Contents

[ACID 2](#_Toc146622924)

[CAP (Brewer's) theorem 2](#_Toc146622925)

[Service discovery 4](#_Toc146622926)

[Client-Side Service Discovery 4](#_Toc146622927)

[Server-Side Service Discovery 5](#_Toc146622928)

[Service Registration Options 6](#_Toc146622929)

[Self-Registration 6](#_Toc146622930)

[Third-party Registration 6](#_Toc146622931)

[Обработка ошибок 7](#_Toc146622932)

[Параллельные запросы + Idempotency key 7](#_Toc146622933)

[Retries 7](#_Toc146622934)

[Circuit breaker 8](#_Toc146622935)

[Fault tolerance in microservices 8](#_Toc146622936)

[Sidecar 9](#_Toc146622937)

[Strategies for ensuring data consistency and synchronization across microservices? 10](#_Toc146622938)

[Service mesh 11](#_Toc146622939)

[Role of Docker and Kubernetes 11](#_Toc146622940)

[KUBERNETES STRUCTURE 12](#_Toc146622941)

[Replication / Partitioning /Sharding 12](#_Toc146622942)

[Saga vs 2PC 13](#_Toc146622943)

[GraphQL vs GRPC vs REST 13](#_Toc146622944)

[When to use REST? 14](#_Toc146622945)

[When to use GraphQL? 15](#_Toc146622946)

[When to use gRPC? 15](#_Toc146622947)

[Event driven architecture 15](#_Toc146622948)

[Streaming vs Messaging 16](#_Toc146622949)

[EDA Considerations 16](#_Toc146622950)

[EDA Anti-Patterns 17](#_Toc146622951)

[Transaction isolation levels 17](#_Toc146622952)

[**Read uncommitted** 17](#_Toc146622953)

[**Read committed** 17](#_Toc146622954)

[**Repeatable read** 18](#_Toc146622955)

[**Serializable** 18](#_Toc146622956)

<https://github.com/checkcheckzz/system-design-interview#tips>

Eventsore

Snapshots -> Projections

# How to check quality of job done

Ask team for:

* Архитектура
* Архитектурный дизайн
* Диаграма компонентов

OLAP cubes

# ACID

Atomicity

Atomicity guarantees that each transaction is treated as a single "unit", which either succeeds completely or fails completely: if any of the statements constituting a transaction fails to complete, the entire transaction fails and the database is left unchanged.

Consistency

Consistency ensures that a transaction can only bring the database from one consistent state to another, preserving database invariants: any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof. This prevents database corruption by an illegal transaction.

Isolation

Isolation ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially.

Durability

once a transaction has been committed, it will remain committed even in the case of a system failure. completed transactions (or their effects) are recorded in non-volatile memory

# CAP (Brewer's) theorem

The CAP (Brewer's theorem states that in a distributed system, it is impossible to simultaneously achieve all three of the following guarantees:

1. **Consistency (C)**: Every read operation on the system returns the most recent write, or an error.
2. **Availability (A)**: Every request (read or write) to the system receives a response, without guaranteeing that it contains the most up-to-date data. The system is always responsive and accessible to clients.
3. **Partition Tolerance (P)**: works ok with communication failures between nodes.

Here are the possible trade-offs and implications:

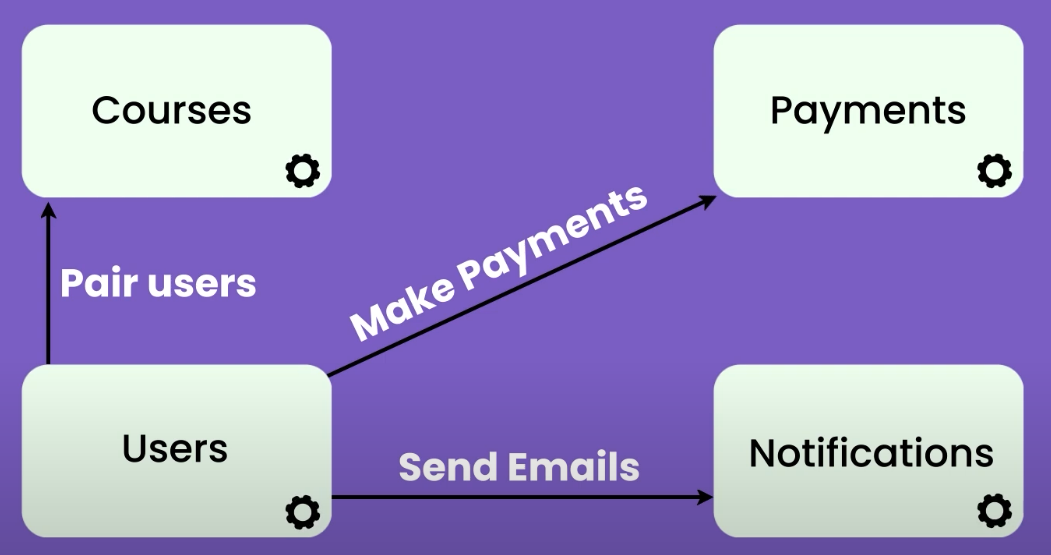
1. **CA Systems (Consistency and Availability)**: if data consistency and immediate availability are critical, you can sacrifice partition tolerance. These systems prioritize ensuring that all nodes have the most recent data and are available to serve requests.
2. **CP Systems (Consistency and Partition Tolerance)**: In scenarios where data consistency and fault tolerance are essential, you can sacrifice immediate availability. These systems prioritize maintaining data consistency even in the presence of network partitions, which may result in temporarily unresponsive nodes during partition events.
3. **AP Systems (Availability and Partition Tolerance)**: In scenarios where high availability and fault tolerance are paramount, you can sacrifice strong consistency. These systems prioritize always being available to serve requests, even if it means that different nodes might have slightly inconsistent views of the data during network partitions.

