Evolving Software Architectures from Monolithic Systems to Resilient Microservices: Best Practices, Challenges and Future Trends

Martin Kaloudis
Provadis School of International Management and Technology, Frankfurt, Germany

Abstract—Microservice architecture has emerged as a widely adopted methodology in software development, addressing the inherent limitations of traditional monolithic and Service-Oriented Architectures (SOA). This paper examines the evolution of microservices, emphasising their advantages in enhancing flexibility, scalability, and fault tolerance compared to legacy models. Through detailed case studies, it explores how leading companies, such as Netflix and Amazon, have leveraged microservices to optimise resource utilisation and operational adaptability. The study also addresses significant implementation challenges, including ensuring data consistency and managing APIs. Best practices, such as Domain-Driven Design (DDD) and the Saga Pattern, are evaluated with examples from Uber's cross-functional teams and Airbnb's transaction management. This research synthesises these findings into actionable guidelines for organisations transitioning from monolithic architectures, proposing a phased migration approach to mitigate risks and improve operational agility. Furthermore, the paper explores future trends, such as Kubernetes and AIOps, offering insights into the evolving microservices landscape and their potential to improve system scalability and resilience. The scientific contribution of this article lies in the development of practical best practices, providing a structured strategy for organisations seeking to modernise their IT infrastructure.

Keywords—Service-Orientated Architecture; SOA; microservices; monolithic architecture; migration

I. Introduction

Microservice architectures [1] have gained significant traction in recent years, primarily due to their ability to address the scalability and flexibility limitations of traditional monolithic systems. This architectural paradigm shift is driven by the growing need to manage complex, distributed applications efficiently. By decomposing applications into independently deployable services, microservices offer enhanced modularity, fault tolerance, and adaptability, positioning themselves as a superior alternative to monolithic architectures in large-scale, dynamic environments.

While the benefits of microservices, including independent scalability, enhanced fault isolation, and faster deployment cycles, are well-documented, their adoption is not without challenges. Ensuring consistency in data across distributed services remains a critical issue, particularly in environments where services manage their own databases. Furthermore, the operational overhead of managing an increasing number of APIs can result in significant complexity, particularly as systems grow in scale. These issues underscore the need for robust strategies to mitigate the operational challenges inherent in microservice architectures.

This study contributes to the ongoing discourse by proposing a structured approach to the transition from monolithic to microservice architectures, focusing on best practices derived from industry case studies. While existing literature extensively covers the theoretical benefits of microservices, there is a notable gap in actionable, empirically validated strategies for managing the complexities associated with their implementation. By analyzing case studies from industry leaders such as Netflix and Amazon, this research offers a phased migration strategy that minimizes risks and operational disruption. The novelty of this study lies in its practical framework for managing the inherent challenges of microservices, particularly in the context of large-scale enterprise systems.

II. Theoretical Basics

A. *Definition and Characteristics of Microservice Architecture*

Microservice architecture is a software development approach in which an application is developed as a collection of small, independent services. Each service fulfils a specific business requirement and communicates with other services via precisely defined APIs [2]. Microservices are small, independent services that fulfil specific business requirements. In [3] it is emphasised that this architecture simplifies the development and maintenance of complex systems due to its loose coupling and high cohesion. The author in [4] emphasises that microservices are particularly suitable for systems that place high demands on scalability and flexibility. One of the features of microservice architecture is decentralisation, in which services, functions and data are decentralised, resulting in a loose coupling of components. This promotes the application's reliability and fault tolerance. Another feature is independent development and deployment, which means that each service can be developed, tested and deployed independently. Errors in one service do not affect the entire application, which increases fault tolerance. Services can be reused in different applications, which increases efficiency and development speed [5].

B. *Monolithic Architecture*

Monolithic architecture is a traditional approach to software development in which all components of an application are integrated into a single, cohesive code base. This tight integration means that the application is developed, tested and deployed as an inseparable whole. A key feature of this approach is dependent scaling: if one component of the application experiences a higher load and requires more resources, the entire application must be scaled. This can be inefficient and resource-intensive, as the underutilised parts of the application also have to be scaled.

Another feature of monolithic architecture is that development cycles tend to be longer. As all components are closely interlinked, a change in one part of the application potentially affects many other parts. This requires extensive testing and can delay the release of new features. Any change, no matter how small, often requires a redeployment of the entire application, which is not only time-consuming but can also lead to downtime. This downtime can be particularly critical if the application provides business-critical functionality. Monolithic systems (see Fig. 1) have traditionally been favoured for their consistency and simplicity of implementation and management. Developers only have to deal with one code base and one deployment process. This can speed up initial development and simplify management, especially for smaller applications or teams. The clear structure and centralised management of dependencies and configurations make monolithic architectures attractive for many use cases.

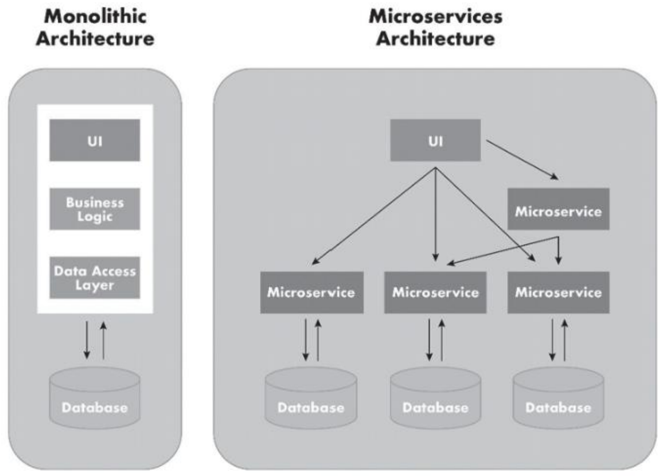


Fig. 1. Monolithic vs. Microservices architecture from [6].

