SAVONIA
University of Applied Sciences

THESIS – BACHELOR'S DEGREE PROGRAMME
TECHNOLOGY, COMMUNICATION AND TRANSPORT

MICROSERVICES ARCHITECTURE

Practical implementations, benefits, and nuances

AUTHOR Anton Korotenko

Field of Study Technology, Communication and Transport	
Degree Programme Degree Programme in Internet of Things	
Author(s) Anton Korotenko	
Title of Thesis Microservices Architecture. Practical implementations, benefits, and nuances	
Date 17.12.2023	Pages/Number of appendices 33
Client Organisation /Partners -	

Abstract

The goal of the thesis was to conduct comprehensive research on the implementations, adoption, and evolution of microservices architecture, with a focus on C# and .NET frameworks. The study aims to help organizations and developers who are considering transitioning to microservices architecture.

The research methodologies included historical analysis, a comparative study, and an examination of real-world cases and generally accepted approaches. Moreover, the intention was to emphasize the benefits of practically utilizing resources and ideas provided in the study.

As a result of this thesis, the architecture's historical evolution was shown, explaining the reasons for different changes. The case studies were analyzed, providing insightful typical failures and challenges in adopting a microservices architecture. A suitability table and an objectively created roadmap were provided, aiming to help with organizations' and developers' decisions. The work can be taken as a skeleton for those who want to adopt a microservices architecture.

Keywords
Microservices Architecture, C# and .NET Core, Software Development, Cloud Computing, Containerization, Docker, Kubernetes, Monolithic Architecture, Service-Oriented Architecture, DevOps, Scalability, Software Architecture Transition, Agile Development, Cloud Technologies

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

The information technology field is evolving rapidly, which provokes every engineer around the world to adapt to new changes. So, to keep up with the pace, this thesis deepens into the software development sphere and particularly into the domains of microservices architecture. We will start simple, go through key terms, and historical development and move on to the main part of the thesis.

Microservices Architecture - is a method of developing software applications as a suite of small, independent services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. This approach is the opposite of the traditional Monolithic Architecture, where an application is built as a single, indivisible unit, often leading to challenges in scalability and agility.

Scalability and agility here refer to two crucial attributes in software architecture.
Scalability can be explained as the ability of the application to handle increasing loads or to be easily expanded to handle new workloads. In the context of microservices, this means each independent service can be scaled (increased or decreased in capacity) independently of others. This is particularly beneficial when specific components of an application can experience varying demands.
Agility, on the other hand, refers to the flexibility and speed with which changes can be implemented in the software. In a microservices architecture, because the application is broken down into smaller, independent services, changes can be made to a single service without impacting others. Such an approach allows for rapid deployment of new features, enhancing the ability to respond swiftly to market changes and customer needs.

The historical journey from monolithic to microservices architectures shows how the industry responded to evolving business needs and technological advancements. Everything started from simplicity and straightforward development, where the focus was on deployment. However, as applications grew in complexity and user demands expanded, the monolithic model showed its limitations. This led to the introduction of Service-Oriented Architecture (SOA), which further evolved into what we now understand as microservices architecture, where the rise of cloud computing and containerization technologies like Docker significantly influenced its development.

My thesis aims to thoroughly explore the implementation of microservices architecture. The study starts from the basic principles of microservices architecture and provides a perspective on the past and future of microservices architecture, discussing its sustainability with modern business strategies and technological advancements.

As we progress through the study, we will review the complexities, challenges, and solutions that one might encounter when transitioning to microservices. The final section unfolds into a detailed roadmap for their adoption. This roadmap includes all the critical aspects of development processes, deployment, and maintenance.

2 LITERATURE REVIEW

The literature review chapter aims to explore already scientifically proven research papers about different topics involving Microservices Architecture to build a strong idea about the latter. Summarising the achievements and ideas of the articles will help to better understand concepts discussed further in the study.

2.1 Resilience of Microservice's Network-link

Automated Testing and Resilience of Microservice's Network-link using Istio Service Mesh research conducted by Kanth, Heikkonen and others (2022) emphasizes the concern about increased network complexity and the risks of failure due to faults in service communications protocols, which often occurs as a matter of overwhelming nature of Microservices Architecture. The study states that smooth and fast network link testing procedures are required in order to build a resilient application. It points to services' dependencies with each other and that a problem in one service can further affect other services linking to the whole application going down.

Therefore, to deal with this matter, a tool, independent of programming language and business logic, that enables automated testing of network links between services is needed. The research shows the successful implementation of such a tool. The use of Istio service mesh to monitor communication between services and build automated test systems is depicted. Additionally, Locust is being utilized to stress test microservices artificially. Lastly, to correct faults found by Jaeger and Grafana dashboards already integrated into Istio, the study suggests establishing temporary connections between affected microservices to address found issues.

It is important to say how this research improves the microservices approach even further, by enabling more effective and independent network troubleshooting and performance measurements.

2.2 Comparative Analysis of Monolith, Microservice API Gateway and Microservice Federated Gateway

Another great research that is worth mentioning was done by Adrio, Tanzil, Lianto and Erlisa (2023) - Comparative Analysis of Monolith, Microservice API Gateway and Microservice Federated Gateway on Web-based application using GraphQL API. The paper shows and explains the test results made by comparing how well one application made with 3 different approaches – Monolithic, Microservice with standard API Gateway, and Microservices with Federated Gateway, can handle user requests.

The Federated Gateway approach is another way of addressing Microservices Architecture's downsides. The base idea is to combine several services into a cluster before they send data to the gateway, reducing complexity at the Gateway level.

The study concludes that Monolithic, while providing significant advantages in speed and number of requests, lacks the scalability which Microservices can easily provide. Another conclusion comes as Federated Gateway is often a safer choice that prevents bottlenecks from huge usual API Gateway, at the cost of performance.

2.3 Microservice Architecture Reconstruction and Visualization Techniques

Microservice Architecture Reconstruction and Visualization Techniques: A Review is also a good study to refer to. Cerny, Abdelfattah, Bushong, Maruf and Taibi (2022) explore the importance of application structure visualization.

The research explains how crucial it is to be able to reconstruct old applications, to better understand their structure, and already degraded or wrongly implemented parts. Through reconstruction, developers can understand necessary implementations to meet new requirements or eliminate existing software. The paper also covers methods that can be used to reconstruct an application and another important reason to even do that.

Microservices systems very often consist of hundreds and thousands of services, which leads to a hard understanding of an application and struggles with its visualization. Thoroughly dividing the whole application into smaller, more understandable parts can help with system visualization. Researchers put forward an idea that the next step in the Microservices Architecture approach evolution might be the development of proper tools and practices for application structure visualization, which might be a 3D software city, interactive solar-system-like approach, or some other method, proved its worth.

2.4 Microservices: Migration of a Mission-Critical System

The research conducted by Mazzara, Dragoni, Bucchiarone, Gieretta, and Dustdar (2021) presents the case study of Danske Bank, Denmark's largest bank, which migrated its Mission-Critical System from Monolithic to Microservices architecture.

The study refers to Monolithic architecture as an easy-to-understand and fast-to-deliver approach, but as soon as the business logic goes beyond some point in size, it becomes drastically slow and hard to implement new features. This is when Microservices architecture comes into play. Danske Bank's utilization of automation, clustering, load balancing, service discovery, containerization, and orchestration, which all will be explained in more detail further in the thesis, resulted in a significantly enhanced system's efficiency and agility. The adoption of these technologies allowed Danske Bank to break down its monolithic architecture into smaller, manageable services that could be tested and developed independently.

The paper emphasizes the importance of a careful and thoughtful adoption process. It suggests such techniques as incremental approach, utilization of agile methodologies, proper DevOps integration and prioritization of migration. In Danske Bank, developers focused on implementing one business functionality at a time, precisely stating their purpose and role in a system. Therefore, it is worth paying attention to this approach, after looking at their huge success.

2.5 Efficient Resources Utilization by Different Microservices Deployment Models

Buzato, Goldman, Batista (2018) explained the performance differences of various microservices deployment models in their Efficient Resources Utilization by Different Microservices Deployment Models research. Mainly the difference between the two models was discussed – one involving a single container for both application and data layers, and the other using separate containers for these layers.

The study found that significant reductions in network consumption might be observed, when deploying each microservice in a single container, up to 99%. This approach can lead to significant optimization of resources and, on the other hand, can also lead to increased coupling between layers, affecting microservice availability. However, the separation of application and data layers can lead to a more maintainable and resilient system.

Multiple experiments and a variety of approaches show a huge interest in Microservices Architecture. The future and availability of resources for development is a matter of concern nowadays, which affects thousands of people all working toward the same goal – to make the application development process faster, easier, and more efficient.

3 EVOLUTION AND CASE STUDIES: FROM MONOLITHIC TO MICROSERVICES ARCHITECTURE

3.1 Historical Evolution from Monolithic to Microservices Architecture

The Era of Monolithic Architecture

In the initial stages of software development, applications were primarily built as monolithic structures. In this model, all components of the application - the user interface (UI), database operations, application configuration, and data access layer - are tightly integrated and run as a single process. Newman (2019, 12-16) gave a great explanation of the monolithic approach. The example structure is presented in Figure 1. This design approach was favoured in the early days of software development due to its straightforwardness and the technological constraints of that time, which made it easier and more cost-effective to build, deploy, and manage applications as a single entity.

One of the key drawbacks of this architecture, as applications began to grow, was scalability. It became challenging to scale a specific function or service independently, as doing so would require scaling the entire application, which is resource-intensive and inefficient. Another drawback was the difficulty in implementing updates or new features; any change, no matter how small, required redeploying the entire application, leading to potential downtime and disruption of service. This rigid structure also made it difficult to adopt new technologies or frameworks, as it would often require a complete overhaul of the application.

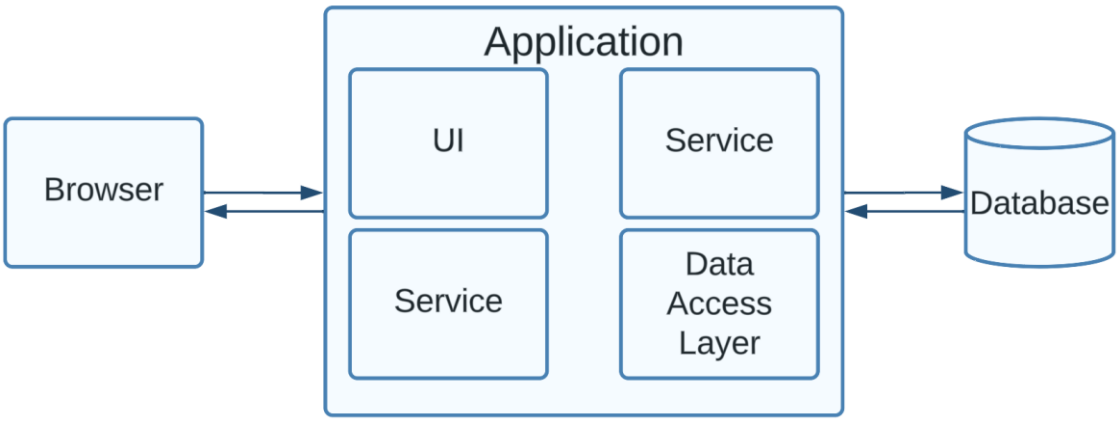

Figure 1. Monolithic Architecture structure example

Transition to Service-Oriented Architecture (SOA)

To address the limitations of monolithic applications, the concept of Service-Oriented Architecture appeared. SOA divided applications into distinct services with specific functions, connected through communication protocols. This provided a significant step forward in how software could be designed and managed. SOA allowed services to be developed and deployed independently, often resulting in improved maintainability due to the isolation of services. This approach meant that individual teams could work on different services simultaneously, resulting in speeding up development cycles and making the system more agile.

However, SOA came with its complexities. The Enterprise Service Bus (ESB) was introduced as a communication tool, that provided communication support between services. Taking from Endrei (2004, 38-40), the ESB can facilitate, for example, the interaction between a mobile client service and the application service without requiring a direct connection between them. An example is presented in Figure 2. It handles the complexities of transforming data formats, ensuring that the data sent from one service can be understood by another. The ESB also deals with different communication protocols, ensuring that services can communicate over the network. Security services within an ESB are crucial and can include authentication, authorization, encryption, and decryption.

As the number of services grew, the ESB could become overwhelmed, transforming from a coordinator to a bottleneck. The fact that ESB was at the center of everything meant that it had to process a tremendous amount of data and manage complex interactions, which could slow down the overall system performance and increase response times. Moreover, the ESB itself became a critical point of failure - if it went down, the entire system could potentially freeze in one place.

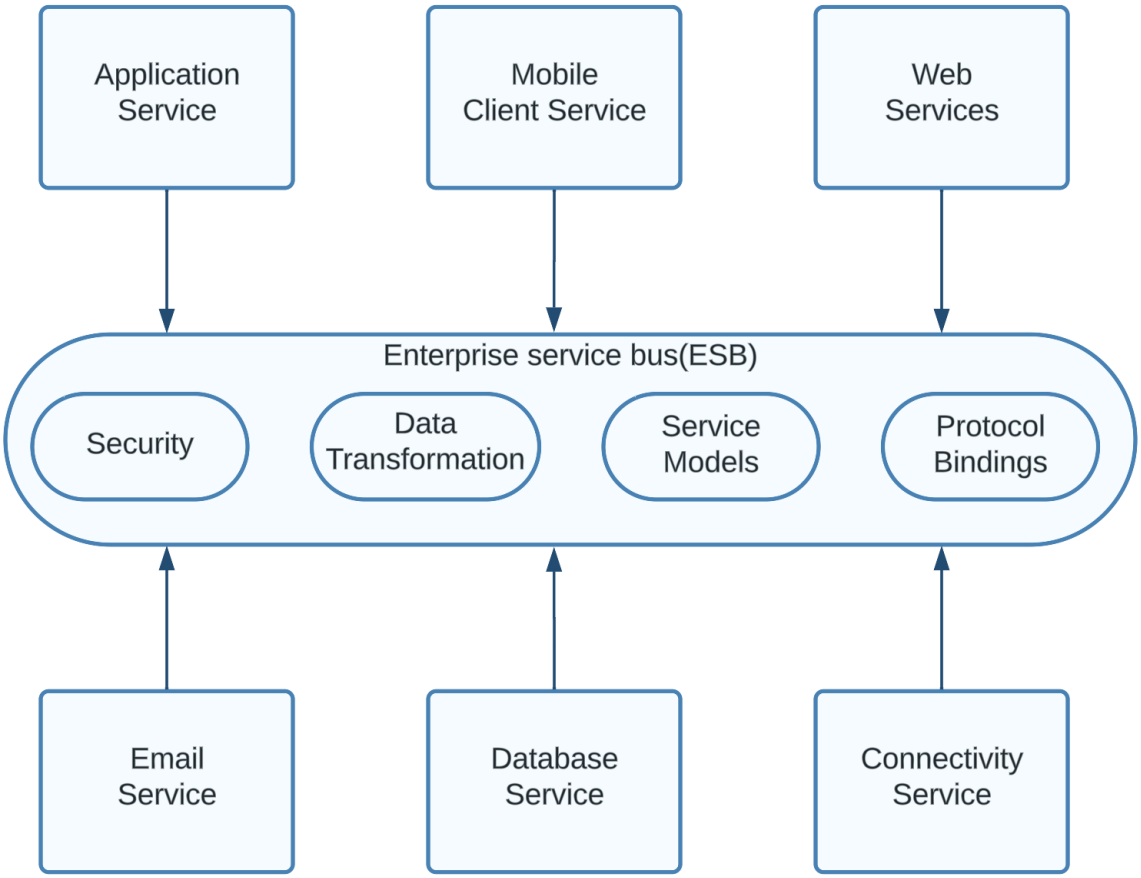

Figure 2. Service-Oriented Architecture structure example

Emergence and Adoption of Microservices Architecture

- Birth of Microservices: Microservices architecture took the principles of SOA further by enhancing services structure and working on more lightweight communication protocols (Richards 2016). The evolution of the approaches appeared as a response to the need for more agile and complex architectures, especially for online services and cloud-based applications.

- Early Adopters: Tech giants like Netflix, Amazon, and eBay were among the first to adopt and popularize microservices. For instance, Netflix started its transitioning process from a monolithic to a microservices architecture in 2009. They refactored the monolithic architecture service by service. The decision was made to handle the growing scale and complexity of ever-escalating data flow and the popularity of online services.
  To emphasize how hard and time-consuming this process is, Netflix finalized the transitioning process in 2012. (Hillpot 2023.)
- Netflix's Role: Netflix's successful implementation of microservices, particularly their ability to handle massive scale and rapid deployment cycles, became a model for other organizations. Their move to open source several tools that accompanied the microservices adoption, like NetflixOSS, which included components for service discovery, load balancing, and fault tolerance, helped other organizations to get a glimpse into the load and complexity of the new approach. (Netflix 2016.)

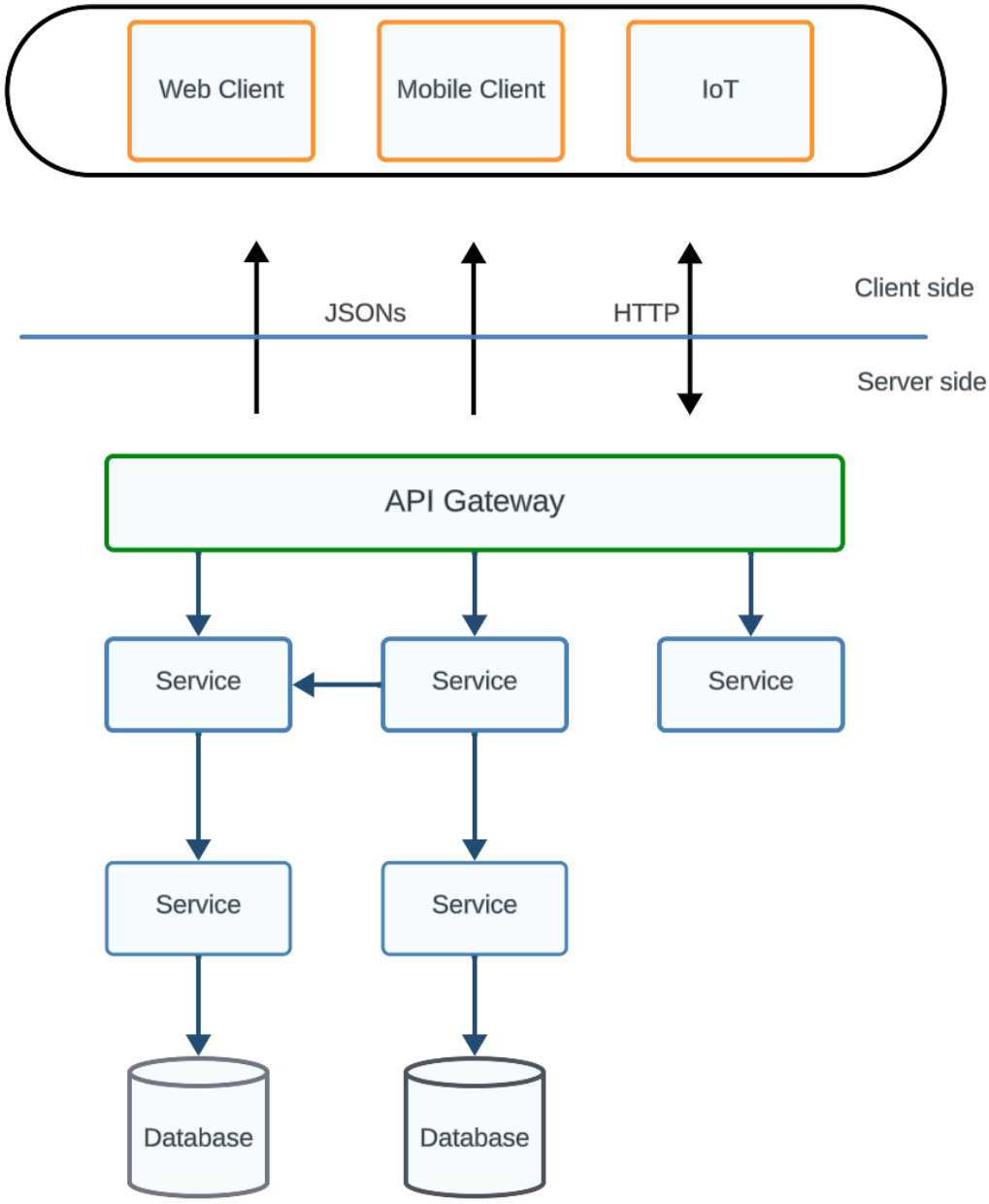

Figure 3. Microservices architecture structure example

Influence of Cloud Computing and Containerization

- Cloud Computing: The rise of cloud technologies provided the necessary tools and services to support the distributed nature of microservices. Cloud platforms like AWS, Azure, and Google Cloud offered services that helped well with the spread and scalable model of microservices.
- Containerization and Orchestration Tools: The rise of container technologies like Docker and orchestration tools like Kubernetes made another step into further developing microservices architecture. These technologies provided a way to efficiently package, deploy, and manage microservices at a high scale (Kirani 2019).
  Containerization can be explained as an approach to encapsulate microservices in self-contained environments, ensuring their consistency throughout the whole development cycle. In other words, with Containerization developers can package each component of the application – its code, system tools, libraries, and settings – within one container. This container can then be easily moved and run on any system that can support it.
  Continuing this, orchestration comes into play. When the number of containers increases, the need for coordination and proper management arises. Tools like Kubernetes can automate the deployment, scaling, and operation of containers across a group of server hosts. It monitors the health of the containers and manages their lifecycle, ensuring that the system is fault-tolerant and that containers can properly communicate with each other.

Current State and Ongoing Evolution

- Widespread Adoption: Today, microservices architecture is widely adopted across various industries, from startups to large enterprises, due to its flexibility, popularity, ease of access and alignment with modern DevOps practices.
- Continuous Evolution: The architecture continues to evolve and to answer new countless challenges, it integrates new technologies and patterns like service mesh architectures, which manage service-to-service communication, and serverless computing, which further abstracts infrastructure management and helps with the utilization of resources (Daya 2015, 38).

3.2 Analysis of Real-World Case Studies: Understanding Practical Implementations and Industry Trends

Two exceptional case studies arise when the topic comes to adopting Microservices Architecture; for their impact and industry transformation: Uber and Spotify. These companies did their best to announce the success in utilizing new technologies, and how they improved their services, making user experience and business operations even more exceptional.

Uber, initially a small-scale application focused on ride-hailing services, rapidly evolved into a global transportation platform offering a variety of services. In its first steps, Uber operated on a monolithic architecture. However, as the company grew, this architecture proved insufficient for handling its rapid expansion and the diverse needs of a global market. Continuing to stick to Monolithic Architecture brought its risks, a single failure in the code base could have brought the whole system down, and it also became too difficult for teams to operate independently and autonomously. To

address these challenges, Uber transitioned to a microservices architecture. This strategic shift involved breaking down the application into distinct microservices, each responsible for specific functions like passenger management, driver management, and billing. The transition to microservices allowed Uber to scale specific aspects of its operations with no dependency on others, providing the agility needed to adapt to the unique requirements of different local markets. Additionally, Uber's engineers could easily define why exactly they needed one service or another. In the end, this architectural shift supported the seamless introduction of new services, such as food delivery, which further diversified Uber's service offerings and strengthened its market presence. However, benefits can't come alone. The company reported that the system became too complex, and it became hard to understand the whole architecture properly. For instance, to find the root of one problem, it was required to work through around 50-60 services and about 15 different teams. (Gluck 2020.)

Now, Uber's main direction is to make features less difficult to build. The system has grown to around 2200 microservices, therefore the reduction in complexity should be happening at the same time as the emergence of new features. One of the ideas on how to achieve that is the unitization of several services into one domain so that API Gateway receives fewer requests and services are categorised under similar logic. Uber already has been able to classify their microservices into 70 domains, which reduced new feature implementation and adoption time by roughly 40 %. (Gluck 2020.)

Spotify, a leader in the music streaming service industry, faced the challenge of managing a vast library of content while handling a rapidly growing user base with diverse preferences. To effectively manage these complexities, Spotify adopted a microservices architecture. This approach, again, involved dividing the application into numerous small, autonomous services. In parallel, Spotify embraced an agile organizational structure characterized by small, cross-functional teams, known as Squads, each responsible for specific microservices. This organizational and architectural transformation enabled Spotify to provide highly personalized user experiences. The microservices approach also allowed for efficient handling of the extensive volume of content and the continuous introduction of innovative features, ensuring that Spotify remained at the forefront of the digital music streaming industry.

Spotify's goal is to provide as seamless experience for users as possible. Therefore, the Hermes communication protocol, which reminded HTTP protocols, was developed from scratch by Spotify's engineering team. It was needed since almost all services were written with Python and Java, so they needed a dedicated and unique communication solution. Microsoft at that time was mostly focused on .Net technologies. A common codebase for web, mobile, and desktop applications is another interesting solution from Spotify, which enabled an even faster and more universal development process. (Varshneya 2022.)

Transitioning to Kubernetes which started in 2017 was the main contributor to Spotify's microservices architecture adoption success. Services were able to handle significantly more requests per second, up to 10 million per second. Moreover, shared information from Spotify's director of engineering talked about 2.5 CPU utilization improvement. (Kubernetes 2019.)

The insights gained from these case studies highlight several key industry trends associated with the adoption of microservices architecture. Firstly, microservices provide the scalability and flexibility required by companies experiencing rapid growth or dealing with large volumes of transactions and data. Secondly, the architecture supports decentralization and independence, allowing for the independent development, deployment, and scaling of different application segments. This independence is crucial for organizations seeking to innovate and adapt quickly to changing market conditions. Thirdly, microservices enhance system resilience. Unlike monolithic architectures, where the failure of one component can impact the entire system, microservices limit the scope of failure, ensuring that the overall system remains robust and reliable. Additionally, microservices architecture allows for even more unique development processes and technology stack utilization, which can easily answer all the companies' needs. Lastly, the adoption of microservices often aligns with DevOps and agile methodologies. This alignment promotes faster development cycles, continuous delivery, and a more responsive approach to software development, corresponding to the dynamic demands of modern business environments.

4 COMPARISON WITH MONOLITHIC ARCHITECTURE

Talking about how Microservices Architecture is gaining more popularity and public interest, we should not make this approach entirely universal. There are still quite numerous amounts of scenarios when Monolithic Architecture may be a better choice for organizations and developers, who want to find profit and usefulness in their digital solutions and needs. Sometimes, the simpler the better.

4.1 Scenarios Favoring Monolithic Architecture

As was already stated, Monolithic Architecture might outperform Microservices Architecture under different scenarios, where the specific case is simple and has a unified nature.

We can favor Monolithic Architecture when talking about:
Small-scale projects, like startup's minimum viable products, often lean towards a monolithic architecture for its straightforward setup and ease of deployment. Similarly, businesses with limited resources might find the complexity of microservices overwhelming, preferring the cost-effective nature of a monolithic system. Applications requiring tight integration, such as legacy financial systems, benefit from the simpler inter-module communication that monolithic architecture provides. Additionally, testing and debugging are often more manageable in monolithic setups due to the ability to run the entire application end-to-end on a single machine. Deployment and operational overheads are also reduced in monolithic architectures, making them suitable for applications like small business websites or blogs. Moreover, data management is often more straightforward in a monolithic system due to centralized data handling.

When compared to microservices architecture, which excels in scalability, and flexibility, and enables decentralized and independent service management, monolithic architecture offers simplicity and cohesiveness (Newman 2019, 15-16). This makes it an appropriate choice for projects where these qualities align with the project's goals and constraints.

4.2 Real-World Examples Favoring Monolithic Architecture

4.2.1 Basecamp

Basecamp, known for its project management and team communication tools, stands as a compelling example of monolithic architecture's effectiveness. Despite the growing popularity of microservices, Basecamp has consciously chosen to stick with a monolithic setup for its applications. This decision stems from their business philosophy, which puts a premium on simplicity and maintainability. Basecamp's leadership believes that a monolithic architecture aligns perfectly with the size of their team and the nature of their application, which, by their assessment, does not warrant the complexity of microservices. They argue that a single, unified codebase is far more manageable and less prone to the complications that can arise from a distributed system. The result of this approach has been quite positive for Basecamp. They have successfully maintained a robust and efficient application, sidestepping the overhead and intricate coordination often associated with microservices architectures. (Hansson 2017.)

4.2.2 Etsy

Etsy, the global online marketplace for handmade and vintage items, presents another interesting case. Originally built on a monolithic architecture, Etsy faced the dilemma of whether to continue with this model or transition to microservices as it grew. The simplicity and cohesiveness of a single codebase were crucial factors for Etsy. These aspects were particularly important given their commitment to continuous integration and deployment practices. The ease of deploying and managing a single, interconnected system was a significant advantage. Since they already had a great number of experienced developers, their code base was enormously big with tons of documentation. However, while Etsy did explore service-oriented elements to enhance certain aspects of its platform, it tried to maintain a monolithic core. It was truly a challenge for them, so, adopting a combined architecture seemed like the best choice. This approach has proven effective, as evidenced by Etsy's ability to handle high traffic volumes and complex e-commerce operations efficiently. Etsy's experience shows that a well-maintained and thoughtfully structured monolithic architecture with the combination of Microservices Architecture can competently support even large-scale, high-traffic online platforms. (Hillpot 2023.)

4.3 Advantages of Microservices Architecture

The advantages of Microservices Architecture are numerous. Such an approach became popular for a reason. It answers the demands of the market, modern technologies, and business models. After evaluating Richards' (2016, 6-9) and Newman's (2021, 4-8) opinions on Microservices' benefits it is reasonable to conclude with the next cons:

- **Scalability:** Microservices allow for easy scaling of individual components without needing to scale the entire application.
- **Flexibility in Technology Choices:** Teams can choose the best technology or framework for each microservice, with no worry of affecting the whole project.
- **Resilience and Isolation:** Failures in one service have minimal impact on others, enhancing overall system resilience.
- **Continuous Deployment and Delivery:** Enables quicker and more frequent updates and releases without disrupting the entire application.
- **Modularity and Maintenance:** Smaller, well-defined services are easier to understand, develop, and maintain.
- **Improved Fault Isolation:** Reducing the scope of potential problems, since they are happening in individual services, simplifying debugging and recovery.
- **Adaptability to Business Needs:** Supports agile development that aligns with changing business requirements and market conditions.

4.4 Typical Failures in Microservices Implementations

The bigger they are the harder they fall, the same goes for Microservices Architecture. There are countless ways to break it. The digital world is truly detail-hungry, even the smallest error can lead to big, unresolvable issues. Concluding from the already pointed out features of Microservices Architecture, we can say that it is often an enormously complex system, that includes hundreds of steps

and actions to make everything work as intended. Therefore, by inspiring from Richardson (2023), it is easy to summarize typical failures in Microservices Architecture:

- **Complexity Overload:** Microservices can introduce complexity in development and operations, leading to potential failures in system integration and management.
- **Inadequate Infrastructure:** Without the right infrastructure, such as container orchestration systems like Kubernetes, managing microservices can become unwieldy, leading to system failures.
- **Poorly Defined Service Boundaries:** If microservices are not adequately separated based on business capabilities, it can result in tangled services, causing maintainability issues and operational inefficiencies.
- **Network Issues and Latency:** The distributed nature of microservices can introduce network latency and communication issues, potentially leading to system failures.
- **Security Vulnerabilities:** Microservices require robust security protocols at each service level. Failures in ensuring security can lead to vulnerabilities and potential breaches.
- **Database Management Challenges:** Implementing a distributed database system that works efficiently with microservices can be challenging and, if not done correctly, can lead to data consistency issues and failures.
- **Difficulty in Monitoring and Logging:** Properly monitoring and logging a distributed system with multiple microservices can be challenging, and failures in this area can lead to unnoticed issues escalating into major problems.
- **Lack of communication and leadership:** Having an appropriate leadership role in a company can be hard. Lack of local human resources, stagnation in the market and simply bad recruitment. This can lead to weak communication practices between teams, which can affect the whole development process. In addition, no delivery metrics, bad team infrastructure, and wrong decisions are all caused by scattered adoption.

However, such reasons can affect every software application out there, but the complexity and size of Microservices architecture might require even more attention and accuracy.

The true downfall of Microservices Architecture that might trigger the next big software development model shift is the exponentially growing complexities in IT. For instance, it is reported that a web mobile transaction is going through 35 different services, compared to 22 just five years ago. Another good example can be found in the survey conducted by Dynatrace, which shows that IT teams of 800 different companies around the globe spend about 29% of their time investigating and managing software performance problems. (Mass 2018.)

It is becoming impossible to monitor the microservices' performance in real-time, applications are becoming so populated with containers that it is too difficult to notice each service's individual impact. In simple words, the monstrosity of digital products is leading to the evolution of a new approach, that can embark in the near future, to become a new successor of Microservices Architecture.

5 SUITABILITY AND DEVELOPMENT CHALLENGES

Every technical question is a puzzle to solve. It is practically impossible to avoid challenges during any development process. Building an application using Microservices Architecture might lead to overly complex solutions that will require additional efforts from organizations and developers to solve all the challenges along the way. We will familiarize ourselves with possible issues that might appear and think about how to overcome them.

5.1 Challenges and Solutions

Data Synchronization and Consistency

Ensuring data consistency across multiple, independently managed microservices is a major challenge. In addition, keeping data synchronized across different services and databases can be complex.
Therefore, utilizing event-sourcing architecture and the SAGA design pattern might be effective in addressing these challenges. These methods rely on asynchronous cross-service communication to maintain data consistency. SAGA is a series of local transactions, that makes sure if one transaction fails due to an error or business fault, compensating transactions are executed to undo the changes made (Richardson 2023.)

Security

Microservices increase the need for more complex security instruments due to multiple points of communication among services and the broad structure of the architecture itself. Most of the time communication happens with API calls, therefore, it is good to assume that those are the riskiest components of the structure. We can think about:
Implementing an API Gateway to centralize security measures, using tools like Spring Cloud Gateway, Apigee etc. Then integrate JWT tokens to custom our security solution and enhance API defense. Moving even further with two-step authentication, creating a dedicated key vault, and utilizing strong cloud solutions like AWS or Azure.

Services Communication:

Microservices architecture, in itself, means multiple small services combined to work together in one big application. Thus, it is crucial for services to be able to dynamically discover each other. The foundation of the architecture is interactions between services, moreover, seamless, and reliable communication is what everyone should aim for when considering using Microservices Architecture, which might become a real challenge.

DevOps Support

Deploying and supporting microservices can lead to complexities in CI/CD and DevOps processes. So, introducing mature DevOps practices becomes a necessity. Microservices need robust continuous integration and delivery pipelines, so such practices can help automate build, test, and deployment processes (Daya 2015, 40). Additionally, Kubernetes-like solutions are exceptional when it comes to managing deployment processes.

5.2 Suitability of Microservices Application

Deciding on the business model is hard, as well as deciding on which approach to use in digital product development. Answering the questions in this table could help organizations and developers decide on whether Microservices Architecture is the right choice for them. By assessing each factor, it is possible to quantify how well microservices architecture aligns with the project's specific needs and characteristics.

Scoring explanation:

- **0-80 Points:** Microservices may not be the best fit. Consider simpler architecture.
- **81-150 Points:** Microservices could be suitable, but further analysis is needed. Consider hybrid approaches.
- **151-200 Points:** Microservices are likely a very good fit for the project requirements.

Table 1. Criteria to find suitable architecture.

Criteria	Description	Points
Complexity of Application	Does your application include several distinct and complex functionalities that would benefit from being broken down into smaller, manageable services?	10
Scalability Needs	Do you anticipate the need to independently scale different parts of your application based on varying demand or usage patterns?	10
Rapid Market Adaptation	Is your business environment dynamic, requiring the ability to quickly adapt and implement changes in response to market demands?	10
Frequent Updates	Does your application need to be updated often, with the ability to deploy these updates independently without affecting the entire system?	10
Diverse Tech Stack	Would different parts of your application benefit from using different technology stacks, languages, or frameworks?	10
DevOps Maturity	Does your organization have established DevOps practices, including continuous integration and delivery?	10
Autonomous Teams	Can your development team be divided into small, independent groups, each responsible for different services?	10
Resilience and Fault Isolation	Is minimizing the impact of a service failure on the entire application a priority, thus requiring resilient and isolated services?	10
Decentralised Data Management	Do different components of your application have unique data management and storage requirements?	10
Long-term Strategic Fit	Does adopting microservices align with your organization’s long-term goals and strategies for growth and development?	10
Cloud Infrastructure Readiness	Do you have access to cloud infrastructure capable of supporting the distributed nature of microservices, such as container orchestration and service discovery tools?	10

Organizational Agility	Is your organization agile and flexible enough to adapt to the changes in processes and structure required by microservices architecture?	10
Resource Availability	Do you have or can you acquire the talent skilled in microservices architecture, containerization technologies, and cloud computing?	10
Budget for Infrastructure	Are you prepared to invest in the necessary infrastructure, including cloud services and monitoring tools, to support a microservices architecture?	10
Monitoring and Logging Requirements	Does your application require advanced monitoring and logging to track the behaviour and performance of distributed services?	10
Security Management	Do you have the capacity to handle complex security requirements, including securing multiple service endpoints and inter-service communications?	10
Testing Capabilities	Do you have the resources and tools required for comprehensive testing in a distributed system, including unit, integration, and end-to-end testing?	10
Service Discovery and Load Balancing	Is there a need for dynamic service discovery and effective load balancing due to the distributed nature of your application?	10
Data Consistency Requirements	Are high levels of data consistency across different services crucial for your application's functionality?	10
Inter-service Communication Complexity	Does your application necessitate complex communication patterns between services, such as synchronous calls or event-driven communication?	10

In the end, everyone should understand that Microservices Architecture is just a tool, not a panacea. It is easy to use it wrong. Inadequate documentation, lack of automation solutions, overly complex and difficult-to-understand code, reliance on manual testing and deployment or simply, no communication between teams. Supporting Newman's (2021, 11) thoughts, it is logical that Microservices Architecture demands fast-paced and mostly flawless development, with an agile approach and proactive organization, otherwise, adopting such a model can lead to even more problems than before.

6 ROADMAP FOR ADOPTING MICROSERVICES ARCHITECTURE

Transitioning to a microservices architecture is associated with a significant shift in developers' mindset, not just in technology. This transition, while offering numerous benefits like improved scalability, flexibility, and faster deployment cycles, also accompanies businesses with unique challenges and complexities.

The next roadmap is meticulously crafted, from my personal experience and generally accepted approaches, which have already been described many times by Newman (2021) and Richardson (2023) and summarized by Daya (2015) and Pachghare (2017), to guide organizations and developers through the process of adopting a microservices architecture, specifically focusing on systems built with C# and .NET Core, since this is one of the most popular systems for development and tools that I have a personal experience with. In addition, approaching a roadmap from one direction opens for more freedom and attention to detail. It is structured into distinct phases, each addressing critical aspects of the transition - from initial assessment and planning to future-proofing the architecture. Each phase provides a clear and sequential path for a successful transition. The roadmap is not just a series of steps but a journey, that fosters the importance of considering each phase in relation to the others.

Also, it is important to consider this roadmap as an adaptable tool and not as direct instructions. The users should recognize their specific needs and the context of the project. All in all, the goal is to provide a structured yet flexible framework that can be adjusted as the developers or organizations progress through their Microservices Architecture adoption.

Comparing this roadmap to other, already scientifically proven ones, is hard. The IT world is truly a fast-paced place, new tools are being developed every day, while others become outdated. More and more people are being exposed to the Internet, how it works and what it can offer, therefore, more and more businesses are trying to find ways to lure more clients. This, in its place, provokes more interest in digital products, which end up on the shoulders of developers and engineers, being asked to invent new services and solutions. From this short discussion, we can conclude that without proper updates, content and technologies become disregarded fast, and the same goes for similar roadmaps. There is almost no similar modern research available currently. The only fresh one from Craske (2022), which also acted as a scientifically proven material for creating the roadmap, offers general guidelines for Adopting Microservices Architecture. However, my goal is to deliver a detailed and specific tool.

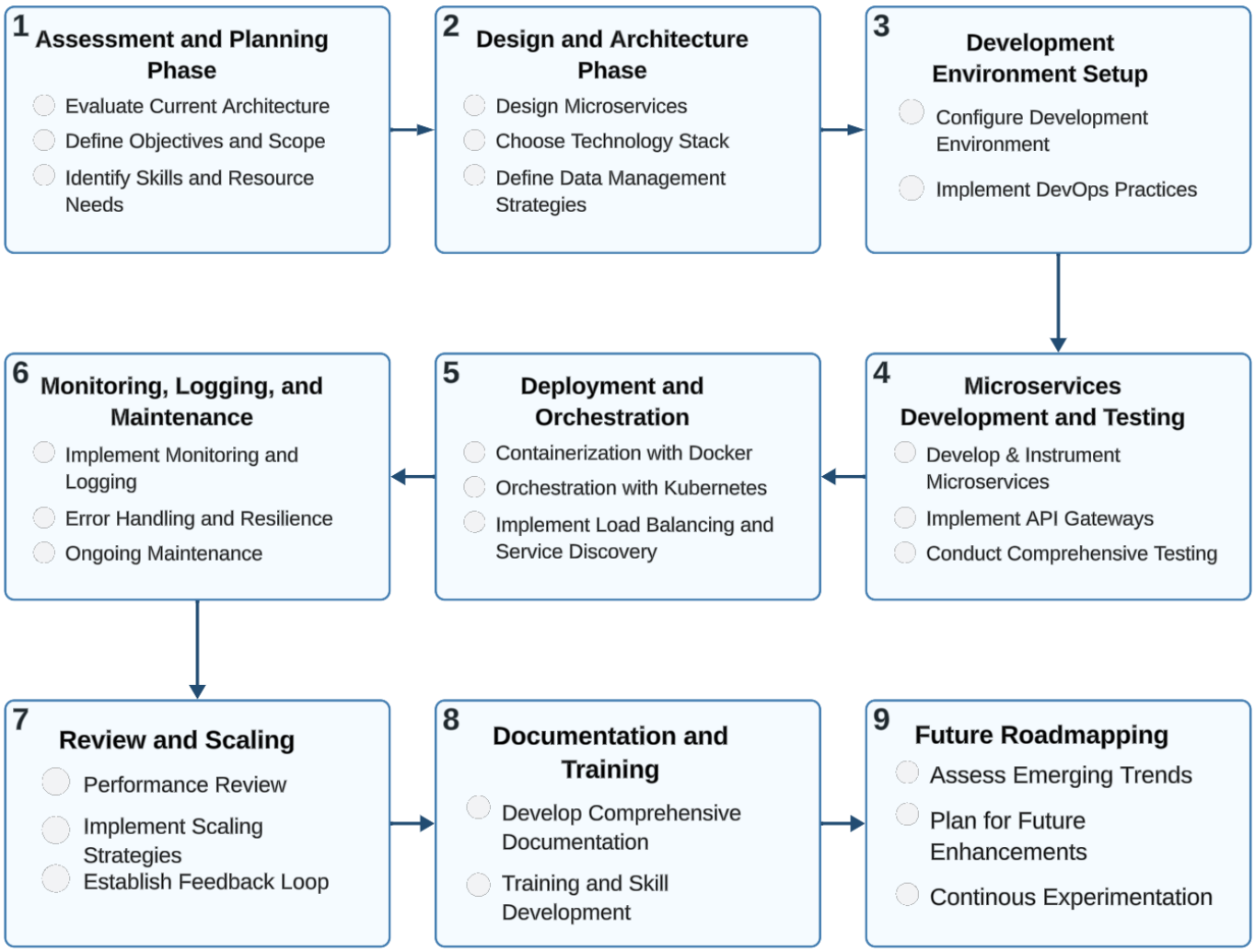

Figure 4. Roadmap for adopting Microservices Architecture

6.1 Assessment and Planning Phase

Evaluate Current Architecture

Begin with a comprehensive analysis of your current architecture, whether it's monolithic or service-oriented. Understand how different components are interconnected, their dependencies, and the overall data flow. Identify components that show high internal connectivity and low dependencies with other components, as these are ideal candidates for transforming into microservices. Focus also on parts of the application that require frequent updates or scaling and those with distinct business functionalities or varying development cycles. Document and map all dependencies in the current system to then help in planning the decomposition process.

Define Objectives and Scope

Set clear objectives for the transition to microservices. If handling increasing loads or user growth is a priority, then your strategy should focus on scalable components. If agility, rapid deployment, and market adaptability are essential, concentrate on services needing frequent updates. Align the transition with specific business goals, like enhancing user experience or expanding services. Decide on the extent of the transition, whether it involves a complete shift to microservices or starts with a hybrid approach, transitioning one component at a time. Consider beginning with a test or pilot project or a less critical system component to evaluate the effectiveness of the transition.

Identify Skills and Resource Needs

Conduct a skill set evaluation of your team about C#, .NET Core, and microservices-related technologies. Identify skill gaps, especially in areas like containerization (Docker), orchestration (Kubernetes), and Azure cloud platforms. Plan for training sessions and development programs for your team to upgrade their skills. This could include workshops, online courses, or consultation with microservices experts or those who have gone through the same process already. Also, assess the need for additional team members or the reallocation of current resources. Determine if new tools or technologies are required for the development, deployment, and monitoring of microservices.

Consider Building Cross-Functional Teams

Assessing your organization's brain power is important, but what might be more important is how it is utilized. In the Microservices Architecture approach, each team is usually responsible for one or more services, including not only the development but also the testing, deployment, and ongoing maintenance of those services. These teams usually consist of members with various backgrounds like software development, database management, UI/UX design, quality assurance, security, and DevOps engineering. Therefore, the cross-functional teams come into play. Bringing together diverse skill sets will ensure each service's effectiveness, health, and reaction to business changes, which is crucial in the Microservices model.

This first phase is crucial as it builds the foundation for a successful transition to microservices architecture. It ensures that the change aligns with your business objectives, technological capabilities, and the skills of your team, setting the stage for detailed planning and subsequent execution.

6.2 Design and Architecture Phase

Design Microservices

Begin by analyzing your business domain to pick distinct functional areas. Each of these areas typically correlates to a microservice. Apply the Domain-Driven Design (DDD) principles, which can be explained, according to Newman (2019, 28), as an approach to designing software that reflects the realities and complexities of the business it is meant to serve. Proceed to define a microservice for each functional area, considering its business logic, data, and relevant functionalities. It is crucial to ensure that each microservice is self-efficient and keeps minimal dependency on others, as this autonomy is key to fully exploring the full benefits of a microservices architecture. Develop a detailed service blueprint for each microservice, outlining its responsibilities, interfaces, and interactions with other services, keeping the service boundaries clear and straight.

Choose the Tech Stack

Focus on C# and .NET Core, evaluating the latest versions and features suitable for your microservices. Consider aspects like performance, security, and compatibility. The cross-platform nature of .NET Core and its support for containerization make it an ideal choice for microservices development. In terms of containerization, Docker is the preferred choice for encapsulating your microservices, providing isolated environments for each service and streamlining deployment and scaling. For orchestrating these containerized microservices, Kubernetes or Docker Swarm are suitable, with

Kubernetes offering advanced features like auto-scaling, self-healing, and load balancing. Additionally, selecting an API Gateway such as Ocelot (for .NET applications) is crucial, as it manages requests and routes them to the appropriate microservices. Also, consider integrating tools for CI/CD like Jenkins or Azure DevOps, version control systems such as Git, and configuration management tools like Consul or Azure App Configuration.

Define Data Management Strategies

Plan for each microservice to have its database to ensure minimal dependency, allowing services to evolve independently and maintain data consistency. Evaluate and choose database technologies that align with the service's requirements, whether it's SQL, CosmosDB or Identity Management tools. In scenarios where shared databases are necessary, establish protocols to manage database access and prevent dependencies. Ensure that database schema changes do not adversely impact other services using the same database. Design the data access layers in your microservices, considering aspects such as caching and data replication. For integrating data across services, use asynchronous communication methods like message queues to minimize direct dependencies between services. If migrating from a monolithic architecture, formulate a comprehensive plan for transferring existing data to the new microservices databases, which may include data transformation and cleansing.

The second phase involves making crucial decisions regarding the architecture, technology stack, and data management strategies, setting the ground for a robust, scalable, and maintainable microservices ecosystem.

6.3 Development Environment Setup

Configure Development Environment

Choose an Integrated Development Environment (IDE) that effectively supports C# and .NET Core development. Visual Studio and Visual Studio Code are popular choices, offering extensive support for .NET Core, debugging tools, and integration with version control systems. Ensure that all necessary plugins and extensions for .NET Core development are installed and configured in the IDE.

Set up code repositories for source control using Git, which integrates well with various CI/CD tools and project management platforms. Consider platforms like GitHub, Bitbucket, or Azure Repos for hosting your repositories and ensuring the proper branch management.

Choose Continuous Integration/Continuous Deployment tools that best fit your workflow. Options include Jenkins, Azure DevOps, and GitHub Actions. Jenkins offers flexibility and a wide range of plugins. Azure DevOps provides an integrated set of services from code repositories to build and release pipelines. GitHub Actions offers native integration with GitHub repositories for CI/CD.

For the local development environment, set up Docker on developers' machines to allow for local testing of microservices in a containerized environment. If using Kubernetes, consider setting up Minikube or Docker Desktop's Kubernetes for local orchestration testing.

Implement DevOps Practices

Establish Continuous Integration pipelines to automate the process of code integration, including steps for code compilation, unit tests, and static code analysis. Utilize tools like SonarQube for code quality analysis and ensure adherence to defined code quality thresholds.

Set up Continuous Deployment pipelines for the automated deployment of microservices, which should include steps for containerization and deployment to staging and production environments.

Adopt a version control strategy that supports your development workflow, like GitFlow or Trunk-Based Development. Ensure that branching strategies are aligned with CI/CD practices for smooth integration and deployment processes.

Integrate monitoring tools into your DevOps pipeline to provide insights into application performance and usage. Establish feedback loops within the DevOps cycle to continuously improve development and deployment practices based on real-time feedback and performance metrics.

This setup phase is vital for ensuring a productive and efficient development process for microservices architecture. Proper configuration of development environments and the implementation of robust DevOps practices lay the foundation for successful development, testing, and deployment of microservices.

6.4 Microservices Development and Testing

Develop Microservices

Begin by incrementally developing one microservice at a time, focusing on those that are relatively isolated and offer clear business value. Apply Domain-Driven Design principles to model each microservice around a specific business domain. Utilize the latest features of .NET Core and C# for optimal performance, security, and maintainability, and follow SOLID principles for object-oriented design to create code that is maintainable, flexible, and testable. Employ asynchronous programming techniques like async/await to enhance the scalability and responsiveness of the services. Implement critical microservices design patterns like API Gateway, Circuit Breaker, CQRS, and Event Sourcing as needed for each service. Use Dependency Injection, as supported by .NET Core, to manage dependencies effectively and promote loose coupling.

Implement API Gateways

Choose an appropriate API Gateway that integrates well with .NET Core, such as Ocelot or Azure API Management. The API Gateway should be configured to handle routing, load balancing, authentication, and authorization for the microservices. Design the API Gateway to accumulate responses from multiple microservices when necessary, forming a proper composition of responses for the client. Ensure the API Gateway can efficiently handle request forwarding, including the transformation of requests and responses as required.

Testing

Write independent unit tests for each microservice, using .NET Core's built-in testing framework or alternatives like NUnit and xUnit.net. Utilize tools like Moq or NSubstitute to mock external dependencies and isolate the service being tested. Perform integration tests to ensure correct interaction between microservices and databases. Use tools like Postman or HttpClient for testing API integrations. Execute end-to-end tests to validate the entire application flow, automating these tests using tools like Selenium to mimic real user scenarios. Implement contract testing using tools like Pact to verify that communication between services adheres to predefined contracts. Utilize performance and load testing tools like JMeter or k6 to simulate various load and performance scenarios, assessing how the microservices behave under stress.

This phase of microservices development and testing is vital in ensuring the reliability and functionality of each service. A systematic and scrupulous approach to development, coupled with comprehensive testing, forms the strong frame for a robust microservices architecture.

6.5 Deployment and Orchestration

Containerization with Docker

Containerizing each microservice in its own Docker container is essential. Docker provides an isolated environment, ensuring consistency across different deployment environments. Create Dockerfiles for each microservice that specify the base images, environment setup, and commands required to run the services. Build Docker images for the microservices and store them in a Docker registry, such as Docker Hub, Azure Container Registry, or Amazon Elastic Container Registry. Implement version control for Docker images to efficiently track and roll back to specific versions of the services if necessary. Utilize Docker Compose for local development and testing, allowing you to define and run multi-container Docker applications and simulate a microservices environment on developers' machines.

Orchestration with Kubernetes

Setting up a Kubernetes environment suitable for your deployment needs is crucial. Options include managed Kubernetes services like Azure Kubernetes Service (AKS), Amazon Elastic Kubernetes Service (EKS), or Google Kubernetes Engine (GKE). Set up Kubernetes clusters, defining nodes where the microservices will be deployed. Create Kubernetes services for each deployment, enabling communication between different microservices and exposing them to the outside world when necessary.

Configuration and Secrets Management

Also, use Kubernetes ConfigMaps and Secrets or Azure Key-Vaults for managing configuration settings and sensitive information, ensuring they are not hard-coded in microservices.

Load Balancing and Service Discovery

Implement load balancing using Kubernetes' built-in load balancer or integrate with cloud-provided load balancers to efficiently distribute incoming traffic among microservice instances. Ensure that

the load-balancing strategy aligns with the microservices' scalability and performance requirements. Utilize Kubernetes for service discovery, enabling microservices to dynamically discover and communicate with each other. Kubernetes services and DNS play key roles in enabling microservices to locate and interact with each other seamlessly.

This deployment and orchestration phase is vital for the smooth functioning of a microservices architecture. Proper containerization, orchestration, load balancing, and service discovery are critical in ensuring the microservices are reliable, scalable, and maintain high performance under different operational conditions.

6.6 Monitoring, Logging, and Maintenance

Implement Monitoring and Logging

For monitoring setup, implement Prometheus to capture time-series data from your microservices. Configure it to scrape metrics exposed by each service. Utilize Grafana for visualizing data collected by Prometheus, creating dashboards that provide insights into your microservices' performance and health.

Also, configure your microservices to expose relevant metrics like response times and error rates, and log important events. Ensure logs are structured and contain sufficient context for effective analysis.

Error Handling and Resilience

Implement the circuit breaker pattern using libraries like Polly in .NET Core applications to prevent cascading failures. Configure circuit breakers to take into action in case of repeated failures and reopen after a pre-defined timeout, with the option of a fallback mechanism. Use Retry patterns to handle transient failures in communication between microservices, defining policies for the number of retries and intervals between them. Design fallback mechanisms to ensure the system operates effectively in case of partial failures.

Ongoing Maintenance

Plan regular updates to microservices, including dependency updates, bug fixes, and feature enhancements. Continuously monitor performance data to identify optimization opportunities. Regularly refactor microservices to improve code quality and reduce technical inconsistency, keeping the technology stack updated and aligned with current best practices. Maintain up-to-date documentation for each microservice, covering APIs, configurations, and deployment procedures, and integrate a culture of knowledge sharing within the team to ensure collective ownership and understanding of the microservices architecture.

This phase is essential in ensuring that the microservices architecture is not just functional but also robust, efficient, and maintainable over the long term. Implementing effective monitoring, logging, and maintenance practices provides the necessary insights and tools to keep the system healthy and responsive to evolving business needs.

6.7 Review and Scaling

Performance Review

Regularly schedule performance assessments of the microservices architecture. These reviews should include assessing response times, resource usage, error rates, and user satisfaction. Use monitoring tools like Prometheus and Grafana, set up in the previous phase, to gather and analyze performance data. During these reviews, identify any bottlenecks or performance issues in the architecture. Analyze logs and metrics to determine the root causes of performance degradation or failures. Establish performance benchmarks based on initial metrics and industry standards and compare current performance against these benchmarks to understand improvements or regressions.

Scaling

Implement scaling mechanisms that respond to real-time demand. Use the auto-scaling features of Kubernetes or similar orchestration tools to scale services up or down based on usage. Scale individual microservices independently to ensure efficient and cost-effective resource allocation. Use performance metrics as triggers for scaling decisions. For instance, increase the number of instances of service if response times begin to degrade beyond a certain threshold. Continuously refine scaling policies based on historical performance data and predictive analysis.

Feedback Loop

Establish a continuous improvement process that incorporates insights from performance reviews into development and operational processes. Encourage regular communication between development, operations, and business teams to align architectural improvements with business goals. Collect feedback from end-users and stakeholders to understand how changes in the architecture impact user experience and business outcomes. Integrate this feedback into the development cycle to ensure that the microservices architecture remains aligned with user needs and expectations. Plan for iterative enhancements to the architecture, including not only scaling and performance optimizations but also feature updates and technical debt reduction. Foster a culture of continuous learning and adaptation, encouraging teams to experiment and innovate.

This phase is essential in ensuring the microservices architecture remains efficient, scalable, and aligned with both current and future business needs. Regular performance reviews, responsive scaling strategies, and a robust feedback loop are key to maintaining the vitality and relevance of the architecture over time.

6.8 Documentation and Training

Documentation

Documentation development involves detailing the design and architecture of each microservice, including its business logic, APIs, and interactions with other services. It's important to maintain high code documentation standards, ensuring that inline comments and API documentation are always available and up-to-date.

Deployment documentation includes creating comprehensive guides on deploying and configuring each microservice in different environments like development, staging, and production. This also covers documenting the CI/CD pipeline process and outlining steps for building, testing, and deploying microservices.

Maintenance and operational documentation should also be present, consisting of operational manuals with procedures for monitoring, scaling, and troubleshooting microservices. It includes guidelines for managing common issues, failure scenarios, and disaster recovery processes.

Best practices and standards documentation captures the best practices for C# and .NET Core and Object-Oriented Programming development. It documents coding standards, review processes, and architectural guidelines.

Training

Onboarding and skill development involve developing an onboarding program for new team members, focusing on the specific technologies and architectures used in the microservices system. This includes providing skill development opportunities in areas like .NET Core, Docker, Kubernetes, DevOps, Microservices Architecture, version control, Database operations, and cloud platforms.

Workshops and collaborative learning inspire team members to share insights, discuss challenges, and explore new solutions. It encourages participation in external webinars, conferences, and courses relevant to microservices development, current organizational needs, and management.

Hands-on training sets up training environments where developers can experiment with new technologies and architectures without affecting production systems. This approach includes implementing pair programming or mentorship programs to support knowledge transfer and collaborative problem-solving.

A continuous learning culture fosters an environment of ongoing learning and improvement. This encourages team members to stay updated with the latest trends and advancements in the information technology world. Providing access to online resources, technical literature subscriptions, and supporting professional development is also crucial and creates a positive work environment.

This phase ensures the team is well-equipped and informed to effectively develop and maintain the microservices architecture. Comprehensive documentation acts as a valuable reference, while ongoing training and skill development keep the team's expertise high and aligned with technological trends and best practices.

6.9 Future Roadmapping

Assess Emerging Trends

Regularly monitor industry developments by following news, publications, and thought leaders to stay updated on trends in microservices, .NET Core, and cloud technologies. Engage in forums, online communities, and professional networks to discuss new ideas and practices. Assess emerging tools, frameworks, and methodologies in the microservices field, and keep an eye on advancements in .NET Core, exploring new features and capabilities from Microsoft. Encourage team members to

attend relevant conferences, workshops, and seminars to gain insights into new trends and technologies and use these learnings to inform decisions about technology adoption and architectural changes.

Plan for Future Enhancements

Schedule regular development process reviews to evaluate the current state of the system and identify areas for improvement, involving cross-functional teams for diverse perspectives. Collect and analyze feedback from developers, operations teams, and end-users to pinpoint blank spots and areas needing enhancement, using this feedback to guide future planning and prioritize valuable updates. Develop a roadmap for upgrading technologies and tools. Align the microservices architecture with evolving business strategies and objectives, staying agile and adaptable to evolve the architecture in response to new business opportunities and market demands.

Lastly, keep your microservices architecture cutting-edge, efficient, and aligned with both technological advancements and business objectives. This proactive approach positions your organization to capitalize on new opportunities and effectively respond to the dynamic world of software development.

7 CONCLUSION

7.1 Reflection on Goals

This thesis presents a comprehensive exploration of microservices architecture, particularly emphasizing its practical implementations, benefits, and nuances. The journey of this research began with understanding the historical evolution from monolithic to microservices architectures, highlighting the shift driven by the need for scalability, agility, and adaptability in software development.

The core objective of this thesis was to create a detailed roadmap for organizations and developers considering transitioning to microservices, specifically in the context of C# and .NET Core frameworks. This roadmap, crafted through extensive research, own experience, and analysis, includes various phases: assessment and planning, design, development environment setup, microservices development and testing, deployment and orchestration, monitoring, logging, maintenance, review and scaling, documentation and training, and future road mapping. Each phase was concluded to provide practical guidelines, best practices, real-world technologies and insights into the complexities and challenges one might encounter during the transition.

7.2 Self-Evaluation

The thesis was grounded on several reliable research papers about Microservices Architecture, real-world case studies, reported by companies themselves or creditable informational recourses, and developers' and organizations' discussions about adopting and utilizing new approaches in architectural design. The analysis was done objectively, focusing on providing a balanced view of microservices architecture, including its advantages and potential pitfalls.

This research journey has led to significant professional growth. The full accomplishment of the plan involved not only acquiring in-depth technical knowledge about microservices but also developing critical thinking skills to evaluate various architectural scenarios. This would definitely come in handy in the field, acting as another support for future professional development.

The comprehensive guide that is not just theoretical but practically applicable in real-world scenarios, can be counted as a success of the thesis. It addresses the needs of its target audience – developers and organizations looking to transition to microservices. However, the rapidly evolving nature of technology might require continuous updating and expansion of the content to keep the research relevant.

In conclusion, this thesis serves as a valuable resource for those who want to step into the realm of microservices architecture. The insights and guidelines provided here aim to equip developers and organizations with the knowledge to make thorough decisions, implement effective strategies, and embrace continuous learning and adaptation in the ever-evolving world of software development.

Artificial intelligence has been used in the work as follows:

ChatGPT 2023. OpenAI. GPT-4. Accessed for language check, November 2023. https://chat.openai.com

REFERENCES

Buzato, Fernando, Goldman, Alfredo and Batista, Daniel 2018. Efficient Resources Utilization by Different Microservices Deployment Models. IEEE.

Cerny, Tomas, Abdelfattah, Amr, Bushong, Vincent, Maruf, Abdullah Al, Taibi, Davide 2022. Microservice Architecture Reconstruction and Visualization Techniques: A Review. IEEE.

Craske, Antoine 2022. The Microservices Adoption Roadmap. Internet publication. QE Unit. Updated August 23, 2022. https://qeunit.com/blog/the-microservices-adoption-roadmap/. Accessed 11.2023.

Daya, Shahir 2015. Microservices from Theory to Practice. IBM: Redbooks.

Endrei, Mark 2004. Patterns: Service-Oriented Architecture and Web Services. IBM: Redbooks.

Gluck, Adam. 2020. Introducing Domain-Oriented Microservice Architecture. Internet publication. Uber Engineering. Updated July 23, 2020. https://www.uber.com/en-FI/blog/microservice-architecture. Accessed 11.2023.

Hansson, David Heinemeier. 2017. The Majestic Monolith. Internet publication. Signal V. Noise. Updated February 29, 2017. https://m.signalvnoise.com/the-majestic-monolith/. Accessed 11.2023.

Hillpot, Jeremy. 2023. Microservices Examples. Internet publication. DreamFactory Web site. Updated May 25, 2023. https://blog.dreamfactory.com/microservices-examples. Accessed 11.2023.

Kanth, Rajeev, Rupesh, Raj Karn, Rammi, Das, Dibakar, Raj Pant, Jukka, Heikkonen 2022. Automated Testing and Resilience of Microservice's Network-link using Istio Service Mesh. IEEE.

Kirani, Darshan 2022. Microservices Orchestration Using Azure Kubernetes Service. Apexon. Internet publication. Apexon. Updated May 19, 2022. https://www.apexon.com/blog/microservices-orchestration-using-azure-kubernetes-service-aks-2. Accessed 11.2023

Kubernetes. 2019. Case study - Spotify: Kubernetes. Internet publication. Kubernetes. Updated 2019. https://kubernetes.io/case-studies/spotify/. Accessed 11.2023.

Mass, Waltham. 2018. IT Complexity Soars. Internet publication. Dynatrace. Updated January 31, 2018. https://www.dynatrace.com/news/press-release/76-cios-say-become-impossible-manage-digital-performance-complexity-soars/. Accessed 11.2023.

Mazzara, Manuel, Nicola, Dragoni, Antonio Bucchiarone, Alberto Giaretta, Schahram Dustdar 2021. Microservices: Migration of a Mission Critical System. IEEE.

Netflix. 2016. NetflixOSS. Internet Publication. Netflix. Updated 2023. https://netflix.github.io. Accessed 12.2023.

Newman, Sam 2021. Building Microservices, 2nd Edition. O'Reilly.

Newman, Sam 2019. Monolith to Microservices. O'Reilly.

Pachghare, Vinod 2016. Microservices Architecture for Cloud Computing. MAT Journals.

Richards, Mark 2016. Microservices vs. Service-Oriented Architecture. O'Reilly.

Richardson, Chris. 2023. Adopt the Microservice Architecture. Internet publication. Microservices Architecture. Updated 2023. https://microservices.io/adopt/. Accessed 11.2023.

Varshneya, Ketan 2022. Decoding Software Architecture Of Spotify. Internet publication. TechAhead. Updated July 13, 2022. https://www.techaheadcorp.com/blog/decoding-software-architecture-of-spotify-how-microservices-empowers-spotify/. Accessed 12.2023.