Online-Journals.org

iJOE International Journal of Online and Biomedical Engineering

iJOE | eISSN: 2626-8493 | Vol. 20 No. 1 (2024) | OPEN ACCESS

https://doi.org/10.3991/ijoe.v20i01.44021

PAPER

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

**Sidath Weerasinghe(✉), Indika Perera**

Department of Computer Science & Engineering, University of Moratuwa, Moratuwa, Sri Lanka

weerasingheldsb.20@uom.lk

**ABSTRACT**

Cloud computing has become a prominent technology in the software development industry. The term "cloud-native" is derived from cloud computing technologies and refers to the development and deployment of applications in a cloud environment. In the software industry, most enterprise-grade software buildings use the microservice architecture and cloud natively, ultimately leading to an expansive development in the software development framework. Microservices are deployed in a distributed environment and function as independent services. However, they need to communicate with each other in order to fulfill the functional requirement. Additional latency will be introduced when communicating with other services. Hence, it will impact the overall application response time and throughput. This research proposes a solution for the aforementioned problem in the cloud-native environment. A Request-response-based TCP communication solution has been developed and tested in the cloud-native, containerized environment. Experimental results showed that the turnaround time of the proposed solution is shorter than that of traditional HTTP communication methods. Furthermore, the results summarize that both vertical and horizontal scaling are improving the overall performance of the systems performance in terms of response time. Conclusively, the proposed solution improved the microservice performance and preserved the existing cloud-native qualities, such as scalability, maintainability, and portability.

**KEYWORDS**

cloud native, microservice, inter-service communication

## 1 INTRODUCTION

As user requirements become increasingly complex, it is challenging to efficiently address them using a monolithic architecture. Consequently, companies have begun transitioning to microservice architecture in software development to better accommodate these evolving needs. This approach involves breaking down large, monolithic applications into small, reusable services that can be utilized

Weerasinghe, S., Perera, I. (2024). Optimized Strategy in Cloud-Native Environment for Inter-Service Communication in Microservices. *International Journal of Online and Biomedical Engineering (iJOE)*, 20(1), pp. 40–57. https://doi.org/10.3991/ijoe.v20i01.44021

Article submitted 2023-08-13. Revision uploaded 2023-10-24. Final acceptance 2023-10-31.

across multiple applications. In a monolithic architecture, all components are tightly coupled. Therefore, scaling the application necessitates scaling the entire system, which can be inefficient and costly. With microservices, components are loosely coupled and can be scaled independently, allowing for more efficient resource utilization. Each microservice can be developed, tested, and deployed independently. This makes it easier to incorporate complex user requirements, fix bugs, and update the system without impacting the entire application. In contrast, in a microservices architecture, components are designed to be resilient and capable of continuing to function even if other components fail. With a microservices architecture, developers have the flexibility to select the most suitable technology stack for each component, potentially resulting in improved scalability and reliability. Many of these patterns are pioneered by companies such as Netflix, which is famous for migrating from a monolithic architecture to a microservice architecture.

In a microservice architecture, each microservice must communicate with one or more microservices to fulfill the functional requirements. Two main methods are used for inter-service communication: synchronous and asynchronous. Numerous technologies are involved in these two communication methods. In every environment, networking plays a crucial role in the system, as network behavior is constantly changing and serves as a dynamic resource. Working in a dynamic environment can be challenging, requiring individuals to take responsibility for environmental changes. For instance, when a particular service sends a large message request to another service over the network, it consumes resources, ultimately affecting the overall network. As a result, the message flow will slow down. As a result, inter-service communication during execution will cause additional latency in the operation. As a result, the application's overall performance is likely to degrade in terms of both application response time and throughput. In a microservices architecture, individual microservices use their own data sources. To tackle the performance challenges that come with this setup, caching mechanisms have been implemented. However, there is still no standard method or guideline being implemented to improve application performance in service-to-service communication.

Cloud-native is a modern software development and deployment approach that utilizes cloud technologies and related methods to construct and deploy software that is more resilient, easily scalable, and flexible. It is a modern approach to developing and managing software that leverages cloud computing to enhance the delivery of software products. The cloud-native approach involves developing applications as a set of small, autonomous services running in their own containers, which can be deployed and managed independently. These services communicate with each other using lightweight protocols, such as HTTP, which makes it easier to build and maintain complex applications. Containers offer a lightweight and portable environment for applications, enabling them to seamlessly transition between development, testing, and production environments. Overall, cloud-native is a contemporary approach to microservice software development that harnesses the capabilities of cloud computing to construct and deploy scalable, resilient, and adaptable services. It enables organizations to enhance the delivery of software products by leveraging the latest tools and technologies available in the cloud computing ecosystem. Cloud-native containerized environments are the optimal deployment environments for microservice-based architectural software. However, the performance issue continues to escalate due to the need for microservice containers to exchange

data over the network. This research primarily focuses on developing a method to optimize inter-service communication in microservice architectures deployed in cloud-native environments.

## 2 LITERATURE REVIEW

### 2.1 Background

In the early days, software was developed using a monolithic architecture. However, due to the constant complexities of user requirements, monolithic architectural software did not have the capability to accommodate those changes in requirements at that time. Service-Oriented Architecture (SOA) was introduced with the separation of concepts. Subsequently, most software companies adopted the SOA architecture for their development. The Enterprise Service Bus is utilized in the SOA architecture for service orchestration, which has proven to be a major bottleneck for that architecture. Nevertheless, SOA has gained additional quality attributes, such as maintainability, scalability, and security, in comparison to monolithic architecture. To address the challenges of SOA architecture, developers have introduced a microservice-based architecture to software development. This research conducted several test cases to evaluate the performance of SOA and microservice architecture in terms of response time and throughput. In addition, the study calculated the resource consumption and associated costs for these two architectures [1]. The test results showed that the microservice architecture can overcome the bottleneck of the SOA architecture and is a cost-effective solution compared to the SOA architecture.

A systematic review was conducted following the PRISMA model to examine the past, present, and future [2]. Microservices research began in the early 2000s. Most of the research conducted on microservice architecture focuses on frameworks, observability, and cloud computing. Most high-tech companies, such as Spring Boot, Vertx, and Go Micro, focus on developing and improving their framework. However, the microservice architecture is facing performance issues related to latency and throughput due to interservice communications in a distributed environment. In previous studies, we evaluated inter-service communication in the microservice architecture using commonly used communication mechanisms such as HTTP, gRPC, and WebSocket [3]. The HTTP protocol is frequently used for interservice communication in microservices, and most microservice frameworks also support this communication method. According to our research findings, the gRPC protocol outperforms other protocols. Nevertheless, managing the message flow will be more challenging than with the HTTP protocol, as most enterprise software relies on HTTP-based communication. In that regard, only a few frameworks support gRPC communications. It was also observed that latency and throughput depend on the content of the messages communicated between the microservices. All these tests are run in a VM-based environment. Currently, most microservices are deployed in cloud-native environments because the quality attribute can be easily achieved through the correct microservice architecture in cloud-native environments. Based on all the research findings, we have determined that the majority of enterprise software is developed using HTTP for inter-service communication. This research also examines the behavior of HTTP-based communication to evaluate and propose a solution.