To summarise, although monolithic architecture offers advantages due to its simplicity and consistency, it has significant disadvantages in an increasingly dynamic and scaled IT landscape. The lack of flexibility and scalability as well as the potential risks due to the tight integration of components make it unsuitable for many modern use cases. These disadvantages have led to the development and spread of more flexible and scalable architectures such as microservice architecture, which eliminate the specific weaknesses of the monolithic approach.

C. Microservices Architecture

In contrast to monolithic architecture, microservices divide an application into a collection of loosely coupled, independently deployable services, each of which fulfils specific business requirements. This architecture offers better scalability, flexibility and fault tolerance, but requires advanced knowledge of distributed systems development and DevOps practices. Each microservice has its own database and can be developed, tested and deployed independently, reducing infrastructure complexity and enabling more efficient resource utilisation [2].

1) Decentralisation and loose coupling: The microservice architecture is characterised by a fundamental decentralisation of services, functions and data. Instead of developing a monolithic application that combines all functions and processes in a single, closely linked structure, the microservice architecture breaks down the application into a large number of smaller, independent services. Each of these services, also known as a microservice, is designed to fulfil a specific business requirement or functionality. These services are not only functionally independent, but also often operate in isolated runtime environments and have their own databases. This means that each microservice manages and stores its own data, which ensures better data consistency. This decentralisation leads to a loose coupling of the components. Loose coupling means that the individual services are only minimally dependent on each other. Changes or errors in one service therefore have little to no impact on the other services. This decoupling enhances the overall resilience of the application, defined as its capacity to maintain operational continuity in the presence of faults or failures. Resilience describes the ability of a system to remain functional despite errors or faults. In a monolithic architecture, an error in one component can affect the entire application, whereas in a microservice architecture, an error remains isolated and the other services continue to function normally. This not only reduces fault tolerance, but also increases the overall reliability of the application [2].

One of the critical features of microservice architecture is its ability to scale individual services independently. As each microservice has its own database and is operated independently of the other services, each service can be scaled individually depending on the specific requirements and the load to be managed. This is particularly beneficial in cloud environments where resources can be allocated dynamically. For example, if a particular service has a high volume of traffic, it can be scaled independently of the other services without having to scale the entire application. This leads to more efficient resource utilisation and lower operating costs. Separation into independent services also improves fault isolation. Fault isolation means that problems in one service do not directly affect other services. If a microservice fails or a problem occurs, this error is limited to the affected service and does not affect the entire application. This not only makes troubleshooting easier, but also increases the reliability of the application. Developers can focus on fixing the specific problem without having to worry about changes to one service negatively impacting other parts of the application.

By decentralising services, functions and data, the microservice architecture offers considerable advantages in terms of reliability, fault tolerance, data consistency and scalability. The loose coupling of the services leads to increased robustness of the application, as errors remain isolated and the other services can continue to work undisturbed. Independent scalability enables efficient resource utilisation and reduces operating costs, while improved fault isolation and recovery increases the overall reliability of the application [6].

2) Scalability and fault tolerance: The ability to scale independently is one of the most outstanding features of microservice architecture and brings significant benefits in terms of resource management and increased efficiency. In a conventional monolithic architecture, all components of an application must be scaled together, even if only a small part of the application actually experiences an increased load. This leads to inefficient resource utilisation and increased costs, as not all parts of the application require the same scalability. In contrast, microservices enable targeted and demand-orientated scaling of individual services. Each microservice can be scaled independently of the other services, based on the specific requirements and the current load [7]. Splitting the application into independent services also has a positive effect on the fault tolerance of the entire system architecture. In a monolithic system, an error in one component can affect the entire system and lead to a total failure. This is because the components are closely interconnected and there is a dependency that disrupts the entire operating process. Microservices, on the other hand, isolate these errors to the affected service. If a microservice fails or an error occurs, this has no impact on the other services. The application remains functional and the affected microservice can be analysed and repaired in isolation.

Another aspect that increases fault tolerance is the ability to recognise and rectify errors automatically. Modern microservice architectures often utilise monitoring and management tools that continuously monitor the status of the services and react automatically in the event of anomalies or errors. This can be done by restarting the faulty service, switching to redundant services or dynamically reallocating resources. These automated processes reduce downtimes and improve the overall availability of the application [8]. The resilience, i.e. the ability of a system to recover from disruptions, is significantly improved by the microservice architecture. The loose coupling of the services means that they can work largely independently of each other. This independence allows the system to react flexibly to changes or failures without affecting the entire application. If a service is overloaded by a sudden increase in requests, it can be scaled in isolation to cope with the increased load. Should a service nevertheless fail, alternative services or failover mechanisms can be activated to ensure the continuity of business processes.

3) Independent development and provision: Microservice architecture facilitates the independent development and deployment of software components, yielding considerable improvements in both the efficiency and agility of the development process. In traditional monolithic architectures, all parts of an application must be developed, tested and deployed as a single unit. This means that even small changes to a component require extensive testing and full deployment of the entire application. This dependency leads to longer development cycles, an increased risk of errors and downtime as well as limited flexibility when implementing new functions [5]. A key advantage of this independent development and deployment is improved fault isolation. In a monolithic architecture, an error in one component can affect the entire application, which can lead to extensive downtime and difficult troubleshooting. In a microservice architecture, an error in one service remains limited to that specific service and does not affect the other parts of the application. This independence of services also encourages parallel development by different teams. In monolithic systems, development teams must coordinate their work closely to avoid conflicts, which can slow down development processes. In a microservice architecture, different teams can work on different services at the same time without their work interfering with each other.