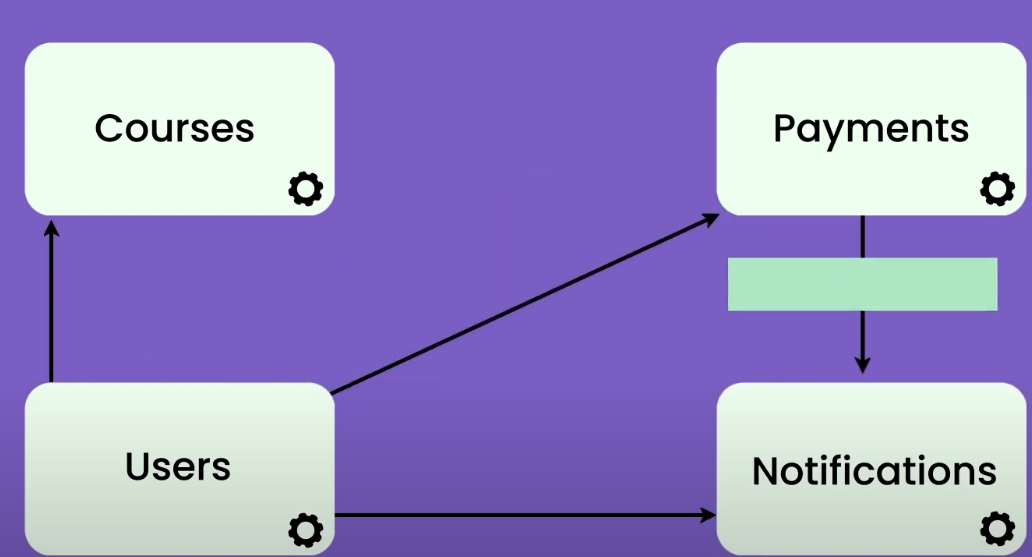
Balancing consistency, availability, and partition tolerance is a complex task that depends on the specific requirements of your distributed system and its use cases. Some strategies to achieve a balance include:

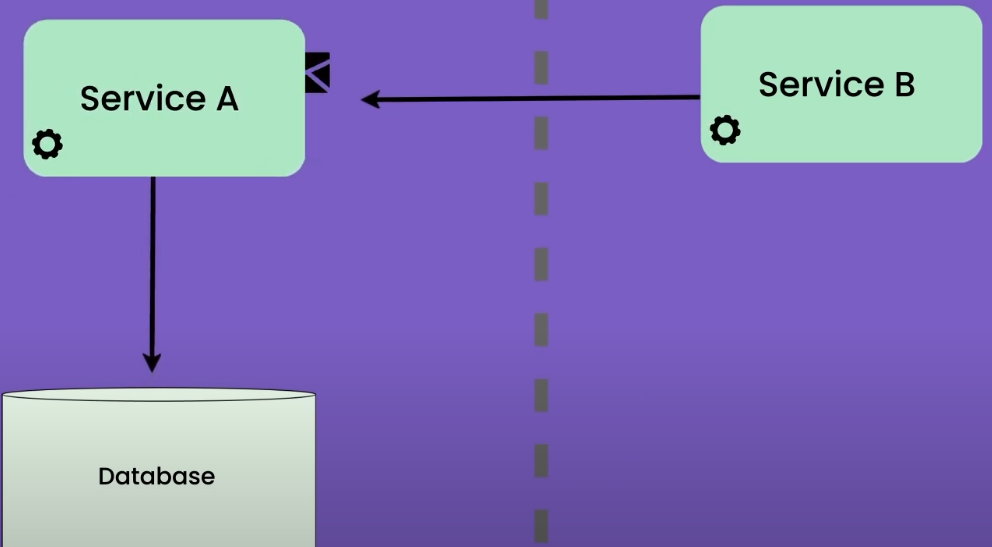
* **Tuning Configuration Parameters**: Depending on the distributed database or system you're using, you can adjust configuration parameters to favor one aspect of the CAP triangle over the others. For example, you can configure replication factors or consistency levels in databases like Apache Cassandra.
* **Architectural Choices**: You can design your system's architecture to mitigate the impact of network partitions, such as by using quorum-based approaches, leader-follower models, or consensus algorithms like Raft or Paxos.
* **Use Case-Specific Strategies**: Different use cases within the same distributed system might have varying requirements. You can apply different CAP trade-offs to different parts of your system based on their specific needs.
* **Hybrid Approaches**: In some cases, you might employ hybrid strategies that adapt dynamically to the network conditions. For example, during periods of network stability, you can prioritize consistency, and during network partitions, you can relax consistency to ensure availability.

Ultimately, achieving the right balance among consistency, availability, and partition tolerance requires careful consideration of your system's requirements and trade-offs. There's no one-size-fits-all solution, and the appropriate approach will depend on the specific goals and constraints of your distributed system.

# Coupling

Domain – сервисы зависят от запросов и результатов других сервисов 

Pass-through – пересылка данных из другого сервиса 

Content – просим данные через другой сервис 

Common – разные сервисы обращаются к одной базе. 1 Service – 1 Domain

# Service discovery

the dynamic process by which individual microservices can find and communicate with each other within a distributed system.   
Purposes:

1. **Dynamic Nature of Microservices:** Microservices are designed to be independently deployable and scalable. They may be distributed across multiple servers or containers. As a result, their locations (IP addresses, ports, etc.) can change frequently due to scaling, failures, or updates. Service discovery helps microservices adapt to this dynamic environment.
2. **Load Balancing:** Service discovery can be used to implement load balancing strategies. When multiple instances of a service are available, a service discovery mechanism can evenly distribute incoming requests among them, improving performance and fault tolerance.
3. **Failover and Redundancy:** if a microservice instance fails, service discovery can automatically redirect traffic to healthy instances, minimizing downtime and service disruption. This is critical for maintaining high availability.
4. **Service Composition:** Microservices often need to interact with other services to perform complex tasks. Service discovery helps them locate and establish connections with the required services, enabling seamless communication between microservices.
5. **Simplified Configuration:** Without service discovery, microservices would need to maintain a static list of IP addresses and ports for all the services they depend on. Service discovery simplifies this by allowing microservices to query a central registry or service discovery server to obtain the necessary information dynamically.
6. **Scaling:** As the number of microservices instances grows or shrinks based on demand, service discovery helps each microservice instance discover and communicate with its peers without manual intervention. This supports efficient auto-scaling.

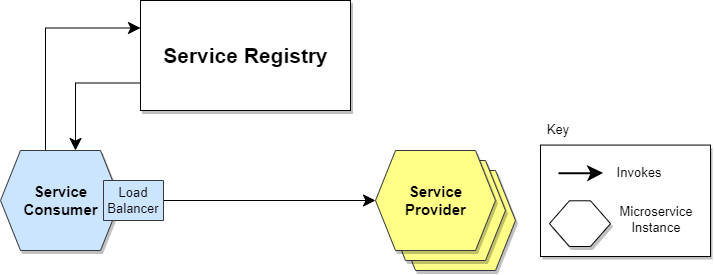
Service discovery can be implemented using various tools and approaches. Some popular options include:

* **Service Registries:** Maintaining a central registry that contains information about all available microservices, including their locations and health status. Tools like Consul, ZooKeeper, and Eureka are examples of service registry systems.
* **DNS-Based Discovery:** Using DNS to resolve service names to their corresponding IP addresses and ports. This approach can be simpler to set up but may have limitations in terms of load balancing and failover.
* **API Gateway:** Implementing service discovery at the API gateway level, where incoming requests are routed to the appropriate microservices based on a configuration or routing rules.

### Client-Side Service Discovery

When using Client-Side Discovery, **the Service Consumer is responsible for determining the network locations of available service instances and load balancing requests between them.** The client queries the Service Register. Then the client uses a load-balancing algorithm to choose one of the available service instances and performs a request.

The following diagram shows the pattern just described:

[](https://www.baeldung.com/wp-content/uploads/sites/4/2022/01/Service-Discovery-Client-Side.png)

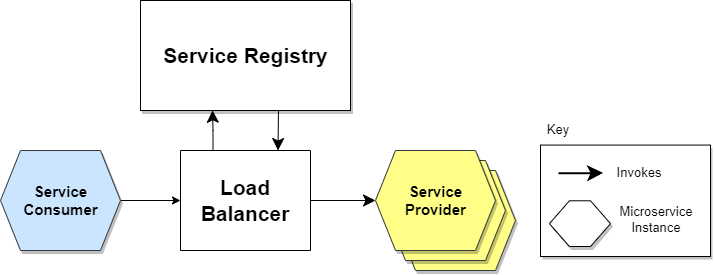
Giving responsibility for client-side load balancing is both a burden and an advantage.**It’s an advantage because it saves an extra hop that we would’ve had with a dedicated load balancer. It’s a disadvantage because the Service Consumer must implement the load balancing logic.**

We can also point out that **the Service Consumer and the Service Registry are quite coupled**. This means that Client-Side Discovery logic must be implemented for each programming language and framework used by the Service Consumers.

### Server-Side Service Discovery

**The alternate approach to Service Discovery is the Server-Side Discovery model, which uses an intermediary that acts as a**[**Load Balancer**](https://www.baeldung.com/zuul-load-balancing)**.** The client makes a request to a service via a load balancer that acts as an orchestrator. The load balancer queries the Service Registry and routes each request to an available service instance.

The following diagram shows how communication takes place:

[](https://www.baeldung.com/wp-content/uploads/sites/4/2022/01/Service-Discovery-Server-Side.png)

In this approach, a dedicated actor, the Load Balancer, does the job of load balancing. This is the main advantage of this approach. Indeed, **creating this level of abstraction makes the Service Consumer lighter, as it doesn’t have to deal with the lookup procedure.** As a matter of fact, there’s no need to implement the discovery logic separately for each language and framework that the Service Consumer uses.

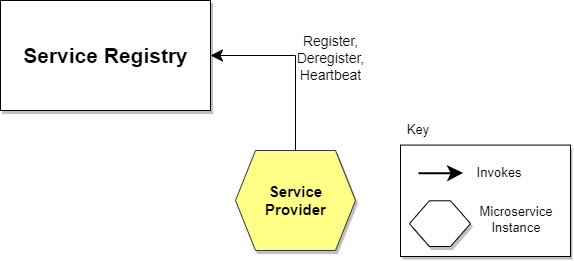
**On the other hand, we must set up and manage the Load Balancer, unless it’s already provided in the deployment environment.**

Now that we’ve delved into the different approaches to the discovery mechanisms, let’s move on to registration mechanisms.

## Service Registration Options

### Self-Registration

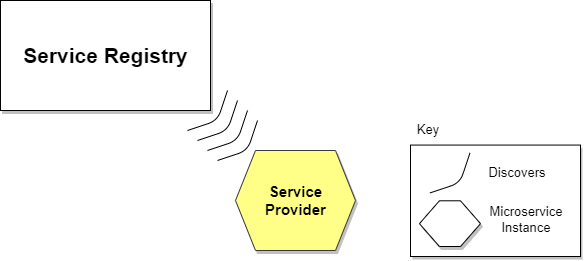
**When using the self-registration model, a service instance is responsible for registering and de-registering itself in the Service Registry.** In addition, if necessary, a service instance sends [heartbeat](https://martinfowler.com/articles/patterns-of-distributed-systems/heartbeat.html) requests to keep its registration alive. The following diagram shows the structure of this pattern:

[](https://www.baeldung.com/wp-content/uploads/sites/4/2022/01/Service-Discovery-Self-Registration.png)

The self-registration model has several pros and cons. One **advantage** is that it’s relatively simple and doesn’t require other system components as intermediaries. **However**, a significant disadvantage is that it couples service instances to the Service Registry, which means we must implement the registration code in each language and framework used.

