**Micro Services Design Patterns**

**Strangler:**

* Strangler Pattern is one of the **Decomposition Pattern**.
* The strangler design pattern is a popular design pattern to **incrementally** **transform** your monolithic application to microservices by **replacing old functionality** with a new service.
* The **facade interface**, which acts as primary interface between the legacy system and the other apps.
* Once the new component is ready, the old component is **strangled** and a new one is ready to use.
* **Application availability** is the top most priority.
* This pattern allows an incremental and reliable process of refactoring code
* The best part is that changes are incremental, monitored at all times, well-defined and the chances of things going wrong are pretty low.
* During migration, an API Gateway acts like a facade which directs users’ requests to the correct application.
* Once the migration is complete, the monolithic architecture is “strangled,” and the monolithic application is ready to be retired.

**Disadvantages**

* Requires a lot of ongoing attention to routing and network management.
* A refactor effort can get stuck in “adapter hell.” Each instance of strangling an old service in favour of a new one will require special logic to accommodate the rerouting from the old service to the new service. When you have dozens, if not hundreds of services in play, this can be a lot of work.
* Requires making sure that you have a rollback plan in play for each refactored instance. Things will go wrong. You need to be able to roll back to the old way of doing things quickly and safely.

**Saga Pattern:**

* A **distributed transaction management** pattern used in **microservices**.
* Ensures **data consistency** without using a traditional **two-phase commit (2PC)**.
* Breaks a transaction into **multiple local transactions** across microservices.
* If a step fails, **compensating transactions** are executed to roll back previous steps.
* **Avoids distributed transactions (2PC)**, which are not scalable
* Ensures **eventual consistency** across services.

**Choreography-Based Saga**

* **Each microservice listens to events and publishes events triggers the next step**.
* No central coordinator; each service manages its own actions.
* Choose **Choreography for simple event-driven workflows**
* **Debug**: Hard to trace and rollback.
* Used with **event-driven** with **Kafka**
* Each service knows how to trigger the next service and what to do if there’s a failure. Each service in the Saga carries out its transaction and publishes events. The other services respond to those occurrences and carry out their tasks. In addition, depending on the scenario, they may or may not publish additional events.
* **Pros:** No single point of failure and can be useful for simple workflows

**Orchestration-Based Saga**

* A central coordinator (like a workflow engine) controls the process, telling each service when to act.
* No Need to interact with messaging frameworks
* Choose **Orchestration for complex business processes**.
* **Orchestrator calls each microservice explicitly** and handles failures.
* In the Orchestration saga, each service participating in the saga performs their transactions and publish events. The other services respond to those events and complete their tasks
* **Pros:** Centralized control makes it easier to manage and Easier to debug and monitor.
* **Best For:** Complex workflows, Easier to debug and rollback.

**Disadvantages**

* It is less suitable for tight coupling applications.
* Hard to manage as the number of services grows.
* Debugging and monitoring become complex.
* Complexity of the SAGA design pattern is high from the programmer's point of view and developers are not well accustomed to writing sagas as traditional transactions.

### **When NOT to Use the Saga Pattern?**

❌ If your system **requires strong consistency** (e.g., banking transactions).  
❌ If **rollback logic is too complex**.  
❌ If **latency-sensitive operations** require immediate responses.

**🔹 Example:**

* **Scenario:** Order Processing  
  ✅ **Step 1:** Order Service → **Creates an order**.  
  ✅ **Step 2:** Payment Service → **Deducts money**.  
  ✅ **Step 3:** Inventory Service → **Reserves stock**.  
  ✅ If Step 2 fails, Step 1 needs to be **rolled back** (order cancellation).

**🔹 Benefits:**

✅ Ensures **eventual consistency** in distributed transactions.  
✅ Improves fault tolerance using **compensating actions**.  
✅ Works well with **asynchronous messaging (Kafka, RabbitMQ)**.

|  |  |
| --- | --- |
| Complex mapping between domains | Use **Event-Driven Communication** (Kafka, RabbitMQ) |

|  |  |
| --- | --- |
| Managing consistency across services | Use **CQRS or Event Sourcing** |

**Event Sourcing**

* Event Sourcing is a powerful **alternative to traditional CRUD**.
* It **records every state change as an event**, ensuring **auditability, scalability, and event replay**.
* **Use it for financial systems, IoT, and real-time event-driven applications**.
* **Event Store:** Stores all changes as events (Kafka, MongoDB, or Event Store DB).
* **Persist state as a series of events**, reconstructing it from events.
* **Concept:** Stores the **sequence of events** instead of the latest state in the database.
* Instead of updating data directly, changes are stored as **immutable events**.
* Services **rebuild the current state** by replaying the stored events.
* Events are **persisted** in an **Event Store (Kafka, EventStoreDB, PostgreSQL)**.
* Can be combined with **CQRS** for better performance.

**🔹 Example:**

* **Scenario:** Bank Transactions  
  ✅ **Step 1:** A **deposit event** of $500 is recorded.  
  ✅ **Step 2:** A **withdrawal event** of $200 is recorded.  
  ✅ **Step 3:** To get the current balance, all events are replayed → **Balance = $300**.

**🔹 Benefits:**

✅ Provides **auditability and traceability**.  
✅ Supports **event replay** for debugging or restoring data.  
✅ Works well in **distributed systems** where data changes frequently.

| **Challenge** | **Solution** |
| --- | --- |
| Event storage growth | Implement **event compaction** or **snapshotting** |
| Complexity of event handling | Use **Kafka Streams or Debezium** |

**Domain Driven Design Pattern (DDD):**

* Stop adding functionality to the monolithic application.
* Split the frontend from the backend
* Extract monolithic modules into micro services. Decompose and decouple the monolith into a series of micro services.
* **Concept:** Aligns microservices with **business domains** for better separation of concerns.
* **Entities:** Objects with a unique identity (e.g., Order, Customer).
* **Value Objects:** Objects that don’t have an identity (e.g., Address).
* **Aggregates:** A group of domain objects treated as a single unit (e.g., Order with multiple Line Items).
* **Repositories:** Handles database access for aggregates.
* **Bounded Contexts:** Defines service boundaries (e.g., Order Service vs. Payment Service).

**🔹 Benefits:**

✅ **Better modularization** of microservices.  
✅ Improves **team ownership** by defining clear service boundaries.  
✅ Helps avoid **tight coupling** between microservices.

| **Challenge** | **Solution** |
| --- | --- |
| Complex mapping between domains | Use **Event-Driven Communication** (Kafka, RabbitMQ) |
| Managing consistency across services | Use **CQRS or Event Sourcing** |

**API Gateway**

* API Gateway is one of **the Integration Pattern**.
* API gateway sits between the client apps and the microservices and it serves as a reverse proxy, forwarding client requests to services.
* It provides **Routes**, **Filter,** [**Authentication**](https://javarevisited.blogspot.com/2018/01/how-http-basic-authentication-works-in.html#axzz6hhgr3Uqg)**, SSL termination**, and **caching** are some of the other cross-cutting services.
* Acts as a **reverse proxy** for microservices.
* Handles **authentication, authorization, rate limiting, logging, and monitoring**.
* Reduces the **number of client-to-service calls**.
* Supports **load balancing** and **caching** for improved performance.
* Can be implemented using tools like **Spring Cloud Gateway, Zuul, Kong, Apigee**.

**🔹 Benefits:**

✅ Reduces client-side complexity.  
✅ Provides a security layer before requests reach microservices.  
✅ Supports **protocol translation** (e.g., HTTP to WebSockets).  
✅ Handles retries, fallbacks, and circuit breaking.

| **Challenge** | **Solution** |
| --- | --- |
| Single point of failure | Deploy API Gateway in a **highly available cluster** |
| Increased latency | Use **caching and load balancing** |
| Managing authentication | Implement **OAuth2, JWT, API Keys** |

**Database Per Microservice Pattern**

* Concept: Each microservice has its own database and does not share it with others.
* Ensures **loose coupling** between microservices.
* Prevents **cross-service failures** due to shared database issues.
* Allows microservices to use the **most suitable database** (SQL, NoSQL, etc.).
* Improves **scalability** and **autonomy** of each service.
* Common databases used: PostgreSQL, MySQL, MongoDB, Cassandra, DynamoDB**.**

🔹 **Benefits**:

✅ **Better modularization** of microservices.  
✅ Improves **team ownership** by defining clear service boundaries.  
✅ Helps avoid **tight coupling** between microservices.

| **Challenge** | **Solution** |
| --- | --- |

|  |  |
| --- | --- |
| Data consistency across microservices | Use **event-driven architecture** (Kafka, RabbitMQ) |

|  |  |
| --- | --- |
| Complex queries across services | Implement **API Composition** or **CQRS** |

|  |  |
| --- | --- |
| Increased storage costs | Optimize data retention and use - **data partitioning** |

**Aggregator Pattern**

* **Concept:** Aggregates data from multiple microservices and presents a **single response**.
* Used when a **client needs data from multiple microservices**.
* Helps **reduce network calls** and improves performance.
* Can be implemented via:  
  ✅ **API Gateway Aggregation** – Combines responses from multiple microservices.  
  ✅ **Backend Aggregation** – A dedicated service handles aggregation and caching.

**Example:**

* **Scenario:** A shopping app needs **User Profile, Order History, and Payment Details**.
* Instead of calling three separate APIs, an **Aggregator Service** collects and returns a **unified response**.

| **Approach** | **Description** |
| --- | --- |

|  |  |
| --- | --- |
| **API Gateway Aggregation** | API Gateway calls multiple microservices and merges responses. |

|  |  |
| --- | --- |
| **Backend Aggregation** | A dedicated service (e.g., GraphQL) fetches and combines data. |

|  |  |
| --- | --- |
| **Client-side Aggregation** | The frontend sends parallel requests and merges data. |

**CQRS (Command Query Responsibility Segregation) Pattern**

* **Concept:** Separates **read (query)** and **write (command)** operations into different models.
* Improves **performance** and **scalability** by using different data stores for reads and writes.
* Helps in **event-driven architectures** where real-time data updates are needed.
* Commonly used in high-throughput applications like **financial transactions, IoT, and analytics**.

**Example:**

* **Scenario:** A banking app handles **money transfers**.
* **Write Model (Command Service):** Updates balances in **PostgreSQL**.
* **Read Model (Query Service):** Retrieves balances from **Redis** for fast access.
* **Event Store (Kafka):** Keeps a history of all transactions.

| **Component** | **Role** |
| --- | --- |

|  |  |
| --- | --- |
| **Command Model** | Handles **write operations** (Insert, Update, Delete). |

|  |  |
| --- | --- |
| **Query Model** | Handles **read operations**, optimized for fast queries. |

|  |  |
| --- | --- |
| **Event Store** | Stores changes as **events** for auditability. |

|  |  |
| --- | --- |
| **Message Broker (Kafka, RabbitMQ)** | - Ensures **event-driven updates** between models. |

**The Spring Cloud Gateway has three important parts to it:**

**Route** − These are the building blocks of the gateway which contain the URL to which the request is to be forwarded to and the predicates and filters that are applied to the incoming requests.

**Predicate** − These are the set of criteria that should match for the incoming requests to be forwarded to internal microservices. For example, a path predicate will forward the request only if the incoming URL contains that path.

**Filters** − These act as the place where you can modify the incoming requests before sending the requests to the internal microservices or before responding back to the client.

**Disadvantages:**

* It can cause performance degrade due to lots of happenings on the API Gateway.
* With this, discovery service should be implemented.
* Sometimes, it becomes the single point of failure.

**Micro Services Design Approach:**

Designing microservices involves breaking down a monolithic application into smaller, independently deployable services that are organized around business capabilities. Each microservice typically encapsulates a specific function or domain of the application. Here’s a high-level approach to designing microservices:

**1. Define the Service Boundaries**

* **Domain-Driven Design (DDD):** Start by understanding the business domain and identifying the bounded contexts. Each bounded context can become a microservice. This ensures that the services are aligned with business capabilities.
* **Single Responsibility Principle:** Each microservice should have a single responsibility and should focus on doing one thing well. This makes the service easier to maintain and scale independently.

**2. Design for Decentralization**

* **Data Decentralization:** Each microservice should own its data, avoiding a single shared database across services. This can be achieved by designing services with their own database schemas or instances. This autonomy allows services to evolve independently.
* **Decentralized Governance:** Allow teams to choose the appropriate technologies, tools, and patterns that best suit the service's needs. This encourages innovation and flexibility.

**3. Service Communication**

* **Synchronous Communication:** Use RESTful APIs or gRPC for synchronous communication between services. However, be cautious of the increased coupling and potential for cascading failures.
* **Asynchronous Communication:** Implement message brokers (e.g., RabbitMQ, Kafka) for asynchronous communication. This pattern is useful for event-driven architectures and reduces tight coupling between services.
* **API Gateway:** Implement an API Gateway to handle cross-cutting concerns such as authentication, logging, and routing. This can help abstract the complexity of managing multiple microservices from the client.

**4. Service Contracts and Versioning**

* **APIs as Contracts:** Treat the APIs exposed by microservices as contracts with other services or clients. Ensure backward compatibility when making changes to these APIs.
* **Versioning:** Implement versioning in your APIs to support gradual migrations and avoid breaking changes. Common strategies include URI versioning (e.g., /v1/resource) and header-based versioning.

**5. Resilience and Fault Tolerance**

* **Circuit Breakers:** Implement circuit breakers to prevent cascading failures across microservices. When a service is down or slow, the circuit breaker will short-circuit the call and return a fallback response.
* **Retries and Timeouts:** Use retries with exponential backoff and set reasonable timeouts for external service calls. This helps in handling transient failures and improving overall system resilience.
* **Bulkheads:** Isolate resources (e.g., threads, memory) for different services to prevent failures in one service from affecting others.

**6. Scalability**

* **Independent Scaling:** Design microservices to scale independently based on demand. For example, a service that handles heavy data processing might need more instances than a service that handles lightweight requests.
* **Load Balancing:** Implement load balancing to distribute incoming requests evenly across instances of a microservice, ensuring optimal resource utilization.

**7. Security**

* **Authentication and Authorization:** Implement strong authentication and authorization mechanisms, often centralized through an identity provider (e.g., OAuth2, JWT tokens). The API Gateway can handle authentication and delegate authorization to individual services.
* **Secure Communication:** Ensure that communication between services is secure, typically using HTTPS for transport-level security.
* **Least Privilege Principle:** Services should only have access to the resources and data they need. This minimizes the impact of a potential security breach.

**8. Observability**

* **Logging:** Implement structured logging to capture detailed information about service operations, errors, and events. Correlate logs across services using unique request IDs.
* **Monitoring:** Use monitoring tools to track the health, performance, and resource usage of each service. Metrics like response times, error rates, and request counts are essential for proactive management.
* **Tracing:** Implement distributed tracing (e.g., with tools like Jaeger, Zipkin) to track requests as they flow through multiple services. This helps in identifying bottlenecks and understanding service dependencies.

**9. Continuous Integration and Deployment (CI/CD)**

* **Automated Testing:** Implement automated unit, integration, and end-to-end tests to ensure each service behaves as expected. Use CI pipelines to run tests automatically on code commits.
* **Deployment Pipelines:** Create CI/CD pipelines to automate the build, test, and deployment process for each microservice. Use feature toggles and blue-green deployments to minimize downtime and mitigate deployment risks.

**10. Data Management**

* **Data Consistency:** Design for eventual consistency rather than strong consistency, especially in distributed systems. Use patterns like Saga or two-phase commit for managing distributed transactions.
* **Data Migration:** Plan for schema evolution and data migration. Use versioned migrations to handle changes to the database schema without disrupting the running services.

**11. Service Discovery**

* **Dynamic Service Registry:** Implement a service registry (e.g., Consul, Eureka) where services register themselves on startup. Other services can query this registry to discover the addresses of services they need to communicate with.
* **Load Balancing:** Use client-side or server-side load balancing to distribute requests among available instances of a service.

**12. Handling Data Coupling**

* **Event-Driven Architecture:** Use events to decouple services. For instance, when a service updates its state, it can publish an event, and other services can act on that event asynchronously.
* **Data Replication:** For scenarios where services need data from another service, consider replicating the necessary data to minimize cross-service calls.

**13. Testing Microservices**

* **Unit Testing:** Test individual methods and classes within the microservice.
* **Integration Testing:** Test the service’s integration with external systems, databases, and APIs.
* **Contract Testing:** Ensure that the contracts (APIs) between services are adhered to by both providers and consumers.
* **End-to-End Testing:** Test the entire workflow across multiple services to ensure that they work together as expected.

**14. Governance**

* **Standardization:** Establish standards and best practices for developing microservices, such as coding guidelines, documentation requirements, and common libraries.
* **Cross-Cutting Concerns:** Handle cross-cutting concerns (e.g., logging, security, monitoring) consistently across all services, often by using shared libraries or the API Gateway.

**15. Team Organization**

* **Cross-Functional Teams:** Organize teams around microservices, where each team owns the entire lifecycle of one or more services, from development to deployment to operations.
* **DevOps Culture:** Foster a DevOps culture where teams are responsible for both the development and operations of their services. This encourages ownership and accountability.

| * **Feature** | **Saga Pattern** | **Event Sourcing Pattern** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| **Purpose** | Manages distributed transactions across multiple microservices using a sequence of events. | Captures all changes to an application's state as a sequence of events. |

|  |  |  |
| --- | --- | --- |
| **State Management** | Uses **compensating transactions** to handle failures and ensure consistency. | Stores **immutable events** instead of the current state. |

|  |  |  |
| --- | --- | --- |
| **Persistence** | Does **not** store all past events; only necessary events for handling transactions. | Stores **every event** in an event store (e.g., MongoDB, Kafka, PostgreSQL). |

|  |  |  |
| --- | --- | --- |
| **Data Recovery** | If a step fails, a **compensating event** is triggered to roll back previous steps. | The system can **replay events** to reconstruct the current state. |

|  |  |  |
| --- | --- | --- |
| **Implementation** | Can be implemented using **Choreography (event-driven)** or **Orchestration (centralized controller).** | Events are stored and replayed to restore state in CQRS-based architectures. |

|  |  |  |
| --- | --- | --- |
| **Failure Handling** | If one service fails, Saga executes a compensating action (e.g., refunding a payment if an order fails). | Failure does not affect consistency, as event replay can recover lost state. |

|  |  |  |
| --- | --- | --- |
| **Use Case Example** | **Order processing:** If a payment fails, rollback inventory reservation. | **Bank transactions:** Stores a full history of transactions to rebuild account balance. |

|  |  |  |
| --- | --- | --- |
| **Complexity** | More complex due to **distributed coordination**. | Simpler but requires **event versioning and snapshots** to optimize performance. |

|  |  |  |
| --- | --- | --- |
| **Scalability** | Scales well for distributed microservices where consistency needs to be managed asynchronously. | Scales well for applications needing a **historical record of chang** |

 **Use Saga Pattern** when handling **distributed transactions** (e.g., order processing, money transfers).

 **Use Event Sourcing** when **maintaining a full history of events** is important (e.g., banking, audit logging).

Convert Monolithic to Micro Services to follow design patterns:

**1. Strangler Fig Pattern**

* **Purpose:** The Strangler Fig pattern enables incremental migration by allowing you to gradually replace parts of the monolith with microservices, without disrupting the entire application at once.
* **How It Works:**
  + Identify a specific module or functionality within the monolith.
  + Develop a new Spring Boot microservice to handle this functionality.
  + Route traffic for this functionality to the new microservice, often using an API Gateway.
  + Gradually replace other parts of the monolith with microservices until the monolith is entirely decomposed.
* **Use Case:** Ideal when you need to migrate gradually, allowing parts of the monolith to coexist with microservices during the transition.

**2. Domain-Driven Design (DDD)**

**Domain Driven Design Pattern (DDD):**

* Stop adding functionality to the monolithic application.
* Split the frontend from the backend
* Extract monolithic modules into micro services. Decompose and decouple the monolith into a series of micro services.
* **How It Works:**
  + Use DDD principles to define the bounded contexts within your application.
  + Each bounded context corresponds to a microservice, encapsulating a specific domain of your application.
  + Implement each microservice using Spring Boot, ensuring it handles its own data and logic.
* **Use Case:** Effective when your monolith is large and complex, with distinct business domains that can be isolated.

