National College of Ireland

Enhancing Microservice Performance: A Comparative Analysis of Thread & Fiber Model

MSc Research Project
Cloud Computing

Suraj Lumba
Student ID: 21195218

School of Computing
National College of Ireland

Supervisor : Diego Lugones

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	**Suraj Lumba**
Student ID:	**21195218**
Programme:	**Cloud Computing**
Year:	2023
Module:	**MSc Research Project**
Supervisor:	**Diego Lugones**
Submission Due Date:	14/08/2023
Project Title:	**Enhancing Microservice Performance: A Comparative Analysis of Thread & Fiber Model**
Word Count:	7461
Page Count:	24

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Suraj Lumba
Date:	18th September 2023

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Enhancing Microservice Performance: A Comparative Analysis of Thread & Fiber Model

Suraj Lumba
21195218

Abstract

Microservices in the today's world are becoming an essential paradigm to make a software system highly scalable, available as well as reliable by ensuring its maintainability and flexibility at the same time. The inherent difficulties and complexity of managing interactions across loosely linked services create the need for efficient microservices communication solutions. In order to guarantee smooth cooperation and coordination across services, microservices architectures need effective communication. The necessity for efficient microservices communication solutions is examined in this abstract, emphasizing the value of low latency in response time. Even if we talk about the real time processing, **Amazon** claimed that with the delay of **100 ms**, their sales get affected by **1%** Einav (2019). **Google** discovered a **20%** traffic loss for every additional **1/2 a second** of time in search page production Einav (2019). For instance, If a broker's computerized trading platform lags behind the competition by **5 milliseconds**, they forfeit **$4 million in revenue per millisecond** Einav (2019). We will carefully examine several studies that looked at the criteria for time delays in microservices and discuss many strategies which were carried out in earlier studies to reduce the response time delay. As an innovative approach, the thread management that is involved in the inter-process communication or say (IPC) on which every microservice has been implemented, shall now become thread-free. And this Thread-based management will be replaced by the Fiber-based management framework. As a consequence of this, it will effectively optimize the response time latency by just having fewer number of main threads multiplexed with multiple Fibers(Goroutines). These Goroutines are managed by Go run-time scheduler. The experimental evaluations discussed in this paper consists of 2 Implementation Models: Thread based Model and Fiber based Model, and will prove that Thread based model consumes **4X** time as a minima and almost **8.5X** time as a maxima in comparison to the Fiber based model. Consequently, the Microservice Architecture oriented with the Fiber based implementation will perform **6.25 times faster** than that of Thread based implementation.

Keywords- Inter-process communication(IPC), Microservices, Spring Boot Framework, Gin Framework, Golang, Fiber(Goroutines)

1 Introduction

Better microservices communication response times are more important than ever in the fast-paced digital world of today. User expectations for instantaneous and smooth inter-

actions with applications are rising at a massive rate, and any delays drive users away or cause companies monetary losses. Any increase in the volume of requests made for communication between the services, whether within the container package or between actual servers connected by a network, results in an increase in latency, especially when the services are virtualized, which is actually a serious problem. Overall, latency is influenced by a number of variables, including: the quantity of request calls, network delay, time on wire, the service availability, the processing time, and the variation in demand or load. Lower response times are crucial to guarantee seamless and uninterrupted experiences for applications that operate in real time, such as performing any fetch operation to gather any information say financial information, or any personal data, collaboration tools and post operations like persisting image data over cloud server via microservice.

Additionally, as mobile phones and the Internet of Things, or IoT, gain in popularity, so does the requirement for microservices designs that can manage a sudden surge of concurrent requests from many sources **Shadija et al. (2017)**. Communication lags between microservices can become more pronounced in these distributed systems, which can cause a chain reaction of performance problems. The ability to manage large traffic loads effectively, keep systems responsive, and adhere to service-level agreements are all made possible by achieving improved reaction times.

Even though it is true that every new technology has its own drawbacks, if we critically examine the drawbacks of moving from the current application-specific infrastructure to the microservices architecture, we may need to deal with common problems that may arise during this phase of migration. The first step will be to dissociate from the monolithic, old application infrastructure. Second, every record in the old database system will be partitioned, with the hope that each separate micro-service would utilize access to its own unique database. Then once all of these tasks are completed in the realm of migration, the most delicate aspect of implementing a micro-service architecture is the communication between services **Taibi et al. (2017)**.

In the context of a virtualized environment, the primary emphasis of our study will be particularly on the delay in the time taken to respond corresponding to requests made of one service to another service. In this architecture, remote procedure calls (RPCs) are frequently used for communication since they make it simple for one service to request information or a feature from a different service across the cloud **Benni et al. (2020)**.

Through a RPC, or remote procedure calls, a client asks a server to carry out a certain action or function. The request is acknowledged by the server, which then performs the required action and sends the client the outcome. There are two types of RPC requests: synchronous and asynchronous. Before continuing with any synchronous RPC tasks, the end user awaits for the server to respond. In asynchronous RPC, the client sends the request and then continues with other tasks while waiting for the server to respond **Suljkanović et al. (2022)**.

The objective of this research is to lower down the response time latency of each individual microservice interacting with the applications or other microservices, in a way to enhance their performance.

When considering the influence of network latency, it is important to note that inter-process communication has a significant impact on the response time of each service in relation to each service request. The reason for investigating delay in communication among threads in microservices is the requirement to boost the performance and scalability of microservices-based systems. Representational State Transfer abbreviated as a REST API, Remote Procedure Calls abbreviated as a RPC and the message queues are just a few of the protocols used by services to communicate with other services in a microservices architecture. Since it influences both the system's overall latency and throughput, IPC is a crucial component of this communication. When threads in different microservices have to interact with one another, the IPC technology used incurs a cost that can have a significant impact on the system's overall effectiveness. **Shafabakhsh et al. (2020)**

1.1 Motivation

People as well as the researchers evaluated and provided many effective ways to optimize the communication among the microservices in a way to improve the response time latency for every individual request. There has been different optimized techniques as well as the algorithms involved in improving the time latency among the microservices. A research improved latency by making up the optimal combination of dimensionality reduction and optimization techniques. Another research proposed a new framework mutune which could extract the IPC communication from the application code itself and managing it on its own.

There has been another research that emphasized over the Thread implementation process which involves inter-process communication, which could be a major reason to worry when there comes the latency as a parameter in measuring the performance of any Microservice.

But so far, the results of every research undoubtedly improved but they were not consistent enough when it comes to heavy requests payload. This is just because the frameworks used which are built and developed on top of the language uses inter-process communication(IPC) in which every task is taken care by an individual application thread. This application thread associates to the kernel of the operating system(OS) via OS-level threads.

So when an application requests a microservice for any operation(fetch/post), as an initial step, a thread is cloned from the thread pool and associates application thread to the OS via kernel, which is also called as OS-level Thread. The main thread in the system takes care of this **1:1** mapping among helper thread and application thread. **Samsad (2021)**

Thread communication among the microservices is one of the major reasons of causing latency in response time of each microservice in the languages like PHP, Node or Java.

1.2 Research Question

Can we eliminate the root cause of the problem that is, latency in response time of microservices by eliminating the Thread scheduling of processes/tasks and propose a thread-less way for task scheduling to significantly reduce the response time latency?

1.3 Solution Overview

The solution that we are going to discuss further in this research will involve removal of Threads that are involved in background of several task execution and process completion of any application. This functionality shall be overtaken by Fibers. Fiber makes use of the lightweight concurrency mechanism known as Goroutines in Golang to decrease latency **Robert Griesemer and Thompson (2009)**.

Goroutines allow Fiber to effectively manage a high number of concurrent requests, cutting down on the amount of time it takes to respond. In order to further improve performance and speed up response times, Fiber also offers middle-ware features, enabling developers to incorporate caching, request/response logging, as well as rate restriction **Shóstak (2021)**.

Additionally, Fiber places the least amount of cost possible on microservices because of its performance-focused basic architecture. As a result, requests are processed more quickly and there are less network delays, which improves the microservices architecture's overall responsiveness. Developers may efficiently address latency concerns in Golang microservices by using Fiber, which will enhance the scalability, stability, and user experience of their distributed systems.

We shall see the complete model based on Fiber in section of Methodology and will understand how it is more effective than Thread based Implemented Frameworks.

2 Related Work

The quality of service (QoS) for consumers is significantly impacted by the response time/delay associated with cloud microservices. The significance of comprehending the variables that affect the time of response associated with cloud microservices is highlighted by a review of the research that has already been published. This review of the literature offers a thorough examination of the studies on the response time delay associated with cloud microservices, highlighting their major conclusions, methods, and limitations.

2.1 Ideal Configuration of Microservices

The response time in the worst case is measured by tail latency, which is a crucial performance statistic for microservices. Over the past few years, there has been an increase in interest in tail latency reduction as a way to enhance microservices' overall service quality. By changing the configuration settings of microservices to surge their performance, or "optimal configuration tuning," one method for lowering tail latency can be achieved. Optimal configuration tweaking for lowering tail latency in microservices has

been studied in the literature in this work.**Somashekar et al. (2022)**

The performance metric tail latency, which measures the worst-case customer response time, is essential for microservices. Tail latency may plummets down by optimal configuration tuning method, which comprises changing the configuration parameters for the microservices to enhance their efficiency. A study of multiple joint optimization methodologies revealed that configuration change may significantly reduce tail latency by about 46% **Somashekar et al. (2022)**.

2.2 Resource-effective Microservice Provisioning

As in the aforementioned literature work, we saw how a microservice's configuration may be tuned effectively to cut tail latency in half. The literature that supports successful resource provisioning methodologies for microservices will be discussed in this section. While maintaining a high level of service quality, Automan, a resource provisioning system, may greatly increase resource consumption. This research supports our claim that it outperforms other cutting-edge resource provisioning frameworks since an experimental finding indicates that it can conserve up to 49.6% and 29.1%, respectively, of CPU and memory resources while still achieving the same end-to-end tail latency objective **Cai et al. (2023)**.

What we observed is, this framework is still being improved as part of ongoing development because it cannot yet function with the horizontal scaling method.

2.3 Effective Microservice Load Balancing

One crucial area to concentrate on is load balancing, which entails dividing incoming network traffic among several microservice instances in order to increase overall throughput, application scalability, and the response time. The literature contrasts the performance of two alternative algorithms, the fundamental Round Robin Algorithm and the Equally Spread Current Execution (ECSE) algorithm. After examining and contrasting the results of two other approaches, it was discovered that the ESCE algorithm solution is simple, efficient, and offers a 70% improvement in total microservice response time **Agavane (2020)**.

This study using the ECSE Algorithm revealed through experimental assessment that it is significantly more efficient than the standard Round Robin Algorithm.

2.4 Optimization of Service Chains Latency

2.4.1 D3G Algorithm Approach

When many services join to meet a user request, the concept of service chains latency is raised. This component forms the basis of the research article that will be reviewed. They put forth a method for improving overall delay for chained applications while guaranteeing workload equity, dubbed (D3G), for cloud-based service chains. It establishes two server queues, first for the interactive type requests and the other for non-interactive type requests, and makes sure that the delay is never more than the user's tolerance level **Zhang et al. (2018)**. The service chains' latency was generally reduced by 35%

according to their evaluations.

However, their study did not even name the range for the same, this technique lacked the ability to describe the precise values of the user's tolerance value.

2.4.2 Dynamic Priority based Approach

Here in this study piece, the Dynamic Priority based Weighted Scheduling Algorithm or say (DPWS) Algorithm proved that the service-chain requests latency can be readily lowered in a way that is effective. As we observed above in the previous method, the User's tolerance value was not known, which called into doubt the usefulness of the D3G Algorithm **Zhang et al. (2018)**.

It divides the service into two categories: a leaf-level service, that is defined as a client's direct request to the individual service like an application calling a service directly for operation, and a chained-level service, which is a situation when a service gets requested by another service. It then assigns each category to a different queue with a different priority (1 is always assigned to the leaf-level service and 1 is always incremented every time it calls another service in a chain), and also the weighted fair queues are utilized in a way to dynamically improve the prioritization **Xu and Shang (2019)**.

2.5 Appropriate Protocol Characterization for Microservices

An ideal protocol must be employed to guarantee effective communication with decreased time latency and interactivity amongst microservices. The researchers looked at and enhanced inter-microservice communication during the study examination of this literature on just the single host. They examined REST, gRPC, and Thrift, three widely used communication protocols, for network, memory, CPU utilization, and response time. According to the evaluations, Thrift performed best for communication across microservices since communication packets could be serialized and deserialized quickly. gRPC is the second-best performance thanks to its rapid protocol buffers. REST, a programming methodology used for client-server architectural designs, is not the best choice for backend microservice communication, nevertheless **Kumar et al. (2021)**.

However, even on choosing an effective protocol for usage, the results do show improvement, and may help in optimizing the latency but might not show a higher difference in reducing the latency in the time of response.

2.6 Migration from Cloud to the Edge

Moving portions of an application's microservices from a centralized cloud infrastructure to edge nodes or devices is referred to as "moving microservices from the cloud to the edge." IoT gadgets, mobile devices, local servers, and other equipment that are close to end users are referred to as the edge. Because it increases the effectiveness of development, the microservice approach has been adapted from cloud services. This essay explores several architectural styles, including: a microservice which was originally meant for operation on the cloud is being downshifted.

The microservice must be moved to edge devices, though, in order to improve response time/latency to the storefronts and meet customer privacy concerns. The other architectural design pattern is called Edge to Cloud Load Balancing, and it describes how client requests are dynamically routed to either the cloud or the edge of the network to

ensure that microservices have the best run-time performance **Li et al. (2018)**.

Why couldn't they optimize the service infrastructure over the cloud rather than moving the service down to edge devices or local servers?

2.7 Serverless mechanism for latency-sensitive Microservices

Well, the researchers in this study designed a serverless mechanism in a way to improve the latency of response time of microservices. They introduced Nightcore, a serverless function run-time that offers container-based isolation between functions with microsecond-scale overheads. As per this study, the architecture of Nightcore takes into account a number of microsecond-scale overheads, such as the scheduling of function requests, primitives for communication, threading models for I/O, and concurrent function executions. As per this function, for now only Python, Go, NodeJS, and C/C++ serverless functions are supported by Nightcore. Their analysis demonstrated that Nightcore can achieve 1.36–2.93 better throughput and up to 69% less tail latency while executing latency-sensitive interactive microservices **Jia and Witchel (2021)**.

2.8 Technologies for Service Discovery for Microservices

There are various service discovery technologies that may be integrated with microservice applications to improve throughput, performance, and perhaps shorten the microservices' response times. In their test scenario, the authors of this study compared two microservices systems built on several service discovery technologies, including Eureka and Consul. The outcomes of the experiment proved that the microservices application utilizing Consul service discovery performed better than the application utilizing Eureka service discovery technology in terms of throughput and the amount of handled requests per second, with a 4 % improvement when using Consul **Al-Debagy and Martinek (2018)**.

As a result, Consul-based service discovery tools for microservices applications may be preferred to Eureka-based ones for microservices. This is also considered as one of the ways of a followed design pattern for implementing microservices architecture.

2.9 Microservices Auto-scaling

The method of dynamically altering a microservice's instance count according on the level of demand for its offerings. By using this strategy, it is made sure that the microservice can manage traffic variations without becoming overloaded or underused. In their article, the authors employed an unbiased technique to automatically scale microservices in cloud-oriented applications.

They ran an algorithm using Google Kubernetes Engine that was implemented using Kubernetes HPA. Reducing time of reaction at the service level was an essential QoS need, and it was implemented on a Twitter analytics program for disaster management. When it is compared to the default auto-scaling paradigm, the method's first testing on a variety of microservice demands showed a 20% reduction in microservice response time **Abdel Khaleq and Ra (2019)**. They concentrated on memory-intensive and high-traffic services.

In this study, the relevance of the Google Kubernetes Engine was briefly explored. Their evaluation led to improved service auto-scaling, which in turn reduced response time.

2.10 Express-Lane Scheduling for Microservices

The hardware-based scheduling of a Simultaneously Multi-Threaded (SMT) core is extended in this study to offer a "express-lane" execution environment for short activities, preventing them from waiting behind infrequent, lengthy ones. This is known as Express-Lane SMT (ESMT).

In order to avoid Head-of-Line (HoL) blocking, ESMT preempts tasks as they approach predetermined service cutoffs and migrates them to the following queue to be served by the forthcoming SMT execution lane. In order to increase usage, they also suggested an improved version of ESMT that enables execution lanes to collaborate with one another. According to their analysis, ESMT with work stealing lowers tail latency under moderately (40%) and heavy (70%) system loads by a mean of 56% and 67%, respectively, compared to a normal SMT core **Mirhosseini et al. (2019)**.

2.11 Auto-tuned Thread Mechanism for Microservices

We learned that threads are a crucial component of an application's inter-microservice communication in the section of the thorough introduction. The intricate interactions among several threads and the various resource requirements of various microservices make it challenging to develop effective threading models for microservices. Auto-tuning approaches have become a potential method for improving threading models in microservices to meet this difficulty. The researchers in this study introduced µTune, a framework for automatically tweaking threading models in microservices that isolates the threading paradigm's implementation from the service/application code. It also offers quick load shift detection for efficient dynamic flexibility. Without the requirement for thread creation, removal, or administrative overhead, it also manages thread pool sizing.

This study showed that by separating the thread model based implementation from the application programming in a separate framework, load-optimal threading models might minimize tail latency up to 1.9x **Sriraman and Wenisch (2018)**.

2.12 Matrix-based Performance Evaluation: Java and Go

Researchers in this study created straightforward matrix multiplication algorithms in Go and Java to examine concurrency aspects. Go's version makes use of Channel and Goroutine, while Java's makes use of Java Thread. They conducted a matrix multiplication experiment. Go outperformed Java in the experiment in terms of concurrency and compile time.

Go code also demonstrates how simple concurrent programming is. They demonstrated that Java does offer robust and interactive multi-threading features, but their development costs are rather high. As per their evaluations, They found that a typical matrix multiplication required 0.670 milliseconds to build whereas Golang required just

0.214 milliseconds, demonstrating that Go is 3x faster than Java **Togashi and Klyuev (2014)**.

2.13 Manipulating Threads Mechanism for Microservices

We are aware that thread management is crucial for communication between microservices because service requests are nothing more than threads that carry out their specific tasks by sending requests, waiting for microservice in a way to complete its business logic, and then sending back a response. The researchers did discuss that the Thread implementation is also a major reason for higher value of response time which ultimately is a solemn reason for higher latency.

However, their research did not yield practical effective assessments, but their conclusion gives an idea that there may be a greater chance of lowering the microservice's response time latency if the thread management could be substituted in the manner they suggested **Eyerman and Hur (2022)**.

Research Topic & Author	Base of Research	Improvement in the factor of Response Time Latency
Configuring parameters of Microservice by **Somashekhar and Suresh (2022)**	Response time latency may get reduced by an optimal configuration tuning method, which involves modifying the configuration attributes for the microservices to improve their efficiency.	Reduced the latency by 46%.
Effective Load Balancing Technique by **Prashant Bharat Agawane (2020)**	The literature distinguishes the performance of two alternative algorithms, the basic Round Robin(RR) Algorithm, and the one that the researcher implements as a part of solution which is, Equally Spread Current Execution (ECSE) algorithm. After evaluating and comparing the results of both showed impressive results.	Offered almost 70% improvement in total response time of microservice.
Optimization of Service Chain Latency by **Yuchao Zhang (2018)**	It sets up two different server queues, one for handling the interactive requests and the second one for non-interactive requests, their research analysis made sure that the latency could never be more than a User Tolerance figure.	Evaluations showed that service chain latency of response reduced by 35%
Serverless Mechanism for Microservices by **Jia & Emmett (2021)**	The researchers proposed Nightcore, a novel serverless function run-time which offered container-based isolation and which can consider a range of microsecond-scale overheads.	Reduced the tail latency of microservice interactions by 69%
Service Discovery Design pattern implementation for microservice architecture by **Omar and Peter (2018)**	the researchers in this paper compared two microservices systems that are built on many service discovery mechanism, inclusive of Eureka and Consul.	Consul attained a better throughput in comparison to the Eureka of about 4% to reduce latency of response time
Auto-scaling of Microservices by **Abdel and Ra (2019)**	Researchers in this paper executed an algorithm using Kubernetes Engine powered by Google which was put into action using Kubernetes HPA.	showed 20% of reduction in microservices response time
Auto-tuned Thread Mechanism by **Sriraman and Wenisch (2018)**	The authors of this paper presented µTune, a framework for automatically adjusting threading models in microservices that is responsible for decoupling the implementation of the threading paradigm from the service application code.	threading models which are specifically load-optimal demonstrated minimization in tail latency up to 1.9x

Figure 1: **Tabular Representation of Related Works performed and their Performance Achievement**

2.14 Contribution of the Research

According to the aforementioned literature reviews for different approaches, as well as the corresponding evaluations and experimental analyses, we have seen that all of them have the same goal of reducing the reaction time/latency of a microservice.

Here in this paper, I would like to perform an investigation by adding my own set of research, which includes an appropriate experimental analysis and exact evaluations by substituting the Thread management in the microservice architecture in a programmatic approach for Application programming Interface(API) calls involved in inter-microservice communication in the hope that this evaluation in the near times may reduce the response time latency in the inter-microservice communication.

3 Methodology

As we saw in the previous section, various works of literature have been found to manipulate a variety of factors to improve the latency of a microservice's response time. As a result, some of these works have been found to improve performance and, as a result, to reduce latency up to a particular factor based on the relevant experimental analysis. Here, I want to conduct a critical examination of an Remote Procedure call's central role in microservice communication.

As we have seen, a microservice's cloud intercommunication is exclusively implemented through threads. It is a thread that forwards a service request to the microservice, processes it, and then returns the result on the same thread. Despite the fact that they overlap each other's waiting times, threads uphold greater levels of concurrency in interprocess communication when there are very fewer/limited service requests in demand, that too, which are light-weighted, allowing for the use of parallelism and a consequent decrease in response time latency.

However, a bottleneck situation occurs in the microservice architecture which involves Thread scheduling for each request, that results in plummeting peak throughput and lengthening response times when any of the following cases occurs:

CASE 1. First when the number of requests becomes way too higher in number.
CASE 2. Second, when there comes the request with a massive payload.

Due to the fact that thread management in such situations involves both serialization and synchronization, the creation of heavily loaded threads, scheduling them from the thread pool, and terminating them can limit the throughput of the RPC call to the service.

