Report: Microservices

Summary:

This study aims to explain what microservices are and how they're different from the usual monolith architecture. It covers the basics and what they focus on. The study is from the early days when microservices were not so well-known. It's short and gives a quick overview, but it doesn't get into the nitty-gritty details of how to use them or the good and bad sides of working with them.

Keywords:

micro-services characteristics

Research Questions:

1. **What are the most common characteristics for microservices architecture?**

Key Points:

➔ **Componentization via Services**

- **Primary method of componentizing** - breaking down into services.
- **Libraries** - components linked into a program, called using in-memory function calls.
- **Services** - out-of-process components communicating through mechanisms like web service requests or remote procedure calls.
- Advantages of using services instead of libraries is, that they are independently deployable and provide more explicit component interface
- But remote calls are more expensive than in-process calls and remote APIs may need to be coarser-grained *[5]*

➔ **Organised around Business Capabilities**

- Traditionally teams are split into technology layers: UI teams, server-side logic teams, and database teams. Changes may require cross-team collaboration, leading to time and budget approvals.
- Microservices prefers cross-functional teams with a full range of development skills: user experience, database, and project management. *[6-7]*

➔ **Products not Projects**

- Traditionally aim is to deliver a completed piece of software and handover it to a maintenance organisation after completion.
- In case of microservices, teams own a product over its full lifetime, developers stay in day-to-day contact with software behaviour in production. *[8]*

➔ **Smart endpoints and dumb pipes**

- Microservices aim for high decoupling and cohesion, services own their own domain logic, acting as filters like classical Unix programs.
- Most common approach is to use simple REST protocols for choreography instead of complex ones like WS-Choreography or BPEL, most commonly used protocols are HTTP request-response with resource APIs and lightweight messaging.
- Second approach is messaging, often done over a lightweight message bus; infrastructure is typically "dumb," acting as a message router (e.g., RabbitMQ or ZeroMQ).
- Smart functionality resides in the end points (services) that produce and consume messages. *[8-10]*

➔ **Decentralised Governance**

- Monolithic applications tend to standardise on single technology platforms, experience indicates this approach can be constraining.
- Microservices allows for choices in technology when building each service. *[10]*

➔ **Decentralised Data Management**

- Microservices prefer for each service to manage its own database.
- Embrace Polyglot Persistence, allowing different instances or entirely different database systems. *[12]*

➔ **Infrastructure Automation**

- Many microservices products/systems developed by teams with extensive Continuous Delivery and Continuous Integration experience.
- Utilise infrastructure automation techniques for efficiency.
- Emphasis on Automated Testing. *[13-14]*

➔ **Design for failure**

- Applications must be designed to tolerate service failures.
- Emphasis on graceful response to potential service unavailability.

- Prioritisation of real-time monitoring.
- Semantic monitoring serves as an early warning system, prompting development teams to investigate potential issues. *[15]*

➔ **Evolutionary Design**

- Emphasis on independent replacement and upgradeability.
- Components can be rewritten without impacting its collaborators.
- Many microservice groups explicitly anticipate scrapping services rather than evolving them in the longer term. *[16]*

➔ **Challenges associated with microservices**

- Defining the right boundaries can be challenging, it requires a deep understanding of the system's functionality and the business domain
- Microservices rely on remote communication, which can introduce latency, network failures, and other issues.
- Changes in microservices often require coordination between teams that own different services.
- Testing becomes more complex, as changes may have cascading effects on other services.
- Ensuring that the entire system behaves correctly after an update becomes a challenging task.
- Maintaining backward compatibility becomes crucial for allowing a smooth transition during updates.
- If components don't integrate smoothly, you're essentially transferring complexity from within each component to the connections between them. *[19]*