The independent development and provision of microservices also supports better scalability of development resources. In monolithic systems, the scaling of development teams is often limited, as all teams have to work on the same code base and coordinate changes. In a microservice architecture, development teams can be scaled flexibly as they work independently on different services. This allows organisations to use their development resources more efficiently and respond more quickly to business requirements, resulting in faster implementation and improving the flexibility and agility of development processes. Improved fault isolation, parallel development by different teams and support for CI/CD practices lead to faster and more reliable releases, higher productivity of development teams and better scalability of development resources [5].

4) Reusability and flexibility in technology selection: Flexibility in technology selection allows teams to develop customised solutions that are optimised for their specific business needs. For example, a team working on a data-intensive analytics service might choose a programming language such as Python, which is known for its powerful data science libraries and frameworks. Another team developing a high-performance, critical real-time service might choose a language like Go or Rust, which are known for their efficiency and low latency. This freedom in technology choice leads to a better customisation of solutions to the specific needs of each service and therefore to business requirements [5]. Another advantage of this flexibility is the ability to introduce and use specialised technologies that are particularly suitable for specific tasks. Teams can select technologies that best fit the requirements and challenges of their specific microservices without having to consider the rest of the application. This can lead to a significant improvement in performance and efficiency. For example, a team working on a machine learning model could use specific frameworks and hardware acceleration to optimise training times and model accuracy [9].

The reusability and technological independence of microservices also help to reduce technical debt. Technical debt arises when short-term solutions are chosen that lead to higher maintenance costs in the long term. By using proven and reusable services, development teams can create consistent and maintainable code bases that reduce long-term maintenance efforts. In addition, flexibility in technology selection allows teams to continuously use the best tools and practices to minimise technical debt [4].

These advantages contribute to the microservice architecture being a favoured choice for modern, scalable and flexible software development projects [7].

III. Why and How Split Monoliths?

Industries such as retail, travel and transport and automotive have increasingly begun to break up their monolithic applications into microservices in order to become more flexible and scalable. This change is being driven by the need to respond more quickly to market changes and reduce total cost of ownership (TCO). However, the transition from monolithic to microservice architectures is a complex process that requires careful planning and a systematic approach. Fig. 2 shows cost-based determination of granularity services.

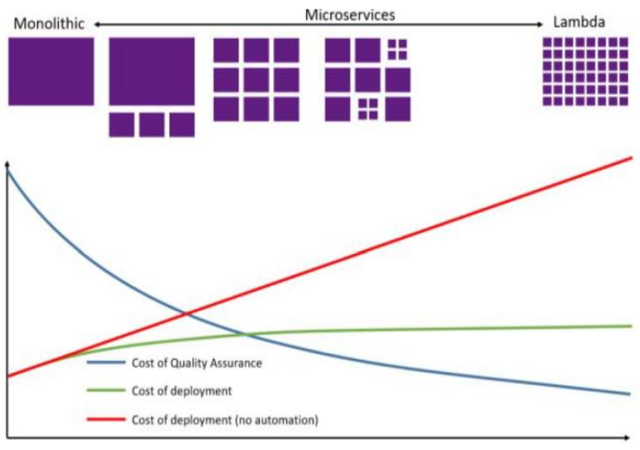


Fig. 2. Cost-based determination of granularity services [10].

A. Reasons for the Switch to Microservices

An important reason for this migration is the greater flexibility that microservices offer. In retail, for example, companies can develop and introduce new functionalities faster by splitting their applications into smaller, independent services. This is particularly important in a market that is constantly evolving and where competition is fierce. Retailers need to be able to respond quickly to new trends and customer demands, be it by introducing new payment methods, optimising supply chains or personalising the shopping experience.

B. Systematic Analysis and Step-by-Step Migration

The process of migrating from monolithic to microservice architectures often begins with a comprehensive and systematic analysis of the existing architecture. The aim of this analysis is to identify the current dependencies, bottlenecks and weak points. Based on these findings, companies can develop a clear migration strategy that is implemented step by step. A complete switch to microservices in a single step is usually too risky and too complex. Therefore, many companies prefer a step-by-step migration in which they gradually break down the application into microservices.

C. Identification of Business Areas

An important aspect of migration is the definition of business units. Business units are functional areas within an organisation that have clearly defined responsibilities. For example, a retailer might define business domains such as inventory management, order fulfilment, customer service and payment processing. Each of these domains can then be implemented as an independent microservice. This clear demarcation makes it possible to reduce the complexity of the overall application and clearly define responsibilities.

D. Formation of Cross-Functional Teams

Another important step in the migration process is the formation of cross-functional teams. Traditionally, development and operations teams are separate in many organisations, which can lead to communication problems and delays. However, microservice architecture requires close collaboration between these teams. Cross-functional teams consisting of developers, testers, operations experts and other relevant professionals can make the development and deployment of microservices more efficient. These teams are responsible for the entire lifecycle of a microservice, from development and testing to deployment and maintenance.

E. Introduction of DevOps Processes

The introduction of DevOps practices is another key component in the transition to microservices. DevOps stands for the integration of development and operations and aims to improve collaboration between these two areas. DevOps practices include Continuous Integration (CI) and Continuous Delivery (CD), which enable faster and more reliable delivery of software. By using automation tools and processes, companies can increase their efficiency, reduce the error rate and shorten the time to market for new functions.

IV. Challenges of Migration

Overall, the transition from monolithic applications to microservices offers significant benefits for many companies in the retail, travel and transport and automotive industries. By conducting systematic analyses, identifying business units, forming cross-functional teams and implementing DevOps practices, companies can make their IT infrastructure more flexible and scalable. A step-by-step migration minimises risks and enables continuous adaptation to changing market requirements. Research and practical reports prove the positive effects of this transformation on the efficiency and competitiveness of companies [10].

Procedure: The migration from a monolithic to a microservice architecture is a complex process that requires careful planning. It usually starts with the identification and extraction of business domains as independent microservices. Business domains, i.e. specific areas within an organisation, form the basis for the new architecture [11].

