

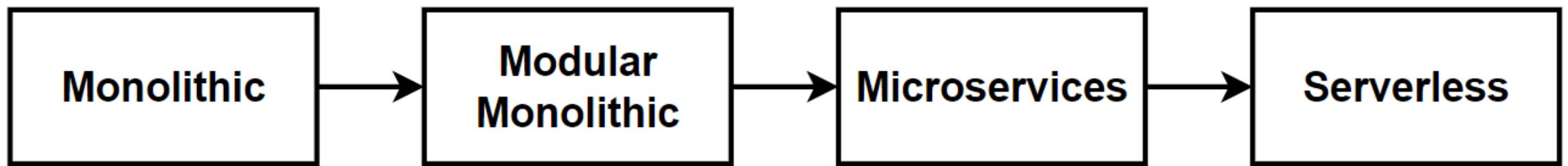
Design Microservices Architecture with Patterns & Best Practices

A step-by-step process for software system design and evolve from monolithic to microservices following the patterns & principles.

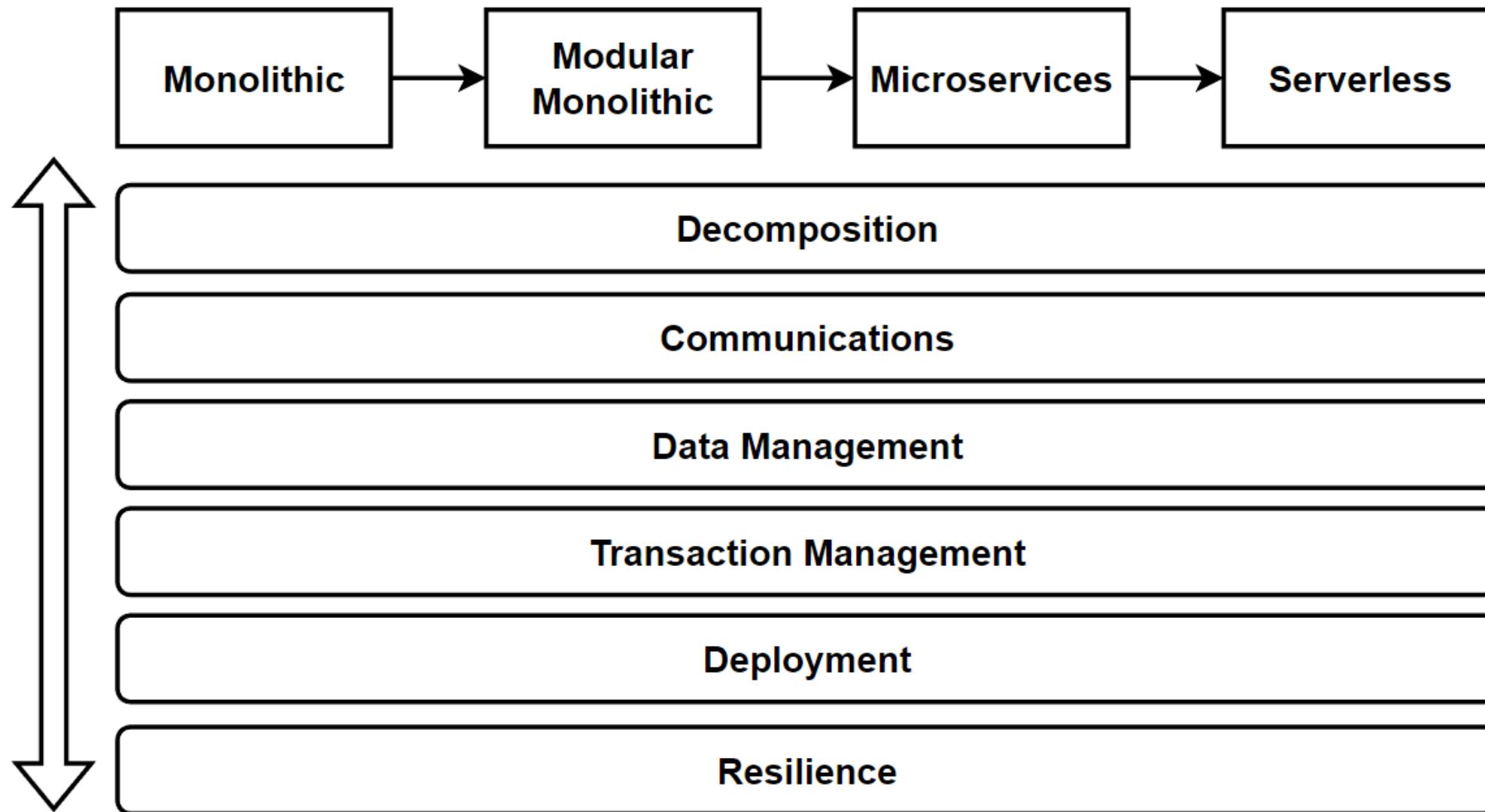
Refactor architectures with different aspects of microservices pillars.



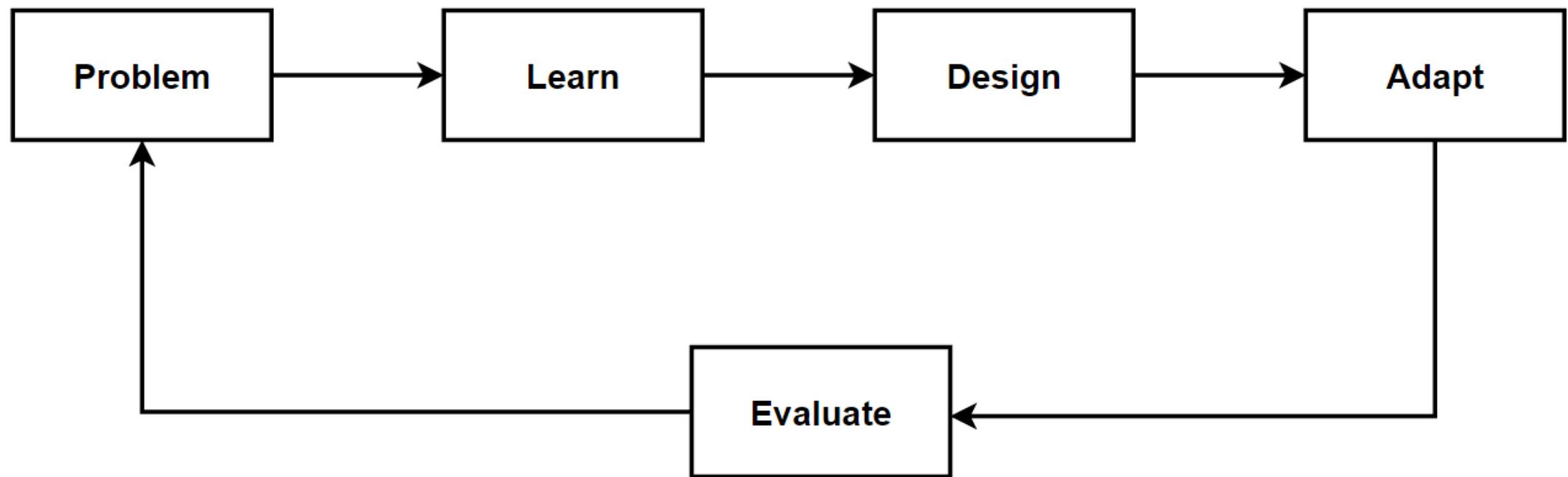
Architecture Design Journey



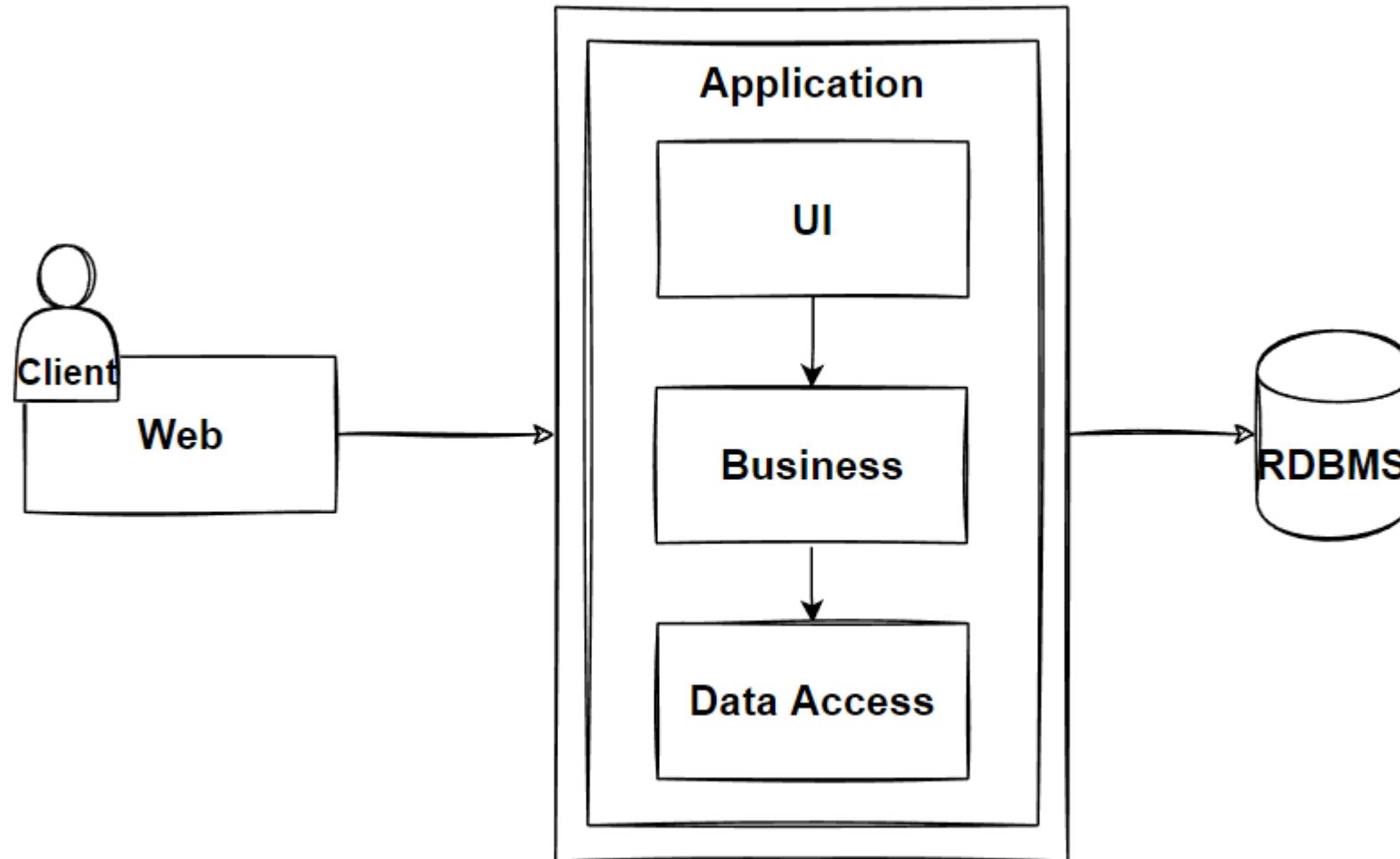
Architecture Design – Vertical Considerations



Way of Learning – The Course Flow



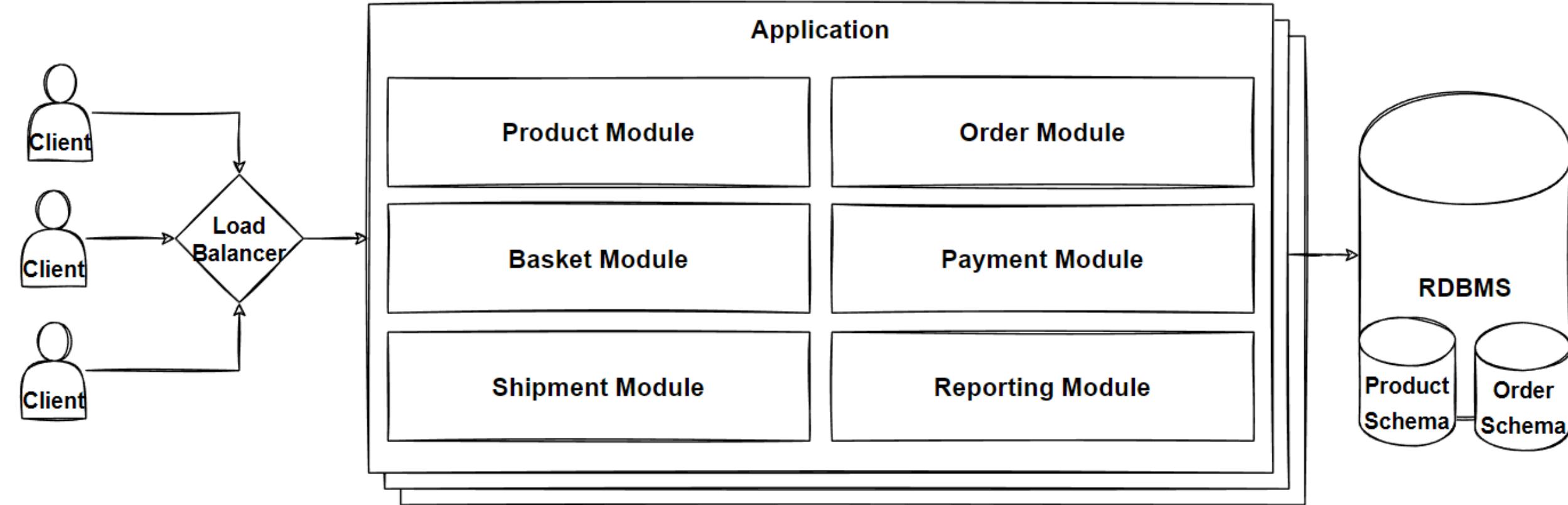
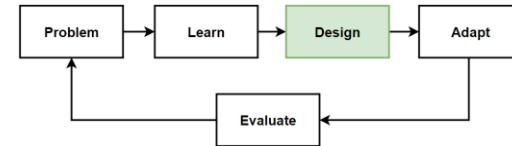
Monolithic Architecture



Modular Monolithic Architecture Patterns

Architectures	Patterns&Principles	Non-FR	FR
• Monolithic Architecture	• KISS • YAGNI	• Availability • High number of Concurrent User	• List products
• Layered Architecture	• Separation of Concerns (SoC)	• Maintainability	• Filter products as per brand and categories
• Clean Architecture		• Flexibility	• Put products into the shopping cart
• Modular Monolithic Architecture	• SOLID • The Dependency Rule • Horizontal Scaling • Load Balancer • Monolithic-First Strategy	• Testable • Scalability • Reliability • Re-usability	• Apply coupon for discounts • Checkout the shopping cart and create an order • List my old orders and order items history

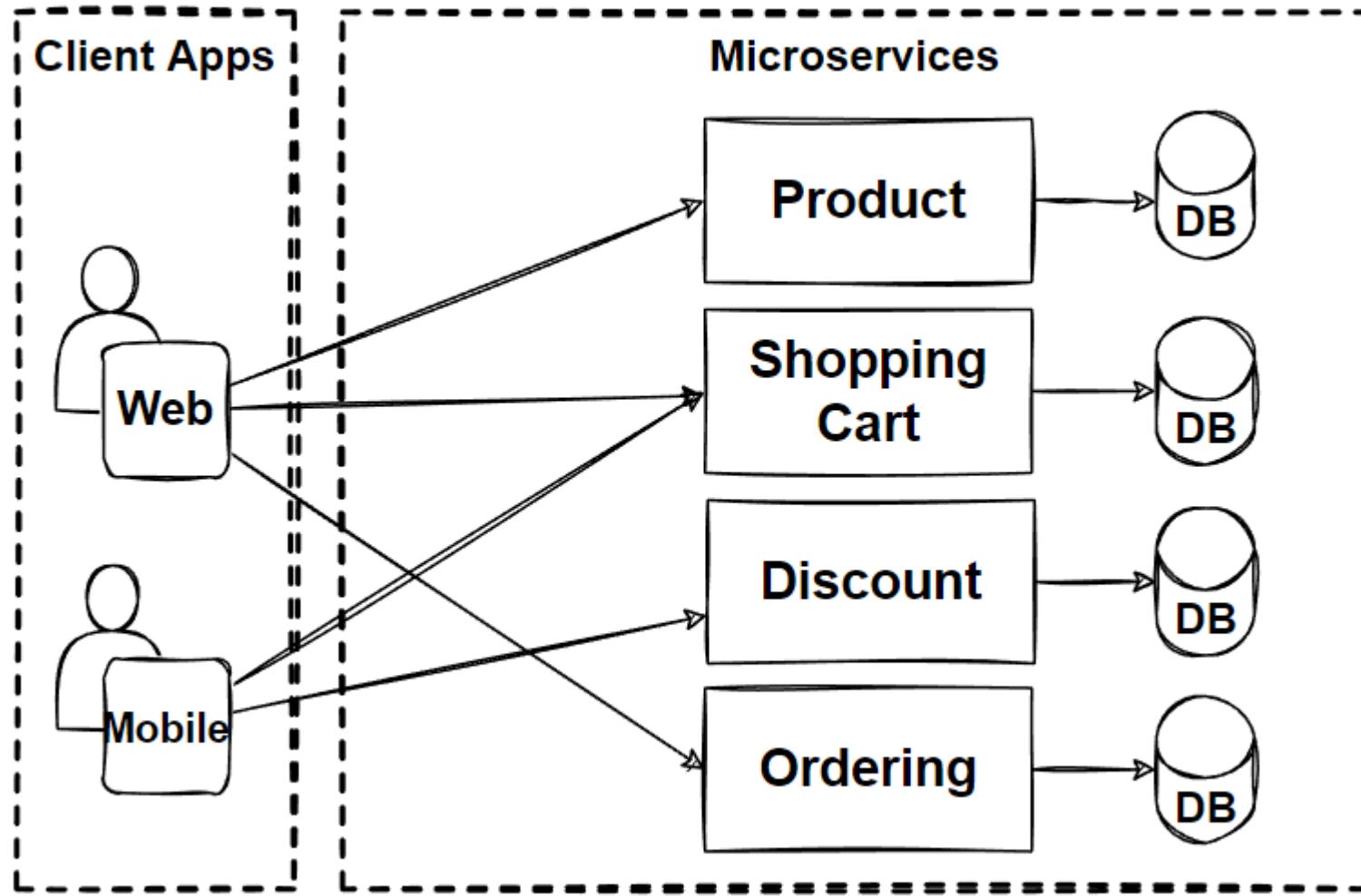
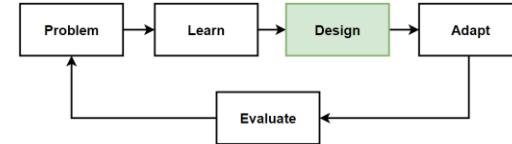
Design: Modular Monolithic Architecture



Microservices Architecture Patterns

Architectures	Patterns&Principles	Non-FR	FR
<ul style="list-style-type: none">• Microservices Architecture	<ul style="list-style-type: none">• The Database-per-Service Pattern• Polygot Persistence	<ul style="list-style-type: none">• High Scalability• High Availability• Millions of Concurrent User• Independent Deployable• Technology agnostic• Data isolation• Resilience and Fault isolation	<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history

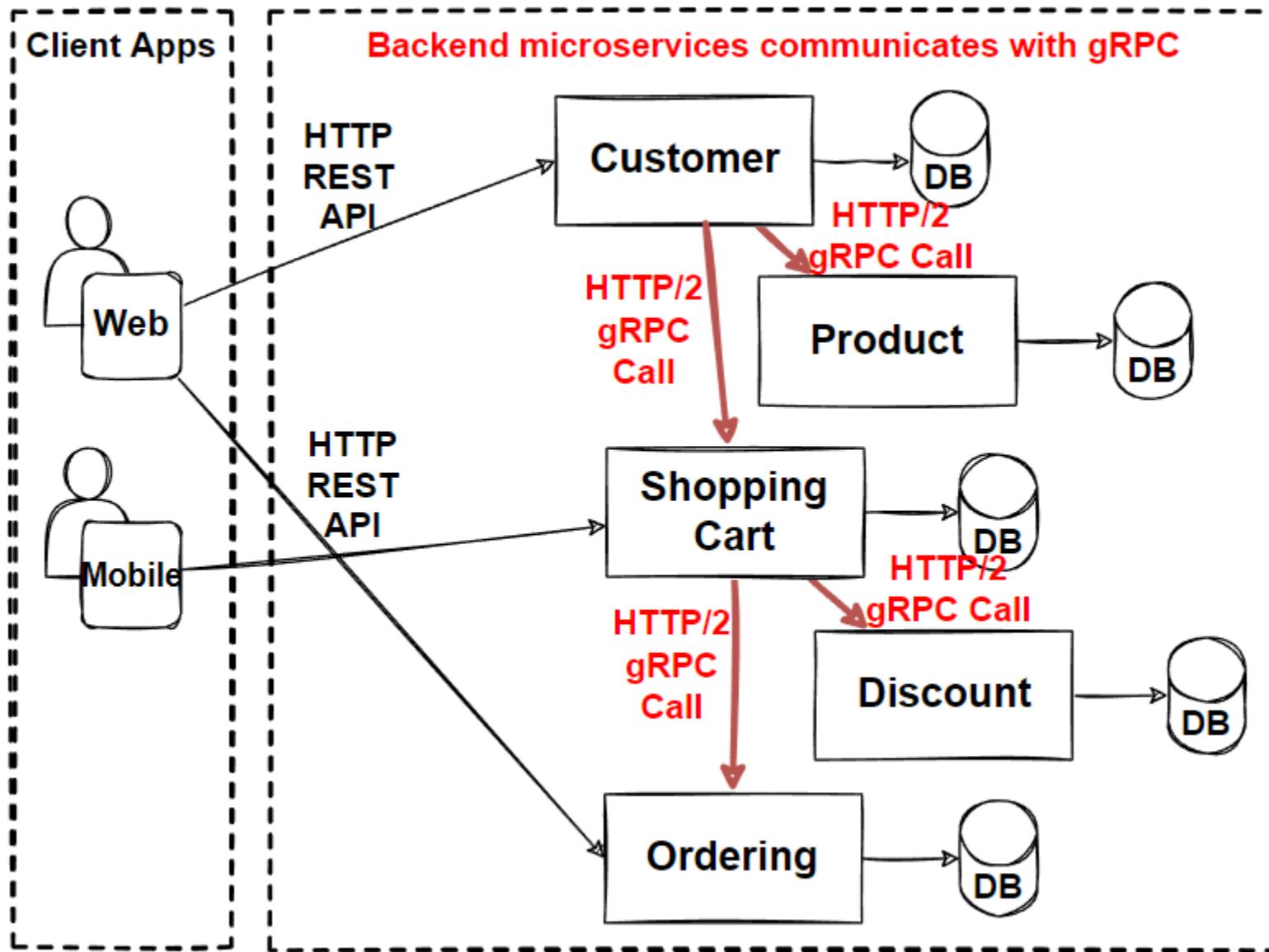
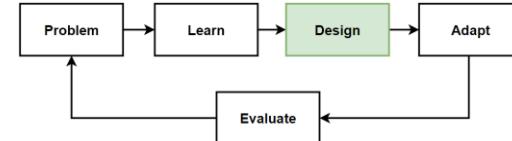
Design: Microservices Architecture



Microservices Communications

Architectures	Patterns&Principles	Non-FR	FR
<ul style="list-style-type: none">• Microservices Architecture	<ul style="list-style-type: none">• The Database-per-Service Pattern• Polygot Persistence• Decompose services by scalability• The Scale Cube• Microservices Decomposition Pattern• Microservices Communications<ul style="list-style-type: none">• HTTP Based RESTful API design• GraphQL API design• gRPC API Design	<ul style="list-style-type: none">• High Scalability• High Availability• Millions of Concurrent User• Independent Deployable• Technology agnostic• Data isolation• Resilience and Fault isolation	<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history

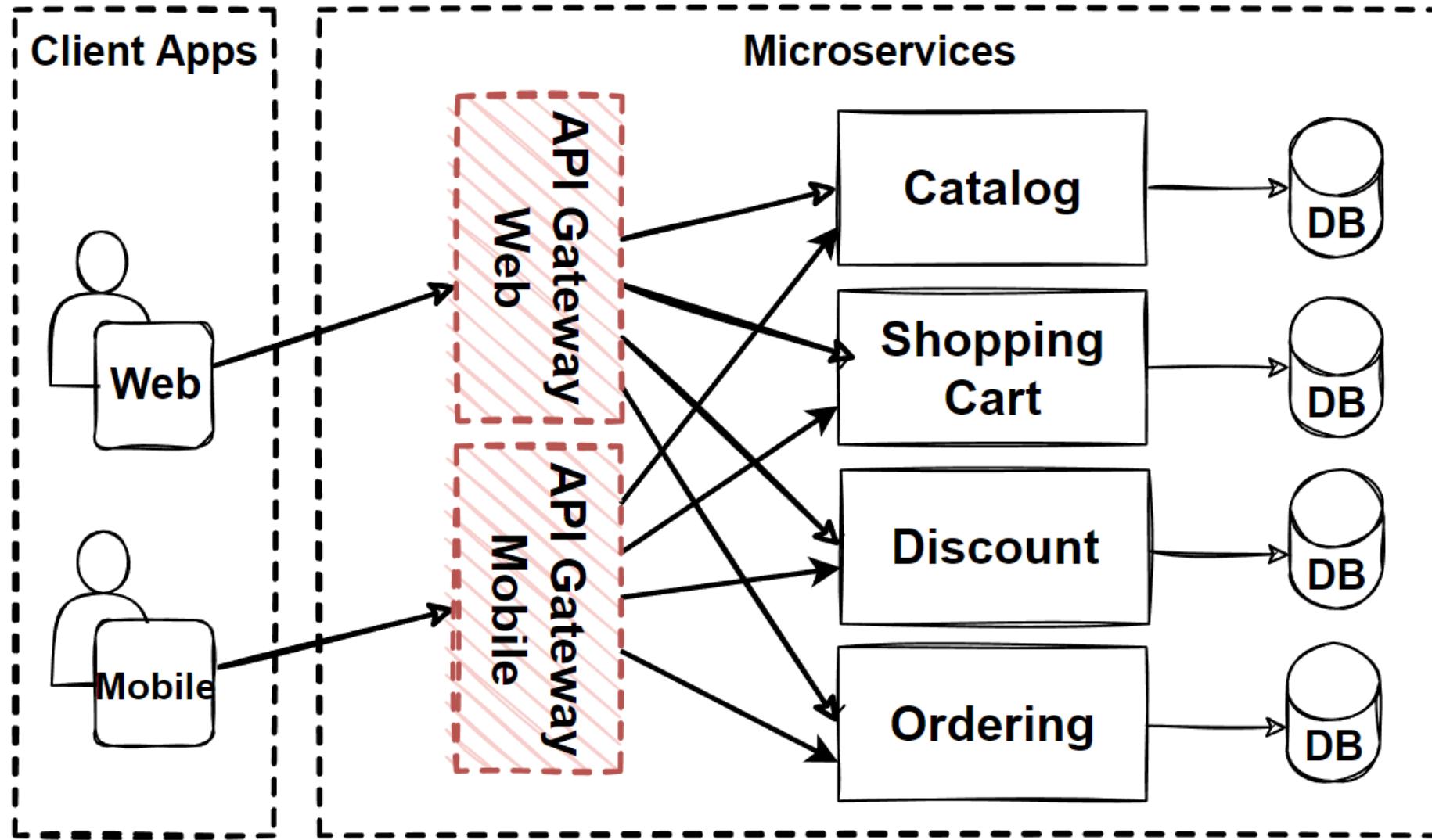
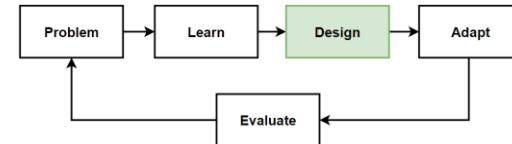
Design: Microservices Architecture with gRPC APIs



Microservices Communications Patterns

Architectures	Patterns&Principles	Microservices Communications	Non-FR	FR
<ul style="list-style-type: none">• Microservices Architecture• The Database-per-Service Pattern• Polygot Persistence• Decompose services by scalability• The Scale Cube• Microservices Decomposition Pattern• Microservices Communications Patterns	<ul style="list-style-type: none">• The Database-per-Service Pattern• Polygot Persistence• Decompose services by scalability• The Scale Cube• Microservices Decomposition Pattern• Microservices Communications Patterns	<ul style="list-style-type: none">• HTTP Based RESTful API• GraphQL API• gRPC API• WebSocket API• Gateway Routing Pattern• Gateway Aggregation Pattern• Gateway Offloading Pattern• API Gateway Pattern• Backends for Frontends Pattern-BFF	<ul style="list-style-type: none">• High Scalability• High Availability• Millions of Concurrent User• Independent Deployable• Technology agnostic• Data isolation• Resilience and Fault isolation	<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history

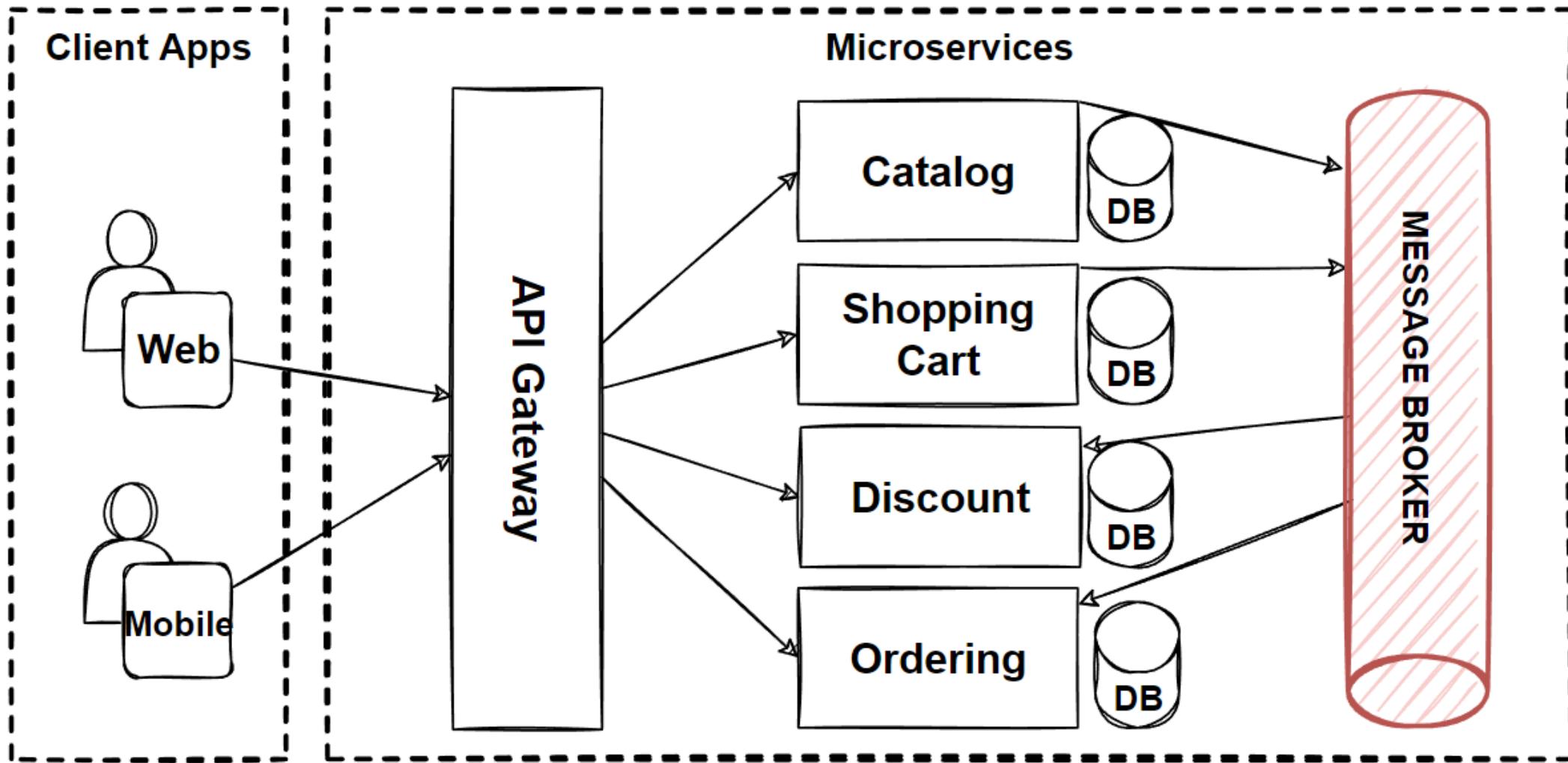
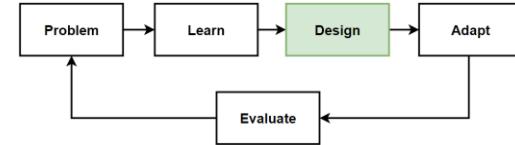
Design: Microservices Architecture with BFF



Microservices Async Communications Patterns

Architectures	Patterns&Principles	Microservices Communications	Microservices Async Communications	FR
• Microservices Architecture	<ul style="list-style-type: none">The Database-per-Service PatternPolygot PersistenceDecompose services by scalabilityThe Scale CubeMicroservices Decomposition Pattern	<ul style="list-style-type: none">HTTP Based RESTful APIGraphQL APIgRPC APIWebSocket APIGateway Routing PatternGateway Aggregation PatternGateway Offloading PatternAPI Gateway PatternBackends for Frontends Pattern-BFFService Aggregator PatternService Registry/Discovery Pattern	<ul style="list-style-type: none">Single-receiver Message-based Communication (one-to-one model)Multiple-receiver Message-based Communication (one-to-many model-topic)Dependency Inversion Principles (DIP)Fan-Out Publish/Subscribe Messaging PatternTopic-Queue Chaining & Load Balancing Pattern	<ul style="list-style-type: none">List productsFilter products as per brand and categoriesPut products into the shopping cartApply coupon for discountsCheckout the shopping cart and create an orderList my old orders and order items history
• Microservices Communications Patterns				Non-FR
				<ul style="list-style-type: none">High ScalabilityHigh AvailabilityMillions of Concurrent UserIndependent
				Mehmet Ozkaya 14

Design: Microservices Architecture with Fan-Out Publish/Subscribe Messaging Pattern



Microservices Data Management Patterns

Architectures Patterns&Principles

- **Microservices Architecture**
 - The Database-per-Service Pattern
 - Polygot Persistence
 - Decompose services by scalability
 - The Scale Cube
 - Microservices Decomposition Pattern
 - Microservices Communications Patterns
 - **Microservices Data Management Patterns**

Microservices Data Choosing Database

- The Shared Database Anti-pattern
- Relational and NoSQL Databases
- CAP Theorem—Consistency, Availability, Partition Tolerance
- Data Partitioning: Horizontal, Vertical and Functional Data Partitioning
- Database Sharding Pattern

Microservices Data Commands&Queries

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

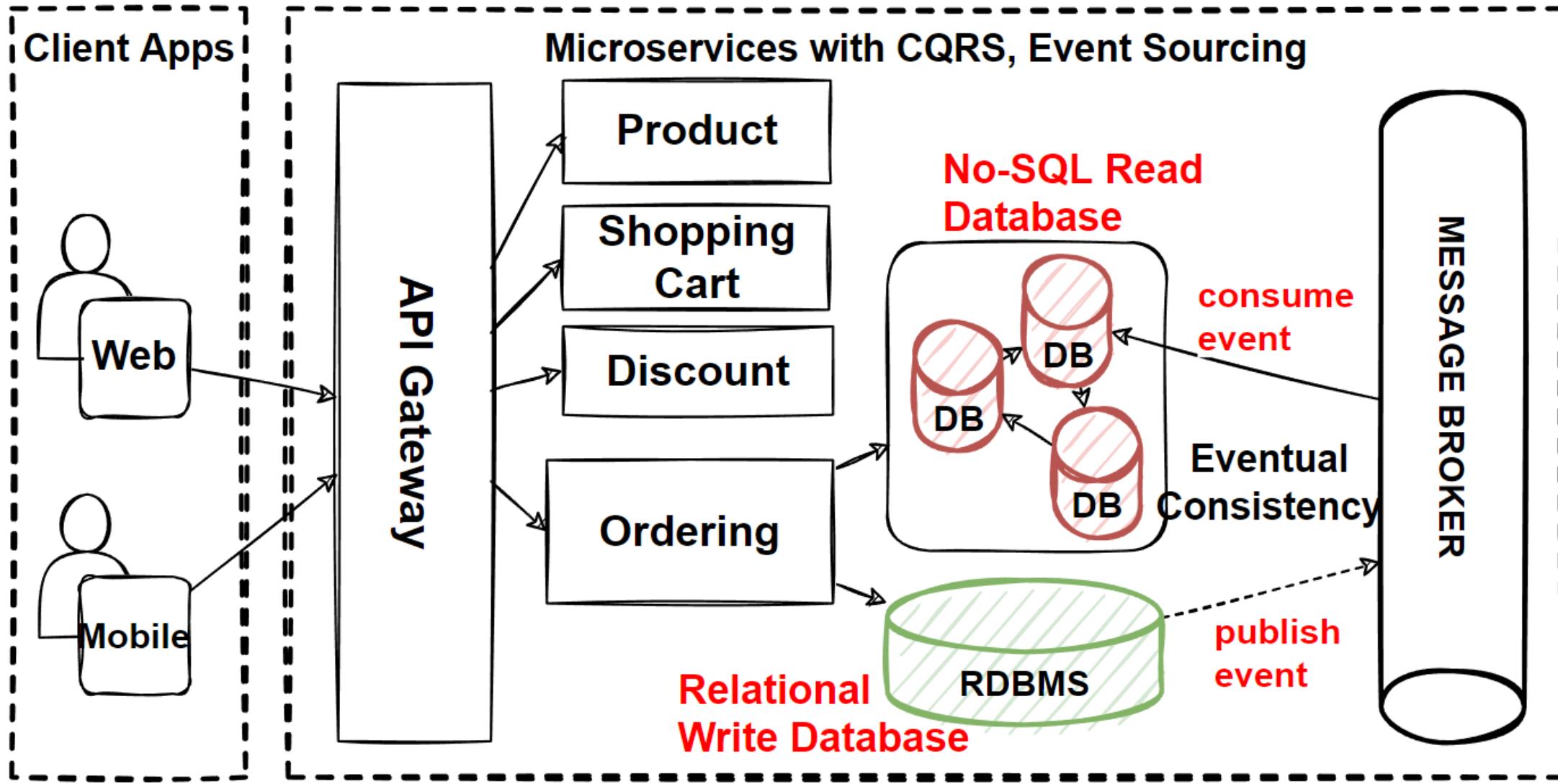
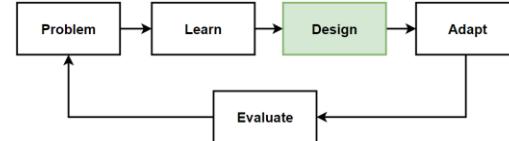
FR

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

Non-FR

- High Scalability
- High Availability
- Millions of Concurrent User
- Independent

Design: Microservices Architecture with CQRS, Event Sourcing, Eventual Consistency



Microservices Distributed Transaction Patterns

Architectures Patterns&Principles	Microservices Data Choosing Database	Microservices Distributed Transactions	FR
<ul style="list-style-type: none">• Microservices Architecture<ul style="list-style-type: none">• The Database-per-Service Pattern• Polygot Persistence• Decompose services by scalability• The Scale Cube• Microservices Decomposition Pattern• Microservices Communications Patterns• Microservices Data Management Patterns• Microservices Distributed Transaction Pattern	<ul style="list-style-type: none">• The Shared Database Anti-pattern, Relational and NoSQL Databases• CAP Theorem–Consistency, Availability, Partition Tolerance• Data Partitioning: Horizontal, Vertical and Functional Data Partitioning• Database Sharding Pattern	<ul style="list-style-type: none">• SAGA Pattern• Choreography and Orchestration-based SAGA• Compensating Transaction Pattern• Dual-Write Problem• Transactional Outbox Pattern• CDC - Change Data Capture	<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history

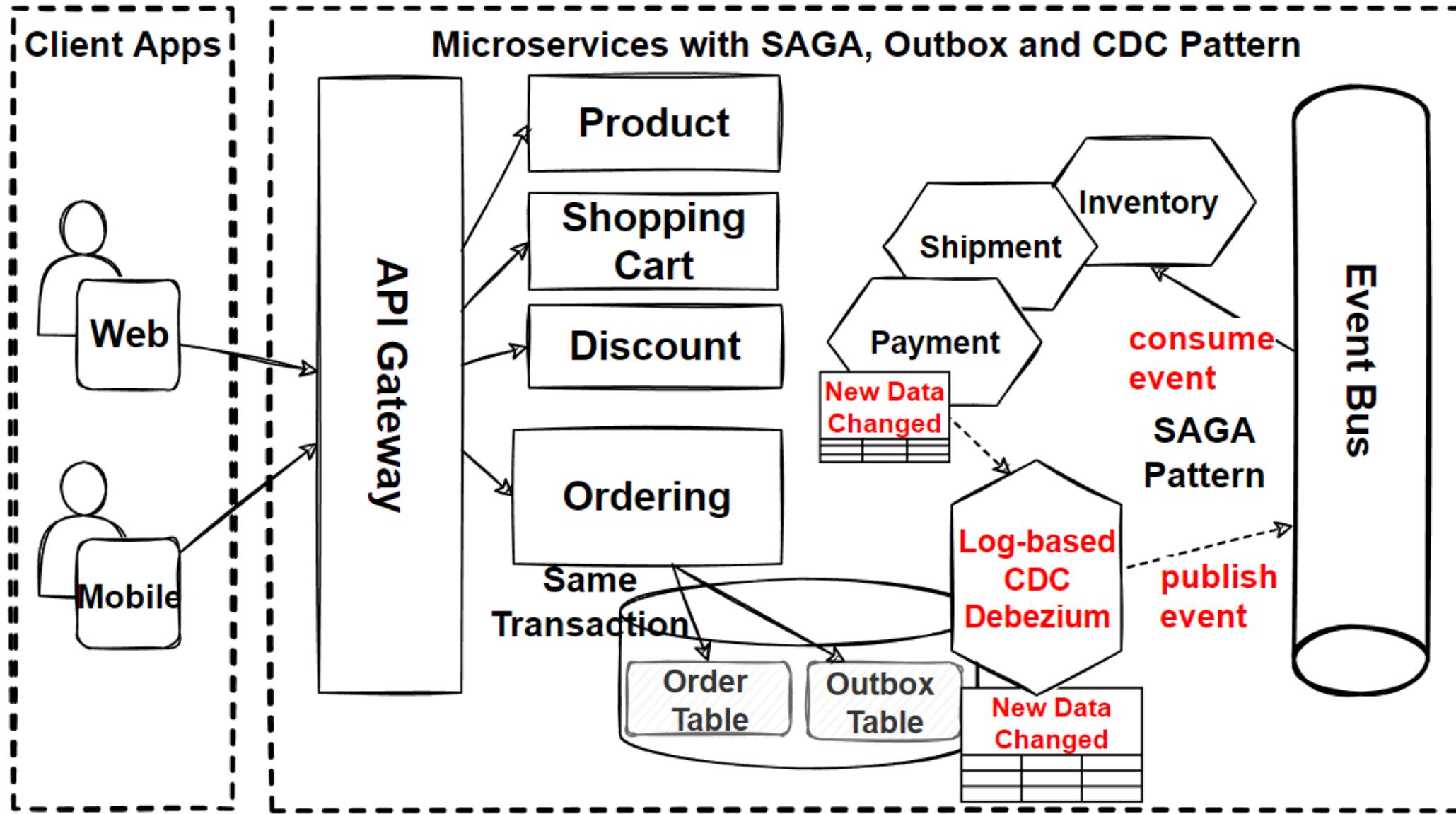
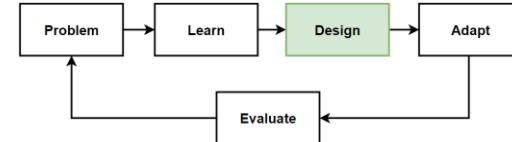
Non-FR

- High Scalability
- High Availability
- Millions of Concurrent User
- Independent

Mehmet Ozkaya

18

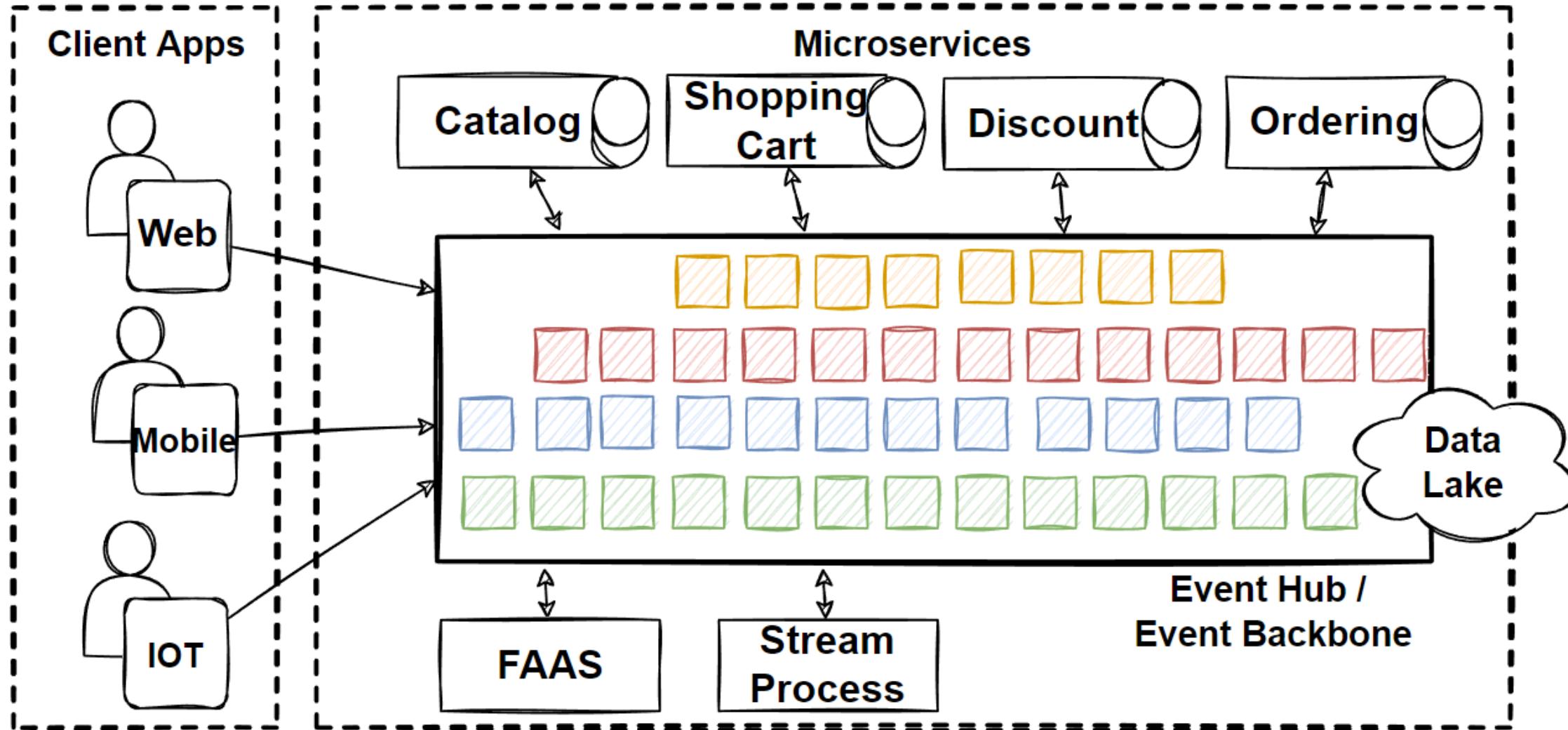
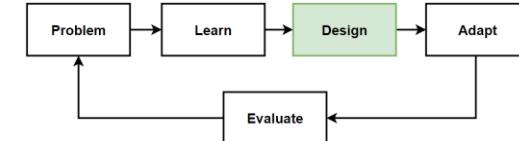
Microservices with SAGA, Transactional Outbox and CDC Pattern



Event-Driven Microservices Patterns

Architectures	Patterns&Principles	Microservices EDA	Non-FR	FR
<ul style="list-style-type: none">• Microservices Architecture• Event-Driven Microservices Architecture	<ul style="list-style-type: none">• The Database-per-Service Pattern• Polygot Persistence• Decompose services by scalability• The Scale Cube• Microservices Decomposition Pattern• Microservices Communications Patterns• Microservices Data Management Patterns• Event-Driven Architecture	<ul style="list-style-type: none">• Asynchronous, Decoupled communication• Event Hubs• Stream-Processing• Real-time processing• High volume events	<ul style="list-style-type: none">• High Scalability• High Availability• Millions of Concurrent User• Independent Deployable• Technology agnostic• Data isolation• Resilience and Fault isolation	<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history

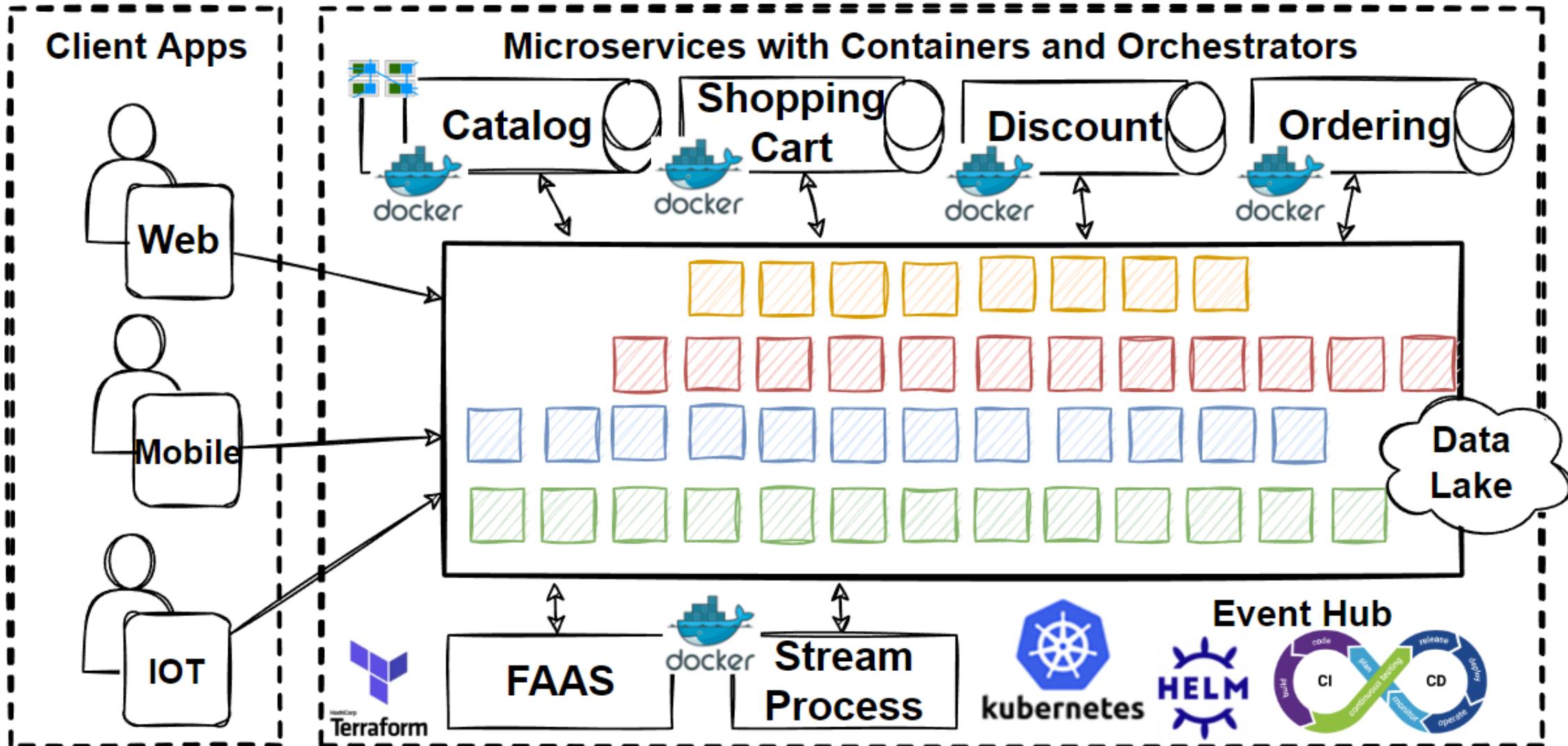
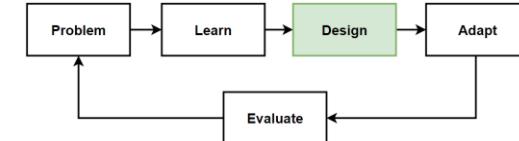
Event-Driven Microservices Architecture



Microservices Deployment Patterns

Architectures	Patterns&Principles	Microservices Deployment	Non-FR	FR
• Microservices Architecture • Event-Driven Microservices Architecture	<ul style="list-style-type: none">• The Database-per-Service Pattern, Polygot Persistence, Decompose services by scalability, The Scale Cube• Microservices Decomposition Pattern• Microservices Communications Patterns• Microservices Data Management Patterns• Event-Driven Architecture• Microservices Distributed Caching• Microservices Deployments with Containers and Orchestrators	<ul style="list-style-type: none">• Docker and Kubernetes Architecture, Helm Charts• Kubernetes Patterns; Sidecar Patterns, Service Mesh Pattern• DevOps and CI/CD Pipelines• Deployment Strategies; Blue-green, Rolling, Canary and A/B Deployment.• Infrastructure as code (IaC)	<ul style="list-style-type: none">• High Scalability• High Availability• Millions of Concurrent User• Independent Deployable• Technology agnostic• Data isolation• Resilience and Fault isolation	<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history
				Mehmet Ozkaya 22

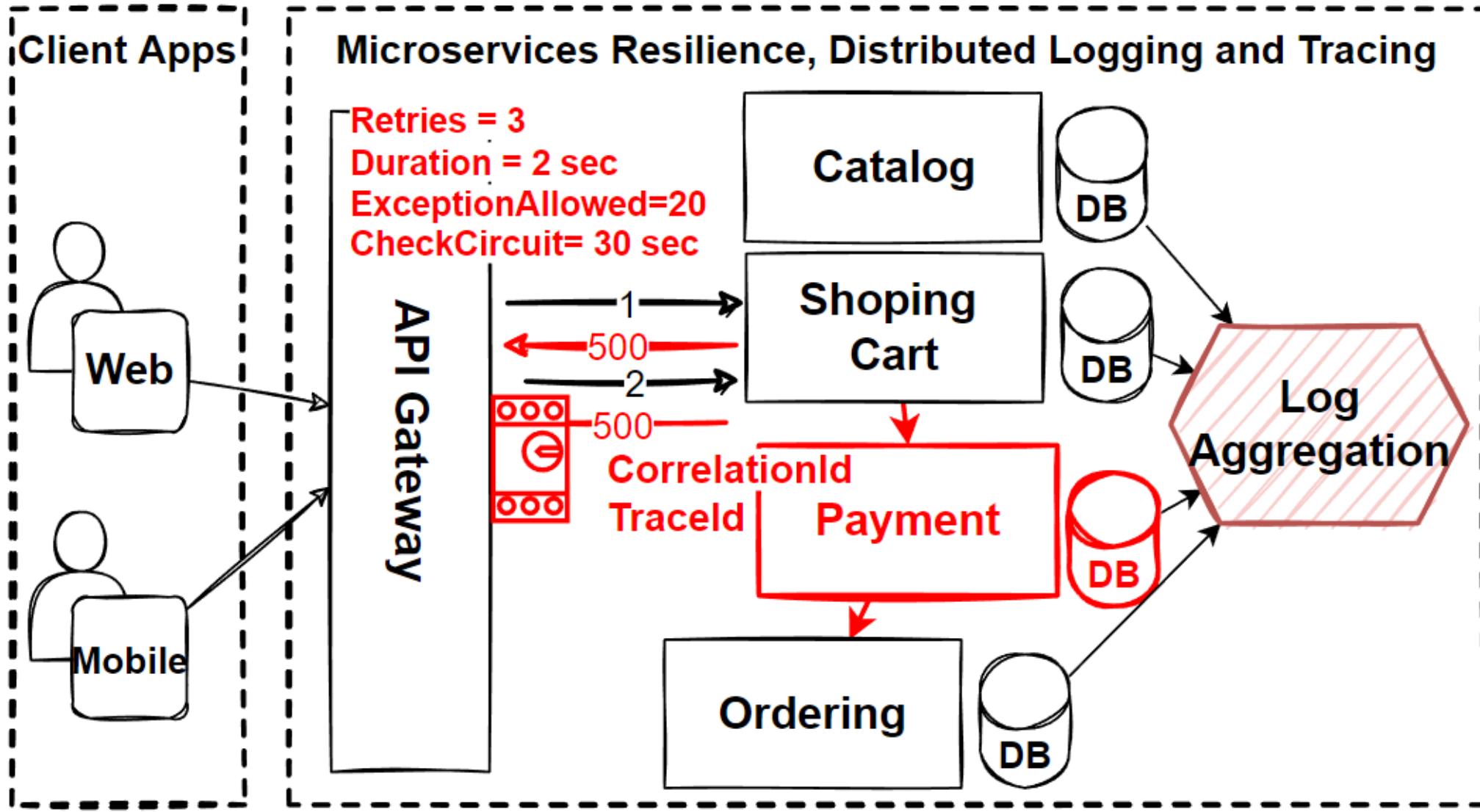
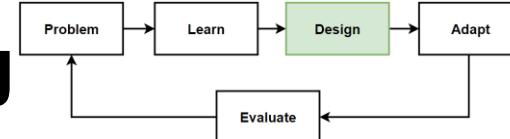
Microservices using Containers and Orchestrators



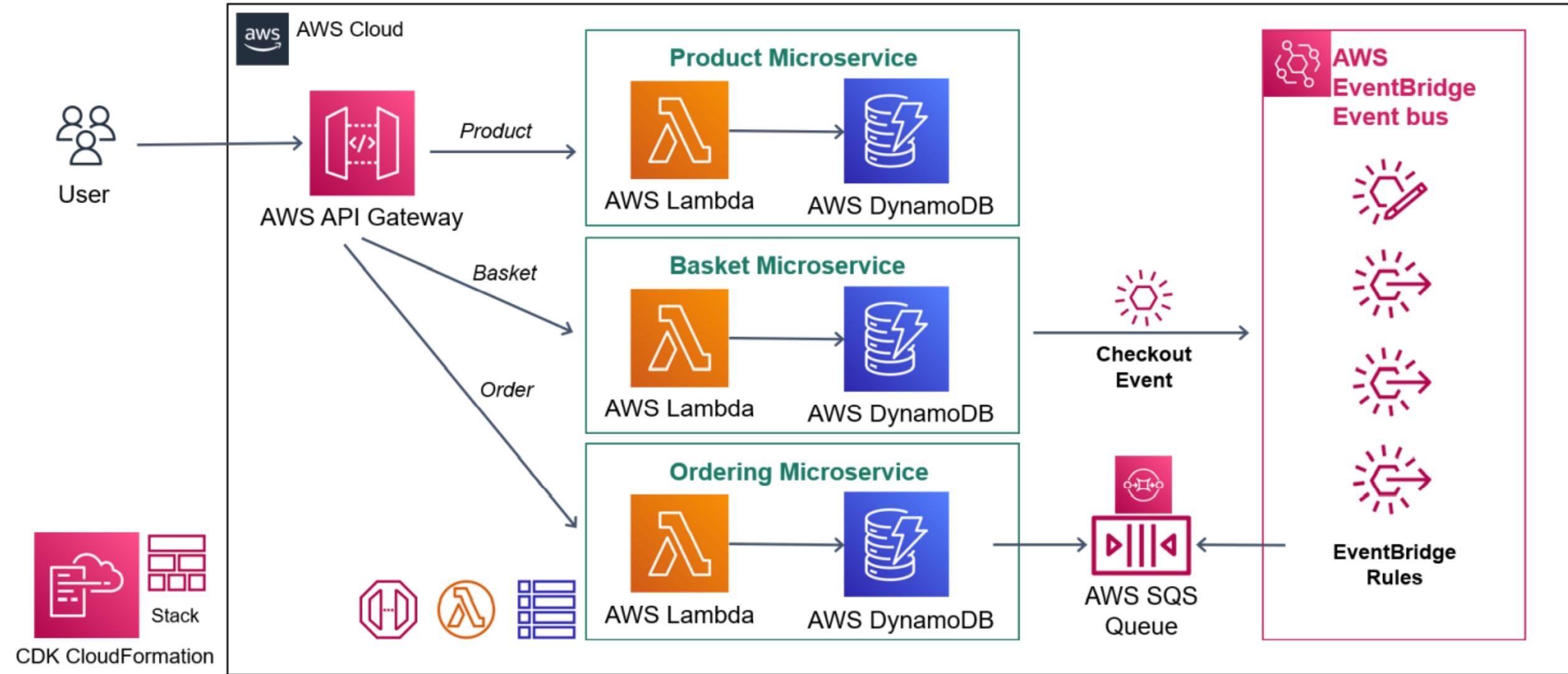
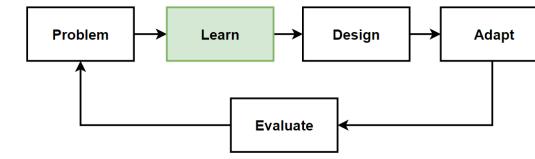
Microservices Resilience Patterns

Architectures	Patterns&Principles	Microservices Resilience	Non-FR	FR
• Microservices Architecture • Event-Driven Microservices Architecture	<ul style="list-style-type: none">The Database-per-Service Pattern, Polygot Persistence, Decompose services by scalability, The Scale CubeMicroservices Decomposition PatternMicroservices Communications PatternsMicroservices Data Management PatternsEvent-Driven ArchitectureMicroservices Distributed CachingMicroservices Deployments with Containers and Orchestrators	<ul style="list-style-type: none">Resilience Patterns; Retry, Circuit-Breaker, Bulkhead, Timeout, Fallback PatternDistributed Logging and Distributed TracingElastic Stack; Elasticsearch + Logstash + KibanaOpenTelemetry using ZipkinKubernetes Health Monitoring with tools like Prometheus and Grafana	<ul style="list-style-type: none">High ScalabilityHigh AvailabilityMillions of Concurrent User	<ul style="list-style-type: none">List productsFilter products as per brand and categoriesPut products into the shopping cartApply coupon for discountsCheckout the shopping cart and create an orderList my old orders and order items history
		Microservices Resilience, Observability and Monitoring		

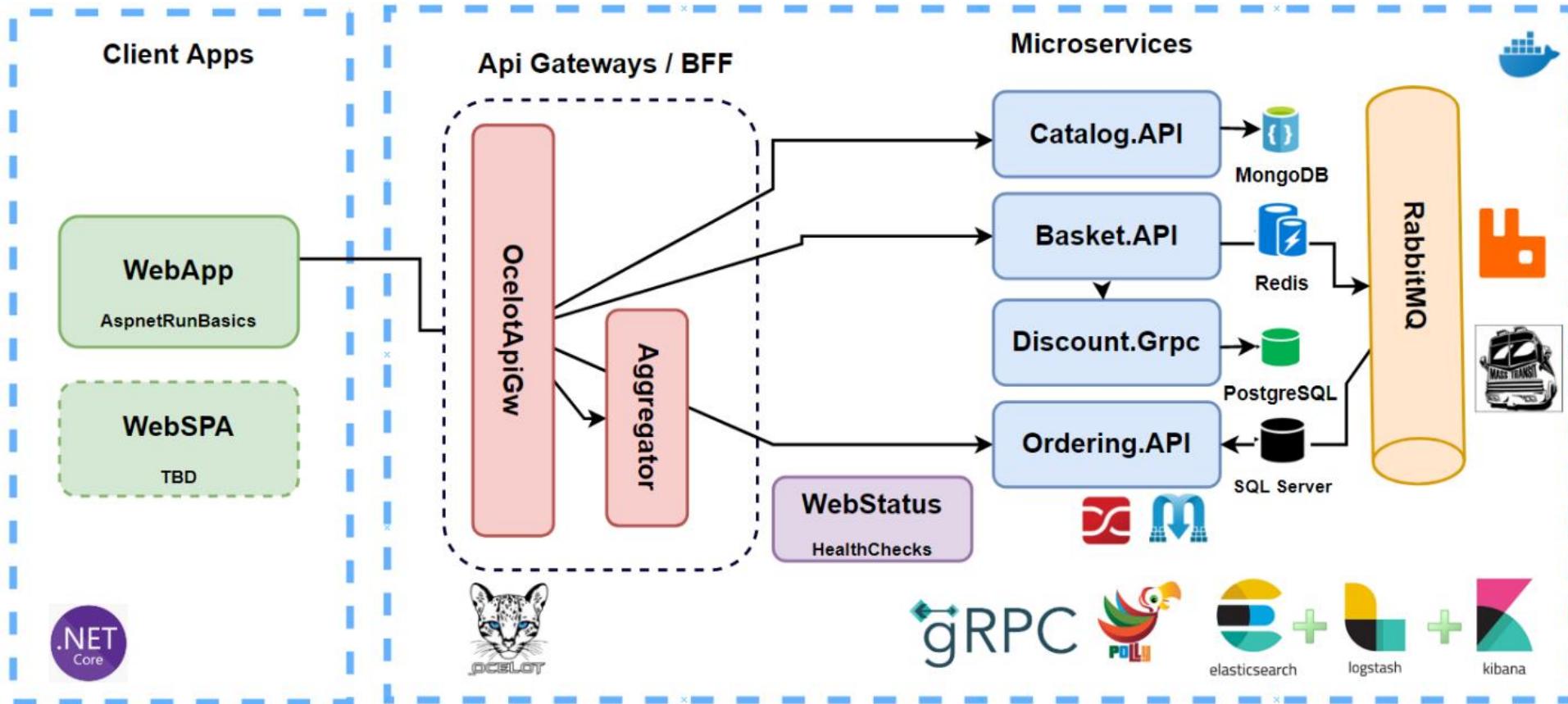
Microservices Resilience, Observability and Monitoring



Design: Serverless E-Commerce Microservices



DEMO: Microservices Architecture Code Review



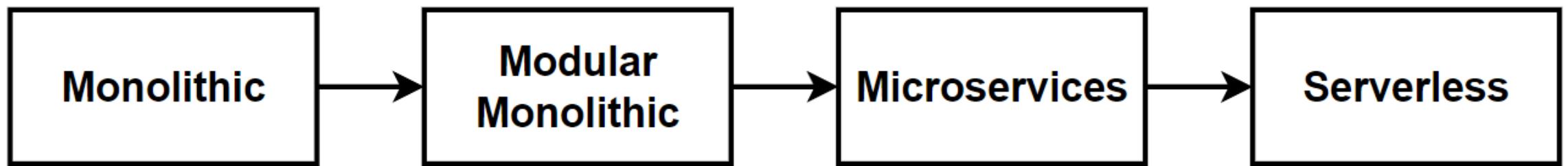
DEMO: Code review of Microservices Architecture .NET Implementation

- <https://github.com/aspnetrun/run-aspnetcore-microservices>
- <https://github1s.com/aspnetrun/run-aspnetcore-microservices>

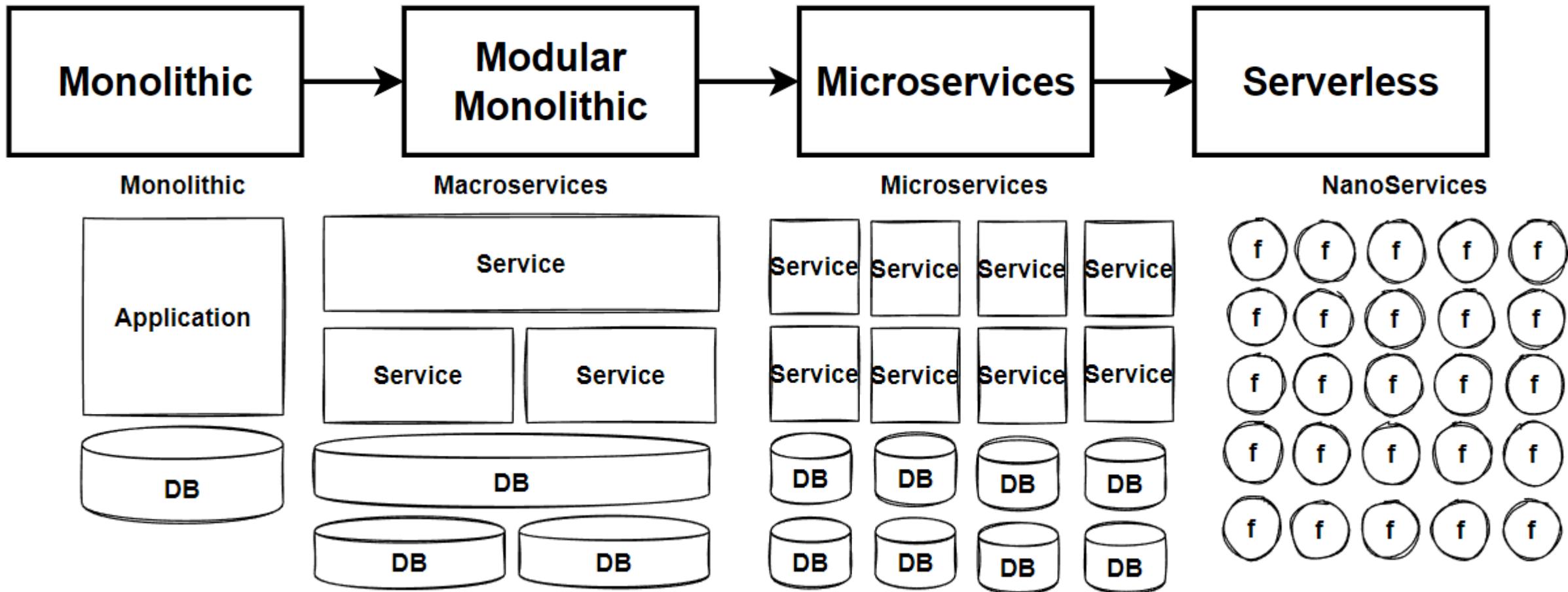
Course Target

- **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 Design Journey

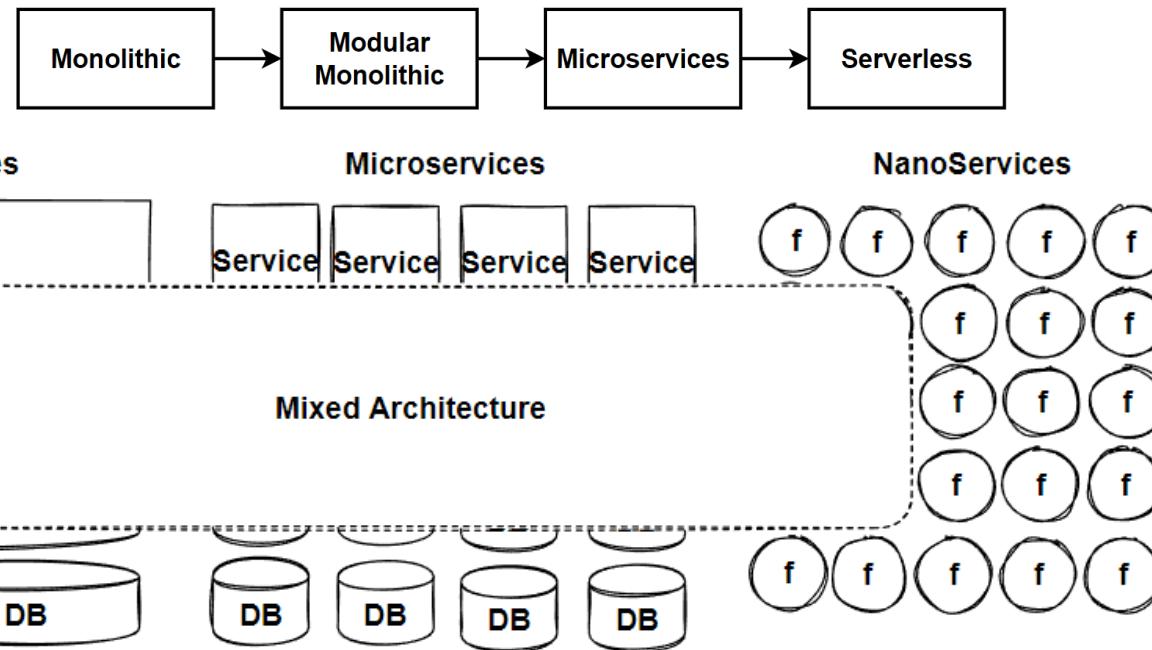


Macroservices to Nanoservices

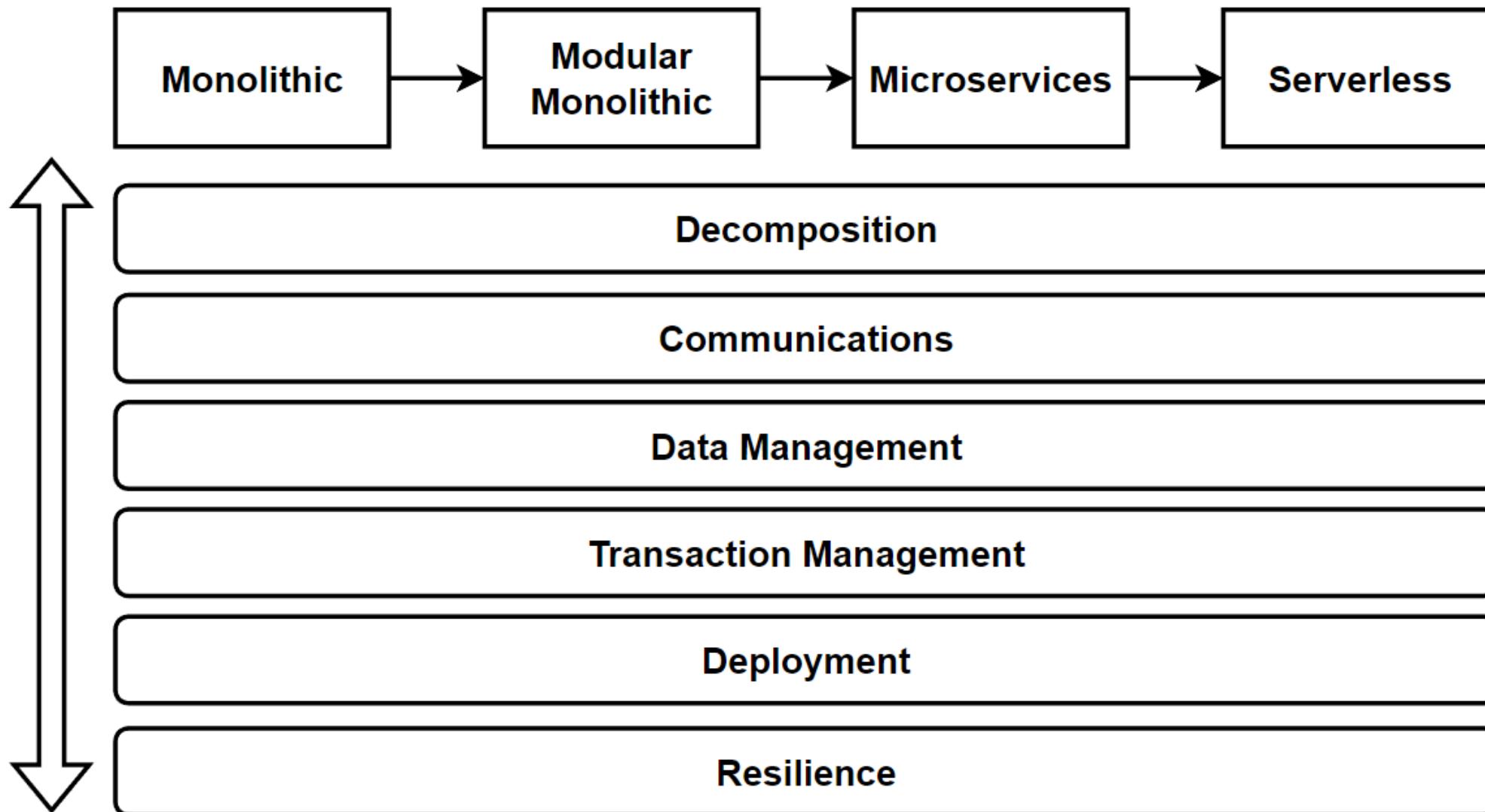


Which architecture approach we should choose ?

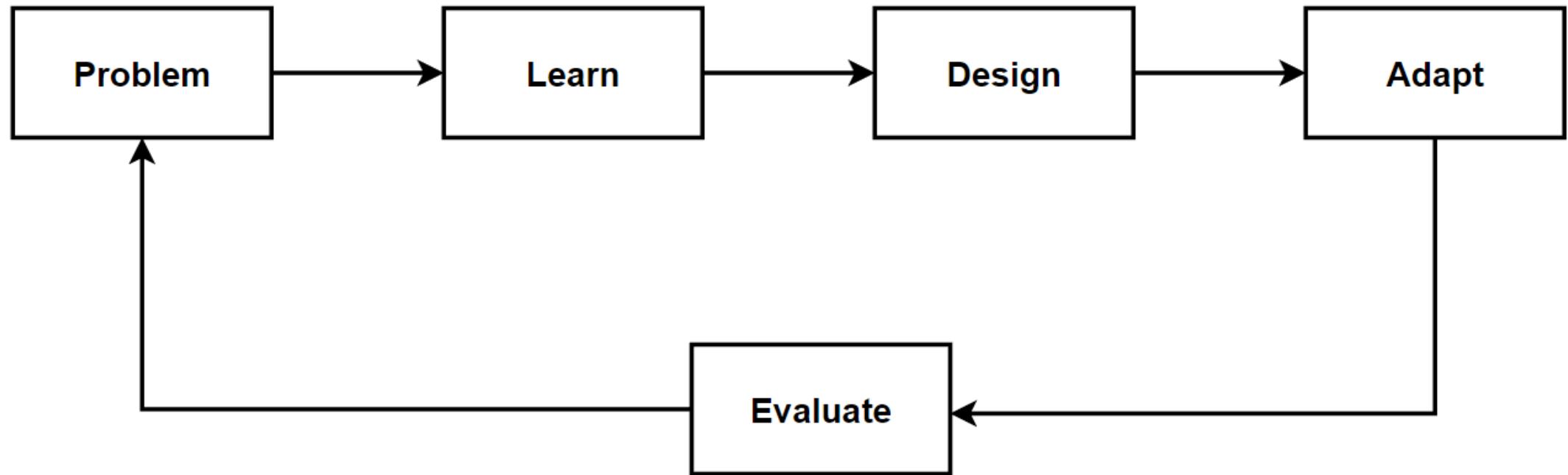
- It depends on your project
- Use Mixed Architecture
- Provide transitions between architectures



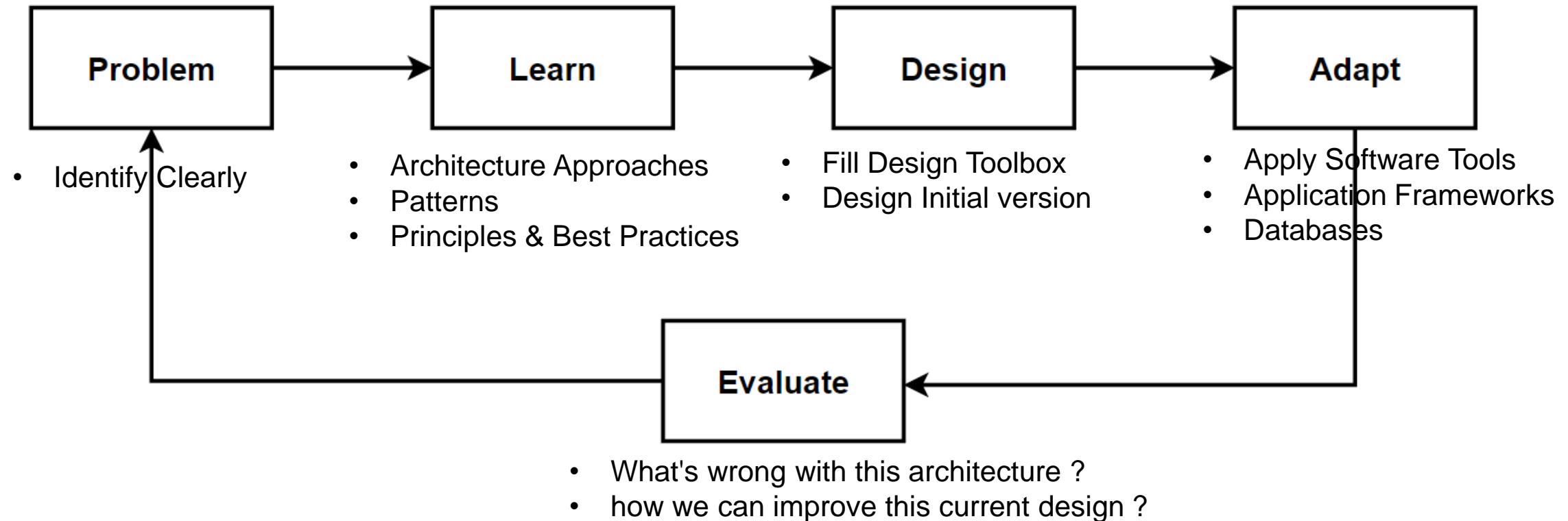
Architecture Design – Vertical Considerations



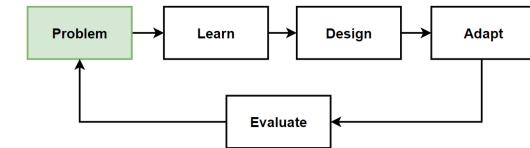
Way of Learning – The Course Flow



Way of Learning – The Course Flow



Problem: Increased Traffic, Handle More Request

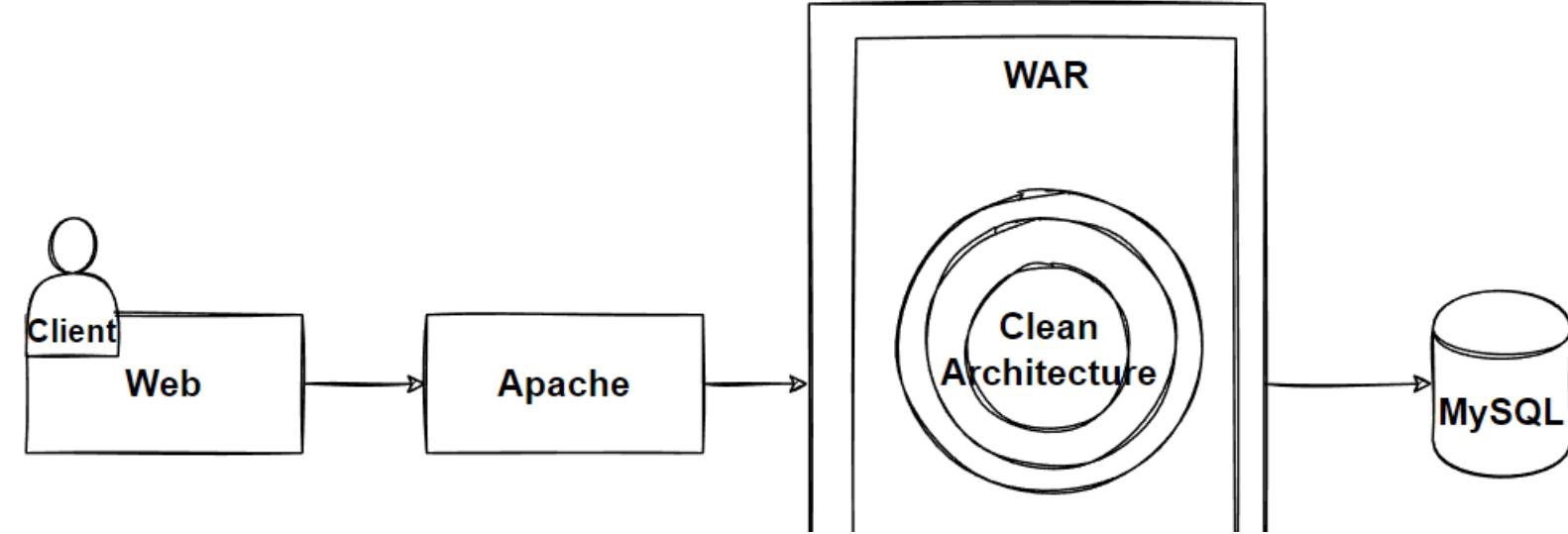


Problems

- Our E-Commerce Business is growing
- Need to handle greater amount of request per second
- Provide acceptable latency for users

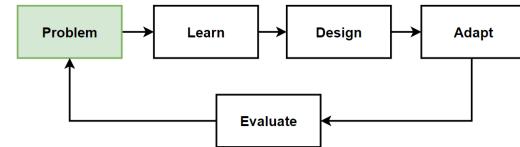
Solutions

- Scalability
- Vertical and Horizontal Scaling
- Scale Up and Scale Out
- Load Balancer



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

Problem: Break Down Application into Microservices

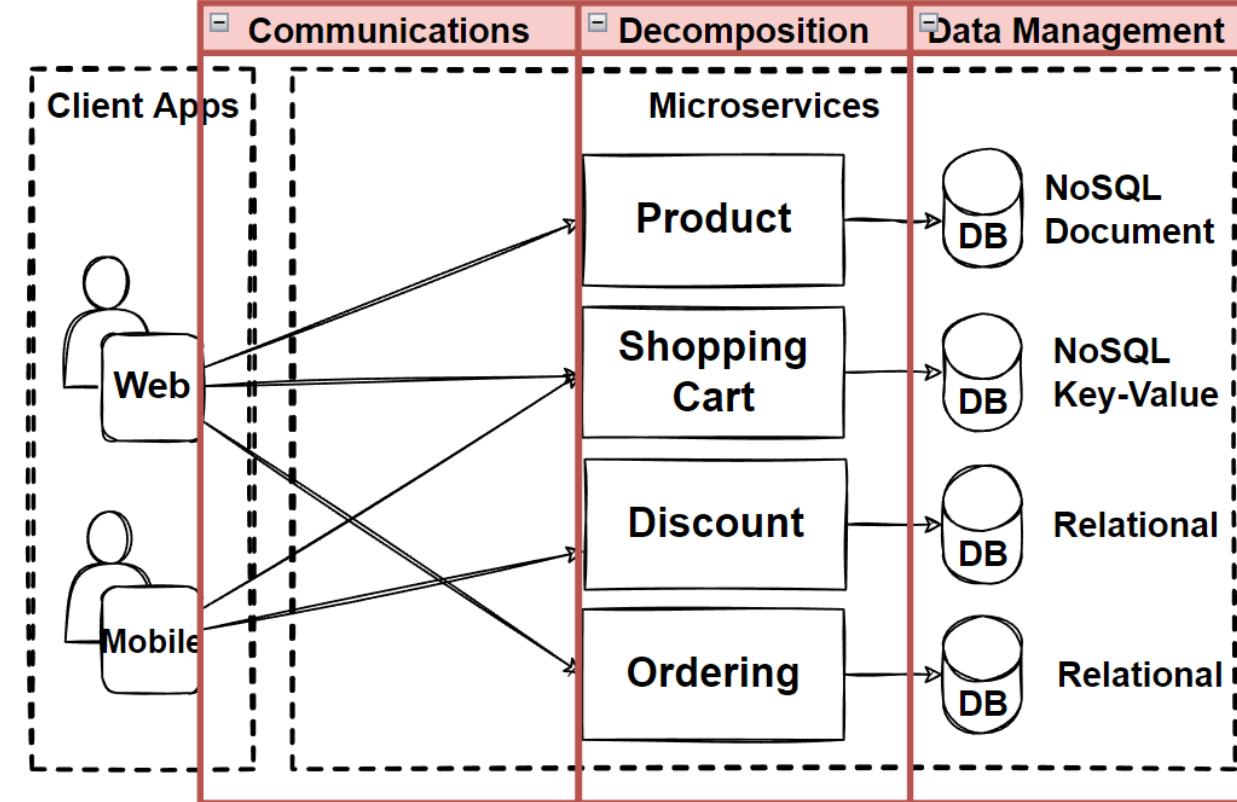


Problems

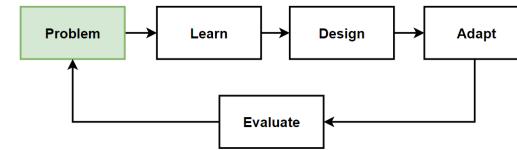
- Our E-Commerce Business is growing
- Teams want to be agile and add new features immediately to compete the market
- Required Independent Scale and Deployments
- We should clearly identify microservices which parts could be independent scale and deploy

Solutions

- Microservices Decomposition Patterns



Problem: Direct Client-to-Service Communication

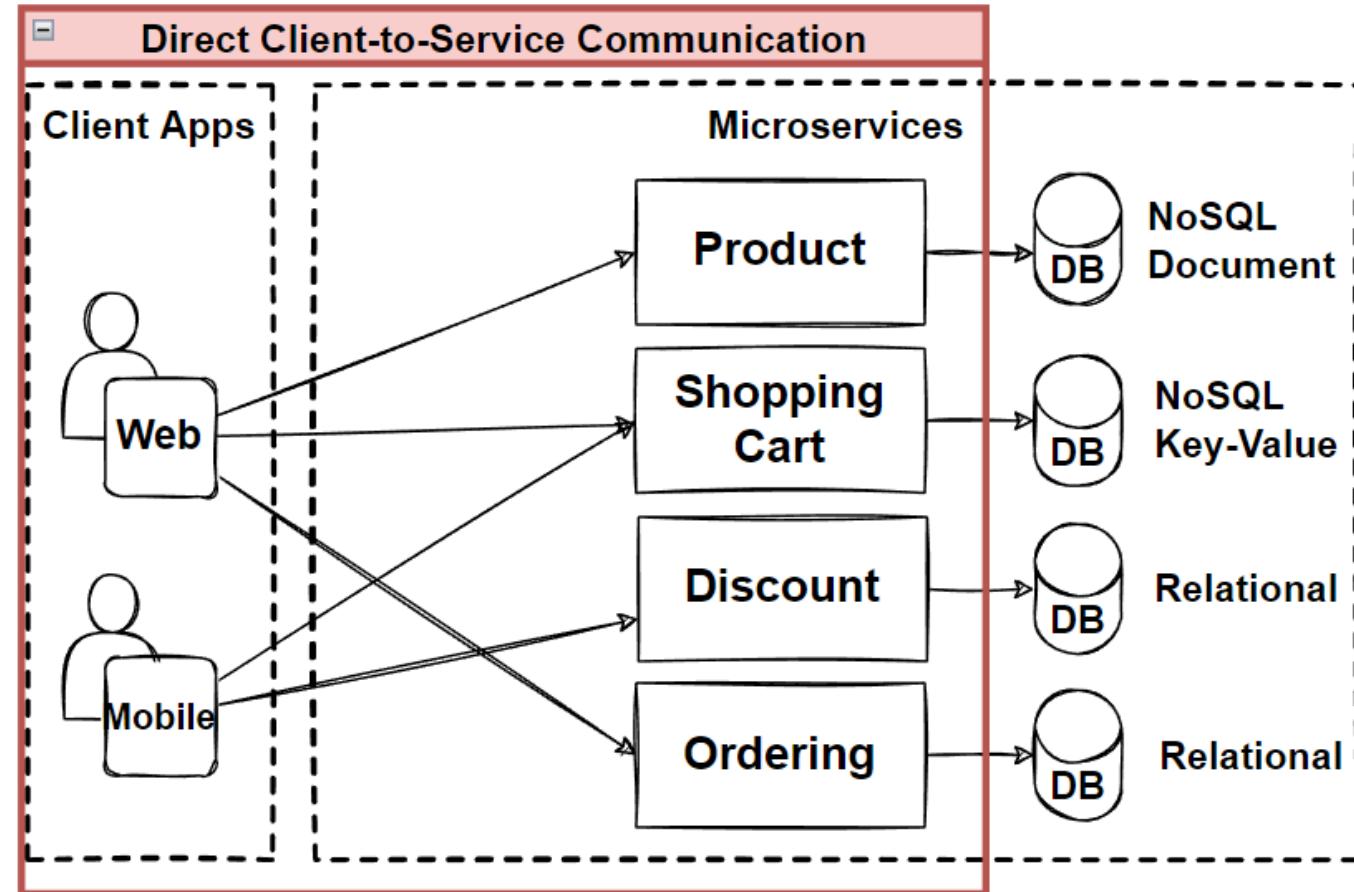


Problems

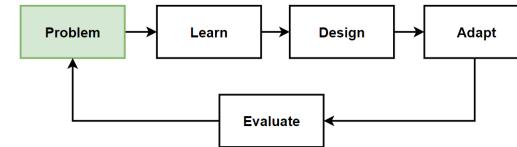
- Direct Client-to-Service Communication
- Cause to chatty calls from client to service
- Hard to manage invocations from client app.

Solutions

- Well-defined API Design
- Microservices Communication Patterns



Problem: Inter-service communication makes heavy load on network traffic



Problems

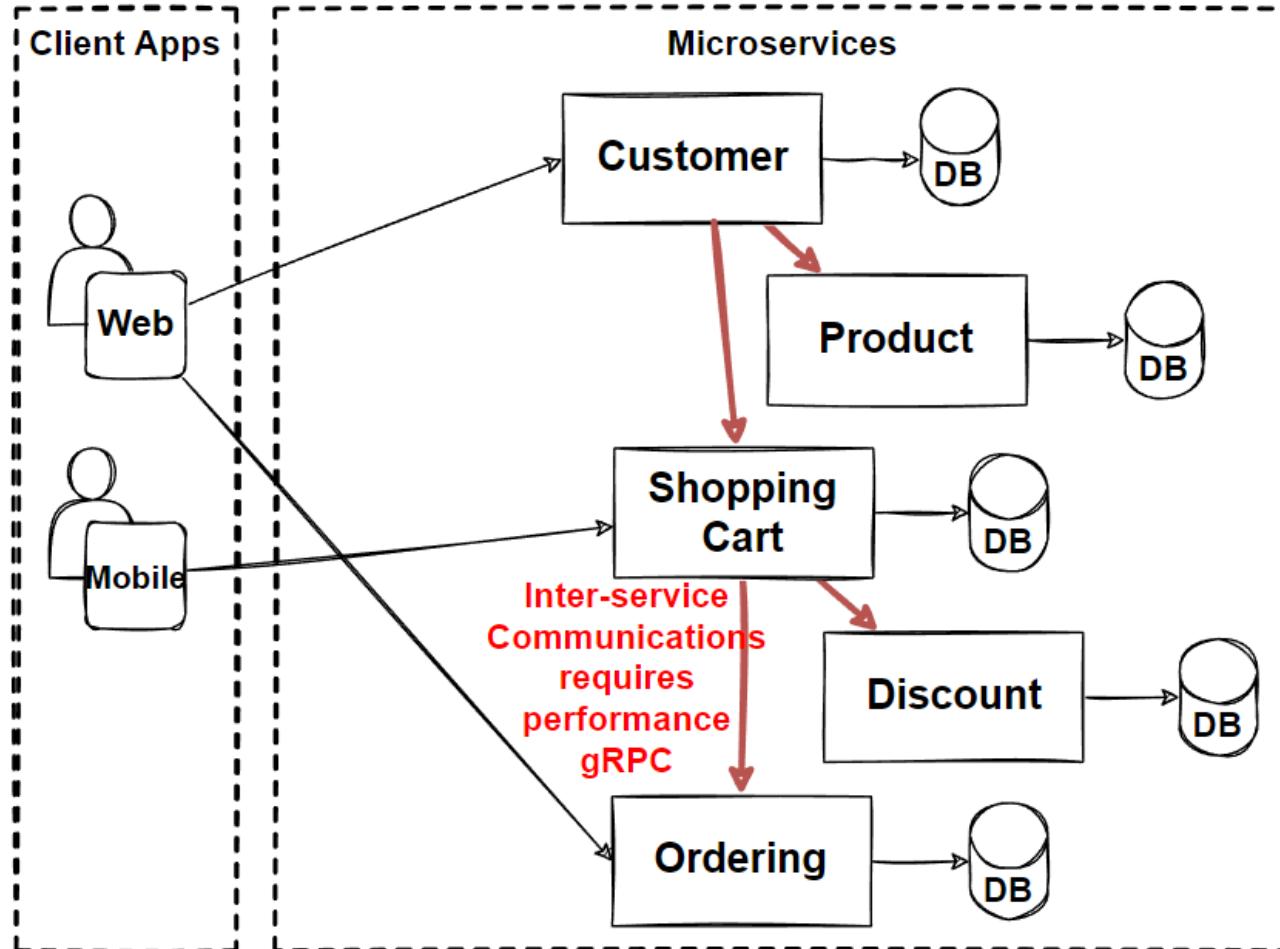
- Network performance issues on inter-service communication
- Backend Communication performance requirements
- Real-time communication requirements
- Streaming requirements

Example Use Case

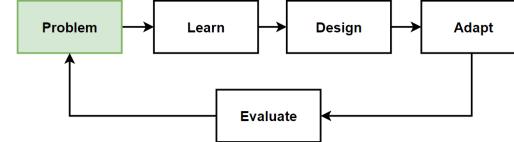
- Add Item into Shopping Cart that need to calculate with up-to-date discounts

Solutions

- gRPC APIs scalable and fast APIs
- Able to develop with different technologies with RPC framework

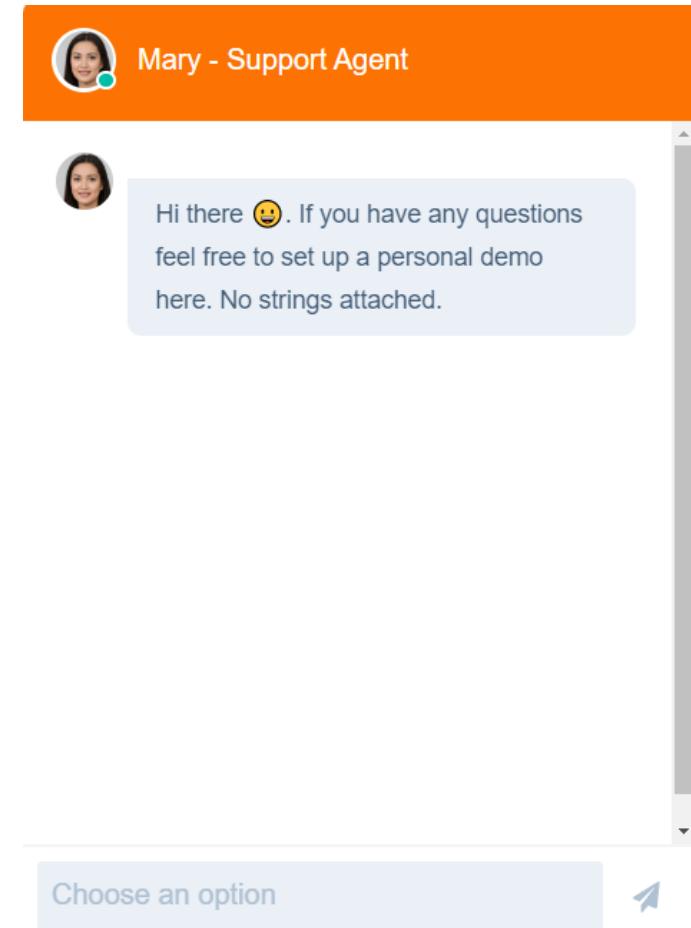


Problem: Chat with Support Agent



Problems

- Business teams request to answer Customer queries by chatting with Support Agents
- Real-time communication requirements
- Sending/receiving messages in Chat window



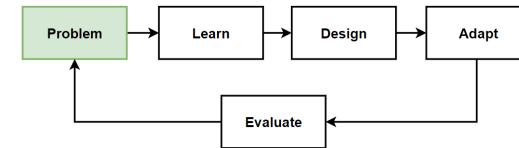
Example Use Case

- E-commerce Online Agent help customer preferences as per product features on website

Solutions

- WebSocket APIs: Build real-time two-way communication applications

Problem: Service-to-Service Communications Chain Queries

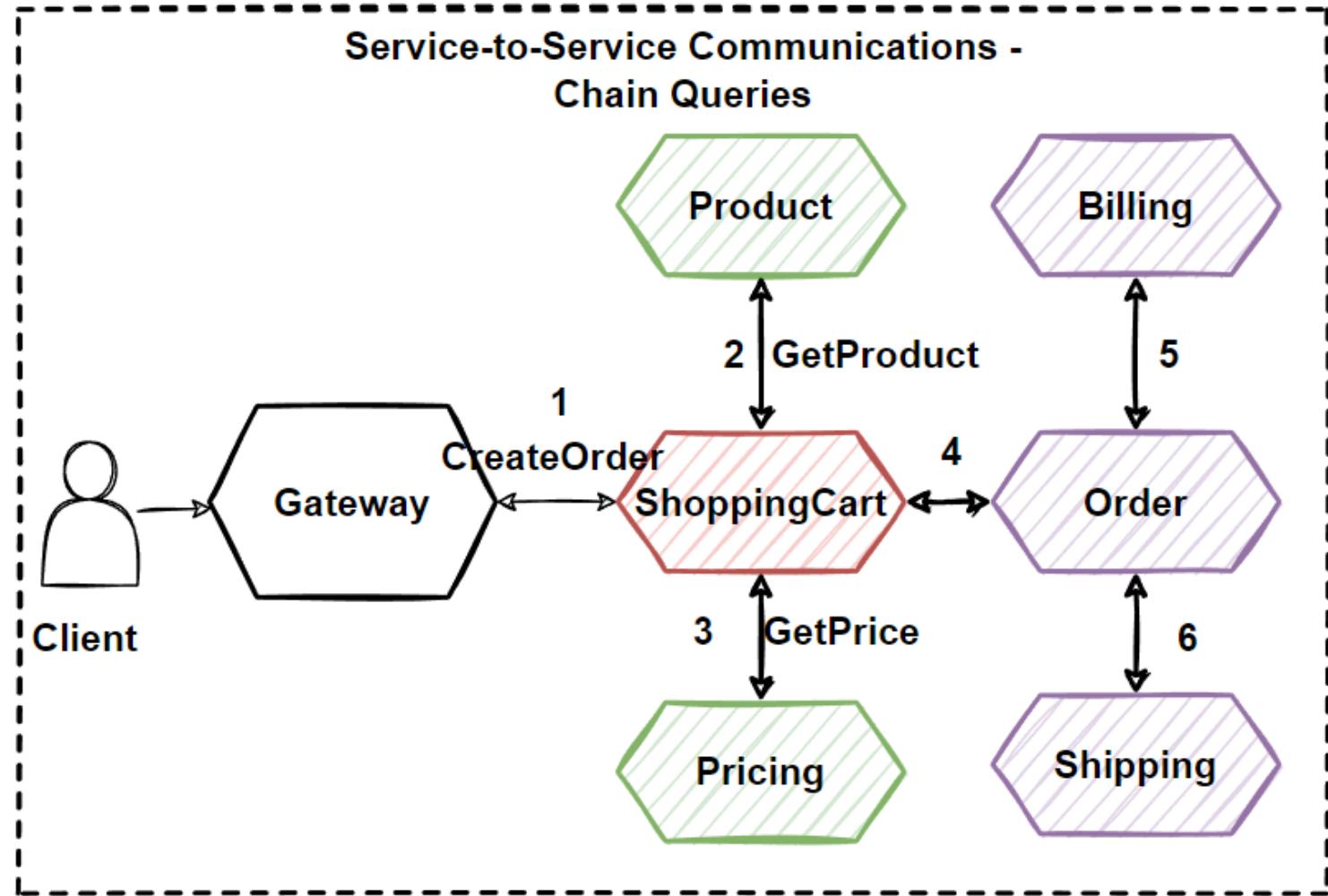


Problems

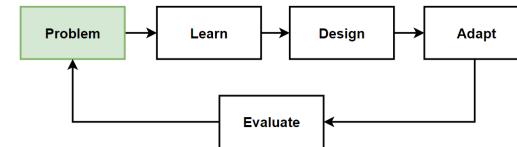
- HTTP calls to multiple microservices
- Chain Queries
- Visit more than a few microservices
- Increased latency

Solutions

- Aggregate query operations
- **Service Aggregator Pattern**



Problem: Long Running Operations Can't Handle with Sync Communication



Problems

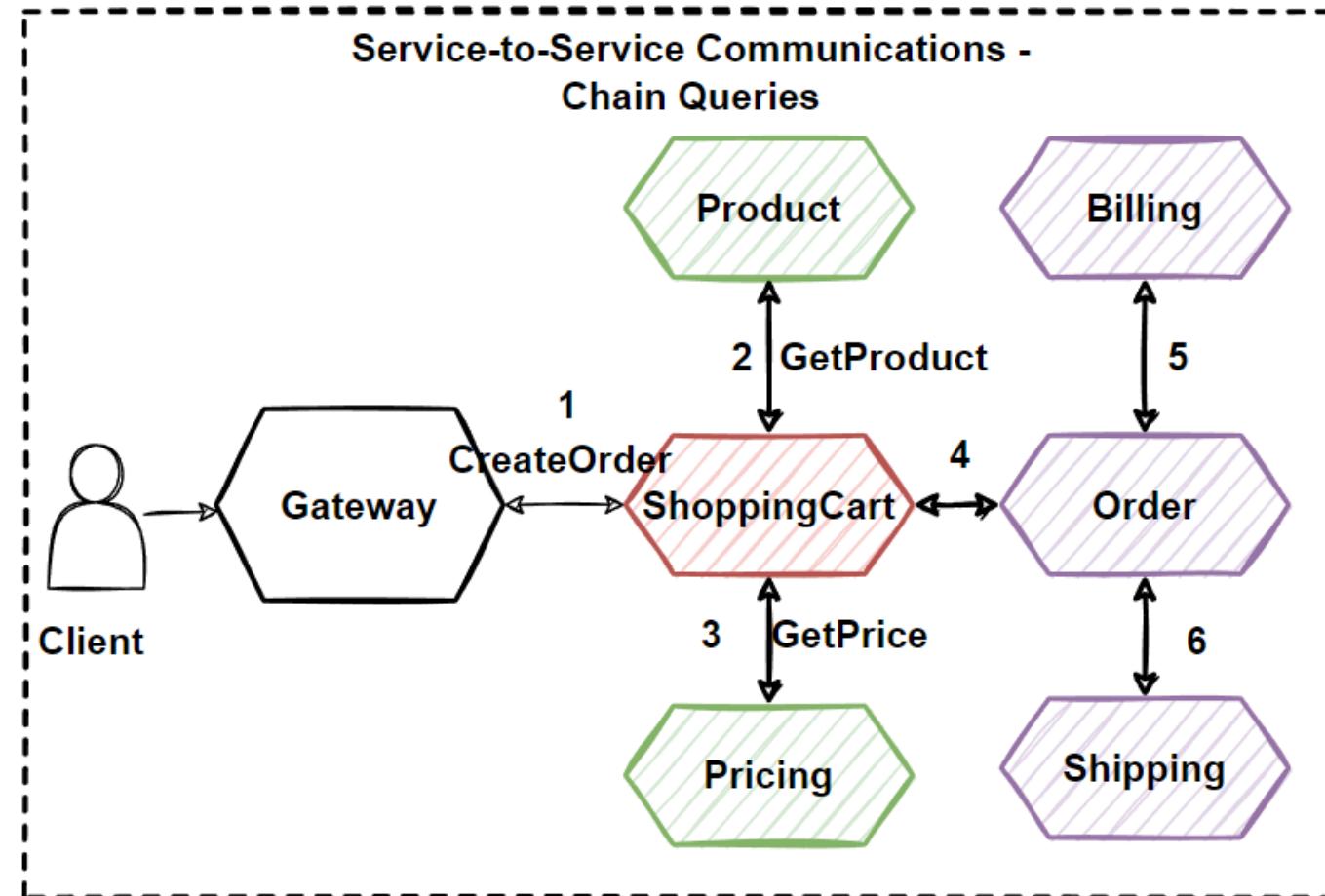
- HTTP calls to multiple microservices
- Chain Queries
- Visit more than a few microservices
- Increased latency with Highly Coupling Services
- Performance, scalability, and availability problems

Best Practices

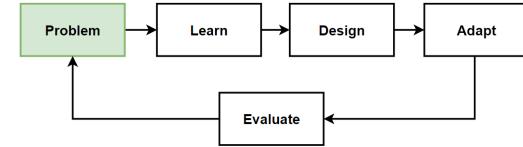
- Minimize the communication between the internal microservices
- make microservices communication in Asynchronous way as soon as possible.

Solutions

- **Asynchronous Message-Based Communications**
- Working with events



Problem: Database Bottlenecks when Scaling, Different Data Requirements For Microservices

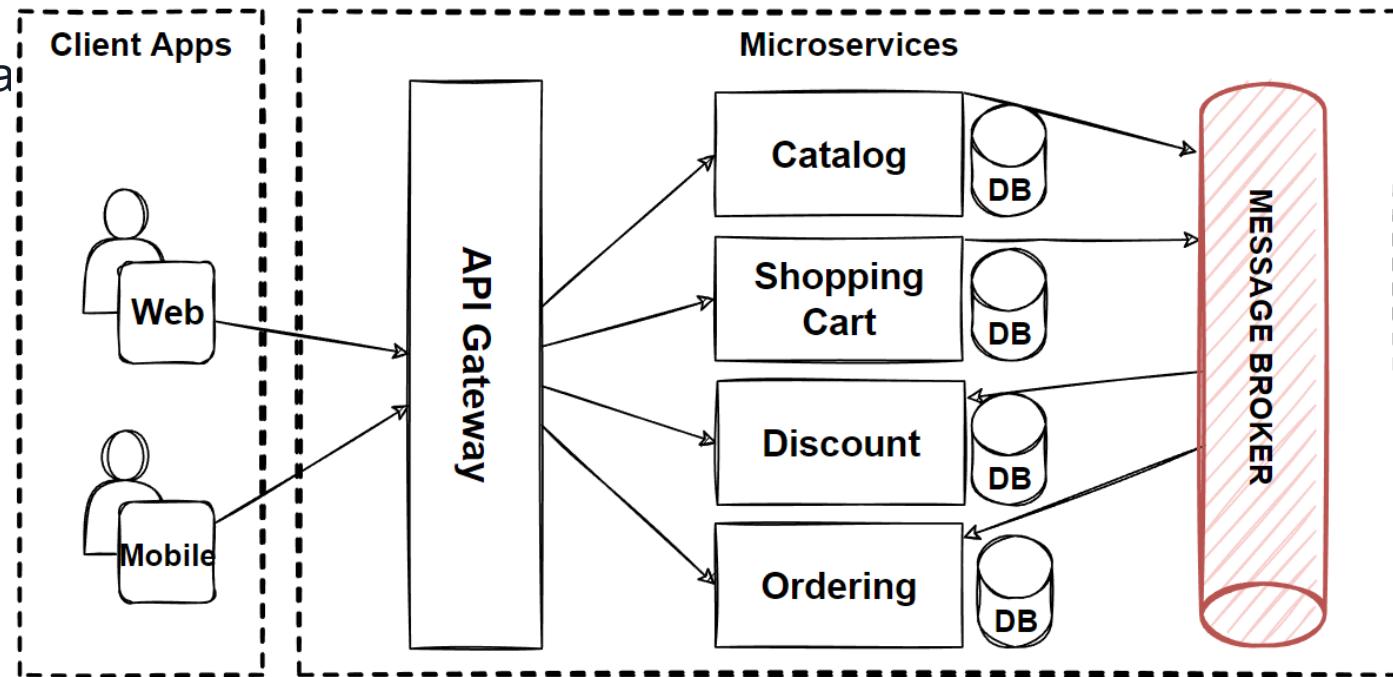


Problems

- Database are stateful service
- Scaling stateful services are not easy
- Vertical scaling has limits need to scale Horizontal
- **Different Data Requirements For Microservices**

Solutions

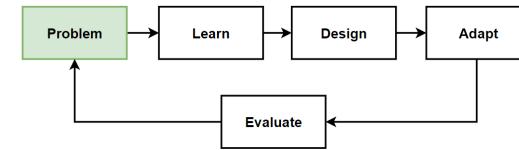
- Scale Stateful Application Horizontal Scaling
- Service and Data Partitioning along Business Boundaries - Shards/Pods
- Use NoSQL Database to gain partitioning
- **Identify Database Requirements following best practices**



Question

- **How to Choose a Database for Microservices ?**

Problem: Cross-Service Queries and Write Commands on Distributed Scaled Databases



Considerations

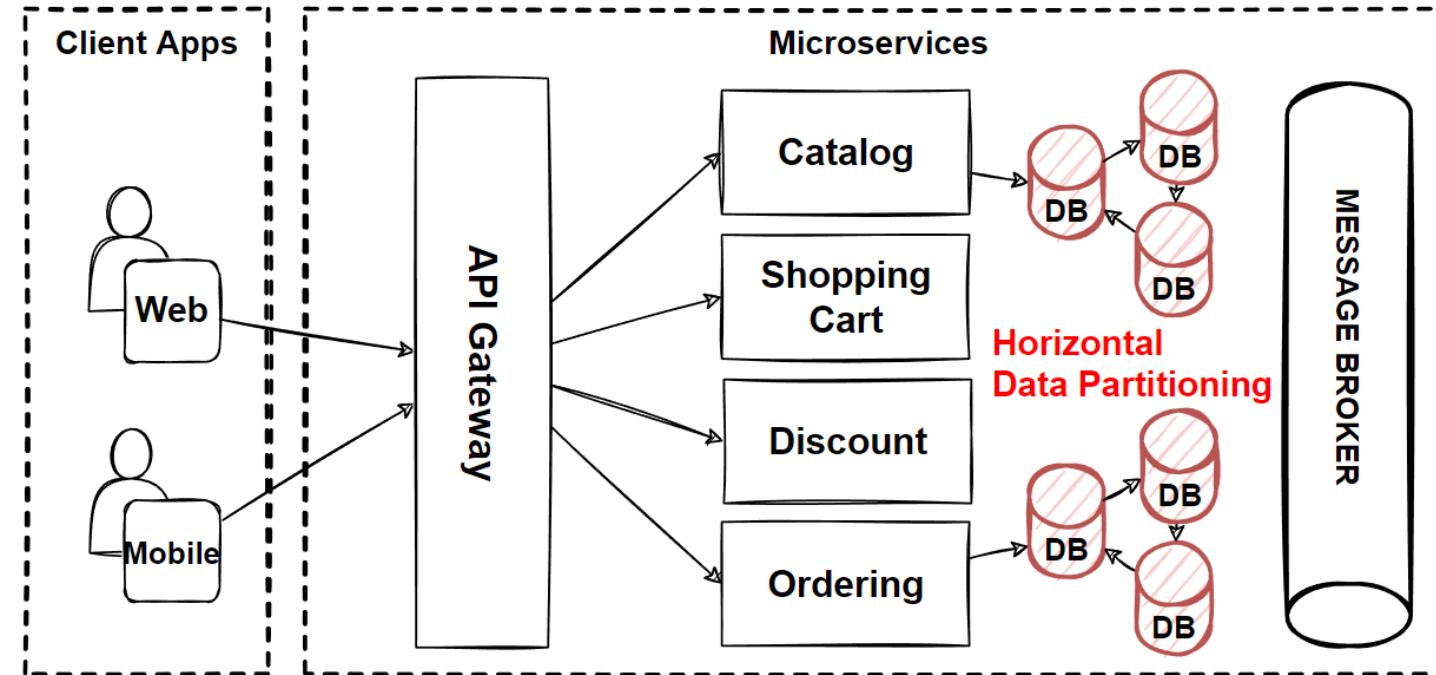
- Cross-services queries that retrieve data from several microservices ?
- Separate read and write operations at scale ?

Problems

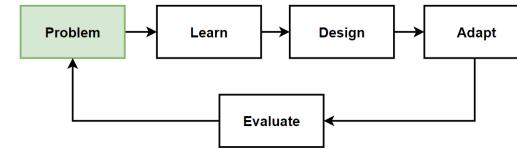
- Cross-Service Queries with Complex JOIN operations
- Read and write operations at scale
- Distributed Transaction Management

Solutions

- Microservices Data Query Pattern and Best Practices
- Materialized View Pattern
- CQRS Design Pattern
- Event Sourcing Pattern



Problem: Manage Consistency Across Microservices in Distributed Transactions



Considerations

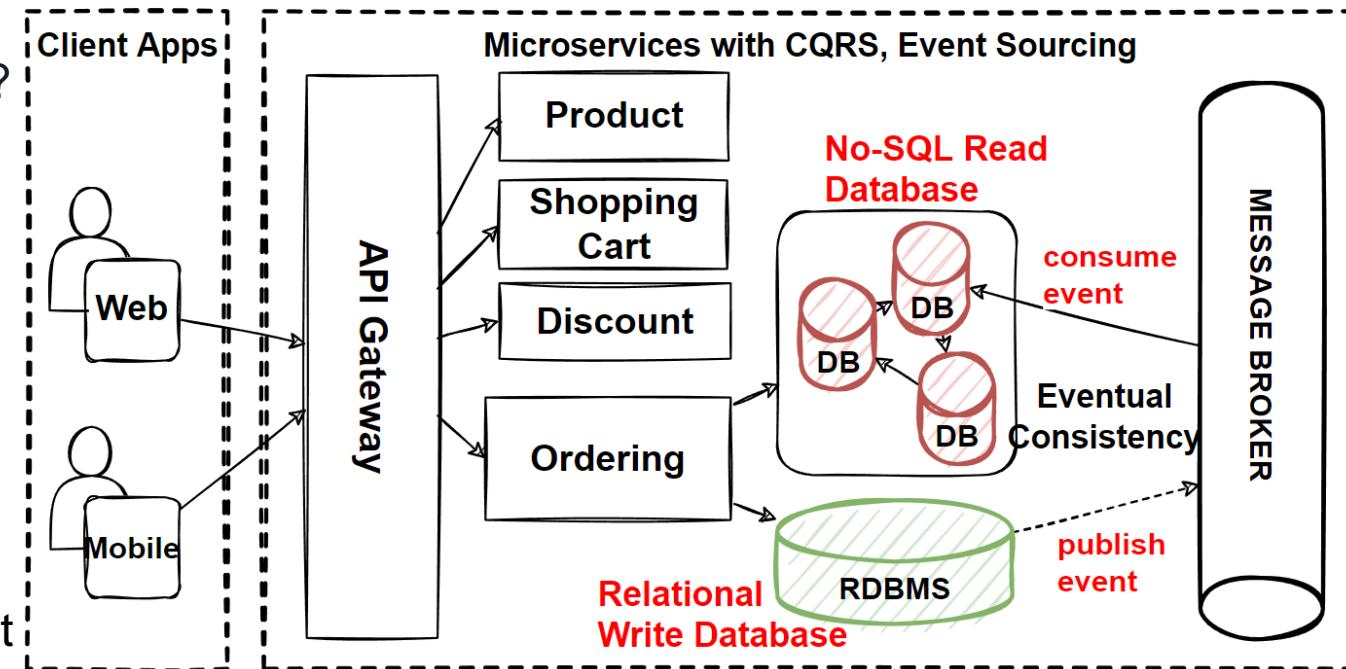
- Distributed Transactions that required to visit several microservices ?
- Consistency across multiple microservices ?
- Rollback transaction and run compensating steps ?

Problems

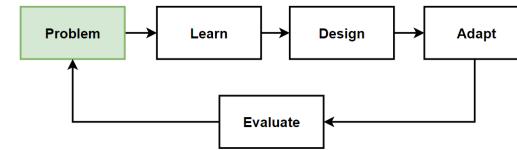
- Distributed Transaction Management
- Rollback Transaction on Distributed Environment
- Run Compensate Steps if one of service fail

Solutions

- Microservices Distributed Transaction Management Pattern and Best Practices
- Saga Pattern for Distributed Transactions
- Transactional Outbox Pattern
- Compensating Transaction pattern
- CDC - Change Data Capture



Problem: Handle Millions of Events Across Microservices

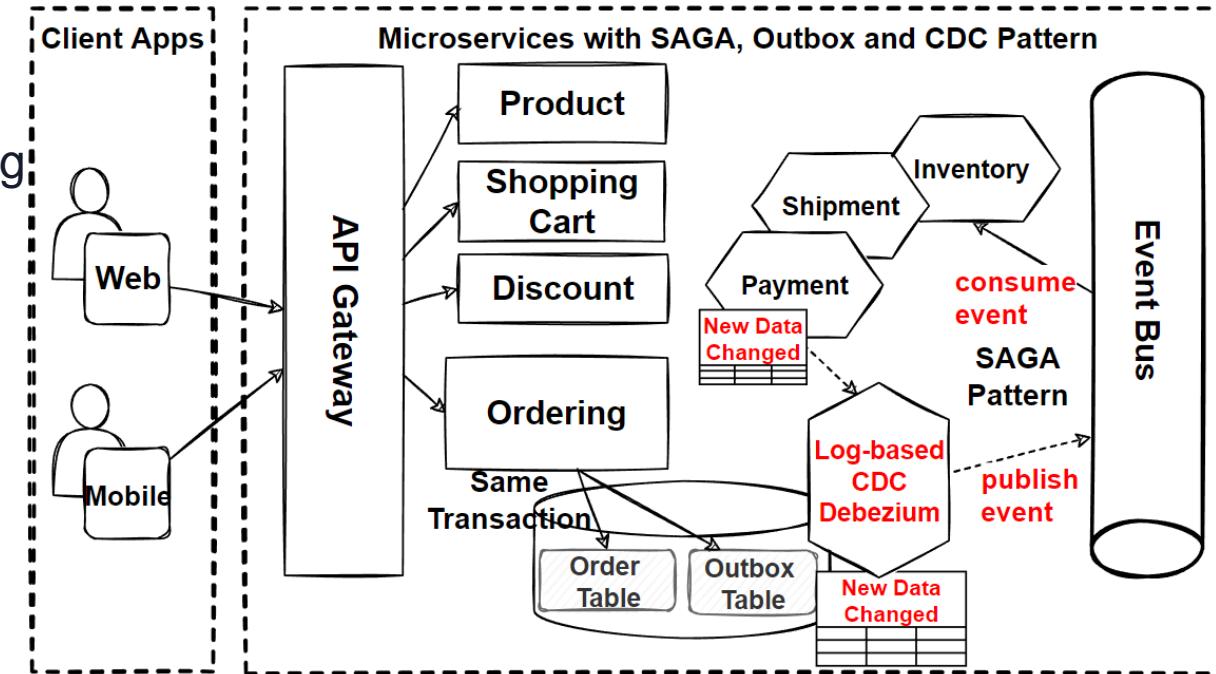


Considerations

- What if we have thousands of microservices that need to communicate with millions of events ?
- If multiple subsystems must process the same events
- Required Real-time processing with minimum latency.
- Required complex event processing, like pattern matching
- Required process high volume and high velocity of data, i.e. IoT apps.

Problems

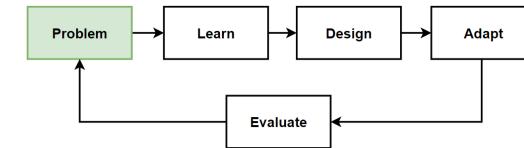
- Decoupled communications for thousands of microservices
- Real-time processing
- Handle High volume events



Solutions

- Event-driven architecture for microservices

Problem: Database operations are expensive, low performance

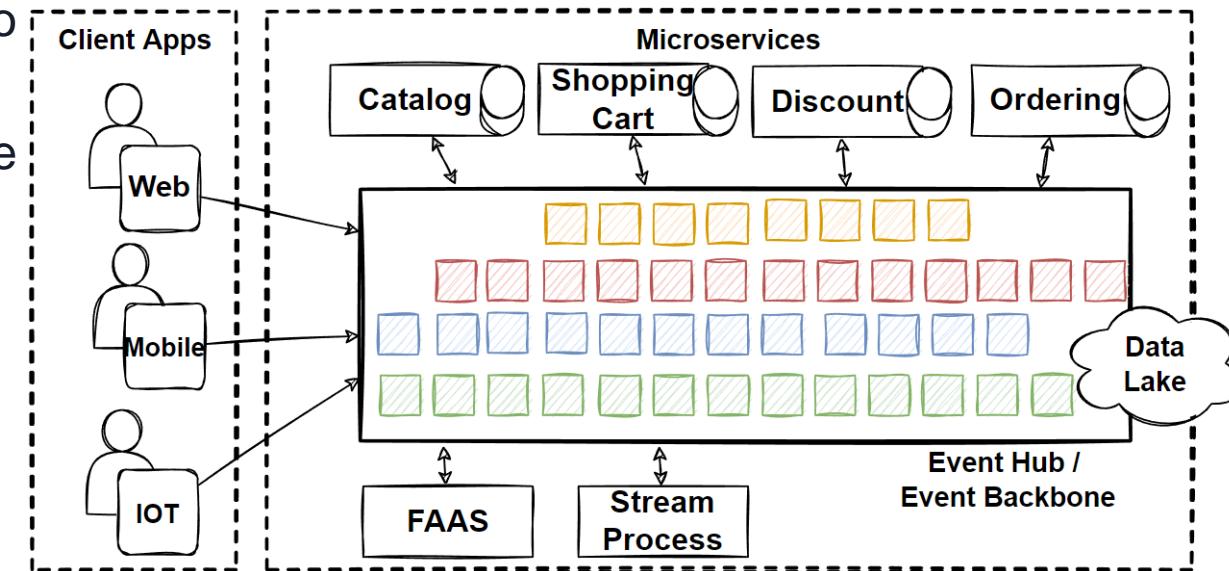


Considerations

- Event-driven architecture comes with latency when publishing and subscribing events from the Event Hub.
- Sync REST APIs communication make expensive calls to a database that reduce performance.
- How can we make more faster that increase performance of communications in Microservices Architecture ?

Problems

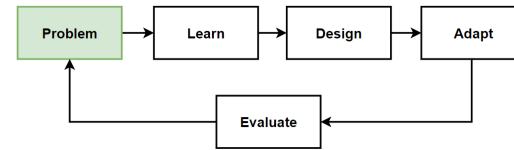
- Slowness and Low Performance Communication
- Latency when publishing and subscribing events
- Rest APIs make Database calls that are expensive, low performance



Solutions

- Distributed cache
- Storing frequently accessed data in a distributed cache

Problem: Deploy Microservices at Anytime with Zero-downtime and flexible scale



Problems

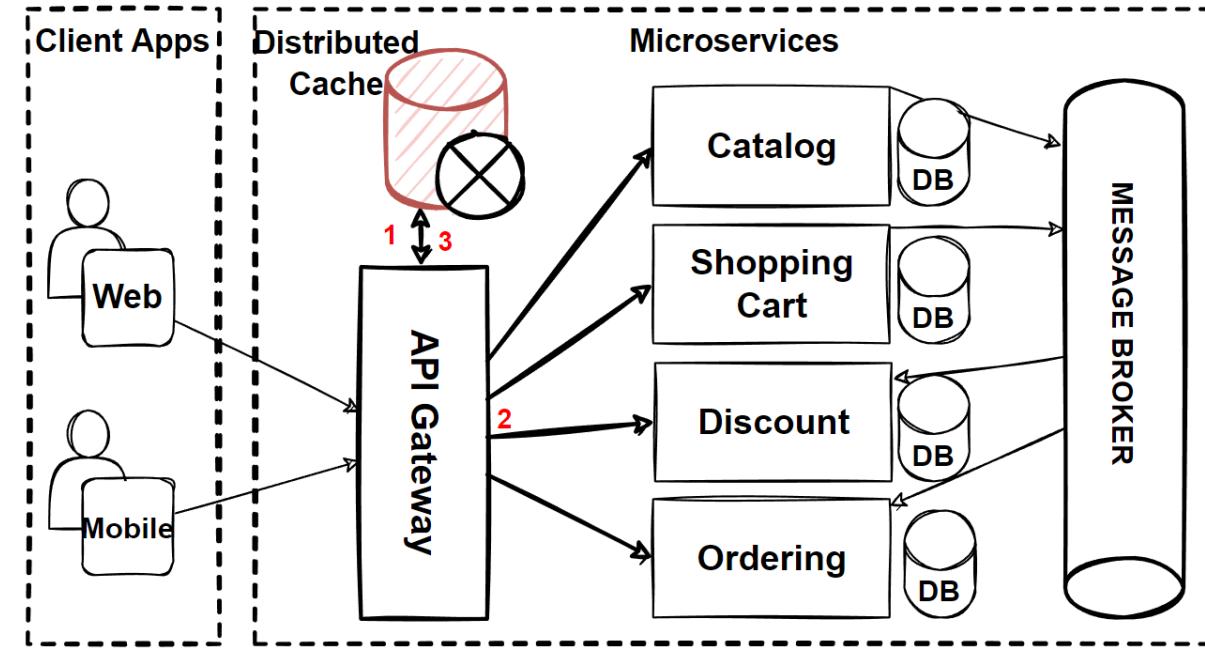
- Business teams wants to add new features immediately
- Innovate and experiment with new features
- Deploy features immediately, not waiting for deployment dates.
- Flexible scale for market peek times

Considerations

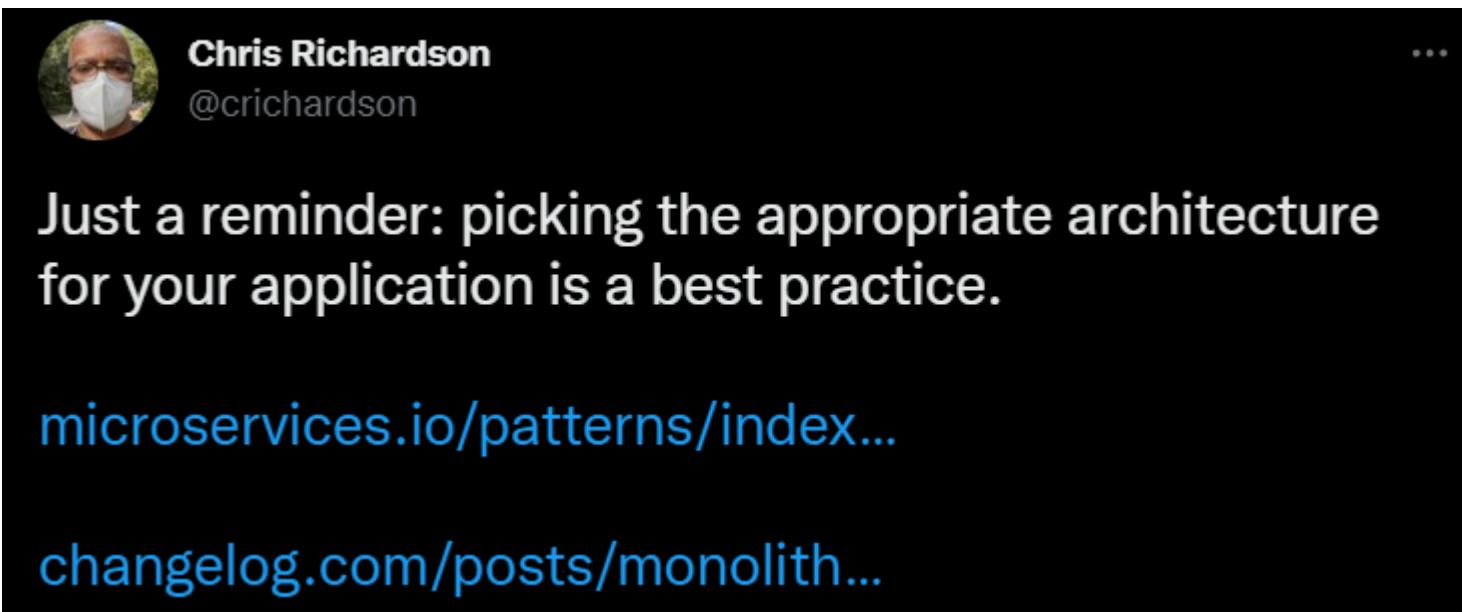
- Ensure continuity of service and minimize disruption
- Allow for continuous delivery
- Support high-traffic environments

Solutions

- Containers and Orchestrators
- Deployment strategies; blue-green deployment, rolling deployment, and canary deployment.
- Kubernetes Patterns; Sidecar Patterns, Service Mesh Pattern
- DevOps and CI/CD Pipelines and Infrastructure as code (IaC)



Choosing the Right Architecture for your Application



Chris Richardson
@crichtson

Just a reminder: picking the appropriate architecture for your application is a best practice.

microservices.io/patterns/index...

changelog.com/posts/monolith...

**Monoliths are ~~the future~~
~~a pattern~~**

“microservices are a ~~best practice~~
~~pattern~~

Choosing the Right Architecture for your Application

- [The Majestik Monolithic](#)

The image shows a composite of two social media posts. On the left is a tweet from DHH (@dhh) with a black background. The tweet text is: "Monoliths are the future because the problem people are trying to solve with microservices doesn't really line up with reality", I aspire to such savage language as that which flows from @kelseyhightower. PREACH BROTHER! 😂❤️✊. Below this is a screenshot of a Changelog.com article by Kelsey Hightower. The article title is "Monoliths are the future" and the first few sentences of the content are: "Unpopular opinion! Monoliths are the future because the problem people are trying to solve with microservices doesn't...". Both posts feature profile pictures of the respective authors.

DHH
@dhh

"Monoliths are the future because the problem people are trying to solve with microservices doesn't really line up with reality", I aspire to such savage language as that which flows from @kelseyhightower. PREACH BROTHER! 😂❤️✊

changelog.com

Monoliths are the future

Unpopular opinion! Monoliths are the future because the problem people are trying to solve with microservices doesn't...

Every Architecture has Pros and Cons

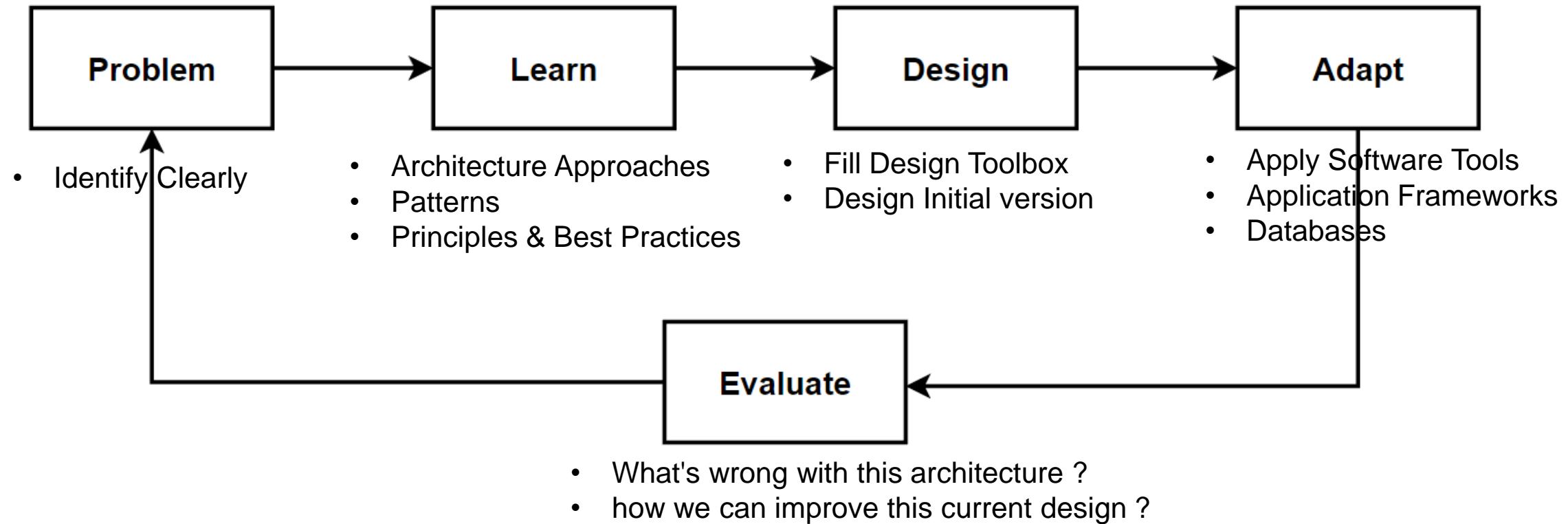
- Understand that **every architecture** has **benefits** and also **drawbacks**
- Consider every architecture style as a SA
- What is the Right Architecture for your Application?
- **It Depends..**
- Monolithic Architecture and Microservice Architecture are **architectural patterns**.
- No architecture pattern **is better than other**.
- Design your system with focusing on **context** and **non-functional requirements (-ilities)**
- **WRONG:** Microservices > Monolithic



Design for Business Requirements

- Every **design decision** must be **justified by a business requirement**.
- To avoid **over-engineering** the application architectures, keep to drive design for business requirement.
- Engineers are **tend to be over-engineering** with latest architecture styles, use latest tools.
- It **becomes** an **experimental application** that use all latest fancy tools and architectures.
- We should clearly **define functional** and **nonfunctional** requirements.
- **Define our limits, constraints and assumptions** of the application and define business objectives clearly.
- **Start and Grow Application with Metrics**
 - How many Concurrent Users that our application handle ?
 - What is the target of expected Requests/second Latency ?
 - What level of application outage is acceptable?
- **Business requirements drive these design considerations.**

Way of Learning – The Course Flow



How to Follow the Course

- I strongly recommended that you should take this course from **beginning to end**.
- If you already **familiar some architectures**, **jump into** your target architectures and start to learn from that section.
- All **sections are independent** from each other and you can easily **switch on sections** with following different architectures.
- Every section **starts with specific problem** and **solve this problem** with learning new **patterns-principles**.
- If you are only interested in microservices architectures, skip first monolithic, layered, clean and modular monolithic architectures and **jump to microservices** architectures.
- You can skip some of Microservices **vertical topics** as per your experience and requirements. If you don't interest **Decomposition**, skip "Microservices Decomposition" and continue with "**Microservices Communications**" sections.
- **Jump your problem and learn patterns & principles** related that problem to **design** your **final architecture**.
- This is **reference architecture course** that you can take any part of the course according to your architectural requirements.

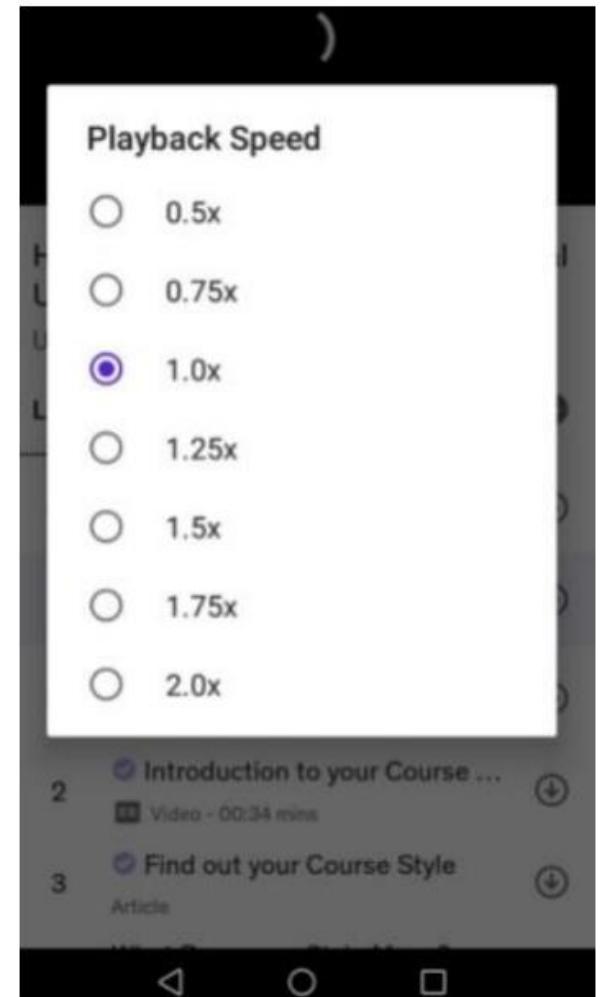
How to Follow the Course - 2

- **Increase Speed**

If you feel comfortable on any particular topic, please increase the video speed to avoid losing motivation of the course.

- **Put a Review**

Please put a comment and review the course, when you feel ready at any time of the course, this will help me a lot for further courses.



Course Slides

- **Powerpoint Slides**

Find full PowerPoint slides the link in the resource of this video.

The screenshot shows a course slide interface. At the top, it says "Section 1: Introduction" and "1 / 9 | 39min". Below this, there is a list of topics:

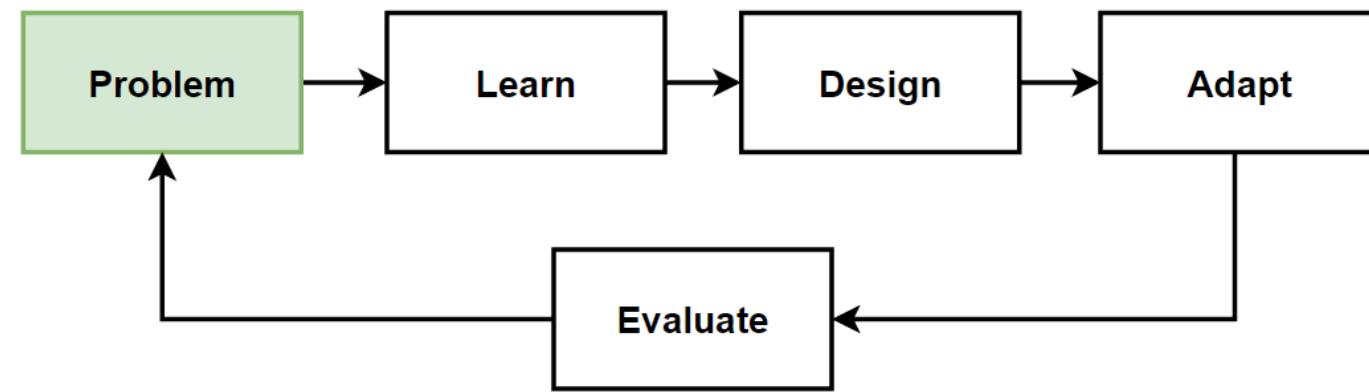
- 1. Introduction (7min) Resources
- 2. Prereq... Microservices Architecture on...

Understand E-Commerce Domain

Our First Problem: Sell Products Online. To Solve this problem, we should understand our domain which is E-Commerce.

Problem: Sell Products Online

- Create E-Commerce Web Application
- Identify use cases and non-functional requirements
- List products, add basket and ordering products.
- Available 7/24
- Handle good amount of request per second
- Provide acceptable latency for users



Understand Problem

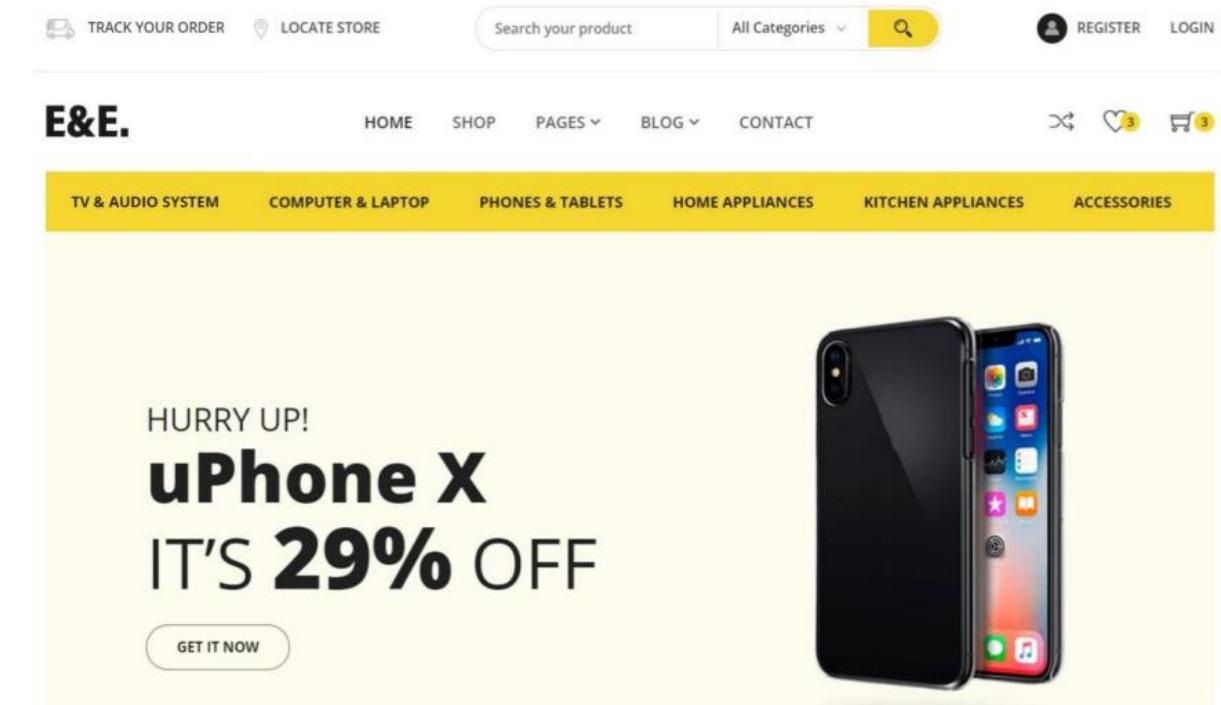
- Understand E-Commerce Domain
- Functional & non-functional requirements
- Use cases

Understand E-Commerce Domain

- Our Domain: E-Commerce
- Understand Domain and Decompose small pieces
 - Use Cases
 - Functional Requirements

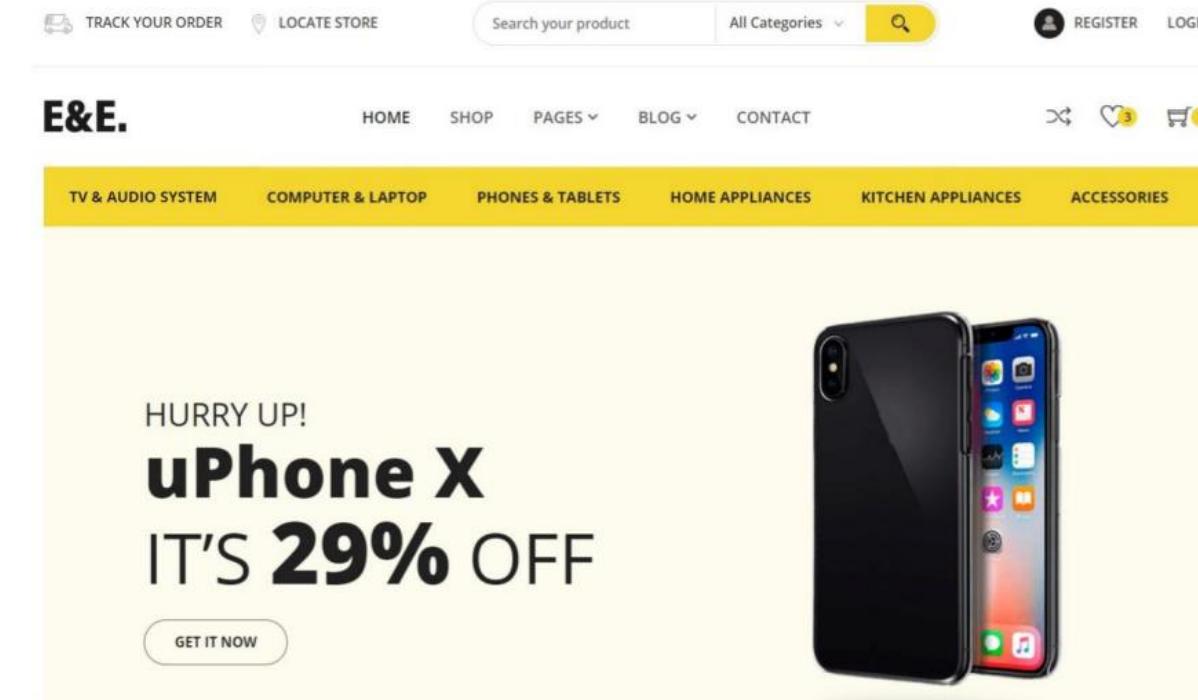
Identify steps:

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



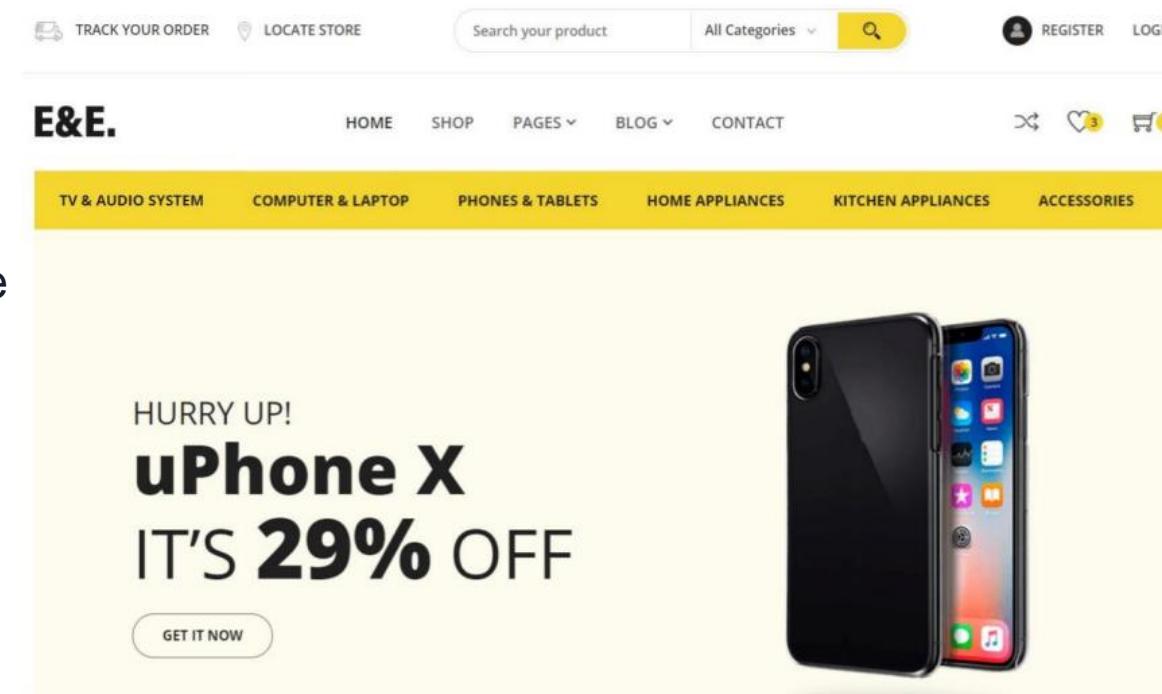
Understand E-Commerce Domain: Functional Requirements

- 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



Understand E-Commerce Domain: User Stories (Use Cases)

- 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



Understand E-Commerce Domain: Non-Functional Requirements

- ilities

Scalability

Availability

Reliability

Maintability

Usability

Eficiency

Request per Second and Acceptable Latency

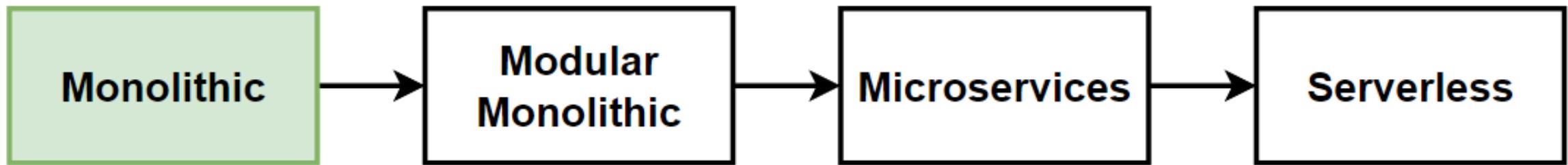
Concurrent Users	Requests/second	Latency (Expected)
2K	0.5K	
20K	12K	
100K	80K	<= 2 sec
500K	300K	?

Monolithic Architecture

Benefits and Challenges of Monolithic Architecture.

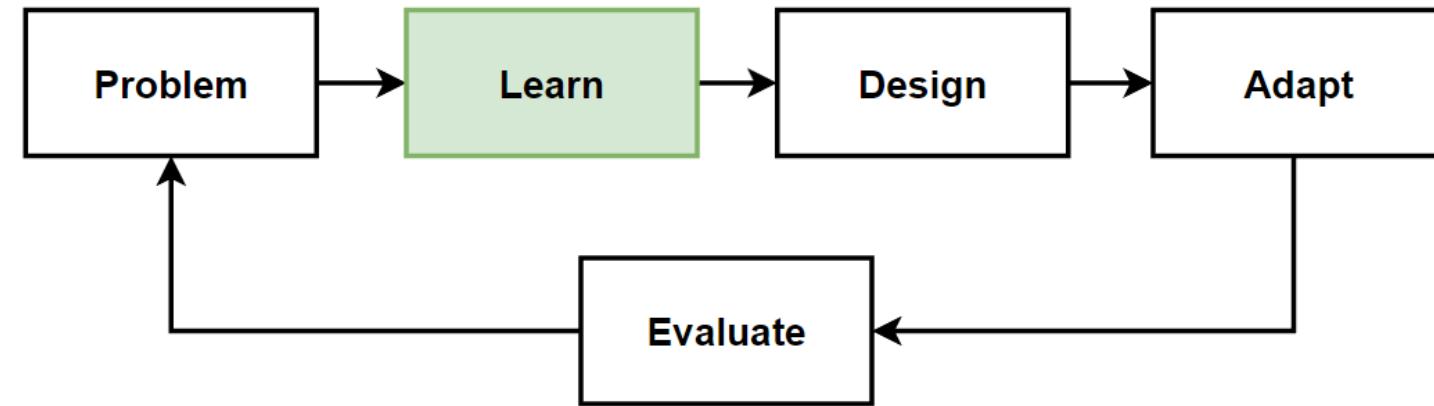
Design our E-Commerce application with Monolithic Architecture

Architecture Design Journey



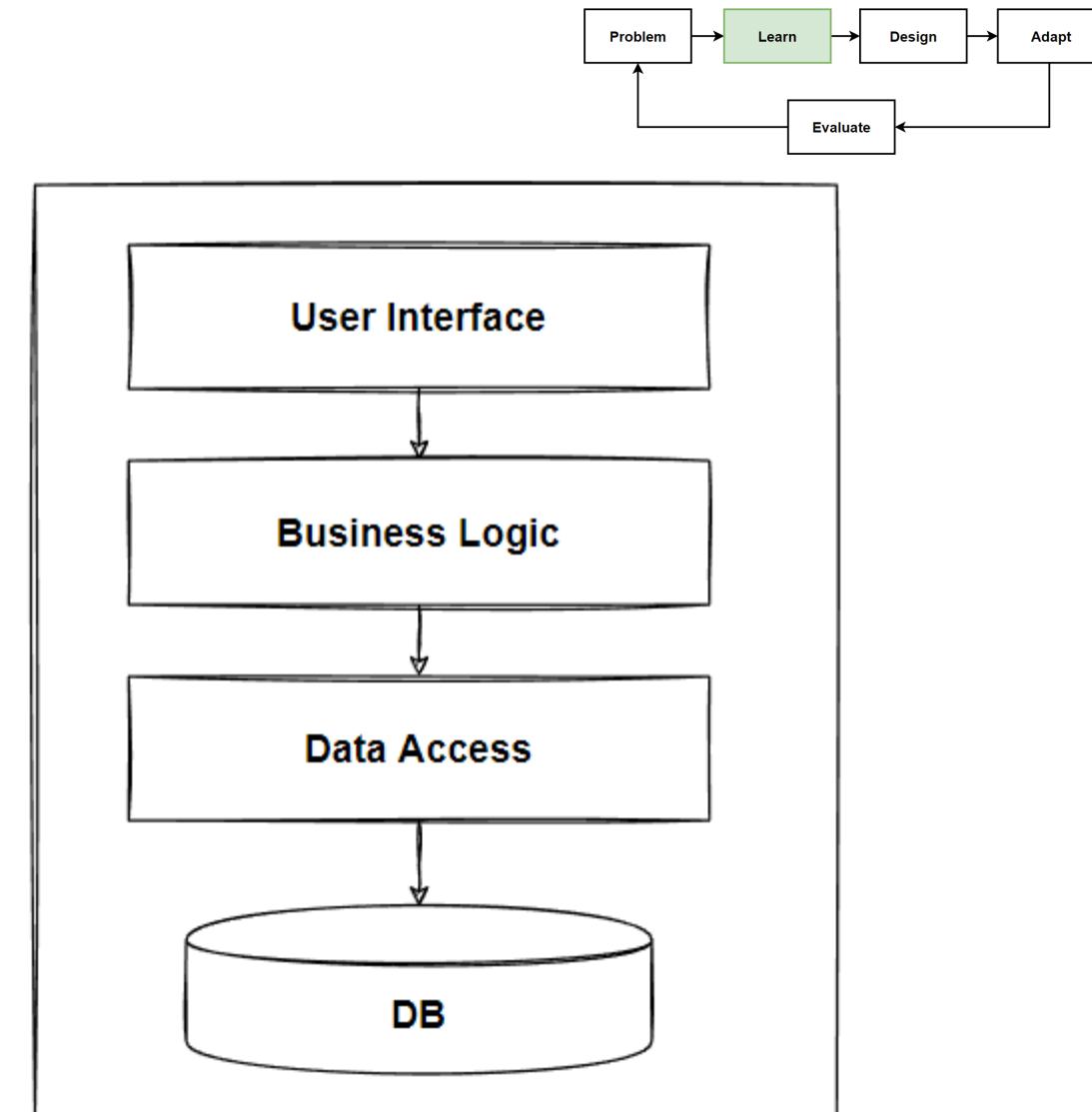
Learn: Monolithic Architecture

- Monolithic Architecture
- When to use Monolithic Architecture
- Benefits of Monolithic Architecture
- Challenges of Monolithic Architecture
- Monolithic Architecture Pros-Cons
- Reference Architectures of Monolithic



What is Monolithic Architecture ?

- Traditional approach to software development
- Developing a complete application as a **single unit**
- Most of legacy applications are mainly implemented as a monolithic architecture
- Developed **single codebase**
- UI, Business and DB calls is in **same codebase**
- All application ships with **single big deployment** with single jar/war file
- Can't say old style arcitecture, **still valid** for particular scenarios
- **Advantages:** Easy to get start and debug
- **Disadvantages:** Difficult to manage, Hard to implement new features

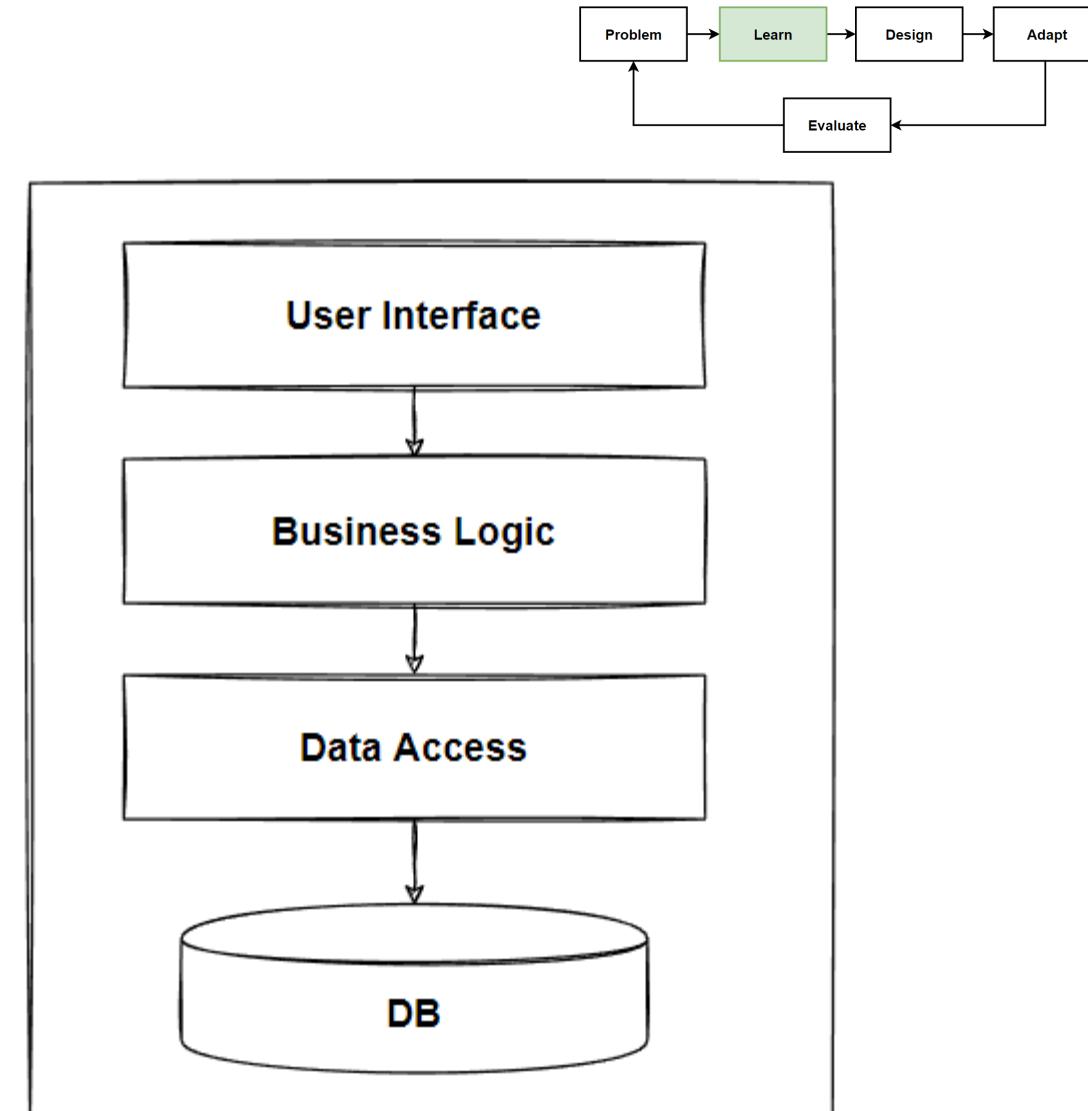


When to use Monolithic Architecture

- If you are **building small application**, still **monolithic architecture** is **one of the best architecture**.

They're straightforward to:

- Build
- Test
- Deploy
- Troubleshoot
- Scale vertically (scale up)
- **Simple to develop** relative to microservices
- **Easier to deploy** as only a single jar/war file



When to use Monolithic Architecture

- **Small team at Founding Stage**

If you are a startup and your team is small like 2 to 5 members, you don't need to deal with the complexity of the high-overhead microservices architecture.

- **Simple application with Predictable Scale and Complexity**

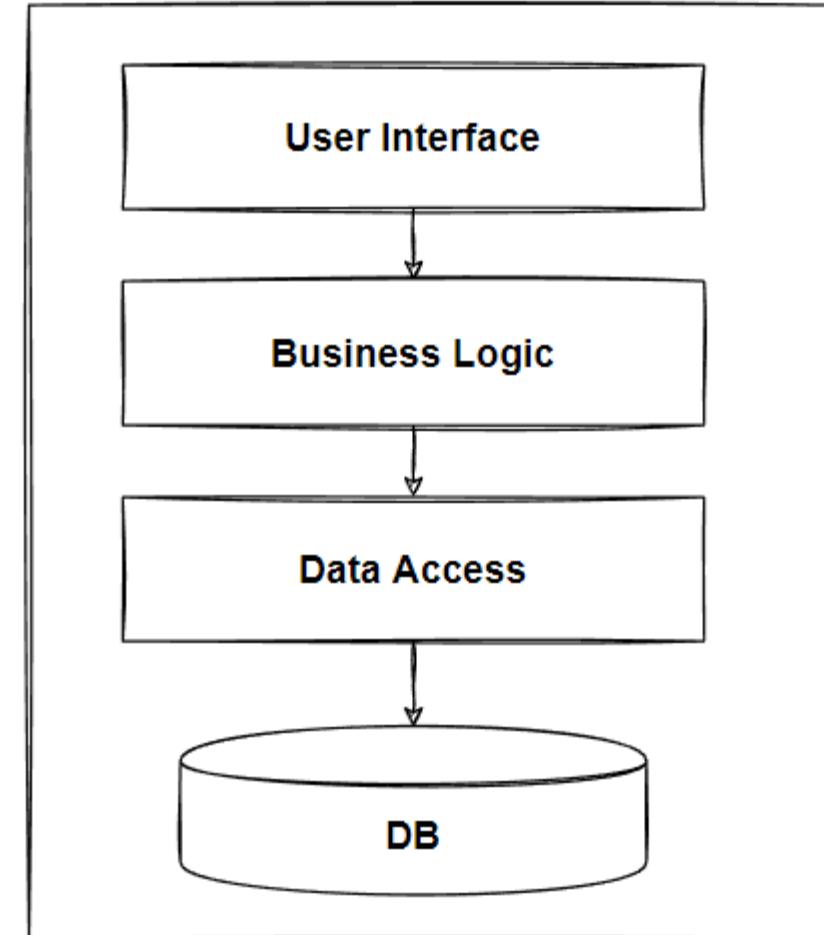
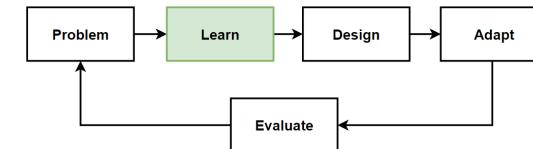
If your application doesn't require advanced scalability and the complexity is manageable, then a monolith architecture is the best option to start.

- **Proof of Concept and Quick Launch**

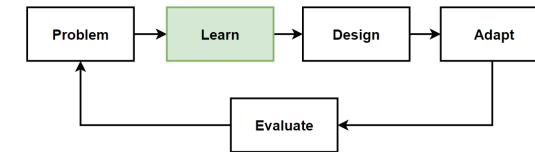
Building a proof of concept, like testing a new idea on market, that means your new products will pivot and evolve a lot over time, when you figure out what will be useful to your users.

- **No Microservices Expertise**

If your team has no prior experience with microservices architecture, that will really hard to ship your application effectively and timely.



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**

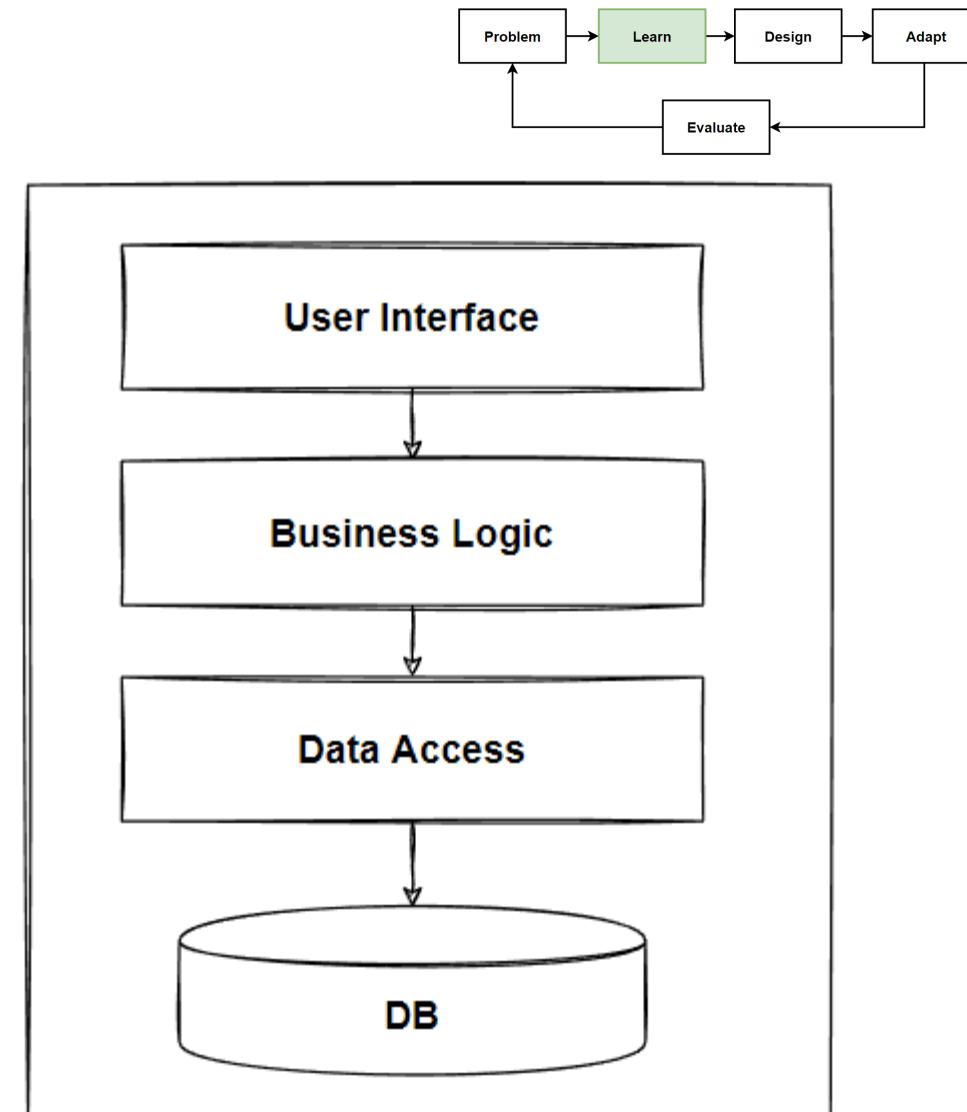
As long as the monolithic approach is a standard way of building applications, any engineering team has the right knowledge and capabilities to develop a monolithic application.

- **Easier debugging and testing**

Monolithic applications are much easier to debug and test. Since a monolithic application has a single code base, we can run end-to-end testing much faster.

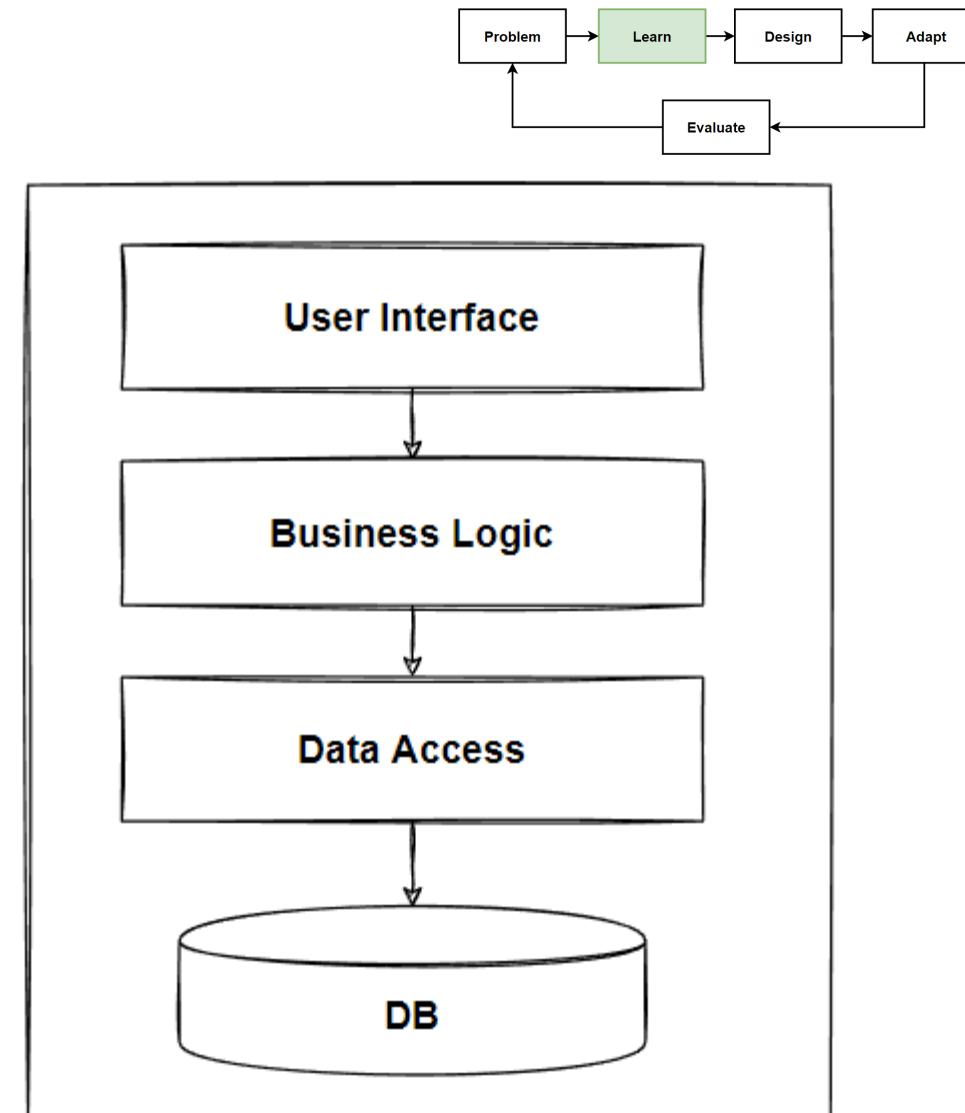
- **Simple to deploy**

When it comes to monolithic applications, you do not have to handle many deployments, just one file or directory. Easier to deploy as only a single jar/war file is deployed.



Challenges of Monolithic Architecture

- **Become Complex over time - Hard to Understand**
It becomes too large in size with time and that's why its difficult to manage. Application grows with adding new functionalities, a monolithic codebase can become extremely large and complex.
- **Hard to Making New changes**
It is harder to implement new changes in such a large and complex application with highly tight coupling. Any code change affects the whole system.
- **Barrier to new technology adoption**
It is extremely problematic to apply a new technology because the entire application has to be re development due to the interlocking dependencies found in a monolith.
- **Difficult to Scale**
You can't scale components independently, the only option is the scaling the whole application. You can't scale individual components.



Learn: Design principles - KISS, YAGNI, DRY

- **DRY – Don’t Repeat Yourself**

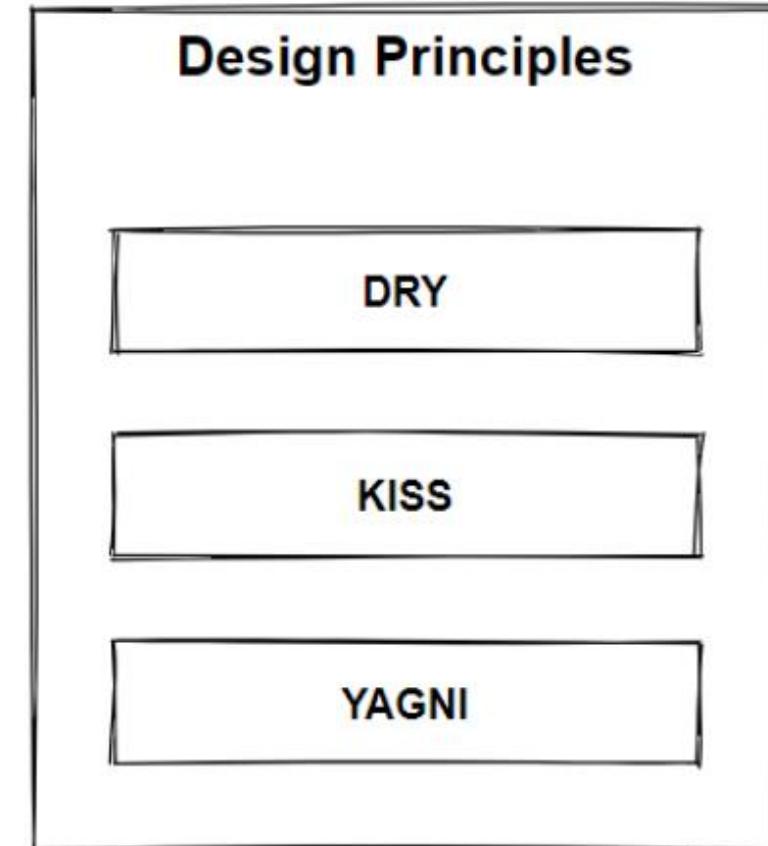
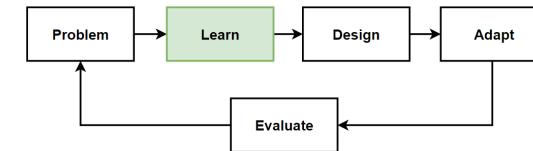
Every piece of knowledge must have a single, unambiguous, authoritative representation within a system. Try to maintain the behavior of a functionality of the system in a single piece of code, it should not have duplicated code or design item.

- **KISS – Keep It Simple, Stupid**

Make your code or system simple. You should avoid unnecessary complexity. A simple code it's easier to maintain and easier to understand.

- **YAGNI – You Ain’t Gonna Need It**

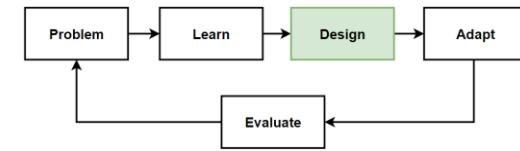
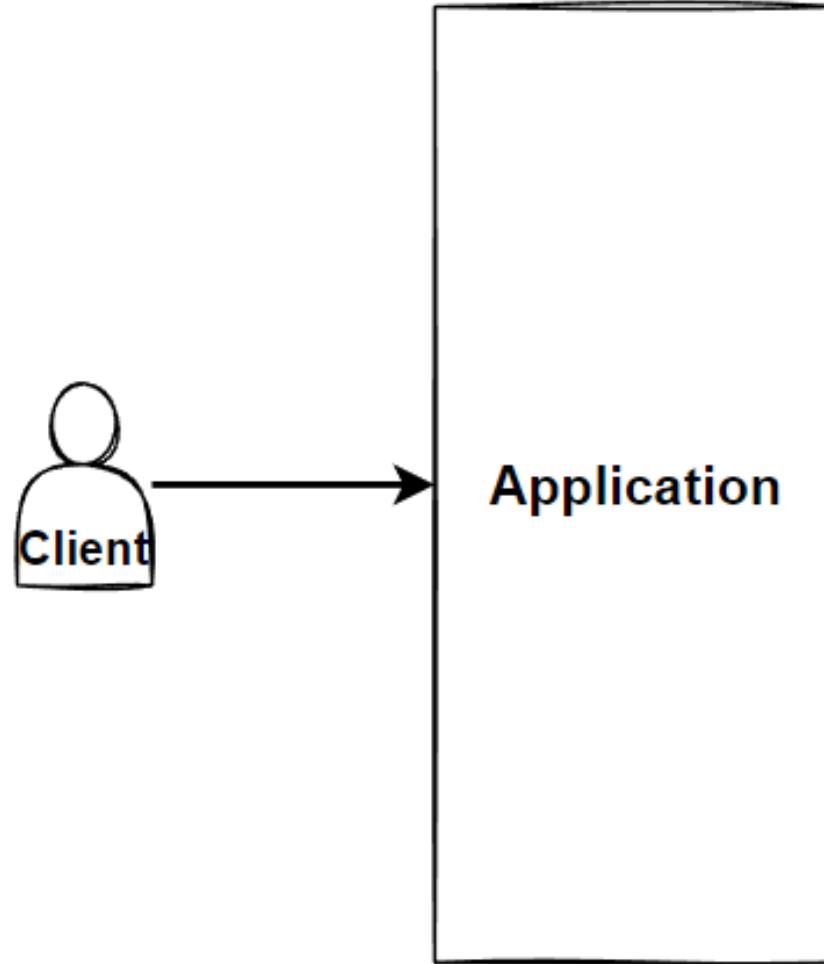
This methodology said “Do the Simplest Thing That Could Possibly Work”. This principle says that you should not create features that it's not really necessary.



Before Design – What we have in our design toolbox ?

Architectures	Patterns&Principles	Non-FR	FR
<ul style="list-style-type: none">• Monolithic Architecture	<ul style="list-style-type: none">• DRY• KISS• YAGNI	<ul style="list-style-type: none">• Availability• Small number of Concurrent User	<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history

Design: First version of Monolithic Architecture

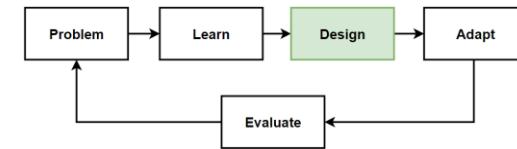
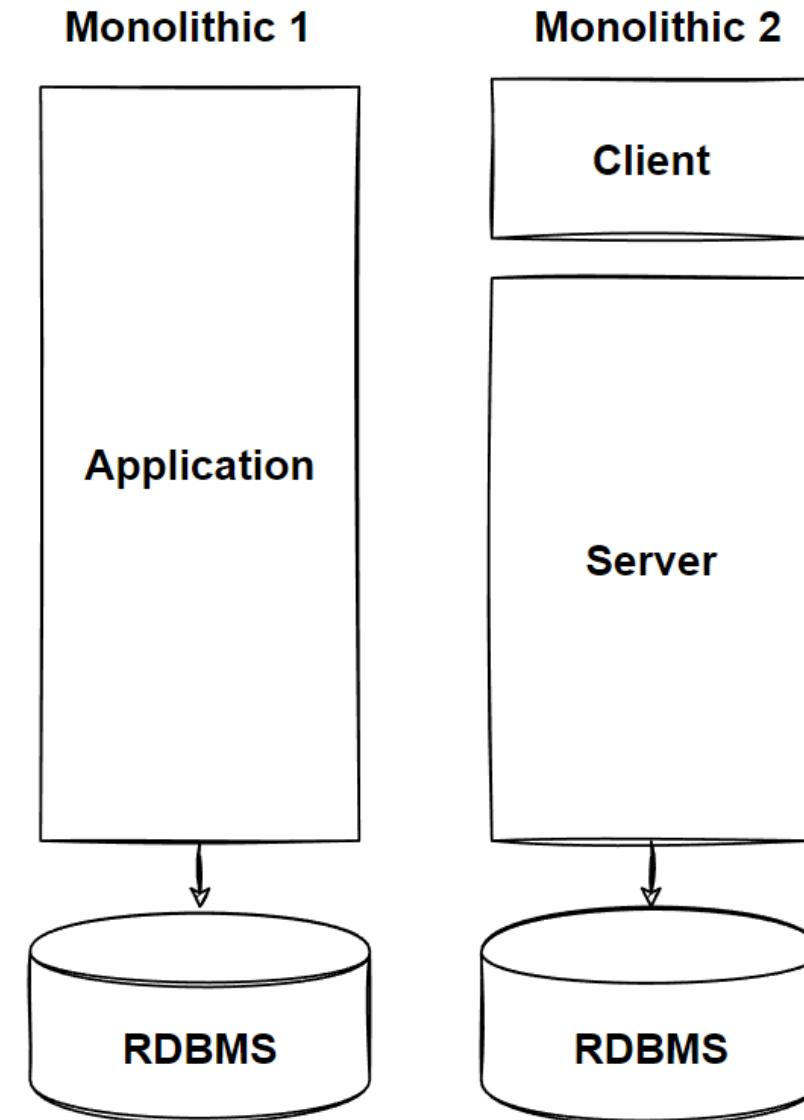


Before Design – What we have in our design toolbox ?

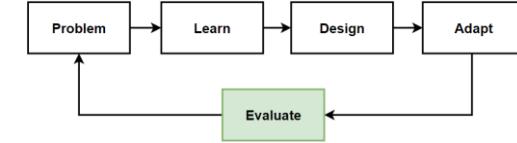
Architectures	Patterns&Principles	Non-FR	FR
<ul style="list-style-type: none">• Monolithic Architecture	<ul style="list-style-type: none">• DRY• KISS• YAGNI	<ul style="list-style-type: none">• Availability• Small number of Concurrent User	<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history

Design & Iterate : Monolithic Architecture

- Add Database into our design
- Separate Client application



Evaluate : Monolithic Architecture - E-Commerce App

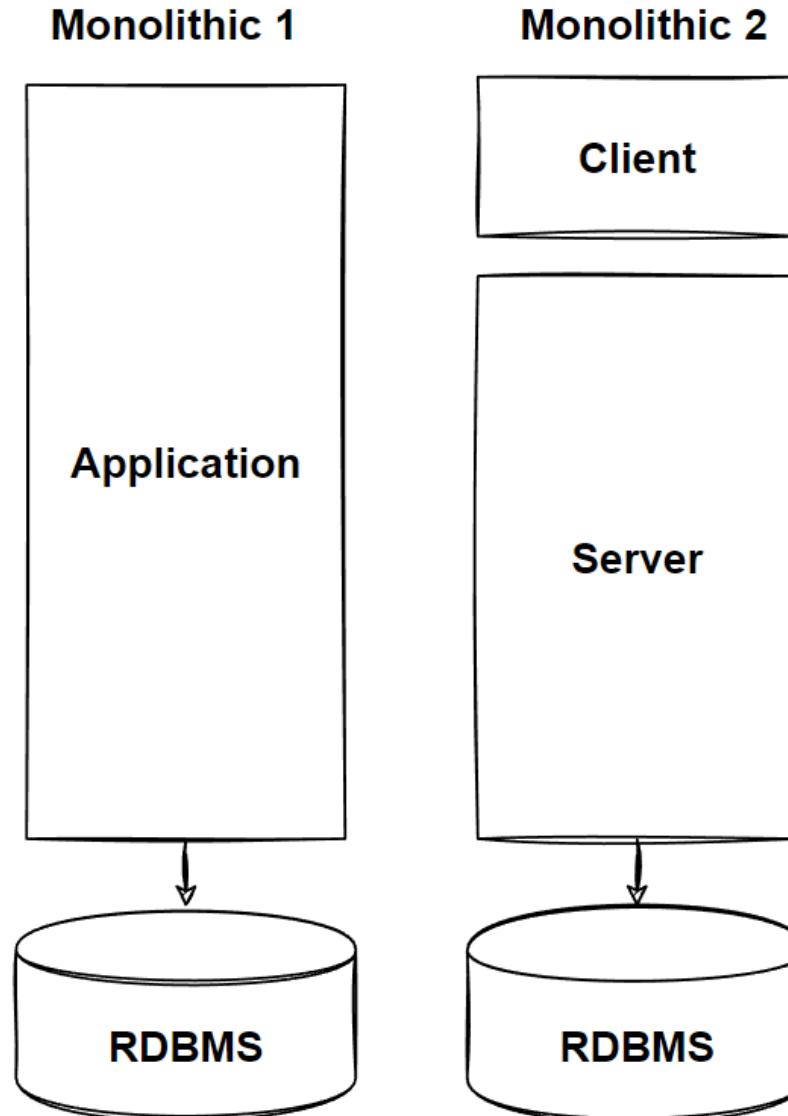


Benefits

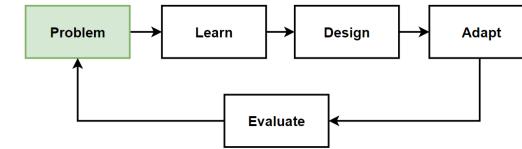
- Easy Development
- Easy Debug and Test
- Easy to Deploy

Drawbacks

- Highly tight coupling
 - Hard to Splitting the code
 - Violate Separation of concerns
 - Interlocking Dependencies
- without Layers of isolation



Problem: Code Become Too Complex Over Time

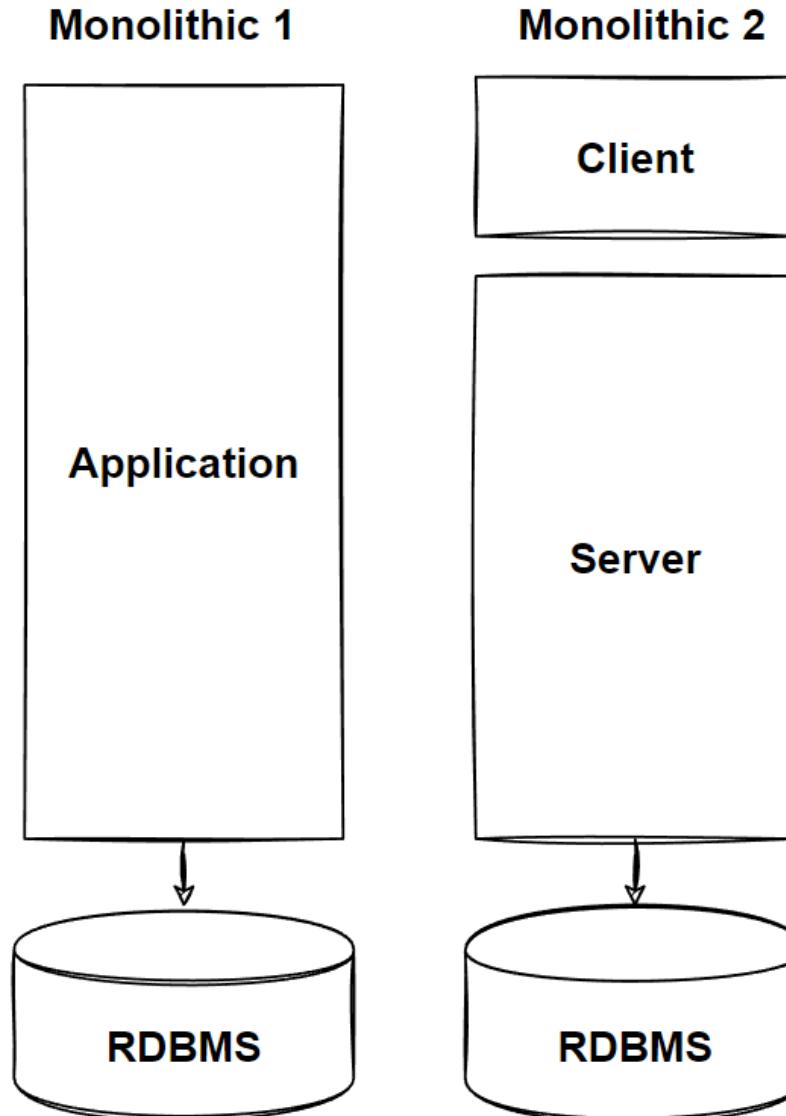


Problems

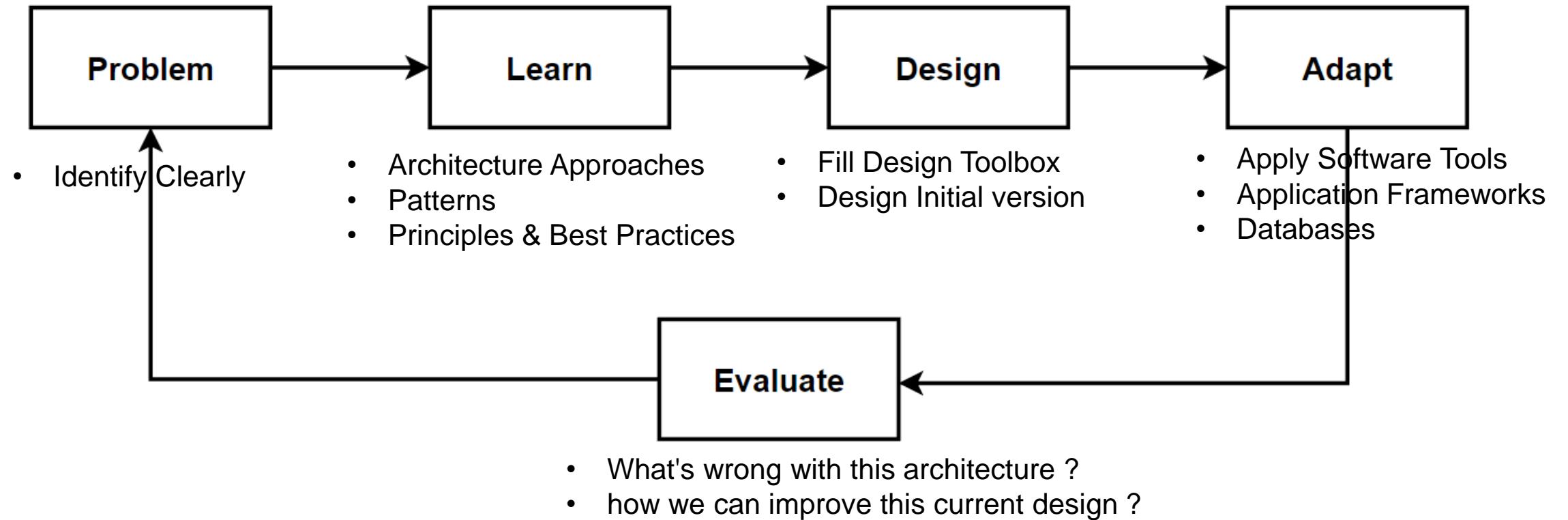
- Become Complex over time
- Hard to Understand Codes
- Need Code Organization

Solutions

- Separate UI, Business and Data Layers as logical layers
- Layered Architecture
- SOLID Design



Way of Learning – The Course Flow

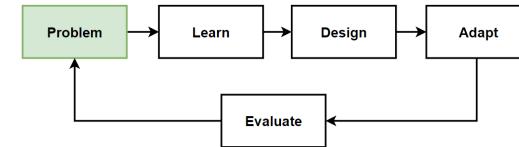


Layered (N-Layer) Architecture

Benefits and Challenges of Layered (N-Layer) Architecture

Design our E-Commerce application with Layered Monolithic
Architecture

Problem: Code Become Too Complex Over Time

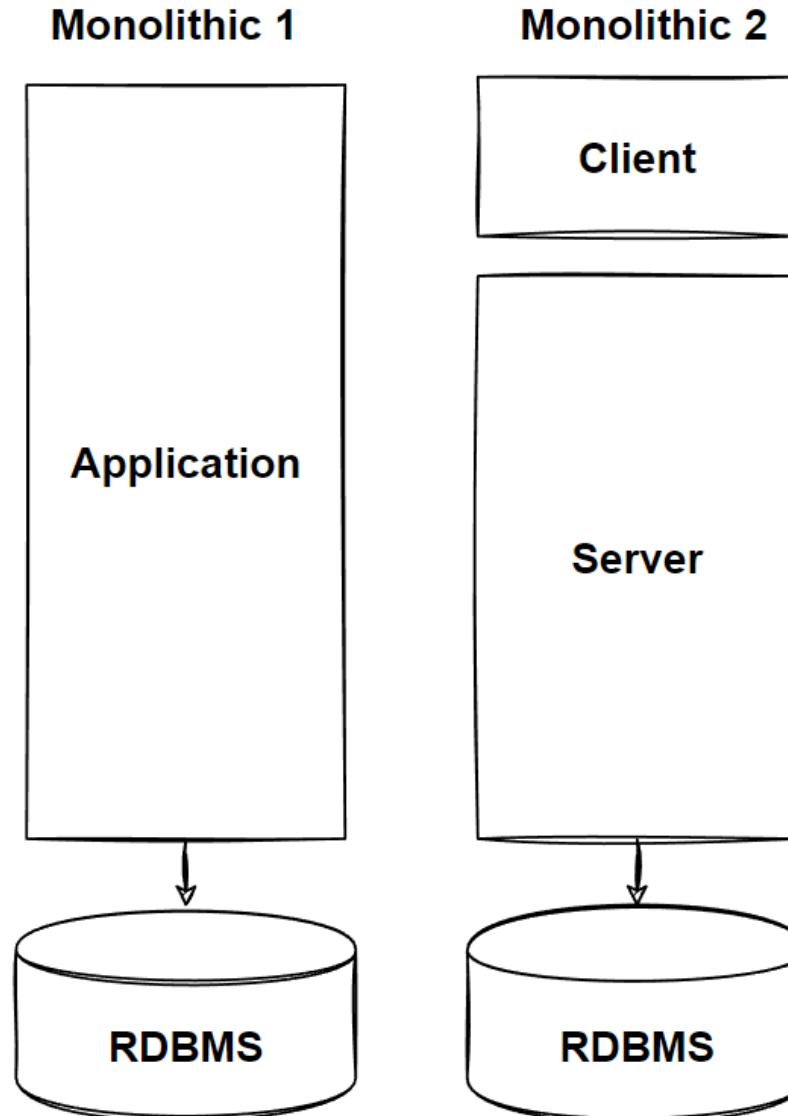


Problems

- Become Complex over time
- Hard to Understand Codes
- Need Code Organization

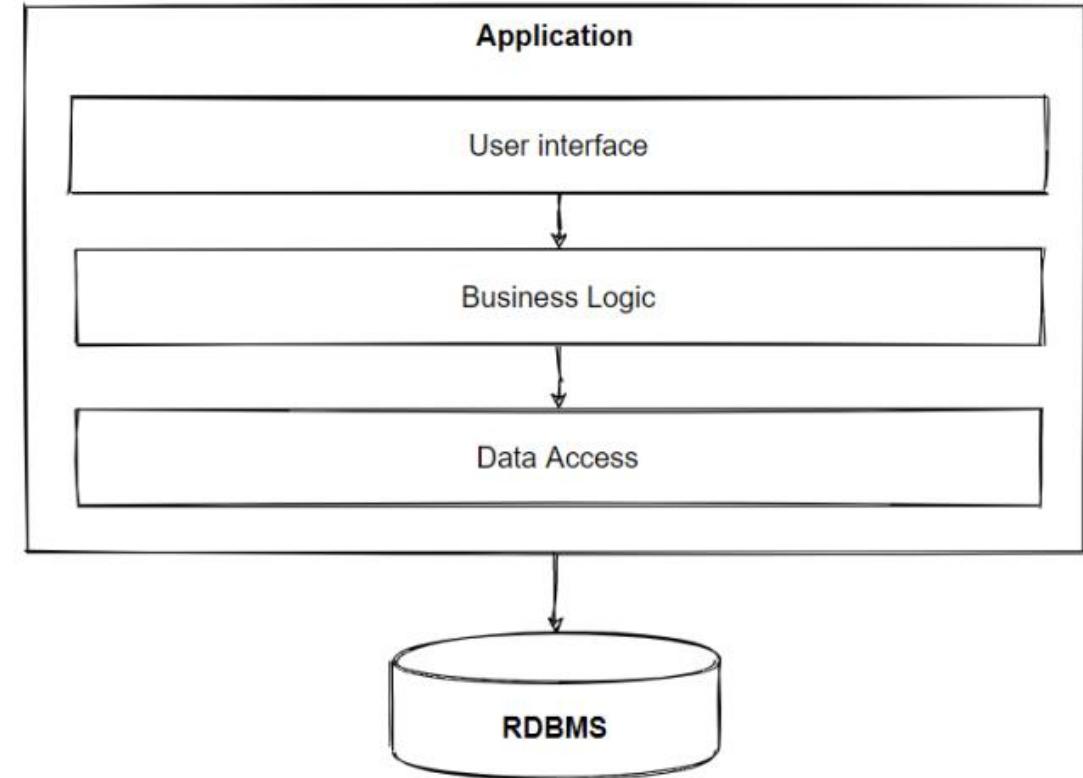
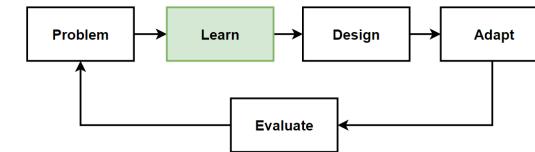
Solutions

- Separate UI, Business and Data Layers as logical layers
- Layered Architecture
- SOLID Design



Learn: Layered (N-Layer) Architecture

- The layered architecture pattern is the most commonly used architecture pattern. Known as the **n-tier architecture style** or the **multi-layered** architecture style.
- Organize the components of an application with similar functionalities into **horizontal logical layers**. Each layer performs a specific role within the application.
- Still using Monolithic architecture separating horizontal logical layers, components are interconnected but **don't depend** on each other.
- Organizing code for **separation of concerns**
- **Layers of isolation** that layers can be modified and the change won't affect other layers.



Components of a Layered Architecture

- **Presentation Layer**

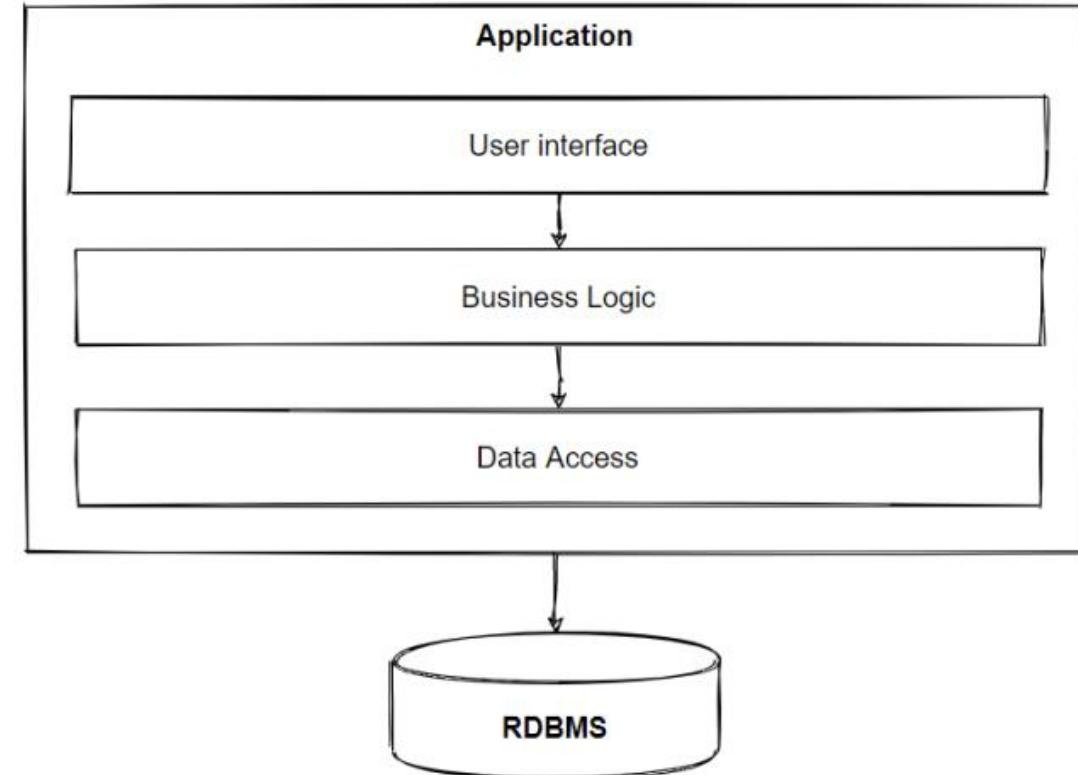
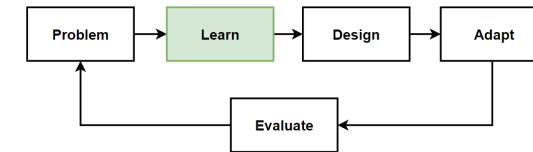
Responsible for user interactions with the software system, for example, a web app.

- **Application/Business Layer**

Handles aspects related to accomplishing functional requirements including use case implementations.

- **Database Layer**

Responsible for handling data, databases, such as a SQL database.



Design principles - Separation of Concerns (SoC)

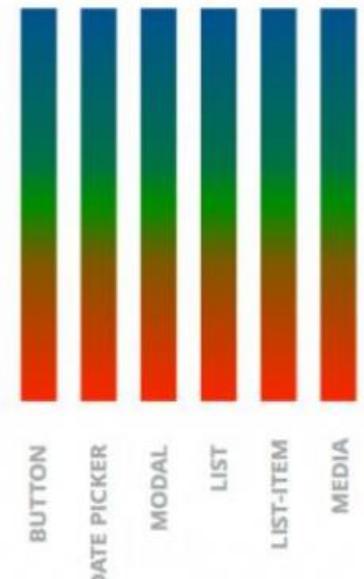
- **Separation of concerns(SOC)** is one of the core software design principle
- Separation of concerns is a design principle for separating a computer program into **distinct sections**
- **Isolate** the software application into separate sections, **manages complexity** by partitioning the software system
- **Distinguish between the concepts** of layer and tiers with certain responsibilities.
- Elements in the software should be **unique**
- Limits to allocate responsibilities
- Low-coupling, high-cohesion

Separation of Concerns



Separation of Concerns

(only, from a different point of view)



Design principles - SOLID

- **Single Responsibility**

Each of your components or modules should responsible only one functionality.

- **Open-Closed Principle**

When we design the system, it should able to extend without changing existing architecture.

- **Liskov Substitution Principle**

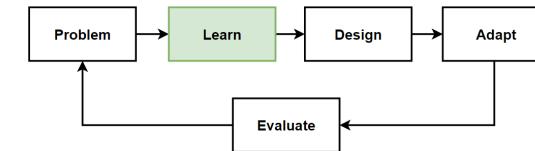
Systems can be substitute each other easily. In our case we can use plug-in services that we can shift them easily.

- **Interface Segregation Principle**

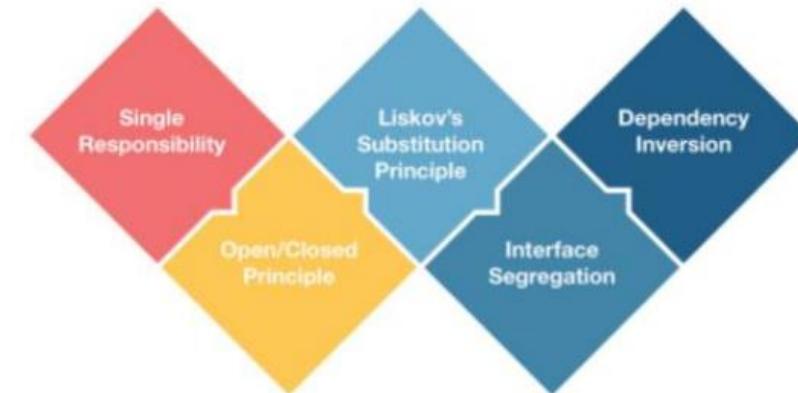
States that no code should be forced to depend on methods it doesn't use.

- **Dependency Inversion Principle**

States that high-level modules should not depend on low-level modules; both should depend on abstractions.



S.O.L.I.D.

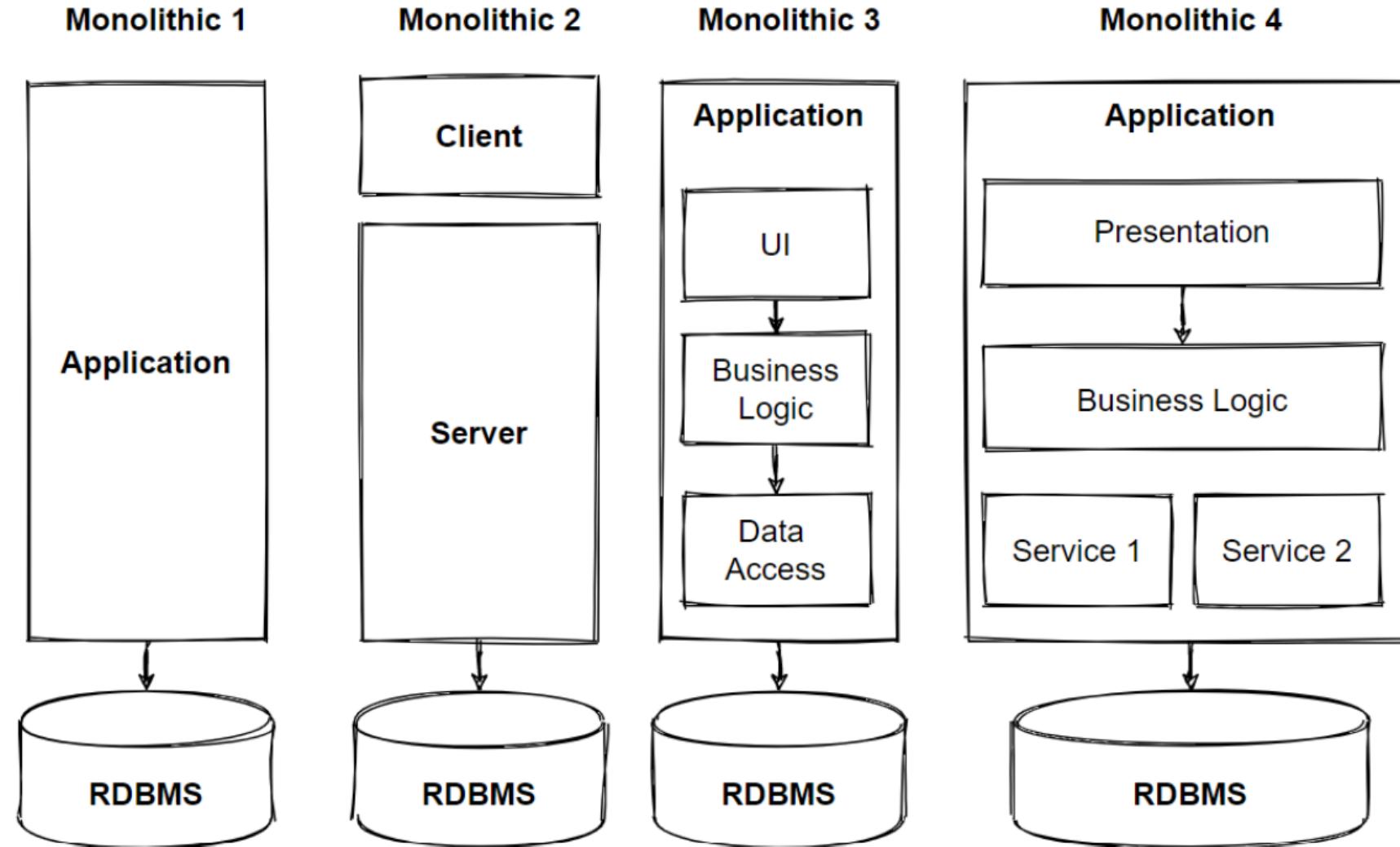
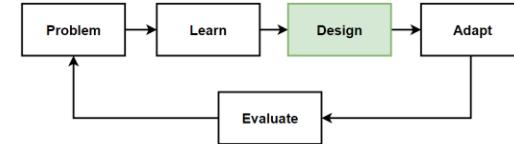


<https://medium.com/bgl-tech/what-are-the-solid-design-principles-c61feff33685>

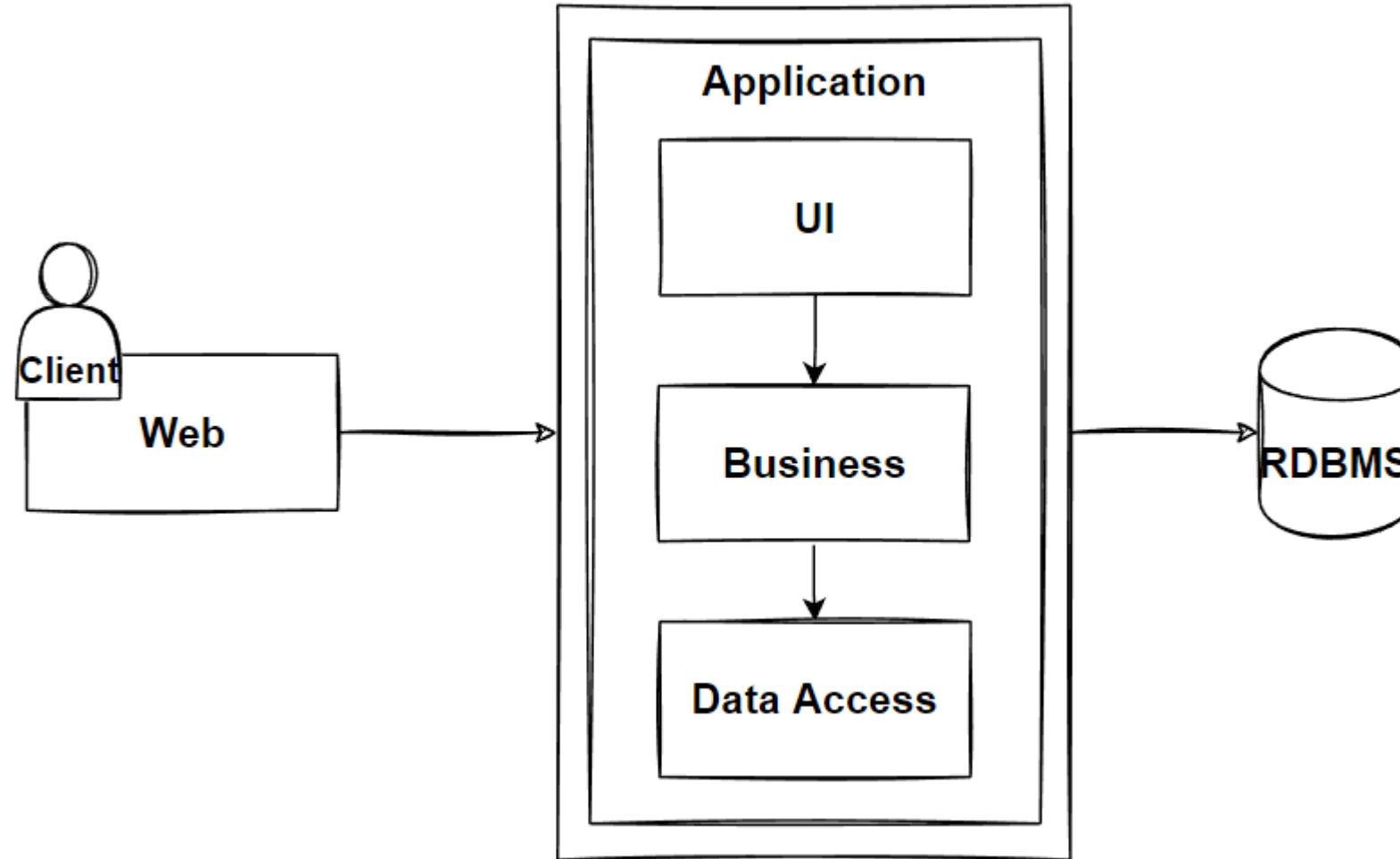
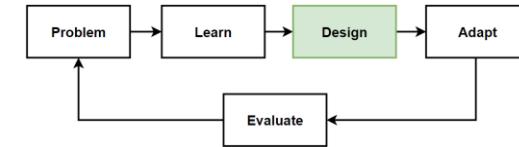
Before Design – What we have in our design toolbox ?

Architectures	Patterns&Principles	Non-FR	FR
• Monolithic Architecture	• KISS • YAGNI • Seperation of Concerns (SoC) • SOLID	• Availability • Small number of Concurrent User • Maintainability	• List products • Filter products as per brand and categories • Put products into the shopping cart • Apply coupon for discounts • Checkout the shopping cart and create an order • List my old orders and order items history
• Layered Architecture			

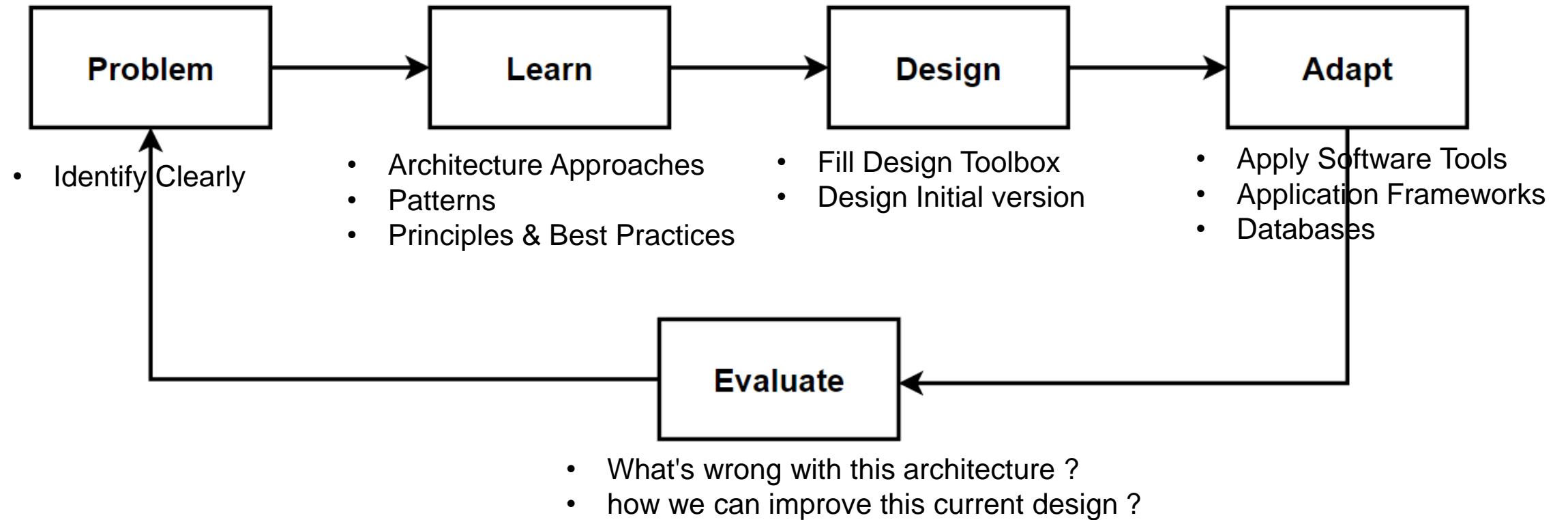
Design: Layered Monolithic Architecture Steps



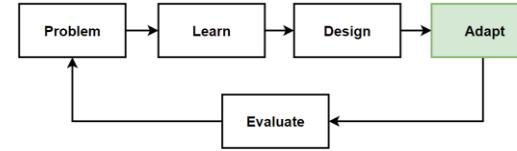
Design: Layered Monolithic Architecture



Way of Learning – The Course Flow



Adapt: Layered Monolithic Architecture

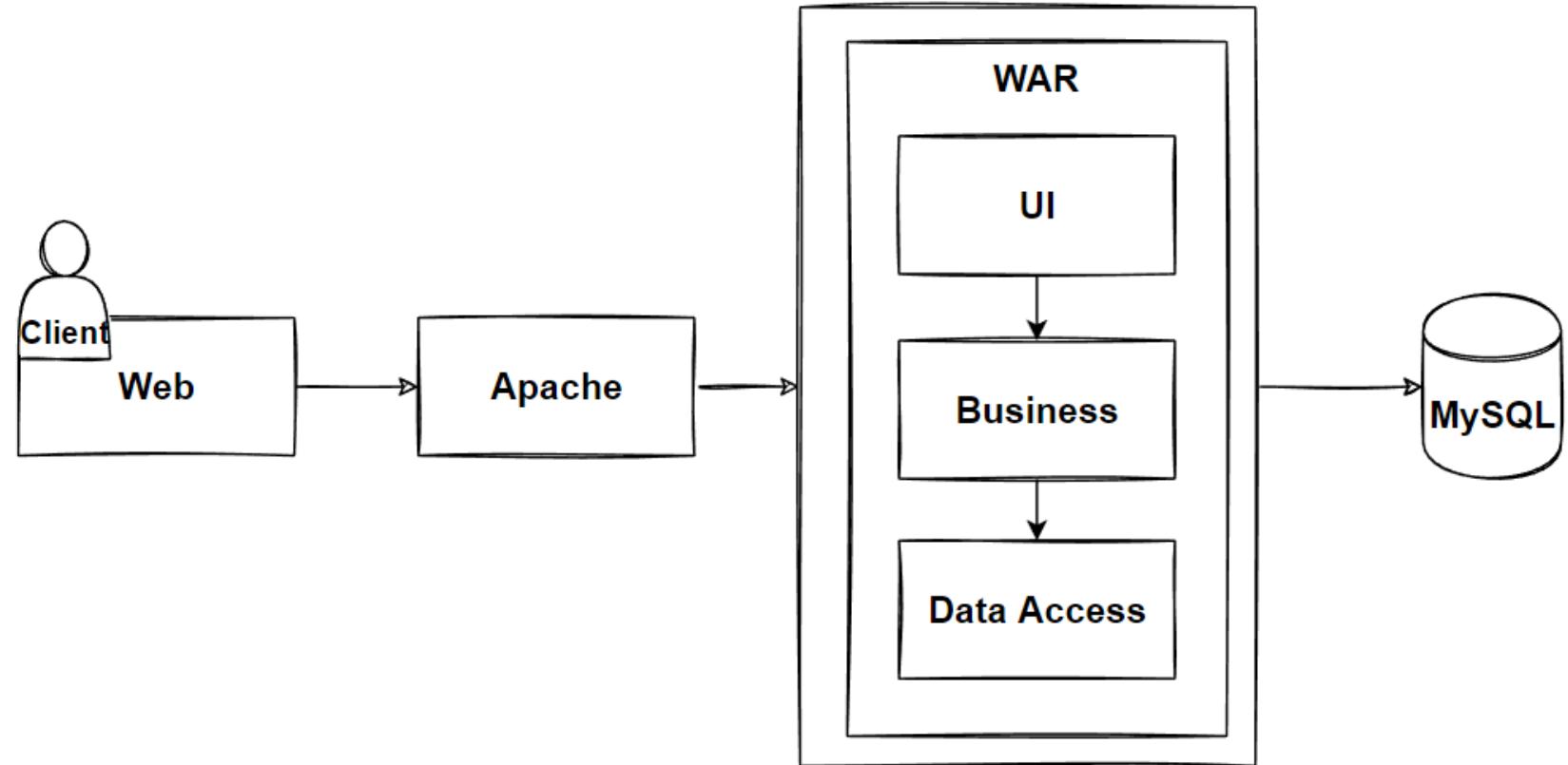


Java World

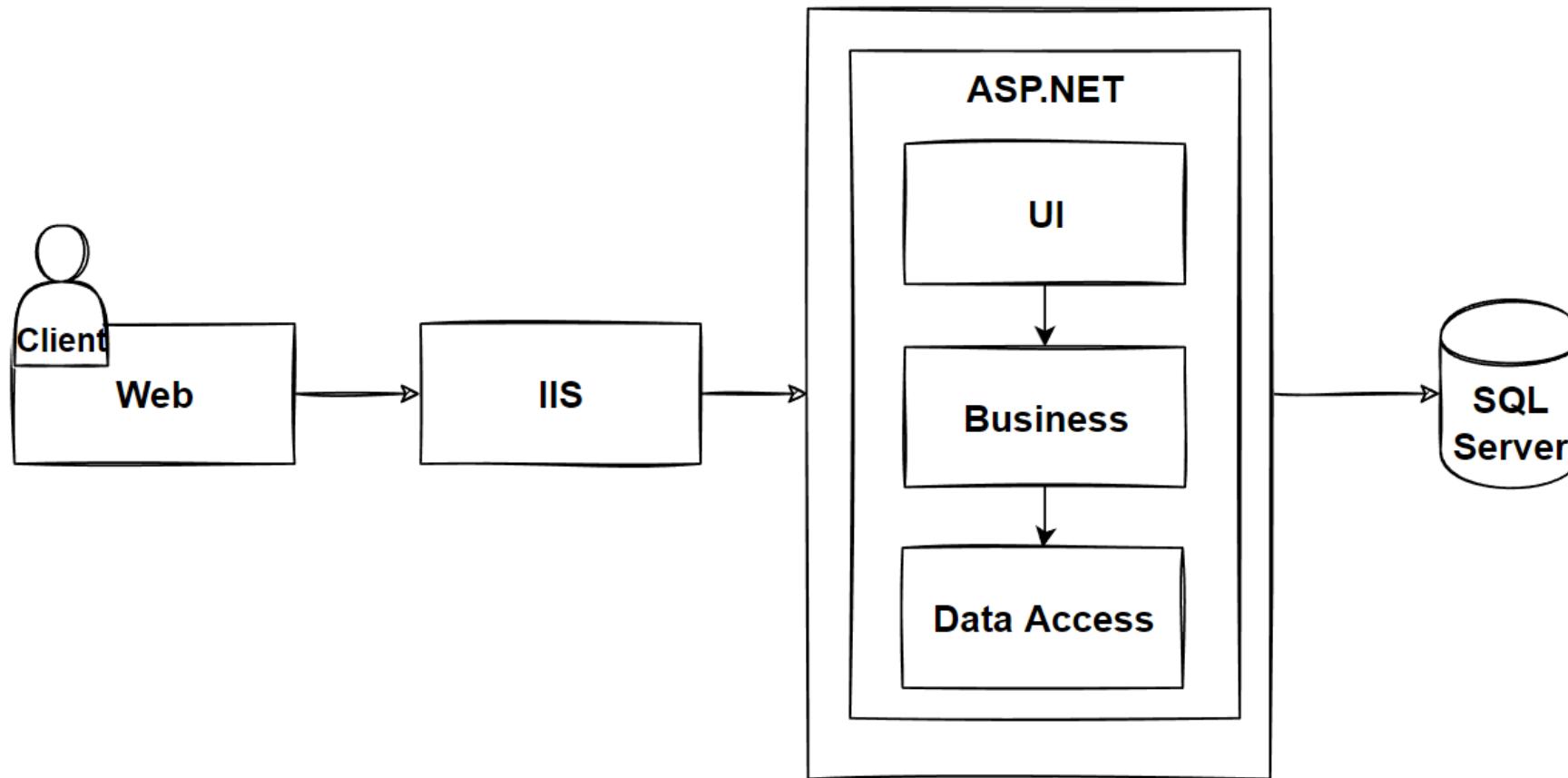
- Apache Web Server
- J2EE
- MySQL

.NET World

- IIS Web Server
- Asp.Net
- SQL Server



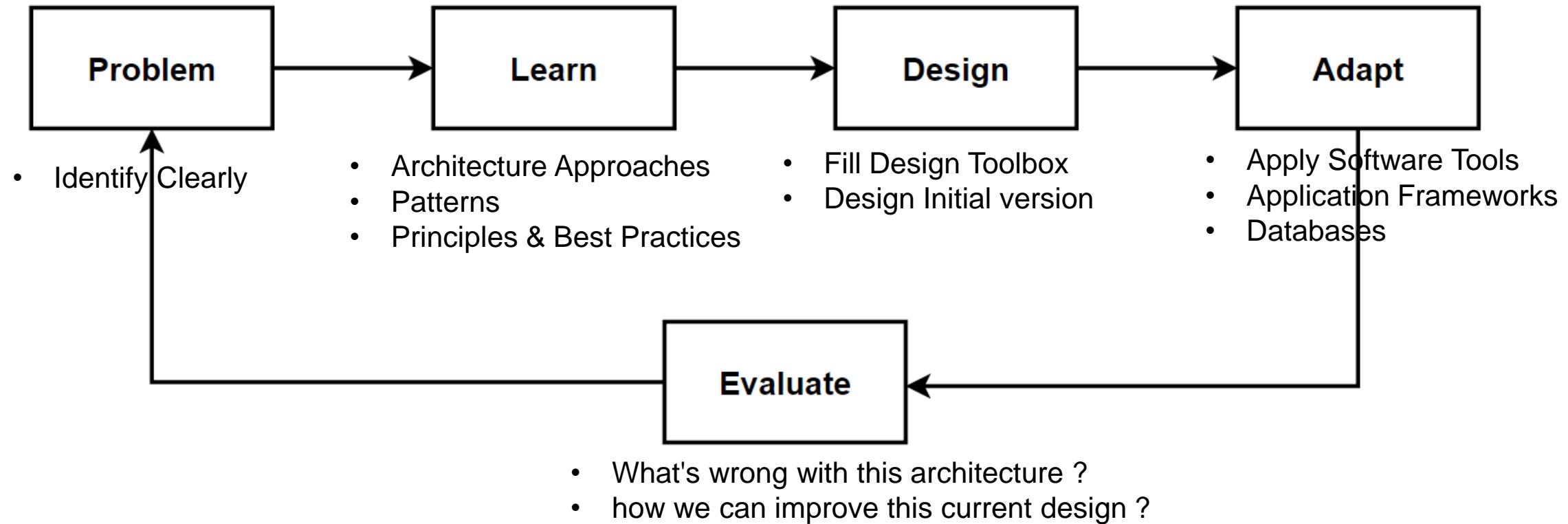
DEMO: Layered Monolithic Architecture Code Review



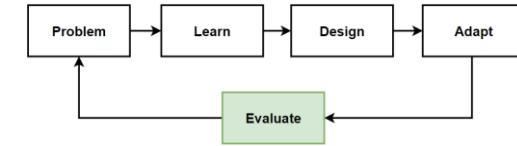
DEMO: Code review of .NET Implementation

- Asp.Net e-commerce web application
- <https://github.com/aspnetrun/run-aspnetcore-basics>
- <https://github1s.com/aspnetrun/run-aspnetcore-basics>

Way of Learning – The Course Flow



Evaluate: Layered Monolithic Architecture



Benefits

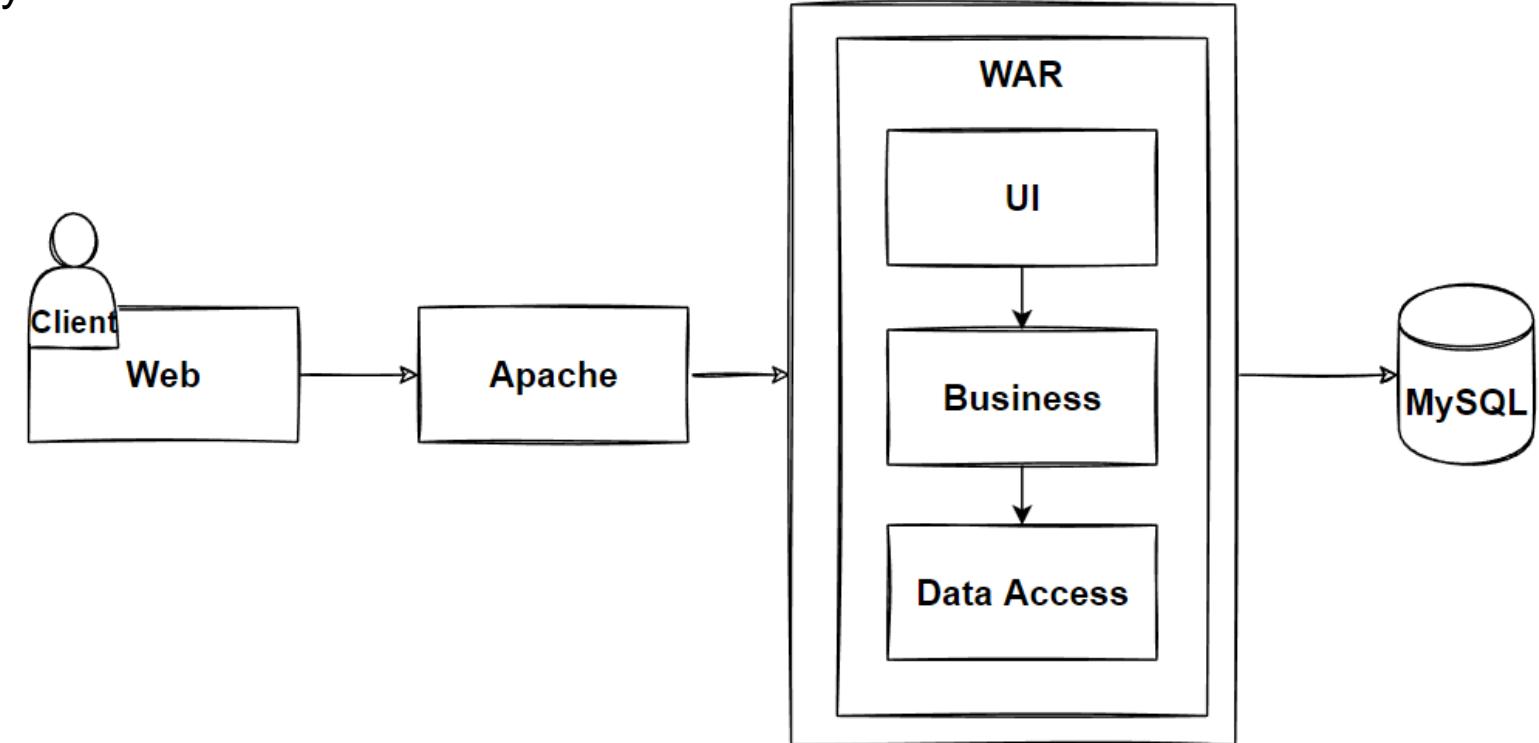
- Easy Development, Debug and Deploy
- Horizontal Logical Layers
- Separation of Concerns

Drawbacks

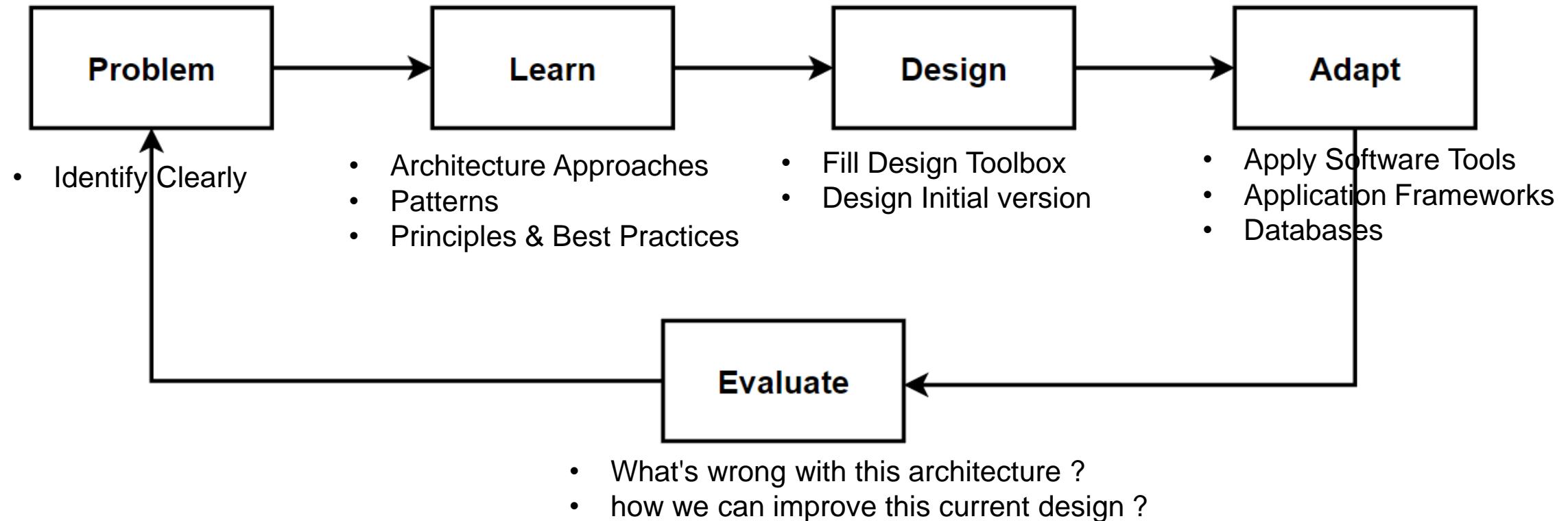
- Layers Dependent each other
- Highly Coupling
- Hard to maintenance
- Complexity of codebase
- Hard to Change libraries; i.e.

Change orm tool with different library

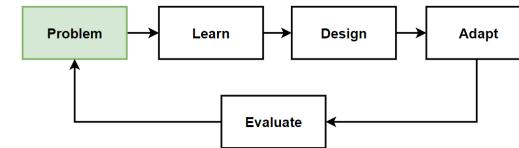
Requires to modify business layer.



