Garanti BBVA Technology

Introduction to Microservices

Functions: The Universal Descriptors

Functions describe the world. **Everything is described by functions**. The sound of my voice on your eardrum isn't just noise — it's a function, a wave oscillating over time, measurable and predictable.

The light that's kind of hitting your eyeballs right now? That's a function too, a spectrum of energy bending through space to paint your vision.

$$y = f(x)$$

Monolithic Architecture

A monolithic application can be expressed as a single function $M(x)$, where x represents the input (such as user requests or data). The entire system is tightly coupled:

$$M(x) = f_1(f_2(f_3(\ \ldots f_n(x)\ \ldots)))$$

where:

- f_i represents **individual components** (e.g., *authentication, business logic*).
- The system is **tightly coupled**, meaning **all components must work together** in sequence.

Microservices Architecture

In a microservices based system, we break it down into independent services:

$$M(x) = S_1(x_1) + S_2(x_2) + S_3(x_3) + \ldots + S_n(x_n)$$

where:

- Each $\mathcal{S}(x)$ is an independent microservice.
- They can operate, scale and be deployed independently.

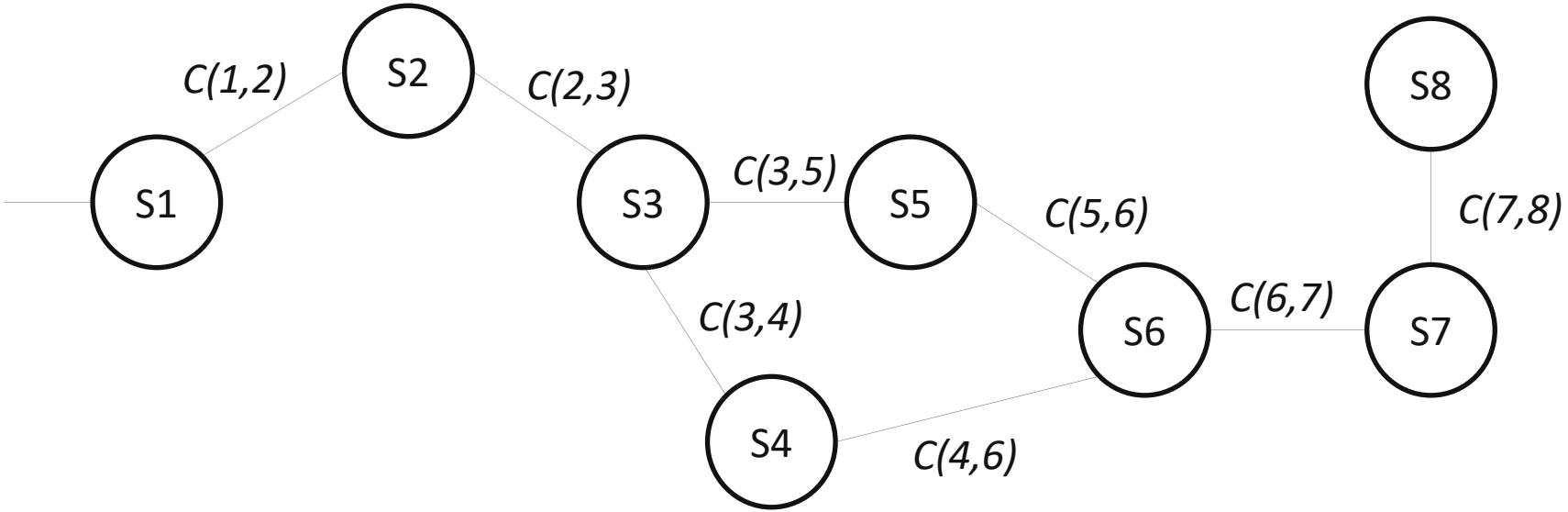
If we consider **inter-service communication**, we can extend the formula:

$$M(x) = \sum_{i=1}^{n} S_i(x_i) + \sum_{i,j} C_{ij}(S_i, S_j)$$

where:

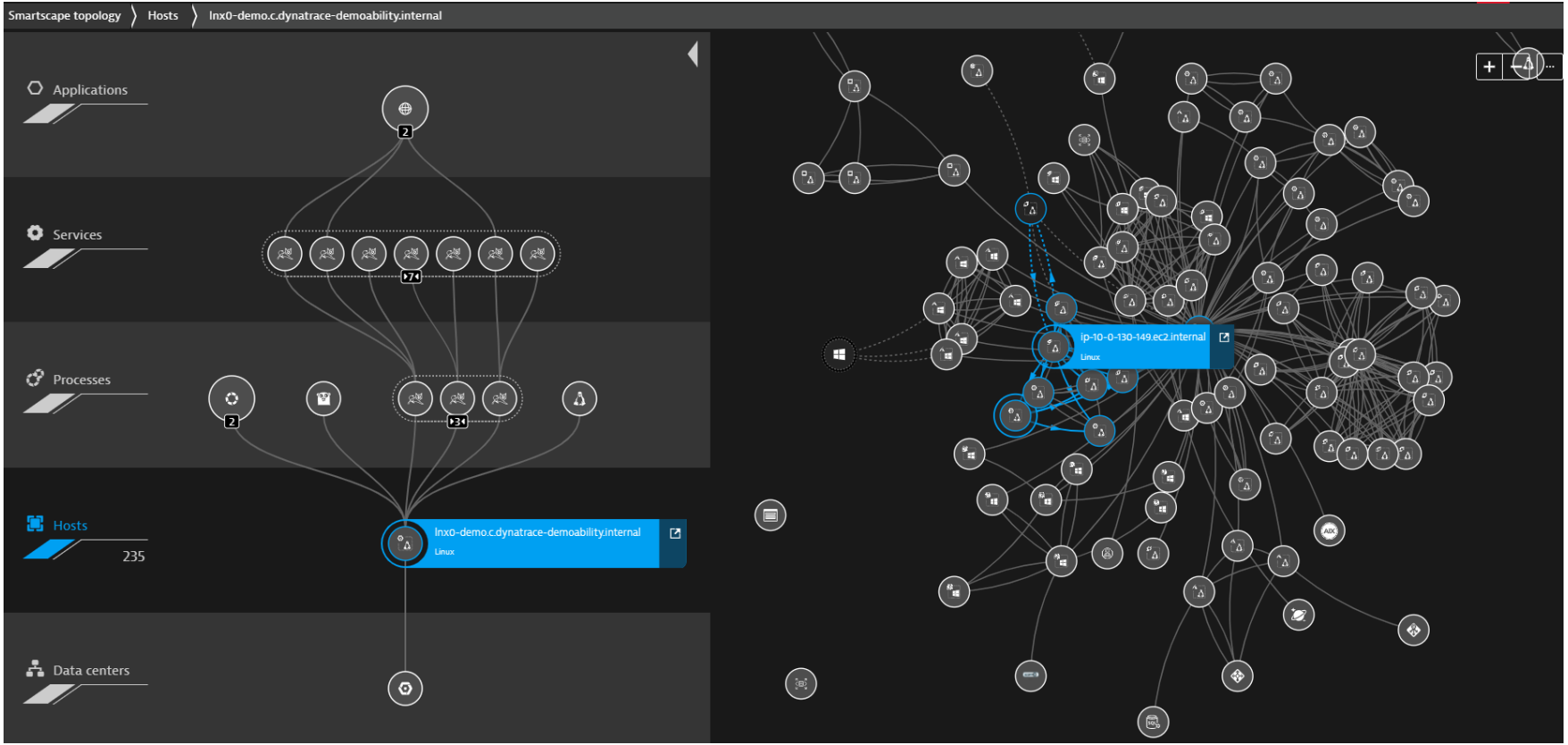
- $C_{i,j}$ represents the communication cost between services $\mathcal{S}_i$ and $\mathcal{S}_j$. This accounts for network latency, API calls and data exchange overhead.

Microservices Architecture (Graphs)



$$M(x) = \sum_{i=1}^{n} S_i(x_i) + \sum_{i,j} C_{ij}(S_i, S_j)$$

Example View of Real-Life Microservices Topology



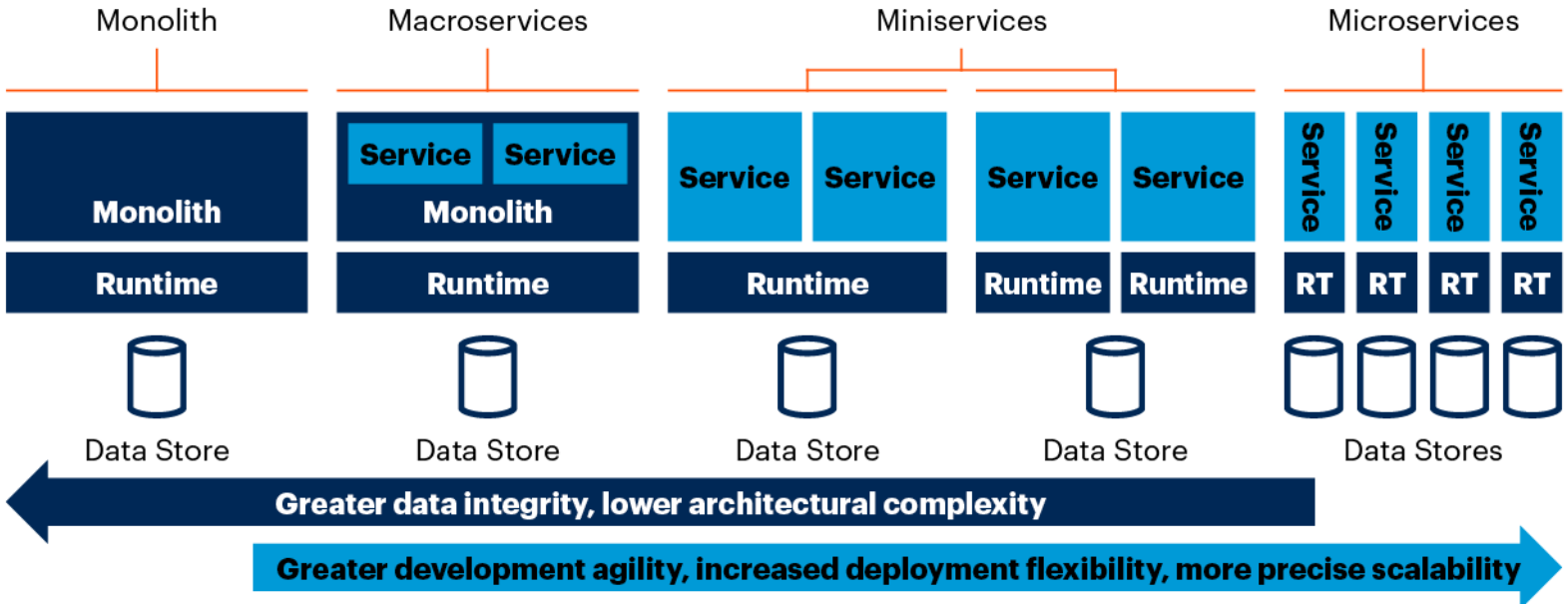
Is it a magical Architecture ?

ONE SOLUTION TO ONE PROBLEM MAKES TWO PROBLEMS

EUGENE J MARTIN

What are microservices ?

Microservice Architecture breaks down application into small, independent services that communicate over APIs. It contrasts with Monolitich Architecture, offering better scalability, flexibility and fault isolation. However, it also introduces **complexity** in ***communication and management***.



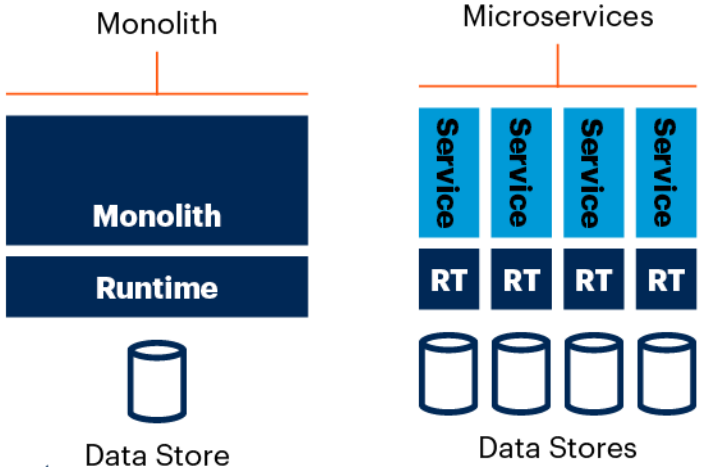
Comparision with Monolith Architecture

Monolithic Architecture:

- A single, unified application where all components are tightly integrated.
- Easier to develop and deploy but **hard to scale** and modify.
- A failure in one module can impact the entire system.

Microservices Architecture:

- Applications are broken into independent services communicating via APIs.
- Enables **scalability**, **flexibility** and **faster** deployments.
- Increases **complexity** in communication, deployment and monitoring.



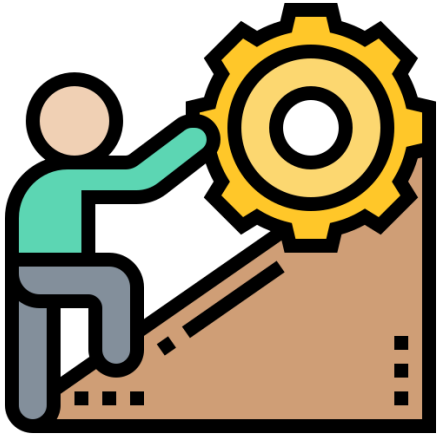
Benefits of Microservices

- **Scalability:** Scale individual services independently based on demand.
- **Flexibility**: Each service can use different technologies and frameworks.
- **Faster Development & Deployment**: Teams can develop, test and deploy services separately.
- **Fault Isolation**: A failure in one service doesn’t crash the entire system.
- **Better Maintainability**: Small, focused services are easier to manage and update.
- **Improved Security**: Services can have different security policies and access controls.

Microservices enable **agility**, **resilience** and **innovation**!