Analyse the existing architecture: The first step is to analyse the monolithic system to understand the dependencies between the components. Tools can automatically create dependency diagrams that help to visualise the interactions and control the migration process.

Development of a migration plan: Based on this analysis, a detailed migration plan should be created outlining the steps to minimise risk and ensure continuity. The plan should prioritise the domains to be migrated according to their business value and technical complexity.

Identifying and extracting business areas: Business areas, such as the product catalogue or payment processing in an e-commerce platform, must be clearly defined before they can be implemented as independent microservices.

Support for DDD: DDD helps to manage complexity by dividing systems into manageable units. The concept of "bounded context" ensures that each microservice has clear boundaries, which simplifies development and scaling.

Minimising risk and ensuring continuity: During the migration, mechanisms such as APIs or messaging systems help to ensure communication between microservices and the monolith and maintain continuity during the process.

Iterative development and continuous improvement: Migration should be viewed as iterative. Each migrated domain provides insights for optimising the process for future domains.

Static and dynamic analysis: Static analysis checks the source code to determine dependencies, while dynamic analysis monitors runtime behaviour and helps to prioritise services based on usage patterns.

By combining these approaches, companies can reduce system complexity and build scalable, maintainable architectures.

Development of a comprehensive migration plan: Based on the findings from the static and dynamic analysis, a comprehensive migration plan can be developed. This plan should take into account the identified services and interfaces as well as the prioritised usage patterns. It contains detailed steps for carrying out the migration, including the order of migration of the individual services, the necessary changes to the infrastructure and the implementation of transition mechanisms to ensure business continuity.

Static and dynamic analyses are crucial methods for preparing a successful migration from monolithic to microservice architectures. While static analysis reveals the structure and dependencies of the existing system, dynamic analysis provides valuable insights into the actual usage and performance of the application. By combining both approaches, organisations can develop a solid and low-risk migration strategy that takes into account both technical and operational aspects.

V. Advantages of the Integration of Microservices

Microservices offer numerous advantages that improve software development, reduce operating costs and simplify maintenance. Key benefits include independent development and deployment, efficient resource utilisation, technological flexibility, fault isolation and reusability of services.

One of the most important advantages of microservices is the ability to develop and deploy services independently of each other. In contrast to monolithic architectures, where every change requires extensive testing and a complete deployment of the entire system, microservices allow individual services to be updated independently of each other. This speeds up development, reduces errors and facilitates continuous deployment [11].

Efficient use of resources: Microservices enable independent scaling of services and thus optimise resource allocation. For example, if a service experiences increased demand, it can be scaled independently without affecting the rest of the system. This improves performance and reduces costs, especially compared to monolithic systems where the entire application has to be scaled [12].

Technological flexibility: Each microservice can be developed with the technology best suited to its needs. This allows development teams to innovate and customise solutions more effectively. Teams working on performance-critical components may opt for faster, more efficient languages, while others may favour simple development [13].

Isolation of errors: Errors in a microservice do not affect the entire application, which increases reliability. Isolated errors make it easier to diagnose and rectify problems without causing system-wide downtime [14].

Reusability of services: Microservices are independent units that can be reused in different applications. For example, an authentication service developed for one application can be reused in other projects, which saves development time and ensures consistency and security.

These advantages make microservices a favoured choice for modern, scalable and flexible software systems.

VI. Performance of Microservice Architectures: Case Studies

A. Case Study I: Comparison of Performance

A performance comparison between monolithic and microservice architectures shows clear differences in efficiency and scalability under different load conditions. In this case study, extensive tests were carried out to analyse the performance of the two architectures (see Fig. 3).

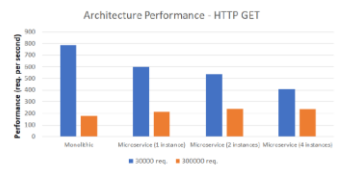


Fig. 8. Architecture performance (HTTP GET) - results

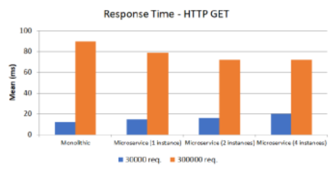
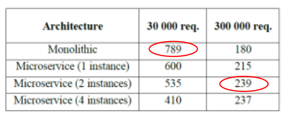


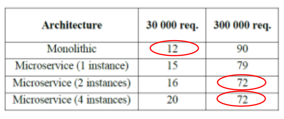
Fig. 9. Response time (HTTP GET) - results

TABLE I. ARCHTECTURE PERFORMANCE – HTTP GET



Architecture	30 000 req.	300 000 req.
Monolithic	789	180
Microservice (1 instance)	600	215
Microservice (2 instances)	535	239
Microservice (4 instances)	410	237

TABLE II. RESPONSE TIME – HTTP GET



Architecture	30 000 req.	300 000 req.
Monolithic	12	90
Microservice (1 instance)	15	79
Microservice (2 instances)	16	72
Microservice (4 instances)	20	72

Fig. 3. Performance test from [15].

Performance under lower load: The tests showed that monolithic architectures work more efficiently under lower load. This is because monolithic architectures combine all components and functions into a single, integrated application. This tight integration enables optimised use of resources and minimal communication latency between components. With low user numbers and few requests, the monolithic architecture is therefore able to deliver stable and fast response times. The reduced complexity and the lack of communication effort between distributed services contribute to greater efficiency under low load.

Performance at higher loads: As the load increases, however, the results change in favour of the microservice architecture. The tests showed that microservices scale better at higher loads and therefore achieve better performance results. In scenarios with increasing numbers of users and requests, the microservice architecture was able to handle the load more efficiently thanks to horizontal scaling. This means that additional instances of the individual microservices were provided to meet the increased demand. Of particular note is the use of replication, where multiple copies of a microservice are operated simultaneously to evenly distribute the load and increase availability. This ability to scale flexibly and on demand leads to improved performance under high load compared to monolithic systems. Fig. 4 shows the performance tests.

Monolithic architectures can therefore be more efficient at low loads, while microservice architectures show their strengths at high loads and scalability. The choice of architecture should therefore be based on the specific load requirements and the expected usage patterns.

B. Case Study II: Scalability and Reliability

The scalability and reliability of an application are critical factors for its performance and usability. This case study highlights the differences between horizontal and vertical scaling and shows how the choice of scaling strategy influences the scalability and reliability of the application.

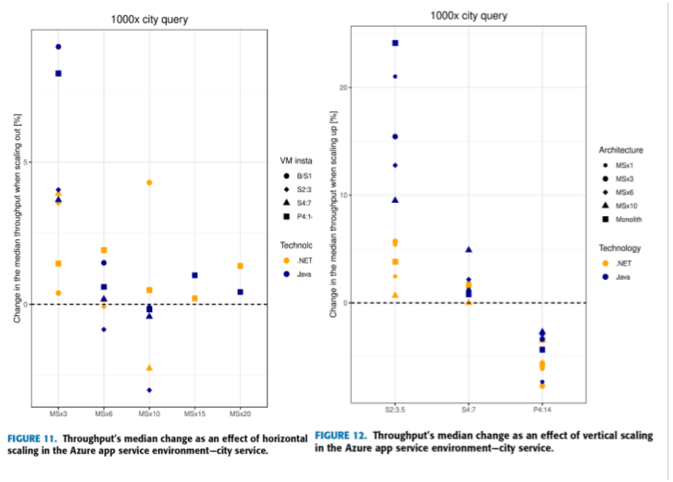


FIGURE 11. Throughput's median change as an effect of horizontal scaling in the Azure app service environment—city service.

FIGURE 12. Throughput's median change as an effect of vertical scaling in the Azure app service environment—city service.

Fig. 4. Performance test from [16].

Horizontal scaling: With horizontal scaling, also known as "scaling out", additional instances of a service are added to cope with the load. This method is particularly suitable for applications with a high load and the need for flexible scalability. Horizontal scaling involves running multiple copies of the service in parallel, which distributes the load evenly. This not only improves performance, but also increases fault tolerance, as the failure of one instance can be compensated for by the other instances. An example of this could be a web server that is supported by additional server instances when data traffic increases in order to distribute requests efficiently and minimise response times.

Vertical scaling: Vertical scaling, also known as "upscaling", involves adding additional resources to a single instance of a service, e.g. more CPU, RAM or storage space. This method is better suited to low to medium load applications where requirements can be met by upgrading existing hardware. Vertical scaling can be easier to implement as no changes to the software architecture are required. However, it comes up against physical and economic limits, as the performance of a single instance cannot be increased indefinitely. A typical example would be a database that is scaled by adding more memory and more powerful processors to enable the processing of larger amounts of data.

C. Decision Criteria for the Scaling Strategy

The decision between horizontal and vertical scaling depends on various factors, including the specific requirements of the application, the expected load patterns and the existing infrastructure. Horizontal scaling offers more flexibility and higher fault tolerance, but is more complex to implement and manage. Vertical scaling is easier to implement, but has limited scalability and can reach its limits with very high loads.

Horizontal scaling may therefore be ideal for applications with high loads and flexible scaling requirements, while vertical scaling may be suitable for applications with moderate loads and specific hardware requirements. The decision in favour of one or the other strategy should be carefully weighed up based on the individual requirements and objectives of the application [15].

VII. Results and Best Practices

The scientific literature and documented case studies, for example on the performance of monolithic and microservices architectures, which this article provides an overview of, are diverse. What has been missing so far is a best-practice approach that generically describes how to proceed with a monolithic application landscape in order to achieve a decentralised and resilient microservices architecture - in other words, a kind of "recipe". This is proposed in this article and inductively derived from the existing articles cited above, consolidated and outlined below.

Migrating from monolithic systems to a microservice architecture is a strategically challenging task that requires not only technical expertise, but also careful planning and implementation. There is no "best-of-breed" approach because, as described above, the procedural and technological complexity of application architectures in companies is individual.

However, reference can be made to examples from which a generic migration path can be derived. Based on the case studies analysed above and the scientific work, the author recommends the following best practice derived from case studies and thus a strategy for the analysis and implementation of microservice architectures.

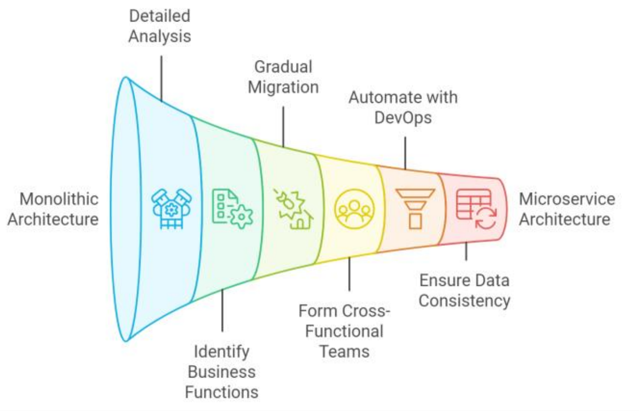


Fig. 5. Migration from monolithic to microservices.

1) Detailed analysis of the existing architecture: A thorough analysis of the existing monolithic architecture is an important first step on the way to a successful migration to a microservice architecture (see Fig. 5). Without a deep understanding of the current components, their dependencies and interactions, splitting them into microservices can lead to unforeseen complications. The migration process should therefore begin with a comprehensive analysis of both the system architecture and the underlying code. Two primary approaches play a central role in this analysis: static and dynamic analysis.

Static analysis: In static analysis, the source code is analysed to determine the dependencies between individual modules and their relationships. This method helps to map the existing structure of the monolith and visualise the connections between the components. The tools used for static code analysis can create dependency diagrams that provide a clear overview of how the system works as a whole. These insights are crucial for identifying potential microservices and ensuring that the modularisation of the system is effective and sustainable [10].

Dynamic analysis: While static analysis focuses on the structure, dynamic analysis captures the behaviour of the application during runtime. Monitoring the real-time behaviour of the system allows engineers to understand usage patterns and critical business processes supported by specific modules. This method provides insight into performance bottlenecks and areas of the system in need of optimisation, which can inform which components should be prioritised for migration into standalone microservices.

Example: Amazon carried out a comprehensive analysis of its monolithic architecture before switching to a microservice architecture. Critical areas such as the product catalogue and the payment system were identified and spun off as independent microservices, which significantly improved the scalability of the system [13].

2) Identification and delimitation of business functions: Decomposing a monolithic system into well-defined business functions is essential for creating clear service boundaries when migrating to microservices. The domain-driven design approach ensures that each microservice corresponds to a logical business domain, resulting in loosely coupled services that can operate independently.

DDD: DDD is a methodological approach that focuses on mastering complexity by modelling business domains. A key concept in DDD is the "bounded context", which delineates a specific area within a business domain where a consistent model is applied. Defining these bounded contexts ensures that microservices have clear responsibilities, reducing interdependencies and simplifying development, maintenance and scaling.

Example: Netflix used DDD to separate user management from its video streaming service. This logical separation enabled independent development and provision of functions and minimised the risk of system-wide failures [13].

3) Gradual migration and minimisation of risks: A phased or step-by-step migration strategy is critical to minimising the risks associated with the transition from a monolithic architecture to microservices. A "big bang" migration - where the entire system is migrated at once - can lead to serious system failures and business disruption. Instead, migrating smaller, less critical components initially allows for a smoother and safer transition.

Step-by-step migration strategy: The aim of a step-by-step migration is to divide the migration process into smaller steps so that individual components of the monolith can be migrated gradually. This approach allows each microservice to be thoroughly tested to ensure that it works independently and integrates smoothly with the remaining monolith. By prioritising low-risk areas, companies can significantly reduce the likelihood of critical failures during migration [12].

Example: Spotify opted for a step-by-step migration by first converting its playlist management system to a microservice. This approach enabled the company to tackle problems in isolation and minimise the risk of widespread outages [5].

4) Formation of cross-functional teams: In addition to the technical changes, the migration to microservices also requires organisational restructuring. The formation of cross-functional teams is essential for the efficient development and maintenance of microservices. These teams consist of employees from different disciplines - development, operations and quality assurance - who work together to provide services more effectively.

Autonomous teams: Each cross-functional team has full responsibility for one or more microservices and works independently within the scope of the services assigned to it. This autonomy enables the teams to develop, deliver and optimise their services without being dependent on other teams, which increases productivity and flexibility within the company.

Example: Uber formed cross-functional teams that were responsible for individual microservices, such as route calculation. These teams worked independently of each other, which enabled faster adaptation to changes in the business model or technology, which significantly improved productivity [12].

5) Automate with DevOps practices: Automation plays a central role in the successful provision and management of microservices. Continuous Integration (CI) and Continuous Delivery (CD) are two key DevOps practices that ensure reliable and rapid delivery of microservices. CI/CD pipelines automate the processes of testing, building and deploying software, enabling faster releases and greater system stability.

DevOps practice: DevOps emphasises collaboration between development and operations teams to ensure continuous improvement of both development and deployment processes. By automating testing and deployment, human error is minimised, allowing teams to release frequent, smaller updates that improve the quality and reliability of the overall system.

Example: Facebook implemented CI/CD pipelines to provide several microservices every day. This enabled faster releases and greater agility in the development process [13].

6) Consideration of data consistency and error tolerance: Ensuring data consistency in distributed systems is one of the biggest challenges in microservice architectures (see Fig. 6). In a microservice system, each service often manages its own database, which can make consistency across the entire system difficult. Two important techniques for overcoming this challenge are event sourcing and the saga pattern.

Event sourcing: With event sourcing, the state of the application is saved as a sequence of events that change this state instead of saving the current state directly in the database. These events are stored in an event log that can be replayed as required to reconstruct the current state. This ensures that all changes are traceable and recoverable and that data consistency is maintained across distributed systems.

Saga pattern: The Saga pattern enables the decomposition of long transactions into smaller, atomic transactions that can be managed by individual microservices. Each transaction is designed to either complete or roll back in the event of an error, ensuring that all services involved either reach a consistent state or return to their previous state, thereby avoiding inconsistencies.

Example: Airbnb uses the Saga pattern to coordinate transactions such as bookings across multiple microservices. This pattern ensures that the system remains consistent even if errors occur in individual services [3].

7) Summary of the 6 phases: Successful migration from a monolithic to a microservice architecture requires a strategic, well-planned approach that takes both technical and organisational aspects into account.

Firstly, a thorough analysis of the existing system is crucial. Using static and dynamic analysis techniques ensures a comprehensive understanding of component dependencies and enables informed decisions on how to split the monolith into manageable microservices. Once the architecture is understood, it is important to identify and delineate the business functions using approaches such as DDD. This ensures that the microservices are aligned with the logical business domains, resulting in clear boundaries, reduced dependencies and more effective scaling. To minimise risks, a step-by-step migration strategy is recommended. Gradually migrating smaller, low-risk components allows for testing and optimisation, reducing the likelihood of critical errors and ensuring a smooth transition. On the organisational side, the formation of cross-functional teams is essential. These teams, made up of experts from development, operations and quality assurance, should be able to manage individual microservices independently to increase both productivity and flexibility. Automating processes through DevOps practices, particularly CI and CD, ensures that microservices are deployed efficiently and reliably. Automation minimises human error and speeds up the development cycle, allowing for frequent, smaller updates. Finally, ensuring data consistency and fault tolerance in distributed systems is a major challenge that can be addressed with techniques such as event sourcing and the Saga pattern. These methods help maintain consistent states and handle failures gracefully to ensure system reliability and robustness.

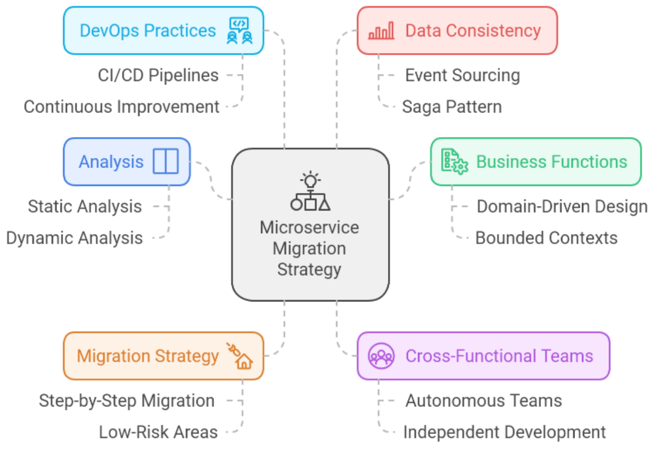


Fig. 6. Challenges and best practices in migrating from a monolithic to a microservice architecture.

If companies follow these best practices an included recommendations, they can minimise the risks of switching to microservices and at the same time benefit from the advantages of a scalable, flexible and resilient system architecture.

VIII. Discussion

A. *Data Consistency and Management of Distributed Data*

Microservice architecture's distributed nature enables greater flexibility in data management, allowing each service to optimize its data storage based on specific requirements. However, maintaining data consistency across distributed services presents significant challenges. Techniques like event sourcing and the Saga pattern are often employed to ensure synchronized, up-to-date transactions across services. While effective, these methods increase the complexity of system architecture, adding to operational overhead and necessitating advanced technical knowledge [3]. This complexity may not always justify the benefits for smaller organisations or systems with lower scalability needs, where a monolithic approach could be more practical [6].

B. Trade-offs Between Microservices and Traditional Architectures

Microservices offer clear advantages in scalability and fault tolerance, but there are notable trade-offs. For example, smaller organisations may struggle with the overhead of managing numerous independent services, APIs, and maintaining data consistency. Monolithic architectures, by contrast, can offer faster development cycles and simpler management, especially in smaller applications that don't require significant scalability [6]. As discussed by [12], the benefits of microservices tend to emerge in large-scale environments where scalability is critical. Thus, the decision to adopt microservices should be weighed against the organisation's size, technical capabilities, and future growth plans [7].

C. API Management and Overheads

While microservices communicate through APIs, promoting modular design and interoperability, the management of multiple APIs can become a burden as systems grow. This issue, often termed API proliferation, can increase operational complexity and management costs [5]. Solutions like API gateways and service meshes help centralize API management and streamline communication, providing advanced features such as load balancing and security. However, these tools also introduce new layers of infrastructure, which may pose challenges for smaller organisations without the technical resources to maintain them [10].

D. Testing Strategies for Microservices

One advantage of microservices is their ability to support isolated unit testing, reducing the risk of system-wide failures [18] [20]. Automated testing tools can efficiently validate microservices before deployment. However, ensuring that multiple microservices function as a cohesive system requires extensive integration testing, which can slow deployment cycles and increase operational complexity. As emphasized, integration testing in microservices introduces an additional layer of complexity not present in monolithic systems [17].

E. Increased Focus on Emerging Trends

Technologies like Kubernetes, AIOps, and serverless computing are shaping the future of microservices by offering advanced automation and orchestration capabilities. Kubernetes, for instance, simplifies the management of microservices by providing container orchestration tools for scaling and fault tolerance [18]. However, Kubernetes' complexity often requires specialised expertise, making it more suitable for larger enterprises. AIOps—which integrates machine learning to predict system failures and optimize performance—offers significant potential for improving microservice reliability, but also introduces additional complexity [19]. As pointed out, the success of these technologies depends on how well organisations can manage this complexity [8].

F. Linking Case Studies More Critically

The adoption of microservices by companies like Netflix, Amazon, and Uber has been widely documented, but the unique contexts that facilitated their success must be critically evaluated [6]. For example, Netflix's need for high reliability in streaming services and Amazon's demand for rapid scalability due to their e-commerce platform both necessitated the use of microservices [13]. However, smaller companies, or those in industries with lower scalability needs, may not experience the same benefits. Case studies of large companies should therefore be viewed with caution when attempting to generalise these strategies to smaller firms [5].

G. Potential Gaps in Existing Research

Although microservices have been widely adopted across various sectors, there remain gaps in research concerning their implementation in highly regulated industries like fintech and healthcare, where data security and regulatory compliance are crucial [11]. These industries face challenges in adopting decentralised architectures due to the need for strict data governance. More research is needed to explore how microservice best practices can be adapted for these sectors. Moreover, there is limited empirical data evaluating the long-term performance of microservices in such contexts [16].

H. More Quantitative Evaluation

Empirical studies have shown that monolithic systems tend to perform more efficiently under lower loads, while microservice architectures excel at higher loads, thanks to their ability to scale horizontally. Quantitative data on resource utilization, fault tolerance, and operational costs would provide a stronger foundation for decision-making when comparing these architectures [15]. For instance, valuable insights into the performance benefits of microservices under varying conditions is provided [16].

I. Critical Look at Migration Strategies

A phased migration approach, where organisations gradually transition from monolithic to microservice architectures, is often recommended to mitigate risks [7]. This approach allows for continuous testing and ensures that each microservice functions independently before migrating the entire system. However, this can also extend the migration [21] process and lead to technical debt, as both systems must be maintained during the transition. In some cases, a "big bang" migration, where the entire system is migrated at once, might be more efficient, especially for smaller systems [5]. Organisations must carefully assess their specific needs, technical capacity, and risk tolerance before deciding on the best migration strategy [10].

IX. Conclusion

To summarise, the evolution from monolithic to service-oriented and finally to microservice architectures represents a significant advance in the development and maintenance of modern software applications. Microservice architectures address many of the challenges associated with traditional monolithic systems and service-oriented architectures and offer significant improvements in flexibility, scalability and fault tolerance. By splitting complex applications into independent, loosely coupled services, microservices enable organisations to better respond to changing business requirements and optimise resource utilisation. As highlighted in the best practices and recommendations, the key benefits of microservices include their modular structure, which enables independent development, deployment and scalability. This modular approach allows organisations to scale services based on specific requirements without impacting the overall system.

However, data consistency and Application Programming Interface (API) management remain a major challenge and require sophisticated strategies such as event sourcing and the Saga pattern to maintain synchronisation across distributed systems.

The recommended step-by-step migration strategy helps to minimise the risks associated with the transition from monolithic systems to microservices. This step-by-step approach, together with cross-functional teams, ensures a smoother migration process and promotes collaboration between development, operations and quality assurance. In addition, the adoption of DevOps practices such as CI and CD increases the efficiency and reliability of microservice delivery, even though this requires a high level of technical maturity. While the benefits of microservices are obvious, the increased complexity and operational overhead created by their distributed nature, as well as the need for advanced skills in DevOps and container orchestration, present challenges that must be carefully managed. However, the integration of new technologies such as AIOps, Kubernetes and serverless computing increases the potential of microservice architectures and positions them as the dominant model for scalable, flexible and resilient software systems in the future.

Companies that strategically apply these best practices and recommendations will be better equipped to overcome the challenges of microservice architectures while maximising the benefits of scalability, flexibility and fault tolerance in their IT infrastructures.

REFERENCES

[1] Valdivia, J.A., Lora-González, J., Limón, X., Cortes-Verdin, K., & Ocharán-Hernández, J.O. (2020): “Patterns related to microservice architecture: a multivocal literature review”. Programme. Comput. Software, 46, 594-608.

[2] Newman, S. (2021). “Building Microservices”. O’Reilly Media, Inc.

[3] Abdelfattah, A.S. & Cerny, T. (2023): Roadmap to reasoning in microservice systems: a rapid review. Appl. Sci, 13, 1838.

[4] Cadavid, H., Andrikopoulos, V., & Avgeriou, P. (2020): “The architecture of systems of systems: A tertiary study”. Inf. Software Technol. 118, 106202.

[5] Lewis, J., & Fowler, M. (2014). “Microservices: A definition of this new architectural term”.

[6] Vecherskaya, S.E. (2023). "Tasks and evoluti on of microservice architecture", pp. 37-43, Complex systems: models, analysis, management, Bulletin of the Russian New University.

[7] Baškarada, S., Nguyen, V., & Koronios, A. (2020). „Architecture of Microservices: Practical Opportunities and Challenges”.

[8] Van Eyk, E., Iosup, A., Seif, S., & Thömmes, M. (2017). „The spec cloud group's research vision on faas and serverless architectures”. In WOSC 2017 - Proceedings of the 2nd International Workshop on Serverless Computing, Part of Middleware 2017 (pp. 1-4). Association for Computing Machinery, Inc. https://doi.org/10.1145/3154847.3154848.

[9] Zimmermann, O. “Microservices tenets”. Comput Sci Res Dev 32, 301-310 (2017).

[10] Gouigoux, P., & Tamzalit, D. (2017). “From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture”.

[11] Evans, E. (2004). Domain-Driven Design: “Tackling Complexity in the Heart of Software”. Addison-Wesley Professional.

[12] Baškarada, S., Nguyen, V., & Koronios, A. (2020). „Architecture of Microservices: Practical Opportunities and Challenges”.

[13] Newman, S. (2015). “Building Microservices”. O'Reilly Media.

[14] Dragoni, N., Lanese, I., Larsen, S.T., Mazzara, M., Mustafin, R., Safina, L. (2018). Microservices: “How To Make Your Application Scale”. In: Petrenko, A., Voronkov, A. (eds) Perspectives of System Informatics. PSI 2017. Lecture Notes in Computer Science(), vol 10742. Springer, Cham. https://doi.org/10.1007/978-3-319-74313-4_8.

[15] Gos, K., & Zabierowski, W. (2020). “The comparison of microservice and monolithic architecture”.

[16] Blinowski, G., Ojdowska, A. and Przybyłek, A., 2022, "Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation", in IEEE Access, vol. 10, pp. 20357-20374, 2022, doi: 10.1109/ACCESS.2022.3152803.

[17] Tran, H.K.V., Unterkalmsteiner, M., Börstler, J., & bin Ali, N. (2021). „Evaluating the quality of test artefacts - a tertiary study”. Inf. Software Technol.

[18] Arvanitou, E.M., Ampatzoglou, A., Bibi, S., Chatzigeorgiou, A., & Deligiannis, I. (2022): “Application and exploration of DevOps: A tertiary study”. IEEE Access, 10, 61585-61600.

[19] Liu, X., Li, S., Zhang, H., Zhong, C., Wang, Y., Waseem, M., & Babar, M.A. (2022): “Microservice architecture research: A tertiary study”. SSRN Electron. J.

[20] Alaasam, A.B., Radchenko, G., Tchernykh, A., & González Compeán, J.L. (2020): “Analytical study on containerisation of stateful stream processing as a microservice to support digital twins in fog computing. Programme”. Comput. Software, 46, 511-525.

[21] Fritzsch, J., Bogner, J., Wagner, S., & Zimmermann, A. (2019). „Microservices Migration in the Industry: Intentions, Strategies, and Challenges”.