Way of Learning – The Course Flow



Problem: Highly Coupling Dependent Layers

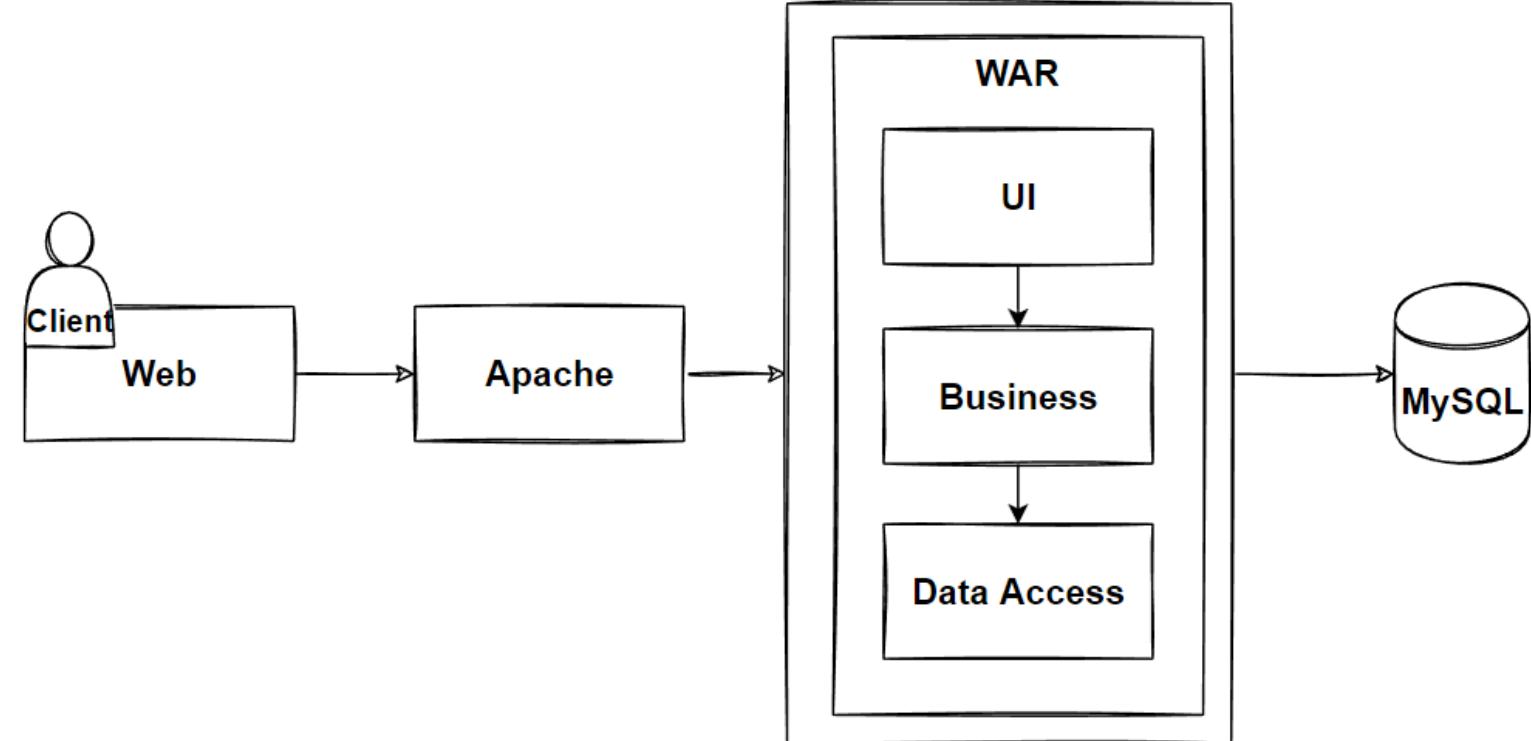


Problems

- Layers are highly coupled and dependent each other
- Code Organization hard to maintain
- Locking of frameworks hard to change

Solutions

- Clean Architecture
- The Dependency Rule



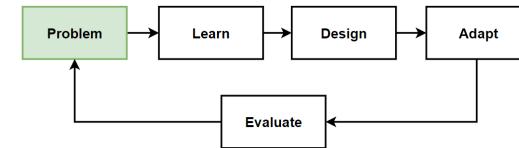
Clean Architecture

Benefits and Challenges of Clean Architecture

The Dependency Rule of Clean Architecture

Design our E-Commerce application with Clean Architecture

Problem: Highly Coupling Dependent Layers



Problems

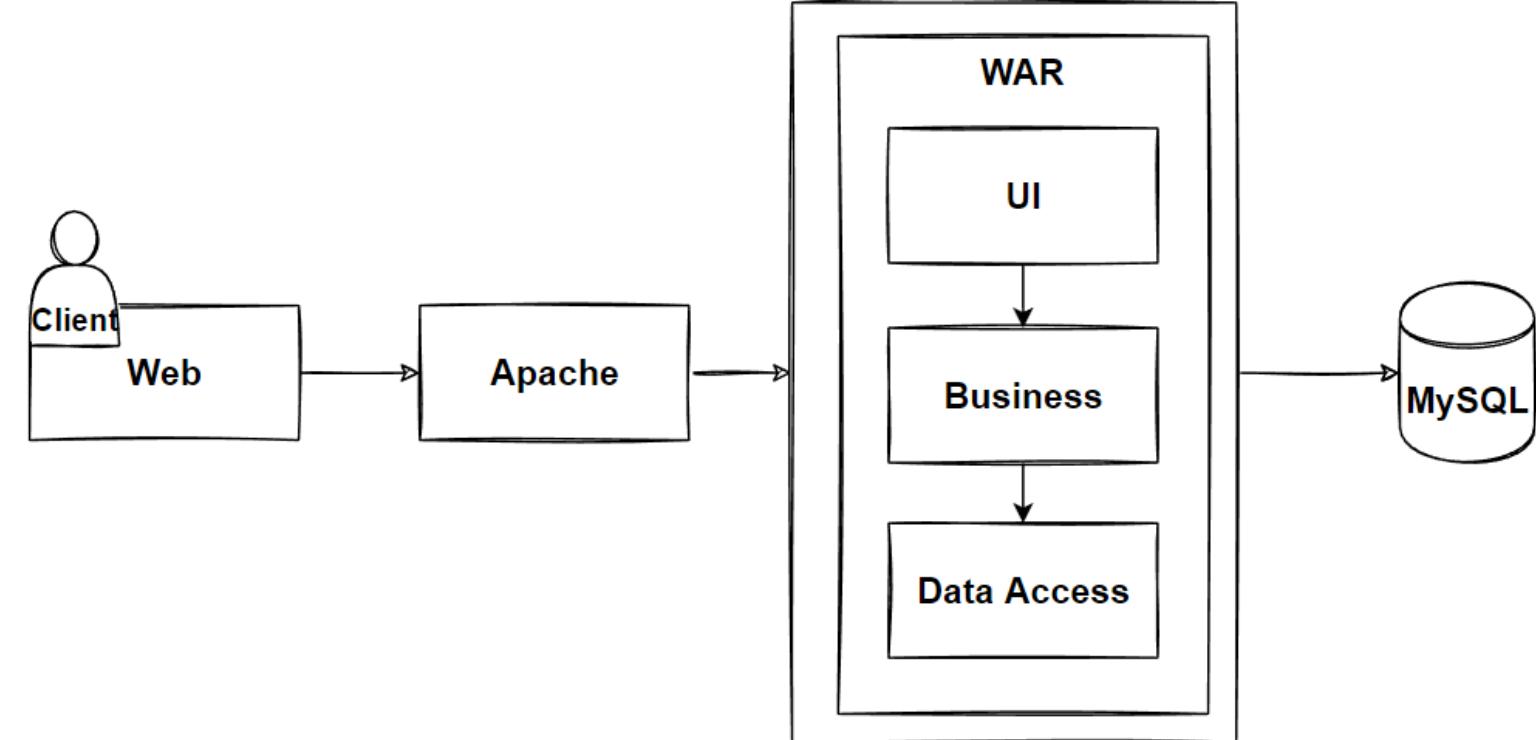
- Layers are highly coupled and dependent each other
- Code Organization hard to maintain
- Locking of frameworks hard to change

Solutions

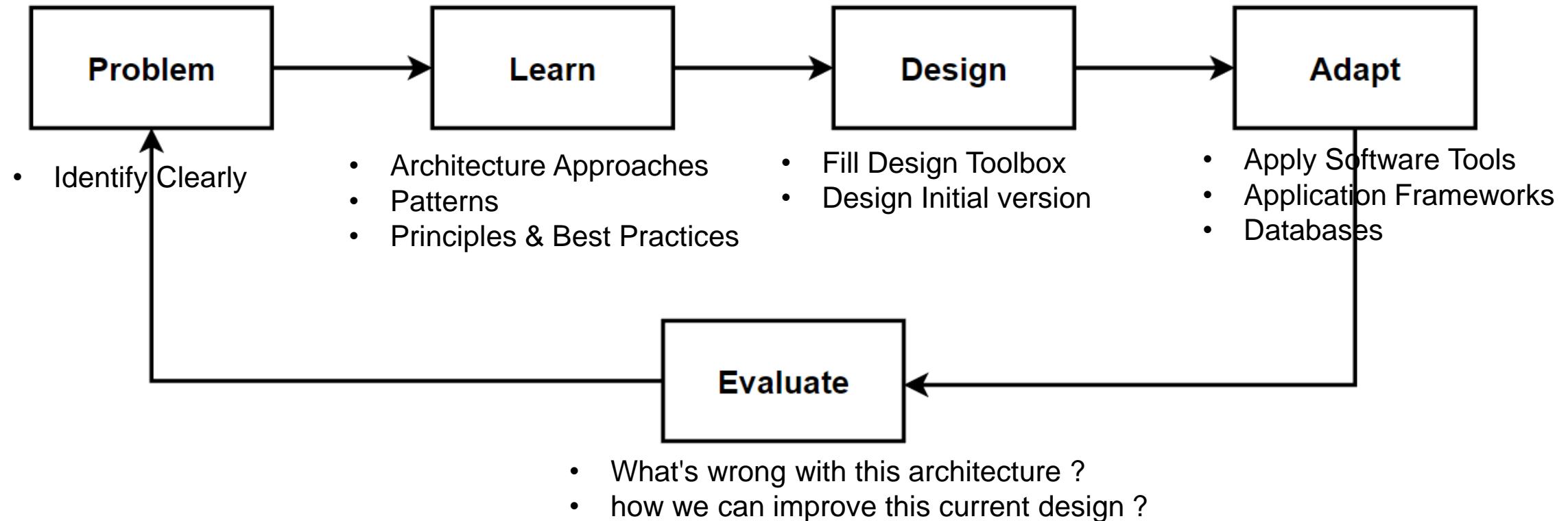
- Clean Architecture
- The Dependency Rule

Add New N-FRs

- Layered Architecture -> Maintainability
- Clean Architecture -> Flexibility, Testability

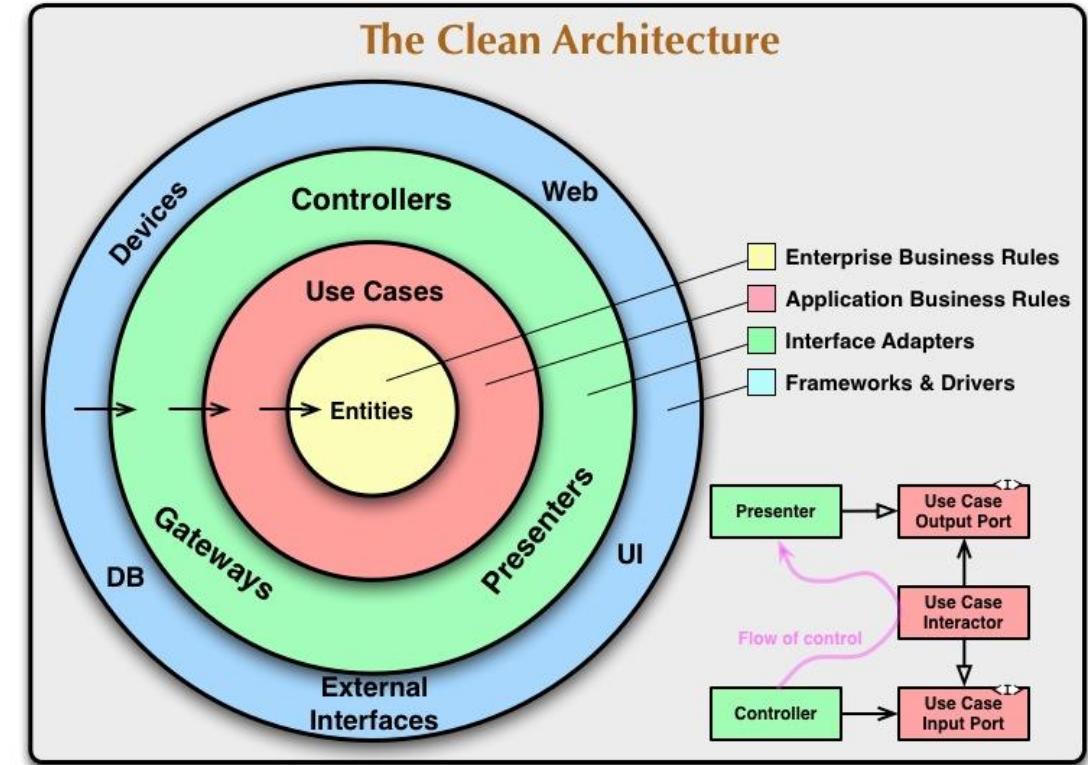
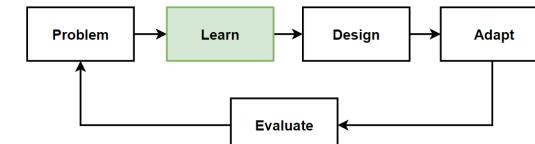


Way of Learning – The Course Flow



Learn: Clean Architecture

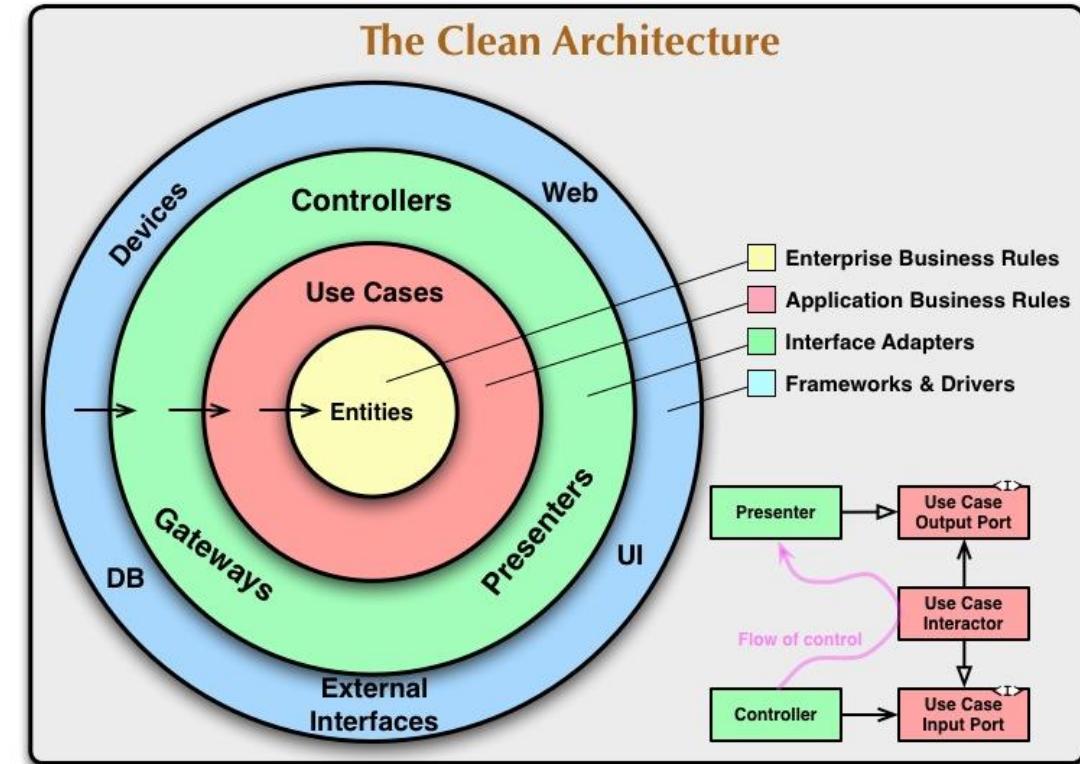
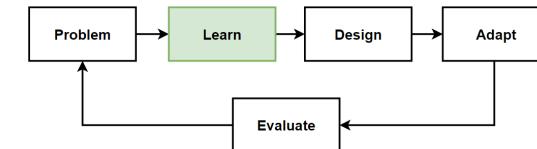
- Separates the elements of a design **into circle levels**.
- Clean architecture was created by **Robert C. Martin** and promoted on his blog, **Uncle Bob**.
- Organize code with encapsulates the business logic.
- Keep the **core business logic** and application domain at the **center of the solution structure** that **independent** with presentation and data access layers.
- Clean architecture divided into **two main elements**: the policies and the details.
 - The **policies** are the business rules and procedures
 - The **details** are the implementation code to carry out the policies
- Focus on the **policies** and **business logics** that build on project requirements
- The internal layers contains the business rules and has **not dependency** of any third-party library.



<https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

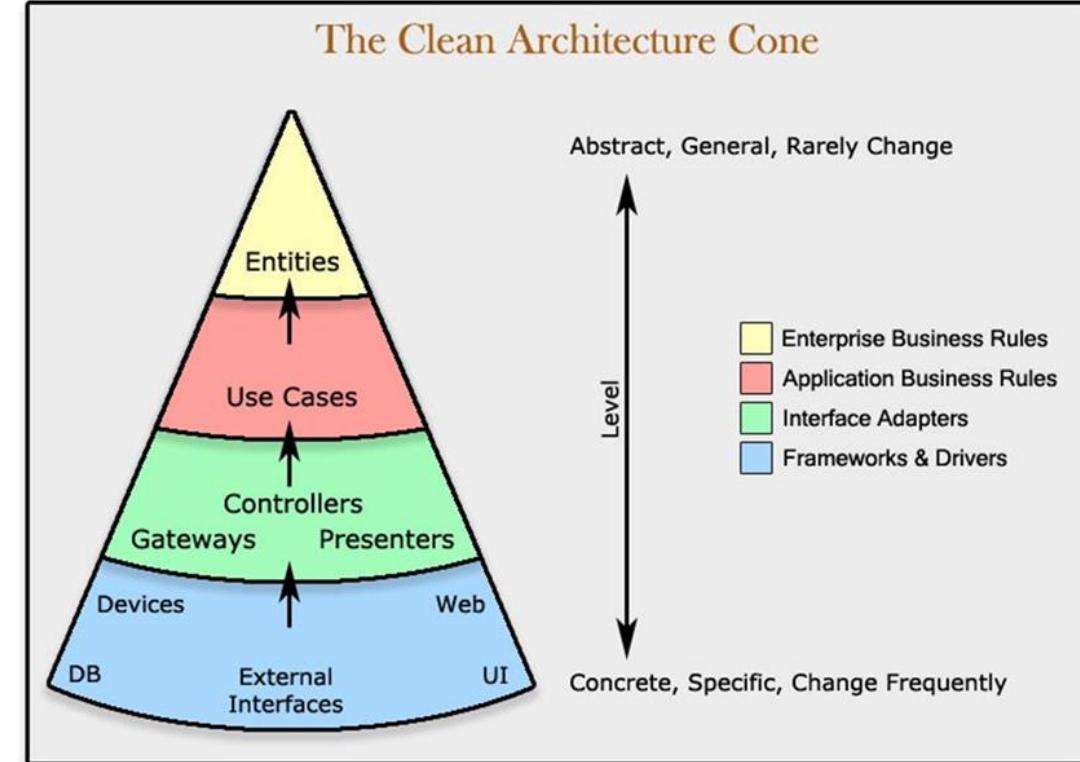
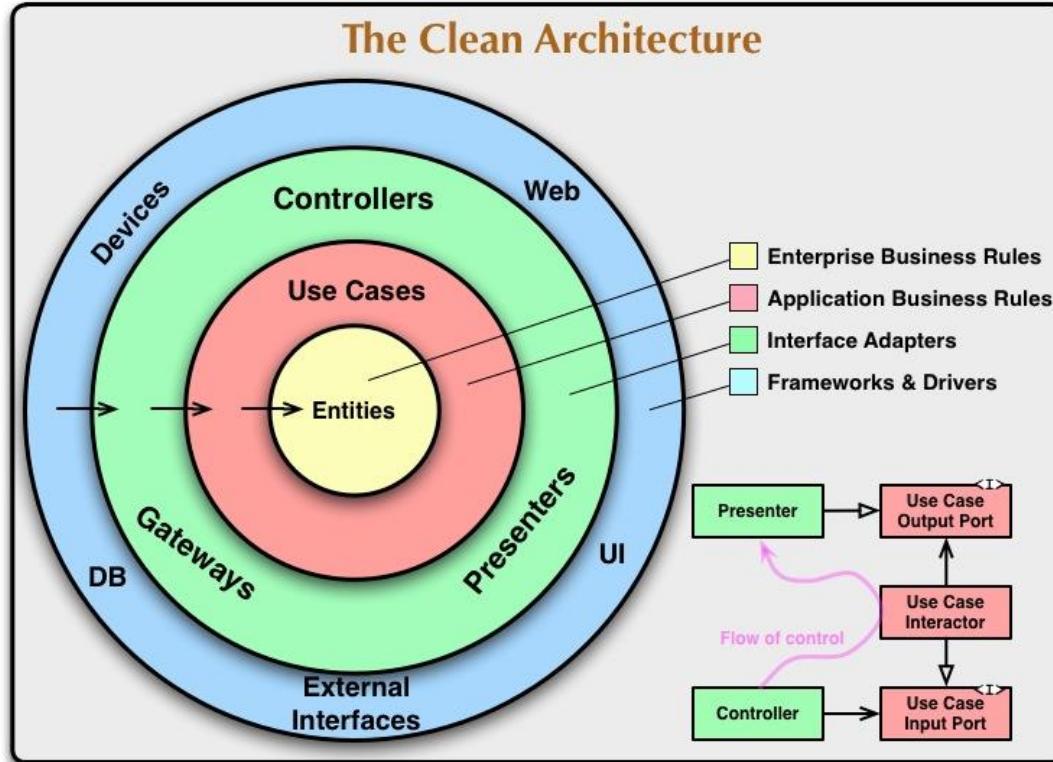
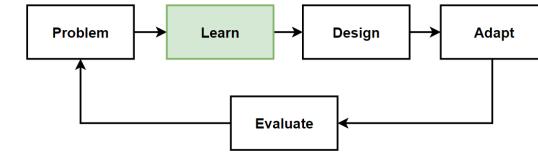
The Dependency Rule

- The dependencies of a source code can only **point inwards**.
- Code dependencies can only **move from the outer levels inward**.
- Code on the inner layers can **have no knowledge** of functions on the outer layers.
- Inner layer cannot **have any information** about elements of an outer layer.
- Classes, functions, variables, data format, or any entity declared in an **outer layer** must not be mentioned by the code of an inner layer.



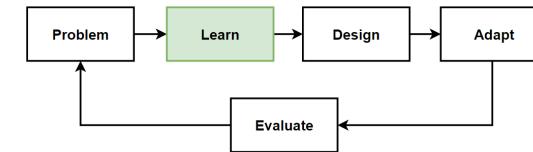
<https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

Layers of Clean Architecture



<https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

Benefits of Clean Architecture



- **Independent of Database and Frameworks**

The software is not dependent on an ORM or Database. You can change them easily.

- **Independence of UI**

The UI can change easily, without changing the rest of the system and business rules.

- **Testable**

It is naturally testable. You can test business rules without considering UI, Database, Mock servers, etc.

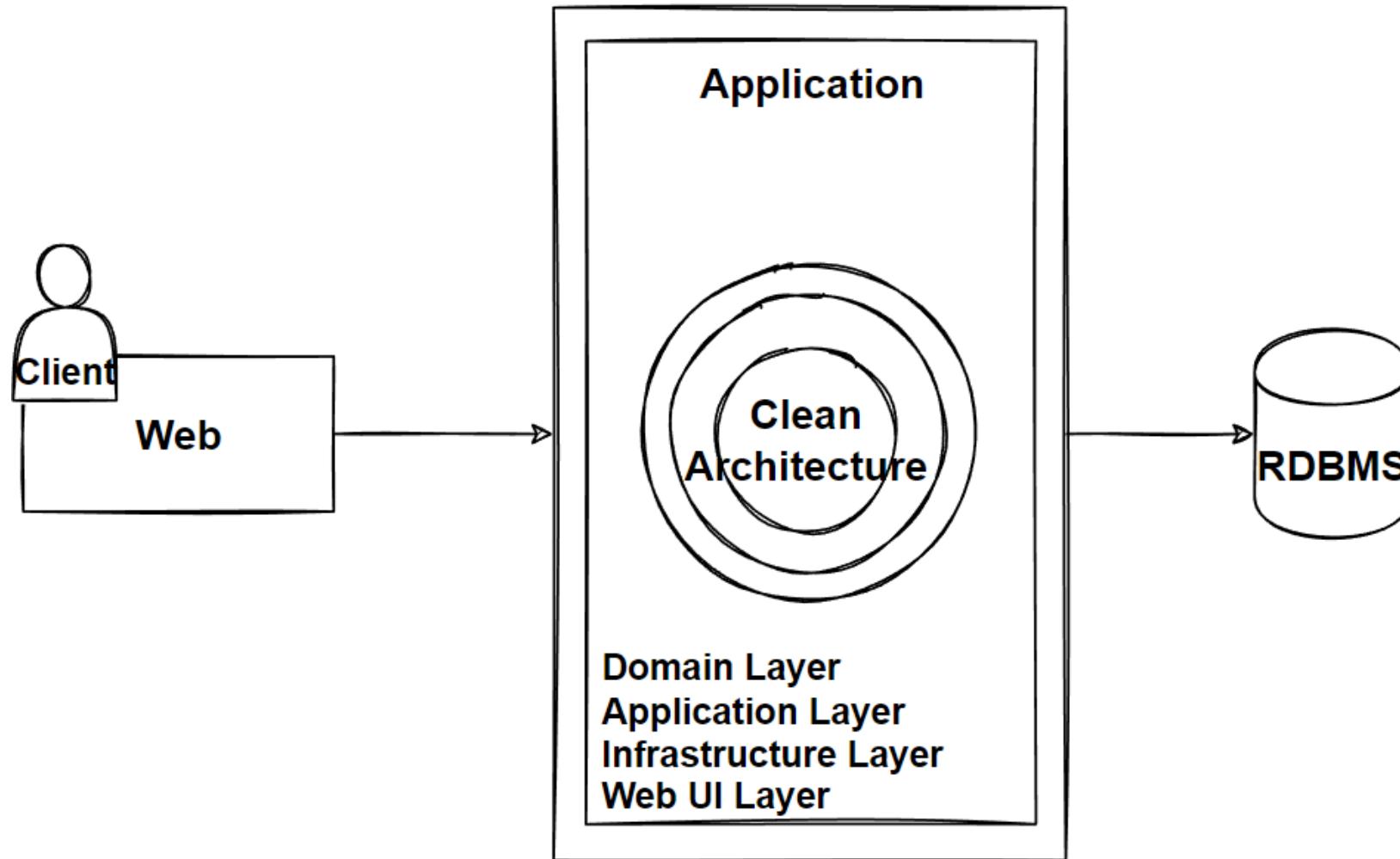
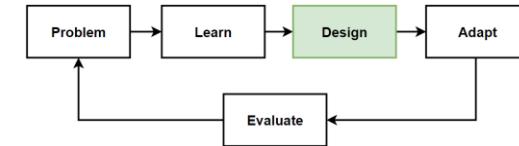
- **Independence of any external agency**

In fact, your business rules simply don't know anything at all about the outside world.

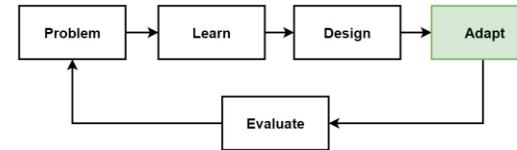
Before Design – What we have in our design toolbox ?

Architectures	Patterns&Principles	Non-FR	FR
• Monolithic Architecture	• KISS • YAGNI	• Availability • Small number of Concurrent User	• List products • Filter products as per brand and categories
• Layered Architecture	• Separation of Concerns (SoC)	• Maintainability	• Put products into the shopping cart
• Clean Architecture	• SOLID • The Dependency Rule	• Flexibility • Testable	• Apply coupon for discounts • Checkout the shopping cart and create an order • List my old orders and order items history

Design: Clean Architecture



Adapt: Clean Architecture

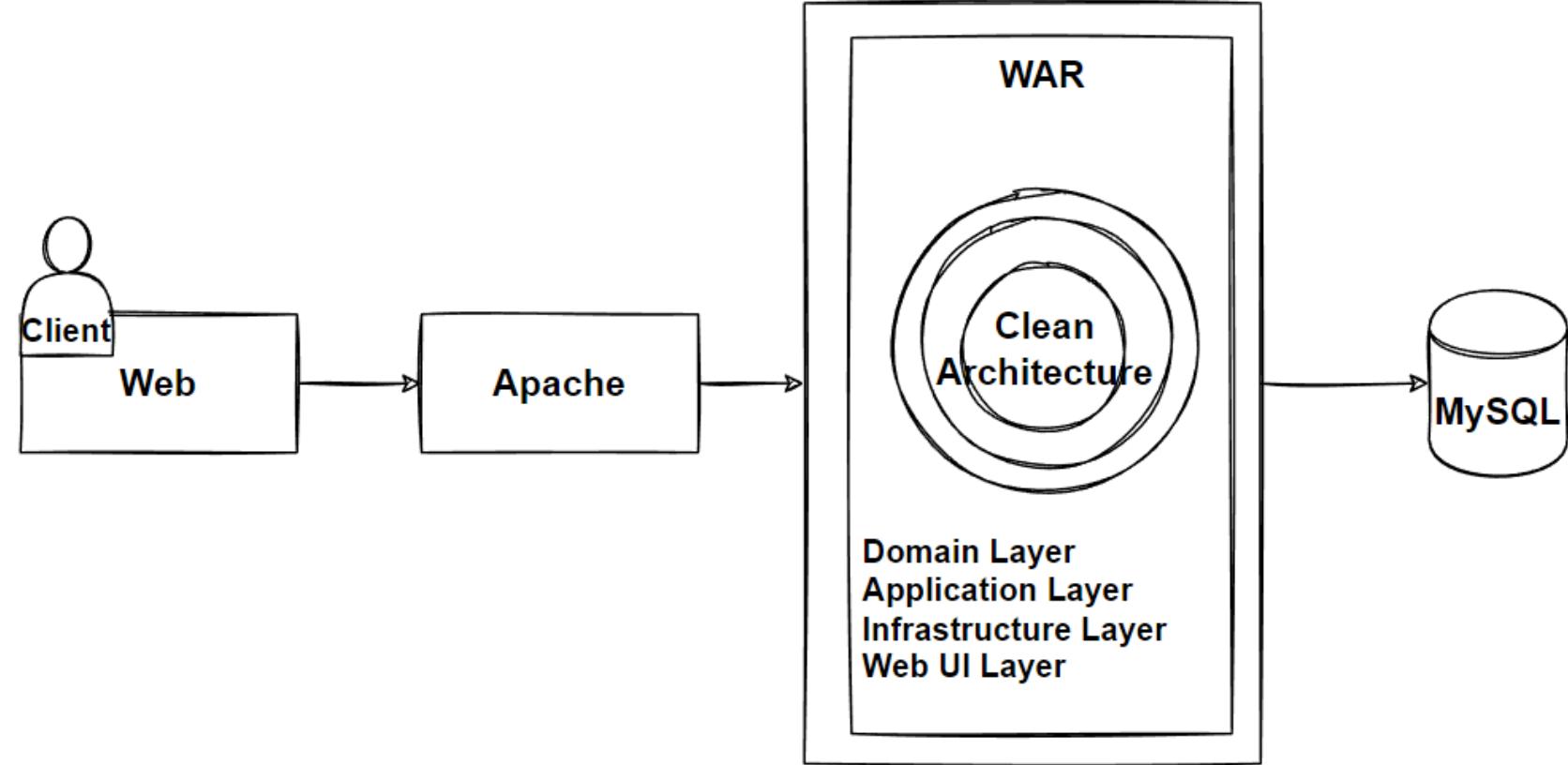


Java World

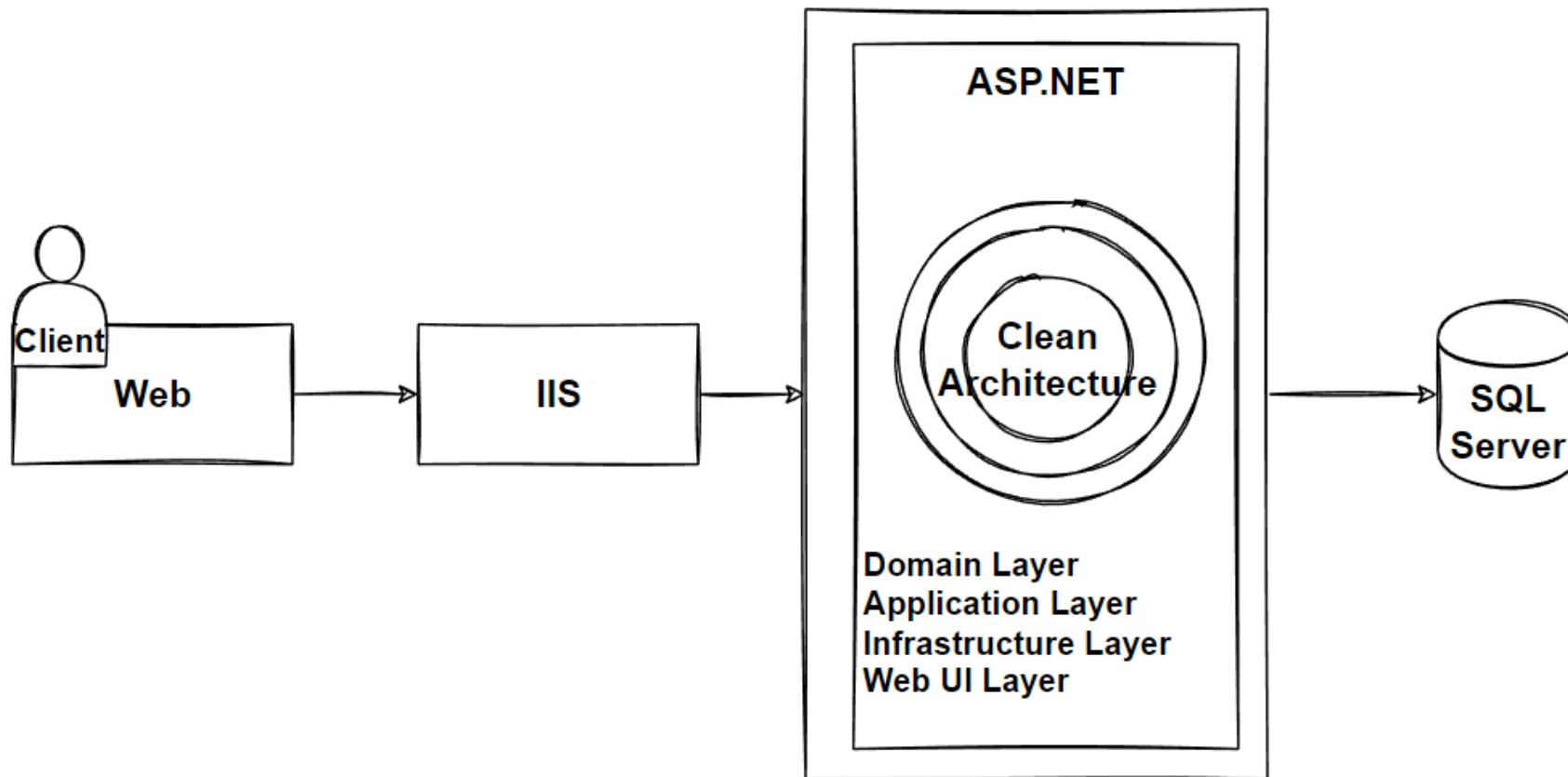
- Apache Web Server
- J2EE
- MySQL

.NET World

- IIS Web Server
- Asp.Net
- SQL Server



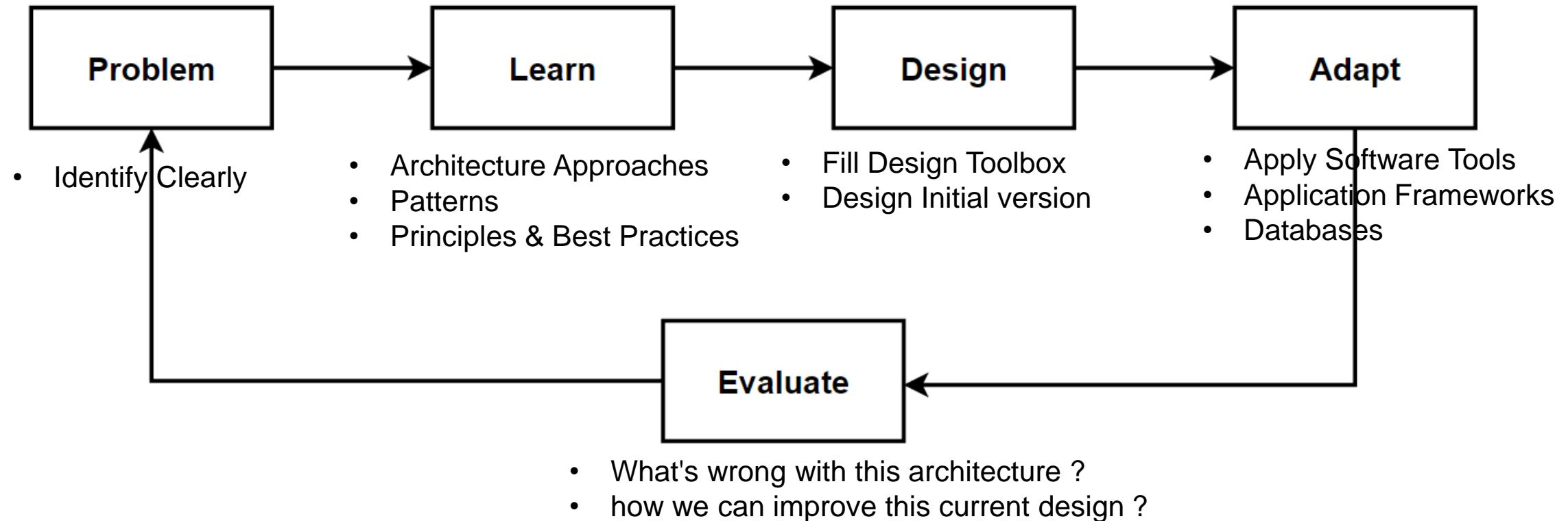
DEMO: Clean Architecture Code Review



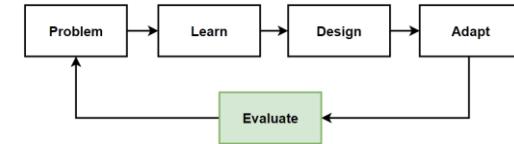
DEMO: Code review of Clean Architecture .NET Implementation

- Real-world - Clean Architecture
- <https://github.com/aspnetrun/run-aspnetcore-realworld>
- <https://github1s.com/aspnetrun/run-aspnetcore-realworld>

Way of Learning – The Course Flow



Evaluate: Clean Architecture

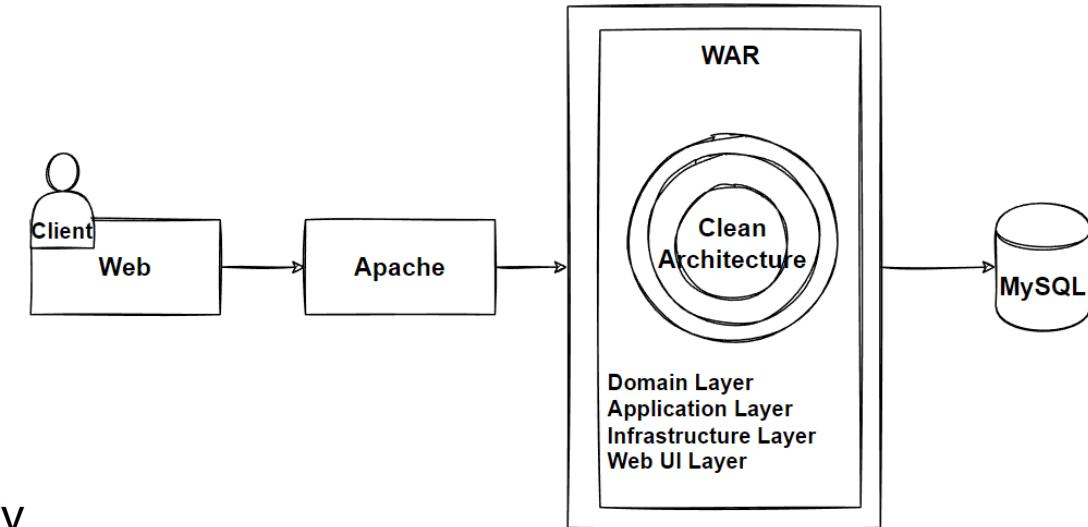


Benefits

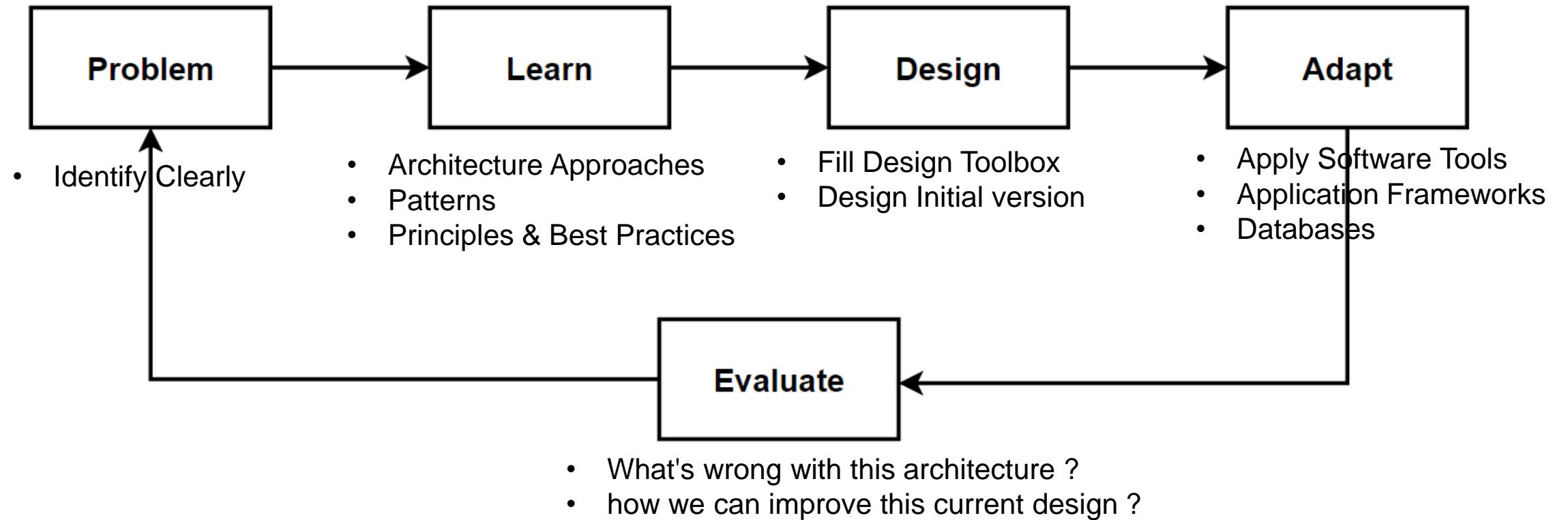
- Easy Development, Debug and Deploy
- Loosely Coupled Independent Layers
- Flexible Logical Layers
- Testable and Independent changeable to 3rd parties libraries

Drawbacks

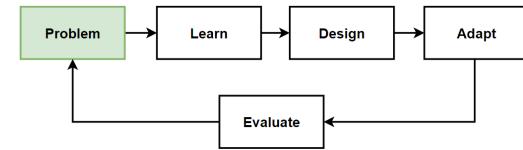
- Layers are independent but those are technical layers:
 - Domain, Infrastructure, Application and UI Layer
- Vertical business logic implementation codes required to modify all layers: i.e. add to basket, checkout order use cases
- It is **still Monolithic** and has **Scalability Issues**
 - How many concurrent request can accommodate our design ?



Way of Learning – The Course Flow



Problem: Increased Traffic, Handle More Request

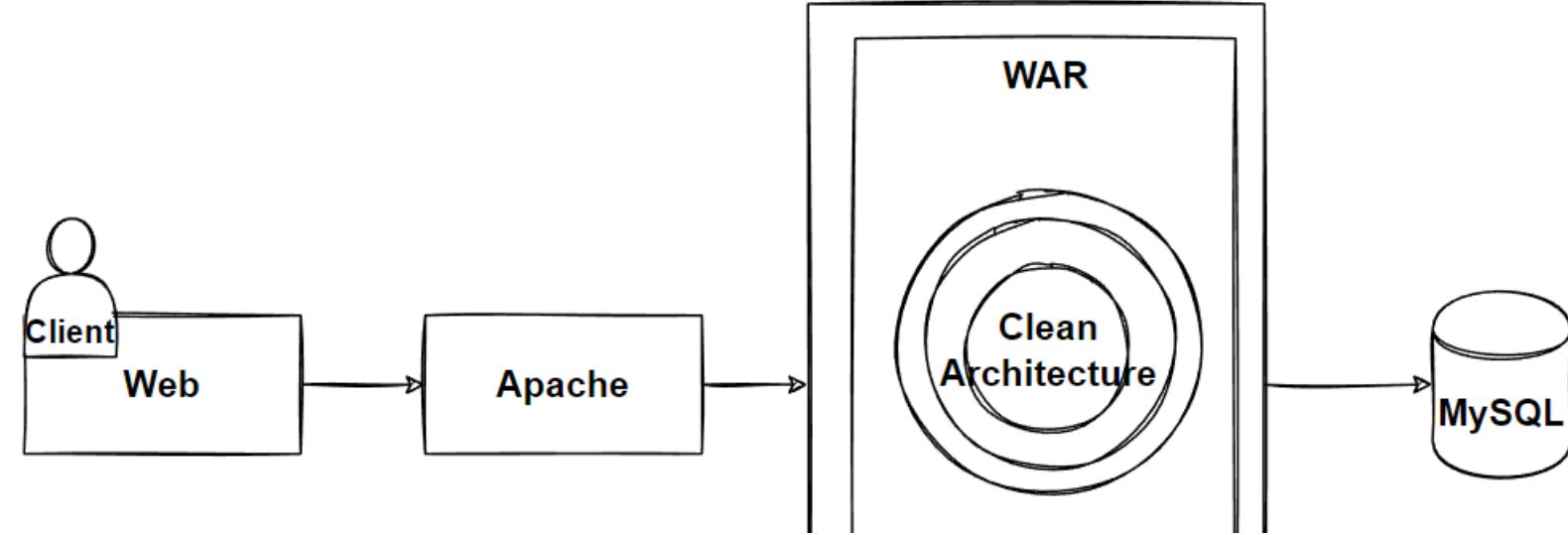


Problems

- Our E-Commerce Business is growing
- Need to handle greater amount of request per second
- Provide acceptable latency for users

Solutions

- Scalability
- Vertical and Horizontal Scaling
- Scale Up and Scale Out
- Load Balancer



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

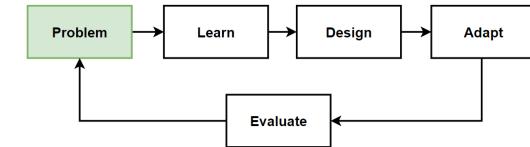
Scalability

Vertical Scaling - Horizontal Scaling of Application

Calculate How many concurrent request can accommodate our design ?

Load Balancer with Consistent Hashing

Problem: Increased Traffic, Handle More Request

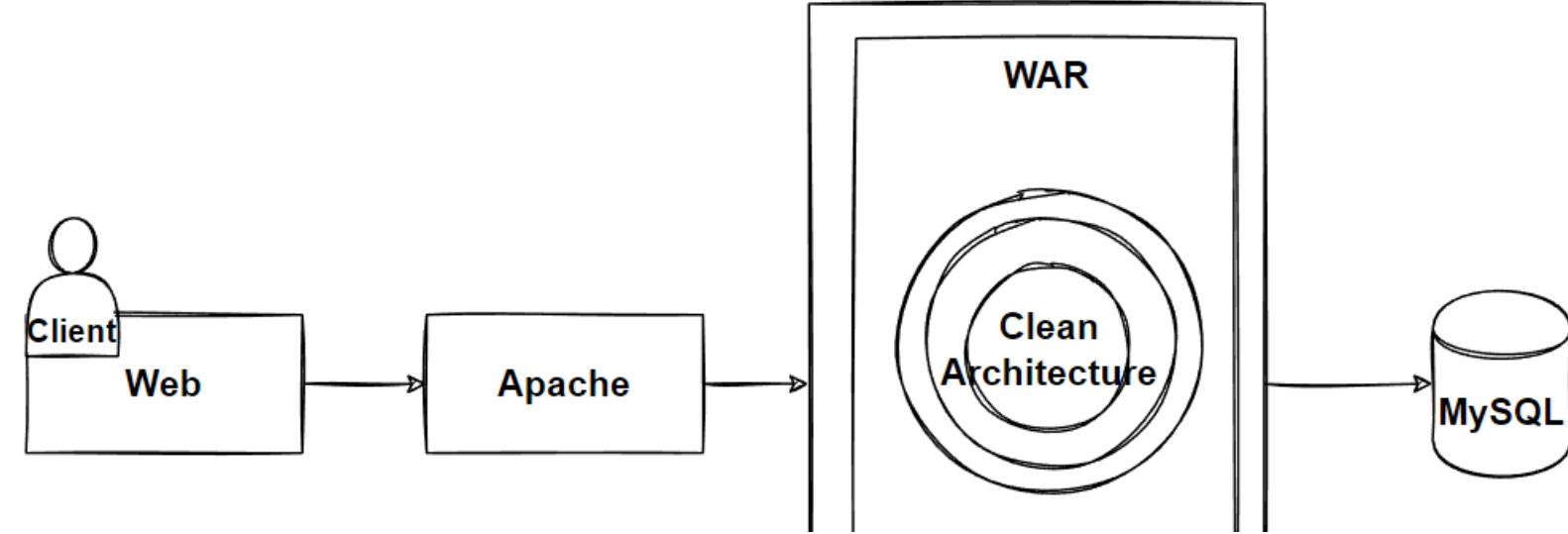


Problems

- Our E-Commerce Business is growing
- Need to handle greater amount of request per second
- Provide acceptable latency for users

Solutions

- Scalability
- Vertical and Horizontal Scaling
- Scale Up and Scale Out
- Load Balancer



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

Understand E-Commerce Domain: Non-Functional Requirements

- ilities

Scalability

Availability

Reliability

Maintability

Usability

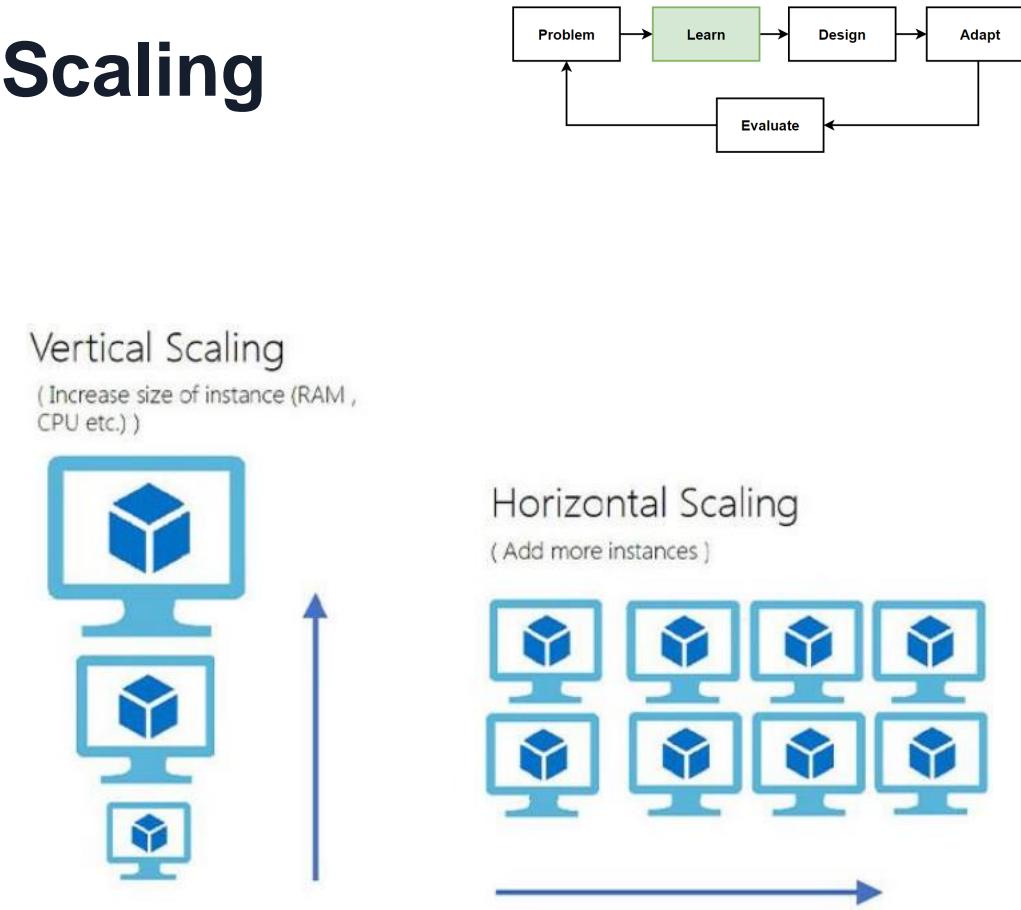
Eficiency

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	?

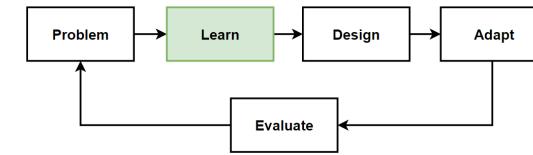
Scalability - Vertical Scaling - Horizontal Scaling

- **Scalability** is the **number of requests** an application can handle
- Measured by the number of requests and it can effectively support **simultaneously**.
- If no longer handle any more simultaneous requests, it has reached its **scalability limit**.
- To prevent **downtime**, and **reduce latency**, you must scale
- **Horizontal scaling** and **vertical scaling** both involve adding resources to your computing infrastructure
- **Horizontal scaling** by adding more machines
- **Vertical scaling** by adding more power



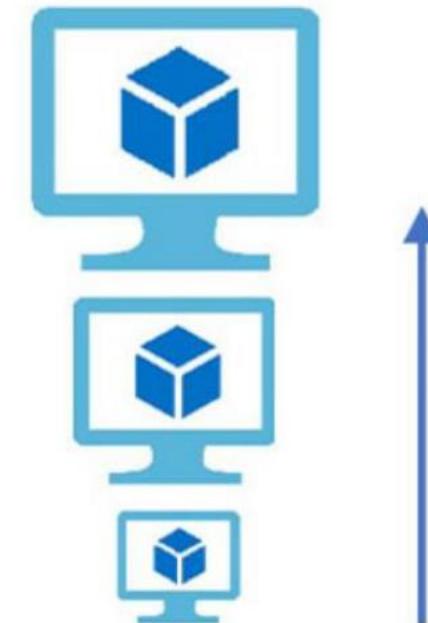
Vertical Scaling - Scale up

- **Vertical scaling** is basically makes the **nodes stronger**.
- Make the server stronger with **adding more hardware**. Adding more resources to a **single node**.
- Make **optimization** the hardware that will allow you to **handle more requests**.
- Vertical scaling keeps your existing infrastructure but **adding more computing power**.
- Your existing **code doesn't need to change**.
- Adding additional **CPU, RAM, and DISK** to cope with an increasing workload.
- By scaling up, you increase the capacity of a single machine. And it has limits. That is named **Scalability Limits**.
- Because even the hardware has **maximum capacity limitations**.
- For handling millions of request, we **need horizontal scaling** or scaling out.



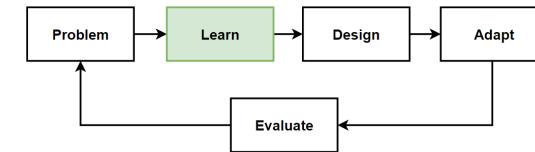
Vertical Scaling

(Increase size of instance (RAM , CPU etc.))



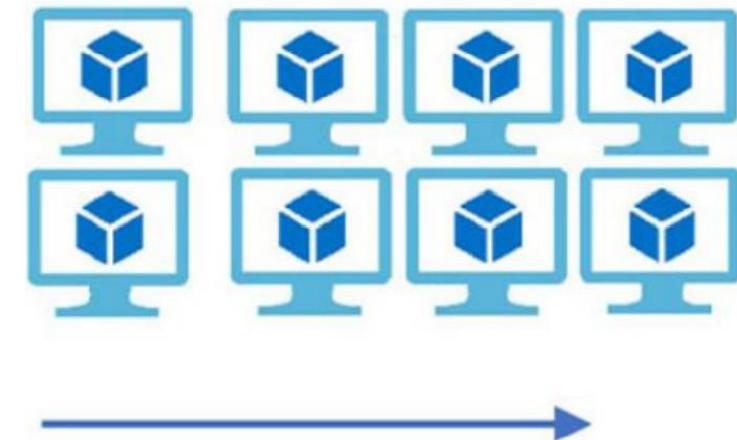
Horizontal Scaling - Scale out

- **Horizontal scaling is splitting the load between different servers.**
- Simply **adds more instances** of machines without changing to existing specifications.
- **Share the processing power and load balancing** across multiple machines.
- Horizontal scaling means adding more machines to the resource pool.
- **Scaling horizontally** gives you scalability but also **reliability**.
- Preferred way to scale in **distributed architectures**.
- When splitting into multiple servers, we need to consider if you have a **state or not**.
- if we have a state like database servers, than we need to manage more considerations like **CAP theorem**.



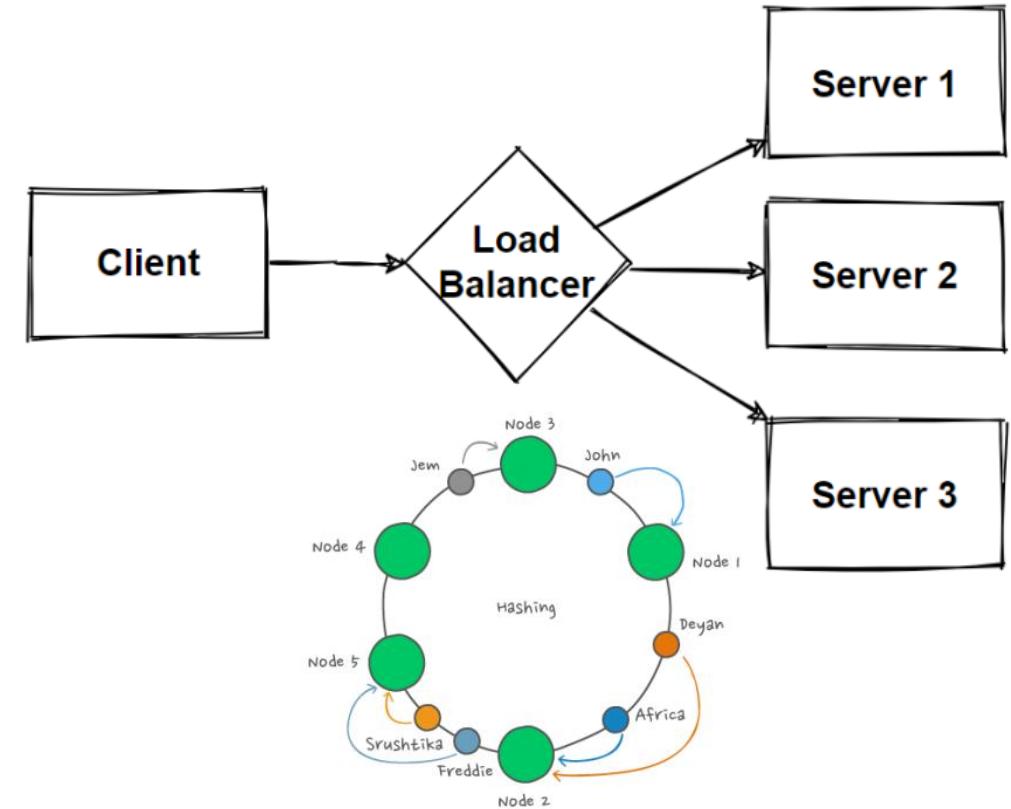
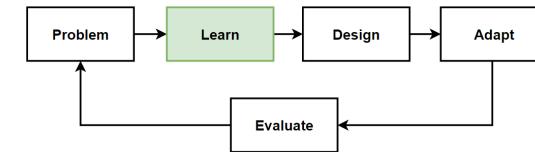
Horizontal Scaling

(Add more instances)



Learn: Load Balancer

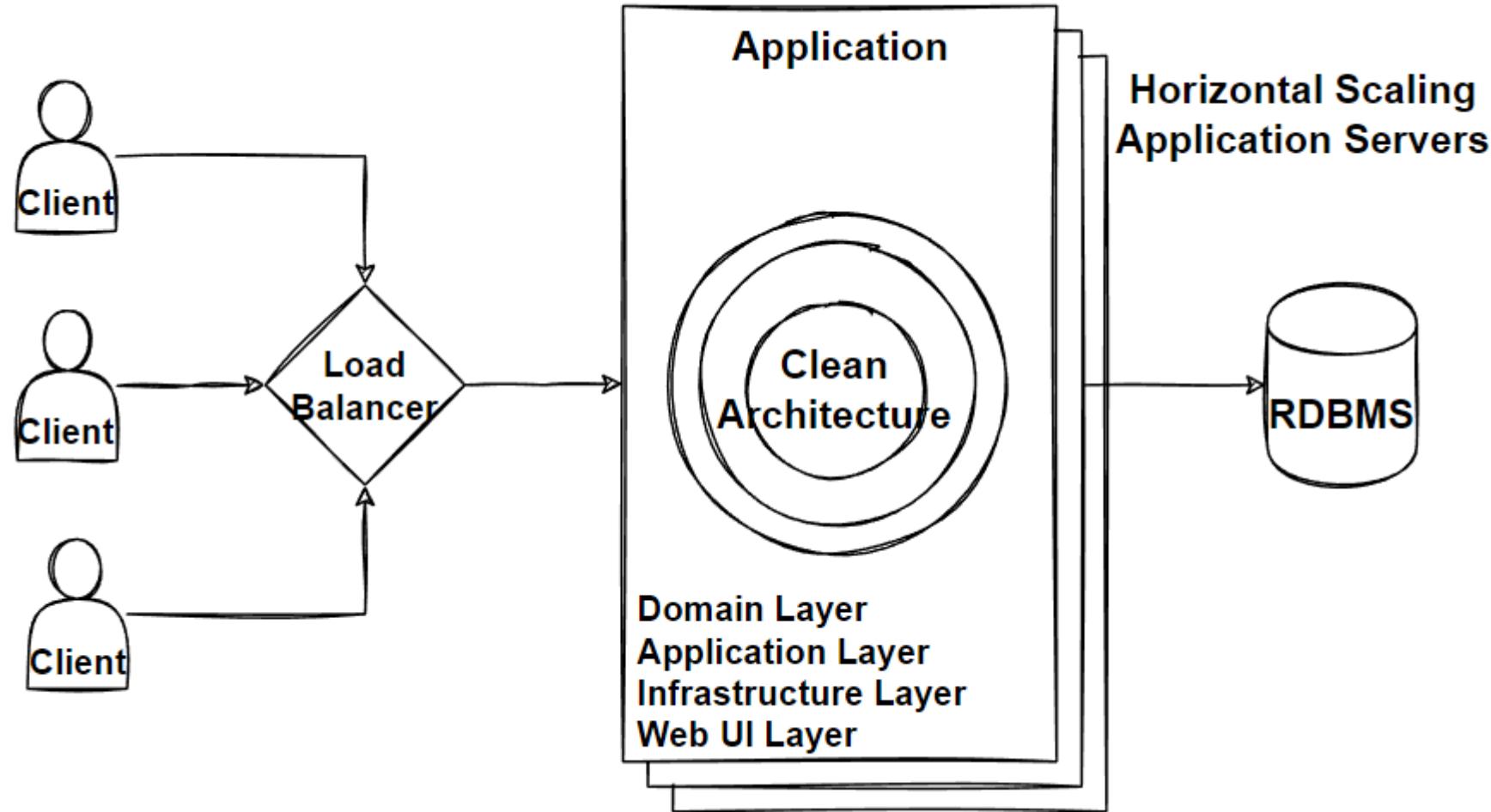
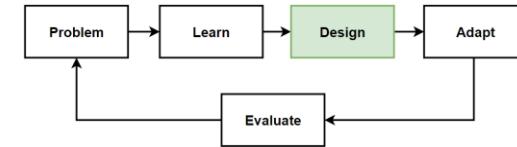
- Balance the traffic across to all nodes of our applications.
- Spread the traffic across a cluster of servers to improve responsiveness and availability.
- Load Balancer sits between the client and the server.
- Load Balancer is accepting incoming network and application traffic.
- Distributing the traffic across multiple backend servers using different algorithms.
- Load Balancers should be fault tolerance and improves availability.
- Mostly uses the consistent hashing algorithms. Consistent hashing is an algorithms for dividing up data between multiple machines.
- It solves the horizontal scalability problems. Don't have to re-arrange all the keys.



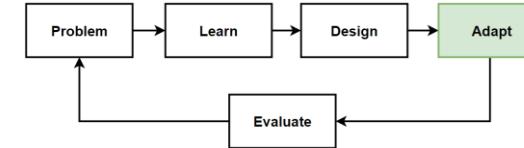
Before Design – What we have in our design toolbox ?

Architectures	Patterns&Principles	Non-FR	FR
• Monolithic Architecture	• KISS • YAGNI	• Availability • High number of Concurrent User	• List products
• Layered Architecture	• Separation of Concerns (SoC)	• Maintainability	• Filter products as per brand and categories
• Clean Architecture	• SOLID • The Dependency Rule • Horizontal Scaling • Load Balancer	• Flexibility • Testable • Scalability • Reliability	• Put products into the shopping cart • Apply coupon for discounts • Checkout the shopping cart and create an order • List my old orders and order items history

Design: Scalability – Horizontal Scaling



Adapt: Scalability – Horizontal Scaling, Load Balancer



Load Balancer

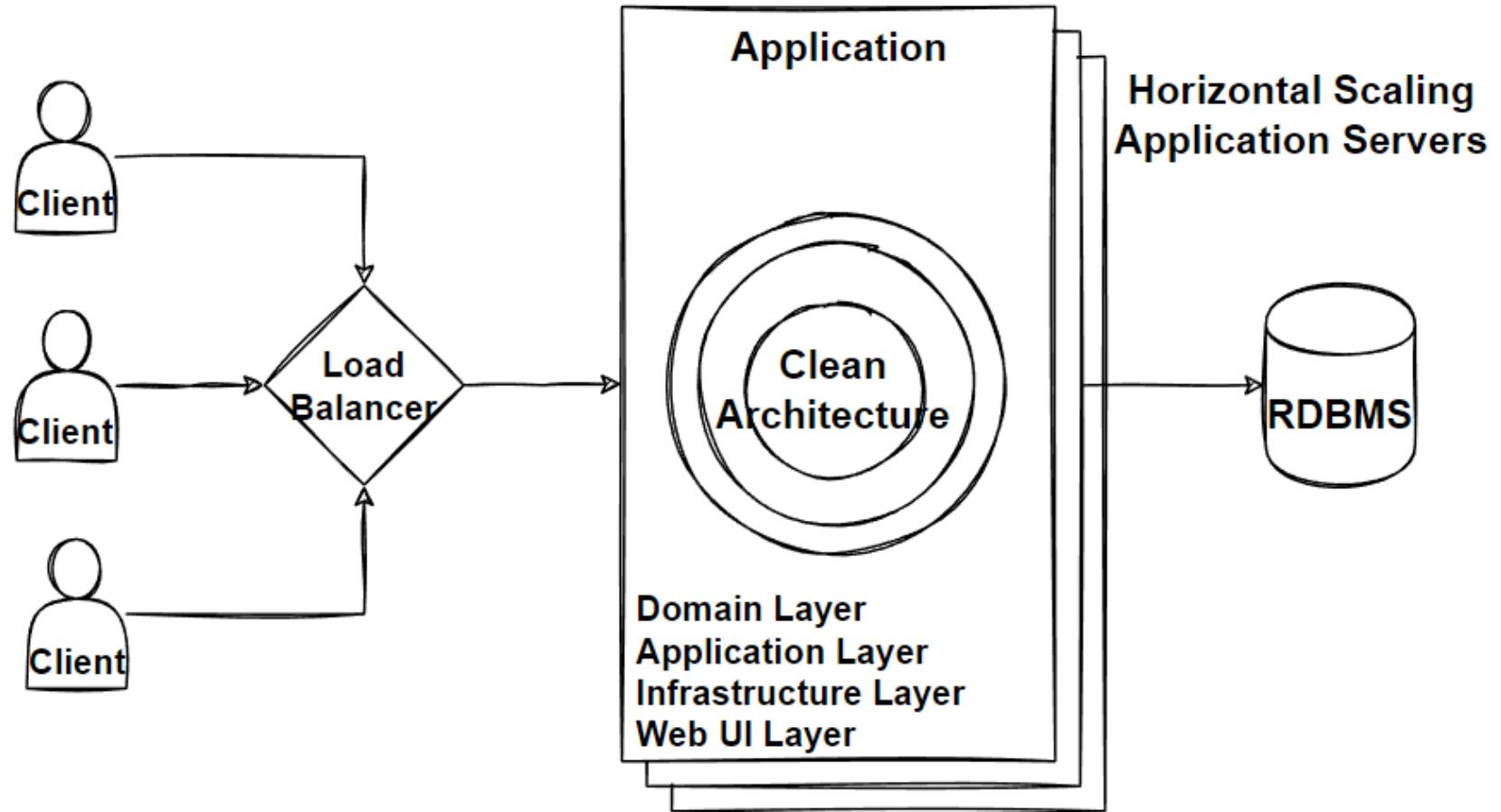
- Apache LB
- NGINX

Application Server

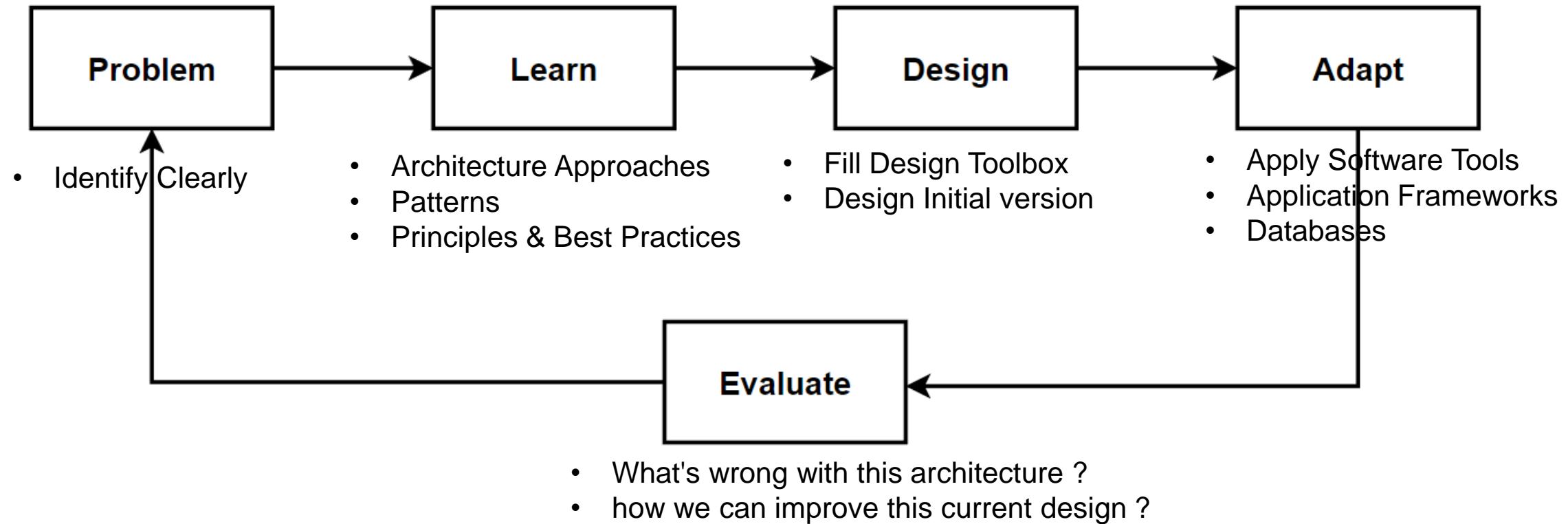
- Single JAR / WAR File
- Tomcat Container

Database

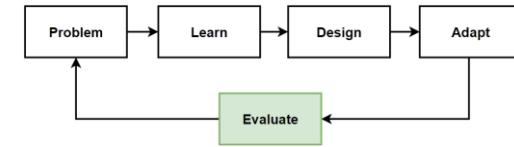
- Oracle
- Postgres
- SQL Server
- MySQL



Way of Learning – The Course Flow



Evaluate: Clean Architecture

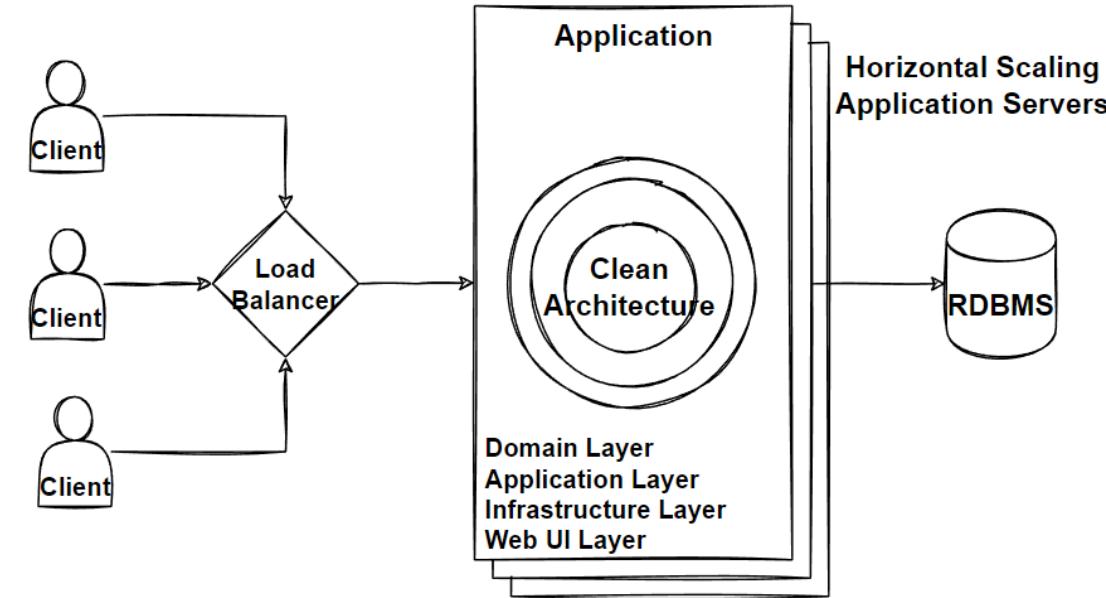


Benefits

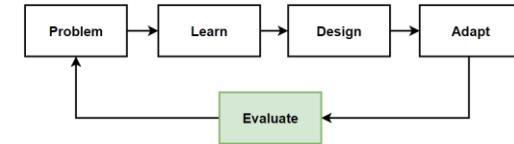
- Easy Development, Debug and Deploy
- Loosely Coupled Independent Layers
- Flexible Logical Layers
- Testable and Independent changeable to 3rd parties libraries

Drawbacks

- Layers are independent but those are technical layers:
 - Domain, Infrastructure, Application and UI Layer
- **Vertical business logic** implementation codes required to modify all layers: i.e. add to basket, checkout order use cases
- It is still Monolithic and has Scalability Issues
 - How many concurrent request can accommodate our design ?



Pain Points of Clean Architecture



- **Context Switching**

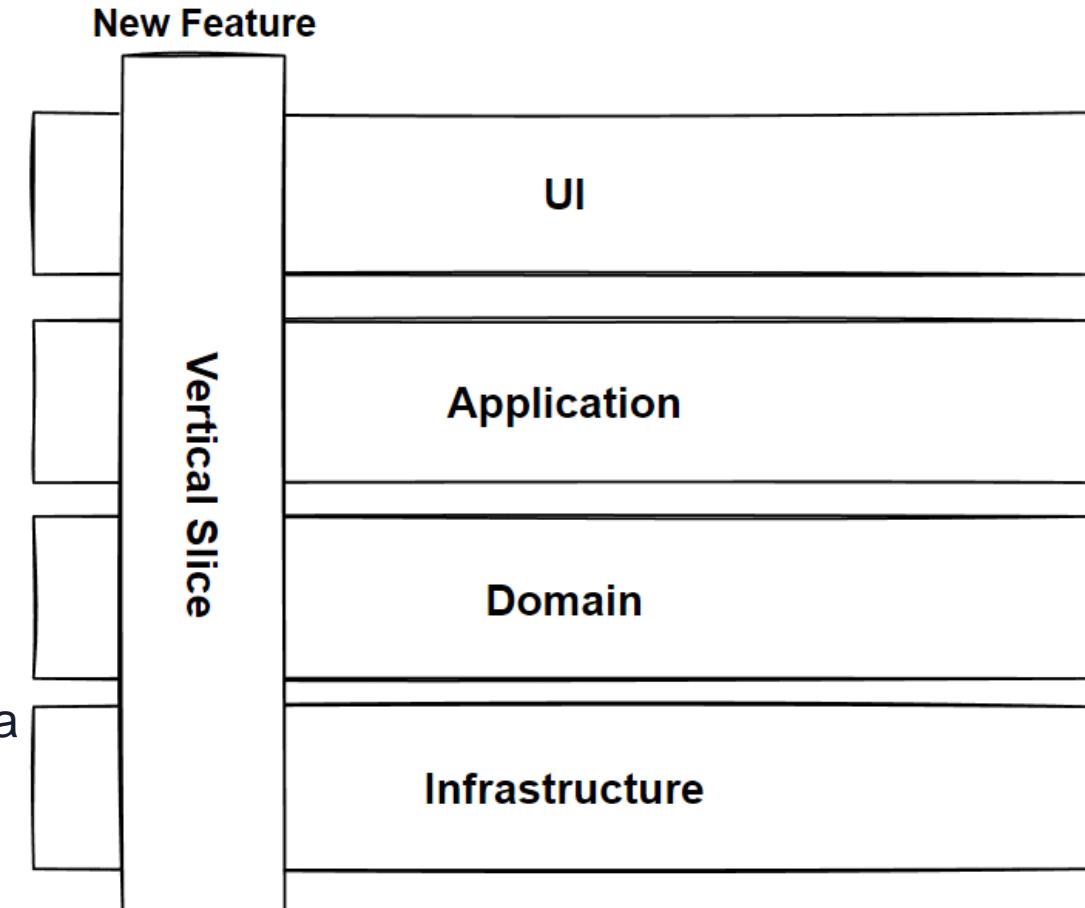
When we were working on a feature and needed to work on any combination of data access, domain or application logic, we had to switch contexts into different software project.

- We had to **remember** and **re-visit codes** from other folder and continue with totally different code base. The folder structure requires DDD-bounded contexts approaches.

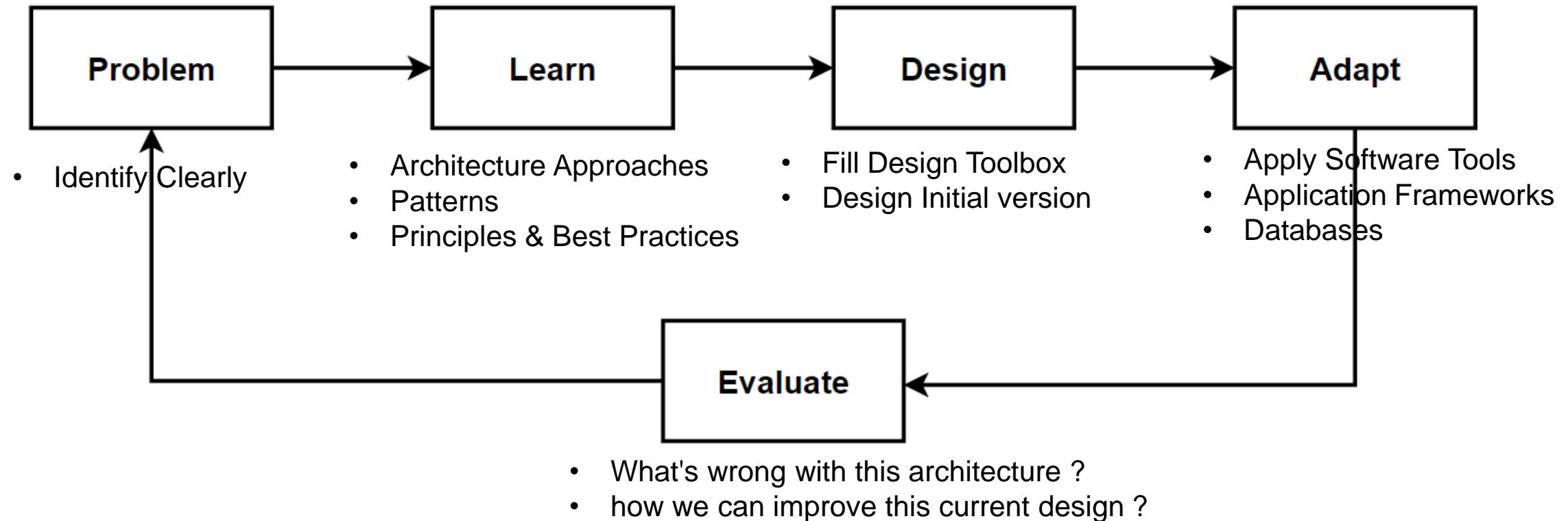
- **Vertical Slices**

When adding a feature in an application, we are developing into almost all layers in the application code.

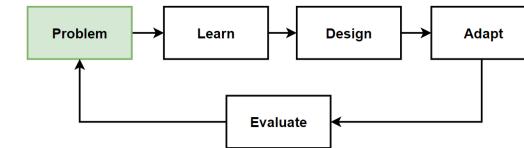
- Changing the user interface, adding new Use Case classes into Application Layer, adding fields to models, modifying Data Access Codes, and so on.
- So that means we are **highly couple** with **vertically slice** when developing features.



Way of Learning – The Course Flow



Problem: Agility of New Features, Split Agile Teams

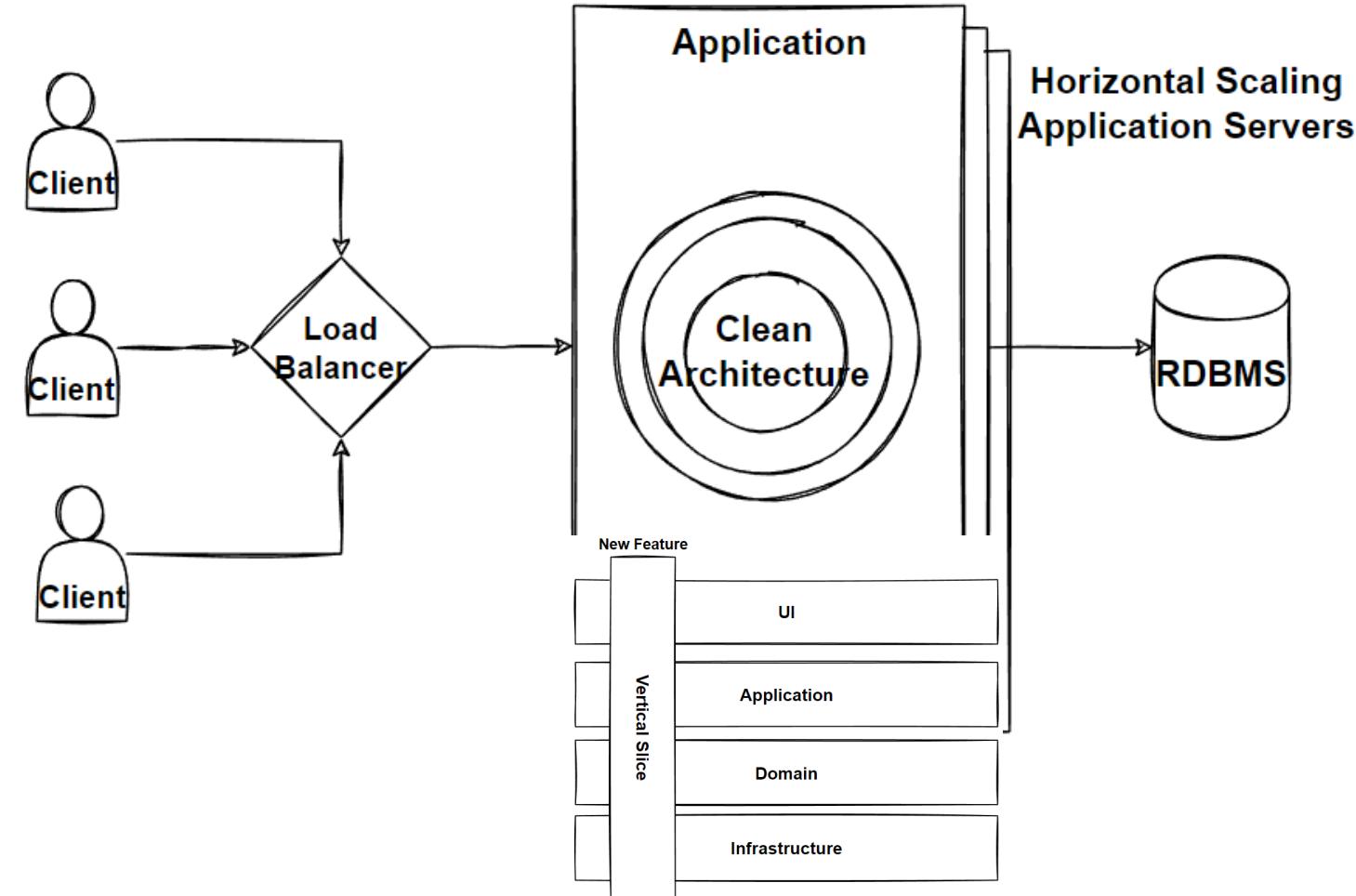


Problems

- Our E-Commerce Business is growing
- Business teams are separated teams as per departments; Product, Sale, Payment..
- And all teams wants to add new features to compete the market
- Codebase not allowed to manage it
- Context Switching and Vertical Slices problems on Clean Architecture

Solutions

- Modular Monolithic Architecture



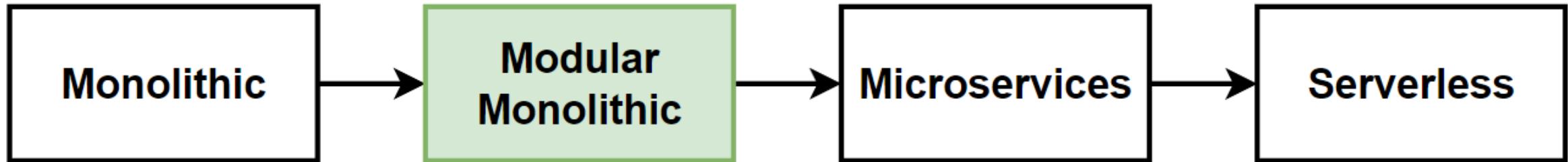
Modular Monolithic Architecture

Benefits and Challenges of Modular Monolithic Architecture

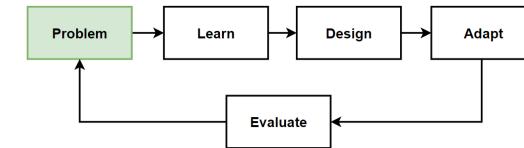
The Dependency Rule of Modular Monolithic Architecture

Design our E-Commerce application with Modular Monolithic Architecture

Architecture Design Journey



Problem: Agility of New Features, Split Agile Teams

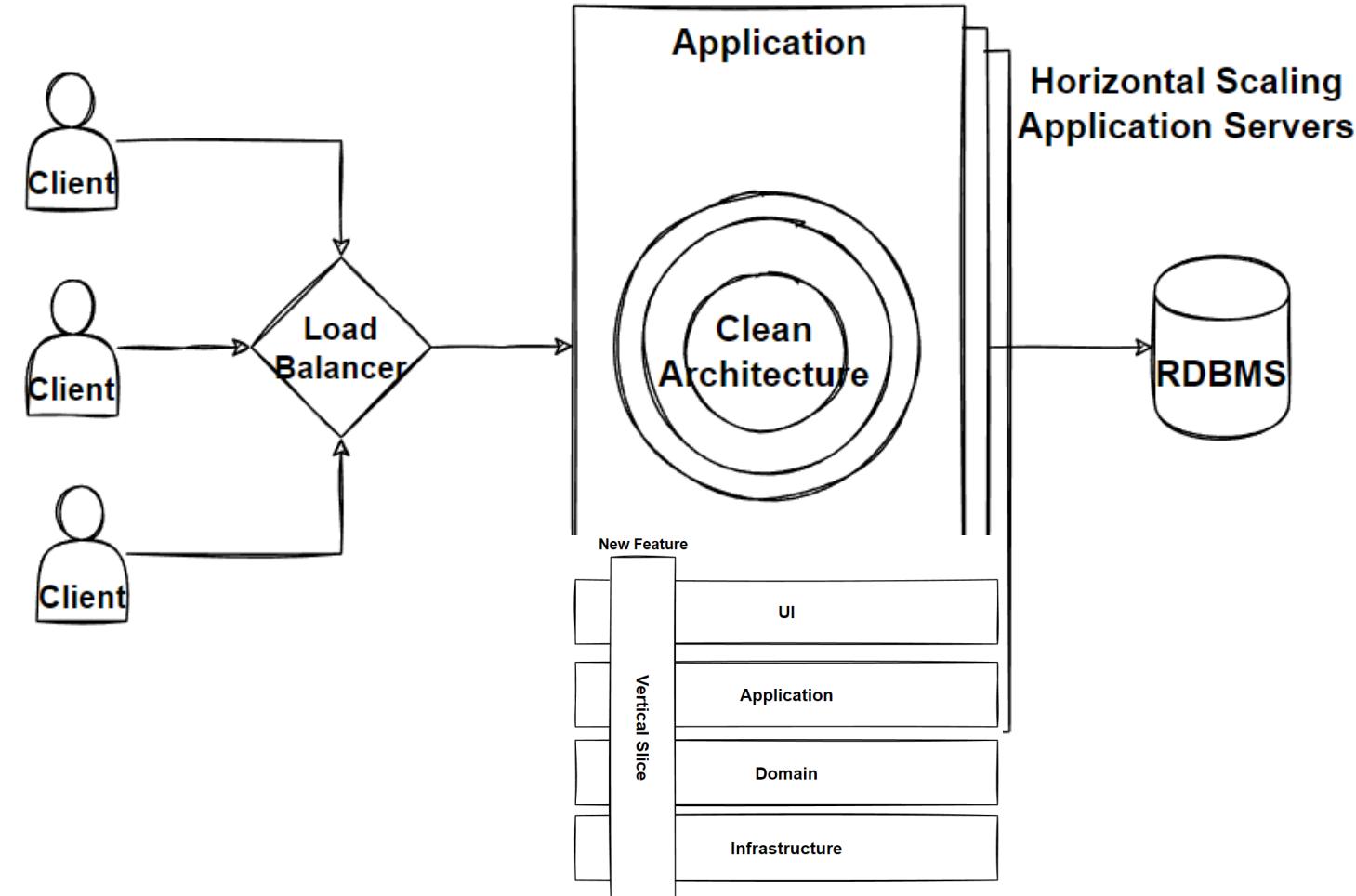


Problems

- Our E-Commerce Business is growing
- Business teams are separated teams as per departments; Product, Sale, Payment..
- And all teams wants to add new features to compete the market
- Codebase not allowed to manage it
- Context Switching and Vertical Slices problems on Clean Architecture

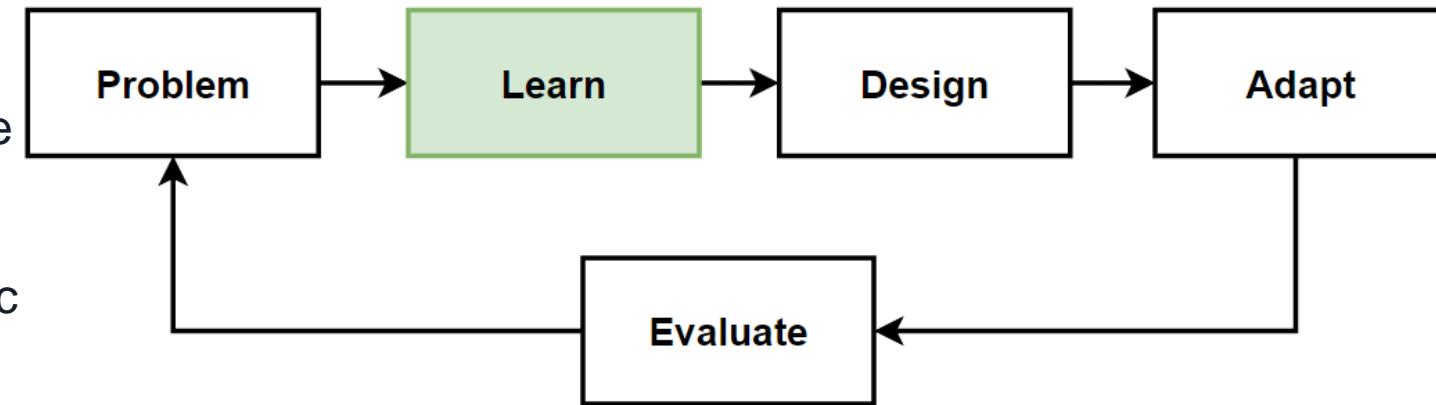
Solutions

- Modular Monolithic Architecture



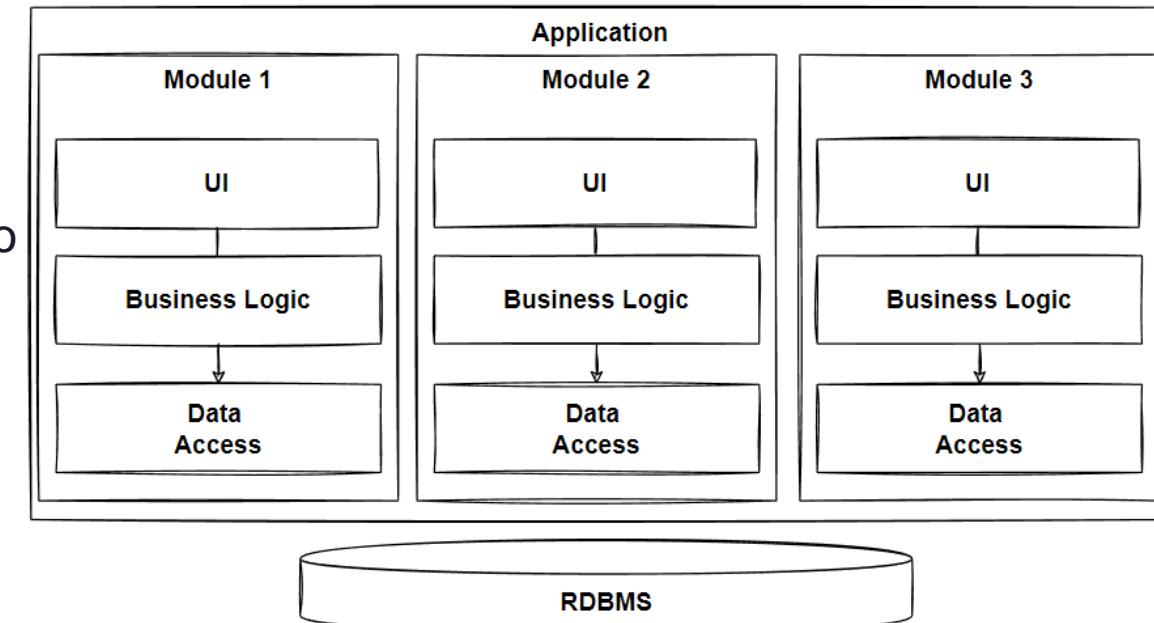
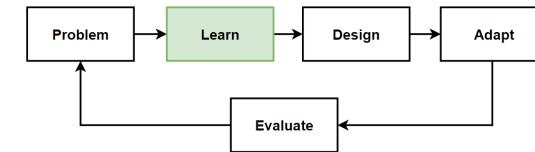
Learn: Modular Monolithic Architecture

- Modular Monolithic Architecture
- When to use Modular Monolithic Architecture
- Benefits of Modular Monolithic Architecture
- Challenges of Modular Monolithic Architecture
- Modular Monolithic Architecture Pros-Cons
- Reference Architectures of Modular Monolithic



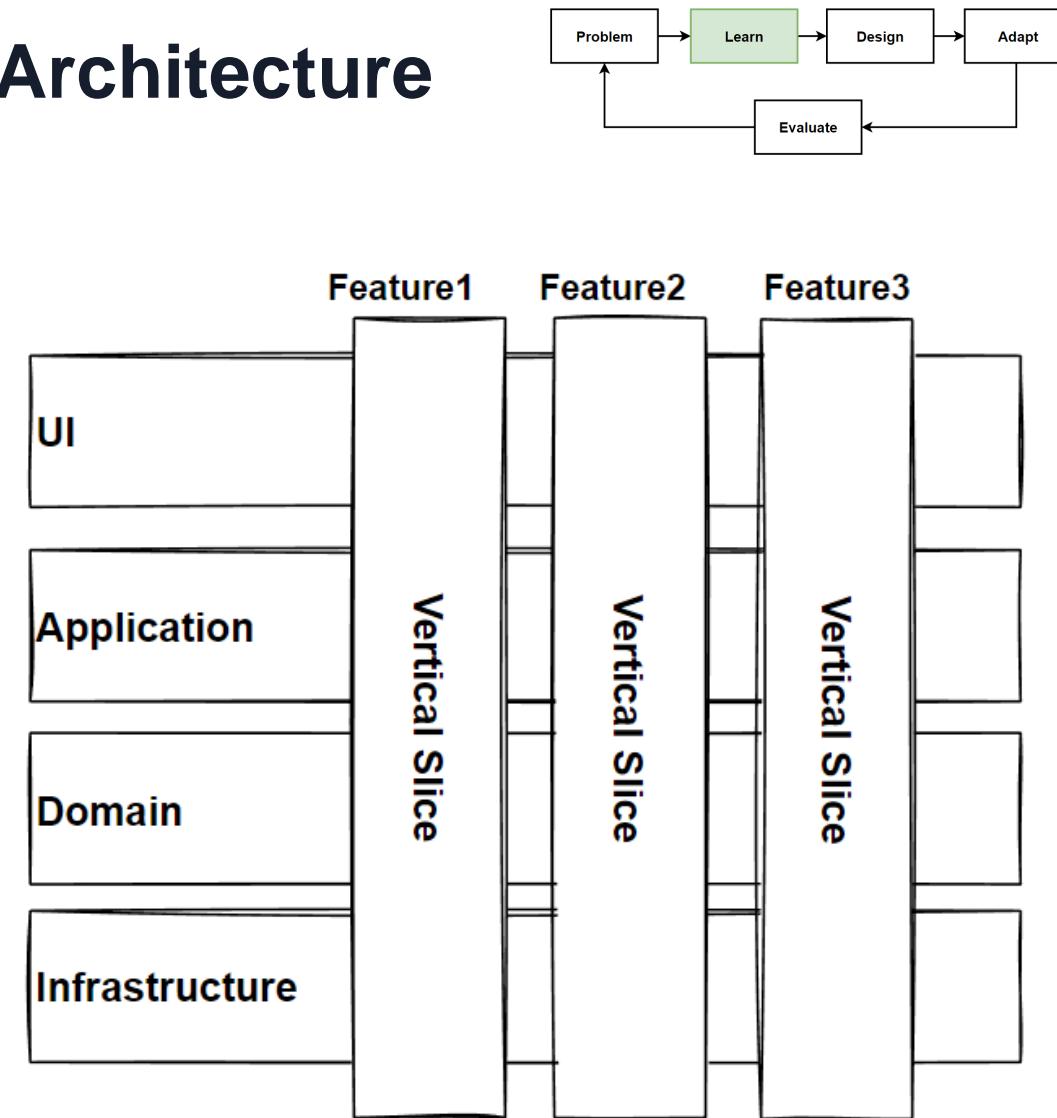
What is Modular Monolithic Architecture ?

- Modular monolithic architecture **divided** our application logic **into modules** and each module will be **independent** and **isolated** and should have its **own business logic**, database schema.
- Each module can follow their **own logical separations**, layered architecture style or clean architecture.
- The Modular Monolith architecture **breaks up the code** into **independent modules**, each module **encapsulates** their **own features**.
- Modules are represents **Bounded Context** of our application domain and we **group features of Domain contexts** in modules.
- **Reduces the dependencies** of a module and we can develop or modify a module **without it effecting other modules**.

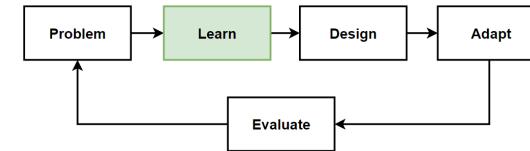


Vertical Slices with Modular Monolithic Architecture

- Instead of using layered architecture with horizontal logical layers, we can **organize our code across vertical slices of business functionality**.
- These **slices** are determined based on **business demands**, rather than enforced by technical constraints.
- When we add or change a feature in an application, our changes are **scoped to the area of business concern** not technical logical layers.
- With the modular monolith architecture, organize our codes as a **vertical slices**, and as our system continues to grow, organizing our code around of **business functionalities** into Modules.
- **Modules** can be a potential **microservices** when need to independently deployed and scale in the future **refactorings** our architecture.



E-Commerce with Modular Monolithic Architecture

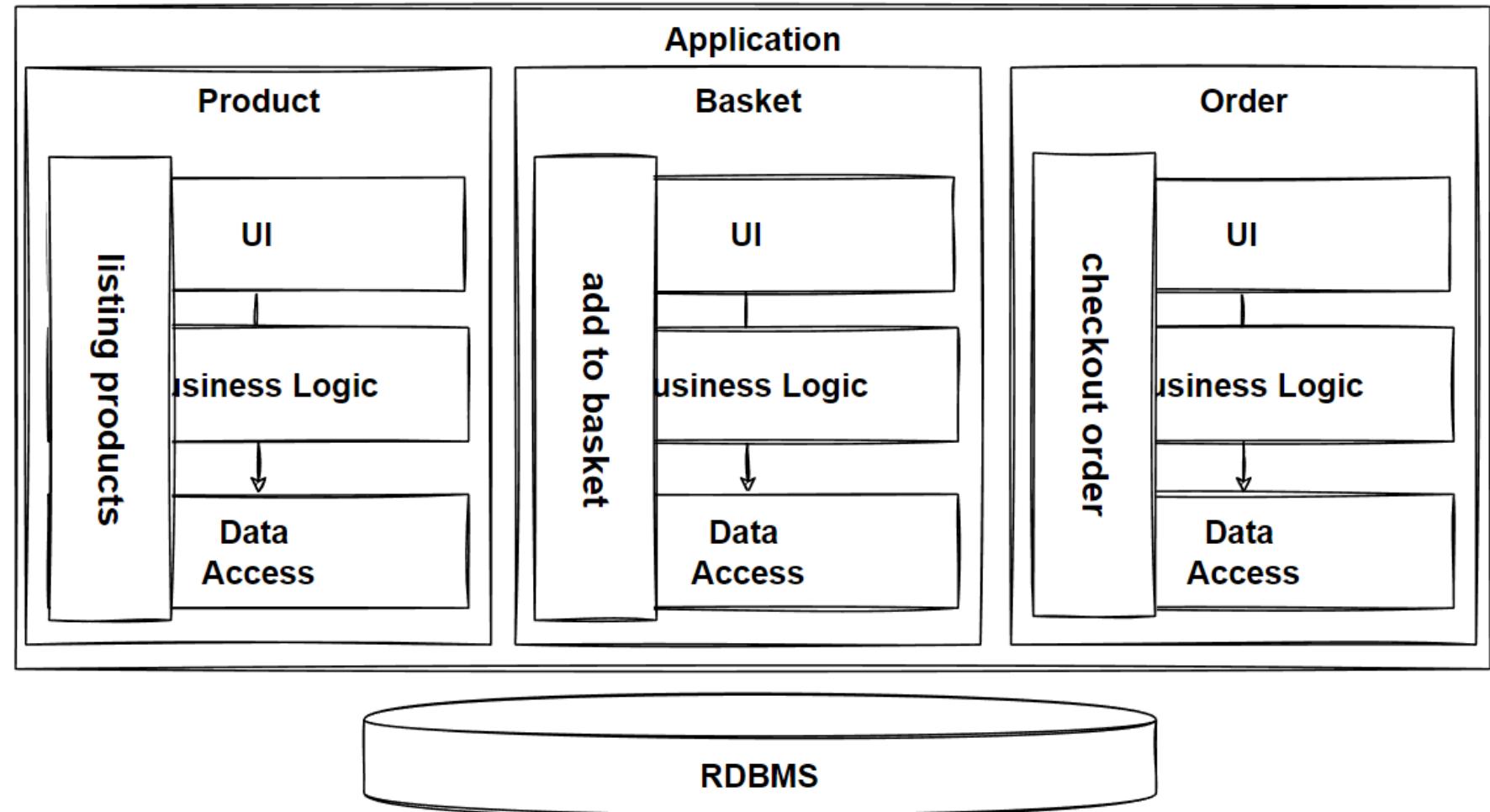


Example Use Cases

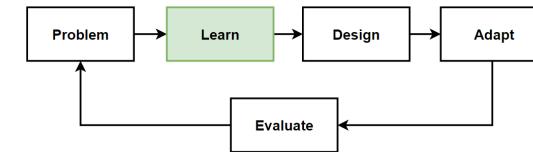
- Listing products = Product Module
- Add to basket = Basket Module
- Checkout order = Order Module

Main Benefits

- Reduced complexity
- Easier to refactor
- Better for teams



Benefits of Modular Monolithic Architecture



- **Encapsulate Business Logic**

Business logics are encapsulated in Modules and it enables high reusability, while data remains consistent and communication patterns simple.

- **Reusable Codes, Easy to Refactor**

For large development teams, developing modular components of an application will increase reusability. Modular components can be reused that can help teams establish a single source of truth.

- **Better-Organized Dependencies**

With modular monoliths architecture, application dependencies will be more organized and visible. This will help developers to easily assess which parts of the application require which dependencies.

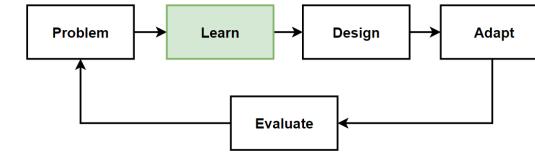
- **Less-Complex than Microservices Architecture**

Easier to manage a modular monolith rather than hundreds of microservices, because Modular Monolithic comes with basic underlying infrastructure and operational costs low.

- **Better for teams**

Easier for developers to work on different parts of the code. with Modular Monolithic architecture, we can divide our developer teams effectively and implement business requirements with minimum affect to each other.

Challenges of Modular Monolithic Architecture



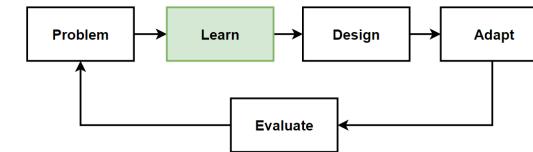
- **Can't diversifying technology**

Modular monoliths don't provide all benefits of microservices. If you need to diversifying technology and language choices, you can't do it with Modular Monolithic Architecture. These types of polyglot technology stacks can't use with Modular Monolithic Architecture.

- **Can't Scale and Deploy Independently**

Since the application is a single unit, it can't be scale separated parts or deploy independently like microservices. And this kind of applications has to move microservices due to reaching out scalability limits and also performance issues.

When to use Modular Monolithic Architecture



- **Strict Consistency is Mandatory Cases**

For many companies unable to make the move to microservices, due to their database and data not appreciate for distributed architecture.

- For example if your application store **high important data** like debit on bank account, then you need **strong data consistency** that means your data should be correct for every time, if you got any exception you have to rollback immediately.

- **Modernization**

If you already have a big complex monolithic application running, the modular monolith is the perfect architecture to help you refactor your code to get ready for a potential microservices architecture.

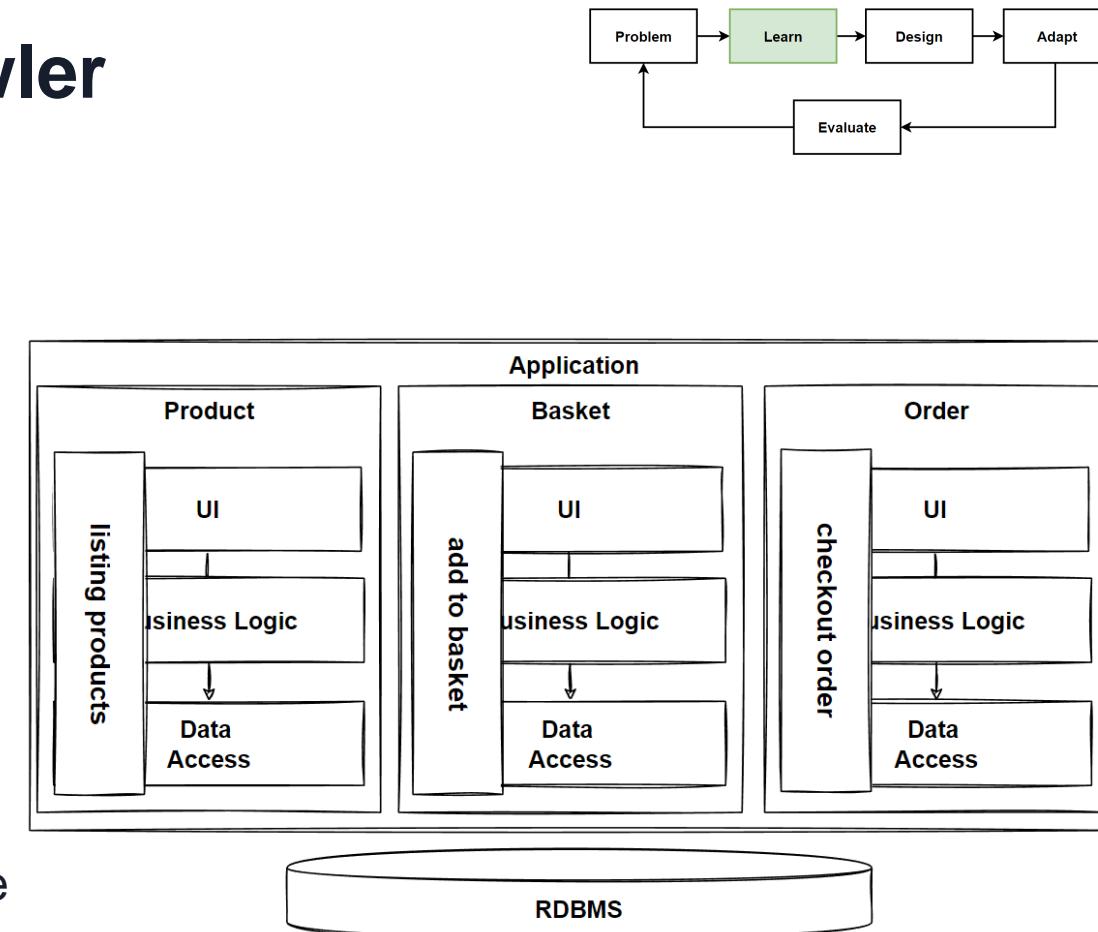
- Instead of **jumping into microservices**, you can move modular monolithic without effecting your business and get benefits like speed up with a well-factored modular monolith.

- **Green Field Projects**

A modular monolith allow you to learn your domain and pivot your architecture much faster than a microservices architecture. You **won't have to worry** about things like **Kubernetes** and a **services mesh** at day1. Your deployment topology will be drastically simplified.

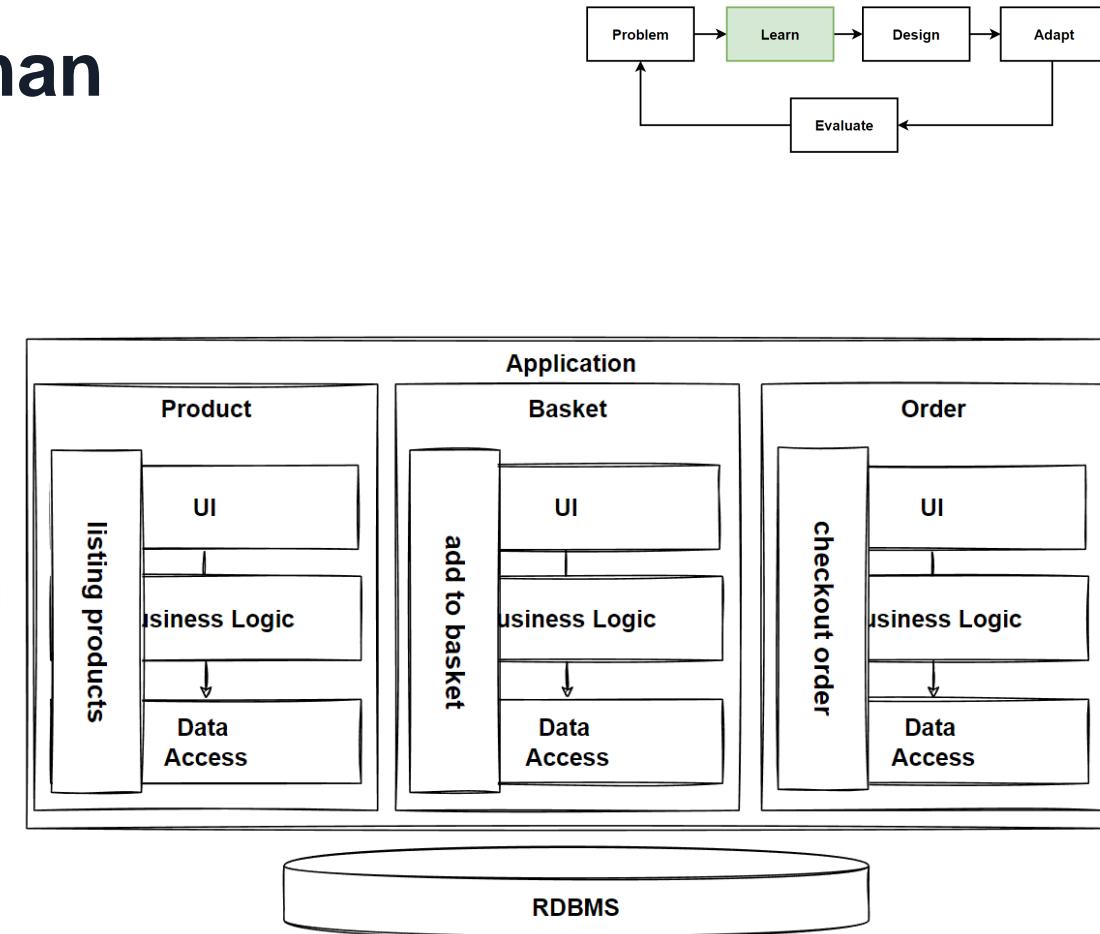
Monolith First Approaches, Martin Fowler

- Martin Fowler: Monolithic First, 2015
- <https://martinfowler.com/bliki/MonolithFirst.html>
- **Successful microservice stories have started with a monolith**
- A system that was **built as a microservice** system from scratch, it has ended up in **serious trouble**.
- **Microservices** are a useful architecture, but it incurs a **significant complexity**, which means they are only **useful with more complex systems**.
- Consider **Yagni**. When you begin a new application, how sure are you that it will be useful to your users?
- Starting with microservices works well if you come up with good, stable boundaries between the services which is **BoundedContexts**.



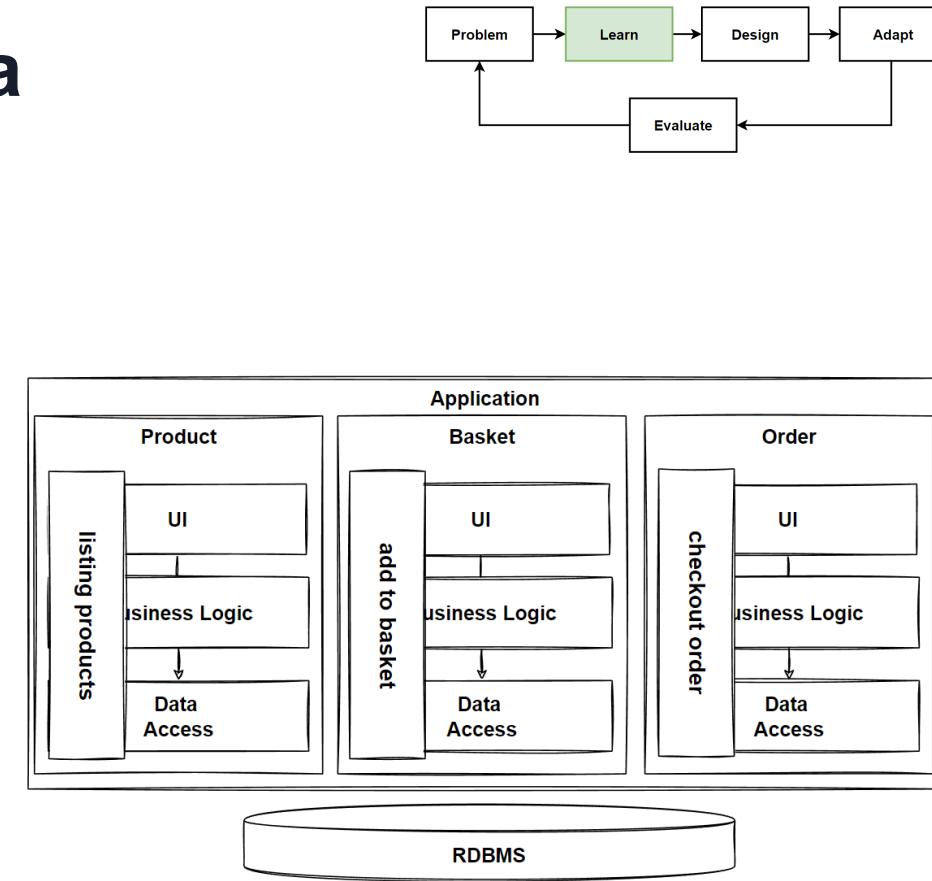
Monolith First Approaches, Sam Newman

- Sam Newman — Building Microservices, 2nd Edition
- https://samnewman.io/books/building_microservices/
- He agrees with the Martin Fowler approach which is recommend **beginning development** of new projects and systems as a **single deployable unit** — the monolith.
- Leveraging microservices only if you can **become convinced** of the benefits for your system, not as a default for every project.
- *A monolithic architecture is a choice, and a valid one at that. I'd go further and say that in my opinion it is the sensible default choice as an architectural style. In other words, I am looking for a reason to be convinced to use microservices, rather than looking for a reason not to use them.*

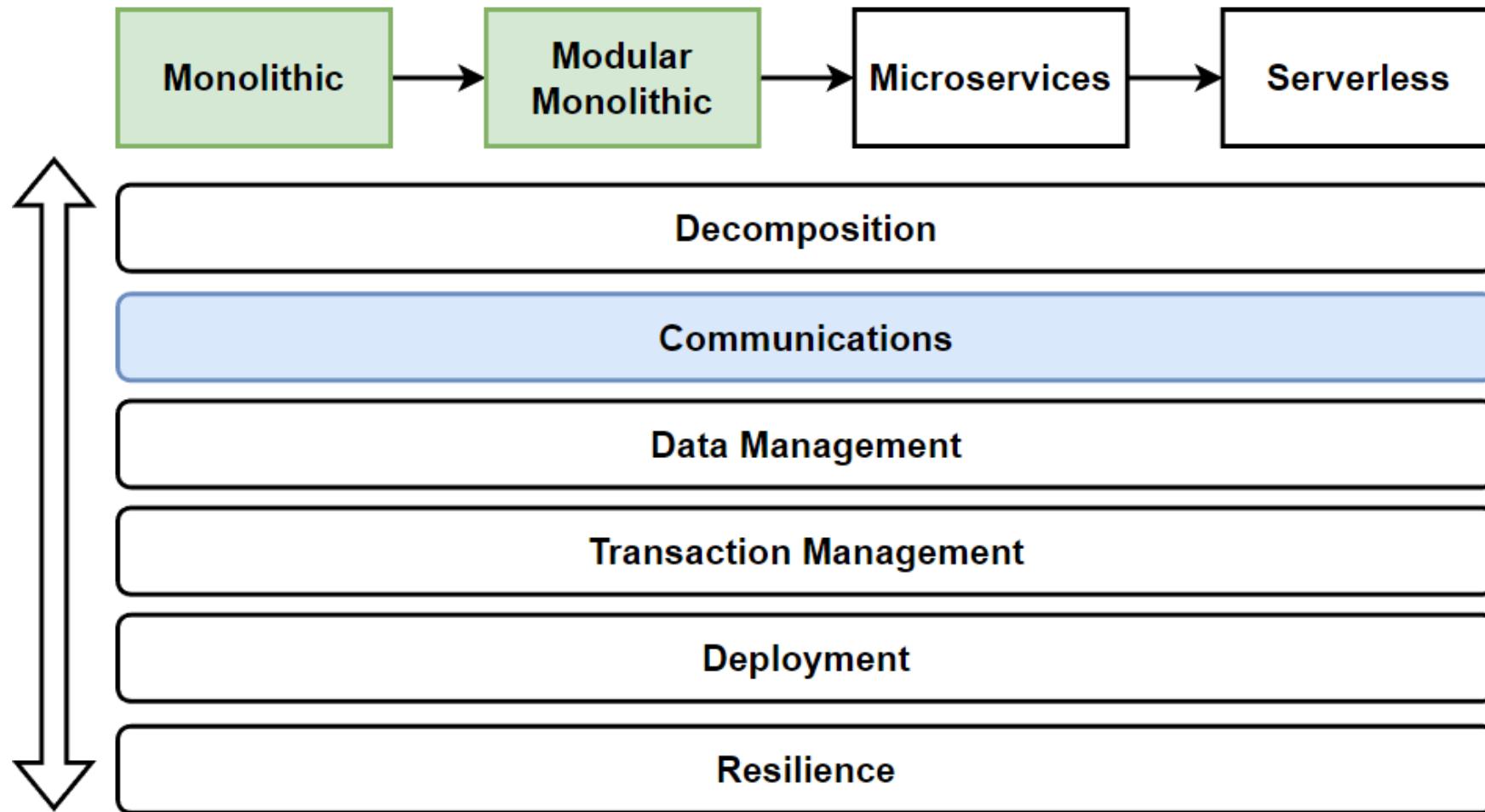
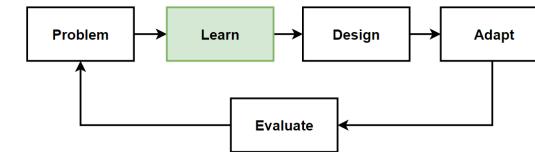


Monolith First Approaches, Mehmet Ozkaya

- Monolithic architecture does not mean that the system is **poorly designed or bad**. Instead
- If you follow Modular approach, It will come with lots of similar benefits with microservices like re-usability, easy to Refactor, Better organized dependencies and teams.
- So Should we always start with Monolith First Approaches ? – **NO**
- How we can decide ? If
- **Strong Consistency is Mandatory & Independent Scale and Deploy is not Required = Modular Monolithic**
- **Strong Consistency is not Mandatory, Eventual Consistency is OK & Independent Scale and Deploy is Required = Microservices**

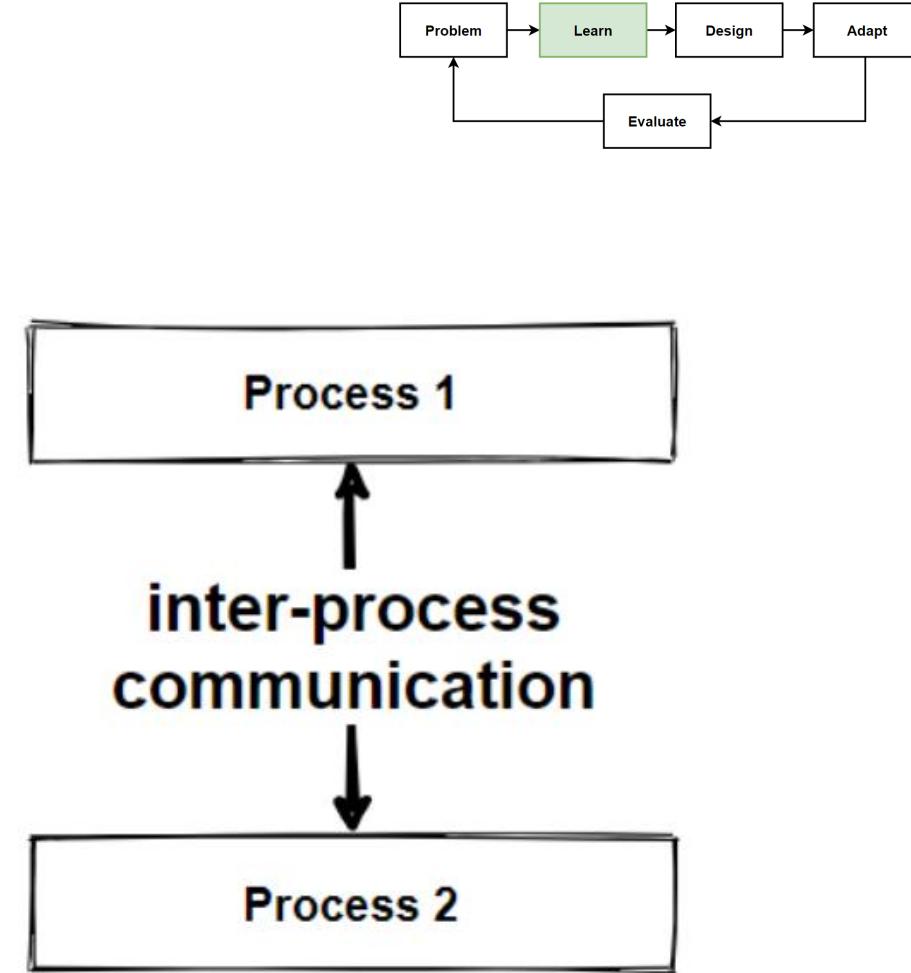
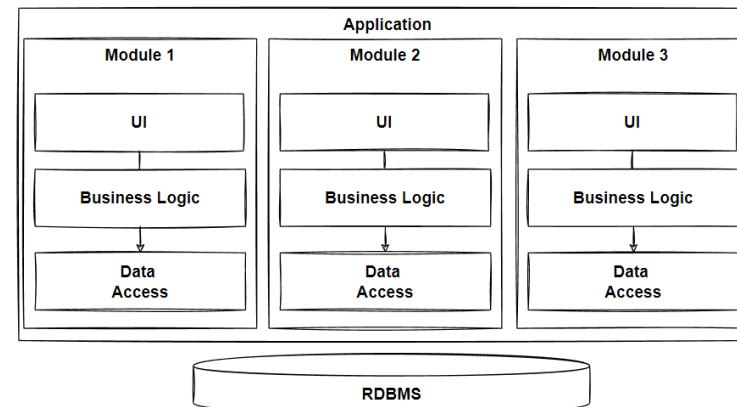


Monolithic Architecture Vertical Considerations

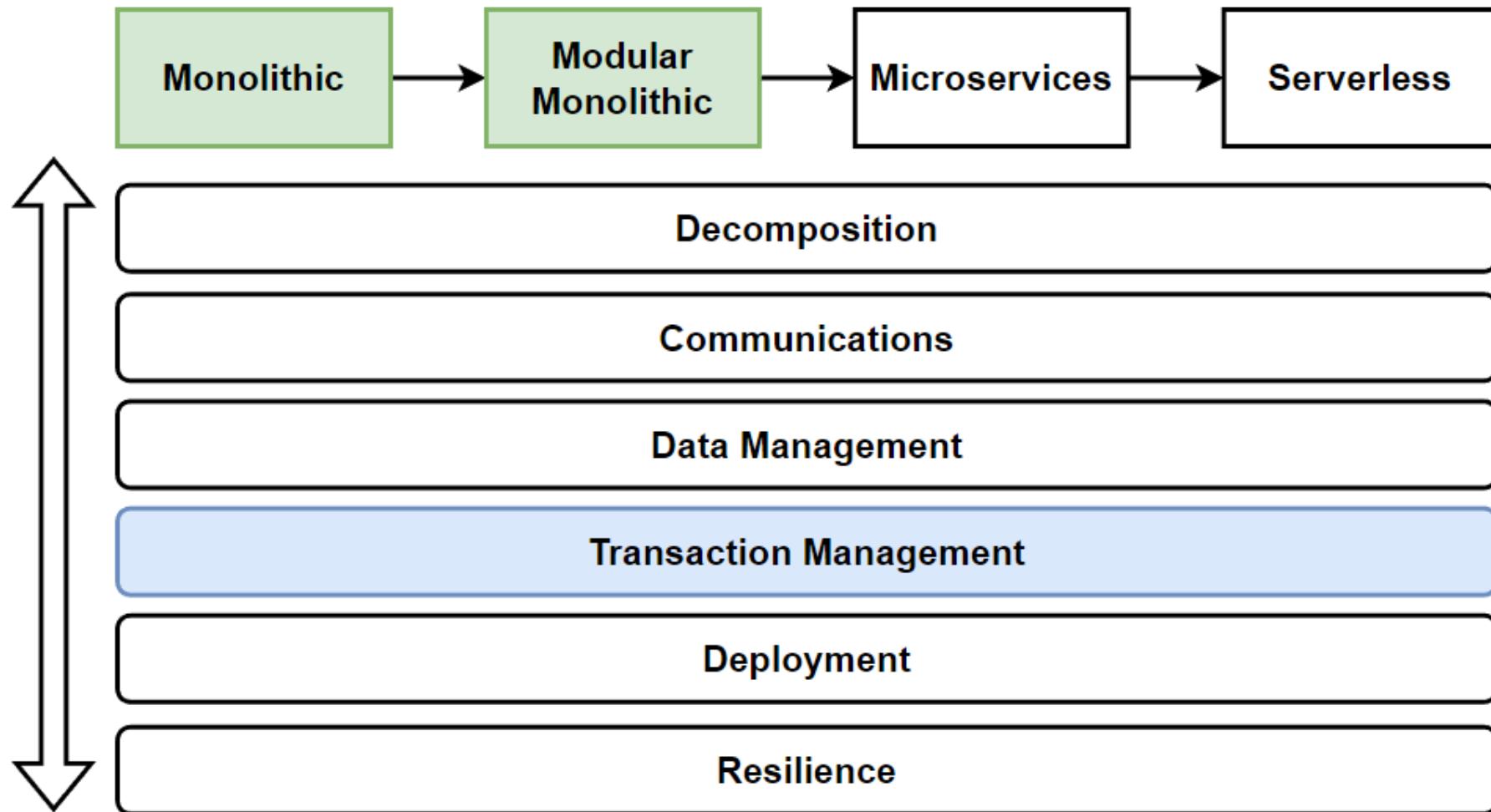
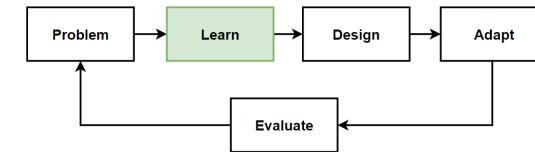


Communication of Monolithic Architecture

- **Monolithic application** sitting in the **same server** with all modules.
So we **don't need** to make any **network call**.
- It is very easy and **fast** to **communicate** between **modules**.
- The communication will be **Inter-Process Communication**.
- Inter-process communication is the mechanism provided by the operating system that allows **processes** to **communicate** with each other.
- It processes to communicate with each other **by method calls** into **the code**.

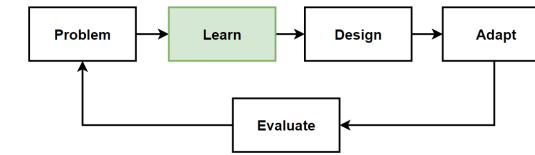


Monolithic Architecture Vertical Considerations



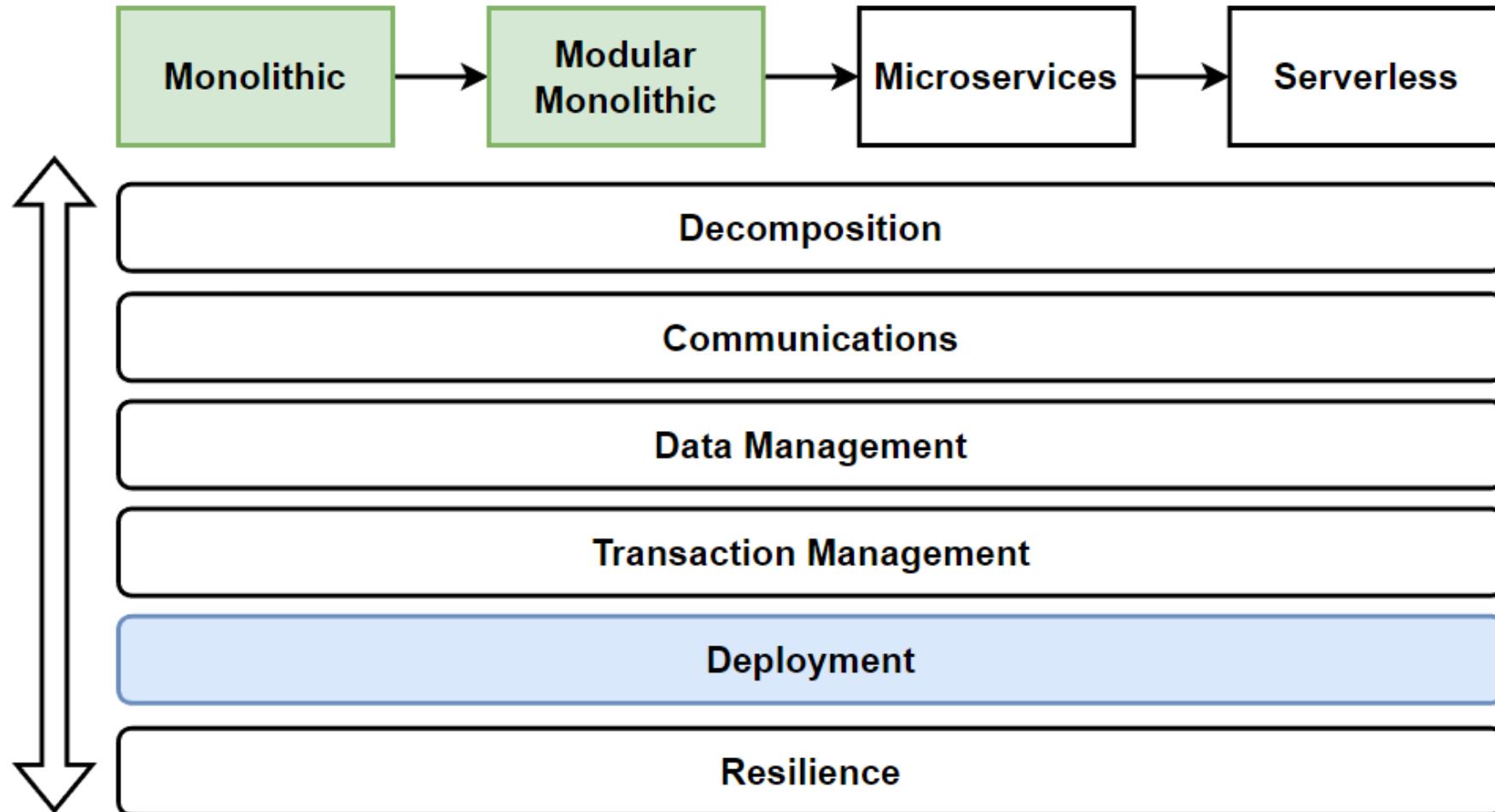
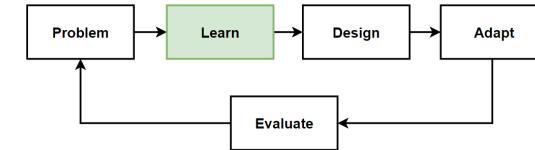
Transaction Management of Monolithic Architecture

- **Transaction management** in Monolith architecture is quite easy compared to Microservice Architecture. Many frameworks or languages contains some mechanism for transaction management.
- These mechanism have a **single database** of the **whole application**. They are developed for scenarios where all transactions are running on a **single context**.
- Simply **commit** and **rollback** operations with these mechanism in monolith architectures.
- Transactions operated in the **transaction scope** are kept in memory without writing to the database until they are **committed**, and if a **Rollback** is made at any time, all transactions in the scope are **deleted** from memory and the **transaction is canceled**.
- When **Commit** is **written** to the **database**, the transaction is **completed successfully**.

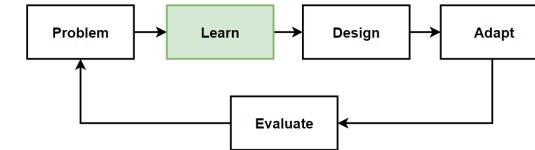


```
function place_order()
    do_payment
    decrease_stock
    send_shipment
    generate_bill
    update_order
```

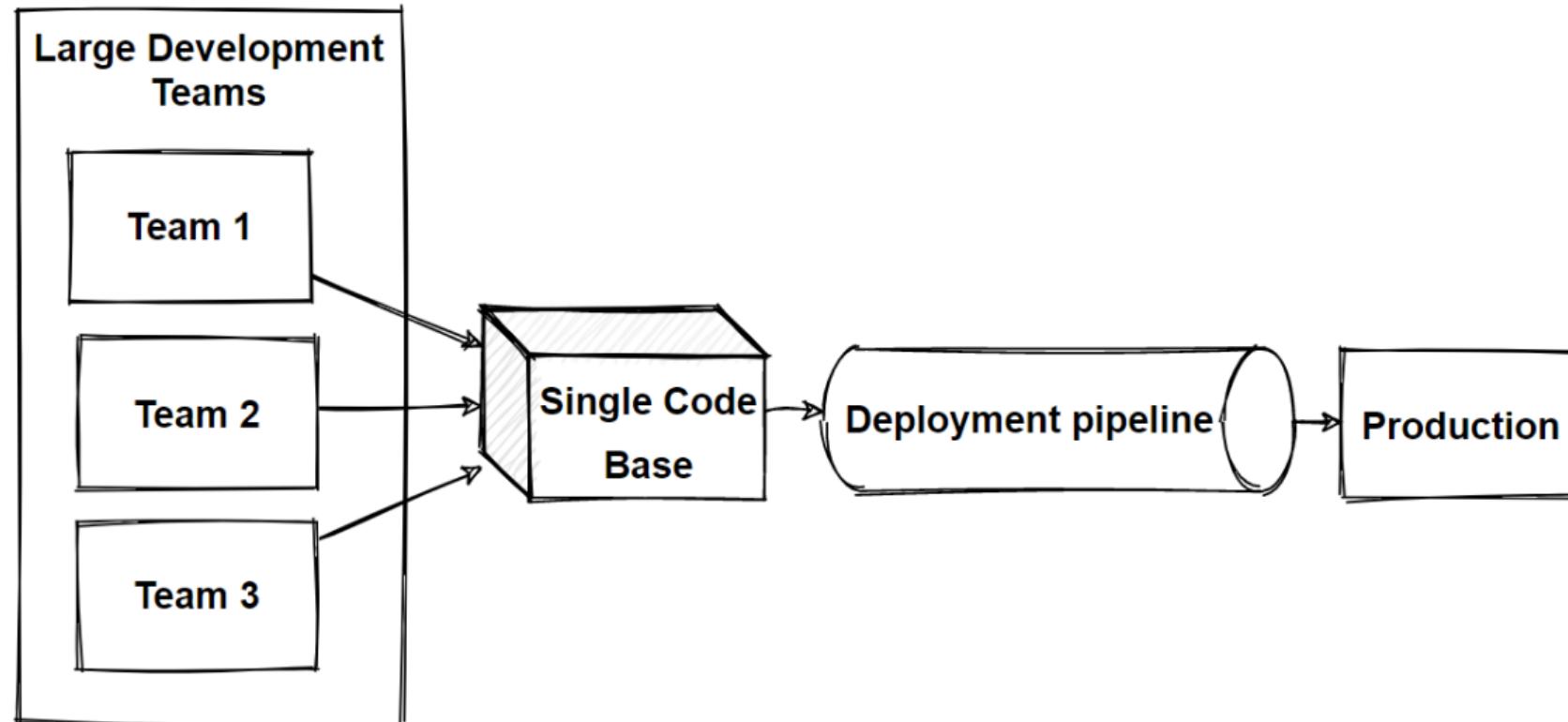
Monolithic Architecture Vertical Considerations



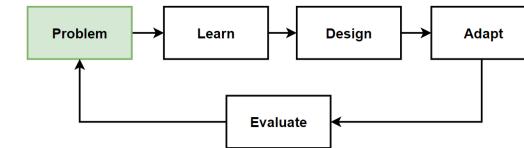
Deployments of Monolithic Architecture



- **Single code base** harder to implement **new changes** especially in a large and complex application.
- Any code change **affects the whole system**. Even the **smallest change** requires **full deployment** of the entire application.
- Pain point of Monolithic Architectures that is not reliable that a **single bug** in any module can **bring down** the whole monolithic application.



Problem: Agility of New Features, Split Agile Teams

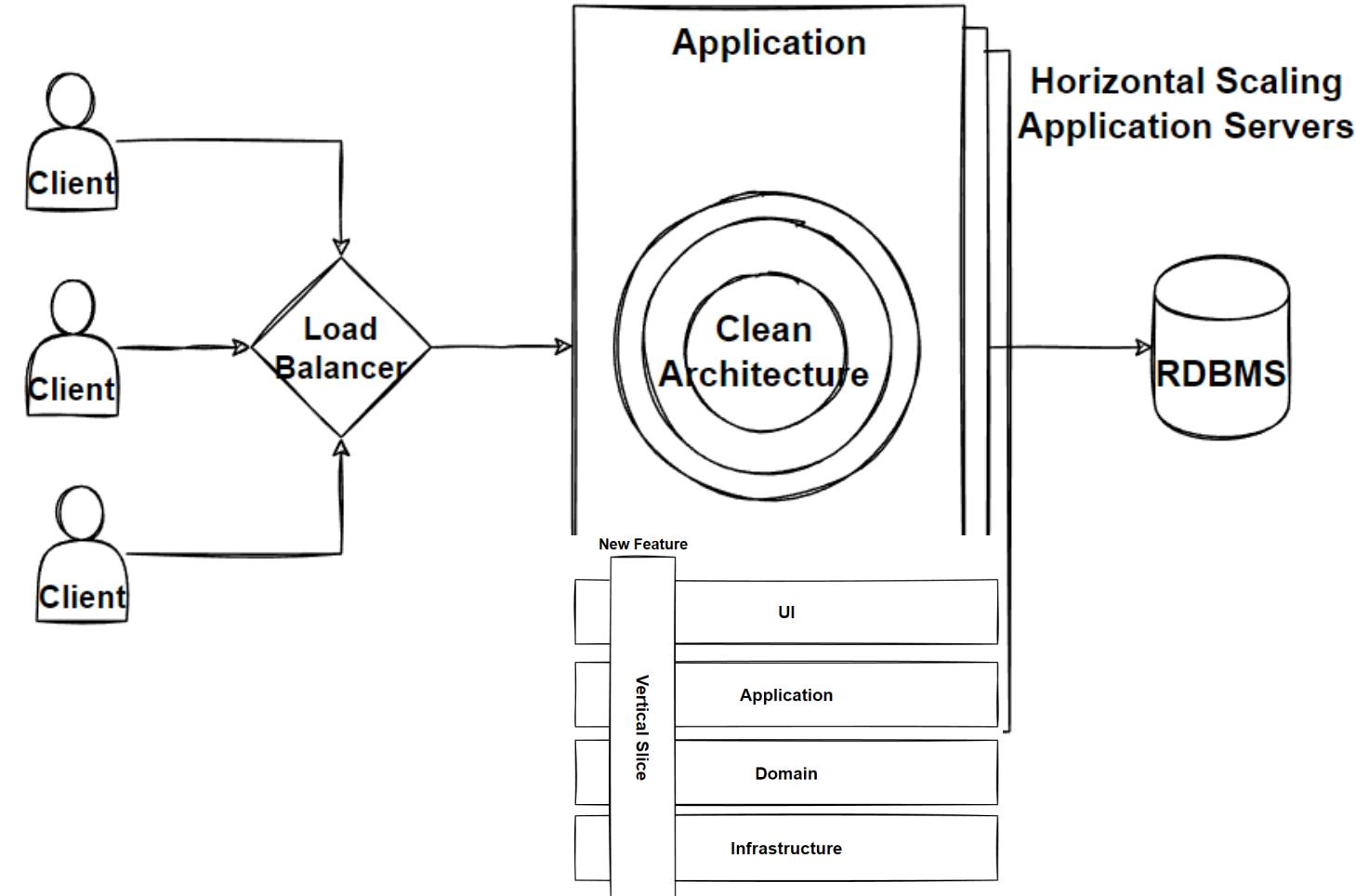


Problems

- Our E-Commerce Business is growing
- Business teams are separated teams as per departments; Product, Sale, Payment..
- And all teams wants to add new features to compete the market
- Codebase not allowed to manage it
- Context Switching and Vertical Slices problems on Clean Architecture

Solutions

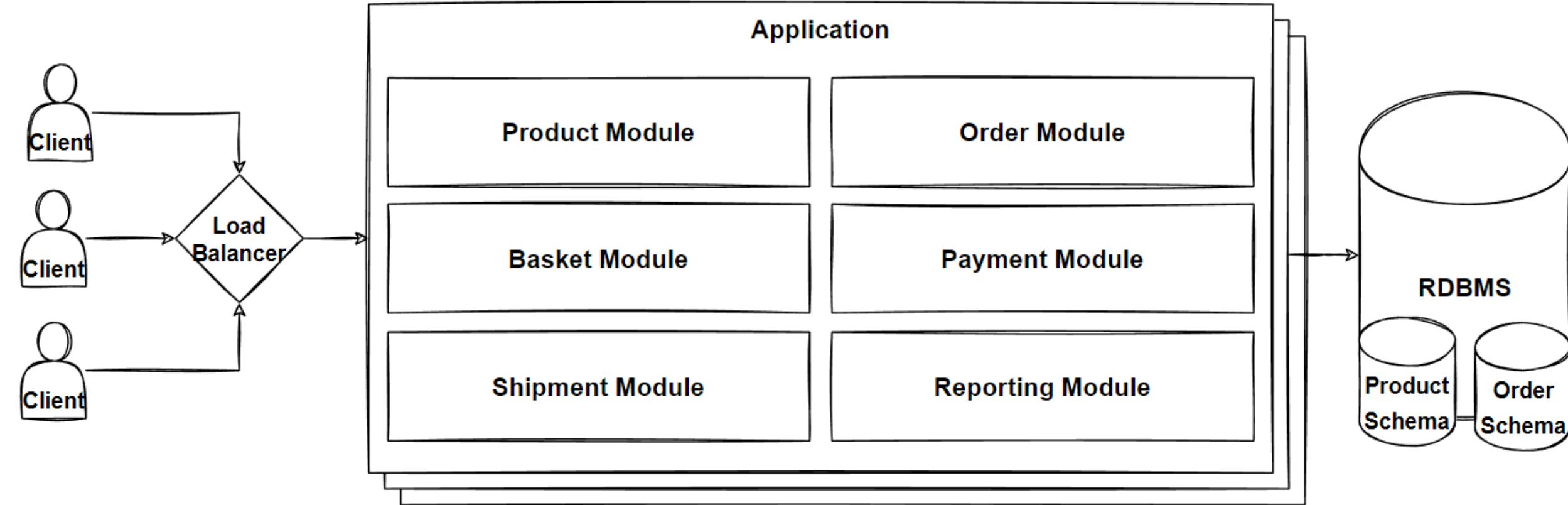
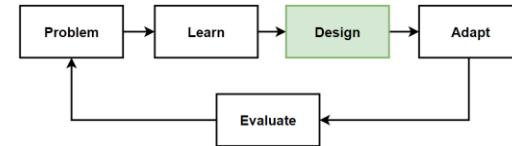
- Modular Monolithic Architecture



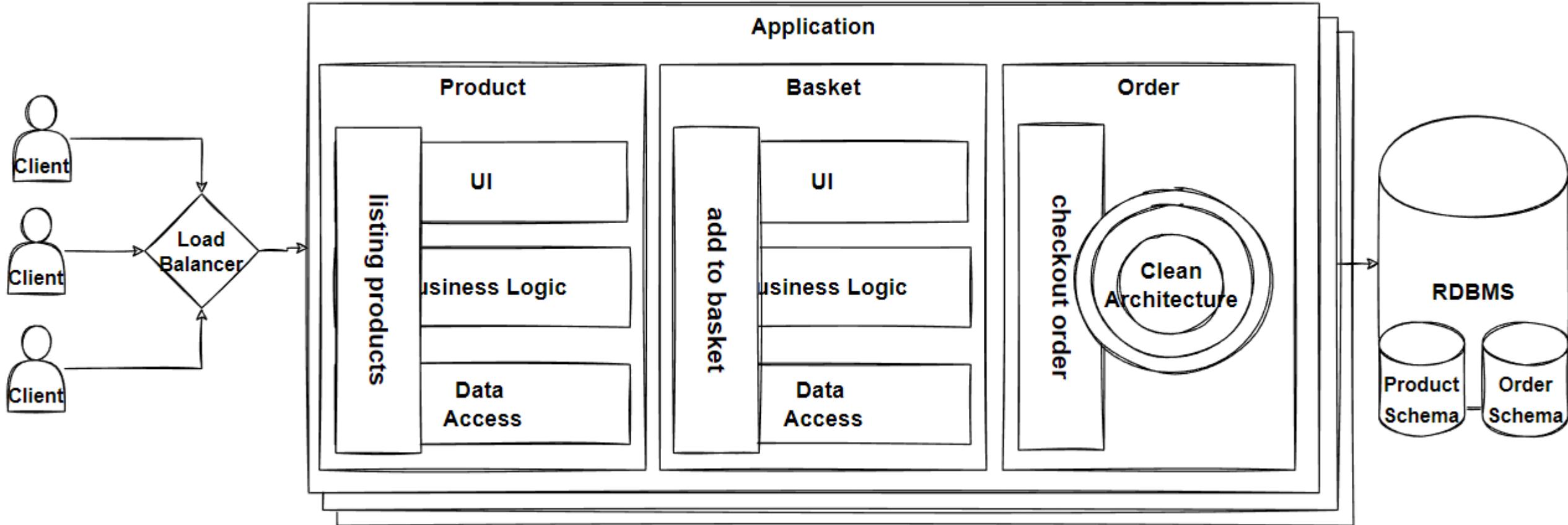
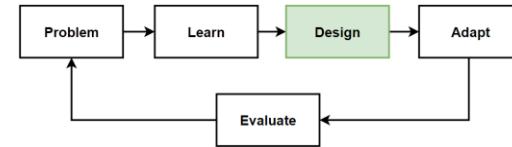
Before Design – What we have in our design toolbox ?

Architectures	Patterns&Principles	Non-FR	FR
• Monolithic Architecture	• KISS • YAGNI	• Availability • High number of Concurrent User	• List products
• Layered Architecture	• Separation of Concerns (SoC)	• Maintainability	• Filter products as per brand and categories
• Clean Architecture		• Flexibility	• Put products into the shopping cart
• Modular Monolithic Architecture	• SOLID • The Dependency Rule • Horizontal Scaling • Load Balancer • Monolithic-First Strategy	• Testable • Scalability • Reliability • Re-usability	• Apply coupon for discounts • Checkout the shopping cart and create an order • List my old orders and order items history

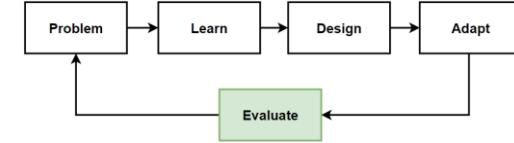
Design: Modular Monolithic Architecture



Design: Modular Monolithic Architecture-Internal



Evaluate: Complexity of Presentation UI Operations

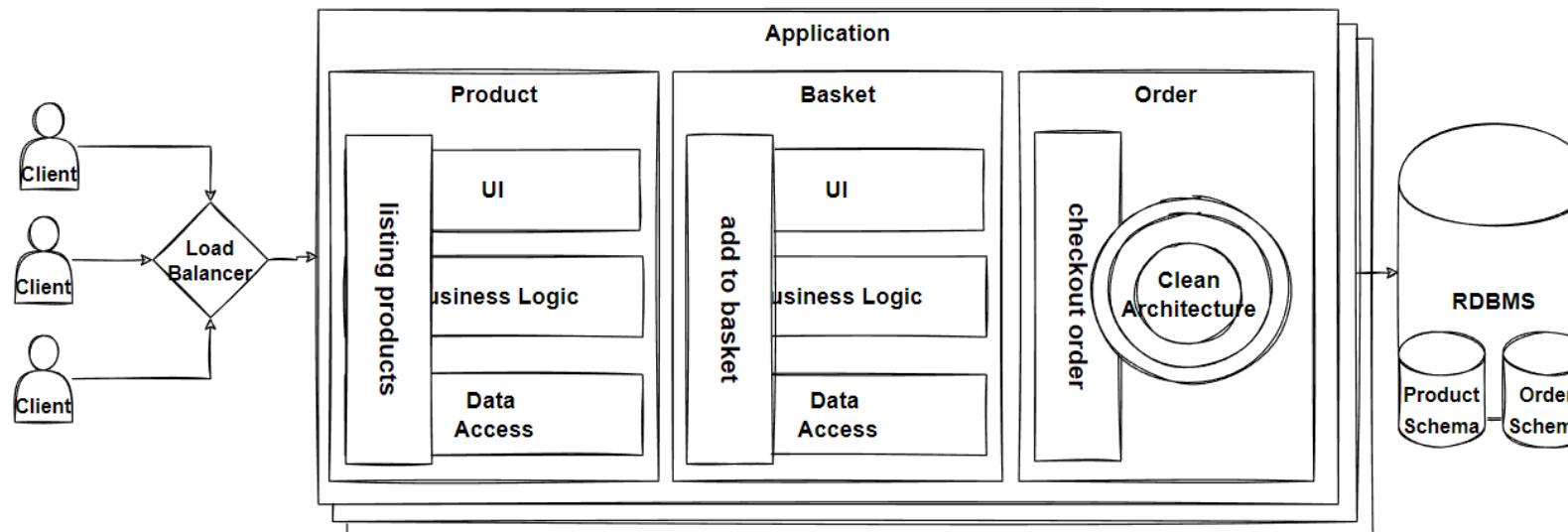


Benefits

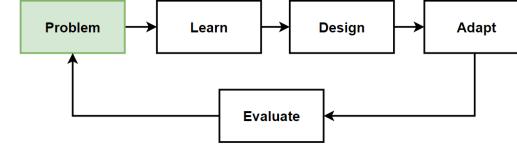
- Easy Development, Debug and Deploy
- Encapsulated Vertical Slices Modules

Drawbacks

- UI operations are handled in our big monolithic application.
- With adding new features, the Complexity of Presentation UI Operations are going to be nightmare.
- UI layer is generated from server-side, every module is trying to generate their own page.
- UI Layer has different minor requirements, requires full deployment.
- Business teams has UI related requirements are increased.



Problem: Improve Customer Experience with SPA

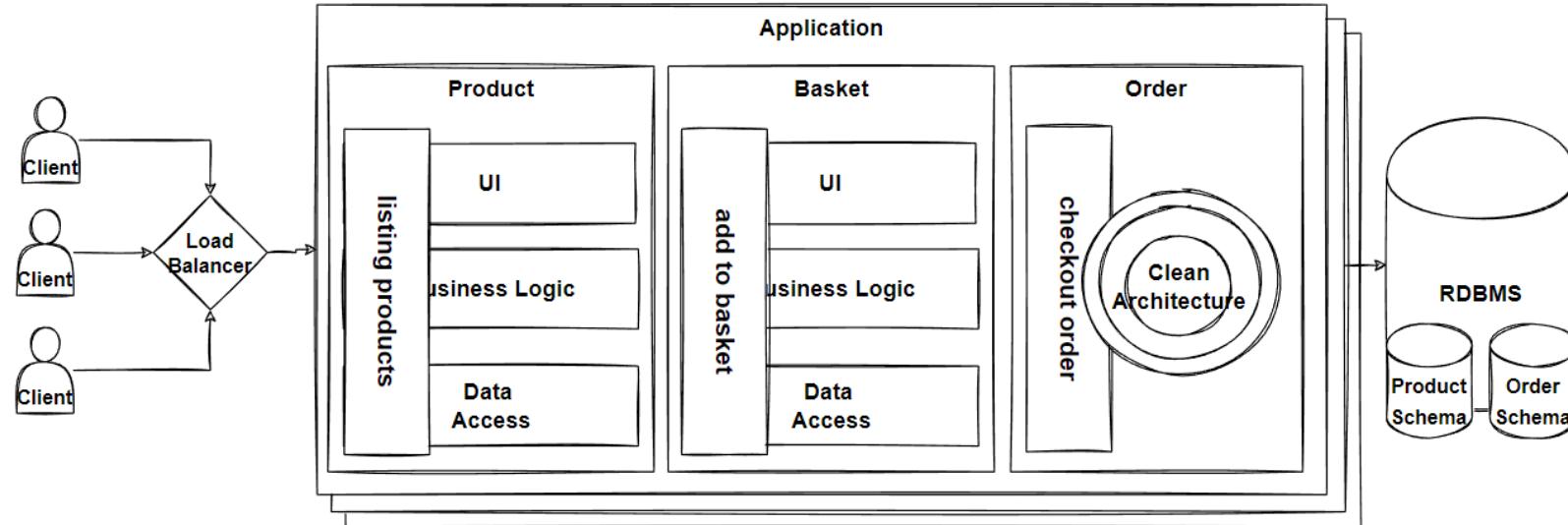


Problems

- Our E-Commerce Business is growing
- Improved customer experience with Separated UI and Omnichannel
- Responsive Pages with SPA
- Omnichannel expectations

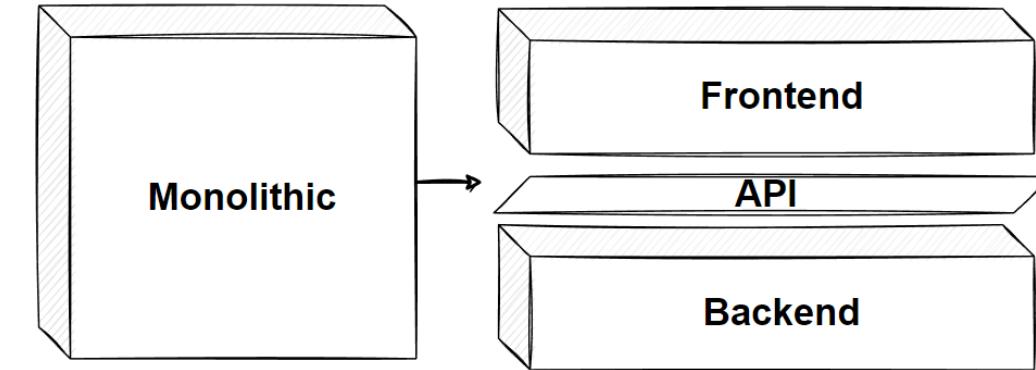
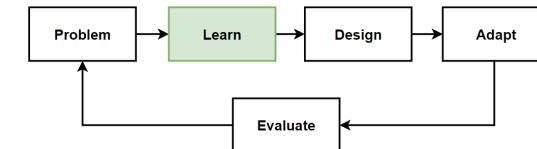
Solutions

- Separated Presentation with SPA
- Separate FrontEnd - BackEnd
- Headless Architecture



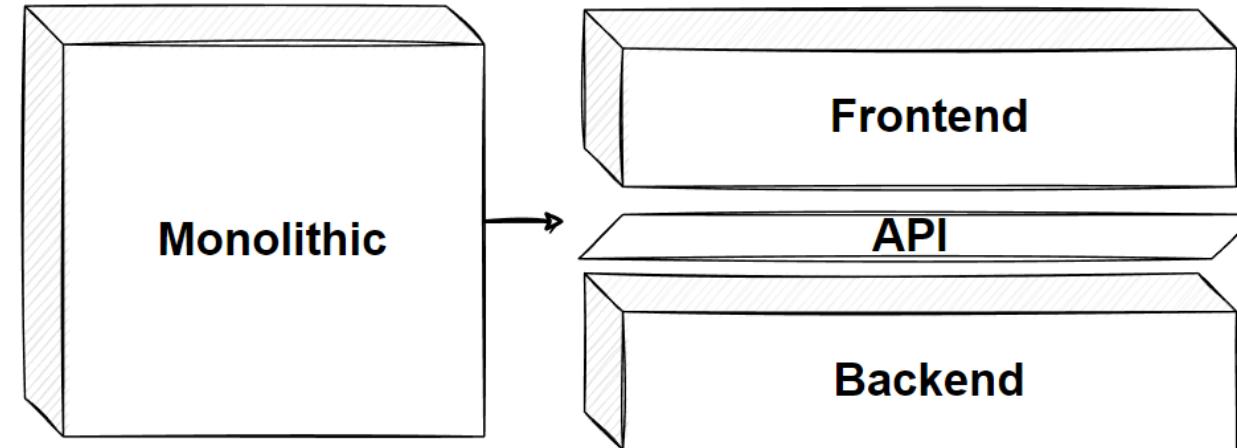
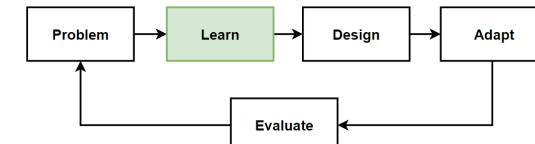
Headless Architecture and Separated UI-SPA

- **Headless architecture** separates the frontend from the backend layer of the application. Separates UI and business logic.
- Headless architecture emphasizes mainly **decoupling frontend and backend layers**, and it is the first step before moving to microservices.
- It uses **APIs** to connect the front and backends applications.
- **Application programming interfaces (APIs)** are software intermediaries that enable communication between applications.
- **REST APIs** which created from Backend application are consumed from the Frontend Application. These API consume operations are handled in **SPA** application in frontend side.
- **Single page applications (SPA)** are applications contained in a single web page without having full-page reload, and able to update some portion of the page with responsive way.



Benefits of Headless Architecture

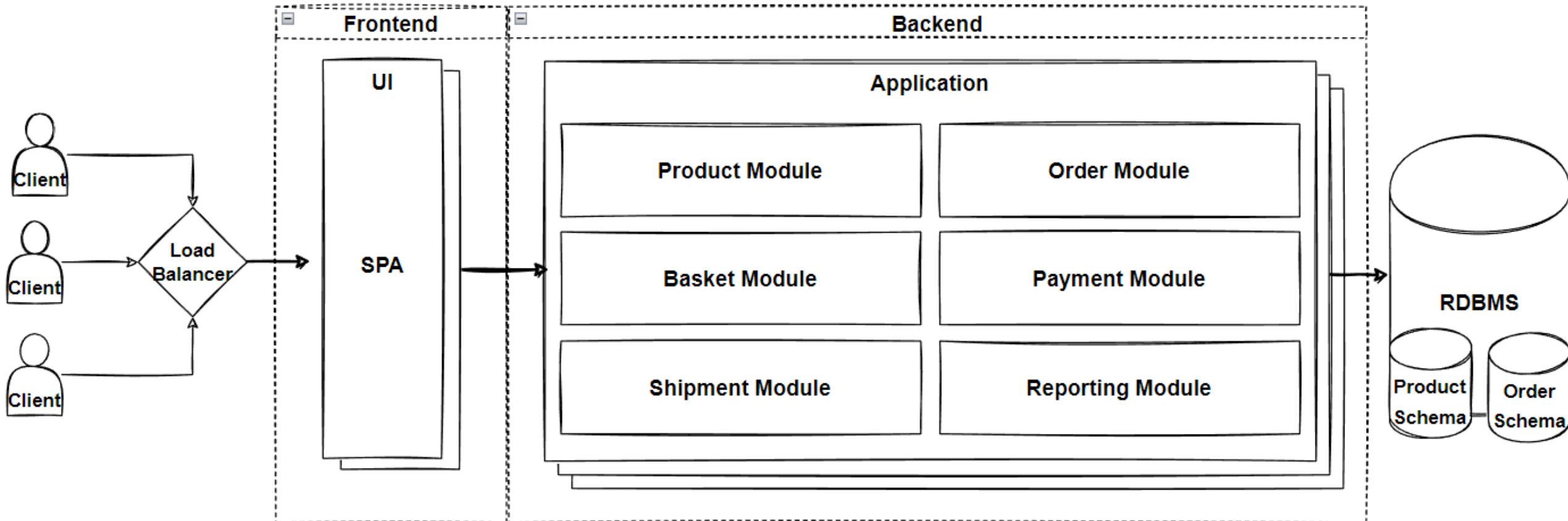
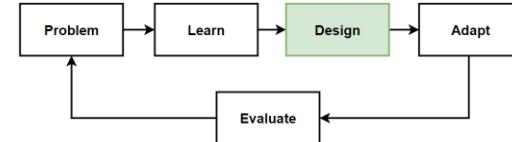
- Headless architecture **separates the frontend from the backend** layer of the application. Separates UI and business logic.
- **Flexible** to use any **frontend framework** or modern approach to web developments
- Easily **share services** across all channel heads
- Update frontend **independently** of the services
- Innovate and **experiment** with new channels
- Better performance



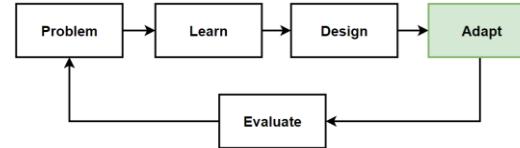
Before Design – What we have in our design toolbox ?

Architectures	Patterns&Principles	Non-FR	FR
• Monolithic Architecture	• KISS • YAGNI	• Availability • High number of Concurrent User	• List products
• Layered Architecture	• Separation of Concerns (SoC)	• Maintainability	• Filter products as per brand and categories
• Clean Architecture		• Flexibility	• Put products into the shopping cart
• Modular Monolithic Architecture	• SOLID • The Dependency Rule	• Testable	• Apply coupon for discounts
• Headless Architecture	• Horizontal Scaling • Load Balancer • Monolithic-First Strategy • Separated UI	• Scalability • Reliability • Re-usability	• Checkout the shopping cart and create an order • List my old orders and order items history

Design: Modular Monolithic Architecture with SPA



Adapt: Modular Monolithic Architecture with SPA



Load Balancer

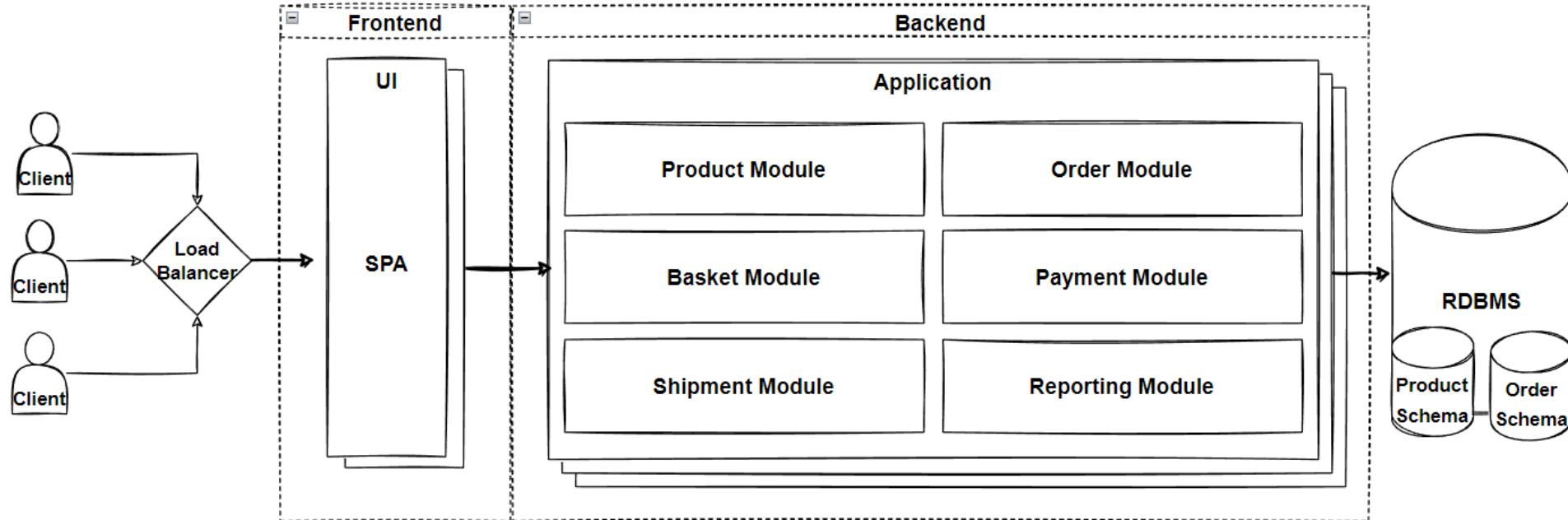
- Apache LB
- NGINX

Frontend SPAs

- Angular
- Vue
- React

Backend Application

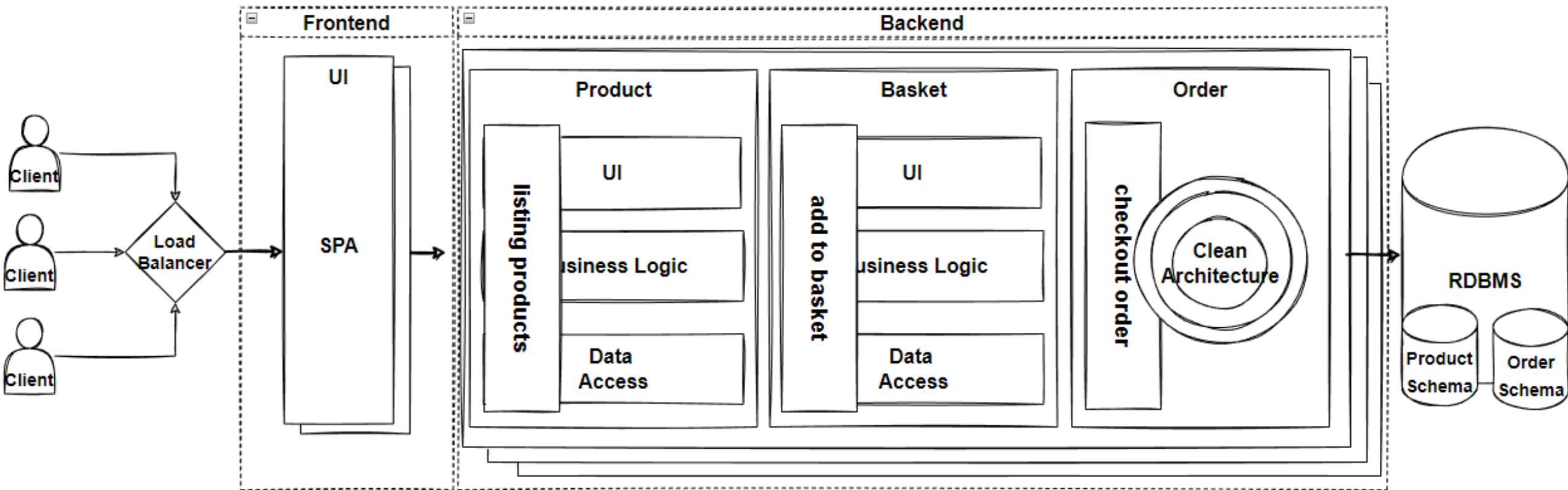
- Single JAR / WAR File
- Tomcat Container



Database

- Oracle
- Postgres
- SQL Server

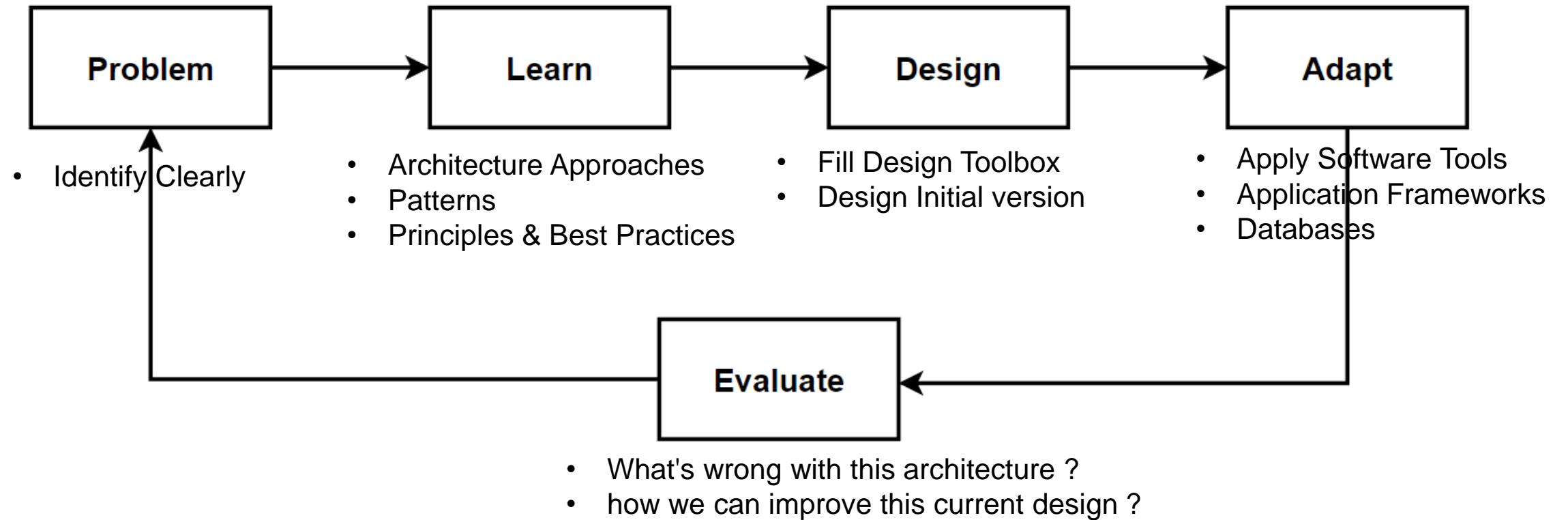
DEMO: Modular Monolithic Architecture with SPA Code Review



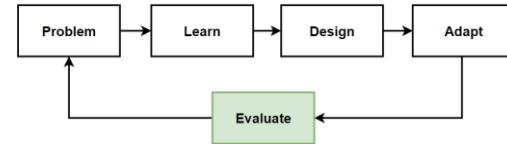
DEMO: Kamil Grzybek – Modular Monolithic with DDD

- <https://github.com/kgrzybek/modular-monolith-with-ddd>
- <https://github1s.com/kgrzybek/modular-monolith-with-ddd>

Way of Learning – The Course Flow

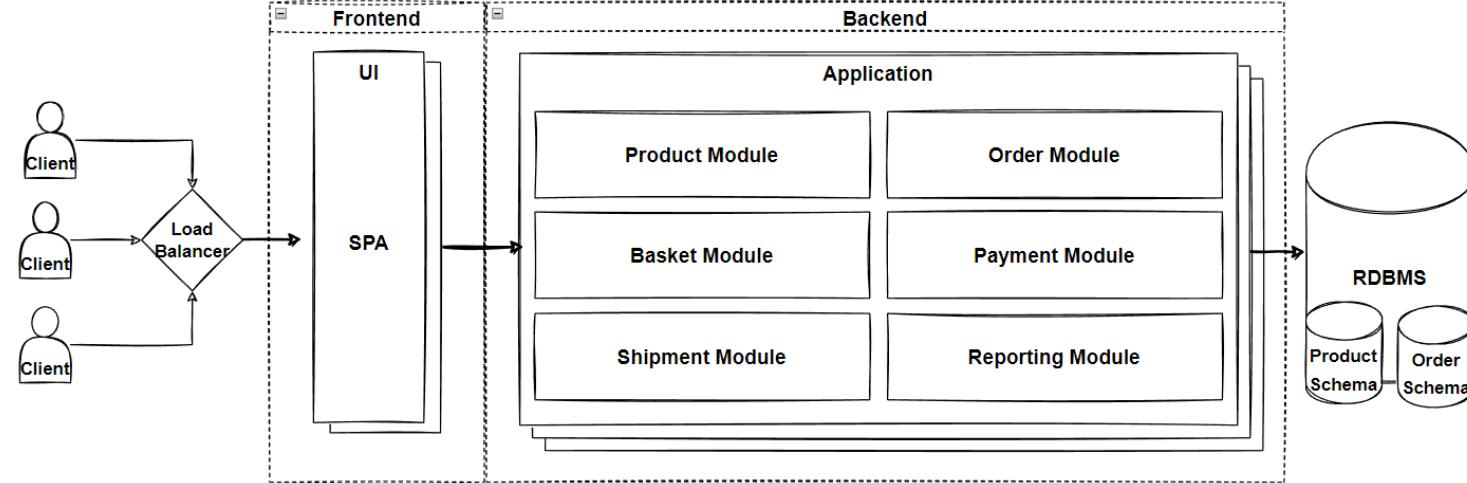


Evaluate: Modular Monolithic Architecture with SPA



Benefits

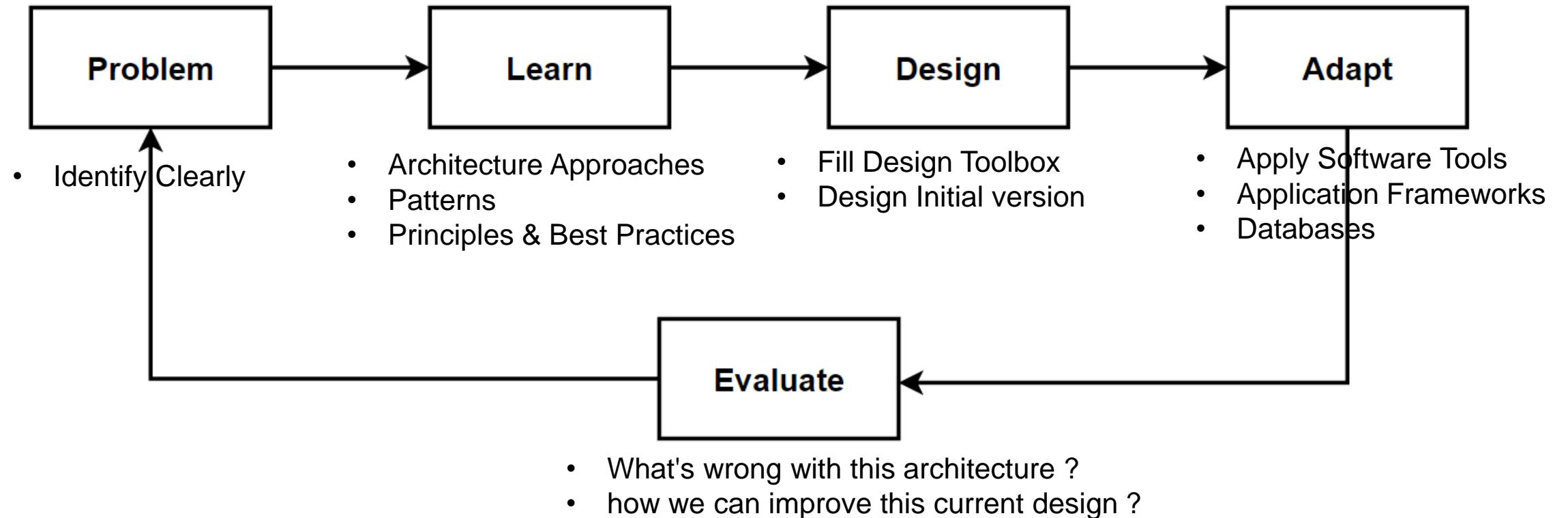
- Easy Development, Debug and Deploy
- Encapsulate Business Logic
- Reusable Codes, Easy to Refactor
- Better-Organized Dependencies and Teams
- Update frontend independently, Flexible UI



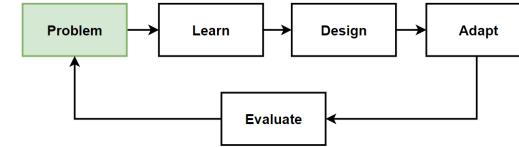
Drawbacks

- Scalability Limits, Database Can't Scale
- We have 1 big relational database that can't scale and become bottleneck for our architecture. Million request got timeout exception.
- It is **still Monolithic** and has **Scalability Issues**
- Can't Scale Modules Independently
- Can't Deploy Modules Independently
- It is **still Monolithic** and has **Deployment Issues**

Way of Learning – The Course Flow



Problem: Scale and Deploy Independently

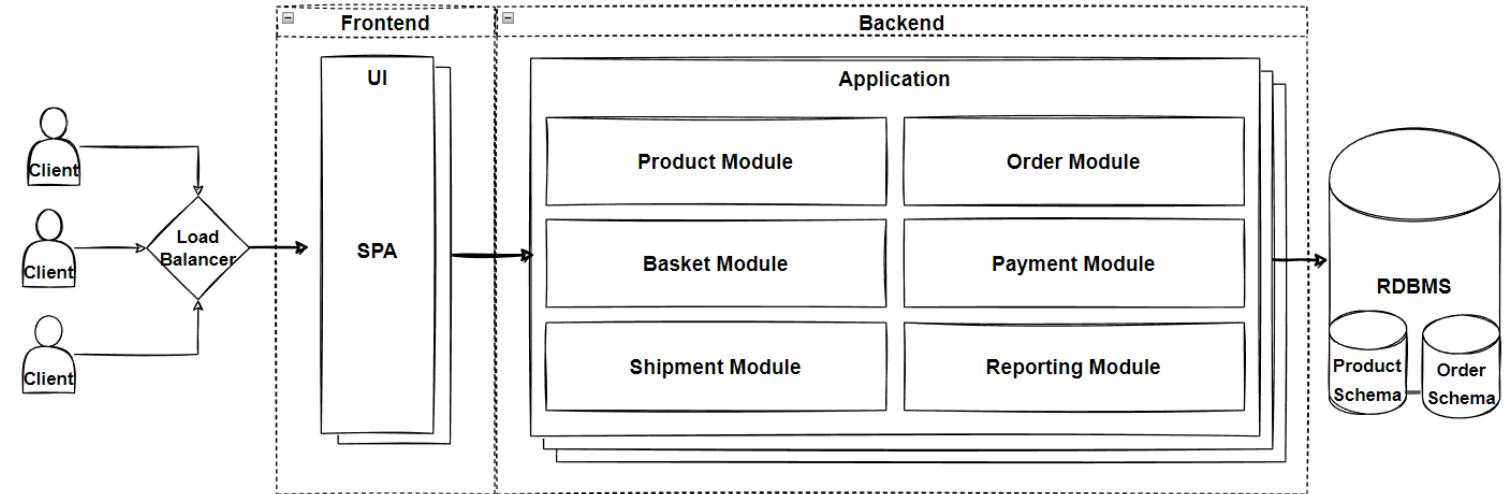


Problems

- Our E-Commerce Business is growing
- Business teams are **separated teams** as per departments; Product, Sale, Payment.
- Teams want to be agile and **add new features immediately** to compete the market
- Innovate and experiment with new features as soon as possible
- **Deploy features immediately**, not waiting for deployment dates
- **Flexible scale** for **market peak times** like blackfriday sales
- Handle and process **millions of request** in an acceptable latency with better performance.
- Required not only technology change but also **organizational change is mandatory**.

Solutions

- Microservices Architecture



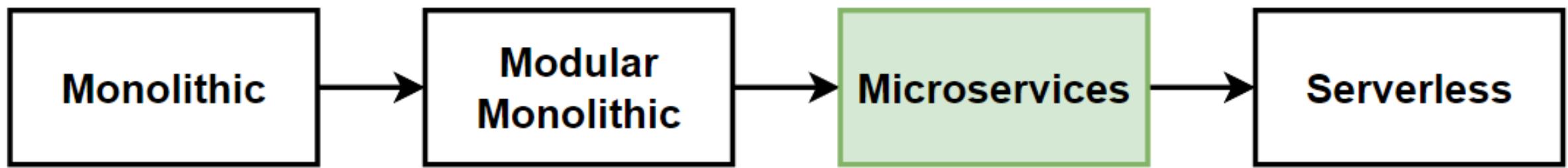
Microservices Architecture

Benefits and Challenges of Microservices Architecture

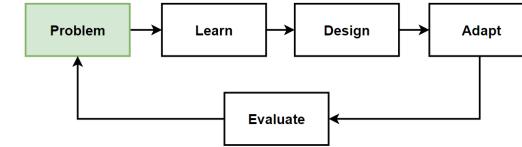
When to use Microservices Architecture

Design our E-Commerce application with Microservices Architecture

Architecture Design Journey



Problem: Scale and Deploy Independently

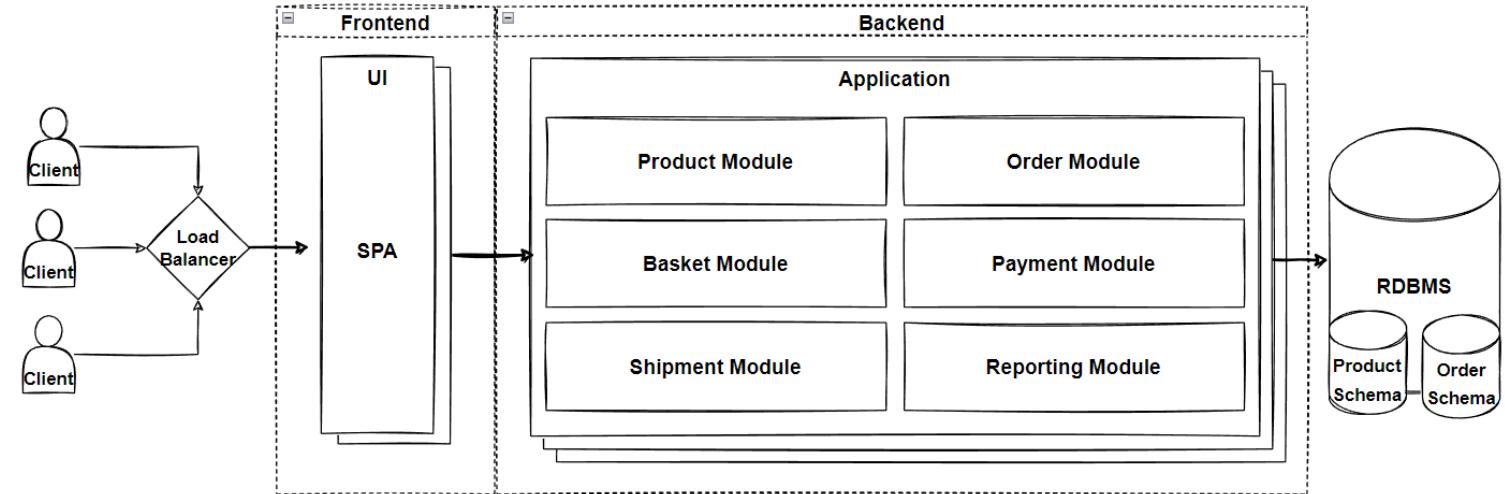


Problems

- Our E-Commerce Business is growing
- Business teams are **separated teams** as per departments; Product, Sale, Payment.
- Teams want to be agile and **add new features immediately** to compete the market
- Innovate and experiment with new features as soon as possible
- **Deploy features immediately**, not waiting for deployment dates
- **Flexible scale** for **market peak times** like blackfriday sales
- Handle and process **millions of request** in an acceptable latency with better performance.
- Required not only technology change but also **organizational change is mandatory**.

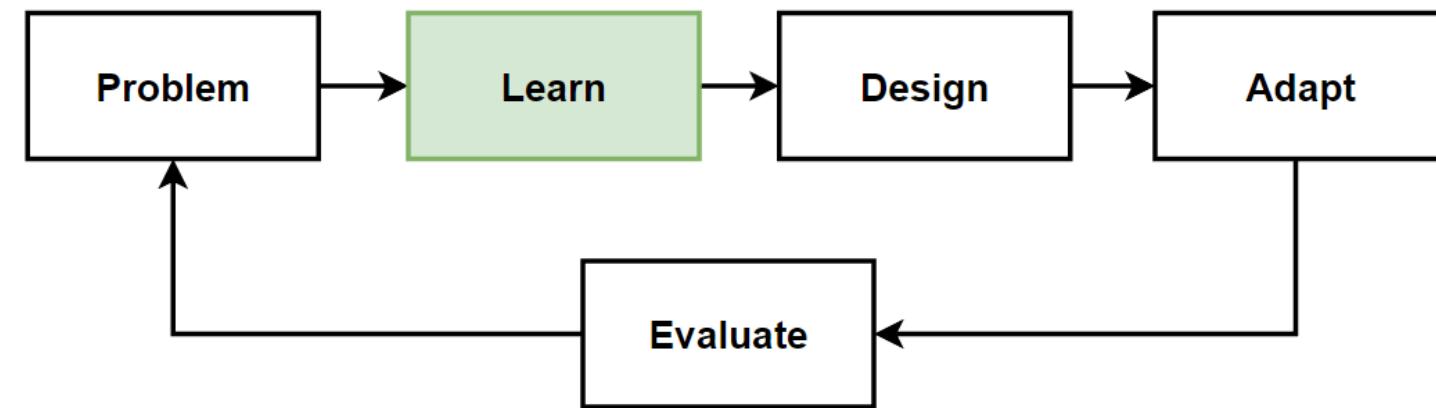
Solutions

- Microservices Architecture



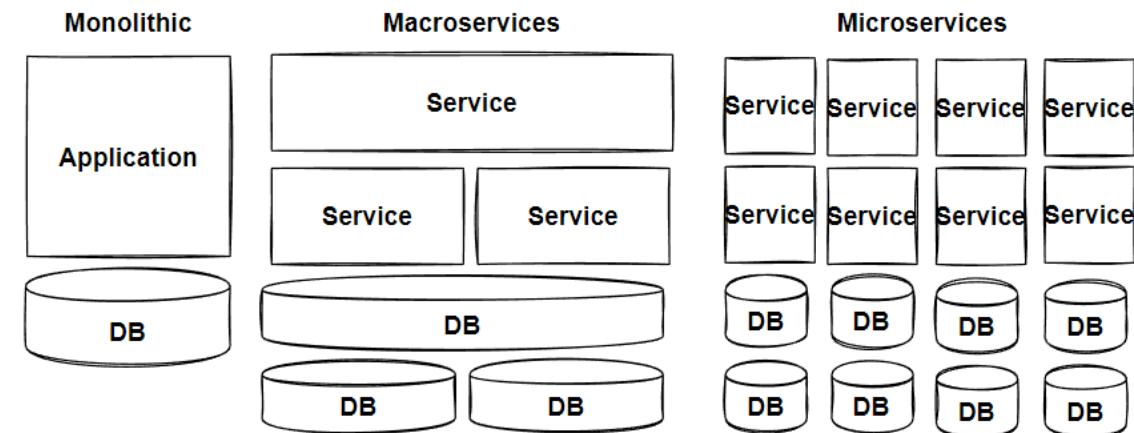
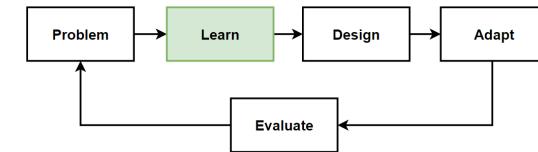
Learn: Microservices Architecture

- Microservices Architecture
- When to use Microservices Architecture
- Benefits of Microservices Architecture
- Challenges of Microservices Architecture
- Microservices Architecture Pros-Cons
- Reference Architectures of Microservices

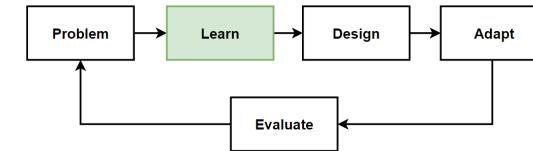


What are Microservices ?

- Microservices are **small, independent, and loosely coupled** services that can work together.
- Each service is a **separate codebase**, which can be managed by a small development team.
- Microservices **communicate** with each other by using **well-defined APIs**.
- Microservices can be **deployed independently and autonomously**.
- Microservices can work with many different technology stacks which is **technology agnostic**.
- Microservices has its **own database** that is not shared with other services.



What is Microservices Architecture ?



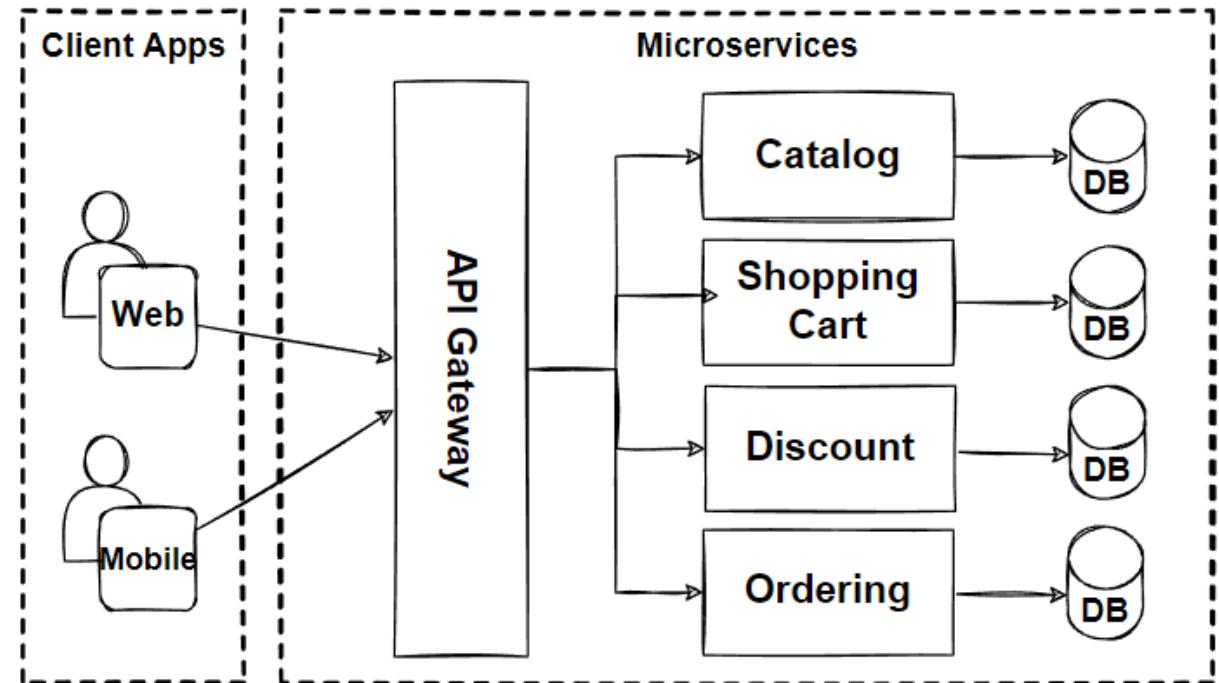
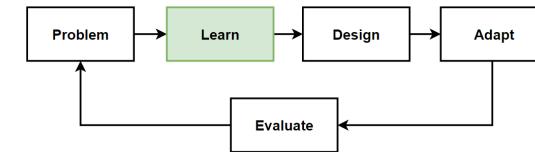
- From Martin Fowlers Microservices article:

*The microservice architectural style is an approach to developing a single application as a **suite of small services**, each running in **its own process** and **communicating** with lightweight mechanisms, often an **HTTP or gRPC API**.*

- Microservices are **built around business capabilities** and **independently deployable** by fully automated deployment process.
- Microservices architecture **decomposes** an application into **small independent services** that communicate over **well-defined APIs**. Services are owned by **small, self-contained teams**.
- Microservices architecture is a **cloud native architectural approach** in which services composed of many **loosely coupled** and **independently deployable** smaller components.
- Microservices have their **own technology stack**, communicate to each other over a combination of **REST APIs**, are organized by **business capability**, with the **bounded contexts**.
- Following **Single Responsibility Principle** that referring separating responsibilities as per services.

Microservices Characteristics

- From Martin Fowlers Microservices article;
- Componentization via Services
- Organized around Business Capabilities
- Products not Projects
- Smart endpoints and dumb pipes
- Decentralized Governance
- Decentralized Data Management
- Infrastructure Automation
- Design for failure



Benefits of Microservices Architecture

- **Agility, Innovation and Time-to-market**

Microservices architectures make applications easier to scale and faster to develop, enabling innovation and accelerating time-to-market for new features.

- **Flexible Scalability**

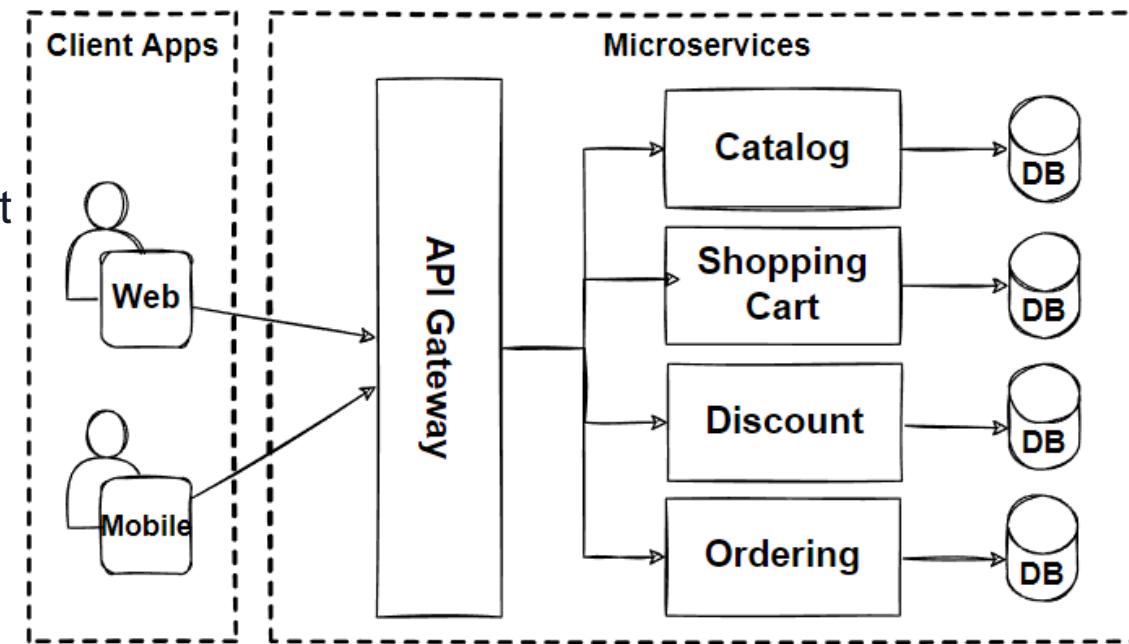
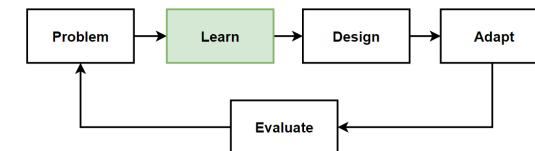
Microservices can be scaled independently, so you scale out sub-services that require less resources, without scaling out the entire application.

- **Small, focused teams**

Microservices should be small enough that a single feature team can build, test, and deploy it.

- **Small and separated code base**

Microservices are not sharing code or data stores with other services, it minimizes dependencies, and that makes easier to adding new features.



Benefits of Microservices Architecture -2

- **Easy Deployment**

Microservices enable continuous integration and continuous delivery, making it easy to try out new ideas and to roll back if something doesn't work.

- **Technology agnostic, Right tool for the job**

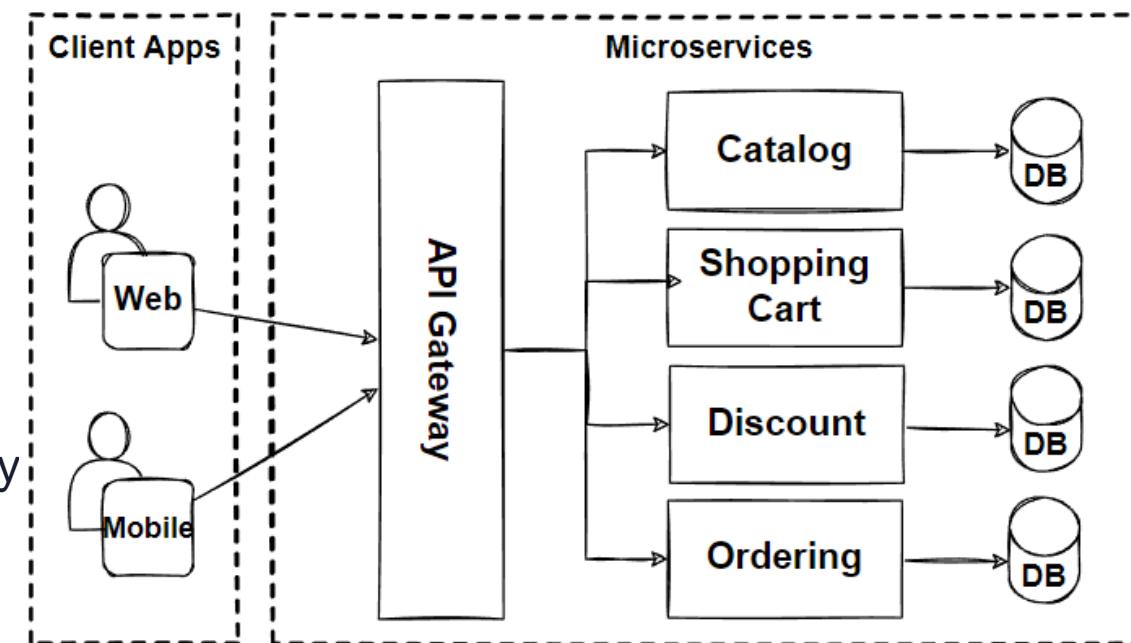
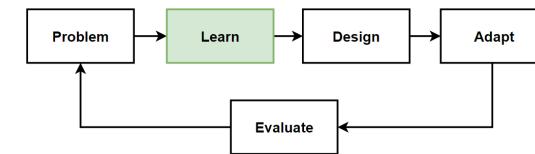
Small teams can pick the technology that best fits their microservice and using a mix of technology stacks on their services.

- **Resilience and Fault isolation**

Microservices are fault tolerated and handle faults correctly for example by implementing retry and circuit breaking patterns.

- **Data isolation**

Databases are separated with each other according to microservices design. Easier to perform schema updates, because only a single database is affected.



Challenges of Microservices Architecture

- **Complexity**

Each service is simpler, but the entire system is more complex. Deployments and Communications can be complicated for hundreds of microservices.

- **Network problems and latency**

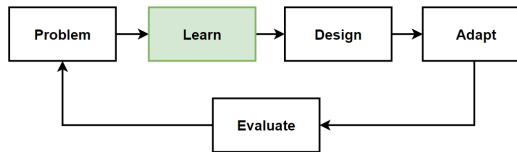
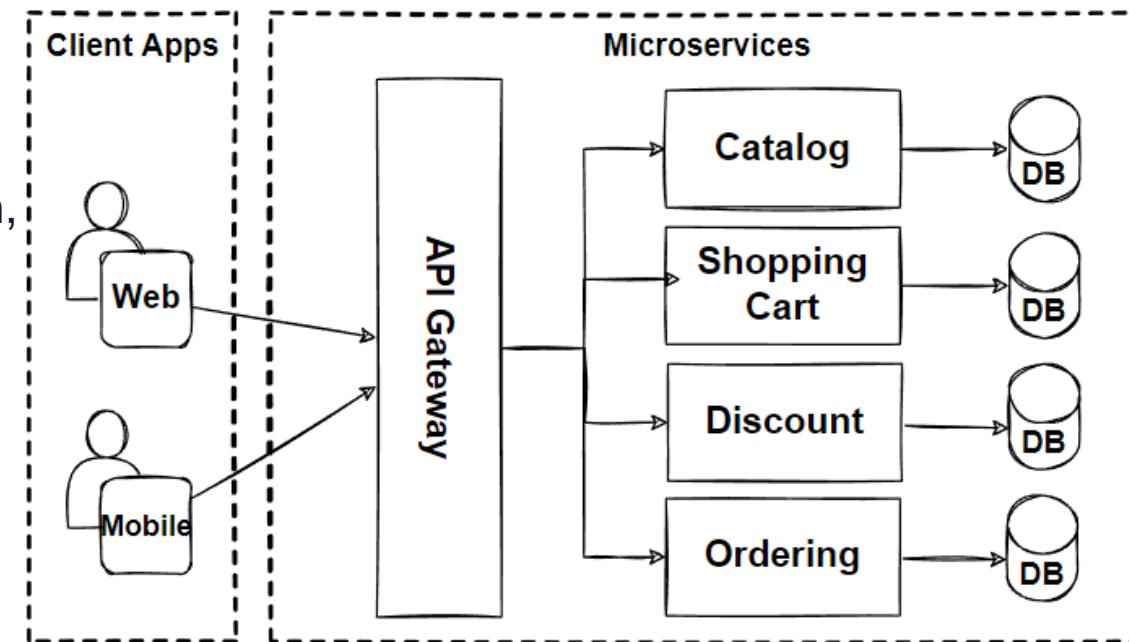
Microservice communicate with inter-service communication, we should manage network problems. Chain of services increase latency problems and become chatty API calls.

- **Development and testing**

Hard to develop and testing these E2E processes in microservices architectures if we compare to monolithic ones.

- **Data integrity**

Microservice has its own data persistence. Data consistency can be a challenge. Follow eventual consistency where possible.



Challenges of Microservices Architecture -2

- **Deployment**

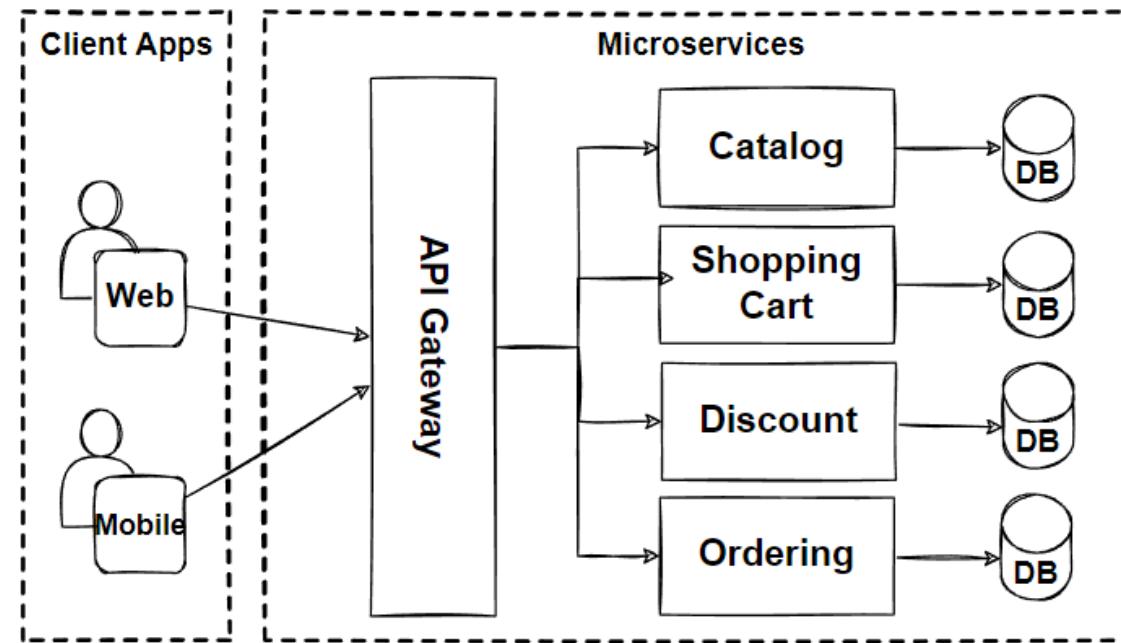
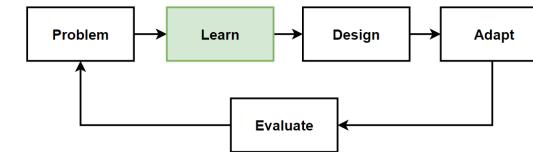
Deployments are challenging. Require to invest in quite a lot of devops automation processes and tools. The complexity of microservices becomes overwhelming for human deployment.

- **Logging & Monitoring**

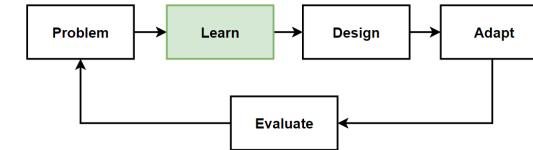
Distributed systems are required to centralized logs to bring everything together. Centralized view of the system to monitor sources of problems.

- **Debugging**

Debugging through local IDE isn't an option anymore. It won't work across dozens or hundreds of services.



When to Use Microservices Architecture



- Make Sure You Have a “Really Good Reason” for Implementing Microservices**

Check if your application can do without microservices. When your application requires agility to time-to-market with zero-down time deployments and updated independently that needs more flexibility.

- Iterate With Small Changes and Keep the Single-Process Monolith as Your “Default”**

Sam Newman and Martin Fowler offers Monolithic-First approach. Single-process monolithic application comes with simple deployment topology. Iterate and refactor with turning a single module from the monolith into a microservices one by one.

- Required to Independently Deploy New Functionality with Zero Downtime**

When an organization needs to make a change to functionality and deploy that functionality without affecting rest of the system.

- Required to Independently Scale a Portion of Application**

Microservice has its own data persistence. Data consistency can be a challenge. Follow eventual consistency where possible.

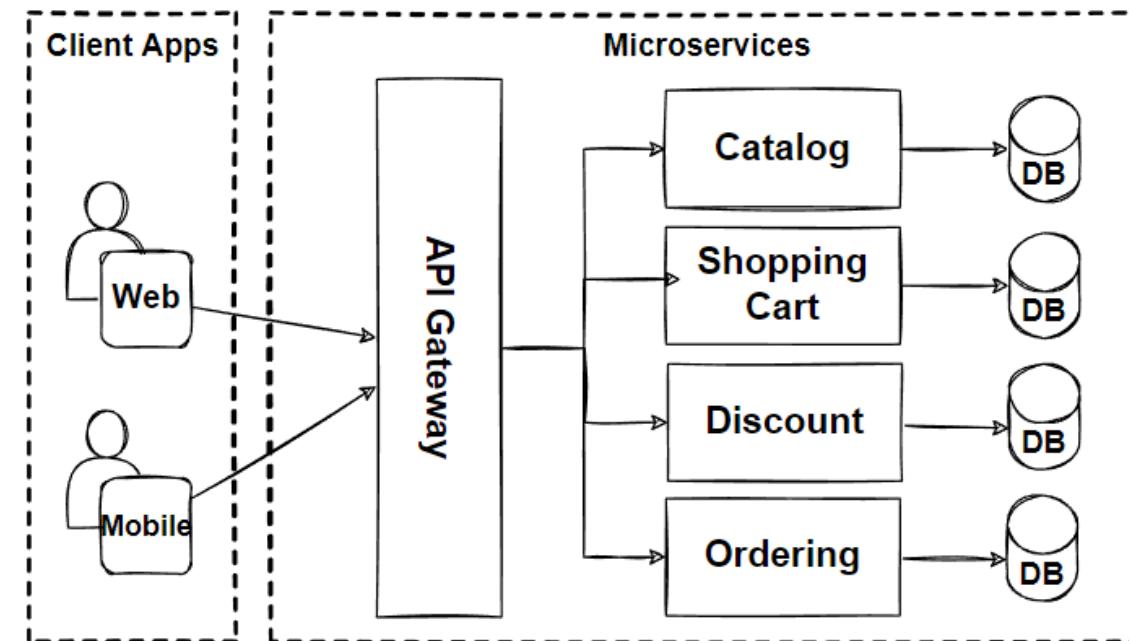
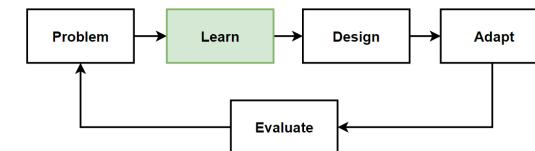
When to Use Microservices Architecture -2

- **Data Partitioning with different Database Technologies**

Microservices are extremely useful when an organization needs to store and scale data with different use cases. Teams can choose the appropriate technology for the services they will develop over time.

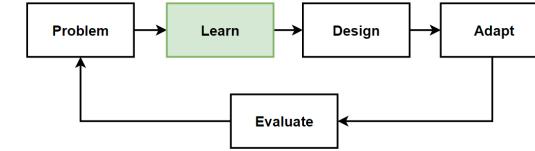
- **Autonomous Teams with Organizational Upgrade**

Microservices will help to evolve and upgrade your teams and organizations. Organizations need to distribute responsibility into teams, where each team makes decisions and develops software autonomously.



When **Not** to Use Microservices

Anti-Patterns of Microservices



- **Don't do Distributed Monolith**

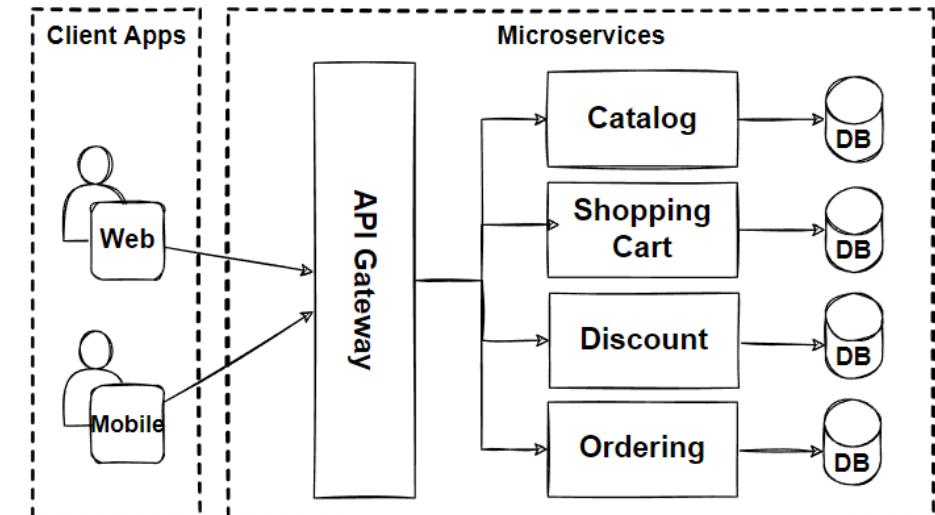
Make sure that you decompose your services properly and respecting the decoupling rule like applying bounded context and business capabilities principles.

- Distributed Monolith is the worst case because you increase complexity of your architecture without getting any benefit of microservices.

- **Don't do microservices without DevOps or cloud services**

Microservices are embrace the distributed cloud-native approaches. And you can only maximize benefits of microservices with following these cloud-native principles.

- CI/CD pipeline with devops automations
- Proper deployment and monitoring tools
- Managed cloud services to support your infrastructure
- Key enabling technologies and tools like Containers, Docker, and Kubernetes
- Following asnyc communications using Messaging and event streaming services



When **Not** to Use Microservices

- **Limited Team sizes, Small Teams**

If you don't have a team size that cannot handle the microservice workloads, This will only result in the delay of delivery.

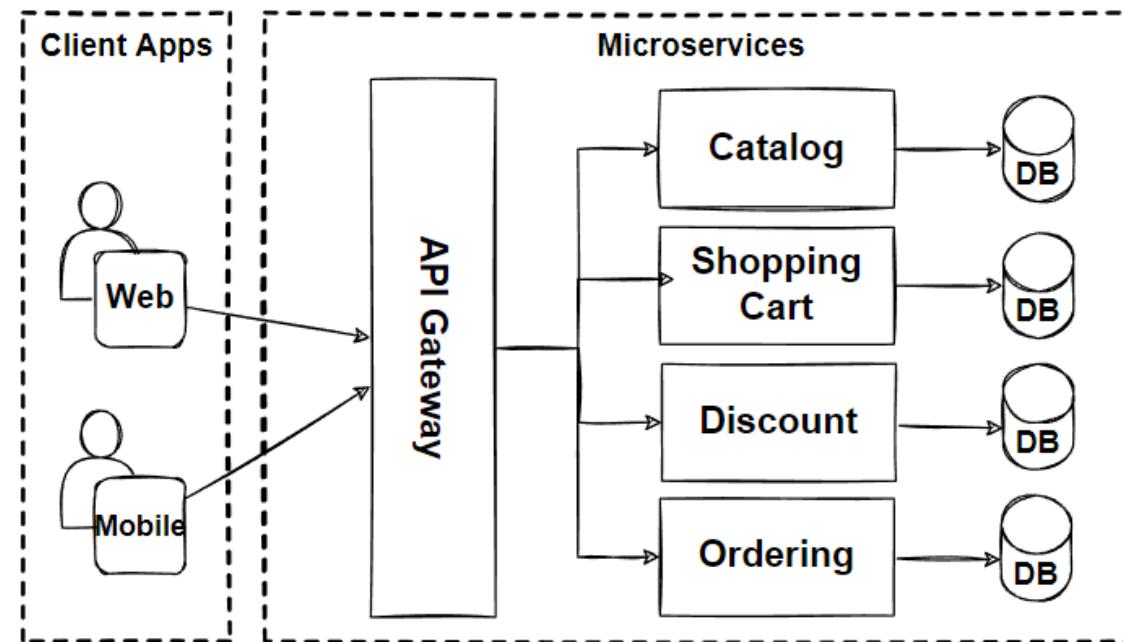
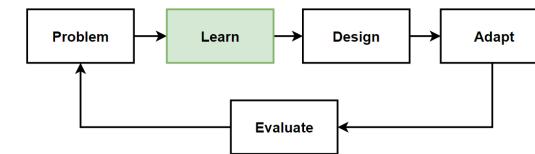
- For a small team, a microservice architecture can be hard to justify, because team is required just to handle the deployment and management of the microservices themselves.

- **Brand new products or startups**

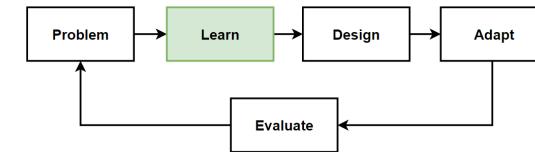
If you are working on a new startup or brand new product which require significant change when developing and iterating your product, then you should not start with microservices.

- Microservices are so expensive when you re-design your business domains. Even if you do become successful enough to require a highly scalable architecture.

- **The Shared Database anti-pattern**



Monolithic vs Microservices Architecture Comparison



- **Application Architecture**

Monolith has a simple straightforward structure of one undivided unit. Microservices have a complex structure that consists of various heterogeneous services and databases.

- **Scalability**

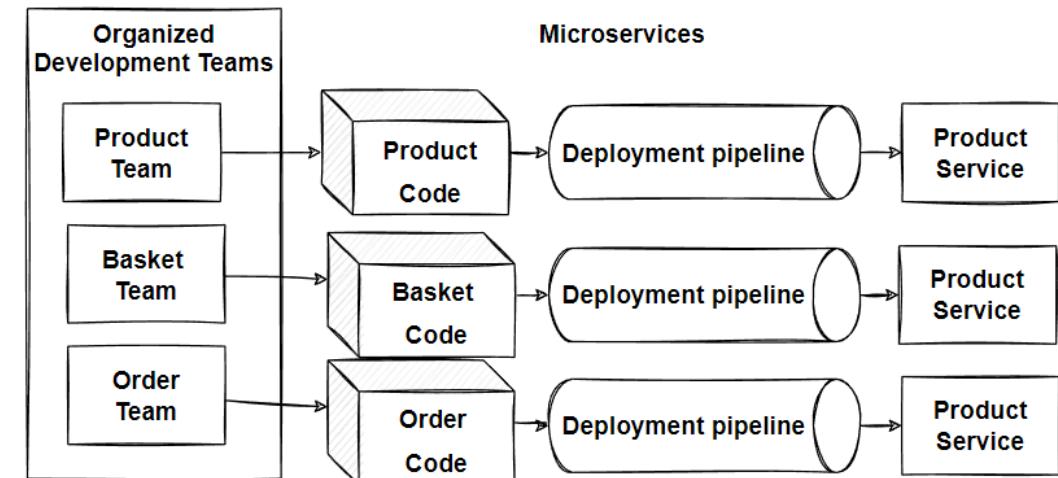
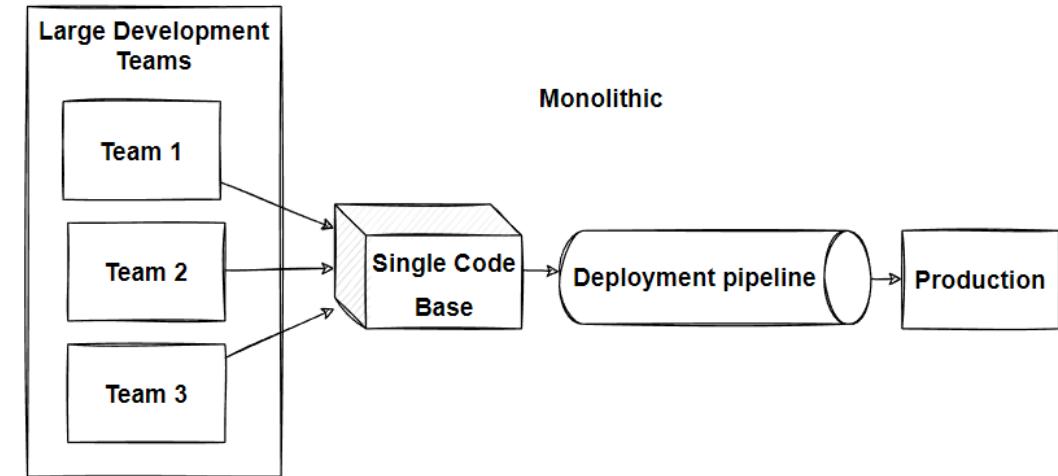
Monolithic application is scaled as a whole single unit, but microservices can be scaled unevenly. encourages companies to migrate their applications to microservices.

- **Deployment**

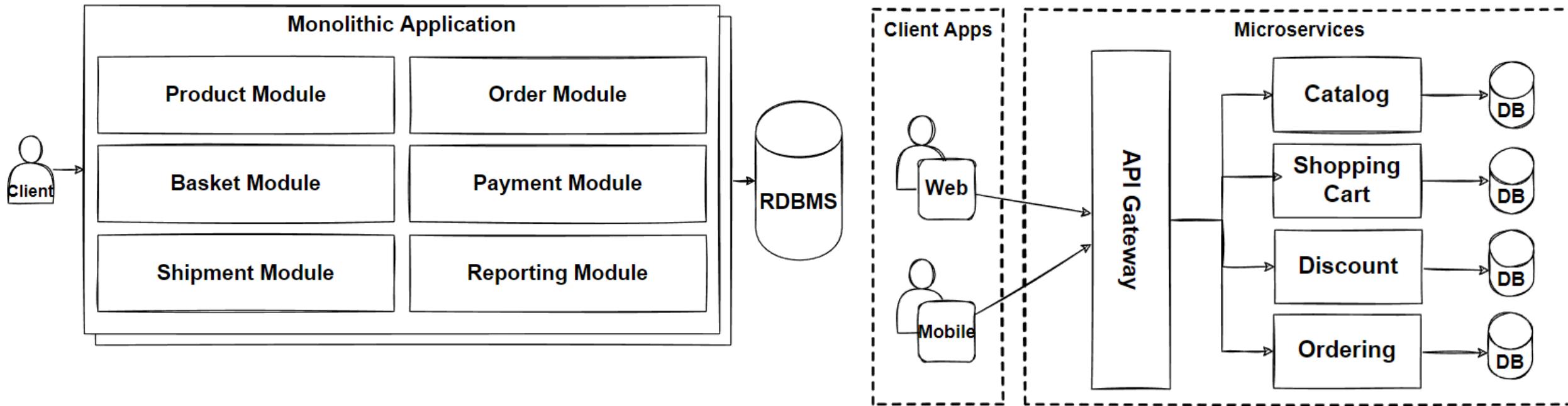
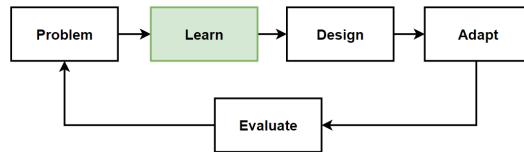
Monolithic application provides fast and easy deployment of the whole system. Microservices provides zero-downtime deployment and CI/CD automation.

- **Development team**

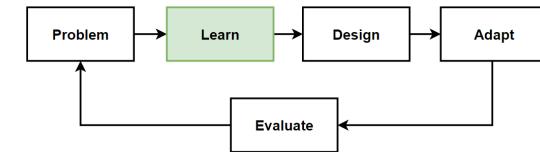
If your team doesn't have experience with microservices and container systems, building a microservices-based application will be difficult.



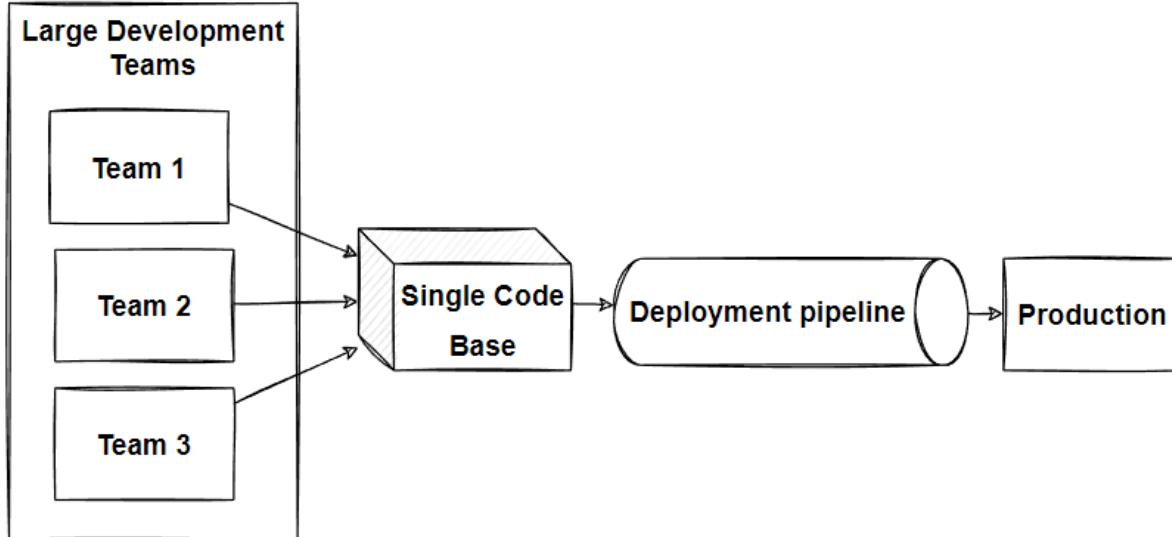
Architecture Comparison



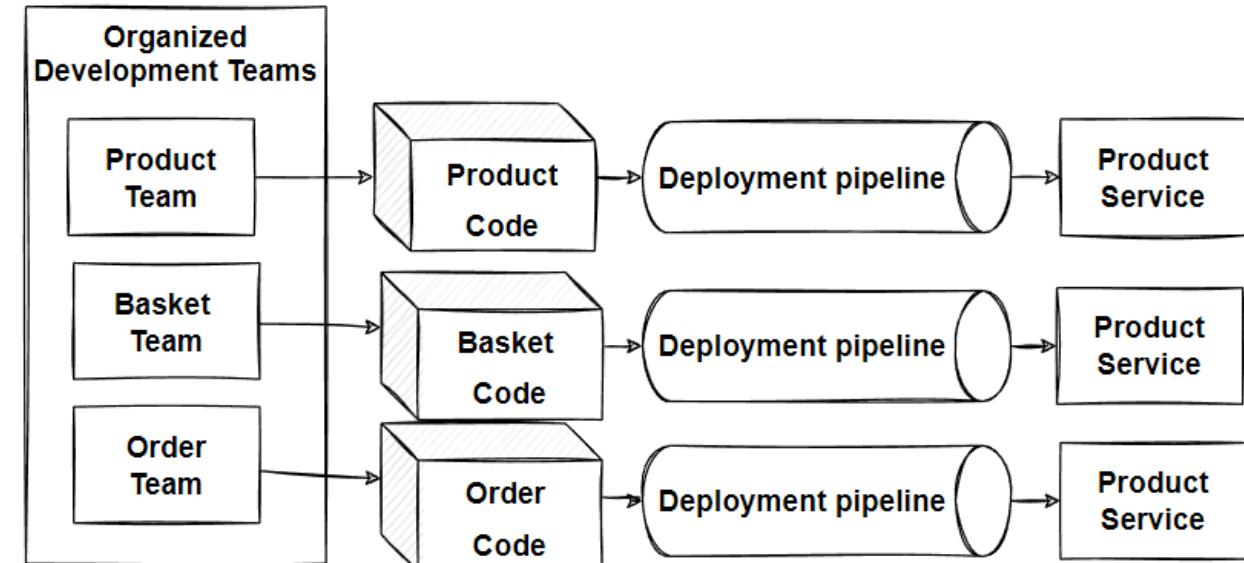
Deployment Comparison



Monolithic

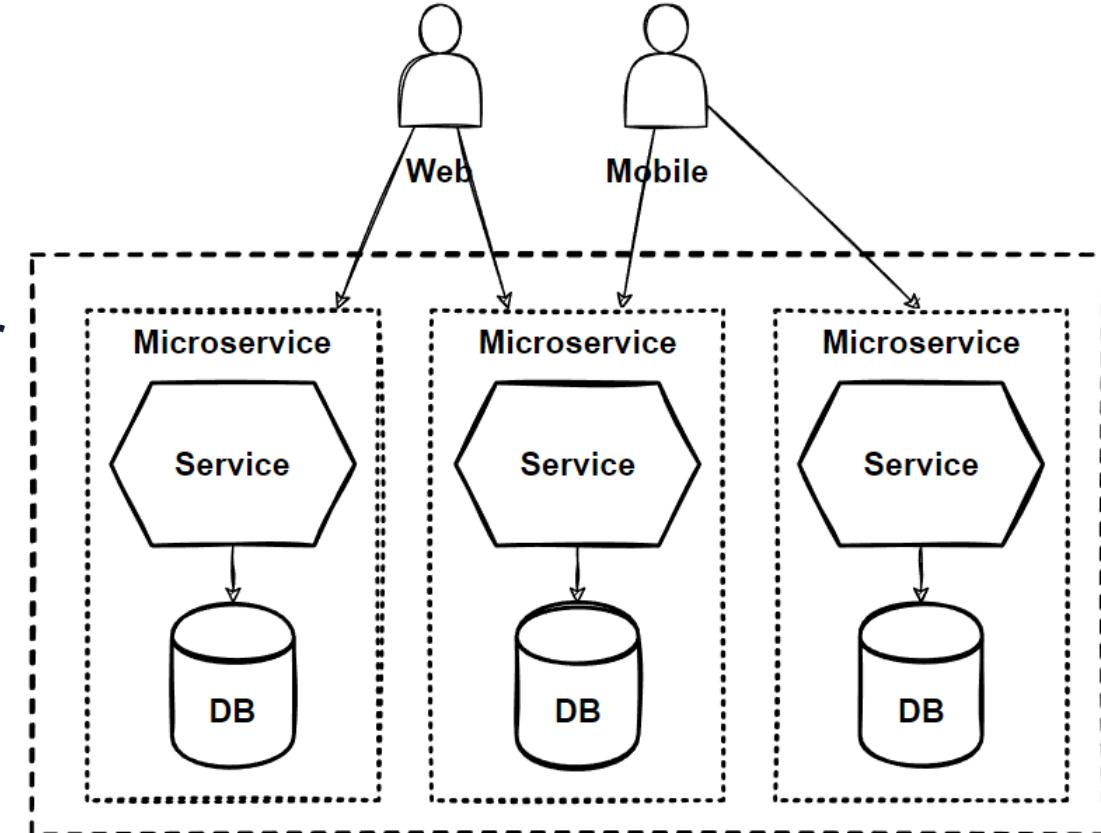


Microservices



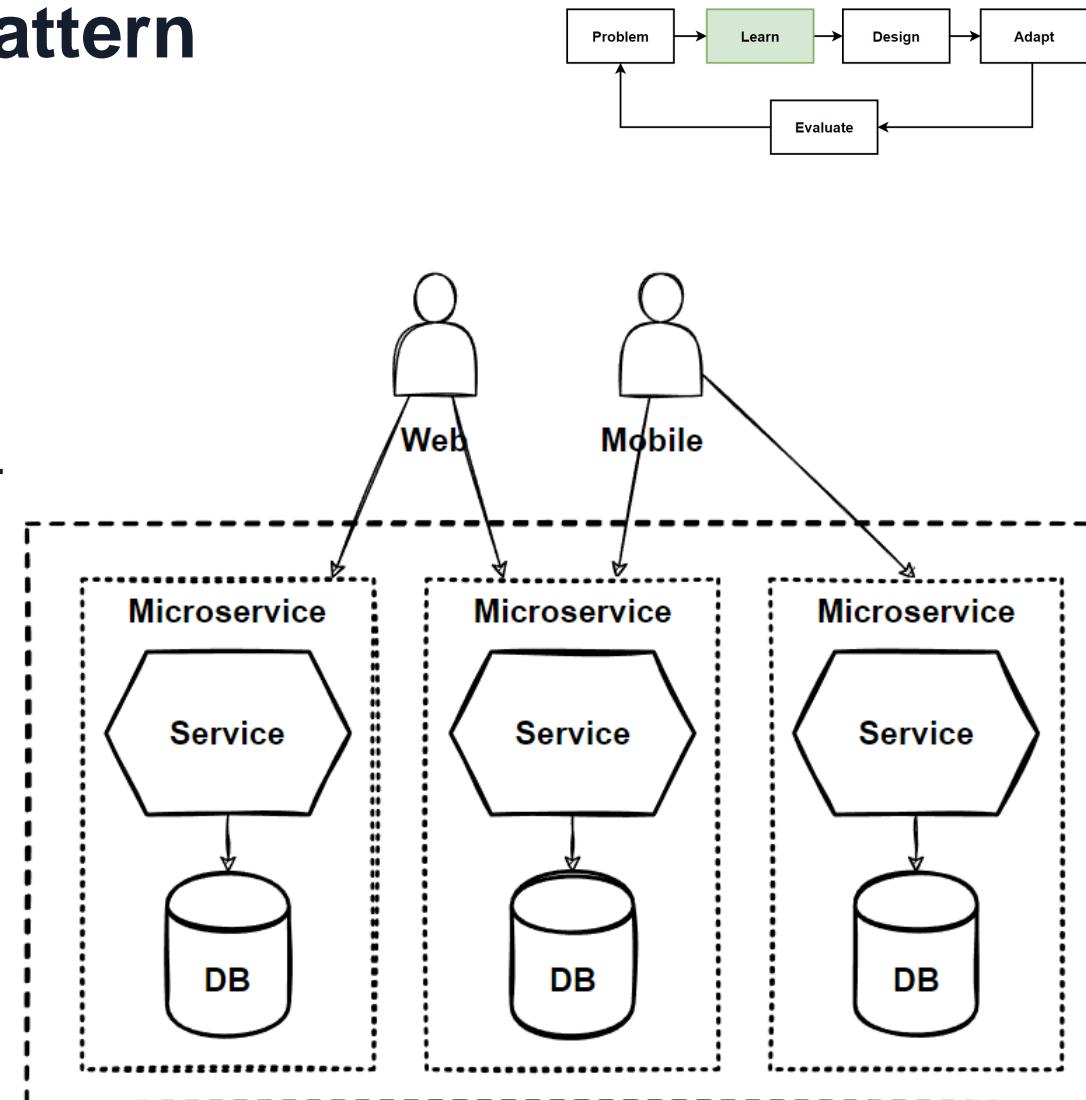
The Database-per-Service Pattern

- Core characteristic of the microservices architecture is the **loose coupling of services**. every service should have its own **databases**, it can be **polyglot persistence** among to microservices.
- E-commerce application. We will have **Product - Ordering** and **SC** microservices that **each services data in their own databases**. Any changes to one database **don't impact other microservices**.
- The **service's database can't be accessed directly** by other microservices. Each service's persistent data can only be accessed via **Rest APIs**.



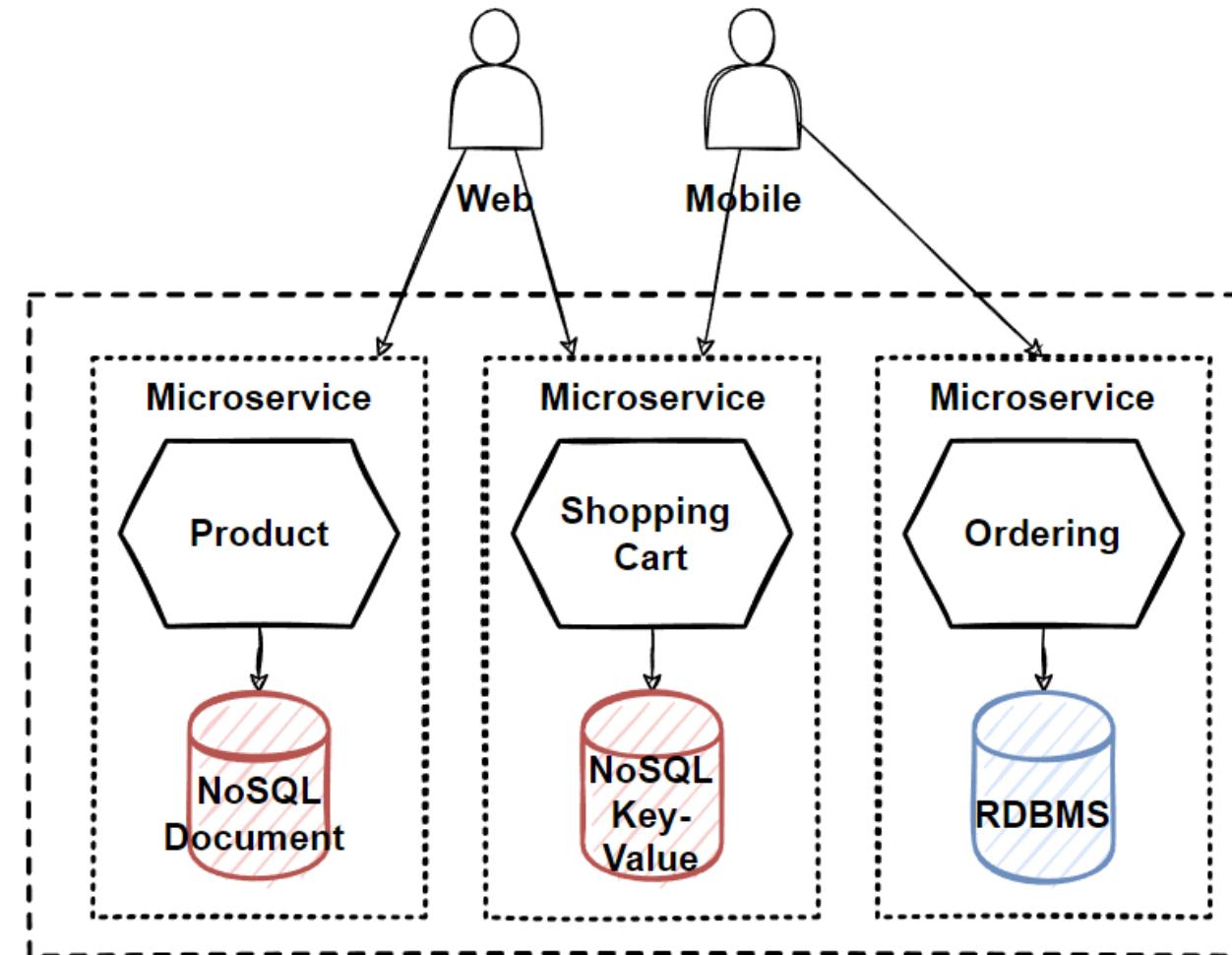
Benefits of the Database-per-Service Pattern with Polygot Persistence

- **Data schema changes** made easy without impacting other microservices.
- Each **database can scale independently**.
- Microservices domain data is **encapsulated** within the service.
- If one of the database server is down, this will **not affect to other services**.
- **Polyglot data persistence** gives ability to select the **best optimized storage** needs per microservices.

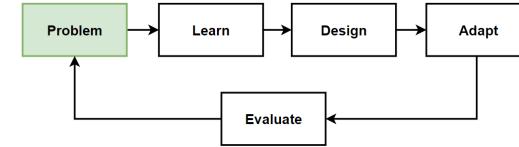


E-Commerce with Database-per-Service Pattern and Polygot Persistence

- Product service using **NoSQL document database** for storing catalog related data.
- Shopping cart service using a **distributed cache** that supports its simple, **key-value data store**.
- Ordering service using a **relational database** to handle the rich relational structure.
- **NoSQL databases** able to massive scale and high availability, and also schemaless structure give flexibility.



Problem: Scale and Deploy Independently

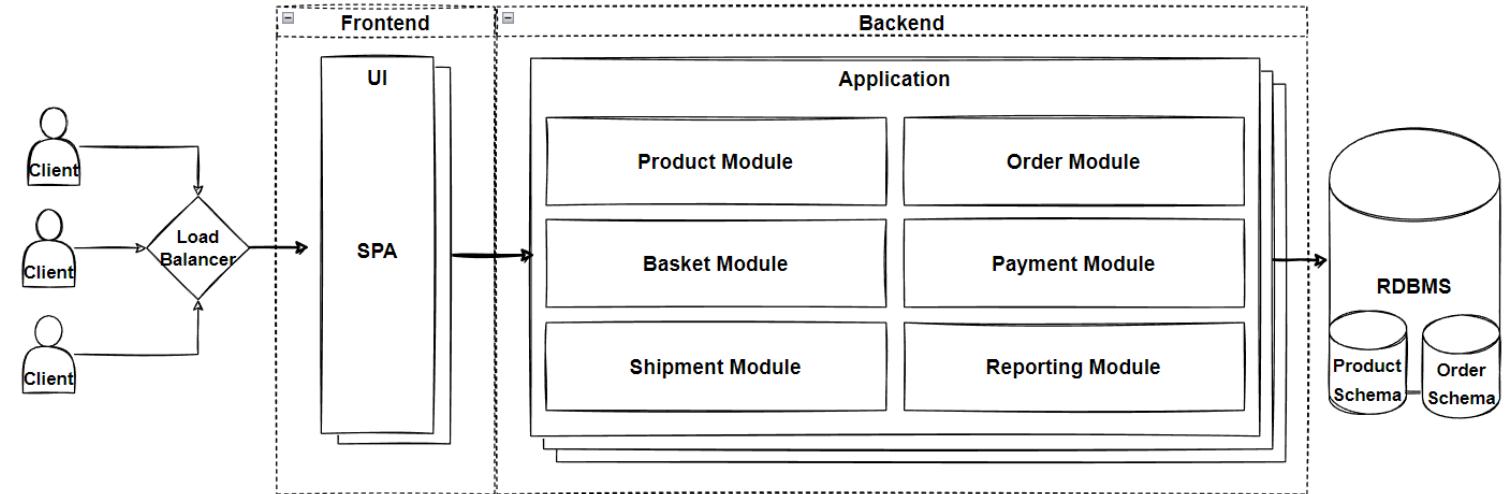


Problems

- Our E-Commerce Business is growing
- Business teams are **separated teams** as per departments; Product, Sale, Payment.
- Teams want to be agile and **add new features immediately** to compete the market
- Innovate and experiment with new features as soon as possible
- **Deploy features immediately**, not waiting for deployment dates
- **Flexible scale** for **market peak times** like blackfriday sales
- Handle and process **millions of request** in an acceptable latency with better performance.
- Required not only technology change but also **organizational change is mandatory**.

Solutions

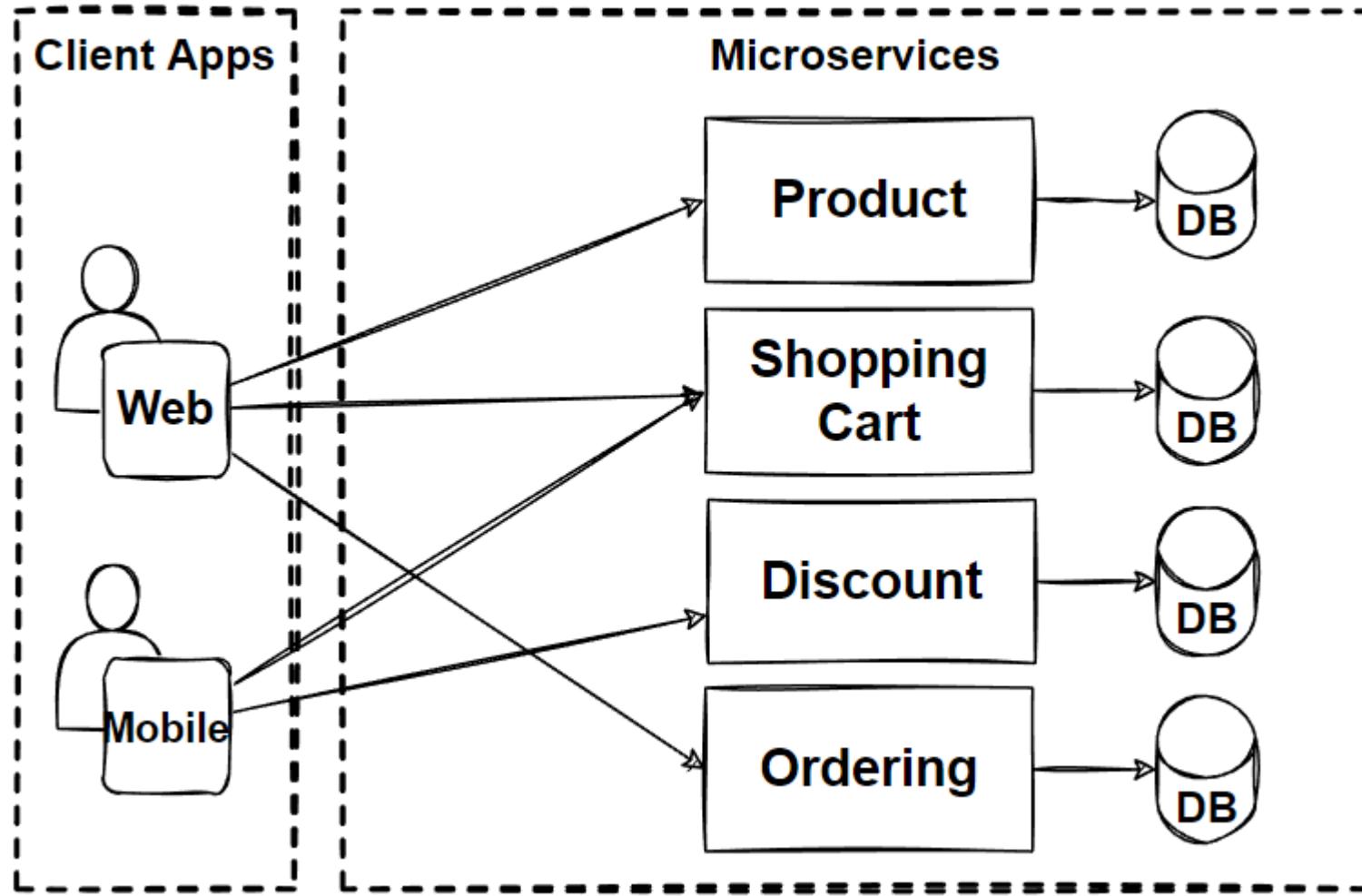
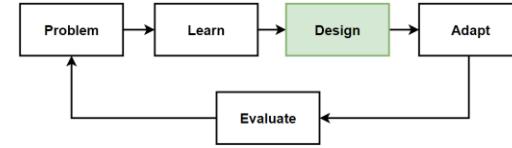
- Microservices Architecture



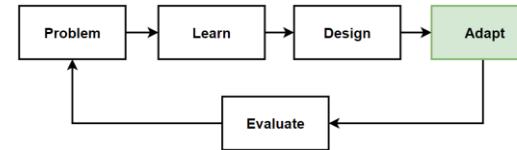
Before Design – What we have in our design toolbox ?

Architectures	Patterns&Principles	Non-FR	FR
<ul style="list-style-type: none">• Microservices Architecture	<ul style="list-style-type: none">• The Database-per-Service Pattern• Polygot Persistence	<ul style="list-style-type: none">• High Scalability• High Availability• Millions of Concurrent User• Independent Deployable• Technology agnostic• Data isolation• Resilience and Fault isolation	<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history

Design: Microservices Architecture



Adapt: Microservices Architecture



Frontend SPAs

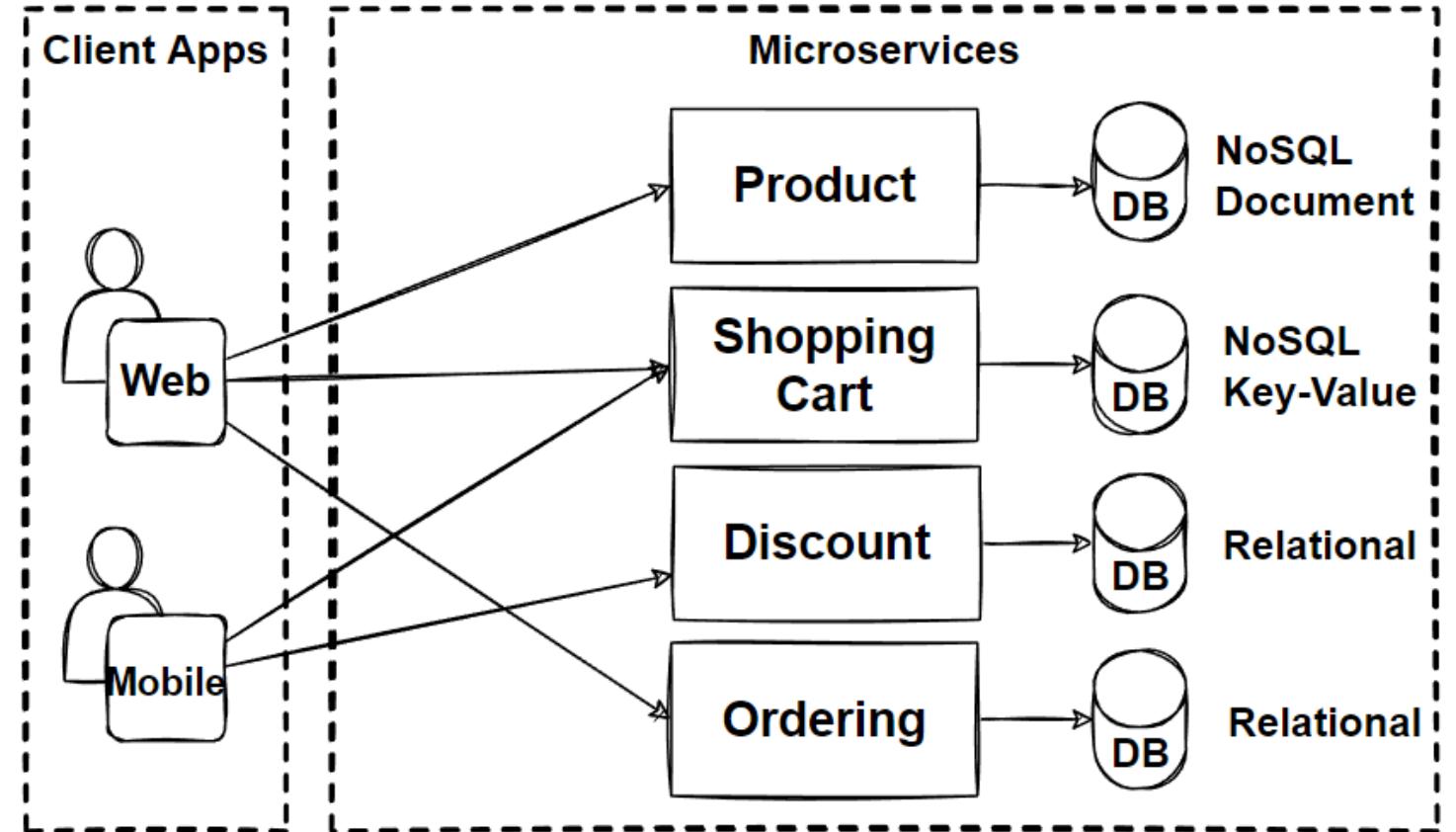
- Angular
- Vue
- React

Backend Microservices

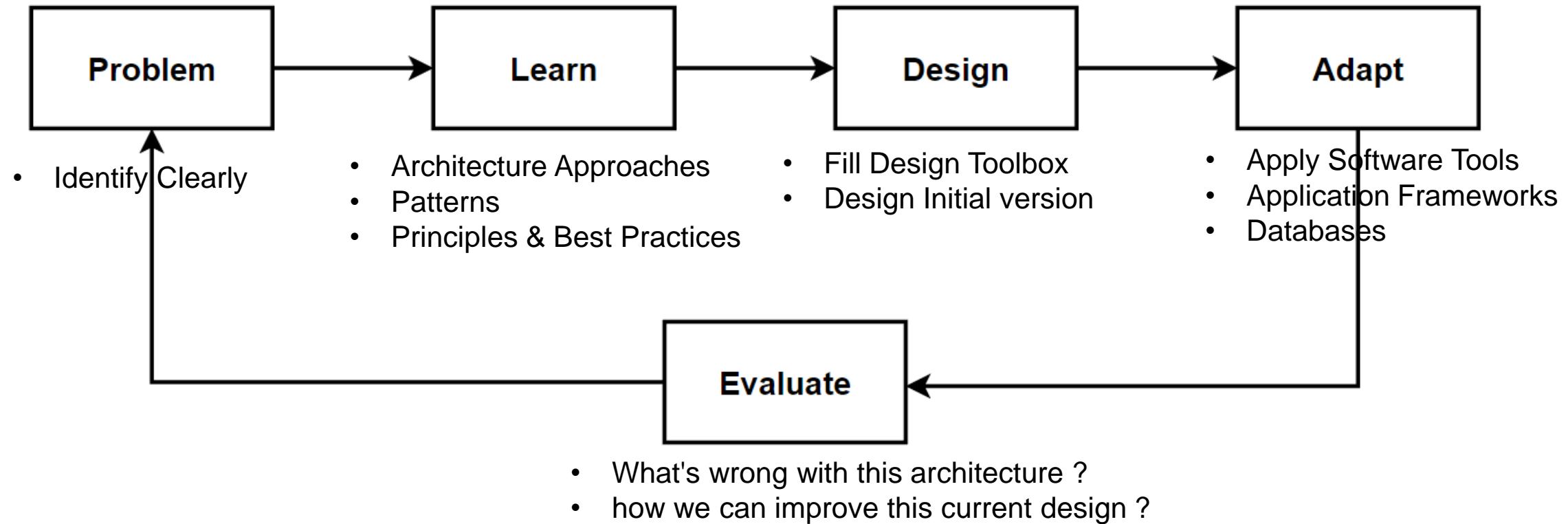
- Java – Spring Boot
- .Net – Asp.net
- JS – NodeJS
- Python – Django, Flask

Database

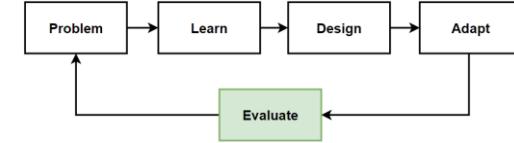
- MongoDB – NoSQL Document DB
- Redis – NoSQL Key-Value Pair
- Postgres – Relational
- SQL Server – Relational



Way of Learning – The Course Flow

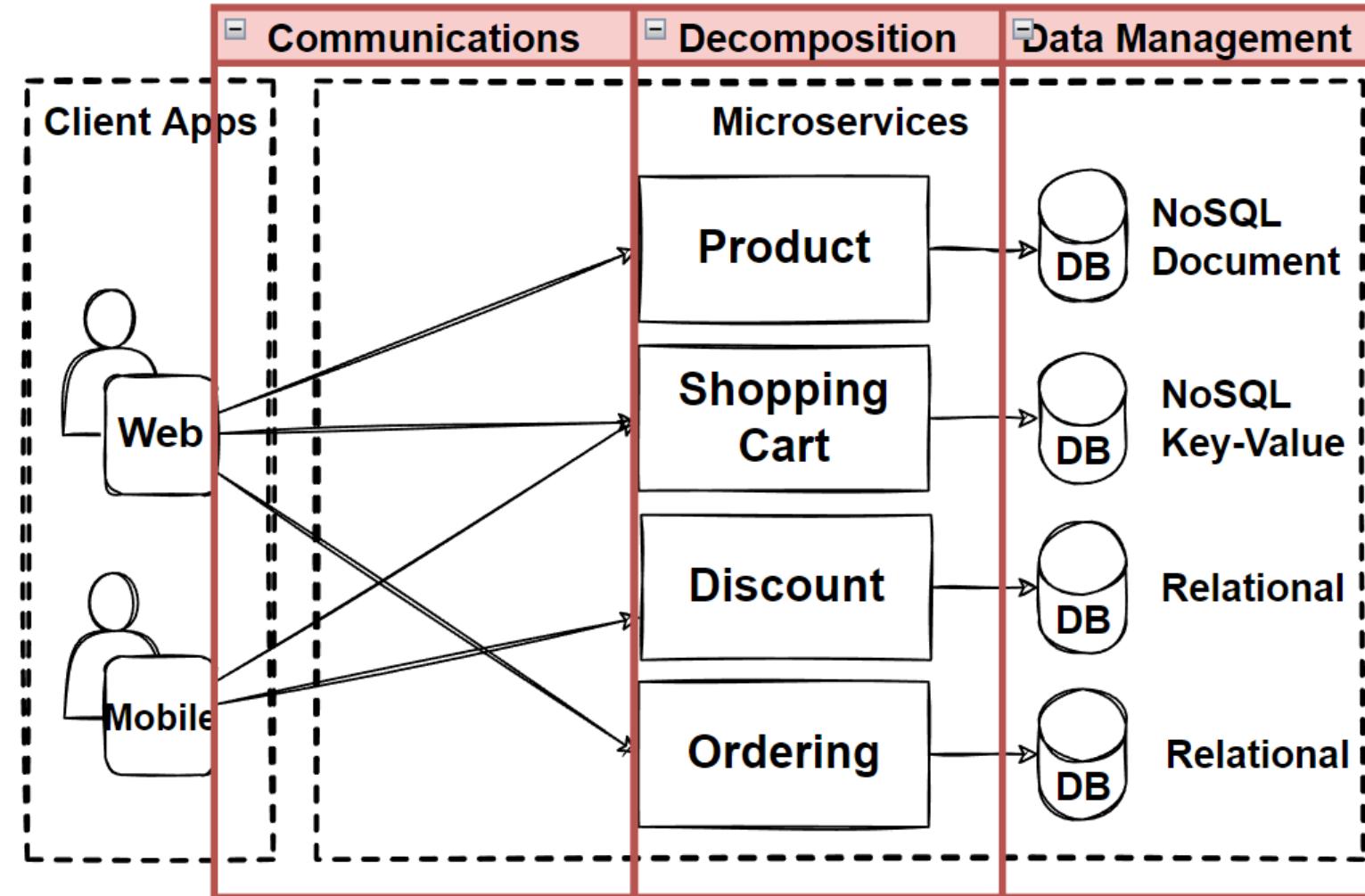


Evaluate: Microservices Architecture

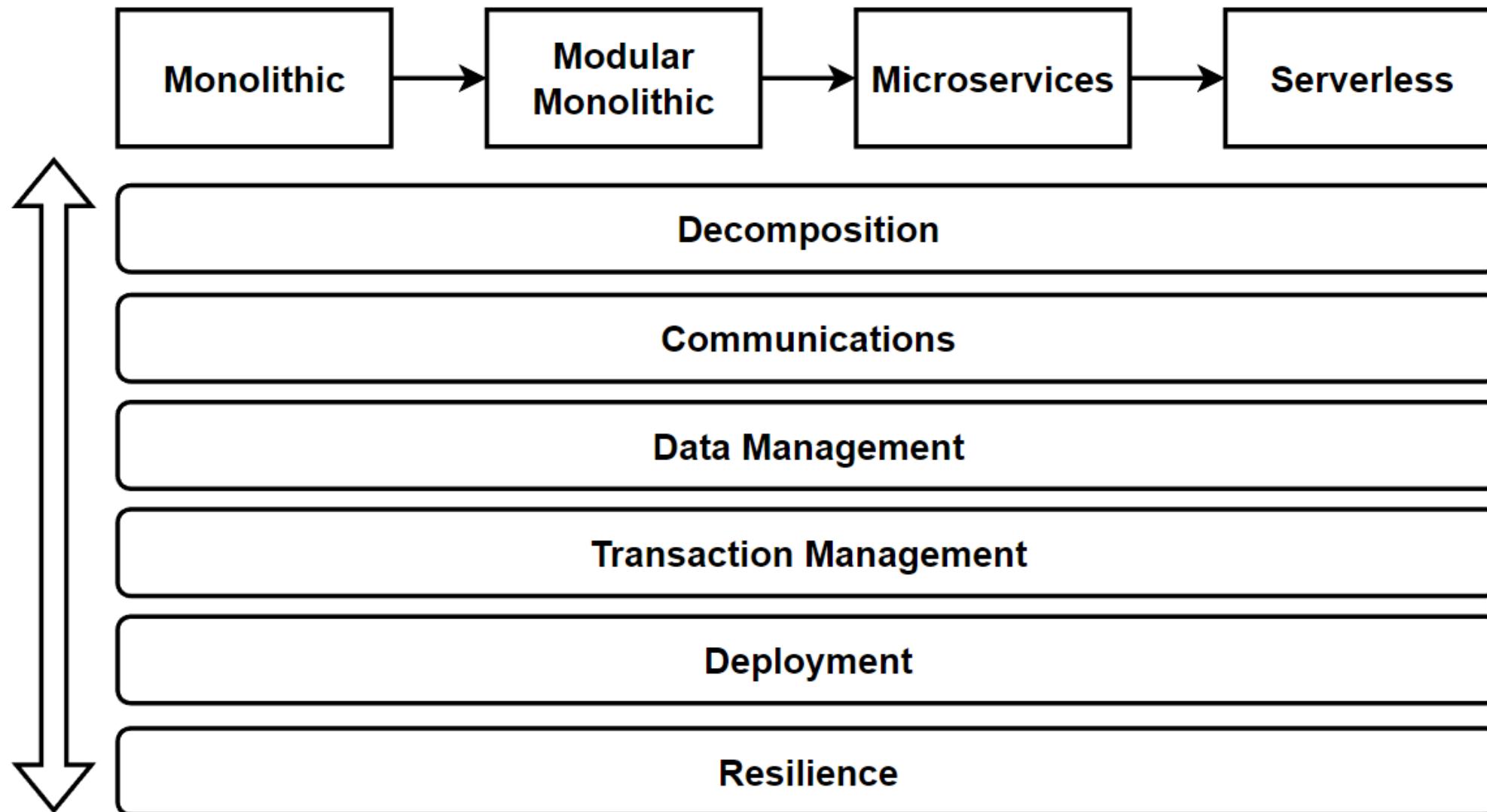


Main Considerations

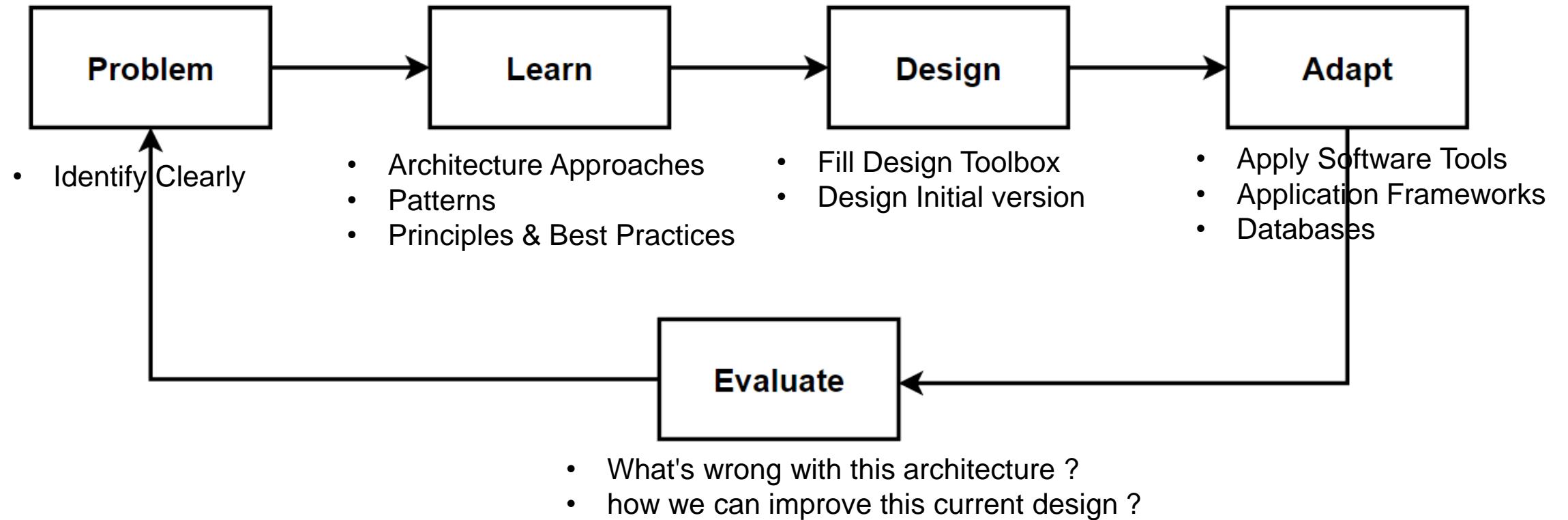
- Decomposition – Breaking Down Services
- Communications
- Data Management
- Transaction Management
- Deployments
- Resilience



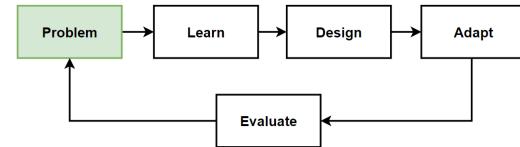
Architecture Design – Vertical Considerations



Way of Learning – The Course Flow



Problem: Break Down Application into Microservices

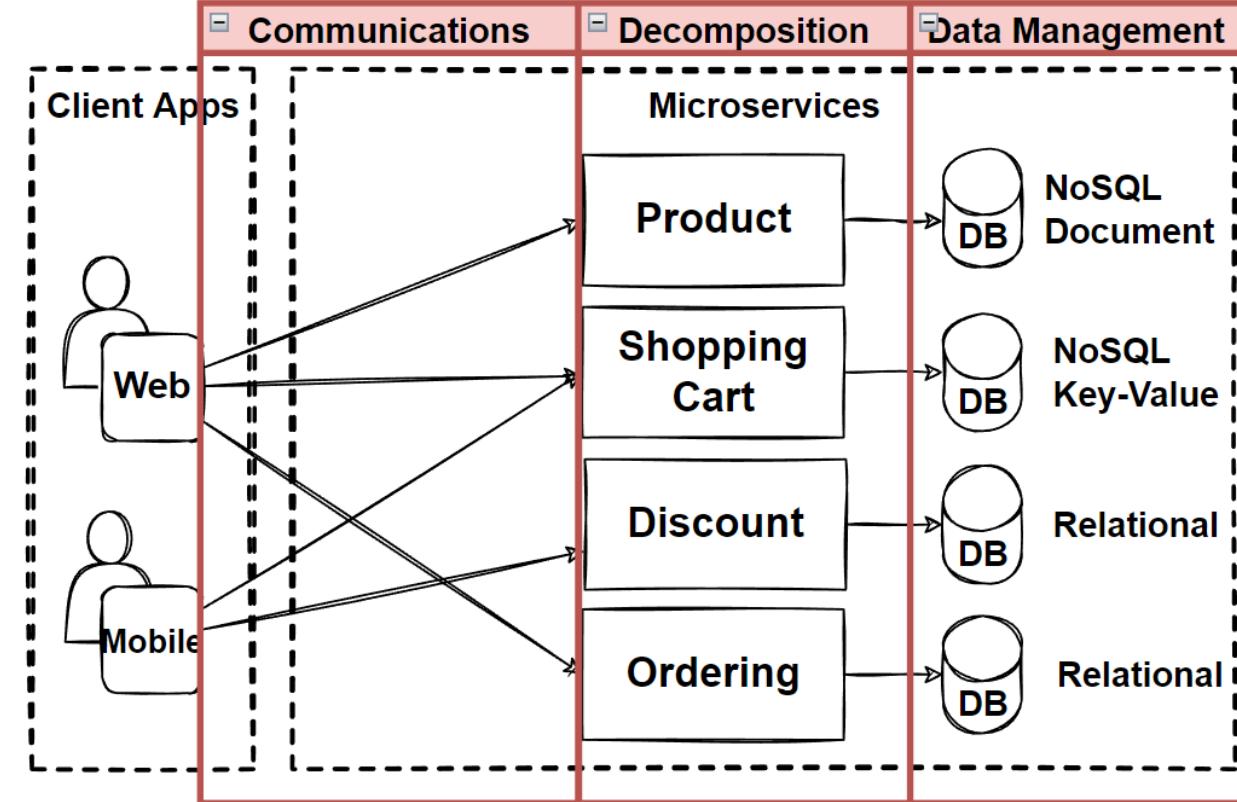


Problems

- Our E-Commerce Business is growing
- Teams want to be agile and add new features immediately to compete the market
- Required Independent Scale and Deployments
- We should clearly identify microservices which parts could be independent scale and deploy

Solutions

- Microservices Decomposition Patterns



Decomposition of Microservices Architecture

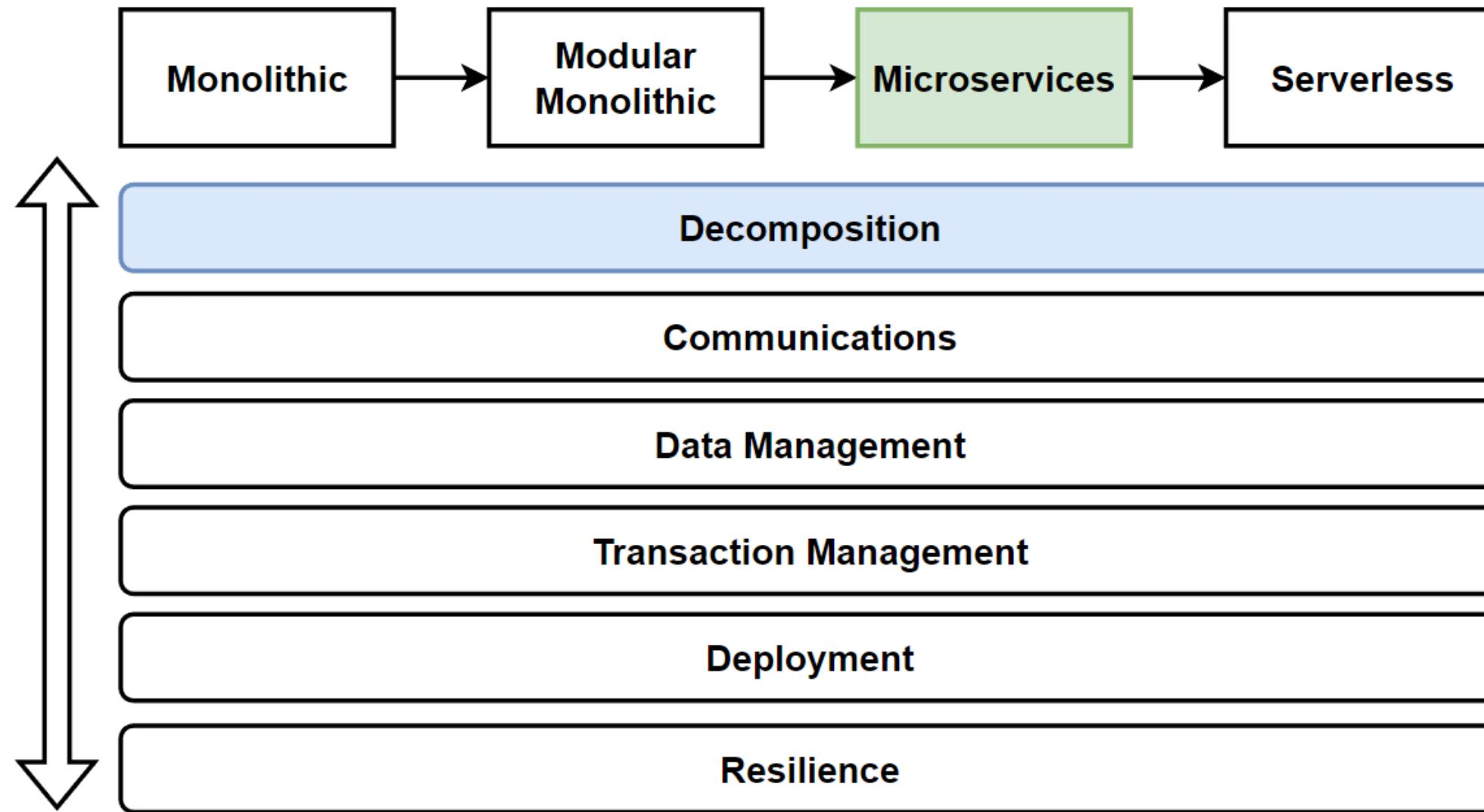
Breaking Down Application into Microservices

Microservices Decomposition Patterns

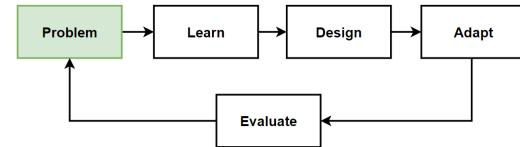
Decomposing Applications for Independent Deployability and Scalability

Breaking Down our E-Commerce application with Microservices Decomposition Patterns

Architecture Design – Vertical Considerations



Problem: Break Down Application into Microservices

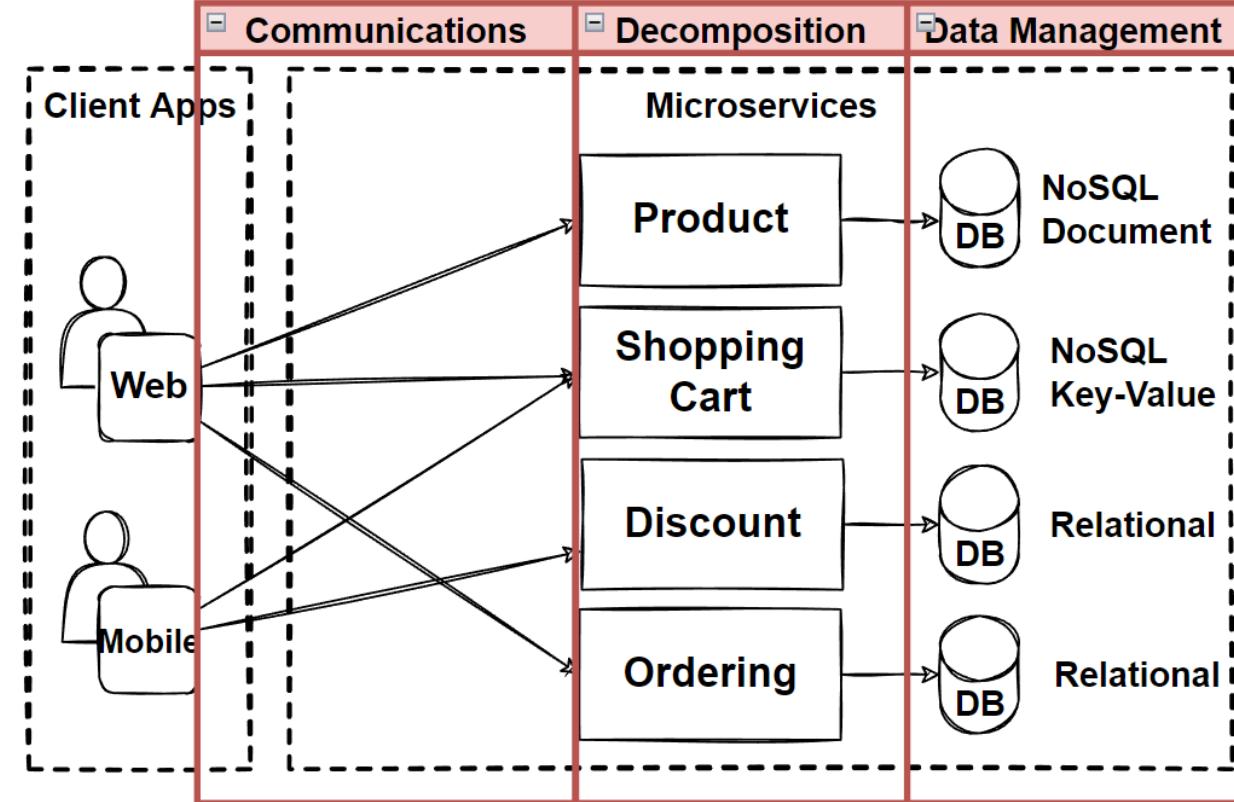


Problems

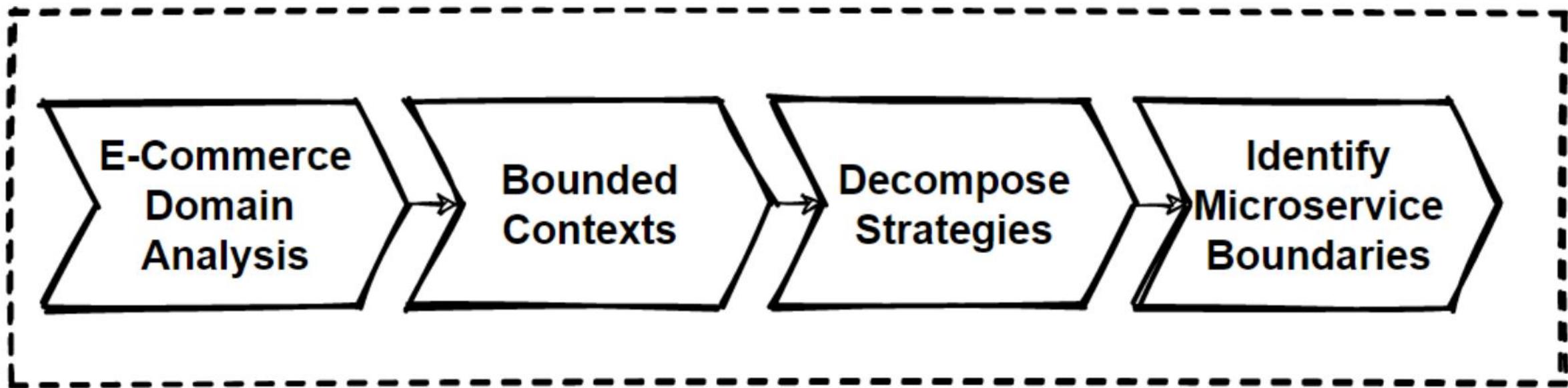
- Our E-Commerce Business is growing
- Teams want to be agile and add new features immediately to compete in the market
- **Required Independent Scale and Deployments**
- We should clearly identify microservices which parts could be independent scale and deploy

Solutions

- **Microservices Decomposition Patterns**

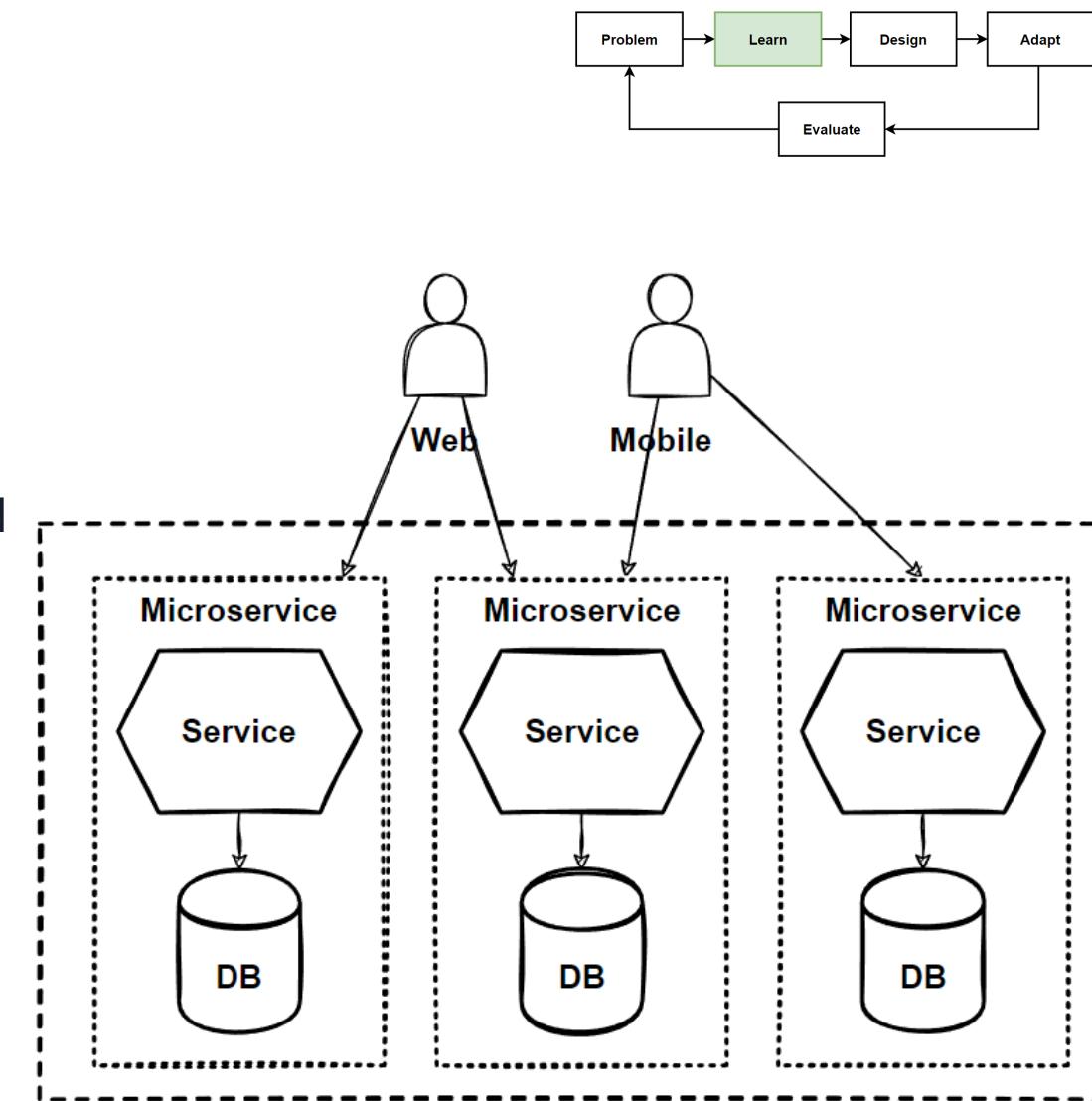


Microservices Decomposition Path



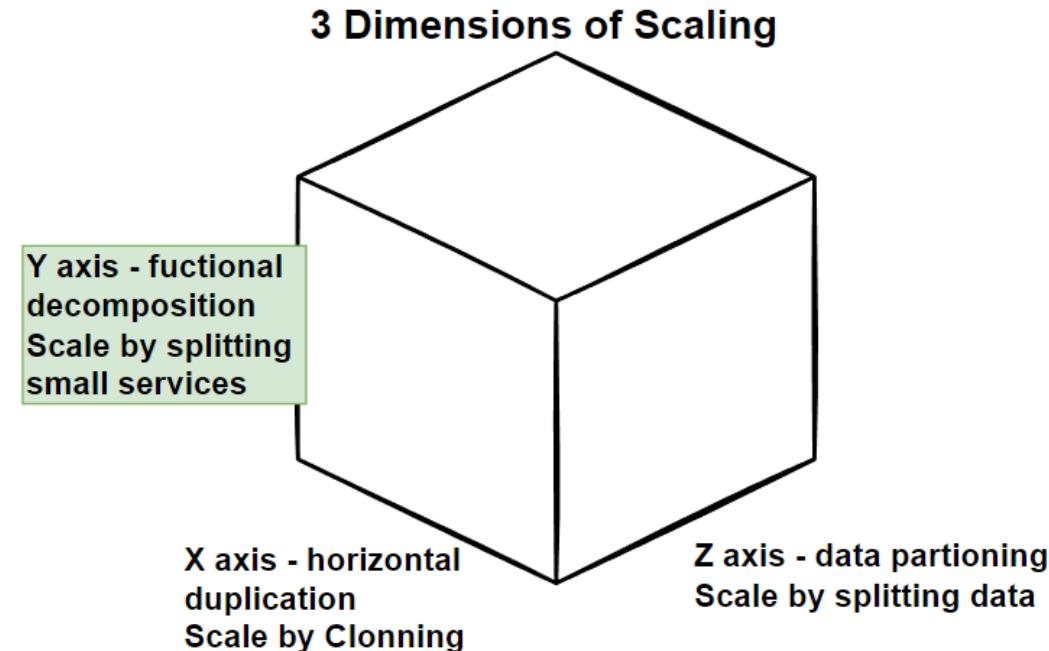
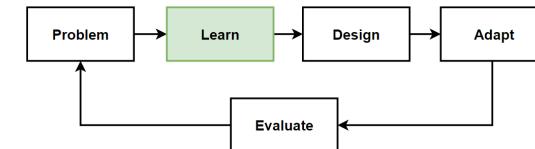
Why we need to Decompose ?

- What is the **main reason** behind the **decomposition** of microservices ? provide to **Scale Independently**.
- Main benefits of microservices are "**Independent Scale and Deployments**".
- **Clearly identify microservices** which parts could be required independent scale and deploy.
- **Design Principle: Decompose services by scalability requirements**
- Applications are consist of multiple modules or services, with **different requirements for scaling**.
- Example: Our **e-commerce application** have a public-facing website and a separate administration website.
- 2 services required **different scalability requirements**.



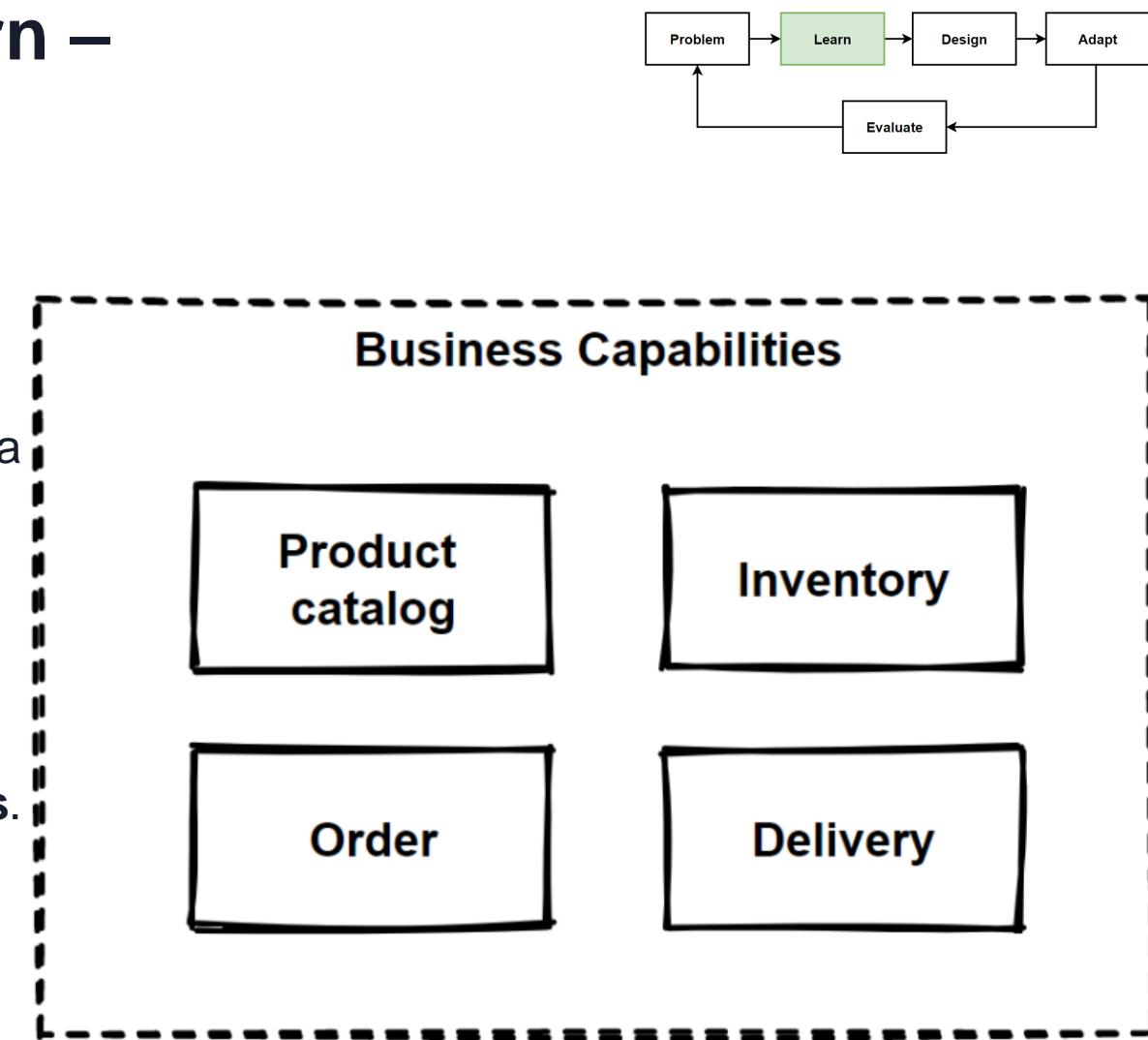
The Scale Cube

- **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
- **Functional decomposition** can be achieved by decoupling your architecture into functions **with Y-axis**.
- **Microservices** is an example for **functional decomposing**.
- **Y-axis** scaling splits the application into multiple, different services.
- **Combining** both **X** and **Y-Axis** scaling with microservices architecture can give **better scalability**.



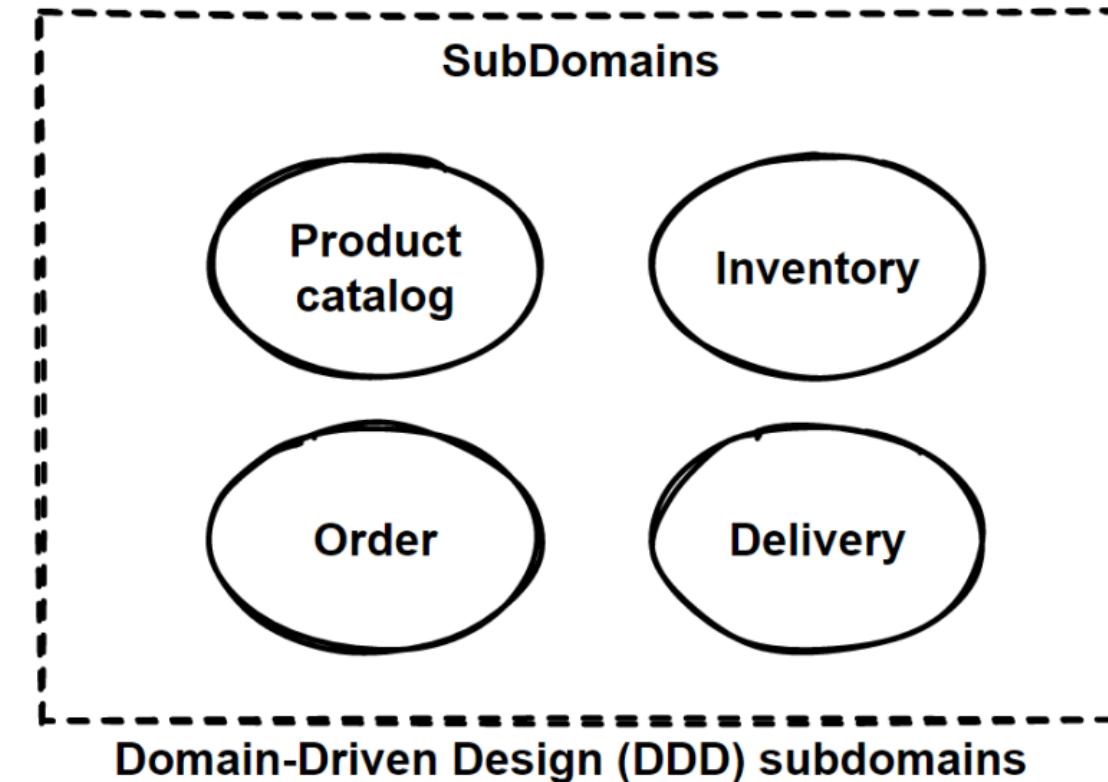
Microservices Decomposition Pattern – Decompose by Business Capability

- In Microservices Architecture, **split the application** as a set of **loosely coupled services**.
- **2 Prerequisite** of decomposition of microservices:
- **Services** must be **cohesive**. A service should implement a small set of strongly related functions.
- **Services** must be **loosely coupled** - each service as an API that encapsulates its implementation.
- "Decompose by Business Capability" pattern offer:
- **Define services corresponding to business capabilities**.
- A **business capability** is a concept from **business architecture modeling**.
- A **business** service should **generate value**.



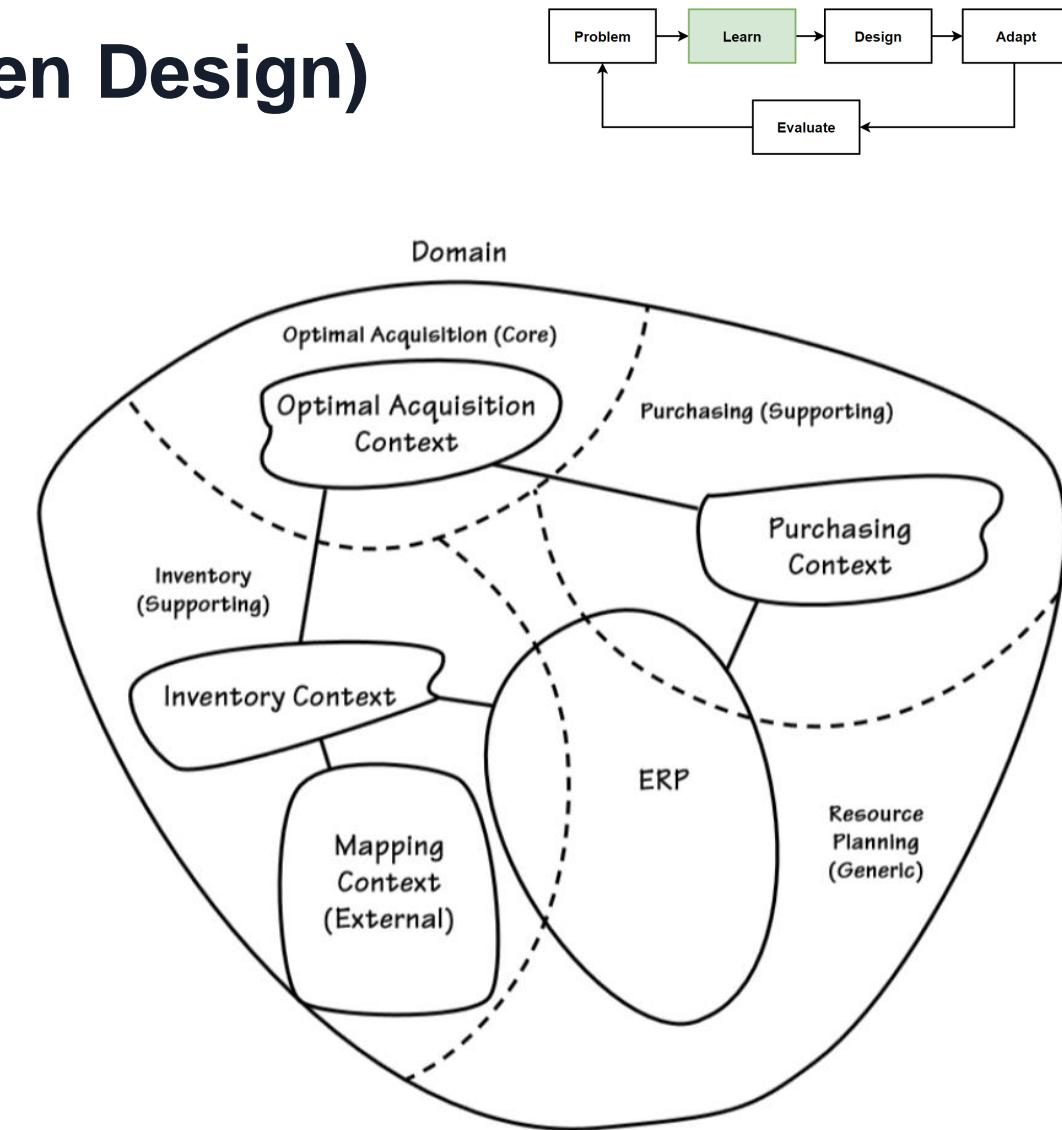
Microservices Decomposition Pattern – Decompose by Subdomain

- **Services must be cohesive.** A service should implement a small set of strongly related functions.
- **Services must be loosely coupled** - each service as an API that encapsulates its implementation.
- "Decompose by Subdomain" pattern offer:
- Define services corresponding to **Domain-Driven Design (DDD) subdomains**.
- DDD refers to the application's problem space, the **business as the domain**. A **domain** consists of **multiple subdomains**.
- **Each subdomain** corresponds to a **different part** of the business.



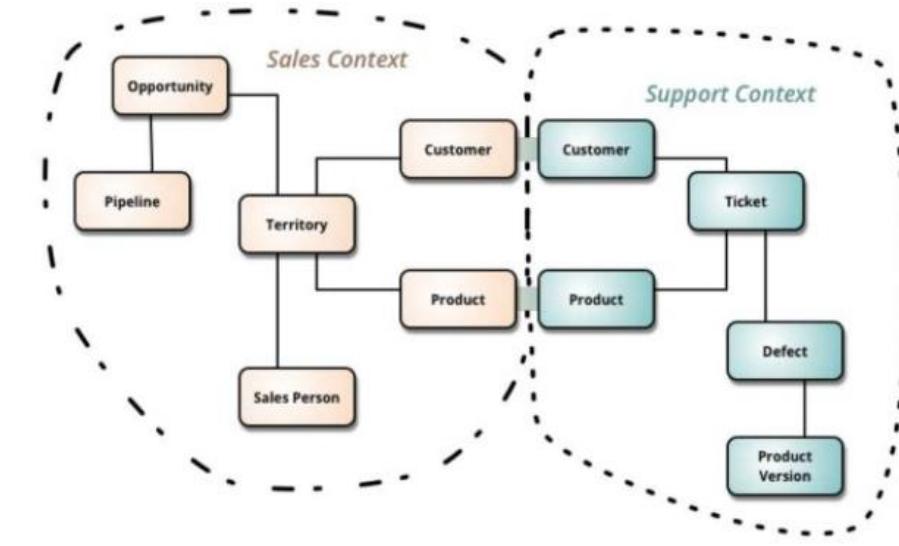
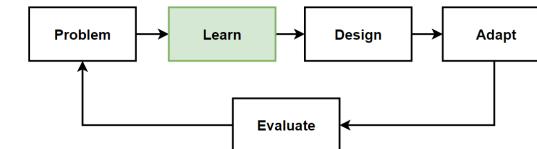
Bounded Context Pattern (Domain-Driven Design)

- **DDD - Bounded Context Pattern** which is one of the main pattern that we mainly use when **decomposing microservices**.
- **Domains** are require **high cooperation** and have a certain complexity by nature are called **collaborative domains**.
- DDD has **2 phases, Strategic and Tactical DDD**.
- **Strategic DDD**, we define the large-scale model of the system, defining to the business rules that allow designing loosely coupling units and the context map between them.
- **Tactical DDD** focuses on implementation and provides design patterns that we can use to build the software implementation.
- Include concepts such as **entity, aggregate, value object, repository, and domain service**.



Bounded Context Pattern (Domain-Driven Design)-2

- DDD domain defines its **own common language** and divides boundaries into specific, independent components. Common language is called **ubiquitous language**, and independent units are called **bounded context**.
- DDD is solving a **complex** problem is to break the problem into **smaller parts** and focus on smaller problems that are relatively easy.
- A **complex domain** may contain **sub domains**. And some of sub domains can **combine and grouping** with each other for **common rules and responsibilities**.
- **Bounded Context** is the grouping of closely related scopes that we can say **logical boundaries**.
- **Bounded context** is the **logical boundary** that represents the smaller problem particles of the complex domain that **are self-consistent** and as **independent** as possible.

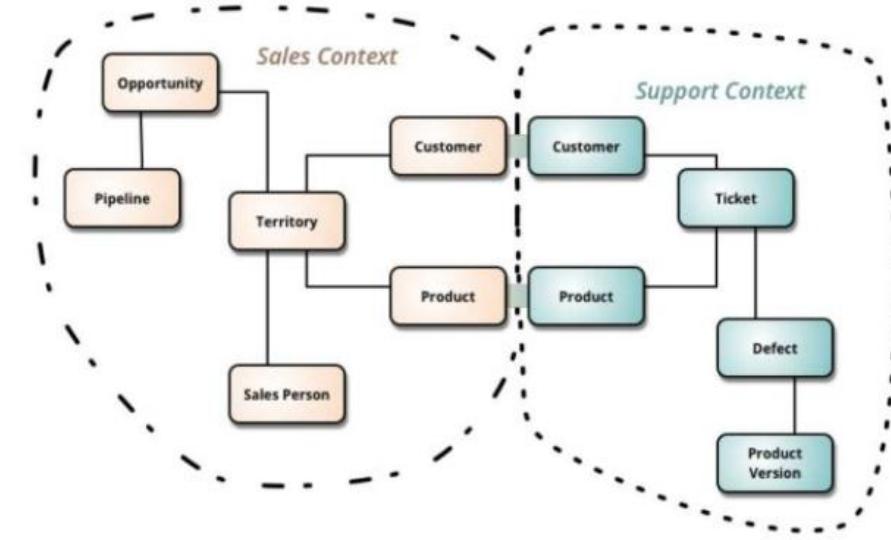
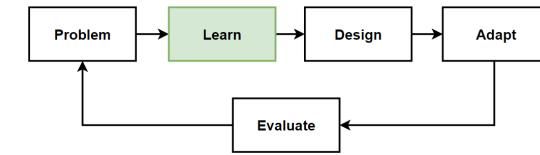


DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.

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

Identify Bounded Context Boundaries for Each Microservices

- How we can identify **Bounded Context** ?
- To identify bounded contexts use **DDD**.
- In DDD, use the **Context Mapping pattern** for identification of bounded contexts.
- With **Context Mapping Pattern**, we can identify the whole bounded contexts in the application and with their **logical boundaries**.
- The solution is using **Context Mapping**.
- The **Context Map** is a way to define **logical boundaries** between domains.

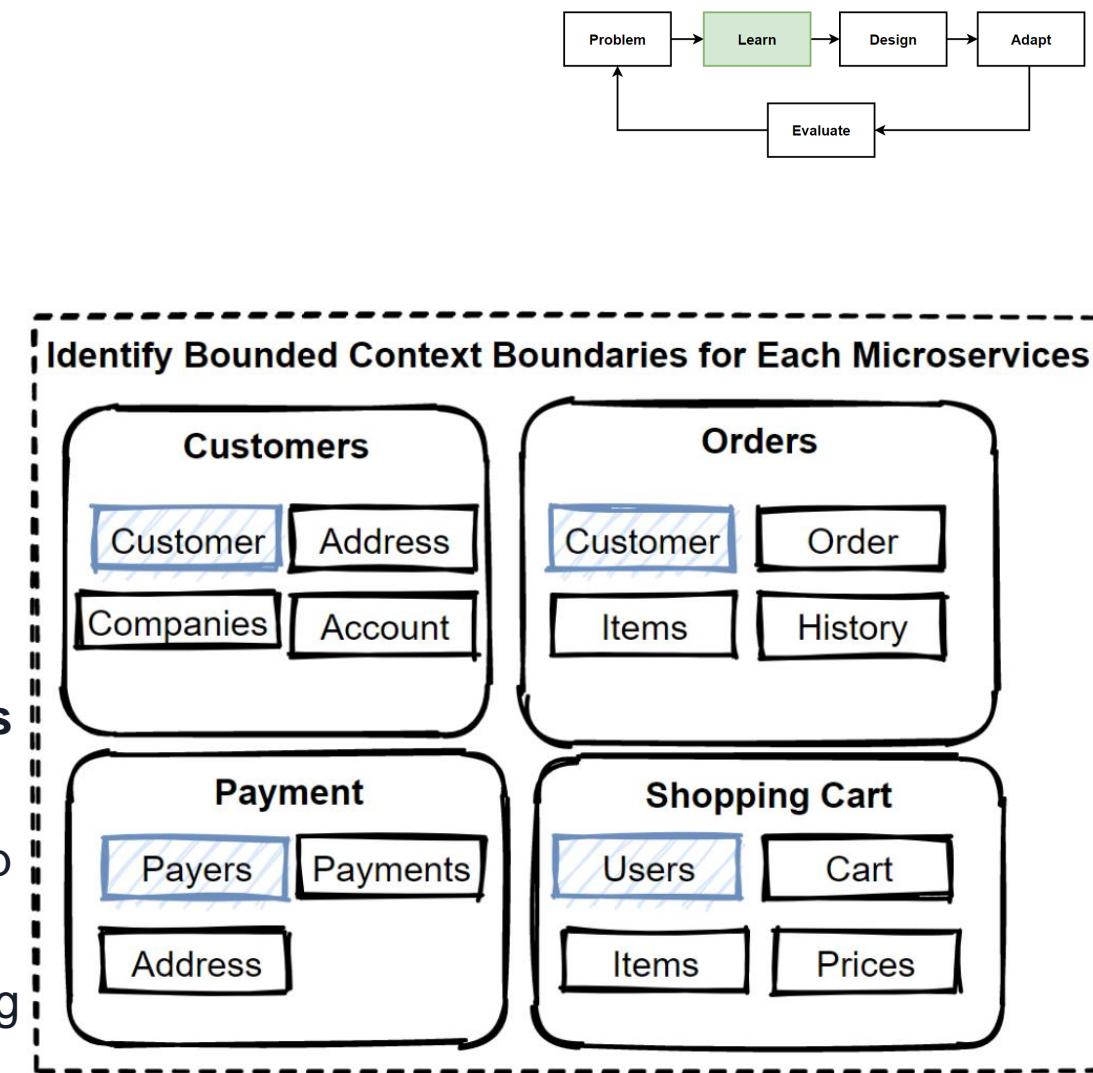


DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.

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

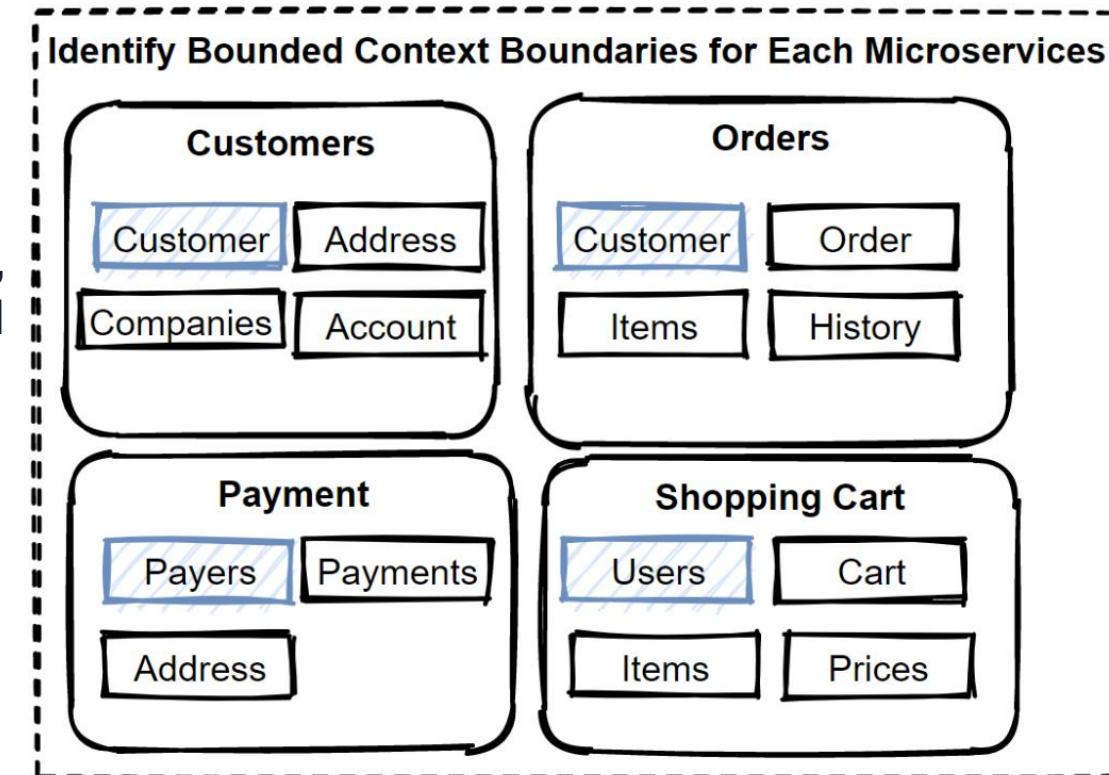
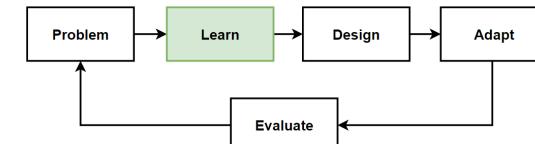
Identify Bounded Context Boundaries for Each Microservices-2

- Identify the **Bounded Contexts** by talking to the **domain experts** and using some clues.
- Once defined the Bounded Contexts, iterate design, those are **not immutable**.
- Reshape your **Bounded Contexts** by talking to the domain experts and consider **refactoring's** with the changing conditions.
- Its crucial to discuss with **domain experts** to defining **domains** and **sub domains**.
- Evaluate **Bounded Context** with the **domain experts** will help you identify to microservices.
- **Sub domains** inside of the **Bounded Context** are representing same data but naming differently due to domain experts areas.
- Should discuss **several domain experts** for their expertise areas.



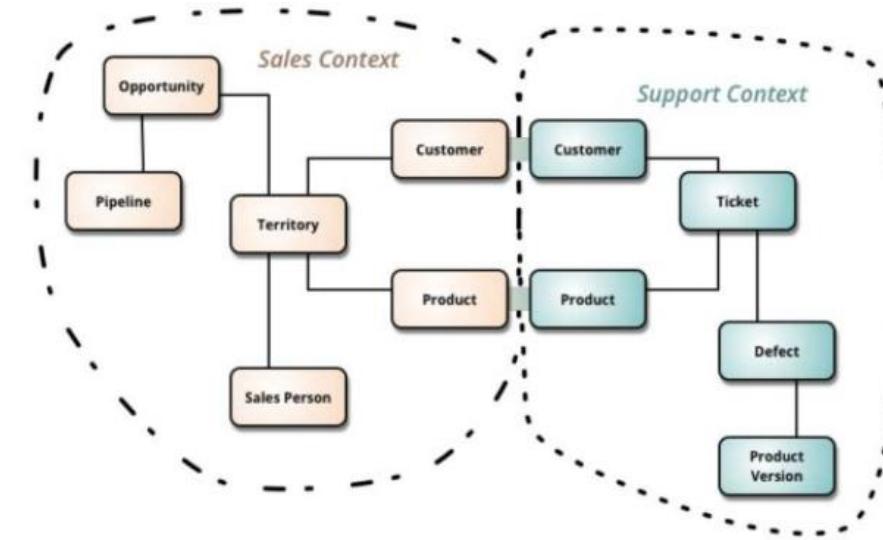
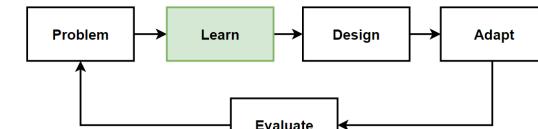
A Bounded Context == A Microservice ?

- **No right answer** to this question under all circumstances.
- **Bounded Context** can create **more than one Microservice**.
- Decision to be made based on the **microservice's** need for **scalability** and **independence**.
- Since Bounded Context defines the boundaries of the domain, a **Microservice** determines the **technical and organizational boundaries**.
- Similar to Microservices, **Bounded Contexts** are **autonomous** and **responsible** by certain **domain capability**.
- **Context Mapping** and the **Bounded Context pattern** are good approaches for **identifying microservices**.



Using Domain Analysis to Model Microservices

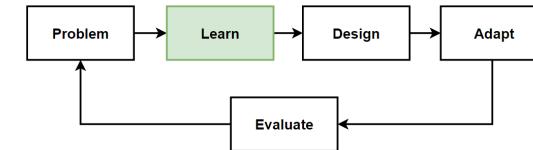
- **Microservices** should be designed by **business capabilities** and should have **loose coupling** and **autonomous services**.
- We **can change** a particular microservices **without affecting** other services. Each service can be change **independently**.
- **Domain-driven design (DDD)** provides a set of methodology that we can follow the principles and create a well-designed microservices.
- Follow **DDD-Bounded Context** which following **Context Mapping Pattern** and **decompose** by **sub domain models** patterns.



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

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

Checklist After Decompose Microservices



- **Microservice should do "one thing"**

No certain process that will produce the right design. But Each service should has a single responsibility. Think deeply about our business domain, bounded contexts and sub domain models patterns.

- **Microservice size should not too big and not too small**

Start from a carefully designed domain model and group small sub models with obeying this rule. Each service is small enough that it can be built by a small team working independently.

- **Avoid Chatty Communication**

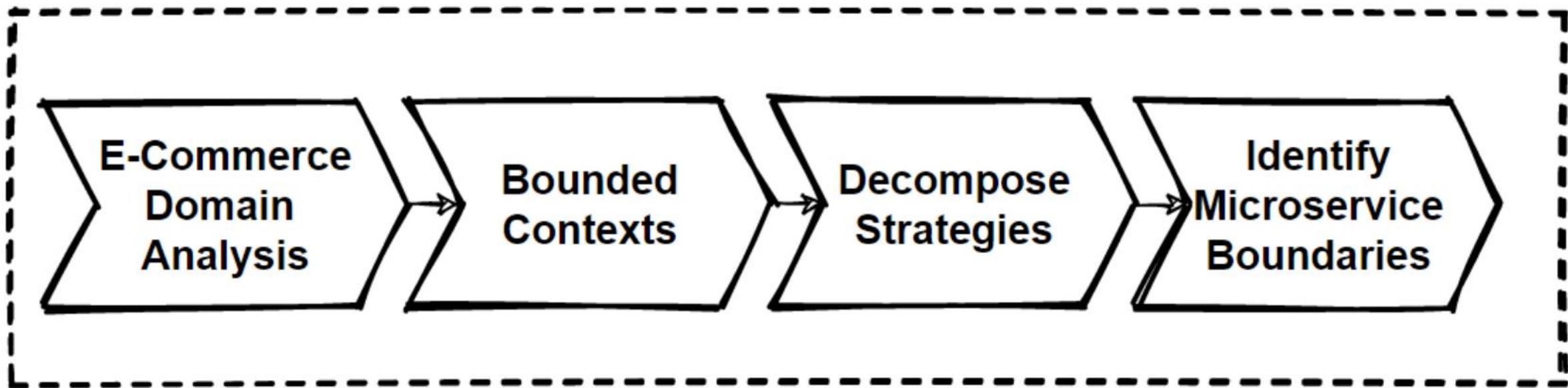
When you splitting functionality into two services, if those services becomes overly chatty communications, then its good to combine them into 1 service.

- **No Locking Dependencies**

If your services has inter-service dependencies more than 2 or 3, and if those are required to move and deploy together that means there are pain points of your design and its good to re-think again.

- It should always be possible to deploy a microservice without re-deploying any other services. Services should not be tightly coupled, and can evolve independently.

Microservices Decomposition Path

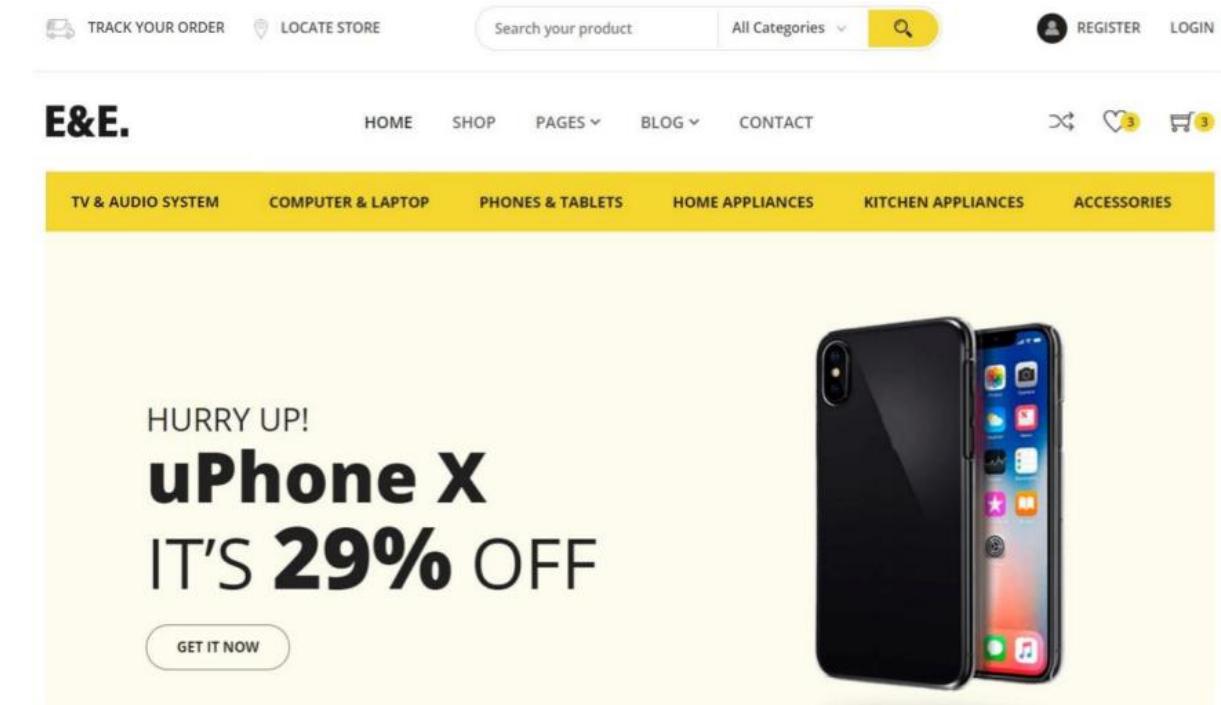


Understand E-Commerce Domain

- Our Domain: E-Commerce
- Understand Domain and Decompose small pieces
 - Use Cases
 - Functional Requirements

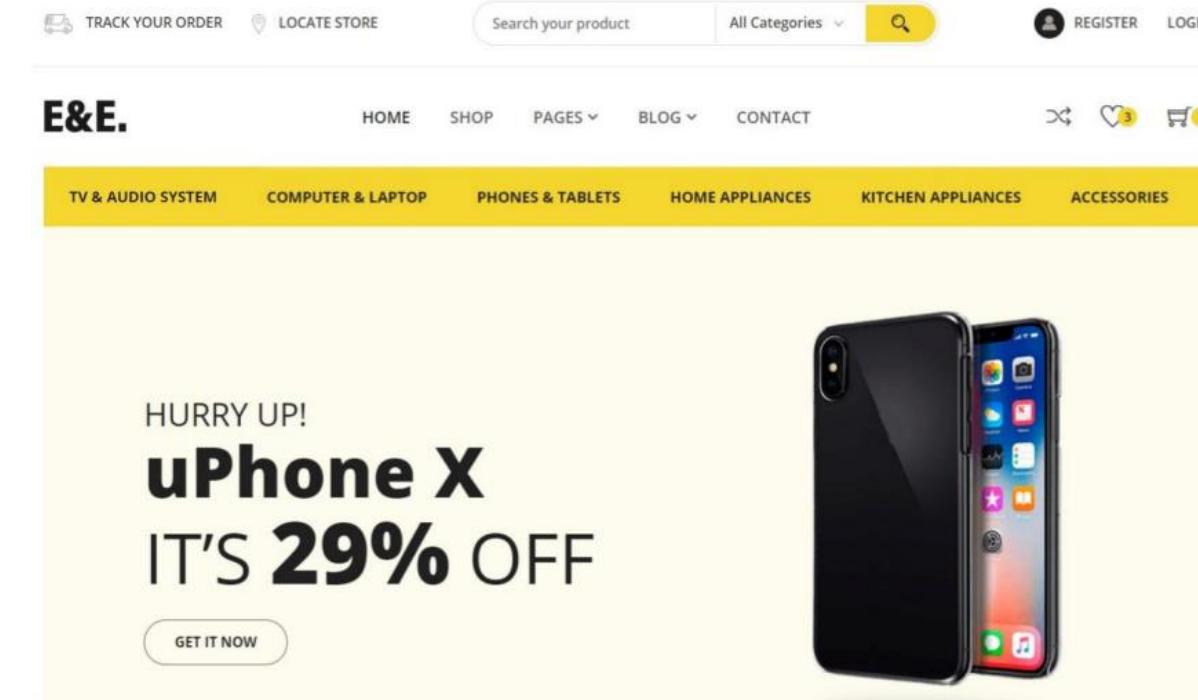
Identify steps:

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



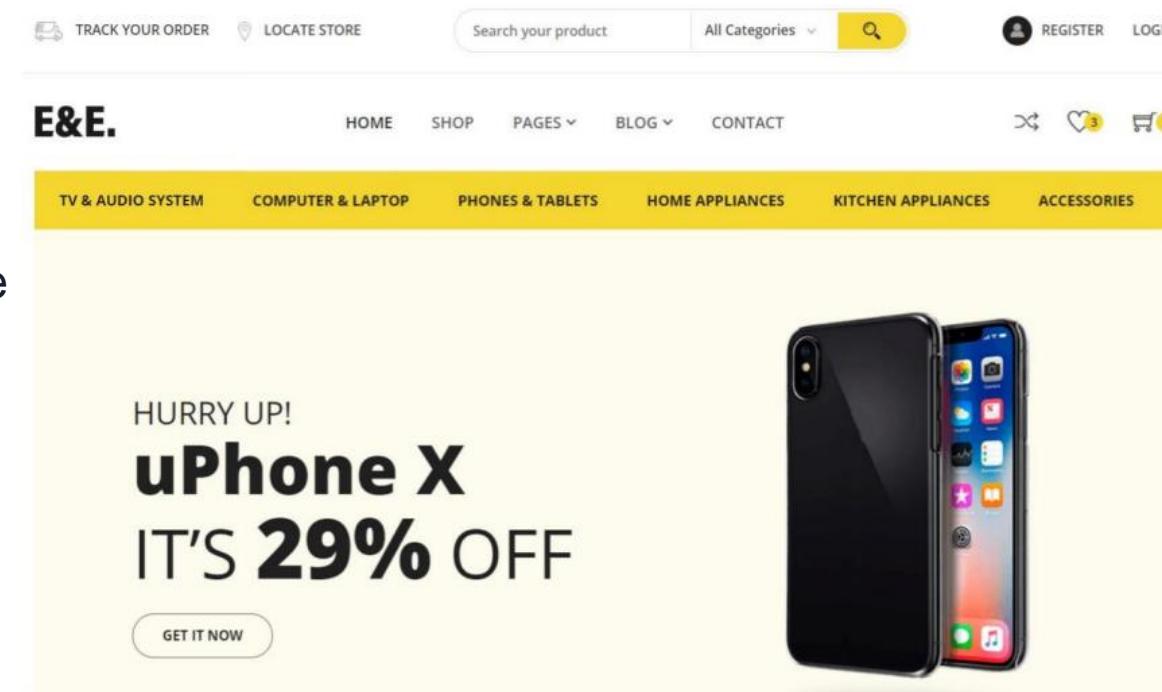
Understand E-Commerce Domain: Functional Requirements

- 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



Understand E-Commerce Domain: User Stories (Use Cases)

- 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

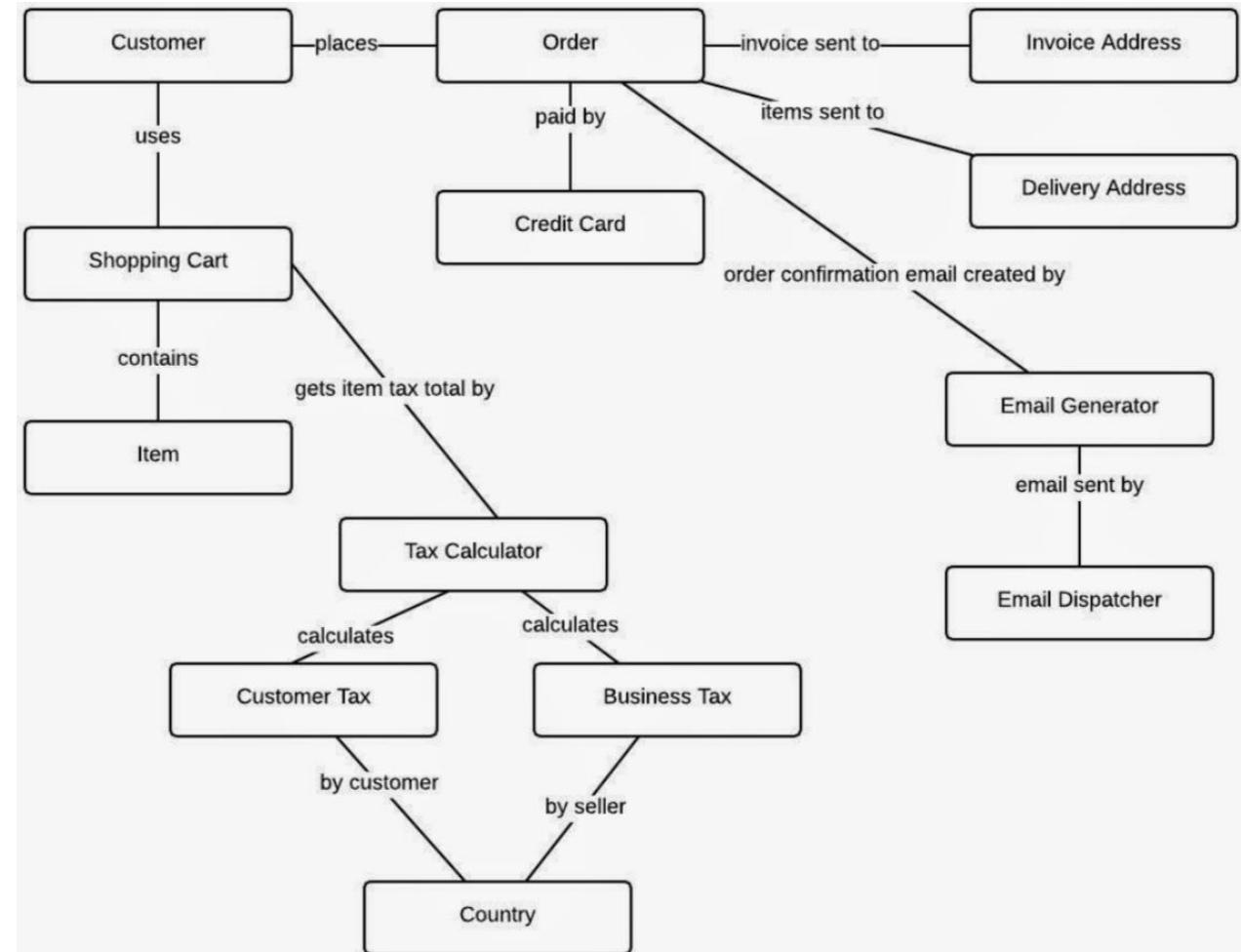


Analysis E-Commerce Domain - Nouns and Verbs

- 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 I can check out
- 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
- 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
- As a user I want to be able to **specify** my **credit card** information during **check out** so 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

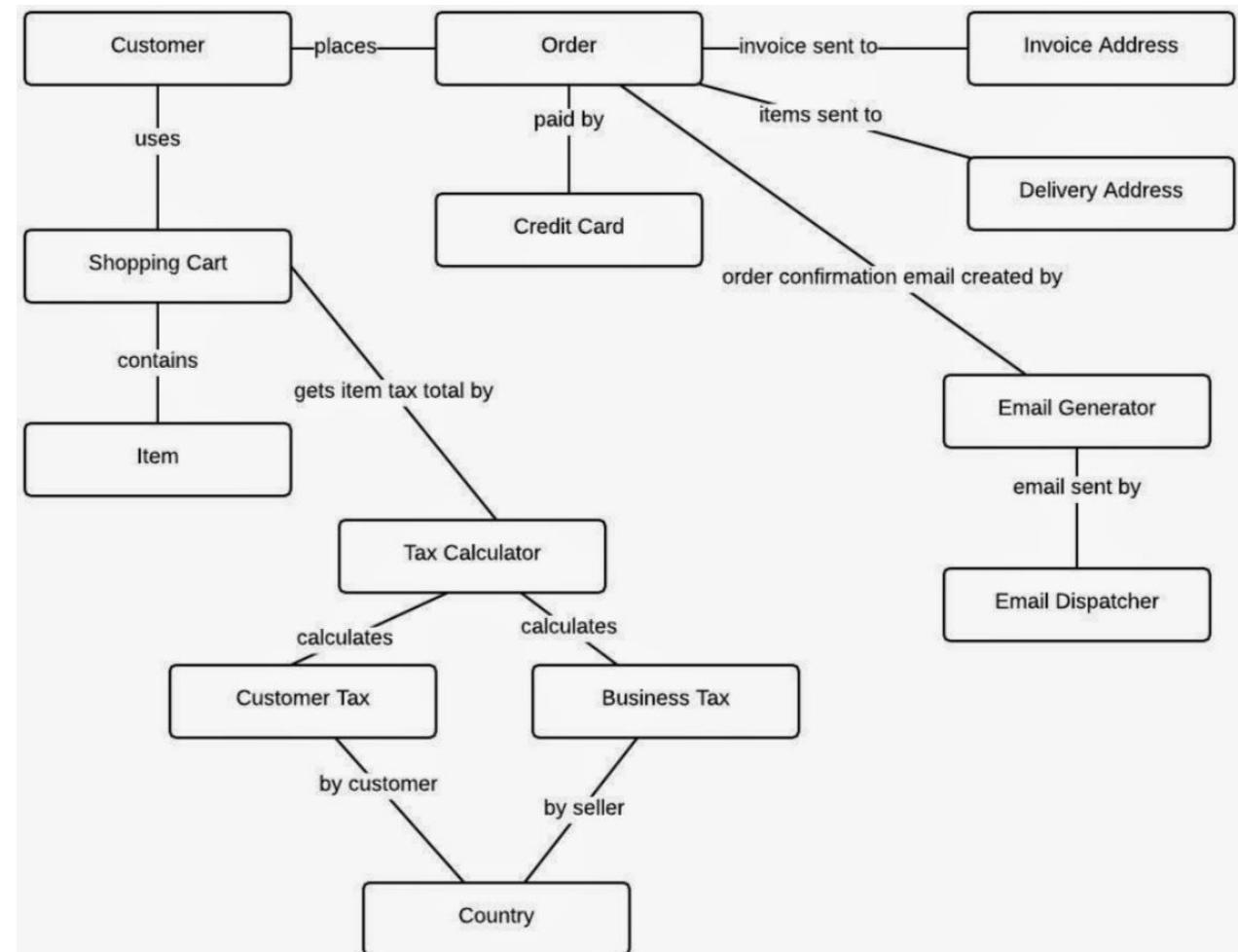
Analysis E-Commerce Domain - Nouns

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

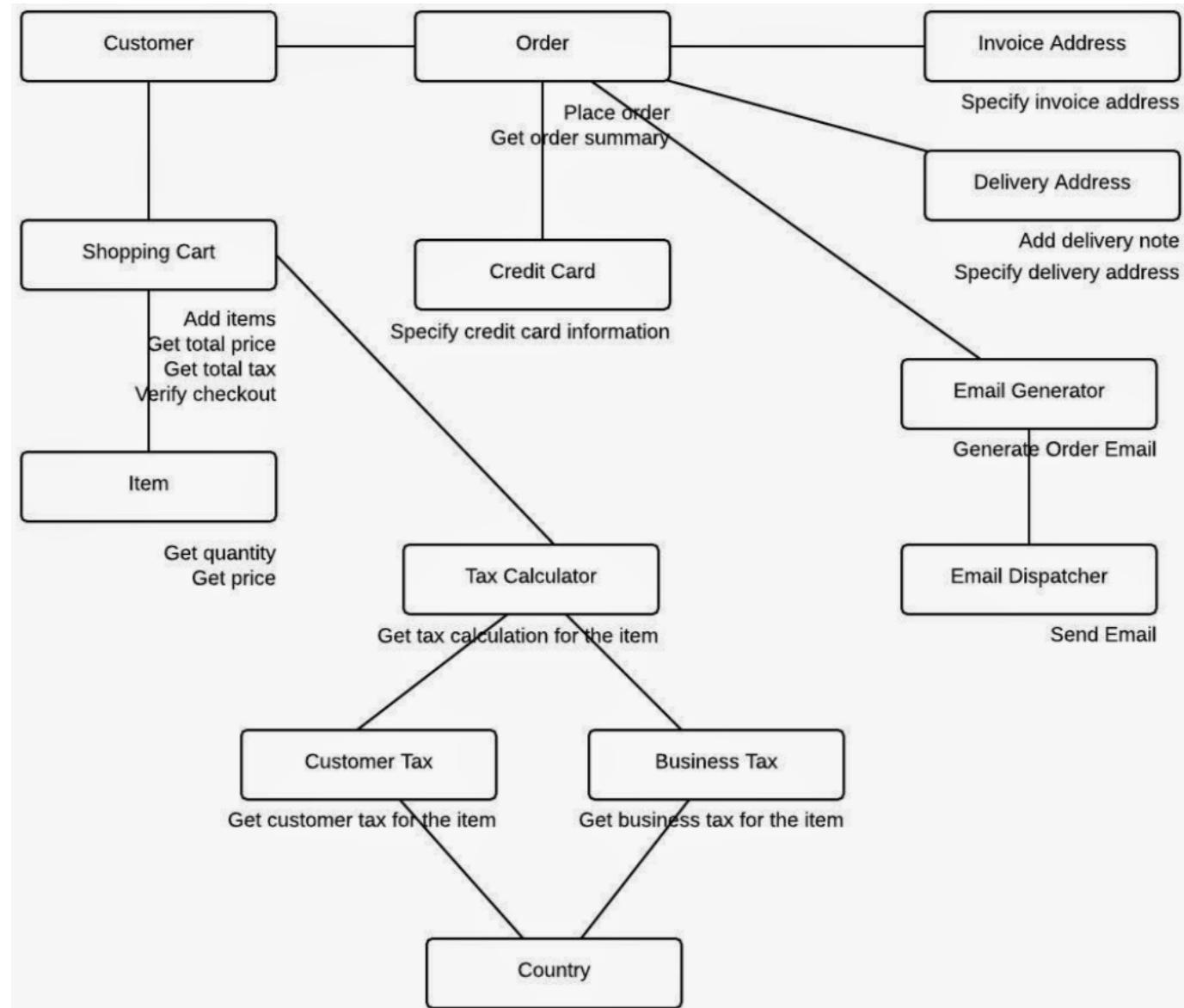


Analysis E-Commerce Domain - Verbs

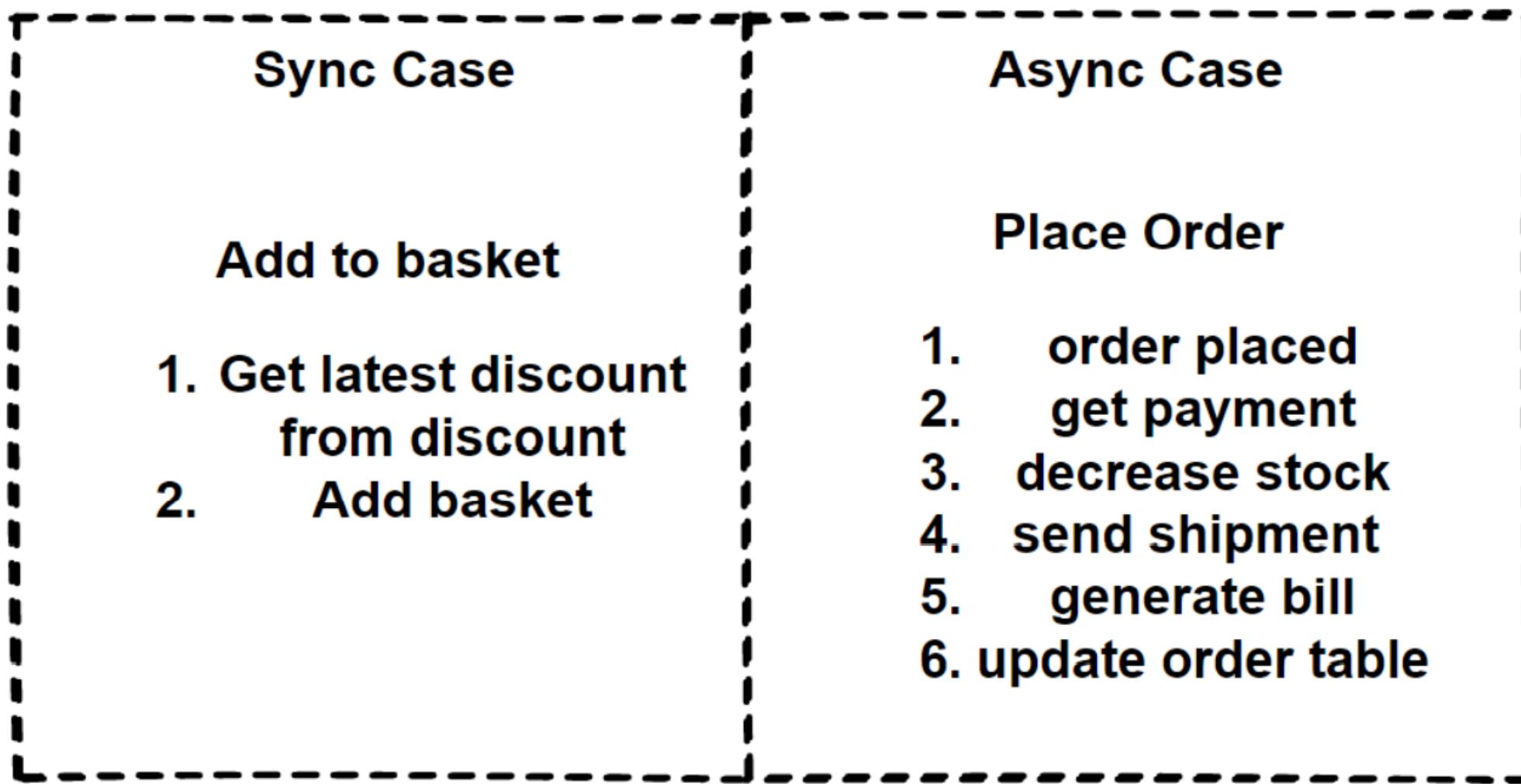
- List products applying to paging
- Filter products by brand, category and supplier
- See product all information in the details screen
- Put products in to the shopping cart
- See total cost for all of the items
- See total cost for each item
- Checkout order with purchase steps
- Specify delivery 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



Object Responsibility Diagram



2 Main Use Case of Our E-Commerce Application



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

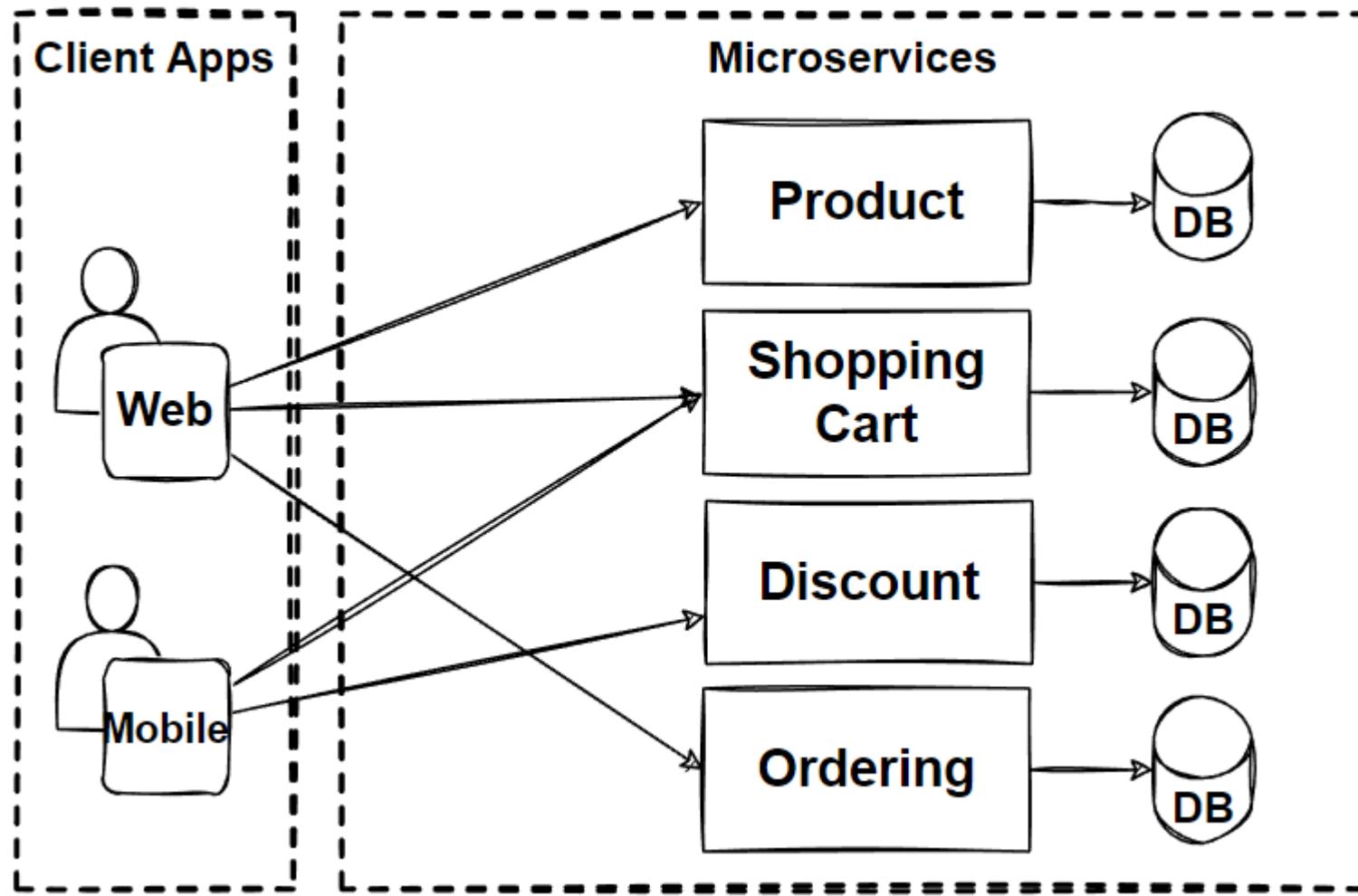
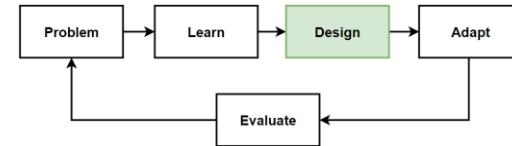
Rating

Recommendation

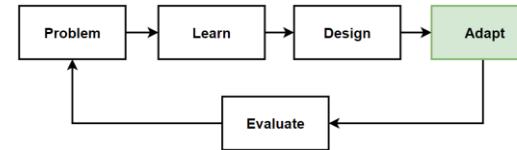
Before Design – What we have in our design toolbox ?

Architectures	Patterns&Principles	Non-FR	FR
<ul style="list-style-type: none">• Microservices Architecture	<ul style="list-style-type: none">• The Database-per-Service Pattern• Polygot Persistence• Decompose services by scalability• The Scale Cube• Microservices Decomposition Pattern<ul style="list-style-type: none">• Decompose by Business Capability• Decompose by Subdomain• Bounded• Context Pattern• Context Mapping pattern	<ul style="list-style-type: none">• High Scalability• High Availability• Millions of Concurrent User• Independent Deployable• Technology agnostic• Data isolation• Resilience and Fault isolation	<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history

Design: Microservices Architecture



Adapt: Microservices Architecture



Frontend SPAs

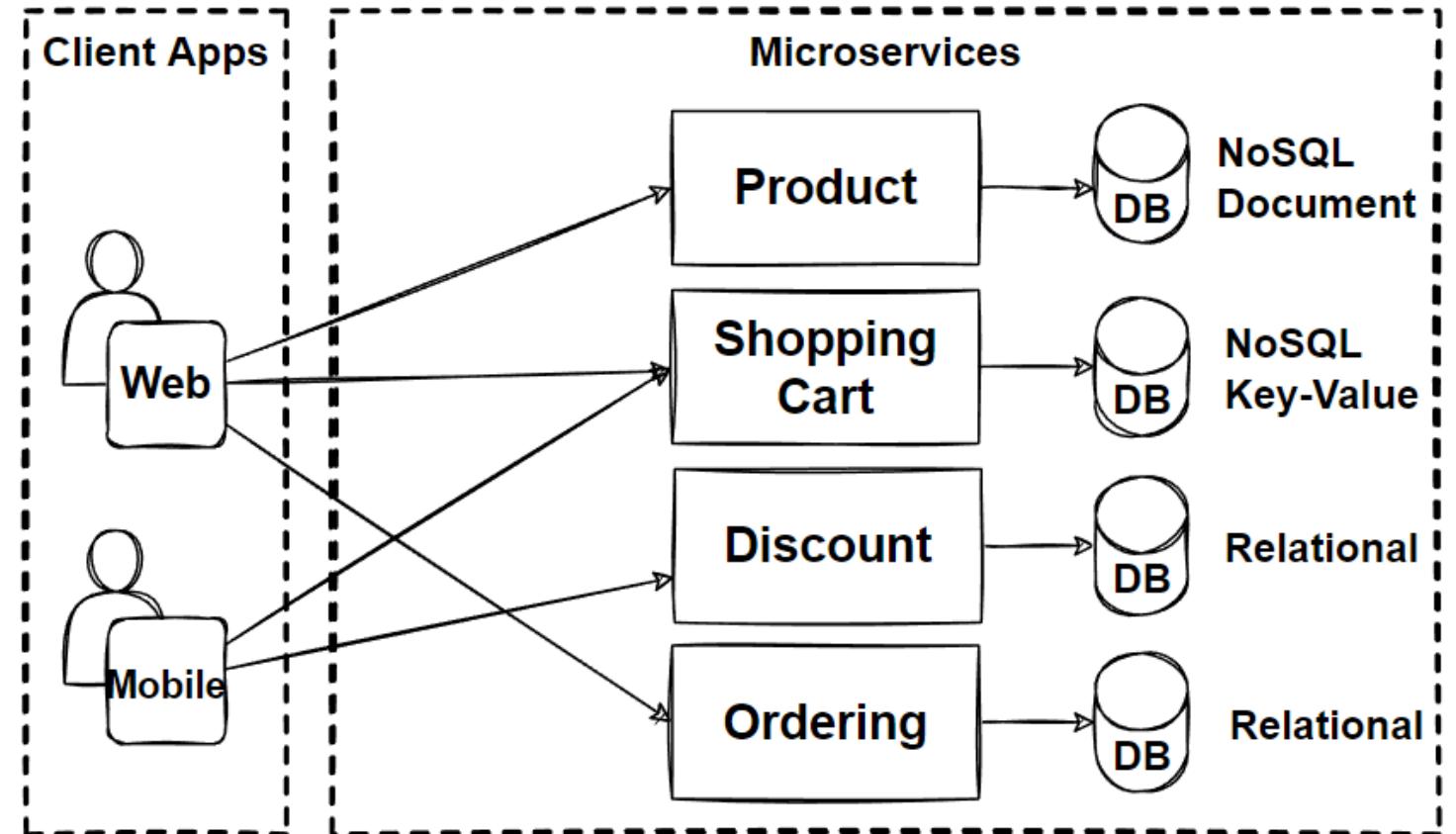
- Angular
- Vue
- React

Backend Microservices

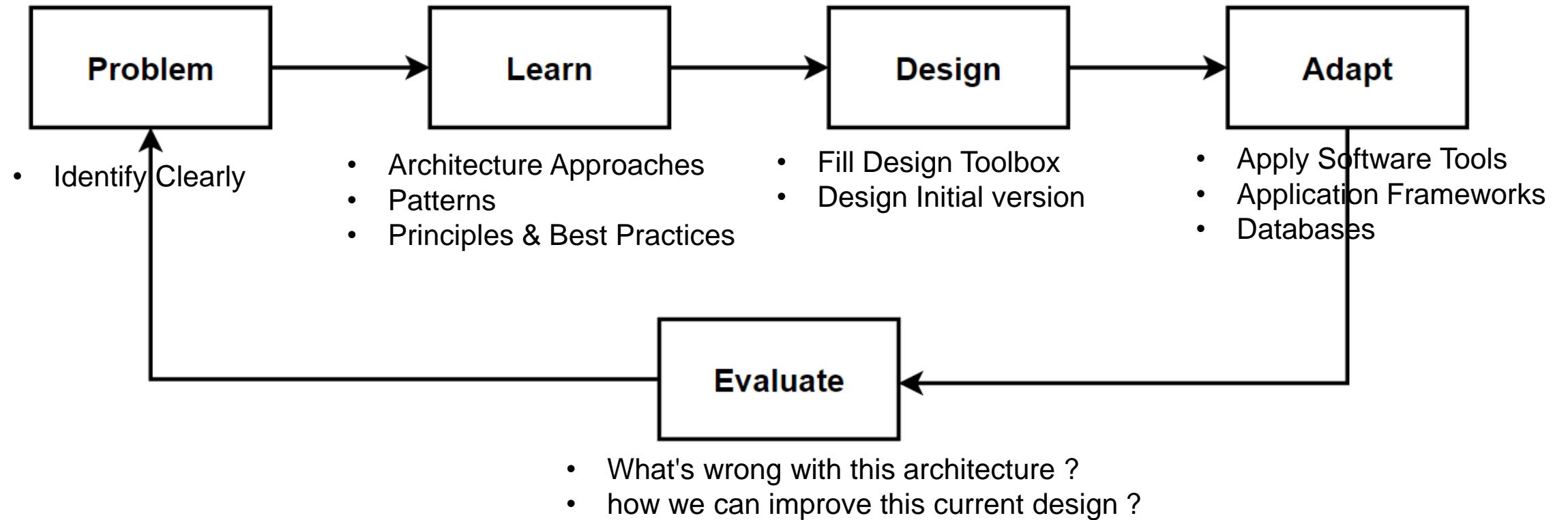
- Java – Spring Boot
- .Net – Asp.net
- JS – NodeJS
- Python – Django, Flask

Database

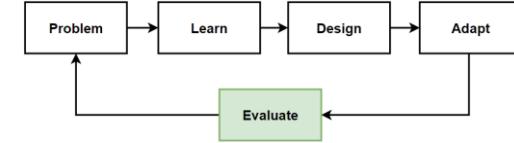
- MongoDB – NoSQL Document DB
- Redis – NoSQL Key-Value Pair
- Postgres – Relational
- SQL Server – Relational



Way of Learning – The Course Flow

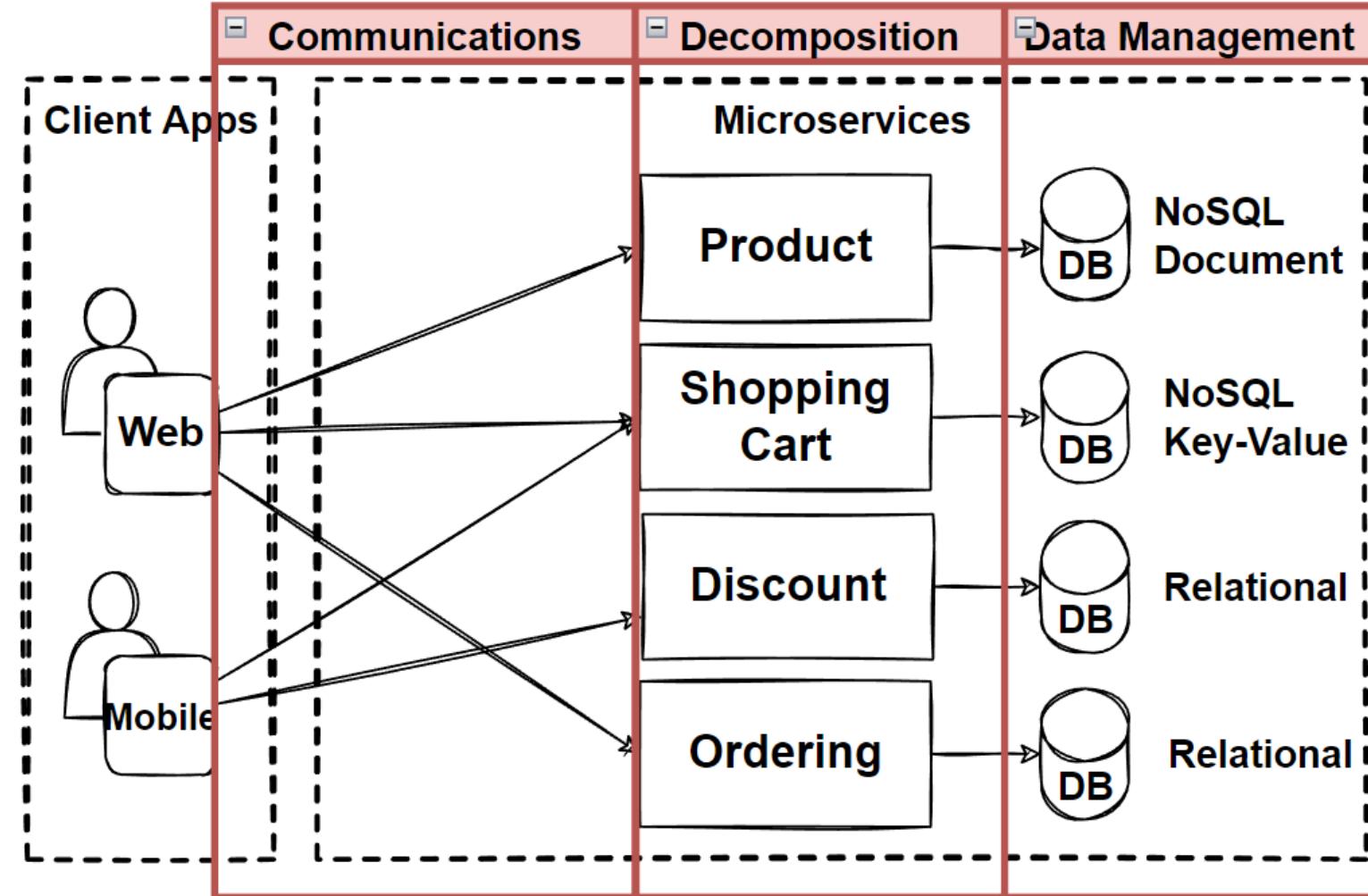


Evaluate: Microservices Architecture

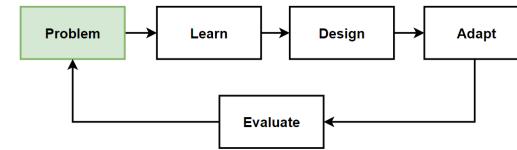


Main Considerations

- Decomposition – Breaking Down Services
- Communications
- Data Management
- Transaction Management
- Deployments
- Resilience



Problem: Direct Client-to-Service Communication

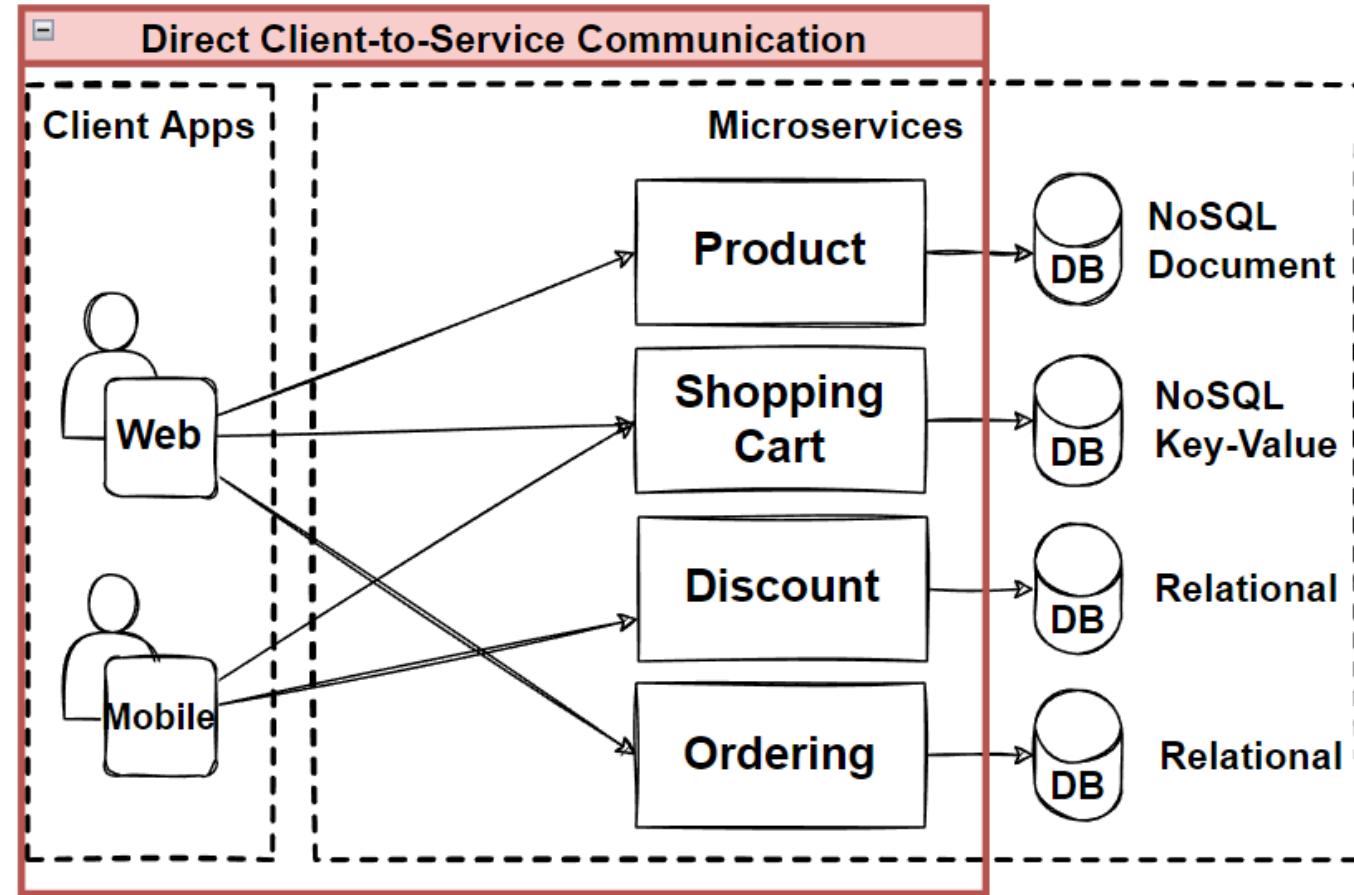


Problems

- Direct Client-to-Service Communication
- Cause to chatty calls from client to service
- Hard to manage invocations from client app.

Solutions

- Well-defined API Design
- Microservices Communication Patterns



Microservices Communications – The Basics

Microservices Communication Types - Synchronous or Asynchronous Communication

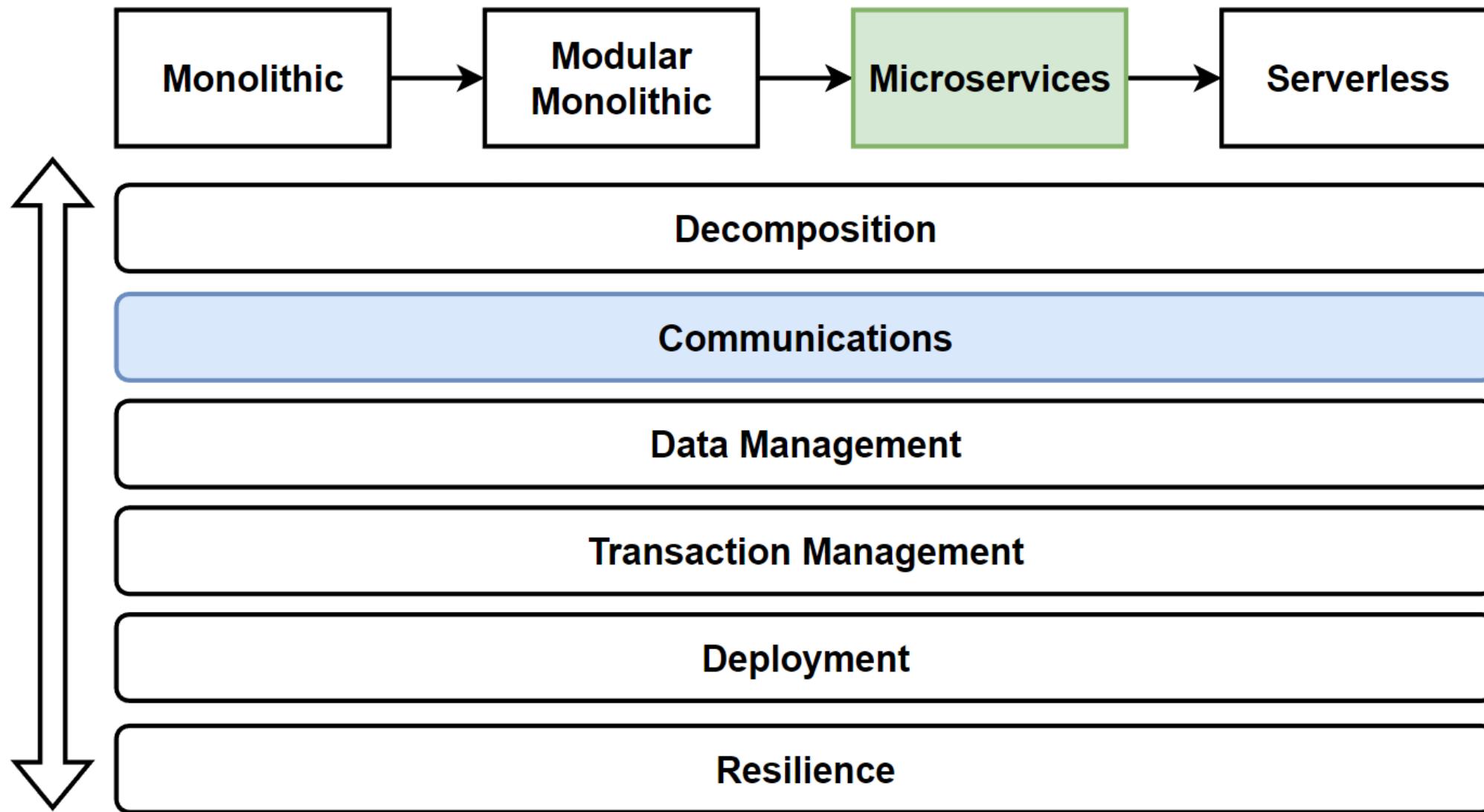
RESTful API design for Microservices

GraphQL API flexible structured relational data for Microservices

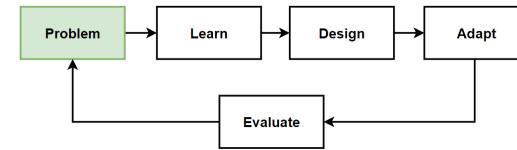
gRPC high performance communication between internal Microservices

WebSocket API Real-time two-way communications

Architecture Design – Vertical Considerations



Problem: Direct Client-to-Service Communication



Problems

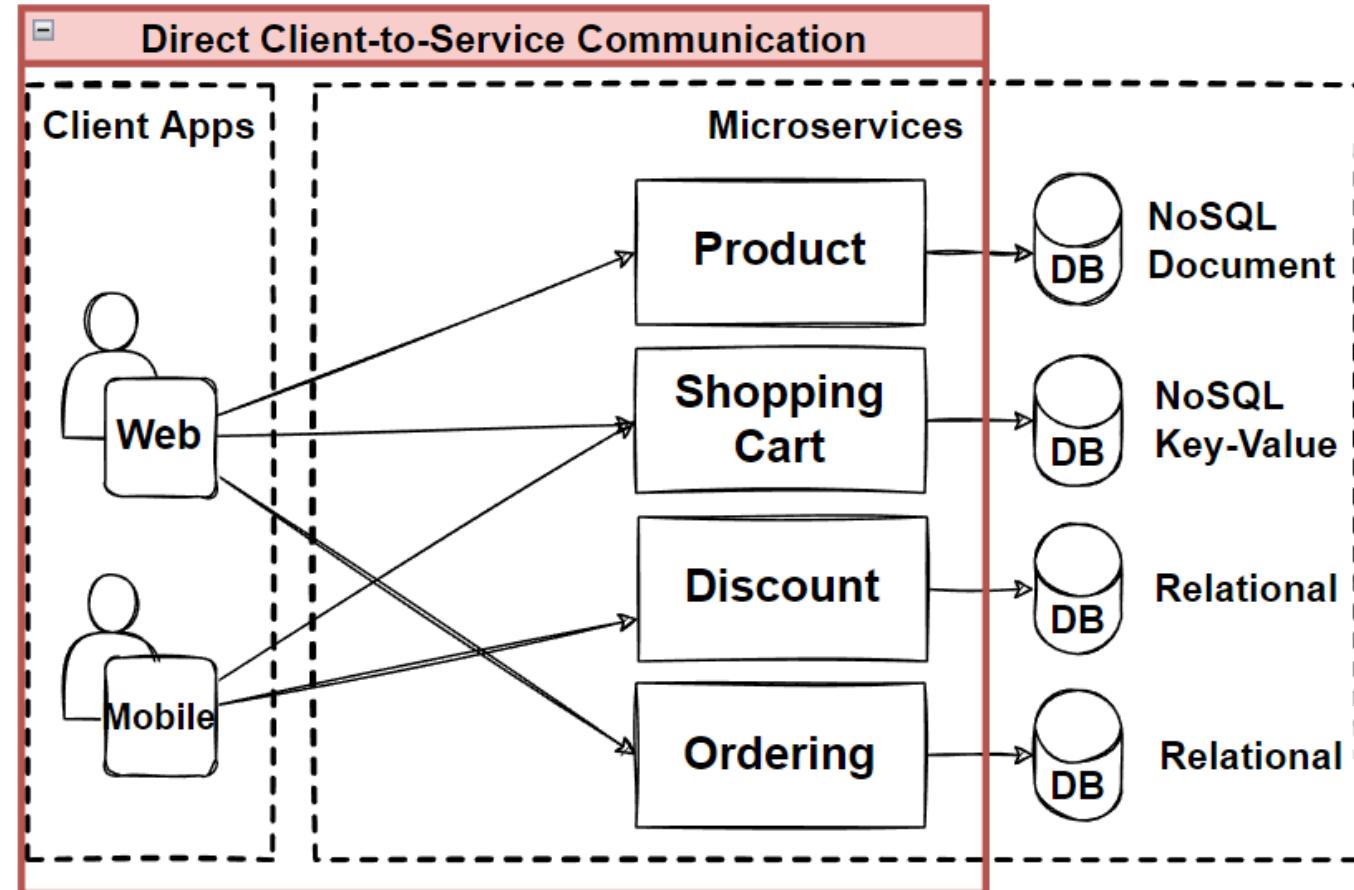
- Direct Client-to-Service Communication
- Cause to chatty calls from client to service
- Hard to manage invocations from client app.

Solutions

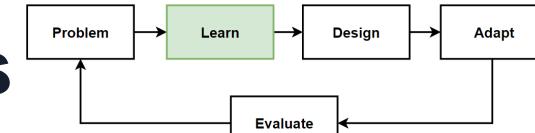
- Well-defined API Design
- Microservices Communication Patterns

Steps

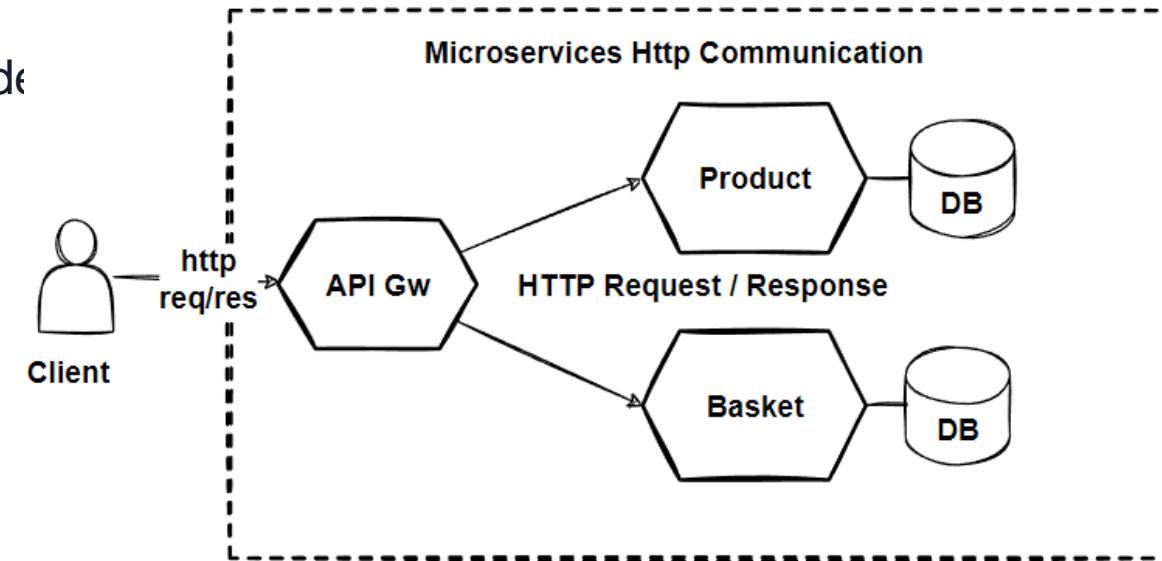
- Microservice Communications and Types
- Well-defined RESTful API Design for services
- Microservices Communication Patterns (API Gateway, BFF, Publish/Subscribe..)



Communications Between Monolithic to Microservices

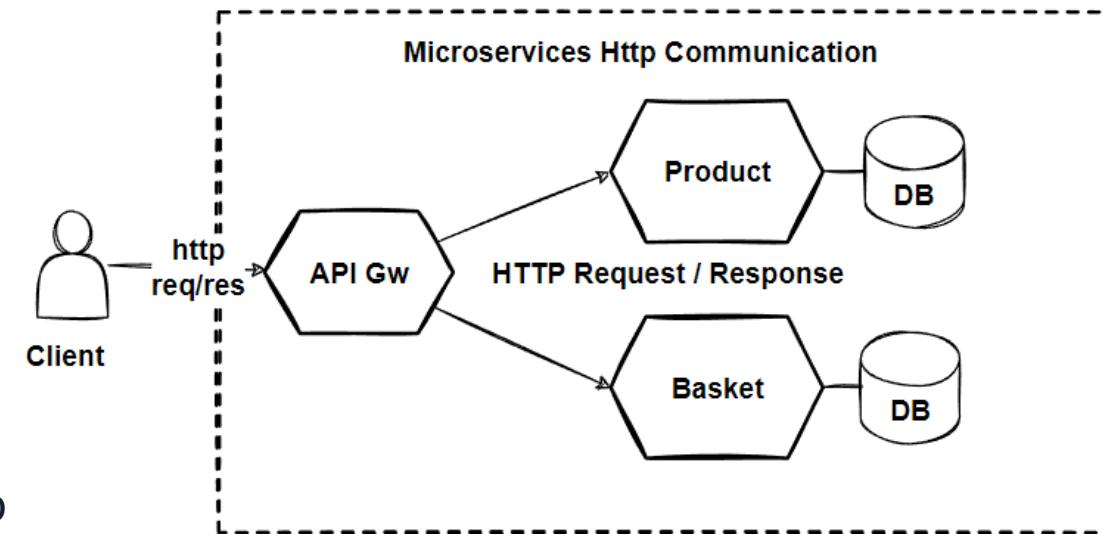
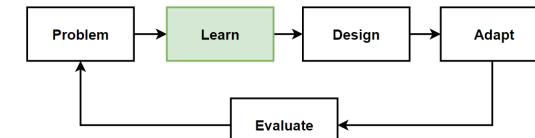


- The communication in **Monolithic** applications are **inter-process** communication.
- Working on **single process** that invoke one to another by using **method calls**. Create class and call the method inside of target module. All running the **same process**.
- The biggest challenge when moving to **microservices-based** application is **changing the communication mechanism**.
- Microservices are **distributed** and microservices communicate with each other by **inter-service communication** on network level.
- Services must interact using an **inter-service communication** protocols like **HTTP**, **gRPC** or message brokers **AMQP** protocol.

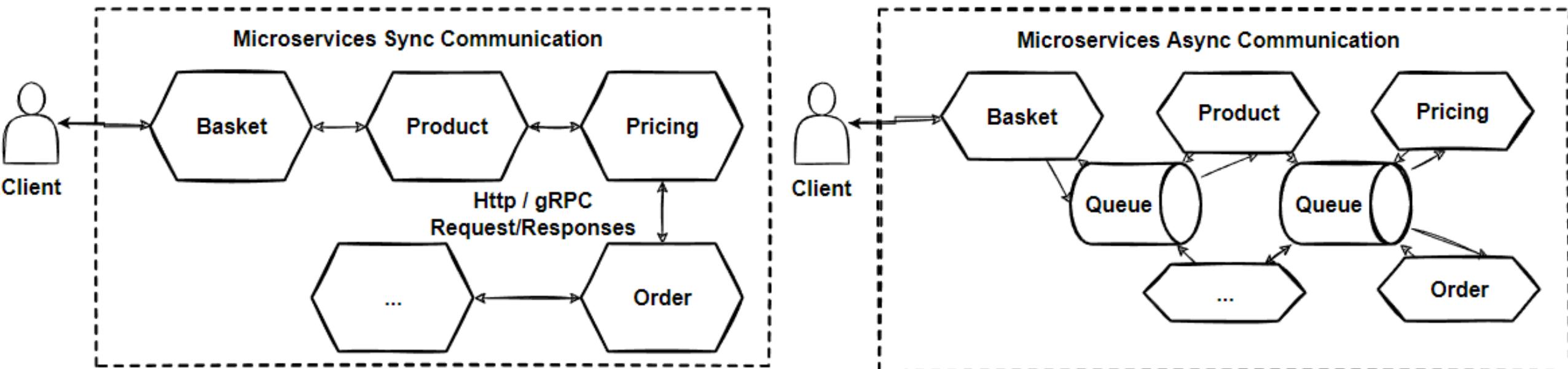
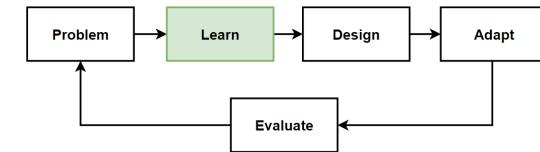


Microservices Communications

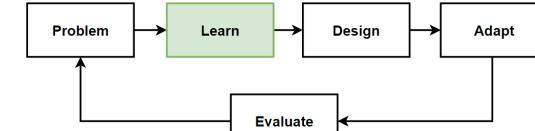
- **Isolate the business** into microservices as much as possible.
- Use **asynchronous communication** between the internal microservices as much as possible.
- Create **well-defined APIs** for **inter-service** communications.
- Monolithic inter-process method calls becomes **well-defined APIs** into Microservices.
- Group some operations and **expose aggregated APIs** that cover several calls from multiple sources.
- **Smart endpoints and dumb pipes:** microservices loosely coupling and expose endpoints with RESTful APIs in order to provide end-to-end use cases.
- Microservices communication types:
 - **Synchronous communications:** Request/Response
 - **Asynchronous communications:** Message broker event buses



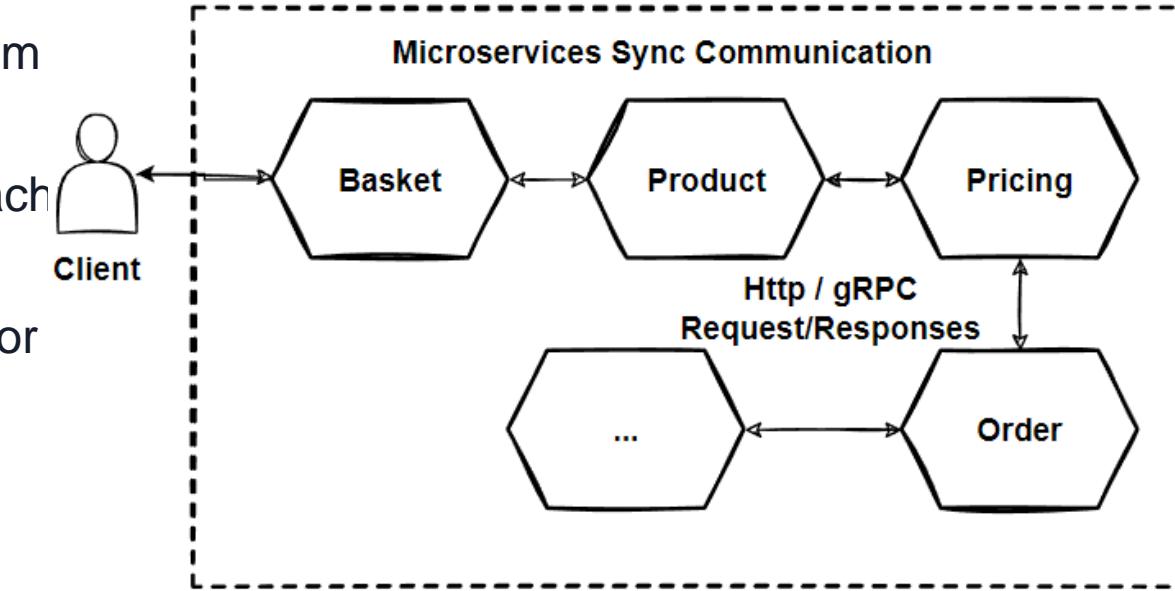
Microservices Communication Types - Sync or Async



Microservices Synchronous Communication

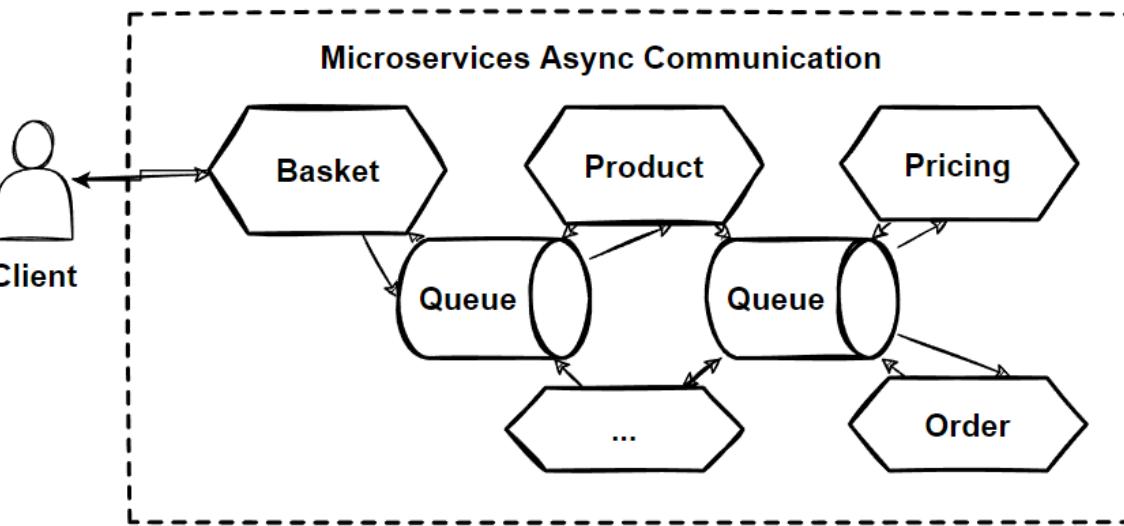


- **Synchronous communication** is using **HTTP** or **gRPC** protocol for returning synchronous response.
- The client **sends a request** and **waits for a response** from the service.
- The client code **block their thread**, until the response reaches from the server.
- The synchronous communication protocols can be **HTTP** or **HTTPS**.
- The client sends a request with using **http protocols** and waits for a response from the service.
- The client **call the server** and **block** client their operations.
- The client code will **continue** its task when it **receives** the HTTP server **response**.



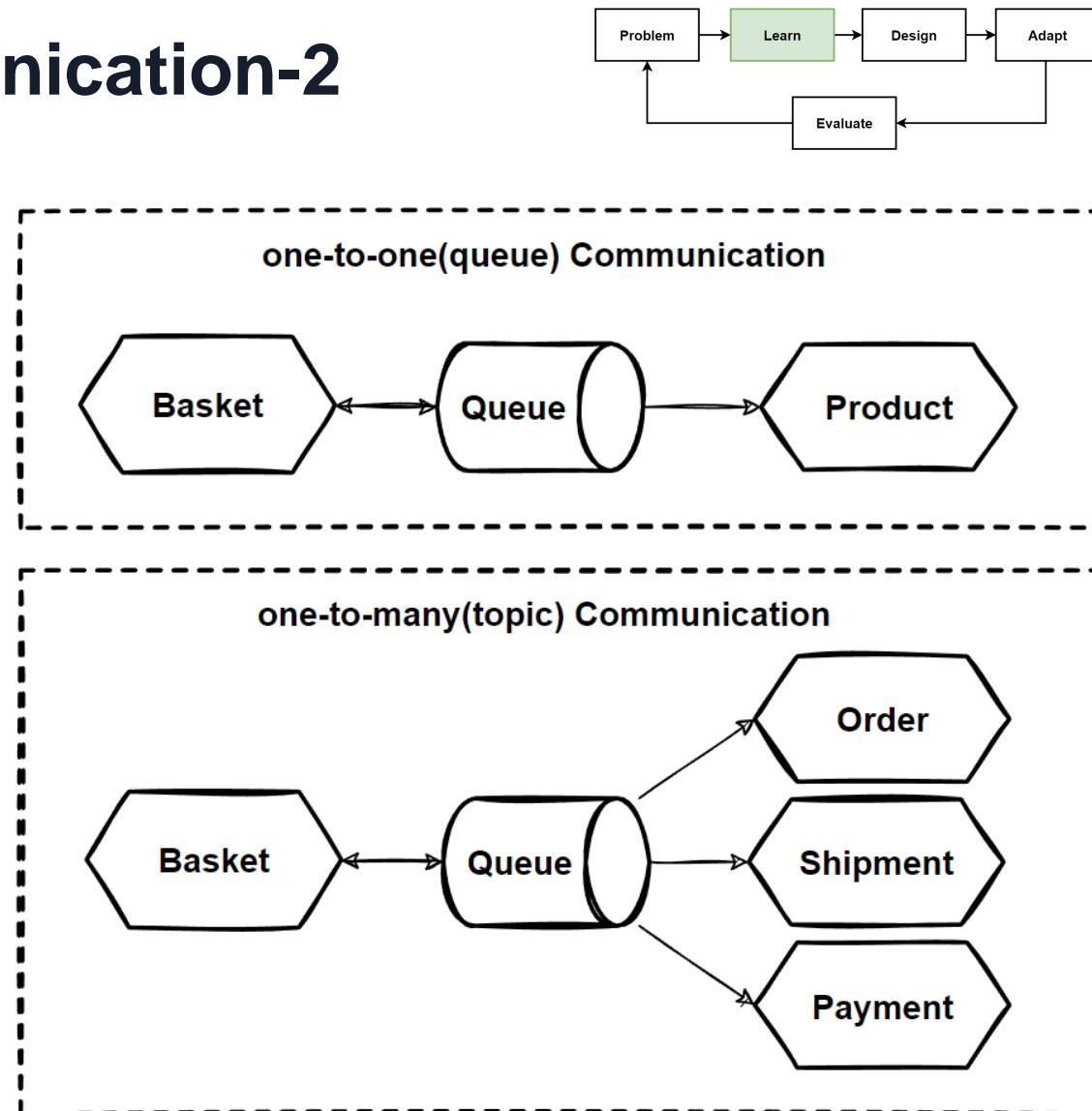
Microservices Asynchronous Communication

- The client sends a request but it **doesn't wait for a response** from the service.
- The client **should not have blocked** a thread while waiting for a response.
- **AMQP** (Advanced Message Queuing Protocol)
- Using AMQP protocols, the client **sends the message** with using message broker systems like **Kafka** and **RabbitMQ** queue.
- The message **producer does not wait** for a **response**.
- Message consume from the **subscriber** systems in async way, and no one waiting for response **suddenly**.
- **Asynchronous** communication also divided by 2:
 - **one-to-one(queue)**
 - **one-to-many (topic)**

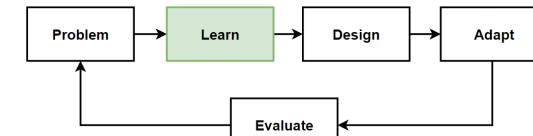


Microservices Asynchronous Communication-2

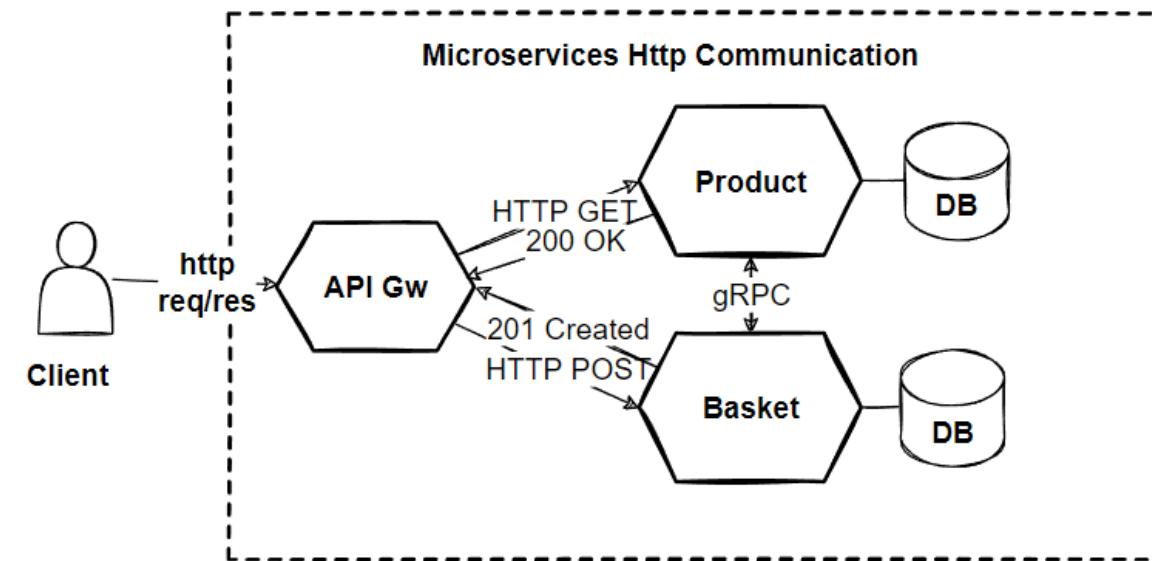
- **one-to-one(queue) implementation**
one-to-one(queue) implementation there is a single producer and single receiver.
- Command Patterns offers to receive one queue object and after that execute the command with incoming message.
This process restarts with receiving new command queue item.
- **one-to-many (topic) implementation**
In one-to-many (topic) implementation has Multiple receivers. Each request can be processed by zero to multiple receivers.
- Event-bus or message broker system is **publishing events** between multiple microservices and communication provide with subscribing these events in an async way.
- **Publish/subscribe** mechanism used in Event-driven microservices architecture.



Microservices Communication Styles

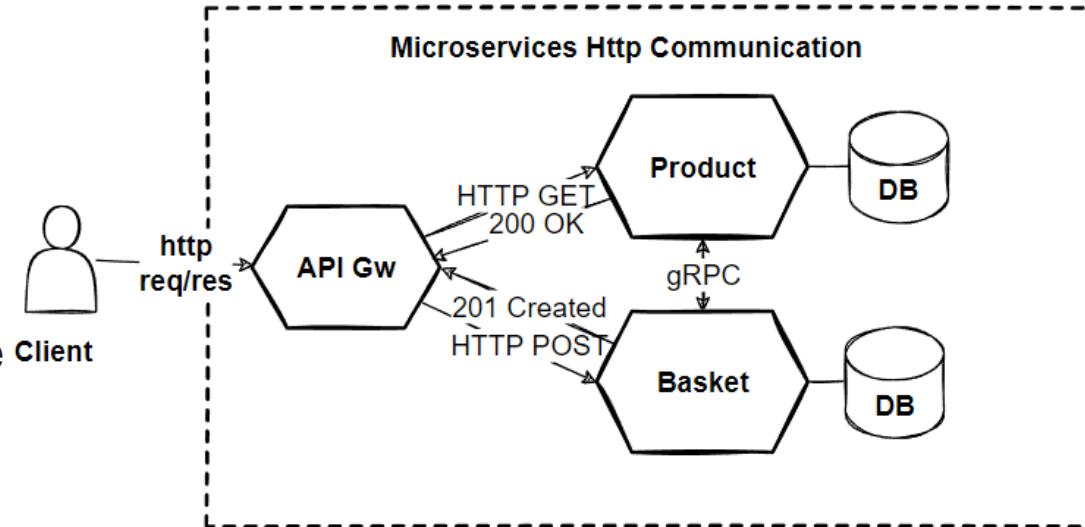
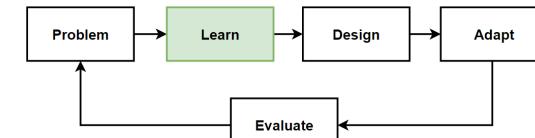


- **Request/response communication** with HTTP and REST Protocol (extends gRPC and GraphQL)
- **Push and real-time communication** based on HTTP, WebSocket Protocol
- **Pull communication** based on HTTP and AMQP (short polling - long polling)
- **Event-Driven communication** with Publish/Subscribe Model

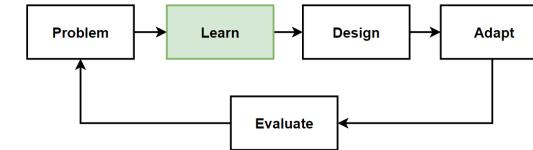


Request/response communication

- When using **synchronous** request/response-based communication mechanism, use **HTTP** and **REST** protocols that are the most common protocols.
- **Expose APIs** from our microservices using the HTTP and REST protocols.
- **REST HTTP calls** using HTTP verbs like GET, POST, and PUT.
- If our communication held between internal microservices, use **gRPC protocol** communication mechanisms to provide high performance and low latency.
- **Use GraphQL** instead of the REST APIs when performing Request/response communication.
- With GraphQL, we can define the **structure of the data** required and get 1 response in 1 request with more flexibility and efficiency way to get whole data.



Push/pull communication



- **Push and real-time communication based on HTTP, WebSocket Protocol**

Use case about real-time and one-to-many communication like chat application, use Push Model with HTTP and WebSocket Protocols.

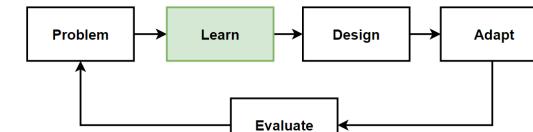
- Build real-time two-way communication applications, such as chat apps and streaming dashboards like the score of a sports game, with WebSocket APIs.
- The client and the server can both send messages to each other at any time. Backend servers can easily push data to connected users and devices.

- **Pull communication based on HTTP and AMQP (short polling - long polling)**

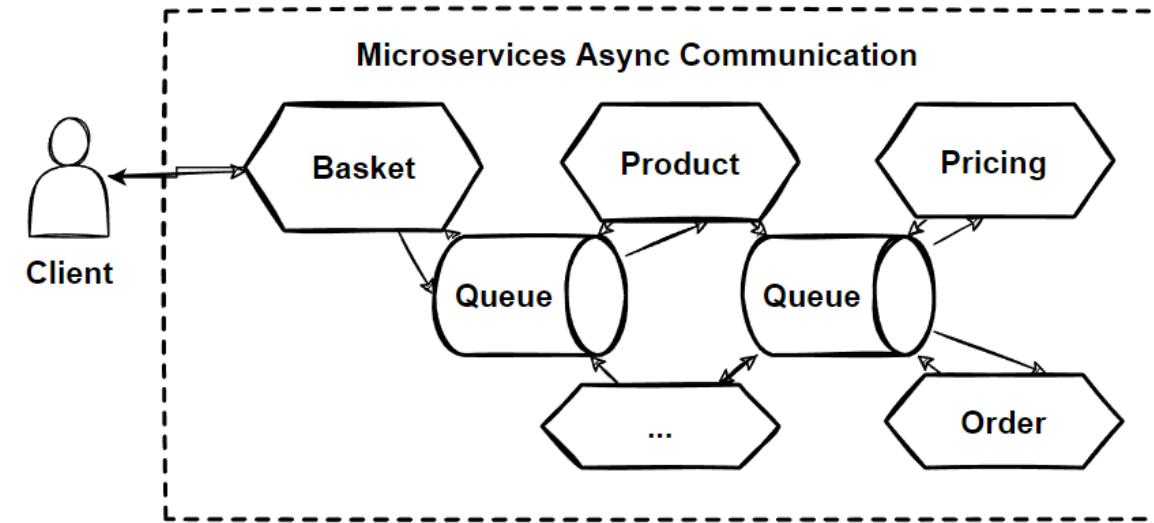
Also called "Polling" and it's basically the same as refreshing your mail inbox every 5 minutes to check for new mail. It is a call and ask model.

- This model is become a waste of bandwidth if there are no new messages and responses comes from the server.
- Opening and closing connections is expensive. And we can say that this model doesn't scale well.
- Typically have limits like on Twitter on how often they allow you to call their API.

Event-Driven communication with Publish/Subscribe

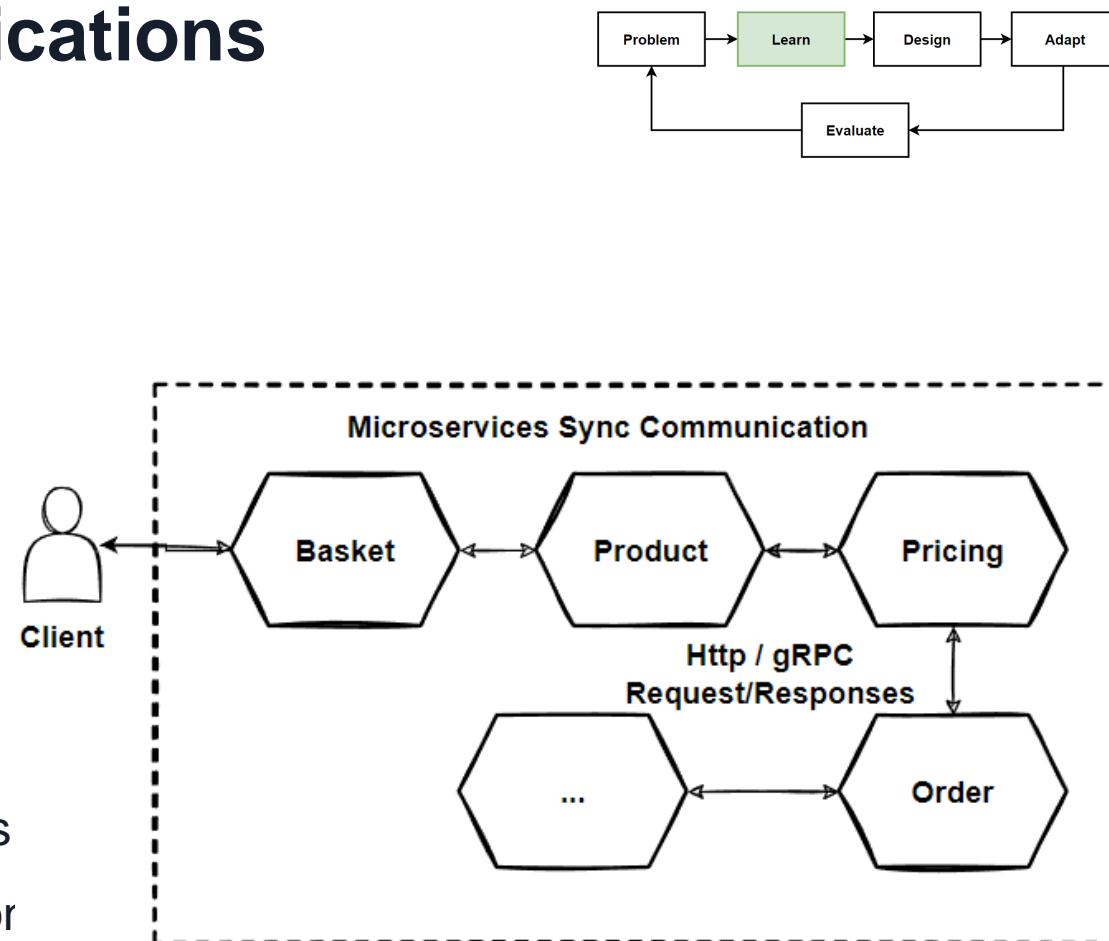


- Microservices don't call each other, instead, they created **events** and **consume events** from message broker systems in an async way.
- **AMQP** (Advanced Message Queuing Protocol)
- **Publish/Subscribe pattern** with events
- Using AMQP protocols, the client sends the message with using message broker systems like **Kafka** and **RabbitMQ** queue.
- The producer service of the events **doesn't know** about its consumer services. the consumers also don't necessarily know about the producer.
- **Decouple services** and build loosely coupled microservices.
- No clear central place or orchestrator, This increase **complexity** of architecture.

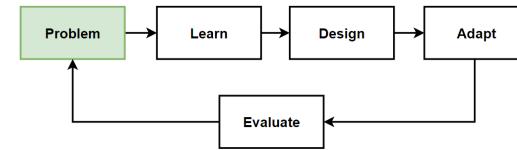


Microservices Synchronous Communications and Best Practices

- The client **sends a request** with using http protocols and **waits for a response** from the service.
- The synchronous communication protocols can be **HTTP** or **HTTPS**.
- **Request/response communication** with HTTP and REST Protocol (extends gRPC and GraphQL)
- **REST HTTP APIs** when exposing from microservices
- **gRPC APIs** when communicate internal microservices
- **GraphQL APIs** when structured flexible data in microservices
- **WebSocket APIs** when real-time bi-directional communication
- How can we design and exposing APIs with HTTP protocols for our microservices ?



Problem: Direct Client-to-Service Communication



Problems

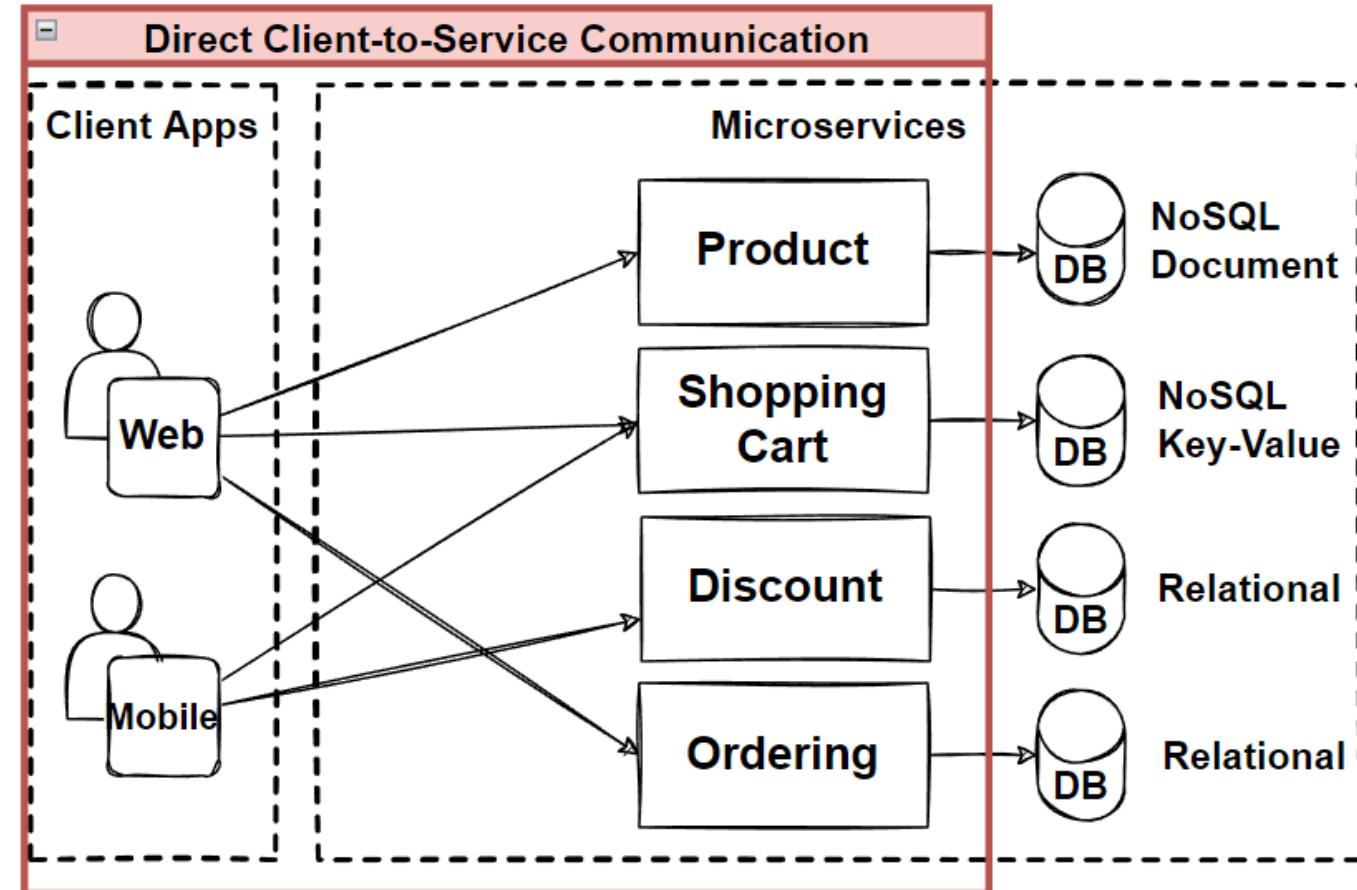
- Direct Client-to-Service Communication
- Cause to chatty calls from client to service
- Hard to manage invocations from client app.

Solutions

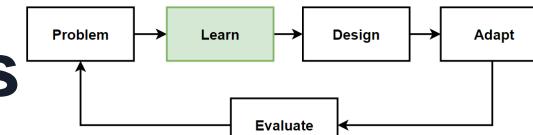
- Well-defined API Design
- Microservices Communication Patterns

Steps

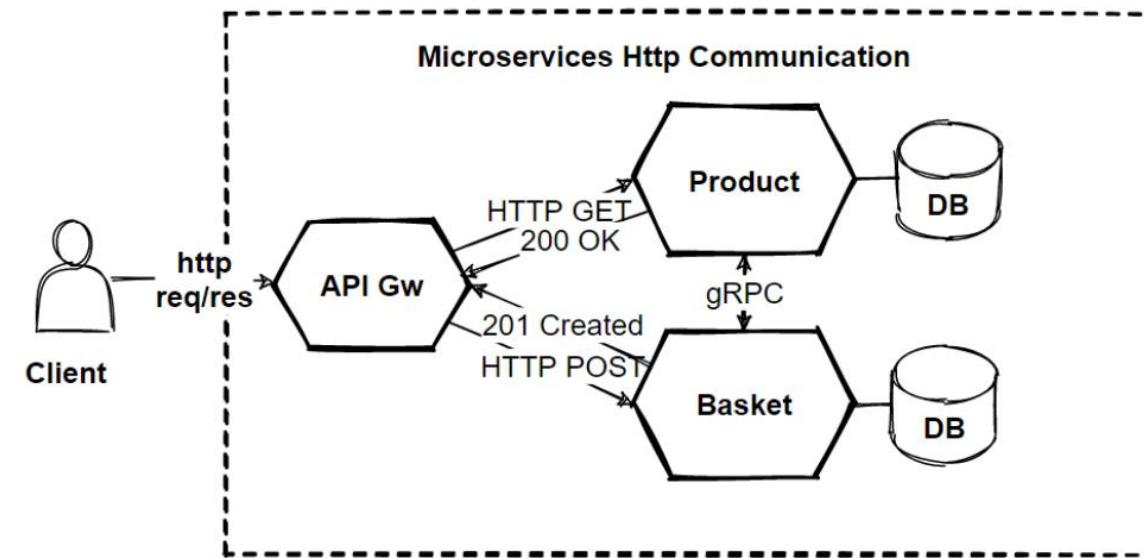
- Microservice Communications and Types
- Well-defined RESTful API Design for services
- Microservices Communication Patterns (API Gateway, BFF, Publish/Subscribe..)



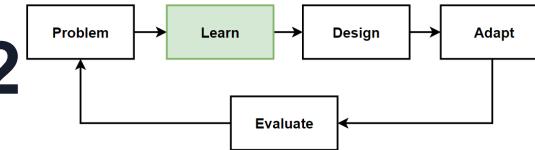
Designing HTTP based RESTful APIs for Microservices



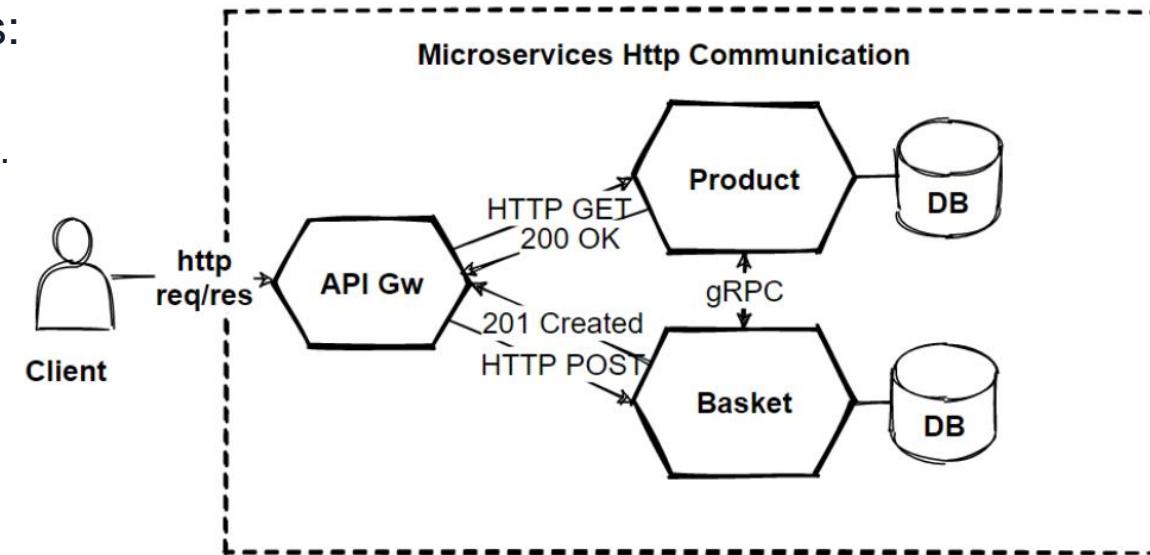
- When making **request/response communication**, we should use **REST** when designing our APIs. Its also called **Restful APIs**.
- **REST** approach is following the **HTTP protocol**, and implementing HTTP verbs like GET, POST, and PUT.
- **REST** is the most commonly used architectural communication approach when creating **APIs** for our **microservices**.
 - Java and Sprint Boot framework
 - C# with ASP.NET Core Web API
 - Python with Flask Web API framework
- How to **design** our **APIs** for **microservices** ?



Designing HTTP based RESTful APIs for Microservice2

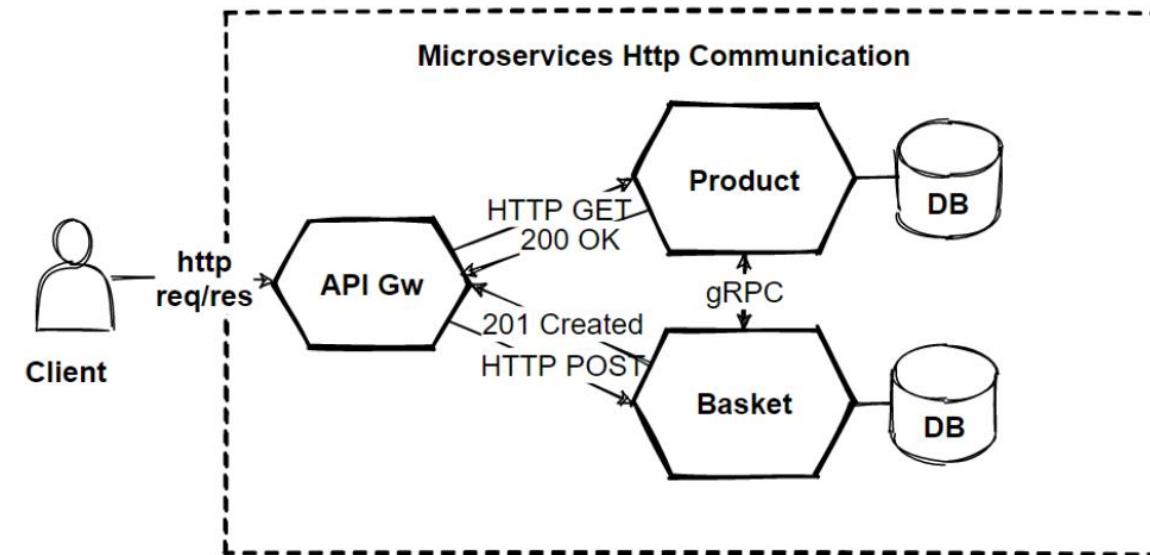
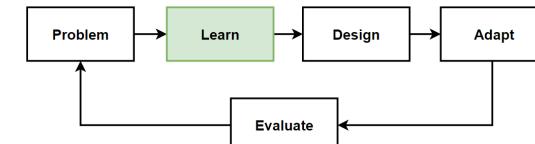


- **Well-defined API** design is very important in a microservices architecture, communication happens API calls.
- Designed APIs should be **efficient** and **not to be chatty communications**. APIs must have well-defined documented and versioning.
- There are 2 type APIs sync communication in microservices:
 - Public APIs: API calls from the client apps.
 - Backend APIs: inter-service communication between backend services.
- **Public APIs**
Use RESTful APIs over HTTP protocol. RESTful APIs use JSON payloads for request-response, that easy to check payloads and easy agreement with clients.
- **Backend APIs**
Inter-service communication can result in a lot of network traffic. serialization speed and payload size become more important. Using gRPC is mandatory for increase network performance.



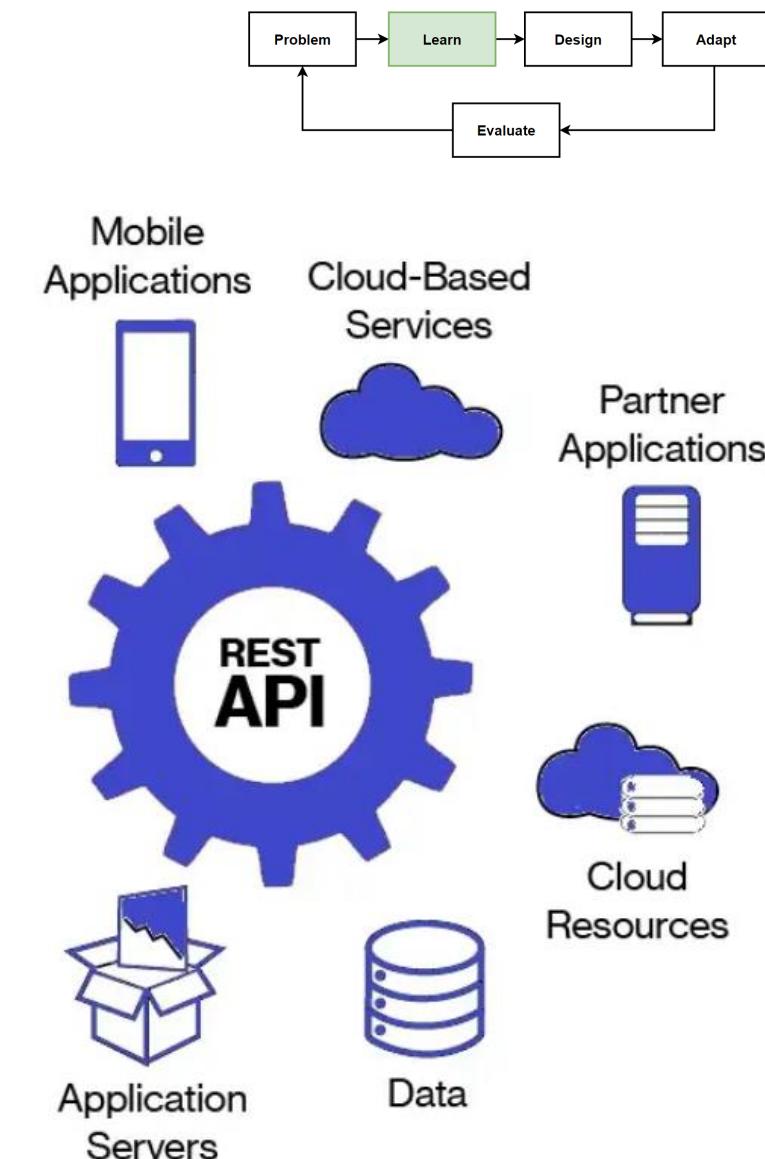
Comparison with REST and gRPC

- **REST** is using HTTP protocol, and request-response structured JSON objects.
- API interfaces design based on HTTP verbs like GET-PUT-POST and DELETE.
- **gRPC** is basically Remote Procedure Call, that basically invoke external system method over the binary network protocols.
- Payloads are **not readable** but its **faster** than **REST APIs**.
- 2 Main Approaches for Public and Backend APIs:
 - RESTful API Design over HTTP using JSON
 - gRPC binary protocol API Design



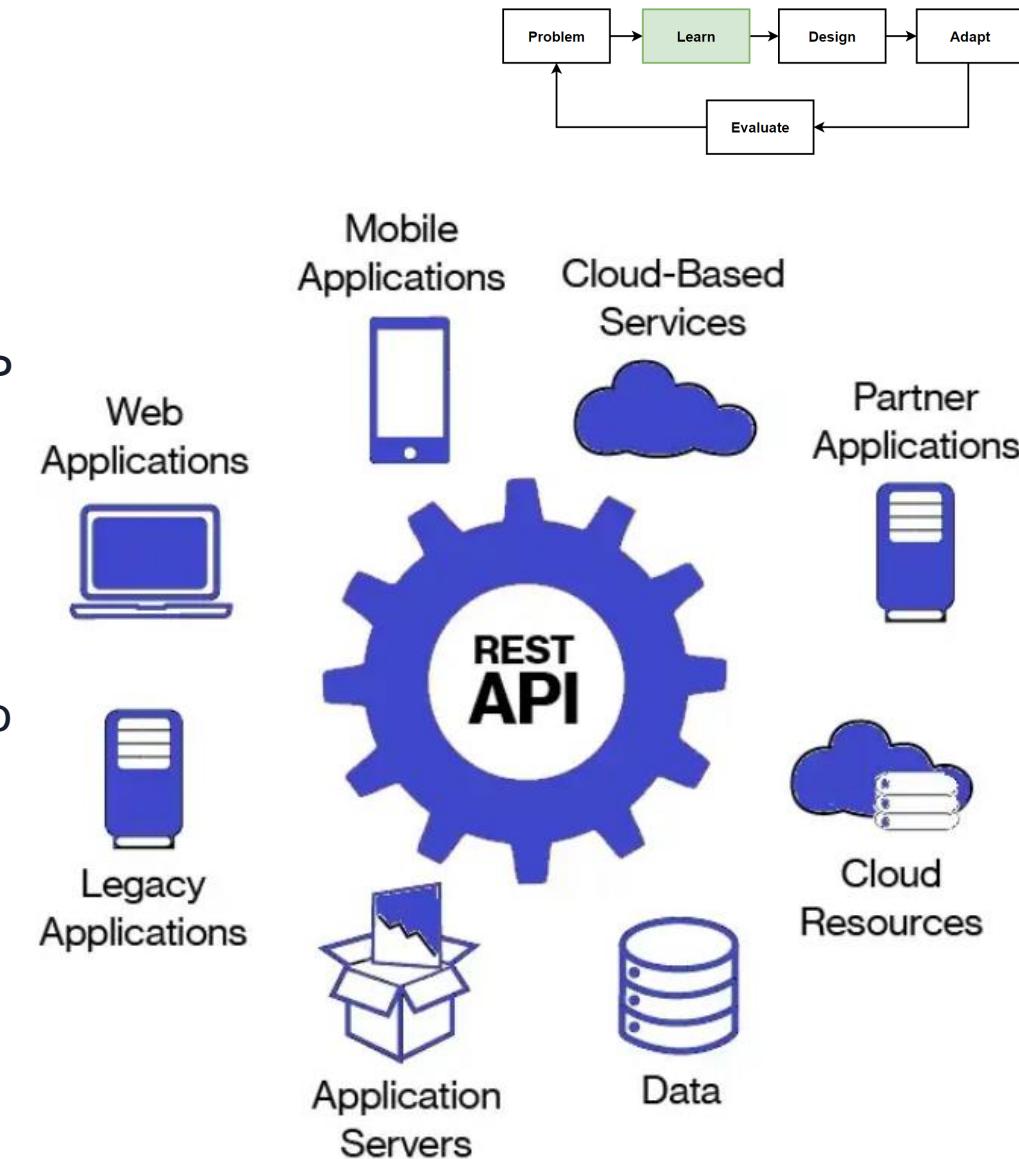
RESTful API design for Microservices

- **RESTful services** are widely used in modern Web architectures.
- It is lightweight, extensible and simple easy develop services.
- **What is REST ?**
REST (Representational State Transfer) is a service structure that enables easy and fast communication between client and server. Roy Fielding introduced and developed REST in his doctoral thesis in 2000.
- REST allows applications to communicate with each other by **carrying JSON data** between the client and server.
- **Characteristics of the REST Architecture:**
 - Stateless
 - Uniform Interface
 - Cacheable
 - Client-Server
 - Layered System
 - Code on Demand

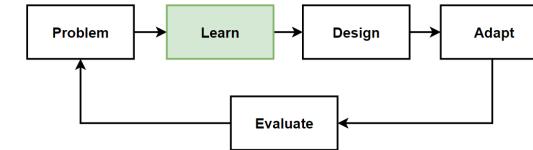


What is RESTful APIs?

- Web services that use REST architecture are called **RESTful services**.
- **RESTful systems** communicates over **HTTP protocol** with HTTP methods used by Web Browsers to transfer pages.
- **Richardson Maturity Model**
Leonard Richardson proposed the Richardson Maturity Model for web APIs:
 - **Level 0:** Define one URI, and all operations are POST requests to this URI.
 - **Level 1:** Create separate URIs for individual resources.
 - **Level 2:** Use HTTP methods to define operations on resources.
 - **Level 3:** Use hypermedia

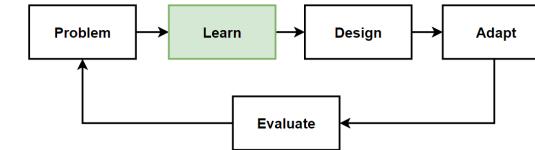


HTTP Methods

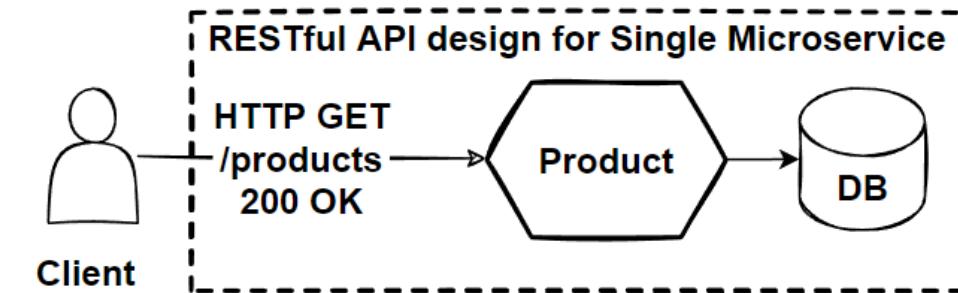


HTTP Methods	
GET	Retrieve information
HEAD	Retrieve Resource Header
POST	Submit Data to the server
PUT	Save an object at the location
DELETE	Delete the object at the location
PATCH	Update single piece of data

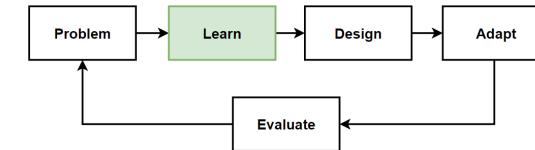
RESTful API design for Single Microservice



- **Focus on the business entities** that we expose APIs for our application.
Organize our resources according to business entities and expose via APIs.
- Design RESTful APIs for single **Product Microservices**.
- **Rule1- REST APIs are designed around "resources"**
The best practice is the resource URLs should be based on nouns = the resource and not verbs. REST API Url: <https://e-shop.com/products>
 - https://e-shop.com/products -- **Correct**
 - https://e-shop.com/create-product -- **Wrong**
- **Rule2- Every Resource has an identifier**
Unique identifiers for that resource. REST API URL becomes for a particular product: <https://e-shop.com/products/4>
- This will return single product which id = 4.
- API frameworks route requests based on parameterized URI paths, so we can define a route for the path like **/products/{id}**.



RESTful API design for Single Microservice-2



- **Rule3- REST APIs are using JSON**

REST API url has GET request, then it will return this response body as a JSON response:

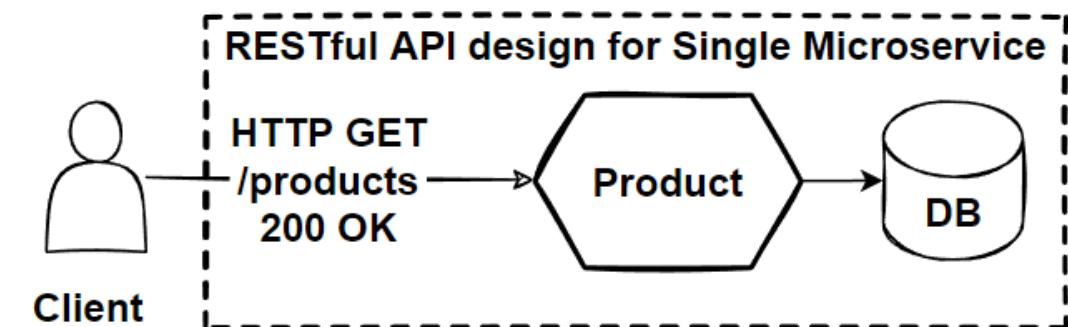
- `{"productId":1,"name":"iPhone", "price":44.90,"category":"Electronics"}`

- **Rule4- REST APIs perform operations on Resources with HTTP verbs:**

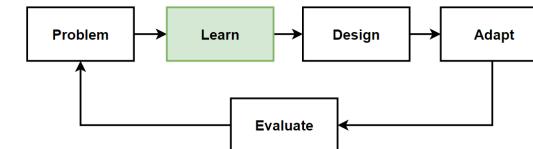
REST APIs built on HTTP protocol and perform operations on Resources with HTTP verbs, these verbs are GET, POST, PUT, PATCH, and DELETE.

- When creating a product, we send HTTP POST request with contains product detail informations and return back to HTTP response that including 200 OK success response.

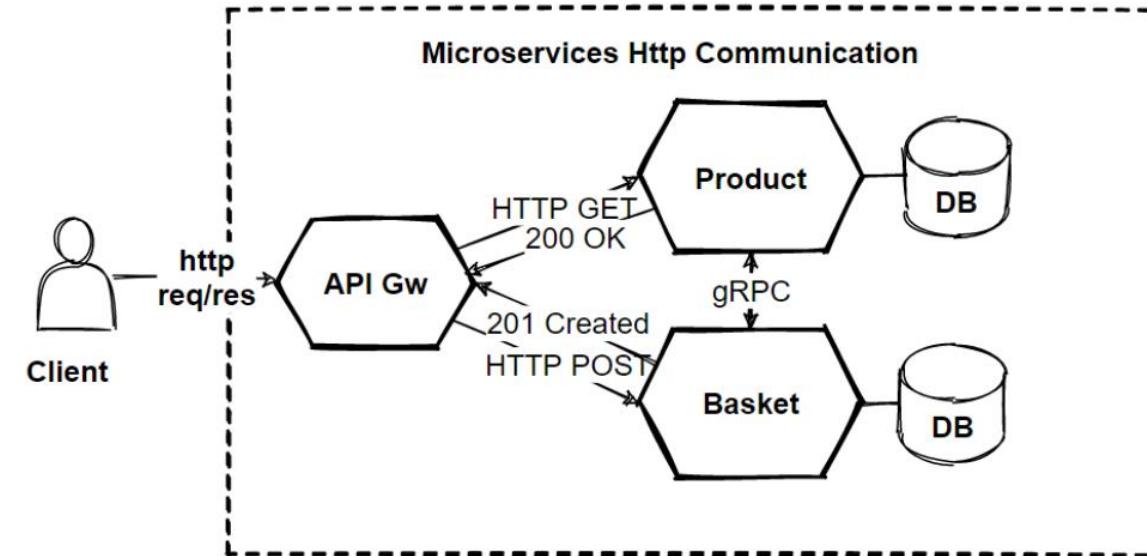
- Request:
- HTTP: POST
- Url: <https://e-shop.com/products>
- Body: `{"productId":1,"name":"iPhone"}`
- Response: 201 Created



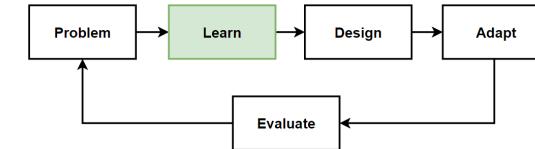
RESTful API design for E-Commerce Microservices



- RESTful API design for E-Commerce Microservices: Customer, Order and Product resources.
- **How should we design Rest API when you required to get customer orders ?**
 - URL: <https://e-shop.com/customers/6/orders>
- I want to retrieve products into order number 22 which ordered by customer 6:
 - URL: <https://e-shop.com/customers/6/orders/22/products> --**WRONG**
- This level of complexity can be difficult to maintain.
- Instead, we should keep URIs relatively simple.
- Resource URIs should not be more complex than "**collection/item/collection**" path.
- First we should get Customer orders with this API:
 - URL: <https://e-shop.com/customers/6/orders>
- After that we should locate the Order no 22 and get all products:
 - URL: <https://e-shop.com/orders/22/products>

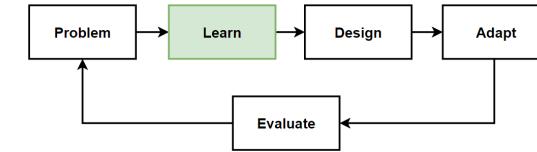


RESTful API design with HTTP Methods

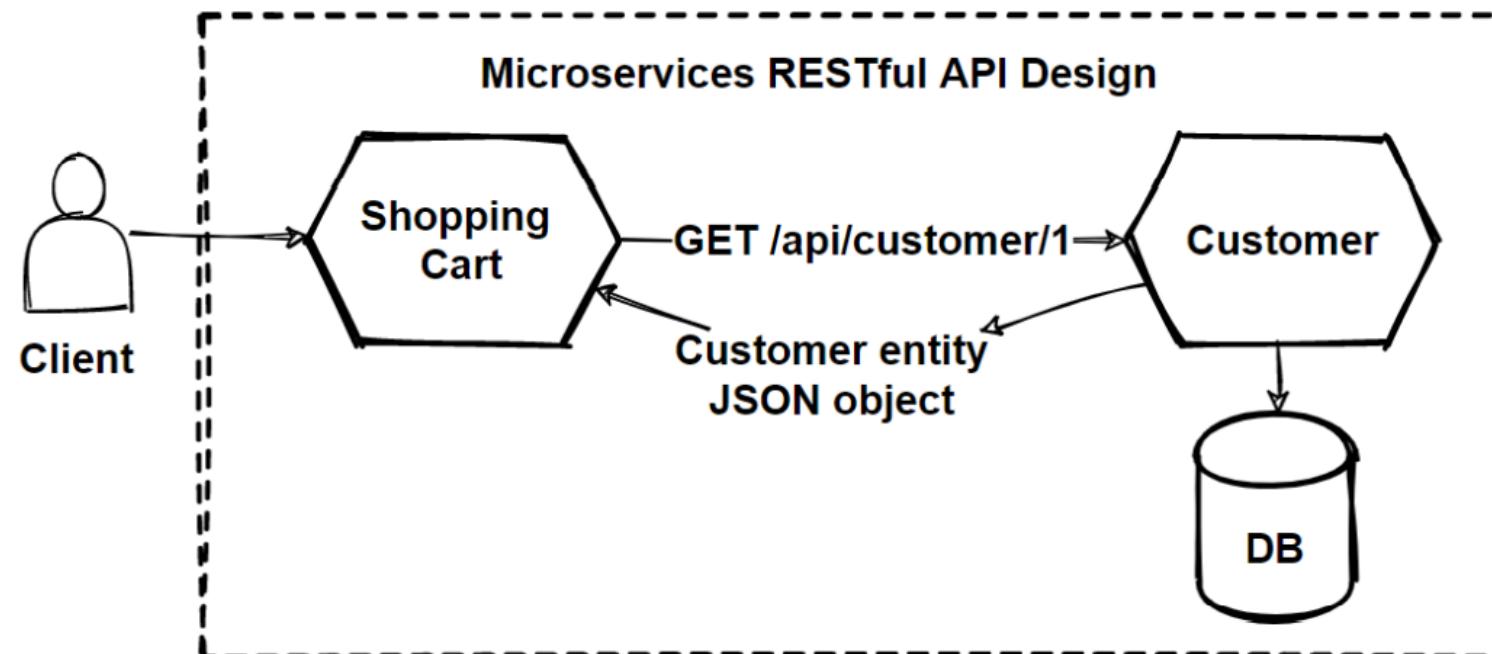


Resource	POST	GET	PUT	DELETE
/customers	Create new customer	Retrieve all customers	Update all customers	Remove all customers
/customers/1	N/A	Retrieve data for customer 1	Update data for customer 1 if exist	Remove customer 1
/customers/1/orders	Create new order for customer 1	Retrieve all orders for customer 1	Update all orders for customer 1	Remove all orders customer 1

RESTful API design for Relational Data

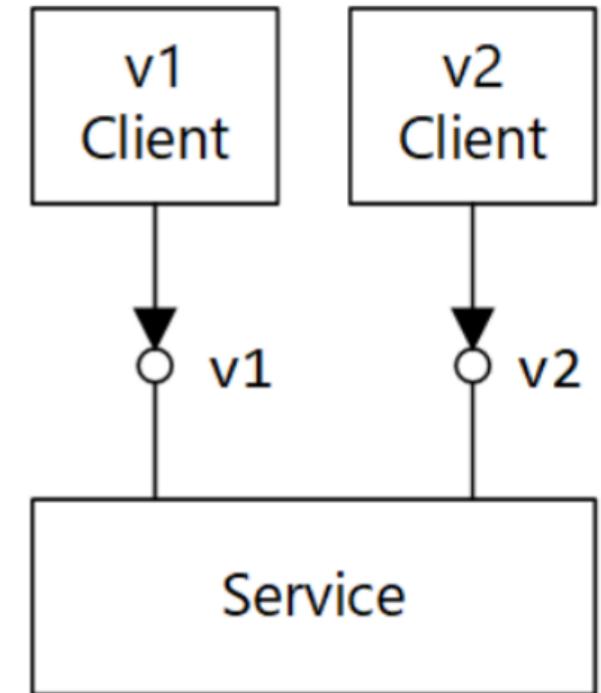
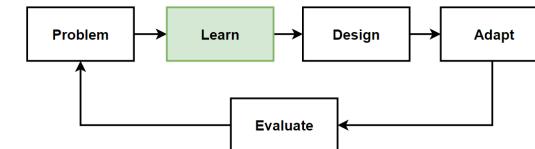


- Microservices don't share the same code base and **don't share data stores**. They communicate **through APIs** for data operations.
- Shopping Cart service **requests information** about a customer from the Customer service.
- Customer service **retrieve** data with using Repository classes and return **Customer entity model** as a **JSON object** in an HTTP response.



API versioning in Microservices RESTful APIs

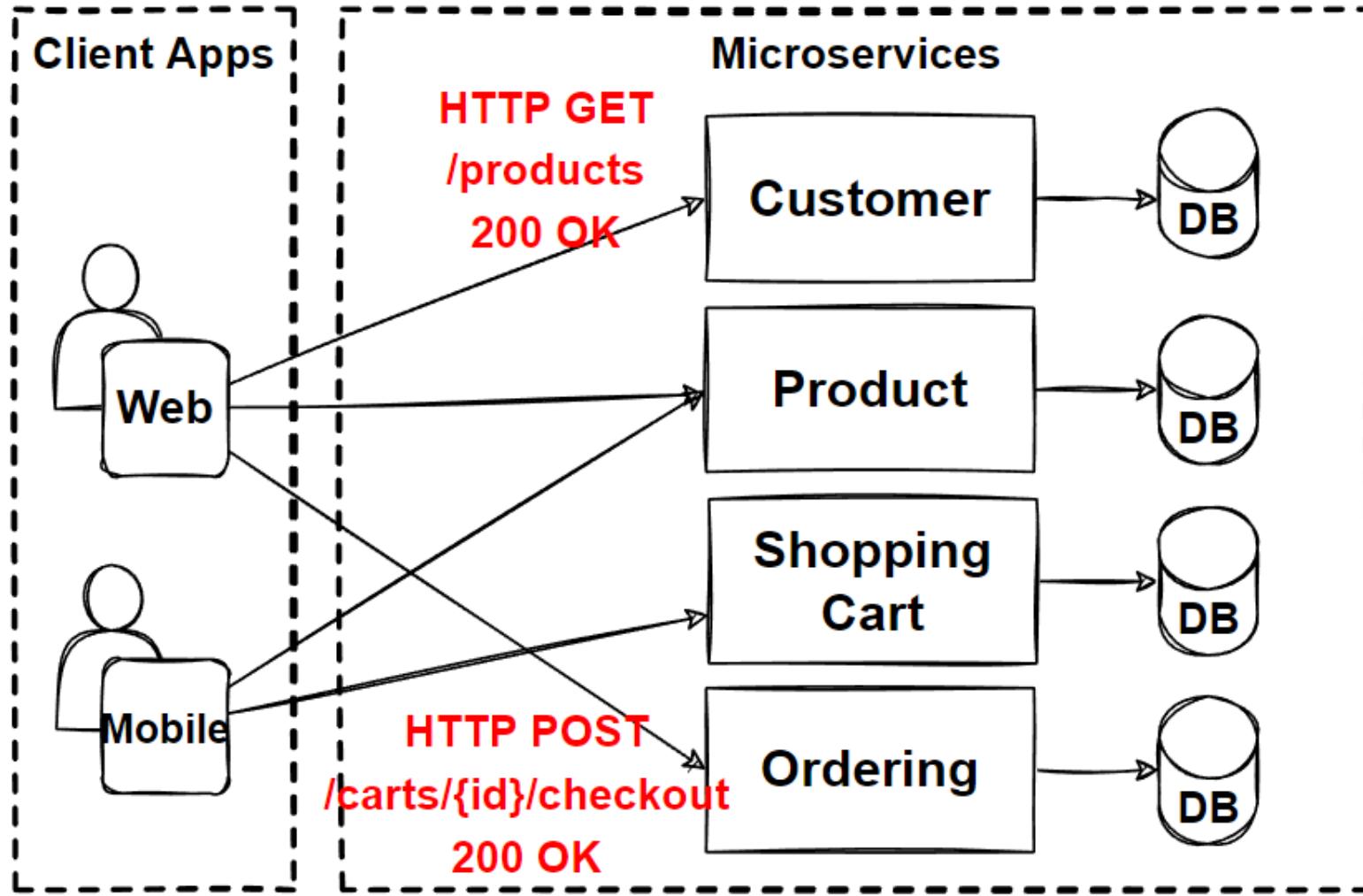
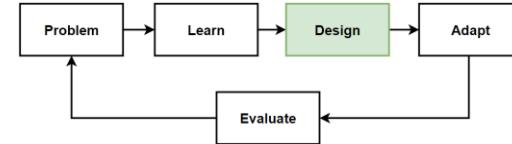
- If an **API changes**, there may broke some communications which depending on that API from other microservices.
- Services may required to add **new features** or **bug fixes** that require changing an existing API.
- API changes should be **backward compatible** and **not break** any communications.
- Best practice for the **roll-out** or **canary deployments** on **Kubernetes**.
- If your microservices has one more pods, new version of API deployments handle one by one and if there is a problem its **easy to rollback** to previous version.
- Microservices should **supports two versions** with following backward compatible.



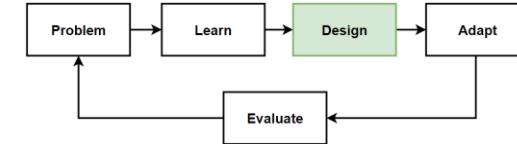
Before Design – What we have in our design toolbox ?

Architectures	Patterns&Principles	Non-FR	FR
<ul style="list-style-type: none">• Microservices Architecture	<ul style="list-style-type: none">• The Database-per-Service Pattern• Polygot Persistence• Decompose services by scalability• The Scale Cube• Microservices Decomposition Pattern• Microservices Communications<ul style="list-style-type: none">• HTTP Based RESTful API design	<ul style="list-style-type: none">• High Scalability• High Availability• Millions of Concurrent User• Independent Deployable• Technology agnostic• Data isolation• Resilience and Fault isolation	<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history

Design: Microservices Architecture with RESTful APIs

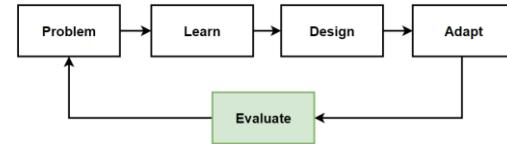


Design: Microservices Architecture with RESTful APIs



Microservices	Resource	HTTP Methods	REST APIs
Customer	/customers	HTTP GET HTTP POST HTTP DELETE	GET /customers GET /customers/{id} POST /customers DELETE /customers/{id}
Product	/products	HTTP GET HTTP POST HTTP DELETE	GET /products GET /products/{id} POST /products DELETE /products/{id}
Shopping Cart	/carts	HTTP GET HTTP POST HTTP DELETE	GET /carts GET /carts/{id} POST /carts/{id}/checkout DELETE /carts/{id}
Ordering	/orders	HTTP GET HTTP POST HTTP	GET /orders GET /orders/{id}

Evaluate: Microservice Architecture with RESTful APIs

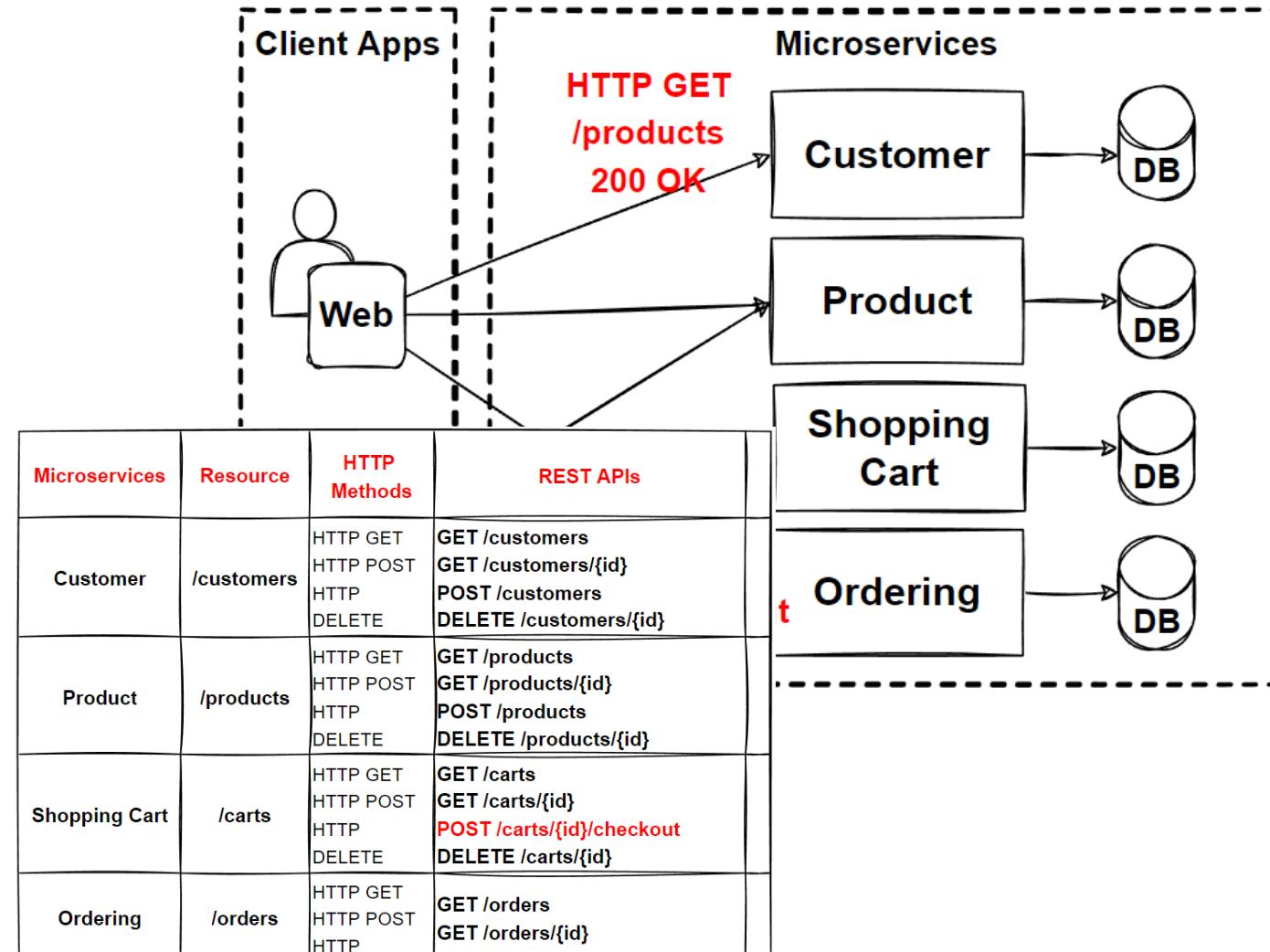


Benefits

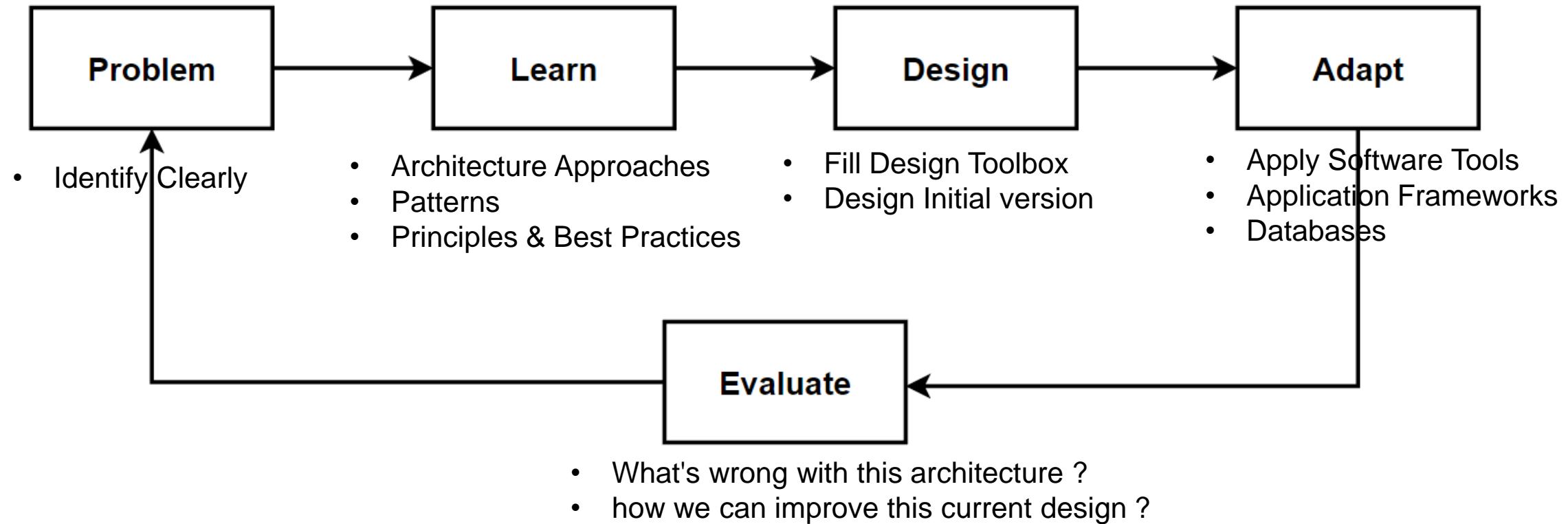
- Request can simply sent using browsers.
- HTTP protocol
- HTTP GET has organic caching options
- JSON representative request-responses

Drawbacks

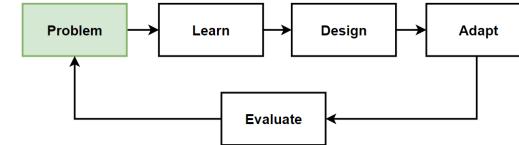
- Send multiple request to get relational data
- Chatty communication when enriching data



Way of Learning – The Course Flow



Problem: Multiple Request for Retrieve Relational Data

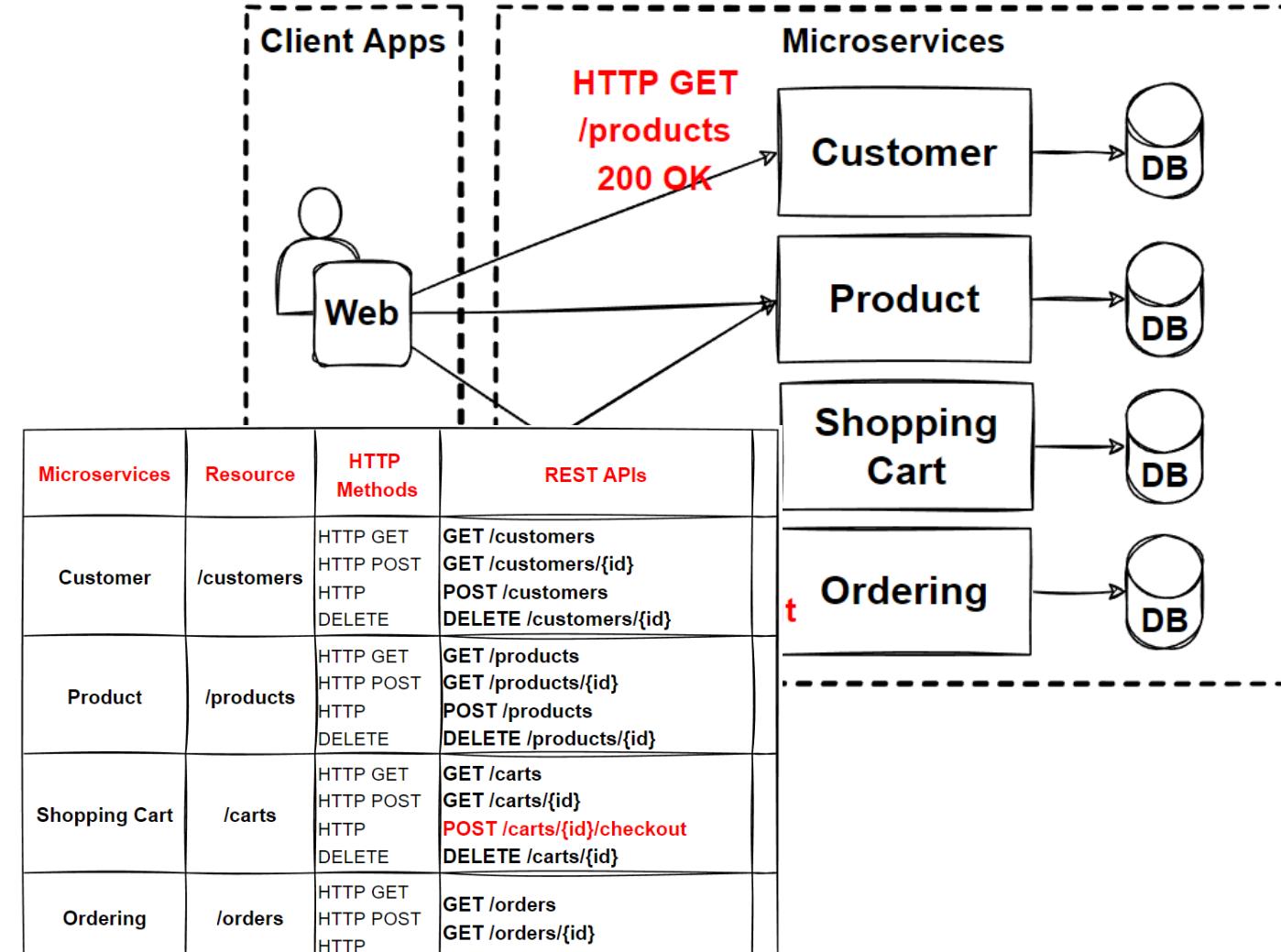


Problems

- Business teams want to see relational data on screen. N+1 problem.
- Customer – Orders – Products
- REST: /customer/3/orders/4/products ?
- How we can get Products from Order X for Customer Y ?
- Chatty Calls for enriching data

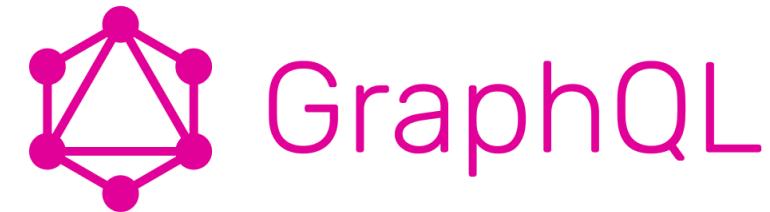
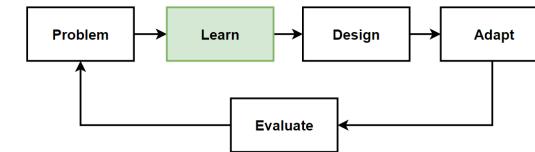
Solutions

- GraphQL API Design
- Structural relational data with querying GraphQL API



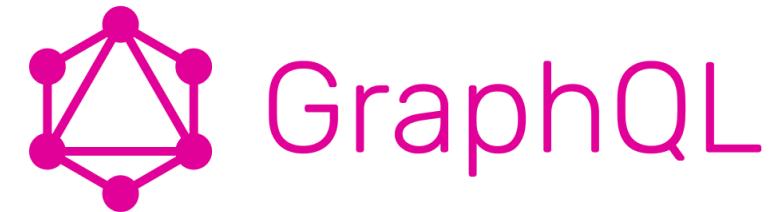
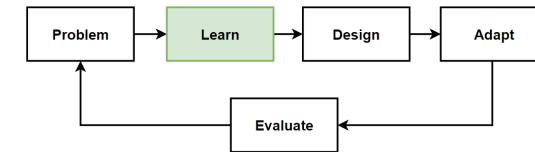
GraphQL: A query language for APIs

- **GraphQL** is a **query and manipulation** language for APIs and a runtime for fulfilling queries with your existing data.
- It was developed internally by **Facebook** in 2012 before being publicly released in 2015.
- It allows **clients to define the structure of the data required**, and the **same structure** of the data is returned from the server.
- It provides a **complete and understandable description** of the data in our API, clients request what they need, **easier to evolve** APIs over time, has powerful developer tools.
- It provides a **flexible syntax** to describe data requirements and interactions.
- It provides to **access many sources** in a single request, **reducing the number of multiple network calls**.
- Developers to ask for **exactly what is needed** and get back **predictable results**.



Characteristics of GraphQL

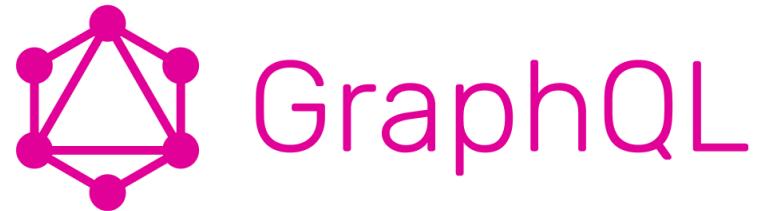
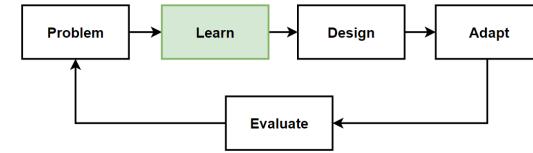
- **Ask for what you need, get exactly that**
Send a GraphQL query to your API and get exactly what you need.
GraphQL queries always return predictable results.
- **Get many resources in a single request**
GraphQL queries can access the properties of one resource and also relations of particular resource references between them.
- GraphQL APIs get all the data your applications needs in a single request.
- **Evolve APIs without versions**
As you know that REST APIs requires versions when change your resource properties. We can add new fields and types to GraphQL API without impacting existing queries.
- By using a single evolving version, GraphQL APIs give apps continuous access to new features.



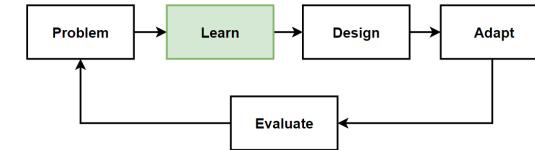
Core Concepts of GraphQL

Schemas, Queries, Mutations and Resolvers

- Developers can use **GraphQL** in order to create a **schema**, which describes all the possible data that clients can query on it.
- **GraphQL** schema is made up of **object types**, that define which kind of object you can request and which fields it has.
- Client can **send queries**, GraphQL validates the queries against the schema then executes the **validated queries**.
- **Resolver** is a function that attaches to fields in a schema. During execution, the resolver is called to produce the value.
- **Mutation** is a GraphQL Operation that allows you to **insert new data** or **modify the existing data** on the server-side.



Example of GraphQL Query



- Simply **send a single query** to the GraphQL server that includes the **concrete data requirements**. The server then responds with a JSON object.
- See example **Order** and **Order Products** list as below:

POST Request:

```
{  
  orders {  
    id  
    productsList {  
      product {  
        name  
        price  
      }  
      quantity  
    }  
    totalAmount  
  }  
}
```

Response:

```
{  
  "data": {  
    "orders": [  
      {  
        "id": 1,  
        "productsList": [  
          {  
            "product": {  
              "name": "orange",  
              "price": 1.5  
            },  
            "quantity": 100  
          }  
        ],  
        "totalAmount": 150  
      }  
    ]  
  }  
}
```

Advantages of GraphQL

- **GraphQL is fast**

GraphQL is way faster than other communication APIs like REST APIs because it reduces multiple calls to get data and provides the ability to query by choosing only the specific fields.

- **Single Request**

GraphQL gets all the data your application needs in a single request. Clients get what they request with no over-fetching.

- **Strongly typed**

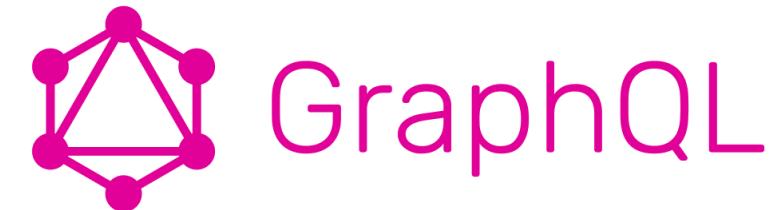
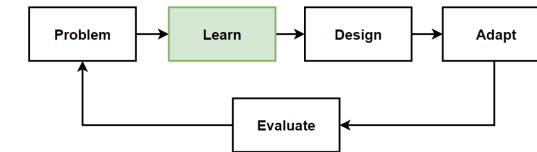
In GraphQL we describe data before querying it. Strongly defined data types reduce miscommunication.

- **Hierarchical Structure**

GraphQL provides a hierarchical structure where relationships between objects are defined in a graphical structure.

- **Evolve API**

GraphQL allows an application API to evolve without breaking existing queries.



Disadvantages of GraphQL

- **GraphQL Query Complexity**

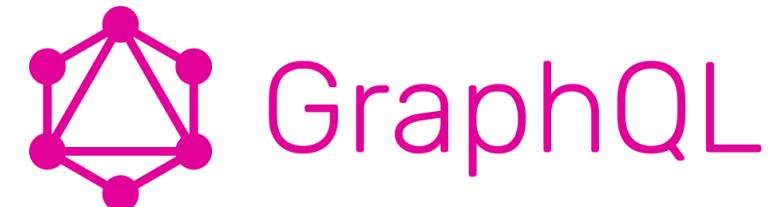
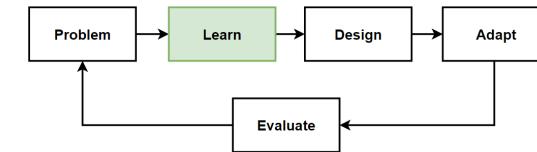
When access multiple fields in one query, it may requests too many nested fields data at a single time that could cause performance problems. Consider maximum query depths, query complexity.

- **GraphQL Caching**

More complicated to implement a simple cache with GraphQL. Complex because each query can be different, even though it operates on the same entity.

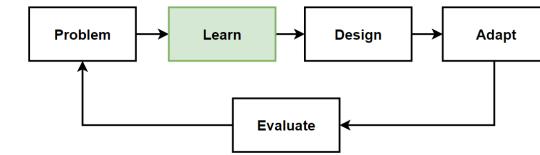
- **GraphQL Rate Limiting**

In REST API, we can simply specify that we allow only the specific amount of requests in specific time, but In GraphQL, it is difficult to specify this type of statement.

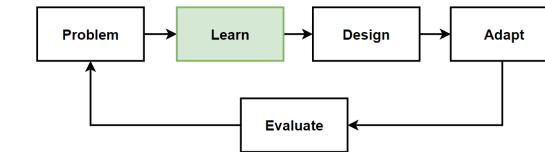


REST vs GraphQL APIs

- **REST** has become the de-facto standard for designing web APIs, However, REST APIs are too inflexible to rapidly changing requirements of the clients.
- **GraphQL** was developed for more flexibility and efficiency that solves many of the shortcomings and inefficiency calls.
- **REST** has the advantages of being a popular API, and also it is easy to understand, as well as scalable.
- **GraphQL** is more complex than REST, main drawbacks are error reporting, caching, and N+1.
- **GraphQL** solves the under-fetching problem that REST can't fetch hierarchical data at once.
- **GraphQL** and **REST** are two API styles that have a different approach when dealing with the transfer of data over HTTP protocols.



REST vs GraphQL APIs



REST APIs

- Easy to understand
- Defacto standard Popular APIs for designing web APIs
- Provide better scalability
- Built-in caching that minimizes multiple trips

Drawbacks:

- Over-fetching and under-fetching requests are hard to implement and required multiple calls.

GraphQL APIs

- Solving the over and under-fetching problem
- More flexible with Schema and Type System
- Quick response time
- Protocol agnostic

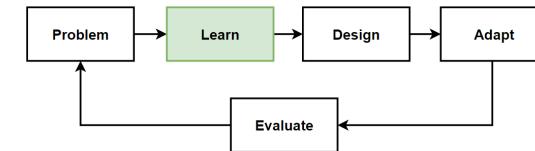
Drawbacks:

- Complexity and caching problems.



Fetching Data with REST and GraphQL

- **Blogging application**, an application needs to display screen,
- "posts" of the specific "user" and the last 3 "follower" names at the same screen.
- users -> posts, users -> followers
- How can we fetch this data with REST and GraphQL ?



Fetching Data with REST APIs

- **REST API**, we have to call multiple endpoints to fetch data.

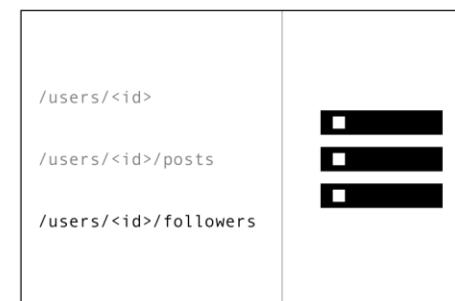
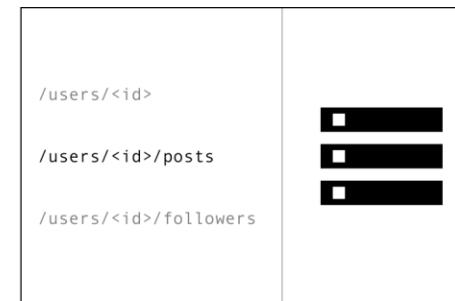
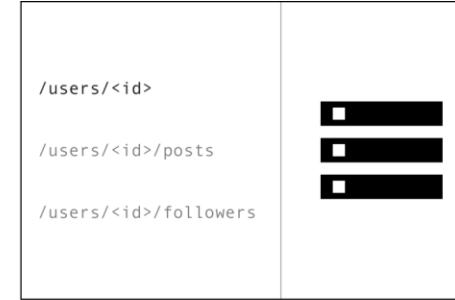
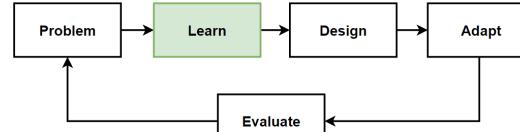
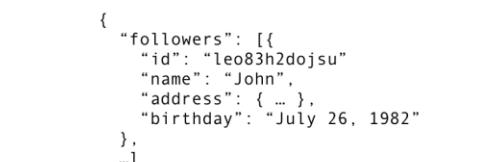
1. **/users/<id>** endpoint to fetch the initial user data.
2. **/users/<id>/posts** endpoint that returns all the posts for a user.
3. **/users/<id>/followers** that returns a list of followers per user.

- With REST, have to **make 3 requests** to different endpoints to fetch the required data.
- It is **over-fetching** since the endpoints return additional information that's not required.

1

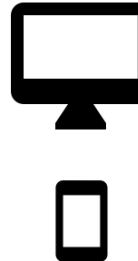
2

3



Fetching Data with GraphQL APIs

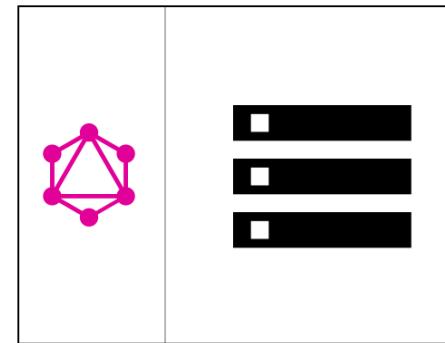
- **GraphQL API**, simply **send 1 single query** to the GraphQL server that includes the concrete data requirements.
- Then the server responds with a **JSON object** that includes required fields.
- Using GraphQL, the client can specify **exactly the data** it needs in a query.
- The structure of the server's response follows **precisely the nested structure** defined in the query.



```
query {  
  User(id: "er3tg439frjw") {  
    name  
    posts {  
      title  
    }  
    followers(last: 3) {  
      name  
    }  
  }  
}  
  
HTTP POST  
→  
Cloud icon  
←  


```
{
 "data": {
 "User": {
 "name": "Mary",
 "posts": [
 { "title": "Learn GraphQL today" }
],
 "followers": [
 { "name": "John" },
 { "name": "Alice" },
 { "name": "Sarah" }
]
 }
 }
}
```


```

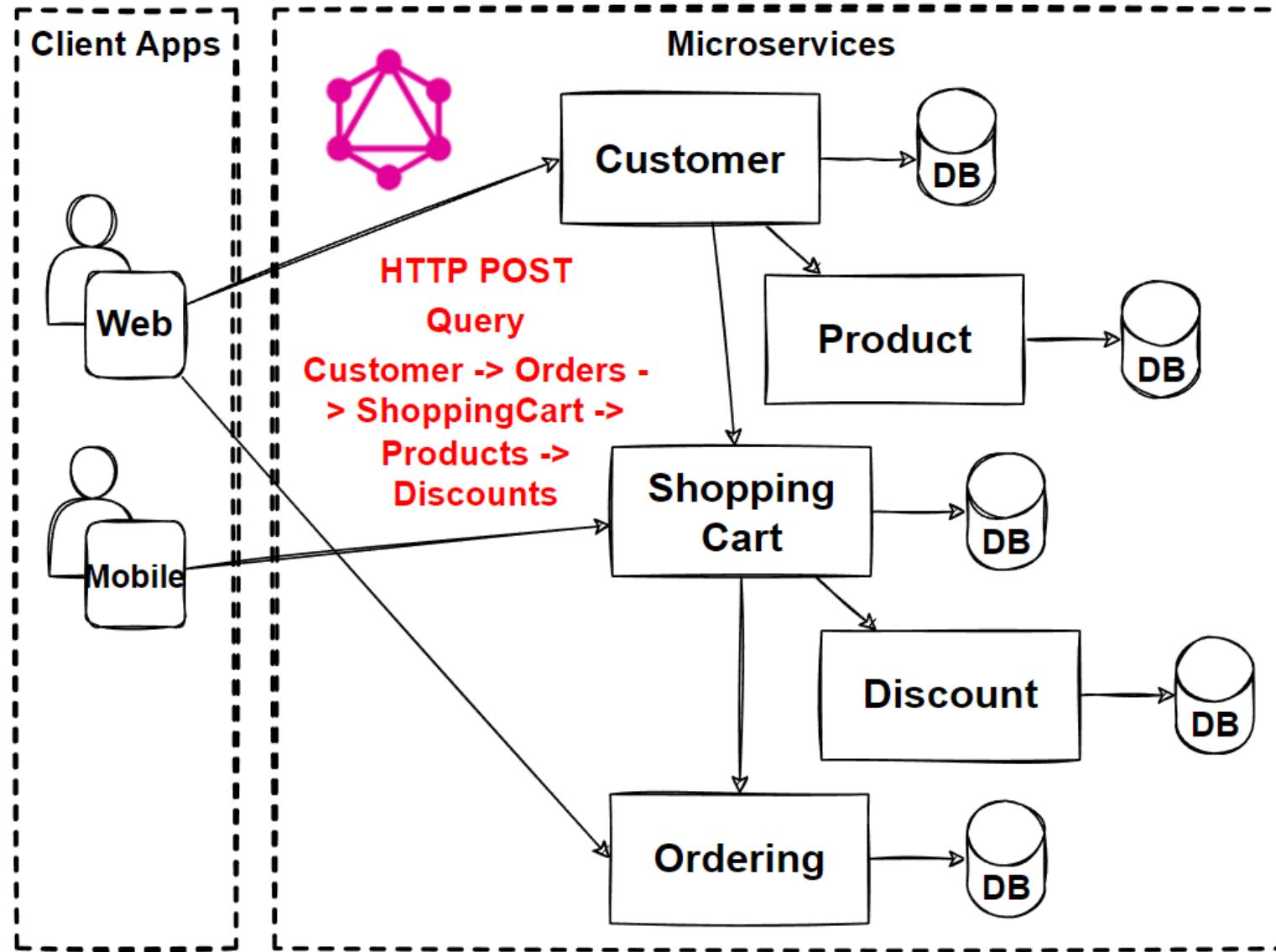
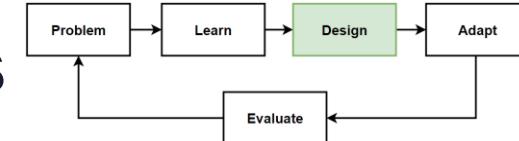


<https://www.howtographql.com/basics/1-graphql-is-the-better-rest/>

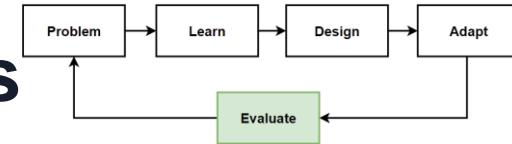
Before Design – What we have in our design toolbox ?

Architectures	Patterns&Principles	Non-FR	FR
• Microservices Architecture	<ul style="list-style-type: none">• The Database-per-Service Pattern• Polygot Persistence• Decompose services by scalability• The Scale Cube• Microservices Decomposition Pattern• Microservices Communications<ul style="list-style-type: none">• HTTP Based RESTful API design• GraphQL API design	<ul style="list-style-type: none">• High Scalability• High Availability• Millions of Concurrent User• Independent Deployable• Technology agnostic• Data isolation• Resilience and Fault isolation	<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history

Design: Microservices Architecture with GraphQL APIs



Evaluate: Microservice Architecture with GraphQL APIs

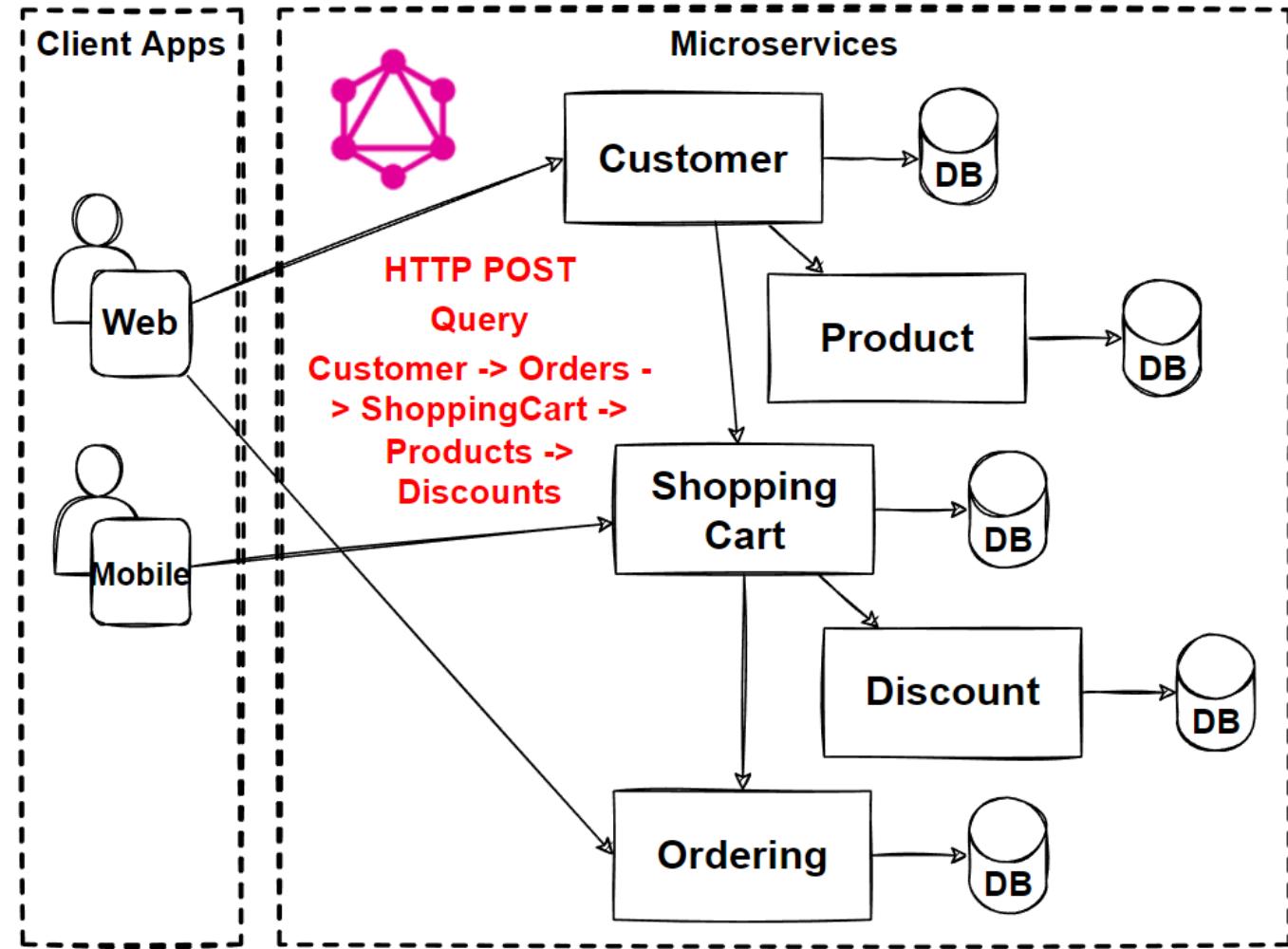


Benefits

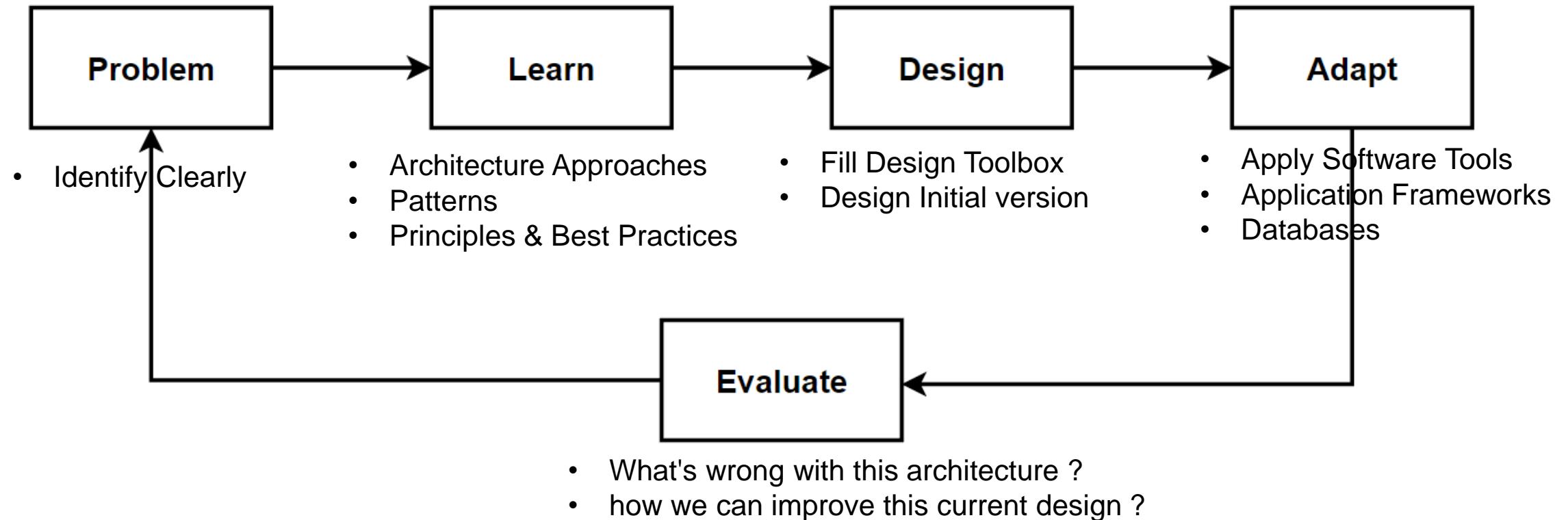
- Solving the over and under-fetching problem
- More flexible with schema and type system
- Quick response time
- Protocol agnostic

Drawbacks

- Increase complexity
- Caching problems

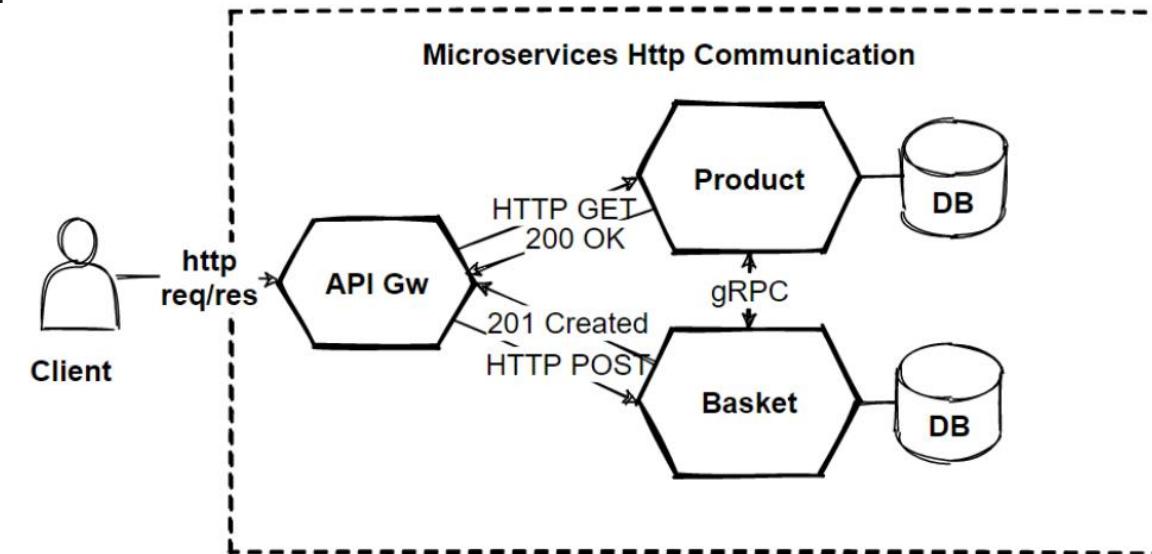
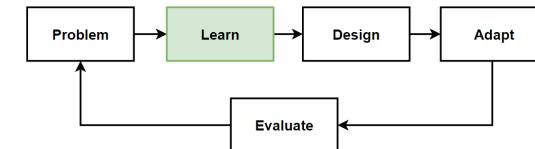


Way of Learning – The Course Flow

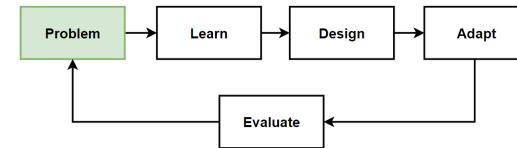


Designing Sync APIs for Microservices

- There are 2 type APIs sync communication in microservices:
 - Public APIs: API calls from the client apps.
 - Backend APIs: inter-service communication between backend services.
- **Public APIs**
Use RESTful APIs over HTTP protocol. RESTful APIs use JSON payloads for request-response, that easy to check payloads and easy agreement with clients.
- **Backend APIs**
Inter-service communication can result in a lot of network traffic. serialization speed and payload size become more important. Using gRPC is mandatory for increase network performance.



Problem: Inter-service communication makes heavy load on network traffic



Problems

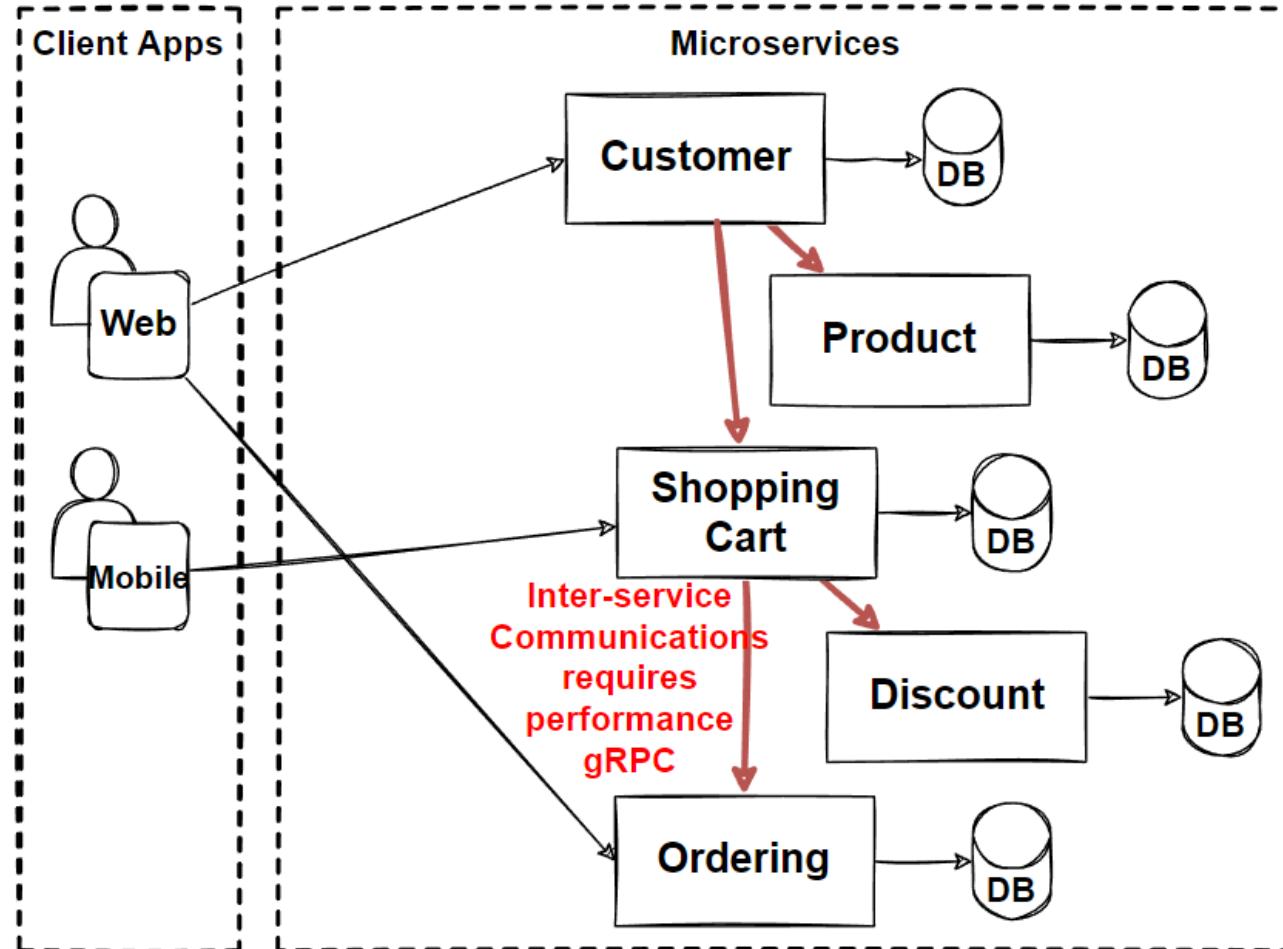
- Network performance issues on inter-service communication
- Backend Communication performance requirements
- Real-time communication requirements
- Streaming requirements

Example Use Case

- Add Item into Shopping Cart that need to calculate with up-to-date discounts

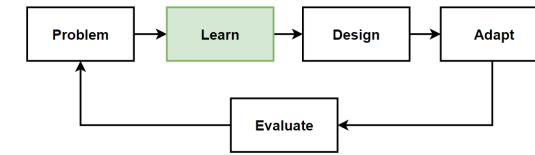
Solutions

- gRPC APIs scalable and fast APIs
- Able to develop with different technologies with RPC framework



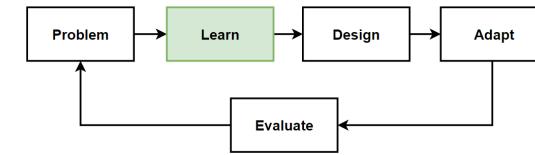
gRPC: High Performance Remote Procedure Calls

- gRPC is an open source **remote procedure call** (RPC) system developed at Google.
- gRPC is a framework to **efficiently connect services** and build distributed systems.
- It is **focused on high performance** and uses the **HTTP/2 protocol** to transport binary messages.
- It relies on the **Protocol Buffers** language to define service contracts.
- **Protocol Buffers (Protobuf)**, allow to define the interface to be used in service to service communication regardless of the programming language.
- It generates **cross-platform client and server bindings** for many languages.
- Most common **usage** scenarios include connecting services in **microservices style architecture**.



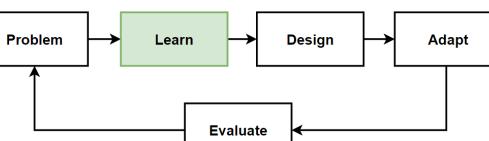
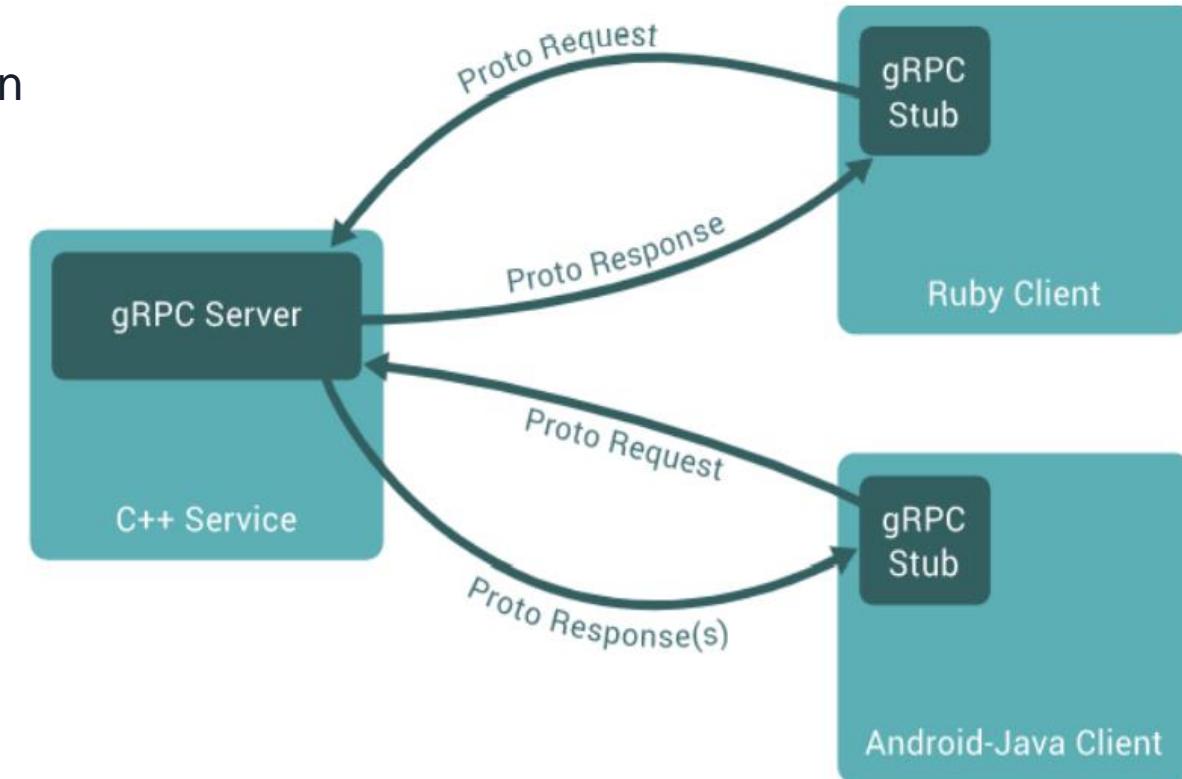
gRPC: High Performance Remote Procedure Calls

- **gRPC framework** allows developers to create services that can communicate with each other **efficiently** and **independently** with their **preferred programming language**.
- Once you **define a contract** with **Protobuf**, this contract used by each service to automatically **generate the code** that sets up the communication infrastructure.
- This feature simplifies the creation of service interaction and together with **high performance**, makes **gRPC** the **ideal framework** for creating **microservices**.



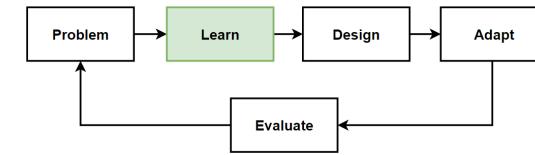
How gRPC works ?

- RPC is a form of **Client-Server Communication** that uses a **function call** rather than a usual HTTP call.
- In **GRPC**, client application can **directly call a method** on a server application on a **different machine** like a local object.
- **gRPC** is based on the idea of **defining a service** and **methods**, and these can be called remotely with their parameters and return types.
- On the **server side**, the server implements this interface and runs a gRPC server to handle client calls.
- On the **client side**, the client has a stub that provides the same methods as the server.
- **gRPC clients and servers** can work and talk to each other in a different of environments.



Main Advantages of gRPC

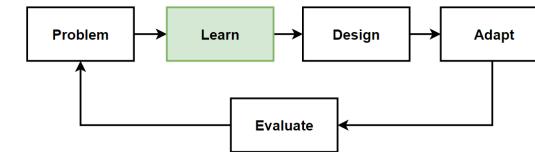
- Using **HTTP/2** provides **30-40% more performance**.
- **gRPC** uses **binary serialization**, provides **higher performance** and **less bandwidth** usage than **json** with binary serialization.
- Supporting a widely using **multi-languages** and **platform supports**.
- **Open Source** and the powerful community behind it.
- Supports **Bi-directional Streaming** operations.
- Support **SSL/TLS** usage.
- Supports many **Authentication methods**.



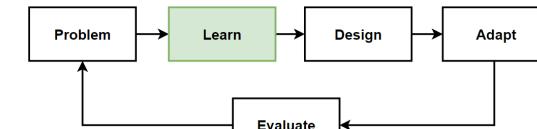
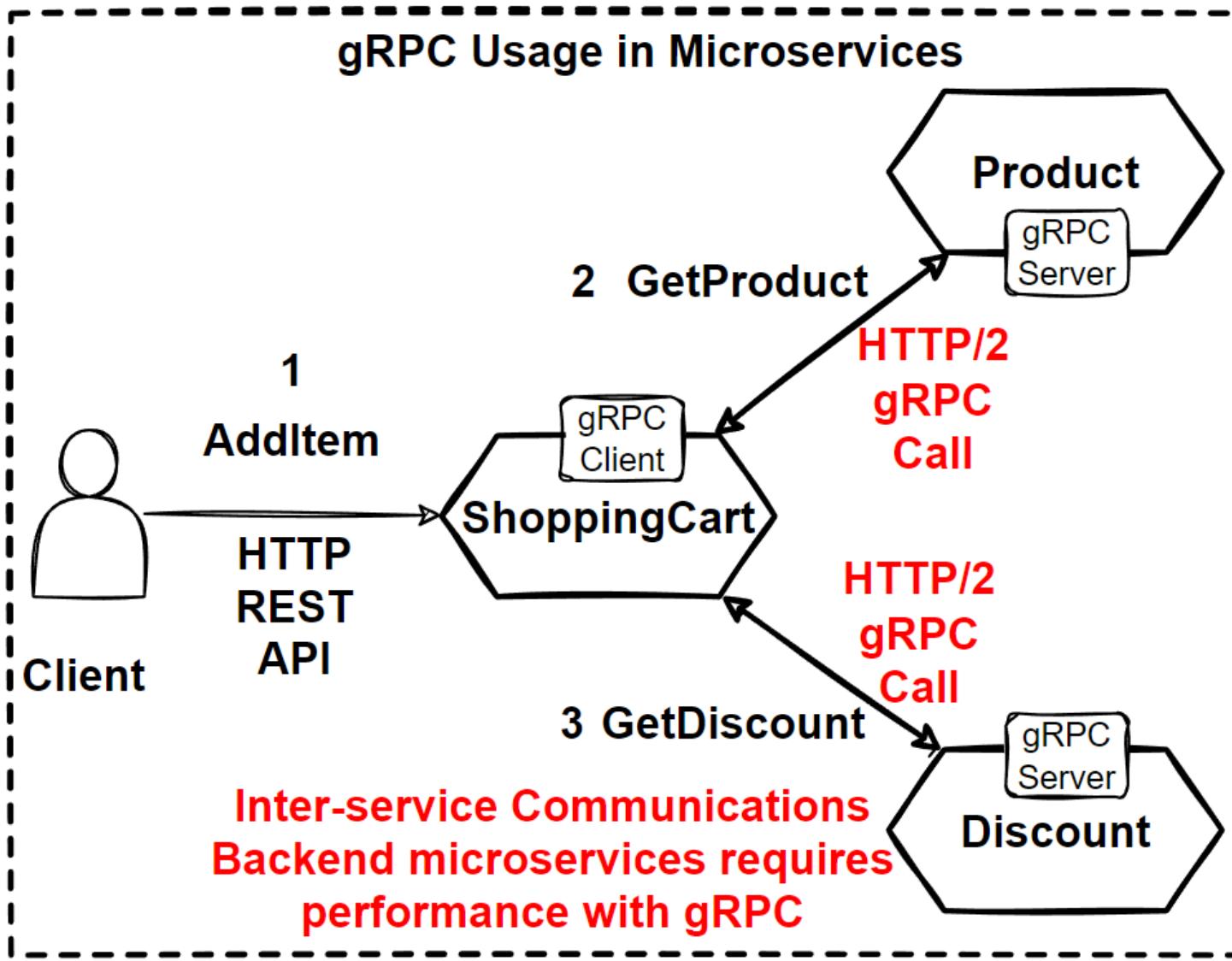
↑ GRPC ↓

When to use gRPC ? Use Cases of gRPC

- gRPC is primarily used with **backend services**.
- **Synchronous backend microservice-to-microservice communication** where an immediate sync response is required to continue processing.
- **Polyglot environments** that need to **support mixed programming platforms**.
- **Low latency and high throughput communication** where performance is critical.
- **Point-to-point real-time communication** that gRPC can push messages in real time and requires to support for bi-directional streaming.
- **Network constrained environments** which requires **less bandwidth usage** cases, binary gRPC messages are always smaller than text-based JSON message.



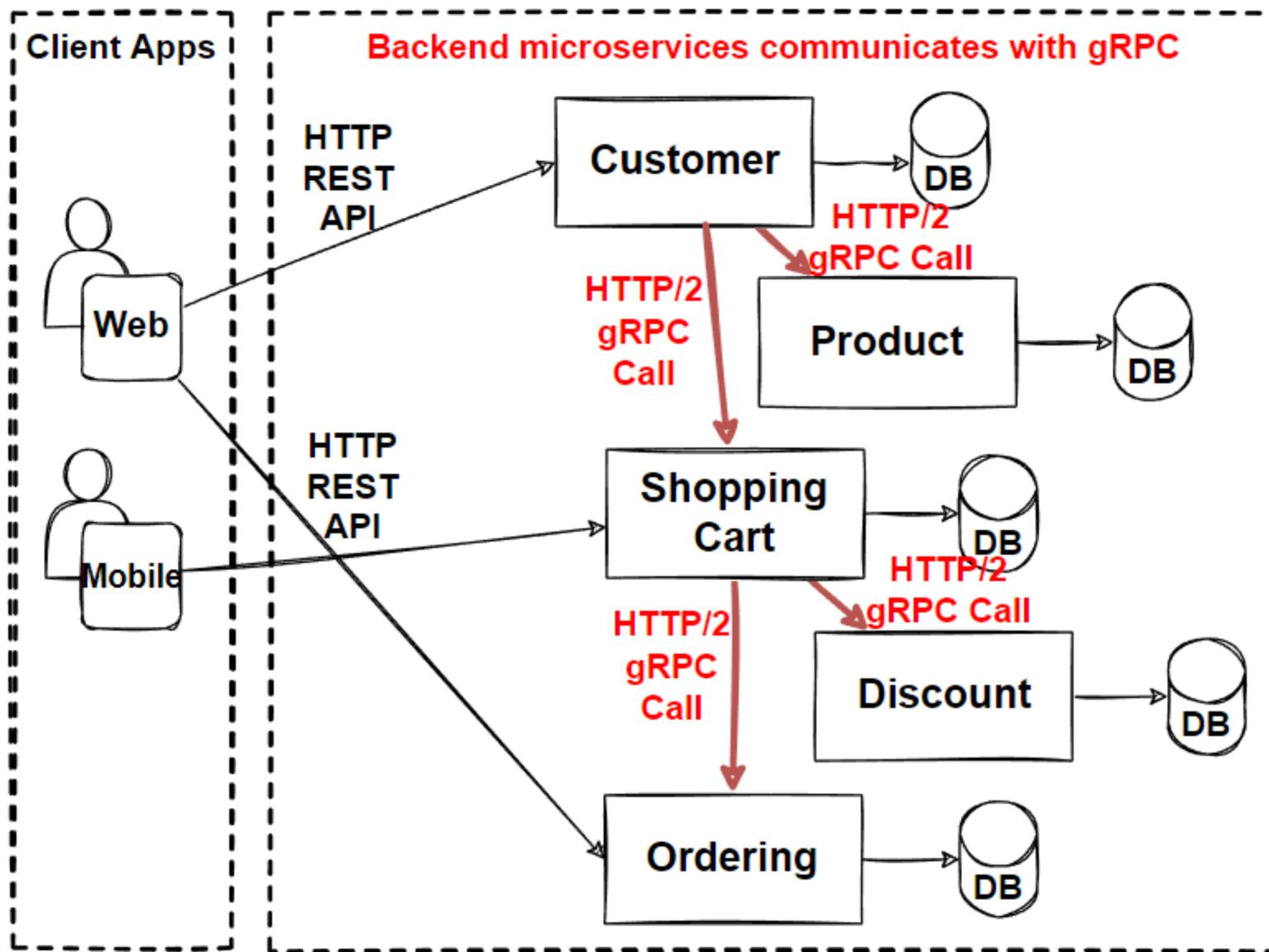
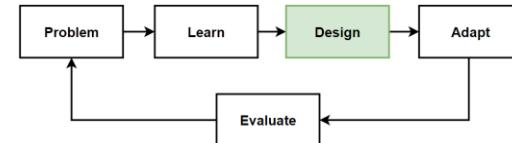
gRPC Usage in Microservices Communication



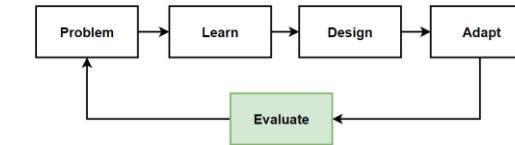
Before Design – What we have in our design toolbox ?

Architectures	Patterns&Principles	Non-FR	FR
<ul style="list-style-type: none">• Microservices Architecture	<ul style="list-style-type: none">• The Database-per-Service Pattern• Polygot Persistence• Decompose services by scalability• The Scale Cube• Microservices Decomposition Pattern• Microservices Communications<ul style="list-style-type: none">• HTTP Based RESTful API design• GraphQL API design• gRPC API Design	<ul style="list-style-type: none">• High Scalability• High Availability• Millions of Concurrent User• Independent Deployable• Technology agnostic• Data isolation• Resilience and Fault isolation	<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history

Design: Microservices Architecture with gRPC APIs



Evaluate: Microservice Architecture with gRPC APIs

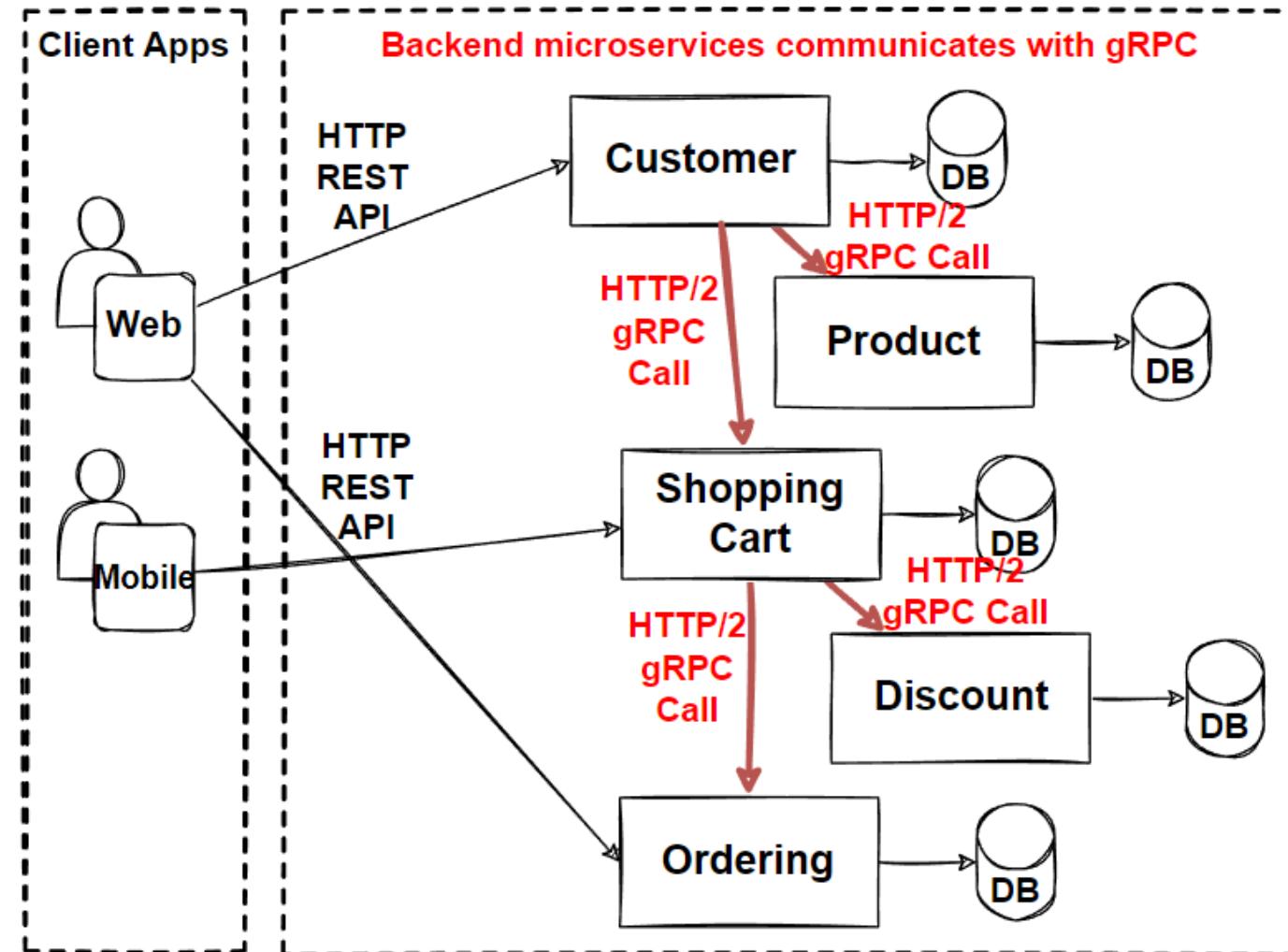


Benefits

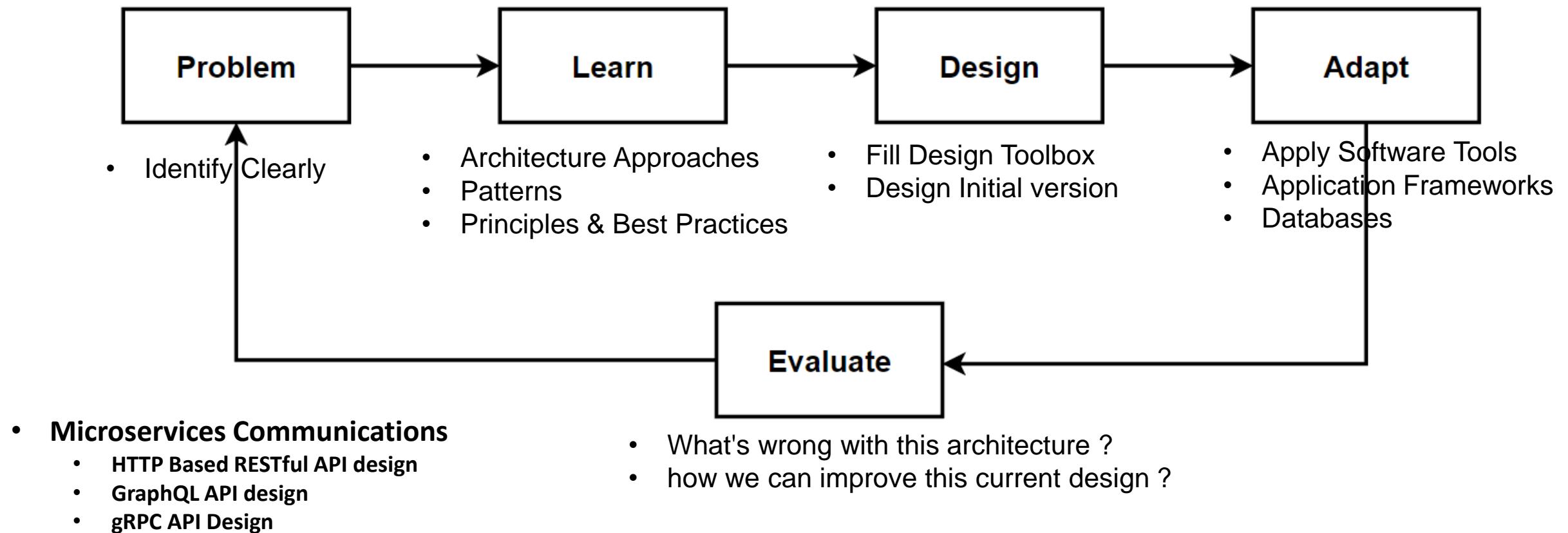
- High-performance communication
- Less bandwidth usage
- Multi-language/platform supports
- HTTP/2 and SSL/TLS usage
- Supports Bi-directional Streaming operations

Drawbacks

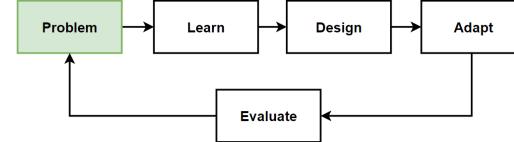
- Need additional library to generate codes that increases complexity
- Can't read incoming-outgoing payloads that is not good for client to service communications



Way of Learning – The Course Flow

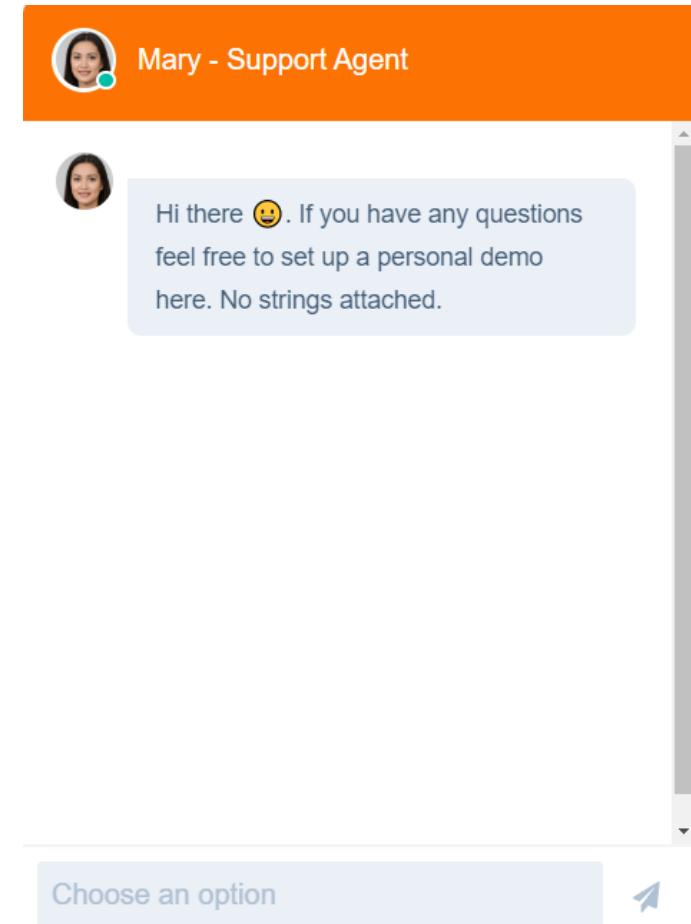


Problem: Chat with Support Agent



Problems

- Business teams request to answer Customer queries by chatting with Support Agents
- Real-time communication requirements
- Sending/receiving messages in Chat window



Example Use Case

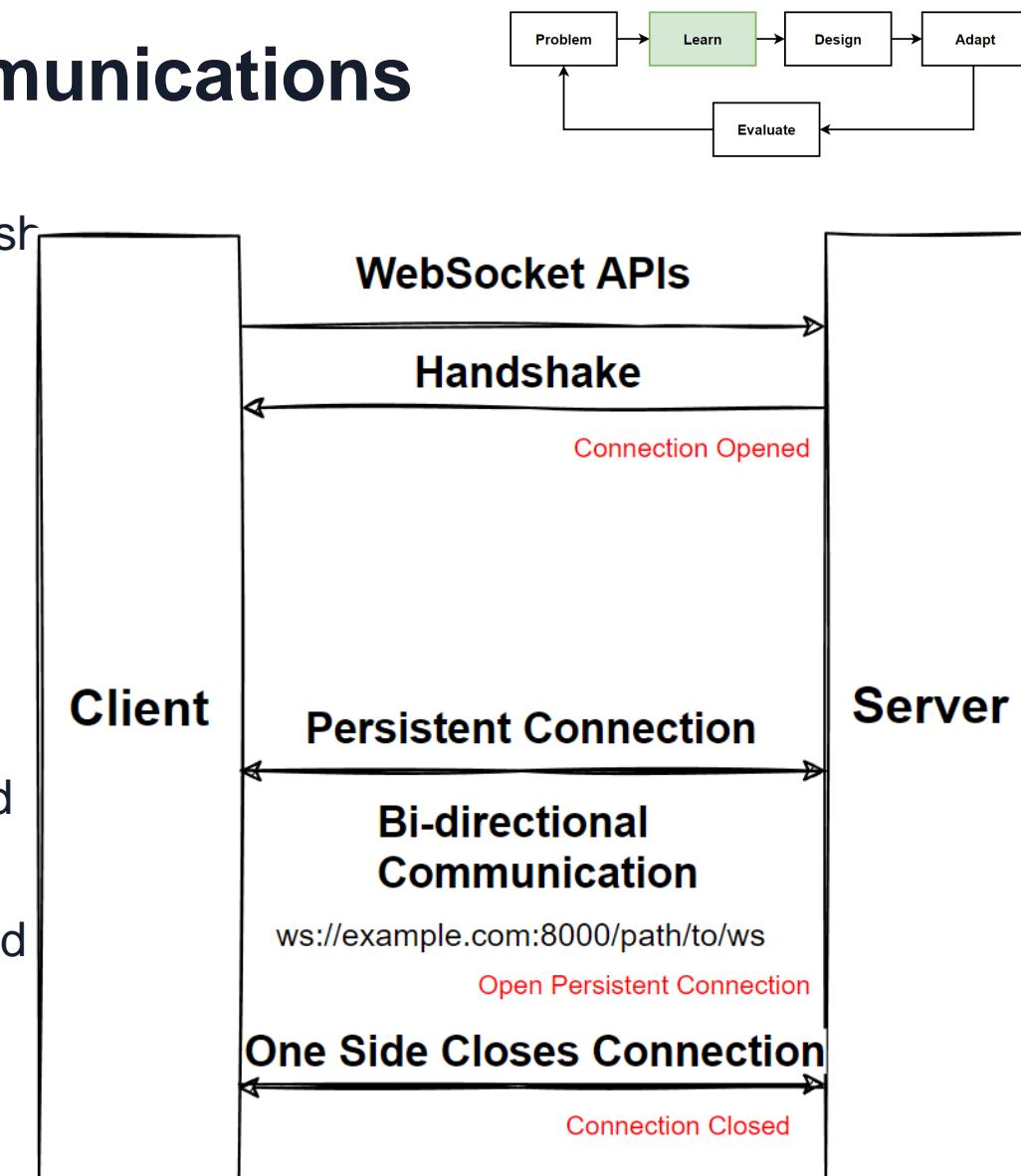
- E-commerce Online Agent help customer preferences as per product features on website

Solutions

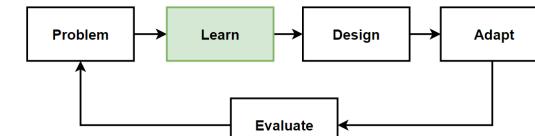
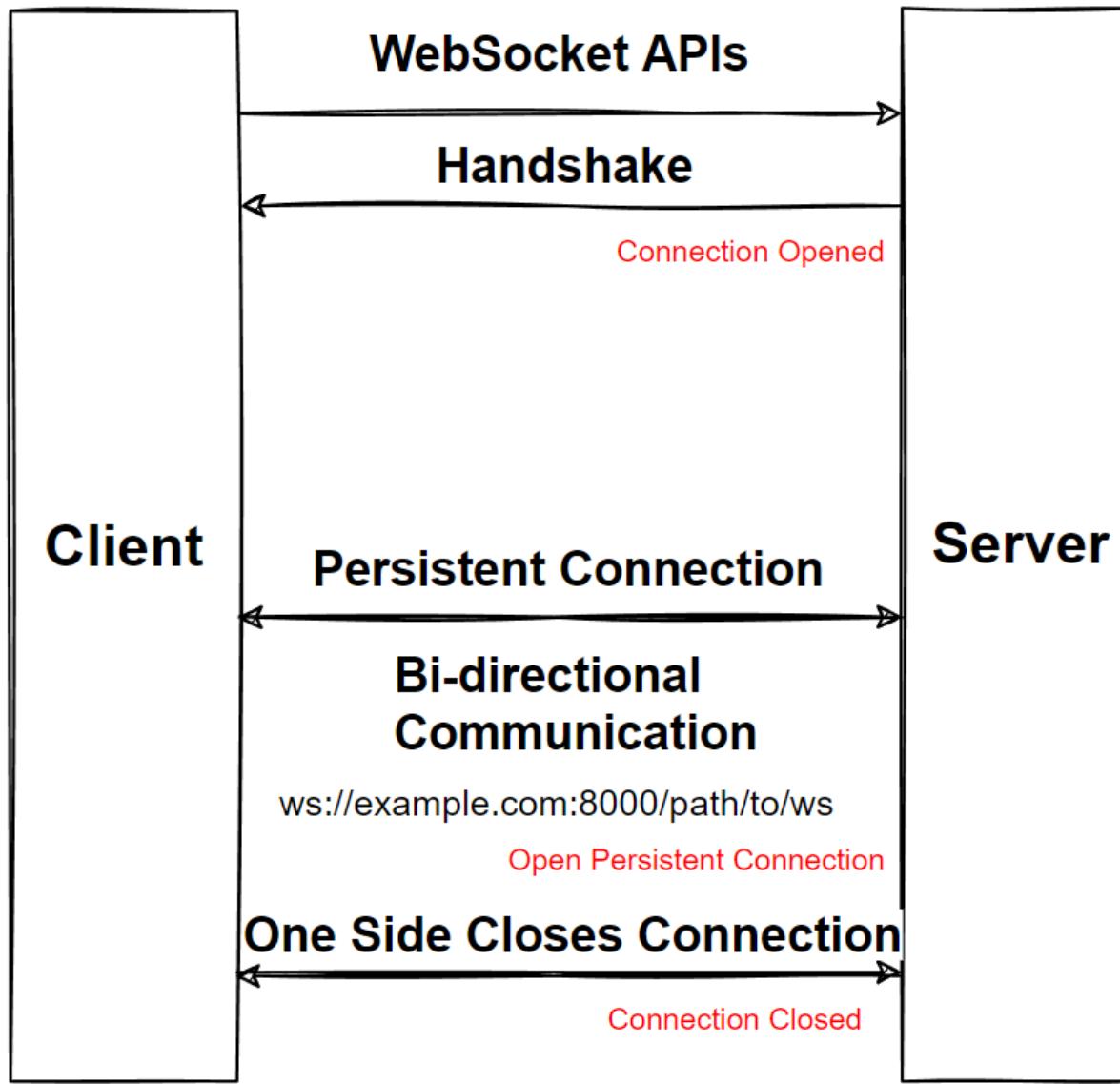
- WebSocket APIs: Build real-time two-way communication applications

WebSocket API: Real-time two-way communications

- **WebSocket API** is an advanced technology that user can establish an **interactive two-way communication session** between **browser** and a **server**.
- **Send a message** to a **server** and **receive event-based responses** without having to **poll** the server for a response.
- **WebSockets** is a good way to handle high-scale data transfers between server-client.
- **WebSocket** is **bidirectional**, it starts with **ws://** or **wss://**
- Unlike HTTP, WebSocket provides a **bidirectional protocol** used in the **same client-server communication** cases.
- **WebSocket** is a **stateful protocol**, connection between client and server stays alive until terminated by one of the parties.
- After closing the connection by either client or server, the **connection is terminated from both ends**.



WebSocket API: Real-time two-way communications



When to use WebSocket API

- **Developing Real-time Web application**

The most usage area of WebSocket is in real-time application development where it is required to continuous display of data at the client end. In WebSocket, data is constantly pushed/forwarded to the same connection that is already open.

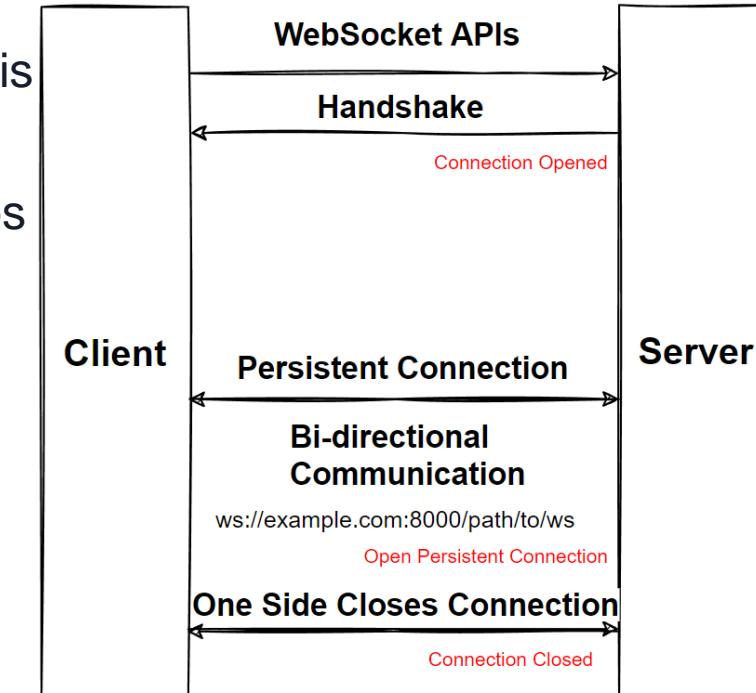
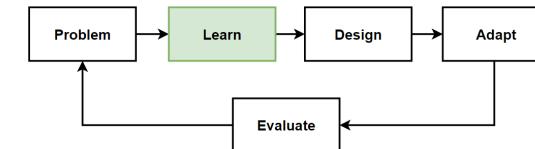
- Real-life example of WebSocket is stock trading applications. WebSocket helps to data handling that is drives by the deployed backend server to the client.

- **Developing Game application**

Gaming application requires to changes on the UI with WebSockets without updating the page and refreshing UI. WebSocket accomplish this goal without disturbing the UI of the gaming app.

- **Developing Chat application**

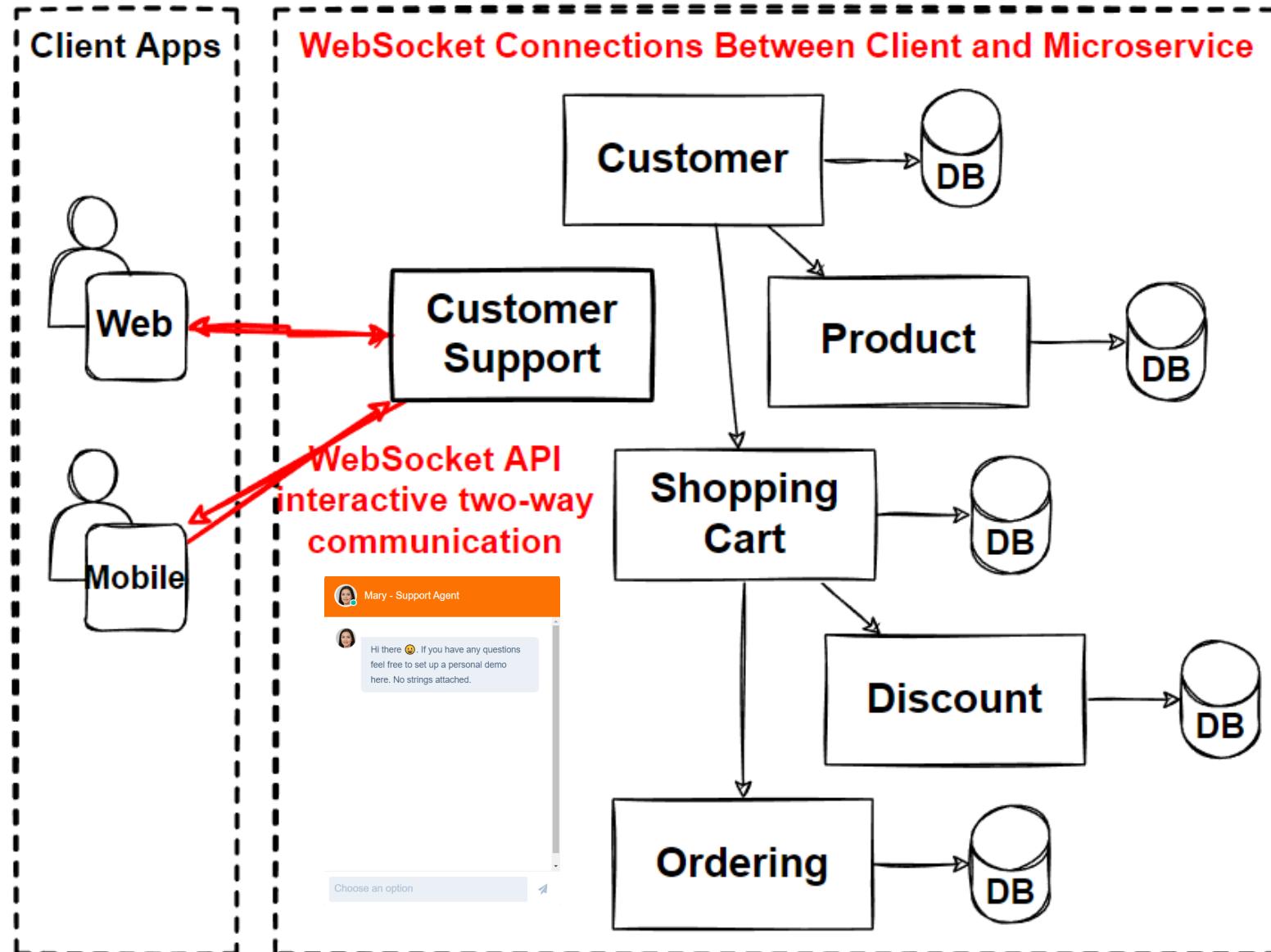
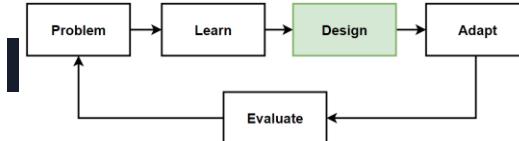
Chat application uses WebSocket to use fixed connection when exchange the message between subscriber, broadcasting channels. WebSocket connection is used for sending/receiving messages, communication becomes easy and quick.



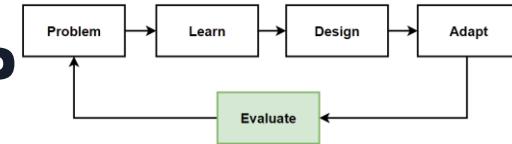
Before Design – What we have in our design toolbox ?

Architectures	Patterns&Principles	Non-FR	FR
<ul style="list-style-type: none">• Microservices Architecture	<ul style="list-style-type: none">• The Database-per-Service Pattern• Polygot Persistence• Decompose services by scalability• The Scale Cube• Microservices Decomposition Pattern• Microservices Communications<ul style="list-style-type: none">• HTTP Based RESTful API design• GraphQL API design• gRPC API Design• WebSocket API	<ul style="list-style-type: none">• High Scalability• High Availability• Millions of Concurrent User• Independent Deployable• Technology agnostic• Data isolation• Resilience and Fault isolation	<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history

Design: Microservices Architecture with WebSocket API



Evaluate:Microservice Architecture with WebSocket AP

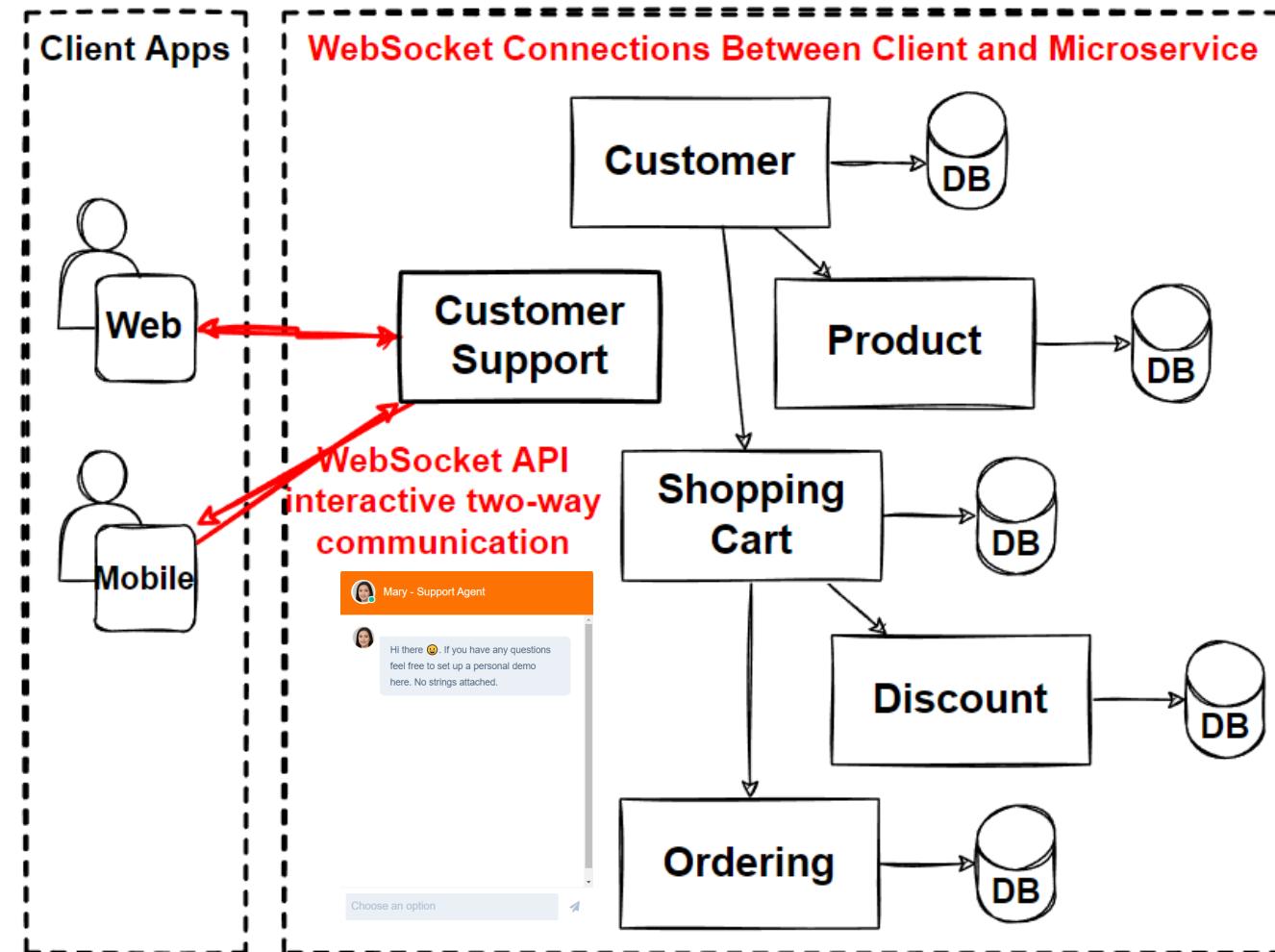


Benefits

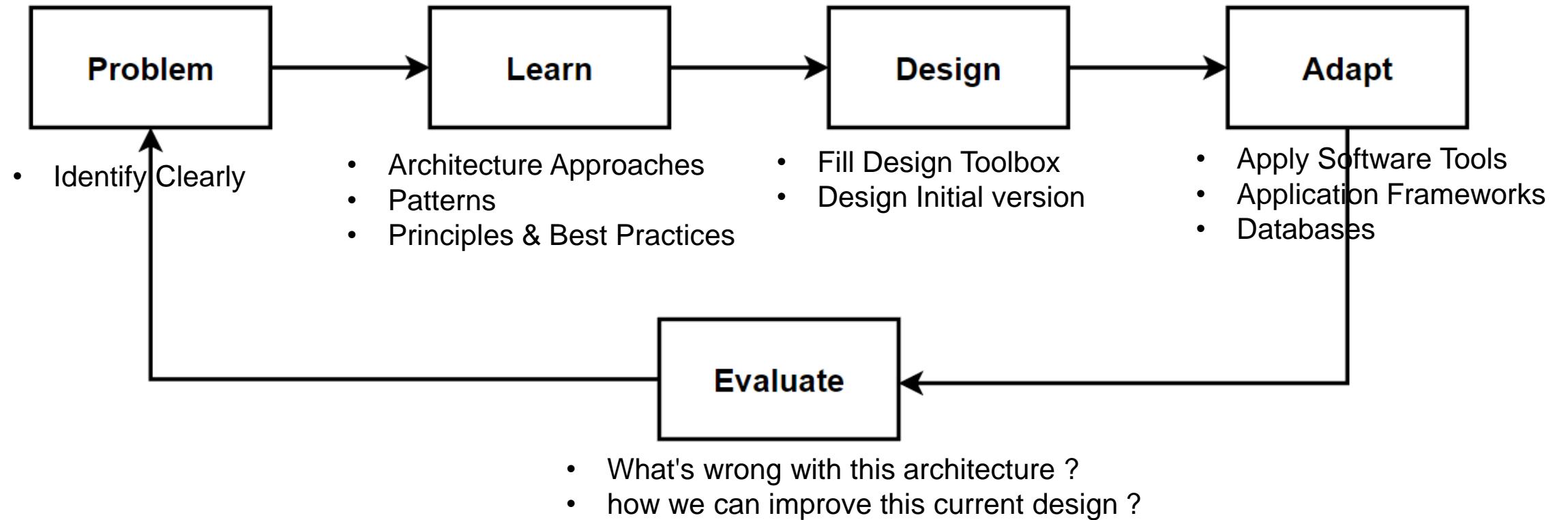
- Full-duplex communication
- Reduces unnecessary network traffic
- Real-time communication
- Persistent, Bidirectional Communication

Drawbacks

- A fully HTML5-compliant web browser is required.
- Not provide intermediary/edge caching.

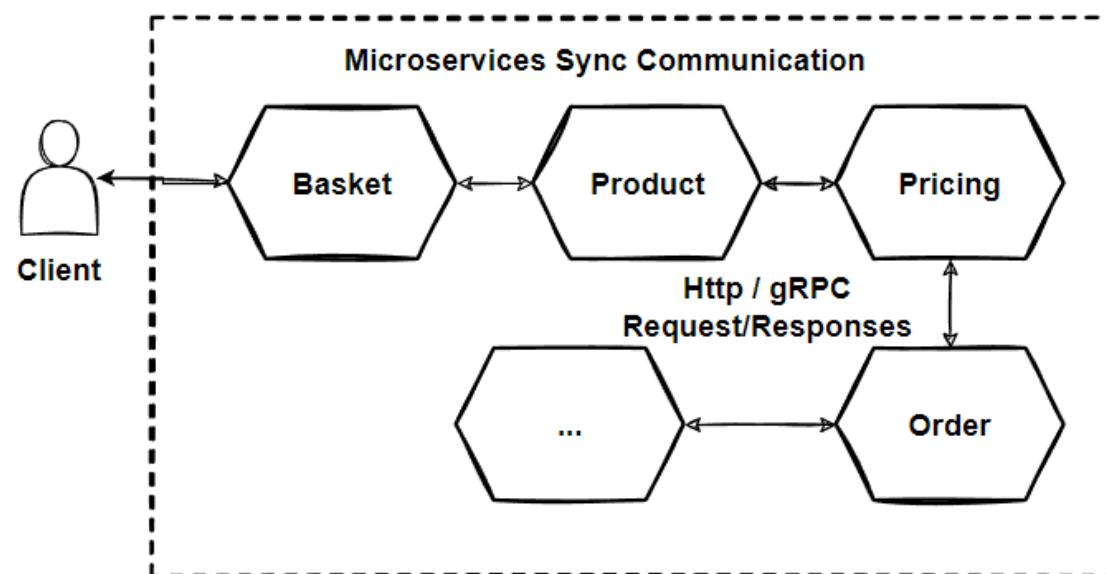
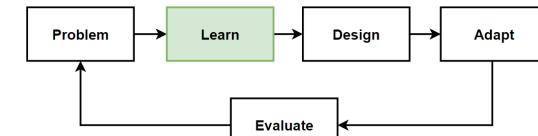


Way of Learning – The Course Flow



Microservices Synchronous Communications and Best Practices

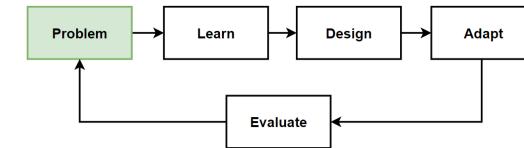
- The client sends a request with using http protocols and waits for a response from the service.
- The synchronous communication protocols can be HTTP or HTTPS.
- **Request/response communication** with HTTP and REST Protocol (extends gRPC and GraphQL)
 - REST HTTP APIs when exposing from microservices
 - gRPC APIs when communicate internal microservices
 - GraphQL APIs when structured flexible data in microservices
 - WebSocket APIs when real-time bi-directional communication



Question:

- How client applications can manage all these different protocols and communications inside of their applications ?

Problem: Direct Client-to-Service Communication



Problems

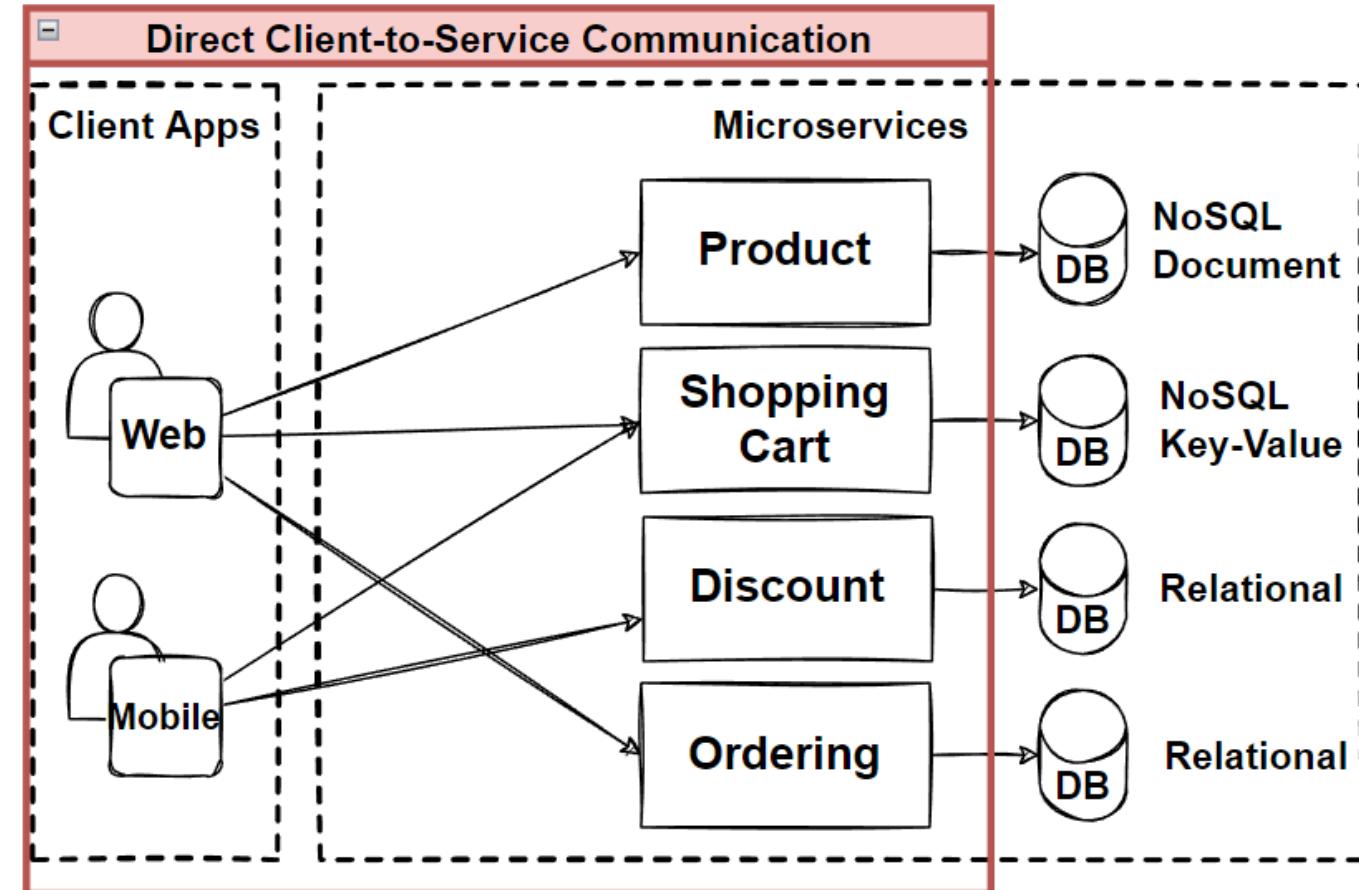
- Direct Client-to-Service Communication
- Cause to chatty calls from client to service
- Hard to manage invocations from client app with all different protocols (HTTP, GraphQL, gRPC, WebSocket)

Solutions

- Well-defined API Design
- **Microservices Communication Patterns**

Steps

- Microservice Communications and Types
- Well-defined RESTful API Design for services
- **Microservices Communication Patterns (API Gateway, BFF, Publish/Subscribe..)**



Microservices Communications – API Gateways

API Gateway Pattern

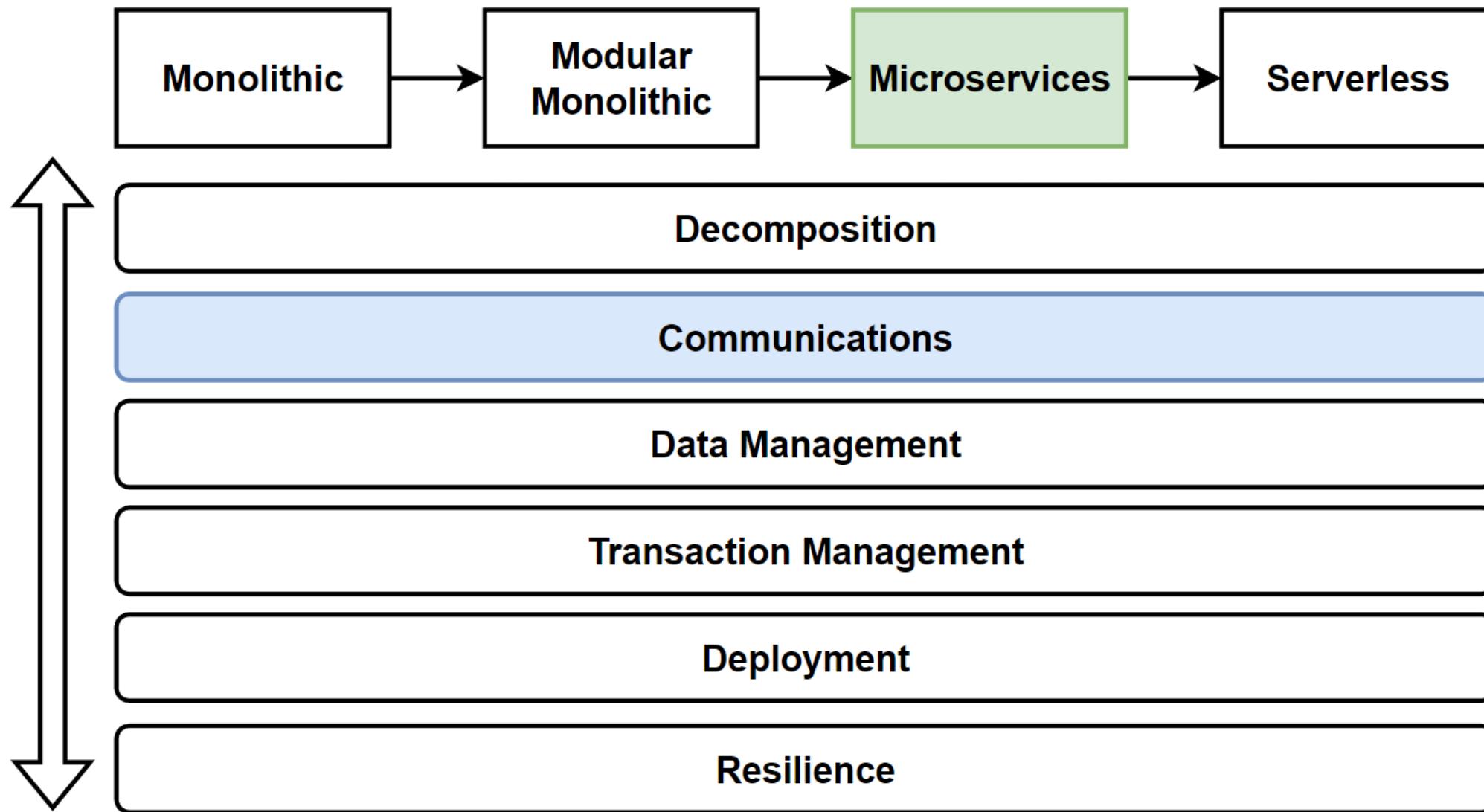
Gateway Routing Pattern

Gateway Aggregation Pattern

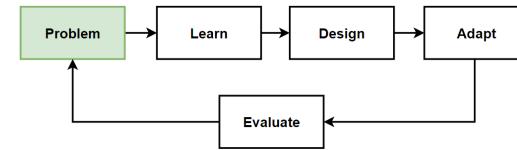
Gateway Offloading Pattern

Backends for Frontends Pattern-BFF

Architecture Design – Vertical Considerations



Problem: Direct Client-to-Service Communication



Problems

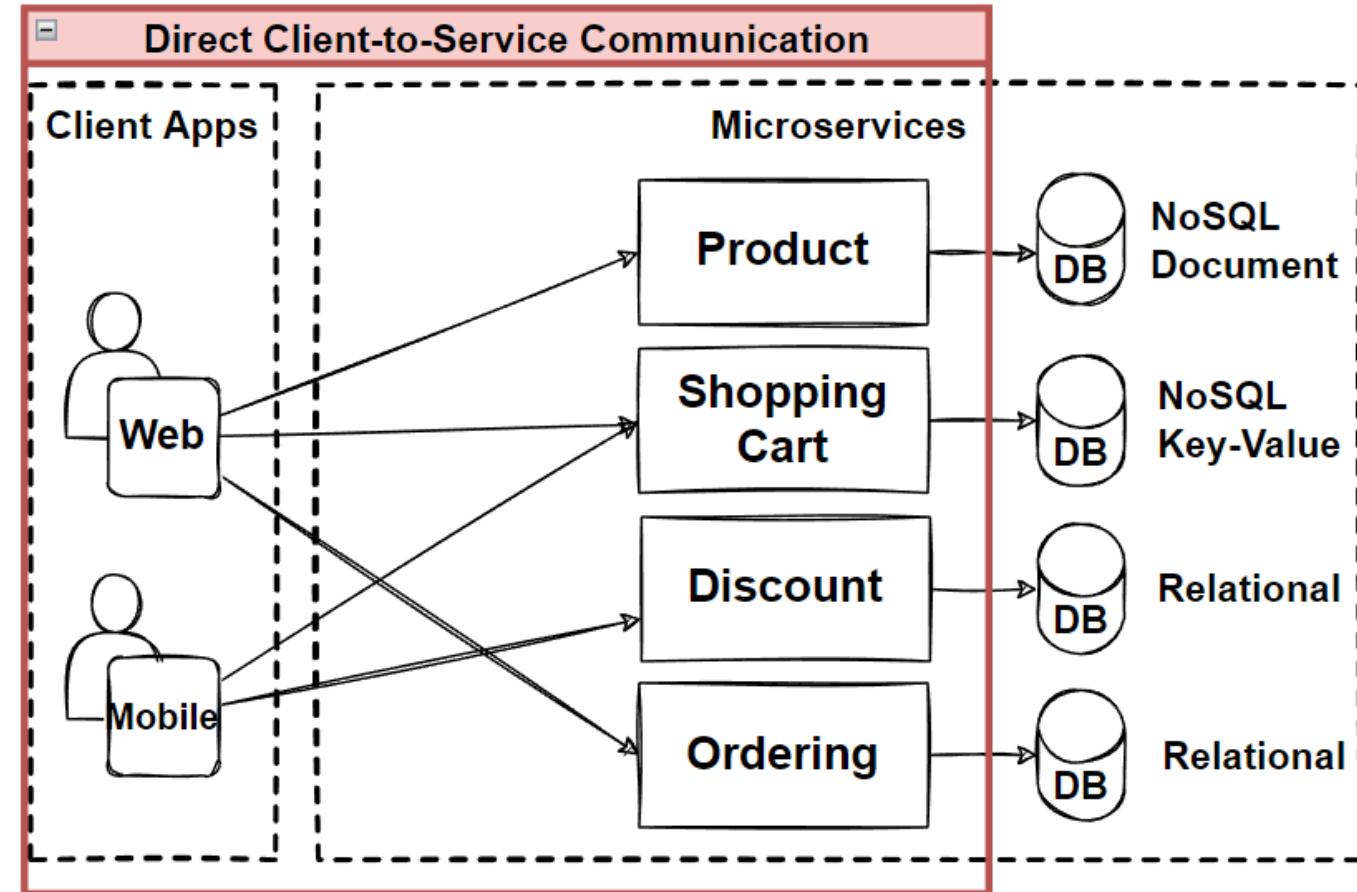
- Direct Client-to-Service Communication
- Cause to chatty calls from client to service
- Hard to manage invocations from client app with all different protocols (HTTP, GraphQL, gRPC, WebSocket)
- Increased latency and complexity on the UI side

Solutions

- Well-defined API Design
- **Microservices Communication Patterns**

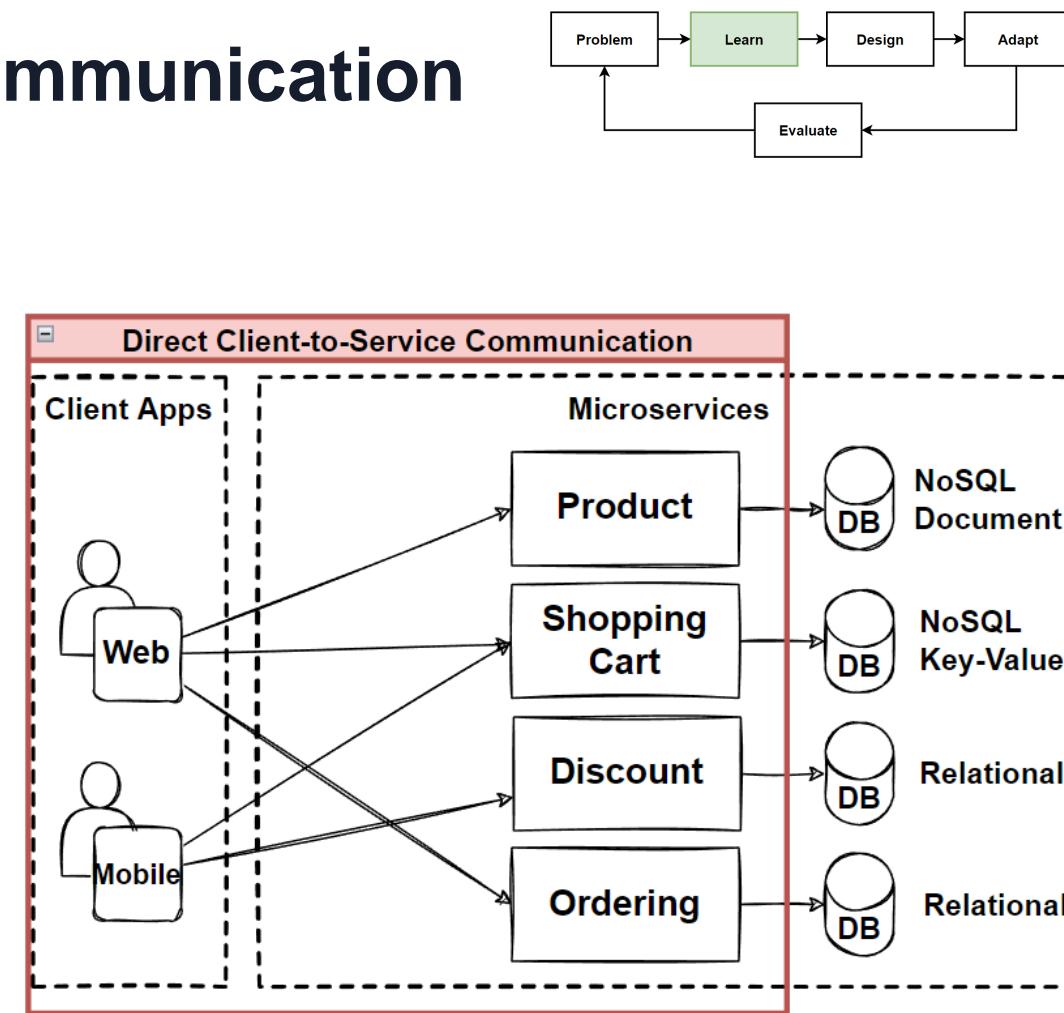
Steps

- Microservice Communications and Types
- Well-defined RESTful API Design for services
- **Microservices Communication Patterns (API Gateway, BFF, Publish/Subscribe..)**

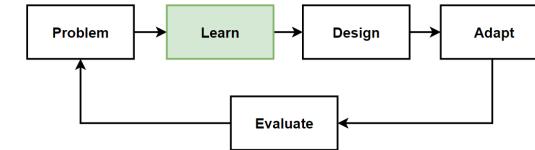


Problems of Direct-to-Microservices Communication

- Drawbacks of the **direct client-to-microservices** communications, Comparison with API Gateway and direct client-to-microservice communications.
- In diagram, each microservices **has to open public endpoint** to the client applications but it has lots of drawbacks.
- If it is **small microservice-based application**, and if you don't have so much client applications, it could be good fit to use direct client-to-microservices communication.
- When you build **large and complex microservice-based** applications, if you have **dozens of microservices**, and several client applications that need to communicate multiple microservices.
- Direct-to-microservices communication can **creates major problems**.



Main Reasons of Why should use API Gateway



- **Client Can't Manage Communications**

The client applications can try to handle multiple calls to microservice endpoints, not manageable when it comes to communicates with different protocols like http, GraphQL, gRPC and WebSocket.

- **Complex Client Code**

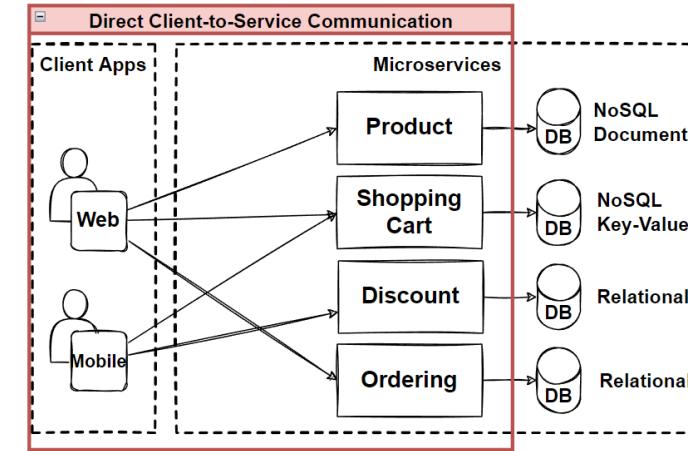
Causes lots of requests to the backend services, creates possible chatty communications. Increases latency and complexity on the UI side. Becomes complex client code.

- **Strong Coupling**

Creates coupling between the client and the backend. The client needs to know how the specific microservices are decomposed. Increases the coupling between client and services and also makes it harder to maintain the client.

- **Cross-cutting Concerns**

Authentication, Authorization, Rate Limiting, SSL certifications, Logging, Monitoring, Load Balancing and Circuit Breaker. Implementing all these cross-cutting concerns for every microservice is not a good solution.



Main Reason why should use API Gateway ? -2

- **Protocol Exchanges**

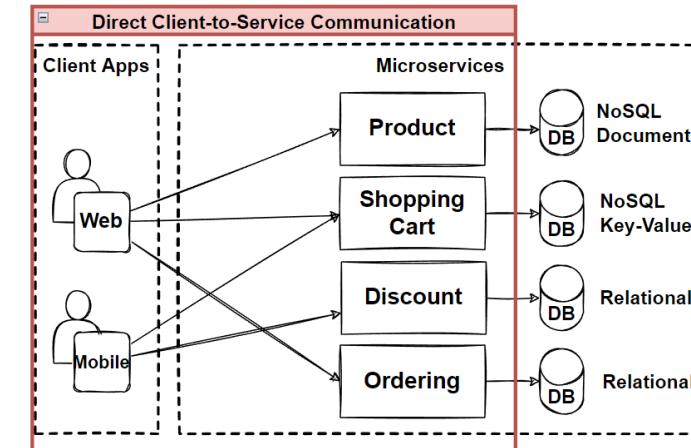
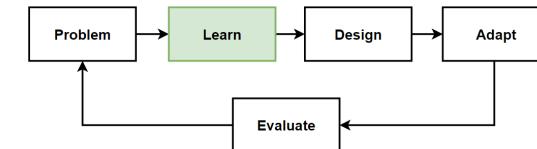
Communication required to use different protocols like exposing HTTP Rest API but continue to gRPC for internal communications or exposing WebSocket protocol. Requests must be translate to the other protocols afterwards.

- **Async Communication Requirements**

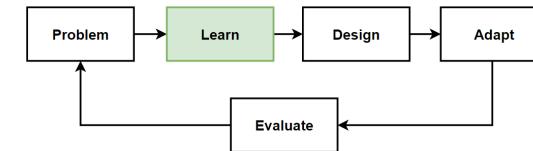
Client application can't use async communication due to not supported AMQP protocols. Microservices uses async communication in order to decouple communication as soon as possible. Hard to implement event-driven publish-subscribe model.

- **Payload Changes for Different Clients**

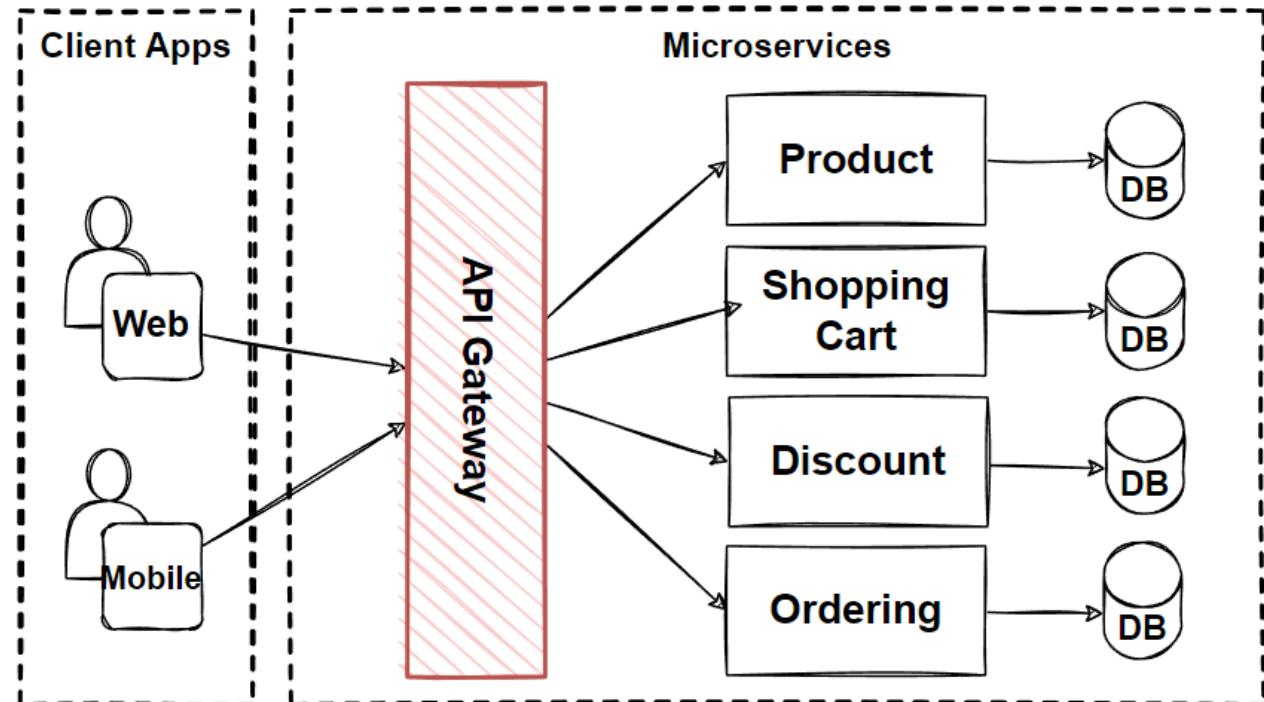
Web, Mobile and other client applications might be required different payloads when communicating with internal services. It requires to optimize data responses from services tailored with client applications.



The Solution - API Gateway Patterns

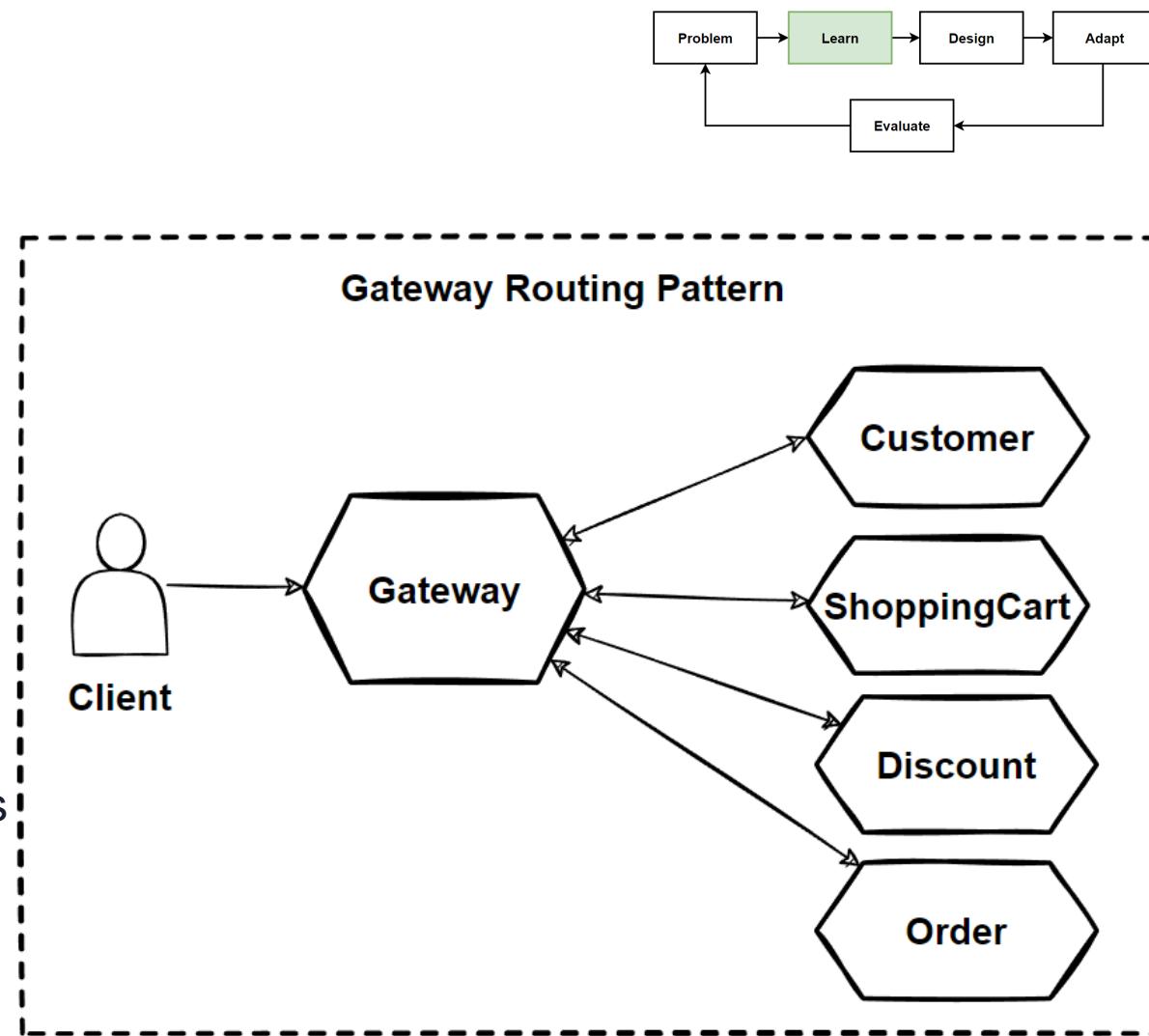


- Use **API Gateways** between client and internal microservices and it is a **single point of entry** to the client applications.
- **API Gateway** sits between **the client and multiple backend services** and manage **routing** to internal microservices.
- **API Gateways** can handle that **Cross-cutting concerns** like authentication, authorization, protocol translations, Rate Limiting, Logging, Monitoring, Load Balancing.
- API Gateway Patterns:
 - **Gateway Routing pattern**
 - **Gateway Aggregation pattern**
 - **Gateway Offloading pattern**



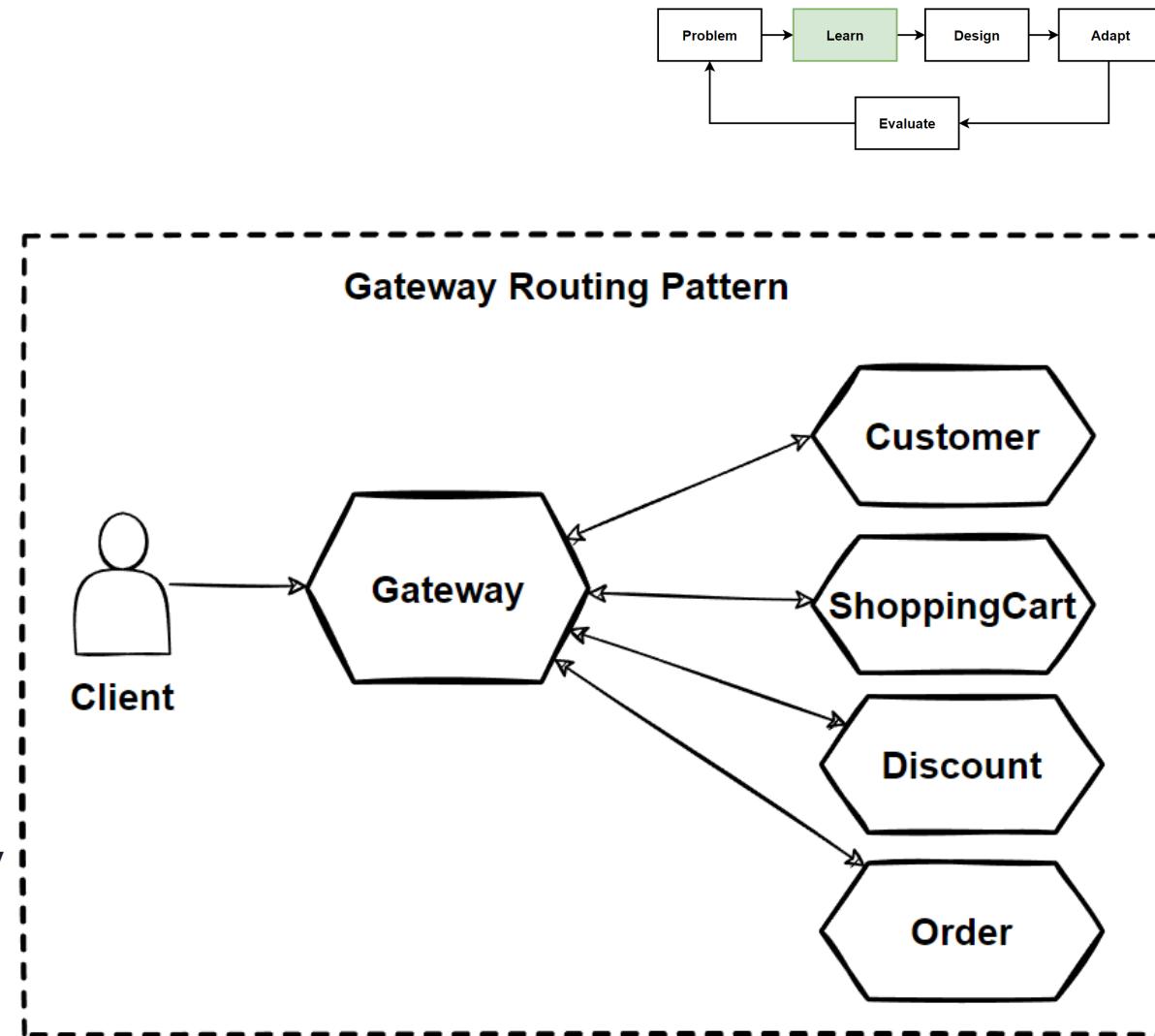
Gateway Routing pattern

- **Route requests to multiple microservices** with exposing a single endpoint.
- Useful when expose multiple services on a **single endpoint** and **route them to internal backend microservices** based on the request.
- The client needs to **consume several microservices**, **Gateway Routing pattern** offers to create a new endpoint that handle the request and route this request for each service.
- **E-commerce application** might provide services such as search Customers, Shopping cart, discount, and order history.
- **If one of microservices are changed**, the **Client doesn't know anything** and not need to change any code on client side, the only changes will be configuration routing changes.



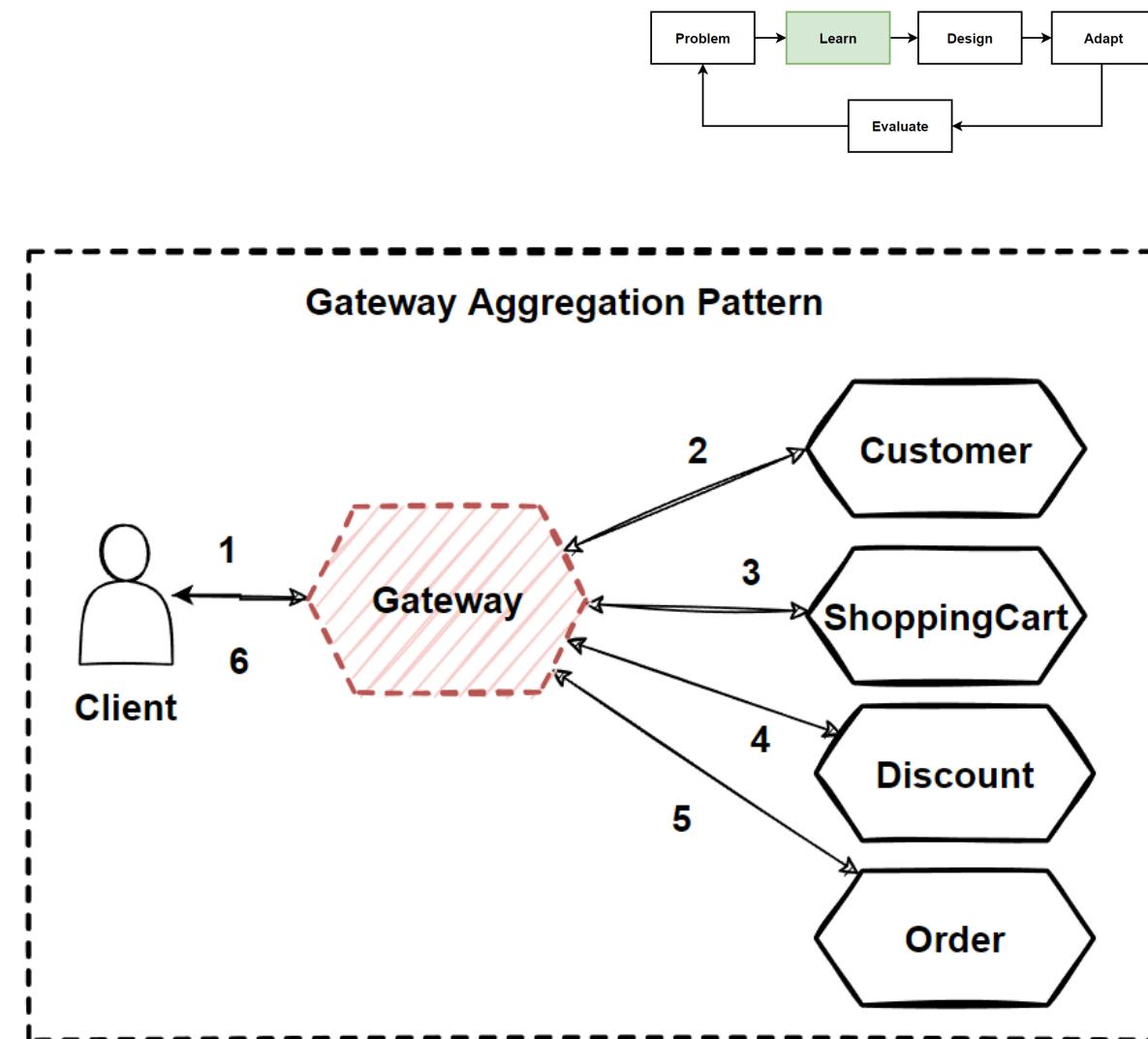
Gateway Routing pattern -2

- It **abstracts your backend microservices** from the client applications, with allowing to **keep client codes simple**, even backend services are changed on behind the API Gateway.
- **Deploy microservices APIs** with **blue/green or canary deployments** with multiple API versions of same microservices, you can **gradually shift your request** to new API with using the Gateway Routing Pattern.
- **Gateway Routing Pattern** gives you the **flexibility** to use different API versions routing to the requests.
- If new version of API has got exceptions, it can be quickly **reverted back** by making only a configuration change.
- Use **Application Layer 7 routing** to route the request to the internal services.

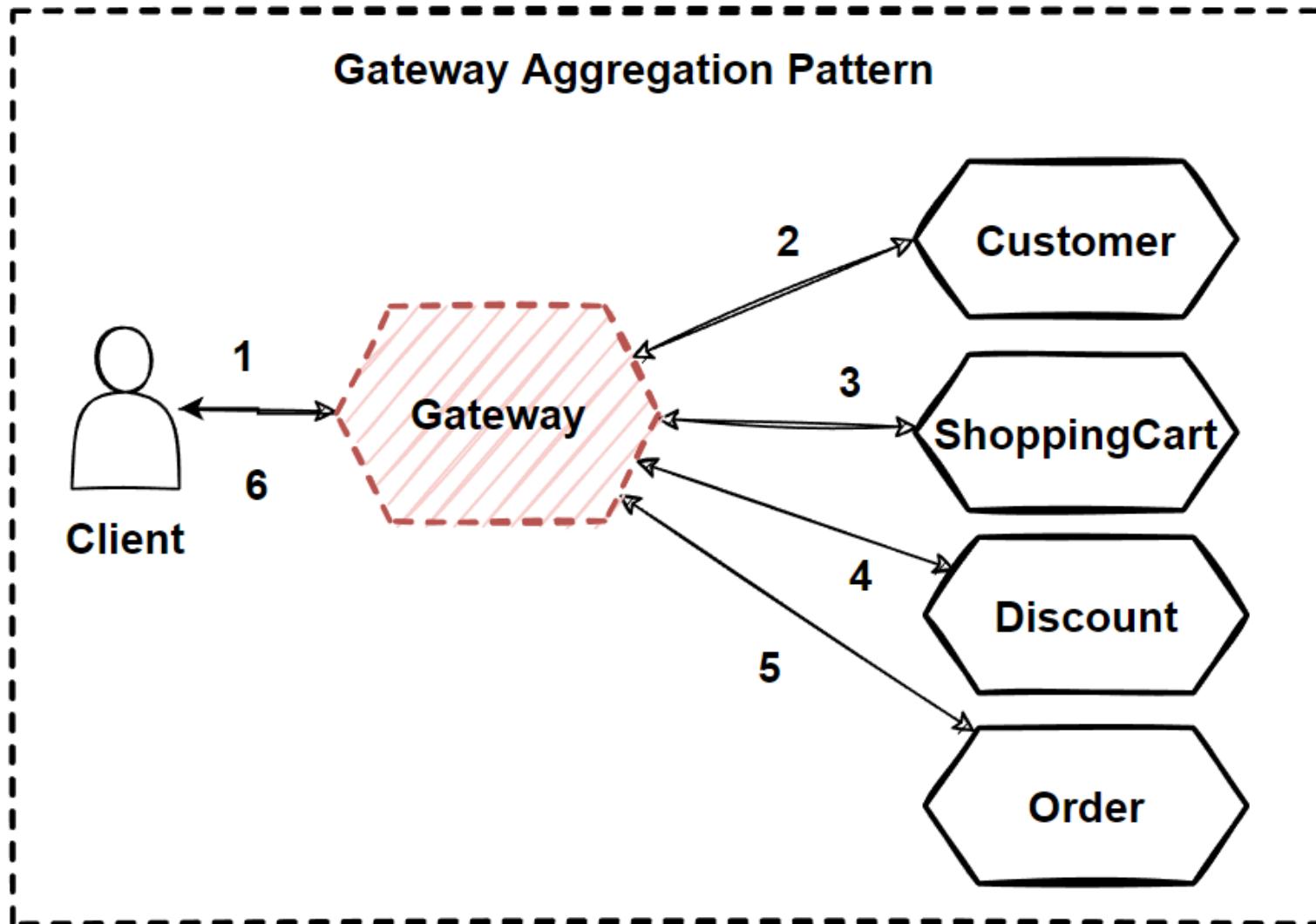
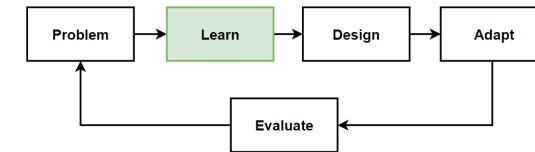


Gateway Aggregation Pattern

- API Gateway service that provide to **aggregate multiple individual requests** towards to internal microservices with **exposing a single request** to the client.
- Use if client application have to **invoke several different** backend microservices to perform its logic.
- Client applications need to **make multiple calls** to different backend microservices, this leads to **chattiness communications** and impact the performance.
- If **new service added** into use cases, then client need to send **additional request** that increase the network calls and latency.
- To **abstract complex** internal backend service communication from the client applications.
- **Dispatches requests** to the several backend services, **then aggregates** the results and sends them back to the requesting client.

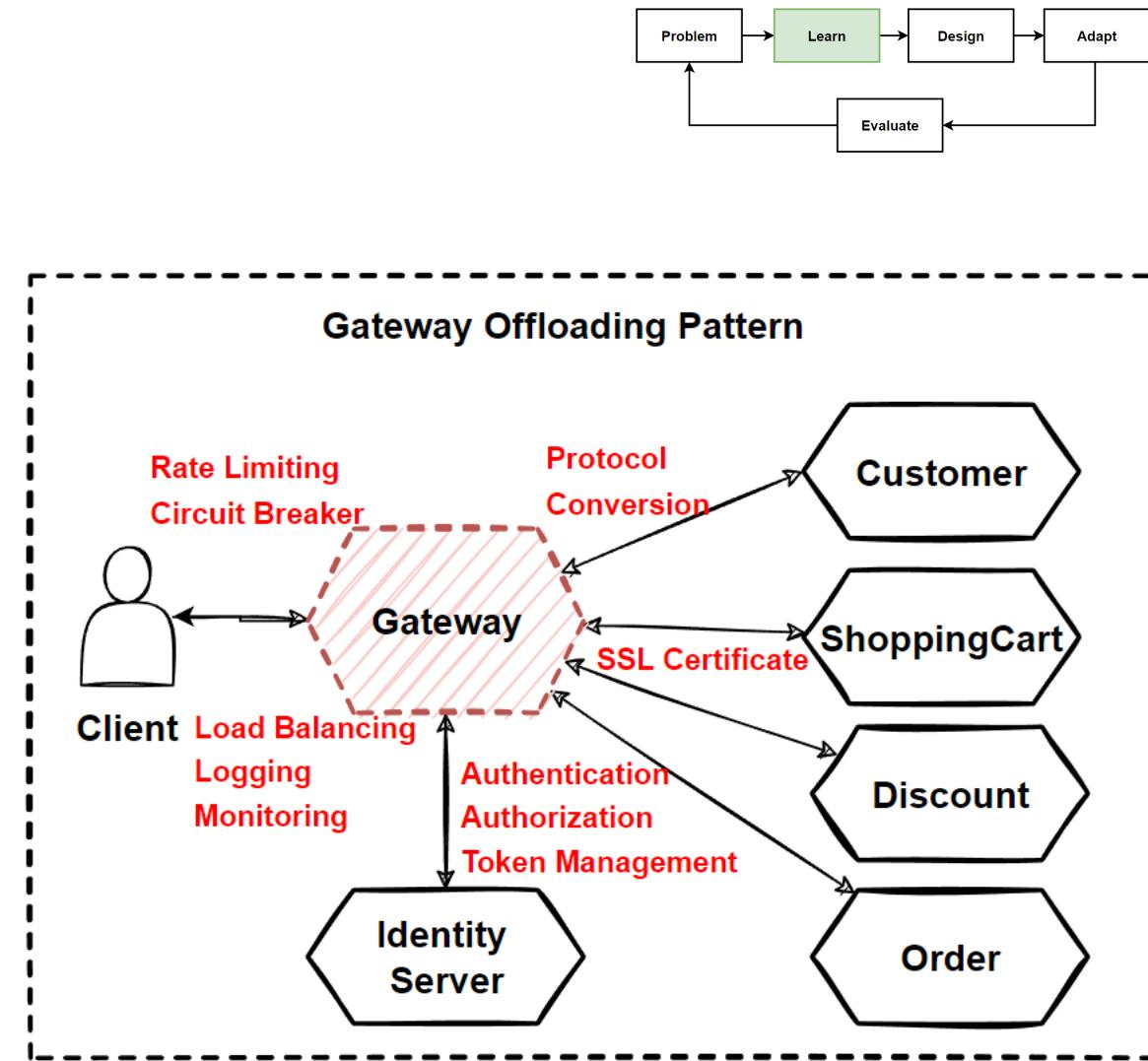


Gateway Aggregation Pattern-2

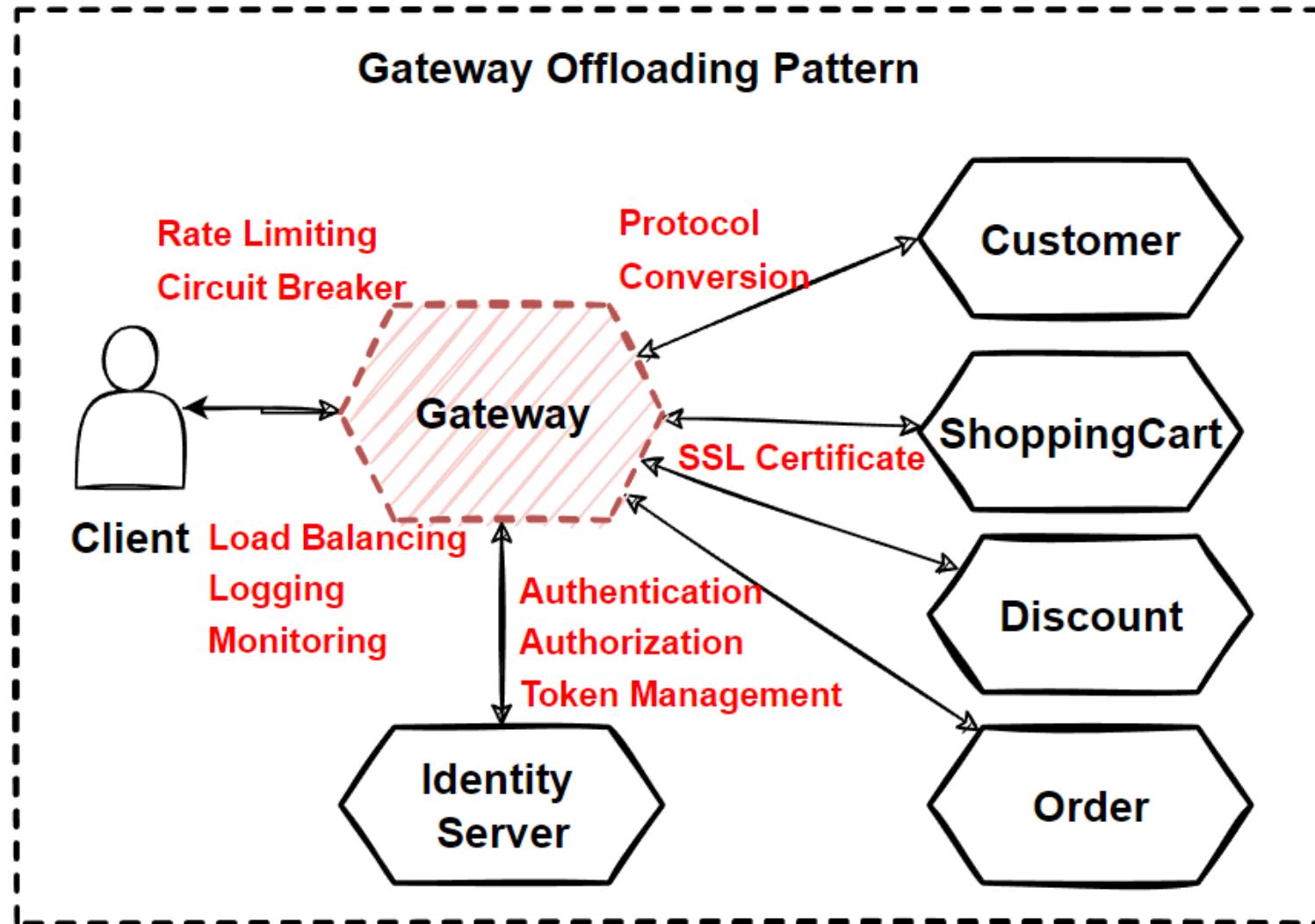
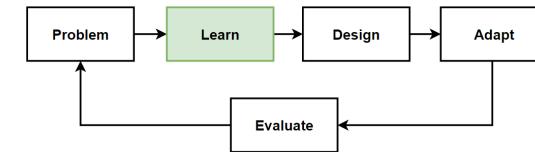


Gateway Offloading Pattern

- **Gateway Offloading Pattern** offers to combine commonly used shared functionalities into a gateway proxy services.
- **Shared functionalities** can application development by moving shared service functionality into centralized places.
- Authentication, Authorization, Rate Limiting, SSL certifications, Logging, Monitoring, Load Balancing.
- **Implementing cross-cutting concerns** for every microservice is **not a good solution**.
- Gateway Offloading Pattern offers to manage all those **Cross-cutting Concerns** into **API Gateways**.
- **Simplify the development of microservices** by removing the cross-cutting concerns on services to maintain supporting resources, allow to **focus on the application** functionality.

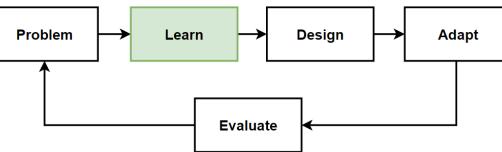
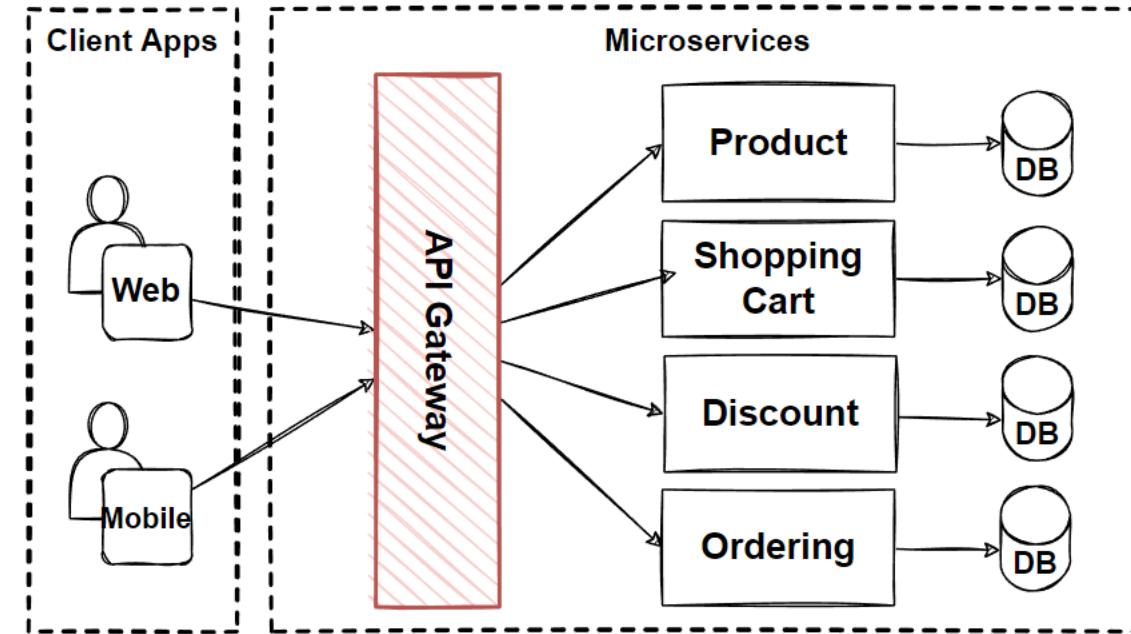


Gateway Offloading Pattern -2



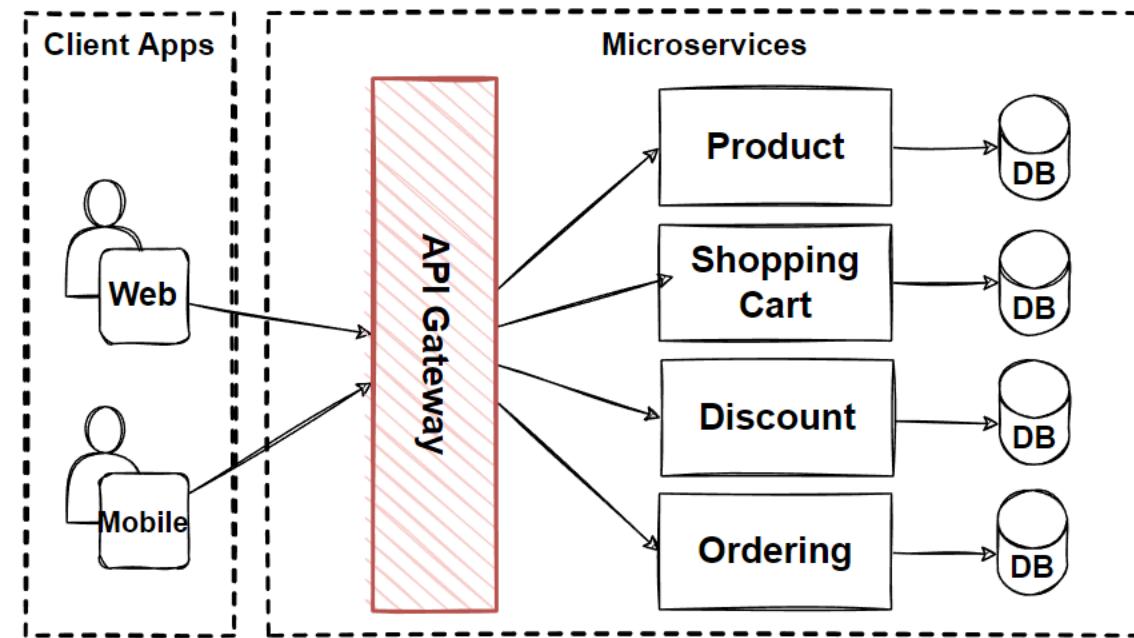
API Gateway Pattern

- API Gateway is a **single point of entry** to the client applications, sits between the **client** and **multiple backend**.
- API Gateways **manage routing** to internal microservices and able to **aggregate several microservice** request in 1 response and handle **cross-cutting concerns**.
- Recommended when design **complex large** microservices-based applications with **multiple client** applications.
- Similar to the **facade pattern** from object-oriented design, but it is a distributed system **reverse proxy or gateway routing** for using in synchronous communication.
- The pattern provides a **reverse proxy to redirect or route requests** to your internal microservices endpoints.
- API Gateway provides a **single endpoint** for the client applications, and it **maps the requests** to internal microservices.

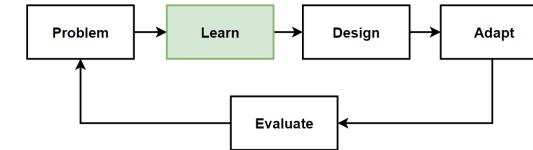


API Gateway Pattern - Summarized

- API gateway **locate between the client apps and the internal microservices.**
- Working as a **reverse proxy** and **routing requests** from clients to backend services and provide **cross-cutting concerns** like authentication, SSL termination, and cache.
- Several client applications connect to single API Gateway possible to **single-point-of-failure risk**.
- If these client applications increase, or adding more logic to **business complexity** in API Gateway, it would be **anti-pattern**.
- Best practices is **splitting the API Gateway** in multiple services or multiple smaller API Gateways: **BFF-Backend-for-Frontend Pattern**.
- Should **segregated based on business boundaries** of the client applications.



Main Features of API Gateway Pattern



- **Reverse Proxy and Gateway Routing**

Reverse proxy to redirect requests to the endpoints of the internal microservices. Using Layer 7 routing for HTTP requests for redirections. Decouple client applications from the internal microservices. Separating responsibilities on network layer and abstracting internal operations.

- **Requests Aggregation and Gateway Aggregation**

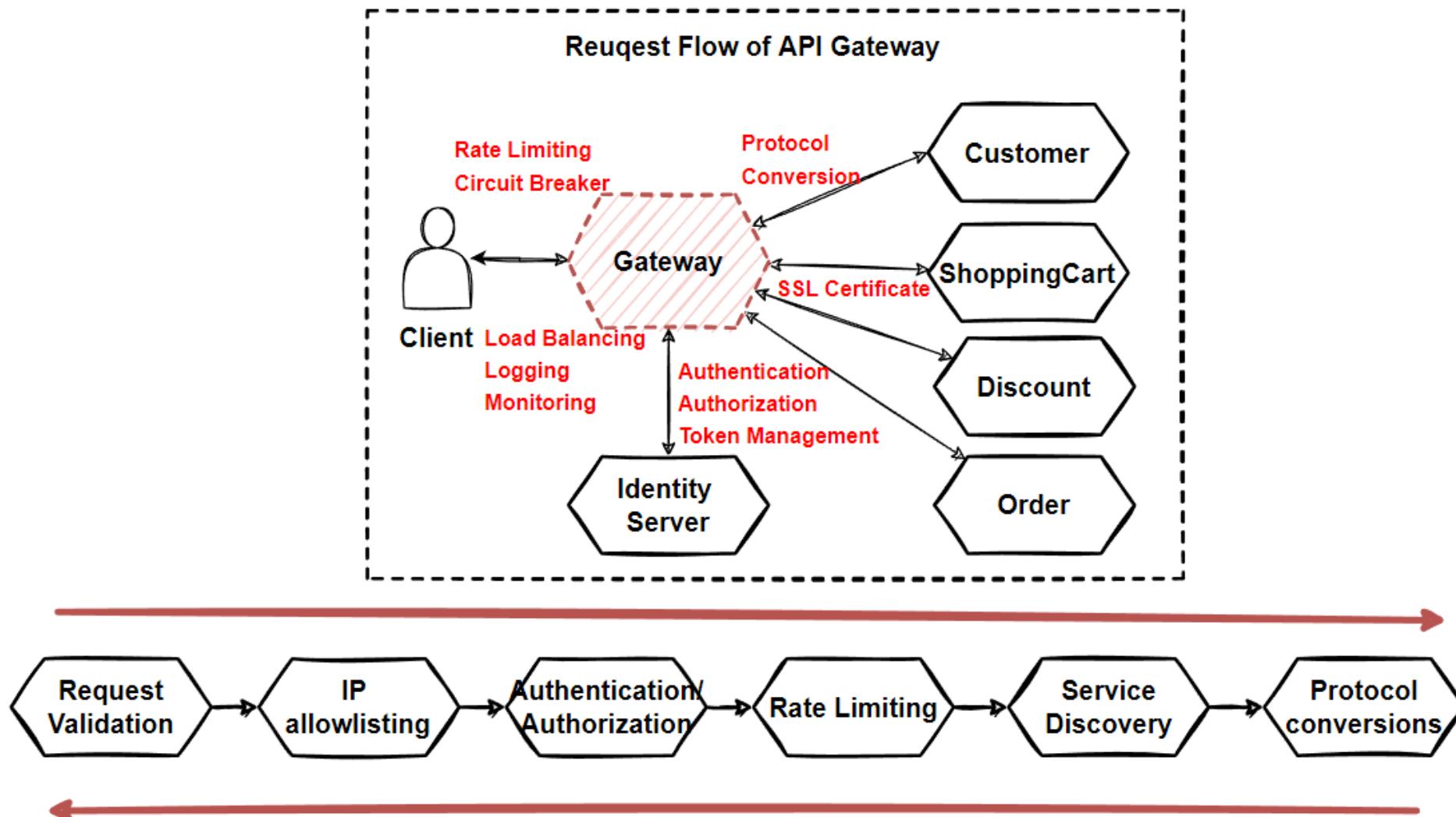
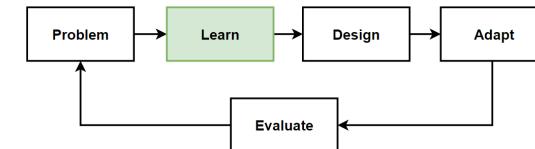
Aggregate multiple internal microservices into a single client request. Client application sends a single request to the API Gateway and it dispatches several requests to the internal microservices and then aggregates the results and sends back to the client application in 1 single response. Reduce chattiness communication.

- **Cross-cutting Concerns and Gateway Offloading**

Best practice to implement cross-cutting functionality on the API Gateways. Cross-cutting functionalities can be; Authentication and authorization, Service discovery, Response caching, Retry policies, Circuit Breaker, Rate limiting and throttling, Load balancing, Logging, tracing, IP allowlisting.

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

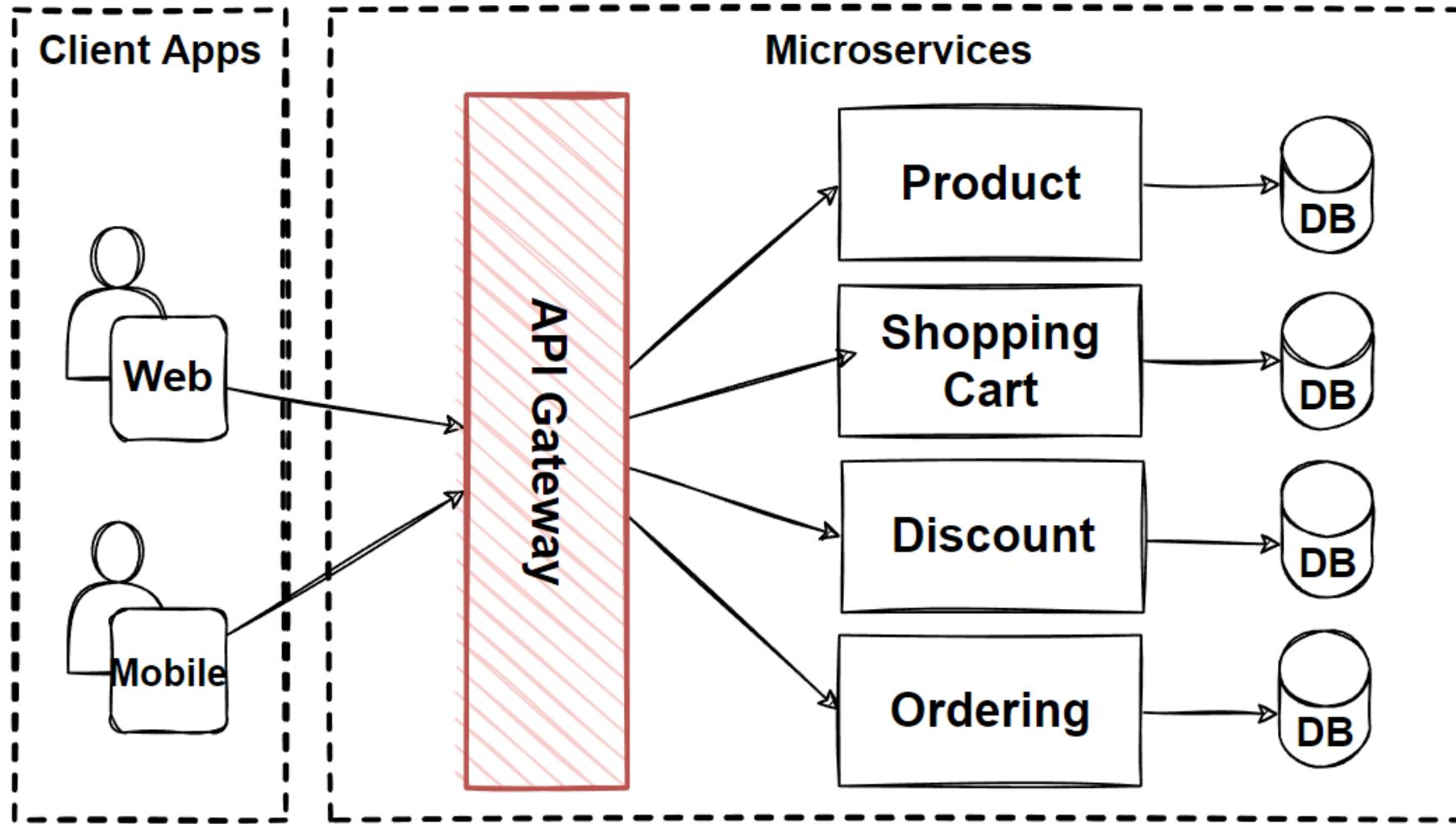
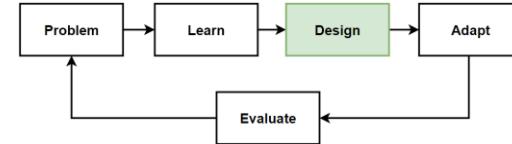
Request Flow of API Gateway Pattern



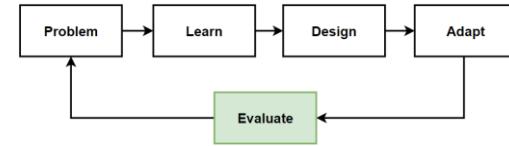
Before Design – What we have in our design toolbox ?

Architectures	Patterns&Principles	Microservices Communications	Non-FR	FR
<ul style="list-style-type: none">• Microservices Architecture• The Database-per-Service Pattern• Polygot Persistence• Decompose services by scalability• The Scale Cube• Microservices Decomposition Pattern• Microservices Communications Patterns	<ul style="list-style-type: none">• HTTP Based RESTful API• GraphQL API• gRPC API• WebSocket API• Gateway Routing Pattern• Gateway Aggregation Pattern• Gateway Offloading Pattern• API Gateway Pattern	<ul style="list-style-type: none">• High Scalability• High Availability• Millions of Concurrent User• Independent Deployable• Technology agnostic• Data isolation• Resilience and Fault isolation		<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history

Design: Microservices Architecture with API Gateway



Evaluate: Microservice Architecture with API Gateway

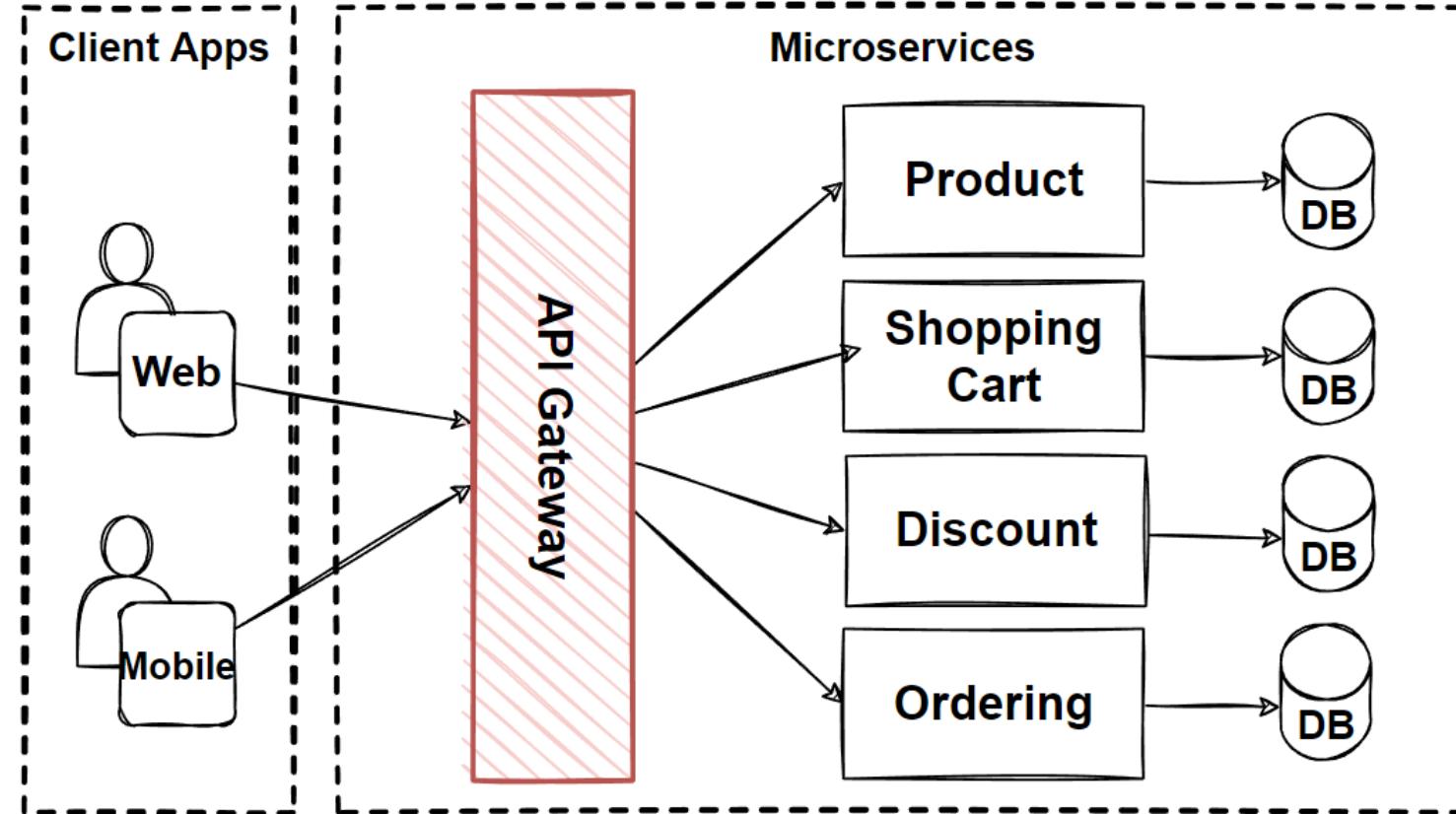


Benefits

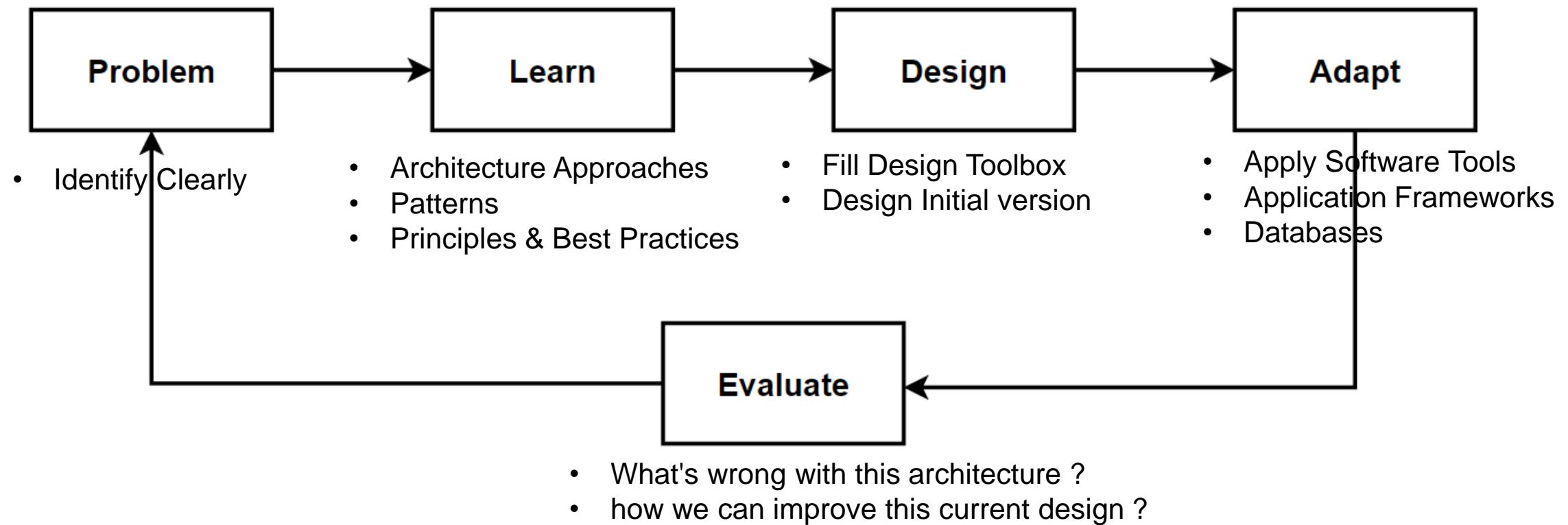
- Reverse proxy, Routing, Aggregation
- Abstraction Backend services
- Reduces chattiness network traffic
- Implements Cross-cutting concerns

Drawbacks

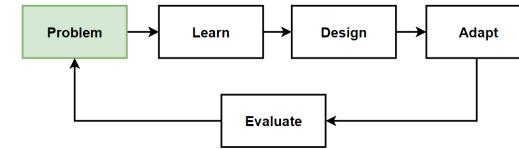
- Single API Gateway = single-point-of-failure risk
- Business complexity in API Gateway
- If not scale out, API Gateway can become a bottleneck on network traffic



Way of Learning – The Course Flow



Problem: Client Apps has Different UI Requirements

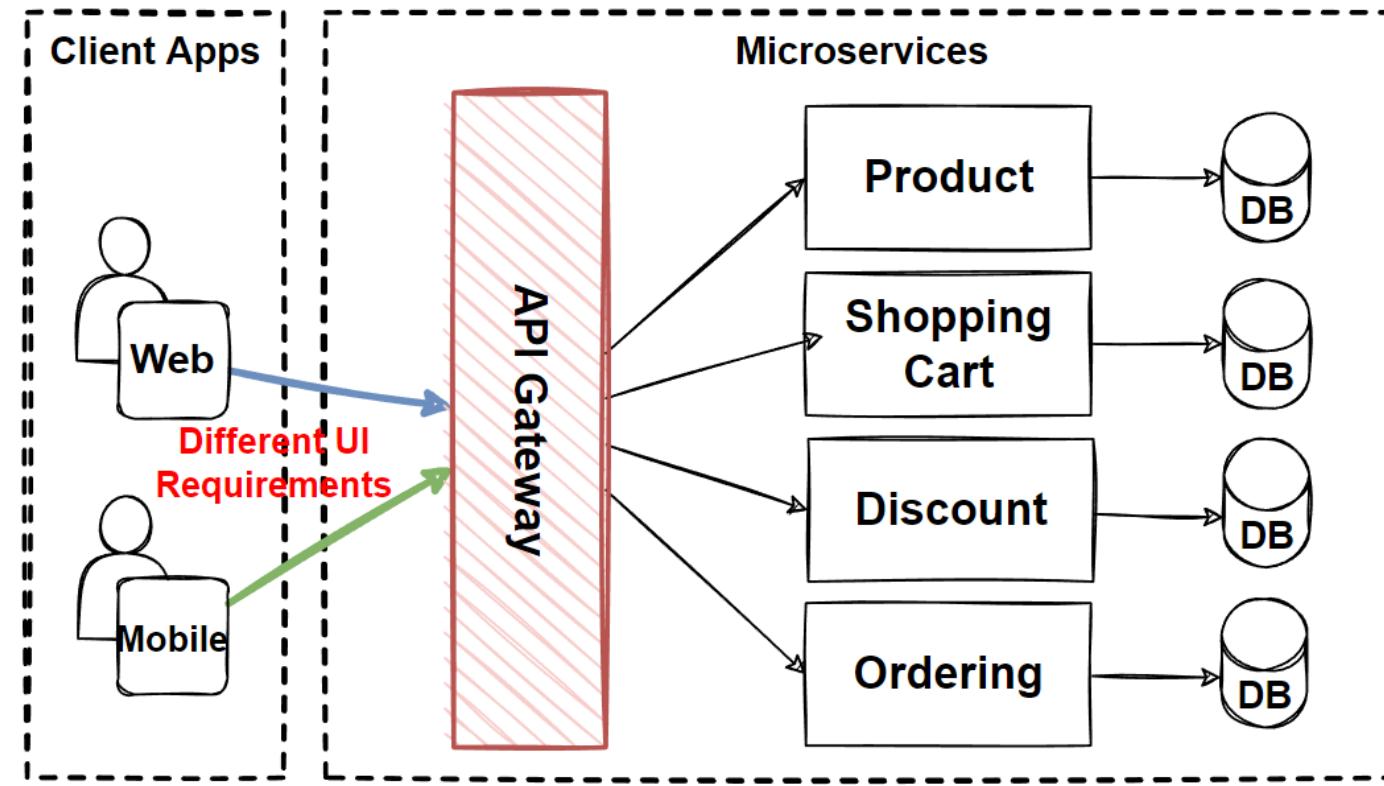


Problems

- Frontend client applications has different UI and Data consumption requirements
- Different data representation requirements need extra effort to adapt data for specific Client UI
- Clients code has custom logic to re-format these data.
- Client codes complex and hard to maintain for future developments. Also it increase browser's usage of high resources.

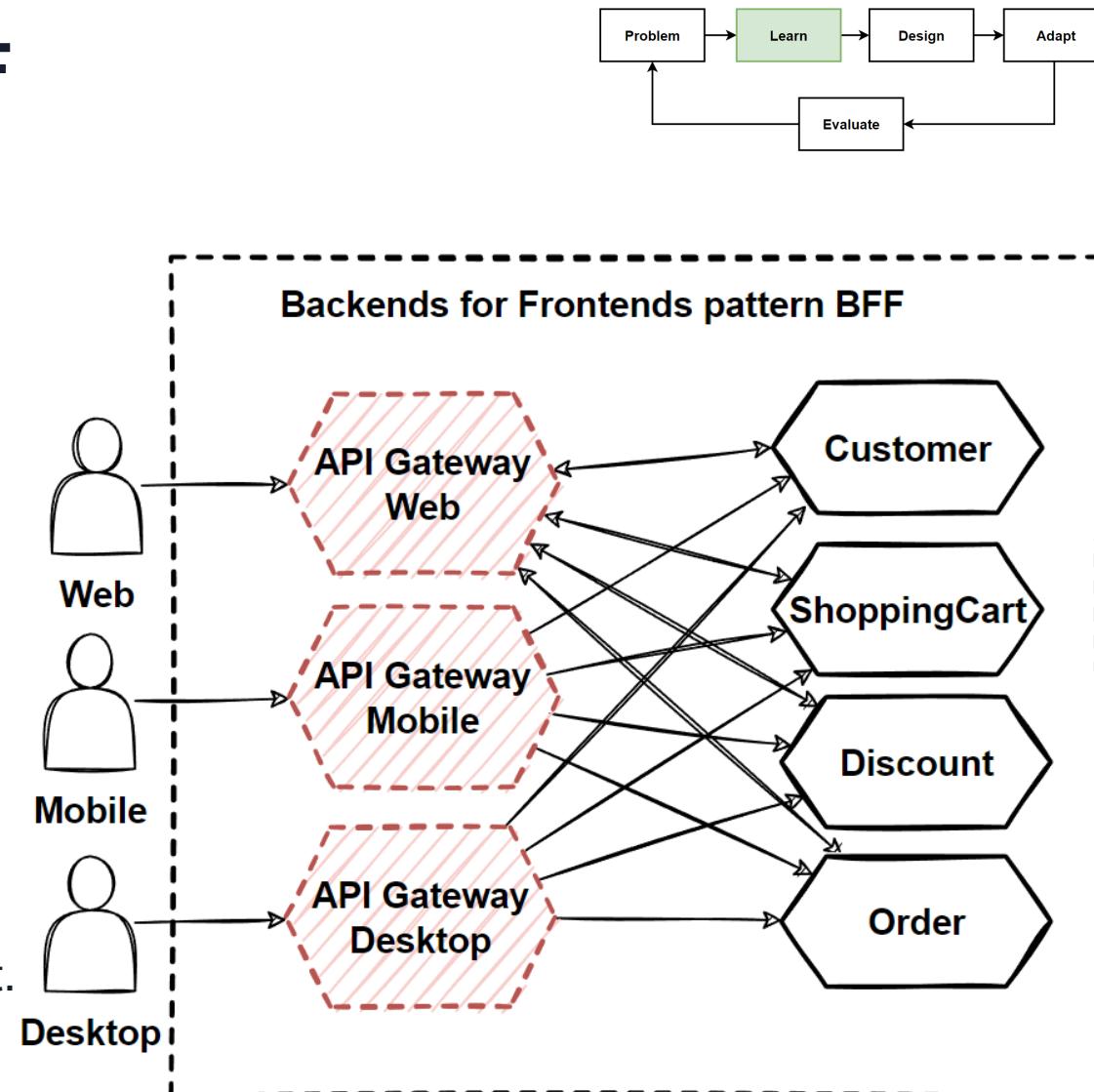
Solutions

- Client focused Interfaces
- **BFF-Backend for Frontend Pattern**

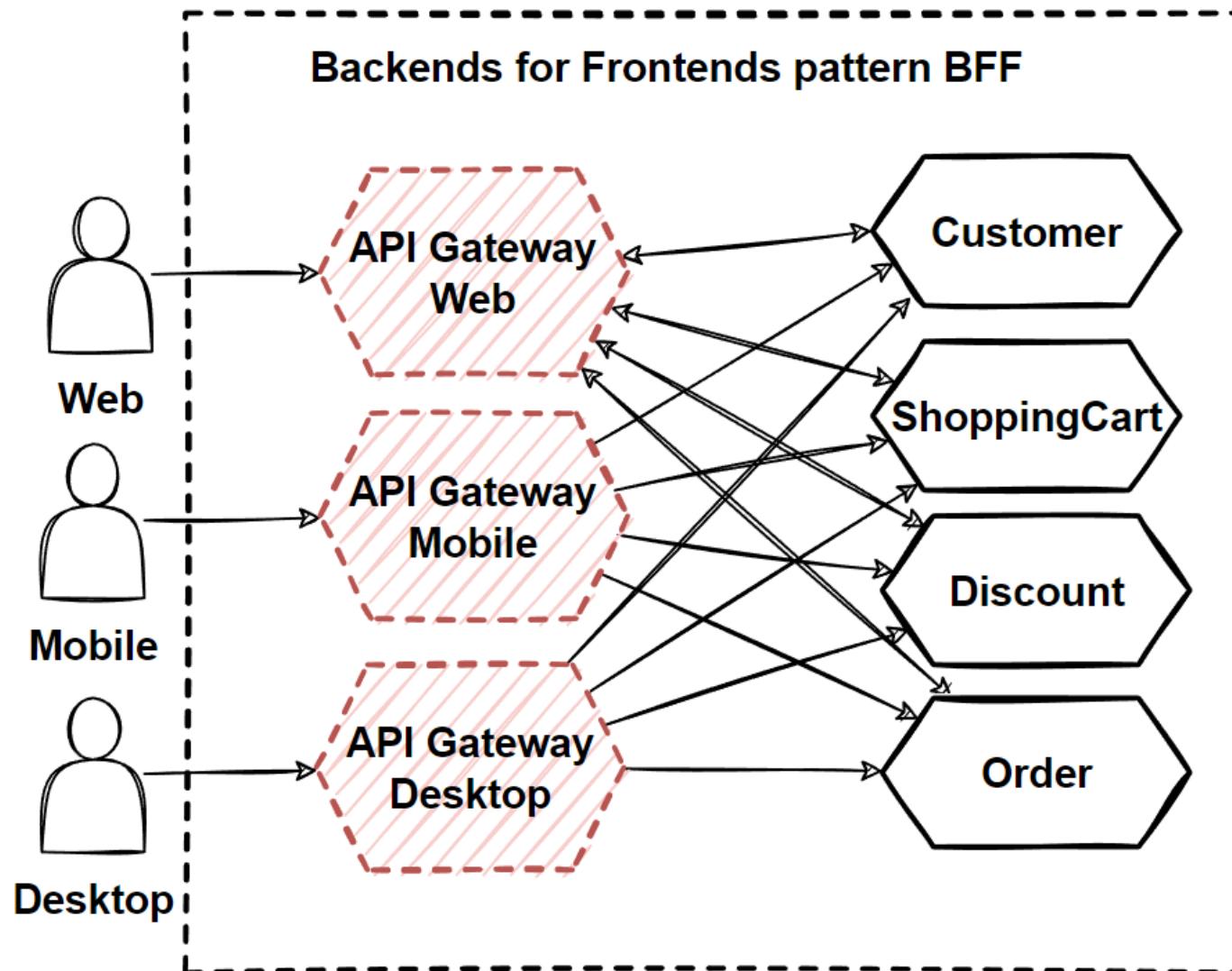
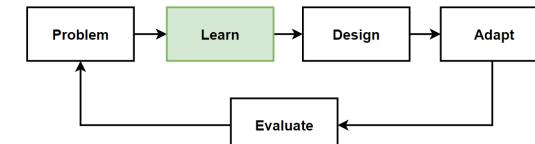


Backends for Frontends Pattern - BFF

- Backends for Frontends pattern basically **separate API Gateways** as per the **specific frontend** applications.
- Single API Gateway makes a **single-point-of failure**.
- BFF offers to create several API Gateways and **grouping the client applications** according to their boundaries.
- A **single and complex API Gateway** can be **risky** and becoming a **bottleneck** into your architecture.
- Larger systems often expose multiple API Gateways by **grouping client type** like mobile, web and desktop functionality.
- **Create several API Gateways** as per user interfaces to provide to best match the needs of the frontend environment.
- **BFF Pattern** is to provide client applications has a **focused interfaces** that connects with the internal microservices.



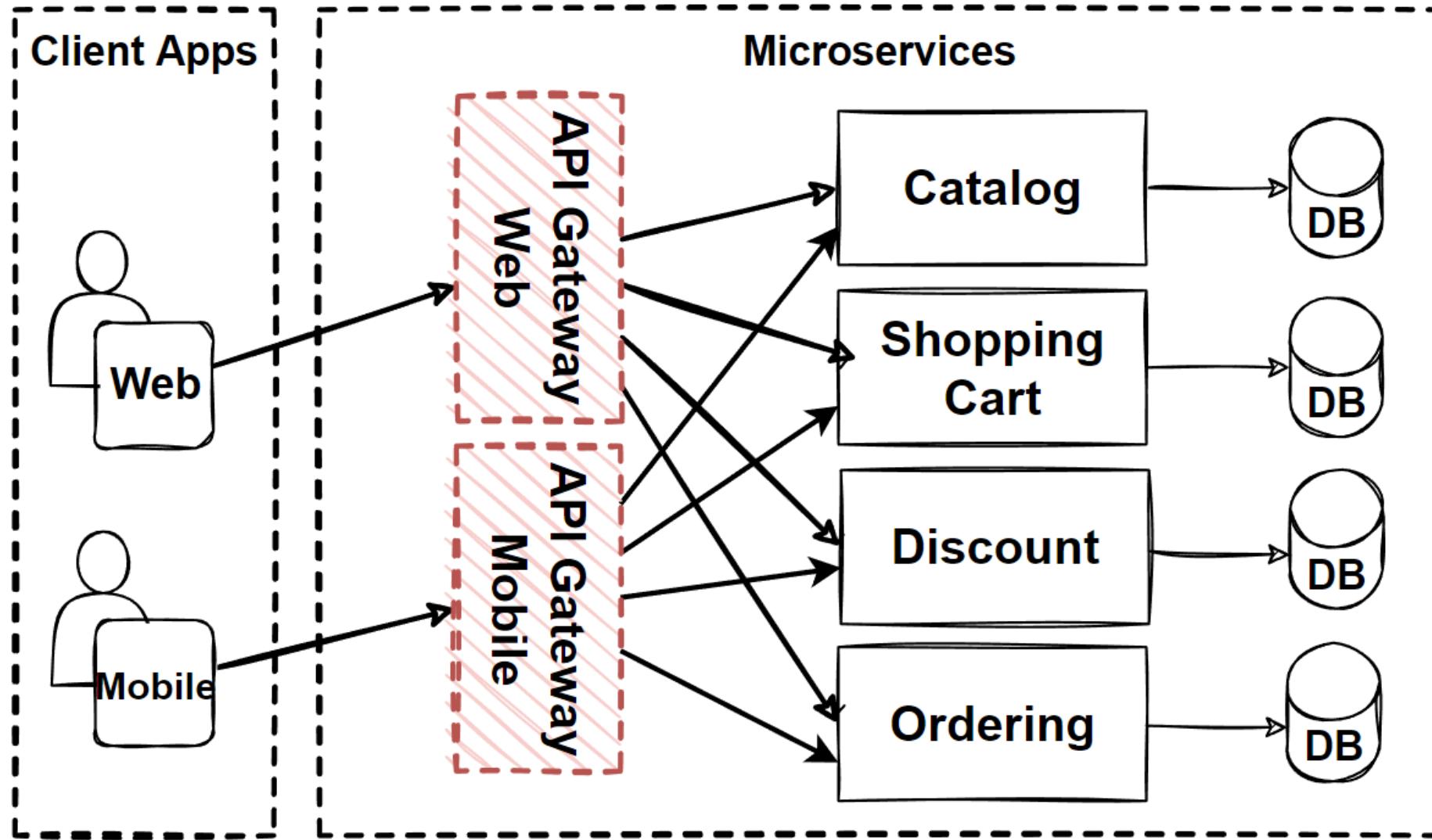
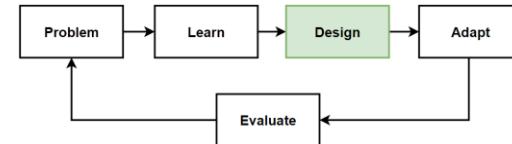
Backends for Frontends Pattern - BFF



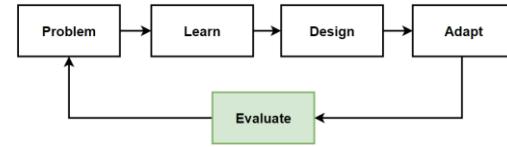
Before Design – What we have in our design toolbox ?

Architectures	Patterns&Principles	Microservices Communications	Non-FR	FR
<ul style="list-style-type: none">• Microservices Architecture• The Database-per-Service Pattern• Polygot Persistence• Decompose services by scalability• The Scale Cube• Microservices Decomposition Pattern• Microservices Communications Patterns	<ul style="list-style-type: none">• The Database-per-Service Pattern• Polygot Persistence• Decompose services by scalability• The Scale Cube• Microservices Decomposition Pattern• Microservices Communications Patterns	<ul style="list-style-type: none">• HTTP Based RESTful API• GraphQL API• gRPC API• WebSocket API• Gateway Routing Pattern• Gateway Aggregation Pattern• Gateway Offloading Pattern• API Gateway Pattern• Backends for Frontends Pattern-BFF	<ul style="list-style-type: none">• High Scalability• High Availability• Millions of Concurrent User• Independent Deployable• Technology agnostic• Data isolation• Resilience and Fault isolation	<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history

Design: Microservices Architecture with BFF



Evaluate:Microservice Architecture with BFF

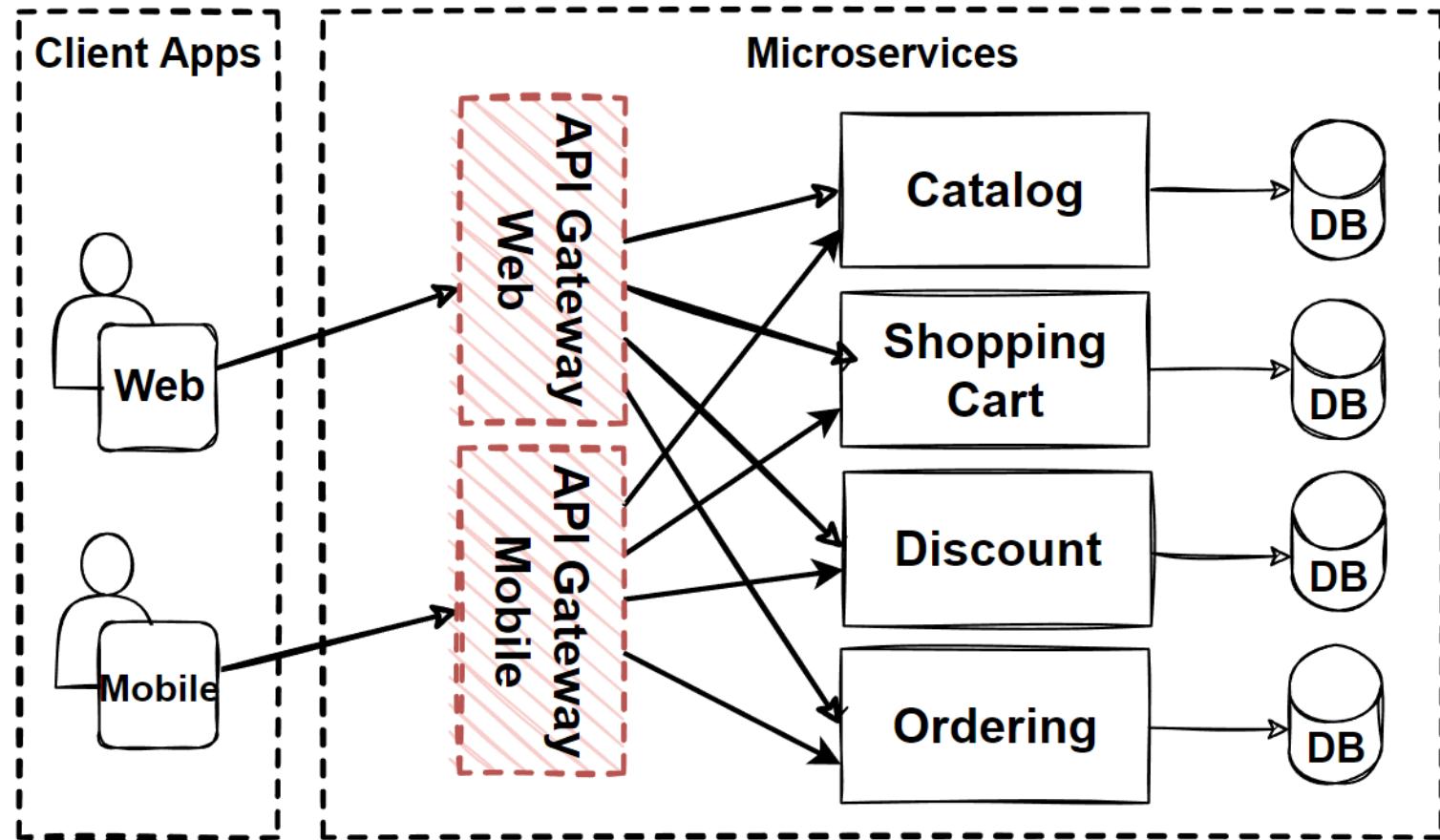


Benefits

- Client focused Interfaces
- Minimal logic on the frontend code
- Streamline data representation
- Well-focused interface for the frontend client applications

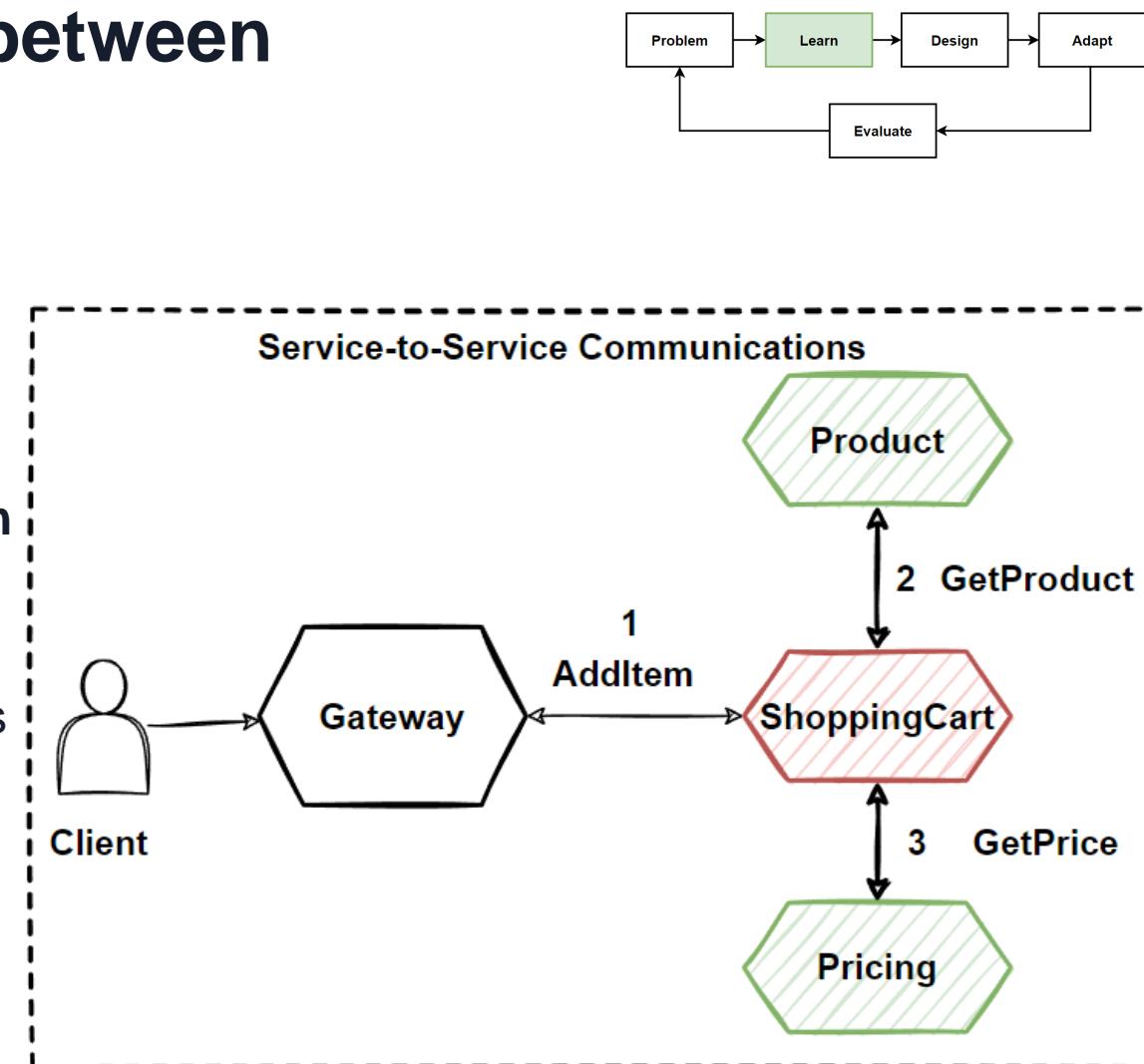
Drawbacks

- Increased Latency
- Good for large-scaled microservices application which has several client applications



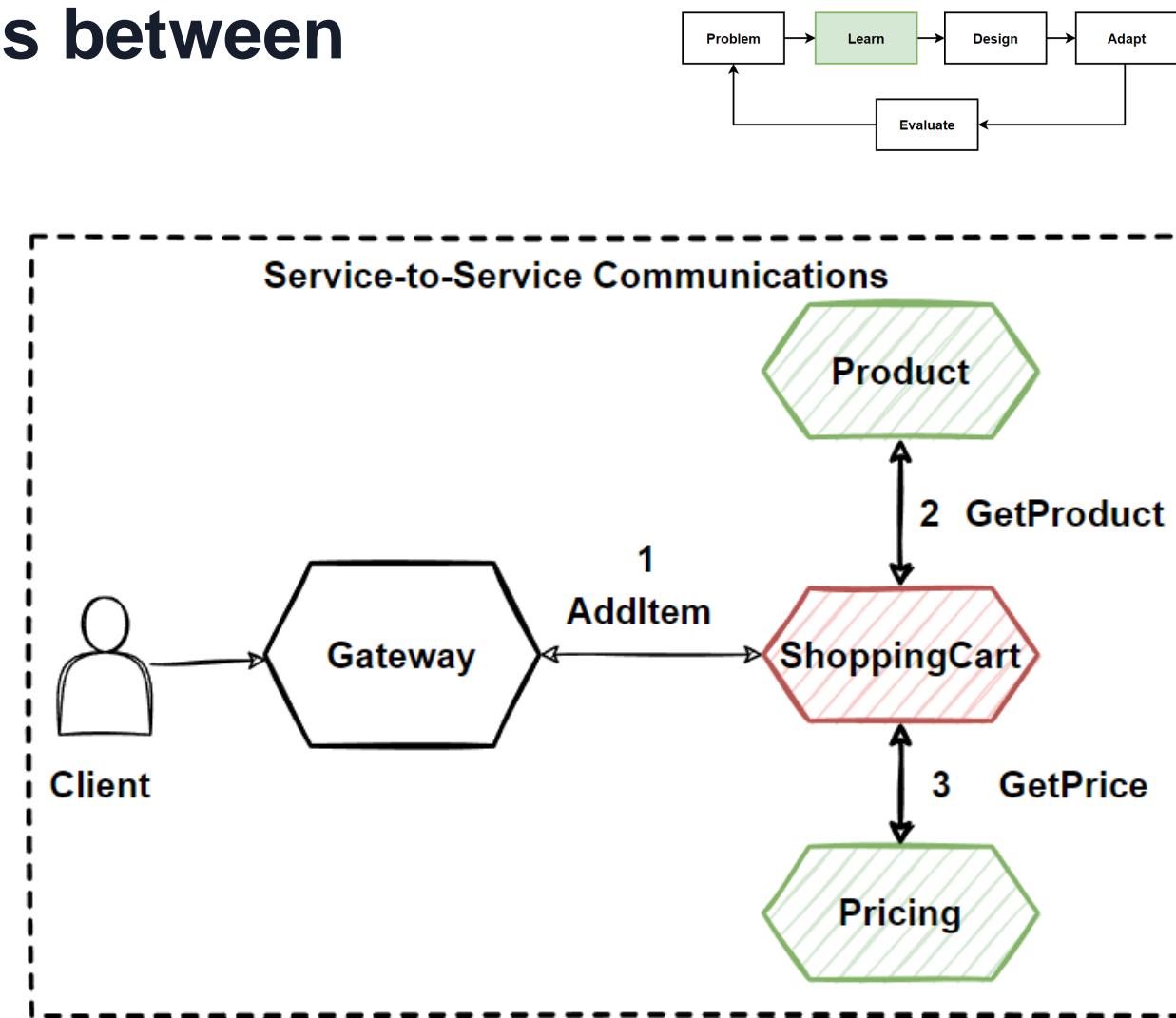
Service-to-Service Communications between Backend Internal Microservices

- Created **API Gateways** and separate this API Gateways following the **BFF Pattern**.
- **Sync request** comes from the clients and goes to internal microservices **over the API Gateways**.
- What if the **client requests** are required to **visit more than one internal microservices** ? How we can manage internal microservice communications ?
- Best practice is **reducing inter-service communication** as much as possible.
- In some cases, we can't reduce these internal communications due to operation **need to visit several internal services**.
- **Client send query request** to internal microservices to accumulate some data in the screen page of frontend.

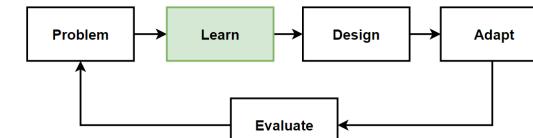


Service-to-Service Communications between Backend Internal Microservices-2

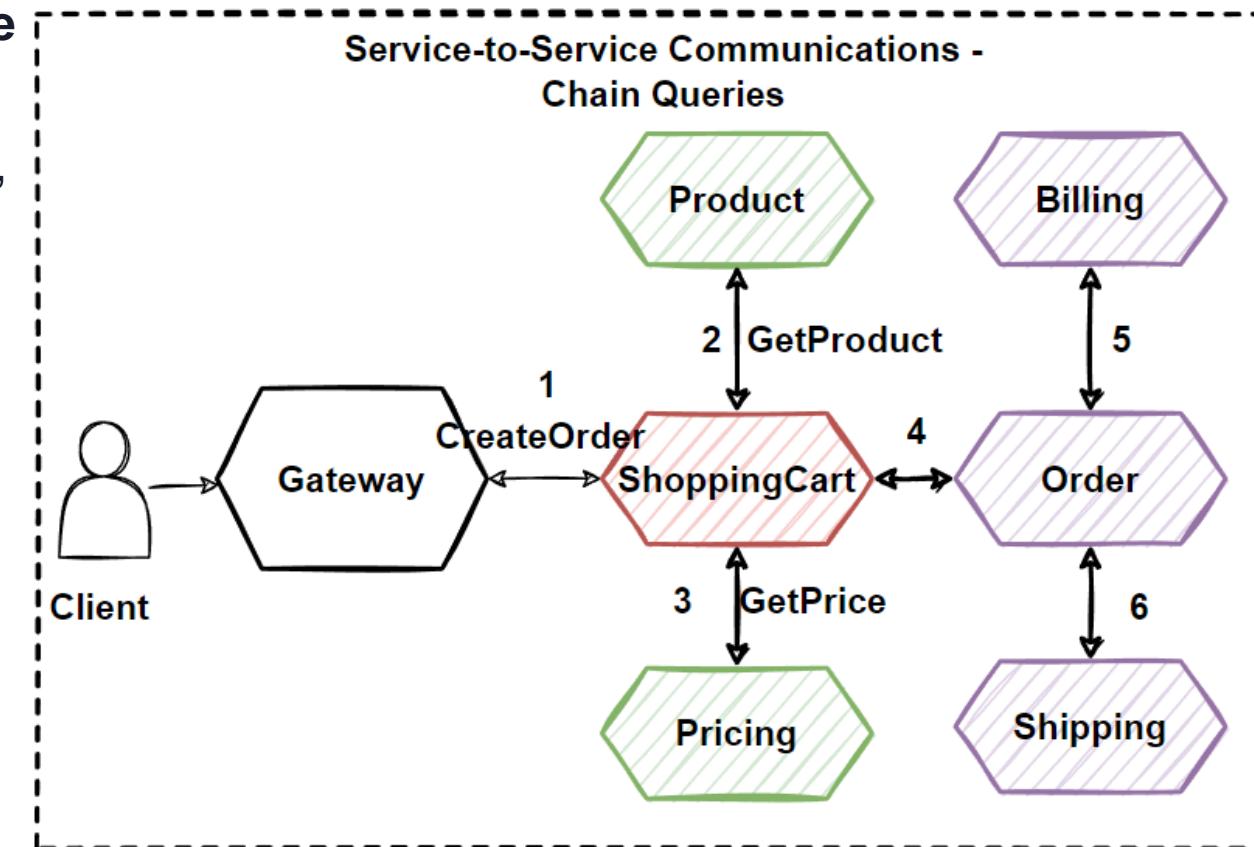
- User **addItem** into shopping cart, these operation need to get data from Product and Price microservices.
- **Synchronous communication** over the API Gateways which is the Request/Response Messaging.
- The client **send http request** to the microservices for querying or adding item into Shopping Cart:
 - 1- The client send request to API Gateway
 - 2- API Gateway dispatch request to ShoppingCart
 - 3- SC need to extend information by sending sync request to Product and Pricing microservices
- **Internal calls** makes **coupling** each microservices.
- If one of the **microservices** is **down**, it can't return data to the client so **its not any fault-tolerance**.



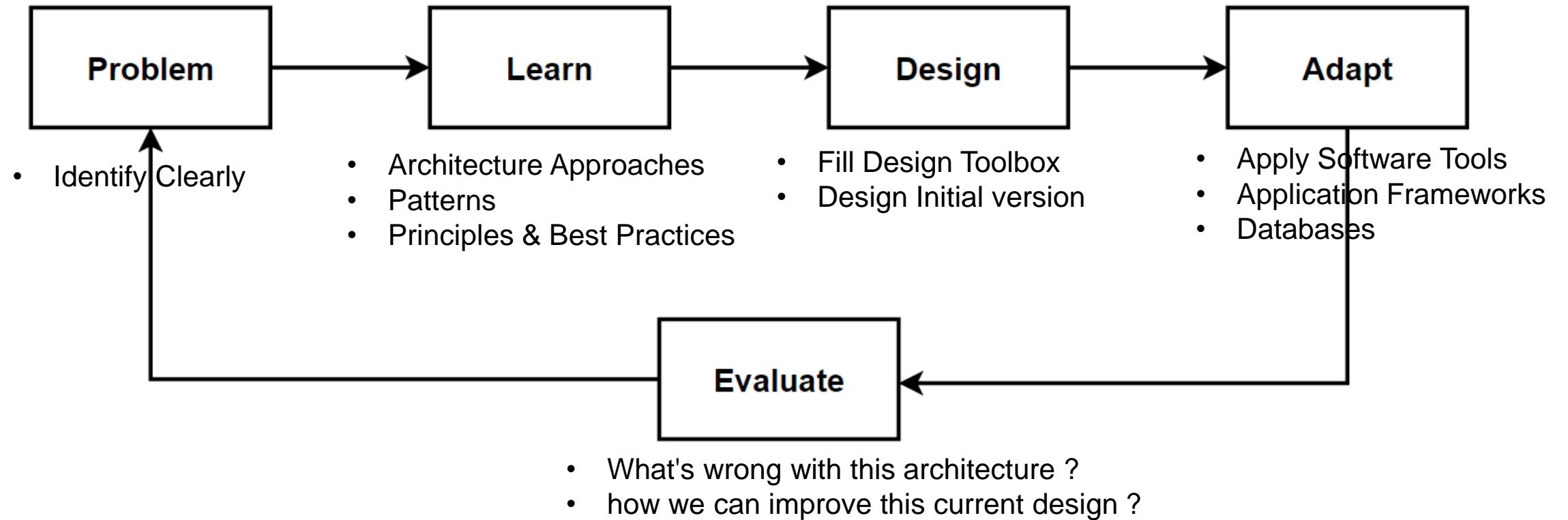
Service-to-Service Communications Chain Queries



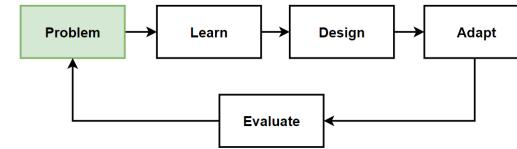
- If **service calls** are much then a few HTTP calls to multiple microservices, than it goes to **un-manageable** situation.
- **Place Order Use Case:** client checkout shopping cart, start order fulfillment processes.
- Request/Response Sync Messaging end up **with 6 sync chain HTTP Request**.
- **Increase latency** and **negatively impact** the performance, scalability, and availability.
- What if the **step 5 or 6 is failed ?**
- We can **apply 2 way** to solve this issues:
 - 1- Change microservices communications to async way
 - 2- Use Service Aggregator Pattern to aggregate some query operations in 1 API Gateway.



Way of Learning – The Course Flow



Problem: Service-to-Service Communications Chain Queries

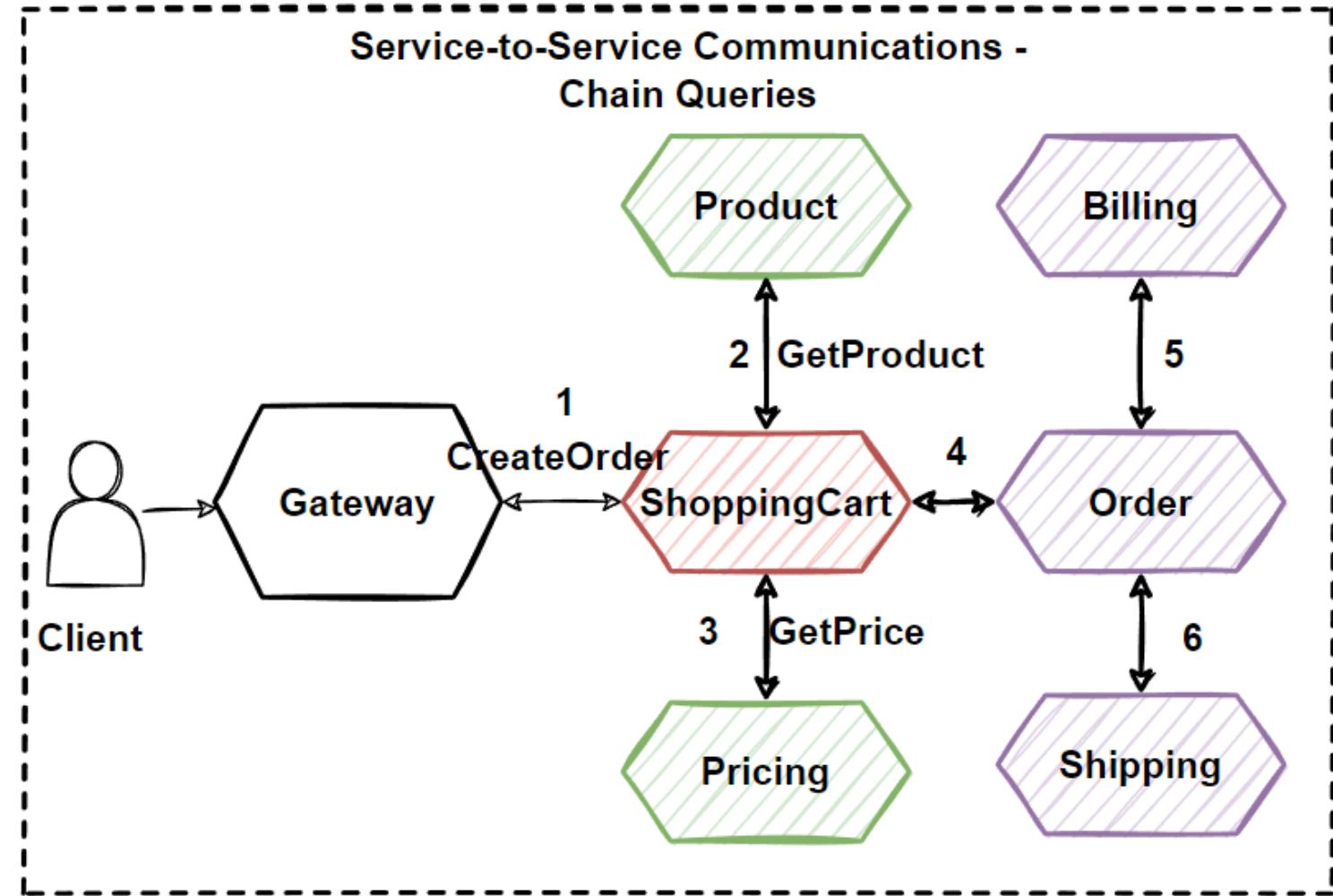


Problems

- HTTP calls to multiple microservices
- Chain Queries
- Visit more than a few microservices
- Increased latency

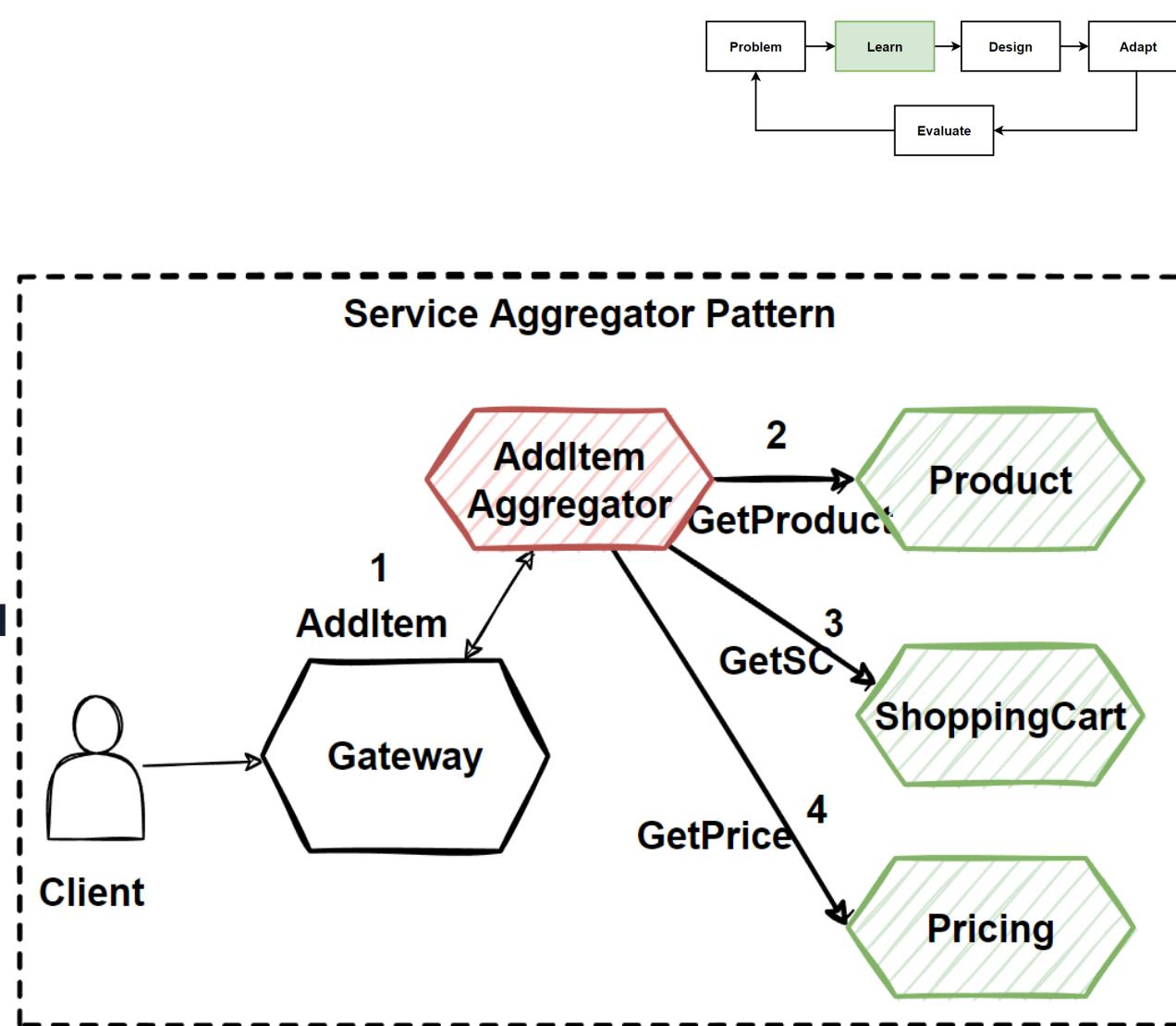
Solutions

- Aggregate query operations
- **Service Aggregator Pattern**



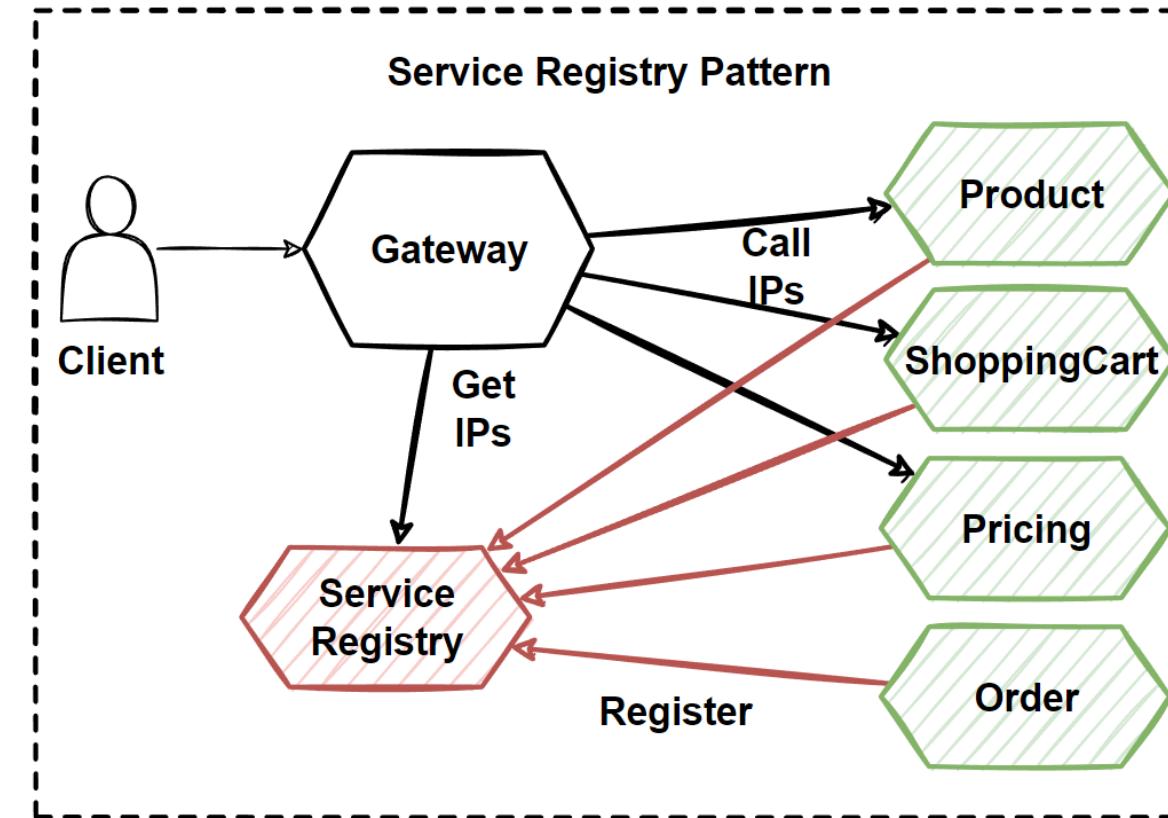
Service Aggregator Pattern

- **Service Aggregator Pattern** receives a request from the client or API Gateway.
- **Dispatches requests** of multiple internal backend microservices.
- **Combines the results** and responds back to the initiating request in 1 response structure.
- **Reduces chattiness and communication overhead** between the client and microservices
- **AddItem aggregates request data** from several back-end microservices: Product - SC and Pricing.
- **Isolates the underlying addItem operation** that makes calls to multiple back-end microservices.
- **Centralizing its logic** into a specialized microservice.



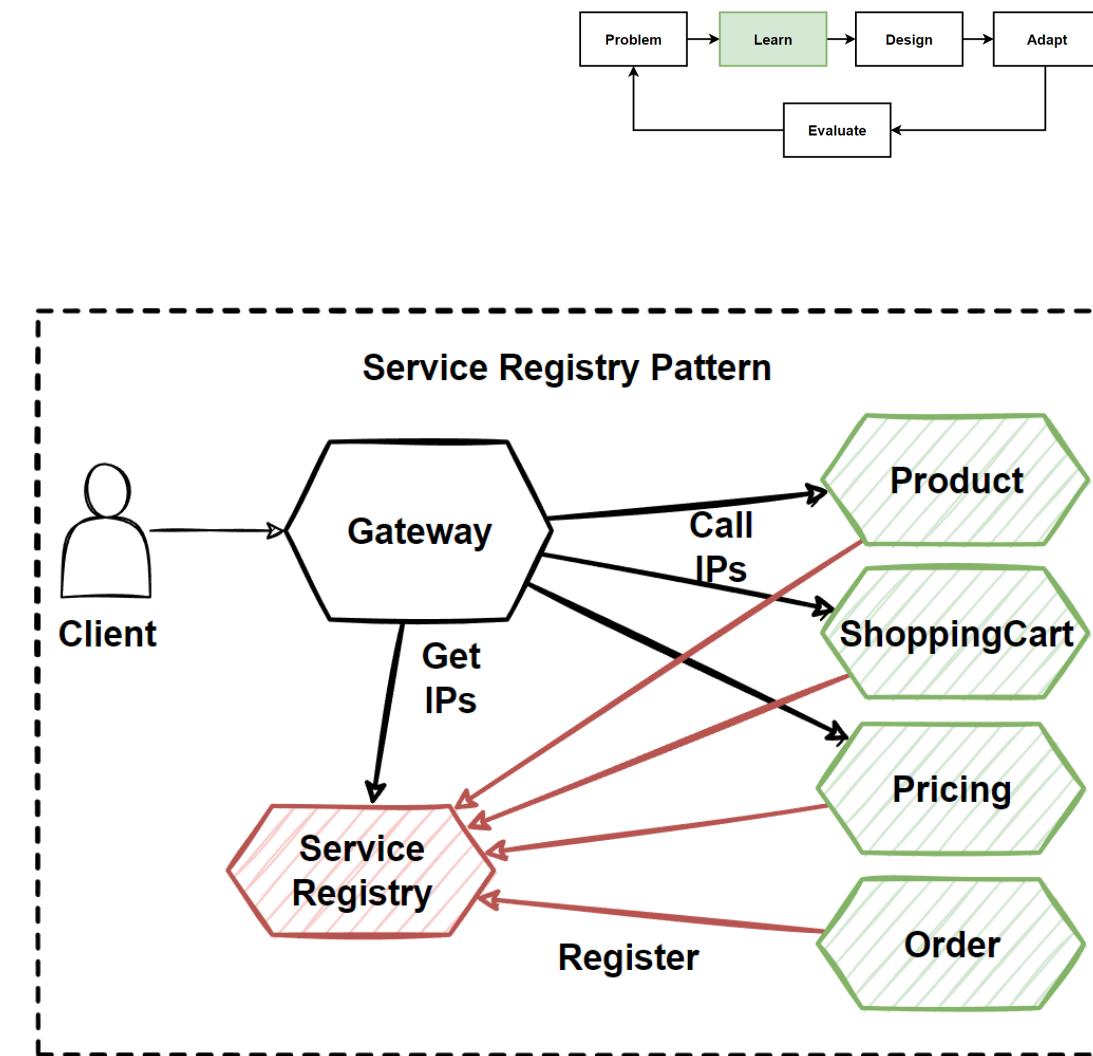
Service Registry/Discovery Pattern

- Microservice **Discovery Patterns** and **Service Registry** provide to register and discover microservices in the cluster.
- **Why We Need Service Discovery Pattern ?**
When deploying and scaling microservices, more microservices and instances could be added to the system to provide scalability to the distributed application.
- **Service locations** can **change frequently**, more dynamic configuration is needed for the microservice architecture.
- **IP Addresses** and **network locations** are **dynamically** assigned and often change due to auto-scaling features.
- **Service Registry** and **Discovery pattern** allows to find the network locations of microservices without injecting or coupling services.



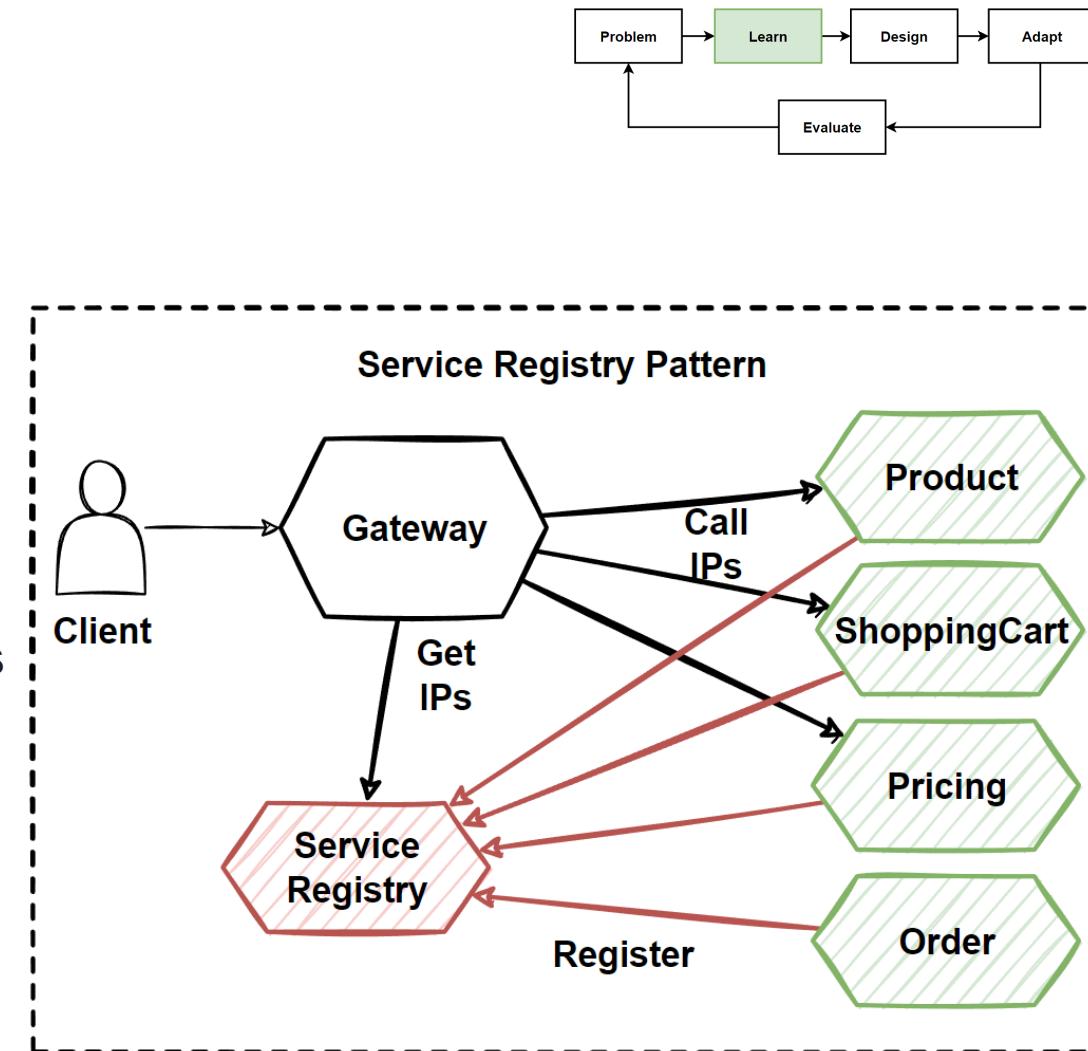
What is Service Discovery Pattern?

- To find the **network locations** of microservices, the **service discovery pattern** is used in microservices applications.
- It will provide to **register and discover microservices** in the cluster.
- **API Gateways** for **routing the traffic** with client and internal microservices. How API Gateways access the internal backend microservices ?
- **Service discovery pattern** uses a **centralized server** for «service registry» to maintain a central view of microservices network locations.
- Services **update their locations** in the **service registry** at fixed intervals. Clients can connect to the service registry and fetch the locations of microservices.
- There are **2 main service discovery patterns**:
 - Client-side service discovery
 - Server-side service discovery



How Does Service Discovery Work?

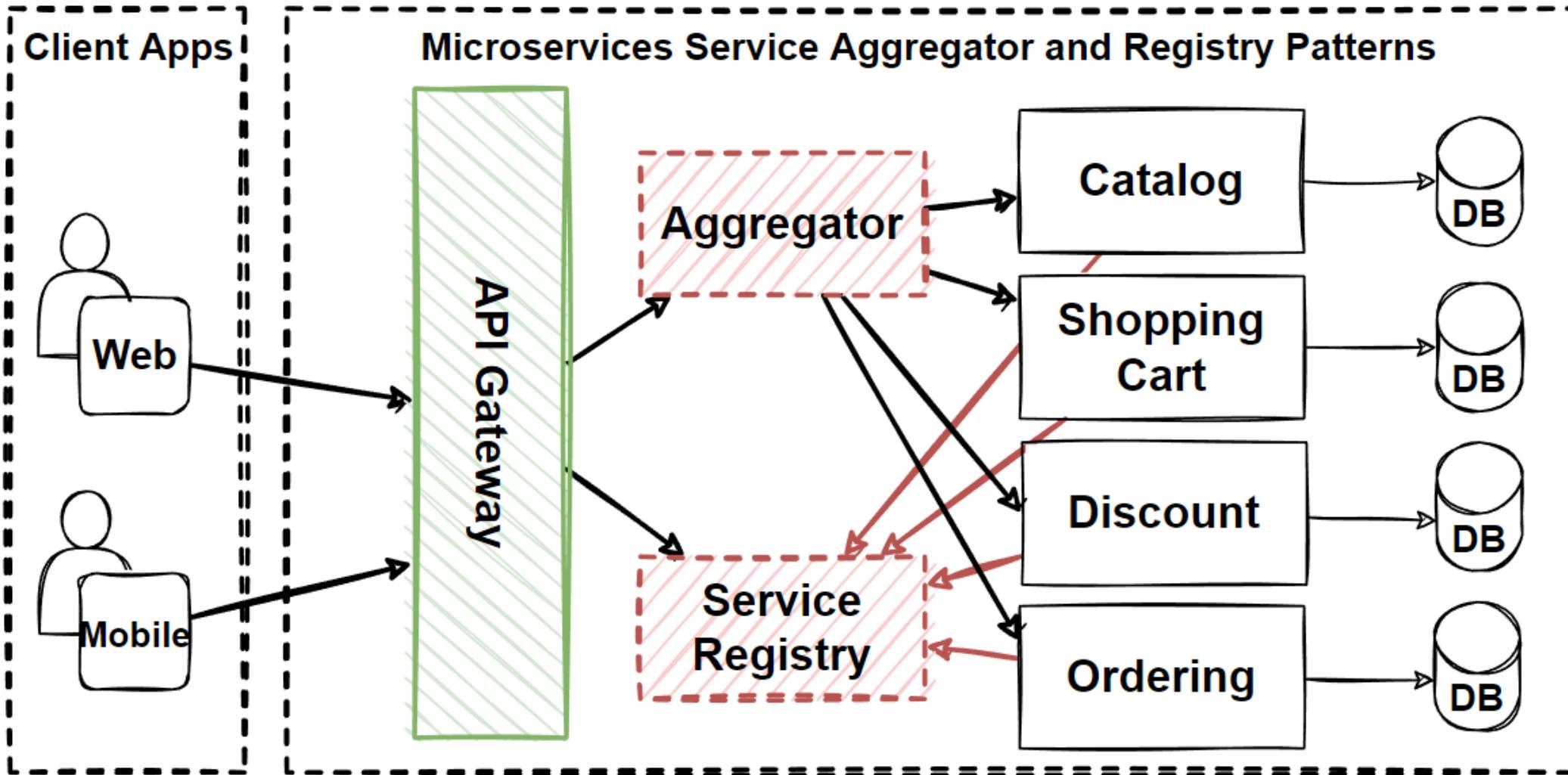
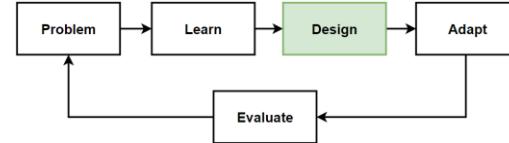
- **3 participants:** Service Registry, Client, Microservices
- **Client service finds the location of other service instances by querying a Service Registry.**
- Internal backend services **register to Service Registry** service before. The Service Registry is a **database for microservice instances**.
- The service registry **keeps records of the network locations** of the microservices.
- When the client service need to access internal services, then it will **query from Service Registry** and access them.
- Clients can **find the locations** of microservices using the **service registry** and directly call them.
- **Netflix** provides a service discovery pattern called «**Netflix Eureka**»
- **Kubernetes** container orchestrator **automatically handle** to service discovery operations.



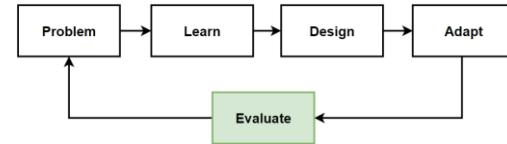
Before Design – What we have in our design toolbox ?

Architectures	Patterns&Principles	Microservices Communications	Non-FR	FR
<ul style="list-style-type: none">• Microservices Architecture• The Database-per-Service Pattern• Polygot Persistence• Decompose services by scalability• The Scale Cube• Microservices Decomposition Pattern• Microservices Communications Patterns	<ul style="list-style-type: none">• HTTP Based RESTful API• GraphQL API• gRPC API• WebSocket API• Gateway Routing Pattern• Gateway Aggregation Pattern• Gateway Offloading Pattern• API Gateway Pattern• Backends for Frontends Pattern-BFF• Service Aggregator Pattern• Service Registry/Discovery Pattern	<ul style="list-style-type: none">• High Scalability• High Availability• Millions of Concurrent User• Independent Deployable• Technology agnostic• Data isolation• Resilience and Fault isolation		<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history

Design: Microservices Architecture with Service Aggregator, Registry/Discovery Pattern



Evaluate: Microservice Architecture with Service Aggregator, Registry/Discovery Pattern



Benefits

Aggregator

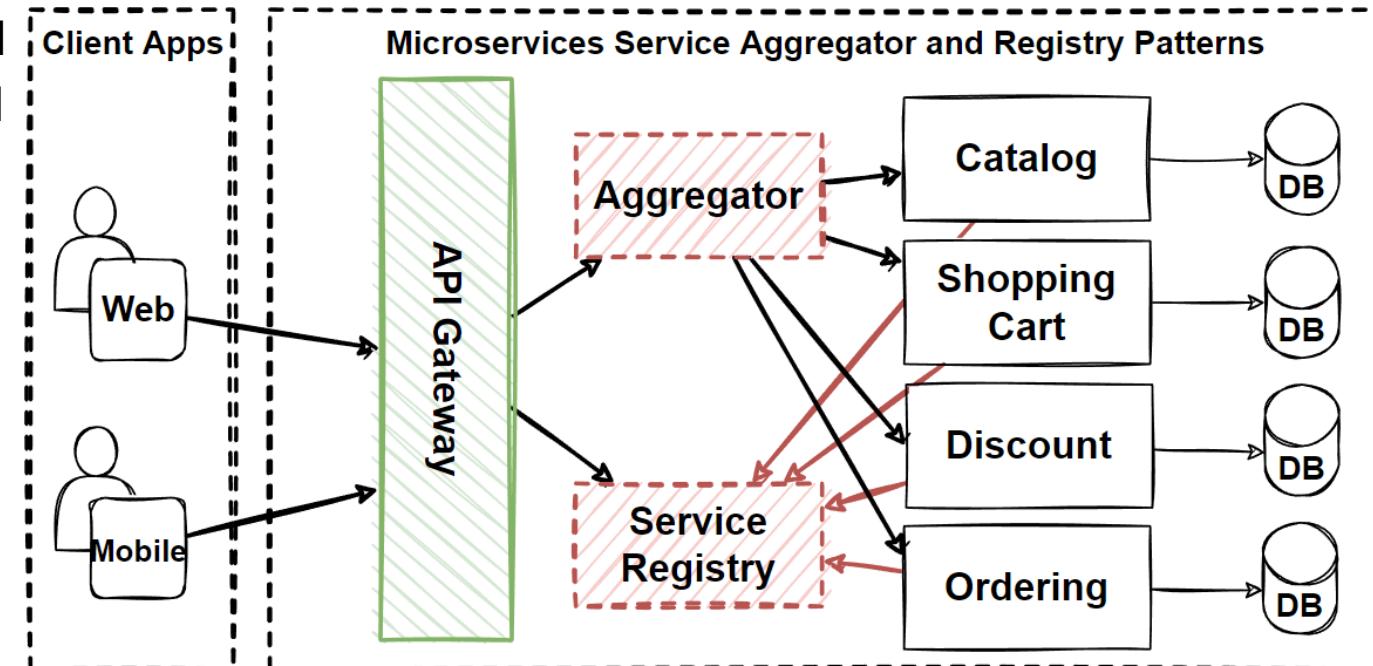
- Reduces chattiness and communication overhead
- Abstract operations that calls to multiple back-end microservices
- Centralizing logic into a specialized microservice

Registry/Discovery

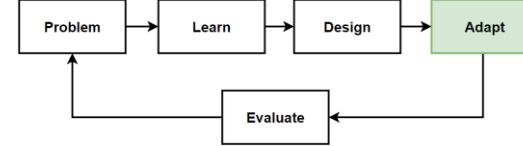
- Provide to register and discover microservices in the cluster
- Keep track of microservice locations beforehand

Drawbacks

- Increased Latency
- Good for large-scaled microservices application which has several client applications
- Only for Sync communications problems



Adapt: Microservice Architecture with Service Aggregator, Registry/Discovery Pattern



Frontend SPAs

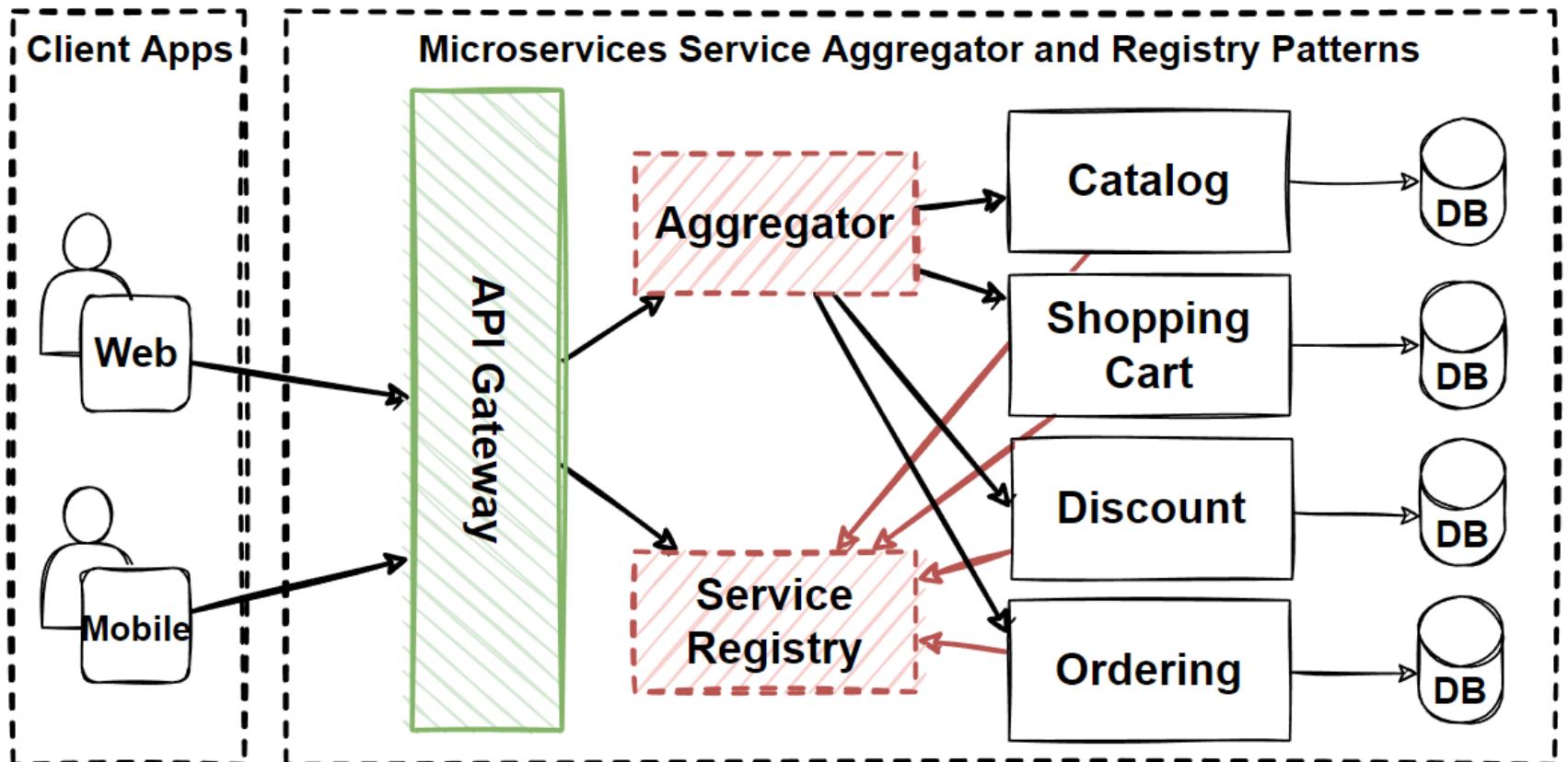
- Angular
- Vue
- React

API Gateways

- Kong Gateway
- Tyk API Gateway
- Express Gateway
- Amazon AWS API Gateway

Service Registry

- Netflix Eureka
- Kubernetes
- Serverless Orchestrators



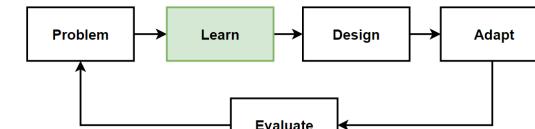
Backend Microservices

- Java – Spring Boot
- .Net – Asp.net
- JS – NodeJS

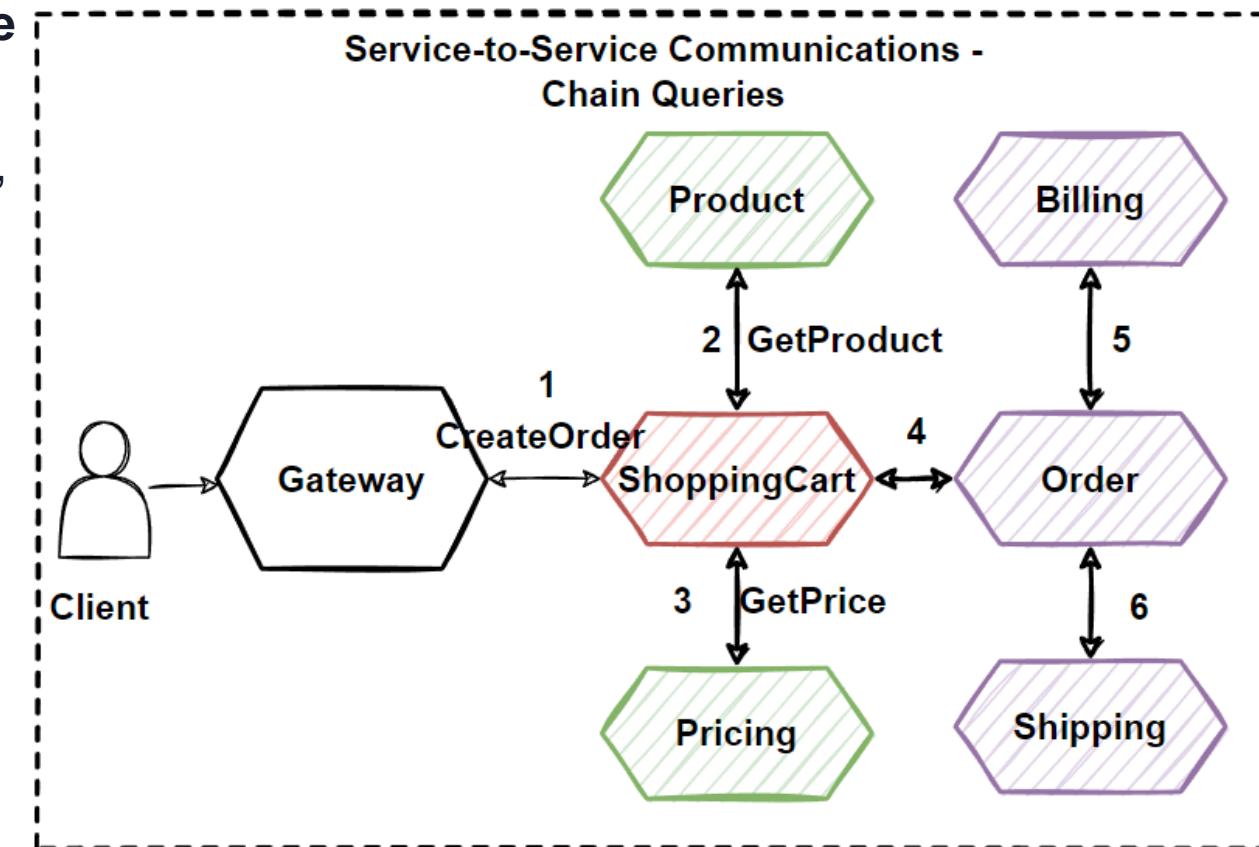
Database

- MongoDB – NoSQL Document DB
- Redis – NoSQL Key-Value Pair
- Postgres – Relational

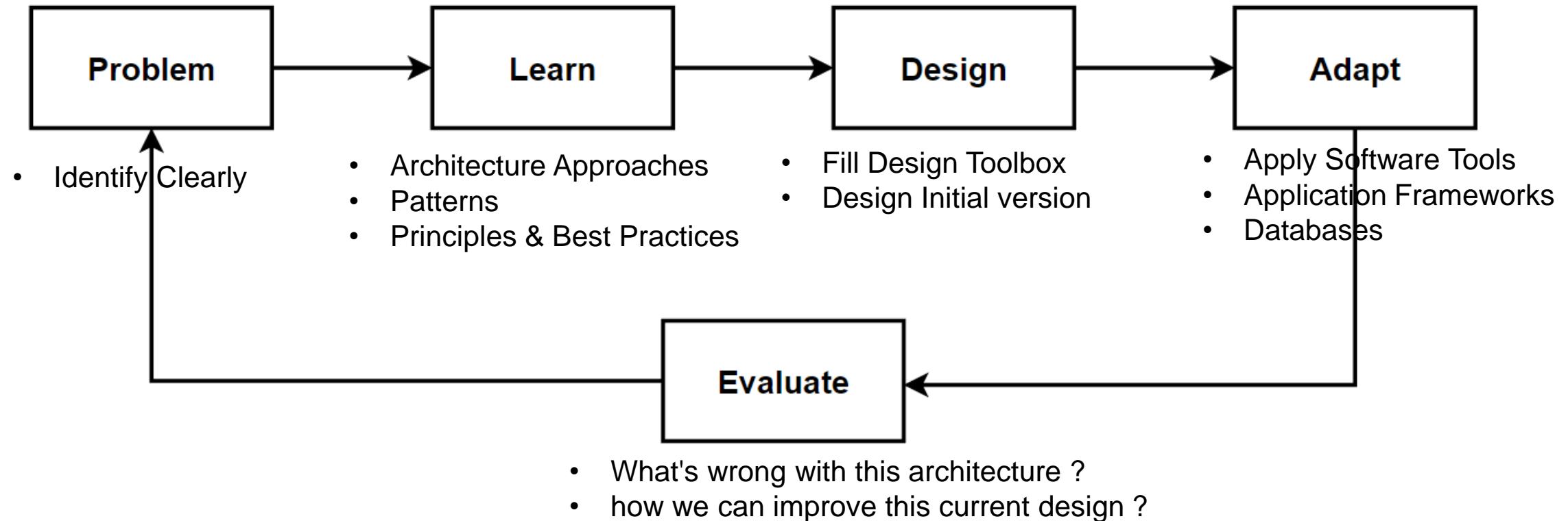
Service-to-Service Communications Chain Queries



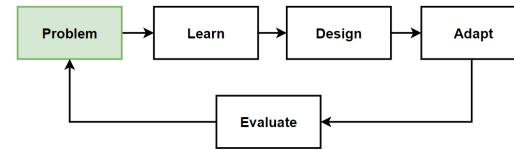
- If **service calls** are much then a few HTTP calls to multiple microservices, than it goes to **un-manageable** situation.
- **Place Order Use Case:** client checkout shopping cart, start order fulfillment processes.
- Request/Response Sync Messaging end up **with 6 sync chain HTTP Request**.
- **Increase latency** and **negatively impact** the performance, scalability, and availability.
- What if the **step 5 or 6 is failed ?**
- We can **apply 2 way** to solve this issues:
 - **1- Change microservices communications to async way**
 - 2- Use Service Aggregator Pattern to aggregate some query operations in 1 API Gateway.



Way of Learning – The Course Flow



Problem: Long Running Operations Can't Handle with Sync Communication



Problems

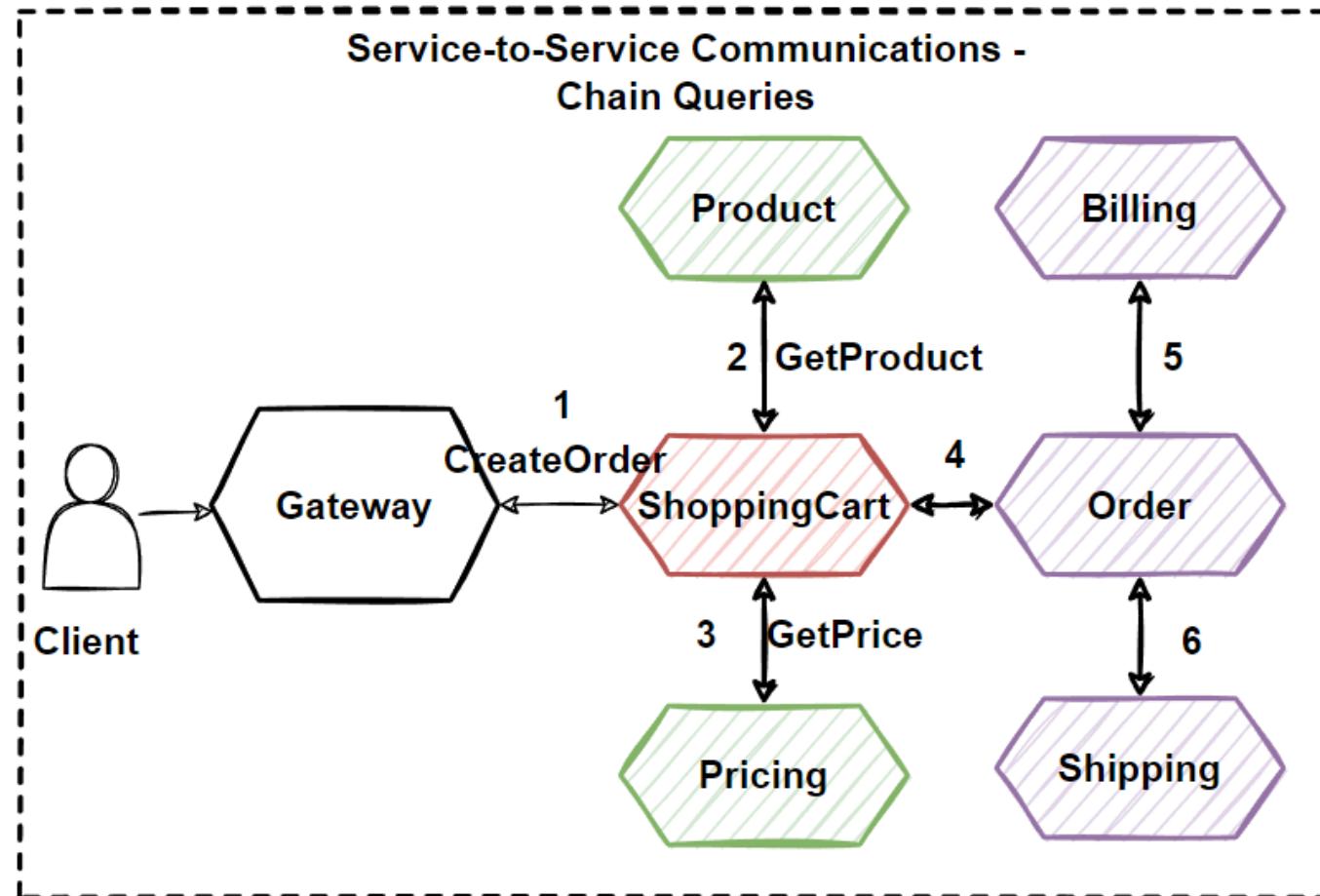
- HTTP calls to multiple microservices
- Chain Queries
- Visit more than a few microservices
- Increased latency with Highly Coupling Services
- Performance, scalability, and availability problems

Best Practices

- Minimize the communication between the internal microservices
- Make microservices communication in Asynchronous way as soon as possible.

Solutions

- **Asynchronous Message-Based Communications**
- Working with events



Microservices Asynchronous Message- Based Communication

Benefits and Challenges of Asynchronous Communication

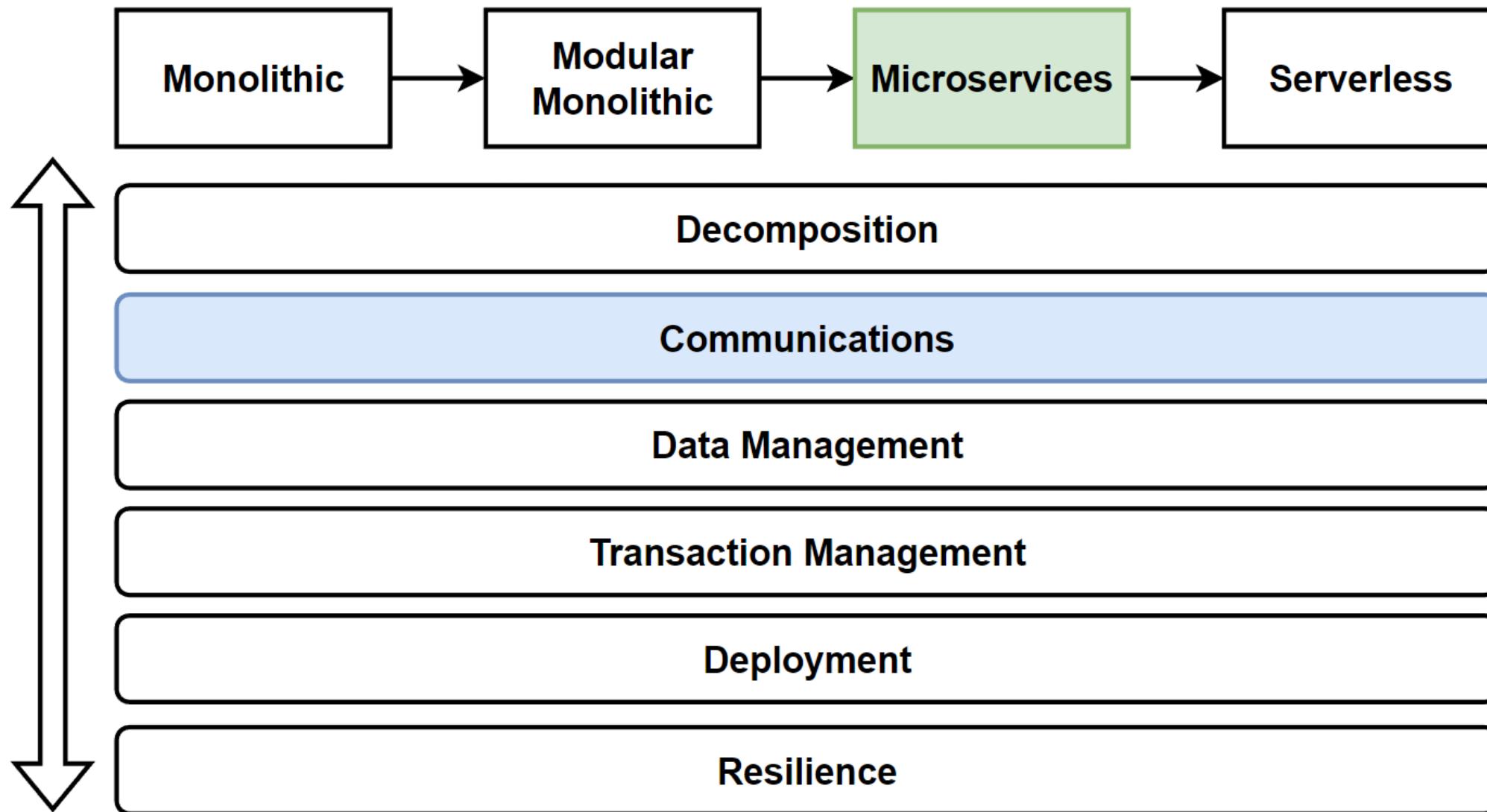
Asynchronous Message-Based Communication Types in Microservices Architecture

Single-receiver Message-based Communication (one-to-one model-queue)

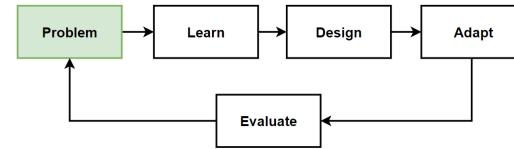
Multiple-receiver Message-based Communication (one-to-many model-topic)

Fan-Out Publish/Subscribe Messaging Pattern

Architecture Design – Vertical Considerations



Problem: Long Running Operations Can't Handle with Sync Communication



Problems

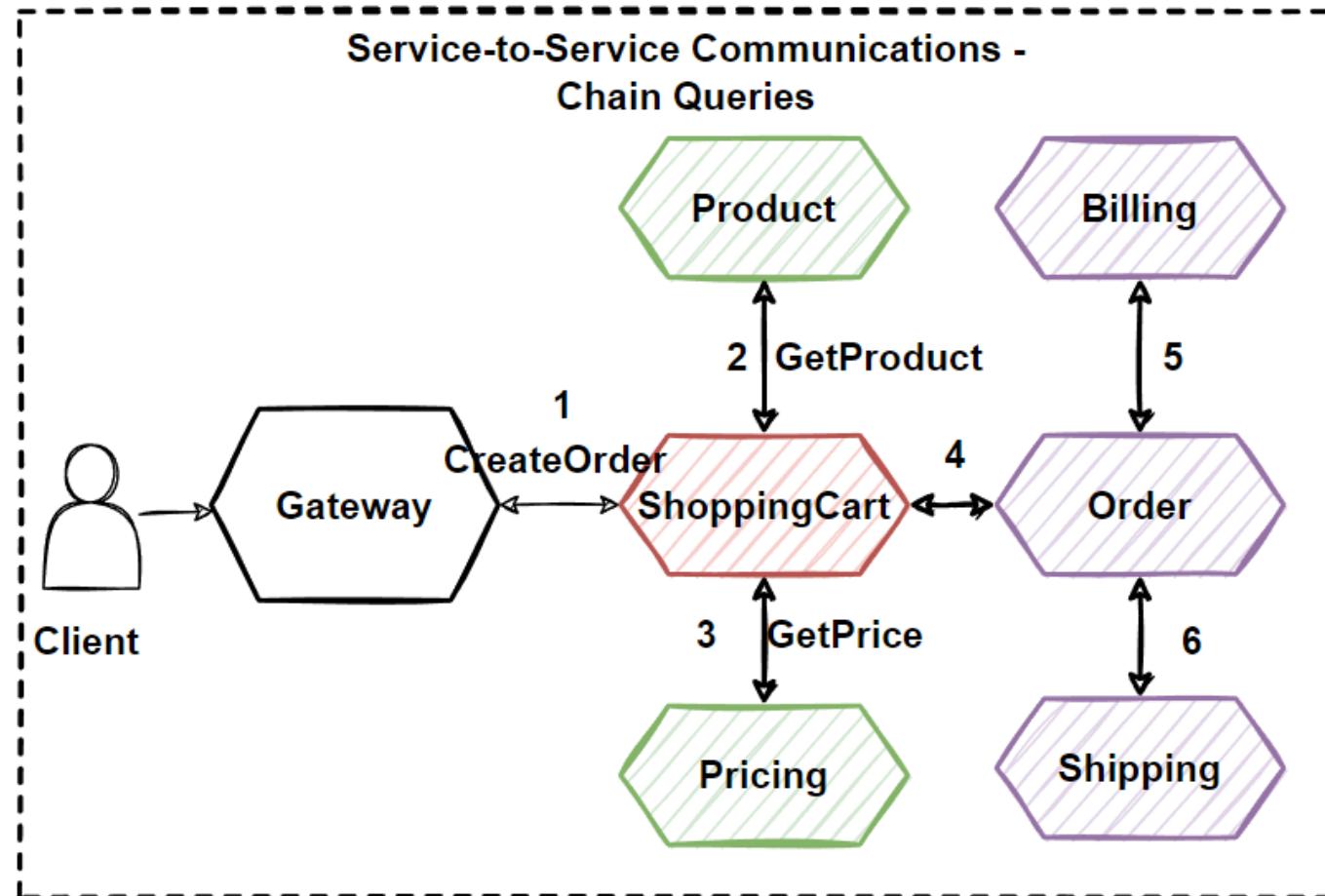
- HTTP calls to multiple microservices
- Chain Queries
- Visit more than a few microservices
- Increased latency with Highly Coupling Services
- Performance, scalability, and availability problems

Best Practices

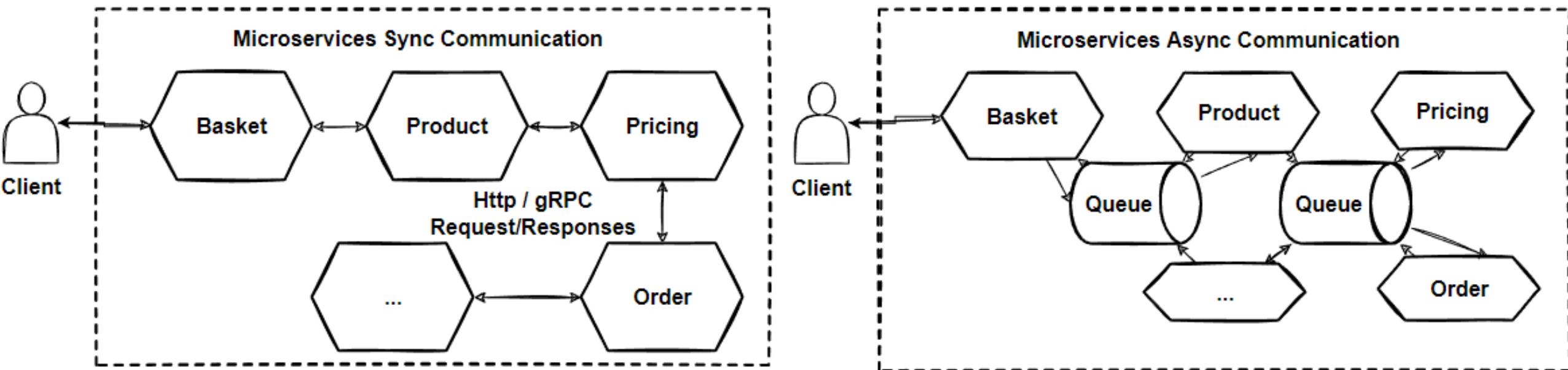
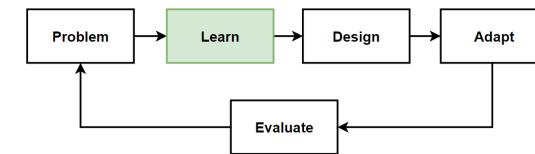
- Minimize the communication between the internal microservices
- Make microservices communication in Asynchronous way as soon as possible.

Solutions

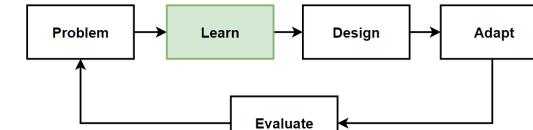
- **Asynchronous Message-Based Communications**
- Working with events



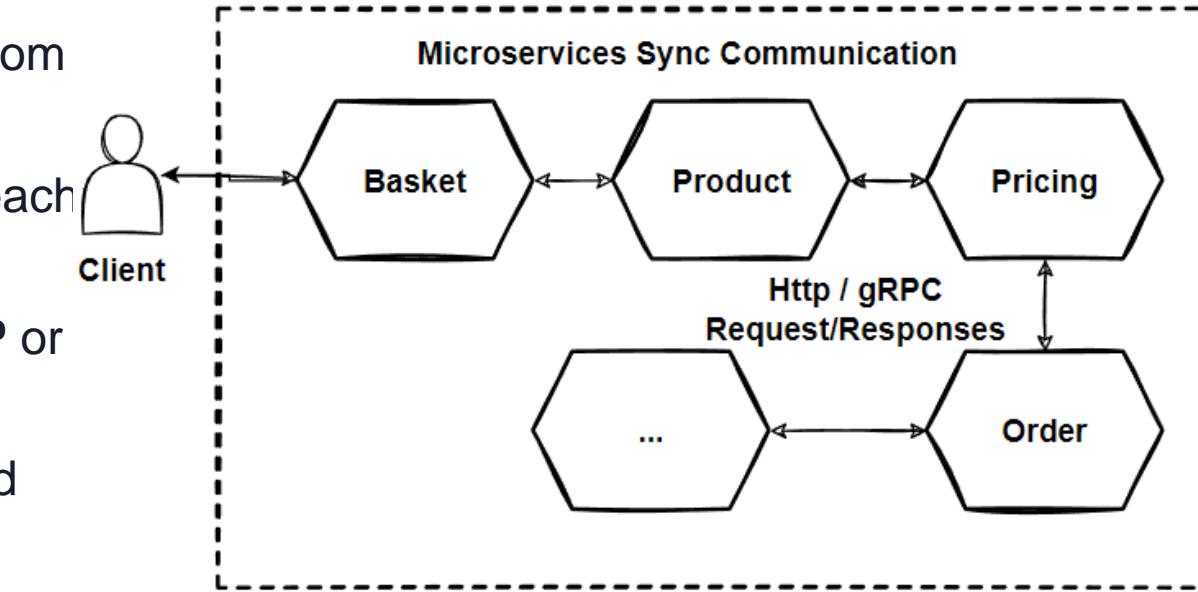
Microservices Communication Types - Sync or Async



Microservices Synchronous Communication

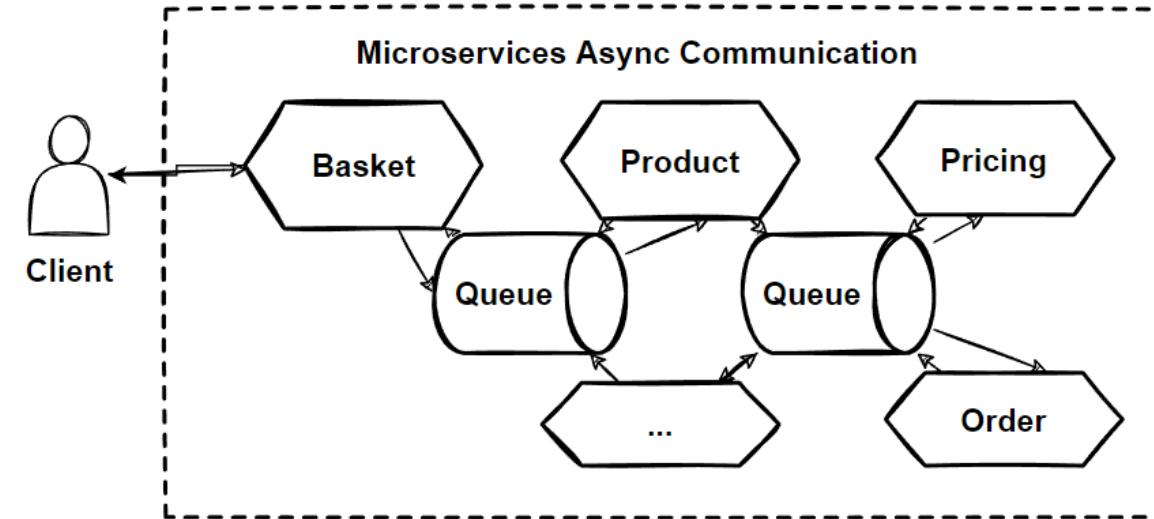
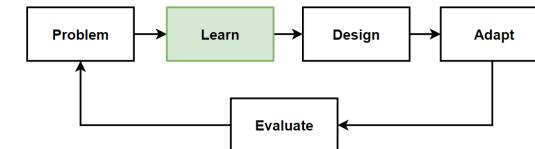


- **Synchronous communication** is using **HTTP** or **gRPC** protocol for returning synchronous response.
- The client **sends a request** and **waits for a response** from the service.
- The client code **block their thread**, until the response reaches from the server.
- The synchronous communication protocols can be **HTTP** or **HTTPS**.
- The client sends a request with using **http protocols** and waits for a response from the service.
- The client **call the server** and **block** client their operations.
- The client code will **continue** its task when it **receives** the HTTP server **response**.

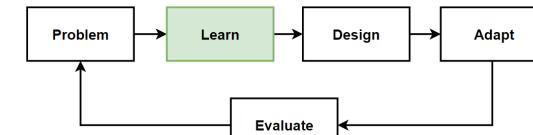


Microservices Asynchronous Communication

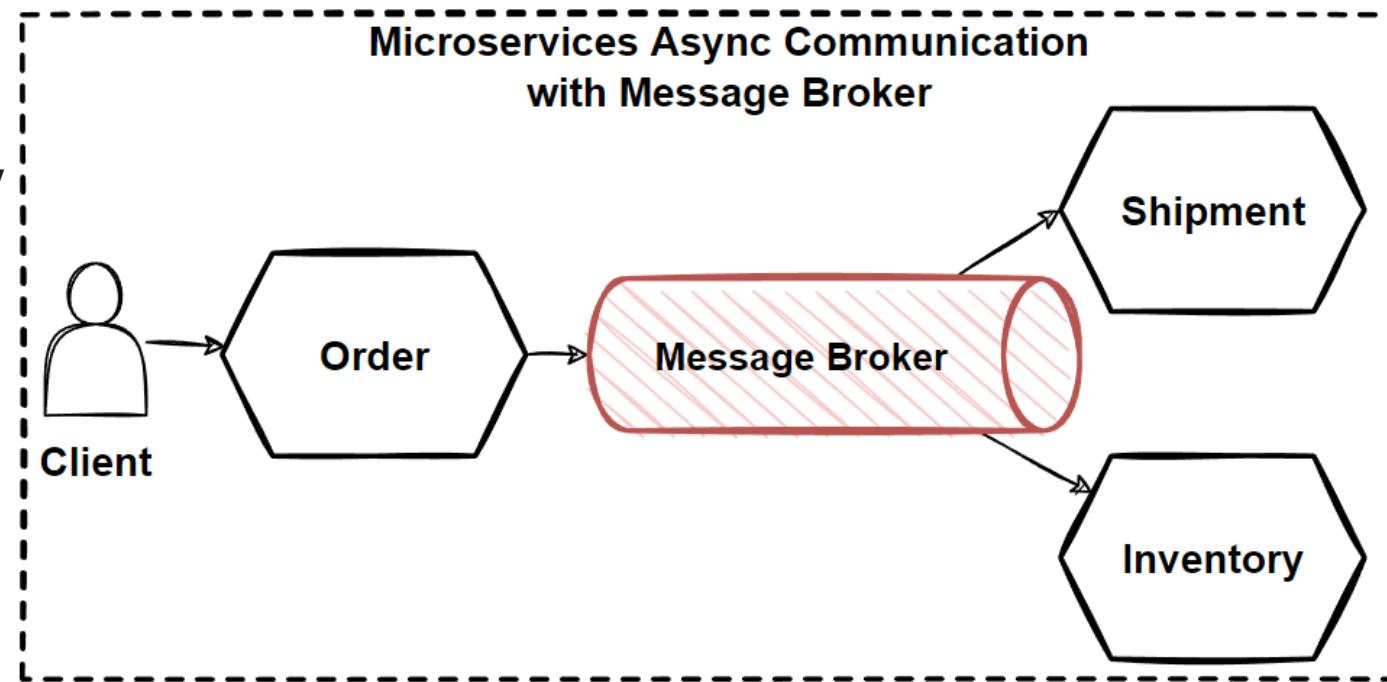
- The client sends a request but it **doesn't wait for a response** from the service. The client **should not have blocked** a thread while waiting for a response.
- **AMQP** (Advanced Message Queuing Protocol)
- Using AMQP protocols, the client **sends the message** with using message broker systems like **Kafka** and **RabbitMQ** queue.
- The message **producer does not wait** for a **response**.
- Message consume from the **subscriber** systems in async way, and no one waiting for response **suddenly**.
- If there is **busy interactions** in communication across multiple microservices, then use **asynchronous messaging platforms**.



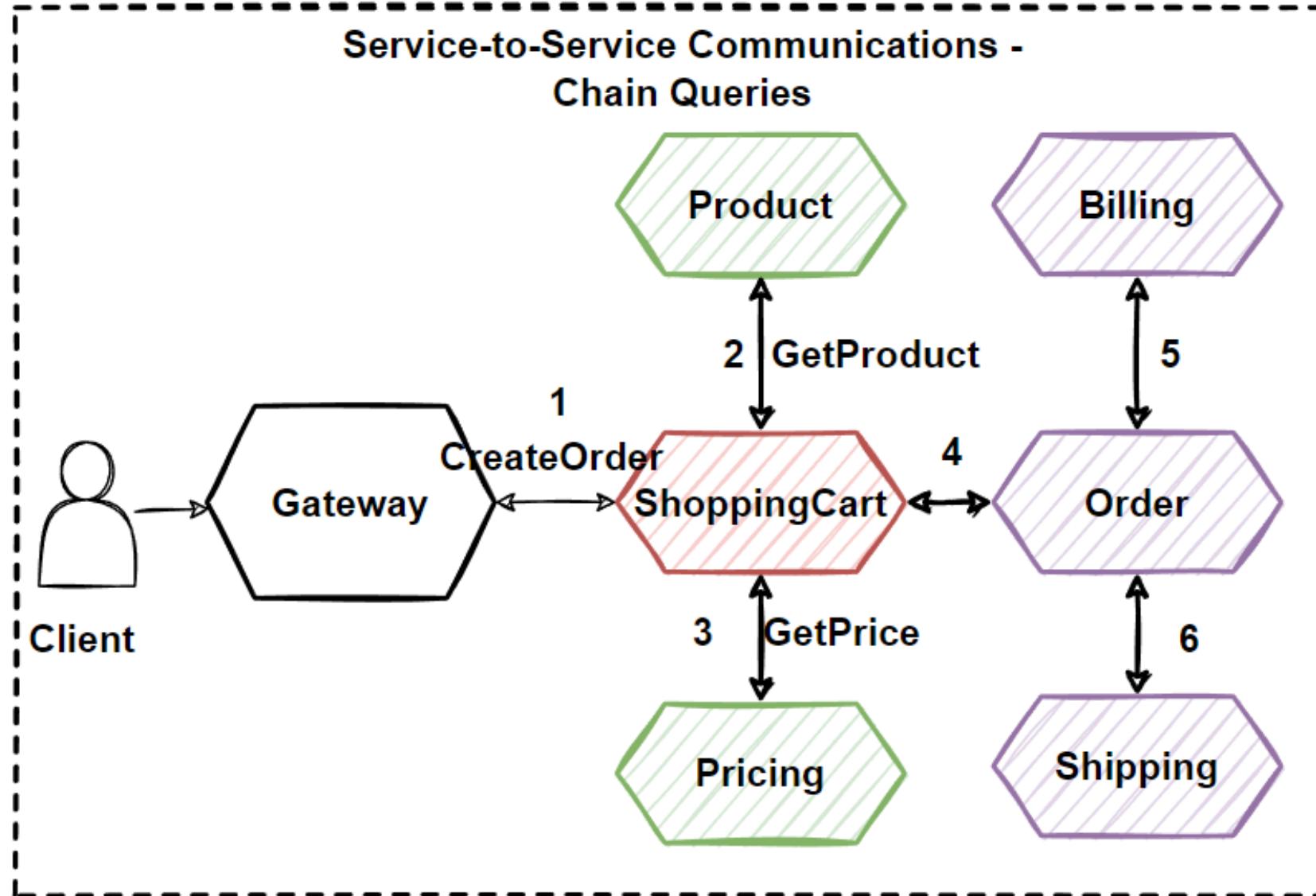
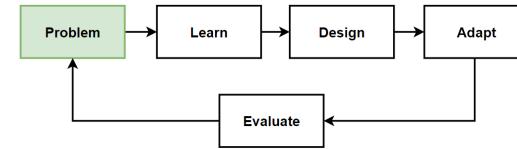
Microservices Asynchronous Communication-2



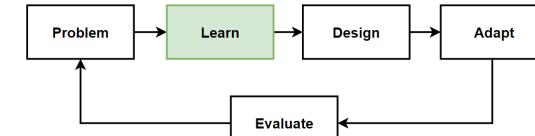
- **Message brokers** are responsible for handling the message sent by the producer service in async messaging-based communication.
- If the **consumer service** is **down** at the moment, the broker might be configured to **retry** as long as necessary for successful delivery.
- Messages can be **persisted if required** or stored only in memory.
- Message broker is **responsible for delivering** the message.
- No longer necessary for both microservices to be **up and running** for successful communication.
- Async messaging provides **loosely couple communication**.



Problem: Chain of request and highly coupled dependent microservices



Benefits of Asynchronous Communication



- **New Subscriber Services**

Adding new services is very simple. We can easily subscribe to message that we want to receive. The producer doesn't need to be know about subscribers, we can remove and add subscribers without affecting producer service.

- **Scalability**

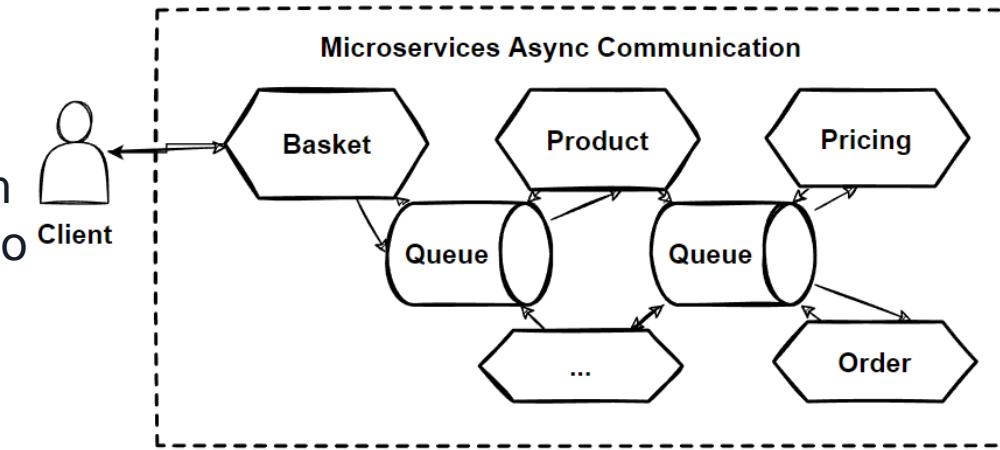
With async communications we can easier to manage scalability issues, can scale producer, consumer and message broker system independently. Scale services according to incoming messages into event bus system. Kubernetes KEDA Auto-scalers.

- **Event-driven Microservices**

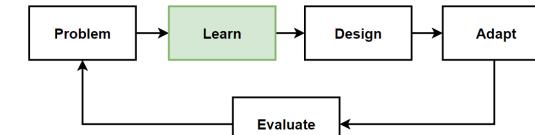
With async communication, we can provide event-driven architectures which is best way to communicate between microservices.

- **Retry mechanisms**

Brokers can retry to sending message and keep trying automatically without any custom solutions.



Challenges of Asynchronous Communication



- **Single Point of Failure - Message Broker**

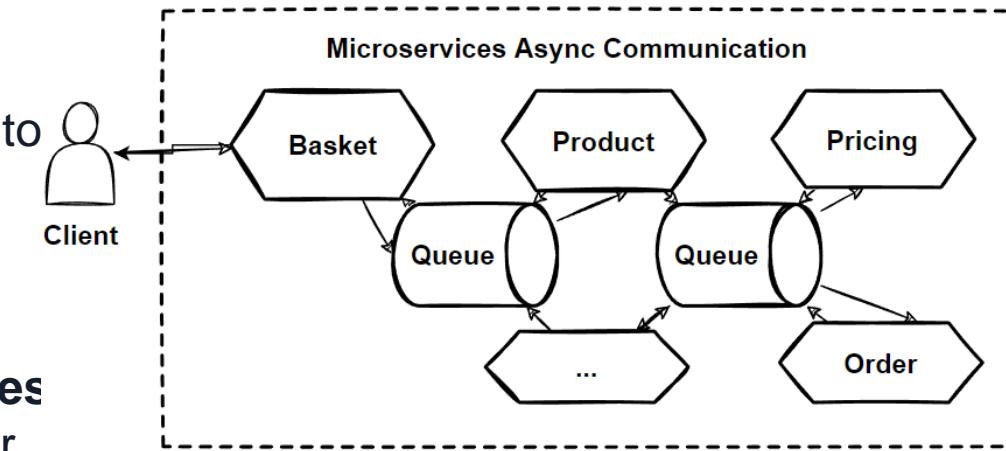
The message broker becomes a single point of failure. We should not rely of all communication with a single node of message brokers, instead we should scale it and use hybrid communication with sync and async in your cases.

- **Debugging**

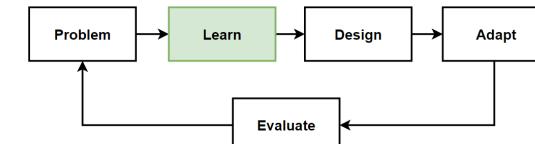
difficult to debug issues with async communication, it can be hard to trace the flow of a single operation across service boundaries. debugging of the flow and the payload of events takes so many times and hard to debug at the same time.

- **At-least-once delivery and Not Guarantee an order of messages**

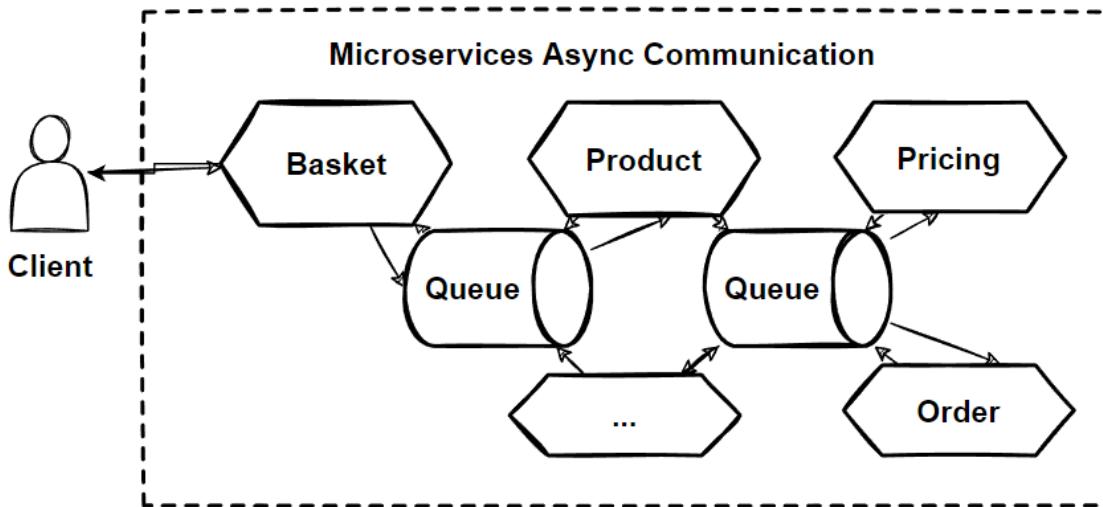
Mostly Brokers use at-least-once delivery and not Guarantee order of messages. Should embrace these message delivery mechanism with applying idempotency consumers and not designing FIFO requires cases.



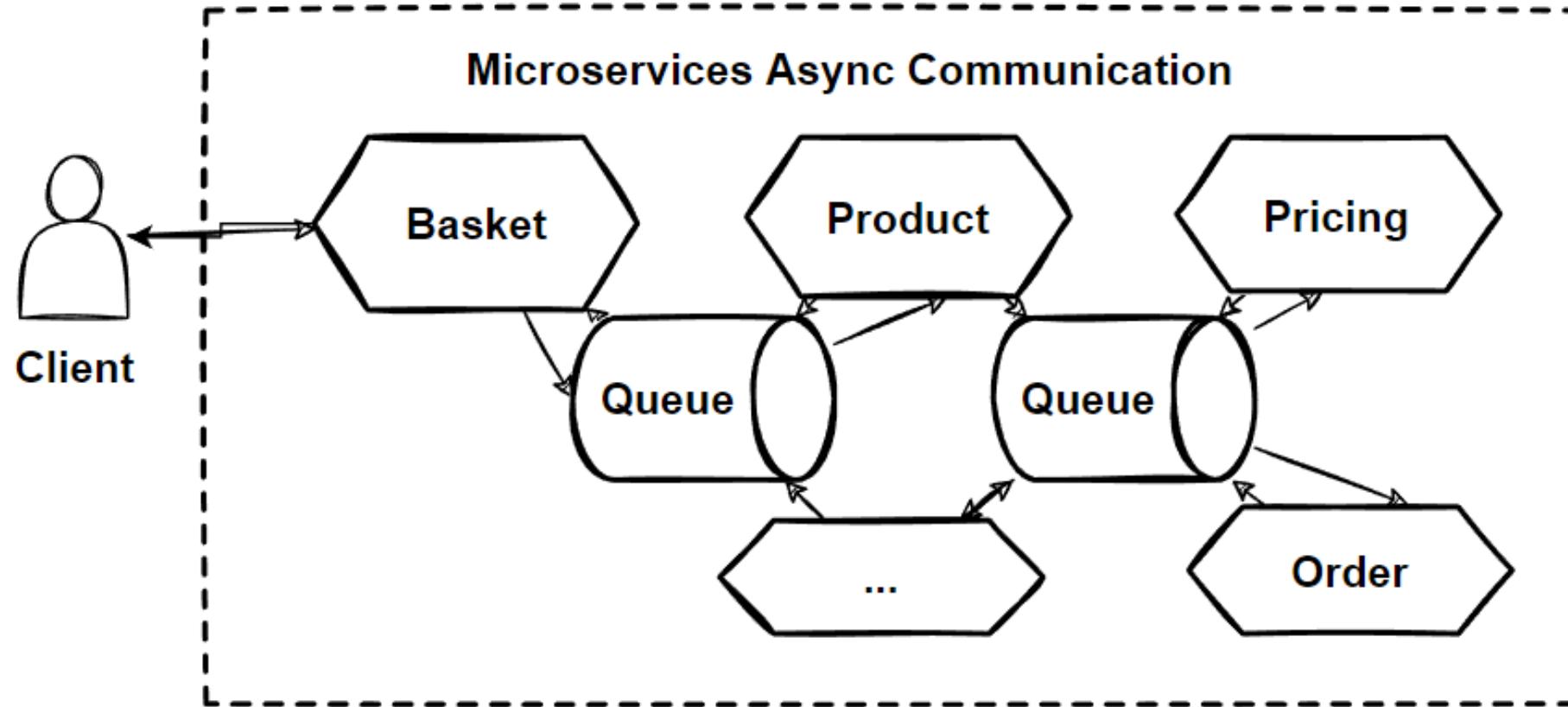
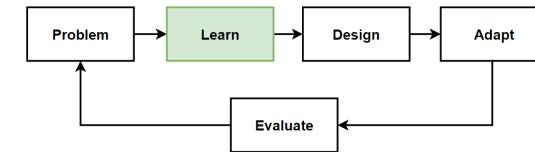
Asynchronous Message-Based Communication



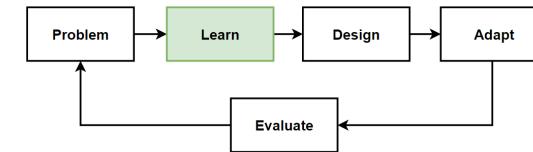
- **Use Asynchronous** message-based communication when you have **multiple microservices are required to interact each other** without any dependency.
- Asynchronous message-based communication is works with **events**.
- **Events** can place the communication between microservices: **Event-driven communication**.
- If **any changes happens** in microservices, it is **propagating changes** across **multiple microservices** as an event.
- Events consumed by **subscriber microservices**.
- **Event-driven** communication bring us **Eventual consistency**.



Asynchronous Message-Based Communication-2



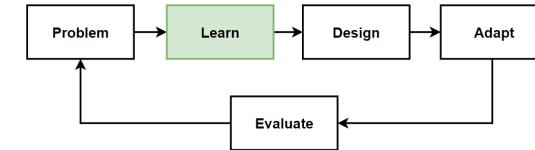
2 Type of Asynchronous Messaging Communication



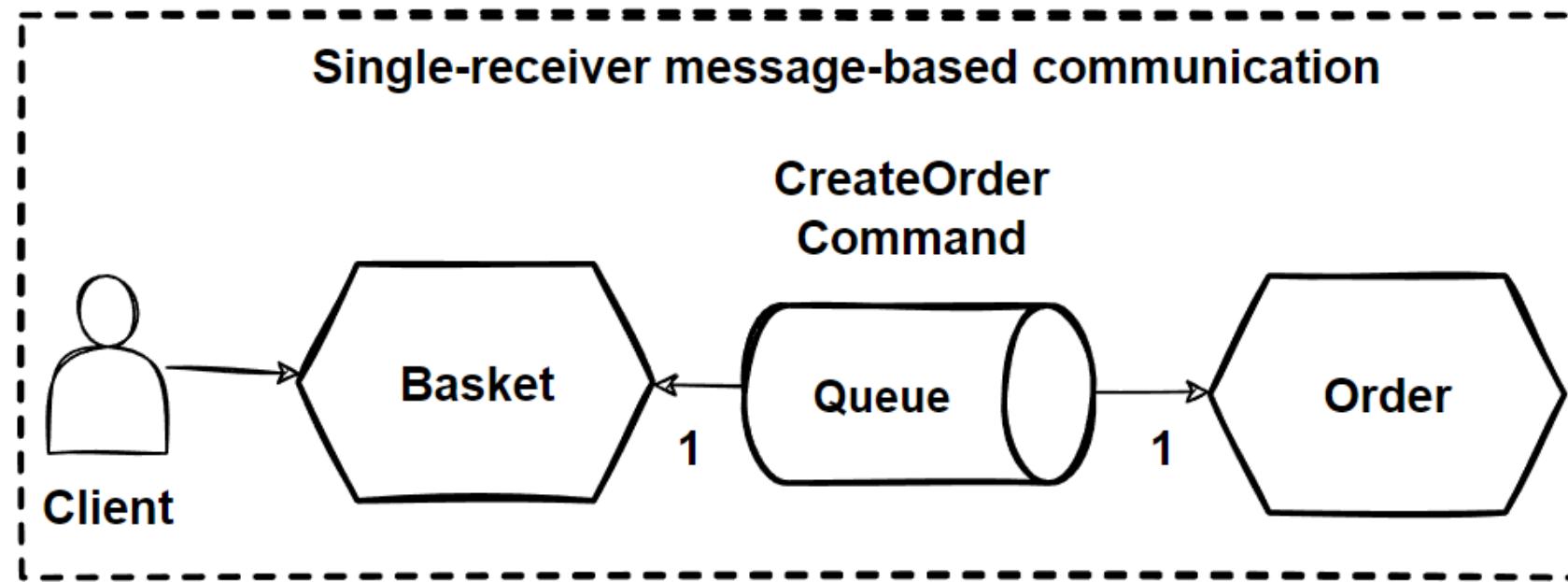
Single receiver message-based communication
one-to-one(queue) model or
Point-to-point model

Multi receiver message-based communication
one-to-many (topic) model or
publish/subscribe model

Single-receiver Message-based Communication 1-1

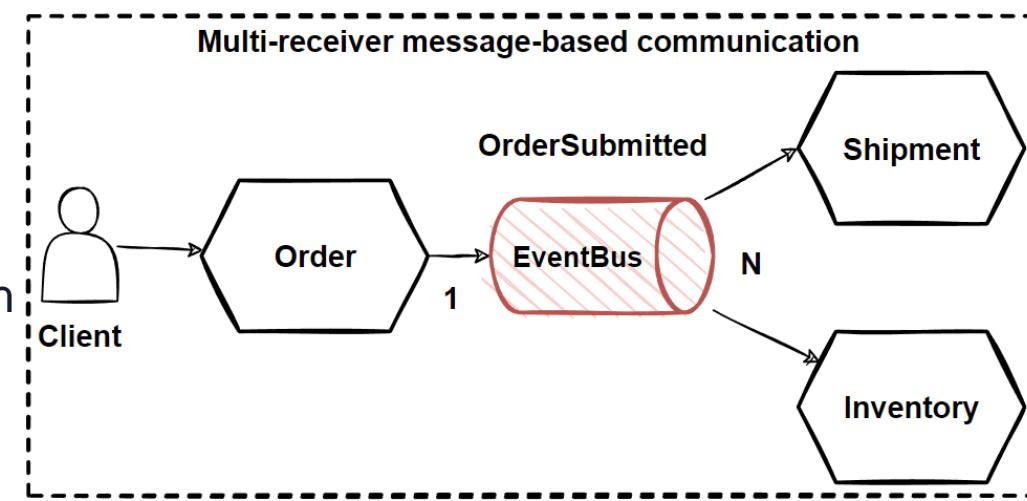
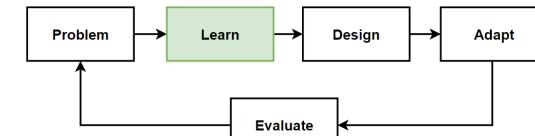


- one-to-one (queue) or point-to-point communications.
- If you **send 1 request** to the specific consumer, and this operation will take long time, use this **Single-receiver async one-to-one** communication.
- **Single producer** and **single receiver**; implementation as a **Command Pattern**.

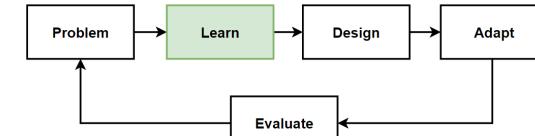


Multiple-receiver Message-based Communication

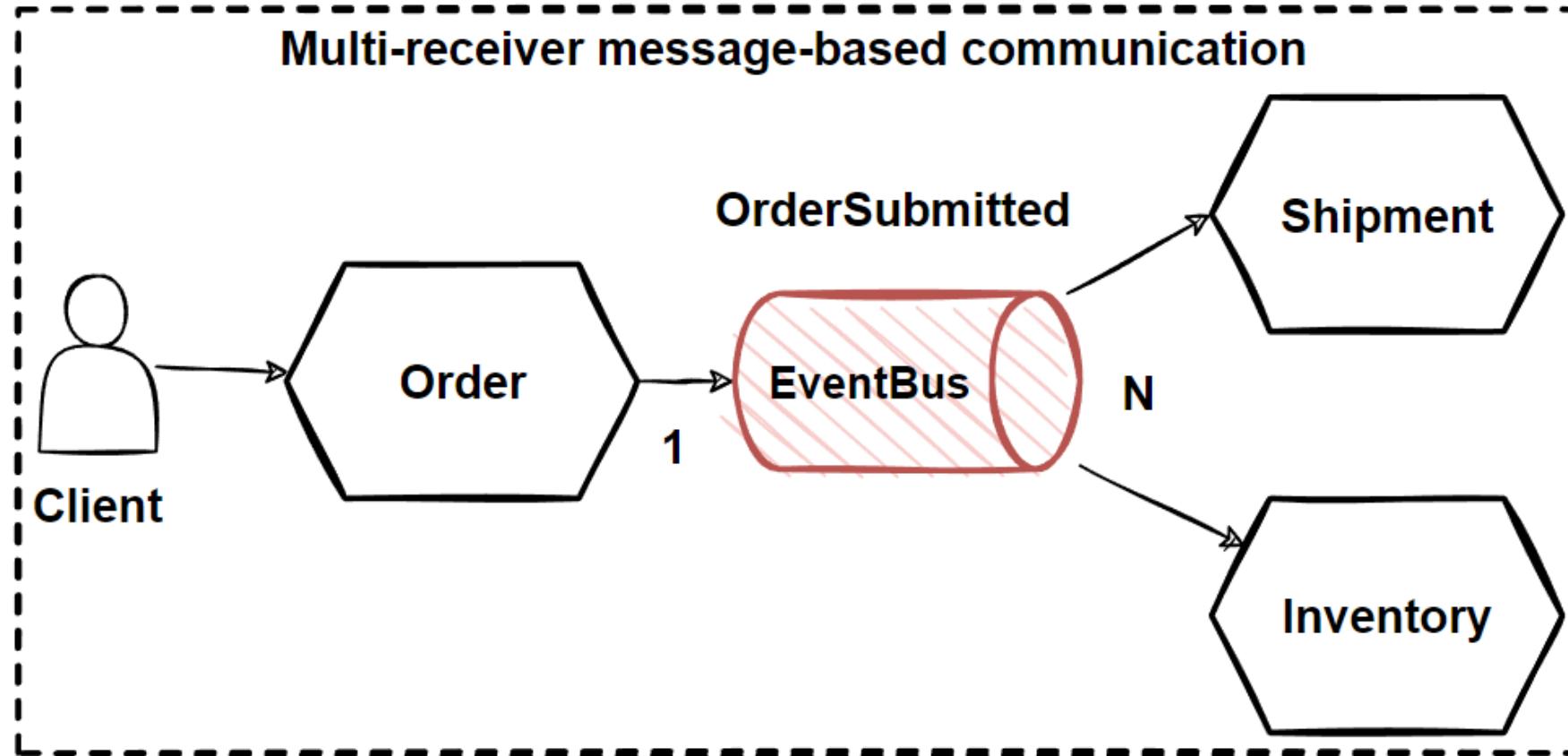
- Publish/subscribe mechanisms that has multiple receivers.
- Producer service **publish a message** and it **consumes** from several **microservices** by **subscribing message** on the message broker system.
- **Publisher don't need to know any subscriber**, no any dependency with communication parties.
- **one-to-many (topic)** implementation has Multiple receivers. Each request can be processed **by zero to multiple receivers**.
- Publish/subscribe used in patterns **Event-driven microservices** architecture.
- Message broker system is **publishing events** between multiple microservices, **subscribing these events** in an async way.
- Messages are **available to all subscribers** and the topic can have **more than one subscriber**.
- The message **remains persistent** in a topic until they are deleted.
- **Kafka, RabbitMQ or Amazon SNS and EventBridge**.



Multiple-receiver Message-based Communication

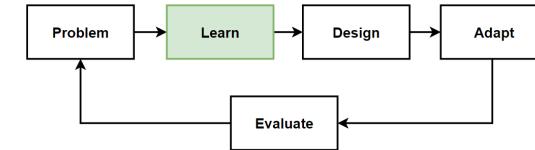


- Microservice **publishes an event** when something happens. Price change in a product microservice: **Price Changed** event can be subscribed from SC microservice in order to update basket price.

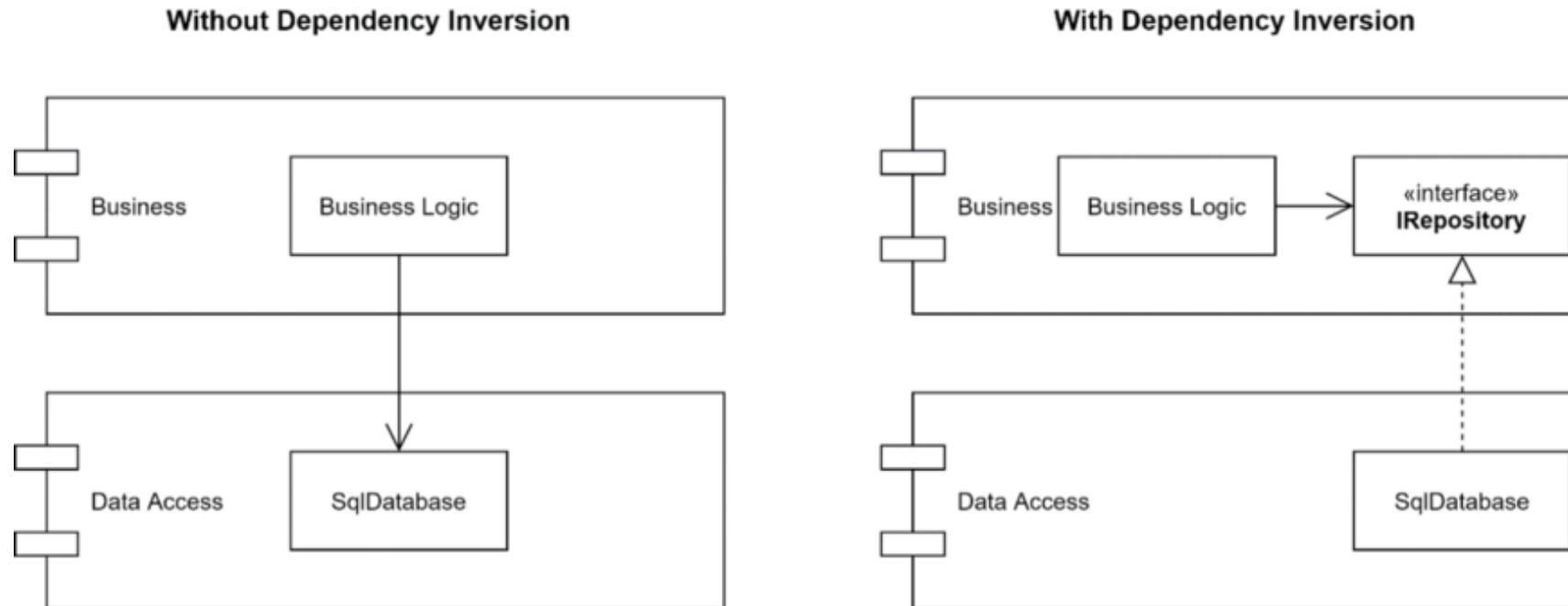


- Event-driven Architecture, **CQRS pattern**, **event storming**, **eventual consistency** principles.

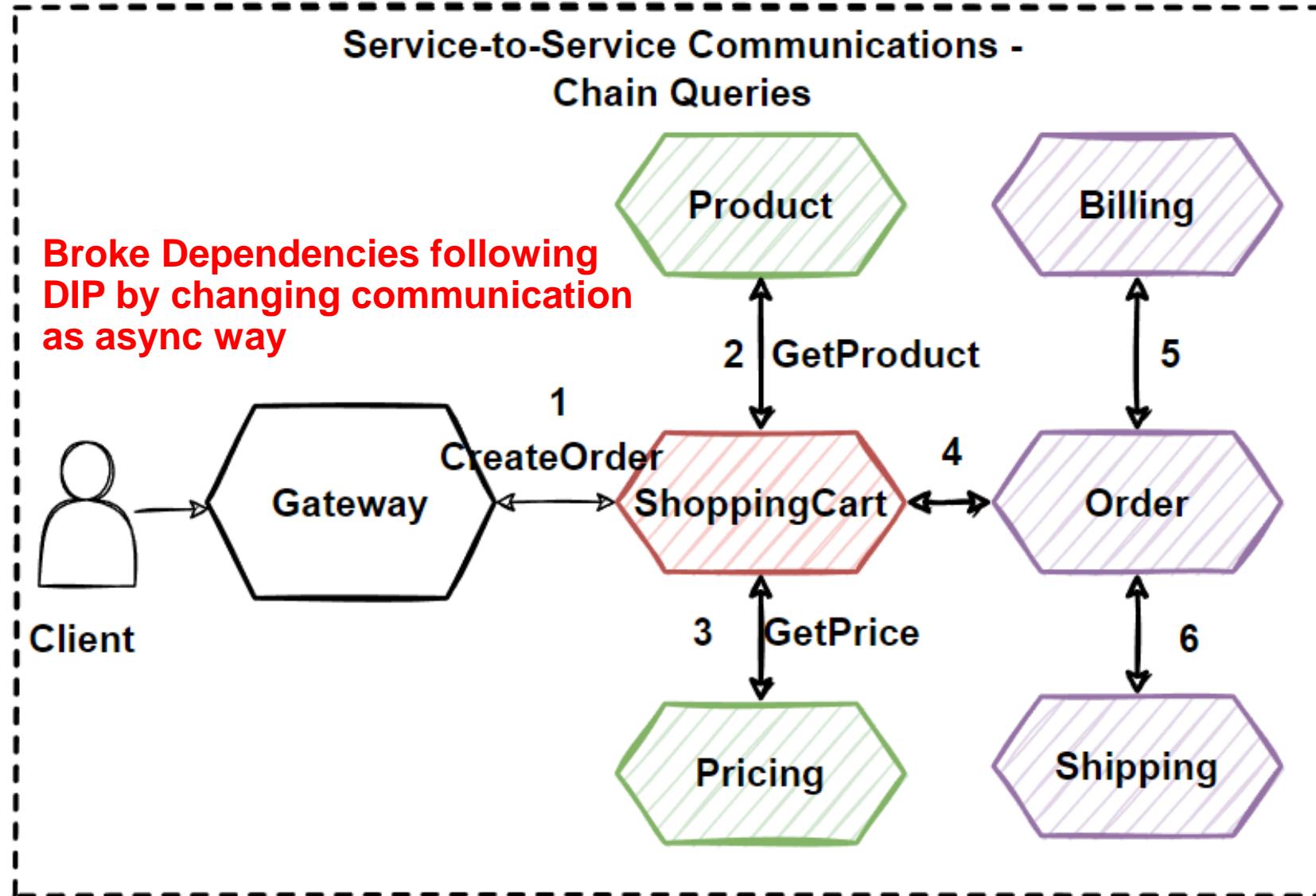
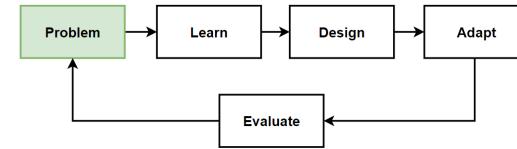
Dependency Inversion Principles (DIP)



- Provide to broke dependency of classes by inverting dependencies, inject dependent classes via constructor.
- Upper-level modules or classes and lower-level classes must not be dependent on modules.
- Lower-level modules must be dependent on higher-level modules (interfaces of modules).

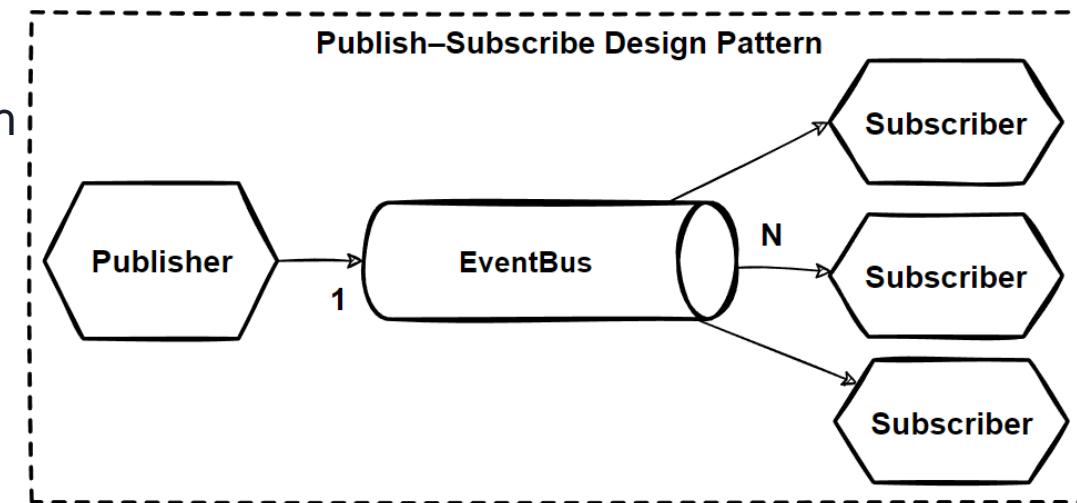
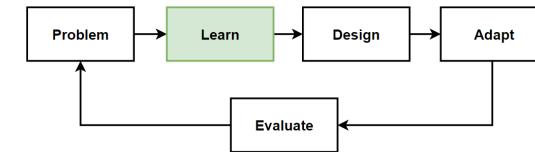


Problem: Chain of request and highly coupled dependent microservices



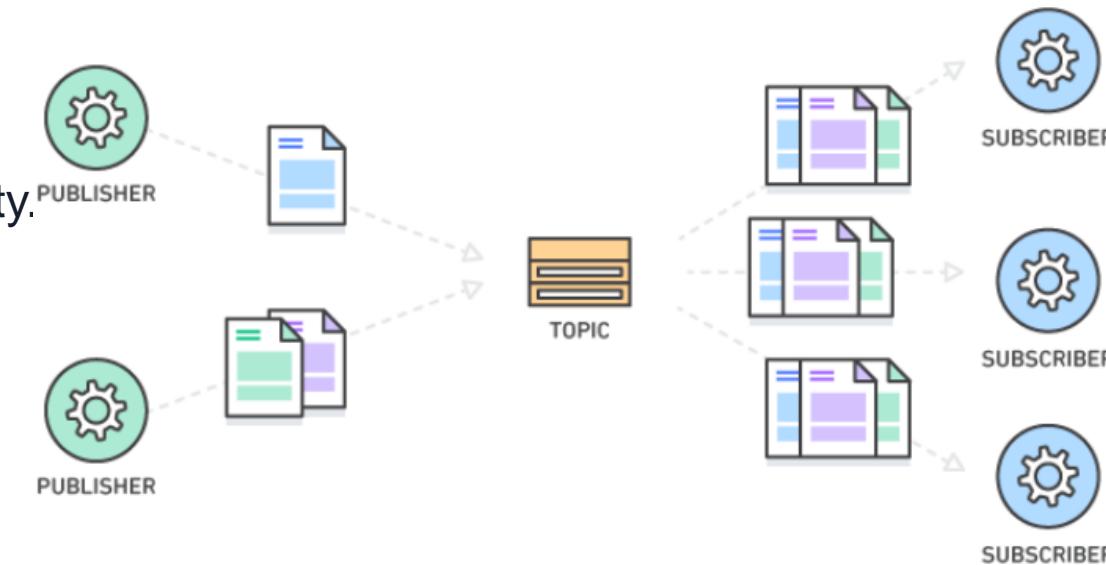
Fan-Out & Message Filtering with Publish/Subscribe Pattern

- **Fan-out** is a messaging pattern; ‘fanned out’ to multiple destination in parallel.
- Each of destinations can work and process this messages in parallel.
- **Publisher/subscriber model** to define a topic which is logical access point to enabling message communication with asynchronously.
- Publisher sends the message to the topic, message is immediately fanned out to all subscribers of this topic.
- Each service can operate and scale independently and individually that completely decoupled and asncronously.
- The publisher and the subscribers don't need to know who is publishing / consuming this message that is broadcasting.
- Deliver the same message to multiple receivers is to use the **Fanout Publish/Subscribe Messaging Pattern**.

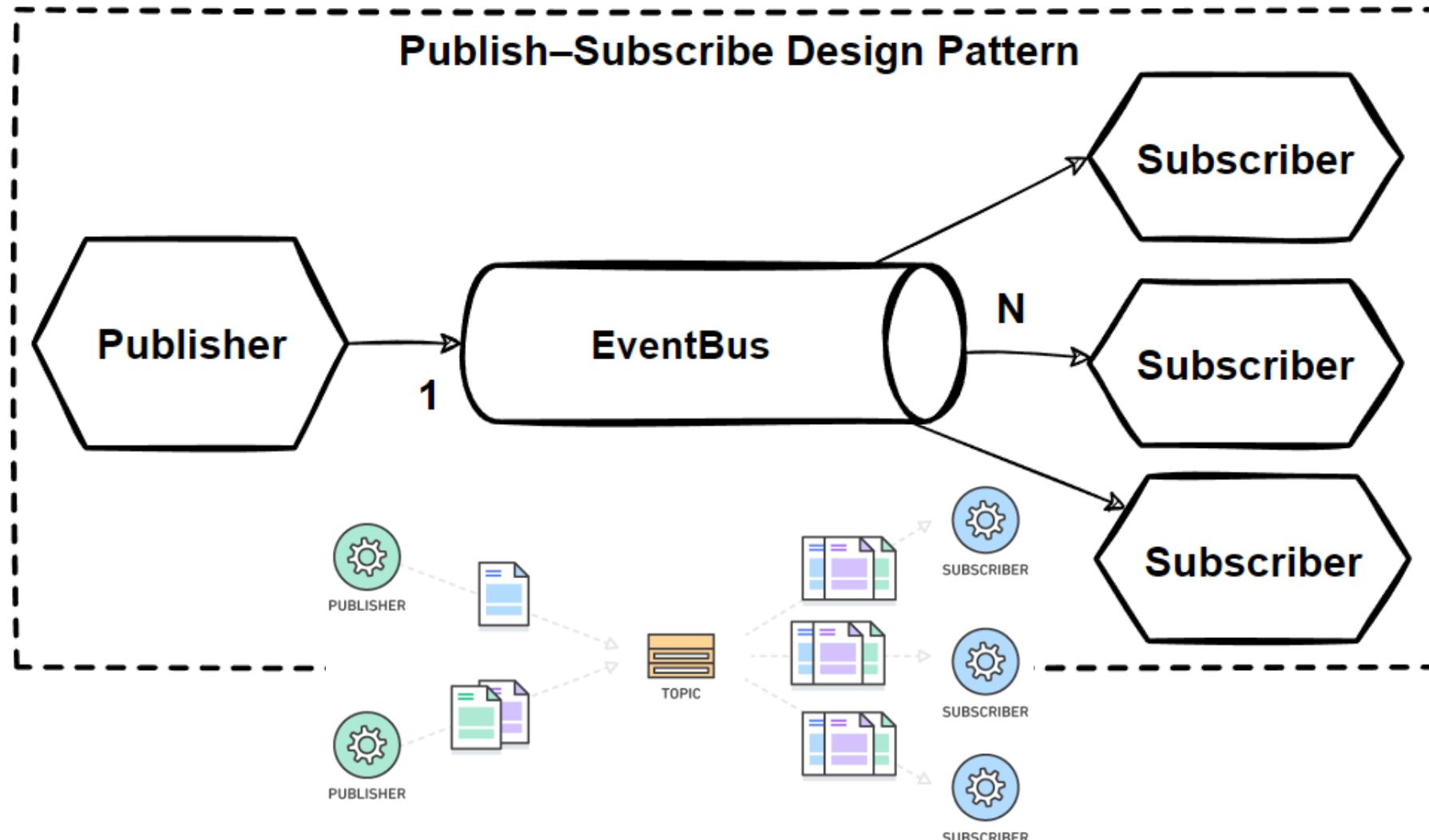
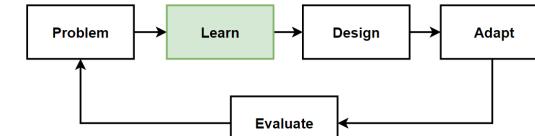


Publish/Subscribe Messaging Pattern

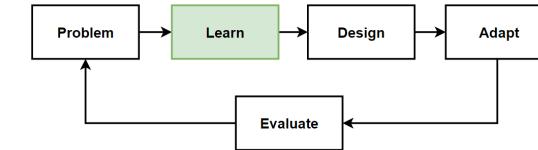
- **Publish/subscribe messaging** is a form of **asynchronous service-to-service communication**.
- Any message **published** to a **topic** is **immediately received** by all of the **subscribers** to the **topic**.
- Enable **event-driven architectures**, and **decouple applications** to increase performance, reliability and scalability.
- Applications are **decoupled** into **smaller, independent building blocks** that are easier to develop, deploy and maintain.
- Publish/Subscribe (Pub/Sub) messaging provides **instant event notifications** for these distributed applications.
- A **message topic** provides a **lightweight mechanism** to broadcast **asynchronous event notifications**.
- All components that subscribe to the topic receive every message, unless a **message filtering policy** is set by the subscriber.



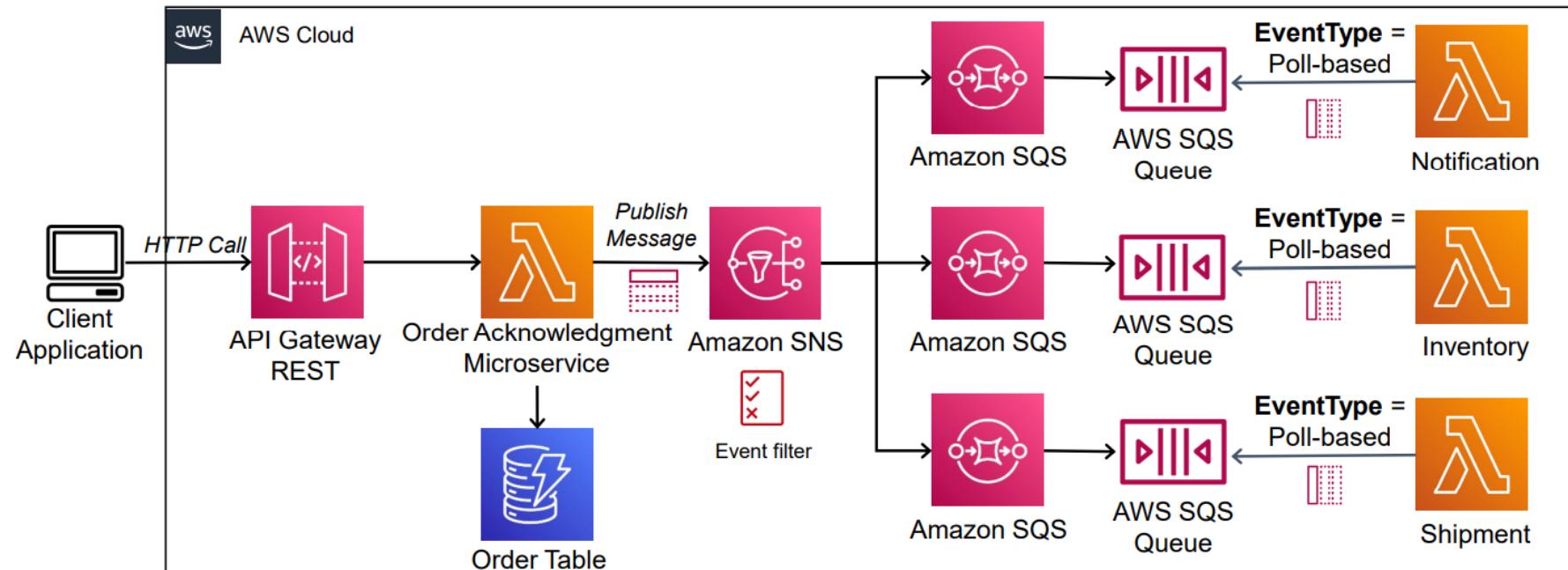
Publish/Subscribe Messaging Pattern-2



Topic-Queue Chaining & Load Balancing Pattern



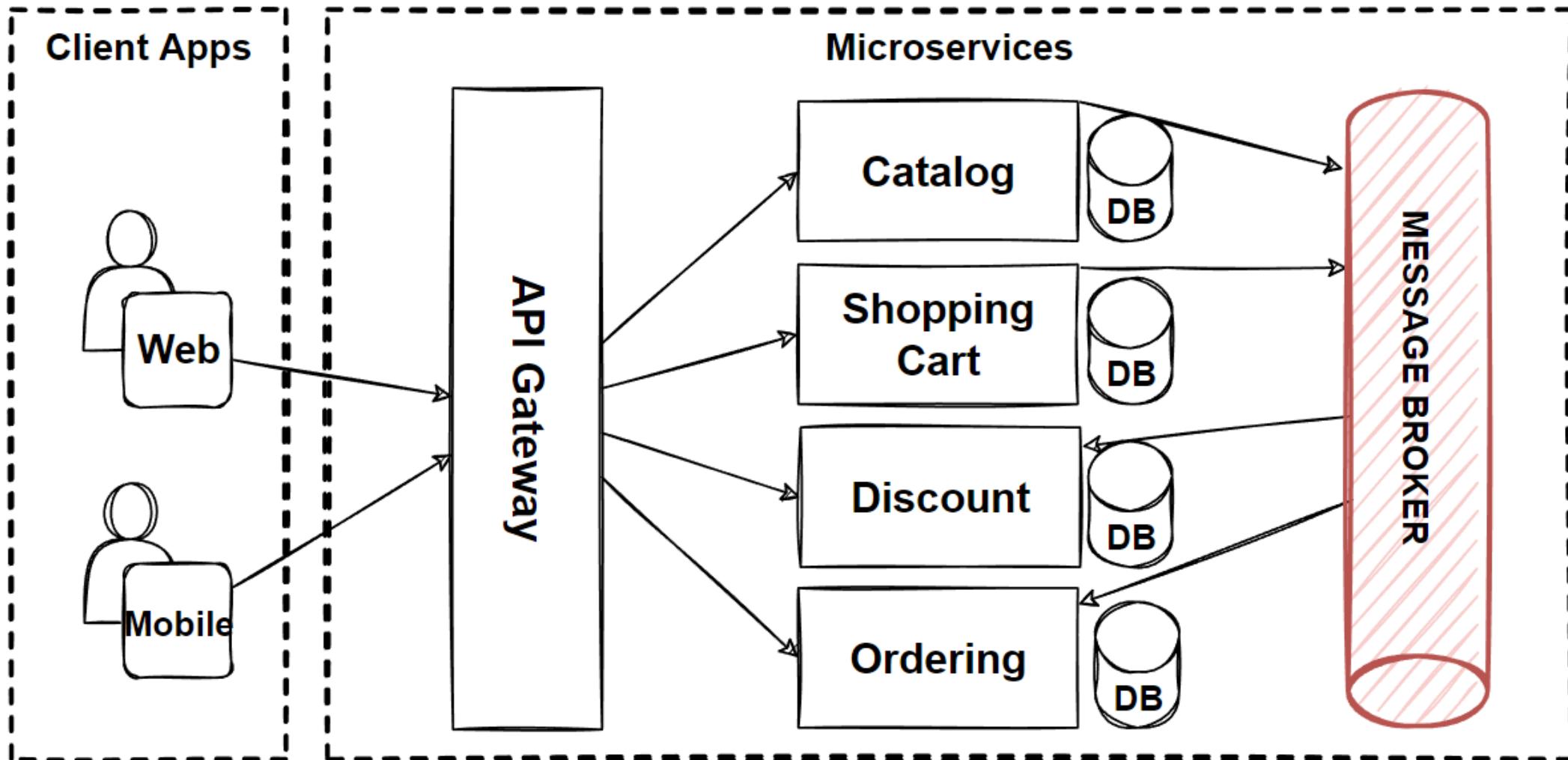
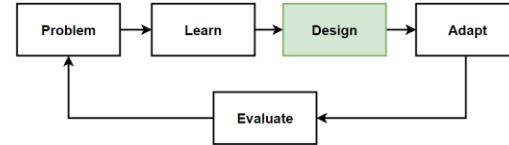
- Use a queue that acts as a **buffer** between the **service** to avoid loss data if the service to fail.
- Services can be down or getting exception or taken offline for maintenance, then events will be loses, disappeared and can't process after the **subscriber service** is up and running.
- Put **Amazon SQS** between EventBridge and Ordering microservices.
- Store this event messages into **SQS queue** with durable and persistent manner, no message will get lost. Queue can act as a **buffering load balancer**.



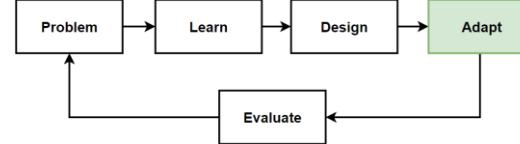
Before Design – What we have in our design toolbox ?

Architectures	Patterns&Principles	Microservices Communications	Microservices Async Communications	FR
• Microservices Architecture	<ul style="list-style-type: none">• The Database-per-Service Pattern• Polygot Persistence• Decompose services by scalability• The Scale Cube• Microservices Decomposition Pattern• Microservices Communications Patterns	<ul style="list-style-type: none">• HTTP Based RESTful API• GraphQL API• gRPC API• WebSocket API• Gateway Routing Pattern• Gateway Aggregation Pattern• Gateway Offloading Pattern• API Gateway Pattern• Backends for Frontends Pattern-BFF• Service Aggregator Pattern• Service Registry/Discovery Pattern	<ul style="list-style-type: none">• Single-receiver Message-based Communication (one-to-one model)• Multiple-receiver Message-based Communication (one-to-many model-topic)• Dependency Inversion Principles (DIP)• Fan-Out Publish/Subscribe Messaging Pattern• Topic-Queue Chaining & Load Balancing Pattern	<ul style="list-style-type: none">• List products• Filter products as per brand and categoriesPut products into the shopping cart• Apply coupon for discountsCheckout the shopping cart and create an order• List my old orders and order items history
				Non-FR
				<ul style="list-style-type: none">• High Scalability• High Availability• Millions of Concurrent User• Independent
				Mehmet Ozkaya 358

Design: Microservices Architecture with Fan-Out Publish/Subscribe Messaging Pattern



Adapt: Microservice Architecture with Fan-Out Publish/Subscribe Messaging Pattern



Frontend SPAs

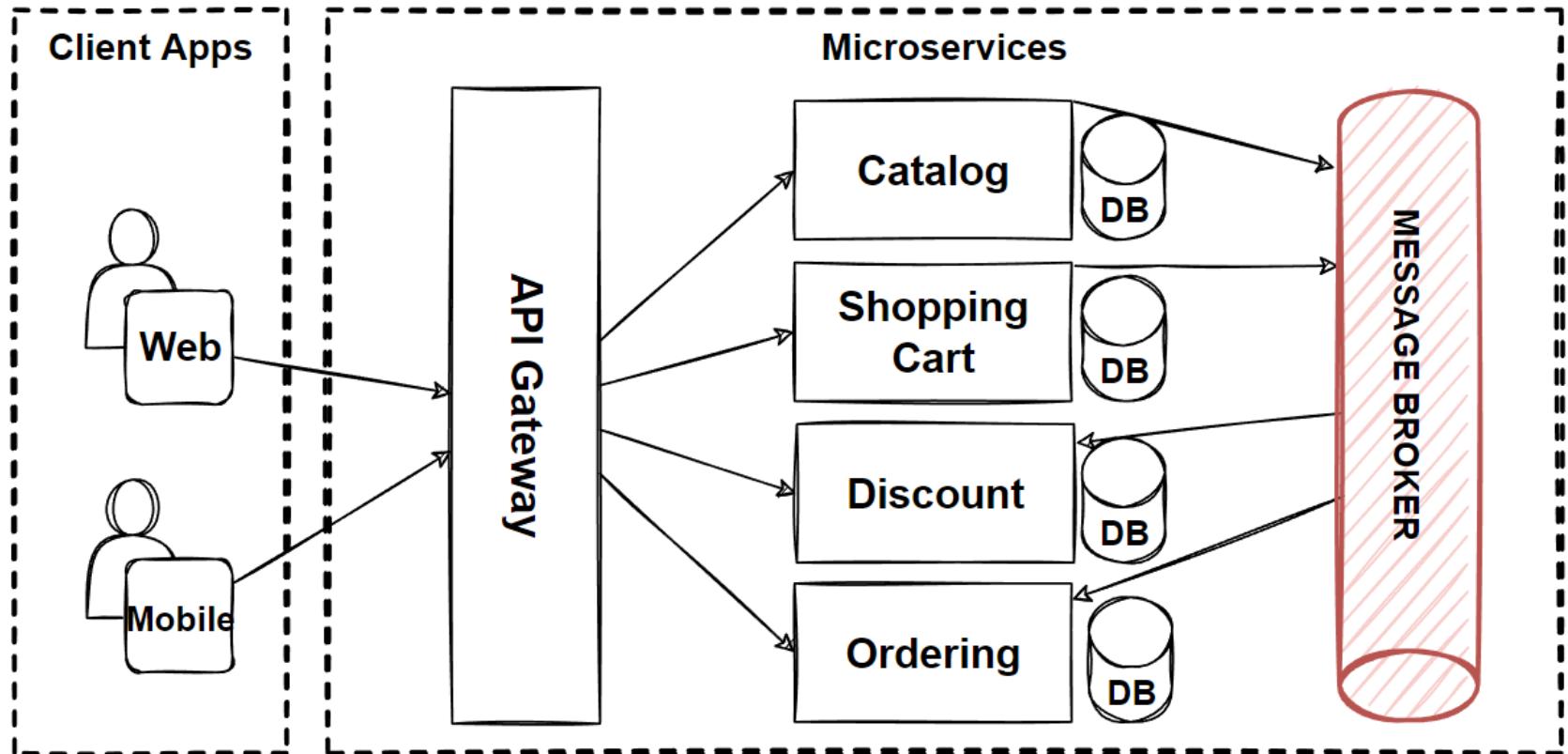
- Angular
- Vue
- React

API Gateways

- Kong Gateway
- Tyk API Gateway
- Express Gateway
- Amazon AWS API Gateway

Message Brokers

- Kafka
- RabbitMQ
- Amazon EventBridge, SNS



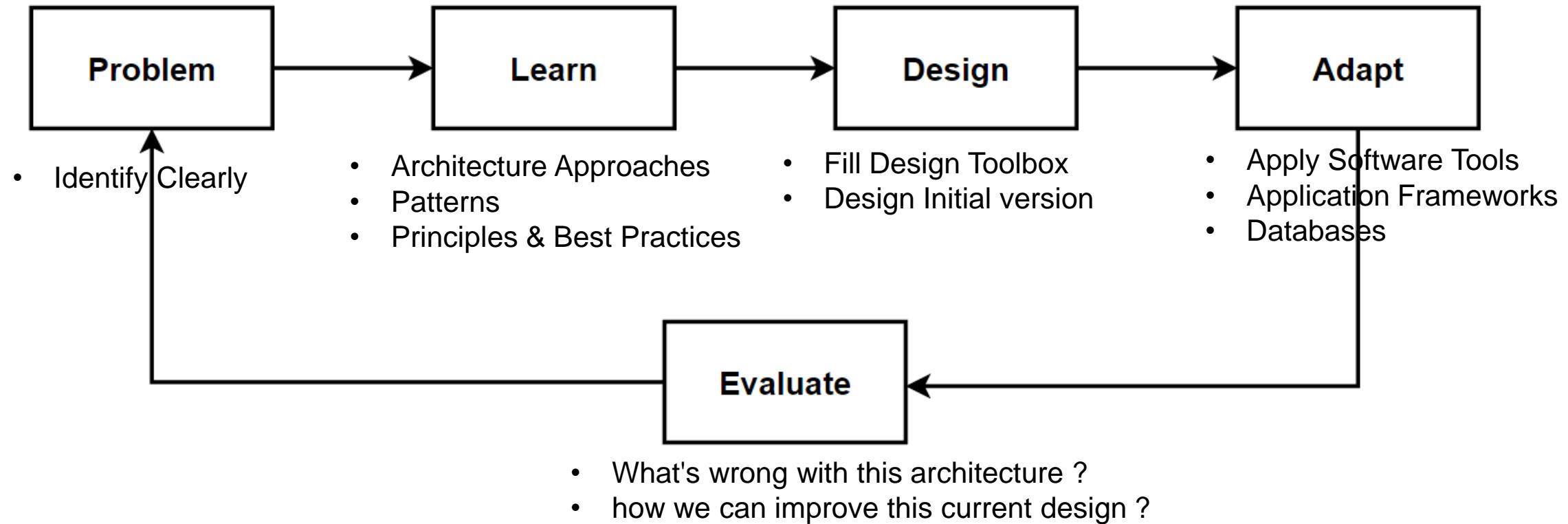
Backend Microservices

- Java – Spring Boot
- .Net – Asp.net
- JS – NodeJS

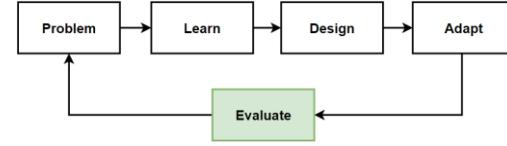
Database

- MongoDB – NoSQL Document DB
- Redis – NoSQL Key-Value Pair
- Postgres – Relational

Way of Learning – The Course Flow



Evaluate: Microservice Architecture with Fan-Out Publish/Subscribe Messaging Pattern

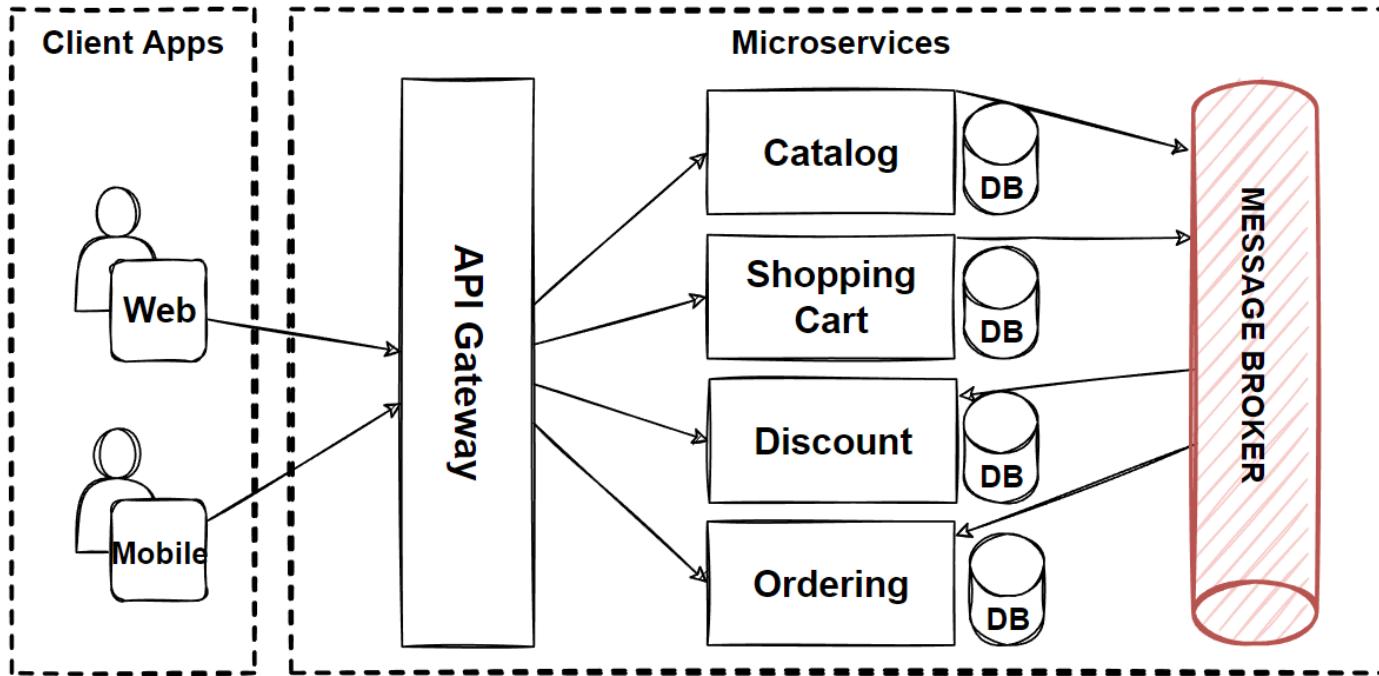


Benefits

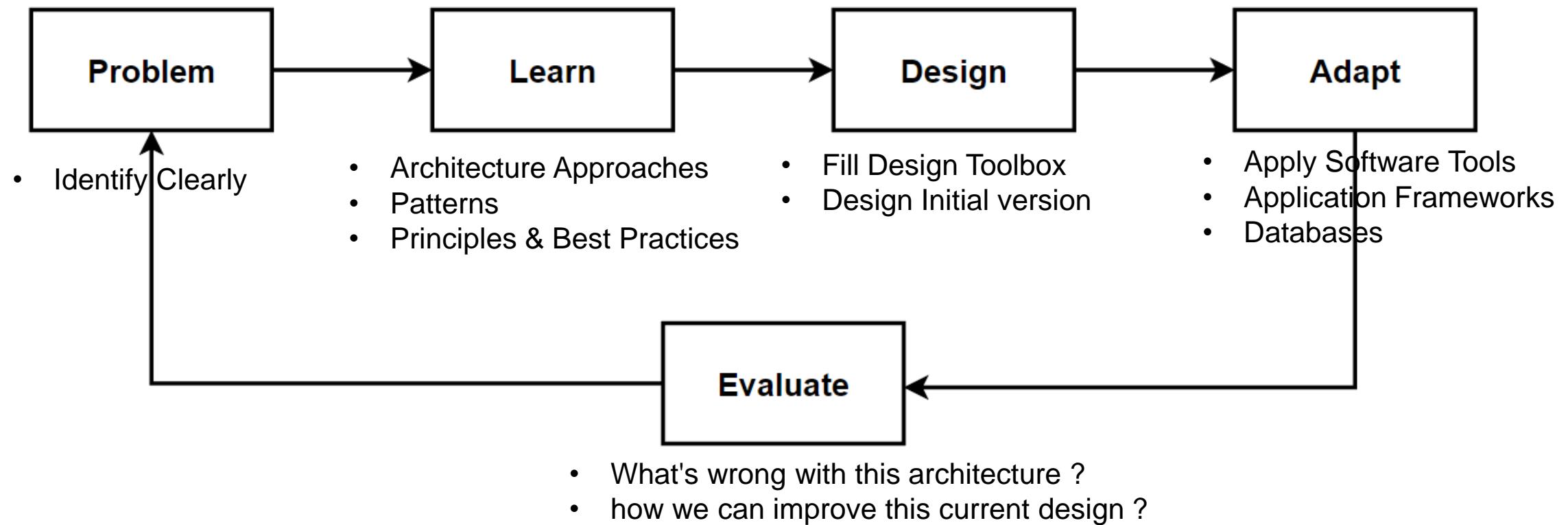
- Adding new subscriber services very easily
- Increase flexible scalability
- Distribute received messages at scale
- Event-driven Microservices

Drawbacks

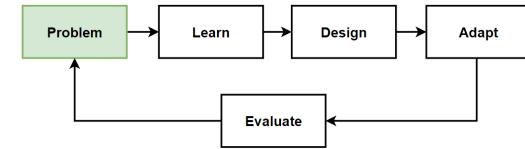
- Single Point of Failure - Message Broker
- Hard to Debugging
- Not Good for FIFO and exact-once requirements
can cause performance and scalability problems



Way of Learning – The Course Flow



Problem: Database Bottlenecks when Scaling



Considerations

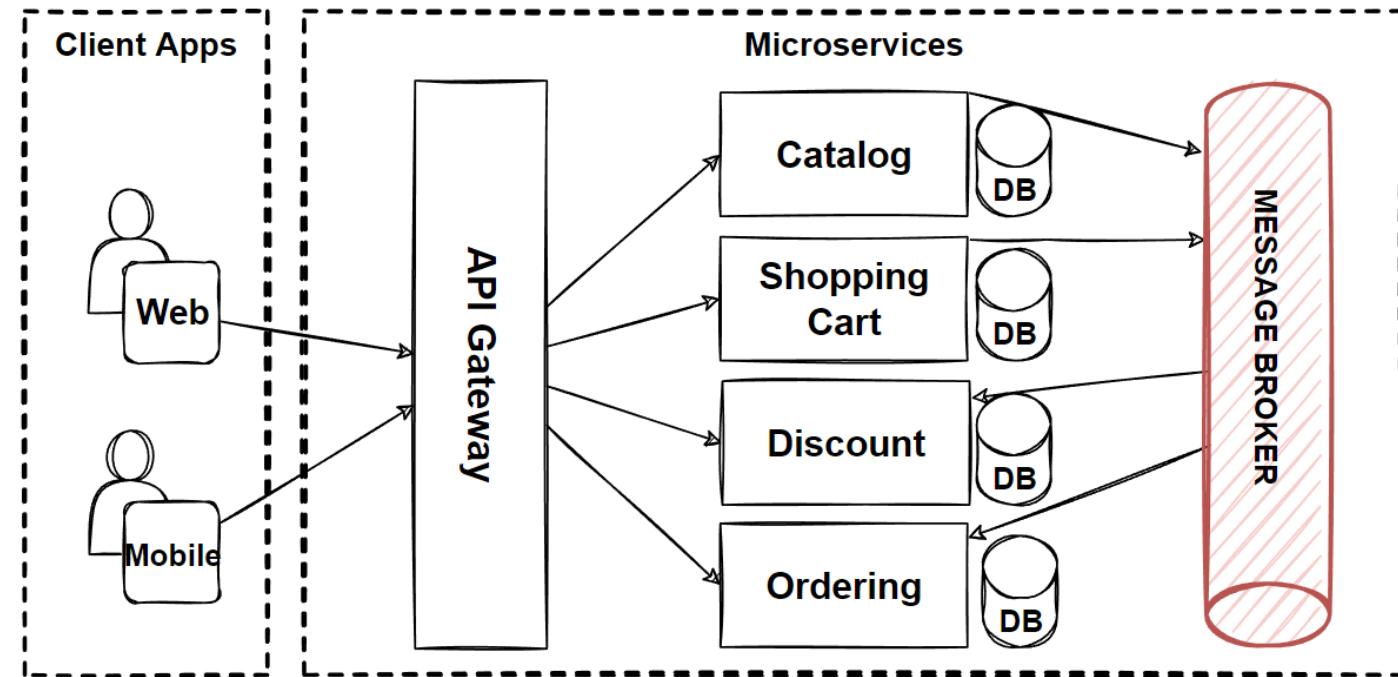
- State-less or state-ful services
- **Are they can scale independently ?**

Problems

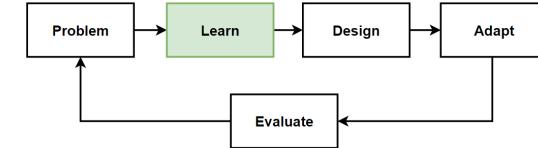
- Database are stateful service
- Scaling stateful services are not easy
- Vertical scaling has limits need to scale horizontally

Solutions

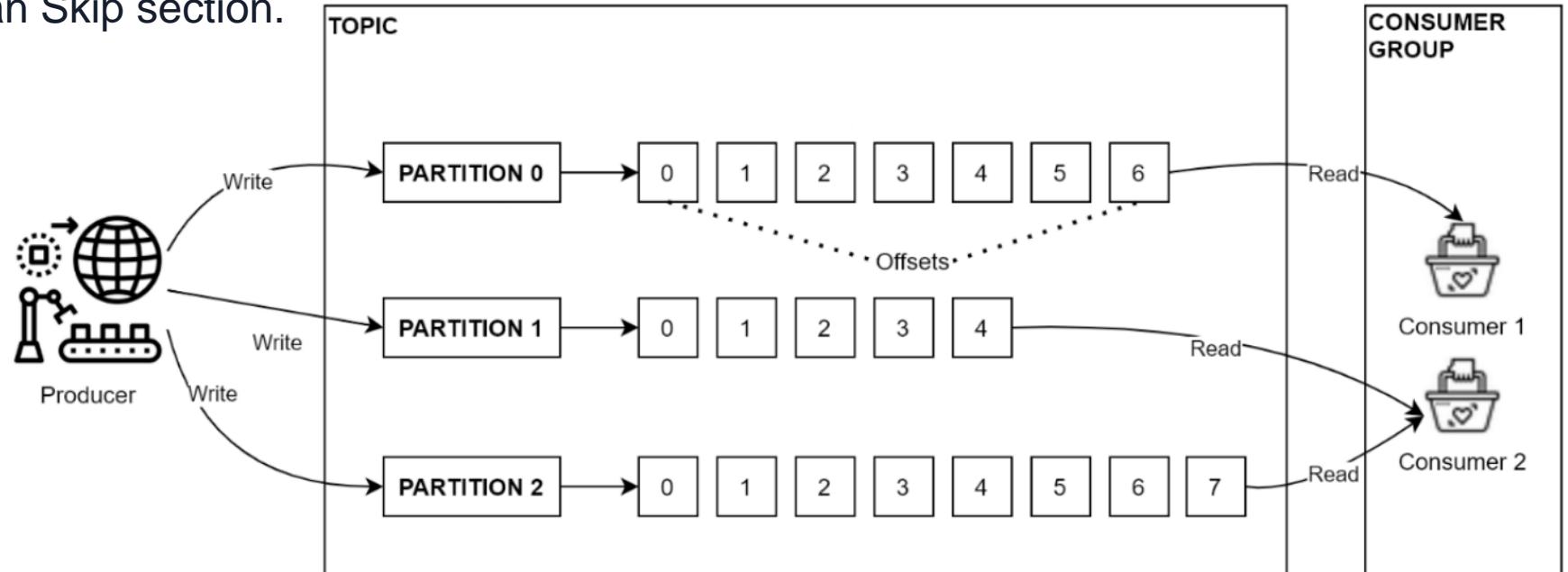
- Scale Stateful Application Horizontal Scaling
- Service and Data Partitioning along Business Boundaries - Shards/Pods
- Use NoSQL Database to gain easy partitioning features



Optional Section: Learning Kafka and RabbitMQ Architecture



- Message broker system that provide to communicate Asynchronously between microservices.
- Kafka and RabbitMQ is good example of distributed architecture that is good to learn with microservices.
- Optional Section, you can Skip section.



Kafka and RabbitMQ Architectures [OPTIONAL]

Kafka Components:Topic, Partitions, Offset and Replication Factor

Apache Kafka Cluster Architecture:Kafka Brokers and Zookeeper

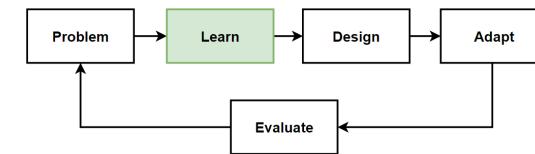
Apache Kafka Core APIs:Producer, Consumer, Streams and Connect API

RabbitMQ Components: Producer, Queue, Consumer, Message, Exchange, Binding

RabbitMQ Exchange Types

What is Apache Kafka ?

- Apache Kafka is the most popular **open-source event streaming** platforms.
- Designed for **horizontally scalable, distributed, and fault-tolerant**
- Distributed **publish-subscribe event streaming** platform
- Publishers **send** messages to **topics**, **subscriber** of a **topic** receives all the messages published to the topic.
- **Event-driven Architecture:** Kafka is used as an event router, and the microservices publish and subscribe to the events.
- **Distributed working**, and it provides a **horizontal scalable** system.
- Built on top of **ZooKeeper synchronization** service.
- **Key Components:** Topics, Partitions, Brokers, Producer, Consumer, Zookeeper



kafka

Apache Kafka Benefits

- **Reliability**

Kafka is distributed, partitioned, replicated and fault tolerance. So it is also High Availability.

- **Scalability**

Since its distributed architecture, Kafka scales easily without any down time.

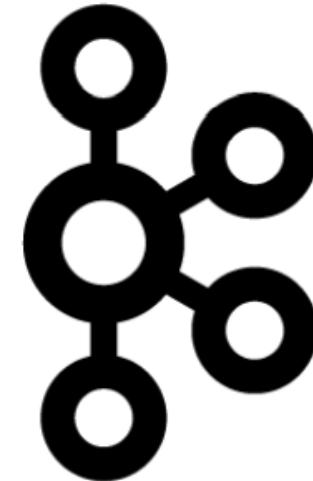
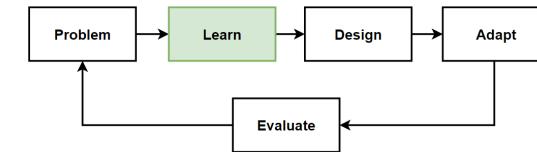
- **Durability**

Kafka uses Distributed commit log. This persists to events on disk as log commit very fast. Its durable for infinitive or a given parameter days or weeks.

- **Performance**

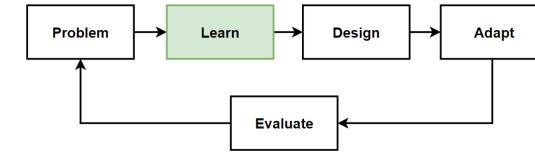
Kafka has high throughput for both publishing and subscribing messages. It is distributed event streaming platform capable of handling trillions of events a day.

- Kafka has better throughput, built-in partitioning, replication, and fault-tolerance architecture.



kafka

Apache Kafka Use Cases



▪ Messaging

Message brokers are used to decouple services from each other. In event-driven architecture, Kafka is used as an event router, and microservices publish and subscribe to the events.

▪ Metrics

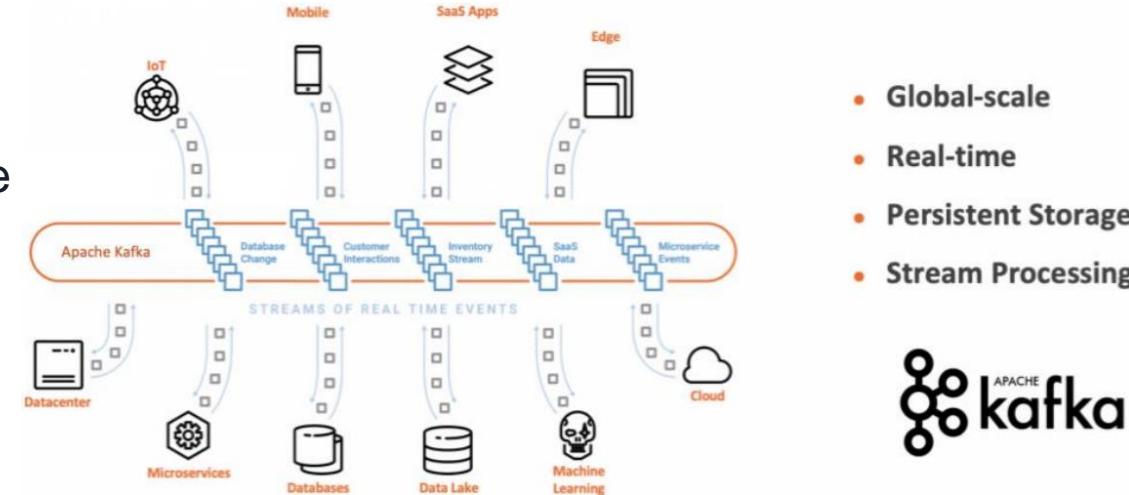
Kafka is often used for operational monitoring data. Aggregating statistics from distributed applications to produce centralized feeds of operational data.

▪ Log Aggregation

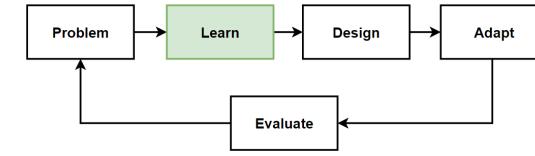
Log aggregation solution that collect logs from multiple services. Log aggregation collects physical log files servers and puts them in a central place for processing.

▪ Stream Processing

Kafka process data in processing pipelines consisting of multiple stages. Storm and Spark Streaming read data from a topic and processes it.



Apache Kafka Use Cases

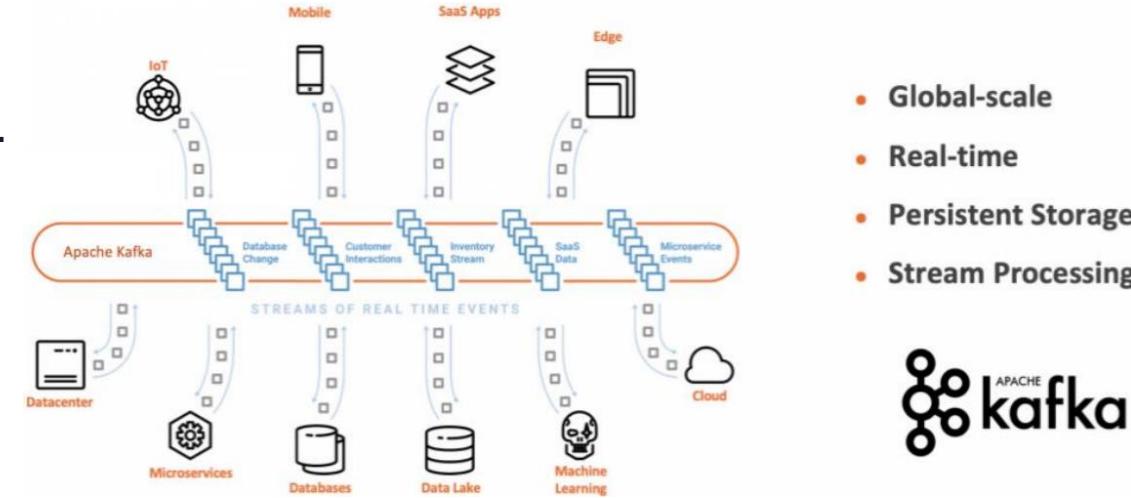


- **Website Activity Tracking**

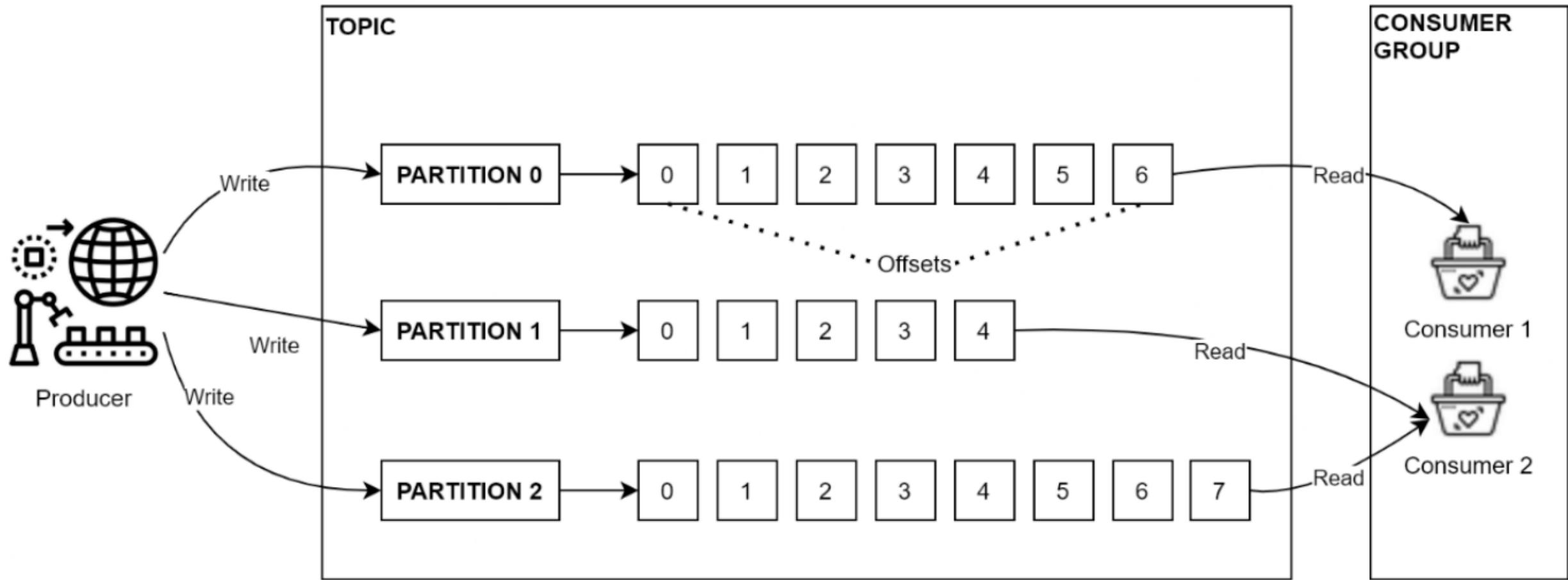
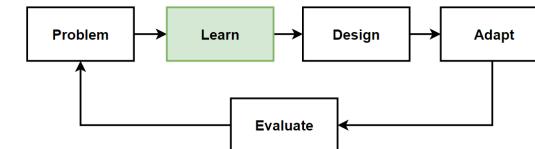
Kafka follows the user activity tracking pipeline as a set of real-time publish-subscribe feeds.

- **Event Sourcing**

Kafka is used with the event-driven architecture, so in that case it can also be used for event storing and event sourcing.

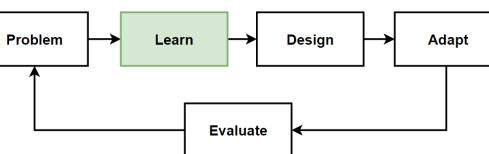
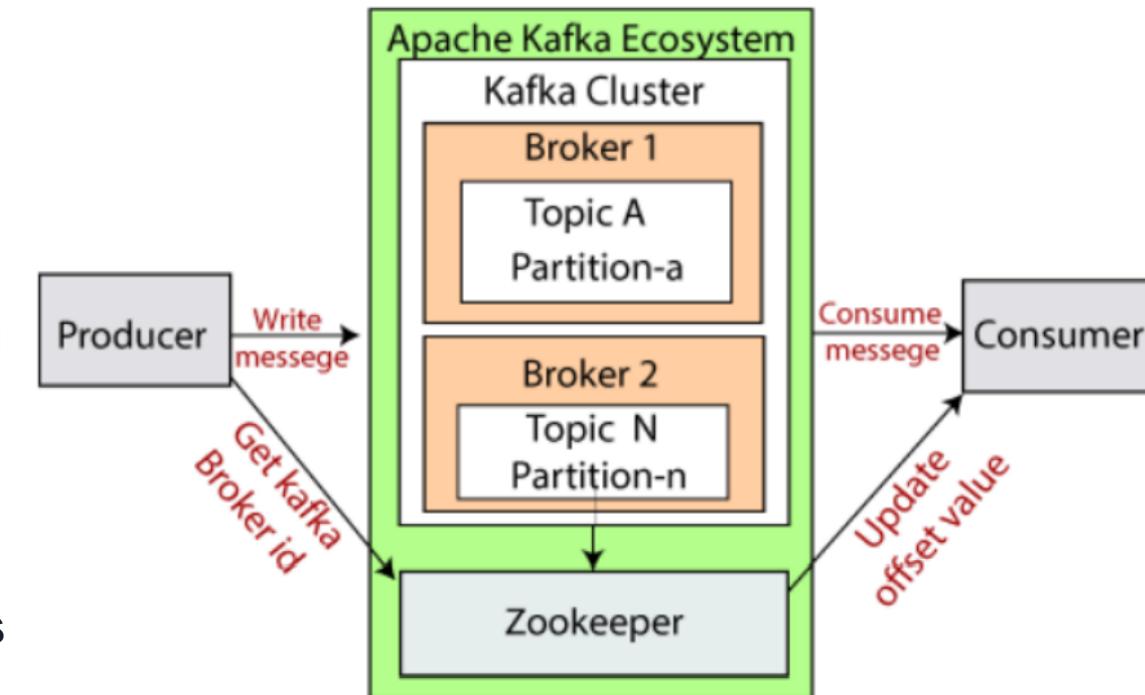


Kafka Components: Topic, Partitions, Offset and Replication Factor

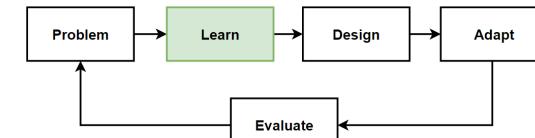


Apache Kafka Cluster Architecture: Kafka Brokers - Kafka Cluster

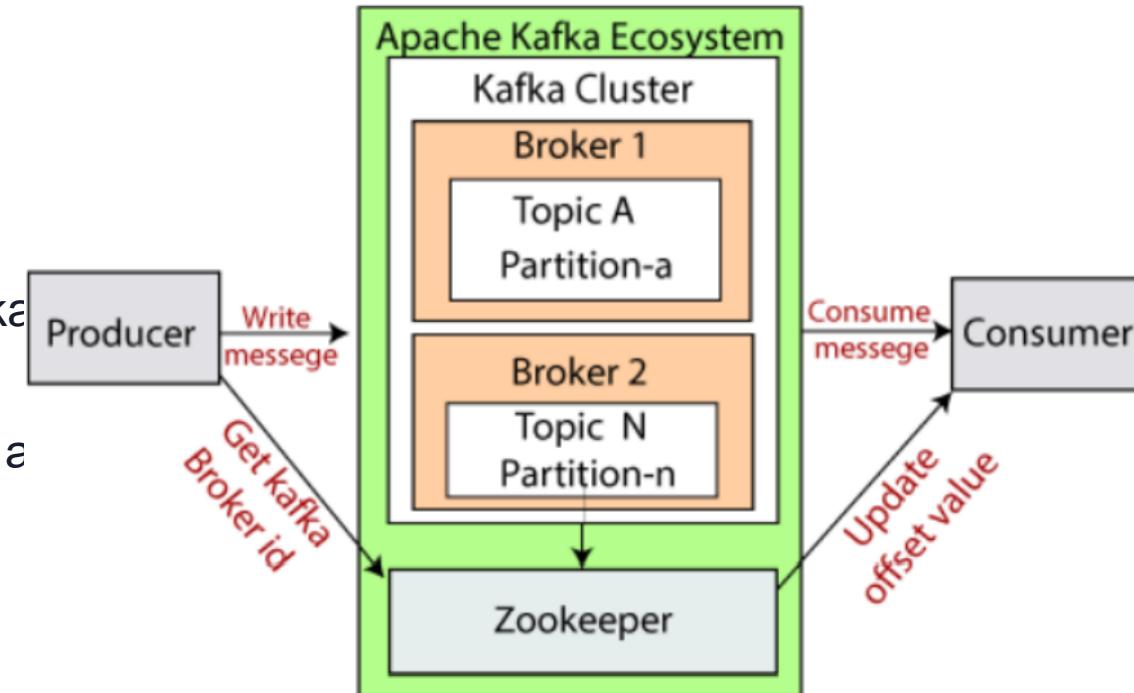
- Kafka can be **set up across multiple servers**, which are called **Kafka Brokers**.
- Mostly prefer to run **multiple Kafka broker instance** at the same time into our **Kafka cluster**.
- With multiple brokers, we can get the benefit of **data replication, fault tolerance, and high availability** of your Kafka cluster.
- **Kafka cluster** consists of multiple brokers to **maintain load balance**.
- Kafka brokers are **stateless**, so they use **ZooKeeper** for maintaining their cluster state.
- **Kafka broker instance** can handle hundreds of **thousands of reads and writes per second**.
- Kafka broker **leader election** can be done by **ZooKeeper**.



Apache Kafka Cluster Architecture: Zookeeper



- How we are managing and coordinating Kafka brokers ?
- Kafka Cluster and including Kafka brokers need to manage.
- **Management job** is handled by the **master node** in the distributed systems.
- This master node **manage** and **maintenances** the **other worker(nodes)** in order to work correctly.
- In Apache Kafka, **there isn't any master** in the Apache Kafka Cluster.
- Apache Kafka Cluster isn't a master-worker architecture; it's a **master-less architecture**.
- **ZooKeeper** is used for **managing** and **coordinating Kafka broker**. The zookeeper manages and maintains the brokers in the cluster.
- Finds which brokers have been **crashed** or which broker recently **added** to the cluster and **manage** their **lifecycle**.



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

- **Producers API**

Producers push data to brokers. Publish a stream of data to Kafka topics. When the new broker is started, all the producers search and sends a message to that new broker.

- **Consumer API**

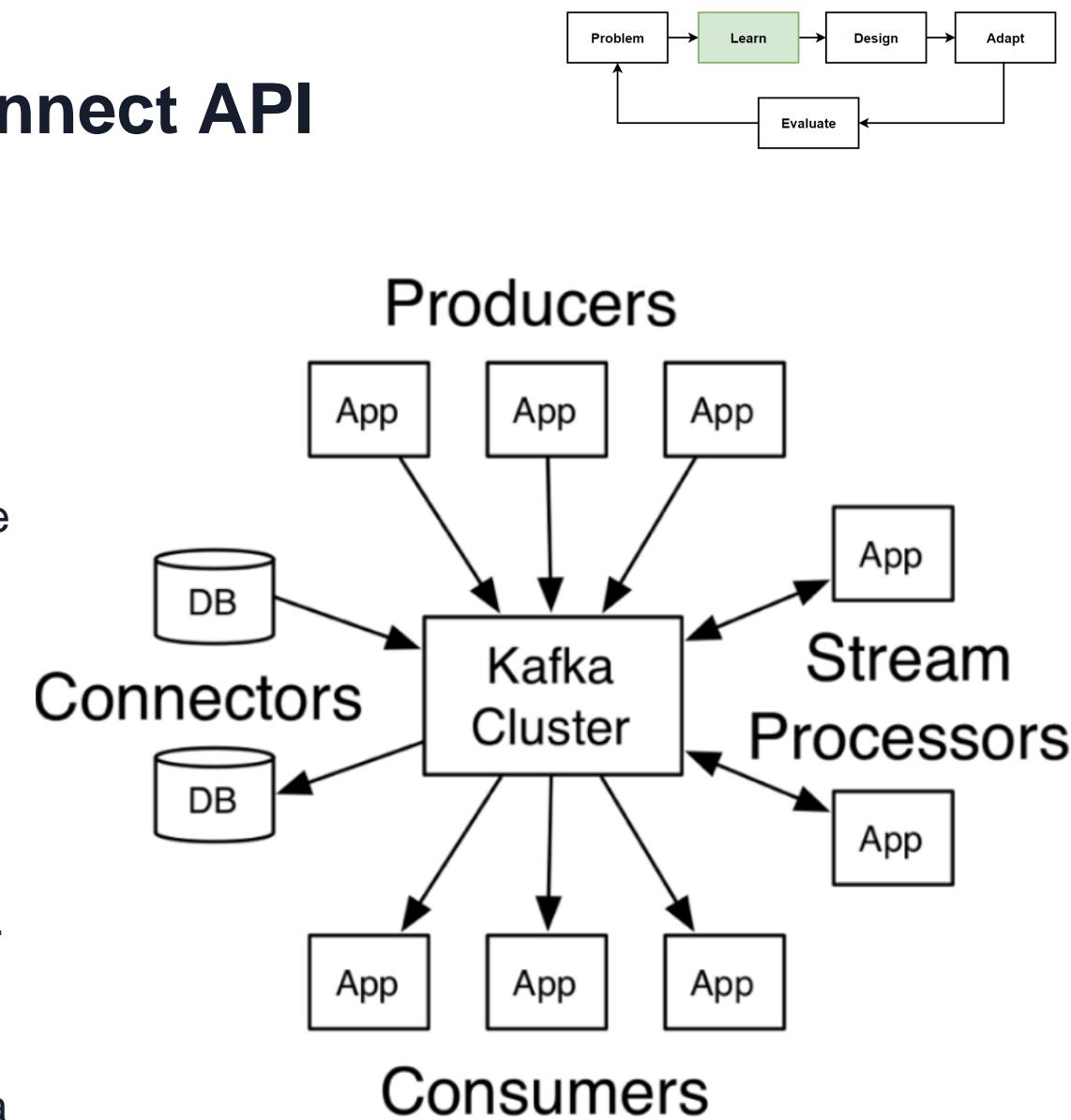
Consumer API provide to subscribe to topics and process the streams of data. Manage messages have been consumed from Kafka by using partition offset. Consumer offset values are managed by ZooKeeper.

- **Streams API**

Streams API that is useful for transforming data from one topic to another. Allows applications to act as a stream processor. Transforming the input streams to output streams.

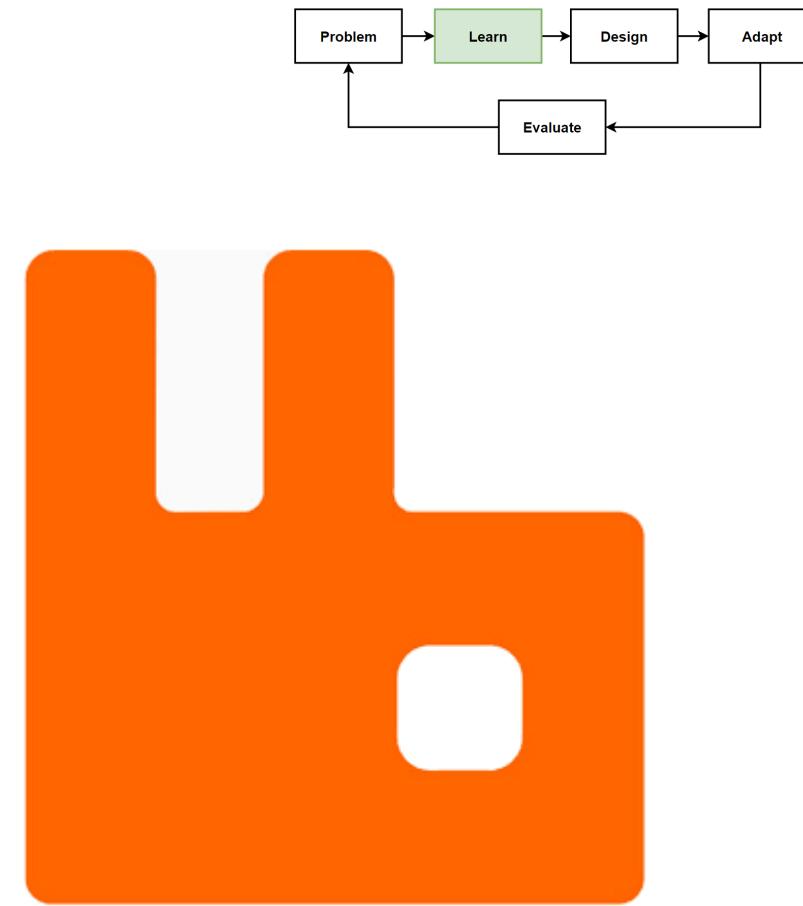
- **Connect API**

Provide to implement connectors that pulling data from external systems into Kafka. Use stream data between Kafka and other external systems. Collect metrics from servers into Kafka topics.



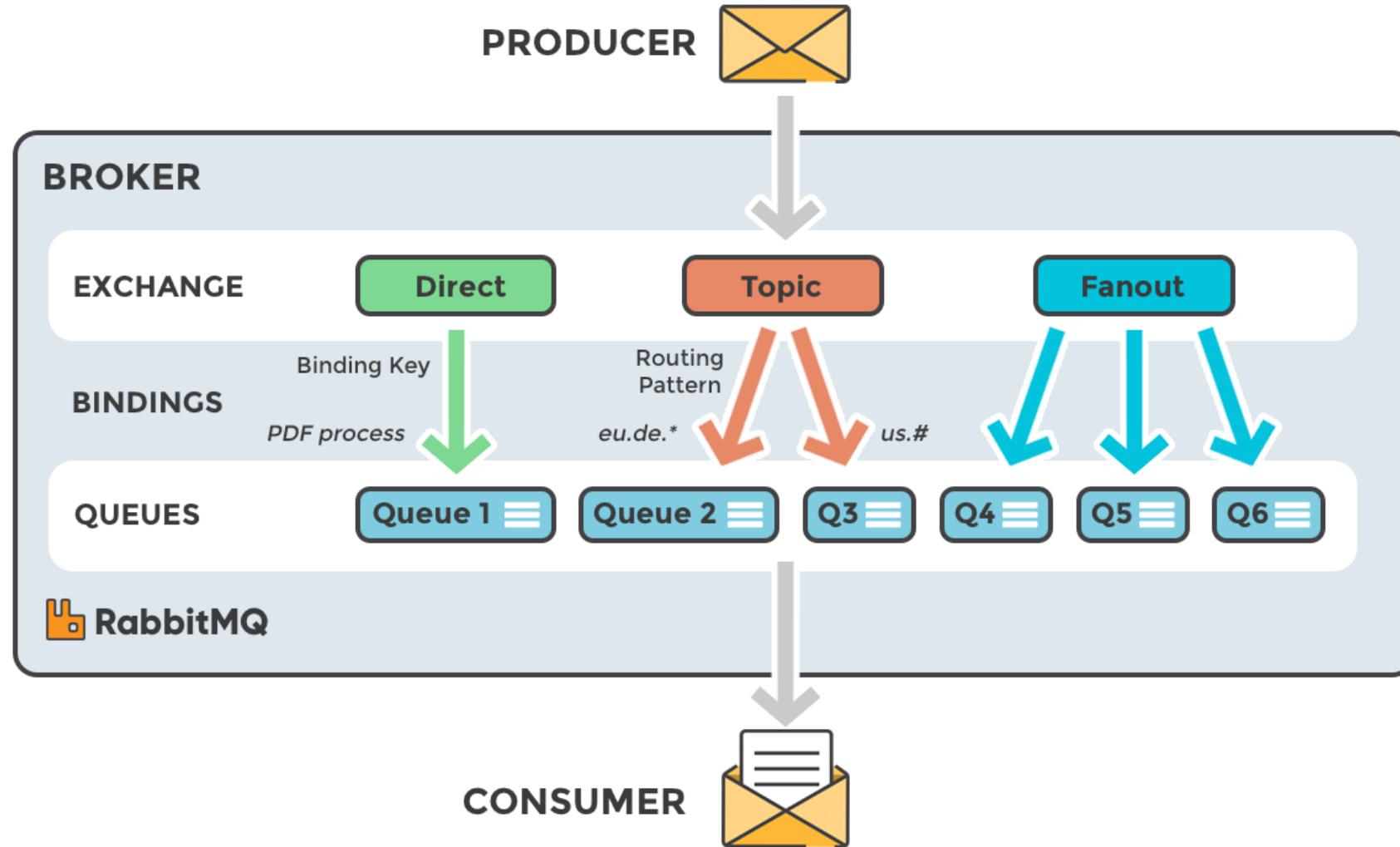
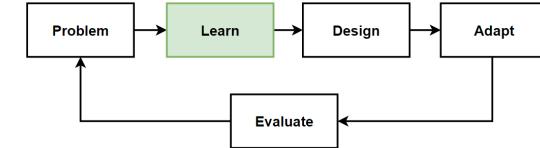
What is RabbitMQ ?

- **RabbitMQ** is a **message broker software** that implements the **Advanced Message Queuing Protocol (AMQP)**.
- It allows applications to communicate with each other by **sending and receiving messages** through **queues**.
- **RabbitMQ** is a **message queuing system** that transmit a message received from any source to another source.
- Similar ones can be listed as **Apache Kafka, Msmq, Microsoft Azure Service Bus, Kestrel, ActiveMQ**.
- All **transactions** can be **listed in a queue** until the source to be transmitted gets up.
- It allows to **send and receive messages asynchronously**.
- **RabbitMQ's** support for **multiple operating systems** and open source code is one of the most preferred reasons.
- Main Components of RabbitMQ: **Producer, Queue, Consumer, Message, Exchange, Binding** and **FIFO**.

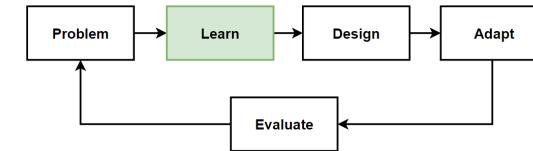


RabbitMQ

RabbitMQ Components: Producer, Queue, Consumer, Message, Exchange, Binding



RabbitMQ Queue Properties



- **Queue Name**

The name of the queue we have defined.

- **Durable**

Determines the lifetime of the queue. If we want persistence, we have to set it true.

- **Exclusive**

Contains information whether the queue will be used with other connections.

- **AutoDelete**

Contains information about deletion of the queue with the data sent to the queue passes to the consumer side.



RabbitMQ Exchange Types

- RabbitMQ provides four types of exchanges:
Direct, Fanout, Topic, and Headers.

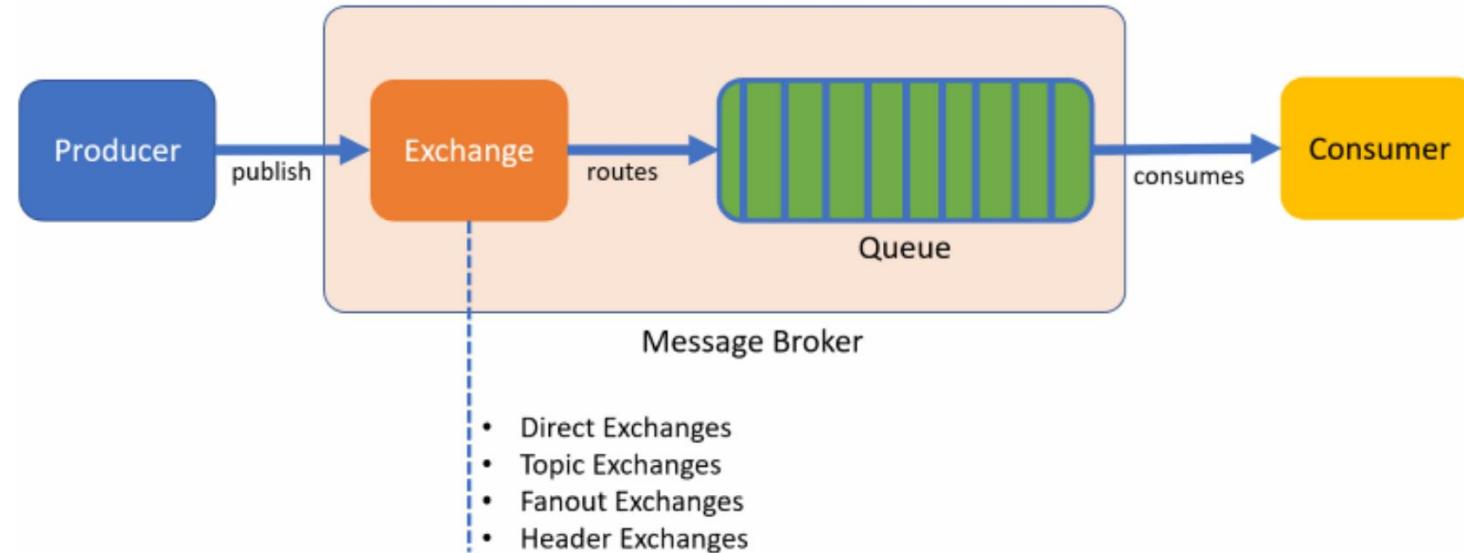
- **Direct Exchange**

Uses of a single queue is being addressed.
The most appropriate queue is reached with
the relevant direct exchange.

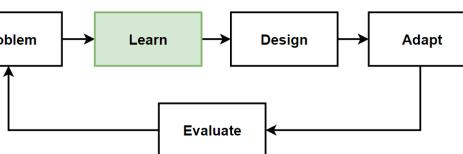
- **Topic Exchange**

Messages are sent to different queues
according to their subject. Incoming
message is classified and sent to the related
queue.

- Variation of the Publish / Subscribe pattern.
Topic Exchange should be used to
determine what kind of message they want
to receive.

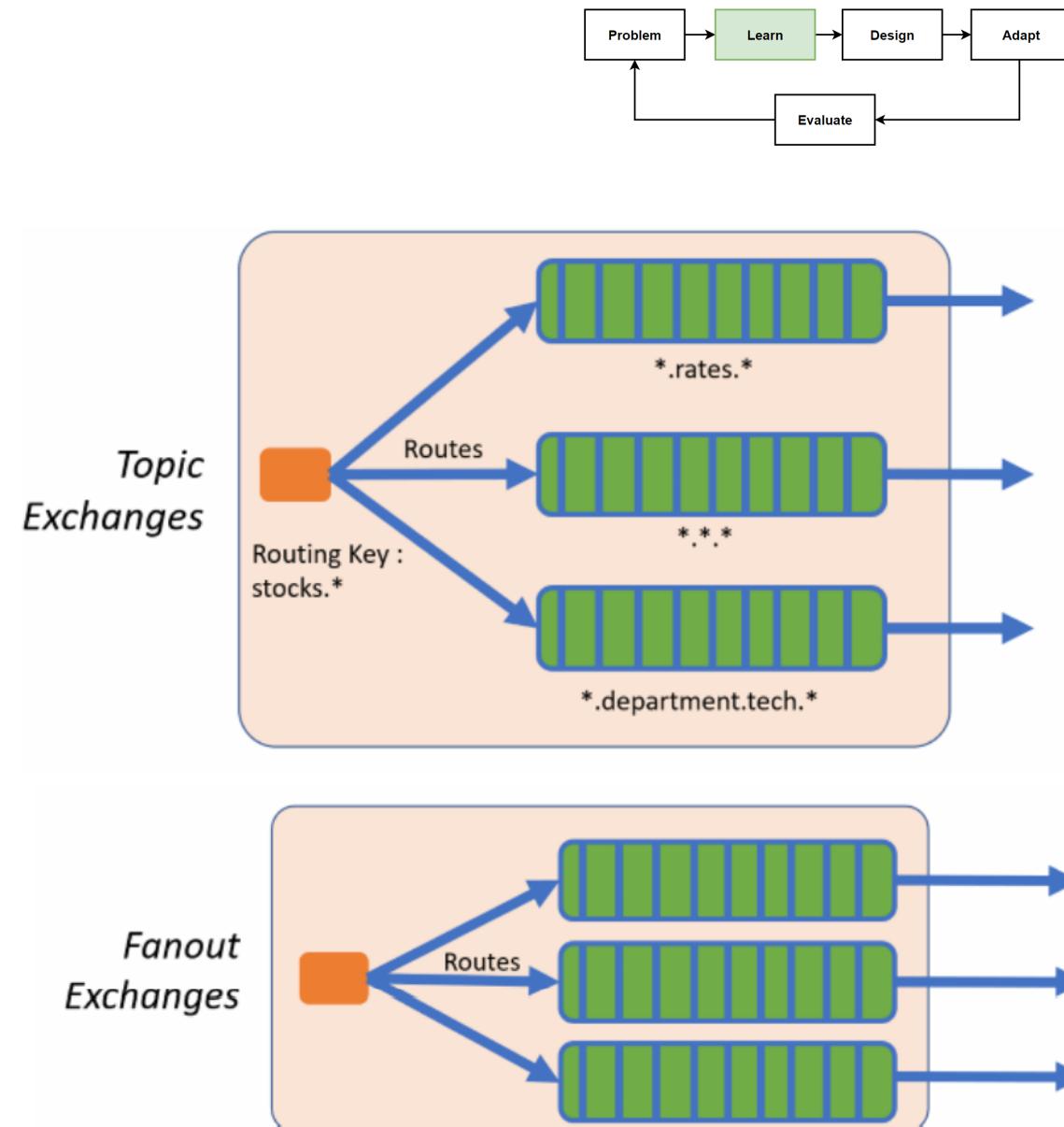


<https://www.rabbitmq.com/tutorials/amqp-concepts.html>



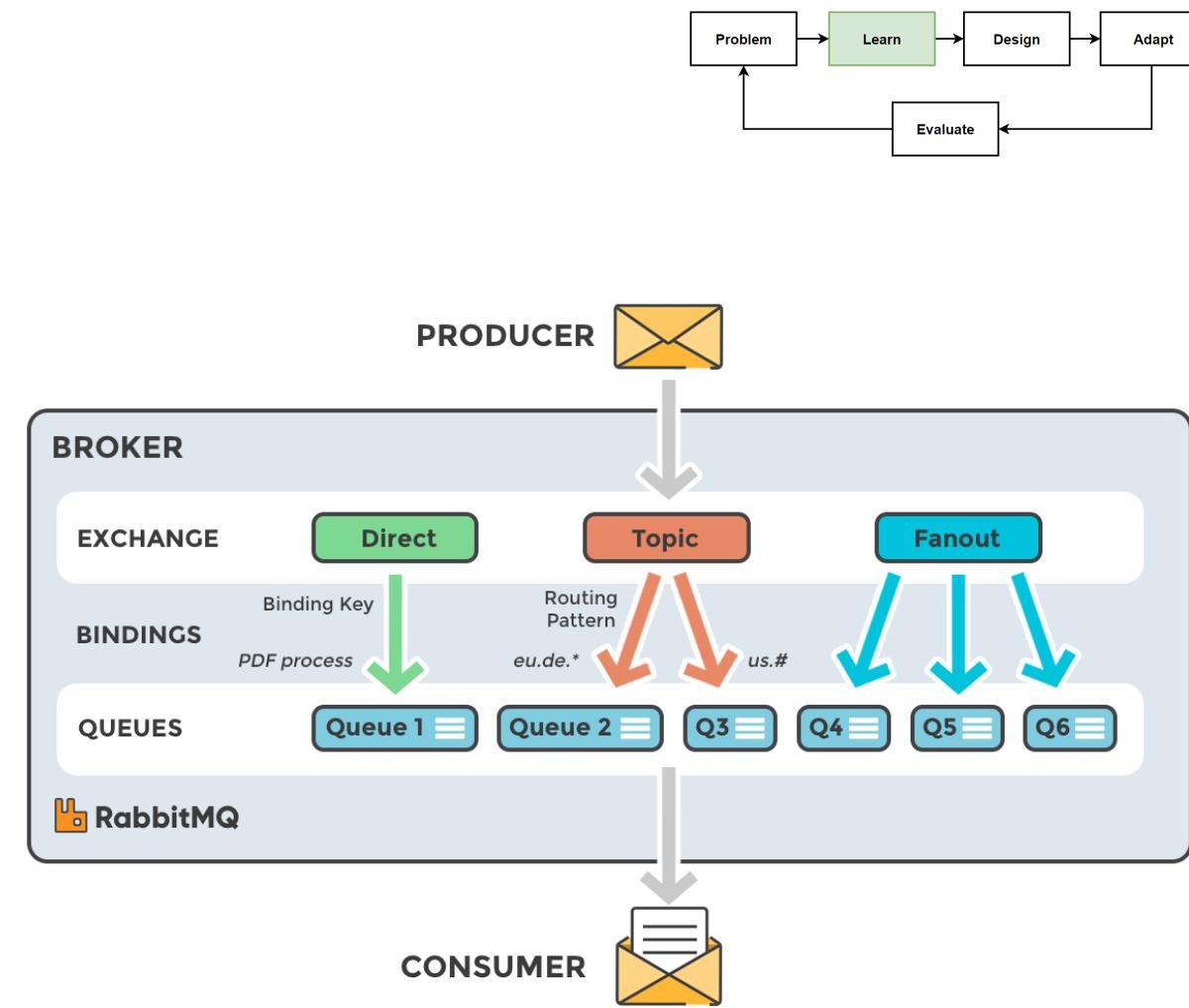
RabbitMQ Exchange Types-2

- **Fanout Exchange**
Used in situations where the message should be sent to more than one queue. Especially applied in Broadcasting systems.
- Mainly used for games for global announcements.
- **Headers Exchange**
Guided by the features added to the header of the message. Routing-Key used in other models is not used.
- Transmits to the correct queue with a few features and descriptions in message headers.
- The attributes on the header and the attributes on the queue must match each other's values.



RabbitMQ Architecture

- The producer **never sends any messages directly** to a queue. Instead, the producer can only **send messages** to an **exchange**.
- **Exchange has 2 sides:** it **receives messages from producers**, it **pushes them to queues**.
- Exchanges control the **routing of messages to queues** according to **exchange types**.
- RabbitMQ uses a **push model**.
- **Push-based queues** can overwhelm consumers if messages arrive faster than the consumers can process them.
- Each consumer can configure a **prefetch limit**.
- Push model is to **distribute messages individually** and quickly, and ensure that work is **concurrent way**.
- Messages are **processed in the same order** how they **arrived** in the **queue**.



<https://www.cloudamqp.com/img/blog/exchanges-topic-fanout-direct.png>

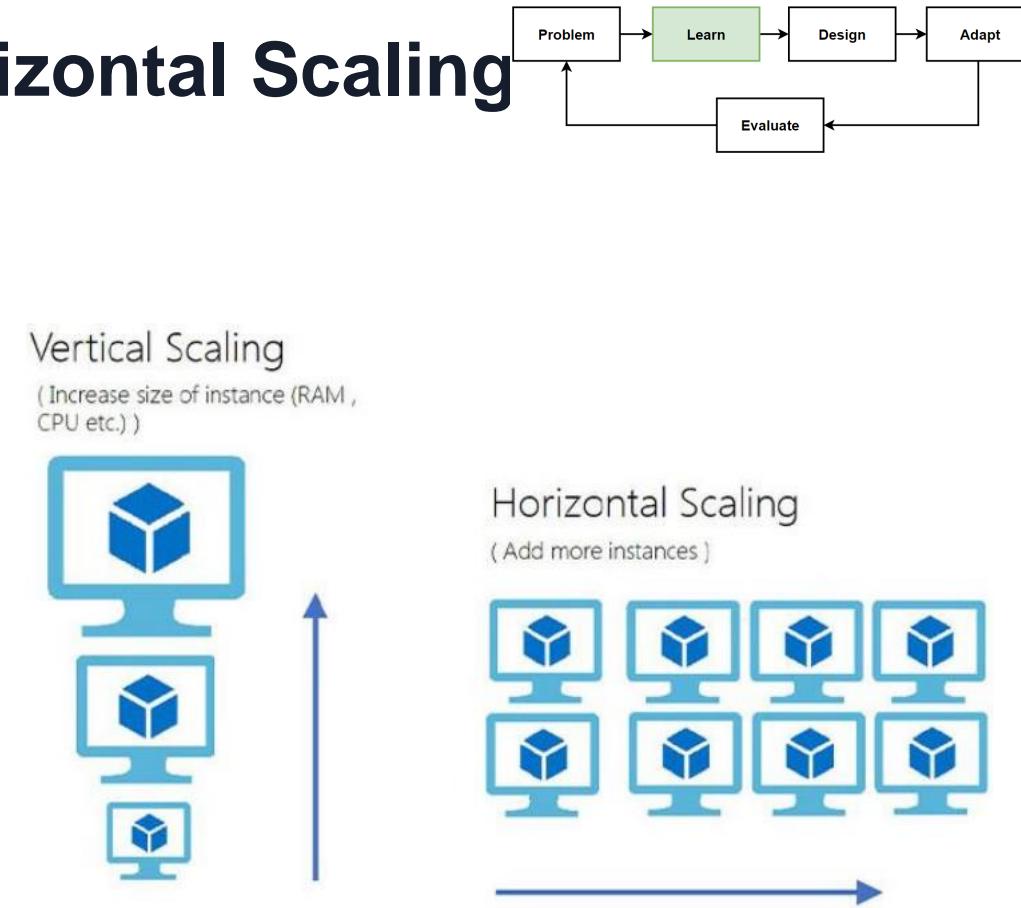
Scale the Microservices Architecture Design

Stateless and Stateful Application Horizontal Scaling

The Scale Cube

Learn: Scalability - Vertical Scaling - Horizontal Scaling

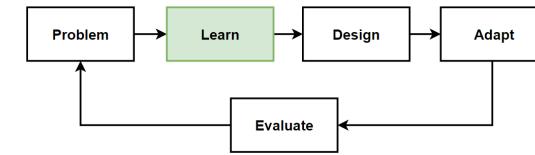
- **Scalability** is the **number of requests** an application can handle
- Measured by the number of requests and it can effectively support **simultaneously**.
- If no longer handle any more simultaneous requests, it has reached its **scalability limit**.
- To prevent **downtime**, and **reduce latency**, you must scale
- **Horizontal scaling** and **vertical scaling** both involve adding resources to your computing infrastructure
- **Horizontal scaling** by adding more machines
- **Vertical scaling** by adding more power



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

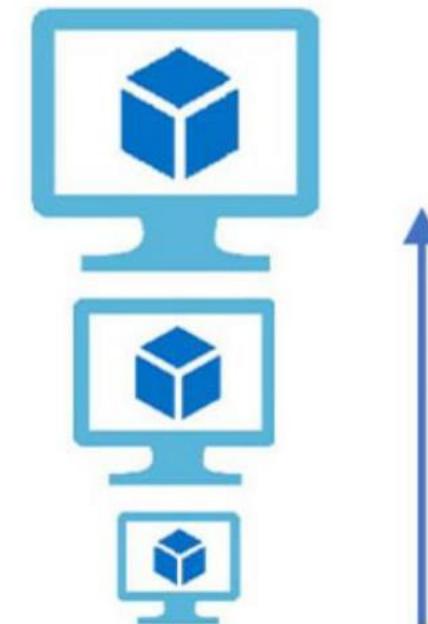
Vertical Scaling - Scale up

- **Vertical scaling** is basically makes the **nodes stronger**.
- Make the server stronger with **adding more hardware**. Adding more resources to a **single node**.
- Make **optimization** the hardware that will allow you to **handle more requests**.
- Vertical scaling keeps your existing infrastructure but **adding more computing power**.
- Your existing **code doesn't need to change**.
- Adding additional **CPU, RAM, and DISK** to cope with an increasing workload.
- By scaling up, you increase the capacity of a single machine. And it has limits. That is named **Scalability Limits**.
- Because even the hardware has **maximum capacity limitations**.
- For handling millions of request, we **need horizontal scaling** or scaling out.



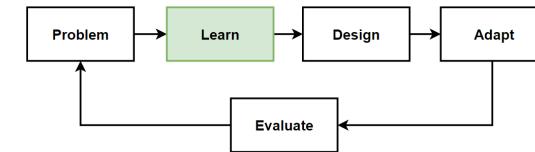
Vertical Scaling

(Increase size of instance (RAM , CPU etc.))



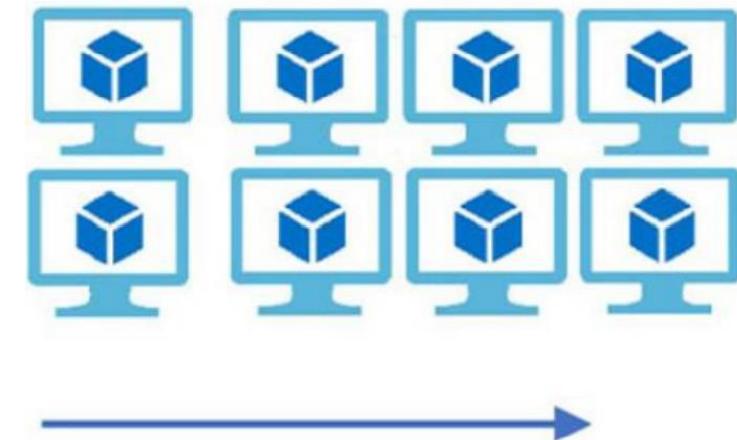
Horizontal Scaling - Scale out

- **Horizontal scaling is splitting the load between different servers.**
- Simply **adds more instances** of machines without changing to existing specifications.
- **Share the processing power and load balancing** across multiple machines.
- Horizontal scaling means **adding more machines** to the resource pool.
- **Scaling horizontally** gives you scalability but also **reliability**.
- Preferred way to scale in **distributed architectures**.
- When splitting into multiple servers, we need to consider if you have a **state or not**.
- if we have a state like database servers, than we need to manage more considerations like **CAP theorem**.

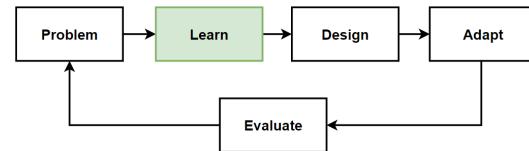


Horizontal Scaling

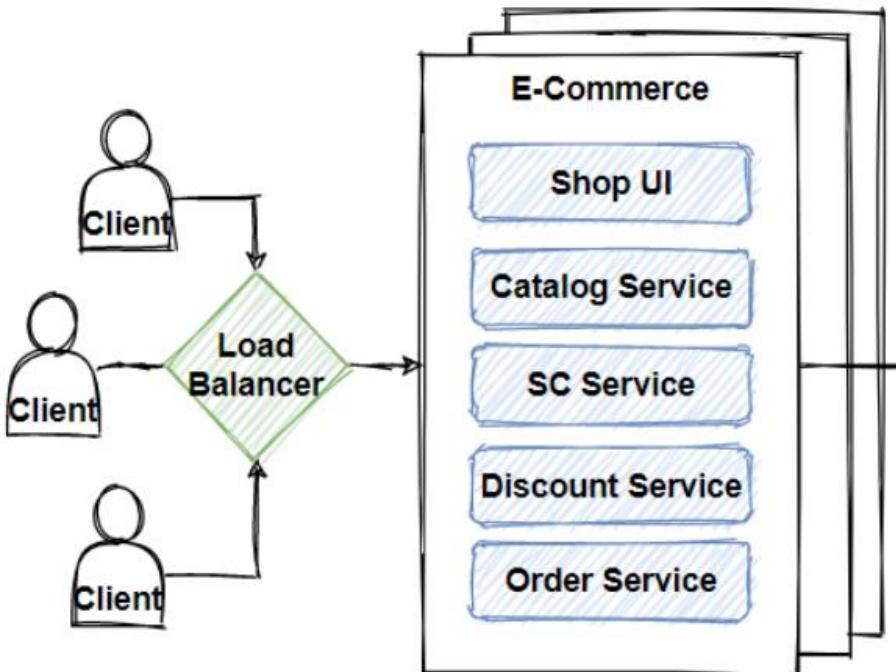
(Add more instances)



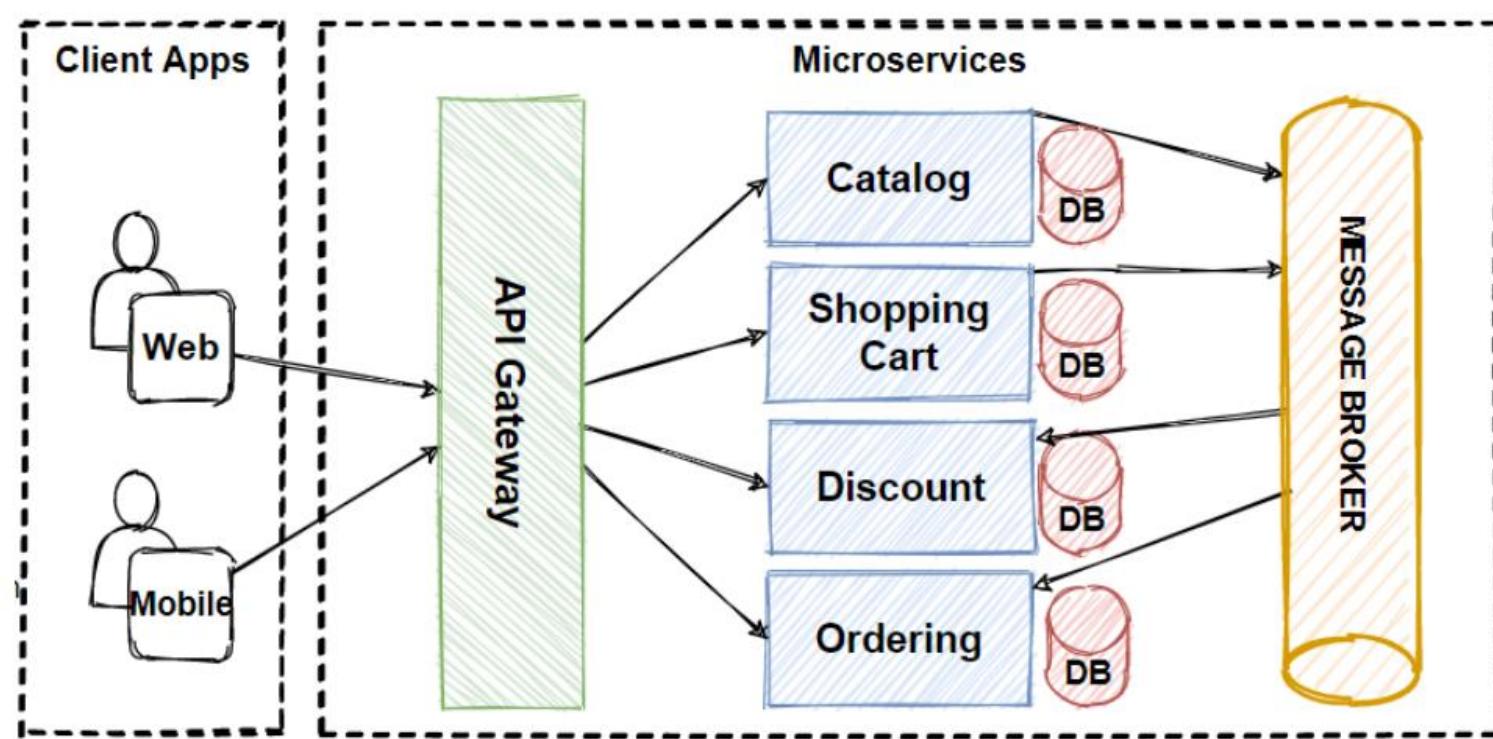
Scale the Microservices Architecture Design



Monolithic Architecture

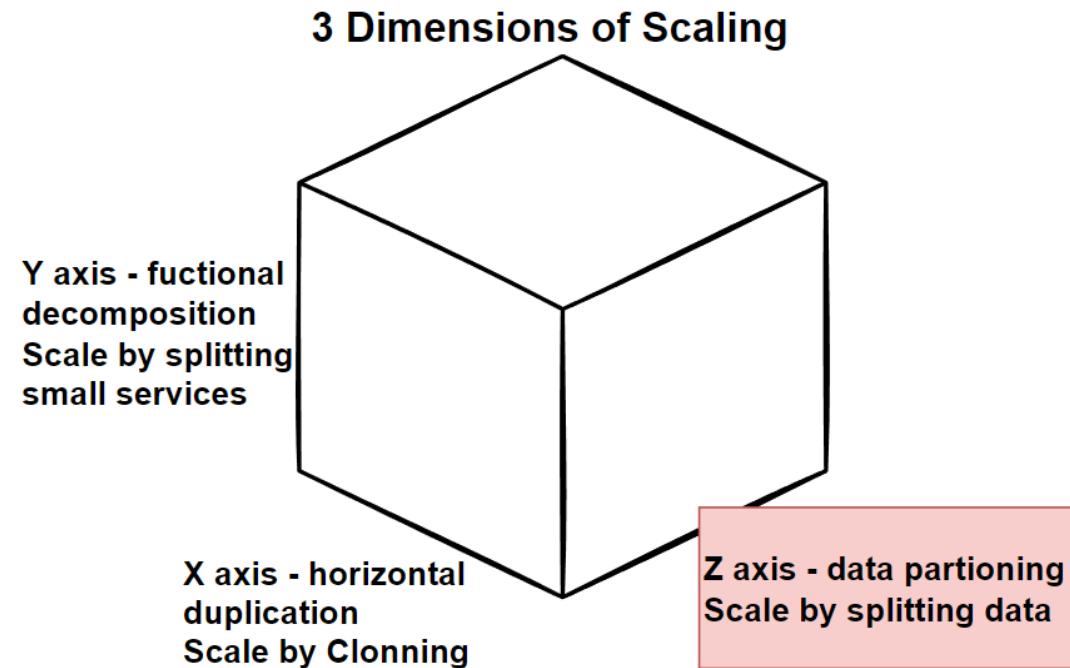
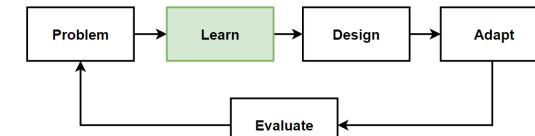


Microservices Architecture - Message Broker

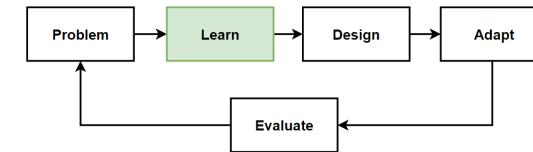


The Scale Cube

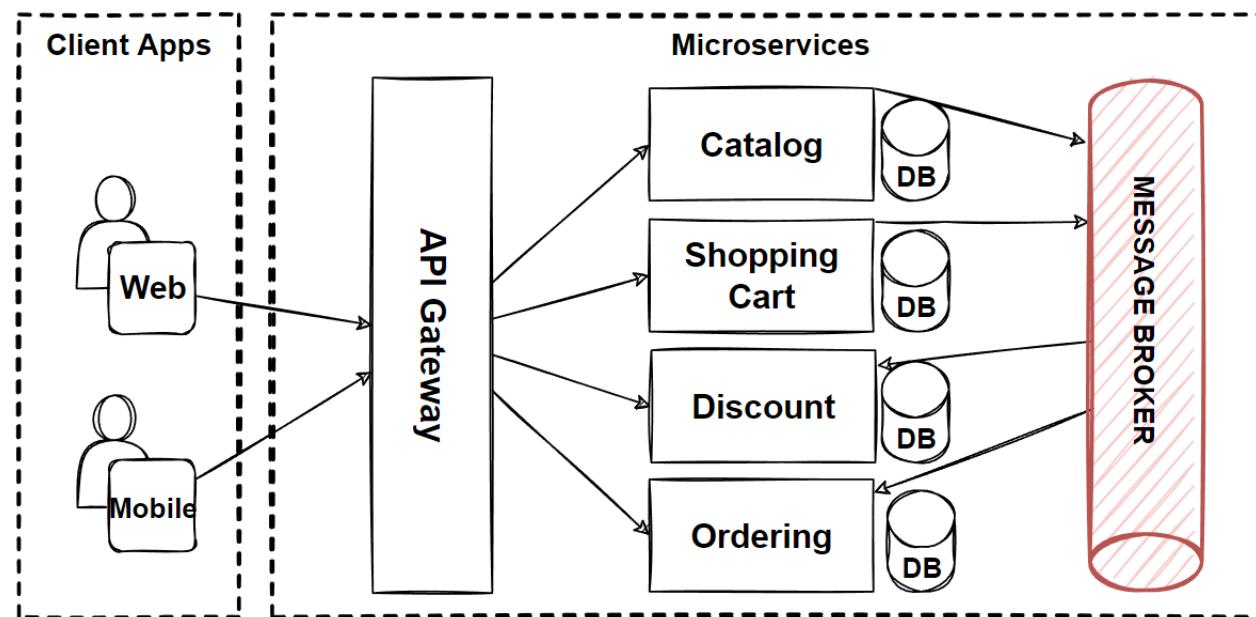
- **X-Axis:** Horizontal Duplication and Cloning of services and data
- **Y-Axis:** Functional Decomposition and Segmentation – Microservices
- **Z-Axis:** Service and Data Partitioning along Customer Boundaries - Shards/Pods
- **Functional decomposition** can be achieved by decoupling your architecture into functions **with Y-axis**.
- **Microservices** is an example for **functional decomposing**.
- **Y-axis** scaling splits the application into multiple, different services.
- **Combining** both **X** and **Y-Axis** scaling with microservices architecture can give **better scalability**.



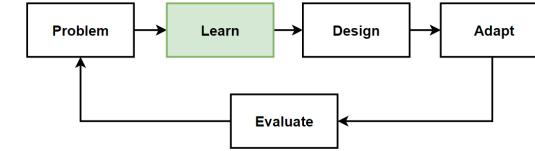
Stateless and Stateful Application Horizontal Scaling



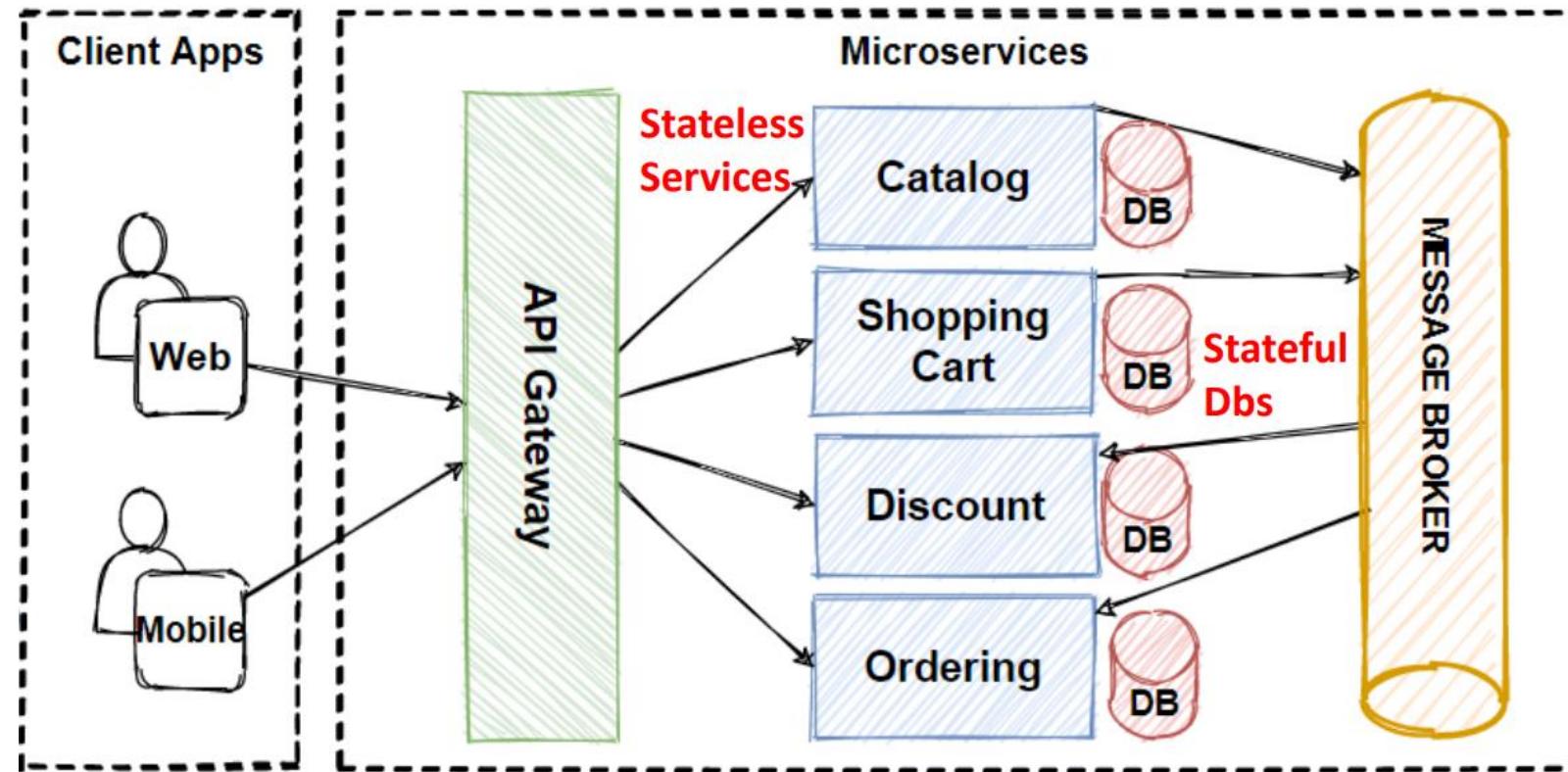
- **Scalability** of our architecture to **handle more** number of **concurrent requests**.
- **Vertical scaling** "scale up" is **increase the hardware** resources for exiting servers.
- **Horizontal Scaling** "scaling out" is **splitting the load** between different servers.
- If the server have a **state or not** ?
- If the scaling server **don't have state**, just put the different servers and have a load balancer that direct the traffic.
- Orchestrator tools like **Kubernetes** have replica sets and we can increase to pod numbers easily.



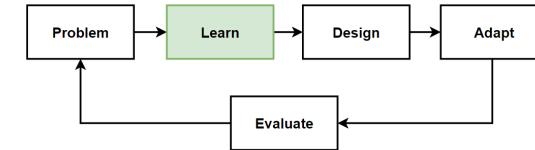
How to Scale Stateful Application Horizontal ?



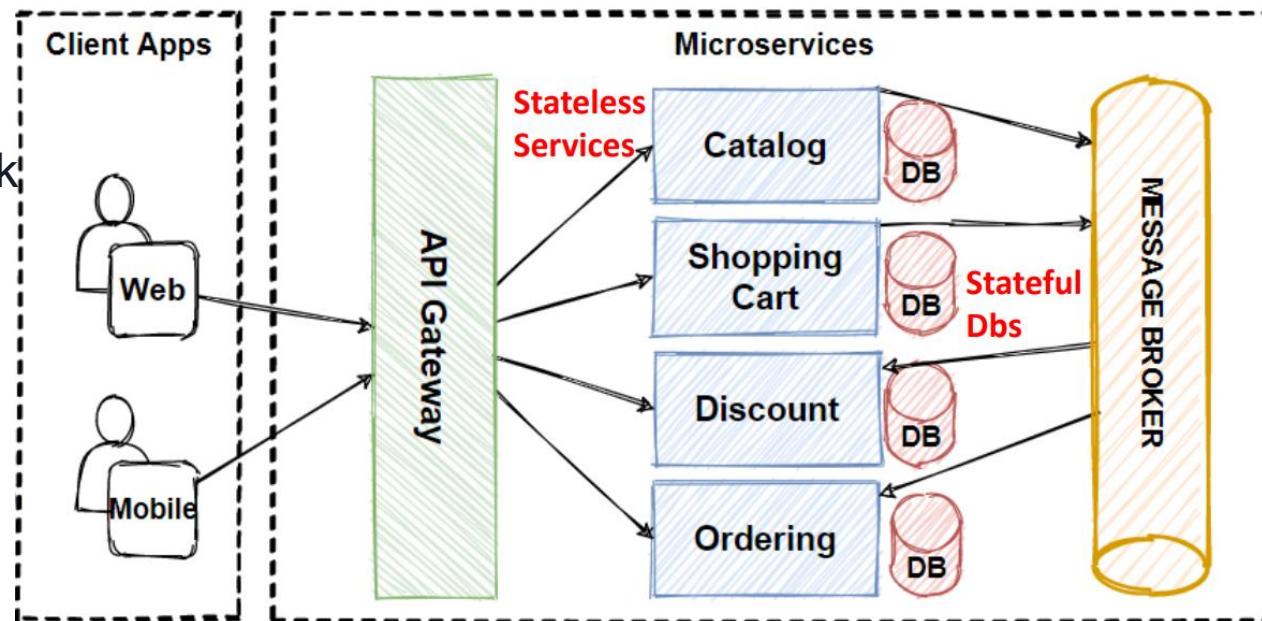
- How can we **manage** this **stateful data** when **scaling** the application ?
- **CAP Theorem** to manage stateful server horizontal scaling operation.
- Microservices are "**stateless**" services that can scale easily.
- Databases have "**state**" and need to **consider CAP Theorem** when horizontally scaling for stateful servers.



2 type of Consistency Level



- What is the **consistency level requirements** of our microservices databases ?
- **2 type** of "consistency level":
- **Strict Consistency**
When we save data, the data should affect and seen immediately for every client. Debit or withdraw on bank account.
- **Eventual Consistency**
When we write any data, it will take some time for clients reading the data. Youtube video counters.
- When we horizontally scaling with the stateful database services, should **split the data** which is calling **sharding**.
- **Database sharding** is also the way of **partitioning** and splitting database servers.
- How can we manage data in distributed systems ?



Microservices Data Management - Choosing Right Database

Polyglot Persistence, Database-per-Service Pattern, Shared Database Anti-pattern

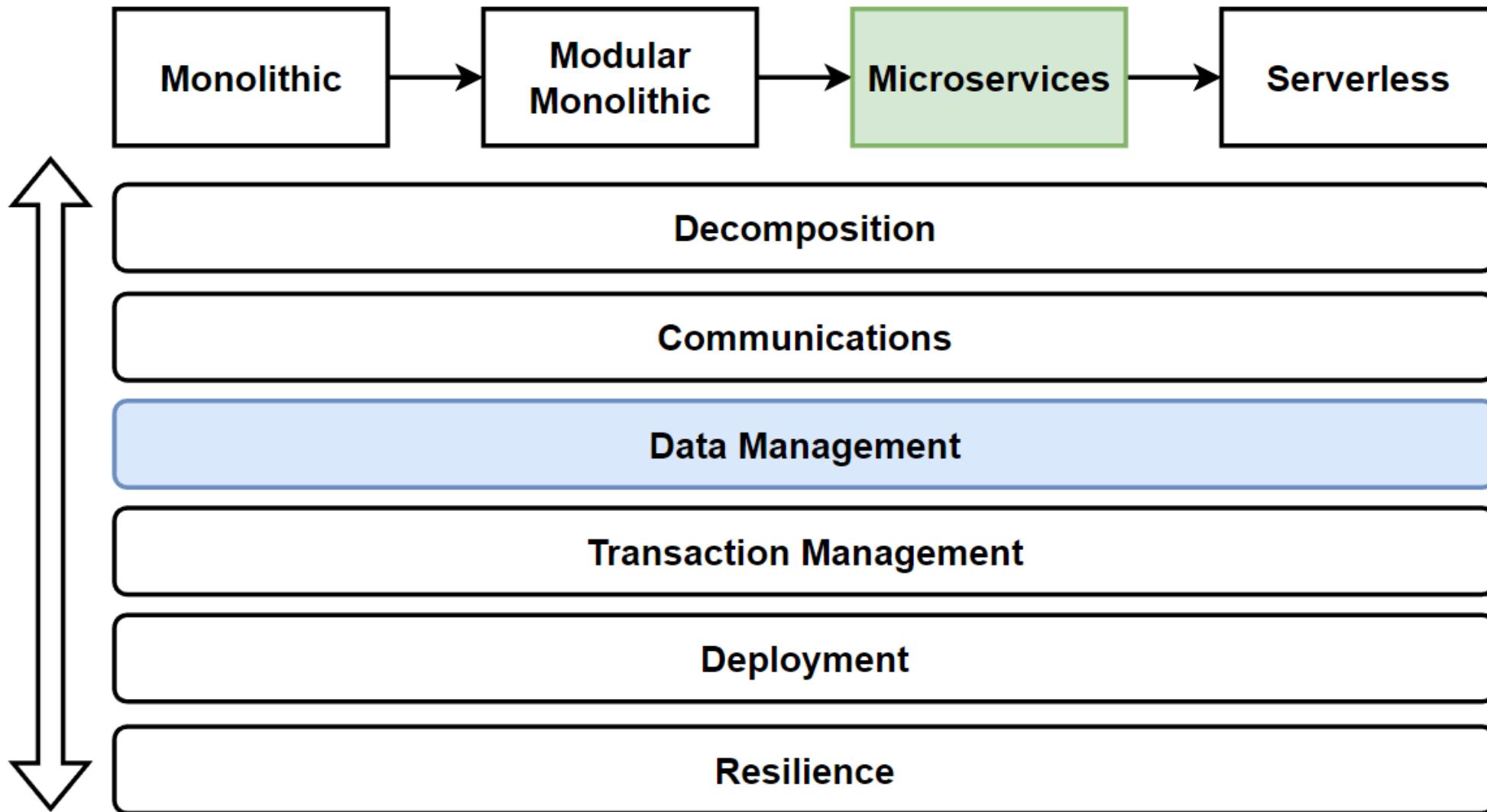
Relational and NoSQL Databases

CAP Theorem – Consistency, Availability, Partition Tolerance

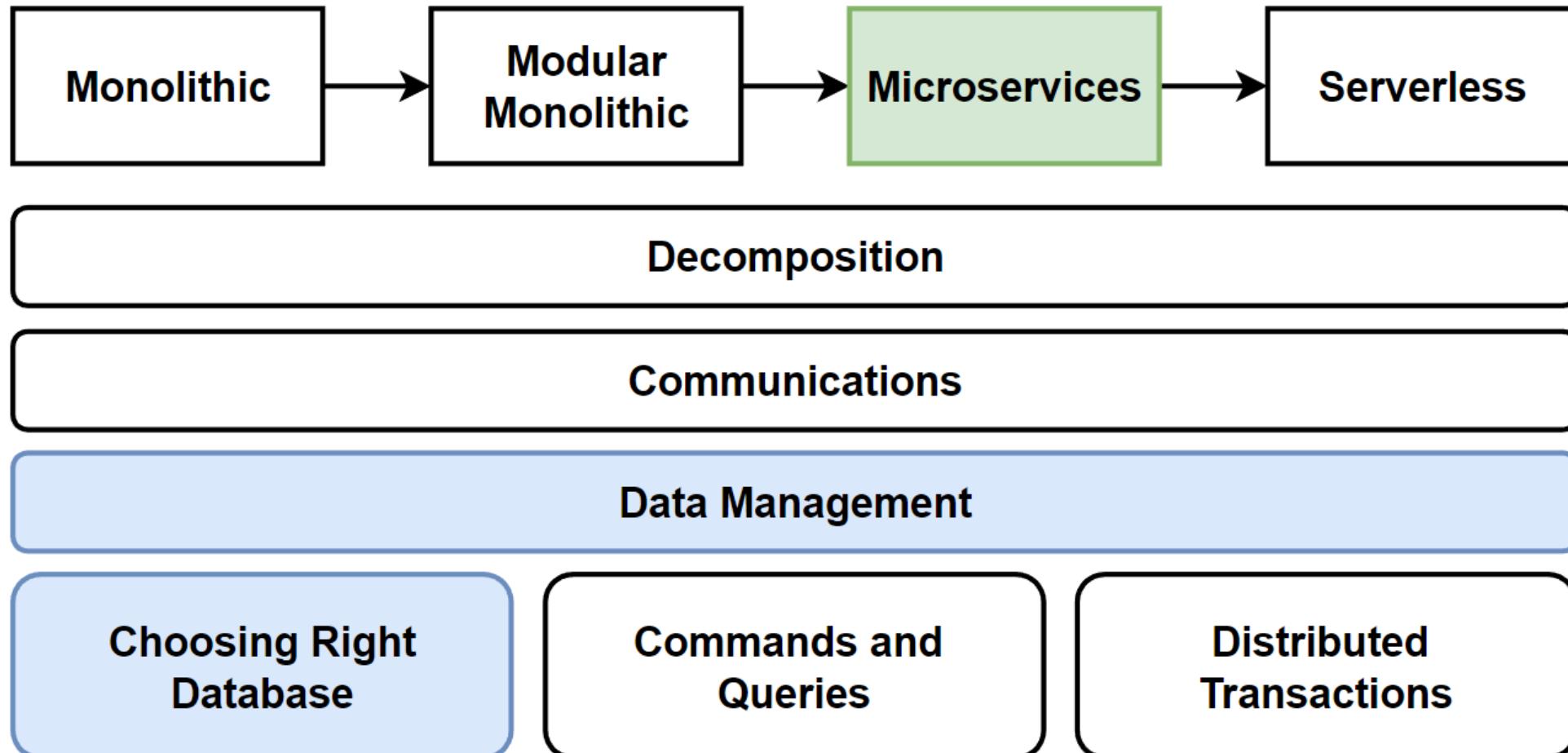
Data Partitioning: Horizontal, Vertical and Functional Data Partitioning

Database Sharding Pattern

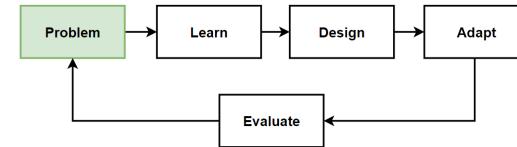
Architecture Design – Vertical Considerations



Microservices Data Management - Main Topics



Problem: Database Bottlenecks when Scaling

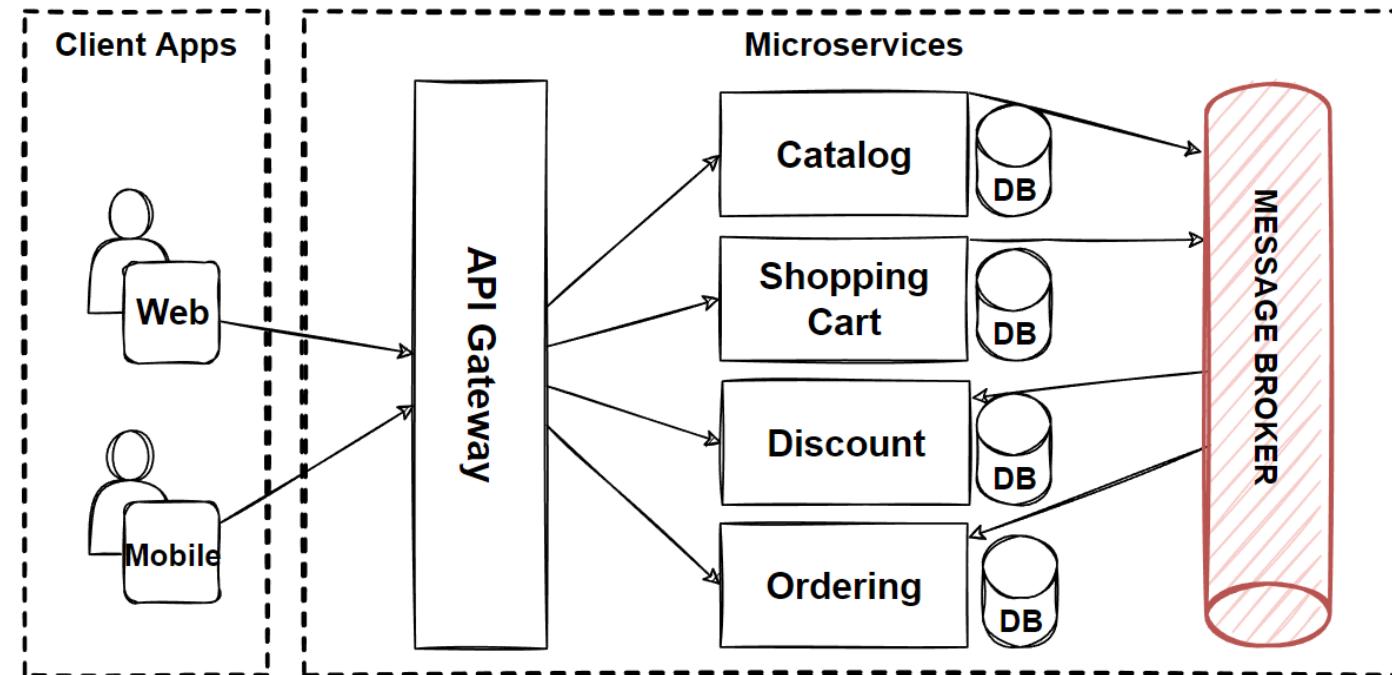


Problems

- Database are stateful service
- Scaling stateful services are not easy
- Vertical scaling has limits need to scale horizontally

Solutions

- Scale Stateful Application Horizontal Scaling
- Service and Data Partitioning along Business Boundaries - Shards/Pods
- Use NoSQL Database to gain easy partitioning features

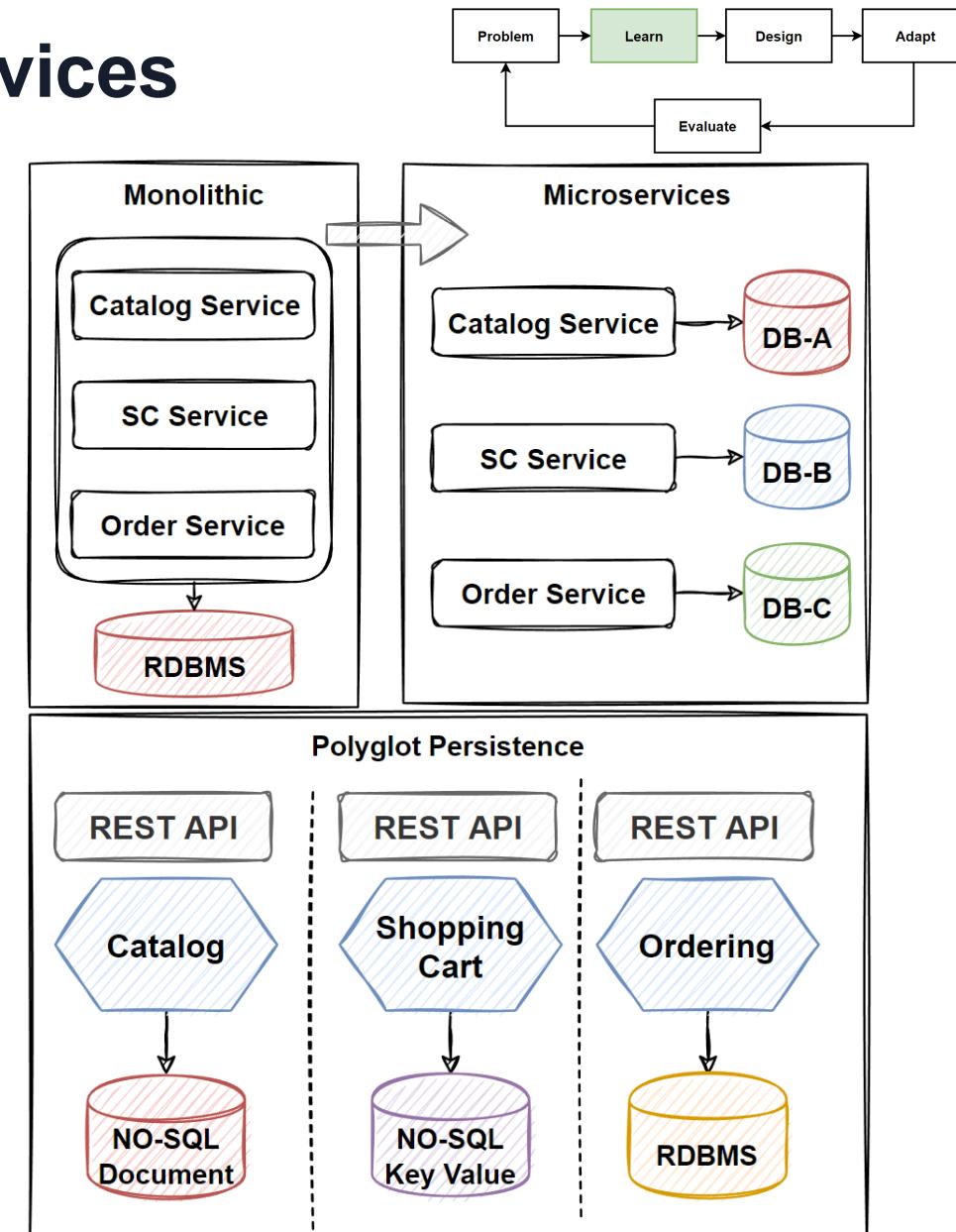


Question

- How to Choose a Database for Microservices ?

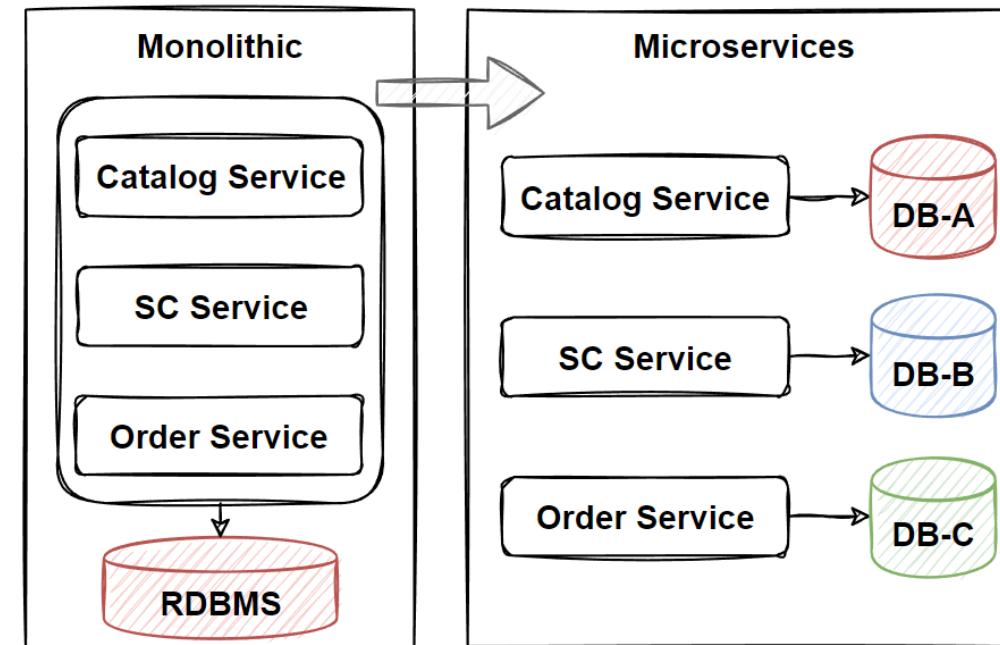
Polyglot Persistence Principle in Microservices

- Every microservice has its **own data**, the **data integrity** and **data consistency** should consider very **carefully**.
- Microservices should **not share database** with each other.
- Microservices **share data** over the application microservices with **using Rest APIs**. **Broke the unnecessary dependencies** between microservices.
- If there is an **update on 1 microservices database schema**, the update **shouldn't be directly affect** to other microservices.
- **Limit the scope of changes** on microservices when any database schema changes happened.
- We can **pick different databases** as per microservices which database can pick the best option: "**polyglot persistence**" principle.
- **Challenges:** Duplicated or partitioned data can make problems of data integrity and data consistency. Strict or Eventual consistency.



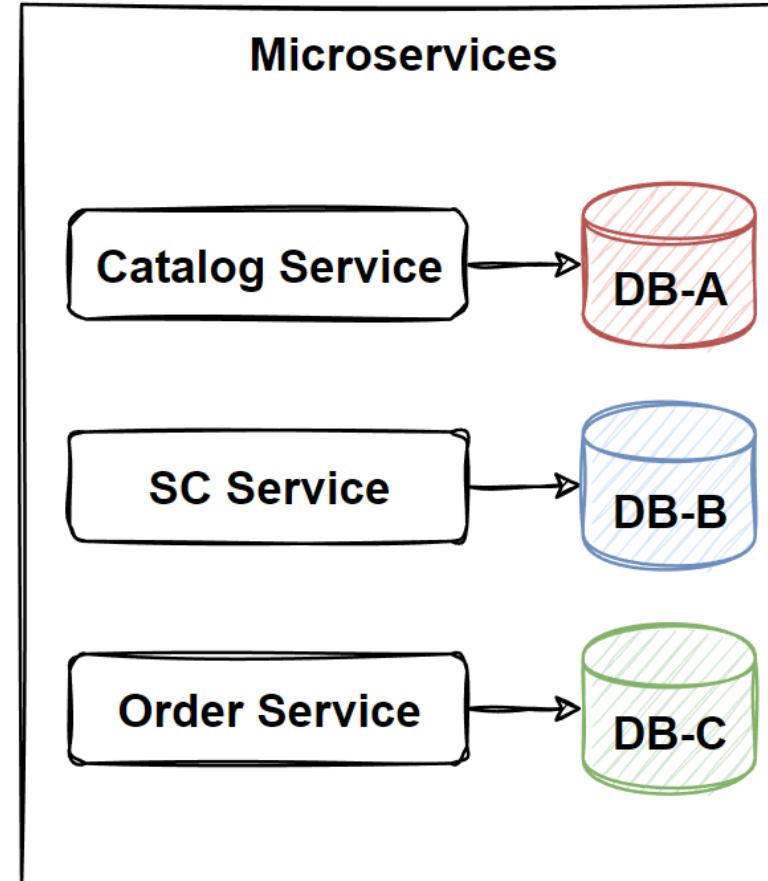
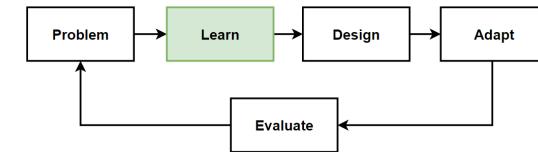
Polyglot Persistence Principle in Microservices-2

- In monolithic applications **can't be duplicate** and not be any data consistency problems, strict database transaction management.
- Microservices architecture, we have to **welcome duplicate data** and **eventual consistency**.
- How we can **welcome duplication** and **un-consistent data** ?
- **Accept eventual consistency** data in microservices where it is possible.
- **Define our consistency requirements** before design the system.
- If need to **strong consistency** and **ACID** transactions, we should follow traditional approaches.
- Use **event driven architecture**: microservices publish an event when any changes happened on data model.
- Subscriber microservices **consume and process event** afterwards in eventual consistency model.
- Polyglot Persistence gives us scale independently and **avoid single-point-of-failure** of bottleneck databases.



Microservices Database Management Patterns and Principles

- Managing a microservices **database** is **challenging job**, should have a strategy.
- ACID: atomicity, consistency, isolation and durability.
- Challenging to implement queries and transactions that visits to several microservices.
- Microservices Database Management Patterns and principles:
 - 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 - Overview

■ The Database-per-Service Pattern

In order to be a loose coupling of services, each microservice should have its own private database. Design database architecture for microservices.

■ The API Composition Pattern

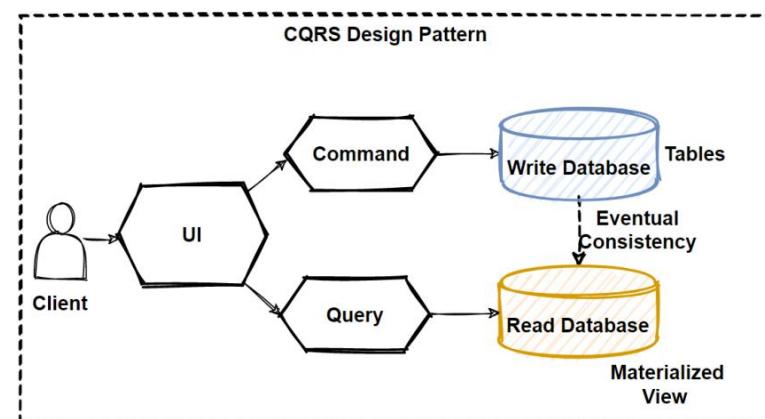
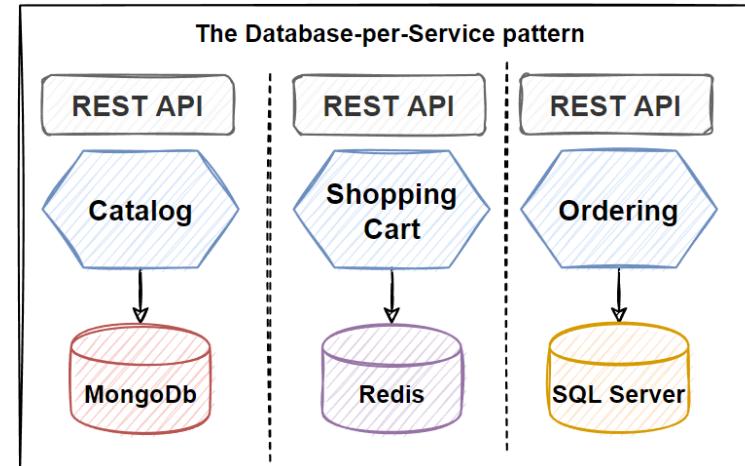
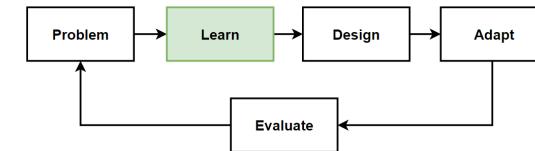
Retrieving data from several services also need a set of patterns and practices. When implements a query by invoking several microservices, we will follow the API Composition, Gateway Aggregation patterns for combining the results.

■ The CQRS Pattern

The command query responsibility segregation (CQRS) is provide to separate commands and queries database in order to better perform querying several microservices.

■ The Event Sourcing Pattern

The Event Sourcing pattern basically provide to accumulate events and aggregates them into sequence of events in databases. We can replay at certain point of events.



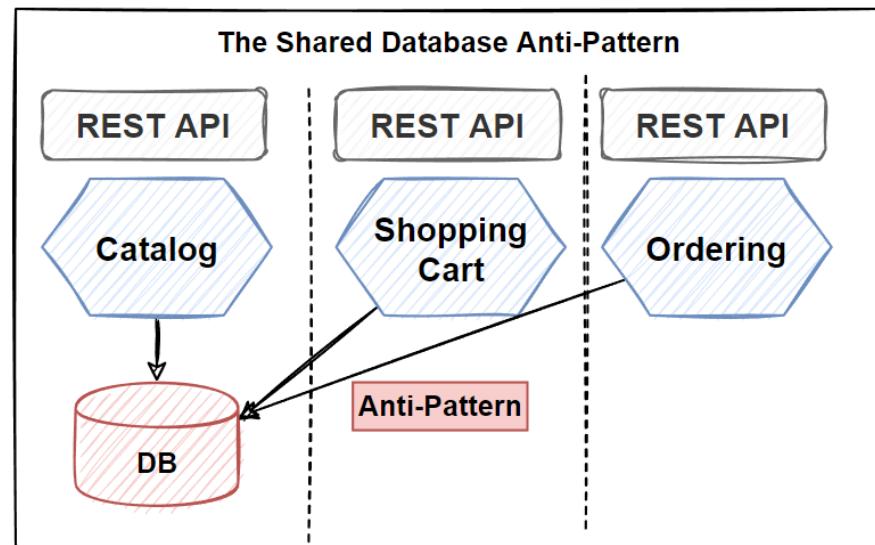
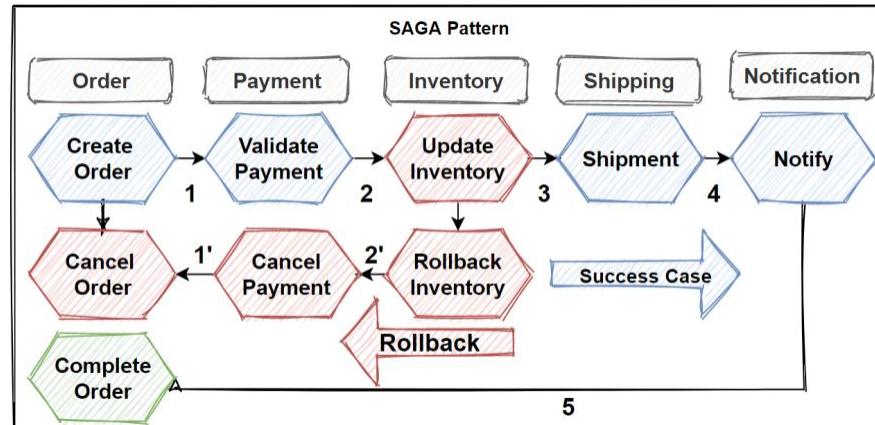
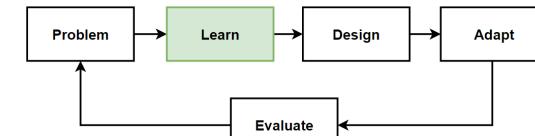
Microservices Database Management Patterns and Principles – Overview 2

■ The Saga Pattern

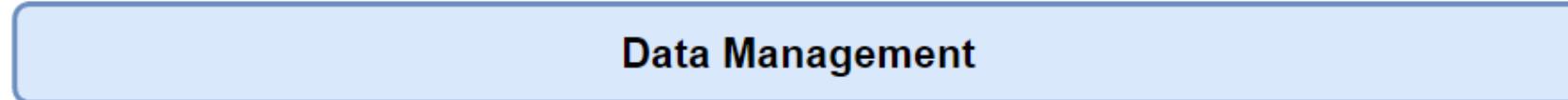
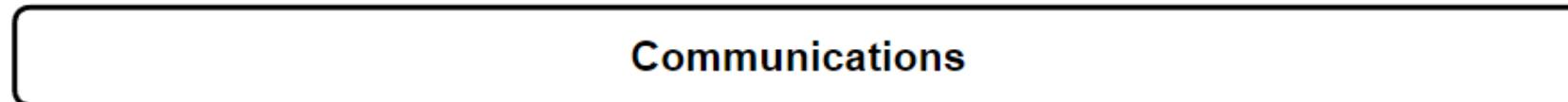
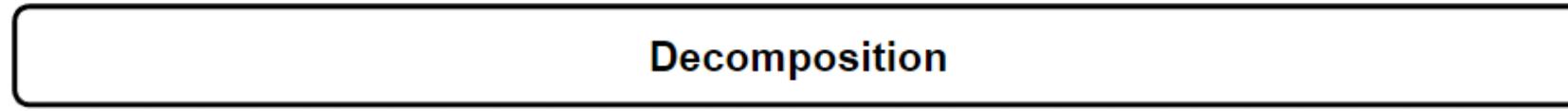
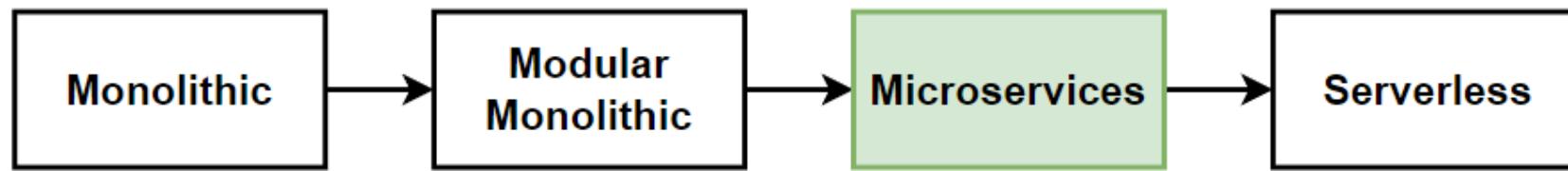
Transaction management is really hard when it comes to microservices architectures. We use Saga Pattern to implementing transactions between several microservices and maintaining data consistency.

■ The Shared Database Anti-Pattern

You can create a single shared database with each service accessing data using local ACID transactions. But it is against to microservices nature and will cause serious problems in the future of applications.



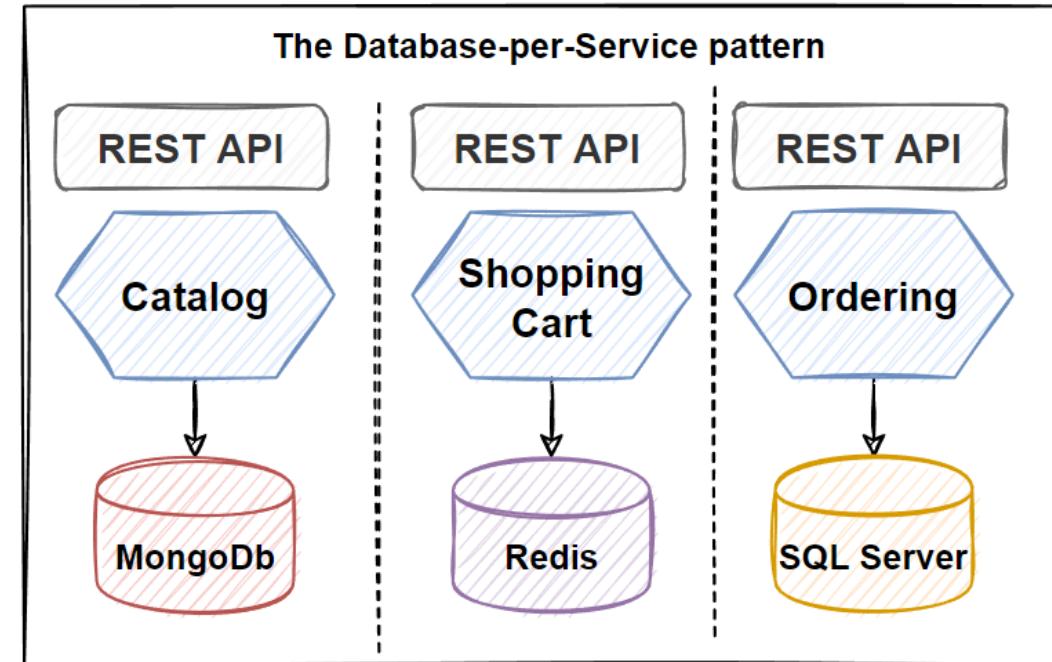
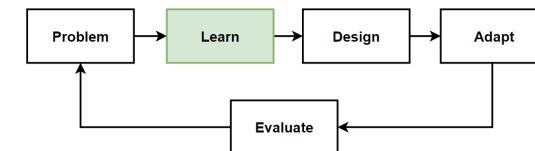
Microservices Data Management – Patterns and Principles



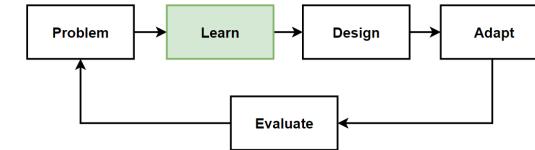
- Polyglot persistence
- The API Composition
- The Saga Pattern
- The Database-per-Service
- The CQRS pattern
- The Event Sourcing
- The Shared Database Anti- Pattern
- The Shared Database Anti- Pattern

The Database-per-Service Pattern

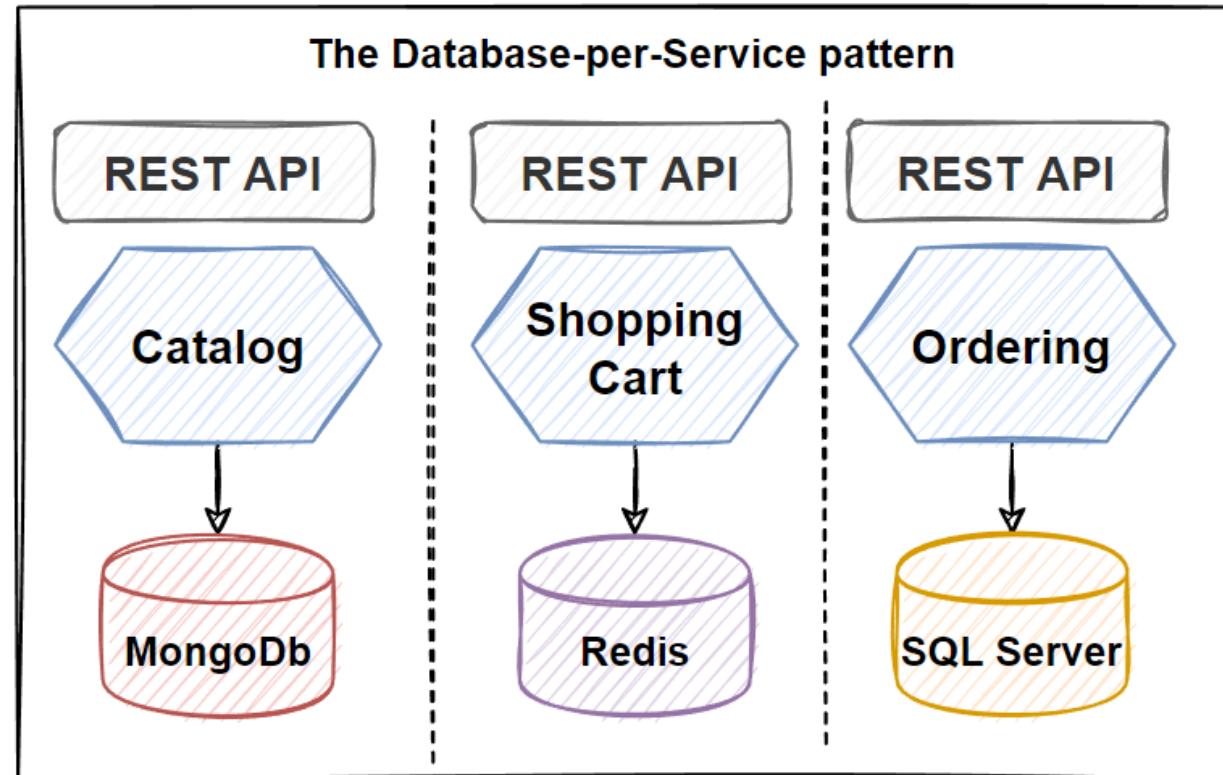
- Microservices should be **loosely coupled**, **scalable**, and **independent**. When shifting to the monolithic architecture to microservices architecture, should **decomposes databases**.
- **Distributed data model** with many **smaller databases** for particular microservice.
- **Data schema changes** can perform **without any impact** on other microservices, Changes to an individual database don't impact other services.
- There isn't a **single point of failure** in the application, the application is becomes more resilient.
- **Individual databases** are **easier to scale**, if 1 service peek the requests that only that microservice can scale independently.



The Database-per-Service Pattern-2

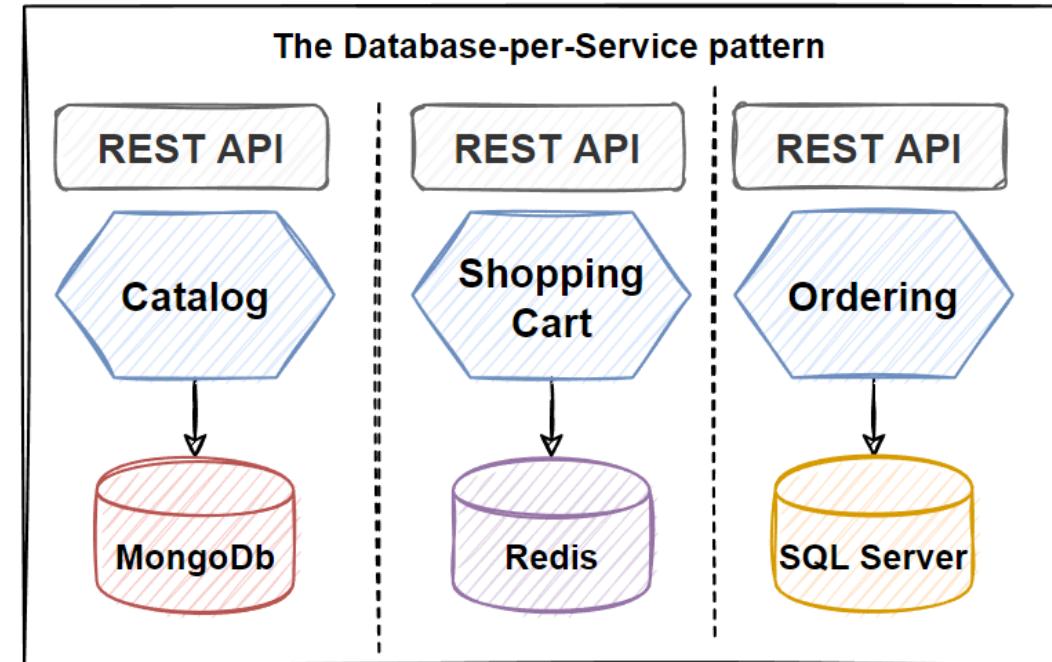
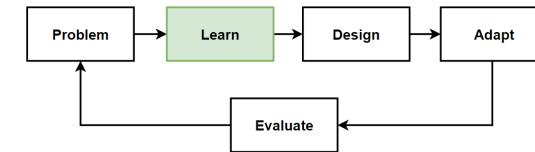


- **Separating databases** can give us the ability to **pick the best optimized database** for our microservices.
- Our choices include **relational**, **document**, **key-value**, and even **graph-based** data stores.
- Using the **most efficient database** depending on the service requirements and functionality.



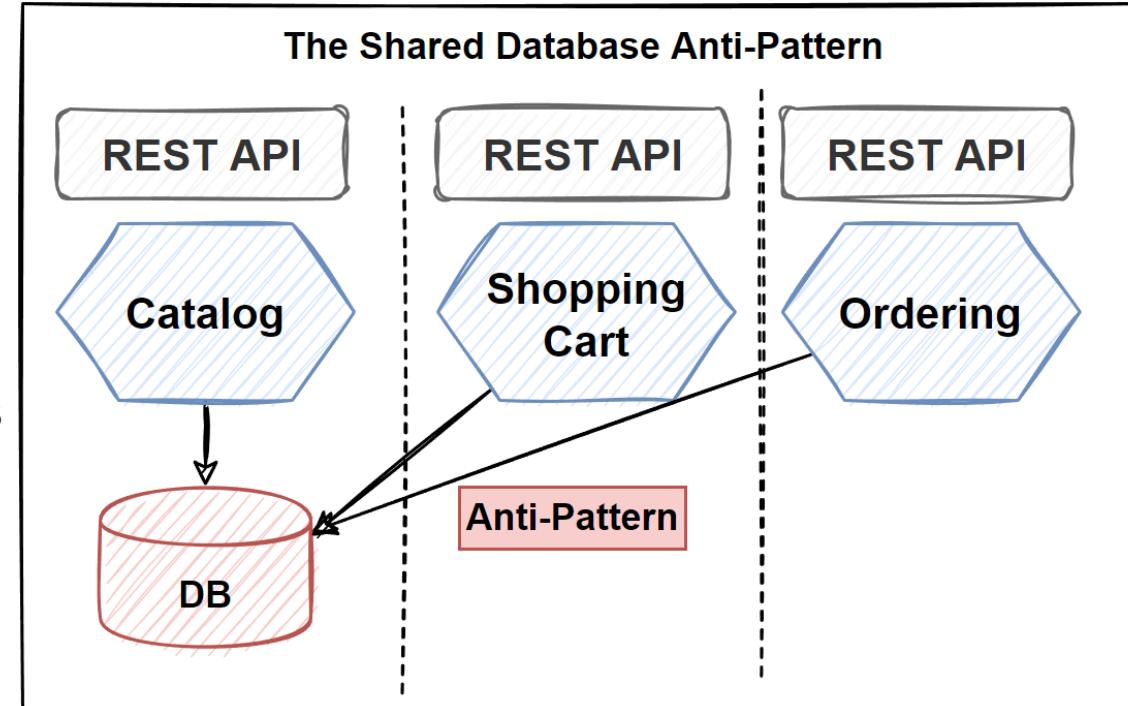
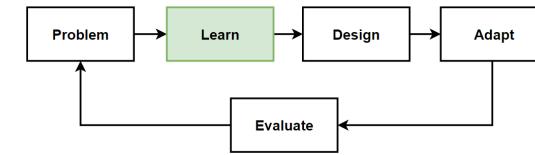
Drawbacks of Database-per-Service Pattern

- Services need a communication method to **exchange data, inter-service communication**.
- Each service **must provide a clear API**, that need to make resilience of these communications like applying **retry** and **circuit breaker patterns**.
- **Distributed transactions** across microservices can negatively **impact consistency and atomicity**.
- **Complex queries**, no simple way to execute join queries on multiple data stores.

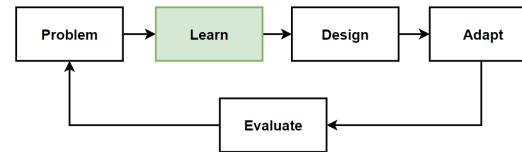


The Shared Database Anti-Pattern

- If we **don't follow Database-per-Service** pattern and use **Shared Database** for several microservices = **Anti-Pattern**
- It is **against to microservices nature** and will cause serious problems in the future of applications.
- When using a **shared database**, the microservices **lose** their **core properties**: scalability, resilience, and independence.
- You will face to **develop big a few monolithic applications** instead of microservices.
- Shared database can **block microservices** due to **single-point-of-failure**.
- If shared database seems to be the best option for the microservices project, **should re-think** that we really need the microservices, the monolith would be the better choice.



The Shared Database Anti-Pattern - Pros and Cons

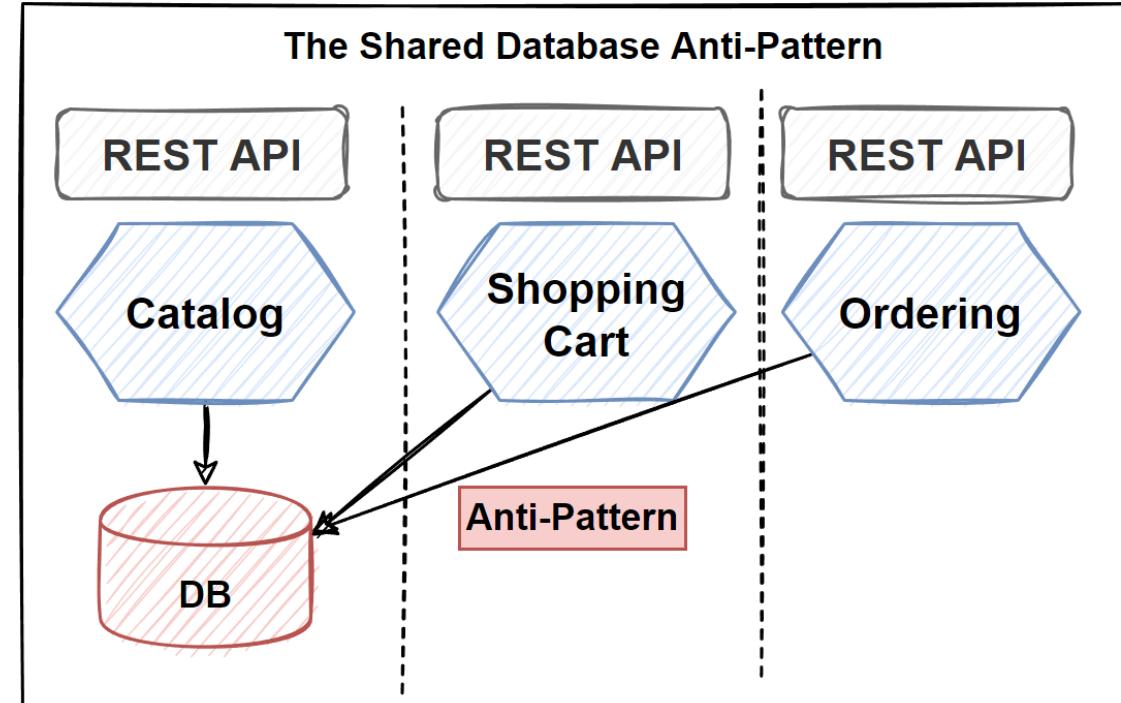


Benefits

- Transaction management, that we don't need to span the transactions over the microservices.
- Decrease duplicate data. Since data is fully constrained, we can easily execute complicated queries with joins.
- Able to follow ACID. Consistency of data and state easily manage when process fails.

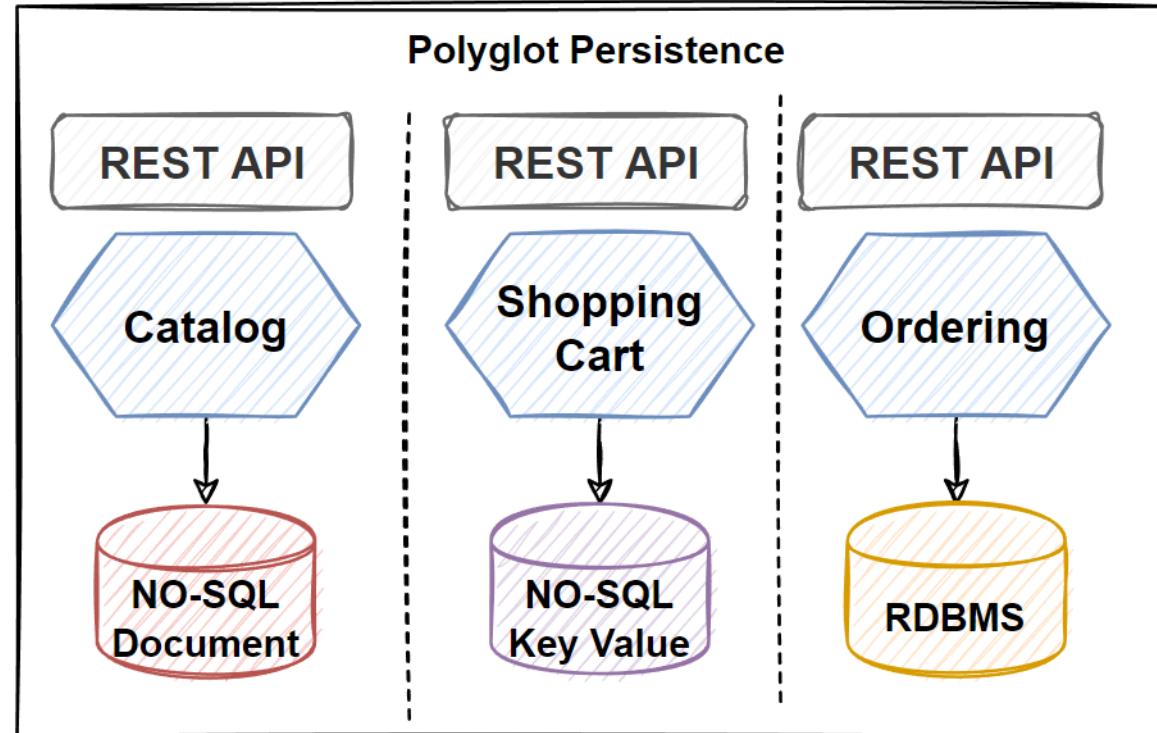
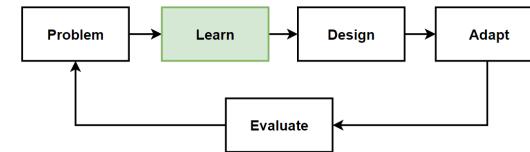
Drawbacks

- Microservices with shared databases can't easily scale.
- Shared database will become a single point of failure.
- Microservices won't be independent in terms of development and deployment.

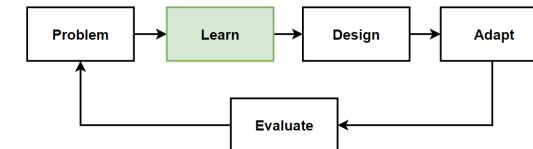


Polyglot Persistence

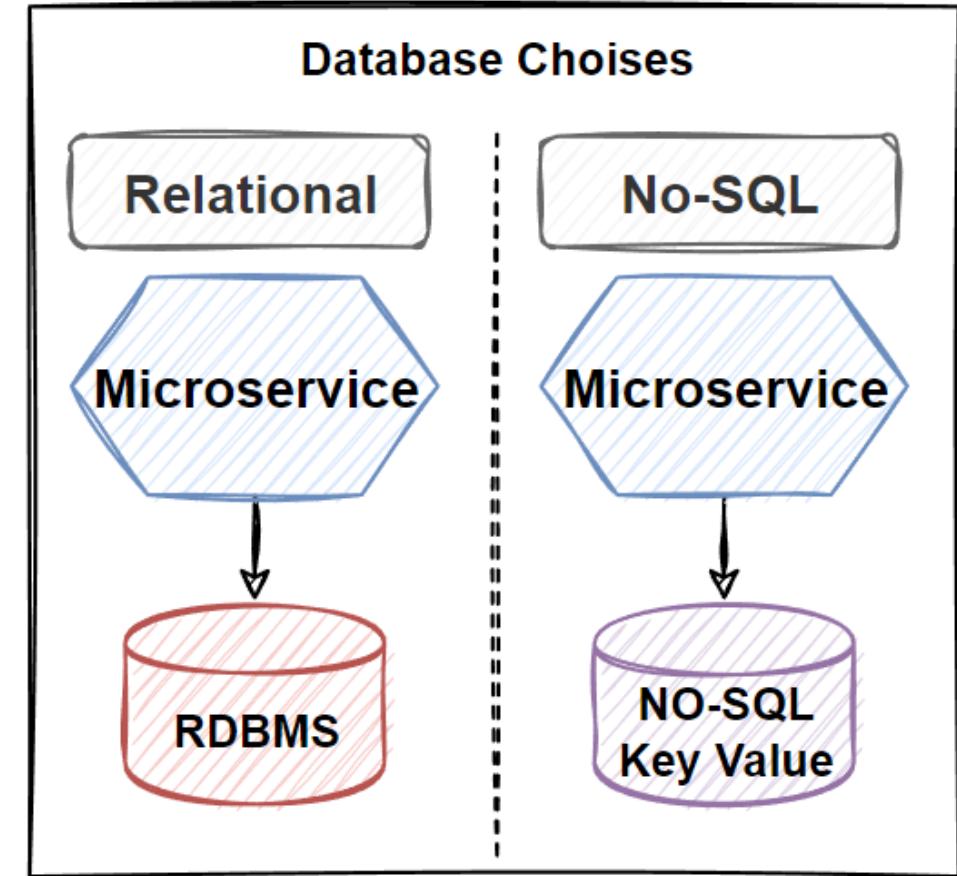
- Microservices enables using **different kinds of data storing** technologies.
- Each development team can **choose the persistence technology** that suits the needs of their service best.
- Martin Fowler - **Polyglot persistence** will **come with a cost** - but it will come because the benefits are worth it.
- When **relational databases** are used **inappropriately**, they give damaged on application development.
- Example of **looked up page elements by ID**, much better suited to a key-value No-SQL databases than relational databases.
- **How to Choose a Database for Microservices ?**



Database Choises - Relational and NoSQL Databases

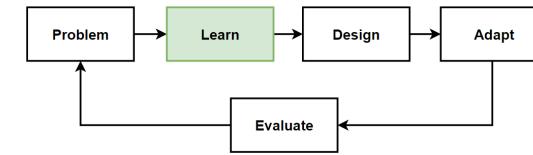


- **Relational** and **NoSQL** are **two types** of database systems commonly implemented in microservices architecture.
- New approach of databases which is called **NewSQL**.
- **Relational** and **NoSQL** are different in terms of the **storing data**, and **accessing data**.



Relational Databases - RDBMS

- **Relational databases** provides storing data into related data tables.
- Relational database tables have a **fixed schema**, use SQL to manage data and support transactions with **ACID principles**.
- A table uses **columns** and **rows** for storing the actual data. Each table will have a column that must have **unique values**—known as the **primary key**.
- When one **table's primary key** is used in **another table**, this column in the second table is known as the **foreign key**.
- On microservices, we should **not consider** using **single big relational database**.
- The main advantages of Relational database is **ACID compliance**. If one change fails, the **whole transaction will fail**.
- Polyglot persistence in microservices, still **relational databases** is **good option** some certain type of problems.
- **Example of relational databases** are Oracle, MS SQL Server, MySQL, PostgreSQL.



Relational

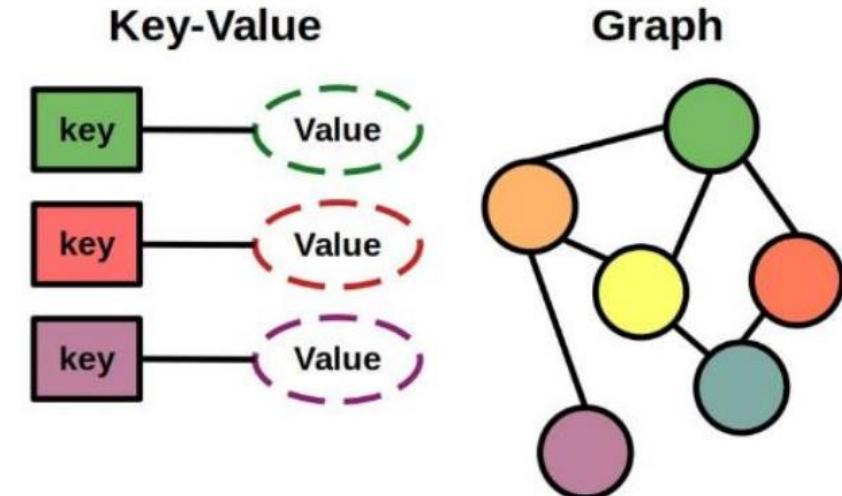
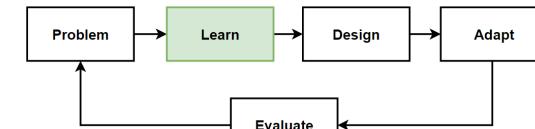
Yellow	Purple	Green

Green	Orange

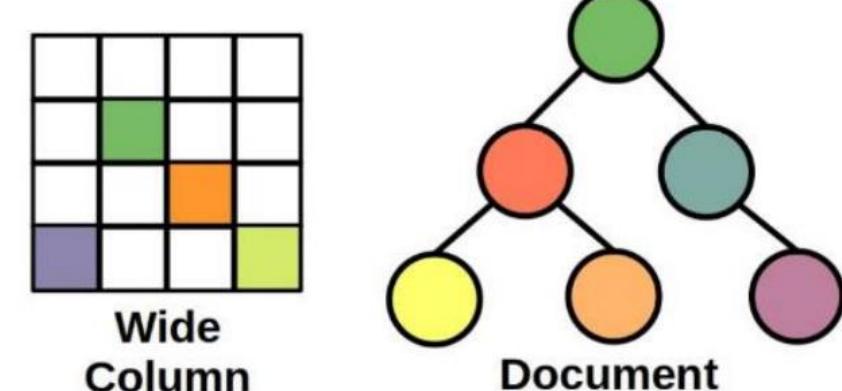
SQL

No-SQL Databases (Non-Relational Databases)

- No-SQL databases has different types of stored data and data models: **Document**, **Key-value**, **Graph-based**, **Column-based** databases.
- Ease-of-use, scalability, resilience, and availability characteristics.
- NoSQL databases stores **unstructured data**, and this gives huge performance advantage.
- NoSQL stored unstructured data in key-value pairs or JSON documents.
- No-SQL databases don't provide ACID guarantees.
- Drawback is transaction management.

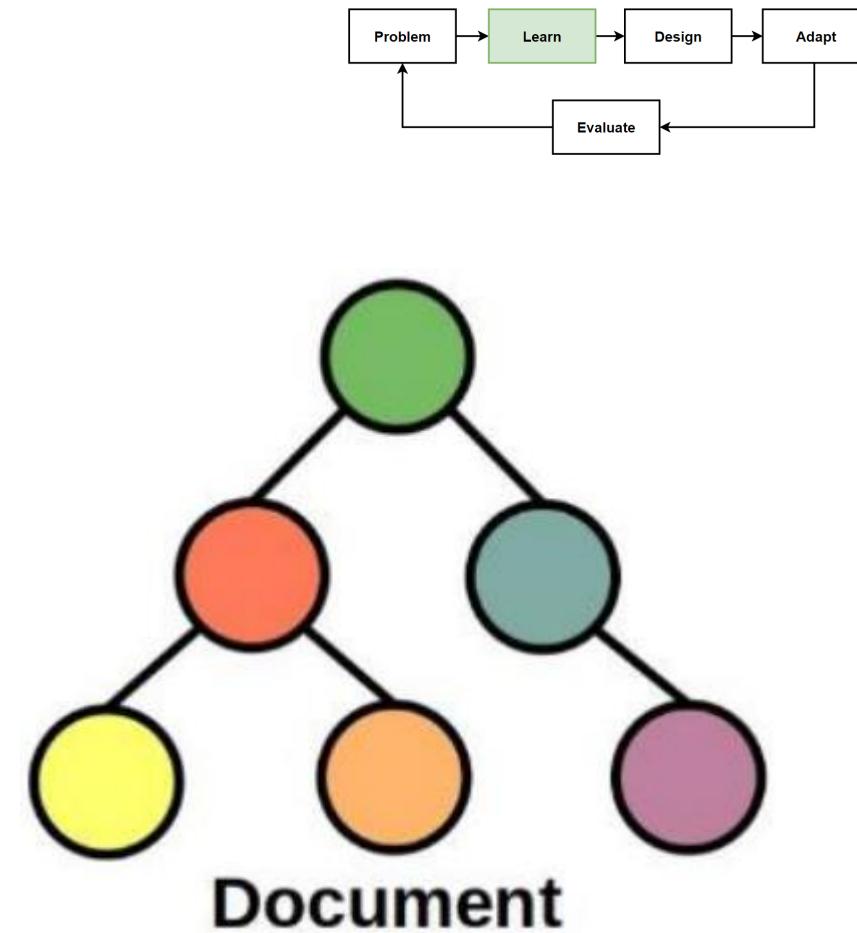


NoSQL



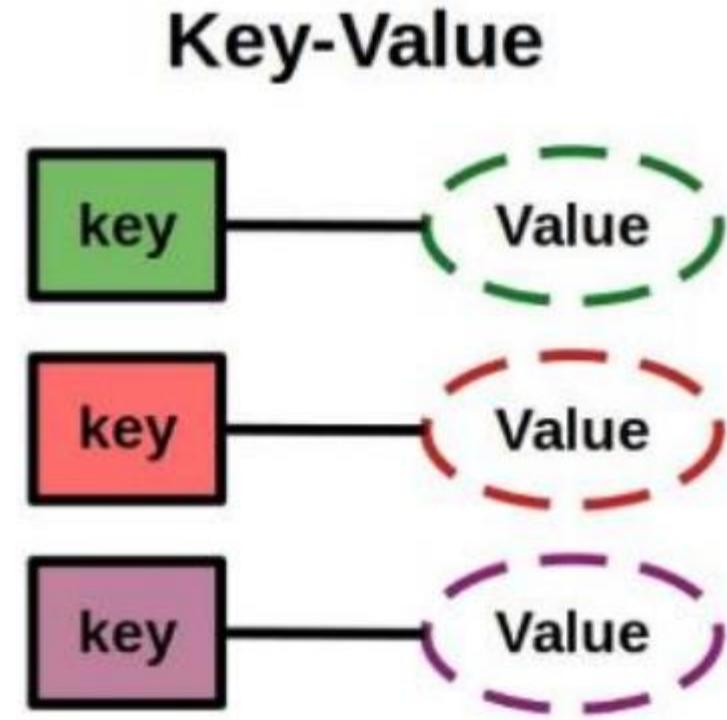
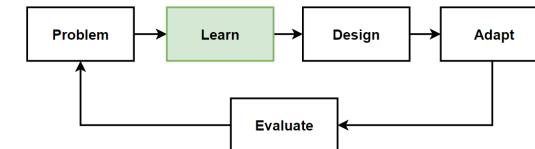
No-SQL Document Databases

- **Document databases** stores and query data in JSON-based documents.
- **Data and metadata are stored hierarchically.**
- **Objects are mapping to the application code.**
- **Don't have to run JOINs** or decompose data across tables.
- **Scalability**, document databases can **distributed very well**.
- Best choice for **content management** and **storing catalogs**.
- I.e. **products** data can store in **document database** for **e-commerce applications**.
- **Example Document Databases:** MongoDB and Cloudant.



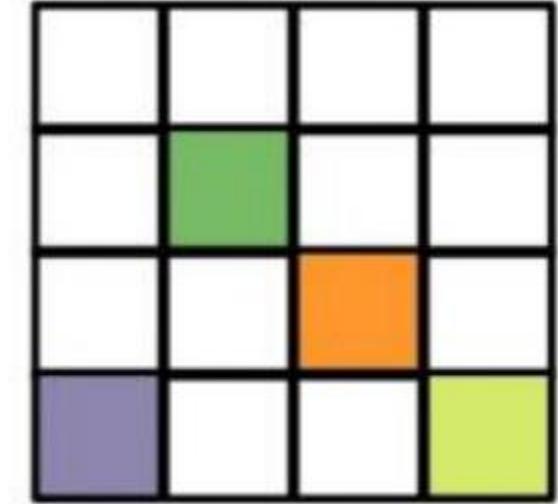
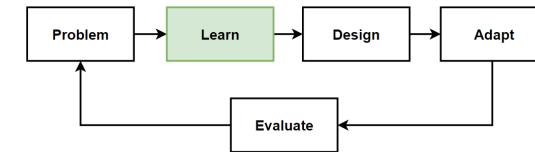
No-SQL Key-Value Databases

- Data is stored as a collection of **key-value pairs** in Key-value **NoSQL database**.
- Data is represented as a **group of key-value** in the database.
- **Best choice** for **session-oriented** applications.
- I.e. storing **customer basket data** into **key-value database**.
- **Example Key-Value Databases:** Redis, Amazon DynamoDB, Azure CosmosDB, Oracle NoSQL Database.



No-SQL Column-Based Databases

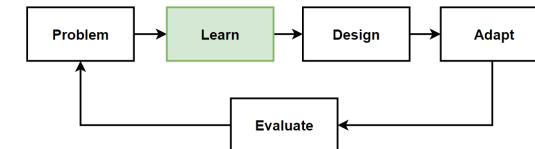
- **Column-based databases** also known Wide-Column Databases.
- **Data is stored in columns**, by this way, it can access necessary data **more faster** than if we compare to storing data in rows.
- If you **select mostly same columns** in your databases, its good to use this databases.
- It **doesn't scanning the unnecessary information** in a whole row.
- **Column-based databases can scale by columns independently.**
- **Columns could be different database servers.**
- I.e. building a **Data warehouse**, **Big Data processing**.
- **Apache Cassandra, Apache HBase or Amazon DynamoDB, Azure CosmosDB.**



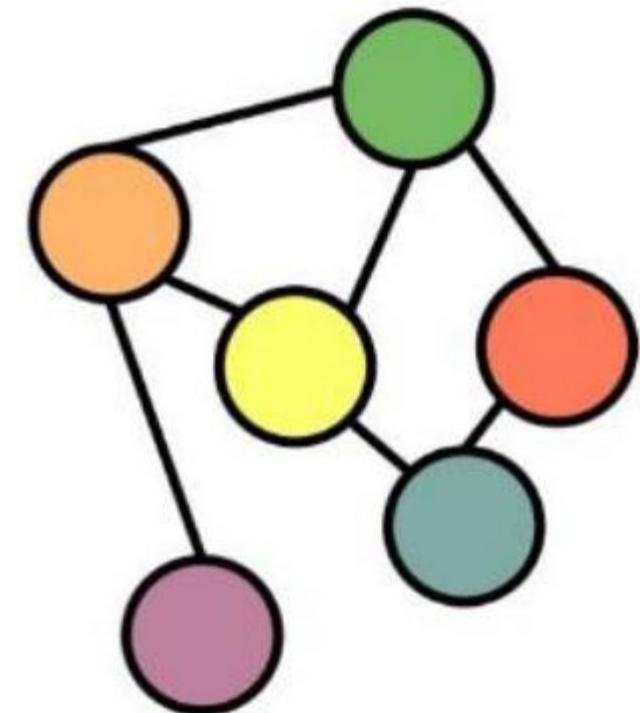
**Wide
Column**

No-SQL Graph-Based Databases

- **Graph-based databases** stores data in a **graph structure** into **node**, **edge**, and **data properties**.
- **Data entities** are **connected in nodes**.
- The main benefit of a **graph-based databases** is to **store and navigate graph relationships**.
- I.e. **fraud detection**, **social networks**, and **recommendation engines**.
- **Example of Graph-based databases** are OrientDB, Neo4j, and Amazon Neptune.



Graph



When to Use Relational Databases ?

- **ACID compliance, Data Consistency**

Relational database is ACID compliant. ACID - Atomicity, Consistency, Isolation, and Durability (ACID) guarantees the reliability of database transactions.

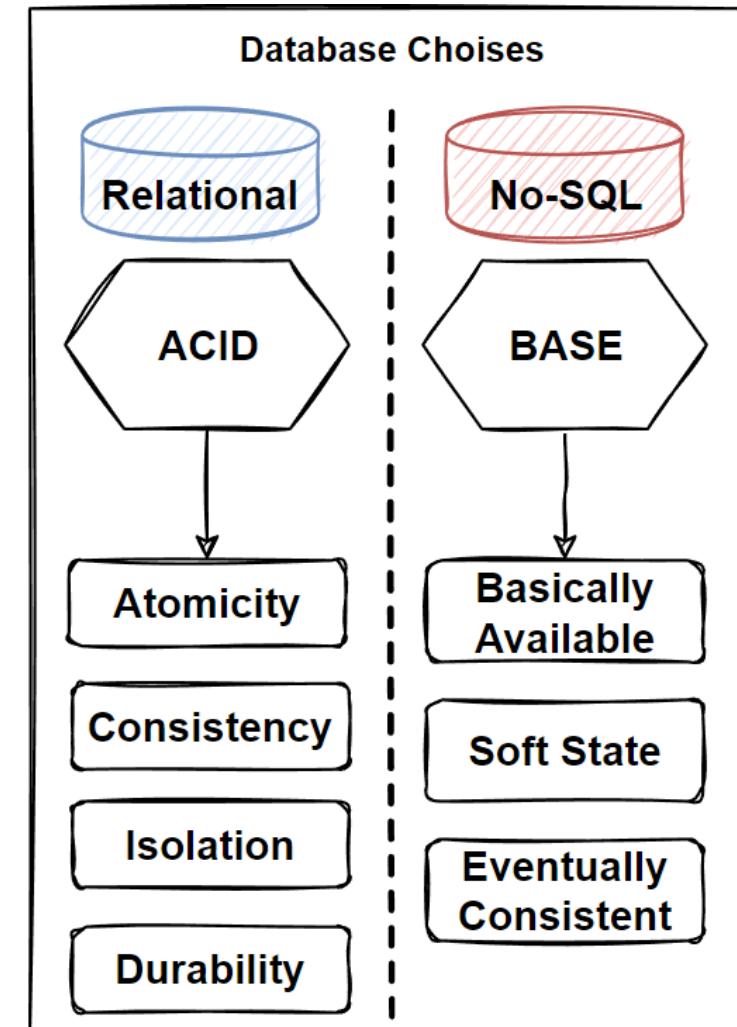
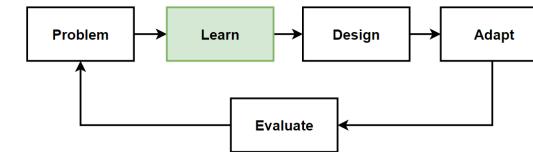
- If one change fails on database, database should remain in the previous state that was before the transaction.

- Data Consistency is fully provided, if you need strong consistency, good to choose Relational Databases, supports complex transactions.

- **Predictable Data, Low Workload Volume**

If application data is predictable, as per structure, size, and frequency of access, within thousands of transactions per second, relational databases are still the best choice.

- Normalization also reduces the size of the data on disk by limiting duplicate data and anomalies.



When to Use Relational Databases ? - 2

- **Read Requirements, Complex Join Queries**

Relational Database has a fixed schema. When relationships between tables are important and data is highly structured and requires referential integrity.

- Can work with complex queries, table joins and reports on normalized data models.

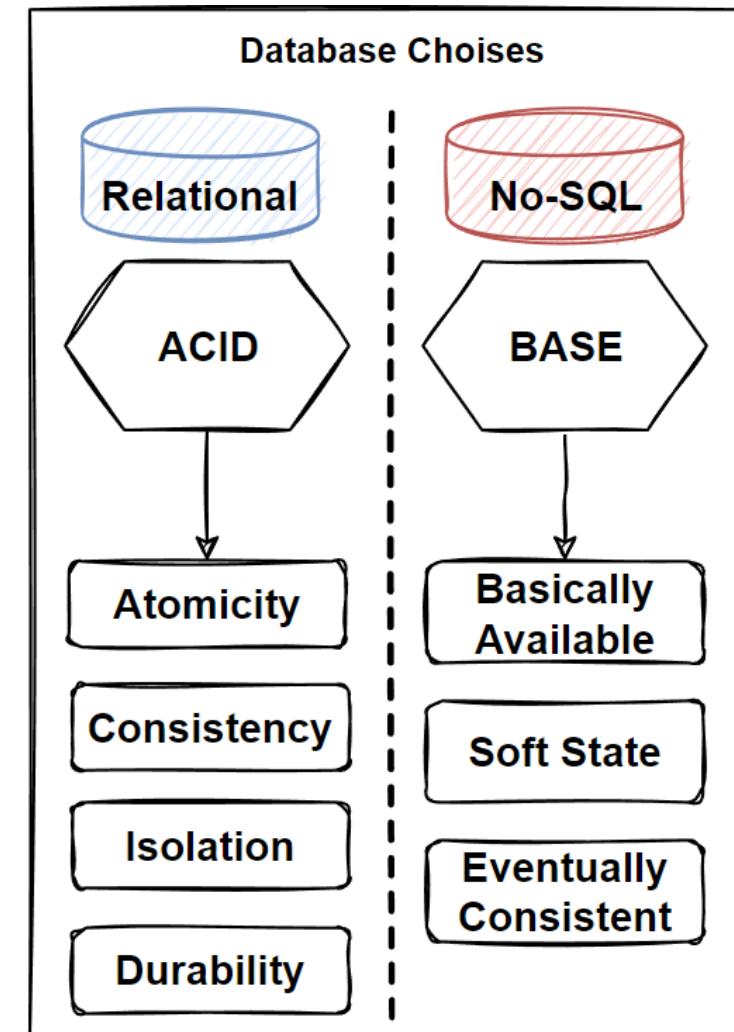
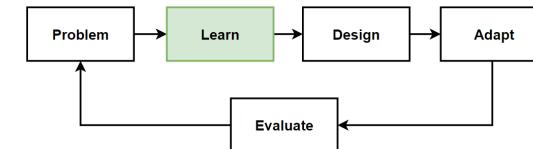
- Relational Database supports a powerful SQL query language.

- **Deployments, Centralized Structure**

Relational Databases will be deployed to large and one or few locations. Relational database has centralized structure.

- Relational databases have a single point of failure with failover.

- Relation database is deployed in vertical fashion.



When to Use No-SQL Databases ?

- **Flexible Schema, Dynamic Data**

NoSQL Database has no fixed schema. Allows to add or remove attributes into their model with dynamically. When your data is dynamic and frequently changes.

- Use case of implement an IoT platform that stores data from different kinds of sensors with frequently changed the attributes of your data.

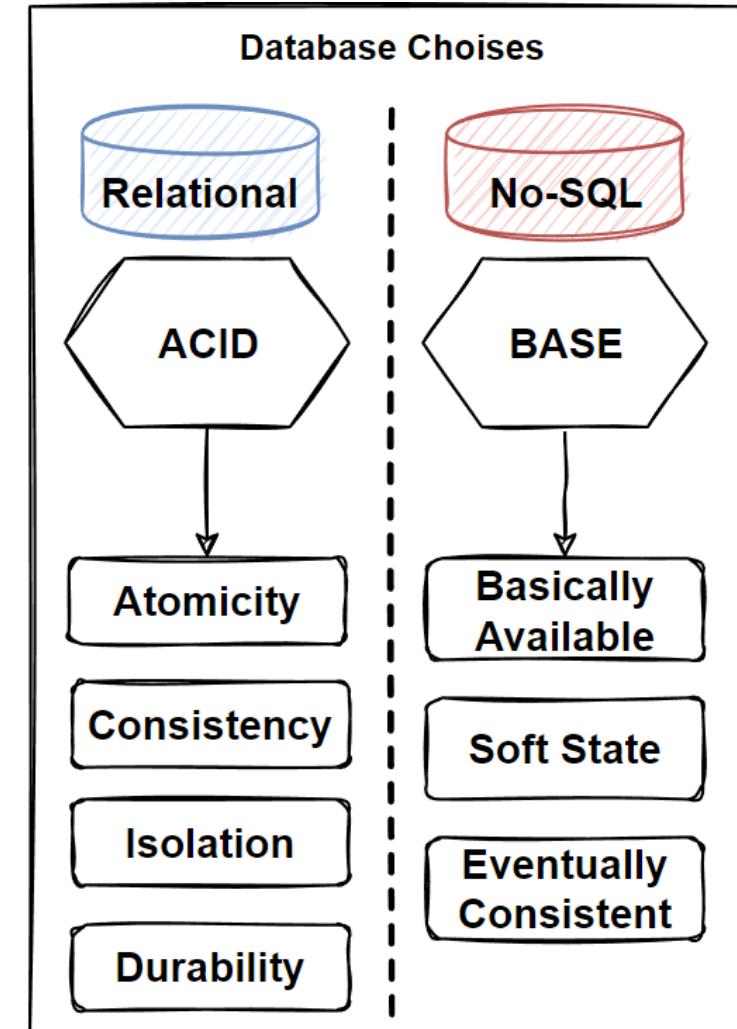
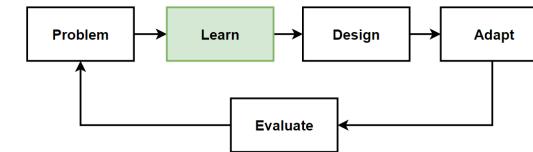
- **Un-predictable Data, High Workload Volume**

When you have high volume workloads and needs to horizontal scale with low latency. NoSQL databases have been designed for the cloud that naturally good for horizontal scaling.

- NoSQL Databases prioritize partition tolerance that designed for handling large amount of data or data coming in high velocity.

- **Frequently Change Data and Read Requirements**

When data is dynamic and frequently changes and Relationships are denormalized data models and Data retrieve operations are simple and performs without table joins.



When to Use No-SQL Databases ? - 2

- **Data Consistency, BASE Model - Basically Available, Soft State, Eventually Consistent**

NoSQL Database is only eventually consistent and don't support transactions, focus on high volume data and horizontal scaling.

- **Write Performance Requirements**

NoSQL Database compromise consistency to achieve fast write performance, offers fast write operations with Eventual consistency.

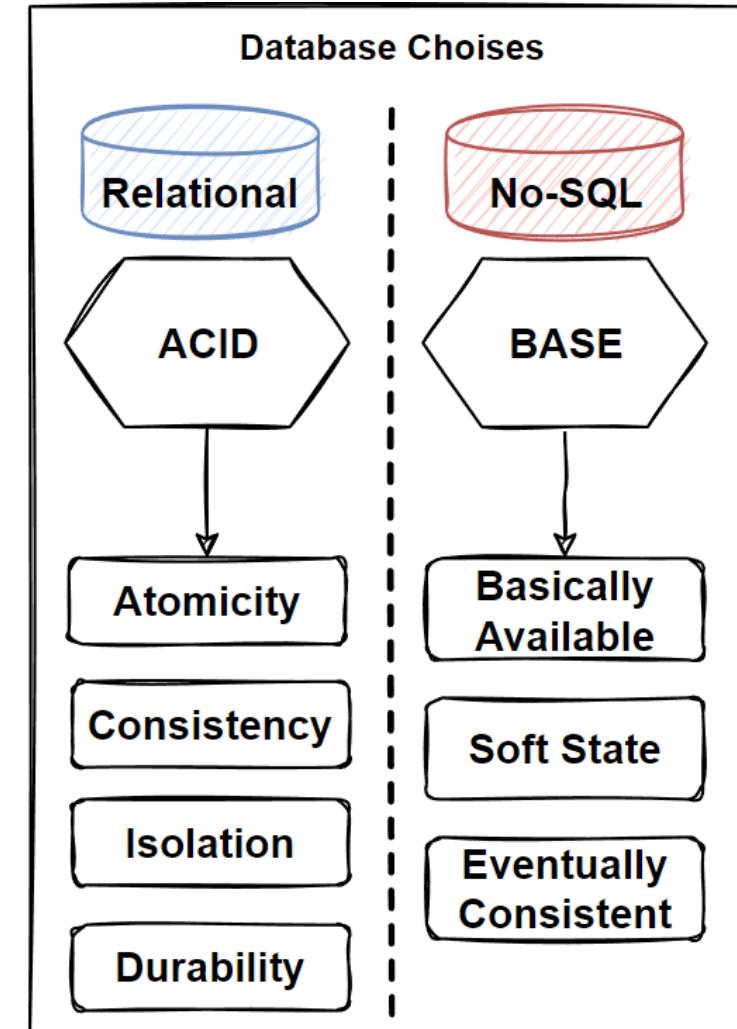
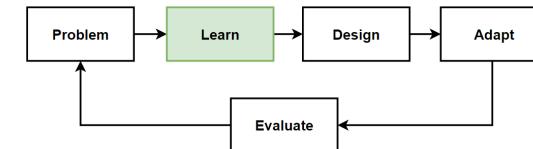
- **Not Good for Complex Join Queries**

NoSQL Databases perform best when data is stored in the same format not require relation and join operations.

- **Deployments, De-centralized Structure**

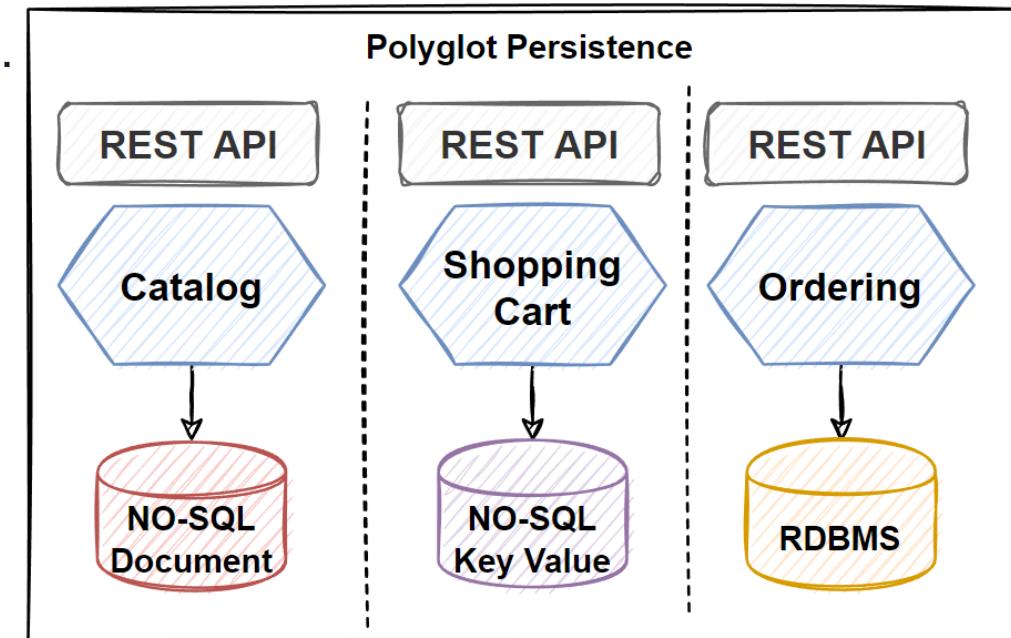
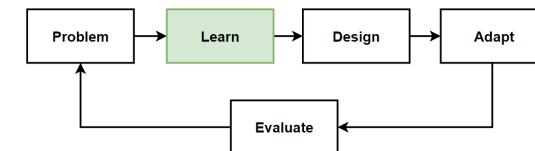
NoSQL scales horizontally so data is replicated across different geographical zones and provides better control over consistency, availability, and performance.

- NoSQL databases have no single point of failure, has decentralized structure, gives both read and write scalability, deployed horizontally.



Best Practices When Choosing Data Store

- Use Right Tool for Right Job, Use the best data store for your data.
- **Don't use Relational Database Everywhere**
If using Relational Database for all microservices, would probably going wrong way, consider other data stores to data requirements.
- **Don't use Single Data Store Technology, Differentiate**
Choose Alternatives to relational databases; Key/value stores, Document, Search engine, Time series, Column family, Graph databases.
- **Focus The Data Type That Need to Store**
Consider the type of data that you have.
 - Store JSON documents in No-SQL Document database.
 - Put transactional data into a Relational SQL database.
 - Use a time series data base for telemetry databases.
 - Choose Blob Data Storage for blob datas.
 - Put application logs into Elastic Search Databases.



Best Practices When Choosing Data Store - 2

- **Trade-offs between Availability and Consistency**

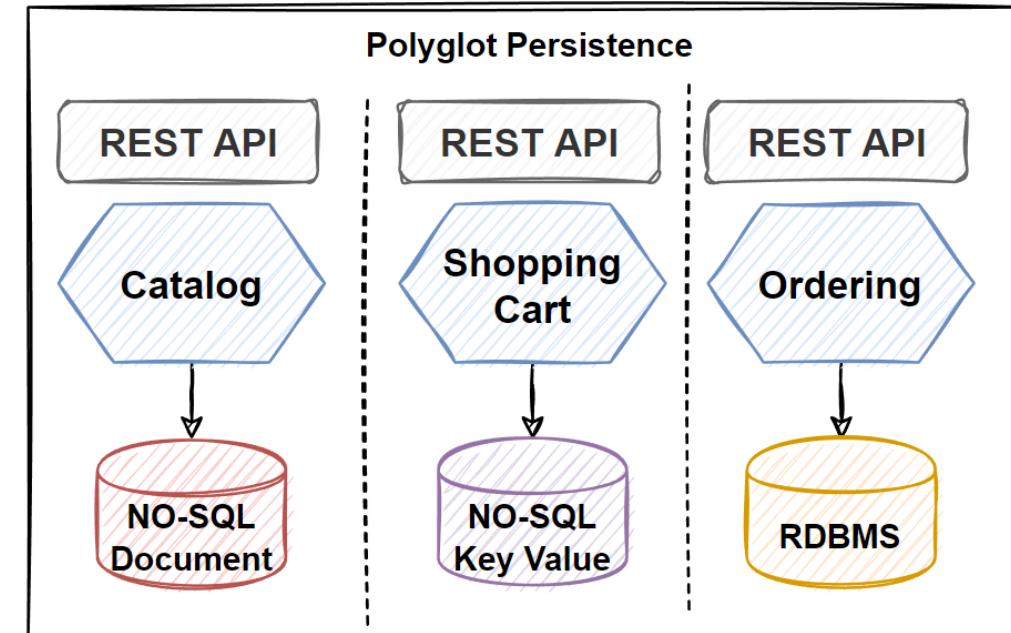
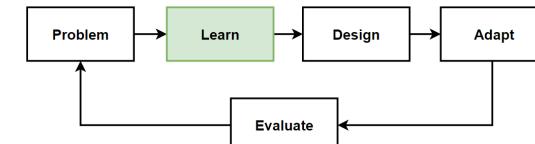
Understand the Trade-offs between Availability and Consistency.
Should prefer High Availability over strong consistency as soon as possible to scale horizontally. (the CAP theorem)

- **Transactional Boundaries Between Microservices**

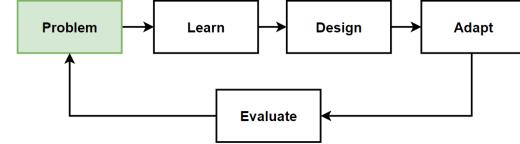
Consider business Transactional Boundaries Between Microservices, that data need to consistent across those microservices. Prepare for compensating transactions in case of fail.

- **Competence of Development Teams**

Consider your team competences about database technologies.
Skill set of your development team should cover optimize queries and tune for performance improvements.



Problem: Database Bottlenecks when Scaling, Different Data Requirements For Microservices

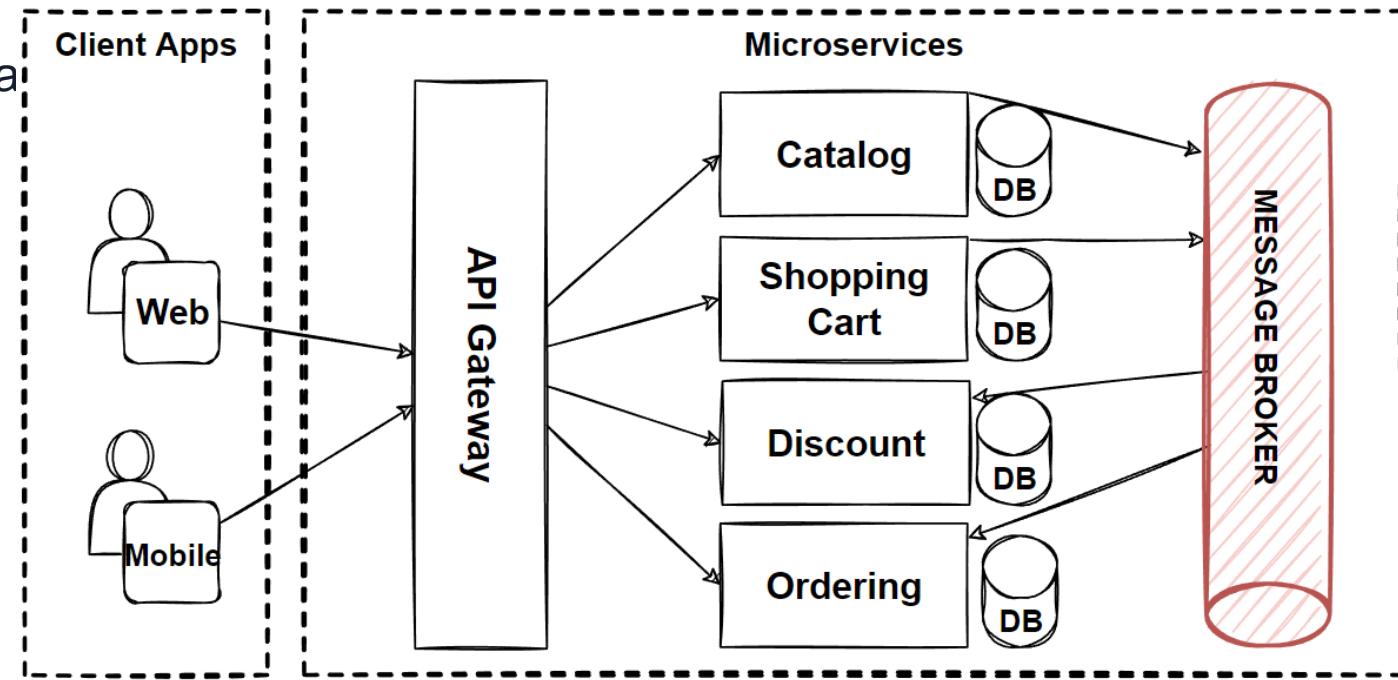


Problems

- Database are stateful service
- Scaling stateful services are not easy
- Vertical scaling has limits need to scale Horizontal
- **Different Data Requirements For Microservices**

Solutions

- Scale Stateful Application Horizontal Scaling
- Service and Data Partitioning along Business Boundaries - Shards/Pods
- Use NoSQL Database to gain partitioning
- **Identify Database Requirements following best practices**

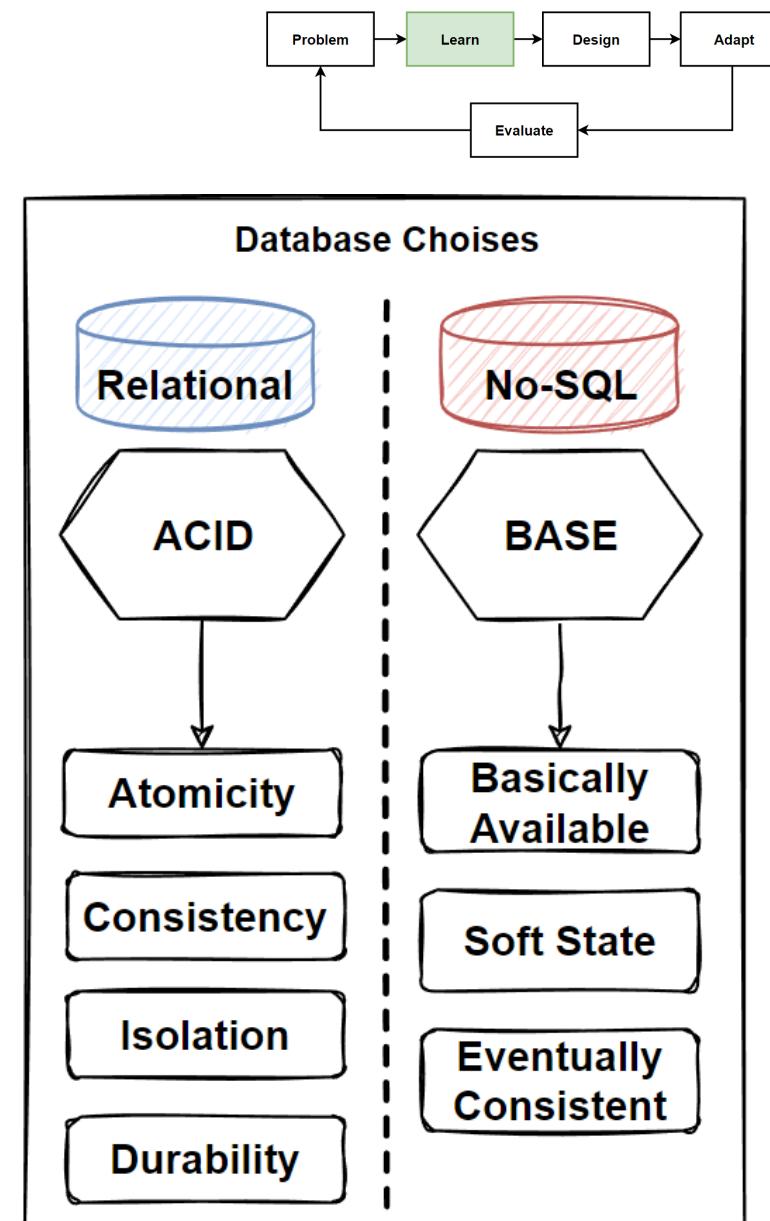


Question

- **How to Choose a Database for Microservices ?**

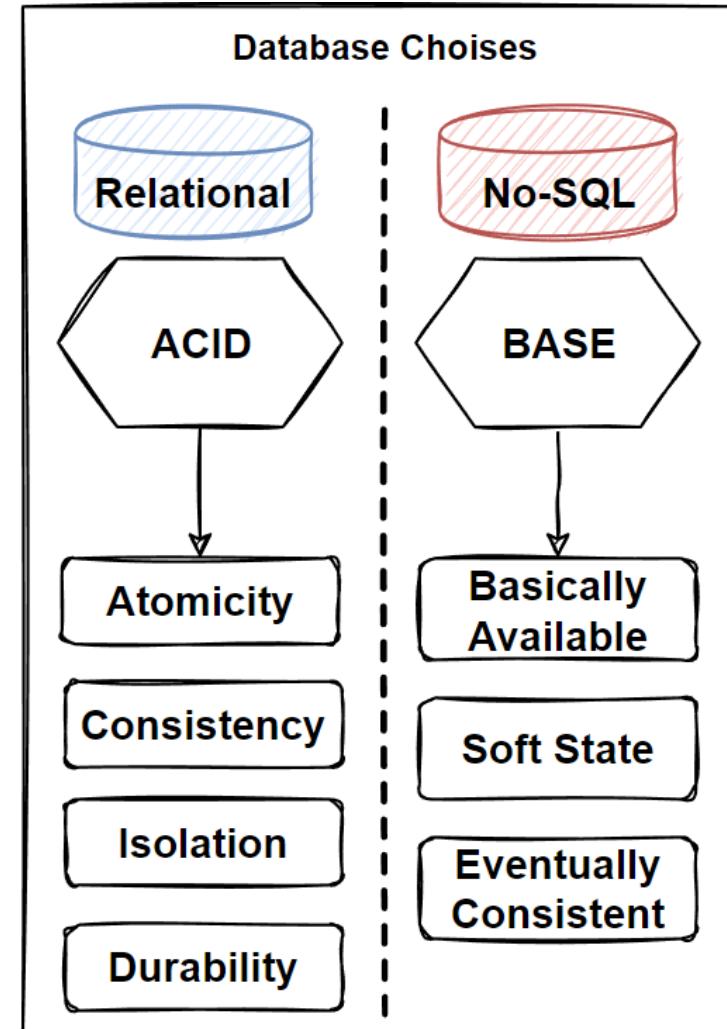
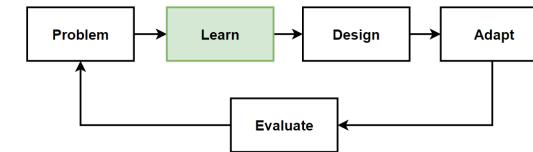
How to Choose a Database for Microservices ? (Question Set)

- **Data Consistency Level**
Do we need Strict-Strong consistency or Eventual consistency ?
- Do we need ACID compliance ? Should follow Eventual consistency in microservices to gain high scalability and availability.
- **Fixed or Flexible Schema Choise, Predictable or Dynamic Data**
Are we work with fixed or flexible schema that need to change frequently, dynamically changed data ?
- Are we have Predictable Data or Dynamic Data ?
- **High or Low Data Volume, Predictable or Un-predictable Data**
Are we work with High Volume Data or Low Volume Data ?
- Can we have predictable data that we store our microservices database ?
- NoSQL Databases prioritize partition tolerance that handling large amount of data or data coming in high velocity.



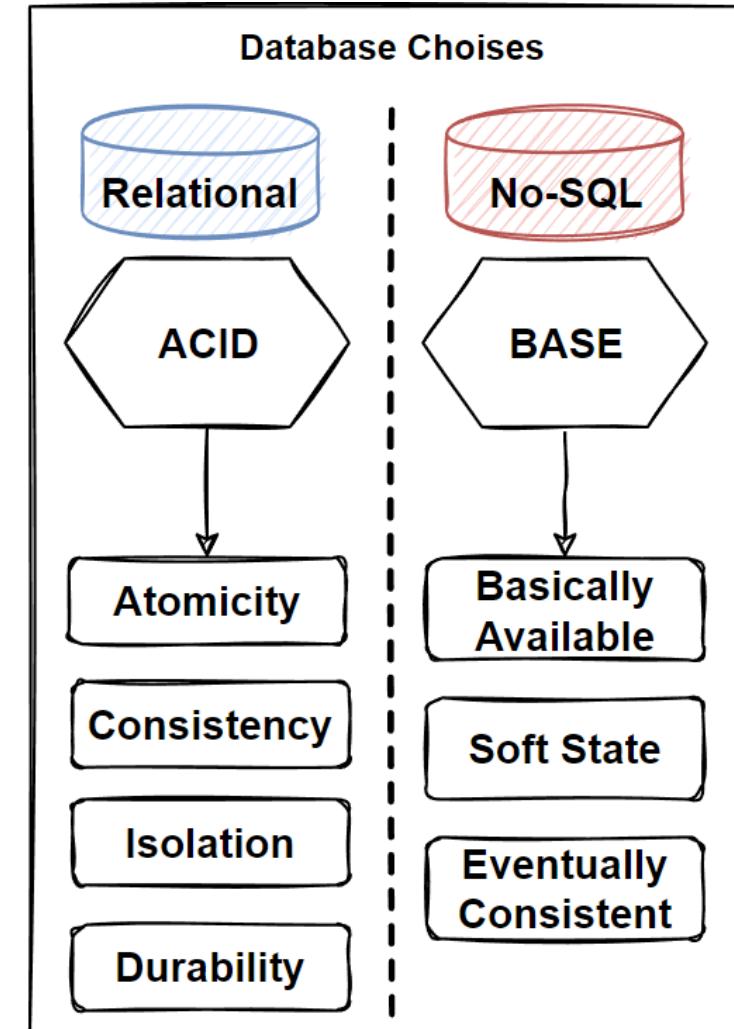
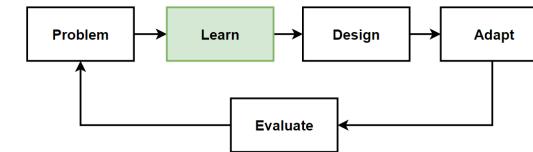
How to Choose a Database for Microservices ? (Question Set) - 2

- **Read Requirements, Relational or non-Relational Data, Complex Join Queries**
Our data is highly structured and requires referential integrity or not required for relationships that is dynamic and frequently changes ?
- Should it work with complex queries, table joins and run SQL queries on normalized data models or Retrieve data operations are simple and performs without table joins ?
- **Deployments, Centralized or De-centralized Structure**
Do we deployed to large and one or few locations with centralized structure ? or Do we need to deploy and replicate data across different geographical zones ?
- **High Performance Requirements**
Do we need to achieve fast read-write performance ?



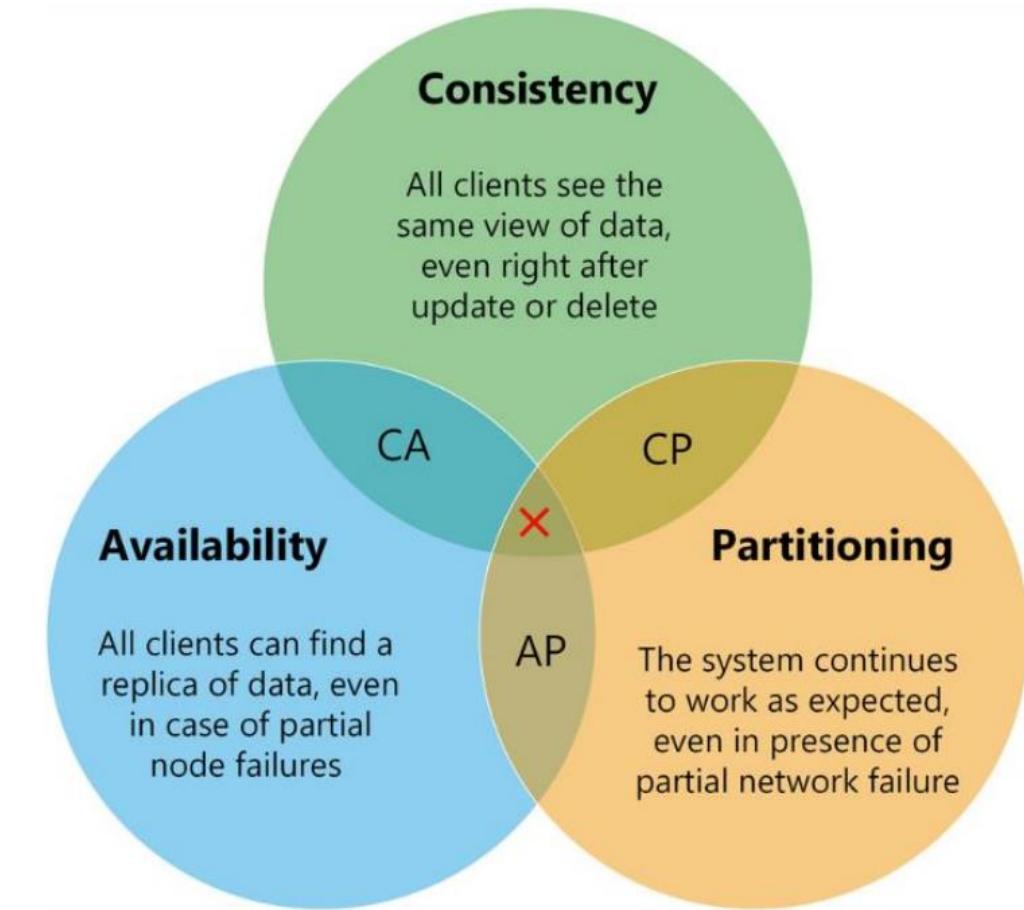
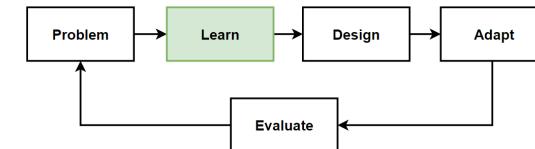
How to Choose a Database for Microservices ? (Question Set) - 3

- **High Scalability Requirements**
Do we need High Scalability Requirements both vertical and horizontally scaling ?
- To accomodate millions of request should sacrifice strong consistency.
- **High Availability and Low Latency Requirements**
Do we need High Availability and Low Latency Requirements that need to separate data across different geographical zones ?
- **Can we provide ALL OF THESE FEATURES at the same time ?**
Is it possible to provide High Scalability, High Availability and Low Latency with High Performance and able to run Complex Join Queries providing with ACID principles strong data consistency ?
- **Unfortunately, NO.**
- **WHY ?**
- **The CAP Theorem.**

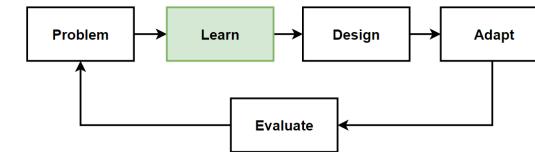


CAP Theorem

- 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**.
- Any database can **only guarantee two of the three concepts**: **Consistency**, **Availability**, and **Partition tolerance**.



CAP Theorem – Consistency, Availability, Partition Tolerance



▪ Consistency

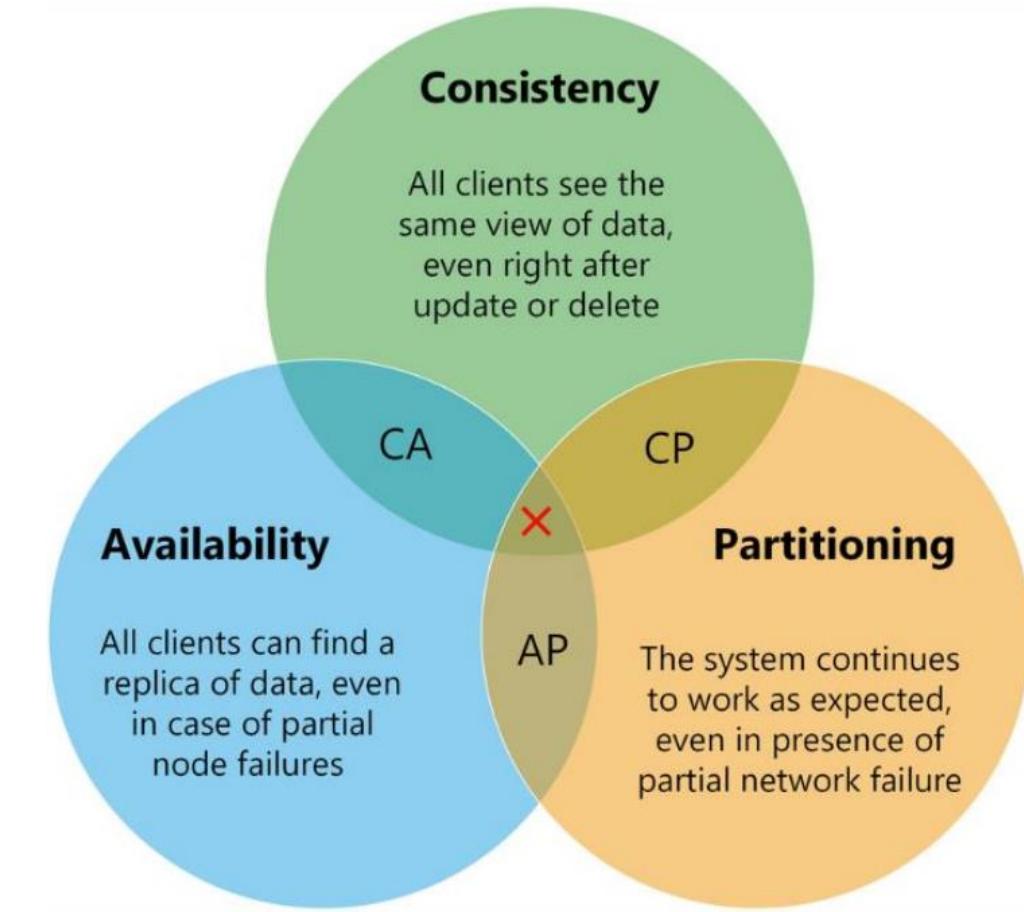
When the system get any read request, the data should return last updated value from database under all circumstances. If the data cannot be retrieved, an error should be throw. When consistent not provide, the system must block the request until all replicas update.

▪ Availability

The ability of a distributed system to respond to requests at any time. That is a distributed system can respond all request any time. Even if one node in any cluster is down, the system should be able to survive with other nodes. (fault-tolerance)

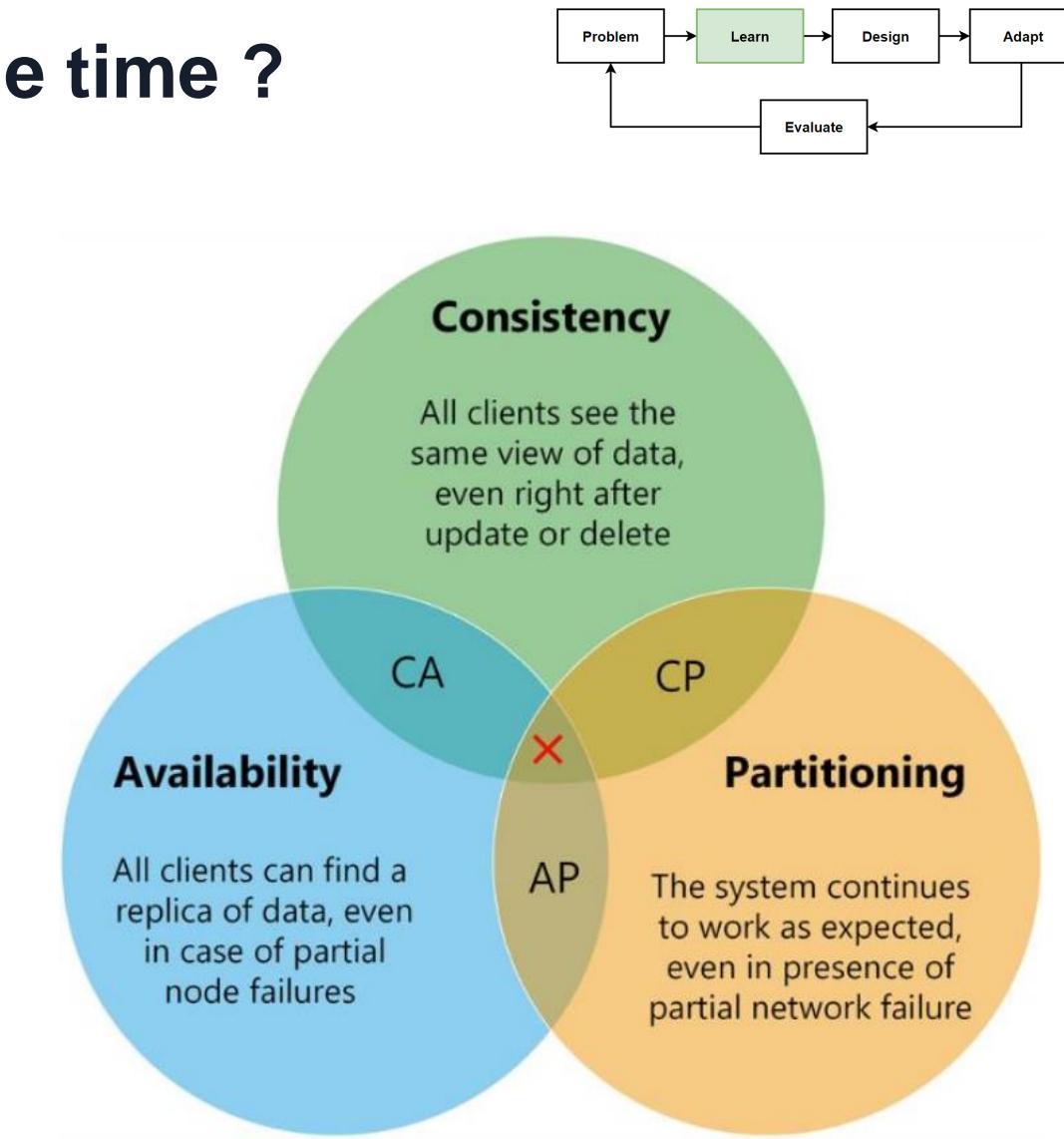
▪ Partition Tolerance

Network partitioning; parts of system are located in different networks. The ability of the system to continue its life in case of any communication problem. Guarantees the system continues to operate if one node is down.

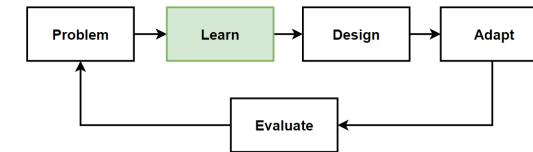


Consistency and Availability at the same time ?

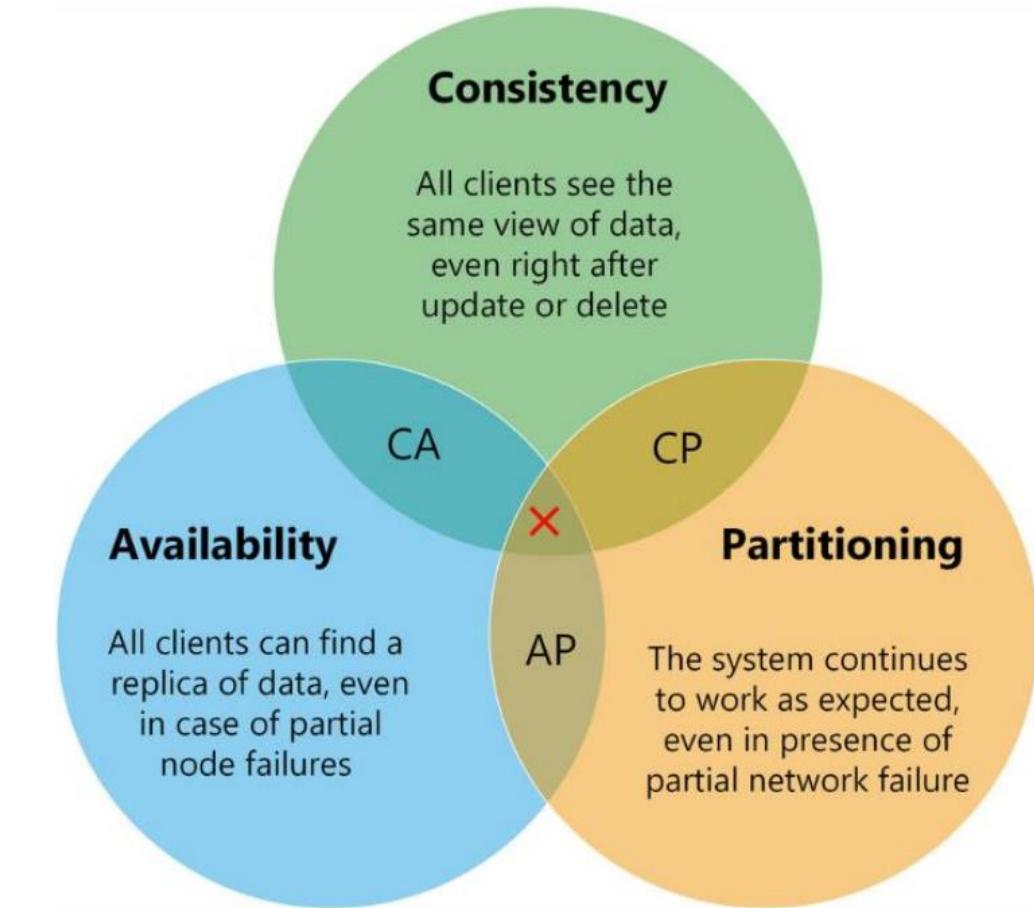
- If there is **Partition Tolerance**, either Availability or Consistency should be selected. **Should sacrifice Availability** or **Consistency** in distributed systems.
- **Partition Tolerance is a must** for distributed architectures. The emergence of NoSQL databases is to easily **overcome** the **Single Point of Failure** problem.
- Relational databases **prevent distribute data** from different nodes. **NoSQL databases don't include foreign keys, joins**, that is, relations between data.
- **Un-related data** allows it to be stored in a **distributed manner**, NoSQL databases easily scalable.
- Distributed system **doesn't have the luxury of not providing Partition Tolerance** anyway.
- **MongoDB, Cassandra**; None of them gave up on Partition Tolerance, and made a choice between Availability and Consistency.



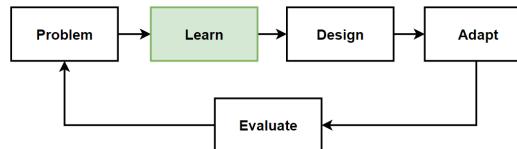
Consistency and Availability at the same time ? - 2



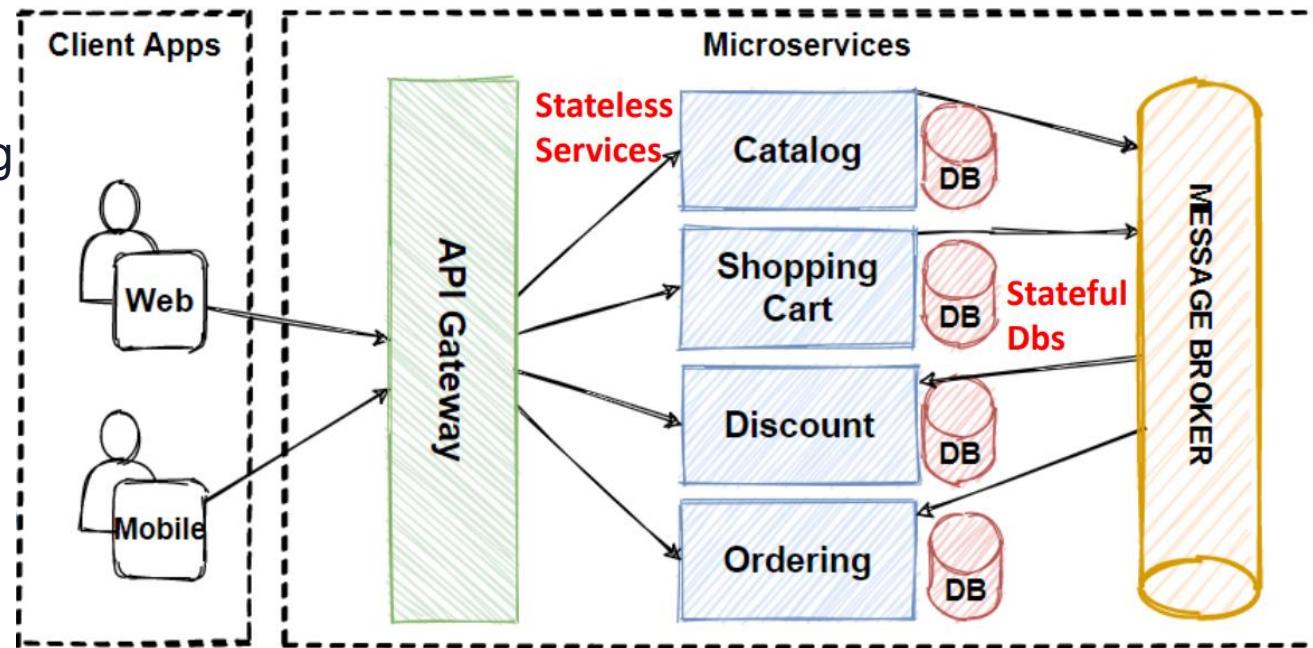
- In Distributed architecture, **Partition Tolerance** seems to be a **must-have feature**. it is necessary to choose between Consistency or Availability.
- If a system is to be **fully consistent**, it **must be sacrifice** that always available. (**High Availability**)
- If it is **desired** to be **accessible at all times**, the **consistency** should be **sacrificed**.
- In **microservices architectures** choose **Partition Tolerance** with **High Availability** and follow **Eventual Consistency**.



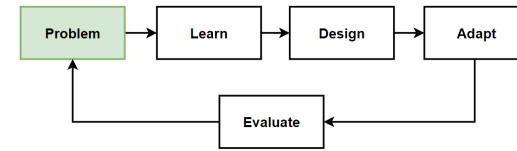
Scale Database in Microservices



- **Problem of Single Database Server**
- Why we need to use **Data Partitioning** ?
- **Scaling databases** in microservices architecture
- **Splitting database** servers with Database Sharding
- Single Database Server leads to **low performance**



Problem: Single Database Server Performs Low Performance

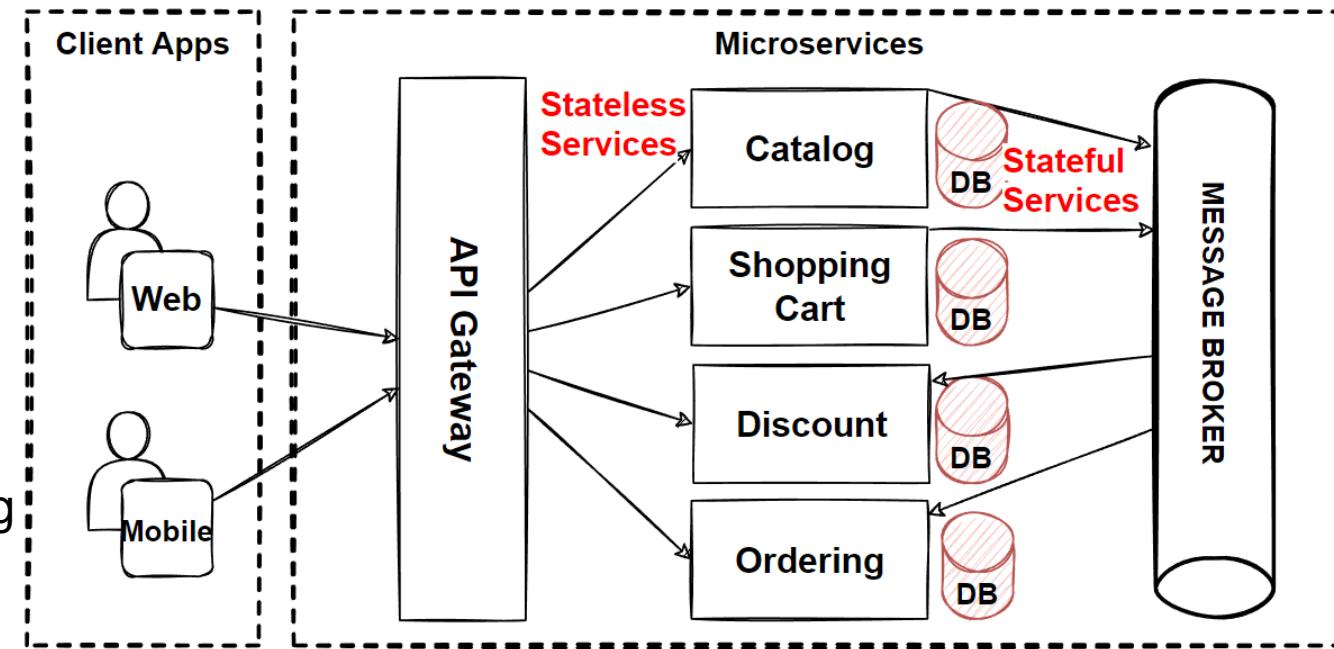


Problems

- Databases become very slow by time with app grows
- Single database goes un-manageable situation
- Database operations are become slower
- Very low performance due to slowness of single database
- Running out of disk space

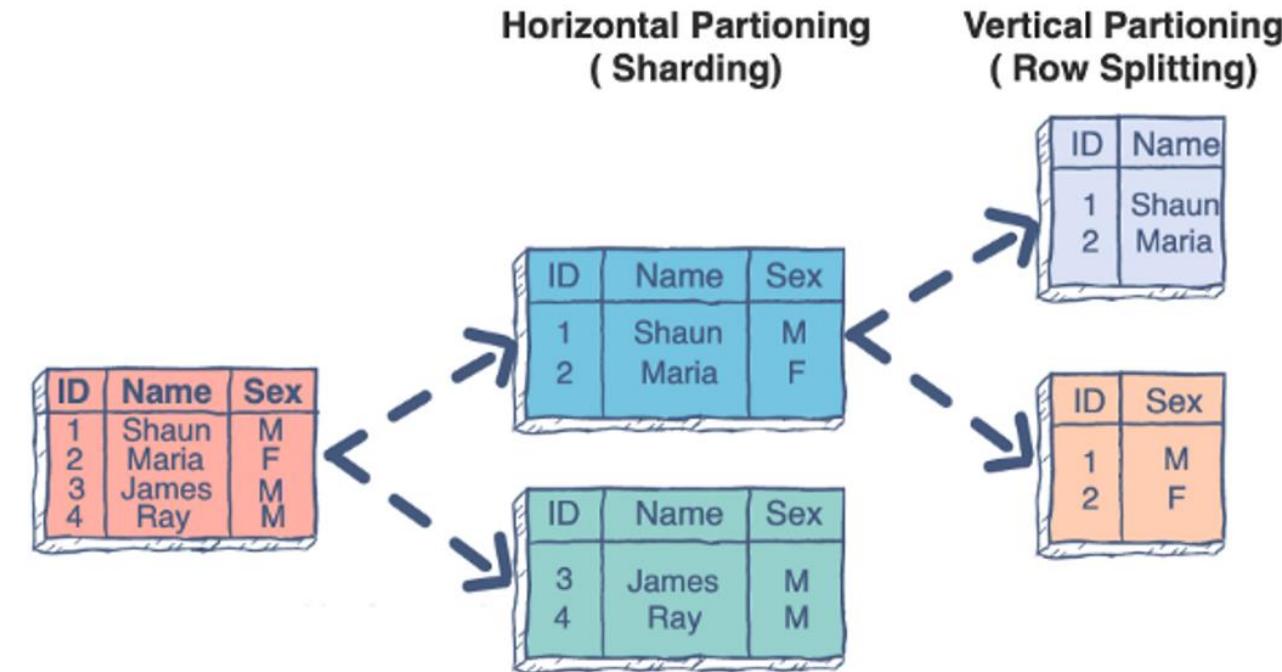
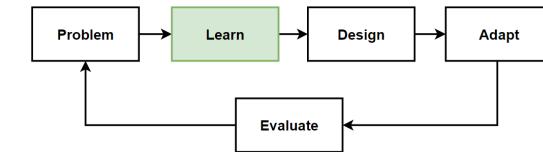
Solutions

- Data Partitioning
- Horizontal, Vertical, and Functional Data Partitioning
 - Improve availability
 - Increase scalability
 - Increase performance
 - Improve security
 - Improve data management, Operation

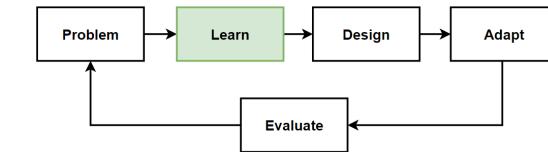


What is Data Partitioning ?

- **Data Partitioning** is divides a large dataset into several small partitions.
- Data is **divided into partitions** that can be easily managed and accessed separately.
- **Distributes data across different partitions** to improve database availability, scalability and performance.
- **Abstracting partitioning architecture from the client applications.**
- **Client applications** seems as a **single unit databases**.



Why we are using Data Partitioning ?



- **Increase Scalability**

When divide data across multiple data partitions into separate db servers, we can scale out the system without worrying any hardware limits.

- **Improve Availability**

With Separating data across multiple servers, we protect our systems for a single point of failures.

- **Increase Performance (Query and operations)**

Instead of querying whole database, system query only smaller components.

- **Improve Security**

By storing sensitive and non-sensitive data into different partitions.

- **Improve Data Management**

Due to divide tables and indexes with different partitions, its more easy to manage small units and easy to maintenance.

**Horizontal Partitioning
(Sharding)**

ID	Name	Sex
1	Shaun	M
2	Maria	F
3	James	M
4	Ray	M

ID	Name	Sex
1	Shaun	M
2	Maria	F

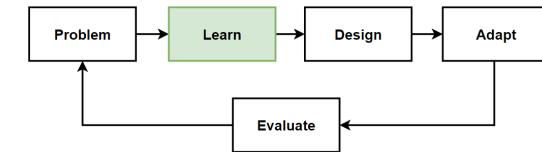
ID	Name	Sex
3	James	M
4	Ray	M

**Vertical Partitioning
(Row Splitting)**

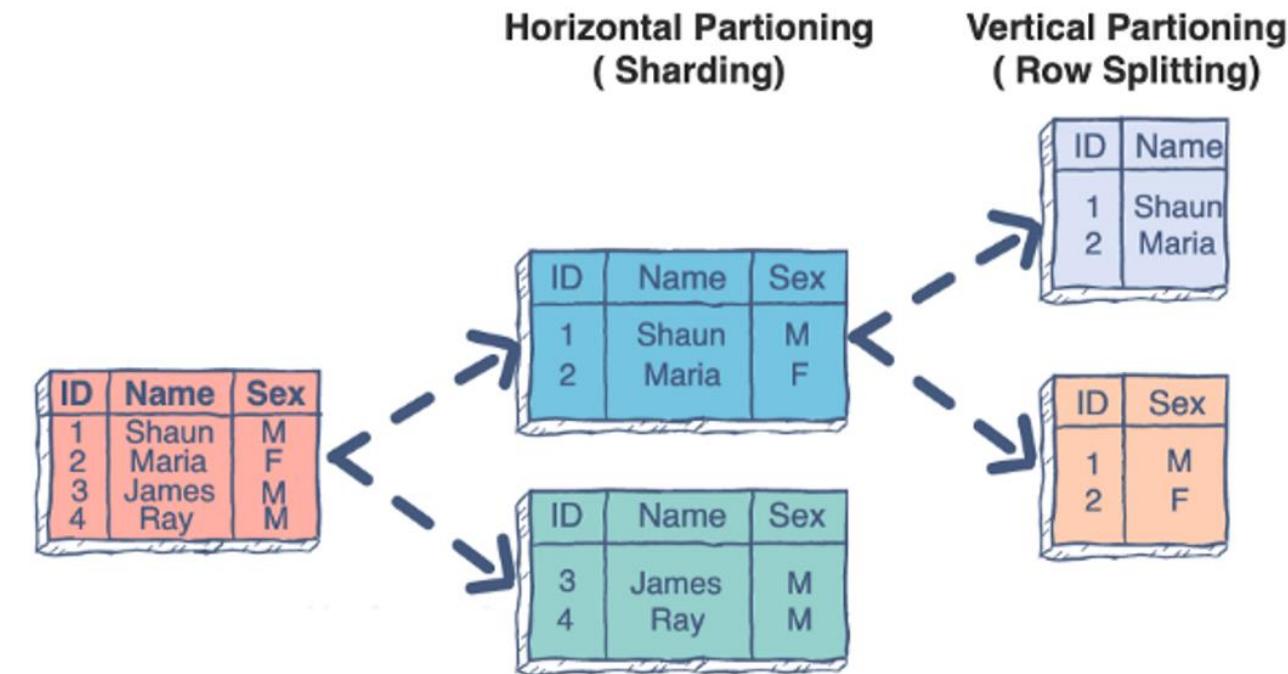
ID	Name
1	Shaun
2	Maria

ID	Sex
1	M
2	F

Horizontal, Vertical, and Functional Data Partitioning

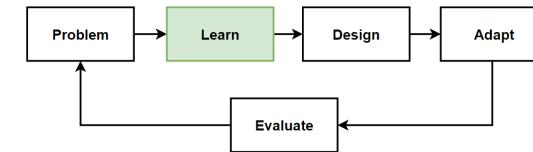


- There are three methods for partitioning data:
- Horizontal Partitioning (often called sharding) and we mostly use this portioning
- Vertical Partitioning
- Functional Partitioning



Horizontal Partitioning - Sharding

- **Horizontal partitioning** is also called **sharding**.
- **Each partition** is a **separate data store**, but all partitions have the same schema.
- Shards holds a **specific subset of the data**.
- **Splitting the table data horizontally based on partition key**.
- Tables are **divided to smaller tables** that table rows are stored different partitions.
- **The partition key** provide to distribute data among all the partitions.
- With the partition key, database **locate the specific partition**.
- Data requests are **spreading the load** over more servers and provide more performance.
- The **disadvantages of horizontally partitioning** could be the setting **wrong partition key**.



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			VS2	
CUSTOMER ID	FIRST NAME	LAST NAME	CUSTOMER ID	CITY
1	Alice	Anderson	1	Austin
2	Bob	Best	2	Boston
3	Carrie	Conway	3	Chicago
4	David	Doe	4	Denver

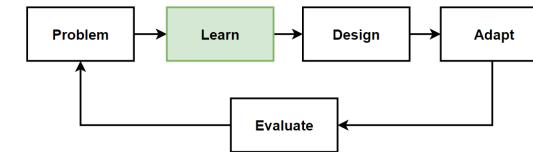
Horizontal Shards

HS1				HS2			
CUSTOMER ID	FIRST NAME	LAST NAME	CITY	CUSTOMER ID	FIRST NAME	LAST NAME	CITY
1	Alice	Anderson	Austin	3	Carrie	Conway	Chicago
2	Bob	Best	Boston	4	David	Doe	Denver

[An Overview of Sharding & Partitioning | Hazelcast](#)

Vertical Partitioning

- **Vertical partitioning** is a "Row Splitting".
- **Each partition holds a subset of the columns for table** in the database.
- The data **divided based on columns** and we can divide by **mostly visited columns** and the other columns in different servers.
- **Frequently accessed columns** can be used in one vertical partition and less frequently accessed fields in another.
- **Example case:** Facebook, the user profile data can be in different partition and user mostly visited data can be different server.
- **Benefit of Vertical Partitioning** is you can separate the critical and mostly visited columns in a separate server.
- **Best practice:** Divide by rarely-changes and frequently-change columns into different servers.



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

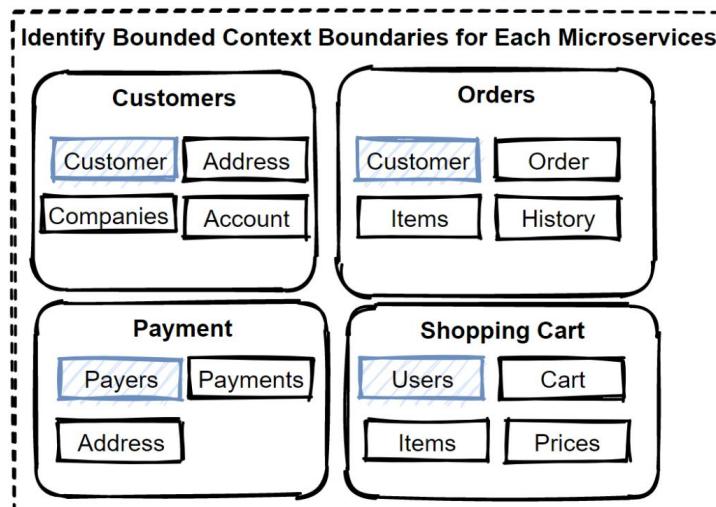
HS2

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

[An Overview of Sharding & Partitioning | Hazelcast](#)

Functional Partitioning

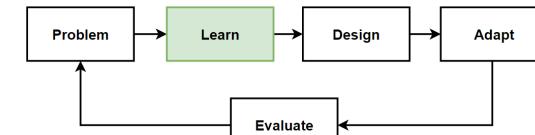
- The data is divided by following the **bounded context** or subdomains.
- Data is segregated according to **usage of bounded contexts** in the system.
- Its like **decomposing microservices** as per responsibilities with considering bounded contexts.
- **Functional partitioning** into **Microservices Decomposition** section.
- Apply functional partitioning following the **Decomposition patterns**:



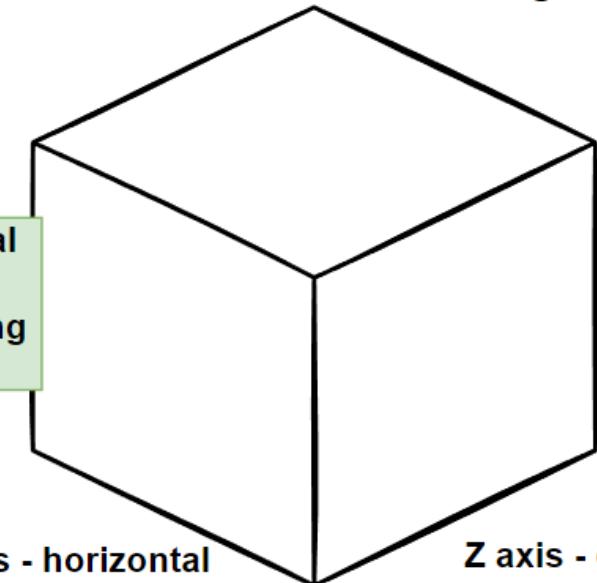
Y axis - functional decomposition
Scale by splitting small services

X axis - horizontal duplication
Scale by Clonning

Z axis - data partioning
Scale by splitting data

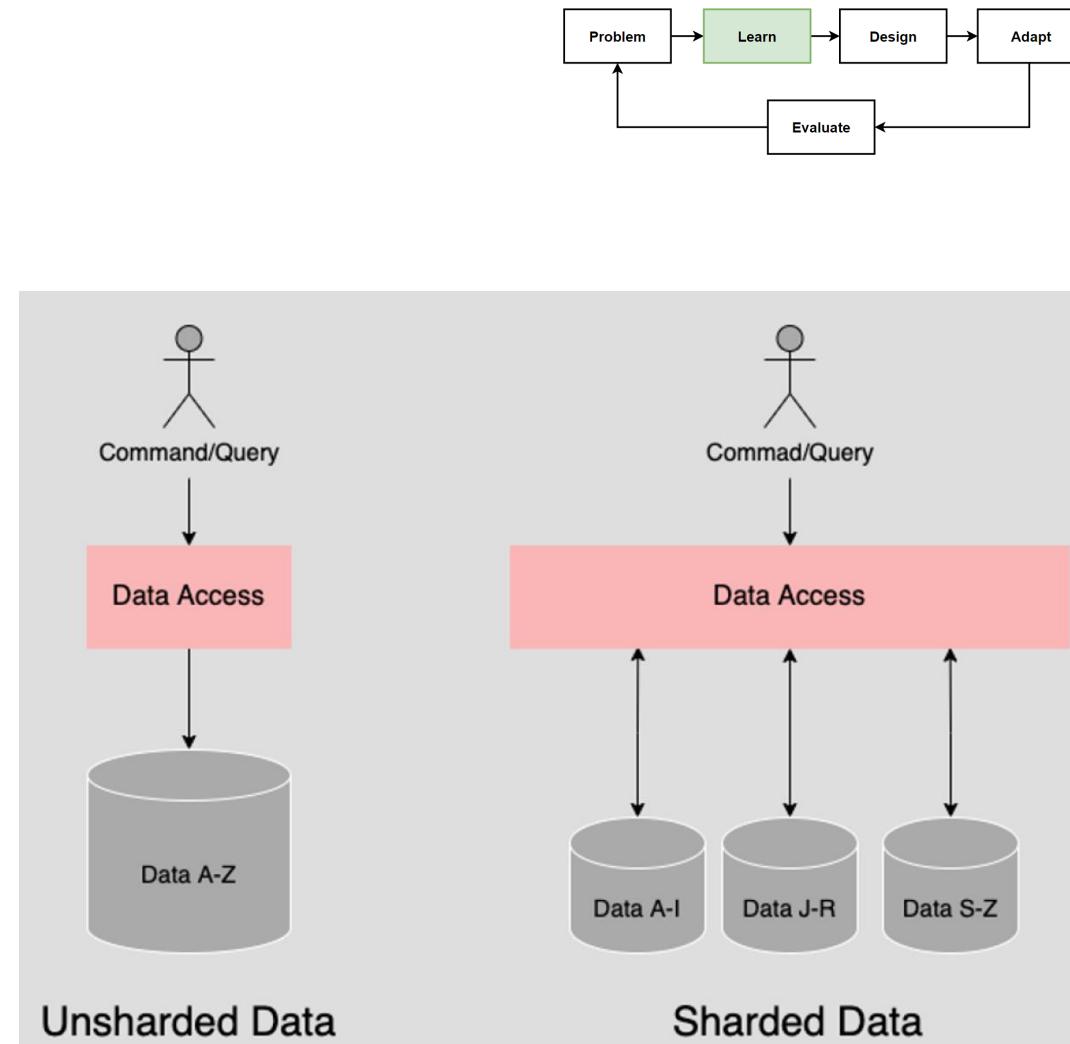


3 Dimensions of Scaling



Database Sharding Pattern

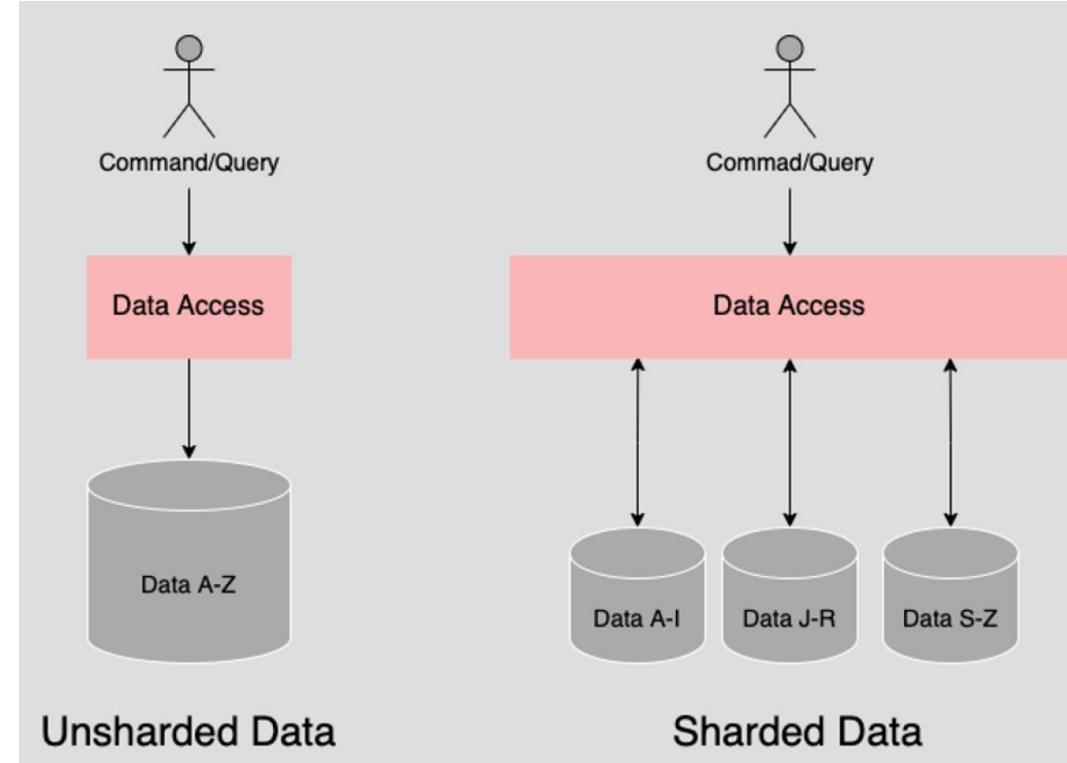
- **Sharding** is "a small piece or part", meaning to be a small piece or part of something.
- **Database sharding** is the **separation of the data** into unique small pieces of database which called shards or chunks.
- When storing and accessing large volumes of data in microservices architectures, we should **divide a databases** into a **set of horizontal partitions or shards**.
- **Each shard has the same schema**, holds its own distinct subset of the data.
- Shardings **enable to scale** the system with adding new shards according to storage needs.
- It can **improve performance** and **reduce contention** by balancing the workload across different shards.
- We can **gain cloud power** with locating physical servers closely which users can access the data performantly.



<https://www.redhat.com/architect/pros-and-cons-sharding>

Database Sharding Pattern - 2

- **Databases** dividing into **shards** with **partition keys**.
- These **partition keys** provide to decide which data should be placed in each shard.
- The **partition key** should be **static**. It shouldn't affect by the data changes.
- **Database Sharding** provides dividing a **data store physically** and **organizes** the data into **shards**.
- When an application **retrieves data by querying database**, the sharding logic **redirects** the application to the correct **shard server**.
- All these database sharding operations are **abstracting the physical location** of the data in the sharding logic from client applications.



<https://www.redhat.com/architect/pros-and-cons-sharding>

Benefits of Database Sharding Pattern

- **Increase Scalability**

When divide data across multiple data partitions into separate db servers, we can scale out the system without worrying any hardware limits.

- **Improve Availability**

With Separating data across multiple servers, we protect our systems for a single point of failures.

- **Increase Performance (Query and operations)**

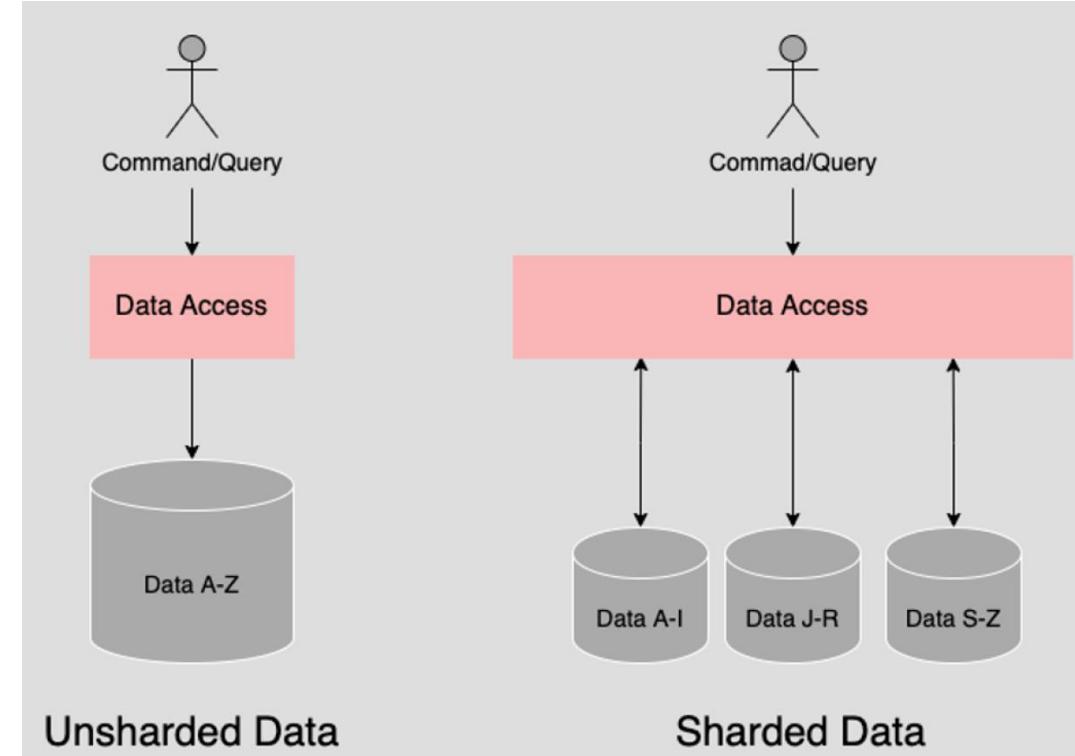
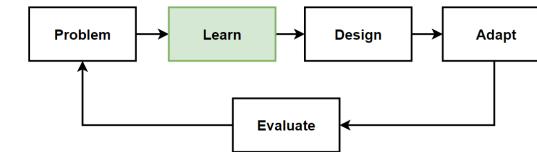
Instead of querying whole database, system query only smaller components.

- **Improve Security**

By storing sensitive and non-sensitive data into different partitions.

- **Improve Data Management**

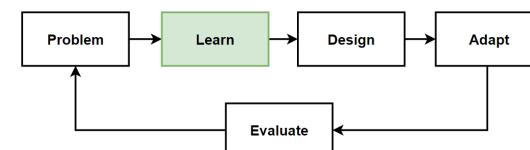
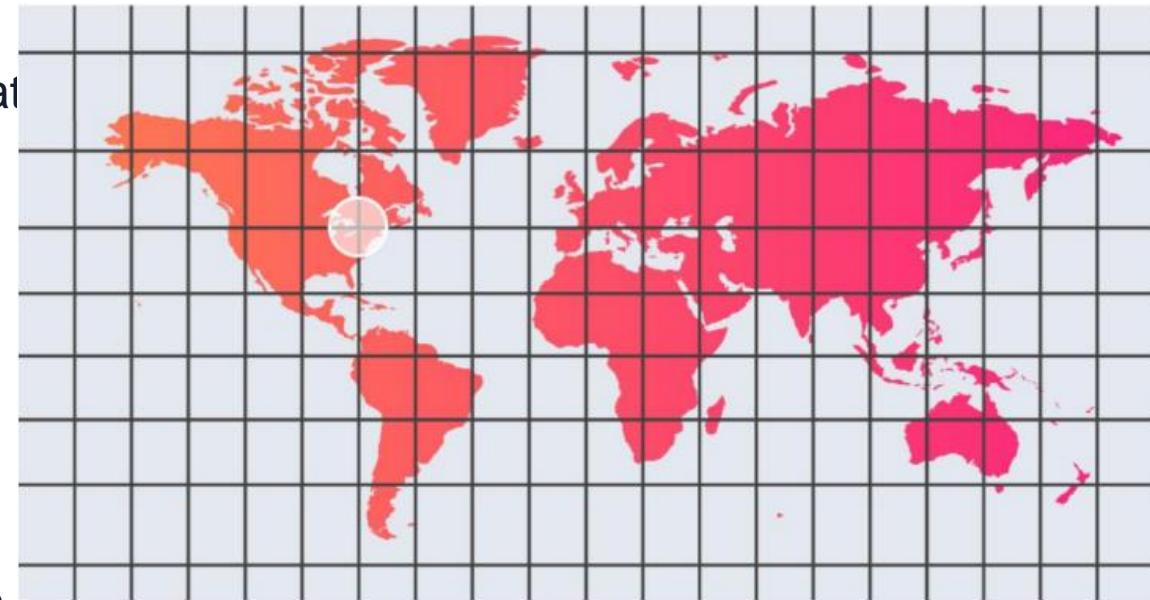
Due to divide tables and indexes with different partitions, its more easy to manage small units and easy to maintenance.



<https://www.redhat.com/architect/pros-and-cons-sharding>

Tinder System Design Example of Database Sharding Pattern

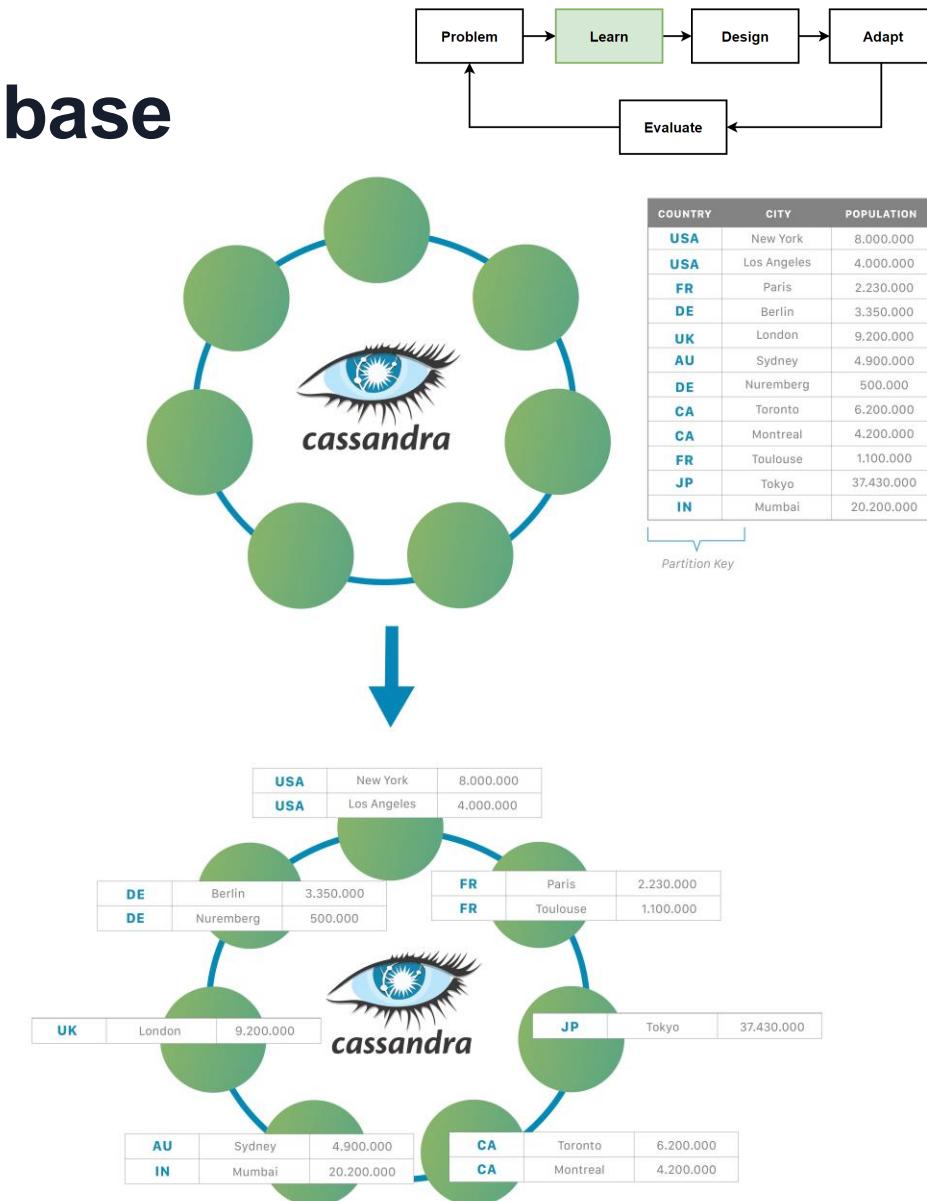
- Tinder is very good example of Database **Sharding Pattern**.
- Tinder allows to **match** and **meet** other **people** who use the application near you (around 160km) **based on location**.
- To **find people near you very quickly** and offer choices that meet the criteria what you set.
- How **Tinder match peoples** who are **near to each other** ?
- Tinder **segments users** based on **their location**.
- This is called **GeoSharding**, that is, **location-based database sharding**.
- Sharding by dividing the world map into boxes with their locations and matches them in only into that box locations in the world.



Cassandra No-Sql Database

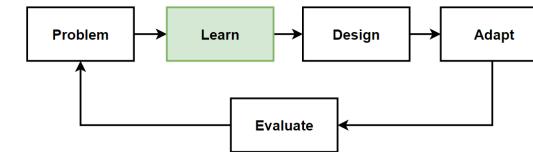
Peer-to-Peer Distributed Wide Column Database

- Cassandra is very good example of implementation **Database Sharding Pattern** and using **Horizontal Scaling** features.
- Cassandra is a **distributed database** from Apache Foundation.
- It is **highly scalable** and designed to manage very **large amounts of structured data**.
- It provides **high availability** with **no single point of failure**.
- Apache Cassandra is a highly scalable, high-performance **distributed database** designed to **handle large amounts of data** across many **different located servers**.
- **Apache Cassandra feautures;**
 - Elastic scalability - It is scalable, fault-tolerant, and consistent.
 - No single point of failure.
 - Column-oriented database.
 - Flexible data storage.
 - Easy data distribution.

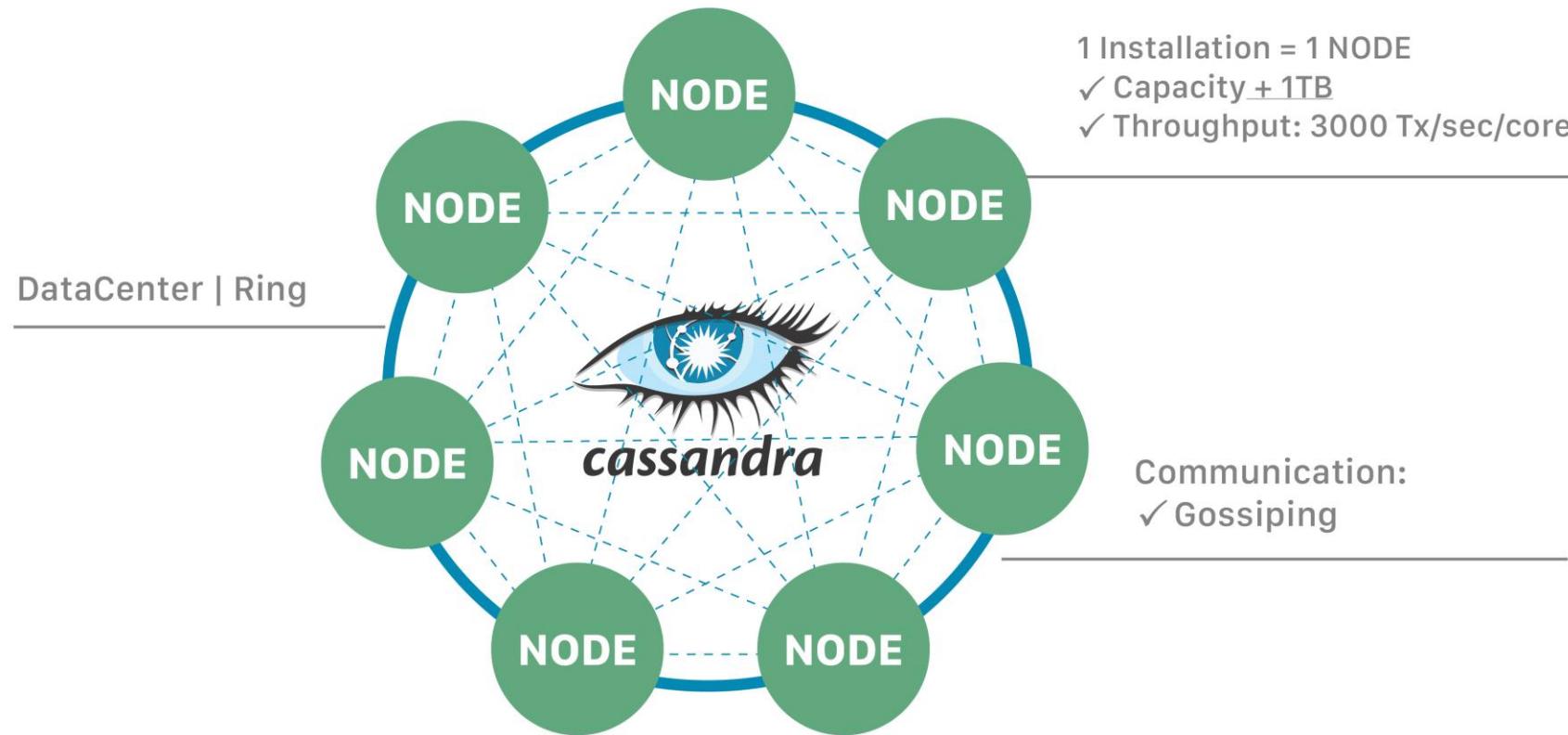


https://cassandra.apache.org/_cassandra-basics.html

Cassandra Architecture



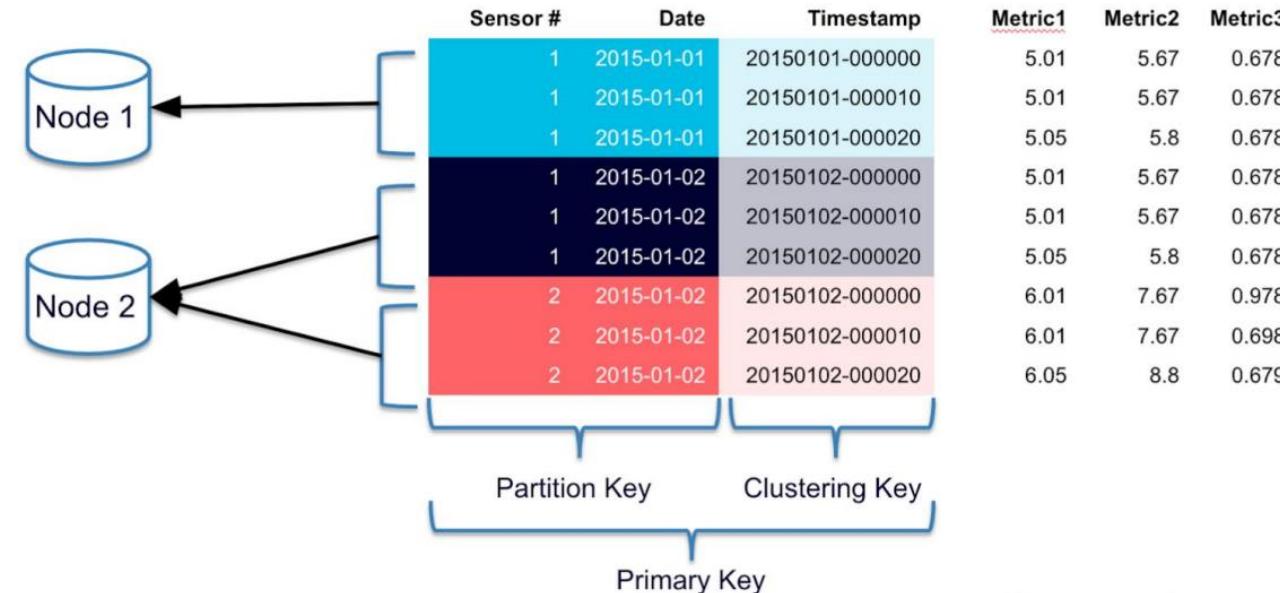
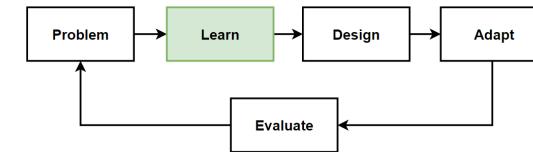
ApacheCassandra™= NoSQL Distributed Database



https://cassandra.apache.org/_cassandra-basics.html

Why Cassandra No-Sql Database ?

- **Cassandra** has an **auto-sharding feature** and it helps keep data divided among nodes.
- All of the columns that will **decide on which node** the data should be kept are called "**Partition Key**".
- **Partition Key** is set to **Sensor#** and **Date**.
- The data will be **stored in nodes** determined by inserting the value of these **two columns** with hash function. It becomes **Partition Keys**.
- **CAP Theorem**: either Availability or Consistency must be selected in a distributed system.
- **Microservice architectures** decided that **Availability is more important** to them and sacrifice for strict consistent and **embrace to Eventual Consistency** would be sufficient.



https://cassandra.apache.org/_cassandra-basics.html

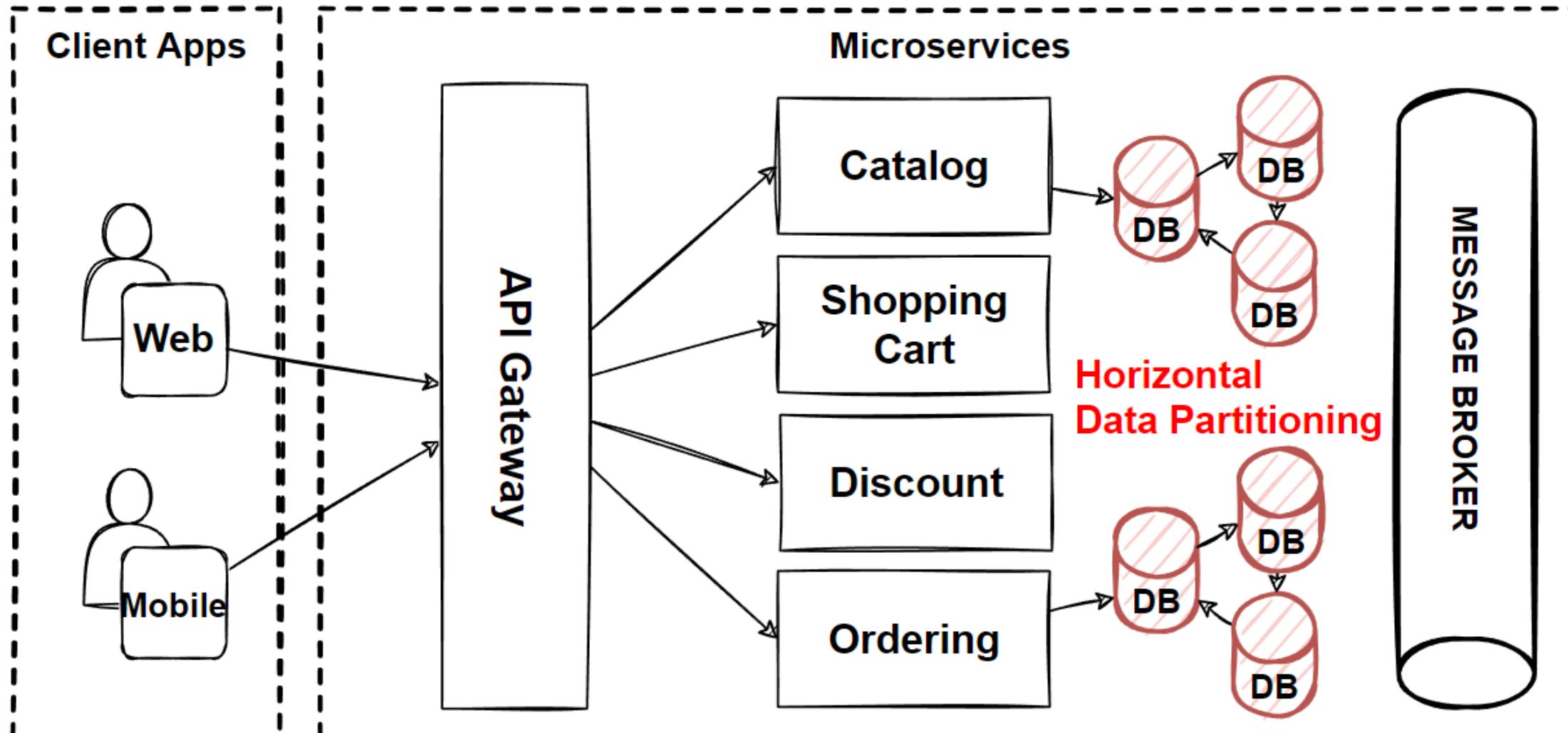
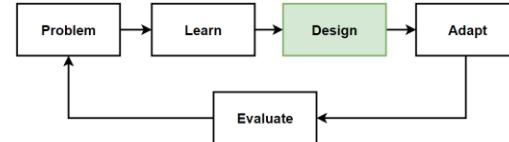
Before Design – What we have in our design toolbox ? - Previous

Architectures	Patterns&Principles	Microservices Communications	Microservices Async Communications	FR
• Microservices Architecture	<ul style="list-style-type: none">The Database-per-Service PatternPolygot PersistenceDecompose services by scalabilityThe Scale CubeMicroservices Decomposition Pattern	<ul style="list-style-type: none">HTTP Based RESTful APIGraphQL APIgRPC APIWebSocket APIGateway Routing PatternGateway Aggregation PatternGateway Offloading PatternAPI Gateway PatternBackends for Frontends Pattern-BFFService Aggregator PatternService Registry/Discovery Pattern	<ul style="list-style-type: none">Single-receiver Message-based Communication (one-to-one model)Multiple-receiver Message-based Communication (one-to-many model-topic)Dependency Inversion Principles (DIP)Fan-Out Publish/Subscribe Messaging PatternTopic-Queue Chaining & Load Balancing Pattern	<ul style="list-style-type: none">List productsFilter products as per brand and categoriesPut products into the shopping cartApply coupon for discountsCheckout the shopping cart and create an orderList my old orders and order items history
• Microservices Communications Patterns				Non-FR
				<ul style="list-style-type: none">High ScalabilityHigh AvailabilityMillions of Concurrent UserIndependent

Before Design – What we have in our design toolbox ? - Data

Architectures	Patterns&Principles	Microservices Data Management	Non-FR	FR
<ul style="list-style-type: none">• Microservices Architecture	<ul style="list-style-type: none">• The Database-per-Service Pattern• Polygot Persistence• Decompose services by scalability• The Scale Cube• Microservices Decomposition Pattern• Microservices Communications Patterns• Microservices Data Management Patterns	<ul style="list-style-type: none">• The Shared Database Anti-pattern• Relational and NoSQL Databases• CAP Theorem – Consistency, Availability, Partition Tolerance• Data Partitioning: Horizontal, Vertical and Functional Data Partitioning• Database Sharding Pattern	<ul style="list-style-type: none">• High Scalability• High Availability• Millions of Concurrent User• Independent Deployable• Technology agnostic• Data isolation• Resilience and Fault isolation	<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history

Design: Microservices Architecture with Database Sharding Pattern

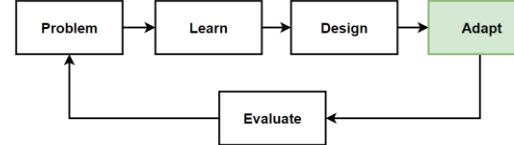


Peer-to-Peer Distributed Wide Column Database
Master-Master (Master-less) architecture

Mehmet Ozkaya

444

Adapt: Microservice Architecture with Database Sharding Pattern - Cassandra



Frontend SPAs

- Angular
- Vue
- React

API Gateways

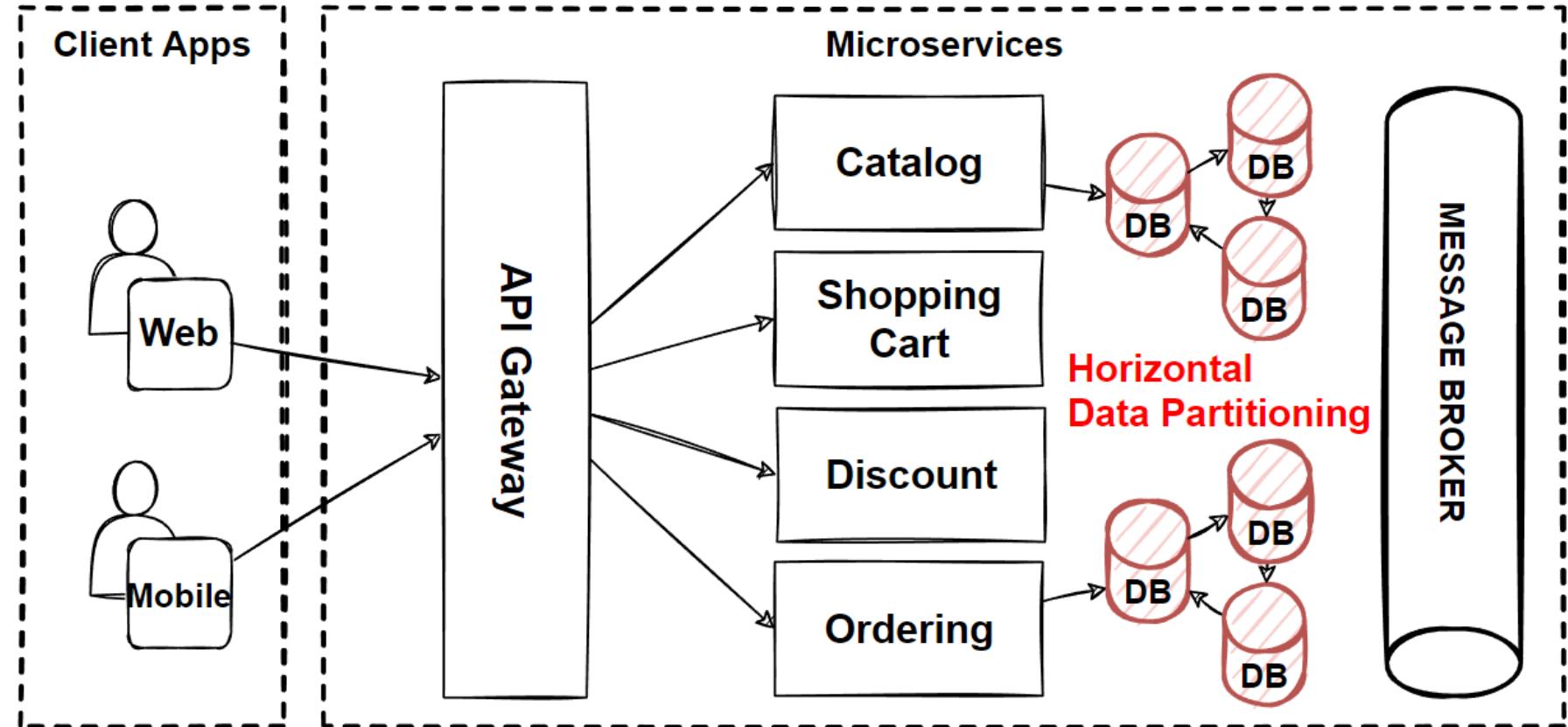
- Kong Gateway
- Express Gateway
- Amazon AWS API Gateway

Message Brokers

- Kafka
- RabbitMQ
- Amazon EventBridge, SNS

Backend Microservices

- Java – Spring Boot
- .Net – Asp.net
- JS – NodeJS



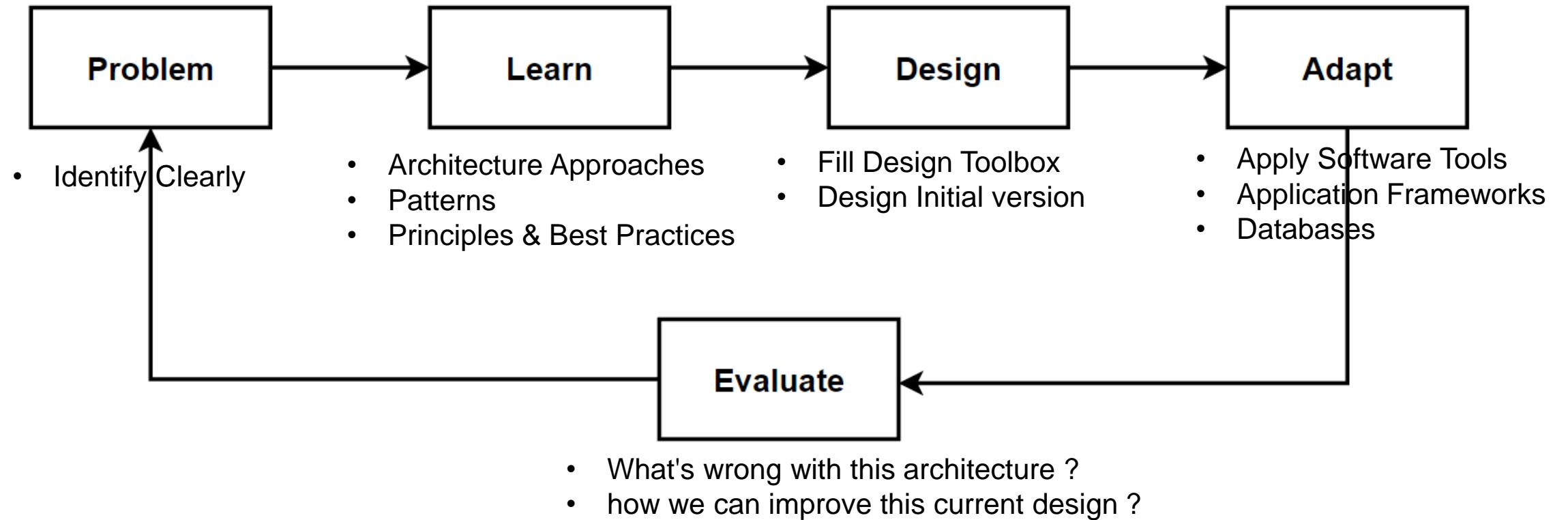
No-SQL Database

- Cassandra – NoSQL Peer-to-Peer
- MongoDB – NoSQL Document DB
- Redis – NoSQL Key-Value Pair

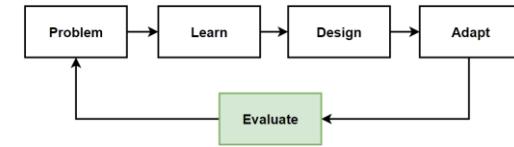
Cloud No-SQL Database

- Amazon DynamoDB
- Azure CosmosDB

Way of Learning – The Course Flow

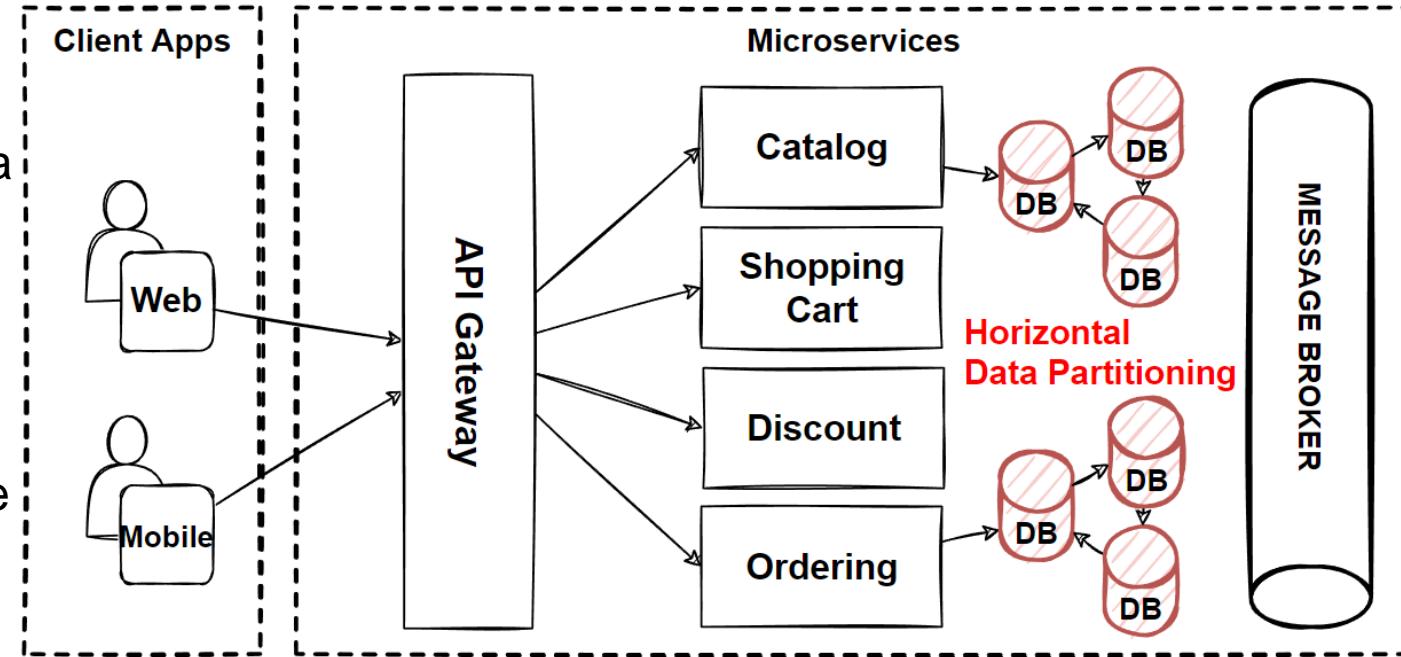


Evaluate: Microservice Architecture with Database Sharding Pattern - Cassandra



Benefits

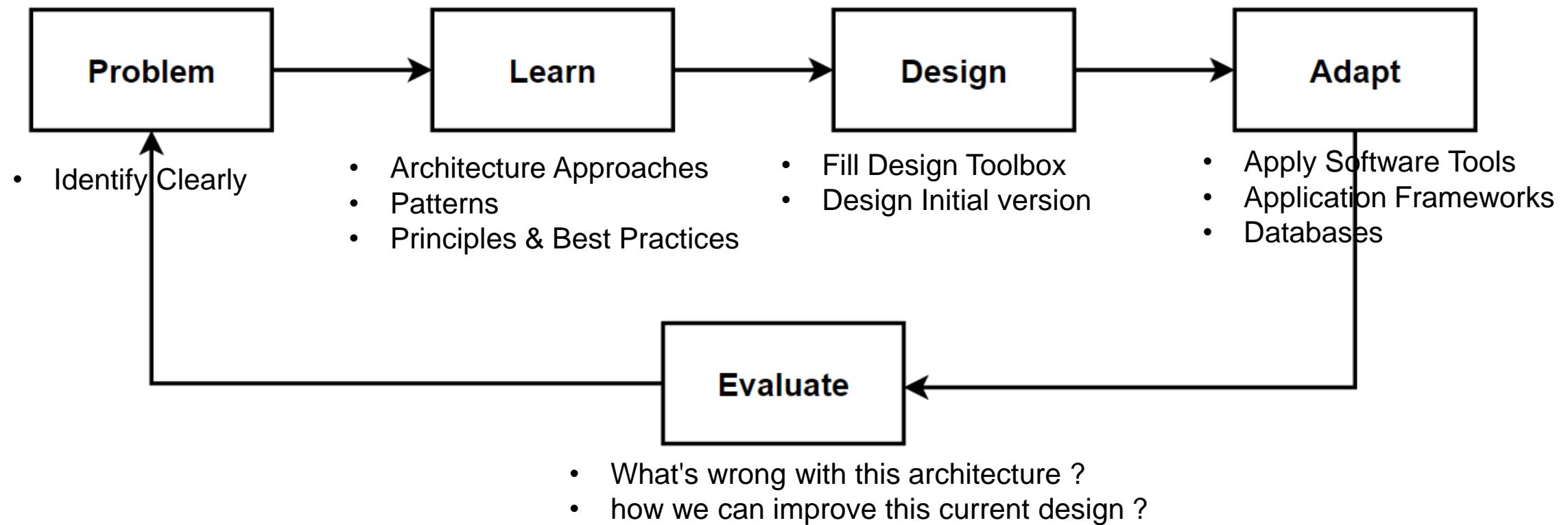
- Distributed Horizontally Scaled Databases that ready for Kubernetes Deployments
- Data Partitioning: Horizontal and Functional Data Partitioning
- Database Sharding Pattern: Increase Database Scalability, Improve Availability, Manageability
- Master-Master (Master-less) architecture with Peer-to-Peer Distributed Wide Column Database



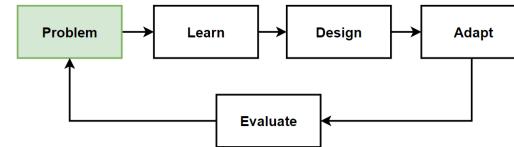
Drawbacks

- CAP Theorem: Data Consistency, Sacrifice Strong Consistency to Partition Tolerance
- Cross-services Queries
- Complex JOIN requires operations
- Lack of Read-Write Database Separations
- Distributed Transaction Management

Way of Learning – The Course Flow



Problem: Cross-Service Queries and Write Commands on Distributed Scaled Databases



Considerations

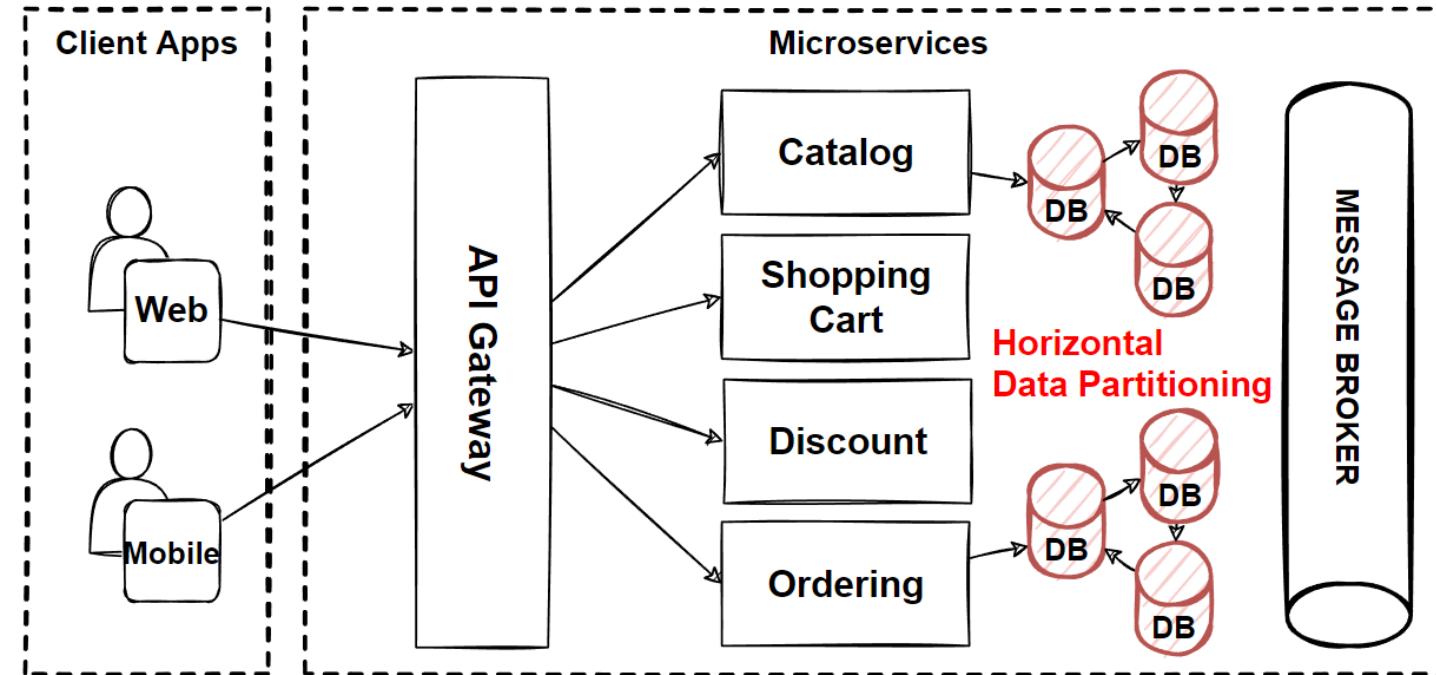
- Cross-services queries that retrieve data from several microservices ?
- Separate read and write operations at scale ?

Problems

- Cross-Service Queries with Complex JOIN operations
- Read and write operations at scale
- Distributed Transaction Management

Solutions

- Microservices Data Query Pattern and Best Practices
- Materialized View Pattern
- CQRS Design Pattern
- Event Sourcing Pattern



Microservices Data Management - Commands and Queries

Microservices Data Query Pattern and Best Practices

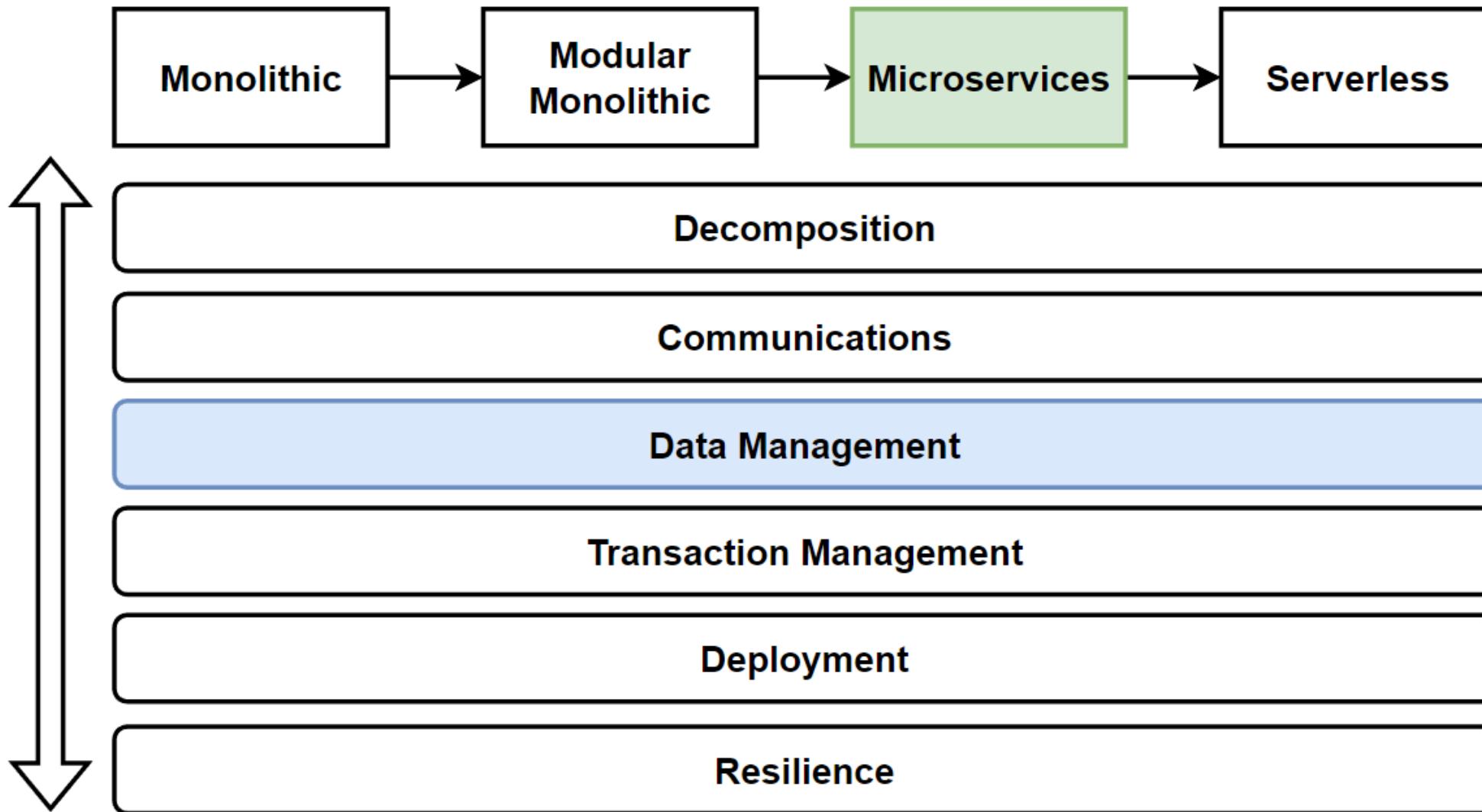
Materialized View Pattern

CQRS Design Pattern

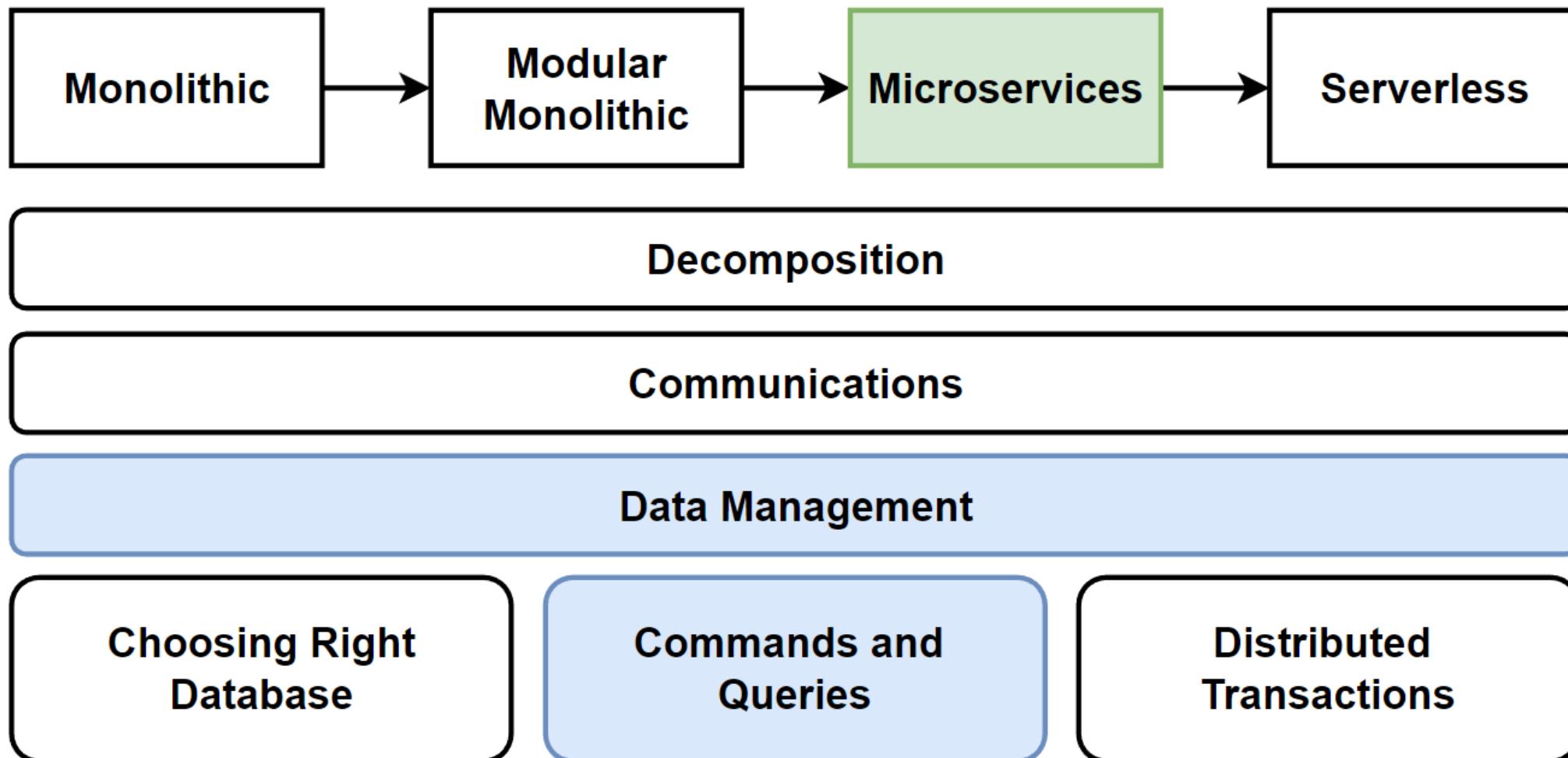
Event Sourcing Pattern

Eventual Consistency Principle

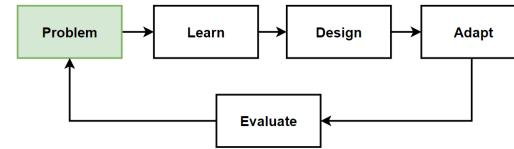
Architecture Design – Vertical Considerations



Microservices Data Management - Main Topics



Problem: Cross-Service Queries and Write Commands on Distributed Scaled Databases



Considerations

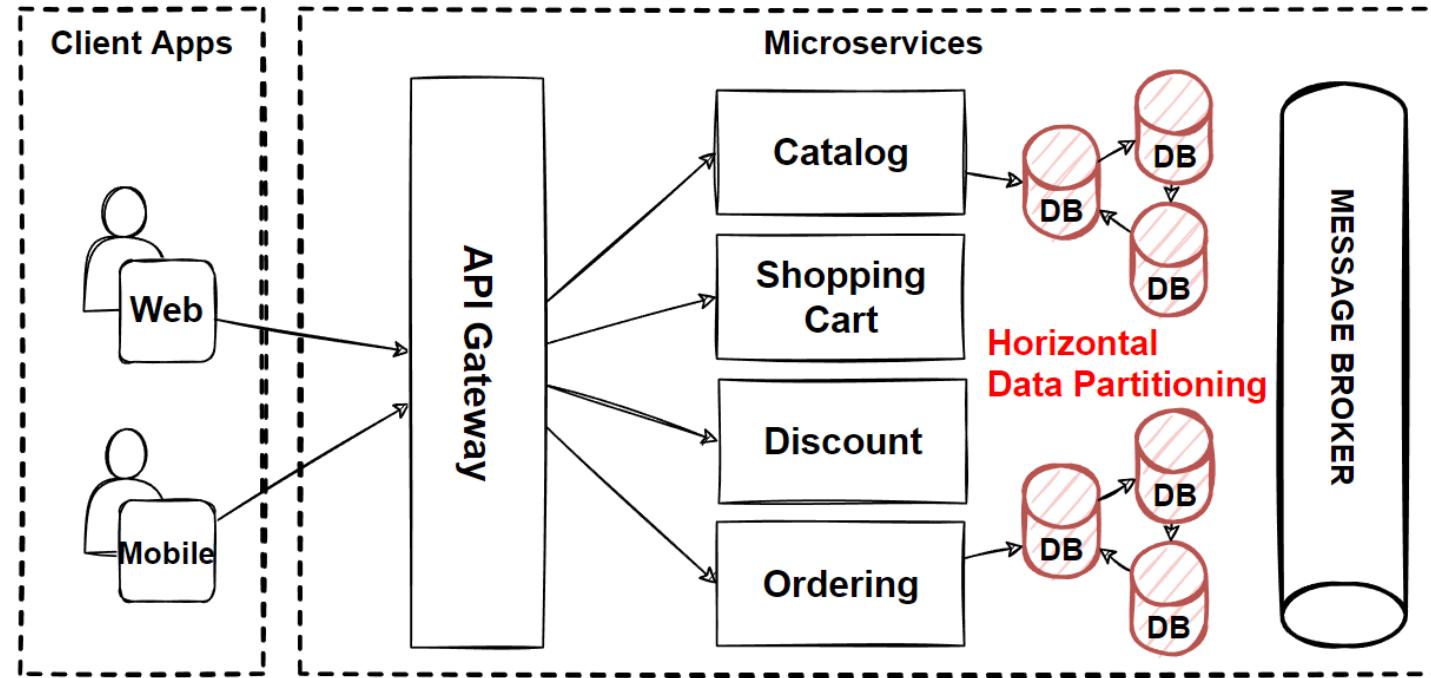
- Cross-services queries that retrieve data from several microservices ?
- Separate read and write operations at scale ?

Problems

- Cross-Service Queries with Complex JOIN operations
- Read and write operations at scale
- Distributed Transaction Management

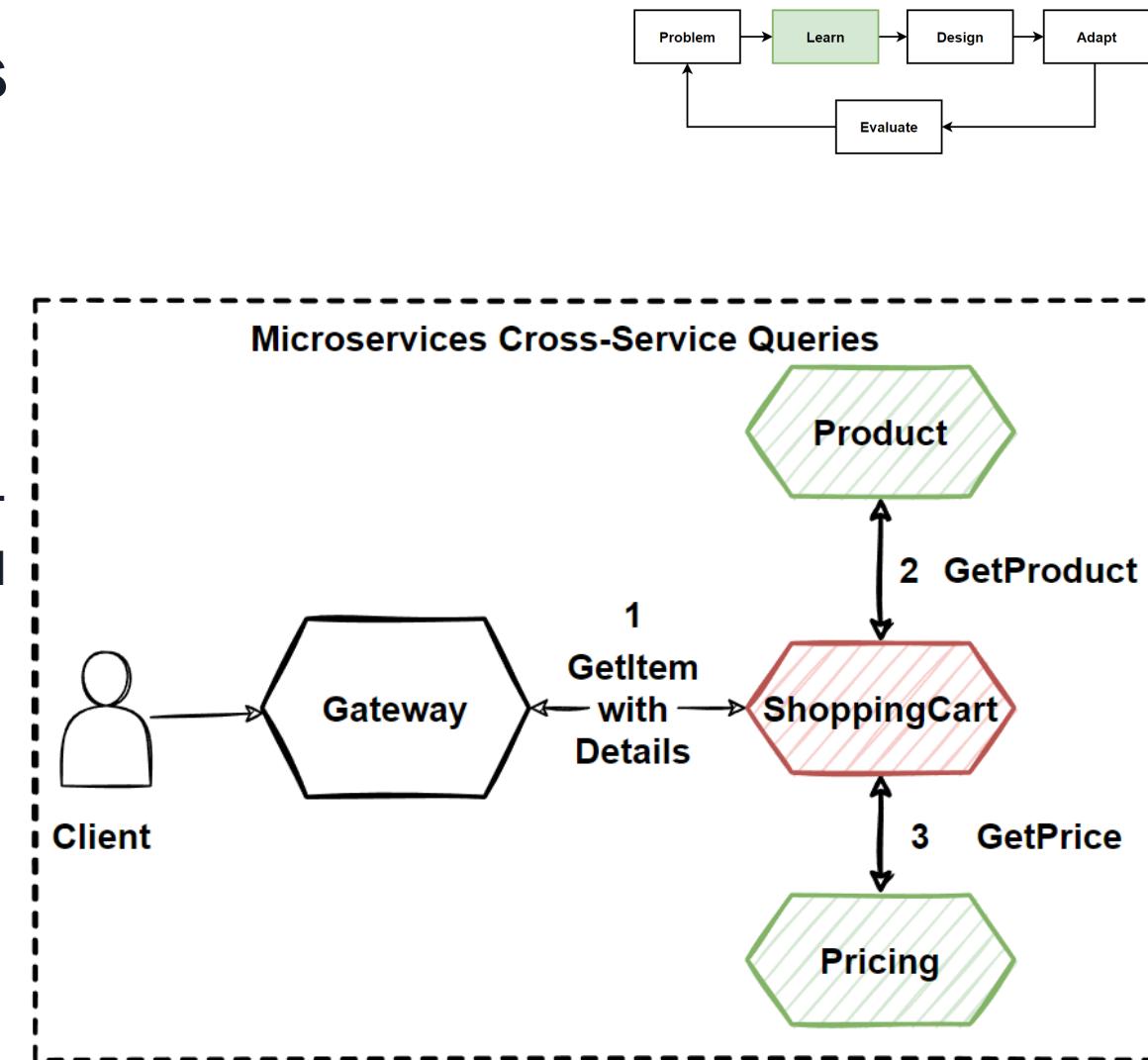
Solutions

- Microservices Data Query Pattern and Best Practices
- Materialized View Pattern
- CQRS Design Pattern
- Event Sourcing Pattern

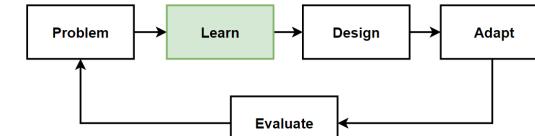


Microservices Cross-Service Queries

- **Monolithic architectures**, its very **easy to query** different entities, **Querying data** across multiple tables is **straightforward**.
- **Microservices architectures** uses **polyglot persistence**, has different databases, **need strategy** to manage queries.
- What if the client requests are **visit more than one** internal microservices ?
- **E-commerce application** we have **product, basket, discount, ordering** microservices that **needs to interact** each other to perform customer use cases.
- **Integrations** are **querying each services** data for aggregation or perform logics.



How can we manage these cross-services queries ?



- **Direct HTTP Communication**

Not a good solution that makes coupling each microservices and loose power of microservices independency.

- **Async Communication**

The best practice is reducing inter-service communication as much as possible and use async communication. Can't reduce these internal communications due to customer requirement.

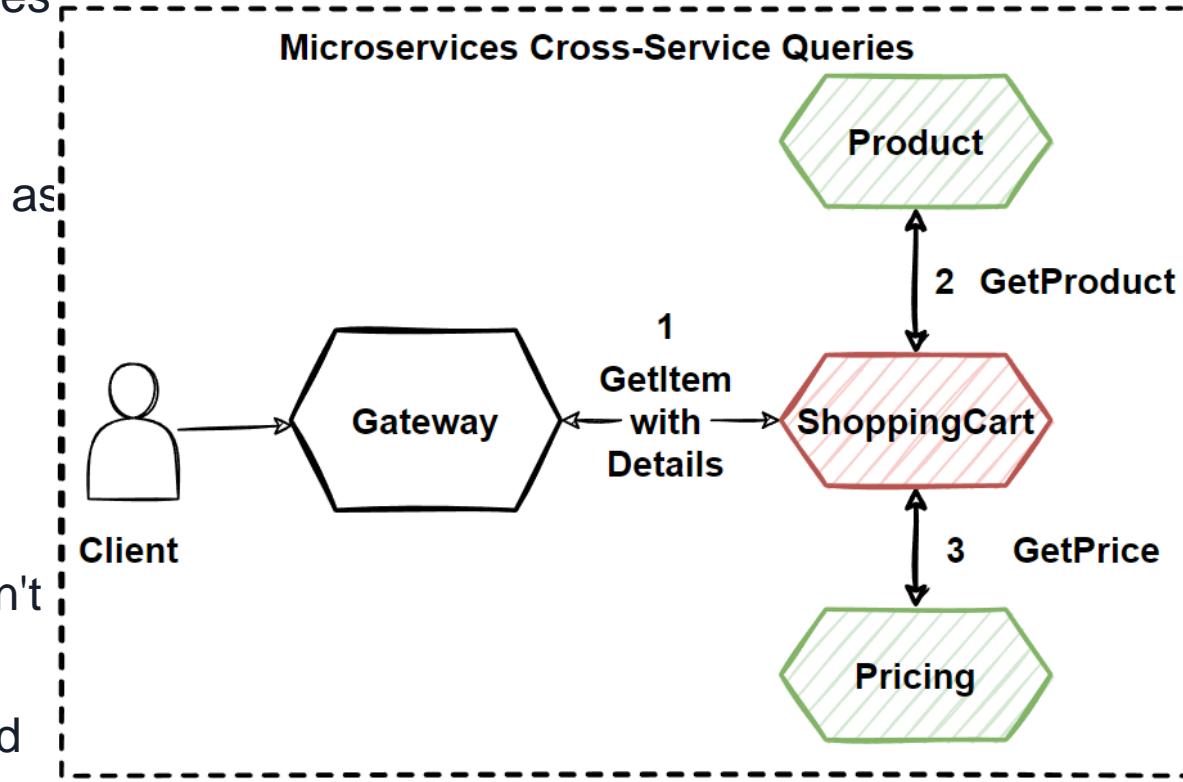
- Client **send query request to internal microservices** to **accumulate some data**.

- Those query request wait **immediate response** so we can't proceed with async communication.

- **Transient errors, Network congestion** or any overloaded microservice can result in long-running and failed operations.

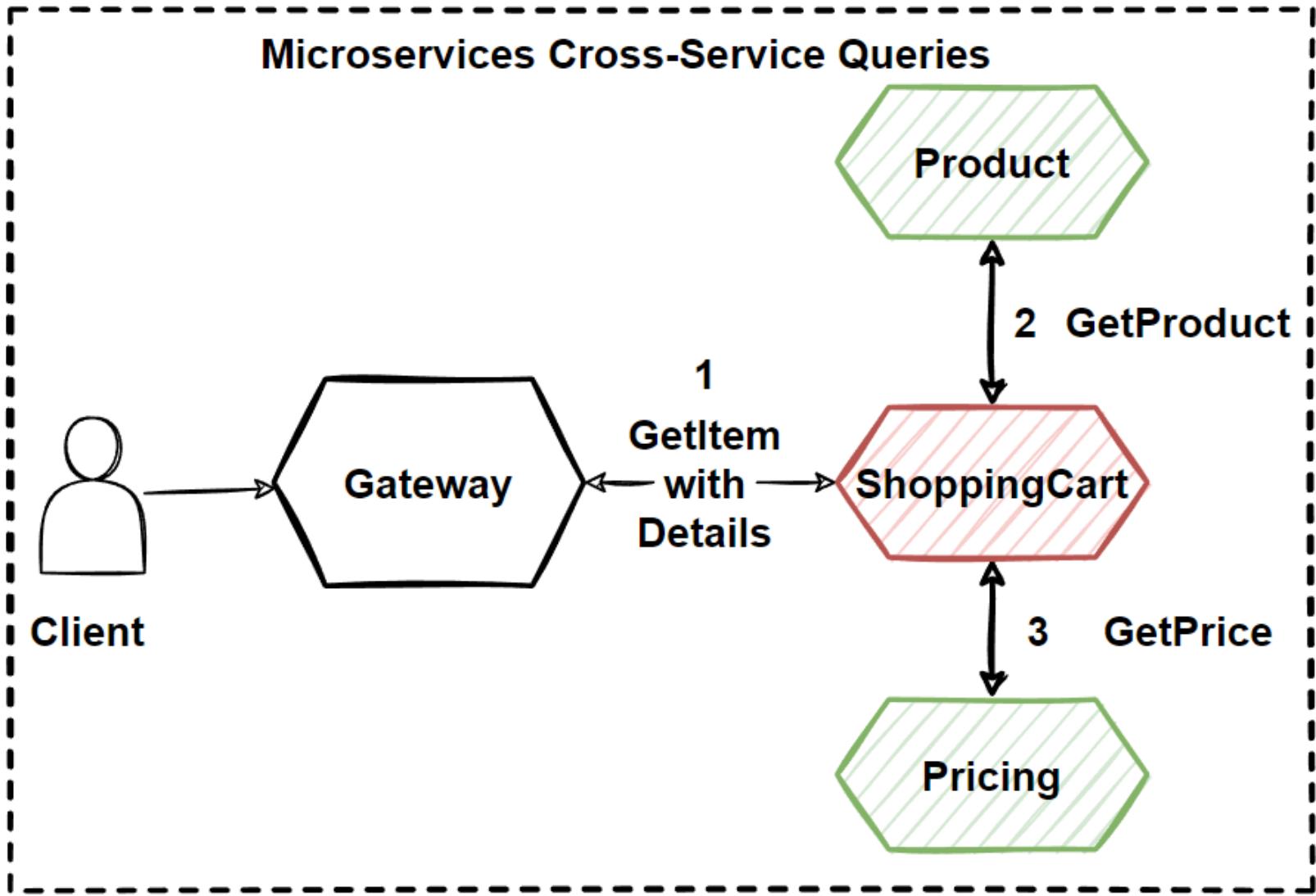
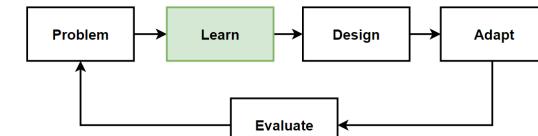
- **Materialized View Pattern**

Reduce inter-service communication and provide sync response.



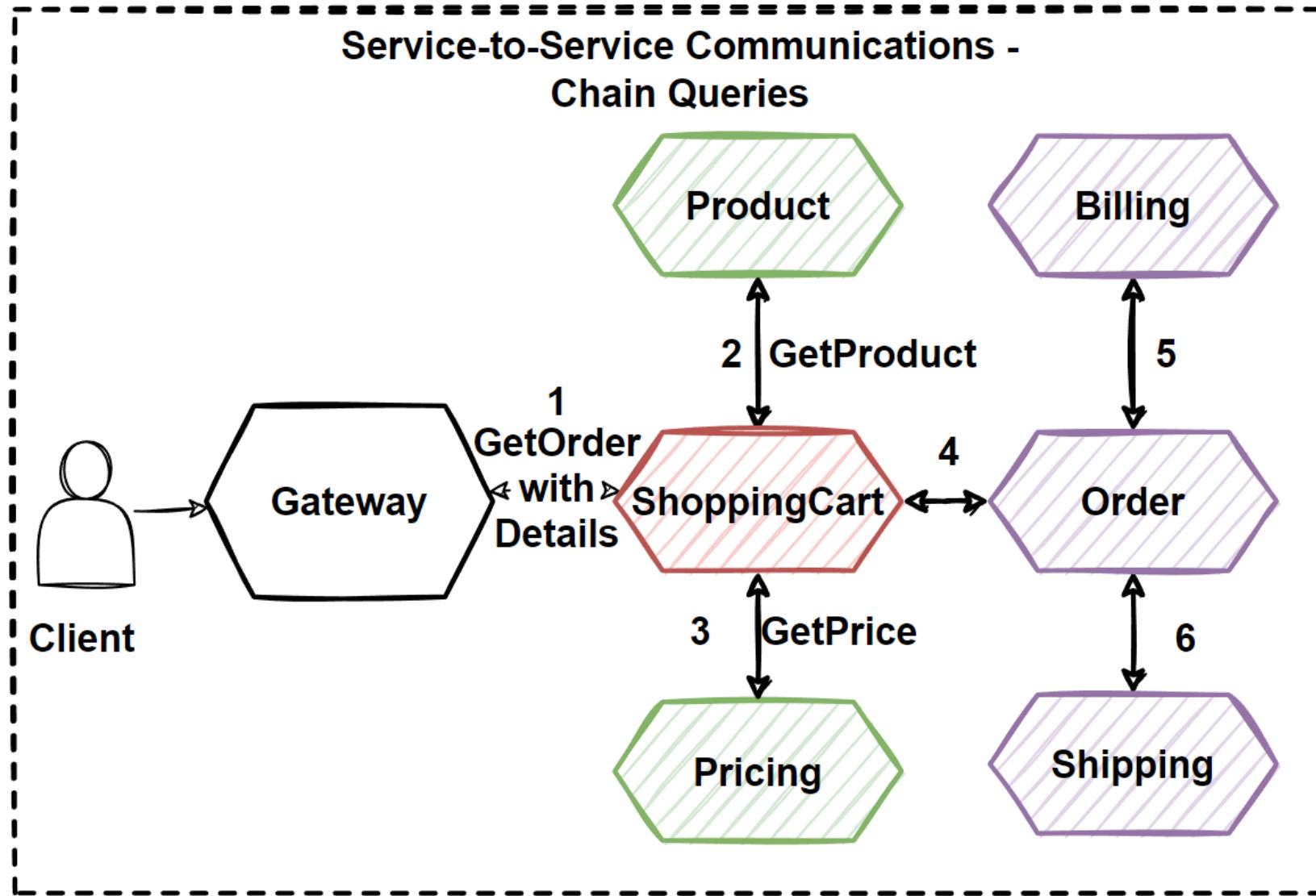
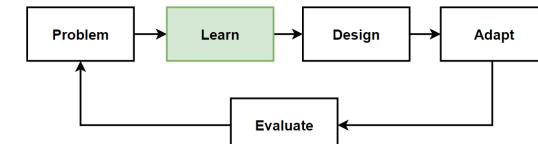
Microservices Cross-Service Queries

Example Use Case – Get Item with Details



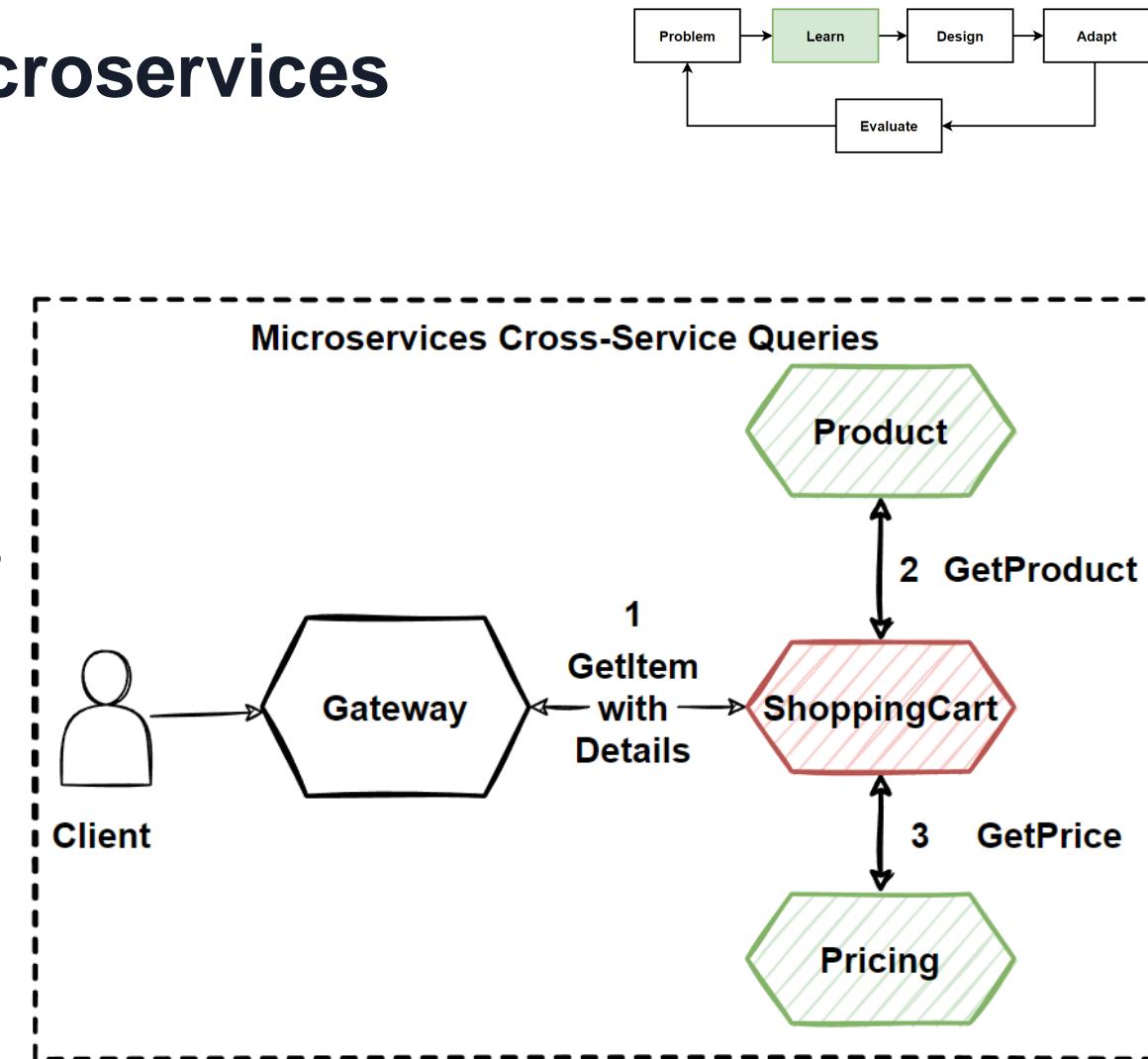
Microservices Cross-Service Queries

Chain Queries – Get Order with All Details

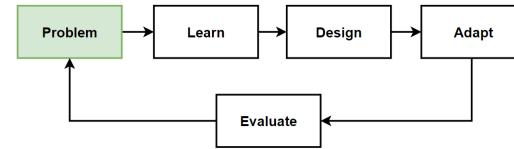


Cross-Service Query Solutions in Microservices

- Perform **queries across microservices** database level in data stores.
- **API Gateway patterns**, we can solve cross-queries with applying Service Aggregator Pattern with create new Aggregator microservices.
- How to **solve problem** with **database level** in data stores?
- How **get item info** from the **user's shopping cart** with get data **Catalog** and **Pricing** microservice ?
- Don't want to interact with **direct HTTP calls** to get data from other catalog and pricing microservices.
- **Direct synchronous HTTP calls makes couple** microservices together and reduce independency of microservices and makes chatty communications.



Problem: Cross-Service Queries with Sync Response, Decouple Way and Low Latency



Problems

- Cross-Service Queries with Complex JOINS
- Return Sync Response with low latency
- Provide loosely coupling with decouple services
- Reduce inter-service communication

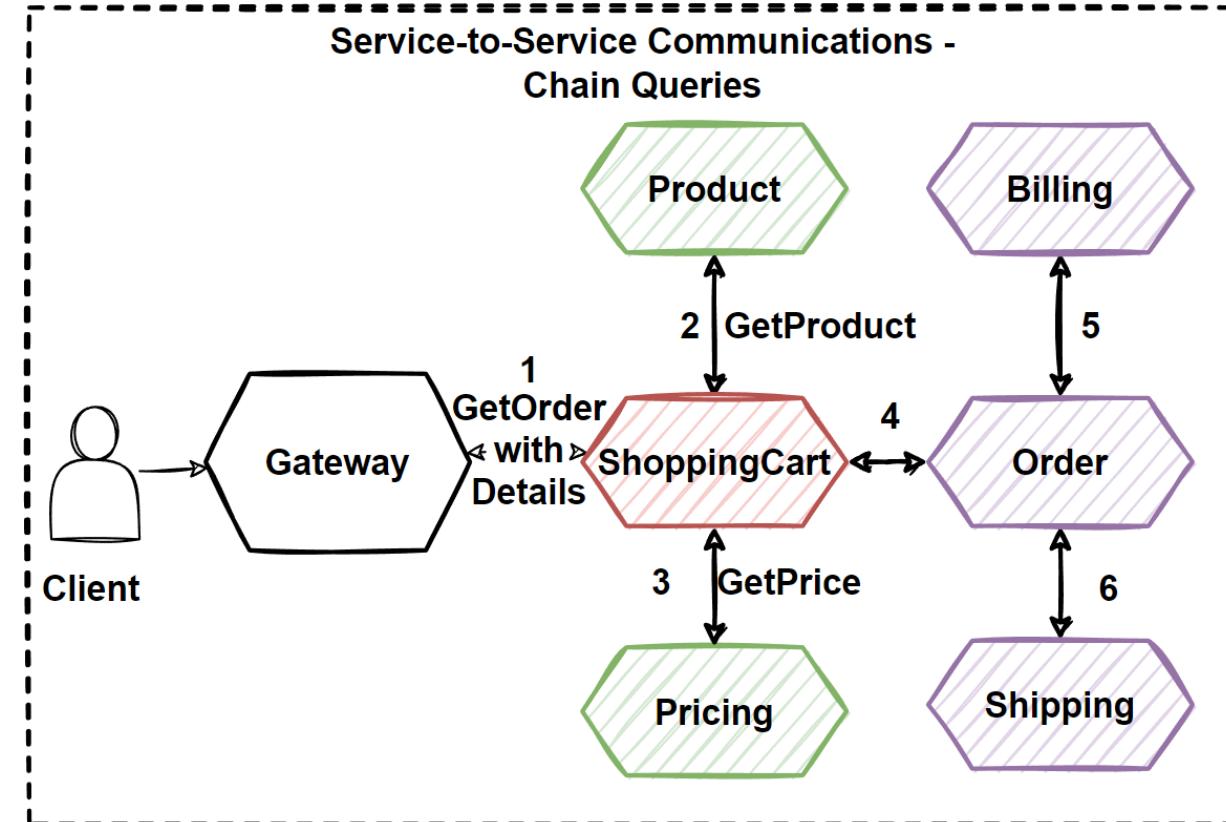
SOLVE ALL PROBLEMS AT THE SAME TIME !

Considerations

- Sync Communication: Use Service Aggregator Pattern but it increase coupling and latency.
- Async Communication: Provide decoupling but query request are waiting immediate response.

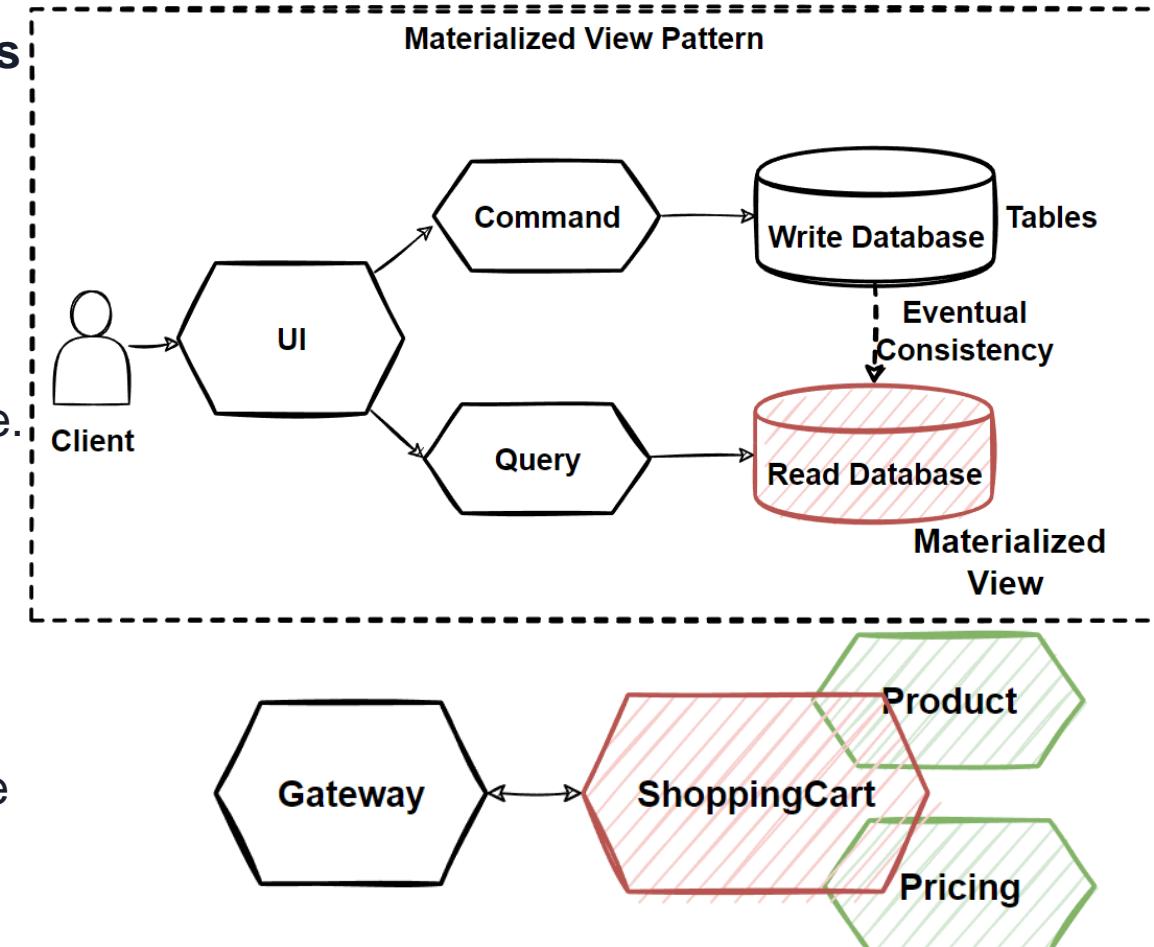
Solutions

- Materialized View Pattern
- CQRS Design Pattern



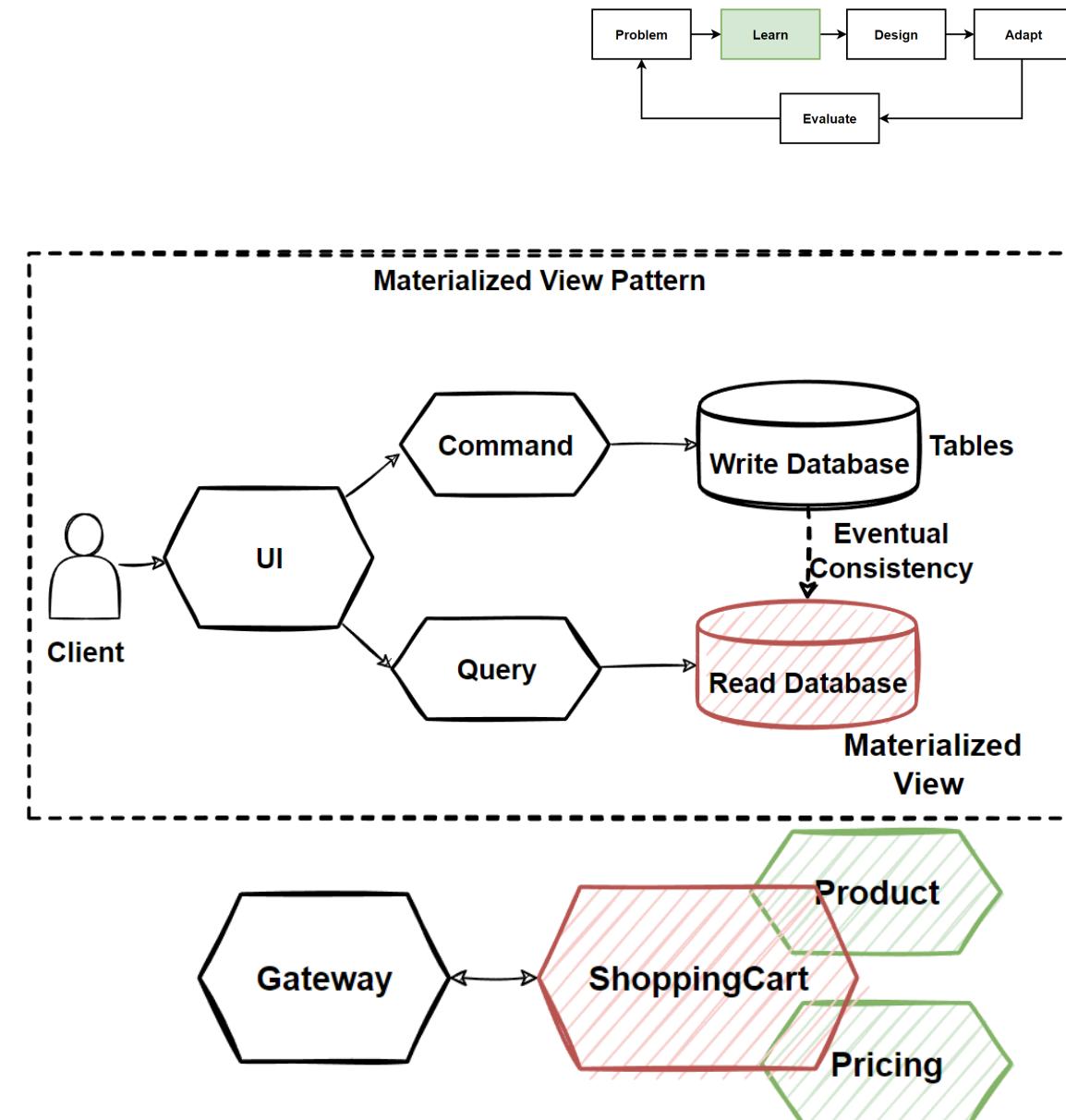
Materialized View Pattern

- Why do we need **Materialized View Pattern** ?
- Mostly we design our systems with **focusing on databases** and **tables** for managing **data size** and **data integrity**.
- **How the data is stored** ? Instead, **how data is read** ?
- **Negative effect on queries** on our system.
- **Materialized View Pattern** is **store its own local** and **denormalized copy of data** in the microservice's database.
- Shopping cart service should have table **contains a denormalized copy of the data** from the product and pricing microservices.
- **Eliminates** the need for **expensive cross-service calls**.
- **Reduces coupling** and **improves reliability** and response time with reducing latency.
- Can **execute the entire operation** with a single process.



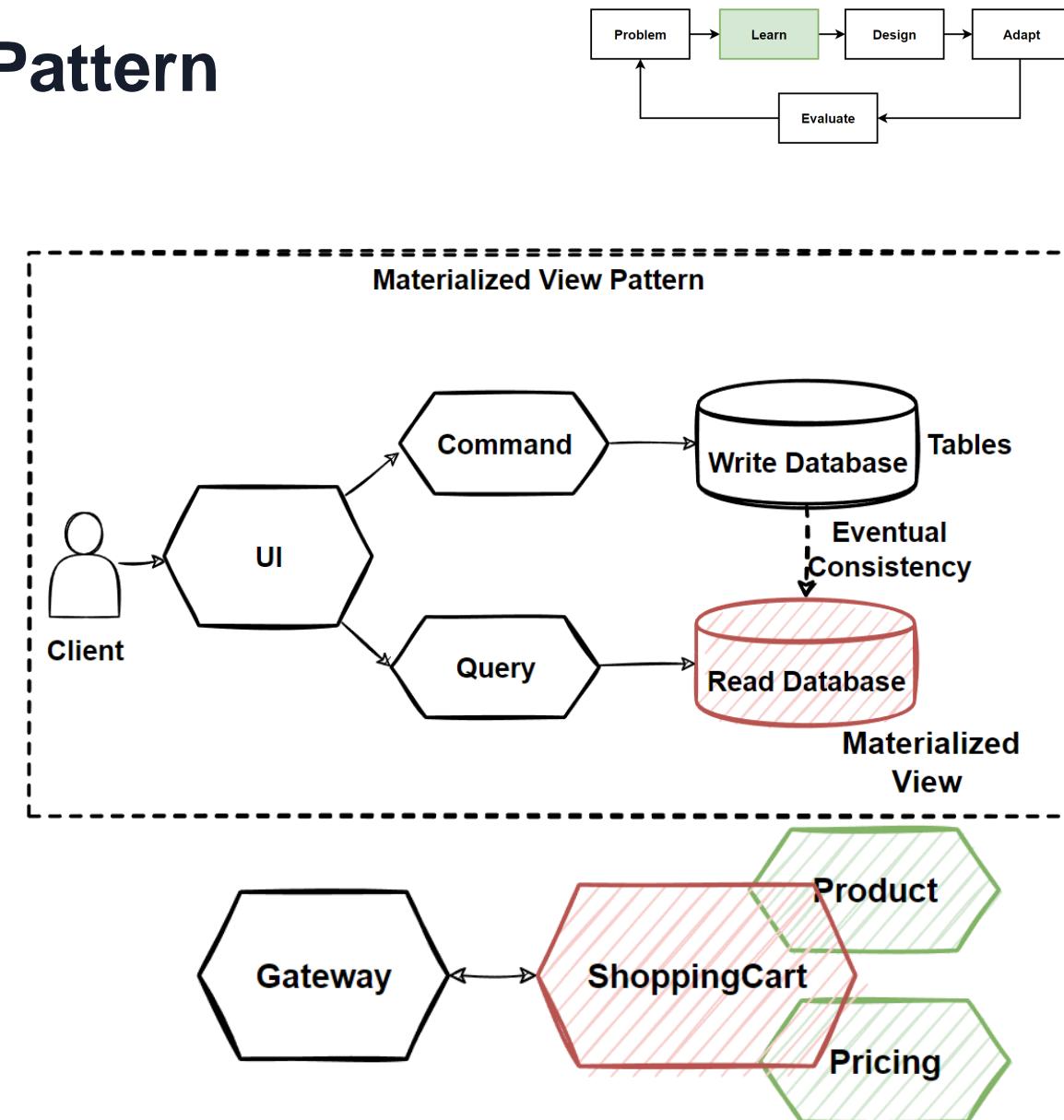
Materialized View Pattern - 2

- Also called this local copy of data as a **Read Model**.
- **Instead of querying** the Product Catalog and Pricing services, it maintains **its own local copy** of that data.
- Makes SC microservice is **more resilient**.
- **If one of the service is down**, then the **whole operation** could be **block or rollback**.
- With **Materialized View Pattern**, even if the Catalog and Pricing services are down, **Shopping Cart can continue**.
- **Broke the direct dependency** of other microservices and make faster the response time, help efficient querying and improve application performance.
- **Generate pre-populated views of data**, more suitable format for querying and provide good query performance.
- Includes **joining tables** and **combining data entities** and calculated columns and execute transforms.
- **Views are disposable** and can **rebuilt** from the source.

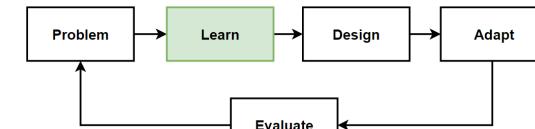


Considerations of Materialized View Pattern

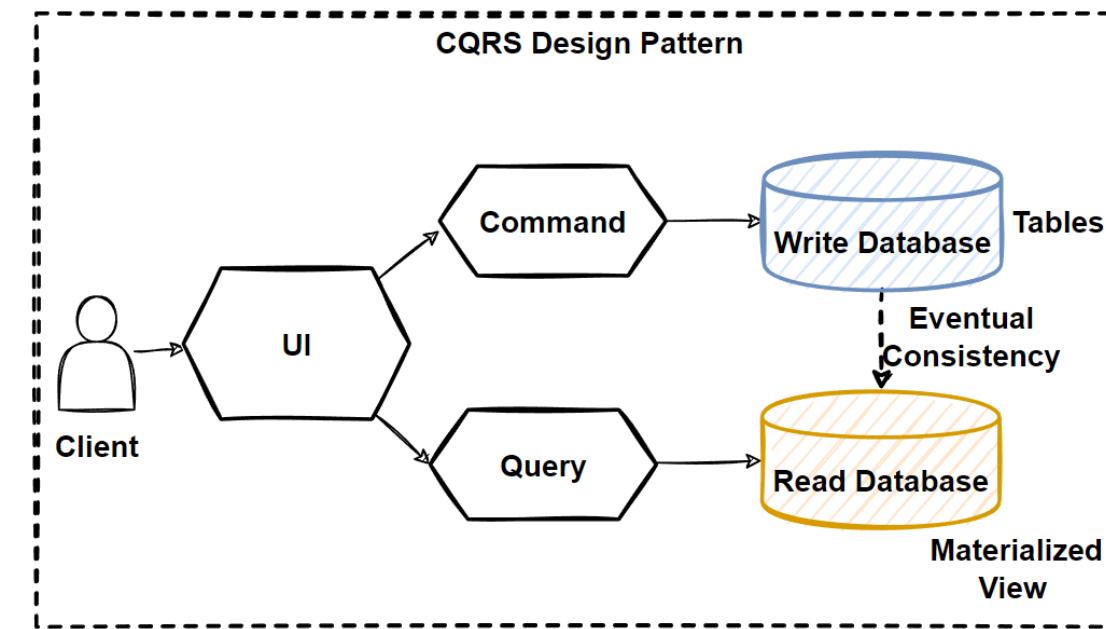
- With applying **Materialized View Pattern**, we have **duplicated data** into our system.
- **Duplicating data is not a anti-pattern**, have **strategically duplicating** our data for microservice communications.
- Only one service can be a data ownership.
- How and when the **denormalized data** will be **updated** ?
- When the **original data changes** it should update into sc microservices.
- **Need to synchronize** the **read models** when the main service of data is **updated**.
- Solve with using **asynchronous messaging** and **publish/subscribe pattern**.
- **Publish an event** and **consumes** from the **subscriber** service to **update** its **denormalized table**.
- Using a **scheduled task**, an **external trigger**, or a manual action to regenerate the table.



CQRS - Command Query Responsibility Segregation

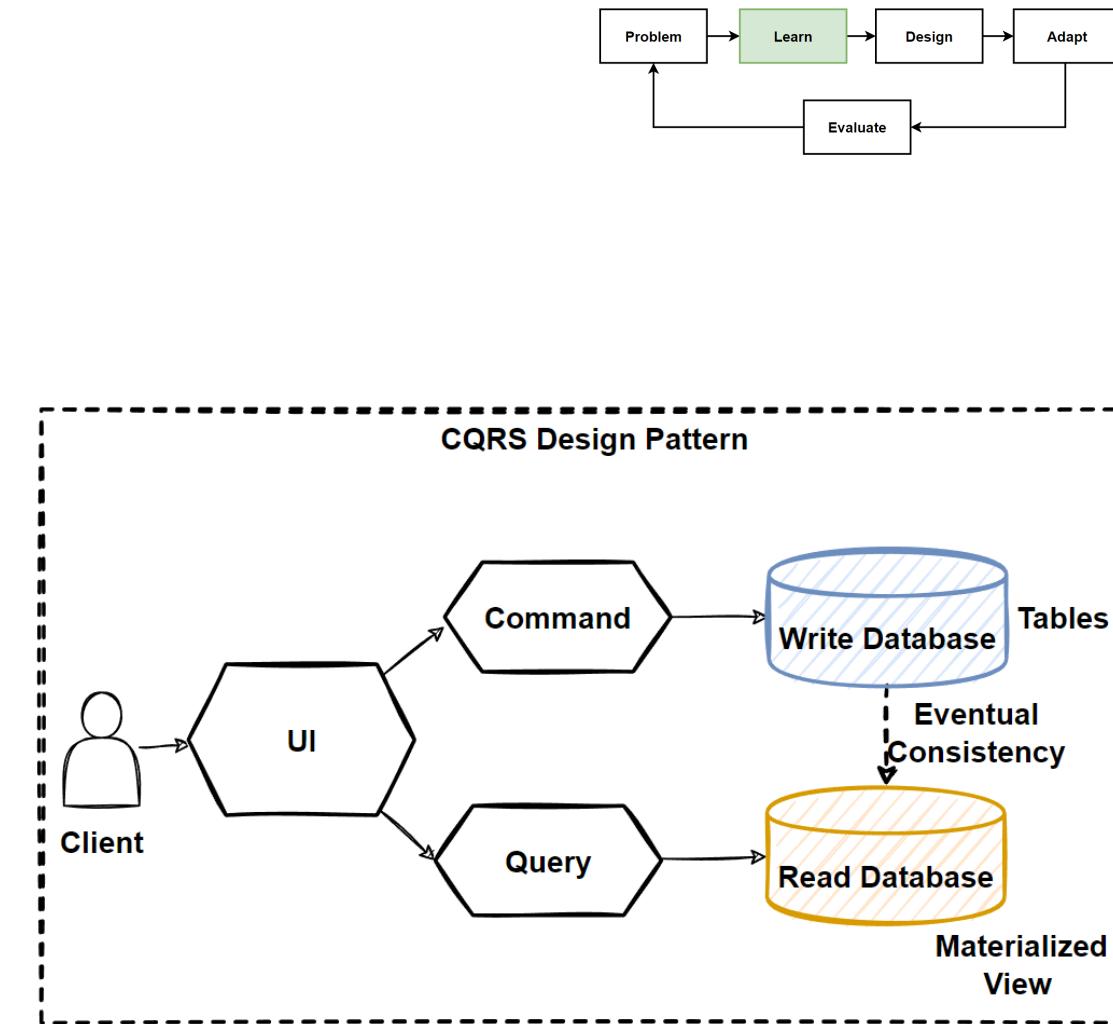


- **CQRS design pattern** in order to avoid **complex queries** to get rid of **inefficient joins**.
- **Separates read and write operations** with separating databases.
- **Commands**: changing the state of data into application.
- **Queries**: handling complex join operations and returning a result and don't change the state of data into application.
- Large-scaled **microservices architectures** needs to manage **high-volume data requirements**.
- **Single database** for services can **cause bottlenecks**.
- Uses both **CQRS** and **Event Sourcing** patterns to improve application performance.
- **CQRS** offers to **separates read and write data** that provide to maximize query performance and scalability.



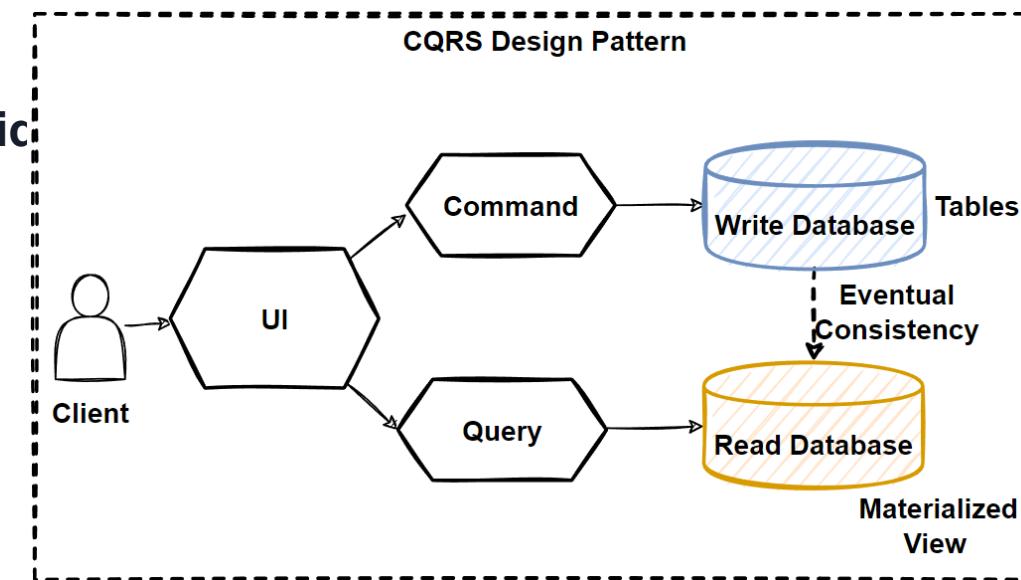
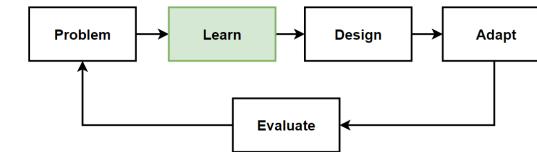
CQRS – Read and Write Operations

- Monolithic has **single database** is both working for **complex join queries**, and also perform **CRUD operations**.
- When **application goes more complex**, this **query** and **CRUD** operations will become **un-manageable situation**.
- Application required some **query** that **needs to join** more than **10 table**, will **lock the database** due to **latency** of **query computation**.
- Performing **CRUD operations** need to make **complex validations** and process **long business logics**, will cause to **lock database** operations.
- **Reading and writing database** has different approaches, define different strategy.
- «**Separation of concerns**» principles: separate reading database and the writing database with 2 database.
 - **Read database** uses No-SQL databases with denormalized data.
 - **Write database** uses Relational databases with fully normalized and supports strong data consistency.



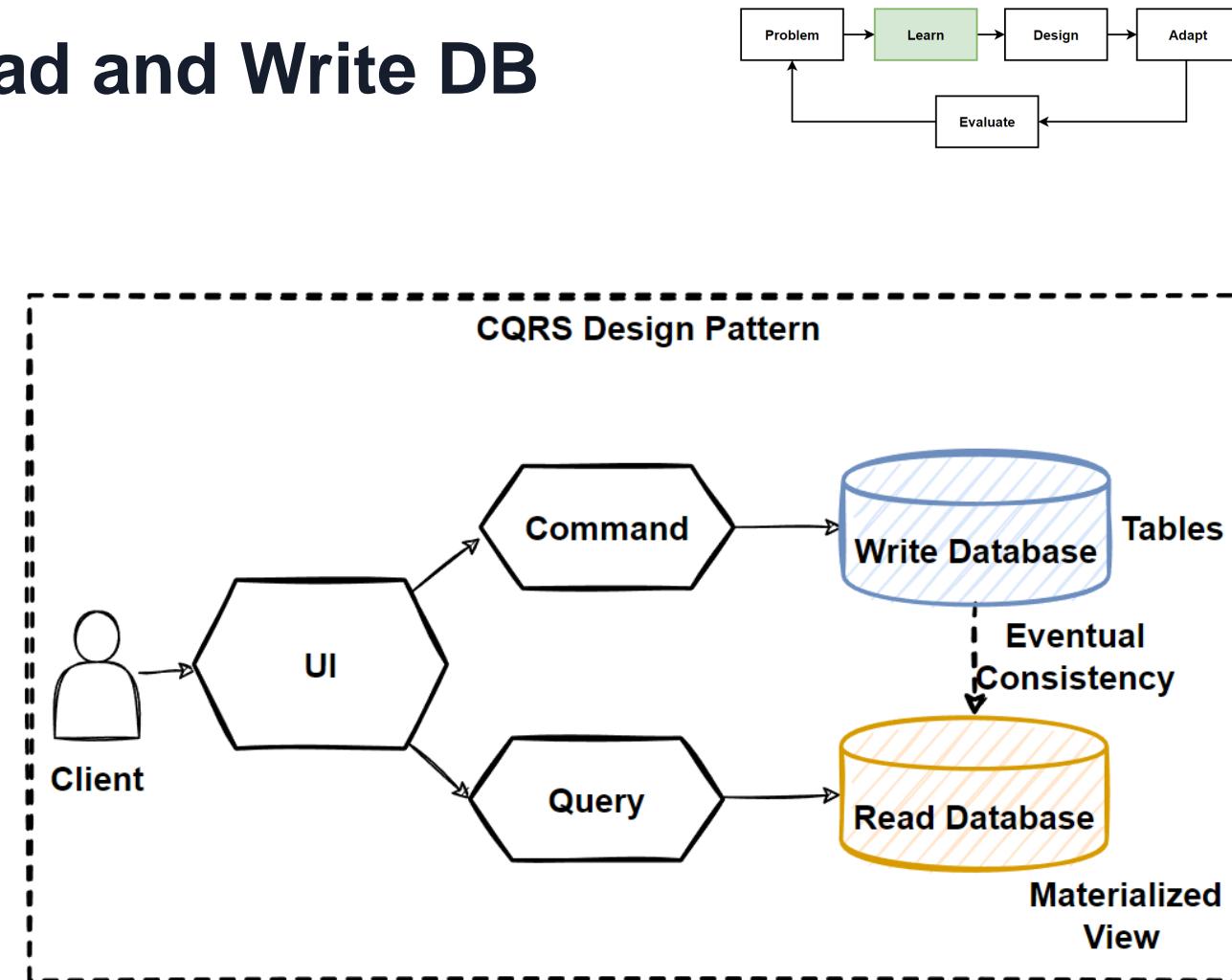
CQRS – Read and Write Operations - 2

- If our **application** is mostly **reading use cases** and not writing so much, it is **read-incentive application**.
- **Read and write operations** are **asymmetrical** and has very different performance and scale requirements.
- To improve **query performance**, the read operation perform **queries** from a **highly denormalized materialized views** to avoid expensive repetitive **table joins** and **table locks**.
- **Write operation** which is **command operation**, can perform commands into separate **fully normalized relational database**.
- Supporting **ACID transactions** and **strong data consistency**.
- Commands: **task-based operations** like "add item into shopping cart" or "checkout order".
- **Commands** can be handle with message broker systems that provide to process commands in async way.
- **Queries**: never modify the database, always return the JSON data with DTO objects.



CQRS – Synchronization with Read and Write DB

- Keep sync both **read** and **write databases**.
- Publishes an event that subscribe from **Read Database** and **update the read table** accordingly.
- **Synchronization** handles with **async communication** using message brokers.
- This creates **Eventual Consistency principle**.
- The **Read database** **eventually synchronizes** from the **Write database**.
- **Some lag** in the process due to **async communication** with message brokers that applies publish/subscribe pattern.
- Welcome to **Eventual consistency**.



Benefits of CQRS

- **Scalability**

When we separate Read and Write databases, we can also scale these separate databases independently. Read databases follows denormalized data to perform complex join queries.

- If application is read-incentive application, we can scale read database more than write database.

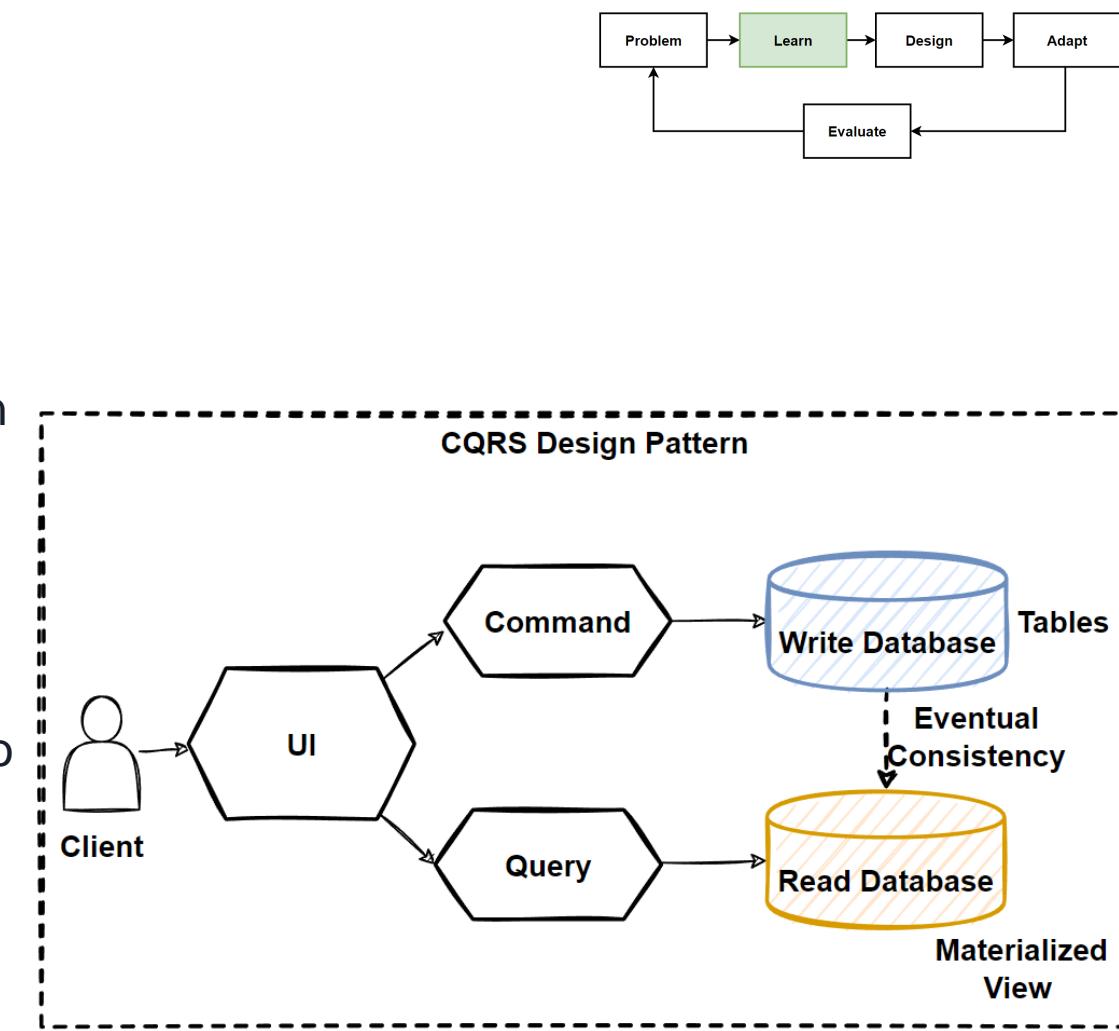
- **Query Performance**

The read database includes denormalized data that reduce to complex and long-running join queries. Complex business logic goes into the write database. Improves application performance for all aspects.

- **Maintability and Flexibility**

Flexibility of system that is better evolve over time and not affected to update commands and schema changes by separating read and write concerns into different databases.

- Better implemented if we physically separate the read and write databases.



Drawbacks of CQRS

- **Complexity**

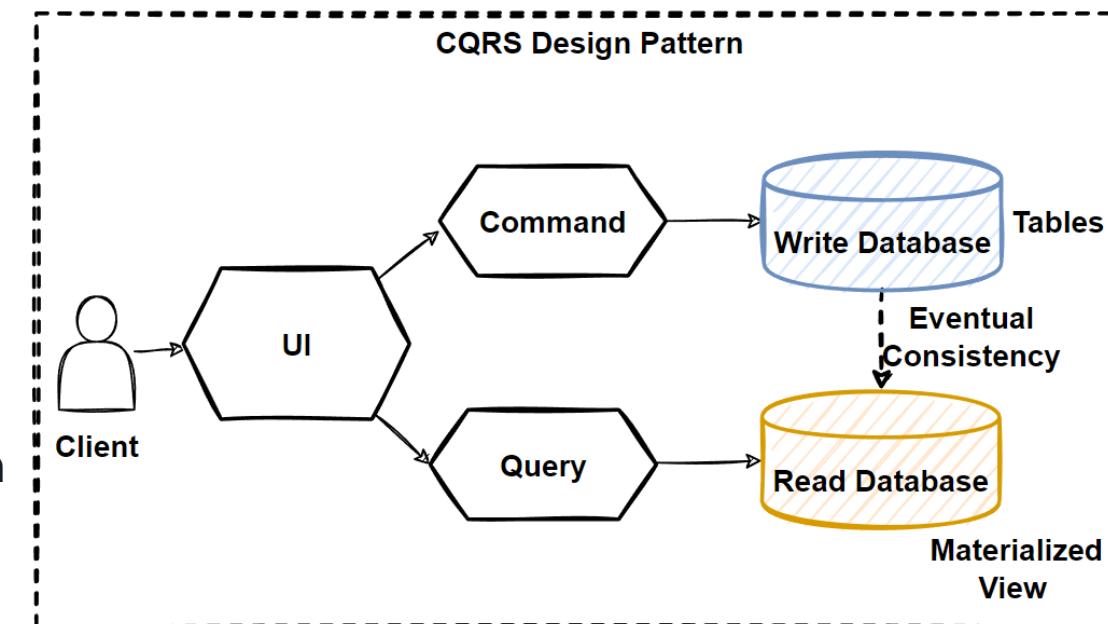
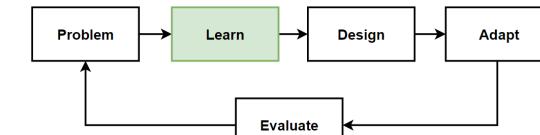
CQRS makes your system more complex design.

Strategically choose where we use and how we can separate read and write database.

- **Eventual Consistency**

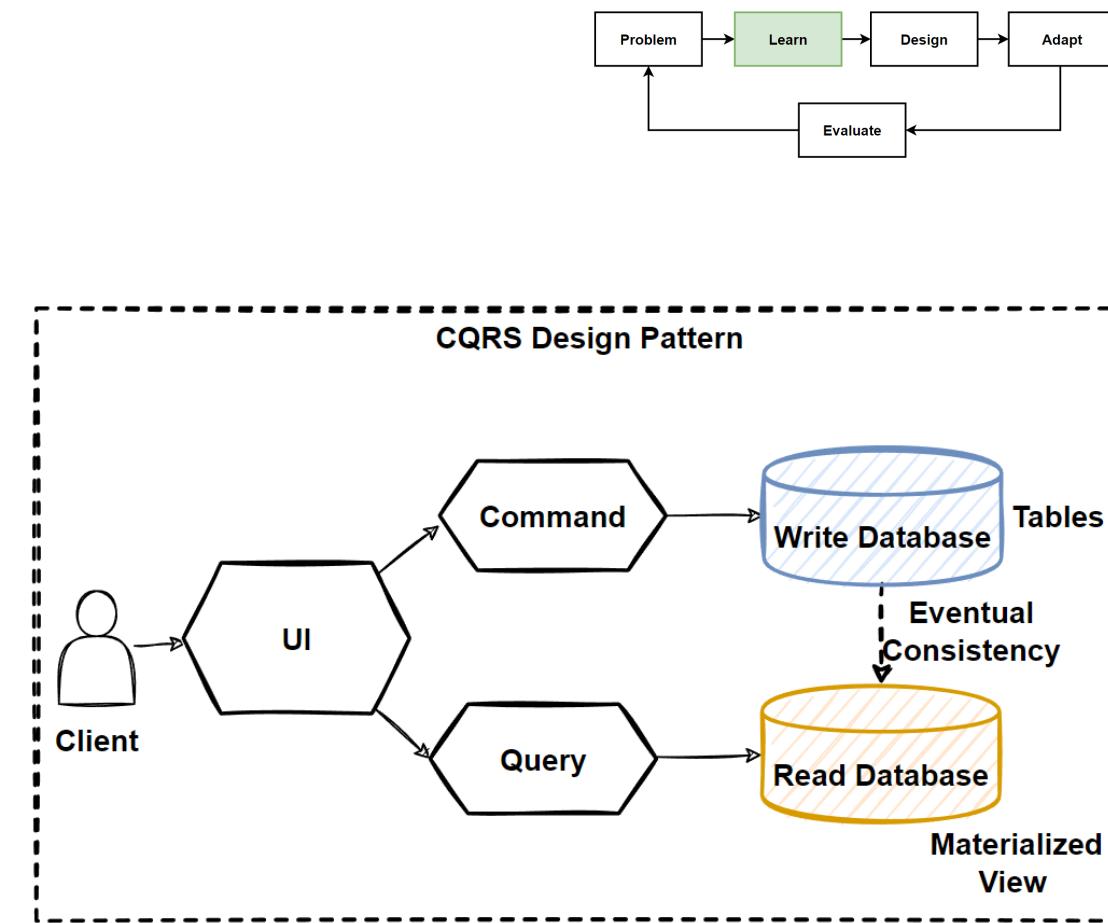
The read data may be stay old and not-updated for a particular time. So the client could see old data even write database updated, it will take some time to update read data due to publish/subscribe mechanism.

- We should **embrace the Eventual Consistency** when using **CQRS**, if your application **required strong consistency** than **CQRS is not good to apply**.



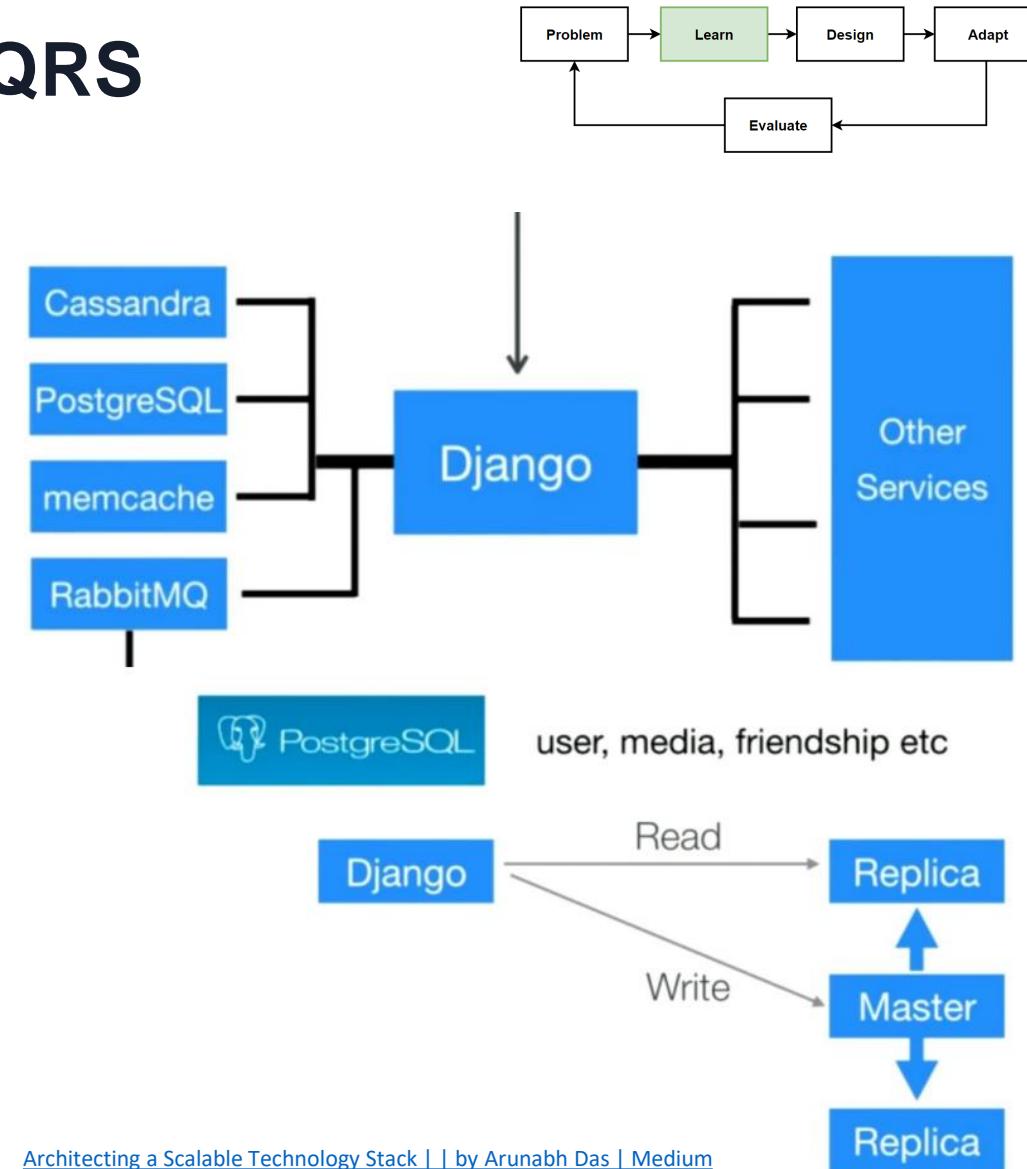
Best Practices for CQRS

- **Best practices to separate read and write database with 2 database physically.**
- **Read-intensive** that means **reading more than writing**, can define **custom data schema** to optimized for queries.
- **Materialized View Pattern** is good example to implement reading databases.
- **Avoid complex joins** and mappings with **pre-defined fine-grained data for query operations**.
- **Use different database** for reading and writing database types.
- Using **No-SQL document database** for reading and using **Relational database** for **CRUD** operations.



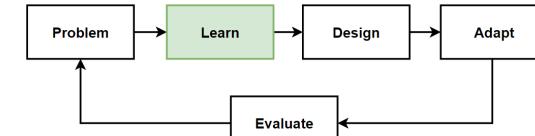
Instagram Database Architecture with CQRS

- 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.

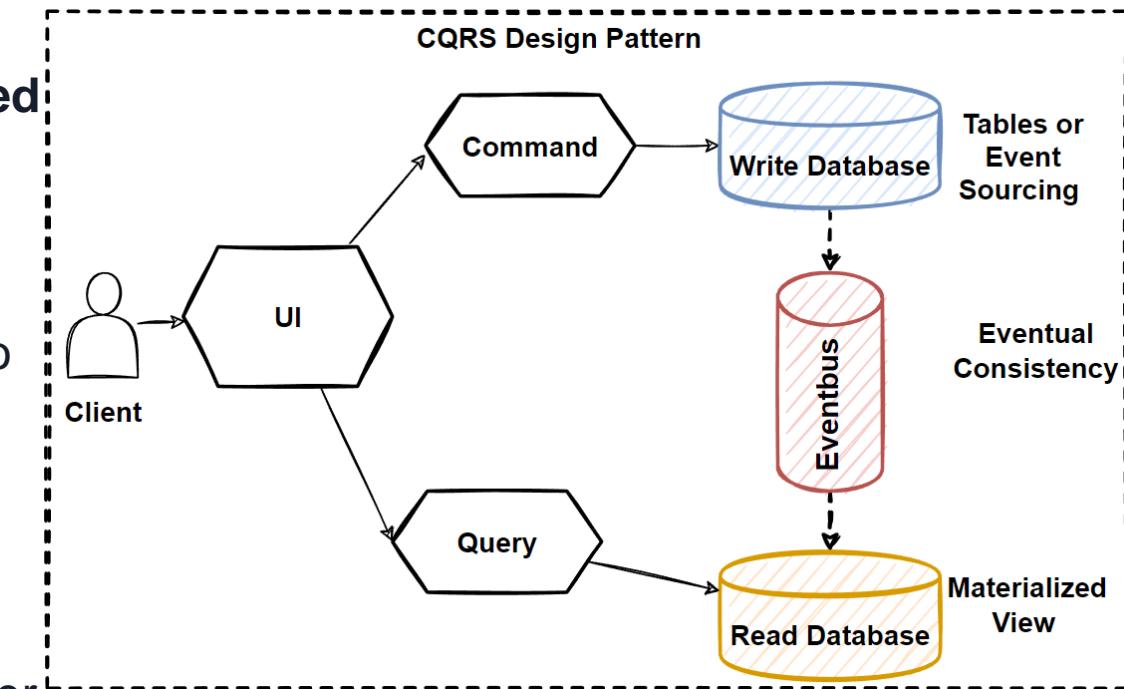


Architecting a Scalable Technology Stack | by Arunabh Das | Medium

How to Sync Read and Write Databases in CQRS ?

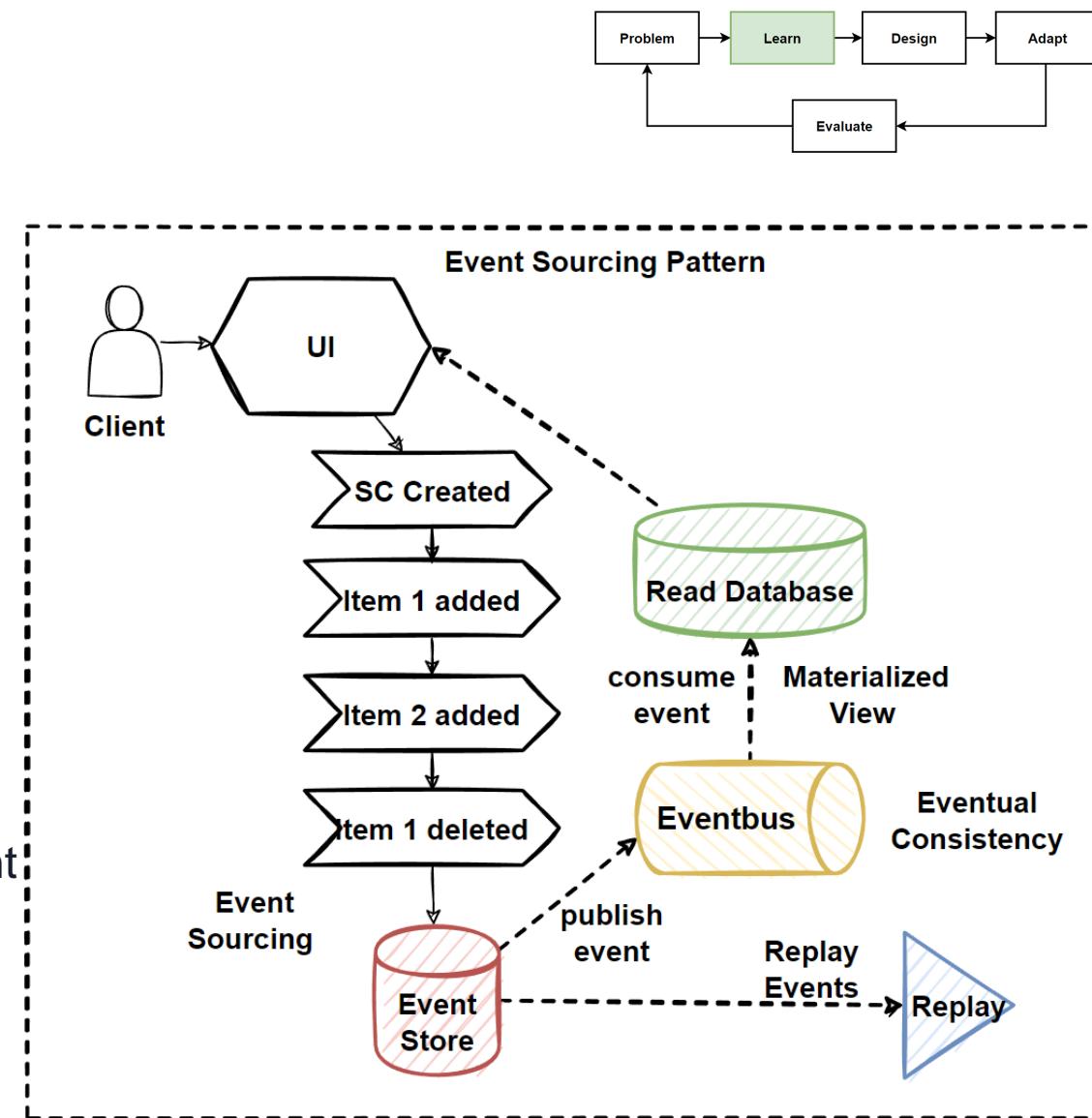


- **Event-Driven Architecture**, when something update in write database, **publish an update event** with using message broker systems, **consume by the read database** and **sync data** according to latest changes.
- Creates a **consistency issue**, the **data** would not be **reflected immediately** due to async communication with message brokers.
- «**Eventual Consistency**» The read database **eventually synchronizes** with the write database, and take some time to update read database in the async process.
- Take read database from replicas of write database. applying **Materialized View Pattern** can significantly increase query performance.
- **Event Sourcing Pattern** is the first pattern we should consider to use with CQRS.
- **CQRS** is using with "Event Sourcing Pattern" in Event-Driven Architectures.



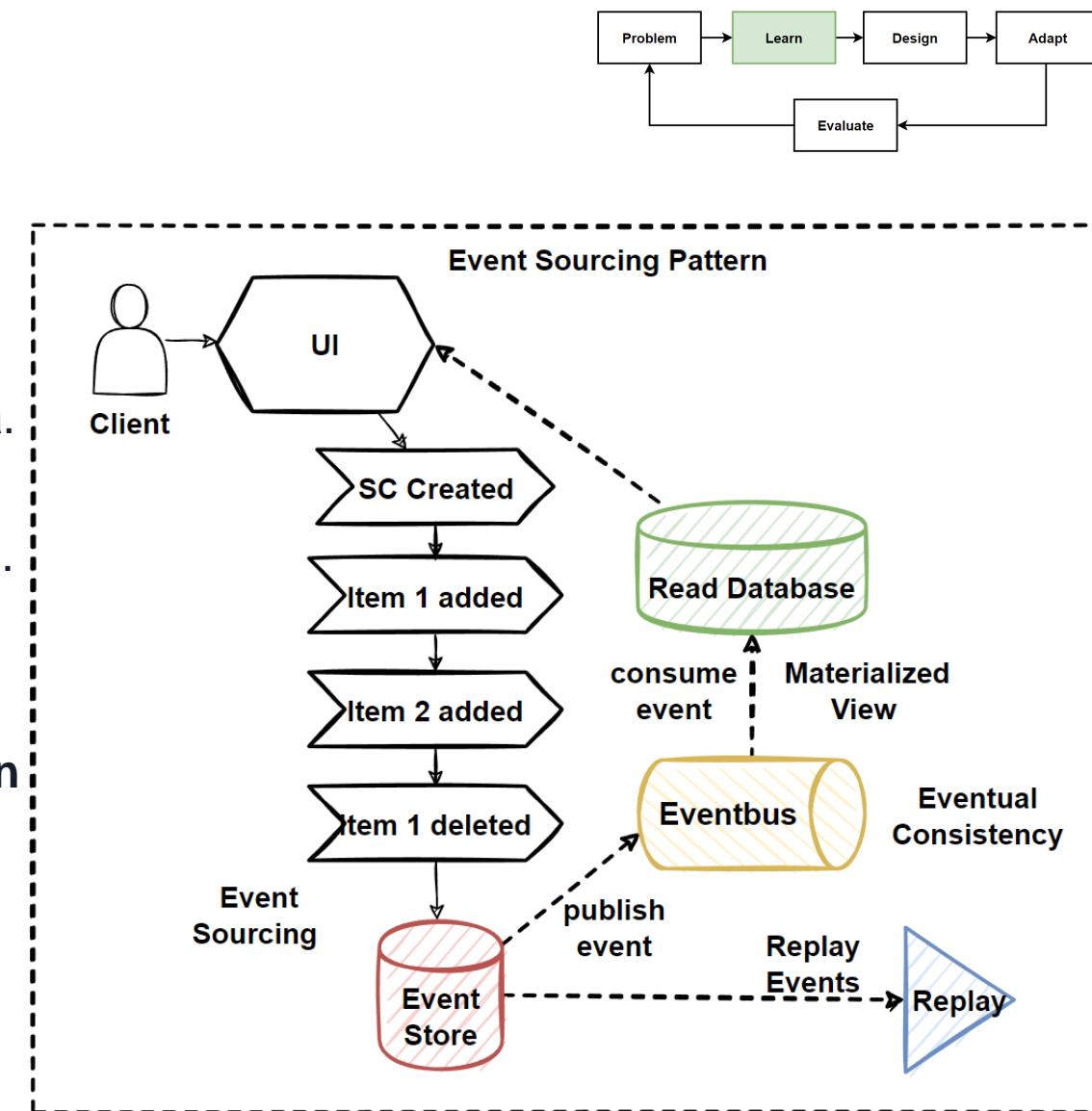
Event Sourcing Pattern

- Most applications **saves data** into databases with **the current state of the entity**. I.e. user change the email address table, email field updated with the latest updated one. Always know the **latest status** of the data.
- In large-scaled architectures, **frequent update database operations** can **negatively impact database performance, responsiveness, and limits of scalability**.
- **Event Sourcing pattern** offers to persist each action that affects to data into **Event Store database**. And call all these **action** as a **event**.
- **Instead of saving latest status** of data into database, Event Sourcing pattern offers to **save all events into database** with **sequential ordered** of data events.
- This **events database** called **Event Store**.



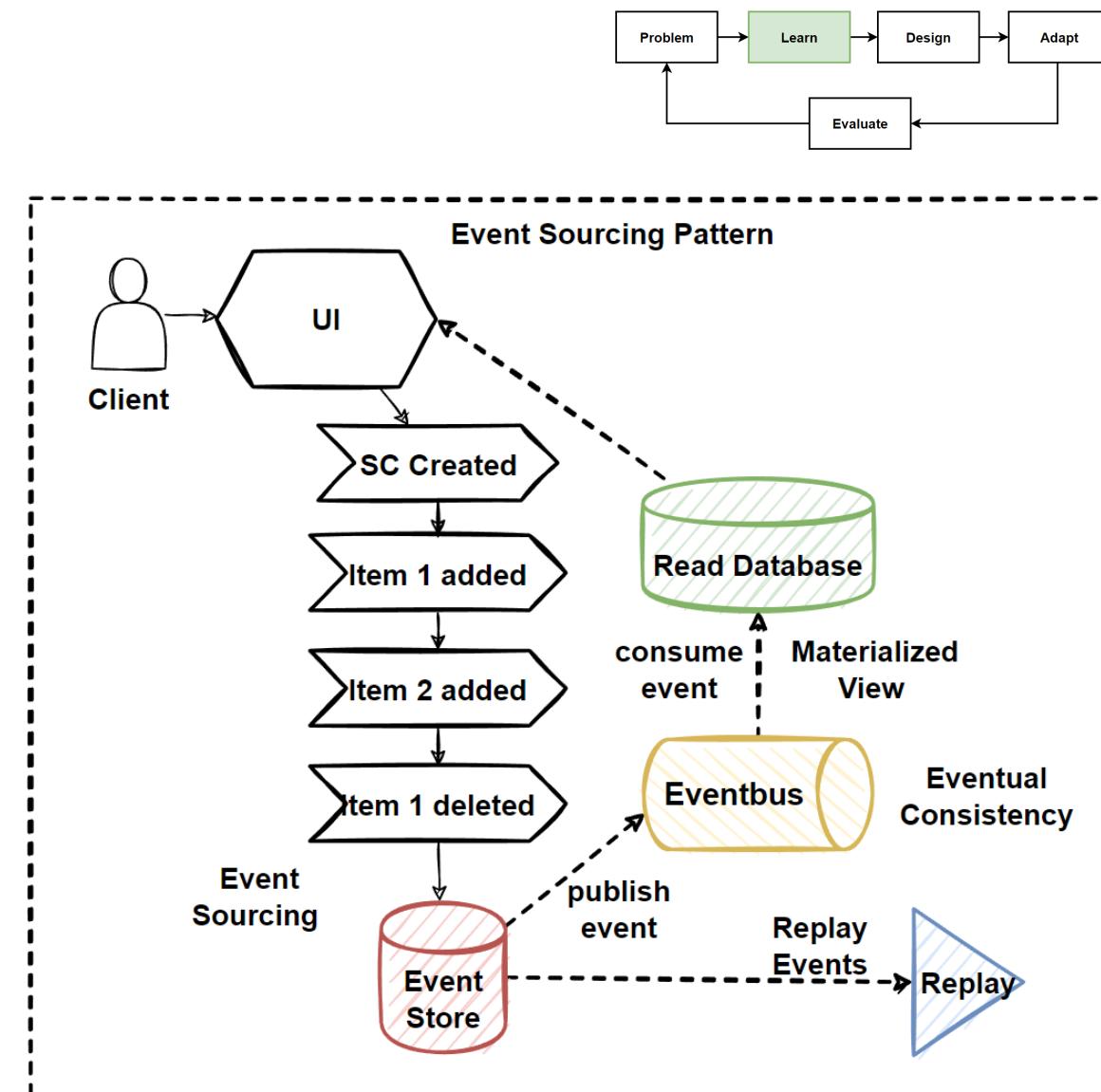
Event Sourcing Pattern - 2

- Instead of overriding the data into table, It **create a new record** for each change to data, and it becomes **sequential list** of past events.
- Event Store database become the **source-of-truth** of data.
- **Sequential event list** using for generating **Materialized Views** that represents **final state** of data to perform queries.
- Event Store convert to read database with following the **Materialized Views Pattern**.
- Convert operation can handle by **publish/subscribe pattern** with **publish event** with message broker systems.
- Event list gives ability to **replay events** at given certain timestamp.
- It is able to **re-build latest status** of data with **replaying events**.

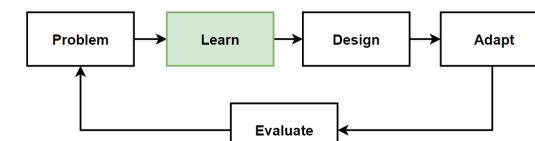
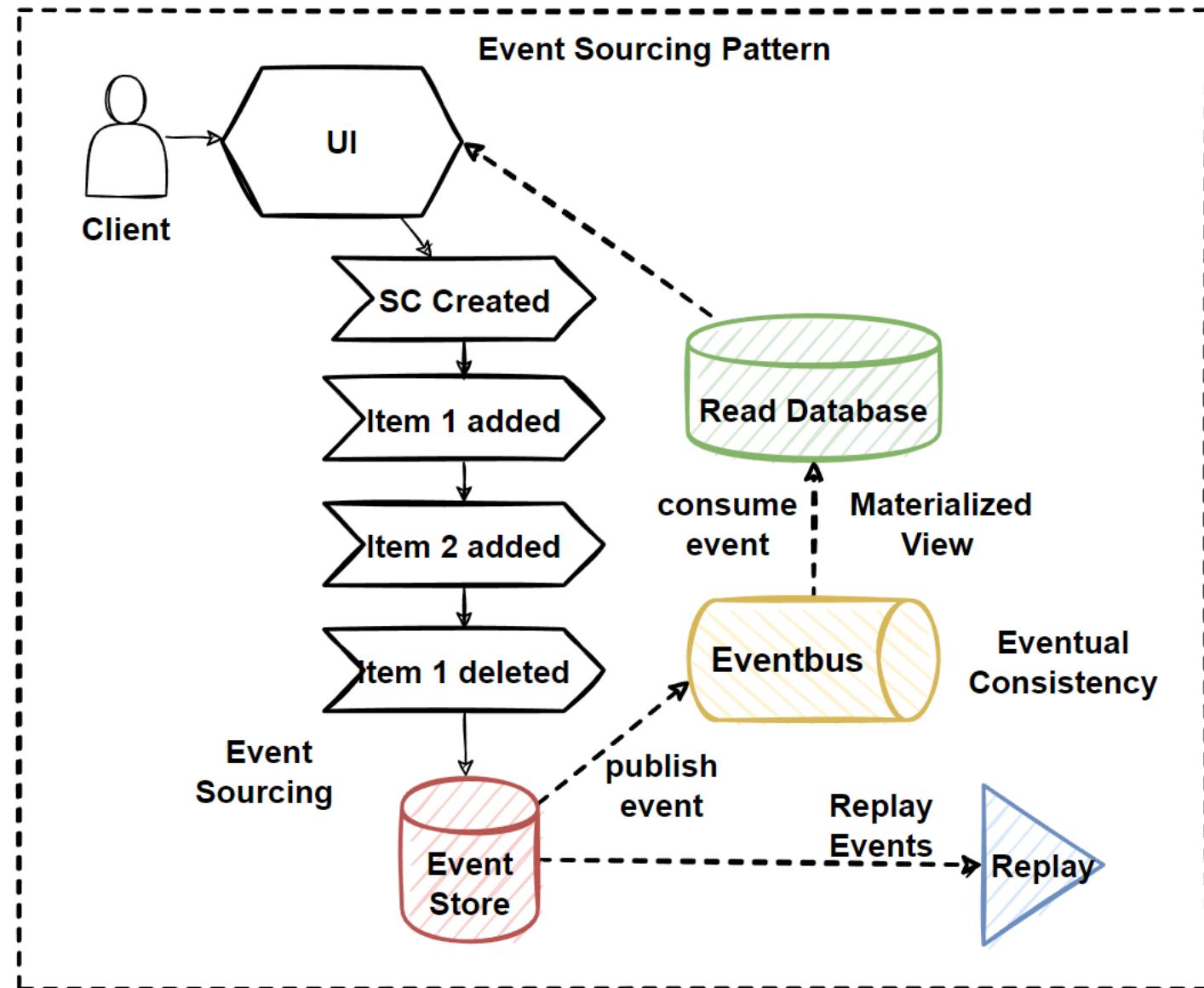


CQRS with Event Sourcing Pattern

- CQRS pattern is mostly using with the Event Sourcing pattern.
- Store events into the write database; source-of-truth events database.
- Read database of CQRS pattern provides materialized views of the data with denormalized tables.
- Materialized views read database consumes events from write database convert them into denormalized views.
- The writing database is never save status of data only events actions are stored.
- Store history of data and able to reply any point of time in order to re-generate status of data.
- System can increased query performance and scale databases independently.

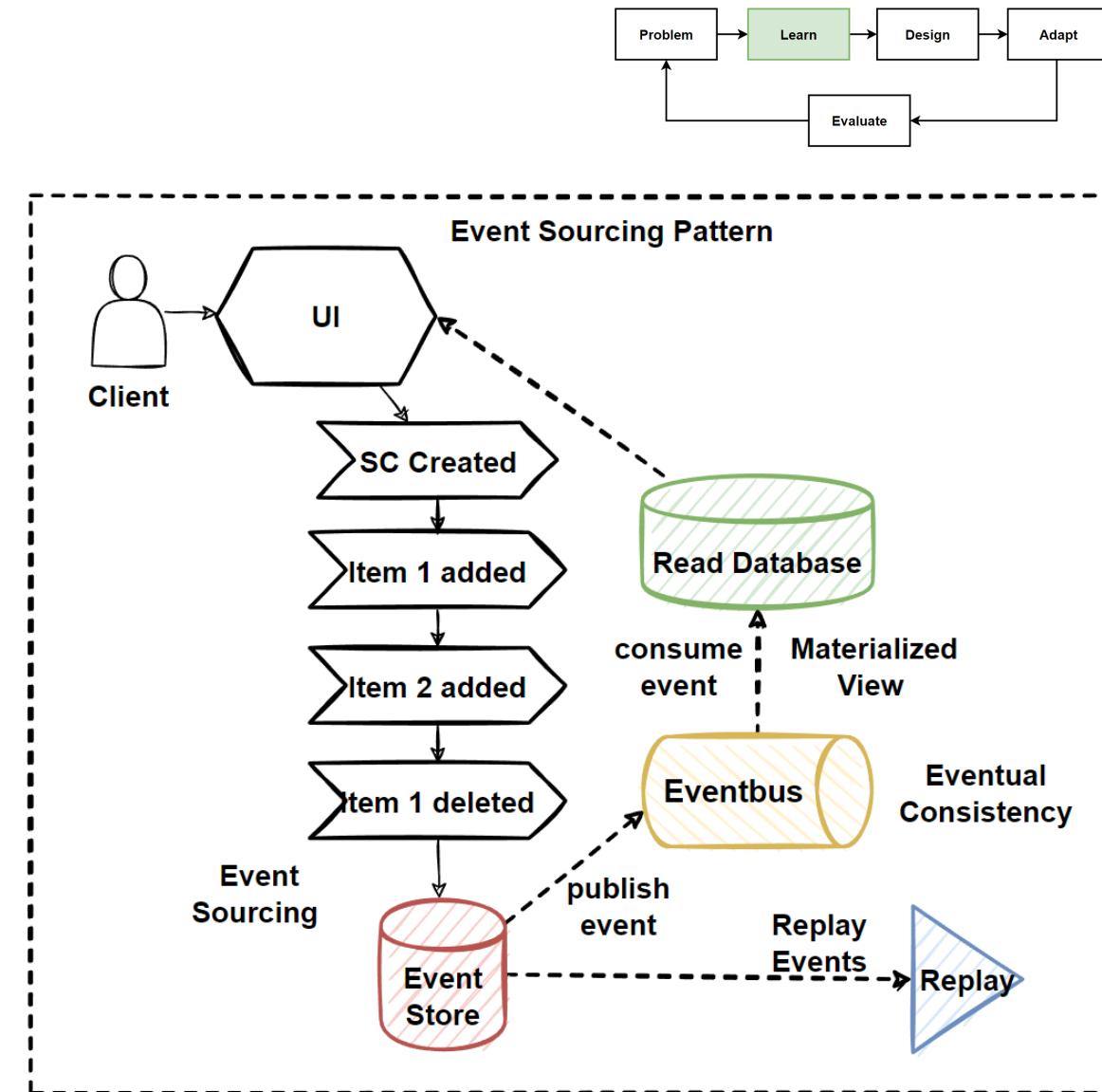


CQRS with Event Sourcing Pattern

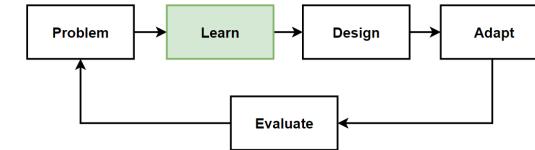


Eventual Consistency Principle

- CQRS with Event Sourcing Pattern leads **Eventual Consistency**.
- **Eventual Consistency** is especially used for systems that prefer **high availability** to **strong consistency**.
- The system will **become consistent after a certain time**.
- We called this latency is a **Eventual Consistency Principle** and offers to be **consistent after a certain time**.
- There are **2 type of "Consistency Level"**:
- **Strict Consistency**: When we save data, the data should affect and seen immediately for every client.
- **Eventual Consistency**: When we write any data, it will take some time for clients reading the data.



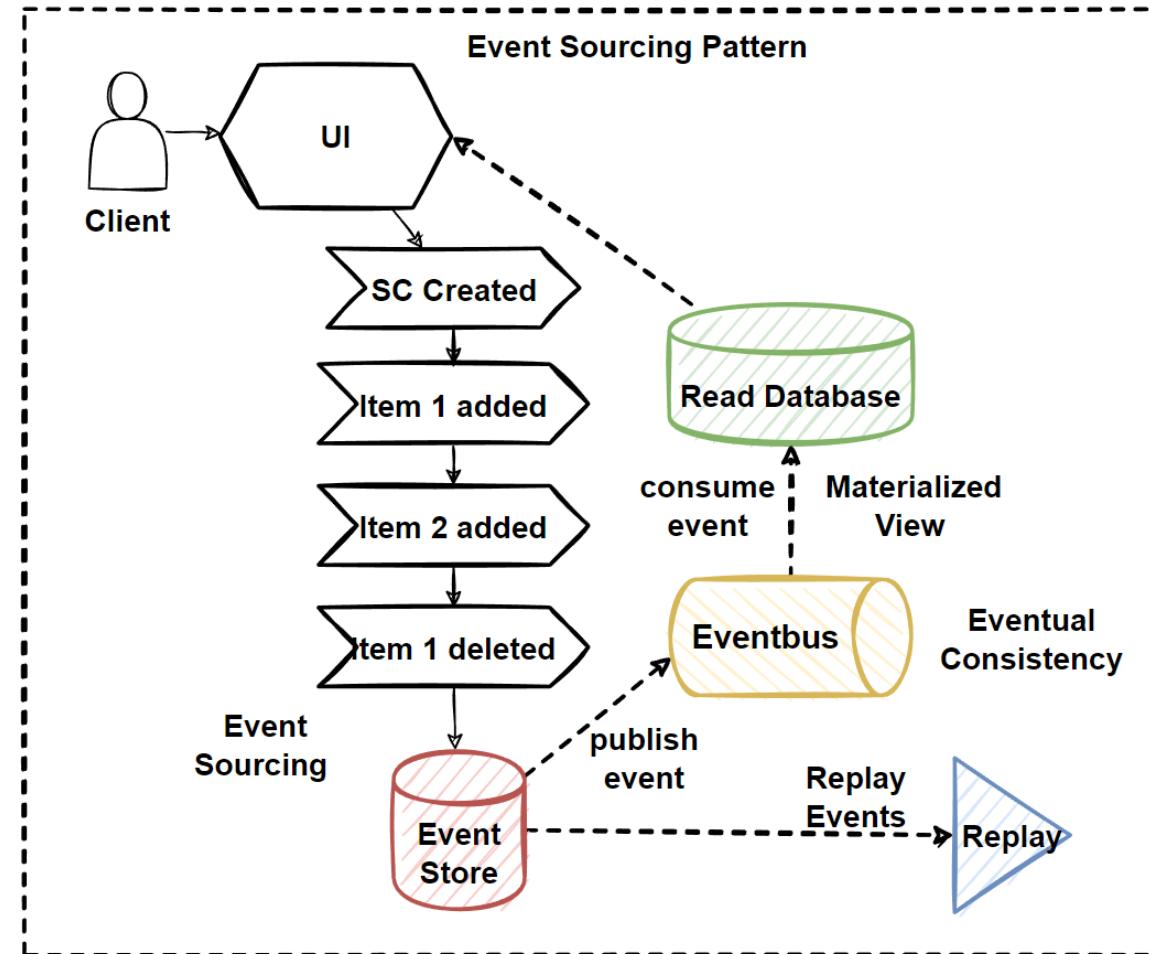
Eventual Consistency Principle - 2



CQRS Design Pattern and Event Sourcing patterns:

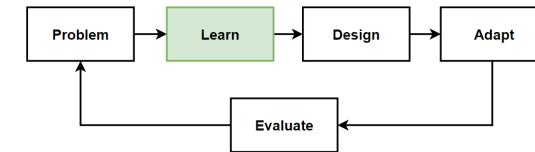
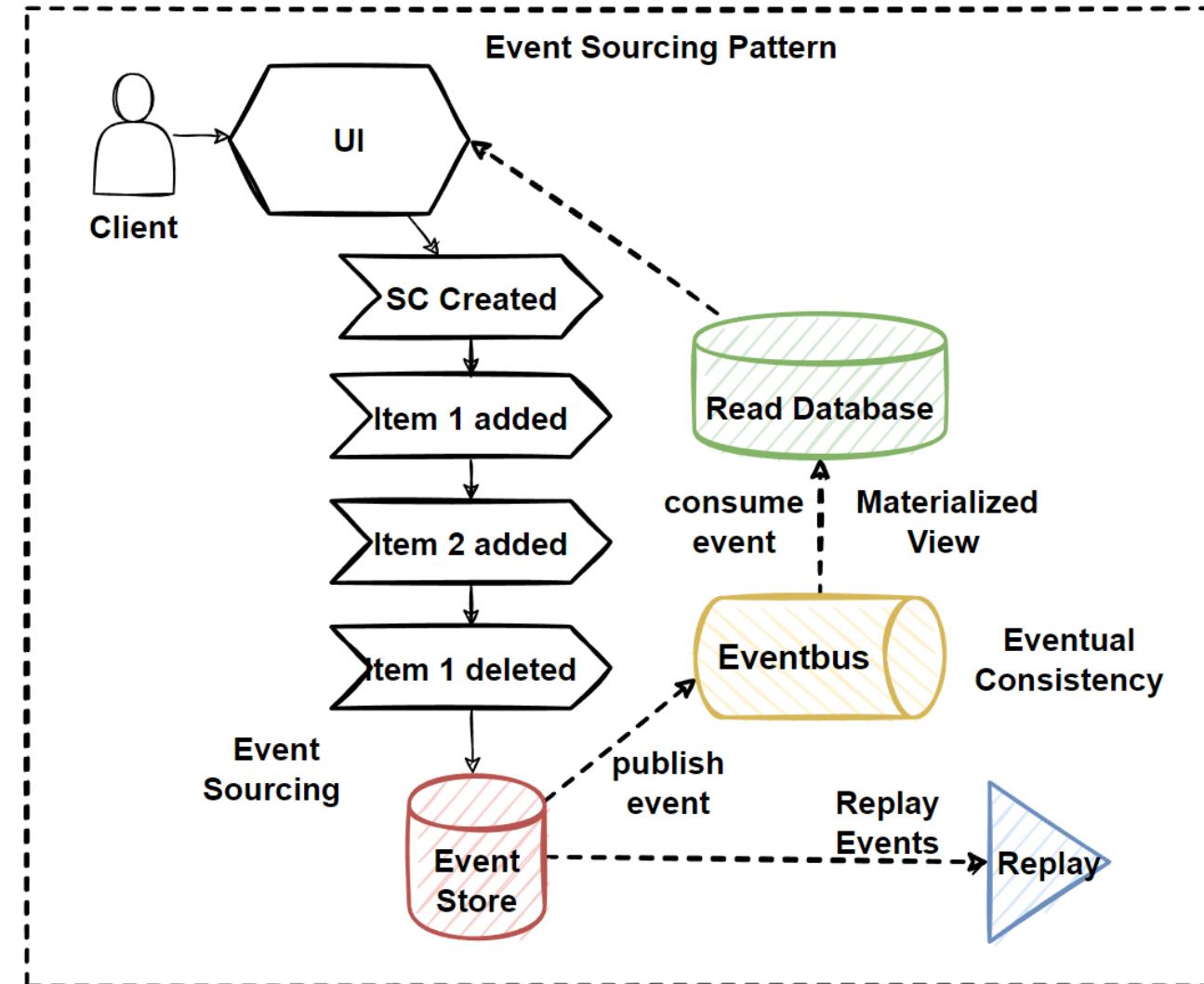
1. When user perform any action into application, this will save actions **as a event into event store**.
 2. Data will **convert to Reading database** with following the publish/subscribe pattern with using message brokers.
 3. Data will be **denormalized** into **materialized view database** for querying from the application.

We call this process is a **Eventual Consistency Principle**.



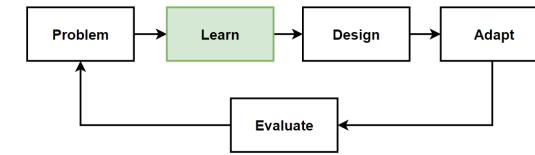
Microservices Data Patterns and Principles

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



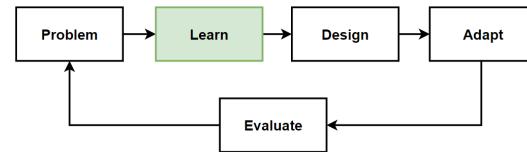
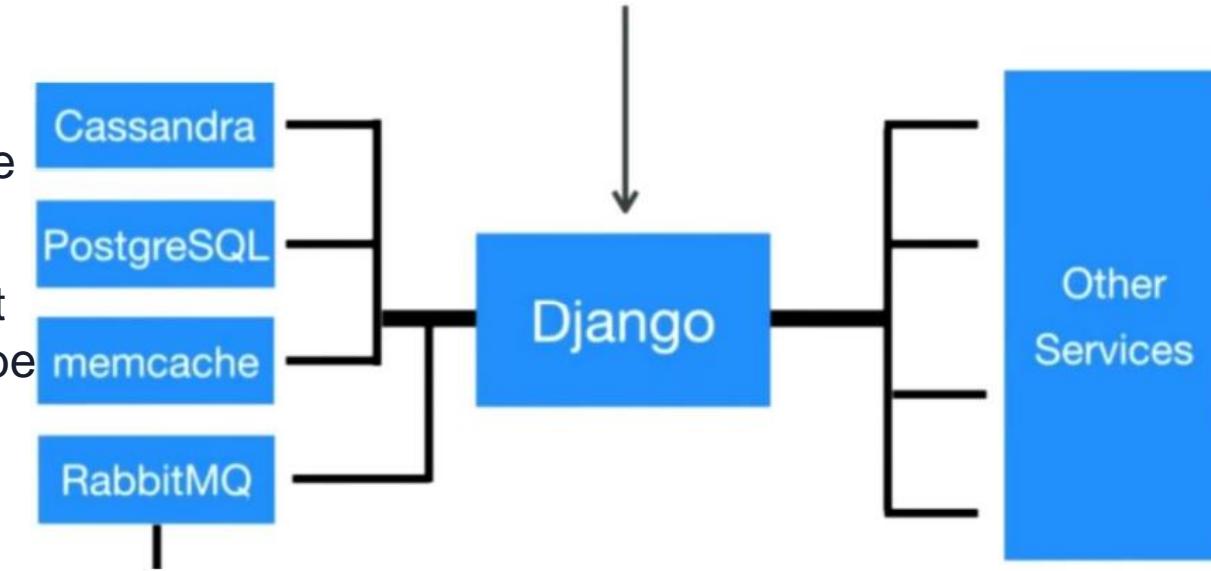
Instagram System Architecture

- Instagram is an application that is used **all over the world** with **under heavy traffic**.
- Visited by **500 million people**, Almost **100 million photos** are shared, **400 million stories**.
- **Huge amount of data** and **interactions** that need to handle with the strong architecture of Instagram.
- Have **many servers**, **data center to spread to many parts** of the world.
- **Network delays** can cause wait for a long time for the **user sending requests**.
- Instagram has a **Data Center** in many parts of the world.



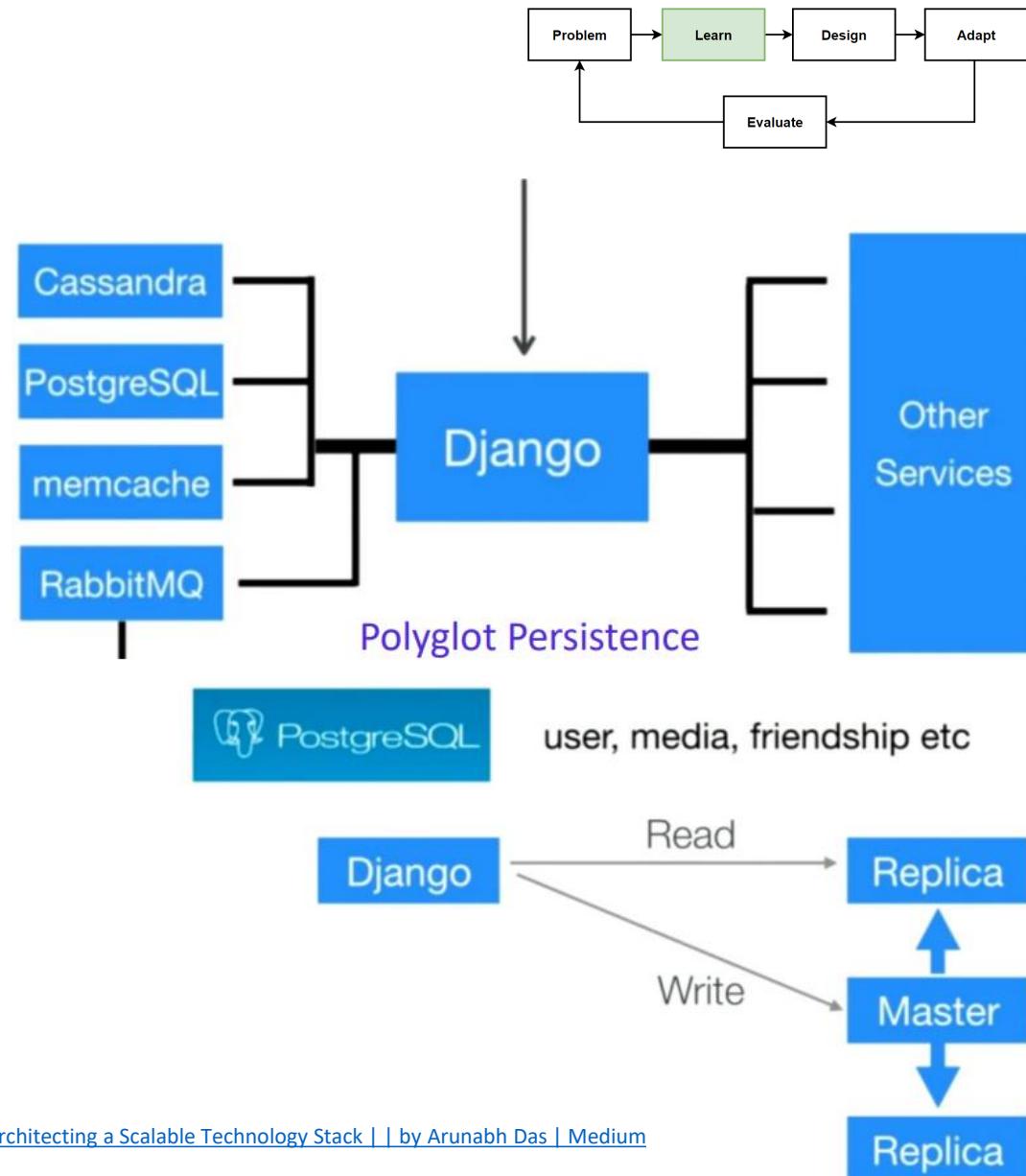
Instagram System Architecture - 2

- Instagram uses **two database systems: PostgreSQL, Cassandra**.
- Designed that users can communicate with **the nearest data center** and **receive quick responses** to requests.
- **CAP Theorem**: either Availability or Consistency must be selected in a distributed system.
- Instagram **decided that Availability** was more important to them and thought that **Eventual Consistency** would be sufficient.
- **Polygot Persistence**: uses relational PostgreSQL and No-SQL Cassandra databases.
- **CQRS Design Pattern**: separated read-incentive use cases with Cassandra No-SQL database which uses is user stories.
- **Eventual Consistency**: Updates reflect with lag, decide the be Availability every time.



Instagram Database Architecture

- Instagram uses two database systems: PostgreSQL, Cassandra.
- **Why use PostgreSQL relational database ?**
PostgreSQL has a "Master-Slave" architecture.
PostgreSQL on Instagram is configured to have one Master and multiple read-only Slaves.
- If the transaction causes a change in the data like update or delete user info, that request must be forwarded to the Master.
- Any update that occurs in the master is sent from the master to the slaves, ensuring the synchronization of the databases.
- **What data does Instagram keep in PostgreSQL?**
user information, user relations (friendship), tags, albums, photos and videos.



[Architecting a Scalable Technology Stack | by Arunabh Das | Medium](#)

Instagram Database Architecture - 2

- **Why use Cassandra No-SQL database ?**

Cassandra has a Master-Master architecture we can call Master-less architecture. Cassandras communication is multi-directional between Data Centers of Instagram.

- **What data does Instagram keep in Cassandra ?**

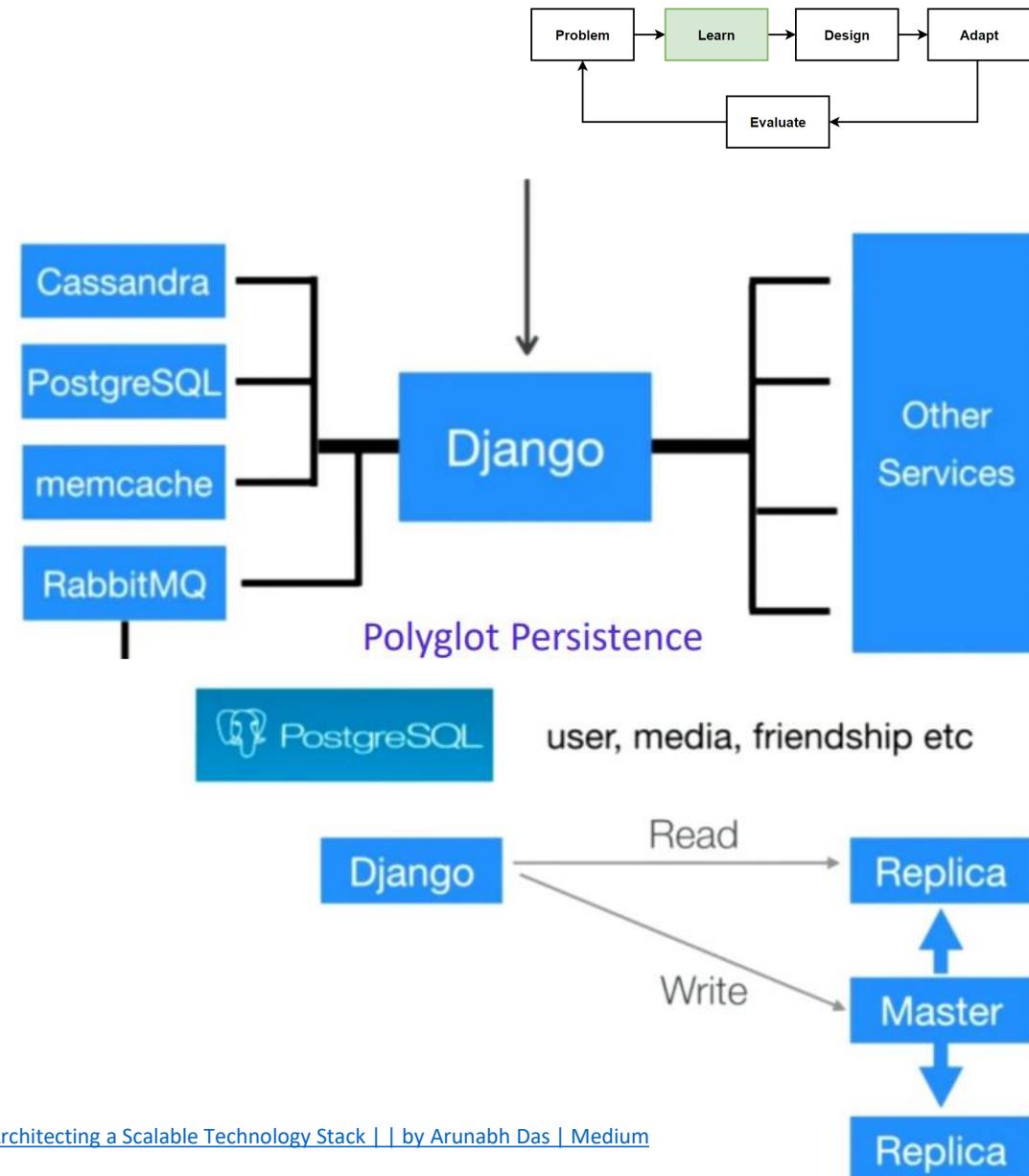
User feed, Counters, Messaging

- The user flow of Instagram is includes the last posts of the people that user follows, resolved by making use of Cassandra.

- **Why was Cassandra chosen ?**

Cassandra has an auto-sharding feature like many NoSQL databases. Sharding helps keep data divided among nodes.

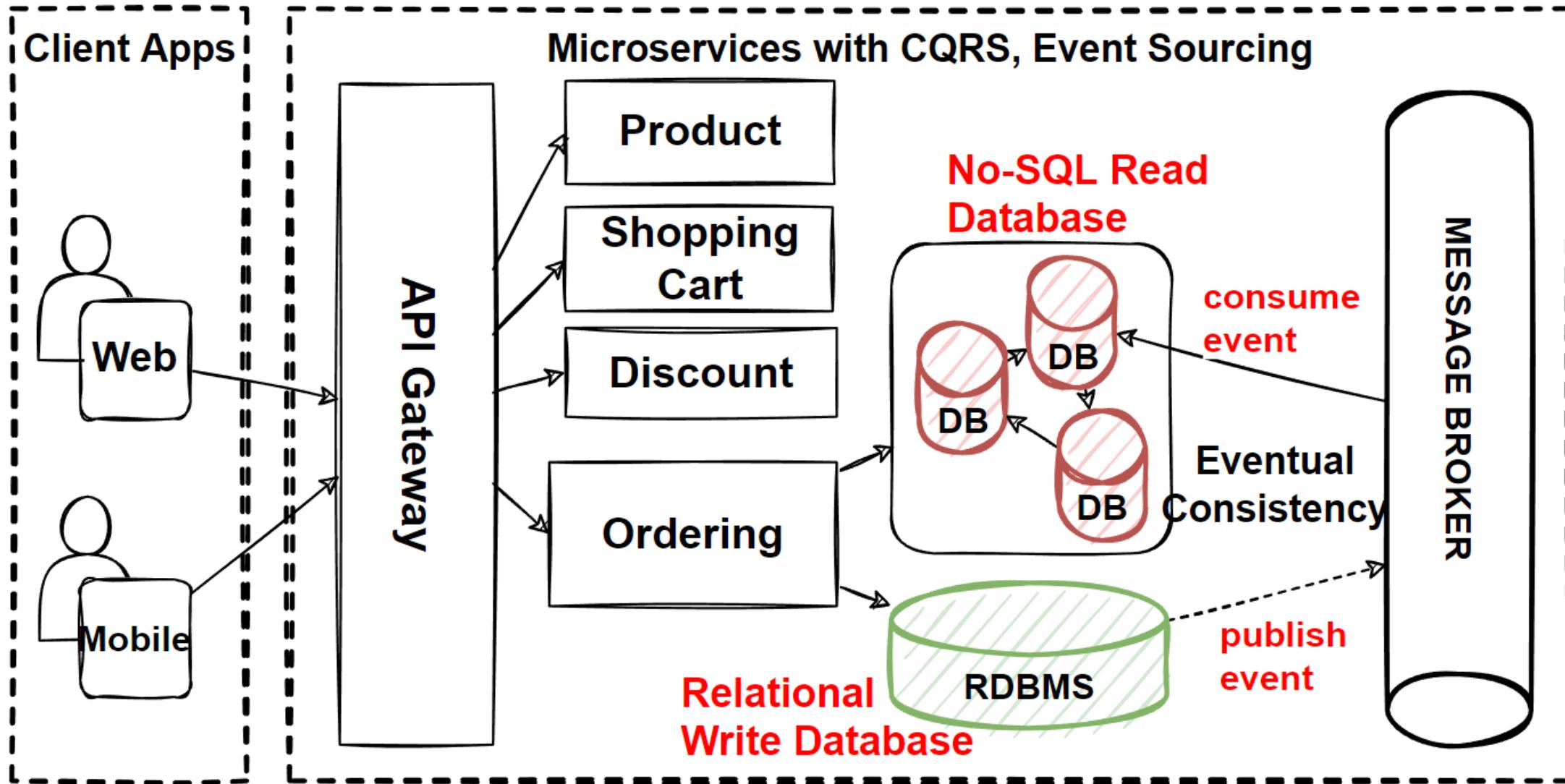
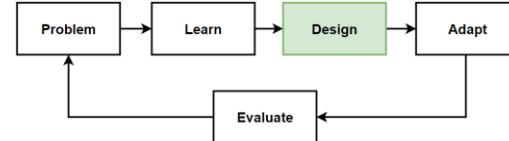
- Disk usage of the servers are optimized and the workload between the nodes is easily distributed.



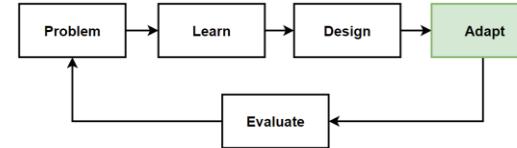
Before Design – What we have in our design toolbox ? - Data

Architectures Patterns&Principles	Microservices Data Choosing Database	Microservices Data Commands&Queries	FR
<ul style="list-style-type: none">• Microservices Architecture• The Database-per-Service Pattern• Polygot Persistence• Decompose services by scalability• The Scale Cube• Microservices Decomposition Pattern• Microservices Communications Patterns• Microservices Data Management Patterns	<ul style="list-style-type: none">• The Shared Database Anti-pattern• Relational and NoSQL Databases• CAP Theorem–Consistency, Availability, Partition Tolerance• Data Partitioning: Horizontal, Vertical and Functional Data Partitioning• Database Sharding Pattern	<ul style="list-style-type: none">• Materialized View Pattern• CQRS Design Pattern• Event Sourcing Pattern• Eventual Consistency Principle	<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history
			Non-FR <ul style="list-style-type: none">• High Scalability• High Availability• Millions of Concurrent User• Independent

Design: Microservices Architecture with CQRS, Event Sourcing, Eventual Consistency



Adapt: Microservice Architecture with CQRS, Event Sourcing, Eventual Consistency



Frontend SPAs

- Angular
- Vue
- React

API Gateways

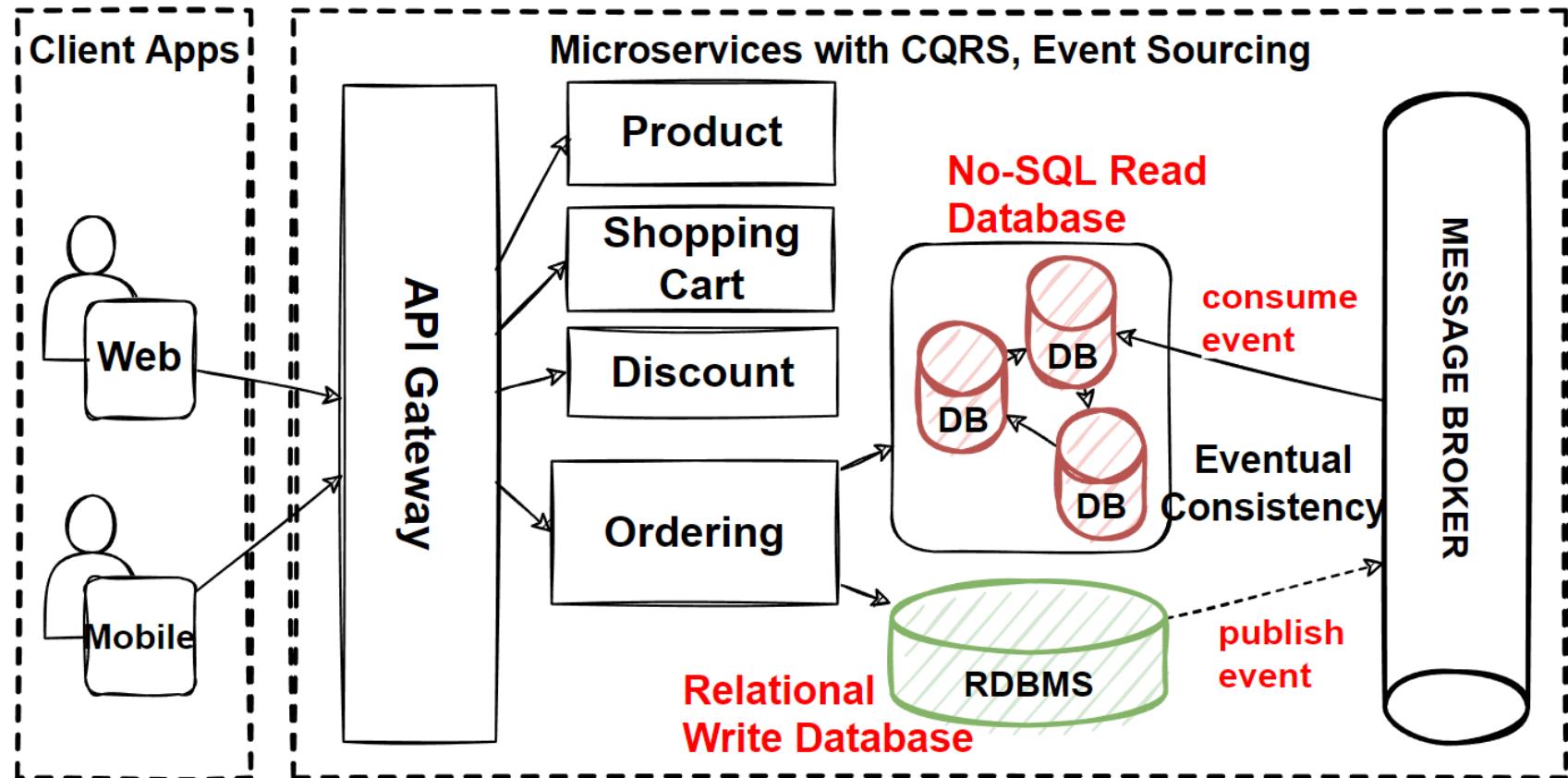
- Kong Gateway
- Express Gateway
- Amazon AWS API Gateway

Message Brokers

- Kafka
- RabbitMQ
- Amazon EventBridge, SNS

Backend Microservices

- Java – Spring Boot
- .Net – Asp.net
- JS – NodeJS



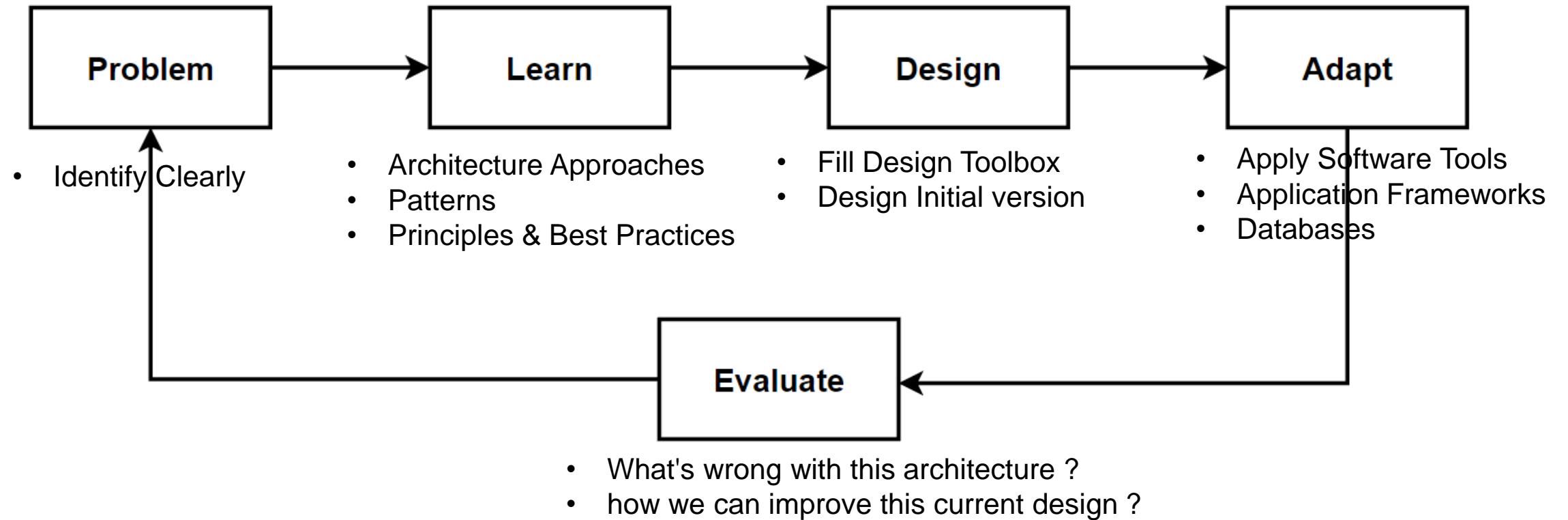
READ DB - No-SQL Database

- Cassandra – NoSQL Peer-to-Peer
- MongoDB – NoSQL Document DB
- Redis – NoSQL Key-Value Pair
- Amazon DynamoDB, Azure CosmosDB

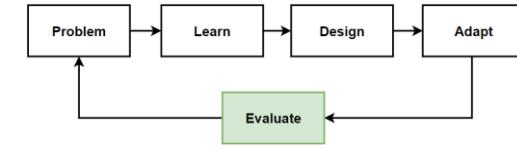
WRITE DB - Relational Database

- PostgreSQL
- SQL Server
- Oracle

Way of Learning – The Course Flow

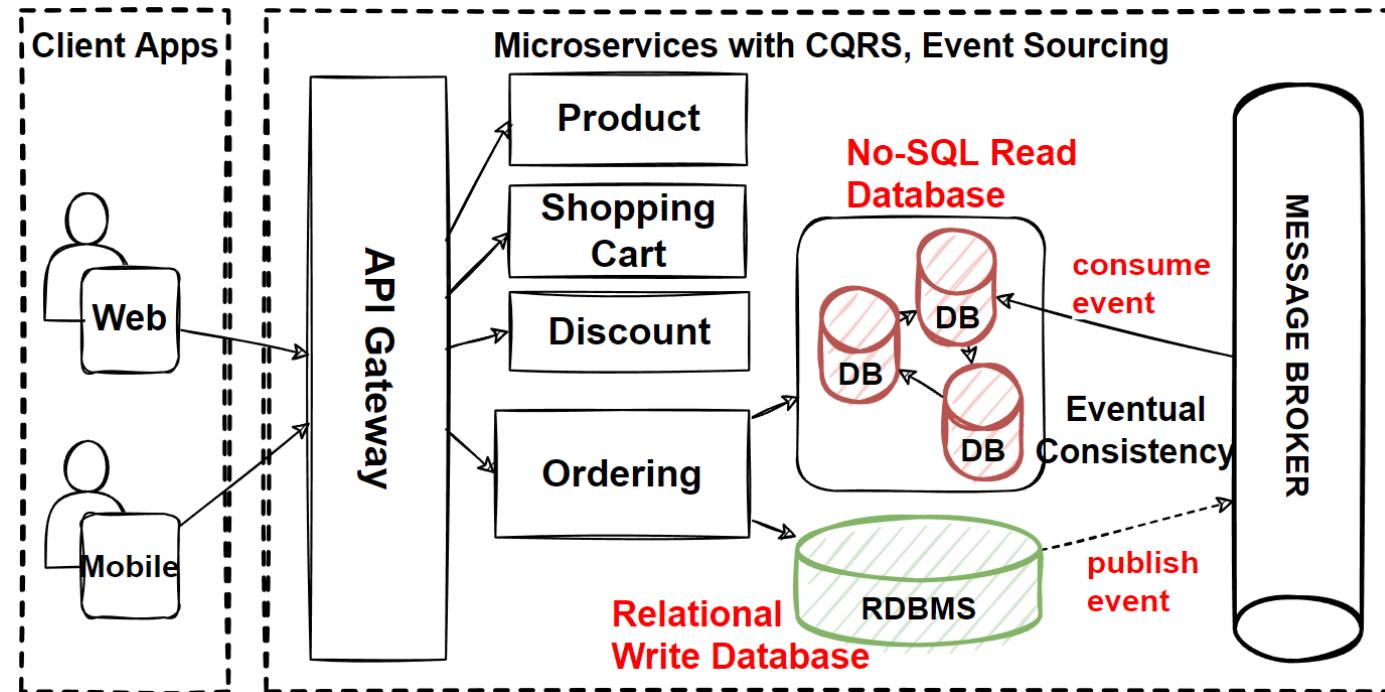


Evaluate: Microservice Architecture with CQRS, Event Sourcing, Eventual Consistency



Benefits

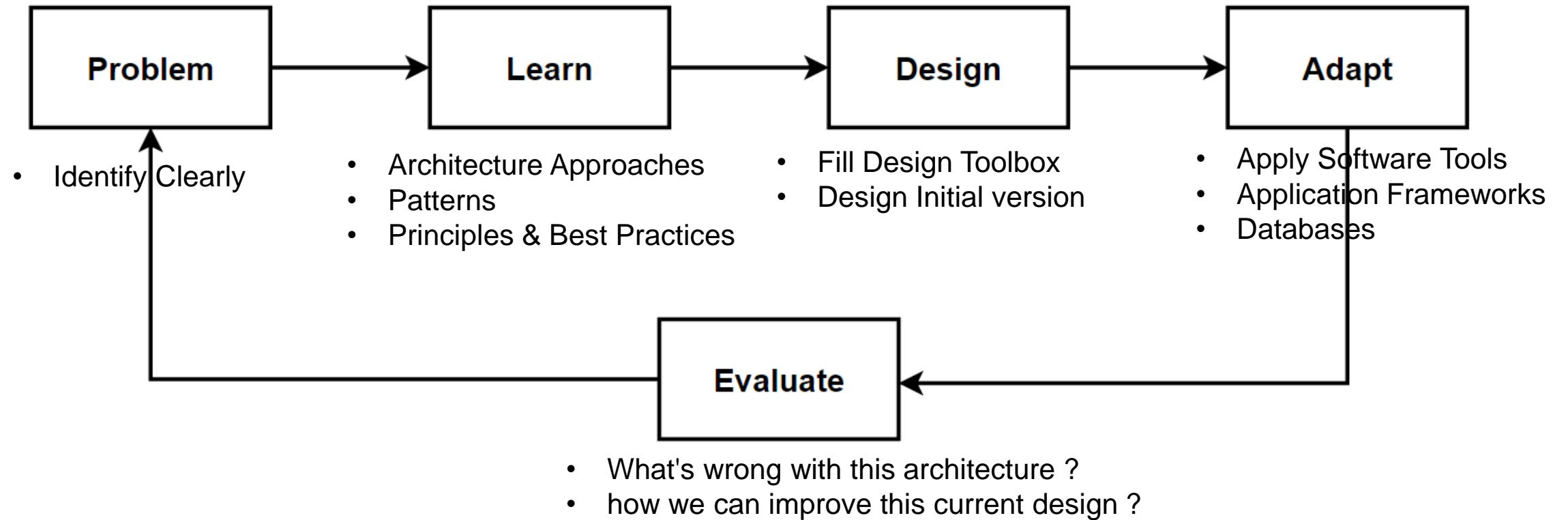
- Better Scalability, Separate Read and Write databases scales independently.
- Increased Query Performance, read database denormalized data reduce to complex and long-running join queries.
- Increased Maintainability and Flexibility, system better evolve over time
- Distributed Horizontally Scaled Databases that ready for Kubernetes Deployments



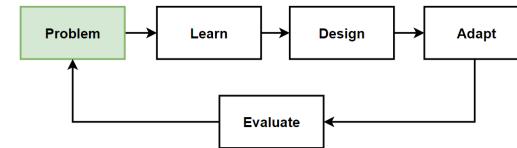
Drawbacks

- Increased Complexity, CQRS makes your system more complex design.
- Eventual Consistency
- Distributed Transaction Management

Way of Learning – The Course Flow



Problem: Manage Consistency Across Microservices in Distributed Transactions



Considerations

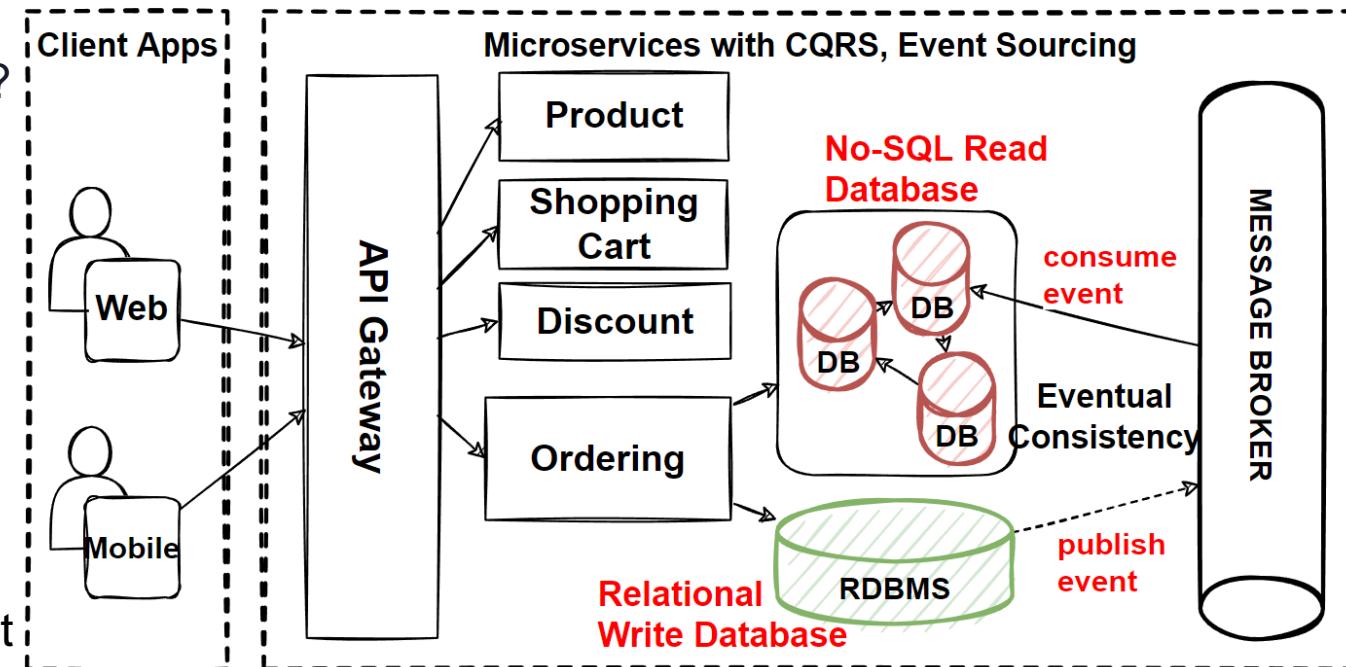
- Distributed Transactions that required to visit several microservices ?
- Consistency across multiple microservices ?
- Rollback transaction and run compensating steps ?

Problems

- Distributed Transaction Management
- Rollback Transaction on Distributed Environment
- Run Compensate Steps if one of service fail

Solutions

- Microservices Distributed Transaction Management Pattern and Best Practices
- Saga Pattern for Distributed Transactions
- Transactional Outbox Pattern
- Compensating Transaction pattern
- CDC - Change Data Capture



Microservices Distributed Transactions

Saga Pattern for Distributed Transactions - Choreography and Orchestration

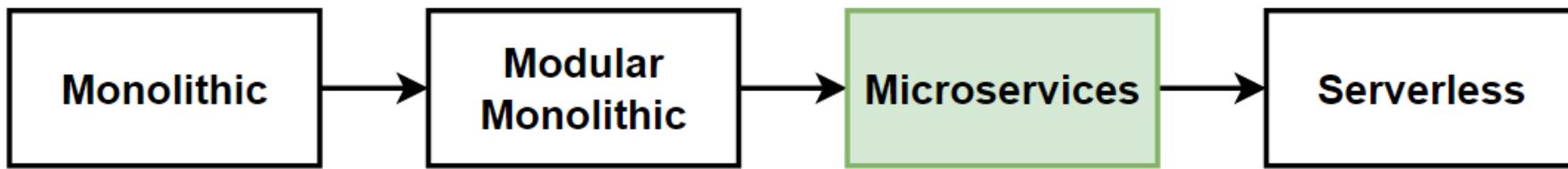
Transactional Outbox Pattern

Compensating Transaction pattern

Dual-Write Problem

CDC - Change Data Capture

Microservices Data Management - Main Topics



Decomposition

Communications

Data Management

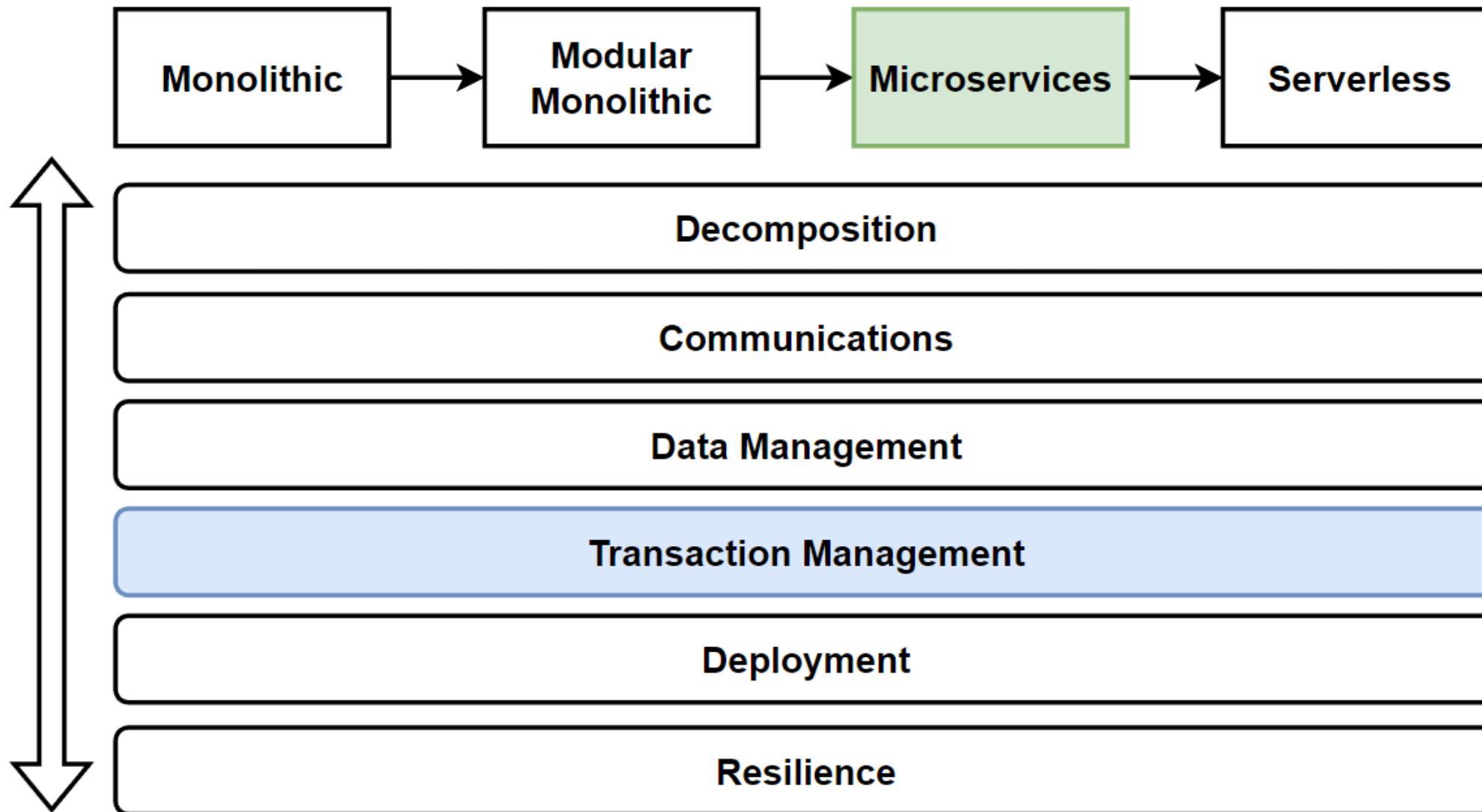
Choosing Right Database

Commands and Queries

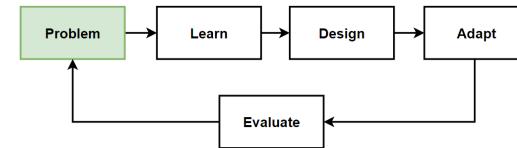
Distributed Transactions

- Polyglot persistence
- The Database-per-Service
- The Shared Database Anti-Pattern
- The API Composition
- The CQRS pattern
- The Event Sourcing Pattern
- The Saga Pattern
- Transactional Outbox
- Compensating Transactions
- CDC - Change Data Capture

Architecture Design – Vertical Considerations



Problem: Manage Consistency Across Microservices in Distributed Transactions



Considerations

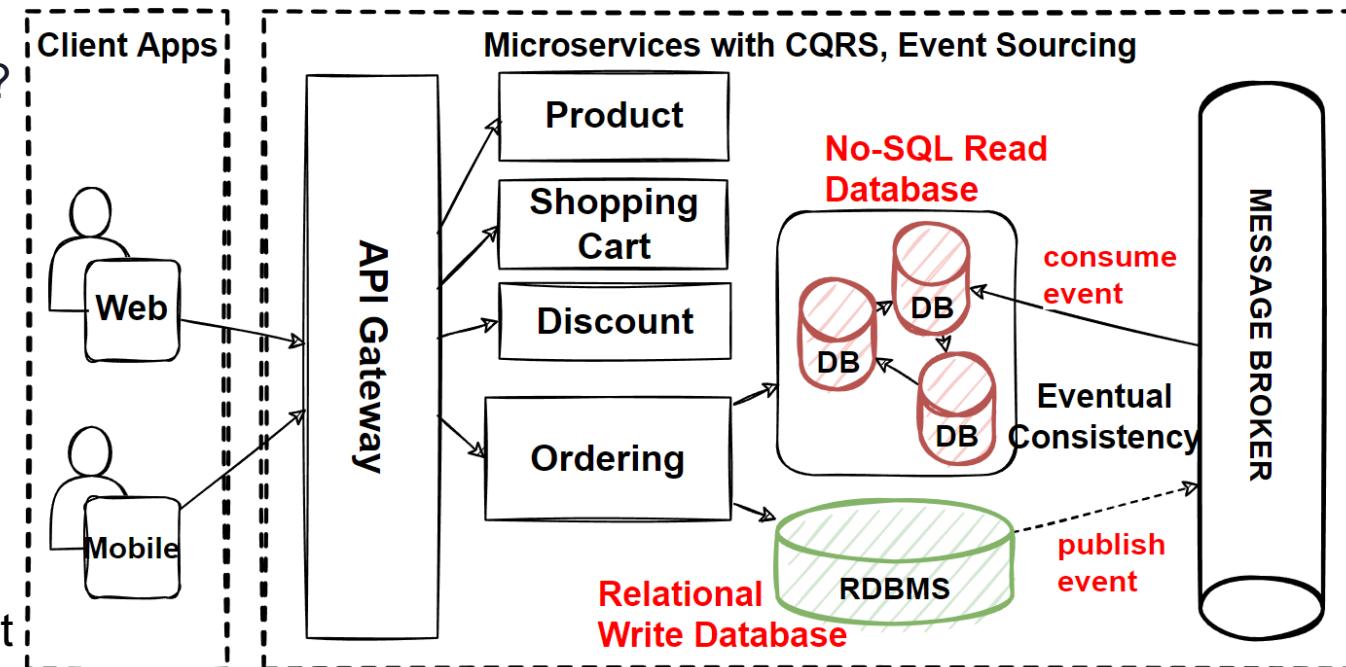
- Distributed Transactions that required to visit several microservices ?
- Consistency across multiple microservices ?
- Rollback transaction and run compensating steps ?

Problems

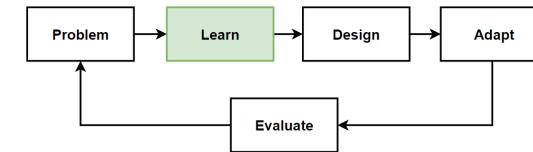
- Distributed Transaction Management
- Rollback Transaction on Distributed Environment
- Run Compensate Steps if one of service fail

Solutions

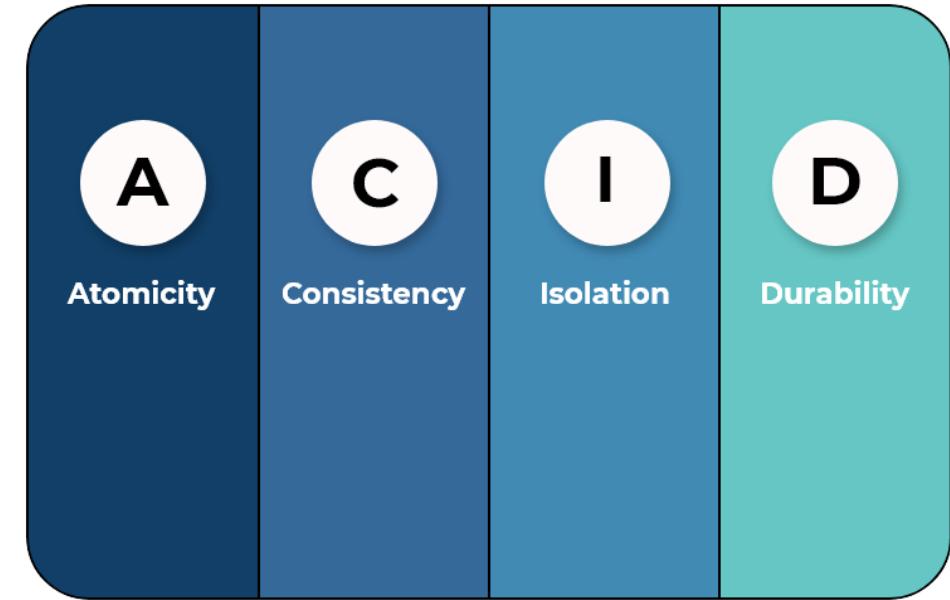
- Microservices Distributed Transaction Management Pattern and Best Practices
- Saga Pattern for Distributed Transactions
- Transactional Outbox Pattern
- Compensating Transaction pattern
- CDC - Change Data Capture



Transaction Fundamentals and ACID Principles

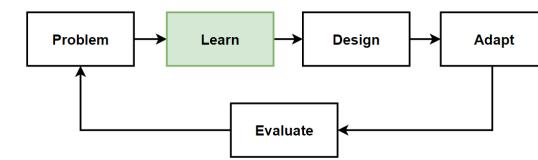
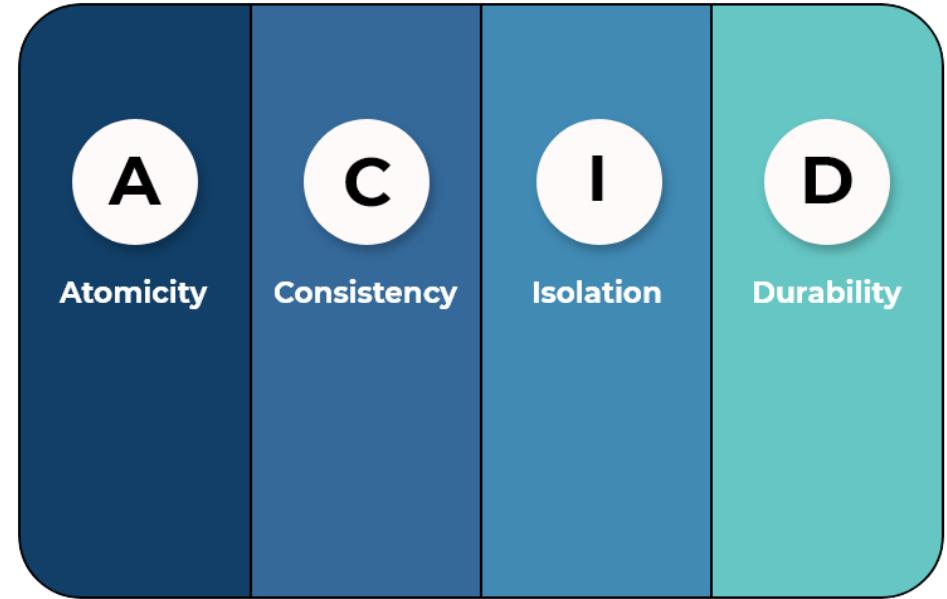


- **Transaction** is the **context of database operations** that should act to be **single unit of work**.
- Should **fully complete the operation** or if something fail, it should **rollback the whole operation** and go back to consistent state.
- **ACID - Atomicity, Consistency, Isolation, and Durability**
- If one change fails on database, the **whole transaction** is going to fail, and the database **should remain** in the **previous state**.
- **Relational Databases** are providing ACID principles to ensure Data Consistency.
- If you need **strong consistency**, its good to choose **Relational Databases**.



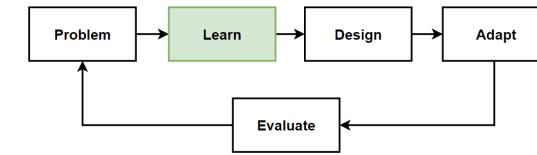
ACID Principles

- **ACID** - Atomicity, Consistency, Isolation, and Durability
- **Atomicity**
All operations are executed successfully or everything fails and rollback together.
- **Consistency**
The data in the database is kept in a valid and consistent state, after any operations on data.
- **Isolation**
Separated transactions running concurrently and they can not interfere with each other.
- **Durability**
After a transaction is committed, the changes are stored durably, for example persisted in a database server.
- **Example:** Money transfer transaction.



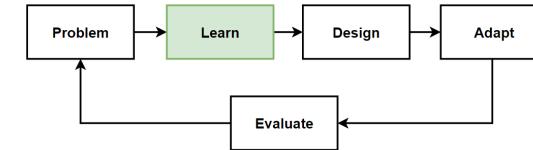
Transaction Management of Monolithic Architecture

- **Transaction management** in Monolith architecture is quite easy compared to Microservice Architecture. Many frameworks contains mechanism for transaction management.
- These mechanism have a **single database** of the **whole application**. They are developed for scenarios where all transactions are running on a **single context**.
- Simply **commit** and **rollback** operations with these mechanism in monolith architectures.
- **Transactions operated** in the **transaction scope** are kept in memory without writing to the database until they are **committed**, and if a **Rollback** is made at any time, all transactions in the scope are **deleted** from memory and the **transaction is canceled**.
- When **Commit** is **written** to the **database**, the transaction is **completed successfully**.

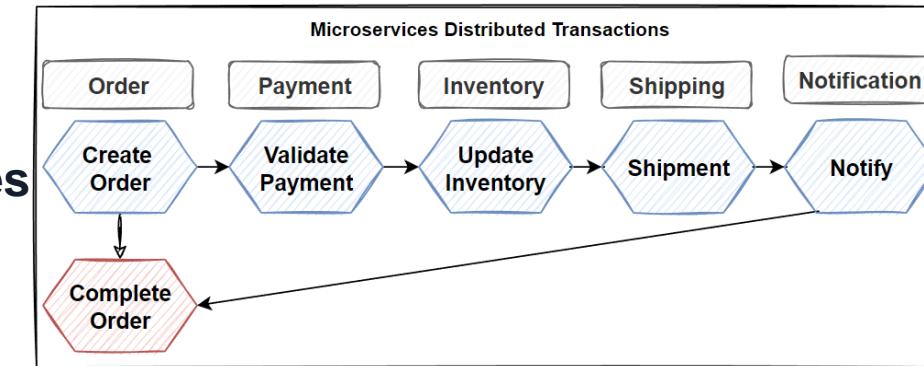


```
function place_order()
    do_payment
    decrease_stock
    send_shipment
    generate_bill
    update_order
```

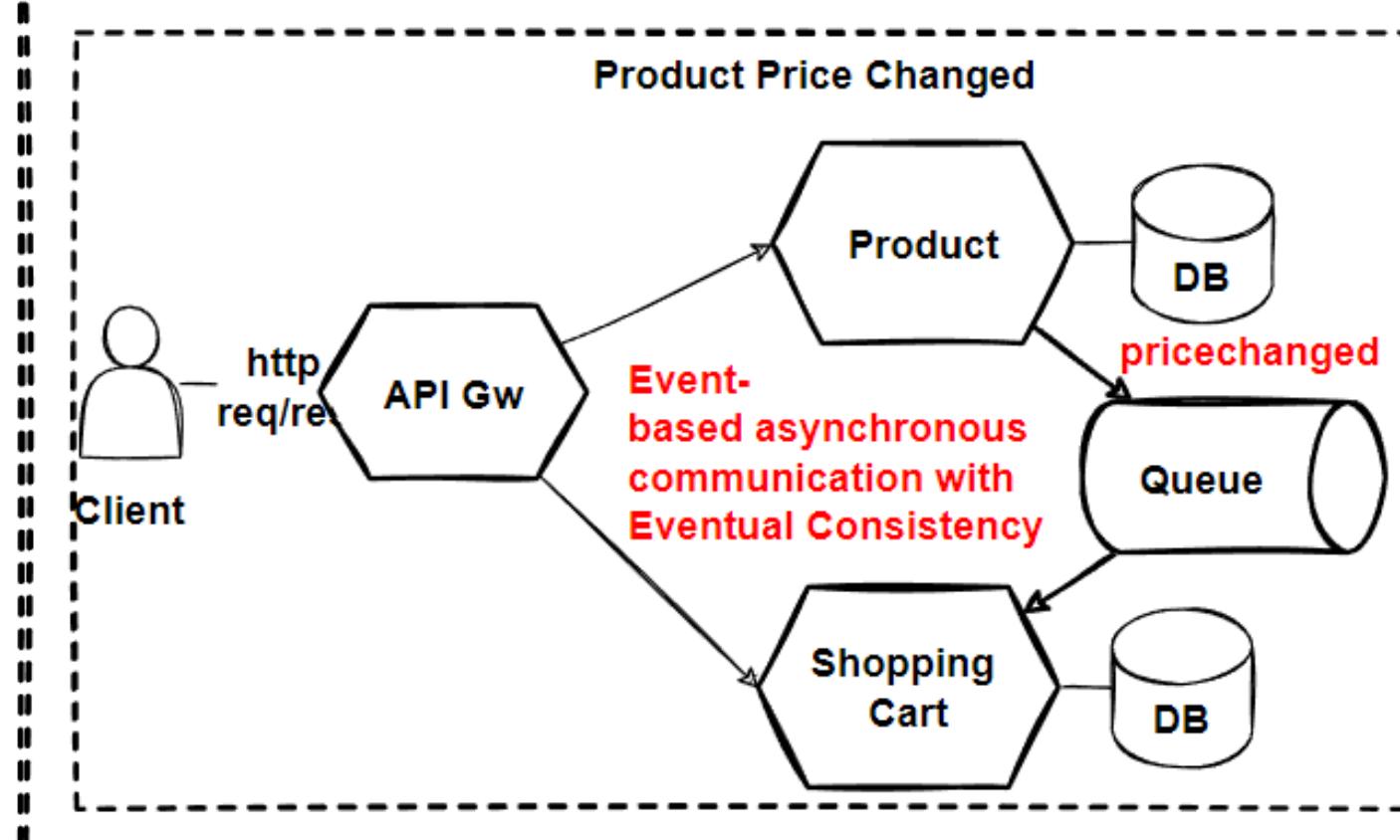
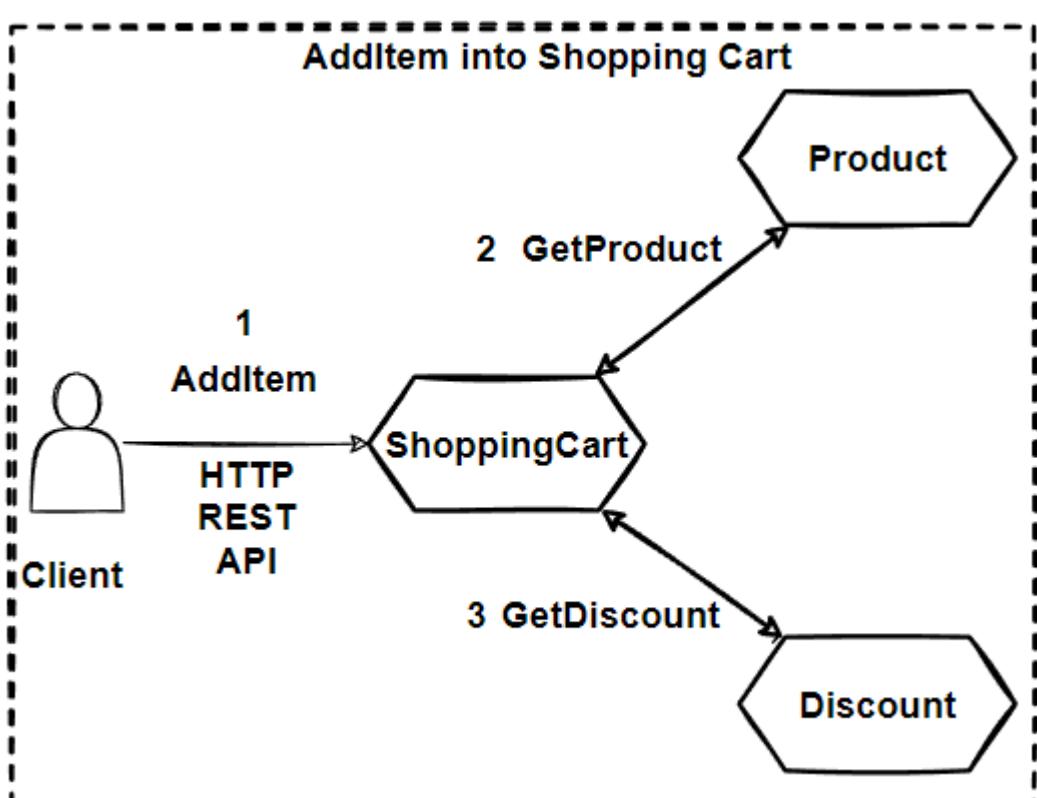
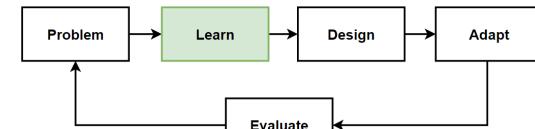
Microservices Distributed Transactions



- How to perform **transactional operations across microservices** and which patterns we should apply ?
- **Querying data and run complex join queries** across microservices is not easy but implement **transactional operations** across microservices is **more complex**.
- We should handle **distributed transaction managements** on microservices by manually implementing some **patterns and practices**
- **Distributed transaction managements** on microservices, mostly working with **eventual consistency**.
- Microservices has its own database and communicates each other with exposing APIs, it makes **harder to perform end-to-end business cases** across multiple microservices when trying to **keep data consistency**.
- How to **achieve consistency** across multiple microservices ?

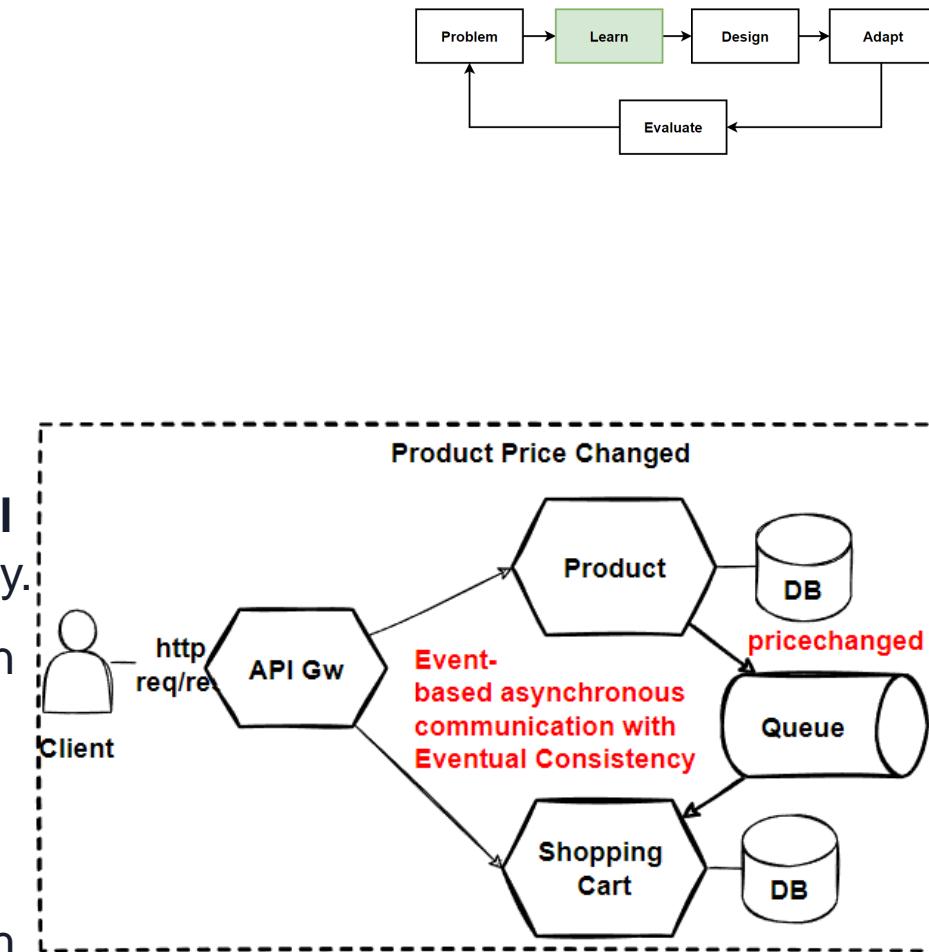


How to achieve Data Consistency across multiple microservices ?



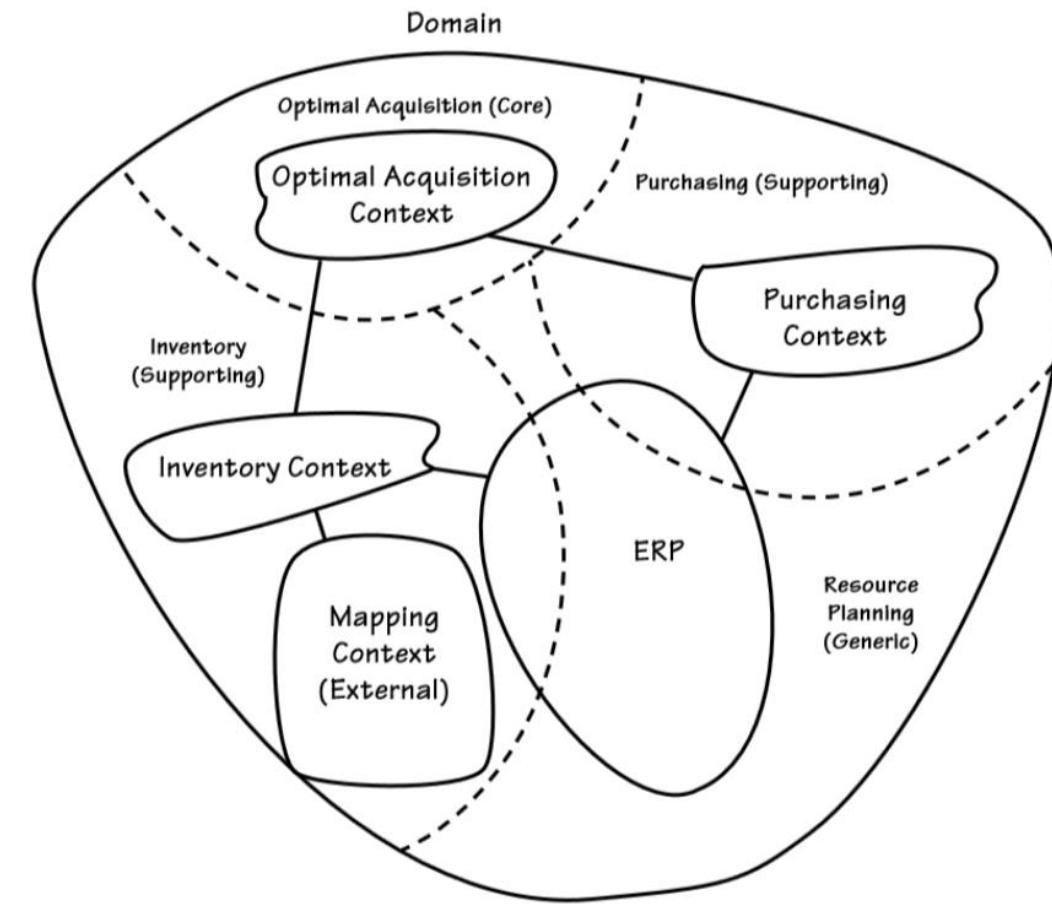
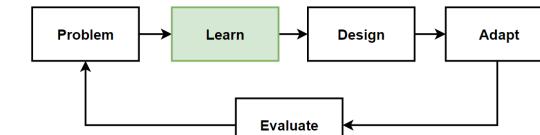
How to achieve Data Consistency across multiple microservices ? - 2

- **CAP Theorem:** need to **sacrifice ACID strong consistency** to the **High Availability** and **Partition Tolerance**.
- Most **microservice-based applications** are **choosing high availability** and **high scalability** against to strong consistency.
- Microservices **sacrifice strong consistency** and follow with **eventual consistency** to get benefits of microservices availability and scalability.
- **NoSQL database** are mostly using in microservices because they can easily and **horizontally scale** in distributed environments.
- Must use **asynchronous event-driven communication** and **publish/subscribe pattern** following to **eventual consistency**.
- When **product price updated**, Product service **publish changes** with Product price changed event to the Event Bus, that **subscribe by** ShoppingCart service with following eventual consistency principle.
- This process should be **resilient** and **idempotent** for redundant event processing; **Microservices Resillience**.



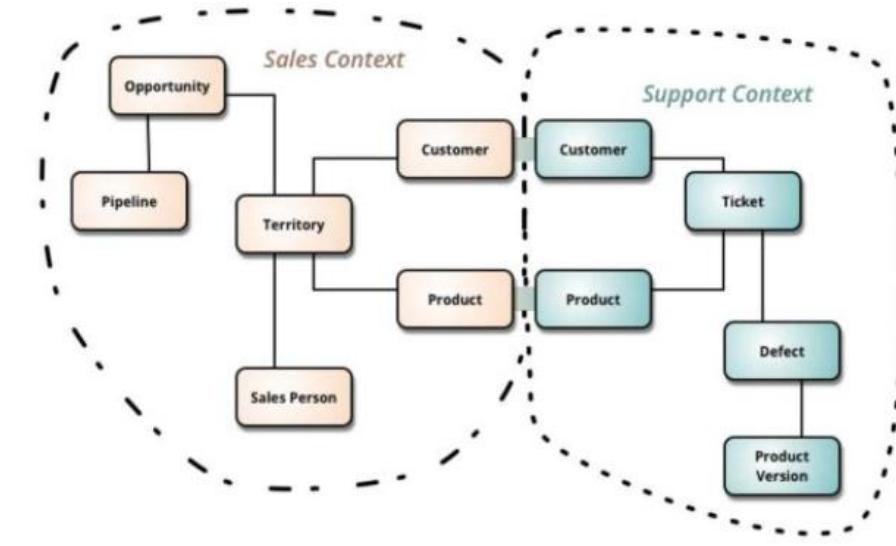
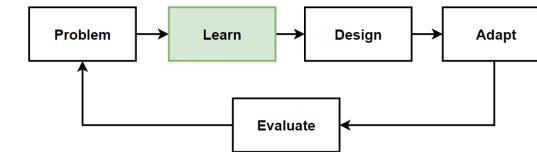
Bounded Context Pattern (Domain-Driven Design)

- **DDD - Bounded Context Pattern** which is one of the main pattern that we mainly use when **decomposing microservices**.
- Domains are **require high cooperation** and have a certain complexity by nature are called **collaborative domains**.
- DDD has 2 phases, Strategic and Tactical DDD.
- Strategic DDD, we define the large-scale model of the system, defining to the business rules that allow designing loosely coupling units and the context map between them.
- Tactical DDD focuses on implementation and provides design patterns that we can use to build the software implementation.
- Include concepts such as entity, aggregate, value object, repository, and domain service.



Bounded Context Pattern (Domain-Driven Design)-2

- DDD domain defines its **own common language** and divides boundaries into specific, independent components. Common language is called **ubiquitous language**, and independent units are called **bounded context**.
- DDD is solving a **complex** problem is to break the problem into **smaller parts** and focus on smaller problems that are relatively easy.
- A **complex domain** may contain **sub domains**. And some of sub domains can **combine and grouping** with each other for **common rules and responsibilities**.
- **Bounded Context** is the grouping of closely related scopes that we can say **logical boundaries**.
- **Bounded context** is the **logical boundary** that represents the smaller problem particles of the complex domain that **are self-consistent** and as **independent** as possible.

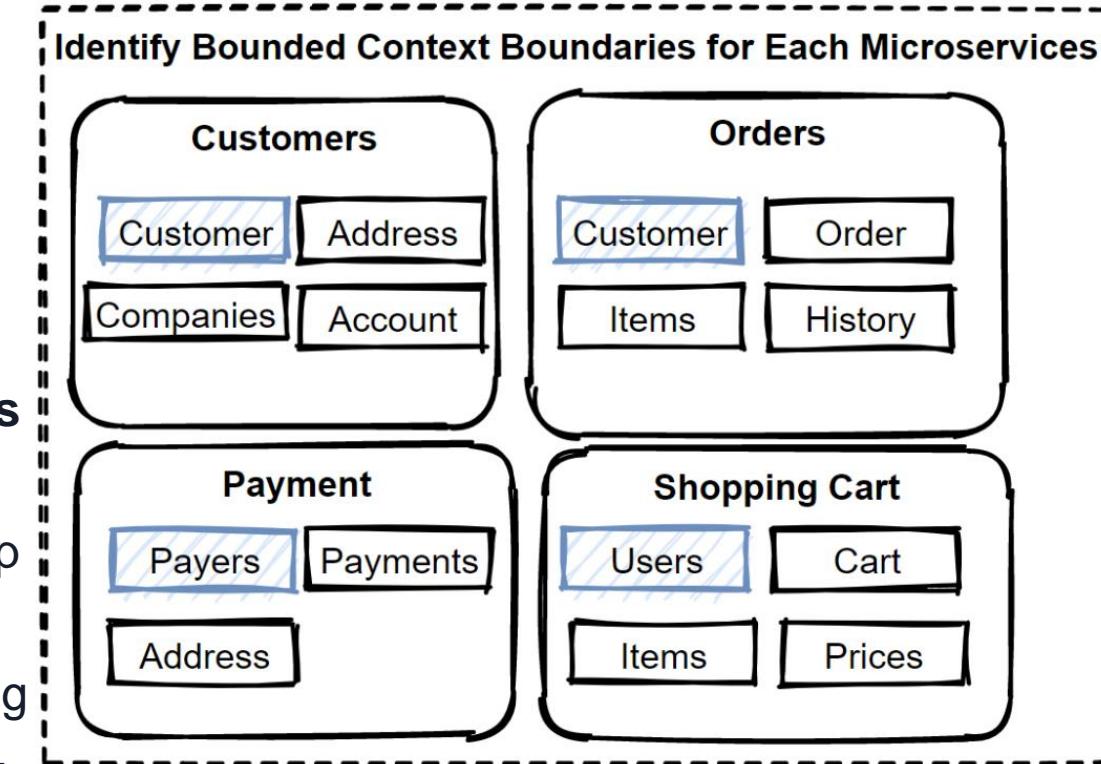


DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.

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

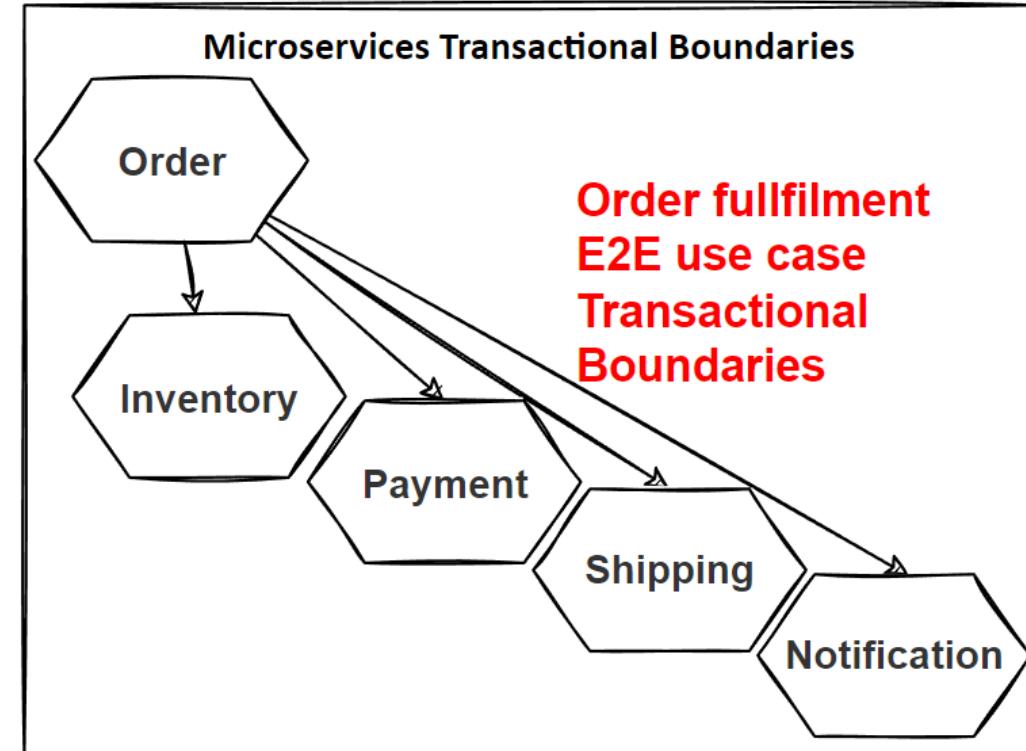
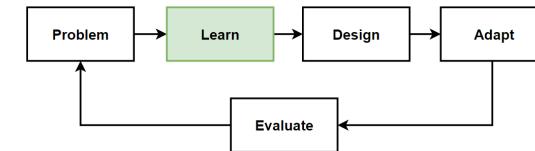
Identify Bounded Context Boundaries for Each Microservices

- Identify the **Bounded Contexts** by talking to the **domain experts** and using some clues.
- Once defined the Bounded Contexts, iterate design, those are **not immutable**.
- Reshape your **Bounded Contexts** by talking to the domain experts and consider **refactoring's** with the changing conditions.
- Its crucial to discuss with **domain experts** to defining **domains** and **sub domains**.
- Evaluate **Bounded Context** with the **domain experts** will help you identify to microservices.
- **Sub domains** inside of the **Bounded Context** are representing same data but naming differently due to domain experts areas.
- Should discuss **several domain experts** for their expertise areas.

