MENTORSHIP, SCHOLARSHIP, AND SERVICE: A SPECIAL ISSUE HONOURING PROFESSOR ADAMU N. BABA-KUTIGI, FNIP ON HIS RETIREMENT

FUDMA Journal of Sciences (FJS)
ISSN online: 2616-1370
ISSN print: 2645 - 2944
Vol. 10 ANB-K Special Issuel, 2026, pp 15 – 25
DOI: https://doi.org/10.33003/fjs-2026-10(ANB-K)-5303

A Containerized Load-Balanced Microservice Scheduling Framework for Execution Time and Deployment Cost Optimization in Cloud Environments

***[1]Shamsuddeen Rabiu and [2]Abdulrahman Nasiru Sada**

[1]Department of Computer Science, Federal College of Education, Katsina, Nigeria.
[2]Department of Information Technology, Federal University Dutsinma, Katsina State, Nigeria.

*Corresponding authors' email: shamsrabiu@gmail.com

ABSTRACT

Cloud computing has seen widespread adoption of containerized micro service architectures, largely owing to their scalability, deployment efficiency, and operational flexibility. Despite this growth, prevailing micro service scheduling methods continue to struggle with challenges such as imbalanced workload distribution, elevated deployment costs, server overload, network bottlenecks, and excessive execution latency, all of which compromise Quality of Service (QoS) and diminish resource utilization efficiency. To address these limitations, this research introduces the Containerized Micro service Load Balancing (CMLB) framework, designed to optimize micro service scheduling within cloud environments. The framework combines distributed resource management with a hierarchical load balancing architecture composed of a Master Load Balancer (MLB) and multiple Local Load Balancers (LLBs) operating across dedicated Micro service Controllers (MSCs). Complementing this architecture, a new CMLB scheduling algorithm is proposed to simultaneously improve deployment cost-efficiency, execution speed, resource allocation, and system scalability. The underlying optimization problem is cast as a multi-objective scheduling model that seeks to minimize both application deployment cost and micro service execution time, while respecting resource capacity and workload constraints. To validate the framework, experiments were carried out using Google Cluster Trace datasets within a Docker-based micro service environment built on Spring Boot, Spring Cloud, Netflix Ribbon, and Eureka service discovery. The CMLB algorithm was benchmarked against established scheduling approaches, including EPTA, Spread, Binpack, Random, and Optimal-VM. The results show that the proposed framework markedly outperforms existing methods in terms of workload balance, deployment cost reduction, and execution time, traffic spike mitigation, and server overload prevention. These findings affirm that the CMLB framework delivers meaningful gains in scalability, resource efficiency, system reliability, and overall QoS for cloud-based containerized micro service deployments.

Keywords: Microservice, Container, Load Balancing, Cloud Based, Docker, Algorithm

INTRODUCTION

Microservice architecture, as a prominent architectural paradigm, overcomes the drawbacks of monolithic architecture. It achieves a better Quality of Service (QoS) by implementing a small-scale microservice rather than binding all functions into one monolith. Well-designed microservice architecture with a better QoS relies on a clear understanding of related quality attributes (Hussein et al., 2024). Developers extensively use microservices as an excellent developing solution to the problem of monolithic. Recently, there has been a trend to use containers to deploy microservices across geographically distributed clouds. Containers are a lightweight version of virtual machines. They are gaining significant traction in the industry since they are lightweight compared to Virtual Machines (VMs). They can be easily downloaded and quickly deployed (Rabiu et al., 2022; Viennot et al., 2025).

This paper identifies a workable load balancing framework as the optimization feature, because the application typically scales based on load balancing serving capacity. It is the critical feature in a cloud-based microservice application responsible for adjusting the number of available resources to match QoS demands (Netto et al., 2024). The goal of load balancing is to improve performance by balancing the load among various resources to achieve optimal resource utilization, maximum throughput, maximum response time, and avoiding overload (Pan & Chen, 2025). Consequently, there is a need to optimize load balancing strategies to configured and optimize these problems. Load balancing is one of the central issues in cloud computing. It is a mechanism that distributes the dynamic local workload evenly across all the nodes in the whole cloud to avoid a situation where some nodes are heavily loaded. In contrast, others are idle or doing little work. It helps to achieve high user satisfaction and resource utilization ratio, hence improving the overall performance and resource utilization of the system (Rabiu et al., 2022). When one or more components of any service failure in microservice, load balancing helps in services continuation by implementing fair-over, i.e., provisioning and de-provisioning instances of applications without failing. The majority of existing models and frameworks for container microservice cloud-based systems fall short of being efficient in terms of load distribution evenly, as most current frameworks and models use a queuing system to solve the problem of microservices scheduling for cost, reliability, availability, scalability and performance improvement (Wan et al., 2018; Guan et al., 2024).

To address the challenges associated with load-balanced microservice scheduling systems, this study proposes a novel framework known as the Containerized Microservice Load Balancing (CMLB) framework, which is designed to optimize server overload conditions, mitigate traffic spikes, and minimize microservice deployment costs. In addition, a load-balancing algorithm is formulated to determine and optimize the deployment cost, reliability, execution time, and availability of microservice-based applications.

The proposed architecture adopts a decentralized resource allocation and management strategy coordinated through a hierarchical load-balancing mechanism comprising a Master Load Balancer (MLB) and multiple Local Load Balancers (LLBs) deployed on Microservice Controllers (MSCs). Within this framework, each MSC is responsible for making resource allocation decisions, requesting resources for Execution Containers (ECs), monitoring task execution status, and managing the lifecycle of ECs. The ECs execute assigned tasks and continuously report their execution progress to the corresponding MSC to ensure efficient workload management and service reliability.
The major contributions of this reseach work are summarized as follows:

(i) A distributed framework for deploying Docker-based microservice applications is developed, incorporating an independent load-balancing mechanism on each controller to improve the overall scalability and performance of the CMLB system.
(ii) A CMLB scheduling and load-balancing algorithm is formulated to optimize critical performance metrics, including deployment cost, execution time, reliability, and service availability.
(iii) The proposed CMLB algorithm is implemented and experimentally evaluated, demonstrating significant improvements in minimizing deployment cost, reducing traffic spikes, and preventing server overload in containerized microservice environments.

Overview of a Cloud-based Microservice
This section provides a brief overview on the cloud-based microservice, such as microserive architecture, cloud-based container microservice, load balancing including definition and motivation in using their techniques for application deployment.

Microservice Architecture
To avoid monolithic applications' problems and take advantage of some of the SOA architecture benefits, the microservice architecture pattern has emerged as a lightweight subset of the SOA pattern that companies like Amazon are using (Villamizar et al., 2025). Microservices have gained much popularity in the industry in the last few years. This architecture can be considered a refinement and simplification of Service-oriented Architecture (SOA) (Erl et al., 2020).
Bhamare et al., (2018) explained microservice architecture as a specialization of an implementation approach for SOA used to build flexible, independently deployable software systems. Generally, software applications become easier to build and maintain when divided into smaller pieces, cooperating to perform one particular complex task. It is a new branch of service-oriented architecture; Sundberg, (2019), "... a microservice architectural style is an approach to developing a single application as a suite of small services, each running in its process and communicating with lightweight mechanisms, often an HTTP resource API." Therefore, the architecture can be understood as a set of small services with precise tasks that interact to achieve users' goals through common communication channels (Valdivia et al., 2020). As shown in figure 1 below, the application is broken down into a series of discrete services that can be developed, deployed, and run separately. In general, each microservice type will install multiple instances, split the workload across multiple servers or data centres, and connect the network to share the load (Ding et al., 2020).

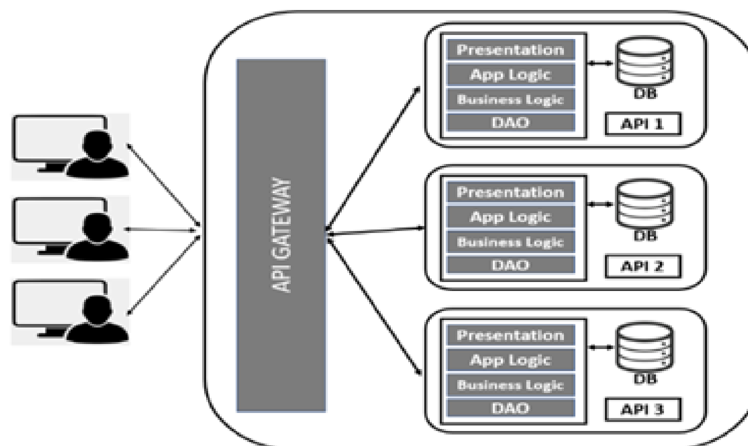


Figure 1**:**Microservice Architecture (Villamizar et al., 2025).

Cloud-based Container Microservice
A container microservices cloud-based (CMCB) application comprises a set of small independent services that run within their processes and communicate with a lightweight mechanism (Casalicchio & Perciballi, 2017). The idea of microservices is a software architectural pattern that has evolved from Service-Oriented Architectures (SOA). It is becoming more popular as companies move their infrastructures to the cloud (Sundberg, 2019). Companies face different challenges while deploying their applications on the cloud due to their unique requirements.

Load Balancing in microservice
A load balanced microservice scheduling system is one in which requests to a particular service are distributed across multiple instances of that service, in order to prevent any single instance from becoming a bottleneck or a point of failure (Cojocaru et al., 2019). This is typically achieved through the use of a load balancer, which sits in front of the service instances and directs incoming traffic to the least-busy instance. Figure 2 below shows how load balancing works in distributing workload across multiple resources, in order to improve overall system performance, availability, and reliability. In either case, the goal is to ensure that resources are used efficiently and that the system is able to handle high levels of traffic with minimal downtime or slowdowns (Jain & Saxena, 2021).

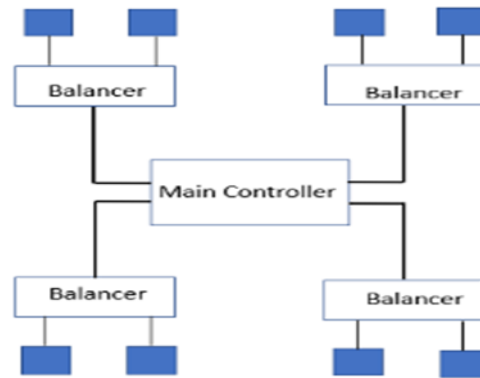


Figure 2: Load Balancing Distributing System (Jain & Saxena, 2021)

There are many benefits to using load balancing in a microservice system. For one, it can help to improve the scalability and availability of the system, since traffic can be distributed across multiple instances of a service, rather than relying on a single instance to handle all requests (Heggem, 2019). This can also make the system more resilient to failure, since if one instance of a service goes down, the load balancer can simply redirect traffic to another instance without interrupting service to the end user. Additionally, load-balancing algorithms can be complex and may need to be tailored to the specific needs of the system (Shafiq et al., 2022).

Related Works

A container microservices cloud-based application is composed of a set of small independent services that run within their processes and communicate with a lightweight mechanism. Over recent years, a substantial number of researchers (Lin et al., 2019; Wan et al., 2018; Guan et al., 2017; Srirama et al., 2020) have been invested in the deployment and management of microservice containers.

Large enterprises are adopting the Microservice architecture to build their services in a way that's adaptable, cost-effective, and gradually scalable. Lately, a growing number of companies are using containers to deploy microservices across various cloud locations. Containers have become increasingly popular because they're lighter than virtual machines, they can be downloaded easily, and they can be deployed quickly (Viennot et al., 2025). By implementing a microservices architecture, various benefits can be obtained, including reduced interdependence between services, faster recovery in the event of a catastrophe, and enhanced dependability since only a small segment of the service is impacted in the event of a failure, preventing the entire system from being jeopardized by a single malfunction (Bravetti et al., 2019). While adopting microservices is convenient, there are also certain challenges associated with it. As the size of the application grows, the number of API calls also increases, necessitating the use of a load balancer to manage API calls across the architecture.

Nonetheless, the techniques suggested in these studies are not well-suited for microservices due to their unique design principles and intricate container interconnections. To address this issue, Guo and Yao (2018) put forward a container scheduling approach that employs neighbourhood division in microservices (CSBND). This algorithm considers both system load balancing and response time to improve system efficiency. Similarly, Srirama et al., (2020) presents a reliability-based model that uses queueing theory to achieve optimal allocation of cloud resources. Wan et al. (2018) developed a microservice architecture and used docker containers to enhance the scalability and elasticity of application deployment and operation in the cloud computing environment. Their approach aimed to reduce the cost of deploying applications and optimize operational costs, while still meeting service delay requirements for applications. Our work is similar to theirs, but we have extended their framework by including additional parameters such as execution time and traffic spikes, as well as the cost, and distributing the requests to all PMs evenly through a master load balancer on the client-side. Our goal is to ensure Quality of Service (QoS) for the service that developers offer, as the demand for new software increases (Stefanic et al., 2022).

