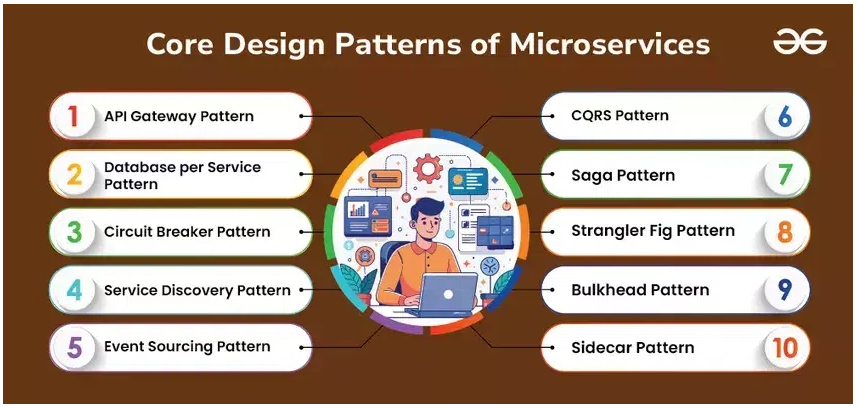
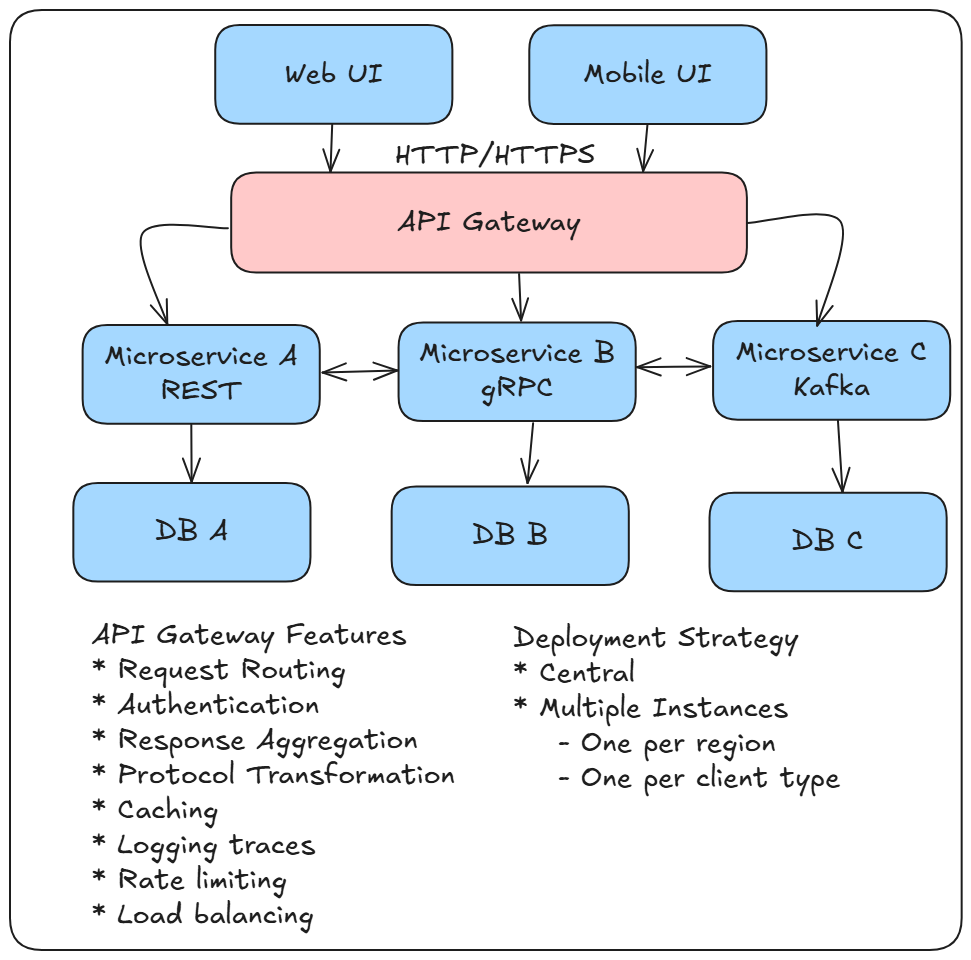
Microservices Design Pattern



# **API Gateway pattern:**

Main responsibility of this pattern is to provide a roadmap on how requests are handled, approved, rejected or re-routed.



# **Database per service**

Database per service means each microservice will have it’s own database.

Multiple ways to implement this pattern,

* Private tables per service: tables are created for each microservices in shared database. However, these tables are not shared between different microservices.
* Schema per service: each microservice has it’s own database schema in shared database. However these schemas are not shared between different microservices
* Database server per service: each microservice has it’s own database server.

All services share data through API and not directly through database queries.

Advantages:

* Each microservice can be developed, scaled and deployed independently
* Eliminate risk of affecting other services if one service changes

Challenges:

* Ensuring data consistency is a challenge as data is distributed across various tables/databases
* Distributed transactions become complex which required eventual consistency model
* Data duplication
* Cross service Queries are tricky are we may have to call multiple services to fetch required data

To handle these challenge we can implement event driven architecture where we have an event broker where each service will send it’s event when updated and subscribers to that event broker will be updated as soon as event is published.

Steps to implement pattern,

1. Designing the service boundaries
   1. Identify business domains
   2. Define service boundaries: Ensure each service has defined responsibility and does not overlap with others.
   3. Establish APIs
2. Select Database technologies for each service
   1. Evaluate requirements
   2. Database type
   3. Ensure compatibility
3. Define data ownership and Schema design
   1. Design schemas
   2. Define data ownership
   3. Avoid shared data
4. Implement communication between services
   1. Communication method: Synchronous (REST, gRPC) vs Asynchronous(Message queue, event streaming)
   2. API Gateways
   3. Message brokers
5. Handle Data consistency and synchronization
   1. Consistency methods: Strong consistency (e.g. distributed transactions) or eventual consistency (e.g. using event sourcing)
   2. Event sourcing: capture and store all changes to the data as sequence of events. This helps in maintaining consistency and recovering from failure
   3. Data synchronization: use techniques as data replication, synchronization services or periodic batch updates
6. Monitor and manage database for each service
   1. Setup monitoring: use metrics like query performance, resource utilization and error rates.
   2. Implement backup and recovery
   3. Manage resources: monitor and manage db resources to handle changing load and ensure optimal performance

Data management techniques

Techniques for managing data consistency

1. Eventual consistency
2. Distributed transactions
3. SAGA pattern

Strategies for handling data replication and synchronization

1. Data replication
2. Data synchronization

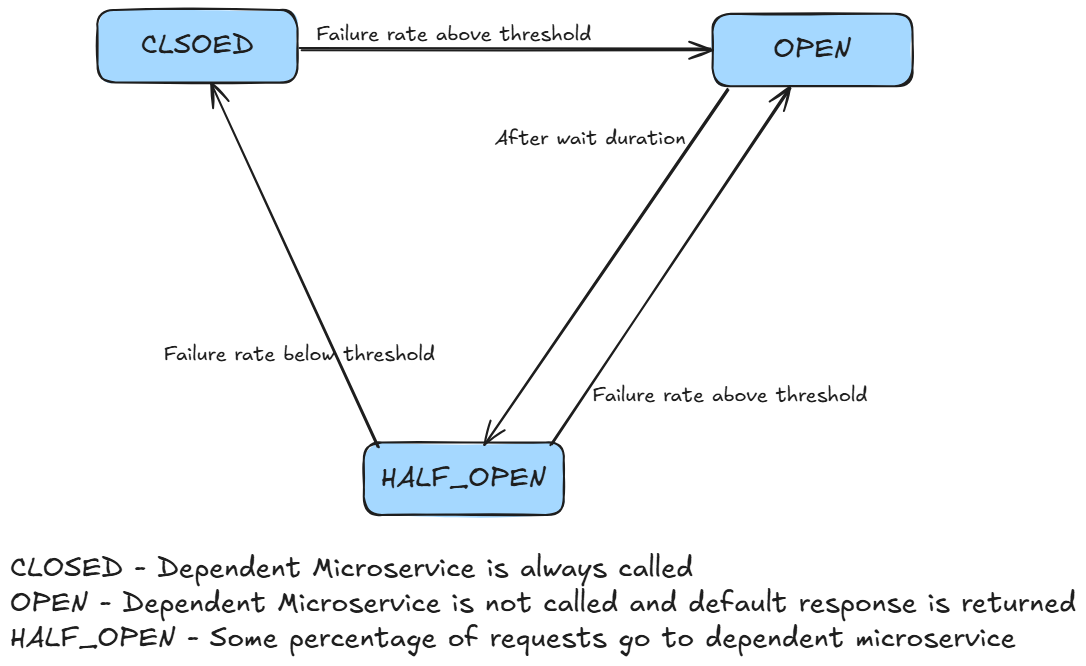
Approaches to handle cross services queries and reporting

1. CQRS (Command Query responsibility segregation)
2. Data lakes
3. Materialized views

# **Circuit Breaker**

Circuit breaker is used to enhance system resilience and fault tolerance.

Circuit breaker pattern states,



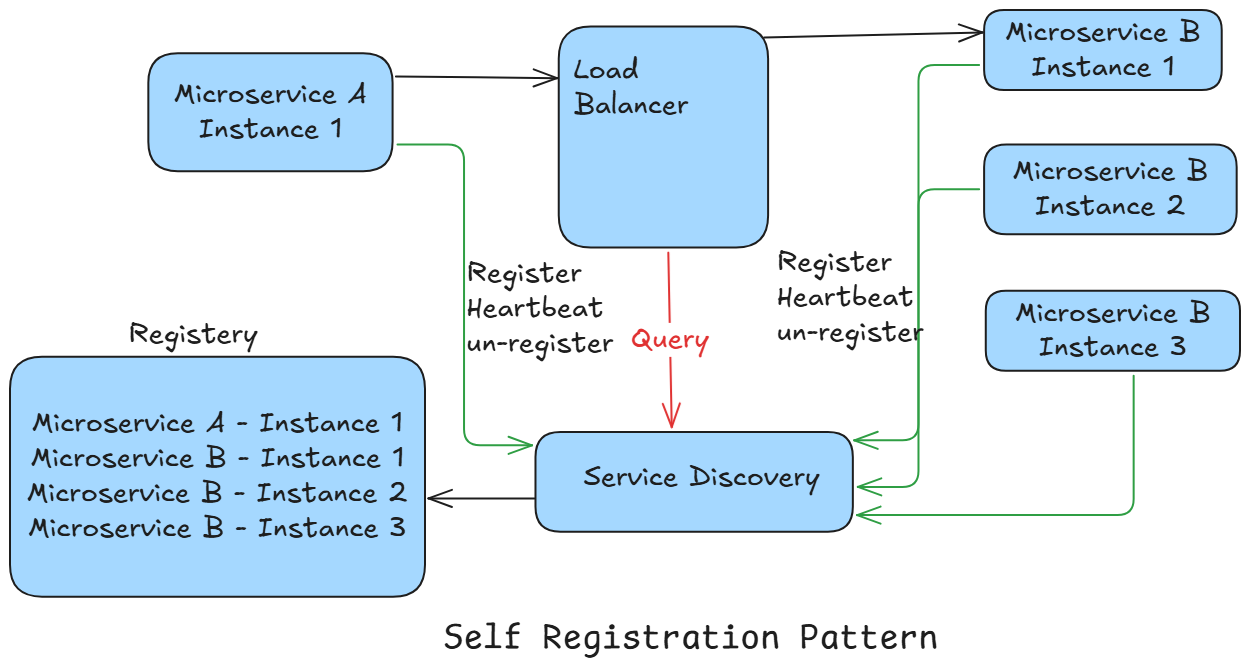
Benefits:

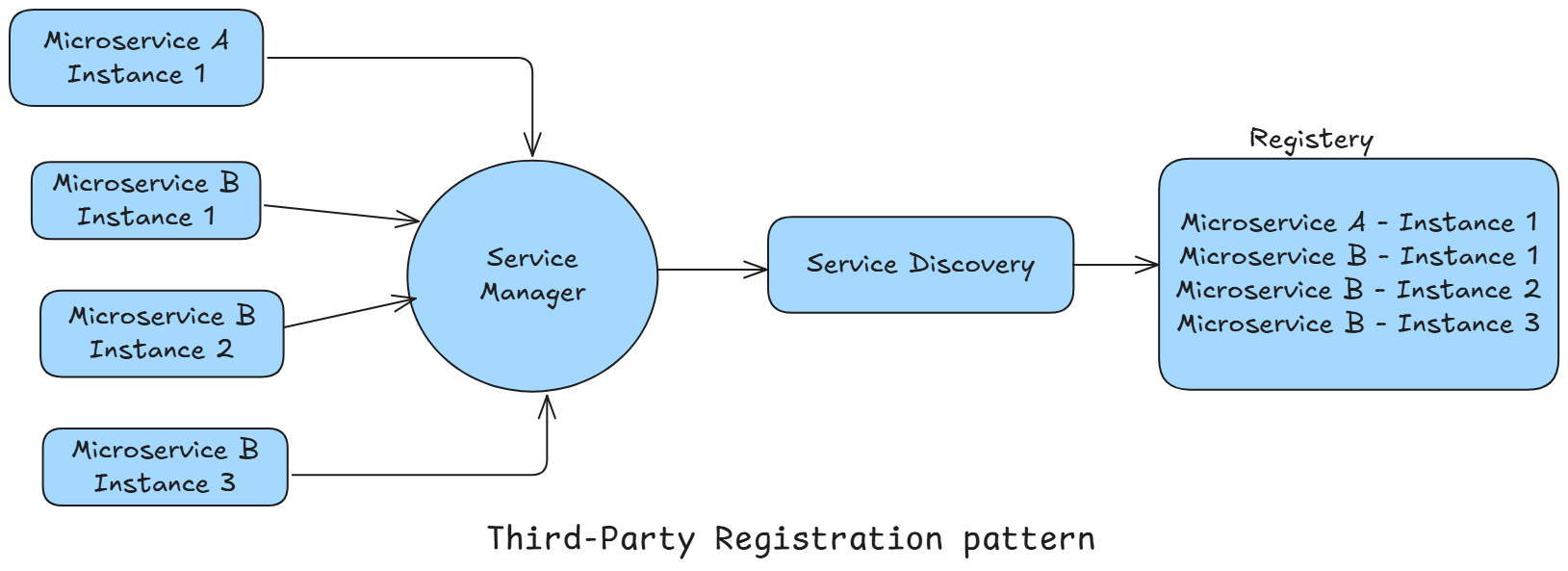
* By stopping calls to failing service, the CB helps prevent entire system from being overwhelmed.
* Allows applications to handle failures gracefully by returning fallback response.
* When a service fails, CB provides a cool down time for service to recover from failures before attempting to connect.
* Improves user experience by minimizing response time due to failures.

Challenges:

* Adds complexity by adding extra layer. Developer has to manage it’s states and integration
* Proper tuning of params like timeout, failure thresholds and recovery period can be tricky.
* When multiple services use CB, understanding interdependencies and potential point of failures can become complex.

# **Service Discovery**

****

****

Above is example of server-side service discovery. Because, client (MS A) makes a request to server which could be LB, Router or middleware and server makes a call to service discovery component.

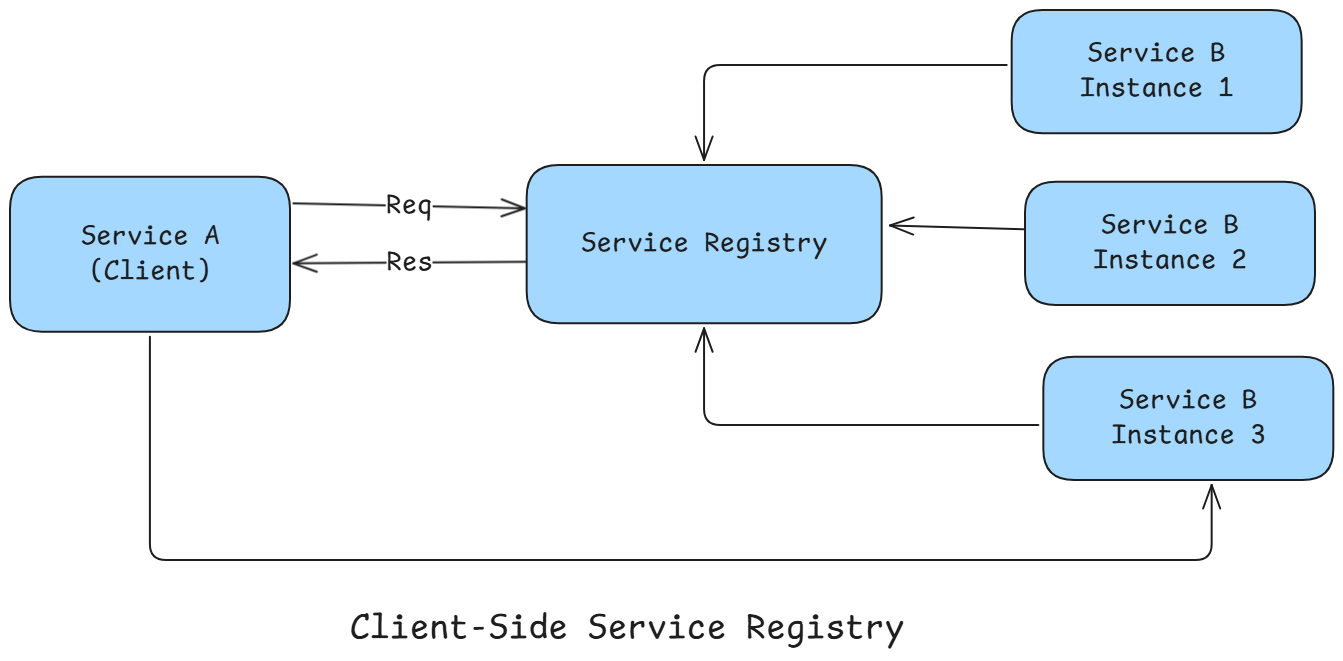
Advantages:

* LB does job of routing requests so consumer becomes lightweight and has only 1 uri to call
* No need to implement client-side service discovery logic

Disadvantages:

* Need to maintain LB

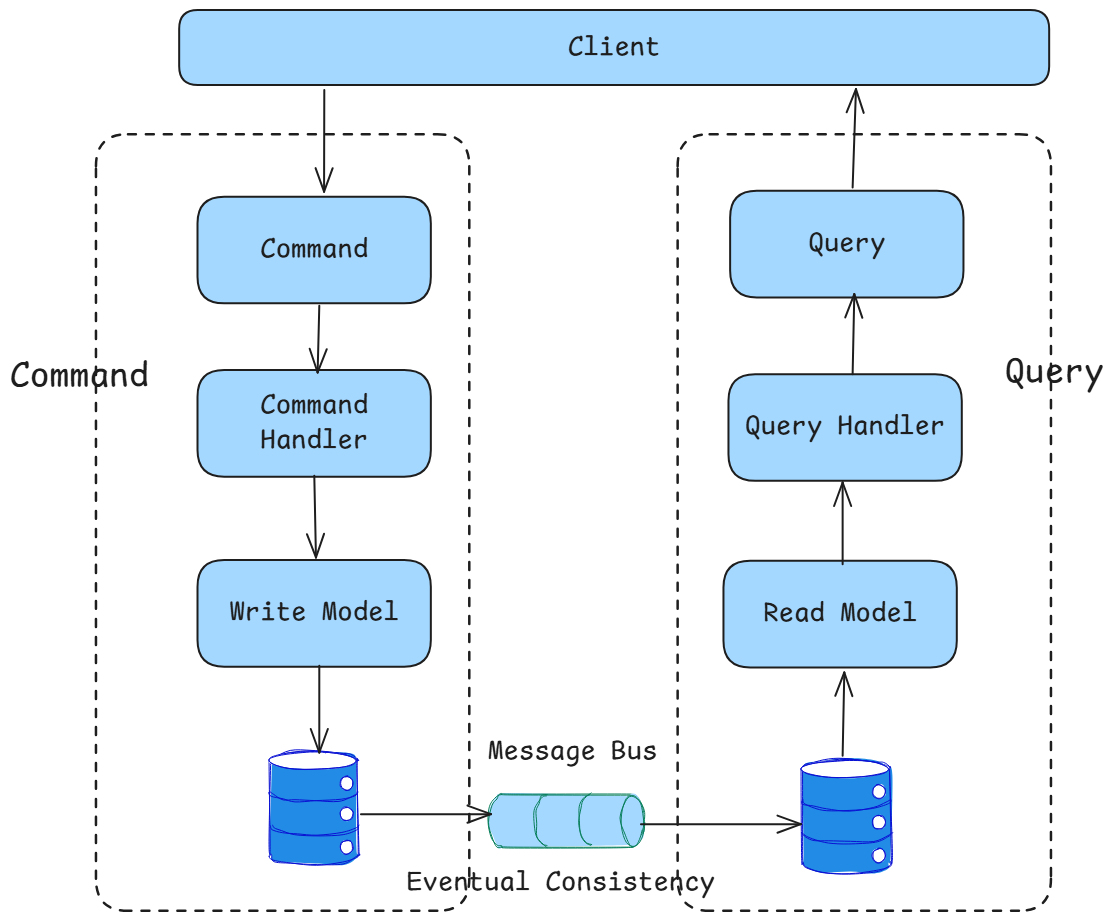
Client-Side registry



# **Event Sourcing**

To be covered later

# **CQRS – Command Query Responsibility Segregation pattern**



Separating the methods for reading and publishing data is primary goal of CQRS.

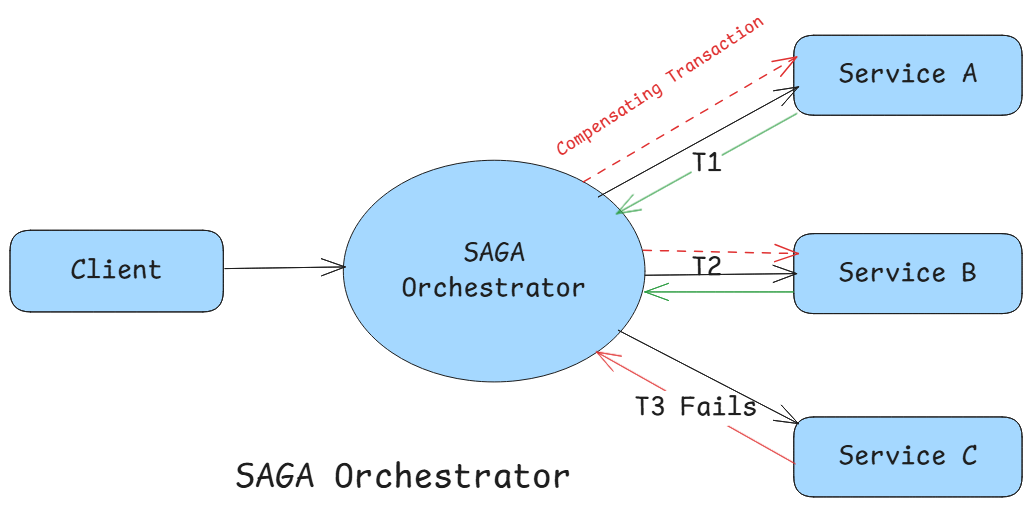
Basic Architecture:

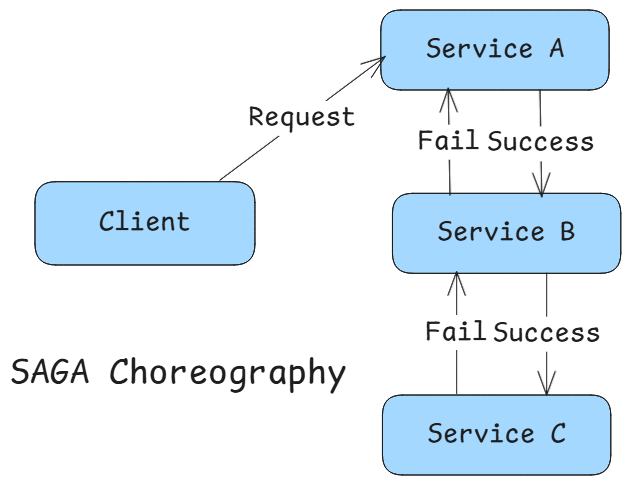
* Commands: Commands are the instructions that indicate desired change in the state of an entity. These commands execute operations like insert, update and delete. Commands do not return any data.
* Command Handlers: CommandHandlers interpret these commands and return an event
* Queries: Queries are used to retrieve information from database.

When to use CQRS?

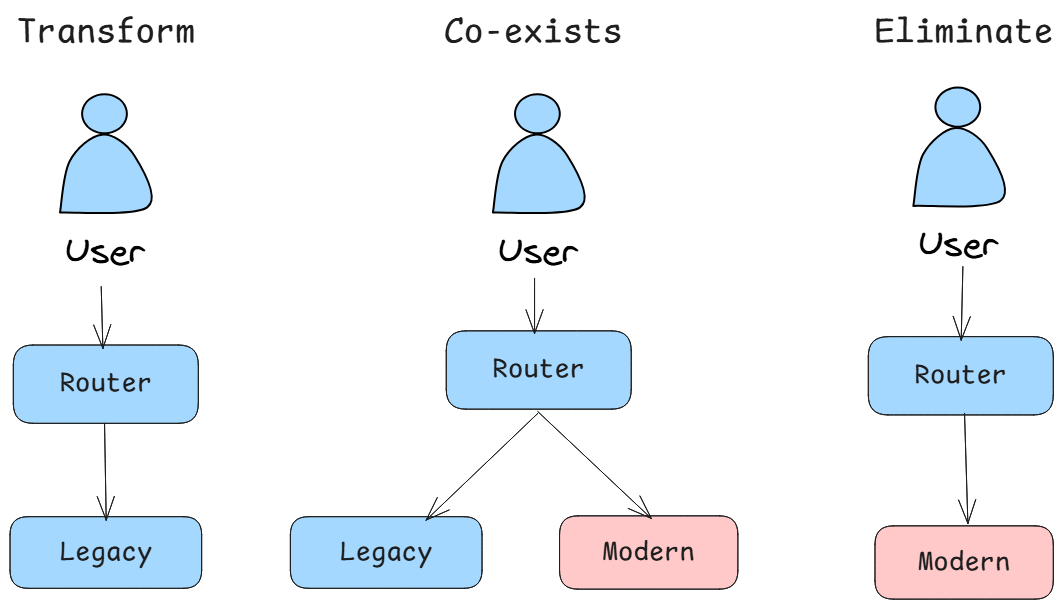
* Handling complex queries
* Scalability
* Event-driven systems
* When write data model different from the way read data model

# **SAGA Pattern**





# **Strangler pattern**



Strangler pattern is employed during migration from monolithic system to microservice architecture.

It has 3 steps,

* Transform
* Co-exists
* Eliminate

Features,

* Gradual migration: this pattern enables step-by-step migration of system into microservices. It allows orgs to replace specific functionalities incrementally.
* Coexistence: during migration process both monolithic and microservice co-exists ensuring uninterrupted functionality
* Strangling behaviour: it gradually replaces components/modules of monolith application with microservices leading to eventual replacement of monolith application.

Advantages:

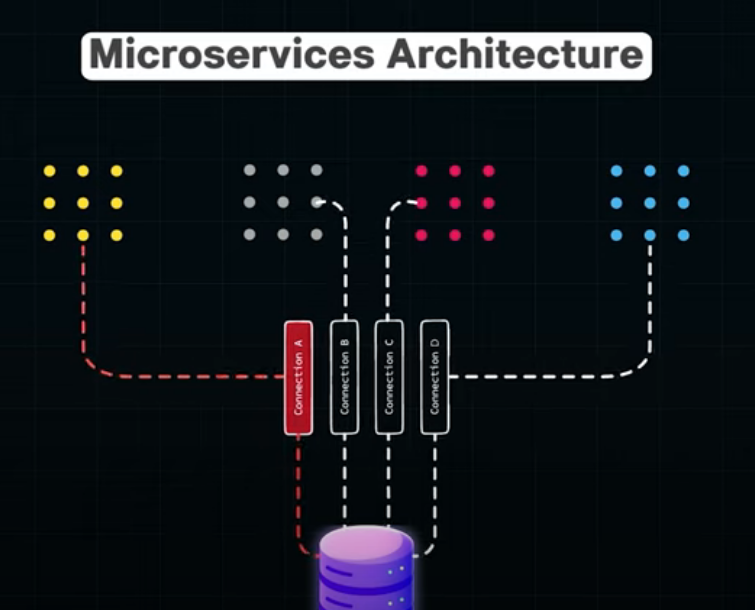
* Gradual migration
* Coexistence
* Flexibility

Challenges:

* Complexity
* Data consistency
* Increased n/w calls
* Dependency management between monolith components and microservice.

# **Bulkhead pattern**

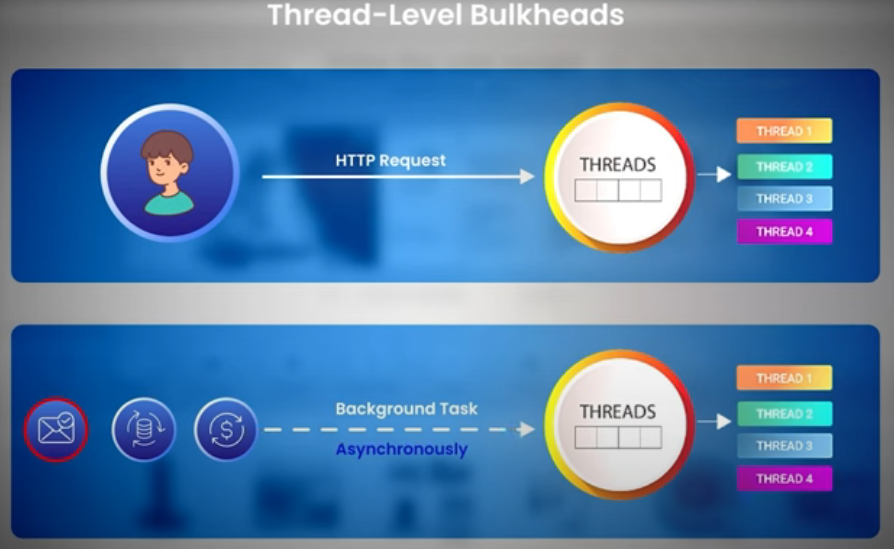
Database connection pool bulkhead



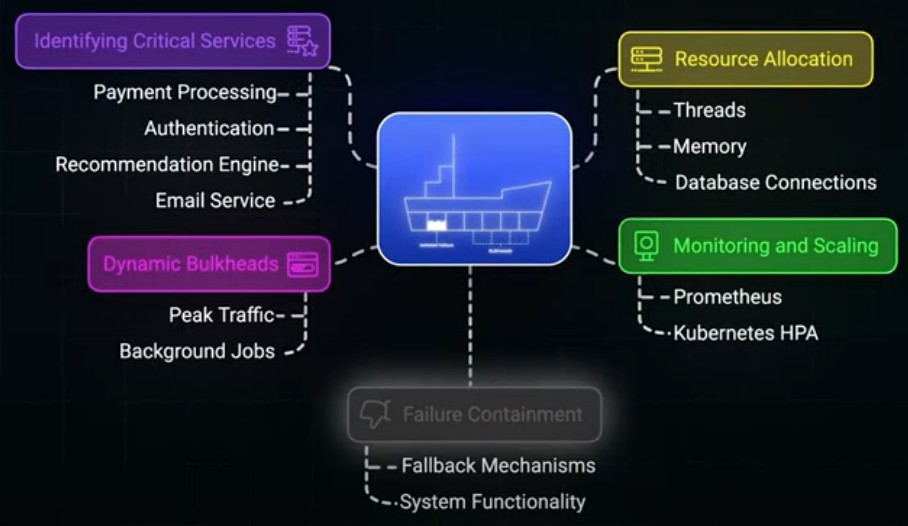
Memory and CPU bulkheads ensures that every microservice gets a dedicated memory and CPU allocated.

Service-level bulkhead makes sure that mission critical microservices gets a dedicated resources and higher priority e.g. authentication service over recommendation service.

Thread-level bulkheads



Key factors while designing bulkhead,



Bulkhead challenges:

