DECiMAL POiNT

Innovative Data & Research Solutions

MICROSERVICES

-Documentation by Nehitha

Submitted to:

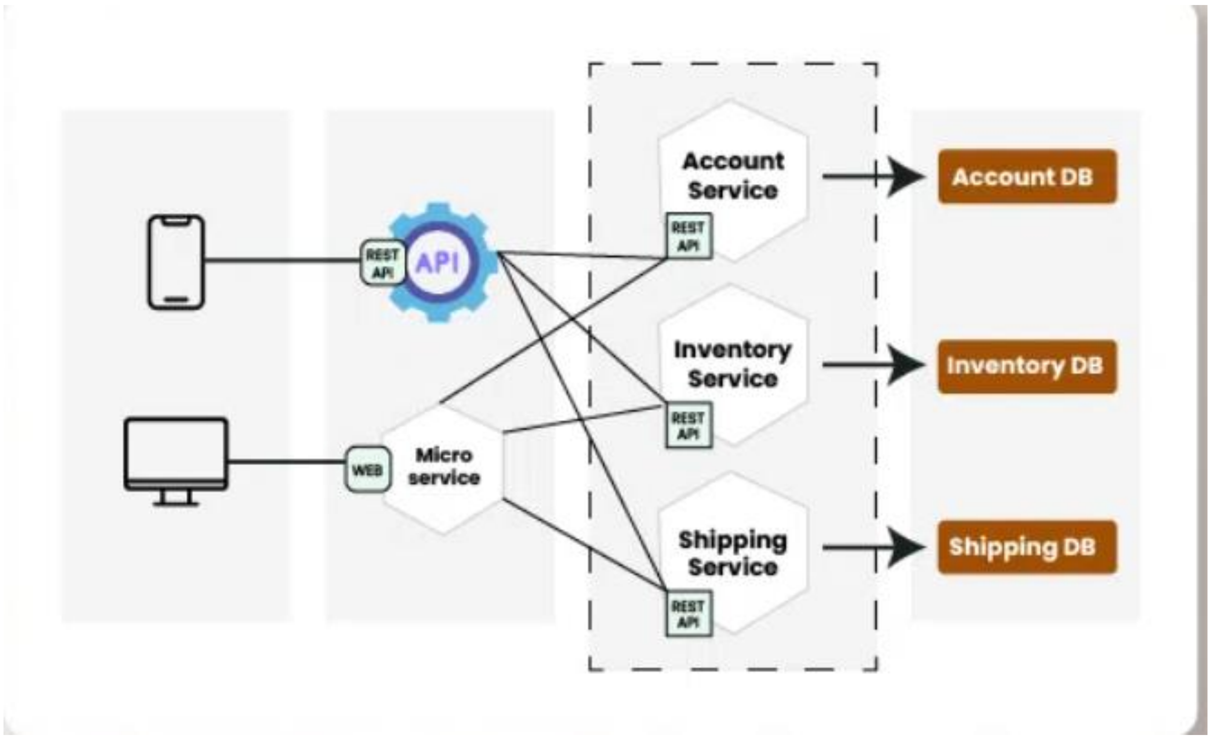
Akshay Kulkarni

Muskan Agarwal

Index of Document

What are Microservices:

Microservices are an architectural approach where software applications are developed as a collection of small, independent services. Unlike monolithic applications with tightly integrated functionality, microservices break the application into loosely coupled services that communicate over a network.



How do microservices work:

Microservices improve flexibility, scalability, and maintenance by dividing a large, complex applications into smaller, autonomous services that interact and communicate with one another. Here's a quick rundown:

Modular Structure: To facilitate development and maintenance, applications are separated into self-contained modules, each with a distinct business capability.

Independent Functions: Each service manages a particular task, like product catalog management or user authentication, enabling customized development and upkeep.

Communication: Services communicate through clearly defined APIs to ensure flexibility and interoperability.

Flexibility: Teams can select the most appropriate tools for their jobs by using various services that employ a range of technologies.

Independence and Updates: Services are more resilient and less likely to experience disruptions when they are updated independently of the system as a whole.

Scalability: The capacity of a service to be individually scaled to meet rising demand is essential for adapting.

Main components of microservices:

The main components of microservices include:

Microservices: Small, independent services that each handle a specific business function.

API Gateway: Manages and routes requests from clients to the appropriate microservices, often handling tasks like authentication, rate limiting, and load balancing.

Service Registry: Maintains a list of available services and their instances, enabling dynamic discovery and routing of services.

Configuration Management: Centralized management of configuration settings for microservices, ensuring consistency and simplifying updates.
Event Bus/Message Broker: Facilitates asynchronous communication between services by allowing them to publish and subscribe to events, decoupling their interactions.

Database per Service: Each microservice manages its own database, promoting data encapsulation and reducing dependencies between services.
Monitoring and Logging: Tools and frameworks to track the health, performance, and behaviour of services, providing insights for maintenance and debugging.

CI/CD Pipeline: Continuous integration and continuous deployment tools that automate the building, testing, and deployment of microservices, supporting rapid and reliable updates.
Security: Mechanisms to ensure secure communication and data protection between microservices, including authentication, authorization, and encryption.

Service Mesh: A dedicated infrastructure layer that handles service-to-service communication, providing advanced features like load balancing, failure recovery, and observability

Design Patterns of Microservice:

Database per Service: Each microservice has its own database to ensure data encapsulation and independence.

API Gateway: A single entry point for all clients, managing requests and routing them to the appropriate microservices, often handling cross-cutting concerns like authentication and rate limiting.

Circuit Breaker: Prevents cascading failures by stopping requests to a failing service and allowing it to recover before resuming communication.

Service Registry and Discovery: A centralized registry where services register themselves and discover other services, enabling dynamic and flexible communication.

Event Sourcing: Captures all changes to the application state as a sequence of events, providing a reliable and auditable way to manage state changes.

Saga: Manages distributed transactions by coordinating a series of local transactions in each service, ensuring data consistency across services.

Strangler: Incrementally replaces parts of a monolithic application with microservices, allowing for a gradual transition to a microservices architecture.

Sidecar: Deploys auxiliary components (e.g., logging, monitoring) alongside the main service, enabling common functionalities without modifying the service code.

Service Mesh: Manages service-to-service communication with features like load balancing, failure recovery, and observability, abstracting network complexities.

CQRS (Command Query Responsibility Segregation): Separates read and write operations into different models, optimizing for scalability and performance.

Decomposition Patterns:

By Business Capability: Breaks down services based on distinct business functions.
By Subdomain: Uses domain-driven design to decompose services based on subdomains within a larger business domain.
Bulkhead: Isolates services into separate pools to prevent failures in one service from cascading to others, enhancing resilience.

Anti-patterns in Microservices:

Monolith in Microservices:
When building microservices, preserving a monolith architecture can hinder scalability and fault tolerance due to:

1. **Shared Databases:** Use a separate database for each service to manage database type, schema rules, and IOPS capacity individually, enhancing scalability.
2. **Complex Deployment**: Despite smaller services, if deployment remains intricate and manual, it negates microservices' agility benefits. Automate and streamline deployment processes.
3. **Inadequate Service Boundaries:** Clearly define service boundaries and ownership using Domain-Driven Design (DDD) to avoid overlapping functionalities and duplication.

Avoid monolith pitfalls by implementing individual databases per microservice and employing DDD for clear service ownership.

Chatty Microservices:

Frequent Inter-Service Communication

Issue: Sending numerous requests for minor tasks or small data increments network traffic and response delay.
Solution: Aggregate requests and responses to reduce the number of calls.

Fine-Grained APIs
Issue: Numerous API calls to complete a single user request increase serialization, network overhead, and blocking I/O operations.
Solution: Design more coarse-grained APIs encapsulating multiple operations within a single call.

Cascade of Calls
Issue: A user request triggering a series of dependent calls can cause system-wide failures if one service fails or is delayed.
Solution: Implement circuit breakers and fallback mechanisms to handle failures gracefully.

Distributed Monolith:

A "Distributed Monolith" is an anti-pattern where a system is designed as a distributed application but behaves like a monolith due to tight coupling between its components. This anti-pattern can be identified by several key characteristics:

1. **Lack of Service Autonomy**: Components depend heavily on each other, making independent scaling, deployment, and evolution difficult.
2. **Complex Interdependencies**: Services are interwoven with intricate dependencies, creating a web of relationships that increases complexity and risk.
3. **Shared State:** Services share state directly, leading to data consistency issues, race conditions, and scaling challenges.

Over – Microservices:
"Over-Microservices" is an anti-pattern where the system is excessively fragmented into microservices, each encapsulating a very small portion of functionality. This can lead to several issues:

Excessive Fragmentation: The system is divided into too many microservices, making it overly complex.

Low Cohesion: Microservices lack logical coherence, causing scattered and fragmented business logic.

High Coupling: Despite partitioning, services are tightly coupled, with inter-service interactions and dependencies that complicate changes and increase risk.

To address this, adopt Domain-Driven Design (DDD). DDD helps you create microservices that align with specific domains of your application, ensuring each service is cohesive and appropriately scoped. For a detailed guide on DDD, check this out.

Single Responsibility Principle:
Violating the Single Responsibility Principle (SRP) occurs when a function or microservice handles multiple responsibilities that should be separated. For example, a payment processing service also manages user registration.

This anti-pattern can arise from:

1. **Lack of Design Principles Awareness**: Developers may not fully understand or prioritize design concepts like SRP.
2. **Inadequate Planning**: Poor planning or analysis can lead to unclear roles for components, causing mixed responsibilities.
3. **Misinterpretation of Requirements:** Misunderstanding requirements can introduce unnecessary functionalities, violating SRP.

Spaghetti Architecture:
Spaghetti Architecture" is an anti-pattern where software lacks clear structure and organization, resulting in a tangled mess of components. Key characteristics include:

1. **Lack of Separation of Concerns**: Different concerns (business logic, presentation logic, data access) are mixed within the same components, leading to poor modularity.
2. **Complex Control Flow:** The control flow is convoluted with hard-to-track dependencies and interactions, making the system unpredictable and difficult to understand.
3. **High Coupling:** Components are tightly connected, so changes in one often necessitate changes in others, creating a ripple effect and increasing system fragility.

Distributed Data Inconsistency:
"Distributed Data Inconsistency" occurs when data is replicated across multiple nodes or services, leading to inconsistencies due to synchronization delays or failures. This results in incorrect or outdated information being accessed, causing incorrect behaviour or data corruption.
Key characteristics include:

1. **Asynchronous Updates:** Delays in data updates across replicas.
2. **Network Partitions:** Network failures prevent updates from reaching all replicas, causing discrepancies.
3. **Conflicting Operations**: Concurrent operations on the same data cause conflicts and inconsistencies.

To address these issues, use Microservice patterns like the Saga Pattern to manage distributed transactions effectively in your microservices.

Tight coupling:
Tight coupling in microservices leads to heavy dependencies and scaling issues, contributing to several anti-patterns:

1. Monolithic Architecture
2. Spaghetti Architecture
3. God Object
4. Distributed Data Inconsistency
5. Vendor Lock-in

Lack of Observability:
Lack of Observability is an anti-pattern where the application fails to provide adequate insight into its internal state, operations, and performance, making troubleshooting difficult.
Key characteristics include:

1. **Limited Logging**: Insufficient logging to capture significant events and errors, hindering problem identification.
2. **Inadequate Metrics:** The lack of useful metrics on performance and resource use makes it hard to assess system health.
3. **Sparse Tracing**: The absence of distributed tracing complicates the detection of performance issues and breakdowns in distributed systems.

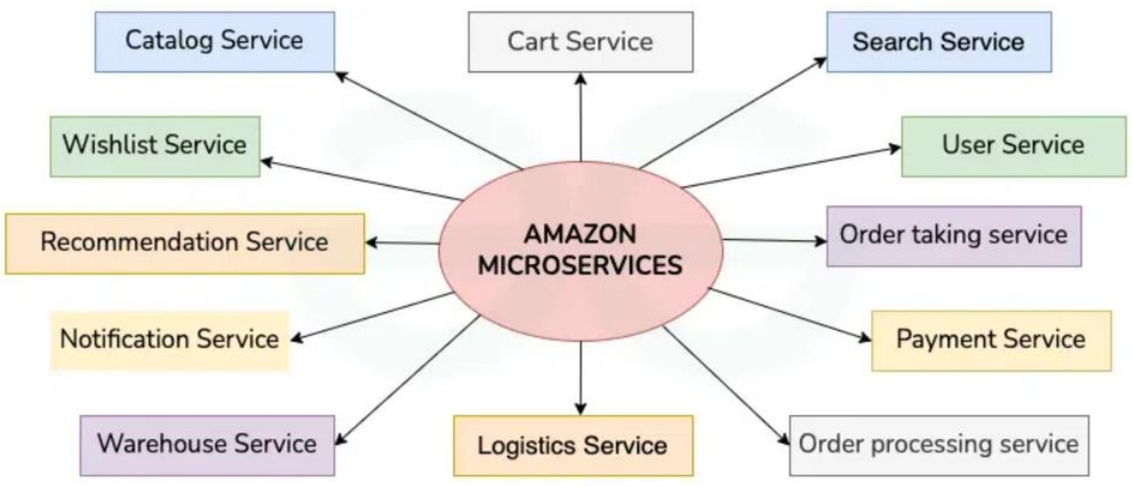
Ignoring the Human Cost:
Ignoring the Human Cost is an anti-pattern focused solely on technical goals and deadlines, neglecting the team's well-being.

