https://doi.org/10.31436/iiumej.v26i2.3089

SCALABILITY AND COST OPTIMIZATION IN LOAD-BALANCED MICROSERVICE SCHEDULING SYSTEM

SHAMSUDDEEN RABIU [1], CHAN HUAH YONG [1*], SHARIFAH MASHITA SYED-MOHAMAD [2]

[1]*School of Computer Sciences, Universiti Sains Malaysia (USM), Penang, Malaysia*
[2]*Faculty of Computer Science and Mathematics, Universiti Malaysia Terengganu, 21030 Kuala Nerus, Terengganu, Malaysia*

**Corresponding author: hychan@usm.my*

(Received: 31 October 2023; Accepted: 22 April 2025; Published online: 15 May 2025)

ABSTRACT: Microservice, a widely adopted architectural paradigm to overcome monolithic limitations, faces difficulties in efficient load balancing, scalability, and cost-effective deployment. To address these issues, we introduce a Container Microservice Load Balanced (CMLB) framework, which integrates the novel OEPTA algorithm. This framework aims to optimize microservice-based applications deployed on Docker within cloud environments. Common microservices scheduling strategies often grapple with load distribution challenges, resulting in suboptimal resource utilization. Concurrently, traditional containerization methods face difficulties reconciling trade-offs between scalability, deployment cost, and execution time. Our primary goal is to present a comprehensive solution that enhances the scalability, cost efficiency, and execution time of microservices deployment. This paper introduces a novel deployment framework for microservices, leveraging Docker for decentralized resource allocation across Microservice Controllers (MSCs). Additionally, a specialized algorithm is introduced to evaluate the cost, execution time, and availability aspects of microservice applications, enabling optimized resource allocation in a distributed manner. The evaluation results demonstrate that the CMLB framework, driven by the OEPTA algorithm, surpasses existing algorithms in achieving optimal scalability, cost efficiency, and execution times. This research provides a robust solution to enhance microservices deployment in cloud environments, effectively addressing key challenges in the field.

ABSTRAK: Mikroservis, sebuah paradigma seni bina yang diadaptasi secara meluas untuk mengatasi keterbatasan monolitik, menghadapi kesulitan dalam penyeimbangan beban yang cekap, skalabiliti, dan penyebaran kos efektif. Untuk mengatasi masalah ini, kami memperkenalkan rangka kerja Container Microservice Load Balanced (CMLB), yang mengintegrasikan algoritma OEPTA yang baru. Rangka kerja ini bertujuan untuk mengoptimumkan aplikasi berasaskan perkhidmatan mikroservis yang digunakan pada Docker dalam persekitaran awan. Strategi penjadualan mikroservis umumnya bergelut dengan cabaran pengagihan beban, yang menghasilkan penggunaan sumber daya yang kurang optimal. Pada masa yang sama, kaedah pengkontenaan tradisional menghadapi kesulitan dalam menyeimbangkan pertukaran antara skalabiliti, kos penggunaan, dan masa pelaksanaan. Matlamat utama kami adalah untuk membentangkan penyelesaian komprehensif yang meningkatkan skalabiliti, kos kecekapan, dan masa pelaksanaan dalam penggunaan mikroservis. Dalam makalah ini, kami memperkenalkan rangka kerja penggunaan yang baru untuk perkhidmatan mikroservis, dengan memanfaatkan Docker untuk peruntukan sumber terdesentralisasi merentas Pengawalan Perkhidmatan Mikroservis (MSCs). Selain itu, algoritma khusus diperkenalkan untuk menilai kos, masa pelaksanaan, dan ketersediaan aplikasi mikroservis, membolehkan peruntukan sumber dioptimumkan dalam cara yang diedarkan. Keputusan penilaian menunjukkan bahawa rangka kerja CMLB, didorong oleh algoritma OEPTA, mengatasi algoritma sedia ada dalam mencapai skalibiliti

optimum, kecekapan kos, dan masa pelaksanaan. Penyelidikan ini memberikan penyelesaian yang teguh untuk meningkatkan penggunaan mikroservis dalam persekitaran awan, menangani cabaran utama dalam lapangan dengan berkesan.

KEYWORDS: *Algorithm, Cloud-based, Container, Docker, Load balancing, Microservice.*

1. INTRODUCTION

Monolithic architecture refers to an application with a unified code base housing multiple services that interact with external systems or consumers through diverse interfaces like Web services, HTML pages, or REST API [1]. In this architectural model, all functionalities are consolidated within a single application, resulting in modules that cannot function independently [2]. This inherent characteristic of tight coupling means that all logic for handling a request operates within a single process [3]. Despite the initial advantages of ease in development, testing, and deployment for simpler applications, the drawbacks of monolithic architecture become apparent as the application becomes more intricate. The monolith's structure expands in size, transforming into a cumbersome and challenging piece of software to manage and scale [2]. As the application size and team grow, the limitations of this architecture become increasingly significant [3]:

- Complexity in understanding and modifying the application, leading to a deceleration in development speed.
- Difficulty in continuous deployment, where even minor changes necessitate the complete rebuilding and redeployment of the entire monolith.
- Challenges in scaling the application, restricted to horizontal scaling within the confines of monolithic architecture.

To overcome the challenges associated with monolithic applications and harness the benefits of Service-Oriented Architecture (SOA), the microservices architecture pattern has emerged as a lightweight subset of SOA, as exemplified by companies like Amazon [4]. In recent years, microservices have gained significant traction in the business landscape, representing an enhanced and streamlined version of SOA [5]. According to [6], microservices architecture is a specific implementation approach within SOA that facilitates the creation of flexible and independently deployable software systems. This approach typically involves breaking down a software application into smaller components that collaborate to achieve a specific, complex task, thereby facilitating easier development and maintenance [7].

The Microservices architecture represents a paradigm shift away from traditional development methods, placing emphasis on building the applications through small, autonomous services that communicate via lightweight mechanisms [6]. It's essential to note that the term "micro" in Microservices refers to the size of their contribution to the application, rather than the lines of code they encompass [2]. Therefore, the architecture can be understood as a set of small services with precise tasks that interact to achieve users' goals through standard communication channels [8, 9, 10]. This architectural approach offers a strategy for developing a cohesive application as a collection of small services. Each service operates independently in its dedicated process and communicates through lightweight mechanisms, such as HTTP APIs [10]. These Microservices are designed around specific business capabilities, enabling them to be independently deployable through automated deployment processes [3]. Lately, there has been a surge in using containers to distribute microservices across various cloud locations [11].

Containers, an emerging virtualization technology, are gaining popularity over virtual machines (VMs) due to their superior performance, lightweight design, and enhanced scalability [12]. These containers act as comprehensive packages, bundling applications with their dependencies, facilitating easy and consistent deployment across diverse environments. Dependencies, including binaries, libraries, and configuration files, are essential for the application's functionality [13]. Containers encapsulate self-contained, deployable components of applications, and may include middleware and business logic in the form of binaries and libraries [10]. Container engines, such as Docker, leverage containers as portable entities for packaging applications. This shift emphasizes the critical need to manage container dependencies [14]. The application consists of individual, self-contained services that operate in their own processes and communicate through a lightweight mechanism [14].

To mitigate the risk of a single instance becoming a bottleneck or a potential point of failure [16], a load-balanced microservice scheduling system distributes requests for a specific service across multiple instances of that service. Typically, a load balancer, positioned in front of the service instances, achieves this distribution by directing incoming traffic to the least busy instance [17]. The primary objective of load balancing is to optimize resource utilization, enabling the system to handle increased traffic levels with minimal downtime or performance slowdowns [18]. In microservices, load balancing plays a crucial role in maintaining uninterrupted services even if one or more components fail, utilizing failover [17]. This involves adding and removing instances of applications in a balanced manner, preventing failures [19]. Numerous research studies have aimed to improve Quality of Service (QoS) in Container Microservices through load balancing [20]. However, many of these studies have found existing methods ineffective in enhancing user QoS as the methods often rely on queuing systems, leading to issues like increased network traffic, longer processing times, server overloads, and high deployment costs [6,17,18,19,21]. This results in sudden load spikes, disrupting system balance and degrading performance. While load balancing traditionally considers factors like traffic and performance constraints, adjusting computational resources dynamically to optimize costs based on load changes is also crucial [17]. Hence, applying load balancing strategies becomes crucial for optimizing performance in such scenarios.

To address the challenges associated with load-balanced microservice scheduling systems, we introduced a pioneering framework called CMLB. This framework is designed to optimize server overloads, manage traffic spikes, and enhance the cost efficiency of microservices. We developed a load balancing algorithm to determine the service deployment cost, improve reliability, and ensure the availability of microservice applications. In our proposed system, the allocation and management of resources for applications are decentralized and orchestrated by the Master Load Balancer (MLB), operating through Local Load Balancers (LLB) on Microservice Controllers (MSCs). The MSCs play a vital role in decision-making related to resource allocation, requesting resources for Execution Containers (ECs), monitoring task progress on ECs, and overseeing the life cycle of ECs. Simultaneously, the ECs are responsible for executing assigned tasks, providing progress updates to the MSCs, and comparing their performance against expected outcomes. The primary contributions of this paper are outlined as follows:

- Proposed a novel Container Microservice Load Balanced (CMLB) framework, designed for deploying microservice-based applications with Docker, incorporating autonomous resource allocation on each controller in a distributed manner.
- Presented an OEPTA algorithm to optimize the cost, execution time, and scalability aspects of microservice applications.

- Implemented and evaluated the OEPTA algorithm's effectiveness, showcasing its capacity to deliver positive outcomes regarding traffic spike management and server overload minimization.

The remaining sections of this paper are organized as follows: Section 2 presents a literature review on related works. Section 3 offers an overview of the methodology. Section 4 presents results and discussions. Section 5 provides a summary of the conclusion and outlines future work. Table 1 highlights key processes, components, and challenges within the system. Figure 1 summarizes the deployment of microservice applications with Docker Container, while Figure 2 describes the elements of the cluster and introduces the concept of a Global ResourceSpace. Figure 3 illustrates the workflow for application deployment and introduces the Optimize EC Placement and Task Assignment Algorithm (OEPTA) for resource allocation decisions. The presented frameworks aim to address the challenge of deploying and managing microservice applications efficiently within clusters. The key problems include optimal resource allocation, load balancing, and timely task execution. The primary objectives are to ensure minimal delays, maximize resource utilization, and maintain overall system performance. The introduced OEPTA algorithm balances application demands, available resources, and cluster configurations to make informed decisions for microservice deployment and execution. Table 1 lists the notations used in this section to aid understanding of the algorithm.

Table 1. Summary of the Notations used and their descriptions.

Notations	Descriptions
Adr	Application Deployment Request
MS	Microservice
$MSCs$	Microservice Controllers
ECs	Execution Containers
PM	Physical Machine
N^p	Set of physical nodes
Λ	Set of applications needed to be deployed
G^a	Set of all clusters in which the application is deployed.
M^a	Set of microservice M^a for application G^a
$R^a(u)$	Total resources available on PM 'i' for microservice 'u' in application 'a'.
$CR^a(u)$	Resource requirement of microservice 'u' in application 'a'.
$CN^{node}(i,a)$	Cost associated with deploying application 'a' on physical machine (PM) 'i' in the cluster.
C^0	Cost of using resources
$x(i,a,u)$	Resource usage of microservice 'u' in application 'a' on PM 'i'.
$l^a(k,u)$	Processing time of task 'k' for microservice 'u' in application 'a'
$l^p(k,i)$	Predefined execution time requirement for task 'k' on PM 'i'.
$l(k,i)$	Communication time between tasks on PM 'i'.
$RR^p(i)$	Predefined resource requirement for cluster 'i'.
$C^p.s_i$	Processing cost of the PM 'i'.
$C^l(k)$	Cost of accessing the PM, which may depend on the microservice 'k'.

2. LITERATURE REVIEW

In recent years, an expanding body of research has focused on deploying and managing microservice containers [17, 19, 20, 22]. The Microservices architecture has garnered significant attention for its adaptability, cost-effectiveness, and scalability, leading large enterprises to deploy microservices across diverse cloud locations [17, 20, 22]. Containers, known for their lightweight nature compared to virtual machines, enable easy downloads and swift deployments [11,23]. Adopting a microservices architecture presents several advantages, including reduced interdependence between services, faster recovery in the face of catastrophic events,

and heightened reliability achieved by isolating the impact of failures to a small service segment [24]. However, as applications scale, challenges arise, notably in increased API calls. This necessitates the implementation of effective load balancing solutions to manage API calls across the architecture.

Contributing to this domain, [17] developed a microservices architecture utilizing Docker containers to enhance scalability and elasticity in the cloud computing environment. Their approach aims to reduce deployment and operational costs while meeting service delay requirements. Expanding upon their work, our research enhances their framework by incorporating additional parameters, including execution time, traffic spikes, and cost considerations. We further augment the framework by distributing requests evenly across all PMs through a master load balancer on the client side. This extension is designed to ensure Quality of Service (QoS) for developers, responding to the growing demand for new software [25].

A key concern in container microservices cloud-based systems revolves around the intricacies of load balancing. This process entails the equal distribution of workloads across servers to forestall service failures, minimize response time, alleviate downtime, and safeguard against data loss [17, 26, 27]. Effective load balancing is paramount for averting resource overload, enhancing performance, handling unforeseen traffic spikes, curtailing response time, and optimizing resource utilization [27]. In a correlated investigation, an inventive Load Balancing Ant Colony Optimization (LBACO) algorithm was devised to distribute workloads throughout the entire system, thereby reducing the makespan [21]. Meanwhile, [28] introduces a groundbreaking approach to enhance performance and diminish latency by leveraging SmartNICs on edge servers for middlebox processing. Their SmartLB methodology deploys a load balancer and an auto scaler entirely on the SmartNIC, resulting in judicious decisions and decreased CPU load. Despite numerous existing methodologies for optimizing load balancing, container placement, application deployment costs, operational costs, service failure, and traffic issues, many treat these challenges as knapsack problems, often neglecting essential load balancing features. Therefore, integrating these crucial features into our system will significantly enhance its overall performance.

3. METHODOLOGY

This section outlines the systematic scalability and cost optimization approach in the proposed Load-Balanced Microservice Scheduling System. It details the overarching strategy for developing and implementing the microservices framework with Docker containers and evaluates its performance. The methodology involves creating a simulation model, implementing a novel scheduling algorithm, and thoroughly evaluating results based on predefined criteria.

3.1. System Model

To achieve a balanced distribution of requests across microservice instances, we defined this process in a structured manner using our equations. These equations serve as a foundation for the OEPTA algorithm, enabling it to make well-informed choices on resource distribution, task delegation, and load equilibrium. The algorithm aims to minimize costs, ensure scalability, and fulfill execution time criteria for deploying applications.