### Third-party Registration

When using the third-party registration model, the service instances aren’t responsible for registration in the Service Registry. Instead, another system component known as the Service Register is responsible for registration. **The Service Register keeps track of changes to running instances by polling the deployment environment or subscribing to events.** When it detects a newly available service instance, it records it in its database. The Service Registry also de-registers terminated service instances. The following diagram illustrates this:

[](https://www.baeldung.com/wp-content/uploads/sites/4/2022/01/Service-Discovery-3rd-Registration.png)

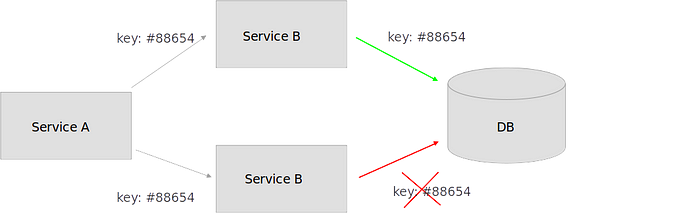
Like self-registration, the third-party registration scheme also has various pros and cons. One of the main advantages is that services are decoupled from the Service Registry. There’s no need to implement service registration logic for each programming language and framework. Instead, the registration of service instances is managed centrally within a dedicated service.

One disadvantage of this model is that, unless it’s embedded in the deployment environment, it’s yet another highly available system component that needs to be set up and managed.

# Обработка ошибок

## Параллельные запросы + Idempotency key

Idemp: GET, HEAD, PUT, DELETE.



## Retries

* **без ожидания (no delay)**, когда сразу без паузы повторяем отправку запроса
* **с константным значением (constant)**, когда устанавливаем строго заданный лимит
* **с линейным значением (linear)**
* **с экспоненциальным значением (exponencial)**

Использовать когда при работе с удаленным сервисом могут возникнуть временные ошибки.

## Circuit breaker

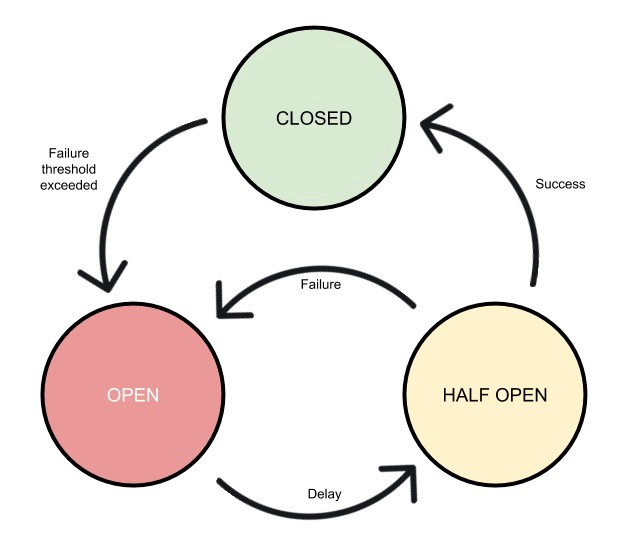
добавляет стабильности, когда система восстанавливается после падения и минимизирует влияние этого падения на производительность.

**Когда стоит использовать**

* Для предотвращения попыток обращения к сервису или разделяемым ресурсам, когда вероятность возникновения ошибки высока и эти ошибки имеют продолжительный характер.

**Когда не стоит использовать**

* Для обращения к приватным ресурсам приложения — это даст только дополнительный overhead
* Как замена обработки исключений бизнес-логики приложения



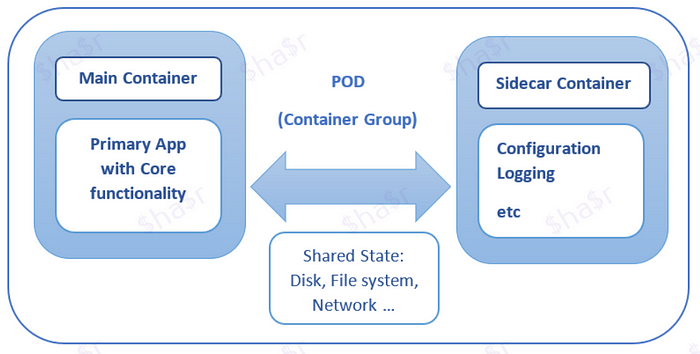
# Fault tolerance in microservices

1. **Use Redundancy:**
   * Deploy multiple instances of critical microservices to ensure redundancy. This can be within the same data center or across multiple data centers or cloud regions.
   * Employ load balancing to distribute traffic evenly among these instances, and set up health checks to detect and route away from unhealthy instances.
2. **Circuit Breaker Pattern:**
   * Implement the circuit breaker pattern to prevent cascading failures. A circuit breaker monitors the health of a service and temporarily blocks requests to that service if it is experiencing issues.
   * Provide a fallback mechanism or default response when the circuit breaker is open to maintain some level of service.
3. **Timeouts and Retries:**
   * Set appropriate timeouts for requests between microservices. Timeouts prevent long waits in case a service is unresponsive.
   * Implement retry mechanisms with exponential backoff to automatically retry failed requests. This can help when transient failures occur.
4. **Bulkheads:**
   * Apply the bulkhead pattern to isolate different parts of your application from each other. For example, allocate a separate thread pool for each microservice, so issues in one service won't affect others.
5. **Distributed Tracing and Monitoring:**
   * Use distributed tracing tools (e.g., OpenTelemetry, Zipkin) to track the flow of requests across microservices. This helps in diagnosing performance bottlenecks and failures.
   * Implement comprehensive monitoring with alerting. Monitor system metrics, logs, and application-specific metrics to detect and respond to anomalies quickly.

