IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 14 February 2024, accepted 27 February 2024, date of publication 4 March 2024, date of current version 11 March 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3373316

RESEARCH ARTICLE

Optimizing Cloud Performance: A Microservice Scheduling Strategy for Enhanced Fault-Tolerance, Reduced Network Traffic, and Lower Latency

ABDULLAH ALELYANI, AMITAVA DATTA, AND GHULAM MUBASHAR HASSAN, (Senior Member, IEEE)
Department of Computer Science and Software Engineering, The University of Western Australia, Perth, WA 6009, Australia
Corresponding author: Abdullah Alelyani (abdullah.alelyani@research.uwa.edu.au)

This work was supported by the University of Western Australia.

ABSTRACT The emergence of microservice architecture has brought significant advancements in software development, offering improved scalability and availability of applications. Cloud computing benefits from microservice architecture by mitigating the risks of single failures and ensuring compliance with service-level agreements. However, using microservice architecture presents two challenges: 1) managing network traffic, which leads to latency and network congestion; and 2) inefficient resource allocation for microservices. Current approaches have limitations in addressing these challenges. To overcome these limitations, we propose a novel scheduling strategy that schedules microservice replicas using a modified particle swarm optimization algorithm to place them on the most suitable physical machine. Additionally, we balance the load across physical machines in the cluster using a simple round-robin algorithm. Furthermore, our scheduling strategy integrates with Kubernetes to tackle resource allocation and deployment challenges. The proposed strategy has been evaluated by simulating two scenarios using Alibaba and Google datasets. The experimental results demonstrate the effectiveness of our strategy in reducing traffic, balancing load, and utilizing CPU and memory efficiently.

INDEX TERMS Microservice, network traffic cost, PSO, resource utilization, Alibaba, VM, container, scheduling strategy.

I. INTRODUCTION

Cloud computing has emerged as a cornerstone of the global economy. Cloud providers such as Google Cloud, Amazon ECS, and IBM Cloud are expanding worldwide. These platforms offer users access to on-demand services anytime and from anywhere. These cloud providers are responsible for ensuring the quality of services (QoS) for their customers, which include factors like availability, throughput, and reliability [1].

Virtualization technology has played a crucial role in addressing the service demands of cloud computing. It has demonstrated efficient resource management by allowing resource sharing between jobs arriving in the cloud. It is a tool for managing multi-tenant services to run on shared physical machines (PMs) [2]. It increases resource utilization, which reflects on the cost of the services [3]. Moreover, it ensures service isolation, enhances security, and meets user expectations. Hypervisors and containers are the two most common types of virtualization [2]. Hypervisor-based virtualization isolates each virtual machine (VM) with a limited number of resources from the host and its operating system. Container-based virtualization, on the other hand, allows containers to share all of the host's resources [2], [4].

VMs allow resources to be allocated, utilized, and shared by applications deployed in the cloud. In terms of

The associate editor coordinating the review of this manuscript and approving it for publication was Yunlong Cai.

building services (applications) on the cloud, there are two distinct methodologies: monoliths and microservices. The monolithic approach is to build all application tasks at once. However, the microservices approach involves constructing the applications as small dependent tasks that can function independently [5]. Recently, microservices have gained the attention of most businesses because of their simplified debugging, deployment, maintenance, scalability, and high-performance output. Studies such as [1] and [6] explore the role of microservices architecture in cloud computing, focusing on scalability and performance bottlenecks. They also highlight potential challenges, including organizational transformation, service decomposition, distributed monitoring, and bug localization.

The architecture of microservices plays a crucial role in increasing the availability ratio of the services using the replica method [7], [8]. Multiple replicas of each service are deployed across the cloud resources [7]. Moreover, it also improves service resilience in the cloud environment by preventing the failure of a single service from causing the entire system to fail [7]. However, building applications as tiny services that are dependent on each other requires frequent communication between them. This communication between microservices can occur either through local I/O or the network. Various communication approaches, including HTTP protocol, APIs [1], RPC, and RESTful APIs, are utilized to manage the high volume of network traffic between microservices [1].

Another critical aspect of microservices is scalability. Scaling and auto-scaling microservices are the two techniques that are widely used in data centers to enhance the availability of services. Research studies, such as [9], [10], and [11], attempt to address the issues of service unavailability due to the high volume of requests using different scaling techniques. Similarly, studies like [12] and [13] utilize scaling techniques to reduce end-to-end latency.

The communication between microservices and their scalability increases network overhead. Therefore, effective network traffic management becomes a critical aspect of microservice architecture. In this paper, we propose a novel scheduling strategy that reduces network traffic by using a scalable approach to deploy replicas of microservices near their neighboring services. Our scheduling strategy, based on modified Swarm Particle Optimization (SPO), replicates microservices near their dependencies. Our approach aims to minimize network traffic, latency, and the risk of single points of failure. In addition, it aims to enhance load balancing, improve application performance, and increase availability. The key contributions of this article are:

1) Proposing a novel scheduler that reduces the usage of network traffic using scaling microservice replicas. The proposed scheduler identifies optimal solutions for selecting new hosts (PMs) for microservice replicas. The selection process is driven not just by minimizing network traffic costs but also by maximizing system resilience. The proposed scheduler aims to identify PMs with the best fitness for replicating microservices.
2) To optimize load balancing, we propose a modified round-robin (RR) algorithm to significantly enhance cloud performance and reduce latency and resource contentions.
3) Proposing a plug-in tool that seamlessly integrates with Kubernetes. It enables the efficient initiation of microservice replications and the intelligent routing of network traffic towards the nearest microservices and their dependent components. We use call graph representation to analyze microservices architecture, which enriches our research by characterizing microservices, including microservices topology, different types of microservices, various communication paradigms, and the quality of deploying microservices.

The current research on scheduling microservices aims to tackle issues such as execution time, cost, and resource allocation. However, issues such as latency reduction and fault tolerance that affect reliability are important, as described in Section II. We identify this gap and propose a novel approach that minimizes network latency and enhances fault tolerance and load balancing. Our research aims to address these reliability challenges and improve overall system availability and performance by replicating microservices close to their dependents with the help of the Kubernetes plugin.

The rest of the paper is organized as follows: We introduce the background of the technologies of microservices in Section II. The discussion of the related work is presented in Section III. We present our proposed scheduling strategy in Section IV. The problem formulation and assumptions are explained in Section V. The experimental setup and metrics used for evaluating the proposed strategy are presented in Section VI. The comparison between our proposed strategy and the baseline strategies is conducted in Section VII. The experimental results are discussed in Section VIII. Finally, we conclude our study in Section IX.

II. BACKGROUND

In this section, we briefly introduce the microservices architecture, Kubernetes, and Docker Swarm. Our focus here is to explain their container deployment and configuration.

A. MICROSERVICES ARCHITECTURE

In a monolithic architecture, updates or modifications to one component can have a significant impact on other components. It requires redeveloping and redeploying the applications. In addition, as the application is deployed as a single unit into a single container, this limits the flexibility of scaling resources. Furthermore, any increase in the size and complexity of the application increases the challenge of meeting the requirements of resources [14].

On the other hand, a microservices architecture builds and deploys applications as lightweight and loosely coupled components. These components run in containers.

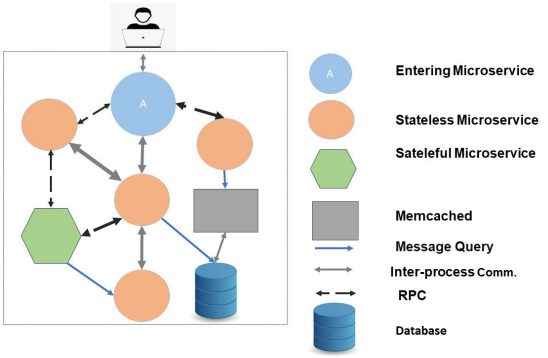


FIGURE 1. Call graph depicting the relationships between different types of microservices. This graph also highlights the various types of microservices and the types of connections between them, providing a visual representation of the system architecture.

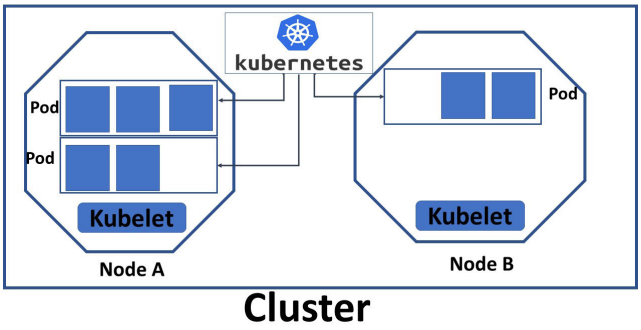


FIGURE 2. Cluster architecture created by Kubernetes. Nodes A and B represent two separate worker nodes in the cluster, each one with a limited number of pods.

We refer to ''microservices'' as the components running separately in containers. Therefore, a microservices architecture provides the ability to update and modify each microservice independently without affecting others. Furthermore, as each microservice deploys and provisions different resources, it improves the modularity, scalability, efficiency of resource utilization, and performance of the application [14].

There are two main types of graphs for representing microservices topology: call graphs (CG) and directed acyclic graphs (DAG). As the main objective of using the graph is solely to trace dependencies between microservices, we employ the call graph to represent the interactions between tasks represented by microservices. In [15], a call graph was introduced to analyze microservice interactions within Alibaba's data centre. The call graph illustrates the frequency of these interactions and their performance implications, representing microservices and their communication patterns. Additionally, it categorizes the types of communication observed within the system, as shown in Figure 1. The topology of microservices is represented as a directed graph G, which consists of vertices and edges $G = \{V, E\}$, where vertices represent microservices and edges represent communications between them.

Two functional types of services can be run by microservices: stateless and stateful. Stateless microservices do not affect data, while stateful microservices interact with data, such as databases and Memcached, as shown in Figure 1.

A variety of communication paradigms are used in data centers, such as remote procedure calls (RPC) and RESTful APIs. These communication paradigms enable microservices to communicate frequently and efficiently. Streaming communication between microservices typically involves three types of microservices: the entry microservice (EM), the downstream microservice (DM), and the upstream microservice (UM). EM communicates with the user, while the DM and UM require back-end communication [15].

B. KUBERNETES

Kubernetes is an open-source platform used for container orchestration [16]. It manages the deployment of containers, load balancing, scaling, and their communication, making it an ideal platform for managing microservices. In Kubernetes, a cluster is a group of nodes consisting of a master node and worker nodes. The master node consists of various components that manage worker nodes in the cluster, including Kubelet, which plays a crucial role in managing containers on each node.

Each node contains multiple pods, each pod hosting a microservice container that utilizes the resources of the physical machine (PM). Additionally, Kube-proxy is a component located in each node that is responsible for managing communication between containers and the master node. The API server is another crucial component that serves as a user interface for accessing the Kubernetes cluster. Figure 2 illustrates the infrastructure of Kubernetes.

C. DOCKER SWARM

Docker Swarm is part of the Docker ecosystem that clusters nodes, which is a cluster orchestration tool. It ensures the reliability and maintainability of microservices during their distribution to containers. It consists of two types of nodes: manager nodes and worker nodes. The manager node schedules the workload and balances the load between workers. The worker nodes run the services in the containers and pull images of applications that run the container using the Docker daemon as shown in Figure 3. Docker Swarm does not monitor resource utilization but instead works with other tools in the Docker ecosystem to build, manage, and scale the cluster workload [17].

