Evaluation of Decomposition to Microservices with Optimized Inter-Service Communication Strategy

L.D.S.B Weerasinghe, Indika Perera

Abstract— **During the early stages of software development, monolithic architectures predominated within software systems. However, their limitations in effectively accommodating dynamic user demands prompted the widespread adoption of microservices architecture. Microservices offer solutions by anticipating changes in user needs and technological advancements. Most enterprise-grade software is now deployed in a cloud-native environment and transitioning towards a microservice-based architecture. This study investigates how inter-service communication affects the performance of the overall microservices architecture. In order to find the optimal communication segments, the researchers have conducted several experiments. The research experiments also considered software quality attributes such as maintainability, scalability, portability, testability, and reliability. The evaluations' findings indicate that a discernible decrease in overall system performance occurs when more than six microservices are communicated to fulfil user requirements. This emphasizes how crucial it is for architects to consider communication patterns when designing or transitioning to microservices prudently. The findings and insights from this study are anticipated to contribute valuable knowledge and guidance for future advancements in microservices developments.**

Index Terms—**Microservices, Inter-service communication, Performance, Reference architecture**

I. Introduction

In the '90s and early '20s, most enterprise-grade software followed monolithic architecture. Back in the day, the user requirements were found to be very simple and not subject to constant change. Additionally, change requests did not come through often, and software maintainability was much easier for development and support teams. However, user requirements became complex over time, requiring frequent changes. This complexity resulted in the need to change software to accommodate those changes in requirements. However, the behaviour and the structure of the monolithic software development do not allow further changes to the system.

Manuscript received April 3, 2024; revised January 31, 2025

L.D.S.B Weerasinghe is a postgraduate student of the Department of Computer Science & Engineering, University of Moratuwa, Sri Lanka. (e-mail: weerasingheldsb.20@uom.lk).

Indika Perera is a professor at the Department of Computer Science & Engineering, University of Moratuwa, Sri Lanka (e-mail: indika@cse.mrt.ac.lk).

The Service-oriented architecture (SOA) was introduced to mitigate this obstacle, assisting in the system's design by separating concepts [1]. The monolithic-based system moved towards service-oriented architecture, using that architecture to implement changes in user requirements. All the services in the SOA system are orchestrated by an Enterprise Service Bus (ESB) [2]. When the overall system traffic is getting high traffic, the service orchestration becomes a bottleneck and decreases the application performance. Microservice architecture has been introduced to the world as a solution to the possible problems when using software architecture.

The microservices paradigm has emerged as a transformative approach in the software architecture landscape, offering a migration from traditional monolithic architecture. With this transformation, most services migrated to cloud-based deployments [3]. Microservices represent a service and distributed architectural style where applications are composed of independently deployable and scalable services. Unlike monolithic architectures, where the entire application is a single, tightly integrated unit, microservices advocate breaking down complex systems into smaller, self-contained services. Each microservice operates as a discrete entity, communicating with others through well-defined interfaces. This promotes agility, as changes to one service do not necessitate modifications to the entire application. Hence, the developers can introduce new requirement changes to the software without stirring the entire application. It increases the software's maintainability and helps all teams. Microservice decentralisation fosters a service development process where services can be written in different programming languages per technological advancements, use diverse data storage solutions, and evolve at their own pace. Such autonomy facilitates parallel development, leading to faster innovation cycles for the engineering teams [4]. Moreover, adopting microservices necessitates a cultural shift, with teams embracing new practices such as DevOps and continuous integration [5]. Microservices enable scalable and resilient systems. Each service can be independently scaled based on demand, and failures in one service do not necessarily compromise the integrity of the entire application. This architecture improves fault isolation, identifying and addressing problems without affecting the overall system.

Despite the advantages, implementing microservices comes with its set of challenges. Effective communication is crucial in a microservices architecture. Microservices communicate through various patterns, including synchronous HTTP/REST APIs, asynchronous messaging, and event-driven architectures. The choice of

communication pattern depends on the specific requirements of the system and the nature of the tasks that each microservice performs. This distributed communication model allows flexibility and resilience, ensuring that the failure of one microservice does not cascade throughout the entire application. Ensuring the security of inter-service communication is a top priority in microservices architecture. Implementing robust authentication, encryption, and authorisation mechanisms safeguards sensitive data between services. The landscape of inter-service communication is dynamic, with continuous advancements in technologies and practices. Striking a balance between low-latency communication and fault tolerance is a complex undertaking that demands thoughtful microservice architectural design.

Transitioning from monolithic architectures to microservices involves a strategic approach to ensure a seamless and effective decomposition. The foremost step in decomposition is to identify logical boundaries for microservices. This involves analysing the existing monolith to pinpoint distinct business functionalities or modules that can operate independently. The goal is to define services encapsulating specific concerns, minimising dependencies between them. Leveraging principles of Domain-Driven Design aids in creating a shared understanding of the business domain and informs the delineation of microservices [6]. The main point is how much microservices need to interact in order to meet the user's requirements. Unlike in the monolithic architecture, those microservice interactions must pass the data through the network. Each communication interaction brings an additional latency to the application. This research study mainly focuses on how inter-service communication impacts microservice-based applications.

The primary challenge encountered during the transition from a monolithic to a microservices architecture is the performance issue arising from microservices' independent and distributed nature. Unlike in monolithic architectures, where all functionalities are contained within a single server, microservices require communication between multiple servers to generate output, leading to potential performance bottlenecks. This research primarily focuses on evaluating the impact of inter-service communication during the decomposition of monolithic systems to microservices architecture. By delving into these aspects, the research aims to provide insights into how architects and developers can effectively manage inter-service communication to optimize the decomposition process and enhance the overall performance of microservices-based systems. Optimized inter-service communication based on TCP streams is identified as a critical factor contributing to enhanced performance in microservices architectural software regarding response time and throughput. Architects need to optimise communication strategies and consider the number of communication pathways required to meet user requirements. The research evaluation emphasizes the efficacy of decomposing microservices from monolithic systems, indicating the potential for attaining optimal performance within a microservices architecture.

II. Literature Review

A. Background

Microservice architecture has become widely popular in software architecture patterns in recent years. In the earlier days, most of the systems were built as monolithic-based architecture since they did not have complex user requirements or many change requests for the software [7]. With the user requirements becoming complicated, the engineers could not anticipate the requirement changes in the software with the assistance of monolithic software architecture. Service-oriented architecture (SOA) was introduced in software development to address the concerns of monolithic architecture [8]. In this architecture, services are segregated as components and are orchestrated by the Enterprise Service Bus (ESB). Research has shown that ESB in service-oriented architecture becomes a bottleneck; hence, it will cause performance degradation of SOA architecture [2]. Many vendors are offering ESB using different technologies. However, according to the SOA architecture, the service component calls passed through the ESB will cause a performance impact. Nevertheless, SOA architecture-based reference architectures perform better than monolithic-based systems. The same research statistically proves that microservice architecture performs better than SOA-based software systems. A systematic review of microservices has been conducted using the PRISMA model, showing that most microservice-related researches were conducted from 2015 onwards [9]. Most software engineers seek quality attributes such as maintainability, performance, security, cloud support, observability, etc. [10]. People are moving towards microservices to achieve software quality attributes quickly. Ample software frameworks are available in different programming languages to support building the microservice with quality attributes. Spring Boot and Vert.X frameworks for Java language, Go Micro framework for Go language, and Molecular for Node.js language are a few examples of microservice development frameworks [11]. Microservices are deployed in the distributed environment, and inter-service communication brings in additional latency for the inter-service communication, which results in a performance impact on the overall system response time and throughput. An optimized strategy for inter-service communication has been introduced from a research study for microservice architecture [12]. TCP-based stream communication is enabled as a request/response method to communicate with internal microservices. Redis server helps guarantee the exact delivery of the message so that reliability and quality attributes will be preserved [13]. Academic and industrial testing has statistically proven that the introduced strategy performed better than the traditional HTTP methods. Most microservices are now deployed as containers, yet some continue using on-premises server deployments [14]. The proper reference architecture will help bring this solution to the enterprise level for both clouds and on-premises data centres [15].

B. Others Work

Most of the research was on monolithic to microservice conversion strategies and their technologies. Florian Auer and team surveyed the monolithic to microservice conversion by interviewing industry professionals with the software [11]. Most architects move from monolithic architecture to microservices architecture to achieve maintainability, scalability, and cost [16]. Most people also consider metrics like response time, resource utilization, complexity, and functional suitability before migration. By moving to microservice, software and the organization need to be reorganized. A group of researchers in Finland compared technical debt before and after the monolithic to microservice migration of the 12-year-old monolithic software, which has 280k lines of code [17]. They have used the SonarQube software to perform qualitative studies, and the results have shown a reduced technical debt in the long run when moving to microservice architecture [18]. However, during the migration period, development activities are prolonged because of the adaptation and planning of new technology. Fujitsu Laboratories Ltd extracted the decomposed candidates from the monolithic applications code using the SarF algorithm [19]. They have conducted a case study using a small application and showed single microservices that can be decomposed from the existing system. However, their research needed to improve execution timing and the business domain segregation. Victor Velepucha and the team researched the problems and challenges of migrating from monolithic to microservices[20]. They have highlighted that no single theory exists when migrating from a monolithic application to microservices. No such toolkit to use for the migration, the need to reorganize the development team from scratch, problems identifying the microservices, issues with consistency, and the need to wait until the entire application is migrated are some of the problems they have identified in the study [21]. The challenges researchers identified during the study include selecting a suitable framework, changing the hierarchical structure, and prioritizing the requirements.

III. Methodology

This research study presents a comprehensive methodology to derive the impact on microservice communication when migrating to the microservices from the monolithic software architecture. At the design stage of microservices, architects are tasked with considering the comprehensive spectrum of system functional requirements, non-functional requirements, and software quality attributes. Microservice decomposing includes the below steps,

- Understanding the context – When designing the microservices or migrating the system from monolithic architecture to microservices architecture, one needs to understand the organisation's goal, business objectives, and challenges that the microservice architecture should address.
- Identify the responsibilities and define services – Breakdown the system into manageable and independent components by identifying the business requirements. Services should have well-defined responsibilities. This can be based on the principles of domain-driven design (DDD) [22]. Those components can be mapped to microservices in the software architecture.
- Data model – Choose which data is stored in the microservices or what data needs to be communicated with each microservice. Data storage can either be a relational database or a non-relational database. Or else it can be a volatile memory. As this is a crucial step, architects must make a solid decision to maintain the entire system's performance [23].
- API design – Define the straightforward API needed by the external parties, design communication interfaces, and determine how internal microservices should be communicated to each other.
- Technology selection - Choose appropriate technology stacks for each microservice based on its requirements. Consider programming languages, frameworks, databases, communication protocols, people expert areas, and the time required to complete the project.
- Deployment methodology – Decide whether to use the cloud or on-premise environment, based on the environment availability and determine whether to use containerisation or orchestration to manage microservices.
- Testing strategies – Decide on developing comprehensive testing strategies, including unit testing, integration testing, and end-to-end testing [24]. Implement Continuous Integration and Continuous Deployment (CI/CD) pipelines for automated testing and deployment [25].
- Evolution and maintenance - Be prepared to iterate on microservices architecture as a requirement change. Keep architecture flexible and maintainable to accommodate evolving business needs.

The process begins with a deep understanding of business objectives, paving the way for identifying independent services aligned with domain-driven design principles. The meticulous consideration of data models, API designs, and technology stacks ensures the coherent functioning of microservices. Architects must make informed decisions on data storage, communication interfaces, and technology selections to optimize system performance. Deployment methodologies, testing strategies, and recognizing the need for evolutionary flexibility round out the methodology. As microservices architecture inherently accommodates change, architects should remain adaptable, fostering an environment that allows for iterative evolution and maintenance, ensuring the sustained alignment of the architecture with dynamic business requirements. When considering the above points, architects can decompose the monolithic system into microservices.

A. Inter-service Communication

Intercommunication lies at the heart of microservices architecture and is pivotal in shaping distributed systems' performance, scalability, and overall functionality [26]. The significance of microservice communication stems from several key aspects that contribute to the success and

efficiency of this architectural paradigm. By decomposing the microservices and establishing clear communication paths, this approach ensures that each microservice can contribute to fulfilling a single user requirement effectively. However, system designers need a proper idea of the inter-service communication paths to meet the user's requirements.

The diagram below (Fig. 1) shows how microservices communicate in a worst-case scenario. Each microservice calls to each other to produce the output of that service.

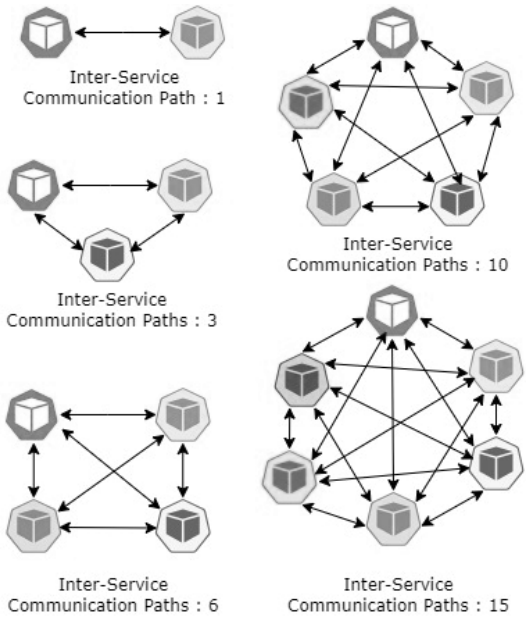

Fig. 1. Worst-case inter-service communication pattern

In this intricate communication pattern, each microservice is intricately linked to every other service in a one-to-one relationship, creating an intricate mesh of dependencies. While such direct communication paths may facilitate data exchange and collaboration between services, they inherently raise concerns about system complexity, maintainability, and scalability. The challenge with this communication approach lies in the potential for increased coupling between microservices, where changes in one service may have a cascading effect, necessitating modifications in numerous interconnected services. This can result in challenges related to versioning, code maintenance, and the overall agility of the system. Moreover, the extensive inter-service calls in this worst-case scenario may introduce latency and performance bottlenecks. The need for each microservice to communicate directly with others may lead to higher network traffic and increased response times, potentially impacting the overall system's responsiveness. The calculation of communication paths in the worst-case scenario depicted in the microservices architecture can be approached through graph theory. In this context, the number of communication paths corresponds to the potential connections between individual microservices. Considering a set of microservices as nodes and direct communication links between them as edges, the number of communication paths can be determined using the formula for combinations in graph theory. The equation for calculating the number of communication paths y in a worst-case scenario with x microservices is denoted by:

$$y = (x * (x - 1)) / 2 \quad (1)$$

In this formula:

- x represents the total number of microservices.
- (x−1) denotes the number of potential communication partners for each microservice.
- The division by 2 accounts for avoiding double-counting, as the communication path from microservice A to B is the same as from B to A.

By utilizing this formula, architects and system designers can quantitatively assess the complexity introduced by the direct communication paths in the microservices architecture's worst-case scenario. Below is a simple pseudocode snippet for calculating the number of communication paths in the worst-case scenario.

TABLE I
PSEUDOCODE FOR A WORST-CASE SCENARIO

```
//Function
function calculateCommunicationPaths(x):
   // n is the total number of microservices

   // Ensure n is a positive integer
   if n <= 0 or not isInteger(x):
      return "Invalid input. Please provide a positive integer for the number of microservices."

   // Calculate the number of communication paths(y)
   paths = (x * (x- 1)) / 2

   return paths

// Helper function to check if a number is an integer
function isInteger(num):
   return num == floor(num)
```

```
// Example usage:
numberOfMicroservices = 5
result = calculateCommunicationPaths(numberOfMicroservices)
print("Number of Communication Paths:", result)
```

This function, called calculateCommunicationPaths, takes a single parameter x, representing a system's total number of microservices. It first checks whether x is a positive integer. If x is not a positive integer, or if it's not a whole number (i.e., not an integer), the function returns an error message indicating that the input is invalid. Assuming x passes this validation, the function calculates the number of communication paths required among the microservices. It does so using a mathematical formula where each microservice needs to communicate with every other microservice except itself. The calculated number of communication paths is then returned as the output. Additionally, a helper function named isInteger is used within calculateCommunicationPaths to check if a given number is an integer, ensuring the validity of the input. Overall, this function aids in determining the optimal communication structure needed for a system with a specified number of microservices, contributing to efficient information flow within the architecture. This analysis aids in understanding the scale of interdependencies and potential challenges associated with managing communication within the system, providing valuable insights for optimizing the architecture for improved scalability and maintainability.

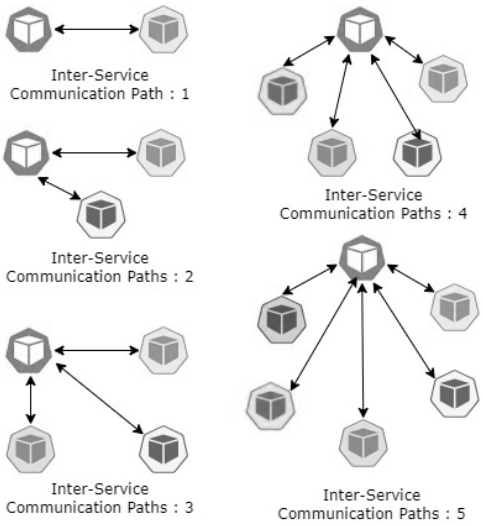

Fig. 2. Best case inter-service communication

The above diagram (Fig. 2) shows the minimal communication interactions between the microservices. This architecture reflects a simple, efficient design with direct, point-to-point interactions. Unlike more complex communication patterns, such as an entirely interconnected mesh, this minimalist approach minimizes the number of communication paths and potential dependencies between microservices. Each microservice communicates only with those directly relevant to its functionality, contributing to a clear and straightforward system. This design choice emphasizes clarity, ease of management, and reduced coupling between microservices. The streamlined communication paths enhance system maintainability and scalability, as changes to one microservice are less likely to cascade through an extensive network of interconnected services. The equation for calculating the number of communication paths y in a best-case scenario with x microservices is expressed by:

$$y = (x - 1) \tag{2}$$

This equation signifies a simplified communication pattern where each microservice communicates directly with all others except itself, resulting in a linear relationship between the number of microservices and the communication paths. Specifically, each microservice establishes direct communication links with x−1 other microservices.

If the two equations mentioned above are plotted, the number of communication paths in the worst-case scenario is growing exponentially than in the best-case scenario. Impact analysis can be found under the evaluation section.

IV. Implementation

Implementing a robust microservices architecture in a distributed environment presents considerable challenges [27]. The reference ecosystem serves as a crucial step, transforming abstract designs into concrete building blocks, thus realising the envisioned communication strategy. Each microservice can be broken down into smaller pieces that help implement the microservice better.

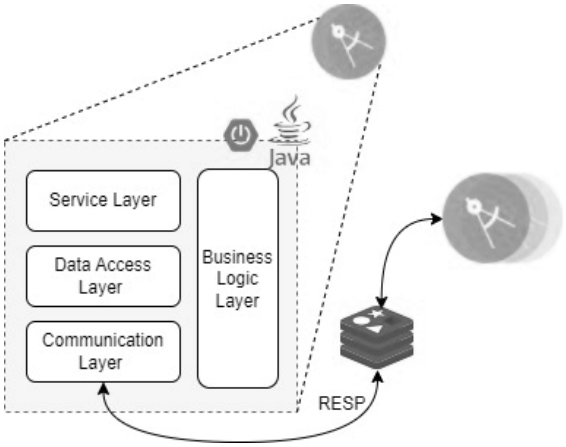

Fig. 3. Microservice component diagram

Figure 3 above illustrates the small-scale components within the microservice, encompassing distinct layers such as the service, data access, business logic, and communication layers. These components exhibit varying ownerships and depict their interactions within the microservice architecture. The service layer is responsible for having external APIs for the external systems. This layer mainly uses the generic REST-based API interface to enable a better integration layer for the microservices. The data access layer mainly interacts with the data storing services, such as relational databases and non-relational databases. The business logic layer implements all the business logic related to the functional requirements. The communication layer is responsible for interacting with other microservices or third-party systems. Java language is used as the primary programming language for developing microservices, leveraging its robust capabilities and utilizing the well-known Spring Boot framework as a microservices framework [28]. The decision to use Java brings numerous advantages, including its widespread adoption, platform independence, and extensive community support. Java's versatility enables the researchers to address diverse requirements within the microservices architecture. Its object-oriented nature facilitates modular and scalable code structures, aligning well with the principles of microservices. Additionally, Java's rich ecosystem of libraries and tools enhances development efficiency and provides comprehensive solutions to various challenges. Spring Boot simplifies the complexities associated with microservices development by offering conventions and defaults, reducing boilerplate code, and providing built-in support for essential features such as configuration management and dependency injection. Its emphasis on convention over configuration promotes a streamlined and standardized approach to microservices development.

Used newly implemented optimized inter-service communication method for microservices-to-microservices communication. Redis streams have been used, and communication is done using the Redis serialization protocol (RESP) [29]. This method allows developers to send messages to other microservices in the same way as REST-based clients. When sending the message to other microservices, the destination needs to be specified, and those messages are sent as a byte buffer. Hence, the network layer resource consumption is less than that of the other methods. When the microservice starts, it automatically creates a connection to the Redis server and persists until the

microservices die. Therefore, when sending the message to other microservice developers, there is no need to open and close the connection at the code level.

V. Results and Evaluation

Most enterprise-grade software is now moving from monolithic-based applications to microservice-based architecture, and those microservices are deployed in cloud-based environments. In order to evaluate inter-service communication in the microservices architecture, several microservices were implemented and deployed in a cloud-native Kubernetes cluster, as shown in Figure 4 [30]. The evaluation mainly focuses on how the number of inter-service communications and the allocated resources impact the overall system performance in terms of response time.

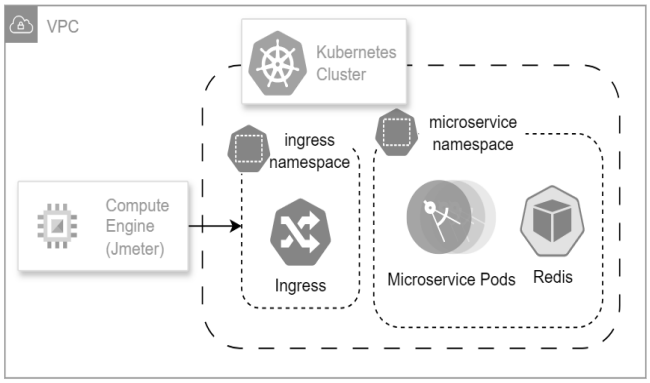

Fig. 4. Testing cloud deployment architecture

The performance of the newly introduced communication solution was evaluated and compared with the traditional and the most common HTTP protocol for inter-service communication. All microservices were developed in Java using the Spring Boot framework. Standard REST templates were used for the HTTP inter-service communication. Both microservices were deployed within the same subnet in a Kubernetes environment for testing purposes, as per figure 5.

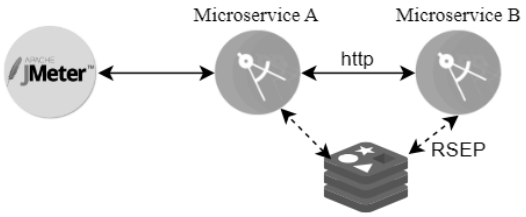

Fig. 5. Deployment for solution comparison

The turnaround time was measured by checking the logs, excluding processing time, focusing solely on the request and response times between Microservice A and Microservice B. Used URL-only GET request without body and 1KB payload size POST for both request and response for testing. Each test case was executed for 1 hour and repeated three times to account for data variations. Logs were captured using the Google Logging service and processed with Python scripts for data analysis. JMeter is installed in a separate virtual machine in the cloud and placed in the same network subnet in which the Kubernetes cluster resides to reduce network latencies [31]. The traffic is sent in a controlled manner, as there are logically 50 concurrent users at a 150 TPS rate.

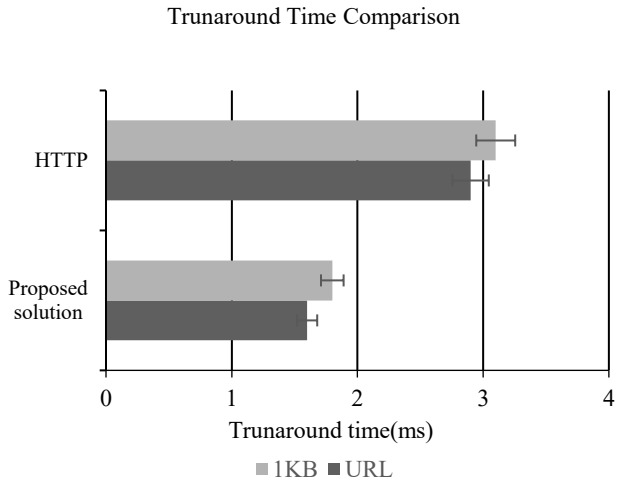

Fig. 6. HTTP and proposed solution comparison

Figure 6 illustrates the turnaround times for traditional HTTP versus the implemented solution methods. The graph shows that the implemented solution has a shorter turnaround time than the traditional HTTP method. This indicates that microservice A receives responses from microservice B at a faster pace with the implemented solution. Unlike the HTTP method, the implemented solution does not involve socket opening and closing activities. Additionally, data is serialised and transmitted as a byte buffer record, reducing network consumption. These factors contribute to the superior performance of the implemented solution over the traditional method. Based on the test results, the newly introduced method outperforms the traditional method. Therefore, further testing will be continued with the new strategy to assess its impact on microservice decompositions, inter-service communication, and overall performance.

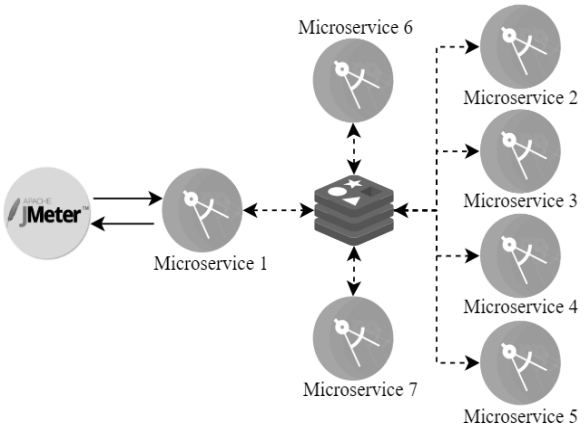

Fig. 7. Deployment architecture with proposed solution

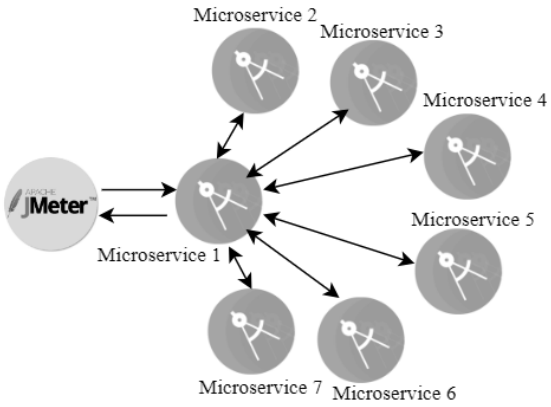

Fig. 8. Deployment architecture with HTTP communication

The diagram in Figure 7 illustrates the microservice architecture deployed to test in the cloud environment. Java language with Spring boot is used for microservice development, and an optimized TCP-based stream communication method is used for service-to-service communication, just as in the previous scenario. Figure 8 shows the deployment architecture used to evaluate the HTTP protocol. Both are deployed in the same environment, and Spring boot REST templates are used to create HTTP inter-service communications. 1 KB-sized payload is used as a request and response. As mentioned earlier, JMeter is placed on the same subnet on a different VM. The best and minimal communication approach discussed in the methodology has been used for microservice communications. Apache JMeter generates REST-based API traffic with controlled parameters such as concurrent users and throughput [32]. One of the microservices faces the JMeter and accepts the HTTP-based REST API traffic generated from logical 50 concurrent users at a 150 TPS rate as previously.

The diagram (Fig. 9) illustrates how increasing the number of microservices affects the overall system's average response time within a single-user functional requirement scenario. This analysis is grounded in data extracted from JMeter statistic reports, with each experiment conducted three times to ensure accuracy and reduce potential errors. The results reveal a definitive pattern: As the number of microservices grows, the system's average response time increases proportionally. In the early stages of decomposing the microservices, up to seven microservices are decomposed, and the proposed system demonstrates remarkable efficiency. During this phase, the response time remains below the linear average, indicating that the system can absorb the additional microservices without significantly impacting performance. However, a pivotal change occurs when the number of microservices exceeds six. At this point, the system's response time escalates rapidly, surpassing the linear average and moving into an exponential growth curve. This shift suggests that the system's ability to manage the increased load efficiently diminishes as more microservices are introduced, ultimately leading to a substantial decline in overall performance. When the proposed system is compared to the standard HTTP protocol, both exhibit similar trends in response time as the number of microservices increases. Yet, a critical difference emerges in their performance thresholds. The HTTP protocol not only begins to show a marked increase in response time at a lower number of microservices, but this increase is also more pronounced. Specifically, beyond the six-microservice mark, the HTTP protocol's response time escalates more sharply, indicating that it struggles to maintain efficiency under the same conditions where the proposed system still operates more effectively. This comparative analysis highlights the proposed system's superior capacity for handling a higher number of microservices than the HTTP protocol.

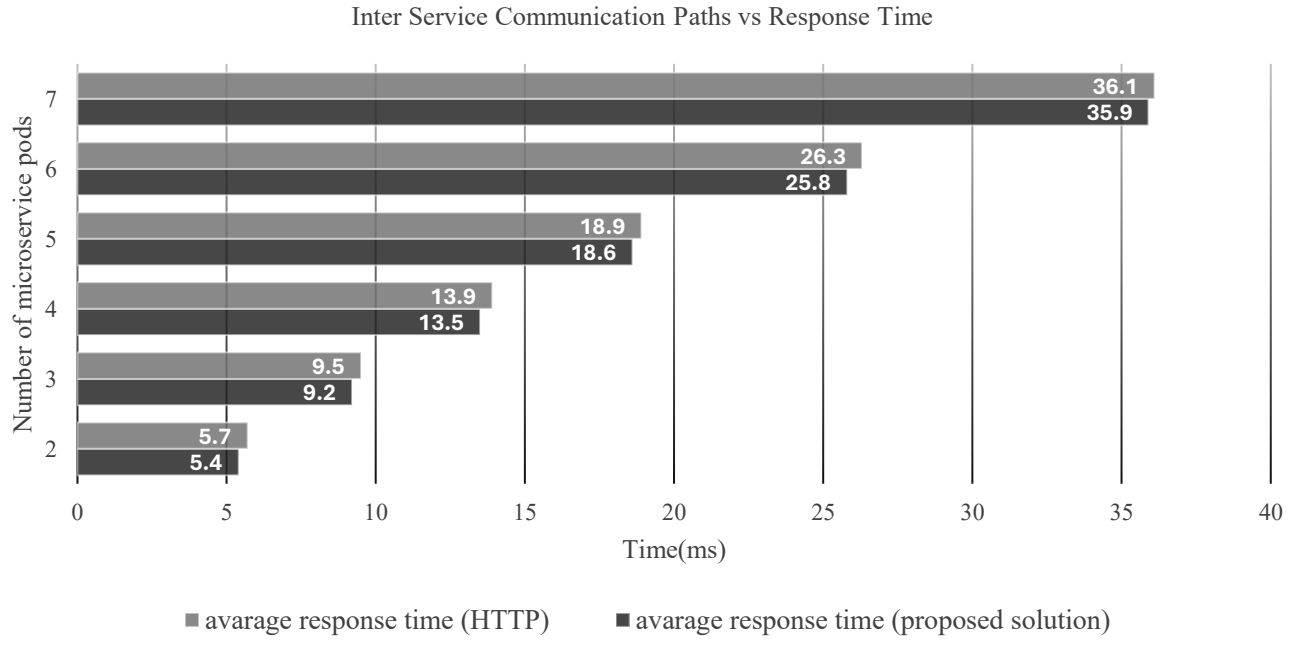

Fig. 9. Inter service communication path and response time comparison

TABLE II
INTER SERVICE COMMUNICATION TURNAROUND TIME VARIATION WITH MICROSERVICES

# Micro services	Time taken for each individual communication segment (ms)						Total inter-service communication time (ms)	Avg. inter-service communication time (ms)
	segment 1	segment 2	segment 3	segment 4	segment 5	segment 6		
2	1.8	N/A	N/A	N/A	N/A	N/A	1.8	1.8
3	1.7	2.1 (+16.67%)	N/A	N/A	N/A	N/A	3.8	1.9
4	1.8	2 (+11..11%)	2.5 (+38.89%)	N/A	N/A	N/A	6.3	2.1
5	1.8	2.3 (+27.78%)	2.5 (+38.89%)	3 (+66.67%)	N/A	N/A	9.6	2.4
6	1.8	2.1 (+16.67%)	2.5 (+38.89%)	3.5 (+94.44%)	4.5 (+150%)	N/A	14.4	2.88
7	1.9	2.2 (+22.22%)	2.7 (+50.00%)	4.1 (+127.78%)	4.9 (+172.22%)	5.4 (+200.00%)	21.2	3.53

The table above (Table II) illustrates the variation in inter-service communication turnaround time across different service-to-service communication segments as the number of microservices increases. This data was meticulously collected from each microservice log using Google's logging service, ensuring no logic processing time was included and focusing purely on communication overhead. The results highlight a critical challenge in microservices architecture: while the decomposition of services into smaller, isolated units offers advantages in scalability and modularity, it also introduces significant communication overhead that can impact overall system performance. When a new microservice is introduced by decomposing an existing one to isolate single responsibilities, the turnaround time for inter-service communication tends to increase. This increase is primarily due to the consumption of network layer resources for each communication packet. The analysis reveals that with only 2 microservices, the total inter-service communication time was 1.8ms, corresponding to an average of 1.8ms per segment. However, as the system expanded to 3 microservices, the total communication time increased to 3.8ms, with the second segment experiencing a 16.67% increase in turnaround time. This trend continues as the system grows: with 4 microservices, the total communication time rose to 6.3ms, with the second and third segments showing increases of 11.11% and 38.89%, respectively. The impact of additional microservices becomes even more pronounced with further scaling. At 5 microservices, the total time reached 9.6ms, with significant increases observed across multiple segments: 27.78% in the second, 38.89% in the third, and 66.67% in the fourth. By the time the system included 6 microservices, the total time climbed to 14.4ms, and the incremental increases in turnaround time became even more substantial: 16.67% in the second segment, 38.89% in the third, 94.44% in the fourth, and 150% in the fifth. The most significant jump occurs when the system scales to 7 microservices. Here, the total inter-service communication time reaches 21.2ms, with each segment showing substantial increases: 22.22% in the second segment, 50.00% in the third, 127.78% in the fourth, 172.22% in the fifth, and a striking 200.00% in the sixth segment. The average communication time per segment increases from 1.8ms with two microservices to 3.53ms with seven microservices, highlighting the cumulative impact of adding more microservices. This data underscores the complex trade-offs inherent in microservices architecture. While microservices improve modularity and scalability, they also introduce additional communication overhead, especially as system complexity grows. Each additional microservice contributes to a significant increase in latency and has a compounding effect on subsequent communication segments. This could potentially degrade the overall performance and responsiveness of the system, especially in high-traffic environments or in applications where low-latency communication is critical.

To better understand the implications of these findings, it is crucial for microservices architects to carefully consider the number of communication segments involved in each business scenario when decomposing services. If the number of segments exceeds sixth, it could significantly increase communication latencies, leading to overall application performance issues. Therefore, it is important to keep microservice decompositions as streamlined as possible, minimizing the number of communication segments to reduce latency and maintain optimal application responsiveness. This careful balance will help in achieving the benefits of microservices architecture while mitigating potential latency problems.

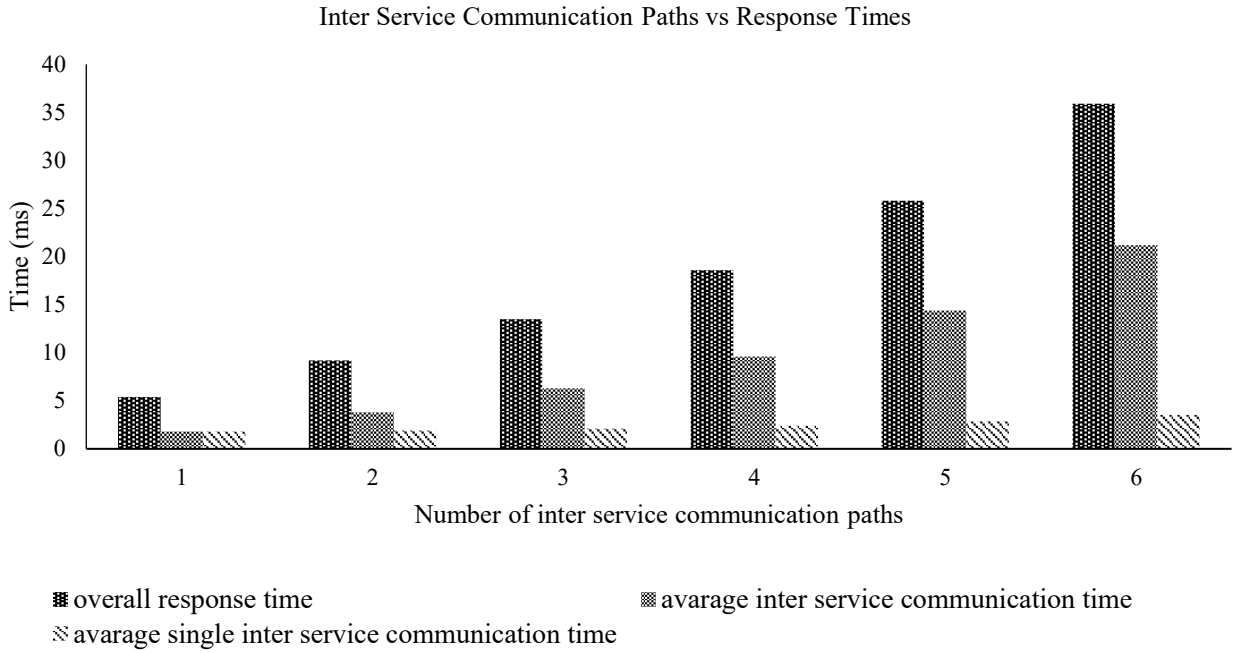

Fig. 10. Response time variation with the number of communication paths

The above graph (Fig. 10) compares response times across various layers within a microservice architecture, offering critical insights into how inter-service communication affects overall system performance. The overall response time values, derived from JMeter statistics, capture the complete end-to-end latency experienced by the user. In contrast, the average single inter-service communication turnaround times, calculated using Python scripts from microservice logs, focus exclusively on the latency introduced by the communication between individual services. A closer examination of the graph reveals a clear trend: as additional communication paths are introduced between microservices, the overall application response time progressively increases. This escalation is not linear but exhibits a compounding effect, especially as the number of communication segments grows.

After the fifth communication path, there is a marked and dramatic increase in the time required for inter-service communication, suggesting that the architecture reaches a critical point where additional communication segments introduce substantial latency. This phenomenon can be attributed to several underlying factors. First, each communication path introduces additional overhead, including network latency, protocol handling, and data serialization/deserialization processes. These overheads accumulate as more paths are added, resulting in longer delays. Additionally, the increased complexity of managing multiple communication channels can strain system resources, leading to inefficient processing and routing of messages between services. From a scientific perspective, the observed increase in response time after the fifth communication path may indicate a threshold in the system's capacity to efficiently handle distributed communication. This threshold likely corresponds to network bandwidth limitations and service orchestration bottlenecks. The dramatic rise in inter-service communication time highlights the non-linear nature of latency growth in microservice architectures, emphasizing the importance of carefully managing the number of communication paths to avoid significant performance degradation. Furthermore, the impact on user experience can overlay with this scenario. As response times increase, users may perceive the application as slow or unresponsive, leading to dissatisfaction and potential loss of engagement. This direct correlation between the number of communication paths and user-perceived performance underscores the critical need for optimization in microservice decomposition, particularly in scenarios involving complex service interactions.