Our system model analyzes a physical network comprising a collection of Physical Machines (PMs). Each PM possesses finite physical resources, with our focus being on computational resources as a representative illustration of the resource allocation challenge. We assume uniformity in capacity and pricing among the PMs. However, the PMs may install different libraries in advance to support different microservices for applications.

https://doi.org/10.31436/iiumej.v26i2.3089

In the OEPTA algorithm, there are three equations used to guide the decision-making process:

a. Cost

$$CN^{node}(i,a) = C^p.s(i) + \sum_k C^l(k) + \sum_{u \in M^a} C^0.\frac{x(i,a,u).CR^a(u)}{R^a(u)} \quad (1)$$

Eq. (1) calculates the cost of deploying an application on a specific physical machine (PM) in a cluster. It considers various factors such as the processing cost of the PM ($C^p.s(i)$), the cost of accessing the PM ($C^l(k)$), and the cost of using resources on the PM ($C^0.\frac{x(i,a,u).CR^a(u)}{R^a(u)}$). By evaluating the cost for each PM in the cluster, the algorithm can select the PM with the lowest cost for the given application.

b. Scalability

$$\sum_{G^a \in \Lambda} \sum_{u \in M^a} \frac{x(i,a,u).CR^a(u)}{R^a(u)} < RR^p(i), \forall i \in N^p \quad (2)$$

Eq. (2) is the scalability equation, which assesses the overall resource utilization across all clusters for a specific application. It calculates the ratio of resource usage $\frac{x(i,a,u).CR^a(u)}{R^a(u)}$ for each microservice) u in the application across all clusters (G^a) and compares it to a predefined resource requirement ($RR^p(i)$). The algorithm proceeds to the next step if the total resource utilization meets the scalability requirement. Otherwise, it revisits the allocation and assignment process for better resource utilization.

c. Execution Time

$$\sum_{G^a \in \Lambda} \frac{l^a(k,u).x(i,a,u)}{R^a(u)} - l(k,i) \leq l^p(k,i), \forall i \in N^p, u \in M^a \quad (3)$$

Eq. (3) evaluates the expected execution time for each task (u) of an application on a specific $PM_{i,}$. It considers the processing time for the task on the PM ($l^a(k,u)$ and the communication time between tasks ($l(k,i)$). The equation sets a constraint ($\leq l^p(k,i)$) on the execution time, ensuring that the execution time for each task meets the predefined requirement. If the execution time constraint is satisfied for all tasks, the deployment is considered successful. Otherwise, the algorithm revisits the allocation and assignment process to optimize task execution time.

These equations help the OEPTA algorithm make informed decisions regarding resource allocation, task assignment, and load balancing, aiming to minimize costs, ensure scalability, and meet application deployment execution time requirements.

3.2. A Framework for Deploying Cloud-Based Microservice Container Applications with Docker

This section introduces an adapted Container Microservice Load Balanced (CMLB) framework that integrates the innovative OEPTA algorithm. This framework aims to improve the performance of microservice-based applications deployed on Docker in cloud environments. The system uses microservices on ECs to process application requests within a framework. Resource allocation and application management are decentralized, with Registry and Service Discovery coordinating the process through a Load balancer on MSCs. The MSCs are

https://doi.org/10.31436/iiumej.v26i2.3089

responsible for making decisions on resource allocation, requesting resources for ECs, monitoring task progress on ECs, and managing the life cycle of ECs. Once a task is complete, the EC reports back to the MSC on the progress compared to what was expected.

Fig. 1 shows the deployment of microservice applications with Docker containers. The process begins with the user sending an application deployment request ($Adr1$) to the gateway. The gateway then sends the request to the registry and Service Discovery (RSD) system, which registers and assigns the request to the first cluster in the resource table (since the cluster placement is done automatically, with a priority given to the least busy). Once the cluster receives the registration request, the load balancer is notified and will register and update the status of the microservices controllers and execution containers to execute the job. If more resources are needed, the load balancer will select the best candidate for the available cluster resource status in the global resourcespace to avoid delays or traffic. When a resource is picked from the global resourcespace, it disappears, and the following available resource is released for use. While the load balancer manages jobs, the other jobs in the queue will be assigned to subsequent clusters that are available concurrently, following the same process.

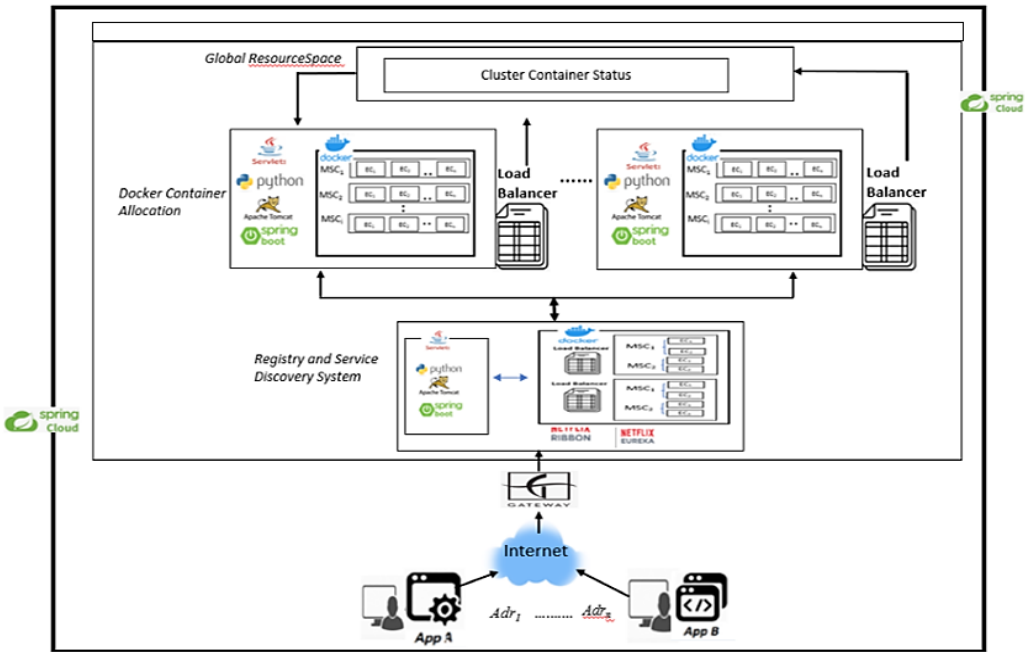

Figure 1. Framework for deploying microservice applications with Docker containers

Fig. 2 illustrates the main components of PM in a Cluster. Each PM within the cluster has a host operating system that runs Docker. This Docker engine is responsible for maintaining the container's operating environment, embedding containers, and isolating containers operating on the same PM. The Registry and Service Discovery system that registers and directs requests to the PM is also introduced. Each PM includes a load balancer that registers and updates the status of the MSCs and ECs operating on that PM. Furthermore, a Global ResourceSpace is introduced, allowing for the scaling up or down of the resources if needed.

The workflow of deploying the application deployment requests is illustrated in Figure 3. When an application request is sent to the registry and service discovery system, the registry and Service Discovery (RSD) system will register and assign the request to the load balancer to manage the microservice controllers and execution containers in the physical machine within the clusters. If more resources are needed, the load balancer will select the best candidate from the available global resourcespace to avoid delays or traffic. When a resource is selected from

https://doi.org/10.31436/iiumej.v26i2.3089

the global resourcespace, it disappears, and the following available resource is released from the cluster. While the load balancer manages cluster one, the other jobs in the queue will be assigned to subsequent clusters that are available concurrently, following the same process. Based on the application's requirements and available resources on PMs in the clusters, the load balancer makes resource allocation decisions using the proposed Optimize EC Placement and Task Assignment Algorithm (OEPTA).

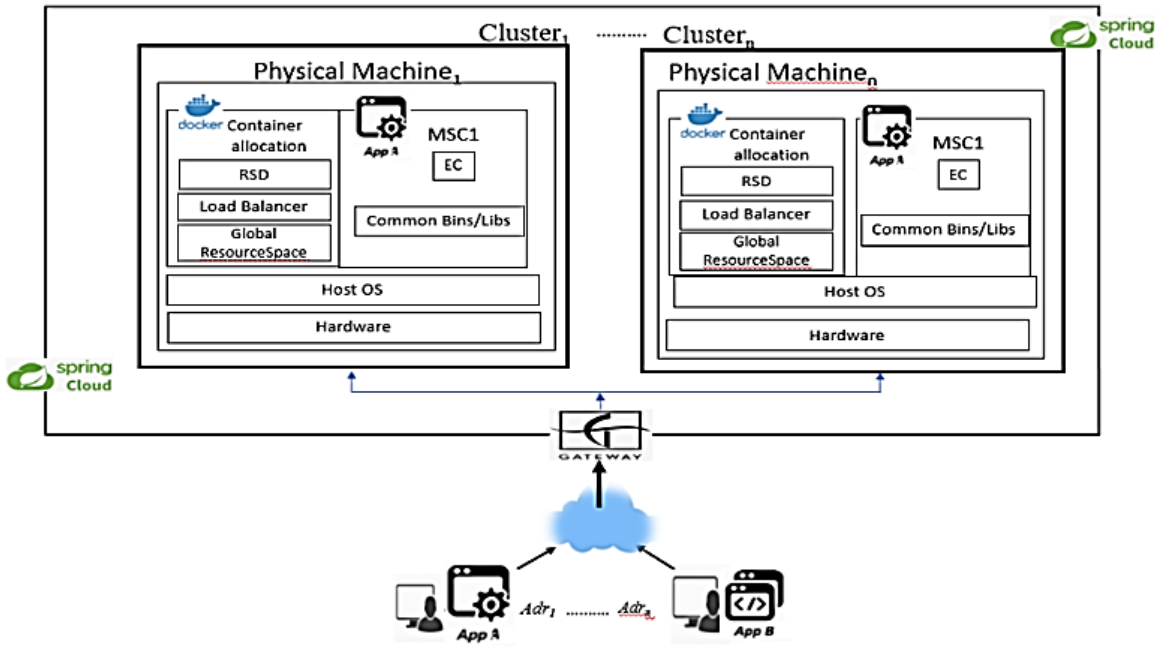

Figure 2. Components of physical machines ($PM_1, \ldots, PM_i$) in the clusters embed microservice controllers and execution containers.

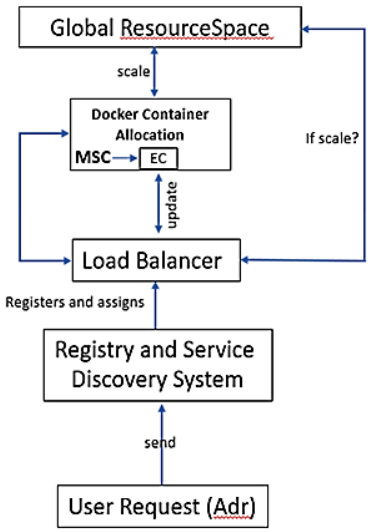

Figure 3. Workflow for deploying the application deployment requests.

3.3. Develop a Simulation Model and Implement the Proposed Algorithm.

This section outlines the overall design of the cloud-based container microservices framework and the specific implementation of the microservices function. The simulation model is established following the previously mentioned methodology. To assess the efficiency of the suggested algorithm, Netflix, Spring Boot, and Spring Cloud are employed as microservice

https://doi.org/10.31436/iiumej.v26i2.3089

implementations and communication simulation tools. To address the scalable distributed-systems problems, the registry and service discovery system, load balancer, and global resource space have all been utilized, resulting in practical solutions for scheduling requests. The design framework is anticipated to boost the performance of the CBCM system.

3.3.1. Algorithm Implementation

A comparison of the algorithms under consideration reveals some of their strengths and weaknesses. All of the algorithms are effective at resolving microservice scheduling issues [24, 26, 27]Although they are time-consuming and have a slow convergence speed for solving complex scheduling problems, the algorithms have been modified to address more specific issues such as cost, server overload, traffic spikes, and performance issues.

The OEPTA (Optimized Executive Containers Placement Task Assignment) algorithm optimizes the placement and assignment of executive containers in a distributed computing environment. It initializes variables and data structures, sets up clusters and physical machines, and receives application deployment requests. The algorithm registers and assigns requests, calculates costs for each physical machine, and selects the best candidate based on cost and other factors. Tasks are assigned to the selected machine, and load balancing is performed. A scalability check ensures resource utilization meets requirements, and execution time is calculated for each task. The system is updated by releasing resources, marking them as unavailable, and updating load balancing. The process is repeated for the remaining deployment requests. Overall, OEPTA aims to achieve efficient task execution through optimized placement and assignment of executive containers.

Algorithm: (Optimized Executive Containers Placement Task Assignment)

Input: User Application deployment requests $\left[Adr\middle|1, Adr_{2,} \dots \dots \dots \dots \dots \dots Adr_{n,}\right]$

Output: $min\{NT, Cost, Overloading\}$

1: Initialization:

2: Initialize necessary variables, data structures, and parameters.

3: Set up the clusters and their associated physical machines PMs.

4: Set up the global resourcespace.

5: For each application deployment request $Adr_{i,}$ in $\left[Adr\middle|1, Adr_{2,} \dots \dots \dots \dots \dots \dots Adr_{n,}\right]$:

6: Application Deployment Request:

7: Receive $Adr_{i,}$ and its associated parameters.

8: Register and Assign Request:

9: Send $Adr_{i,}$to the registry and service discovery system (RSD).

10: RSD registers and assigns $Adr_{i,}$ to the load balancer (LB).

11: Resource Allocation and Task Assignment:

12: Calculate the cost for each PM in the cluster for $Adr_{i,}$ using Eq. 1:

13: $$CN^{node}(i,a) = C^{p}.s(i) + \sum_{k} C^{l}(k) + \sum_{u \in M^{a}} C^{0}.\frac{x(i,a,u).CR^{a}(u)}{R^{a}(u)}$$

14: Select the best candidate from the available resources in the global resourcespace, considering cost and other factors.

15: Assign $Adr_{i,}$'s tasks to the selected PM, updating the resource allocation.

16: Release the next available resource from the cluster's resource pool.

17: Cluster Management and Load Balancing:

18: Manage the microservice controllers and execution containers on the assigned PM.

19: If there are other jobs in the queue, assign them to subsequent clusters that are available concurrently, following the same process.

20: Scalability Check:

21: Calculate the total resource utilization across all clusters for Adr(i) using Eq. 2:

22: $$\sum_{G^a \in \Lambda} \sum_{u \in M^a} \frac{x(i,a,u).CR^a(u)}{R^a(u)} < RR^p(i), \forall i \in N^p$$

23: If the condition in Eq. 2 is satisfied, proceed to the next step. Otherwise, revisit the allocation and assignment process.