Challenges of Microservices

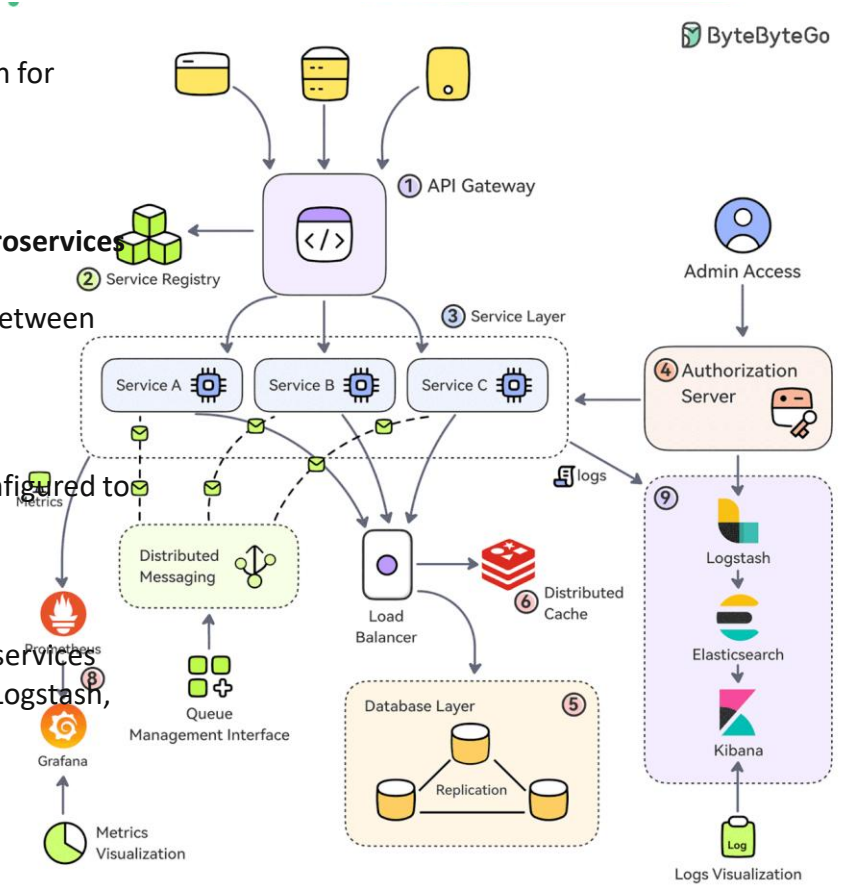
- **Increased Complexity**: Managing multiple services, databases and deployments is challenging.
- **Communication Overhead:** Services must handle network latency, failures and data consistency.
- **Data Management:** Ensuring consistency across distributed databases is complex.
- **Deployment & Monitoring**: Requires orchestration tools (Kubernetes, CI/CD pipelines) for automation.
- **Security Risks**: More attack surfaces due to multiple exposed APIs.



Microservices offer flexibility but require strong **DevOps**, **monitoring** and **security practices**!

Main Components of Typical Microserice Application

- **Service Registry**
  The service registry contains the details of all the services. The gateway discovers the service using the registry.
- **API Gateway**
  The gateway provides an unified entry point for client applications.
- **Service Layer**
  Each microservices serves a specific business function and can run on multiple instances.
- **Authorization Server**
  Manage identity and access control.
- **Data Storage**
  Databases like PostgreSQL and MySQL store application data
- **Distributed Caching**
  Caching is a great approach for boosting the application performance.
- **Event Brokers/ Async Microservices**
  Communication between microservices async communication
- **Metrics/Tracing**
  Microservices can be configured to publish metrics
- **Log Aggregation**
  Logs generated by the services are aggregated using Logstash, stored in Elasticsearch and visualized with Kibana.

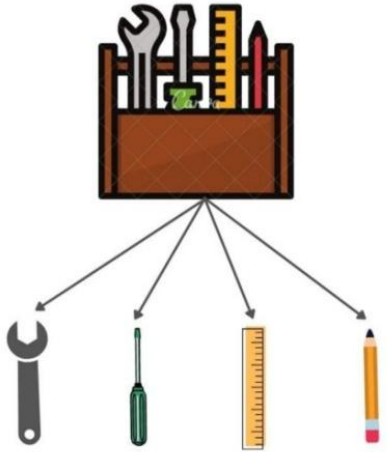


Garanti BBVA Technology

Core Principles of Microservice Architecture

Single Responsibility Principle (Each service does one thing well)

- Each microservice should focus **on a single business capability**.
- **Simplifies development** : Smaller codebases are easier to manage.
- **Improves maintainability** – Changes affect only one service.
- **Enhances scalability** – Scale specific services based on demand.
- **Reduces dependencies** – Services operate independently.
- "**Do one thing and do it well**" – The key to effective microservices!

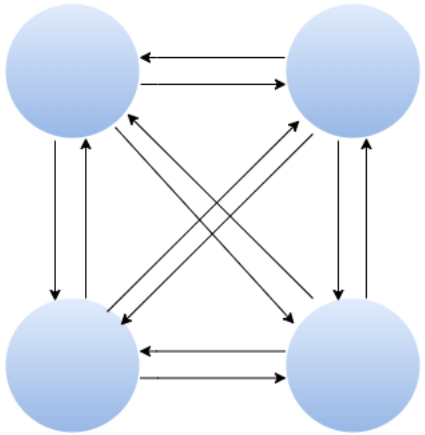


A Class should have only one reason to change

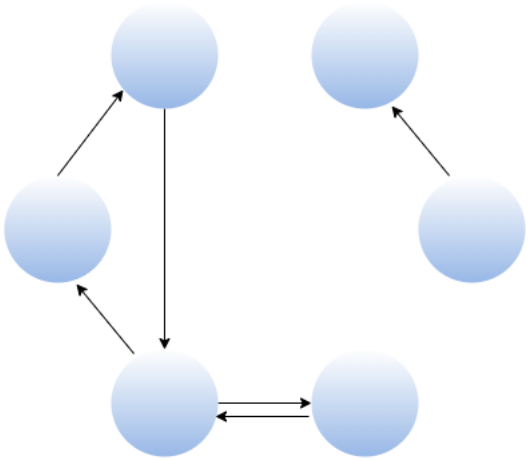
Robert C. Martin - *SOLID Principles*

Independence (Services are loosely coupled)

- Microservices operate **independently, minimizing dependencies** between them.
- Teams can build and **deploy services separately**.
- Each service can use **different programming languages or databases**.
- One service failure **doesn't crash the entire** system.
- **Scale only the required services**, not the whole application.



Tightly Coupled
Many Dependencies



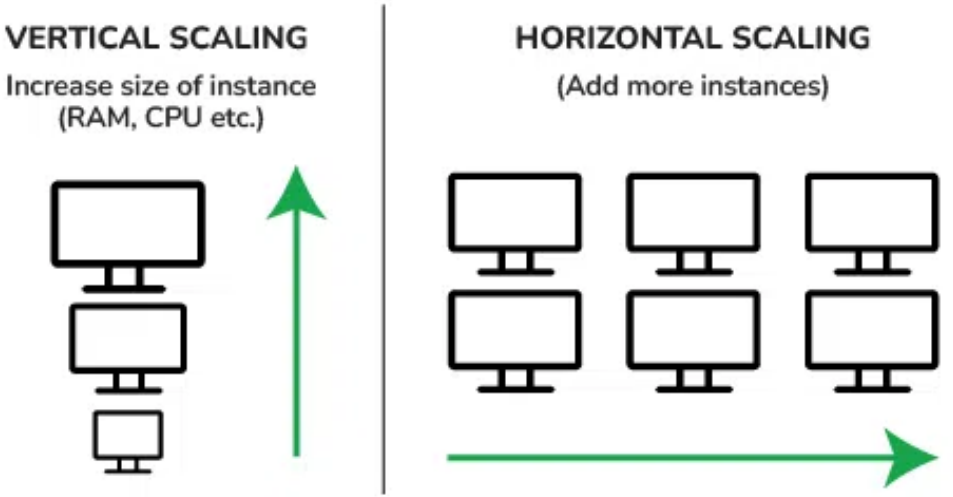
Loosely Coupled
Some Dependencies

Loose coupling = More flexibility, better resilience!

Scalability (Scaling individual services)

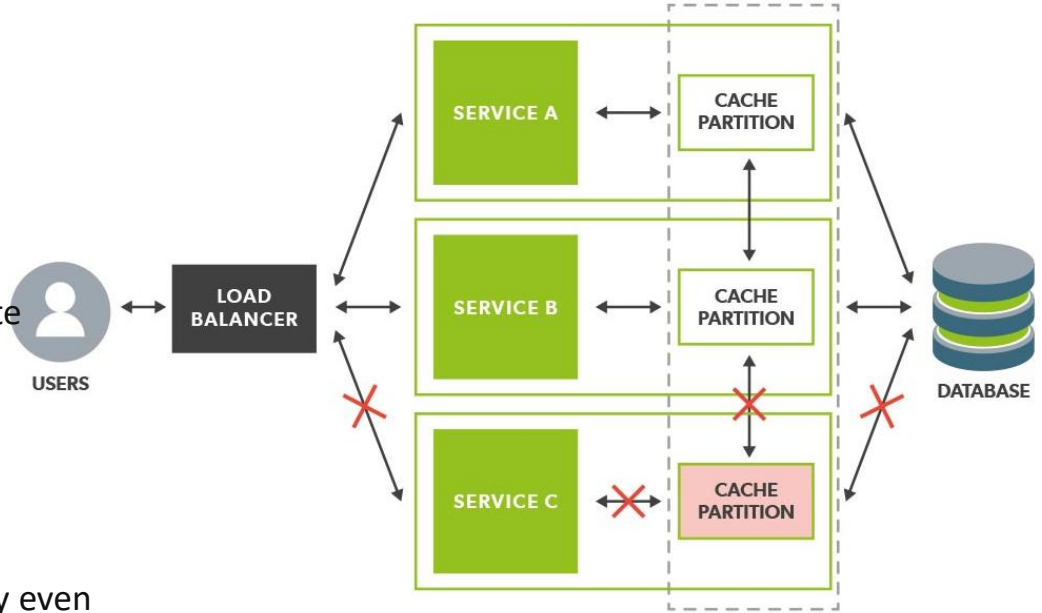
- Microservices allow scaling only the needed components, not the entire system.
- **Optimized Resource Usage:** Scale high-traffic services independently.
- **Cost Efficiency**: Avoid over-provisioning resources for the whole application.
- **Better Performance:** Improves response times and user experience.
- **Supports Global Traffic**: Easily distribute services across multiple regions.

Scale smart, not hard – Optimize performance with microservices!



Resillience (Failure isolation and recovery)

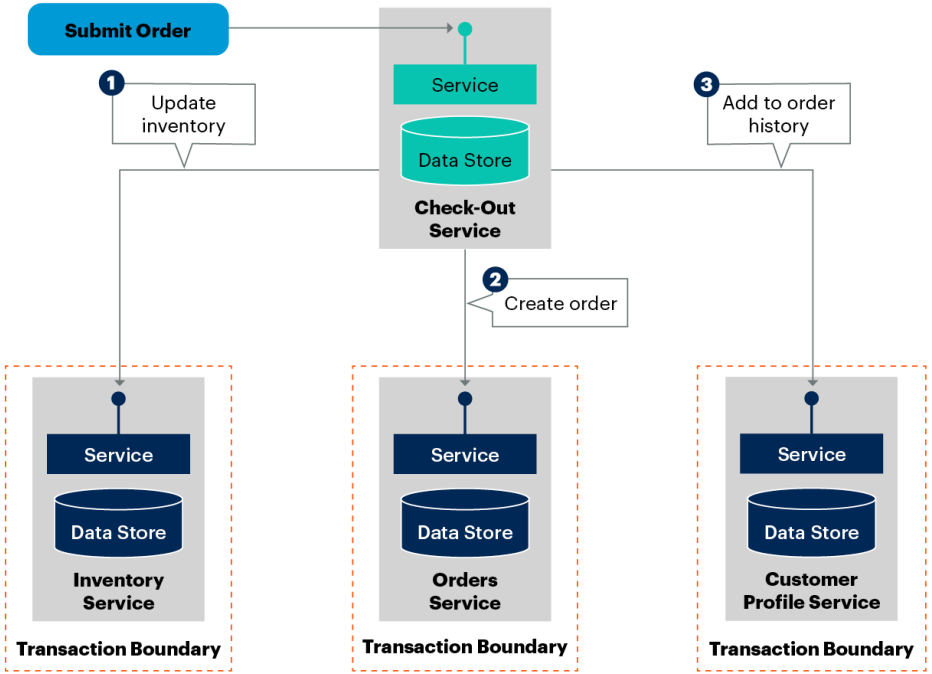
- Microservices are designed to **handle failures without affecting** the entire system.
- **Failure Isolation**: If one service crashes, others continue running.
- **Automatic Recovery:** Services restart or reroute traffic automatically.
- **Circuit Breakers:** Prevent cascading failures by stopping unhealthy services
- **Redundancy & Replication:** Ensures availability even during failures.



Stay online, stay resilient – Microservices make systems fault-tolerant!

Decentralized Management (services manages its own database)

- Each microservice has its own database, **avoiding tight coupling**.
- **Independence**: Services manage their own data without affecting others.
- **Scalability**: Different databases can be optimized fc specific needs.
- **Flexibility**: Services can use SQL, NoSQL or other storage solutions.
- **Improved Performance**: Reduces bottlenecks and contention issues.



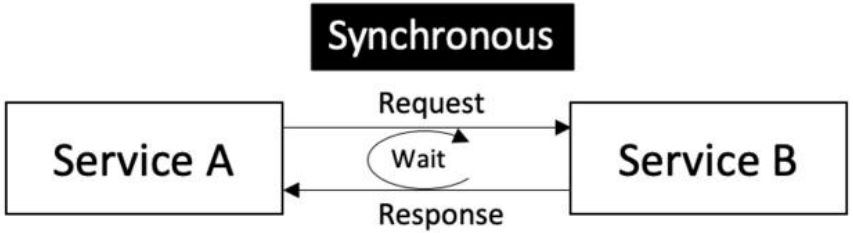
Ensuring data consistency across services requires strategies like **Event Sourcing, CQRS or Saga Patterns**. Therefore, "***Own your data***" – Microservices ensure autonomy and efficiency!

