Nguyen Huu Chi - 617310

Introduction to Microservices

The goal of this assignment is to cover the basics of microservices, their benefits and challenges, popular tools, and case studies of their implementation.

Principles of Microservices

**Decentralization**: Services are developed, deployed, and managed independently. Different teams handle different microservices, that allow faster development cycles and improve ownership. Teams also can choose different tech stacks that best suit their business requirements. This flexibility promotes innovation and faster problem-solving.

**Componentization**: Services are treated as independent components that can be easily replaced and upgraded. Microservices break down an application into smaller services that map directly to business needs. They are highly focused, self-contained components with clear boundaries context, they are loose coupling and can be reused across different applications. Compared to Monolith usually tightly coupled and difficult to reuse in other applications.

**Autonomy**: Teams work independently on each service, reducing the coordination overhead.

Teams can self-decide the programming languages, frameworks and databases that best suit the needs of their microservice. Teams can develop, test and deploy independently, leading to faster development cycles and improve time-to-market. Individual services can be scaled independently based on their specific needs. If one microservice fails, it has minimal impact on other services. This isolation improves overall application resilience and availability.

**Technology Diversity**: Teams can choose the best tool for their specific needs, fostering innovation. Teams have freedom of choice and select the most appropriate tech stack for each microservice based on its business requirement. For instance, choosing Python for data science tasks, Node.js for real-time applications. This enables specialization and Performance optimization overall.

Compared to Monolithic architecture, limited tech stack including programming languages, frameworks and databases, the chosen technology stack has to be suitable for all functionalities within the application, which might not be ideal for some specific features.

Advantages of Microservices

**Scalability**: Services can be scaled independently, allowing for more efficient use of resources.

In monolithic architecture applications have to scale up by adding more memory and CPU to the existing servers, it will reach to certain limitations at some point. While in Microservices Architecture we focus on Horizontal scaling (Scale-out), individual services can be scaled independently by adding more instances. This allow for efficient allocation of resources based on actual demand, help to reducing the cost. In the other hand, by scaling only the services under high load, we ensure that application performance is optimal and improving overall availability.

**Resilience**: Faults in one service do not impact others, improving overall system robustness.

Failure in one microservice is less likely to impact other services due to loose-coupling and isolation. The overall application remains partially functional. Microservices can be designed to automatically restart or failover to healthy instances in case of failures, minimizing downtime. Updates and bug fixes can be deployed to individual services without affecting the entire application, reducing downtime and rollbacks risks.

**Technological Agility**: Allows the adoption of new technologies and processes without overhauling the entire system.

We can experiment with new technologies in specific microservices without impacting the entire application. This fosters faster innovation and adoption of new tools and frameworks. Improved developer productivity by having smaller codebases in each microservice allow for shorter development cycles and easy to maintain. Microservices also enable the use of different technologies for each service, allow quicker development and deployment, faster delivery of new features and functionalities.

Challenges of Microservices

**Complexity**: Increased operational and management complexity.

* Increased operational: Monitoring, logging, tracing and maintaining the health of numerous services requires additional tools and processes. Identifying the root cause of issues across multiple services can be more complex compared to monolithic applications.
* Management complexity: Managing numerous independent services, their interactions and communication can be painful. Ensuring data consistency across the system can be challenging and requires good design and implementation. While offering advantages, managing a diverse tech stack with different languages and frameworks can add complexity.

**Data Integrity**: Ensuring data consistency across services can be challenging.

In Microservices, data can be spread across multiple databases owned by individual services. Maintaining perfect data consistency across all databases can be complex and might require eventual consistency approaches. This means data might not be immediately reflected across all services, but eventually becomes consistent.

We have several strategies to address Data Integrity issue in Microservices:

* Data Ownership: Define clear ownership of data entities by specific microservices.
* API Contracts: Establish clear contracts between microservices for data exchange and updates, ensuring consistency and reducing errors.
* Data Consistency Patterns: Implement patterns like Saga or Two-Phase Commit (when applicable) to manage data consistency across transactions involving multiple microservices.
* Data Validation: Enforce data validation rules within each microservice to prevent invalid data from entering the system.

**Network Issues**: Dependency on network latency and load balancing.

Since microservices are self-contained and run independently, communication between them can generate more network traffic compared to a monolithic application. This can amplify the impact of network congestion or bandwidth limitations. If a critical service experienced network issues it will impact other services rely on its data/functionalities. To mitigate the risk, we can apply the following patterns:

* API Gateway: An API Gateway can act as a single-entry point, potentially masking network issues with specific microservices from clients.
* Service Discovery: Dynamic service discovery can help route requests around unavailable services, improving overall application resilience.
* Circuit Breaker Pattern: Implementing a circuit breaker pattern can automatically stop sending requests to a failing microservice, preventing further overload and allowing for retries after a timeout.

**Skill Set**: Requires a broad set of skills from development teams, including DevOps capabilities.

Developers may need to be familiar with a wider range of programming languages and frameworks as different microservices can be built with different technologies. Also need to have strong skills in designing and implementing APIs because it’s crucial for communication and data exchange between microservices.

Understanding of distributed systems concepts like service discovery, communication protocols, and eventual consistency is essential, including automation tools that Microservice rely on and DevOps practices for continuous integration, delivery, and monitoring of numerous services.