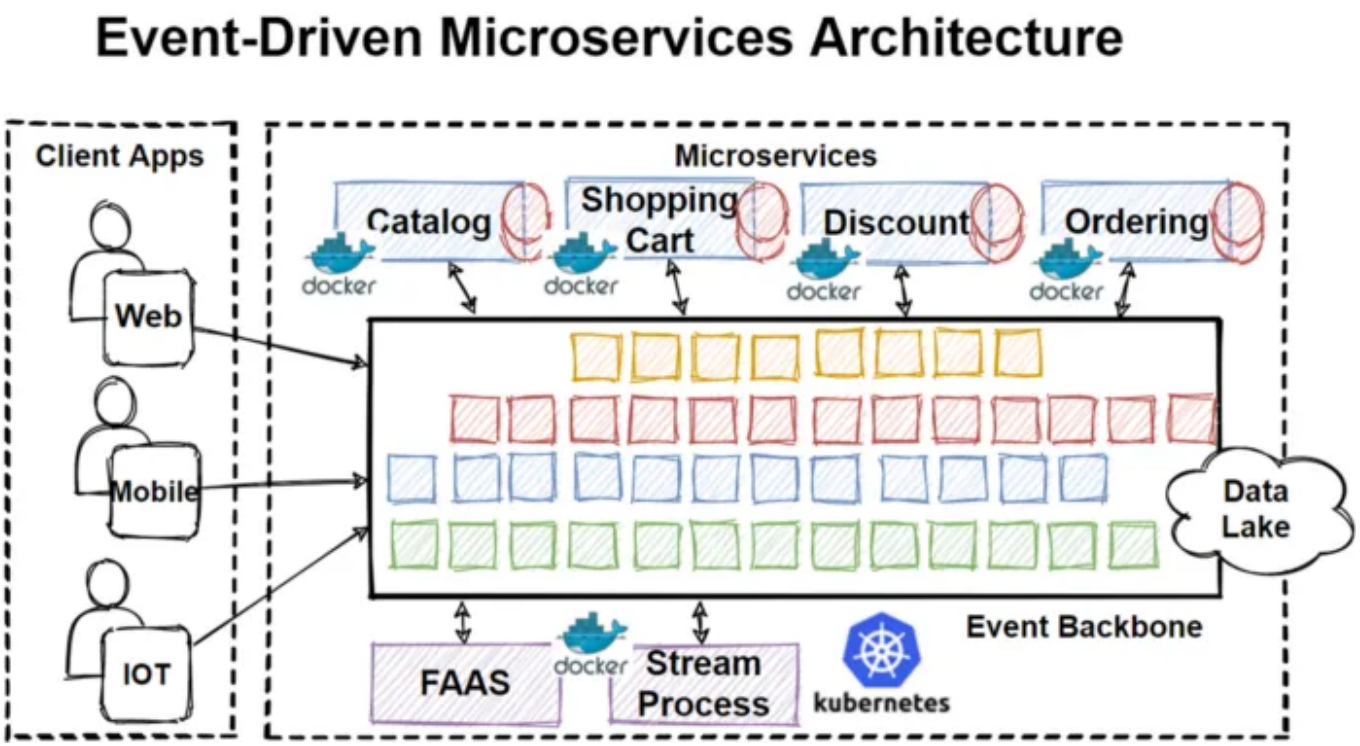
Monolithic to Microservices Architecture with Patterns & Best Practices

Mehmet Ozkaya · Follow
Published in Design Microservices Architecture with Patterns & Principles ·
19 min read · Aug 20, 2021

In this article, we will learn **how to Design Microservices Architecture** using **Design Patterns, Principles**, and **Best Practices.** We will use proper architectural design patterns and techniques.

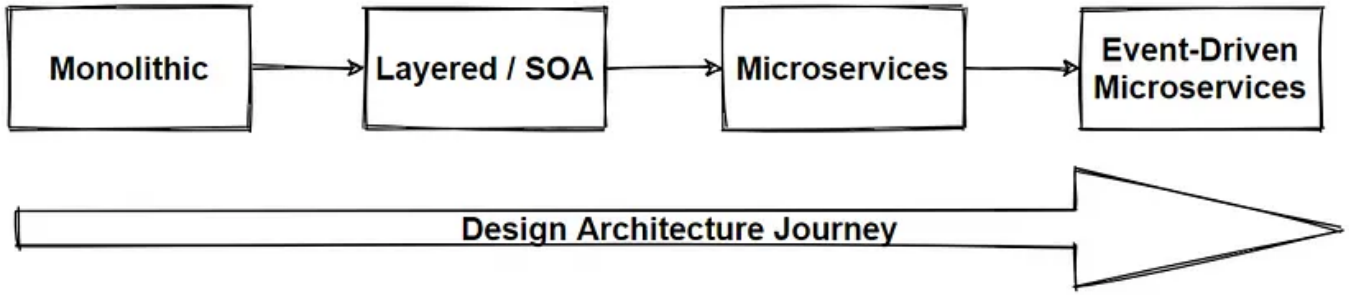
By the end of the article, you will learn how to **handle millions of requests** by designing systems for **high availability, high scalability, low latency,** and **resilience** to network failures on microservices distributed architectures.



Event-Driven Architecture

This course will be the **journey of software architecture design** with a step-by-step process, evolving from a monolithic architecture to an event-driven microservices architecture.

We will start with the basics of software architecture by designing **monolithic e-commerce architecture** that could handle a low number of requests.



Journey of Design Architectures

After that, step by step, the architecture will be evolved as follows:

- **Layered Architecture**
- **Service-Oriented Architecture (SOA)**
- **Microservices**
- and lastly **Event-Driven Microservices Architectures**

The end-state will provide the capability to handle millions of requests following this approach.

Step by Step Design Architectures w/ Course

I have just published a new course — Design Microservices Architecture with Patterns & Principles.

In this course, we will learn **how to Design Microservices Architecture** using **Design Patterns, Principles,** and **Best Practices**.

We will start with designing **Monolithic** that will evolve into an **Event-Driven Microservices** step by step and using the correct architecture design patterns and techniques.

Article Flow

The article will provide theoretical and practical information.

- We will learn a specific pattern, **why** and **where** we should **use**

- After that, we will see the **reference architectures** that applied these patterns
- After that, we will design our architecture by applying this **newly learned pattern** together
- And lastly, we will decide which **technologies** to use based on those architectures.

So, we will **Iterate** and **Evolve** from the **Monolithic architecture** to **Event-Driven Microservices Architecture.**

Evolve architecture

We will evolve these architectures as per the questions:

- How can we scale the application?
- How many requests that we need to handle in our application?
- How many seconds of latency is acceptable for our architecture?

So, we evolve these questions according to:

Non-Functional Requirements

Scalability and reliability are measures of how well your application will behave to serve end-users. If our e-commerce application can handle millions of users without noticeable downtime, we can say the system is highly scalable and reliable. **Scalability** and **Availability** are probably the relevant characteristics when designing a good architecture.

Non-Functional Requirements:

- **Scalability**: e-commerce applications should be able to serve millions of users
- **Availability**: e-commerce applications should be available 24/7
- **Maintainability**: e-commerce applications should not be complex to maintain.
- **Efficiency**: e-commerce applications should respond with acceptable latency, less than 2 seconds.

Request per Second and Acceptable Latency

OK, let's talk about acceptable latency; how can we make our application for an acceptable latency if our application gets used by more and more users?

See the following table for more information:

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

Request per Second and Acceptable Latency

As you can see in the table, we will start a small e-commerce application that gets only 2.000 concurrent users and 500 requests per second.

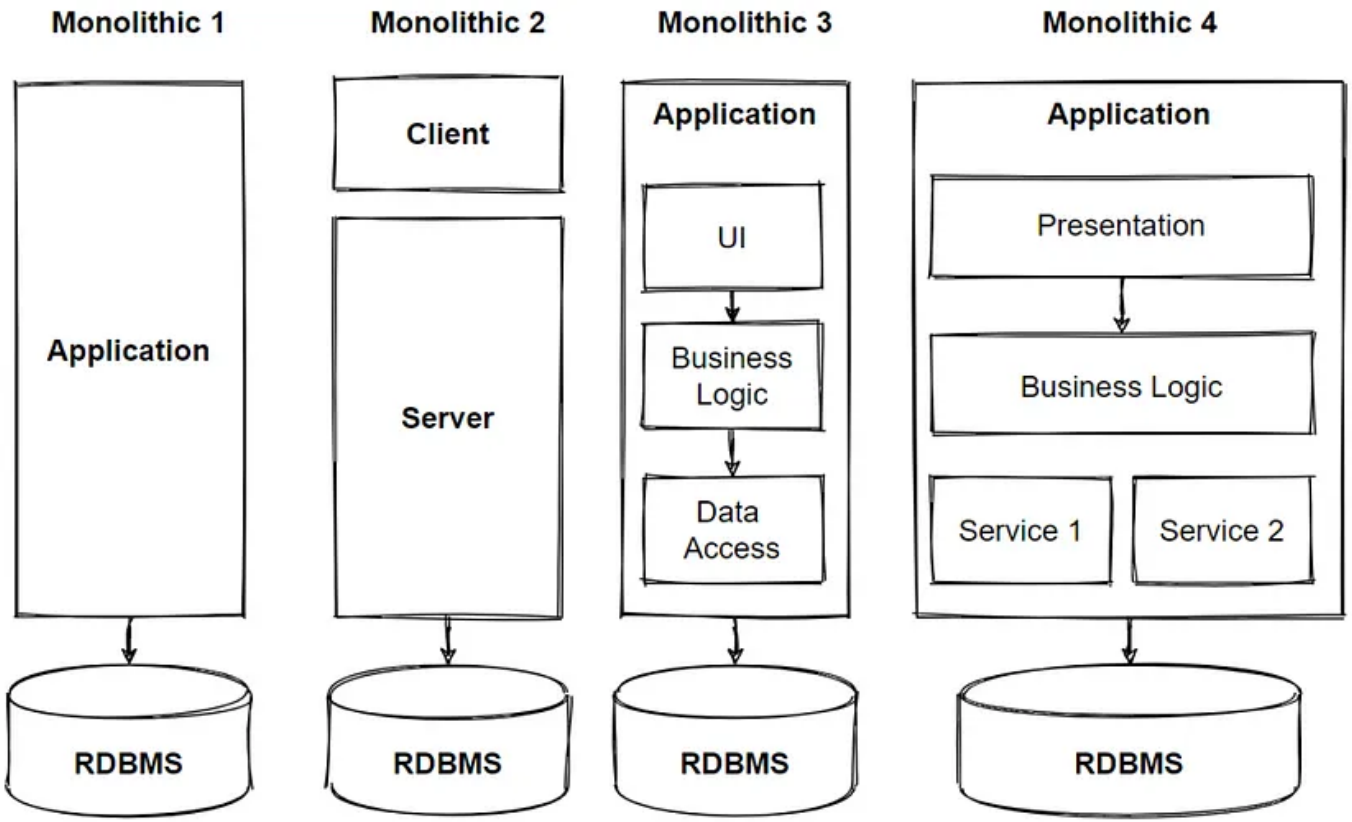
And we will design our e-commerce architecture as per these expected volumes.

After that, when our business grows, it will require more resources to accommodate more requests, and we will see how we can evolve our architecture as per these numbers.

Monolithic Architecture

Many approaches and patterns evolved over decades of software development, and all have benefits and challenges.

So, we will start understanding existing approaches to architecting our e-commerce application, evolving, and shifting it to the cloud.
To understand cloud-native microservices, we need to understand what monolithic applications are and how they led us to move from monolithic to microservices.



Monolithic Architecture

When it comes to legacy applications, we can say that most legacy applications were implemented as a monolithic architecture.

If all project functionalities exist in a single codebase, then that application is known as monolithic. The monolith pattern includes everything from the user interface, business codes, and database calls in the same codebase.

All application artifacts are contained in a single huge deployment.
Even the monolithic applications can design in different layers like presentation, business, and data layers and then deploy that codebase as a single jar/war file.

There are several advantages to the monolith approach that we will discuss upcoming. But let me detail some main advantages and disadvantages:

- Since it is a single code base, it’s easy to pull and start a project.
- Since this project structure is contained in one project and easy to debug interactions across different modules.
- With fewer moving parts, it´s less complex to maintain and troubleshoot.

Unfortunately, monolith architecture has lots of disadvantages, we can list some of them:

- It becomes too large in code size with time; managing is challenging.
- Difficult to work in parallel in the same code base.
- Hard to implement new features on legacy big monolithic applications
- Any change requires deploying a new version of the entire application, and so on.

As you can see, we better understand Monolithic Architecture pros and cons.

When to use Monolithic Architecture

Monolithic architecture has many disadvantages, but if you are building a small application, monolithic architecture is one of the best architectures you can apply to your project. Because, in many ways, monolithic apps are straightforward.

Monolithic architecture provides simplicity for:

- Build

- Test
- Deploy
- Troubleshoot
- Scale vertically (scale up)

It is simple to develop compared to microservices, where skilled developers are required to identify and develop the services. It is easier to deploy as only a single jar/war file can be used to deploy the complete application.

Design Monolithic Architecture

In this section, we will design our e-commerce application with the monolithic architecture step by step. We will iterate the architecture design one by one as per requirements.

We should always start with writing down FRs (Functional Requirements) and NFRs (Non-Functional Requirements).

Functional Requirements

- List products
- Filter products as per brand and categories

- Put products into the shopping cart
- Apply coupons for discounts and see the total cost for all the items in the shopping cart
- Checkout the shopping cart and create an order
- List my old orders and order items history

Non-Functional Requirements

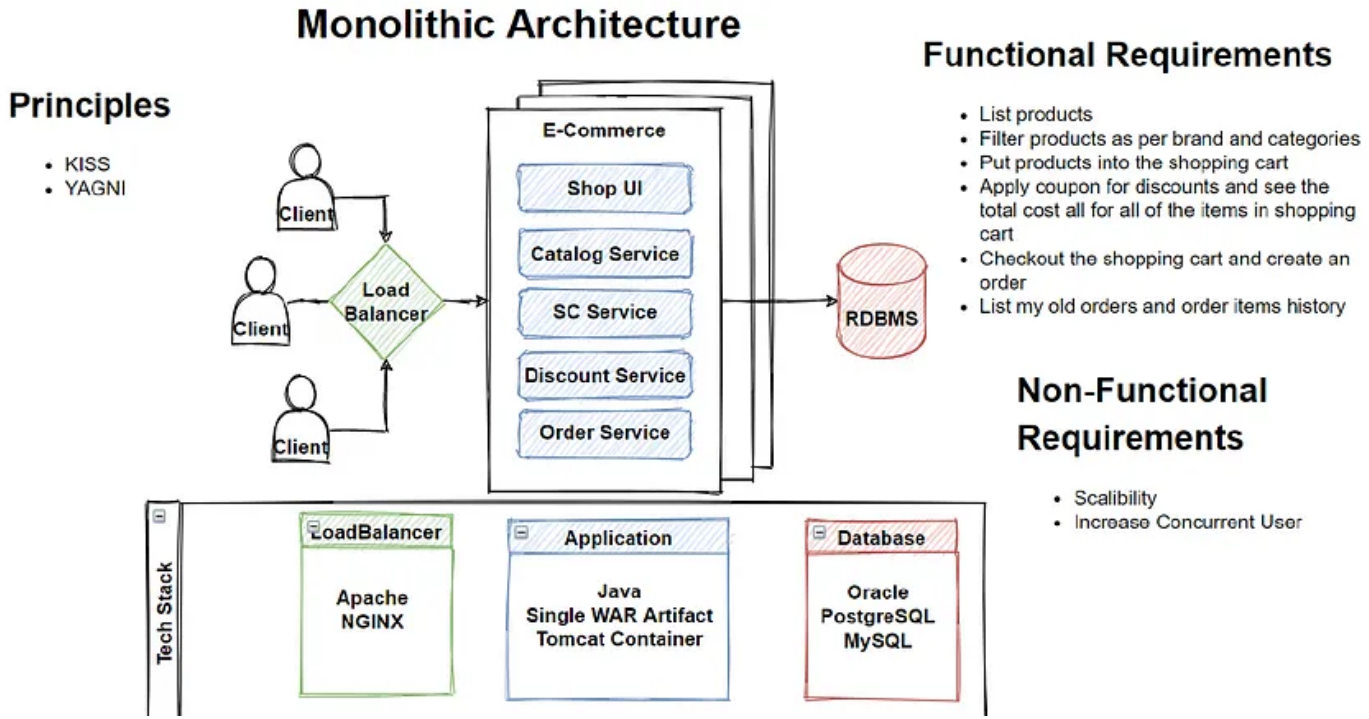
- Scalability
- Acceptable performance due to increase of concurrent users

Also, it's good to add principles to our picture always to remember them.

Principles

- KISS — Keep It Simple, Stupid
- YAGNI — You Ain't Gonna Need It

We are going to consider these principles when designing our architecture.



Design Monolithic Architecture

As you can see, we have designed our e-commerce application with Monolithic Architecture.

We have added the big e-commerce box; those are the components of our e-commerce application:

- Shopping UI
- Catalog Service
- Shopping Cart Service

- Discount Service
- Order Service

As you can see, all modules of this traditional web application are single artifacts in the container.

This monolithic application has a massive codebase that includes all modules.

If you introduce new modules to this application, you have to make changes to the existing code and then deploy the artifact with a different code to the Application Server, for example, the Tomcat server. We followed our KISS principle, which is to keep it simple.

And we will refactor our design as per requirements and, step by step, iterate together.

Scalability of Monolithic Architecture

As you can see from the following picture, we have scaled the Monolithic Architecture with horizontal scaling by adding two more application servers and putting a Load Balancer in front of the monolithic application between the client and the e-commerce application.

To provide scalability on Monolithic architecture, we need to increase the number of instances of the e-commerce application server and provision a new load balancer in front of our application.

Load Balancer will accommodate and send requests to our e-commerce application servers using consistent distribution algorithms. This capability will provide to load equally for the servers.

Adapting Technology Stack

We are going to make some technology choices — Adapting Technology Stack.

As you can see from the image, we have picked the potential options for our monolithic e-commerce application. NGINX is an excellent option for Load Balancing, and also Oracle Java is the standard implementation of this kind of application.

Microservices Architecture

Microservices are small business services that can work together and can be deployed autonomously / independently.

> *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.*

So, we can say that **Microservices Architecture** is a cloud-native architectural approach in which applications are composed of many loosely coupled and independently deployable smaller components. Let´s describe some **microservices** characteristics:

- They have their technology stack, including the database and data management model.
- Communicate with each other over REST APIs, event streaming, and message brokers.
- They are organized by business capability, with the line separating services, often referred to as a bounded context.

- We will also see how we can decouple microservices with a bounded context in the upcoming sections.

Microservices Characteristics

Microservices are small, independent, and loosely coupled. A single small team of developers can write and maintain a service. Each service is a separate codebase, which a small development team can manage.

Those services can be deployed independently. A team can update an existing service without rebuilding and redeploying the entire application.

Each service is responsible for persisting its data or external state. This capability differs from the traditional model, where a separate data layer handles data persistence.

Benefits of Microservices Architecture

Agility — One of the essential characteristics of microservices is that the services are smaller and independently deployable.

Small, focused teams — A microservice should be small enough for a single feature team to build, test, and deploy.

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

Challenges of Microservices Architecture

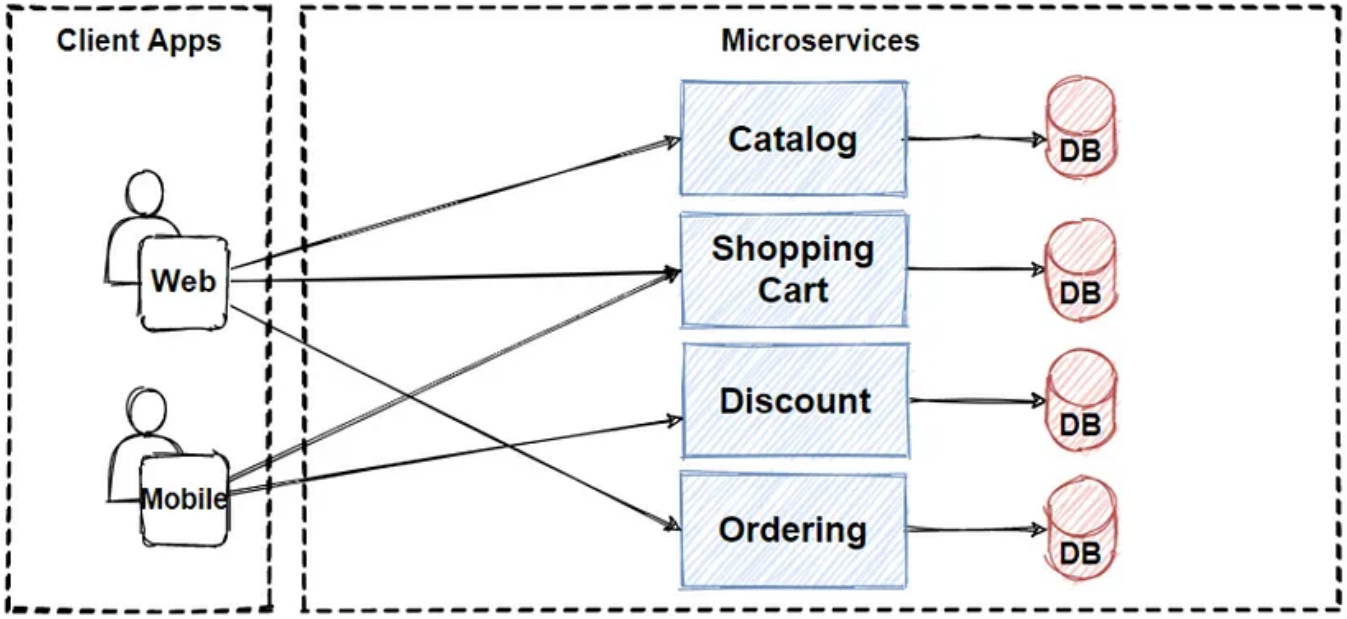
Complexity — The Microservices application has many services that need to work together and create value. Since there are many services, there are more moving parts than the monolithic application.

Network problems and latency — Since microservices are small and communicate with inter-service communication, we should manage network problems.

Data integrity — Each microservice has its own data persistence. So, data consistency can be a challenge.

Design Microservice Architecture

In this section, we will design the Microservice architecture step by step. Iterate the architecture design one by one as per requirements.



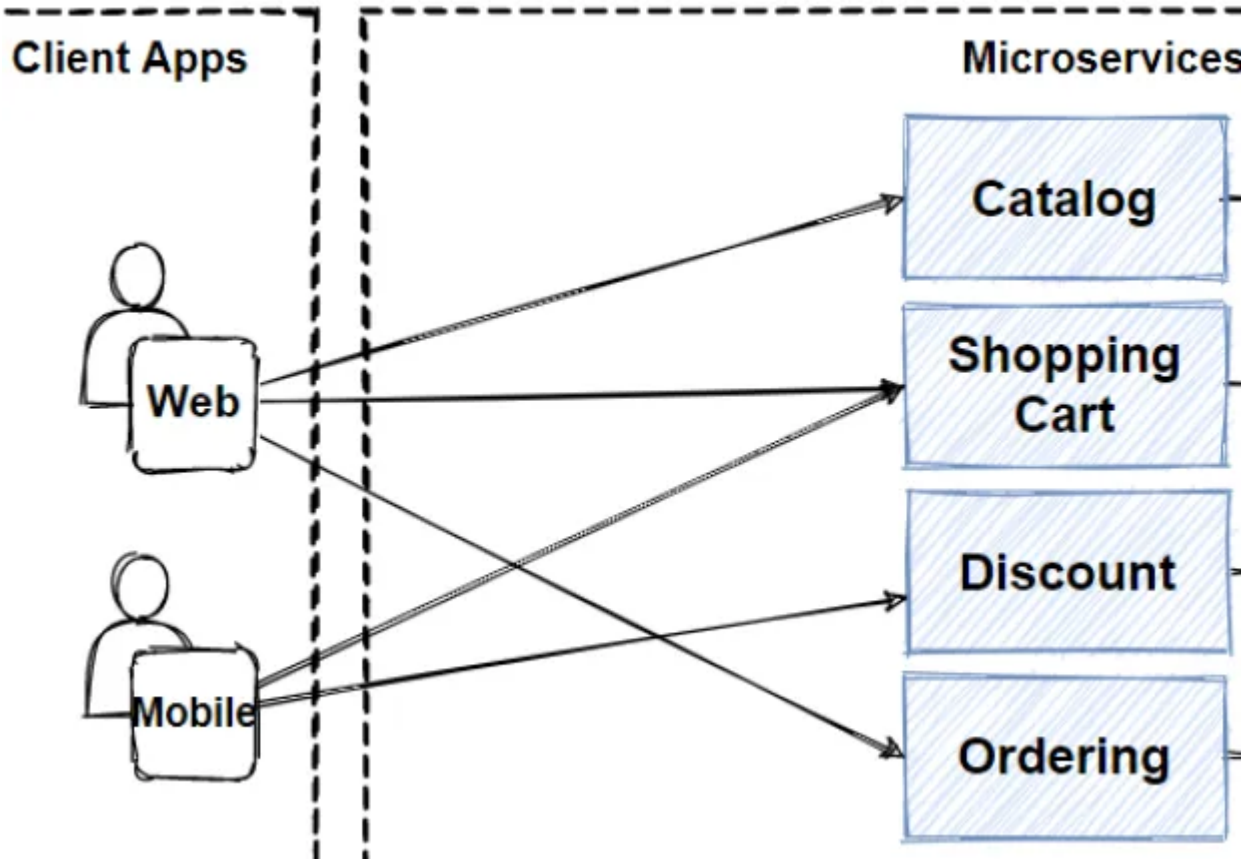
Design Microservice Architecture

We have followed the **Database-per-Services Pattern** when designing microservices architecture and put a database of each microservices. And microservices are decomposed from monolithic application modules with separated standalone services.

So now, these databases can be polyglot persistence. That means Product microservice can use the NoSQL document database, Shopping Cart microservice can use NoSQL key-value pair database, and Order microservice can use the Relational database following each microservices data storage requirements.

First Architectural Evolution

Let's look at the microservices architecture picture and consider what is missing from this architecture? What is the pain point of this architecture? How can we evolve this architecture to a better one to be more scalability, available, and able to accommodate more concurrent requests?



Frontend and Backend tightly-coupled

See that Front end UI elements and Microservices communication are directly, and it seems complicated to manage all those connection points. Now we should focus on Microservices communications!

Microservices Communications

Changing the communication mechanism is one of the biggest challenges when moving to a microservices-based application.

By nature, microservices are distributed; microservices communicate with each other by inter-service communication on the network level. Each microservice has its instance and process.

Therefore, services must interact using an inter-service communication protocol like HTTP, gRPC, or message brokers using the AMQP protocol.

Since microservices are complex structures into independently developed and deployed services, we should consider communication types and manage them into design phases.

Microservices Communication Design Patterns — API Gateway Pattern

The API gateway pattern is recommended if you want to design and build complex large microservices-based applications with multiple client

applications.

The pattern provides a reverse proxy to redirect or route requests to your internal microservices endpoints. An API Gateway provides a single endpoint for the client applications and internally maps the requests to internal microservices. We should use API Gateways between client and internal microservices.

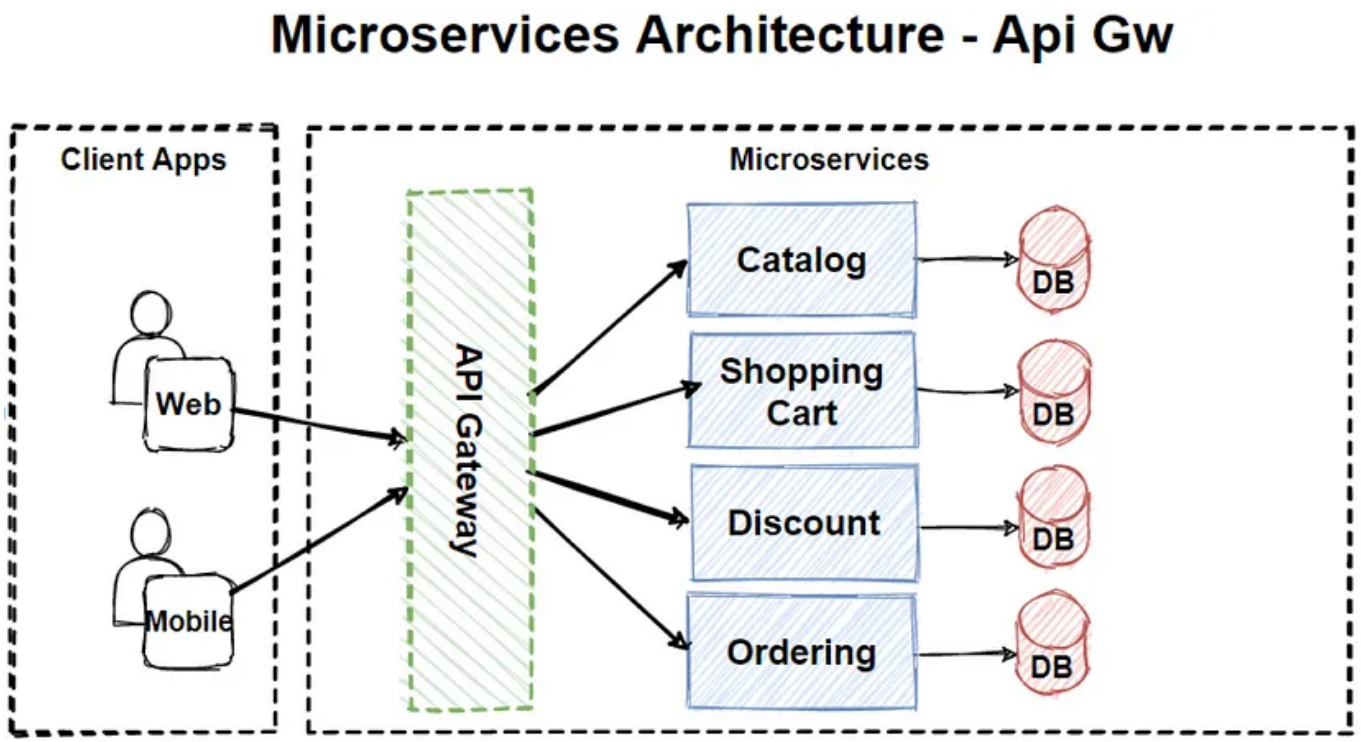
API Gateways can handle generic technical concerns like authorization, so instead of writing the same functionality in every microservices, authorization can be handled in a centralized way using the API Gateways and sent to internal microservices.

Also, the API Gateway manages routing to internal microservices and can aggregate several microservice requests in one response to the client.

In summary, the API Gateway will be placed between the client apps and the internal microservices working as a reverse proxy and routing client requests to backend services. It also provides generic technical concerns like authentication, SSL termination, and cache.

Design API Gateway — Microservices Communications Design Patterns

We will iterate our e-commerce architecture by adding an API Gateway pattern.



Using API Gateway

You can see the image that collects client requests in a single entry-point and routes requests to internal microservices.

This capability will handle the client requests and route the internal microservices, aggregate multiple internal microservices into a single client

request, and perform cross-cutting concerns like authentication and authorization, rate limiting and throttling, and so on.

Second Architectural Evolution

We will continue to evolve our architecture, but please look at the current design and consider how we can improve the design?

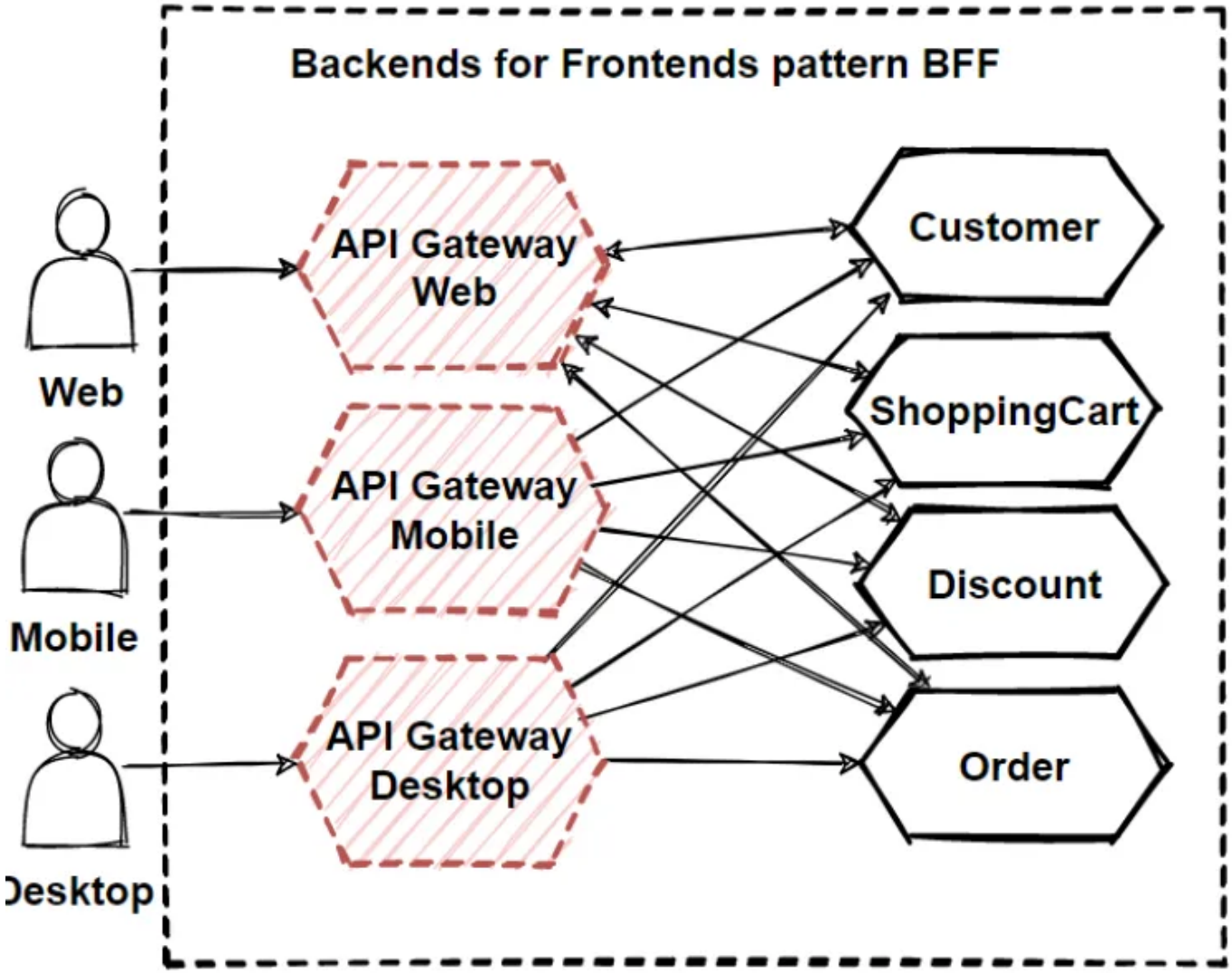
There are several client applications connected to a single API Gateway here. We should be careful about this situation because if we put a single API gateway here, it's possible to include risk associated with a single-point-of-failure.

If these client applications increase or add more logic to business complexity in API Gateway, it would be an anti-pattern. So, we should solve this problem with the Backends for Frontends pattern (BFF).

Backends for Frontends pattern BFF — Microservices Communications Design patterns

Backends for Frontends pattern can separate API Gateways per the specific frontend applications. So, we have several backend services consumed by frontend applications, and between them, we include the API Gateway for handling, routing, and aggregation operations.

But this makes a single point of failure. To solve this problem, BFF opens the possibility of creating several API Gateways, grouping the client applications according to their boundaries, and splitting them into different API Gateways.



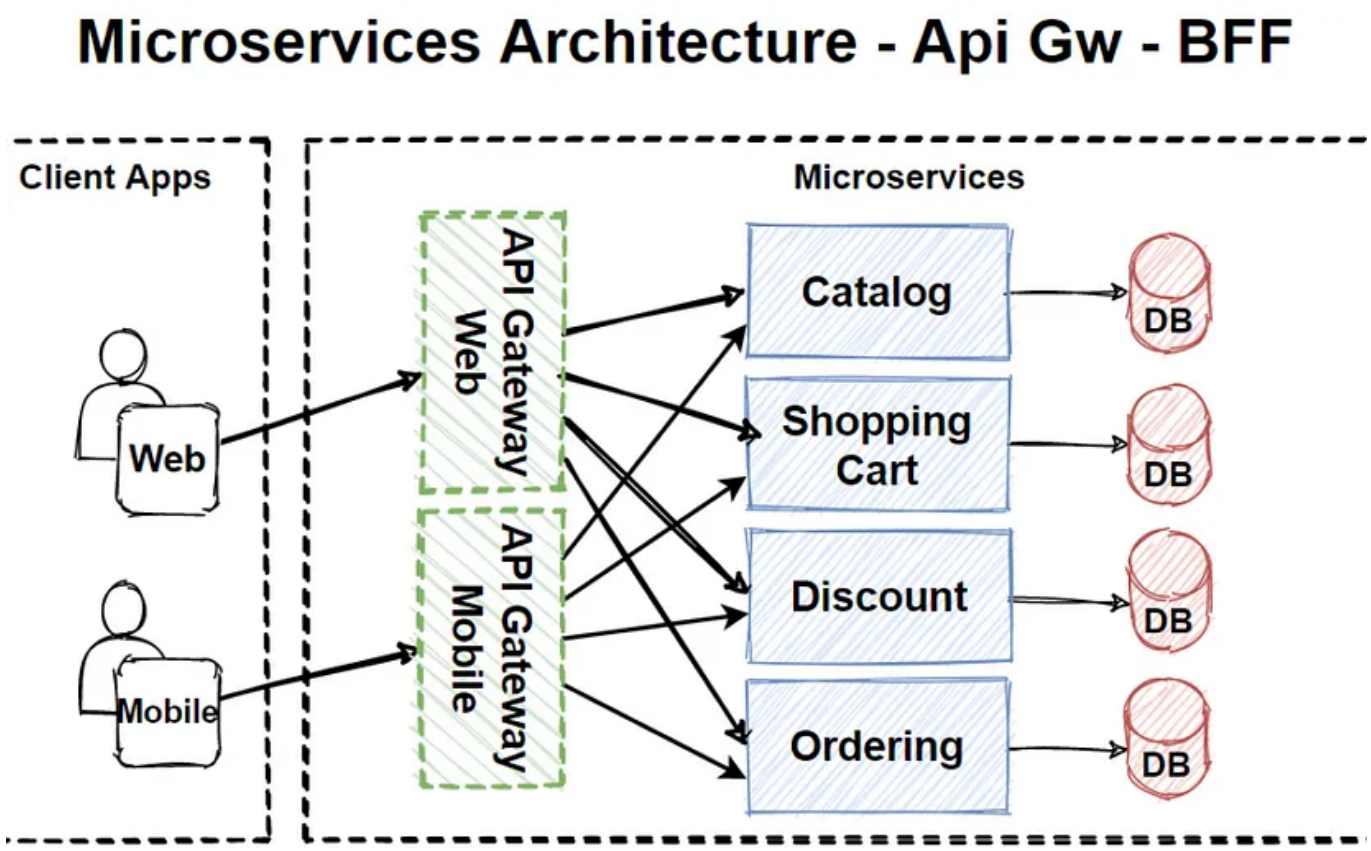
BFF Pattern

A single and complex API Gateway can be risky and become a bottleneck in our architecture. Larger systems often expose multiple API Gateways by grouping client types by functionality, like mobile, web, and desktop. BFF pattern is useful when you avoid customizing a single backend for multiple interfaces.

So, we should create several API Gateways as per user interfaces. These API Gateways best match the needs of the frontend environment without worrying about affecting other frontend applications. The Backend for Frontends pattern provides direction for implementing multiple gateways.

Design Backends for Frontends pattern BFF — Microservices Communications Design Patterns

We will iterate our e-commerce architecture by adding more API Gateway patterns according to the Backends for Frontends pattern BFF.



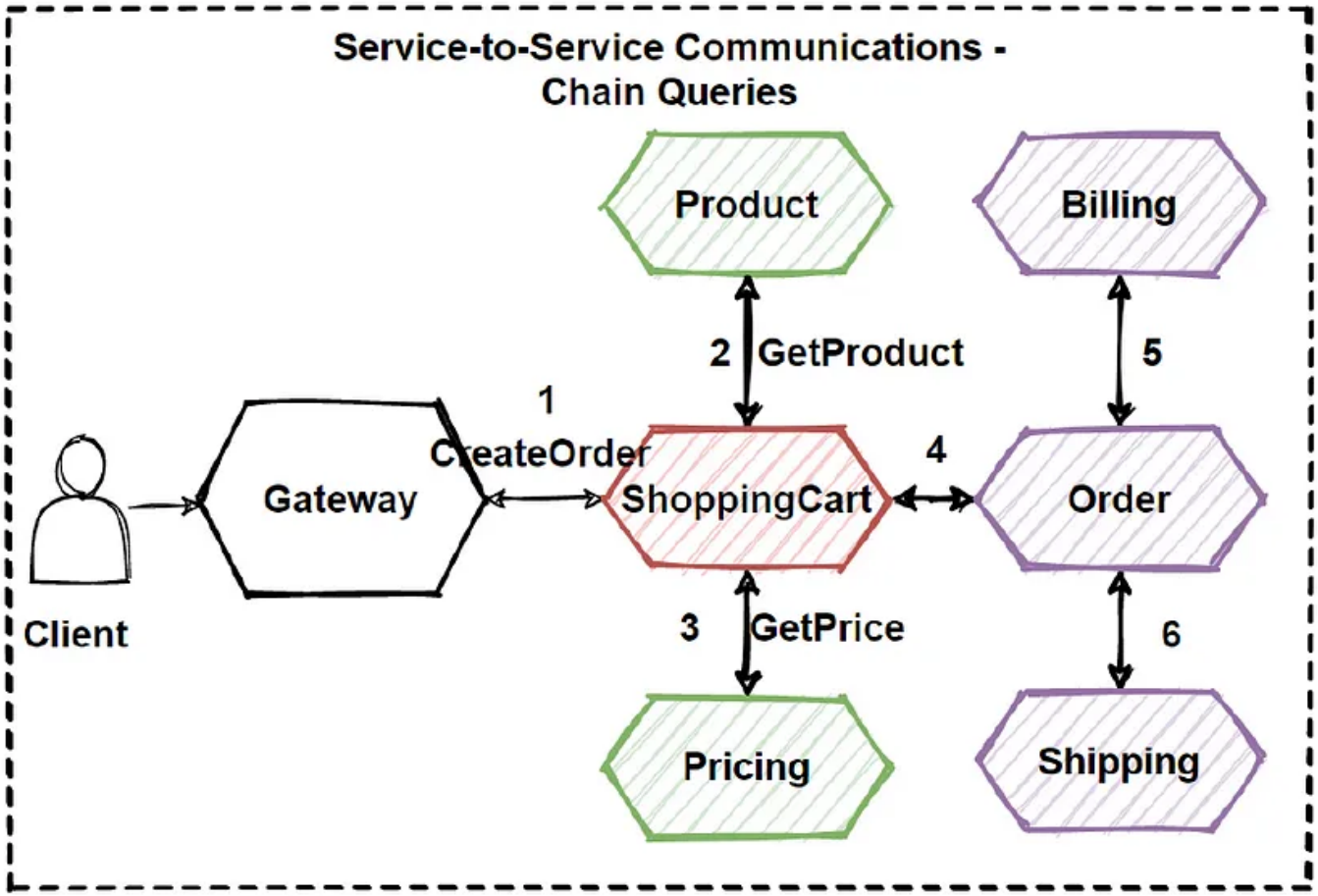
API Gateway and BFF patterns

As you can see, we have added several API Gateways to our application. These API Gateways best match the needs of the frontend environment without worrying about affecting other frontend applications. The Backend for Frontends pattern provides directions for implementing multiple gateways.

Service-to-Service Communications between Backend Internal Microservices — Microservices Communications Design patterns

OK, we have created API Gateways in our microservices architecture. All those sync requests come from the clients and go to internal microservices over the API Gateway.

But what if the client requests require more than one internal microservices? How can we manage internal microservice communications?



Service-to-Service Communications — Chain Queries

When designing microservices applications, we should be careful about how backend internal microservices communicate with each other. The best practice is to reduce inter-service communication as much as possible.

However, sometimes, we can't reduce these internal communications due to customer requirements or the requested operation need to call several internal services.

For example, look at the picture above and think about the use case like:

- The user checks out the shopping cart and creates an order

So how can we implement this request? These internal calls make coupling for each microservices; in our case, Shopping Cart, Product, and Pricing microservices depend on each other.

And if one of the microservices is not responding, it can't return data to the client, so it's not fault-tolerant. If the dependency and coupling of microservices are increased, it creates lots of problems, and we cannot deliver the microservices architecture's full potential.

If the client checks out the shopping cart, this will start a set of operations. So, if we try to perform this place order use case with the Request/Response

sync Messaging pattern, it will seem like this picture.

As we can see, we have six sync HTTP requests for one client HTTP request. So, it is evident that it increases latency and negatively impacts our system's performance, scalability, and availability.

If we have this use case in place, what happens if **step 5 or 6** fails or some middle services are down? Even if they are not down, it could be busy, and some services can't get a response on time, and in that case, high latencies are not acceptable.

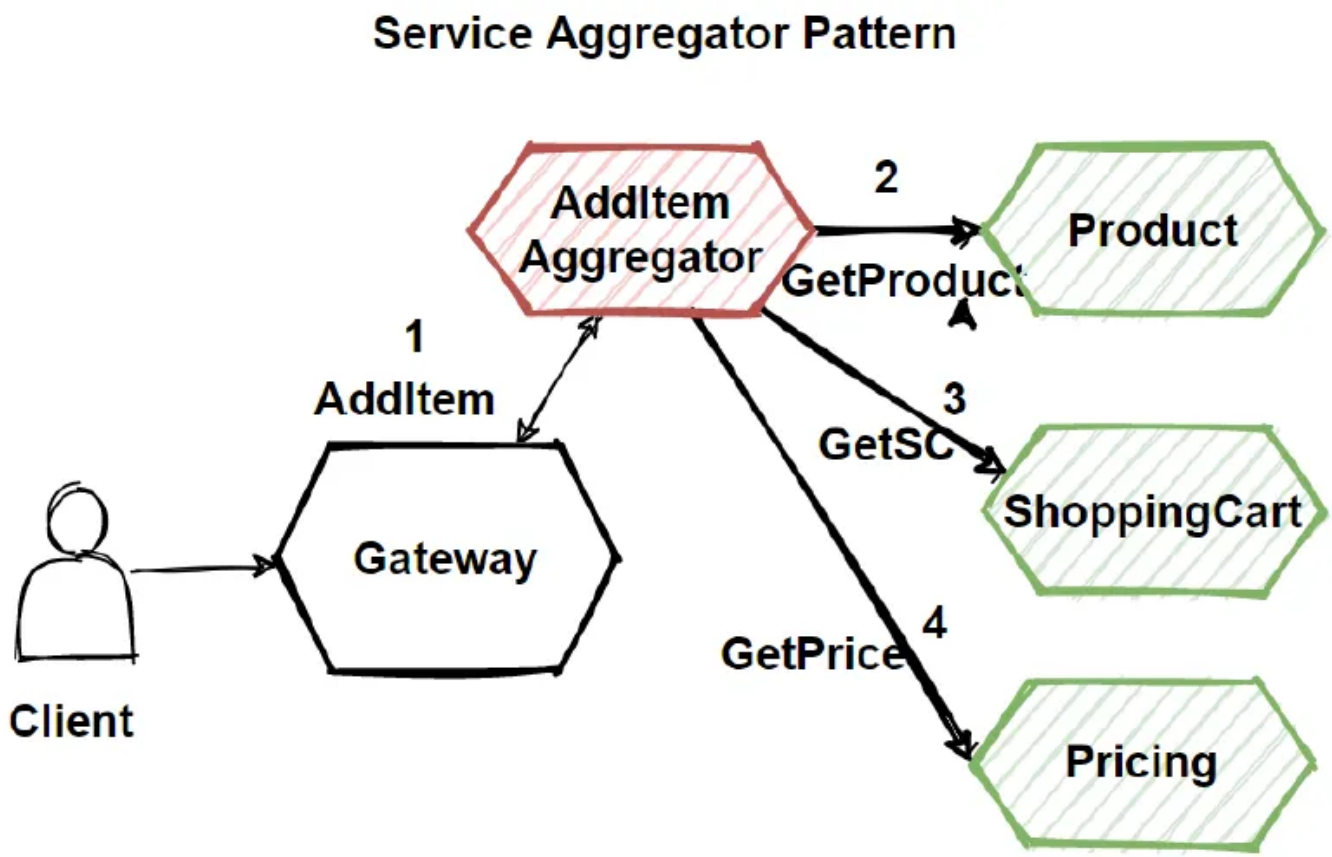
So, what could be the solution to this kind of requirement?

We can apply two ways to solve these issues:

1. Change microservices communications to async with message broker systems; we will see this in the next section.
2. Use Service Aggregator Pattern to aggregate some query operations in one API Gateway.

Service Aggregator Pattern — Microservices Communications Design patterns

To minimize service-to-service communications, we can apply the **Service Aggregator Pattern.** The Service Aggregator Design Pattern receives a request from the client or API Gateway, dispatches requests to multiple internal backend microservices, and then combines the results and responds to the initial requester in one response structure.

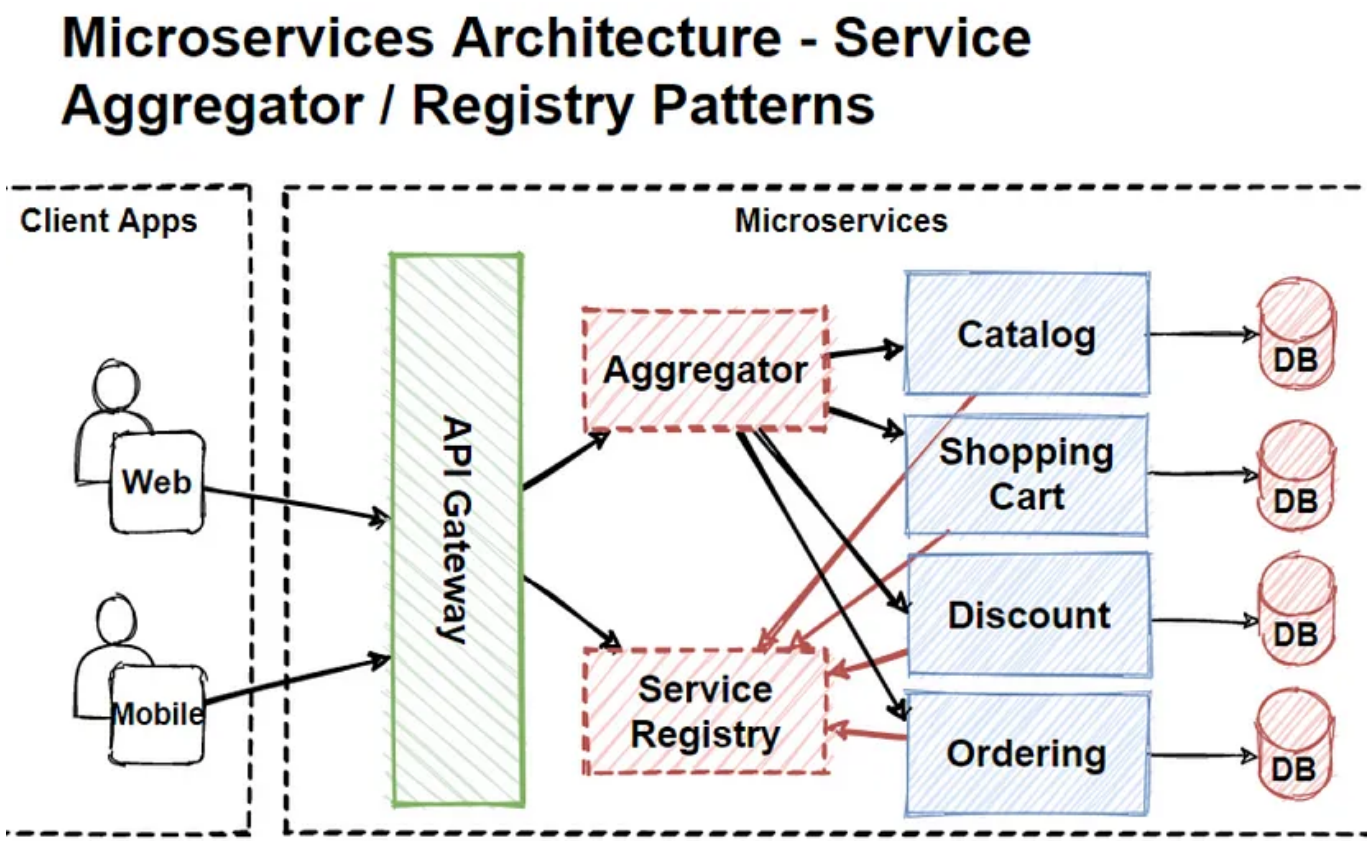


Service Aggregator Pattern

By Service Aggregator Pattern implementation, we can reduce chattiness and communication overhead between the client and microservices.

DESIGN — Service Aggregator Pattern — Service Registry Pattern — Microservices Communications Design patterns

This section will iterate our e-commerce architecture by adding Service Aggregator Pattern — Service Registry Pattern — Microservices Communications Design patterns.

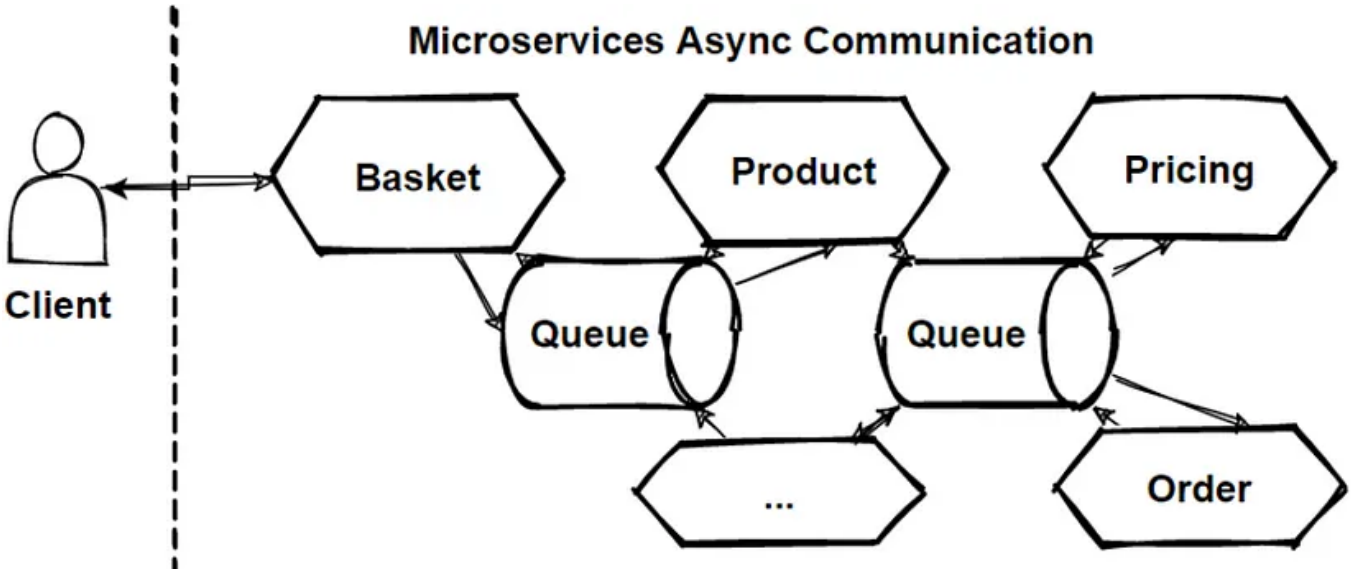


Aggregator and Service Registry Pattern

As you can see, we have applied Service Aggregator Pattern — Service Registry Pattern for our e-commerce architecture.

Microservices Asynchronous Message-Based Communication

Synchronous communication is suitable if your communication is only between a few microservices. But we should use async communication when it comes to several microservices needing to call each other and could wait for long or complex operations until they are finished.



Microservices Async Communication

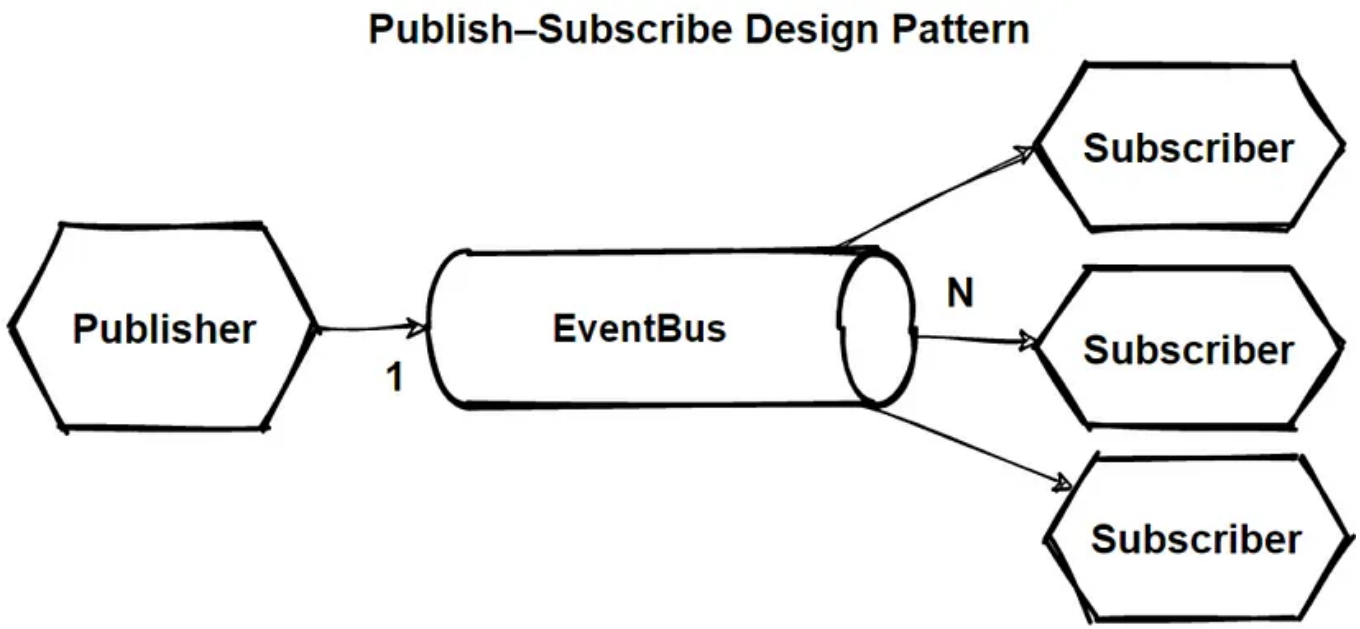
Otherwise, the dependency and coupling of microservices will create bottlenecks and serious architecture problems.

If you have multiple microservices that are required to interact with each other and if you want to interact with them without any dependency or make loosely coupled, then we should use asynchronous message-based communication in our Microservices Architecture.

Because asynchronous message-based communication works with events, events can be a form of communication between microservices. We call this communication **event-driven communication**.

Publish–Subscribe Design Pattern

Publish-Subscribe is a messaging pattern that has a sender of messages which are called publishers, and specific receivers, which are called subscribers.



Publish-Subscribe Design Pattern

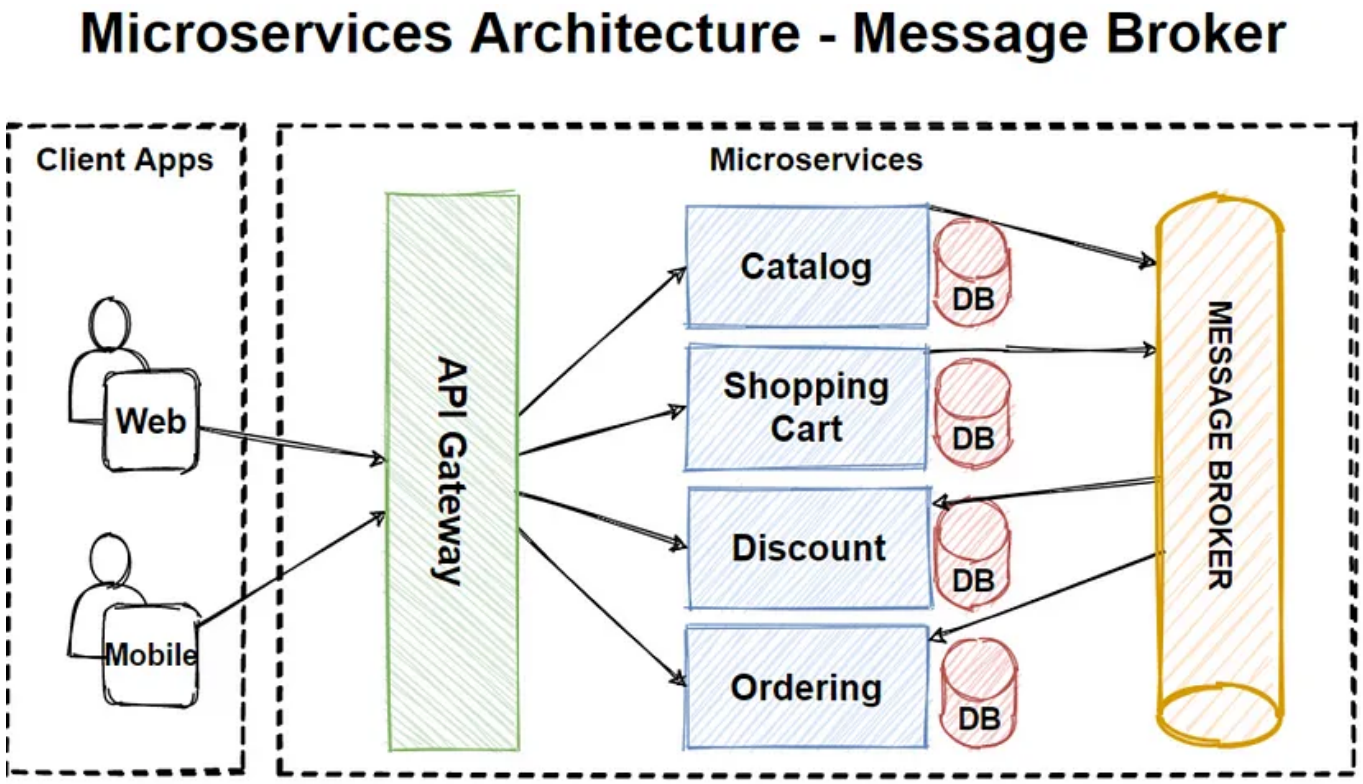
So, publishers don't send messages directly to the subscribers. Instead, each service categorizes and publishes messages into message broker systems without knowing which subscribers are there.

Similarly, subscribers express interest and only receive messages that are of interest without knowing which publishers send to them.

DESIGN — Pub/Sub Message Broker — Microservices Asynchronous Communications Design patterns

In this section, we will iterate our e-commerce architecture by adding a Publish-Subscribe Message Broker for providing Microservices

Asynchronous Communications Design.



Message Broker

As you can see, we have applied Publish-Subscribe Message Broker — Microservices Asynchronous Communications Design patterns.

If we adapt our technology stack, we could start thinking about of what options can be for used for Publish-Subscribe Message Brokers functionality. There are two good alternatives that you can choose from:

1. Kafka
2. RabbitMQ

Microservices Data Management

In monolithic architectures, it's acceptable to query different entities because a single database keeps data. Querying data across multiple tables is straightforward; any unwanted changes in data can easily roll back. Relational databases with strict consistency have an ACID (atomicity, consistency, isolation, and durability) transaction guarantee, so it's easy to manage and query data.

But in microservices architectures, we normally use a **polyglot persistence.** This means that each microservices has different repositories, both relational and NoSQL databases. We should set a strategy to manage this data when performing user interactions.

So that means we have several patterns and practices when handling data interactions between microservices; we will learn these patterns and principles in this section.

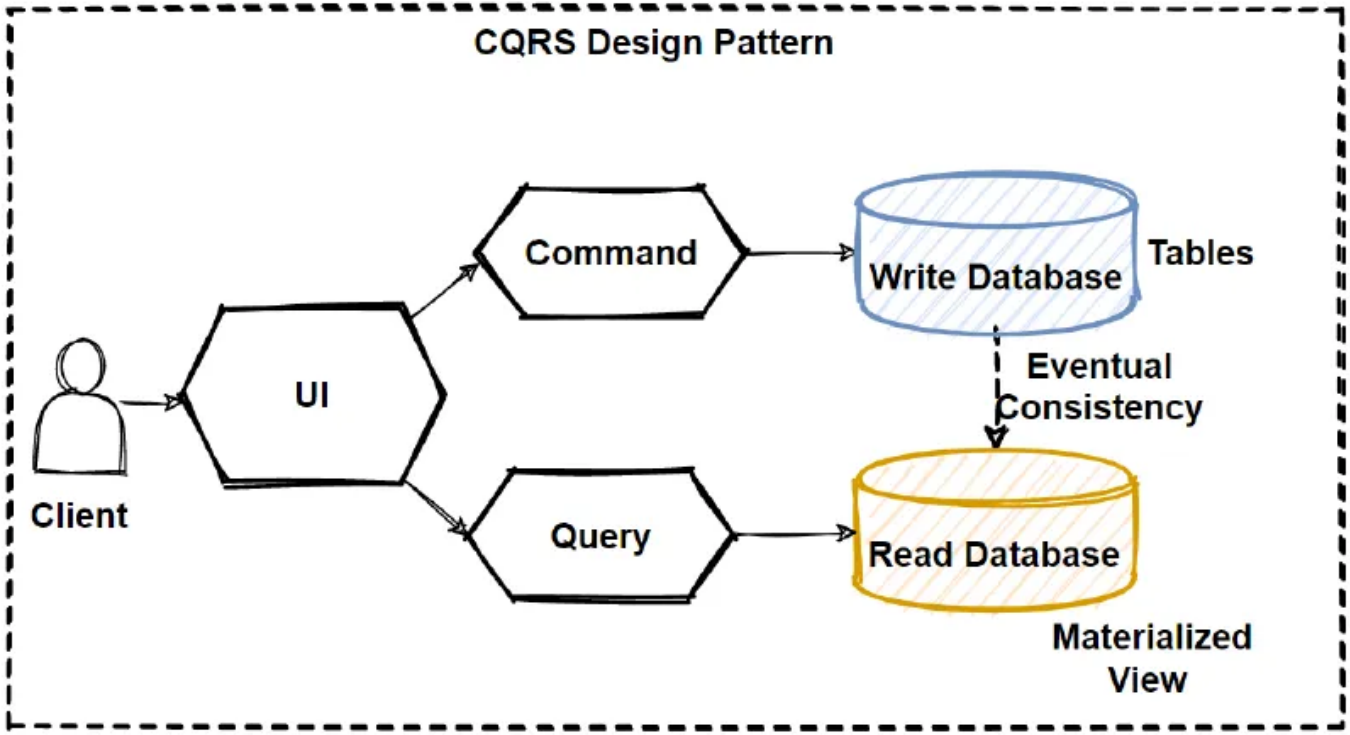
Microservices are independent and perform only specific functional requirements. In our case, an e-commerce application that has a product, basket, discount, and ordering microservices. Those microservices need to interact with each other to perform our customer use cases.

So that means they need to integrate frequently with each other. And mainly, these integrations are querying each other service's data for aggregation or performing logic.

CQRS Design Pattern

CQRS is one of the critical patterns when querying between microservices. We can use the CQRS design pattern to avoid complex queries and eliminate inefficient joins. CQRS stands for Command and Query Responsibility Segregation. This pattern separates read and update operations for a database.

To isolate Commands and Queries, it is best practice to separate the read and write database into two databases. In this way, if our application is read-intensive, that means reading more than writing, we can define a custom data schema optimized for queries.



CQRS Design Pattern

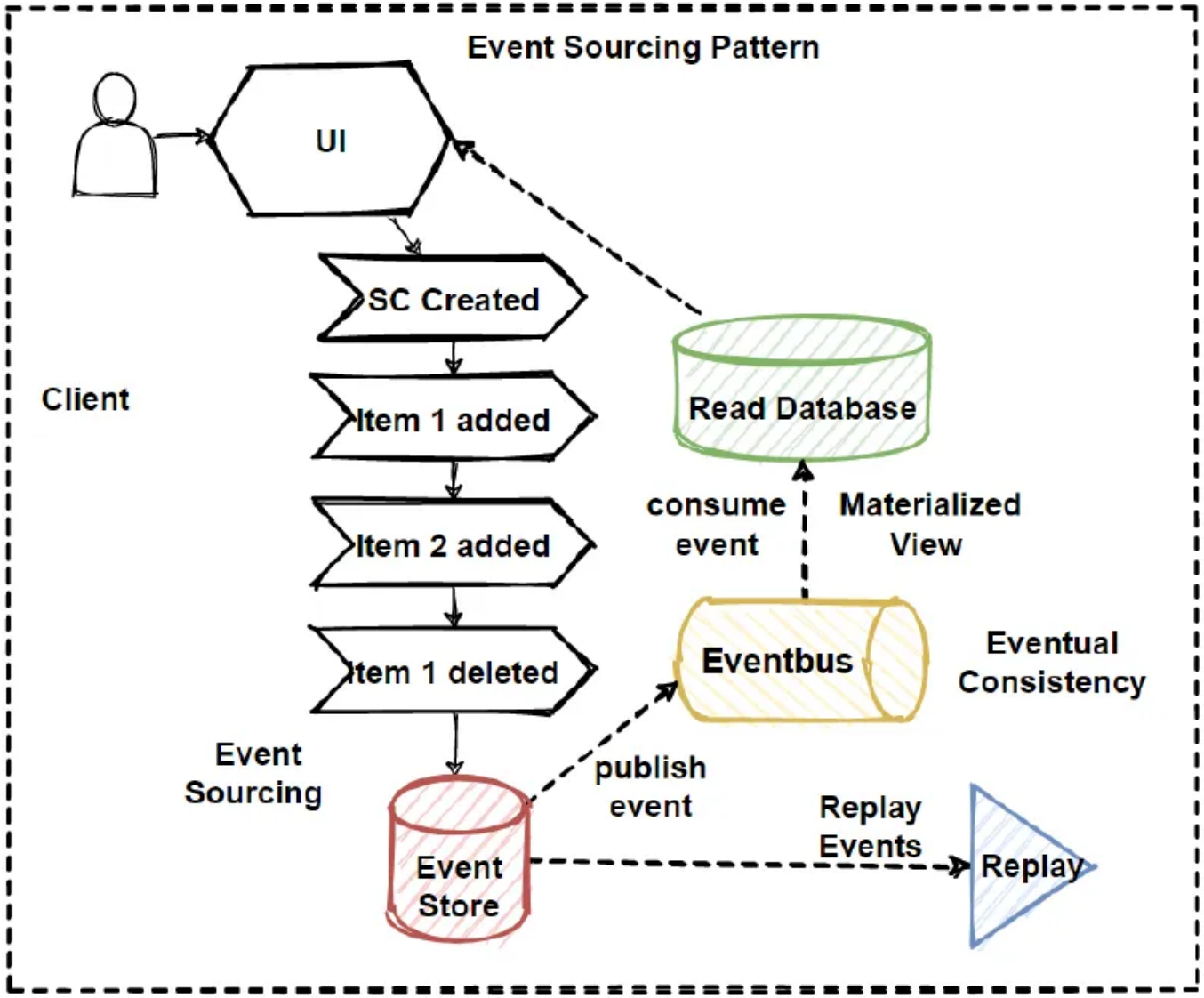
Materialized view pattern is an excellent example of implementing reading databases. Because by this way, we can avoid complex joins and mappings with pre-defined fine-grained data for query operations.

By this isolation, we can even use different databases for reading and writing databases like NoSQL document databases for reading and using relational databases for CRUD operations.

Event Sourcing Pattern

We have learned the CQRS pattern, and the CQRS pattern is mainly used with the Event Sourcing pattern. When using CQRS with the Event Sourcing pattern, the main idea is to store events in the write database, which will be the source-of-truth events database.

After that, the read database of the CQRS design pattern provides materialized views of the data with denormalized tables. Of course, these materialized views read database consume events from the write database and convert them into denormalized views.



Event Sourcing Pattern

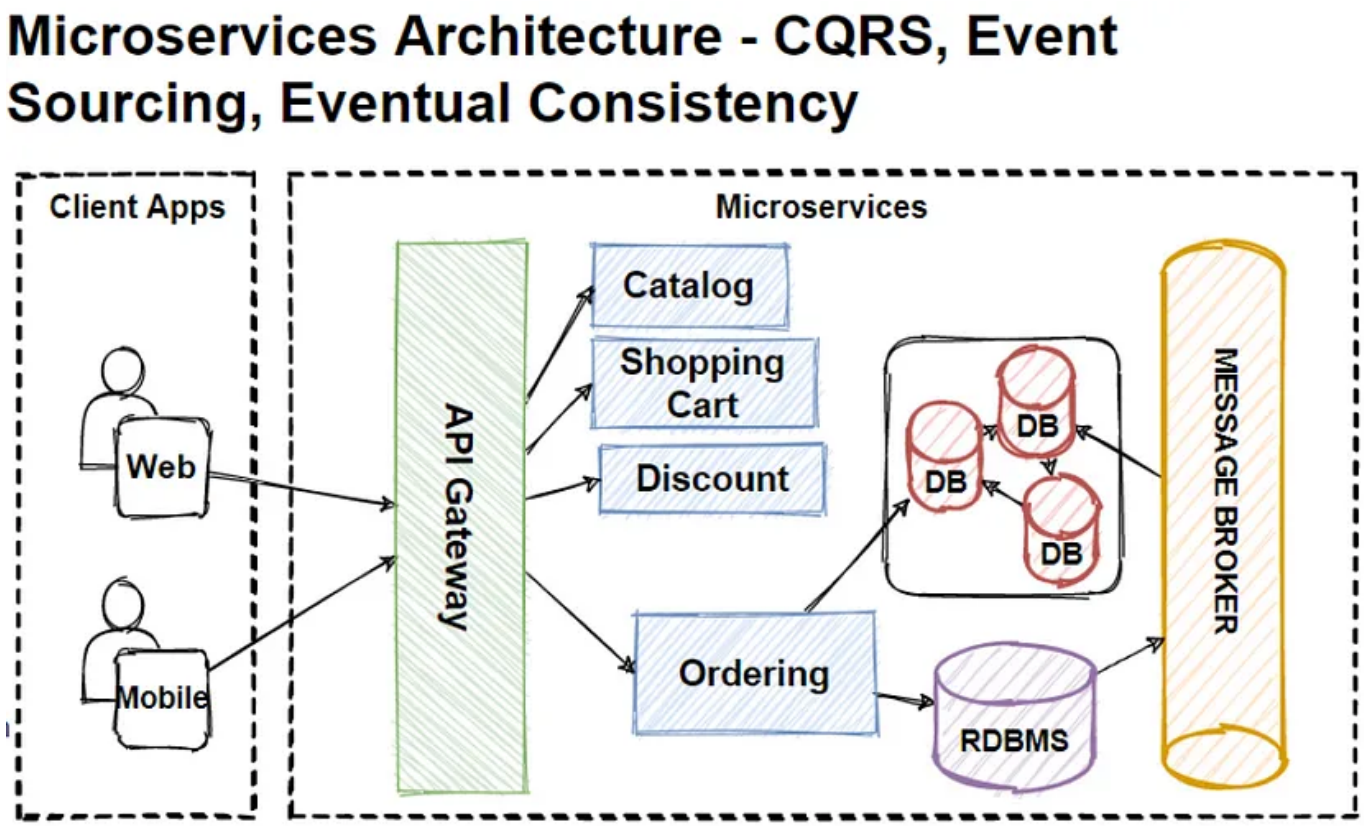
Applying the Event Sourcing pattern is changing data save operations into the database. Instead of keeping the latest data status in the database, the

Event Sourcing pattern offers to hold all events in the database in sequential order of data events. This events database is called an event store.

Instead of updating the status of a data record, it appends each change to a sequential list of events. So, The Event Store becomes the source of truth for the data. After that, these event stores convert to a read database. This conversion operation can be handled by publish/subscribe pattern with post-event message broker systems.

Design the Architecture — CQRS, Event Sourcing, Eventual Consistency, Materialized View

We will design our e-commerce architecture by applying CQRS, Event Sourcing, Eventual Consistency, and Materialized Views.



CQRS, Event Sourcing, and Eventual Consistency

So, when a user creates or updates an order, I will use a relational database. When the user queries a single order or order history, I will use NoSQL read database and make them consistent when syncing two databases by using a message broker system with a Publish/Subscribe pattern.

Now we have to consider the tech stack of these databases; let´s assume the usage of Microsoft SQL Server as the relational database for writing operations and Cassandra as the NoSQL database for reading operations. Of

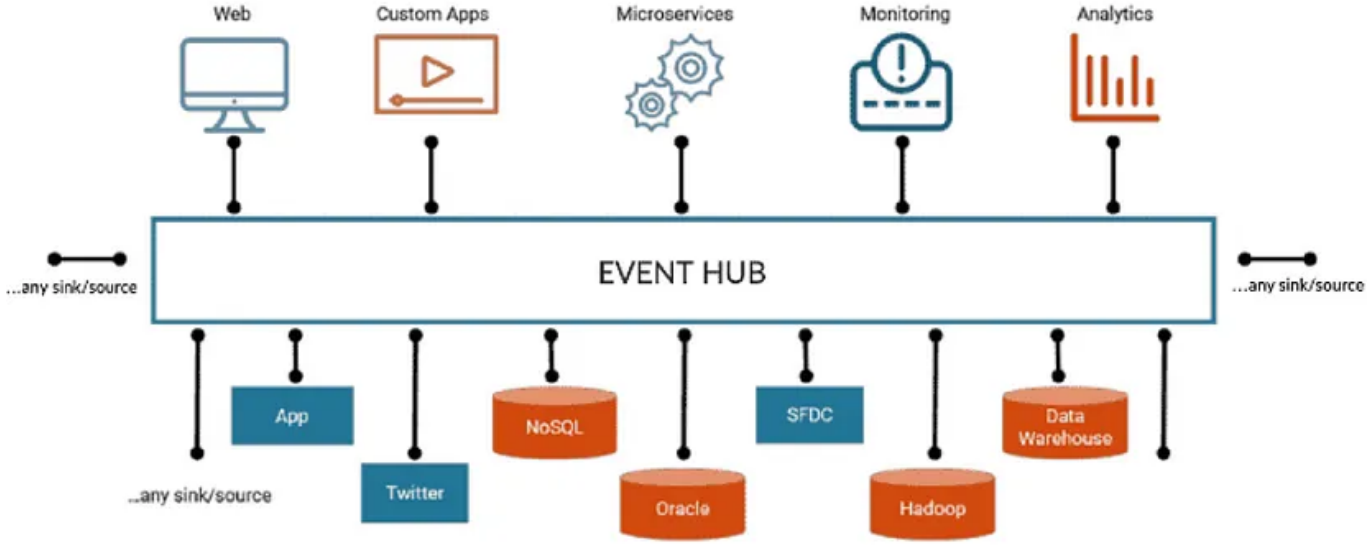
course, we will use Kafka for syncing these two databases with Publish/Subscribe Kafka topic exchanges.

As you can see, we have finished designing microservices database patterns. Let's deep dive into these event-driven architectures in microservices.

Event-Driven Microservices Architecture

Event-driven microservices architecture means communicating with microservices via event messages. And we saw that with Publish/Subscribe pattern and Kafka as a message broker system at microservices async communication sections.

We said we could do asynchronous behavior and loosely coupled structures with event-driven architectures. For example, services consume data via events instead of sending requests when data is needed. This capability will provide performance increases.



Event Hub

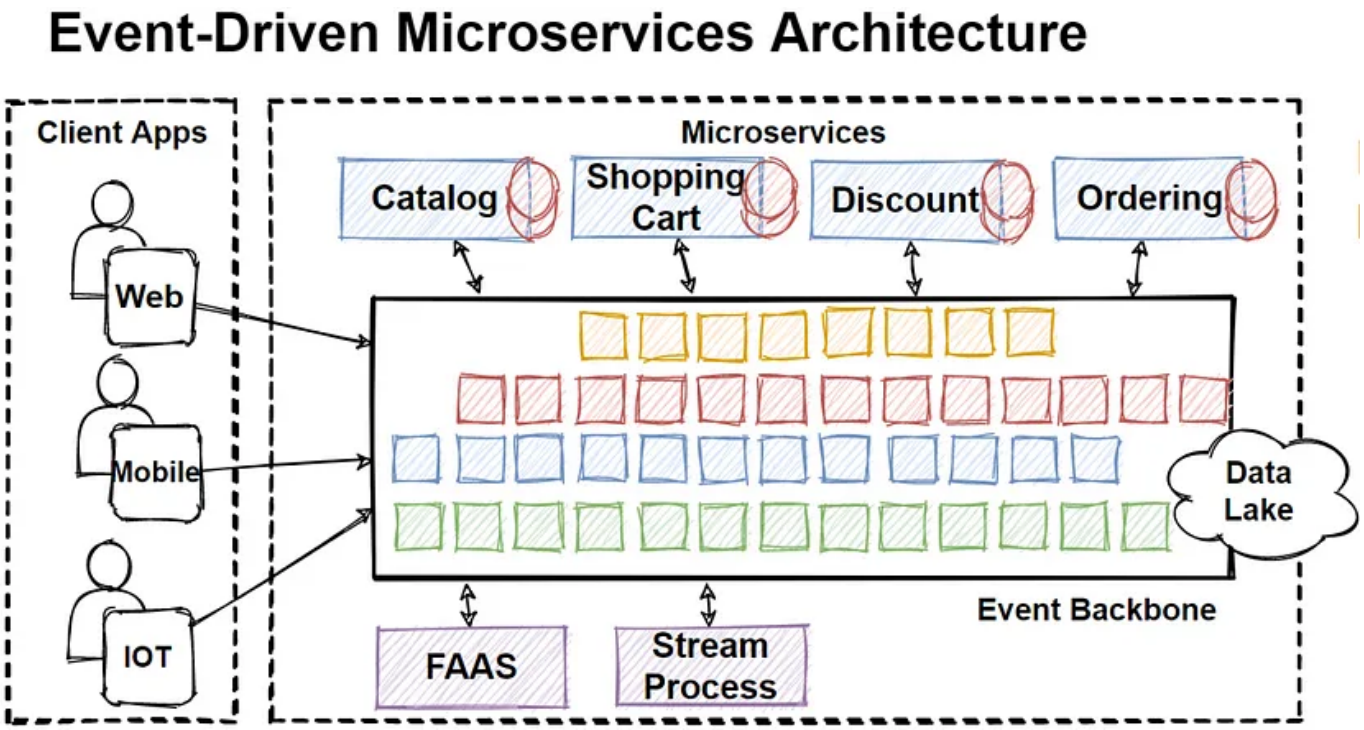
But also, there are many innovations in the Event-Driven Microservices Architectures like using real-time messaging platforms, stream-processing, event hubs, real-time processing, batch processing, data intelligence, and so on.

So, we can make this event-driven approach more generic and real-time event processing features by evolving this architecture.

According to this new event-driven microservices architecture, we can have everything connected via Event-Hubs. We can think of Event-Hubs as a large event store database that can provide real-time processing.

Design the Architecture — Event-Driven Microservices Architecture

We will design our e-commerce application with the Event-Driven Microservices Architecture.



Event-driven Microservices Architecture

Now we can decide to technology stack in this architecture. Of course, we should pick **Apache Kafka** — as a **Event hub** and **Apache Spark** for **real-time** and near-real-time **streaming** applications that transform or react to the streams of data.

As you can see that now we have **reactive design** with **Event-Driven Microservices Architecture.**

Now we can ask the same question;

- **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

With this latest event-driven microservices architecture which Deployments with Containers and Orchestrators, can be accommodate target concurrent request in a low latency. Because this architecture is fully loosely coupled and design for high scalability and high availability.

As you can see that we have designed our **e-commerce microservices architecture** with all aspects of **design principles** and **patterns.** Now, you can ready to design your own architectures with these learning and know how to use these patterns toolbox in your designs.

Step by Step Design Architectures w/ Course

I have just published a new course — Design Microservices Architecture with Patterns & Principles.

In this course, we're going to learn **how to Design Microservices Architecture** with using **Design Patterns, Principles** and the **Best Practices.** We will start with designing **Monolithic** to **Event-Driven Microservices** step by step and together using the right architecture design patterns and techniques.

> *A special thanks to **André Venâncio** for editing article and help with grammar and clarity.*

Microservices | Software Architecture | System Design Interview | Software Design | Architecture Pattern

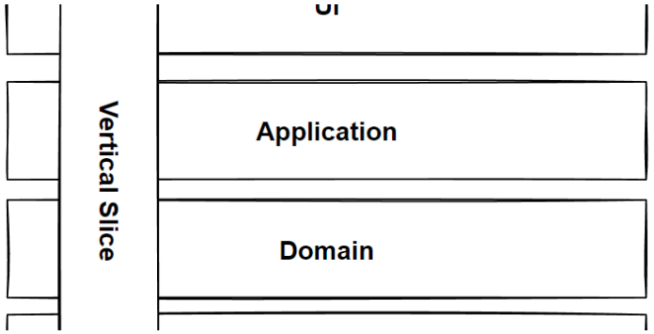
Written by Mehmet Ozkaya

Follow

6.1K Followers · Editor for Design Microservices Architecture with Patterns & Principles

Software Architect | Udemy Instructor | AWS Community Builder | Cloud-Native and Serverless Event-driven Microservices https://github.com/mehmetozkaya

More from Mehmet Ozkaya and Design Microservices Architecture with Patterns & Principles



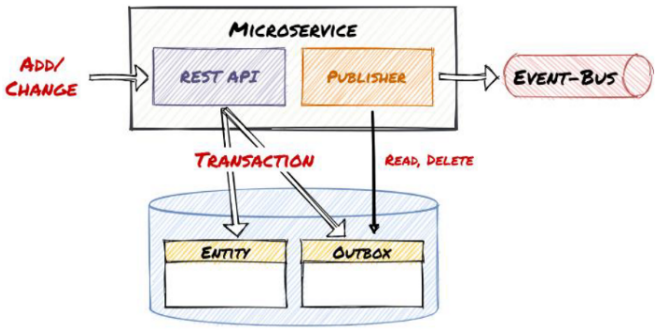
Mehmet... in Design Microservices Architecture w...

The Problem with Clean Architecture: Vertical Slices

In this article, we are going to learn the Problem with Clean Architecture which is...

6 min read · Feb 15

77 7



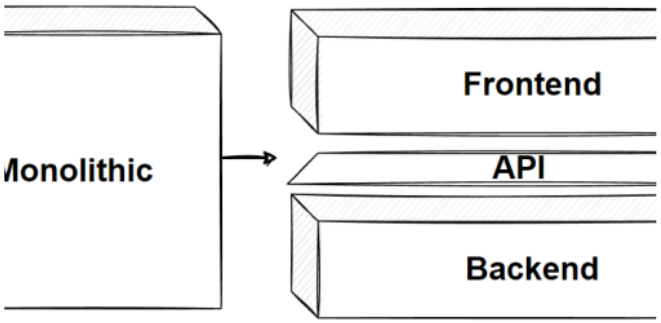
Mehmet... in Design Microservices Architecture w...

Outbox Pattern for Microservices Architectures

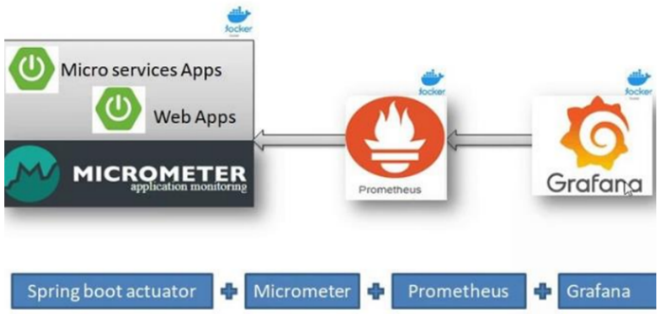
In this article, we are going to talk about Design Patterns of Microservices architectur...

3 min read · Sep 8, 2021

202 4



Mehmet... in Design Microservices Architecture w...



Mehmet Ozkaya

Headless Architecture with Separated UI for Backend and...

In this article, we are going to learn Headless Architecture and Best Practices when...

8 min read · Feb 17

98

Monitor Spring Boot Custom Metrics with Micrometer and...

In this article, we will explore the concepts around Spring Actuator, Micrometer and...

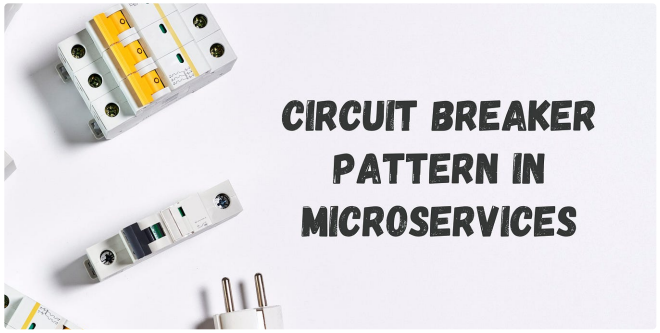
6 min read · Feb 14

23

See all from Mehmet Ozkaya

See all from Design Microservices Architecture with Patterns & Principles

Recommended from Medium



Chameera Dulanga in Bits and Pieces

Circuit Breaker Pattern in Microservices

How to Use the Circuit Breaker Software Design Pattern to Build Microservices

6 min read · Jan 11

390

Denat Hoxha

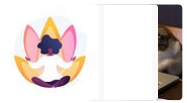
Sharing Data Between Microservices

Robust distributed systems embrace eventual consistency to share data between...

7 min read · Oct 24, 2022

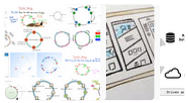
1.1K 25

Lists



Stories to Help You Grow as a Software Developer

19 stories · 269 saves



General Coding Knowledge

20 stories · 201 saves

Now in AI: Handpicked by Better Programming

262 stories · 85 saves

Jack Pritom Soren

Software Design Pattern: Bridge Pattern

Bridge is a structural design pattern that lets you split a large class or a set of closely...

3 min read · May 30

1

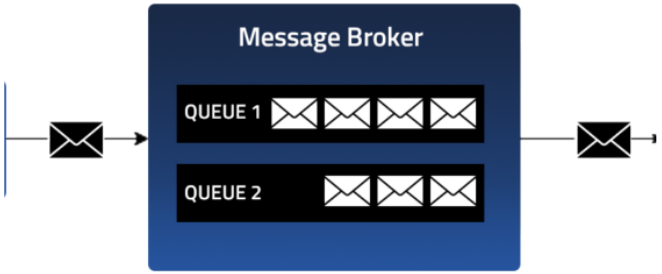
Muhammad Taha

Designing Modern Applications: Chapter 1—Genesis

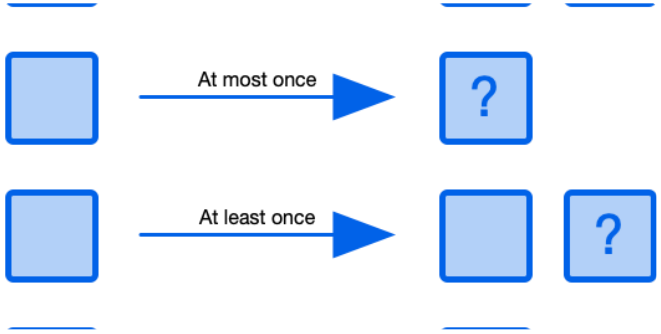
A guide on how to build software, from understanding requirements to architectural...

7 min read · Jul 6

21



David Mosyan



Andy Bryant

Processing guarantees in Kafka

Cloud Design Patterns: 3 messaging patterns

Messaging patterns help connect services in a loosely coupled manner. What this means i...

5 min read · Jun 4

82 3

Each of the projects I've worked on in the last few years has involved a distributed messag...

21 min read · Nov 16, 2019

1.91K 5

See more recommendations