As shown above in **Figure 1**, we see that there are three different microservices involved for individual task to perform as per the request invoked from the application or any other service.

Out of them, there is one which involves a **POST** as well as a **FETCH** operation and includes a **heavy payload(an image object)** as a part of request which relates very well with the CASE 2 as discussed above. We will consider this case as a core experiment of our research in which we shall first calculate the span of time of the complete cycle of

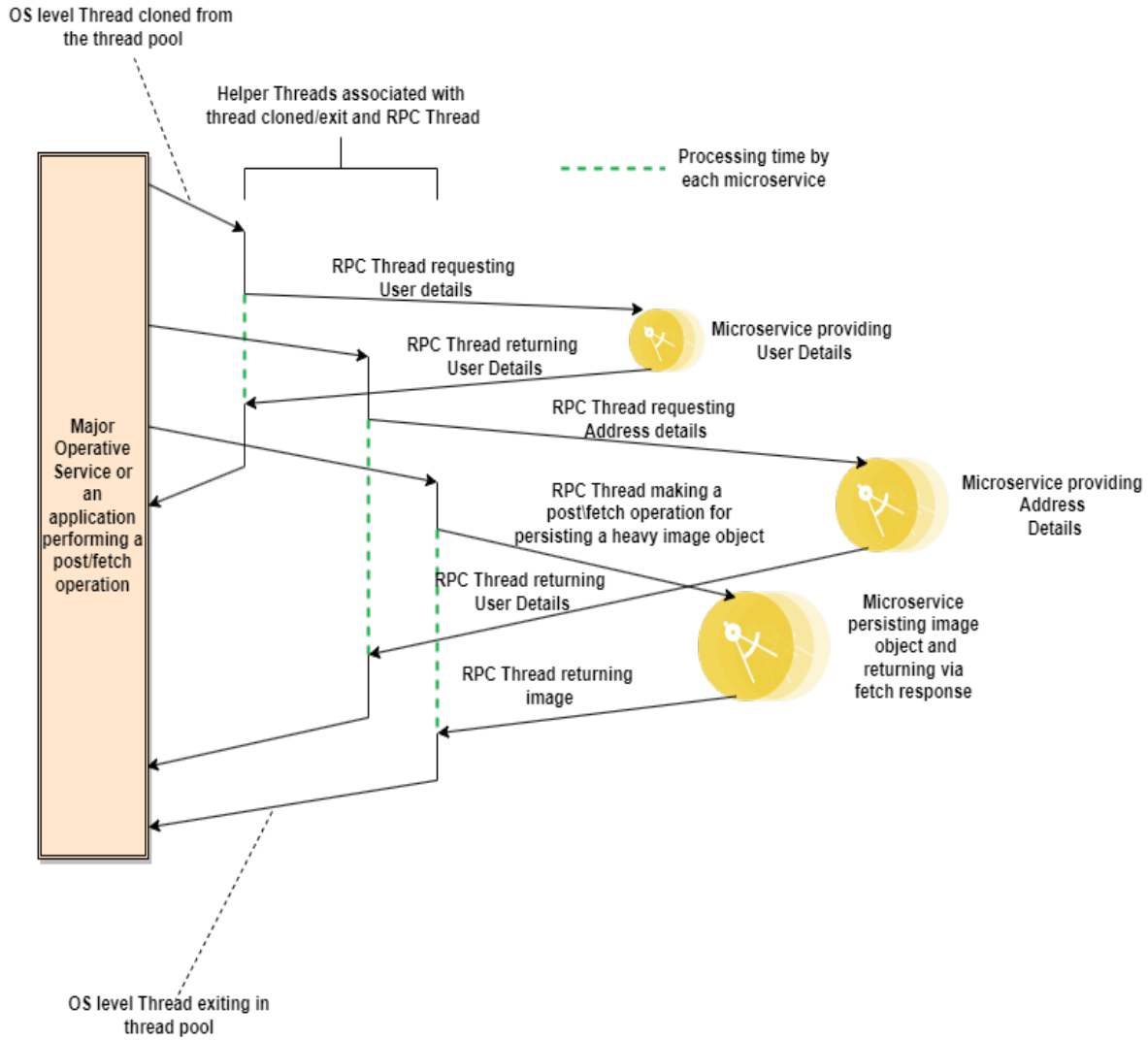

Figure 2: **Thread calls for different services in a Microservice Architecture**

a request, where the total time taken will be calculated as :

$$T_t = T_r + T_p + T_f$$

The above formulated terms are explained below:
T(t) = Total time taken by request
T(r) = Time taken when the request is invoked and reaches microservice
T(p) = Time taken by the Microservice to process the request
T(f) = Time taken when microservice sends back the response

And this shall be tested with different sizes of the image object which will be passed as a part of POST request in a way to make the request payload heavy, and then, the total time will be calculated for each request, from the time when the request shall be invoked till the time we get the FETCH response back after persistence and processing of it.

3.1 Research Procedure

In this section, we will present a way to improve the response time delay of a microservice. For better perception, we shall be needing two different models for comparison and to prove the efficacy of our proposed model.

3.1.1 Thread Management-based Model

This model is an existing concurrency model which is utilized in computer systems since many years to manage as well as schedule tasks. Our experiment will first have different sizes of image objects for persistence as a part of request payload and then the request shall be invoked as a part of a microservice involving Inter-process communication of Thread. This process of experimentation shall be consuming an increased amount of time which consequently will result in increased amount of latency in the response time. It is indeed, due to the fact that when the application invokes the request, 2 different threads get cloned.

First, an application thread which will do the following:

1. Interaction with the microservice.
2. Processing of the request.
3. And carrying the response back after processing the request.

Second, the **application thread** which shall make the **kernel calls** associated with the OS level thread invoked from Thread pool. In our case, there occurs 1:1 mapping between the OS level Thread and Application Thread. There could be more amount of threads associated with this main Thread or OS Thread as per the scheduling of the tasks involved to perform any operation.

So at the back, when the request is invoked, at the same time, apart from application thread, an OS level thread gets cloned from the Thread pool. Then the costly application thread created for the operation takes the request to the microservice for processing where the remote procedure gets executed and sends back the response via application thread. On receiving this response, the OS level Thread associated with the application thread does exit from the Thread pool.

For better understanding please refer the step by step representation of the Thread management based communication in the Figure 2 shown below.

In a way to prove the tail latency as per the aforementioned point, for this experiment we will have an application as a front-end interface which will communicate with the microservice responsible for the processing and persisting of the image as a part of POST operation and sending back the persisted image as a part of FETCH operation to the application. So there shall be two components of this experiment where first component, our application will interact with the microservice, and the other one shall be the microservice whose endpoints will be exposed in a way interact with the application or other microservices as per the requirement.

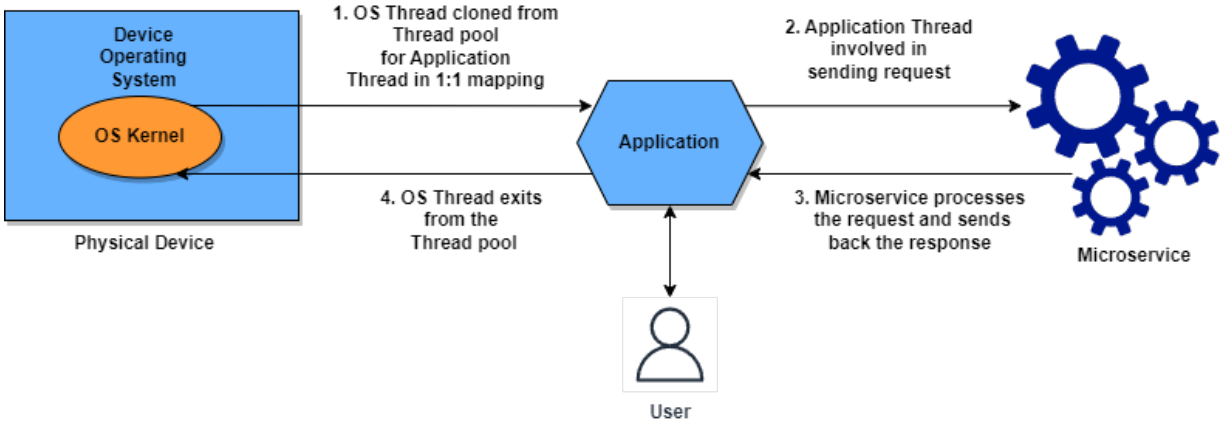

Figure 3: **Step by step representation of Thread Management-based communication in backend**

3.1.2 Fiber(Go-routines) Management-based Model

Here I present a model as a part of an effective solution to the above problematic model. This model is highly based of the fibers(Goroutines) instead of Threads, which are involved in scheduling, managing, as well as processing of tasks. But before we get into the understanding of this model, let us first understand that what Goroutines are.

Goroutines are lightweight concurrent functions in Go (Golang) that can be executed concurrently with other Goroutines. They are managed by the Go run-time scheduler and enable efficient concurrent programming. Goroutines communicate using channels, allowing for synchronized data exchange and simplifying the development of concurrent applications **Vogel (2022)**.

Hereon, a question arises that how these Goroutines are more efficient than Threads? Goroutines are more efficient than traditional threads because they are managed by the Go run-time scheduler, which uses a lighter memory footprint and supports concurrent execution on a smaller stack size, just 2KB of stack space is sufficient to create a Goroutine. Goroutines have low overhead, making their creation and management faster than that of Threads. Additionally, Go's built-in scheduler handles multiplexing Goroutines onto available OS threads, optimizing CPU usage and minimizing contention, leading to better performance in concurrent scenarios.

Goroutines are not mapped to OS threads directly. Instead, Goroutines are scheduled onto a pool of OS threads that are managed by the Go run-time scheduler. This scheduler handles this scheduling by multiplexing the execution of Goroutines across a few OS threads **Vogel (2022)**.

During the running state, when the Goroutine waits for an event, then Go run-time scheduler transfers that Goroutine that is blocking/waiting (for instance, because of I/O operations or channel synchronization) off the OS thread so that other Goroutines that are ready to run can run on that thread instead **Vogel (2022)**. Through a concept called "Goroutine Preemption," which provides effective multitasking and permits mul-

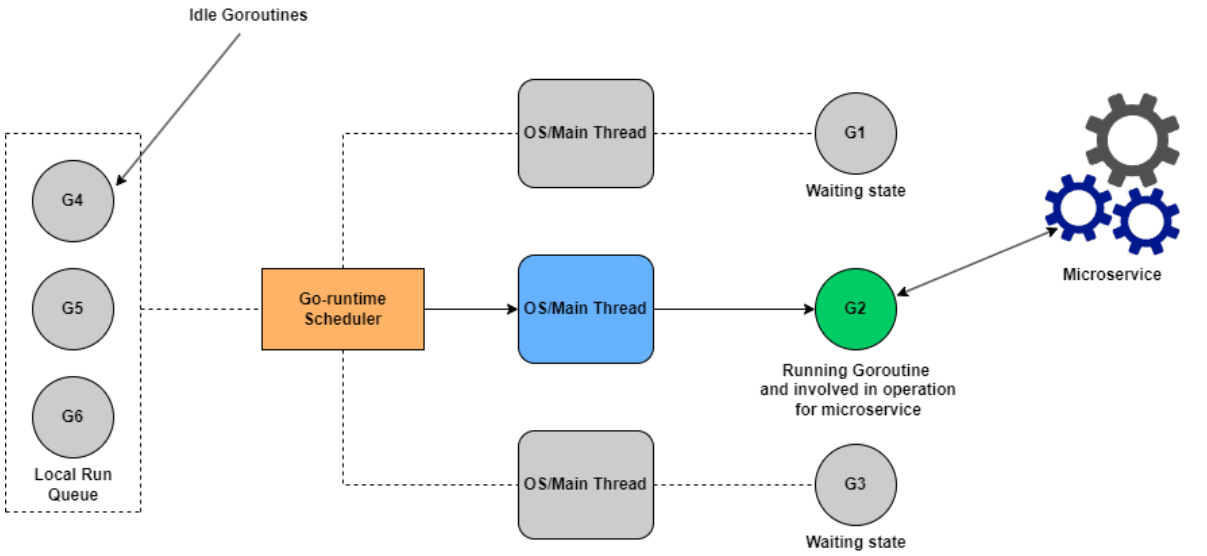

Figure 4: **Fiber(Goroutine) Management based Architecture**

tiple Goroutines to share fewer OS threads, this ultimately plummets the memory usage and increases the speed. As shown in the figure 3, **G2** is the running Goroutine which is aligned with the Main Thread of the system and is responsible for scheduling, managing as well as processing of the task which gets assigned as a part of the request invocation of the application to the microservice.

NOTE: As shown in Figure 3, the Goroutines **G1** and **G3** are in their waiting state and we have **G2** in its running state which is aligned with the main thread of the application. In case, if **G2** requests any input/output event which consequently will force **G2** to go to its waiting state, then any of the idle Goroutines from the local run queue shall be assigned to the main thread for further execution of the task.

3.2 Critical Comparison between Threads & Fibers

As discussed above, in the context of microservices, both Fibers as well as Threads plays an essential role in managing concurrency as well as the responsiveness. Their differences depend upon certain characteristics which we will discuss below.

3.2.1 Overhead on Resource

The scheduling of both the models differs. For Fibers, the application-specific run time scheduler is responsible for managing the Fibers which follows cooperative scheduling and is not associated to the Operating System which leads to reduced overhead.

Whereas, Threads have higher overheads due to the reason that Threads are created and managed by Operating System scheduler.

3.2.2 Method of Concurrency

There is no doubt that Threads offer optimum Parallelism, which makes the Threads relevant for complex computational operations which are required to be executed concurrently. But with the benefit of Parallelism, there comes a pitfall of having deadlock situations where different threads try to access the same resource, this abrupt execution.

But Fibers on the other side offers concurrency through two different types of scheduling which are, Cooperative as well as Preemptive. This offers more control over the flow of execution and lowering down the chances of Resource Contention.

3.2.3 Responsiveness

Fibers gives a better level of responsiveness than Threads as the current instance execution of microservice never blocks other services execution as the run time scheduler effectively switch between Fibers in comparison to the switching of threads.

Also, the application run time scheduler does not make any OS specific kernel calls for scheduling unlike Threads, which also promotes higher responsiveness of Fibers.

4 Design Specification & Implementation

Any application's design and implementation are of utmost importance in producing a successful and dependable software product. The application which we are going to utilize for this experiment is a modern web platform that is designed and developed using a modern and most commonly used stack of technology. The primary objective is to create a user-friendly system which enables and ensures seamless data exchange and real-time interactions.

As a part of implementation, we shall be having performance comparison among the backend API being developed in Java using the Framework called Spring Boot and another backend API with the same functionality which shall be developed in Golang using Gin framework. For this comparison, we will have the **factor of Latency** which will be considered as an **evaluation metrics** to evaluate the performance of both the backend API.

4.1 Thread Model Based Implementation

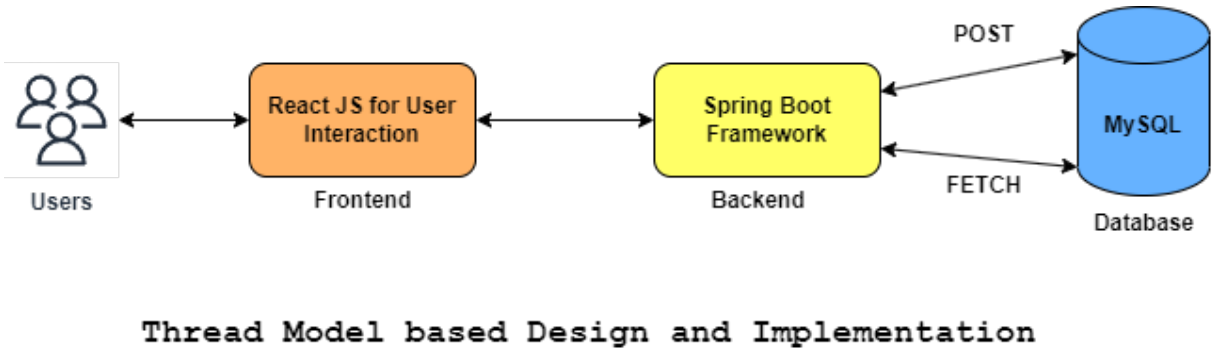

Figure 5: **Thread Management based Implementation**

In a way to achieve the implementation of the **Thread Management based Model** , we shall be using the **technical stack** which involves an **application in front-end for user interface developed in ReactJS**. Talking of the **back-end** part, we have **Spring Boot Framework** exclusively developed in **Java** where the business logic for the endpoints, that is, POST as well as the FETCH operation will be developed. After the proper functioning and relevant unit testing of end points, they will be **exposed to the front-end application for utilization** or may be for a group of end users to use them in accordance to their requirement usage.

4.2 Goroutine Model Based Implementation

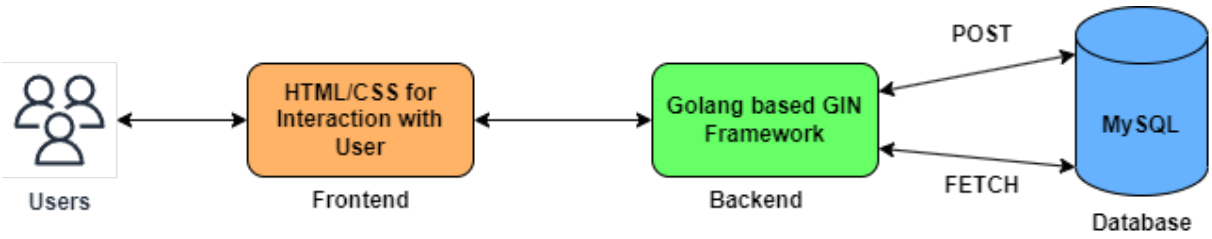

Figure 6: **Fiber Management based Implementation**

In a way to achieve the implementation of the **Goroutine Management based Model** , we shall be using the **technical stack** which involves an **application in front-end for user interface developed in HTML/CSS**. Talking of the **back-end** part, we have **Gin Framework** exclusively developed in **Golang** where the business logic for the endpoints, that is, POST as well as the FETCH operation will be developed. After the proper functioning and relevant unit testing of end points, they will be **exposed to the front-end application for utilization** or may be for a group of end users to use them in accordance to their requirement usage.

5 Evaluation

Evaluation is very essential part of the thesis because it assures the accuracy, dependability, and trustworthiness of study results. So in this section, on the basis of above implementation of our models in their respective technical stack as discussed in the previous section, we will evaluate the results of both the models by performing certain experiments on both the model based implementations on the basis of the performance metrics, that is, the **factor of Latency**. To maintain the state of unbiasedness for both the model-based evaluations, we shall have equal input payloads provided.

5.1 Thread Model Evaluation

Hoping on to this part of evaluation, here we are going to perform some experiments in which we will interact with the user application and will invoke requests to upload

and persist several image objects of different sizes in the database. In this case, we have Spring Boot framework based on Java which will process each incoming request from the application. Please refer the below tabular readings observed while running test cases :

Image size (KB)	Response Time (ms)	Image Type
37	497	PNG
46	821	PNG
124	1287	PNG
24	486	PNG
74	884	JPG
61	817	JPG
93	1158	JPG
17	450	JPG
21	467	PNG
128	1392	PNG

Figure 7: **Thread Implementation Test Readings**

We shall be evaluating the performance metrics, that is Latency, on the basis of two evaluation metrics, that are, size of the image object as well as the total response time taken to perform that operation.

5.2 Fiber Model Evaluation

And in this part of evaluation, we are going to perform same experiments with exact same payload as input in which we will interact with the user application and will invoke requests to upload and persist same image objects as of previous experiment of different sizes in the database. In this case, we now have Golang based GIN framework which will process each incoming request from the application, persist the image in the database and responds back to the application.

Image size (KB)	Response Time (ms)	Image Type
37	89	PNG
46	97	PNG
124	325	PNG
24	78	PNG
74	114	JPG
61	143	JPG
93	152	JPG
17	48	JPG
21	64	PNG
128	345	PNG

Figure 8: **Goroutine Implementation Test Readings**

Gin is a well-liked web framework for creating APIs and web applications in the Go (Golang) computer language. It offers a compact and quick HTTP routing framework together with a selection of practical middle-ware for typical jobs. Gin is ideal for creating Restful APIs since it is built for great performance **Sulaksono (2023)**. Please refer the below tabular readings observed while performing tests :

Here as well, we shall be evaluating Latency as the performance metrics, on the basis of two evaluation metrics, that are, size of the image object as well as the total response time taken to perform that operation.

5.3 Discussion

As we saw that we performed testing of the application in the previous subsections of both, the Thread model evaluation as well as the Fiber model evaluation. We gave images as an input and performed testing to process and persist the image and then calculated the response time for both the model for a range of image objects with different formats (**JPG** and **PNG**) in our case.

5.3.1 Thread Model Readings

As we can see above, the test readings in the tabular format for both the models. Lets plot the readings in the graphical representation for better understanding of the behavior of first model, that is, Thread-based Model.

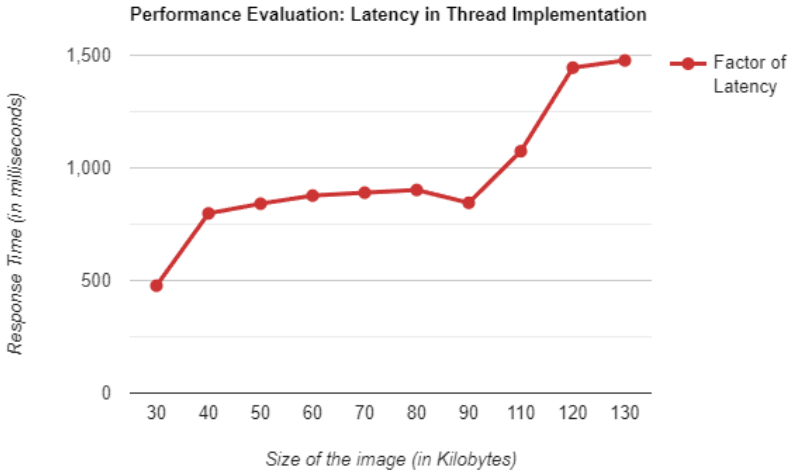

Figure 9: **Graphical Representation of Thread model as per the Test Readings**

Let us now analyze the readings, starting from the first test reading where the image has been given as an input of the size nearly about **30 KB**, and the minimum amount response time taken is nearly about **500 milliseconds**. Then when we increased the size of the image, in the range of **40-90 KB**, we observed that the latency in this size range of the image is more **consistent**, though there is a spike in the response time of about **300 milliseconds** when the size of the image goes **beyond 30 KB**.

Then as we again move beyond the **90 KB** size of the image as an input, we can again observe a linear spike in the response time of the microservice. For instance, if an input image with the size of **90 KB** produces a response time of nearly 850 milliseconds, then going beyond this size up till **110 KB**, the response time reaches to about **1100**

milliseconds, that is, a spike of **250 milliseconds**. Hereon, going beyond **110 KB** up till **120 KB**, the response time graph spikes to about **1400 KB**.

5.3.2 Fiber Model Readings

Lets now plot the readings in the graphical representation for better understanding of the behavior of second model, that is, Fiber-based Model.

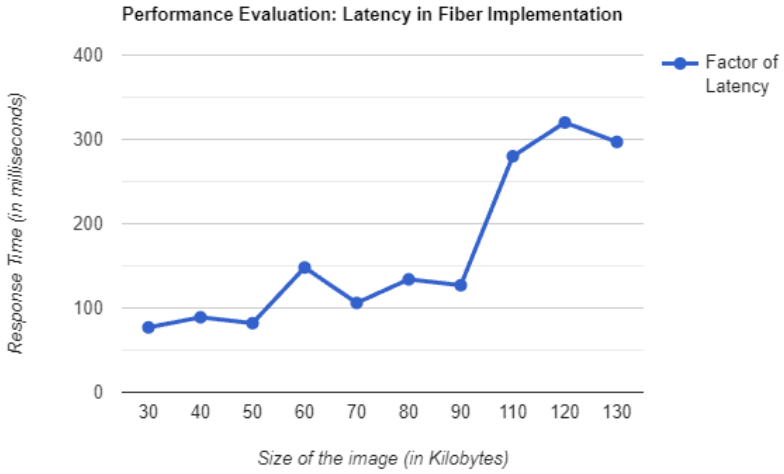

Figure 10: **Graphical Representation of Fiber model as per the Test Readings**

Let us now analyze the readings, starting from the first test reading where the image has been given as an input of the size nearly about **30 KB**, and the minimum amount response time taken is nearly about **80 milliseconds**. Then when we increased the size of the image, up till **50 KB**, we observed that the response time in this size range of the image is more consistent lying below **100 milliseconds**.

Now we move beyond **50 KB** as an input image object to the request payload, we see a the graph of response time spikes to **150 milliseconds**, though then the graph again plummets down to **100 milliseconds** as we again increase the image size beyond **65 KB**. Hereon, more or less, we find a level of consistency in the response time scale lying between **105 milliseconds to 150 milliseconds** for the input image size range between **65-90 KB**. Then we get to see a massive spike in the scale of response time that reaches to **280 milliseconds** as we increase the image size beyond **90 KB up till 110 KB**. And now on increasing the image size linearly to **120 KB**, we see again a minute spike in the response time scale that reaches beyond **300 milliseconds**.

5.3.3 Critical Performance Comparison

On the basis of the above test readings in the previous part, we observed that the response time of the image is directly proportional to the size of the image. As when we increased the image size in the input, the value of the response time increased subsequently. Let

us now critically analyze both the latencies and on the basis of them we shall prove that which implementation model is more effective and reliable.

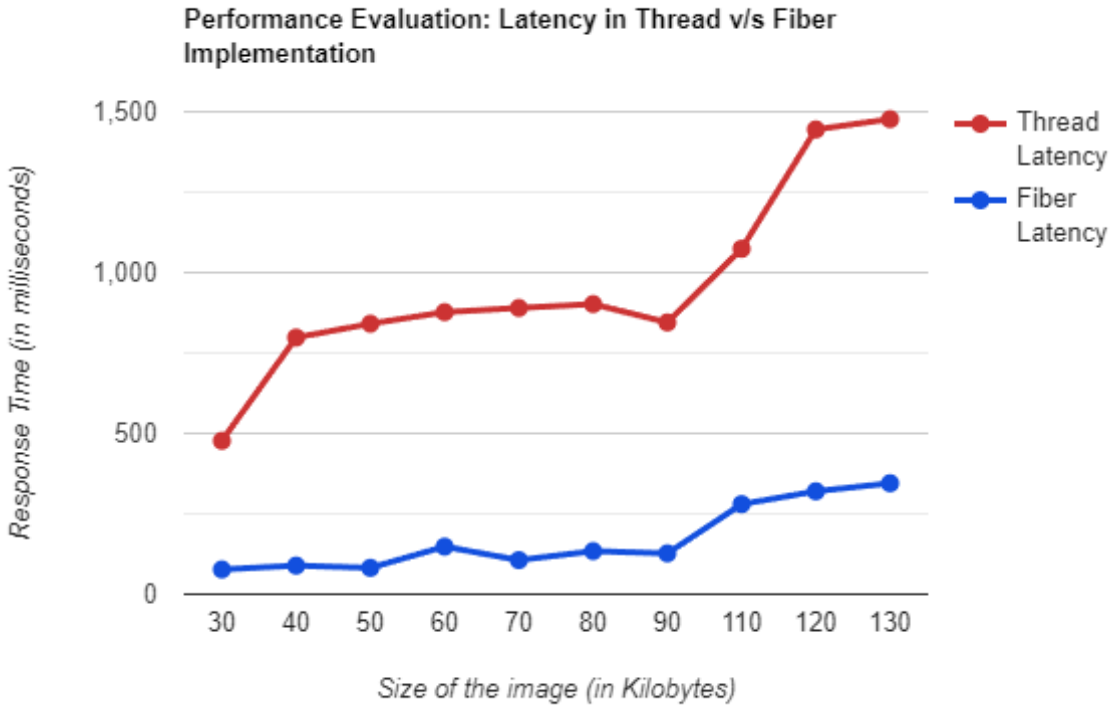

Figure 11: **Critical Comparison of Thread and Fiber model**

If we can compare the readings of both the models as shown in the **figure 10**, where the red line represents the latency in the response time of microservice having Thread based mechanism and the blue line represents the latency caused in Fiber(Goroutine) mechanism for the same input in the microservice response time. We will critically examine both models for few input readings.

1. Firstly, the response time of the microservice when given **30 KB** of image object as an input, the Thread model shows the latency of nearly **500 milliseconds** to process and send the response back, whereas the Fiber based model for the same input consumes only **80 milliseconds**.

2. Secondly, if we see the consistency range of both the models, we see that the Thread model shows stability for the image size ranging from **40-90 KB** with the response time ranging from **800-850 milliseconds**, which is when compared to the Fiber based model which shows stability from **30-90 KB** of image size with the respective response time ranging from **80-130 milliseconds**.

3. And then, if we talk about the spike after the **critical point** of the image size, that is **90 KB**, we see that the Thread model's latency factor **surges** with **600 milliseconds** from the range **90-120 KB** of image. Whereas in the case of Fiber implementation model, the factor of Latency **surges** with just **190 milliseconds** for the input image

objects ranging from **90-120 KB**.

A major criticality to prove the performance and effectiveness of the Fiber based model in comparison with the Thread based model is:

The response time taken by the microservice when processing the **maximum size image(128 KB in our case)** given as an input for processing which comes out to be around **345 milliseconds** in the case of **Fiber based Model** and the response time taken by the microservice when processing the **minimum size image(17 KB in our case)** in the **Thread based Model**, which comes out to be around **400 milliseconds**, creates a solemn **performance gap of 55 milliseconds** and proves Fiber based model to be more time-efficient and ensuring higher efficacy for microservices architecture.

5.3.4 Result

As per the above practical readings from the experimentations as well as the critical comparison of the two models we observed that for any image object of any size as a given input:

1. Minima : if the microservice is implemented with Fiber based Model and takes **(X)** amount of response time to process, then the Thread based model oriented microservice takes almost **4(X)** amount of time to process the same input image object.

2. Maxima : if the microservice is implemented with Fiber based Model and takes **(X)** amount of response time to process, then the Thread based model oriented microservice takes almost **8.5(X)** amount of time to process the same input image object.

Hence, it is proved that Fiber based model for microservice implementation in the processing of heavy image object as request payload is **6.25 times faster** on a average than that of Thread based model for microservice implementation.

6 Conclusion and Future Work

In this paper, we have discussed the need of this research by stating the question that can we reduce latency in response time of microservices by eliminating the Thread scheduling of processes/tasks and propose a thread-less way for task scheduling to improve the latency in response time to a massive extent. And then as a part of methodology, we have discussed and understood both the models' respective internal working by defining the respective procedures. Then as a part of Implementation section, we have seen the complete implementation architecture involved for both the models following layered architecture practically by making an application to interact with the backend service implemented in both the model based approaches, which are based on Spring Boot as well as Gin based frameworks.

In comparison to the Thread-based Model, the Fiber-based Model, as illustrated by Golang's Goroutines, exhibits higher efficiency and high performance. We have observed that due to many limitations, microservices oriented using the Thread based model consequently consume a greater amount of response time for processing the request, which is four times the amount time a Fiber oriented microservice takes to process the same image with Fiber based model. In conclusion, we prove that on an average, the Fiber

based model for microservice interaction is **6.25 times faster** than that of Thread based Model specifically in the processing of heavy image object as request payload.

In future, on the basis of this experiment, we can have a thought of deploying this fiber based model application entirely over cloud and can be tested for scalability evaluation with varying loads, in terms of massive concurrent requests as well as increasing the number of microservices. Additionally, this can be performed with different types of microservices like Cache-intensive, Computational-intensive, as well the microservices that depend on other microservices for response. And this definitely will help in understanding how well this solution performs as the system grows by adapting continuous development.

References

Abdel Khaleq, A. and Ra, I. (2019). **Agnostic Approach for Microservices Autoscaling in Cloud Applications**, *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, pp. 1411–1415.

Agavane, P. B. (2020). ***Improve Load Balancing Performance and Efficiency Using Equally Spread Current Execution Algorithm working with response time clustering in Microservices***, PhD thesis, Dublin, National College of Ireland.

Al-Debagy, O. and Martinek, P. (2018). **A Comparative Review of Microservices and Monolithic Architectures**, *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pp. 000149–000154.

Benni, B., Mosser, S., Caissy, J.-P. and Guéhéneuc, Y.-G. (2020). **Can Microservice-Based Online-Retailers Be Used as an SPL? A Study of Six Reference Architectures**, *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*, SPLC '20, Association for Computing Machinery, New York, NY, USA.
URL: *https://doi.org/10.1145/3382025.3414979*

Cai, B., Wang, B., Yang, M. and Guo, Q. (2023). **AutoMan: Resource-efficient provisioning with tail latency guarantees for microservices**, *Future Generation Computer Systems* **143**: 61–75.
URL: *https://www.sciencedirect.com/science/article/pii/S0167739X23000213*

Einav, Y. (2019). ***Amazon real time Instance***, PhD thesis.
URL: *https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales*

Eyerman, S. and Hur, I. (2022). **Efficient Asynchronous RPC Calls for Microservices: DeathStarBench Study**.

Jia, Z. and Witchel, E. (2021). Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices, *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, Association for Computing Machinery, New York, NY, USA,

p. 152–166.
URL: *https://doi.org/10.1145/3445814.3446701*

Kumar, P. K., Agarwal, R., Shivaprasad, R., Sitaram, D. and Kalambur, S. (2021). **Performance Characterization of Communication Protocols in Microservice Applications**, *2021 International Conference on Smart Applications, Communications and Networking (SmartNets)*, pp. 1–5.

Li, F., Fröhlich, J., Schall, D., Lachenmayr, M., Stückjürgen, C., Meixner, S. and Buschmann, F. (2018). **Microservice patterns for the life cycle of industrial edge software**, *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, pp. 1–11.

Mirhosseini, A., L. West, B., W. Blake, G. and F. Wenisch, T. (2019). Express-lane scheduling and multithreading to minimize the tail latency of microservices, *2019 IEEE International Conference on Autonomic Computing (ICAC)*, pp. 194–199.

Robert Griesemer, R. P. and Thompson, K. (2009). ***Go Documentation***, PhD thesis.
URL: *https://go.dev/doc/*

Samsad, S. (2021). ***Why Golang performs better?***, PhD thesis.
URL: *https://dev.to/samsadsajid/designing-a-reverse-proxy-why-golang-performs-better-than-java-spring-boot-an-in-depth-analysis-18oe*

Shadija, D., Rezai, M. and Hill, R. (2017). **Microservices: Granularity vs. Performance**, *Companion Proceedings of The10th International Conference on Utility and Cloud Computing*, UCC '17 Companion, Association for Computing Machinery, New York, NY, USA, p. 215–220.
URL: *https://doi.org/10.1145/3147234.3148093*

Shafabakhsh, B., Lagerström, R. and Hacks, S. (2020). **Evaluating the Impact of Inter Process Communication in Microservice Architectures.**, *QuASoQ@ APSEC*, pp. 55–63.

Shóstak, V. (2021). ***Working with middlewares***, PhD thesis.
URL: *https://dev.to/koddr/go-fiber-by-examples-working-with-middlewares-and-boilerplates-3p0m*

Somashekar, G., Suresh, A., Tyagi, S., Dhyani, V., Donkada, K., Pradhan, A. and Gandhi, A. (2022). Reducing the tail latency of microservices applications via optimal configuration tuning, ***2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)***, pp. 111–120.

Sriraman, A. and Wenisch, T. F. (2018). μtune: Auto-tuned threading for {OLDI} microservices, *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 177–194.

Sulaksono, W. B. (2023). ***Golang Gin Gonic Framework***, PhD thesis.
URL: *https://medium.com/@wahyubagus1910/build-scalable-restful-api-with-golang-gin-gonic-framework-43793c730d10*

Suljkanović, A., Milosavljević, B., Inđić, V. and Dejanović, I. (2022). **Developing Microservice-Based Applications Using the Silvera Domain-Specific Language**, *Applied Sciences* **12**(13).
URL: *https://www.mdpi.com/2076-3417/12/13/6679*

Taibi, D., Lenarduzzi, V. and Pahl, C. (2017). **Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation**, *IEEE Cloud Computing* **4**(5): 22–32.

Togashi, N. and Klyuev, V. (2014). Concurrency in go and java: Performance analysis, *2014 4th IEEE International Conference on Information Science and Technology*, pp. 213–216.

Vogel, K. (2022). ***A Deep Dive Into Go Concurrency***, PhD thesis.
URL: *https://betterprogramming.pub/deep-dive-into-concurrency-of-go-93002344d37b*

Xu, Y. and Shang, Y. (2019). **Dynamic Priority based Weighted Scheduling Algorithm in Microservice System**, *IOP Conference Series: Materials Science and Engineering* **490**(4): 042048.
URL: *https://dx.doi.org/10.1088/1757-899X/490/4/042048*

Zhang, Y., Xu, K., Wang, H., Li, Q., Li, T. and Cao, X. (2018). **Going Fast and Fair: Latency Optimization for Cloud-Based Service Chains**, *IEEE Network* **32**(2): 138–143.