Computer Systems Science & Engineering
DOI: 10.32604/csse.2023.032509
Article

Tech Science Press

Latency Minimization Using an Adaptive Load Balancing Technique in Microservices Applications

G. Selvakumar[1,*], L. S. Jayashree[2] and S. Arumugam[3]

[1]Department of Computer Science and Engineering, KPR Institute of Engineering and Technology, Coimbatore, 641407, India
[2]Department of Computer Science and Engineering, PSG College of Technology, Coimbatore, 641004, India
[3]Department of Computer Science and Engineering, Nandha College of Engineering, Erode, 638052, India
*Corresponding Author: G. Selvakumar. Email: selvakumarguru@gmail.com
Received: 20 May 2022; Accepted: 04 July 2022

Abstract: Advancements in cloud computing and virtualization technologies have revolutionized Enterprise Application Development with innovative ways to design and develop complex systems. Microservices Architecture is one of the recent techniques in which Enterprise Systems can be developed as fine-grained smaller components and deployed independently. This methodology brings numerous benefits like scalability, resilience, flexibility in development, faster time to market, etc. and the advantages; Microservices bring some challenges too. Multiple microservices need to be invoked one by one as a chain. In most applications, more than one chain of microservices runs in parallel to complete a particular requirement To complete a user's request. It results in competition for resources and the need for more inter-service communication among the services, which increases the overall latency of the application. A new approach has been proposed in this paper to handle a complex chain of microservices and reduce the latency of user requests. A machine learning technique is followed to predict the weighting time of different types of requests. The communication time among services distributed among different physical machines are estimated based on that and obtained insights are applied to an algorithm to calculate their priorities dynamically and select suitable service instances to minimize the latency based on the shortest queue waiting time. Experiments were done for both interactive as well as non interactive workloads to test the effectiveness of the solution. The approach has been proved to be very effective in reducing latency in the case of long service chains.

Keywords: Microservices; load balancing; cloud computing; latency optimization; netflix

1 Introduction

In recent years, virtualization techniques have gained much attention due to their advantages in optimum resource utilization, reduced capital and operating expenditure, increased agility and responsiveness, and, most importantly, minimized downtime [1]. In cloud computing, virtualization revolutionized how

Enterprise applications are developed. With cloud-based services, the remote servers take care of the infrastructure and platform requirements of the applications, which make the applications' faster development, flexibility, and easier maintainability [2]. Microservices Architecture is one of the latest innovations happening in this segment which has influenced several companies like Amazon, Netflix, Twitter, eBay, Uber, and many more.

Microservices have evolved from the concepts of service-oriented architecture (SOA) by inheriting all its fundamental principles of it. SOA allows designing an application as a collection of loosely coupled software services that can communicate with each other to perform the business functionalities [3]. According to Martin Fowler and James Lewis, microservices are an approach to developing an application as a composition of multiple more minor services. Also, the microservices can be upgraded or replaced independently to support significant levels of scalability. It makes microservices a preferred architectural pattern for many enterprise application development activities [4]. Horizontal scaling is one of the primary reasons for migration from monolithic systems, which is essential to maintain the user experience irrespective of the higher traffic. Conventional enterprise applications are developed as monolithic systems and grow in size over time due to multiple updates and enhancements. The strong coupling among the components of monolithic systems makes them very difficult to maintain and scale. It also results in cascading failures [5]. On the other hand, Microservices architecture addresses these problems. It suggests creating a system evolved from a group of independent services that are scalable and resilient to a single point of failure shown in Fig. 1.

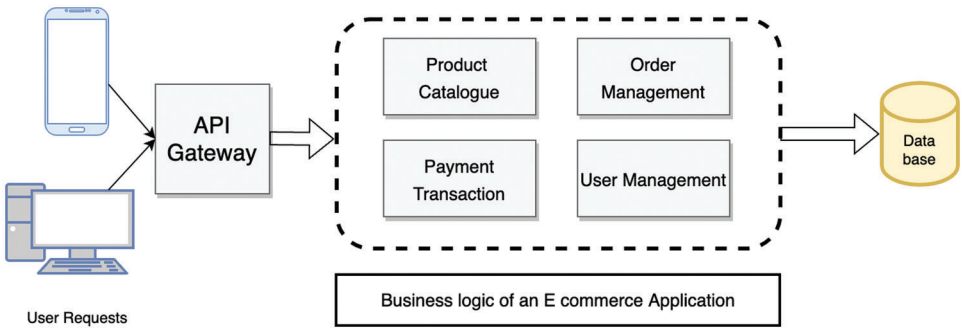

Figure 1: Monolithic architecture

Distributing the requests across the group of backend servers in which multiple instances of the services reside is called load balancing. The primary objective of load balancing is to optimize the use of resources, maximize throughput, and minimize response time [6]. In Microservice Architecture, applications are constructed as a collection of separate and independent services that need to communicate with each other to complete the operation shown in Fig. 2. These service chains ensure that the system scales better with growing demand and low operating costs. However, these service chains and communication among them results in higher latency. In cloud-native microservices applications, there are three main optimization objectives. To reduce the cost of cloud services by optimizing the usage, reducing the overall latency, and handling the application when one of the service providers becomes unavailable [7]. In our paper, we try to focus on the latency problem.

The main contributions of our work can be summarised as follows. The literature on load balancing in a cloud environment and cloud-native application development activities has been reviewed. The one area where load balancing has been still challenging is managing the service chains in complicated applications with multiple tasks spanning physical machines and various service instances [8]. Then, a load balancing index by calculating the various dependencies concerning the inter-service communication

and communication among the physical machines has been calculated. Based on the inferences, a default load-balancing algorithm is modified with the proposed approach and implemented. The implementation of Netflix Eula was used for this purpose [9]. The solution was deployed in the Google Cloud Platform, and experiments were conducted with different client requests to test the latency. Finally, we proved that the proposed solution could significantly reduce the latency in microservices chains.

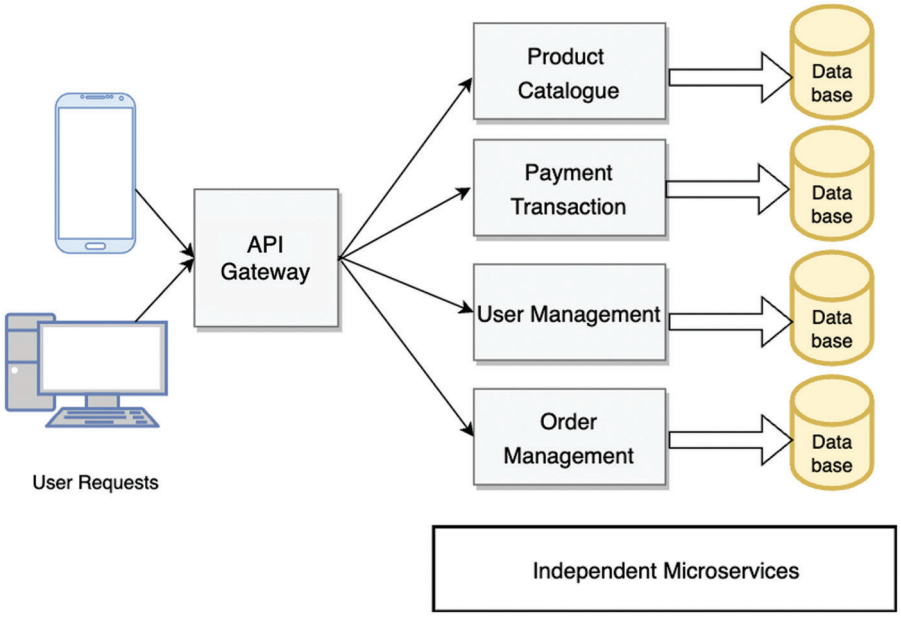

Figure 2: Microservices architecture

The rest of the paper has been organized in the following manner. Section 2 discusses the state-of-the-art research work that has happened so far in microservices load balancing and the identified limitations in the existing approaches. Section 3 discusses Microservices service chains and the latency problems in detail. Section 4 illustrates the system under consideration and analyses contemporary cloud-native application architecture characteristics to describe the research challenge. Section 5 explains how the proposed solution has been derived mathematically and how the load balance index can be calculated, which plays a vital role in the load balancing process. In section 6, we discuss the experimental setup we used and the implementation details we used with the support of Netflix open source components. In section 7, the results are documented along with the discussion, and in section 8, we conclude the paper.

2 Related Research

Several noteworthy contributions have been made to reduce the latency in Microservices applications, primarily when a chain of Microservices handles the user request. Xu, Yan & Shang propose a dynamic priority-based weighted scheduling algorithm, Yanlei, to address the issue of latency when multiple microservices are required to collaborate with each. The latency in these systems becomes more than in Monolithic systems and collapses the purpose of using Microservices Architecture [10].

A load balancing technique supported by priority queues has been proposed to minimize the latency of microservices long chains, considering the competition for resources among the different chains. They used message queues with different priorities to differentiate the microservices chains, and the resource allocation is done dynamically by adjusting their priority dynamically [11]. Multiple priority queues were proposed for

microservices at the edge cloud server. They implement a multiple-level feedback system to set the priorities. The transition from higher to lower queues happens systematically based on the packet size. It results in short microservices being prioritized over long microservices. In general, it follows the shortest job first technique for load balancing.

A latency estimation approach to manage the interactive workloads is proposed to handle the uncertainty in latency due to multiple queues. They also suggest a feedback scheme to validate workload distribution's fairness and ensure that non-interactive workloads are not affected. A Task Chain-Based Load Balancing Algorithm is proposed that focuses on the chain of service calls and the data transmission between the servers. It uses the three algorithms, Particle Swarm Optimization, Simulated Annealing and Genetic algorithm, to develop the load balancing methodology. They try to address the problems of unbalanced load and lengthier turnaround time in microservices container applications with the help of a load model which combines the service discovery and performance monitoring of the server ecosystem [12]. They made use of the Optimized Ant Colony algorithm for this purpose. Power consumption has been analysed along with latency improvement. A black bo monitoring infrastructure is created to measure performance and power consumption. They designed and developed a graph-based analysis that maps each microservice in the cluster to a service time requirement. Also, a feedback and control loop named Observe Decide Act (ODA) was created along with an interface to ensure maximum power usage. Their methodology plays a vital role in reducing the energy consumption of the workloads significantly. Similarly, proposes shark smell optimization and a fuzzy logic approach to balance the load for the sole purpose of decreasing the energy consumption. The load balancing for energy preservation plays a vital role IoT based environments.

A simple linear progression approach was proposed to predict the time-weight of the request queues and was used in the shortest queue-waiting-time load balancing algorithm. This approach appears effective. However, it has an overhead in running a machine learning algorithm in parallel with the load balancing algorithm. Optimal latency may not always be guaranteed in this approach since it depends on the efficiency of the learning algorithm. As per the analysis, load balancing has two essential aspects. Firstly, the types and numbers of service instances should be selected, and secondly, the physical or virtual machines should be selected. Combined, they propose an approach for scheduling microservices in multi-cloud scenarios.

Since many approaches have proven that the complex requests are optimized significantly at the cost of increasing the latency of standard requests, they used a latency estimation approach to identify the final latency and focus on interactive workloads. According to achieve a high-performance workload should be migrated among the servers. These load balancing activities are implemented as a part of container solutions. The commercially available container orchestration platforms like Kubernetes, Docker Swarm and others provide various strategies for these objectives. Their effectiveness is yet to be demonstrated clearly in enterprise applications. The container based cloud-native applications experience the application level isolation rather than server level, so that if anything goes wrong in a container it will not impact the entire virtual machine or the server. However, handling excessive consumption of a set of specific resources due to increase in load is still a problem not perfectly solved [13].

Moreover, when the number of microservices instances is large, the load balancing solution could also become a bottleneck. While load balancers are traditionally available as hardware for balancing network traffic, service-oriented models are analysed to provide load balancing. The developed load balancers can be rented to the customers, similar to software as a service.

3 Microservices Service Chains

3.1 Service Chains in Applications

When a request from a user arrives at the micros services system, it is processed by more than one service implementation. They form a chain in which each service has a specific role, and the request processing is completed only when all the services are executed. This need for communication among multiple services is the primary reason for higher latency in microservices systems.

As shown in Fig. 3, the user request reaches Service B through A and A and C. They may ask for the. Service B simultaneously. The chain of services ACB is more extended than AB, and obviously, it would experience higher latency. In real-world applications, the length of the chains varies according to the implementation complexity of the functionalities. Many times, more than one chain happens to compete for resources during the execution. The load balancing methods used in recent times fail to consider the increase in latency due to the resource competition among the multiple chains in a single microservices application.

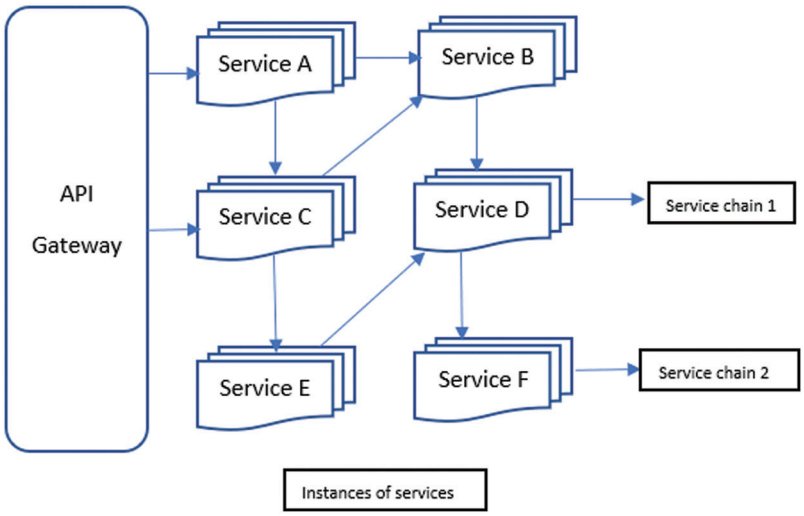

Figure 3: Microservices chains

The service chains are the primary cause of increased latency in the application since different queues on the cloud servers handle each service. Multiple queues should be handled to complete a given user request. It significantly challenges optimising the queueing delay for a microservices request. While most of the current research solutions focus on load balancing and latency optimization in various perspectives concerning cloud-native applications, no complete solution is available to optimize latency in complicated service chains that ensure reasonable delay for interactive and non-interactive workloads. While most of the solutions are working fine for the interactive workloads, it is achieved only at the cost of increasing the delay for leaf requests. Our approach ensures that all kinds of workloads are given a fair allocation while optimizing the latency.

3.2 Latency in Service Chains

As per the literature survey, most load balancing solutions focus wholly on individual tasks. However, they discount the dependencies among the tasks. In most enterprise applications, complex and hybrid service chains are created during the run time by several microservices. Security functionalities like deep packet inspection, firewall and database operations may take longer to complete and contribute to increased latency most of the time.

Understanding latency is significant while deriving a solution for latency optimization in cloud-native applications. The time elapsed from the time user requested the time user received the response can be due to three components. The delay was due to network communication, time taken to complete the service, and the time spent waiting in the queues. The latency due to network communication is caused by several factors that the application owners cannot handle. The time that is taken by the physical server to process the user request is known as the service time. Various research works focus on techniques like enhancing cache efficiency and optimizing the instructions or query statements. Our objective is primarily to focus on the third component, which is the time taken by the user request to wait in a queue for execution. This component is the major contributor to increased latency. However, few studies have been done in this segment since the vast adoption of microservices in enterprises has happened only recently. The complexity of this problem is also higher due to the run-time competition among microservices chains for the resources.

4 The System Analysis

In this section, we discuss the system under consideration for load balancing and the basic ideas behind the techniques proposed for load balancing.

The system we consider for our research work has containers as its core component in which the microservices application runs independently. A container is a place where the service is executed with all its dependencies like runtime, libraries and the database. Tools like Kubernetes are used to schedule and manage the applications defined by the containers. It enables running hundreds or thousands of containers as a part of a complete application which is common in an enterprise scenario. A logical wrapper for a container is usually applied, known as a pod, and pods are the minor deployable units. They may contain one or more containers that could share resources. The containers within the pod share the specifications of their execution-style. We make use of clusters to run all the pods in a more organized manner. Node is a physical or a virtual machine used to host the pods. The collection of nodes is called clusters, and they are responsible for managing the instances and need to take care of scaling the service instances when the load increases.

The communication between the microservices happens through HTTP protocols, and message queues are shown in Fig. 4. Here the microservices are connected directly with the help of service calls without any middleware. The target instance is located by a specific URL of the microservice.

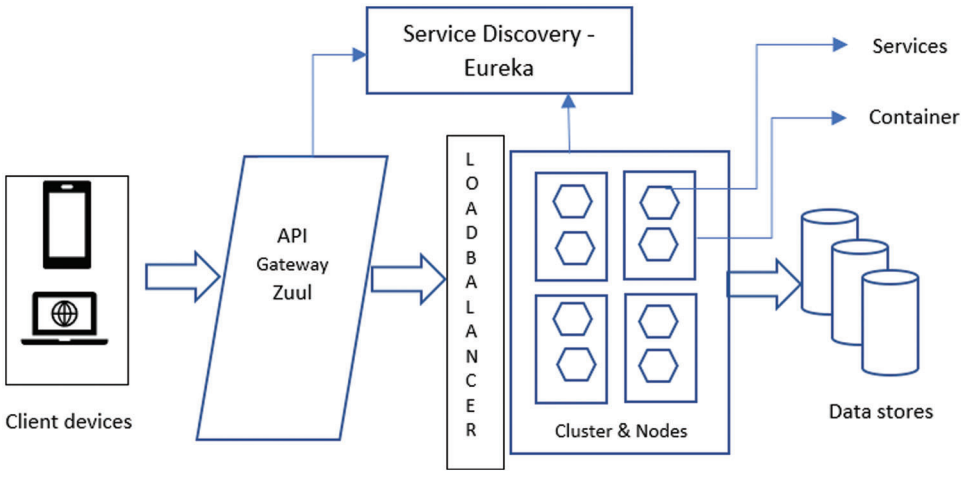

Figure 4: The system architecture

Since the services are small composable units, a unified interface should be provided for the users. Netflix has provided the solution to this problem. Netflix corporation is one of the primary adopters of microservices architecture and has created open-source several microservices-based solutions. Zuul proxy server is one of them that proxies requests from users to the services running in the background. It acts as a unified entrance to the system that can be used by the browser, mobile application or any other user application to consume the services.

Zuul can easily integrate with other Netflix projects such as Hystrix and Eureka. Hystrix is for fault tolerance, and Eureka is used for service discovery and load balancing. Zuul functions as a server-side load balancer.

5 Proposed Approach for Optimal Latency

Scalability is one of the primary objectives of microservices architecture, and one service is deployed multiple times as instances to achieve this. These instances are available on different physical machines. A task is usually executed on different service instances, which may present in different machines. When tasks are executed as a chain of services, we must select the appropriate service instance to reduce latency and increase API response time. Our algorithm selects the appropriate services to be invoked for the given request and tries to minimize the latency by estimating the system resource utilization for each service instance during the execution shown in Fig. 5.

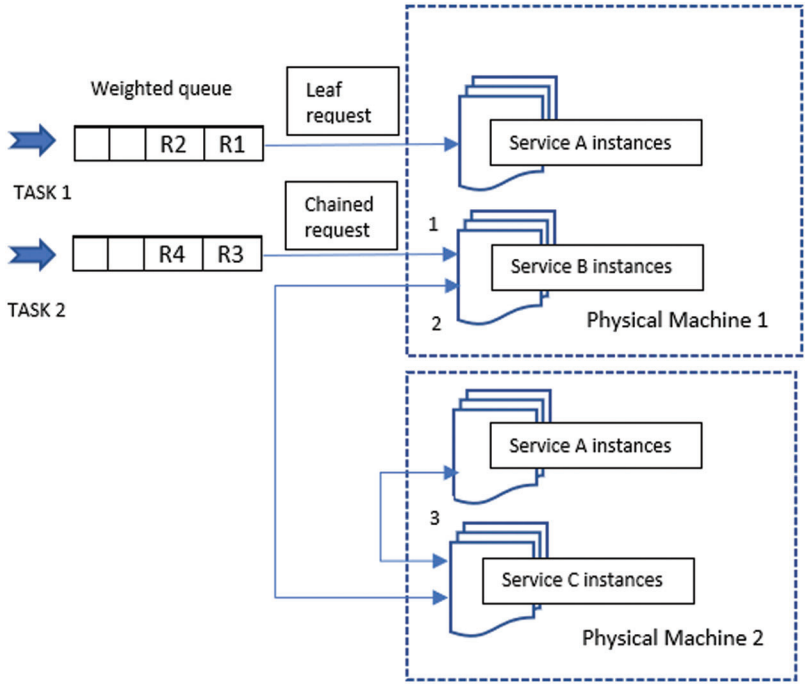

Figure 5: Leaf and chained requests

5.1 Analysis of the Proposed Approach

The proposed approach can be analysed using a sample microservices application. The application is deployed in more than one physical machine, p^1, p^2, p^3…, p^n. The application consists of 'm' service instances in each physical machine s^1, s^2, s^3…, s^m. Each task may contain multiple service invocations that create a chain of an API call containing q tasks $\{t^1, t^2, t^3…., t^q\}$.

Our objective is to choose a service instance s_i in a physical machine p_i for a given task t_i. Our algorithm considers various system resource usages. CPU, memory, and bandwidth utilization are the primary things to be considered in the approach. Services are available to use in the different service instances, and load balancing can be done by ensuring that the use of system resources is roughly the same across the instances. No instance is neither overloaded with its resources or underutilized. To measure this, each service instance is characterised by an indicator called load balance degree, and it is obtained by following the following equations.

$$D(t_i) = \frac{U^R t_i - AU^R t_i}{AU^R t_i} \tag{1}$$

D = Usage indicator of a particular system resource. The utilization of one system resource is taken, and its relative difference to the average of all the other service instances of the same type is calculated. This is considered the usage indicator.

U^R = usage of one system resource on one service instance

AU^R = average usage of all the service instances of same type.

The usage indicator of the service instance plays a vital role in estimating the utilization of CPU, memory and other system resources for a specific request.

$$LBD_1 = \sum_R \left(\frac{\sum_{i=1}^{q} D(t_i)}{q} \right) \tag{2}$$

The above equation is to find out the load balance degree of a particular type of service instance. The usage indicator found in the previous step in each instance is used to calculate the load balance degree of the whole application.

Multiple data transmissions across the physical machines are required to complete an API call and considered for load balancing calculations. The application developers can predetermine a sequence of tasks to be accomplished for one particular API request. Thus, the task chain can be quickly evaluated for its resource requirements. If every service instance has the exact technical specifications and configurations, the processing time would be the same for any given task. Considering the above facts, the service instances in which a specific task needs to be done can be chosen for a sequence of tasks when the system receives an API request. When the service instances are on the same machine, the necessity for more data transmissions across different machines is reduced. It will reduce the delay in responding to the request.

The relationships among the tasks are very significant to analyse since they directly influence the response time. When two tasks are involved, there can be a single data transmission from one task to another, or it may be a call-back request which needs two data transmissions. On the other hand, we have another set of tasks among which there is no dependency.

In the next step, we find the association among the tasks. Three types of associations can exist among the tasks. The association between tasks t_i and tj, A (t_i, tj) = 1, if s_i calls s_j and it is 2 if task t_i needs a call back from t_j. If there is no known association, then A becomes zero.

M (S_i, S_j) = 1 if S_i and S_j are located in the different physical hosts. Otherwise, the value is zero.

$$DT(ti,\ tj) = A(ti,\ tj) * M(S(ti),\ S(tj)) \tag{3}$$

The number of data exchanges required among the machines for executing the tasks ti and tj, can be written as,

$$TP(t_i) = \sum_{j=0}^{q} (DT(t_i,\ t_j) + DT(t_j,\ t_i)) \tag{4}$$

$$\mathrm{N} \quad = \frac{\sum_{i=0}^{q} TP(t_i)}{2} \tag{5}$$

The above is the estimated value for the API request. The calculated load balance degree and the communications across the machines are the primary parameters in the proposed methodology. The above parameters are combined to form an index called LBI to simplify the approach. However, every application is designed and developed differently, necessitating a factor to balance the above equation, which is indicated by the weight 'k'. It is estimated with the help of experiments. Our algorithm calculates this load balance indicator, LBI, for every task chain.

$$LBI = k * LBD_1 + (1 - k) * N \tag{6}$$

The above load balance indicator will be used in our algorithm to determine the priority of each service chain. Here we are trying to use an algorithm inspired by the GPS (Generalized processor sharing) algorithm related to the fair queuing principle. Different tasks require different priority levels in a cloud-native application with service instances distributed across physical machines. For example, transaction-sensitive and financially critical service chains may be given more priority since they may get failed if latency exceeds a specific limit. When multiple service chains are queued up on one end of the API gateway, the load balancing mechanism should decide how it should serve the chains. In our approach, the service chains are provided different weights, and the algorithm ensures that scheduling is maintained to minimize the latency of critical service chains and also the less critical service chains are taken care of with adequate fairness.

As per the GPS algorithm, considering there are N different service chains configured with different weights 'ω', and the capacity of the cluster is R, then the rate of service that can be guaranteed for a particular service chain 'i' can be proved to be,

$$\mathrm{R}i = \frac{\omega \mathrm{i}}{\sum_{j=0}^{N} \omega \mathrm{j}}\ R \tag{7}$$

The generic GPS algorithm can be extended with fair queuing and weighted fair queuing approaches to specify which service chain needs to be given how small amount of priority. Weighted fair queuing is the natural extension of fair queuing, and our algorithm uses this to allocate resources for our service chains.

5.2 Algorithm to Find out the Optimum Values for the K Factor

Algorithm to predict the k value of request

Input:

Sample requests received in the past, RP

'k' values considered,

Relative error values

Output:

Newer values for 'k'

From sklearn.linear_model import Linear Regression as linReg

(Continued)

Algorithm (continued)

Step 1: Initialize the values

K = [0]

RType = []

QLength = []

Step 2: Calculate relative error for all sample requests for i in range (0, len(RP))

- Populate the RType and QLength values for all given requests
- linReg.predict(RP)
- calculate the relative error for a given request type and queue length for the predicted values of waiting time and actual values of waiting time

end for

Step 3: Calculate the new optimum values for all sample requests for i in range (0, len(RP))

- assign new values of k for the request type so that the relative error identified in the previous step is close to zero.

end for

While the above section focuses on determining the load balance indicator for the requests, the queue waiting time of each request depends mainly on the nature of the application, and it is represented as 'k' in the calculations. A linear regression model can be created based on a set of requests. To find out the optimum value for the 'k'. A supervised machine learning technique is applied here. This approach will impact the precision of our results immensely since the environment has become very dynamic due to the factors like different nature of requests, heterogeneous clusters and ever-changing database sizes. Hence, we develop a data science problem and use machine learning to solve this. The linear correlation between queue waiting for time and type of request is simpler to implement, and it is ensured that it would not create any performance bottleneck for the proposed approach. The computing complexity of this process has been maintained less so that it can be exercised in any environment with burst internet traffic.

The implementation of the above algorithm is based on the scikit-learn library. This algorithm continuously collects the new samples from different types of applications. The predicted values of weighting time and the actual weighting times are compared to find the relative errors. It helps identify the optimal value that can be assigned for a given request type.

5.3 Algorithm to Select the Appropriate Service Instance Chain for Optimal Latency

Algorithm to choose the service instance chain

Step 1: for Every new Request from the API Gateway

Please place it in the queue and fetch the request specifications created already.

Step 2: If the request is a leaf request, assign the priority to 1 and set the maximum delay 'W_{max} = x ms' to ensure fair allocation.

Step 3: Else, if the request has to invoke a chain of services, do the following,

Step 3.1: calculate the LBI for the entire service chain.

(Continued)

Algorithm (continued)

Step 3.2: the LBI for each service cain is stored for further calculations and dynamically changing priorities.

Step 3.3: store the set of candidate solutions based on the GPS approach for further processing.

Step 3.4: based on the resource availability in each physical machine that has been chosen in the service chain, calculate the L_T, the threshold value of latency.

Step 3.5: For all the service chains exceeding LT, update the priority by 1, recalculate the latency, and update the candidate solutions.

Step 3.6: If a leaf request in a queue exceeds the delay W_{max}, allocate the resources; keep them in the queue.

Step 4: The candidate solution with the least latency is chosen, and the service chain sequence is finalised accordingly.

Step 5: Repeat steps 2, 3, and 4 until all the API requests are completed servicing.

6 Experimental Set-Up and Implementation Details

The proposed approach has been experimented with in the following environment with an Intel Core-i7 processor system supported by 16GB of RAM. Eclipse IDE is used as a development environment with Java 1.8. To develop test applications, we used Spring boot as the framework and REST API to expose the microservices. Maven tool is used for building the application. For deployment, we used the Google cloud platform and Kubernetes clusters.