Garanti BBVA Technology

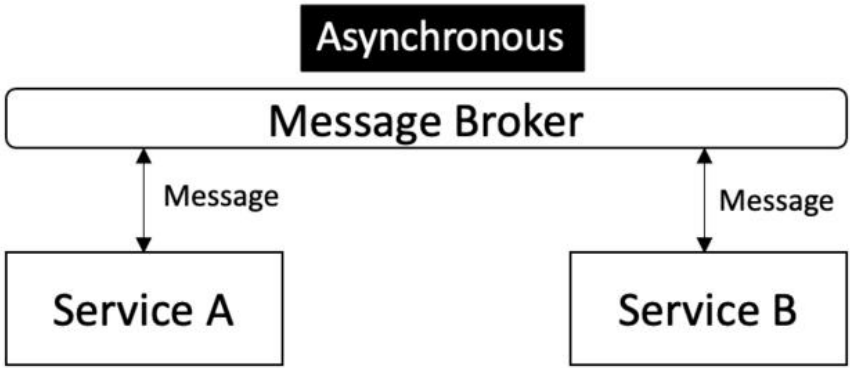
Microservice Communication

Communication Types : Synchronous & asynchronous

- **Synchronous communication** allows microservices to interact by making direct requests and waiting for responses. In the simplest case, the client **blocks** until the service returns with a response. Popular protocols are REST, SOAP, gRPC, WebSocket



- In an **asynchronous communication** model, the requester **need not wait** for the response. Services communicate by exchanging messages using a message broker, which acts as an intermediary between the services.

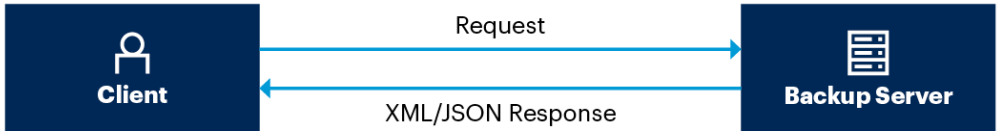


- Using a messaging framework, services can implement both **synchronous** and **asynchronous** style of interactions.

Synchronous Communication : REST, gRPC

REST

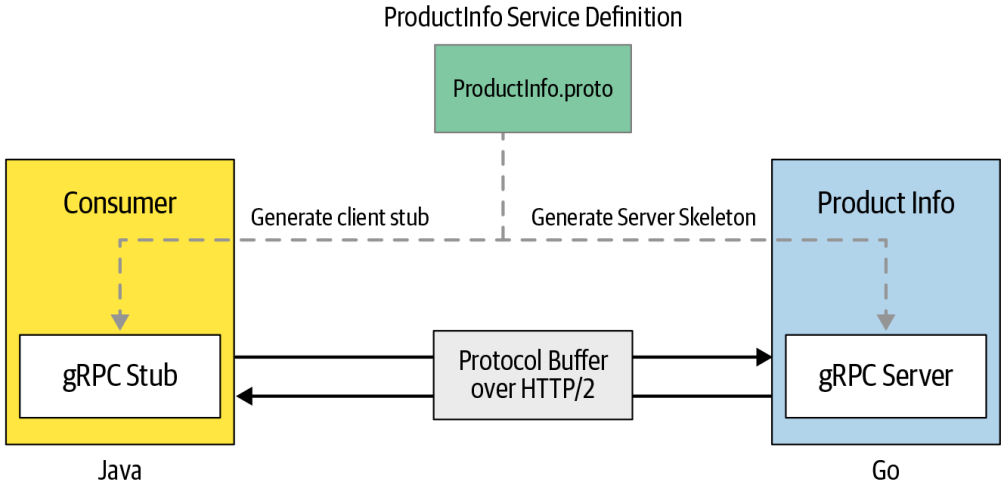
- Based on **HTTP**, widely used and easy to implement.
- Works with **JSON**, making it suitable for web and mobile applications.
- Stateless communication, simple and scalable for web APIs.



Synchronous communication allows microservices to interact by making direct requests and waiting for responses.

gRPC

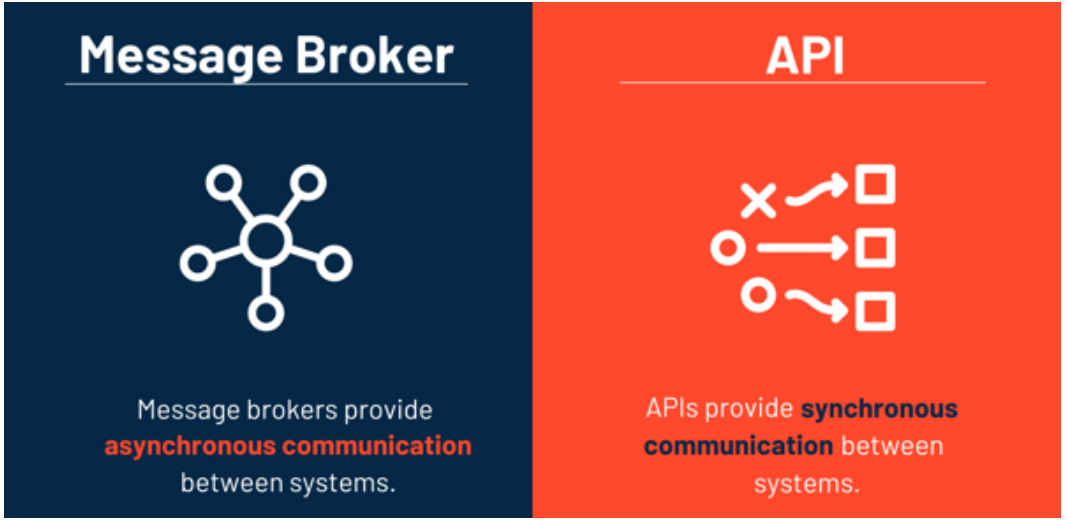
- **Faster than REST** due to reduced payload size and efficient serialization.
- High-performance communication using **Protocol Buffers** (binary format)
- Supports bidirectional streaming and real-time communication



Synchronous communication = Real-time, direct service interaction

Asynchronous Communication : Message Brokers

- Microservices can communicate asynchronously through **message brokers**, **reducing direct dependencies**.
- **Loose Coupling**: Services send messages without waiting for immediate responses.
- **Scalability**: Message queues handle spikes in traffic by buffering requests.
- **Reliability**: Message brokers ensure message durability and delivery guarantees (e.g., retry mechanisms).
- **Event-Driven Architecture**: Enables real-time processing and event-based workflows.



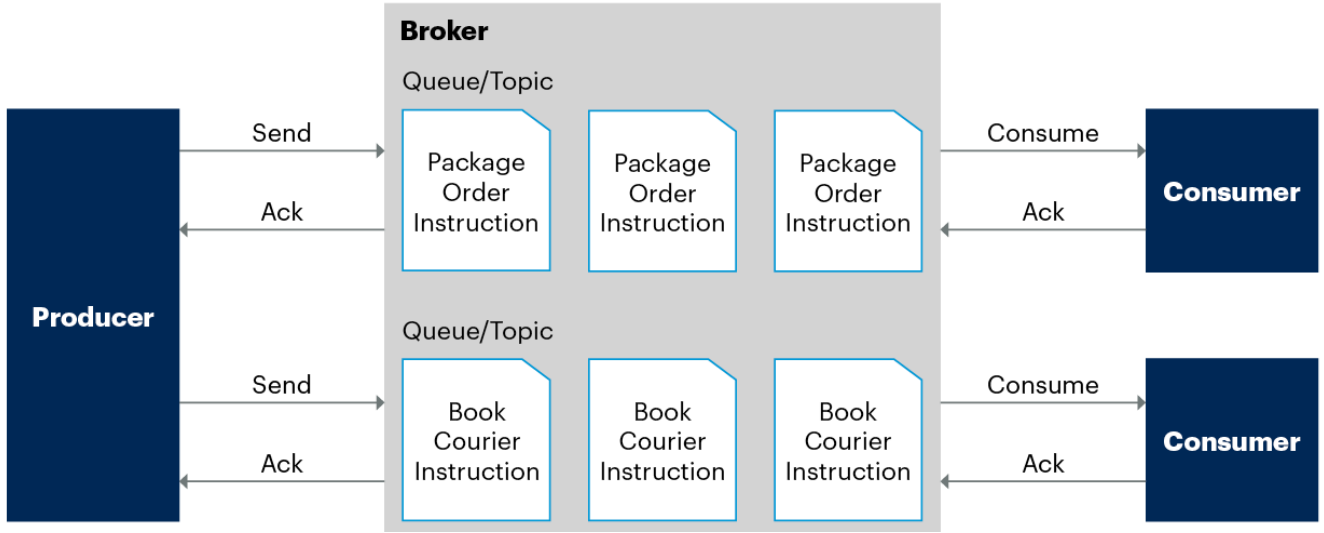
Asynchronous communication with message brokers = Faster, more resilient systems!

Garanti BBVA Technology

Message Brokers

Message Driven-Communication

- In message-driven communication, a producer sends a message to a consumer.
- This messaging is **fire-and-forget**. The producer does not receive any immediate feedback or confirmation that the consumer has received or processed the message



- An intermediary component, such as an **event** or **message broker**, will usually transport the message.
- The broker can **buffer messages** until the consumer is ready to receive them. The use of a broker decouples the performance and life cycle of producer and consumer

Types of Events

Asynchronous (Async) Events

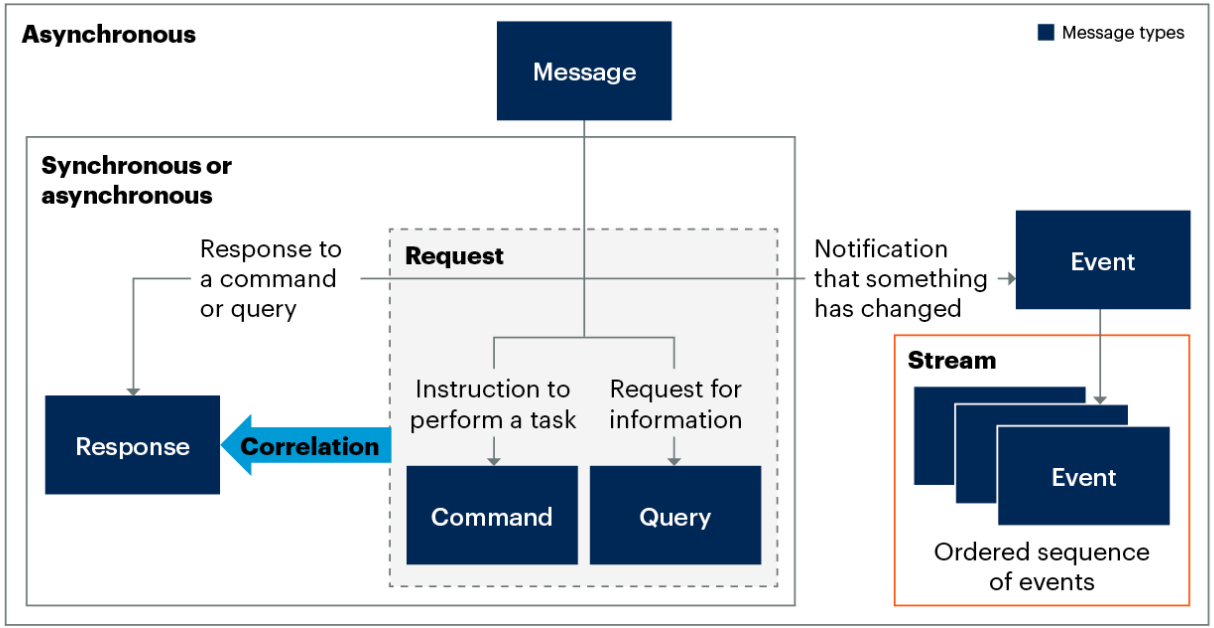
- Events are triggered and processed independently, with n immediate response required.

Streaming Events

- Events are continuously streame in real-time, allowing real-time data exchange between services

Synchronous (Sync) Events

- Events require immediate action and response from the receiving service.