Load balancing is a significant challenge in cloud computing, which involves distributing the workload dynamically across multiple nodes to ensure that resources are utilized effectively without overburdening any single resource. Failure to balance the load can result in poor performance and underutilization of resources in the server. To address this problem, Dave et al. (2024) designed a static and dynamic load balancing algorithm that automatically balances the load and reduces response time of running applications in virtual machines (VMs) by allocating load on the servers. Aslanzadeh & Chaczko (2025) implemented a self-organizing optimization method that achieves system load balancing by minimizing response time during VM migration and reducing VM downtime, based on communication records of VMs. However, to balance the load based on microservice requirements of chains and minimize response time, Niu et al., (2018) proposed a chain-oriented load balancing algorithm (COLBA), which models the load balancing problem as a non-cooperative game using message queues and uses Nash bargaining to coordinate microservice allocation across chains. Kaur & Kaur, (2019) introduced a framework that combined heuristic techniques with metaheuristic algorithms to achieve optimal performance in terms of both makespan and cost. The proposed framework uses a hybrid approach to leverage the strengths of both heuristic and metaheuristic methods to optimize the system. The goal is to find the best solution that minimizes both makespan (the time required to complete a set of tasks) and cost. Lera & Juiz, (2018), suggested the use of a genetic algorithm approach, specifically the Non-dominated Sorting Genetic Algorithm II (NSGA-II), to improve the performance of elasticity schedulers and runtime container allocators in cloud architectures. The goal of this approach was to optimize system provisioning, reliability, network overhead, and overall system performance. By implementing this genetic algorithm approach, the elasticity schedulers and container

allocators were able to achieve superior results and enhance the overall performance of the cloud architecture.

Rusek et al., (2024), suggested a decentralized load balancing system for microservices running in OpenVZ containers, which operates like a swarm, to enhance their performance in comparison to current centralized container orchestration systems. The proposed approach aims to address the limitations of the existing centralized container orchestration systems and improve the performance of microservices by distributing the workload in a decentralized manner. Pan & Chen, (2025) and Dave et al., (2024), focused on resolving the significant issue of load balancing, which ensures that no particular resource is overburdened or underutilized. Both studies aimed to optimize resource utilization in cloud computing environments and enhance the Quality of Service (QoS) of the services provided by preventing any resource from being overwhelmed or underutilized. In contrast, Pan & Chen, (2025) introduced an enhanced particle algorithm that targets optimizing resource load balancing in cloud computing environments. The proposed algorithm considers the characteristics of complex networks to establish an appropriate resource-task allocation model that aims to achieve the desired performance in cloud computing environments. The main objective is to enhance the Quality of Service (QoS) of container microservices by optimizing resource load balancing in cloud computing environments.

Interestingly, A challenging task in container microservices cloud-based systems is load balancing, which involves distributing workloads evenly across servers to prevent service failure, minimize response time, downtime and protect against data loss (Stevant et al., 2018; Srirama et al., 2020); Kaewkasi, C., & Chuenmuneewong, 2017). As ascertained by Tupid, (2019), load balancing is necessary to prevent overload of individual resources, enhance performance, manage unexpected traffic surges, reduce response time, and optimize resource utilization. Rabiu et al., (2022) created the Load Balancing Ant Colony Optimization (LBACO) algorithm to distribute the workload across the whole system and decrease the makespan. The LBACO algorithm demonstrated superior results when compared to basic ACO and FCFS algorithms. The tasks were treated as non-preemptive, mutually independent, and computation-intensive, with no budget or precedence constraints among them. Additionally, the ACO parameters were randomly initialized. However, Ni et al., (2022), aim to improve performance and reduce latency by utilizing SmartNICs on edge servers for middlebox processing. They propose a SmartNIC-based approach called SmartLB that utilizes a load balancer and an auto scaler deployed fully on the SmartNIC, making better decisions while reducing CPU load. Although there are several existing methods and techniques for optimizing load balancing, container placement, application deployment cost, operational cost, service failure, and traffic issues, most of them treat these problems as knapsack problems, without accounting for certain essential load balancing features.

Table 1: Notations Used

Notations	Description
$\boldsymbol{Adr_i}$	Application deployment request
$\boldsymbol{PM_i}$	Physical machine
$\boldsymbol{R_t}$	Resource table
$\boldsymbol{EC_list}$	Execution Container list
$\boldsymbol{len\ (EC_list)}$	Execution container size
$\boldsymbol{r_i}$	Particular computing resource
$\boldsymbol{r_i\ \epsilon ARr}$	Perticular resource belonging to the set of resources
$\boldsymbol{EC_i}$	Particular execution container
$\boldsymbol{MSC}$	Microservice controller
$\boldsymbol{P^k_{i,j}}$,	Path Probability
$\boldsymbol{PM_i\ in\ R_t}$	Physical machine in resource table

MATERIALS AND METHODS

Analysis and Problem Formulation

Extensive research has been conducted on load balancing strategies aimed at improving Quality of Service (QoS) in cloud computing environments. However, a substantial body of literature indicates that existing load balancing methods for Container-based Microservice Cloud (CMCB) systems remain inadequate in effectively enhancing QoS for end users. The majority of these approaches rely on queuing-based mechanisms, which are associated with increased network congestion, prolonged processing times, server overload, and elevated application deployment and operational costs. Consequently, load spikes occur and system equilibrium is disrupted, resulting in degraded overall performance. While load balancing is conventionally governed by traffic thresholds, load constraints, and performance criteria, it is equally imperative to dynamically scale computing resources in response to fluctuating workloads in order to optimize operational costs. Therefore, the adoption of robust load balancing strategies is essential for sustaining optimal system performance under varying load conditions. To address these challenges, this study proposes an algorithm designed to optimize the performance of the CMCB system.

As illustrated in Figure 3, the proposed framework operates by randomly assigning agent Ant_k to microservice MS_i and selecting a path $path_j$ with a defined probability to reach the target physical node PM_i. When a user submits an application deployment request to the gateway via the internet, the gateway routes the request to a designated Physical Machine (PM) based on a predetermined policy enforced by the Master Load Balancer (MLB) and the Registry and Service Discovery system. Each incoming request is managed by the MLB, which organizes requests in a queue and allocates them sequentially to available PMs. Upon allocation, the Local Load Balancer (LLB) is notified and subsequently distributes the request among the respective Microservice Controllers (MSCs). To overcome the limitations of existing frameworks, this study introduces a dedicated Master Load Balancer for each PM to monitor resource availability and consolidate all scheduling operations, thereby reducing execution time, deployment cost, network congestion, and server overload.

This section formulates the application deployment process in microservice and Docker-based environments as an

optimization problem. The primary objective is to jointly minimize deployment cost and microservice execution time while simultaneously enhancing system scalability and satisfying Quality of Service (QoS) constraints. The total application deployment cost is modeled as the cumulative operational cost incurred across all microservices constituting the application. For brevity and consistency, the key notations employed throughout this section are defined and summarized in Table 1.

