OPTIMIZED STRATEGY IN CLOUD-NATIVE ENVIRONMENT FOR INTER-SERVICE COMMUNICATION IN MICROSERVICES

L.D.S.B Weerasinghe

208092X

Doctor of Philosophy

Department of Computer Science and Engineering
Faculty of Engineering

University of Moratuwa
Sri Lanka

May 2025

OPTIMIZED STRATEGY IN CLOUD-NATIVE ENVIRONMENT FOR INTER-SERVICE COMMUNICATION IN MICROSERVICES

L.D.S.B Weerasinghe

208092X

Dissertation submitted in partial fulfillment of the requirements for the degree

Doctor of Philosophy

Department of Computer Science and Engineering

Faculty of Engineering

University of Moratuwa

Sri Lanka

May 2025

DECLARATION

I declare that this is my own work, and this dissertation does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any other University or Institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: Date: 20/05/2025

The above candidate has carried out research for the PhD dissertation under my supervision. I confirm that the declaration made above by the student is true and correct.

Name of Supervisor: Prof. Indika Perera

Signature of the Supervisor: Date: 20.05.2025

Dedicated to my family for their constant love, support, and encouragement throughout my journey.

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my supervisor, Prof. Indika Perera, for his invaluable support and guidance throughout my research journey. His commitment to my work, coupled with his constructive feedback and insightful criticisms, has been instrumental in shaping my research skills and helping me navigate this complex field. Prof. Indika's support in publishing my work in reputed conferences and journals has been a cornerstone in advancing my academic and professional growth.

I am deeply appreciative of the time and effort dedicated by my examiners, Dr. Amal Shehan Perera and Dr. Kutila Gunasekera, whose constructive feedback during my progress reviews has been invaluable. Their insights have helped refine and strengthen the contributions of this thesis.

I would also like to extend my gratitude to Axiata Digital Labs for the flexibility that allowed me to take academic responsibilities. This support has been invaluable in my ability to pursue my research with dedication.

My heartfelt thanks go to the Head of the Department of Computer Science and Engineering, the Dean of the Faculty of Engineering, and the Dean of the Faculty of Postgraduate Studies for their steadfast support and guidance through the administrative aspects of my PhD journey. I also wish to extend my appreciation to the staff of the postgraduate division and the establishment division for their assistance and friendliness in handling the procedural aspects of my studies.

Finally, I am deeply grateful to my loving mother and father, whose unwavering belief in my potential and constant encouragement have been the foundation of my success. Their understanding and patience throughout my PhD journey, even when my commitments took me away from family obligations, have been a source of strength. To my wife, my heartfelt thanks for her steadfast support, encouragement, and understanding as I dedicated long hours to this work. Her empathy and sacrifices have been invaluable in helping me persevere through the challenges of this journey. This accomplishment would not have been possible without the unconditional love and support of my family.

ABSTRACT

Microservices architecture has become the most popular architecture for building scalable and flexible applications due to its inherent advantages over monolithic and Service-Oriented Architecture (SOA) systems. It allows developers to scale individual components independently, improve maintainability through smaller codebases, and increase resilience by isolating service failures. Additionally, microservices are well-suited for cloud-native environments, supporting containerization and orchestration platforms. However, transforming monolithic systems into microservice-based systems introduces challenges with inter-service communication, leading to performance bottlenecks and increased latency due to the decentralized and distributed nature of the architecture. This research addresses the above challenge by proposing a novel communication model for cloud-native environments, focusing on optimizing inter-service communication in microservice architecture. The proposed strategy operates on top of the TCP network layer rather than the application layer, leveraging TCP streams but using a request-response-based communication model. All distributed systems typically rely on TCP-based communication over a network. To reduce the overhead of opening and closing socket connections, the proposed strategy persists the TCP connection when a microservice starts and maintains it until the service goes down. This solution reduces network resource consumption by serializing messages into binary form, thereby enhancing response times and application throughput. When sending messages over the TCP connection, we have established a method for ensuring exact-once delivery, improving the reliability and efficiency of message transfer. The proposed solution was implemented within a microservice architecture and deployed in a cloud-native environment to assess its effectiveness for cloud-native applications. The solution has been evaluated against the traditional HTTP-based communication method in the same microservice-architected environment and has shown improved performance in terms of latency and overall throughput. This research offers a robust communication model for microservices that addresses performance challenges while maintaining compatibility with cloud-native concepts. The proposed approach is a step toward optimizing microservices communication, enabling developers to build scalable, efficient, and resilient applications in distributed environments.

Keywords: Microservices, Inter-service communication, Performance

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	Description
OOA	Object-Oriented Architecture
SOA	Service Oriented Architecture
ESB	Enterprise Service Bus
IaaS	Infrastructure as a Service
SaaS	Software as a Service
PaaS	Platform as a Service
FaaS	Function as a Service
TPS	Transactions per second
API	Application Programming Interfaces
JMS	Java Message Service
XML	Extensible Markup Language
WSDL	Web Services Description Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
RPC	Remote Procedure Call
TCP	Transmission Control Protocol
IP	Internet Protocol
REST	Representational State Transfer
RESP	Redis Serialization Protocol
LLD	Low Level Diagram
VM	Virtual Machine
JVM	Java Virtual Machine
GCP	Google Cloud Platform
GKE	Google Kubernetes Engine
CI	Continues Integration
CD	Continues Delivery

CHAPTER 1

INTRODUCTION

Before the widespread use of software, people relied on manual methods to accomplish tasks and conduct daily activities. Record-keeping involved handwritten ledgers, where businesses meticulously documented sales, expenses, and inventory. Communication was slower, primarily relying on letters, phone calls, or face-to-face interactions, without the instant connectivity provided by today's messaging apps and emails. Complex calculations were performed by hand or using mechanical calculators, lacking the advanced spreadsheet software we have today. Research meant physically going to libraries, sifting through books, and referencing documents, as there was no internet for quick online information retrieval. Creative work, such as art and design, utilized traditional tools like pencils and brushes, with no digital aids for graphic design or computer-aided drafting. In essence, work was more labor-intensive and time-consuming, and software has revolutionized these processes, bringing about efficiency and accessibility in various facets of daily life and professional endeavors that depend upon information processing.

Today's world is full of people interested in using new technologies. This has made our society very dependent on information systems - things that help us manage and use information. People want to perform daily tasks in the simplest and fastest possible way. Due to this high demand for efficiency, software engineers realized that building systems that solve real problems was super important. In order to address these challenges, the software industry began to investigate and propose new ideas for how to design software. The goal is to create the more reliable software in the world. These new software design methods help us better understand how software works, provide services, and improve its lifestyle. To start this process, engineers created a type of design called Object-Oriented Architecture (OOA). Object-oriented architecture designs software by organizing and structuring code to resemble real-world objects and how they interact [1]. It follows critical principles like encapsulation, inheritance, and polymorphism. In this approach, a software system comprises objects like examples of classes. A class acts as a blueprint, defining objects' characteristics (attributes) and actions (methods). Objects can communicate through well-defined interfaces, promoting easy-to-understand and reusable code. Encapsulation bundles together data and methods into a single unit for better management. Inheritance allows a new class to inherit features from an existing class, promoting code reuse. Polymorphism lets objects from different classes be treated as belonging to a standard base class, offering flexibility and expandability [2]. Object-oriented architecture is widely used for designing complex and scalable software systems. It promotes a modular and organized code structure, making maintaining and modifying applications easier over time. Popular programming languages like Java, C++, and Python extensively utilize object-oriented principles in their design [3].

The evolution of programming languages is a dynamic and multifaceted process shaped by various factors. Technological advances, such as changes in hardware architecture and the introduction of new computing platforms, significantly influence the development of programming languages. Paradigm shifts, including the adoption of object-oriented and functional programming, contribute to creating languages that embody these concepts [4]. The preferences of the developer community and industry trends also play a crucial role, with languages gaining popularity based on their alignment with community needs and the availability of supporting libraries and frameworks. Moreover, the emergence of problem-specific languages designed for specific domains or applications reflects the demand for more expressive and abstract tools. Improvements in language features, syntax, and expressiveness contribute to the evolution of programming languages as designers incorporate new concepts and constructs to enhance developer productivity. Additionally, the growing importance of cybersecurity has led to integrating security features in programming languages to mitigate vulnerabilities. The open-source movement and collaborative development efforts worldwide have significantly shaped and enhanced programming languages. Education and training needs also impact language development, leading to the creation of beginner-friendly languages and environments. Finally, the availability of a robust ecosystem, including libraries, frameworks, and development tools, contributes to programming languages' success and ongoing evolution. Overall, programming languages evolve in response to the ever-changing landscape of technology, the needs of developers, and the demands of modern software development [5].

1.1 Software Architecture

Software architecture is one of the most important parts of software engineering. The high-level structuring of software systems is a composite of the set of major decisions about the organization of a software system, the selection of structural elements, and their interfaces and behavior. In essence, software architecture is a blueprint of a system guiding design, implementation, and evolution [6]. It is important for a variety of reasons. First, it gives developers a clear roadmap to follow, ensuring continuity of vision and goals throughout the project. This is critical to maintain coherence and consistency during the development process. Second, it handles complexity, breaking down the system into manageable parts, which are easier to understand, develop, and maintain. This modularity not only makes development easier but also enhances the scalability and flexibility of the system. Third, software architecture provides a vehicle for meeting requirements that are not related to functionality, such as performance, security, and reliability. It ensures that the system meets the most critical quality standards that are prescribed, and it is done by first addressing these aspects right from the design stage. In addition, software architecture provides a common language and framework through which the structure and behavior of the system can be expressed and understood by all stakeholders. Common understanding facilitates coordination

and aligns expectations. Last, it supports decision-making about design trade-offs and resource allocation. By considering various architectural options and their implications, architects can make informed choices that balance various constraints and requirements. In all, a well-defined software architecture lays a solid foundation not only for the current system but also ensures its ability to adapt and evolve in response to future challenges and opportunities. Academia and the software industry both play crucial roles in advancing the field of software architecture. Academia contributes through research, education, and the development of theoretical frameworks, investigating new paradigms and methodologies such as model-driven architecture, microservice architecture, and cloud native architectures [7]. Academic institutions educate the next generation of software architects, ensuring they enter the industry with a solid understanding of fundamental principles and best practices. Collaborative projects between academic institutions and industry partners provide real-world contexts for applying and testing new architectural theories and models, facilitating knowledge transfer and ensuring that academic insights are grounded in practical experience. The software industry, on the other hand, applies architectural principles in the development of commercial software products and systems, validating and refining theoretical concepts. Industry professionals drive the development of tools and platforms that support architectural design and analysis, such as integrated development environments (IDEs) and performance analysis tools [8]. Industry organizations and consortia develop standards and best practices, ensuring consistency and quality across projects, while professional development opportunities such as certifications and training programs validate expertise and support continuous learning. The synergy between academia and industry is crucial for the continuous evolution of software architecture. Collaborative efforts, such as joint research initiatives, industry-academic conferences, and internships, bridge the gap between theory and practice, ensuring that software architecture continues to address both current challenges and future opportunities.

Software architecture is a foundational aspect of software development, but it comes with several challenges that architects and developers must address to ensure the system's success. Some of the most common software architecture challenges include managing complexity, ensuring scalability, maintaining performance, achieving security, balancing flexibility and stability, dealing with legacy systems, and facilitating effective communication among stakeholders [9].

Each of these challenges requires careful consideration and strategic planning.

- Managing complexity - As software systems grow in size and functionality, their complexity increases. Managing this complexity involves abstractions and decomposition.
- Ensuring scalability - Scalability is crucial for accommodating growth in user base, data volume, and transaction loads. Implementing a distributed architecture to spread the load across multiple servers or services, which introduces complexities in data consistency and fault tolerance.

- Maintaining performance - Performance is a critical non-functional requirement, and maintaining it involves application latency, throughput, computing resource management and optimization.
- Achieving security - Security is paramount to protect data and ensure trust in the system. Ensuring data is encrypted both at rest and in transit to prevent unauthorized access will be a biggest challenge.
- Dealing with legacy systems - Integrating and maintaining legacy systems presents unique challenges such as interoperability and gradual migration.
- Facilitating effective communication – Communication between the decentralized distributed components will be more challenging.

Addressing these common software architecture challenges requires a combination of strategic planning, best practices, and continuous improvement. By managing complexity, ensuring scalability and performance, achieving security, balancing flexibility and stability, dealing with legacy systems, and facilitating effective communication, architects can create robust and adaptable systems that meet both current and future needs.

1.1.1 Monolithic architecture behavior

Monolithic architecture is a traditional software design model that structures an application as a single, unified unit. In a monolithic application, all the components, including user interface, business logic, and data access, are integrated and run as a single process [10]. This approach is straightforward and has been widely used in software development due to its simplicity and ease of initial deployment. A monolithic application typically starts with a small codebase and a single development team. As the application grows, new features and functionalities are added to the existing codebase, making it larger and more complex. This single-codebase model simplifies development initially, as developers can easily access and modify any part of the application. Furthermore, deploying a monolithic application is relatively simple: you build and deploy the entire application as a single unit, without worrying about inter-service communication or orchestration. One of the primary advantages of monolithic architecture is its simplicity. Developers only need to manage one codebase, and understanding the system is straightforward since all components are in one place. This simplicity extends to testing and debugging as well, as the entire application can be tested as a whole, and there are no issues with inter-process communication or network latency. Additionally, deployment is uncomplicated; with a single deployment package, there’s no need to manage multiple services or dependencies. However, as the application grows, monolithic architecture can lead to several challenges. The first major issue is the difficulty of managing a large codebase. As more features are added, the codebase becomes more complex and harder to understand, even for experienced developers. This complexity can slow down development, as changes in one part of the system may have unintended consequences in another. Another significant challenge is scalability [11]. In a monolithic

application, scaling typically involves replicating the entire application on multiple servers. This approach can be inefficient, especially if only certain parts of the application experience high load. For instance, if only the user interface layer needs more resources, replicating the entire application wastes computing resources on duplicating data access layers and the business logic as well. Despite these drawbacks, monolithic architecture remains a viable choice for small to medium-sized applications or for startups and projects in their early stages. It provides a simple and effective way to develop and deploy applications quickly. However, as an application scales and its complexity grows, many organizations find it beneficial to transition to a microservices architecture. This architectural style decomposes the application into smaller, independently deployable services, each responsible for a specific piece of functionality, thereby addressing many of the scalability and maintainability issues associated with monolithic architecture.

The market data over the past decade shows a significant shift from monolithic architecture to microservices architecture. In 2010, monolithic architecture dominated the market, holding an 80% share, while microservices had only a 20% share [12]. This trend continued with monolithic architecture maintaining a strong presence until around 2014. However, as the limitations of monolithic architecture became more apparent and the benefits of microservices architecture gained recognition, a clear shift began to emerge. By 2016, the market share of monolithic architecture had decreased to 60%, with microservices architecture increasing to 40% [13]. This trend accelerated as businesses sought more scalable, flexible, and maintainable solutions for their growing and increasingly complex applications. By 2018, the market share for both architectures was nearly equal at 50%, indicating a pivotal moment where microservices started to gain a dominant position. In the following years, the adoption of microservices architecture continued to rise rapidly. By 2020, microservices architecture had taken a 60% market share, leaving monolithic architecture with only 40%. This trend has only intensified, and projections for 2024 show that microservices architecture will hold an 80% market share, with monolithic architecture reduced to just 20% [14]. This shift is driven by several factors, including the need for better scalability, the ability to deploy and manage services independently, and the adoption of cloud-native technologies.

1.1.2 Service oriented architecture behavior

Service-Oriented Architecture (SOA) is an architectural pattern in software design that emphasizes the use of loosely coupled services to support the requirements of business processes and software users. This approach allows for the creation of modular, reusable services that can be distributed across a network and accessed by different applications or users. Each service in an SOA is a discrete piece of functionality that is self-contained and can be independently deployed and managed, which promotes flexibility and scalability in software systems. At its core, SOA focuses on designing services around business functions, which are then made available over a network

using standard communication protocols, typically over HTTP/HTTPS [15]. These services interact through well-defined interfaces and data contracts, often described using Web Services Description Language (WSDL) for SOAP services or OpenAPI for RESTful services. This ensures that different services can communicate effectively, regardless of the underlying platform or programming language used to implement them. One of the key advantages of SOA is its ability to integrate disparate systems. By encapsulating business logic within services, organizations can expose the functionalities of legacy systems and integrate them with newer applications without extensive redevelopment. This integration capability is particularly valuable in large enterprises with heterogeneous IT landscapes [16]. For example, an enterprise resource planning (ERP) system can be connected to a customer relationship management (CRM) system via SOA services, facilitating seamless data exchange and process automation across departments. SOA promotes reusability, as services designed for one application can be reused by other applications, reducing duplication of effort and improving consistency across an organization's IT systems. This reuse is facilitated by the loosely coupled nature of SOA, where services are designed to be independent of each other. This independence allows for easier maintenance and updates, as changes to one service do not necessarily impact others. Moreover, because services communicate through standard protocols, they can be composed into larger, composite applications that address complex business processes more effectively. However, SOA also comes with its challenges. ESB became a main bottleneck for the SOA architecture which is used for the service orchestration [17]. The management of multiple services can become complex, especially as the number of services grows. Ensuring consistent security, monitoring, and governance across all services requires robust infrastructure and sophisticated tools. Additionally, the performance overhead introduced by network communication between services can be a concern, particularly in latency-sensitive applications. In recent years, the principles of SOA have influenced the development of microservices architecture, which can be seen as an evolution of SOA. While both architectures emphasize service decomposition and reuse, microservices take these principles further by promoting even finer-grained services and greater independence.

Service-Oriented Architecture (SOA) continues to be a pivotal framework in modern software development, with its market showing steady growth driven by the increasing adoption of cloud computing, big data, and mobile technologies. The SOA market is expected to grow at a substantial rate. According to a report by Allied Market Research, the global SOA market was valued at $9.5 billion in 2020 and is projected to reach $25.14 billion by 2027, growing at a CAGR of 14.4% from 2020 to 2027 which is very less compared to the microservice architecture [18]. Major players in the SOA market include IBM, Oracle, Microsoft, TIBCO Software, and SAP. These companies are continually investing in R&D to innovate and enhance their SOA offerings. The competitive landscape is characterized by strategic partnerships, mergers, and acquisitions aimed at expanding product portfolios and enhancing market

presence. In summary, Service-Oriented Architecture provides a flexible and scalable approach to designing software systems by modularizing functionality into discrete, reusable services. This architecture enables better integration of disparate systems, promotes reuse, and allows for targeted scalability. Despite its complexities, SOA remains a powerful approach to building adaptable and maintainable software systems, laying the groundwork for modern architectural paradigms like microservices.

1.1.3 Decentralized and distributed architecture

Decentralized and distributed architectures, particularly in the context of microservices, represent a transformative approach to designing and managing complex software systems. These architectures have become increasingly popular due to their ability to enhance scalability, resilience, and flexibility in software development. Decentralized architecture in microservices involves breaking down a system into smaller, independent services that operate without a central authority [19]. Each microservice handles a specific piece of functionality and communicates with other services through well-defined APIs. This decentralized nature allows each service to be developed, deployed, and scaled independently, which is a significant departure from the monolithic architecture where all components are tightly coupled. One of the primary benefits of a decentralized microservices architecture is its fault tolerance. Since each service operates independently, the failure of one service does not necessarily bring down the entire system. This enhances the overall resilience of the application. Furthermore, decentralized architecture supports continuous deployment and integration practices, enabling teams to deploy updates to individual services without affecting the whole system, thereby reducing downtime and accelerating the development process. Distributed architecture complements decentralization by spreading services across multiple servers or data centers. In a distributed microservices setup, services can be hosted on different nodes that might be geographically dispersed. Additionally, it enhances the system's scalability, as new instances of services can be added to handle increased load. In a distributed microservices architecture, orchestration and service discovery are crucial. Service discovery mechanisms enable microservices to find and communicate with each other dynamically, which is essential for maintaining the flexibility and efficiency of the system.

The combination of decentralized and distributed architectures in microservices offers several advantages [20]. It enables better resource utilization, as services can be scaled independently based on specific needs. This is particularly useful in cloud environments, where resources can be allocated dynamically. Moreover, this architecture enhances the ability to handle complex and evolving business requirements, as new services can be added or modified without significant rework. A practical example of this architecture is seen in companies like Netflix, which employs a highly decentralized and distributed microservices architecture to manage its vast content delivery network [21]. Each service, from user interface to recommendation

algorithms, operates independently but coordinates with others to provide a seamless user experience. This setup allows Netflix to deploy updates rapidly, handle massive traffic loads, and ensure high availability and performance across the globe. Despite its benefits, implementing a decentralized and distributed microservices architecture presents challenges. Managing multiple services increases operational complexity, requiring sophisticated monitoring and management tools. Ensuring data consistency across distributed services can be difficult, particularly in scenarios involving transactions that span multiple services. Additionally, network latency and partitioning issues need to be addressed to maintain performance and reliability. Security is another critical concern, as each microservice exposes an API that could become a potential attack vector. Implementing robust security measures, such as API gateways, encryption, and authentication mechanisms, is essential to protect the system.

The adoption of microservices architecture has surged significantly in recent years, driven by the growing demand for scalable and resilient applications. According to market research, the global cloud microservices market size was valued at approximately USD 1.4 billion in 2023 and is projected to grow at a compound annual growth rate (CAGR) of over 20.3% from 2024 to 2032, reaching around USD 7.2 billion by 2032 [22]. This growth is fueled by several factors, including the increasing use of mobile-based applications, the rise of DevOps practices, and the ongoing digital transformation initiatives across various industries [23]. The IT and telecommunications sectors are among the leading adopters of microservices, accounting for a significant portion of the market share. These industries benefit from the agility and rapid deployment capabilities that microservices offer, enabling them to swiftly introduce new features and services to meet evolving customer demands. For instance, telecommunications providers can efficiently roll out new network functionalities, enhancing service delivery and operational efficiency. Geographically, North America holds a substantial share of the cloud microservices market, driven by advanced IT infrastructure and the presence of major cloud service providers like Amazon Web Services (AWS), Microsoft Azure, and Google Cloud. The region's dynamic industries, such as e-commerce, finance, and healthcare, demand agile and scalable applications, making microservices a natural fit. The COVID-19 pandemic further accelerated the adoption of microservices as businesses sought agile solutions to rapidly adapt to changing market conditions and remote work requirements [23]. Despite the clear benefits, the transition to microservices architecture is not without challenges. Migrating from monolithic systems can be complex and resource-intensive, often requiring significant rewrites of existing applications. Additionally, managing a distributed system introduces new operational complexities, such as ensuring security, managing inter-service communication, and maintaining data consistency. However, the long-term benefits of enhanced scalability, resilience, and flexibility often outweigh these initial hurdles. Decentralized and distributed architectures, especially in the realm of microservices, represent a significant evolution in software design. They offer unparalleled flexibility, scalability, and resilience,

making them well-suited for modern, complex applications. While the transition to such architectures involves addressing several challenges, the benefits they provide in terms of performance, fault tolerance, and development agility make them an attractive choice for forward-thinking organizations. As technology continues to advance, the adoption of decentralized and distributed microservices architectures is likely to grow, driven by the need for robust, scalable, and maintainable software solutions.

1.2 Motivation

Most of the software is transitioning from monolithic architecture to microservices architecture to enhance software performance, which is the most concerning quality attribute. Application performance directly impacts the application end users and their day-to-day operations. It is evaluated based on the system's measurable aspects while executing specific functions within defined constraints like capacity, latency, and resource usage. A straightforward definition of software performance is its responsiveness, signifying how promptly the software behaves. Real-time systems aim to provide immediate responses to the software user. Over the past decade, many performance issues surfaced in production environments, often resulting from unpredictable user and environmental behaviors. Presently, assessing performance is an integral part of the system design phase, with various criteria used to gauge software system performance.

- Latency / Response Time – This refers to the duration taken to complete a task and respond. A smaller time difference between the start and end indicates good system performance. API-based synchronized systems measure API response time in microseconds and milliseconds.
- Throughput – This represents the number of tasks completed within a specified time interval, often referred to as transactions per second (TPS). Measurement methods for throughput vary between applications. Higher throughput signifies favorable software performance.
- Capacity – This gauges the amount of work software can handle, with maximum throughput defining system capacity. It indicates the maximum number of events the software can execute within a unit of time while utilizing all available resources. For instance, if software A can support a maximum of 250 TPS with 1s latency on a backend AWS m4.large VM and considers network bandwidth, this defines its capacity. A high capacity implies excellent software performance.

Software capacity and throughput will increase after transforming the system from the monolithic to the microservice architecture compared to the monolithic architecture. But overall, software system response time will increase because of the distribution of the components.

There are many protocols to do microservice-to-microservice internal communication. Most of the microservice exposes the Application Programming Interfaces (APIs) and

communication using the REST/SOAP using the HTTP/HTTPS protocols. Some microservices use the topics and queues using the AMQP protocols. The modern application uses gRPC with the protocol buffers. People are using different mechanisms and trying to find an optimized internal communication method between microservices. As we transitioned from monolithic applications to microservices, it became necessary for the services to interact efficiently while minimizing latency.

1.3 Problem Definition

"Converting monolithic architecture to microservice architecture involves deploying components as independent services in a cloud-native environment. While this enhances the architecture, it introduces challenges on service communication such as network latency and communication overhead, which impact overall application performance in terms of response time and throughput. This research aims to investigate these performance implications and optimize inter-service communication to improve application response time and throughput."

Many projects still use the monolithic system because they are concerned about the performance after the system migrates into microservices. Once the microservice-based system is deployed, each service operates independently, and communication between microservices becomes essential to support overall system functionality and business logic. This communication is named Inter-service communication. The main overhead of these systems is the service communication done in the network layer and based on that communication mechanism. Another part of the side is that most architects do not perfectly design microservices because of the inter-process communication issues with the current practice. They pack all the services into one bundle and put it like a microservice. Most developers use synchronized communications and synchronized communication methods with protocols such as HTTP / JMS / MQTT. However, there is no properly defined inter-service communication protocol or method with the minimum impact on the application performance.

The research is aimed at satisfying the abovementioned objectives and solving the identified problem. Based on the outcome of this research, system architects can design better microservice architectures using the researched method. It will help the software industries move their monolithic application into the new microservice architecture and deploy them in environments like cloud native.

1.4 Objectives

This research aims to find the optimized and effective mechanism for inter-service communication between the microservice architecture services deployed in the cloud-native environment with new technologies and measure the impact on overall application performance. Research objectives can be listed as follows,

- To identify the strengths and limitations of existing inter-service communication methods in a microservice architecture.
- To investigate similar research that explored this problem and identify their weaknesses in the present context.
- To propose a suitable communication mechanism to improve overall system performance in terms of latency and throughput.
- To develop the prototype using the proposed solution with cloud-native technologies.
- To evaluate the proposed mechanism in the cloud-native environment.
- To present a reference architecture of the proposed mechanism for a cloud environment.

1.5 Research Questions

Microservices have emerged as a leading paradigm for building scalable and resilient applications. A critical aspect of microservice architecture is the mechanism for inter-service communication, which profoundly impacts overall system performance and reliability. As organizations increasingly adopt cloud-native environments, the need for optimized inter-service communication methods becomes more pressing. This research aims to explore and propose effective communication mechanisms within microservice architectures deployed in cloud-native settings. To achieve this, a comprehensive set of research questions has been formulated, each targeting a specific objective of the study. These questions will guide the investigation into current methodologies, recent advancements, and the development and evaluation of a novel communication mechanism designed to enhance system performance. Through this research, we aim to provide a robust reference architecture that can serve as a benchmark for future implementations in cloud-native microservice environments.

Q1: What are the key architectural differences and trade-offs between monolithic and microservice architectures, specifically in terms of inter-service communication mechanisms and their impact on performance and scalability?

The complexity of modern software applications necessitates a well-defined structure, known as software architecture. This architecture acts as a blueprint, outlining the organization of various components within the system. A critical aspect of this organization is inter-service communication, which enables individual components (services) to interact and collaborate effectively. This research question delves deeper into two key aspects, software architecture and inter-service communication. We'll

explore different types of software architectures and their defining characteristics. This might include monolithic architecture, microservices architectures, and cloud native architecture. Understanding these architectures helps us grasp how services are organized and interact within the system. The research will analyze the various methods the services use to communicate with each other. Each method offers unique advantages and disadvantages in terms of performance and scalability. By examining both software architectures and communication methods, we gain a comprehensive understanding of how data flows within a software system. This knowledge empowers developers to create robust, efficient, and maintainable applications.

Q2: How do most trending existing inter-service communication methods perform in terms of latency and throughput?

As technology evolves, new tools and approaches emerge to address the persistent challenges in inter-service communication within microservice architectures. These advancements, however, often reveal limitations in established methods, particularly regarding latency and throughput. This research question focuses on examining how existing communication methods for microservices measure up in a rapidly advancing technological landscape. While traditional methods offer known strengths, they may now face new limitations, especially in areas like scalability, efficiency bottlenecks, and integration challenges as architectures grow more complex. By evaluating these communication methods through the lens of recent breakthroughs such as protocol optimizations and cloud-native enhancements this research question seeks to uncover potential weaknesses in established approaches. Identifying these weaknesses in light of current technological advancements can guide the development of more robust and adaptable communication strategies, ultimately helping systems capitalize on new technology to improve performance, scalability, and integration in microservice environments.

Q3: What are the critical characteristics and requirements for an optimized inter-service communication mechanism within microservice architecture?

In the ever-changing world of software development, communication between services is paramount. Existing methods, while offering some benefits, often have limitations that impact performance. This research question delves into the key characteristics an optimized communication mechanism should possess. By analyzing the weaknesses of current approaches and exploring technological advancements, we can define the ideal communication method. This method should prioritize aspects like speed, scalability, reliability, and loose coupling between services. Additionally, it should seamlessly integrate with cloud-native technologies to ensure smooth operation and leverage the full potential of modern advancements. Defining these characteristics will pave the way for the development of a communication mechanism that overcomes existing limitations and fosters a high-performing software ecosystem.

Q4: How can cloud-native technologies such as container orchestration be leveraged to design a novel communication mechanism that addresses the limitations of existing approaches?

The rise of cloud-native development has revolutionized how applications are built and deployed. This new paradigm emphasizes scalability, agility, and resilience – qualities essential for efficient inter-service communication. This research question explores how emerging cloud-native technologies such as containerization and container orchestration can be harnessed to design a novel inter-service communication solution that surpasses the limitations of traditional methods. Briefly touch upon the drawbacks of existing communication methods, such as latency issues in synchronous communication and complexities. We will highlight the capabilities of cloud-native technologies that can address these challenges and explain how these technologies can be combined to create a novel communication solution. By leveraging these cloud-native technologies, this research question aims to propose a novel inter-service communication solution that offers improved performance, scalability, and resilience compared to traditional methods. This solution will be designed to seamlessly integrate with the cloud-native ecosystem, fostering a more efficient and agile application development environment.

Q5: How can a prototype of the proposed communication mechanism be implemented with the cloud-native technologies and environments?

The proposed communication mechanism, designed to revolutionize inter-service communication, needs a platform to come to life. This research question dives into the process of implementing a prototype utilizing the power of cloud-native technologies. Here briefly recap the key characteristics of the proposed communication mechanism. This might include aspects like how it fit with the existing microservice platform. Delineate the specific cloud-native technologies that will be employed in the prototype. Explain the steps involved in building the prototype, such as developing the communication strategy, configuration management for that prototype and deployment strategy. This research question aims to showcase the feasibility of the proposed communication mechanism by developing a functional prototype within the cloud-native landscape. By leveraging the tools and services offered by cloud providers, the prototype will demonstrate the real-world application of this novel approach and pave the way for its future implementation in larger-scale projects.

Q6: Compared to the existing HTTP communication method, how does the proposed mechanism impact key performance metrics (latency, scalability) in a test environment?

The pursuit of optimal inter-service communication has resulted in the creation of an innovative mechanism that leverages cloud-native technologies. This research question delves into the core of the matter: how does this proposed mechanism work against existing HTTP method in terms of key performance metrics such as latency, application throughput and resource utilization. Briefly outline the existing inter-service communication methods that the proposed mechanism will be compared to. These could be traditional approaches. Also going to explain how cloud-native features contribute to the proposed mechanism's performance edge. By conducting a rigorous evaluation, this research question aims to demonstrate the potential performance benefits of the proposed communication mechanism. By leveraging cloud-native technologies, this novel approach aims to outperform existing methods and contribute to the development of high-performing and resilient cloud-native applications.

Q7: What are the key design principles and best practices for implementing the proposed mechanism in real-world applications particularly in cloud native environments?

Key design principles and best practices are essential for implementing the proposed communication mechanism in real-world applications, guiding its effective deployment and integration within cloud-native environments. The introduced communication mechanism has undergone rigorous evaluation, showcasing its potential for efficient inter-service communication in cloud-native environments. Research question considering the crucial next step in designing a reference architecture for a cloud environment that seamlessly integrates this novel approach. Highlight the strengths of the proposed mechanism, particularly its impact on performance metrics like latency, throughput and scalability. Explain the concept of reference architecture. It serves as a blueprint, outlining the recommended components, their interactions, and best practices for deploying the proposed communication mechanism in a cloud environment. It Describes how the evaluation results will inform the design of the reference architecture by considering the performance and workload optimization. By leveraging these design principles and incorporating evaluation-driven best practices, the reference architecture will provide developers and cloud architects with a robust guide for integrating the proposed mechanism. This ensures the development of efficient and scalable cloud-native applications, offering a practical approach to optimizing inter-service communication and ultimately advancing performance and reliability in distributed systems.

1.6 Publication Contribution

This highlights the key contributions made at each stage of the research, which were published in well-reputed Scopus-indexed journals and presented at esteemed conferences. These publications support the progression of our work in microservices architecture and inter service communication, offering valuable insights and

innovations to the academic and professional community. By sharing our findings through these channels, we ensured that our research was critically evaluated, refined, and recognized within the broader field of cloud-native microservices. The following provides an overview of these publications, each reflecting a significant milestone in our research journey.

An Exploratory Evaluation of Replacing ESB with Microservices in Service-Oriented Architecture

Conference - International Research Conference on Smart Computing and Systems Engineering (IEEE conference)

Contribution - This paper explored the transition from Enterprise Service Bus (ESB) to microservices within a service-oriented architecture (SOA). It provided foundational insights into the challenges and benefits of microservices, setting the stage for my later work. The evaluation highlighted the potential performance improvements and architectural flexibility offered by microservices, which informed your decision to pursue further research in this area.

Taxonomical Classification and Systematic Review on Microservices

Journal - International Journal of Engineering Trends and Technology (Scopus indexed journal)

Contribution - The study classifies microservice architecture using the PRISMA model and compares its evolution with past and future trends. This systematic review laid the groundwork for my research by providing a comprehensive review of microservice architecture as an emerging alternative to traditional monolithic systems, which have posed challenges in cloud migration due to issues like scalability and performance. By analyzing the state of the art, this publication helped to identify gaps in existing research, particularly in inter-service communication within cloud-native environments. It also provided a framework for understanding the diverse approaches to microservices, influencing the direction of your subsequent investigations.

Evaluating the Inter-Service Communication on Microservice Architecture

Conference - 7th International Conference on Information Technology Research (IEEE conference)

Contribution - In this stage, mainly focused specifically on microservices inter-service communication, a critical aspect of microservice's performance. The paper presented an evaluation of various communication mechanisms, including synchronous and asynchronous methods, and their impact on performance. The findings emphasized the need for an optimized strategy that balances performance with reliability, directly contributing to the development of our proposed strategy.

Optimized Strategy for Inter-Service Communication in Microservices

Journal - International Journal of Advanced Computer Science and Applications (Scopus indexed journal)

Contribution - This paper introduced proposed optimized strategy for inter-service communication in microservices. It detailed the technical aspects of our approach, including the use of TCP-based communication and Redis Stream for reliable message delivery. The publication demonstrated the effectiveness of our strategy in reducing latency and improving throughput, positioning it as a viable alternative to traditional methods.

Optimized Strategy in Cloud-Native Environment for Inter-Service Communication in Microservices

Journal - International Journal of Online & Biomedical Engineering (Scopus indexed journal)

Contribution - Expanding on our research into current cloud trends, this publication contextualized our optimized strategy within a cloud-native environment. It addressed the unique challenges posed by cloud-native microservices and brought optimized interservice communication and shows how our strategy adapts to cloud native concepts. The paper provided empirical evidence of the strategy's performance in a cloud-native setting, further validating its applicability and effectiveness.

Reference Architecture for Microservices with an Optimized Inter-Service Communication Strategy

Conference - International Research Conference on Smart Computing and Systems Engineering (IEEE conference)

Contribution - In this final stage, we presented a comprehensive reference architecture that incorporates our optimized communication strategy. This architecture serves as a blueprint for implementing microservices in a cloud-native environment, offering guidelines for best practices and design patterns. The publication underscored the practical implications of our research, demonstrating how it can be applied to real-world systems to achieve enhanced performance and reliability.

1.7 Thesis Structure

This thesis explores methods to improve communication between services in software applications. Microservices architectures, where applications are divided into smaller, independent services, are the current standard. While offering flexibility and scalability, challenges arise when these services need to interact.

Chapter 2 presents the literature review, delving into existing research on relevant topics. The pros and cons of various software architectures are discussed, with a focus on microservices and their potential for building modular and adaptable applications. Current communication methods used between microservices, like REST APIs and message queues, are explored. Their strengths and weaknesses are analyzed, identifying areas where they might fall short in terms of performance or scalability. Additionally, existing solutions to challenges faced in microservice communication are explored, but limitations in these approaches, especially considering recent advancements in technology, are pinpointed.

Chapter 3 provides a deep dive into microservices architectures and inter-service communication. The key characteristics of microservices, such as loose coupling and independent deployment, and how these features contribute to the overall functionality and maintainability of a system, are explained. Then, the challenges associated with microservices, such as ensuring efficient communication and data consistency across independent services, are explored. Finally, existing communication methods used in microservices are analyzed, highlighting limitations of popular protocols.

Chapter 4 unveils a novel communication mechanism designed to address the limitations of existing methods. The architecture of this mechanism is described, outlining the components and their interactions. This architecture leverages cutting-edge cloud-native technologies to overcome the limitations identified earlier. The specific technologies chosen to implement this communication mechanism, such as cloud platforms, containerization technologies, or messaging systems, are also explored.

Chapter 5 showcases the details of the proposed communication mechanism. A high-level architecture diagram is presented, followed by an explanation of the functionalities of each component and how they work together. This detailed design likely touches upon aspects like request flow through the system, message processing, and response handling. Finally, how the proposed mechanism addresses the quality attributes identified as important for efficient communication, such as low latency, scalability, and reliability, is discussed.

Chapter 6 evaluates the proposed communication mechanism. The results of the evaluation process are presented. The mechanism is compared with existing methods using specific metrics to demonstrate the performance advantages it offers in a cloud-native environment. The trade-offs associated with using this mechanism in different application scenarios are explored, considering factors like real-time vs. batch processing workloads and how the approach adapts to each. This analysis highlights the situations where the approach is most effective and identifies areas where it might require additional optimization for specific scenarios. Finally, this chapter might

introduce a reference system developed based on the evaluation results. This reference system would serve as a blueprint for integrating the proposed communication mechanism into a cloud environment, considering best practices and lessons learned during the evaluation.

The final chapter concludes the research journey. The initial problem identified is revisited, explaining how the proposed communication mechanism addresses it. The research objectives outlined in the first chapter are revisited, demonstrating how the research has achieved them. Finally, the limitations of the research are discussed, and potential areas for future work to further improve communication between services in software applications are suggested.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

There is numerous research being conducted in the computing field related to the software architecture. The aim of this change is to identify the latest scientific findings related to microservice architecture and what are the potential problems that software industry people are facing when using software. Most of the enterprise grade software is now moving from the monolithic based architecture to the microservice based architecture to achieve the software quality attributes. Currently microservice development is the biggest trend in the software world and most of the frameworks are built to support that. In this literature review part researcher is going to deeply analyze reasons for migrating monolithic based systems to microservice architecture and its advantages. Also carefully consider the most conserving quality attributes and issues in archiving those in the microservices architecture.

2.2 Monolithic architecture

In the early stages of the software industry, enterprise-grade software predominantly relied on monolithic architecture. At that time, monolithic architecture was widely adopted across various types of software systems due to its suitability in addressing relatively simple and static user requirements. The complexity of user needs was limited, and changes in requirements occurred infrequently. As a result, monolithic systems were able to fulfill the needs of organizations effectively, offering a straightforward development and deployment process that aligned with the technological landscape of that era [24]. Monolithic software is also designed as a self-contained one large binary file. All the software components and functions are tightly coupled with each other, and all the components must be complied with before running the software. Most of the monolithic applications are single-tiered. The diagram below shows how monolithic architecture components are tightly coupled.

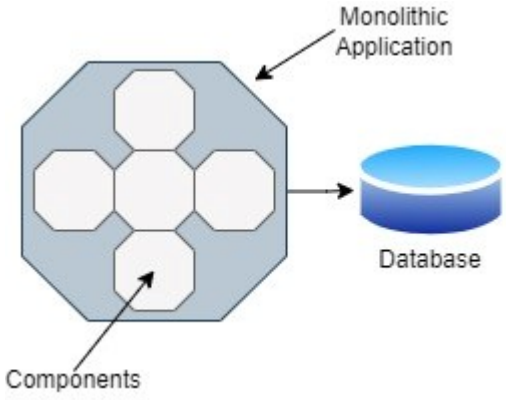

Fig. 2.1: Monolithic architecture

Characteristics of monolithic architecture:

- Single codebase: Monolithic applications have a unified codebase built as a single binary file. All modules, components, and features reside within the same code repository.
- Tight coupling: Components in the monolith systems are tightly coupled with each other. This means they are interconnected and dependent on each other to share the data. Changes to one part of the component may have ripple effects throughout the whole software.
- Unified technology: Monolithic architect software uses the same technology stack for the entire application. All modules share the same programming language, libraries, and frameworks.
- Centralized database: Monolithic architectures use a single, centralized database that serves the entire application. This centralized data store is shared among all components. That can be the relational or nonrelational database.
- Scalability: There are many challenges when scaling the monolithic-based software. The whole monolithic application must be replicated, even if only a specific component must be scaled.
- Deployment strategy: The entire application, including all its functionalities, is deployed as a single unit. Updates or changes to one application part require redeploying the entire system.
- Development simplicity: Developing and testing a monolithic application can be more straightforward. Developers work within the same environment, and the codebase is cohesive.
- Limited flexibility: Monolithic architectures may lack flexibility. Introducing new technologies or frameworks frequently involves broader system changes since all components share the same technology stack.
- Risk of a single point of failure: If one part of the monolith fails, it can potentially affect the entire application, leading to a single point of failure.

- Challenges in maintenance: As monolithic applications grow in complexity, maintaining and updating the codebase can become complex and time-consuming. Isolating and addressing issues without impacting the entire system may be challenging.

Monolithic architectures have been prevalent for many decades and are associated with legacy systems and traditional enterprise applications. They have historically provided a straightforward approach to application development. Their characteristic monolithic architecture offers certain advantages, especially in specific contexts [25].

Advantages of monolithic architecture:

- Simplicity: Monolithic architectures are more straightforward to develop, deploy, and manage. A single codebase exists, and developers work within a unified environment, leading to straightforward development processes.
- Ease of development: Building and testing a monolithic application is often more effortless since all components are part of the same codebase. This simplicity can result in faster development cycles, especially for smaller projects.
- Scoped technology Stack: Monolithic applications use a consistent technology stack throughout the whole system. This uniformity simplifies development, reduces compatibility issues, and fosters a cohesive development environment.
- Centralized database: Using a single, centralized database promotes data consistency. Data integrity is easier to maintain when all components interact with a shared data store.
- Straightforward deployment: Deploying a monolithic application involves packaging and deploying the entire system as a single unit. While this may limit flexibility, it simplifies the deployment process.
- Resource efficiency: A monolithic architecture may be resource-efficient in specific scenarios where the application size is small to moderate. It eliminates the need for additional infrastructure and communication overhead associated with more distributed architectures.
- Easy to debug: Debugging in a monolithic codebase is straightforward compared to debugging other distributed service-based systems. Developers can refer to the entire codebase at once and identify and resolve issues within less time.
- Cost-effective for small applications: A monolithic approach may be cost-effective for small-scale projects with limited complexity. The overhead associated with managing a distributed system may outweigh the benefits in such cases.

Monolithic architectures have been used for decades and are based on proven technologies and design patterns. Many legacy systems and traditional enterprise applications have been built using monolithic architecture. This stability can be an advantage in specific scenarios where adopting newer architectural styles may be deemed unnecessary. Monolithic architecture, while having certain advantages, also comes with notable disadvantages as well.

Disadvantages of monolithic architecture:

- Limited scalability: Monolithic applications are not easy to scale because of the application's states. Scaling the whole monolithic application, even if only specific functionality requires additional resources, can lead to inefficiencies and increased infrastructure costs.
- Flexibility and agility: Monolithic architectures may lack the flexibility to adopt new technologies or frameworks. Introducing changes or updates often involves modifying the entire system, limiting agility and adaptability.
- Complex maintenance: As monolithic applications grow in size and complexity, maintaining and updating the codebase can become challenging. Minor changes may have unintended consequences across the entire system.
- Single point of failure: If one part of the monolithic application fails, it can potentially affect the entire system. The centralized nature of monolithic architectures makes them susceptible to single points of failure.
- High resource utilization: For scaling, the monolithic software needs more resources and more computational power to gain more capacity.
- Limit technology adaptation: Monolithic architectures typically use a uniform technology stack throughout the entire system. This lack of technology diversity can be a limitation when specific components could benefit from different technologies.
- Development bottlenecks: In larger development teams, collaboration can become challenging due to the centralized nature of the codebase. Concurrent development by multiple teams may lead to bottlenecks and conflicts.
- Limited deployment strategies: Deploying updates or changes to a monolithic application requires deploying the entire system. This limitation can impact deployment speed and may lead to downtime during updates.
- Difficulty in scaling dev teams: As the codebase grows, managing larger development teams becomes more challenging. Coordinating efforts and maintaining consistency across a large codebase can result in productivity issues.
- Low performance: In a monolithic architecture with tightly coupled components, changes to one module may necessitate changes in multiple others. This interdependency can lead to performance impact for implementing updates or fixes.

While monolithic architectures have been used successfully for many years, these disadvantages have contributed to exploring alternative architectural patterns, such as service-oriented architecture, to address modern software development's changing needs and complexities [26]. Software enterprises often weigh the trade-offs carefully when choosing between monolithic and more distributed architectures.

Early days most of the software is build using the monolithic architecture. In those days most of the research conducted related to monolithic architecture. Researchers from Lodz University of Technology, Gos and Zabierowski, investigated the performance of monolithic architecture in comparison to microservices. [27]. They have explained monolithic architecture using the e commerce application which is having the business logic, authorization and notification module is a single platform as a single program as below.

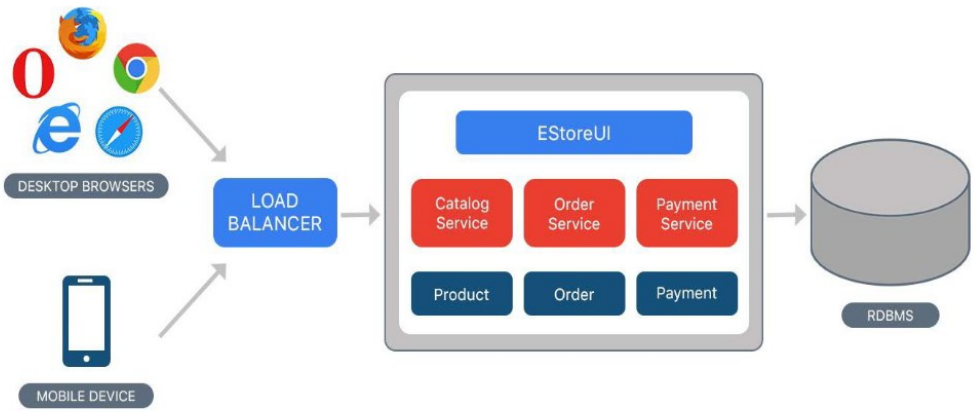

Fig. 2.2: Proposed monolithic architecture for testing

For testing proposed monolithic application has been developed using the Java language and used the PostgreSQL as a DB. Based on the results of the testing, they have confirmed that monolithic architect software is easy to developed and simple to deploy rather than the microservice architecture. But by considering the maintainability quality attribute they have state that complexity of the maintenance is higher than other software architectures. To gain reliability, availability and the scalability on the monolithic architecture is much harder. Blinowski, Ojdowska, and Przybyłek from Portland conducted a comprehensive study on the performance and scalability of monolithic architecture compared to microservices [28]. For the performance comparison they have choose the web applications which are written in Java and the .NET languages and deployed in the Azure cloud environment. Research outcome shows that monolithic architecture is suitable for small scale applications which is running in a single server instance. But if it is large and distributed system then monolithic architecture is not a good choice. Java language monolithic application perform well with compared to the .NET language application because of Java bring the computation intensive services compared to other programming languages. Researchers state that when using the cloud VM based deployment, vertical scaling is

better than the horizontal scaling in the Azure cloud because of the cost perspectives. There is no impact to the scalability factor with the implemented programming language. R Belafia and research team conducted research related to the integrated development environments and its extensibility for the monolithic and microservice development [29]. According to the researcher monolithic architect software is bound to the specific properties of the IDE. And also, those IDE are built as monolithic software architecture.

There are not much research is going on related to monolithic architecture. The relatively reduced focus on research related to monolithic architecture could be attributed to several factors. One primary reason is the industry's shift towards modern and more flexible architectural paradigms, such as microservices architecture. These newer architectures offer advantages in terms of scalability, flexibility, and responsiveness to changing business requirements [30]. Monolithic architectures, while still prevalent in many existing systems, may be considered more traditional and less aligned with contemporary trends in software development. Researchers and software industries may be more inclined to explore innovative solutions that address the evolving challenges of modern applications. Additionally, the challenges associated with monolithic architectures are well-established, and extensive research has already been conducted in this area [31]. As a result, the current focus of research efforts might be directed towards addressing emerging issues in newer architectural models. It's important to note that while the spotlight may be on newer architectures, monolithic systems still play a significant role in various domains, and research efforts continue to explore ways to enhance their performance, maintainability, and adaptability to modern computing environments.

2.3 Service Oriented Architecture

With the separation of concepts coming into the world, people are thinking about the breakdown of the monolithic system into the services. To overcome some limitations of monolithic architecture, architects introduced the Service Oriented Architecture (SOA) [32]. These services represent discrete units of functionality and are designed to be reusable, discoverable, and accessible over a network. SOA promotes the creation of modular, scalable, and flexible systems that can adapt to changing business requirements. As per the image below SOA system consists of the Enterprise Service Bus(ESB) for the orchestration of the services of the SOA system [33].

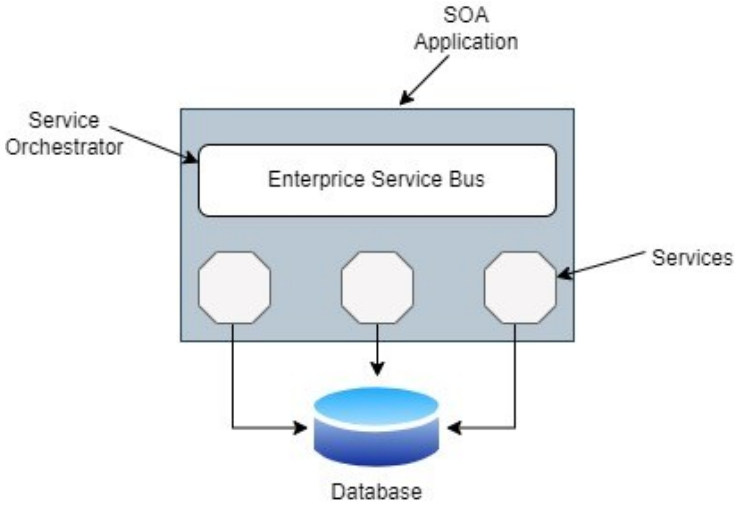

Fig. 2.3: Service oriented architecture

Characteristics of service-oriented architecture:

- Service modularity: Functionalities are segregated into the services and developed as separate services. This modularity brings reusability and maintainability than the monolithic architecture.
- Loose coupling: Allows services to evolve independently without affecting other services. Changes to one service do not require changes to the services that interact with it. But all the services are bind with the service orchestrator.
- Interoperability: This has been achieved through standardized communication protocols like HTTP or SOAP and data formats. Services can be developed using different technologies but can still interact effectively.
- Reusability: Services encapsulate specific business functionalities and can be reused across different applications. This reduces redundancy and accelerates development.
- Discoverability: SOA often involves service registries or directories that catalogue available services in ESB. This process makes it easy for developers or support team members to discover and use existing services.
- Abstraction: Consumers interact with services based on their interfaces and functionality in ESB without knowing how the services are implemented internally in the service layer.
- Enterprise Service Bus (ESB): An ESB facilitates communication, transformation, and orchestration of services within the architecture. It serves as a central hub for routing messages.
- Scalability: Services can be deployed across multiple servers, facilitating horizontal scaling to handle increased traffic.

Service-oriented architecture provides a flexible and modular approach to designing and building software systems, making it well-suited for monolithic-based applications. Service-Oriented Architecture (SOA) offers several advantages that contribute to its popularity in designing and developing complex software systems [34].

Advantages of service-oriented Architecture:

- Faster time to market: SOA enables the reuse of services for different business functions, saving development time and costs and allowing for quicker assembly of applications.
- Efficient maintenance: Easy maintenance because of the service segregation. Changes to a service do not impact the overall functionality of the business process.
- Increase adaptability: SOA systems can adapt to the technology upgrades, allowing for efficient and cost-effective modernization of applications.
- Interoperability: Services in SOA can communicate across platforms and languages, facilitating the use of services written in different programming languages within the same business processes.
- Cost efficiency: SOA's reusability and modularity contribute to cost efficiency. Organizations can leverage existing services, reducing development time and costs. Additionally, changes or updates to one service do not necessitate extensive modifications to the entire system.

While SOA offers numerous advantages, it is essential to consider factors such as proper design, governance, and adherence to best practices to fully realize these benefits. Additionally, the specific advantages experienced by an organization may vary based on its unique requirements and implementation [35]. SOA has certain disadvantages and challenges that organizations should consider when deciding whether to adopt this architectural style.

Disadvantages of service-oriented architecture:

- More complex: Implementing and managing a service-oriented system can introduce complexity, especially as the number of services grows. Coordinating and orchestrating interactions between services may become challenging.
- Performance overhead: The additional layers of abstraction, communication protocols, and middleware introduced in SOA can result in performance overhead. Service calls over a network may incur latency compared to in-process calls in a monolithic architecture.
- Service dependency: Services in SOA are loosely coupled, but dependencies between services can still exist. Changes to one service may have a cascading

effect on dependent services, requiring careful management to avoid disruptions.

- Service version control challenges: Managing different versions of services and ensuring backward compatibility can be challenging. Changes to service contracts may require careful planning to avoid disrupting existing consumers.
- Tooling and standards variability: The tools and standards used for implementing SOA can pose challenges. Ensuring interoperability and consistency across diverse toolsets may require additional effort.
- Governance complexity: Effective governance is crucial to manage services throughout their lifecycle in SOA. Establishing and enforcing policies, version control, and service discovery may introduce governance complexities.
- Vendor lock-in: Depending on specific middleware or vendor solutions for SOA may result in vendor lock-in. Switching to alternative solutions can be complex and costly.

Organizations should carefully assess their specific needs, the nature of their applications, and their development team's expertise before adopting SOA. While SOA offers many benefits, addressing these challenges requires proper planning, design, and ongoing governance [36]. To overcome these challenges, people invented the microservice architecture.

Service Oriented Architecture (SOA) became more popular for integrating the various distributed systems into one system landscape. To overcome the monolithic architecture issue most of the enterprise software are designed using the service-oriented architecture. Mike Papazoglou and Willem-Jan conducted a review on approaches and techniques related to service-oriented architecture with its concepts and principles and also review extended to the backbone of the SOA system which is Enterprise Service Bus (ESB) middleware capabilities [37]. Most challenging area of the software integration is that to integrate internal and external services to provide the business functionalities. SOA brings a solution for that using the its architecture behaviors and capabilities of ESB. Every service in the SOAs are orchestrate and routed though the ESB and ESB become a bottleneck for SOA.

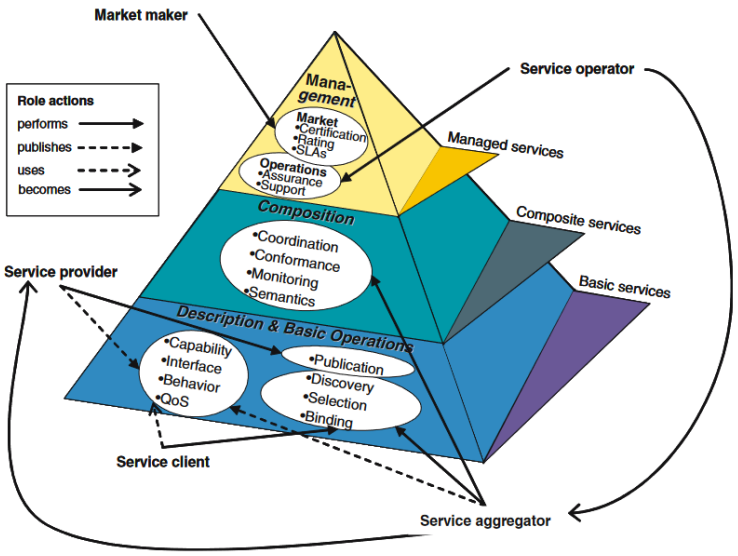

Fig. 2.4: Extended SOA

Researchers introduced extended service-oriented architecture (xSOA) which includes the ESB functionalities such as intelligent routing, service orchestration and security. Above image shows multi-dimensional layers on the xSOA which is build using the separation of concept. These layers encompass more sophisticated functionality compared to what is present in an ESB. The research study did not provide the quantitative measurement to get the idea about the newly introduced architecture. However, ESB is playing an important role in the service-oriented architecture. Group of researchers in Germany and United Kingdom performance issues and method to overcome those issues in the service oriented architecture when doing the bulk data processing[35]. Research define application performance as a three categories such as latency of the application, throughput of the application and the overall application response time. They have highlighted that performance impact occurred due to the communication overhead and the bottleneck of the ESB [38]. Some SOA system using the various of communication protocols for system integration and considerable executing time need to spend for dealing with those protocol conversations. To overcome these issues, researchers suggest designing a coarse-grained service instead of the fine-grained services [39]. But it will again be going for a monolithic behavior. Another provided suggestion is that to increases the computing resources to improve the system's performance. But that will also increase the cost of the software. A better approach is that to check the system performance while developing the batch processing systems. Hatitye Chindove and Lisa Seymour conducted a study alone with the large business enterprise in South Africa review about the benefits of the service oriented architecture and classify them into the various enterprise architectures [40]. Research has categorized the SOA benefits into 5 categories as below.

TABLE 2.1: Categories of SOA benefits

Categories	Benefits
Strategic	Enhanced agility
	Prospects for agile development
	Increased impact of change
	Development focused on domains
Operational and Maintenance	Decreased maintenance effort
	Enhanced operational support
	Shortened development time
	Increased flexibility
Organizational	Enhanced collaboration
	Increased learning opportunities
	Improved understanding of the system
Managerial	Simplified integration
	Enhanced data visibility
	Opportunity to measure cost
Governance	Enhanced application control
	Enhanced change control
	More effective compliance management

Florian Auer, Michael Felderer and Valentina Lenarduzzi developed an application using the internet of things (IOT) and service oriented architecture for the home healthcare domain [41]. SOA facilitates seamless integration for the different application, platform and the services. Research took advantage of that and integrated the several subsystems in the remote locations. In this research, heath care data has been processed passing to the different systems so that within SOA architecture provides extra security to those data. They conclude the research by asserting that service-oriented architecture exhibits a high level of interoperability. Z.D Patel from Sarvajanik College of Engineering and Technology also checked the feasibility SOA for IoT [42]. In this research below metrics are used to assess the performance of the SOA for IoT.

- Fault diagnosis parameter: Evaluate whether the service-oriented architecture incorporates a fault detection mechanism to identify potential faults. SOA, particularly adept at detecting data-centric errors, is intricately interconnected through services with higher levels of the system.

- Approach parameter: This articulates the fundamental principles employed in crafting service-oriented architectures for the IoT. Diverse SOAs may adopt varying methodologies based on specific application needs.
- Evaluation metric parameter: Defines the metrics utilized in the functioning of the architecture for a particular objective.
- Algorithm parameter: Details the algorithms employed in service-oriented architectures based on the specifications of the application.
- Application Parameter: Definition of a system or application area addressed in the SOA.
- Secure Parameter: Indicates whether the SOA incorporates security measures. Security can be established via authentication and encryption of communication.

Every SOA incorporates a fault diagnostic mechanism. Researcher state that due to its engagement with heterogeneous networks, SOA lacks security [43]. Nevertheless, security in all alternative service-oriented architectures is guaranteed through the implementation of authentication and additional security measures. The methods and assessment criteria for each SOA differ based on their particular emphasis. Governors State University conducted a survey that aim to explore SOA principles, key components, and the widespread adoption of Web Services as an implementation of SOA [44]. In old SOA based system used technologies like DCOM and CORBA for communications and modern implementations focus on Web Services, primarily using HTTP/HTTPS protocols. Web Services can be SOAP-based, involving HTTP/HTTPS POST with XML payloads, exposing service interfaces through WSDL and maintaining pre-defined contracts via XSD [45]. Alternatively, RESTful architecture, with JSON/XML payloads and a resource-based communication style, is a lightweight web service implementation. Research shows this approach allows applications in various languages to access services through HTTP calls, fostering reusability and interoperability. Arvind Kumar and team conducted a join research related to the maintainability quality metric in the SOA using the Hybrid K-means Clustering method [46]. Researchers state that maintainability factors involve several other factors such as changeability, analyzability, testability and stability. Testing phares in the software development lifecycle plays a major role to ensure the software maintainability. The research team introduced a methodology to reduce predefined test cases using a hybrid clustering approach, leveraging K-means clustering to determine the number of test cases. The introduced approach is applied not only in software testing but also in medical science for diagnosing diseases like Covid. The research emphasizes the importance of outlier detection in refining data sets for processing [47]. Identifying outliers helps eliminate errors and their impact on data integrity, particularly when dealing with non-randomly distributed outliers that can affect statistical tests, increase error variance, and bias estimates. Researcher's algorithm outperforms both leader-follower implementation and the standard k-means clustering approach [48]. In any software development process, the consideration of non-

functional requirements is crucial to ensure the system aligns with stakeholder expectations. Previous research on model-driven development for SOA identified a limited number of approaches addressing non-functional requirements. David Ameller and team's study delves deeper into the management of non-functional requirements in model-driven development processes that generate SOA, utilizing a systematic literature review based on a rigorous protocol [49]. The findings reveal that the majority of approaches predominantly concentrate on security and reliability, with non-functional requirements often expressed as annotations on functional models represented in unified modeling language. The research study suggested that existing research in this domain remains insufficient, lacking evidence of transferability to industry practices. This underscores the necessity for further research efforts to develop validated model-driven development methods capable of generating SOA that meet the non-functional requirements specified by stakeholders.

The literature on Service-Oriented Architecture provides a comprehensive overview of key themes and findings. SOA is defined as an architectural approach fostering the development of loosely coupled, interoperable, and reusable services. Its benefits include increased flexibility, agility, and reusability, along with improved interoperability. However, challenges such as complexity, governance issues, and security concerns are highlighted [50]. Many organizations have adopted SOA to modernize IT infrastructure and align it with business goals, typically through phased implementations. Technological aspects, including web services, middleware, and API management, are crucial components. Case studies explore successful implementations, emphasizing best practices in service design and versioning. The literature also delves into the evolution of SOA, considering contemporary architectural approaches like microservices and integration with emerging technologies.

2.4 Monolithic to Microservices Architecture

Microservices architecture is a way of designing software applications organizing them as a set of small, independent, and loosely connected distributed services. Each service in the microservice architect software is designed to focus on a particular business functionality and can be implemented, deployed, and scaled individually [51]. Microservices aim to improve software systems' scalability, maintainability, and portability. Each microservice communicates with necessary microservices and produces the required functional requirements. The below image shows how microservices are integrated with others.

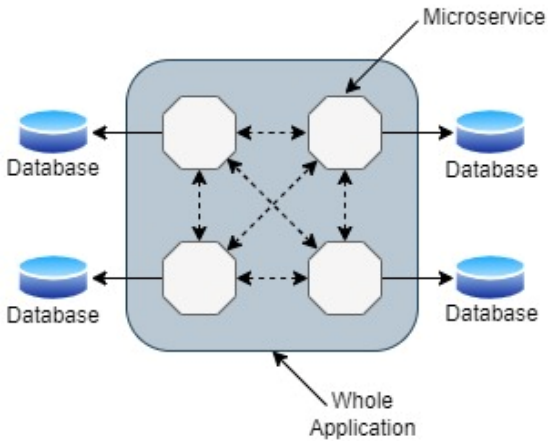

Fig. 2.5: Microservice architecture

Characteristics of microservice architecture:

- Independent services: Microservices decompose a monolithic software application into a set of small, independent, and modular services. Each service contains a specific business capability.
- Loose coupling: Microservices are designed to be loosely coupled. Changes to one service do not affect others, allowing for each service's independent development, deployment, and scaling.
- Scalability: Microservices support independent scalability. Each service can be scaled independently based on its specific demand, optimizing resource utilization.
- Single responsibility: Each microservice has a single responsibility (business capability). This focused functionality enhances clarity and makes understanding and maintaining each service easier.
- Decentralized data store: Microservices often have their own databases, promoting decentralization and avoiding a monolithic, centralized database.
- Independent deployment/development: Microservices can be deployed, developed and tested independently. This autonomy supports agile development practices, continuous integration, and continuous deployment.
- Technology variety: Microservices allow the use of diverse technologies, programming languages, and frameworks for each service. Teams can choose the most suitable technology stack for their specific microservice.
- Event-driven communication: Microservices often communicate asynchronously through events or messages. This event-driven communication supports decoupling and enhances system responsiveness.
- Containerization: Microservices are often deployed in containers (e.g., Containerd, Docker) to ensure consistency across different environments, streamline deployment, and enhance scalability.

These characteristics collectively contribute to the agility, scalability, and maintainability of microservices architecture systems. While microservices offer various benefits, their successful implementation requires careful consideration of design principles and operational practices.

Advantages of microservice architecture:

- Easy scalability: Only required microservice can be scaled at any time based on the traffic load.
- New technologies adaptation: In the microservice architecture, each microservice behaves as an independent service so that based on the need, any microservice can developed using any technology. Based on the suitability, developers have the freedom to use that and microservice communication needs to be unified.
- Resilience and fault isolation: Failures in one microservice do not necessarily affect the entire system. Microservices architecture promotes fault isolation, improving the overall resilience and robustness of the system.
- CI/CD and DevOps: Microservices align well with continuous integration(CI), continuous deployment (CD), and DevOps practices. Independent deployment of microservices supports faster and more frequent releases.
- Improved fault tolerance: Microservices architecture enhances fault tolerance by isolating failures from individual services. This prevents widespread outages and allows the system to remain functional despite issues in specific microservices.
- Improved maintainability: Microservices' modularity and independence contribute to improved maintainability. Changes to one service can be made without affecting others, making updates and bug fixes more manageable.
- Independent development and deployment: Microservices can be developed, tested, and deployed independently. This autonomy allows teams to work on different services concurrently, facilitating faster release cycles and continuous delivery.
- Suitable for cloud: Microservices are small in size and do not consume many resources, making them suitable for cloud-native development.

While microservices offer numerous advantages, it is essential to consider factors such as the complexity of managing a distributed system, proper design practices, and the need for effective communication and coordination among teams [52]. While offering several advantages, Microservices architecture comes with its own challenges and disadvantages.

Disadvantages of microservices:

- Distributed system challenges: Microservices operate as a distributed system, leading to challenges such as network latency, potential points of failure, and the need for robust communication mechanisms.
- Integration Testing Challenges: Testing the integration of multiple microservices can be challenging. Ensuring that all services work seamlessly together requires effective testing strategies and tools.
- Increased latency: Communication between microservices often occurs over a network, introducing latency. This can impact overall system performance, especially in scenarios with high inter-service communication.
- Data consistency issues: Ensuring consistency in data across microservices can be complex due to the different data stores. Managing transactions and maintaining data integrity may require careful design and implementation.

Organizations considering the adoption of microservices should carefully assess their specific requirements, team capabilities, and the nature of their applications [53]. While microservices offer benefits, addressing these challenges requires thoughtful design, proper tooling, and adherence to best practices and new strategies.

The transition from monolithic to microservices architecture is a hot topic in both academia and industry. Yalemisew Abgaz and team conducted a systematic review on the decomposition of monolithic applications into microservices architectures, providing a comprehensive analysis of the challenges and strategies involved in this transition.[54]. They focus on the methods of architectural refactoring, comparing different approaches and providing decision guides. Al-Debagy and Martinek conducted a comparative analysis of microservices and monolithic architectures, examining how microservice-based applications manage workloads with respect to scalability, adaptability, and performance [25]. Their study provides an in-depth analysis of the advantages and challenges of each architectural style, offering insights into the suitability of microservices for dynamic and resource-intensive applications [25]. The modern trend in system migration is to move from legacy and monolithic systems towards microservices architectures. This migration is often driven by the desire to improve certain quality attributes. Interestingly, some companies are choosing to revert from microservices back to monoliths, sparking debates within the industry. Reasons for this reversal and key considerations during the process have been investigated in multivocal literature reviews. These diverse sources provide a comprehensive understanding of the current state of the transition from monolithic to microservices architectures, highlighting the complexities involved in this process.

Monolithic architecture poses challenges in scalability, performance, and maintenance due to its integrated nature and inability to implement diverse technology stacks in different modules. Microservices architecture offers a solution by allowing modular development, deployment flexibility, and improved efficiency. Zhongshan Ren and

team proposes a approach for migrating monolithic legacy web applications to microservices [55].

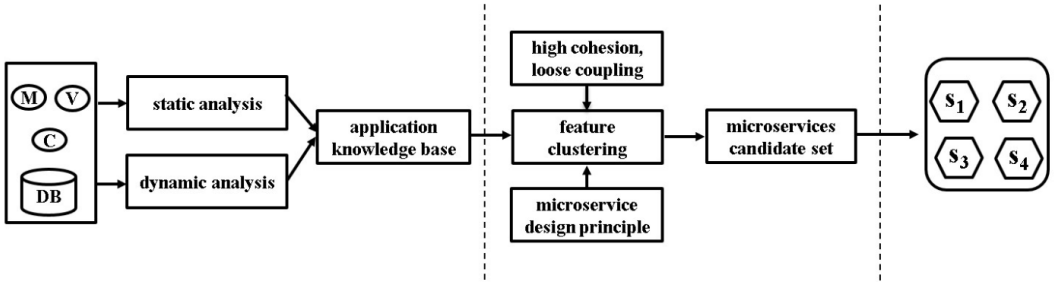

Fig. 2.6: Monolithic to microservice decomposition using static and dynamic analysis

The above approach combines static and dynamic analyses to capture both the code's characteristics and its runtime behavior. Dynamic tracing is employed to gather features, such as user input parameters and function invocations, addressing the limitations of static analysis. The authors utilize hierarchical clustering to extract tightly coupled behavior characteristics and generate a candidate set of microservices. Data migration is based on the application's data access and structure, dividing data into microservices through database depots and tables. Researchers evaluates the performance and scalability differences between monolithic and microservices applications, demonstrating the efficiency of the proposed migration method through experiments and comparisons with existing works. Dilshodbek Kuryazov, Bekmurod Khujamuratov and Dilshod Jabborov introduced a approach to decomposing the old monolithic systems to the microservice architecture [56]. In this research mainly highlighted that most of the monolithic systems are difficult to maintain and very hard to adapt to the new technologies. They have stated that decomposing these monolithic systems into smaller subsystems, components, and services, known as microservices, emerges as a solution and introduces a approach for that as below.

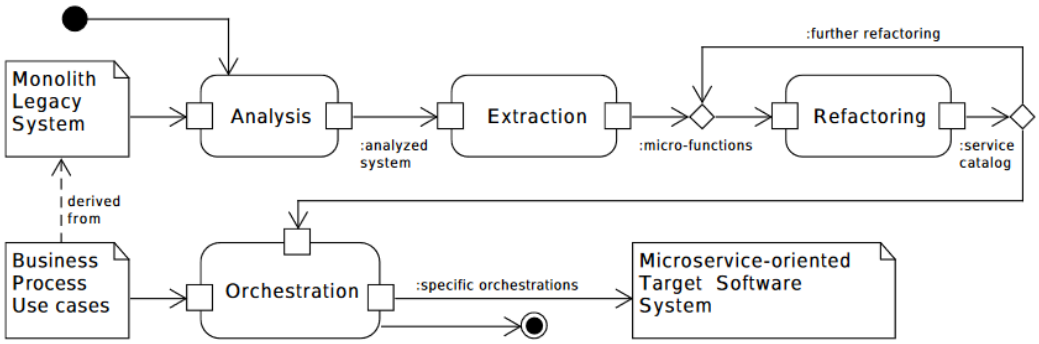

Fig. 2.7: Framework for monolithic to microservice conversion

Microservice based architectures, emphasizing reduced complexity and improved maintenance, offer a promising approach for developing large-scale software

applications. This approach facilitates easier design, implementation, and maintenance of software systems, allowing for the division of system features into more manageable units with optimal granularity. In industries, there is a noticeable migration from monolithic legacy applications to microservices-oriented architectures [57]. However, a significant challenge lies in extracting microservices from existing monolithic code bases. While informal migration patterns exist, there is a need for formal models and automated tools. Hence Genc Mazlami and Philipp Leitner address this challenge by presenting a formal microservice extraction model, enabling algorithmic recommendations for microservice candidates in refactoring and migration scenarios [58]. The implementation of the model is manifested in a prototype hosted on the web, with the performance evaluation establishing its suitability. Quantitative evaluation of recommendation quality, using microservice-specific metrics, demonstrates that the approach significantly reduces the average development team size while aligning with established microservice sizing standards. Furthermore, the approach effectively minimizes domain-specific redundancy within the recommended microservices. A comprehensive assessment of performance and quality was executed across 21 open-source programs employed in Python, Ruby, and Java [59]. The findings from the performance evaluation reveal scalability, particularly concerning the size of the revision history, demonstrating the general scalability of the approach. Regarding quality, the approach consistently reduces the team size associated with microservices to a fraction of the monolith's team size. Furthermore, most combinations of strategies successfully diminish average domain redundancy. The outcomes of the quality evaluation offer valuable insights for software architects, emphasizing the efficacy of the approach in reducing team size and mitigating domain redundancy in the extracted services. Anfel Selmadji and team conducted a joined research and presented a novel approach for identifying microservices from object-oriented (OO) source code, leveraging key types of information as source code details, including element relationships and their connections with persistent data, and architect knowledge about the system [60]. The proposed approach employs a hierarchical clustering algorithm and incorporates architect recommendations. But they haven’t compared the proposed method with the other methods.

2.4.1 Migration Issues and Problems

Tooling challenges - Undertaking the migration without leveraging suitable tools can result in inefficiencies and complexities. Adopting specialized tools can streamline the process, ensuring a more successful transition to a microservices architecture [61].

Team restructuring - Microservices adoption requires a reorganization of work teams due to the distinct development and operational paradigms. Teams may need to be reshaped to align with the decentralized and independent nature of microservices [62].

Technological incorporation difficulty - Monolithic architectures often struggle to incorporate new technologies seamlessly. The transition to microservices allows for

greater flexibility and innovation, overcoming the limitations imposed by the monolith but to move to that taking longer time for the team [63].

Comprehensive migration approach - Opting for a complete migration to microservices, as opposed to a piecemeal approach, is recommended for a more efficient and holistic transformation [64].

Microservices identification and design challenges - Identifying and designing microservices can be complex. Defining appropriate service boundaries and functionalities requires careful consideration to avoid creating overly complex or tightly coupled services. There is no such a benchmark for the number of microservices and its performance impact [65].

Data consistency across multiple databases - Moving from a single database in a monolithic architecture to multiple databases in microservices introduces challenges related to data consistency. Strategies must be implemented to guarantee information coherence across distributed databases [66].

2.4.2 Migration Challenges

Selecting appropriate frameworks/methods/models - In the process of transitioning from a monolithic architecture to microservices, the critical initial step involves the careful selection of frameworks / methods / models that align with the organization's goals and technical requirements. This decision shapes the entire migration journey, influencing factors such as scalability, maintainability, and adaptability [67].

Restructuring hierarchical organization for microservices - To fully embrace microservices, an organization needs to reconfigure its hierarchical structure. This entails shifting from traditional, centralized approaches to a more decentralized model, fostering cross-functional teams that align with the principles of microservices architecture [68].

Leveraging automation with cloud computing concepts - Efficient migration relies heavily on automation facilitated by tools, especially in the realms of cloud computing and containerization. Embracing these technologies streamlines deployment, scaling, and management of microservices, optimizing operational efficiency. Simultaneously, investing in staff training ensures the workforce is adept at utilizing these tools effectively [68].

Prioritizing business functionality with agile methodologies - When migrating to microservices, a strategic approach involves prioritizing business functionality. Using agile methodologies helps in breaking down the migration process into incremental steps, focusing on delivering value to end-users at each iteration [69]. This iterative approach ensures adaptability and responsiveness to changing requirements.

Implementing logging and design patterns - To enhance the monitoring and debugging capabilities of microservices, adapting the application to generate detailed logs becomes crucial. Furthermore, incorporating design patterns, such as Domain-Driven

Design (DDD), aids in structuring microservices around business domains, promoting modularity and maintainability [70].

Ensuring data consistency with frameworks - The choice of frameworks plays a pivotal role in guaranteeing data consistency across microservices. Utilizing frameworks specifically designed to handle distributed data ensures that information remains coherent and synchronized, even in a decentralized environment [71].

Performance impact – In the microservice architecture most challenge is that to choose the correct inter service communication method to minimize the performance impact with the distributed architecture [72].

Thorough analysis for informed decision making - An indispensable aspect of successful migration involves conducting a meticulous analysis of the application earmarked for migration. This analysis informs decision-making, enabling the identification of dependencies, potential challenges, and optimal strategies for a smooth transition to microservices [55].

2.5 Microservice Architecture

During the 90s, the predominant choice for software development among companies was the adoption of monolithic software architecture [27]. This architectural approach was widely embraced for constructing enterprise-grade software solutions. During this era, client requirements were unique, static, and well-defined within specific scopes. However, as we transitioned into the 2000s, significant technological advancements reshaped the landscape. The continuous introduction and improvement of technology led to more sophisticated and intricate client demands. Adapting to these evolving requirements prompted a shift away from traditional software architecture. People's choices in selecting architecture can be segmented into considerations of product, cost, and process. Regarding the product aspect, engineers prioritize scalability over the cloud, maintainability, performance, and security. The growing preference for digital services among users necessitates application owners to scale their applications. However, the monolithic architecture proves costly and hinders the scalable provision of specific services, putting service providers at a disadvantage, particularly in competing with dynamic industries like news [73]. Consequently, there's a heightened motivation to transition to microservices-based architecture. The digitization of services brings about changing day-to-day requirements, demanding adaptability and the introduction of new features. In monolithic architecture, tightly coupled components make it challenging to implement changes, affecting the entire application. The maintainability of the code becomes problematic due to these tightly coupled modules. In contrast, a microservices architecture allows independent operation of services, enabling easy addition of new features and code changes with high testability. Different industries have varying security needs, with some requiring a higher level of security. Monolithic architecture typically features security modules, often consolidated at a single checkpoint. However, implementing code-level security

changes, especially when dealing with dependencies, poses challenges [74]. In microservices architecture, developers can introduce multiple checkpoints, enhancing security validation. Assessing software performance involves considering aspects such as response time, throughput, and capacity. Real-time systems prioritize minimal response time, posing challenges for microservices due to their distributed deployment and added latency in service-to-service communication. However, in terms of throughput, microservices demonstrate significant capacity, with a single microservice efficiently handling extensive workloads due to its independent behavior. This capacity advantage propels the increasing adoption of microservices. Many companies have embraced frequent software releases, a process streamlined by microservices. In monolithic architecture, tightly coupled modules result in significant engineering effort for adding features or patching logic, translating into higher costs compared to project expenses [75]. In the business context, where quick and cost-effective solutions are crucial, microservices enable easy adaptation to new requirements within short cycles, leveraging Continuous Integration (CI) and Continuous Deployment (CD) pipelines. Incorporating microservices architecture equips software companies with the flexibility to adopt an agile-based software development approach, effectively managing the complexities introduced by frequent shifts in requirements.

Nowadays, there are having lot of programming languages for application development in the software field. These languages serve as tools for developers to create various types of software. Some well-established languages like Java, Python, and C++ have been around for a while, while newer ones like Rust, Kotlin, and Swift are gaining popularity. Each programming language has its strengths and is suitable for different kinds of tasks. For example, Java is widely used in big business applications, Python is known for its simplicity and versatility, and C++ is often used for system-level programming [76]. Recently, languages like Rust have gained attention for their focus on both safety and performance [77]. In the mobile app world, Swift is commonly used for iOS apps, while Kotlin is preferred for Android development [78]. JavaScript, the language of the web, has become even more popular with the rise of powerful front-end frameworks like React and Vue.js [79]. Selecting a programming language is contingent on the unique requirements of a project and the partialities of the software engineers involved. As technology advances, new languages emerge, and existing ones evolve to meet the demands of modern software development. In this ever-changing landscape, developers have a variety of options to explore and pick the language that best fits their project and coding style.

One important aspect is choosing the correct programming language and framework. There are many tools and technologies available to developers to build applications. When it comes to creating microservices, which are small, specialized components that work together to make an application, developers have a variety of options. Each programming language has its own strengths, and developers can choose based on factors like their familiarity with the language, the requirements of the project, and the features provided by the framework. For example, some popular choices include Java

with Spring Boot for its extensive ecosystem, JavaScript with Node.js and Express.js for efficiency, Python with Flask for simplicity, and Go with Go Micro for a clean and efficient solution. Additionally, there are frameworks like Micronaut for Java, NestJS for JavaScript/TypeScript, and Laravel Lumen for PHP. The key is to pick the combination that best suits the needs of the project and the preferences of the development team.

Here are the release dates for stable, production-ready versions of the most popular microservice frameworks in different programming languages [67]:

- Spring Boot: Spring Boot 1.0, the first stable version, was released on April 1, 2014.
- Go Micro: Initially released in 2015.
- Moleculer: Moleculer 0.14.0, one of the early stable versions, was released on March 1, 2018.
- Vert.X: Vert.x 3.0, a major stable release, was made available in January 2016.

Various frameworks provide distinct features and advantages when it comes to developing microservices. The selection of a framework depends on factors like the development team's expertise, project complexity, and performance considerations. Developers often choose frameworks that align with their development philosophy and the specific needs of the microservices they are building.