# Sidecar

Sidecar pattern is a single node pattern made up of two containers.

In microservice architecture, it’s very common to have multiple services/apps often require common functionalities like logging, configuration, monitoring & networking services. These functionalities can be implemented and run as a separate service within the same container or in a separate container.

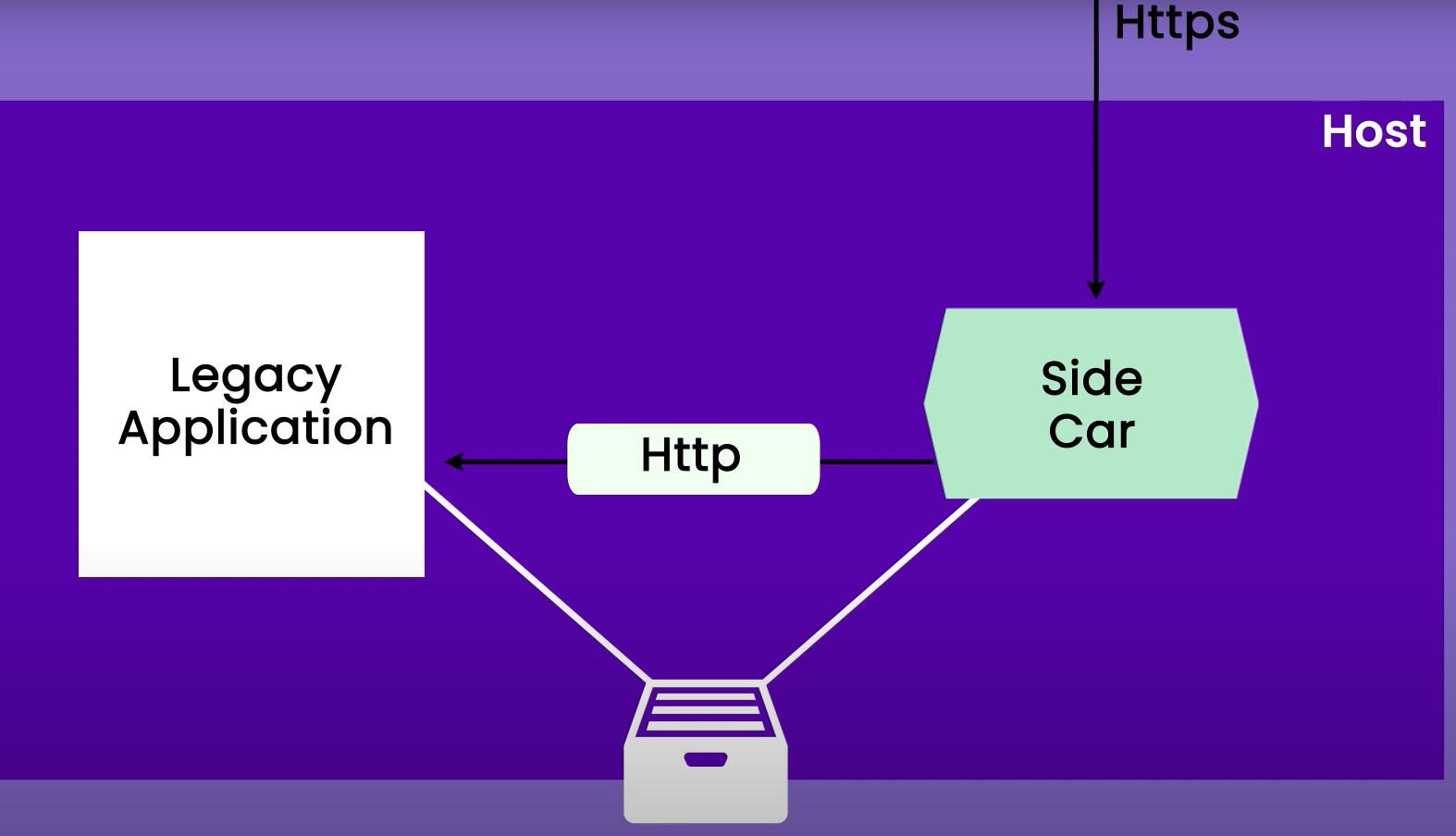


**When Sidecar pattern is useful ?**

* services/components are implemented with multiple languages or technologies.
* A service/component must be co-located on the same container group (pod) or host where primary application is running.
* A service/component is owned by remote team or different organization.
* A service which can be independently updated without the dependency of the primary application but share the same lifecycle as primary application.
* If we need control over resource limits for a component or service.

**Examples**:

1. Adding HTTPS to a Legacy Service  
2. Dynamic Configuration with Sidecars  
3. Log Aggregator with Sidecar

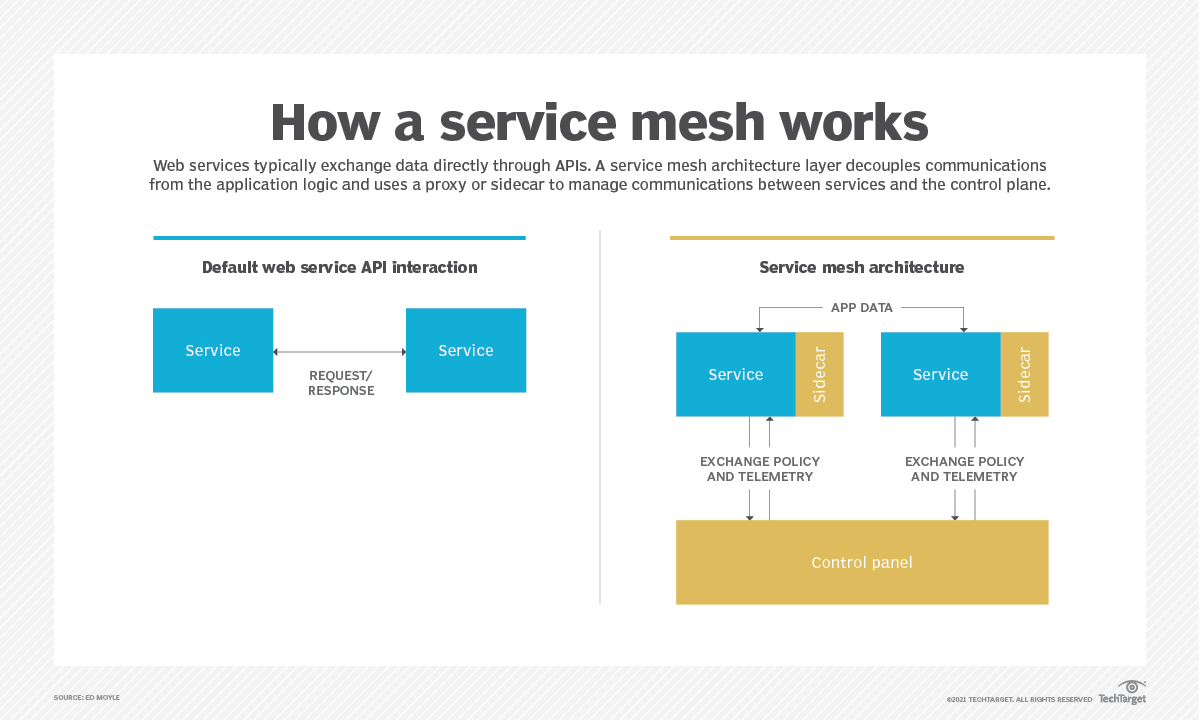


CQRS на минималках

# Strategies for ensuring data consistency and synchronization across microservices?

1. **Use a Single Source of Truth (SSOT):** Designate one microservice or database as the authoritative source for a particular type of data. Other microservices that need this data should query or subscribe to it rather than maintaining their copies. This ensures that there is only one version of truth for the data.
2. **Synchronous Communication:** In cases where strong consistency is crucial, use synchronous communication patterns such as request-response (e.g., HTTP) when accessing critical data. This allows one microservice to wait for a response from another, ensuring that the data is up-to-date before proceeding.
3. **Event-Driven Architecture:** Implement an event-driven architecture using message brokers like Apache Kafka, RabbitMQ, or AWS SNS/SQS. Microservices can publish events when data changes occur, and other services can subscribe to these events to update their local data stores accordingly. Eventual consistency can be acceptable in scenarios where real-time consistency is not mandatory.
4. **CQRS (Command Query Responsibility Segregation):** Separate the responsibility for handling commands (updates) from that of handling queries (reads). This allows you to optimize data storage and access patterns for each, potentially leading to better consistency for both. Event sourcing is often used in conjunction with CQRS to maintain a history of changes.
5. **Saga Pattern:** Implement long-running, multi-step transactions using the saga pattern. A saga is a sequence of local transactions across multiple microservices that ensure data consistency. If a step fails, compensating transactions can be executed to undo the changes made so far.

# Service mesh



# Role of Docker and Kubernetes

**1. Containers (e.g., Docker):**

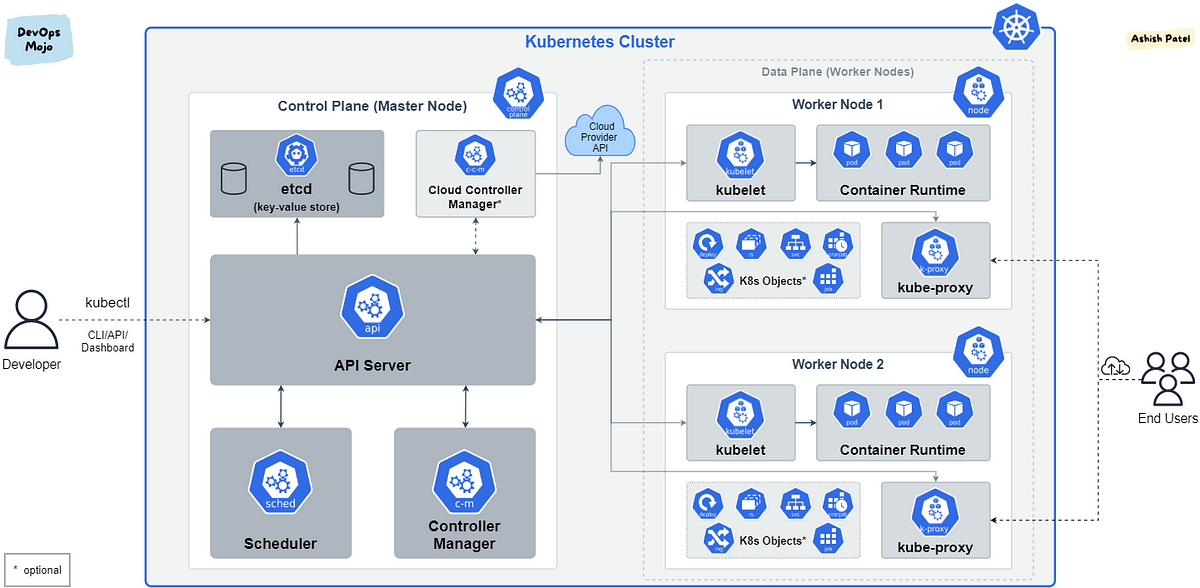
* **Isolation:** Containers provide a lightweight form of virtualization that isolates applications and their dependencies from the underlying infrastructure. Each microservice can run in its own container, ensuring that it has everything it needs to function correctly without interfering with other services.
* **Consistency:** Containers package the application code, runtime, libraries, and configuration into a single, consistent unit. This eliminates the "it works on my machine" problem and ensures that the microservice behaves the same way across different environments, from development to production.
* **Portability:** Containers are highly portable. You can run the same containerized microservice on different cloud providers or on-premises infrastructure, which simplifies deployment and reduces vendor lock-in.
* **Scalability:** Containers are well-suited for horizontal scaling. You can easily replicate containers to handle increased loads, and container platforms provide tools for managing container lifecycles.

**2. Container Orchestration (e.g., Kubernetes):**

* **Cluster Management:** manage clusters of machines where containers run. They automate tasks like **deploying, scaling, updating, and monitoring** containers across the cluster.
* **Service Discovery:** Container orchestration tools often include service discovery mechanisms that help microservices discover and communicate with each other. Kubernetes, for instance, offers DNS-based service discovery and load balancing.
* **Scaling:** Container orchestration platforms enable auto-scaling, which means you can automatically adjust the number of running microservice instances based on resource utilization or other metrics. This ensures that your application can handle varying workloads efficiently.
* **Rolling Updates:** Orchestrators facilitate rolling updates, allowing you to update microservices without causing downtime. New containers can be gradually rolled out while old ones are phased out, ensuring a smooth transition.
* **Health Checks and Self-healing:** Container orchestration platforms continuously monitor the health of microservice instances. If an instance becomes unhealthy (e.g., due to crashes or unresponsiveness), the orchestrator can automatically replace it with a healthy one.
* **Resource Management:** Orchestrators allocate resources (CPU, memory, etc.) to containers based on defined policies. This ensures that microservices have the necessary resources to operate efficiently.
* **Configuration Management:** Many orchestration tools support configuration management, allowing you to manage environment-specific configurations for microservices. This is particularly important in microservices deployments where each service may have unique requirements.
* **Rollback:** In case of issues during updates or deployments, container orchestrators offer rollback mechanisms to revert to a previous, stable state of the application.

Bulkheads

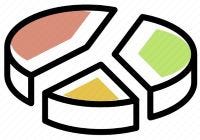
# KUBERNETES STRUCTURE



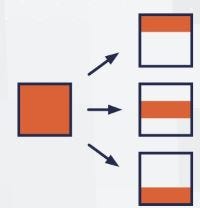
# Replication / Partitioning /Sharding



**Replication**(Copying data)— Keeping a copy of same data on multiple servers that are connected via a network.



**Partitioning**— Splitting up a large monolithic database into multiple smaller databases based on data cohesion. e.g. Horizontal (sharding) and Vertical (increase server size) partitioning.



**Sharding**(Horizontal Partitioning)— A type of horizontal partitioning that splits large databases into smaller components, which are faster and easier to manage. In other words — Splitting up a large table of data horizontally i.e. row-wise.

# Saga vs 2PC

**The Two-Phase Commit (2PC)** ensure that all nodes in a distributed**system agree to commit or abort a transaction**.

It works by coordinating transactions between a coordinator node and multiple participant nodes. The coordinator sends a request to the participants to prepare for the transaction, and once all participants respond with a positive acknowledgement, the coordinator sends a commit message to each participant to **commit the transaction**.

If any participant fails to respond or sends a negative acknowledgement, the coordinator sends an abort message to all participants, and the transaction is rolled back.

**2PC guarantees that all nodes will commit or abort a transaction, but it can be slow and vulnerable to failure.**

**SAGA** Pattern is for **managing long-lived transactions in a distributed system**. A saga is a sequence of local transactions, where each local transaction updates the state of a single service, and each service has its own database.

Each local transaction is an atomic operation that either completes successfully or compensates for its effects. If a local transaction fails, the saga executes a compensating transaction that undoes the effects of the failed transaction.

# GraphQL vs GRPC vs REST

**REST**

Representational State Transfer (REST) is all about resources: each resource has its own unique URI, and you can interact with it using a set of standard HTTP verbs (GET, POST, PUT, DELETE, etc.).

One of the biggest strengths of REST is its **simplicity**, plus, because REST is **stateless (no sessions)**, there are a lot of problems we simply get to ignore and it’s **great** for building **scalable systems**.

BUT **inefficient with large amounts of data**. Because each **resource** has its **own URI**, you often need to make multiple requests to fetch all the data you need.

**GraphQL**

is a query language for APIs allows you to retrieve all the data you need in a single request. This makes it much more efficient, especially when dealing with large amounts of data.

Another advantage of GraphQL is its flexibility. With REST, the server dictates what data is returned, but with GraphQL, the client specifies what data it needs.

More difficult to optimize your API for performance.

**gRPC** (Remote Procedure Call)

uses a **binary format for its messages**, which makes it much more efficient than REST or GraphQL, especially for large payloads. Second, gRPC uses protocol buffers for its serialization, which is a language-agnostic way of serializing structured data. This means you can easily generate client libraries in a variety of languages.

Another advantage is the support **for bidirectional streaming**. Both the client and server can send and receive data simultaneously (real-time chat or gaming). **REST** and **GraphQL** **don’t** have native support for it.

Harder to implement and debug due to binary format.

## When to use REST?

REST is a great choice for situations where:

* a simple API with a limited number of endpoints.
* data is structured in a hierarchical manner (such as with a traditional SQL database).
* you need a stateless API that can be easily cached.
* want to use HTTP’s built-in caching and authentication mechanisms.

Some common use cases for REST include:

* Retrieving information about a single resource (such as a user or product).
* Submitting data to the server (such as updating a user’s information).
* Retrieving a collection of resources (such as a list of products).

## When to use GraphQL?

GraphQL is ideal for situations where:

* You have a complex API with a large number of endpoints.
* Your data is not structured in a hierarchical manner (such as with a NoSQL database).
* You want to provide clients with the ability to request only the data they need.

Some common use cases for GraphQL include:

* Building a client-side application that needs to retrieve data from multiple sources.
* Providing a flexible API that can evolve over time without breaking existing clients (while this is also possible with REST, it’s much easier to do with GraphQL).
* Allowing clients to query for exactly the data they need, reducing the amount of data transmitted over the network. Which can be a major benefit if your app is used on slow mobile networks, or by users located in remote areas where the connection isn’t always that stable.

## When to use gRPC?

gRPC shines in situations where:

* You need a high-performance API that can handle a large number of requests.
* You need to transmit large amounts of data between the client and server.
* You want to use a strongly-typed API that can be generated automatically.
* You want to use bi-directional streaming to enable real-time communication between the client and server.

Some common use cases for gRPC include:

* Building a microservices architecture that requires fast and efficient communication between services.
* Developing real-time applications that require bi-directional communication between the client and server.
* Building APIs for resource-constrained environments where bandwidth is limited.

# Event driven architecture

<https://hackernoon.com/best-practices-for-event-driven-microservice-architecture-e034p21lk>

<https://medium.com/digitalfrontiers/the-good-the-bad-and-the-ugly-how-to-choose-an-event-store-f1f2a3b70b2d>

<https://medium.com/nerd-for-tech/apache-kafka-quick-start-1edeef53773e>

# Streaming vs Messaging

**Message Processing**

In traditional message processing, a component creates a message then sends it to a specific (and typically single) destination. The receiving component, which has been sitting idle and waiting, receives the message and acts accordingly. Typically, when the message arrives, the receiving component performs a single process. Then,  the message is deleted.

A typical example of a message processing architecture is a Message Queue. Though most newer projects use stream processing (as described below), architectures using message (or event) queues are still popular. Message queues typically use a “store and forward” system of brokers where events travel from broker to broker until they reach the appropriate consumer.[ActiveMQ](https://activemq.apache.org/?ref=hackernoon.com) and[RabbitMQ](https://www.rabbitmq.com/?ref=hackernoon.com) are two popular examples of message queue frameworks. Both of these projects have years of proven use and established communities.

**Stream Processing**

On the other hand, in stream processing, components emit events when they reach a certain state. Other interested components listen for these events on the event stream and react accordingly. Events are not targeted to a certain recipient, but rather are available to all interested components.

In stream processing, components can react to multiple events at the same time, and apply complex operations on multiple streams and events. Some streams include persistence where events stay on the stream for as long as necessary.

With stream processing, a system can reproduce a history of events, come online after the event occurred and still react to it, and even perform sliding window computations. For example, it could calculate the average CPU usage per minute from a stream of per-second events.

One of the most popular stream processing frameworks is[Apache Kafka](https://kafka.apache.org/?ref=hackernoon.com). Kafka is a mature and stable solution used by many projects. It can be considered a go-to, industrial-strength stream processing solution. Kafka has a large userbase, a helpful community, and an evolved toolset. + Pulsar

# EDA Considerations

**Discovering Event Information**

One of the greatest challenges in event-driven architecture is cataloging services and events. Where do you find event descriptions and details? What is the reason for an event? What team created the event? Are they actively working on it?

**Dealing with Change**

Will an event schema change? How do you change an event schema without breaking other services? How you answer these questions becomes  critical as your number of services and events grows.  
  
Being a good event consumer means coding for schemas that change. Being a good event producer means being cognizant of how your schema changes impact other services and creating well-designed events that are documented clearly.

# EDA Anti-Patterns

As with most architectures, an event-driven architecture comes with its own set of anti-patterns. Here are a few to watch out for.

**Too much of a good thing**

Creating too many events will create unnecessary complexity between the services, make deployment and testing more difficult, and cause congestion for event consumers. Not every method needs to be an event.

**Generic events**

Events should be specific in that they model a single business process. It should be possible to understand what an event means from the title alone – e.g. order placed.

**Complex dependency graphs**

Watch out for services that depend on one another and create complex dependency graphs or feedback loops. Each network hop adds additional latency to the original request, particularly north/south network traffic that leaves the datacenter.

**Depending on guaranteed order, delivery, or side effects**

Events are asynchronous; therefore, including assumptions of order or duplicates will not only add complexity but will negate many of the key benefits of event-based architecture. If your consumer has side effects, such as adding a value in a database, then you may be unable to recover by replaying events.

**Premature optimization**

Most products start off small and grow over time. While you may dream of future needs to scale to a large complex organization, if your team is small then the added complexity of event-driven architectures may actually slow you down. Instead, consider designing your system with a simple architecture but include the necessary separation of concerns so that you can swap it out as your needs grow.

**Expecting event-driven to fix everything**

On a less technical level, don’t expect event-driven architecture to fix all your problems. While this architecture can certainly improve many areas of technical dysfunction, it can’t fix core problems such as a lack of automated testing, poor team communication, or outdated dev-ops practices.

# Transaction isolation levels

### **Read uncommitted**

Ничего не изолирует, но быстрый. грязное чтение

### **Read committed**

исполняющиеся транзакции видят только зафиксированные изменения из других транзакций.   
Нет **грязного чтения**.

Но есть   
**неповторяющееся чтение**, когда мы видим обновленные и удаленные строки (UPDATE, DELETE),   
**чтения фантомов**, когда мы видим добавленные записи (INSERT).

### **Repeatable read**

Нет **неповторяющегося чтения**. Т.е. не видим в исполняющейся транзакции измененные и удаленные записи другой транзакцией. Но все еще видим вставленные записи из другой транзакции. **Чтение фантомов** никуда не уходит.

### **Serializable**

Уровень, при котором транзакции ведут себя как будто ничего более не существует, никакого влияния друг на друга нет. В классическом представлении этот уровень избавляет от эффекта **чтения фантомов**.

Лок на таблицу