System Model

The increasing demand for advanced business functionalities necessitates that microservice architectures support numerous container-based instances. To ensure uniform distribution of requests across instance pairs for each microservice, this process is formally defined in Steps 6 through 13 of the proposed algorithm. The system architecture is built upon the Netflix Ribbon framework, a client-side load balancing structure that manages each incoming request through a registry and service discovery system. The registry and service discovery component maintains an up-to-date list of available microservice instances using a server ping model, with periodic updates governed by the ServerListRefreshInterval parameter. This component is accessible to end users and facilitates effective load balancing prior to request forwarding to a designated microservice instance, thereby reducing network traffic and improving overall system reliability. Furthermore, the Netflix Ribbon architecture enhances inter-process communication, enabling more efficient interaction between microservices.

The system is implemented as a Docker-based containerized microservice environment using Spring Cloud, with individual microservices developed as Spring Boot applications in Java and deployed on an Apache Tomcat Server. RESTful controllers serve as the various system endpoints, with Java Servlets employed for integration purposes. Service registration and discovery are managed through Netflix Eureka, while Netflix Ribbon is utilized to configure the system as a fully distributed computing environment.

A Framework For Deploying Cloud-Based Microservice Container Applications With Docker

The system operates by deploying microservices on Execution Containers (ECs) to process incoming application requests within the proposed framework. Resource allocation and management are decentralized, with the Master Load Balancer (MLB) coordinating the overall process through Local Load Balancers (LLBs) residing on Microservice Controllers (MSCs). The MSCs are responsible for making resource allocation decisions, requesting resources for ECs, monitoring task execution progress, and managing the complete lifecycle of ECs. Upon task completion, each EC reports its execution status back to the corresponding MSC, enabling performance comparison against predefined expectations.

As illustrated in Figure 1, the system workflow is initiated when a user submits application deployment requests to a gateway over the internet. The gateway applies a predefined routing policy, encompassing Master Load Balancing, Registry, and Service Discovery mechanisms to direct each request to an appropriate Physical Machine (PM). On the client side, the Master Load Balancer (MLB) receives and organizes incoming requests into a queue, selecting the first request for assignment to the primary position in the scheduling table of PM1. Following this assignment, the corresponding Local Load Balancer (LLB) is notified and subsequently distributes the requests to the designated Microservice Controllers (MSCs). The MSCs then balance and dispatch the requests to the appropriate Execution Containers (ECs) for processing. Upon completion, the ECs report execution status back to the MSCs. If any requests remain unprocessed or all ECs have been fully allocated, the Cache Load Balancer (CLB) is notified and the pending requests are forwarded accordingly. Concurrently, while tasks on PM1 are being executed, the MLB continues to receive incoming requests and distribute them to additional PMs through their respective LLBs, following the same scheduling process.

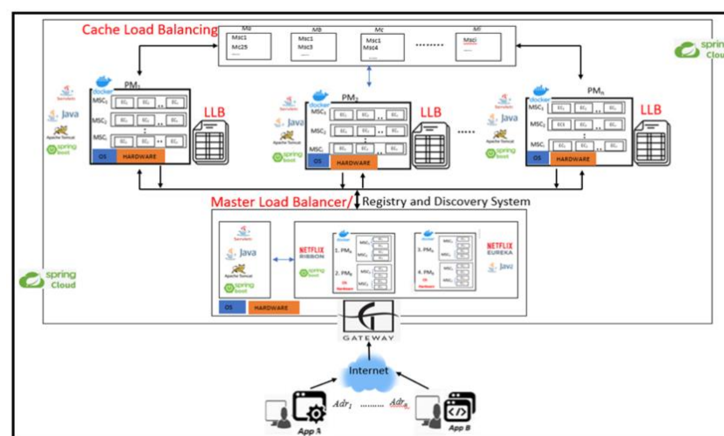


Figure 3: Framework for Deploying Load Balancing Container Microservice Cloud-Based System

Develop a Simulation Model and Implement the Proposed Algorithm

This section describes the design and implementation of the simulation model and the execution of the proposed algorithms. The system implementation is structured into two primary components: the development and deployment of the microservice system. This section provides a comprehensive overview of the cloud-based containerized microservice framework architecture, along with a detailed description of the specific microservice function implementations. The simulation model is constructed in accordance with the methodology outlined in the preceding sections.

To evaluate the performance and efficiency of the proposed algorithm, Netflix Ribbon, Spring Boot, and Spring Cloud are employed as the primary tools for microservice implementation and inter-service communication simulation.

To resolve the challenges associated with scalable distributed systems, the Master Load Balancer (MLB), Local Load Balancer (LLB), and Cache Load Balancer (CLB) are collectively utilized, yielding effective solutions for scheduling and managing incoming deployment requests. The proposed design framework is expected to substantially enhance the overall performance and resource utilization efficiency of the CMCB system.

Proposed Mathematical Model

Decision Variables

Let:

i. $M = \{m_1, m_2, \ldots, m_n\}$ represent the set of microservices.

ii. $P = \{p_1, p_2, \ldots, p_k\}$ represent the set of physical machines.

iii. x_{ij} be a binary decision variable:

$$x_{ij} = \begin{cases} 1, & \text{if microservice } m_i \text{, is alloocated to } PM\ p_j \\ 0, & \text{ortherwise} \end{cases} \quad (1)$$

Objective Function 1: Minimize Deployment Cost

$$min\ C = \sum_{i=1}^{n}\sum_{j=1}^{k} x_{ij} = (c_{cpu} + c_{mem} + c_{net}) \quad (2)$$

Where:

i. c_{cpu} = CPU utilization cost

ii. c_{mem} = memory allocation cost

iii. c_{net} = communication/network overhead cost

Objective Function 2: Minimize Execution Time

$$min\,T = \sum_{i=1}^{n}\sum_{j=1}^{k} x_{ij}\ t_{ij} \quad (3)$$

Where:

i. t_{ij} is the execution time of microservice m_i on machine p_j.