Key characteristics include:

1. **Overworking:** Frequent extended hours leading to burnout and decreased productivity.
2. **Unrealistic Expectations:** Timelines and deliverables set without considering team resources and capabilities, causing undue pressure.

3. **Micromanagement:** Excessive control undermining autonomy and creativity.
4. **Lack of Support:** Team members feel unsupported, increasing stress and disengagement.

Real World application of microservices:



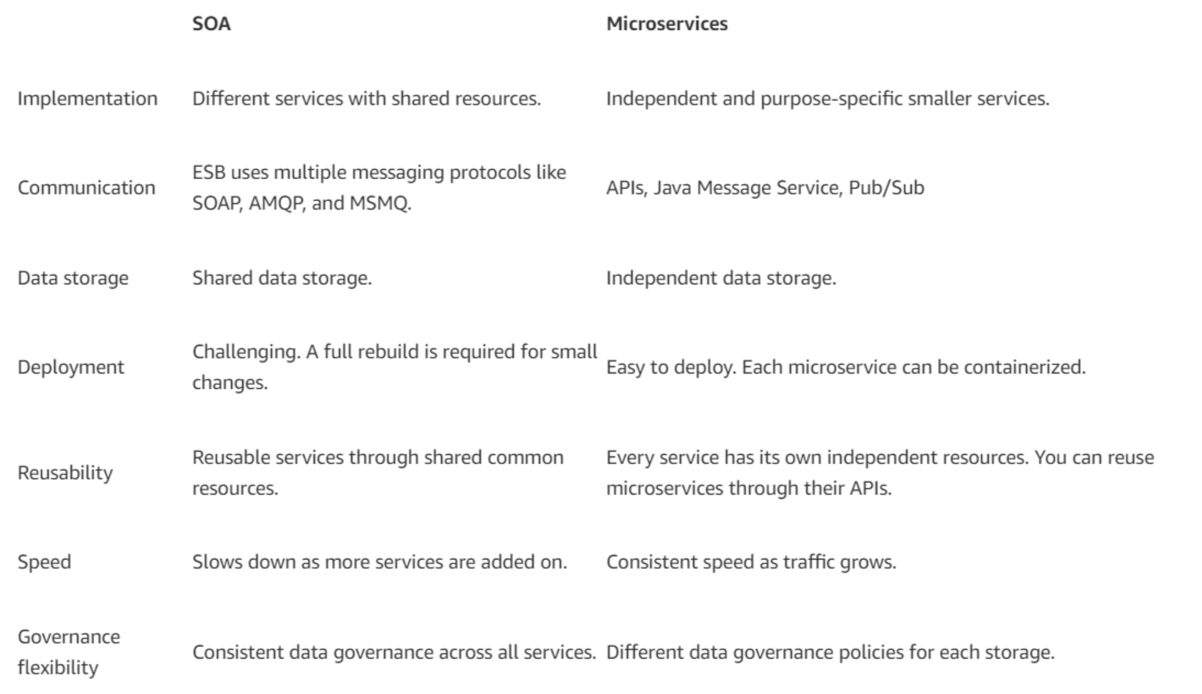
How to move from Monolithic to Microservices?

Steps to Transition from Monolith to Microservices

1. Evaluate Monolith: Analyse the existing application and identify components for migration.
2. Define Microservices: Break down the monolith into distinct business capabilities.
3. Strangler Pattern: Gradually replace monolithic parts with microservices.
4. API Definition: Clearly define APIs for seamless communication.
5. CI/CD Implementation: Set up automated testing and deployment pipelines.
6. Decentralize Data: Move to a database-per-service model.
7. Service Discovery: Implement mechanisms for dynamic microservices communication.