**3. Decompose by Business Capability**

* **Purpose:** This pattern involves breaking down the monolith into microservices based on specific business capabilities, ensuring each service aligns with a particular business function.
* **How It Works:**
  + Identify core business capabilities within your application (e.g., Customer Management, Order Processing).
  + Build separate Spring Boot microservices for each capability.
  + Each microservice should have its own database and communicate with others via REST APIs or messaging.
* **Use Case:** Suitable when you have clear, distinct business functions that can operate independently.

**4. Decompose by Subdomain**

* **Purpose:** Similar to business capability decomposition but focused on subdomains within the application’s business logic. Each subdomain becomes a separate microservice.
* **How It Works:**
  + Analyze the application to identify subdomains (e.g., Billing, Inventory, Shipping).
  + Create Spring Boot microservices for each subdomain, ensuring each service is responsible for its own data and logic.
  + Use REST, gRPC, or messaging (Kafka, RabbitMQ) for inter-service communication.
* **Use Case:** Useful in applications with complex business domains where each subdomain has a distinct role.

**5. Anti-Corruption Layer (ACL)**

* **Purpose:** The Anti-Corruption Layer ensures that the new microservices can interact with the monolith or other legacy systems without being affected by their design or constraints.
* **How It Works:**
  + Implement an ACL between the microservices and the monolith.
  + Use the ACL to translate requests and responses between the microservices and the legacy system.
  + This allows microservices to use modern practices without being tied to the monolith's outdated architecture.
* **Use Case:** When the new microservices need to interact with the monolithic application during the transition phase.

**6. API Gateway Pattern**

* **Purpose:** The API Gateway pattern provides a single entry point for clients to interact with microservices, handling tasks like routing, load balancing, authentication, and monitoring.
* **How It Works:**
  + Implement an API Gateway (e.g., Spring Cloud Gateway) that routes client requests to the appropriate microservices.
  + The API Gateway can also handle cross-cutting concerns such as security, logging, rate limiting, and response transformation.
* **Use Case:** Essential in microservices architecture to centralize and simplify client interactions with multiple services.

**7. Event Sourcing**

* **Purpose:** Event Sourcing is a pattern where state changes in the application are captured as a series of events. This is useful when migrating complex workflows that involve multiple services.
* **How It Works:**
  + Implement event-driven architecture using Spring Boot with event messaging tools like Kafka or RabbitMQ.
  + Instead of storing the current state, store the events that led to the current state.
  + Microservices subscribe to these events and update their state accordingly.
* **Use Case:** Ideal for applications that require tracking of complex workflows or maintaining audit trails.

**8. Saga Pattern**

* **Purpose:** The Saga pattern manages distributed transactions across microservices by coordinating a series of local transactions, ensuring eventual consistency.
* **How It Works:**
  + Implement the Saga pattern using Spring Boot, where each microservice completes its transaction and publishes an event.
  + If a step fails, compensating actions are triggered to rollback the transaction.
  + Sagas can be implemented using either choreography (event-driven) or orchestration (central controller).
* **Use Case:** Suitable for distributed transactions where consistency across microservices is required, such as order processing systems.

**9. Circuit Breaker Pattern**

* **Purpose:** The Circuit Breaker pattern protects microservices from cascading failures by temporarily blocking requests to a service that is failing.
* **How It Works:**
  + Implement the Circuit Breaker pattern using Spring Cloud Netflix Hystrix (deprecated) or Resilience4j.
  + If a microservice fails, the circuit breaker trips, and subsequent calls to that service are blocked until it recovers.
* **Use Case:** Necessary for maintaining system stability, especially when decomposing a monolith into microservices where service dependencies are common.

**10. Bulkhead Pattern**

* **Purpose:** The Bulkhead pattern isolates different parts of the system to prevent failure in one part from affecting the others.
* **How It Works:**
  + Apply the Bulkhead pattern in Spring Boot by setting up isolated thread pools for each microservice or group of services.
  + This ensures that if one microservice becomes overloaded, it doesn’t consume all the resources, allowing other services to continue operating.
* **Use Case:** Useful in microservices architecture to ensure that a failure in one service doesn't lead to a system-wide outage.

**Steps to Convert a Monolithic Application to Microservices in Java**

**1. Analyze and Understand the Existing Monolith**

* **Assess the Application**: Identify dependencies, modules, and tightly coupled components.
* **Break Down Functionalities**: Identify business capabilities that can be isolated.
* **Check Performance Bottlenecks**: Identify database-heavy operations and optimize them.