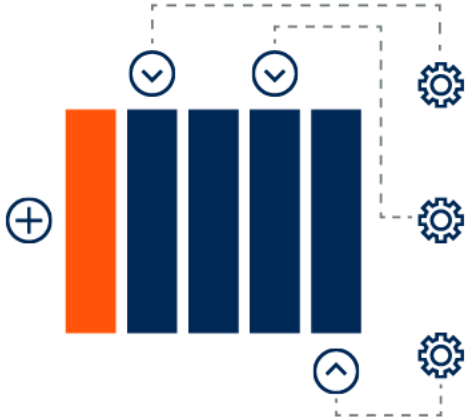


Three Common Classes of Event Broker

Queue-Oriented Brokers

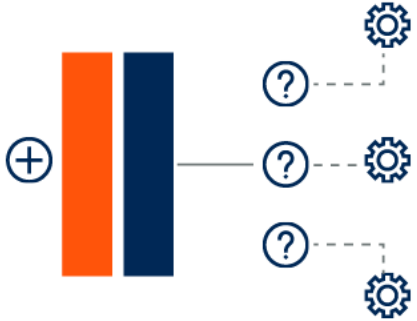
- Queue per subscriber group
- Destructive read on consumption
- Flexible topic structure

Log-Oriented Brokers



- Append only, partitioned log
- Non-destructive read from log offset
- Flat topic structure

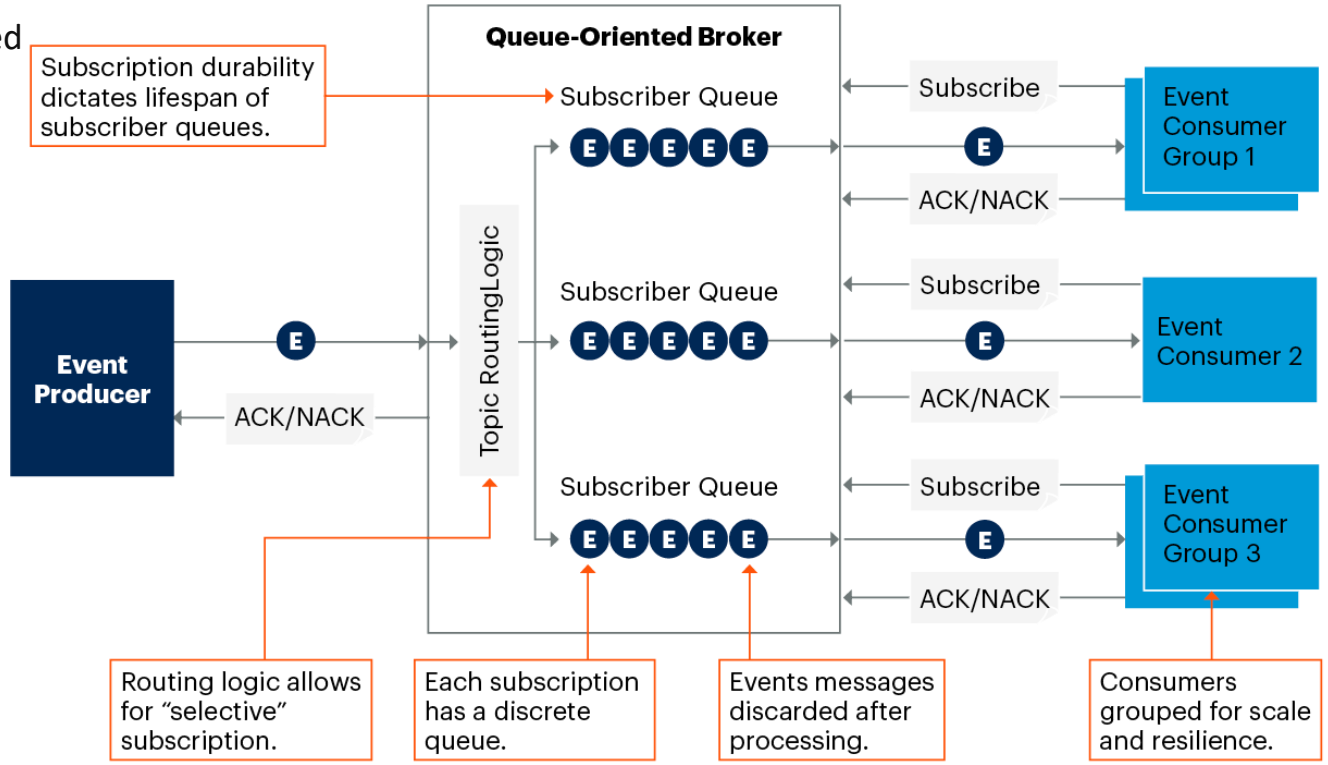
Subscription-Oriented Brokers



- Non-ordered push delivery
- Filtered subscriptions
- Serverless architecture

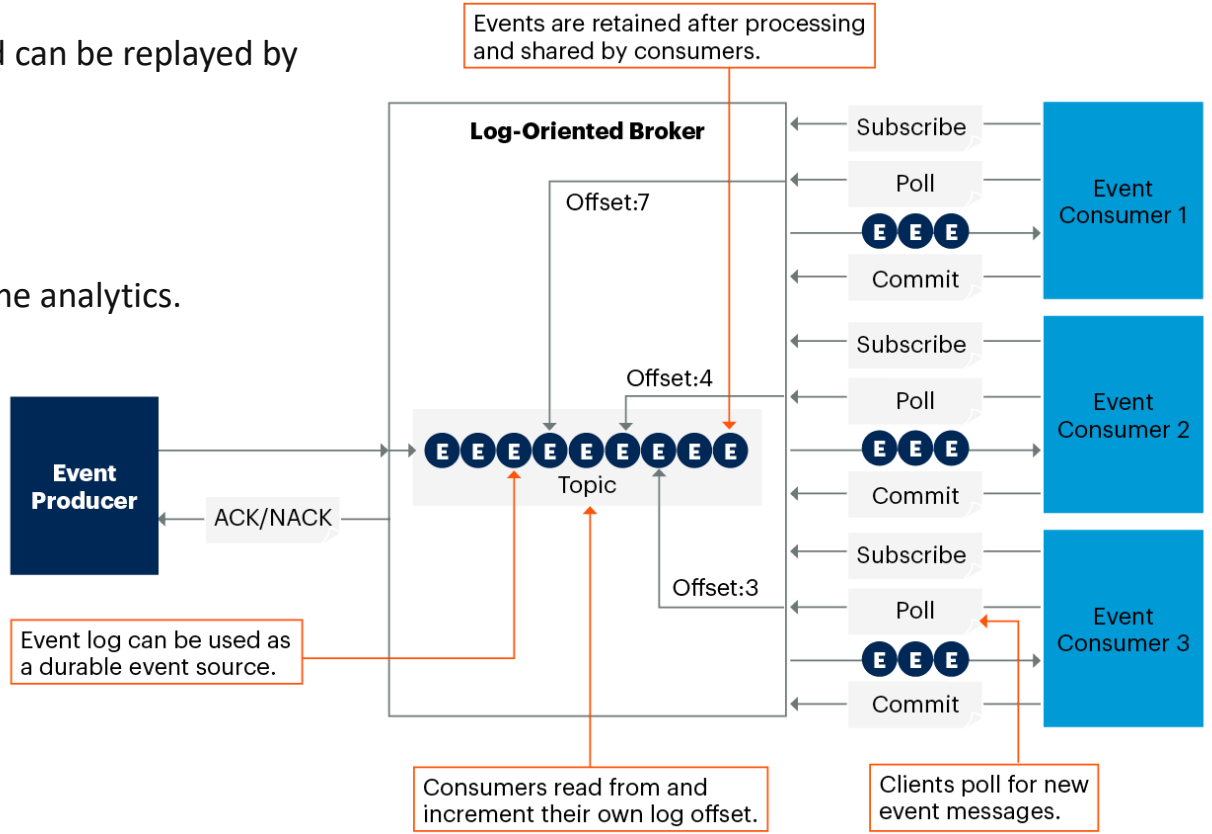
Queue-Oriented Brokers

- **Events**/messages are stored in a **FIFO queue** and processed once by a single consumer.
- Example: **RabbitMQ**, ActiveMQ
- Best for: Task distribution, job processing.



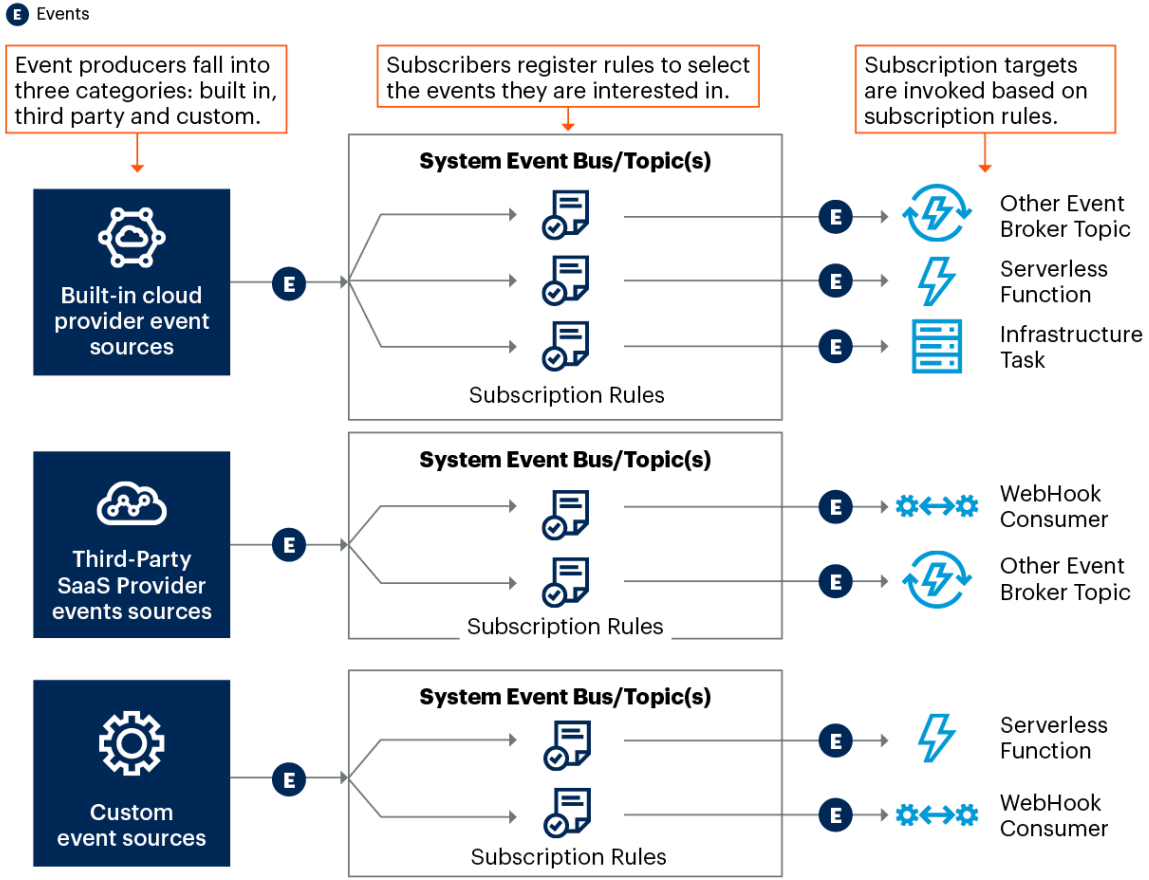
Log-Oriented Brokers

- **Events** are persisted in a log and can be replayed by **multiple consumers**.
- Example: **Apache Kafka**, Pulsar
- Best for: Event sourcing, real-time analytics.



Subscription-Based Broker Architecture

- Events are published to a topic and multiple subscribers receive them asynchronously.
- Example: Google Pub/Sub, AWS SNS, **RabbitMQ**
- Best for: Broadcasting events to multiple services.



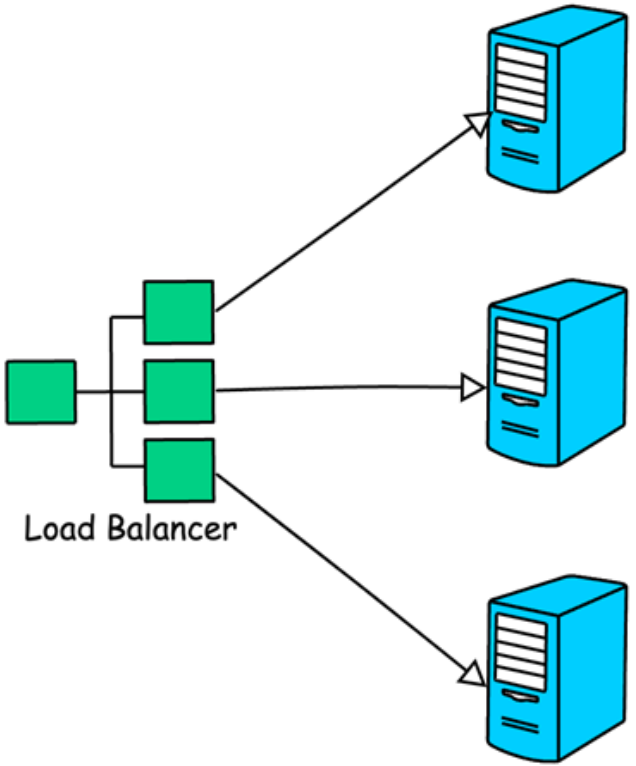
Garanti BBVA Technology

Load Balancing

Load Balancing

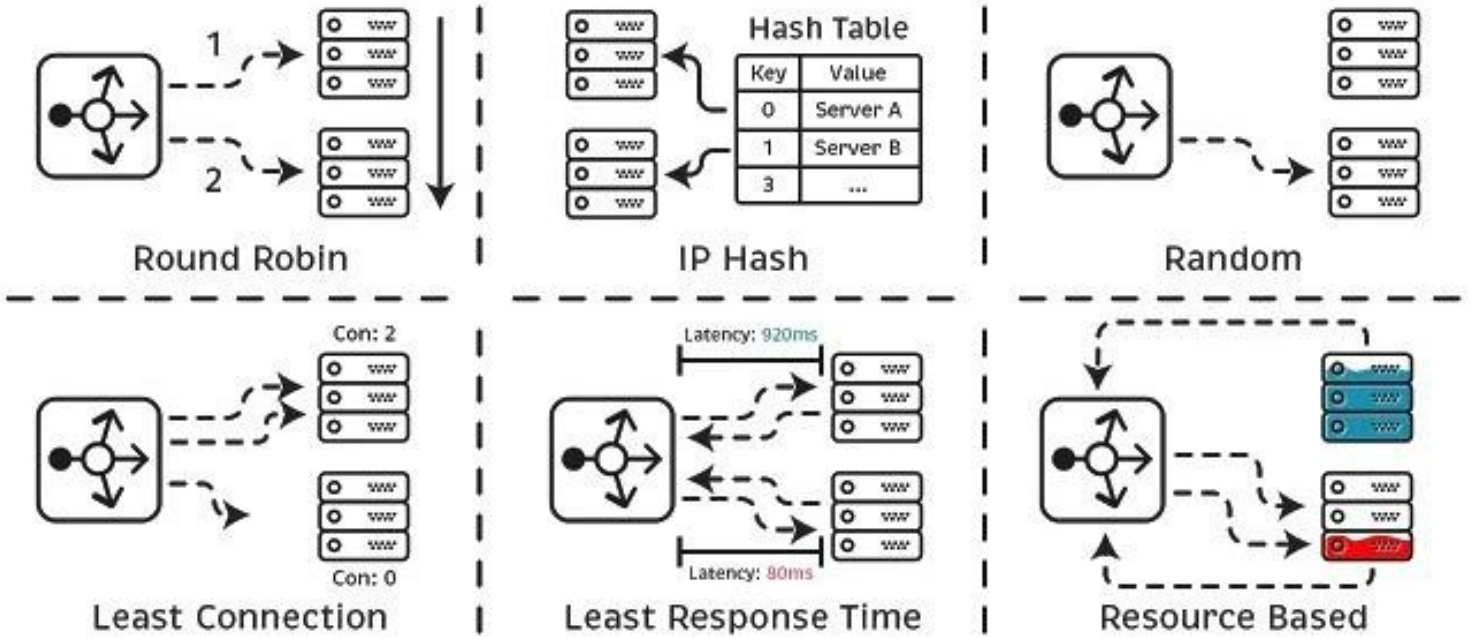
Load balancing is the process of distributing incoming network traffic across multiple servers to ensure that no single server is overwhelmed

- **Client-Side Load Balancing** : The client (e.g., API Gateway) decides which service instance to communicate with.
- **Server-Side Load Balancing :** distribute incoming requests to services
- **Round-Robin Load Balancing**: Requests are distributed sequentially to all available service instances
- **Weighted Load Balancing:** Assigns weights to instances based capacity or health, sending more traffic to the more powerful or available instances.
- **Health Checks and Failover :** Automatically routes traffic away from unhealthy and redirects it to healthy ones.



If we do not have Load Balancing,…

Load Balancing Strategies

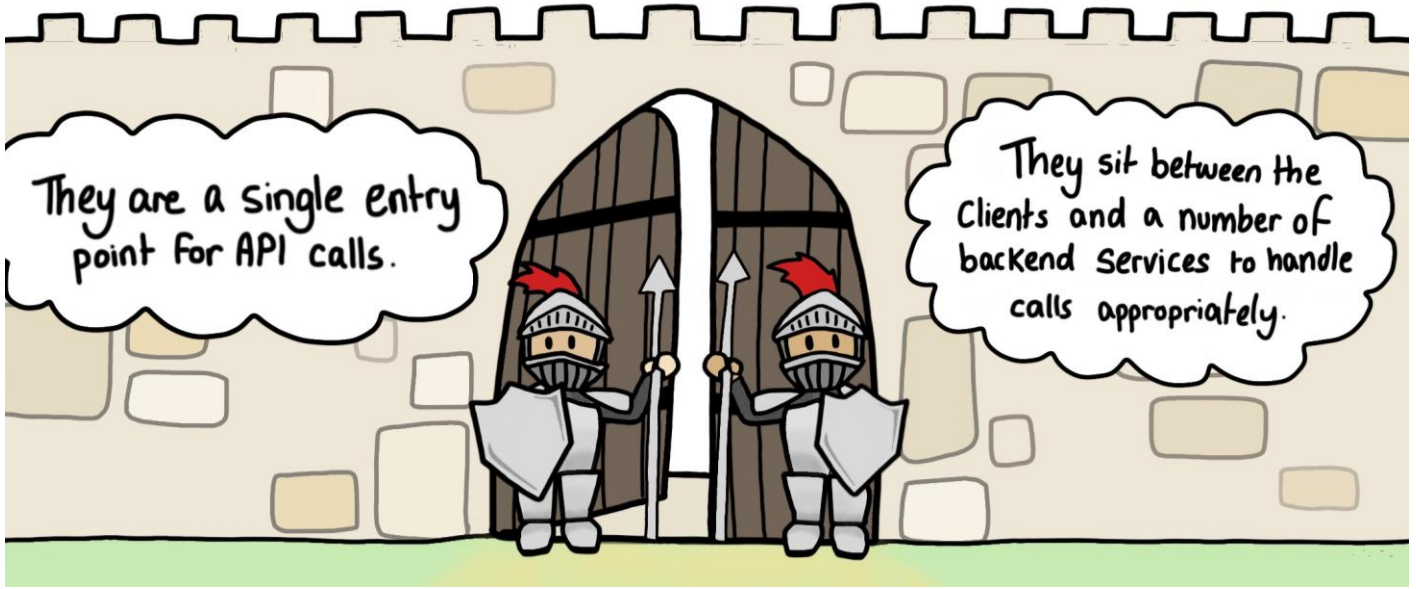


Garanti BBVA Technology

API Gateway

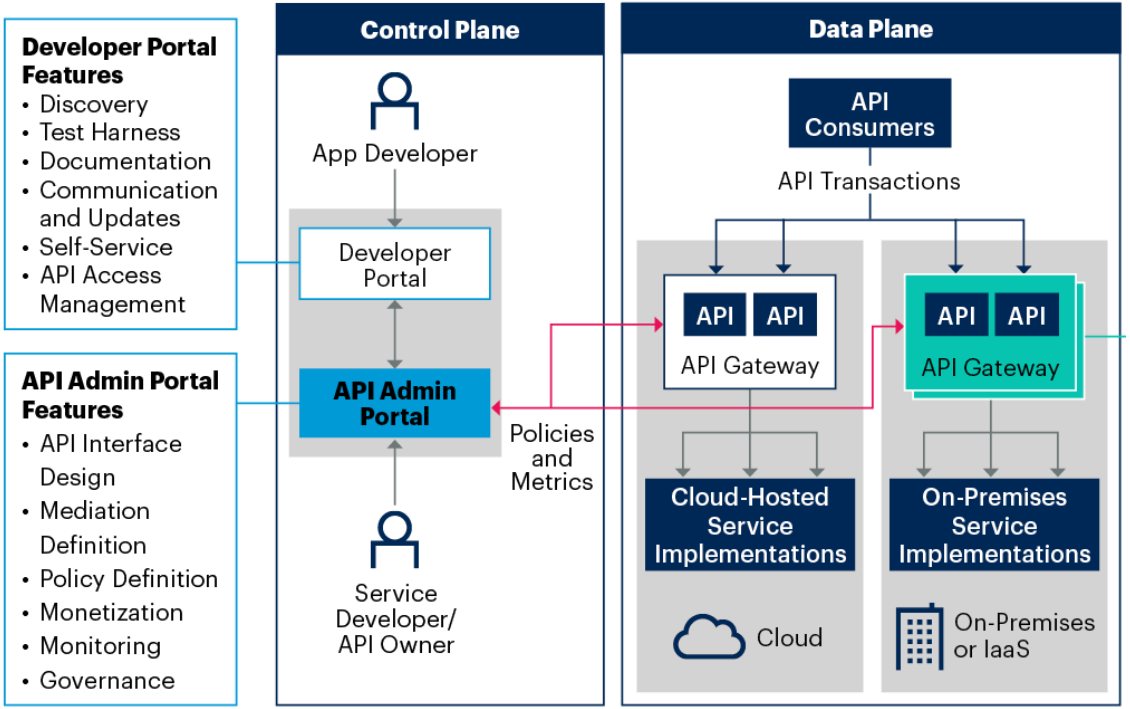
API Gateway

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



API Gateway (Single entry point)

- An API Gateway acts as a **single entry point** for all client requests to microservices.
- **Centralized Routing**: Directs requests to the appropriate microservice.
- **Simplifies Client Interaction**: Clients only need to interact with one endpoint, not multiple services.
- **Cross-Cutting Concerns**: Handles authentication, rate limiting, logging and load balancing.
- **Security**: Can enforce security policies like OAuth2 or JWT for all services.



API Gateway = Simplified, secured and unified access to microservices!

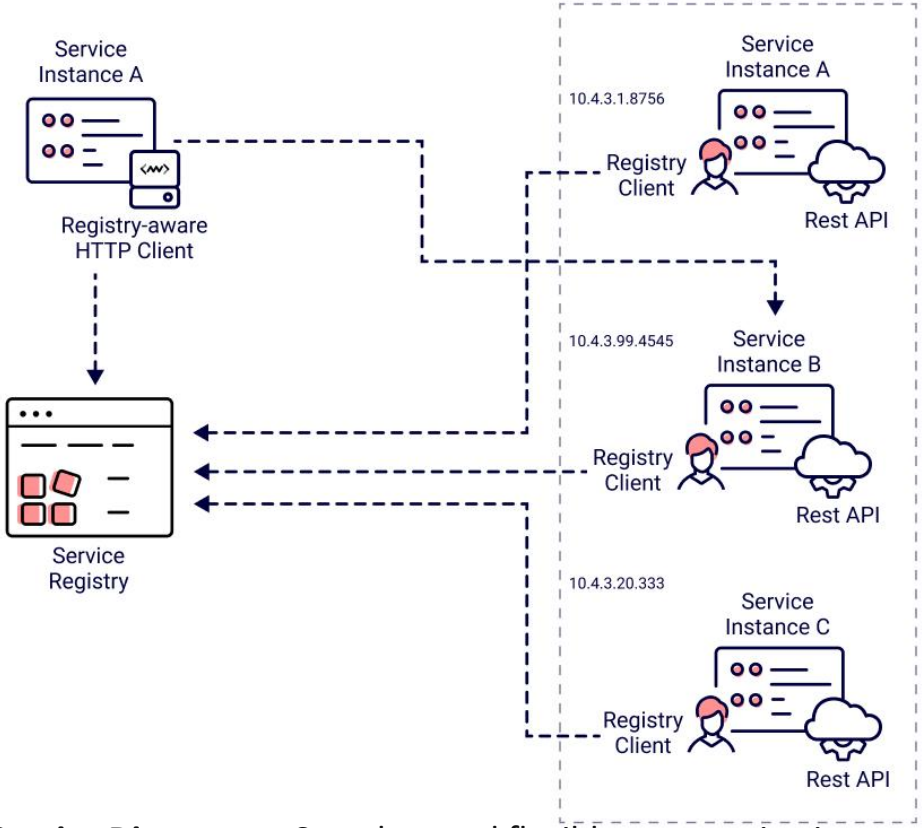
Garanti BBVA Technology

Service Registry & Discovery


I CAN'T FIND IT
ANYWHERE!
IT'S JUST GONE!
IT.
WHENEVER I SEND MY KIDS
TO FIND SOMETHING
facebook.com/FowLanguageComics
FowLanguageComics.com
©Brian Gordon

Service Registry & Discovery

- **Service Discovery** allows microservices to automatically locate and communicate with each other without hardcoding addresses.
- **Dynamic Registration**: Services register themselves with a discovery server (Eureka, Consul).
- **Automatic Detection**: Clients can find services without needing to know their location or IP addresses.
- **Load Balancing**: Ensures requests are distributed across healthy instances of services.
- **Fault Tolerance**: Automatically reroutes traffic if a service instance becomes unavailable.

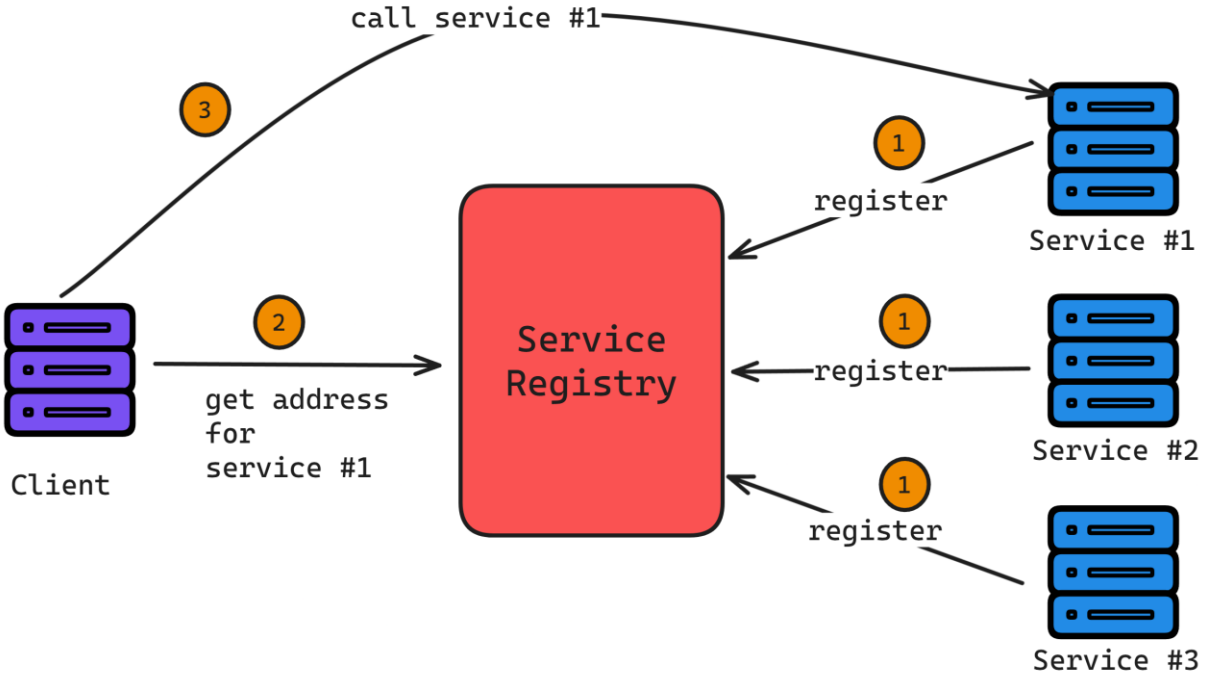


Service Discovery = Seamless and flexible communication across microservices!