8. Logging and Monitoring: Centralize logging and monitoring for performance visibility.
9. Cross-Cutting Concerns: Consistently manage security and authentication.
10. Iterative Improvement: Continuously refine and expand microservices based on feedback.

Service-Oriented Architecture (SOA) vs. Microservices Architecture:



	SOA	Microservices
Implementation	Different services with shared resources.	Independent and purpose-specific smaller services.
Communication	ESB uses multiple messaging protocols like SOAP, AMQP, and MSMQ.	APIs, Java Message Service, Pub/Sub
Data storage	Shared data storage.	Independent data storage.
Deployment	Challenging. A full rebuild is required for small changes.	Easy to deploy. Each microservice can be containerized.
Reusability	Reusable services through shared common resources.	Every service has its own independent resources. You can reuse microservices through their APIs.
Speed	Slows down as more services are added on.	Consistent speed as traffic grows.
Governance flexibility	Consistent data governance across all services.	Different data governance policies for each storage.

Benefits of using Microservices architecture:

Modularity and Decoupling:
Independent Development: Teams can work on and deploy services independently.
Isolation of Failures: Issues in one service don't impact others.

Scalability:
Granular Scaling: Scale each service based on its needs.
Elasticity: Adapt to varying workloads by scaling services dynamically.

Technology Diversity:

Freedom of Technology: Use the best technology stack for each service.

Autonomous Teams:
Team Empowerment: Small teams can independently manage specific services.
Reduced Coordination Overhead: Services can be released and updated with minimal inter-team coordination.

Rapid Deployment and Continuous Delivery:
Faster Release Cycles: Independent development and deployment facilitate quicker releases.
Continuous Integration and Deployment (CI/CD): Automation tools support continuous integration and deployment.

Easy Maintenance:
Isolated Codebases: Smaller codebases are easier to manage and troubleshoot.
Rolling Updates: Update or roll back services without affecting the whole application.

Challenges of Microservices Architecture:

Complexity of Distributed Systems: Managing inter-service communication, network latency, and data consistency is challenging.

Increased Development and Operational Overhead: More effort is required for developing, testing, deploying, and monitoring numerous services.

Inter-Service Communication Overhead: Network communication between services can increase latency and complexity in managing protocols and data transfer.

Data Consistency and Transaction Management: Ensuring data consistency and managing distributed transactions is complex, often complicating traditional ACID transactions.

Deployment Challenges: Coordinating deployments of multiple interdependent services requires careful planning to ensure consistency and avoid downtime.

Monitoring and Debugging Complexity: Identifying issues involves tracing across multiple services, necessitating centralized logging for effective debugging.

Role of Microservices in DevOps:

Enhanced Collaboration:
Autonomous Teams: Small, cross-functional teams manage specific services independently, fostering collaboration and ownership.

Continuous Integration/Continuous Deployment (CI/CD):
Faster Release Cycles: Microservices enable frequent, independent deployments, facilitating faster and more reliable software delivery.
Automation: Supports automation of testing, integration, and deployment processes, aligning with DevOps practices.

Scalability and Flexibility:
Granular Scaling: Each microservice can be scaled independently, optimizing resource utilization and enhancing flexibility.

Improved Fault Isolation:
Service Independence: Faults in one microservice do not affect others, improving overall system reliability and ease of maintenance.

Technology Diversity:
Optimal Tooling: Teams can choose the best tools and technologies for each microservice, promoting innovation and efficiency.

Monitoring and Logging:
Centralized Monitoring: Essential for tracking the health and performance of individual services, aiding in quick detection and resolution of issues.
Enhanced Debugging: Centralized logging helps trace issues across services, improving debugging processes.

Agility and Speed:
Rapid Iteration: Microservices enable quick iterations and updates, allowing for faster adaptation to changing requirements and market conditions.

By embracing microservices, DevOps teams can achieve greater agility, scalability, and efficiency, aligning closely with DevOps principles of continuous improvement and rapid delivery.

Technologies for microservices

Docker and Kubernetes.
JAVA
RabbitMQ
Prometheus.
.NET Core
. NET Core
Consul
Consul.
REST API
REST API
AUTOMATION
Docker and Kubernetes.
Python