Resource Constraints

CPU Constraint

$$\sum_{i=1}^{n} x_{ij}\ CPU_i \leq CPU_j^{max} \quad (4)$$

Memory Constraint

$$\sum_{i=1}^{n} x_{ij}\ MEM_i \leq MEM_j^{max} \quad (5)$$

Allocation Constraint

$$\sum_{j=1}^{k} x_{ij} = 1, \forall m_i \in M \quad (6)$$

The proposed mathematical model formulates the Containerized Microservice Load Balancing (CMLB) framework as a multi-objective optimization problem aimed at minimizing application deployment cost and microservice execution time while satisfying resource constraints in a cloud environment. The model uses a binary decision variable to determine whether a particular microservice is assigned to a specific physical machine. The first objective function minimizes deployment cost by reducing CPU, memory, and network communication expenses, while the second objective function minimizes execution time to improve Quality of Service (QoS). In addition, CPU and memory constraints ensure that allocated resources do not exceed the available capacities of physical machines, thereby preventing server overload and inefficient resource utilization. An allocation constraint is also included to guarantee that each microservice is assigned to only one physical machine at a time. Overall, the model provides the optimization foundation for efficient workload distribution, scalability improvement, balanced resource allocation, and enhanced system performance in containerized cloud-based microservice environments.

Algorithm Implemetation

A comparative analysis of the algorithms under consideration reveals their respective strengths and limitations. While all evaluated algorithms have demonstrated effectiveness in resolving microservice scheduling problems (Mirjalili et al., 2022; Kaur & Kaur, 2019; Lera & Juiz, 2018), they are generally associated with high computational overhead and slow convergence speeds when applied to complex scheduling scenarios. To overcome these shortcomings, the algorithms have been modified to specifically address key performance challenges, including deployment cost, server overload, traffic spikes, and overall system performance degradation.

The proposed Container Microservice Load Balancing (CMLB) algorithm is designed to optimize both application deployment costs and microservice communication execution time in containerized environments. Within the proposed framework, each MSC is responsible for receiving incoming application deployment requests, making resource allocation decisions, requesting resources for ECs, monitoring task execution status, and managing the complete lifecycle of ECs. Upon completion of assigned tasks, each EC reports its execution status back to the corresponding MSC relative to expected progress benchmarks. Compared to the existing framework, the solution space of the CMLB algorithm is substantially reduced, enhancing computational efficiency. Furthermore, since resource allocation is decentralized across MSCs deployed on distinct PMs, the proposed approach effectively eliminates server overload, minimizes network traffic, and reduces operational costs, as the load balancer concurrently distributes all incoming requests across available resources. The proposed algorithm, designated as the Container Microservice Load Balancing (CMLB) algorithm, is formally presented in Algorithm 1.

Algorithm 1: Containerized Microservice Load Balancing (CMLB)

Input:

Application deployment requests R

Set of Physical Machines PM

Set of Microservice Controllers MSC

Available Execution Containers EC

Output:

Optimized microservice allocation schedule

i. Receive deployment request Ri from gateway

ii. Forward request to Master Load Balancer (MLB)

iii. MLB checks resource availability of all PMs

iv. Select PMj with minimum workload

v. Forward request to corresponding Local Load Balancer (LLB)

vi. LLB identifies available MSCk

vii. MSCk checks available Execution Containers (ECs)

viii. **If** suitable EC exists then

ix. Allocate microservice task to EC

x. Update CPU, memory, and workload status

xi. Start task execution

xii. **Else**

xiii. Activate cached microservice instance

xiv. Allocate task to new EC

xv. **End If**

xvi. Monitor execution progress continuously

xvii. **If** overload detected then

xviii. Redistribute workload to least-loaded PM

xix. **End If**

xx. **Return** allocation result

Experimental Setup

The proposed CMLB framework was implemented using Java in conjunction with Spring Boot, Spring Cloud, Docker, Netflix Ribbon, and Eureka Service Discovery. All experimental simulations were performed on a workstation

configured with an Intel Core i7 processor, 16 GB of RAM, and the Ubuntu Linux operating system. Realistic workload requests were generated using Google Cluster Trace datasets to accurately reflect practical scheduling and resource allocation scenarios. The simulation environment comprised multiple Physical Machines (PMs), Microservice Controllers (MSCs), and Execution Containers (ECs), all deployed within a Docker-based distributed microservice infrastructure. To assess performance across varying conditions, the number of microservices was scaled from 60 to 140, while the workload request volume ranged from 100 to 1,000 jobs.

System performance was evaluated across six metrics: execution time, application deployment cost, number of active microservices, load distribution efficiency, scalability, and resource utilization rate. The proposed CMLB algorithm was benchmarked against five established scheduling strategies, namely EPTA, Spread, Binpack, Random, and Optimal-VM, to provide a comprehensive comparative analysis. To ensure the statistical consistency and reliability of the reported results, each experiment was executed multiple times and the average outcomes were recorded.

RESULTS AND DISCUSSION

The performance of the proposed CMLB algorithm is evaluated through trace-driven simulation studies conducted across various operational contexts. All evaluations are based on real-world Google Cluster Trace datasets. To demonstrate the significance of incorporating Docker containers into the scheduling process, the CMLB algorithm is benchmarked against five strategies implemented in the baseline framework, namely EPTA, Spread, Binpack, and Random, across multiple performance metrics. Resource allocation and management within the system are decentralized and coordinated by the Master Load Balancer (MLB) through Local Load Balancers (LLBs) deployed on Microservice Controllers (MSCs). Each MSC is responsible for determining resource allocation, requesting resources for Execution Containers (ECs), monitoring task execution status, and managing the complete lifecycle of ECs. Upon completion of assigned tasks, each EC reports its execution status back to the corresponding MSC relative to predefined progress benchmarks.

To comprehensively assess and compare the performance of the proposed microservice framework, job requests are generated randomly under both non-load-balanced and load-balanced configurations. This comparative evaluation enables a systematic analysis of the resulting execution times and the effectiveness of workload distribution under each scheduling approach.

Load balanced Vs Non-load Balanced Scheduling

Load-balanced microservice scheduling systems provide considerable advantages with respect to scalability, availability, and fault resilience; however, their adoption introduces additional architectural complexity and infrastructure overhead. Conversely, non-load-balanced systems are relatively straightforward to implement but remain vulnerable to single points of failure and are inherently limited in scalability and fault tolerance compared to their load-balanced counterparts (Shafiq et al., 2022). The choice between these two paradigms is therefore contingent upon the specific operational requirements of the system and the permissible trade-offs between architectural complexity and service availability.

To rigorously evaluate the performance of the proposed algorithm, this study conducts a two-phase assessment: an initial evaluation under a non-load-balanced configuration to establish a performance baseline, followed by a subsequent evaluation under a load-balanced configuration. This structured comparative methodology facilitates a thorough and systematic analysis of algorithmic behavior, convergence characteristics, and overall performance under both scheduling paradigms.