III. RELATED STUDY

We now discuss current research proposed to enhance resource utilization and reduce network traffic costs. Recent studies, such as [18], [19], and [20], aim to improve metrics like makespan, reliability, delay, and resource access using the Johnson sequencing method, Shortest Gap-Priority-Based Fair Scheduling (SG-PBFS), and priority-based fair schedul-

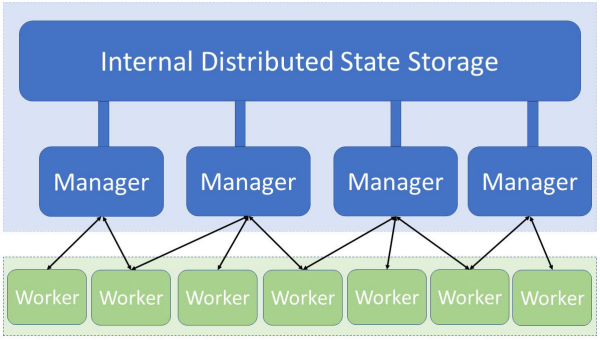


FIGURE 3. A cluster with Docker orchestration and its communication between manager nodes and worker nodes.

ing (PBFS), respectively. These algorithms contribute to the development of robust resource management frameworks that enhance resource utilization. In our research, we aim to further improve the availability of microservices and ensure efficient resource utilization.

A multi-objective approach using particle swarm optimization was proposed in [21]. The proposed approach for scheduling employed a hybrid multi-objective particle swarm optimization (HPSO) algorithm. The objective was to optimize the scheduling of multiple tasks with dependencies as scientific workflows to be distributed on cloud or grid environments. The proposed algorithm integrated PSO with multi-objective optimization techniques to address conflicting objectives such as minimizing makespan, resource utilization, and workflow execution costs. By leveraging PSO's ability to explore the search space efficiently and balancing multiple objectives, the proposed algorithm aims to enhance cloud performance. Several other studies explore multi-objective optimization for workflow scheduling in cloud computing, such as [22], [23], [24], [25], [26], and [27]. They aim to address various challenges and objectives summarized in Table 1.

Several proposed approaches, including a specialized scheduling algorithm based on particle swarm optimization (PSO-DS) [28], directional and non-local-convergent particle swarm optimization (DNCPSO) [29], an adaptive elite-based particle swarm optimization (NAEB-PSO) [30], and a multi-objective particle swarm optimization (MSMOOA) [31], utilize and improve modified particle swarm optimization methods. They aim to enhance PSO's performance in workflow scheduling. Certain objectives, such as local optimization and the optimal solution, were targeted by those studies. However, these approaches focus on optimizing execution time and cost, neglecting the reliability aspect of workflow execution.

To the best of our knowledge, all current research studies on multi-objective algorithms have been presented, and we observed that none of them considers essential factors such as resource utilization, latency, load balancing, and fault tolerance. Studies like [21] and [31] aim to reduce the cost of running time, makespan, and mandatory costs, including monetary expenses. However, the factors attempted to be optimized are affected by latency, resource utilization, and fault tolerance.

An analysis of large-scale microservices deployment in the cloud was conducted in [15]. The study investigated the topology of microservices in a production cluster at the Alibaba data center, represented as a *call graph*. The study first characterized the dependencies between microservices and their performance to gain a deep understanding of the scheduling process and resource utilization in the cluster. Then, a learning algorithm was proposed to cluster the graph into several classes, which provided a better understanding of the microservices structure. This method generated new benchmarks by optimizing the performance of large-scale microservices. The study suggested that reducing dependencies between microservices improves performance.

A runtime deployment solution called *Nautilus* was proposed in [32] to optimize microservice deployment in a cloud-edge continuum environment. It proposed to use a mapper that deploys microservices at the same nodes to reduce the overhead of network usage. Nautilus consists of a resource manager and a load-aware scheduler to enhance resource utilization and maintain quality of service (QoS). Nautilus considers tasks to use external I/O; therefore, it proposed an I/O-sensitive scheduler that migrates critical microservices to idle nodes. The experimental results showed that Nautilus reduced resource utilization (CPU) by 23.9% and network usage by 53.4% compared to standard resource management technologies.

The microservice-oriented topology-aware scheduling framework (MOTAS) was proposed in [33]. It aimed to reduce network traffic and improve resource utilization for the microservice architecture. The framework employs two main strategies to achieve its goals. First, it partitions the microservices graph into two sub-partitions based on dependencies using a hierarchical traversal approach. The scheduling strategy aims to ensure that dependent microservices are deployed before their independent microservices. Second, it eliminates nodes that violate resource balance provisions to reduce resource fragmentation. During deployment, MOTAS considers three objectives: reducing resource fragmentation, minimizing network traffic costs, and balancing workloads across nodes. The experimental results demonstrated that the framework successfully achieved the proposed goals. However, the scale of the experiment did not align with real-world cloud environments. Therefore, the authors suggested further investigation in future research.

A strategy was proposed in [34] to optimize network utilization using machine learning models. The machine learning model collaborates between nodes in the edge network and the cloud platform. The network architecture is divided into three layers: access nodes located on the edge network, the transfer network connecting the edge to the cloud, and the cloud platform. Two prediction models work together to predict upcoming network usage. The first

model aims to optimize incoming network utilization for each node. The second model uses prediction data from each access node and the current network utilization to predict upcoming network usage. The proposed algorithm predicts and enhances resource utilization based on predictions and network traffic. The experimental results showed that the proposed strategy enhances the prediction of network traffic by 9% compared to single-point traffic prediction. The experimental results also showed an increase in resource utilization.

The study in [35] suggested that Kubernetes has limitations in measuring the requirements of scheduled microservices because it does not consider runtime resource utilization. The authors also suggested that the limitations of metrics used during scheduling microservices to containers limit Kubernetes from satisfying the requirements of microservices. Additionally, Kubernetes only empowers the CPU and memory metrics during scheduling. It does not consider network communication requirements, which affect node-to-node latency. Therefore, the study proposed a modification in Kubernetes to consider the network requirements between nodes, application topology, and runtime resource utilization. The proposed modification works as an extension of the default Kubernetes. The study reported that these metrics enhance the quality of Kubernetes scheduling results.

The existing studies in the literature propose valuable optimization solutions for enhancing resource allocation and balancing the load. However, a few gaps are identified. Firstly, network traffic and latency are the aspects that affect cloud performance, yet they have not been considered efficiently. Secondly, microservice replication contributes to resiliency. Therefore, there is a need to consider fault tolerance using replication. The existing studies primarily focus on execution time and mandatory costs (monetary expenses), which are indeed important factors. However, resource utilization, network traffic costs, load balancing, and latency directly impact makespan and other associated costs. Table 1 provides an overview of current studies, highlighting their methodology, key contributions, and evaluation criteria.

IV. PROPOSED SCHEDULING STRATEGY

We aim to replicate microservices close to their neighbours while considering resource utilization. This approach reduces communication between dependent microservices and minimizes latency. Additionally, our strategy ensures that critical services are not co-located on the same PMs. This step reduces the risk of a single point of failure. The most suitable PMs to host microservice replicas are carefully selected by our strategy, taking into account resource utilization thresholds.

Our proposed scheduling strategy involves replication, which is specifically designed to meet the demands of microservice architectures. It aims to improve the underlying infrastructure and performance of the system. Our approach enhances the load balancing of PMs used to run microservices in a data center. It improves the deployment of replicas and manages the quality of service (QoS) and tolerance for failures. Additionally, our approach benefits the cloud platform by reducing network traffic and increasing the availability, reliability, and resilience of the deployed systems.

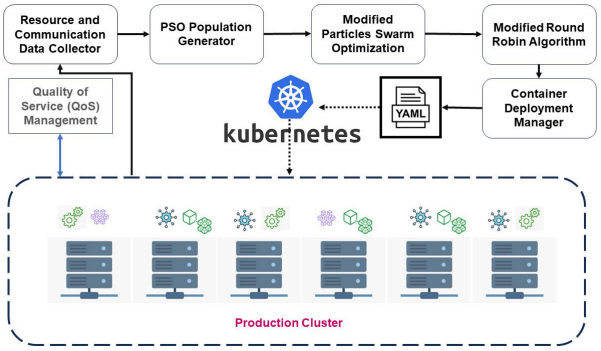


FIGURE 4. The components of our proposed scheduling strategy and its workflow. The components work sequentially to ensure the efficient and effective placement of microservice replicas in the cloud platform, making the overall system resilient and available at all times.

A. SCHEDULING STRATEGY ARCHITECTURE

In this subsection, we provide an overview of the proposed architecture of the scheduling strategy and demonstrate the collaboration between its components. As illustrated in Figure 4, our scheduling strategy consists of six components. The following highlights the components:

B. RESOURCE AND COMMUNICATION DATA COLLECTOR

It is responsible for collecting data for both resource utilization and microservice resource requirements. It also collects data on microservice communications. The data is then used by other components to decide the placement of microservice replicas. We have used historical data traces that describe the production cluster in data centers [15], [36], and [37] to simulate the cluster status and consider potential scheduling circumstances. However, the component can collect utilization resource data by using the Alibaba and Google Cloud Config services [37], [38].

C. PSO POPULATION GENERATOR

This component generates two populations. The first one represents a swarm of particles, which is the requirement of resources for each microservice. The second generation of data represents the resource availability of the potential PMs.

D. MODIFIED PSO

We aim to propose a scheduler based on population-based optimization, utilizing the Particle Swarm Optimization (PSO) algorithm. PSO is based on simulating a swarm of bees exploring the solution space and finding the optimal solution [39], [40]. However, our strategy involves a modified PSO. It guides each particle throughout the entire solution space. Therefore, we modified PSO as follows:

TABLE 1. A comprehensive overview of current research in microservices deployment optimization highlighting their objectives, focus, employed methodology and key contributions.

Study	Focus	Methodology	Key Contributions	Resource utilization	Load Balance	Latency	Fault Tolerance	Cost- money	makespan
[15]	Microservice topology analysis	Characterizing and analysing microservice dependencies and performance	Clustering algorithm for understanding microservice structure	×	×	×	×	×	×
[22]	Optimization of multi-objective scheduling problems in grid	Evolutionary algorithms for exploring the solution space and finding Pareto optimal solutions	Addressing conflicting objectives such as execution time, total cost, reliability, and energy consumption	×	×	×	×	✓	✓
[23]	Generating Pareto optimal solutions	Extending the Non-dominated Sorting Genetic Algorithm II (NSGA-II)	Trade-offs between objectives and finding the optimal solution for all objectives	×	×	×	✓	✓	✓
[24]	Heuristics scheduling algorithms for workflow	Approximating the optimal solution of scheduling based on user-defined constraints for each objective	Trade-off between execution time and reliability objectives	×	×	×	✓	✓	✓
[25]	Task scheduling	Bi-objective dynamic level scheduling algorithm and the bi-objective genetic algorithm	Prioritizing execution time against reliability	×	×	×	✓	✓	✓
[26]	Executing large programs in the cloud	Pareto dominance to map more efficiently and reduce the cost for non-critical task	Utilizing Pareto dominance to schedule the application to the cloud, considering time and cost	×	×	×	✓	✓	✓
[27]	Muily objective scheduling using PSO algorithm	Fuzzy-based mechanisms using PSO	PSO generating optimal solutions using Pareto to optimize makespan, cost and reliability	×	×	×	✓	✓	✓
[28]	Strategic tasks scheduling method	Addressing the discrete scheduling problem using integer representation	Reducing the makespan and within limited budget	×	×	×	×	✓	✓
[29]	Obtaining the optimum scheduling result	Non-linear inertia weight with a meta-heuristic algorithm within the directional search process	Utilizing modified parameters of PSO in D dimensions for faster convergence and to reduce makespan and total resource cost	×	×	×	×	✓	✓
[30]	Multi-objectives scheduling algorithm	Utilizing adaptive elite-based particle swarm optimization (NAEB-PSO)	Balanced load resulting in utilizing the resources efficiently	×	✓	×	×	✓	✓
[31]	Multi-objectives scheduling algorithm	partitioning the PMs into different swarms and scheduling them based on distinct objectives. Sharing information among swarms during scheduling helps find non-dominated solutions	Utilizing a multi-swarm mechanism and modifying the method of updating particles' velocity in the PSO	×	✓	×	×	✓	✓
[32]	Microservices deployment in cloud-edge continuum	Nautilus runtime deployment scheduling	Reduction in CPU usage and network bandwidth compared to traditional approaches	✓	✓	×	×	×	×
[33]	Microservices topology-aware scheduling	Partitioning microservices graph based on dependencies	Resource fragmentation reduction, minimized network traffic costs, workload balancing	✓	✓	×	×	×	×
[34]	Edge-cloud network utilization optimization	Resource allocation using collaborative machine learning prediction	Enhanced network traffic prediction and resource utilization	✓	×	×	×	×	×
[35]	Kubernetes limitations	Modification of Kubernetes using network traffic metric	Consideration of network communication and runtime resource utilization	×	×	×	×	×	×

1) For each PM, we group dependent microservices located in different PMs into one swarm.
2) Neighboring PMs for each PM are grouped, forming targets for the particles.
3) Swarm each group of particles towards their respective targets.

For instance, consider a PM denoted as PM_c, hosting a set of microservices M_1, which depend on a set of microservices M_2 distributed across different PMs. In this scenario, M_2 forms a swarm of particles labelled as Sw_{M2}. Additionally, neighbouring PMs of PM_c, sharing the same pod in the data center and having the shortest network path to

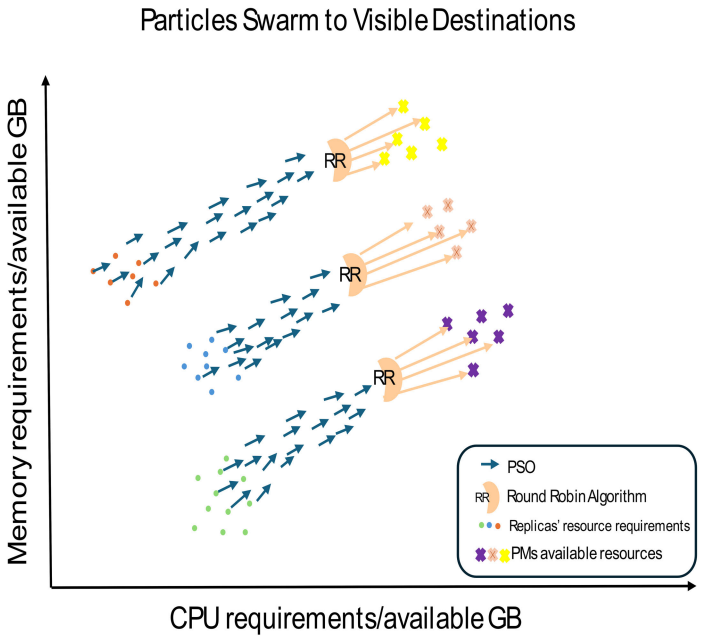


FIGURE 5. The particle swarm approach is used to find the best replica placement for microservices in a data center. Each particle in the swarm represents the best solution for the replica, and its position in the space corresponds to the resource requirements of the replica using the round-robin algorithm (RR). The viable solutions, representing available resources in PMs, are also depicted in the space. The swarm moves towards viable solutions, guided by a combination of individual and social factors.

PM_c, are formed as the target for Sw_{M2}. This group of targets is denoted as tr_{PM_c}. The algorithm swarms Sw_{M2} towards the target tr_{PM_c}.

Instead of directing each particle towards the entire solution space, our modified PSO directs a group of particles towards a group of targets. Each target group contributes to reducing dependencies between dependent microservices. While the obvious optimum solution for each particle is to replicate the same PM of its dependent microservice, this approach degrades system reliability by increasing the probability of a single failure. Therefore, the group of targets should include all PMs close to PM_c, meaning those sharing the same pod in the data center and having the shortest network path to PM_c. PSO then find the optimal solutions of targets to be utilized in the next component of our strategy, which require modified RR as shown in Figure 5.

E. MODIFIED ROUND ROBIN ALGORITHM

It is used to schedule replicas across the potential PMs, taking into account their load balancing. Once the modified PSO determines the optimal PMs, the Round Robin (RR) algorithm then places the microservice replicas into those PMs. RR is used to fulfil the requirement of load balancing in our strategy. The combination of modified PSO and RR algorithms ensures the efficient and effective placement of microservice replicas.

F. CONTAINER DEPLOYMENT MANAGER

It manages the deployment of replicas to selected PMs identified by the modified PSO and RR algorithms. The component receives the output from the modified PSO and RR algorithms, such as the number of replicas and the resource requirements for each replica, as well as the available resources of the target PMs. It then configures the deployment for each microservice in YAML or JSON format and hands them over to Kubernetes to deploy containers that run the microservices. Furthermore, the component configures the communication between microservices by setting up network policies and configuring service discovery in Kubernetes.

G. QUALITY OF SERVICE (QOS) MANAGEMENT

This component is responsible for ensuring that all QoS requirements are met during the deployment process. It includes the minimization of the risk of intensive resource utilization. It also manages the resource threshold for each PM, monitors for single system failures, and reports any violation of the service level agreement (SLA).

V. KEY ASSUMPTIONS OF THE PROPOSED SCHEDULING STRATEGY

We believe that many of our assumptions would apply to any data centers around the world. However, for this research, we used traces made public by the Alibaba and

Google data centers. It was assumed that the physical machines (PMs) used in the data centers have heterogeneous capacities. We also assume that each cluster in the data center has a limited number of PMs, represented as a set pm=$\{pm_1,pm_2 \ldots pm_n\}$. Additionally, we assume that each PM has limited resources with a capacity of C. Each resource type is represented as PM_c=$\{pm_{cpu},pm_{mem},pm_{io}\}$, where cpu, mem, and io subscripts represent CPU, memory, and I/O capacities in each PM, respectively.

We also assume that an application A is built using a microservice architecture and consists of microservices. Each microservice is represented by a loose task or function. The application A consists of a limited number of microservices: $A = m_1, m_2, \ldots, m_s$, where the subscript represents the identity of the microservices. We assume Kubernetes is used to manage a cluster of applications, and it allocates each microservice to a pod in the cluster. Therefore, a cluster K consists of a limited number of pods p as: K_i=$\{p_1,p_1,\ldots,p_k\}$ where k represents the number of pods. Kubernetes is also responsible for managing the routing between microservices in the cluster. As our scheduling strategy is based on microservice replicas, we assume that Kubernetes manages the startup communication between replicas and reroutes traffic between the same replicas in case one of them fails, as suggested in [41]. As presented in Figure 2, every microservice runs in a container that is allocated in a pod within a cluster.

Additionally, the network architecture is considered to be a fat-tree topology. This topology is popular in data centers as it provides a data center with high bandwidth and multiple paths, increasing the efficiency of communication between data center nodes [42], [43], [44]. Finally, the communication bandwidth is considered to be heterogeneous in the data center.

Before describing the components of the proposed scheduling strategy, it is emphasized that our proposed scheduling strategy does not involve scheduling new microservices into the cluster. Instead, we aim to enhance the network traffic of existing scheduled microservices in the data center by using a replication scheduling strategy. We also assume that each pod is located in a different zone that requires inter-communication.

A. PROPOSED SCHEDULING STRATEGY FORMULATION

The proposed strategy aims to optimize network traffic in the data centers by scheduling replicated microservices close to their dependents. In addition, building upon the six components outlined earlier, our strategy aims to enhance system availability and resilience. It also balances the load across potential PMs. We will explain how these components collaborate to achieve the objectives. In addition, Table 2 illustrates the notations.

1) RESOURCE AND COMMUNICATION DATA COLLECTOR MODEL

We model each physical machine in the cluster as a node N. Each cluster consists of a limited number of nodes. Each node has maximum resource capacities denoted as NC_{cpu}, NC_{mem}, and NC_{io} for CPU, memory, and I/O, respectively. Each node is occupied by R containers that require an elastic amount of resources. As the utilization of the resources by the containers varies, we consider the maximum utilization of each resource in a fixed time interval t. Therefore, the total utilization by all containers in a fixed time interval $U(t)$ can be described by the following equations:

$$U_{cpu}(t) = \frac{\sum_{i=1}^{R} U_{i,cpu}(t)}{NC_{cpu}}$$
$$U_{mem}(t) = \frac{\sum_{i=1}^{R} U_{i,mem}(t)}{NC_{mem}}$$
$$U_{io}(t) = \frac{\sum_{i=1}^{R} U_{i,io}(t)}{NC_{io}} \quad (1)$$

Furthermore, we take into account the dependencies between the microservices using a call graph. Each node has R containers with d dependencies. Each node has its own call graph that represents the dependencies between its dependent microservices as shown in Figure 6. Kubernetes can collect microservice dependencies by utilizing microservice manifest files [45]. However, a dataset of microservice call graphs was reported to be used in experiments [15]. Therefore, this component aims to collect data on dependencies between nodes using Kubernetes.

TABLE 2. List of notations.

Notation	Meaning
N	Node representing a physical machine in the cluster
NC_{cpu}	Maximum CPU resource capacity of a node
NC_{mem}	Maximum memory resource capacity of a node
NC_{io}	Maximum I/O resource capacity of a node
R	Number of containers occupying a node
$U(t)$	Total resource utilization by all containers at time t
$U_{cpu}(t)$	CPU resource utilization at time t
$U_{mem}(t)$	Memory resource utilization at time t
$U_{io}(t)$	I/O resource utilization at time t
d	Number of dependencies for a container
$pb_{c,D}$	Best position for the cognitive component in dimension D
$pb_{g,D}$	Best position for the swarm in dimension D
$pb_{t,D}$	Best position for the target component in dimension D
$vel_{cognitive}$	Velocity component influenced by cognitive component
vel_{social}	Velocity component influenced by social component
vel_{target}	Velocity component influenced by target component
c_1, c_2, c_3	Constants for cognitive, social, and target
$rdom_{1,2and3}$	Random numbers between 0 and 1
w	Inertia weight controlling the influence of previous velocity
$SP_{(N_h)}$	Swarm population for node N_h
$x_{i,d}$	Resource demand of CPU for dependent microservice d
$y_{i,d}$	Resource demand of memory for dependent microservice d
xy_i	Initial position vector for particles in the swarm
$pbest$	Best position for an individual particle in the swarm
$gbest$	Best position for the entire swarm
$target$	viable solution for particles in the swarm
L_r	List of replicas to be scheduled
L_{pm}	List of eligible PMs that can host replicas
NF_{cpu}	CPU resource availability of a potential PM
NF_{mem}	Memory resource availability of a potential PM
U_{th}	Upper threshold constraint on PMs to maintain SLA
$dep_{i,h}$	Number of hops between source and destination nodes
S	Set of viable solutions
$f(x)$	Fitness function for evaluating positions of particles
N_{prt}	Number of particles in the swarm
D	Number of dimensions in the search space

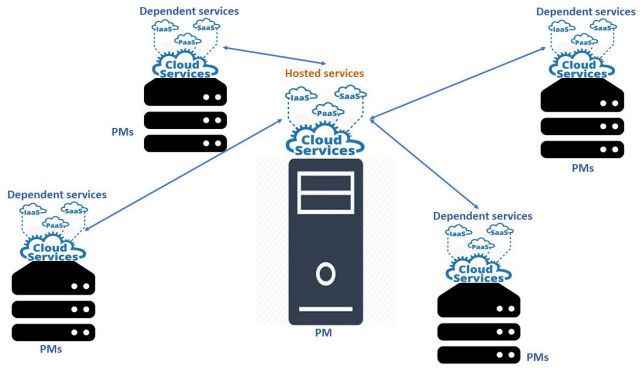


FIGURE 6. The call graph of PMs shows dependencies between hosted microservices allocated to PMs. The hosted microservices depend on other microservices located in other PMs, which are called dependent microservices. Each PM in the cluster has its call graph that consists of hosted and dependent microservices.

2) PSO POPULATION GENERATOR MODEL

We modified the particle swarm optimization (PSO) algorithm to optimize the allocation of microservice replicas to the most suitable PM. The initial population of the PSO algorithm consists of a swarm of particles, with each particle representing a vector of resource demands for a microservice. Two resources are considered in this research: CPU and memory.

For each node N in the data center, a swarm population is created based on two types of microservices: hosted microservices and dependent microservices. Hosted microservices are those that are hosted by the node N_h. In contrast, dependent microservices communicate with hosted microservices but are hosted on different nodes, as shown in Figure 6. The swarm population for node N_h is represented by the resource demands of dependent microservices that need to be replicated on N_h or its neighbour node N_h^{nieg}. In this research, we attempt to replicate dependent microservices in the hosted node N_h, or in the neighbour nodes, which are nodes that share the same pod as the host node N_h.

Each particle is positioned in a two-dimensional search space, with x representing the resource demand of the CPU and y representing the resource demand of memory. The swarm population for node $SP_{(N_h)}$ is represented as follows:

$$SP_{(N_h)} = (x_{i,d}, y_{i,d}) \tag{2}$$

where $d = 1, 2, \ldots, R_{i_d}$, $x_{i,d}$ represents the resource demand of CPU for dependent microservice d that needs to be replicated on node N_h or N_h^{nieg}, and $y_{i,d}$ represents the resource demand of memory for the same microservice d.

A potential solution for a problem is represented as vector values of available resources of a potential PM. Instead of searching for a solution in the solution space, our strategy is to constrain the swarm to move toward viable solutions. For example, assuming that node N_h has dependent microservices located in several PMs, the particles in the swarm are a set of particles that represent the demand for dependent microservices, and are denoted as N_h=$\{pr_1, pr_2 \ldots pr_n\}$. Each pr searches for viable solutions, which are a set of S=$\{s_1, s_2 \ldots s_n\}$. The viable solutions are the available resources of potential PMs that can be N_h or a list of its neighbours represented as s_i=(NF_{cpu}, NF_{mem}), where NF_{cpu} and NF_{mem} are calculated as follows:

$$\begin{aligned} NF_{CPU} &= NC_{cpu} - U_{i,cpu}(t) \\ NF_{mem} &= NC_{mem} - U_{i,mem}(t) \end{aligned} \tag{3}$$

where $U_{i,cpu}(t)$ and $U_{i,mem}(t)$ represent the utilization of CPU and memory, respectively, at a specific time point t. These values indicate the amount of resources currently used at time t.

Each viable solution should minimize the network traffic cost. The objective function for selecting a viable solution is represented by the following: equation:

$$\begin{aligned} &min\, f(s) = \sum_{i=1}^{n} dep_{i,h} \\ &Subject \quad to: \\ &\qquad s_i \in S \;\; where \;\; i = 1, \ldots, n \end{aligned} \tag{4}$$

where $dep_{i,h}$ represents the number of hops between the source and destination nodes for the communication between the dependent microservice d_i and its host microservice N_h, which can be calculated using Dijkstra's algorithm.

Each viable solution s_i is a potential PM that has enough resources to host the dependent microservice as follows:

$$\begin{aligned} NF_{cpu}(s_i) &\geq x_{i,d} \; and, \\ NF_{mem}(s_i) &\geq y_{i,d} \\ \forall d &= 1, \ldots, R_{i_d} \end{aligned} \tag{5}$$

3) MODIFIED PARTICLE SWARM OPTIMIZATION (PSO) MODEL

Particle Swarm Optimization (PSO) is a population-based optimization algorithm [46] inspired by the behaviour of animals, such as flocks of birds. The algorithm is shaped by three main elements that govern its behaviour. Those three elements are described below:

1) Swarm initialization: This element is responsible for initializing the particles of the swarm. Three types of positions are considered in our research: particle position, denoted as *pbest*, swarm position denoted as *gbest* and viable solution, denoted as *target*. Our scheduler initializes the positions as XY=$\{xy_1, xy_2, \ldots xy_z\}$, $pbest$=$\{pb_1, pb_2, \ldots pb_k\}$, $target$=$\{tg_1, tg_2, \ldots tg_l\}$ which have the initial, best, and target positions, respectively, for all particles in the swarm. This component assigns an initial value of XY, *pbest* and *target* for each particle based on the demand for resources and the viable solution as *pbest* and *target*, whereas it initializes XY and *gbest* randomly.
We modified PSO to swarm each group of particles. In our case, microservice replicas swarm to the best solutions, which are denoted as *target*. On the other

hand, *pbest* is determined based on how close or far it is from the optimum solution *target* for each particle in the same group. The best position for all particles is *gbest*, which encourages collaboration and movement between particles from the same group toward a viable solution.

2) Fitness function: It evaluates each position of all particles based on the objective function. In our scheduling strategy, the aim is to minimize the distance between the best position of the particle *pbest* and its viable solution *target*. In our strategy, a fitness function is defined as:

$$f(x) = \frac{1}{N_{prt}} \sqrt{\sum_{j=1}^{D} (pbest_{ij} - target_{ij})^2} \tag{6}$$

where N_{prt} is the number of particles, D is the number of dimensions in the search space, j is the jth dimension of the position of the ith particle.

3) Updating the particle position: The movement of the swarm is updated to the optimal solution using cognitive, social, and target values of velocities. Each of those values updates the position either for the individual particle or for the swarm. The velocity *vel* components are calculated as [46]:

$$\begin{aligned} vel_{cognitive} &= c_1 \cdot rdom_1 \cdot (pb_{c,D} - XY_D) \\ vel_{social} &= c_2 \cdot rdom_2 \cdot (pb_{g,D} - XY_D) \\ vel_{target} &= c_3 \cdot rdom_3 \cdot (pb_{t,D} - XY_D) \end{aligned} \tag{7}$$

where c_1, c_2 and c_3 are the cognitive, social, and target constants, respectively. $rdom_1$, $rdom_2$, $rdom_3$ are random numbers between 0 and 1. $pb_{c,D}$, $pb_{g,D}$ and $pb_{t,D}$ are the best positions, the current position, and the best position for the swarm, respectively, in dimension D.

The velocity of each particle is updated using these three values, where the weights c_1, c_2, and c_3 control the velocity direction and speed. An equation that combines the three values and updates each particle's position is proposed as [46]:

$$\begin{aligned} vel_{D,t+1} = w \cdot vel_{D,t} + vel_{cognitive} + vel_{social} \\ + vel_{target} \end{aligned} \tag{8}$$

where w is denoted as a constant inertia weight that controls how much the previous velocity influences current velocity.

4) MODIFIED ROUND ROBIN ALGORITHM MODEL

The round-robin (RR) algorithm is a well-known algorithm used to share CPU time among processes [47]. We aim to utilize RR to distribute replicas among potential PMs. In our strategy, the modified RR algorithm is responsible for scheduling replicas equally among the eligible PMs. The aim is to schedule the replicas close to their dependent microservices. Therefore, list L_r for the replicas and another list, denoted as L_{pm}, for the eligible PMs are created. Then, the RR algorithm schedules the replicas into the PMs equally while meeting the resource demands for the replicas. Additionally, we apply an upper threshold U_{th} constraint on PMs to maintain the service level agreement (SLA).

The scheduling of the replicas into the PMs by the RR algorithm is as follows: For each replica r_i in L_r, the scheduling strategy searches for the first PM pm_j in L_{pm} that can satisfy the resource requirements of r_i. Once a suitable PM pm_j is identified and allocated to host r_i, the scheduling strategy proceeds to search for placing the next replica. It starts with the subsequent PM in the list L_{pm}. This process repeats for all replicas in L_r until no replica is left in the list or insufficient resources are available to host the remaining replicas.

5) CONTAINER DEPLOYMENT MANAGER

Our scheduling strategy uses Kubernetes as a container deployment manager. This part of our strategy involves defining a configuration file in "YAML" format that specifies the image name, version, resource requirements, and number of replicas. It also includes other details related to the deployment of the microservices into containers. Kubernetes manages the deployment and scaling of the microservices. Additionally, it provides microservice discovery and health-checking mechanisms. Our approach involves configuring replica scaling, which aims to be more efficient in management. It also provides reliability and resiliency in the event of failure.

6) QUALITY OF SERVICE (QOS) MANAGEMENT

This component of our scheduling strategy is responsible for monitoring resource utilization in each PM. It also reports the resource utilization to the RR algorithm component whenever a new replica is deployed. Moreover, it updates the resource utilization data to ensure that the RR algorithm component is aware of the current state of the system. Additionally, this component reports any SLA violations that may arise at any given point in time.

7) ALGORITHM COMPLEXITY ANALYSIS

As shown in the proposed algorithm, the proposed algorithm combines PSO and RR algorithms to schedule the dependent replicas to the closest PMs to their dependencies. Therefore, in this section, the complexity of the algorithm is calculated as follows:

1) The algorithm iterates through all PMs N to find the eligible PM to host the replica. The complexity of the proposed algorithm at this stage is $O(N)$.
2) Additionally, the PSO algorithm swarms particles microservice replicas to their destinations; therefore, PSO also iterates through all replicas M, which makes the cost $O(M)$.

3) Allocation of replicas to PMs using Round Robin involves operations for each replica and PM, making the overall complexity of the algorithm $O(N \times M)$.

Algorithm 1 Proposed Scheduling Strategy

Resource and Communication Data Collection:
for $pm \in PMs$ **do**
  $U_{cpu} \leftarrow getCPUUtilization(pm)$
  $U_{mem} \leftarrow getMemoryUtilization(pm)$
  $depe \leftarrow calculateDependency(pm)$
end for
PSO Population Generator:
$Particles \leftarrow population(microservice(PM_i))$
$BestPosition \leftarrow neighbour(PM_i)$
Particle Swarm Optimization (PSO):
for $p \in Particles$ **do**
  if $p_{position} \neq BestPosition$ **then**
    $p_{bestposition} \leftarrow velocity(pb_D, pb_{g,D}, pb_{t,D})$
  end if
end for
Round Robin Algorithm (RR):
for $pm \in pm$ **do**
  $L_r \leftarrow generatReplicas(pm)$
  $L_{pm} \leftarrow getNeighbour(pm)$
end for
for $r \in L_r$ **do**
  for $l \in L_{pm}$ **do**
    if $l_{avCPU} \geq r_{reqCPU}$ **then** and
      if $l_{avMem} \geq r_{reqMem}$ **then**
        Allocate(pm, r)
      end if
    end if
    $YAMEL \leftarrow CreateYAML(r)$
    call(Kubernetes(pm,YAMEL))
  end for
end for

VI. EXPERIMENTAL SETUP AND METRICS

This section introduces the experimental setup details, followed by discussing the metrics used for measuring the performance of our scheduling strategy. The code of the experiment is provided in Download Code [48].

A. EXPERIMENTAL SETUP

The experiments were conducted on an HP computer equipped with an AMD Ryzen 7 5800U CPU and 16.0 GB of RAM. The experiments were simulated using Python programming language. We simulated Alibaba's data center network infrastructure using Fat-Tree topology to connect over 1000 PMs. The network topology has k switches with three levels of switches (access, pod, and core). It consists of a limited number of pods denoted by P. The data for the 100 PMs that were selected for the experiment are located in 17 different pods as shown in Figure 7 and each pod is equipped with a switch with 18 ports.

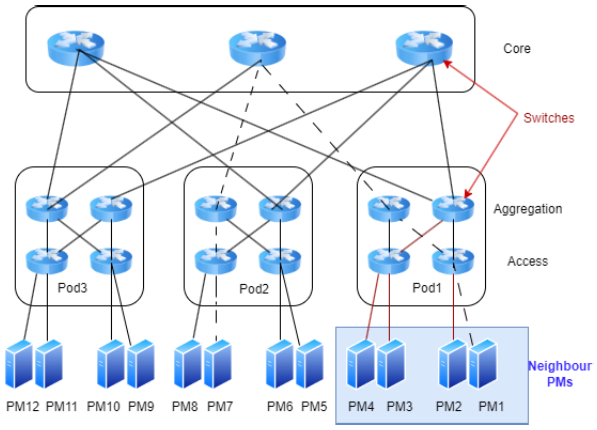


FIGURE 7. The architecture of a Fat-tree network topology consists of three layers of switches: core, aggregation, and access. Close neighbours are PMs located in the same pod that share the same switch, while far neighbours are those that use different switches in the same pod. The figure also shows the communication between PMs from different pods.

B. DATASET

There are two popular real-world datasets for testing scheduling strategies: Google [37], [49] and Alibaba [15], [36]. Alibaba data trace describes a production cluster and is made publicly available by the Alibaba data center. The dataset includes essential details such as node IDs, timestamps, and comprehensive resource utilization information. We found that the dataset is ideal for our study because it describes the infrastructure as a microservices-based data center. It also provides in-depth insights into job allocation, machine utilization, and dependencies.

The data-trace describes the activities on a production cluster for a month. The dataset comprises the resource utilization data of over 90,000 containers running on more than 1,300 PMs. The dataset includes node ID, timestamp and resource utilization details for each container. Additionally, it includes microservice IDs, types, and communication patterns. We extracted data from over 12,125 microservices running for one hour on nearly 100 PMs.

The second dataset is a Google trace collected by the company's system called Borg. The data is described in [37]. It consists of a monthly trace detailing all activities, including job scheduling, resource utilization, and job durations. Due to the dataset's size and our machine's limitations, records of approximately 9,000 jobs were then replicated to simulate a workload of 24,000 jobs scheduled across 100 physical machines (PMs). The jobs that arrived at the data center at different times make it suitable for simulating scenarios with new job arrivals at various times. Additionally, the data describes resource utilization every 5 minutes.

In contrast to the Alibaba dataset, the Google dataset does not include a description of the dependencies between microservices. Therefore, to simulate the real-world scenario of scheduling microservices, dependencies between microservices were generated. These dependencies are rep-

resented in four separate matrices, each corresponding to a different time interval. The first matrix describes the dependencies between microservices at the first interval, the second matrix at the second interval, and so forth. It is important to note that microservices may arrive at later intervals but still have dependencies with microservices that arrive at earlier intervals. For example, a microservice arriving in the third interval may have dependencies with microservices that arrived in the first or second interval. This dynamic nature of dependencies reflects the evolving nature of microservice architectures and ensures a realistic simulation of scheduling scenarios.

C. EXPERIMENT SCENARIOS

Two scenarios are designed for the experiment. The first scenario aims to evaluate the performance of the proposed algorithm in its convergence to the optimum solution. We assume that users submit applications to be scheduled with a fixed number of replicas. This scenario is common in cloud computing, where users operate with limited resources. We also assume that external traffic fluctuates and is managed by the Kubernetes load balancer. The objective of this scenario is to assess the algorithm's performance in scheduling applications that run for long-term durations (months) in the cloud.

In the second scenario, users can request scheduling for their microservices at any time. The algorithm operates in defined time intervals for gathering data on PMs and microservices, then generates particles to find the optimal scheduling solution. To simulate this scenario, we utilize a Google Trace dataset, which records microservices every 5 minutes. We record the performance of the algorithm at four different time intervals. Additionally, we measure the percentage of newly arriving microservices in each interval to assess the escalation of the number of arrivals of microservices and evaluate the algorithm's performance in response to this escalation.

D. PSO SETUP

Two types of populations are generated by our proposed strategy. The first type consists of the demands on resources made by microservices. Our strategy divided the first population based on the hosts (PMs). Therefore, around 100 populations for the first type were created by our strategy. Each population consists of a different number of particles. The particles hold all information about the original microservice, such as the microservice name, and host PM.

The second type of population is generated as the available resources of the hosted PM and its neighbouring PMs. This population is known as the best destination or viable solution. The aim is to move each particle of each group of the first population to the viable destinations and then use the RR algorithm to schedule a new replica of the original microservice Figure 5.

TABLE 3. Experimental Parameters.

Experimental Parameters	Description
Datasets	Alibaba's data trace and Google trace.
Workload Size	The experiment simulated over 12,125 microservices running on nearly 100 physical machines (PMs) from Alibaba's dataset trace. In addition, 24,000 jobs from the Google dataset were used.
Arrival Frequency	Microservices arrived at the data centre at different intervals, recorded every 3 seconds and 5 minutes in the Alibaba trace and Google trace datasets, respectively.
PMs	Approximately 100 physical machines (PMs) were involved in hosting microservices.
Microservice resource requirements	Over 12,125 microservices from Alibaba's dataset and 24,000 from the Google dataset were simulated, with various capacities between 0 and 1 representing the resource requirements.
PSO Parameters	Constant inertia weight (w = 0.5), cognitive constant (c1 = 0.2), social constant (c2 = 0.2), target constant (c3 = 0.1), velocity values (r1, r2, r3 randomly generated between 0.1 and 0.9), iterations of 500.
RR Parameters	Generating replicas for selected neighbouring PMs, and then allocating resources based on CPU and memory requirements.
Network Topology	A fat-tree architecture with three layers of switches: core, aggregation and access.
Resource Capacity	128 cores, 512 GB and 10 GB as capacities of CPU, memory, and network bandwidth, respectively.
Network Architecture	Network Architecture simulated consisted of 17 pods, 9 switches each, and 9 ports for each switch with a capacity of 512 PMs.
Metric Evaluation	Network traffic cost, latency, and load balancing.

The parameters for the PSO algorithm are set as follows:

1) The constant inertia weight $w = 0.5$ determines the influence of the previous velocity of the particle on its current position.
2) The cognitive constant is set as $c_1 = 0.2$, which represents the weight of the best position of an individual particle in its movement towards the viable solution.
3) The social constant is set to $c_2 = 0.2$, which represents the weight of the best position of the swarm that influences the movement of the particle towards the viable solution.
4) The target constant is set to $c_3 = 0.1$, which is used to update the position of the viable solution of each particle.
5) The values for velocity r_1, r_2, and r_3 are randomly generated between 0.1 and 0.9. Those values are used to update the velocity of each particle.
6) The maximum number of iterations for each swarm to move to a viable solution was set to 500.

Choosing these parameters allows for balancing the exploration process in PSO. In addition, it enhances convergence and encourages the efficiency of the movement of the particles, which results in improving the speed and effectiveness of optimizing the search. The parameter values are chosen as they have proven their effectiveness in literature [50], [51] [52].

E. EXPERIMENTAL METRICS

Three metrics are used in our experiments to evaluate the efficiency of our proposed scheduling strategy: network

traffic cost, latency, and load balance. These metrics are explained below in detail.

1) NETWORK TRAFFIC COST

Latency represents the cost of network traffic due to microservice-based application dependencies. In this research, we are not considering any traffic that comes from outside the data center including HTTP requests made by users. Instead, we count the number of edges between the PMs that host dependent microservices and consider the shortest path between those PMs. There are two types of network traffic: 1) traffic between PMs located in the same pod; 2) The bottleneck in terms of bandwidth is located between the aggregation layer and the core layer, which is considered the most expensive part of the network traffic. The historical data trace is represented as a call graph. Therefore, the network cost is to aggregate the edges between dependent microservices m_i and m_j that are located in different PMs.

$$\text{Network Traffic Cost} = \sum_{1}^{m_k} \sum_{1}^{m_j} (E_{m_k,m_j}) \tag{9}$$

where E_{m_i,m_j} is the count of communication edges between m_i and m_j located in different PMs.

2) LATENCY

It is a result of the delay in packet travel between switches during communication between microservices. Routing algorithms used for packet delivery also affect latency. In addition, the distance between microservices contributes significantly to latency. We measure latency in our experiment considering the bandwidth bottleneck and the number of packets sent in a certain amount of time t, as shown in the following equation:

$$\text{Latency(t)} = \frac{\sum_{1}^{m_k} \sum_{1}^{m_j} E_{m_k,m_j} * Z}{Bandwidth} \tag{10}$$

where m_i, m_j are the two dependent microservices, E is the number of edges between those microservices, and Z is the size of the packet sent by the microservices.

According to Alibaba's dataset documentation, the average latency between data centres in different zones is extremely low. Specifically, the average latency between Alibaba's data centers in Silicon Valley, USA, and Hangzhou, China, is reported to be only 175ms [53]. In our experiment, we intend to use these metrics, and therefore the latency equation becomes:

$$\text{Latency(t)} = \sum_{1}^{N} D_{source,N} * lat_V$$

$$D_{souc,N} = \begin{cases} 1 \text{ if there are dependencies between source,d} \\ 0 \text{ otherwise} \end{cases} \tag{11}$$

where N represents the number of nodes in the cluster, '$D_{source,N}$' denotes the metric of dependency between the source node and all cluster nodes, and 'lat_V' represents the latency value in Alibaba's data center, which is taken as 175ms.

To mitigate network traffic costs, our proposed strategy aims to minimize the usage of the network. Therefore, our scheduling strategy deploys replicas of microservices closer to their dependents to reduce the distance between them.

3) LOAD BALANCING

It is a significant factor in microservices performance. We are using the RR algorithm to balance the load of PMs. We aim to improve the utilization of resources in all PMs in the data center. Therefore, the total resource utilization is calculated for each PM using Eqn. 1.

However, to measure the load balance, we use the standard deviation (SD) along with the mean to estimate the effectiveness of the load balance. The following equations illustrate SD and the mean, respectively [54]:

$$\text{SD} = \sqrt{\frac{1}{N_{total}} \sum_{p=1}^{N_{total}} \left(\text{PM}_p - \text{mean}\right)^2} \tag{12}$$

$$\text{mean} = \frac{1}{N_{total}} \sum_{p=1}^{N_{total}} \text{PM}_p \tag{13}$$

where N_{total} is the total number of nodes in the cluster.

VII. COMPARISON WITH BASELINE STRATEGIES

Recently, cloud platforms have offered a vast amount of resources that enable the consolidation of a large number of microservices into a smaller number of PMs. This opportunity has encouraged researchers to propose strategies focused on replication services to scale the service and ensure its availability. However, these approaches increase resource utilization without considering other factors such as load balancing or network traffic.

In our proposed strategy, we employed two distinct standard PSO strategies. The first is PSO with the best-fit algorithm, which focuses on consolidating replicas to enhance resource utilization. The second is PSO with an adaptive threshold, aiming to maintain QoS by preventing resource over-utilization and SLA violations. It also aims to balance the load by using an adaptive upper threshold of resource utilization. Both strategies aim to reduce network traffic and enhance resource utilization.

A. PSO WITH THE BEST-FIT ALGORITHM

This strategy focuses on distributing replicas of services within the same PM. It uses the PSO algorithm to generate particles representing replicas of microservices. Then, it moves them to the best solutions, which are the PMs hosting dependent microservices. The best-fit algorithm is then used to schedule the replicas by sorting the potential PMs based on two factors: reducing network traffic utilization and increasing resource utilization. The sorting is based on the distance between the microservices and their replicas,

with the PMs hosting the microservices at the top of the list. A static upper threshold for resource utilization is utilized to minimize SLA violations.

B. PSO WITH AN ADAPTIVE THRESHOLD

This strategy adjusts the replication threshold dynamically based on current resource utilization. It employs a PSO algorithm to direct particles representing microservice replicas to the best PMs. It uses an adaptive upper threshold of resource utilization as a constraint. This technique efficiently uses resources and reduces the cost of network traffic. However, the utilization of resources by replicas is highly unpredictable and can experience sudden spikes at any time. This unpredictability increases the risk of violating the SLA. Furthermore, the aggressive consolidation of replicas into a small number of PMs amplifies the risk of a single PM failure resulting in system failure.

However, we realize that both of these PSO strategies have limitations, and we propose a modified PSO which was explained earlier. We compared the results of all three PSO strategies to demonstrate that modified PSO works best in our proposed scheduling strategy.

VIII. RESULTS AND DISCUSSION

In this section, different aspects of experimental results and their comparison with baseline algorithms are discussed in detail. The experiment ran for approximately three hours, covering the simulation of the cloud computing system from the start to the completion of the first scenario. Similarly, the second scenario also took nearly the same amount of time.

A. REDUCING NETWORK TRAFFIC COST

For the first scenario, our scheduling strategy achieved a 36% reduction in network traffic compared to Alibaba's scheduling strategies by deploying 11,004 replicas near the dependent microservices. 12,125 microservices were distributed across 100 physical machines in the Alibaba data center. All of the microservices had a replica near the dependent microservice (on a neighbouring PM). The neighbouring PM shares the same pod and uses the same switch, or is located in the same pod but uses a different switch Figure 7. Our strategy successfully reduces the cost of using the network significantly, enhancing the performance of the overall system.

For the second scenario, our strategy processes microservices in a real-time scenario. In total, our strategy schedules approximately 24,000 microservices to the data center in four-time intervals as follows: 8.5%, 62%, 8.6% and 0.5%. In addition, we consider that 20.4% of stateful microservices are already scheduled for the cloud. The algorithm reduces network traffic between all microservices by 39.7%, 31.6%, 22.9% and 31.1%. Our strategy replicates 19,209 microservices to their dependents, resulting in an average network traffic reduction of 31.32% over the four-time intervals. The new arrival of microservices in the cloud at different times contributes to a lower percentage of network traffic. Our strategy replicates the majority of microservices in the same pod as their dependents, with over 61% of the replicas, while just over 12% of the replicas share the same PMs. However, some replicas could not be placed close to their dependents due to reliability (fault tolerance) and resource utilization constraints as shown in Figure 8. As our strategy's objectives are to reduce single points of failure and balance resource utilization in each PM, we believe that our strategy achieves a significant reduction in network traffic.

B. LATENCY COST

Alibaba scheduled 11,377 microservices that depended on other microservices in different pods, which they considered to represent the different zones. In addition, Alibaba's approach distributed just 659 microservices in the same pod as their dependent microservices (in the same zone), and it scheduled just 5 microservices in the same PM as their dependent microservices, Figure 8.

In the first scenario of our experiment, our strategy replicates 1,695 microservices close to their dependent microservices and 8,606 microservices in neighbouring PMs (same pod). Our strategy achieved an 84% reduction in latency compared to the Alibaba scheduling approach. As a result, our strategy reduces the latency significantly while maintaining the availability of microservices and reducing the risk of single failure by replicating microservices in different PMs of their dependents but in the same pod (zone).

In the second scenario, over 19,206 microservices were scheduled. 18,221 of those microservices were replicated to the same PMs of their neighbours that are located in the same pods. As a result, our strategy reduces the latency to 94.9%. Alibaba's scheduling strategy produces around 84% of the total latency, whereas our strategy produces 14% and 2% of the total latency in the first and second scenarios, respectively as shown in Figure 8.

C. ENHANCING PERFORMANCE RESILIENCY

Our proposed scheduling strategy enhances the performance resiliency of the system through the maintenance of three main strategies. Firstly, we ensure the availability of the system by reducing the risk of a single point of failure. Secondly, we manage resource utilization by using a predefined threshold; therefore, the utilization of resources does not exceed it, preventing the violation of service level agreements (SLAs). Finally, we balance the load for all PMs.

We compare our proposed scheduling strategy against the two approaches explained in Section VI-E by comparing the results of resource utilization: CPU and memory.

In the first scenario, our scheduling strategy achieves a mean CPU utilization of 43.45 with a standard deviation of 12.04. This indicates that our scheduling strategy effectively balances the load of replicas among the PMs. In comparison, the Best Fit Algorithm and the adaptive threshold approach have higher standard deviations of 26.4 and 26.2, respectively, suggesting a more variable and potentially less reliable CPU resource allocation. By maintaining a lower standard

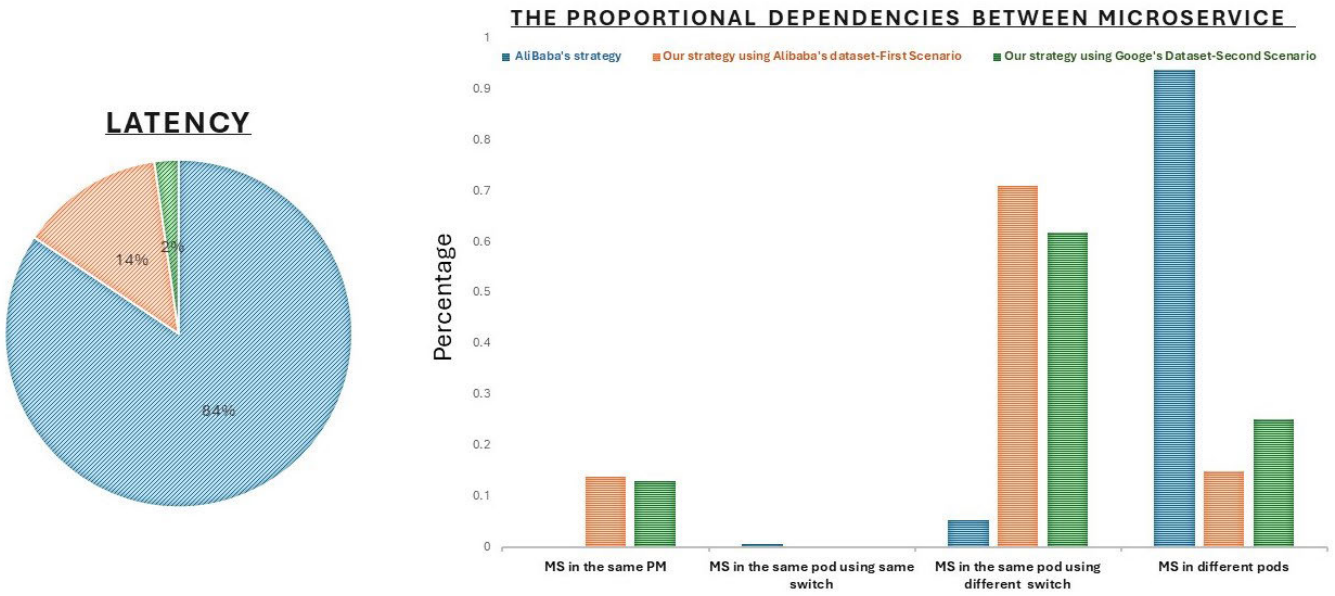


FIGURE 8. The number of dependencies between microservices in Alibaba's scheduling system, our proposed scheduling strategy using Alibaba's and Google's datasets, and the dependencies between microservices after applying our proposed scheduling strategy. The pie chart represents the proportional latency, while the bar plot represents the network traffic between microservices.

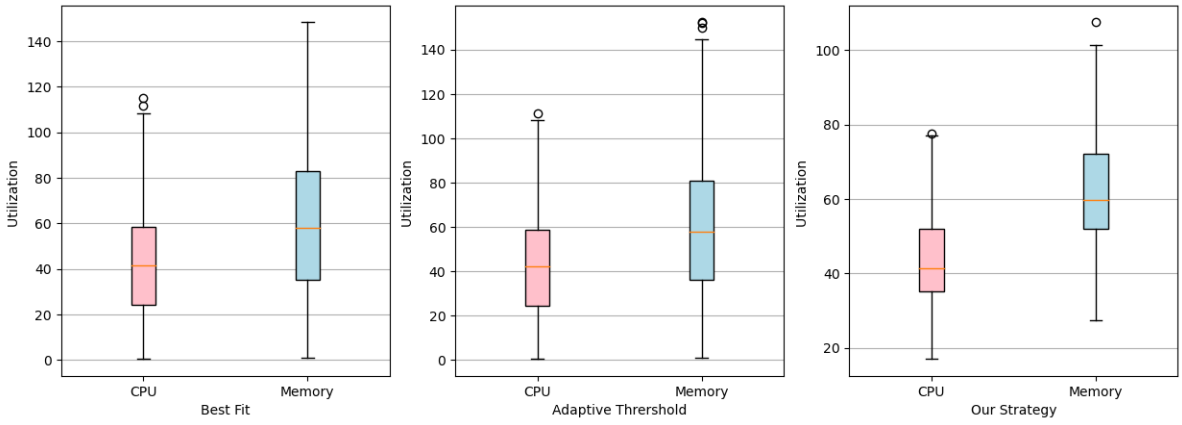


FIGURE 9. The three subplots showcase the resource utilization of three different approaches: the best-fit algorithm, the adaptive threshold approach, and our proposed scheduling strategy.

deviation, our algorithm minimizes the likelihood of resource overloads or bottlenecks, enhancing the overall efficiency and stability of the system [54].

In addition to the memory utilization in the first scenario, our proposed scheduling strategy achieves a mean memory utilization of 43.45 with a standard deviation of 16.5. This demonstrates that our strategy efficiently manages memory resources, ensuring optimal utilization without excessive waste or shortages of memory resources in PMs. In contrast, the Best Fit Algorithm and the adaptive threshold approach exhibit higher standard deviations of 34.82 and 35.24, respectively, indicating more significant variability in memory allocation. The lower standard deviation of our scheduling strategy contributes to more consistent and balanced memory utilization.Figure9 demonstrates how our strategy reduces the variation in both resource utilization. The figure illustrates that most PMs utilize CPU and memory around the mean. In contrast, the other algorithm shows that some PMs were not utilized while others were fully utilized. Our strategy minimizes the gap between the maximum and minimum resource utilization, with the minimum CPU utilization at just 20% and the highest utilization at less than 80%. This indicates that none of the PMs violated our QoS constraints.

Before delving into resource utilization in the second scenario, it's important to note the following:

1) Resource utilization in the second scenario is cumulative, meaning each PM hosts new microservices at any given time. Our objective is to measure the average utilization across all time intervals.

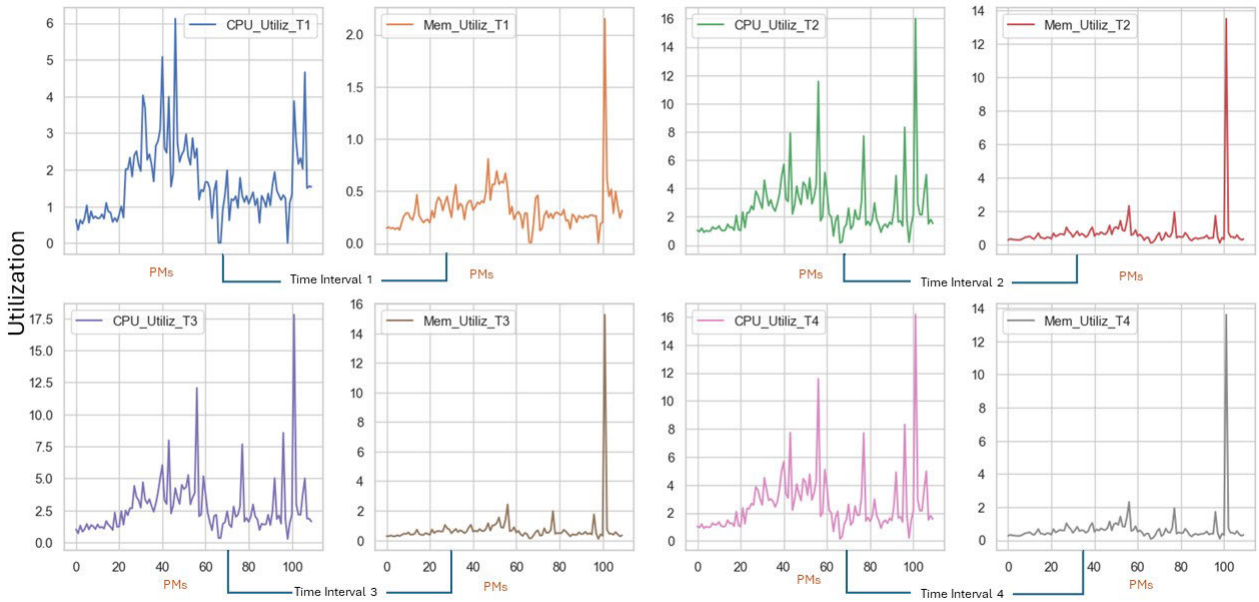


FIGURE 10. visualizes the utilization of CPU utilization and memory for 100 PMs that host microservices deployed using our proposed algorithm strategy. Each time interval (T1, T2, T3, T4) shows the variation in CPU and memory usage across the PMs.

2) In the Google Borg system, memory utilization is computed using bytes and normalized by dividing it by the maximum machine size in the cluster. Therefore, the largest memory size is represented as 1.
3) CPU utilization is measured in "Google compute units" (GCUs), calculated based on how much the workload of the unit needs to be processed. CPU utilization is measured in GCU seconds per second, where 1 GCU equals the total CPU needed for a workload's computation.

In our algorithm, we calculate the GCU units utilized during the experiment for each PM. Each PM has a capacity of 128 GCU and 512 GB of memory.

Figure 10 illustrates the resource utilization across each PM for both CPU and memory. It is evident from the figure that our algorithm significantly enhances resource utilization. Specifically, the mean CPU utilization increased from 0.78 GCU to 2.13 GCU, marking a 63% increase. Additionally, the mean memory utilization increased from 0.26 to 0.42, reflecting a 39% increase.

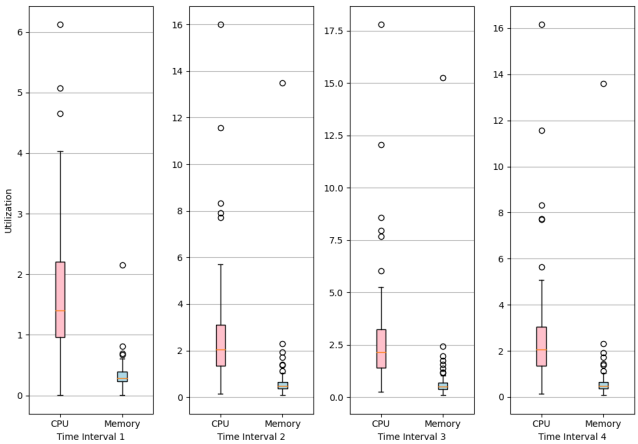


FIGURE 11. visualizes the scheduling of microservices using our proposed strategy, where microservices are replicated next to their neighbours. Each sub-graph represents time intervals, and the y-axis represents the mean of CPU and memory utilization across 100 PMs in each interval. A lower standard deviation indicates more balanced utilization across PMs.

D. COMPARISON WITH EXISTING STUDIES

Figure 11 shows the CPU utilization distribution across PMs for each time interval. In the first time interval, CPU utilization varied between 1 and 1.3. For the second, third, and fourth-time intervals, CPU utilization ranges were 1.25 and 2.8, 1.25 and 2.85, and 1.25 and 2.8, respectively. We found that most CPU utilization across most PM ranges from 1.25 to 2.8 units per CPU per workload. Furthermore, the memory utilization for most of the PMs was between 0.25 and 0.41, meaning that our strategy balances the load in most of the PMs.

Overall, our proposed scheduling strategy maintains consistent utilization of resources (CPU and memory) with the lowest SD for both resources. This indicates that our strategy manages the consistent allocation of CPU resources, which results in managing any sudden spikes or drops in resource usage. By avoiding significant fluctuations in resource usage, our strategy reduces the risk of resource-related performance issues and potential SLA violations. It ensures that the system has adequate resource availability to handle the workload. It helps to predict resource utilization and improve resource management and system performance.

TABLE 4. Comparison of our proposed scheduling strategy with existing strategies in literature.

Study	Resource Utilization		Load Balancing	Latency	Fault Tolerance	Use Graph Representation
	CPU	Memory				
[15]	×	×	×	×	×	Call Graph
[26]	×	×	×	×	×	DAG
[27]	×	×	×	×	×	DAG
[28]	×	×	×	×	×	DAG
[29]	×	×	×	×	×	DAG
[30]	×	×	×	×	×	DAG
[31]	×	×	×	×	×	DAG
[32]	24.4%	×	20%	99% tail latency	×	×
[33]	≃ 0.1%	≃ 0.1%	×	×	×	Graph Mapping Algorithm
[34]	8.64%	×	×	15.16%	×	×
[35]	×	×	×	×	×	×
Our Replication Strategy						
- *Using Alibaba's dataset*	**46%**	**45**	✓	**84%**	✓	**Directed Acyclic Graphs**
- *Using Google's dataset*	**63%**	**39%**	✓	**94.9%**	✓	

Moreover, our strategy outperforms the Best Fit Algorithm and the adaptive threshold approach in resource management by maintaining lower standard deviations for both CPU and memory. Our strategy ensures stable and efficient resource allocation, reducing the risk of bottlenecks and resource shortages. These benefits translate into improved system performance, enhanced reliability, and better utilization of resources.

It can be observed from Table 1 and Table 4 that our proposed scheduling strategy outperforms existing studies in different aspects that directly influence cloud performance. Most existing studies do not propose a multi-objective approach to improving cloud performance; thus, most of the results in the table 4 are left blank as they have not been reported in the research. For example, studies [15], [33], [34], and [35] do not consider load balance in their approaches. Additionally, none of them consider improving fault tolerance in their approaches. Therefore, the compared studies are limited in terms of addressing all aspects, such as resource management, load balancing, fault tolerance, and latency.

By replicating microservices close to their dependents, our proposed strategy optimizes the utilization of cloud resources. Moreover, our strategy also achieves load balancing and fault tolerance. Our strategy enhances resource utilization by 43.54% and balances the load by 46%. With the replication of all microservices, at least one for each, our algorithm outperforms existing strategies by enhancing fault tolerance. Additionally, our algorithm significantly reduces network traffic, leading to an 84% reduction in latency. This latency reduction results in improved response times, ultimately enhancing the overall performance of the cloud. In conclusion, these outcomes demonstrate the effectiveness and superiority of our strategy.

IX. CONCLUSION

Microservices architecture has emerged as a widely adopted approach for building applications, replacing the traditional monolithic architecture. It has changed how applications are developed by making them more accessible, scalable, and flexible. However, the dynamic nature of cloud computing poses challenges in effectively allocating, deploying, and managing microservices applications. Existing approaches have shown limitations in efficiently utilizing resources and meeting QoS requirements. To address these challenges, we propose a novel scheduling strategy that combines modified PSO and RR algorithm, integrated with Kubernetes as a container deployment manager.

Our novel proposed scheduling strategy empowers the PSO to initialize a swarm of particles, representing replicas of microservices. In addition, it initializes position populations representing the available resources of PMs, that particles swarm to optimize resource demands and viable solutions. Our algorithm proposes a fitness function that minimizes the distance between the particles and their best positions. When particles reach the best positions, the RR algorithm is used to distribute the particles into the PMs equally. The integration of modified PSO allows for exploration and exploitation of the search space to reduce network traffic between microservices, whereas the RR algorithm enhances load balancing. Our proposed strategy improves resource allocation, system performance, scalability, and availability.

During the scheduling of replicas of microservices, our novel scheduling strategy integrates with Kubernetes to manage the deployment, communication, and redirection of communication between replicas. Kubernetes provides robust health-checking mechanisms for containers and microservices. In addition, by monitoring the system through QoS management, our algorithm improves QoS.

In conclusion, our proposed scheduling strategy presents a novel and comprehensive approach for addressing the resource allocation and deployment of microservices in challenging and dynamic environments. It improves the overall performance of the microservice system and minimizes network traffic costs, resulting in reduced latency and improved system efficiency. For potential future research, optimizing the convergence of the combined PSO and RR algorithms to reach the optimum solution more efficiently and with reduced computational costs is suggested. In addition, machine learning algorithms like Q-learning need to be

investigated to schedule microservices to cloud resources utilizing microservice resource requirements.

REFERENCES

[1] Y. Gan and C. Delimitrou, "The architectural implications of cloud microservices," *IEEE Comput. Archit. Lett.*, vol. 17, no. 2, pp. 155–158, Jul. 2018.

[2] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li, and R. Ranjan, "A holistic evaluation of Docker containers for interfering microservices," in *Proc. IEEE Int. Conf. Services Comput. (SCC)*, Jul. 2018, pp. 33–40.

[3] J.-E. Dartois, J. Boukhobza, A. Knefati, and O. Barais, "Investigating machine learning algorithms for modeling SSD I/O performance for container-based virtualization," *IEEE Trans. Cloud Comput.*, vol. 9, no. 3, pp. 1103–1116, Jul. 2021.

[4] A. Abuabdo and Z. A. Al-Sharif, "Virtualization vs. containerization: Towards a multithreaded performance evaluation approach," in *Proc. IEEE/ACS 16th Int. Conf. Comput. Syst. Appl. (AICCSA)*, Nov. 2019, pp. 1–6.

[5] A. Saboor, A. K. Mahmood, A. H. Omar, M. F. Hassan, S. N. M. Shah, and A. Ahmadian, "Enabling rank-based distribution of microservices among containers for green cloud computing environment," *Peer-to-Peer Netw. Appl.*, vol. 15, no. 1, pp. 77–91, Jan. 2022.

[6] H. Zhang, S. Li, Z. Jia, C. Zhong, and C. Zhang, "Microservice architecture in reality: An industrial inquiry," in *Proc. IEEE Int. Conf. Softw. Archit. (ICSA)*, Mar. 2019, pp. 51–60.

[7] C. Santana, L. Andrade, F. C. Delicato, and C. Prazeres, "Increasing the availability of IoT applications with reactive microservices," *Service Oriented Comput. Appl.*, vol. 15, no. 2, pp. 109–126, Jun. 2021.

[8] S. Pallewatta, V. Kostakos, and R. Buyya, "Placement of microservices-based IoT applications in fog computing: A taxonomy and future directions," *ACM Comput. Surv.*, vol. 55, no. 14, pp. 1–43, Dec. 2023.

[9] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach," *IEEE Trans. Cloud Comput.*, vol. 10, no. 2, pp. 1100–1116, Apr. 2022.

[10] A. Kwan, J. Wong, H.-A. Jacobsen, and V. Muthusamy, "HyScale: Hybrid and network scaling of dockerized microservices in cloud data centres," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2019, pp. 80–90.

[11] M. Xu, L. Yang, Y. Wang, C. Gao, L. Wen, G. Xu, L. Zhang, K. Ye, and C. Xu, "Practice of Alibaba cloud on elastic resource provisioning for large-scale microservices cluster," *Softw., Pract. Exper.*, vol. 54, no. 1, pp. 39–57, Jan. 2024.

[12] S. Luo, H. Xu, K. Ye, G. Xu, L. Zhang, J. He, G. Yang, and C. Xu, "Erms: Efficient resource management for shared microservices with SLA guarantees," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, vol. 1, Dec. 2022, pp. 62–77.

[13] B. Cai, B. Wang, M. Yang, and Q. Guo, "AutoMan: Resource-efficient provisioning with tail latency guarantees for microservices," *Future Gener. Comput. Syst.*, vol. 143, pp. 61–75, Jun. 2023.

[14] X. Wan, X. Guan, T. Wang, G. Bai, and B.-Y. Choi, "Application deployment using microservice and Docker containers: Framework and optimization," *J. Netw. Comput. Appl.*, vol. 119, pp. 97–109, Oct. 2018.

[15] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proc. ACM Symp. Cloud Comput.*, Nov. 2021, pp. 412–426.

[16] J. Shah and D. Dubaria, "Building modern clouds: Using Docker, Kubernetes & Google cloud platform," in *Proc. IEEE 9th Annu. Comput. Commun. Workshop Conf. (CCWC)*, Jan. 2019, pp. 184–189.

[17] N. Singh, Y. Hamid, S. Juneja, G. Srivastava, G. Dhiman, T. R. Gadekallu, and M. A. Shah, "Load balancing and service discovery using Docker swarm for microservice based big data applications," *J. Cloud Comput.*, vol. 12, no. 1, pp. 1–9, Jan. 2023.

[18] P. Banerjee, S. Roy, A. Sinha, M. M. Hassan, S. Burje, A. Agrawal, A. K. Bairagi, S. Alshathri, and W. El-Shafai, "MTD-DHJS: Makespan-optimized task scheduling algorithm for cloud computing with dynamic computational time prediction," *IEEE Access*, vol. 11, pp. 105578–105618, 2023.

[19] S. A. Murad, Z. R. M. Azmi, A. J. M. Muzahid, M. K. B. Bhuiyan, M. Saib, N. Rahimi, N. J. Prottasha, and A. K. Bairagi, "SG-PBFS: Shortest gap-priority based fair scheduling technique for job scheduling in cloud environment," *Future Gener. Comput. Syst.*, vol. 150, pp. 232–242, Jan. 2024.

[20] S. A. Murad, Z. R. M. Azmi, A. J. M. Muzahid, M. M. H. Sarker, M. S. U. Miah, M. K. B. Bhuiyan, N. Rahimi, and A. K. Bairagi, "Priority based job scheduling technique that utilizes gaps to increase the efficiency of job distribution in cloud computing," *Sustain. Comput., Informat. Syst.*, vol. 41, Jan. 2024, Art. no. 100942.

[21] A. Verma and S. Kaushal, "A hybrid multi-objective particle swarm optimization for scientific workflow scheduling," *Parallel Comput.*, vol. 62, pp. 1–19, Feb. 2017.

[22] A. Zhou, B.-Y. Qu, H. Li, S.-Z. Zhao, P. N. Suganthan, and Q. Zhang, "Multiobjective evolutionary algorithms: A survey of the state of the art," *Swarm Evol. Comput.*, vol. 1, no. 1, pp. 32–49, Mar. 2011.

[23] R. Garg and A. K. Singh, "MultiObjective optimization to workflow grid scheduling using reference point based evolutionary algorithm," *Int. J. Comput. Appl.*, vol. 22, no. 6, pp. 44–49, May 2011.

[24] H. M. Fard, R. Prodan, J. J. D. Barrionuevo, and T. Fahringer, "A multi-objective approach for workflow scheduling in heterogeneous environments," in *Proc. 12th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGRID)*, May 2012, pp. 300–309.

[25] A. Dogan and F. Özgüner, "Biobjective scheduling algorithms for execution time-reliability trade-off in heterogeneous computing systems," *Comput. J.*, vol. 48, no. 3, pp. 300–314, Jan. 2005.

[26] S. Su, J. Li, Q. Huang, X. Huang, K. Shuang, and J. Wang, "Cost-efficient task scheduling for executing large programs in the cloud," *Parallel Comput.*, vol. 39, nos. 4–5, pp. 177–188, Apr. 2013.

[27] R. Garg and A. K. Singh, "Multi-objective workflow grid scheduling using ε-fuzzy dominance sort based discrete particle swarm optimization," *J. Supercomput.*, vol. 68, no. 2, pp. 709–732, May 2014.

[28] I. Casas, J. Taheri, R. Ranjan, and A. Y. Zomaya, "PSO-DS: A scheduling engine for scientific workflow managers," *J. Supercomput.*, vol. 73, no. 9, pp. 3924–3947, Sep. 2017.

[29] Y. Xie, Y. Zhu, Y. Wang, Y. Cheng, R. Xu, A. S. Sani, D. Yuan, and Y. Yang, "A novel directional and non-local-convergent particle swarm optimization based workflow scheduling in cloud-edge environment," *Future Gener. Comput. Syst.*, vol. 97, pp. 361–378, Aug. 2019.

[30] Y. Kothyari and A. Singh, "A multi-objective workflow scheduling algorithm for cloud environment," in *Proc. 3rd Int. Conf. Internet Things, Smart Innov. Usages (IoT-SIU)*, Feb. 2018, pp. 1–6.

[31] G.-S. Yao, Y.-S. Ding, and K.-R. Hao, "Multi-objective workflow scheduling in cloud system based on cooperative multi-swarm optimization algorithm," *J. Central South Univ.*, vol. 24, no. 5, pp. 1050–1062, May 2017.

[32] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo, "Adaptive resource efficient microservice deployment in cloud-edge continuum," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 8, pp. 1825–1840, Aug. 2022.

[33] X. Li, J. Zhou, X. Wei, D. Li, Z. Qian, J. Wu, X. Qin, and S. Lu, "Topology-aware scheduling framework for microservice applications in cloud," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 5, pp. 1635–1649, May 2023.

[34] B. Bao, H. Yang, Q. Yao, L. Guan, J. Zhang, and M. Cheriet, "Resource allocation with edge-cloud collaborative traffic prediction in integrated radio and optical networks," *IEEE Access*, vol. 11, pp. 7067–7077, 2023.

[35] A. Marchese and O. Tomarchio, "Communication aware scheduling of microservices-based applications on kubernetes clusters," in *Proc. 12th Int. Conf. Cloud Comput. Services Sci.*, 2022, pp. 190–198.

[36] Shutian. (2021). *Cluster-Trace-Microservices-V2021*. [Online]. Available: https://github.com/alibaba/clusterdata/blob/master/cluster-trace-microservices-v2021/README.md

[37] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: Format+ schema," Google, Menlo Park, CA, USA, White Paper Version 2.1, 2011, pp. 1–14.

[38] *Spring Cloud Config*, Alibaba Group, Hangzhou, China, Mar. 2021.

[39] A. R. Kashani, C. V. Camp, M. Rostamian, K. Azizi, and A. H. Gandomi, "Population-based optimization in structural engineering: A review," *Artif. Intell. Rev.*, vol. 55, no. 1, pp. 345–452, Jan. 2022.

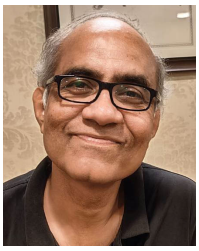
[40] A. Flori, H. Oulhadj, and P. Siarry, "QUAntum particle swarm optimization: An auto-adaptive PSO for local and global optimization," *Comput. Optim. Appl.*, vol. 82, no. 2, pp. 525–559, Jun. 2022.

[41] S. Sekigawa, C. Sasaki, and A. Tagami, "Toward a cloud-native telecom infrastructure: Analysis and evaluations of kubernetes networking," in *Proc. IEEE Globecom Workshops (GC Wkshps)*, Dec. 2022, pp. 838–843.

[42] R. P. Dhanya and V. S. Anitha, "Implementation and performance evaluation of load balanced routing in SDN based fat tree data center," in *Proc. 6th Int. Conf. Inf. Syst. Comput. Netw. (ISCON)*, Mar. 2023, pp. 1–6.

[43] M. Zhao, Z. Han, and X. Du, "A survey of data center network topology structure," in *Proc. 25th Int. Conf. Adv. Commun. Technol. (ICACT)*, Feb. 2023, pp. 303–309.
[44] P. Arabas, T. Jóźwik, and E. Niewiadomska-Szynkiewicz, "Router activation heuristics for energy-saving ECMP and valiant routing in data center networks," *Energies*, vol. 16, no. 10, p. 4136, May 2023.
[45] D. R. Mathews, M. Verma, J. Lakshmi, and P. Aggarwal, "Towards more effective and explainable fault management using cross-layer service topology," in *Proc. IEEE 15th Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2022, pp. 94–96.
[46] D. Wang, D. Tan, and L. Liu, "Particle swarm optimization algorithm: An overview," *Soft Comput.*, vol. 22, no. 2, pp. 387–408, Jan. 2018.
[47] T. Balharith and F. Alhaidari, "Round Robin scheduling algorithm in CPU and cloud computing: A review," in *Proc. 2nd Int. Conf. Comput. Appl. Inf. Secur. (ICCAIS)*, May 2019, pp. 1–7.
[48] A. Alelyani, "Microservice projects GitHub repository," abo2512d1_ repository, GitHub, Tech. Rep. ff66426, 2023.
[49] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. 10th Eur. Conf. Comput. Syst.*, Apr. 2015, pp. 1–17.
[50] R. C. Eberhart and Y. Shi, "Tracking and optimizing dynamic systems with particle swarms," in *Proc. Congr. Evol. Comput.*, vol. 1, Feb. 2001, pp. 94–100.
[51] V. Truong Vu, "A comparison of particle swarm optimization and differential evolution," *Int. J. Soft Comput.*, vol. 3, no. 3, pp. 13–30, Aug. 2012.
[52] H. Wang, L. L. Zuo, J. Liu, W. J. Yi, and B. Niu, "Ensemble particle swarm optimization and differential evolution with alternative mutation method," *Natural Comput.*, vol. 19, no. 4, pp. 699–712, Dec. 2020.
[53] A. Clouder, "Alibaba cloud CDN and low-latency global cloud solutions," Alibaba Cloud Community, Alibaba Cloud, Tech. Rep. 596309, 2021.
[54] S. C. H. Lu, D. Ramaswamy, and P. R. Kumar, "Efficient scheduling policies to reduce mean and variance of cycle-time in semiconductor manufacturing plants," *IEEE Trans. Semicond. Manuf.*, vol. 7, no. 3, pp. 374–388, Aug. 1994.

ABDULLAH ALELYANI received the B.Sc. degree in computer science from King Abdulaziz University, Saudi Arabia, and the M.S. degree in software system engineering from The University of Melbourne, Australia. He is currently pursuing the Ph.D. degree with The University of Western Australia. He is a dedicated Academic Researcher specializing in computer science and software system engineering. His commitment to sustainable technology and innovative approaches has the potential to revolutionize data center efficiency and advance the field of cloud computing. His research interest includes reducing energy consumption in data centers, particularly in cloud computing. His academic interests also extend to cloud computing, deep learning, and mobile cloud offloading.



AMITAVA DATTA is currently a Professor of computer science with more than 25 years of experience. He was a Visiting Professor with the University of Freiburg, Germany, Warsaw University of Life Sciences, Poland, and Southwest University, China. He designed and developed the first OLAP engine on GPUs along with colleagues from Jedox AG, Germany. He has also developed a prototype technical risk aggregation information system for the Royal Australian Navy. He has taught most bachelor's and master's students in computer science and information technology. He has published more than 160 research papers in reputed international journals and conference proceedings and supervised more than 21 Ph.D. students to completion. His research interests include the design and analysis of algorithms, computational molecular biology, applications of machine learning, unsupervised learning (in particular subspace clustering), high-performance computing (parallel and distributed systems and GPUs), mobile and wireless networks, social network analysis, sociophysics, and quantum computing.

GHULAM MUBASHAR HASSAN (Senior Member, IEEE) received the B.S. degree from the University of Engineering and Technology, Peshawar, Pakistan, the M.S. degree from Oklahoma State University, Stillwater, OK, USA, and the Ph.D. degree from The University of Western Australia (UWA). He is currently a Faculty Member with the Department of Computer Science and Software Engineering, UWA. His research interests include artificial intelligence, machine learning, and their applications in multidisciplinary problems. He was a recipient of multiple teaching excellence and research awards.