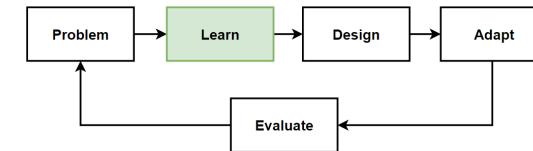


Microservices Transactional Boundaries

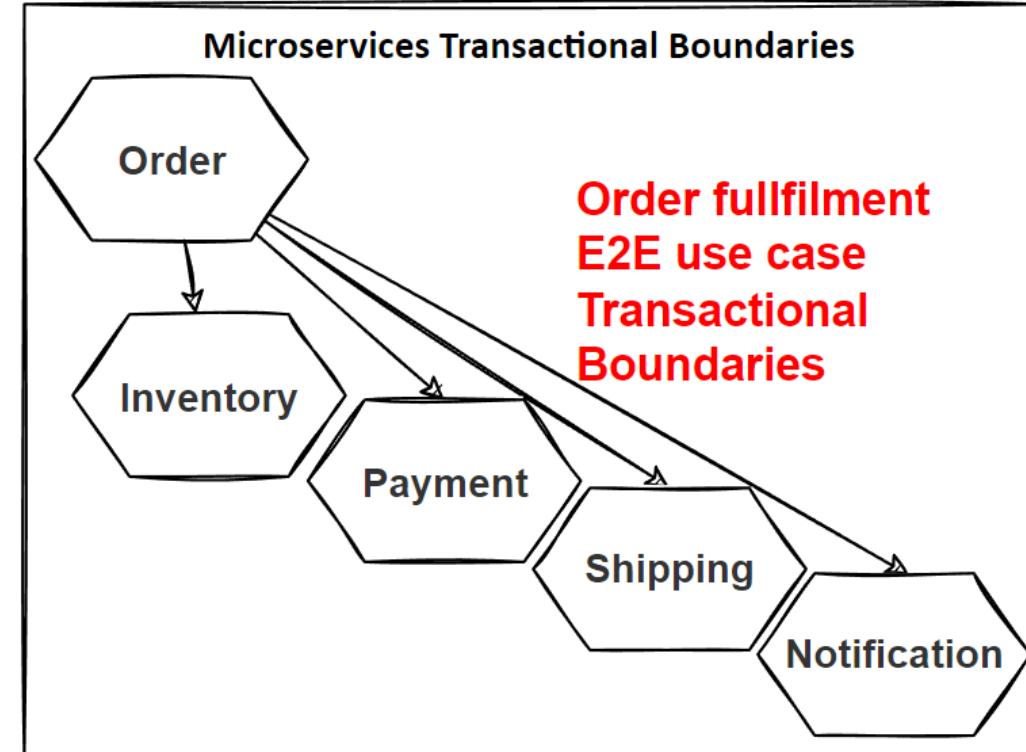
- We have several Bounded Context and Microservices for our e-commerce application:
- Customer-User, Product, Shopping Cart, Discount
- Ordering, Payment, Inventory, Shipment, Notification
- Some of these bounded context need to organize and communicate each other to perform End-to-end business use cases.
- After identifying Bounded Contexts and microservices, should also identify our microservices transactional boundaries.
- What are Microservices Transactional Boundaries ?



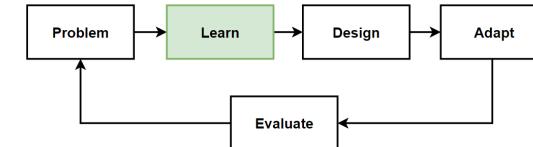
What are Microservices Transactional Boundaries ?



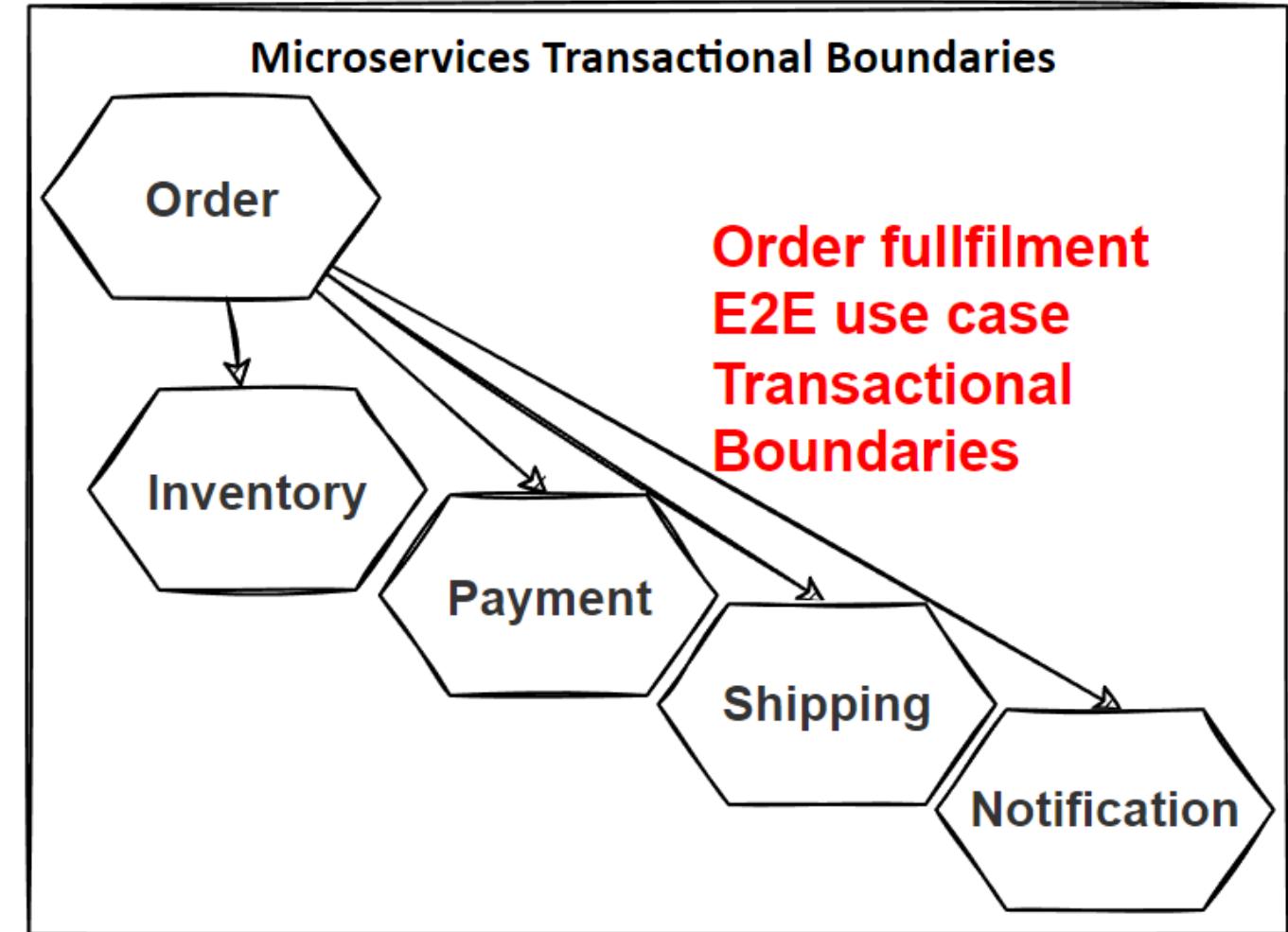
- **Transactional Boundaries** are **smallest unit of atomicity** that need to provide consistency between services.
- In **distributed architecture** we **can't provide ACID principles**.
- Should define **Transactional Boundaries** and perform **E2E use cases** to try to **keep data consistency**.
- Use **asynchronous event-driven communication** and **publish/subscribe pattern** following to **eventual consistency**.
- Not enough for **complex and long E2E business case** that required to visit more than 5 microservices in a distributed environment.
- **Microservices Transactional Boundaries** is identify **minimum smallest unit of atomicity** use cases and design communication between those boundaries.



E-Commerce Microservices Transactional Boundaries

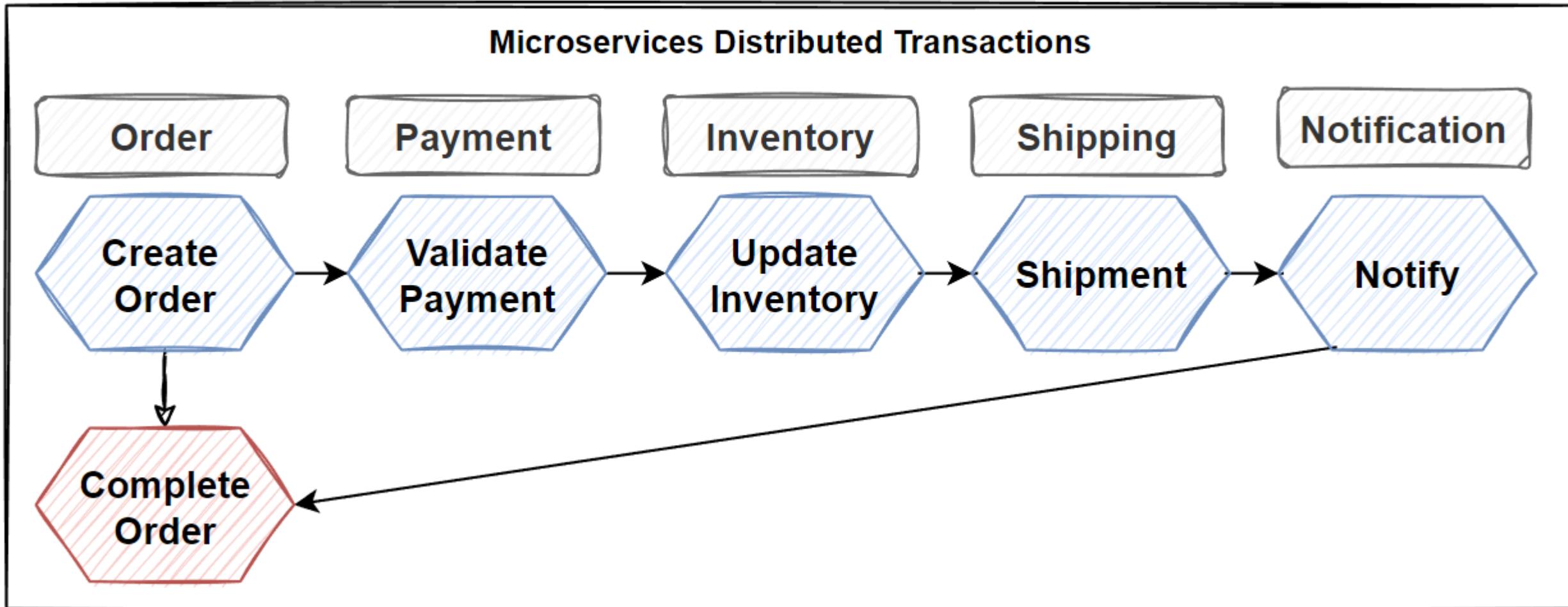
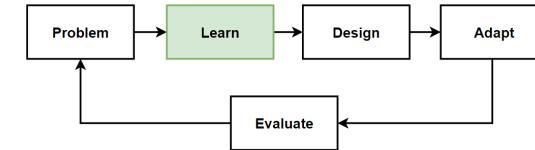


- Customer checkout Shopping Cart and start Ordering process
- Order created into Order microservices and start E2E fulfillment process
- Payment should be validate and done
- Deduct items into Inventory with update Inventory Service
- Shipping to Customer Address manages from Shipment Service
- Notification send to Customer from Notification Service



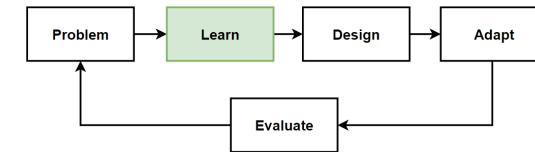
Microservices Transactional Boundaries

Order Fulfillment - Success

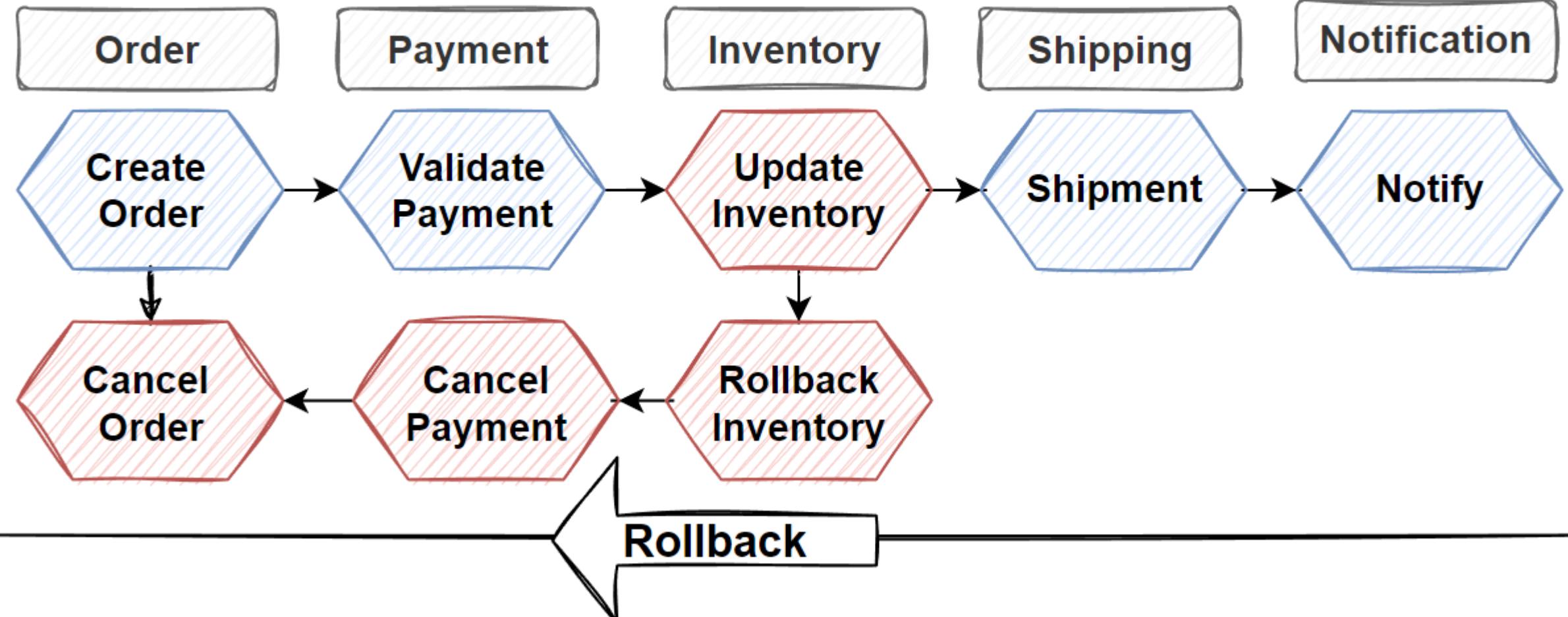


Microservices Transactional Boundaries

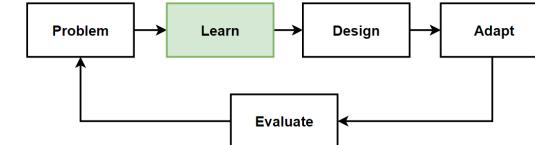
Order Fulfillment - Rollback



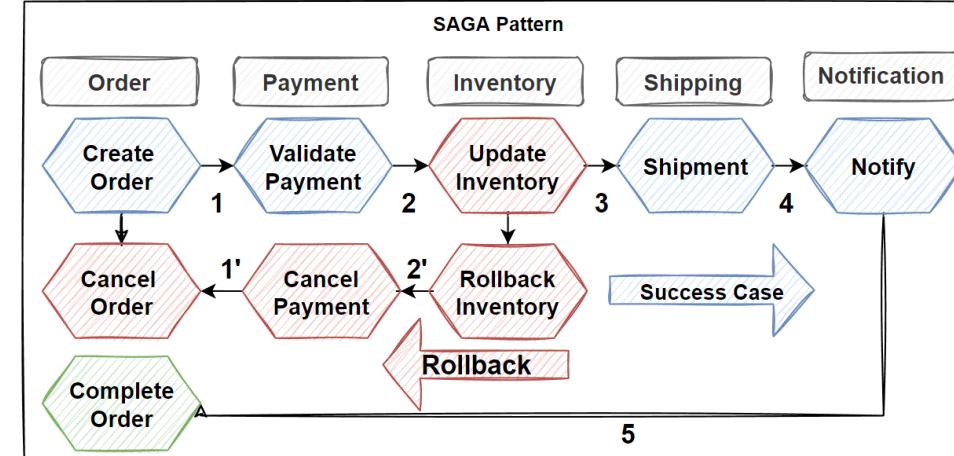
Microservices Distributed Transactions - Rollback



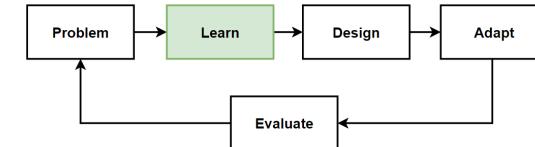
Saga Pattern for Distributed Transactions



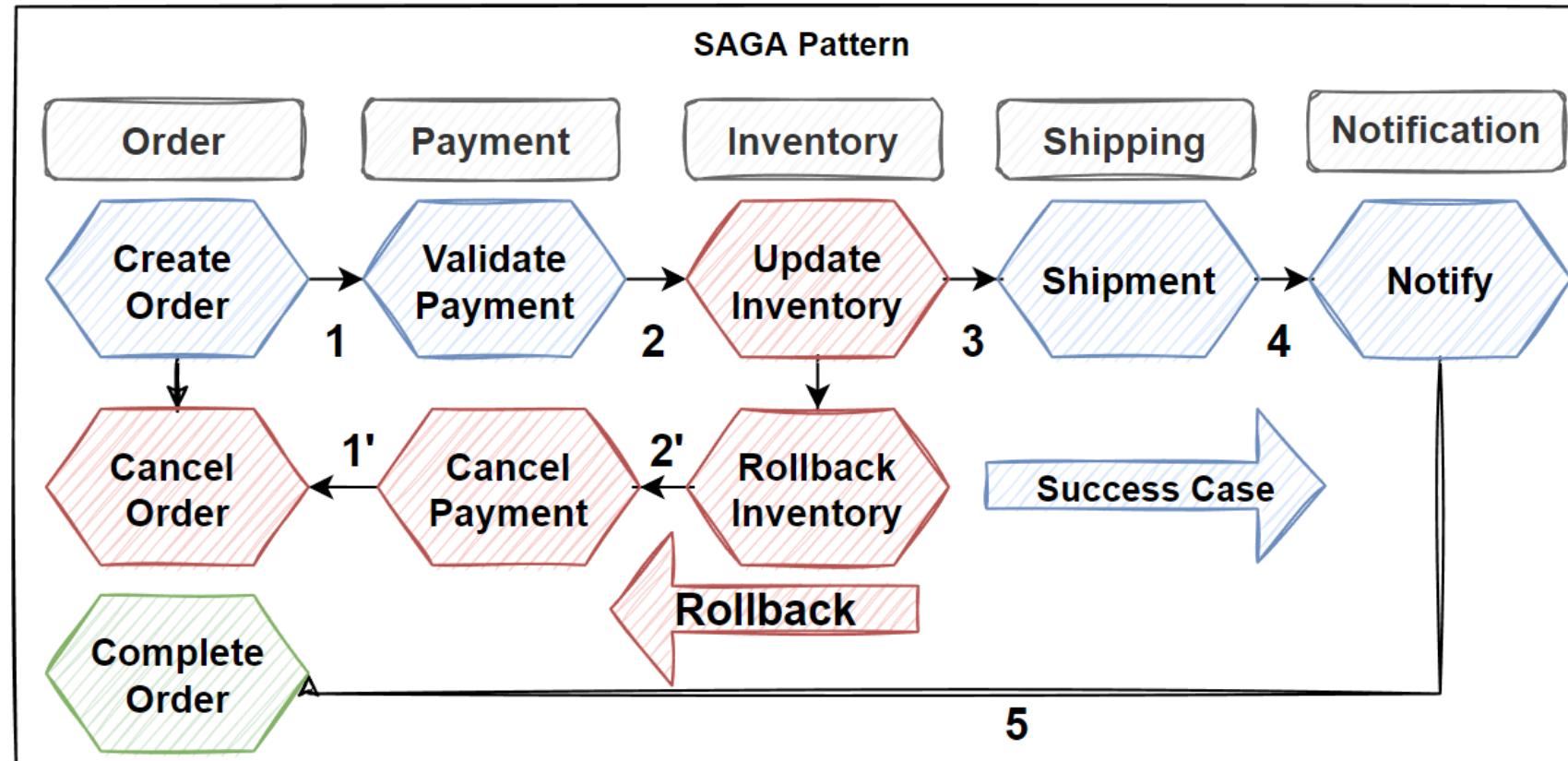
- **Saga design pattern** is provide to **manage data consistency** across microservices in **distributed transaction cases**.
- **Saga** offers to **create a set of transactions** that **update microservices sequentially**, and **publish events** to trigger the next transaction for the next microservices.
- If **one of the step is failed**, than saga patterns **trigger to rollback transactions**, do **reverse operations** with publishing **rollback events** to previous microservices.
- **Publish/subscribe pattern** with brokers or **API composition**.
- **SAGA pattern** manage **long-running transactions** that involve multiple microservices which is a **series of local transactions** that work together to achieve **E2E use case**.
- Useful in distributed systems, where **multiple microservices** need to **coordinate their actions**.
- Ensure that the **overall transaction** is either **completed successfully**, or is **rolled back** to its **initial state**. (compensating transaction)



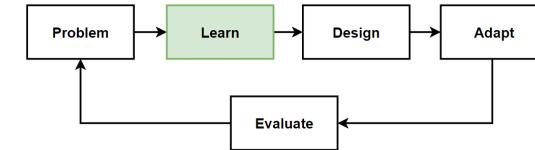
Saga Pattern for Distributed Transactions - 2



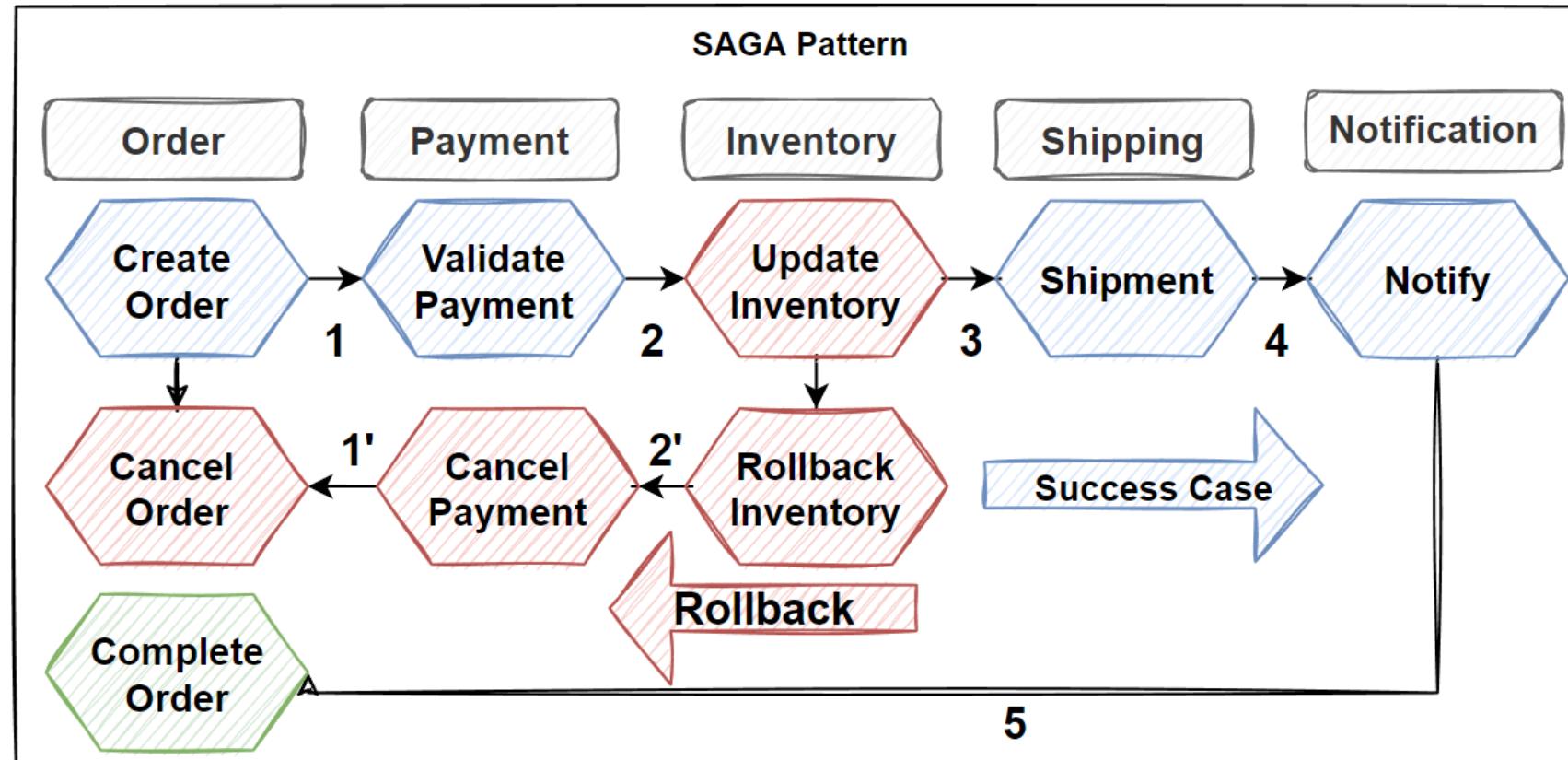
- Saga pattern provides transaction management with **using a sequence of local transactions** of microservices. And **grouping these local transactions** and **sequentially invoking** one by one.
- Each **local transaction updates the database** and publishes an event to trigger the **next local transaction**. If **one of the step is failed**, than saga patterns **trigger to rollback transactions**.



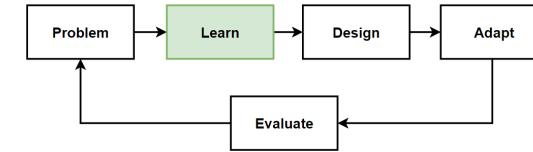
Types of Saga Implementation



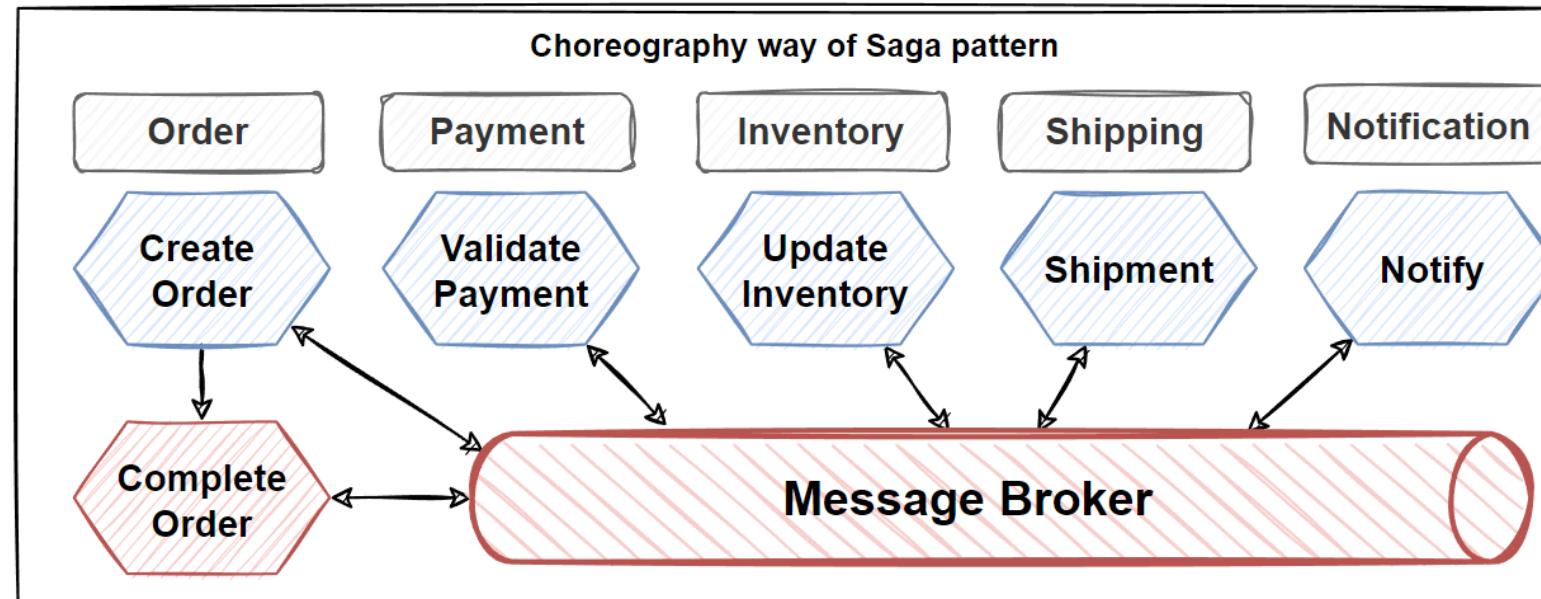
- There are **two type of saga** implementation ways:
- **Choreography-based SAGA** Implementation
- **Orchestration-based SAGA** Implementation



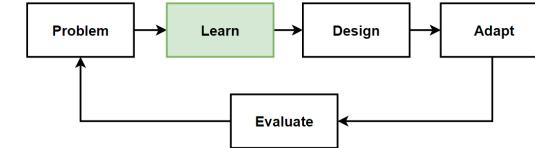
Choreography-based SAGA Implementation



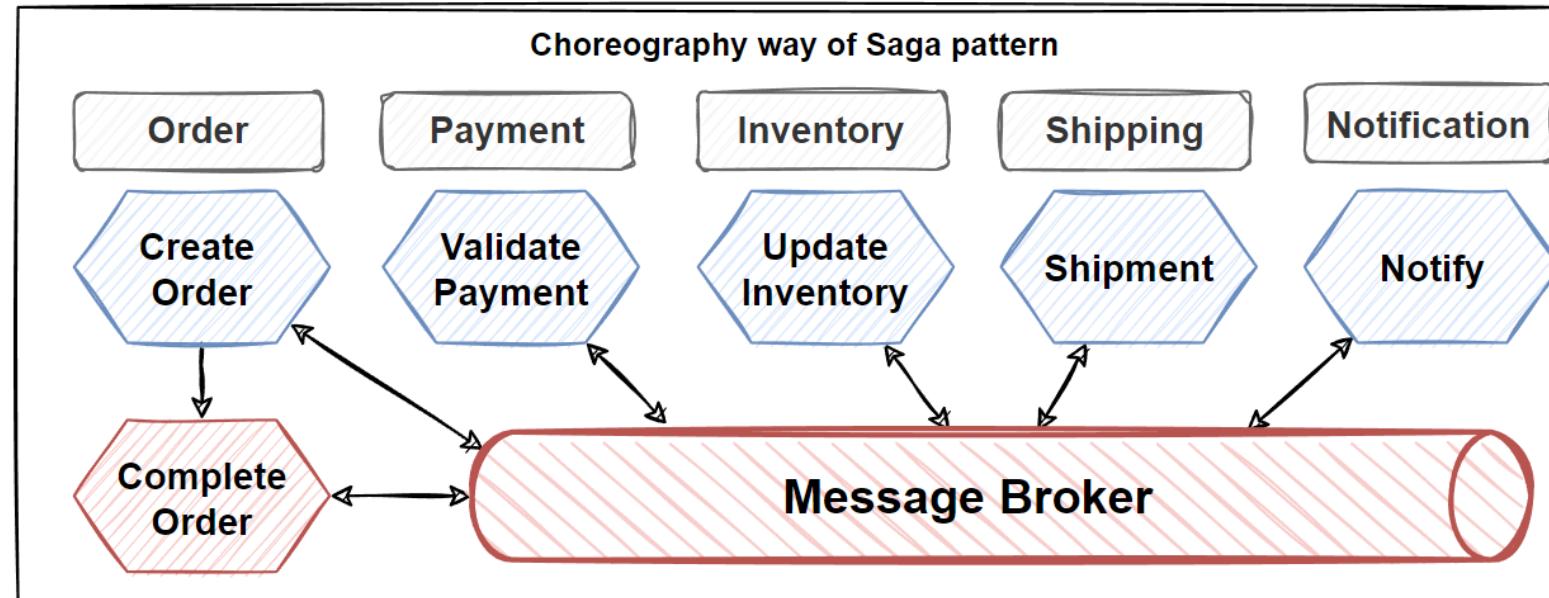
- Each microservice communicates with the other microservices by exchanging events using a message broker.
- Event-based approach allows for a more decentralized and flexible way to implement the SAGA pattern, as each microservice can publish to events to the others in real-time.
- Choreography provides to coordinate sagas with applying publish-subscribe principles.
- Each microservices run its own local transaction, publishes events to message broker system and trigger local transactions in other microservices.



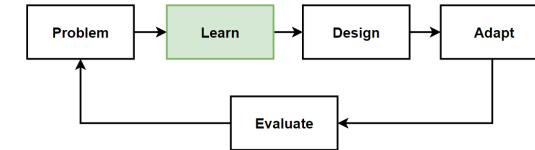
Order Fulfilment with Choreography-based SAGA Implementation



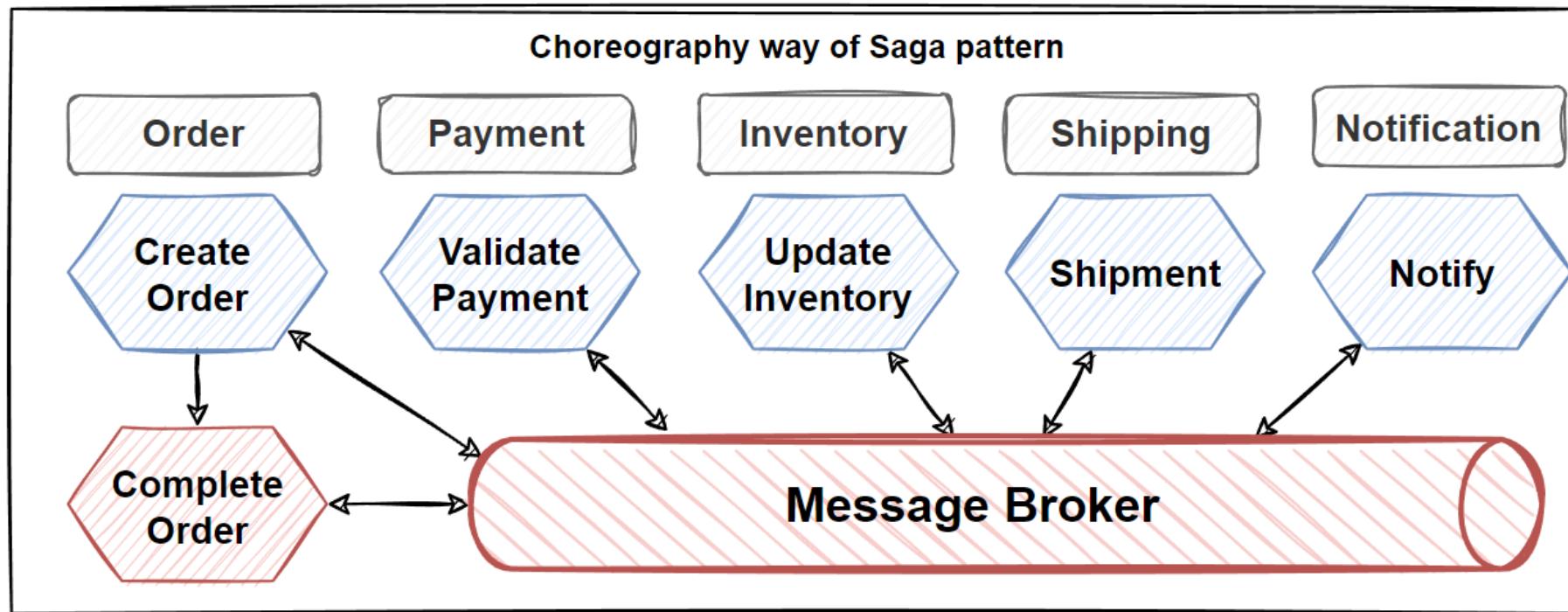
- The customer **places an order** on the e-commerce application and provides their payment information.
- The order microservice **begins a local transaction** and publishes an event to the message broker.
- The inventory microservice **listens for the event published** by the order microservice and, it begins a local transaction and reserves the items in the customer's order.
- The inventory microservice publishes an event to the message broker, the items were reserved successfully.
- The payment microservice **listens for the event** published by the inventory microservice and it charges the customer's payment method and commits its own local transaction.



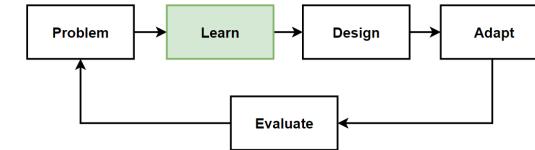
Order Fulfilment Rollback with Choreography-based SAGA Implementation



- SAGA pattern provides a way to **roll back** the **changes** made by each microservice.
- If the Inventory microservice encountered an error while reserving the items in the customer's order, it could **publish a failure event** to the message broker.
- Order fulfillment microservice would then **execute a compensating transaction** to undo the charges to the customer's payment method and cancel the order.



Benefits of Choreography-based SAGA



- **Decentralized and flexible**

By using an event-based approach, each microservice can react to events published with the others to coordinate their actions. This allows for a more decentralized and flexible approach to implementing the SAGA pattern.

- Decouple direct dependency of microservices when managing transactions.

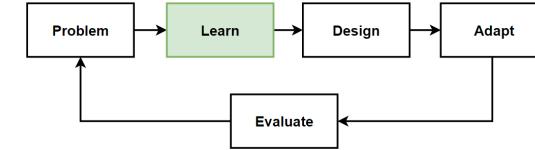
- **Avoid Single Point of Failure**

Since there is no orchestrator, responsibilities are distributed across the saga participants.

- **Simple workflows**

This way is good for simple workflows, if they don't require too much microservices transaction steps.

Drawbacks of Choreography-based SAGA



- **More complex to manage**

Each microservice communicates via events with the others, the choreography-based approach can be more complex to manage and may require more coordination among the microservices.

- SAGA workflow become confusing when adding new steps into flow.

- **Cyclic Event Consume Risk**

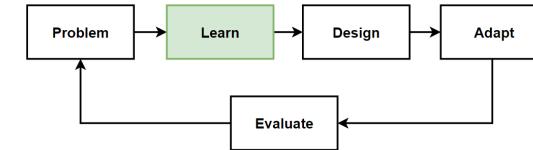
There's a Cyclic Event Consume Risk dependency between saga participants because they have to consume each other's commands.

- **Result**

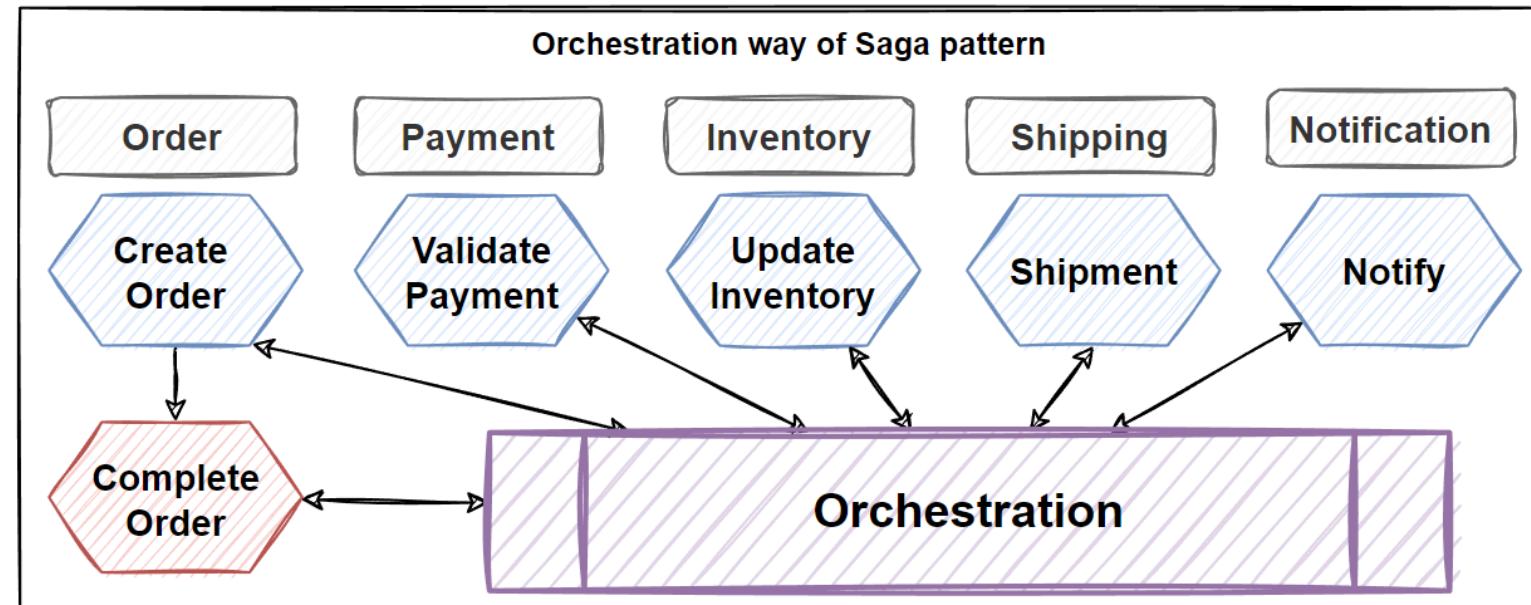
Implementing the SAGA pattern will depend on the specific needs and constraints of the distributed system. It may be a good fit for some systems, but may not be suitable for others.

- Choreography-based implementation of SAGA is good for simple workflows if they don't require too much microservices transaction steps.

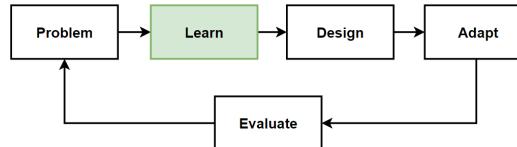
Orchestration-based SAGA Implementation



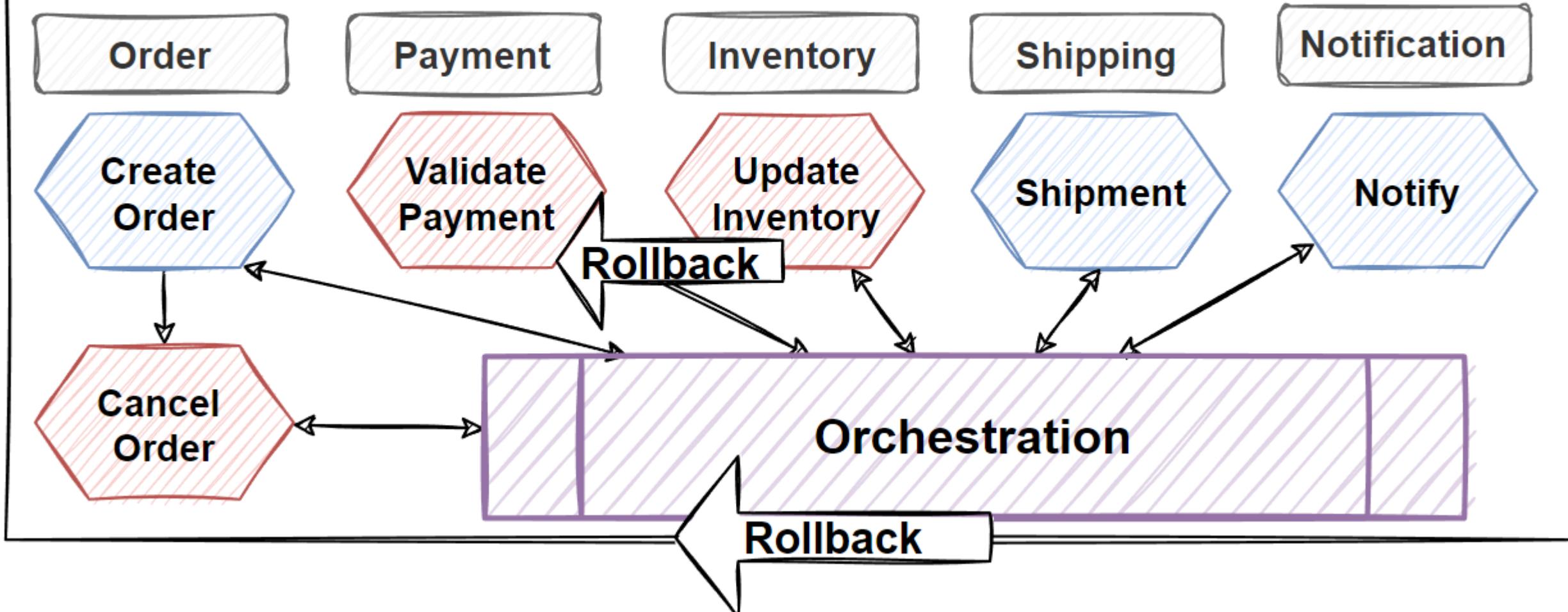
- **Orchestration-based SAGA pattern** involves using a **central orchestrator** service to **coordinate** and **manage** the **individual sagas** or microservices that make up a transaction.
- The orchestrator is responsible for **initiating the transaction** and **ensuring** that each **saga performs** its step in the **correct order**.
- **If any of the sagas fail** to complete their step, the **orchestrator** can use the **compensating transactions** to **roll back** the **changes** and **restore** the system to its **original state**.
- **Orchestration** provides to **coordinate sagas** with a **centralized controller** microservice that **orchestrate** the saga workflow and invoke to **execute local microservices transactions** in **sequentially**.



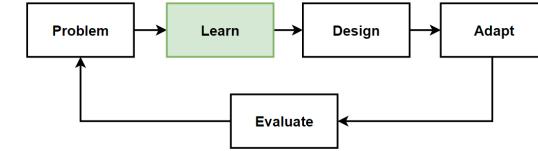
Orchestration-based SAGA Implementation - Rollback



Orchestration way of Saga pattern - Rollback



Benefits and Drawback of Orchestration-based SAGA Implementation



Benefits

- Provides a clear and centralized point of control for managing transactions.
- Make it easier to understand and debug the system, and to add new transactions or modify existing ones.

Drawbacks

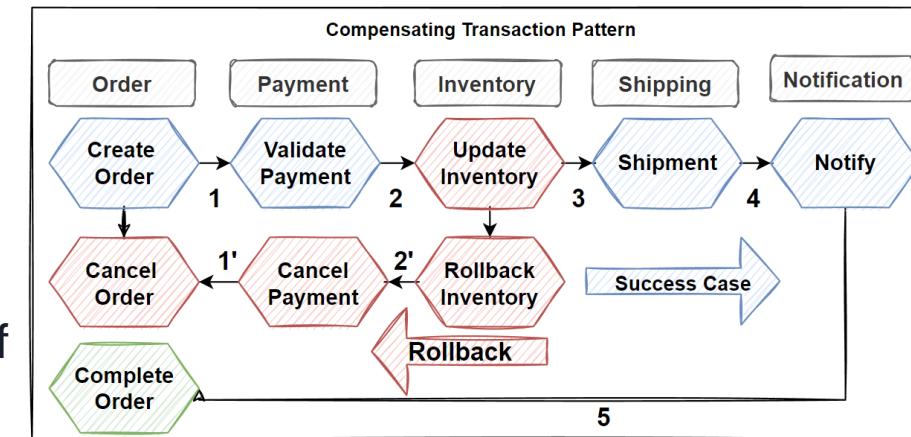
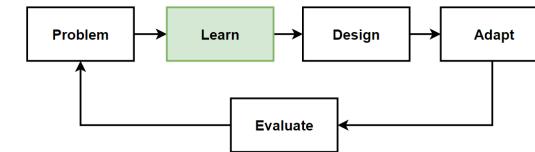
- The orchestrator can also become a single point of failure, and if it goes down, the entire system may be unable to complete transactions.
- The orchestrator can become a bottleneck if the system is heavily loaded, as all transactions must go through it.

Result

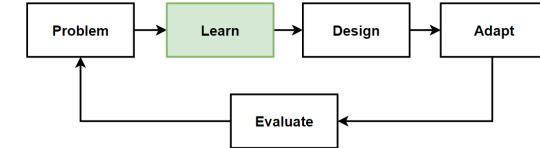
- It can be a useful approach for managing transactions in a distributed system, but it is important to carefully consider the trade-offs and potential drawbacks.
- Orchestration way is good for complex workflows which includes lots of steps.
- But this makes single point-of-failure with centralized controller microservices and need implementation of complex steps.

Compensating Transaction Pattern

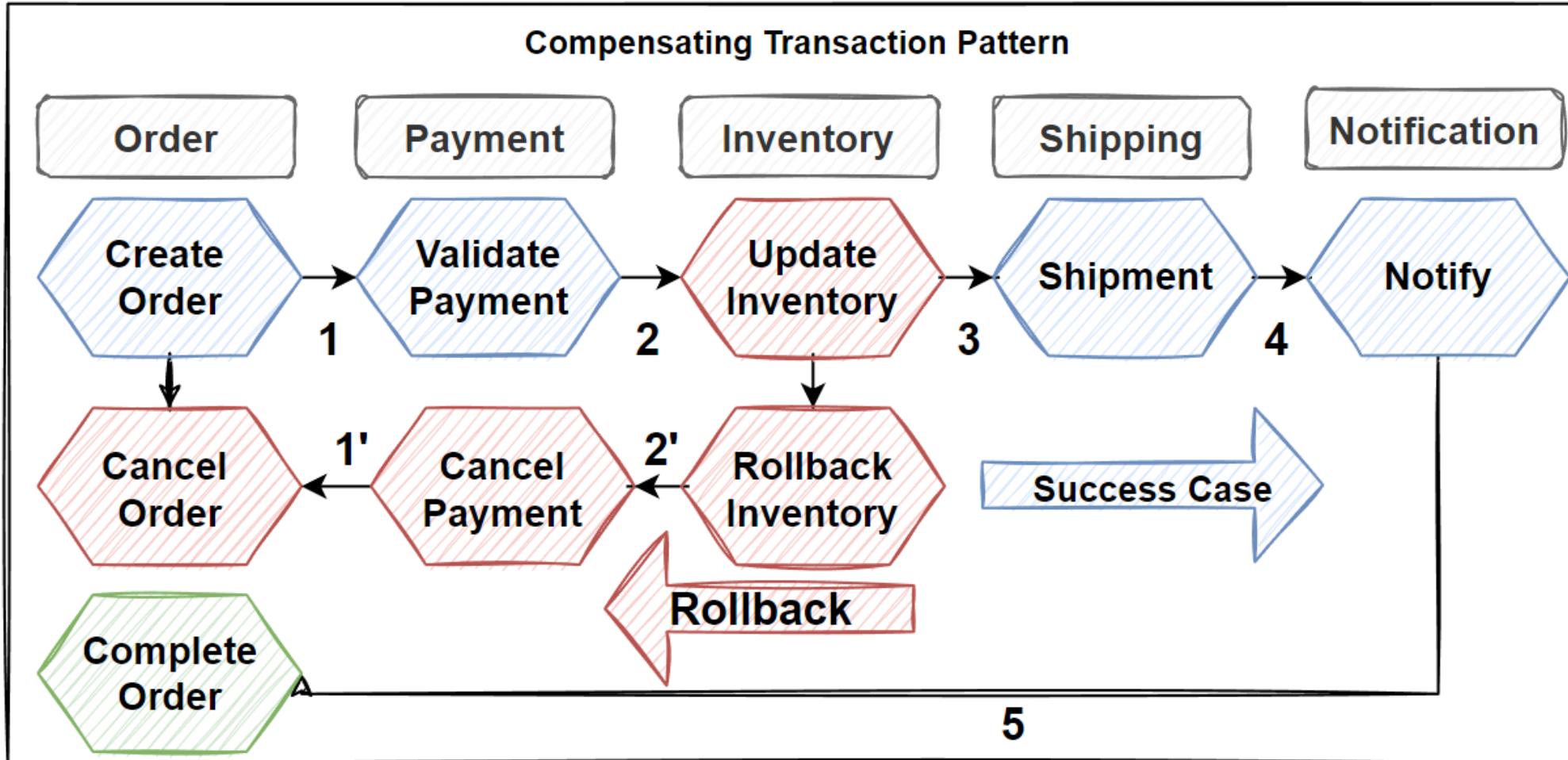
- Compensating Transaction pattern is a **rollback process** of SAGA Pattern.
- Compensating Transaction pattern provides to **reverse the steps** for a **previously executed transaction**.
- In microservice architectures where **multiple services** may be involved in a **distributed transaction**, if any of the services fail to complete their part of the transaction, the effects of the **entire transaction need to be undone**.
- The steps in a **compensating transaction** should **undo the effects** of the steps in the original operation.
- Compensating transaction is also an **eventually consistent** operation and it could be fail.
- The system should be able to **resume the compensating transaction** at the point of failure and continue.
- The steps in a compensating transaction should be defined as **idempotent commands**.



Ecommerce Order Fulfilment Compensating Transaction Pattern



- Customer places an order on the e-commerce application

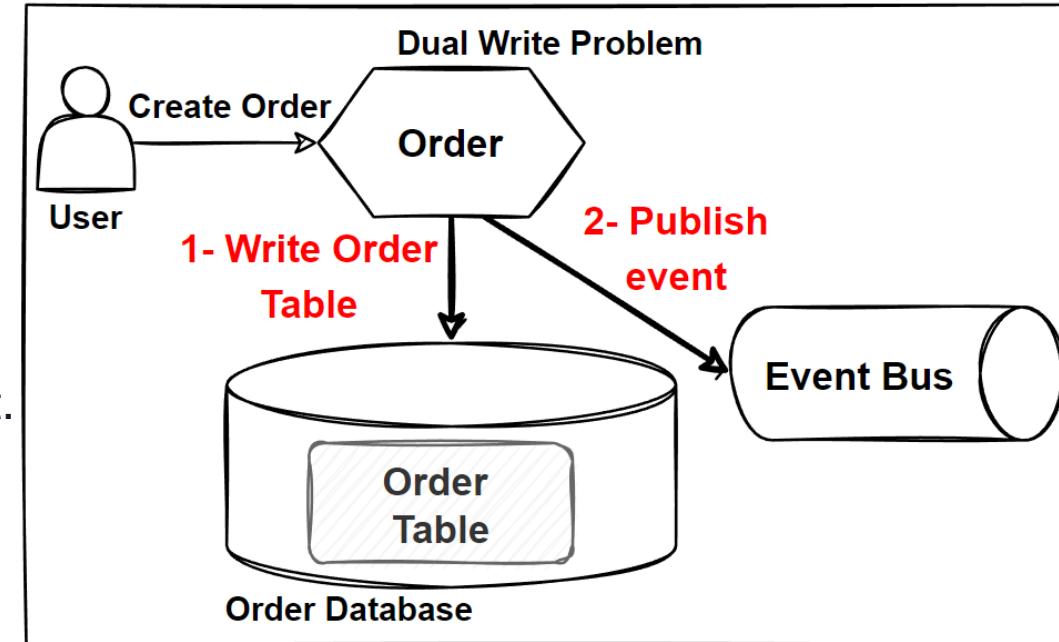
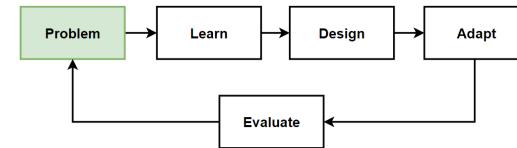


Problem: Dual Write Problem

- When application **needs to change data in two different systems**, i.e. a database and a message queue, if one of the writes fails, it can result in inconsistent data.
- Happens when you use a **local transaction** with **each of the external systems** operations.
- I.e. app needs to **persist data** in the **database** and **send a message to Kafka** for notifying other systems.
- **If one of these two operations fails**, the **data will be inconsistent** and these two systems becomes inconsistent.

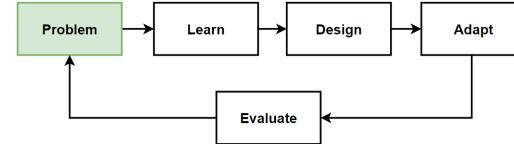
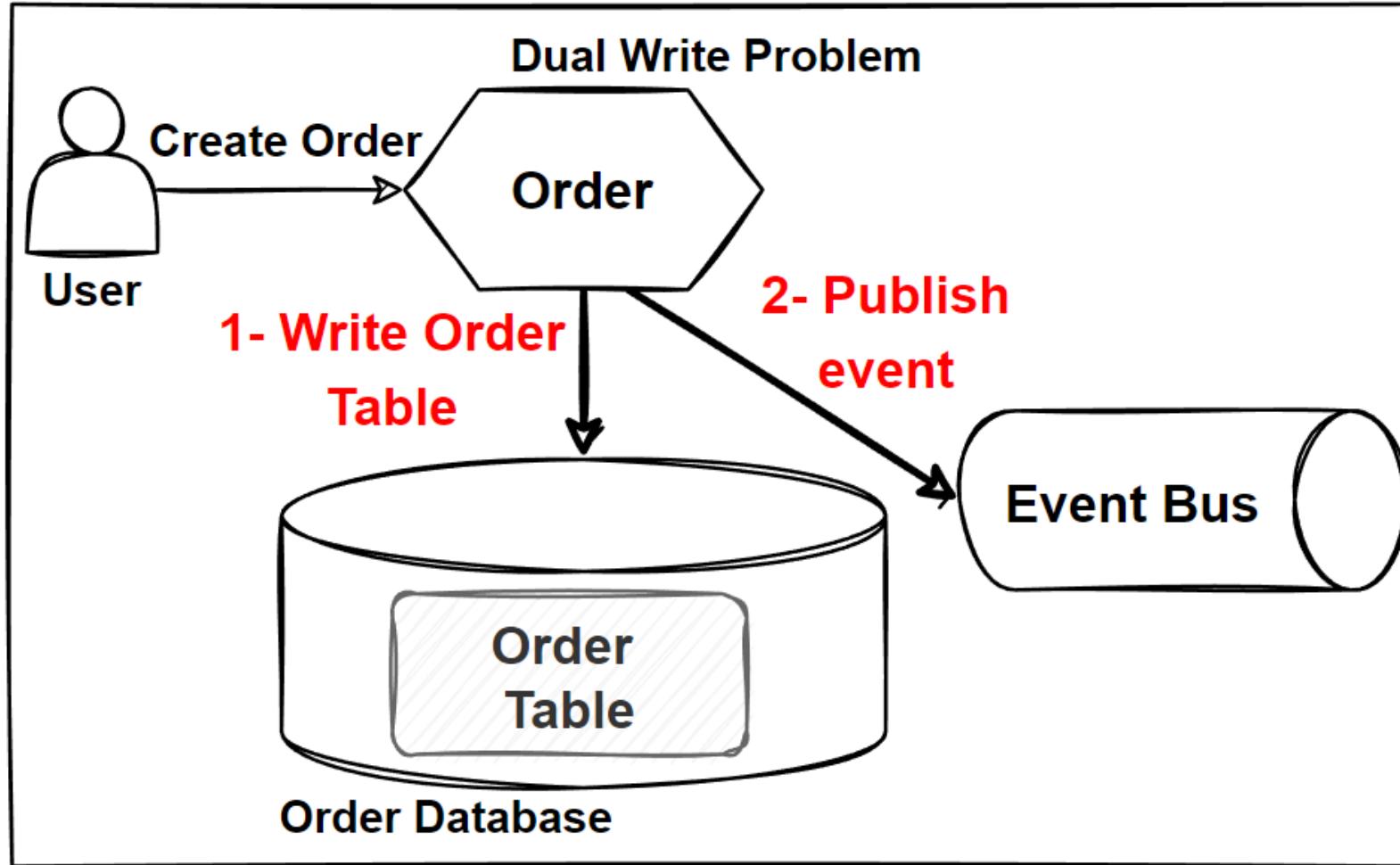
Problems

- Data loss or corruption.
- Difficult to resolve without proper error handling and recovery mechanisms.
- Dual writes can be hard to detect and fix.



Example of Dual-write problem: Create Order

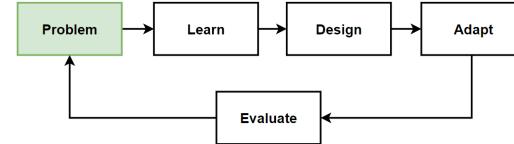
- 1- Change data in Order database with creating new record
- 2- Send an order_created event to the EventBus like Apache Kafka.



Run Transactions

- 1, 2 or
- 2, 1
- Both becomes dual writes

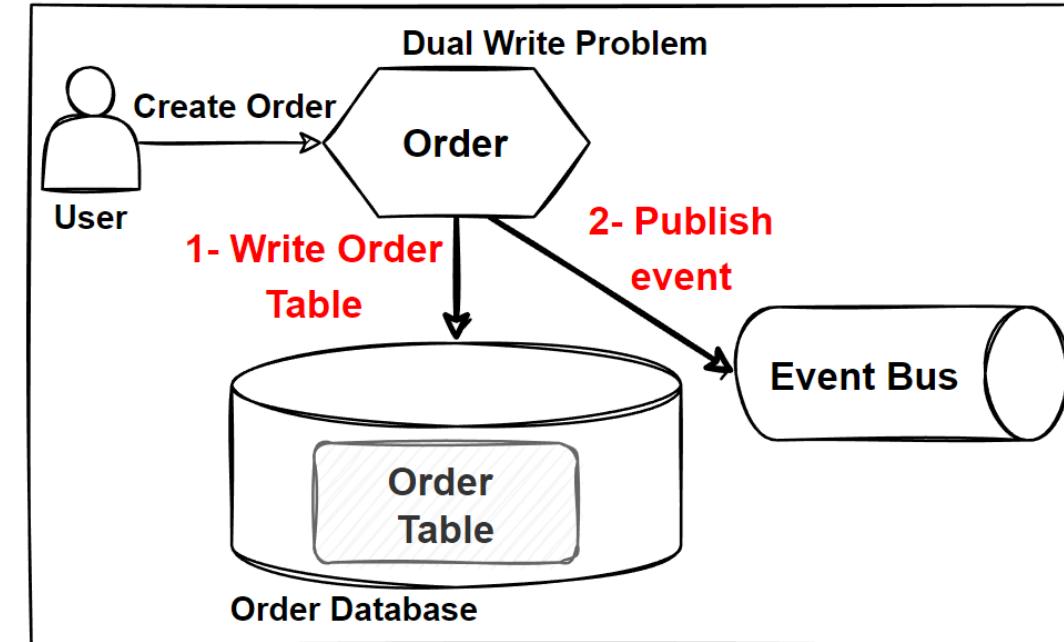
How to avoid dual write problems in microservices ?



- Monolith applications use the **2 phase commit** protocol.
- It **splits the commit process** of the transaction into 2 steps and ensures the **ACID principles** for all systems.
- **Can't use 2-phase commit** transactions when building **microservices**.
- These transactions **require locks** and **don't scale well**.
- Need all systems to be **up and running** at the same time.

Solutions

- Transactional Outbox Pattern
- CDC - Change Data Capture

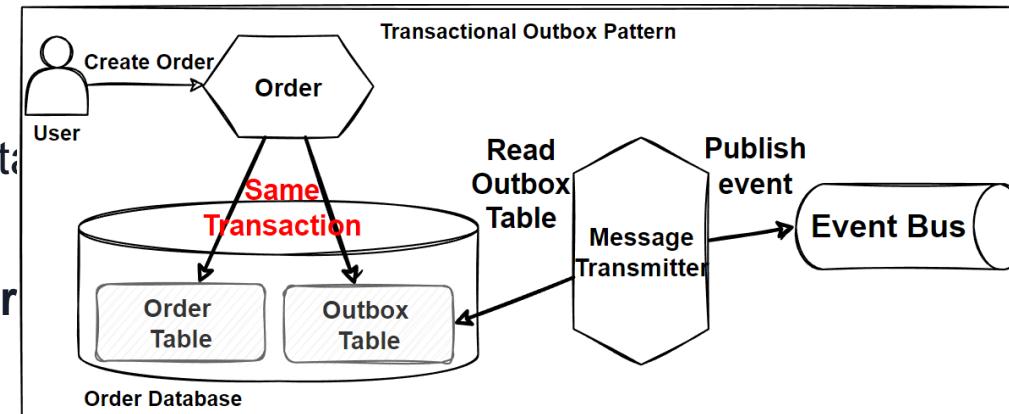
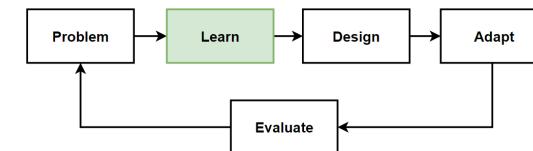


Best Practice

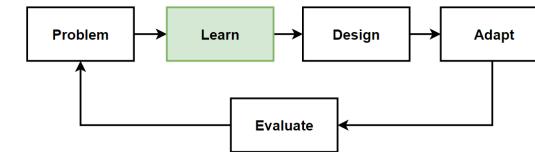
- Use Red Hat **Apache Kafka** and **CDC** using **Debezium** in event-driven applications.
- Use New databases like **CockroachDB** which has built-in **Change Data Capture** feature.

Transactional Outbox Pattern

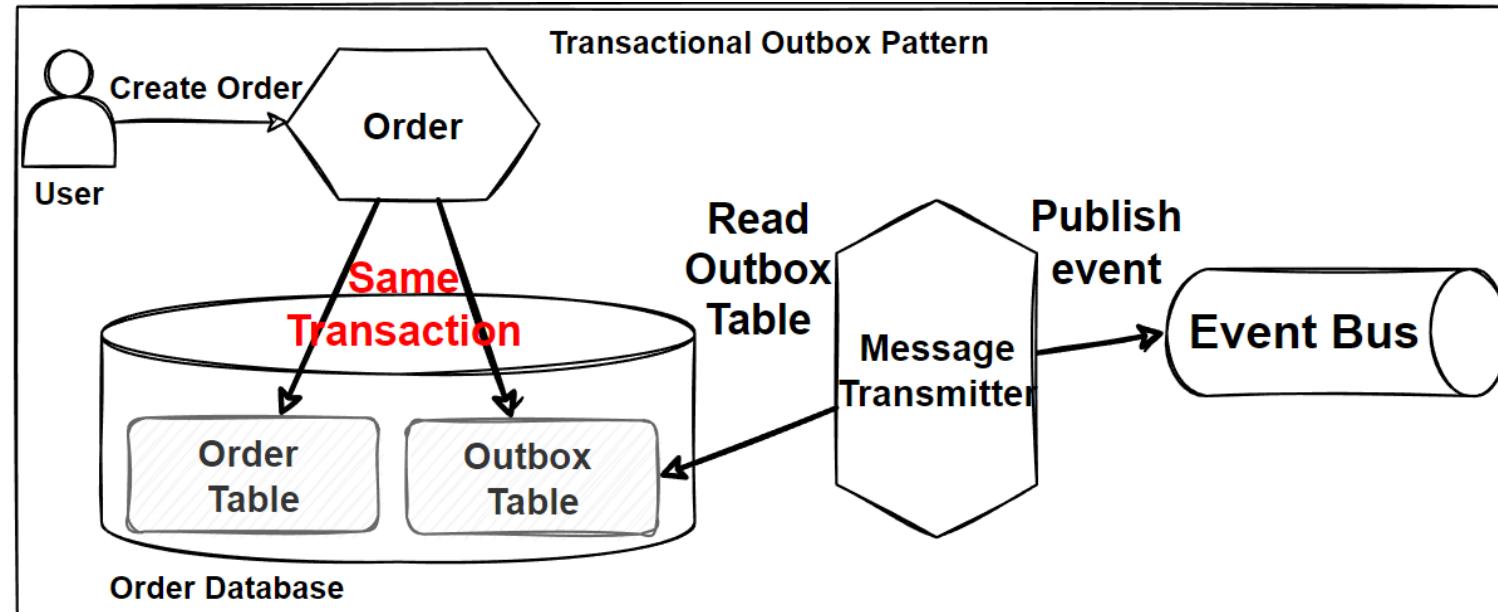
- The idea is to have an “**Outbox**” table in the microservice’s database. It provides to publish events reliably.
- **Dual write problem** happens when application needs to change data in two different systems.
- Instead of sending the data to two separate locations, **send a single transaction** that will **store two separate copies** of the data on the database.
- **One copy is stored** in the relevant **database table**, and the **other copy is stored** in an **outbox table** that will publish to event bus.
- When API **publishes event messages**, it doesn’t directly send them, Instead, the **messages are persisted** in a **database table**.
- After that, **a job publish events** to message broker system in predefined time intervals.
- **Events are not written directly** to a event bus, it is written to a table in the “**Outbox**” role of the service.



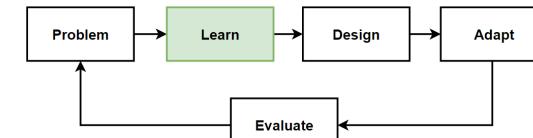
Transactional Outbox Pattern - 2



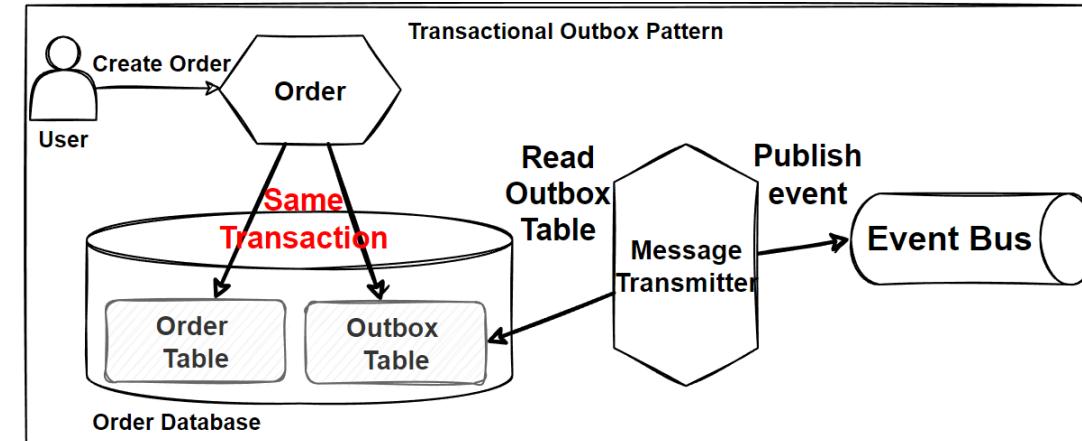
- Transaction **performed before the event** and the **event written** to the **outbox table** are part of the **same transaction**.
- When a **new order is added** to the system, the process of adding the order and writing the **Order_Created event** to the **Outbox table** is done **in the same transaction** to ensure the event is saved to the database.
- If one of the **process is fail**, this will **rollback the whole operations** with following ACID principles.
- The second step is to **receive these events** written to the **Outbox table** by an **independent service** and **write** them to the **Event bus**. Another service **listen** and **polls** the **Outbox table** records and publish events.



Transactional Outbox Pattern in Microservices

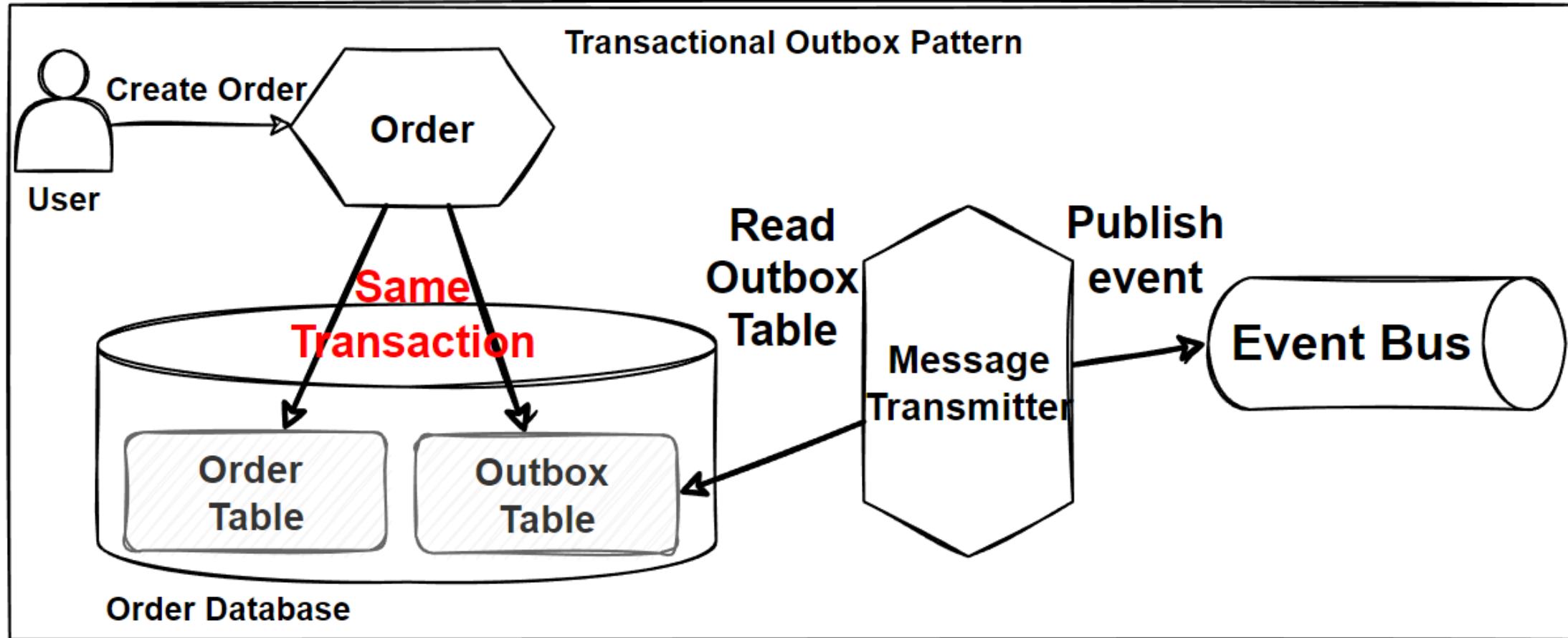
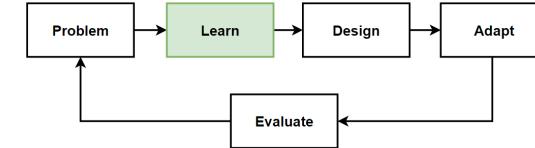


- **Microservice** provides an **outbox table** within its **database**.
Outbox table will **include** all the **events**.
- There will be a **CDC (change data capture) plugin** that **reads** the **commit log** of the **outbox table** and **publish the events** to the relevant queue.
- It provides that **messages** are **reliably delivered** from a **microservice** to **another microservice** even if the transaction that triggered the message fails.
- It involves **storing the message** in a local "**Outbox**" **table** within the microservice, that message sent to the consumer after the transaction is committed.
- **Outbox pattern** can be used to ensure that **messages** are **delivered consistently**, even if the microservice that sent the message is **unavailable** or **experiencing errors**.
- Useful for **communicating important information** or updates between services.



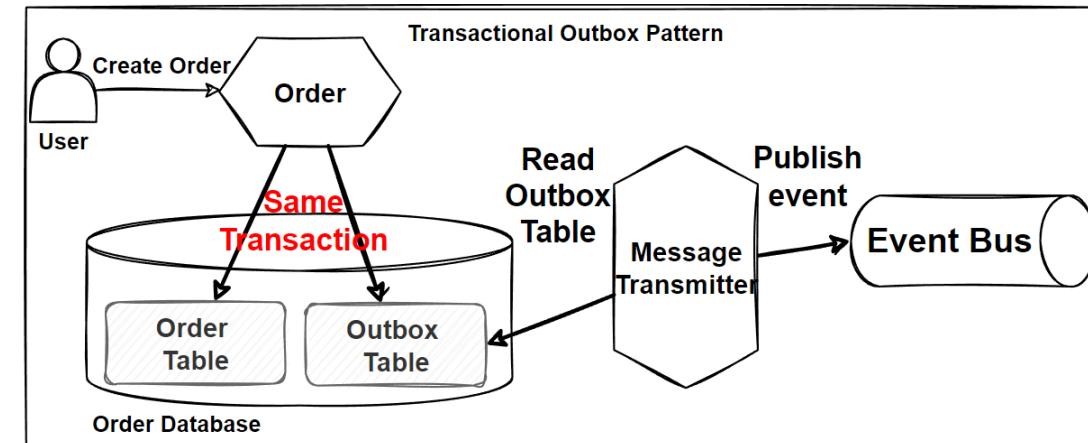
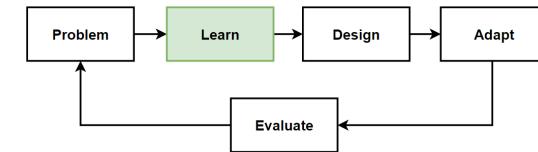
Customer Places an Order Use Case

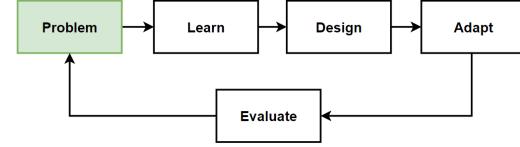
Transactional Outbox pattern in Microservices



Why Use Outbox Pattern ?

- When **working with critical data** that **need to consistent** and **need to accurate** to catch all requests.
- When the **database update** and **sending of the message** should be **atomic** to make sure **data consistency**.
- For example the **order sale transactions**, because they are about **financial business**. Thus, the calculations **must be correct 100%**.
- To access this **accuracy**, must be sure that our system is not **losing any event messages**.
- **The Outbox Pattern** should be applied this kind of cases.



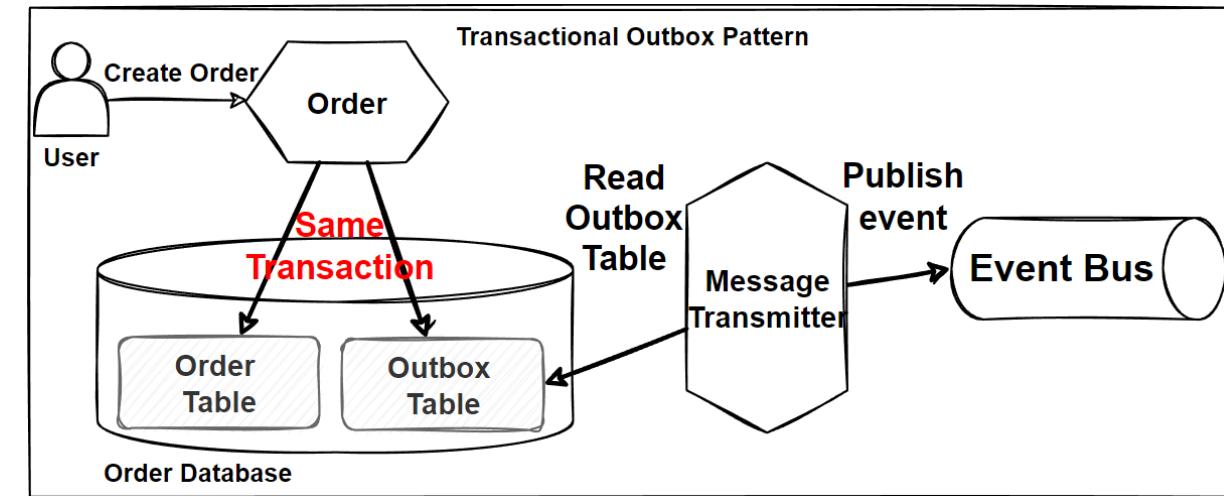


Problem: Listen and Polling Outbox Table

- **Creating new order record** into order table and **creating new order_created event** into outbox table **are in the same transactions**.
- Another service **listen** and **polls** the **outbox table records** and publish events into separate processes.
- The current architecture **required additional microservices** that listen and polls records from the outbox table.

Problems

- Reduces the performance and error-prone operations.
- Time is wasted during Polling, wasted resources.
- Consuming system resource unnecessary pull requests.
- Come with a latency overhead, reach system limitations.
- Can cause additional Dual-write problem.

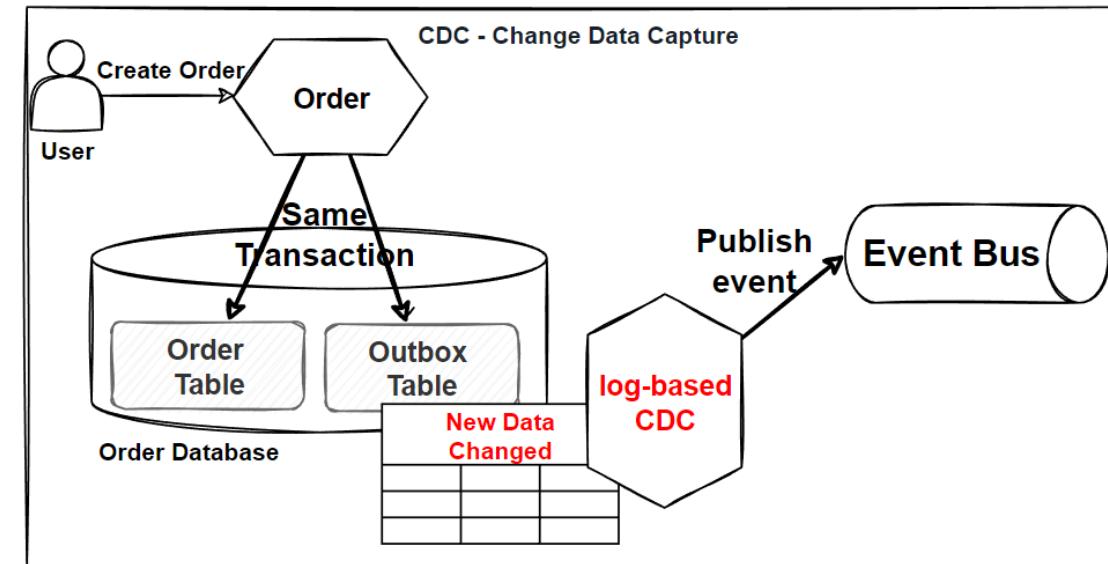
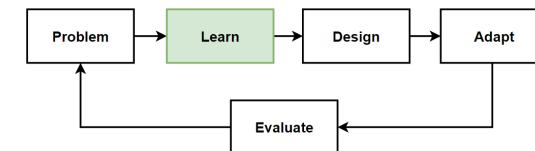


How solve these problems when applied outbox pattern ?

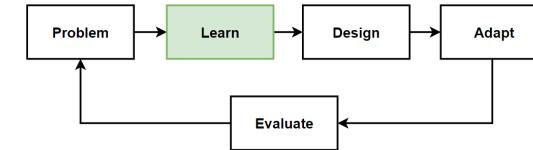
- CDC - Change Data Capture with Outbox Pattern.

CDC - Change Data Capture

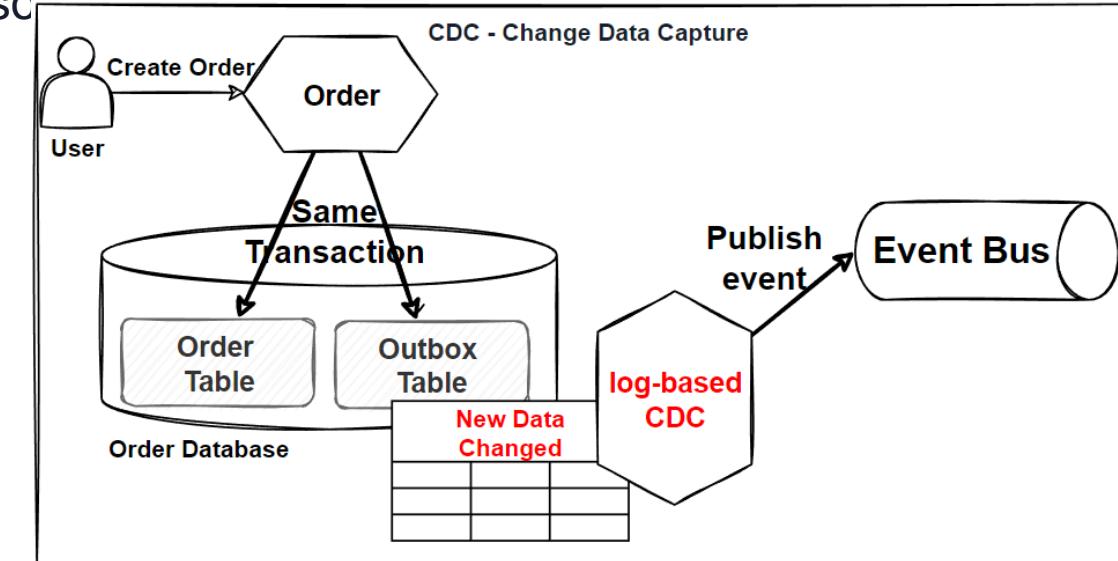
- **Change Data Capture (CDC)** is a technology that captures insert, update, and delete activity on a database.
- **CDC** typically works by **continuously monitoring** the **transaction log** of a database for **changes**, and then **extracting and propagating** those **changes** to the target system.
- This allows the target system to **stay up-to-date** with the source system **in near real-time**, instead of relying on batch-based data synchronization processes.
- **CDC** can be used in **replicating data between databases**, **synchronizing data** between systems in a microservices architecture, and **enabling real-time data analytics**.
- **CDC** is a way to **track changes** that happen to data in a database that **captures insert, update, and delete activity** and makes this information available to other systems.
- This allows those systems to **stay up-to-date** with the data in the **database in real-time**.



CDC - Change Data Capture with Outbox Pattern

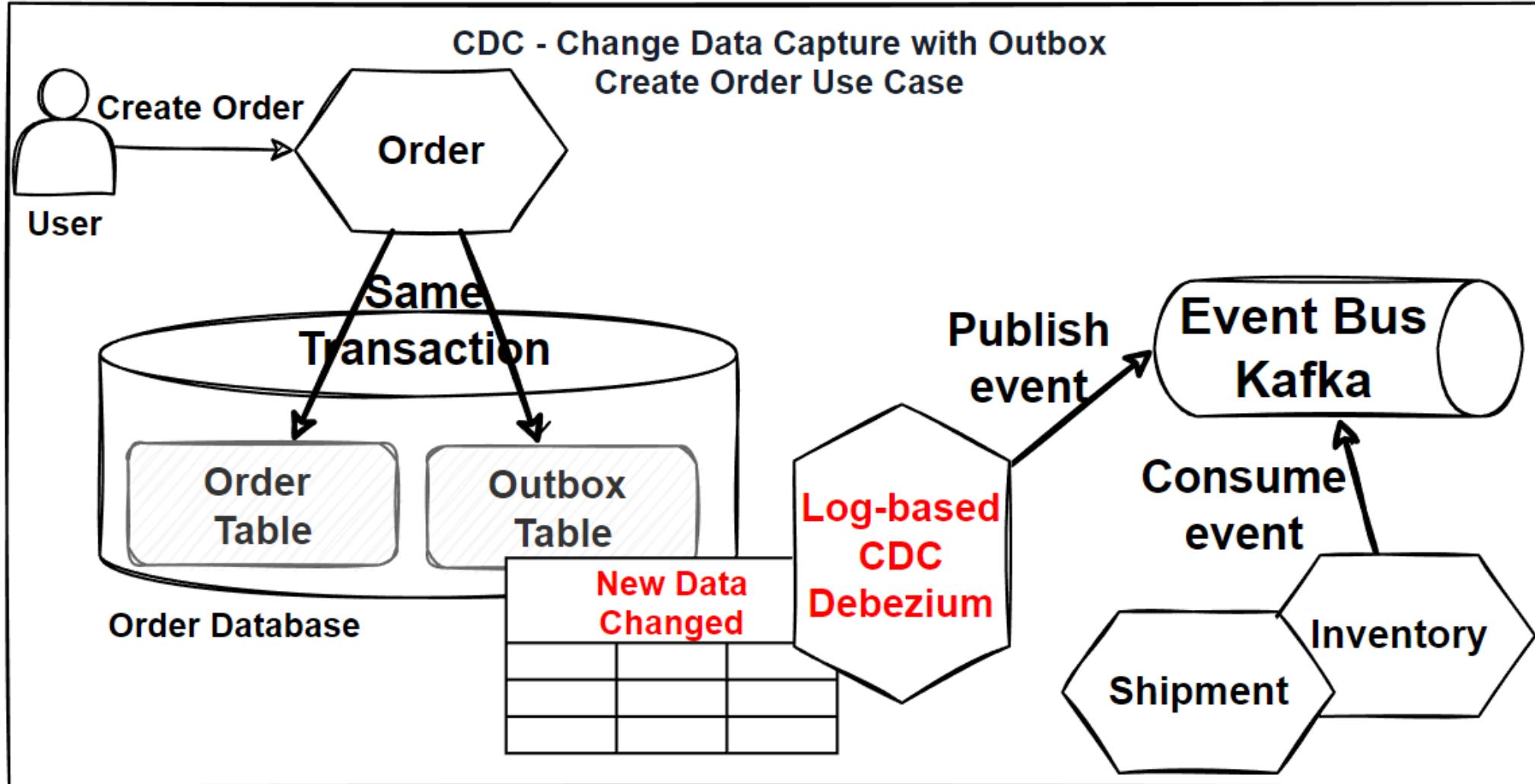
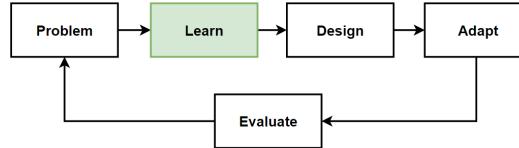


- **Outbox pattern** is ensuring **data changes** made by a microservice are **eventually propagated** to other microservices.
- Whenever a microservice **updates data** in its database, it also writes a **record to the outbox table** with the details of the change.
- **CDC** can then be used to **monitor** the **outbox table** for new records, **extract the data changes** that propagated to the **target microservices** to be **kept up-to-date** with the data.
- Using **CDC** with the **Outbox pattern** allows microservices to **decouple** their **data updates** from the **process** of **propagating** those **updates** to other microservices.
- This can make it **easier to scale** and **maintain** a microservices architecture.
- Each microservice can **focus on its own data updates** and let **CDC handle the synchronization** of data between services.



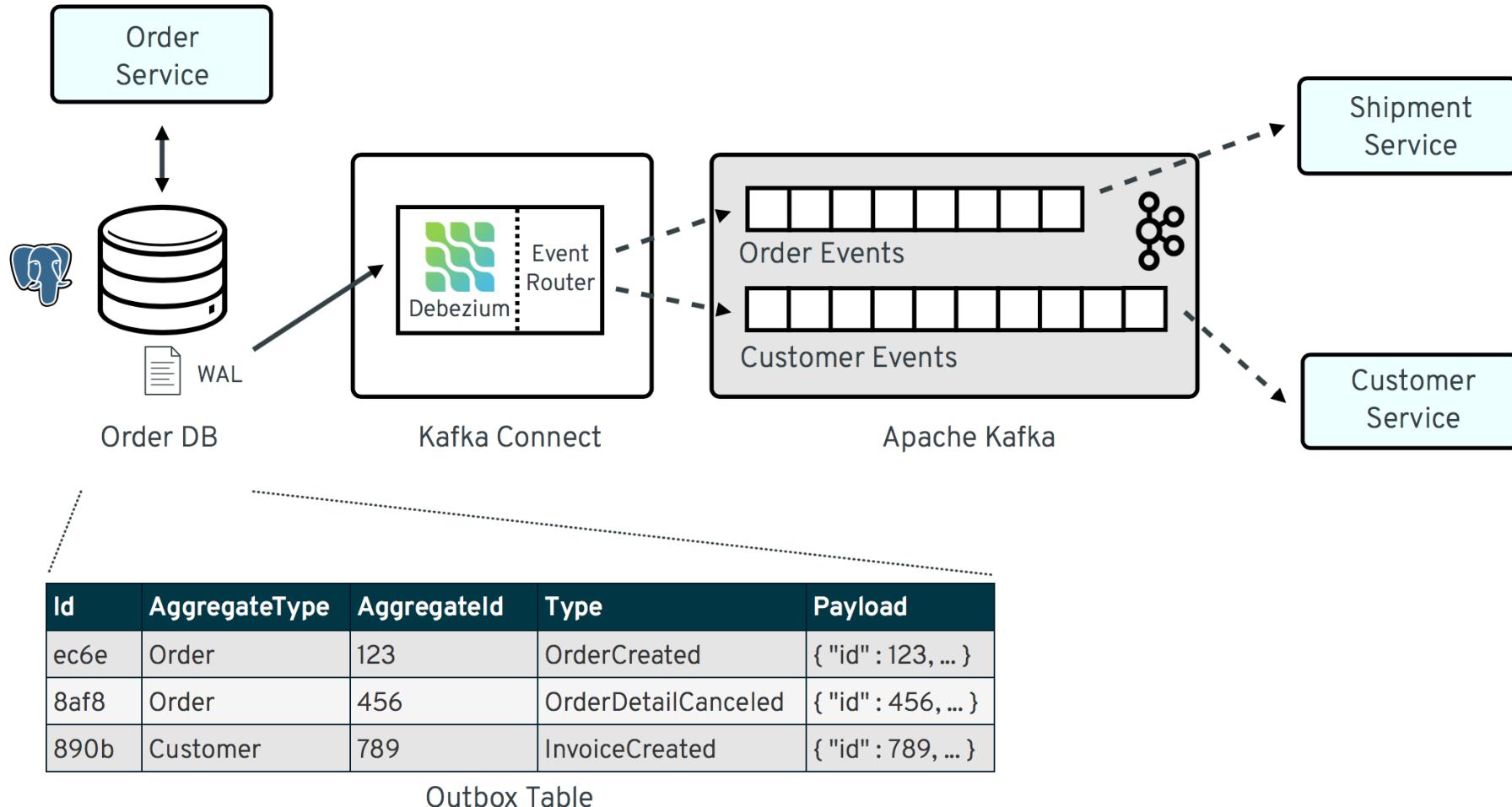
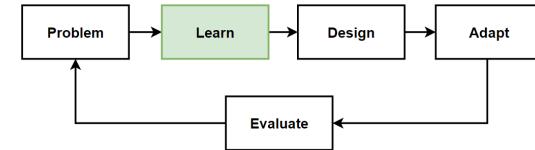
CDC - Change Data Capture with Outbox Pattern

E-commerce Create Order Use Case



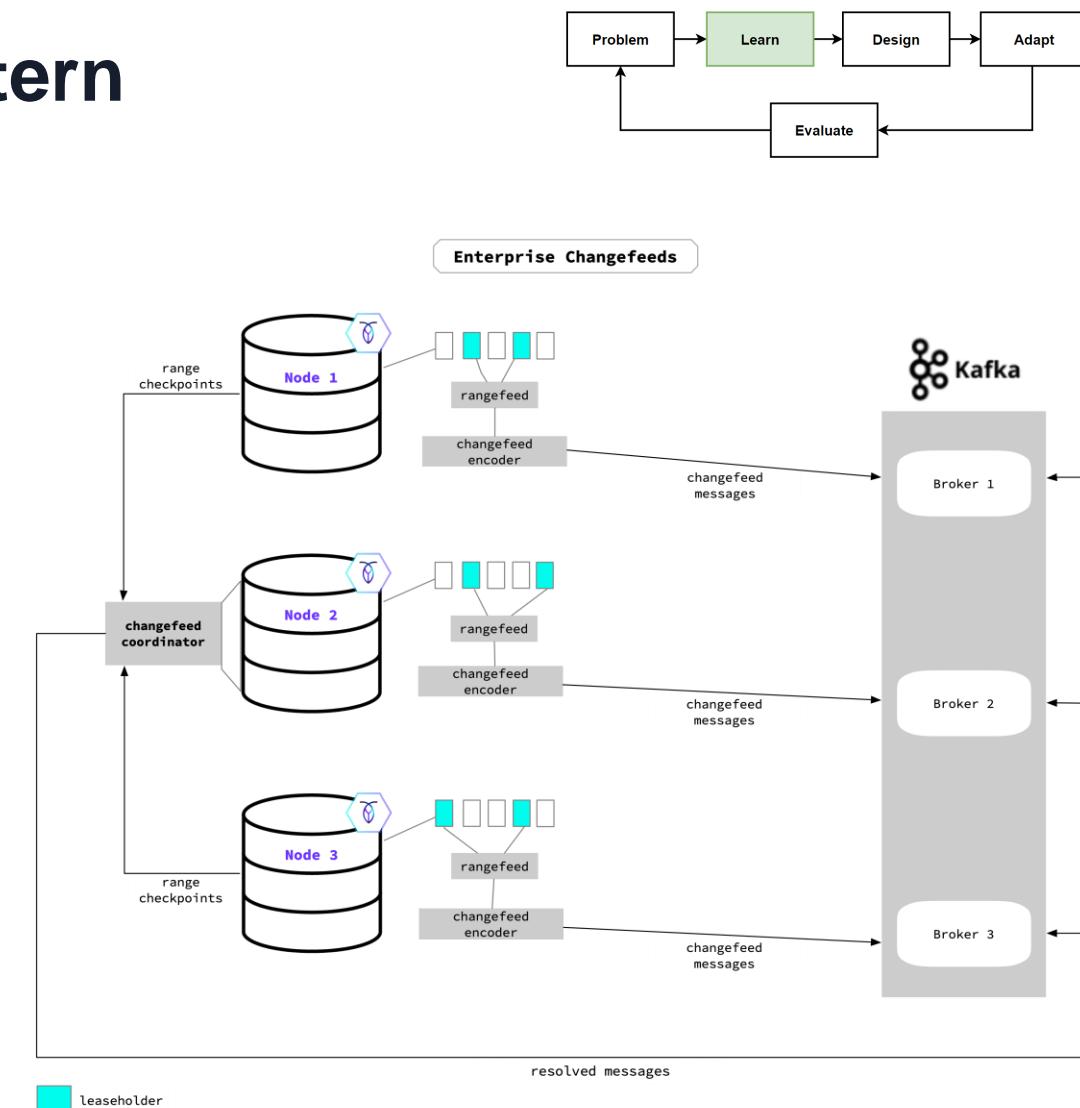
CDC - Change Data Capture with Outbox Pattern

E-commerce Create Order Use Case



CockroachDB for CDC and Outbox Pattern

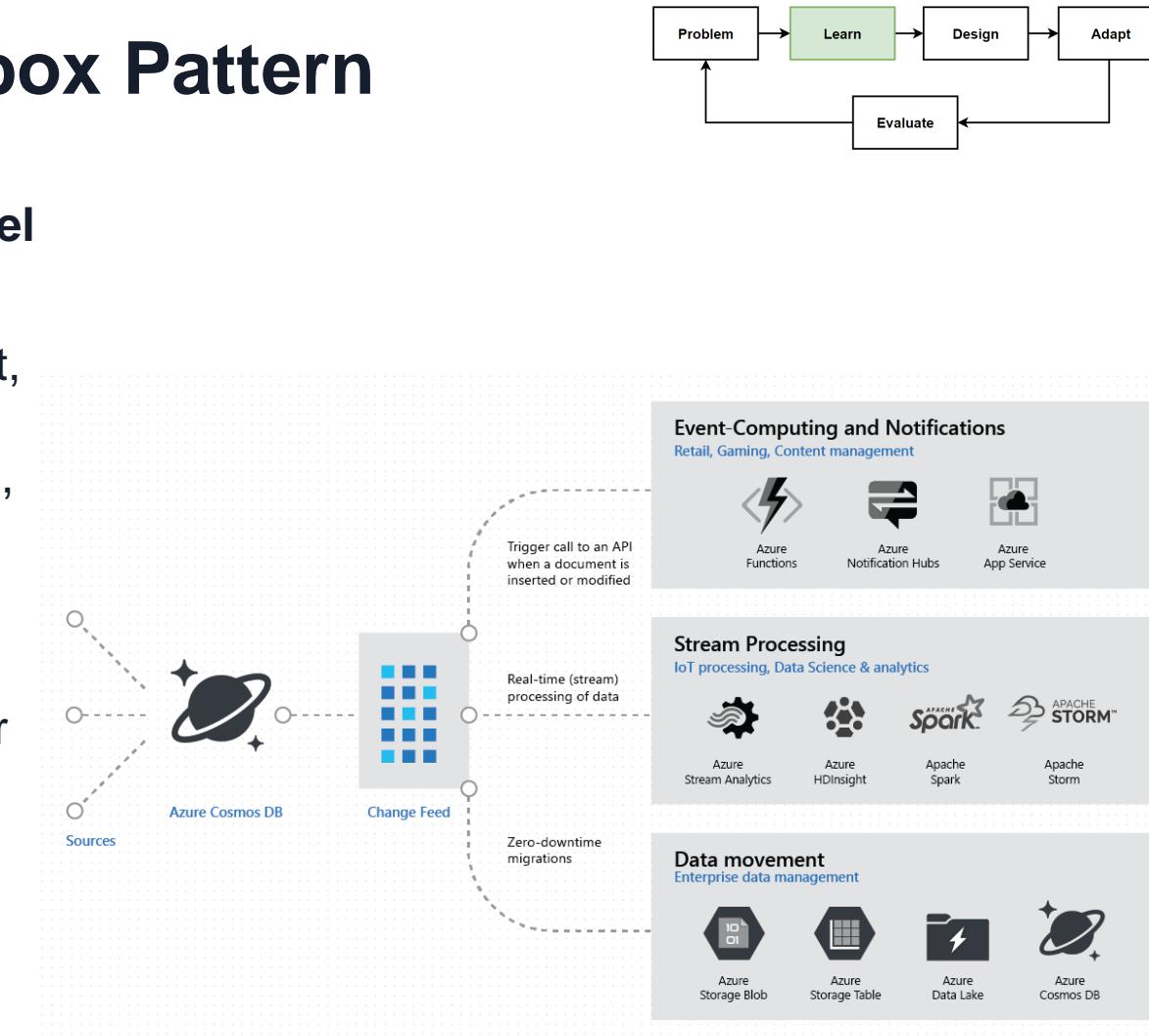
- CockroachDB is a **distributed database** management system that is designed to be **scalable, resilient**, and easy to use.
- **Cockroach cluster**, which is a **group of database nodes** that work together to form a single, highly available database.
- Ability to **scale horizontally** by adding more nodes to the cluster as the **workload increases**.
- CockroachDB also has strong support for **data consistency** and **durability**, with features such as **multi-active availability** and **distributed transactions**.
- CockroachDB is written in the **Go programming** language.
- CockroachDB has built-in Change Data Capture feature, that you can build the **Transactional Outbox Pattern** with **CDC** into your own application.



<https://www.cockroachlabs.com/docs/stable/change-data-capture-overview.html>

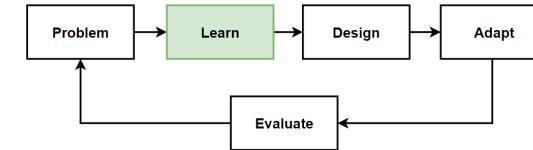
Azure Cosmos DB for CDC and Outbox Pattern

- Azure Cosmos DB is a **globally distributed, multi-model** database service offered by **Microsoft Azure**.
- It supports various **database models**, including document, **key-value**, **column-family**, and **graph**, and can be accessed through multiple APIs, such as **SQL**, **MongoDB**, **Cassandra**, and **Azure Table Storage**.
- Azure Cosmos DB has built-in support for **Change Data Capture (CDC)**, allows it to **track and propagate data changes** made to the **database to other systems** in near real-time.
- CDC in Azure Cosmos DB works by **continuously monitoring the transaction logs** of the database for **changes** and extracting those changes to be sent to target systems.
- Azure Cosmos DB change feed API to access the data changes and process them in your application.

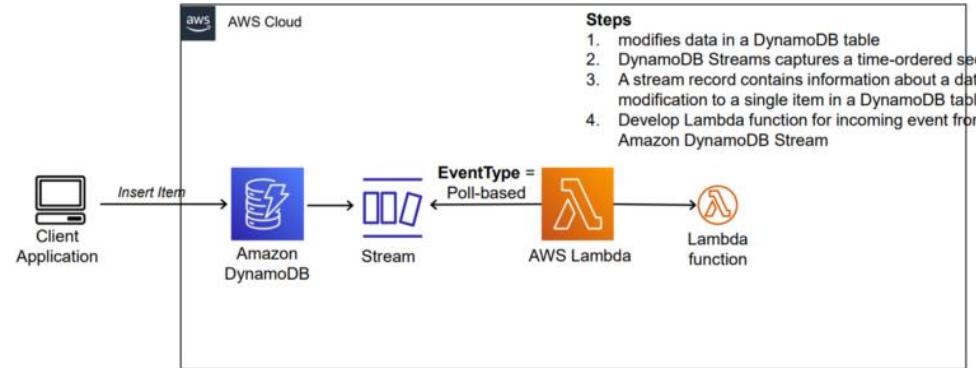


<https://learn.microsoft.com/en-us/azure/cosmos-db/nosql/change-feed-design-patterns>

Amazon DynamoDB Streams for CDC and Outbox

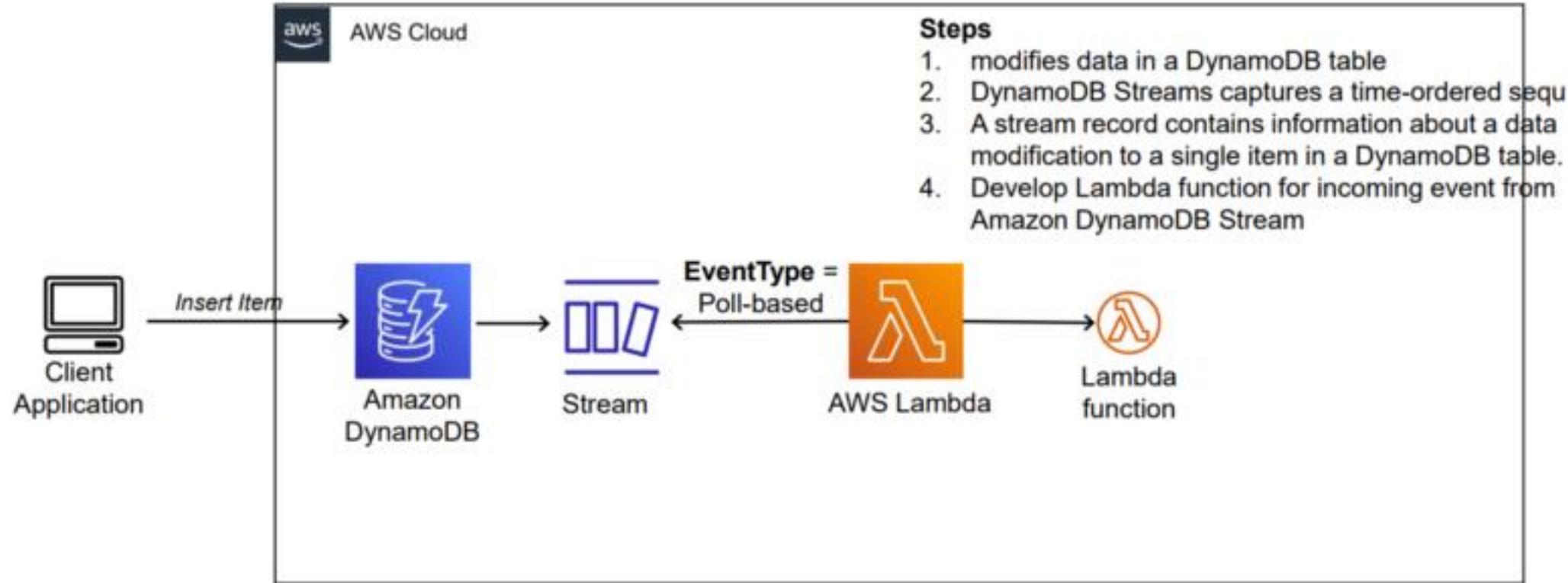
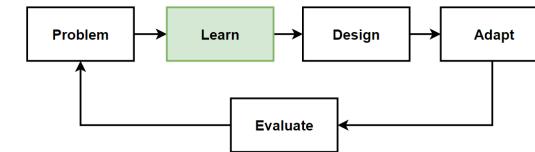


- **DynamoDB** supports **streaming of item-level change data capture records in the near-real time.**
- **DynamoDB stream** is an **ordered flow of information** about changes to items in a **DynamoDB table**.
- Whenever an application **creates, updates, or deletes items** in the **table**, **DynamoDB Streams writes a stream record** with the **primary key attributes** of the items that were modified.
- A **stream record** contains information about a **data modification** to a **single item** in a **DynamoDB table**. That can includes capture additional information, such as the “**before**” and “**after**” **images** of modified items.
- **DynamoDB Streams** writes **stream records** in near-real time so that you can build applications that **consume these streams** and **take action** based on the contents.
- Most of the applications can **benefit** from **data capturing changes** into DynamoDB table; **Notifications, Mobile Apps, Financial Apps.**



<https://medium.com/aws-lambda-serverless-developer-guide-with-hands/dynamodb-streams-using-aws-lambda-to-process-dynamodb-streams-for-change-data-capture-2e3ab8df27ca>

Amazon DynamoDB Streams for CDC and Outbox



<https://medium.com/aws-lambda-serverless-developer-guide-with-hands/dynamodb-streams-using-aws-lambda-to-process-dynamodb-streams-for-change-data-capture-2e3ab8df27ca>

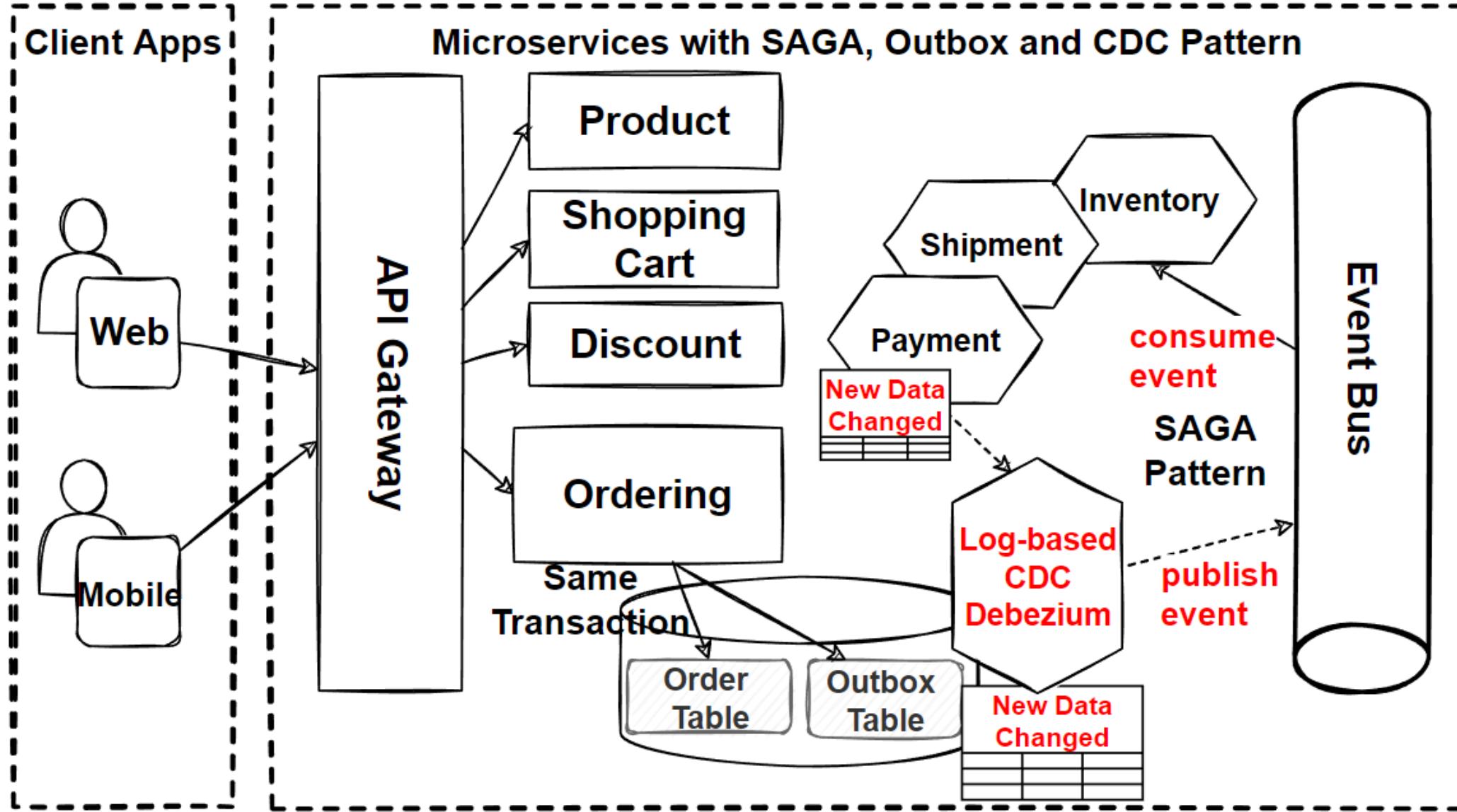
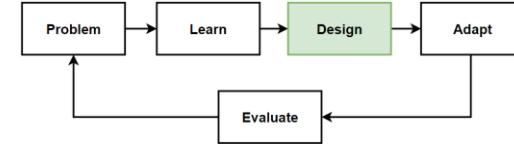
Before Design – What we have in our design toolbox ? - Data

Architectures Patterns&Principles	Microservices Data Choosing Database	Microservices Data Commands&Queries	FR
<ul style="list-style-type: none">• Microservices Architecture• The Database-per-Service Pattern• Polygot Persistence• Decompose services by scalability• The Scale Cube• Microservices Decomposition Pattern• Microservices Communications Patterns• Microservices Data Management Patterns	<ul style="list-style-type: none">• The Shared Database Anti-pattern• Relational and NoSQL Databases• CAP Theorem–Consistency, Availability, Partition Tolerance• Data Partitioning: Horizontal, Vertical and Functional Data Partitioning• Database Sharding Pattern	<ul style="list-style-type: none">• Materialized View Pattern• CQRS Design Pattern• Event Sourcing Pattern• Eventual Consistency Principle	<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history
			Non-FR <ul style="list-style-type: none">• High Scalability• High Availability• Millions of Concurrent User• Independent

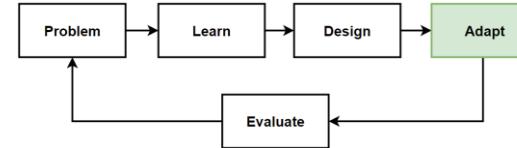
Before Design – What we have in our design toolbox ? - New

Architectures Patterns&Principles	Microservices Data Choosing Database	Microservices Distributed Transactions	FR
<ul style="list-style-type: none">• Microservices Architecture• The Database-per-Service Pattern• Polygot Persistence• Decompose services by scalability• The Scale Cube• Microservices Decomposition Pattern• Microservices Communications Patterns• Microservices Data Management Patterns• Microservices Distributed Transaction Pattern	<ul style="list-style-type: none">• The Shared Database Anti-pattern, Relational and NoSQL Databases• CAP Theorem–Consistency, Availability, Partition Tolerance• Data Partitioning: Horizontal, Vertical and Functional Data Partitioning• Database Sharding Pattern	<ul style="list-style-type: none">• SAGA Pattern• Choreography and Orchestration-based SAGA• Compensating Transaction Pattern• Dual-Write Problem• Transactional Outbox Pattern• CDC - Change Data Capture	<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history
	Microservices Data Commands&Queries <ul style="list-style-type: none">• Materialized View Pattern• CQRS Design Pattern• Event Sourcing Pattern• Eventual Consistency		Non-FR <ul style="list-style-type: none">• High Scalability• High Availability• Millions of Concurrent User• Independent

Microservices with SAGA, Transactional Outbox and CDC Pattern



Adapt: Microservice Architecture with SAGA, Transactional Outbox and CDC Pattern



Frontend SPAs

- Angular
- Vue
- React

API Gateways

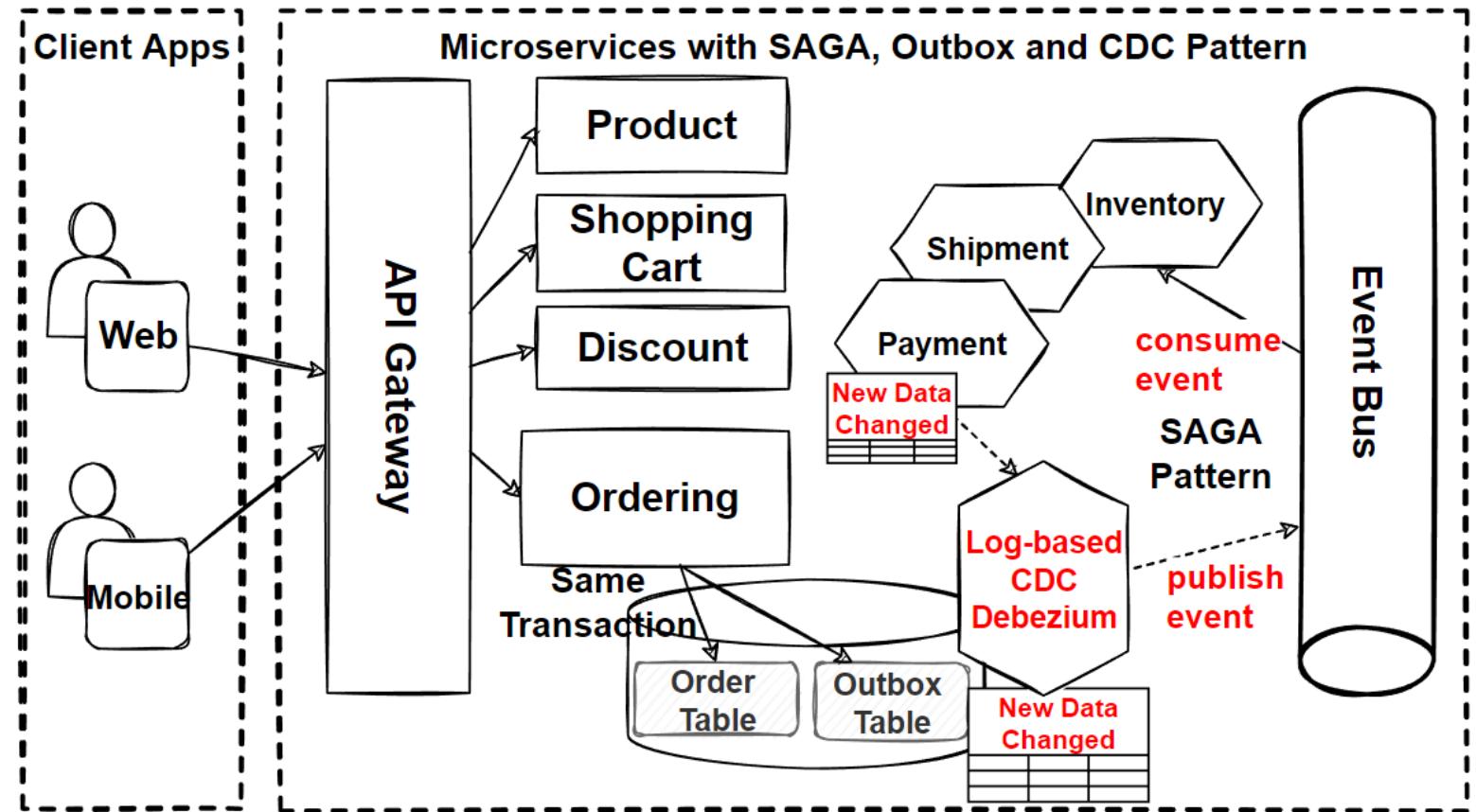
- Kong Gateway
- Express Gateway
- Amazon AWS API Gateway

Message Brokers Event Bus

- Kafka
- RabbitMQ
- Amazon EventBridge, SNS

Backend Microservices

- Java – Spring Boot
- .Net – Asp.net
- JS – NodeJS



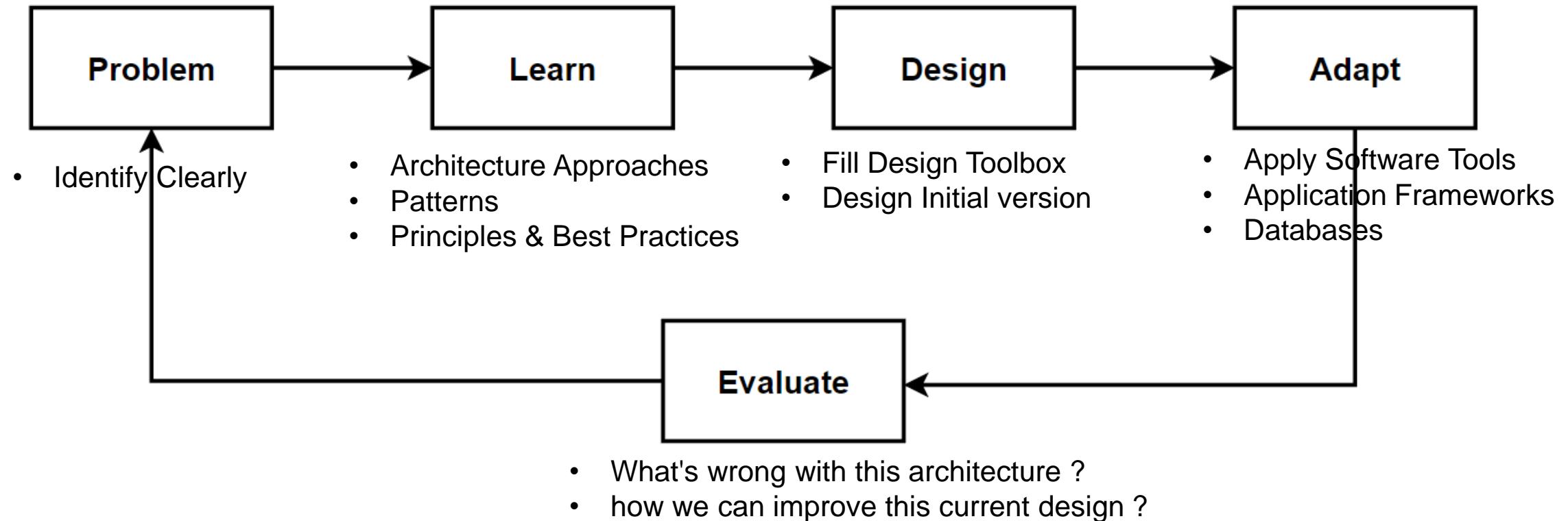
Log-based built-in CDC Database

- CockroachDB
- Azure CosmosDB
- Amazon DynamoDB Streams

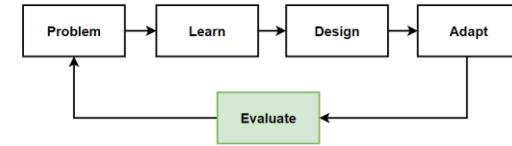
CDC – Open Source Tool

- Debezium

Way of Learning – The Course Flow



Evaluate: Microservice Architecture with SAGA, Transactional Outbox and CDC Pattern

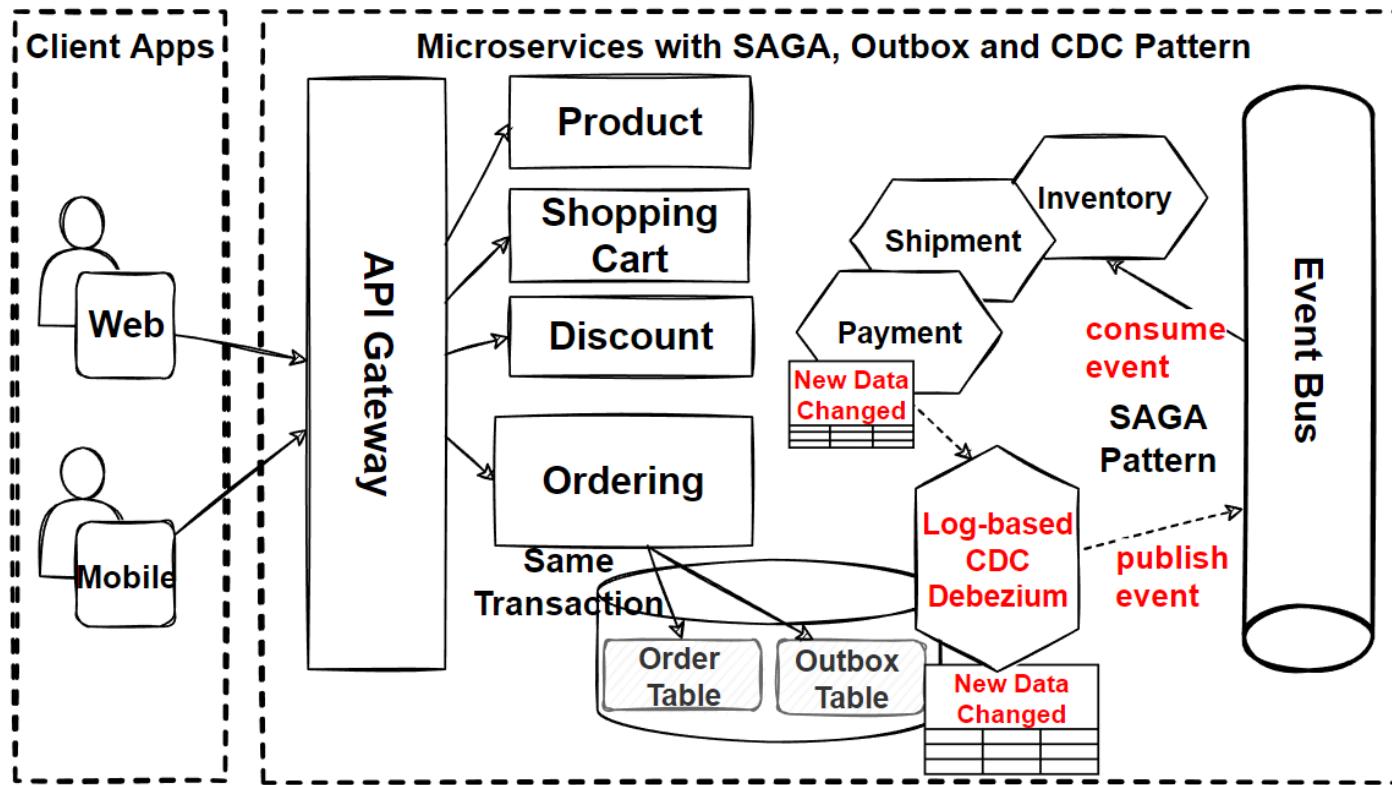


Benefits

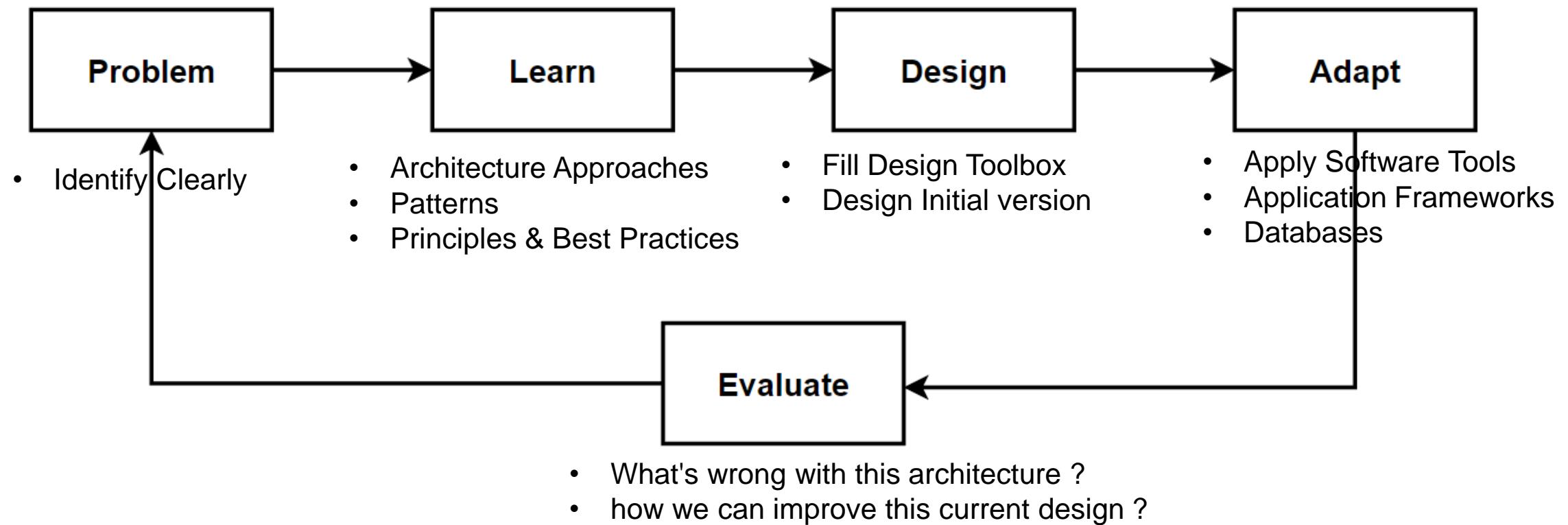
- Maintain the integrity of your database transactions
- Real-time synchronization
- Extracting and propagating changes to be kept up-to-date with the data
- Decouple data and propagate events
- Easier to scale and maintain a microservices

Drawbacks

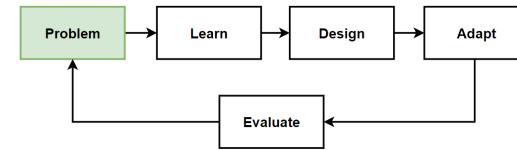
- Increased Complexity, Outbox Pattern and CDC makes your system more complex design.
- Add latency to the event publishing process
- Performance burden on the database
- Difficult to set up and maintain



Way of Learning – The Course Flow



Problem: Handle Millions of Events Across Microservices

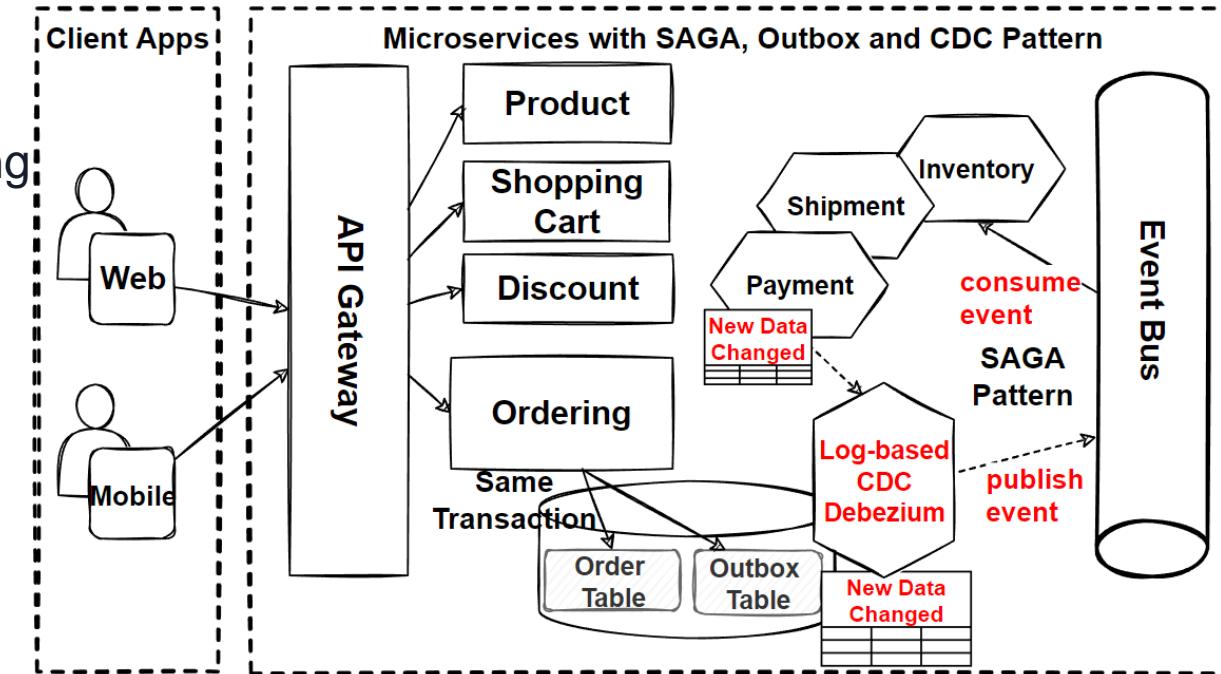


Considerations

- What if we have thousands of microservices that need to communicate with millions of events ?
- If multiple subsystems must process the same events
- Required Real-time processing with minimum latency.
- Required complex event processing, like pattern matching
- Required process high volume and high velocity of data, i.e. IoT apps.

Problems

- Decoupled communications for thousands of microservices
- Real-time processing
- Handle High volume events



Solutions

- Event-driven architecture for microservices

Event-Driven Microservices Architecture

Asynchronous communication, Decoupled communication

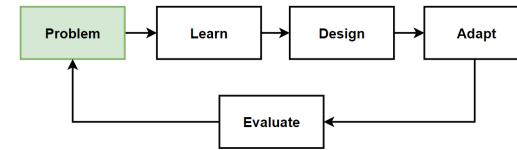
Event Hubs

Stream-Processing

Real-time processing

High volume events

Problem: Handle Millions of Events Across Microservices

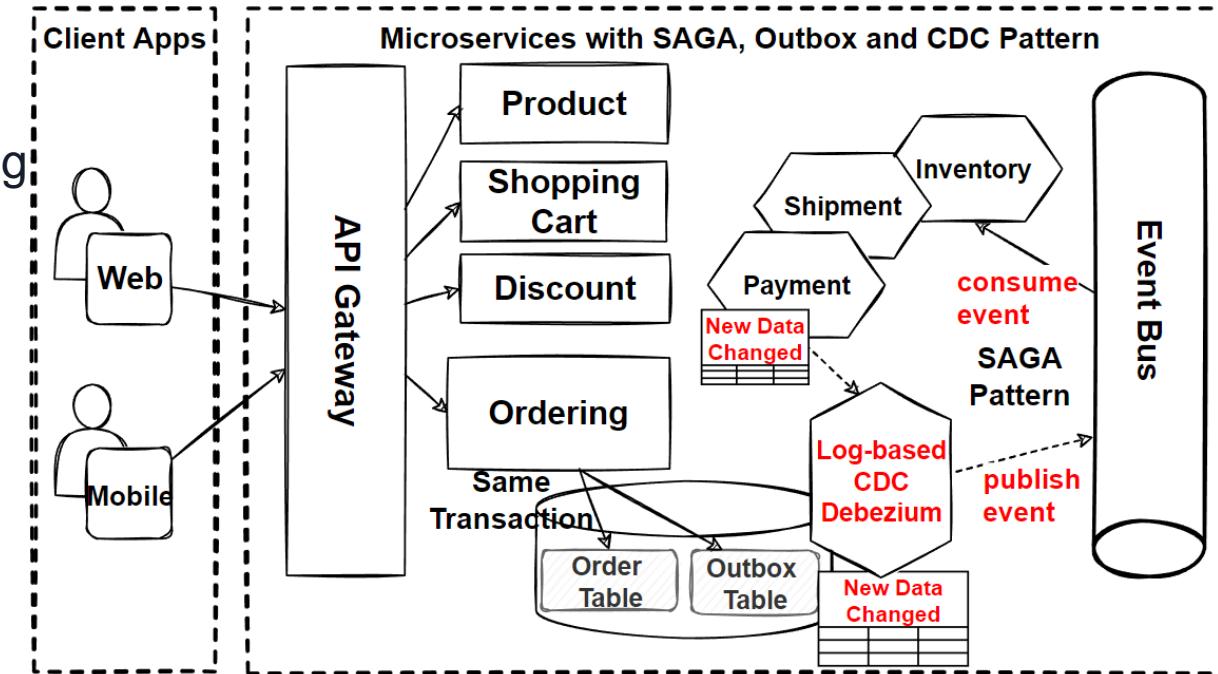


Considerations

- What if we have thousands of microservices that need to communicate with millions of events ?
- If multiple subsystems must process the same events
- Required Real-time processing with minimum latency.
- Required complex event processing, like pattern matching
- Required process high volume and high velocity of data, i.e. IoT apps.

Problems

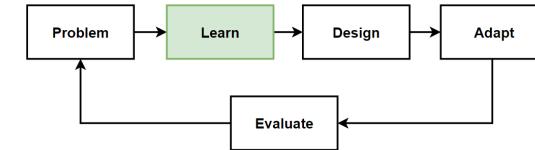
- Decoupled communications for thousands of microservices
- Real-time processing
- Handle High volume events



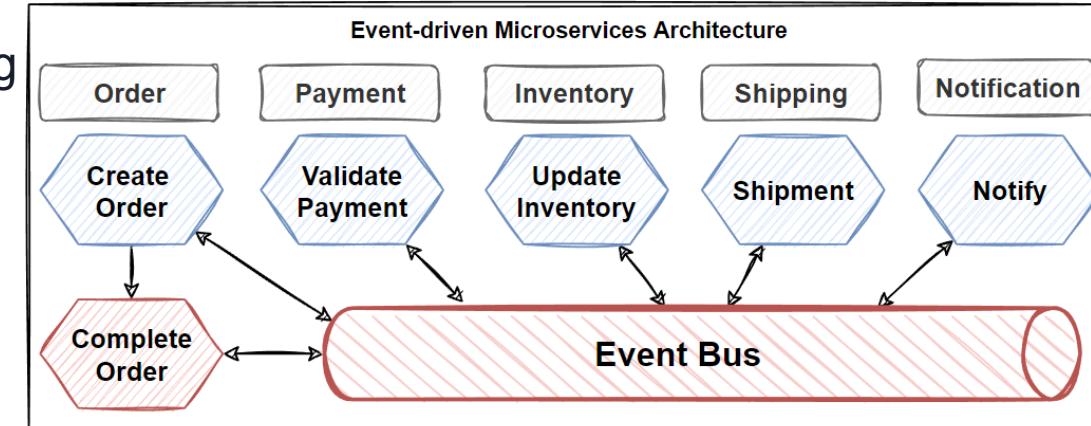
Solutions

- Event-driven architecture for microservices

Introduction - Event-driven Architecture

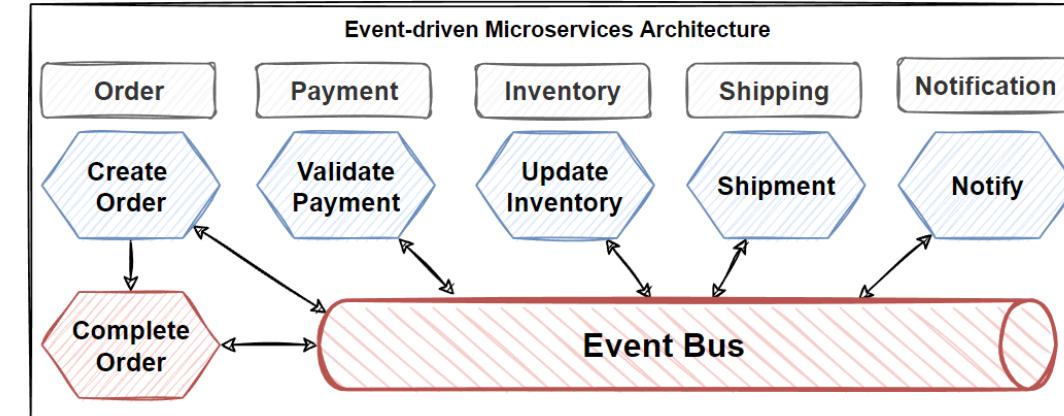
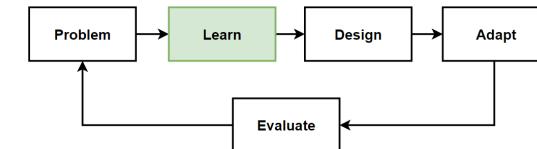


- **Microservices** architectures are designed to be **highly modular and flexible**, that can be **scaled and managed separately** and use of **APIs** to communicate between services.
- **Event-driven architecture**, microservices can communicate by **publishing and subscribing to events**, than directly calling each other's APIs.
- **Event-driven microservice** architecture is means **communicating with microservices via event messages** and we can do **asynchronous behavior and loosely coupled**.
- Instead of **sending request** when data needed, services **consume them via events**.
- **Huge Innovations** on the Event-Driven Microservices Architectures;
- **Real-time messaging platforms, stream-processing, event hubs, real-time processing, batch processing, data intelligence**.



Event-Driven Microservices Architecture

- **Event-driven microservices architecture**, services communicate with each other by **publishing** and **subscribing** to **events**.
- When a service needs to communicate with another service, it **publishes an event** to a message queue or event bus. Other services can then **subscribe** to that **event** and take appropriate action when the **event is received**.
- **Asynchronous communication**
Allows services to communicate asynchronously. Service can publish an event and continue processing without waiting for a response from the other service.
- **Decoupled communication**
Decouples the publisher and subscriber, allows to evolve independently without affecting each other.



Event-Driven Microservices Architecture - 2

- **Real-time processing**

Support real-time processing, as events are published and consumed as soon as they occur. Need to react to events in real-time, such as in systems that use CDC to track changes to a database.

- **High volume events**

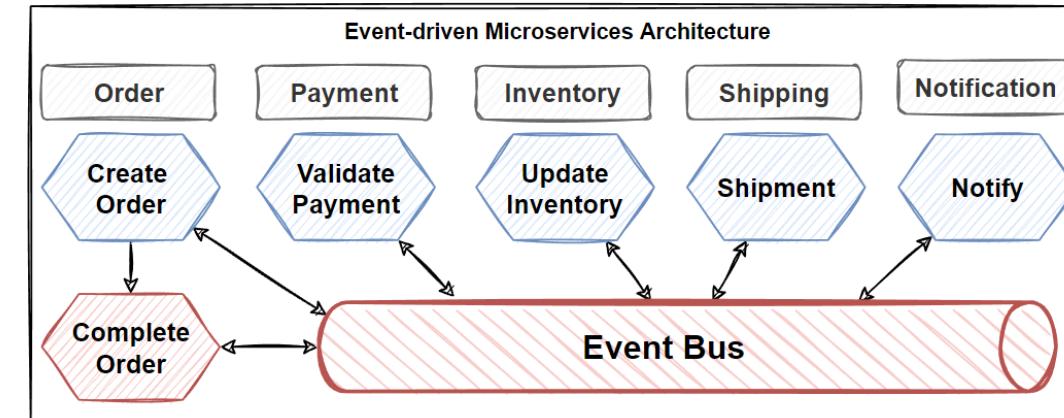
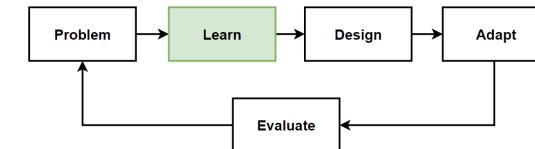
Well-suited to handling high volume events, as they can scale horizontally by adding more event consumers as needed. Can be scaled independently to handle increased load.

- **Responsible business capability**

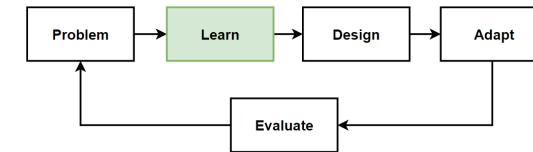
Each service is responsible for a specific function or business capability.

- Services communicate with each other by publishing and subscribing to events, that make it easier to build and maintain complex systems.

- Allows to work on different parts of the system in parallel without having to worry about the impact on other components.



Real-time Processing and High Volume Events in Event-Driven Microservices Architecture



- **Real-time processing**

Real-time processing is achieved by using a message queue or event bus to publish and consume events as they occur.

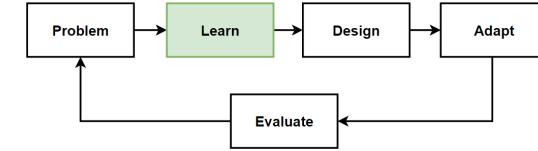
- When an event is generated, it is published to the message queue or event bus and made available to any interested subscribers.
- Allows you to react to events in real-time, as they are published and consumed as soon as they occur.
- When need to perform real-time analytics or trigger actions based on changes to the data.

- **High Volume Events**

Using a message queue or event bus that can handle high volumes of events and distribute them to multiple consumers.

- When you have a system that generates a large number of events, use a event bus to distribute those events to multiple consumers, that can process events in parallel.
- Allows you to scale up the number of event consumers as needed to handle increased load.
- Event-driven microservices architectures to process events in real-time and scale to handle high volumes of events.

Event Hubs and Event Streaming in Event-Driven Microservices Architecture



- **Event Hubs**

Act as a central hub for data ingestion and distribution, allowing microservices to publish and subscribe to data streams.

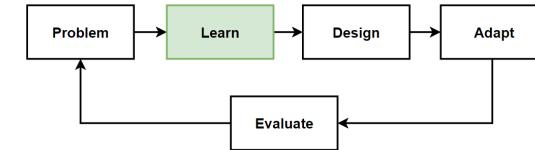
- Each microservice publish its data to an Event Hub. Other microservices can then subscribe to the Event Hub and consume the data as needed.
- Communicate with each other in real-time, that needs to be highly reliable and scalable.

- **Event Streaming**

Allows you to capture and process a stream of events in real-time. Publish and consume events as they occur.

- Allowing to build real-time data pipelines that can process and analyze data as it is being generated.
- One common use case for Event Hubs and event streaming in microservices architectures is real-time analytics.

Use Cases of Event Hubs and Event Streaming



- **Use Case – Customer Purchase on E-Commerce**

E-commerce website that generates a large number of events as customers browse and purchase products.

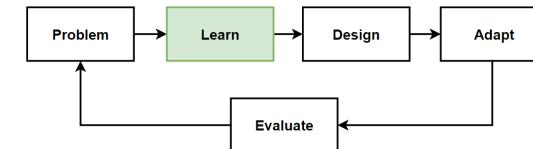
- Using event streaming, build a real-time analytics pipeline that processes these events as they occur and generates insights in real-time.
- This could include tracking customer behavior, identifying trends and patterns, and triggering actions based on changes to the data.

- **Use Case – Data Synchronization**

Different services may need to access the same data, and not be able to access each other's databases directly.

- Using event streaming, we can build a data synchronization pipeline that captures changes made to the data and streams them to other services in real-time. Ensures to access the most up-to-date data.
- Event Hubs and event streaming is a powerful tool for building real-time data pipelines in microservices architectures that provides flexible platform for data ingestion and distribution.

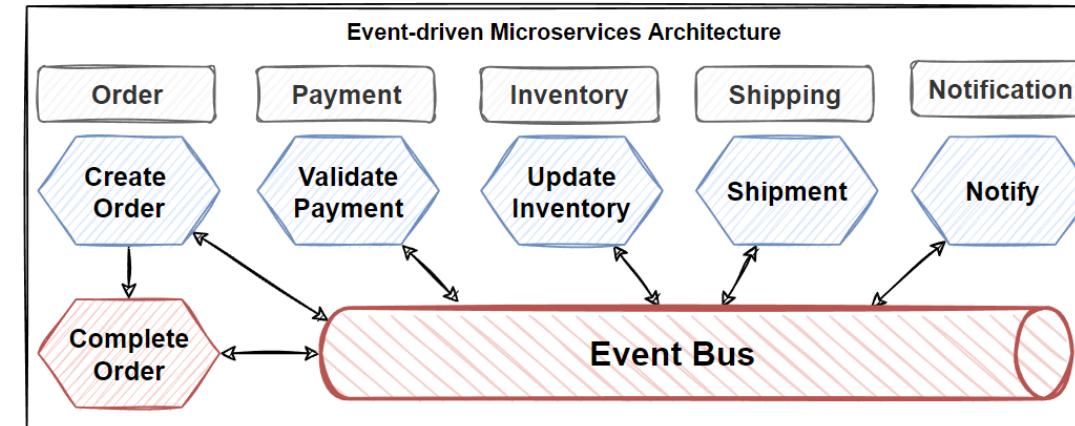
Real-world Examples of Event-Driven Microservices



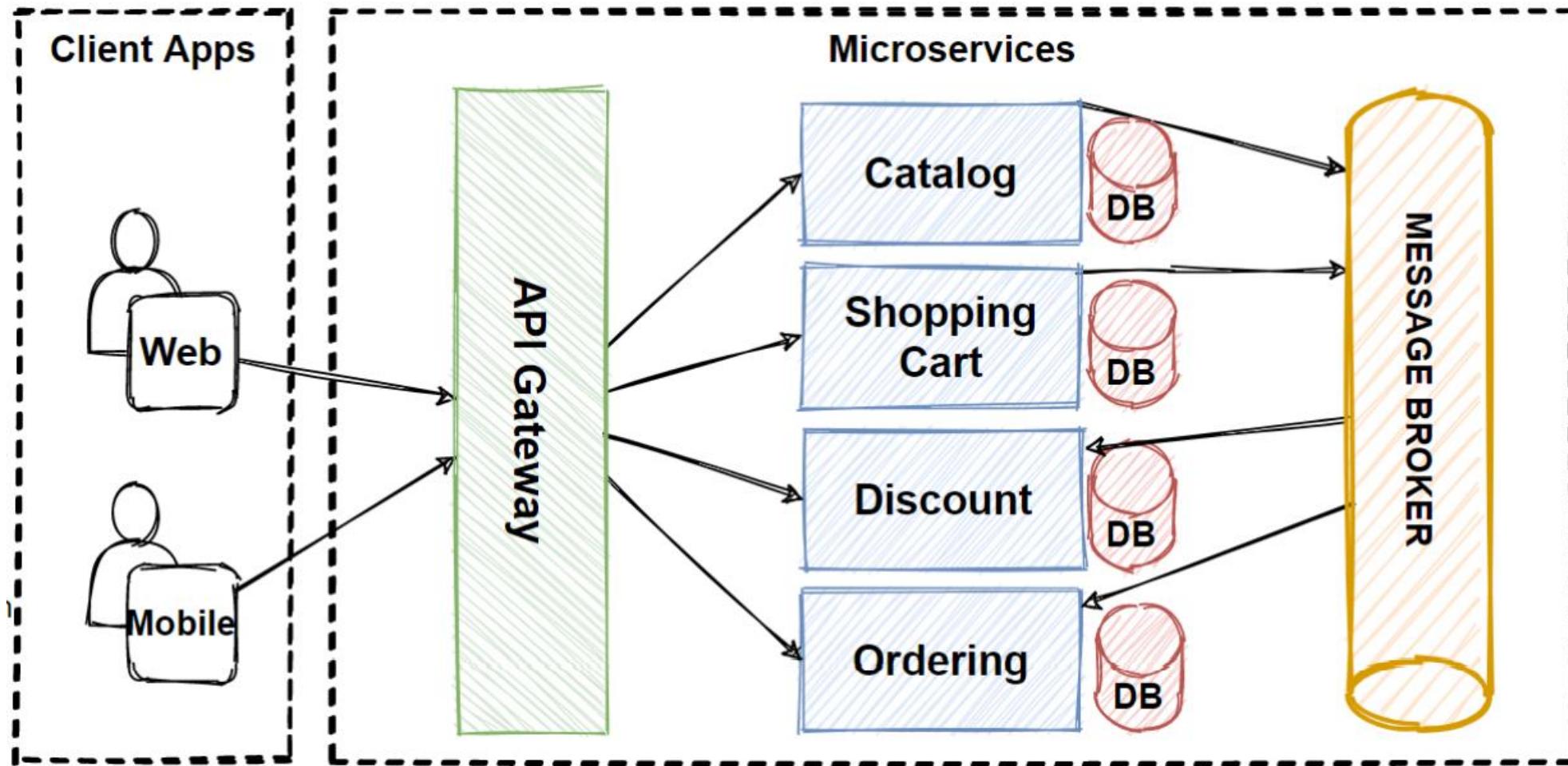
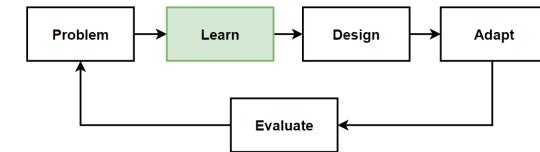
- **E-commerce application** we have a few microservices like: **customer, order, payment and products**.
- **Customer create orders** with some products and, if the payment is successful, the products should be delivered to the customer.

Events

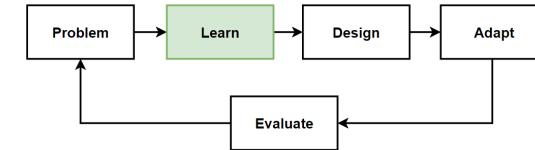
- a customer creates an order
- the customer receives a payment request
- if the payment is successful the stock is updated and the order is delivered
- if the payment is not successful, rollback the order and set order status is not completed.
- This is more **humanly readable** and, if a new business requirement appears, it is **easier to change** the flow.
- **Microservices will only care about the events**, not about the other microservices, **process only events** and **publish new event** to trigger other services.



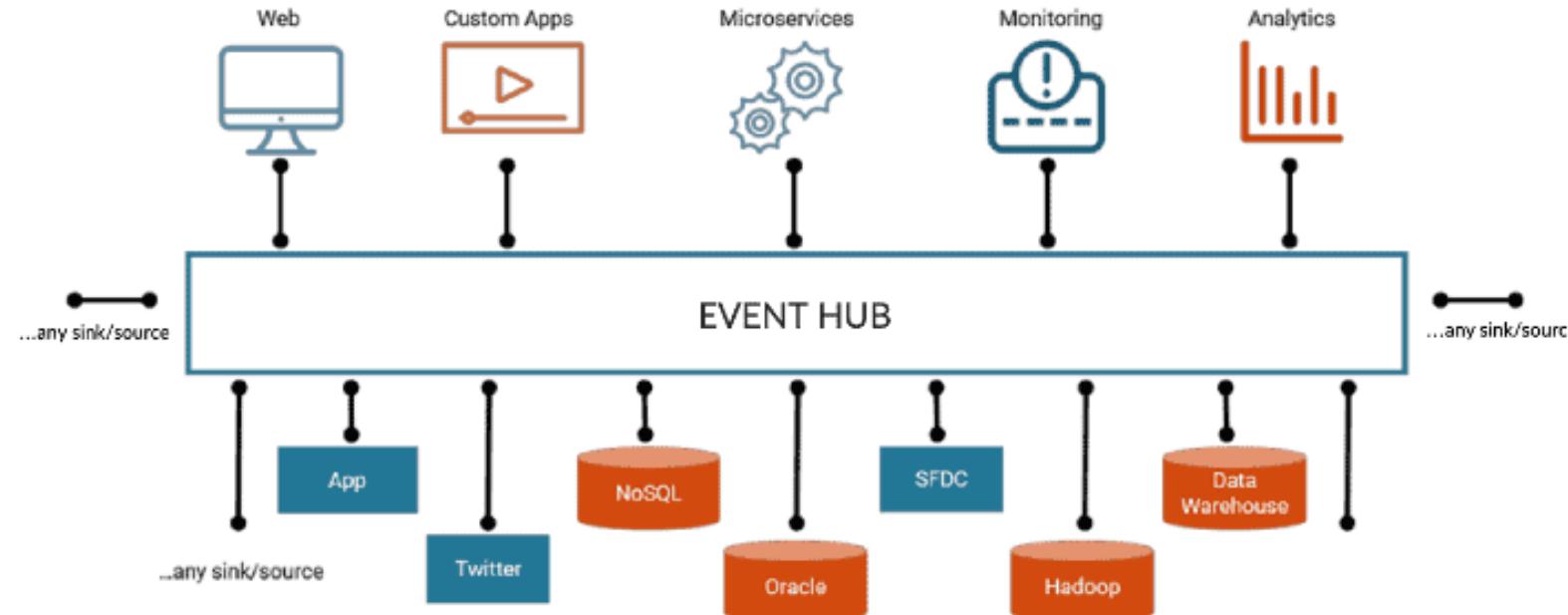
Traditional Event-Driven Microservices Architecture



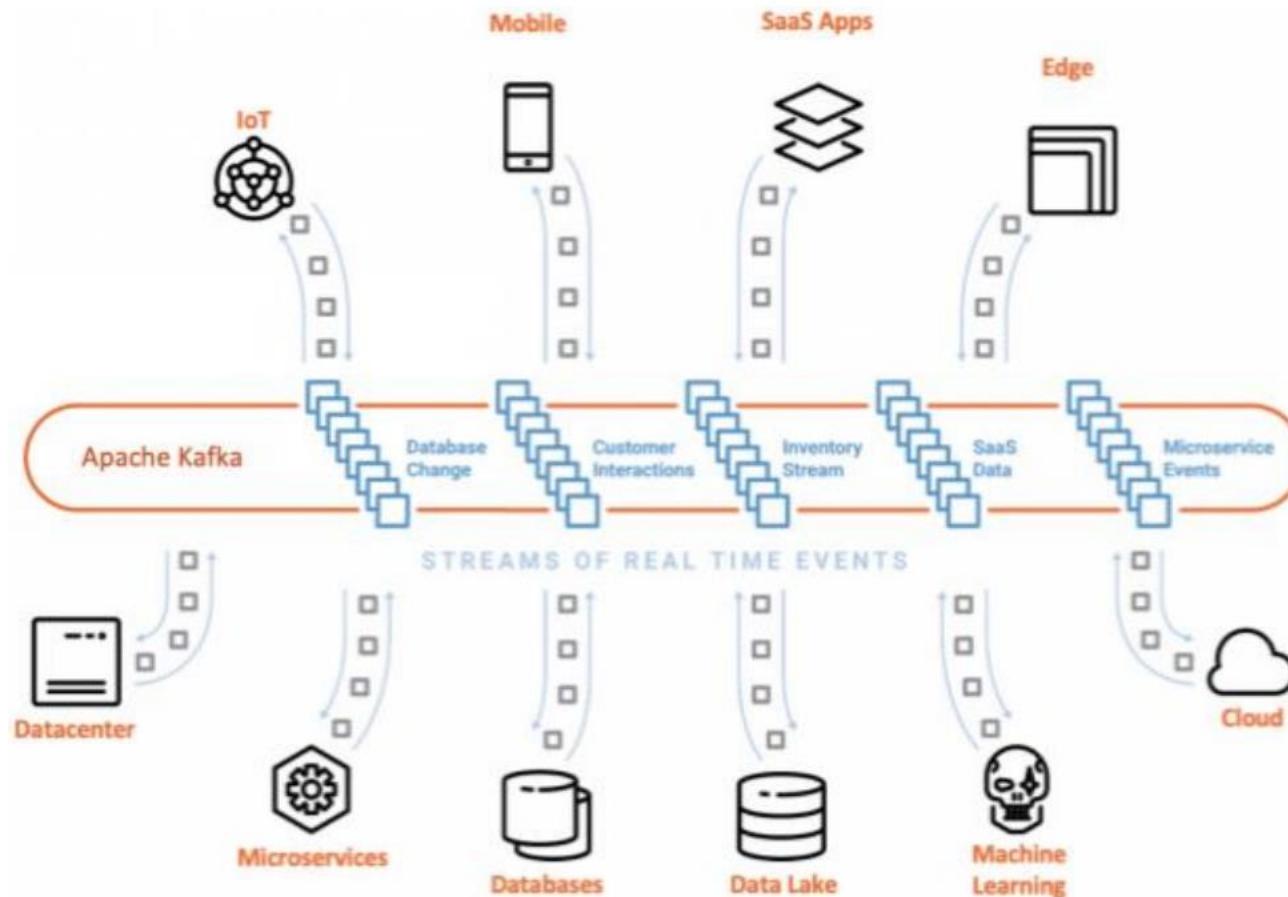
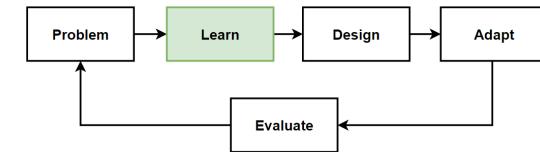
Evolved Event-Driven Microservices Architecture



- Huge Innovations on the Event-Driven Microservices Architectures.
- Real-time messaging platforms, stream-processing, event hubs, real-time processing, batch processing, data intelligence.
- Event-Hubs is huge event store database that can make real-time processing.



Kafka Event-Driven Microservices Architecture



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



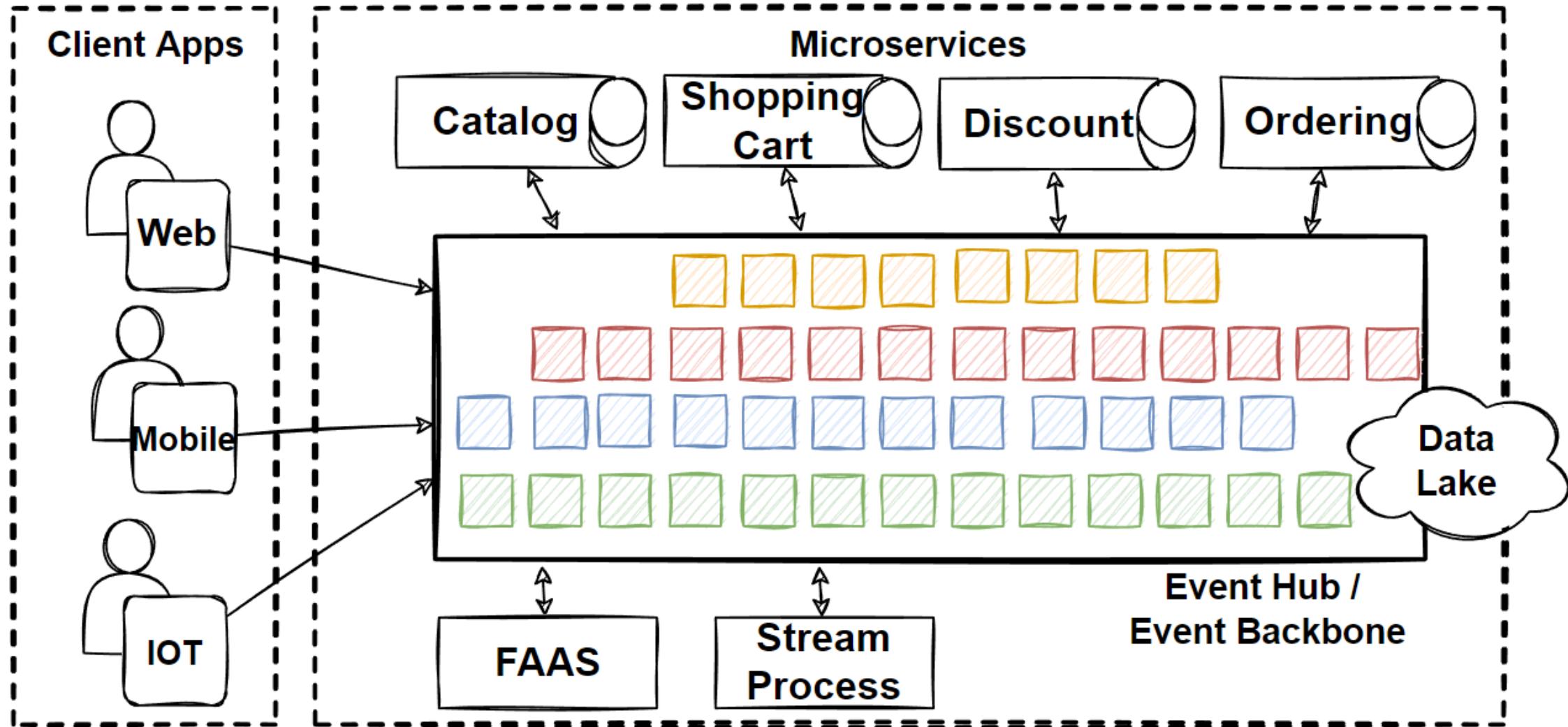
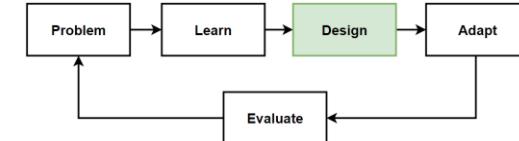
Before Design – What we have in our design toolbox ? - Old

Architectures Patterns&Principles	Microservices Data Choosing Database	Microservices Distributed Transactions	FR
<ul style="list-style-type: none">• Microservices Architecture• The Database-per-Service Pattern• Polygot Persistence• Decompose services by scalability• The Scale Cube• Microservices Decomposition Pattern• Microservices Communications Patterns• Microservices Data Management Patterns• Microservices Distributed Transaction Pattern	<ul style="list-style-type: none">• The Shared Database Anti-pattern, Relational and NoSQL Databases• CAP Theorem–Consistency, Availability, Partition Tolerance• Data Partitioning: Horizontal, Vertical and Functional Data Partitioning• Database Sharding Pattern	<ul style="list-style-type: none">• SAGA Pattern• Choreography and Orchestration-based SAGA• Compensating Transaction Pattern• Dual-Write Problem• Transactional Outbox Pattern• CDC - Change Data Capture	<ul style="list-style-type: none">• List products• Filter products as per brand and categories• Put products into the shopping cart• Apply coupon for discounts• Checkout the shopping cart and create an order• List my old orders and order items history
	Microservices Data Commands&Queries <ul style="list-style-type: none">• Materialized View Pattern• CQRS Design Pattern• Event Sourcing Pattern• Eventual Consistency		Non-FR <ul style="list-style-type: none">• High Scalability• High Availability• Millions of Concurrent User• Independent

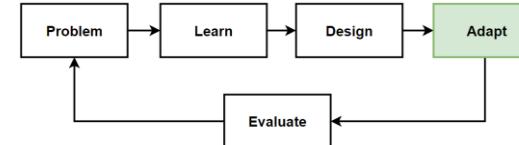
Before Design – What we have in our design toolbox ? - New

Architectures	Patterns&Principles	Microservices EDA	Non-FR	FR
• Microservices Architecture	• The Database-per-Service Pattern	• Asynchronous, Decoupled communication	• High Scalability	• List products
• Event-Driven Microservices Architecture	• Polygot Persistence • Decompose services by scalability • The Scale Cube • Microservices Decomposition Pattern • Microservices Communications Patterns • Microservices Data Management Patterns • Event-Driven Architecture	• Event Hubs • Stream-Processing • Real-time processing • High volume events	• High Availability • Millions of Concurrent User • Independent Deployable • Technology agnostic • Data isolation • Resilience and Fault isolation	• Filter products as per brand and categories • Put products into the shopping cart • Apply coupon for discounts • Checkout the shopping cart and create an order • List my old orders and order items history

Event-Driven Microservices Architecture



Adapt: Event-Driven Microservices Architecture



Frontend SPAs

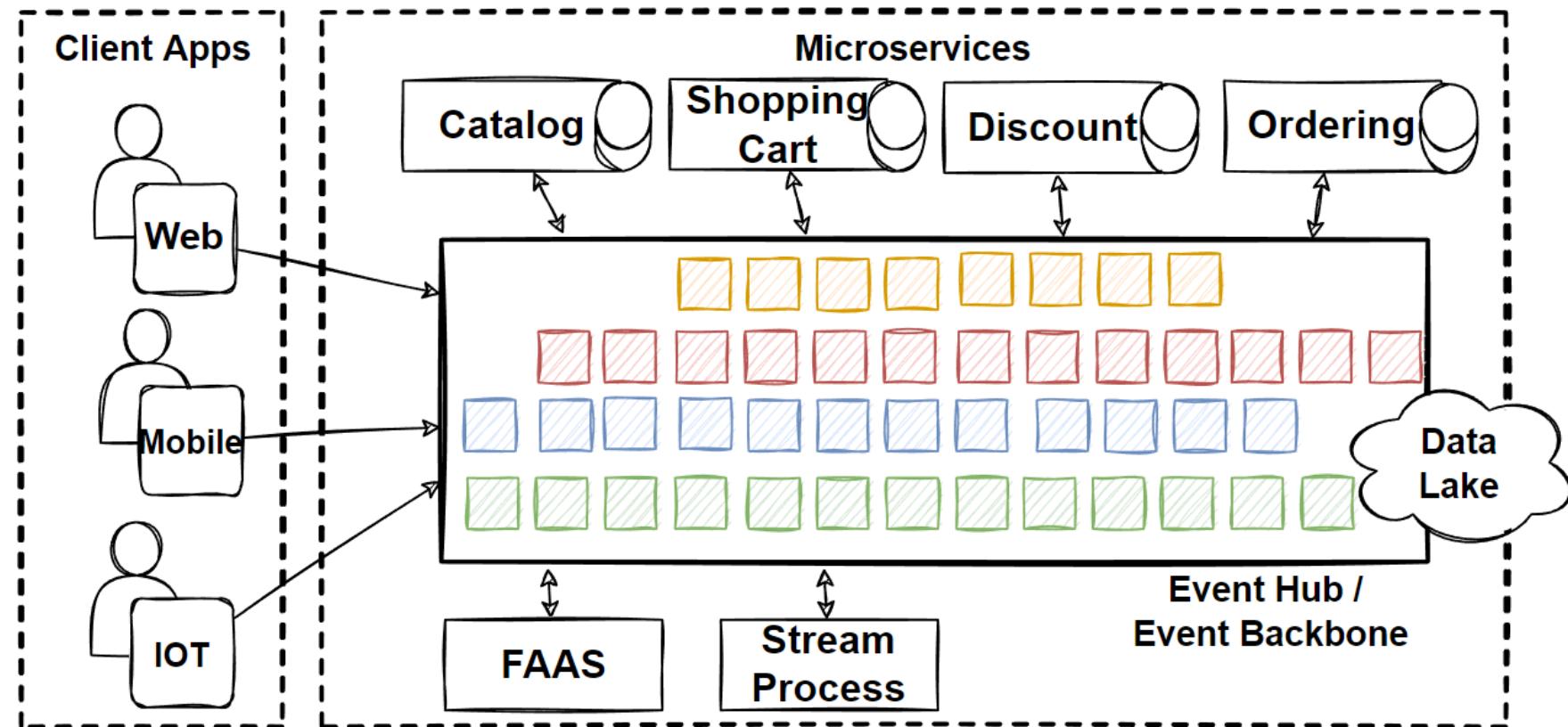
- Angular
- Vue
- React

API Gateways

- Kong Gateway
- Express Gateway
- Amazon AWS API Gateway

Backend Microservices

- Java – Spring Boot
- .Net – Asp.net
- JS – NodeJS



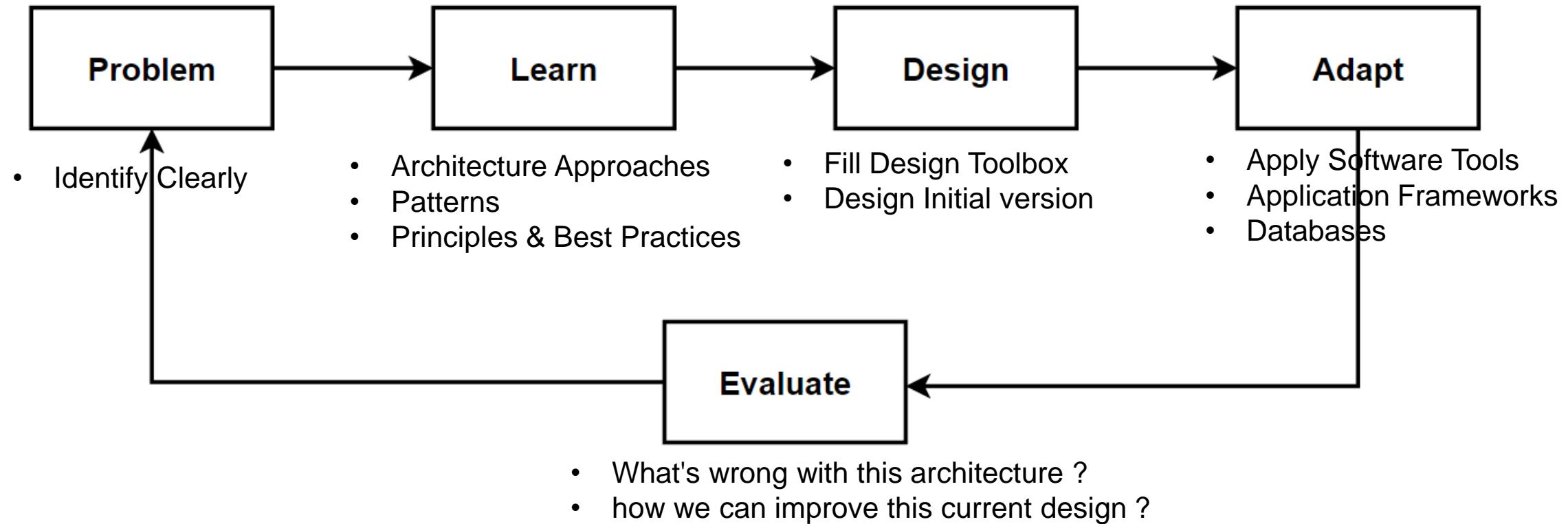
Event Hubs

- Apache Kafka
- Apache Spark

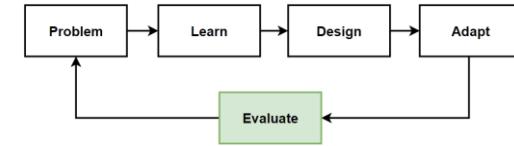
Cloud Event Hubs

- Azure Event Hubs
- AWS Kinesis
- Oracle Event Hubs

Way of Learning – The Course Flow



Evaluate: Event-Driven Microservices Architecture

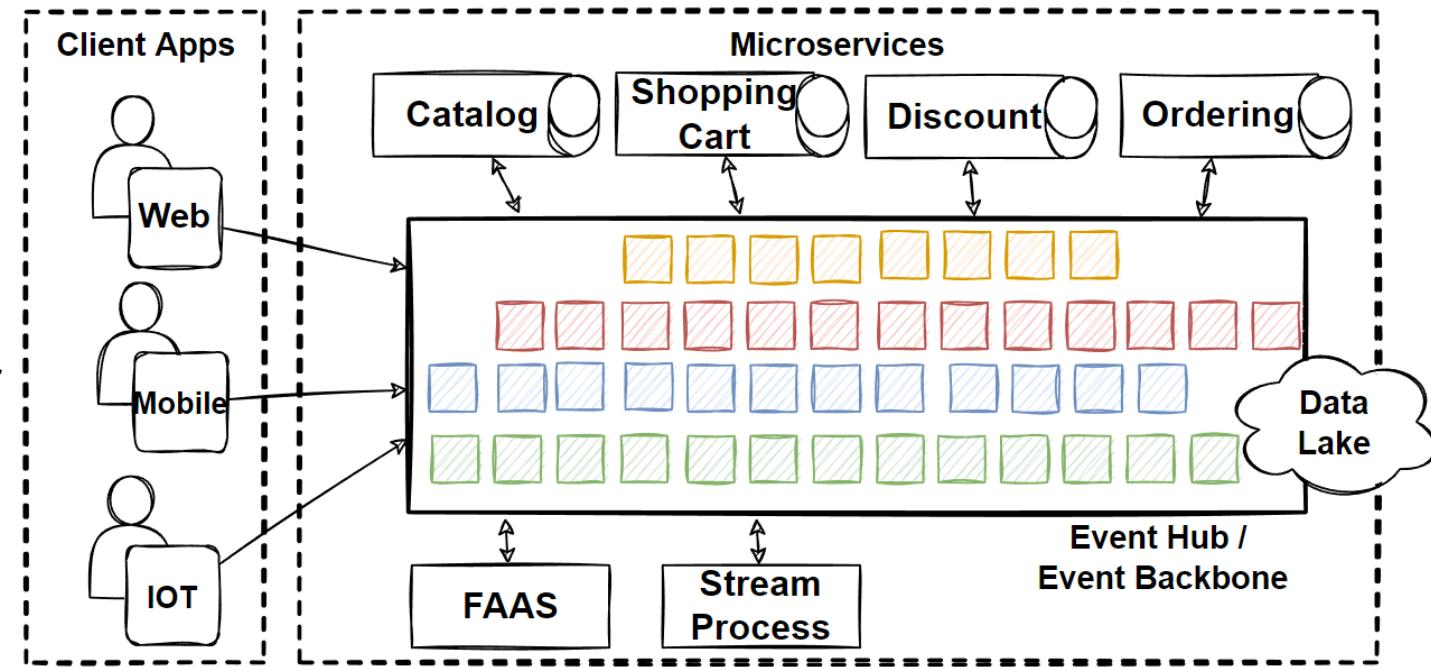


Benefits

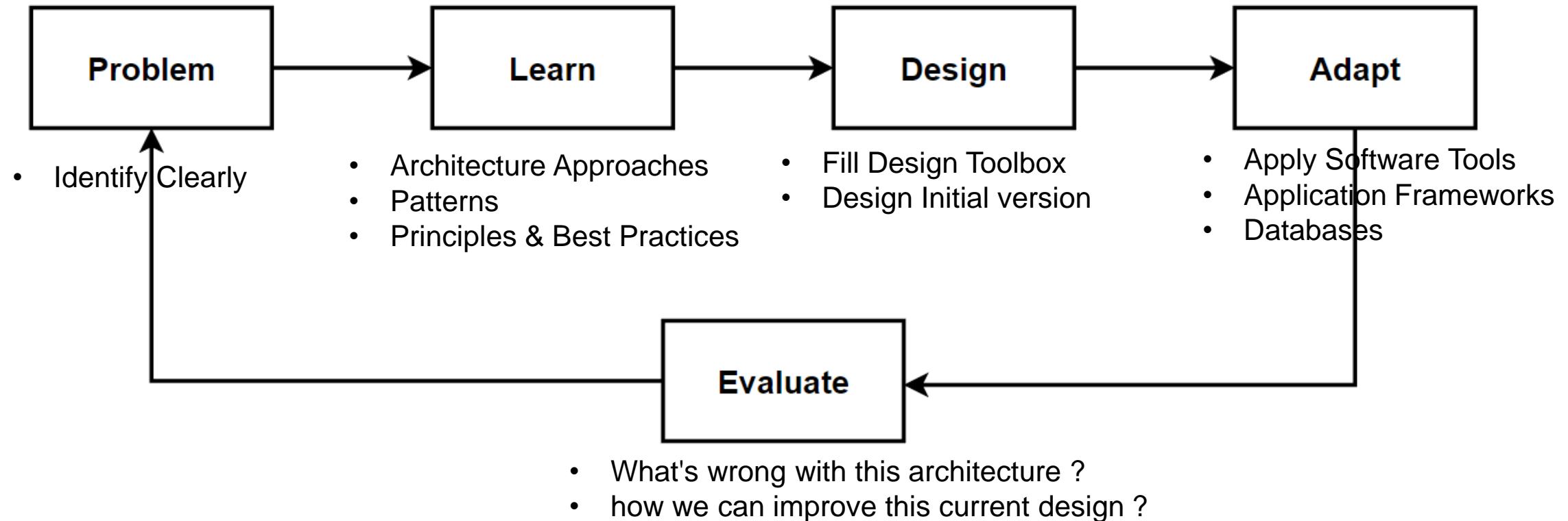
- Event Streaming
- Real-time Processing
- High Volume Events
- Decouple services, increased Scalability
- Resilience

Drawbacks

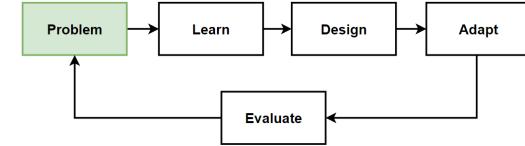
- Increased Complexity, event-driven makes your system more complex design.
- Hard to Debugging
- Add latency to the event publishing process
- Integration into Distributed Transactions
- Difficult to set up and maintain



Way of Learning – The Course Flow



Problem: Database operations are expensive, low performance

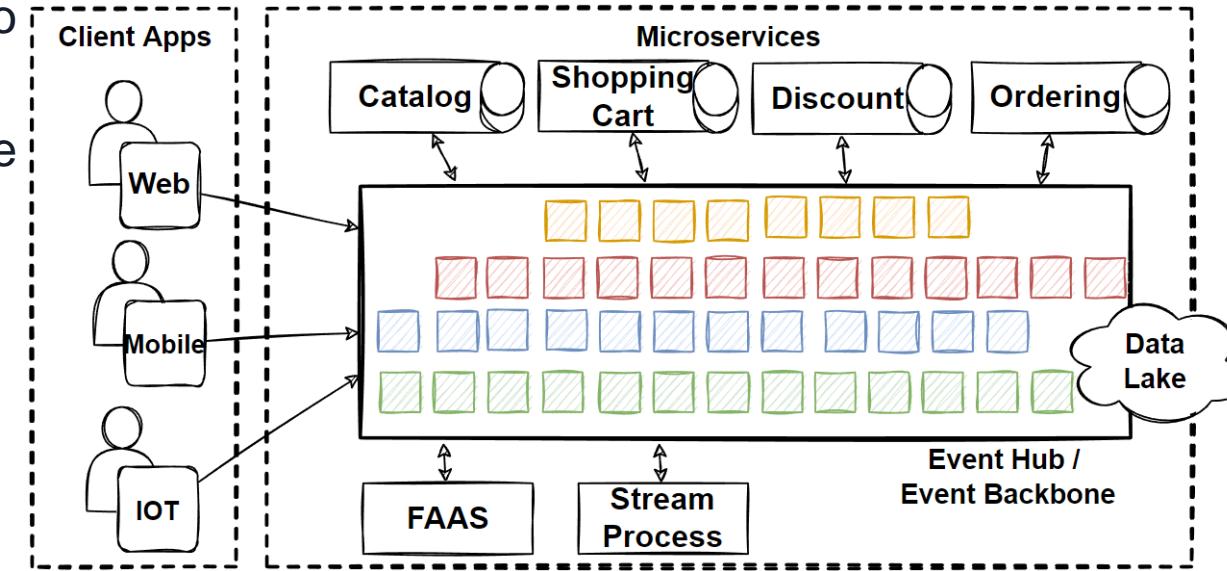


Considerations

- Event-driven architecture comes with latency when publishing and subscribing events from the Event Hub.
- Sync REST APIs communication make expensive calls to a database that reduce performance.
- How can we make more faster that increase performance of communications in Microservices Architecture ?

Problems

- Slowness and Low Performance Communication
- Latency when publishing and subscribing events
- Rest APIs make Database calls that are expensive, low performance



Solutions

- Distributed cache
- Storing frequently accessed data in a distributed cache

Microservices Distributed Caching

Microservices Distributed Caching Patterns and Practices

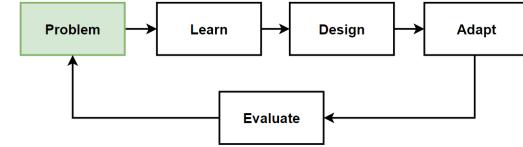
Cache-Aside Pattern

Caching strategies

Cache invalidation

Cache consistency

Problem: Database operations are expensive, low performance

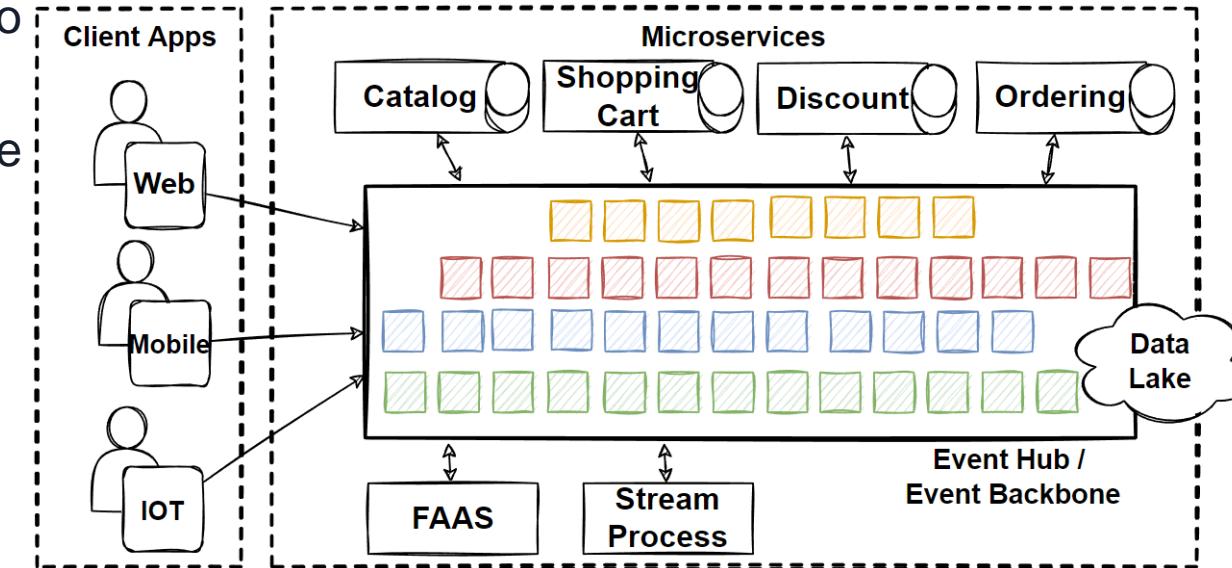


Considerations

- Event-driven architecture comes with latency when publishing and subscribing events from the Event Hub.
- Sync REST APIs communication make expensive calls to a database that reduce performance.
- How can we make more faster that increase performance of communications in Microservices Architecture ?

Problems

- Slowness and Low Performance Communication
- Latency when publishing and subscribing events
- Rest APIs make Database calls that are expensive, low performance

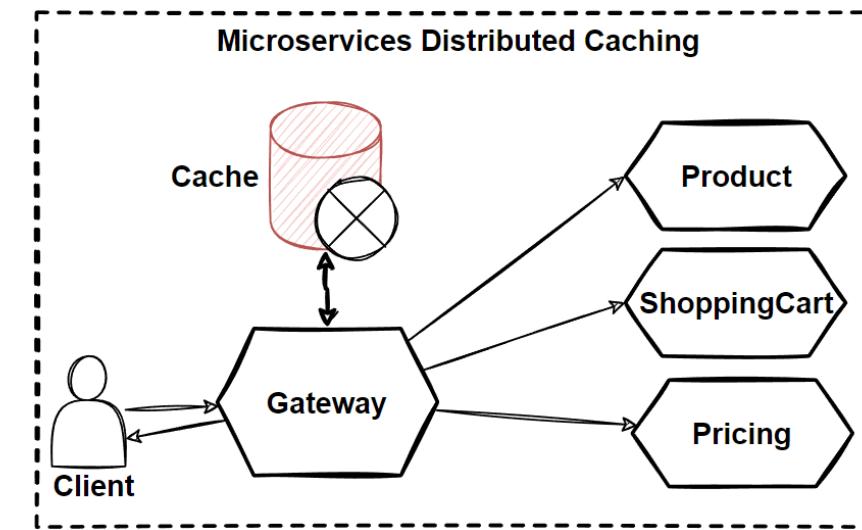
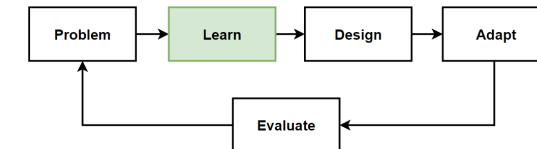


Solutions

- Distributed cache
- Storing frequently accessed data in a distributed cache

What is Caching ?

- **Caching for improving the performance of a system by storing frequently accessed data in a cache that can be quickly accessed from memory.**
- **Caching is to reduce the number of expensive operations, such as database queries or network requests.**
- **Caching can increase performance, scalability, and availability for microservices with reducing latency with cache and makes application faster.**
- When the **number of requests are increased**, caching provide to **handle requests with high availability**.
- If application **request** mostly comes for **reading data** that is not **changes so frequently**, then **Caching** will be so **efficient**.
- I.e. **reading product catalog** from e-commerce application. Caching also provide to **avoid re-calculation processes**.
- By **storing frequently accessed data** in a cache, system can **avoid the overhead of repeatedly expensive operations**.



Types of Caching

- **In-memory cache**

Stores data in the main memory of a computer. In-memory caches are typically the fastest type of cache, but the data is lost when the cache is restarted or the machine is shut down.

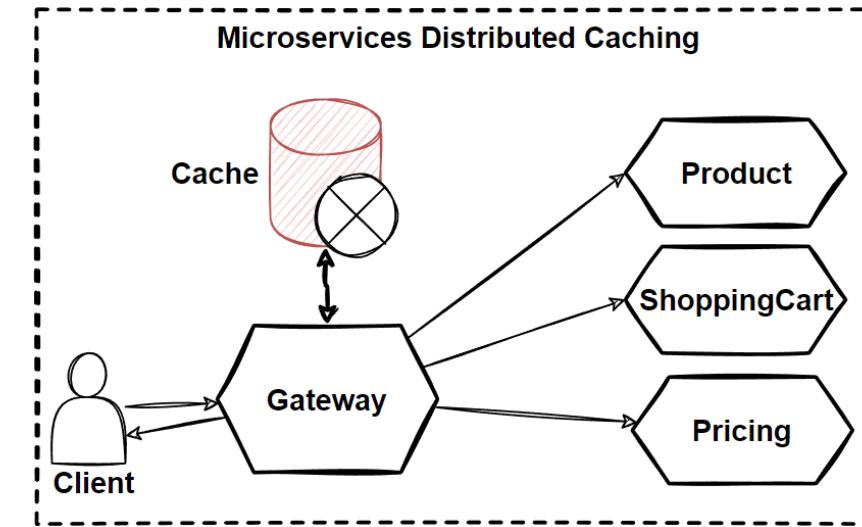
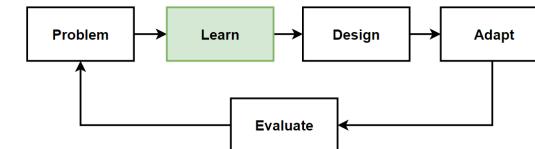
- **Disk cache**

Stores data on a hard drive or solid-state drive. Disk caches are slower than in-memory caches, but they can persist data.

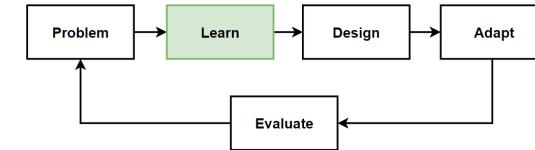
- **Distributed cache**

Cache is distributed across multiple machines and is typically used in distributed systems, such as microservices architectures.

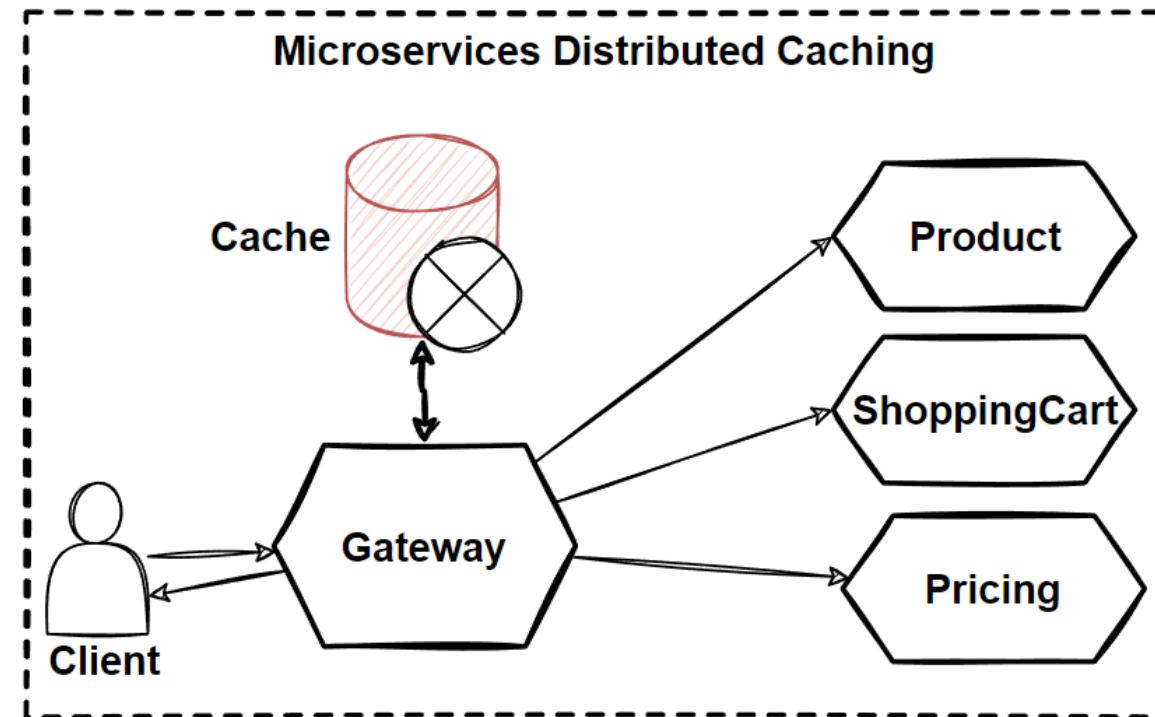
- Distributed caches can improve the performance and scalability of a system by allowing data to be stored and accessed from multiple locations.



Distributed Caching in Microservices

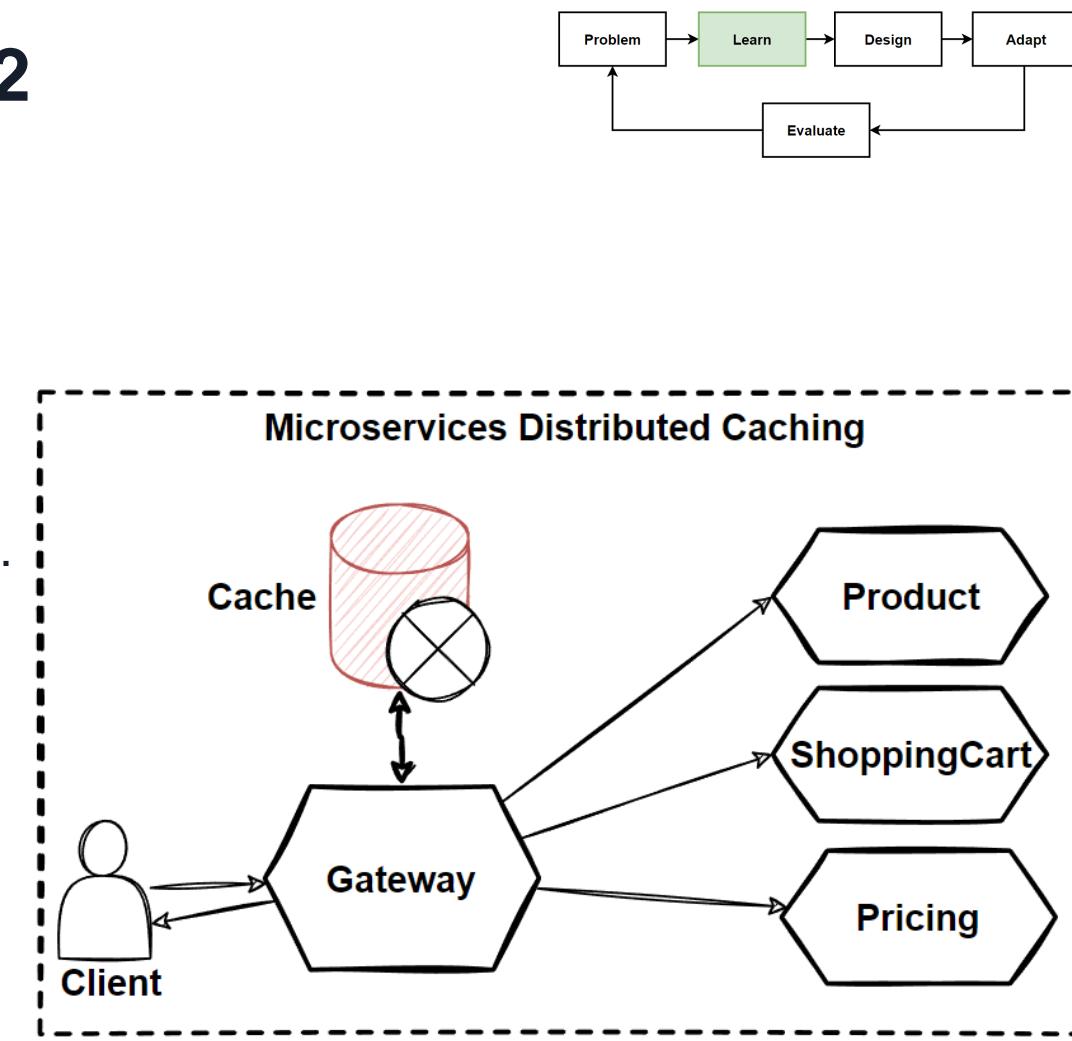


- **Distributed caching** is improving the performance by **storing frequently accessed data** in a **cache** that can be **quickly accessed** from **multiple locations**.
- **Microservices architectures** are typically implement a **distributed caching** architecture:
 - Improve the performance of individual services by storing frequently accessed data locally.
 - Reducing the need to make expensive calls to a database or other external system.
- **How can we increase the speed of the microservices ?** With using **Distributed Cache**.



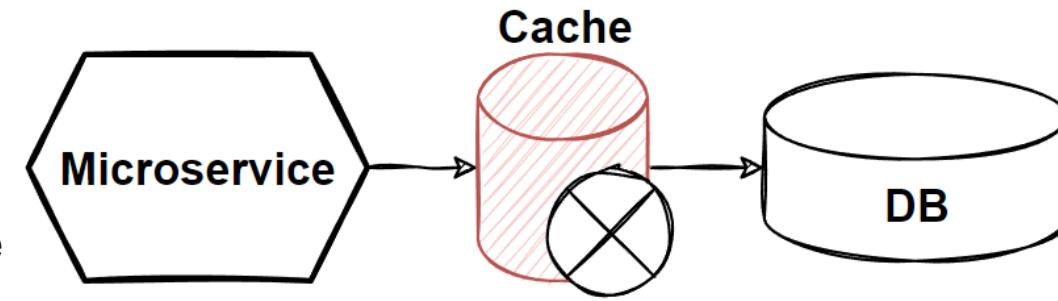
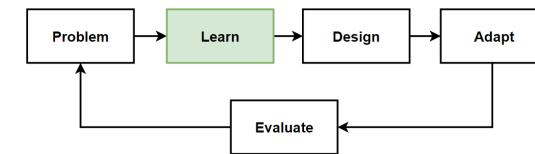
Distributed Caching in Microservices - 2

- Microservices are **responsible for a specific function** and **communicates** with other services through well-defined interfaces, **typically using APIs**.
- By **storing frequently accessed data locally in a cache**, microservices can **avoid the overhead of making repeated calls** to an external system, resulting in **faster response times**.
- Benefits to using distributed caching in a microservices:
- **Improved performance**
Services can avoid the overhead of making repeated calls to a database or other external system.
- **Resilience**
Allowing services to continue functioning even if an external system becomes unavailable.
- **Scalability**
Allowing services to handle increased traffic without the need to scale up the external system.



Cache Hit and Cache Miss

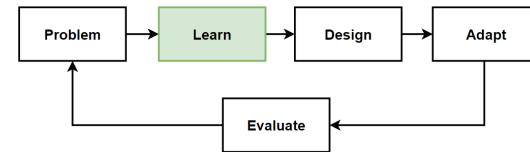
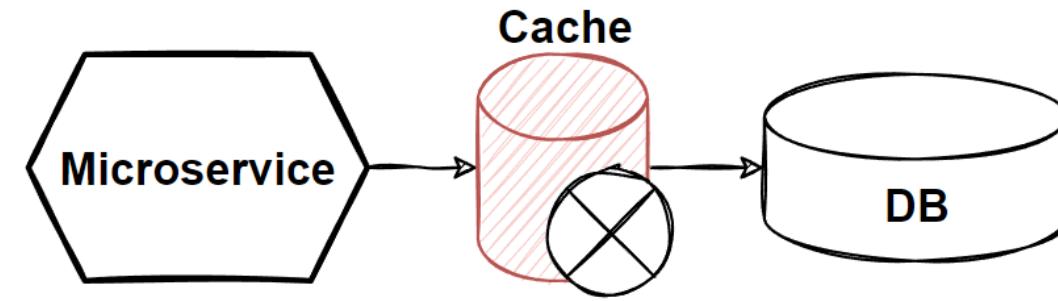
- **Cache hit** occurs when the **requested data** can be **found** in the **cache**.
- **Cache miss** occurs when the **requested data is not** in the **cache** and **must be retrieved** from a slower storage db.
- **Cache hits** are **desirable** because they can improve the performance of a system by **reducing the number of requests**.
- **Cache misses** can have a **negative impact** on performance, because they **require additional time** and resources to retrieve the requested data.
- **The cache hit rate** is a measure of **how often a cache** is able to **fulfill requests** from its own store.
- **High cache hit rate** indicates that the cache is **effective** at storing frequently accessed data.
- **Low cache hit rate** may indicate that the **cache is not large enough**.



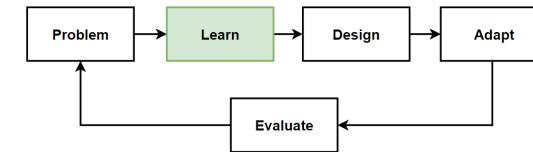
Caching Strategies in Distributed Caching

- There are **several caching strategies** that can be used in distributed microservices:
 - Cache Aside
 - Read-Through
 - Write-Through
 - Write-Back, Write-Behind
- **Cache Aside Strategy**

Client checking the cache for data before making a request to the backend service. When microservices needs to read data from the database, it checks the cache first to determine whether the data is available.
- If the **data is available (a cache hit)**, the cached data is returned. If the **data isn't available (a cache miss)**, the database is queried for the data.
- The client will **retrieve the data from the backend service** and store it in the cache for future requests.
- Data is **lazy loaded into cache** by client application.



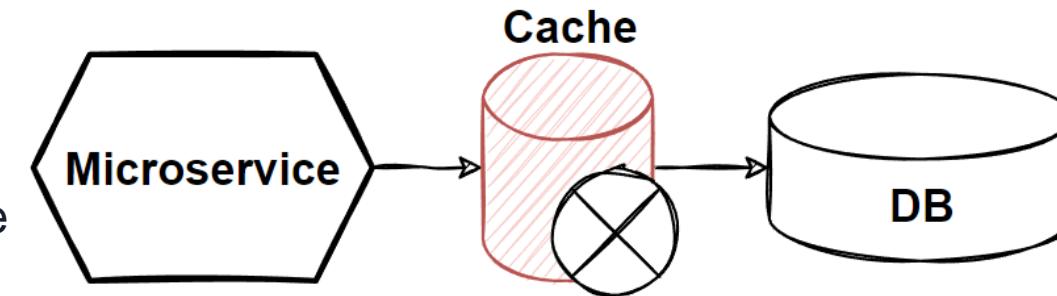
Caching Strategies in Distributed Caching



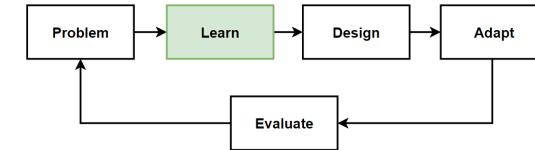
- **Read-Through Strategy**

When there is a cache miss, it loads missing data from the database, populates the cache and returns it to the application.

- When a client **requests data that is not found** in the cache, the cache will **automatically retrieve** the data from the underlying database and **store** it in the **cache** for future requests.
- **Cache-aside strategy**, when a client requests data that is **not found in the cache**, the client is responsible for retrieving the data from the database.
- **Read-through cache strategy**, when a client requests data that is **not found in the cache**, the **cache will automatically retrieve** the data from the database.
- **Cache always stays consistent** with the database.



Caching Strategies in Distributed Caching



- **Write-Through Strategy**

Update the cache whenever data is written to the backend service. Cache always has the most up-to-date data, but it can also result in a higher number of write operations.

- **Instead of lazy-loading** the data in the cache after a cache miss, the cache is proactively updated immediately following the primary database update.

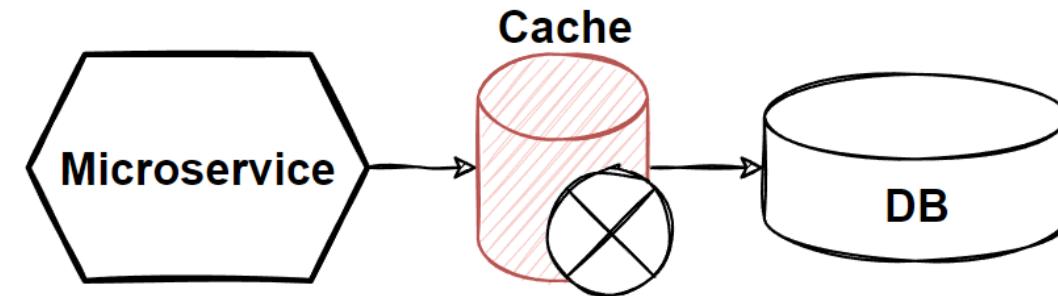
- Data is first written to the cache and then to the database.

- **Write-Back or Write-Behind Strategy**

Delays updating the cache until a later time. This reduces the number of write operations, but the cache may not have the most up-to-date data.

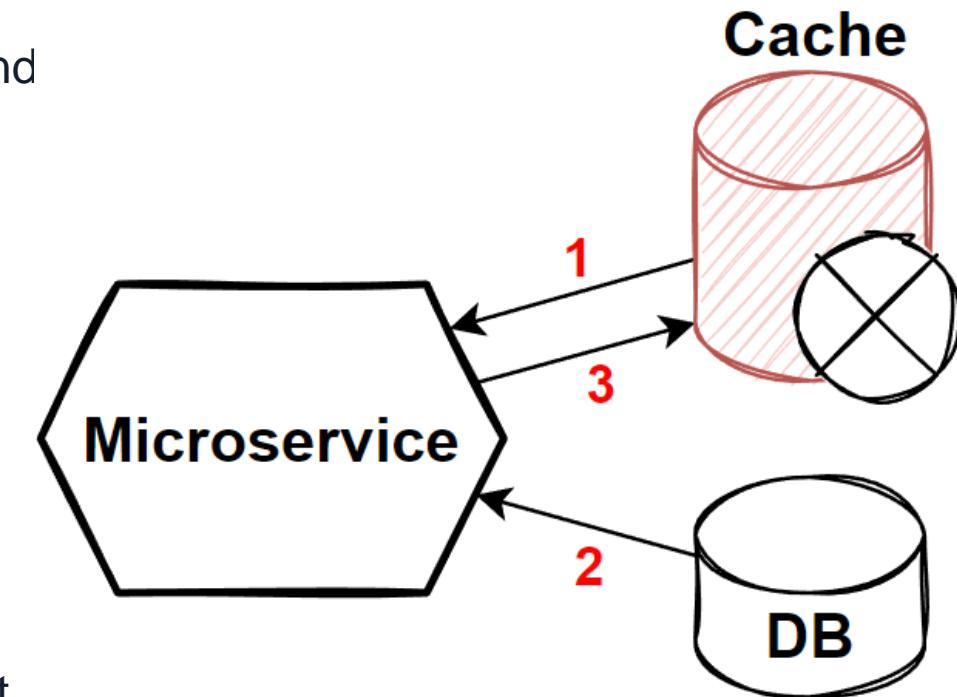
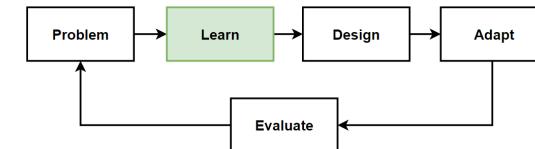
- **In Write-Through**, the data written to the cache is synchronously updated in the main database.

- **In Write-Back or Write-Behind**, the data written to the cache is asynchronously updated in the main database.

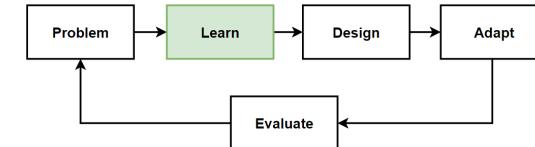


Cache-Aside Pattern for Microservices

- (1) When a client needs to access data, it first checks to see if the data is in the cache.
- (2) If the data is in the cache, the client retrieves it from the cache and returns it to the caller.
- (3) If the data is not in the cache, the client retrieves it from the database, stores it in the cache, and then returns it to the caller.
- Some of caching systems provide **read-through** and **write-through/write-behind** operations. In these systems, **client application retrieves data over by the cache**.
- For not supported Caches, it's the **responsibility the applications use the cache and update the cache** if there is a **cache-miss**.
- **Microservices** good example to implement **Cache-Aside pattern**, it is common to use a **distributed cache** that is **shared across multiple services**.



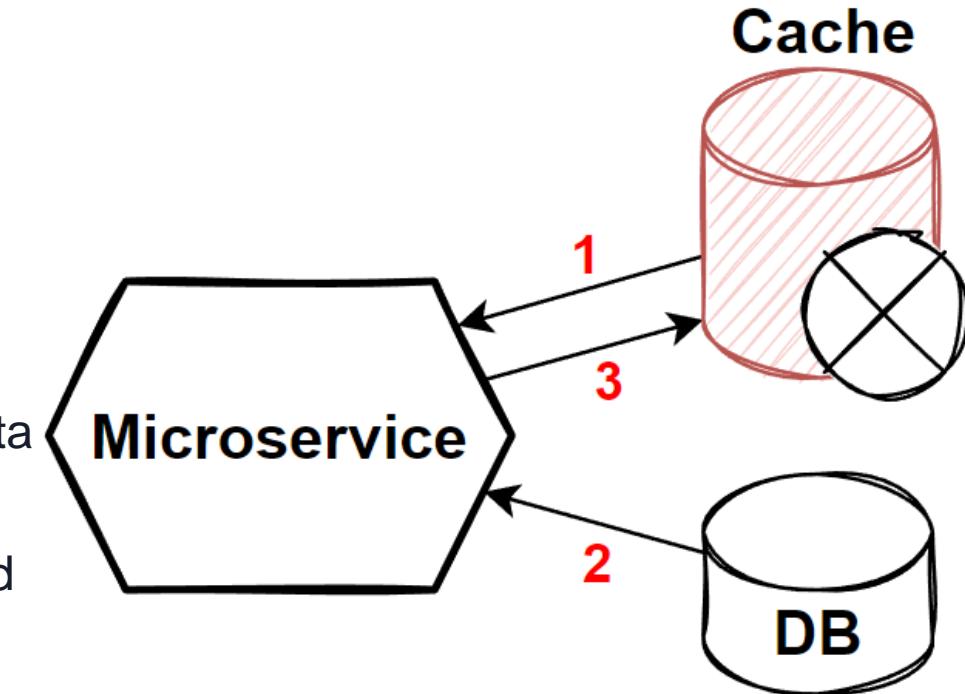
Cache-Aside Pattern for Microservices - 2



- Cache-aside pattern can **improve performance** of a **microservices architecture**, by **reducing the number of expensive database calls**.
- To use the Cache-aside pattern in a microservice, **need to implement a cache layer** in your service.
- Involve using a **Cache library** or framework, such as **Redis** or **Memcached**, or implementing a custom cache solution.

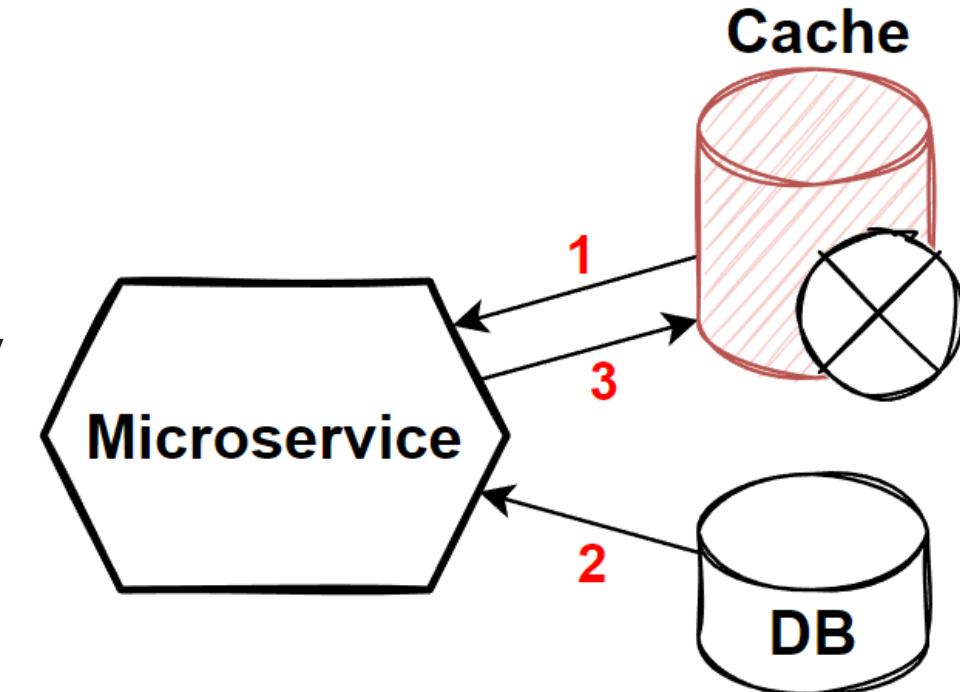
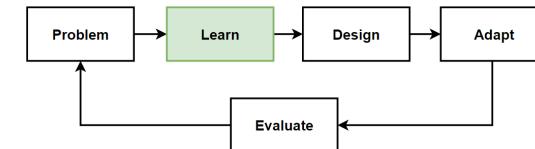
Process:

- When a service needs to access data, it first checks to see if the data is in the cache.
- If the data is in the cache, the service retrieves it from the cache and returns it to the caller.
- If the data is not in the cache, the service retrieves it from the database or other data store, stores it in the cache, and then returns it to the caller.



Drawbacks of Cache-Aside Pattern for Microservices

- Cache can introduce **additional complexity** and may not be suitable for all situations.
- The cache may **need to be invalidated** or **refreshed** when data is updated in the database or data store.
- This can **require additional coordination** between the microservices.
- The cache may introduce **additional latency** if it is located remotely from the microservices that are using it.



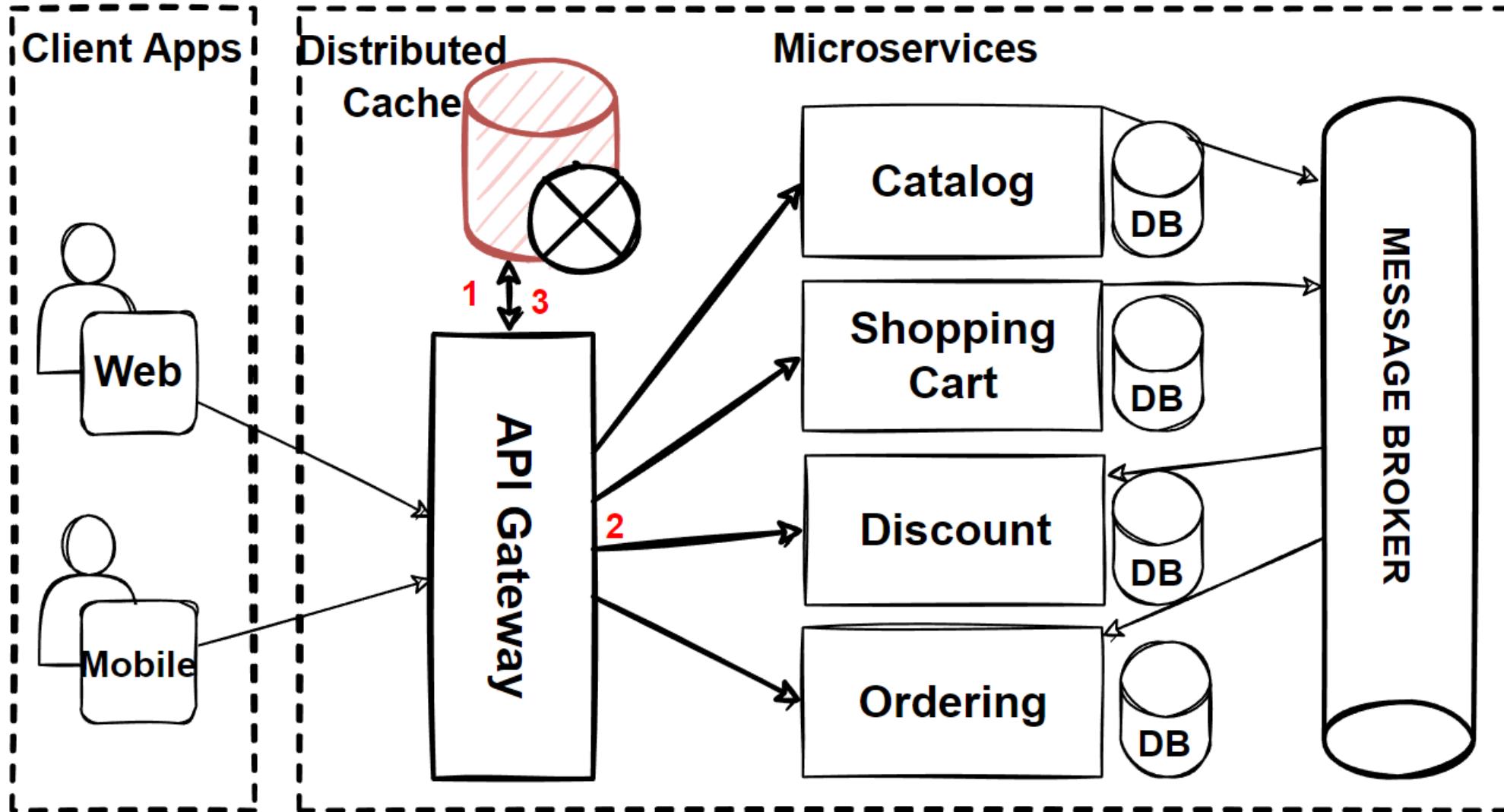
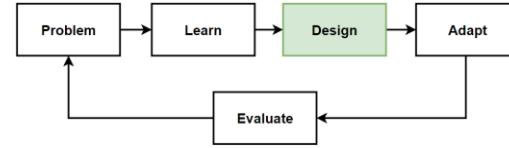
Before Design – What we have in our design toolbox ? - Old

Architectures	Patterns&Principles	Microservices EDA	Non-FR	FR
• Microservices Architecture	• The Database-per-Service Pattern	• Asynchronous, Decoupled communication	• High Scalability	• List products
• Event-Driven Microservices Architecture	• Polygot Persistence • Decompose services by scalability • The Scale Cube • Microservices Decomposition Pattern • Microservices Communications Patterns • Microservices Data Management Patterns • Event-Driven Architecture	• Event Hubs • Stream-Processing • Real-time processing • High volume events	• High Availability • Millions of Concurrent User • Independent Deployable • Technology agnostic • Data isolation • Resilience and Fault isolation	• Filter products as per brand and categories • Put products into the shopping cart • Apply coupon for discounts • Checkout the shopping cart and create an order • List my old orders and order items history

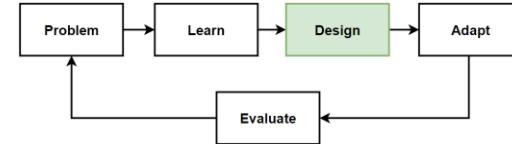
Before Design – What we have in our design toolbox ? - New

Architectures	Patterns&Principles	Microservices Caching	Non-FR	FR
• Microservices Architecture	• The Database-per-Service Pattern	• Caching Strategies	• High Scalability	• List products
• Event-Driven Microservices Architecture	• Polygot Persistence • Decompose services by scalability • The Scale Cube • Microservices Decomposition Pattern • Microservices Communications Patterns • Microservices Data Management Patterns • Event-Driven Architecture • Microservices Distributed Caching	• Cache Invalidation • Cache Hit - Cache Miss • Cache-Aside Pattern	• High Availability • Millions of Concurrent User • Independent Deployable • Technology agnostic • Data isolation • Resilience and Fault isolation	• Filter products as per brand and categories • Put products into the shopping cart • Apply coupon for discounts • Checkout the shopping cart and create an order • List my old orders and order items history

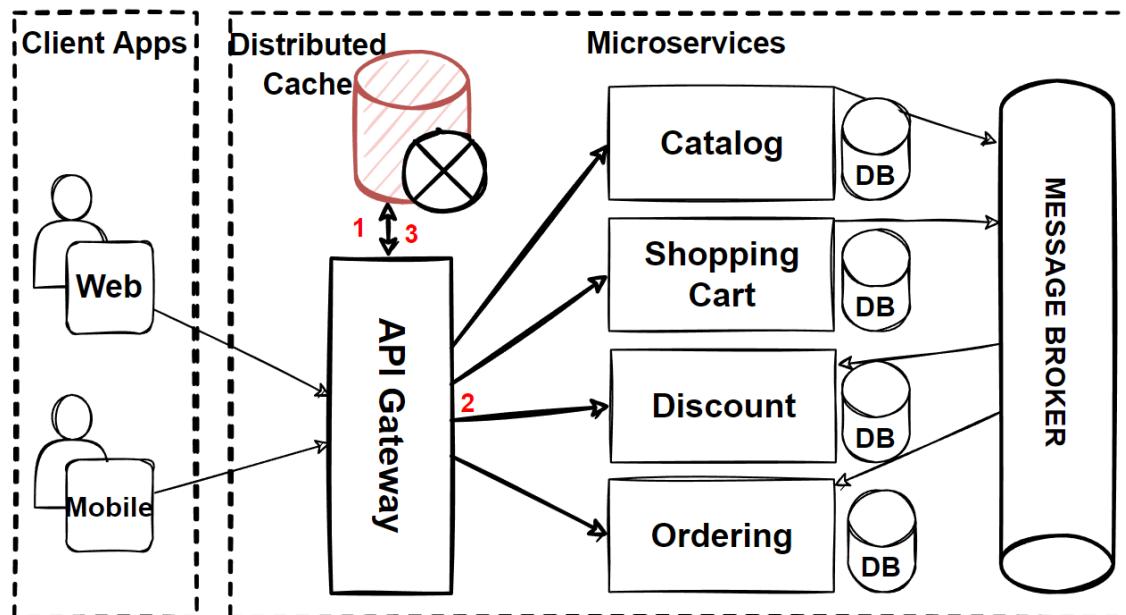
Microservices Distributed Caching with Cache-Aside Pattern



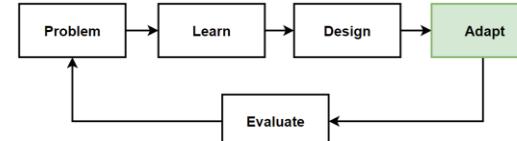
Microservices Distributed Caching apply Cache-Aside Pattern on API Gateway



1. User makes a request to an API gateway to retrieve some data.
2. API gateway determines which microservice is responsible for handling the request and receives the request and checks to see if the data is in the cache.
3. If the data is in the cache, API gateway retrieves it from the cache and returns it to the client.
4. If the data is not in the cache, API gateway forward responsible microservice to retrieves it from the database, and then returns it to the API gateway.
5. The API gateway receives the data from the microservice and stores it in the cache and returns it to the user.



Adapt: Microservices Distributed Caching



Frontend SPAs

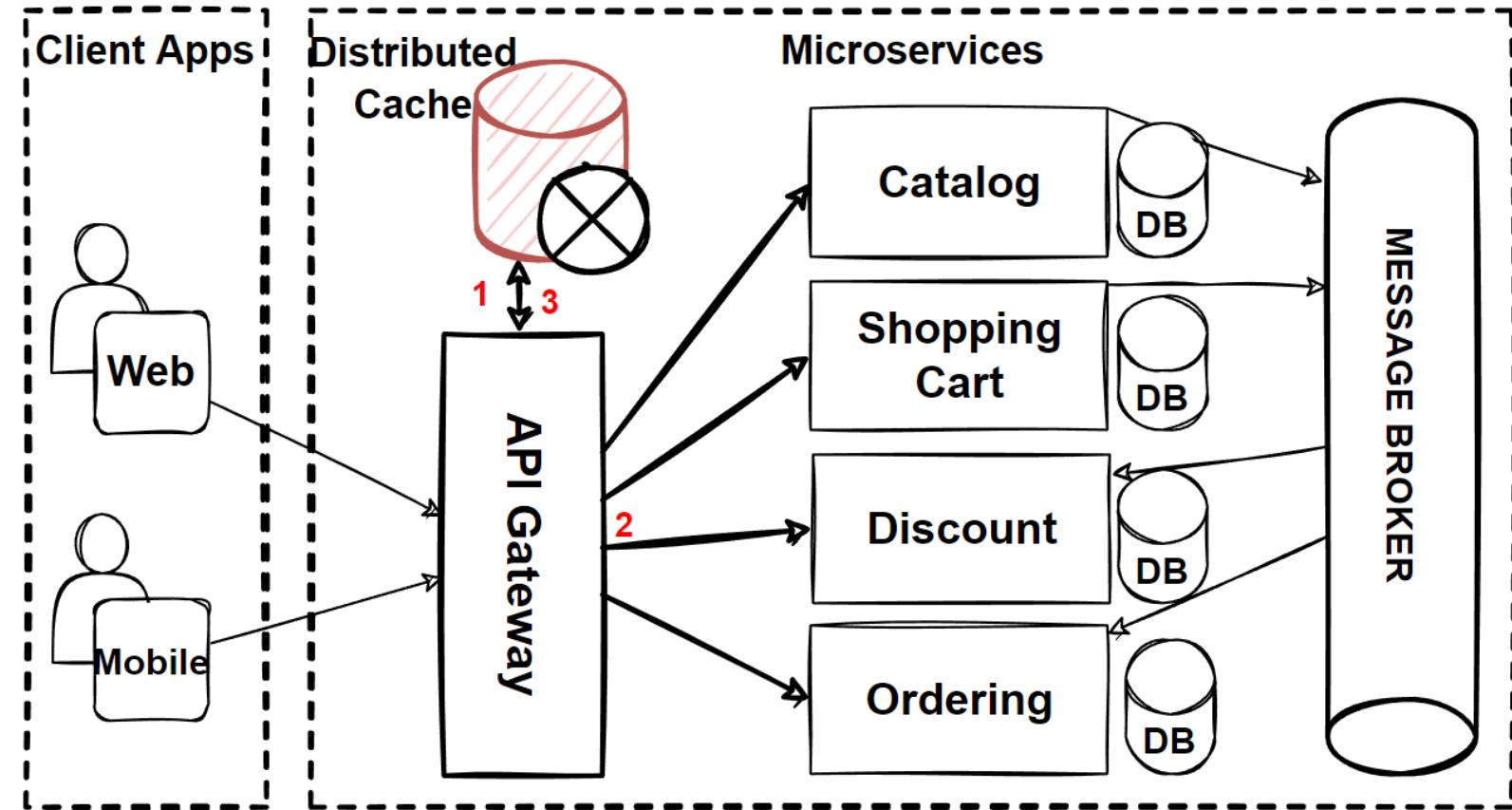
- Angular
- Vue
- React

API Gateways

- Kong Gateway
- Express Gateway
- Amazon AWS API Gateway

Backend Microservices

- Java – Spring Boot
- .Net – Asp.net
- JS – NodeJS



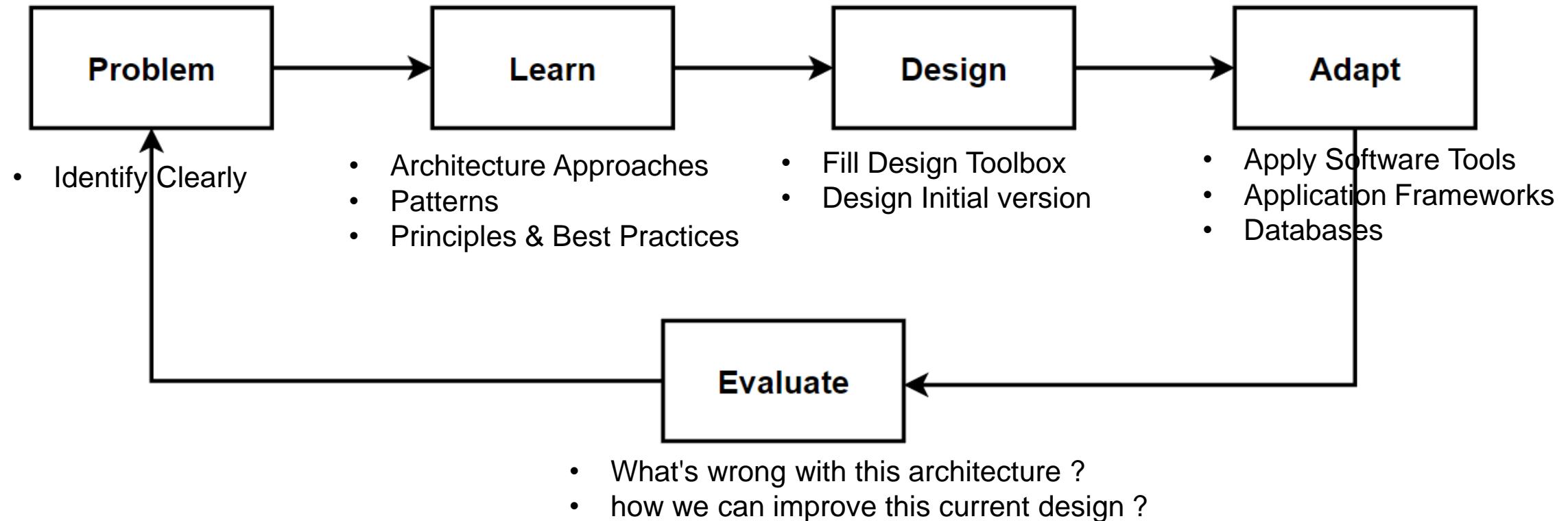
Open Source Distributed Caches

- Redis
- Memcached

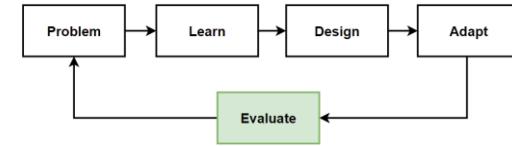
Cloud Distributed Caches

- Amazon ElastiCache
- Azure Cache for Redis

Way of Learning – The Course Flow



Evaluate: Microservices Distributed Caching

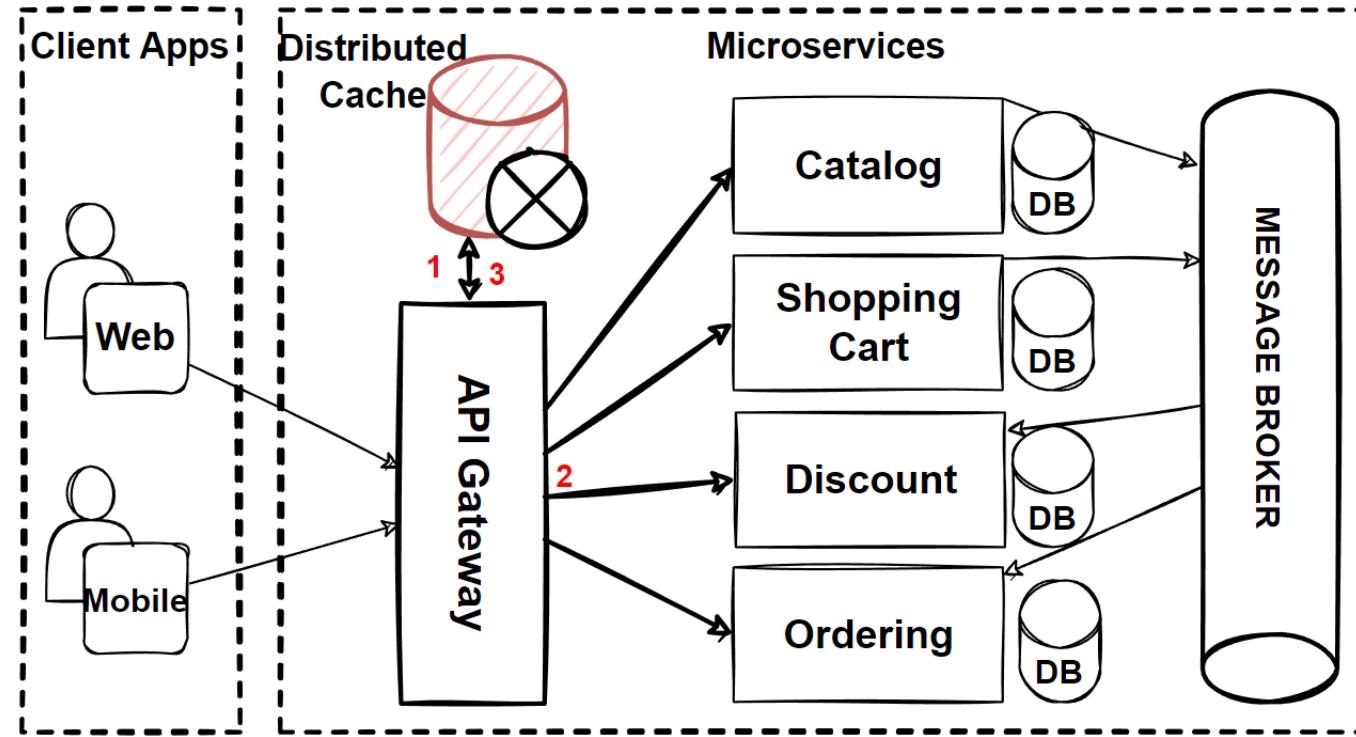


Benefits

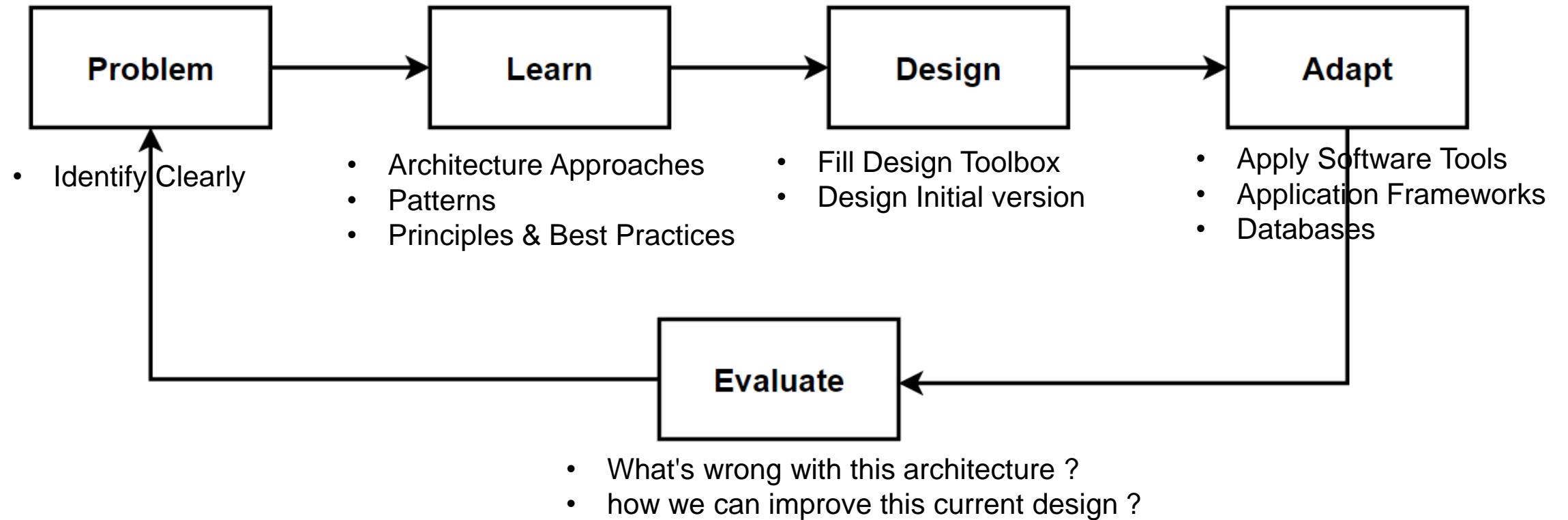
- **Improved performance**; by storing frequently accessed data in a cache
- **Enhanced scalability**; by offloading some of the workload from the database
- **Increased reliability**; cache can continue to serve data even if one of the nodes fails.

Drawbacks

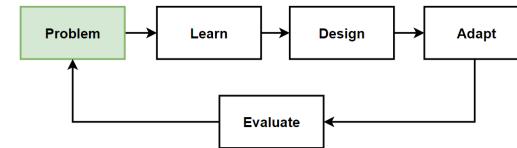
- **Increased Complexity**; cache can be complex and require specialized knowledge.
- **Cache invalidation**; Maintaining the consistency of the cache can be challenging, can lead to issues.
- **Cost**; managed services from cloud providers, can be expensive
- **Cache stampede**



Way of Learning – The Course Flow



Problem: Deploy Microservices at Anytime with Zero-downtime and flexible scale



Problems

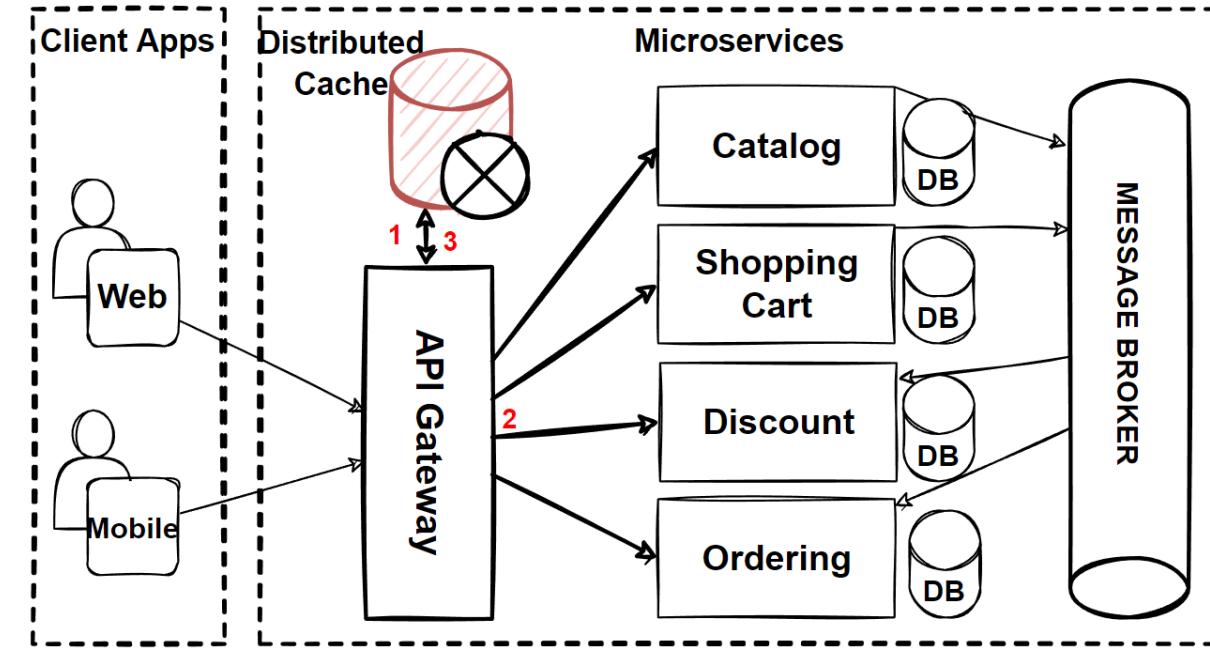
- Business teams wants to add new features immediately
- Innovate and experiment with new features
- Deploy features immediately, not waiting for deployment dates.
- Flexible scale for market peek times

Considerations

- Ensure continuity of service and minimize disruption
- Allow for continuous delivery
- Support high-traffic environments

Solutions

- Containers and Orchestrators
- Deployment strategies; blue-green deployment, rolling deployment, and canary deployment.
- Kubernetes Patterns; Sidecar Patterns, Service Mesh Pattern
- DevOps and CI/CD Pipelines and Infrastructure as code (IaC)



Microservices Deployments with Containers and Orchestrators

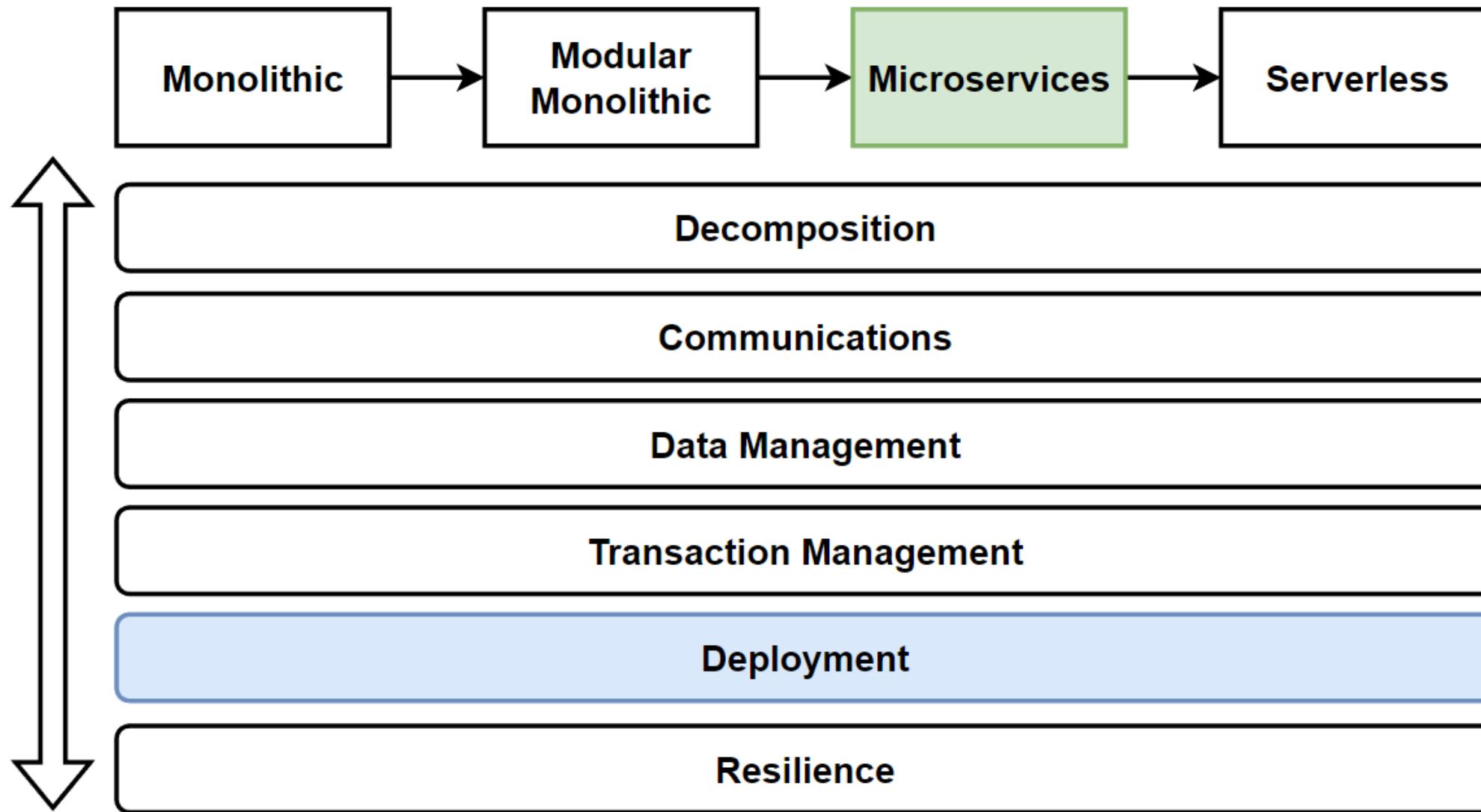
Deployment strategies; blue-green, rolling and canary deployment.

Kubernetes Patterns; Sidecar Patterns, Service Mesh Pattern

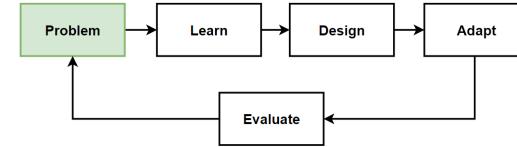
DevOps and CI/CD Pipelines

Infrastructure as code (IaC)

Architecture Design – Vertical Considerations



Problem: Deploy Microservices at Anytime with Zero-downtime and flexible scale



Problems

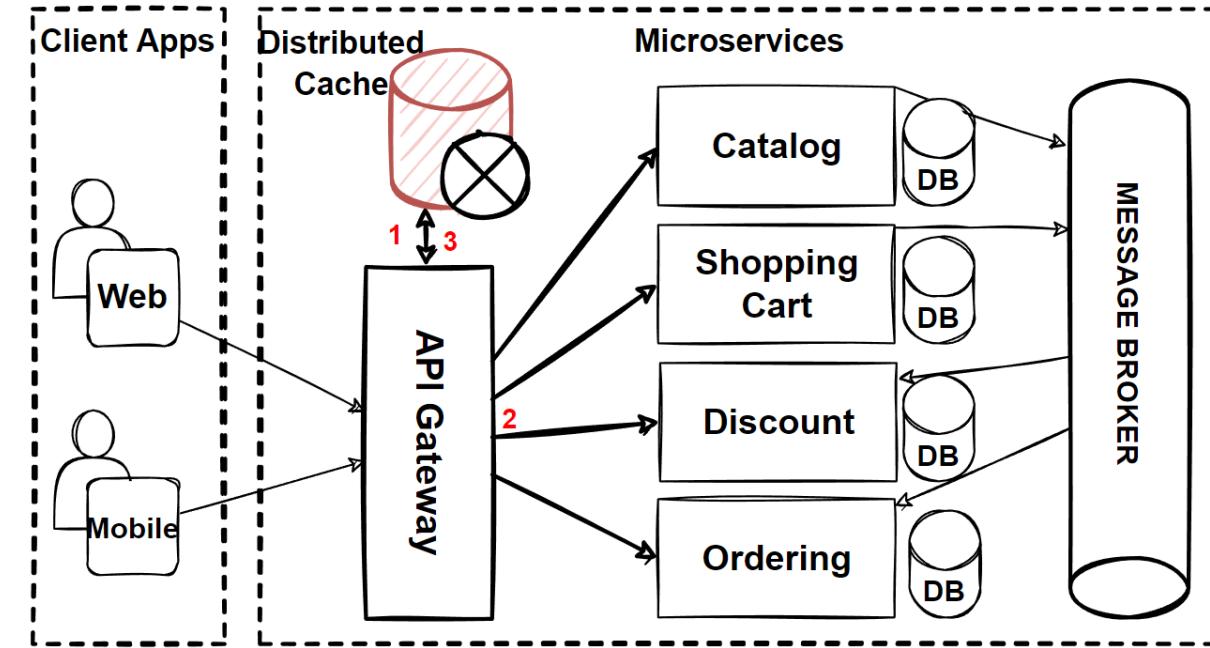
- Business teams wants to add new features immediately
- Innovate and experiment with new features
- Deploy features immediately, not waiting for deployment dates.
- Flexible scale for market peek times

Considerations

- Ensure continuity of service and minimize disruption
- Allow for continuous delivery
- Support high-traffic environments

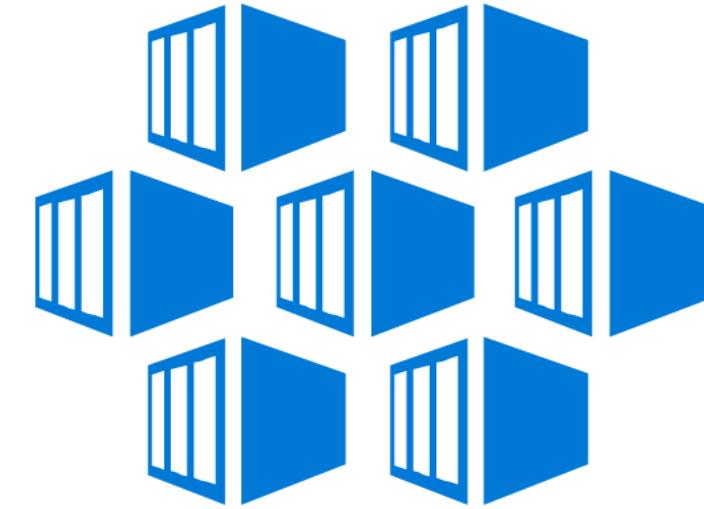
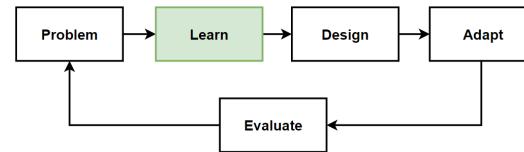
Solutions

- Containers and Orchestrators
- Deployment strategies; blue-green deployment, rolling deployment, and canary deployment.
- Kubernetes Patterns; Sidecar Patterns, Service Mesh Pattern
- DevOps and CI/CD Pipelines and Infrastructure as code (IaC)



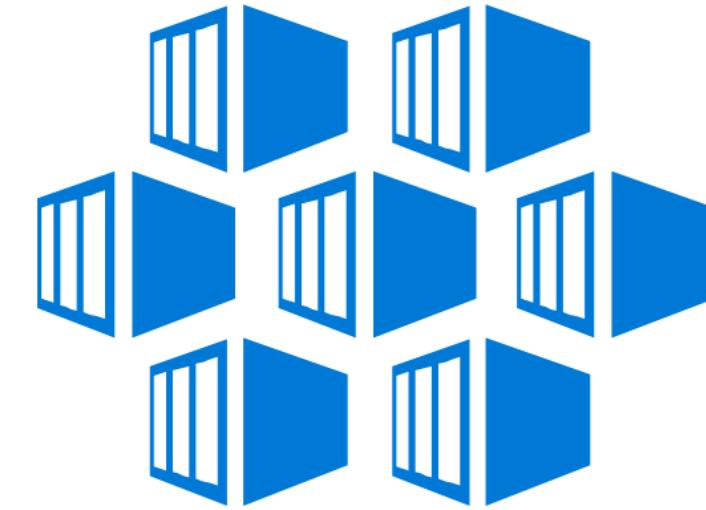
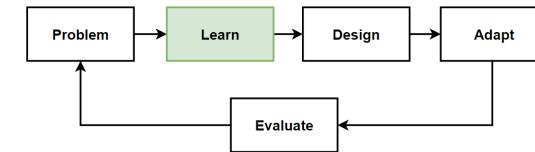
What are Containers ?

- Monolithic applications are deployed as a **single unit** and **deploy whole application** in one time.
- This caused **temporarily un-available** time of application, **needs to rollback** the **whole deployment** process. **Can't split modules** and scale independently.
- It solved in **microservices** architecture with **Containers** and **Orchestrators**.
- **Containers** are **package** and **distribute software applications**, makes them easy to **deploy** and **run** on **different environments**.
- Allow developers to **package an application** and its **dependencies** into a **single, self-contained** unit container images, **easily shipped** and **run** on any computer that has a container runtime.
- **Containers** are often used in the **deployment of microservices**, **independent components** can be developed, tested, and deployed separately.
- **Each microservice** is typically a **self-contained unit**, communicates with other microservices through well-defined interfaces.



What are Containers ? - 2

- **Containers** provide to **decouple applications** with their own **os**, **dependencies** and **libraries**, perfect match to microservices deployments.
- Microservices can **deployed separately** in a **container**. Each microservice can **deploy independently** with containers.
- Since **deployed microservices separate container**, we can **scale** as per their **volume of traffics**.
- **Changes** can be **applied independently** while other container stay not changed.
- **New features** can be **applied** and **rollback** very easy with **container deployments**.
- **Docker** is defacto standard for **containerization of microservices**.



Advantages of Containers

- **Isolation**

Each microservice runs in its own container, provides isolation from the other microservices and the host operating system.

- **Scalability**

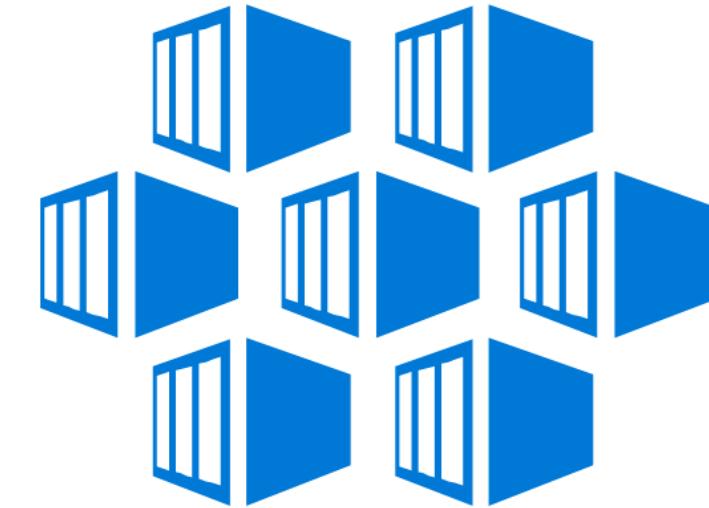
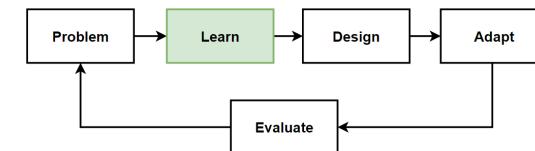
Containers make it easy to scale microservices horizontally by simply running more instances of a containerized microservice.

- **Portability**

Containers allow microservices to be easily deployed and run on any computer or cloud platform that supports container runtime.

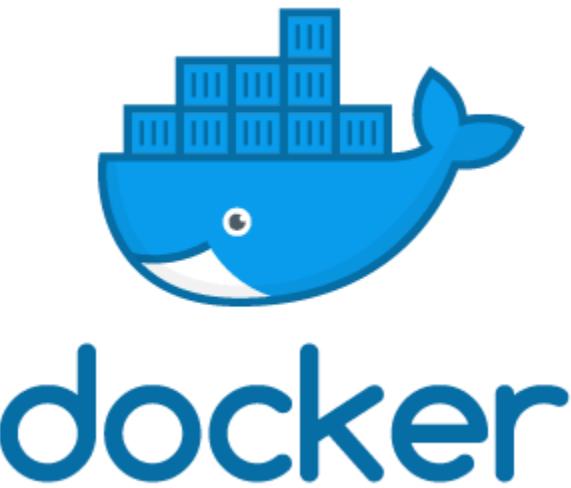
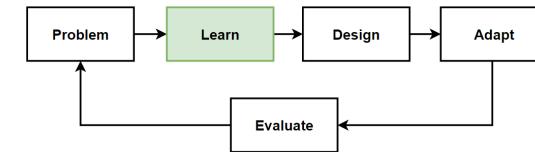
- **Resiliency**

Microservices are intended to be independently deployable and scalable. Using containers in the deployment of microservices that can be quickly started and stopped, increase the resiliency of the application.



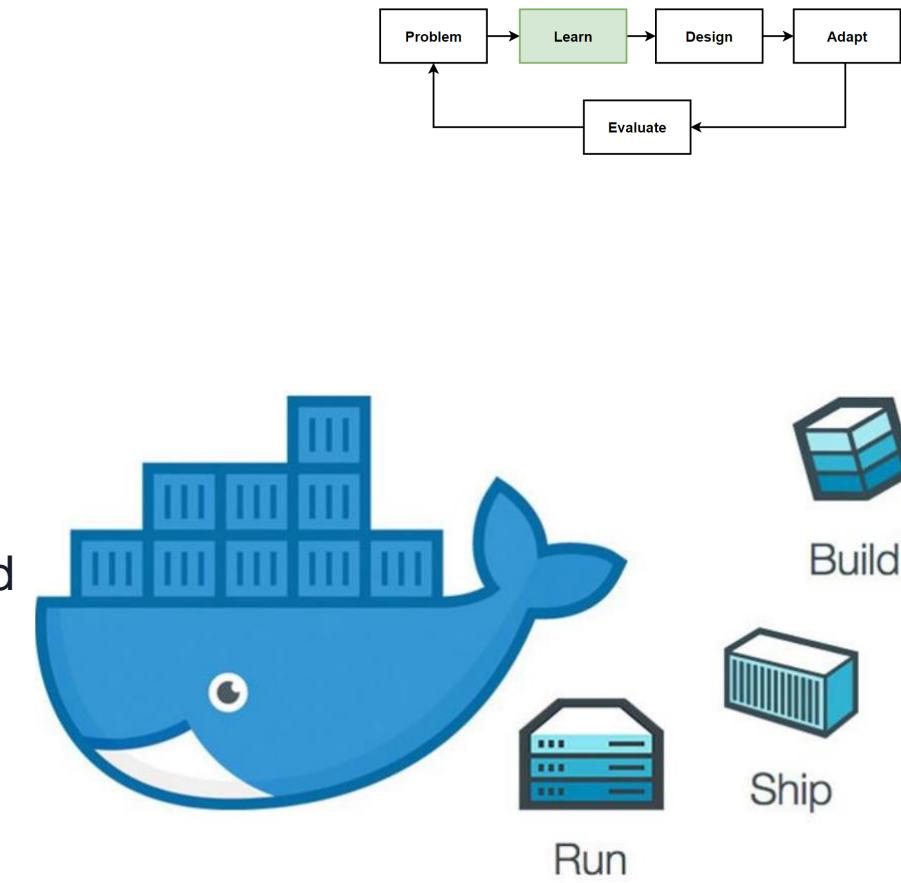
What is Docker ?

- Docker is an **open platform** for **developing, shipping, and running** applications.
- **Separate your applications from your infrastructure** so you can deliver software quickly.
- Advantages of Docker's methodologies for **shipping, testing, and deploying** code **quickly**.
- **Significantly reduce** the delay between **writing code** and **running** it in production.
- Automating the deployment of applications as **portable, self-sufficient containers** that can run on the **cloud or on-premises**.
- **Docker containers** can run anywhere, in your local computer to the cloud.
- **Docker image** containers can run **natively** on **Linux and Windows**.
- **Docker** is defacto standard for **containerization** of **microservices**.

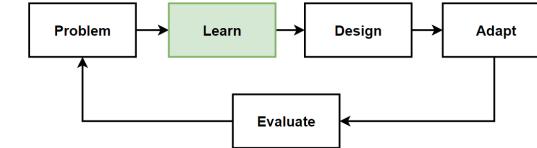


Docker Containers, Images, and Registries

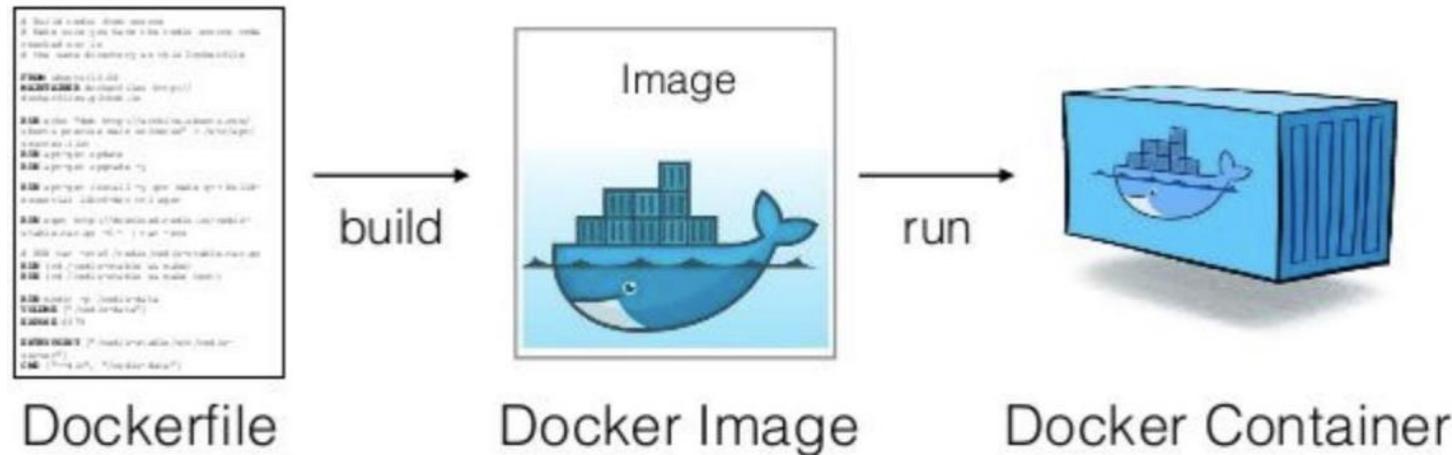
- Developer **develops** and **packages** application **with its dependencies** into a **container image**, that is a **static representation** of the application **with its configuration** and **dependencies**.
- To run the application, the application's image is **instantiated to create** a **container**, which will be running on the Docker host.
- Store **images in a registry**, which is a **library of images** and is needed when deploying to production orchestrators.
- **Docker images are stores** a public **registry** like **Docker Hub**, **Azure Container Registry**.
- Developer **creates container** in local and **push the images** the **Docker Registry**.
- Developer **download existing image** from **registry** and create container from image in local environment.



Application Containerization with Docker



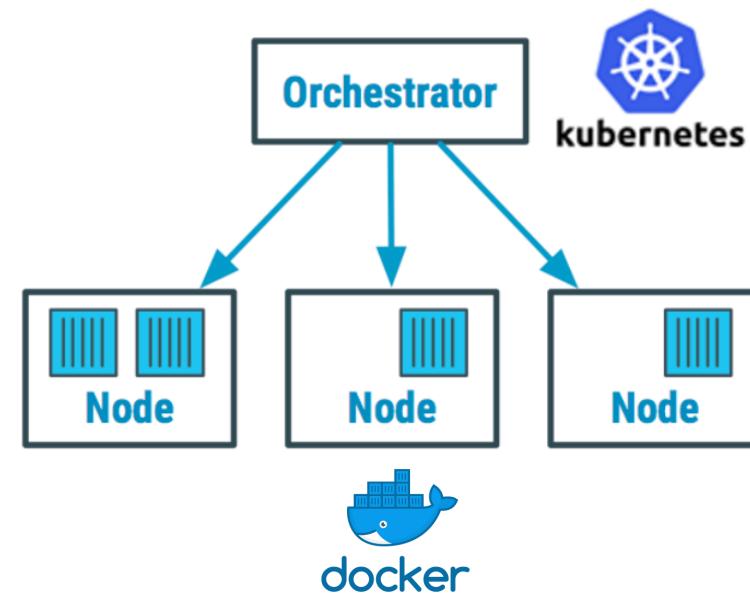
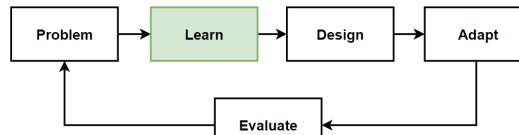
1. Write Dockerfile for our application.
 2. Build application with this docker file and creates the docker images.
 3. Run this images on any machine and creates running docker container from docker image.



- **Orchestrating** whole **microservices** application with **Docker** and **Kubernetes**.

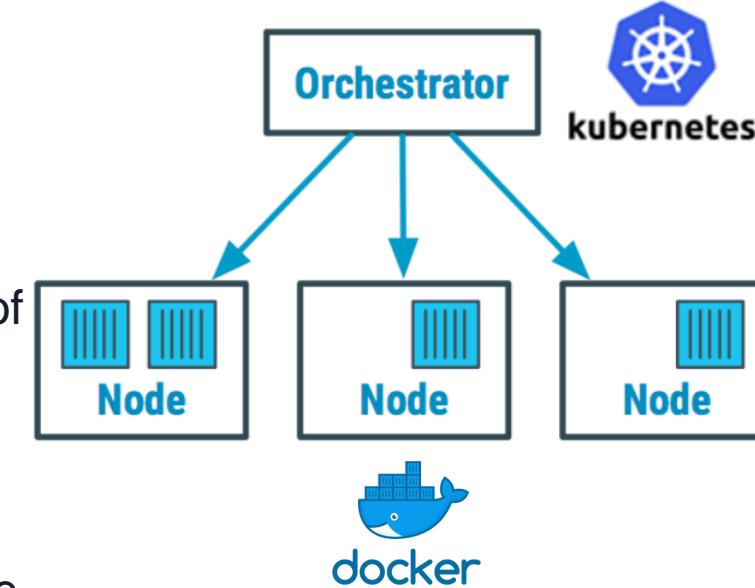
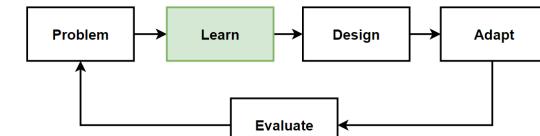
Why need Orchestrator for Containers ?

- Why we **need to orchestrate our containers** ?
- Think to developed and containerize our microservices and ready for shipping and deployment. Why we **need an orchestrator** for our **containers** ?
- **Deployment requirements of microservices**, think that have **hundreds of microservices**, we should ask;
 - How container instances be provisioned into cluster of multiple machines ?
 - After deployment, how will containers discover and communicate with each other ?
 - How can containers scale in or out on-demand and peek traffic ?
 - How do you monitor the health of each container ?
 - How do you protect containers against hardware and software failures ?
 - How do upgrade containers for a live application with zero downtime ?
- All these questions handled by **Container Orchestrators** that automate all these concerns.
- These task **can not be managed by manually** administrated for thousands of independently deployed containers.
- **Containerized services require automated management.**

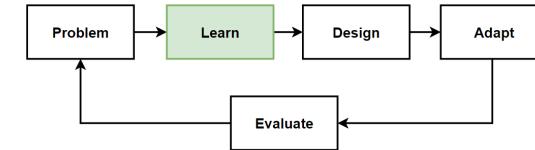


What is Container Orchestrator ?

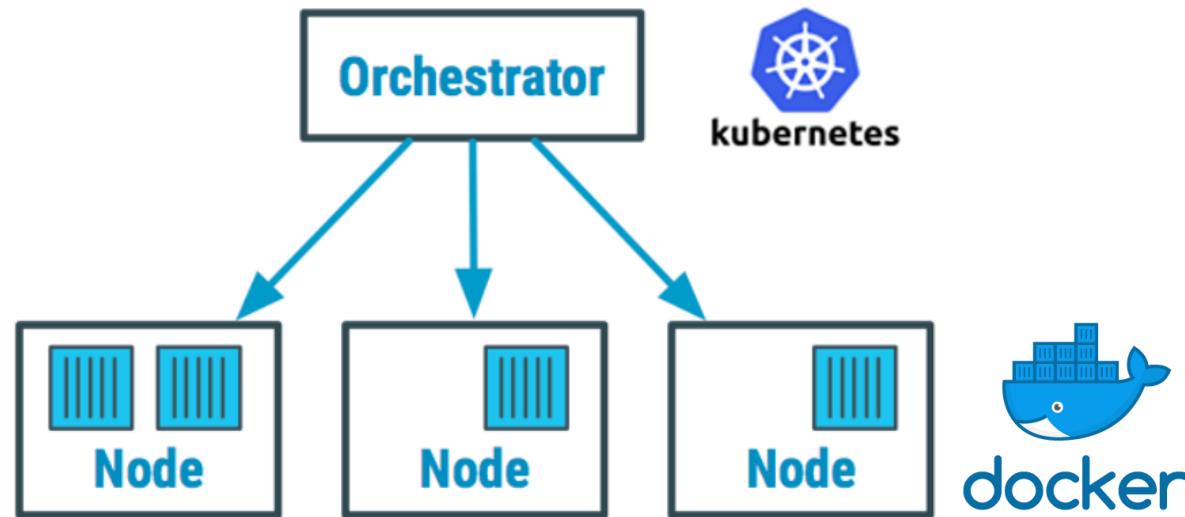
- Container orchestrators are **manage** and **automate** the **deployment**, **scaling**, and management of containerized applications.
- Enable to **run** and **manage** **microservices-based applications** in an efficient and scalable way by **abstracting the underlying infrastructure** and handling tasks such as **resource allocation**, **load balancing**, and **monitoring**.
- Microservices architectures, **containers need to orchestrate** to manage lots of container in your application cluster.
- **Orchestrators automates** the **deployment**, **scaling**, and **operational concerns** of containerized workloads across clusters.
- Container orchestrator **automates the deployment and management** of these microservices, making it **easier to scale and update application** as needed.
- There are several popular **container orchestrators** available, including **Kubernetes**, **Docker Swarm**, and **Apache Mesos**.



Application Containerization and Orchestration

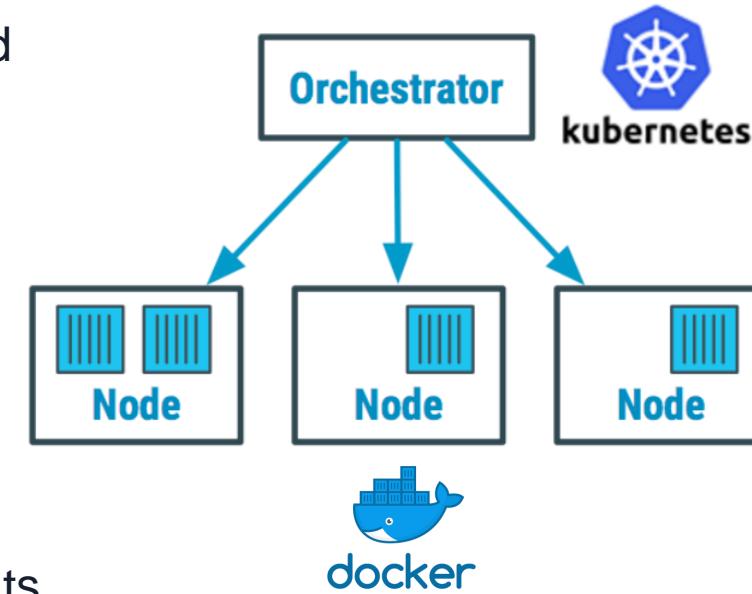
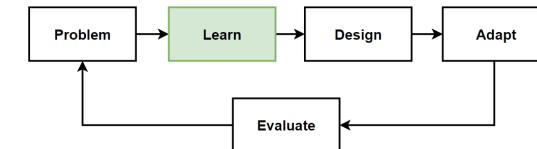


1. Package each of **microservices** into a **container image** and push it to a **container registry**.
2. Then use the **container orchestrator** to **deploy** the **microservices** to a cluster of nodes.
3. **Orchestrator** will handle tasks such as **scheduling containers** onto nodes, **monitoring** the health of the containers, and providing **self-healing** capabilities if a container fails.



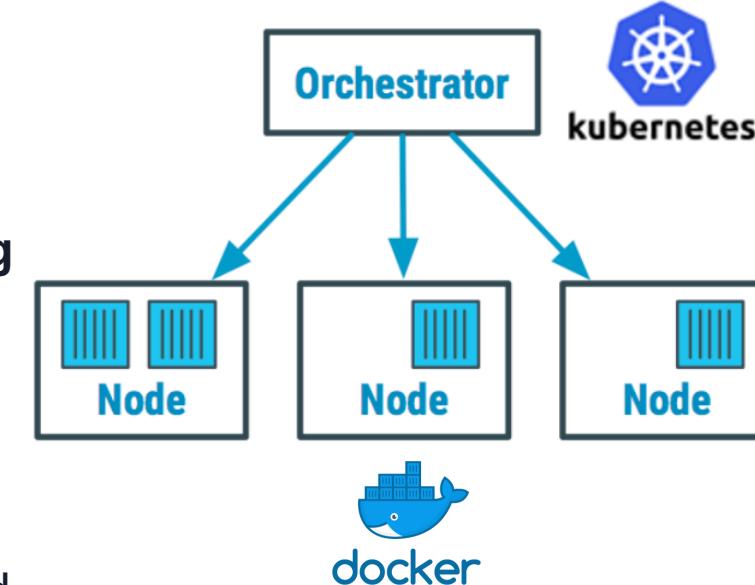
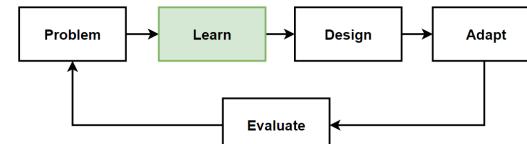
Benefits of Using Container Orchestration

- **Automation**
orchestrator can automate the deployment and management of your microservices.
- **Scalability**
orchestrator can automatically scale your microservices up or down as needed meet changing demand.
- **High availability**
orchestrator can provide self-healing capabilities to ensure that your microservices remain available even if individual containers fail.
- **Improved resource utilization**
Orchestrator can optimize the use of resources in your cluster.
- **Portability**
Containerized applications can be easily moved between different environments and platforms.
- **Security**
orchestrators provide security features such as role-based access control and network segmentation.



What is Kubernetes ?

- **Kubernetes** (also known as **k8s** or "kube") is an open source **container orchestration platform**.
- **Automates** many of the manual processes involved in **deploying, managing, and scaling containerized applications**.
- Developed by **Google** and is now maintained by the **Cloud Native Computing Foundation (CNCF)**.
- **Kubernetes** is a **portable, extensible, open-source platform** for managing containerized workloads and services, both **declarative configuration** and **automation**.
- It has a large, **rapidly growing ecosystem**. Kubernetes services, support, and tools are widely available.
- **Create cluster groups** of hosts running Linux containers, Kubernetes helps you easily and efficiently manage those clusters.
- Kubernetes is designed to provide a **platform-agnostic way to manage containerized applications at scale**.



Benefits of Kubernetes

- **Self-healing**

Kubernetes can automatically restart containers that fail, ensuring that your microservices remain available even if individual containers go down.

- **Automatic scaling**

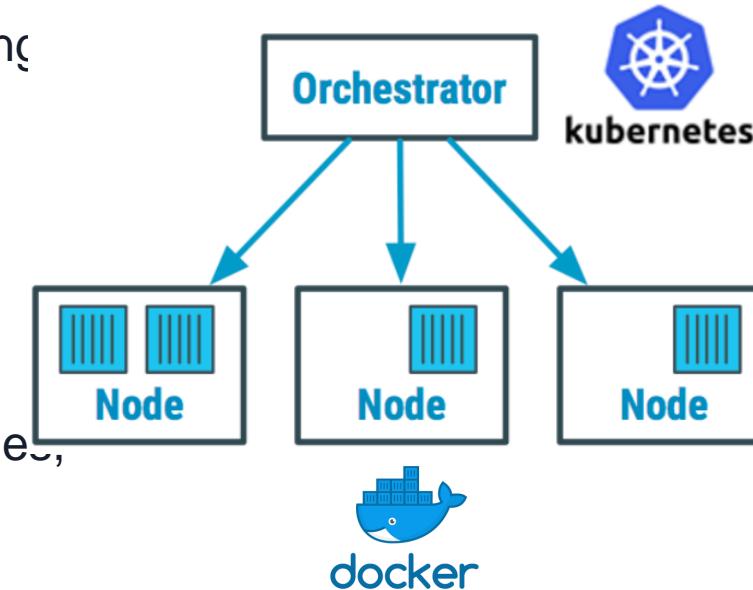
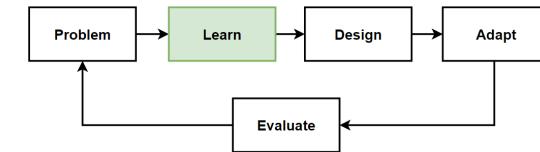
Automatically scale your microservices up or down as needed to meet changing demand, making it easier to handle sudden spikes in traffic.

- **Load balancing**

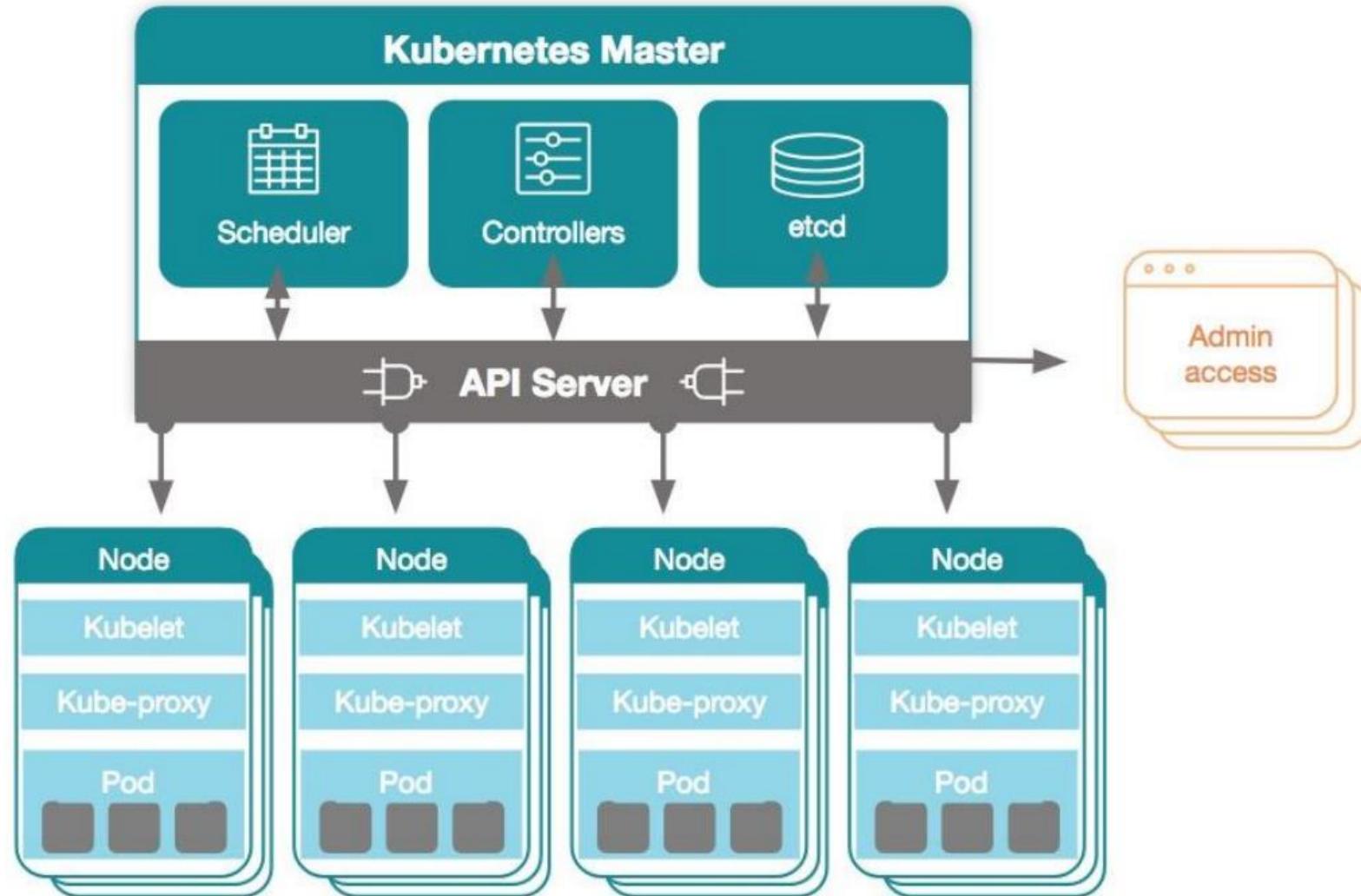
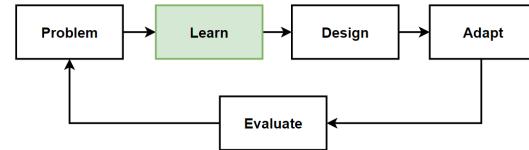
Automatically distribute incoming traffic across multiple instances of a microservice.

- **Declarative configuration**

Allows to specify the desired state of your microservices using configuration files, easy to automate the deployment and management of your applications.



Kubernetes Architecture



<https://kubernetes.io/docs/concepts/overview/components/>

Kubernetes Components

- **Pods**

Pods are the smallest deployable units of computing, that you can create and manage in Kubernetes. Pods stores and manage our docker containers.

- **ReplicaSet**

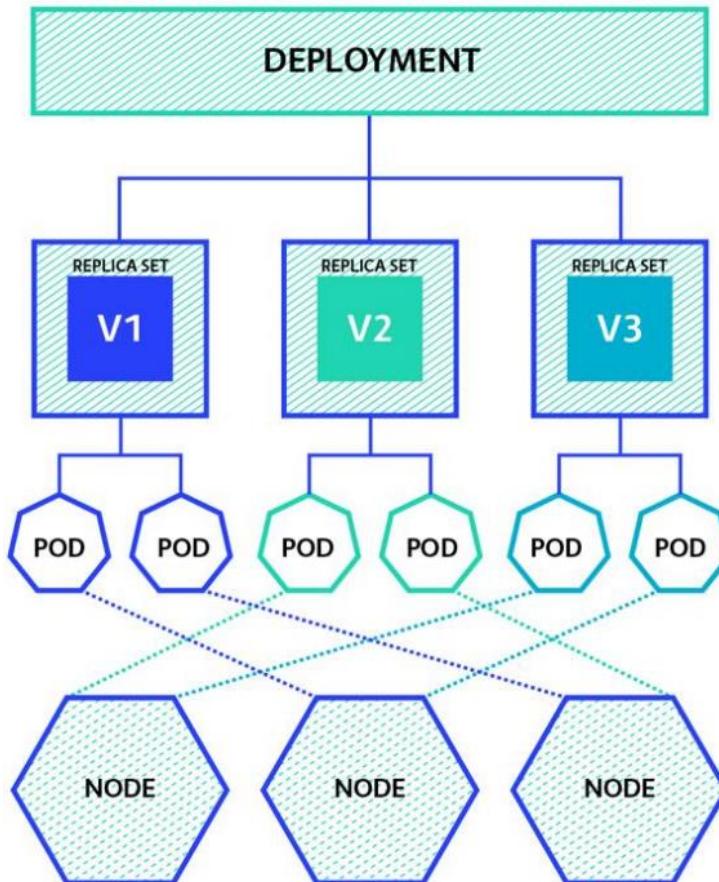
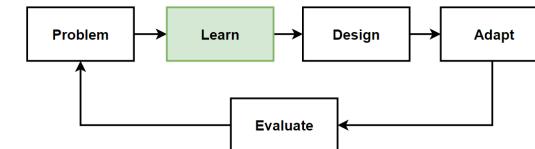
Maintain a stable set of replica Pods running at any given time. Used to guarantee the availability of a specified number of identical Pods.

- **Deployments**

Provides declarative updates for Pods and ReplicaSets. Describe a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate.

- **Deployments are an abstraction of ReplicaSets, and ReplicaSets are an abstraction of Pods.**

- **Pods should not created directly**, if needed, Deployment objects should be created.



Kubernetes Components - 2

- **Service**

Expose an application running on a set of Pods as a network service. Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them.

- **ConfigMaps**

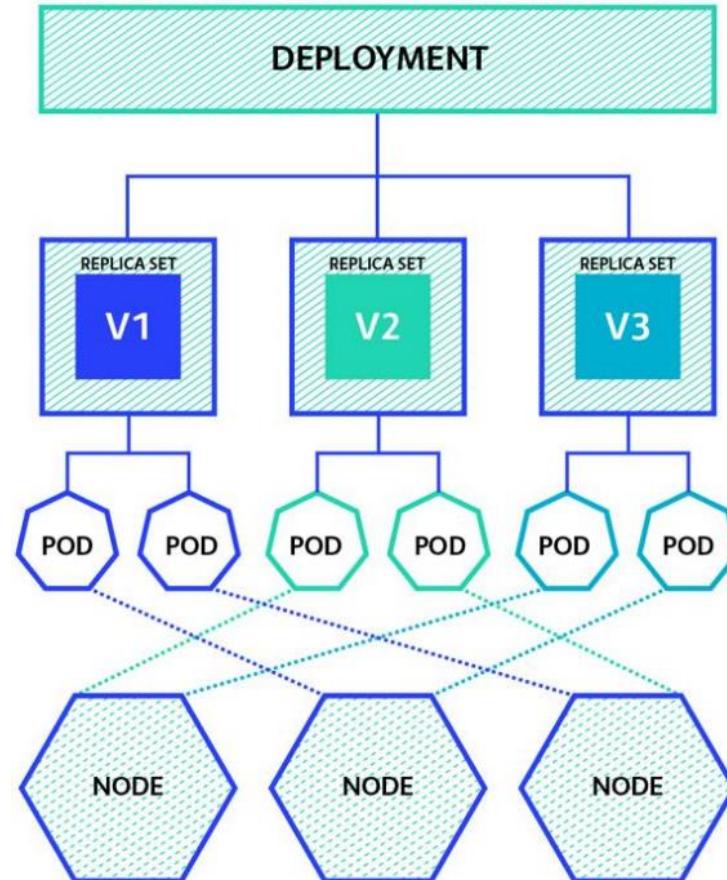
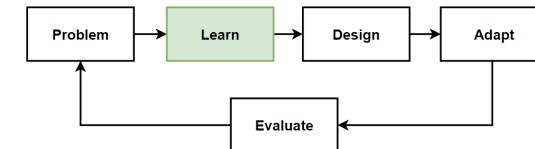
API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume.

- **Secrets**

Store and manage sensitive information, such as passwords, OAuth tokens, and ssh keys.

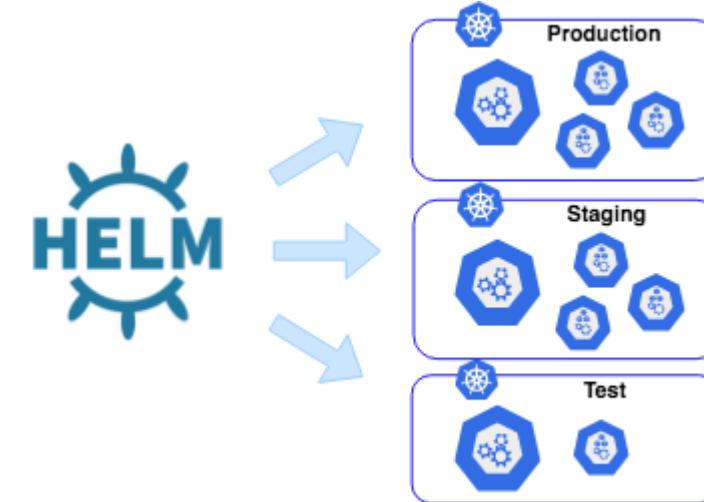
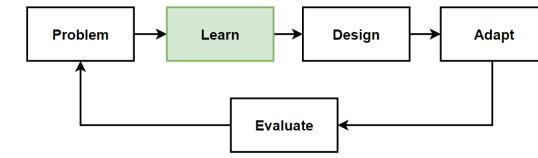
- **Volumes**

Persistent storage location that can be mounted into a pod. Volumes can be used to store data that needs to persist across container restarts or be shared between multiple containers in a pod.



Helm Charts with Kubernetes for Microservices Deployments

- Helm is a **package manager** for **Kubernetes** that makes it easier to manage and deploy applications in a Kubernetes cluster.
- Helm uses a **packaging format** called **charts**, which are **collections of files** that describe the various components of an application (such as pods, services, and deployments)
- It allows you to **automate the deployment of complex applications** in a **Kubernetes cluster**.
- Instead of manually creating and managing each component of your application separately, can use a **single Helm chart** to define all of the components and their relationships.
- This makes it **much easier to manage** and **update** your **application** over time.



Benefits of Helm Charts with Kubernetes for Microservices Deployments

- **Reusability**

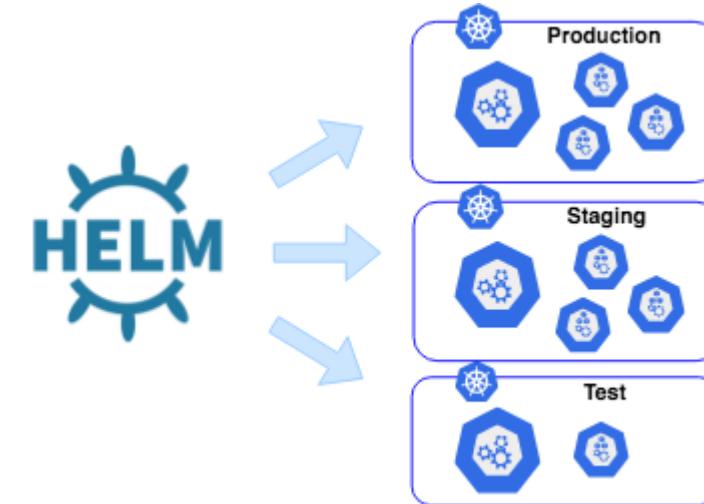
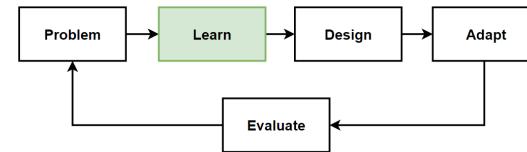
Helm charts can be shared and reused across multiple clusters, making it easy to deploy the same application in different environments.

- **Version control**

Allowing to track changes to your application over time and roll back to previous versions if needed.

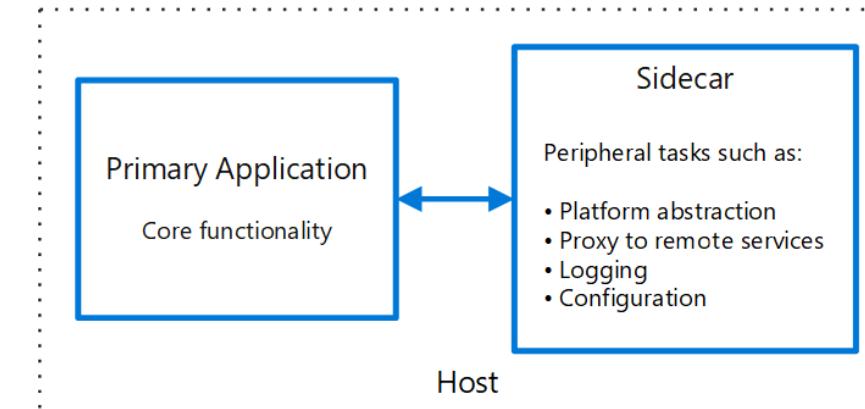
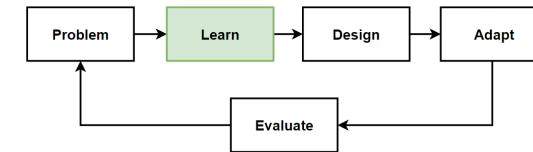
- **Customization**

Allow you to override the default settings for a chart and tailor it to your specific needs.



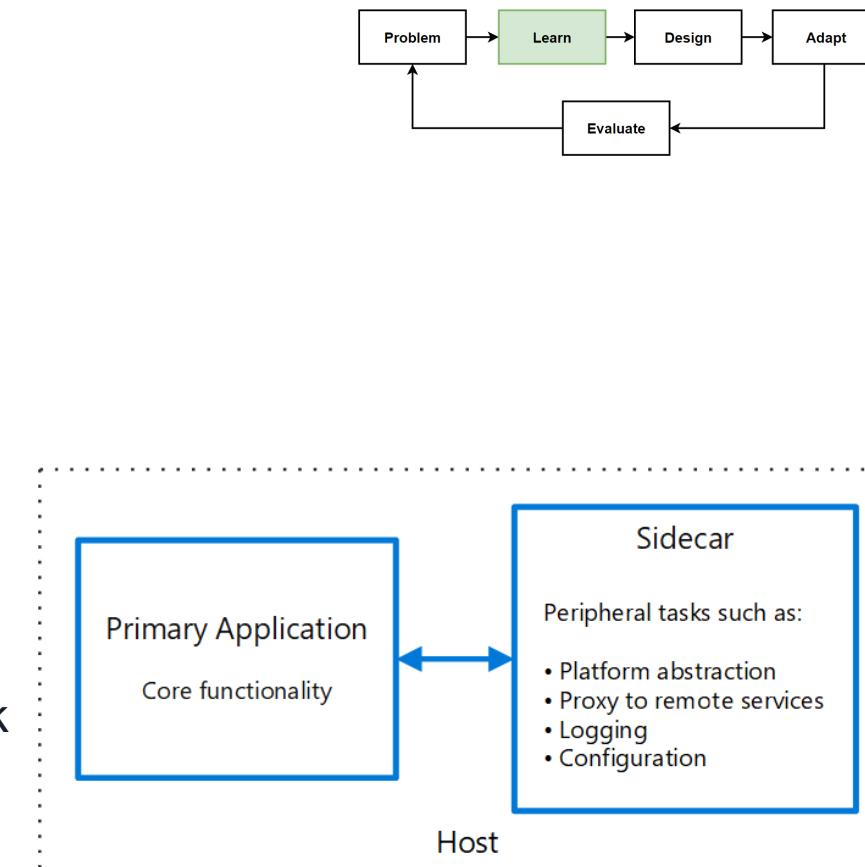
Sidecar Pattern

- **Sidecar pattern** is a design pattern in which a **helper container** is run alongside the **main container** in a pod.
- **The Sidecar container performs cross-cutting tasks** that are related to the main container.
- **Common use case for the sidecar pattern** is to add **logging** or **monitoring** functionality to a container.
- **A sidecar container** that runs a **logging agent** that sends application logs to a **central logging server**, and also **provide to monitoring** the **health** of the **main container**.
- **These cross-cutting tasks** can be implemented as **separate services** with **sidecar pattern** such as **monitoring**, **logging**, **configuration**, and **networking** services.
- The sidecar container and the main container **share the same network namespace**, that allows the sidecar container to **access** and **manipulate** the **data produced** by the **main container** as needed.



Benefits of Sidecar Pattern

- Allows to **add functionality** to a **container** without modifying the container itself.
- Useful if you want to **add logging or monitoring functionality** to an existing container image that you do not control.
- Allows you to **decouple the main container from the sidecar functionality**, easier to update or replace the sidecar without affecting the main container.
- Allows to run **multiple containers in a pod** that share the same network namespace, making it **easier to communicate** between them.
- Allows to **add common functionality** to multiple microservices **without having to modify the microservices** themselves.



Drawbacks of Sidecar Pattern

- **Increased complexity**

Adding an additional layer of complexity to your deployment, more difficult to understand and troubleshoot issues that arise.

- **Increased resource usage**

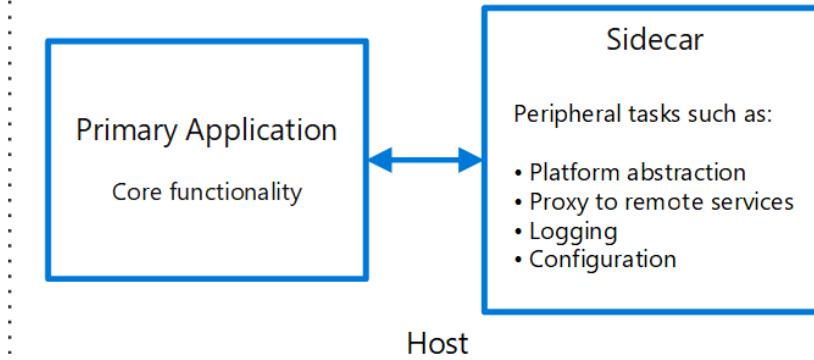
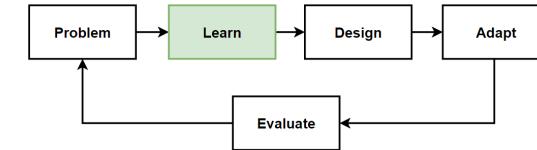
Running an additional container in a pod will increase the resource usage of the pod.

- **Decreased performance**

Pod can potentially decrease the performance of the pod, as the sidecar container will be competing for resources with the main container.

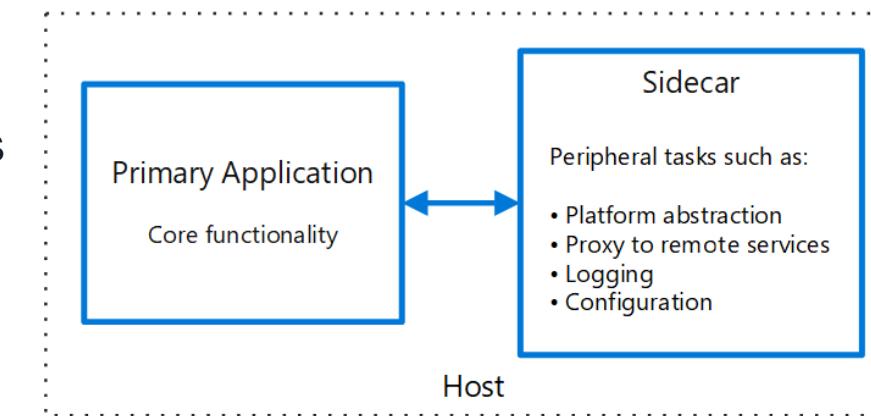
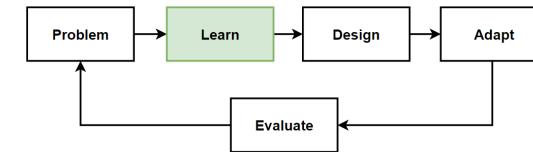
- **Limited flexibility**

Can be inflexible in some cases, as it requires that the main container and the sidecar container run in the same pod.



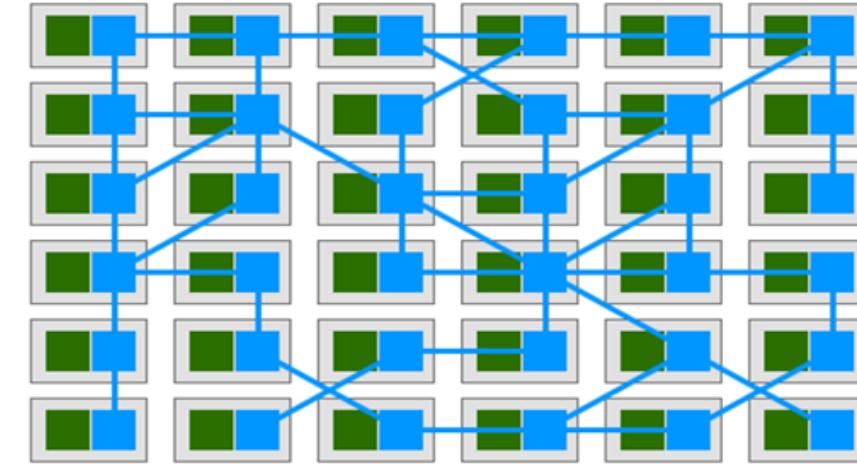
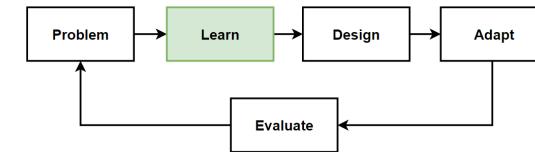
When to use Sidecar Pattern

- When you want to **add functionality** to an **existing container** image
- When you want to **decouple the main container** from the additional functionality
- When you want to **run multiple containers in a pod** that need to communicate with each other
- When you want to **add common functionality** to multiple microservices



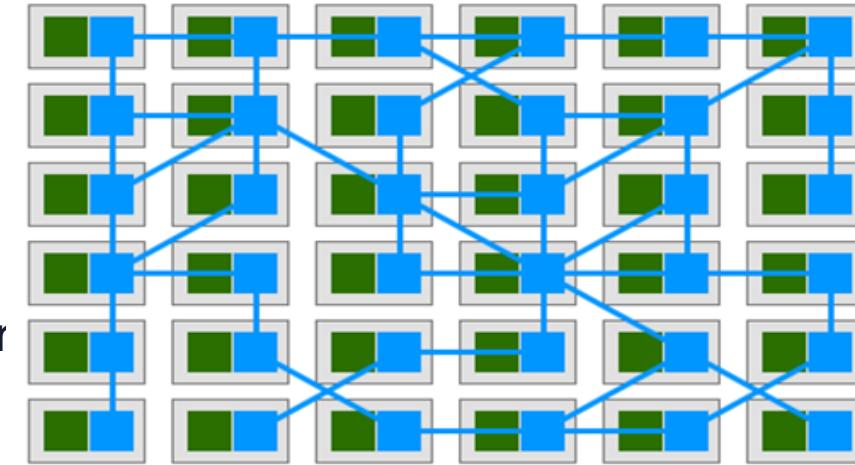
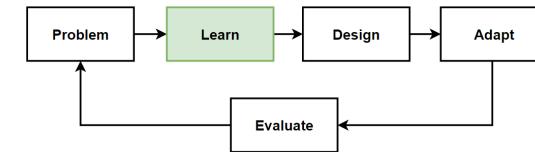
Service Mesh Pattern

- **Service mesh pattern** is managing the **communication** between **microservices** in a distributed system.
- Designed to provide a **uniform way to route traffic** between microservices, handle **load balancing**, and **monitor the healths**.
- Consists of a **set of proxy servers (sidecars)** that are deployed alongside the microservices. Proxy servers **handle the communication** between the microservices.
- **Responsible** for tasks such as **routing requests, load balancing, and monitoring the health** of the system.
- **Abstract away the complexities** of managing communication between microservices.
- **Instead of** having to **manage** these details at the **application level**, use the **service mesh** to handle them **automatically**.
- **Popular Service mesh** implementations, including **Istio** and **Linkerd**, set up and manage a service mesh in a **Kubernetes cluster**.

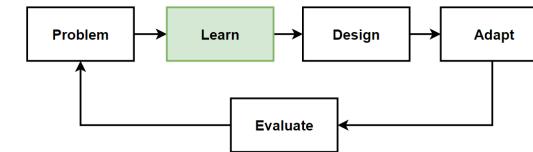


When to use Service Mesh Pattern

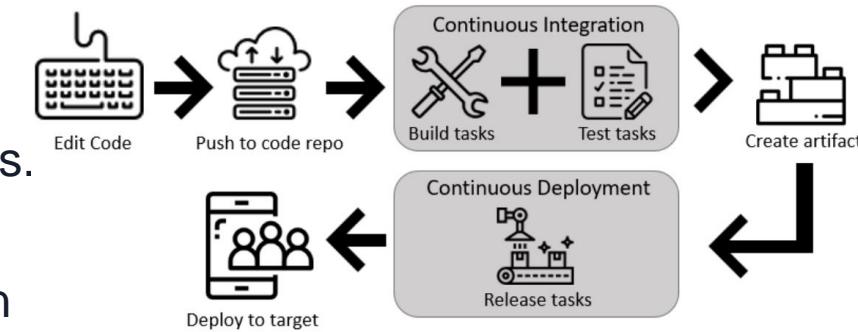
- When you want to **abstract away the complexities of managing communication** between microservices.
- When you want to **centralize the management** of communication between microservices.
- When you want to **add features** such as **load balancing**, **traffic management**, and **monitoring** to your microservices.
- **Using a service mesh**, can add these features to your microservices **without having to modify the microservices themselves**.
- **Service mesh pattern** is a useful tool for **managing the communication** between microservices in a distributed system.
- **Build and maintain complex microservices-based** systems by abstracting away the complexities of managing communication between the microservices.



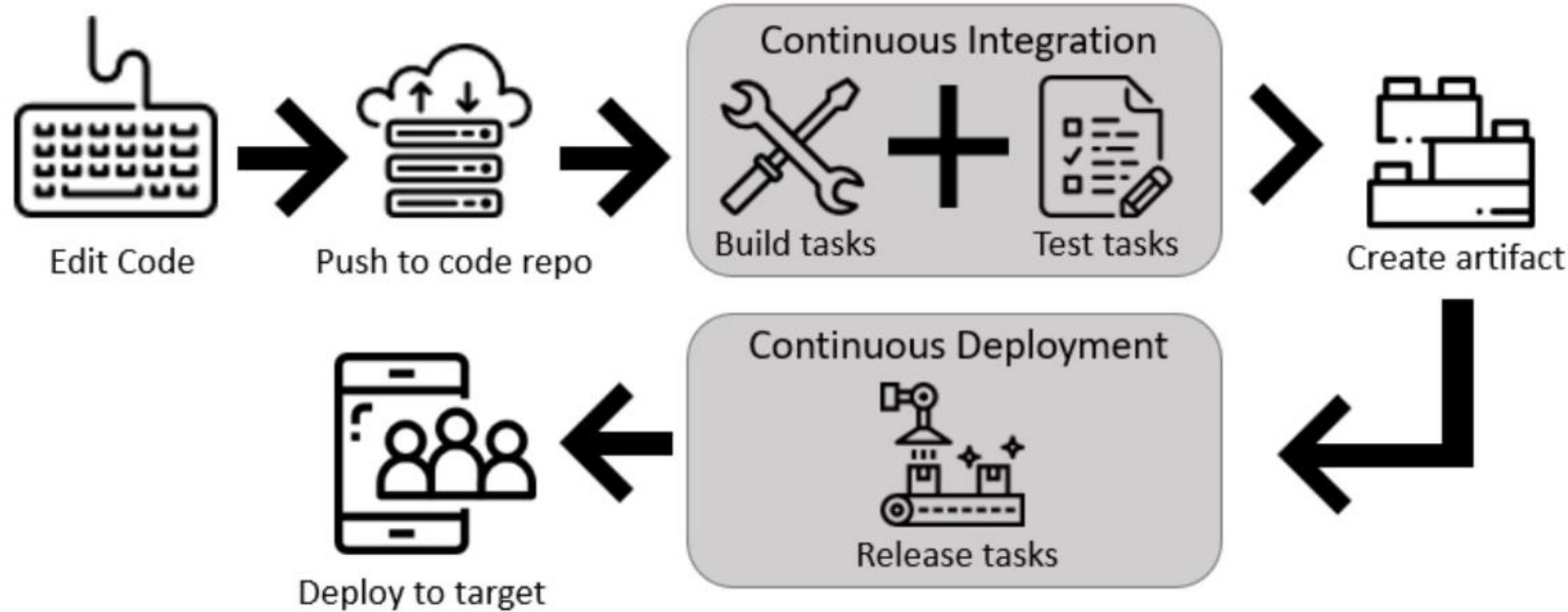
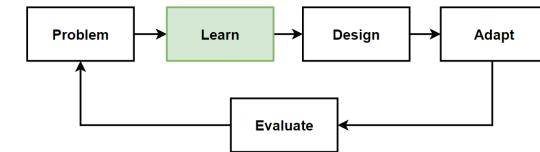
Devops and CI/CD Pipelines



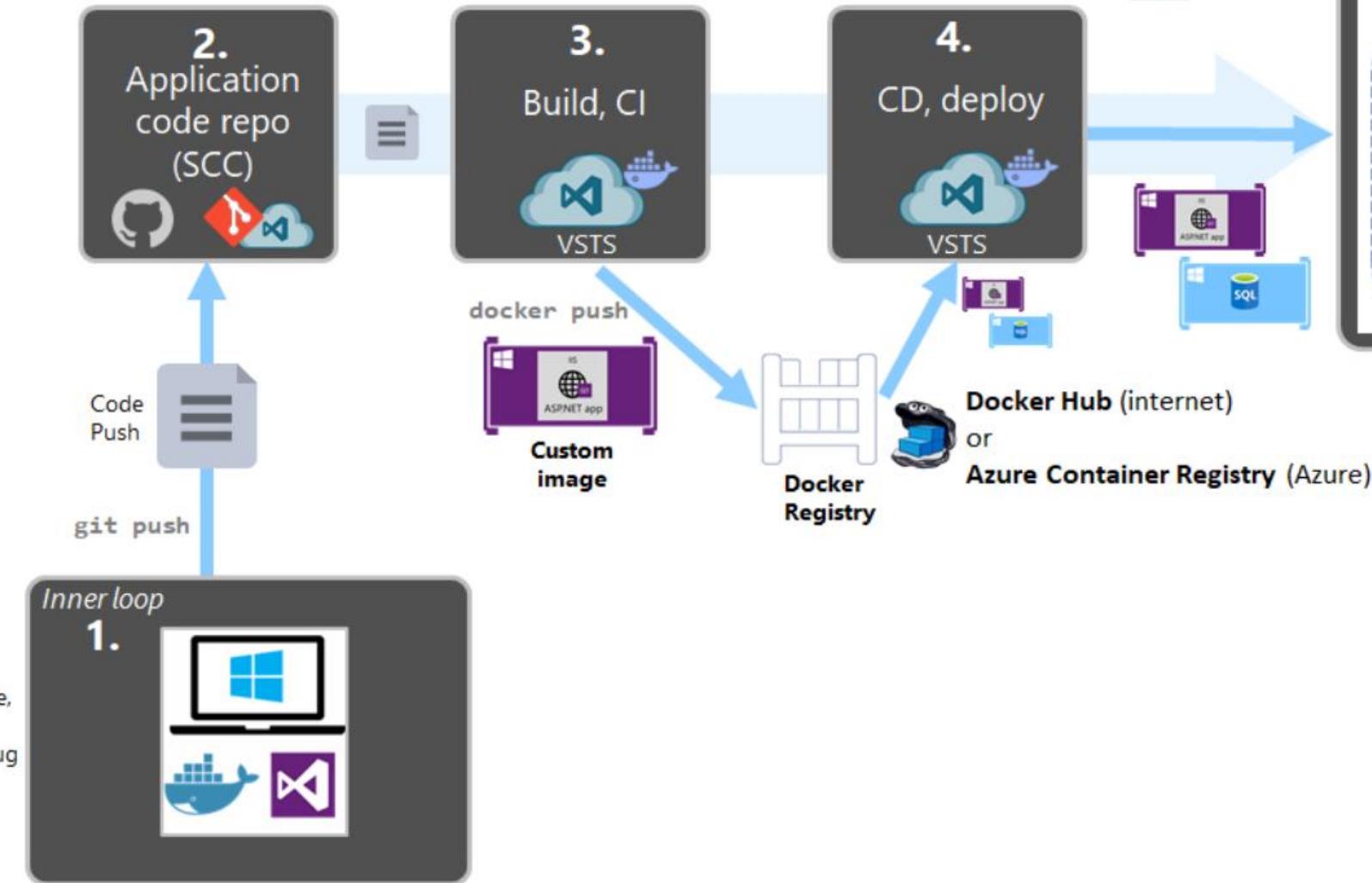
- **DevOps** is to **improve the efficiency and speed** of software **development** and **deployment** by **automating** and streamlining the processes involved.
- **Continuous Integration / Continuous Delivery (CI/CD) pipeline.**
- **CI/CD pipeline** is a **series of automated processes** that are used to **build**, **test**, and **deploy** software.
- **For microservices** deployments, a **CI/CD pipeline** can be used to **automate the process of building, testing, and deploying** microservices.
- **CI/CD pipeline** to **automatically build** and **test new versions** of a microservice, and then **deploy the new version** to a staging or production environment.
- **CI/CD pipeline** is to **enable teams to quickly and reliably deliver** changes to microservices **with zero-downtime**.
- **Deploy updates to individual microservices** as needed rather than having to deploy updates to the entire system at once.
- **Automate and streamline** the process of **building, testing, and deploying** microservices.



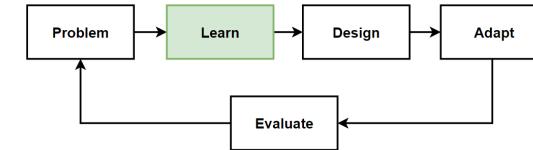
CI/CD Pipeline Steps for Microservices Deployments



CI/CD Tools for Microservices Deployments



Deployment Strategies for Microservices



- **Blue-green deployment**

Deploying updates to a new set of microservices (the "green" deployment), while the old version of the microservices (the "blue" deployment) remains running.

- **Rolling deployment**

Deploying updates to a subset of the microservices at a time, and then rolling the updates out to the rest of the microservices over time.

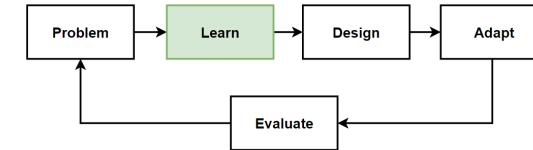
- **Canary deployment**

Deploying updates to a small subset of the microservices, and then gradually rolling the updates out to the rest of the microservices over time.

- **A/B testing**

Deploying updates to a subset of the microservices, and then comparing the performance of the updated microservices with the performance of the unmodified microservices.

Infrastructure as Code (IaC)



- Infrastructure as a codebase that can be managed and versioned in the same way as application code.
- IaC is to enable teams to manage their infrastructure in a more automated and repeatable way.
- IaC can be used to automate the process of deploying and managing the infrastructure needed to run the microservices.
- I.e. define the infrastructure needed to run your microservices in a Kubernetes cluster, including the pods, services, and other resources required.
- Terraform is a tool that allows you to define and manage infrastructure as code.
- Ansible is a tool that allows you to automate the deployment and management of infrastructure.
- They can be used to define the infrastructure needed to run your microservices in a Kubernetes cluster.
- IaC is a useful tool for automating the process of deploying and managing the infrastructure.
- Teams can manage their infrastructure in a more repeatable and automated way, making it easier to update and maintain their microservices over time.

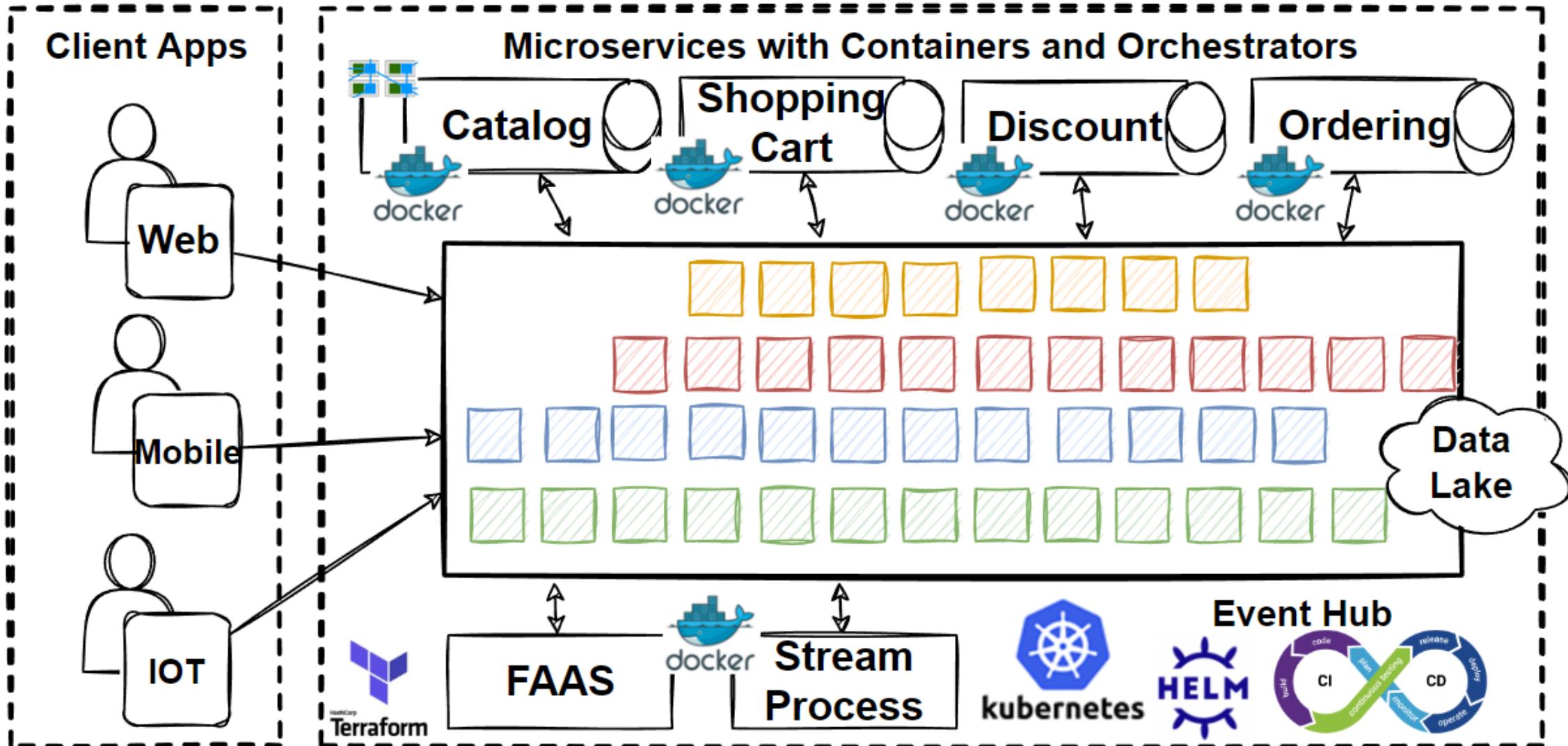
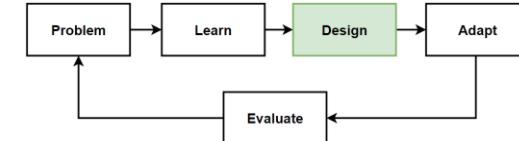
Before Design – What we have in our design toolbox ? - Old

Architectures	Patterns&Principles	Microservices Caching	Non-FR	FR
• Microservices Architecture	• The Database-per-Service Pattern	• Caching Strategies	• High Scalability	• List products
• Event-Driven Microservices Architecture	• Polygot Persistence • Decompose services by scalability • The Scale Cube • Microservices Decomposition Pattern • Microservices Communications Patterns • Microservices Data Management Patterns • Event-Driven Architecture • Microservices Distributed Caching	• Cache Invalidation • Cache Hit - Cache Miss • Cache-Aside Pattern	• High Availability • Millions of Concurrent User • Independent Deployable • Technology agnostic • Data isolation • Resilience and Fault isolation	• Filter products as per brand and categories • Put products into the shopping cart • Apply coupon for discounts • Checkout the shopping cart and create an order • List my old orders and order items history

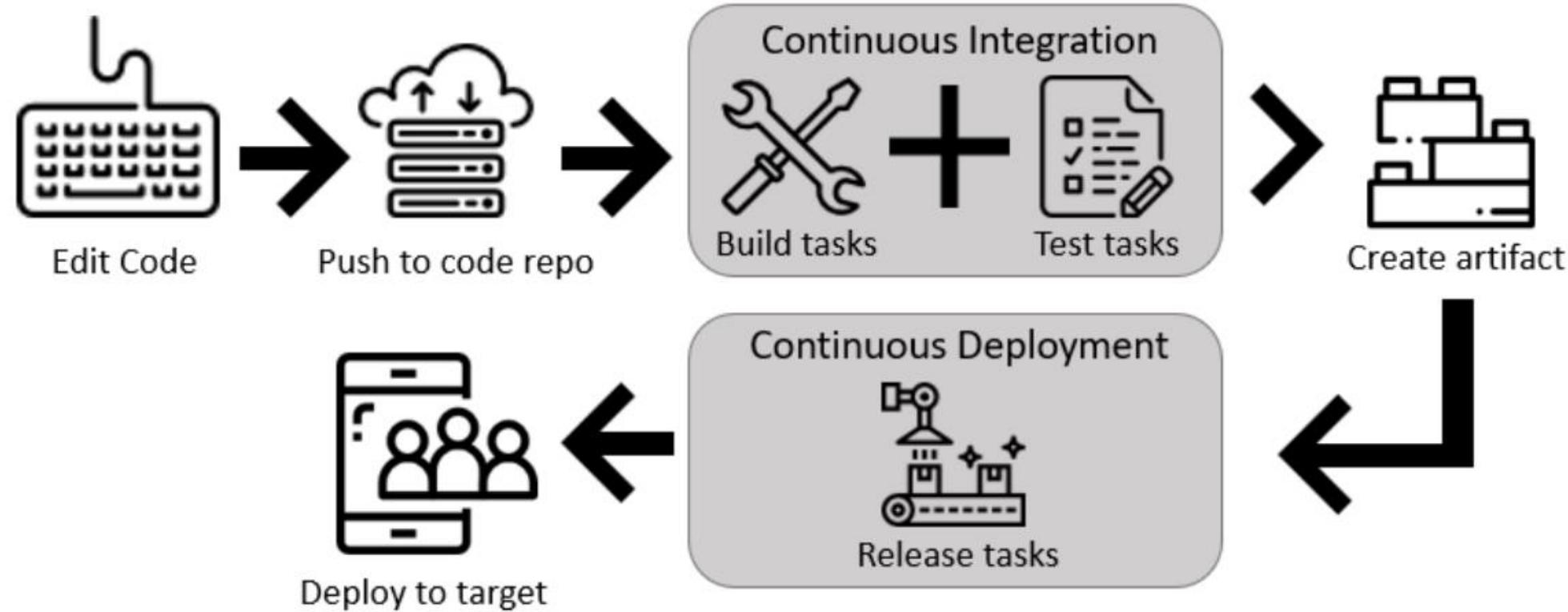
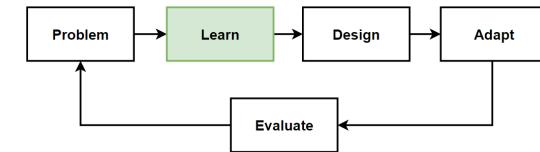
Before Design – What we have in our design toolbox ? - New

Architectures	Patterns&Principles	Microservices Deployment	Non-FR	FR
• Microservices Architecture	• The Database-per-Service Pattern, Polygot Persistence, Decompose services by scalability, The Scale Cube	• Docker and Kubernetes Architecture, Helm Charts	• High Scalability	• List products
• Event-Driven Microservices Architecture	• Microservices Decomposition Pattern • Microservices Communications Patterns • Microservices Data Management Patterns • Event-Driven Architecture • Microservices Distributed Caching • Microservices Deployments with Containers and Orchestrators	• Kubernetes Patterns; Sidecar Patterns, Service Mesh Pattern • DevOps and CI/CD Pipelines • Deployment Strategies; Blue-green, Rolling, Canary and A/B Deployment. • Infrastructure as code (IaC)	• High Availability • Millions of Concurrent User • Independent Deployable • Technology agnostic • Data isolation • Resilience and Fault isolation	• Filter products as per brand and categories • Put products into the shopping cart • Apply coupon for discounts • Checkout the shopping cart and create an order • List my old orders and order items history

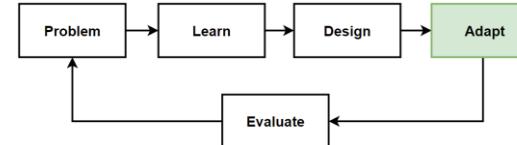
Microservices using Containers and Orchestrators



CI/CD Pipeline Steps for Microservices Deployments



Adapt: Microservices w/ Containers and Orchestrators



Frontend SPAs

- Angular
- Vue
- React

API Gateways

- Kong Gateway
- Express Gateway
- Amazon AWS API Gateway

Backend Microservices

- Java – Spring Boot
- .Net – Asp.net
- JS – NodeJS

Container Tools

- Docker
- Containerd
- Podman

Orchestrator Tools

- Kubernetes
- Helm Charts

Cloud Orchestrator

- AKS, EKS, GKS
- AWS Elastic Beanstalk
- Azure Container Apps

Service Mesh

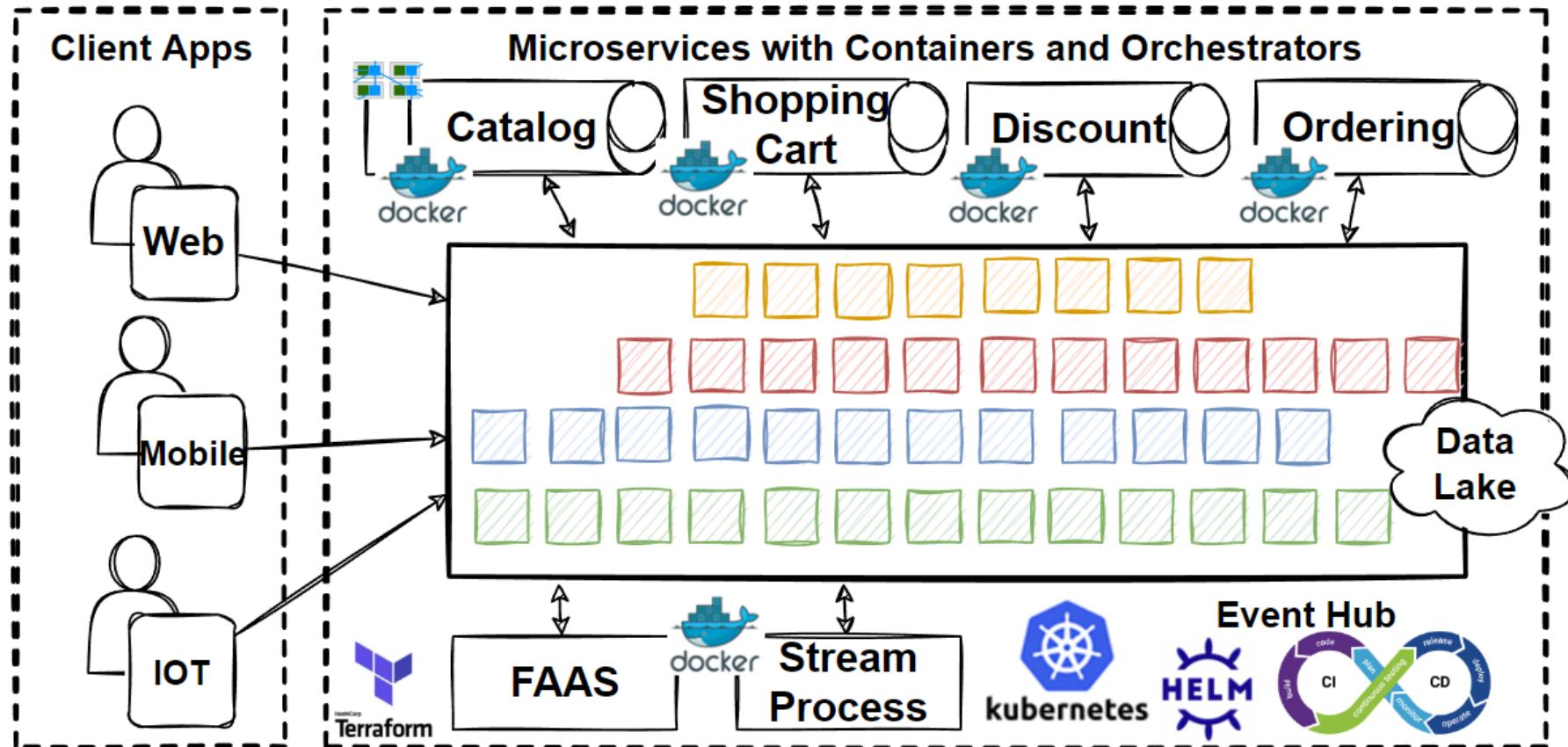
- Istio
- Linkerd

IaC

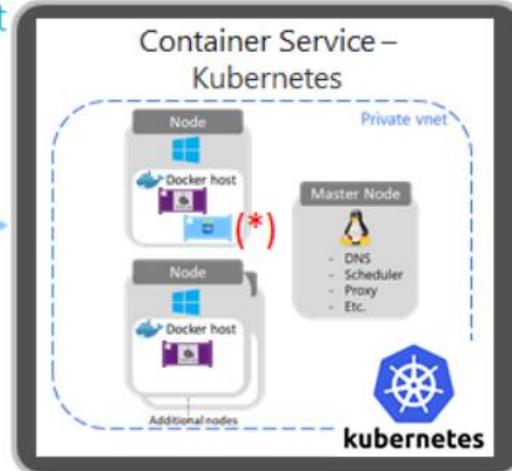
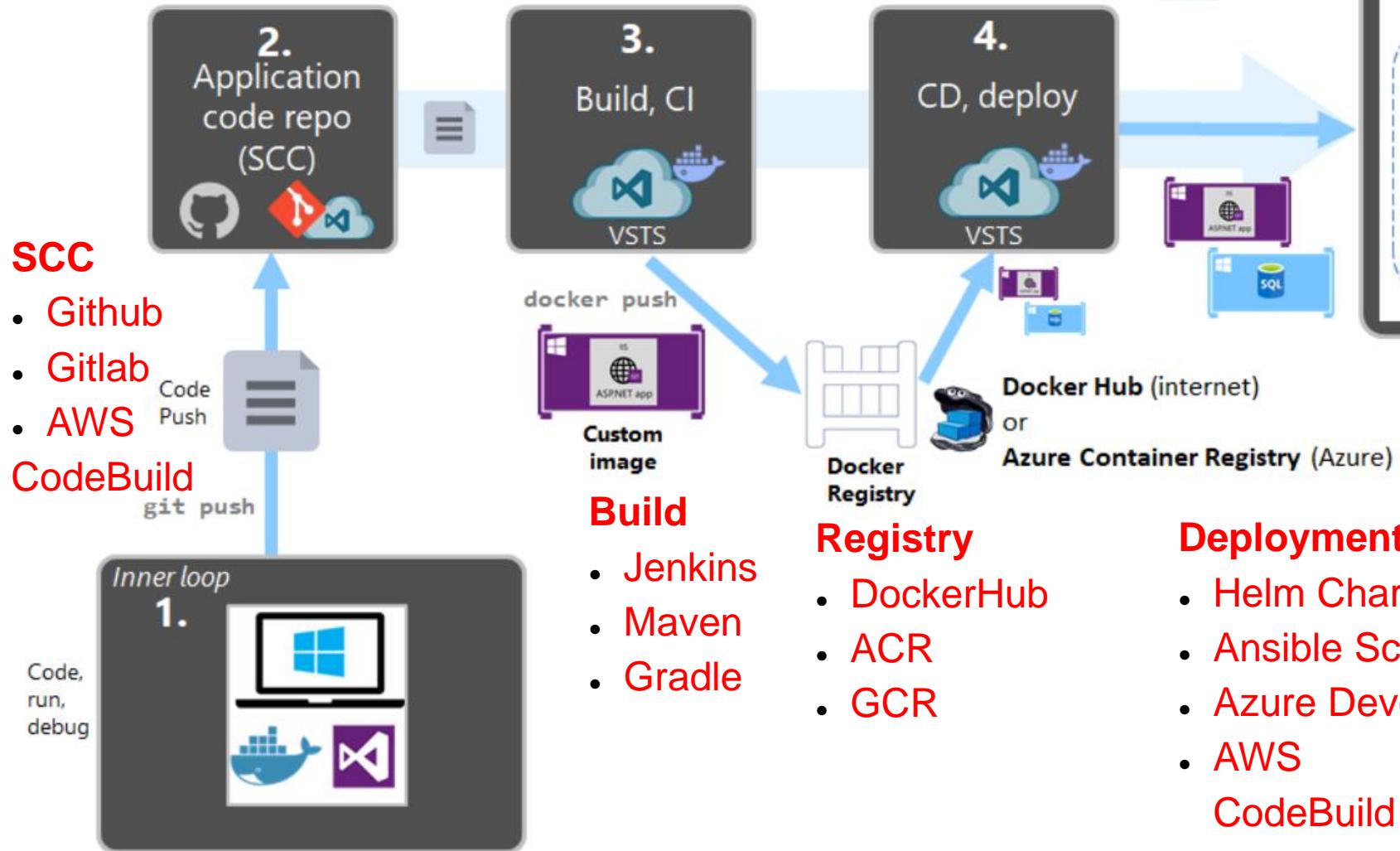
- Terraform
- Ansible

Cloud IaC

- AWS
- CDK



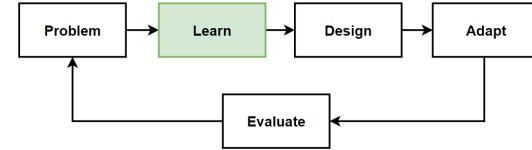
CI/CD Tools for Microservices Deployments



- IaC**
- Terraform
 - Ansible
 - AWS CDK

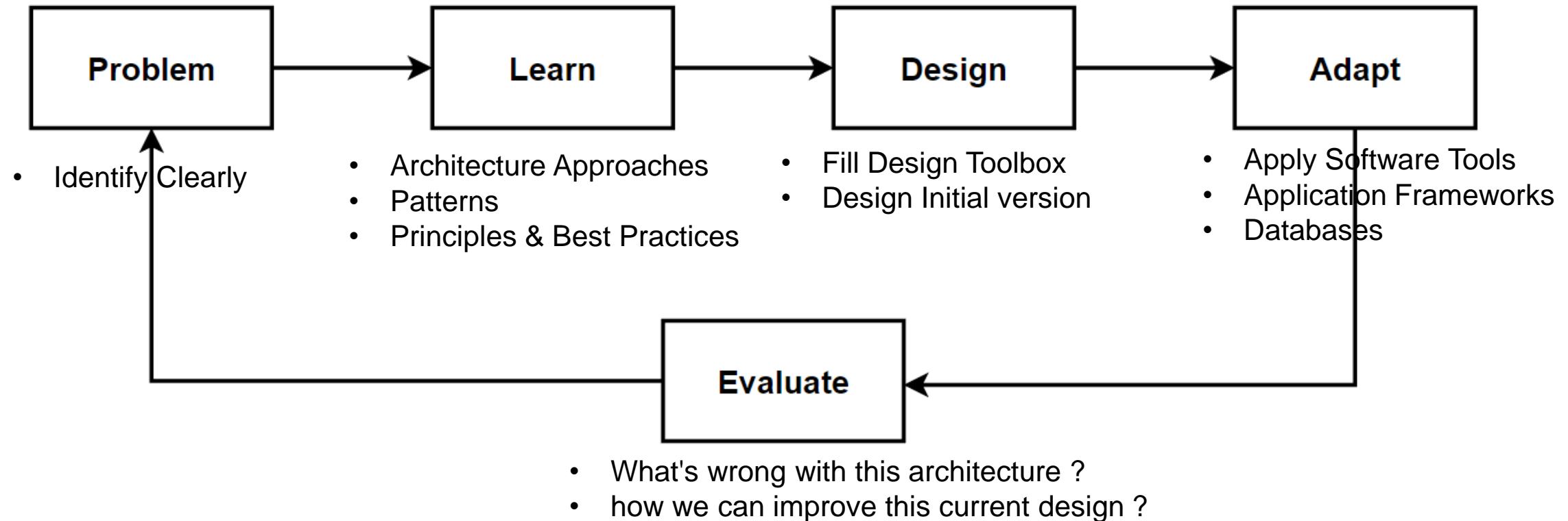
- Monitor**
- Prometheus
 - Datadog
 - Amazon CloudWatch
 - Azure Monitor

How many concurrent request can accommodate our design ?

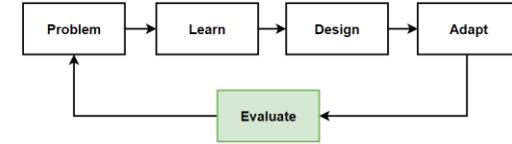


Concurrent Users	Requests/second	Latency (Expected)
2K	0.5K	
20K	12K	
100K	80K	
500K	300K	<= 2 sec

Way of Learning – The Course Flow



Evaluate: Microservices Distributed Caching

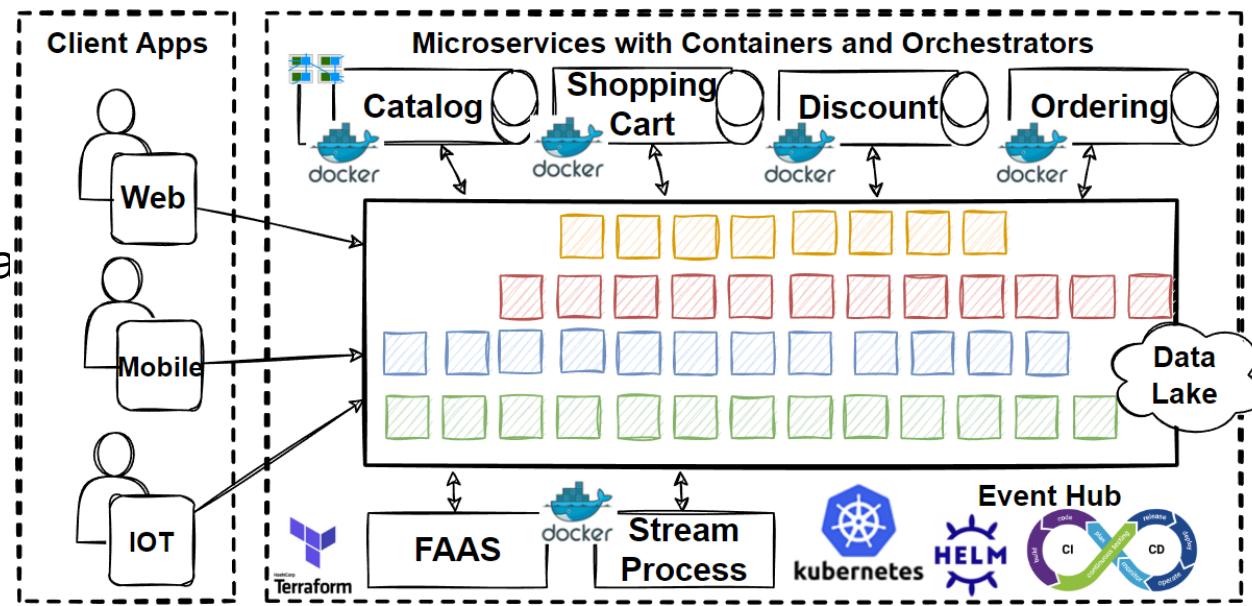


Benefits

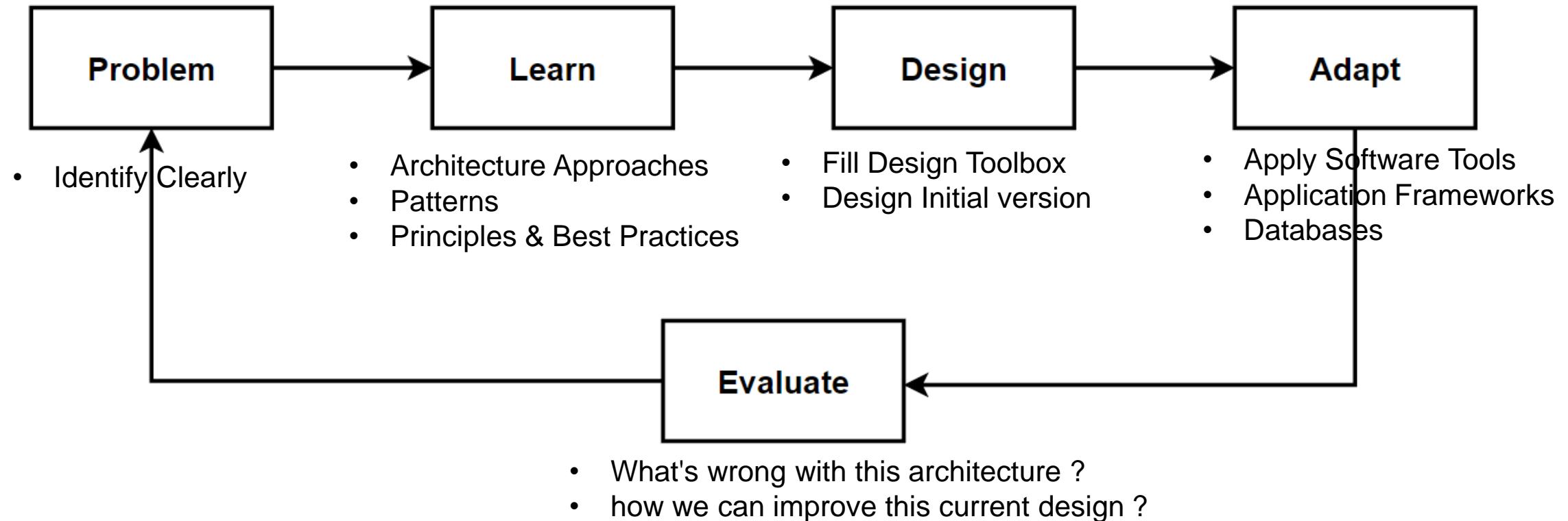
- **Portability**; Containers package applications and their dependencies into a single unit
- **Isolation**; Containers provide isolated environments for each microservice
- **Scalability**; Kubernetes can automatically scale containerized microservices up or down on demand
- **Observability**; platforms provide extensive monitoring and logging capabilities.

Drawbacks

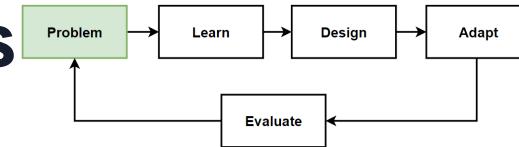
- **Complexity**; complex to set up and manage.
- **Resource overhead**; require a significant amount of resources to run CPU, memory.
- **Performance overhead**; can add some overhead to the performance
- **Dependency on external tools**; like Docker and Kubernetes



Way of Learning – The Course Flow



Problem: Fault tolerance Microservices able to remains operational for any failures or disruptions



Problems

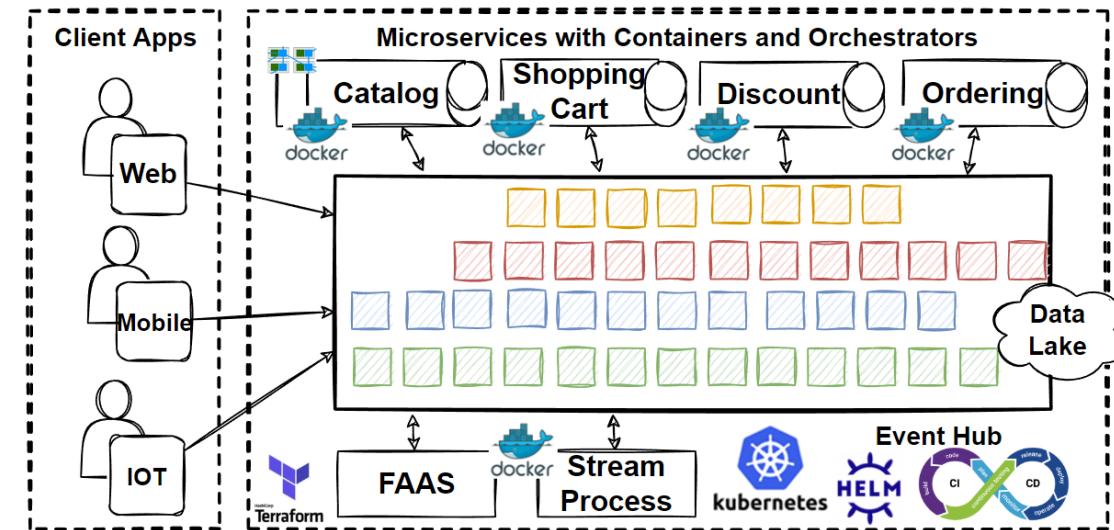
- Business teams able to deploy new features immediately to compete the market
- Due to heavy traffic, microservices got exceptions and broke some major use cases that cause lost revenue
- Create big crisis to customer loyalty

Considerations

- Deploy new features immediately without affecting the system general behaviors
- System should recover from failures and provide ability of a system to withstands.
- Architecture should be designed to be resilient

Solutions

- Microservices Resilience and Fault Tolerance
- Microservices Observability with Distributed Logging and Distributed Tracing
- Microservices Health Monitoring



Microservices Resilience, Observability and Monitoring

Microservices Resilience and Fault Tolerance

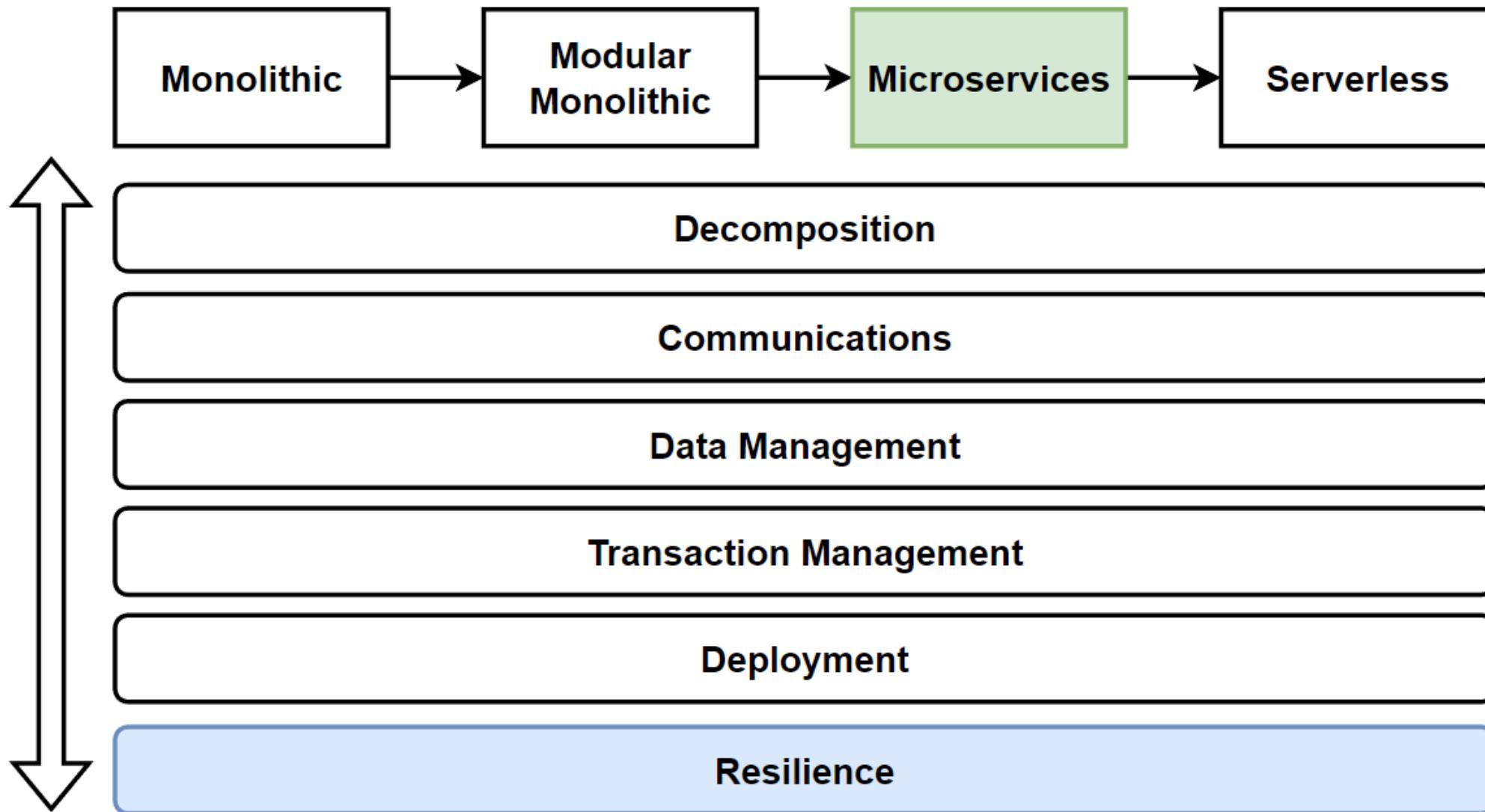
Retry, Circuit-Breaker, Bulkhead Pattern

Microservices Observability with Distributed Logging and Distributed Tracing

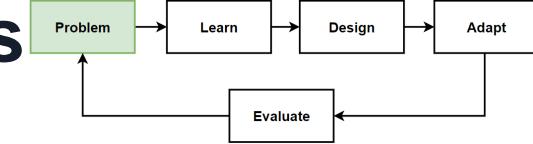
Elastic Stack which includes Elasticsearch + Logstash + Kibana

Microservices Health Monitoring

Architecture Design – Vertical Considerations



Problem: Fault tolerance Microservices able to remains operational for any failures or disruptions



Problems

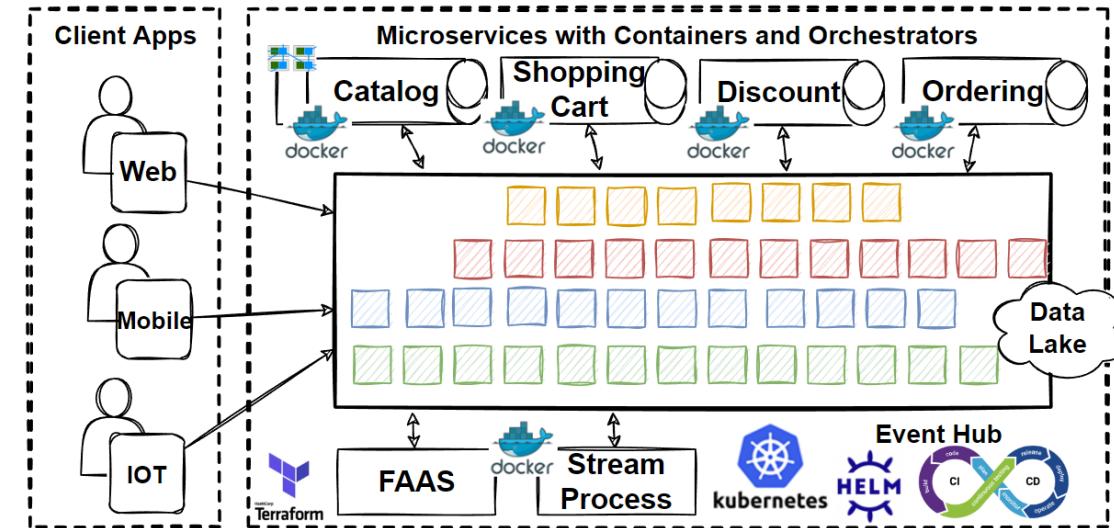
- Business teams able to deploy new features immediately to compete the market
- Due to heavy traffic, microservices got exceptions and broke some major use cases that cause lost revenue
- Create big crisis to customer loyalty

Considerations

- Deploy new features immediately without affecting the system general behaviors
- System should recover from failures and provide ability of a system to withstands.
- Architecture should be designed to be resilient

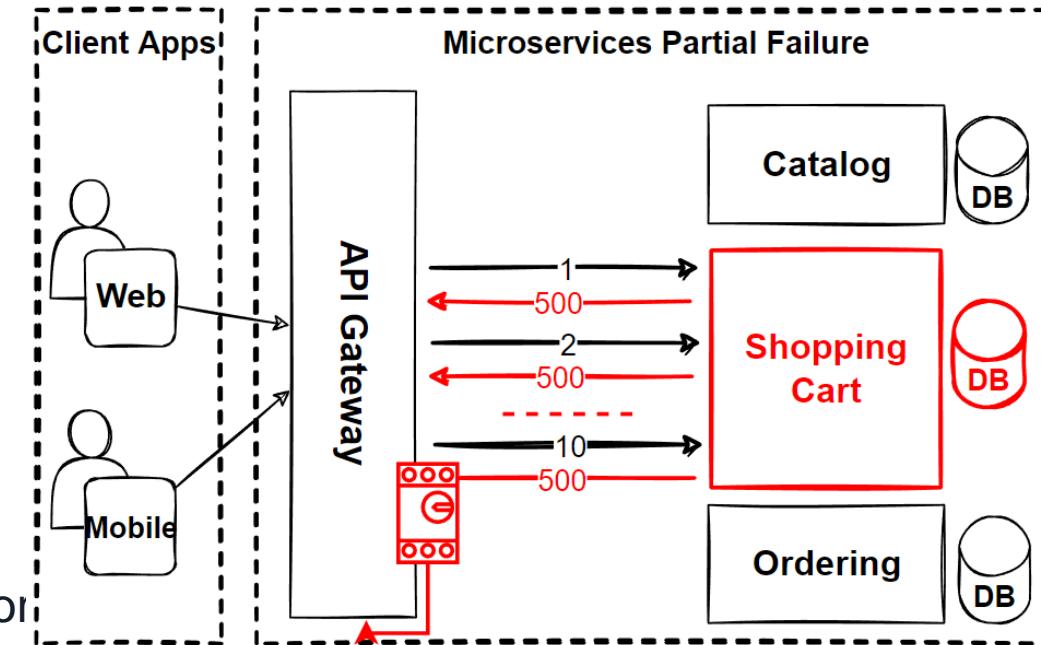
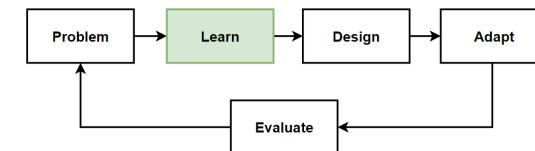
Solutions

- Microservices Resilience and Fault Tolerance
- Microservices Observability with Distributed Logging and Distributed Tracing
- Microservices Health Monitoring



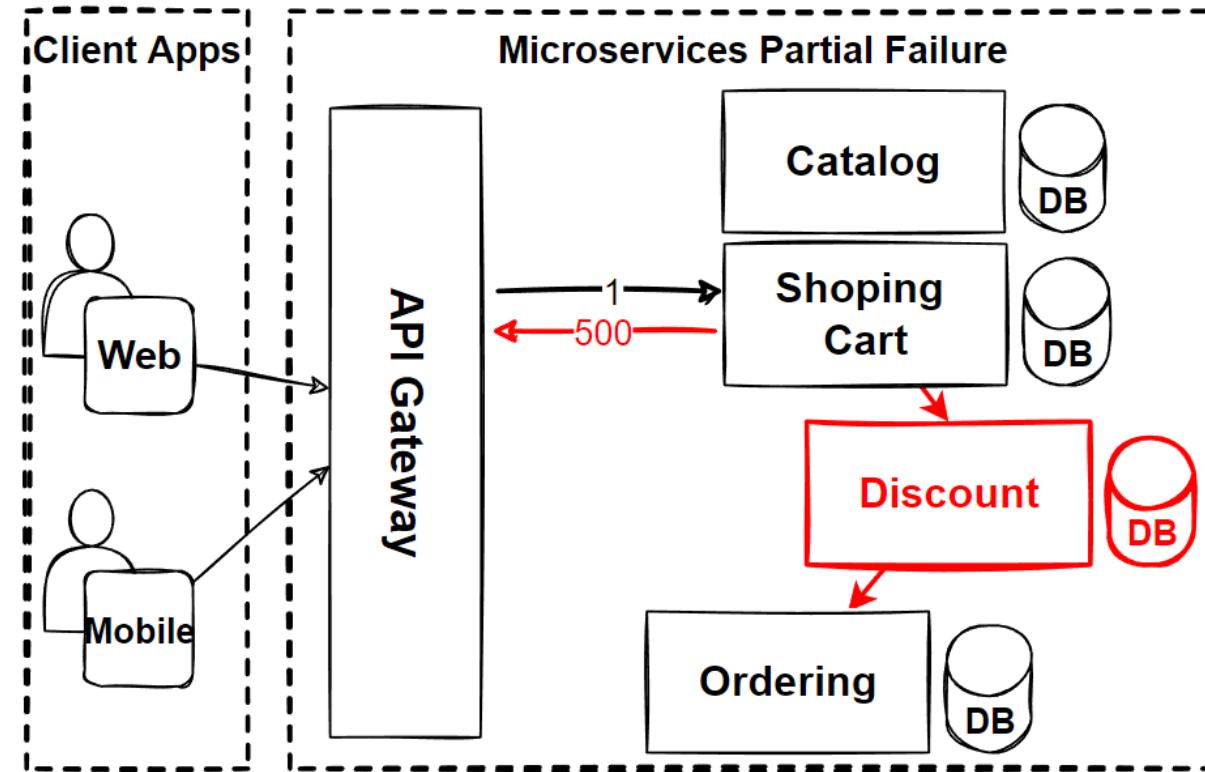
What is Microservices Resiliency ?

- Microservice architecture have become the new model for building modern cloud-native applications.
- But it is increasing interactions between microservices have created a new set of problems.
- Should assume that **failures will happen**, and **dealing with unexpected failures**, especially in a distributed system.
- Network or container failures, microservices must have a **strategy to retry** requests again.
- App is going to **fail at some point** and we need to **embrace failures**.
- **Design microservices with failures** that need to be prepared for the worst case scenario.
- **What happens when the machine where the microservice is running fails ?**



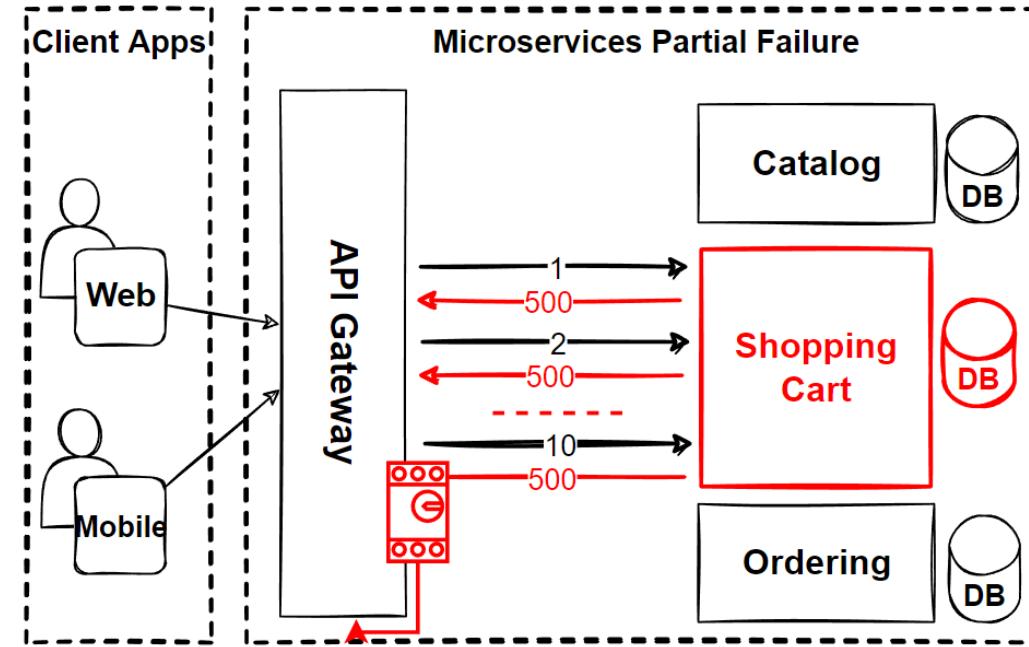
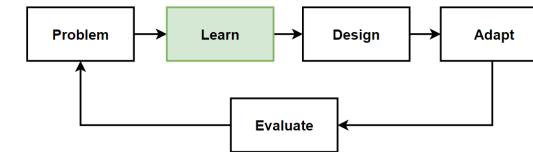
Cascade Failure in Microservices

- **What happens** when the machine where the microservice is running fails ?
- **Single microservice can fail** or might not be available to respond for a short time.
- Since **clients and services are separate processes**, a service might **not be able to respond in time** for client's request.
- The **service might be overloaded and responding very slowly** to requests or might **not be accessible** for a short time because of **network issues**.
- If Service A calls Service B, which in turn calls Service C, what happens when Service B is down ?
- What is your **fallback plan** in such a scenario ?
- How do you **handle the complexity** that comes with **cloud and microservices** ?



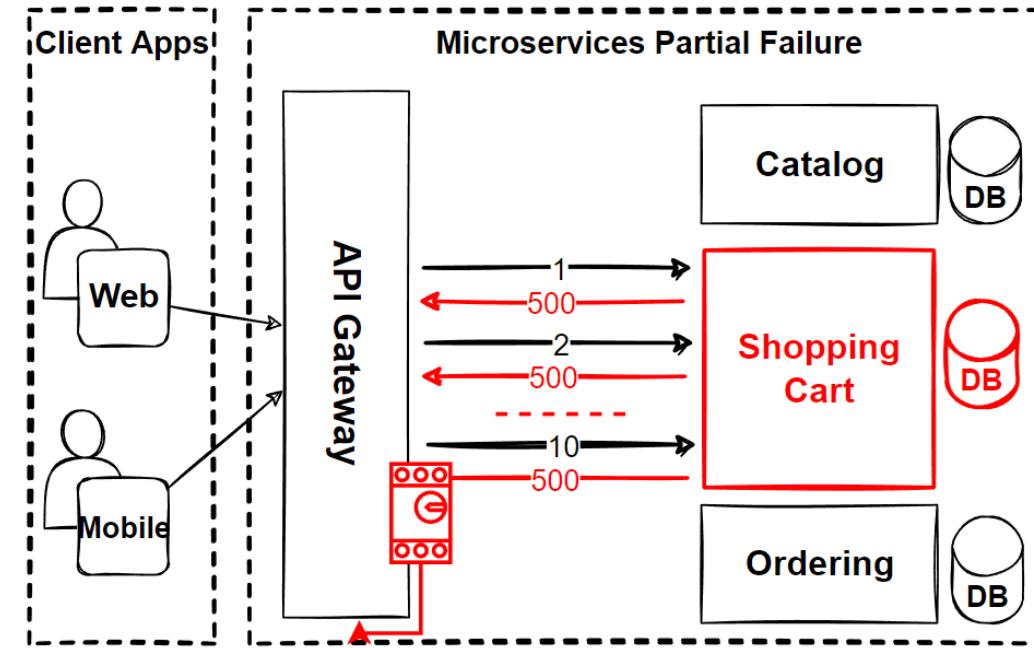
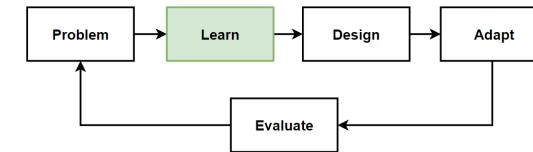
What is Microservices Resiliency ?

- **Microservice** should **design for resiliency**, needs to be **resilient to failures** and must **accept partial failures**.
- Design **microservices** to be **resilient for partial failures**, ability to **recover from failures** and **continue to function**.
- **Accepting failures** and responding to them without any downtime or data loss.
- **Return the application** to a **fully functioning state** after a failure.
- Assume that **failures will happen** and **design our microservices for resiliency**.
- **Microservices** are going to fail at some point, that's why we need to learn **embracing failures**.
- **Microservices** system is considered to be **resilient**, if it can continue to function effectively **despite failures or disruptions**.
- Should be **fault tolerant** and **handle failures gracefully**.

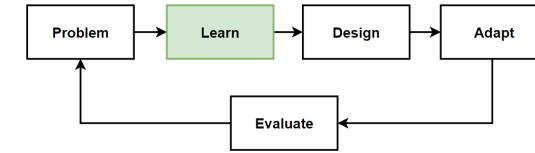


Microservices Resiliency Patterns

- To provide **unbroken microservice**, the **architecture** must be **designed correctly**.
- We can apply to provide **uninterrupted microservice**, with **“Resilience Patterns”**.
- **Resilience Patterns** can be **divided into different categories** according to the problem area they solve.
- Ensuring the durability of services in microservice architecture is relatively difficult compared to a monolithic architecture.
- The communication between services is **distributed** and many **internal or external network traffic** is created for a transaction.
- Communication need between services and **dependence on external services increases**.
- The possibility of **occurring errors will increase**.
- There are **several patterns** that can be used to **improve the resiliency** of a microservices system.



Microservices Resiliency Patterns - 2



- **Retry Pattern**

Retrying a request if it fails or times out. Retries can be implemented at the client or the service level, to handle temporary failures or disruptions.

- **Circuit Breaker Pattern**

Introducing a proxy or "circuit breaker" between a client and a service. It will open the circuit and prevent further requests from being sent to the service.

- **Bulkhead Pattern**

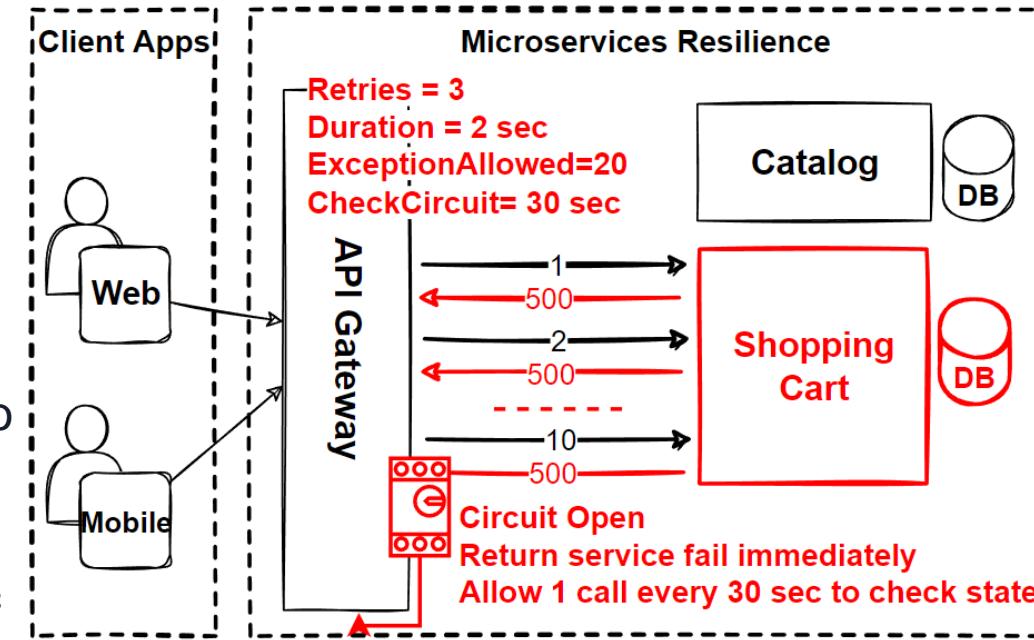
Partitioning a system into isolated components, or "bulkheads," to prevent the failure of one component from affecting the others.

- **Timeout Pattern**

Should not wait for a service response for an indefinite amount of time, throw an exception instead of waiting too long.

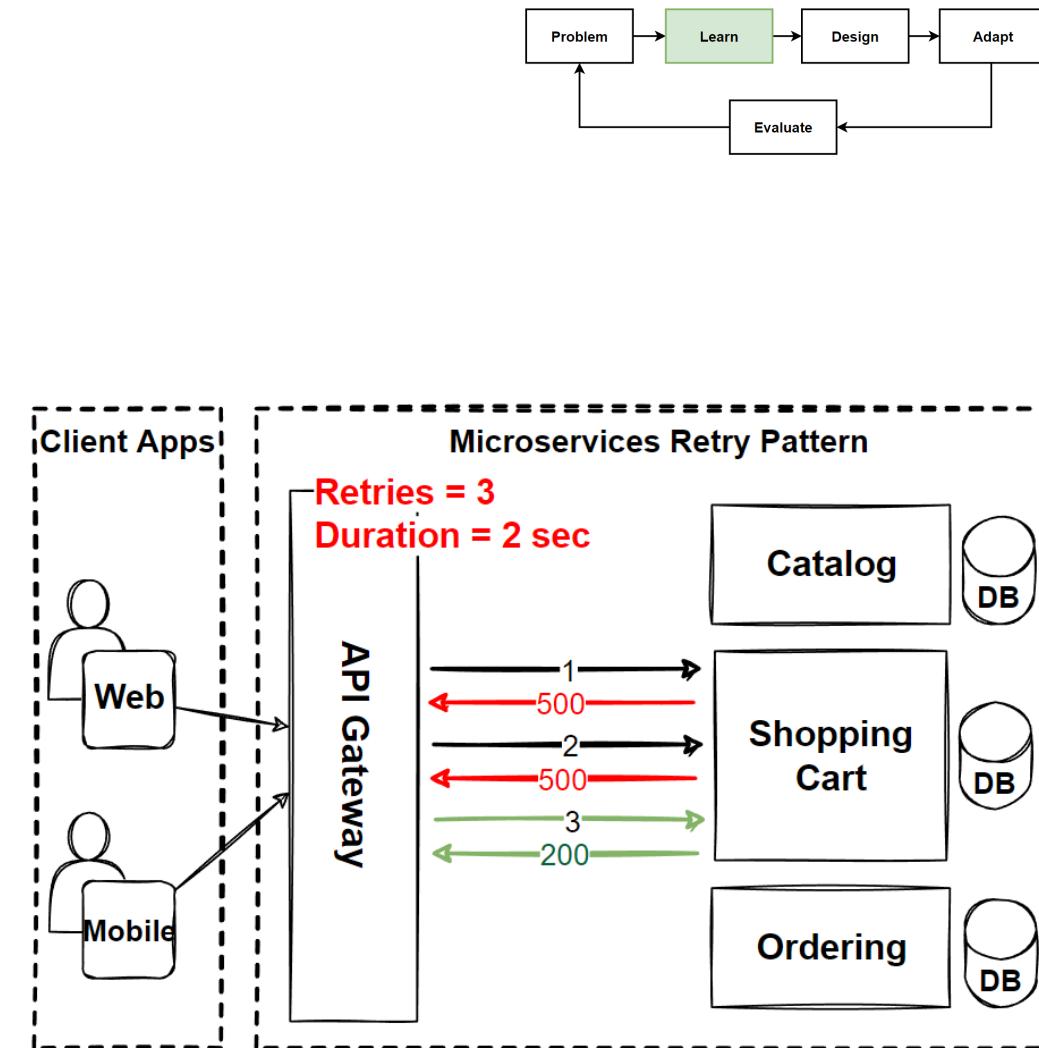
- **Fallback Pattern**

Providing an alternative behavior or response if a request fails or times out.



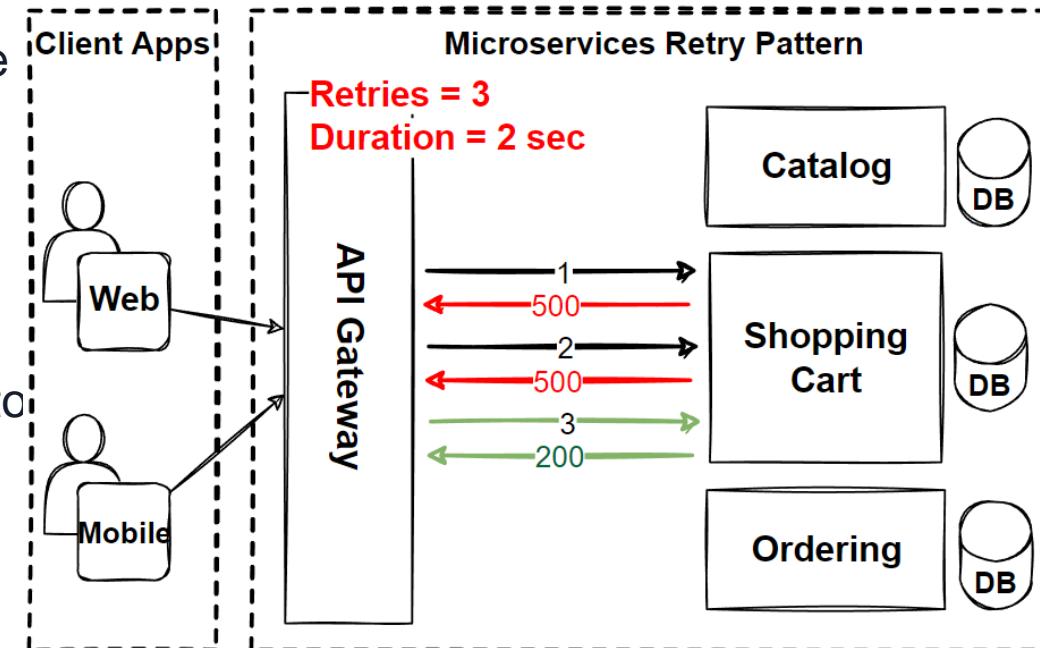
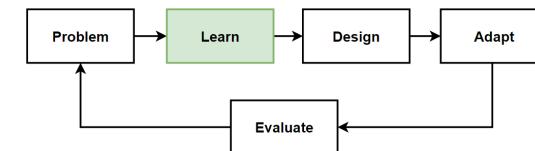
Retry Pattern

- **Inter-service communication** can be performed by **HTTP** or **gRPC** and can be managed by developments.
- When it comes to **network, server and such physical interruptions** may be **unavoidable**.
- If one of the services **returns HTTP 500** during the transaction, the **transaction interrupted** and an **error** may be returned.
- But when the user **restarts the same transaction**, it may work.
- **More logical** to repeat the request made to the **500** returned service, user will be able to **perform the transaction successfully**.
- **Microservices communications** can fail because of **transient failures**, but this **failures happen in a short-time** and **can fix after that time**.
- For that cases, we should implement **Retry Pattern**.

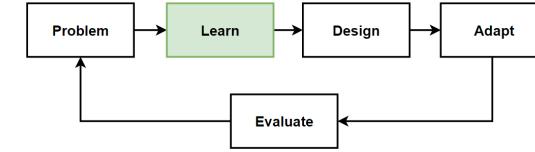


Retry Pattern - 2

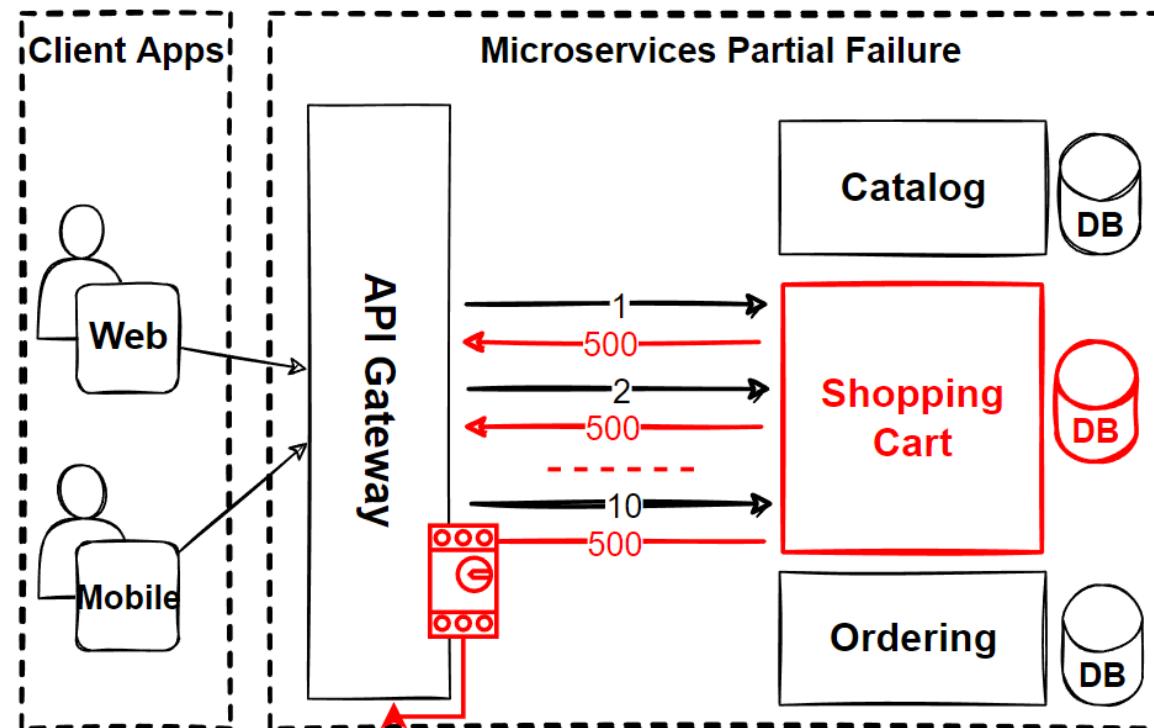
- **Retry Pattern** allows to **automatically retry** an operation **if it fails**.
- Ensure that a service is **able to handle failures or temporary unavailability** of dependent services.
- Common use of the **Retry pattern in microservices** is to handle **temporary failures** when **calling a downstream service**.
- I.e. **shopping cart service** that **relies on a payment service** to process payments.
- If the **payment service** is **temporarily unavailable** due to a **network issue**, the shopping cart service use the **retry pattern** to **automatically retry the payment request**.
- Allow the microservice time to **fix itself** with **self-correct**, and **extend the back-off time** before **retrying the call**.
- **The back-off period should be exponentially incremental withdrawal** to allow sufficient correction time.



Drawbacks of Retry Pattern

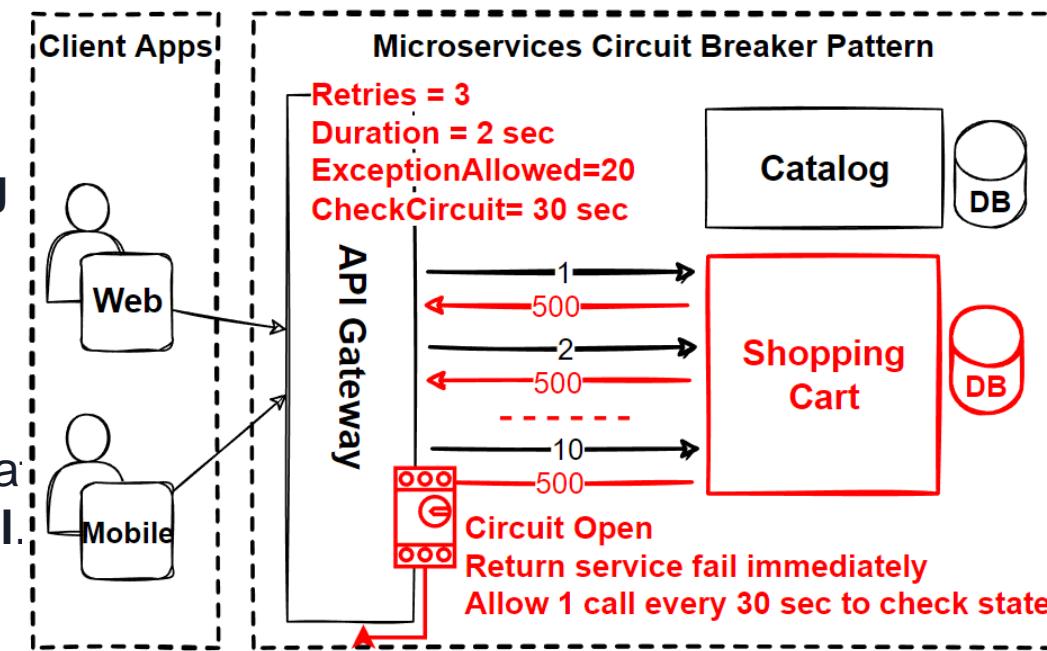
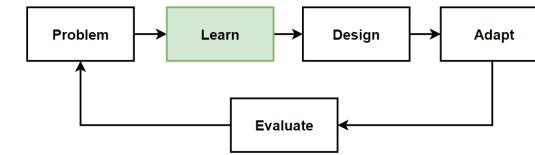


- The **Retry pattern** should be used **with caution**.
- If a service is **experiencing persistent failures** or is **unavailable for an extended period of time**, the retry pattern result in an **excessive number of failed requests**.
- In these cases, it is **necessary to implement additional strategies, circuit breaking or fallback logic** to prevent the retry pattern from further impacting the system.



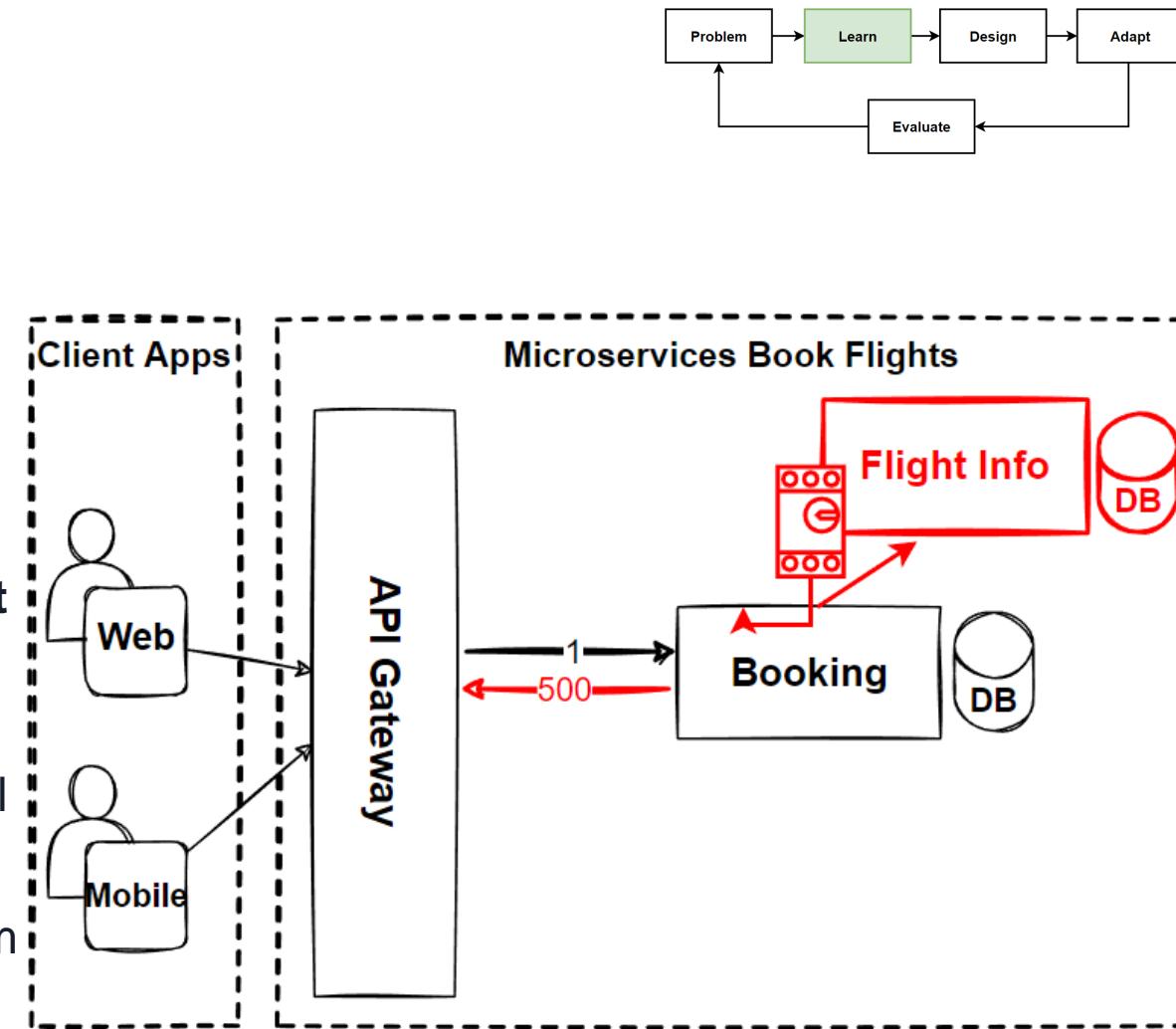
Circuit Breaker Pattern

- Method in **electronic circuits** that is constructed like **circuit breaker switchgear**.
- **Stop the load transfer** in case of a failure in system to **protect** the electronic circuit.
- Protects against failures in **external dependencies**, useful in **microservice** when failure in one service can have **cascading effects** on other services.
- **Circuit breaker pattern** prevents **cascading failures** in a system.
- If one **microservice depends on another microservice** and that second microservice fails, the **first microservice might also fail**.
- If the **first microservice** is using a **Circuit breaker pattern**, it can **prevent further calls** to the second microservice and continue to function.



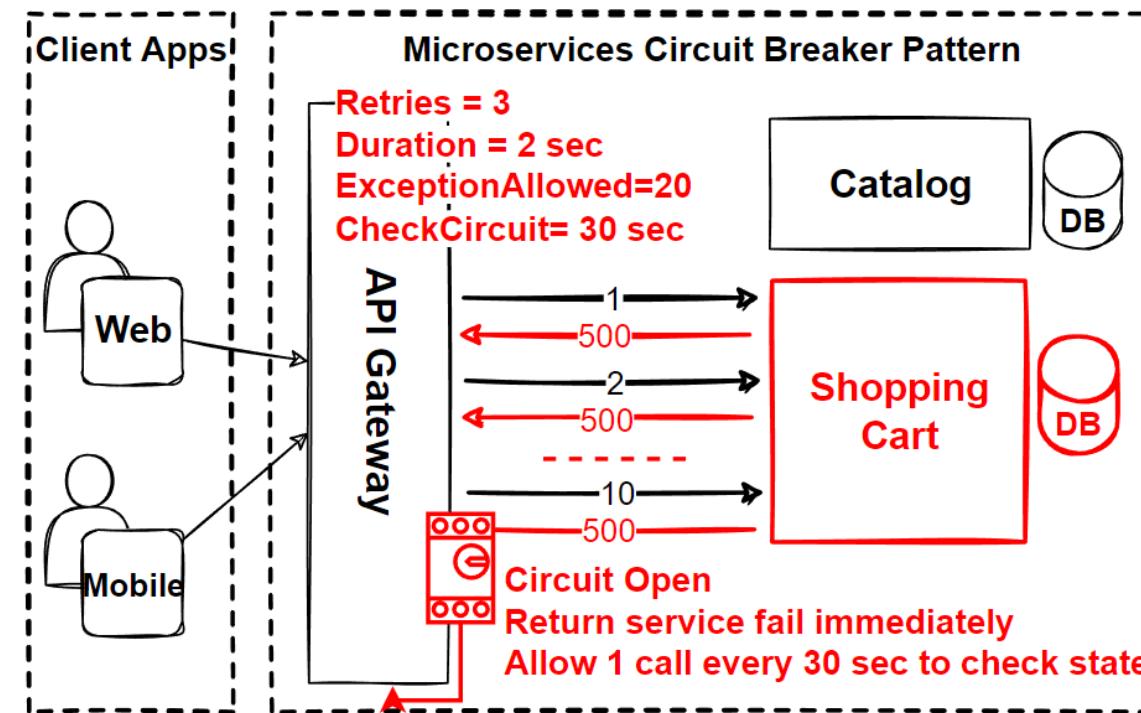
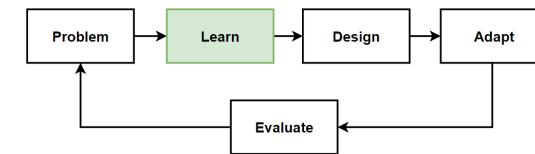
Book Flights Example

- Microservices that allows users to **book flights**.
- This microservice **depends on an external service** for **retrieving flight information**.
- If the **external service becomes unavailable**, the **flight booking** microservice will **also become unavailable**.
- To protect against this type of failure, **implement a circuit breaker pattern** around the call to the external service.
- If the **external service becomes unavailable**, the **circuit breaker will trip**, preventing further calls to the external service.
- This will help to **prevent cascading failures** in the system



Circuit Breaker Pattern - 2

- **Circuit Breakers** pattern **monitors** the communication between the services and **follows the errors** that occur in the communication.
- When the error in the **system exceeds a certain threshold** value, **Circuit Breakers turn on** and **cut off communication**, and **returning previously determined error messages**.
- While **Circuit Breakers is open**, it **continues to monitor** the communication traffic.
- If the requested service **starts to return successful results**, it **becomes closed**.



Circuit Breaker States

- **Closed**

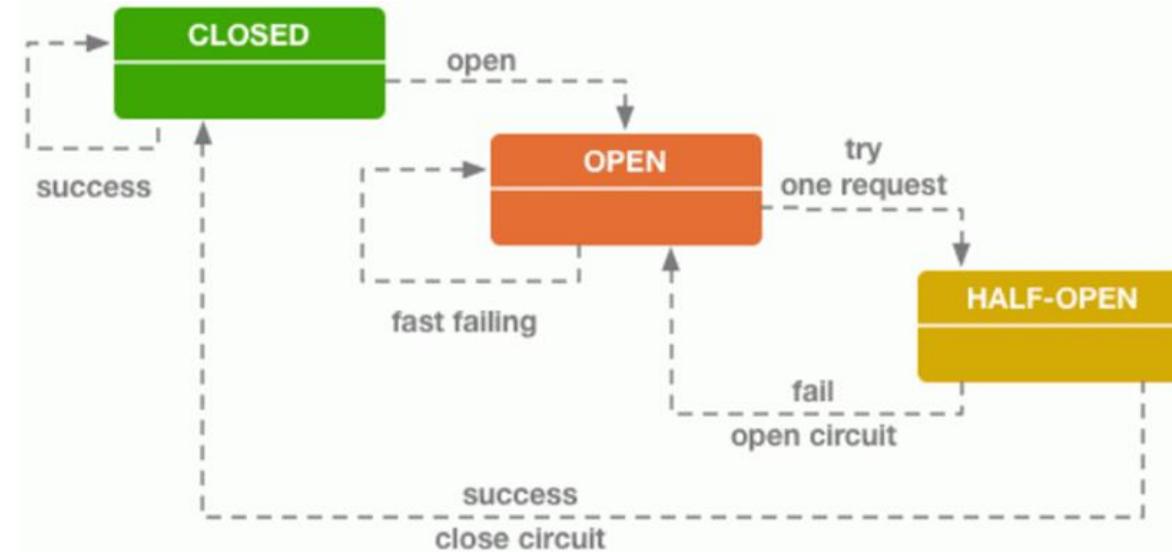
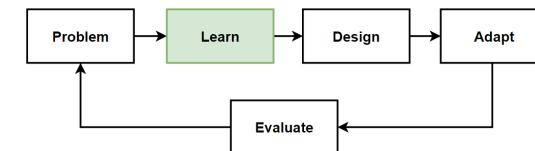
The circuit breaker is not open and all requests are executed.

- **Open**

The Circuit Breaker is open and it prevents the application from repeatedly trying to execute an operation while an error occurs.

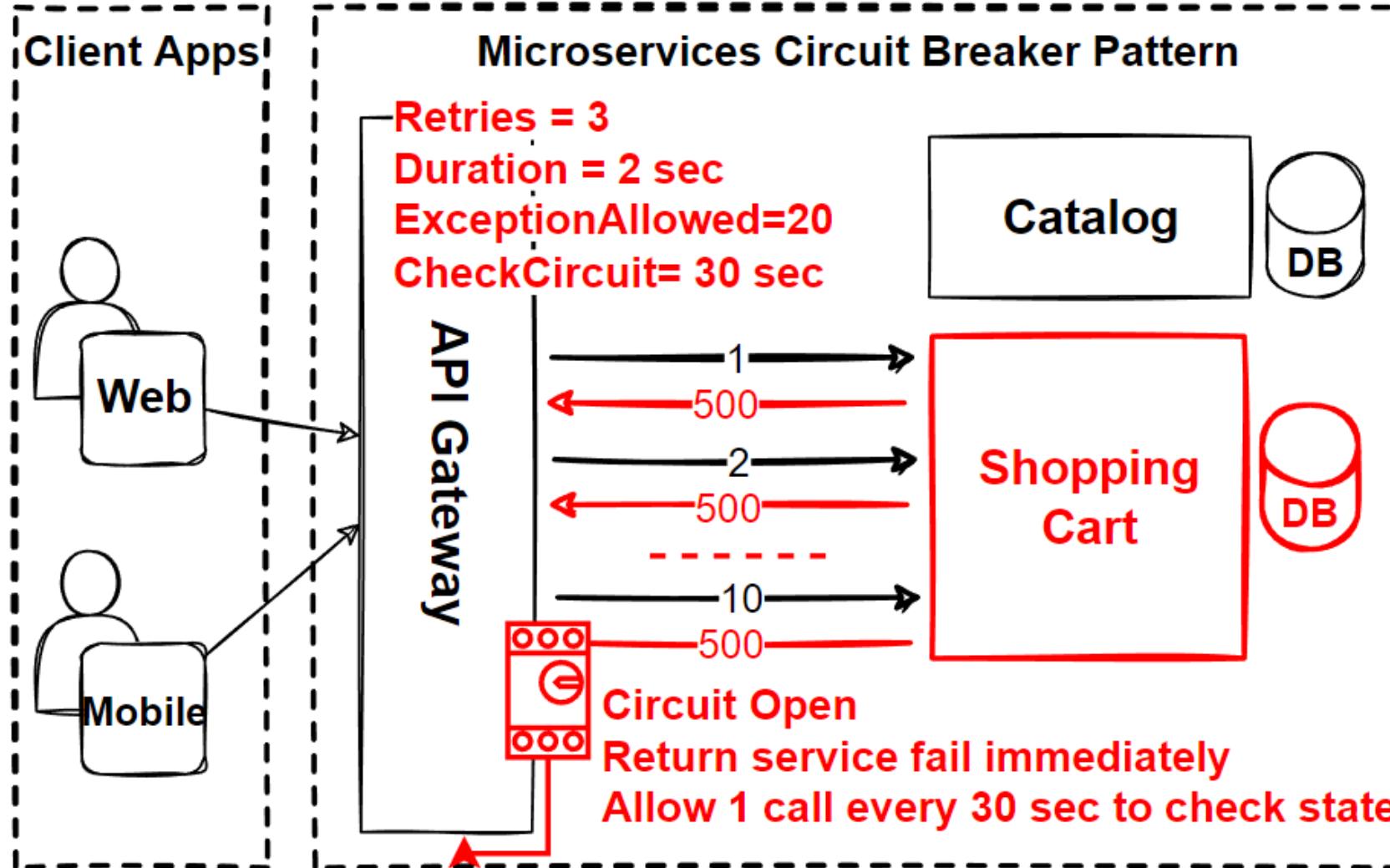
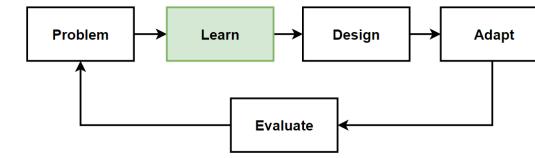
- **Half-Open**

The Circuit Breaker executes a few operations to identify if an error still occurs. If errors occur, then the circuit breaker will be opened, if not it will be closed.

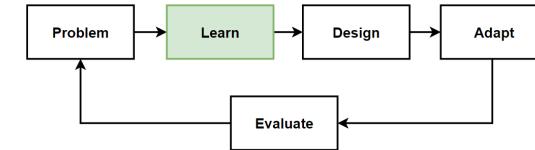


Circuit Breaker State Diagram

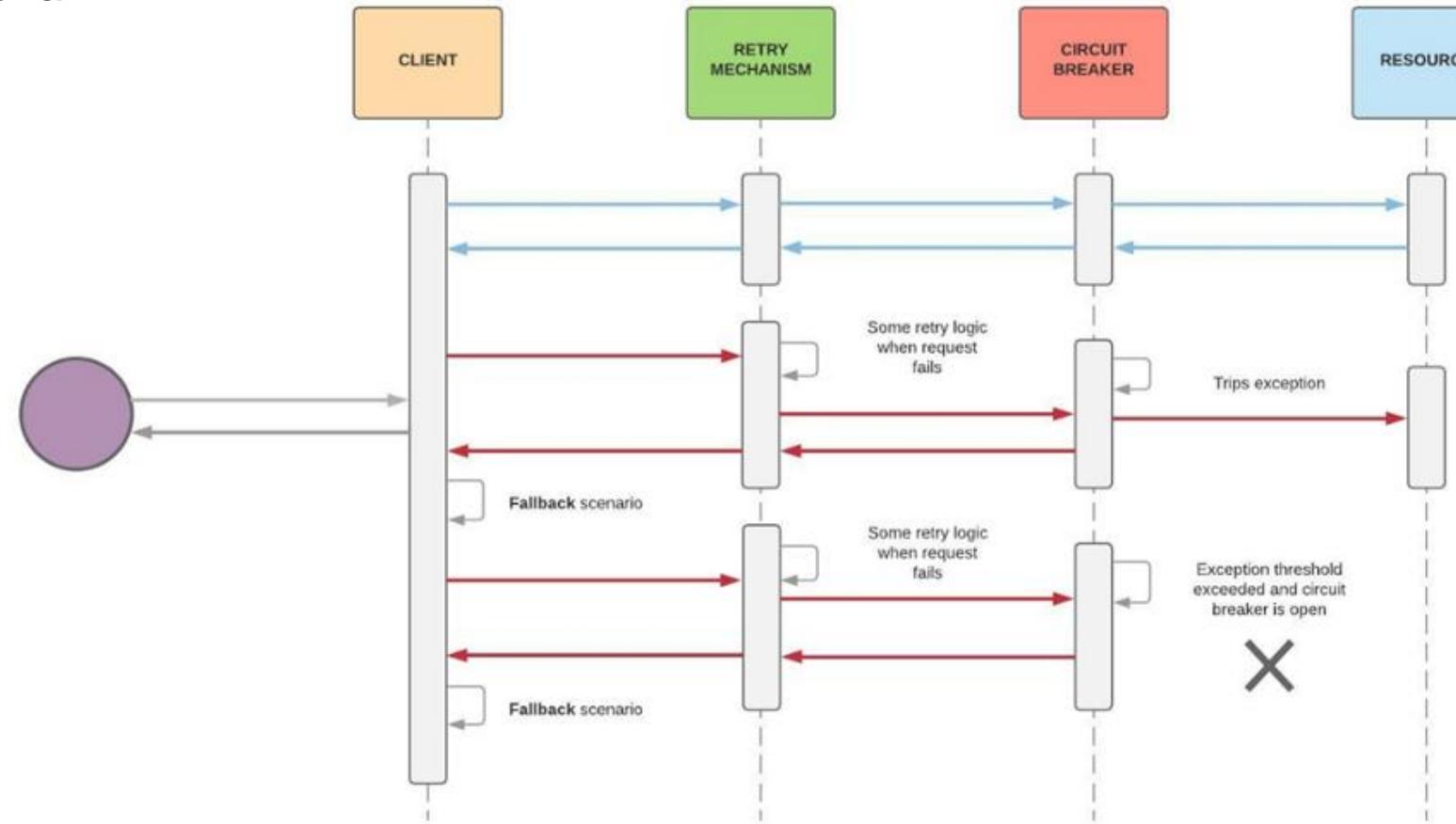
Circuit Breaker Pattern



Retry + Circuit Breaker Pattern

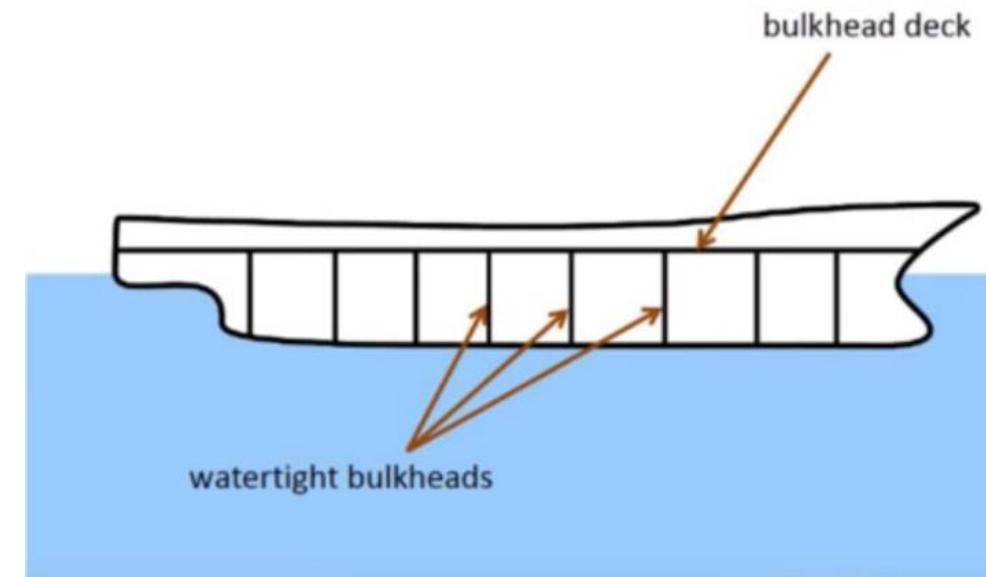
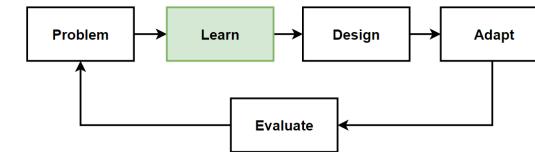


- **Retry pattern** only retry an operation in the expectation calls since it will get succeed.
- **Circuit Breaker pattern** is prevent broken communications from repeatedly trying to send request that is mostly to fail.

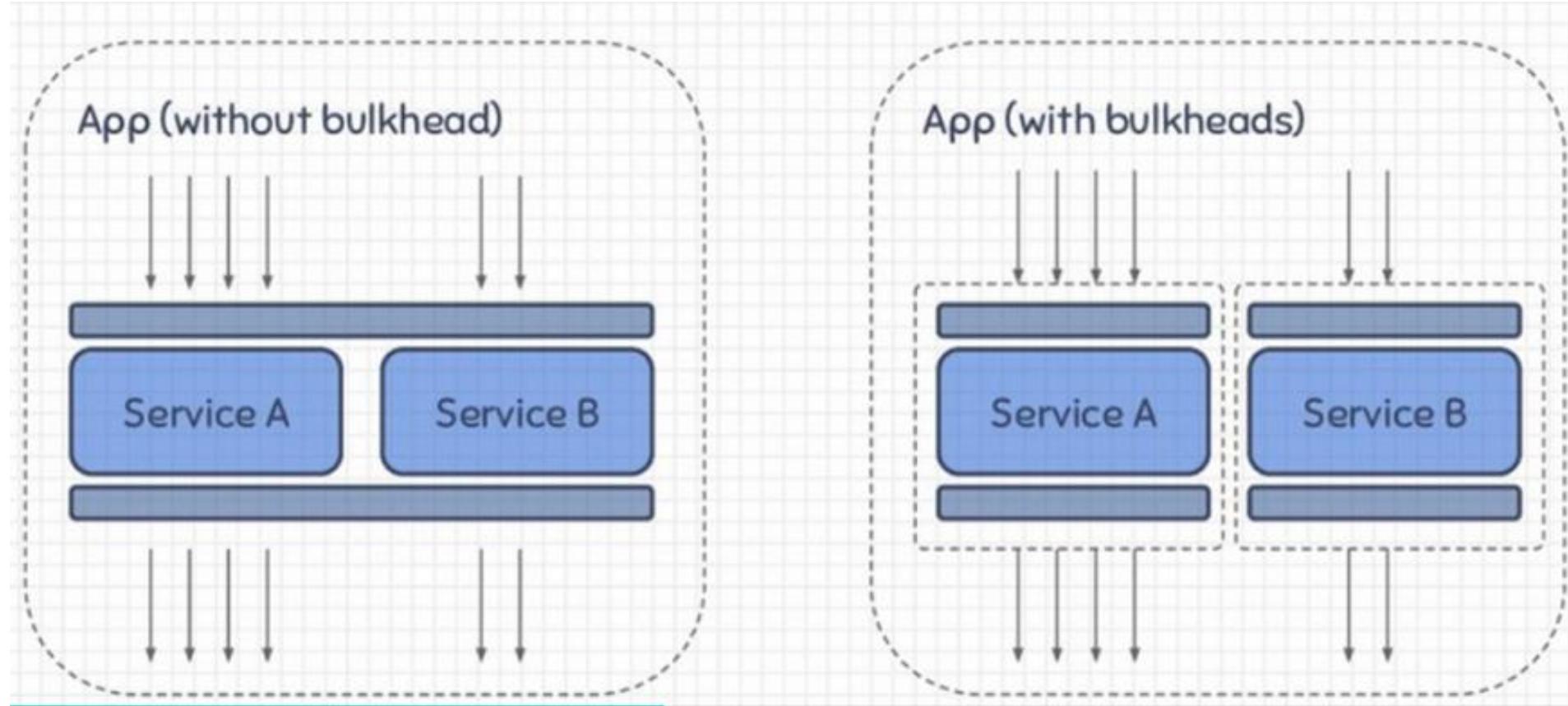
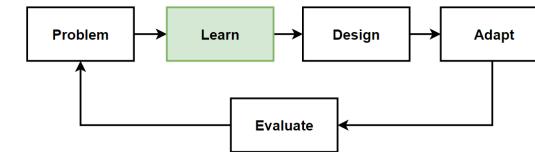


Bulkhead Pattern

- **Bulkhead pattern** is used to **isolate failures** in a **distributed system**.
- Based on the concept of a "bulkhead" in a **ship**, **structural partition** that helps to **prevent the entire ship from flooding** if one **compartment** is **damaged**.
- In **microservices**, **isolate individual service** instances or groups of service instances from each other, to **prevent failures** in one part of the system from **affecting the entire system**.
- **One location does not affect** other services by **isolating** the services. The main purpose is **isolated error place** and **secure** the **rest of services**.
- **Naval architecture**; ships or submarines are made not as a whole, but by shielding from certain areas, if **there is happens** a **flood or fire**, the **relevant compartment** is **closed** and **isolated**.
- **Microservices** able to respond to user requests by **isolating** the **relevant service** so the error.



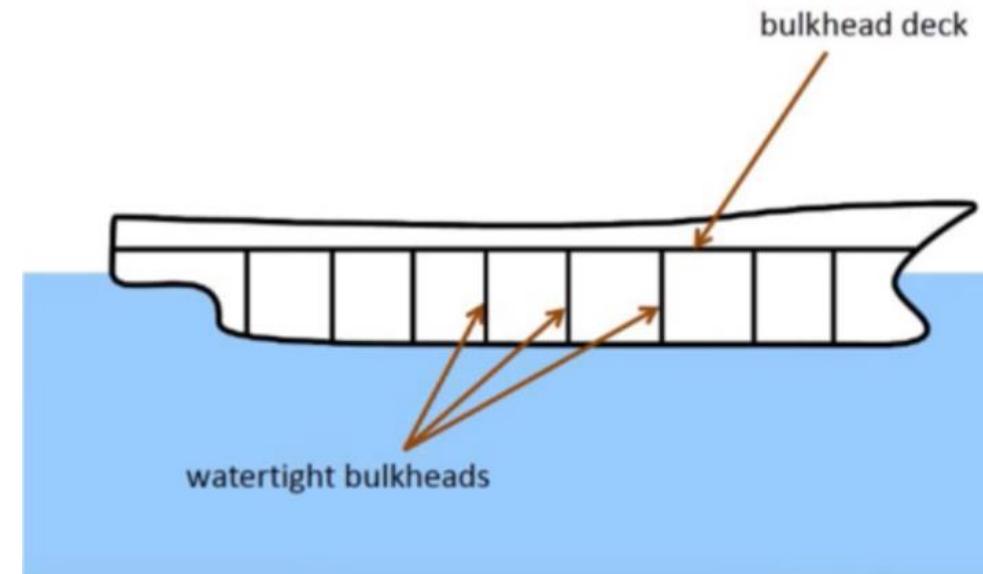
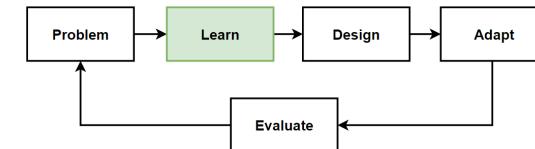
Bulkhead Pattern - 2



<https://speakerdeck.com/slok/resilience-patterns-server-edition?slide=33>

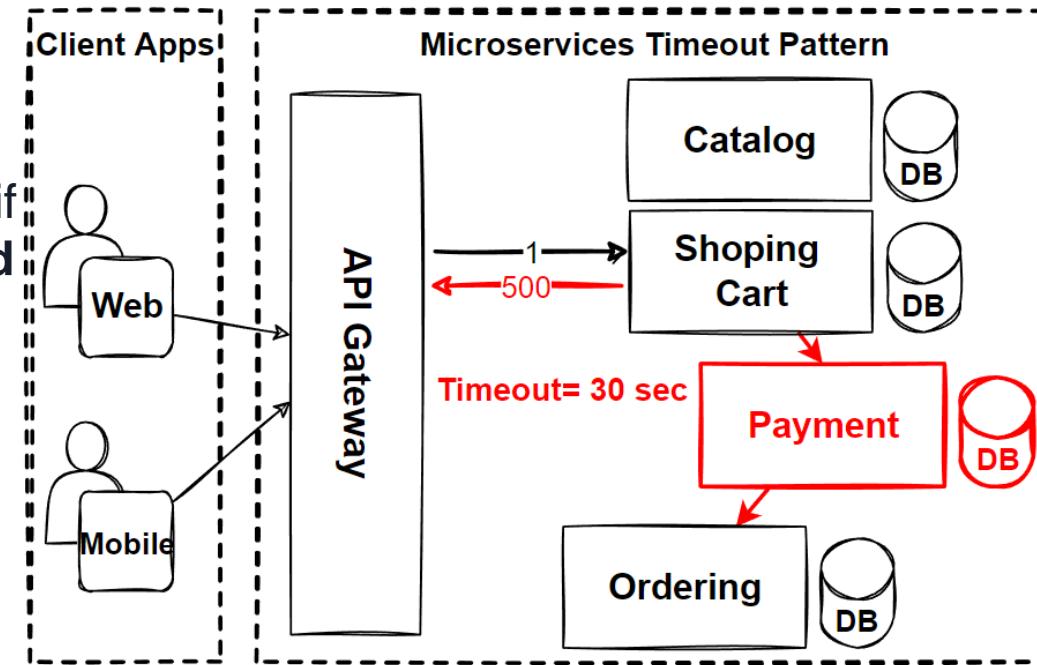
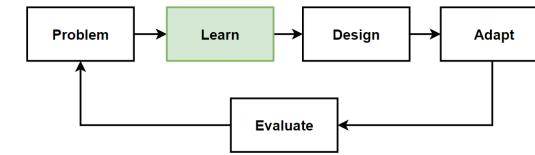
When to use Bulkhead Pattern

- Microservices use **Bulkhead Pattern** to prevent failures in one part of the system from affecting the entire system.
- **Use cases for bulkhead pattern:**
- When a **service has a high load** and there is a **risk of resource contention** with other service instances.
- When a **service depends on one or more downstream services**, and there is a **risk of cascading failures**.
- If a downstream **service becomes unavailable** or experiences a performance issue.
- When a service is **required to make frequent calls** to a downstream service, and there is a **risk of overloading** the downstream service if it **becomes unavailable**.



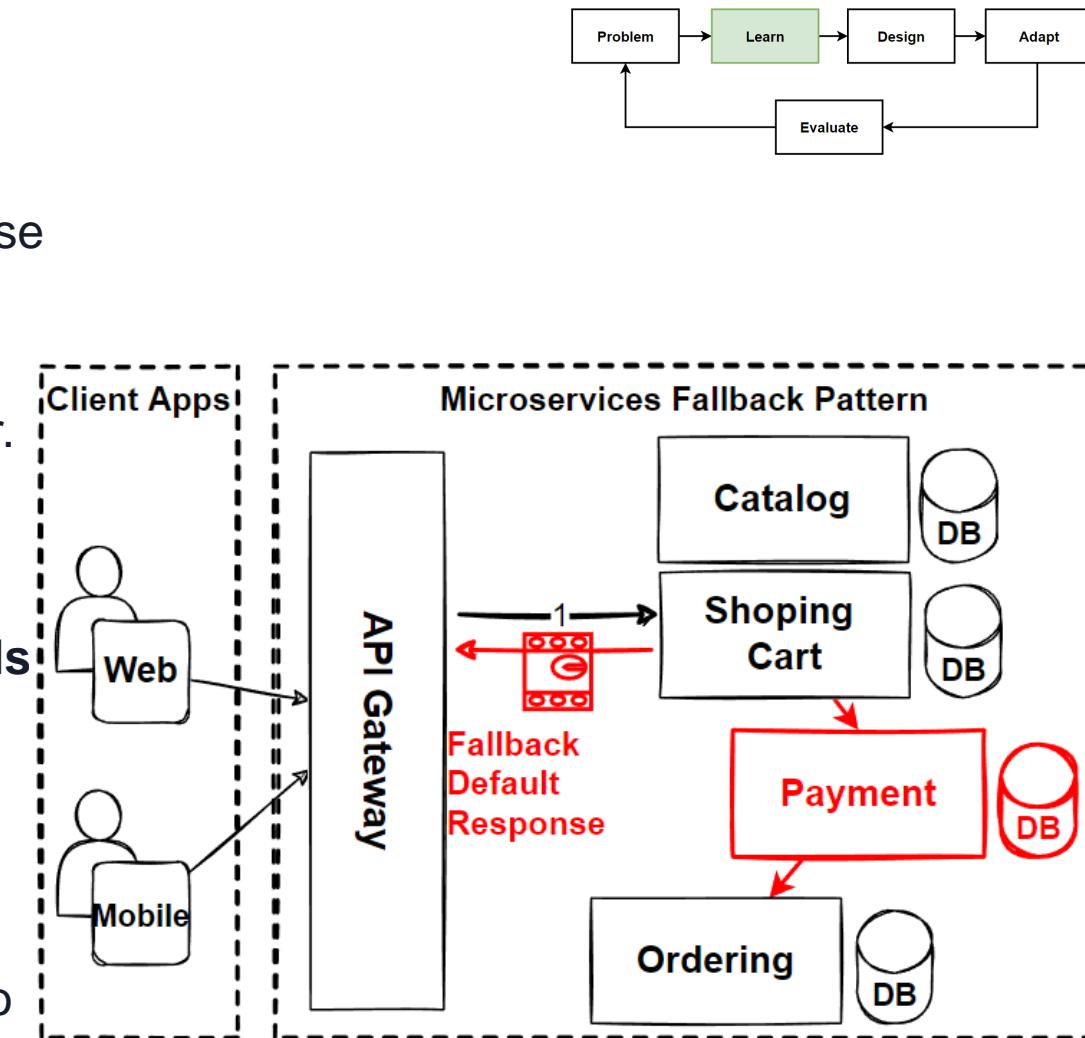
Timeout Pattern

- **Timeout Pattern** provides that **should not wait for a service response for an indefinite amount of time**, throw an exception instead of **waiting too long**.
- It is used to handle scenarios where a **service call takes longer than expected to complete**.
- Setting a **maximum time limit** for the service **call to complete**, if the **time limit is exceeded**, the call is considered to have «**timed out**».
- In microservices, It used to **prevent service calls from taking too long to complete**.
- If the **payment processing service takes longer than expected to complete a request**, the **timeout pattern** used to **cancel the request** and **return an error** to the shopping cart service.



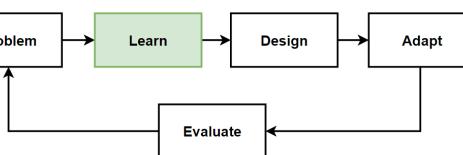
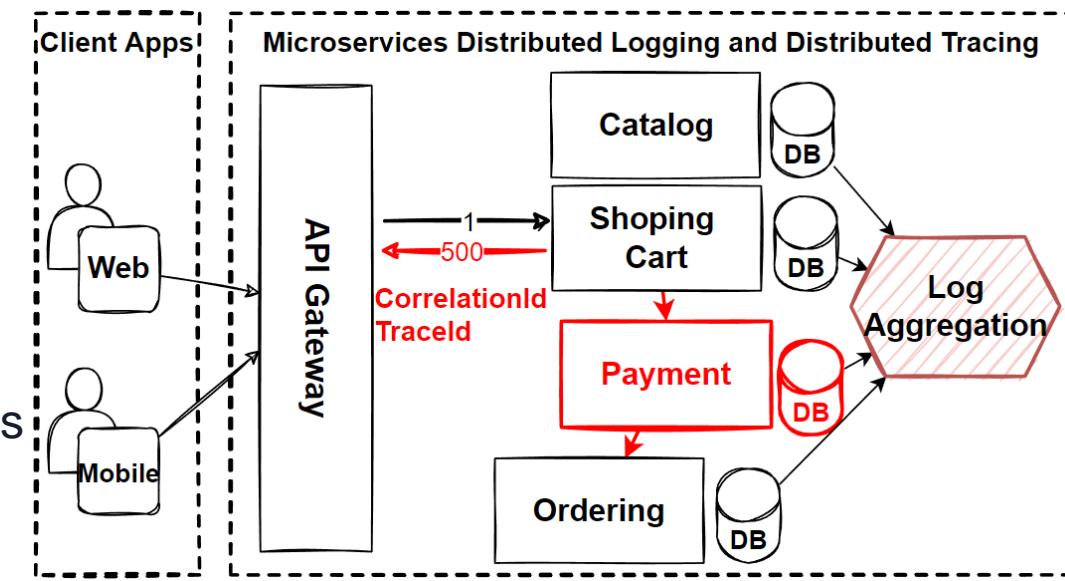
Fallback Pattern

- **Fallback Pattern** provides an **alternative behavior** or response if a **request fails** or **times out**.
- If a **service is unavailable**, the client could **use a cached version** of the **data** or **display a default message** to the user.
- It is used to provide an **alternative course of action** when a **service call fails** or **takes too long to complete**.
- Define a **fallback function** that is called if the **service call fails** or **times out**, and the function provides an **alternative response**.
- In **microservices**, the **fallback pattern** help to **improve the overall reliability and stability of the system**.
- If the **payment processing service is unavailable** or takes too long to complete a request, the **fallback pattern** could be used to **provide an alternative response**.
- I.e. **displaying an error message** or offering the user the option to try again later.



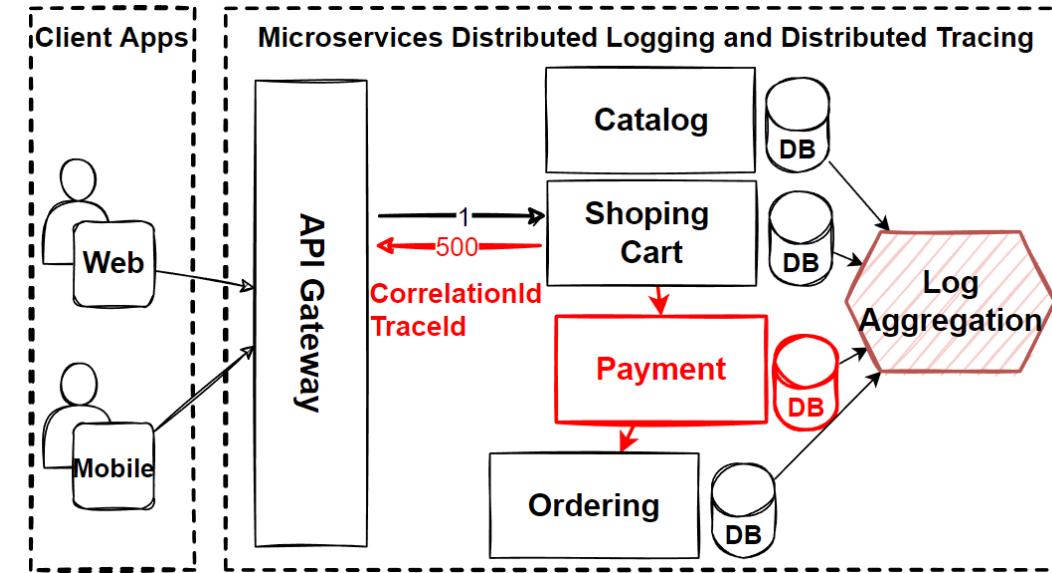
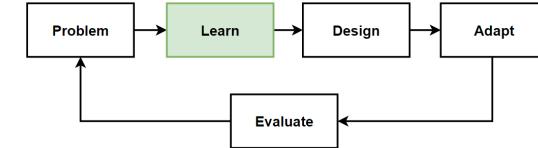
Microservices Observability with Distributed Logging and Distributed Tracing

- Microservice have a **strategy for monitoring and managing the complex dependencies** on microservices
- Need to implement **microservices observability** with using **distributed logging and tracing features**.
- **Microservices Observability** gives us greater **operational insight**.
- **Monitor** and understand the **behavior** and **performance** of a system made up of **microservices**.
- **Distributed Logging** and **Distributed Tracing** are two key tools that improve observability in microservices.
- **Distributed Logging** is a practice of **collecting, storing, and analyzing log data from multiple service instances**.
- **Behavior of the system** over time, identifying patterns and trends, and troubleshooting issues.

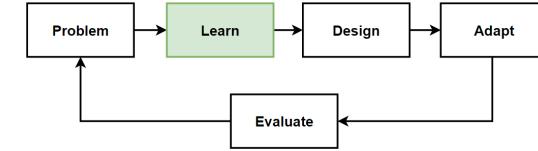


Microservices Observability with Distributed Logging and Distributed Tracing

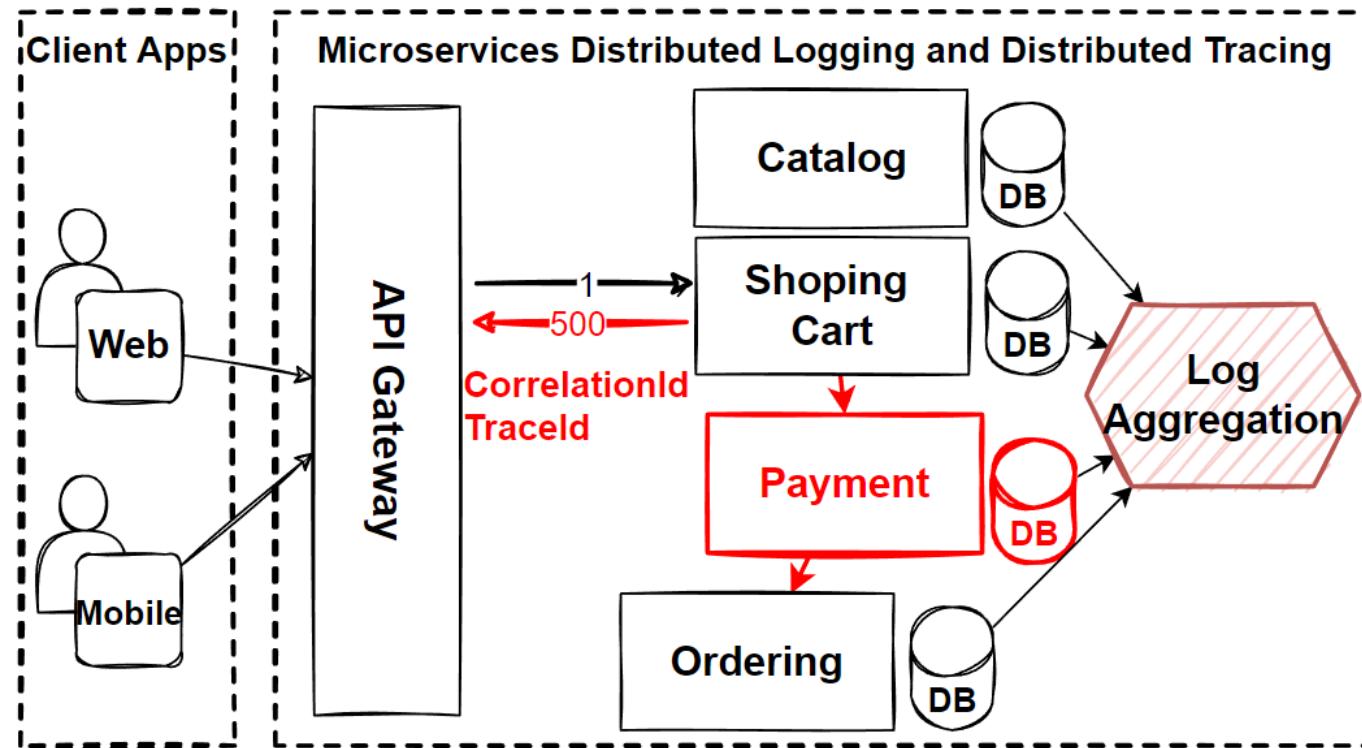
- **Distributed Tracing** is tracking the flow of requests through a microservices architecture, to see how the different service instances **interact with each other**.
- See the **performance of the system, identifying bottlenecks, and troubleshooting issues**.



Real-world Example of Microservices Observability with Distributed Logging and Distributed Tracing

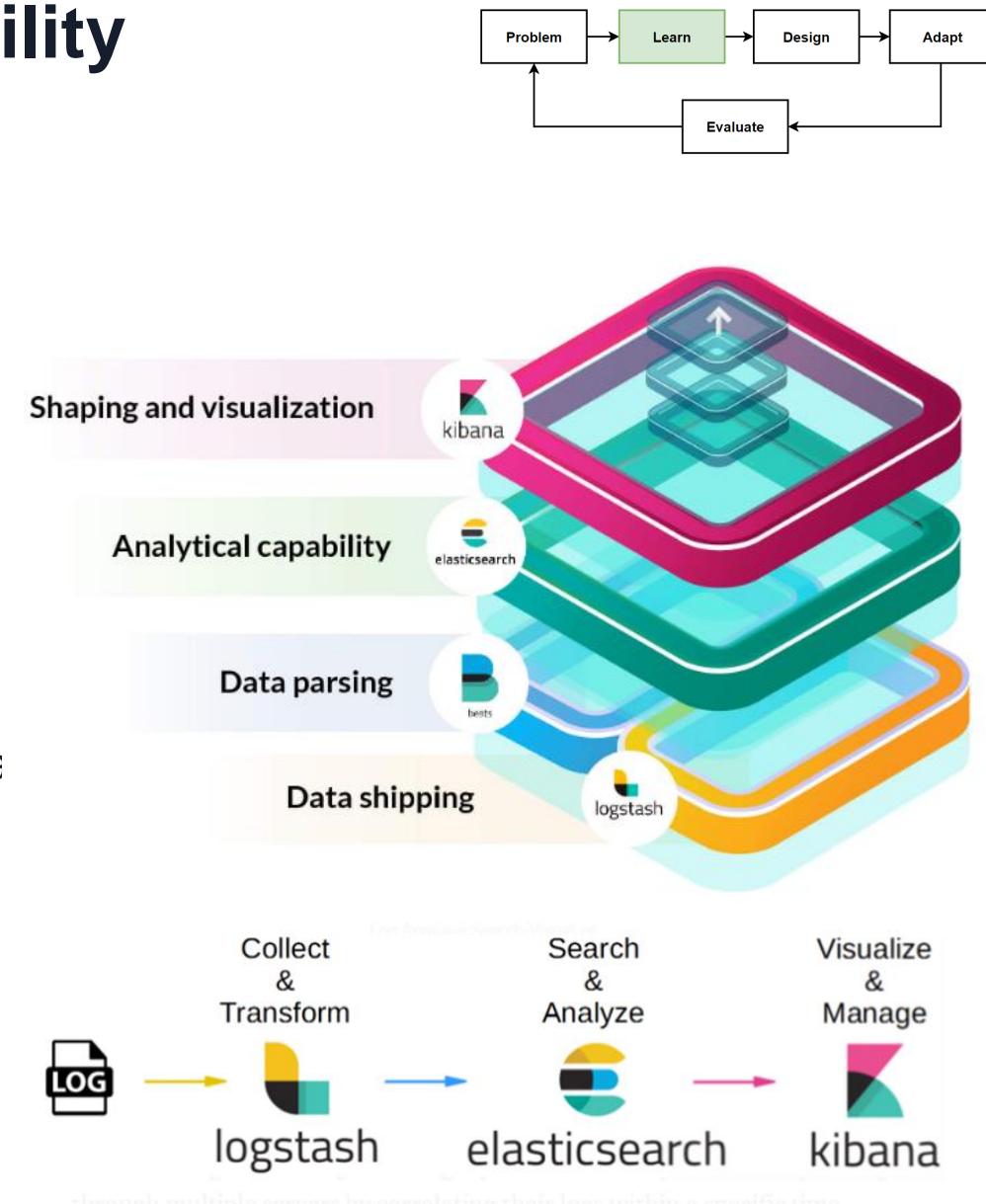


- Implement **distributed logging** to collect, store, and analyze log data from service instances, include information about **service calls**, **errors**, **performance metrics**, and other **relevant data**.
- Implement **distributed tracing** to track the **flow of requests** through the system.
- If a user adds an item to their shopping cart, the trace with **information** about the request flows from the **shopping cart service** to the **payment processing service** to the **product catalog service**.



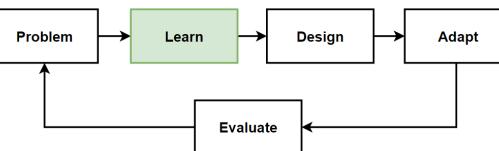
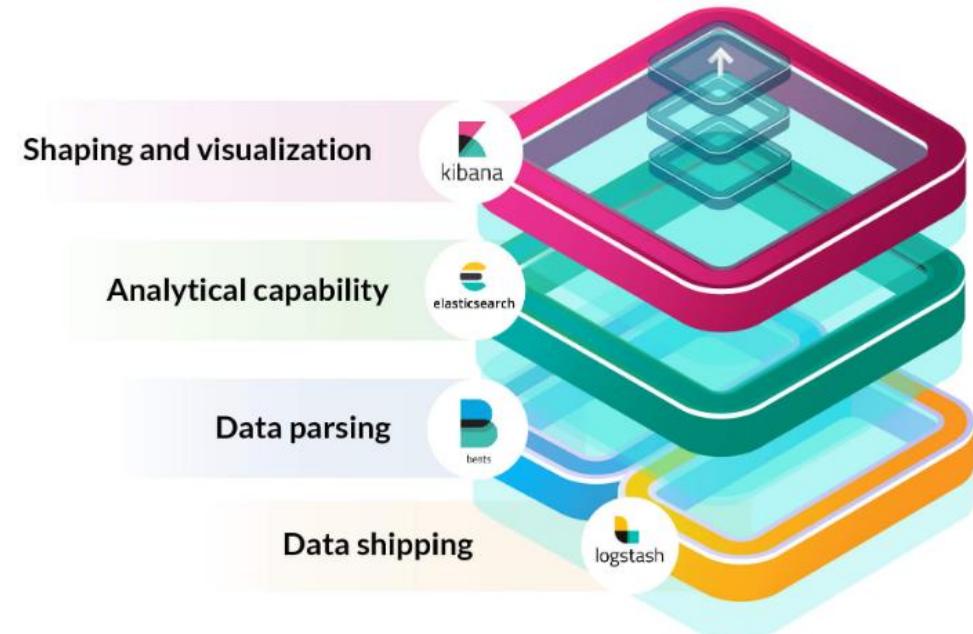
Elastic Stack for Microservices Observability with Distributed Logging

- **Elastic Stack** is a collection of **open-source tools** for **collecting, storing, and analyzing log data** and other types of data.
- Used for **microservices observability** to **provides a flexible and scalable** platform for **monitoring** and understanding the behavior.
- **Elasticsearch**
Distributed search and analytics engine that can be used to store and index log data and other types of data.
- **Logstash**
Data collection and transformation tool that used to collect log data from different sources and send it to Elasticsearch.
- **Kibana**
Data visualization and exploration tool that used to create dashboards and visualizations based on data.
- **Beats**
Collection of lightweight data shippers that used to collect log data and send it to Elasticsearch.

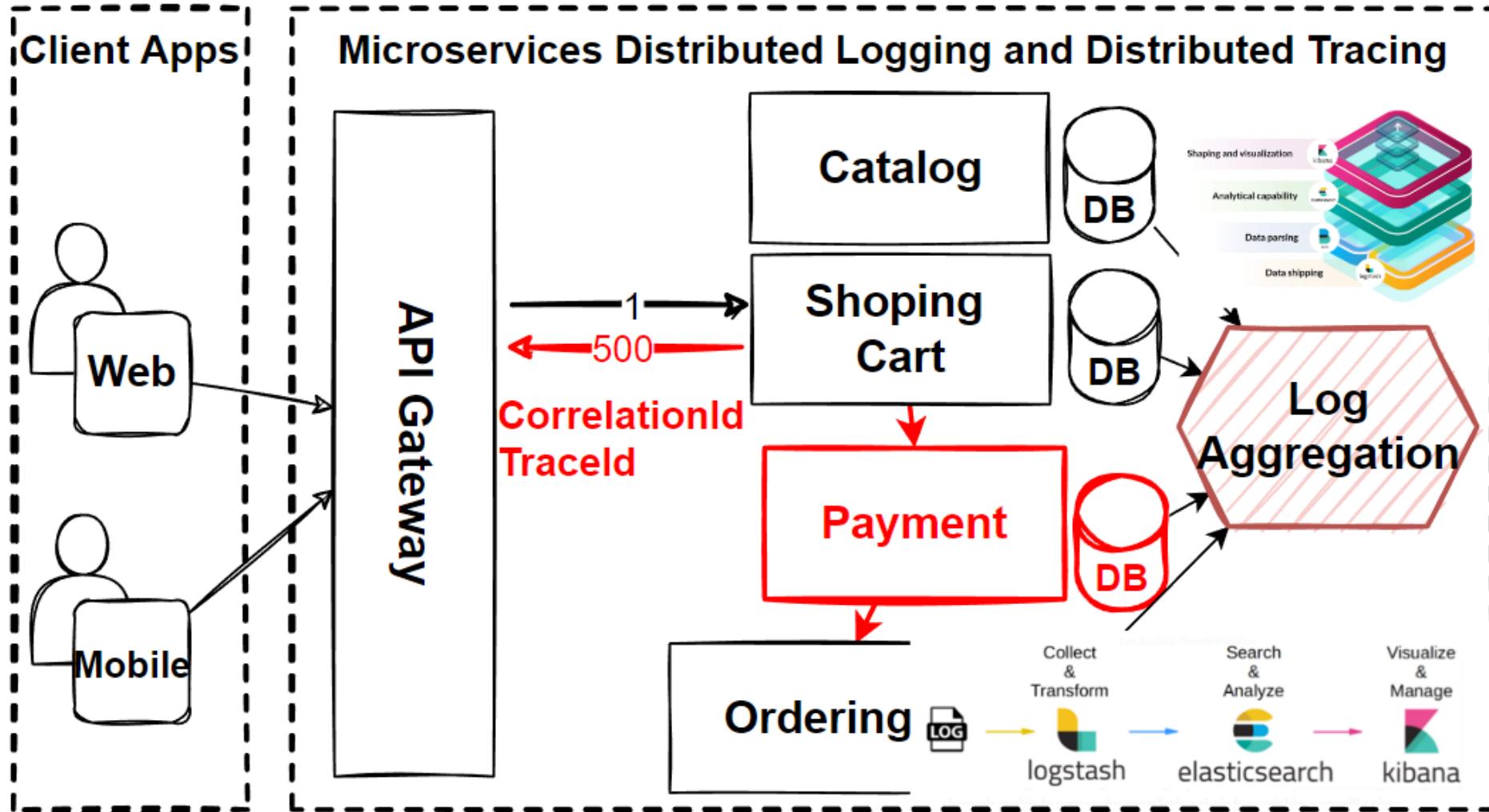
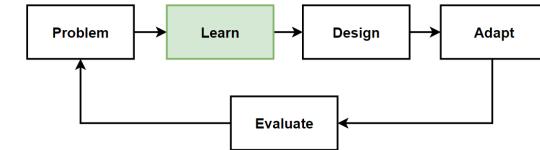


Why logging with Elasticsearch and Kibana in Microservices ?

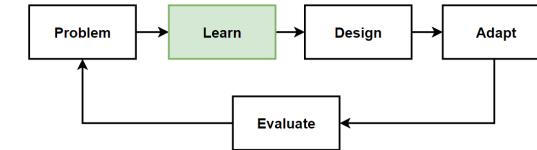
- Logging is critical to implement to identify problems in distributed architecture.
- Elastic Stack used to collect, store, and analyze log data from multiple service instances in a microservices architecture.
- Useful for understanding the behavior of the system over time, identifying patterns and trends, and troubleshooting issues.
- Elastic Stack might be used to collect log data from each service instance.
- Kibana to visualize and analyze the data in order to identify patterns or trends.



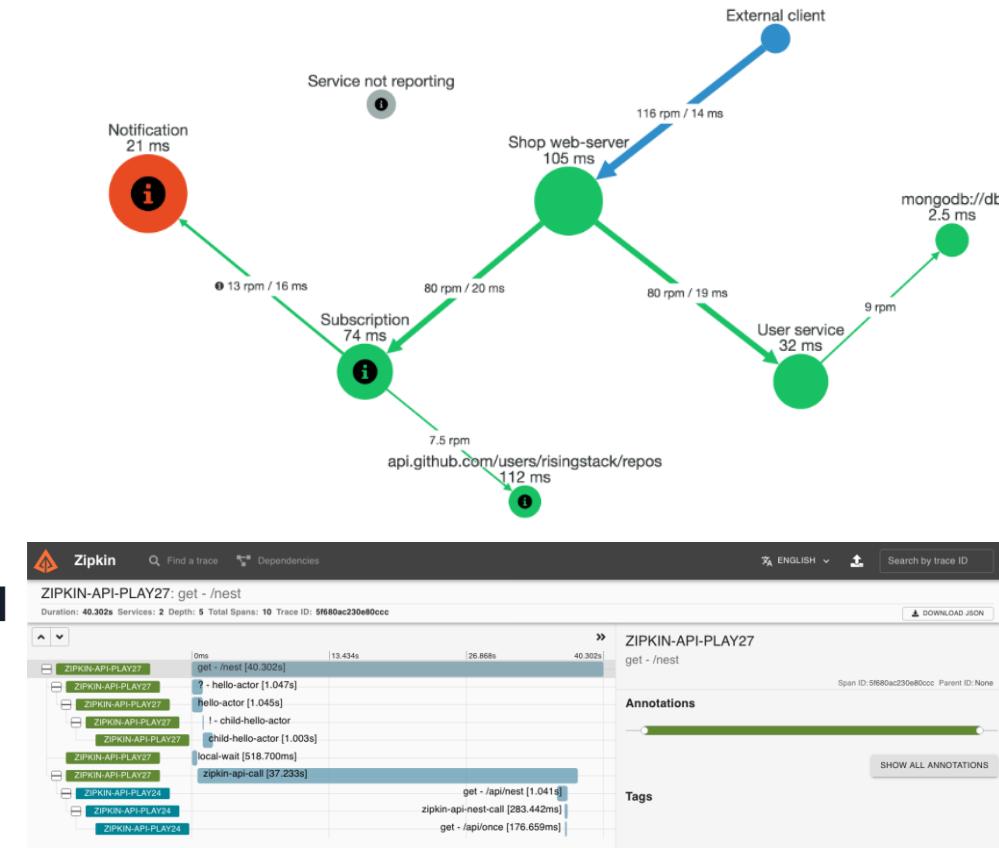
Real-world Example with ElasticStack



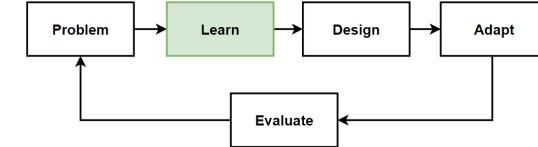
Distributed Tracing with OpenTelemetry using Zipkin



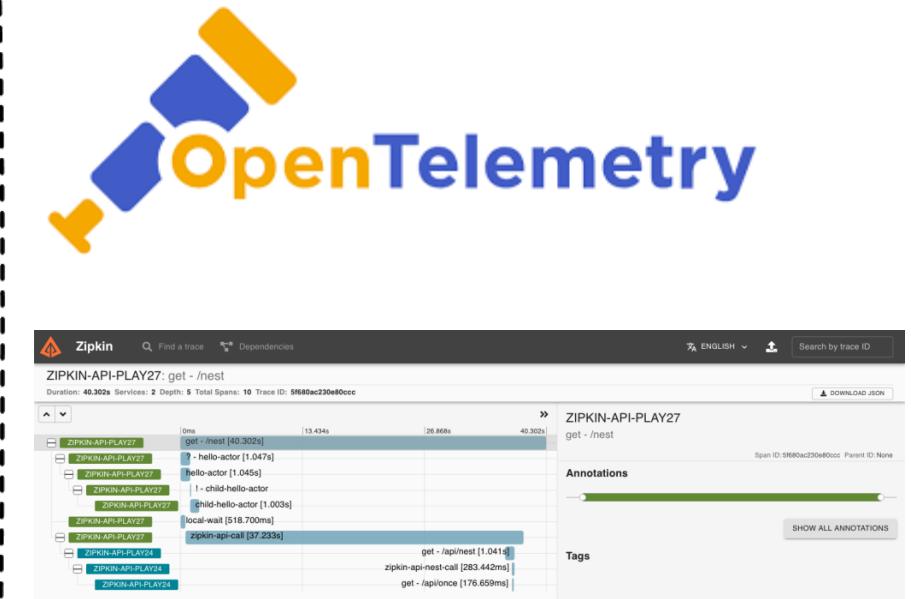
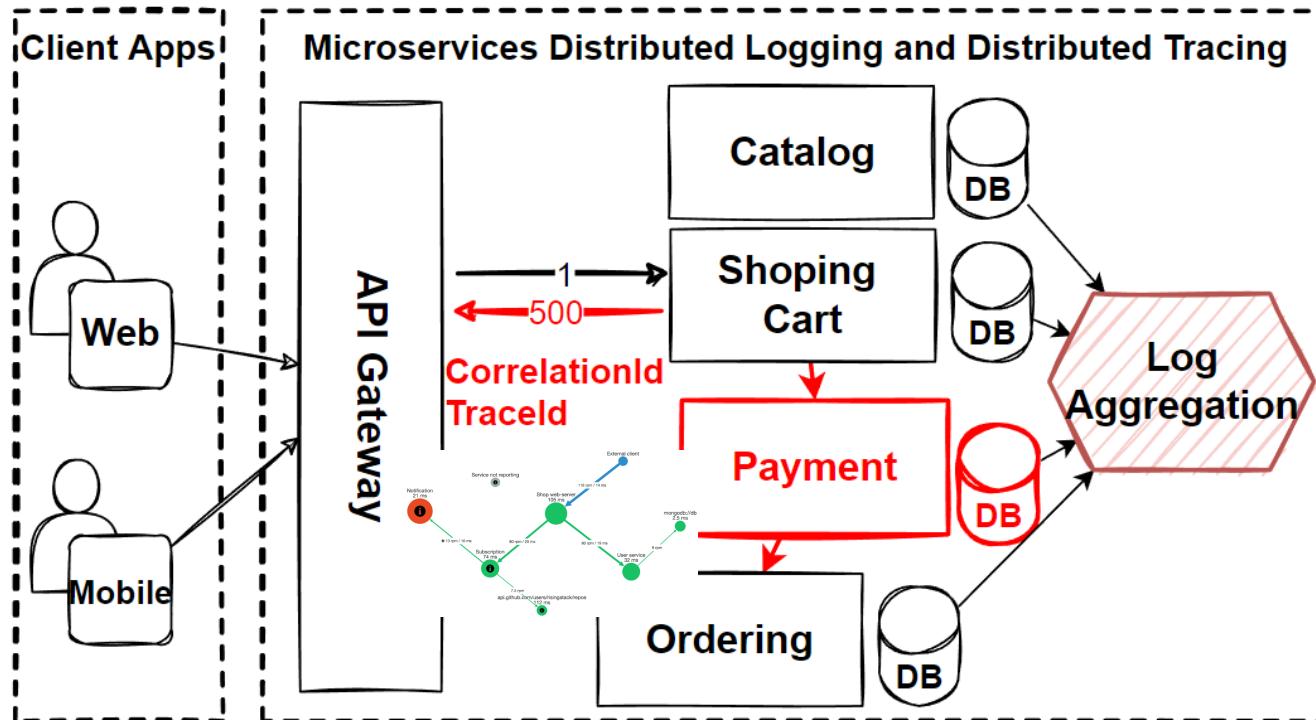
- **Distributed Tracing** is used to **track the flow of a request** as it is processed by different microservices in a system.
- **Different microservices** are interacting and **identify issues** or **bottlenecks** and **troubleshooting issues** in the system.
- **OpenTelemetry** is an **open-source project** that provides a set of APIs for **collecting and exporting telemetry data; traces, metrics, and logs**.
- **Zipkin** is a **distributed tracing system** that **collects and stores** trace data from microservices, provides a web UI for **viewing and analyzing trace data**.
- To use **OpenTelemetry with Zipkin** for **microservices distributed tracing**, **OpenTelemetry SDK** would be **integrated**.
- Allows to **collect trace data** as requests flow through the system, and **send data to a Zipkin server** for **storage and visualization**.



Real-world Example of Distributed Tracing with OpenTelemetry using Zipkin

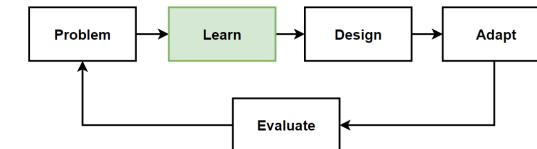


- Use OpenTelemetry with Zipkin to collect trace data as requests flow through the system.
- When a user adds an item to their shopping cart, the trace include information about the request.
- The trace data collected by the OpenTelemetry SDKs integrated into each service instance, and then sent to a Zipkin server for storage and visualization.
- Use the Zipkin UI to visualize and analyze the trace data, to identify patterns or trends.



Microservices Health Checks: Liveness, Readiness and Performance Checks

- The process of monitoring the health and performance of individual microservices in a system.
- The failure of a single microservice can have cascading effects on the rest of the system, important to identify and address issues.
- What is **Health Checks** for microservices ?
- **Health Checks for microservices** are a way to monitor the health and performance of individual microservices in a system.
- Health checks used to determine whether a microservice is functioning properly and is able to handle requests.
- There are 3 types of health checks that can be used for microservices.



Health Checks status				
	NAME	HEALTH	ON STATE FROM	LAST EXECUTION
+	WebMVC HTTP Check	Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	WebSPA HTTP Check	Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Web Shopping Aggregator GW HTTP Check	Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Mobile Shopping Aggregator HTTP Check	Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
-	Ordering HTTP Check	Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
	NAME	HEALTH	DESCRIPTION	DURATION
self		Healthy		00:00:00.0000031
orderingDB-check		Healthy		00:00:00.0008153
ordering-rabbitmqbus-check		Healthy		00:00:00.0097614
+	Basket HTTP Check	Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Catalog HTTP Check	Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Identity HTTP Check	Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Marketing HTTP Check	Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Locations HTTP Check	Healthy	2 minutes ago	12/12/2019, 3:41:18 PM
+	Payments HTTP Check	Healthy	2 minutes ago	12/12/2019, 3:41:18 PM
+	Ordering SignalRHub HTTP Check	Healthy	2 minutes ago	12/12/2019, 3:41:18 PM

Microservices Health Checks: Liveness, Readiness and Performance Checks

▪ Liveness Checks

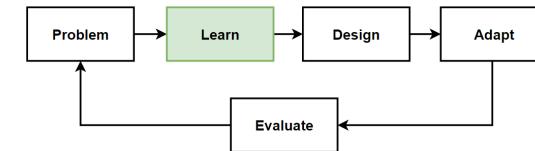
Determine whether a microservice is still running. If a liveness check fails, it may indicate that the microservice has crashed.

▪ Readiness Checks

Determine whether a microservice is ready to handle requests. If a readiness check fails, it may indicate that the microservice is not yet ready to handle traffic.

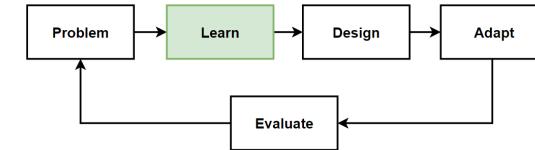
▪ Performance Checks

Monitor the performance of a microservice, such as response times or error rates. If the results of a performance check indicate that a microservice is not performing as expected.



Health Checks status				
	NAME	HEALTH	ON STATE FROM	LAST EXECUTION
+	WebMVC HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	WebSPA HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Web Shopping Aggregator GW HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Mobile Shopping Aggregator HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
-	Ordering HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
	NAME	HEALTH	DESCRIPTION	DURATION
	self	✓ Healthy		00:00:00.0000031
	orderingDB-check	✓ Healthy		00:00:00.0008153
	ordering-rabbitmqbus-check	✓ Healthy		00:00:00.0097614
+	Basket HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Catalog HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Identity HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Marketing HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Locations HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:18 PM
+	Payments HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:18 PM
+	Ordering SignalRHub HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:18 PM

Microservices Health Monitoring with Kubernetes, Prometheus and Grafana



- **Use Liveness and Readiness Probes**

Kubernetes provides liveness and readiness probes that can be used to monitor the health of individual microservices.

- Liveness probes check to see if a microservice is still running, and readiness probes check to see if a microservice is ready to receive traffic.

- **Use Monitoring Tools**

Monitoring tools can be used to monitor the health and performance of microservices that can be integrated with Kubernetes to provide alerts or notifications when issues arise. I.e. Prometheus, Grafana, Datadog.

- **Use Log Analysis Tools**

Analyze log messages generated by your microservices and identify issues or trends. Elastic Stack (Elasticsearch, Logstash, Kibana), Fluentd, Splunk.

- **Set up Alerts and Notifications**

Setting up alerts and notifications can help to ensure that relevant parties are notified when issues arise, allowing them to be addressed quickly. Slack, Teams, Email, SMS.

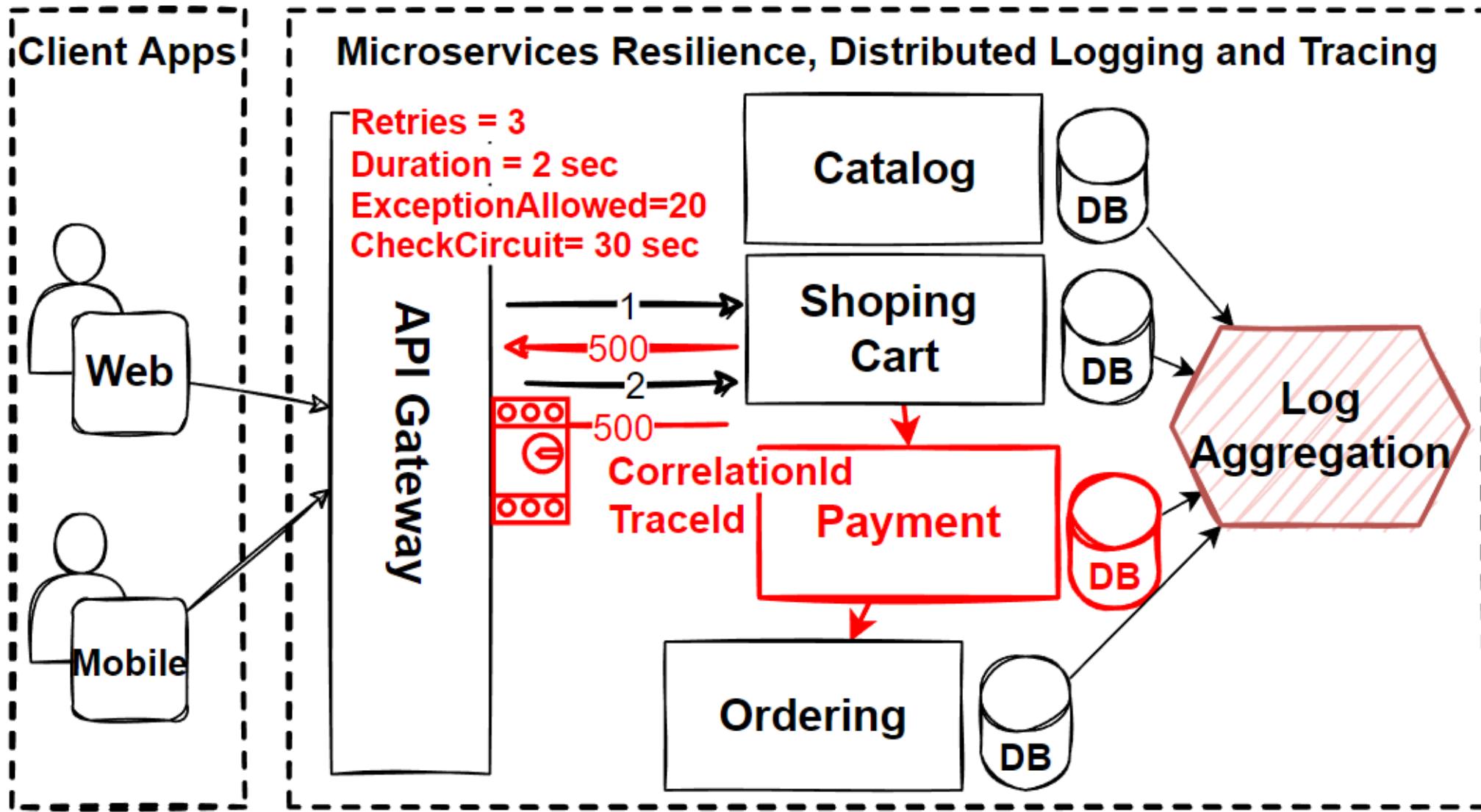
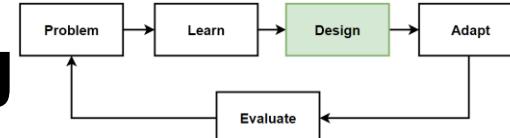
Before Design – What we have in our design toolbox ? - Old

Architectures	Patterns&Principles	Microservices Deployment	Non-FR	FR
• Microservices Architecture	• The Database-per-Service Pattern, Polygot Persistence, Decompose services by scalability, The Scale Cube	• Docker and Kubernetes Architecture, Helm Charts	• High Scalability	• List products
• Event-Driven Microservices Architecture	• Microservices Decomposition Pattern • Microservices Communications Patterns • Microservices Data Management Patterns • Event-Driven Architecture • Microservices Distributed Caching • Microservices Deployments with Containers and Orchestrators	• Kubernetes Patterns; Sidecar Patterns, Service Mesh Pattern • DevOps and CI/CD Pipelines • Deployment Strategies; Blue-green, Rolling, Canary and A/B Deployment. • Infrastructure as code (IaC)	• High Availability • Millions of Concurrent User • Independent Deployable • Technology agnostic • Data isolation • Resilience and Fault isolation	• Filter products as per brand and categories • Put products into the shopping cart • Apply coupon for discounts • Checkout the shopping cart and create an order • List my old orders and order items history

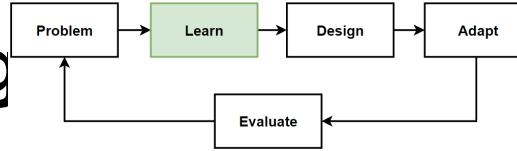
Before Design – What we have in our design toolbox ? - New

Architectures	Patterns&Principles	Microservices Resilience	Non-FR	FR
• Microservices Architecture	• The Database-per-Service Pattern, Polygot Persistence, Decompose services by scalability, The Scale Cube	• Resilience Patterns; Retry, Circuit-Breaker, Bulkhead, Timeout, Fallback Pattern	• High Scalability • High Availability • Millions of Concurrent User	• List products • Filter products as per brand and categories • Put products into the shopping cart • Apply coupon for discounts • Checkout the shopping cart and create an order • List my old orders and order items history
• Event-Driven Microservices Architecture	• Microservices Decomposition Pattern • Microservices Communications Patterns • Microservices Data Management Patterns • Event-Driven Architecture • Microservices Distributed Caching • Microservices Deployments with Containers and Orchestrators	• Distributed Logging and Distributed Tracing • Elastic Stack; Elasticsearch + Logstash + Kibana • OpenTelemetry using Zipkin • Kubernetes Health Monitoring with tools like Prometheus and Grafana • Microservices Resilience, Observability and Monitoring	• Independent Deployable • Technology agnostic • Data isolation • Resilience and Fault isolation	

Microservices Resilience, Observability and Monitoring



Microservices Resilience, Observability and Monitoring



- **Improved fault tolerance**

Help a microservices system continue functioning even in the face of failures or other disruptions, making it more fault tolerant.

- **Improved uptime and availability**

Implementing patterns circuit breakers and fallback mechanisms, you can ensure that your system remains available even when individual components get problems.

- **Increased reliability**

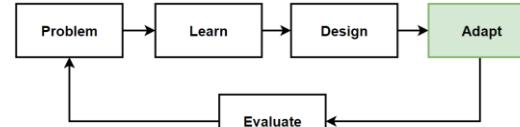
When a microservices system is designed to be resilient, it is less likely to experience failures or disruptions.

- **Enhanced scalability**

Microservices architecture is designed to be scalable, and using resilience patterns can further enhance this capability.

- The bulkhead pattern can allow you to scale different components of your system independently, without affecting the others.

Adapt: Microservices Resilience, Observability and Monitoring



Frontend SPAs Resiliency Pattern

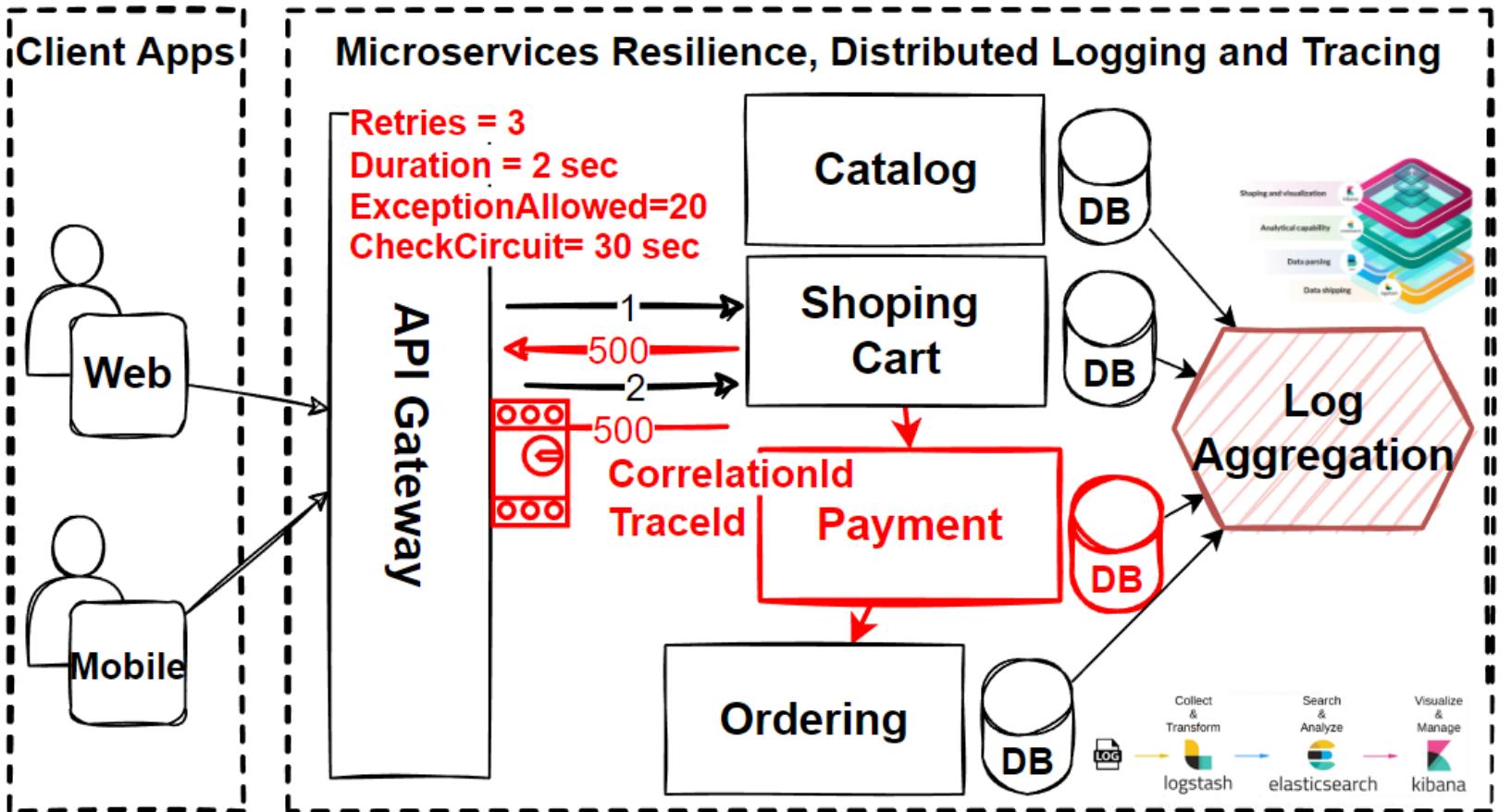
- Angular
- Vue
- React
- Retry
- Circuit breaker
- Bulkhead

API Gateways

- Timeout, Fallback
- Kong Gateway
- Express Gateway
- Amazon AWS API Gateway

Backend Microservices

- Java – Spring Boot
- .Net – Asp.net
- JS – NodeJS



Distributed Logging Cloud Distributed

- Elastic Stack
- Fluentd
- Splunk
- AWS Elastic Cloud
- Azure Elastic

Distributed Tracing

- OpenTelemetry
- Zipkin
- AWS X-Ray

Monitoring

- Kubernetes Health Checks
- Prometheus – Grafana, Datadog

Cloud Monitoring

- Azure Application Insights
- Amazon CloudWatch

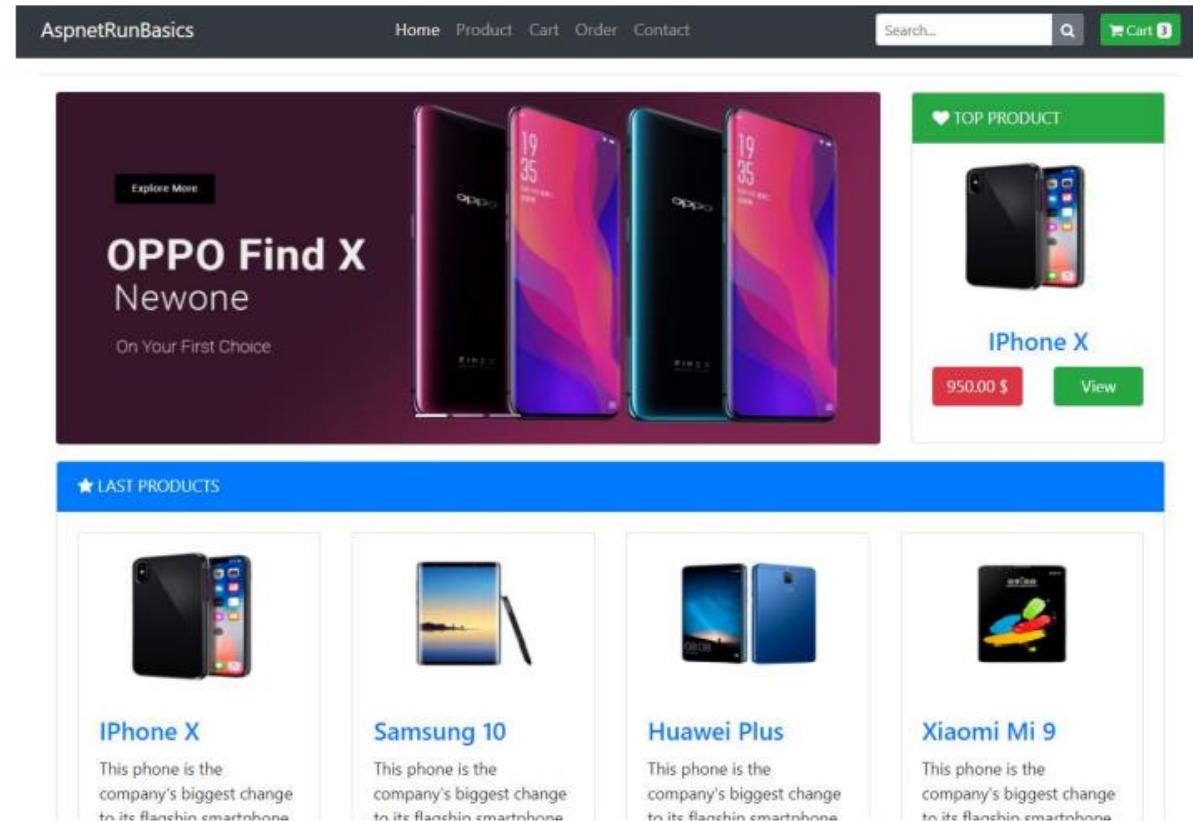
DEMO: E-commerce Implementation of Microservices Architecture

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 Clean Architecture implementation.

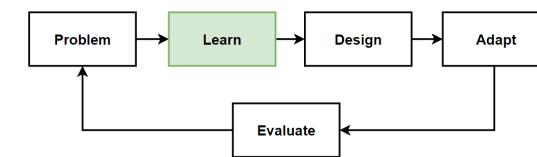
Centralized Distributed Logging with Elasticsearch, Kibana and SeriLog, use the HealthChecks with Watchdog, Implement Retry and Circuit Breaker patterns with Polly.

Implementation of Microservices Architecture

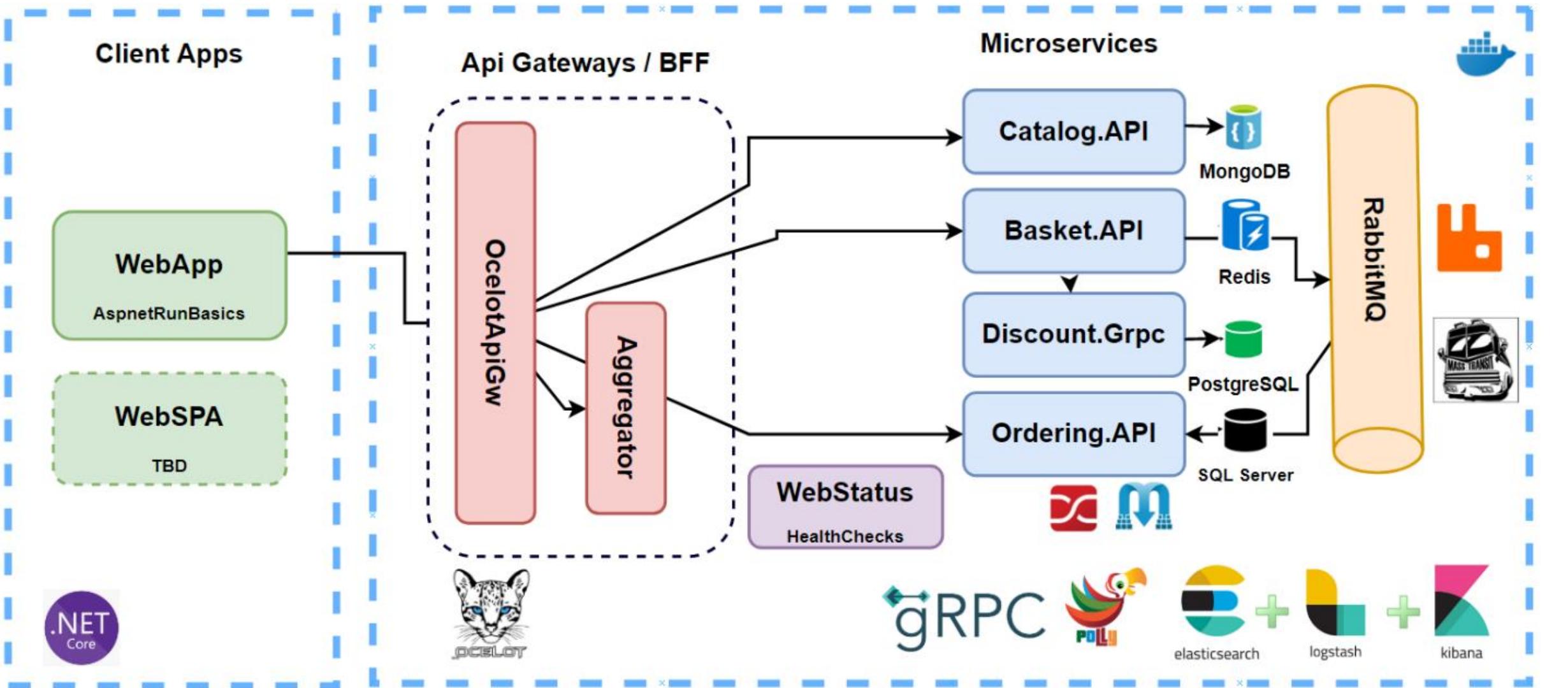
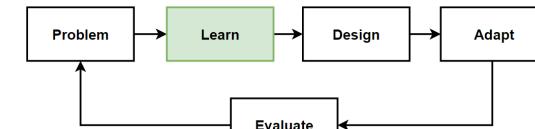
- Clone Microservices Repository
- Docker-compose up
- Follow steps on github documentation
- Run the Project Section
- DEMO



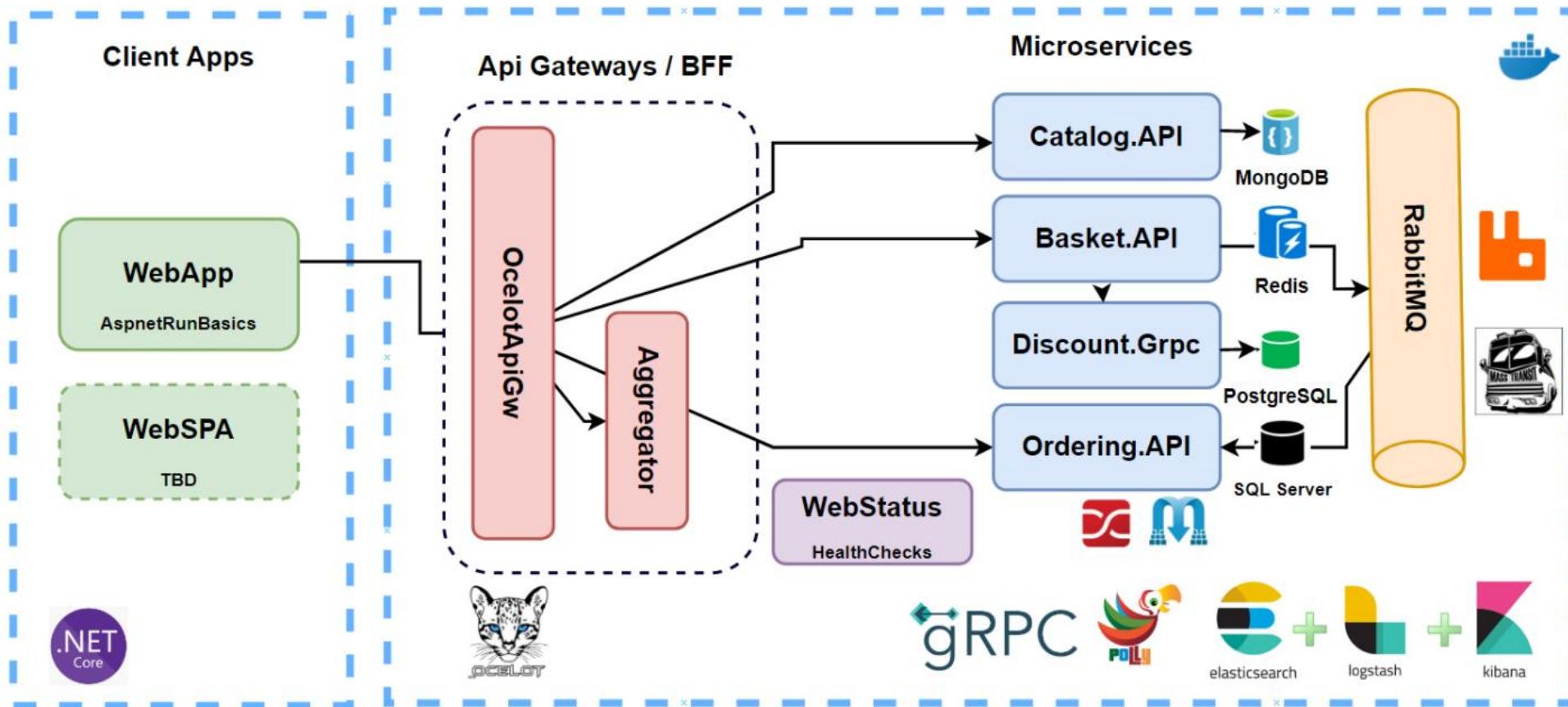
<https://github.com/aspnetrun/run-aspnetcore-microservices>



Big Picture of Microservices Architecture



DEMO: Microservices Architecture Code Review



DEMO: Code review of Microservices Architecture .NET Implementation

- <https://github.com/aspnetrun/run-aspnetcore-microservices>
- <https://github1s.com/aspnetrun/run-aspnetcore-microservices>

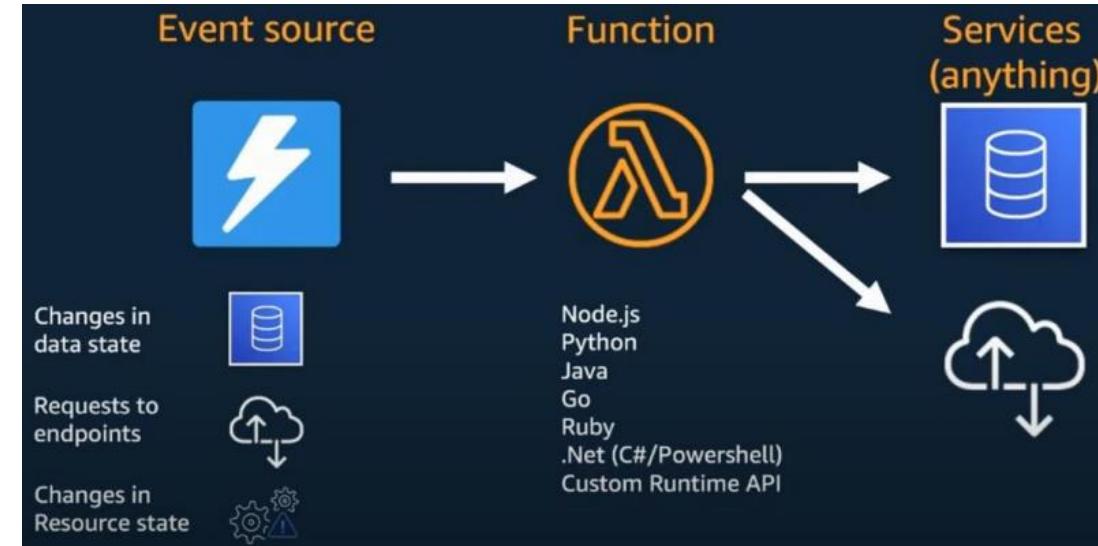
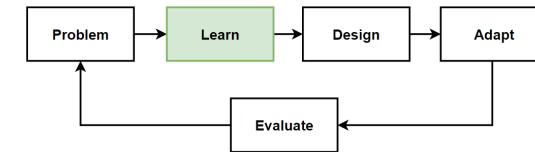
Alternative Link - <https://github.com/mehmetozkaya/MicroservicesApp>

Serverless Microservices Architecture

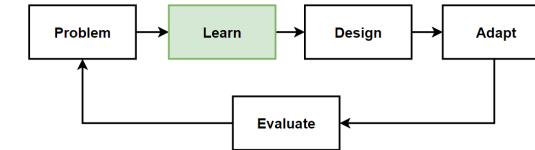
AWS Event-driven Serverless Microservices using AWS Lambda, API Gateway,
EventBridge, SQS, DynamoDB and CDK for IaC

Introduction - Serverless Microservices

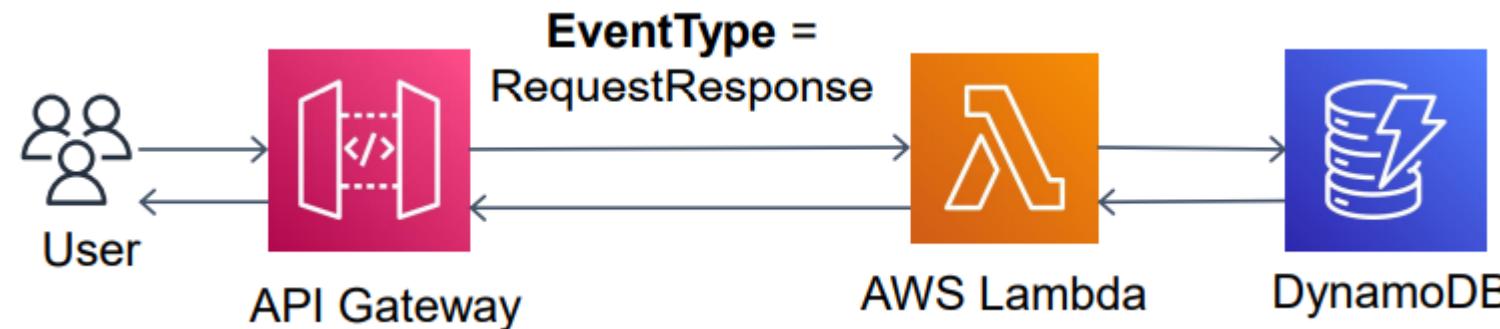
- **Serverless microservices** in which the **infrastructure** is managed by a **cloud provider** and the microservices are **executed in a serverless computing environment**.
- **Microservices are automatically scaled** to meet the **demand of the application** and developer **does not worry** about provisioning and managing servers.
- **AWS Lambda, Azure Functions and Google Cloud Functions** are a **serverless computing** platforms provided by AWS, Microsoft Azure and Google Cloud.
- It allows developers to **write and deploy microservices** that **run in response to specific events**, such as an **HTTP request** or a **change to a database**.



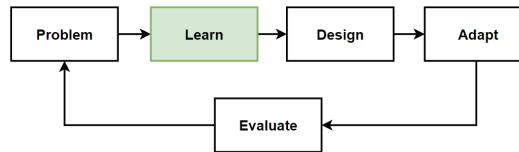
Example of Serverless Microservices



- **Write the microservice code** in a programming language supported by AWS Lambda, such as Node.js, Python, or Java.
- **Package the code** and any dependencies into a deployable package, such as a ZIP file.
- **Create an AWS Lambda function** and upload the package.
- **Configure the function** to be triggered by a specific event, such as an HTTP request.
- **Test and debug the function** using the AWS Lambda console or command-line tools.
- **Deploy the function** to a production environment and monitor its performance using AWS CloudWatch.

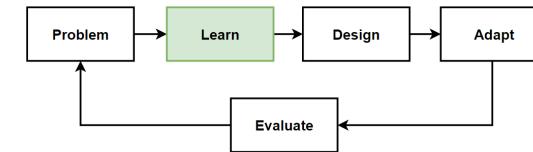


AWS Serverless services used

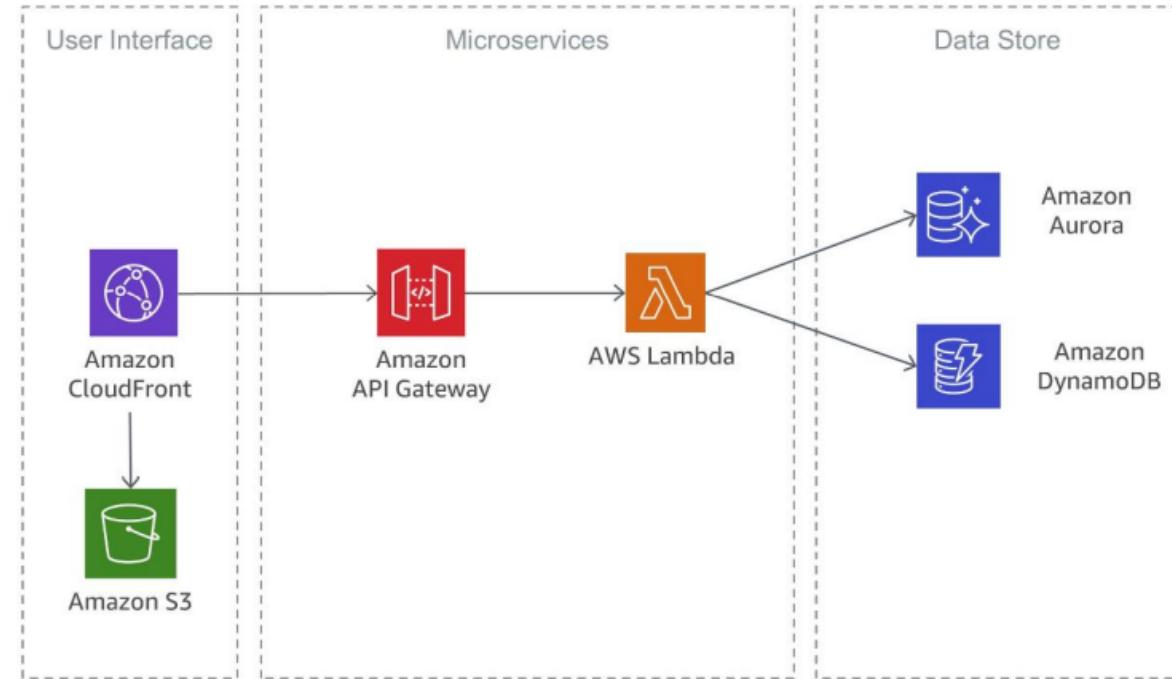


Compute	Databases	Messaging/Application Integrations	API Management	IaC	Monitoring
AWS Lambda	Amazon DynamoDB	Amazon EventBridge	Amazon API Gateway	AWS CloudFormation	Amazon CloudWatch
		Amazon Simple Queue Service (Amazon SQS)			

AWS Lambda as a Microservice

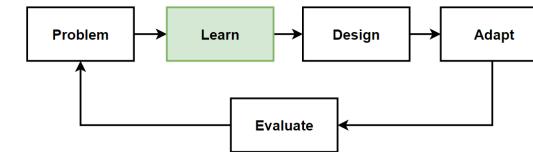


- Microservice are **small business services** that can work together and can be deployed **autonomously / independently**.
- **Lambda** is a service that allows you to run your functions in the cloud **completely serverless** and eliminates the operational **complexity**.
- It **integrates** with the **API gateway**, allows you to invoke functions with the API calls, and makes your architecture completely serverless.
- Microservice with **AWS Lambda**, which removes the architectural overhead of designing for **scaling** and high **availability**,
- Eliminating the **operational efforts** of operating and monitoring the microservice's underlying infrastructure.

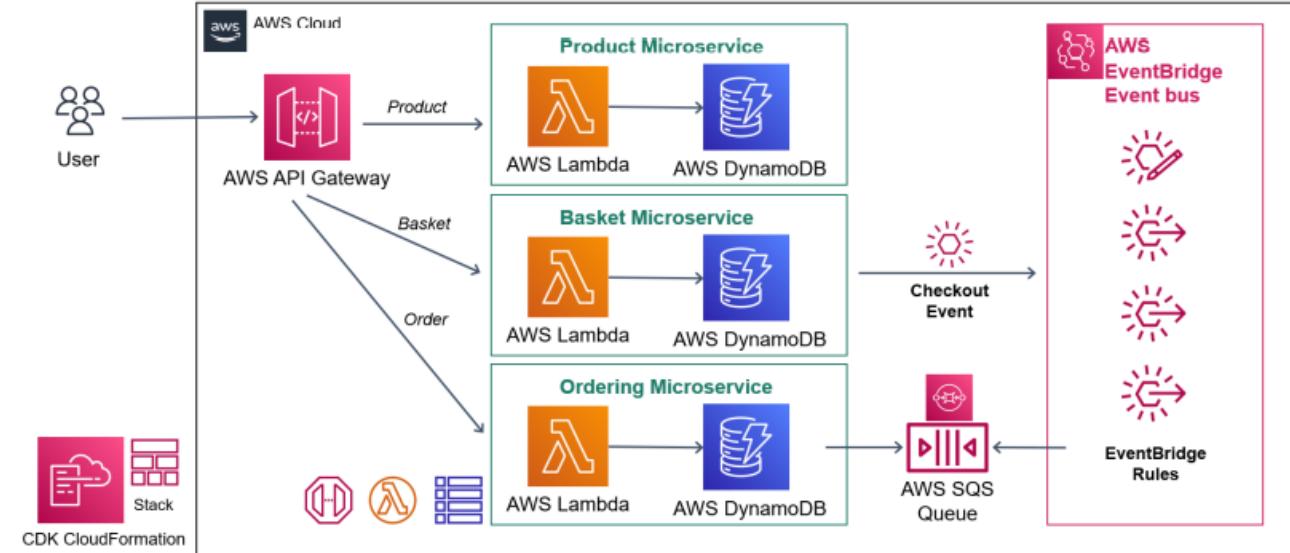


<https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/serverless-microservices.html>

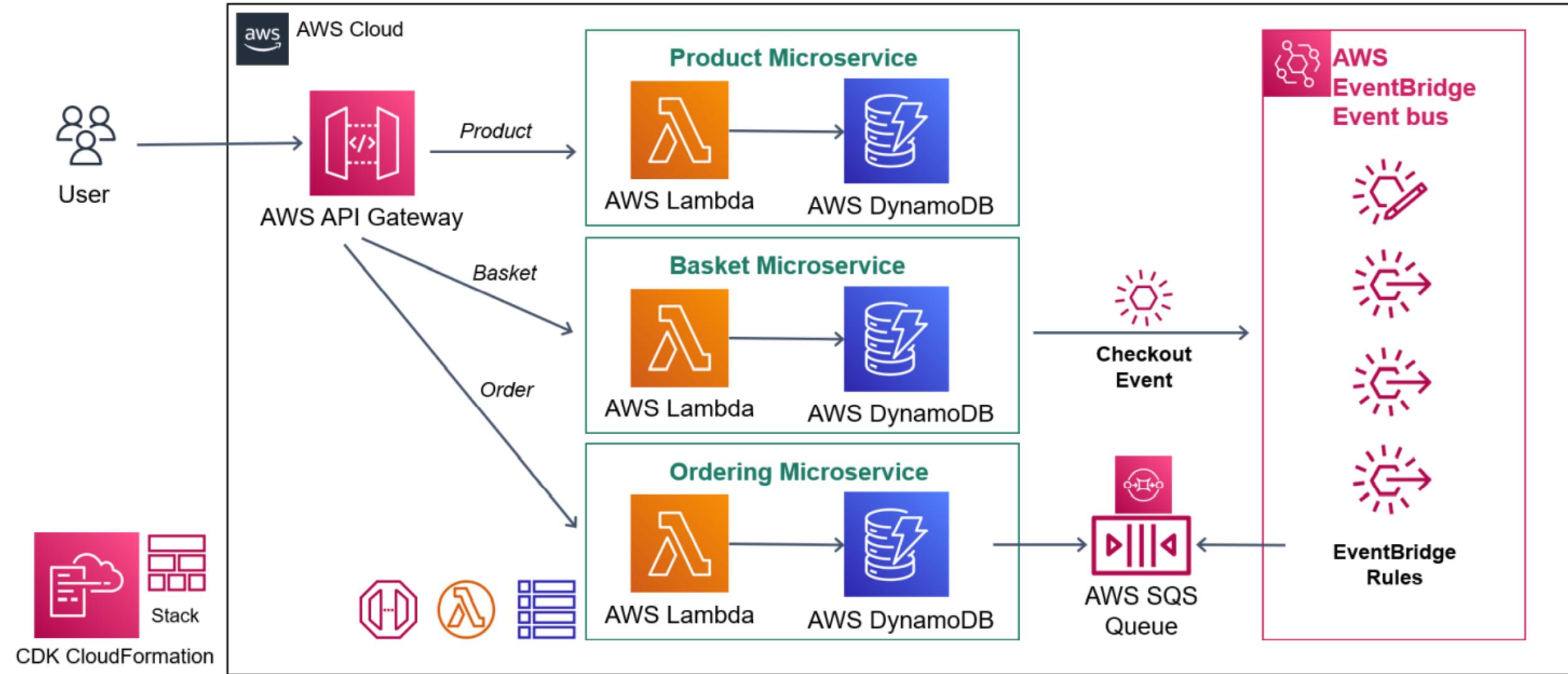
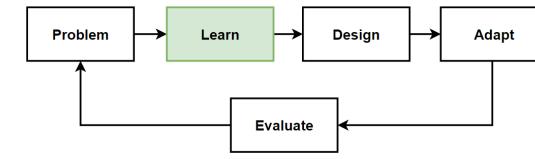
Serverless E-Commerce Microservice Architecture



- Developed in Serverless environment
 - developing with AWS Lambda
 - No Deployment Requirement
 - and IaC by AWS CDK CloudFormation
- **AWS API Gateway**
- **Microservices** developed with AWS Lambda
- **Databases** are no-sql DynamoDB that can be key-value pair or document databases
- **Message broker system** which is Amazon EventBridge Eventbus
- **Amazon CloudWatch** for centralized logging and monitoring
- By default automatic scalability and high availability



Design: Serverless E-Commerce Microservices



Thanks

Thank you so much for being with me on this journey.

Reviews and feedback is really encourage to me for pushing forward to create new courses like this.

Mehmet Ozkaya