**Tools to use**:  
✅ Code analysis tools (SonarQube, Static Analysis, JDepend)  
✅ Profiling tools (JProfiler, VisualVM)

**2. Define Microservices Boundaries (Domain-Driven Design)**

* **Identify Bounded Contexts**: Use **Domain-Driven Design (DDD)** to define business modules as separate services.
* \*\*Apply the **Single Responsibility Principle (SRP)**: Each microservice should have a well-defined purpose.
* **Create API Contracts**: Define RESTful or gRPC APIs for communication.

**Tools to use**:  
✅ Event Storming, Context Mapping (DDD)  
✅ OpenAPI/Swagger for API documentation

**3. Identify Data Strategy**

* **Decouple the Database**: Avoid shared databases; each service should have its own schema.
* **Use Polyglot Persistence**: Choose databases per service (SQL for transactions, NoSQL for scalability).
* **Implement Data Synchronization**: Use **Change Data Capture (CDC)** or event-driven architecture.

**Tools to use**:  
✅ Apache Kafka, Debezium (CDC)  
✅ PostgreSQL, MongoDB, Cassandra

**4. Extract Services Incrementally**

* **Strangle Pattern**: Gradually replace monolith functionality with microservices.
* **API Gateway**: Introduce a **gateway** to route traffic to new services while keeping the monolith running.
* **Refactor Modules**: Start with less critical, loosely coupled modules (e.g., authentication, reporting).

**Tools to use**:  
✅ Spring Cloud Gateway, Kong API Gateway  
✅ Feature Toggles (LaunchDarkly, Unleash)

**5. Implement Inter-Service Communication**

* **Synchronous Communication**: Use REST APIs or gRPC for low-latency calls.
* **Asynchronous Communication**: Use **message brokers** (Kafka, RabbitMQ) to reduce tight coupling.

**Best Practices**:  
✅ Prefer **event-driven** architecture over direct service calls.  
✅ Use **Circuit Breakers** to handle failures (**Resilience4j, Hystrix**).

**6. Implement Service Discovery & Load Balancing**

* **Service Discovery**: Use **Eureka**, **Consul**, or **Kubernetes Service Registry** for automatic service registration.
* **Load Balancing**: Use **Ribbon** (Spring Cloud LoadBalancer) or API gateways for efficient request distribution.

**Tools to use**:  
✅ Netflix Eureka, Consul, Kubernetes Service Registry  
✅ Spring Cloud LoadBalancer

**7. Implement Security & Authentication**

* **Use OAuth2 / JWT**: Secure APIs with **Spring Security & Keycloak**.
* **Enable Service-to-Service Authentication**: Use **mutual TLS (mTLS)** or API Gateway security.
* **Manage Secrets**: Store secrets securely in **Vault, AWS Secrets Manager, or Kubernetes Secrets**.

**Tools to use**:  
✅ OAuth2, Keycloak, Okta, Spring Security  
✅ HashiCorp Vault, AWS Secrets Manager

**8. Implement Monitoring & Logging**

* **Centralized Logging**: Use **ELK Stack (Elasticsearch, Logstash, Kibana)** or **Grafana Loki**.
* **Distributed Tracing**: Track service calls using **Jaeger** or **Zipkin**.
* **Metrics & Alerts**: Monitor services with **Prometheus & Grafana**.

**Tools to use**:  
✅ ELK Stack, Prometheus, Grafana  
✅ Jaeger, Zipkin

**9. Deploy Microservices with CI/CD & Kubernetes**

* **Containerize Services**: Use **Docker** for packaging services.
* **Orchestrate with Kubernetes**: Deploy services using **Helm, Kubernetes ConfigMaps, and Secrets**.
* **Implement CI/CD Pipelines**: Automate builds with **Jenkins, GitHub Actions, or GitLab CI/CD**.

**Tools to use**:  
✅ Docker, Kubernetes, Helm  
✅ Jenkins, GitHub Actions

**10. Optimize Performance & Scale**

* **Enable Auto-Scaling**: Use **Kubernetes Horizontal Pod Autoscaler (HPA)**.
* **Optimize API Performance**: Use caching (**Redis, AWS ElastiCache**).
* **Database Partitioning & Replication**: Optimize database with **sharding** and **replication**.

**Tools to use**:  
✅ Kubernetes HPA, Redis, Hazelcast  
✅ PostgreSQL Replication, Cassandra

**Best Practices for Writing Microservices**

**1️ Design & Architecture Principles**

✅ **Follow Single Responsibility Principle (SRP):** Each microservice should focus on a single business capability.  
✅ **API-First Approach:** Design APIs using OpenAPI/Swagger before implementation.  
✅ **Decentralized Data Management:** Each microservice should have its own database to avoid dependencies.  
✅ **Event-Driven Architecture:** Use Kafka, RabbitMQ, or SNS/SQS for async communication.  
✅ **Implement Circuit Breaker:** Use Resilience4j or Hystrix to handle failures gracefully.  
✅ **CQRS & Event Sourcing (if needed):** Separate read and write models for scalability.

**2️ Communication Between Microservices**

✅ **Use RESTful APIs or gRPC:** Choose based on performance and compatibility needs.  
✅ **Asynchronous Messaging for Scalability:** Use Kafka, RabbitMQ, or AWS SQS for async processing.  
✅ **Service Discovery & Load Balancing:** Use **Eureka, Consul, or Kubernetes Service Discovery**.  
✅ **API Gateway for Routing & Security:** Implement **Spring Cloud Gateway, Kong, or Nginx**.

**3️ Security Best Practices**

✅ **Use OAuth2 & JWT for Authentication:** Secure APIs with Spring Security & Keycloak.  
✅ **Encrypt Sensitive Data:** Store secrets using **AWS Secrets Manager, HashiCorp Vault, or Kubernetes Secrets**.  
✅ **Rate Limiting & Throttling:** Prevent abuse using API Gateway (e.g., Kong, Apigee, or AWS API Gateway).  
✅ **Use Role-Based Access Control (RBAC):** Ensure fine-grained access control for different users.

**4️ Database & Storage**

✅ **Polyglot Persistence:** Use SQL (PostgreSQL, MySQL) & NoSQL (MongoDB, Cassandra) based on use cases.  
✅ **Avoid Distributed Transactions:** Instead, use **Saga Pattern or Eventual Consistency**.  
✅ **Cache Data for Performance:** Use Redis or Memcached to reduce database load.  
✅ **Optimize Query Performance:** Use proper indexing strategies.

**5️ Deployment & Scalability**

✅ **Use Containerization:** Deploy services using Docker.  
✅ **Orchestrate with Kubernetes:** Auto-scale services using **K8s Horizontal Pod Autoscaler**.  
✅ **Blue-Green & Canary Deployments:** Minimize downtime with **progressive rollouts**.  
✅ **Infrastructure as Code (IaC):** Use Terraform, CloudFormation, or Helm for managing environments.

**6️ Logging, Monitoring & Observability**

✅ **Centralized Logging:** Use ELK (Elasticsearch, Logstash, Kibana) or AWS CloudWatch.  
✅ **Distributed Tracing:** Implement **Spring Cloud Sleuth & Zipkin** or **Jaeger**.  
✅ **Metrics & Alerts:** Monitor system health with **Prometheus & Grafana**.  
✅ **Debug with Correlation IDs:** Pass a unique request ID across services for debugging.

**7️ Testing & Quality Assurance**

✅ **Unit Testing:** Use JUnit, Mockito for isolated testing.  
✅ **Integration Testing:** Test API interactions with **TestContainers or WireMock**.  
✅ **Contract Testing:** Use **Pact** for verifying API contracts between services.  
✅ **Performance Testing:** Use **JMeter or Gatling** to test system load.

**8️ CI/CD & DevOps Integration**

✅ **Automated CI/CD Pipelines:** Use GitHub Actions, Jenkins, GitLab CI/CD, or AWS CodePipeline.  
✅ **Zero-Downtime Deployment:** Implement **Rolling Updates with Kubernetes**.  
✅ **Security Scanning in CI/CD:** Use SonarQube, Checkmarx for code security analysis.  
✅ **Rollback Strategies:** Enable **Feature Flags (LaunchDarkly, Togglz)** for controlled rollouts.