6.1 Netflix Zuul and Eureka

Netflix is one of the most famous movies streaming platforms, serving over 200 million users as of 2021. The service availability and global scalability are the two essential requirements of any subscription-based video streaming platform of this scale. Their services are considered to be using nearly 15% of the world's internet bandwidth. Netflix is an excellent example of building a cloud-native microservice application that can handle substantial scalability challenges. Since Netflix is committed to open-source technologies, they have provided many of its components that can be customized and reused by any developer worldwide. In our research, we extensively use the Netflix components for our experiments.

Netflix Zuul is the gateway for all the requests coming from Netflix clients. It is a JVM-based router that serves as a server-side load balancer. It provides a single entry to the system for every client browser, mobile app or similar and takes care of dynamic routing and monitoring of requests to the backend services of Netflix. Zuul makes use of another Netflix component called Ribbon as a load balancer. The round-robin algorithm is defaulted here to select the services on the backend.

Eureka Server is another component by Netflix which consists of all the information about every available client-service application and functions as a telephone directory for the microservices. Eureka Server, also called Discovery Server, registers all Microservice instances with it to serve the client applications as and when requested. Netflix Zuul is connected to the Eureka server to find suitable microservice instances for service calls.

6.2 Implementation Details

A sample microservices application for e-commerce has been developed to test the proposed approach using the spring boot framework. In this sample application, the standard service requirements of an

e-commerce application such as user management, inventory management, cart management, transaction, verification of order, cancellation of the order, shipping and other functionalities are implemented in the form of independent services.

The implementation of our proposed approach also considers DevOps which is defined as a set of methods to organize software development in order to integrate deployment and operations. Microservices and DevOps are associated with each other and they depends upon cloud and virtualization techniques. Collectively, we try to address the challenges of cloud-native applications such as scaling, continuous develivery, continuous integration, automation and effective resource utilization [14].

The services have been developed using spring boot, and the communication among them is enabled by REST API. Netflix provides its JVM-based router Zuul as an open-source server-side load balancer which is used in our experiments. Zuul users a variety of filters that can be easily configured.

The package com provides the load balancer-related interfaces and classes. Netflix. Load balancer, which is a part of Netflix open source. Load Balancer is the interface which declares the standard methods required to be implemented for building our load balancing mechanism. Base Load Balancer is the class with a set of methods that can be overridden as given here. setRule() method plays the role of defining our custom strategy for load balancing.

```
public void setRule (IRule rule) {
If (rule ! = null) {
This.rule = rule;// our custom rule instance
} else {
This.rule = new RoundRobinRule(); //default
}
If (this.rule.getLoadBa;amcer() ! = this) {
This.rule.setLoadBalancer(this);
}
```

Public interface IRule declares the rule for the load balancer. Some known implementing classes are Abstract Load Balancer Rule, Availability Filtering Rule, Client Config Enabled Round Robin Rule, etc. Our proposed rule can be a part of new implementation class and can be quickly sent as a reference variable to the method setRule (IRule rule). The primary aspect of the algorithm would be choosing the server, which has a structure as given here.

```
public Servuer chooseServer(Object key) {
If (counter = = null) {
Counter = createCounter();
}
Counter.increment();
If (rule = = null){
Ruturn null;
}
```

```
Else{
Try{
Return rule.choose(key);
} catch (Exception e) {
Logger.warn("LoadBalancer [}{]:/
Error choosing server for key {}", name, key, e);
Retuen null;
}
}}
```

We deployed the developed microservices on give nodes in the Google cloud platform. The microservices instances were made to run in Docker containers under the clusters. Three service chains were implemented and deployed to test the service chain-oriented load balancing. Each has multiple interactive and non-interactive workloads.

In the netflix environment, when a client request arrives, the Ribbon functions as a client side load balancer. It makes use of the Eureka Server to get a list of components that provide the service. Then the corresponding service instances are accessed via the corrsponding address with the help of load balancing algorithm. The Ribbon load balancer utilizes techniques like polling, randomization and other methods to ensure the load balancing [15]. At present, it is one of the most efficient load balancing techniques existing and the proposed method is compared against this method.

7 Results and Discussion

Our experiments evaluated both interactive workloads as well as non-interactive workloads. Interactive workloads must invoke a chain of service requests deployed in multiple nodes. Without our proposed algorithm, the default load-balancing scheme experience more latency, and the throughput was inadequate for commercial enterprise applications. In our chosen application, e-commerce, adding a product to the cart is a non-interactive workload with no difference in throughput. On the other hand, if the request is for the purchase of an item, then it needs to be completed by invoking multiple services. We can see significant differences in the results, as illustrated here.

Fig. 6 indicates the request latency of a service chain without our proposed approach compared with the algorithm. In the case of longer service chains, the proposed algorithm significantly differs in latency. Fig. 7 indicates the request latency of requests for a leaf request. In the case of leaf requests, the proposed algorithm has no significant difference compared to the existing algorithm. However, we can easily infer that, in modern-day applications, the proportion of leaf requests is negligible compared to the complex service chains shown in Fig. 8. We performed multiple experiments separately with different time estimations. The results from conducted experiments demonstrate that the proposed algorithm can effectively minimize the delay in composing the response, especially during chained-service requests. Fig. 9 illustrates the changes in load balancing factors with various k factors. While the k factor (the representation of request type and average queue length across different physical machines) increases, the load balance degree tends to decrease. The results of existing approach is primarily from the unmodified version of Netflix load balancer which is available for experimentation. The setRule() method is modified in our proposed approach to obtain the results of our algorithm. The results are compared with the default implementation of the method. It will be considered in the proposed algorithm to estimate the queue waiting time and to direct the request to the optimal service instance located in a physical machine. The

arrival rate of requests plays a significant role in the performance of our proposed approach. The experiments were repeated with various request arrival rates. When the number of requests per second increases, the response time optimization appears to be at par with the conventional load balancing due to the algorithm's machine learning component. Beyond a certain point, the proposed approach exhibits only a marginal difference shown in Fig. 10. However, this can be easily overcome when the system is in use for a prolonged period and gets adequate learning. This load balancing is very critical in any cloud-native application deployment scenarios and datacenter managements. Exponentially increasing traffic and abrupt changes in the service requests are the major challenges in cloud application providers. When loads become concentrated on few specific services, the entire application faces downtime due to the operational abnormalities [16]. The effectiveness of our proposed approach can be found to be very effective in several modern applications.

Moreover, the higher request rate we experimented with had limitations in the system resources due to the limited capacity of the environment. Compared with the real time application environments, the results obtained are auspicious for most enterprise applications. Hence, in every given scenario, our proposed approach is the best method to optimize the load balancing. Particularly we considered scenarios like.

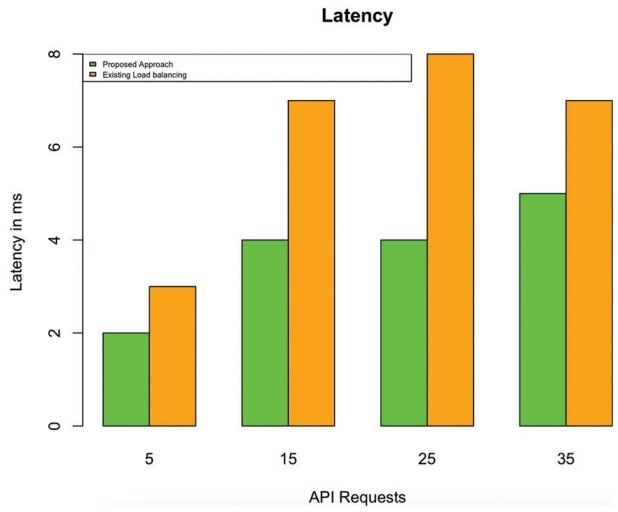

Figure 6: Latency in service chains

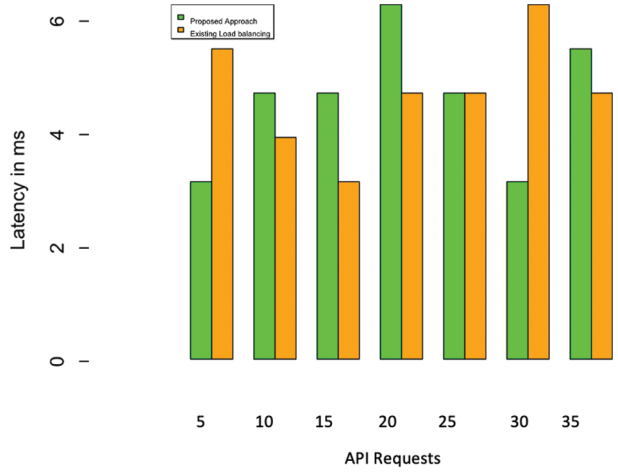

Figure 7: Latency in leaf requests

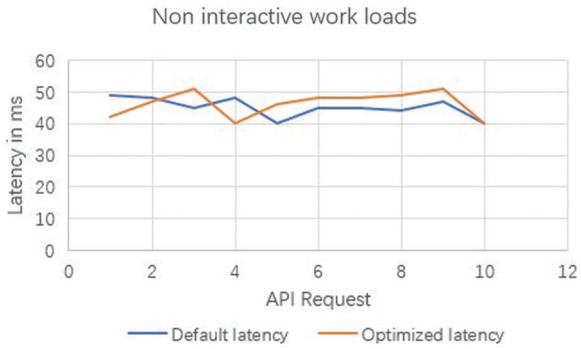

Figure 8: Latency in non-interactive workloads

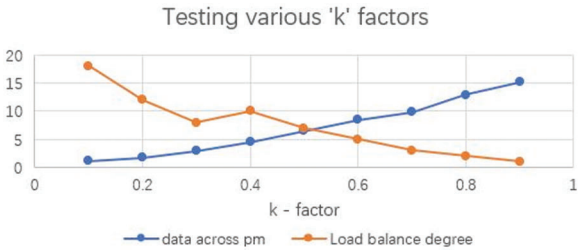

Figure 9: The load balance degree with various ‘k’ factors

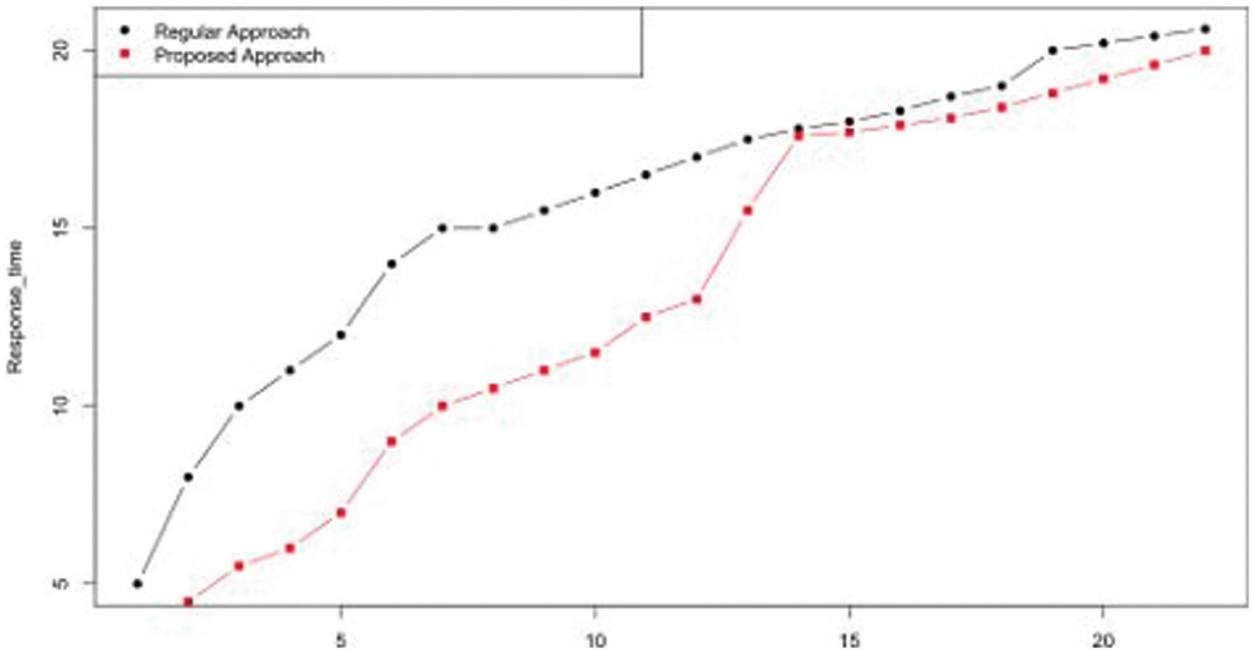

Figure 10: Request arrival rate and response time

8 Conclusion

This paper proposed an efficient approach to ensure optimum latency for cloud-native microservices applications. The selection of service instances in different machines needs to be done with a carefully designed load balancing approach to reduce the latency. This paper addresses this problem and proposes

an algorithm to dynamically evaluate the server capacity and the nature of requests in the queues. Though several load balancing algorithms are available for cloud applications, our solution has a machine learning component to analyse the previous requests to optimize the solution. Also, it specifically addresses the longer service chains in microservices applications, which have not been considered so far. Our solution has been implemented using Netflix open-source software components, and we have conducted experiments to test our approach with different scenarios. The experiments' results have shown that the proposed approach has balanced the load among multiple microservice chains to reduce the latency and phenomenally improve the end-user experience.

Acknowledgement: We would like to thank the supervisors and the anonymous referees for their kind help in this research.

Funding Statement: The authors received no specific funding for this study.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] A. Samanta, Y. Li and F. Esposito, "Battle of microservices: Towards latency-optimal heuristic scheduling for edge computing," in *Proc. 2019 IEEE Conf. on Network Softwarization (NetSoft)*, Paris, France, pp. 223–227, 2019.

[2] M. Autili, A. Perucci and L. De Lauretis, "A hybrid approach to microservices load balancing," in *Proc. Microservices*, Springer, Cham, pp. 249–269, 2020.

[3] D. Bhamare, R. Jain, M. Samaka and A. Erbad, "A survey on service function chaining," *Journal of Network and Computer Applications*, vol. 75, pp. 138–155, 2016.

[4] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta *et al.,* "Multi-objective scheduling of micro-services for optimal service function chains," in *Proc. IEEE Int. Conf. on Communications*, Paris, France, pp. 1–6, 2017.

[5] F. Wan, X. Wu and Q. Zhang, "Chain-oriented load balancing in microservice system," in *Proc. 2020 World Conf. on Computing and Communication Technologies (WCCCT)*, Warsaw, Poland, pp. 10–14, 2020.

[6] C. Guerrero, I. Lera and C. Juiz, "Resource optimization of container orchestration: A case study in multi-cloud microservices-based applications," *The Journal of Supercomputing*, vol. 74, no. 7, pp. 2956–2983, 2018.

[7] H. Zhu, H. Wang and I. Bayley, "Formal analysis of load balancing in microservices with scenario calculus," in *Proc. 2018 IEEE 11th Int. Conf. on Cloud Computing (CLOUD)*, San Francisco, CA, USA, pp. 908–911, 2018.

[8] J. Rahman and P. Lama, "Predicting the end-to-end tail latency of containerized microservices in the cloud," in *Proc. 2019 IEEE Int. Conf. on Cloud Engineering (IC2E)*, Prague, Czech Republic, pp. 200–210, 2019.

[9] Y. Liang and Y. Lan, "Tclbm: A task chain-based load balancing algorithm for microservices," *Tsinghua Science and Technology*, vol. 26, no. 3, pp. 251–258, 2020.

[10] R. Brondolin and M. D. Santambrogio, "PRESTO: A latency-aware power-capping orchestrator for cloud-native microservices," in *Proc. 2020 IEEE Int. Conf. on Autonomic Computing and Self-Organizing Systems (ACSOS)*, Washington, DC, USA, pp. 11–20, 2020.

[11] M. Rahman, S. Iqbal and J. Gao, "Load balancer as a service in cloud computing," in *Proc. 2014 IEEE 8th Int. Symp. on Service Oriented System Engineering*, Oxford, UK, pp. 204–211, 2014.

[12] X. Rui, J. Wu, J. Zhao and M. S. Khamesinia, "Load balancing in the internet of things using fuzzy logic and shark smell optimization algorithm," *Circuit World*, vol. 47, no. 4, pp. 335–344, 2020.

[13] T. Gupta and A. Dwivedi, "Data storage & load balancing in cloud computing using container clustering," *International Journal of Engineering Sciences & Research Rechnology*, vol. 6, no. 9, pp. 656–666, 2017.

[14] M. S. Hamzehloui, S. Sahibuddin and A. Ashabi, "A study on the most prominent areas of research in microservices," *International Journal of Machine Learning and Computing*, vol. 9, no. 2, pp. 242–247, 2019.

[15] H. Wang, Y. Wang, G. Liang, Y. Gao, W. Gao *et al.,* "Research on load balancing technology for microservice architecture," in *Proc. 2nd Int. Conf. on Computer Science, Communication and Network Security*, Sanya, China, pp. 08002, 2021.

[16] J. B. Lee, T. H. Yoo, E. H. Lee, B. H. Hwang, S. W. Ahn *et al.,* "High-performance software load balancer for cloud-native architecture," *IEEE Access*, vol. 9, pp. 123704–123716, 2021.