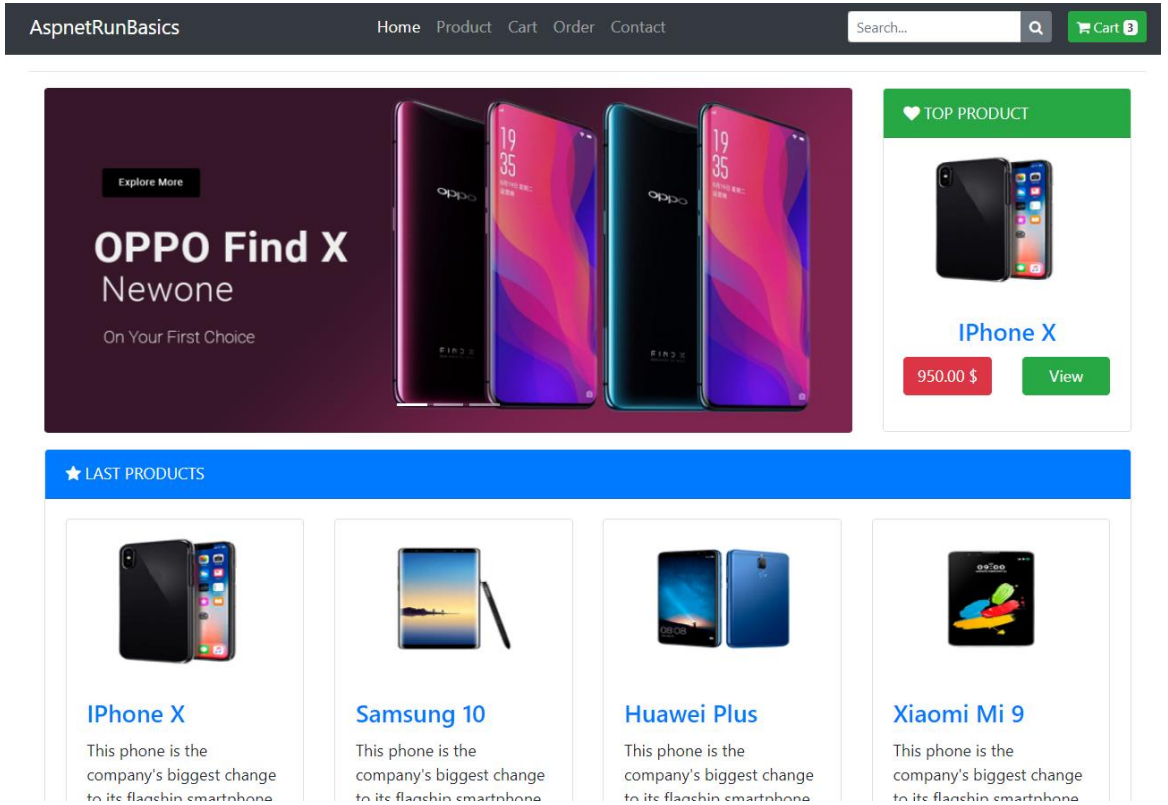
Concurrent Users	Requests/second	Latency (Expected)
2K	0.5K	
20K	12K	
100K	80K	
500K	300K	<= 2 sec

Section 19
Implementation of Microservices Architecture

Code Reviews of existing Microservice Architecture on GitHub

Run Final Application

- Clone Microservices Repository
- Docker-compose up
- Follow steps on github documentation
- Run the Project Section
- DEMO



https://github.com/aspnetrun/run-aspnetcore-microservices

aspnetrun

The best path to leverage your aspnet skills. Onboarding to .Net Software Architect jobs. Download latest real world asp.net core microservices applications.

Istanbul https://aspnetrun.azurewebsites.net ezozkme@gmail.com

Repositories 13 | Packages | People 1 | Teams | Projects 1 | Settings

Pinned repositories

Customize pinned repositories

learn

The best path to .Net Microservices Udemy Learning Path. .Net world evolving to the microservices and Cloud-native systems to provide rapid change, large scale, and resilience cutting-edge systems....

2 1

run-aspnetcore-microservices

Microservices on .Net platforms which used Asp.Net Web API, Docker, RabbitMQ, MassTransit, Grpc, Ocelot API Gateway, MongoDB, Redis, PostgreSQL, SqlServer, Dapper, Entity Framework Core, CQRS and C...

C# 420 175

run-aspnetcore

Template

A starter kit for your next ASP.NET Core web application. Boilerplate for ASP.NET Core reference application, demonstrating a layered application architecture with applying Clean Architecture and D...

C# 228 57

run-aspnet-identityserver4

Secure microservices with using standalone Identity Server 4 and backing with Ocelot API Gateway. Protect our ASP.NET Web MVC and API applications with using OAuth 2 and OpenID Connect in IdentityS...

C# 39 18

run-aspnet-grpc

Using gRPC in Microservices for Building a high-performance Interservice Communication with .Net 5. See gRPC Microservices and Step by Step Implementation on .NET Course w/ discount->

C# 34 10

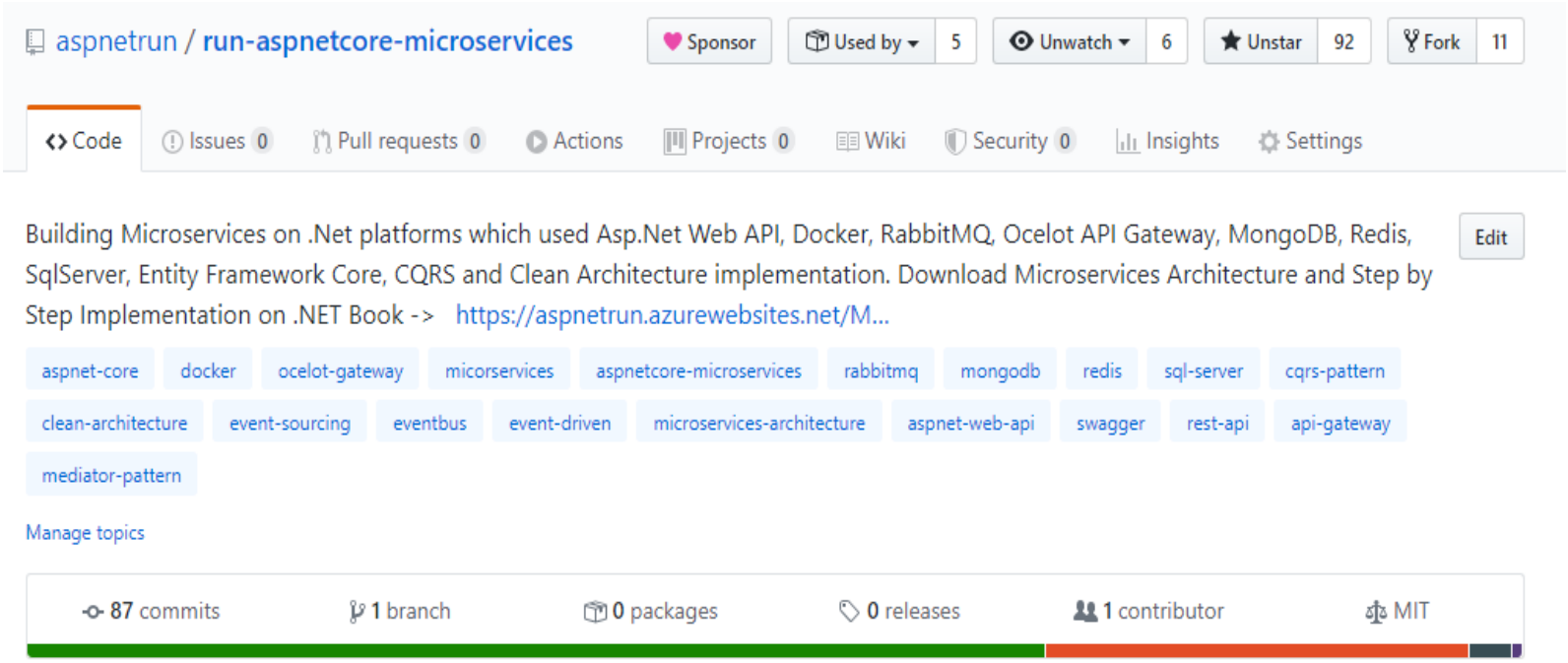
run-devops

Deploying .Net Microservices into Kubernetes, and moving deployments to the cloud Azure Kubernetes Services (AKS) with using Azure Container Registry (ACR) and how to Automating Deployments with Az...

C# 4 8

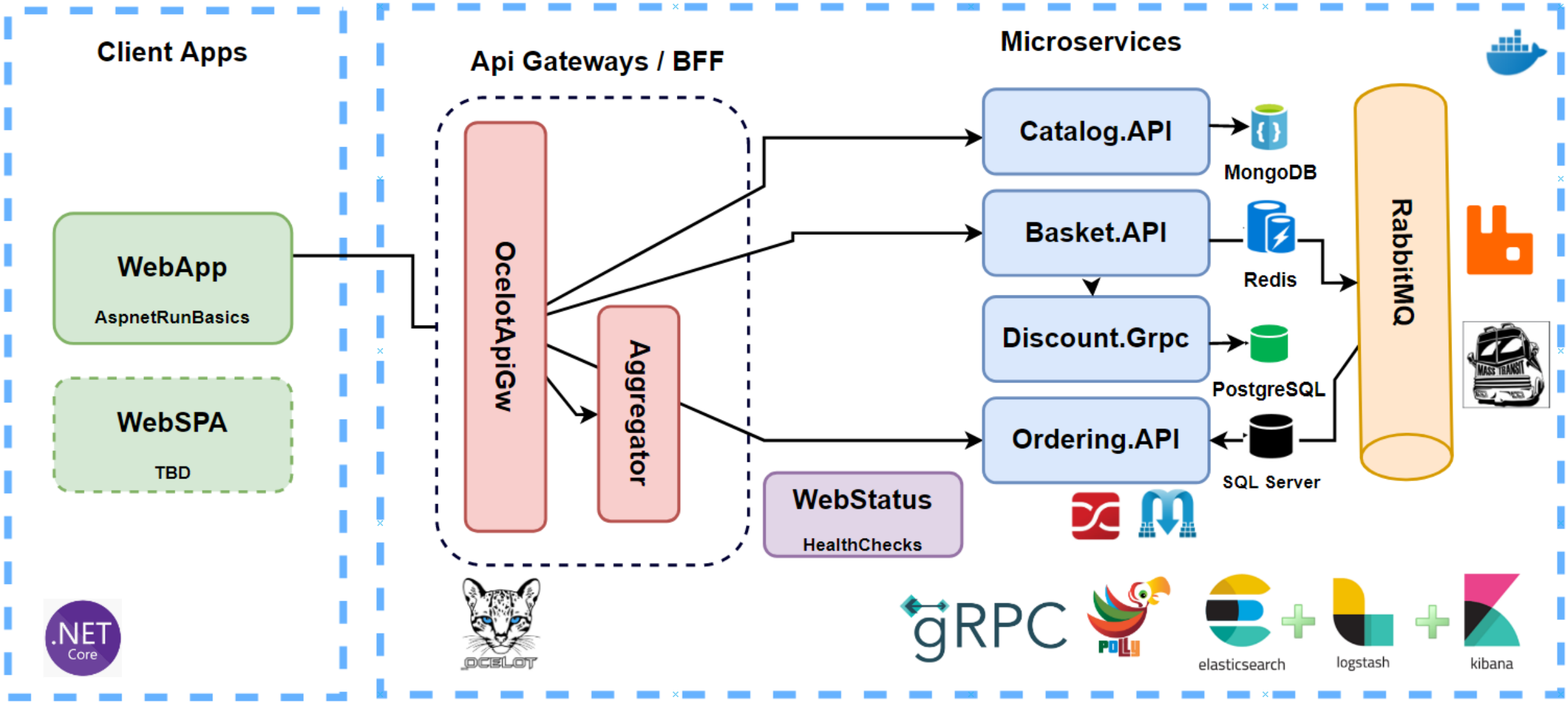
Source Code on Github

- Source code on Github
- Fork the repository
- Open issues
- Send Pull Requests

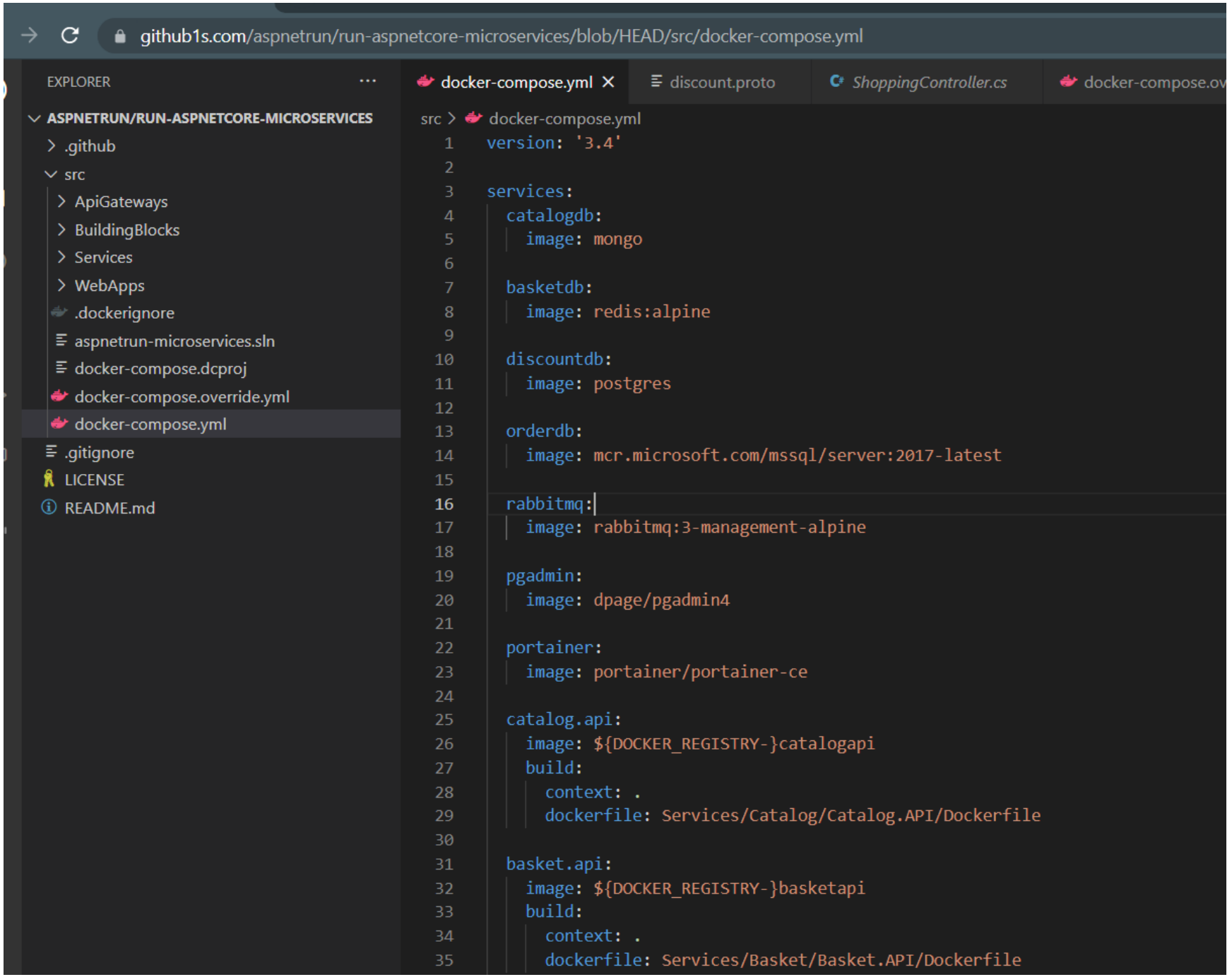


https://github.com/aspnetrun/run-aspnetcore-microservices

Big Picture


Client Apps
WebApp
AspnetRunBasics
WebSPA
TBD
Api Gateways / BFF
OcelotApiGw
Aggregator
WebStatus
HealthChecks
Microservices
Catalog.API
MongoDB
Basket.API
Redis
Discount.Grpc
PostgreSQL
Ordering.API
SQL Server
RabbitMQ
.NET Core
gRPC
elasticsearch
logstash
kibana

Code Review


github1s.com/aspnetrun/run-aspnetcore-microservices/blob/HEAD/src/docker-compose.yml
EXPLORER
ASPNETRUN/RUN-ASPNETCORE-MICROSERVICES
.github
src
ApiGateways
BuildingBlocks
Services
WebApps
.dockerignore
aspnetrun-microservices.sln
docker-compose.dcproj
docker-compose.override.yml
docker-compose.yml
.gitignore
LICENSE
README.md
docker-compose.yml
discount.proto
ShoppingController.cs
src > docker-compose.yml
version: '3.4'
services:
catalogdb:
image: mongo
basketdb:
image: redis:alpine
discountdb:
image: postgres
orderdb:
image: mcr.microsoft.com/mssql/server:2017-latest
rabbitmq:
image: rabbitmq:3-management-alpine
pgadmin:
image: dpage/pgadmin4
portainer:
image: portainer/portainer-ce
catalog.api:
image: ${DOCKER_REGISTRY-}catalogapi
build:
context: .
dockerfile: Services/Catalog/Catalog.API/Dockerfile
basket.api:
image: ${DOCKER_REGISTRY-}basketapi
build:
context: .
dockerfile: Services/Basket/Basket.API/Dockerfile

Thanks!

Follow me on github
https://github.com/mehmetozkaya
https://github.com/aspnetrun

Follow me on twitter
https://twitter.com/ezozkme

Follow me on medium
https://mehmetozkaya.medium.com/

Send mail me anything you can ask
ezozkme@gmail.com

Check my other courses on udemy:
https://www.udemy.com/user/ff0e5c8c-dd71-443e-be0a-e73ba821f7d7/