Service Registry & Discovery

Client-Side Discovery

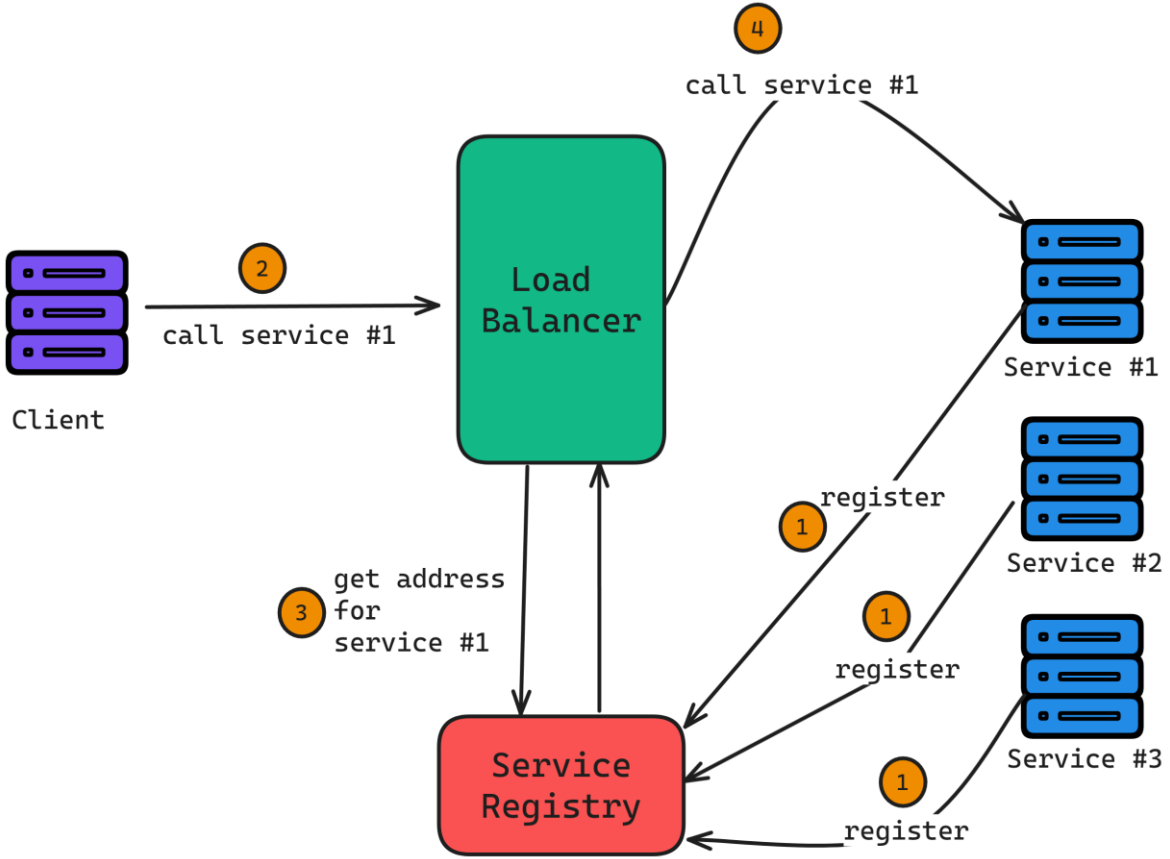
- In client-side discovery, the client is responsible for determining the location of a service instance.
- The client queries a service registry to find available instances of a service and then uses a load balancing algorithm to select one.



Service Registry & Discovery

Server-Side Discovery

- In **server-side discovery**, the client makes a request to a load balancer, which then forwards the request to an available service instance.
- The **load balancer** queries the service registry to find available service instances.



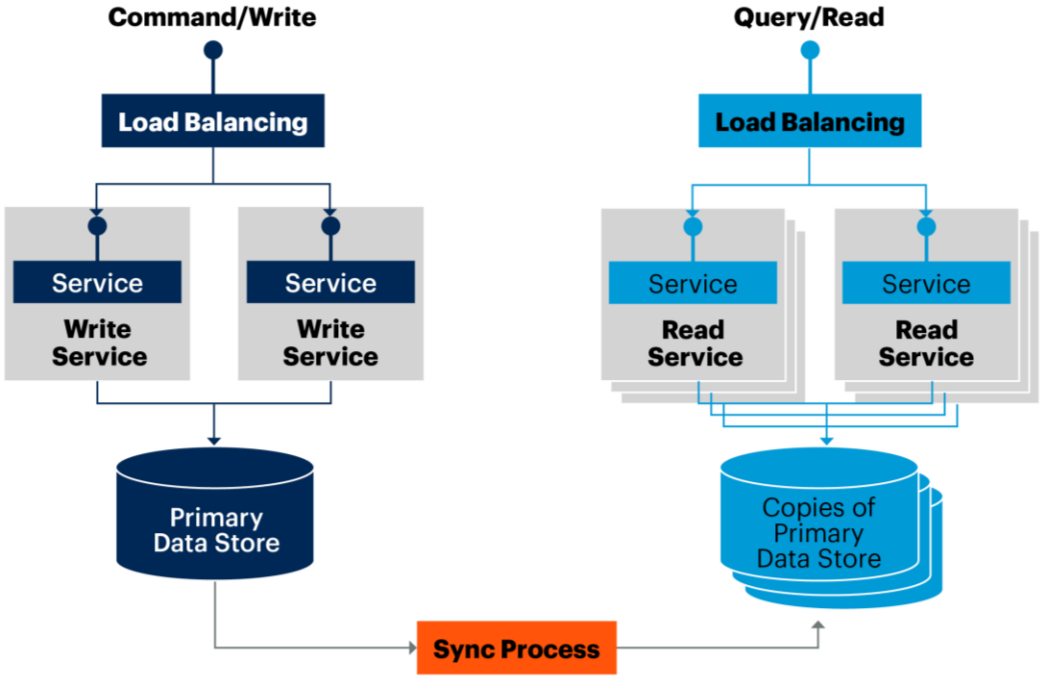
Garanti BBVA Technology

Data Management & Patterns

CQRS (Command and Query Responsibility Segregation)

Separates read and write operations into different models to optimize performance and scalability.

- **Command Model (Write):** Handles updates, inserts and deletes (modifies data).
  Query Model (Read): Optimized for fast data retrieval (fetches data).
  Improves Performance: Reads and writes are scaled independently.
  Enhances Flexibility: Different databases or structures can be used for read and write operations.
  Supports Event Sourcing: Commands generate events that update the query model asynchronously.

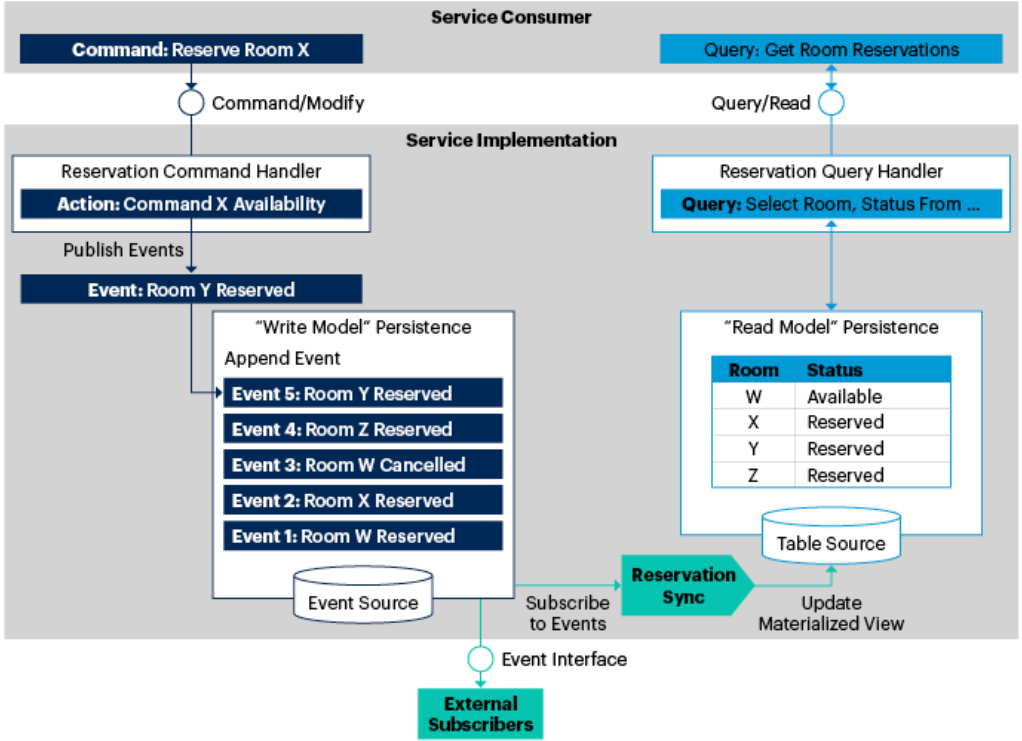


CQRS boosts efficiency in high-load applications by separating concerns

Event Sourcing with CQRS

- **Immutable Event Log**: Every change is stored as an event, ensuring a full history.
- **Rebuild State Anytime**: The current state can be reconstructed by replaying past events.
- **Enhances Auditing & Debugging:** Complete traceability of all changes.
- **Works Well with CQRS**: Events update the write model, while projections update the read model.
- **Supports Real-Time Processing**: Events can trigger actions in other services asynchronously.

Stores system state as a sequence of events



Event Sourcing ensures reliability, traceability and flexibility in microservices!

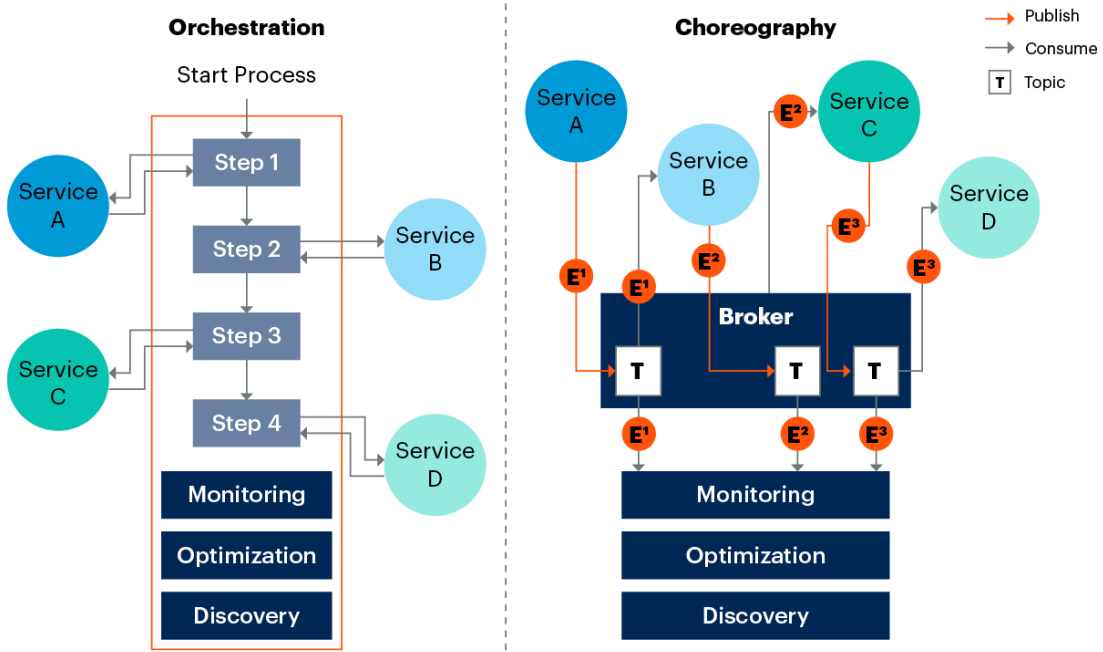
Process Orchestration (Orchestration vs Choregraphy)

Orchestration (Centralized Control)

- A central orchestrator (service or workflow engine) controls the interaction between microservices.
- Example: Camunda, Temporal, Apache Airflow.

Choreography (Decentralized Event-Driven Flow)

- Services communicate independently through events, without a central controller.
- Example: Event-driven architecture using Kafka, RabbitMQ.

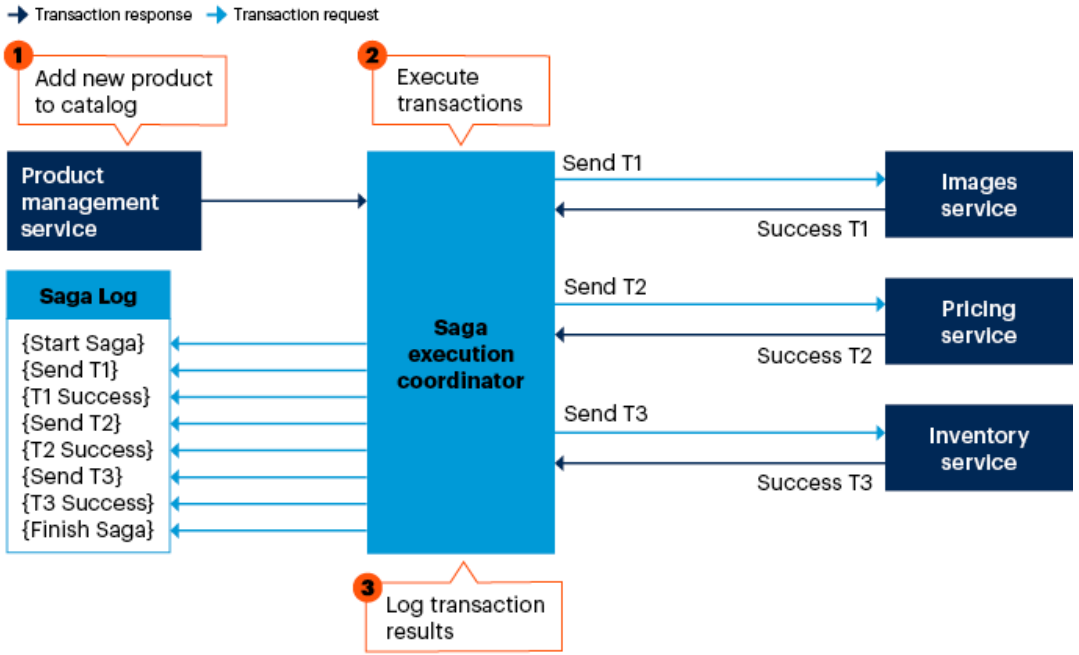


Use Orchestration for structured workflows and Choreography for highly scalable, event-driven systems!

Distributed Transactions (Saga Pattern)

Manages **both short & long-running**, distributed transactions by breaking them into smaller steps across multiple services helping consistency in distributed systems without **2PC** (Two-Phase Commit).

- **Choreography**: Each service reacts to events and triggers the next step (event-driven, no central controller).
- **Orchestration**: A central coordinator (orchestrator) manages the transaction flow.
- **Compensating Transactions**: If a step fails, previous actions are rolled back using compensating transactions.



Saga Pattern ensures reliability and consistency in distributed workflows!.

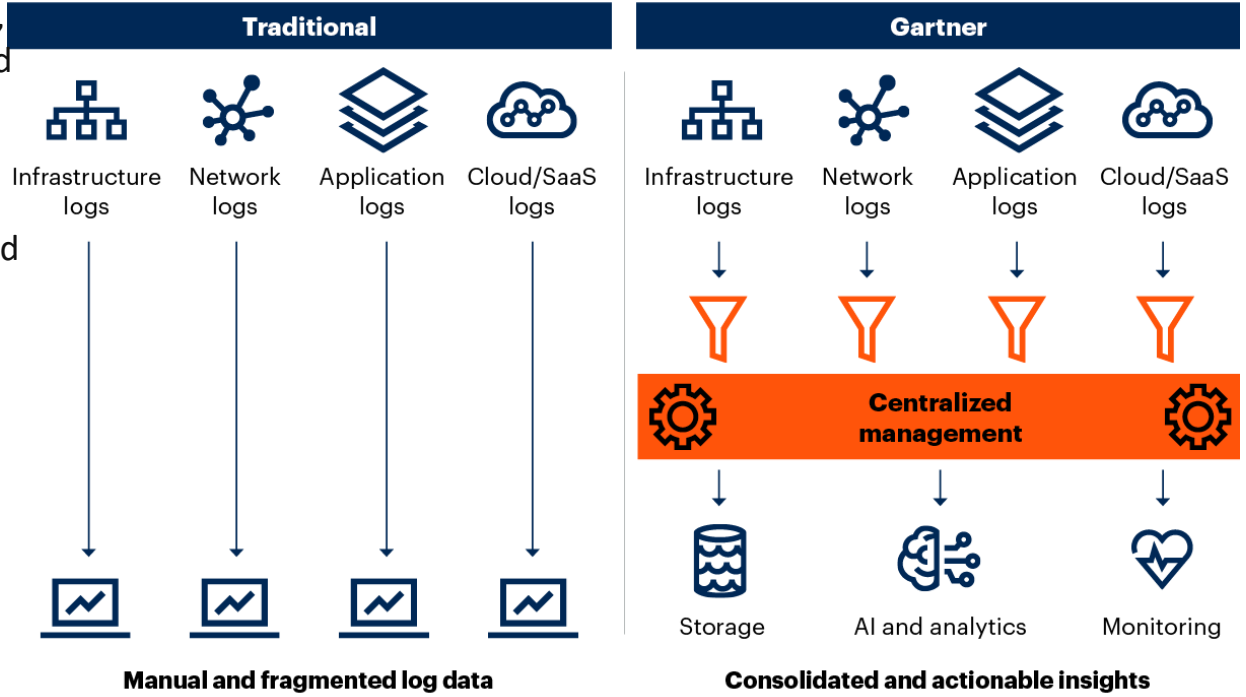
Garanti BBVA Technology

Monitoring and Observalibilility


Microservices
what' happered...
MICROSEVIICES
LOGS

Distributed Logging

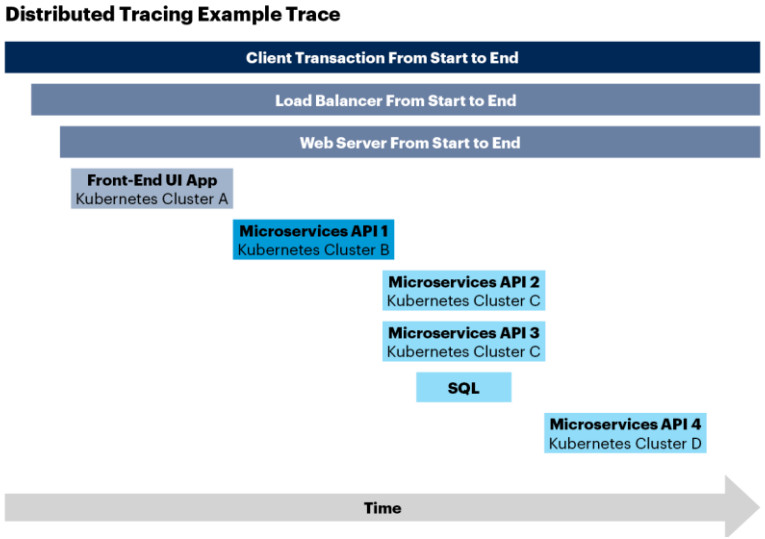
- **Log data** is essential for operations, DevOps, software development and quality assurance (QA).
- It offers **valuable insights** into the **state** and **health of applications** and infrastructure, aiding in proactive monitoring, incident resolution.
- Just as **patient monitors** provide a comprehensive view of a person's health, **log data should be integrated** to offer a complete picture of system health.

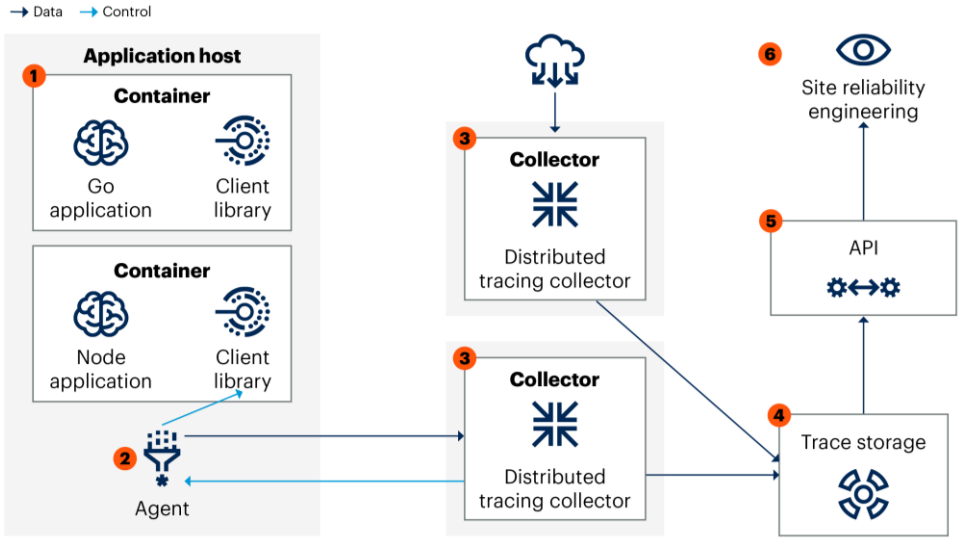


Distributed Tracing

- **Distributed tracing** is used in microservices-based or other distributed applications that are deployed on Kubernetes, because a single operation may touch many services deployed across multiple clusters or platforms.

- Distributed tracing supports the **collection of timing** and other **metadata**, as requests pass through these different services, including those invoked asynchronously.





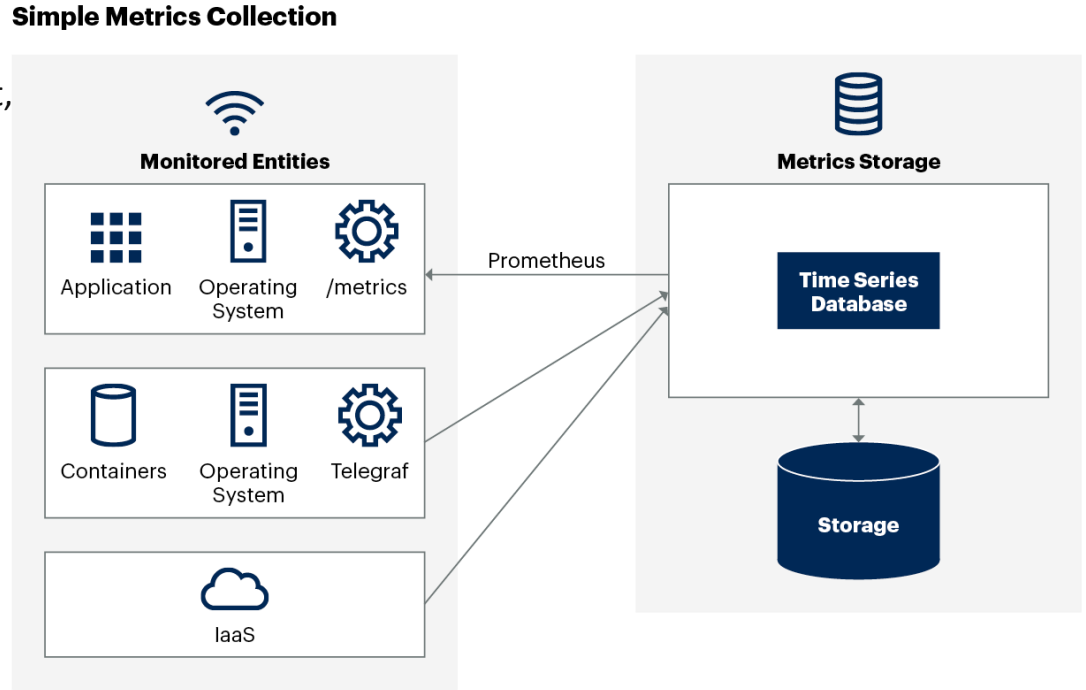
Metrics & Monitoring (Prometheus & Grafana)

Tracks the performance of microservices with quantitative data.

- **Key Metrics:** Latency, Error rate, Throughput, CPU/Memory usage, Request rate.
  Helps identify performance issues and monitor service health.

Continuous tracking of system health and performance.

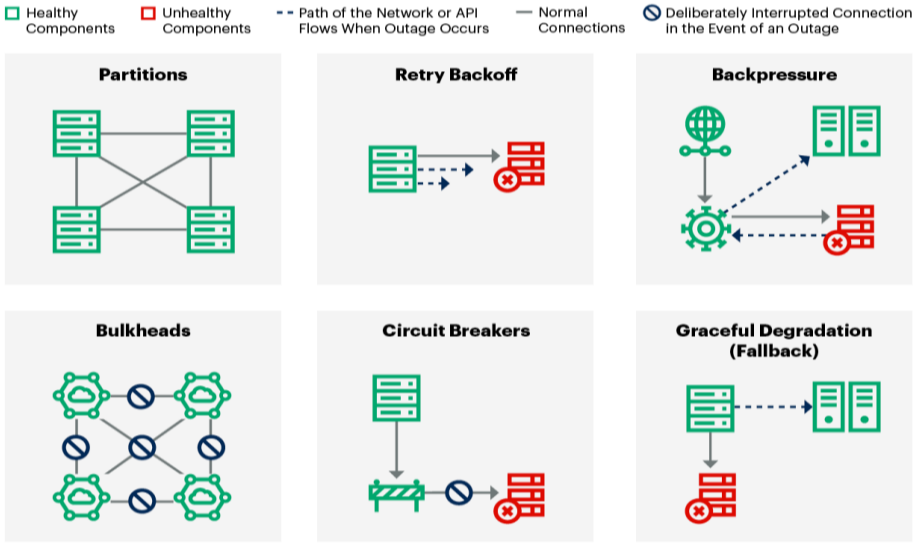
- **Real-Time Alerts**: Notifies when issues or failures occurs and
- **APM** Tracks application-level performance.
- **Tools**: Prometheus, Grafana, ELK Stack.



Health Checks & Circuit Breakers

Monitors the health of microservices to ensure they are running properly.

- **Readiness & Liveness Checks**: Ensures the service is ready to receive traffic and confirms the service is still alive and responsive.
- **Automated Recovery**: Can trigger restarts for unhealthy services.

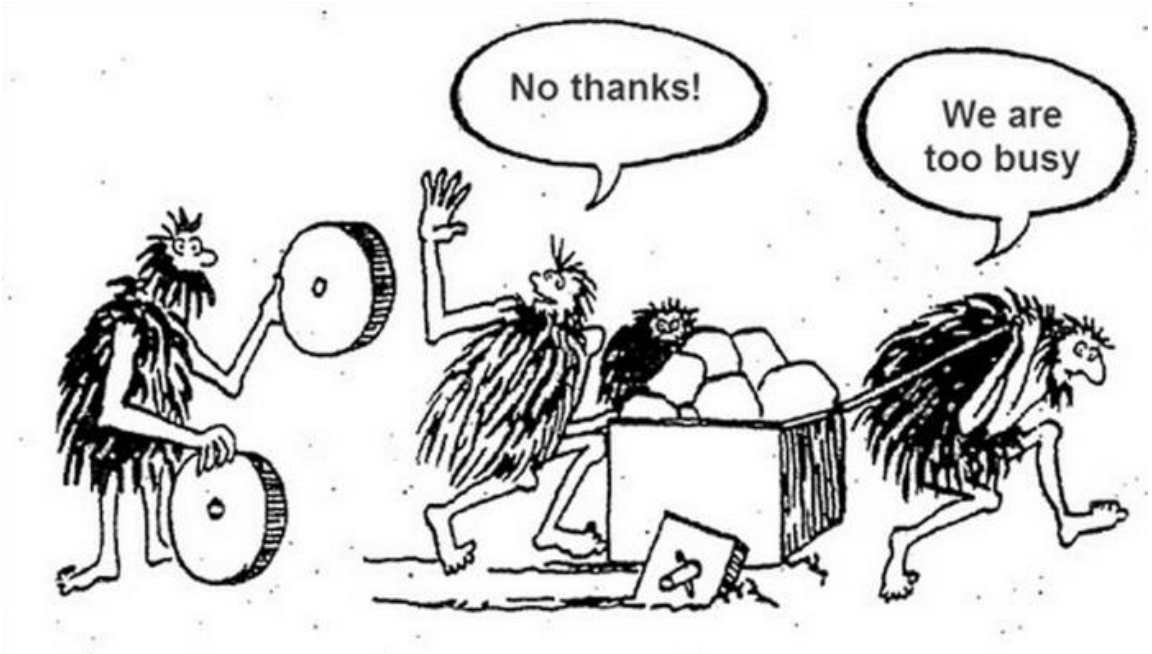


Prevents cascading failures by stopping calls to failing services

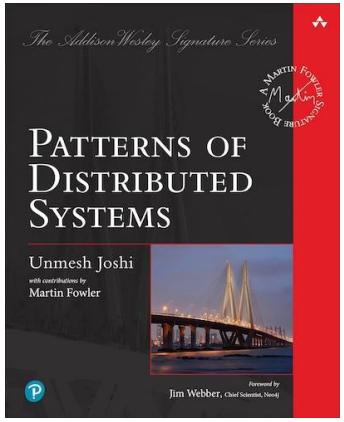
- **Failure Detection**: Identifies when a service is repeatedly failing.
- **Opens the Circuit**: Stops requests to a failing service to prevent overload.
- **Fallback Mechanism**: Routes traffic to alternative services or default responses.