## 2.2 Related work

Most software architects consider various software quality attributes, including scalability, performance, portability, security, and many others. Antonio Bucchiarone and his team investigated the process of achieving scalability by transitioning from a monolithic architecture to a microservice architecture in the mission-critical banking and financial sectors [4]. Through transitioning to the microservice architecture, they reduced system complexity and simplified integrations with lower coupling and higher cohesion. With this separation, the researcher was able to adjust the required services based on traffic. Budapest University of Technology conducted a comparative review of monolithic and microservice architectures to assess the performance impact of concurrent requests [5]. The researcher utilized the JMeter tool to generate concurrent requests and measure the response time and throughput of both microservices and monolithic architecture software. The test results indicated that microservices and monolithic software exhibited similar performance factors under normal load. However, under high load, the response time and throughput degraded in the microservice architecture. Momil Seedat and the research team systematically mapped the transition from monolithic architecture to microservice architecture and presented a technique for identifying microservices within the monolithic architecture using the concept of domain-driven modeling [6]. This model includes domain analysis, bounded context discovery, domain service selection, microservice identification, and aggregation. The technique proposed by the research team can help enhance overall application performance and can be applied to the system transition model. However, that is not a suitable option for software with a lower level of complexity. The Ministry of Technical and Vocational Education in Libya conducted a study on the structural differences between microservices and monolithic architecture [7]. According to their research study, the monolithic architecture is suitable for small software developed by a small team. However, if the software is complex and requires several teams to build it, then microservice architecture is the best fit for that scenario. Nowadays, most software is built to enterprise-grade standards, and such software handles complex logic. Therefore, as we move forward, all software development is involved with microservice development.

In the realm of deployment strategies, microservices are constantly evolving due to technological advancements. Most of the research on microservices is conducted using on-premise virtual machines (VMs) or enterprise-grade cloud VMs. The microservice architecture software is also deployed in the same environments. A team of researchers at the University of Bozen-Bolzano studied different microservice architectural patterns and principles, taking into account their deployment using DevOps techniques. According to research, microservice orchestration and data storage patterns are identified as emerging areas within the scope of microservices. In the DevOps context, continuous integration and continuous deployment (CI/CD) pipelines, cloud orchestration and coordination, and scalability are identified as major research trends. When examining similar research studies, it is evident that cloud-native development and deployments are playing a significant role in the context of microservices. Nane Kratzke and Peter-Christian Quint systematically reviewed the cloud-native application, starting from the perspective of cloud computing, and identified isolated engineering. Trends with the technologies [8]. They are developing microservices in a cloud-native manner, utilizing DevOps, softwareization, elastic platforms, loose coupling, and APIs. The microservice focuses on

developing independent services that can be horizontally scalable and deployed on elastic platforms such as Kubernetes [9], Apache Mesos [10], and Docker Swarm [11]. With DevOps and softwareization technologies, automation, rapid releases, and software builds are facilitated. Concordia University Montreal and Ericsson Inc. conducted research on the Availability Management Framework for cloud-native microservices [12]. This research primarily focuses on deploying microservices in a cloud-native environment using container orchestration techniques. The research team at Vellore Institute of Technology, Chennai, evaluated the impact of replacing on-premise applications with cloud-native applications [13]. They have introduced the cloud-based AppStream, which is deployed in the AWS cloud, and compared the cost, response time, and storage usage with on-premises deployment. According to the statistics, migrating to the cloud reduces overall deployment costs as well as response time and required storage. Manish Saraswat et al. conducted an analysis of leading cloud providers in their respective areas of expertise [14]. According to the researchers, Azure Cloud is ideal for users of Windows-based operating systems, AWS Cloud is suitable for stable services, and Google Cloud is well-suited for users working with container-based models and innovations. Most microservices are now deployed as containers. According to Manish Saraswat, Google Cloud provides greater value for container orchestrations.

In a cloud environment, similar to the on-premise setup, microservices also need to interact with other microservices or cloud services to process the required output. In a VM-based environment, microservices utilize various protocols to communicate with other services, including HTTP, HTTPS, Web Sockets, gRPC, and other messaging protocols, such as JMS and AMQP, which are increasingly popular for microservices communications with a publisher-subscriber architecture. They are facilitated by well-known message brokers such as Apache Kafka, Apache ActiveMQ, and RabbitMQ [15]. Most of the microservices are currently deployed in containerized environments. Therefore, such environments also require a proper mechanism for inter-service communication. The KTH Royal Institute researched the service mesh architecture in the Kubernetes environment for microservices [16]. The Istio service mesh has been utilized for researching and evaluating software quality attributes [17]. According to research analysis, service meshes bring ease of use to application developers. However, after using these tools and technologies, application performance degraded in terms of latency, CPU, and memory consumption. Further studies have found that Service Mess lacks sufficient testing tools and technologies in the quality assurance area. Certain industrial articles highlight the four types of communication models in Kubernetes environments: container-to-container communications, pod-to-service communications, pod-to-pod communications, and external-to-internal communications [18].

## 3 METHODOLOGY

This research aims to implement an optimized solution for inter-service communication in microservices deployed in cloud-native environments, ensuring minimal performance impact during message passing between the microservices. When introducing a solution for inter-service communication, it is essential to focus on message transfer protocol, network layer resource consumption, communication style, and programming complexity. The message transfer protocol is very important because the OSI layer used for communication is solely determined by that decision.

Most applications use the application layer for communication because it supports a wide range of libraries. If an application can utilize the network layer for message transport, it can achieve better performance compared to using the application layer. These protocols handle the addressing, routing, and swift delivery of packets to their destinations. By prioritizing efficiency at the network layer, overall network performance can be improved. The consumption of resources at the network layer directly impacts communication. Each time, the application needs to send the smallest possible network packet to conserve network resources; otherwise, communication can be impacted by network congestion, latency, and bandwidth limitations. Asynchronous communication is gaining popularity for event-driven programming. Most applications use publish-subscribe model for asynchronous communication and the message broker.