In conclusion, the analysis of Fig. 10 reinforces the importance of strategic microservice decomposition and the careful management of communication paths within a microservice architecture. The findings suggest that maintaining a limited number of inter-service communication paths is crucial for preserving application performance and ensuring a positive user experience. This analysis provides a scientific basis for further exploration into optimizing microservice architectures, particularly in understanding the thresholds beyond which additional complexity leads to diminishing returns in performance.

Hardware resources significantly impact the performance of hosted applications, particularly the memory and CPU. To evaluate the effects of memory and CPU on the performance of microservices, the researchers have conducted a series of tests focusing on overall response time and inter-service communication. In this experimental setup, memory and CPU resources were allocated to individual microservice pods with both minimum and maximum resource limits set to the same values, ensuring full resource allocation to each pod. This approach allowed a controlled assessment of how these hardware resources influence microservice performance and the number of communication segments.

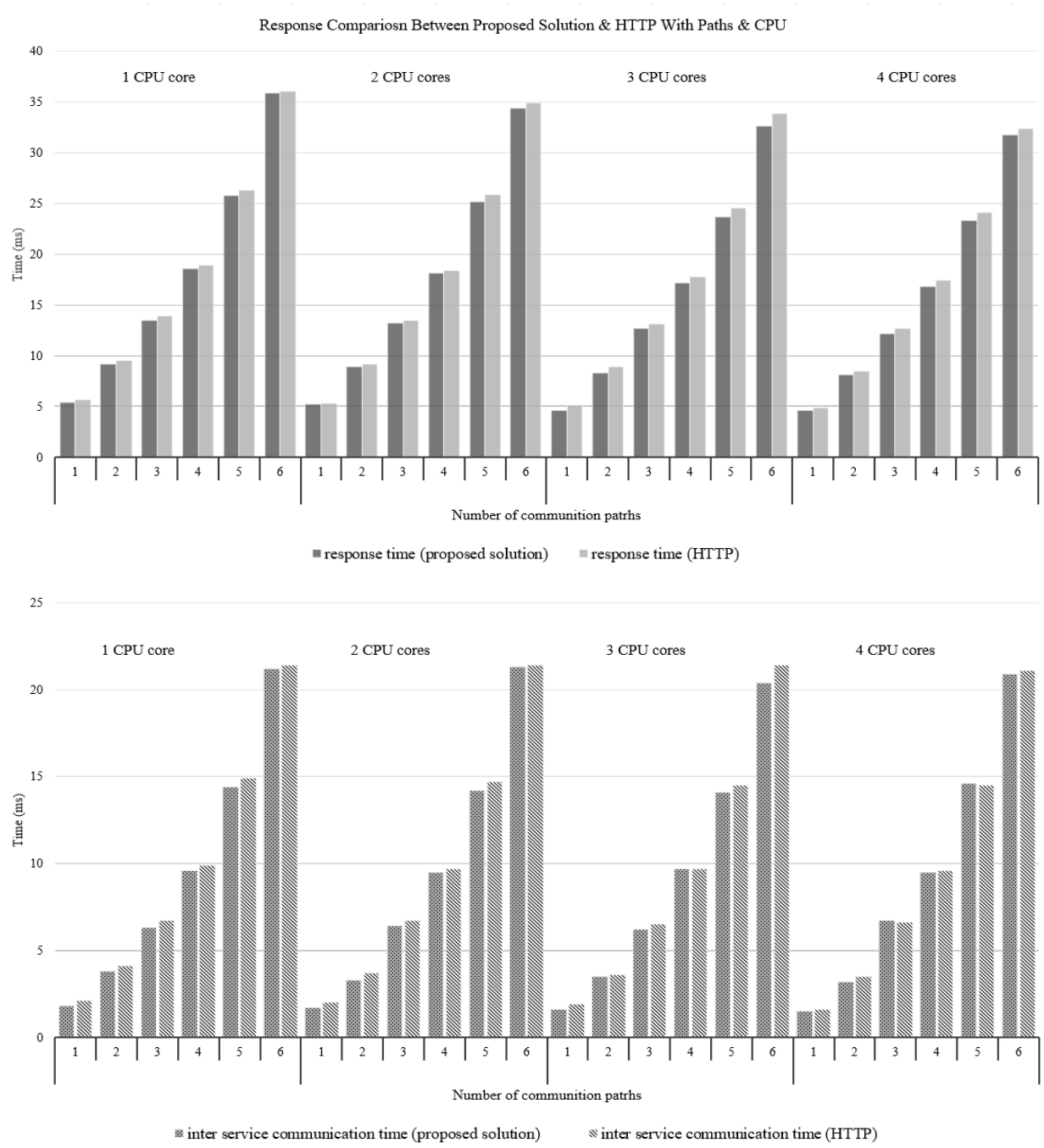

Fig. 11. Response time comparison with proposed solution and HTTP protocol in dynamic CPU allocation

The graph (Fig. 11) provides an insightful comparison of response time variations as a function of the number of communication paths and the CPU resources allocated between the proposed solution and the traditional HTTP protocol. A thorough data analysis reveals several critical observations about the performance dynamics of these two communication strategies. Firstly, the data indicates that inter-service communication times remain relatively stable despite increases in CPU allocation to the respective microservice pods. This stability suggests that the inter-service communication mechanisms, especially within the proposed solution, are not heavily dependent on the available CPU resources. This finding is significant because it highlights the efficiency of the proposed system in managing communication overheads independently of computational power, in contrast to the HTTP protocol, which may not exhibit the same efficiency level in similar scenarios. However, when examining overall application response times, a clear pattern emerges as more CPU cores are allocated, the overall response time improves noticeably. This improvement is due to the direct impact of increased CPU resources on the processing speed of microservices. More CPU cores enable microservices to execute computational tasks more swiftly, reducing internal processing delays and lowering the total response time experienced by the user. This enhancement is particularly evident in the proposed solution, which demonstrates a reduction in response time compared to the HTTP protocol under similar conditions. The proposed system's performance can be attributed to its optimized communication strategy, which is designed to minimize the overhead typically associated with HTTP-based inter-service communication. Unlike HTTP, which incurs substantial protocol overhead, including the need for repeated header exchanges, connection management, and stateless interactions, the proposed solution leverages a

more streamlined and efficient communication protocol. This approach reduces the number of bytes transmitted over the network and decreases connection establishment processes, which are known to be resource-intensive in HTTP-based systems. Furthermore, the proposed system's ability to maintain consistent inter-service communication times despite CPU allocation underscores its robustness in handling varying computational loads. This is particularly advantageous in cloud-native environments where resource allocation can fluctuate dynamically based on demand. The ability of the proposed solution to deliver consistent performance in such environments makes it a more reliable choice for applications requiring high scalability and low latency.

In conclusion, the analysis of Fig. 11 demonstrates that the proposed system not only outperforms the traditional HTTP protocol regarding response time when CPU resource allocation. This makes the proposed solution a more suitable choice for microservices architectures, particularly in scenarios where high efficiency and low latency are paramount. The findings suggest that the proposed system provides a more optimized approach to inter-service communication, enhancing overall application performance and a better user experience.

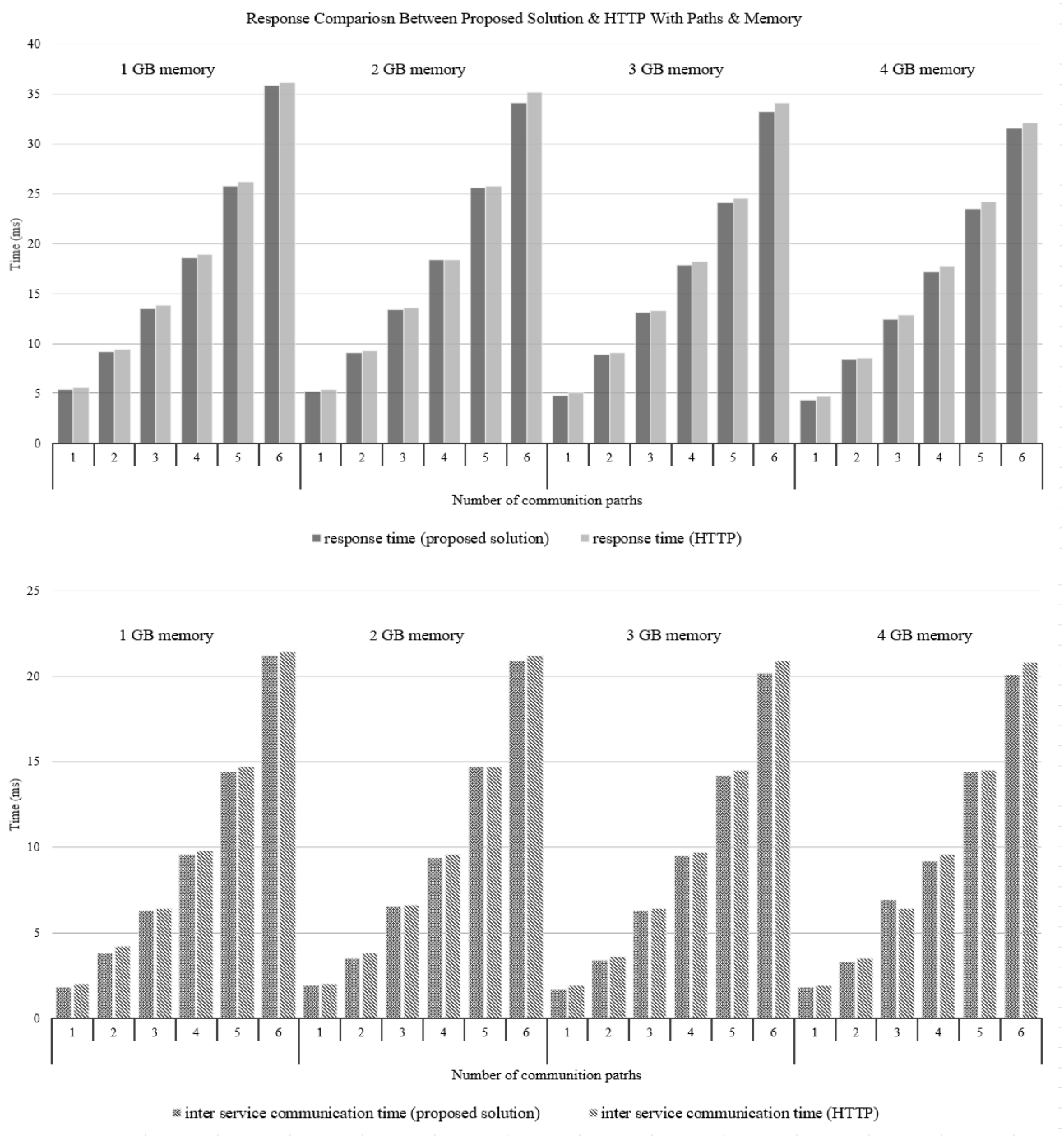

Fig. 12. Response time comparison with proposed solution and HTTP protocol in dynamic memory allocation

In addition to evaluating the impact of CPU allocation, the tests also examined how memory allocation variations influence overall microservice response time and inter-service communication time. The data depicted in the graph (Fig. 12) reveals some critical insights that underscore the strengths of the proposed system compared to the traditional HTTP protocol. Firstly, the graph illustrates that inter-service communication time remains largely unaffected by changes in memory allocation. This observation is significant because it suggests that the communication mechanisms employed by the proposed system are not heavily reliant on the memory resources available to individual microservice pods. This stability clearly indicates the proposed system's robustness in handling inter-service communications efficiently, even when memory resources vary. HTTP-based communication may exhibit the same level of independence from memory constraints but with less performance than the proposed solution. However, while inter-service communication time shows little sensitivity to memory allocation, the graph does indicate a modest improvement in overall application response time with increased memory allocation. This improvement can be attributed to the crucial role that adequate memory plays in maintaining the smooth and efficient operation of microservices. Specifically, sufficient memory allocation helps reduce the frequency and duration of garbage collection cycles, which can reduce latency spikes and degrade performance.

The proposed system demonstrates a more pronounced memory utilisation efficiency than the HTTP protocol. Unlike HTTP, which may suffer from higher memory consumption due to the overheads associated with managing stateless requests, connection management, and data serialization/deserialization processes, the proposed system optimizes memory usage by streamlining these operations. As a result, the proposed system can handle larger data loads more effectively, leading to faster overall response times and better system performance.

Moreover, the proposed system's ability to handle more substantial data loads with increased memory allocation contributes to its superior performance in real-world scenarios. For instance, in high-traffic environments where microservices are required to rapidly process and exchange large volumes of data, the proposed system's efficient memory management ensures that performance remains consistent and responsive. When allocating more memory to the HTTP protocol, there is a performance improvement, but it does not match the level of enhancement seen with the proposed system. Furthermore, while the HTTP protocol can benefit from increased memory allocation to some extent, the proposed system's architecture inherently makes better use of available resources. By minimizing the overhead associated with inter-service communication, the proposed system ensures that the majority of the memory resources are dedicated to actual processing tasks rather than managing communication logistics. This efficiency translates into more consistent and lower response times across various memory configurations, making the proposed system a more reliable choice for memory-constrained environments.

In conclusion, the comprehensive analysis of memory allocation's impact on microservice performance reveals that while both the proposed system and the HTTP protocol benefit from increased memory, the proposed system outperforms HTTP by a significant margin. The proposed system's capacity to maintain stable inter-service communication times and efficient memory utilization leads to faster overall response times and improved performance. These findings underscore the superiority of the proposed system over the traditional HTTP protocol, particularly in environments where memory resources are variable or limited. The proposed system's optimized communication and resource management strategies make it a more suitable solution for modern microservices architectures that demand high efficiency, scalability, and low latency.

Our findings highlight the critical role of adequate memory allocation in enhancing microservice performance. Memory allocation improves overall application response time by optimizing internal processing capabilities but does not significantly impact inter-service communication times. Similarly, increasing CPU allocation enhances processing speed and reduces response time but does not substantially affect inter-service communication times. This distinction emphasizes the need to optimize both computational and communication aspects independently to achieve comprehensive system performance improvements.

VI. Limitations

This study continues our research on a newly introduced communication strategy, aiming to draw concrete conclusions about its impact on microservices architecture performance. While previous findings indicated that HTTP-based communication is predominantly used for inter-service messaging, our work benchmarks our proposed solution against HTTP to demonstrate its performance advantages. Our study primarily investigates how inter-service communication affects the overall response time of microservices architectures, comparing HTTP with our proposed strategy. Unlike previous research, we aimed to identify optimal segments in HTTP and the newly introduced strategy. Our findings are limited to identifying the optimal number of communication segments required to maintain optimal microservice performance regarding response time. Additionally, we assessed the impact of vertical scaling on communication segments. However, we restricted our analysis to experimental data generated within controlled environments, which may not fully capture real-world complexities. Future studies could extend this research by exploring additional parameters related to communication segments and examining real-world deployment scenarios to validate our findings and further generalize the applicability of the proposed strategy.

VII. Summary and Future Works

The transition from a monolithic software architecture to a microservices architecture highlights the critical role of optimized inter-service communication. This emphasis underscores the significance of establishing efficient communication pathways among microservices within the architecture. Organizations can enhance their microservices-based systems' overall performance, scalability, and

maintainability by prioritizing optimized inter-service communication. This recognition underscores a fundamental aspect of successful microservices adoption, as effective communication mechanisms are essential for facilitating seamless interactions between microservices and ensuring the agility and responsiveness required in modern software architectures. The methodology includes understanding the organizational context, identifying responsibilities, defining services based on domain-driven design, establishing data models, designing clear APIs, selecting appropriate technologies, deciding deployment methodologies, implementing testing strategies, and planning for evolution and maintenance. This research investigates two distinct communication scenarios within microservices architecture: a worst-case scenario characterized by intricate meshed dependencies and a best-case scenario featuring minimal and direct service interactions. Graph theory principles are applied to quantitatively analyze the complexity of communication paths in these scenarios. In the evaluation phase of the study, microservices are deployed in a cloud-native Kubernetes cluster environment. Subsequently, the impact of inter-service communication on the overall system performance is rigorously assessed. This evaluation involves measuring various performance metrics, such as response time, throughput, and resource utilization, to gain insights into how different communication patterns influence the efficiency of microservices-based systems deployed in real-world cloud environments. Through this approach, the research aims to provide valuable insights to the optimal design and management of inter-service communication to maximize the performance and reliability of microservices architectures. Optimizing both CPU and memory allocations is crucial for enhancing microservice performance, as they improve processing efficiency and overall response times, while network factors and communication strategy primarily influence inter-service communication. The results underscore the critical importance of planning and optimizing inter-service communication to guarantee microservices-based systems' efficiency, scalability, and responsiveness.

