Common Microservices Design Patterns

Microservices architecture, while offering flexibility and scalability, introduces new challenges in terms of communication, data consistency, and operational complexity. Design patterns help address these challenges by providing proven solutions to recurring problems.

1. API Gateway

- **Description:** A single entry point for all client requests to the microservices. It handles request routing, composition, and protocol translation, abstracting the internal microservice architecture from clients.
- **Example:** AWS API Gateway, Netflix Zuul, Spring Cloud Gateway.

2. Aggregator Pattern

- **Description:** A service that invokes multiple internal microservices, gathers their responses, and then combines or transforms them into a single, comprehensive response for the client.
- **Example:** A `UserProfileService` that calls `OrderService`, `PaymentService`, and `ReviewService` to build a complete user profile view.

3. Chained (or Chain of Responsibility) Pattern

- **Description:** A sequence of microservices where each service performs a specific task and then passes the modified request or data to the next service in the chain for further processing. The output of one service becomes the input for the next.
- **Example:** An e-commerce order processing pipeline: `OrderValidationService` -> `InventoryReservationService` -> `PaymentProcessingService` -> `ShippingService`.

4. Database per Service

- **Description:** Each microservice owns and manages its own private database or data store. No other service can directly access another service's database. Communication happens only via APIs.
- **Example:** A `ProductService` uses a NoSQL database (e.g., MongoDB) for product details, while an `OrderService` uses a relational database (e.g., PostgreSQL) for transactional order data.

5. Event Sourcing

- **Description:** Instead of storing only the current state of an application's data, all changes to the application state are stored as a sequence of immutable events. The current state can be reconstructed by replaying these events.
- **Example:** Financial systems where every transaction (debit, credit) is an event, allowing for a full audit trail and easy reconstruction of account balances at any point in time.

6. Command Query Responsibility Segregation (CQRS)

- **Description:** Separates the concerns of "commands" (operations that change data, like Create, Update, Delete) from "queries" (operations that read data). This often involves using separate data models or even separate data stores optimized for each purpose.
- **Example:** An e-commerce system where product updates (commands) go to a transactional database, but product listings (queries) are served from a highly optimized, denormalized read model (e.g., in Elasticsearch) that is updated asynchronously by events.

7. Circuit Breaker Pattern

- **Description:** A mechanism to prevent a single failing service from causing cascading failures across the entire system. When a service repeatedly fails or becomes unresponsive, the circuit breaker "trips," causing subsequent requests to fail fast (without even attempting to call the unhealthy service) for a defined period, giving the failing service time to recover.
- **Example:** Hystrix (Netflix's former library), Resilience4j.

8. Decomposition Design Pattern

- **Description:** The fundamental approach to breaking down a large application (monolith) into smaller, independent microservices. Common strategies include decomposition by business capability (each service encapsulates a specific business function) or by subdomain (based on Domain-Driven Design principles).
- **Example:** Breaking down a monolithic e-commerce application into separate microservices for `User Management`, `Product Catalog`, `Order Processing`, `Payment`, and `Shipping`.

9. Saga Pattern

- **Description:** A way to manage distributed transactions that span multiple microservices, ensuring data consistency across them. A saga is a sequence of local transactions, where each transaction updates data within a single service and publishes an event that triggers the next local transaction in the saga. If a step fails, compensating transactions are executed to undo previous changes.
- **Example:** A complex order fulfillment process involving `Inventory`, `Payment`, and `Shipping` services. If payment fails, compensating actions might cancel inventory reservation and send a notification.

10. Externalized Configuration

- **Description:** Configuration data (e.g., database connection strings, API keys, feature flags) is stored externally to the application's deployable package and loaded at runtime. This allows changing configurations without rebuilding or redeploying the application.
- **Example:** AWS Systems Manager Parameter Store, HashiCorp Vault, Spring Cloud Config Server, Kubernetes ConfigMaps/Secrets.