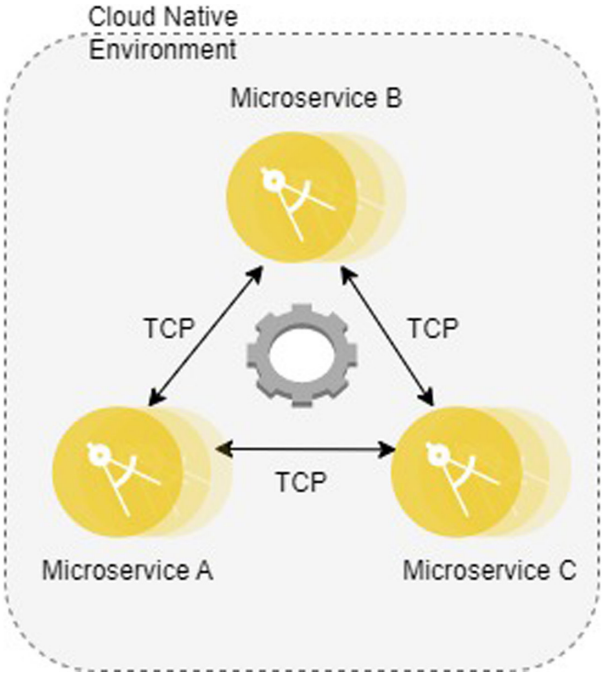


**Fig. 1.** Architecture

When considering all the facts, the optimal solution should utilize the TCP layer for communicating with other services. Conversely, messages should be transmitted to other services in a compressed format over the network. The solution should utilize pub-sub communication with message delivery and a guarantee of exact one-delivery. As shown in Figure 1, when communicating with the TCP layer protocol, there should be a mechanism to ensure message reliability. When considering these requirements, the MQTT and AMQP protocols with topics and queues would not be a better solution in this scenario due to their complexity and performance overhead. Cloud-native environments are dynamically scaled up and down, so these protocols do not include built-in mechanisms for service discovery. Scaling can be more challenging because of the necessity of managing message routing and queues. The selected method must support serialization and deserialization without causing performance overhead to minimize network resource consumption during message transmission between microservices over the network. Additionally, it should also facilitate dynamic scaling in cloud-native environments.

### 3.1 Technology selection

According to the literature review, the majority of enterprise software companies use container orchestration engines for deploying microservices when adopting cloud-native technologies. Kubernetes, the most powerful container orchestration engine, offers robust capabilities for application deployments, including easy vertical and horizontal scalability, high availability, and maintainability [19]. There are various ways to use Kubernetes, including managing it through cloud providers, installing it on on-premise servers using Kubeadm/Kops, and running MiniKube on desktop machines [20]. Major cloud providers such as AWS, Google, and Azure offer managed Kubernetes services known as Amazon Elastic Kubernetes Service (EKS), Google Kubernetes Engine (GKE), and Azure Kubernetes Service (AKS). Any Kubernetes engine can be used for the proposed solution. For research purposes, the Google Kubernetes Engine has been utilized in this scenario. Google manages GKE [21] and offers additional features. Kubernetes manages the containers and container clusters running on the Google Cloud infrastructure. The Google Cloud provider manages the underlying architecture and administration of Kubernetes, allowing users to deploy their containerized applications to the GKE platform. Kubernetes follows the master and worker patterns in its architecture. For the worker node, individuals can determine the specifications and other worker pools based on the nature of the microservice. However, the master node is fully managed by the Google Cloud provider. Therefore, the provider also ensures quality attributes such as security, high availability, and resilience. With the latest autopilot GKE feature, application developers do not need to worry about the nodes and node capacity of the worker node cluster. GKE is responsible for the entire cluster infrastructure, including both the master and the worker nodes. Researchers from Google state that the autopilot feature can save up to 85% of resources and improve operational efficiency [21].

Redis serialization protocol (RESP) is a protocol to communicate with its clients [22]. This protocol utilizes the serialization communication style and the binary-safe protocol mechanism. This approach minimizes bandwidth usage when transforming data between Redis and the clients, and vice versa, compared to other protocols. Microservices can communicate with each other using a TCP-based connection, which offers lower bandwidth and higher speed. This is achieved using RESP without defining a new protocol. Redis streams also communicate with clients using a stream-oriented connection, similar to Unix sockets [23]. One microservice acting as the publisher can publish messages to the stream using the stream key, while the consumer microservice can subscribe to the stream keys. With those subscriptions, consumers are able to access the published content. The primary advantage of Redis streams over other streams, such as Kafka, is that Redis manages the append-only data structure. This feature facilitates real-time messaging, as Redis streams persist messages and ensure the exact delivery of published messages. Furthermore, Redis provides high availability and fault tolerance through its built-in replication mechanism. When considering microservice-inter-service communication, message reliability is crucial. Therefore, the Redis stream architecture matches the proposed communication mechanism.

In the proposed system, inter-service communication is enabled through Redis streams using the RESP protocol. When the microservice pods are initialized, they establish a TCP-based connection with the Redis pod and maintain the connection until the pod is terminated. As a result, when sending and receiving messages, microservices do not need to worry about opening and closing connections.

The opening and closing times of network sockets will not affect this solution, thereby enhancing the overall performance of the microservice. Messages sent and received via the RESP protocol using TCP-based connections, such as Unix sockets, are serialized and sent as byte buffers to minimize network-level consumption in comparison to other protocols. In the proposed solution, all the HTTP headers were used when sending and receiving messages. The developer needs to utilize the developed library to send and receive requests, and the response is the same as for the HTTP clients. Software developers will be able to utilize the developed library without altering the existing programming architecture of the system.

## 4 IMPLEMENTATION

This section emphasizes the implementation aspects of the proposed system. The Java language is widely used as a programming language, and the Spring Boot framework is one of the most popular microservice frameworks. The literature review reveals that the majority of researchers and industry applications utilize the Java language and the Spring Boot frameworks to develop research-based microservices and enterprise applications. Based on those guidelines, we decided to utilize the Java and Spring Boot framework to implement this research component. The latest iterations of the Spring Boot framework support the development of cloud-native applications. The OpenJDK docker base image has been used for creating the docker images since those images are optimized for the Java runtimes and are lightweight compared to a Linux image.

Figure 2 illustrates the component architecture of the microservices and their interfaces with external clients and other internal microservices. This research has implemented a communication layer within the research component. When designing the solution, the microservices are divided into several internal components, including the API interface layer, the business logic and data access layer, and the communication layer. Through this segregation, it was able to separate the communication layer from the other layers. No changes were made to the API, business logic, or data access layers, as this implementation primarily focuses on the communication layer. Another reason for exclusively changing the communication layer is that modifying the API layer would have an impact on all other external microservice consumers. Hence, that would not be the best approach to introducing the new development. Most developers use HTTP client libraries to call each microservice and facilitate inter-service communication. This implementation has also followed the industry standard pattern and implemented the request/response-based stateless client. However, the client that has been implemented utilizes the pub/sub model and streams in the background, which are not visible to the developer. Similar to the HTTP client, developers can utilize the implemented library and progress through their development without encountering any complexity. To facilitate stream communication, Redis streams were utilized with the assistance of the Redis server. The Spring Boot Starter Data Redis library has been utilized at the program level to enable stream communication with the Redis server. This library is one of the most commonly used dependencies in the Spring Boot framework, providing high-level and low-level abstractions for integrating with the Redis server. The implemented communication layer consists of three main components: consumer, stream subscription handler, and

publisher. The consumer component is responsible for consuming the messages. In other words, this refers to receiving responses based on the subscription. The publisher is responsible for sending the messages to other microservices via the Redis server based on the stream key. The stream subscription handler manages all stream subscriptions.