REFERENCES

[1] M. M. Jamjoom, A. S. Alghamdi, and I. Ahmad, "Service Oriented Architecture Support in Various Architecture Frameworks: A Brief Review," 2012.

[2] 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.

[3] King Khalid University and N. Ahmad, "The Structural Modeling of Significant Factors for Sustainable Cloud Migration," *Int. J. Intell. Eng. Syst.*, vol. 14, no. 2, pp. 1–10, Apr. 2021, doi: 10.22266/ijies2021.0430.01.

[4] Srinivas Balasubramanian, Prakash Raghavendra, "HAMP - A Highly Abstracted and Modular Programming Paradigm for Expressing Parallel Programs on Heterogenous Platforms," Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering 2012, WCE 2012, 4-6 July, 2012, London, U.K., pp1130-1135.

[5] L. Liu, X. He, Z. Tu, and Z. Wang, "MV4MS: A Spring Cloud based Framework for the Co-Deployment of Multi-Version Microservices," in *2020 IEEE International Conference on Services Computing (SCC)*, Beijing, China: IEEE, Nov. 2020, pp. 194–201. doi: 10.1109/SCC49832.2020.00033.

[6] S. Rochimah, I. M. B. Gautama, and R. J. Akbar, "Refactoring the Anemic Domain Model using Pattern of Enterprise Application Architecture and its Impact on Maintainability: A Case Study," 2019.

[7] B. P. Gautam and S. K. Shrestha, "Effective Campus Management through Web Enabled Campus-SIA (Student Information Application)," *Hong Kong*, 2012.

[8] L. O'Brien, P. Merson, and L. Bass, "Quality Attributes for Service-Oriented Architectures," in *International Workshop on Systems Development in SOA Environments (SDSOA'07: ICSE Workshops 2007)*, Minneapolis, MN, USA: IEEE, May 2007, pp. 3–3. doi: 10.1109/SDSOA.2007.10.

[9] S. Weerasinghe and I. Perera, "Taxonomical Classification and Systematic Review on Microservices," *Int. J. Eng. Trends Technol.*, vol. 70, no. 3, pp. 222–233, Mar. 2022, doi: 10.14445/22315381/IJETT-V70I3P225.

[10] L. O'Brien, L. Bass, and P. Merson, "Quality Attributes and Service-Oriented Architectures:," Defense Technical Information Center, Fort Belvoir, VA, Sep. 2005. doi: 10.21236/ADA441830.

[11] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi, "From monolithic systems to Microservices: An assessment framework," *Inf. Softw. Technol.*, vol. 137, p. 106600, Sep. 2021, doi: 10.1016/j.infsof.2021.106600.

[12] S. Weerasinghe and I. Perera, "Optimised Strategy for Inter-Service Communication in Microservices," *Int. J. Adv. Comput. Sci. Appl.*, vol. 14, no. 2, 2023, doi: 10.14569/IJACSA.2023.0140233.

[13] N. Levin, "How to Build Apps using Redis Streams",Available: https://www.academia.edu/40811333/How_to_Build_Apps_using_Redis_Streams

[14] S. T. Aung, L. H. Aung, N. Funabiki, S. Yamaguchi, Y. W. Syaifudin, and W.-C. Kao, "An Implementation of Web-based Personal Platform for Programming Learning Assistant System with Instance File Update Function," vol. 32, no. 2, 2024.

[15] Y. Wang, T. Towara, and R. Anderl, "Topological Approach for Mapping Technologies in Reference Architectural Model Industrie 4.0 (RAMI 4.0)," 2017.

[16] M.-D. Cojocaru, A. Uta, and A.-M. Oprescu, "Attributes Assessing the Quality of Microservices Automatically Decomposed from Monolithic Applications," p. 10.

[17] V. Lenarduzzi, F. Lomio, N. Saarimäki, and D. Taibi, "Does migrating a monolithic system to microservices decrease the technical debt?," *J. Syst. Softw.*, vol. 169, p. 110710, Nov. 2020, doi: 10.1016/j.jss.2020.110710.

[18] M. I. Murillo and M. Jenkins, "Technical Debt Measurement during Software Development using Sonarqube: Literature Review and a Case Study," in *2021 IEEE V Jornadas Costarricenses de Investigación en Computación e Informática (JoCICI)*, Oct. 2021, pp. 1–6. doi: 10.1109/JoCICI54528.2021.9794341.

[19] M. Kamimura, K. Yano, T. Hatano, and A. Matsuo, "Extracting Candidates of Microservices from Monolithic Application Code," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, Nara, Japan: IEEE, Dec. 2018, pp. 571–580. doi: 10.1109/APSEC.2018.00072.

[20] V. Velepucha and P. Flores, "Monoliths to microservices - Migration Problems and Challenges: A SMS," in *2021 Second International Conference on Information Systems and Software Technologies (ICI2ST)*, Quito, Ecuador: IEEE, Mar. 2021, pp. 135–142. doi: 10.1109/ICI2ST51859.2021.00027.

[21] X. Liu, S. Jiang, X. Zhao, and Y. Jin, "A Shortest-Response-Time Assured Microservices Selection Framework," in *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, Guangzhou: IEEE, Dec. 2017, pp. 1266–1268. doi: 10.1109/ISPA/IUCC.2017.00192.

[22] A. Singjai, U. Zdun, and O. Zimmermann, "Practitioner Views on the Interrelation of Microservice APIs and Domain-Driven Design: A Grey Literature Study Based on Grounded Theory," in *2021 IEEE 18th International Conference on Software Architecture (ICSA)*, Mar. 2021, pp. 25–35. doi: 10.1109/ICSA51549.2021.00011.

[23] D. Sanchez, A. E. Rojas, and H. Florez, "Towards a Clean Architecture for Android Apps using Model Transformations," vol. 49, no. 1, 2022.

[24] L. Lun, X. Chi, and H. Xu, "Coverage Criteria for Component Path-oriented in Software Architecture," 2019.

[25] E. M. I. M. Ekanayaka, J. K. K. H. Thathsarani, D. S. Karunanayaka, N. Kuruwitaarachchi, and N. Skandhakumar, "Enhancing Devops Infrastructure For Efficient Management Of Microservice Applications," in *2023 IEEE International Conference on e-Business Engineering (ICEBE)*, Nov. 2023, pp. 63–68. doi: 10.1109/ICEBE59045.2023.00035.

[26] Christy Sibi Pachikkal, "Interservice Communication in Microservices," *Int. J. Adv. Res. Sci. Commun. Technol.*.
[27] B. Götz, D. Schel, D. Bauer, C. Henkel, P. Einberger, and T. Bauernhansl, "Challenges of Production Microservices," *Procedia CIRP*, vol. 67, pp. 167–172, 2018, doi: 10.1016/j.procir.2017.12.194.
[28] 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.
[29] "RESP protocol spec," Redis. [Online]. Available: https://redis.io/docs/reference/protocol-spec/
[30] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Kubernetes as an Availability Manager for Microservice Applications".
[31] "Google Kubernetes Engine (GKE) | Google Cloud." [Online]. Available: https://cloud.google.com/kubernetes-engine
[32] R. B. Khan, "Comparative Study of Performance Testing Tools: Apache JMeter and HP LoadRunner," p. 57.

L.D.S.B Weerasinghe is a postgraduate student at the Department of Computer Science and Engineering, the University of Moratuwa. He received B.Sc. (Hons) in Computer Science with first class from the Kotelawala Defence University and M.Sc. in Computer Science (Specialization in Cloud Computing) from the University of Moratuwa. His research interests include software architecture, cloud computing and distributed computing.

Indika Perera is a Professor at the Department of Computer Science and Engineering, the University of Moratuwa. He received the B.Sc. Engineering (Hons.) and M.Sc. degrees from the University of Moratuwa, Sri Lanka, the Master of Business Studies from the University of Colombo, Sri Lanka, and the Ph.D. degree from the University of St Andrews, U.K. His research interests include artificial intelligence, software engineering, and application development for bio-health research.