24: Execution Time Calculation:

25: Calculate the execution time for each task on the assigned PM using Eq. 3:

26: $$\sum_{G^a \in \Lambda} \frac{l^a(k,u).x(i,a,u)}{R^a(u)} - l(k,i) \leq l^p(k,i), \forall i \in N^p, u \in M^a$$

27: If the condition in Eq. 3 is satisfied for all tasks, the deployment is successful. Otherwise, revisit the allocation and assignment process.

28: Update the System:

29: Release the assigned resources from the cluster and update the global resourcespace accordingly.

30: Mark the assigned resources as unavailable in the cluster.

31: Update the load balancer and cluster management for the released resources.

32: Repeat the process for the remaining application deployment requests in the queue.

3.3.2. Algorithm Description

As shown in the algorithm, Steps 1 to 5 involve initializing necessary variables, data structures, and parameters required for the algorithm. Includes setting up the clusters and their associated physical machines (PM) and establishing the global resource space. Step 6 is the Application Deployment Request. The algorithm receives an application deployment request and associated parameters in these steps. The request contains information about the application that needs to be deployed. Steps 8-10 are the Register and Assign Request. The algorithm sends the deployment request to the registry and service discovery system (RSD) for registration and assignment in these steps. The RSD registers the request and assigns it to the load balancer, which handles the distribution of tasks. Steps 11-16, Resource Allocation and Task Assignment; These steps involve calculating the cost for each PM in the cluster for the given application using the cost equation (Eq. 1). The algorithm selects the best candidate PM from the

https://doi.org/10.31436/iiumej.v26i2.3089

available resources in the global resource space based on factors like cost and other considerations. And the application's tasks are then assigned to the selected PM, and the resource allocation is updated. The following available resource is also released from the cluster's resource pool. Steps 17-19, Cluster Management and Load Balancing; In these steps, the algorithm manages the microservice controllers and execution containers on the assigned PM. If there are other jobs in the queue, the algorithm assigns them to subsequent clusters that are available concurrently, following the same process. Steps 20-23, Scalability Check; The algorithm calculates the total resource utilization across all clusters for the assigned application using the scalability equation (Eq. 2). It checks if the resource utilization meets the predefined scalability requirement $RR^{p(i)}$ for all clusters. If the condition is satisfied, the algorithm proceeds to the next step. Otherwise, it revisits the allocation and assignment process to optimize resource utilization. Steps 24-27, Execution Time Calculation; These steps involve calculating the execution time for each task on the assigned PM using the execution time equation (Eq. 3). The algorithm considers the processing time for the task and the communication time between tasks. It checks if the execution time constraint is satisfied for all functions. If yes, the deployment is considered successful. Otherwise, the algorithm revisits the allocation and assignment process to optimize task execution time. Steps 28-31, Update the System; Once the deployment is successful, this step involves releasing the assigned resources from the cluster and updating the global resource space accordingly. The assigned resources are marked as unavailable in the cluster, and the load balancer and cluster management are updated for the released resources. Step 32, Repeat for Remaining Deployment Requests: The algorithm repeats the process for the remaining application deployment requests in the queue. It continues deploying applications and optimizing resource allocation until all requests have been processed.

Overall, the OEPTA algorithm performs a systematic and optimized placement and assignment of application tasks on available physical machines in clusters, considering cost, scalability, and execution time requirements. It aims to efficiently utilize resources and ensure the successful and optimized deployment of applications.

3.4. Evaluation

Through trace-driven simulation studies, we evaluate the performance of our OEPTA algorithm in various contexts. All evaluations are based on actual Google Cluster Traces. In various aspects, we compare the OEPTA algorithm to four strategies implemented in our paper, including EPTA, Spread, Binpack, and Random. The allocation and management of resources for applications are decentralized and performed by the load balancer on MSCs via the registry and service discovery system. An MSC decides on resource allocation, requests resources for ECs, monitors task status on ECs, and manages the life cycle of ECs. ECs complete the assigned tasks and report to the MSC on the status of their task execution in comparison to the expected progress. We evaluate and compare the performance of our microservice by creating job requests at random using non-load balancing and load balancing methods to display the execution time that results from the system's random distribution.

3.4.1. Load Balanced Vs Non-Load Balanced

Load-balanced microservice scheduling systems improve scalability, availability, and resilience, but add complexity and infrastructure. Non-load-balanced systems are less complicated, but they can introduce a single point of failure and do not provide the same level of scalability and resilience [30]. The choice between the two approaches will be determined by the system's specific needs and the acceptable trade-offs between complexity and availability.

https://doi.org/10.31436/iiumej.v26i2.3089

We first evaluate our algorithms randomly using non-load-balancing methods, then with a load-balancing system, to examine the algorithm's behavior in both cases.

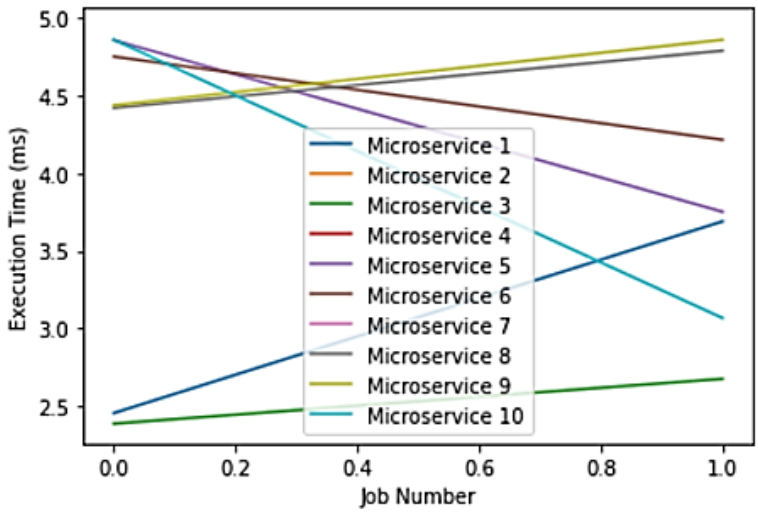

Figure 4. Microservice Non-load Balanced execution time with varied number of jobs.

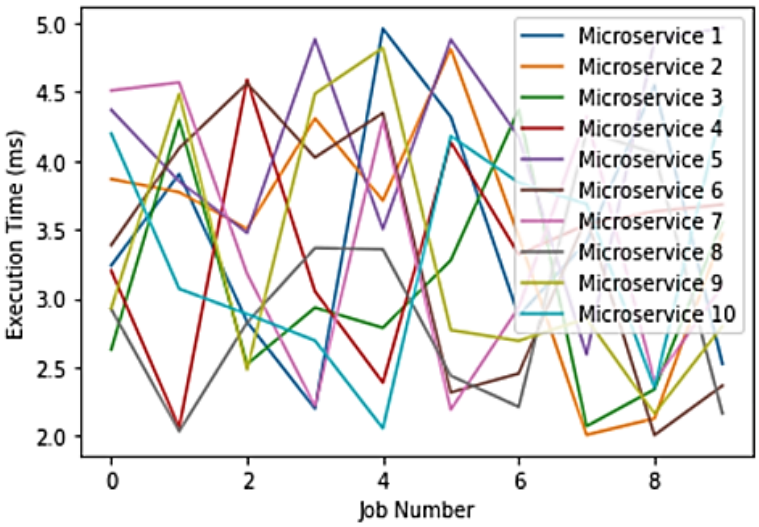

Figure 5. Microservice Load-balanced execution time with a varied number of jobs.

Fig. 4 shows the non-load-balanced algorithm used to test the number of microservices jobs against the execution time to see how the system behaved and changed compared to the load-balanced algorithm in Fig. 5.

Fig. 6 depicts the difference between non-load-balanced and load-balanced microservice job execution. We compared the number of microservices against the execution time of each microservice to see how the CBCM system behaved when load-balanced or not. Based on the figures above, load and non-load balancing are two distinct approaches to scheduling micro-services in a distributed system. It also shows that we require a load balancing system to reduce application deployment costs and execution time.

https://doi.org/10.31436/iiumej.v26i2.3089

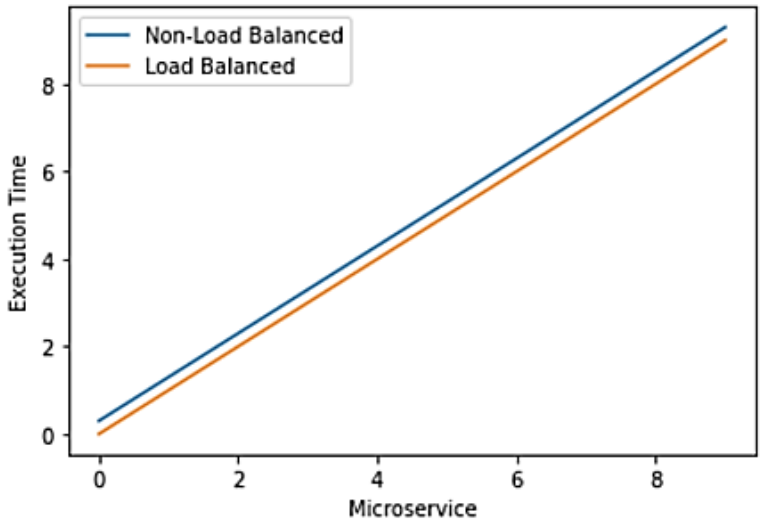

Figure 6. Microservice Load Balanced Vs Non-Load Balanced execution time with varied microservice

3.4.2. On the Number of Scaled Microservices

We assumed we had 200 microservices and used only 150 microservices (Active). The inactive will be pushed to cache load balancing as shown in Fig. 3. In case the computing resource requests overload the active microservices and have high traffic, then the system will up-scale from the cache memory (inactive) automatically. The higher the number of microservices available, the higher the active nodes. Our OEPTA algorithm's number of scaled microservices is being compared to four strategies. We looked at the active rate in the network and the total number of active microservices, while the number of available microservices ranges from 60 to 140. Fig. 7 demonstrates that OEPTA outperforms other algorithms regarding microservice scalability, whereas Spread is the most expensive. The expenses associated with deploying applications using the three Docker swarm strategies are somewhere between those of OEPTA and EPTA. When the number of scaled microservices increases from 60 to 140, the total number of active microservices used by OEPTA decreases slightly. With more microservices available, finding a better PM to handle requests becomes more difficult. However, as illustrated in the figure, the number of scaled microservices used by other strategies and algorithms increases since they occupy more microservices.

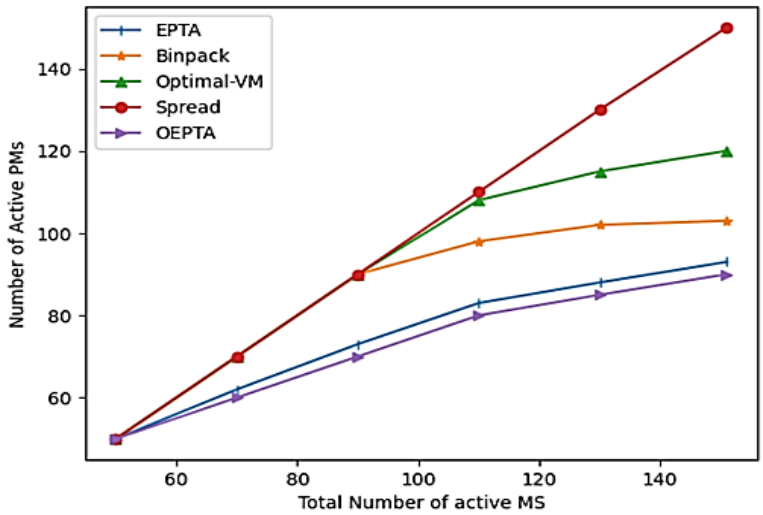

Fig. 7. Number of scaled microservices with a varied number of active microservices

https://doi.org/10.31436/iiumej.v26i2.3089

3.4.3. On the Number Application Deployment Cost

Fig. 8 shows the costs of deploying applications using five strategies and algorithms. The data reveals that OEPTA has the lowest cost, while ETPA has a considerably higher cost. Binpack has the lowest deployment cost among the three Docker Swarm strategies, while spread has the highest. This is because Binpack places microservices from the same application on the same physical machine, whereas Spread and Random distribute microservices across the network for load balancing. As a result, Binpack reduces the communication costs between microservices. The data in the figure indicates that OEPTA outperforms other algorithms in terms of total deployment cost, whereas Optimal-VM has the highest deployment cost. The application deployment costs of the three Docker Swarm strategies fall between those of OEPTA and EPTA. As the number of microservices increases from 60 to 140, the total deployment costs of OEPTA slightly decrease. This is because a larger pool of microservices increases the likelihood of finding a better microservice to place. Conversely, the total deployment costs of other strategies and algorithms increase because they utilize more microservices.

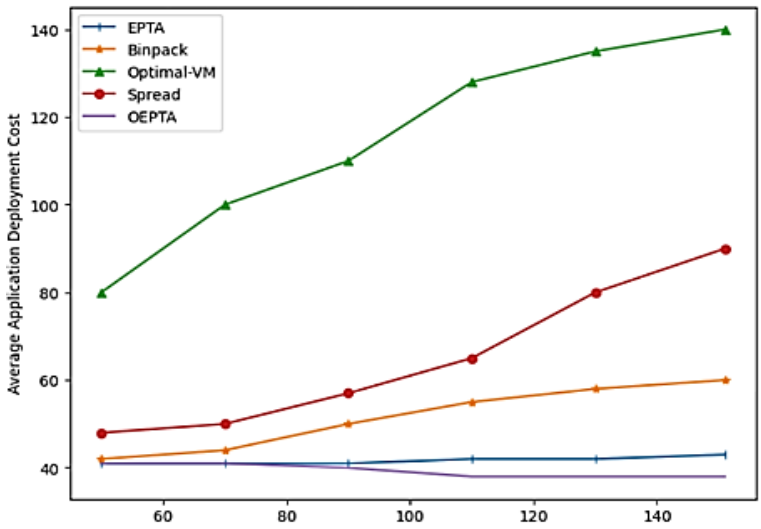

Figure 8. Average application deployment cost with a varied number of microservices.

3.4.4. On the number of Execution time

As depicted in Fig. 9, utilizing the linear programming (LP) solver causes a rise in the execution time of EPTA and Optimal-VM as the number of microservices within an application grows. To improve performance, we have devised a specialized solver customized to their problem and replacing the typical LP solver used in the EPTA algorithm. This new solver is controlled by a load balancer, resulting in optimized execution time. Fig. 9 compares the execution times of the five strategies and algorithms. The figure indicates that as the number of microservices grows, Optimal-VM and EPTA algorithms exhibit slightly better performance than OEPTA. Among the four, Spread has the shortest execution time. Interestingly, as the execution time increases from 0.2 to 2.0, the total number of microservices in OEPTA decreases compared to the other algorithms. This is because Optimal-VM and EPTA employ linear problems and the LP solver. However, OEPTA incrementally expands the search area instead of taking the entire physical network as input, resulting in a time complexity that does not exponentially increase as the network scales.

https://doi.org/10.31436/iiumej.v26i2.3089

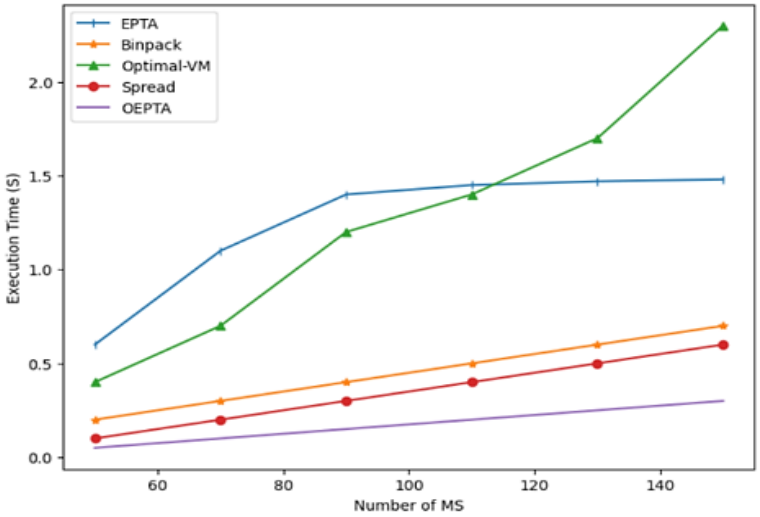

Figure 9. Average execution time with a varied number of microservices.

The paper employs a benchmarking approach to evaluate the proposed methodology against four Docker Swarm strategies and a Hypervisor-based VM embedding algorithm. It emphasizes state-of-the-art results in container microservice cloud-based systems, covering deployment costs, scaled microservices, and execution time (Wan et al. 2018). Detailed comparisons with a VM placement algorithm and three Docker Swarm strategies, utilizing real data traces, validate the proposed schemes. The paper explores strategies and algorithms, assessing deployment costs and overall performance. The OEPTA algorithm outperforms EPTA and other strategies in deployment cost, scaled microservices, and execution time, as shown in Table 2.

Table 2. Algorithm Comparison Results

Algorithms	Microservice Scalability	Deployment Cost	Execution Time
OEPTA	Outperforming EPTA slightly decreased as the total number of active microservices increased (see Figure 7).	It demonstrates the lowest cost compared to all strategies. Deployment costs decrease slightly with more microservices (refer to Figure 8).	Demonstrates competitive execution time performance, showcasing optimized execution time and successful deployment with efficient task execution (refer to Figure 9).
EPTA	Demonstrates higher microservice scalability costs than OEPTA (refer to Figure 7).	It is considered to have a higher deployment cost than OEPTA (refer to Figure 9).	It demonstrates higher execution time, which is attributed to the utilization of a linear programming (LP) solver compared to OEPTA (refer to Figure 9).
Optimal-VM	Demonstrates the highest microservice scalability among the three strategies (Binpack, EPTA, and OEPTA), but is still lower than spread. Consequently, the number of active microservices in the OEPTA algorithm surpasses all four strategies (refer to Figure 7).	Proven to have the highest total deployment cost; however, OEPTA surpasses other strategies, including Optimal-VM (refer to Figure 8).	Deploys linear problems and an LP solver, causing an escalation in execution time with the growing number of microservices, demonstrating slightly inferior performance compared to OEPTA (see Figure 9).
Binpack	Compared to EPTA and OEPTA strategies, more scaled microservices with	Demonstrates optimization of deployment costs by consolidating microservices	It exhibits the shortest execution time compared to EPTA and Optimal-VM strategies

	varying active microservices were demonstrated. Nevertheless, it exhibits lower scalability than the Optimal-VM and Spread strategies (see Figure 7).	from the same application on the same physical machine, leading to a lower deployment cost than Spread and Optimal-VM but higher than OEPTA (see Figure 8).	while displaying higher execution time than Spread and OEPTA (see Figure 9).
Spread	It demonstrates higher microservices scalability among all four strategies (see Figure 7). It distributes microservices across the network for load balancing, increasing deployment costs between microservices.	Compared to Binpack, higher deployment costs are the highest among the three Docker Swarm strategies (see Figure 8).	Demonstrates the shortest execution time compared to Binpack, EPTA, and Optimal-VM, whereas OEPTA emerges as the shortest among these compared to Spread (see Figure 9).

Table 2 presents a comparison with Wan et al. (2018), illustrating that our newly developed optimized algorithm (OEPTA) surpasses the adapted (EPTA) algorithm in terms of microservice scalability, deployment cost, and execution time. This highlights the efficacy of the OEPTA algorithm, emphasizing its superior scalability, optimized execution time, and successful deployment for efficient executional tasks, surpassing other strategies and algorithms in this domain.

4. CONCLUSION AND FUTURE WORK

We developed the OEPTA algorithm and a cutting-edge framework that minimizes the cost of deploying microservice applications while addressing the issue of load-balanced microservice scheduling systems. We use load balancing tools and libraries to describe our framework thoroughly, and we described our improvement in terms of pertinent metrics like execution time, application deployment cost, and scalability. We created a load balancing algorithm for container microservice scheduling optimization to ascertain the service deployment cost, reliability, and availability of the microservice application. The distribution and control of resources for applications occur in a decentralized manner. By conducting a comparative analysis, we confirmed the effectiveness of the proposed strategies. We found that the OEPTA algorithm delivered good results in terms of optimizing costs, traffic spikes, and server overload. These measures can effectively balance user requests for deploying applications and enhance the performance of the cloud-based container microservice system. In future work, we will incorporate load balancing and auto-scaling features by leveraging a multi-objective algorithm. This approach will consider additional optimization objectives, including latency and CPU utilization. By doing so, we aim to address more refined challenges and enhance the overall performance of container-based microservice cloud systems, ultimately improving users' quality of service (QoS).

ACKNOWLEDGMENT

The authors would like to acknowledge and thank the Research University Grant (1011/PKOMP/8014076) from Universiti Sains Malaysia (USM) and the Talent and Publications Enhancement Research Grant (TAPERG/2023/UMT/2223) titled “Empirical Analysis of Software Maintainability Metrics in DevOps Environments” from Universiti Malaysia Terengganu (UMT) for supporting this publication.

REFERENCES

[1] O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures," *18th IEEE Int. Symp. Comput. Intell. Informatics, CINTI 2018 - Proc.*, pp. 149–154, 2018, doi: 10.1109/CINTI.2018.8928192.

[2] L. De Lauretis, "From monolithic architecture to microservices architecture," *Proc. - 2019 IEEE 30th Int. Symp. Softw. Reliab. Eng. Work. ISSREW 2019*, pp. 93–96, 2019, doi: 10.1109/ISSREW.2019.00050.

[3] F. Ponce, G. Marquez, and H. Astudillo, "Migrating from monolithic architecture to microservices: A Rapid Review," *Proc. - Int. Conf. Chil. Comput. Sci. Soc. SCCC*, vol. 2019-Novem, 2019, doi: 10.1109/SCCC49216.2019.8966423.

[4] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, and S. Gil, "Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud Evaluando el Patrón de Arquitectura Monolítica y de Micro Servicios Para Desplegar Aplicaciones en la Nube," *10th Comput. Colomb. Conf.*, pp. 583–590, 2015.

[5] T. Erl, J. Fontenla, M. Caeiro, and M. Llamas, "Web Services and Contemporary SOA," *Serv. A rch itectu re Connceptsts , Technol. , Des.*, pp. 25–81, 2005.

[6] D. Bhamare, M. Samaka, A. Erbad, R. Jain, and L. Gupta, "Exploring microservices for enhancing internet QoS," *Trans. Emerg. Telecommun. Technol.*, vol. 29, no. 11, 2018, doi: 10.1002/ett.3445.

[7] A. Sundberg, "A study on load balancing within microservices architecture," 2019, [Online]. Available: https://www.mendeley.com/catalogue/a4814e2d-827e-3e18-93b5-3f91efa6d98b/?utm_source=desktop&utm_medium=1.19.4&utm_campaign=open_catalog&userDocumentId=%7B0ed39c18-97ea-4c9c-994a-2dd841333607%7D

[8] J. A. Valdivia, X. Limon, and K. Cortes-Verdin, "Quality attributes in patterns related to microservice architecture: a Systematic Literature Review," pp. 181–190, 2020, doi: 10.1109/conisoft.2019.00034.

[9] Z. Ding, S. Wang, and M. Pan, "QoS-Constrained Service Selection for Networked Microservices," *IEEE Access*, vol. 8, pp. 39285–39299, 2020, doi: 10.1109/ACCESS.2020.2974188.

[10] M. Villamizar *et al.*, "Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures," *Serv. Oriented Comput. Appl.*, vol. 11, no. 2, pp. 233–247, 2017, doi: 10.1007/s11761-017-0208-y.

[11] N. Viennot, M. Lécuyer, J. Bell, R. Geambasu, and J. Nieh, "Synapse: A microservices architecture for heterogeneous-database web applications," *Proc. 10th Eur. Conf. Comput. Syst. EuroSys 2015*, 2015, doi: 10.1145/2741948.2741975.

[12] V. Singh and S. K. Peddoju, "Container-based microservice architecture for cloud applications," *Proceeding - IEEE Int. Conf. Comput. Commun. Autom. ICCCA 2017*, vol. 2017-Janua, pp. 847–852, 2017, doi: 10.1109/CCAA.2017.8229914.

[13] J. Mathenge, "Containers vs Microservices: What's The Difference?," *Https://Www.Bmc.Com/Blogs/Containers-Vs-Microservices/*, 2021.

[14] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review," *IEEE Trans. Cloud Comput.*, vol. 7, no. 3, pp. 677–692, 2019, doi: 10.1109/TCC.2017.2702586.

[15] E. Casalicchio and V. Perciballi, "Auto-Scaling of Containers: The Impact of Relative and Absolute Metrics," *Proc. - 2017 IEEE 2nd Int. Work. Found. Appl. Self* Syst. FAS*W 2017*, pp. 207–214, 2017, doi: 10.1109/FAS-W.2017.149.

[16] M. D. Cojocaru, A. Oprescu, and A. Uta, "Attributes assessing the quality of microservices automatically decomposed from monolithic applications," *Proc. - 2019 18th Int. Symp. Parallel Distrib. Comput. ISPDC 2019*, no. June, pp. 84–93, 2019, doi: 10.1109/ISPDC.2019.00021.

[17] S. N. Srirama, M. Adhikari, and S. Paul, "Application deployment using containers with auto-scaling for microservices in cloud environment," *J. Netw. Comput. Appl.*, vol. 160, no. August 2019, 2020, doi: 10.1016/j.jnca.2020.102629.

[18] S. Jain and A. K. Saxena, "A survey of load balancing challenges in cloud environment," *Proc.*

5th Int. Conf. Syst. Model. Adv. Res. Trends, SMART 2016, pp. 291–293, 2017, doi: 10.1109/SYSMART.2016.7894537.

[19] X. Wan, X. Guan, T. Wang, G. Bai, and B. Choi, "Journal of Network and Computer Applications Application deployment using Microservice and Docker containers : Framework and optimization," vol. 119, no. December 2017, pp. 97–109, 2018.

[20] M. Lin, J. Xi, W. Bai, and J. Wu, "Ant Colony Algorithm for Multi-Objective Optimization of Container-Based Microservice Scheduling in Cloud," *IEEE Access*, vol. 7, pp. 83088–83100, 2019, doi: 10.1109/ACCESS.2019.2924414.

[21] K. Li, G. Xu, G. Zhao, Y. Dong, and D. Wang, "Cloud task scheduling based on load balancing ant colony optimization," *Proc. - 2011 6th Annu. ChinaGrid Conf. ChinaGrid 2011*, pp. 3–9, 2011, doi: 10.1109/ChinaGrid.2011.17.

[22] X. Guan, X. Wan, B. Choi, S. Song, and J. Zhu, "Application Oriented Dynamic Resource Allocation for Data Centers Using Docker Containers," vol. 1, no. c, pp. 1–4, 2016, doi: 10.1109/LCOMM.2016.2644658.

[23] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Migrating to Cloud-Native architectures using microservices: An experience report," *Commun. Comput. Inf. Sci.*, vol. 567, pp. 201–215, 2016, doi: 10.1007/978-3-319-33313-7_15.

[24] M. Bravetti, S. Giallorenzo, J. Mauro, I. Talevi, and G. Zavattaro, *Optimal and automated deployment for microservices*, vol. 11424 LNCS. Springer International Publishing, 2019. doi: 10.1007/978-3-030-16722-6_21.

[25] P. Stefanic, M. Cigale, A. Jones, and V. Stankovski, "Quality of Service Models for Microservices and Their Integration into the SWITCH IDE," *Proc. - 2017 IEEE 2nd Int. Work. Found. Appl. Self* Syst. FAS*W 2017*, no. September, pp. 215–218, 2017, doi: 10.1109/FAS-W.2017.150.

[26] B. Stevant, J. L. Pazat, and A. Blanc, "Optimizing the Performance of a Microservice-Based Application Deployed on User-Provided Devices," *Proc. - 17th Int. Symp. Parallel Distrib. Comput. ISPDC 2018*, pp. 133–140, 2018, doi: 10.1109/ISPDC2018.2018.00027.

[27] T. Tupid, "Basic Guide: Load Balancing and Auto-Scaling in Cloud Computing," 2019. https://medium.com/@tudip/basic-guide-load-balancing-and-auto-scaling-in-cloud-computing-219a5f0768a

[28] Z. Ni, C. Wei, T. Wood, and N. Choi, "A SmartNIC-based Load Balancing and Auto Scaling Framework for Middlebox Edge Server," pp. 21–27, 2022, doi: 10.1109/nfv-sdn53031.2021.9665167.

[29] I. Lera and C. Juiz, "Genetic Algorithm for Multi-Objective Optimization of Container Allocation in Cloud Architecture," 2017.

[30] D. A. Shafiq, N. Z. Jhanjhi, and A. Abdullah, "Load balancing techniques in cloud computing environment: A review," *J. King Saud Univ. - Comput. Inf. Sci.*, vol. 34, no. 7, pp. 3910–3933, 2022, doi: 10.1016/j.jksuci.2021.02.007.