Spring Boot - Spring Boot, built on Java, is an open-source framework designed to streamline the development process of stand-alone Spring-based applications, ensuring they are production-ready [80]. It provides a streamlined platform for creating Java applications with minimal setup, allowing developers to focus more on writing code for their business logic. Spring Boot follows the convention-over-configuration paradigm, which means it comes with sensible defaults and configurations, reducing the need for explicit setup. One of its key features is the embedded application server (like Tomcat or Jetty), eliminating the requirement for external server configuration. Spring Boot also offers a wide range of starter templates for various use cases, helping developers kick-start their projects with pre-configured setups. The framework promotes the use of microservices architecture, allowing developers to create independent, modular services that can be easily deployed and scaled. It integrates seamlessly with the larger Spring ecosystem, providing access to features such as data access, security, and messaging. In addition to its developer-friendly features, Spring Boot incorporates production-ready capabilities, including health checks, metrics, and monitoring. It supports various data sources, messaging systems, and cloud platforms, making it versatile for different application scenarios. Overall, Spring Boot accelerates the development process, enhances maintainability, and facilitates the creation of robust, scalable Java applications.

Go Micro - Go Micro is a microservices framework tailored for the Go programming language, offering a streamlined approach to developing distributed systems. Embracing the microservices architectural style, Go Micro simplifies the creation of

independent and scalable services [81]. Developers benefit from a high-level service abstraction that handles intricate aspects such as service discovery, load balancing, and communication complexities. Noteworthy features include modular and pluggable architecture, allowing customization through components like transport, registry, broker, and codec. The framework seamlessly incorporates service discovery for automatic connection between services in a dynamic environment. Load balancing mechanisms ensure optimal resource utilization and high availability. With built-in messaging capabilities, it promotes loose coupling and independence. Its extensibility empowers developers to integrate additional components or customize existing ones, making it a comprehensive and adaptable framework for Go developers venturing into a microservices architecture.

Moleculer - Moleculer is a microservices framework designed for building scalable and efficient distributed systems. It provides a robust set of features and abstractions that enable developers to create independent, modular, and loosely coupled microservices [82]. Moleculer is implemented in JavaScript and follows a decentralized and service-oriented architecture. One of its key features is the support for a range of communication patterns, including request-response, event-driven, and pub/sub, allowing developers to choose the most suitable approach for their specific use case. The framework comes with built-in service discovery and load balancing, ensuring seamless communication and resource optimization in dynamic and evolving environments. Moleculer supports various transporters, including TCP, Redis, NATS, and MQTT, providing flexibility in communication protocols. It includes a comprehensive set of tools for monitoring, logging, and debugging, simplifying the development and maintenance of microservices. Additionally, Moleculer is extensible, allowing developers to integrate custom modules and adapt the framework to their specific needs. Overall, Moleculer serves as a robust and versatile framework for JavaScript developers engaging in microservices architecture.

Vert.x - Vert.x is a reactive, event-driven microservices framework designed for constructing scalable and robust applications. Operating on the Java Virtual Machine (JVM), Vert.x is a polyglot, accommodating various programming languages such as Java, Kotlin, Scala, Groovy, and JavaScript. This flexibility allows developers to choose the language that aligns with their expertise and project needs. Fundamentally, Vert.x employs an event-driven model, enabling asynchronous communication among microservices and efficient handling of numerous concurrent connections [83]. The framework offers lightweight and modular architecture, empowering developers to assemble applications from compact, reusable components. Vert.x supports reactive programming paradigms, enabling the creation of highly responsive and resilient microservices that can react to changes in real time. Vert.x offers a variety of features, including built-in support for reactive patterns and a comprehensive set of libraries and modules for common tasks. With its focus on simplicity, scalability, and flexibility, Vert.x empowers developers to build modern microservices architectures that can efficiently handle the challenges of distributed and reactive systems.

TABLE 2.2: Microservice frameworks

Framework	Documentation Support	Community Support	Project Usage	Architecture and Features	Suitability
Spring Boot	Extensive and well-organized documentation covering various aspects	Large and active community with forums, discussions, and Stack Overflow	Widely adopted in enterprise-level Java microservices	Comprehensive features, seamlessly integrates	Best suited for Java-based enterprise applications
Go Micro	Clear and concise documentation with essential guides	Active community through GitHub issues, discussions, and forums	Used for microservices development in GO	Simple, ease of use, and suitability for smaller projects.	Suitable for small to medium-sized projects requiring simplicity
Moleculer	Comprehensive documentation with guides	Engaged community with discussions on GitHub and a community forum	Used for microservices development in JS	Flexible for various distributed systems.	Suitable for small to medium-sized applications
Vert.x	Documentation covering only core concepts	Active and vibrant community with forums, discussions	Employed for reactive and scalable applications, supporting multiple languages	Supports reactive and event-driven microservices	Good for applications requiring high concurrency

Using those programming languages and frameworks, architects deployed the software applications in different architectural patterns. Microservice architectural patterns are design principles and guidelines that help structure and organize the development of microservices-based systems [84]. One key advantage is scalability, as microservices allow for independent scaling based on specific service needs, optimizing resource allocation and reducing infrastructure costs. Additionally, the flexibility and agility inherent in microservices empower development teams to work on and deploy services independently, fostering faster development cycles and adaptability to changing requirements. The fault isolation characteristic ensures that a failure in one service does not compromise the entire system, contributing to enhanced

system resilience. Microservices also support technology diversity, allowing the use of different technologies and frameworks tailored to specific service requirements. Continuous deployment and DevOps practices align seamlessly with microservices, enabling continuous integration, deployment, and automated testing for more efficient software delivery. The modularity of microservices simplifies maintenance efforts, facilitating updates and tasks on individual services without impacting the entire system [85]. This architectural pattern also promotes autonomy among development teams, reducing dependencies and empowering teams to make decisions based on their domain knowledge. Furthermore, microservices enhance fault tolerance through redundancy, ensuring that the failure of one service does not cripple the entire system. The ability to adopt new technologies easily and implement improved scaling strategies, such as fine-grained scaling, further contributes to the appeal of microservices in modern software development. While these advantages are substantial, organizations must consider the associated challenges, including the complexity of managing distributed systems and inter-service communication, before embracing the Microservice architectural pattern.

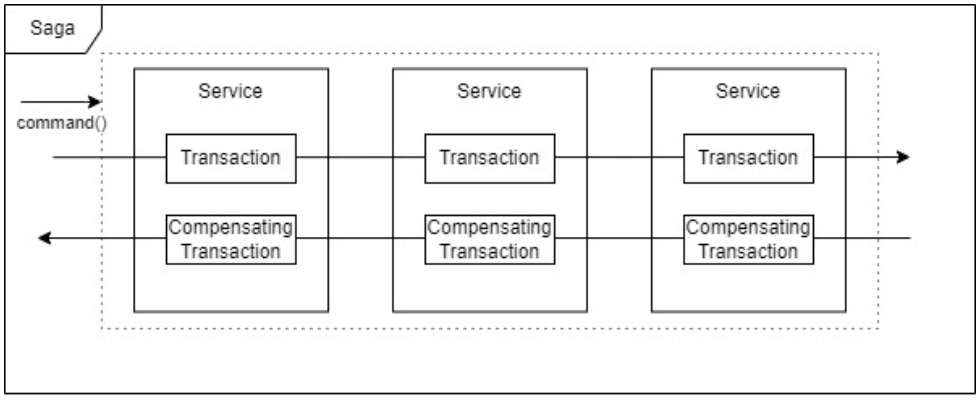

Fig. 2.8: Saga pattern

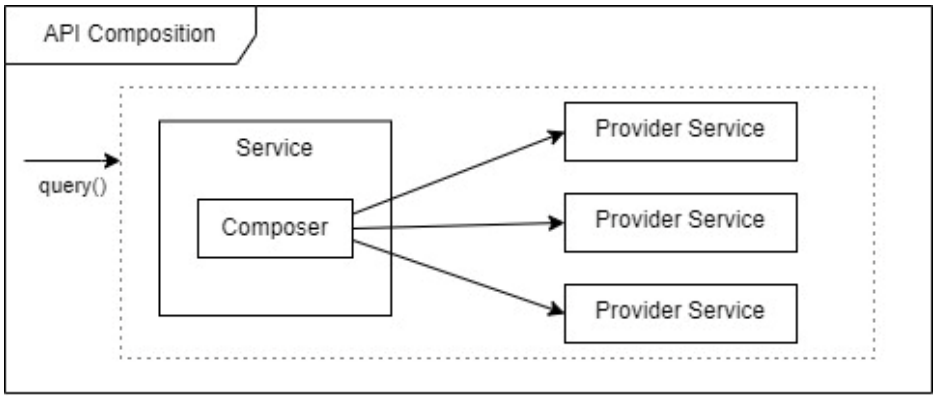

Fig. 2.9: API composition pattern

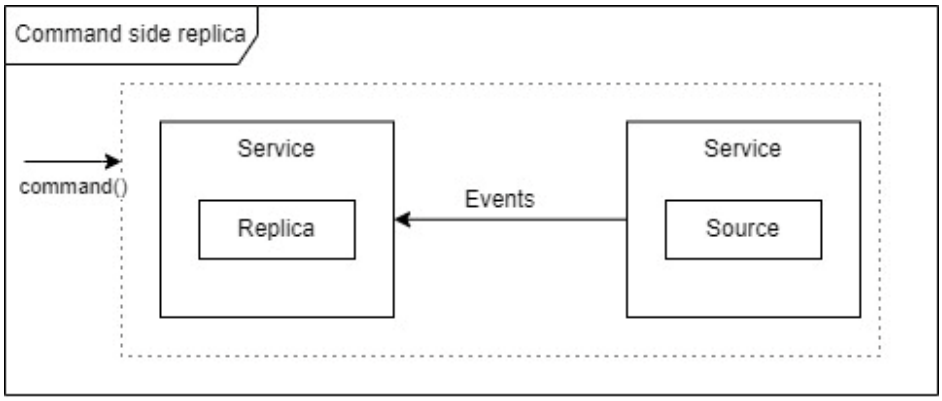

Fig. 2.10: Command side replica pattern

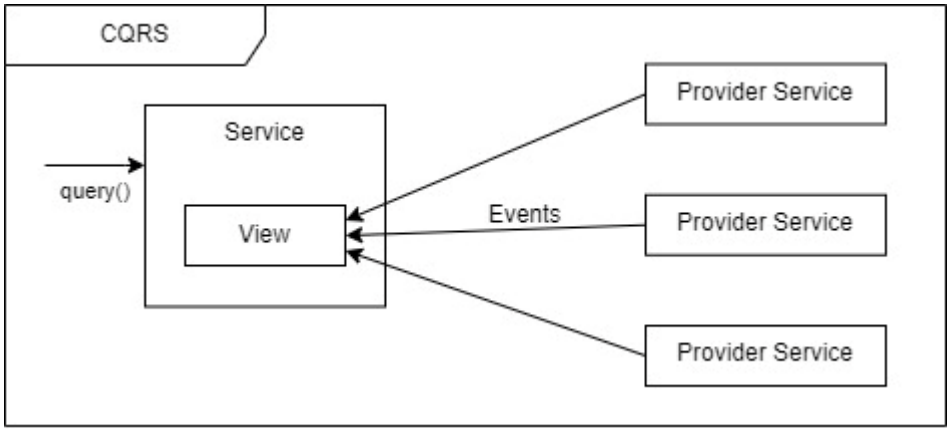

Fig. 2.11: CQRS pattern

In the realm of microservices architecture, there are two most popular patterns such as saga pattern and the command-side replica pattern. The Saga pattern involves breaking down a long-running transaction into a sequence of smaller, more manageable transactions that are easier to handle independently [86]. This approach helps maintain consistency across distributed systems by coordinating and orchestrating these smaller transactions effectively. On the other hand, the Command-side replica pattern focuses on creating replicas of the command side of a system, enabling different services to access and process the same commands independently [87]. This pattern enhances fault tolerance and allows for scalability by distributing command processing across multiple replicas. For implementing queries, there are two notable patterns: API Composition and Command Query Responsibility Segregation (CQRS) [88]. API Composition involves aggregating data from multiple microservices to fulfil a query, providing a unified view to the client. This approach simplifies the client's interaction with the system by consolidating information from various sources. CQRS, on the other hand, decouples the read and write operations, maintaining separate models for queries and commands. This separation allows for independent scaling of read and write components, optimizing performance for each type of operation. These patterns offer diverse approaches to handling commands and queries in microservices architecture, allowing developers to choose the most suitable strategy based on their system requirements and design goals.

Microservices architecture marks a new era in software development, revolutionizing the way complex applications are built, scaled, and maintained [89]. This approach not only enhances flexibility and scalability but also supports faster development cycles and resilience against failures. With advancements in technologies like could computing and AI, microservices continue to evolve, shaping the future of software architecture [90]. This technology adaptability has made microservice architecture a preferred choice for industry leaders, setting a new standard for modern application development.

2.5.1 Challenges & Difficulties in Microservice Architecture

Many individuals adopt microservices to enhance their business and software capabilities, yet challenges persist in implementing this architecture. These challenges span various phases, including design, implementation, and support. At the design level, determining the scope of a microservice poses a significant challenge [91]. Defining the boundaries of service within the architecture is debated among researchers, with varying opinions on focusing on specific business flows or application domains. The absence of a clear principle for defining microservice scope complicates size determination and understanding connection points within architecture. Moving to the implementation level, ensuring data security across microservices in a distributed environment becomes a major concern. Application performance, particularly latency, is another worry, with various communication protocols introducing complexities and challenges in defining services accurately. Deployment in a microservices architecture involves writing scripts for each service, leading to increased deployment times compared to traditional systems. Although implementing CI/CD pipelines can mitigate deployment time, it adds extra effort and costs [92]. Supporting microservices presents unique challenges due to service distribution. Without proper request tracing mechanisms, support engineers face difficulties identifying the root causes of issues. Ensuring fault tolerance and strong monitoring becomes crucial in the distributed environment, demanding additional effort and resources. In summary, challenges in microservices encompass scope definition, data security, application performance, deployment times, and support complexities, requiring careful consideration and mitigation strategies in each phase of implementation.

2.6 Inter Service Communication

In the microservices landscape, inter-service communication plays a crucial role in ensuring that different microservices can work together smoothly within a larger distributed system [93]. Think of it as the communication highway that allows these microservices to talk to each other and share information. Microservices, by nature, are designed to encapsulate specific functionalities. Service-to-service communication allows them to collaborate and collectively provide comprehensive services. Each

microservice can focus on its specialized task, and inter-service communication facilitates the seamless integration of these tasks to deliver a complete application. In microservice architecture, services are often distributed across different servers or even in different geographical locations. Microservice-to-microservice communication allows these decentralized services to communicate effectively, maintaining a cohesive system despite geographical distribution [94]. Microservices can be developed using different technologies and programming languages. Inter-service communication ensures that these diverse services can communicate seamlessly, allowing for technological adaptability and diversity within the overall system. In essence, inter-service communication is the lifeline of microservices, fostering collaboration, adaptability, and resilience. Its careful implementation is essential for realizing the full potential of microservices in building scalable, flexible, and robust applications.

In the microservices landscape, how these distinct components communicate plays a pivotal role in the overall system dynamics. There are two primary modes of communication such as synchronous and asynchronous [95]. Synchronous communication is analogous to having a real-time conversation. When one microservice requires information from another, it sends a request and awaits an immediate response. This approach is swift and effective, facilitating rapid exchanges of information [96]. However, challenges may arise, such as potential bottlenecks if one service takes longer to respond. On the other hand, asynchronous communication does not get an immediate reply from the receiver. A microservice dispatches a message to another microservice but does not wait for an immediate reply. Instead, it proceeds with its tasks. The receiving microservice processes the message at its own pace and responds when prepared. This decoupling empowers each microservice to function independently, contributing to the system's overall robustness and adaptability. The choice between synchronous and asynchronous communication hinges on the system's specific requirements. Synchronous communication is optimal for scenarios demanding real-time responses but necessitates careful management to circumvent potential delays. Asynchronous communication suits situations where immediate responses are unnecessary, fostering a more flexible and resilient system architecture [97]. It involves selecting the appropriate communication method tailored to each scenario, ensuring optimal stability between responsiveness and autonomy.

Communication protocols serve as central frameworks, comprised of complicated rules and principles, orchestrating the complicated flow of data transmission and reception between devices within a network. These protocols form the foundation, carefully defining the parameters that govern effective and reliable communication. They act as the common language, enabling disparate systems to not only exchange data but also to comprehend and interpret the intricacies of the information being shared. At the core of this complex web of protocols lies the Transmission Control Protocol (TCP), a linchpin offering a connection-oriented approach that ensures the steadfast and ordered delivery of data. Collaborating seamlessly with TCP is the

Internet Protocol (IP), an architectural cornerstone responsible for the intricate task of addressing and routing data packets across expansive networks [98]. In the Worldwide web's vast realm, the Hypertext Transfer Protocol (HTTP) takes center stage, playing a pivotal role in the seamless transmission of hypertext navigating the digital landscape. These protocols collectively weave a sophisticated tapestry, shaping the landscape of modern communication and providing the essential infrastructure for the interconnected world we inhabit. In the realm of microservices, communication protocols serve as the foundation for facilitating interactions between various services within a system. These protocols define the rules and conventions that enable seamless data exchange and cooperation among microservices. HTTP/HTTPS, being the backbone of the World Wide Web, is widely employed for synchronous communication in microservices. It operates on a request-response model, where one microservice sends an HTTP request, and the recipient microservice responds accordingly. While HTTP is suitable for scenarios requiring immediate responses, it might introduce latency in certain distributed environments [99]. Message queues and topics play a pivotal role in asynchronous communication. Microservices can communicate by placing messages in a queue, allowing other services to retrieve and process them at their own pace. This decoupled communication method enhances system resilience, enabling services to operate independently without being directly dependent on each other's availability. Selecting the appropriate communication protocol depends on the specific requirements of the microservices architecture. HTTP/HTTPS is preferable for scenarios demanding real-time interactions, while Message Queues excel in situations where asynchronous, resilient communication is more suitable. The combination of these communication protocols contributes to a flexible and adaptable microservices ecosystem. There are having most famous communication protocols in the industry as HTTP/HTTPS, gRPC and WebSocket [100].

2.6.1 HTTP/HTTPS

HTTP (Hypertext Transfer Protocol) and HTTPS (Hypertext Transfer Protocol Secure) represent foundational protocols governing the bidirectional data exchange between servers. Notably, HTTP operates in a stateless manner, where each transaction is autonomously conducted without retention of prior interactions [101]. Conversely, HTTPS introduces an augmented layer of security through the integration of SSL/TLS protocols. During the HTTPS process, a secure handshake transpires between the client and server, culminating in the establishment of an encrypted connection. This cryptographic framework ensures the confidentiality and integrity of the transmitted data. Noteworthy is the comprehensive encryption applied to the entire communication process, thereby rendering the data impervious to unauthorized interception. The validation of server identity is facilitated through digital certificates, thereby instilling a sense of trust in the communicative link. Consequently, while both HTTP and HTTPS facilitate data exchange, the latter, fortified by encryption and authentication

mechanisms, furnishes a heightened level of security conducive to the safeguarding of sensitive information during internet transmissions. There are various programming languages and frameworks that support this communication method. Almost all the microservices use HTTP-based communication for inter-service communication.

2.6.2 gRPC

gRPC, an open-source Remote Procedure Call (RPC) framework developed by Google, serves as a mechanism for facilitating communication among distributed systems through the invocation of remote methods. The procedural sequence commences with the articulation of the service and its associated methods within a .proto file employing Protocol Buffers for structured data definition [100]. Subsequent employment of the Protocol Buffers compiler yields client and server code across a chosen programming language, thereby generating programmatically derived representations of service methods and message types. Servers then proceed to enact the requisite functionality, formally registering to monitor a predetermined network address and port. Client-side engagement involves the instantiation of a stub, functioning as a lightweight proxy for mediating communication with the server, effectuating the conversion of method calls into network requests. Serialization and deserialization of structured data via Protocol Buffers serve to optimize the efficiency of data exchange. Notably, gRPC leverages the HTTP/2 transport protocol to effectuate communication, thereby benefiting from features such as multiplexing and header compression. Beyond mere support for unary Remote Procedure Calls (RPCs), gRPC accommodates streaming RPCs, enabling diverse and sophisticated communication patterns. The framework is further fortified by its commitment to security, integrating intrinsic support for Transport Layer Security (TLS) to fortify data confidentiality and integrity during transit. In summation, gRPC, with its language-agnostic toolkit, expedites the development of scalable and efficient distributed systems, facilitating the definition of services and streamlining communication between clients and servers within an academically rigorous framework. However, in this protocol, there is not much framework support for internal communication in microservices.

2.6.3 WebSocket

WebSocket protocol, integral to modern web communication, undergoes initiation through a client-driven handshake procedure. The client, signalling the intent to establish a WebSocket connection, dispatches an HTTP request featuring an "Upgrade" header [102]. A successful transition from HTTP to WebSocket is confirmed by the server's response with an HTTP 101 status code, marking the protocol switch. This transition establishes a persistent, full-duplex communication channel wherein both client and server engage in autonomous, bidirectional message

exchange. Message content is encapsulated into frames, distinguished as either text or binary in nature. The persistent nature of WebSocket connections mitigates the need for repetitive connection establishment, resulting in decreased latency and minimized overhead. Suited for applications necessitating immediate data transmission, such as real-time chat systems or online gaming platforms, WebSocket's continual connection ensures prompt exchange without recurrent handshakes. Additionally, the protocol accommodates secure connections through TLS/SSL, incorporating an encrypted layer for safeguarding sensitive information. Noteworthy is the protocol's provision for closure initiation, enabling either party to gracefully terminate the WebSocket connection through the transmission of a designated close frame. In summary, WebSocket, with its characteristic features, emerges as a pivotal tool for efficient bidirectional communication in contemporary web applications.

2.6.4 Deep Dive into Factors Influencing Performance Metrics in Inter-Service Communication

Inter-service communication, a cornerstone of modern distributed systems, significantly impacts overall application performance and reliability. This deep dive delves into the intricate factors that influence key performance metrics in inter-service communication, by enabling insights to optimize system performance and resilience in the next chapter. By understanding these factors, we can make decisions to enhance microservices architecture by improving inter service communication.

Communication style is a critical factor in the design and implementation of inter-service communication in microservices architecture. It influences performance, scalability, reliability, and overall system efficiency. The two primary communication styles are synchronous and asynchronous communication, each having distinct impacts based on various theories and principles of distributed systems and software architecture [103]. Synchronous communication involves direct communication between services where the caller waits for the response from the receiver which is the most common communication method in the microservice architecture. According to the theory of distributed systems, synchronous communication can introduce latency as services must wait for responses. This can lead to blocking behavior, where the calling service is held up until the response is received. The coupling theory in software architecture posits that synchronous communication can lead to tight coupling between services [104]. Synchronous communication is generally easier to implement and reason about due to its request-response nature, aligning with the principles of simple interaction models in software design. According to the scalability theory, synchronous communication can hinder scalability because each service instance must handle a complete cycle of requests and responses, which can limit the number of concurrent interactions. Asynchronous communication involves services communicating through messaging systems where the caller does not wait for an immediate response. The non-blocking I/O Theory suggests that asynchronous communication allows services to send messages and continue processing other tasks

without waiting for a response [105]. As per the loose coupling principle, asynchronous communication decouples services, enabling them to operate independently. This enhances system flexibility, making it easier to modify and scale individual services without affecting the overall system. The eventual consistency theory in distributed systems indicates that asynchronous communication can introduce complexity in ensuring data consistency and handling failures. This requires additional mechanisms for message delivery guarantees, retries, and idempotency. Asynchronous communication aligns with the scalable systems design theory, as it allows for better load balancing and handling of large volumes of messages. Services can scale independently based on the message load, enhancing overall system scalability.

Network layer resource consumption is another critical factor influencing inter-service communication performance within microservices architectures. Network layer resource consumption plays a pivotal role in the performance and efficiency of inter-service communication within microservice architectures [72]. The theories and principles of distributed systems and network communication provide a framework for understanding these impacts. Efficient use of bandwidth is critical for high-performance inter-service communication. According to the bandwidth delay product theory, the amount of data that can be in transit in the network directly influences throughput. Efficient inter-service communication requires optimizing bandwidth usage to prevent bottlenecks and ensure high data transfer rates [106]. Bandwidth can be reduced by reducing the network packet size when sending the packets over the network. The protocol overhead theory also proven that communication protocols with high size can reduce effective bandwidth and increase latency. The data compression theory indicates that compressing data before transmission can reduce bandwidth usage and improve throughput. Protocols that support compression can enhance network efficiency. When designing the strategy, we need to optimize the data that we are sending over the network in bring the more efficiency to the inter service communication. The CAP theorem highlights that network issues like packet loss can affect the consistency and availability of services [107]. Reliable inter-service communication protocols must handle packet loss efficiently to maintain data integrity and service reliability. When doing the inter service communication we need to ensure the message delivery over the network. The congestion control theory explains that congestion can significantly reduce throughput and increase latency, affecting the efficiency of inter-service communication.

In microservice architecture, the choice of message transfer protocol plays a pivotal role in shaping the dynamics of inter-service communication. The message transfer protocol dictates the communication patterns between microservices, defining whether the interaction follows a request-response, publish-subscribe, or event-driven model [108]. Moreover, it influences the format and serialization process of messages, impacting the efficiency of data transfer and overall performance. In architecture leveraging message transfer protocols with message queues or brokers, the

asynchronous communication between microservices is managed, influencing reliable message delivery, load balancing, and decoupling. The reliability and delivery guarantees, such as at least once or exactly once semantics, are determined by the chosen message transfer protocol. Additionally, the protocol overhead introduced by the message transfer protocol can affect the efficiency of communication, with some protocols adding more overhead in terms of message size or processing requirements [109]. The compatibility and interoperability of microservices may be influenced by the choice of a widely adopted and standardized protocol, enhancing seamless communication in heterogeneous environments. Scalability is also impacted by the message transfer protocol's ability to handle increasing message loads and efficiently distribute messages across microservices. Consequently, the selection of a message transfer protocol involves careful consideration of trade-offs between communication patterns, reliability, efficiency, and scalability, aligning with the specific requirements and goals of the microservices system [110].

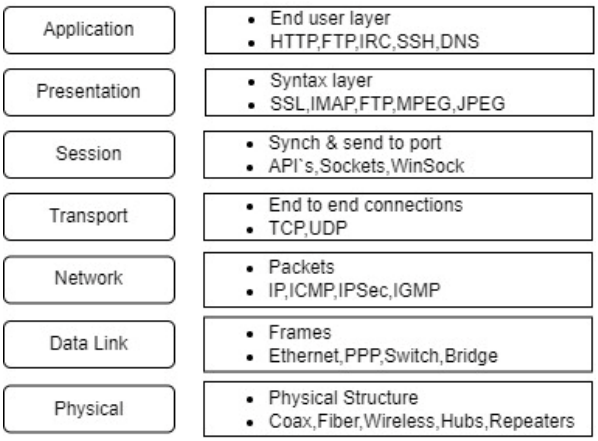

Fig. 2.12: OSI Layer

The synergy between the message transfer protocol and the OSI (Open Systems Interconnection) layer model significantly enhances the effectiveness of inter-service communication within network architectures [111]. HTTP, MQTT, and AMQP bring structured communication patterns, efficient message formatting, and reliable delivery mechanisms to the forefront. These attributes contribute to the predictability, speed, and robustness of inter-service communication. Concurrently, the OSI layer model's layered architecture provides a modular framework, offering clarity and separation of concerns. This modularity, coupled with standardized interfaces between layers, fosters interoperability and scalability. The apparent abstraction provided by the OSI model aids in understanding and optimizing communication protocols, while the flexibility and adaptability allow for the introduction or modification of protocols within specific layers to meet evolving communication requirements. In the landscape of inter-service communication, comparing the speed of the TCP layer and the application layer is a nuanced endeavor.

Programming complexity is another critical factor that influences the design, implementation, maintenance, and performance of inter-service communication in microservice architecture. The complexity can arise from various sources, including the choice of communication protocols, data serialization methods, error handling, and overall system integration [112]. Understanding these impacts through relevant theories and principles of software engineering and distributed systems can help in designing effective communication strategies. Cognitive load theory posits that the human brain has limited capacity for processing information at any given time. High programming complexity increases the cognitive load on developers, making it harder to understand and manage the system [113]. High complexity can lead to increased development time and higher likelihood of bugs and errors. Simplified inter-service communication protocols and clear abstractions can reduce cognitive load, improving developer productivity and reducing the chance of errors. Additionally, complex code without proper documentation poses difficulties in maintenance, hindering collaboration and troubleshooting efforts. Striking a balance between functionality and simplicity is pivotal for establishing a robust, scalable, and maintainable microservices architecture that optimally supports efficient inter-service communication.

The efficacy of inter-service communication within a microservices architecture is a complex interplay of various factors. Communication style, protocol selection, network layer resource usage, and programming complexity collectively shape the performance and efficiency of these interactions. A judicious balance of these elements is crucial for optimizing system responsiveness, reliability, and scalability.

2.7 Software Quality Attributes

Software quality attributes, also known as software quality characteristics and on the other hand they are very much equal to the application's non-functional requirements that represent the various dimensions along which the quality of a software system can be assessed. These attributes play a crucial role in determining how well a software application meets its intended goals and user expectations. Software quality attributes are essential because they define the characteristics and properties that contribute to the overall quality of the software. Software quality attributes directly impact the user experience. Positive user experiences contribute to the success and adoption of software applications. Investing in software quality attributes can lead to cost savings in the long run. A system that is easy to maintain, scalable, and secure can reduce the total cost of ownership and minimize the impact of unexpected issues. On the other hand, identifying and addressing software quality attributes help mitigate risks associated with system failures, poor performance, or security breaches. Proactively managing these attributes reduces the likelihood of costly issues arising during or after deployment. Software with superior quality attributes can provide a competitive advantage. Users are more likely to choose and continue using software that delivers positive and reliable experience. In summary, software quality attributes are essential for delivering software that meets user expectations, aligns with project requirements,

and provides a foundation for ongoing success, growth, and competitive positioning in the market. Most of the architects in the software industry take decisions on the software based on the software quality attributes. Many researches have proven that they have migrated the monolithic-based software to the microservices-based software architecture to archive the most concerning software quality attributes [114].

Most concerning quality attributes in software architecture;

Reliability: The software's capability to consistently and accurately execute its intended functions over time without experiencing failures or errors.

Performance: Involves elements like response time, throughput, and the utilization of resources. A performance system efficiently processes tasks within acceptable time frames.

Scalability: Refers to the software's capability to manage a higher workload or demand by incorporating additional resources while maintaining performance integrity.

Maintainability: Relates to the software's capability for making updates, modifications, or enhancements with ease. A maintainable system allows for efficient and cost-effective changes.

Usability: Centers around the user experience and the software's accessibility for seamless user interaction. Intuitive interfaces, straightforward navigation, and user satisfaction contribute to high usability.

Security: Ensures that the software protects data, resources, and functionality from unauthorized access, modifications, or attacks, maintaining the confidentiality and integrity of information.

Portability: The ability of the software to run on different platforms or environments without requiring extensive modifications. Portable software enhances flexibility and adaptability.

Availability: Refers to the percentage of time that the system is operational and accessible. Highly available systems minimize downtime and ensure continuous service.

Testability: Refers to the simplicity of testing the software to validate its functionality or detect defects. Testable systems support effective quality assurance processes.

Interoperability: The ability of the software to interact and function seamlessly with other systems or components, promoting compatibility and data exchange.

2.8 Cloud Computing

In the early days, most of the software was deployed in the on-premises environments, and the respective business owners managed it. On-premises traditional computing

model where a business owns and operates its own computing infrastructure, including servers, data centers, networking, and other hardware and software resources [115]. In an on-premises environment, all the IT resources are physically located within the organization's premises, typically within its own data centers. However, there are many challenges when running a private data center.

Challenges of maintaining on-premises data centers:

- High maintenance cost: Organizations are responsible for ongoing maintenance and upgrades that will require highly technical resources, which are more expensive. Also, many operational expenses include electricity, cooling, and staffing.
- Limited scalability: Scaling up an on-premises data centre requires additional hardware purchases and physical expansion. This process can be time-consuming and may lead to inefficiencies in resource utilization during periods of low demand.
- Security challenges: On-premises data centres are required to establish their own security on both sides, such as physical and cyber security.
- Limited flexibility: On-premises data centres offer limited flexibility compared to cloud solutions. Adapting to changing technological trends and quickly integrating new services can be challenging.
- Limited geographic redundancy: Achieving geographic redundancy for disaster recovery and caching in on-premises data centres can be challenging and expensive. Implementing redundant systems across different locations may not be feasible for most companies.

In summary, while on-premises data centres provide control and customization, they have notable disadvantages in cost, scalability, maintenance, and environmental impact. Choosing between on-premises or cloud solutions should depend on a comprehensive evaluation of the organization's specific requirements, financial limitations, and future objectives. Cloud computing is modern technology that's gaining popularity quickly. In simple terms, it means using computing services over the Internet. The idea of cloud computing emerged due to improvements and acceptance of existing technologies. According to a recent survey on cloud adoption, 77% of large and 73% of small and medium-sized companies already use the cloud [116]. The main goal of using the cloud is to take advantage of various computing methods. It helps reduce costs and lets clients focus on their business without worrying about information technology challenges.

Advantages of cloud computing:

- Better scalability: Cloud providers provide on-demand scalability. Which allows the operational team to easily scale up or down the application based on

traffic load. This elasticity guarantees optimal resource utilization and cost-effectiveness.

- Cost-effective: The cloud computing model removes the necessity for initial investments in infrastructure. Software companies can pay for their computing resources on a pay-as-you-go model, reducing overall infra costs.
- Better disaster recovery: Cloud providers implement robust backup and disaster recovery solutions. This ensures data redundancy and availability, reducing the risk of data loss and facilitating quick recovery in case of disruption.
- Seamless updates and maintenance: Cloud service providers handle system updates, maintenance, and security patches. This guarantees that users of cloud services consistently enjoy the most recent features and security protocols without requiring manual involvement.
- Less time to market: Cloud computing allows organizations to quickly deploy and experiment with new ideas and applications. This accelerated innovation and speed to market can provide a competitive advantage.
- Improved collaboration: Cloud-based collaboration tools enable real-time communication and file sharing among team members. This enhances collaboration and productivity in a distributed work environment.
- Enhanced portability: With containerization technology, any application can be deployed in any containerized environment.

While cloud computing provides various benefits, it is necessary for software enterprises to thoroughly consider factors such as government compliance, security, and data governance when adopting cloud services. There are three main categories in cloud computing: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [117]. Each category serves different purposes and provides specific benefits to users. Amazon Web Services (AWS), Google Cloud Platform (GCP), Microsoft Azure, and IBM Cloud are some of the examples for the cloud providers [118], [119], [120].

The progression of cloud computing signifies a transformative change in the domain of information technology, restructuring the way computing resources are provided and utilized. This literature review explores the historical context of cloud computing, tracing its roots to the early 2000s when the need for agile and cost-effective solutions became evident. Cloud computing delivers computing services over the internet, providing users with on-demand access to resources without the burden of extensive infrastructure management. Critical attributes such as on-demand self-service, resource pooling, extensive network access, rapid elasticity, and measured service synergistically contribute to the agility and efficiency that characterize cloud computing environments [121]. The examination categorizes cloud computing into service models (Infrastructure as a Service - IaaS, Platform as a Service - PaaS, Software as a Service - SaaS) and deployment models (public, private, hybrid, community cloud), thoroughly exploring their unique features and implications [122].

Examining the advantages, challenges, and emerging trends within cloud computing, this literature review lays the foundation for a deeper understanding of this dynamic and influential technology.

2.8.1 Public Clouds

Public cloud is a service that is provided to the general public by the cloud providers as a cloud service. In this model, resources are offered over the internet with a pay-per-usage basis which allows users to scale their cloud resources as needed without the requirement to invest in hardware [123]. Public cloud providers manage and pool resources efficiently to meet the varying capacity needs of cloud consumers. This form of cloud is accessible to anyone with an internet connection and is crafted to provide a wide array of services and capabilities. Data which is generated from the hosted applications are stored in their own virtual environments, but underline hardware is shared with the other consumers as well.

Advantages of using the public cloud

- One of the notable advantages of cloud computing is the assurance of data availability and continuous uptime. Cloud service providers implement robust infrastructure and redundancy measures, minimizing the risk of downtime and ensuring that data is consistently accessible to users.
- Public cloud platforms provide the advantage of continuous technical expertise around the clock. Users can depend on prompt assistance, troubleshooting, and expert support with dedicated support teams and resources, enhancing the overall reliability of services offered by public clouds.
- Public cloud provides on-demand scalability, allowing users to effortlessly adjust computing resources based on their needs. This flexibility ensures that organizations can efficiently manage varying workloads without the need for substantial upfront investments in hardware.
- The ease and cost-effectiveness of setting up cloud services stand out as key advantages. Public clouds typically offer straightforward onboarding processes, enabling users to deploy applications and services with minimal complexity. This ease of setup contributes to the accessibility of cloud computing for a wide range of users.
- Public cloud minimizes resource wastage by allowing users to utilize computing resources as needed. Unlike traditional infrastructure models where organizations might over-provision to meet peak demands, public cloud environments enable efficient resource allocation, avoiding unnecessary costs and environmental impact.

One notable notable drawback associated with public cloud services is concerns regarding data security [124]. As organizations entrust their data to third-party providers, there is a potential risk of unauthorized access, data breaches, or other security vulnerabilities. The shared nature of the public cloud infrastructure introduces

complexities in ensuring robust security measures, raising apprehensions among users about the protection of their sensitive information. Privacy emerges as another drawback of public cloud adoption. With data stored on servers owned and managed by third-party providers, there may be apprehensions about the privacy of sensitive information. Users may be uneasy about the potential for their data to be accessed, monitored, or used without their explicit consent, leading to privacy concerns that impact the willingness of individuals and organizations to fully embrace public cloud solutions.

2.8.2 Private Clouds

Private cloud belongs to a specific organization and they manage it by own. Unlike public clouds, the access to this infrastructure is restricted to authorized third parties or members of the organization, with no intention of offering services to the general public [125]. Private clouds can be hosted on or off-premises, commonly within the organization's data center. This type of cloud is employed when an enterprise seeks to utilize cloud services internally, such as sharing consumer data across different stores. Key features of private clouds include enhanced security compared to public clouds and potential cost savings, particularly when utilizing unused capacities in existing data centers. Private clouds utilize cloud interfaces, providing analogous tools to public clouds, which include self-service interfaces and automated management of computing resources. The capability to sell surplus capacities to affiliated companies additionally enhances resource utilization optimization. Additionally, a private cloud provides greater control over infrastructure and computational resources. While both private and public clouds offer similar technological advantages, the distinctive strength of a private cloud lies in its superior data security and privacy, making it a preferred choice for organizations prioritizing these considerations.

Main disadvantages in private could is that higher cost [126]. The assertion that private clouds inherently have higher costs compared to public clouds is contingent on various factors within the organizational context. One contributing factor is the substantial upfront capital investment associated with building and maintaining a private cloud infrastructure. The procurement and management of hardware, software, and networking equipment contribute to elevated initial costs. Ongoing operational expenses, including maintenance, upgrades, and staffing costs, can also contribute to the perception of higher costs with private clouds. Additionally, private clouds may face challenges in optimizing resource utilization, particularly if the infrastructure is not fully leveraged, leading to increased costs per unit of computation. Scaling a private cloud to accommodate fluctuating workloads may be more complex and costly compared to the inherent scalability of public cloud environments. The need for specialized expertise in operating a private cloud may necessitate investment in training or hiring, further contributing to elevated costs. However, it is essential to note that the cost dynamics between private and public clouds are nuanced, and the perceived higher costs of private clouds should be evaluated against potential benefits

such as greater control, enhanced security, and compliance with specific regulatory requirements.

2.8.3 Hybrid Clouds

Hybrid cloud mainly consists of two or more cloud models such as private clouds, public clouds and community clouds. Each cloud component retains its distinct identity but is interconnected through standardized or proprietary technology, facilitating the seamless portability of applications and data across them. A hybrid cloud configuration typically involves the integration of at least one private and one public cloud, presenting two primary approaches: collaborative engagement between a private cloud owner and a public cloud provider, or a strategic alliance formed by a public cloud provider with a vendor specializing in private cloud platforms [127]. This infrastructure allows organizations to manage some resources in-house and others externally, exemplified by scenarios where certain data, like HR and CRM information, is stored in a public cloud, while confidential data remains in a private cloud. The hybrid approach enables businesses to leverage the scalability and cost-effectiveness of public clouds without compromising the security of critical applications and data. This configuration is also known as hybrid IT, providing organizations with flexibility and optimization in their cloud strategies [128].

Advantages of hybrid cloud

- Cost efficiency: By outsourcing part of the infrastructure to public cloud providers, organizations reduce upfront investments in hardware, minimizing capital expenses.
- Flexible resource allocation: Public clouds provide cost-effective resources for temporary projects, eliminating the need for large infrastructure investments.
- Lifecycle efficiency: Organizations can use public clouds for development and testing while reserving private clouds for production, optimizing costs across the application lifecycle.
- Control and scalability: Hybrid clouds offer the control of private clouds with the scalability of public clouds, balancing security and flexibility.
- Cloud-bursting: During high demand, organizations can scale operations by utilizing public cloud resources, ensuring optimal performance and cost-efficiency.
- Enhanced agility: Leveraging public cloud capabilities increases organizational flexibility, enabling faster responses to business needs and boosting competitiveness.

Drawbacks of hybrid clouds

- Expanded attack surface: Extending the IT infrastructure beyond organizational boundaries increases the risk of cyber-attacks, as part of the

infrastructure falls under the service provider's control, complicating security management.

- Identity and access management: Extending enterprise identity systems to public clouds can simplify management but raises concerns about potential security compromises, requiring careful integration.
- Tool security: Managing hybrid clouds with integrated or third-party tools introduces security risks. Organizations must ensure consistent security practices across the hybrid environment.
- Data privacy and integrity: The transfer of data between private and public clouds in hybrid models raises concerns about privacy and data integrity, requiring strong privacy controls to protect sensitive information.

2.8.4 Community Clouds

A community cloud occupies a unique space in the cloud computing and positioning itself between public and private clouds. In contrast to a private cloud dedicated to a singular organization, the community cloud model is specifically designed for the collaborative use of two or more organizations that share common concerns regarding privacy, security, and regulatory adherence [129]. Taking inspiration from various computing paradigms, the community cloud model aims to incorporate distributed resource provision reminiscent of grid computing, distributed control characteristic of digital ecosystems, and sustainability principles derived from green computing [130]. This blend is designed to capture the advantages of cloud computing use cases while incorporating self-management advancements from autonomic computing. Notably, the community cloud envisions a collaborative infrastructure by replacing vendor clouds with the underutilized resources of user machines. In this community-driven paradigm, each node within the cloud ecosystem has the potential to play multiple roles, serving as a consumer, producer, and crucially, a coordinator. This approach reflects a vision of a more inclusive, sustainable, and collectively beneficial cloud computing environment, where community members actively contribute to and benefit from the shared resources of the community cloud.

2.8.5 Service Models of Cloud Computing

Cloud computing is further delineated by three foundational service models, each tailored to meet diverse user requirements. The Infrastructure as a Service (IaaS) model furnishes users with virtualized computing resources via the internet, providing flexibility and scalability sans the necessity for physical hardware ownership. Platform as a Service (PaaS) streamlines application development by presenting an inclusive platform, encompassing tools and services, to facilitate the seamless creation, deployment, and management of applications. Concluding the triad, Software as a Service (SaaS) delivers readily deployable software applications via the internet, obviating the need for installation and maintenance. Users can access these

applications directly through a web browser. Collectively, these service models constitute the foundational framework of cloud computing, affording users a range of options catering to specific requirements, spanning from infrastructure management to application development and software consumption [131]

Infrastructure as a Service (IaaS) model provides all kinds of virtualized computing infrastructure to the cloud consumers. Unlike traditional setups requiring physical hardware ownership, IaaS grants users the flexibility to access and oversee fundamental infrastructure elements, including virtual machines, storage, and networking, all delivered over the internet. This model introduces a scalable approach, enabling organizations to adjust their computing resources in response to demand, with the added benefit of paying only for the resources consumed. IaaS proves to be a cost-effective solution by eliminating the need for upfront investments in physical hardware, providing a dynamic and adaptable environment for enterprise businesses. MMU University conducted a research related to the report generation on resource utilization [132]. As an outcome, cloud consumers know how their resources are consumed and based on that they can take decisions on the cloud resources capacities for cost effective solutions. In the IaaS model, users typically retain control over operating systems, applications, and development frameworks, while the cloud provider takes charge of managing the underlying infrastructure, encompassing servers, storage, and networking components. Acting as a foundational layer, IaaS facilitates the construction and deployment of diverse applications, serving as a pivotal component in meeting the computing needs of various workloads.

When considering the PaaS, cloud provider provides all virtualized servers and platforms to the cloud consumer and cloud consumer only need to care about the application and application's data. PaaS offers users a comprehensive platform that includes tools, services, and frameworks, streamlining the entire application lifecycle [133]. Diverging from conventional development methodologies, Platform as a Service (PaaS) abstracts the intricacies associated with infrastructure management, enabling developers to concentrate exclusively on coding and application logic. This model provides a simplified environment where developers can create, test, and deploy applications without the burden of managing underlying infrastructure components. Ganesan and Devi highlighted the security concern related to the PaaS model due to the multi-tenancy architecture [134]. Because of the of the PaaS application used the same data storage mechanism to store the all-consumer application data and shows them to the consumers based on their ownership in the runtime. But PaaS accelerates the development process, promoting collaboration among teams, and facilitates the creation of scalable and resilient applications. With features like automated scaling and built-in services, PaaS enhances efficiency and reduces the time-to-market for applications. The platform's managed services cover aspects such as databases, middleware, and development frameworks, freeing developers from routine tasks and enabling them to concentrate on creating innovative solutions. PaaS is a powerful tool for organizations seeking agility and speed in application development, offering a

robust and streamlined platform for the creation and delivery of modern, cloud-native applications.

When considering the SaaS, cloud provider provides all software and hardware as a service and consumers can use that software with minimal customization. Unlike traditional software deployment, SaaS eliminates the burden of software maintenance, updates, and infrastructure management for users. Instead, these responsibilities are shouldered by the SaaS provider. This model enables a subscription-based paradigm, wherein users incur charges for the software on a recurring basis, typically either monthly or annually. SaaS offers unparalleled flexibility, enabling users to access applications from any device with an internet connection. This accessibility promotes collaboration, as users can seamlessly share and work on documents and projects in real-time.The Software as a Service (SaaS) model encompasses a broad spectrum of applications, including productivity tools, collaboration platforms, customer relationship management (CRM), and enterprise resource planning (ERP) systems [135]. SaaS has become a driving force in the modern business landscape, providing a scalable, cost-effective, and user-friendly approach to software consumption.

2.8.6 Cloud Computing Trends

In the ever-evolving landscape of cloud computing, transformative trends are redefining how organizations leverage digital infrastructure to drive innovation, efficiency, and scalability. As businesses navigate this dynamic environment, several key trends have emerged, each contributing to the evolution of cloud services. Many industries use cloud technologies to boost revenue and reduce IT costs [136]. From the strategic embrace of multi-cloud architectures to the integration of edge computing for real-time data processing, the cloud landscape is witnessing a paradigm shift. Simultaneously, serverless computing is gaining popularity, empowering developers to focus on coding while leaving infrastructure management to cloud providers. Containerization of the application and Cloud-Native development stands major trends in cloud computing and most of the enterprise grade software companies actively do research related to this [137]. Sustainability initiatives, including energy-efficient data centers, underscore commitment to environmentally responsible practices. In this narrative, we delve into these trends, exploring their implications for the future of cloud computing and how they collectively shape the digital landscape.

Containerization, with technologies like Docker, Containerd and container orchestration using Kubernetes, stands at the forefront of modern application development [138]. The transformative shift brought about by containerization revolves around encapsulating applications and their dependencies within lightweight, portable containers. These containers contribute to uniformity across varied environments, addressing the well-known challenge of "it works on my machine." Docker, as a leading containerization platform, has streamlined the packaging and distribution of applications, allowing developers to create, share, and deploy

applications seamlessly. Complementing containerization, Kubernetes emerges as a powerful orchestration tool, scaling, providing a robust framework for automating the deployment, and management of containerized applications. Kubernetes simplifies the complexities associated with deploying applications at scale, offering features such as load balancing, automated rollouts, and self-healing capabilities. The declarative configuration of Kubernetes allows developers to specify the desired state of their applications, enabling efficient scaling and maintenance. This trend not only enhances the efficiency of development and operations teams but also promotes scalability and portability. Applications can be developed and tested in a consistent environment, and then effortlessly deployed across various infrastructure landscapes, including on-premises data centers and multi-cloud environments [139]. The result is a more agile, resilient, and scalable application architecture, aligning with the demands of modern, cloud-native development practices. As organizations continue to adopt containerization and Kubernetes, the ecosystem around these technologies evolves rapidly. This evolution includes the emergence of specialized tools, best practices, and an active community contributing to the advancement of container orchestration. The combined impact of containerization and Kubernetes is reshaping how applications are built, deployed, and managed, contributing significantly to the efficiency and agility of modern software development lifecycles. Cloud-native development embraces a set of principles that prioritize scalability, resilience, and efficiency in the cloud environment. Key pillars of this approach include containerization, microservices architecture, and continuous integration/continuous deployment (CI/CD). Containerization is a foundational aspect, encapsulating applications and their dependencies into lightweight, portable containers. This methodology ensures consistency across diverse environments, fostering the creation of applications that can seamlessly transition from development to testing and, ultimately, production. Leading containerization platforms, such as Docker, have become integral tools, empowering developers to encapsulate their applications with all necessary dependencies, mitigating the infamous challenge of "it works on my machine." Microservices architecture is another pivotal element, advocating the decomposition of applications into smaller, independently deployable services. Each microservice operates as a self-contained unit, facilitating agility in development, deployment, and scalability. This approach enables organizations to iterate on specific services without affecting the entire application, fostering a modular and responsive development environment. Continuous Integration/Continuous Deployment (CI/CD) practices streamline the software delivery pipeline, ensuring that changes are automatically tested and deployed. CI/CD pipelines automate the integration of code changes, testing, and deployment processes, reducing manual intervention and accelerating the delivery of new features and updates [140]. This methodology aligns with the principles of agility, enabling development teams to respond promptly to changing requirements and deliver software at a faster pace. The integration of these cloud-native practices empowers organizations to embrace a DevOps culture, where development and operations teams collaborate seamlessly to deliver high-quality software. Cloud-native

development facilitates the creation of applications that are not only optimized for cloud environments but also inherently designed for scalability and resilience. As this paradigm becomes increasingly standard, organizations are poised to unlock new levels of efficiency, responsiveness, and innovation in their software development endeavors.

Serverless computing, or Function as a Service (FaaS), is witnessing widespread adoption due to its developer-centric approach [141]. Developers can focus solely on coding, letting the cloud provider handle the intricacies of infrastructure management. This model enhances efficiency, reduces operational overhead, and enables organizations to scale based on actual usage, leading to cost optimization. Dr. R. Arokia called serverless architecture is a revolution in the cloud computing domain in his research [142]. He has shown application development and deployment using the AWS Lamda functions and how it is been scaled based on the workloads. FaaS provided scalability quality attribute from the cloud provider end and consumer does not need do anything for that. But cloud consumers need to keep eye on the cost of the function deployment.

2.9 Concluding Remarks

The literature review in this chapter provides a thorough exploration of the transition from monolithic to microservices architecture, emphasizing the complexities and challenges encountered during migration. It begins by establishing a foundational understanding of monolithic and service-oriented architectures, then transitions to the rationale behind adopting microservices, highlighting the associated difficulties. The discussion on inter-service communication is particularly detailed, analyzing protocols such as HTTP/HTTPS, gRPC, and WebSocket. It also examines the factors that influence performance metrics in these communication methods, emphasizing the need for careful consideration when designing microservices-based systems. The review further explores software quality attributes crucial for maintaining the performance and scalability of microservices architectures. In the context of cloud computing, the chapter categorizes different cloud environments such as public, private, hybrid, and community clouds and their service models, underscoring their relevance to microservices deployment. Additionally, the review touches on emerging trends in cloud computing, illustrating how these trends shape the future of microservices architecture. In conclusion, this literature review establishes a critical understanding of the challenges and opportunities in microservices architecture, particularly in relation to inter-service communication and cloud computing. These insights lay the groundwork for the next chapter, which will discuss the research methodology employed to develop an optimized strategy for inter-service communication in cloud-native microservices.

CHAPTER 3

METHODOLOGY

3.1 Introduction

This section discusses the methodology for building the optimized strategy for inter-service communication in microservice architecture that is compatible with cloud-native environments and cloud-native concepts. This strategy will help microservices to communicate with other microservices in a low-latency manner. Further, this model will help enhance cloud-native concepts and critical software quality attributes. In this research endeavour, an exhaustive examination will be conducted, encompassing qualitative and quantitative analyses across diverse research realms and technological domains. This comprehensive exploration will span the intricate landscapes of microservices architecture, cloud computing research domains, microservices frameworks, inter-service communication mechanisms, and application transport models. Chapter 2 will meticulously unravel a rich tapestry of existing research through a literature review, critically assessing similar endeavours in this field. This scrutiny is aimed at extracting valuable insights into how emerging technologies have been judiciously applied to surmount challenges and achieve success in previous research initiatives. The subsequent focus in Chapter 3 will be on microservices and communication protocols, entailing a meticulous comparative analysis. Drawing upon the wealth of knowledge derived from prior research, this research aspires to integrate cutting-edge technologies for optimal research outcomes. Furthermore, the research framework will be elevated through the introduction of innovative methodologies tailored to enhance the overall research landscape.

3.2 Evolutionary Analysis of Microservice Architecture

By considering the users' requirement complexity and dynamic behavior, most of the software development entities are changing their software development architecture from monolithic or SOA-based architecture to microservice-based architecture. The previous chapter discussed migration models from monolithic architecture to microservice architecture. We conducted a thorough examination and comparison of numerous studies on microservices, focusing on their practical applications. The primary objective was to gain a comprehensive understanding of the evolution of microservices, tracing their development from the initial stages to their current state. By employing contemporary and widely accepted research methodologies, we systematically mapped the landscape of microservices, providing insights into their growth, adoption, and technological advancements over time.

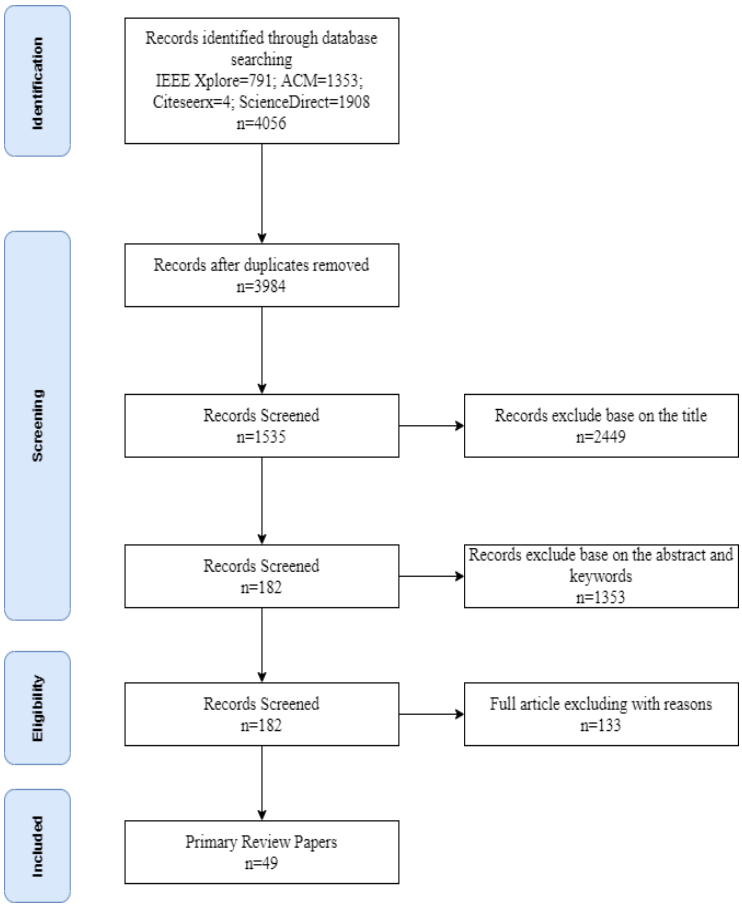

Fig. 3.1: PRISMA model analysis

Using the Preferred Reporting Items for Systematic Reviews and Meta-Analysis (PRISMA) model, a semantic review was conducted regarding microservices to find the answers to the research questions below [143].

Question 01 - Why do organizations choose to transform monolithic applications into microservices, and what drives this shift in architecture?

The aim here is to identify issues that people face with other architectures, and the main intention to move to the microservice architecture. The benefit of identifying is that when moving with the microservice architecture, people can focus more on those areas to improve the technology and the methodologies.

Question 02 – What technologies and architectural patterns are employed in microservices?

Get an understanding of the technologies and different architectural patterns that are used in microservices development and deployment. This will help new developments to incorporate existing microservices and deployments.

Question 03 – What challenges and difficulties are encountered by individuals in the software development life cycle when adopting a microservices architecture?

The most critical facts that have to be identified in the microservices architecture will help to find a solution for overcoming the challenges and issues.

Utilized ScienceDirect, ACM, IEEE Xplore, and Citeseerx digital libraries to find the answers to the above research questions. The rationale for selecting these repositories is their reputation for hosting high-quality research and providing user-friendly accessibility. Additionally, these research databases offer advanced search capabilities, allowing for efficient exploration of publication metadata. From these databases, we specifically selected published research papers and journal articles by manually preprocessing them. When selecting the papers, use the include and exclude criteria based on the keywords during the preprocessing stage.

Search String –

("monolithic application to microservice architecture") OR ("microservice architectural patterns") OR ("microservice framework" OR "microservice development tools") OR ("Challenges of microservice") AND ("Microservices" OR "Microservice" OR "micro-service")

The primary investigations were carried out utilizing influential digital libraries and renowned conference proceedings, ensuring a robust foundation for addressing the research inquiries. The search was confined to online library search engines accessible through university subscriptions. No specific time constraints were imposed on the search process. Additionally, it was observed that research on microservices commenced in the 2000 decade.

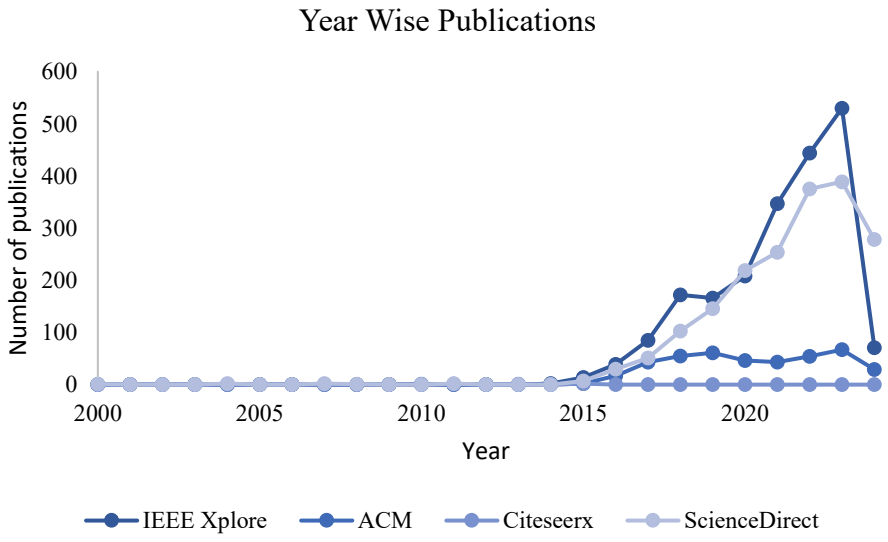

Fig. 3.2: Microservice publication distribution

According to the results, we found that a small number of research was conducted during the year 2000 to 2015. However, from the year 2015, most of the research has been conducted in the area of microservices. In this time frame, most of the

technologies have emerged, and microservices-related research has also been conducted from different perspectives, such as microservices reference architectures, microservices frameworks, language microservices supports, microservices migrations strategies, and so on.

Analysis indicates that microservices architecture is emerging as the next-generation approach to software design, with a growing emphasis on deployment in cloud-native environments. Future studies should prioritize enhancing inter-service communication to further optimize performance, especially within cloud-native systems.

3.3 Key Considerations for Optimizing Inter-Service Communication

The objective of this research is to develop an efficient strategy for inter-service communication within microservices deployed in cloud-native environments, with a focus on enhancing performance during the migration from monolithic architectures to microservices, or within existing microservice applications. The research aims to minimize the performance impact associated with message exchange between microservices. Given the dynamic nature of modern networks, the proposed solution considers several key factors identified through a comprehensive literature review.

Several key factors have been identified by reviewing other similar research as critical to developing an efficient strategy for inter-service communication in microservices. These include network latency, which encompasses network congestion and bandwidth considerations, as well as protocol overhead, which involves the choice of communication protocols and the style of communication (synchronous/asynchronous). Additionally, the overhead associated with serialization and deserialization in programming plays a significant role. Our solution also ensures native compatibility to function effectively within diverse cloud-native environments. Finally, preserving software quality attributes throughout the implementation is essential to maintaining the integrity and performance of the system.

3.3.1 Technique to Reduce Protocol Overhead and Improve the Network Latency

Hybrid approaches combine synchronous and asynchronous communication to leverage the benefits of both styles to improve inter service communication in the microservice architecture. By combining synchronous and asynchronous methods, hybrid approaches can optimize performance based on the context [144]. Usage wise synchronous communication might be used for critical, immediate interactions like day-to-day transactions related applications while asynchronous methods handle less time-sensitive tasks. But most of the B2C applications are based on synchronous communication because customer’s needs the response immediately. Hybrid communication can strike a balance between coupling and flexibility, ensuring critical

services remain tightly coordinated while others remain loosely coupled for scalability and maintainability. But using the hybrid approach can increase system complexity, requiring careful design and implementation to avoid potential pitfalls related to coordination and consistency. This complexity needs to be encapsulated with the solution strategy and for the user it should be a synchronous communication pattern and in underline communication can done as an asynchronous communication style. This kind of hybrid approach can bring more advantages over the performance and scalability to the microservice architecture. The choice of communication style profoundly impacts inter-service communication in microservices architecture. Synchronous communication offers simplicity and direct interaction but can lead to latency and tight coupling issues. Asynchronous communication provides non-blocking interactions and scalability benefits but introduces complexity in ensuring consistency and reliability. Hybrid approaches aim to leverage the strengths of both styles, though they require meticulous design to manage the increased complexity effectively. Understanding these impacts through the lens of distributed systems and software architecture theories is crucial for designing efficient and resilient microservice-based systems.

The Transmission Control Protocol (TCP), located at the transport layer (Layer 4) of the OSI model, is crucial for enabling reliable, connection-oriented communication between services. In our approach to inter-service communication, we leverage TCP-based communication to transmit messages efficiently. By sending messages in binary form over TCP, we can significantly reduce network bandwidth usage while maintaining reliability and ensuring accurate delivery. It also provides the benefits of lightweight, connectionless protocols. Got deep understanding of the TCP layer and its interaction with the application layer help for optimizing communication performance. When treating with TCP-based protocol, it is important to implement mechanisms that ensure message reliability and guarantee accurate delivery.

3.3.2 Techniques to Minimize Programming Overhead to Users

Clear design principles, adherence to best practices, and thoughtful consideration of communication patterns and protocols are essential to navigating and mitigating the impact of programming complexity in the microservices ecosystem. Effective use of abstraction and encapsulation can reduce programming complexity by providing simplified interfaces for inter-service communication, aligning with the abstraction principle by focusing on higher-level operations [145]. In our approach, all serialization, deserialization, and message conversion tasks are handled at the library level, shielding developers from the associated complexity. This minimizes the cognitive load, supports modularity and separation of concerns, and simplifies dependency management. By adopting simpler communication patterns and leveraging robust tools and frameworks, we can further reduce complexity, enhancing system maintainability, scalability, and performance. Understanding these impacts

through software engineering and distributed systems theories is essential for designing efficient and resilient inter-service communication strategies.

3.3.3 Ensuring Cloud Native Compatibility

Ensuring cloud-native compatibility is crucial for both the design and efficiency of inter-service communication in microservice architectures operating in cloud environments. Cloud-native principles, such as containerization, dynamic scaling, and resilience, influence how services communicate and interact. Cloud-native applications are designed to leverage cloud computing frameworks and are typically built using microservices architecture, which divides applications into small, independent services that communicate over a network. Cloud-native applications can scale out by adding more instances of services. Efficient inter-service communication protocols must support this scalability by handling increased traffic without performance degradation. The failure recovery principle emphasizes that cloud-native applications should be resilient to service failures. Communication protocols must support features like automatic retries, failover mechanisms, and load balancing to maintain service availability and reliability. Container orchestration platforms enforce network policies that can impact inter-service communication. Hence, we need to use the well-known communication strategy in order to consume cloud resources. The observability principle highlights the importance of tracing and logging inter-service communication to monitor performance, detect bottlenecks, and diagnose issues. Choosing protocols should support tracing to provide better insights into the communication flow. Cloud-native compatibility profoundly impacts inter-service communication in microservice architecture. Principles such as containerization, dynamic scaling, and observability drive the need for robust, scalable, and resilient communication protocols. Understanding these impacts through theories and principles from distributed systems and cloud computing is essential for designing efficient inter-service communication strategies. Protocols must support dynamic environments, facilitate easy service discovery and load balancing, ensure resilience and observability, and integrate seamlessly with DevOps practices to meet the demands of modern cloud-native applications.

3.3.4 Software Quality Attributes Preservation

Inter-service communication in microservice architecture plays a pivotal role in shaping various software quality attributes within microservices or distributed architecture. The reliability of a system is influenced by inter-service communication mechanisms that demonstrate resilience to connection failures and errors, incorporating retry strategies and robust error handling. Scalability is facilitated by inter-service communication patterns that support horizontal scaling, enabling services to adeptly handle increased workloads through efficient load balancing and communication protocols. Performance is intricately tied to inter-service communication, as the choice of communication protocols directly impacts latency,

with efficient mechanisms such as asynchronous communication contributing to low-latency interactions. Well-designed specifications and versioning strategies foster maintainability. Availability is maintained through inter-service communication strategies supporting redundancy and failover mechanisms, ensuring continuous service operation. Testability is enhanced by inter-service communication mechanisms that facilitate easy testing through support for mocking and simulation. Flexibility is achieved by promoting loose coupling between services in inter-service communication, allowing for independent evolution. Interoperability is supported by adhering to standard communication protocols, ensuring compatibility across diverse technologies. Monitoring and diagnosability in inter-service communication involves logging and tracing mechanisms for effective system health analysis. Addressing these software quality attributes in the realm of inter-service communication is essential for constructing resilient and high-performance microservices or distributed systems.

3.4 Sustainable Framework

To achieve efficient inter-service communication in a cloud-native microservices architecture, a hybrid approach is proposed, leveraging both synchronous and asynchronous communication techniques. This framework addresses key challenges such as network resource consumption, programming complexity, and latency, while ensuring cloud-native compatibility. Frameworks bring powerful tools that can significantly enhance overall application performance and development process by boosting efficiency, improving code quality, fostering collaboration, and promoting reusability. By carefully considering the advantages and potential trade-offs, engineers can leverage frameworks to build robust, maintainable, and high-performing microservice applications.

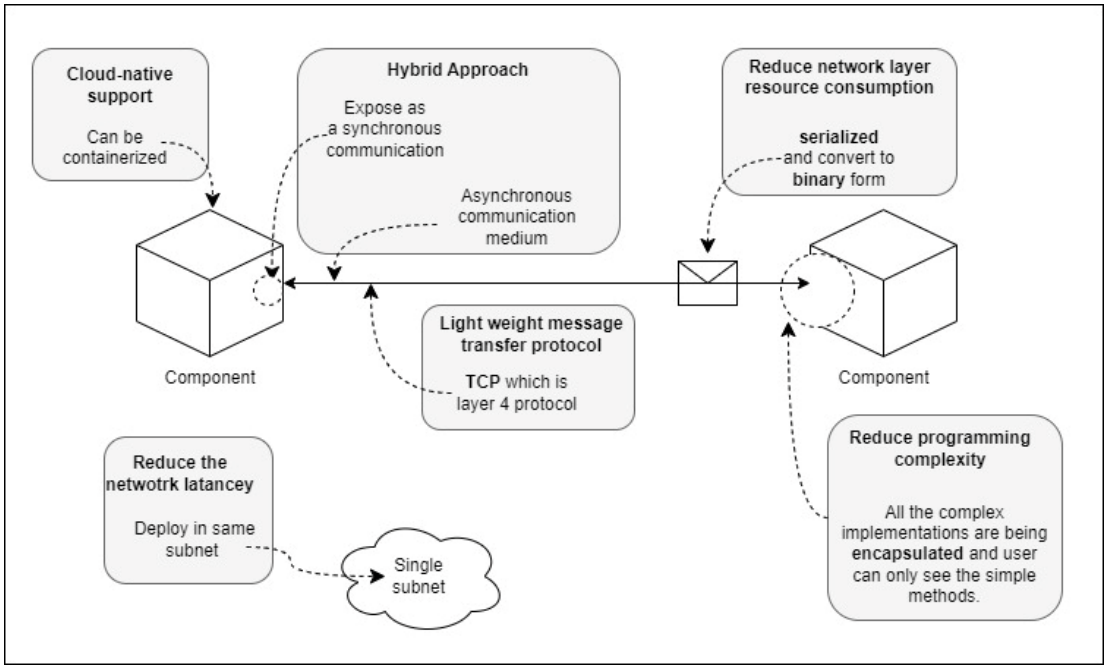

Fig. 3.3 Sustainable framework for the improvement of inter service communication

The framework adopts a hybrid approach where inter-service communication is exposed synchronously at the programmatic level but utilizes asynchronous communication mediums behind the scenes. This allows developers to interact with services in a straightforward, request-response manner while benefiting from the performance and reliability advantages of asynchronous messaging systems. By decoupling service interactions, the system can handle varying loads more gracefully and recover from transient failures more effectively. To minimize network layer resource consumption, messages are serialized and converted to binary form. Binary serialization, as opposed to text-based formats, significantly reduces the size of messages, leading to faster transmission and lower bandwidth usage. This optimization aligns with principles of data serialization and compression, enhancing overall communication efficiency. The framework encapsulates complex implementations, exposing only simple methods to the users. This adheres to the principles of abstraction and encapsulation, reducing cognitive load on developers and simplifying the integration of services. By hiding the intricate details of inter-service communication, the framework allows developers to focus on business logic rather than communication intricacies, improving productivity and reducing the likelihood of errors. A lightweight message transfer protocol based on the TCP layer 4 protocol is utilized. TCP provides reliable, ordered, and error-checked delivery of a stream of bytes, which is essential for maintaining data integrity and ensuring message delivery. This choice balances the need for reliability with minimal overhead, supporting efficient communication between services. To further reduce network latency, services are deployed within the same subnet. Keeping communication local within a subnet minimizes the number of networks hops and reduces latency compared to cross-subnet or cross-region communication. This approach leverages principles of network topology and proximity, ensuring faster and more reliable service interactions. The framework is designed to be fully compatible with cloud-native environments and can be containerized using modern technologies. Containerization ensures consistent environments across development, testing, and production, simplifies deployment, and enhances scalability. This aligns with cloud-native principles, facilitating seamless integration with modern DevOps practices and enabling dynamic scaling and orchestration.

In summary, this framework for optimized inter-service communication integrates synchronous and asynchronous techniques to balance simplicity and performance. By employing binary serialization, encapsulating complexity, using a lightweight TCP-based protocol, minimizing latency through subnet deployment, and ensuring cloud-native compatibility, the framework addresses key challenges in microservice communication. It leverages established theories and principles from distributed systems, software engineering, and cloud computing to enhance efficiency, reliability, and scalability in cloud-native environments.

3.5 Solution Architecture

After carefully considering the above points, we designed the below architecture for achieving the research target.

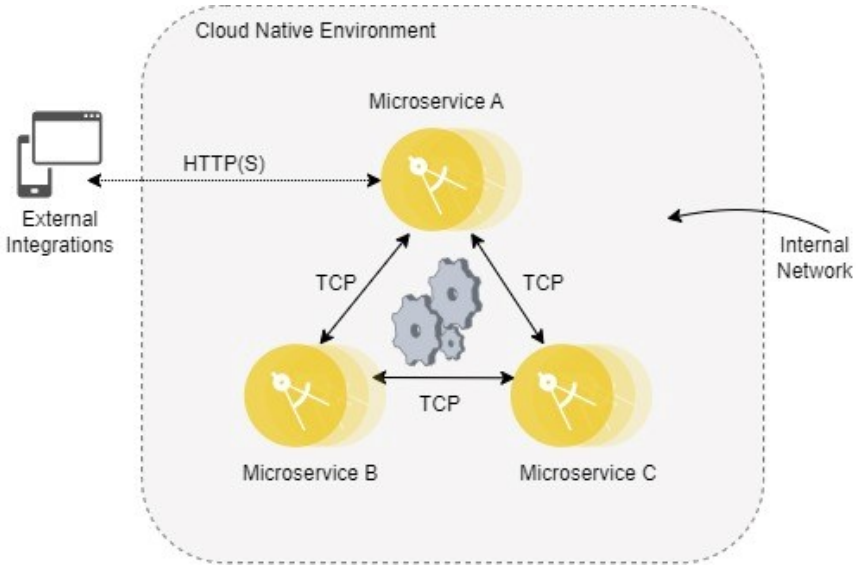

Fig. 3.4: High-level solution architecture

The proposed solution architecture leverages a hybrid approach to optimize inter-service communication within microservice architecture deployed in a cloud-native environment. By integrating synchronous and asynchronous communication methods, we achieve a balance between ease of use and performance, addressing key challenges such as network latency, resource consumption, and programming complexity. All microservices are deployed within the same subnet to minimize network latencies. This local proximity reduces the number of network hops required for communication, thereby significantly lowering latency and enhancing performance. For interactions with external systems, microservices maintain proper interfaces, typically using RESTful APIs over HTTP(S) protocols. This ensures compatibility and ease of integration with a wide range of external systems, which often rely on standardized web communication protocols. Internally, microservice-to-microservice communication utilizes the TCP protocol. TCP's reliable and ordered delivery mechanism is crucial for maintaining data integrity across service interactions. However, to leverage TCP effectively, we need to use lightweight middleware that supports TCP-based streams. This middleware ensures message integrity and guarantees exactly once delivery, a critical feature for maintaining consistency in distributed systems. The middleware operates independently and is scalable, ensuring it does not become a bottleneck for microservice scalability. The middleware facilitates real-time streaming on top of the TCP layer, establishing robust connections between microservices. It should supports message serialization and conversion to binary form and solutions we should abstracting this complexity from the end users.By

doing so, developers interact with simple, high-level methods while the middleware handles the efficient transmission of binary messages. This encapsulation adheres to the principles of abstraction and encapsulation, reducing programming complexity and enhancing developer productivity. The solution architecture is designed to be fully cloud-native, ensuring compatibility with containerization technologies like Docker and orchestration platforms like Kubernetes. This allows for consistent environments across different stages of development and deployment, facilitating dynamic scaling and automated management. The middleware, being lightweight and independently scalable, integrates seamlessly into this cloud-native ecosystem, supporting the scalability and resilience of the overall microservices architecture.

In summary, the proposed solution offers an innovative approach by utilizing a TCP stream-based request-response mechanism combined with lightweight middleware to optimize inter-service communication in microservices. Unlike conventional methods like RESTful APIs over HTTP, which can lead to increased network latency and resource consumption, or gRPC, which introduces programming complexity and higher maintenance effort, this solution strikes a balance between efficiency and simplicity. By leveraging TCP for internal communication, it achieves fast, reliable data exchange while minimizing network usage. The lightweight middleware ensures exactly-once message delivery, avoids scalability issues, and abstracts the complexities of binary serialization, making it more developer-friendly. Additionally, the solution's cloud-native design is fully compatible with containerization and orchestration platforms, offering a scalable, adaptable, and future-proof approach for modern microservices.

3.6 Technology Selection

Stream communication method has been chosen to implement the solution by considering the above-mentioned factors. Stream communication refers to the process of transmitting data continuously or in a continuous flow between two entities, typically over a network or a communication channel. Unlike traditional discrete message-based communication, where data is sent in distinct packets or messages, stream communication involves the continuous transmission and reception of data without predefined boundaries. In stream communication, data is sent and received continuously, with no distinct start or endpoints. This continuous flow of data allows for real-time processing. Stream communication can be bidirectional, meaning data can flow in both directions between the sender and receiver. This enables interactive communication where both parties can send and receive data simultaneously. Stream communication can be more efficient than message-based communication for transmitting large volumes of data or for scenarios where data needs to be processed as it is received. By avoiding the overhead associated with message headers and acknowledgments, stream communication can achieve higher throughput and lower

latency. TCP communication protocols support stream communication. In event-driven architecture, stream communication is employed for event sourcing, where events are continuously appended to a log or stream. This approach enables systems to reconstruct states by replaying events in the order in which they occurred. Stream communication implementations follow a publish-subscribe model, where publishers send messages to a topic or channel, and subscribers receive those messages in real-time. This model supports scalable and decoupled communication.

In the context of microservice's inter-service communication, ensuring reliable message delivery and achieving exactly once delivery semantics are of paramount importance for the overall integrity and consistency of distributed systems. These aspects play a crucial role in maintaining the accuracy of data sharing between services and preventing issues. Previous research has shown that plain TCP streams do not provide guaranteed message delivery [146]. Based on the findings of previous research, we did not pursue evaluating the error rate of message delivery in plain TCP streams. Instead, we focused on alternative solutions to guarantee reliable message delivery. When utilizing a TCP-based stream communication method, it is essential to incorporate a mechanism that ensures message delivery and provides traceability during transmission. So, we have chosen Redis 5.0 to achieve those tasks. Redis streams is designed to support real-time, scalable, and distributed message processing. A stream is essentially an ordered list of messages, where each message is uniquely identified within the stream by a stream ID. Redis Streams enable asynchronous communication between microservices [117]. Microservices can publish messages to a stream asynchronously, and other services can consume and process those messages at their own pace. This helps decouple services and improve overall system responsiveness. Redis streams provide reliable message delivery, ensuring that messages are stored in the order they are received [147]. This reliability is crucial in distributed systems where the order of messages and the prevention of message loss or duplication are essential for maintaining consistency. Events and messages flowing through Redis Streams can be monitored and analyzed for various purposes, such as system health monitoring, performance analytics, and troubleshooting. Redis has its own communication protocol for stream communication. Redis Serialization Protocol (RESP) is the binary protocol used by Redis, and it is a lightweight and efficient protocol designed to be simple [118]. This binary format is more compact and efficient for data transfer between microservices. Network layer resource consumption is very low when transferring the message between the microservices compared to the test-based transmission. This efficiency is significant in scenarios where minimizing data transfer is crucial for performance. Also, RESP provides a precise mechanism for error handling.

Most popular Java programming languages have been chosen to build the research component, and Spring boot was selected for the framework for microservice development. Spring Boot is highly favored for designing microservices due to its emphasis on simplicity and convention-over-configuration approach. This framework

streamlines the development process by providing defaults and reducing the need for extensive manual configuration. It is specifically tailored for microservices architecture, offering embedded web servers like Tomcat or Jetty, eliminating the complexities associated with external server configurations. Leveraging the extensive Spring ecosystem, Spring Boot provides a comprehensive set of tools, libraries, and modules, facilitating the development, testing, and management of microservices. The framework incorporates core Spring principles such as dependency injection and Inversion of Control, fostering the creation of loosely coupled and maintainable microservices. Spring Boot further simplifies configuration management with externalized configuration files, and its integrated testing support ensures thorough testing at various levels. The Actuator module in Spring Boot offers built-in endpoints for monitoring and managing microservices at runtime, providing valuable insights into health, metrics, and environment properties. Spring Boot is widely acknowledged for its commendable performance, owing to several key design principles and optimizations. With a focus on fast startup times, the framework is particularly well-suited for microservices and cloud-native environments. Annotation-based configuration minimizes verbosity in XML configurations, contributing to streamlined code and runtime efficiency. Building upon the well-optimized Spring Framework, each version of Spring Boot undergoes continuous improvements to enhance performance.

Container orchestration systems play a crucial role in the deployment and management of microservices-based applications, offering several advantages that contribute to the scalability, reliability, and efficiency of the microservices architecture. We have chosen Kubernetes as a container orchestration system to deploy the developed microservice to match the cloud-native concepts. Kubernetes, as a widely adopted container orchestration system, aligns seamlessly with cloud-native concepts, offering several advantages for developing and managing microservices in cloud environments. Kubernetes excels in orchestrating containers and providing automated deployment, scaling, and management. This enables seamless deployment and operation of microservices in a distributed and scalable manner. Kubernetes optimizes resource utilization through efficient scheduling and scaling. It ensures that microservices receive the necessary resources, preventing underutilization or overutilization of resources in the cluster. Kubernetes is designed to be cloud-agnostic, allowing it to run across various cloud providers or in hybrid cloud environments. This flexibility aligns with cloud-native principles, providing organizations with the freedom to choose the most suitable infrastructure. Kubernetes integrates seamlessly with continuous deployment (CD) and continuous integration (CI) pipelines. This ensures smooth and automated delivery of microservices, facilitating rapid development cycles. Kubernetes aligns closely with cloud-native principles and offers a robust platform for deploying, managing, and scaling microservices in modern cloud environments. Its flexibility, automation capabilities, and support for cloud-native concepts make it a preferred choice for organizations adopting microservices architectures in the cloud.

3.7 Concluding Remarks

In this chapter, the methodology for developing an optimized strategy for inter-service communication in microservices architecture has been thoroughly outlined. Beginning with an evolutionary analysis of microservice architecture, the chapter identifies the key considerations essential for enhancing inter-service communication. Techniques to reduce protocol overhead, improve network latency, and minimize programming overhead were explored, alongside strategies for ensuring compatibility with cloud-native environments and preserving critical software quality attributes. The chapter also introduced a sustainable framework designed to support the optimization process, providing a structured approach to address the identified challenges. This framework is further supported by a detailed solution architecture, which serves as the blueprint for implementing the optimized strategy. The careful selection of technologies is emphasized, ensuring that the chosen tools and platforms align with the objectives of performance improvement and other critical quality attributes of the microservice architecture. In summary, this chapter establishes a robust methodological foundation that guides the research towards achieving its goals. The insights and frameworks developed here set the stage for the next chapter, which will delve into the proposed solution, detailing the specific techniques and architectural decisions that will be employed to enhance inter-service communication in cloud-native microservices environments.

CHAPTER 4

PROPOSED SOLUTION

4.1 Introduction

In this section, we propose an optimized strategy for inter-service communication in microservices by mainly focusing on implementing the proposed strategy. The following characteristics are targets to achieve from the proposed strategy,

- Reduce the inter-service communication latency
- Improved the overall application throughput
- Less complexity to use
- Enhance the software quality attributes
- Preserve cloud-native concepts

In the implementation chapter, we are diving into putting a research plan into action for how different services talk to each other in a cloud-based setup designed for microservices. In the preceding chapter, a comprehensive examination was conducted, describing the chosen technologies ready to support this implementation. Building upon this groundwork, the forthcoming discourse traverses the practical indication of these selected technologies within the context of inter-service communication in a microservices architecture. Through careful planning and thoughtful execution, this chapter aims to explain how to create an optimized communication framework that enhances the performance of microservices interactions in a cloud-native environment. The implementation chapter of this thesis delves into the practical application of an optimized strategy for inter-service communication within a cloud-native environment tailored for microservices. In the preceding chapter, an exhaustive analysis was conducted to meticulously select the technologies that will serve as the foundation of this implementation. Now, our focus shifts toward achieving several paramount objectives with our proposed strategy. Firstly, we aim to focus on inter-service communication latency, ensuring swift and efficient data exchange between microservices. Concurrently, we endeavour to enhance the overall application throughput, ensuring substantial performance improvement. Moreover, our strategy is geared towards simplifying the intricacies of inter-service communication, thereby reducing complexity and facilitating ease of use for developers. Additionally, we are committed to elevating various software quality attributes, including reliability, scalability, and maintainability, through deploying our optimized communication strategy. Importantly, amidst these objectives, we remain steadfast in preserving the fundamental principles of cloud-native architecture, ensuring seamless integration with the cloud environment. Throughout this implementation chapter, a meticulous plan will be executed, detailing the deployment of our strategy and addressing each targeted objective comprehensively. Through strategic execution and careful planning,

we aim to achieve these objectives and enhance the performance and reliability of microservices within the context of a cloud-native ecosystem.

4.2 High-level Architecture

The high-level system architecture provides a bird's-eye view of the entire distributed system, outlining its major components and how they interact to fulfil the overall system's functional and non-functional requirements. It outlines the interactions between components, including data flow and communication protocols, while addressing software quality attributes. Altogether, this architectural overview serves as a vital reference for stakeholders and developers, aiding in understanding the system's functionality, performance, and scalability, thus facilitating effective decision-making throughout the development and maintenance life cycle.

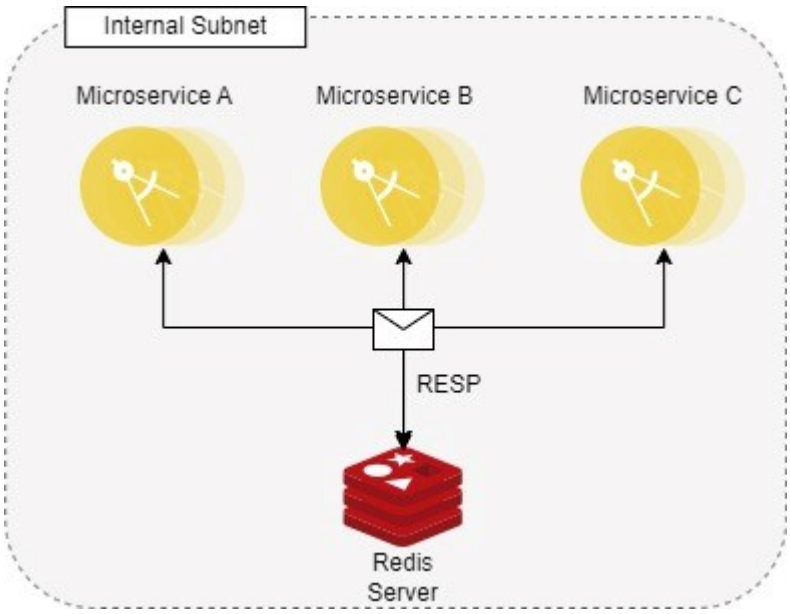

Fig. 4.1 High-level architecture

The above images 4.1 describe the high-level system architecture for microservice architecture with the research solution. Microservices A, B, and C are independent microservices deployed in the same network with the same subnet. Microservices need to communicate with each other in order to produce the system's functional requirements. The Redis server is also placed in the same network in which microservices are deployed, and all microservices are connected to the Redis server. So, the Redis server ensures all communication reliability and the exact one delivery to the respective microservices. Inter-service communication uses the Redis serialization protocol (RESP), a TCP-based low-latency protocol. Per the high-level architecture, each microservices can scale up and down independently without any issues. TCP connecting is maintained as a TCP stream between microservices and the

Redis server. Utilizing Redis streams for inter-service communication in a microservices architecture can offer several benefits, enhancing efficiency and scalability. Redis streams provide a high-performance, distributed message broker that facilitates real-time communication between microservices. By leveraging Redis streams, microservices can communicate with each microservices and share data, enabling decoupling and improving the overall reliability and responsiveness of the system. This approach also aligns well with cloud-native principles, as Redis can be deployed as a managed service in cloud environments, offering scalability and reliability benefits inherent to cloud platforms. Redis streams provide a robust foundation for building resilient and responsive microservices architectures.

4.3 High-level Component Architecture

The high-level component architecture provides an overview of a system's primary building blocks or components and how they interact to fulfill the system's functionalities. A high-level component architecture offers numerous advantages in developing and maintaining software systems. The system is divided into separate components, each handling specific functions, which promotes modularity. This approach simplifies understanding, development, and maintenance of individual parts without impacting the entire system. Furthermore, the modular design allows for flexibility, making it easier to adapt and adjust to changing requirements. With well-defined interfaces and clear responsibilities for each component, maintenance becomes easier, allowing for localized changes and updates without extensive regression testing. High-level component architecture also promotes collaboration among development teams, enhances testing capabilities, facilitates interoperability, and ensures resilience to change, making it well-suited for complex and dynamic environments. Together, these components form a cohesive structure, allowing the microservice to fulfil its purpose efficiently while adhering to principles of modularity and separation of concerns. This architecture enables self-contained, maintainable, and scalable microservices within a larger ecosystem. Overall, the high-level component architecture provides a blueprint for designing scalable, resilient, and maintainable systems by breaking down the system into modular and composable components, each with well-defined responsibilities and interfaces.

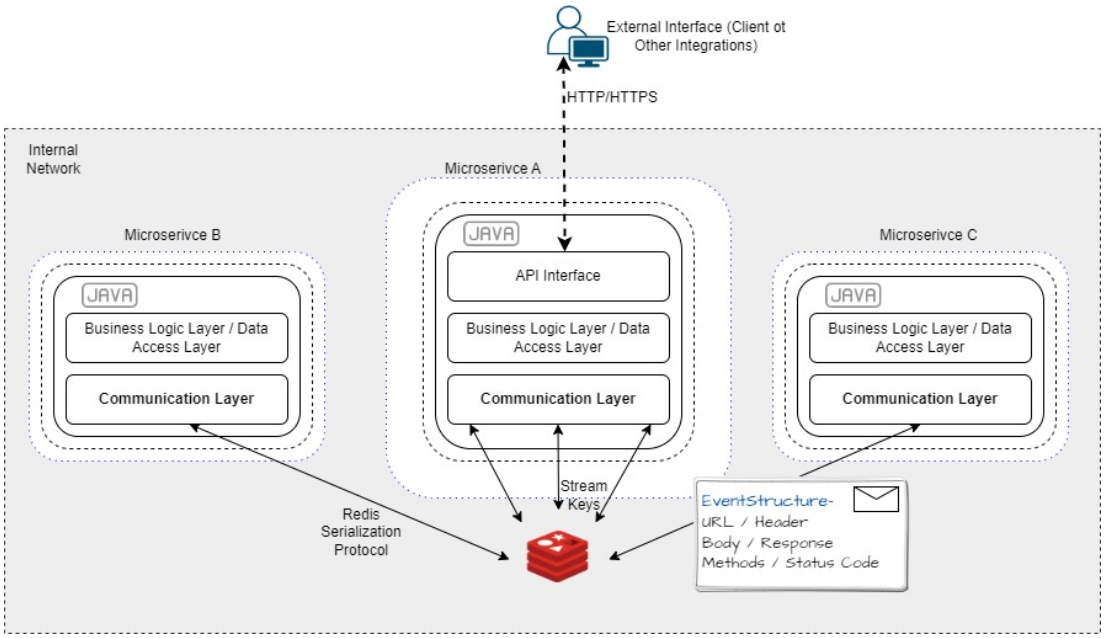

Fig. 4.2: High-level component architecture

The figure 4.2 above illustrates the component architecture of the microservices, along with their interfaces to external clients and internal services. In this research, the communication layer was implemented with the research component. During the solution design phase, microservices were organized into internal components, including the API interface layer, data access layer, business logic layer and communication layer. This structure allowed for the communication layer to be separated from the other layers. The implementation focused primarily on the communication layer, leaving the API, business logic, and data access layers unchanged. This approach was chosen to ensure that changes to the API layer did not impact external microservice consumers, as altering the API layer would not be the most efficient way to introduce new developments. The implementation followed industry standards like HTTP client libraries for inter-service communication, relying on a request/response-based stateless client. However, behind the scenes, the client was designed to use the publisher subscriber model and Streams without requiring developers to manage these complexities, thereby simplifying development. Redis Streams were used for stream communication with the Redis server, facilitated by the Spring Boot starter data Redis library, a popular dependency within the Spring Boot framework that provides both high-level and low-level abstractions for integrating with Redis. The communication layer consisted of three primary components. The consumer was responsible for processing messages and managing responses according to subscriptions. The publisher transmitted messages to other microservices via the Redis server using the stream key. The stream subscription handler managed all stream subscriptions. Eventstructure objects were used to transmit relevant information over the network in a binary format.

Overall, the implemented communication layer, comprising components for message consumption, message publishing, and stream subscription handling, has significantly

improved the robustness and scalability of the microservices architecture. By adhering to industry’s best practices and leveraging cutting-edge technologies, we have successfully enhanced the system's communication capabilities, laying a solid foundation for future development and innovation in the microservices ecosystem. Further, in the following topic, we detail the low-level design of the solution provided.

4.4 Low-level Design

Low-level design (LLD) is the phase of software design where the system's architectural concepts are translated into detailed specifications for individual components or modules. It involves designing each component's internal structure and behavior, specifying how it will function and interact with other components to fulfill the system's requirements. In low-level design, developers focus on the implementation details. This level of design elaborates on the high-level architecture defined during the system design phase and provides the blueprints for building the software system.

During the low-level architecture phase, the below activities are considered,

- Designing interfaces between different modules or components.
- Specifying input/output formats and protocols for data exchange.
- Designing error-handling mechanisms and exception-handling strategies.
- Deciding on resource allocation and optimization strategies for performance and efficiency.
- Identifying dependencies between components and managing coupling and cohesion.

LLD offers several advantages, specifically tailored to microservice inter-service communications. With LLD, developers gain granular control over the design of communication protocols and mechanisms, ensuring optimized interactions between microservices regarding performance, reliability, and scalability. This level of detail enables efficiency optimization by selecting appropriate data serialization formats, compression techniques, and transport protocols to minimize overhead and latency. Additionally, LLD provides flexibility in protocol selection, allowing for the customization of protocols to specific use cases or the adoption of existing standards based on the architecture's requirements. By promoting modularity, LLD facilitates the creation of reusable communication components, enhancing code reuse and simplifying the implementation of common communication patterns across multiple services. Furthermore, LLD enables the design of communication mechanisms that can scale efficiently with the growth of the microservices ecosystem, incorporating strategies for load balancing, service discovery, and distributed tracing to maintain reliability and responsiveness under increasing loads. Additionally, LLD facilitates the integration of security measures, monitoring, and logging features into communication components, ensuring secure data exchange and providing real-time visibility into

communication patterns and performance metrics. Overall, LLD plays a pivotal role in designing efficient, scalable, and resilient communication mechanisms for microservices architectures, enabling seamless interaction between services in distributed systems.

Overall, low-level design is essential for turning abstract system requirements into detailed implementation plans that developers can follow to build the software system effectively. It provides a roadmap for writing code and guides developers through creating well-structured, maintainable, and scalable software solutions.

4.4.1 Request Flow

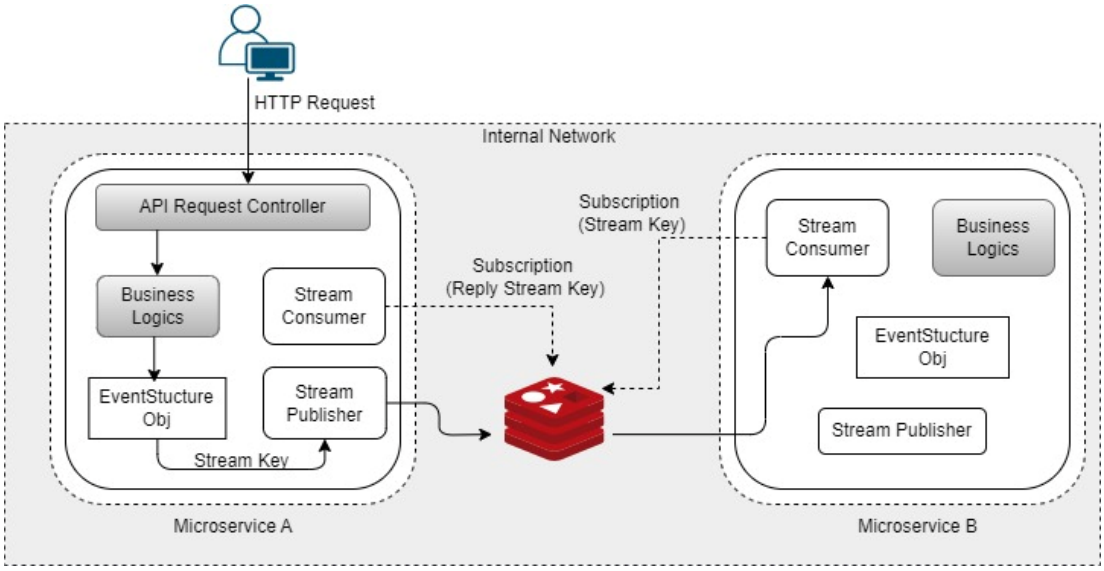

Fig. 4.3: Request flow

As per the above image 4.3, when a third-party client sends an HTTP request to microservice A. That client invokes an API exposed by microservice A using Spring Boot REST Controller. To expose an API using Spring Boot REST Controller, we have created a Java class annotated with @RestController or @Controller.

```
@Slf4j
@RestController
public class Controller {
```

Fig. 4.4: REST controller implementation

Within this class, we define methods to handle different HTTP requests, annotating each method with @RequestMapping, @GetMapping, @PostMapping, @PutMapping, or @DeleteMapping to specify the HTTP method and request mapping. We implement the logic to process the incoming HTTP requests in these methods. We extracted data from the requests using method parameters or annotations such as @RequestParam, @PathVariable, and @RequestBody as necessary way. This allows clients to access the exposed APIs through standard HTTP requests, making integrating Spring Boot applications with external systems or clients easy.

```
@PostMapping("/employees")
public ResponseEntity<?> employeesPost(@RequestBody String PayloadString)  {
```

Fig. 4.5: External API implementation

Upon starting, Microservice A establishes a Unix socket-based connection to a Redis server using the specified IP and port. It then creates a subscription based on the stream keys defined in the properties file. These stream keys correspond to the other microservices that Microservice A needs to communicate with by sending messages.

```
stream.key=order-events
stream.reply.key=order-events-reply
backend.timeout=1000
publisher.name=pub1
spring.redis.host=10.101.1.1
spring.redis.port=60002
```

Fig. 4.6: Additional configurations for proposed solution

```
Subscription subscription = listenerContainer.receiveAutoAck(Consumer.from(getReplyStreamKey(),
                InetAddress.getLocalHost().getHostName()),
        StreamOffset.create(getReplyStreamKey(), ReadOffset.lastConsumed()), eventConsumer);
listenerContainer.start();
```

Fig. 4.7: Initial stream subscription creation

When Microservice A receives a request from a third-party client, it begins processing and creates an EventStructure object that contains the request details and processed data. This object is then sent as a message to Microservice B, which is tasked with retrieving any additional required information. After completing the processing of the HTTP request, microservice A conducts a logical evaluation to determine the specific microservice required to provide an accurate response to the client. Based on this decision, Microservice A selects the appropriate stream key and publishes the message accordingly. During the publication process, the EventStructure object is serialized and transmitted as a byte buffer record. This approach significantly reduces network overhead and optimizes data transfer efficiency.

```
redisTemplate
        .opsForStream() ReactiveStreamOperations<String, Object, Object>
        .add(record) Mono<RecordId>
        .subscribe(e->{
        org.research.lk.HelperClass.getEventStructureHashMap().put(e.getValue(), eventObject);
        });
```

Fig. 4.8: Publish message to stream

A unique event ID is assigned to each published event. The event ID and the EventStructure object are recorded as a callback reference in a HashMap for future response processing. Serializing the EventStructure object and transmitting it as a byte buffer record offers several advantages. Firstly, it reduces network overhead by minimizing the amount of data sent over the network, which is particularly beneficial for transferring large or complex objects, thus reducing network latency and conserving bandwidth. Secondly, byte buffer records are more compact compared to other transmission formats like JSON or XML, resulting in faster data transfer speeds and improved overall performance of the communication layer. Additionally, byte buffer records are platform-agnostic, simplifying interoperability between systems built using different technologies. Moreover, serialized byte buffer records can be encrypted for enhanced data security and privacy during transit, reducing the risk of data breaches. Furthermore, they are simple to process and manipulate on both the sender and receiver sides, simplifying the implementation of communication protocols and reducing complexity. Byte buffer records also efficiently transmit binary data, such as multimedia content, without additional encoding. Lastly, they consume less memory compared to text-based formats, optimizing resource utilization and improving system performance. Overall, serializing the EventStructure object into a byte buffer record offers a lightweight, efficient, and versatile approach to data transmission, contributing to the optimization and effectiveness of the communication layer within the microservices architecture.

```
response = eventPublisher.getResponse(eventStructure, streamkey);
```

Fig. 4.9: How clients used the proposed solution to send the messages

In the proposed solution, sending messages to other microservices is as straightforward as using REST templates in Spring Boot. This streamlined approach simplifies communication between microservices, allowing for seamless interaction. Instead of complex configurations or custom protocols, we leverage the stream key as an endpoint for the target microservice. This means developers can easily communicate with other services by specifying the appropriate stream key. This simplification enhances developer productivity and promotes consistency and standardization across the microservices ecosystem. By abstracting away, the complexities of inter-service communication, our solution ensures that developers can focus on implementing business logic without being burdened by the intricacies of communication protocols or data serialization. Overall, this approach facilitates efficient and agile communication between microservices, fostering a more robust and scalable architecture.

4.4.2 Response Flow

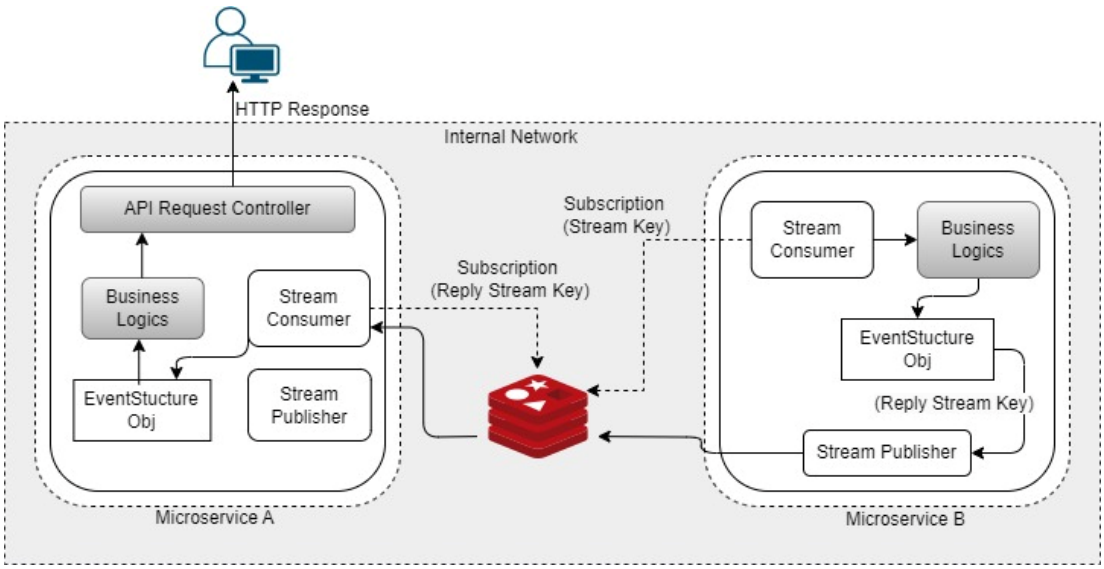

Fig. 4.10: Response flow

As elaborated previously, similar to microservice A, the other microservice (microservice B) also establishes a stream subscription upon startup, enabling it to listen for incoming stream messages over the TCP stream. Once a message is received by microservice B, it undergoes deserialization, transforming it back into the original EventStructure object. This object encapsulates all the request details, akin to attributes

found in a REST protocol. Developers can seamlessly access these attributes, leveraging them in much the same way as they would with a traditional RESTful API. This streamlined approach ensures developers can interact with the received messages effortlessly, utilizing familiar REST attributes for processing and responding to requests.

```
@Override
@SneakyThrows
public void onMessage(ObjectRecord<String, EventStructure> record) {
    //receiving and process the request
    BusinessProcess businessProcess = new BusinessProcess();
    businessProcess.processRequest(record.getValue());
```

Fig. 4.11: Consume message from stream

After deserialization, the message is processed by the corresponding microservice's business logic processors to generate the necessary response. Upon completing this processing, the business logic response is embedded within the EventStructure object. Critical details, including the publisher and reply stream key, are extracted from this object. Using this extracted information, the response is then published back to the appropriate stream via the designated stream key. Consumers within Microservice A then process the stream event to identify the client data by correlating the response with the entries in the HashMap that was previously saved for callback reference. Once the data is matched, the response API controller sends the response back to the client using the HTTP protocol. This systematic workflow ensures efficient message processing, accurate delivery of responses to the originating client, and robust, reliable communication between microservices.

```
//write the response code there
EventStructure eventStructure = new EventStructure();
eventStructure.setPayload(businessProcess.getResponse());
eventStructure.setResponseCode(HttpStatus.CREATED);
eventStructure.setResponseMappingId(record.getId().getValue());
respond(record, eventStructure);
```

Fig. 4.12: Structure of event object

Similarly to a REST API response, the implemented solution ensures that the response payload and corresponding HTTP status are set to provide a comprehensive view of

the response. However, in addition to adhering to REST principles, the solution includes the inclusion of a mapping ID to correlate requests, especially considering the asynchronous nature of communication. This addition enhances simplicity and facilitates easier tracking and managing requests and responses across the microservices architecture. By incorporating this mapping ID, developers can efficiently link requests with their corresponding responses, even in asynchronous scenarios, thereby ensuring seamless communication and effective handling of distributed system interactions. Overall, this approach adds a layer of robustness and reliability to the communication process, contributing to the overall efficiency and effectiveness of the implemented solution.

4.5 Achievement of Quality Attributes

Quality attributes play a crucial role in adopting and implementing a microservices architecture, with performance, scalability, traceability, availability and maintainability among the most vital considerations. The implemented research solution not only preserves but also enhances these quality attributes.

Scalability is a pivotal aspect of any robust system, and the implemented solution excels in this regard by offering both horizontal and vertical scalability. Leveraging cloud-based deployment, the solution enables vertical scaling by allowing for the augmentation of VM specifications. This entails provisioning additional resources within the JVM to accommodate increased computational demands. Furthermore, horizontal scaling is facilitated by the dynamic addition of microservice instances in response to evolving business requirements. When new microservice instances are introduced, subscriptions are established using stream keys, and message delivery is managed by the Redis server. This ensures that each request is directed at a single microservice instance, eliminating the risk of message duplication. Additionally, this approach simplifies service discovery and load balancing, thereby contributing to seamless scalability without compromising the precision of message delivery. Ultimately, the solution's adept handling of scalability ensures that the system can effortlessly accommodate fluctuations in workload and maintain optimal performance levels as demands evolve.

The performance quality attributes of the system are notably enhanced by leveraging TCP-based stream connections, eliminating the need to repeatedly open and close connections for every request. This optimization programmatically reduces processing time and eliminates network layer socket open time, significantly improving performance. Moreover, for service-to-service communication, all messages are transmitted in binary form and serialized, further minimizing network layer resource consumption. This streamlined approach enhances efficiency within the transport layer, reducing network latency and consequently enhancing microservice performance. By optimizing the communication protocol and minimizing overhead,

the system can efficiently transmit data between microservices, resulting in faster response times and improved overall system performance.

Maintainability is a crucial aspect of any software solution, and the proposed implementation excels in this regard. Developers can seamlessly integrate and maintain the system by offering internal communication capabilities without dependency on external HTTP clients. The elimination of reliance on internal load balancers for inter-service communication further streamlines maintenance efforts, reducing the system's complexity and minimizing potential points of failure. This simplification enhances the overall maintainability of the solution, allowing developers to focus on core functionalities and enhancements rather than managing intricate communication infrastructure. As a result, the implementation ensures that maintenance tasks are more straightforward and less prone to errors, ultimately contributing to the long-term sustainability and agility of the system.

Traceability is a critical aspect of system monitoring and troubleshooting, and the implemented solution excels in this regard. By leveraging the Redis server, end-to-end request/response tracing is supported, enabling developers to diagnose issues and provide technical support effectively. With the assistance of a Redis GUI client, developers can easily trace requests and responses by connecting to the Redis server. This comprehensive traceability enhances the ability to debug and analyze system behaviour, allowing for swift identification and resolution of any issues that may arise. Ultimately, the solution's robust traceability capabilities ensure that developers have the necessary tools and insights to maintain the system's reliability and performance over time.

Availability is paramount to ensure a responsive and reliable system, and the implemented solution prioritizes this aspect by leveraging Redis Streams for message delivery. By guaranteeing exact message delivery, the solution enhances overall system availability. This reliability in message transmission contributes significantly to ensuring that services remain accessible and responsive to user requests, even under heavy loads or adverse conditions. By relying on Redis Streams for message delivery, the system minimizes the risk of message loss or duplication, thereby bolstering its availability and resilience. This ensures critical business processes can continue uninterrupted, providing users with seamless and consistent experience. The solution's commitment to ensuring exact message delivery through Redis Streams underscores its dedication to maintaining high availability and responsiveness within the microservices architecture.

In summary, the proposed solution not only upholds the essential quality attributes of microservices architecture but also introduces enhancements that improve scalability, maintainability, traceability, and availability, thereby bolstering the overall effectiveness and robustness of the system.

4.6 Cloud Architecture

Cloud Reference Architecture serves as a blueprint or guideline for designing and implementing cloud-based solutions. It provides a structured approach to building scalable, reliable, and secure applications in the cloud. In contrast, specific architectures may vary depending on factors such as the cloud provider, industry, and application requirements.

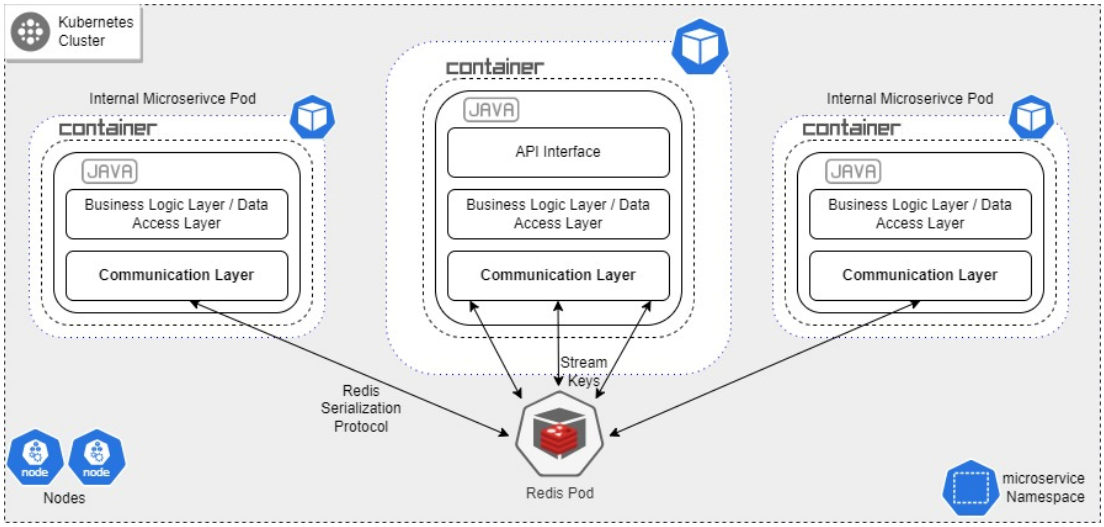

Fig. 4.13: Cloud architecture

All Redis and microservices servers are seamlessly deployed within the Kubernetes engine, a leading platform known for its robust container orchestration capabilities. Kubernetes engine was chosen after a thorough analysis revealed its widespread adoption and comprehensive feature set, making it the preferred choice among container orchestration services. As a pioneer in container orchestration, Kubernetes has continually enhanced its platform with valuable features, providing users with unparalleled management capabilities for their containerized applications. The autopilot cluster mode was adopted to optimize resource management and cost-effectiveness to create Kubernetes clusters in cloud vendors. This mode leverages advanced capabilities to manage cluster resources efficiently, ensuring optimal performance while minimizing operational overhead. Furthermore, the choice of Containerd as the container runtime for Kubernetes pods reflects the platform's commitment to staying current with the latest advancements in container technology. Docker runtime is now deprecated with the latest version of Kubernetes. To facilitate communication between microservices and Redis server pods, NodePort services were utilized. These services provide a reliable pathway for exposing TCP ports, enabling seamless communication within the Kubernetes cluster. NodePort services offer a straightforward means of external access, allowing services to be discovered using the port number and IP address of any node in the cluster. The deployment of all pods such as Kubernetes deployments underscores the platform's commitment to simplifying

application management and scaling. Kubernetes deployments provide a declarative approach to defining and managing application deployments, offering advanced capabilities for scaling, updating, and maintaining application instances. By utilizing deployments, organizations can ensure a consistent application state, facilitate seamless updates, and leverage self-healing mechanisms to maintain high availability and reliability.

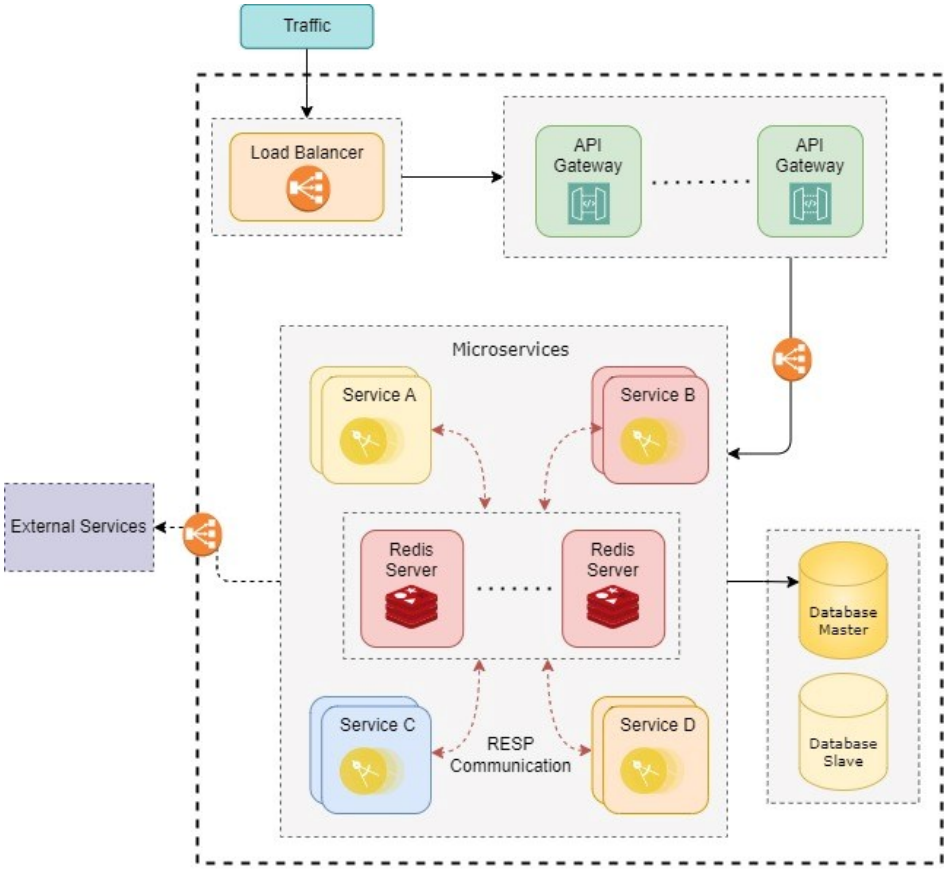

Fig. 4.14: High-level deployment architecture

A microservices reference architecture pattern with gateways typically involves the use of API gateways to facilitate communication between clients and microservices, as well as between microservices themselves. Almost all the software industries use API gateways for microservices, which is the most popular deployment pattern for microservice architecture. A microservices reference architecture pattern with an API gateway at its core provides a structured approach for designing and implementing distributed systems. The API gateway serves as a central entry point for clients to access microservices, handling authentication, authorization, and request routing. Microservices, the fundamental components of architecture, encapsulate specific business functionalities and communicate with each other through well-defined APIs. Serving as centralized entry points, they streamline client access to microservices by handling request routing and abstraction of underlying complexities. Additionally, API gateways facilitate the aggregation and composition of data from multiple microservices, optimizing performance by minimizing round trips for clients. Protocol

translation capabilities ensure seamless interoperability between clients and microservices, regardless of communication protocols. They also enable rate limiting and throttling to prevent resource abuse and ensure fair usage. API gateways provide monitoring, analytics, and versioning capabilities, enabling efficient lifecycle management of APIs, facilitating controlled updates and backward compatibility. In summary, API gateways play a pivotal role in enhancing the development, scalability, security, and management of microservices-based applications. According to the diagram, it's evident that all microservices are connected to the Redis server to facilitate communication among themselves. This architecture suggests the use of Redis as a centralized system that provides the exact delivery and exact once delivery for inter-service communication. Each microservice likely interacts with the Redis server to publish messages or data that need to be consumed by other microservices. By leveraging Redis as a communication medium, microservices can exchange information asynchronously, enabling decoupled and scalable communication patterns. This approach simplifies the implementation of distributed systems and promotes flexibility in the interaction between microservices. High availability of databases is commonly achieved through a master-slave replication setup. In this architecture, a primary database server (master) handles write operations and replicates data changes to one or more secondary database servers (slaves). The master maintains the authoritative copy of the database and serves as the source of truth for data modifications, while slaves replicate data asynchronously from the master. They serve read-only queries and can act as failover targets in case of master failure. Replication involves shipping binary log files or transaction logs from the master to the slaves, where each slave replays the changes on its own copy of the database. Failover mechanisms promote a slave to the new master in case of master failure, ensuring uninterrupted database access. Load balancing and read scaling can be achieved by distributing read traffic across multiple slave servers, enhancing system performance and responsiveness. Overall, the master-slave replication setup enables databases to maintain high availability, fault tolerance, and scalability, ensuring continuous access to critical data even during failures or increased workloads.

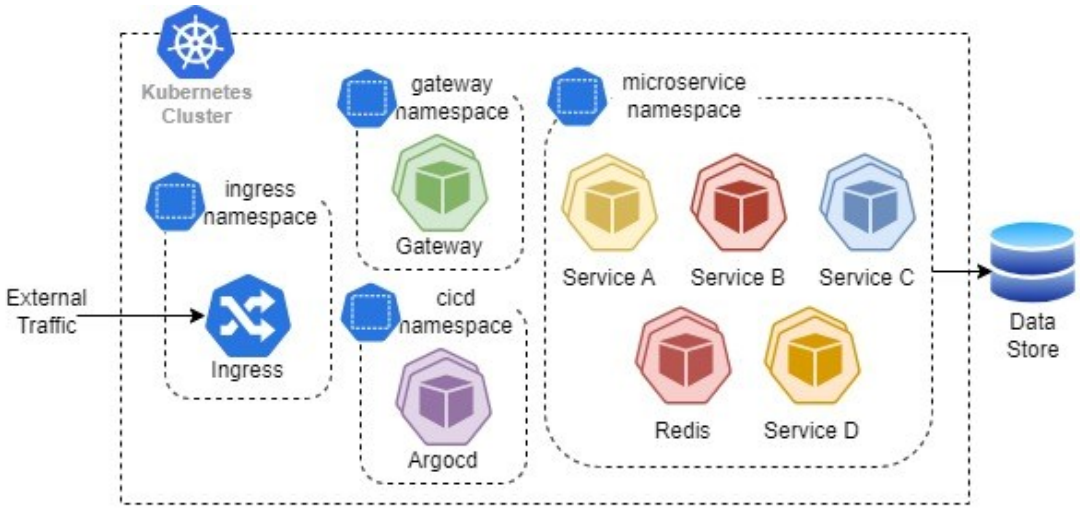

Fig. 4.15: High-level cloud deployment architecture

The above diagram 4.15 illustrates a cloud-native deployment of microservices with an API gateway. In this deployment, all microservices are hosted within the "microservice" namespace in Kubernetes, alongside the Redis servers. Horizontal Pod Autoscaler (HPA) is configured to dynamically scale both microservices and Redis instances based on CPU and memory utilization. This ensures optimal resource allocation and efficient utilization of resources, allowing the system to handle varying workloads effectively. By leveraging Kubernetes namespaces, the deployment provides isolation and organization for the microservices and Redis servers, facilitating easier management and maintenance. The API gateway serves as the entry point for external clients to access microservices, providing centralized routing, security, and monitoring capabilities. This cloud-native deployment architecture enables scalability, resilience, and efficient resource management for microservices-based applications in Kubernetes environments. Nginx Ingress Controller is employed to manage incoming traffic within the Kubernetes cluster. The Nginx Ingress Controller is placed within its dedicated namespace, typically named "ingress," to ensure separation and organization of resources. This controller acts as a gateway for external traffic entering the Kubernetes cluster, providing advanced routing, load balancing, and TLS termination capabilities. By configuring the Nginx Ingress Controller, incoming requests can be efficiently distributed to the appropriate microservices based on defined routing rules, enhancing scalability and resilience. This setup centralizes traffic management within the Kubernetes environment, simplifying configuration and maintenance tasks. Overall, the utilization of Nginx Ingress enhances the cloud-native deployment architecture by facilitating efficient traffic routing and management within the Kubernetes cluster. To enhance the deployment process and ensure seamless management of the system, the Continuous Integration and Continuous Deployment (CICD) concept is incorporated using ArgoCD with GitOps principles. ArgoCD is integrated into the Kubernetes environment to automate deployment workflows and promote infrastructure as code practices. With GitOps, the system's desired state is defined and maintained within a

Git repository, providing version control and auditability. ArgoCD continuously monitors the Git repository for changes and automatically applies them to the Kubernetes cluster, ensuring consistency between the desired and actual states of the system. This approach streamlines the deployment process, reduces manual intervention, and enhances maintainability by enforcing declarative configuration management. By leveraging ArgoCD with GitOps, the system benefits from increased reliability, traceability, and reproducibility of deployments, ultimately leading to improved operational efficiency and reduced risk of configuration drift.

4.7 Concluding Remarks

This chapter has detailed the proposed solution for optimizing inter-service communication in a microservices architecture. Starting with a high-level architecture, the chapter outlined the overall system design, followed by a more granular examination of the high-level component architecture, where the key components and their interactions were described. The low-level design section provided a closer look at the intricacies of the request and response flows, showcasing how the system handles inter service communication between microservices efficiently. The chapter also demonstrated how the proposed solution achieves essential software quality attributes ensuring that the architecture meets the demands of cloud-native environments. Furthermore, cloud architecture was discussed, highlighting how the proposed solution is designed to operate effectively within cloud infrastructures, leveraging the benefits of cloud computing to enhance the microservice architecture. In conclusion, this chapter presents a comprehensive blueprint for implementing an optimized inter-service communication strategy in microservices. The solution is designed to meet the challenges identified in the literature review and is grounded in the methodological framework outlined in the previous chapter. The next chapter will focus on the experimental results and their evaluation, providing an in-depth analysis of how the proposed solution performs under various conditions and validating its effectiveness in real-world scenarios.

CHAPTER 5

RESULTS AND EVALUATION

5.1 Introduction

In the preceding chapters, we delved into the behaviour of the research component within real-world environments, providing insights into its functionalities and how each section contributes to achieving the desired outcomes. It's imperative to thoroughly test and evaluate the entire system, considering both its non-functional and functional requirements. This comprehensive evaluation process aims to ensure the delivery of bug-free software as the final output. By subjecting the system to rigorous testing, we can establish benchmarks and assess its performance against predefined goals. This chapter focuses on the systematic testing and evaluation of the system in real-world environments to ascertain its effectiveness in meeting the expected objectives. We will delve into the methodologies employed for testing various components of the system, including inter-service communication, data processing, and overall system performance. Additionally, we will discuss the criteria used to evaluate the system's compliance with functional requirements, such as accuracy, reliability, and scalability, as well as non-functional requirements, such as responsiveness, security, and resource utilization. Through systematic testing and evaluation, we aim to validate the system's functionality, robustness, and performance under different scenarios and workloads. This process helps identify and rectify any deficiencies or vulnerabilities and provides valuable insights into optimizing system performance and achieving the desired outcomes. Ultimately, this evaluation's findings will inform the system's refinement to ensure it aligns with expectations and delivers the intended benefits to stakeholders.

5.2 Inter-service Communication Protocol Comparison

Here, we extensively explore the evaluation of inter-service communication protocols within a microservice architecture, emphasizing the critical role of selecting the appropriate communication mechanism to optimize response times across service calls. It underscores the significance of this evaluation in addressing performance challenges inherent in network-based inter-service communication within microservices. The study compares gRPC, HTTP, and Web Socket inter-service communication protocols and their effects on system performance. Because these are the protocols heavily used in the industry for various implementations. It highlights challenges within microservice architecture, which is known for breaking down large applications into decentralized components, especially regarding network-based inter-service communication performance. Here, we focus on two fundamental interaction

paradigms: synchronous and asynchronous. It explains how synchronous communication entails bidirectional request-response exchanges, whereas asynchronous communication relies on message-based protocols. The discourse further examines prevalent mechanisms for synchronous communication, including gRPC and REST APIs, alongside asynchronous communication facilitated by the Web Socket protocol. gRPC is a modern, high-performance RPC (Remote Procedure Call) framework developed by Google. It leverages HTTP/2 for transport and Protocol Buffers (protobuf) for efficient serialization. gRPC offers strong typing, bi-directional streaming, and built-in authentication and load-balancing support. When compared to implementation, it is very hard to implement, and maintainability is very complex. Furthermore, gRPC's strong typing and use of Protocol Buffers for message serialization may introduce challenges when dealing with evolving schemas or heterogeneous environments. Changes to the service contract or message formats may require careful coordination and may impact backward compatibility, potentially affecting scalability as the system evolves. HTTP is a widely used protocol for client-server communication on the web. It operates over TCP/IP and follows a request-response model. In contrast, HTTP offers simplicity and compatibility with existing web infrastructure. Most programming libraries and frameworks support the easy implementation of this model. WebSocket is a communication protocol that provides full-duplex communication channels over a single TCP connection. It allows for real-time, bidirectional communication between clients and servers, making it ideal for applications requiring continuous data exchange, such as chat applications and real-time collaboration tools. WebSocket's lightweight nature and support for event-driven architectures.

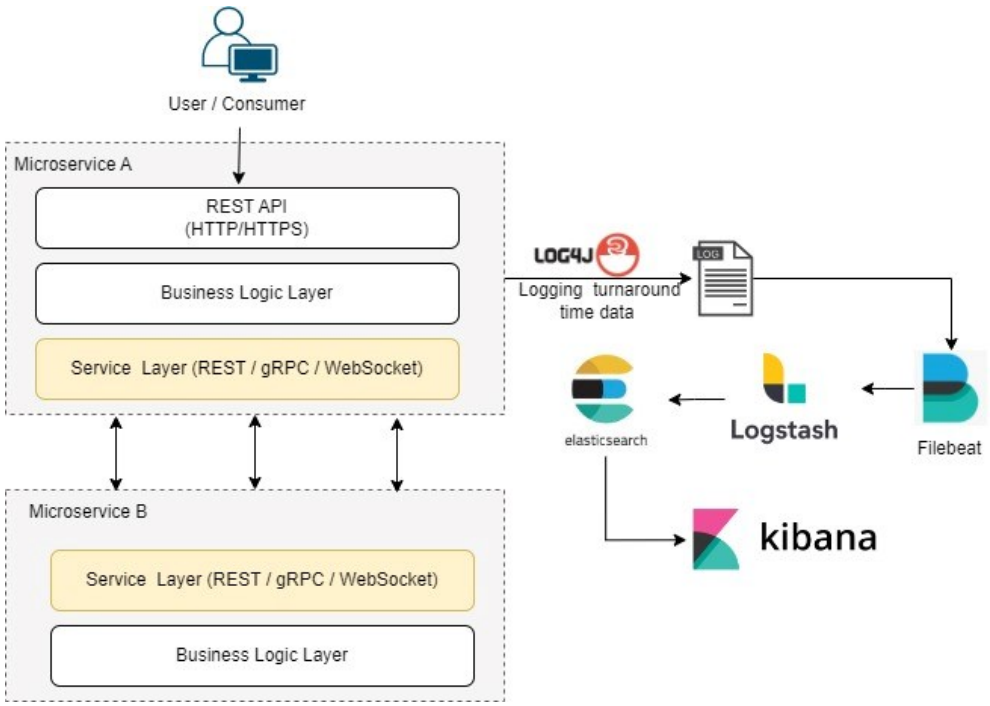

Fig. 5.1: Protocol comparison deployment architecture

For the test, we implemented the two microservices in Java using the Spring Boot framework. Many software development companies prefer Spring Boot due to its annotation-based programming, which boosts development efficiency and minimizes boilerplate code. Additionally, Spring Boot's support for embedded servers further streamlines the development process. In our implementation, one microservice exposes REST APIs to communicate with external services. Each microservice's business logic layer handles specific tasks. In Microservice A, it validates request/response payloads, calculates response times, and logs relevant information. Meanwhile, Microservice B's business logic layer generates appropriate responses based on calls from Microservice A. Response time data, which reflects the turnaround time between a request and the inter-service communication response, is recorded using the Log4j2 logging framework. This calculation only considers the time taken for communication and excludes all other processing time inside the JVM. One of the key performance advantages of Log4j2 is its asynchronous logging capability. Log4j2 uses asynchronous loggers by default, which means that logging operations are performed asynchronously in a separate thread. This asynchronous logging approach significantly reduces the overhead associated with logging in the main application thread. So, logging will not interrupt the API layer performance. The log files generated by Microservice A are read by the Filebeat agent and sent to Logstash for processing using predefined grok patterns. These patterns are designed to extract essential information such as log time and response time. The extracted details are stored in Elasticsearch and visualized through the Kibana dashboard.

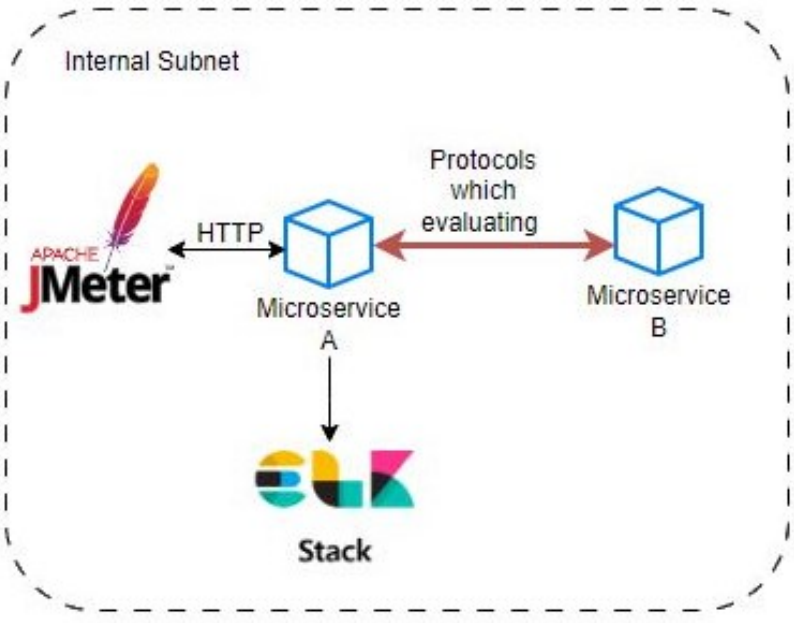

Fig. 5.2: Protocol comparison testing architecture

The developed microservices have been deployed on virtual machines (VMs) equipped with 2 CPUs and 4GB of RAM. VMs hosting the JMeter load testing tool and the ELK (Elasticsearch, Logstash, Kibana) stack for logging and monitoring purposes also adhere to the same specifications [148]. JMeter is utilized for load testing and performance monitoring purposes due to its comprehensive features and capabilities tailored for evaluating the performance of microservices [149]. JMeter enables the simulation of various load scenarios by generating a large number of concurrent requests for the microservices under test. JMeter provides detailed performance metrics and reports, allowing us to monitor key performance indicators such as response times, throughput, error rates, and resource utilization. This uniformity in VM configuration ensures resource allocation and performance consistency across the deployment environment. Furthermore, all VMs are deployed within a single network subnet, facilitating reduced network latencies for inter-service communication. Placing the VMs in the same subnet minimizes the distance data packets need to travel between components, thereby optimizing network performance and enhancing overall system responsiveness. This deployment architecture is designed to provide an efficient and scalable environment for hosting and managing the microservices, load-testing tools, and logging infrastructure. By leveraging VMs with standardized configurations and optimizing network connectivity, the deployment aims to ensure optimal resource utilization and performance across the system. Using these setups we have executed the two different test scenarios in order to measure the performance of the protocols,

1. Controlled adjustments to transactions per second (TPS) and payload sizes to gauge the time taken for service-to-service communication. This approach aimed to systematically evaluate the impact of varying TPS and payload sizes on inter-service communication within the microservices architecture.
2. Another investigation aspect involved exclusively manipulating payload sizes while measuring the consequent effects on service-to-service communication and check on overall application performance, specifically throughput and response time.

5.2.1 Controlled Throughput & Payload Size

We adopted various test scenarios inspired by prior research studies to evaluate the performance of our system. These scenarios involved manipulating payload sizes and controlling transactions per second (TPS) while conducting tests using the JMeter tool. Specifically, we tested the following payload sizes:

- For HTTP GET requests, the backend microservice returned an empty response with a 200 HTTP status code.
- For HTTP POST requests carrying a 1KB payload, the backend microservice responded with a 1KB JSON payload and a 200 HTTP status code.

- For HTTP POST requests with a 5KB payload, the backend microservice returned a 5KB JSON payload along with a 200 HTTP status code.

Each of these payload sizes was tested under controlled throughputs of 10TPS and 100TPS to assess their impact on the system's performance.

TABLE 5.1: Protocol comparison testing scenarios.

HTTP			**gRPC**			**Websockets**		
URL Only	**1 KB payload**	**5 KB payload**	**URL Only**	**1 KB payload**	**5 KB payload**	**URL Only**	**1 KB payload**	**5 KB payload**
10 TPS throughput								
100 TPS throughput								

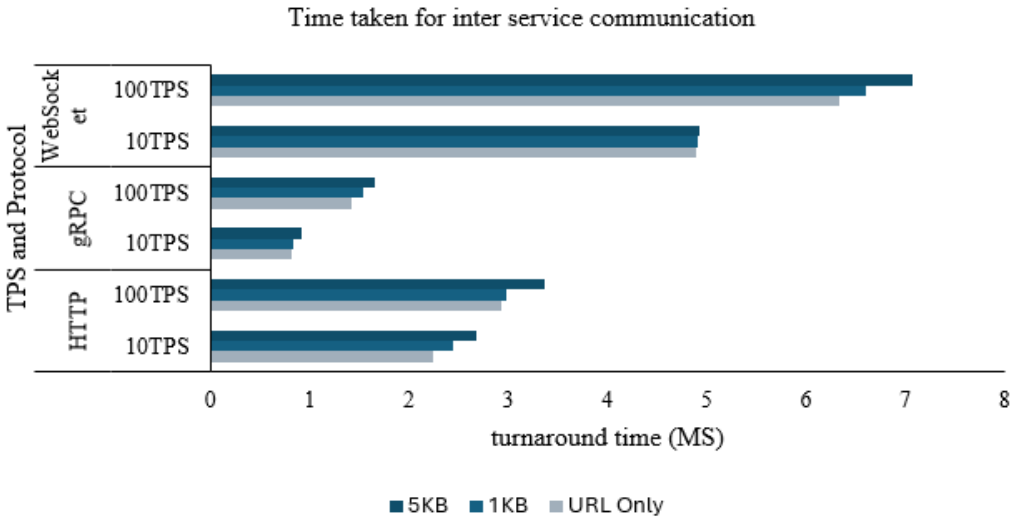

Fig. 5.3: Protocol-wise turnaround time variation

The illustration above depicts a direct correlation between increasing payload size and heightened latency across all communication protocols. This latency surge is primarily attributable to the transmission of larger network packets at the network layer, necessitating a greater allocation of network bandwidth to facilitate packet transfers effectively. Upon thoroughly analysing the turnaround time associated with each communication protocol, it becomes evident that gRPC exhibits superior latency performance when communicating with distributed services. Conversely, the WebSocket protocol demonstrates significantly higher latency levels, particularly when utilized for microservice-to-microservice communication in a request/response manner. This discrepancy underscores the inherent mismatch between the WebSocket

protocol and the specific requirements of inter-service communication within distributed environments. While the WebSocket protocol is known for its efficiency in web browser-level communication, it proves to be less compatible with facilitating inter-service communication in distributed environments. In contrast, gRPC, functioning as a remote procedure call protocol, leverages protocol buffers to streamline data transfer between servers over a single TCP connection. This optimized approach effectively minimizes network overhead compared to the conventional HTTP protocol, wherein connections must be established and terminated for each request without the benefit of packet compression. Consequently, gRPC emerges as a more resource-efficient solution for conserving network layer resources, particularly when compared to HTTP. In addition, it's worth noting that while gRPC boasts lower latency in communicating with distributed services, its implementation complexity and scalability issues render it less suitable for microservice architecture.

5.2.2 Controlled Requests & Payload Size

Testing was conducted to evaluate how latency in the inter-service communication protocol affects overall application performance, with a focus on response time and throughput. This evaluation was performed in a consistent above environment. The testing utilized a standardized JSON payload size of 1KB. To gauge application throughput and overall latency accurately, JMeter scripts were employed and augmented with listeners for measurement. Meantime measurements extracted from the integrated ELK stack were instrumental in determining the average turnaround time for service to service communication across each protocol.

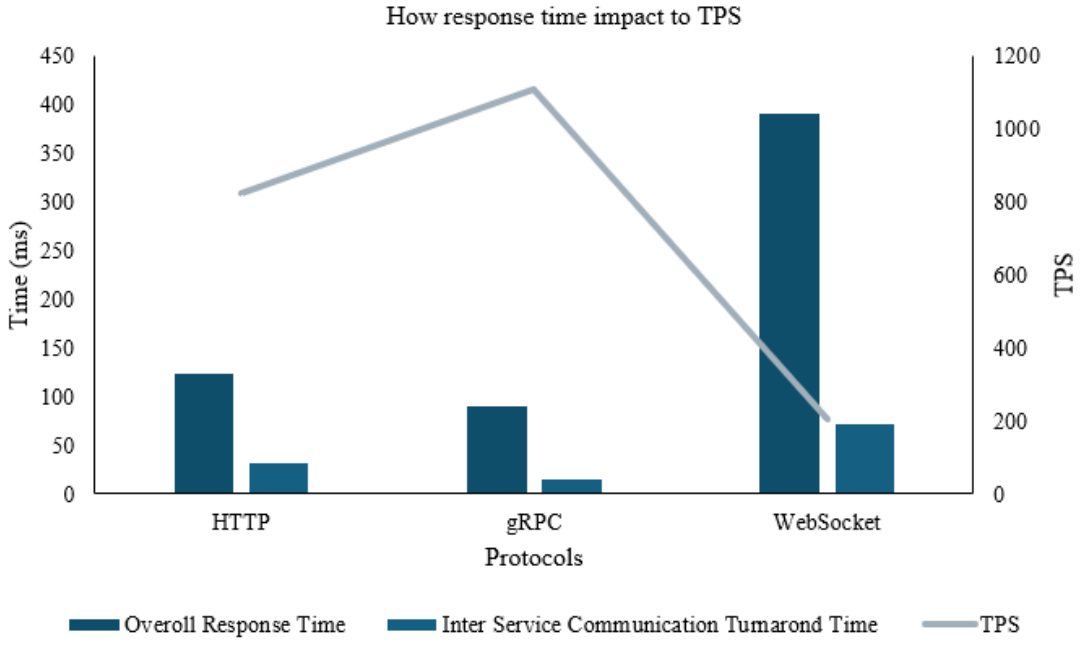

Fig. 5.4: Impact on response time for TPS in various protocols

The graph portrayed above provides a detailed representation, showcasing three axes: the X-axis denotes the protocols utilized, the Y-axis illustrates the time in milliseconds, and the Z-axis represents transactions per second (TPS). The graph suggests a clear relationship between the low latency of inter-service communication and the overall improvement in response time. From that, we can observe that if there is a low latency for the inter-service communication, then it will also improve the overall application response time as well. Furthermore, the graph highlights a notable increase in throughput for the gRPC protocol. This observation underscores the substantial impact of efficient inter-service communication on the overall throughput of applications deployed in distributed environments.

According to the protocol evaluation that we have conducted, we can see that there is a huge impact on the application performance in terms of overall response time and the throughput from the inter-service communication latencies. In the experimentation results, we can see that when inter-service communication is getting high latency overall, the application also gets a considerably high latency to produce the output to the user. We have also observed that when high latency occurs, overall application throughput gets degraded.

$$\text{inter-service communication turnaround time} \propto \text{application response time}$$

$$\text{payload size} \propto 1/\text{throughput}$$

The analysis of the collected data showed that while the gRPC protocol facilitated efficient inter-service communication and enhanced application performance in terms of throughput and response time, its implementation complexity and scalability challenges make it less suitable for dynamic software architectures such as microservices. Research conducted by Najem Hamo and Simon Saberian on the usability of gRPC and HTTP protocols further supports this finding, demonstrating that gRPC presents challenges in usability. These findings align with the results of above study, reinforcing the conclusion that gRPC, despite its performance benefits, may not be the optimal choice for microservices-based architectures [150]. Microservices are developed with shorter development cycles and must accommodate frequent changes according to environmental and business changes. Additionally, microservices require scalability and maintainability as key quality attributes. Lukasz Kaminski and his team demonstrated that as payload size increases, gRPC experiences performance degradation, particularly in terms of inter-service communication latency [151]. Considering these factors, the adoption of the gRPC protocol for microservice architecture may not be feasible. Conversely, WebSocket communication, although effective for communication between user interfaces and services, was found to be unsuitable for inter-service communication within microservices. After considering

the factors mentioned earlier, we have decided to proceed with the HTTP protocol for inter-service communication. Despite its average performance, the HTTP protocol offers ease of implementation and widespread support across programming languages and frameworks. Many enterprise-grade software solutions utilize HTTP for inter-service communication, benefiting from its built-in libraries and compatibility with various environments.

5.3 Proposed Solution Evaluation

A comprehensive analysis of the microservice architecture has demonstrated that inter-service communication is crucial for performance outcomes, significantly impacting overall application response time and throughput. This is particularly notable due to the distributed nature of microservices, which require network calls to generate reliable business requirements. To address this issue, a solution was devised and implemented in this research to enhance inter-service communication, thereby improving overall application performance in terms of response time and throughput.

Considering the above findings, we are evaluating the research component using the HTTP protocol for inter-service communication. While gRPC demonstrated efficiency in throughput and response time, its implementation complexity and scalability challenges make it less suitable for microservices, which require agility, maintainability, and seamless scalability. Similarly, WebSocket communication, though effective for UI-service interactions, was found unsuitable for inter-service communication within microservices. Furthermore, our literature review revealed that most enterprise-grade microservice architectures and academic research studies have primarily used the HTTP protocol for their evaluations. This widespread adoption is due to HTTP's ease of implementation, extensive support across programming languages and frameworks, and its established role in microservices communication. Given these considerations, HTTP has been selected as the protocol for evaluating our research component.

The implemented solution and HTTP inter-service communication were deployed in a virtual machine environment to assess and evaluate the application's response time and throughput.

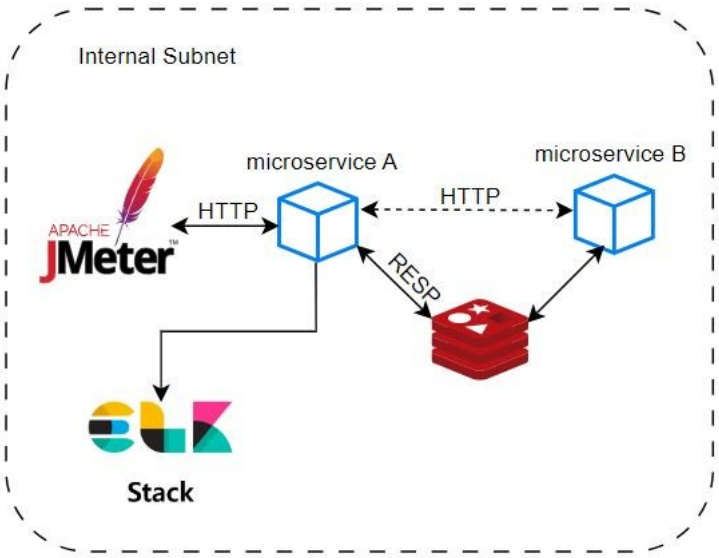

Fig. 5.5: Proposed solution testing architecture

The above high-level architecture shows the testing architecture that we have used to evaluate the proposed solution and the traditional HTTP-based inter-service communication. As with the previous setup, we have deployed the above architect solution in the VMs in a single subnet to reduce network latencies. The developed microservices have been deployed on VMs equipped with 2 CPUs and 4GB of RAM. VMs hosting the JMeter load testing tool and the ELK (Elasticsearch, Logstash, Kibana) stack for logging and monitoring purposes also adhere to the same specifications. However, for the Redis, we have used a lower specification VM, which has 1 CPU and 2GB RAM.

The testing process comprised two distinct methodologies:

- Scenario A: This approach involved controlling the application's overall throughput and the size of request/response payloads. Subsequently, the inter-service communication turnaround time was measured under these controlled conditions.
- Scenario B: In this scenario, only the size of payloads was controlled. The testing was concentrated on measuring the service-to-service communication turnaround time, the overall microservices response time, and the transactions per second.

5.3.1 Proposed Solution Testing Scenario A

This scope encompassed different throughput values and payload sizes. Specifically, the tests involved invoking HTTP GET and POST methods from the microservice, with subsequent responses from the backend microservice. For instance, the HTTP GET method elicited a 200 OK HTTP response code with an empty body, while the HTTP POST method involved a 1KB size JSON payload and a corresponding 1KB JSON format response from the backend microservice. In these test scenarios, comprehensive evaluations were performed on the HTTP communication method, the turnaround time for inter-service communication, and the performance of the newly implemented solution in facilitating inter-service communication. Each scenario was executed for one hour and repeated three times to determine the average service to service communication metrics.

TABLE 5.2: Proposed Solution Test cases

Number	Transport mode	Test Case Scenario
1	URL only	Throughput was regulated to 10 TPS, and HTTP GET requests were sent.
2	1KB payload	Throughput was regulated to 10 TPS, and HTTP POST requests were sent.
3	URL only	Throughput was regulated to 100 TPS, and HTTP GET requests were sent
4	1KB payload	Throughput was regulated to 100 TPS, and HTTP POST requests were sent

The above table provides a summary of the test we conducted to evaluate the proposed solution and traditional HTTP-based communication. In the spring boot framework, we have used the REST templates to establish this communication, which is the most popular mechanism in the industry for inter-service communications. Each test scenario experienced a duration of 1 hour and was replicated three times to derive an average value for inter-service communication.

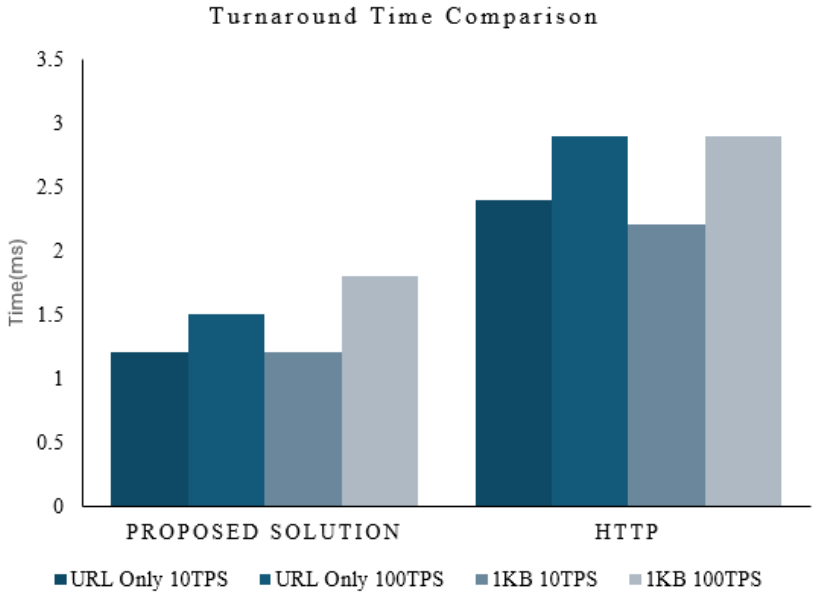

Fig. 5.6: Proposed solution turnaround time comparison

The diagram above highlights the difference in turnaround time between the proposed solution and the HTTP protocol implementations. As the throughput increases, both solutions experience longer turnaround times. While they exhibit similar behaviour concerning payload size, the proposed solution consistently demonstrates significantly lower turnaround times across all test scenarios compared to HTTP. The HTTP protocol should need to create a socket connection for each transaction and close it upon receiving a response, leading to increased latency. Moreover, network packets in HTTP are neither binary nor fully serialized, resulting in substantial transfer time. In contrast, the new implementation establishes a TCP socket-based connection with the Redis server, allowing microservices to function as part of a specific microservice upon initialization. When transmitting data to other microservices, serialization is employed, reducing network resource consumption. Consequently, the implemented solution achieves shorter turnaround times compared to the standard HTTP communication method.

5.3.2 Proposed Solution Testing Scenario B

In this specific scenario, the evaluation focuses solely on controlling and assessing payload sizes to gauge the application's overall response time and the turnaround time for inter-service communication within the Microservice architecture. This testing protocol was executed within the same cloud-based environment. The selected payload sizes included 1KB, 5KB JSON payloads and URL only get request/response. JMeter listeners were integrated into the testing process to measure the overall application response time. Inter-service communication turnaround times were measured using log analysis. Each test scenario ran for one hour and was repeated three times to

calculate average values for service-to-service communication, overall microservices response, and software throughput.

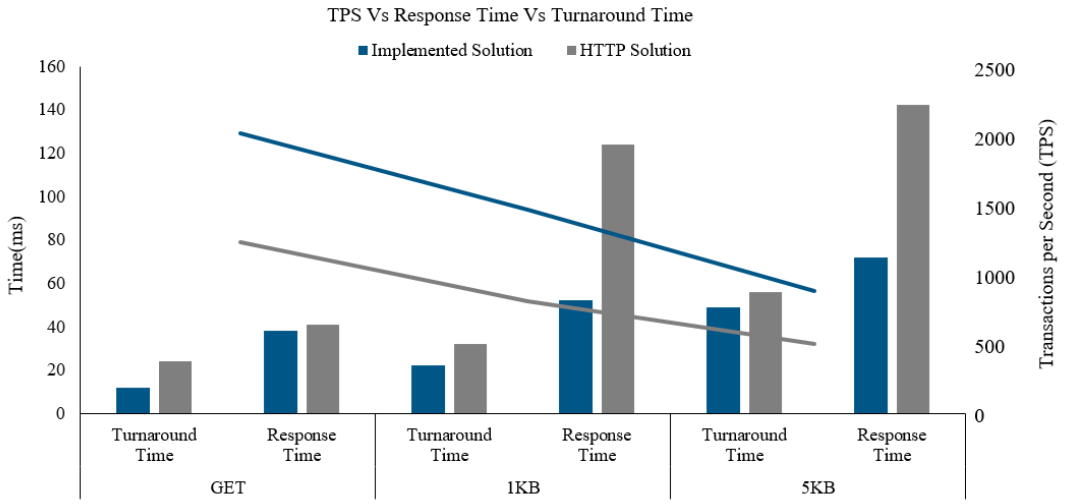

Fig. 5.7: Proposed solution TPS, response time and turnaround time comparison

Analyzing the above diagram reveals a consistent trend when the payload size increases, the throughput decreases. This behaviour aligns with typical network performance patterns, where heavier packets reduce overall network efficiency. Consequently, as the payload size grows, the response time also increases. Comparing the implemented solution with the conventional HTTP inter-service communication method across various scenarios, it becomes evident that the response time consistently improves with the implemented solution.

The turnaround time of inter-service communication significantly impacts the overall application response time. In essence, if communication between services takes longer, it inevitably results in higher overall response times for the system. Moreover, the size of request and response payloads directly influences system throughput. The transfer of large network packets consumes considerable time, thereby increasing response times. Consequently, this negatively affects the overall system throughput. You can see that when compared to the HTTP method implemented solution system's thought is high and gives a better performance to the end users.

Based on our conducted testing, we have statically proven that our proposed solution performs well compared to the traditional HTTP communication protocol for service-to-service communication. Our solution achieves this by reducing network resource consumption through the conversion of transport medium to binary format and serialization. Additionally, there is no additional time or computational power required to open and close socket connections, as we utilize a TCP-based stream connection to send packets. These optimizations enable us to reduce latencies between service-to-service communication. Unlike gRPC, users can seamlessly utilize our library, which is similar to HTTP REST templates. All other logic is handled within the library, and

users only need to configure a simple configuration file. Consequently, this ensures faster development and deployment times while increasing microservice maintainability.

5.4 Proposed Solution Cloud Native Suitability

This evaluation centers on inter-service communication within the microservice architecture of a cloud-native environment. A solution was specifically designed and implemented to operate within Kubernetes, a cloud-native platform, with the objective of improving performance in terms of response time and application throughput. This testing rigorously compares the implemented solution with traditional inter-service communication methods in a cloud-native context. Both the conventional HTTP method and the newly developed solution were deployed in a Google Kubernetes environment, where various test cases were conducted to collect and analyze relevant data. We opted for the Google Cloud environment over other cloud vendors for several compelling reasons. Firstly, Google Cloud Platform (GCP) offers a robust and reliable infrastructure, with a global network of data centers ensuring high availability and low latency. Additionally, GCP provides extensive support for Kubernetes, making it an ideal choice for deploying and managing containerized applications in a cloud-native environment. GKE is tightly integrated with Google Cloud Platform (GCP), providing seamless integration with other GCP services such as storage, networking, monitoring, and logging. GKE allows automatic scaling of application instances based on demand, ensuring optimal resource utilization and cost-efficiency using the auto polit mode. GKE automatically manages the availability of applications by distributing them across multiple zones within a Google Cloud region, minimizing downtime and ensuring reliability. Overall, Google Kubernetes Engine offers a powerful and feature-rich platform for running Kubernetes workloads in the cloud, with native integration, advanced networking, auto-scaling, security, and multi-cloud support setting it apart from other Kubernetes engines. Moreover, Google's commitment to innovation and cutting-edge technologies aligns well with our research goals, allowing us to leverage advanced features and services to enhance our experimentation and analysis. Lastly, Google Cloud offers competitive pricing and flexible pricing models, enabling us to optimize costs while benefiting from a scalable and performant cloud infrastructure. Overall, these factors collectively make Google Cloud Platform a preferred choice for our research endeavors.

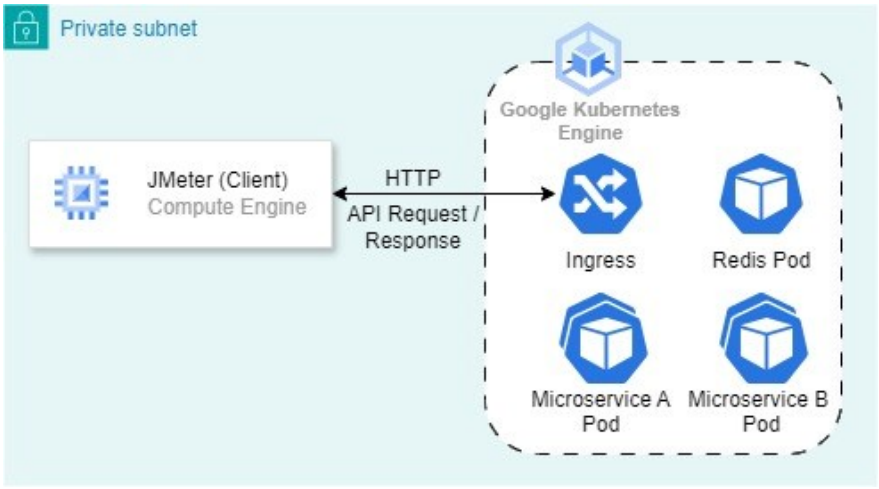

Fig. 5.8: Cloud-native testing architecture

The figure above demonstrates the setup of components within the Google Cloud environment for testing. Traffic was generated using Apache JMeter, which served as an external client or application, similar to previous evaluation processes. A Google Kubernetes Engine Autopilot Kubernetes cluster was set up to manage the application deployment. The Nginx ingress controller was utilized to allow external access to services within the Kubernetes cluster from JMeter. To reduce network latency, both the JMeter client and the Kubernetes cluster were placed in the same subnet. JMeter was deployed on an n1-standard-4 VM, which has 4 virtual CPUs and 15GB of memory. The system was evaluated through a series of test cases designed to assess the effectiveness of the proposed solution in a cloud-native environment. These tests also focused on evaluating scalability in both horizontal and vertical dimensions, with an emphasis on key software quality attributes.

TABLE 5.3: Cloud-native test cases

#	TPS	Payload Size	Description
1	100TPS	URL only request and 1KB payload	Evaluated the turnaround time
2	200TPS	1KB payload	Allocated CPU for microservices pods was adjusted from 1 core to 4 cores to assess its impact on inter service communication time and microservices response time
3	200TPS	1KB payload	Allocated memory for microservices pods increased from 1 GB to 4 GB, and its effects on inter service communication time and microservices response time were evaluated
4	200TPS	1KB payload	The number of microservices pods was increased from 1 to 4, and the impact on inter service communication time and microservices response time was measured

The table above outlines the test cases employed in this analysis to assess the effectiveness of the solution in a cloud-native environment. Each test scenario focused on measuring turnaround time from logs, specifically the duration of request and response between Microservice A and Microservice B, without including processing time. The tests were conducted over one hour and repeated three times to capture data variations. Logs were collected through the Google Logging service, and Python scripts were subsequently used for data processing and analysis.

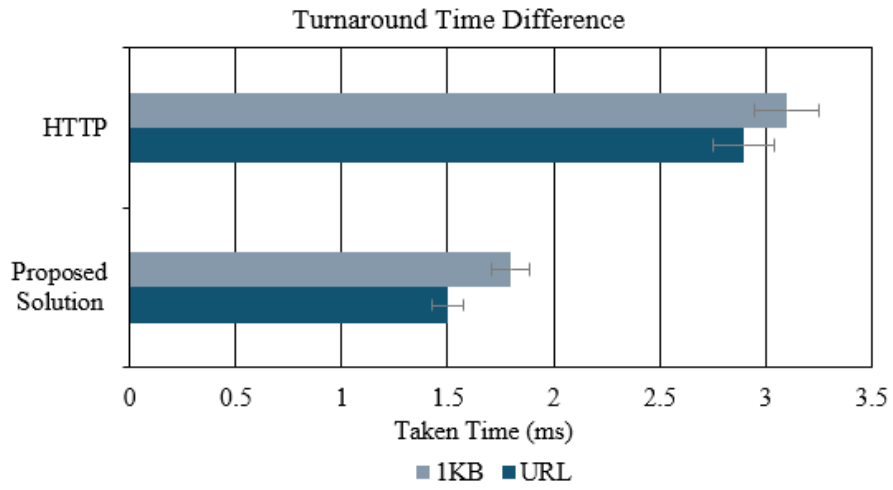

Fig. 5.9: HTTP vs proposed solution turnaround time comparison in cloud environment

The graph presented compares the turnaround time between the traditional HTTP method and the newly implemented solution methodologies within the cloud-native container orchestration environment. In this setup, Spring Boot Rest Templates continue to facilitate inter-service communication between the microservices. The analysis of the graph demonstrates a significant reduction in turnaround time with the implemented solution compared to the traditional HTTP approach, indicating that Microservice A receives responses from Microservice B more promptly. Unlike the HTTP method, the implemented solution does not involve open and close socket operations. Instead, data transmission occurs in a serialized manner, minimizing network consumption in the cloud-native environment as well. Consequently, the implemented solution exhibits more extraordinary performance over the traditional method. However, it's essential to note that the test only considers communication between one microservice and another. In real-world scenarios, multiple microservices need to interact to fulfil business requirements, akin to systems like Netflix. Although the improvement in turnaround time for individual microservice calls may seem slight, the cumulative effect significantly enhances the overall response time. This

observation aligns with the butterfly effect theory, elucidating the profound impact seemingly minor optimizations can have in complex systems.

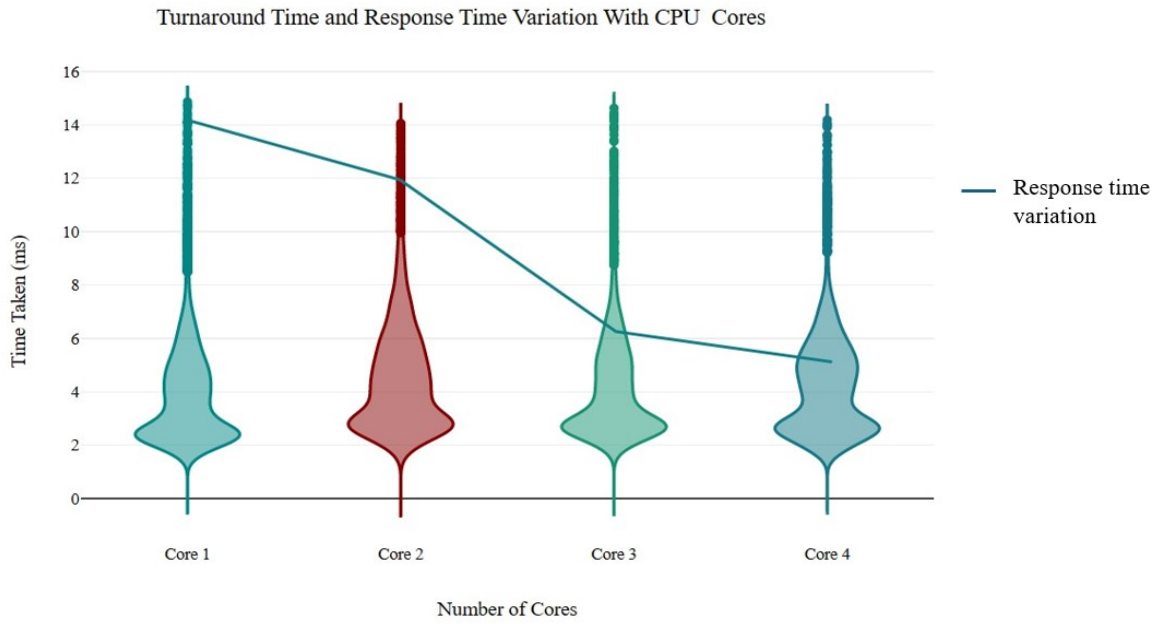

Fig. 5.10: Variation of response time and turnaround time with CPU

The violin and line plots displayed above illustrate the turnaround time and average response time at various CPU core levels for microservices pods. In this test scenario, CPU cores were added to the microservices pods by modifying the Kubernetes deployment YAML files and applying these changes within the Kubernetes cluster. The violin plot offers a detailed view of the distribution of turnaround time across different CPU core allocations. Analysis reveals no significant change in the inter-service communication turnaround time with the addition of CPU cores at the microservices pod level. Conversely, the line plot shows a deterioration in the overall application response time as the number of CPU cores allocated to the pods increases. This observation suggests that the computational demands of handling business logic may vary, resulting in fluctuations in response time. In conclusion, while vertically scaling the CPU cores of pods does not impact inter-service communication, enhancing overall application performance may necessitate allocating appropriate CPU resources to microservices.

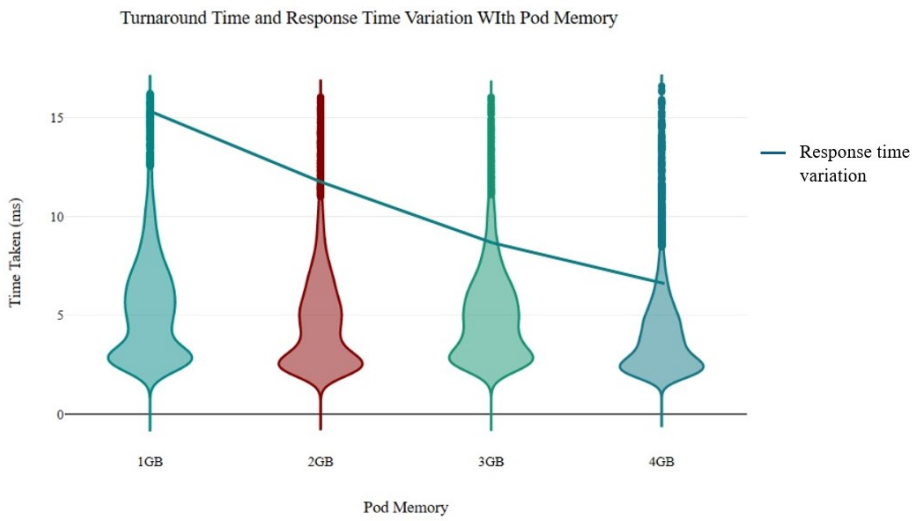

Fig. 5.11: Variation of response time and turnaround time with memory

The graph illustrates the relationship between the turnaround time of microservices and the overall application response time variation with memory allocation for microservice pods. The violin plot shows that the high-frequency distribution of turnaround time remains consistent despite increases in pod memory allocation, suggesting that changes in pod memory do not significantly affect microservice turnaround time. In contrast, the line plot indicates that increasing memory allocation for the microservice pods leads to improved overall application response time. This improvement is attributed to the more efficient processing of logic within the pods, which enhances the response time for the entire application. Therefore, optimizing memory allocation contributes to the improvement of the overall application performance.

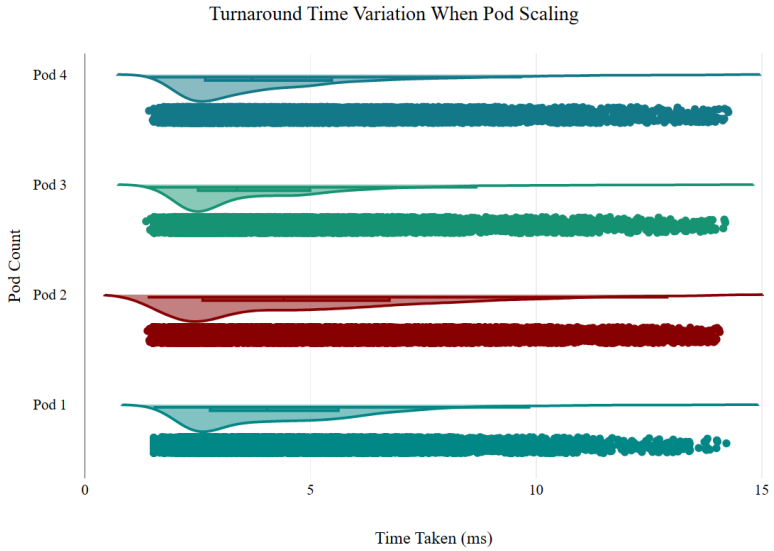

Fig. 5.12: Turnaround time variation with scaling

The Raincloud Plot provides insights into the distribution of turnaround times as microservices pods are scaled. The analysis reveals that prior vertical scaling efforts do not substantially affect the inter-service communication time between microservices, though they do influence the overall system response time. These test scenarios are intended to assess the effectiveness of horizontal scaling mechanisms and analyze turnaround times in service-to-service communication among two microservices. The collected data indicates that adding new pods to the cluster does not result in significant deviations in turnaround time, suggesting that horizontal scaling has minimal impact on inter-service communication.

Based on our evaluation, the proposed solution does not pose any obstacle to microservices' horizontal and vertical scaling. It is well suited to address the most concerning attributes of scalability quality. Our statistical analysis indicates that horizontal and vertical scaling of microservices does not significantly impact service-to-service communication. However, it does enhance overall application response time by empowering the JVM to process requests with greater computational power. Furthermore, our solution performs admirably in cloud-native environments compared to traditional HTTP protocols. Its efficiency in handling inter-service communication contributes to improved system performance and reliability. Thus, our proposed solution offers scalability, performance, and compatibility advantages in cloud-native settings.

5.5 Evaluating the Number of Segments for Inter-service Communication

Interservice communication is the heart of microservices architecture, and previous testing has also proven that statistically. The importance of microservice communication lies in several vital aspects underpinning this architectural model's success and efficiency. Functionalities that break down into microservices and set up clear communication segments ensure that each microservice can effectively help meet specific user needs. However, system designers must comprehensively understand the inter-service communication pathways involved to meet user requirements. In this test, the impact of inter-service communication segments on the overall performance of microservices is evaluated.

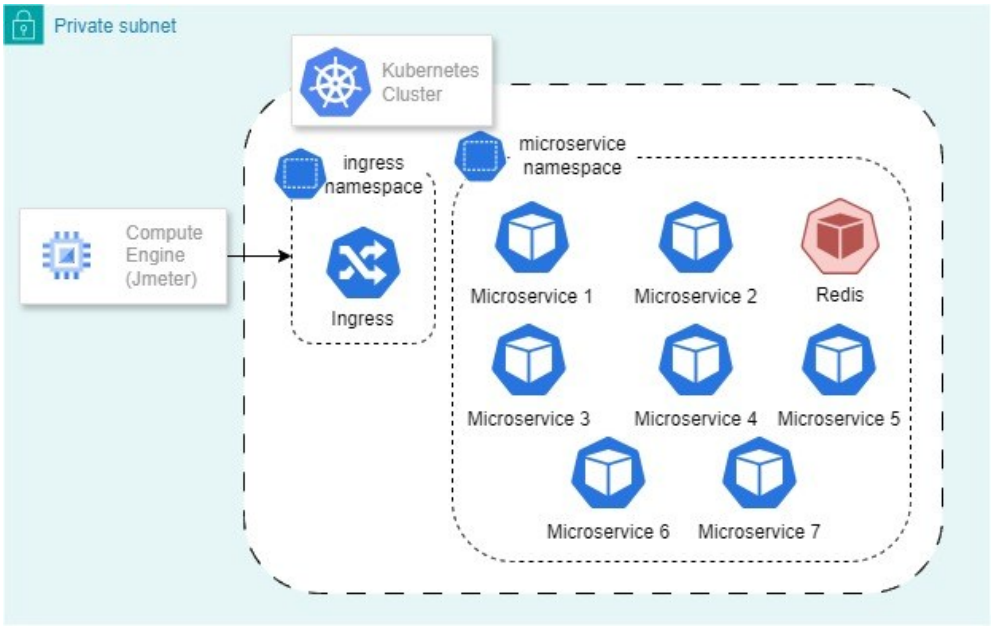

Fig. 5.13: Deployment test architecture for evaluating the segments of communication paths

The diagram depicted above illustrates the microservice architecture deployed for testing purposes in a cloud environment. Microservices are developed using Java language, leveraging the popular microservice framework Spring Boot. The proposed method has been used for microservice inter-service communication. Additionally, JMeter is installed on a separate virtual machine within the cloud, residing in the same network subnet as the Kubernetes cluster to minimize network latencies. The methodology advocates for the best and most minimal communication approach for microservice interactions, as shown in the above image. Apache JMeter is utilized to generate REST-based API traffic, allowing for control over parameters such as concurrent users and throughput. One of the microservices interfaces with JMeter, accepting HTTP-based REST API traffic generated by approximately 50 concurrent users operating at a controlled rate of 150 TPS.

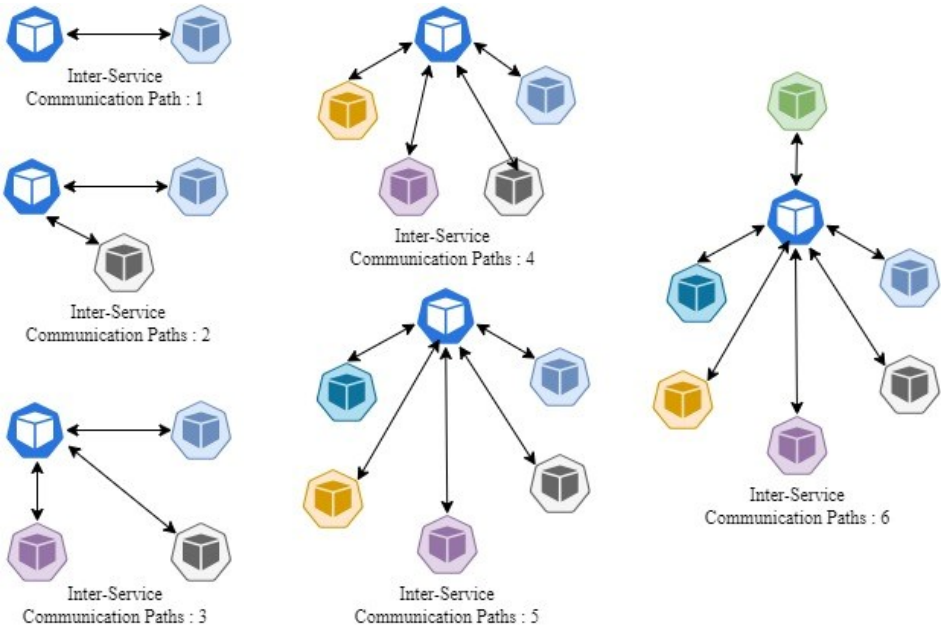

Fig. 5.14: Inter service communication paths

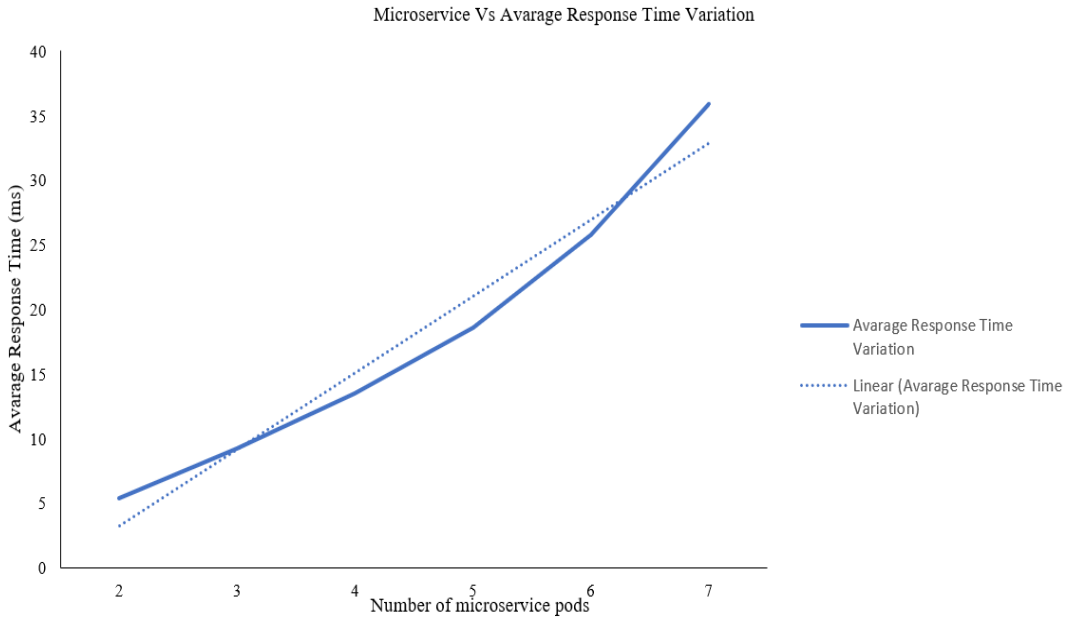

Fig. 5.15: Response time variation with number of microservices

The diagram presented in above image illustrates the relationship between the overall system's average response time and the number of microservices involved in fulfilling a single-user functional requirement. The data presented in the graph is derived from JMeter statistical reports, and each test was conducted three times to minimize data errors. According to the graph, the addition of multiple microservices results in an increase in the overall system's response time. However, with up to six microservices, the response time remains below the linear average response time. Beyond the six

microservices, the response time surpasses the average line, leading to an exponential increase in the overall software system's response time.

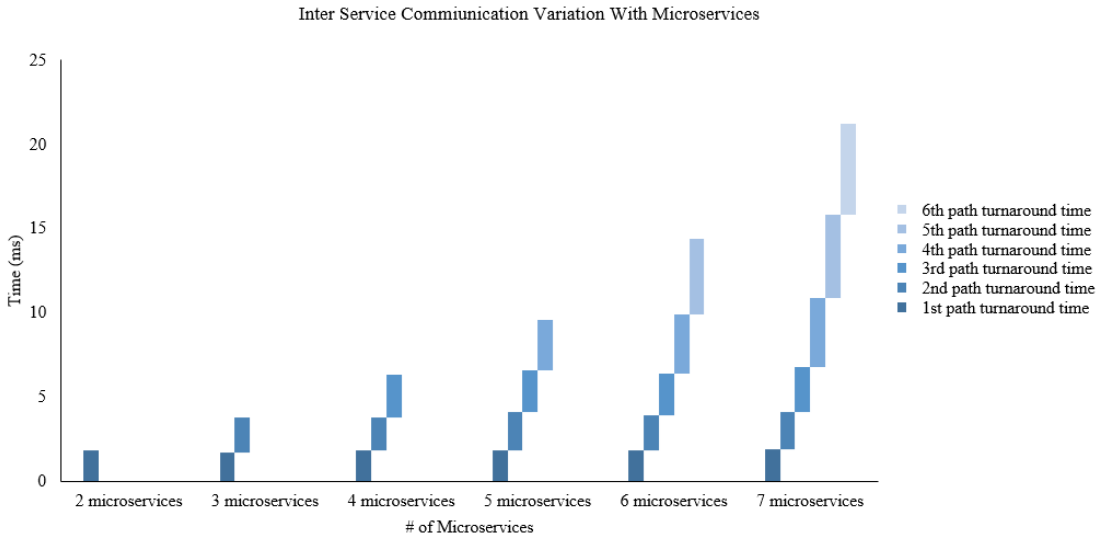

Fig. 5.16: Inter-service communication variation with microservices

The stacked column graph depicted in the above figure illustrates the relationship between the inter-service communication turnaround time and the number of microservices involved in each service-to-service communication. Data for this graph was collected from the logs of each microservice, excluding any logic processing time. When a new microservice is introduced through the decomposition of an existing one to isolate single responsibilities, the turnaround time for inter-service communication tends to increase. This is attributed to the consumption of network layer resources with each communication packet. As indicated in the graph, after the addition of the seventh microservice, the turnaround time for each layer experiences a notable increase. Further additions of microservices to fulfil single-user requirement led to a subsequent rise in the inter-service turnaround time.

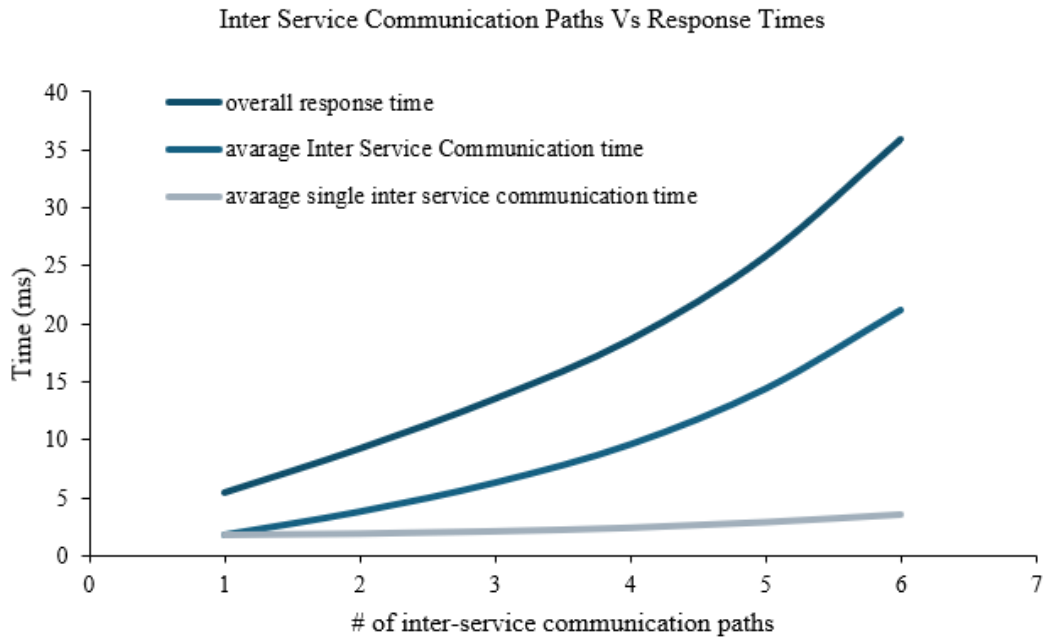

Fig. 5.17: Response time variation with communication paths

The graph up there in above compares how long it takes for different parts of a microservice setup to respond. We got the response time data from JMeter stats, and we figured out the time it takes for messages to go between services using logs and Python scripts. Looking at the graph, it's clear that having more ways for services to talk to each other increases the overall response time, which isn't great for users. After adding the fifth communication path, the time it takes for messages to move between services races up a lot compared to before.

5.6 Reference System

To evaluate the reference architecture, we have developed a sample system consisting of the Order Processing and Stock Management microservices. These two microservices communicate using the proposed solution and are deployed in a cloud-native environment. We have utilized the cloud vendors' services for all components of the system. This evaluation aims to verify that all essential quality attributes are evident in the reference architecture provided.

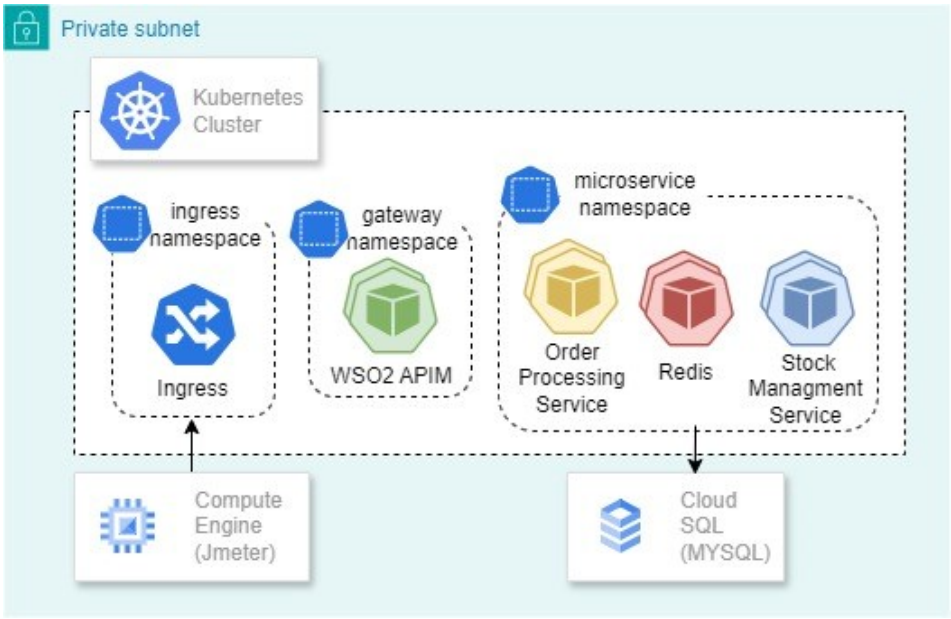

Fig. 5.18: Reference system deployment architecture

The image above illustrates the deployment of architecture utilized to evaluate the order processing system. This system, simulated for evaluation purposes, incorporates a microservice developed with Java and Spring Boot technologies. Upon receiving API requests from external clients, this microservice processes orders and communicates with the stock management microservice via a stream-based communication mechanism to update stock. For the REST-based API front, the WSO2 API Manager is employed alongside an enhanced interservice communication model, which Redis facilitates. The WSO2 API Manager, chosen for its compatibility with cloud-native solutions, is favoured for its lightweight and open-source nature. Deployment on Google Cloud is preferred due to its robust support for Kubernetes services and Cloud SQL. Apache JMeter, a commonly used load-generation tool in enterprises, is employed for testing purposes, leveraging its familiarity and reliability. The JMeter-deployed VM is positioned within the Kubernetes cluster subnet to minimize network latencies. For testing, 50 concurrent users are utilized to generate API traffic.

The graph below illustrates the overall system response time fluctuation observed during the load test. On average, processing a complete request takes approximately 7 milliseconds, encompassing network latencies, WSO2 APIM passthrough, and microservice processing time. Specifically, communication between the two microservices accounts for around 1.8 milliseconds of this turnaround time, excluding any logic processing time, which is captured through the logs of both microservices.

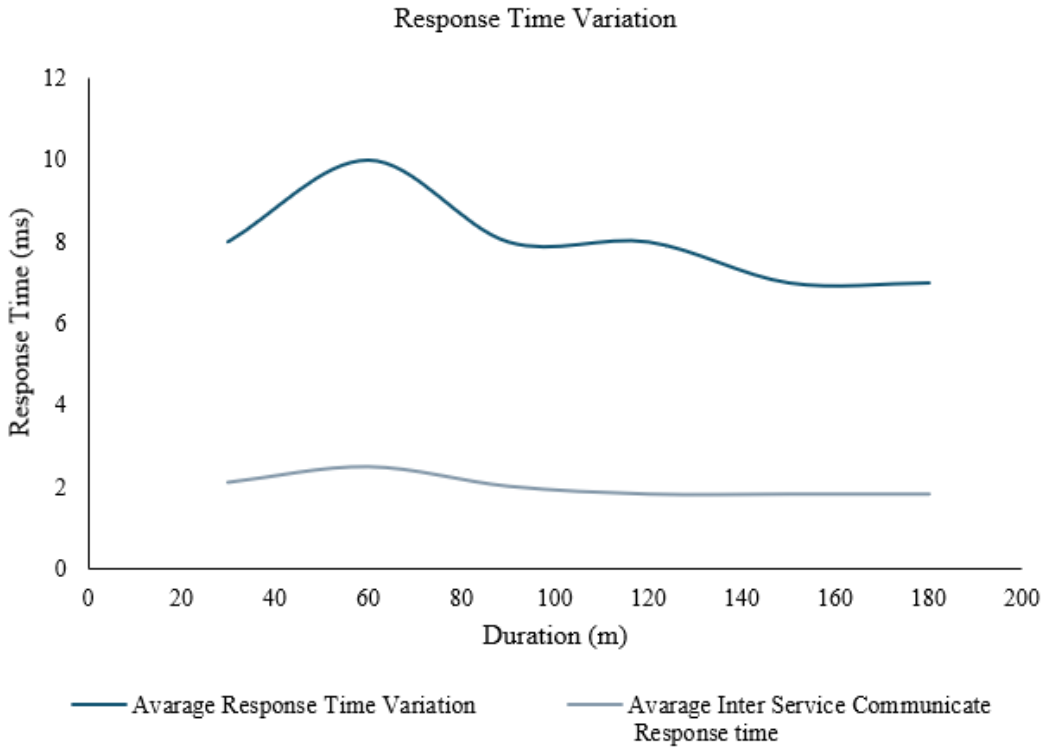

Fig. 5.19: Reference system response time variation

In the proposed reference architecture, all inter-service communication between microservices is managed using the newly developed TCP stream-based request-response mechanism supported by Redis. The API traffic initially passes through the API manager before reaching the appropriate microservice, and further internal communication among microservices utilizes our optimized inter-service communication strategy. This approach demonstrates that our research output can be effectively applied to real-world microservices to achieve business objectives. The architecture comprehensively covers essential aspects of microservices principles, containerization, orchestration, and communication strategies, addressing the complexities of inter-service communication in distributed environments. By using research outcome, organizations can significantly enhance the efficiency, scalability, and performance of their microservice-based applications. The solution minimizes latency, supports cloud-native adoption, and improves business agility, making it a practical choice for meeting the evolving demands and future trends of modern microservice architectures [136]. However, the reliance on Redis as a third-party tool for message delivery could be eliminated if future advancements guarantee reliable message delivery solely through TCP streams, reducing vendor lock-in. By referring to the research methodology, other researchers can implement provided solutions for other programming languages as well.

As a research outcome, the emphasis on selecting appropriate communication mechanisms based on specific use cases offers valuable guidance for architects and

developers navigating the complexities of microservices. Evaluation results demonstrate that the application's response time and turnaround time performance remain stable while meeting software quality attributes with the provided reference architecture. Developers can leverage this proven reference architecture for future microservice-based system development, enhancing agility and scalability. As microservices continue to gain popularity for their adaptability, insights from this reference architecture strengthen the groundwork for resilient, adaptable, and high-performance microservices systems. The refinement and adaptation of this architecture across diverse application domains hold promise for advancing microservices development and evolving best practices in distributed systems.

5.7 Concluding Remarks

This chapter has presented the results and evaluation of the proposed solution for optimizing inter-service communication within microservices architecture. The analysis began with a comprehensive comparison of inter-service communication protocols, focusing on controlled throughput, payload size, and request methods, providing valuable insights into the strengths and weaknesses of each protocol. The evaluation of the proposed solution was conducted through various testing scenarios, demonstrating its performance under different conditions. These scenarios highlighted the effectiveness of the solution in managing communication overhead, reducing latency, and maintaining efficiency across diverse operational contexts. Additionally, the suitability of the proposed solution for cloud-native environments was thoroughly assessed, confirming its alignment with cloud-native principles. Further analysis was conducted to evaluate the impact of the number of communication segments on overall performance, providing a detailed understanding of how segmentation affects inter-service communication in microservices. This was complemented by a reference architecture for cloud native microservices with the optimized inter service communication strategy. In conclusion, the results and evaluations presented in this chapter confirm that the proposed solution significantly enhances inter-service communication in microservices architecture, particularly in cloud-native environments. The findings provide a strong foundation for the practical application of this solution in real-world systems. The final chapter will synthesize these results and discuss the broader implications of the research, offering conclusions and potential directions for future work.

CHAPTER 6

CONCLUSION

6.1 Introduction

In the preceding sections, we delved into the research problem, explored relevant literature, elucidated our methodology for addressing the issue, and presented our findings. In this chapter, we revisit the articulated objectives to ascertain their fulfilment. Additionally, we reflect on any constraints encountered during the research process and delineate prospective directions for advancing this field. Through this endeavor, we aim to provide a comprehensive overview of our accomplishments, identify areas for improvement, and offer insights into potential avenues for further investigation within this domain.

6.2 Problem and Contribution

With microservice architecture, we can implicitly achieve scalability, maintainability, portability, and other attributes of software quality. The nature of the microservices architecture is that we deploy independent services in a distributed environment, and those services need to be communicated with the respective services in order to produce the functional requirements to the users. Unlike in the monolithic system, these communications need to be done over the dynamic network component, which affects the microservice performance in terms of response time and throughput. This is a critical issue in the microservice architecture and some of the points will become constraint points for migrating monolithic systems to microservice based systems. The conducted literature review also indicates this research problem is valid. Most of the microservices-based research is conducted in the area of microservice frameworks and microservice decomposition techniques. In this research, we mainly focused on enhancing inter-service communication in microservice architecture. From the previous studies, we found the most trending and widely used protocols such as HTTP, gRPC, Web Socket in the software industry, and we checked those protocols' suitability with microservice architecture. gRPC protocols we dropped after our evaluation because of the complexity of the implementation and Web Socket dropped due to their poor performance. We chose the HTTP(S) protocol as a benchmark protocol for our research study, and it is also a widely used protocol for academic and industry implementations according to our literature review findings. The primary contribution of this research is the development of an optimized inter-service communication mechanism for microservices, leveraging a TCP-based stream. Unlike conventional methods such as HTTP, Web Socket or gRPC, which introduce increased network overhead, latency, and programming complexity, the proposed solution offers a more efficient alternative while preserving the reliability of traditional approaches.

All the messages that are transmitted over the TCP stream have been serialized and converted into the binary form. Using this, network layer resource utilization will be reduced during the transmission. Moreover, In microservices, TCP streams are established at startup and persist throughout the service's lifecycle, ensuring continuous, low-latency communication until the service shuts down. With that, we could reduce the time taken for network sockets to open and close time per connection. When using the TCP-based streams, we need to guarantee the exact message delivery and only once that message is delivered between the microservices. In order to do that, we used Redis as a middleware to ensure the message guarantee and all the microservices were communicating through the Redis server. Another advantage of this is that Redis brings end-to-end message tracing for TCP-based streams, which brings advantages over monitoring and tracing. By incorporating all these facts, we implemented a user library to communicate with services with less complexity and is similar to the HTTP-based implementation at the code level.

Nowadays, most enterprise-grade organizations are moving their systems to the cloud environment, and now, the trend is cloud-native level microservice architecture. By considering the current trends, we adjusted our contribution to match the cloud-native concepts. All microservices are packed as containers and deployed as pods in the Kubernetes container orchestration system. Adapting to the cloud native concepts, microservice architecture gets more advantages over scalability, maintainability, and portability. Evaluation results also showed that the contributed strategy for inter-service communication in the cloud-native environment performed well and improved overall microservices' performance in terms of response time and throughput compared to the traditional HTTP method. Its seamless integration with cloud-native environments makes it a scalable and future-proof solution for modern microservice architecture, filling a critical gap in both academic research and industry practice. Evaluation not only focuses on the performance, but we also check the scalability factor and show that no performance degradation or blockers for both vertical scaling and horizontal scaling. Hence, we proposed the reference architecture with the suggested strategy for future development. This contribution will be more beneficial to all software architects to design microservice-based architectures with optimized inter-service communication. Further, we have continued research to find out the optimal inter-service communication segments to produce the functional requirements in the microservice architecture. Evaluation results show that up to five inter-service communication segments perform well on microservice architecture. This finding contributes to all system design software architects deciding to decompose microservices with functional requirements. Research contribution over the optimized inter-service communication strategy offers a multitude of benefits for microservices architectures. Primarily, it significantly enhances system performance by reducing latency, thereby improving response times and throughput. This directly translates to better user experience and increased overall application efficiency. Moreover, such optimization facilitates scalability, allowing services to handle higher loads efficiently

without compromising performance. Cost efficiency is another advantage, as optimized communication minimizes resource utilization, particularly in cloud-based deployments where resources are metered. Additionally, these strategies enhance system resilience by mitigating the impact of service failures or network issues. They also contribute to maintainability by simplifying the system's complexity, making it easier to understand, maintain, and debug. Furthermore, optimized communication affords flexibility in technology choices, ensuring compatibility with various deployment environments like Kubernetes. Altogether, an optimized inter-service communication strategy is crucial for building resilient, efficient, and scalable microservices architectures capable of meeting the dynamic demands of modern applications.

6.3 Concluding Findings for Research Questions

To address the research questions, each chapter of the thesis provides a structured exploration, and answers based on the specific aspects of software architecture and inter-service communication methods.

Q1: What are the key architectural differences and trade-offs between monolithic and microservice architectures, specifically in terms of inter-service communication mechanisms and their impact on performance and scalability?

Monolithic and microservices architectures offer distinct structural paradigms with specific trade-offs, particularly in terms of inter-service communication mechanisms and their impact on performance and scalability. Monolithic architecture features a unified, tightly coupled codebase, where components communicate internally without the need for network-based protocols. This simplifies initial development and deployment, allowing for easier testing and faster early-stage development. However, as applications grow, monoliths struggle with scalability and adaptability, as any modification requires rebuilding and redeploying the entire application. In contrast, microservices emphasize a decoupled, modular approach where independently deployable services communicate using protocols such as HTTP, gRPC, or message queues. While this structure provides scalability, technology diversity, and fine-grained control over system resources, it introduces communication overhead, network latency, and complexity in maintaining reliable inter-service communication. Consequently, while monolithic architecture may suit smaller, less dynamic applications, microservices are better suited for scalable, adaptable systems, especially in cloud-native environments that require efficient communication mechanisms. Chapters 1 and 2 of this thesis provide an in-depth discussion of the architectural differences, trade-offs, and the role of communication mechanisms in monolithic and microservices architectures.

Q2: How do most trending existing inter-service communication methods perform in terms of latency and throughput?

Most trending inter-service communication methods, such as HTTP, gRPC, WebSocket, and asynchronous messaging, exhibit varied performance characteristics in terms of latency and throughput. HTTP, widely adopted for its simplicity, often struggles with latency and payload overhead in high-throughput scenarios, leading to performance bottlenecks, especially in complex, large-scale environments. gRPC, designed for efficient inter-service communication, provides lower latency and higher throughput than HTTP due to its lightweight binary protocol, making it more suitable for high-performance applications, although it requires additional setup; it is not as universally adopted, and it is more complex for implementation leading to high maintenance cost for development. WebSocket offers bidirectional, low-latency communication, which can be advantageous in real-time applications, but it lacks robust native support for error handling and scalability, posing challenges in more distributed systems. Asynchronous messaging, typically implemented via message queues, excels in resilience and decoupling but can introduce additional latency due to queuing delays, and its complexity can grow as the system scales. In light of recent technological advancements in cloud-native and containerized environments, these methods, while effective, reveal certain limitations. This indicates a need for more adaptive, optimized communication strategies that align with modern performance and scalability demands, as examined in Chapter 2 of this thesis.

Q3: What are the critical characteristics and requirements for an optimized inter-service communication mechanism within microservice architecture?

An optimized communication mechanism for microservice architecture should combine synchronous and asynchronous communication to leverage their respective strengths, prioritize efficient data transfer through protocols like TCP, ensure scalability to handle increased traffic, and adhere to cloud-native principles. Additionally, it should abstract away complexities of message serialization and deserialization, incorporate message delivery, and support observability for effective monitoring and troubleshooting. By embodying these characteristics, a communication mechanism can effectively enhance performance, scalability, and reliability in microservice-based systems. These characteristics are foundational for overcoming the inherent challenges in microservice communication, as established in Chapter 3.

Q4: How can cloud-native technologies such as container orchestration be leveraged to design a novel communication mechanism that addresses the limitations of existing approaches?

To design a novel inter-service communication solution, cloud-native technologies can be strategically leveraged to address the limitations of traditional communication methods. Containerization, as discussed in Chapter 4, enables both vertical and horizontal scalability, allowing microservices to dynamically adjust resource allocation based on demand. This improves performance and ensures high availability by deploying services across multiple nodes, ensuring continuous operation even during failures. Using container orchestration platforms like Kubernetes provides additional benefits, such as managing network subnets that reduce latency at the network layer. These platforms also facilitate TCP-level communication between services, offering reliable, low-latency message transmission while maintaining the flexibility needed for modern distributed architectures. By integrating these cloud-native technologies, the novel communication mechanism overcomes traditional constraints, improving scalability and performance in microservice environments.

Q5: How can a prototype of the proposed communication mechanism be implemented with cloud-native technologies and environments?

The researched artefact is developed as a Java library, allowing users to easily integrate it into their microservice applications without complex implementation, similar to HTTP communication. The proposed mechanism adheres to cloud-native principles, enabling deployment in cloud environments rather than being restricted to on-premises servers. Once the services are implemented using this communication mechanism, they can be packaged and deployed as containers within a container orchestration platform like Kubernetes. This approach ensures scalability, easy service management, and network optimization. Chapter 5 details this implementation, providing a high-level architecture diagram and functional breakdown of each component, illustrating how cloud-native technologies like containerization, dynamic scaling, and service discovery are used to realize the proposed solution.

Q6: Compared to the existing HTTP communication method, how does the proposed mechanism impact key performance metrics (latency, scalability) in a test environment?

The proposed communication mechanism significantly improves key performance metrics such as latency and scalability compared to existing methods like traditional HTTP communication. As demonstrated in Chapter 6, the implemented solution reduces inter-service communication turnaround time and overall application latency, thereby enhancing system throughput. These improvements are particularly evident in cloud-native environments, where the mechanism's ability to handle dynamic scaling both vertically and horizontally by ensuring seamless adaptation to varying workloads. The evaluation results confirm that the proposed solution offers superior performance,

making it a better fit for cloud-native applications, where efficient scaling and reduced latency are critical.

Q7: What are the key design principles and best practices for implementing the proposed mechanism in real-world applications particularly in cloud native environments?

The key design principles and best practices for implementing the proposed communication mechanism in real-world applications emphasize performance optimization, scalability, and adherence to cloud-native principles. A central consideration is selecting a suitable cloud platform that aligns with the specific requirements of microservices, ensuring support for containerization, service orchestration, and high availability. Effective integration of the introduced communication strategy involves designing for low latency and high throughput, achievable through efficient serialization, binary protocols, and streamlined message handling. Employing a modular microservices architecture facilitates scalability and allows independent deployment of services, minimizing downtime and enhancing flexibility. Monitoring and management tools are essential for observing system performance, tracking communication metrics, and ensuring reliability, especially in high-demand environments. By following the provided a reference architecture and adhering to best practices, organizations can leverage the proposed mechanism's full potential, resulting in scalable, extensible, and high performing solutions that meet the demands of modern, cloud-native applications.

The concluding chapter revisits the initial problem, explaining how the researched communication strategy effectively addresses the limitations of current methods. It demonstrates the achievement of the research objectives outlined in the introduction by developing and evaluating this new mechanism. The chapter also acknowledges the research limitations, identifying areas where the solution may fall short or require further refinement. Additionally, it suggests potential directions for future research, aiming to enhance inter-service communication in software applications by leveraging emerging technologies and addressing unresolved challenges. This chapter provides a comprehensive analysis through a structured approach, addressing each research question and offering a practical solution grounded in the findings.

6.4 Achievement of the Objectives

This analysis serves as the basis for drawing conclusions about the research. The primary aim of this study is to identify an optimized and effective mechanism for inter-service communication among microservices deployed within a cloud-native environment by leveraging emerging technologies and assessing the impact on overall

application performance. In line with that aim the overview of achieving the research objectives can be listed as follows,

- To identify the strengths and limitations of existing inter-service communication methods in microservice architecture.
    - ✓ There is a significant shift in enterprise-grade software from monolithic architecture to microservice architecture. Within this framework, various protocols, such as HTTP, WebSockets, and gRPC, are employed for inter-service communication. Nevertheless, certain protocols exhibit suboptimal performance in facilitating inter-service communication, while others may demonstrate satisfactory performance, but encounter challenges related to scalability and complexity.

- To investigate similar research that explored this problem and identify their weaknesses in the present context.
    - ✓ Some research has proposed protocols operating at the TCP layer; however, these protocols are often unsuitable for dynamic cloud-native environments. Additionally, adapting existing systems to accommodate these protocols typically requires significant modifications. Another limitation of TCP-based approaches is the lack of guaranteed message delivery. Furthermore, much of the research in the realm of microservices focuses primarily on framework development and microservice decomposition techniques.

- To propose a suitable communication mechanism to improve overall system performance.
    - ✓ The proposed TCP stream-based optimized inter-service communication strategy significantly enhances performance compared to traditional HTTP-based communication. Utilizing the developed library, this approach offers a simplified method for inter-service communication while minimizing complexity. To ensure precise message delivery, including an 'exactly once' delivery guarantee, Redis Streams are integrated into the solution. Furthermore, to reduce resource consumption at the network layer, all messages are serialized and transmitted in binary form over the streams. This solution also eliminates open and closed socket connections for each request, thereby improving overall efficiency.

- To develop the prototype using the proposed solution with cloud-native technologies.
    - ✓ The prototype was developed using Java programming language and the Spring Boot microservice framework. These microservices are packaged as containers and deployed within a cloud-based Kubernetes environment. Both the microservice pods and Redis servers are

configured as deployments in Kubernetes, facilitating the seamless scalability of the pods.

- To evaluate the proposed mechanism in the cloud-native environment.
  - ✓ The system was evaluated through various methods to demonstrate that the proposed strategy outperforms traditional inter-service communication methods. A JMeter client was utilized to generate traffic within the same network to minimize latency. Key metrics such as turnaround time, response time, and throughput were collected for analysis. Based on the evaluated data, the proposed strategy performs better than the traditional HTTP protocol.

- To present a reference architecture of the proposed mechanism for a cloud environment.
  - ✓ A reference architecture is designed and presented that incorporates the optimized inter-service communication strategy. Furthermore, this architecture adheres to essential software quality attributes, ensuring the overall performance and reliability of the system.

6.5 Limitations

This research, through its iterative approach to improving the proposed concepts, has presented an evaluated and validated solution as its contribution. However, the following limitations were observed which require to be noted for a critical appraisal of the work. In our proposed solution, all inter-service communication is conducted via TCP layer streams, and middleware is employed to ensure message delivery reliability during transmission over these streams. For the proof of concept, we used Redis as a third-party middleware to facilitate reliable message delivery. However, any middleware supporting TCP-based streaming can be utilized. A key limitation of this approach is the risk of vendor lock-in associated with relying on specific middleware, such as Redis, which can result in a tightly coupled system dependent on a particular vendor. This dependency may restrict flexibility and adaptability to alternative technologies in the future. However, advancements in future technologies that guarantee message delivery without requiring dedicated middleware could alleviate this limitation, offering a more adaptable, decoupled solution and enhancing the versatility of the proposed communication strategy. Additionally, a significant limitation arises from the research's scope, which primarily focuses on service decomposition and its interactions within microservices, particularly addressing performance issues. While service decomposition is crucial for transitioning from monolithic to microservice architecture, it represents only a subset of the challenges involved in such transformations. Other important areas, such as database management, shared state, and transactional integrity across distributed services,

remain unaddressed in this work. Though equally critical, these factors fall outside the research scope, which explicitly targets microservice architecture.

The proposed communication strategy relies heavily on the 'exactly once' delivery guarantee, which may not be maintained under all network conditions, particularly during network partitioning, service failures, or sudden disconnections. Although the system incorporates reliability mechanisms, further enhancements are necessary to improve its robustness against network faults. The assumption of a stable and reliable network infrastructure may not always align with real-world conditions, especially in highly distributed or resource-constrained environments. The research likely focused on controlled environments with stable network conditions and underexplored real-world scenarios with significant variability in network reliability, such as mobile or IoT networks, where the solution might not perform optimally.

Moreover, the solution is best suited for environments where performance optimization is the primary goal. It leverages TCP-based binary serialization to minimize latency and enhance throughput; however, this approach may not be ideal in scenarios prioritizing extensibility, interoperability, or human-readable message formats.

In conclusion, while the proposed strategy effectively addresses a specific performance bottleneck in microservices communication, it does so within the stated scope of this research. Broader considerations for transforming monolithic architectures and managing diverse network conditions necessitate further research and refinements for future solutions.

6.6 Future Work

This dissertation presents ongoing research in computer science focused on the rapidly evolving domains of cloud-native microservice architecture and inter-service communication. These areas are critical in modern software development, as they significantly influence the scalability, flexibility, and performance of distributed systems. The research investigates innovative strategies to optimize communication between microservices, addressing latency, connection overhead, and resource utilization challenges. Given the continuous evolution of these fields, there are numerous avenues for future research. Continued exploration in these areas can contribute to the development of more robust, high-performance microservices that meet the increasing demands of cloud-native applications.

Redis is frequently employed as an in-memory data structure processor in cloud-native microservice architectures to enhance performance and scalability. However, while Redis is highly effective, it introduces dependencies, particularly in large-scale distributed systems. Future research could focus on developing a custom TCP layer stream mechanism tailored specifically to the needs of microservice architecture for inter-service communication. This could involve designing distributed, highly scalable

streams that utilize binary data for communication, thereby minimizing latency, improving fault tolerance, and integrating more effectively with the specific microservice ecosystem in use.

Another area of exploration could involve optimizing existing mechanisms that bypass Redis, such as direct peer-to-peer communication between microservices or leveraging emerging technologies through the framework introduced in this research. Custom mechanisms could also be developed with enhanced security features, reducing the attack surface and minimizing the risks associated with third-party tools. Efficiency improvements could be achieved by minimizing overhead or employing machine learning algorithms to dynamically predict and manage data flows. Developing a system that scales horizontally and vertically while ensuring message reliability over TCP-based streams—without reliance on Redis or similar services—could significantly advance the field by reducing dependencies and lowering costs for microservice-based applications.

Furthermore, enhancing the 'exactly once' delivery guarantee represents another critical area for future investigation. The current strategy's reliance on stable network conditions may not be feasible in all real-world distributed environments. Further research could focus on improving fault tolerance mechanisms, particularly in managing network failures, service disruptions, or sudden disconnections. This would involve developing more robust methods for ensuring message reliability during network partitioning or service outages.

Microservice architectures benefit from language-agnostic communication protocols, yet specific optimizations and integrations may be language-specific. While this research primarily focuses on the Java programming language, there is significant potential to extend these findings to a broader range of programming languages, each with unique strengths. Expanding this research to include various programming languages would enable developers in diverse environments to adopt the proposed communication strategies. For example, exploring how these strategies could be implemented in languages such as Python, Go, or Rust would provide valuable insights into their adaptability and performance. Each programming language has its own paradigms and performance characteristics. Future research could investigate how inter-service communication strategies could be fine-tuned for each language, such as optimizing memory management in C++ or leveraging asynchronous programming models in JavaScript and Python.

Another avenue could involve creating a standardized framework or API that supports multiple languages, enabling seamless communication and integration between microservices developed in different programming languages. This would further promote polyglot microservice architectures, where each service can be developed in the most suitable language without compromising performance or communication efficiency. Comprehensive benchmarking and testing across various languages could provide a detailed analysis of how different programming languages affect the

performance of the proposed inter-service communication strategies, offering practical guidance to developers when selecting the appropriate language for their microservice applications.

In conclusion, these avenues for future research present significant potential to enhance the software quality attributes of cloud-native microservice architectures. By developing custom mechanisms to replace existing dependencies like Redis and expanding support to additional programming languages, the field can continue to evolve, meeting the diverse needs of modern distributed systems. These efforts will promote the adoption of the proposed communication strategy and framework, leading to the development of more robust, adaptable, and high-performing microservices, ultimately driving further innovation in inter-service communication and cloud-native architecture.

6.7 Summary

In a microservices architecture, services are distributed and operate independently, contrasting with monolithic applications where components are tightly coupled. This independence necessitates effective inter-service communication to meet business requirements. However, such communication introduces additional latency, which previous research indicates can degrade overall application performance and increase response times. Existing communication protocols frequently encounter delays in establishing and terminating connections, particularly when managing large payloads, resulting in significant latency challenges. Given its extensive adoption in enterprise-grade microservice architectures, academic research and our experiments, this study evaluates the research component using the HTTP protocol for inter-service communication. While gRPC demonstrates superior performance in terms of throughput and response time, its implementation complexity and scalability limitations, along with the inadequacy of WebSocket for inter-service communication, position HTTP as the most viable option. As the majority of microservices are now deployed in cloud-native environments, this study focuses on optimizing inter-service communication within these architectures. It introduces a novel TCP stream-based communication strategy designed to enhance the efficiency of inter-service interactions. By leveraging Redis Stream to ensure reliable message delivery and constructing a message-passing system modelled on request/response interactions akin to the HTTP protocol, the study effectively addresses inherent latency issues. Upon initialization of each microservice, a TCP-based socket connection is established, streamlining the process of opening and closing connections. Additionally, payloads are serialized and transmitted in binary format, minimizing network resource usage and further optimizing performance. The entire system, encompassing both the microservices and Redis servers, is deployed within a cloud-native Kubernetes environment, capitalizing on the scalability, availability, and portability benefits afforded by cloud-native technologies. Comprehensive testing of the proposed solution demonstrates its efficacy in reducing latency and enhancing overall application response times within cloud-native environments. The research not only addresses all the posed research questions but also successfully meets the established objectives, providing a sustainable solution to the identified problem. While acknowledging the limitations associated with reliance on Redis servers, the research

suggests future directions for developing a custom model to mitigate these constraints. The reference architecture proposed in this study offers a robust framework for software architects aiming to develop reliable, high-performance microservices. This work paves the way for ongoing innovation in optimized inter-service communication strategies, contributing to the advancement of microservices in cloud-native environments.

REFERENCES

[1] D. C. Schmidt and T. Suda, “Experiences with an object-oriented architecture for developing dynamically extensible distributed system management software,” in *1994 IEEE GLOBECOM. Communications: The Global Bridge*, Nov. 1994, pp. 500–506 vol.1. doi: 10.1109/GLOCOM.1994.513571.

[2] I. Kaur, N. Kaur, A. Ummat, J. Kaur, and N. Kaur, “Research Paper on Object Oriented Software Engineering,” vol. 7, no. 4, 2016.

[3] O. Nierstrasz and S. Demeyer, “Object-Oriented Reengineering Patterns,” in *Proceedings of the 26th International Conference on Software Engineering*, in ICSE ’04. USA: IEEE Computer Society, May 2004, pp. 734–735.

[4] K. W. Ng and C. K. Luk, “A survey of languages integrating functional, object-oriented and logic programming,” *Microprocess. Microprogramming*, vol. 41, no. 1, pp. 5–36, Apr. 1995, doi: 10.1016/0165-6074(94)00017-5.

[5] E. Murphy-Hill and D. Grossman, “How programming languages will co-evolve with software engineering: a bright decade ahead,” in *Future of Software Engineering Proceedings*, in FOSE 2014. New York, NY, USA: Association for Computing Machinery, May 2014, pp. 145–154. doi: 10.1145/2593882.2593898.

[6] K. Smolander, M. Rossi, and S. Purao, “Software architectures: Blueprint, Literature, Language or Decision?,” *Eur. J. Inf. Syst.*, vol. 17, no. 6, pp. 575–588, Dec. 2008, doi: 10.1057/ejis.2008.48.

[7] F. Rademacher, J. Sorgalla, P. N. Wizenty, S. Sachweh, and A. Zündorf, “Microservice architecture and model-driven development: yet singles, soon married (?),” in *Proceedings of the 19th International Conference on Agile Software Development: Companion*, in XP ’18. New York, NY, USA: Association for Computing Machinery, May 2018, pp. 1–5. doi: 10.1145/3234152.3234193.

[8] Z. Liu *et al.*, “The architectural design and implementation of a digital platform for Industry 4.0 SME collaboration,” *Comput. Ind.*, vol. 138, p. 103623, Jun. 2022, doi: 10.1016/j.compind.2022.103623.

[9] Z. Wan, Y. Zhang, X. Xia, Y. Jiang, and D. Lo, “Software Architecture in Practice: Challenges and Opportunities,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, in ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, Nov. 2023, pp. 1457–1469. doi: 10.1145/3611643.3616367.

[10] M. Hassan, “Software Architecture Between Monolithic and Microservices Approach,” Feb. 25, 2024, *Rochester, NY*: 4753649. doi: 10.2139/ssrn.4753649.

[11] D. Krug, R. Chanin, and A. Sales, “Exploring the Pros and Cons of Monolithic Applications versus Microservices:,” in *Proceedings of the 26th International Conference on Enterprise Information Systems*, Angers, France: SCITEPRESS - Science and Technology Publications, 2024, pp. 256–263. doi: 10.5220/0012703300003690.

[12] D. Saxena and B. Bhowmik, “Paradigm Shift From Monolithic to Microservices,” in *2023 IEEE International Conference on Recent Advances in Systems Science and Engineering (RASSE)*, Nov. 2023, pp. 1–7. doi: 10.1109/RASSE60029.2023.10363466.

[13] “Microservices Architecture Market Trends, Size And Industry Analysis Report 2033.” Accessed: May 21, 2024. [Online]. Available: https://www.thebusinessresearchcompany.com/report/microservices-architecture-global-market-report

[14] “Microservices Architecture Market Share, Size 2024-2032.” Accessed: May 21, 2024. [Online]. Available: https://www.imarcgroup.com/microservices-architecture-market

[15] V. Alkkiomäki, “The role of service-oriented architecture as a part of the business model,” *Int J Bus Inf Syst*, vol. 21, no. 3, pp. 368–387, Feb. 2016, doi: 10.1504/IJBIS.2016.074764.

[16] M. S. F. Fayaza, “Service Oriented Architecture in Enterprise Integration”.

[17] L. D. S. B. Weerasinghe and I. Perera, “An exploratory evaluation of replacing ESB with microservices in service-oriented architecture,” in *2021 International Research Conference on Smart Computing and Systems Engineering (SCSE)*, Sep. 2021, pp. 137–144. doi: 10.1109/SCSE53661.2021.9568289.

[18] “Soa Industry Statistics • WorldMetrics.” Accessed: May 21, 2024. [Online]. Available: https://worldmetrics.org/soa-industry-statistics/

[19] J. Porter, D. A. Menascé, and H. Gomaa, “A decentralized approach for discovering runtime software architectural models of distributed software systems,” *Inf. Softw. Technol.*, vol. 131, p. 106476, Mar. 2021, doi: 10.1016/j.infsof.2020.106476.

[20] V. Bushong *et al.*, “On Microservice Analysis and Architecture Evolution: A Systematic Mapping Study,” *Appl. Sci.*, vol. 11, no. 17, p. 7856, Aug. 2021, doi: 10.3390/app11177856.

[21] “Mastering Chaos - A Netflix Guide to Microservices,” InfoQ. [Online]. Available: https://www.infoq.com/presentations/netflix-chaos-microservices/

[22] “Cloud Microservices Market Size, Growth Outlook 2024-2032,” Global Market Insights Inc. Accessed: May 22, 2024. [Online]. Available: https://www.gminsights.com/industry-analysis/cloud-microservices-market

[23] E. R. https://www.emergenresearch.com, “Cloud Microservices Market Size, Trend, Demand Analysis till 2032.” Accessed: May 22, 2024. [Online]. Available: https://www.emergenresearch.com/industry-report/cloud-microservices-market

[24] N. Kobayashi, A. Nakamoto, M. Kawase, F. Sussan, M. Ioki, and S. Shirasaka, “Managing a Monolithic System or a System-of-Systems? An Assurance Case Approach to Reach Intra-organizational Consensus,” in *2018 7th International Congress on Advanced Applied Informatics (IIAI-AAI)*, Yonago, Japan: IEEE, Jul. 2018, pp. 688–693. doi: 10.1109/IIAI-AAI.2018.00144.

[25] O. Al-Debagy and P. Martinek, “A Comparative Review of Microservices and Monolithic Architectures,” in *2018 IEEE 18th International Symposium on*

Computational Intelligence and Informatics (CINTI), Budapest, Hungary: IEEE, Nov. 2018, pp. 000149–000154. doi: 10.1109/CINTI.2018.8928192.

[26] M. Hamza, "Transforming Monolithic Systems to a Microservices Architecture," *ACM SIGSOFT Softw. Eng. Notes*, vol. 48, no. 1, pp. 67–69, Jan. 2023, doi: 10.1145/3573074.3573091.

[27] K. Gos and W. Zabierowski, "The Comparison of Microservice and Monolithic Architecture," in *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, Lviv, Ukraine: IEEE, Apr. 2020, pp. 150–153. doi: 10.1109/MEMSTECH49584.2020.9109514.

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

[29] R. Belafia, P. Jeanjean, O. Barais, G. L. Guernic, and B. Combemale, "From Monolithic to Microservice Architecture: The Case of Extensible and Domain-Specific IDEs," in *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, Fukuoka, Japan: IEEE, Oct. 2021, pp. 454–463. doi: 10.1109/MODELS-C53483.2021.00070.

[30] S. Li *et al.*, "Understanding and addressing quality attributes of microservices architecture: A Systematic literature review," *Inf. Softw. Technol.*, vol. 131, p. 106449, Mar. 2021, doi: 10.1016/j.infsof.2020.106449.

[31] N. Salaheddin Elgheriani and N. D. Ali Salem Ahme, "MICROSERVICES VS. MONOLITHIC ARCHITECTURES [THE DIFFERENTIAL STRUCTURE BETWEEN TWO ARCHITECTURES]," *MINAR Int. J. Appl. Sci. Technol.*, vol. 4, no. 3, pp. 500–514, Sep. 2022, doi: 10.47832/2717-8234.12.47.

[32] S. Pulparambil and Y. Baghdadi, "Service oriented architecture maturity models: A systematic literature review," *Comput. Stand. Interfaces*, vol. 61, pp. 65–76, Jan. 2019, doi: 10.1016/j.csi.2018.05.001.

[33] J. Ma, H. Yu, and J. Guo, "Research and Implement on Application Integration Based on the Apache Synapse ESB platform," *AASRI Procedia*, vol. 1, pp. 82–86, 2012, doi: 10.1016/j.aasri.2012.06.015.

[34] K. Baylov and A. Dimov, "Quality Characteristics for Service Oriented Architectures," in *Proceedings of the 2015 European Conference on Software Architecture Workshops*, in ECSAW '15. New York, NY, USA: Association for Computing Machinery, Sep. 2015, pp. 1–5. doi: 10.1145/2797433.2797488.

[35] M. Swientek, U. Bleimann, and P. S. Dowland, "Service-Oriented Architecture: Performance Issues and Approaches".

[36] J. Boleng and R. Sward, "Service-oriented architecture (SOA) concepts and implementations," in *Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology*, in HILT '13. New York, NY, USA: Association for Computing Machinery, Nov. 2013, pp. 11–12. doi: 10.1145/2527269.2527289.

[37] M. P. Papazoglou and W.-J. Van Den Heuvel, "Service oriented architectures: approaches, technologies and research issues," *VLDB J.*, vol. 16, no. 3, pp. 389–415, Jul. 2007, doi: 10.1007/s00778-007-0044-3.

[38] O. Aziz, M. S. Farooq, A. Abid, R. Saher, and N. Aslam, “Research Trends in Enterprise Service Bus (ESB) Applications: A Systematic Mapping Study,” *IEEE Access*, vol. 8, pp. 31180–31197, 2020, doi: 10.1109/ACCESS.2020.2972195.

[39] R. Haesen, M. Snoeck, W. Lemahieu, and S. Poelmans, “On the Definition of Service Granularity and Its Architectural Impact,” in *Active Flow and Combustion Control 2018*, vol. 141, R. King, Ed., in Notes on Numerical Fluid Mechanics and Multidisciplinary Design, vol. 141. , Cham: Springer International Publishing, 2008, pp. 375–389. doi: 10.1007/978-3-540-69534-9_29.

[40] H. Chindove, L. F. Seymour, and F. I. Van Der Merwe, “Service-oriented Architecture: Describing Benefits from an Organisational and Enterprise Architecture Perspective:,” in *Proceedings of the 19th International Conference on Enterprise Information Systems*, Porto, Portugal: SCITEPRESS - Science and Technology Publications, 2017, pp. 483–492. doi: 10.5220/0006383604830492.

[41] K. Avila, P. Sanmartin, D. Jabba, and M. Jimeno, “Applications Based on Service-Oriented Architecture (SOA) in the Field of Home Healthcare,” *Sensors*, vol. 17, no. 8, p. 1703, Jul. 2017, doi: 10.3390/s17081703.

[42] Z. D. Patel, “A Review on Service Oriented Architectures for Internet of Things (IoT),” in *2018 2nd International Conference on Trends in Electronics and Informatics (ICOEI)*, Tirunelveli: IEEE, May 2018, pp. 466–470. doi: 10.1109/ICOEI.2018.8553767.

[43] C. Phan, “Service Oriented Architecture (SOA) - Security Challenges and Mitigation Strategies,” in *MILCOM 2007 - IEEE Military Communications Conference*, Oct. 2007, pp. 1–7. doi: 10.1109/MILCOM.2007.4455012.

[44] K. M. Dhara, M. Dharmala, and C. K. Sharma, “A Survey Paper on Service Oriented Architecture Approach and Modern Web Services,” 2015.

[45] J. Bean, “Chapter 10 - XML Schema Basics,” in *SOA and Web Services Interface Design*, J. Bean, Ed., in The MK/OMG Press. , Boston: Morgan Kaufmann, 2010, pp. 185–209. doi: 10.1016/B978-0-12-374891-1.00010-1.

[46] M. Arvind Kumar, N. Renuka, S. Kirti, and S. Rajni, “Maintainability of Service-Oriented Architecture using Hybrid K-means Clustering Approach,” *Int. J. Perform. Eng.*, vol. 19, no. 1, p. 33, 2023, doi: 10.23940/ijpe.23.01.p4.3342.

[47] *Outlier Detection: Techniques and Applications*. Accessed: May 04, 2024. [Online]. Available: https://link.springer.com/book/10.1007/978-3-030-05127-3

[48] K. Mari, “Performance Analysis of Clustering Algorithms in Outlier Detection Based on Statistical Models and Spatial Proximity,” *Int. J. Comput. Sci. Inf. Technol.*, vol. 2, pp. 1747–1749, Jan. 2011.

[49] D. Ameller, X. Burgués, D. Costal, C. Farré, and X. Franch, “Non-functional requirements in model-driven development of service-oriented architectures,” *Sci. Comput. Program.*, vol. 168, pp. 18–37, Dec. 2018, doi: 10.1016/j.scico.2018.08.001.

[50] S. F. Abbas, R. K. Shahzad, M. Humayun, N. Jhanjhi, and M. Alamri, “SOA Issues and their Solutions through Knowledge Based Techniques – A Review,” 2019.

[51] N. Dragoni *et al.*, “Microservices: Yesterday, Today, and Tomorrow,” in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds., Cham: Springer International Publishing, 2017, pp. 195–216. doi: 10.1007/978-3-319-67425-4_12.

[52] N. Alshuqayran, N. Ali, and R. Evans, “A Systematic Mapping Study in Microservice Architecture,” in *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, Macau, China: IEEE, Nov. 2016, pp. 44–51. doi: 10.1109/SOCA.2016.15.

[53] M. Kalske, N. Mäkitalo, and T. Mikkonen, “Challenges When Moving from Monolith to Microservice Architecture,” in *Current Trends in Web Engineering*, vol. 10544, I. Garrigós and M. Wimmer, Eds., Cham: Springer International Publishing, 2018, pp. 32–47. doi: 10.1007/978-3-319-74433-9_3.

[54] Y. Abgaz *et al.*, “Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review,” *IEEE Trans. Softw. Eng.*, vol. 49, no. 8, pp. 4213–4242, Aug. 2023, doi: 10.1109/TSE.2023.3287297.

[55] Z. Ren *et al.*, “Migrating Web Applications from Monolithic Structure to Microservices Architecture,” in *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*, Beijing China: ACM, Sep. 2018, pp. 1–10. doi: 10.1145/3275219.3275230.

[56] D. Kuryazov, D. Jabborov, and B. Khujamuratov, “Towards Decomposing Monolithic Applications into Microservices,” in *2020 IEEE 14th International Conference on Application of Information and Communication Technologies (AICT)*, Tashkent, Uzbekistan: IEEE, Oct. 2020, pp. 1–4. doi: 10.1109/AICT50176.2020.9368571.

[57] F. Muraca and M. F. Pollo-Cattaneo, “Migration from Monolithic Applications to Microservices: A Systematic Literature Mapping on Approaches, Challenges, and Anti-patterns”.

[58] G. Mazlami, J. Cito, and P. Leitner, “Extraction of Microservices from Monolithic Software Architectures,” in *2017 IEEE International Conference on Web Services (ICWS)*, Honolulu, HI, USA: IEEE, Jun. 2017, pp. 524–531. doi: 10.1109/ICWS.2017.61.

[59] M. Oriol, C. Müller, J. Marco, P. Fernandez, X. Franch, and A. Ruiz-Cortés, “Comprehensive assessment of open source software ecosystem health,” *Internet Things*, vol. 22, p. 100808, Jul. 2023, doi: 10.1016/j.iot.2023.100808.

[60] A. Selmadji, A.-D. Seriai, H. L. Bouziane, R. Oumarou Mahamane, P. Zaragoza, and C. Dony, “From Monolithic Architecture Style to Microservice one Based on a Semi-Automatic Approach,” in *2020 IEEE International Conference on Software Architecture (ICSA)*, Salvador, Brazil: IEEE, Mar. 2020, pp. 157–168. doi: 10.1109/ICSA47634.2020.00023.

[61] D. Taibi and K. Systä, “A Decomposition and Metric-Based Evaluation Framework for Microservices,” 2020, pp. 133–149. doi: 10.1007/978-3-030-49432-2_7.

[62] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, “From Monolithic to Microservices: An Experience Report from the Banking Domain,” *IEEE Softw.*, vol. 35, no. 3, pp. 50–55, May 2018, doi: 10.1109/MS.2018.2141026.

[63] H. Zhang, S. Li, Z. Jia, C. Zhong, and C. Zhang, “Microservice Architecture in Reality: An Industrial Inquiry,” in *2019 IEEE International Conference on Software Architecture (ICSA)*, Mar. 2019, pp. 51–60. doi: 10.1109/ICSA.2019.00014.

[64] T. Ueda, T. Nakaike, and M. Ohara, “Workload characterization for microservices,” in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2016, pp. 1–10. doi: 10.1109/IISWC.2016.7581269.

[65] U. Zdun, E. Wittern, and P. Leitner, “Emerging Trends, Challenges, and Experiences in DevOps and Microservice APIs,” *IEEE Softw.*, vol. 37, no. 01, pp. 87–91, Jan. 2020, doi: 10.1109/MS.2019.2947982.

[66] D. Shadija, M. Rezai, and R. Hill, “Towards an understanding of microservices,” in *2017 23rd International Conference on Automation and Computing (ICAC)*, Huddersfield, United Kingdom: IEEE, Sep. 2017, pp. 1–6. doi: 10.23919/IConAC.2017.8082018.

[67] H. Dinh Tuan, M. Mora, F. Beierle, and S. Garzon, “Development Frameworks for Microservice-based Applications: Evaluation and Comparison,” Oct. 2020. doi: 10.1145/3393822.3432339.

[68] J. Park, D. Kim, and K. Yeom, “An Approach for Reconstructing Applications to Develop Container-Based Microservices,” *Mob. Inf. Syst.*, vol. 2020, p. e4295937, Jan. 2020, doi: 10.1155/2020/4295937.

[69] N. Borhan, H. Zulzalil, N. Mohd Ali, and S. Hassan, “Requirements Prioritization Techniques Focusing on Agile Software Development: A Systematic Literature Review,” *Int. J. Sci. Technol. Res.*, vol. 8, pp. 2118–2125, Nov. 2019.

[70] M. Wang, “Service Integration Design Patterns in Microservices”.

[71] W. Fan, Z. Han, Y. Zhang, and R. Wang, “Method of Maintaining Data Consistency in Microservice Architecture,” May 2018, pp. 47–50. doi: 10.1109/BDS/HPSC/IDS18.2018.00023.

[72] V. Basavegowda Ramu, “Performance Impact of Microservices Architecture,” *Rev. Contemp. Sci. Acad. Stud.*, vol. 3, Jun. 2023, doi: 10.55454/rcsas.3.06.2023.010.

[73] D. Taibi, V. Lenarduzzi, and C. Pahl, “Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation,” *IEEE Cloud Comput.*, vol. 4, no. 5, pp. 22–32, Sep. 2017, doi: 10.1109/MCC.2017.4250931.

[74] A. Koschel, I. Astrova, and J. Dötterl, “Making the move to microservice architecture,” in *2017 International Conference on Information Society (i-Society)*, Jul. 2017, pp. 74–79. doi: 10.23919/i-Society.2017.8354675.

[75] H. Michael Ayas, P. Leitner, and R. Hebig, “An empirical study of the systemic and technical migration towards microservices,” *Empir. Softw. Eng.*, vol. 28, no. 4, p. 85, May 2023, doi: 10.1007/s10664-023-10308-9.

[76] E. Foster, “A COMPARITIVE ANALYSIS OF THE C++, JAVA, AND PYTHON LANGUAGES,” Dec. 2014.

[77] L. Potts, *DOWNLOAD Hands On Microservices with Rust Build test and deploy scalable and reactive microservices with Rust*. Accessed: May 04, 2024. [Online]. Available: https://www.academia.edu/98884822/DOWNLOAD_Hands_On_Microservices_with_Rust_Build_test_and_deploy_scalable_and_reactive_microservices_with_Rust

[78] M. Rebouças, G. Pinto, F. Ebert, W. Torres, A. Serebrenik, and F. Castor, “An Empirical Study on the Usage of the Swift Programming Language,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Mar. 2016, pp. 634–638. doi: 10.1109/SANER.2016.66.

[79] T. K. Patra and D. S. Jain, “Comparison of JavaScript Frameworks: React.js and Vue.js,” vol. 7, no. 9, 2022.

[80] H. Suryotrisongko, D. P. Jayanto, and A. Tjahyanto, “Design and Development of Backend Application for Public Complaint Systems Using Microservice Spring Boot,” *Procedia Comput. Sci.*, vol. 124, pp. 736–743, 2017, doi: 10.1016/j.procs.2017.12.212.

[81] A. C. H. Chen, “Research on Efficiency Analysis of Microservices,” in *2023 IEEE International Conference on Machine Learning and Applied Network Technologies (ICMLANT)*, Dec. 2023, pp. 1–5. doi: 10.1109/ICMLANT59547.2023.10372984.

[82] R. Pontarolli, J. Bigheti, F. Domingues, L. Sá, and E. Godoy, “Distributed I/O as a service: A data acquisition solution to Industry 4.0,” *HardwareX*, vol. 12, p. e00355, Sep. 2022, doi: 10.1016/j.ohx.2022.e00355.

[83] J. Rasheedh and D. Saradha, “Reactive Microservices Architecture Using a Framework of Fault Tolerance Mechanisms,” Aug. 2021, pp. 146–150. doi: 10.1109/ICESC51422.2021.9532893.

[84] D. Taibi, V. Lenarduzzi, and C. Pahl, “Architectural Patterns for Microservices: A Systematic Mapping Study,” Mar. 2018. doi: 10.5220/0006798302210232.

[85] H. Vural and M. Koyuncu, “Does Domain-Driven Design Lead to Finding the Optimal Modularity of a Microservice?,” *IEEE Access*, vol. PP, pp. 1–1, Feb. 2021, doi: 10.1109/ACCESS.2021.3060895.

[86] M. Štefanko, O. Chaloupka, and B. Rossi, “The Saga Pattern in a Reactive Microservices Environment,” Jan. 2019, pp. 483–490. doi: 10.5220/0007918704830490.

[87] J. Daniel, X. Wang, and E. Guerra, “How to design Future-Ready Microservices? Analyzing microservice patterns for Adaptability,” in *Proceedings of the 28th European Conference on Pattern Languages of Programs*, Irsee Germany: ACM, Jul. 2023, pp. 1–7. doi: 10.1145/3628034.3628046.

[88] E. Ntentos, U. Zdun, K. Plakidas, D. Schall, F. Li, and S. Meixner, “Supporting Architectural Decision Making on Data Management in Microservice Architectures,” in *Software Architecture*, vol. 11681, T. Bures, L. Duchien, and P. Inverardi, Eds., in Lecture Notes in Computer Science, vol. 11681. , Cham: Springer International Publishing, 2019, pp. 20–36. doi: 10.1007/978-3-030-29983-5_2.

[89] A. Katal, P. Prasanna, R. Birla, and Kunal, “Evolution from Monolithic to Microservices Architecture: A New Era in Software Architecture,” in *Advancements in Optimization and Nature-Inspired Computing for Solutions in Contemporary Engineering Challenges*, D. Rossit, C. E. Torres-Aguilar, and A. A. Toncovich, Eds., Singapore: Springer Nature, 2025, pp. 235–279. doi: 10.1007/978-981-96-0706-8_12.

[90] D. Narváez, N. Battaglia, A. Fernández, and G. Rossi, “Designing Microservices Using AI: A Systematic Literature Review,” *Software*, vol. 4, no. 1, Art. no. 1, Mar. 2025, doi: 10.3390/software4010006.

[91] M. Söylemez, B. Tekinerdogan, and A. Kolukısa, “Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review,” *Appl. Sci.*, vol. 12, p. 5507, May 2022, doi: 10.3390/app12115507.

[92] B. Erdenebat, B. Bud, T. Batsuren, and T. Kozsik, “Multi-Project Multi-Environment Approach—An Enhancement to Existing DevOps and Continuous Integration and Continuous Deployment Tools,” *Computers*, vol. 12, no. 12, Art. no. 12, Dec. 2023, doi: 10.3390/computers12120254.

[93] B. Shafabakhsh, R. Lagerström, and S. Hacks, “Evaluating the Impact of Inter Process Communication in Microservice Architectures,” p. 9, 2020.

[94] A. Owen, “Microservices Architecture and API Management: A Comprehensive Study of Integration, Scalability, and Best Practices,” Jan. 2025.

[95] M. Gördesli and A. Varol, “Comparing Interservice Communications of Microservices for E-Commerce Industry,” in *2022 10th International Symposium on Digital Forensics and Security (ISDFS)*, Istanbul, Turkey: IEEE, Jun. 2022, pp. 1–4. doi: 10.1109/ISDFS55398.2022.9800784.

[96] R. Wu, Q. Duan, F. Dai, H. Yang, Y. Zhang, and B. Xie, “Research on the Realizability of Microservice Interaction Contract Based on CSP#,” in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, Jul. 2019, pp. 622–627. doi: 10.1109/COMPSAC.2019.10277.

[97] S. A. Asri, I. N. G. A. Astawa, I. G. A. M. Sunaya, I. M. R. Adi Nugroho, and W. Setiawan, “Implementation of Asynchronous Microservices Architecture on Smart Village Application,” *Int. J. Adv. Sci. Eng. Inf. Technol.*, vol. 12, no. 3, p. 1236, Jun. 2022, doi: 10.18517/ijaseit.12.3.13897.

[98] D. Saxena, W. Zhang, M. Tummala, S. Goel, and A. Akella, “Invited Paper: Towards Efficient Microservice Communication,” in *Proceedings of the 5th*

workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems, Orlando FL USA: ACM, Jun. 2023, pp. 1–5. doi: 10.1145/3584684.3597267.

[99] J. Kazanavičius and D. Mazeika, "Evaluation of Microservice Communication While Decomposing Monoliths," *Comput. Inform.*, vol. 42, pp. 1–36, May 2023, doi: 10.31577/cai_2023_1_1.

[100] M. Bolanowski *et al.*, *Eficiency of REST and gRPC realizing communication tasks in microservice-based ecosystems*. 2022. doi: 10.48550/arXiv.2208.00682.

[101] P. Kumar, R. Agarwal, R. Shivaprasad, D. Sitaram, and K. V. Subramaniam, "Performance Characterization of Communication Protocols in Microservice Applications," Sep. 2021, pp. 1–5. doi: 10.1109/SmartNets50376.2021.9555425.

[102] L. Qigang and X. Sun, "Research of Web Real-Time Communication Based on Web Socket," *Int. J. Commun. Netw. Syst. Sci.*, vol. 05, pp. 797–801, Jan. 2012, doi: 10.4236/ijcns.2012.512083.

[103] M. Lim, "Directly and Indirectly Synchronous Communication Mechanisms for Client-Server Systems Using Event-Based Asynchronous Communication Framework," *IEEE Access*, vol. 7, pp. 81969–81982, 2019, doi: 10.1109/ACCESS.2019.2924497.

[104] F. Cristian, "Synchronous and Asynchronous Group Communication (long Version)," 1996.

[105] A. Vrincean, "Optimizing request handling using blocking & non-blocking I/O middleware," 2021. doi: 10.13140/RG.2.2.31709.13282.

[106] M. E. Ekpenyong and P. J. Udoh, "Modeling the Effect of Bandwidth Allocation on Network Performance," *Sci. World J.*, vol. 9, no. 4, Art. no. 4, 2014.

[107] A. Alraddadi, "EVALUATION OF PACKET LOSS EFFECT ON NETWORK PERFORMANCE".

[108] C. Rodríguez-Domínguez, K. Benghazi, M. Noguera, J. Garrido, M. Rodríguez, and T. Ruiz-López, "A Communication Model to Integrate the Request-Response and the Publish-Subscribe Paradigms into Ubiquitous Systems," *Sensors*, vol. 12, pp. 7648–68, Dec. 2012, doi: 10.3390/s120607648.

[109] C. Bayılmış, M. A. Ebleme, Ü. Çavuşoğlu, K. Küçük, and A. Sevin, "A survey on communication protocols and performance evaluations for Internet of Things," *Digit. Commun. Netw.*, vol. 8, no. 6, pp. 1094–1104, Dec. 2022, doi: 10.1016/j.dcan.2022.03.013.

[110] O. Ozkasap, "SCALABILITY, THROUGHPUT STABILITY AND EFFICIENT BUFFERING IN RELIABLE MULTICAST PROTOCOLS°".

[111] Y. Li, D. Li, W. Cui, and R. Zhang, "Research based on OSI model," in *2011 IEEE 3rd International Conference on Communication Software and Networks*, May 2011, pp. 554–557. doi: 10.1109/ICCSN.2011.6014631.

[112] K. Berlin *et al.*, "Evaluating the Impact of Programming Language Features on the Performance of Parallel Applications on Cluster Architectures," May 2004, pp. 194–208. doi: 10.1007/978-3-540-24644-2_13.

[113] L. J. Gonçales, K. Farias, and B. C. da Silva, “Measuring the cognitive load of software developers: An extended Systematic Mapping Study,” *Inf. Softw. Technol.*, vol. 136, p. 106563, Aug. 2021, doi: 10.1016/j.infsof.2021.106563.

[114] S. Montagud, S. Abrahão, and E. Insfran, “A systematic review of quality attributes and measures for software product lines,” *Softw. Qual. J. - SQJ*, vol. 20, Sep. 2012, doi: 10.1007/s11219-011-9146-7.

[115] A. Chalker, C. W. Hillegas, A. Sill, S. Broude Geva, and C. A. Stewart, “Cloud and on-premises data center usage, expenditures, and approaches to return on investment: A survey of academic research computing organizations,” in *Practice and Experience in Advanced Research Computing*, in PEARC ’20. New York, NY, USA: Association for Computing Machinery, Jul. 2020, pp. 26–33. doi: 10.1145/3311790.3396642.

[116] B. B. Rad, T. Diaby, and M. E. Rana, “Cloud Computing Adoption: A Short Review of Issues and Challenges,” in *Proceedings of the 1st International Conference on E-commerce, E-Business and E-Government*, in ICEEG ’17. New York, NY, USA: Association for Computing Machinery, Jun. 2017, pp. 51–55. doi: 10.1145/3108421.3108426.

[117] D. Rani and R. K. Ranjan, “A Comparative Study of SaaS, PaaS and IaaS in Cloud Computing,” *Int. J. Adv. Res. Comput. Sci. Softw. Eng.*, p. 4, 2014.

[118] “Cloud Computing Services - Amazon Web Services (AWS).” Accessed: May 01, 2024. [Online]. Available: https://aws.amazon.com/

[119] “Cloud Computing Services | Google Cloud.” Accessed: May 01, 2024. [Online]. Available: https://cloud.google.com/

[120] “Cloud Computing Services | Microsoft Azure.” Accessed: May 01, 2024. [Online]. Available: https://azure.microsoft.com/en-us

[121] R. Padhy and M. Patra, “Evolution of Cloud Computing and Enabling Technologies,” *Int. J. Cloud Comput. Serv. Sci. IJ-CLOSER*, vol. 1, Oct. 2012, doi: 10.11591/closer.v1i4.1216.

[122] S. Goyal, “Public vs Private vs Hybrid vs Community - Cloud Computing: A Critical Review,” *Int. J. Comput. Netw. Inf. Secur.*, vol. 6, no. 3, pp. 20–29, Feb. 2014, doi: 10.5815/ijcnis.2014.03.03.

[123] B. Ampofo, M. Shrivas, and S. Singh, “Public Cloud Computing; Benefits and Risks to Users and Potential Users,” Sep. 2016.

[124] V. Reddy *et al.*, “Research Issues in Cloud Computing Research Issues in Cloud Computing Research Issues in Cloud Computing,” *Glob. J. Comput. Sci. Technol. 0975-4172*, vol. 11, p. 59, Jul. 2011.

[125] G. Kulkarni, N. Patil, and P. Patil, “Private Cloud Secure Computing,” *Int. J. Soft Comput. Eng. IJSCE*, vol. 2231–2307, pp. 2231–2307, Apr. 2012.

[126] P. Abdalla and A. Varol, “Advantages to Disadvantages of Cloud Computing for Small-Sized Business,” Jun. 2019, pp. 1–6. doi: 10.1109/ISDFS.2019.8757549.

[127] M. Deb and A. Choudhury, “Hybrid Cloud: A New Paradigm in Cloud Computing,” 2021, pp. 1–23. doi: 10.1002/9781119764113.ch1.

[128] G. Aryotejo, D. Y. Kristiyanto, and Mufadhol, “Hybrid cloud: bridging of private and public cloud computing,” *J. Phys. Conf. Ser.*, vol. 1025, no. 1, p. 012091, May 2018, doi: 10.1088/1742-6596/1025/1/012091.

[129] H. Nicanfar, Q. Liu, P. Talebifard, and W. Cai, “Community Cloud: Concept, Model, Attacks and Solution,” presented at the Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom, Dec. 2013, pp. 126–131. doi: 10.1109/CloudCom.2013.163.

[130] A. Sungkar and T. Kogoya, “A REVIEW OF GRID COMPUTING,” *Comput. Sci. IT Res. J.*, vol. 1, pp. 1–6, Apr. 2020, doi: 10.51594/csitrj.v1i1.128.

[131] M. U. Bokhari, Q. M. Shallal, and Y. K. Tamandani, “Cloud computing service models: A comparative study,” in *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, Mar. 2016, pp. 890–895. Accessed: May 04, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/7724392

[132] A. Mehta and D. S. N. Panda, “Design of Infrastructure as a Service (IAAS) Framework with Report Generation Mechanism,” *Int. J. Appl. Eng. Res.*, vol. 13, no. 2, p. 5, 2018.

[133] P. Chavan and G. Kulkarni, “PaaS Cloud,” *Int. J. Comput. Sci. Inf. Secur. IJCSIS*, vol. 1, pp. 21–26, Oct. 2013.

[134] D. T and G. R, “Platform-as-a-Service (PaaS): Model and Security Issues,” *TELKOMNIKA Indones. J. Electr. Eng.*, vol. 15, no. 1, Jul. 2015, doi: 10.11591/telkomnika.v15i1.8073.

[135] T.-H. Al-Madhagy, A. Alanzi, S. Mohd Yusof, and M. Alruwaili, “Software as a Service (SaaS) Cloud Computing: An Empirical Investigation on University Students’ Perception,” *Interdiscip. J. Inf. Knowl. Manag.*, vol. 16, pp. 213–253, Jan. 2021, doi: 10.28945/4740.

[136] Y. Sun, “Cloud and Edge Computing as Effective Trends in Business Model Innovation: A Bibliometric Review,” *J. Softw. Evol. Process*, vol. 37, no. 1, p. e2754, 2025, doi: 10.1002/smr.2754.

[137] S. Deng *et al.*, *Cloud-Native Computing: A Survey from the Perspective of Services*. 2023. doi: 10.36227/techrxiv.23500383.

[138] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, “Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned,” in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, Jul. 2018, pp. 970–973. doi: 10.1109/CLOUD.2018.00148.

[139] J. Alonso *et al.*, “Understanding the challenges and novel architectural models of multi-cloud native applications – a systematic literature review,” *J. Cloud Comput.*, vol. 12, no. 1, p. 6, Jan. 2023, doi: 10.1186/s13677-022-00367-6.

[140] L. Chen, “Microservices: Architecting for Continuous Delivery and DevOps,” Mar. 2018. doi: 10.1109/ICSA.2018.00013.

[141] M. Al-Ameen and J. Spillner, “A systematic and open exploration of FaaS research,” presented at the ESSCA, Zurich, 21 December 2018, CEUR-WS, 2019, pp. 30–35. doi: 10.21256/zhaw-3271.

[142] R. A. P. Rajan, “Serverless Architecture - A Revolution in Cloud Computing,” IEEE, Dec. 2018, pp. 88–93. doi: 10.1109/ICoAC44903.2018.8939081.

[143] A. Liberati *et al.*, “The PRISMA Statement for Reporting Systematic Reviews and Meta-Analyses of Studies That Evaluate Health Care Interventions: Explanation and Elaboration,” *J. Clin. Epidemiol.*, vol. 62, pp. e1-34, Aug. 2009, doi: 10.1016/j.jclinepi.2009.06.006.

[144] A. M, M. V. Thomas, and S. Pai, “A Hybrid Communication Protocol for IoT Applications in Smart Domain,” in *2021 International Conference on Communication, Control and Information Sciences (ICCISc)*, Jun. 2021, pp. 1–6. doi: 10.1109/ICCISc52257.2021.9484992.

[145] J. Thomsen, “Abstraction: Simplifying Complexity in Software Engineering,” *Am. J. Comput. Sci. Inf. Technol.*, vol. 11, no. 4, Apr. 2023, Accessed: Aug. 17, 2024. [Online]. Available: https://www.imedpub.com/articles/abstraction-simplifying-complexity-in-software-engineering.php?aid=50800

[146] K. Evensen, A. Petlund, C. Griwodz, and P. Halvorsen, “Redundant bundling in TCP to reduce perceived latency for time-dependent thin streams,” *Commun. Lett. IEEE*, vol. 12, pp. 324–326, May 2008, doi: 10.1109/LCOMM.2008.071957.

[147] S. Tallberg, “A COMPARISON OF DATA INGESTION PLATFORMS IN REAL-TIME STREAM PROCESSING PIPELINES,” p. 39.

[148] S. Barakat, “Monitoring and Analysis of Microservices Performance,” *J. Comput. Sci. Control Syst.*, vol. 10, pp. 19–22, May 2017.

[149] R. B. Khan, “Comparative Study of Performance Testing Tools: Apache JMeter and HP LoadRunner,” p. 57.

[150] N. Hamo and S. Saberian, “Evaluating the performance and usability of HTTP vs gRPC in communication between microservices”.

[151] Ł. Kamiński, M. Kozłowski, D. Sporysz, K. Wolska, P. Zaniewski, and R. Roszczyk, “Comparative review of selected Internet communication protocols,” Dec. 14, 2022, *arXiv*: arXiv:2212.07475. doi: 10.48550/arXiv.2212.07475.