IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 21 December 2023, accepted 4 January 2024, date of publication 9 January 2024,
date of current version 18 January 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3351944

RESEARCH ARTICLE

Optimal Resource Allocation Using Genetic Algorithm in Container-Based Heterogeneous Cloud

QI-HONG CHEN[1] AND CHIH-YU WEN[1,2], (Senior Member, IEEE)
[1]Department of Electrical Engineering, National Chung Hsing University, Taichung 40227, Taiwan
[2]Smart Sustainable New Agriculture Research Center (SMARTer), National Chung Hsing University, Taichung 40227, Taiwan

Corresponding author: Chih-Yu Wen (cwen@dragon.nchu.edu.tw)

This work was supported by the Smart Sustainable New Agriculture Research Center (SMARTer), National Science and Technology Council, Taiwan, under Grant 111-2634-F-005-001.

ABSTRACT This paper tackles the complex problem of optimizing resource configuration for microservice management in heterogeneous cloud environments. To address this challenge, an enhanced framework, the multi-objective microservice allocation (MOMA) algorithm, is developed to formulate the efficient resource management of cloud microservice resources as a constrained optimization problem, guided by resource utilization and network communication overhead, which are two important factors in microservice resource allocation. The proposed framework simplifies the deployment of cloud services and streamlines workload monitoring and analysis within a diverse cloud system. A comprehensive comparison is made between the effectiveness of the proposed algorithm and existing algorithms on real-world datasets, with a focus on resource balancing, network overhead, and network reliability. Experimental results reveal that the proposed algorithm significantly enhances resource utilization, reduces network transmission overhead, and improves reliability.

INDEX TERMS Resource allocation, genetic algorithm, container-based heterogeneous cloud, multi-objective optimization, microservice.

I. INTRODUCTION

In recent years, with the rise of microservices architecture for breaking large-scale applications down into smaller independent components, microservice applications invoke numerous internal microservices to construct responses. For instance, a container is a typical example that meets the requirements of a microservices architecture. By using containers, developers can focus on service development via operating system virtualization. Since docker is one of the most successful container frameworks [1], providing independent execution environments with isolated file systems, portability, and superior resource utilization compared to virtual machines [2], it has become an important technology in current microservices. Examples of container orchestration platforms that offer automated deployment include Docker Swarm, Apache Mesos, and Google Kubernetes [3].

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

Despite the rapid technological development of microservices architecture, there are still many tasks to be tackled. For example, the default resource allocation method in Kubernetes only aims at physical resource utilization [4] and does not address the costs and reliability of network transmission. Moreover, reliability is a critical issue in cloud service environments, which has been specifically addressed in [5] and [6]. Given that the approaches of the existing works mainly operate across homogeneous clouds, handling resource heterogeneity within multi-cluster environments may pose even more problems. Accordingly, in view of the characteristics of microservices, monitoring all service components and their interactions can be complex. Monitoring metrices, resource metrics (e.g., CPU and memory utilization) and platform metrics (e.g., number of requests per second, distribution of time required for each request, and average execution time for the queries), may be built for individual services, which provide visibility into the distributed system for evaluating the application's

performance [7]. In this work, we use resource utilization as the optimization goal. Thus, resource metrics, CPU utilization and memory consumption, are applied for effectively managing the resources and further enhancing both the performance and service reliability.

With recognized as an NP-hard problem for container resource allocation [8], finding polynomial-time complexity algorithms remains an open issue. Many researchers turn to meta-heuristic algorithms to obtain optimal solutions for these resource allocation problems. In various heuristic algorithms, each possesses its own set of strengths and weaknesses. By employing contextual analysis, we can determine the most suitable algorithm for a given scenario [9]. For instance, [10] compares different algorithms to propose the most fitting one for a specific context and acquires results through iterative experimentation. Genetic algorithms (GA) are considered effective in addressing such problems [11], and Non-Dominated Sorting Genetic Algorithm II (NSGA-II) is one of the most widely used genetic algorithms [12]. Accordingly, further research is needed to address these concerns and advance the field.

In this study, an elitism-based genetic algorithm, the multi-objective microservice allocation (MOMA) algorithm, is developed and utilized to determine the optimal placement of microservices within the cluster, taking into account the current state of the cluster and the microservices themselves, where a cluster is a group of servers or nodes, which participate in workload management. The proposed framework aims to facilitate the placement of workloads into the Kubernetes cluster, which may consist of a PC and physical computer systems (e.g., Raspberry Pi and an NVIDIA Jetson Nano). This operation involves considering factors such as resource balancing, inter-dependencies among microservices, network characteristics, and performance requirements. By analyzing these factors, the system can devise an effective distribution strategy that ensures efficient resource utilization for the microservices. Thus, the goal is to find the best possible arrangement that maximizes the overall system performance and minimizes any potential bottlenecks or resource constraints.

The main contributions and features of this study are as follows:

1) This work addresses the heterogeneity challenge of microservice resource allocation by developing an enhanced GA algorithm with two objective models, considering resource utilization and network communication overhead, and the empirical parameter settings via real-world data, for optimizing resource management in heterogeneous cloud environments.
2) The proposed framework tackles the heterogeneity aspects of cluster monitoring and resource selection, simplifies the deployment of cloud services, and streamlines workload management and analysis within a diverse cloud system.
3) A comprehensive evaluation of the proposed framework is presented on real-world datasets. The experimental results show that the proposed framework outperforms existing methods in terms of resource balancing, network overhead, and network reliability.

The organization of this paper is as follows: Section II reviews related works about cluster resource allocation and multi-objective optimization. Section III presents the proposed system architecture and workload analysis framework. Section IV describes a customized multi-objective optimization model for evaluating the heterogeneous system performance. Section V examines the framework characteristics and presents a performance comparison between the proposed scheme and the existing works. Finally, Section VIII draws conclusions and outlines future research directions.

II. RELATED WORKS

This section reviews related works about cluster resource allocation and multi-objective optimization in various cloud environments.

A. RESOURCE ALLOCATION

Resource management has always been a critical issue in cloud computing [13]. In the literature, numerous issues have been discussed for solving resource management problems (e.g., scheduling approaches [14] and allocation strategies [15]). For instance, cluster-based resource management schemes, which improve Kubernete algorithms [16], [17], [18], [19], [20], and resource allocation algorithms based on multi-objective evolutionary algorithms (MOEAs), such as particle swarm optimization (PSO) [21], simulated annealing (SA) [22], ant colony optimization (ACO) [23], and GA algorithm [24], [25], [26]. Note that the Elitist non-dominated Sorting Genetic Algorithm II (NSGA-II) [27] is one of the most widely applied MOEAs in this context. The interested reader is referred to [28] and [29] for comprehensive surveys of metaheuristic optimization algorithms.

B. CLOUD ENVIRONMENTS

1) SINGLE-CLOUD SCENARIO

A large portion of literature focuses on issues related to single-cloud scenarios. Fu et al. [21] use a PSO algorithm to allocate resources and improve efficiency in a single-cloud environment. Abdallah et al. [22] utilize SA algorithm and tabu search to emphasize fair allocation procedures of multiple resource types. Liu et al. [30] propose a multi-objective optimization container scheduling algorithm that considers five criteria to select the most suitable node for deployment. Kaewkasi et al. [31] develop a new Docker scheduler and use the ACO algorithm to balance resources. Gupta et al. [32] implement enhanced algorithms (e.g., Max-Min and Greedy) for load balancing in cloud environments. Qiu et al. [33], Guo et al. [34], Li [35], and Ali [36] adopt machine learning models (e.g., reinforcement learning and transfer learning) as the main algorithms for service deployment, microservice selection, optimizing the delay, and reducing deployment cost with a fixed service set. However, due to the issue

TABLE 1. Summary of resource management.

Reference	Architecture	Management	Objectives	Algorithm/Platform
[21]	Single cloud	Allocation	Resource	PSO
[22]	Single cloud	Allocation	Energy consumption	SA,tabu search
[31]	Single cloud	Scheduling	Performance	ACO
[30]	Single cloud	Scheduling	Performance, association, clustering	Own algorithm
[32]	Single cloud	Allocation	Resource	Max-Min, Greedy
[33]	Single cloud	Allocation	Resource	Machine learning
[34]	Single cloud	Scheduling	Microservice selection	Machine learning
[35]	Single cloud	Service deployment	Delay and deployment cost	Machine learning
[24]	Multiple clouds	Allocation	Service cost, latency, availability	NSGA-II
[25]	Multiple clouds	Allocation	Load balancing	NSGA-II
[26]	Multiple clouds	Allocation	Availability, energy consumption	NSGA-II
[37]	Multiple clouds	Placement	CPU performance, microservice interaction	Greedy algorithm
[38]	Multiple clouds	Scheduling	Resource, availability, costs	GA
[39]	Multiple clouds	Traffic	Availability, reliability	Grafana and Prometheus
[40]	Multiple clouds	Allocation	Resource, cost	Own algorithm
[41]	Heterogeneous cloud	Scheduling	Performance, energy consumption	Own algorithm
[42]	Heterogeneous cloud	Scheduling	Execution times, costs	NSGA-II
[43]	Heterogeneous cloud	Allocation	Resource	Own framework
This work	Multiple heterogeneous clouds	Microservices placement, resource allocation	Resource utilization, reliability and overhead	Elitist NSGA-II

of model retraining, theses algorithms may not be suitable for a microservices system with new services within short execution time.

2) MULTI-CLOUD SCENARIO

In the context of multi-cloud environments, [24], [25], [26] employ NSGA-II to address application availability and energy consumption requirements in container-based clouds. Han et al. [37] propose a Greedy algorithm for optimizing microservice placement across multiple Kubernetes clusters. They also introduce an empirical analysis framework to provide systematic and reliable measurement data. Frincu et al. [38] utilize a GA algorithm to achieve high availability and fault tolerance for applications. In [39], the monitoring of multi-cloud services is discussed and implemented via Prometheus and Grafana. Moreover, Lee et al. [40] propose a hierarchical monitoring framework for multi-cloud environments that takes workloads into account but does not specifically address heterogeneity.

3) HETEROGENEOUS CLOUD SCENARIO

Instead of considering single and multi-cloud scenarios, Rocha et al. [41] address the importance of heterogeneous clusters in the cloud and propose an algorithm for accessing heterogeneous resources, which significantly reduces energy consumption and runtime. Ali et al. [42] present an improved NSGA-II algorithm for minimizing range and total cost in heterogeneous environments. Hasan [43] introduces a resource monitoring framework for heterogeneous clusters without detailed consideration of workloads.

In this work, we further extend the scope of the resource management in a scenario of multiple heterogeneous clouds, focusing on microservice placement and resource allocation. We summarize these findings in Table 1, including different cloud architectures, resource management approaches, objective models, and corresponding algorithms. Here we examine the problem background and the contributions of the proposed framework from two different perspectives.

From the scenario management perspective, in the single-cloud scenario, the latest studies (e.g., [21], [22], [30], [31], [32], [33], [34], [35]) on allocation and scheduling management emphasize the importance of resource,energy consumption, clustering, microservice selection, delay, and deployment cost. In the multi-cloud scenario, several studies (e.g., [26], [37], [38]) develop analytical architectures, considering placement, allocation and scheduling management with respect to CPU performance, energy consumption, microservie interaction, and deployment cost. Moreover, in the heterogeneous cloud scenario, Rocha et al. [41] and Ali et al. [42] focus on scheduling management with optimization objectives of deployment cost, execution time, and energy consumption. However, none of the mentioned algorithms have specifically addressed the challenges of heterogeneous and multi-cloud environments.

From the algorithm perspective, the default algorithm in Kubernetes considers too few factors, focusing solely on resource allocation balance without applying optimization strategies. The Greedy algorithm (e.g., [37]) is applied to simplify calculations and achieve a local optimal solution, which may fail to deal with the multifaceted considerations we aim for in multi-cluster algorithms in the cloud. Moreover, the existing GA-based algorithms (e.g., [24], [25], [26], [41], [42]) only focus on resource management in multiple clouds or heterogeneous clusters in the cloud.

Therefore, upon addressing the problems of lacking in heterogeneous requirements and optimization strategies for container management, the proposed enhanced framework allows for a broader exploration of resource management considerations. Thus, this study focuses on addressing this gap and proposes a microservice placement and workflow scheduling approach along with resource allocation strategies tailored for heterogeneous environments. The proposed

approach is further analyzed based on two pivotal factors: resource utilization and network communication overhead, for tackling these aspects of heterogeneity in cluster monitoring and resource selection.

III. FRAMEWORK

A. SYSTEM MODEL

Referring to the microservice placement framework in [37], we propose a novel system framework. Derived from empirical analysis, the proposed framework provides several improvements with respect to throughput, latency, and distribution strategies for microservices, which are depicted in Figure 1. The proposed framework is divided into four main components: the Monitoring Unit, the Data Analysis Unit, the Optimization Algorithm Placement Unit, and the Kubernetes Management Unit. Through the interactions of these units, the framework facilitates the placement of workloads into the Kubernetes cluster. The four components are described as follows:

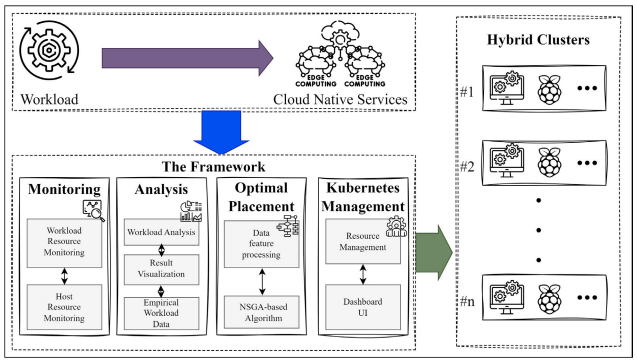


FIGURE 1. Architecture diagram of the empirical analysis framework.

1) MONITORING UNIT

It monitors the resource usage of the cluster, keeping track of metrics like CPU utilization and memory consumption. This information helps in managing and optimizing resource allocation. Moreover, the system collects performance data of microservices, including metrics such as latency and throughput, which enables performance evaluation and identification of bottlenecks for further optimization. By monitoring these aspects, the system helps maintain the stability, performance, and overall health of the cluster environment.

2) DATA ANALYSIS UNIT

The collected monitoring data undergoes comprehensive analysis to evaluate the state of the cluster and assess the performance of microservices. This analysis involves examining various metrics, such as resource utilization, response time, and throughput. By analyzing this data, valuable insights can be gained regarding the efficiency and effectiveness of the cluster and its microservices. The analyzed data is then stored for further use by other components or units within the system, enabling informed decision-making, optimization of resource allocation, and performance enhancements.

3) PLACEMENT OPTIMIZATION UNIT

The optimization algorithms are utilized to determine the optimal placement of microservices within the cluster, taking into account the current state of the cluster and the microservices themselves. This involves considering factors such as resource balancing, inter-dependencies among microservices, and performance requirements. By analyzing these factors, the system can devise an effective distribution strategy that ensures efficient resource utilization for the microservices. The goal is to find the best possible arrangement that maximizes the overall system performance and minimizes any potential bottlenecks or resource constraints.

4) KUBERNETES MANAGEMENT UNIT

The resource management system interacts with the cluster through the Kubernetes API, enabling it to perform various tasks. By leveraging the Kubernetes infrastructure, the system ensures efficient deployment of the workload by assigning the microservices to the appropriate nodes within the cluster.

B. FRAMEWORK WORKFLOW

Figure 2 illustrates the overall workflow of the three-stage framework and explains how it communicates with users and the cluster. In Stage one, the user sends a request that is received by the Kubernetes Management Unit within the framework, where the Kubernetes Management Unit is responsible for deploying the application to the selected cluster. Next, the Monitoring Unit is utilized to monitor the workload and gather information about resource utilization and microservices performance within the cluster. The collected data are then stored using persistent volume.

In Stage two, the stored measurement data are passed to the Data Analysis Unit for analyzing the captured values and deriving stable workload results, which are then stored in the Analysis Database. This procedure is repeated for each microservice within the application, ensuring completion for all microservices.

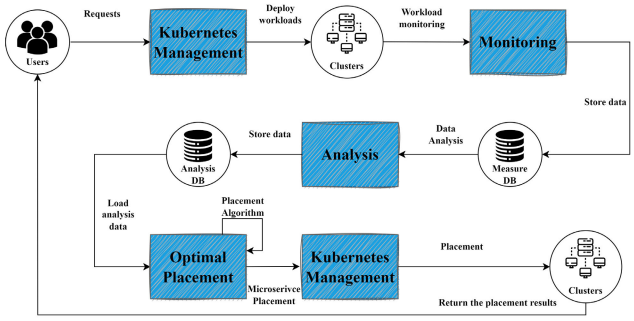


FIGURE 2. Information flowchart of the workflow process.

In Stage three, after organizing the analyzed data in the database, they are passed to the Placement Optimization Unit, which executes the designed algorithm for determining an approximate optimal placement strategy. The results of the algorithm execution provide insights into the placement of microservices. Subsequently, the Kubernetes Management

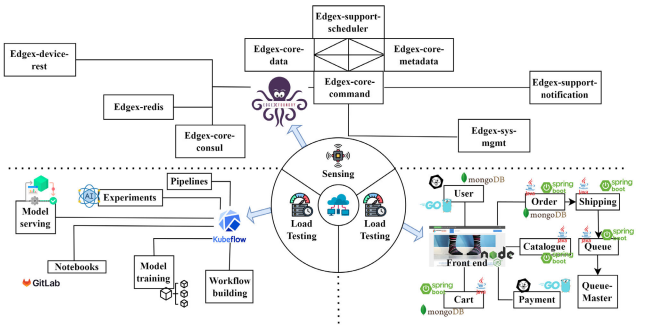


FIGURE 3. Heterogeneous Cloud applications.

Unit is used to deploy the microservices onto the cluster. Finally, the results can be applied for strengthening the monitoring and evaluation of microservice management.

C. APPLICATION TYPE

Since cloud-native technologies empower organizations to build and execute scalable applications in a modern, dynamic environment, including public, private, and hybrid clouds, in this work, we consider a cloud service that can operate for both edges and clouds. We establish three cloud environment services to represent the heterogeneous environment, as illustrated in Figure 3.

1) KUBEFLOW APPLICATION

Kubeflow is a model development platform, built on top of Kubernetes, which provides all the necessary tools for developing models and leverages Kubernetes to achieve flexible control over resources and networking. During the execution of the sample program, we utilize the Chicago Taxi Trips dataset, which is included in Kubeflow's built-in test dataset. We adopt the Xgboost demo example in Kubeflow for training. The process involves various units such as data preprocessing, model training, prediction, data normalization, and data validation.

2) SOCK SHOP APPLICATION

The application is a well known microservices application, widely used in demonstration and testing of microservice environments such as Kubernetes. It is built using Spring Boot, Go kit and Node.js and is packaged in Docker containers. We use Locust to conduct HTTP workload testing and simulate the performance of the store application under real-world usage scenarios.

3) EDGEX FOUNDRY APPLICATION

EdgeX is used to provide an open-source platform for industrial-grade edge computing in the Internet of Things (IoT) domain. We use Raspberry Pi 4 as the edge device, combined with DHT-11 sensor, to collect temperature and humidity data as an example for IoT services.

The three mentioned applications serve as deployable applications in a heterogeneous hybrid cloud environment, where these components are often used as reference points for testing. By integrating the proposed framework with these applications, it opens up greater possibilities for future adoption and promotion of cloud-native services, which allows us to make significant advancements in utilizing and promoting cloud-native services.

TABLE 2. Notations and Definitions for the Problem Model.

Element	Notation	Description
Micro-services	$Microservices$	The set of microservices.
	$\|Microservices\| = m$	The total number of microservices comprising the application.
	$ms_i \in Microservices$	Representing the ith microservice.
Cluster	$Cluster^{(i)}$	The collection of clusters of the ith microservice.
	$\|Cluster^{(i)}\| = c_i$	The number of Kubernetes clusters of the ith microservice.
	$cn_\ell^{(i)} \in Cluster^{(i)}$	Representing the ℓth cluster for the ith microservice.
Node	$Host_\ell$	The collection of physical nodes in the ℓth cluster.
	$\|Host_\ell\| = n_\ell$	The total number of physical nodes in the ℓth cluster.
	$pn_j^{(\ell)} \in Host_\ell$	Representing the jth physical node of the ℓth cluster.
Resource	cpu_{req_i}	The CPU resource requirement of the ith microservice.
	mem_{req_i}	The Memory resource requirement of the ith microservice.
	$cpu_{res_{\ell j}}$	The amount of available computing capacity on the jth node in the ℓth cluster.
	$mem_{res_{\ell j}}$	The amount of available memory capacity on the jth node in the ℓth cluster.
Network	$Fail_{\ell j}^{(i)}$	The probability or frequency at which the node (e.g., the jth node in the ℓth) may experience a failure or become unavailable by the ith microservice.
	$Distance^{(i)}_{(pn_j^{(\ell)}, pn_k^{(\ell)})}$	The network distance between two nodes (e.g., the ith node and the kth node in the ℓth cluster) by the ith microservice.
	$Interaction^{(i)}_{(pn_j^{(\ell)}, pn_k^{(\ell)})}$	The total volume of data transmission for sending and receiving operations between two nodes (e.g., the ith node and the kth node in the ℓth cluster) by the ith microservice.

IV. THE OPTIMIZATION MODEL

This section provides an overview of the optimization model, integrating the objectives of establishing load-balancing cluster environments and reducing network transmission overhead for reliable microservice communication specified by the problem model. The notations and descriptions are summarized in Table 2.

A. PROBLEM MODEL

1) OBJECTIVE 1: MAXIMUM RESOURCE UTILIZATION

The problem model aims to balance the resources in multiple clusters, which is referred to as the multi-resource load balancing problem. To tackle this, we adopt the "server's dominant load" method to maximize the load of all resource types [44]. To avoid significant disparities, the proportional

values between loads in each cluster and its associated nodes are calculated and normalized with the standard deviations $\sigma_{\ell,1}$ and $\sigma_{\ell,2}$ as scalar factors for the memory and CPU resources of cluster ℓ, respectively. Comparing to the similar model in [45], the proposed load balancing model achieves a more even distribution of resources in heterogeneous multi-cluster systems, which yields

$$CLU\_BAL = \frac{1}{\sigma_{\ell,1}+\sigma_{\ell,2}} \times \max_{\substack{1\le\ell\le c_i \\ 1\le j\le n_l}} \left(\sum_{i=1}^{m} \sigma_{\ell,1} \frac{mem_{req_i}}{mem_{res_{\ell_j}}} + \sum_{i=1}^{m} \sigma_{\ell,2} \frac{cpu_{req_i}}{cpu_{res_{\ell_j}}} \right), \quad (1)$$

where i is the microservice index, ℓ is the cluster index, j is the node index, c_i represents the number of Kubernetes clusters of the ith microservice, n_ℓ represents the total number of physical nodes in the ℓth cluster, and m is the total number of microservices comprising the application. Note that mem_{req_i}, $mem_{res_{\ell_j}}$, cpu_{req_i}, and $cpu_{res_{\ell_j}}$ are resource elements as described in Table 2.

2) OBJECTIVE 2: REDUCING COMMUNICATION OVERHEADS

To improve data availability in edge computing, the importance of retransmission mechanisms in the context of IoT and edge computing is emphasized [46], [47]. Therefore, we take this aspect into account and design a model that focuses on the impact of retransmission mechanism in heterogeneous clouds, which is given by

$$REL\_NET\_OVH = \sum_{i=1}^{m} \sum_{l=1}^{c_i} \sum_{j=1}^{n_\ell} \sum_{k=1 \bigwedge k\neq j}^{n_\ell} \times (1+Fail_{\ell j}^{(i)}) * Distance^{(i)}_{(pn_j^{(\ell)}, pn_k^{(\ell)})} * Interaction^{(i)}_{(pn_j^{(\ell)}, pn_k^{(\ell)})}, \quad (2)$$

where $Fail_{\ell j}^{(i)}$, $Distance^{(i)}_{(pn_j^{(\ell)}, pn_k^{(\ell)})}$, and $Interaction^{(i)}_{(pn_j^{(\ell)}, pn_k^{(\ell)})}$ are network elements as described in Table 2. Here *Fail* represents the probability or frequency at which the node may experience a failure or become unavailable by the microservice. *Distance* represents the network distance between two nodes by the microservice. *Interaction* represents the total volume of data transmission for sending and receiving operations between two nodes by the microservice.

B. MULTI-OBJECTIVE MICROSERVICE ALLOCATION MODEL

Based on the aforementioned problem model, an allocation model aiming at optimizing two objectives is designed to fulfill the requirements through the following constraints.

$$minimize \quad CLU\_BAL, \quad (3)$$

$$minimize \quad REL\_NET\_OVH, \quad (4)$$

$$s.t. \quad \sum_{i=1}^{m} \sum_{\ell=1}^{c_i} \sum_{j=1}^{n_\ell} mem_{req_i} < mem_{res_{\ell j}}, \quad \forall pn_j^{(\ell)}, \quad (5)$$

$$\sum_{i=1}^{m} \sum_{\ell=1}^{c_i} \sum_{j=1}^{n_\ell} cpu_{req_i} < cpu_{res_{\ell j}}, \quad \forall pn_j^{(\ell)}. \quad (6)$$

Equations (3) and (4) represent two optimization objectives: minimizing the maximum resource utilization on multiple cluster physical nodes and minimizing network transmission overhead for reliable microservice communication. Note that equations (5) and (6) represent the constraints of the model, which ensure that the resources of the microservices can be fully allocated on a single node. Therefore, constraints are imposed specifically for this aspect.

As we know, multiple-objective problems are difficult to find exact solutions and often require searching for optimal approximate solutions. With the parallel computing capability and scalability for efficiently solving complex and challenging problems, GA algorithms have been widely used. However, GA algorithms may suffer from the issue of easily getting trapped in local optima. Therefore, based on the NSGA-II algorithm, we enhance its adaptability for the domain-specific problem via the proposed multi-objective microservice allocation (MOMA) algorithm, as detailed in Section V.

V. MOMA ALGORITHM DESIGN

This section explains the design principles of the proposed MOMA algorithm, which aims to provide better resource allocation for a container-based heterogeneous cloud. The proposed MOMA algorithm defines chromosome representation, usage of crossover operators, mutation operator methods, parameter settings, and algorithm flow. Since the quality of a GA algorithm is greatly influenced by the definition of each component, in the following subsections, the overall algorithm structure is described and the operation procedures are summarized in Algorithm 1.

A. REPRESENTATION

When using a GA algorithm to solve a problem, it is essential to analyze the problem for determining the decision variables (i.e., the genes) [48]. After encoding the genes through a series of processes, we refer to them as chromosomes, which typically consist of individual genes. To manipulate and optimize the chromosomes, the binary encoding scheme [49] is used to ultimately find the optimal or near-optimal solutions. Thus, we define a microservice list based on different allocations, which represents the assignment of containers to various workers in a cluster with implementing microservices. The workers consist of a heterogeneous combination of general-purpose computers and edge devices. Figure 4 shows a typical run of a microservice list based on different allocations, which represents the assignment of containers to various workers in a cluster with implementing microservices. The workers consist of a heterogeneous

combination of general-purpose computers and edge devices. Note that ms_i represents the ith microservice and 1-2 represents performing the microservice via node 2 in cluster 1.

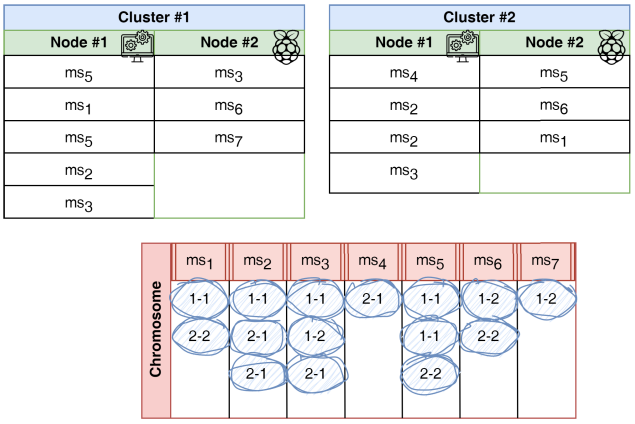


FIGURE 4. An example of chromosome representation.

B. CROSSOVER

During the crossover process, genes are randomly paired from replicated genetic material with the pairing methods, including the single-point crossover, two-point crossover, uniform crossover, mask-arithmetical crossover, and simulated binary crossover (SBX) [50], [51]. However, not every individual is required to mate in each generation, which leads to the introduction of a crossover probability, $Prob_{crossover}$.

The SBX operator primarily aims to emulate the characteristics of single-point crossover in binary-encoded chromosomes. When applying the SBX, assuming two parent individuals (P_1 and P_2), two offspring individuals (Q_1 and Q_2) can be generated using the SBX operator, which are

$$\begin{aligned} Q_1 &= 0.5((P_1 + P_2) - \beta(P_2 - P_1)), \\ Q_2 &= 0.5((P_1 + P_2) + \beta(P_2 - P_1)), \end{aligned} \tag{7}$$

where β depends on the random number u, as shown by equation (8)

$$\beta = \begin{cases} (2u)^{\frac{1}{\eta+1}} & \text{if } u \le Prob_{crossover} \\ \left(\dfrac{1}{2(1-u)}\right)^{\frac{1}{\eta+1}} & \text{otherwise,} \end{cases} \tag{8}$$

u is a random number between 0 and 1, and η is a constant representing the distribution index. The value of η is set to a commonly used value of 10. When η has a larger value, the offspring will be more inclined to resemble their parents. Given two parent individuals (P_1 and P_2) and referring to the SBX operator described in equations (7) and (8), Figure 5 shows the generation process of two offspring individuals (Q_1 and Q_2).

C. MUTATION

Mutation is a method of changing the genetic genes of offspring with a certain probability to prevent from falling into the local optimal solution and to maintain genetic diversity. Referring to [48], the displacement-based operators, the insertion and deletion operators, are applied on newly generated individuals.

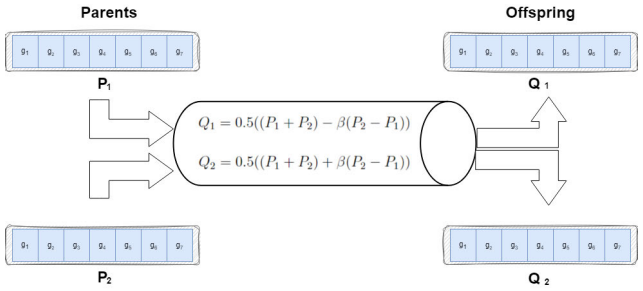


FIGURE 5. An example of SBXcrossover.

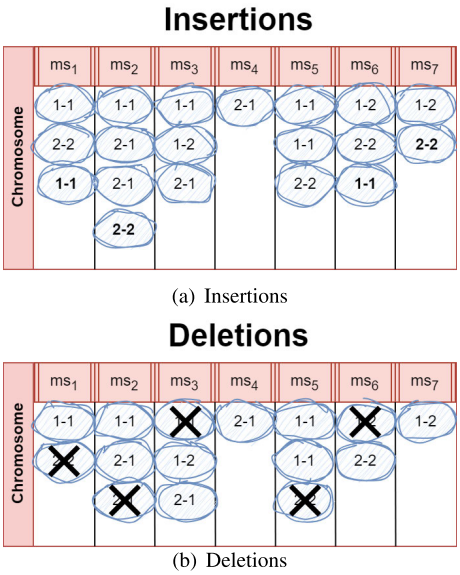


FIGURE 6. An example of the insertion and deletion mutation operations.

To maintain genetic diversity, the insertion and deletion operations are point mutations that insert or delete a gene in a DNA sequence. Based on the chromosome representation in Figure 4, for the insertion mutation operation, it adds variations, where values are randomly added to the distribution list. As shown in Figure 6(a), the microservice ms_7 is newly arranged to be performed via node 2 in cluster 2. For the deletion mutation operation, it randomly deletes values from the allocation list, where Figure 6(b) depicts the task deletion of performing the microservice ms_1 via node 2 in cluster 2. Accordingly, as shown in Figure 6, the manifest can be scaled to make mutations more flexible.

D. ALGORITHM PARAMETERS

The parameter configuration is a crucial aspect in the GA algorithm. Referring to the empirical parameter settings in existing studies, in this work we derive the parameter settings through empirical analysis, including population size, offspring size, crossover rate, mutation rate, mutation type, and

termination criterion. When adjusting the population size and offspring size, the number of individuals ranged from 50 to 500 in increments of 50. Through experimentation, a size of 200 is determined as the optimal value. For the crossover rate, a suggested setting of 0.5 is adopted. In terms of mutation, two types operations are considered, decrease mutation and increase mutation, with probabilities set at 0.5 each. This configuration is found to be suitable for our experimental environment. As for the termination criterion, experiments are conducted at intervals of 1000, and it is observed that the value of termination criterion 25000 provides the best parameter configuration. Table 3 summarizes the parameters derived from all the evaluated experiments.

TABLE 3. The values of execution parameters.

Parameter	Value
Population size	200
Offspring size	200
$Prob_{mutation}$	0.3
$Prob_{crossover}$	0.5
Termination Criterion	25000

E. ALGORITHM DESIGN

In the algorithm workflow, the parameters (i.e., cluster information, microservice information, and predefined problem model parameters and algorithm parameters) are fed into the system. The workflows and the inputs/outputs of the proposed MOMA algorithm are briefly described in Algorithm 1. In Step 1, we initialize the individuals by creating a population P. In Step 2, the algorithm's operations are executed until reaching the termination criterion. Then, the algorithm is performed based on the predefined population size. Next, the Binary Tournament Selection method is applied to select two parents and offspring is further generated by the SBX method. Consequently, we determine if mutation is required, and if so, we apply mutation to the two offspring. After performing the mutation, we combine these two offspring to form a new generation, sort the parents and offspring, and then calculate the crowding distance to measure how close an individual is to its neighbors. Accordingly, we set the next generation of individuals P. In Step 3, the final output is the ultimate result (i.e., the Pareto front).

Algorithm 1 The MOMA Algorithm

Input:
$Cluster = \{cn_l | l = 1, 2, \ldots, c\}$;
$Host = \{pn_j | j = 1, 2, \ldots, m\}$;
$Microservices = \{ms_i | i = 1, 2, \ldots, n\}$;
$Distance = \{Distance^{(i)}_{(p_{lj}, p_{lk})}\}$;
$Interaction = \{Interaction^{(i)}_{(p_{lj}, p_{lk})}\}$
$PopulationSize \Leftarrow 200$;
$OffspringSize \Leftarrow 100$;
$MutationProbability: \ Prob_{mutation} \Leftarrow 0.3$;
$CrossoverProbability: \ Prob_{crossover} \Leftarrow 1.0$;
$TerminationCriterion \Leftarrow 25000$;
P represents a population of individuals;
P_1 and P_2 represent the parents;
Q_1 and Q_2 represent the offspring;
U is a new population from two offspring;
Output:
$Solution = ParetoFront$;
1 $Initialize \ P$;
2 **while** $TerminationCriterion \neq 0$ **do**
  for $\{j = 1; j \leq PopulationSize; j++\}$ **do**
    $(P_1, P_2) \Leftarrow BinaryTournament2Selection(P)$;
    $(Q_1, Q_2) \Leftarrow SBXCrossover(P_1, P_2)$;
    if $\{rand() \leq Prob_{mutation}\}$ **then**
      $Q_1 \Leftarrow Mutation(Q_1)$;
      $Q_2 \Leftarrow Mutation(Q_2)$;
    end
    $U = U \cup \{Q_1, Q_2\}$;
  end
  $ranking \Leftarrow FastNonDominatedRanking(P \cup U)$;
  $density \Leftarrow CrowdingDistance(P \cup U)$;
  $P \Leftarrow Estimator(ranking, density, P \cup U)$;
  $TerminationCriterion = TerminationCriterion - 1$;
end
3 return the Pareto front;

VI. EXPERIMENTAL SETTINGS AND ANALYSIS

This section describes the experimental environment, examines the workload distribution for each application. Figure 7 depicts the cluster structure for generating experimental data on the heterogeneous cluster with workload, where a private repository is set up to store images, allowing us to easily perform local pulls.

A. EXPERIMENTAL ENVIRONMENT

To set up the cluster nodes, we use Ubuntu 20.04 and partition the disk adequately to meet the requirements of cloud architecture, including backup and mount areas. We then install the NVIDIA driver, CUDA, and cuDNN on the system. Once these steps are completed, we proceed to combine the heterogeneous systems by using Kubeadm. We utilize Helm to install Prometheus and Grafana (Figure 8) for monitoring the cluster and visualizing its current status. Additionally, we install DCGM to specifically monitor GPU resource usage.

Referring to Figure 7, three different types of clusters are established. First, for the primary of each cluster, we use the local PC to create virtual machines (VM) for the purpose of controls. The PC specifications are as follows: Intel Core i9-10900KF CPU @ 3.7GHz with 20 cores, 256GB of Micron Crucial PRO DDR4 2666GHz RAM, and an NVIDIA RTX 3080 GPU. The VM specifications for the primaries follow the official recommendations of a minimum of 2 cores and 4GB RAM. In the first cluster,

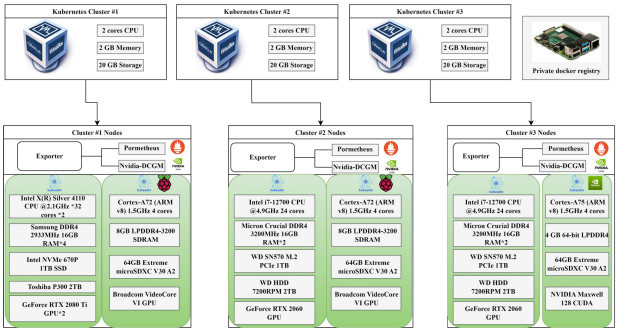


FIGURE 7. Experimental platform on multiple heterogeneous Kubernetes clusters.

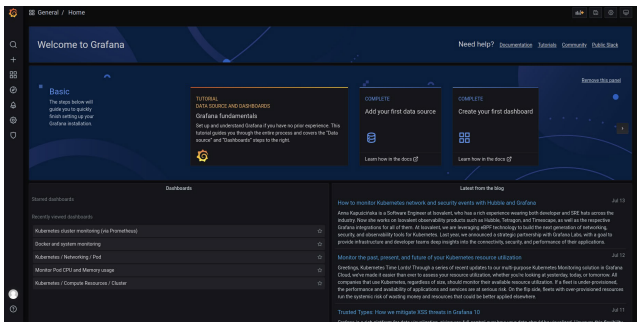


FIGURE 8. Prometheus and grafana.

FIGURE 9. Prometheus and Grafana monitoring graphs.

the configuration of the first node consists of an Intel X(R) Silver 4110 CPU @ 2.1GHz with 32 cores (2 CPUs), 64GB of Samsung DDR4 2933MHz RAM (4 modules), and 2 NVIDIA GeForce RTX 2080 Ti GPUs. Additionally, we add a Raspberry Pi to create a different architecture. In the second cluster, the configuration of the first node includes an Intel i7-12700 CPU @ 4.9GHz with 24 cores, 32GB of Micron Crucial DDR4 3200MHz RAM (2 modules), and an NVIDIA GeForce RTX 2060 GPU. It also includes a Raspberry Pi. Finally, in the third cluster, the configuration of the first node is the same as the second cluster, with an Intel i7-12700 CPU @ 4.9GHz with 24 cores, 32GB of Micron Crucial DDR4 3200MHz RAM (2 modules). Instead of adding a Raspberry Pi, here an NVIDIA Jetson Nano 4GB is applied to generate architectural diversity.

After setting up the clusters, the proposed framework is utilized to deploy workloads across the three heterogeneous cloud applications, which involves selecting the

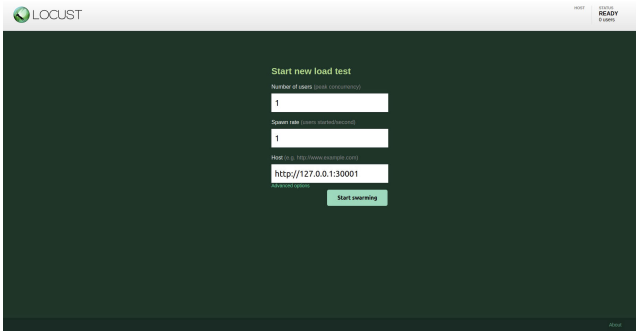


FIGURE 10. Locust workload request graph.

appropriate deployment strategies and configurations tailored to each application's specific requirements and characteristics. By leveraging our framework, we can effectively distribute and manage the workloads, optimizing performance and resource utilization across the heterogeneous cloud environment. This paper integrates the jMetal framework [52] and a generic framework based on metaheuristic multi-objective optimization for implementing and validating the proposed algorithm. However, to better address the challenges of multi-objective optimization problem, we utilize a modified version of the jMetalPy framework [53] for the algorithm development. To determine the quality of solutions, we employ the hypervolume metric, calculated using the pygmo framework [54], which is particularly advantageous for large scale parallel environments. By evaluating the hypervolume values, we can assess the dispersion in solutions and ensure the retention of a superior Pareto front.

B. EMPIRICAL ANALYSIS OF RESOURCE REQUIREMENT

This subsection explains the generation of measurement data, primarily within the Monitoring Unit. We utilize the Prometheus system to monitor nodes and display the results through Grafana. This setup enables a continuous integration and continuous deployment (CI/ CD) process, allowing us to complete the entire data collection workflow in this manner. Figure 9 showcases the data results displayed in Grafana after Prometheus measures the metrics.

In the experiment, we use Locust to request and distribute the workload for each application (Figure 10). Shell scripts are applied to send 1 to 10 user requests per second continuously for 450 seconds. Thus, the interval time between each microservice is subject to a uniform arrival distribution, which is applied for empirical analysis of resource requirement in this work. Consequently, 20 iterations of the results are collected as the data for analysis. Note that here we only present the data related to the resources. For instance, as for the GPU part, Kubernetes does not provide specific deployment options for GPUs. Therefore, we monitor GPU usage, but do not take any specific actions regarding it.

Tables 4 and 5 present the resource requirements with respect to the number of users, ranging from 1 to 10. The CPU and Memory columns represent the analysis results obtained

TABLE 4. Network interactions and resource requirements with one user request.

Application	Container	CPU	Memory	Interaction
Kubeflow	application-controller	0.01	162.37	2.74×10^{-2}
	metadata	0.01	175.80	6.87×10^{-3}
	metadata-writer	0.01	459.85	9.45×10^{-5}
	metadata-envoy-deployment	0.02	208.86	3.49×10^{-4}
	cache-server	0.01	229.99	2.34×10^{-2}
	ml-pipeline-viewer	0.01	184.03	1.25×10^{-4}
	ml-pipeline-scheduledworkflow	0.01	192.84	9.45×10^{-4}
	mysql	0.01	1486.93	2.37×10^{-3}
	minio	0.01	189.36	9.02×10^{-5}
	ml-pipeline-ui	0.01	511.34	6.74×10^{-2}
	ml-pipeline-visualizationserver	0.01	763.44	3.41×10^{-1}
	workflow-controller	0.01	301.26	4.59×10^{-1}
	tran-pipeline	2.39	1500.28	1.00×10^{-1}
	ml-pipeline-persistenceagent	0.01	286.53	1.57×10^{-2}
EdgeX foundry	edgex-device-rest	0.01	31.98	5.11×10^{-10}
	edgex-core-consul	0.01	1.65	1.00×1.02^{-10}
	edgex-core-command	0.04	18.41	1.04×10^{-9}
	edgex-support-scheduler	0.01	45.31	2.41×10^{-3}
	edgex-redis	0.12	99.41	1.19×10^{-3}
	edgex-coredata	0.03	124.55	2.51×10^{-2}
	edgex-coremetadata	0.08	28.52	5.05×10^{-2}
	edgex-support-notification	0.10	29.61	5.14×10^{-3}
	edgex-sys-mgmt	0.01	15.31	2.25×10^{-2}
Sock shop	orders-db	0.01	80.11	6.18×10^{-4}
	user-db	0.02	169.93	9.09×10^{-4}
	rabbitmq	0.10	180.60	5.21×10^{-5}
	session	0.01	96.20	4.02×10^{-3}
	carts-db	0.01	186.91	7.81×10^{-3}
	catalogue-db	0.01	445.84	6.82×10^{-6}
	orders	0.04	37.38	7.42×10^{-5}
	user	0.04	499.16	9.80×10^{-3}
	queue-master	0.10	39.51	1.18×10^{-6}
	carts	0.10	75.04	2.22×10^{-4}
	catalogue	0.02	21.53	3.55×10^{-3}
	front-end	0.17	38.03	3.29×10^{4}
	shipping	0.01	1.10	6.76×10^{12}
	payment	0.01	0.93	5.07×10^{-10}

from monitoring and measurement, measured in Cores and Mbytes, respectively. The Network Interaction column represents the total amount of network communication (send/receive) per second.

VII. PERFORMANCE EVALUATION

To assess the effectiveness of the proposed resource allocation scheme, the performances of three related algorithms (i.e., Multiopt algorithm [24], Greedy-based heuristic algorithm [37], and Kubernetes default algorithm [55]) are compared and contrasted in three distinct cloud environments with respect to resource utilization, network communication overheads, and reliability. The experiments are conducted using the data measured by the framework with user requests, ranging from 1 to 10 at an interval of 1.

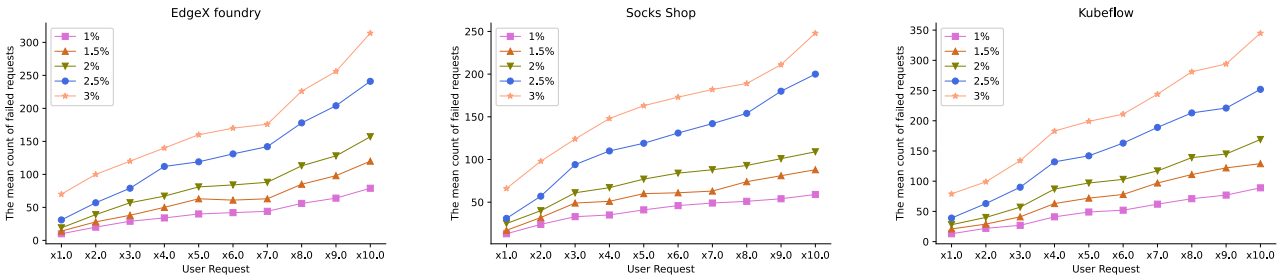
A. SYSTEM PERFORMANCE

This subsection explores the characteristics of the proposed system. Assume the failure rate *Fail* of the multi-heterogeneous clusters is in a given range, which yields *Fail* = [0.01, 0.03] [56]. In a multi-heterogeneous cluster, there may be variations in the performance (e.g., reliability and failure rates) among different nodes, which are the factors we need to consider in resource allocation and load management. Similarly, in the context of multi-heterogeneous networks, we set the distance in a given range, which is *Distance* = [1.0, 4.0]. Notice that the variation in distance is an important consideration in network communication and transmission overheads, especially in scenarios with diverse node characteristics and geographic locations.

To further depict the performance efficiency, a test is conducted with varying the percentage of the number of failed requests to the total number of user requests (e.g., 1%, 1.5%, 2%, 2.5%, and 3%, respectively). and executed on three different microservices. The results are shown in Figures 11(a), 11(b), and 11(c). It can be observed that the overall number of failed requests increases rapidly when the percentage of the number of failed requests to the total number of user requests exceeds 2.5%, which effectively demonstrates the system usability with respect to the number of failed requests. Through this analysis, we can gain insights into the overall system performance and

TABLE 5. Network interactions and resource requirements with ten user requests.

Application	Container	CPU	Memory	Interaction
Kubeflow	application-controller	0.03	232.64	6.14×10^{-2}
	metadata	0.03	199.31	1.02×10^{-2}
	metadata-writer	0.01	653.21	3.41×10^{-4}
	metadata-envoy-deployment	0.05	314.60	8.23×10^{-4}
	cache-server	0.01	311.24	4.52×10^{-2}
	ml-pipeline-viewer	0.01	224.42	6.68×10^{-3}
	ml-pipeline-scheduledworkflow	0.01	316.52	5.61×10^{-3}
	mysql	0.02	1524.01	8.57×10^{-3}
	minio	0.02	352.24	1.51×10^{-4}
	ml-pipeline-ui	0.01	592.58	1.14×10^{-2}
	ml-pipeline-visualizationserver	0.01	843.52	9.64×10^{-1}
	workflow-controller	0.03	421.14	8.26×10^{-1}
	tran-pipeline	4.39	1678.91	5.84×10^{-1}
	ml-pipeline-persistenceagent	0.02	351.23	2.62×10^{-2}
EdgeX foundry	edgex-device-rest	0.02	63.68	4.15×10^{-8}
	edgex-core-consul	0.05	88.49	6.12×1.02^{-7}
	edgex-core-command	0.08	39.42	6.23×10^{-8}
	edgex-support-scheduler	0.53	424.63	3.15×10^{-3}
	edgex-redis	0.43	103.21	5.51×10^{-3}
	edgex-coredata	0.35	425.71	3.12×10^{-2}
	edgex-coremetadata	0.32	75.15	8.23×10^{-2}
	edgex-support-notification	0.21	58.12	7.53×10^{-3}
	edgex-sys-mgmt	0.01	42.51	7.99×10^{-2}
Sock shop	orders-db	0.29	381.84	2.95×10^{-2}
	user-db	0.25	193.10	1.03×10^{-2}
	rabbitmq	0.18	173.45	5.00×10^{-5}
	session	0.10	1268.35	5.41×10^{-2}
	carts-db	0.10	1148.19	4.90×10^{-2}
	catalogue-db	0.04	156.86	2.42×10^{-5}
	orders	0.02	25.23	5.01×10^{-5}
	user	0.02	28.37	6.19×10^{-3}
	queue-master	0.24	144.87	4.33×10^{-6}
	carts	0.23	7.97	2.38×10^{-5}
	catalogue	0.19	6.15	1.01×10^{-3}
	front-end	0.35	0.20	1.89×10^{2}
	shipping	0.04	1.10	7.26×10^{-8}
	payment	0.03	7.72	5.14×10^{-5}



(a) EdgeX foundry (b) Socks shop (c) Kubeflow

FIGURE 11. Performance efficiency with varying the percentage of the number of failed requests to the total number of user requests.

identify its condition, which may further enhance the system capabilities.

B. COMPARATIVE PERFORMANCE ANALYSIS

This subsection presents a comparative analysis of the effectiveness of the MOMA algorithm, comparing to Multiopt algorithm [24], Greedy-based heuristic algorithm [37], and Kubernetes default algorithm [55]. The key characteristics of these algorithms are outlined as follows.

For the Kubernetes default algorithm [55], similar to the Binpack algorithm used in Docker Swarm [57], it seeks to allocate nodes with the lowest resource utilization, which sorts the nodes based on their available resources and assigns pods to nodes with the lowest resource utilization. For the multiopt algorithm [24], it considers CPU and memory usage of every node, the association between containers and nodes, and the clustering of containers, where these objectives align with the goals considered in the proposed algorithm as well.

For the greedy-based heuristic algorithm [37], it aims to place all microservices in the same cluster by selecting the target cluster and then prioritizing the placement based on the high interaction value among the microservices. After examining the characteristics of these algorithms, they will be examined from the perspectives of resource utilization, network communication overhead, and reliability.

1) RESOURCE UTILIZATION

Figures 12(a), 12(b), and 12(c) depict the computing resource usage in three different applications. Similarly, Figures 12(d), 12(e), and 12(f) respectively indicate the utilization of memory resources. We observe that the standard deviation of CPU and memory usage for Kubeflow is lowest compared to EdgeX Foundry and Socks Shop. This is primarily because Kubeflow experiences a higher workload, requiring more significant resource consumption. As a result, the standard deviation is lower due to the consistent and substantial resource utilization demands placed on the system, distinguishing it from EdgeX Foundry and Socks Shop. Moreover, it is evident that regardless of the application, the proposed algorithm consistently exhibits the lowest resource utilization, indicating greater efficiency in resource consumption. From a statistical perspective, the overall standard deviation of a cluster $\sigma_{cluster}$ can be obtained by combining the two standard deviations of cpu and memory, σ_{cpu} and σ_{mem}, which is given by

$$\sigma_{cluster} = \frac{1}{2}\sqrt{\sigma_{cpu}^2 + \sigma_{mem}^2}. \tag{9}$$

Referring to equation (9), Figures 13(a), 13(b), and 13(c) show that the greedy algorithm [37] has the largest standard deviation among all the algorithms in terms of resource utilization. This is because it focuses more on the interaction of microservices and does not prioritize and balance the cluster workloads. In Multiopt [24], the performance is similar to the default Kubernetes algorithm [55] due to the consideration of CPU and memory utilization for placement. However, none of these algorithms take into account the heterogeneity of the architecture, so their performance is slightly inferior to the proposed algorithm.

Observe that for the EdgeX Foundry application, given a smaller number of user requests, say 2, the Multiopt and the proposed MOMA algorithms respectively realizes 5.6% and 11.1% improvement of cluster resource utilization with respect to the default Kubernetes algorithm. In contrast, for a larger number of user requests, say 8, only the proposed MOMA algorithms realizes 7.0% improvement of cluster resource utilization with respect to the default Kubernetes algorithm. Similarly, for the Socks Shop application, with the number of user requests equal to 2, the Multiopt and the proposed MOMA algorithms respectively realizes 6.5% and 11.3% improvement of cluster resource utilization with respect to the default Kubernetes algorithm. In contrast, for the number of user requests equal to 8, the Multiopt and the proposed MOMA algorithms respectively realizes 8.3%

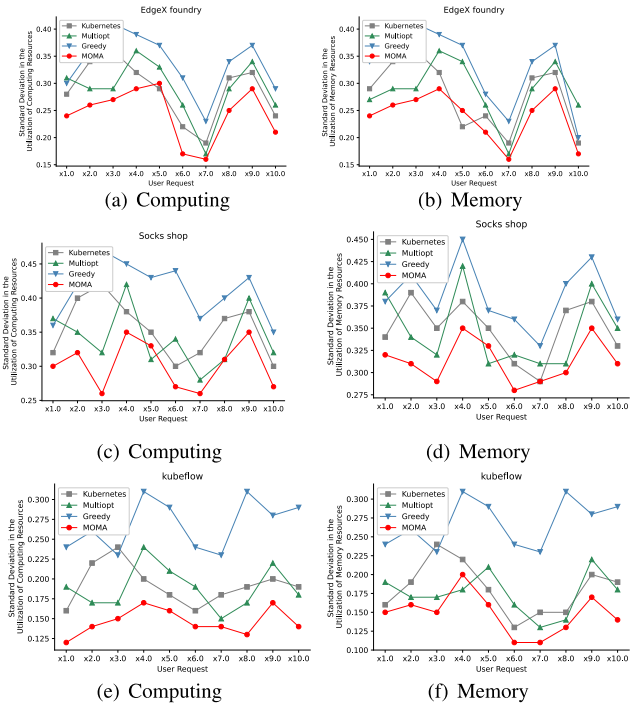


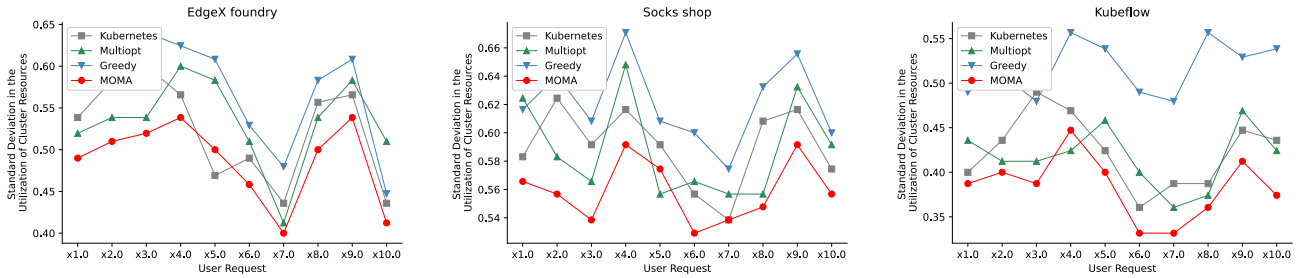
FIGURE 12. The comparison results of standard deviations in resource utilization.

and 10.0% improvement of cluster resource utilization with respect to the default Kubernetes algorithm. Furthermore, for the Kubeflow application, with the number of user requests equal to 2, the Multiopt and the proposed MOMA algorithms respectively realizes 4.7% and 7.0% improvement of cluster resource utilization with respect to the default Kubernetes algorithm. In contrast, for the number of user requests equal to 8, the Multiopt and the proposed MOMA algorithms respectively realizes 2.6% and 5.3% improvement of cluster resource utilization with respect to the default Kubernetes algorithm.

2) NETWORK COMMUNICATION OVERHEAD

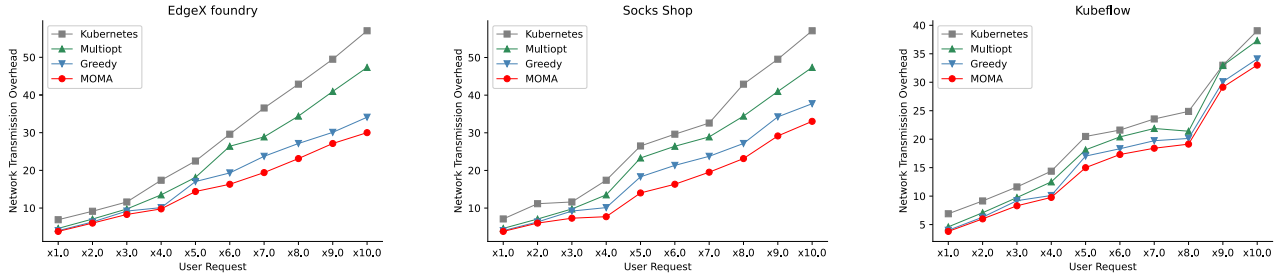
Given different number of user request scenarios, Figure 14 shows that with a smaller number of user requests (e.g., less than or equal to 3), the four algorithms have similar performance of the network communication overhead for these three applications. However, with a larger number of user requests (e.g., larger than 3), the default Kubernetes algorithm [55] contributes the largest number of communication overhead due to the only consideration of resource aspect. Moreover, Multiopt [24] attempts to place related containers together but does not consider the placement order, while Greedy [37] places all containers in the same cluster, significantly reducing network transmission costs and approaching the performance of the proposed MOMA method.

In the three different applications, as shown in Figures 14(a) and 14(b), EdgeX Foundry and Socks Shop have significantly higher overheads because of higher



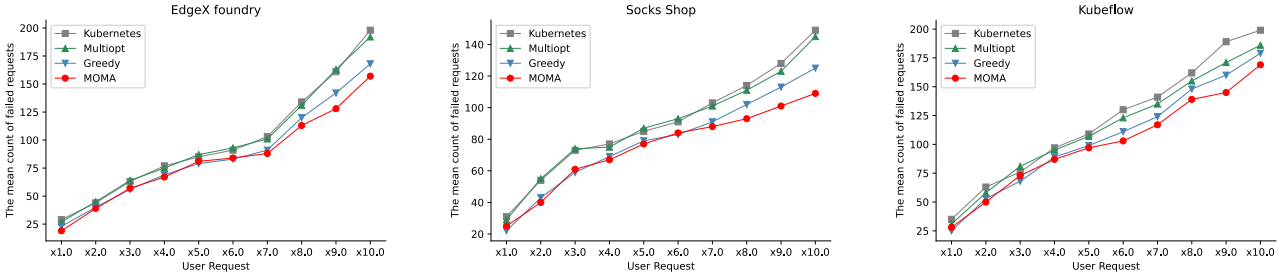
(a) EdgeX foundry (b) Socks shop (c) Kubeflow

FIGURE 13. The comparison results of standard deviations in cluster resource utilization.



(a) EdgeX foundry (b) Socks shop (c) Kubeflow

FIGURE 14. The comparison results of network communication overhead.



(a) EdgeX foundry (b) Socks shop (c) Kubeflow

FIGURE 15. The comparison results of the mean number of failed requests.

network demands, involving data creation and transmission to databases and employing socket-based communication mechanisms. In contrast, as shown in Figure 14(c), Kubeflow primarily tests data through standard databases and does not require redundant data fetching. Therefore, the overhead in Kubeflow is not as high when compared to EdgeX Foundry and Socks Shop.

3) RELIABILITY

Due to the neglect of the node failure rate, the default Kubernetes [55], Multiopt [24] and Greedy [37] algorithms obtain degraded performances compared with that of the proposed algorithm. However, observe that as shown in Figures 15(a), 15(b), and 15(c), Greedy's approach of placing the majority of services in the same cluster may mitigate the impact of the failure rate to some extent. Nevertheless, as the user requests increase (e.g., greater than 6), there is still an issue of increased failure rates. As user requests increase, the number of failures gradually becomes more challenging to control.

Different applications exhibit varying proportions of failures. For instance, in the case of Kubeflow, where training units require substantial resources, a failed unit creation can result in significant cascading errors, leading to a higher rate of failures. In EdgeX Foundry, the high interdependency among units means that a failure at an earlier stage can have a pronounced ripple effect, contributing to a higher rate of failures. Conversely, Socks Shop, with fewer constraints among its web microservices, doesn't exhibit as much interdependence in the event of failures, allowing it to continue functioning relatively independently.

C. COMPARATIVE SUMMARY

The comparative analysis explores the characteristics and performances of the Kubernetes default algorithm [55], Multiopt algorithm [24], Greedy algorithm [37], and the

TABLE 6. A summary of the results of EdgeX Foundry.

EdgeX foundry												
User Request	Kubernetes			Multiopt			Greedy			Ours		
	Clu	*Tra*	*Fai*	*Clu*	*Tra*	*Fai*	*Clu*	*Tra*	*Fai*	*Clu*	*Tra*	*Fai*
x1.0	0.54	6.90	29	0.51	4.60	27	0.58	4.00	23	**0.48**	**3.80**	**19**
x2.0	0.58	9.15	44	0.53	7.08	45	0.6	6.30	40	**0.50**	**6.00**	**39**
x3.0	0.6	11.61	63	0.53	9.75	64	0.64	9.20	**56**	**0.51**	**8.30**	57
x4.0	0.57	17.38	77	0.60	13.5	75	0.62	10.11	69	**0.53**	**9.77**	**67**
x5.0	**0.47**	22.48	85	0.58	18.13	87	0.60	17.03	**79**	0.5	**14.39**	81
x6.0	0.49	29.60	91	0.50	26.40	93	0.52	19.32	**83**	**0.45**	**16.31**	84
x7.0	0.44	36.55	103	0.41	28.88	101	0.47	23.73	91	**0.4**	**19.41**	**88**
x8.0	0.56	42.88	134	0.53	34.40	131	0.58	27.13	120	**0.5**	**23.13**	**113**
x9.0	0.57	49.50	161	0.58	40.94	163	0.60	30.09	142	**0.53**	**27.13**	**128**
x10.0	0.44	57.04	198	0.50	47.31	192	0.44	34.10	168	**0.41**	**30.01**	**157**

[1] Clu represents the cluster resource utilization in Figure 13(a).
[2] Tra represents the network transmission overhead in Figure 14(a).
[3] Fai represents the mean count of fail requests in Figure 15(a).

TABLE 7. A summary of the results of Socks shop.

Socks shop												
User Request	Kubernetes			Multiopt			Greedy			Ours		
	Clu	*Tra*	*Fai*	*Clu*	*Tra*	*Fai*	*Clu*	*Tra*	*Fai*	*Clu*	*Tra*	*Fai*
x1.0	0.58	7.11	31	0.62	4.60	29	0.61	4.00	**22**	**0.56**	23.80	25
x2.0	0.62	11.16	54	0.58	7.09	43	0.64	6.31	43	**0.55**	**6.01**	**40**
x3.0	0.59	11.62	73	0.56	9.76	59	0.60	9.21	**59**	**0.53**	**7.31**	61
x4.0	0.61	17.40	77	0.64	13.51	69	0.67	10.12	69	**0.59**	**7.71**	**67**
x5.0	0.59	26.50	85	**0.55**	23.31	79	0.60	18.31	79	0.57	14.02	**77**
x6.0	0.55	29.63	91	0.56	26.42	83	0.60	21.34	**83**	**0.52**	**16.33**	84
x7.0	**0.53**	32.59	103	0.55	28.91	91	0.57	23.75	91	**0.53**	**19.53**	**88**
x8.0	0.60	42.92	114	0.55	34.43	102	0.63	27.16	102	**0.54**	**23.15**	**93**
x9.0	0.61	49.55	128	0.63	40.98	113	0.65	34.23	113	**0.59**	**29.16**	**101**
x10.0	0.57	57.10	149	0.59	47.36	125	0.6	37.73	125	**0.55**	**33.04**	**109**

[1] Clu represents the cluster resource utilization in Figure 13(b).
[2] Tra represents the network transmission overhead in Figure 14(b).
[3] Fai represents the mean count of fail requests in Figure 15(b).

TABLE 8. A summary of the results of Kubeflow.

Kubeflow												
User Request	Kubernetes			Multiopt			Greedy			Ours		
	Clu	*Tra*	*Fai*	*Clu*	*Tra*	*Fai*	*Clu*	*Tra*	*Fai*	*Clu*	*Tra*	*Fai*
x1.0	0.40	6.90	35	0.43	4.6	31	0.48	4.00	**25**	**0.38**	**3.80**	28
x2.0	0.43	9.15	63	0.41	7.08	58	0.50	6.30	53	**0.40**	**6.00**	**50**
x3.0	0.48	11.61	76	0.41	9.75	81	0.47	9.20	**68**	**0.38**	**8.30**	73
x4.0	0.46	14.38	97	**0.42**	**12.5**	95	0.55	10.11	89	0.44	9.77	**87**
x5.0	0.42	20.48	109	0.45	18.13	107	0.53	17.03	99	**0.40**	**15.00**	**97**
x6.0	0.36	21.6	130	0.40	20.4	123	0.48	18.32	111	**0.33**	**17.31**	**103**
x7.0	0.38	23.55	141	0.36	21.88	135	0.47	19.73	124	**0.33**	**18.41**	**117**
x8.0	0.38	24.88	162	0.37	21.4	155	0.55	20.13	148	**0.36**	**19.13**	**139**
x9.0	0.44	32.99	189	0.46	32.94	171	0.52	30.09	160	**0.41**	**29.13**	**145**
x10.0	0.43	39.04	199	0.42	37.31	186	0.53	34.10	179	**0.37**	**33.01**	**169**

[1] Clu represents the cluster resource utilization in Figure 13(c).
[2] Tra represents the network transmission overhead in Figure 14(c).
[3] Fai represents the mean count of fail requests in Figure 15(c).

proposed MOMA algorithm. As shown in Figures 14 and 15, for the Greedy algorithm [37] with consolidating most services within the same cluster, the network communication overheads may be suppressed and the failure rates may have less of an impact on network reliability. However, as shown in Figures 12 and 13, this operation may deteriorate the performance of cluster resource utilization. As to the Kubernetes default algorithm [55] and Multiopt algorithm [24], they

respectively only allocate nodes with the lowest resource utilization and consider CPU and memory usage of every node to handle microservice interaction. Therefore, as depicted in Figures 12 to 15, these two algorithms perform similarly for the microservice task with respect to resource utilization, network communication overheads, and network reliability.

Overall, the proposed MOMA algorithm outperforms the Kubernetes default algorithm [55], Multiopt algorithm [24], and Greedy algorithm [37] in terms of resource utilization, network transmission overhead, and reliability usage in the three applications within multi-heterogeneous cluster environments. This is because the above three algorithms that are being compared lack consideration for node failure rates and heterogeneous architectures, resulting in relative performance degradation. Moreover, the proposed system is scalable to be able to ingest an increasing number of heterogeneous Kubernetes clusters and services. For instance, when the system includes a new heterogeneous Kubernetes cluster or a service, the proposed MOMA algorithm can be applied to optimize resource allocation. For the robustness issue, the proposed framework can be employed to implement a robust microservices environment with resource balancing. Referring to the analysis in Section VII-B3, the architecture principle and design pattern of the proposed framework architecture can help in building a reliable microservice architecture. The summarized findings are presented in Tables 6-8.

VIII. CONCLUSION

This work establishes a bi-objective optimization model: (1) maximum resource utilization and (2) reducing network communication overhead. To evaluate the performance of the proposed MOMA model, we apply three different microservice applications (i.e., edgex foundry, socks shop, and kubeflow) and examine the framework via microservice workload analysis with the measurement data from heterogeneous architectures of real-world scenarios. To achieve a more diverse and better set of solutions, we develop the MOMA algorithm based on the improved Elitist NSGA-II. We design a genetic representation, utilize SBX crossover operator, and employ two different mutation operators. To evaluate the quality of our solutions, hypervolume is used as a metric. Compared with the existing algorithms, the experimental results show that the proposed algorithm demonstrate significant improvements in resource utilization, network transmission overhead, and reliability across the three different applications.

To further extend this study on resource allocation in multiple heterogeneous clouds, possible future works include (1) considering GPU management in microservice resource allocation due to emerging microservice applications with GPU utilization, (2) integrating cloud-native services from certain Graduated projects into our framework, (3) deriving the theoretical bounds of the resource matrices for further investigating important characteristics of a microservices system and providing a baseline for the overall health of the system, (4) exploring platform metrics for monitoring the microservice health, energy consumption, or the entire microservices application, (5) investigating the algorithm's performance by enlarging the number of heterogeneous Kubernetes clusters and services, and (6) using a large and diverse evaluation set to abstract the system characteristics (e.g., scalability, generalizability, and reliability), and to benchmark cloud/edge computing platforms [58], [59]. We plan to explore and experiment with a wider array of meta-heuristic algorithms through comparative analysis to further optimize our approach and extend the current research by incorporating a larger heterogeneous resource pool, such as investigating the possibility of adding virtual machines as additional work nodes in the architecture and making the heterogeneous infrastructure more comprehensive and versatile.

REFERENCES

[1] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014, Art. no. 2.

[2] R. R. Yadav, E. T. G. Sousa, and G. R. A. Callou, "Performance comparison between virtual machines and Docker containers," *IEEE Latin Amer. Trans.*, vol. 16, no. 8, pp. 2282–2288, Aug. 2018.

[3] I. Al Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, and A. Palopoli, "Container orchestration engines: A thorough functional and performance comparison," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2019, pp. 1–6.

[4] A. N. Jing Hui and B. S. Lee, "Epsilon: A microservices based distributed scheduler for kubernetes cluster," in *Proc. 18th Int. Joint Conf. Comput. Sci. Softw. Eng. (JCSSE)*, Jun. 2021, pp. 1–6.

[5] E.-H. El Baqqaly and A. H. Khaleel, "Optimizing big data analytics for reliability and resilience: A survey of techniques and applications," *Mesopotamian J. Big Data*, vol. 2023, pp. 118–124, Nov. 2023.

[6] M. R. Mesbahi, A. M. Rahmani, and M. Hosseinzadeh, "Reliability and high availability in cloud computing environments: A reference roadmap," *Hum.-Centric Comput. Inf. Sci.*, vol. 8, no. 1, p. 20, Jul. 2018.

[7] (Nov. 18, 2023). *Microservices Monitoring: Challenges, Metrics, and Tips for Success*. [Online]. Available: https://lumigo.io/microservices-monitoring/

[8] R. G. Shooli and M. M. Javidi, "Using gravitational search algorithm enhanced by fuzzy for resource allocation in cloud computing environments," *Social Netw. Appl. Sci.*, vol. 2, no. 2, p. 195, Feb. 2020.

[9] E. F. Khor, K. C. Tan, and T. H. Lee, "Evolutionary algorithms for multi-objective optimization: Performance assessments and comparisons," *Artif. Intell.*, vol. 17, pp. 251–290, Jun. 2002.

[10] V. Pereira, P. Sousa, and M. Rocha, "A comparison of multi-objective optimization algorithms for weight setting problems in traffic engineering," *Natural Comput.*, vol. 21, no. 3, pp. 507–522, Sep. 2022.

[11] H. Seo and C. Lee, "A new GA-based resource allocation scheme for a reader-to-reader interference problem in RFID systems," in *Proc. IEEE Int. Conf. Commun.*, May 2010, pp. 1–5.

[12] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[13] K. Chandrasekaran, "Tutorial: Resource management in cloud computing," in *Proc. IEEE/ACM 6th Int. