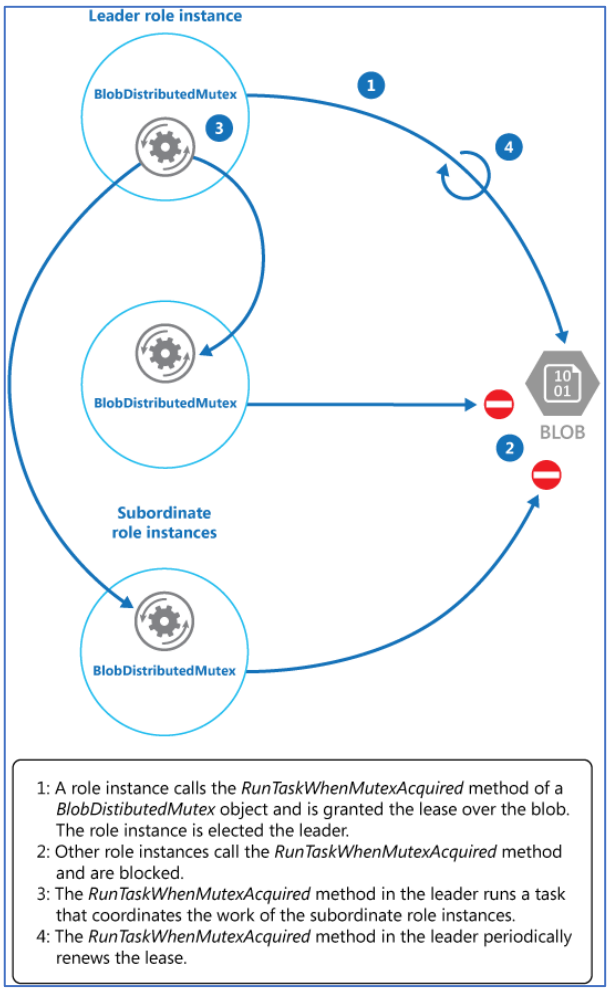
Design patterns used when designing & implementing Microservices.

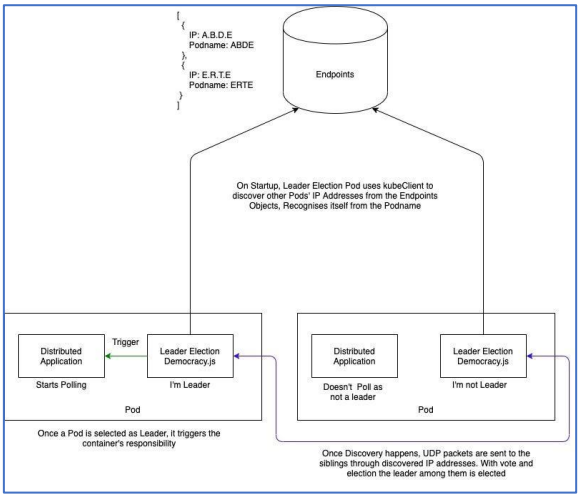
For **each pattern**, I've included:

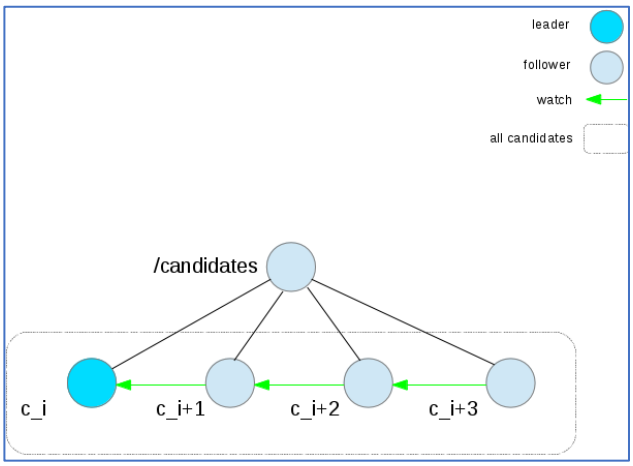
- **What it is**
- **Why it's needed**
- **Real-world problem**
- **Practical solution example**

---

1 Leader Election Pattern







What it is

In a distributed system, **only one service instance is elected as the leader** to perform a critical task.

Why needed

Multiple instances running the same job can cause:

- Duplicate processing
- Data corruption
- Race conditions

Real Problem

You have **5 instances** of a microservice that runs a **daily billing job**.

❌ Without leader election → Billing runs **5 times**

✅ With leader election → Billing runs **once**

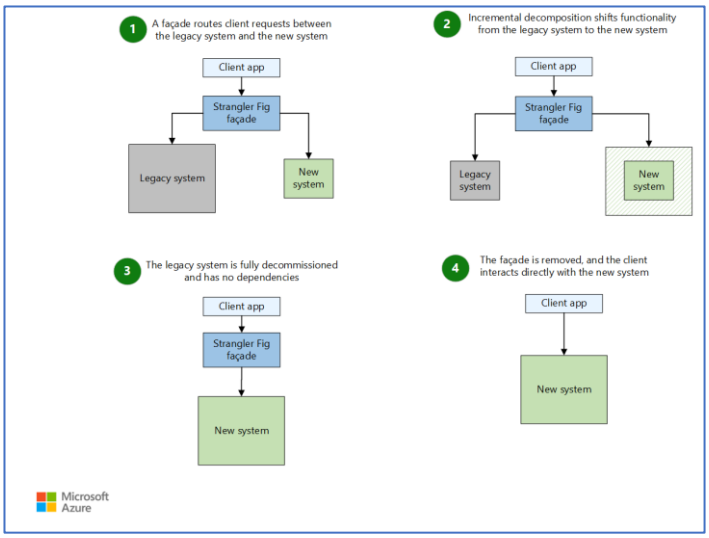
Solution

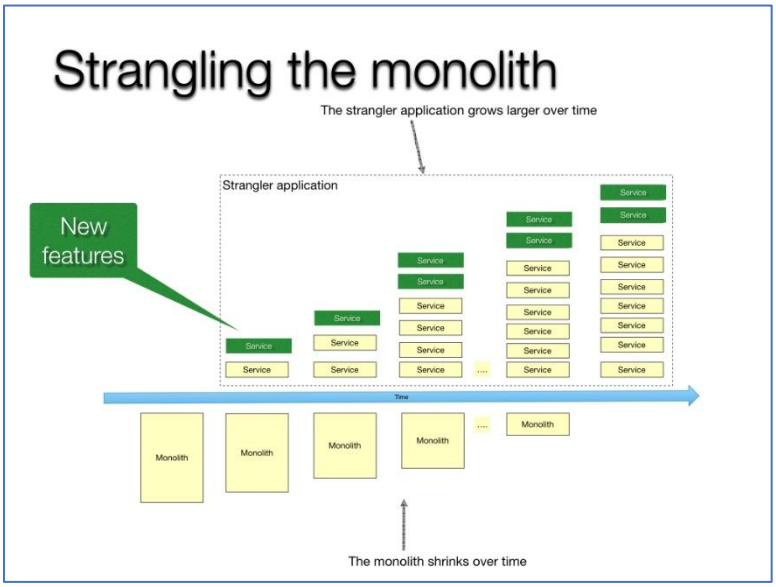
- Use **ZooKeeper / etcd / Consul / Kubernetes Lease**
- One instance becomes **leader**
- Others stay **standby**

📌 Example:

Only the leader processes scheduled payments; if it crashes, another instance takes over.

2 Strangler Fig Pattern





What it is

Gradually **replace a monolithic system with microservices** without shutting it down.

Why needed

- Big bang rewrites are risky
- Business cannot stop

Real Problem

A **15-year-old monolith** handling:

- Orders
- Payments
- Users

You want microservices but **cannot rewrite everything at once**.

Solution

1. Put an **API Gateway** in front
2. Route new features to microservices
3. Slowly disable old monolith parts

📌 Example:

Orders service rewritten first, while Payments still run in monolith.

---

3 Retry Pattern

What it is

Automatically **retry a failed request** instead of failing immediately.

Why needed

Distributed systems have:

- Network glitches
- Temporary outages

Real Problem

Payment service fails due to **temporary network timeout**.

❌ Immediate failure → Bad user experience

✅ Retry → Success on second attempt

Solution

- Retry with **limits**
- Use **exponential backoff**
- Combine with circuit breaker

📌 Example:

Retry 3 times with delays: 100ms → 300ms → 900ms

4️⃣ Database per Service Pattern

What it is

Each microservice **owns its own database**.

Why needed

- Loose coupling
- Independent scaling
- No shared schema chaos

Real Problem

Order service and Payment service share same DB.

❌ Schema change breaks multiple services

✅ Independent databases = safe changes

Solution

- One DB per service
- Communication via APIs/events
- No direct DB access

📌 Example:

Order DB → MySQL

Analytics DB → MongoDB

5️⃣ API Composition Pattern

What it is

A **composite service** calls multiple microservices and combines responses.

Why needed

Client shouldn't make **10 API calls**.

Real Problem

Frontend needs:

- Order details
- Payment status
- Shipment info

Solution

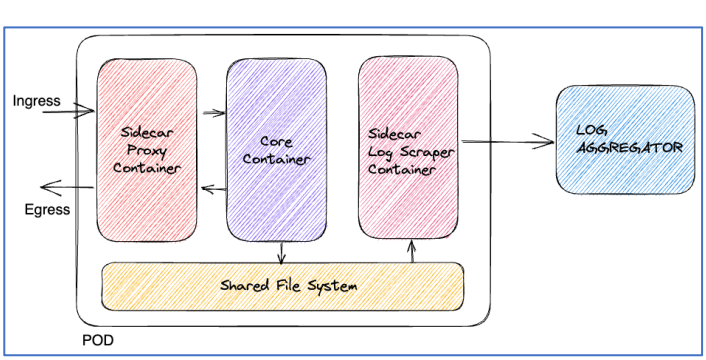
Create **API Composer Service**:

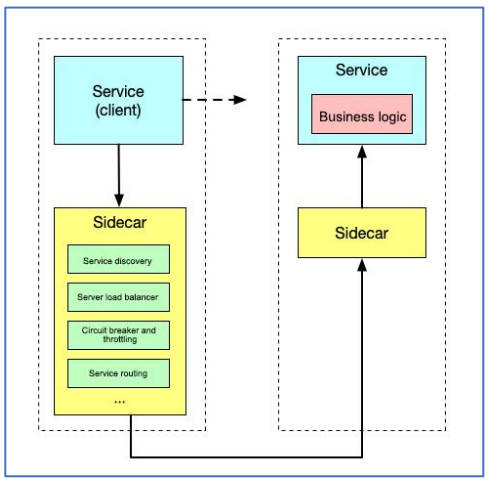
Client → API Composer → Order + Payment + Shipping

📌 Example:

/order-summary/{id} returns combined response

6️⃣ Sidecar Pattern





What it is
Deploy **helper components alongside a service**, sharing the same lifecycle.
Why needed
Keep business logic clean.
Real Problem
You need:

- Logging
- Security
- Monitoring

❌ Adding code in every service
✅ Use sidecar
Solution

- Sidecar handles cross-cutting concerns
- Popular in **Service Mesh (Istio, Linkerd)**

📌 Example:
Envoy proxy as sidecar for traffic control

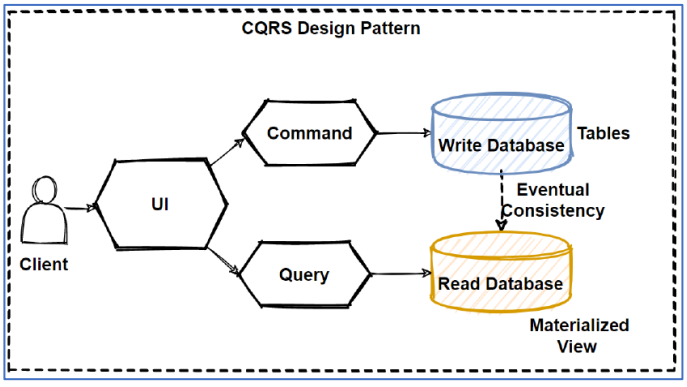
7️⃣ Bulkhead Pattern

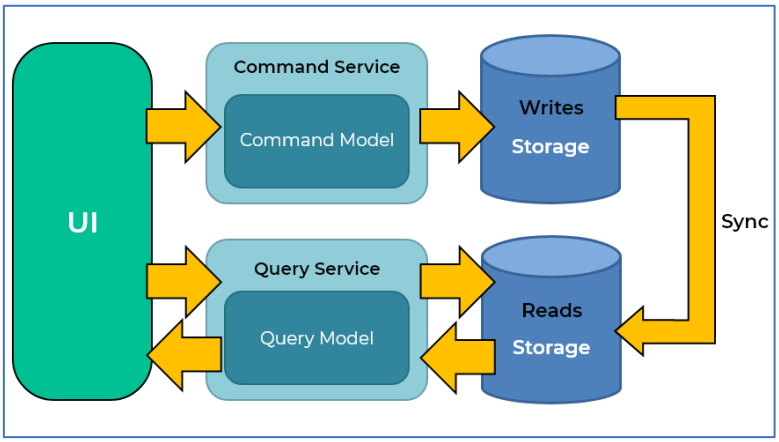
What it is
Isolate resources so **failure in one area doesn't crash everything**.
Why needed
Inspired by **ship bulkheads**.
Real Problem
Report service overload crashes entire system.
Solution

- Separate thread pools
- Separate connection pools

📌 Example:
Payment threads isolated from Reporting threads

8️⃣ CQRS Pattern (Command Query Responsibility Segregation)





What it is
Separate **read model** and **write model**.
Why needed
Reads and writes have different needs.
Real Problem
E-commerce site:

- Heavy reads (product listing)
- Complex writes (orders)

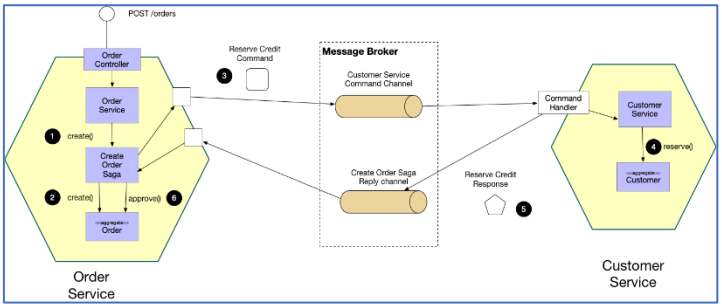
Solution

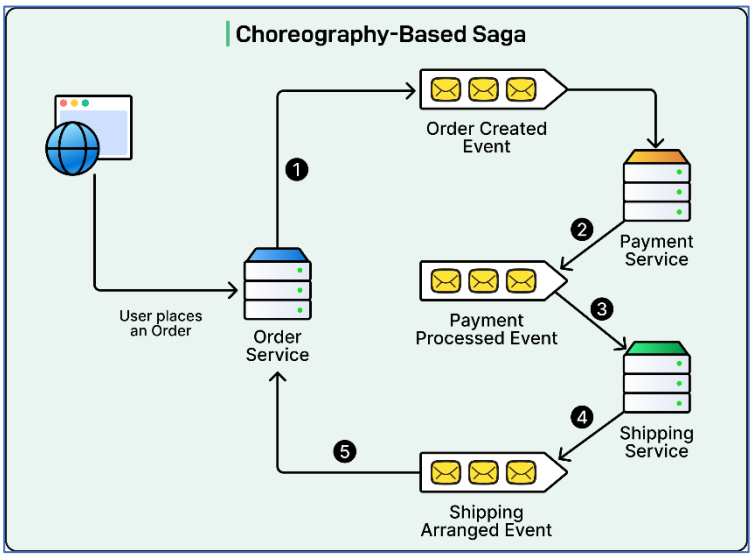
- Write DB → normalized
- Read DB → denormalized / cached

📌 Example:
Orders written to SQL, read from Elasticsearch

---

9 SAGA Pattern





What it is
Manage **distributed transactions** using compensating actions.
Why needed
No ACID transactions across microservices.
Real Problem
Order → Payment → Inventory
Payment fails after inventory reserved.
Solution

- **Choreography**: services react to events
- **Orchestration**: central saga controller

📌 Example:
Payment fails → Inventory release → Order cancelled

---

10 Circuit Breaker Pattern

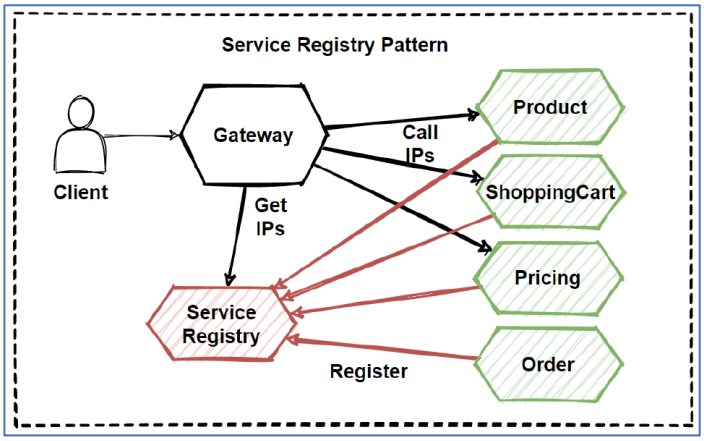
What it is
Stop calling a failing service temporarily.
Why needed
Prevent **cascading failures**.
Real Problem
Recommendation service down → entire checkout hangs.
Solution
States:

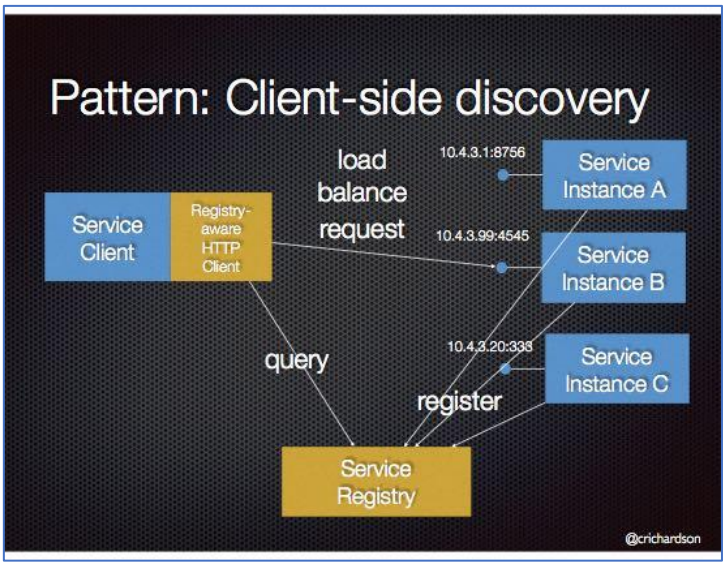
- **Closed** (normal)
- **Open** (fail fast)
- **Half-open** (test)

📌 Example:
Use Resilience4j / Hystrix

---

1 1 Service Registry Pattern





What it is

Services **register themselves**, clients discover dynamically.

Why needed

IP addresses change in cloud.

Real Problem

Hardcoded URLs break during scaling.

Solution

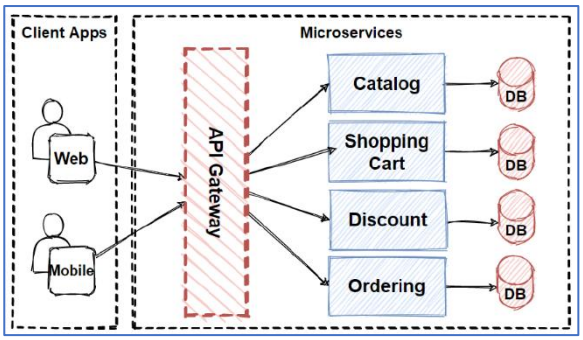
- Eureka / Consul / Kubernetes DNS

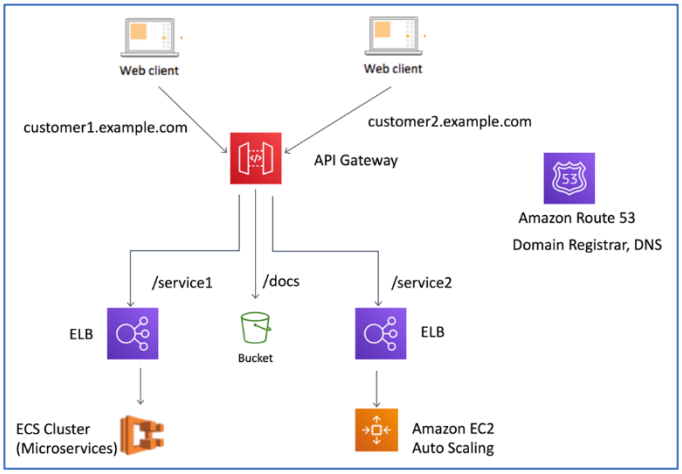
📌 Example:

order-service → auto-discovered

---

1 2 Gateway Pattern





What it is

Single **entry point** for all clients.

Why needed

Avoid exposing internal services.

Real Problem

Mobile app calling 20 services.

Solution

Gateway handles:

- Auth
- Rate limiting
- Routing
- Aggregation

📌 Example:

Netflix Zuul / Spring Cloud Gateway

1️⃣3️⃣ Configuration Externalization Pattern

What it is

Store configs **outside application code**.

Why needed

Different environments → different configs.

Real Problem

DB URL hardcoded → redeploy needed.

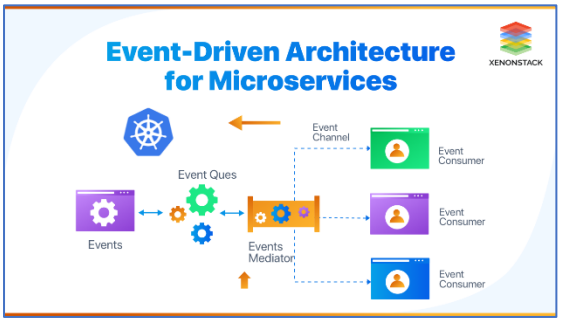
Solution

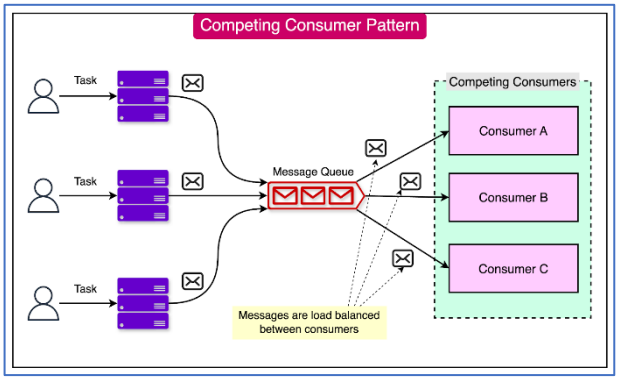
- Config Server
- Environment variables
- Kubernetes ConfigMaps

📌 Example:

Change DB URL without restart

1️⃣4️⃣ Event-Driven Architecture Pattern





What it is

Services communicate using **events**, not direct calls.

Why needed

Loose coupling & scalability.

Real Problem

Order placed → notify email, inventory, analytics.

Solution

- Publish event: OrderCreated
- Subscribers react independently

📌 Example:

Kafka / RabbitMQ / AWS SNS

🧠 Quick Memory Tip (Interview Gold)

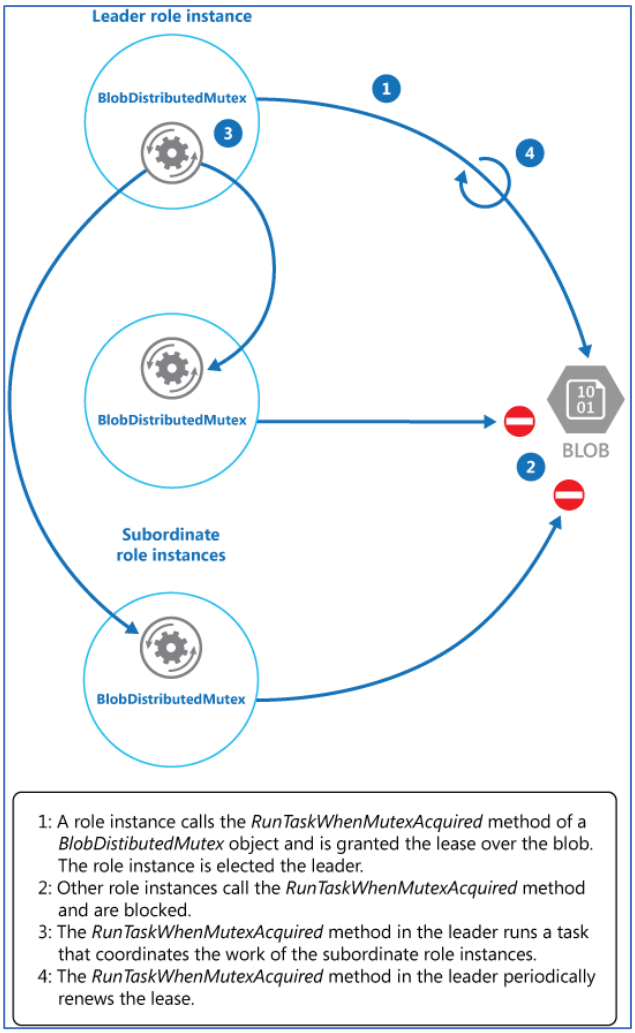
Problem	Pattern
Distributed job	Leader Election
Legacy migration	Strangler Fig
Temporary failure	Retry
DB coupling	Database per Service
Multiple API calls	API Composition
Cross-cutting concerns	Sidecar
Overload protection	Bulkhead
Read/write scaling	CQRS
Distributed transaction	SAGA
Downstream failure	Circuit Breaker
Dynamic discovery	Service Registry

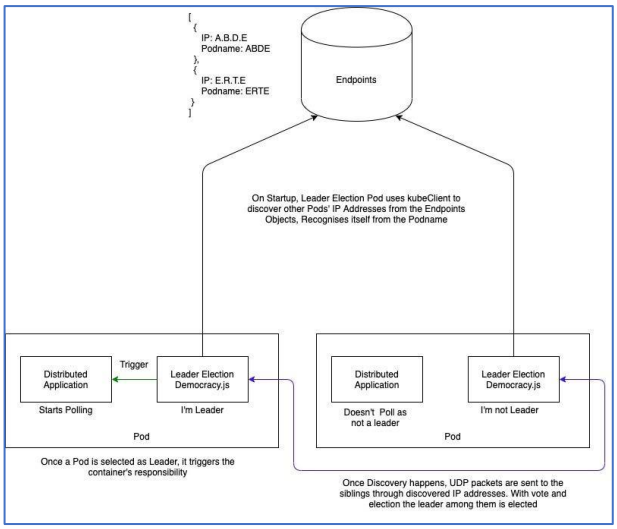
Problem	Pattern
Single entry	Gateway
Env configs	Externalized Config
Async communication	Event-Driven

Microservices Design Patterns – Exam Notes

---

1. Leader Election Pattern





Definition:
Leader Election Pattern ensures that **only one instance** of a microservice performs a critical task at a time in a distributed system.

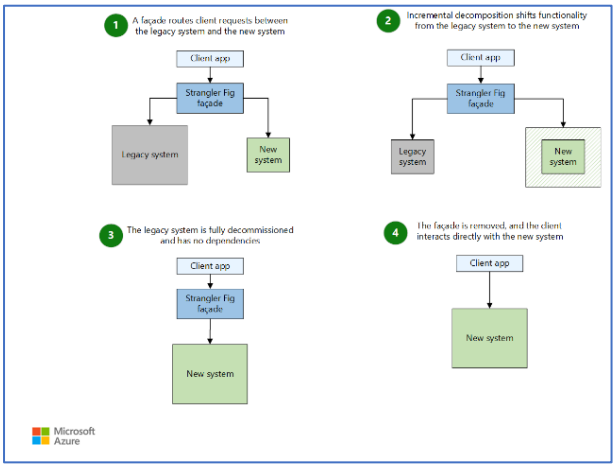
Key Points:

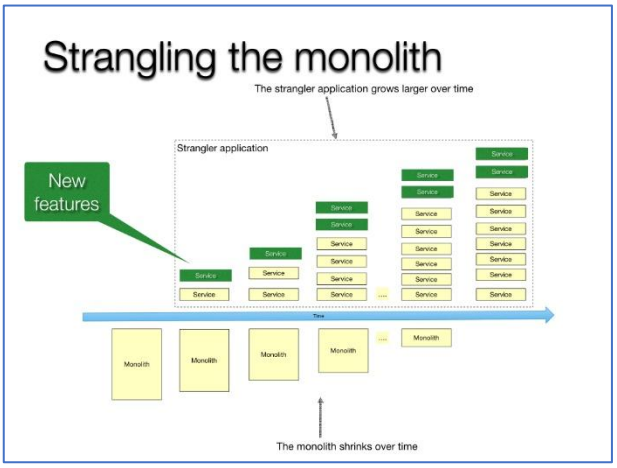
- Prevents duplicate processing
- Handles failover automatically
- Common tools: ZooKeeper, etcd, Kubernetes Lease

Example:
Only one service instance runs a scheduled billing job; if it fails, another instance becomes leader.

---

2. Strangler Fig Pattern





Definition:

Strangler Fig Pattern incrementally **replaces a monolithic application with microservices**.

Key Points:

- Legacy system continues running
- New functionality built as microservices
- Old parts removed gradually

Example:

User service migrated first while payment logic remains in monolith.

---

3. Retry Pattern

Definition:

Retry Pattern automatically retries a failed operation due to **temporary failures**.

Key Points:

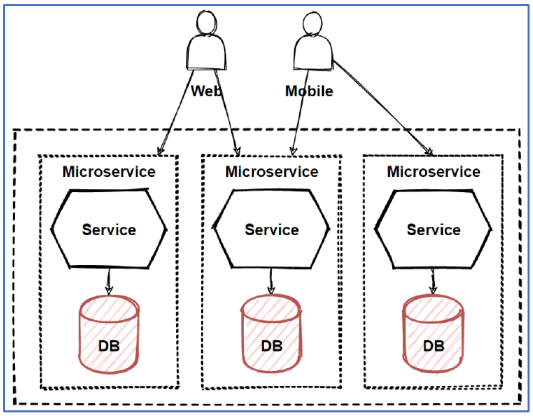
- Handles transient faults
- Uses retry limits
- Often combined with circuit breaker

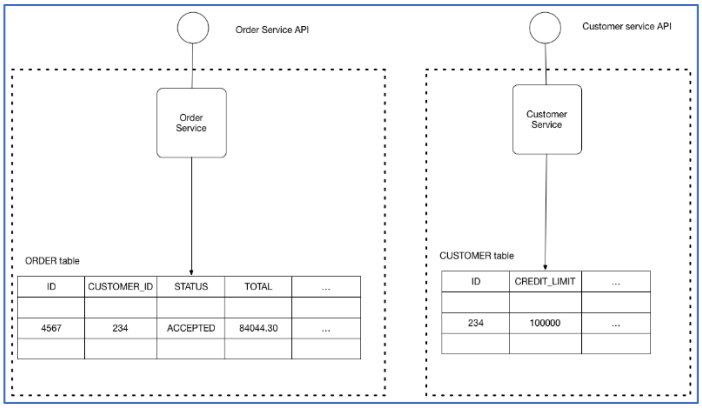
Example:

Retry payment request after network timeout.

---

4. Database per Service Pattern





Definition:

Each microservice has its **own dedicated database**.

Key Points:

- Loose coupling
- Independent schema changes
- Improves scalability

Example:

Order service uses MySQL, Product service uses MongoDB.

5. API Composition Pattern

Definition:

API Composition Pattern aggregates responses from **multiple microservices** into one API response.

Key Points:

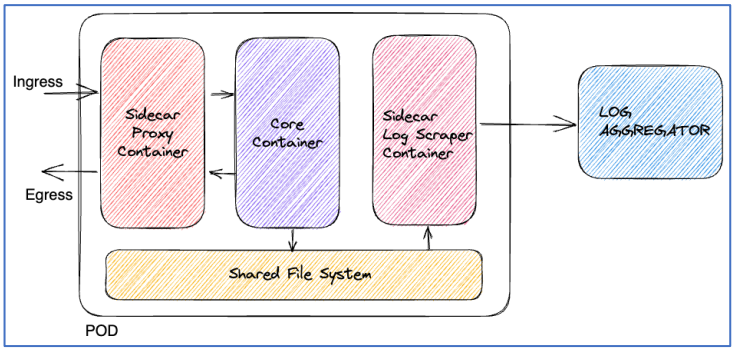
- Reduces client complexity
- Used in backend-for-frontend (BFF)
- Avoids multiple client calls

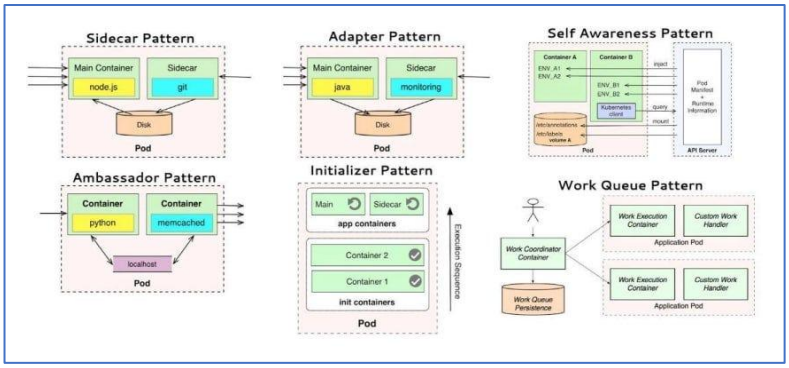
Example:

Order summary API combines order, payment, and shipping data.

---

6. Sidecar Pattern





Definition:

Sidecar Pattern deploys a helper service alongside the main service to handle cross-cutting concerns.

Key Points:

- Same lifecycle as main service
- No changes to business logic
- Common in service mesh

Example:

Envoy proxy handles logging and security.

---

7. Bulkhead Pattern

Definition:

Bulkhead Pattern isolates components so failure in one does not affect others.

Key Points:

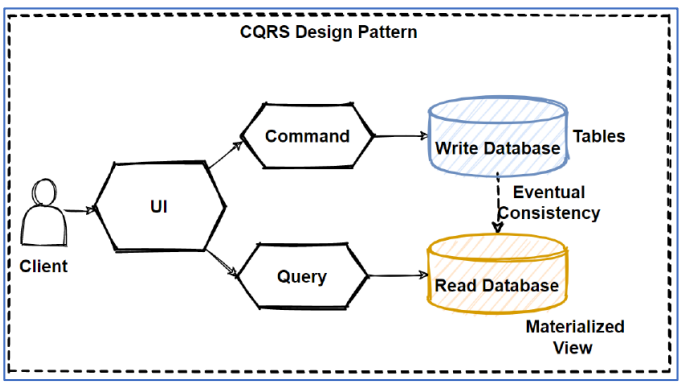
- Resource isolation
- Prevents cascading failures
- Inspired by ship bulkheads

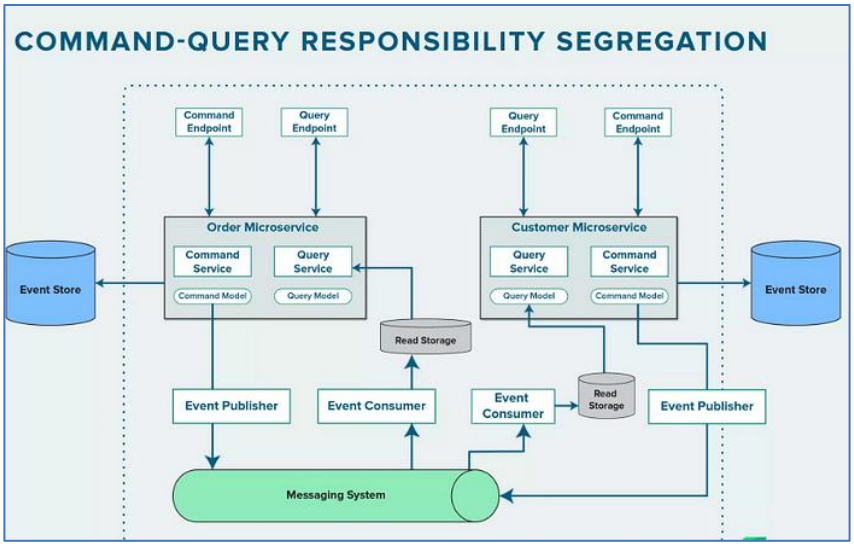
Example:

Separate thread pools for payment and reporting services.

---

8. CQRS Pattern





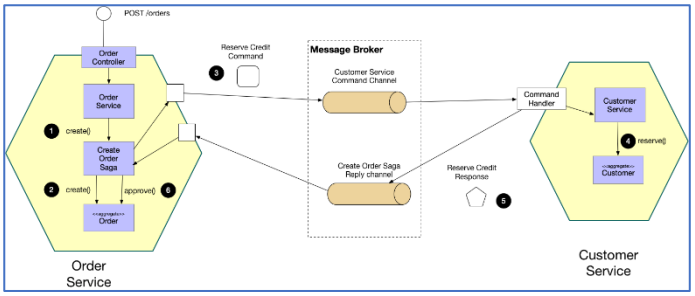
Definition:
CQRS separates **read operations** from **write operations** into different models.
Key Points:

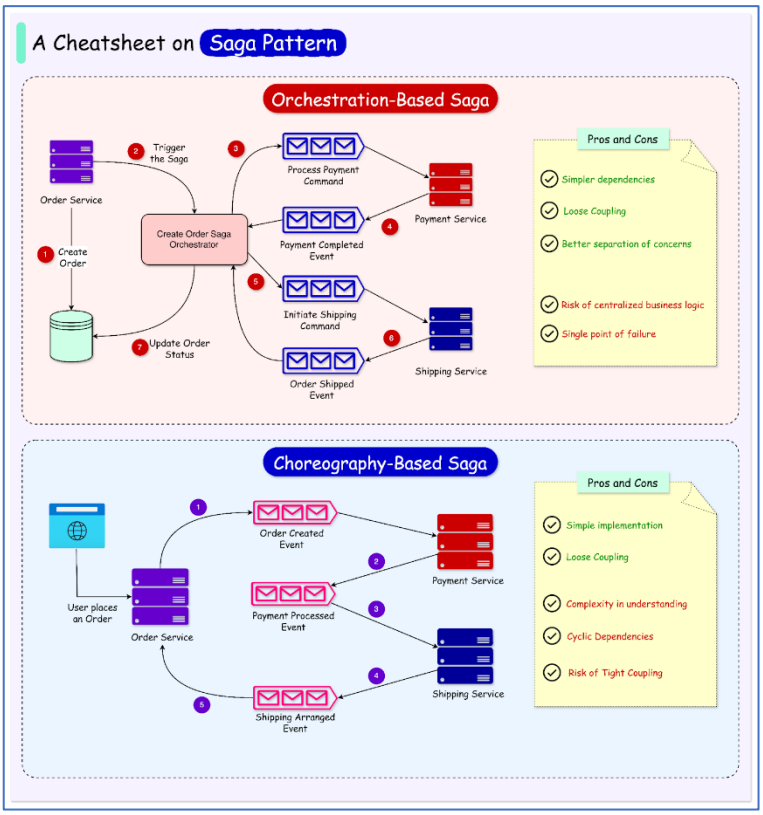
- Optimizes read/write workloads
- Improves performance
- Often used with event sourcing

Example:
Orders written to SQL DB, read from Elasticsearch.

---

9. SAGA Pattern





Definition:
Saga Pattern manages **distributed transactions** using a sequence of local transactions with compensating actions.
Key Points:

- Avoids distributed ACID transactions
- Two types: Orchestration, Choreography
- Ensures data consistency

Example:
If payment fails, inventory reservation is rolled back.

---

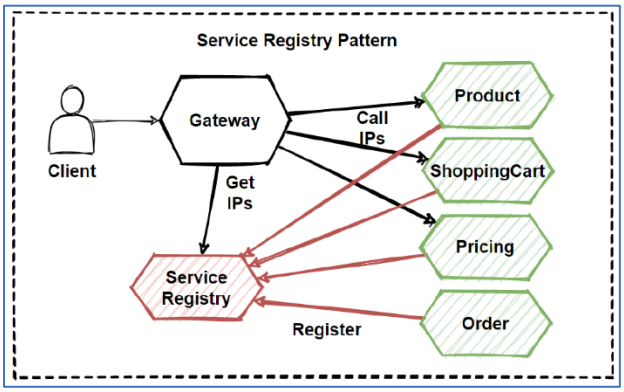
10. Circuit Breaker Pattern
Definition:
Circuit Breaker Pattern prevents a service from calling a **failing dependency**.
Key Points:

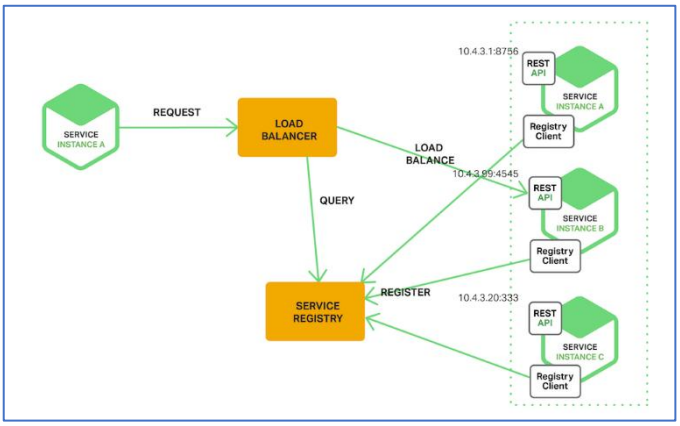
- Improves system stability
- Has states: Closed, Open, Half-Open
- Prevents cascading failures

Example:
Stop calling recommendation service when it is down.

---

11. Service Registry Pattern





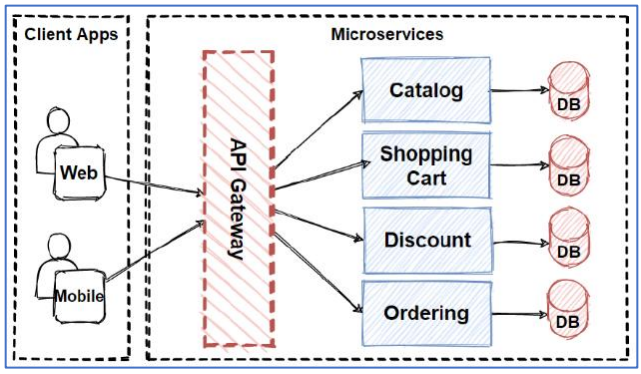
Definition:
Service Registry Pattern allows services to **register and discover** each other dynamically.
Key Points:

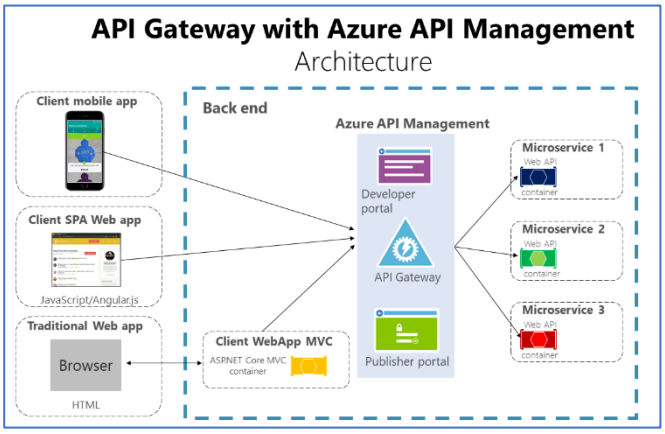
- No hardcoded service URLs
- Supports auto-scaling
- Enables service discovery

Example:
Order service discovers payment service via registry.

---

12. Gateway Pattern





Definition:
Gateway Pattern provides a **single entry point** for all client requests.
Key Points:

- Handles authentication and routing
- Improves security
- Reduces client complexity

Example:
Mobile app accesses all services via API Gateway.

---

13. Configuration Externalization Pattern

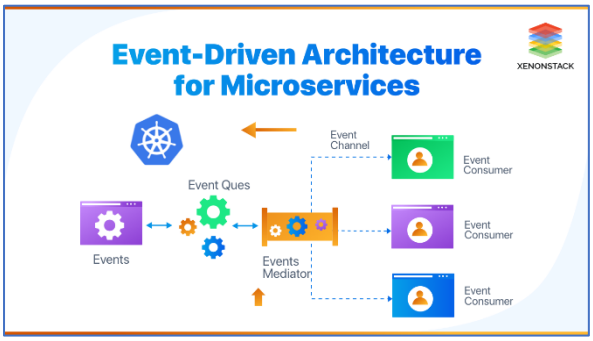
Definition:
Configuration Externalization stores application configuration **outside the codebase**.
Key Points:

- Environment-specific configuration
- No redeployment required
- Centralized management

Example:
Database URL stored in Config Server.

---

14. Event-Driven Architecture Pattern



Definition:
Event-Driven Architecture enables services to communicate using **events** asynchronously.

Key Points:

- Loose coupling
- High scalability
- Supports eventual consistency

Example:
Order service publishes OrderCreated event consumed by inventory and notification services.

One-Line Exam Memory Table

Pattern	Purpose
Leader Election	Single active instance
Strangler Fig	Gradual monolith migration
Retry	Handle transient failures
Database per Service	Data isolation
API Composition	Combine service responses
Sidecar	Cross-cutting concerns
Bulkhead	Fault isolation
CQRS	Separate read/write
Saga	Distributed transactions
Circuit Breaker	Prevent cascading failures
Service Registry	Dynamic discovery
Gateway	Single entry point
Config Externalization	External configs
Event-Driven	Async communication