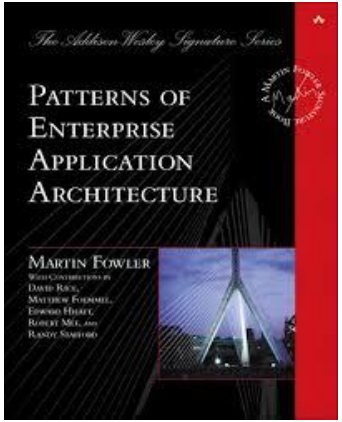
Garanti BBVA Technology

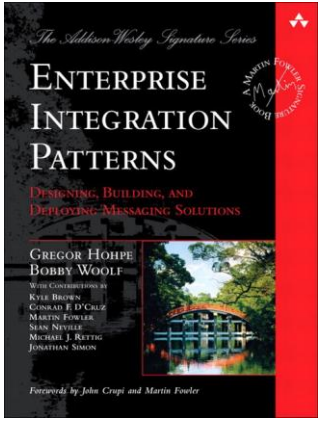
References

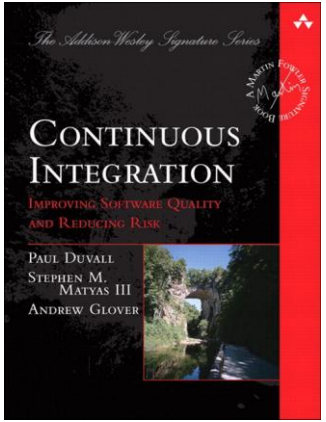

No thanks!
We are too busy

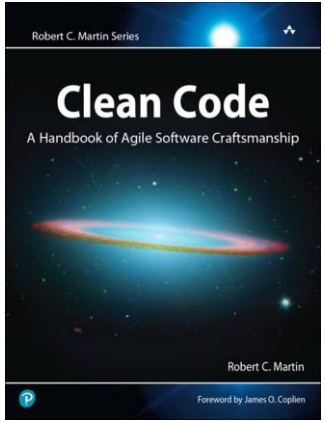
Recommended to Read (1)

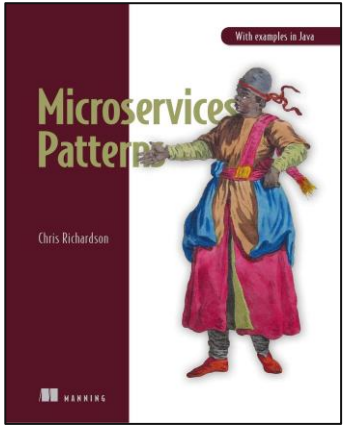


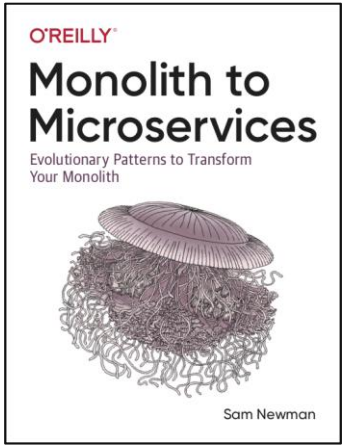


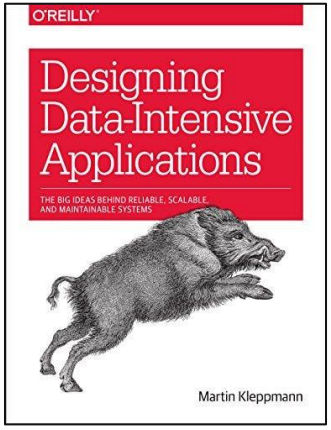


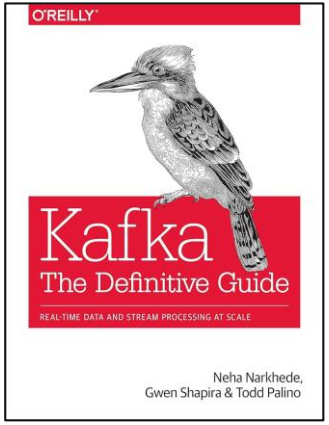


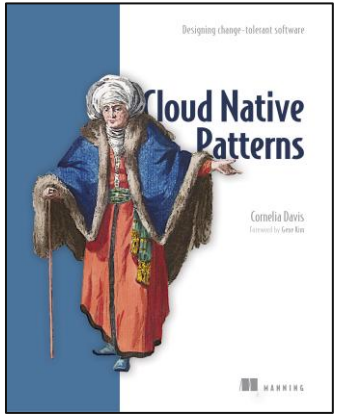




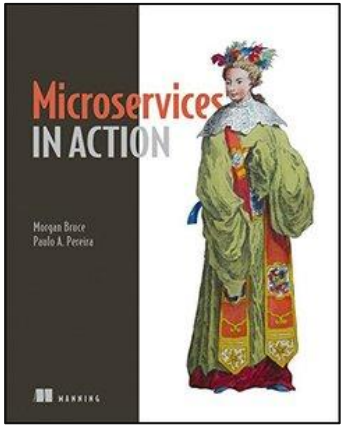


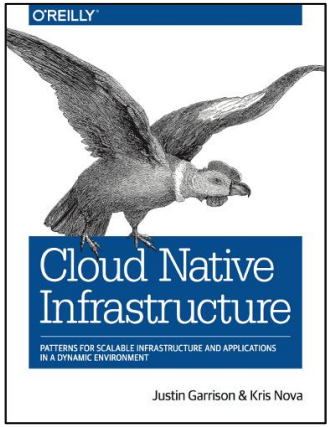


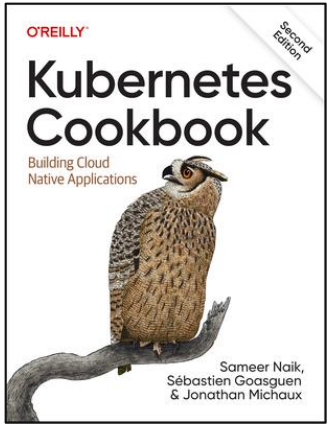


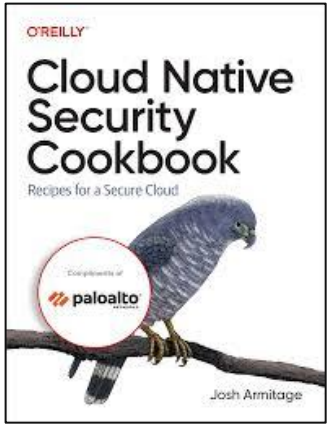


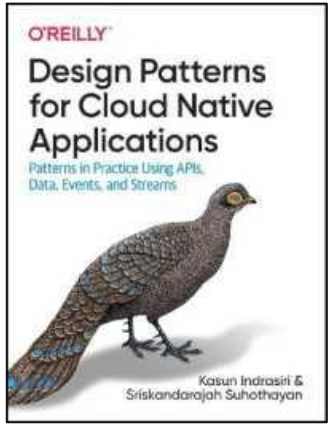
Recommended to Read (2)

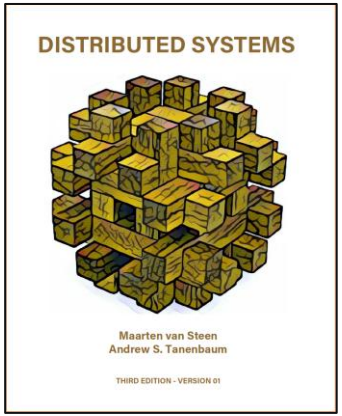


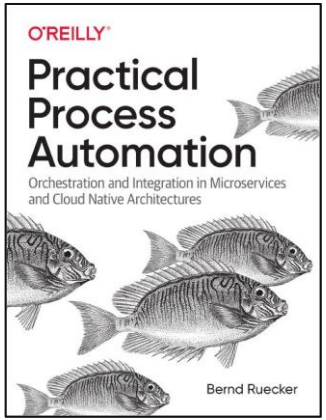


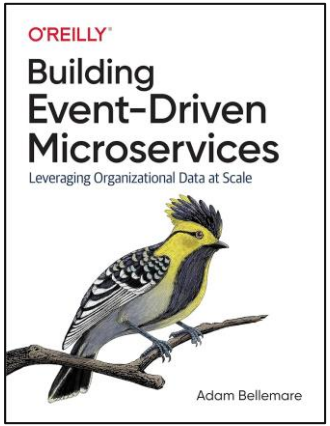


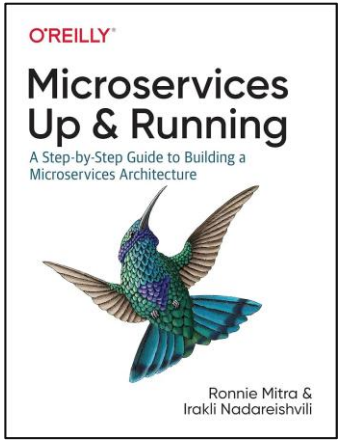


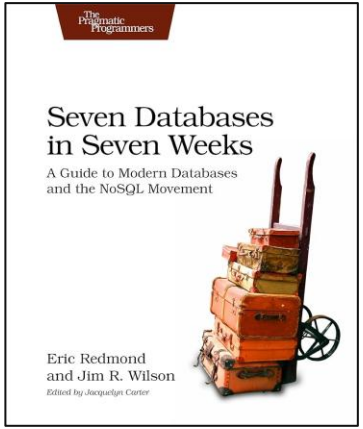




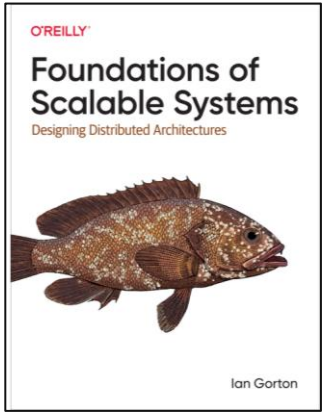


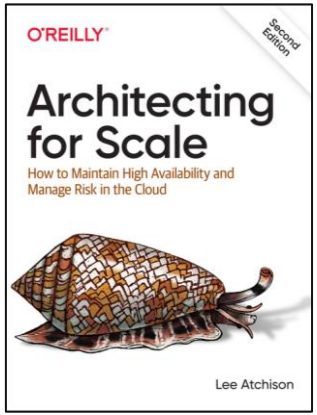


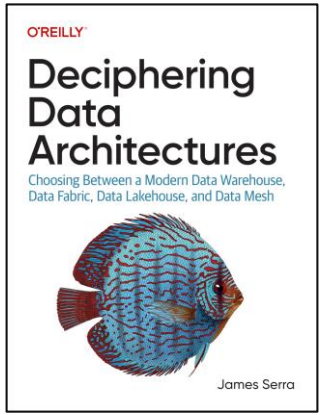


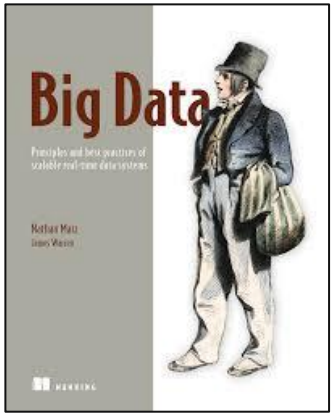


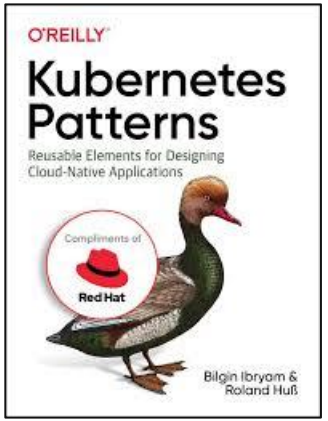
Recommended to Read (3)

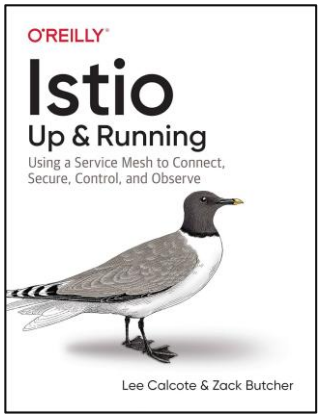


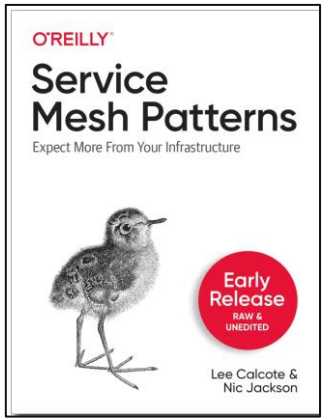


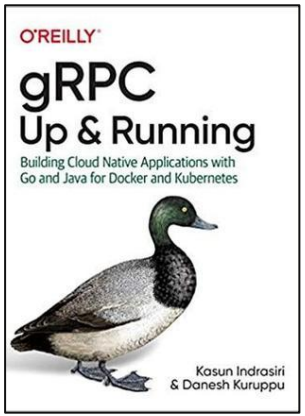


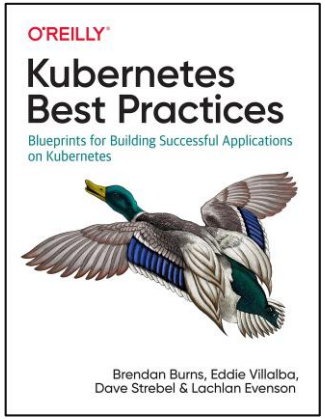


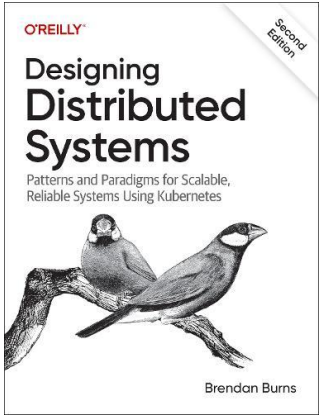












Garanti BBVA Technology

Thank You