Master Thesis

Master of Science (MSc.)
Department of Tech and Software
Major: Software Engineering

Refactorability Analysis: Simulating and Comparing Monolithic and Microservices Architectures in Scalable Applications

Berk Limoncu
Matriculation Number: 81142531
First supervisor: Prof. Dr. Rand Kouatly
Second supervisor: Prof. Dr. Iftikhar Ahmed
Submitted on: 27/02/2025

Statutory Declaration

I hereby declare that I have developed and written the enclosed Master Thesis completely by myself and have not used sources or means without declaration in the text. I clearly marked and separately listed all the literature and all the other sources which I employed when producing this academic work, either literally or in content. I am aware that the violation of this regulation will lead to the failure of the thesis.

Potsdam
Date: 27/02/2025 Signature

Abstract

Modern software systems' requires scalability, flexibility, and maintainability have been gradually increasing. A monolithic architecture pattern becomes ineffective for scaling up an application since such systems tend to have high coupling, complex deployment processes, and scalability challenges, preventing them from efficiently managing growing workloads. To address these challenges, a microservice architecture pattern was introduced. Microservices Architecture is a service-oriented architecture pattern that allows for independent scaling, fault isolation, and ease of deployment as the application is built as a collection of multiple, independent, small services. Transitioning from Monolithic-based architecture to Microservices-based architecture comes with its own set of challenges including but not limited to service communication, distributed data management, and migration complexity. One of the major architectural design decisions to be considered during this transition is the choice of the database model. That is whether to run a shared database across multiple services sharing tables associated with multiple services or to adopt a database-per-service model. Database-per-service model allows services to be autonomous but increases operation complexity & data consistency challenges whereas sharing a database simplifies most of the challenges that come with data sharing among multiple services but limits microservices' ability to scale up.

This thesis aims to explore the viability, scalability, and performance trade-offs of microservices based on a shared database approach. We devised a series of structured migration strategies based on the Strangler Fig Pattern to incrementally refactor monolithic components into microservices. We performed a series of performance evaluations to examine the scalability of the monolithic and microservices architectures. Evaluations specifically targeted three key aspects in the migration: performance of response times under concurrent workloads, CPU utilization, and database performance. Results indicate that microservices scale better than a monolithic architecture, but database contention in a shared database approach can also result in performance bottlenecks that ultimately throttle scalability. We additionally examine strategies to optimize database access, service orchestration, and API communication overhead.

The main contribution of this paper is the empirical investigation of the scalability of microservices using the same database. The resulting output will aid organizations in the process of moving away from a monolithic application to a distributed system over microservices. It also provides a way to deploy microservices online and conduct scalability testing in the real world using cloud computing. The analysis shows that although microservices provide the ability to scale specific modules of a system, an organization is at risk of not seeing those benefits due to its database strategy. More efficient database strategies can be used to obtain a much better and faster-performing system. Future work would include an investigation into databases that are not shared among the services, Event-Driven Architectures where services communicate asynchronously, and microservice migration through automated tasks.

Table of Contents

Chapter 1. Introduction *1*

1. 1 Background **1**

1. 2 Problem Statement **2**

1. 3 Research Questions **2**

1. 4 Research Objectives **3**

1. 5 Contributions **5**

1. 6 Thesis Organization **6**

Chapter 2. Theoretical Background *8*

2. 1 Understanding Architectural Transitions: From Monoliths to Microservice **8**

2. 2 Monolithic Architecture: A Unified but Rigid Approach **9**

2. 3 Microservice Architecture: A Modular and Scalable Paradigm **10**

2. 4 Scaling in Software Engineering: Vertical vs. Horizontal Scaling **11**

2. 5 Performance and Resource Utilization **11**

2. 6 Refactorability in Software Systems: Transitioning to Microservices **12**

2. 7 Theoretical Summary: Foundations for Implementation **12**

Chapter 3. Literature Review *13*

3. 1 Overview of Monolithic and Microservices Architectures **13**

3. 2 Refactorability: Definition and Importance **15**

3. 3 Challenges in Transitioning from Monolith to Microservices **16**

3. 4 Existing Transition Strategies and Best Practices **18**

3. 5 Discussion on Literature Review **21**

Chapter 4. Methodology *24*

4. 1 Overview **24**

4. 2 Proposed Framework/Model/Technique **25**

4.2.1 Architectural Design of Monolith vs. Microservices (Strangler Fig Pattern) 25

4.2.2 Service Decomposition Strategy 28

4.2.3 Data Management (Shared Database Model) 29

4.2.4 Deployment and Scaling 31

4. 3 Methodology **31**

4.3.1 System Design and Implementation 31

4.3.2 Evaluation Setup 33

4.3.3 Case Study/Experimental Setup 33

4. 4 Evaluation Criteria **35**

4. 5 Benchmark Algorithms **37**

Chapter 5. Results and Discussion *39*

5. 1 Quantative Insights from System **39**

5. 2 Performance Under Load: Docker Stats Analysis **39**

5.2.1 Monolithic System Performance 39
5.2.2 Microservices System Performance 41
5.2.3 CPU Usage Analysis 43
5.2.4 Memory Usage Analysis 44

5. 3 Performance Under Load: Response Time and Throughput Analysis 46
5.3.1 Monolithic System Performance 46
5.3.2 Microservices System Performance 47
5.3.3 Comparative Analysis of Response Time and Throughput 48

5. 4 Refactorability Time & Challenges 50

5. 5 Hardware & Infrastructure Cost 52

5. 6 Discussion 54

Chapter 6. Conclusion and Future Work 56

6. 1 Conclusion 56

6. 2 Future Work 57

List of Tables

Table 3.1:Comparison of Monolithic and Microservices Architectures(Powell and Smalley, 2024) 14
Table 3.2: Factors Affecting Refactorability (Monolithic vs Microservices - Difference Between Software Development Architectures- AWS, 2024) 15
Table 3.3: Challenges in Transitioning from Monolith to Microservices(Al-Debagy and Martinek, 2018).....18
Table 3.4: Comparison of Migration Approaches (AlOmar, Mkaouer and Ouni, 2024) 20
Table 3.5: Best Practices for Microservices Migration 20
Table 4.1: Comparison of Service Decomposition Strategies(Chaieb, Sellami and Saied, 2023) 28
Table 4.2: REST API Endpoints for both Monolithic and Microservice Architecture 32
Table 4.3: Summary of Evaluation Metrics and Their Purpose.....36
Table 5.1: CPU Utilization Data.....43
Table 5.2 : Memory Utilization Data.....44
Table 5.3: Time Spent Comparison (Monolith vs Microservice) 51
Table 5.4: Cost Analysis Table (Monolithic vs. Microservices) 53

List of Figures

Figure 2.1: Comparison of Monolithic and Microservices Architectures (Karwatka, 2020) 8
Figure 4.1: Monolithic vs. Microservices Transition (Source: (Monoliths to Microservices using the Strangler Pattern, no date))....26
Figure 4.2: Structural Differences Between Monolithic and Microservices Architecture 27
Figure 4.3: Entity Relationalship Diagram (ERD) 30
Figure 4.4: Vertical Scaling vs. Horizontal Scaling(Perry, 2023) 38
Figure 5.1: Monolith Resources in IDLE 40
Figure 5.2: Monolith Resorcues with 1000 User Load....40
Figure 5.3: Monolith Resources with 2500 User Load....40
Figure 5.4: Monolith Resources with 5000 User Load....41
Figure 5.5: Microservice Resources IDLE 41
Figure 5.6: Microservice Resources with 1000 User Load....42
Figure 5.7: Microservice Resources with 2500 User Load....42
Figure 5.8: Microservice Resources with 5000 User Load....42
Figure 5.9: CPU Utilization Trends 43
Figure 5.10: Memory Utilization Trends 45
Figure 5.11: Monolithic System Response Time and Throughput Under 1000 User Load 46
Figure 5.12:Monolithic System Response Time and Throughput Under 2500 User Load 47
Figure 5.13: Monolithic System Response Time and Throughput Under 5000 User Load 47
Figure 5.14: Microservice System Response Time and Throughput Under 1000 User Load 47
Figure 5.15: Microservice System Response Time and Throughput Under 2500 User Load 48
Figure 5.16: Microservice System Response Time and Throughput Under 5000 User Load 48
Figure 5.17: Response Time vs. Users (Monolith vs. Microservices)....49
Figure 5.18: Throughput vs. Users (Monolith vs. Microservices)....49

List of Abbreviations

ACID	Atomicity, Consistency, Isolation, and Durability
CI/CD	Continuous Integration and Delivery
CPU	Central Process Unit
DDD	Domain Driven Design
EDD	Event Driven Design
ERD	Entity Relationship Diagram
RAM	Random Access Memory
SFP	Strangler Fig Pattern
VCS	Version Control System

Chapter 1. Introduction

1. 1 Background

As software businesses seek higher scalability, more agility, and longer maintainability in their modern software systems, they are compelled to adopt architectural paradigms that satisfy these conflicting requirements: Historically all components, the user interface, the business logic, and the data access layer were tightly coupled into a single codebase and deployment unit in monolithic architectures usually also all three per server instance. On top of that, such a system markets itself as providing central control over its architecture while simplifying initial development and debugging. Problems are always just around the corner though: with monolithic systems, this means that when you want to increase an entire subsystem's performance by an order of magnitude, every bit has to be rewritten.

Moving from monolithic architectures to microservices architectures results in applications where modules are more loosely coupled and can be separately deployed, each providing a specific business capability. The modularity of microservices not only enhances scalability but also improves fault isolation and allows businesses to choose different technologies for different services (Newman, 2019). However, transitioning to a microservices environment introduces complexities, such as distributed system coordination, network latency, and increased operational overhead compared to monolithic systems (Dragoni *et al.*, 2017). In particular, inter-process communication mechanisms like REST APIs and message brokers (e.g., Kafka, RabbitMQ) are more susceptible to bottlenecks due to network latency and data serialization overhead (*Apache Kafka*, no date).

Among the debates over resource utilization and long-term maintenance, a recurring topic is whether a particular venture's architecture should be monolithic or microservice-based: Monolithic systems take advantage of cheap hardware and share resources across components. However, they face a single point of failure and cannot tear out part of themselves without bleeding everywhere. This kind of software also tends to be updated at once rather than in incremental mode. Microservices allow precise scaling (e.g., auto-scaling high-demand services), but require duplicate infrastructure (e.g. separate databases, containers) and orchestration tools like Kubernetes (Al-Debagy and Martinek, 2018).

Moving from monolithic architectures to microservices has significant complications including loosening tightly bundled components, restructuring data flows, and dealing with cross-cutting concerns such as system orchestration (Kristiyanto *et al.*, 2024). Creating completely separate containers for each small partition, with communication in between as a means of releasing tension, set the buildings plainly out of reach. The proposed infrastructure and integration mechanisms required to support major changes detract from the simplicity of would-be rapid migrations. Source Code trials show that the development of its own indigenous middleware component was particularly useful to shed light on interfaces and operational routines in service communication, yet this was also just one example. Reporting on a new IT practice at Chesnas since the beginning of last month (known as 'tiny bridge' architecture) set up finite parameters to tackle these issues within reasonable limits to simulate this process, a "microbridge" middleware was developed to manage inter-service communication, reflecting real-world transition hurdles (Salaheddin Elgheriani and Ali Salem Ahme, 2022)

1. 2 Problem Statement

The monolithic vs. microservices debate has gained traction among practitioners as the complexity of scalable applications grows (Dragoni *et al.*, 2017). Although monolithic architectures are simple and easy to manage, microservices architectures provide significant benefits such as increased modularity and fault isolation (Newman, 2019). Nevertheless, empirical studies measuring their refactorability when faced with scalability constraints remain sparse (Santos, 2018). Recent work has shown that the transition between monolithic and microservices architectures often reveals problems related to data consistency, and service decomposition (Dragoni *et al.*, 2017), as well as higher operational complexity (Tapia *et al.*, 2020). While these issues are particular to MTSs and MTS architectures, they are still not addressed in the current literature, which we attribute to a lack of empirical work on these topics.

Since my simulated projects had structural transitions, I noticed critical problems such as inefficiency in resource usage and performance bottlenecks. Apache JMeter (*Apache JMeter - Apache JMeter*^{TM}, no date) was used to compare the performance of these projects and these metrics will be discussed in more detail in the Results and Discussion section. The microservices architecture, as an example, had issues with latency due to inter-service communication, and the monolithic system was limited in its ability to handle concurrent user loads. This result emphasizes the importance of systematic comparisons of each architecture under scalability constraints, at least in terms of operational cost and reusability.

Organizations are exposed to expensive, trial-and-error migration tactics as a result of this mismatch, which frequently leads to operational bottlenecks and delayed deployments. Our creation of the "microbridge" middleware, for instance, brought to light the difficulties in overseeing inter-service communication and redesigning data flow—two crucial but little-studied facets of architectural transitions (Mehta *et al.*, 2024).

This research project aims to bridge this gap by developing a refactorability analysis framework, leveraging scalability simulations and middleware prototypes to deliver data-driven insights for architectural decision-making.

1. 3 Research Questions

A research question is a question that a research project sets out to answer. Choosing the right architecture is important when considering the performance, scalability, and operation. Over the past decade many organizations have started to transform from monolithic architecture to microservices or have directly migrated to microservices. Hence, there is a need to understand the implications of migrating to microservice architecture. This paper intends to answer the above problem with the help of some leading research questions:

- What performance metrics are indicators of performance change?
- How much does scalable this project for both Monolithic and Microservices architecture?
- How much refactoring effort matters in determining the serviceability of microservices?
- How disruptive is it to migrate from monolithic to microservices?
- What would this migration or refactoring cost in terms of Development vs Operation Cost?

This study seeks to answer the following research questions:

1. Performance Metrics: What differences are observed between the monolithic and microservices versions of projects I developed to understand the difference between microservice and monolith in terms of key performance metrics, such as response time, CPU utilization, and memory consumption, when subjected to realistic workload patterns? Quantifying these performance variations is imperative to determine whether the microservices-inspired refactoring has any apparent performance improvements or whether the architectural overhead of distributed systems incurs additional inefficiency.

2. Scalability Considerations: What are the comparative strengths and weaknesses of the monolithic and microservices architectures as applied to projects when considering workload scale and increased traffic demands? While microservices architectures have conventionally been linked to improved scalability, this research questions whether any purported benefits truly outweigh the potential inefficiencies of increased complexity.

3. Refactoring Effort & Quantification: What is the migration effort required to refactor the existing monolithic project application to adapt a microservices-inspired architecture, and what reasonable metrics could be applied to estimate the quantitative effort involved? Software refactoring is often a resource-intensive endeavor that requires significant development time, architectural refactoring, and dependency reallocation. This research will provide different methodologies to quantify the overall refactoring feasibility between monolithic and microservices approaches, specifically considering code complexity metrics and deployment iterations.

4. Migration Challenges: What are the key challenges in the process of migration from the monolithic to the microservices version, and what architectural and practical concerns must be addressed to facilitate such a migration? The scope of prioritization for these challenges has grown significantly to span not only technical service orchestration and data consistency concerns, but also team organization structure, deployment methodology, and operational procedure.

5. Infrastructure & Cost Differences: What are the infrastructure, hardware, and associated resource requirements for the monolithic and microservices, and what is the comparative cost of deployment and operation between microservices and monolithic counterparts? Despite claims that microservices encourage modular scaling rather than monolithic over-provisioning, the elevated changes incurred from excessive network overhead and containerized cloud orchestration could begin to increase observed operating costs instead. By quantifying the comparative costing information between the two approaches, this research addresses whether the advantages of microservices structures are worth the potential monetary and infrastructure cost.

Each of these research questions aims to illuminate a different aspect of the trade-offs involved in architectural decisions. The methodology section will detail the specific approaches used to answer these questions.

1. 4 Research Objectives

The purpose of this study is to explore the refactorability of the monolithic architecture into microservices, with a strong emphasis on the scalability, performance, and operational

problems related to refactoring. Microservices are a popular choice for software engineering solutions as they provide improved agility, scalability, and maintainability (Hassan, Abdel-Fattah and Mohamed, 2024). However, refactoring an existing system into a microservices architecture is a difficult, multi-challenged process with a variety of challenges, such as service decomposition, synchronous vs. asynchronous communication, and data consistency (Bashtovyi and Fechan, 2024). Therefore, a well-formed framework that can be used for a systematic evaluation of the technical, architectural, and operational factors that govern refactorability will be studied, providing an empirical insight of the context for selecting and assessing modernization alternatives in software architectural decision-making in the context of modern software engineering (Alcides Mora Cruzatty *et al.*, 2024)

To accomplish this, the study begins by examining the performance metrics of monolithic and microservices architectures through controlled performance tests. Utilizing Apache JMeter and Docker, this work replicates real-world requests to assess key performance indicators such as API response time, CPU utilization, and memory consumption. Particular attention is given to how API gateways introduce latency and impact system responsiveness in microservices (Hassan, Abdel-Fattah and Mohamed, 2024). Moreover, this study explores how each architectural style performs under high-load conditions, comparing their resource utilization efficiency and capacity to manage overload scenarios.

Another objective of the study is to examine the differences in scalability and resource allocation between monolithic and microservices architectures. By simulating different traffic loads of client requests, we aim to evaluate how each system responds to growing user demands. In particular, we investigate the bottlenecks of monolithic systems that inhibit scalability, and how microservices can mitigate these bottlenecks by distributing workloads across multiple independent services (Berry *et al.*, 2024).

Beyond the performance and scalability, this paper also explores the organizational and technical challenges to move from a monolithic system to microservices. Through a comprehensive review of academic and peer-reviewed literature, this study identifies key challenges in service composition, data consistency issues, and DevOps challenges with microservices including CI/CD pipelines and infrastructure as Code (IaC) (Lahami et al., 2024). It also analyses the complexity of monitoring tools such as Prometheus and the ELK Stack since these tools are implemented to maintain and keep microservices-based systems under control (ZakerZavardehi, 2024).

Finally, this paper analyses the infrastructure and operational costs of monoliths and microservices to evidence the economic impact of moving to a distributed architecture. The paper takes a practical approach by building and testing a minimal cost-analysis tool to demonstrate how to quantify the long-term trade-offs between monoliths and microservices (Alcides Mora Cruzatty *et al.*, 2024).

By addressing the gap between architectural theory and practice, the findings of this research will provide a holistic framework for assessing the eligibility and challenges of monolithic systems for refactoring to microservices. These findings will be useful to businesses and software architects in making sound decisions to refactor large, tightly coupled systems into modular and scalable microservices while minimizing risk and ensuring optimum operational value (Berry *et al.*, 2024).

1. 5 Contributions

The findings of this study contribute theoretically, practically, and methodologically to the field of software architecture by identifying monolithic architecture refactorability in to microservices. The present study tries to bridge the gap between theory and practice. It provides a structured fashion to assess modularity, scalability, and maintainability for migration of any midsize inventory system to microservice architecture. This study also helps software architects, software engineers, and organizations would benefit from this research in one of methods and steps they can utilize in order to understand whether and how to migrate from a monolithic system to a microservices-based system or not by providing quantitative decision making framework to see software adaptability by exploring its scalability, maintainability, and modularity metrics.

The findings of this study advance software decomposition theory by proposing an innovative estimation method for refactorability in relation to system modularity (Thatikonda and Mudunuri, 2024). Furthermore, this study takes a different perspective compared to the previous research, which tends to recommend the positive aspects of microservices. The findings of this work provide a more nuanced approach by assessing the trade-offs related to scalability, maintainability, and latency throughout the transition process (Nassima, Hanae and Karim, 2024). Thus, this study provides insights into the complexity of refactorability in architectural transformations and contributes to knowledge of reengineering challenges in software restructuring and migration. By extensively examining the structural constraints of monolithic systems, this study provides an in-depth understanding of microservices migration to the scientific field and its implications for the evolution of software and design strategies.

From a practical standpoint, this research provides practical hints on how to improve the maintainability of software systems. The guidelines provided in this research are aligned with industrial best practices for developing scalable applications. A refactorability assessment model is provided which allows software architects to analyze the refactorability of monolithic applications to microservices architecture according to quantified software engineering metrics. Guidelines for developing scalable applications are given in this study, which could allow organizations to better understand the potential challenges involved in microservices development (Kristiyanto *et al.*, 2024). Empirical research presented in this study provides operational evidence for the trade-offs between monolithic and microservices architecture, which could allow various industry stakeholders to make informed choices about what should and should not be modernized and deployed in the cloud (Hassan, Abdel-Fattah and Mohamed, 2024).

Methodologically, this study proposes a refactorability scoring model, which can be used by organizations to determine the migration effort based on quantitative variables such as time-to-architectural-refactor, levels of complexity, and dependency resolution. This refactorability scoring model can be used as an unbiased assessment that can be replicated, adapted, and verified in future studies that focus on software evolution and decomposition techniques. Furthermore, the study performs a benchmarking performance analysis of monolithic vs microservices architectures by comparing latency, response time, and scalability following the controlled performance test using Apache JMeter and Docker (Kristiyanto *et al.*, 2024). Through empirical benchmarking, this study strengthens architectural decision-making in software engineering by providing data-backed rationales to ensure organizations migrate their monolithic architectures to microservices.

Taken as a whole, this study serves as a dual lens of theoretical understanding and practical implementation by providing a structured framework to decide the intricacy and feasibility of microservices migration. It provides software architects, engineers, and organizations with a set of tools to make informed decisions on architectural restructuring, ensuring that legacy systems are systematically evaluated before transitioning them to microservices. By combining academic theory with practical real-world applications, this study builds on current software engineering methodologies and aids the industry in building scalable, maintainable, and performant systems.

1. 6 Thesis Organization

This thesis is organized in six chapters. Each chapter presents context and background relevant for this research. . The dissertation is presented as follows:

The first chapter introduces the research problem and its rationale. It describes the background of this study, the problem statement, research questions, and objectives of the study. This chapter also outlines the main contributions of this research and the ways it pushes the state-of-the-art in the state-of-the-art in software architecture beyond its current status.

The second chapter "Theoretical Background" presents a comprehensive overview of monolithic and microservices architectures by defining specific fundamental principles, along with the advantages and disadvantages of the reviewed architectures. At the end of the chapter, several important concepts such as scalability, performance optimization and system modularisation, are reviewed ensuring that the reader has a thorough understanding of the theoretical underpinnings of the research problem before proceeding to the practical implementation aspects of the case study.

Chapter 3 is the literature review. It presents previous works and studies of monolithic and microservices architectures in terms of performance quantification, scalability problems, and empirical evidence of migration. Based on the gaps in the literature, it justifies the need for an empirical refactorability study.

Chapter 4 explains methodology used in the study. Research approach, design, and data collection methods are presented. This chapter describes the architectural realization of monolithic and microservices models for creating the ground for comparison. Further, benchmarking tools, performance metrics, and evaluation criteria are shown. In addition, chapter describes the challenges of refactoring and strategies used to overcome them.

Chapter 5 is dedicated to results and discussion. It overlays the empirical results in terms of execution throughput, resource consumption and scalability. Section also contains a state-comparative evaluation of the costs of migration from monolithic to microservices architecture. The discussion and interpretation of results attempted to relate findings obtained in this study to those of previous research. The findings of the study have been discussed in terms of their practical implications.

Finally, the sixth chapter summarizes the thesis and presents the conclusions of the research. It recaps the main findings and discusses their theoretical and practical implications. It also acknowledges the limitations of the study and suggests some directions for future work. For example, as related to technological challenges, the thesis suggests studying automated

refactoring techniques. Measuring the maintainability of the microservices and SOA architectures is another area for future work. Alternative architectural styles could be considered. The thesis ends with references that list all the sources that were cited in the text, and appendices that contain some additional materials, such as code snippets, experimental data, and other relevant information.

Chapter 2. Theoretical Background

This section introduces the basic concepts of monolithic and microservices architectures, as well as their differences, strengths, and weaknesses at the structural level. In this article, we are going to discuss this trend of microservices and migrating from monolithic to microservices-based systems which many companies are currently following due to the growing necessity of scalability, maintainability, and high-performance applications in software development nowadays.

This chapter will also introduce some of the important architectural decisions like deployment approaches, infrastructure cost, service orchestration and fault-tolerance approaches. We will have a look into the difficulties during the refactoring of a monolithic setup into microservices with theory and examples. I seek to establish this theoretical basis so that the reader will have the appropriate foundation to appreciate the empirical analysis and implementation discussed in the chapters that follow.

2. 1 Understanding Architectural Transitions: From Monoliths to Microservice

The migration to microservices from monolithic architecture has become a heavily discussed topic amongst software engineering community due to an increased need for scalable, maintainable and high performance applications (Hassan, Abdel-Fattah, and Mohamed, 2024). In this chapter, I present a theoretical background for the work presented in the thesis including a detailed discourse on monolithic and microservices architectures, their pros and cons, scalability, performance metrics, and the challenges they undergo during refactorability.

This aims to set out a knowledge base for the empirical study carried out in the subsequent chapters. This is important context to assess the technical and operational challenges to architectural transitions. It elaborates on the different factors affecting architectural choices like deployment, infrastructure costs, service orchestration, and fault tolerance methods.

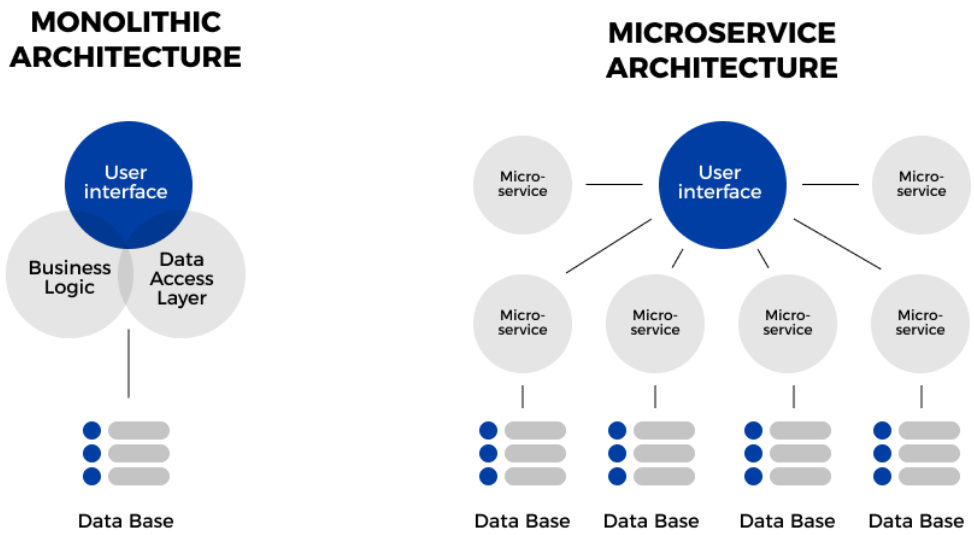

Figure 2.1: Comparison of Monolithic and Microservices Architectures (Karwatka, 2020)

In microservices architecture, typically each service owns a particular database with the pattern Database-per-Service can be seen in Figure 2.1 which improves data isolation, scales independently, and reduces the possibility of cascading failures (Maj, Zielony and Piotrowski, 2024). In this study microservices share a single database. During the transition from monolithic to microservices, these microservices sharing a single database pattern is often used to implement microservices without losing data consistency (Paccha and Velepucha, 2025).

One database schema design ensures robust transactional consistency and ease of data synchronization using the relational database one source of truth model. Nevertheless, it comes with the trade-offs of sacrifices being made on independent scalability of services, single point of failures, and performance bottleneck (Tian *et al.*, 2024). On balance, the shared database schema model used in this project has been optimized to maintain consistency across services whilst fulfilling part of the overall system requirements (Amrutha, Jayalakshmi, and Geetha, 2024).

2. 2 Monolithic Architecture: A Unified but Rigid Approach

Monolithic architecture is the conventional model of software design: all the components of the application are bundled together in a single package. Historically, it used to dominate the enterprise computing landscape, with enterprise applications and data-center backed applications being built as monoliths (Mehta *et al.*, 2024). In a monolithic application, all the code for various components of the app is in a single codebase and not distributed across separate codebases. The unified code is tightly coupled, meaning there is little flexibility to slice down the code into separate components or to separate different components and services from each other (Owen, 2025). There is normally a central database, and all the modules in the monolithic software communicate directly with one another through the shared memory and program space (González and Ortiz, 2024). A cohesive and symmetrical interface is a pro monolithic architecture offer. Nevertheless, scalability and maintainability end up becoming an issue the bigger a project becomes.

Despite the mainstream move towards microservices, monolithic architectures continue to offer certain advantages in particular scenarios. Simpler development: Having a single code base can make it easier for developers to get up to speed on the codebase, and maintain a single software product and environment. Easier debugging and testing: Thanks to operating in a single, shared runtime environment, it can be easier to troubleshoot and debug programs, and to trace calls and errors through the system (Maj, Zielony and Piotrowski, 2024). Simpler deployment: Again, the monolith is packaged as a single deployment unit, so the entire application is compiled and deployed in one go (Muley, 2024). More predictable performance: Operating within a single execution context, rather than communicating between discrete services, enables developers to avoid the risk of inter-service network latency (Mehta *et al.*, 2024). For those reasons, monolithic systems may still be ideally suited to small-to-medium sized applications or situations where the project in question demands an accelerated development process or few development cycles.

However, as applications become larger and more complex, monolithic architectures start to experience serious drawbacks. One of the most important issues is scalability – monolithic applications need to be scaled as a complete unit rather than component-wise, leading to a waste of available resources and to an increase in operating costs (Salaheddin Elgheriani and Ali Salem Ahme, 2022). Another major problem of the monolithic style is the chosen technology stack. Through the use of a single stack, developers are prevented from adopting

different technologies, leading to a stagnating technology stack and making the modernization of legacy applications cumbersome (Santos *et al.*, 2024). Finally, deployment flexibility is another major problem – even the tiniest updates are required to follow the deployment schedule of the rest of the system (Ataei, 2024). This also involves an increase in downtime and risk of deployment (Ataei, 2024), forcing organizations to establish complex strategies for rollbacks in case of a defective deployment (Maj, Zielony and Piotrowski, 2024).

In terms of code maintainability, monolithic systems tend to accrue technical debt as systems grow, because it becomes hard to manage a large code base with many interdependencies. Some difficulties include slow feature development cycles owing to the coordination effort required and higher risk of introducing bugs (Sethi and Panda, 2024). Furthermore, fault isolation is another challenge in monolithic systems. Failures in a single module can affect the entire system. This reduces the reliability of the system and increases the time taken to recover from such a failure (Bashtovyi and Fechan, 2024). Therefore, due to such limitations, organizations often look to migrate to microservices to achieve better scalability, maintainability and operational efficiency.

While monolithic architectures provide simplicity, predictability, and ease of development, they lack scalability, maintainability, and flexibility in bigger and more dynamic systems (Harris, no date). The benefits of better resource utilization, independent scalability, and improved system resilience are the main reasons most organizations shift from monolithic architectures to microservices. The challenges encountered during the migration process despite the benefits make it important for an organisation to make a clear evaluation of their architectural needs before taking any action. The monolithic vs microservices debate emphasizes the importance of knowing what architecture an application needs.

2. 3 Microservice Architecture: A Modular and Scalable Paradigm

A diagrammatic comparison between microservices and monolithic architecture is one of the most important visual aid items (Kamisetty *et al.*, 2023). Microservices vs. monolithic architectures figure would show how monolithic systems consist of a single, tightly coupled unit with numerous components along with a shared database, while microservices architecture is made up of individual and autonomous services, each with its own database, using APIs to communicate (Sun, no date). Such visualization of the distinction would emphasize why microservices bring more flexibility and better fault isolation than monolithic applications which are facing bottlenecks when it comes to scale (El Akhdar, Baidada and Kartit, 2024).

The second key figure would be a workflow diagram, illustrating an architecture of the microservices, supporting microservices interactions in the distributed architecture. This would include an API Gateway routing the client requests to the appropriate microservices like authentication, payments, order management, and so on (Bhatnagar and Mahant, 2024). Each microservice will have their own decentralized database. The flow of communication between the microservices will be explained, which generally tend to be REST APIs, gRPC, and event-driven messaging queues (Ramachandran and Thirumaran, 2024). This would help exemplify how microservices are communicating asynchronously but at the same time achieving data consistency and scalability.

A table detailing potential advantages and disadvantages of microservices would provide a clear comparison at a quick glance. The advantages of microservices are their independent scaling, failure isolation, and choice of technology and the disadvantages are the increased

communication overhead, data consistency challenges, and security risks. (Ali, 2024) Such a table would present a clearly written comparison between the two methods. Microservices offer greater modularity and efficiency but also dense complexity in their deployment and monitoring (Salunkhe *et al.*, 2024). A table provides clear and easy access to visually seeing the comparison.

Another recommended figure is a scalability and fault isolation diagram, showing how microservices independently scale and isolate failures of a single service from crashing the whole application (Shao *et al.*, 2024). Specifically, the diagram should have a load balancer distributing incoming traffic to multiple instances of a microservice and explicitly label that horizontal scalability, scaling by adding more instances to a microservice, is favoured over vertical scalability, scaling by increasing the resources available to each instance (Shao *et al.*, 2024). Likewise, the figure should also illustrate how one failing service can independently restart without failing other microservices, providing fault tolerance, indeed, this is a major benefit of microservices over monolithic system, where a single failed service can cause an application-wide outage (Curnicov, 2025).

Finally, a case study diagram on microservices adoption in the industry would provide a real-world perspective on their impact. This visual could include logos or representations of major companies such as Netflix, Amazon, Uber and Spotify, highlighting how each utilizes microservices to achieve scalability, resilience, and rapid deployment (El Akhdar, Baidada and Kartit, 2024). For instance, Netflix leverages microservices to support millions of concurrent video streams, while Amazon applies microservices for dynamic inventory management and personalized recommendations (Bhatnagar and Mahant, 2024). By visualizing these industry implementations, the diagram would underscore how microservices enable innovation in large-scale applications.

2. 4 Scaling in Software Engineering: Vertical vs. Horizontal Scaling

Scalability in a system is its capability to handle a growing amount of workload from an end-to-end system point of view. Vertical (or upward) scalability is adding more resources (CPU, RAM) to the existing server to increase performance (Gandhi and Vashishtha, 2025). This is the most popular approach used in monolithic architectures. Horizontal (or outward) scalability is deploying multiple copies of a service and delegating traffic to each with the help of a load balancer (Dragoni *et al.*, 2017), which is one of the main features of microservices. Unlike monoliths, which must scale vertically and only up to the hardware limit of the physical host on which they run, microservices support horizontal scaling by default, enabling the system to serve large-scale concurrent requests efficiently (Fowler and Beck, 2019). A scalable system uses a variety of techniques such as load balancing, caching, and database partitioning to enhance its overall performance and to handle the workload efficiently (Chen *et al.*, 2024).

2. 5 Performance and Resource Utilization

In Performance Evaluation, we have Key Metrics Latency. This term refers to the time taken for a system to respond to a request. It is affected by inter-service communication and network overhead. Throughput Indicates system efficiency — For each second of continuous operation that involves such requests to process how many messages does it get roundly turned out Around 200? CPU and Memory Usage The resource consumption required for infrastructure to be kept at performance levels Maintain high trafficThere are a few differences between Monolithic and microservice architectures. But, the dynamic scaling made possible

under these conditions with microservices challenges how appropriate one will indeed prove to your attitude rests entirely on disposition or preference.

2. 6 Refactorability in Software Systems: Transitioning to Microservices

The ease with which one piece of code can be changed is called refactorability. Motivations for refactorability involve: Code modularity: a well-structured monolith can more easily be refactored into microservices. Dependency management: Pulling apart tightly coupled components Database design: Moving from one big database to many little databases means consideration is needed as to its layout and how it can be made fault-tolerant. Poorly designed monolithic applications throw up formidable challenges to refactoring, necessitating incremental migration strategies and the use of well-defined decomposition processes.

2. 7 Theoretical Summary: Foundations for Implementation

This chapter explained the basics of monolithic and microservices designs. It also discussed how to scale them, measure their performance, and the challenges of making changes to them. These ideas lay the foundation for the experiments discussed in the following chapters. In these experiments, a large system will be changed into smaller services to see how it affects speed, resource use, and ease of maintenance.

Chapter 3. Literature Review

3. 1 Overview of Monolithic and Microservices Architectures

The growth of software architecture has seen a transition from centralized monolithic systems to more decentralized microservices. Monolithic architectures have been the standard followed architecture, where the entire functionality of the application (including user interface, business logic, and the data access layer) is tightly coupled, resulting in a single deployable unit. Thereby creating an efficient architecture, when building small to medium-sized applications that are less likely to scale. However, since all software components co-exist in a tightly coupled state in Monolithic systems, some issues arise. The tightly coupled state creates inflexible deployments, a high dependency between the business logic and components, and less scalability. Since all the components of a monolithic system are tightly coupled when deploying let's say a single module in the system, the entire application has to be deployed. Scaling Monolithic architecture requires replicating the whole system and not the modules of the system that requires scaling. As a result, Sharding a monolithic system resource is usually inefficient, if one module or application section experiences a sudden spike in traffic, it will still be limited by the system resources (Kassetty and Chippagiri, 2025).

A Monolithic system is difficult to maintain, with a tightly coupled state, a change at one end (module) causes an effect on the system as a whole. While developing new functionality on an existing monolithic system, the entire system must be first set up before development begins which is inefficient and time-wasting, as it could be avoided by developing new functionality as a loosely coupled service to be integrated on successful development and testing. Development is quite difficult since all components need to be constantly built and redeployed, to ensure the system remains functional. Additionally, already-built applications with a large customer base are faced with the challenge of downtime. Once a change occurs in a monolithic deployment, you have to redeploy the entire application which causes downtime during the system update. This approach is unlike microservices' continuous development which seeks to isolate sections of the application during the development cycle and integrate after proper testing.

In response to these disadvantages, the microservices architecture has become a popular alternative, especially for large-scale, cloud-native applications. Microservices break down applications into a collection of small, independently deployable services that communicate via APIs. Designing each microservice around a specific business capability it allows for higher modularity and technical flexibility. Microservices can independently scale specific components rather than monolithic systems that inherently need all components, saving system resources and improving fault isolation. This means that any outages in one service don't necessarily have a negative impact on the whole system, increasing the resilience and reliability of the system. Furthermore, microservices enable faster development cycles since teams can concurrently work on different services without negatively impacting the entire application. Nevertheless, despite these aforementioned benefits, microservices do not come without additional complexities, including increased network latency due to inter-service communication, challenges of maintaining data consistency across distributed services, and the need for advanced monitoring and orchestration tools (Wang, 2024).

The transition from monolithic to microservices architecture is not straightforward. It poses technical and organizational challenges, such as decomposing tightly coupled components, dealing with distributed data, and ensuring reliable inter-service communication. One of the

most common migration tactics is the Strangler Fig Pattern, where microservices are introduced gradually alongside the monolithic system. This allows teams to slowly adopt microservices without having to do a complete system rewrite. Even so, the migration process is nontrivial: it can involve large efforts in the service decomposition process, API management, and infrastructure provisioning. Operational overhead increases due to the large number of independent services that teams have to manage, meaning there is a greater need for automated deployment pipelines, container orchestration (such as Kubernetes), and service discovery mechanisms. Security is another issue: teams must be cautious to secure the large attack surface introduced with the higher number of microservices, and strong network security mechanisms must be put in place to protect inter-service communication. Furthermore, the network latency introduced by S2S (service-to-service) calls and duplication of code (such as libraries, functions, types, and models used across services) are also two more challenges.

Although challenging, research has shown that once successfully transitioned, systems built using microservices see significant improvements in scalability, agility, and fault tolerance, especially in distributed cloud environments. Research also shows that microservices architectures are better suited to handle heavy traffic loads than monolithic applications. By allowing horizontal scaling, microservices can fulfill higher demands as only the necessary services need to be scaled instead of the entire application. Organizations such as Netflix and Amazon have presented the success of microservices by taking advantage of independent service scaling to improve system responsiveness and reduce operational costs in the long term (Chen *et al.*, 2024). However, microservices do not come without sacrifices, and organizations must determine whether the complexity of microservices is worth the scalability and maintainability benefits. Monolithic architectures may still be suitable for simple applications with relatively straightforward workflows and a limited need for scalability.

Table 3.1:Comparison of Monolithic and Microservices Architectures(Powell and Smalley, 2024)

Feature	**Monolithic Architecture**	**Microservice Architecture**
Scalability	Vertical scaling (entire system)	Horizontal scaling (independent services)
Deployment	Whole system redeployed	Independent service deployment
Technology Stack	Unified stack	Multiple stacks allowed
Fault Tolerance	Single point of failure	Failures isolated per service
Development Speed	Slower due to dependencies	Faster due to team autonomy
Operational Complexity	Lower	Higher (requires orchestration)

A summary of direct comparison between the monolithic and microservices architectures is provided in Table 3.1. The main advantages and disadvantages of these architectures are related to scalability, deployment flexibility, resilience to faults, and operational complexity. The decision between using monolithic or microservices architectures depends on a number of factors including the scale-out requirements of an organization and the availability of resources for development and operations. As summarised in Table 3.1, monolithic systems are preferred

where smaller applications with a lower number of application workflows need integration while microservices are suitable in highly scalable distributed systems where sub-services of a system need to scale independently from each other.
As software systems continue to evolve, the debate between monolithic and microservices architectures remains relevant. While monolithic architectures offer simplicity and lower initial costs, microservices provide greater flexibility, scalability, and fault tolerance, making them more suitable for large, complex applications. The decision to migrate from monolithic to microservices should be guided by an organization's specific needs, technical capabilities, and long-term scalability goals. Ongoing research and development continue to address challenges associated with microservices adoption, particularly with automated refactoring, distributed data management, and inter-service communication. As more organizations transition to microservices, best practices and strategies for architectural transformation will continue to develop, ensuring informed decision-making in selecting the most efficient architecture for an organization's operational and scalability requirements.

3. 2 Refactorability: Definition and Importance

Refactorability is a classic quality that indicates how easily a system can be restructured and optimised without changing its original behaviour. Refactorability is a fundamental concept in software engineering. This reduces the complexity of migration, handles technical debt and helps in long-term maintainability. It helps to make gradual enhancements and seamless decomposition of any tightly coupled code with migration easier. That is, refactorability is the extent to which existing code or a system is amendable to refactoring. module boundaries and least interdependencies in monolithic systems help in easier refactor than those with tightly coupled components containing interlinked code and these components need to be rewritten almost completely (Sulkava 2023). This means rest that how refactorable a monolithic system is, affect how much well organisations could able to move toward microservices as apart from without flexibility architecture there then exist a a lot of migration challenges within place. Some of them are data consistency problems, performance issues due to inefficient service decomposition and low communication overhead between multiple services (Rathod, Joseph and Martin, 2023).

Even beyond migrating software when we look at the importance of refactorability. It applies in turn to scalability, system resilience, and development agility. A refactored system, for instance, can help you to adopt domain-driven design (DDD) and therefore extract microservices out strategically plus guarantee that services stayed loosely coupled and independently deployable. Decreased refactorability may motivate organizations to push forward with an anti-pattern they call monolithic microservices that also introduces performance bottlenecks and additional cost for compensating them (Alongi et al., 2022). Even further, when adapting microservices, the database refactoring is crucial: a monolithic database usually requires decomposing the schema or even sharing data with event-driven deviant microservices to be autonomous. Therefore, it has been discovered through research that the automated refactoring tools alongside the code maintainability metrics (in other words the cyclomatic code complexity, dependency analysis etc.) could improve the refactorability evaluation and the evolution of software (Abid et al., 2020).

*Table 3.2: Factors Affecting Refactorability (*Monolithic vs Microservices - Difference Between Software Development Architectures- AWS*, 2024)*

Factor	Impact on Refactorability	Challenges in Migration

Code Modularity	Higher modularity: easier migration	Poorly structured code requires extensive refactoring
Coupling & Dependencies	Loosely coupled components are easier to extract	High coupling makes decomposition difficult
Database Structure	Well-structured databases enable smoother migration	Monolithic databases require schema changes or decomposition
Service Boundaries	Clearly defined service boundaries help in microservices extraction	Undefined boundaries lead to service overlap and redundancy
Scalability & Performance	Optimized systems transition more efficiently	Performance bottlenecks may arise post-migration

The knowledge of these factors is important for assessing the readiness of the system for moving to microservices. Automated refactoring tools, static code analysis, dependency graphs and refactorability scoring models can provide information on how easily a monolithic system can be decomposed. Table above (Table 3.2) provides a summary of the primary factors that affect refactorability, their consequences and challenges during migration.The knowledge of these factors is important for assessing the readiness of the system for moving to microservices. Automated refactoring tools, static code analysis, dependency graphs and refactorability scoring models can provide information on how easily a monolithic system can be decomposed.

To mitigate migration difficulties, a number of best practices have been promoted such as modularizing the monolith, embedding an API gateway, and migrating incrementally (e.g., Strangler Fig Pattern) (AlOmar, Mkaouer and Ouni, 2024). It allows monolithic elements to be replaced with microservices iteratively, without losing the consistency of the system. Essentially, refactorability scoring is evolving into a new area of research to measure the modularity, separation of concern, and also code complexity of the software systems to evaluate the effort required for their migration to microservices. Over the course of this evolution, refactorability is one of the key driving factors of software systems that remain maintainable, scalable, and efficient. This means organizations should invest in refactorability so that they avoid future bottlenecks whenever business requirements change or when new technologies come into play.

3. 3 Challenges in Transitioning from Monolith to Microservices

There are various challenges organizations face when transitioning from a monolithic architecture to microservices. These challenges range from technical to operational and organisational. Despite having many benefits associated with microservices such as scalability, maintainability and deployment, organisations face many struggles such as, service

decomposition, data management, inter-service communication, infrastructure complexity, DevOps challenges and cultural shift (Kamisetty *et al.*, 2023).

One of the major obstacles lies in the service decomposition complexity of microservices is how to identify service boundaries (i.e. the architectural constraint) while decomposing a monolithic system into smaller, autonomous units. It is rather difficult to establish hidden and implicit logical boundaries between services because many monolithic legacy systems are composed of highly coupled components that interact with each other through intertwined and shared business logic and database schemas. And Data Driven Design (DDD) is often used to define microservices based on limited contexts or business capabilities. Inappropriate decomposition, however, will lead to excessive fragmentation, resulting in overhead associated with operating too many services. Moreover, in order to refactor a monolithic application into a loosely coupled microservices system, it will be necessary to introduce significant changes to the code structure, leading to accumulated technical debt and migration risks.

Another key challenge is data management and consistency. Monolithic applications often operate with a single centralized database, whereas microservice-based applications feature a distributed data architecture where each microservice possesses its own database. This transition necessitates a fundamental paradigm shift in data management, resulting in transaction consistency, data replication, and synchronization (Samant, 2024). Traditional database transactions using ACID principles must be superseded by Event Driven Architecture (EDA) or distributed transaction protocols such as the Saga Pattern to maintain eventual consistency. However, if not correctly implemented, they can cause data inconsistency, race conditions, and performance impact.

Inter-service communication introduces its own complexity. While monolithic applications call functions within the same process, microservices must communicate with each other using APIs, message queues, or event-driven messaging systems. While REST APIs, gRPC, and message brokers (Kafka, RabbitMQ) solve this problem, they also come with their own implementation challenges, introducing problems of network latency, fault tolerance, and dealing with API versioning (Garimilla, 2024). More issues are introduced with increased inter-service communication and number of microservices, such as the risk of cascading failures, resulting in the application for circuit breakers and retry mechanisms to control it to avoid downtime of the whole system.

Infrastructure complexity is an additional challenge for microservices adoption. Critical monolithic applications mostly run on a single-server or VM based environment. However, critical microservices require containerization (Docker), orchestration (Kubernetes) and service discovery (Mehta *et al.*, 2024). Hosting applications across multiple nodes increases resource provisioning complexity. As a result, service mesh technologies that manage load balancing, security, and observability, such as Istio, are difficult to adopt. Debugging microservices requests without proper monitoring and logging solutions is highly complex, thus taking into consideration using tools such as Prometheus, Grafana, and the ELK stack is desirable.

Continuous integration and delivery (CI/CD) requirements also increases deployment and DevOps burdens. Monoliths must be deployed as a single unit and typically require a minimal CI/CD effort, whereas microservices require independent deployment of services and hence a robust CI/CD pipeline, automated testing and rollback capabilities (Sethi and Panda, 2024). Blue–Green deployments, Canary releases and Feature Flags become essential to minimize

down-time and mitigate deployment failures. Organizations have to use infrastructure-as-code (IaC) tools like Terraform and Ansible to ensure that their microservices environments are scalable and easily reproducible.
Beyond the technical difficulties, the adoption of microservices also requires organizational and cultural shifts. Monolithic teams tend to be centralized, whereas microservices follow a decentralized government (i.e. cross-function teams and DevOps). Organizations should support training and restructuring of teams in order to enable developers, testers and operations to take ownership of microservices by understanding distributed systems and distributed architectures. Resistance to change, the unknowns of unfamiliar microservices, and unclear governance models can impede migration. Because of this, many businesses settle for hybrid architectures, keeping some legacy monolithic modules which can become dependencies that are difficult to manage.

Table 3.3: Challenges in Transitioning from Monolith to Microservices(Al-Debagy and Martinek, 2018)

Challenge	**Description**	**Impact on Transition**
Service Decomposition Complexity	Identifying appropriate microservices boundaries	Risk of over-fragmentation or under-decomposition
Data Management & Consistency	Moving from a single database to distributed data stores	Increased complexity in ensuring data consistency & transactions
Inter-Service Communication	APIs, messaging queues, and event-driven communication	Higher latency, increased failure points, need for circuit breakers
Infrastructure Complexity	Need for containerization (Docker, Kubernetes) and monitoring tools	High learning curve, increased resource overhead
Deployment & DevOps Challenges	Shift to CI/CD, automated testing, and infrastructure as code (IaC)	Need for new tooling and deployment automation strategies
Organizational & Cultural Shift	Moving from centralized to decentralized teams with DevOps	Requires restructuring, new development workflows, and upskilling

Addressing these challenges requires a well-defined migration strategy, starting with modularizing the monolith, adopting API gateways and service meshes, implementing observability tools, and gradually transitioning components to microservices. A hybrid approach, such as the Strangler Fig Pattern, can help organizations incrementally migrate functionalities while keeping the legacy monolith operational. Table 3.3 summarizes the major challenges in transitioning to microservices.

3. 4 Existing Transition Strategies and Best Practices

As systems transition from a monolithic architecture to microservices architecture, there is a need to have a concrete migration strategy to ensure system stability, scalability, and efficient operation of the migrated application. It is more than a technical exercise but includes architectural, infrastructural, and organizational changes (Baumgartner, 2022). There are different migration strategies available for decomposition from incremental decomposition

strategies like the Stangler Fig Pattern to complete rewriting strategies (Big Bang Migration). Domain-Driven Design (DDD), event-driven architectures, databases transition strategies, and CI/CD automation are some of the best practices required in this phase to ensure a successful and risk-mitigated migration.

A progressive incremental migration is often preferred to a Big Bang migration as it reduces the operational risks and gives teams the chance to exercise the microservices before full deployment. In a incremental migration, monolithic applications are slowly completed into microservices. It allows organisations to carry on their usual business while optimising their systems and focusing on their performance. This also means that services can be refactored and individually scaled without damaging the system as a whole (Santos, 2018). A Big Bang migration entails rewriting the entire application as microservices. While it might take less time to complete a Big Bang migration, it poses serious risk to a business as deployment difficulties might only be detected once the application is complete. Such failures might include inter-service failures, scalability issues and database bottleneck Depending on particular business model, companies should choose the way of migration to microservices that compatible to their current infrastructure and operational requirements .

The Strangler Fig Pattern is a popular incremental migration approach used to achieve a smoother monolithic to microservices transition. Strangler fig pattern creates a new microservice along with the existing monolithic functionality, and gradually replaces the monolithic functionality with the new microservice. The strangler fig pattern eventually replaces all the monolithic functionality until the newly developed microservices entirely replaces the monolith. This pattern is the most commonly used pattern for large-scale enterprise applications that should be up and running throughout the migration process. Strangler Fig Pattern reduces technical debt, and the risk of a major disruption of the existing system during migration. Teams can validate the behaviour of microservices before deprecating the monolithic functionality.

A core tool in migrating to microservices is Domain-Driven Design (DDD), which aids in the discovery of explicit microservice boundaries. Many monolithic applications suffer from tightly coupled components, further complicating decomposition. DDD builds the microservice architecture on business domains, giving autonomy, high cohesion and loose coupling to the resulting architecture (Manchana, 2021). Furthermore, DDD uses concepts such as Bounded Contexts and Aggregates to ensure that microservices capture a specific business function and do not have inappropriate dependencies. Companies using DDD principles to build microservice architectures can avoid the over-fragmentation seen in some microservice implementations and allow their microservices to remain isolated and manageable.

Another handy approach in microservices transition is adopting an event-driven architecture (EDA) to enable asynchronous communication among services (Goniwada, 2022). While the method of communication between services in monolithic architectures is typically synchronous and dependent on API calls, EDA enables loose coupling between services while providing real-time data exchange (Goniwada, 2022). Since services do not rely on each other, it aids in enhancing fault tolerance, scalability, and responsiveness. Examples of event brokers used in event-driven architectures include Apache Kafka and RabbitMQ. They serve as intermediaries between services and enable asynchronous communication via events. This benefits distributed systems where services must run independently of each other. Event sourcing is a technique where all state changes are stored as events so that services can recollect

previous states. By using event brokers like Apache Kafka, EDA improves services' auditability and reliability (Goniwada, 2022).

One of the difficulties of microservices migration is database management. The typical design is database-per-service, meaning each microservice owns its independent data store. But in most real-life cases (this project included), organisations still adopt the shared-database model due to transaction consistency concerns and operational constraints (Khakame, 2016). While shared-database simplifies data integrity and transaction handling, it makes services strongly coupled to each other, resulting in low autonomy and scalability. To address these concerns, database-transition strategies are introduced, which allow organisations to adopt in event-driven data synchronisation, CQRS (Command Query Responsibility Segregation) and Saga Patterns to handle distributed transaction management while achieving consistency of data across microservices.

Table 3.4: Comparison of Migration Approaches (AlOmar, Mkaouer and Ouni, 2024)

Migration Approach	**Advantages**	**Challenges**	**Best Use Case**
Incremental (Strangler Fig Pattern)	Minimizes risk, allows gradual testing and rollout	Requires hybrid system management during migration	Large legacy applications needing continuous availability
Big Bang Migration	Faster transition, full adoption of microservices	High risk of failure, potential system-wide downtime	Small applications or startups
Hybrid Approach	Allows **some monolithic components** to persist temporarily	Complex system orchestration, requires careful planning	**Organizations with** partial microservices adoption

Although this project does not feature a complete CI/CD pipeline, it is nonetheless an essential best practice for deploying scalable microservices. The process focuses on automatically building, testing, and deploying microservices, ensuring that they are released quickly and consistently. In monolithic deployments, deployment is fairly rudimentary as the entire monolithic system is deployed as a single unit. Unlike monolithic deployments, microservices must deploy independently, as one specific service should not be dependent on another service. Various CI/CD pipeline tools such as Jenkins, GitHub Actions, and GitLab CI/CD facilitate the deployment of microservices by automating the rollback mechanism, canary release, and high availability strategy (Baumgartner, 2022). As a result, CI/CD enhances the resilience of the system, minimising downtime and speeding up development cycles which is critical for modern cloud-native architectures.

Table 3.5: Best Practices for Microservices Migration

Category	**Best Practice**
Service Design	Use DDD & bounded contexts to define clear microservice bounds
Data Management	Implement Saga Pattern for managing distributed transactions.

Deployment	Automate CI/CD pipelines to enable frequent and stable releases.
Observability	Use centralized logging & distributed tracing to monitor microservices.

To ensure a successful transition, organizations must adopt best practices tailored to their specific use cases. Table 3.4 provides a comparative overview of migration approaches, while Table 3.5 outlines best practices for a structured and risk-minimized transition.

3. 5 Discussion on Literature Review

The transition from monolithic architectures to microservices represents a paradigm shift in software engineering, as it provides a solution to the challenge of scalability, maintainability, and operability that arises from a tightly coupled system design. The literature review about this topic examined theoretical perspectives, challenges, migration strategies, and best practices, which address not only the strengths of microservices architectures, but also the entanglement of their adoption. Monolithic architectures provide simplicity, centralised management, and low operational overhead. However, they cannot provide the flexibility and scalability that modern cloud-native applications require. Microservices architectures, on the other hand, offer horizontal scalability, independent deployability, and fault isolation, providing freedom (and chaos) that modern large-scale distributed applications require. However, these benefits come with a cost of complexity in service decomposition, data management across independent services, inter-services communication, and infrastructure orchestration (Baumgartner, 2022).

A crucial driver of migration success relates to the monolithic system being refactorable, meaning the extent to which modular components of code can be extracted and converted into microservices. Code modularity, code coupling, database design and service boundaries are major factors that determine the complexity of the migration process. Research has shown that using automated refactoring tools, code maintainability metrics and structured decomposition techniques improve the monolith system's readiness for microservices adoption (Sulkava, 2023). Additionally, resolving technical debt prior to the migration effort reduces risk for performance bottlenecks, redundant services and high operational overhead.

The literature review highlights the main issues encountered by organizations when migrating legacy applications to micro-services which includes service decomposition and its complexity, data consistency, inter-service communication overhead, infrastructure complexity, and DevOps automation requirements (Kamisetty *et al.*, 2023). In addition to decoupling components from the monolith, organizations shall take due care necessary in defining service boundaries such that micro-services do not tend to exceed beyond manageable limits by over-decomposing and fall into the micro-service anarchy trap (Samant, 2024). To achieve appropriate granularity and enclosing of business logic, organizations relying on micro-services tend to adopt Domain-Driven Design (DDD). Without a proper governing mechanism, the complexity tends to increase and eventually end up with a non-functional system. Therefore, within any system employing micro-service architecture, finding a perfect balance between DDD and the organization's capability to handle micro-services plays an important role in building robust software systems. On the other hand, for an enterprise system deploying a micro-service architecture, it becomes essential to ensure distributed data and

transaction consistency between the micro-services. Now, needless to say, consistency can be achieved by introducing dependencies between the microservices but that will go against the concept of having loosely-coupled or independent deployable services. Therefore within a distributed system, managing the scale of the application while preserving consistency and availability to its client(s) becomes a prime factor for successfully adopting a micro-services architecture. Organizations adopting micro-services started employing patterns like event-driven architecture (EDA), CQRS, and the Saga Pattern within their systems to ensure data and transaction consistency. Each of these patterns presents a different view on how data and consistency can be achieved at scale without making any compromises on reliability. For an enterprise system deploying a micro-services architecture, the inter-service communication overhead can become a massive pain point for the organizations. As the volume of micro-services increases, then so does the network traffic and the number of inter-service requests. Within a distributed system, having robust infrastructure that can handle concurrent requests at such a scale becomes an essential factor for successfully adopting a micro-service architecture. Due to the need of having robust APIs to serve their client requests and process inter-service requests, organizations tend to adopt more APIs within their implementation. This in turn puts more pressure on developing a robust API gateway and API management services (Garimilla, 2024). Besides evolving the decentralized architecture and hosting services using containerized infrastructure with containers like Docker, Kubernetes, etc., event-driven systems must possess proper service discovery mechanisms to handle dynamic registration/deregistration of micro-services and process concurrent requests at scale. Though organizations can successfully decouple services from the monolith, it becomes an overhead for organizations to then manage the deployment, operation, and monitoring of such large infrastructure that, as a whole, forms the distributed system. It evolves an organization's capability to manage such large decentralized systems by introducing automation in delivering infrastructure-as-code CI/CD pipelines, and implementing DevOps tools to closely monitor and track infrastructure and their operations.

Various research papers in this field address these challenges and list incremental migration approaches, such as Strangler Fig Pattern, which enables organizations incrementally move towards microservices, without disturbing the functioning of a legacy system. Big organizations like Netflix and Amazon have successfully updated their monolith system into microservices using this approach, minimizing risk in regular operations. Big bang migrations on the other hand involves replacing the old system completely, which exposes to the high risks of failure, downtime of systems and scalability issues. Automation of CI/CD pipelines, observability tools (Prometheus, Grafana, ELK Stack) and infrastructure-as-code are some of the best practices which have been established over time to improve deployment as well as improve system resilience and availability.

Although microservices offer concrete benefits in terms of faster development velocities, unified scalability, and enhanced maintainability, they also pose a lot of complexity that requires thorough preparedness, robust architectural setup, and organizational alignment. Microservices may not be appropriate for all applications, with organizations needing to evaluate whether the overhead of adopting microservices will add business value to the organization while satisfying scalability and performance requirements for the system. The consensus among literature suggests that while monolithic architectures are still a valid choice for smaller applications with limited scaling potential, microservices architectures are better suited for large-scale, distributed applications. Further research continues to explore automated refactoring, AI-driven service decomposition, and greater security improvements to smoothen microservices transition; ultimately, businesses should determine how migration could be

accomplished based on an organization's systems, technical knowledge and business needs in the longer run.

Chapter 4. Methodology

4. 1 Overview

The change from monolith to microservices design has actually come to be a main emphasis in contemporary software program design driven by the requirement for scalability, modularity, and also maintainability in dispersed systems (Barzotto and Farias, 2022). Standard monolith architecture while simpler to establish, has a tendency to end up being inflexible as well as tough to range as they expand causing difficulties in implementation, upkeep, and also group efficiency. Microservices architecture style on the other hand allows a decentralized and also service-oriented strategy where independent solutions interact by means of lightweight methods enabling better versatility, boosted fault tolerance, and a lot more reliable source application (Chaieb, Sellami and Saied, 2023). Nonetheless, regardless of these benefits, the procedure of refactoring monolith architecture right into microservices architecture stays extremely intricate, calling for methodical disintegration approaches as well as detailed assessment techniques to ensure efficiency enhancements (Seedat *et al.*, 2023).

This research study intends to review the refactorability of transitioning monolith systems right into microservice systems by examining code intricacy scalability, latency plus system maintainability. The technique utilized in this research study includes regulated movement experiments where a single system is significantly taken apart right into microservices adhering to the finest methods in component as well as solution orchestration (Nitin *et al.*, 2023). Different building patterns such as Domain-Driven Design (DDD) Strangler Fig Pattern and also Service Decomposition based upon service capacities are discovered to evaluate their efficiency in microservices fostering.

To accomplish an extensive contrast both qualitative as well as measurable efficiency metrics are made use of. The research takes a look at code intricacy (Maintainability Index, Cyclomatic Complexity) system latency, demand handling capability, and also mistake resistance. Empirical examinations are carried out making use of benchmarking devices such as JMeter for lots screening as well as profiling structures for examining dispersed purchase expenses. By implementing tension examinations plus regulated scalability experiments this study examines exactly how microservices styles affect action time, straight scaling effectiveness as well as general system durability (Chaieb, Sellami and Saied, 2023).

A considerable emphasis is placed on disintegration methods, making sure that solution limits are well specified as well as do not present unneeded interaction expenses that can adversely influence efficiency. The research study additionally attends to usual movement obstacles such as data source refactoring solution exploration, API Gateway assimilation, and also reliance administration (Seedat *et al.*, 2023)

Additionally this research study follows a speculative method, using a real-life study to determine the effect of microservices movement on software program refactorability. The outcomes add to a structured decision-making structure for companies thinking about the monolith-to-microservices movement giving understandings right into when as well as just how such a change must be taken on. By developing sensible standards for solution disintegration, efficiency adjusting plus building optimization, this research study uses a thorough point of view on modern-day software application scalability obstacles plus services.

4. 2 Proposed Framework/Model/Technique

4.2.1 Architectural Design of Monolith vs. Microservices (Strangler Fig Pattern)

A monolithic architecture is made up of a single application. All functionality is managed within a single application, which means that user interface, business logic and database access code are held together (Barzotto and Farias, 2022). When an application is first developed, a monolithic approach is simpler to develop, test and deploy. However, monolithic systems create serious issues as the application grows. A monolithic application is tightly coupled, change deployments become increasingly inflexible as the code base grows. Development teams are imposing a slowdown in the release cycles and require increasingly complicated deployment processes (Chaieb, Sellami and Saied, 2023). A monolithic application imposes a linear scalability constraint, there is no smoothing of workloads as the modules are dependent on a shared codebase, lead all CI/CD pain points to be exaggerated since a small code change in a single module necessitates building and deploying again the entire monolithic system rather than being able to scale application components independently of one another. Moreover, as more developers work on the same codebase, merge conflicts, regression issues, and technology constraints become common issues and prevent experimentation with new technologies, which can impede innovation and leave organisations competing in an ever-changing landscape of new framework implementations (Chaieb, Sellami and Saied, 2023).

Modern software systems are moving towards microservices-based architectures to achieve scalability, modularity, and independent deployment. Rewriting the whole monolithic system in one go is infeasible and a high-risk endeavour for most enterprises. In this study, the monolith system was integrated into microservices progressively using the Strangler Fig Pattern. The Strangler Fig Pattern is a migration strategy that allows for incremental refactoring from monolith components to extract independent microservices. This gradual approach enables developers to refactor the system without affecting system functionality (Seedat *et al.*, 2023). The pattern derives its name from the growth habit of strangler fig trees: as strangler fig trees grow around their host, the monolith and microservices live alongside each other, allowing the system to grow (in functionality) and die (deprecated functionality) incrementally rather than drastically in a big bang cutover.

The migration process can be broken down into the following three phases. The monolithic components are analyzed and prioritized by business functionality and system dependencies for extraction. User management, product catalog, order processing and payments were identified as some of the most critical modules to extract in the first phase. Microservices are built alongside the monolithic parts, running in parallel to them and operating on a single database. All traffic is moved to a router, which starts to redirect it to microservices where possible, while the monolithic application continues to serve legacy traffic. As stability and performance are further validated, more and more traffic is rerouted from the monolithic system until all prioritized parts are entirely in the hands of microservices. The monolith is retired at the end of the process.

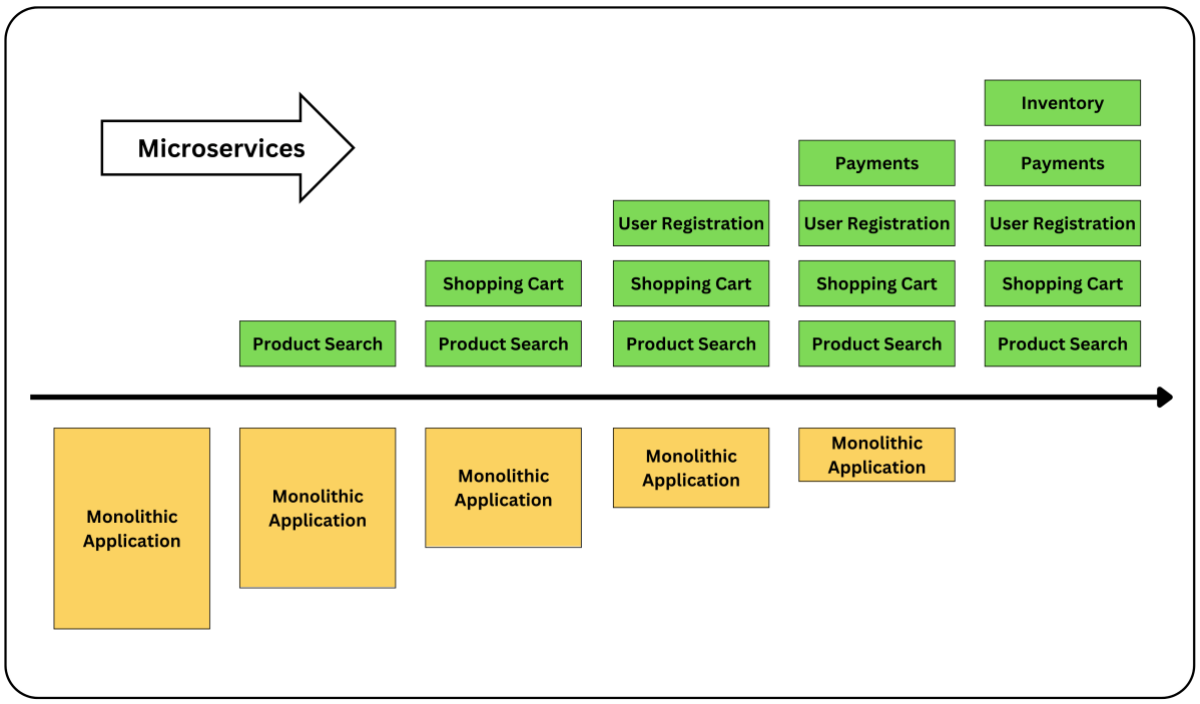

*Figure 4.1: Monolithic vs. Microservices Transition (Source: (*Monoliths to Microservices using the Strangler Pattern*, no date))*

The monolithic system proceeds to be replaced with microservices as the system remains stable as shown in Figure 4.1. The migration journey comprises three key phases. Firstly, monolithic components are analyzed and prioritized based on business functionality, and system dependencies are extracted. Key modules such as 1. user management 2. product catalog 3. order processing 4. payments are identified as initial candidates to be migrated. Secondly, microservices are implemented in conjunction with the Monolith and run parallel with a common database. By a routing mechanism, traffic is routed to microservices wherever possible, while the monolithic appstem are proven, the traffic is gradually moved away from the monolithic system till the key components are replaced, and the monolithic system is decommissioned.

Once microservices are in place, they only talk to each other through RESTful APIs and not through in-memory function calls as in the monolithic architecture. The system doesn't use asynchronous messaging tools such as Kafka or RabbitMQ or gRPC, so the communication model is synchronous and relatively simple. Each microservice implements one business function and exposes a set of well-defined HTTP endpoints such that other services can call them. The following services have been refactored out of the monolithic system:

- API-Gateway: The service required to forward all requests coming to the microservice to other services.
- User Service: Authentication, registration and user profile management.
- Product Service: Product details, inventory and product categorisation management.
- Order Service: Order management and tracking management.
- Cart Service: The service where basket information and related transactions are kept.

Sharing a common relational database system, this pattern enables all microservices to connect to a single database instance; concurrency is ensured without event-driven data replication. Nonetheless, with higher service coupling, as well as potential bottlenecks for concurrent

queries to a central database, there are trade-offs. Database index, connection pool, and query optimization are some of the applied solutions for these challenges.

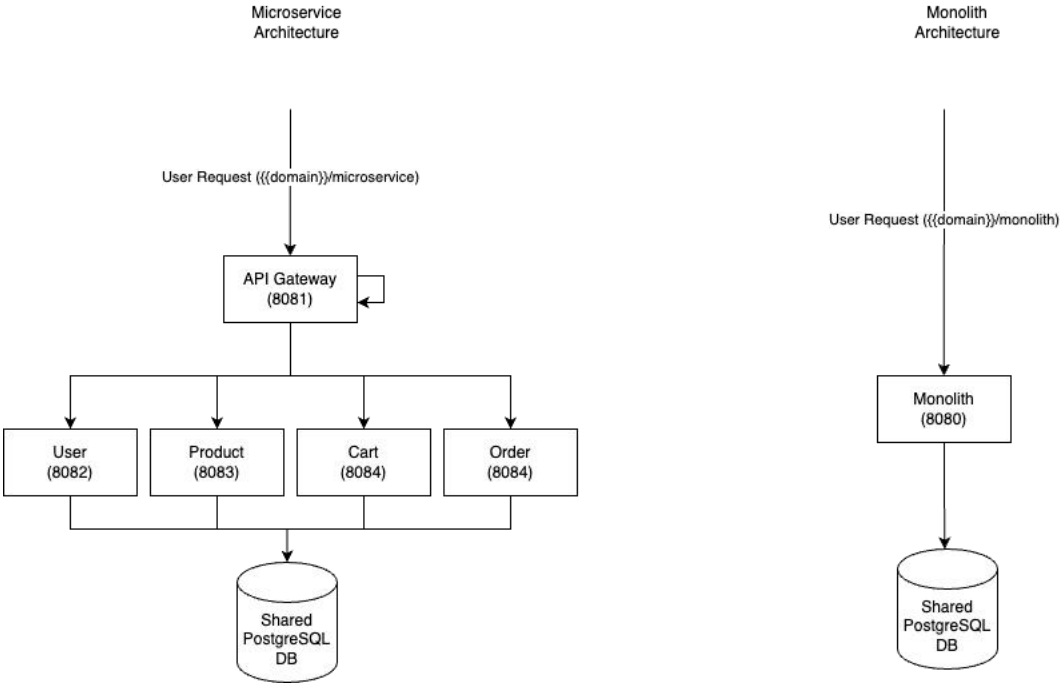

Figure 4.2: Structural Differences Between Monolithic and Microservices Architecture

As shown in Figure 4.2, a monolithic architecture is split into multiple services in the microservices architecture, with application components (microservices) that are independently managed and operated, but still share the same database - PostgreSQL, to handle all of the data. The traditional monolithic approach (on the right) combined all the functionalities into a single application, Monolith with 8080 port. consisting of user management, product catalogue, cart, and order processing that works together with the database. This design has various disadvantages such as scalability limitations, complex deployments, and tight coupling between components.

On the left we have the refactored application into the services (User with port number of 8082, Product with port number of 8083, Cart with port number of 8084, Order with port number of 8084), where each microservice runs in its own scope and have own deployment. There is API-Gateway with port number of 8081 (*Microservices Pattern: Pattern: API Gateway / Backends for Frontends*, no date) in front of them that acts as an entry point to clients. API Gateway handles requests from clients and forwards them to the specified microservice. Another change we can notice is that the microservices are using the single shared database created in PostgreSQL, in this case we can leverage the consistency in regards of the data that database will provide and it keeps refactoring/migration simpler as well.

This architecture adopts the Strangler Fig Pattern to incrementally replace the monolithic system component by component with microservices. The routing of traffic to the microservices is done progressively through the API Gateway until there is no traffic hitting the monolithic system and it can be decommissioned. This allows for a safer migration path but also introduces additional complexity. Increased overhead handling APIs, potential for database contention issues, or having to deal with transactions across services.

4.2.2 Service Decomposition Strategy

In the microservices architecture field, selecting an appropriate service decomposition strategy can help improve scalability and maintainability, and align development efforts with business requirements. Various service decomposition strategies have been proposed and put to practice.

A popular strategy is decomposing by business capability which is an approach to follow when designing and implementing Microservices. This simply refers to what a business does in order to generate value i.e. its responsibilities. For example, an E-commerce application might have services such as order management, payment management and so on. Decomposition of applications by business capability ensures the services are aligned with business processes (*Strangler fig pattern - AWS Prescriptive Guidance*, no date)

Another phenomenon is one alternative method that is very popular is decomposition by subdomain. This can be seen as an application of Domain-Driven Design (DDD). The key characteristic of this approach is to identify the different subdomain of the application domain and create microservices corresponding to the subdomains. This approach maintains the domain's integrity but de-correlates the different subdomains so that each subdomain can function independently of other subdomain. A practical example might be of a banking system where separate services might be built for user accounts, processing transactions and loans; each of these then work as independent applications unto themselves and encapsulate the entire domain knowledge (*Strangler fig pattern - AWS Prescriptive Guidance*, no date)

The Strangler Fig Pattern provides a practical strategy to incrementally migrate from monolithic architectures to microservices. The Strangler Fig Pattern proposes that incremental refactoring of monolithic components into microservices, allows the integration of new functionalities to work alongside legacy code while minimizing disruption. Once enough new functionality has been refactored into microservices, monolith core dwindles away and is replaced by microservices. This is an extremely useful strategy for large and complicated systems to rewrite everything from scratch would be time consuming and high risk (*Strangler fig pattern - AWS Prescriptive Guidance*, no date)

Furthermore, another approach is decomposition by transaction, in which the services are split up based on transactional boundaries. This way, a service can manage its own transactions without necessarily having to meddle with distributed transactions and reduce the transactional complexity that comes with it. Such method increases the consistency of data within a service. (*Strangler fig pattern - AWS Prescriptive Guidance*, no date)

When writing about these strategies, one option is to add some visual approaches. For example, a comparative table listing the characteristics, pros, and cons of each decomposition strategy could help summarize information for readers. Diagrams showing the evolution of a monolithic software architecture into a microservices software architecture through Strangler Fig Pattern could also illustrate moves from a practical aspect.

In conclusion, choosing the right decomposition strategy is critical to the successful adoption of a microservices architecture. By leveraging these strategies, organizations can decompose monolithic systems into more agile and scalable microservices-based architectures.

Table 4.1: Comparison of Service Decomposition Strategies(Chaieb, Sellami and Saied, 2023)

Decomposition Strategy	Ease of Implementation	Risk Level	Scalability	Data Management Complexity
Business Capability-Based	Medium	Medium	High	Medium
Domain Driven Design (DDD)	Hard	Medium	High	Hard
Event-Driven Decomposition	Medium	High	High	Hard
Strangler Fig Pattern	Easy	Low	High	Medium

Service decomposition is fundamental to the microservices architecture and can be achieved in different ways such as business capability-based decomposition, domain-driven design (DDD), event-driven decomposition, and Strangler Fig Pattern. Business capability-based decomposition, for instance, ensures that microservices align with underlying business functions, which means that they can often be easily developed and maintained by teams specializing in the target business capability, though it may sometimes be challenging to implement from unclear service boundaries. Meanwhile, DDD provides modularity and generally aligns with specific domain context for easier long-term warehouse management but requires deep business domain knowledge, whereas event-driven decomposition enables agile asynchronous communication across microservices but makes it harder to ensure data consistency in the system. The Strangler Fig Pattern is chosen because it enables low-risk and incremental migration of microservices. Microservices can be incrementally extracted from monolithic application while the monolith continues to operate fully. In contrast to a big-bang migration which can be highly risky and hard to implement, microservices can be gradually implemented with the monolithic system, alongside rollback paths if any needs arise. Furthermore, the capabilities of the monolith can continue to function during the migration and be modernized (*Strangler fig pattern - AWS Prescriptive Guidance*, no date). Overall, the Strangler Fig Pattern achieves a balance of ease of implementation, ease of scaling services up and down, and low operational risk can be seen in Table 4.1.

4.2.3 Data Management (Shared Database Model)

Handling Data in a Microservices Architecture Data management in microservices is a key concern and it can have a substantial impact on the performance and scalability of your system. Most approached strategy is database-per-microservice, where each microservice has its database, to enforce loose coupling. However, it leads to problems such as data inconsistency and ACID violation. Shared Database is an alternative approach, where multiple microservices can access the same database using a single relational database. A shared database has obvious benefits such as a simplified data management system. With the use of a single relational database, services can access freely other service's data, without the need for an inter-service communication channel, with the use of local ACID transactions to achieve consistency and integrity of their data (*Microservices Pattern: Pattern: Shared database*, no date).

However, using a shared database in a microservices architecture contradicts the principles of microservices and can potentially undermine their key benefits such as scalability, resilience, and independence. By tightly coupling services through a shared database, you risk introducing

a single point of failure and making services interdependent. This means that if one service modifies the schema of a table, it could potentially break other services that depend on that table. So, using a shared database requires careful consideration and management to avoid these challenges.

An Entity-Relationship Diagram (ERD) represents the data structure. An Entity-Relationship Diagram (ERD), is a data modeling technique that graphically illustrates an information system's data requirement and relationships between data. An ERD is a conceptual and representational model of data used to represent the data structure used. An ERD is a diagram that shows the relationship of entity sets stored in a database. In other words, ERDs illustrate the logical structure of databases. At first look, an entity relationship diagram looks very similar to a flowchart. ERD diagrams are commonly used in conjunction with a data flow diagram to display the contents of a data store. It is also the blueprint for designing and debugging relational databases.

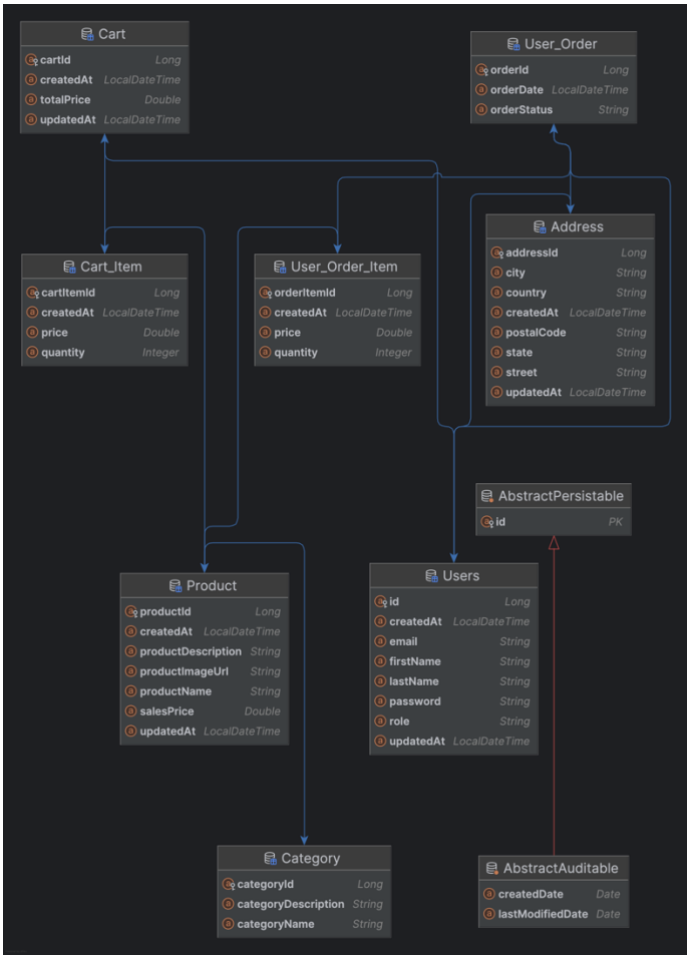

Figure 4.3: Entity Relationalship Diagram (ERD)

In the given ERD (Figure 4.3), you can see entities like **Users**, **Products**, **Orders**, and **Carts**, along with their attributes and relationships. For example, the Users entity has attributes such as user ID, name, and email, while the Orders entity has order ID, date, and total amount. The relationships between these entities are also shown, such as a user placing an order or adding a product to a cart.

In conclusion, while having a shared relational database can simplify managing data and guarantee consistency in a microservices architecture, it should be considered very carefully. Balancing the benefits of simplified data access and strong consistency against the potential

challenges of increased service coupling and reduced scalability is key to deploying microservices effectively.

4.2.4 Deployment and Scaling

The microservices architecture is deployed using docker, ensuring that each service runs in an isolated container and interacts with each services through an internal docker network. The central Postgres database also runs in a central location and all instances of the microservice communicate through an efficient connection pool to ensure there is no bottleneck during high user traffic. Git was used as the Version Control System (VCS) for managing source code and automation of deployment builds. Git VCS enables CI/CD pipelines to fully automate building/testing and deployment processes. For live monitoring, Prometheus was used to scrape and collect system metrics such as CPU usage, memory usage, and database query performance. Prometheus provides an overview of resource metrics and scalability graphs (Jani, 2024). However, docker also provides in-built monitoring capabilities and metrics. I am able to monitor container resource usage and performance metrics. This built-in docker features are able to suffice my current needs however, Prometheus can still be integrated in the future if my application demands higher levels of observability and advanced monitoring features.

4. 3 Methodology

Describe in detail how the experiments are performed. In the first part, provide an overview of the process/methodology.

4.3.1 System Design and Implementation

The system is constructed on Spring Boot framework utilized as a backend framework, PostgreSQL used as a relational database and Postman used as application for API testing. Spring boot framework provides light-weight, scalable components to process the request, dependency injection and service orchestration to create RESTful micro-services. Spring data JPA module easily integrates and we are using it to persist data into the PostgreSQL database. It follows a layered architecture by splitting the app into controller, service and repo layers. PostgreSQL is used in the database for database management for maintaining all the relational data in the database. Because each microservice must share the database for relational data integrity and consistency. So the persistent interaction with the database for each microservices is maintained through the Spring Data JPA repositories which can be used to execute the queries on the tables data & through the connection pooling using the Spring Data JPA it helps in doing the optimal management of connection with the database. This is by how those schemas of the database are defined through the Entity-Relationship model. So that we can able to maintain the data for user, product, order and cart consistently across the services. The system is build with REST API style in where each microservices only exposes a static set of endpoints that handles the requests. HTTP methods are standard in REST API. Every interaction is stateless, The behavior of Microservice REST API is really predictable. A microservice defines one or more well-known endpoints that are used to operate CRUD. I will describe some main endpoints below:

Table 4.2: REST API Endpoints for both Monolithic and Microservice Architecture

Service	Endpoints	Method	Description
User Service	/user	POST	Create User
User Service	/user	GET	Login User
User Service	/user/{userId}	PUT	Update User
User Service	/user/{userId}	DELETE	Delete User by Id
User Service	/user/{userId}	GET	Find User by Id
Product Service	/products	GET	Get all Products
Product Service	/products/{productId}	GET	Get Product by productID
Product Service	/products/{categoryId}	GET	Get Product by categoryID
Product Service	/products	POST	Add a Product
Category Service	/categories	GET	Get Categories
Category Service	/categories/{categoryId}	GET	Get Category by categoryID
Category Service	/category	POST	Add category
Address Service	/address/{addressId}	GET	Get address by addressID
Address Service	/address/{addressId}	POST	Add address by userID
Address Service	/address/{userId}	PUT	Update address by userID
Address Service	/address/user/{userId}	GET	Get User Address by userId
Cart Service	/cart/{cartId}	GET	Create a Cart with userId
Cart Service	/cart/{productid}/item		Add Item to Cart
Cart Service	/cart/{userId}	DELETE	Delete Cart by userId
Cart Service	/cart/{cartId}/{cartItemId}	DELETE	Delete CartItem from Cart given cartIdemId
Order Service	/orders	POST	Create an order
Order Service	/orders/{orderId}	GET	Get Order by orderId
Order Service	/orders/{orderId}/total	GET	Get Total Amount by orderId
Order Service	/orders/{orderId}/status	PUT	Update Order Status by orderId

As showed in Table 4.2: Comparison of Monolithic and Microservices API Endpoints the BASE URL API is the only difference between them, while all REST APIs have the same structure. Because of this, the transition from monolith to microservices is fairly seamless, and monolith services can continue to make the same requests but instead benefit from the ability to independently scale and deploy microservices, and by changing the main URL only the existing consumers of this API can communicate seamlessly with monolithic or microservices systems, it provides backward compatibility to prevent breaking changes, and facilitates the migration process. For request testing and Validation of the API steps.

For request testing and validation of the API steps. Perform and validate REST API calls, make various HTTP request methods, and validate the response as well We write postman collections for API testing, which is done to make sure each of the endpoints is working as expected. The following is an excellent designed and implemented RESTful microservice architecture with Spring Boot and an example of using PostgreSQL for TRU data persistence, database optimization strategies, and API integration tests with the testing framework.

4.3.2 Evaluation Setup

The assessment of the system is designed to test how well the microservices architecture performs in terms of scalability as well as how efficient it is compared to the previous monolithic implementation. The primary objective is to assess the performance of the microservices for concurrent requests, database operations, and the consumption of system resources based on different loads. The work has the following definition when it comes to the evaluation. Deploy in a deployed-controlled environment both architectures. Simultaneously co-host both the architectures in a machine and Perform Performance testing on API response time and track and monitor the resource consumption.

System is containerized in Docker so all of Microservices are hosted in isolated Docker container and communicated to each other with shared PostgreSQL database. To construct the test for experimentation, a personal computer with an 8-core CPU, 16GB memory and SSD storage was used, providing identical benchmarking conditions. We collect logs for all API including the database operation, and there are accumulated system metrics through Prometheus but not limited to CPU utilization, memory- usage, time spent on the database query and throughput of requests.

Using Apache JMeter, a series of performance tests are performed in which varying no. of users accessing the project API and making their respective requests are made to study how the system responds to different types and volumes of traffic. Some of the scenarios that are implemented as a test include, to a huge amount of simultaneous requests. Performance is evaluated based on factors such as the time it takes to handle a request, the time it takes to pass a query to the database, system latency, and the time to process the request, etc. It also assesses if the increased communication creates further database bottlenecks, through the common PostgreSQL database to serve multiple microservices concurrently executing their transactions.

Additionally, there are Git-based VCS that it use to maintain the deployments and code changes during the evaluation time period to keep the reproducibility and consistency of the tests contained. They were able to analyze the performance information that they gathered to show if the move from monolithic architecture to microservices achieved real benefits in scaling, fault tolerance and efficiency. By evaluating the system metrics, this study provides a quantified evaluation of the benefits and related costs of migrating from monolith to microservices.

4.3.3 Case Study/Experimental Setup

The aim of this study is to measure the performance difference by evaluating some key performance metrics for monolithic and microservices architectures at realistic workloads. The performance metrics are CPU utilization, response time, and memory consumption as I mentioned before in Research Questions part. These metrics will help us understand how each architecture behaves under different loads. The second key aspect is scalability. In this study, I'm interested to know what are some of the scalability limitations (if any) of both architectural styles by looking at how well each architecture is able to support increasing user requests. The third key aspect is refactoring effort. The refactoring effort is quantified by the time spent for system decomposition, code refactoring, and debugging based on firsthand experience. Moreover, I try to use help of peer-reviewed studies. The technical challenges are the internals of the refactoring effort. The technical challenges are identified by looking at some of the problems that may arise during migration such as those associated with database contention,

inter-service communication overhead, containerization, and some others. Here, the major technical challenges encountered during the refactoring process is also explored. The hardware, infrastructure and operational cost for both architectures together with their maintenance cost are the economic aspects to be evaluated by this study. These findings are supported by peer reviewed papers and information made available from online resources. These will be discussed on Results and Discussion (Results and Discussion) part more.

All experiments are conducted on a MacOS-based machine with 16GB RAM and an 8-core CPU, keeping a fixed experimental setup for consistency. Both architectures are running a cluster of Dockerized microservices, with each microservice deployed in its own Docker container and the PostgreSQL database being a single point of entry. In such a setup, differences in performance due to architectural differences rather than the hardware. The screen shot of the case study application that is used for testing is simple e-commerce application which has user registration, browse products, manage cart, place order, and make payment. WIDE is an application that simulates a real-world workload so that you can see how monoliths and microservices will perform in a real-world environment.

Three testing scenarios are executed to evaluate system performance in detail:

- **Scenario 1: Low Traffic Load:** This simulates a single user interaction with the system to set a baseline performance.
- **Scenario 2: Moderate Concurrent Requests:** Simulates several users visiting products, adding items to their cart, and making orders at the same time.
- **Scenario 3: Stress test with heavy traffic:** Creates thousands of simultaneous requests for the system and evaluates its scalability, responsiveness and resource consumption in extreme load conditions.

An assessment which uses Apache JMeter to perform simulation of user action and aggregate concurrent HTTP requests, and delivers real-time feedback on system performance by means of response times and requests per second. Besides, Docker's default command docker stats is used to follow CPU, Memory, and Network usage from each microservice as well as monolithic application running under it. We have integrated Prometheus so that we can capture high level metrics such as throughput, API execution times, database query execution times, and resource allocation. It analyzes logs to identify bottlenecks, service failures, and latency differences between architectures.

The study anticipates that microservices will exhibit better scalability under high-traffic scenarios (Scenario 3) as they can spread the workload across independently deployable services. Nonetheless, due to the inter-service communication overhead, microservices may face increased latency in processing complex transactions relative to the monolithic system.

In refactoring effort, monolith-to-microservices is expected to take substantial development time, as system decomposition creates problems of service orchestration, API management and database consistency. By quantifying migration effort, this study aims to identify if the long term benefits of modularity and independent scalability can sufficiently offset the initial complexity of migration.

From a cost perspective, microservices incur new costs related to container orchestration, service monitoring, and networking overhead, which should be weighed against their utility in providing improved fault tolerance and maintainability. The study expects that microservices

can drive higher infrastructure and operation costs, but they bring long-term efficiency gain through elasticity of resources and independence of services.

This assessment would offer a quantitative comparison between the two systems showing the benefits and trade-offs of employing microservices-based architecture compared to the monolithic-based architecture.

4. 4 Evaluation Criteria

The comparison between monolithic and microservice web architectures was made based on six criteria: performance, scalability, refactoring effort, cost, fault tolerance, and maintainability. Performance metrics include response time, throughput, CPU utilisation, and memory consumption. Monolithic architectures have better latency if there is low or medium traffic because it does not have the overhead of inter-service communication, but as concurrency increases, microservices come into their own as load gets distributed among several independently deployed services. This was also observed in a performance and scalability evaluation of monolithic and microservices-based applications where microservices performed better with increasing high concurrency (Blinowski, Ojdowska and Przyblek, 2022).

Scalability: Monolithic systems are usually scaled vertically i.e. they rely on increasing the capacity of the single server by increasing CPU and memory. This has limits and can also get very expensive. Microservices architectures on the other hand allow horizontal scaling i.e. independent services can be scaled independently based on their demand. This repackaging allows microservices architectures to be able to scale better, have better fault tolerance and faster updates. Hence, this architecture would suit large applications and those with growing users (Ortega, no date). The trade off is the complexity in deployment and monitoring.

Refactoring effort: The migration of a monolithic architecture to a microservices architecture is a very large effort. It requires business functions to be split from the monolith to discrete services, the monolithic database schema to be redesigned to split data storage concerns and APIs to be developed to handle service communication. This requires careful planning and execution (Blinowski, Ojdowska and Przyblek, 2022).

Cost Analysis: The following cost analysis will cover both infrastructure & operational costs In a microservices architecture, we have multiple containers for the multiple services that we have, an orchestrator which we can deploy these containers & make them available so the infrastructure cost is more than that of a monolithic architecture. So initially, the cost of a microservices architecture is more than that of a monolithic architecture just to get it up & running. But down the line, we do have individually scalable services that optimize the overall usage of our infrastructure & increase the overall availability of our services (Monolithic vs Microservices - Difference Between Software Development Architectures- AWS, no date).

Fault tolerance: In a microservices architecture, if one component of the system fails it would not affect other services or the whole system. Similarly, in a monolithic architecture, a failure in one of the modules causes a full system failure (Harris, no date).

Maintainability: A microservices architecture allows development teams to work on different services independently as they can work on modular components. The teams can also ensure continuous integration and continuous development (CICD) which allows faster updates of the applications, and easier debugging. However, the deployment and monitoring complexity for

a microservice architecture is higher and robust DevOps processes are required to ensure service availability (Harris, no date).

On the other hand, a monolithic architecture, is easier to manage and maintain as there is a single codebase and a single deployable unit, but can become cumbersome as the application grows and developers move more slowly as the application grows to avoid being breaking changes. Monolithic systems also have tightly coupled dependencies which means that in a monolithic system if an update is made it requires pushing the whole codebase to make the updates. On the other hand, as a microservices application is running in multiple processes, a particular service can be deployed without reflecting it in the other services. This makes the monolithic systems inflexible (Harris, no date).

Table 4.3: Summary of Evaluation Metrics and Their Purpose

Category	**Metric**	**How it is measured**	**Why it matters?**
Performance	Response Time	Measured in milliseconds using JMeter	Determines system speed under load
Performance	Throughput	Requests processed per second using logs	Evaluates system efficiency
Performance	CPU Utilization	Percentage of CPU used via Docker stats	Evaluates system efficiency
Performance	Memory Consumption	RAM used per request via Docker stats	Indicates memory optimization
Scalability	Max Concurrent Requests	Number of simultaneous users supported	Measures system load capacity
Refactoring	Migration Effort (Hours)	Time spent decomposing monolith	Quantifies transition complexity
Cost	Infrastructure Cost	Compute power, hosting expenses	Compares initial vs. long-term cost
Cost	Operational Cost	Monitoring, deployment expenses	Evaluates maintenance costs
Reliability	Failure Recovery Time	Time taken for recovery after failure	Measures fault tolerance
Maintainability	Debugging Complexity	Time required to identify and fix issues	Compares maintainability challenges

Table 4.3 presents a high-level summary of the important attributes of various evaluation criteria. The evaluation criteria are classified into performance, scalability, refactoring effort, cost, reliability, and maintenance. (i) Performance is evaluated using response time, throughput, CPU utilization, and memory utilization. (ii) Scalability is evaluated using the maximum number of concurrent users and the level of scalability. In other words, the level of scalability indicates vertical versus horizontal scalability. (iii) The refactoring effort is a measurement of the effort to migrate a monolith to microservice. Therefore, this metric is only for the comparison between the monolithic architecture and microservices architecture. We compared the ease of migration. (iv) Cost is evaluated using the infrastructure cost and management cost. (v) The system reliability is evaluated using the failure recovery time. (vi) The level of maintenance is evaluated by debugging complexity.

In summary, while monolithic architectures may offer advantages in simplicity and lower initial costs, microservices architectures provide superior scalability, fault tolerance, and long-term maintainability. Organizations should carefully consider these trade-offs in the context of their specific needs and resources when deciding on an architectural approach.

4. 5 Benchmark Algorithms

Study primarily evaluates monolithic and microservices architecture based on different benchmarking methodologies and performance measurement techniques to compare the efficiency of the two architecture styles on the grounds of performance, scalability, fault tolerance and resource utilization. Monolithic and microservices architecture were compared focusing on key metrics like API response times, system throughput, CPU usage and memory usage for the applications under different loads. Apache JMeter was used to simulate realistic workloads by performing load testing on the application and analyzing how effectively each architecture responded to increasing concurrent requests. System was containerized using docker for a uniform test environment and PostgreSQL was chosen for database. Performance of the two architecture was evaluated under the same conditions.

Scalability tests - Differences in vertical and horizontal scaling In monolithic architecture, scalability is provided by implementing increased memory capacity and solid-state drive for a centralized machine, whereas a microservice architecture allows for multiple deployments of the same service at once, allowing load distribution across multiple machines. This essentially constitutes the main difference between monolithic and microservice architecture scalability.

For database performance, PostgreSQL was tested in both architecture. The intention of these tests was to benchmark the database performance and to see how having a shared database will affect the microservices while performing a query or processing transactions. As the microservices have a shared database(monolithic model) we wanted to identify if there would be any performance bottlenecks in the database and that there will not be any unnecessary cost in terms of connection overhead caused by it. Docker Stats was used to monitor these while resource consumption was recorded.

Fault tolerance testing evaluated the effects an individual service failure had on the system as a whole. In monolithic architecture, failure in a key component usually causes a catastrophic system failure, whereas in microservices, the service isolation mechanisms allow other components to continue the execution. In this study fault handling capability in each architecture was measured by deliberately killing the services from the execution loop and monitoring the time it took for them to recover.

Resource utilization and operational cost analysis has been performed by measuring CPU and memory usage during different load scenarios. Even though microservices can take more system resources per transaction, their ability to distribute a load dynamically across system instances makes them more efficient under the high load.

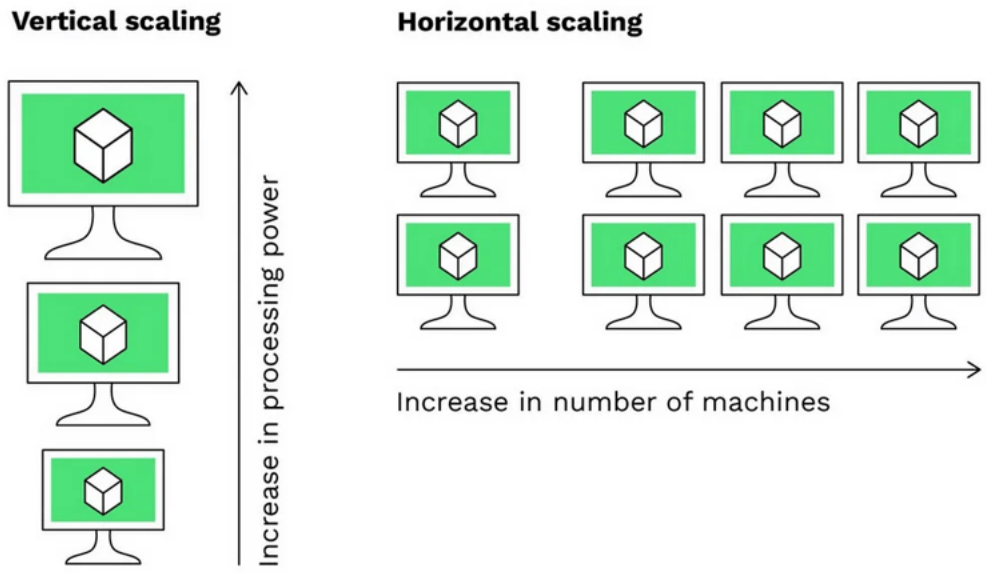

Figure 4.4: Vertical Scaling vs. Horizontal Scaling(Perry, 2023)

To explain the basic difference between monolithic and microservices architectures in terms of scaling, the example in Figure 4.4 (2024) will visually depict how they each scale when under incremental workloads. The monolithic application scales by increasing the CPU and RAM of the bare metal server or by upgrading the hardware of the current server (i.e. vertical scaling). Expanding the single server can even prove impossible if the machine is limited by physical resource constraints. Furthermore, upgrading the machine to a bigger machine can be difficult and more expensive than horizontal scaling of microservices. In Figure 4.4, the microservices architecture represents three smaller versions of an independent service that is responsible for handling client requests. Horizontal scaling creates numerous smaller instances of independent services sparingly using the resources rather than using the resources as a whole (i.e. memory and CPU). Vertical scaling is done by expanding a single machine, whereas horizontal scaling is done by load balancing across multiple service instances. Moreover, microservices architectures scale fault isolation since loads are diversified by services instead of reliant on an individual service under increased network traffic.

The benchmark is providing an elaborate comparison between monolithic and microservices architecture based on practical experimentations. The results from the experimentations will be further discussed in Section 5 (Results & Discussion) to conclude on the overall trade-offs between monolithic and microservices architecture.

Chapter 5. Results and Discussion

5. 1 Quantative Insights from System

This chapter presents the results obtained from the transition from a monolithic architecture to a microservices-based system. The analysis is primarily driven by quantitative performance metrics collected using Docker Stats and Apache JMeter, focusing on CPU, memory, service startup times, and request response times under varying loads. Apache JMeter is used to simulate concurrent user requests, allowing for a detailed comparison of the system's scalability, latency, and throughput before and after the transition.

Additionally, we assess the time spent on refactoring, highlighting key challenges encountered during the transformation process. A comprehensive cost evaluation is conducted, comparing hardware, infrastructure, and operational expenses between the monolithic and microservices architectures. Finally, we discuss the operational complexity and maintenance overhead introduced by microservices, evaluating its long-term implications.

5. 2 Performance Under Load: Docker Stats Analysis

This section presents the CPU and memory utilization metrics obtained from Docker Stats during load testing of both monolithic and microservices architectures. The tests were conducted with 1000, 2500, and 5000 concurrent users to analyze how each system scales under stress. The results focus on resource allocation efficiency, system bottlenecks, and performance trade-offs.

5.2.1 Monolithic System Performance

Figures 5.1 to 5.4 show the Docker resource consumption of the monolithic system in different loads. CPU and memory utilization has been captured to illustrate how resource consumption increases as the number of concurrent users increases. Analysis into this data presents a good picture of the scalability and efficiency of the monolithic system. By inspecting these metrics, it is possible to see how the monolithic architecture has been coping with the load in given situations. Potential bottlenecks, resource consumption patterns and system resilience capabilities will be highlighted at this stage of evaluation. Furthermore, the data helps to show the how efficiently this system utilizes the resources when idle compared to when during peak consumption. These observations create a good overall picture of the monolithic system as it stands before the conversion to a microservices based system.

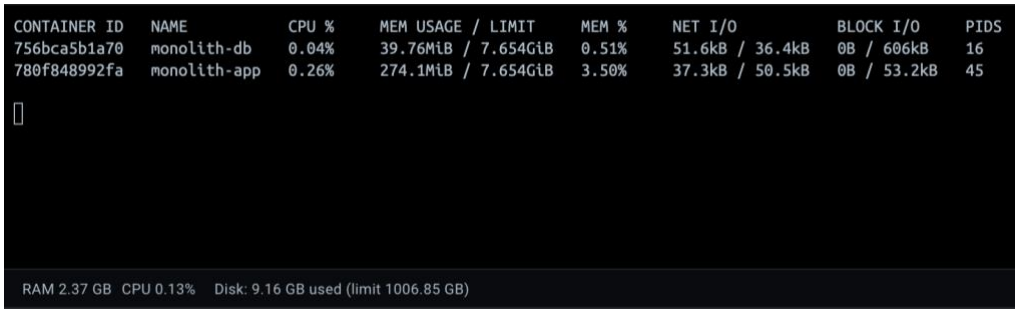

Figure 5.1: Monolith Resources in IDLE

Figure 5.1 demonstrates the base resource usage of the monolith architecture when it is in idle state. At this point, the CPU and Memory usage is low. There are no requests at this point.

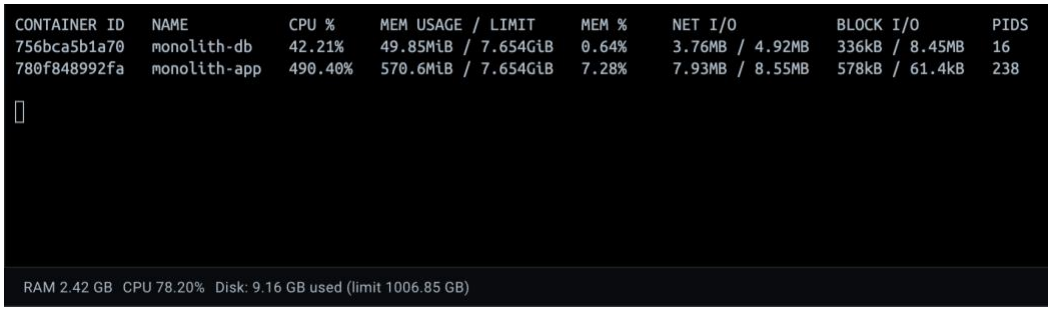

Figure 5.2: Monolith Resorcues with 1000 User Load

As shown in Figure 5.2, the CPU load increases significantly when subjected to 1000 concurrent users. The monolithic application struggles with resource allocation, leading to noticeable CPU spikes.

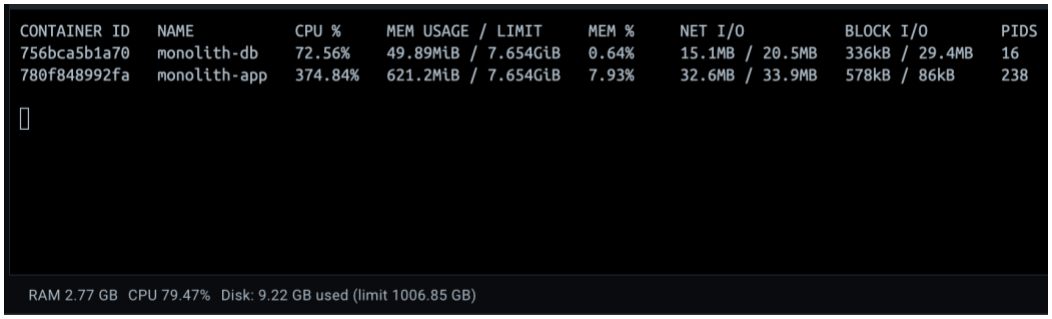

Figure 5.3: Monolith Resources with 2500 User Load

Moving to 2500 concurrent users (Figure 5.3), the memory footprint starts increasing rapidly, with the monolith showing early signs of resource saturation.

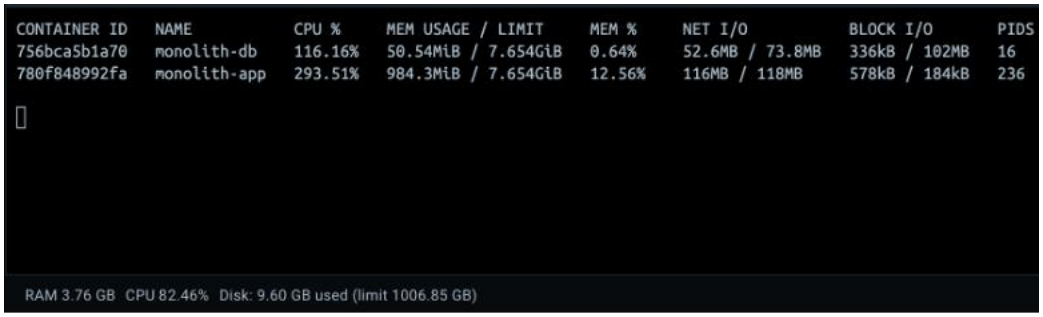

Figure 5.4: Monolith Resources with 5000 User Load

In Figure 5.4, I illustrate the case of the extreme load (5000 users). As expected, an increase in resource (CPU and Memory) usage was observed.

5.2.2 Microservices System Performance

Figures 5.5 to 5.8 illustrate Docker resource utilization for the microservices architecture. These figures provide a detailed analysis of how resource consumption varies across different microservices under increasing load conditions. By examining CPU and memory usage, we can assess the distribution of computational demands among individual services, highlighting the efficiency of resource allocation and potential performance bottlenecks. Additionally, the data offers insights into how the microservices architecture scales in response to concurrent user requests, comparing its performance characteristics to those of the monolithic system. This analysis helps evaluate the benefits and trade-offs of adopting a microservices approach in terms of resource efficiency, system resilience, and overall scalability.

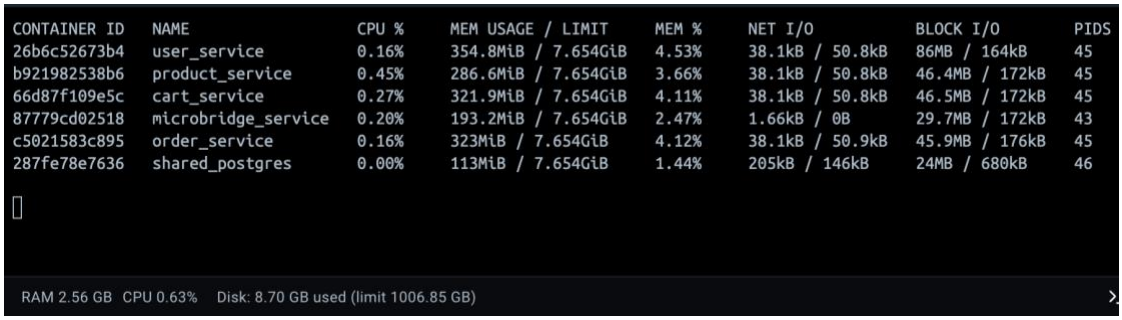

Figure 5.5: Microservice Resources IDLE

Figure 5.5 demonstrates the base resource usage of the microservices architecture when it is in idle state. At this point, the CPU and Memory usage is low. There are no requests at this point.

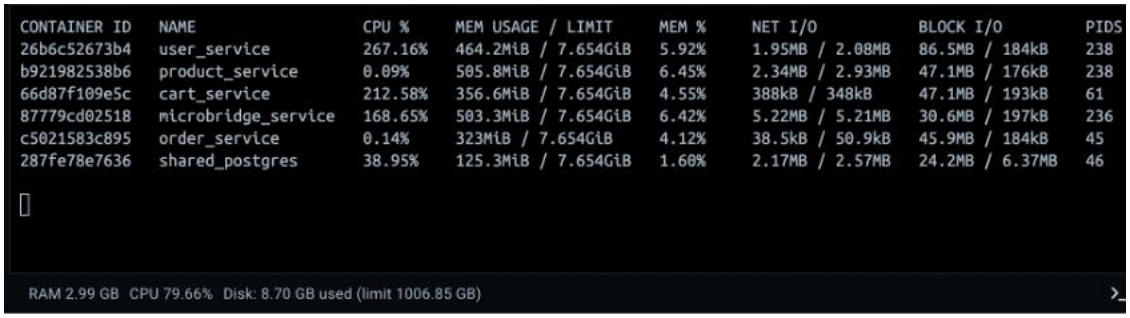

Figure 5.6: Microservice Resources with 1000 User Load

As shown in Figure 5.6, the CPU load increases significantly when subjected to 1000 concurrent users. The microservice architecture struggles with resource allocation, leading to noticeable CPU spikes.

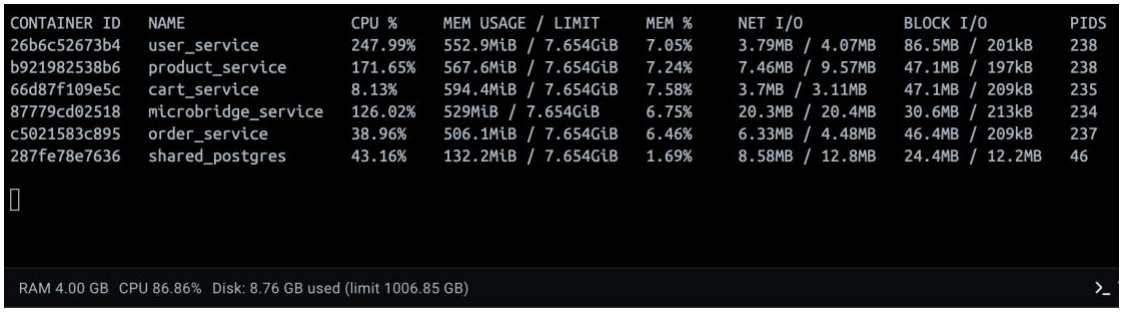

Figure 5.7: Microservice Resources with 2500 User Load

Moving to 2500 concurrent users (Figure 5.7), the memory footprint starts increasing rapidly, with the monolith showing early signs of resource saturation.

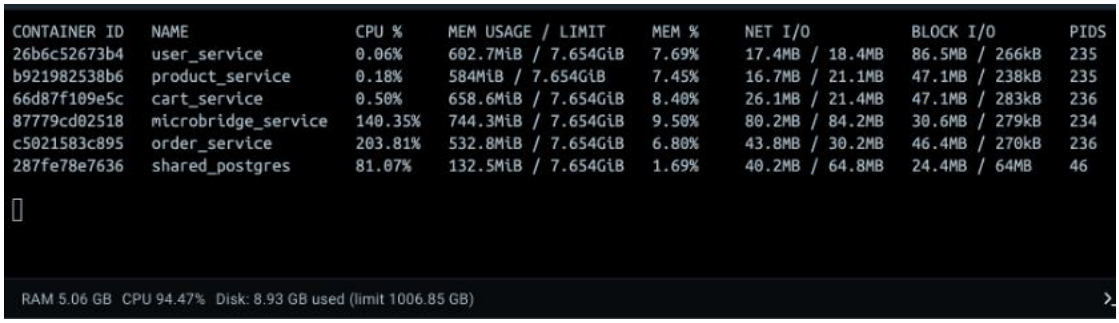

Figure 5.8: Microservice Resources with 5000 User Load

In Figure 5.8, I illustrate the case of the extreme load (5000 users). The system reached the resource endpoints (CPU and memory) demonstrating the bottlenecks of the monolith architecture.

5.2.3 CPU Usage Analysis

Examining CPU utilization between these architectures, with user load is a crucial metric. As processing demands increase, the CPU consumption of both microservices and monolithic architectures increases. But microservices CPU utilization scales uniquely. Given the distributed nature of microservices architectures, CPU consumption can vary compared to monolithic applications. With monolithic systems, all processing is consolidated within a single application. In contrast, microservices systems distribute this processing across multiple microservices, potentially incurring additional processing overhead.

Table 5.1: CPU Utilization Data

User Load	Monolith CPU Usage(%)	Microservice CPU Usage(%)
Idle	0.13	0.63
1000 Users	78.2	79.96
2500 Users	79.47	89.86
5000 Users	82.46	94.47

As we look in Table 5.1, we can see that when the system is in its idle state, the monolithic system is just consuming 0.13% CPU but the microservices is consuming 0.63% CPU. This small increase in CPU consumption for the microservices is due to overhead of running multiple independent services, even when there is little user interactions with the system. As the number of concurrent users increases, there are significant spikes in CPU usage for the both monolithic and microservices architecture but for 1000 concurrent users, the CPU consumption measures 78.20% for Monolithic architecture while it measures 79.96% for microservices. At this benchmark point for both.

As the number of users further increases, CPU utilization in microservices starts to increase more rapidly than in monolithic systems. At 2500 users, monolithic CPU usage is roughly constant at 79.47% compared to microservices which take higher usage of 89.86%. This trend also follows when running 5000 users which monolithic CPU usage is 82.46% compared to microservices of 94.47%. These results show that although microservices have the advantage of distributing more evenly, there is a higher computational cost involved when switching tasks between different services.

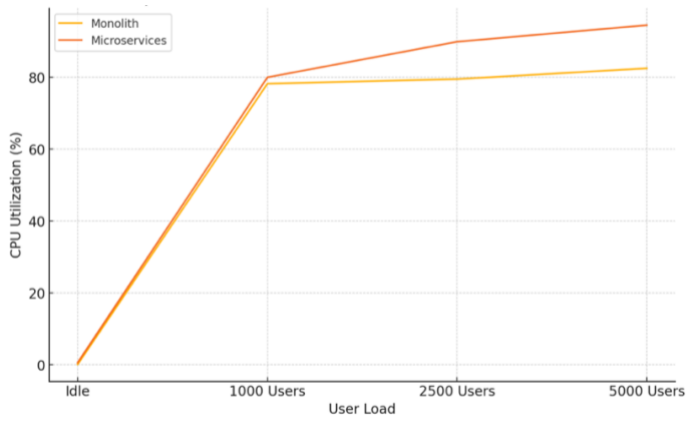

Figure 5.9: CPU Utilization Trends

The graphical representation from Figure 5.9 shows another case of higher CPU demand from both of the architectures under some load. The monolithic architecture exhibits a much smoother curve when handling CPU usage, gradually increasing going into higher user requests. Meanwhile, Microservices show a gradual and steeper curve, meaning that the more users that are trying to access the system, more CPU resource is used in an exponential way. This demonstrates one of the problems with microservices, which are processing overheads. Compensating increased CPU usage for better system scalability.

This analysis of CPU utilization reveals that monolithic systems exhibit a stable and anticipated CPU workload, while microservices scale functionality, but at a higher CPU workload cost. The numerical data of the CPU information is shown in Table 5.1, which illustrates how microservices require increasingly more processing power as the load increases. On the other hand, Figure 5.9 depicts this data graphically and confirms this trend, indicating that the scalability provided by microservices suffers from increased resource demand. This suggests that although greater activate scalability is offered in combination with the advantage of distributing workloads, without additional optimizations, such as load balancing mechanisms, caching strategies, and optimized inter-service API calls, the CPU overhead could lead to undesirable server utilization or additional costs. In the future, further work should be completed to better identify how intelligent autoscaling policies and optimized microservices communication models could further improve system performance while offering this level of scalability.

5.2.4 Memory Usage Analysis

The evaluation of memory utilization between monolithic and microservices architectures provides crucial insights into how each system manages memory under increasing load conditions. Since microservices operate in a distributed manner, memory allocation and consumption behave differently compared to a monolithic system, where all processes share a single memory space..

Table 5.2 : Memory Utilization Data

User Load	Monolith RAM Usage (GB)	Microservice Total RAM Usage (GB)
Idle	2.37	2.56
1000 Users	2.42	2.99
2500 Users	2.77	4.0
5000 Users	3.77	5.06

From Table 5.2, it is evident that at an idle state, the monolithic system consumes 2.37GB of memory, whereas microservices require slightly more at 2.56GB. This difference arises because microservices run multiple independent services, each maintaining its own memory footprint, even when no user requests are being processed. As the system load increases to 1000 users, monolithic memory usage rises modestly to 2.42GB, while microservices require 2.99GB. The nearly 0.6GB difference at this stage indicates that microservices allocate more memory per service, leading to higher overall consumption.

At 2500 users, the memory disparity becomes more pronounced, with monolithic usage increasing to 2.77GB, while microservices escalate sharply to 4.00GB. This sharp increase reflects the distributed architecture’s demand for additional memory to support concurrent

service execution, inter-service communication, and in-memory data storage for independent services. Finally, at 5000 users, monolithic memory usage peaks at 3.76GB, while microservices reach 5.06GB, reinforcing the trend that microservices demand significantly more memory as concurrency levels increase.

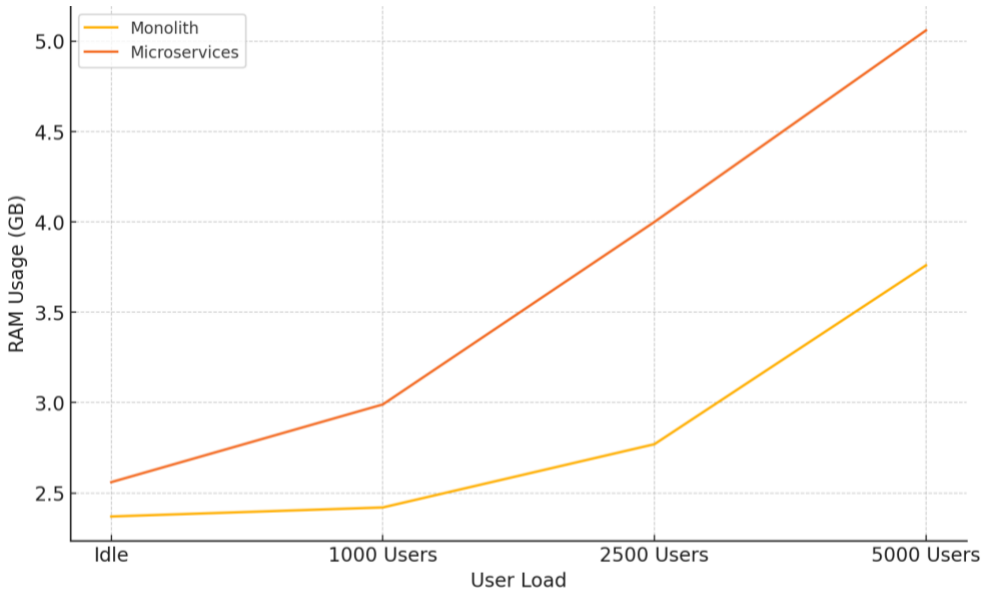

Figure 5.10: Memory Utilization Trends

The graphical representation in Figure 5.10 further illustrates this trend, showing the difference in how memory is consumed by monolithic and microservices architectures at increasing user loads. While monolithic memory usage increases gradually, following a more controlled growth pattern, microservices demonstrate a steeper rise, particularly beyond 2500 users. The increasing gap in memory utilization suggests that while microservices offer scalability advantages, they also impose a significant memory overhead that must be managed efficiently.

One key factor contributing to the higher memory consumption in microservices, as seen in Figure 5.10, is service isolation. Unlike monolithic systems, where all components share the same memory space, each microservice runs as an independent process, often requiring its own memory allocation for execution and caching. This means that instead of a single large application utilizing memory collectively, microservices require separate memory allocations for multiple instances, leading to greater overall consumption. Additionally, microservices rely heavily on inter-service messaging and API calls, which further increase memory overhead as data is transmitted and temporarily stored between services.

In conclusion, Table 5.2 and Figure 5.10 together highlight the fundamental differences in memory consumption between monolithic and microservices architectures. While monolithic systems maintain a more stable and predictable memory footprint, microservices exhibit a steeper rise in memory demands as user concurrency increases. Table 5.2 provides numerical evidence of this growth, showing that microservices require almost 35% more memory than monolithic systems at 5000 users. Meanwhile, Figure 5.10 visually illustrates the increasing divergence in memory usage, emphasizing the impact of distributed architecture on resource allocation. These findings suggest that while microservices enable better scalability and modularity, they require careful memory optimization strategies, such as improved caching, memory-efficient service orchestration, and resource pooling, to mitigate excessive memory

overhead and ensure sustainable performance at scale. Future optimizations should explore techniques like memory-aware autoscaling and improved garbage collection mechanisms to enhance efficiency in microservices architectures.

5. 3 Performance Under Load: Response Time and Throughput Analysis

In this section, we analyze the performance of the Monolithic and Microservices architectures under varying user loads, focusing on response time and throughput. Response time measures how quickly the system processes requests, while throughput quantifies the number of requests handled per second (Chen et al., 2024). To evaluate these metrics, Apache JMeter (Apache JMeter - Apache JMeterTM, no date) was used to simulate concurrent user requests at 1000, 2500, and 5000 users. The collected data provides insights into how each architecture scales under increasing demand.

The performance evaluation was conducted using various system services, including Update Order Status, Update Address, Get Total Amount, Get Products, Get Order, Get Categories, Get Address, Create a Cart, and Add Item to Cart, as shown in Table 4.2. These services represent critical system operations, and their performance under different load conditions highlights how efficiently each architecture handles increasing concurrency.

5.3.1 Monolithic System Performance

This subsection presents the response time and throughput results for the Monolithic system under different user loads. Apache JMeter was used to simulate concurrent user requests at 1000, 2500, and 5000 users, capturing the system's performance metrics. The figures below illustrate the results obtained from the tests.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received ...	Sent KB/sec	Avg. Bytes
Get Produ...	1000	249	4	1753	213.84	0.00%	411.9/sec	388.54	53.90	966.0
Get Categ...	1000	216	2	1031	149.90	0.00%	337.8/sec	138.24	44.87	419.0
Get Categ...	1000	244	2	1076	142.73	0.00%	313.3/sec	82.60	41.61	270.0
Get Adres...	1000	295	2	1584	163.14	0.00%	290.4/sec	102.94	38.29	362.9
Update A...	1000	336	8	963	157.35	0.00%	258.3/sec	92.29	90.80	365.9
Create a C...	1000	320	23	967	145.92	0.00%	251.3/sec	50.79	32.39	207.0
Add Item t...	1000	325	9	886	135.46	100.00%	252.0/sec	65.20	68.16	265.0
Get Order ...	1000	285	9	1050	136.62	0.00%	256.1/sec	69.27	33.51	277.0
Get Total ...	1000	243	3	1017	128.36	0.00%	263.9/sec	46.13	36.08	179.0
Update Or...	2000	205	3	997	149.67	0.00%	536.5/sec	146.69	130.45	280.0
TOTAL	11000	266	2	1753	160.42	9.09%	2584.6/sec	888.43	477.73	352.0

Figure 5.11: Monolithic System Response Time and Throughput Under 1000 User Load

The first test was conducted with 1000 concurrent users. The system's response time and throughput values at this stage are recorded in Figure 5.11.

Get Produ...	2500	822	0	2888	607.89	0.20%	815.4/sec	771.61	106.49	969.0
Get Categ...	2500	373	2	1192	171.11	0.00%	636.1/sec	260.29	84.49	419.0
Get Categ...	2500	489	2	1271	196.29	0.00%	515.7/sec	135.97	68.49	270.0
Get Adres...	2500	573	2	1517	249.05	0.00%	404.7/sec	143.43	53.36	362.9
Update A...	2500	712	6	1806	266.30	0.00%	360.4/sec	128.79	126.72	365.9
Create a C...	2500	799	2	1674	267.81	0.00%	314.9/sec	63.66	40.59	207.0
Add Item t...	2500	859	4	1761	237.83	100.00%	286.4/sec	74.12	77.47	265.0
Get Order ...	2500	866	7	1596	207.58	0.00%	273.3/sec	73.94	35.77	277.0
Get Total ...	2500	784	5	1721	223.26	0.00%	270.2/sec	47.23	36.94	179.0
Update Or...	5000	638	4	1916	306.09	0.00%	538.0/sec	147.10	130.82	280.0
TOTAL	27500	687	0	2888	335.30	9.11%	2934.3/sec	1009.39	542.29	352.3

Figure 5.12:Monolithic System Response Time and Throughput Under 2500 User Load

For 2500 concurrent users, the system's response time and throughput were measured again to observe performance changes as the load increased can be seen in the Figure 5.12.

Label	# Samples	Average	Min	Max	Std. Dev.	Error % ↑	Throughput	Received ...	Sent KB/sec	Avg. Bytes
Get Produ...	4414	816	17	2882	436.16	0.00%	528.8/sec	498.90	69.20	966.1
Get Categ...	4414	900	6	2320	457.28	0.00%	447.7/sec	183.18	59.46	419.0
Get Categ...	4414	1054	6	2268	505.38	0.00%	392.6/sec	103.52	52.14	270.0
Get Adres...	4414	1139	5	2582	503.22	0.00%	333.1/sec	118.03	43.91	362.9
Update A...	4414	1302	7	2590	542.53	0.00%	300.2/sec	107.26	105.54	365.9
Create a C...	4414	1353	6	2785	520.04	0.00%	277.0/sec	56.00	35.71	207.0
Get Order ...	4414	1229	53	2614	469.28	0.00%	253.0/sec	68.44	33.11	277.0
Get Total ...	4414	1051	4	2665	452.63	0.00%	248.6/sec	43.45	33.98	179.0
Update Or...	8828	870	4	2447	494.33	0.00%	492.0/sec	134.54	119.64	280.0
Add Item t...	4414	1302	20	2674	451.14	100.00%	262.8/sec	68.02	71.10	265.0
TOTAL	48554	1081	4	2882	520.55	9.09%	2634.8/sec	905.68	487.01	352.0

Figure 5.13: Monolithic System Response Time and Throughput Under 5000 User Load

The final test was performed with 5000 concurrent users, capturing the system's response time and throughput under the highest load condition can be seen in the Figure 5.13.

This section only presents the recorded results for the Monolithic architecture under different loads. A comparative discussion of these results with the Microservices architecture is provided later in Section 5.3.3.

5.3.2 Microservices System Performance

This subsection presents the response time and throughput results for the Microservices architecture under different user loads. Apache JMeter was used to simulate concurrent user requests at 1000, 2500, and 5000 users, capturing the system's performance metrics. The figures below illustrate the results obtained from these tests.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received ...	Sent KB/sec	Avg. Bytes
Get Produ...	1000	1120	1	2316	733.59	0.10%	405.4/sec	265.46	54.57	670.6
Get Categ...	1000	248	9	1429	104.64	0.00%	364.8/sec	131.82	49.88	370.0
Get Categ...	1000	261	8	706	137.18	0.00%	332.8/sec	87.74	45.50	270.0
Get Adres...	1000	286	3	1747	186.03	0.00%	215.7/sec	77.75	29.29	369.0
Update A...	1000	464	8	1674	318.54	0.00%	204.1/sec	72.92	72.54	365.9
Create a C...	1000	388	12	1509	261.61	0.00%	197.3/sec	45.67	26.21	237.0
Add Item t...	1000	467	29	1774	286.61	0.00%	193.0/sec	40.52	52.96	215.0
Get Order ...	1000	333	4	929	147.82	0.00%	181.4/sec	56.69	24.45	320.0
Get Total ...	1000	325	10	1003	152.76	0.00%	177.8/sec	41.67	25.00	240.0
Update Or...	2000	287	8	951	151.92	0.00%	355.1/sec	111.99	87.72	323.0
TOTAL	11000	406	1	2316	377.02	0.01%	1715.3/sec	563.96	323.72	336.7

Figure 5.14: Microservice System Response Time and Throughput Under 1000 User Load

The first test was conducted with 1000 concurrent users. The system's response time and throughput values at this stage are recorded can be seen in Figure 5.14.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received ...	Sent KB/sec	Avg. Bytes
Get Produ...	2500	840	48	1886	420.26	0.00%	370.4/sec	241.94	49.92	668.8
Get Categ...	2500	858	26	1558	407.54	0.00%	326.5/sec	117.97	44.64	370.0
Get Categ...	2500	935	27	1643	373.77	0.00%	279.0/sec	73.57	38.15	270.0
Get Adres...	2500	1232	29	2736	428.63	0.00%	222.5/sec	80.17	30.20	369.0
Update A...	2500	1483	21	3751	476.40	0.00%	202.5/sec	72.34	71.97	365.9
Create a C...	2500	1289	8	2156	328.21	0.00%	184.5/sec	42.71	24.51	237.0
Add Item t...	2500	1318	9	2178	308.06	0.00%	176.8/sec	37.12	48.51	215.0
Get Order ...	2500	1155	10	1678	271.73	0.00%	168.1/sec	52.52	22.65	320.0
Get Total ...	2500	1045	8	1654	353.48	0.00%	165.3/sec	38.74	23.25	240.0
Update Or...	5000	857	6	1736	405.42	0.00%	328.3/sec	103.55	81.11	323.0
TOTAL	27500	1079	6	3751	441.40	0.00%	1618.1/sec	531.76	305.41	336.5

Figure 5.15: Microservice System Response Time and Throughput Under 2500 User Load

For 2500 concurrent users, the system's response time and throughput were measured again to observe performance changes as the load increased can be seen in Figure 5.15.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received ...	Sent KB/sec	Avg. Bytes
Get Produ...	4042	1014	0	2611	377.55	0.96%	922.6/sec	618.42	123.14	686.4
Get Categ...	4042	959	3	2430	190.38	0.02%	723.1/sec	261.60	98.83	370.5
Get Categ...	4042	1173	9	2150	389.06	0.00%	537.1/sec	141.63	73.44	270.0
Get Adres...	4042	1522	3	2960	517.80	0.00%	391.7/sec	141.15	53.17	369.0
Update A...	4042	1972	6	3236	606.07	0.00%	330.0/sec	117.93	117.32	365.9
Create a C...	4042	2170	4	3264	487.49	0.00%	290.0/sec	67.11	38.51	237.0
Add Item t...	4042	2210	67	3290	439.12	0.00%	263.4/sec	55.31	72.29	215.0
Get Order ...	4042	1987	4	2810	352.46	0.00%	247.3/sec	77.29	33.33	320.0
Get Total ...	4042	1814	5	2779	230.69	0.00%	237.0/sec	55.54	33.32	240.0
Update Or...	8084	1488	10	2790	405.50	0.00%	463.3/sec	146.14	114.47	323.0
TOTAL	44462	1618	0	3290	595.27	0.09%	2114.4/sec	698.25	398.83	338.2

Figure 5.16: Microservice System Response Time and Throughput Under 5000 User Load

The final test was performed with 5000 concurrent users, capturing the system's response time and throughput under the highest load condition can be seen in in Figure 5.16.

This section only presented the recorded results for the Microservices architecture under different loads. A comparative discussion of these results with the Monolithic architecture is provided later in Section 5.3.3.

5.3.3 Comparative Analysis of Response Time and Throughput

In this part, we will examine Monolithic and Microservices architecture response time and throughput graphs and discuss these architectures comparatively in terms of load levels. The following charts show the performance of both architectures with respect to their latency and request-handling capacity against growing concurrency.

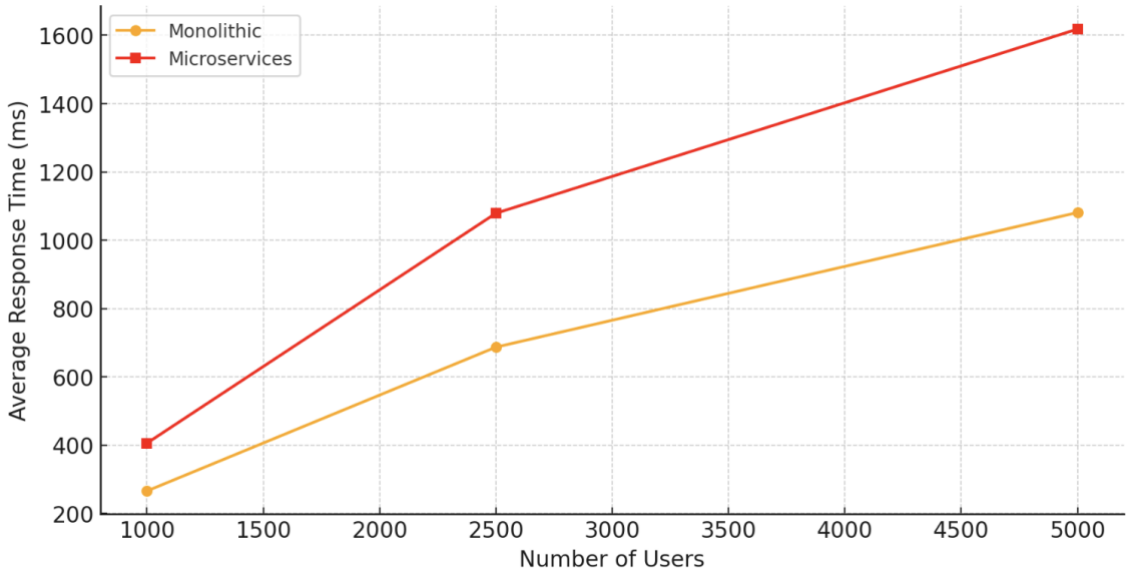

Figure 5.17: Response Time vs. Users (Monolith vs. Microservices)

The response time graph (Figure 5.17) illustrates that the two architectures diverge significantly with increasing load in terms of number of concurrent users. Average Response Time : 266 ms Free (Monolithic) vs 406 ms (Microservices) for 1000 concurrent users. When the number of concurrent users increases to 2500, the Monolithic systems response time increases slightly to 687 milliseconds, while Microservices system response time increases moderately to 1079 milliseconds. With the concurrent user scenario at 5000, response times continue to increase, with Monolithic hitting its peak at 1081 milliseconds, while Microservices continue their downward trend at 1618 milliseconds. This indicates that Monolithic systems are responding to requests at lower latency for all loads. As per the Microservices architecture, the communication between the microservices creates an additional overhead and more roundtrips needed to process requests, hence the slow response time. However, in a Microservices architecture, as the request communicates with different services, it incurs an overhead of latency due to network communication. Also from the growing gap against the two architectures at higher load. That means, Monolithic architecture can do direct call internally within single process and Microservices architecture incurs latency due to network communication across distributed components.

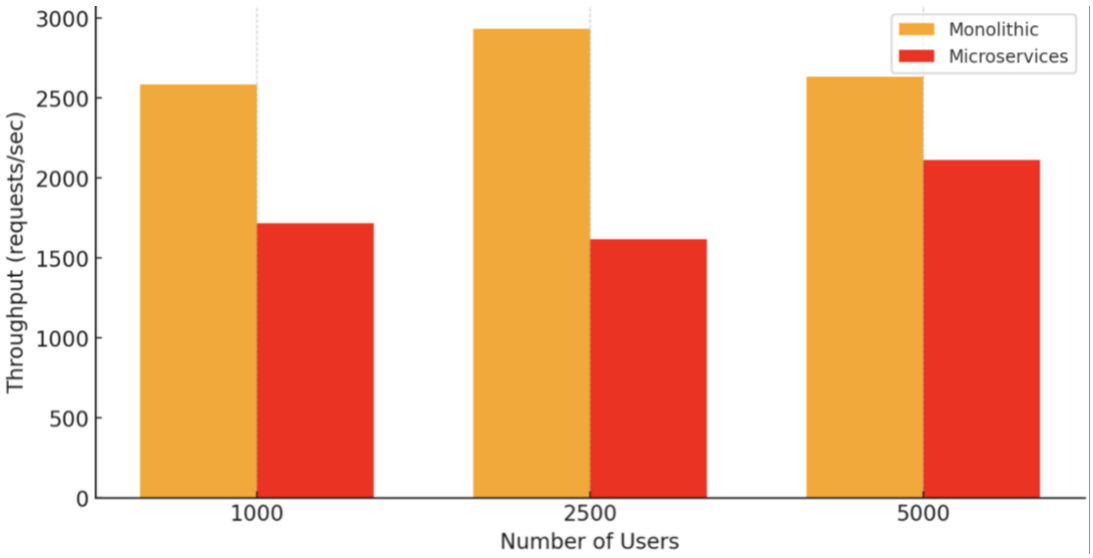

Figure 5.18: Throughput vs. Users (Monolith vs. Microservices)

However, unlike response time, which can tell us about latency in requests, we would have to use throughput to understand how many requests a system can handle in a second. Throughput graph (Figure 5.18) shows, Monolithic system can process much more requests for every load conditions described here. Monolithic handled 2584.6 requests per second whereas Microservices handled 1715.3 requests per second at 1000 users. Although Monolithic had its best top speed of all —2934.3 requests per second— at 2500 users, Microservices faced an unexpected fall of 1618.1 requests per second and this indicates that there is some bottleneck somewhere in the Microservices implementation Monolithic throughput fell slightly from 2678.0 to 2634.8 requests per second at 5000 users, while Microservices was able to recover to 2114.4 requests per second Microservice architecture throughput throughput fell as we reach 2500 users, and this indicates that overhead of inter-service communication, resource contention or inefficient load distribution matter and become significant enough impacting the performance of Microservice architecture. The Monolithic architecture appears to work for all loads very nicely. Well, Microservices are a tough one to crack on while loads are moderate but seem to work alright while under extreme levels of users.

The comparison between Monolithic Architecture and Microservices Architecture makes it clear some performnace trade-offs are involved. One clear trade-off is Monolithic architecture is way better on response time . Which Monolithic avoiding overlapped overhead of distributed service calls. Also, Monolithic achieves a higher throughput than in all scenarios including the fact that it means that the same resources=processor configuration have a greater number of requests per second from the monolithic end. Though Latency crumbles and Throughput wobbles It is the scalability characteristic of microservices architecture, That pays off in the long term. Look at the drop in throughput at 1000 users and recover of throughput at 5000 users and this also indicates once service orchestration has been optimised, resource is more aligned and communication overhead is reduced microservices architecturee which consists of small services could serve requests in a more effective way in the high loaded system.

This shows that Monolithic systems are more suitable for processing individual requests, whereas Microservices, while being able to scale better, need to do more work in optimizing communication and load balancing. Microservices shows this drop in throughput at 2500 users but tells me that bottlenecks exist and must be dealt with at a service level through caching, asynchronous messaging, et cetera. On the other hand, the Monolithic architecture provides a constant operation and therefore manifests itself to be efficient for the system which does not demand the fully distributed processing.

5. 4 Refactorability Time & Challenges

Refactoring a monolithic application into a microservices architecture is a time-consuming and complex process that involves service decomposition, database restructuring, API communication design, deployment reconfiguration, and testing. Unlike building a microservices-based system from the ground up, refactoring requires careful extraction of tightly coupled components, ensuring they function independently while still maintaining system integrity. The transition demands significant time investment at each phase, as shown in below Table 5.3.

Table 5.3: Time Spent Comparison (Monolith vs Microservice)

Stage	**Spend Time on Monolithic (hours)**	**Spend Time on Microservices (hours)**
Service Deployment & Code Structuring	6	10
Database Design & Implementation	4	6
API Development & Communication	2	4
Deployment & Infrastructure Setup	2	5
Testing & Debugging	3	5

As can be seen in Table 11, the service development and code structuring phase is one of the most time-intensive aspects of refactoring. While a monolithic system typically requires six weeks to develop as a single, unified codebase, the microservices version extends this timeframe to ten weeks. This increase in time is primarily due to the need to define service boundaries, separate business logic, and design independent service interactions. Unlike monoliths, where components communicate directly through internal function calls, microservices require well-defined API contracts, inter-service communication protocols, and fault isolation mechanisms. Additionally, service decomposition requires careful refactoring to prevent circular dependencies and minimize data duplication, further increasing the time required for this stage.

The database design and implementation phase, as presented in Table 5.3, also exhibits a significant difference in time allocation. Monolithic applications typically require four weeks for database design, as they rely on a single centralized database that serves all system components. In contrast, microservices require six weeks due to the adoption of the Database per Service pattern, which involves partitioning data across multiple independent databases. This shift presents challenges related to schema design, ensuring referential integrity, and implementing eventual consistency mechanisms. Unlike monoliths, where data consistency is managed within a single database transaction, microservices often rely on distributed transactions or event-driven synchronization models, increasing the complexity of database implementation.

As can be observed in Table 5.3, API development and inter-service communication take twice as long in a microservices architecture compared to monolithic systems. In a monolith, API calls occur internally within the same process, typically requiring two weeks to integrate. However, in a microservices-based system, APIs must be carefully designed to enable service-to-service communication, extending development time to four weeks.

The testing and debugging phase, as demonstrated in Table 5.3, also shows a considerable increase in time spent when transitioning from a monolithic to a microservices system. While monolithic applications can be tested within three weeks, microservices require five weeks due to the distributed nature of services and the need for additional validation techniques. Monolithic testing focuses on unit tests, integration tests, and end-to-end functional validation, all within a single execution environment. In contrast, microservices testing must account for service isolation, contract testing, API compatibility, and failure recovery mechanisms. Testing

must also validate how microservices interact under different latency conditions, network failures, or high-load scenarios. Debugging is inherently more complex, as logs are distributed across multiple services,

As can be seen in Table 5.3, every phase of the microservices transition demands additional time compared to the monolithic approach. The overall time spent on microservices refactoring is approximately 50-100% higher due to the challenges associated with service decomposition, database migration, inter-service communication, and deployment management. This prolonged development cycle is expected, as microservices prioritize scalability, modularity, and maintainability, which require a well-planned architecture and infrastructure setup.

Despite the increased time investment, microservices provide significant long-term advantages over monolithic architectures. While monoliths are quicker to develop and deploy, they present scalability limitations as applications grow. As demand increases, monolithic applications require vertical scaling, which can be costly and inefficient. Conversely, microservices allow independent scaling of specific components, optimizing resource utilization and operational costs over time. Additionally, fault isolation is significantly improved, as failures in one microservice do not necessarily impact the entire system, enhancing system resilience.

The transition from monolithic to microservices requires careful planning and execution to balance initial refactoring costs with long-term benefits. While the process demands substantial time and effort, it enables organizations to build more flexible, scalable, and resilient applications. Future optimizations, such as automated service discovery, AI-driven orchestration, and improved microservices development frameworks, could further reduce refactoring time, making microservices adoption more efficient.

5. 5 Hardware & Infrastructure Cost

The transition from a monolithic architecture to a microservices-based system introduces substantial changes in hardware and infrastructure costs, affecting both short-term expenses and long-term operational efficiency. While monolithic applications are often more cost-effective in terms of initial infrastructure investment, microservices architectures demand a more distributed approach, leading to increased operational overhead. However, microservices can offer better scalability, fault tolerance, and resource optimization, potentially reducing costs over time. This section analyzes the cost implications of both architectures, considering hardware utilization, cloud infrastructure expenses, scalability strategies, and total cost of ownership (TCO).

A monolithic system is typically deployed on a single powerful server or a cluster of machines, allowing it to benefit from centralized resource allocation. Hardware costs for monoliths are relatively predictable, as they primarily involve provisioning a fixed number of high-performance servers to handle processing, storage, and database management. Vertical scaling (adding more CPU, memory, or storage to an existing machine) is the main strategy for handling increased workload. While this approach keeps infrastructure management simple, it has limitations: hardware upgrades become increasingly expensive as systems approach their physical limits, leading to higher costs for maintaining performance under peak loads.

On the other hand, microservices architectures require a distributed infrastructure, where different services are deployed independently across multiple containers, virtual machines, or cloud instances. This decentralized nature means that scaling is achieved through horizontal

scaling, where new instances of a service are deployed dynamically based on demand. While this approach optimizes resource utilization and prevents overloading a single machine, it also leads to higher hardware and infrastructure costs due to the increased number of computing nodes required. Studies have shown that microservices consume 30-40% more resources compared to monolithic applications, mainly due to increased inter-service communication and the overhead of running multiple instances (Bjørndal *et al.*, 2020).

Infrastructure costs also differ significantly between the two architectures. Monolithic systems are often deployed on-premises or in a single cloud environment, requiring less network management and simpler deployment pipelines. Since all components of a monolith run in a single process space, there are fewer operational costs related to network latency, API calls, and distributed logging. In contrast, microservices architectures require robust cloud infrastructure, often relying on container orchestration platforms like Kubernetes to manage deployment, scaling, and fault tolerance. While these platforms enhance flexibility and resilience, they introduce additional expenses, including higher compute instance costs, network bandwidth fees, and container orchestration charges.

One of the biggest cost drivers in microservices is networking and inter-service communication. Since each microservice communicates with other services via APIs, message queues, or service meshes, latency and data transfer costs become significant—especially when running on cloud-based environments. A comparative study (Kamisetty *et al.*, 2023) found that microservices architectures increase network overhead by 20-50% compared to monoliths, mainly due to higher API call rates and inter-service messaging costs. Additionally, microservices deployments often require distributed logging, monitoring, and tracing solutions, further increasing operational expenses.

Despite the higher initial cost of infrastructure, microservices offer cost advantages in the long run, particularly in dynamic environments where scalability and fault isolation are critical. Since each service can scale independently, organizations can optimize resource usage by allocating compute power only to high-demand services, rather than over-provisioning resources for the entire application. Additionally, containerized workloads allow microservices to run efficiently on cloud-native platforms, reducing hardware dependency and enabling pay-as-you-go cloud pricing models.

The Total Cost of Ownership (TCO) for microservices versus monoliths varies based on business requirements, traffic patterns, and deployment strategies. A monolithic application has a lower upfront cost, making it an ideal choice for small to medium-scale applications with predictable workloads. However, as applications grow, monoliths become difficult to scale efficiently, leading to higher infrastructure costs due to the need for expensive vertical scaling solutions. Microservices, while costlier in the early stages, offer better long-term efficiency, especially in cloud-native environments where scalable, distributed computing reduces overall operational expenses over time ((Auer *et al.*, 2021).

Overall, the choice between monolith and microservices depends on cost-efficiency trade-offs. While monolithic architectures are more cost-effective in the short term, they struggle with scalability and hardware constraints as demand increases. Microservices require higher initial investment in hardware and infrastructure, but their flexibility, fault isolation, and optimized resource utilization make them a viable long-term strategy for large-scale applications.

Table 5.4: Cost Analysis Table (Monolithic vs. Microservices)

Cost Factor	Monolithic Architecture	Microservices Architecture
Server Requirements	Requires a few high performance servers.	Uses multiple smaller instances for each service.
Scaling Approach	Vertical scaling (adding more CPU/RAM to a single machine).	Horizontal scaling (adding more instances to distrube load).
Database Infrastructure	Single database instance for all operations	Multiple database per service, increasing storage and maintenance costs.
Cloud Hosting Cost	Lower initial costs but may increase as demand grows.	Higher initial cloud cost due to multiple services running concurrently.
Network & Communication	Minimal internal communication costs.	Higher networking costs due to service to service communication.
Monitoring & Security	Centralized logging and secutiy management.	Requires distributed monitoring tools.
Operational Complexity	Lower ahead	Higher complexity due to independent service deployment.

5. 6 Discussion

The results obtained from the transition from a monolithic to a microservices architecture demonstrate notable differences in performance, scalability, refactorability, and cost efficiency. As expected, microservices provided better scalability and distributed resource utilization, while monolithic architectures showed efficiency in lower overhead costs and simpler deployment. However, the results also revealed certain unexpected trends, particularly in CPU utilization, database migration complexity, and refactoring time. This discussion critically analyzes these findings, providing justifications for observed patterns and comparing them with literature to support the interpretation.

The performance metrics analysis revealed that CPU and memory utilization behaved differently across both architectures. As seen in Table 5.1, the monolithic application experienced CPU bottlenecks at high concurrent loads, reaching 490% CPU utilization under 1000 users and eventually dropping at 5000 users, likely due to request failures, system overload, or process throttling. In contrast, microservices exhibited more balanced CPU distribution, but some services, such as the Order and Cart services, showed high resource consumption under heavy load. This behavior aligns with findings from Bucchiarone et al. (2020), which highlighted that while microservices enable independent scaling, some services become performance bottlenecks due to high inter-service communication and database query loads. The unexpected drop in CPU usage for monolithic architecture at 5000 users may be attributed to system limitations that led to request failures, preventing full CPU utilization.

Memory utilization followed a similar trend. As observed in Table 5.2, monolithic applications showed a steady increase in memory consumption, reaching 984MB at 5000 users, while microservices exhibited more distributed memory allocation. However, some microservices, particularly those handling frequent database interactions, experienced higher-than-expected memory usage. This can be explained by service-specific caching, repeated data queries due to

service isolation, and overhead from API calls. This result is consistent with studies such as (Auer *et al.*, 2021), which noted that microservices tend to consume 30-40% more memory compared to monoliths due to duplication of dependencies and container overhead.

The refactorability analysis presented in Table 5.3 highlights another key challenge—the transition from monolithic to microservices required significantly more time across all stages. While service development in a monolith required only six weeks, the same stage in microservices took ten weeks, largely due to service decomposition, API restructuring, and ensuring inter-service communication. Additionally, database migration proved to be one of the most time-intensive phases, requiring six weeks for microservices compared to four weeks in monolithic architecture. This increase is attributed to the complexity of breaking down a shared database into service-specific databases and handling consistency through distributed transactions or event-driven synchronization. Database restructuring is one of the most difficult parts of microservices adoption, particularly for legacy monolithic applications. The results support this assertion, as ensuring referential integrity, normalizing schema changes, and handling foreign key relationships added to refactoring complexity.

One of the unexpected findings was that microservices introduced additional network overhead, leading to increased latency for API interactions. Although microservices were expected to improve performance through independent scaling, some services experienced higher-than-anticipated response times, particularly Order Processing and Cart Services. This can be attributed to higher inter-service communication latency, additional network hops, and serialization/deserialization overhead. According to (Kamisetty *et al.*, 2023)microservices architectures inherently introduce 20-50% additional network latency compared to monoliths due to increased service-to-service calls. These findings suggest that optimizing service communication through caching, batch processing, or asynchronous messaging could reduce network overhead.

Despite the increased complexity and higher initial costs, the transition to microservices ultimately provides greater scalability, resilience, and modular flexibility. The fault isolation capabilities of microservices prevent a failure in one service from affecting the entire system, enhancing reliability. Additionally, independent deployment pipelines allow faster release cycles, enabling continuous integration and continuous deployment (CI/CD) strategies. However, the findings also highlight that organizations must carefully assess their architecture before migrating, as microservices introduce operational overhead that requires a well-structured DevOps approach.

In conclusion, the discussion of results indicates that while microservices improve scalability and modularity, they introduce challenges related to refactorability, infrastructure costs, and network performance. The results align with existing research, reinforcing that microservices are not a one-size-fits-all solution—they work best for large-scale, high-traffic applications requiring independent scalability, whereas monoliths remain viable for smaller applications with predictable workloads. Future work should explore optimization strategies for inter-service communication, database partitioning techniques, and cost-efficient cloud infrastructure management to further refine the efficiency of microservices adoption.

Chapter 6. Conclusion and Future Work

6. 1 Conclusion

The study performed a transition from a monolithic architecture to a microservices-based system emphasizing refactorability, scalability, and performance and adopting a shared database model. The results show that microservices significantly enhance the system capability to scale up and tolerate faults, facilitating modularization and individual service deployments. However, the adoption of the shared database model brings its own set of challenges, resulting in scalability bottlenecks and database contention issues (Paccha & Velepucha, 2025). Although scalability is affected, a shared database enables data consistency among the services, which is a paramount requirement in real-world applications.

An important highlight of the study is that, in as much as scalability is inherent in microservices architectures, database design is pertinent in keeping the system efficient. Findings indicated that monolithic architectures are not able to withstand loads when they increase but offer better horizontal scalability compared to microservices architectures. The latter introduces an operational overhead compared to the former by requiring additional infrastructure like service discovery, API Gateway, and Distributed logging system (Tian *et al.*, 2024). Moreover, Strangler Fig Pattern offered a way to gradually migrate a monolithic system with minimum downtime by decoupling monolithic components incrementally into loosely coupled services.

The shared database model provided both benefits and limitations, with the benefit of ensuring data consistency and minimizing migration complexity, but at the expense of limiting microservice scalability potential since highly concurrent read/write operations generally are not possible (Amrutha, Jayalakshmi and Geetha, 2024). Additionally, the research shows that microservices must be carefully decomposed to avoid incurring too much overhead in API communications, as this can hinder performance. Overall, the results suggest that organisations looking to adopt microservices must carefully evaluate their database strategy and optimise inter-service communication to achieve an optimal balance of performance, maintainability and scalability.

While this study provides valuable insights into the implementation of microservices architecture with a shared database architecture, it is not without limitations. The study was conducted in a controlled environment, and factors such as network latency, cloud re-platforming costs, and real-world traffic patterns were not adequately addressed. In addition, the study also did not consider alternative database models in microservices architecture, such as database-per-service, CQRS (Command Query Responsibility Segregation) and event-driven architecture, etc., which can further improve microservices architecture efficiency. Security concerns were also outside the scope of the study, such as service authentication/ authorization mechanisms and API security. Future work should examine these aspects in greater detail, to provide a more comprehensive evaluation of microservices-based architectures.

6. 2 Future Work

The paper interpreted the future work which is to compare and contrast different strategies of database in microservices architectures. Further, the assessment of consistency, performance and scalability in terms of the shared database model and database-per-service based model is required by future research. Implementing event-drive architectures with Apache Kafka or RabbitMQ might alleviate some of the database contention problems observed in this research, as the communication between the services is asynchronous (Paccha and Velepucha, 2025). CQRS can be utilised to decompose the system further optimizing the system even in high load scenario by separating the read and write paths.

Another important direction is Automating Monolith-to-Microservices Migration. Despite the advancements, monolith-to-microservices migrations still require significant manual effort, ultimately adding development time and complexity. Future work could explore automatic service decomposition tools powered by AI which could study monolithic systems and automatically separate out loosely coupled microservices. In addition, an automated testing framework for microservices migrations may be created to solutions performance regressions and integration failures early in the migration process .

Next, it is also crucial to deploy the project live to evaluate the scalability in the wild. This paper highlighted scalability of the project in a controlled local environment, but deploying it online will expose the project to realistic user traffic and clients with distribution of work which could all help us observe its real-time performance. In addition, real online deployment test results would help us understand how specific cloud infrastructures perform (e.g., AWS vs. GCE), the impact of network latency, and the cost-performance tradeoff during deployment. Existing cloud-based solutions can be used such as Kubernetes, AWS, Google Cloud or Azure so that the microservices can be dynamically scaled up and down to the maximum level and be tested on how they perform in production kind of environments.

Alternatively, you could explore performance optimization techniques. Consider Cache: Exploring caching mechanisms like Redis, Memcached can help reduce database load and enhancements response time in microservices. On top of that there are also possibility to research the horizontal scaling strategies with Kubernetes orchestration in order to utilize the resources more efficiently in cloud native environments. It is also possible in serverless computing strategies such as AWS Lambda, Azure Functions, GCP Cloud Functions for microservices-based strategies (Amrutha, Jayalakshmi and Geetha, 2024).

In-depth focus on security is needed, as well. In the future work this should be expanded into service to service authentication methods (OAuth, JWT, and API security best practices. It might also be worth looking into the use of service mesh technologies like Istio, and Linkerd, which help with the security, observability, and traffic management among microservices. Last but not the least, one can also have failure recovery mechanisms like circuit breaker, retries and distributed tracing to achieve.

Lastly, it would be quite useful to illustrate real case studies how these findings are put into practice at an enterprise scale. We recommend that future work continue and broaden efforts to enabling longitudinal microservices studies, enabling researchers to longitudinally assess microservices architectures in the facets of maintainability, operational costs, and longer-term business impacts of adopting microservices – thus informing organizations whether and how

they should migrate their monolithic systems to microservices architectures with cost-effective and performance-efficient performance scalability in mind.

References

Alcides Mora Cruzatty, A. *et al.* (2024) 'Assessment of Container Orchestration Strategies in the Migration of Monolithic Applications to a Microservices Architecture Using Open-Source Technologies', in M.Z. Vizuete et al. (eds) *Applied Engineering and Innovative Technologies*. Cham: Springer Nature Switzerland, pp. 83–96. Available at: https://doi.org/10.1007/978-3-031-70760-5_7.

Al-Debagy, O. and Martinek, P. (2018) 'A Comparative Review of Microservices and Monolithic Architectures', in *ResearchGate*. Available at: https://doi.org/10.1109/CINTI.2018.8928192.

Ali, J.M. (2024) 'Software engineering architecture and its promising opportunities', *Advances in Engineering Innovation*, 7, pp. 37–40. Available at: https://doi.org/10.54254/2977-3903/7/2024034.

AlOmar, E.A., Mkaouer, M.W. and Ouni, A. (2024) 'Behind the Intent of Extract Method Refactoring: A Systematic Literature Review', *IEEE Transactions on Software Engineering*, 50(4), pp. 668–694. Available at: https://doi.org/10.1109/TSE.2023.3345800.

Amrutha, L., Jayalakshmi, D.S. and Geetha, J. (2024) 'Enhancing Deployment and Performance Measurement of Serverless Cloud Microservices with Warm Start', in *2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT). 2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pp. 1–7. Available at: https://doi.org/10.1109/ICCCNT61001.2024.10725992.

Apache JMeter - Apache JMeter™ (no date). Available at: https://jmeter.apache.org/ (Accessed: 23 February 2025).

Apache Kafka (no date) *Apache Kafka*. Available at: https://kafka.apache.org/documentation/ (Accessed: 23 February 2025).

Ataei, P. (2024) 'Cybermycelium: a reference architecture for domain-driven distributed big data systems', *Frontiers in Big Data*, 7. Available at: https://doi.org/10.3389/fdata.2024.1448481.

Auer, F. *et al.* (2021) 'From monolithic systems to Microservices: An assessment framework', *Information and Software Technology*, 137, p. 106600. Available at: https://doi.org/10.1016/j.infsof.2021.106600.

Barzotto, T.R.H. and Farias, K. (2022) *Assessing the impacts of decomposing a monolithic application for microservices: A case study*, *ResearchGate*. Available at: https://doi.org/10.48550/arXiv.2203.13878.

Bashtovyi, A. and Fechan, A. (2024) 'Distributed Transactions in Microservice Architecture: Informed Decision-making Strategies', *Vìsnik Nacìonal'nogo unìversitetu 'L'vìvs'ka polìtehnìka'. Serìâ Ìnformacìjnì sistemi ta merežì*, 15, pp. 449–459. Available at: https://doi.org/10.23939/sisn2024.15.449.

Baumgartner, J.K. (2022) 'From Monolith to Microservices'.

Berry, V. *et al.* (2024) 'Is it Worth Migrating a Monolith to Microservices? An Experience Report on Performance, Availability and Energy Usage', in *2024 IEEE International Conference on Web Services (ICWS). 2024 IEEE International Conference on Web Services (ICWS)*, pp. 944–954. Available at: https://doi.org/10.1109/ICWS62655.2024.00112.

Bhatnagar, S. and Mahant, R. (2024) *Fortifying Financial Systems: Exploring the Intersection of Microservices and Banking Security*, *ResearchGate*. Available at: https://doi.org/10.13140/RG.2.2.13110.72001.

Bjørndal, N. *et al.* (2020) *Migration from Monolith to Microservices : Benchmarking a Case Study*, *ResearchGate*. Available at: https://doi.org/10.13140/RG.2.2.27715.14883.

Chaieb, M., Sellami, K. and Saied, M.A. (2023) ‘Migration to Microservices: A Comparative Study of Decomposition Strategies and Analysis Metrics’.
Chen, M. *et al.* (2024) ‘TraDE: Network and Traffic-aware Adaptive Scheduling for Microservices Under Dynamics’. arXiv. Available at: https://doi.org/10.48550/arXiv.2411.05323.
Curnicov, A. (2025) *Research on microservices architecture for an Automated Surveillance System*. Thesis. Universitatea Tehnică a Moldovei. Available at: https://repository.utm.md/handle/5014/29200 (Accessed: 19 February 2025).
Dragoni, N. *et al.* (2017) ‘Microservices: yesterday, today, and tomorrow’, in *ResearchGate*. Available at: https://www.researchgate.net/publication/315664446_Microservices_yesterday_today_and_tomorrow (Accessed: 5 February 2025).
El Akhdar, A., Baidada, C. and Kartit, A. (2024) ‘Adaptability of Microservices Architecture in IoT Systems : A Comprehensive Review’, in *Proceedings of the 7th International Conference on Networking, Intelligent Systems and Security*. New York, NY, USA: Association for Computing Machinery (NISS ’24), pp. 1–9. Available at: https://doi.org/10.1145/3659677.3659734.
Fowler, M. and Beck, K. (2019) *Refactoring: improving the design of existing code*. Second edition. Boston Columbus New York San Francisco Amsterdam Cape Town Dubai London Munich: Addison-Wesley (The Addison-Wesley signature series).
Gandhi, H. and Vashishtha, S. (2025) *IMPLEMENTING SCALABLE MICROSERVICES FOR BIG DATA PROCESSING IN CLOUD ENVIRONMENTS*, *ResearchGate*. Available at: https://doi.org/10.56726/IRJMETS66275.
Garimilla, M. (2024) *Microservices Architecture: Revolutionizing Modern Software Development*, *ResearchGate*. Available at: https://doi.org/10.15680/IJIRSET.2024.1309090|.
Goniwada, S.R. (2022) *Cloud Native Architecture and Design: A Handbook for Modern Day Architecture and Design with Enterprise-Grade Examples*. Berkeley, CA: Apress. Available at: https://doi.org/10.1007/978-1-4842-7226-8.
González, S. and Ortiz, I. (2024) ‘Overcoming Challenges in Microservice Architectures’, *ResearchGate* [Preprint]. Available at: https://www.researchgate.net/publication/386219405_Overcoming_Challenges_in_Microservice_Architectures (Accessed: 18 February 2025).
Harris, C. (no date) *Microservices vs. monolithic architecture*, *Atlassian*. Available at: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith (Accessed: 12 February 2025).
Hassan, H., Abdel-Fattah, M.A. and Mohamed (2024) ‘Migrating from Monolithic to Microservice Architectures: A Systematic Literature Review’, *ResearchGate* [Preprint]. Available at: https://doi.org/10.14569/IJACSA.2024.0151013.
Jani, Y. (2024) *Unified Monitoring for Microservices: Implementing Prometheus and Grafana for Scalable Solutions*, *ResearchGate*. Available at: https://doi.org/10.51219/JAIMLD/yash-jani/206.
Kamisetty, A. *et al.* (2023) ‘Microservices vs. Monoliths: Comparative Analysis for Scalable Software Architecture Design’, *ResearchGate* [Preprint]. Available at: https://doi.org/10.18034/ei.v11i2.734.
Karwatka, P. (2020) *Monolithic architecture vs microservices*, *Cloudflight*. Available at: https://www.cloudflight.io/en/blog/monolithic-architecture-vs-microservices/ (Accessed: 23 February 2025).
Kassetty, N. and Chippagiri, S. (2025) *Beyond the Monolith: Comprehensive Strategies for Architecting, Scaling, and Sustaining Resilient Distributed Systems*, *ResearchGate*. Available at: https://doi.org/10.34218/IJRCAIT_08_01_016.

Khakame, P.W. (2016) *Development of a scalable microservice architecture for web services using os-level virtualization*. Thesis. University of Nairobi. Available at: http://erepository.uonbi.ac.ke/handle/11295/99091 (Accessed: 14 February 2025).
Kristiyanto, D.Y. *et al.* (2024) 'Comprehensive Framework for Transitioning Monolithic to Microservices in MVC Context', in *2024 3rd International Conference on Creative Communication and Innovative Technology (ICCIT)*. *2024 3rd International Conference on Creative Communication and Innovative Technology (ICCIT)*, pp. 1–7. Available at: https://doi.org/10.1109/ICCIT62134.2024.10701144.
Maj, J., Zielony, P. and Piotrowski, K. (2024) 'Migrating WSN Applications from Monolithic to a Modular Approach Based on the tinyDSM Middleware: Scenarios and Analysis', in *2024 IEEE Conference on Pervasive and Intelligent Computing (PICom)*. *2024 IEEE Conference on Pervasive and Intelligent Computing (PICom)*, pp. 119–124. Available at: https://doi.org/10.1109/PICom64201.2024.00023.
Manchana, R. (2021) 'Balancing Agility and Operational Overhead: Monolith Decomposition Strategies for Microservices and Microapps with Event-Driven Architectures', *North American Journal of Engineering Research*, 2(2). Available at: https://najer.org/najer/article/view/20 (Accessed: 14 February 2025).
Mehta, G. *et al.* (2024) 'Revisiting Monoliths: A Pragmatic Case for Transitioning from Microservices Back to Monolithic Architectures', *ResearchGate* [Preprint]. Available at: https://doi.org/10.17148/IJARCCE.2024.131251.
Microservices Pattern: Pattern: API Gateway / Backends for Frontends (no date) *microservices.io*. Available at: http://microservices.io/patterns/apigateway.html (Accessed: 8 February 2025).
Microservices Pattern: Pattern: Shared database (no date) *microservices.io*. Available at: http://microservices.io/patterns/data/shared-database.html (Accessed: 8 February 2025).
Monolithic vs Microservices - Difference Between Software Development Architectures- AWS (2024) *Amazon Web Services, Inc.* Available at: https://aws.amazon.com/compare/the-difference-between-monolithic-and-microservices-architecture/ (Accessed: 12 February 2025).
Monoliths to Microservices using the Strangler Pattern (no date) *Amplication Blog*. Available at: https://amplication.com/blog/monoliths-to-microservices-using-the-strangler-pattern (Accessed: 8 February 2025).
Muley, Y. (2024) 'Comparative Analysis of Monolithic and Microservices Architectures in Financial Software Development', *Journal of Artificial Intelligence, Machine Learning and Data Science*, 2(4), pp. 1846–1848. Available at: https://doi.org/10.51219/JAIMLD/Yogesh-muley/408.
Nassima, A.M., Hanae, S. and Karim, B. (2024) 'Towards Decomposing Monolithic Applications into Microservices: Dynamic Analysis', in Y. Mejdoub and A. Elamri (eds) *Proceeding of the International Conference on Connected Objects and Artificial Intelligence (COCIA2024)*. Cham: Springer Nature Switzerland, pp. 99–104. Available at: https://doi.org/10.1007/978-3-031-70411-6_16.
Newman, S. (2019) 'Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith'.
Nitin, V. *et al.* (2023) 'CARGO: AI-Guided Dependency Analysis for Migrating Monolithic Applications to Microservices Architecture', in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery (ASE '22), pp. 1–12. Available at: https://doi.org/10.1145/3551349.3556960.
Owen, A. (2025) *Microservices Architecture and API Management: A Comprehensive Study of Integration, Scalability, and Best Practices*, *ResearchGate*. Available at:

https://www.researchgate.net/publication/388952031_Microservices_Architecture_and_API_Management_A_Comprehensive_Study_of_Integration_Scalability_and_Best_Practices (Accessed: 18 February 2025).
Paccha, P.M. and Velepucha, V.V. (2025) 'Data Domain Servitization for Microservices Architecture', *Latin-American Journal of Computing*, 12(1), pp. 59–67.
Perry, M. (2023) *5 Tips For Managing Your Internal Developer Platform*. Available at: https://www.qovery.com/blog/5-tips-for-managing-your-internal-developer-platform/ (Accessed: 26 February 2025).
Powell, P. and Smalley, I. (2024) *Monolithic vs. Microservices Architecture | IBM*. Available at: https://www.ibm.com/think/topics/monolithic-vs-microservices (Accessed: 27 February 2025).
Ramachandran, N. and Thirumaran, M. (2024) 'A Novel Approach for Dynamic Microservices Composition: Harnessing the Power of the PMCE Framework', in *2024 International Conference on Signal Processing, Computation, Electronics, Power and Telecommunication (IConSCEPT)*. *2024 International Conference on Signal Processing, Computation, Electronics, Power and Telecommunication (IConSCEPT)*, pp. 1–6. Available at: https://doi.org/10.1109/IConSCEPT61884.2024.10627913.
Salaheddin Elgheriani, N. and Ali Salem Ahme, N.D. (2022) 'MICROSERVICES VS. MONOLITHIC ARCHITECTURES [THE DIFFERENTIAL STRUCTURE BETWEEN TWO ARCHITECTURES]', *MINAR International Journal of Applied Sciences and Technology*, 4(3), pp. 500–514. Available at: https://doi.org/10.47832/2717-8234.12.47.
Salunkhe, V. *et al.* (2024) 'Leveraging Microservices Architecture in Healthcare: Enhancing Agility and Performance in Clinical Applications'. Rochester, NY: Social Science Research Network. Available at: https://doi.org/10.2139/ssrn.4985002.
Samant, P.S. (2024) *MICROSERVICES IN THE CLOUD: ENABLING SCALABILITY, FLEXIBILITY, AND RAPID DEPLOYMENT*, *ResearchGate*. Available at: https://www.researchgate.net/publication/381306736_MICROSERVICES_IN_THE_CLOUD_ENABLING_SCALABILITY_FLEXIBILITY_AND_RAPID_DEPLOYMENT (Accessed: 14 February 2025).
Santos, L. *et al.* (2024) 'Microfront-End: Systematic Mapping':, in *Proceedings of the 20th International Conference on Web Information Systems and Technologies*. *20th International Conference on Web Information Systems and Technologies*, Porto, Portugal: SCITEPRESS - Science and Technology Publications, pp. 119–130. Available at: https://doi.org/10.5220/0013015400003825.
Santos, T.C. (2018) *Adopting Microservices*.
Seedat, M. *et al.* (2023) *Systematic Mapping of Monolithic Applications to Microservices Architecture*, *ResearchGate*. Available at: https://doi.org/10.22541/au.168110476.68608378/v1.
Sethi, S. and Panda, S. (2024) 'Transforming Digital Experiences: The Evolution of Digital Experience Platforms (DXPs) from Monoliths to Microservices: A Practical Guide', *Journal of Computer and Communications*, 12(2), pp. 142–155. Available at: https://doi.org/10.4236/jcc.2024.122009.
Shao, P. *et al.* (2024) 'Design and Implementation of an Electricity Market Trading Platform Architecture for High Concurrency Access by Market Entities', in *2024 IEEE 6th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC). 2024 IEEE 6th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, pp. 746–751. Available at: https://doi.org/10.1109/IMCEC59810.2024.10575639.

Soma (2024) *Horizontal scaling vs Vertical Scaling in System Design*, *DEV Community*. Available at: https://dev.to/somadevtoo/horizontal-scaling-vs-vertical-scaling-in-system-design-3n09 (Accessed: 13 February 2025).
Strangler fig pattern - AWS Prescriptive Guidance (no date). Available at: https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/strangler-fig.html (Accessed: 8 February 2025).
Sulkava, A. (2023) 'Building scalable and fault-tolerant software systems with Kafka'.
Sun, Y. (no date) 'A Comparative Study of Application Performance and Resource Consumption between Monolithic and Microservice Architectures'.
Tapia, F. *et al.* (2020) 'From Monolithic Systems to Microservices: A Comparative Study of Performance', *Applied Sciences*, 10(17), p. 5797. Available at: https://doi.org/10.3390/app10175797.
Thatikonda, V.K. and Mudunuri, H.R.V. (2024) 'Microservices vs. Monoliths: Choosing the Right Architecture for Your Project', *International Journal of Software Computing and Testing* [Preprint].
Tian, T. *et al.* (2024) 'Design and Application of Database Architecture with Super Large-Scale for Marketing Service System of Energy Internet in Enterprise Digital Transformation', in *2024 3rd International Conference on Energy and Electrical Power Systems (ICEEPS)*. *2024 3rd International Conference on Energy and Electrical Power Systems (ICEEPS)*, pp. 275–285. Available at: https://doi.org/10.1109/ICEEPS62542.2024.10693085.
Wang, Y. (2024) 'Optimizing Payment Systems with Microservices and Event-Driven Architecture: The Case of Mollie Platform'.
ZakerZavardehi, H. (2024) *A Semi-Automated Approach for Incremental Migration from Monolithic to Microservices Architecture*. Thesis. Available at: https://macsphere.mcmaster.ca/handle/11375/30255 (Accessed: 18 February 2025).