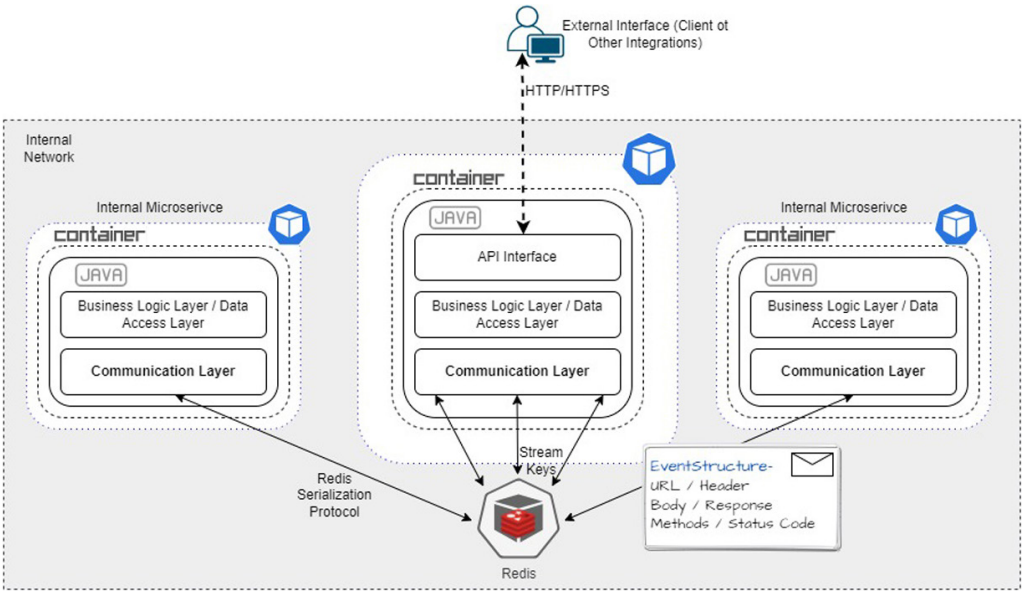


**Fig. 2.** Low-level component architecture

Figure 3 shows the request flow with the implemented solution. External applications or clients send HTTP/HTTPS requests by invoking the APIs exposed by the API request controller in the externally facing microservice. When microservice A is created, it establishes a Unix socket-based TCP connection with the Redis server, facilitated through the Kubernetes pods. After establishing a connection with the open TCP ports on the Redis and microservice pods, the connection remains active until the microservice pod is terminated. After the connection is established, subscriptions are created based on the stream keys, which are mapped to the services exposed in the microservices, similar to the HTTP services in the previous scenario. The stream key serves as an endpoint address in the HTTP communication model. This means that if a request needs to be sent to another microservice, the message must be published to the stream key subscribed to by the other microservice. The necessary business logic and other data access layers are executed based on the HTTP request received by Microservice A. Based on those outputs, Microservice A created an EventStructure object, which will be sent to other microservices as a message. The EventStructure object serves as a facilitator, encapsulating most of the characteristics of the HTTP request, including the HTTP method, headers, body, and other details. The EventStructure object is serialized and passed as a byte buffer over the established TCP connection using the RSEP protocol. In line with that, it was possible to minimize network consumption when transmitting messages across the network. Every EventStructure object receives a unique ID when the messages are stored as a callback reference to map the response.

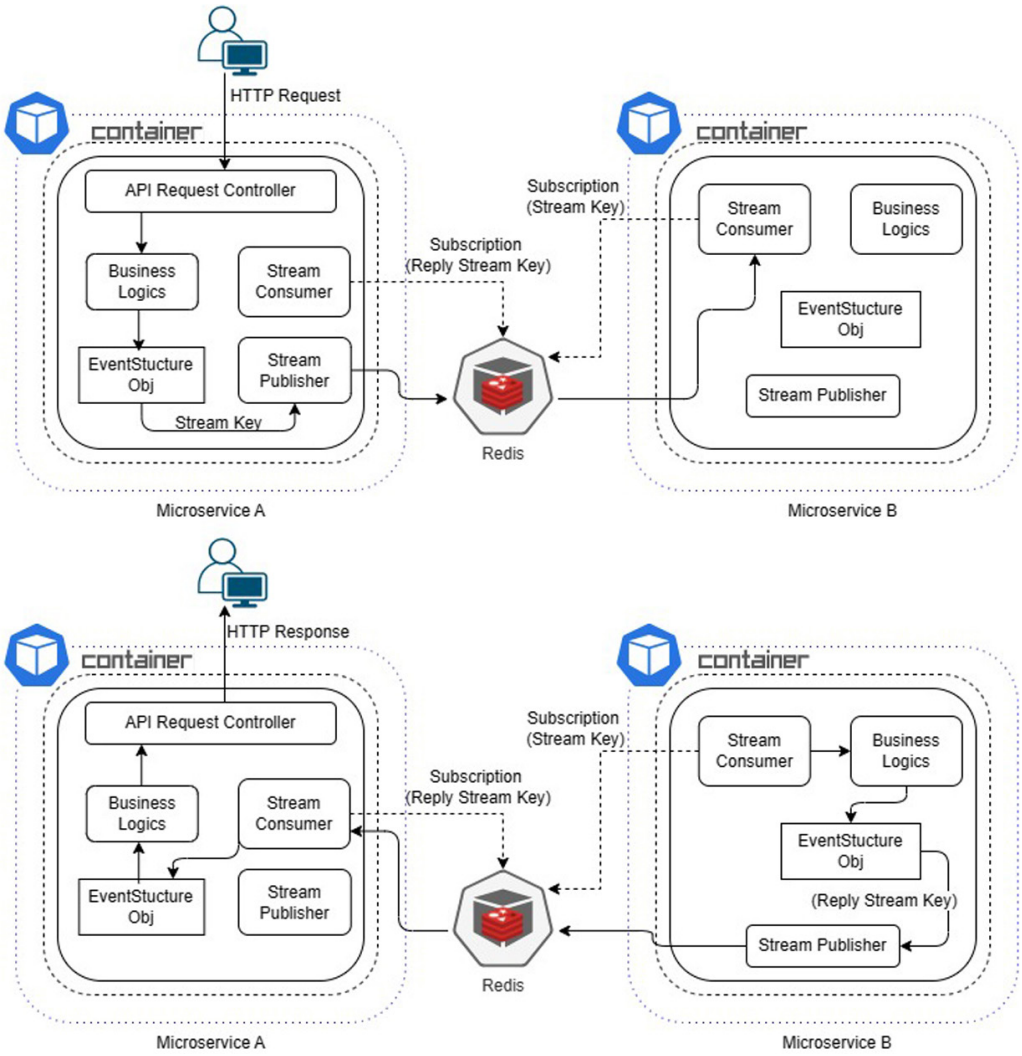


**Fig. 3.** Request/Response flows

Figure 3 illustrates the flow of the response in the implemented research component. According to the stream key, microservice B consumes the subscription messages from microservice A. The stream consumer component processes the message by mapping it back to the EventStructure object. The system then extracts message information, similar to how an HTTP request developer retrieves request metadata from the EventStructure object. Subsequently, Microservice B processes its business logic and prepares the response as an EventStructure object. The stream publisher retrieves the reply stream key from the received EventStructure object and sends the processed request to the corresponding stream. The stream consumer of microservice A processes this message, maps the response to the API request using the unique ID, and sends the response back to the client as an HTTP response.

All microservices and Redis servers are hosted on the Google Kubernetes Engine (GKE). Based on these findings, the literature review revealed that GKE is the most renowned Kubernetes engine in comparison to other Kubernetes services. Google is the first cloud provider to offer the Kubernetes engine for consumer use, and it offers more valuable features than other cloud services. The autopilot cluster mode has been chosen to create the Kubernetes cluster because GKE optimally manages cluster resources, making it more cost-effective. In the latest release, the container ID serves as the container runtime for Kubernetes pods. Kubernetes Ingress was

utilized to expose the API of the client-facing microservice (Microservice A) to external parties. In Kubernetes, an Ingress is an API object that controls external access to services within a cluster. It acts as a traffic controller, allowing inbound connections to reach internal services based on specified rules. The Ingress resource functions as a configuration layer that exposes HTTP and HTTPS routes outside the cluster to services within the cluster. Furthermore, the NodePort service is used to expose the TCP ports for communication between services in each microservice and the Redis server pod. In Kubernetes, a NodePort service provides a way to expose a service externally by assigning a specific port to all nodes within the cluster. NodePort services can be discovered using their port number and the IP address of any node in the cluster. The use of TCP connections enabled the transmission of messages via the RESP protocol. All pods are deployed as Kubernetes' "Deployment" type. In Kubernetes, a deployment refers to a resource object used to manage the deployment and scaling of microservices. Deployments offer a declarative method for defining and managing the desired state of application deployments in a cluster. They offer a higher-level abstraction that simplifies the management of replica sets and pods, enabling efficient scaling, rolling updates, and self-healing capabilities.

## 4.1 Quality attributes

Quality attributes are among the most crucial aspects of software development. Most decision-makers consider quality attributes when making decisions. Scalability, performance, availability, traceability, maintainability, and portability are the most critical quality attributes of microservice architecture [24]. The cloud-native solution enabled the achievement of the required quality attributes for the implemented research component.

**Scalability:** Scalability can be divided into vertical and horizontal scaling. Both types of scalabilities can be easily achieved with the provided cloud-native solution. The requested resource size and the maximum limit of resources can be defined for the microservice container at the pod level. The memory and CPU limits are defined in a way that allows for easy adjustment, enabling the allocated resources to be increased or decreased for specific microservices. Vertical scaling can be achieved either by using the kubectl command to invoke the Kubernetes APIs or through the GKE web console. Vertical scaling does not impact the logic of the implemented solution. However, it enhances the overall capabilities of the application. Horizontal scaling is very challenging when distributed systems are using the asynchronous communication pattern. The implemented solution utilized the Redis stream, which is responsible for delivering only one message to a single consumer. Horizontal scaling will result in multiple subscriptions for the same stream key, but Redis ensures that it delivers messages to only one consumer. The Redis stream originates from the EventStructure object to inform other microservices about the sender of the message. Therefore, based on that mechanism, the response can be sent to the correct recipient. Horizontal scaling can also be easily achieved by adjusting the replica count of the deployment using kubectl commands or the GKE web console. The Redis service includes service discovery and load balancing. As a result, the developer does not need to spend time considering it when scaling the applications. As a result, message duplication does not occur with this implementation, ensuring the exact delivery of the message.

**Availability:** The cloud-native solution is deployed in a Kubernetes environment, so if one of the pods is terminated, Kubernetes is responsible for promptly creating new pods. At the Kubernetes deployment level, the system can be scaled, replicate the pods, and ensure high availability for that microservice. Another scenario

involves maintaining high availability when implementing the new release. In the on-premises solution, we observed certain downtimes for the new release. However, the new cloud-native architecture is designed to implement a rolling update strategy for pushing new releases. Rolling updates is a default deployment strategy in Kubernetes. It involves gradually updating the microservice pods, one at a time, while keeping the remaining pods running. The rolling update process replaces old pods with new ones, ensuring that microservices remain accessible during deployment. In addition to the microservice, the research component includes the Redis server. Redis supports both the master-slave architecture and the Redis clustering solution. However, neither data sharding nor automatic partitioning is utilized to achieve improved solutions in master-slave architecture. This involves deploying multiple Redis pods using the Redis deployment, with one serving as the master Redis pod and the others as Redis slave pods. The master pod handles 'write operations', while slave instances replicate data from the master pod to handle "read operations."

**Traceability:** At the development level, logging is enabled using the log4j2 library, which is performed asynchronously. As a result, there is no overhead for the microservice application. The Redis level also provides real-time messages on the transport layer, which can be viewed using Redis GUI clients. By using those combinations, it was able to achieve end-to-end traceability. Support teams can easily identify issues by reviewing these options, ultimately minimizing the effort required to troubleshoot and resolve issues.

## 5 RESULTS AND DISCUSSION

This research primarily focuses on inter-service communication within the microservice architecture of the cloud-native environment. Based on this, the researcher has designed and implemented a solution that is supported for running in the Kubernetes environment as a cloud-native platform. The solution has demonstrated high performance in terms of response time and throughput compared to traditional methods. This research critically evaluates the implemented solution compared to the traditional inter-service communication method in a cloud-native environment. The traditional HTTP method and the implemented solution were deployed in the Google Kubernetes environment, and several test cases were executed to retrieve the data.

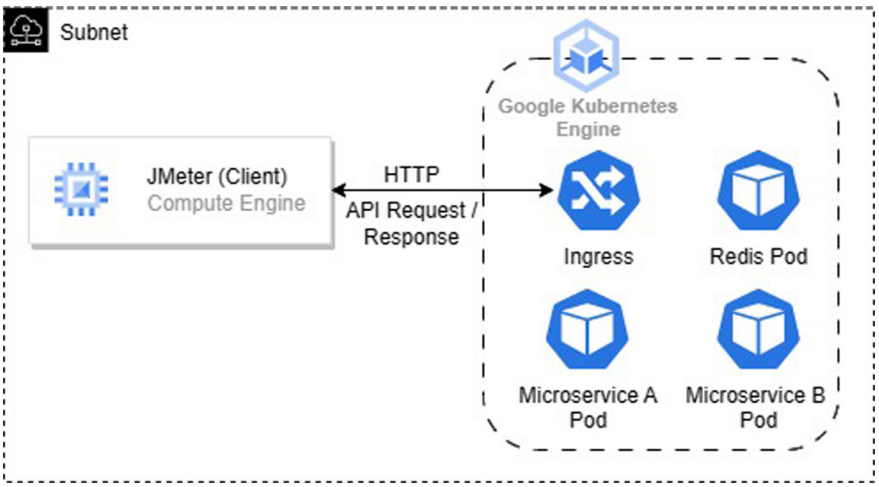


**Fig. 4.** Deployment architecture for testing

Figure 4 illustrates the deployment of components in the Google Cloud environment for testing purposes. Apache JMeter is used to generate traffic and functions as a third-party client or application. The literature review identified that JMeter had been used to generate loads in both academic and software industry testing. The Google Kubernetes Engine Autopilot has provisioned a Kubernetes cluster for deploying applications. The Nginx Ingress controller has been utilized to enable an external JMeter to access the services running within the Kubernetes cluster. The JMeter client and the Kubernetes cluster are located in the same subnet to minimize network latency. The JMeter has been deployed using the n1-standard-4 VM type, which includes 4 virtual CPUs and 15GB of memory. The following test cases have been executed to assess the system and its behavior.

**Table 1.** Executed test cases

| Test Case |
|---|
| Controlled throughput to 100TPS and sent GET / 1KB payload requests. Then checked the turnaround time difference between the implemented solution and the traditional HTTP method. |
| Controlled throughput to 200TPS and sent 1KB payload requests. Changed the Pods allocated CPU from 1 core to 4 cores and analyzed the turnaround and application response times with the implemented solution. |
| Controlled throughput to 200TPS and sent 1KB payload requests. Changed the Pods allocated Memory from 1 GB to 4 GB and analyzed the turnaround time and application response time with the implemented solution. |
| Controlled throughput to 200TPS and sent 1KB payload requests. Changed the Pod count from 1 to 4 and analyzed the turnaround time and application response time with the implemented solution. |

Table 1 lists all the test cases identified in this research to assess the suitability of the solution in a cloud-native environment. Each test scenario measured the turnaround time based on the logs, excluding any processing time. This data only includes the time taken for the request and response between Microservice A and Microservice B. Each test case was executed for one hour and repeated three times to identify the variations in the data. The logs were captured using the Google Logging service and then processed using Python scripts to analyze the data.

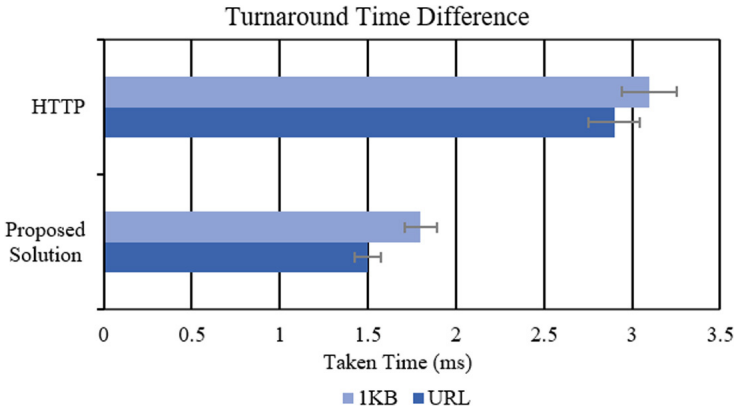


**Fig. 5.** Turnaround time graph

Figure 5 illustrates the turnaround time for traditional HTTP and the implemented solution methods. Spring Boot Rest Templates have been utilized to facilitate

inter-service communication among the microservices. RestTemplate is a class in the Spring Framework that simplifies making HTTP requests and handling responses in a restful manner. Most synchronous-based microservices use this library to facilitate inter-service communication. According to the graph, it is evident that the implemented solution has a shorter turnaround time compared to the traditional HTTP method. This indicates that with the implemented solution, microservice A will receive a response from microservice B in a shorter amount of time. In the implemented solution, there are no socket open and close activities as in the HTTP method. When transferring data through the network, it is serialized and passed as a byte buffer record, resulting in low network consumption. Due to these two reasons, the implemented solutions outperform the traditional method.

The test only considers communication between one microservice and another in real-world scenarios; several microservices need to communicate with each other to meet business requirements (e.g., Netflix) [25]. Although there is only a millisecond improvement in each microservice call and in communication between microservices, the overall response time will be significantly improved. The butterfly effect theory provides a better understanding of this phenomenon in the real world.

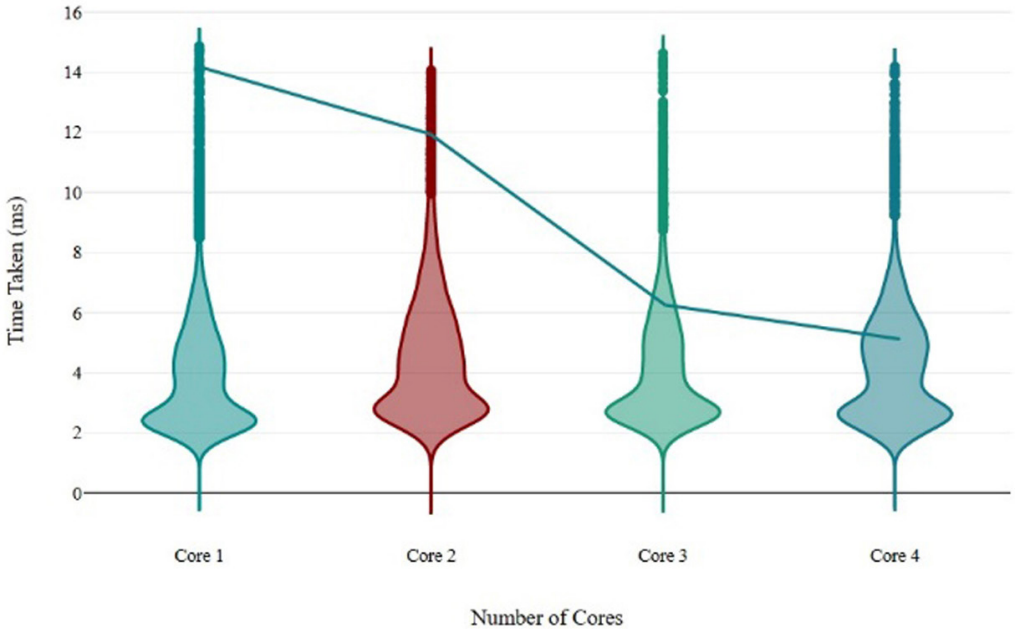


**Fig. 6.** Turnaround time and response time variation with cores graph

The violin and line plots (see Figure 6) display the turnaround time and average response time for each CPU core at the pod level. In this test case, an increased CPU is added at the pod level by editing the Kubernetes deployment YAML files, and the changes are applied to the Kubernetes cluster. The violin plot shows the full distribution of turnaround time for the CPU core. By analyzing the violin plots, it is evident that there is no significant change in the inter-service communication turnaround time when the CPU is added at the pod level. The line plot illustrates the degradation in overall application response time when more CPU is added at the pod level. This means that handling the business logic requires more CPU, so it varies based on the functions of the logic. Conclusively, inter-service communication is not affected by vertically scaling the pod's CPU cores. However, to increase the overall application performance, it is advisable to allocate the relevant CPU cores to the microservices.

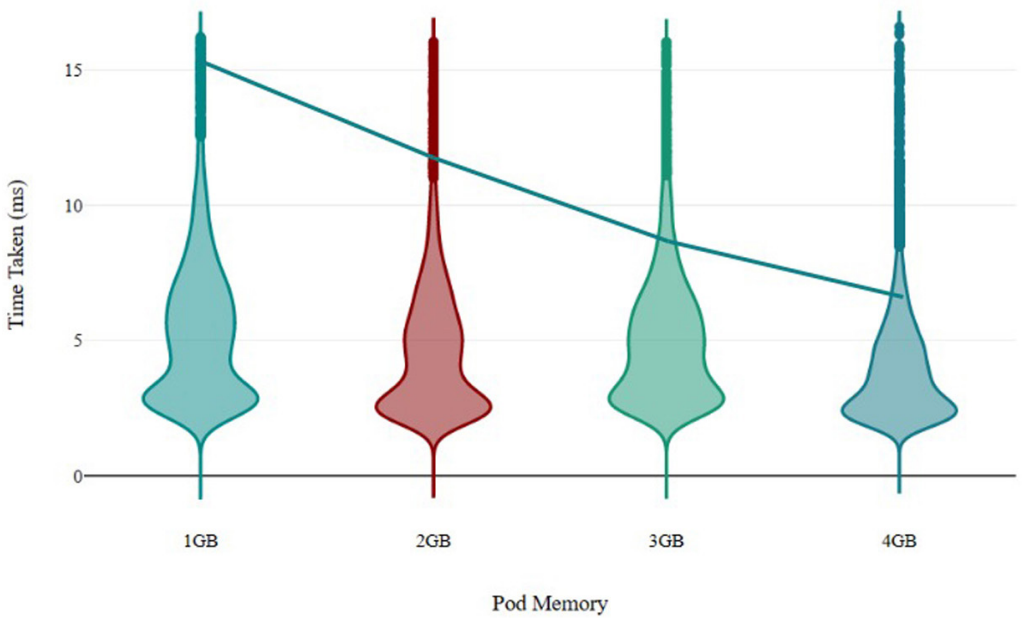


**Fig. 7.** Turnaround time and response time variation with memory graph

Figure 7 illustrates the relationship between the Microservice's turnaround time and the overall application response time as they vary with the pod memory. By examining the graph, it is evident that the high-frequency distribution of the violin plot remains unchanged when additional memory is added to the pod. This indicates that the pod memory does not affect the turnaround time. By examining the line plot, we can conclude that increasing the memory allocated to the pods improves the overall application response time. By allocating memory to the pod application, it will be able to process logic more efficiently with a higher memory allocation. As a result, the overall application response time has improved.

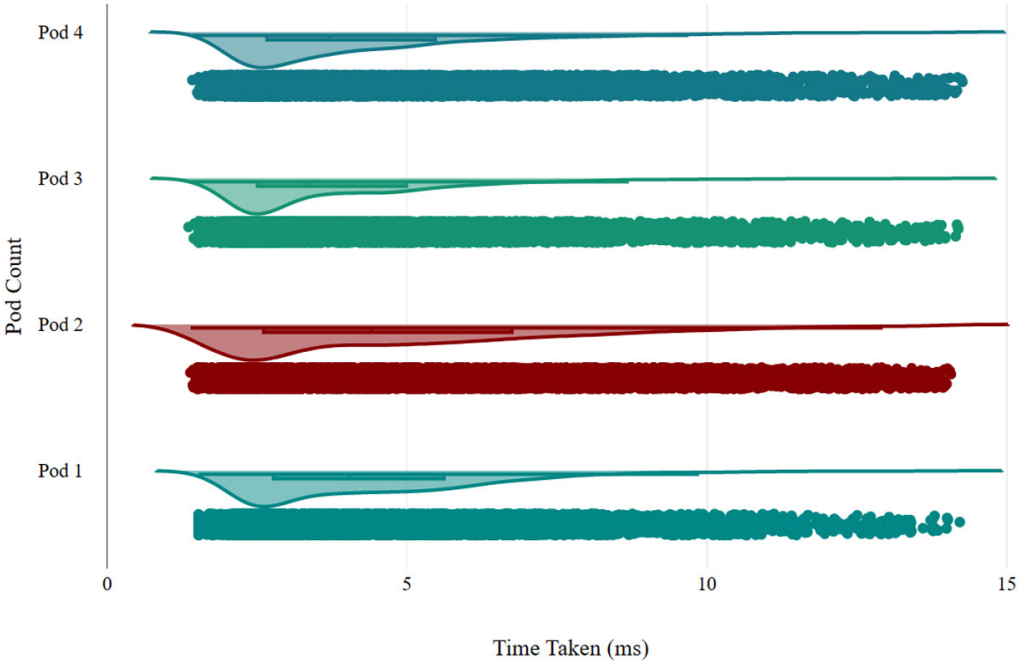


**Fig. 8.** Turnaround time variation with pod count graph

Figure 8, Raincloud Plot, illustrates the distribution of turnaround time when scaling the microservices pods. The previous vertical scaling does not impact the inter-service communication time of the microservice. However, it has impacted the overall response time. These test scenarios evaluate the horizontal scaling mechanism and check the distribution of the overall turnaround time in the inter-service communication between two microservices. According to the collected data, there is no significant deviation observed when adding the new pods to the cluster. This implies that there is no impact on inter-service communication when horizontal scaling is employed.

## 6 CONCLUSION AND FUTURE WORK

In contrast to monolithic applications, in a microservice architecture, all the services are deployed in a distributed environment and function as independent services. However, in order to meet business requirements, those microservices need to communicate with each other. Service calls experience additional latency when communicating over the network. Past research has shown that additional latency can degrade the overall application performance and increase the response time. Additionally, most of the current protocols experience delays in establishing and closing connections, as well as in sending and receiving responses. Furthermore, the transmission of large payloads exacerbates latency issues. Most microservice-based applications are currently being deployed in cloud-native environments.

This research primarily focuses on optimizing inter-service communication among microservices deployed in a cloud-native environment. By addressing the challenges of the cloud-native environment with Kubernetes, we have successfully implemented a solution that reduces latency during service-to-service communication compared to existing methods. The research utilizes the Redis Stream data structure and builds a message-passing system based on request-response interactions similar to the HTTP protocol. When the microservice initializes, it establishes a TCP-based socket connection to record the time it takes to open and close the connection. When transmitting payloads, they are serialized and sent as protocol buffers to minimize network resource usage. The responsibility of delivering messages accurately lies with the Redis server, which depends on subscriptions and the designated stream key. One limitation of the proposed model is that the deployment of the Redis server is a requirement for its implementation. The implemented solution can easily achieve scalability, availability, portability, and other cloud-native quality attributes. Extensive testing has conclusively proven that the implemented solution reduces latency in service-to-service communication and enhances the overall application response time in a cloud-native environment.

## 7 REFERENCES

[1] 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)*, 2021, pp. 137–144. https://doi.org/10.1109/SCSE53661.2021.9568289

[2] S. Weerasinghe and I. Perera, "Taxonomical classification and systematic review on microservices," *International Journal of Engineering Trends and Technology*, vol. 70, no. 3, pp. 222–233, 2022. https://doi.org/10.14445/22315381/IJETT-V70I3P225

[3] L. D. S. B. Weerasinghe and I. Perera, "Evaluating the inter-service communication on microservice architecture," in *2022 7th International Conference on Information Technology Research (ICITR)*, 2022, pp. 1–6. https://doi.org/10.1109/ICITR57877.2022.9992918

[4] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, "From monolithic to microservices: An experience report from the banking domain," *IEEE Software*, vol. 35, no. 3, pp. 50–55, 2018. https://doi.org/10.1109/MS.2018.2141026

[5] 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, 2018, pp. 000149–000154. https://doi.org/10.1109/CINTI.2018.8928192

[6] M. Seedat, Q. Abbas, N. Ahmad, and A. Amelio, "Systematic mapping of monolithic applications to microservices architecture," *arXiv*, 2023. https://doi.org/10.48550/arXiv.2309.03796

[7] N. S. Elgheriani and N. A. S. Ahmed, "Microservices vs. Monolithic architectures [The differential structure between two architectures]," *MINAR International Journal of Applied Sciences and Technology*, vol. 4, no. 3, pp. 500–514, 2022. http://dx.doi.org/10.47832/2717-8234.12.47

[8] N. Kratzke and P.-C. Quint, "Understanding cloud-native applications after 10 years of cloud computing – A systematic mapping study," *Journal of Systems and Software*, vol. 126, pp. 1–16, 2017. https://doi.org/10.1016/j.jss.2017.01.001

[9] Y. Mao, Y. Fu, S. Gu, S. Vhaduri, L. Cheng, and Q. Liu, "Resource management schemes for cloud-native platforms with computing containers of Docker and Kubernetes," *arXiv*, 2020. [Online]. Available: http://arxiv.org/abs/2010.10350; https://doi.org/10.36227/techrxiv.13146548.v1. [Accessed: May 1, 2023].

[10] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2011.

[11] N. Marathe, A. Gandhi, and J. M. Shah, "Docker Swarm and Kubernetes in cloud computing environment," in *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, 2019, pp. 179–184. https://doi.org/10.1109/ICOEI.2019.8862654

[12] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Kubernetes as an availability manager for microservice applications," *arXiv*, 2019. https://doi.org/10.48550/arXiv.1901.04946

[13] A. R. Sri Nandhini, A. Joseph, and S. Ajay, "Impact of implementing cloud native applications in replacement to on-Premise applications," *International Journal of Engineering Research and Technology*, vol. V9, no. 6, 2020. https://doi.org/10.17577/IJERTV9IS061021

[14] M. Saraswat and R. C. Tripathi, "Cloud computing: Comparison and analysis of cloud service providers-AWs, Microsoft and Google," in *2020 9th International Conference System Modeling and Advancement in Research Trends (SMART)*, Moradabad, India: IEEE, 2020, pp. 281–285. https://doi.org/10.1109/SMART50582.2020.9337100

[15] S. Raje, "Performance comparison of message queue methods," *UNLV Theses, Dissertations, Professional Papers, and Capstones*, vol. 3746, 2019. https://doi.org/10.34917/16076287

[16] R. Mara Jösch, "Managing microservices with a service mesh," Thesis, KTH, Royal Institute of Technology, 2020.

[17] "Service mesh: Architecture, concepts, and top 4 frameworks," Aqua. [Online]. Available: https://www.aquasec.com/cloud-native-academy/container-security/service-mesh/.

[18] M. Zand, "Review of pod-to-pod communications in Kubernetes," Superuser. [Online]. Available: https://superuser.openinfra.dev/articles/review-of-pod-to-pod-communications-in-kubernetes/. [Accessed: May 12, 2023].

[19] G. Turin, A. Borgarelli, S. Donetti, E. B. Johnsen, S. L. Tapia Tarifa, and F. Damiani, "A formal model of the Kubernetes container framework," in *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*, T. Margaria and B. Steffen, Eds., in Lecture Notes in Computer Science, Cham: Springer International Publishing, 2020, vol. 12476, pp. 558–577. https://doi.org/10.1007/978-3-030-61362-4_32

[20] P. Jain, "Kubernetes installation options | The hard way | Managed k8s," Cloud Training Program. Available: https://k21academy.com/docker-kubernetes/kubernetes-installation-options/

[21] "Google Kubernetes Engine (GKE)|Google Cloud," Available: https://cloud.google.com/kubernetes-engine

[22] "RESP protocol spec," Redis. Available: https://redis.io/docs/reference/protocol-spec/

[23] N. Levin, "How to build apps using Redis Streams," Available: https://www.academia.edu/40811333/How_to_Build_Apps_using_Redis_Streams

[24] T. Schirgi, *Architectural Quality Attributes for the Microservices of CaRE*, p. 46, 2021.

[25] I. Papapanagiotou, "Microservices at scale – Principles, tradeoffs & lessons learned," IEEE ComSoc Summer School.

## 8 AUTHORS

**Sidath 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 (E-mail: weerasingheldsb.20@uom.lk).

**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 AI architecture, software engineering, user experience and application development for bio-health research (E-mail: indika@cse.mrt.ac.lk).