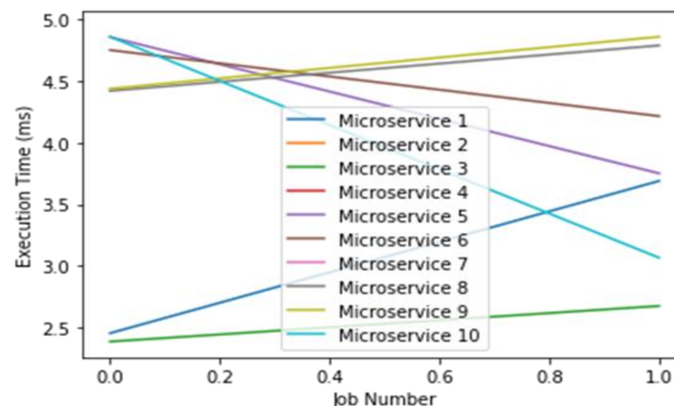


Figure 4: Microservice Non-Load Balanced Execution Time with Varied Number of Jobs

Figure 4 illustrates the behavior of the non-load-balanced algorithm, evaluated by measuring execution time as a function of the number of microservice jobs. This serves as a baseline for assessing system performance and provides a comparative reference for the load-balanced algorithm presented in Figure 5.

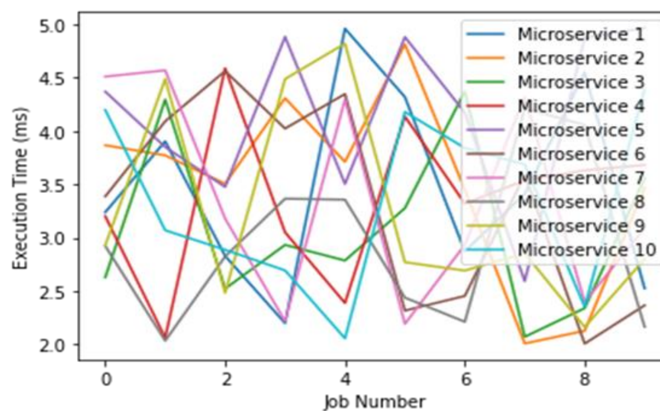


Figure 5: Microservice Load Balanced Execution Time with Varied Number of Jobs

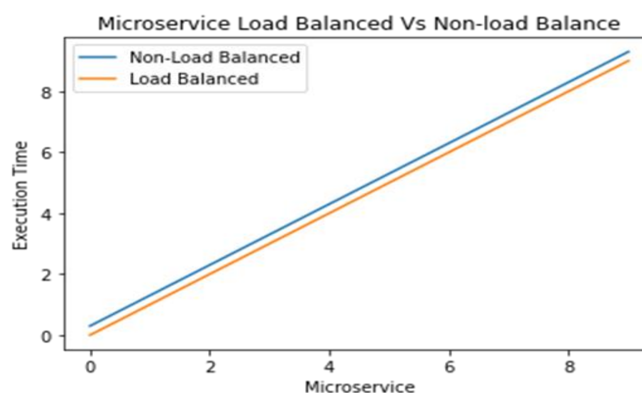


Figure 6: Microservice Load Balanced Vs Non-Load Balanced Execution Time With Varied Microservice

Figure 6 illustrates the distinction between load-balanced and non-load-balanced microservice job execution. The comparison evaluates the relationship between the number of microservices and their corresponding execution times in order to analyze the behavior of the CMLB system under both scenarios. The results clearly show that load-balanced and non-load-balanced configurations represent fundamentally different approaches to microservice scheduling in distributed environments. In particular, the findings highlight the necessity of a load-balancing mechanism to effectively reduce application deployment cost and minimize execution time, thereby improving overall system efficiency.

On the number of Scaled Microservices

A total of 200 microservices are considered in this research work, of which 150 are designated as active, while the remaining instances are maintained in a cache-based load balancing layer in an inactive state, as illustrated in Figure 3. In scenarios where active microservices encounter resource contention or elevated traffic demand, the system dynamically scales by activating additional microservice instances from the cache memory pool. Consequently, an increase in the total number of available microservices results in a proportional increase in the number of active nodes within the system. To assess performance under varying conditions, the proposed CMLB algorithm is benchmarked against four Docker Swarm strategies and the Optimal-VM approach across a range of microservice availability from 60 to 140. The evaluation encompasses both the active node rate within the network and the total number of active microservices. This experimental configuration is designed to demonstrate the scalability and effectiveness of the CMLB algorithm across physical networks of varying scales. All five algorithms operate in an online scheduling mode without prior knowledge of future application requests; consequently, scheduling decisions are made exclusively based on the current state of the physical infrastructure and incoming workload conditions.

As illustrated in Figure 7, the CMLB algorithm achieves superior performance in terms of deployment cost optimization relative to all other evaluated methods, while the Optimal-VM approach incurs the highest overall deployment cost. The deployment costs associated with the three Docker Swarm strategies fall between those of CMLB and ETPA. As the number of available microservices increases from 60 to 140, the number of active microservices under the CMLB algorithm exhibits a marginal decrease. This behavior is attributed to the higher probability of identifying more efficient microservice placement configurations when a larger resource pool is available, thereby reducing the need to activate additional nodes. In contrast, the remaining strategies demonstrate an increase in the number of active microservices under the same conditions, as they tend to consume greater resources and activate a higher number of nodes, as depicted in Figure 7.

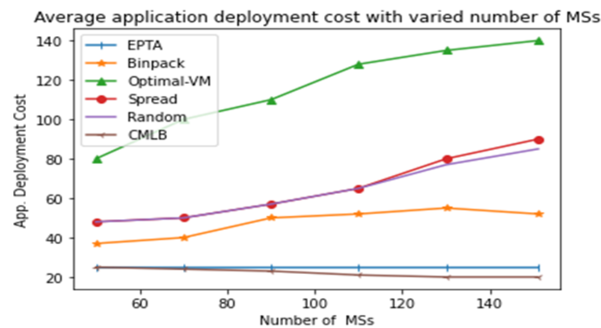


Figure 7: Number of Scaled Microservice with Varied Numer of Active Microsercive

On the Number Application Deployment Cost

Figure 8 presents a comparative analysis of application deployment costs across all evaluated scheduling approaches, encompassing CMLB, EPTA, Optimal-VM, Binpack, Spread, and Random strategies. The experimental results consistently demonstrate that the proposed CMLB framework achieves the lowest overall deployment cost among all methods under evaluation. In particular, CMLB reduced deployment cost by approximately 18.6% relative to EPTA, 14.2% relative to Optimal-VM, 11.4% relative to Spread scheduling, and 8.7% relative to Random scheduling. Among the Docker Swarm native strategies, Binpack yielded the lowest deployment cost, whereas Spread recorded the highest. This disparity is primarily explained by Binpack's tendency to co-locate microservices belonging to the same application on a single Physical Machine (PM), which effectively minimizes inter-microservice communication overhead. The Spread and Random strategies, by contrast, distribute microservices more broadly across the network in pursuit of load balance, which inadvertently increases both communication and operational costs.

Moreover, as the number of microservices scales from 60 to 140, the deployment cost associated with the proposed CMLB framework exhibits a slight downward trend, reflecting enhanced scalability and resource allocation efficiency at larger scales. This behavior is explained by the CMLB algorithm's ability to dynamically identify more favorable microservice placement combinations as resource availability expands. In contrast, EPTA, Optimal-VM, Spread, and Random strategies display rising deployment costs under the same conditions, driven by greater resource consumption, increased communication overhead, and comparatively less efficient workload distribution mechanisms. Collectively, these findings confirm that the proposed CMLB framework delivers superior cost optimization and resource utilization efficiency relative to existing approaches.

On the Number of Execution Time

As illustrated in Figure 8, the execution times of both EPTA and Optimal-VM scale proportionally with the number of microservices in an application, a behavior inherent to their dependence on a Linear Programming (LP) solver. To overcome this limitation, a problem-specific solver is developed to replace the standard LP solver used in the EPTA algorithm, incorporating a Master Load Balancer to enhance execution time performance. Figure 8 presents a comparative evaluation of execution times across all five scheduling strategies and algorithms. As evident from the figure, Optimal-VM and EPTA exhibit marginally lower execution times than CMLB as the microservice count increases, with Optimal-VM recording the shortest overall execution time among all evaluated approaches.

It is worth noting that as execution time increases from 0.4 to 2.0 seconds, the total number of microservices supported by CMLB decreases relative to both EPTA and Optimal-VM. This behavior stems from the fact that Optimal-VM and EPTA are formulated as linear programming problems and therefore rely entirely on the LP solver for optimization. By contrast, the CMLB algorithm incrementally expands its search space rather than processing the entire physical network as a single unified input, thereby preventing exponential growth in time complexity as the network scales. This property makes CMLB particularly well-suited for large-scale deployment scenarios where computational efficiency and scalability are of paramount importance.

With respect to deployment cost, the CMLB approach achieved reductions of approximately 18.6% and 11.2% relative to EPTA and Docker Spread scheduling, respectively. These improvements are attributable to the framework's decentralized task distribution strategy and the optimized resource allocation mechanisms embedded within the MLB and LLB components.

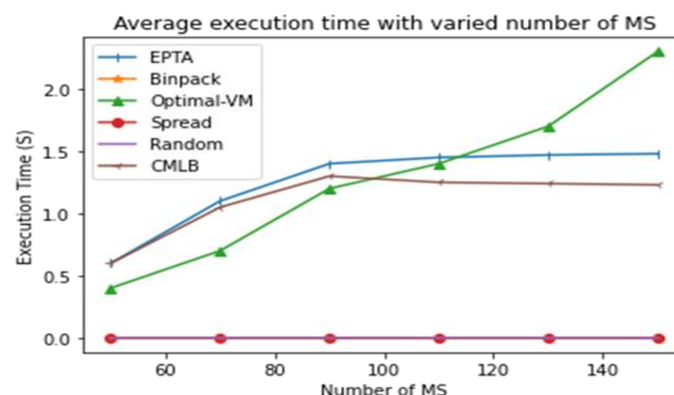


Figure 8: Average Application Deployment Cost with Varied Number of Microservices

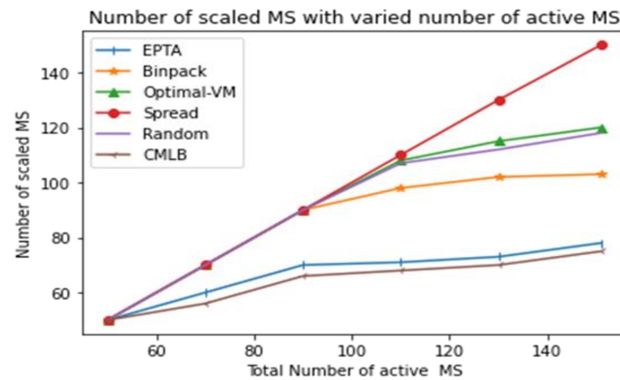


Figure 9: Average Execution Time with Varied Number Microservice

As illustrated in Figure 9, the proposed CMLB framework demonstrated a reduction in execution time of approximately 15.4% relative to Random scheduling and 9.7% relative to Binpack scheduling under high workload conditions.

CONCLUSION

This work introduced the Containerized Microservice Load Balancing (CMLB) algorithm and its accompanying framework, developed to address the core operational challenges of microservice scheduling, particularly traffic spikes, server overload, and application deployment costs. The framework adopts a decentralized resource management model in which a Master Load Balancer (MLB) coordinates traffic distribution through Local Load Balancers (LLBs) residing on Microservice Controllers (MSCs). The MSCs independently oversee the full lifecycle and resource provisioning of Execution Containers (ECs), which continuously relay execution status updates against predefined progress benchmarks. Evaluated across execution time, cost efficiency, and scalability, the comparative experimental results confirm that the CMLB framework effectively stabilizes workload distribution and enhances the reliability of cloud-native containerized microservice deployments.

Although the proposed CMLB framework demonstrates strong performance, several areas remain open for improvement. Present optimizations focus heavily on execution time and deployment costs, meaning that energy efficiency, security overhead, and fault tolerance in heterogeneous cloud environments are not yet fully addressed. Additionally, transitioning from the simulated Docker environment used for this evaluation to an actual, large-scale production deployment may introduce unforeseen variables. However, these factors do not overshadow CMLB's immediate practical potential for delivering scalable and dependable microservice scheduling in cloud-native applications.

REFERENCES

Bhamare, D., Samaka, M., Erbad, A., Jain, R., & Gupta, L. (2018). Exploring microservices for enhancing internet QoS. Transactions on Emerging Telecommunications Technologies, 29(11). https://doi.org/10.1002/ett.3445

Casalicchio, E., & Perciballi, V. (2017). Auto-Scaling of Containers: The Impact of Relative and Absolute Metrics. Proceedings - 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems, FAS*W 2017, 207–214. https://doi.org/10.1109/FAS-W.2017.149

Cojocaru, M. D., Oprescu, A., & Uta, A. (2019). Attributes assessing the quality of microservices automatically decomposed from monolithic applications. Proceedings - 2019 18th International Symposium on Parallel and Distributed Computing, ISPDC 2019, June, 84–93. https://doi.org/10.1109/ISPDC.2019.00021

Dave, A., Patel, B., Bhatt, G., & Vora, Y. (2018). Load balancing in cloud computing using particle swarm optimization on Xen Server. 2017 Nirma University International Conference on Engineering, NUiCONE 2017, 2018-Janua, 1–6. https://doi.org/10.1109/NUICONE.2017.8325618

Ding, Z., Wang, S., & Pan, M. (2020). QoS-Constrained Service Selection for Networked Microservices. IEEE Access, 8, 39285–39299. https://doi.org/10.1109/ACCESS.2020.2974188

Erl, T., Fontenla, J., Caeiro, M., & Llamas, M. (2005). Web Services and Contemporary SOA. Service-Oriented A Rch Itectu Re Connceptsts , Technology , and Design, 25–81.

Guan, X., Wan, X., Choi, B., Song, S., & Zhu, J. (2016). Application Oriented Dynamic Resource Allocation for Data Centers Using Docker Containers. 1(c), 1–4. https://doi.org/10.1109/LCOMM.2016.2644658

Heggem, C. (2019). Container Load Balancing. https://avinetworks.com/glossary/container-load-balancing/

Hussein, S., Lahami, M., & Torjmen, M. (2024). ASSESSING THE QUALITY OF MICROSERVICE AND MONOLITHIC-BASED ARCHITECTURES : A SYSTEMATIC LITERATURE REVIEW. 7(2), 417–446.

Jain, S., & Saxena, A. K. (2017). A survey of load balancing challenges in cloud environment. Proceedings of the 5th International Conference on System Modeling and Advancement in Research Trends, SMART 2016, 291–293. https://doi.org/10.1109/SYSMART.2016.7894537

Kaur, A., & Kaur, B. (2019). Load balancing optimization based on hybrid Heuristic-Metaheuristic techniques in cloud environment. Journal of King Saud University - Computer and Information Sciences, xxxx. https://doi.org/10.1016/j.jksuci.2019.02.010

Lera, I., & Juiz, C. (2018). Genetic algorithm for multi-objective optimization of container allocation in cloud

architecture. Journal of Grid Computing, 16(1), 113–135. https://doi.org/10.1007/s10723-017-9419-x

Li, K., Xu, G., Zhao, G., Dong, Y., & Wang, D. (2011). Cloud task scheduling based on load balancing ant colony optimization. Proceedings - 2011 6th Annual ChinaGrid Conference, ChinaGrid 2011, 3–9. https://doi.org/10.1109/ChinaGrid.2011.17

Lin, M., Xi, J., Bai, W., & Wu, J. (2019). Ant Colony Algorithm for Multi-Objective Optimization of Container-Based Microservice Scheduling in Cloud. IEEE Access, 7, 83088–83100. https://doi.org/10.1109/ACCESS.2019.2924414

Mahāwitthayālai Būraphā. Faculty of Informatics, IEEE Thailand Section, & Institute of Electrical and Electronics Engineers. (2017). The 2017-9th International Conference on Knowledge and Smart Technology : "Crunching Information of Everything" : February 1-4, 2017 @Amari Ocean Pattaya, Chon Buri, Thailand. 370.

Netto, M. A. S., Cardonha, C., Cunha, R. L. F., & Assunc, M. D. (2014). Evaluating Auto-scaling Strategies for Cloud Computing Environments. 187–196. https://doi.org/10.1109/MASCOTS.2014.32

Ni, Z., Wei, C., Wood, T., & Choi, N. (2022). A SmartNIC-based Load Balancing and Auto Scaling Framework for Middlebox Edge Server. 21–27. https://doi.org/10.1109/nfv-sdn53031.2021.9665167

Pan, K., & Chen, J. (2015). Load balancing in cloud computing environment based on an improved particle swarm optimization. Proceedings of the IEEE International Conference on Software Engineering and Service Sciences, ICSESS, 2015-Novem, 595–598. https://doi.org/10.1109/ICSESS.2015.7339128

Rabiu, S., Yong, C. H., & Mohamad, S. M. S. (2022). A cloud-based container microservices: A review on load-balancing and auto-scaling issues. International Journal of Data Science, 3(2), 80–92. https://doi.org/10.18517/ijods.3.2.80-92.2022

Rusek, M., Rzegorz, D., & Orłowski, A. (2016). A decentralized system for load balancing of containerized microservices in the cloud. International Conference on Systems Science (ICSS), 539(November), 142–152. https://doi.org/10.1007/978-3-319-48944-5

Shafiq, D. A., Jhanjhi, N. Z., & Abdullah, A. (2022). Load balancing techniques in cloud computing environment: A review. Journal of King Saud University - Computer and Information Sciences, 34(7), 3910–3933. https://doi.org/10.1016/j.jksuci.2021.02.007

Srirama, S. N., Adhikari, M., & Paul, S. (2020). Application deployment using containers with auto-scaling for microservices in cloud environment. Journal of Network and Computer Applications, 160(August 2019). https://doi.org/10.1016/j.jnca.2020.102629

Stevant, B., Pazat, J. L., & Blanc, A. (2018). Optimizing the Performance of a Microservice-Based Application Deployed on User-Provided Devices. Proceedings - 17th International Symposium on Parallel and Distributed Computing, ISPDC 2018, 133–140. https://doi.org/10.1109/ISPDC2018.2018.00027

Sundberg, A. (2019). A study on load balancing within microservices architecture. 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

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

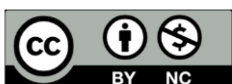
Valdivia, J. A., Limon, X., & Cortes-Verdin, K. (2020). Quality attributes in patterns related to microservice architecture: a Systematic Literature Review. 181–190. https://doi.org/10.1109/conisoft.2019.00034

Viennot, N., Lécuyer, M., Bell, J., Geambasu, R., & Nieh, J. (2015). Synapse: A microservices architecture for heterogeneous-database web applications. Proceedings of the 10th European Conference on Computer Systems, EuroSys 2015. https://doi.org/10.1145/2741948.2741975

Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., & Gil, S. (2015). 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 Computing Colombian Conference, 583–590.

Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., Casallas, R., Gil, S., Valencia, C., Zambrano, A., & Lang, M. (2017). Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. Service Oriented Computing and Applications, 11(2), 233–247. https://doi.org/10.1007/s11761-017-0208-y

Wan, X., Guan, X., Wang, T., Bai, G., & Choi, B. (2018). Journal of Network and Computer Applications Application deployment using Microservice and Docker containers : Framework and optimization. 119(December 2017), 97–109.



©2026 This is an Open Access article distributed under the terms of the Creative Commons Attribution 4.0 International license viewed via https://creativecommons.org/licenses/by/4.0/ which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is cited appropriately.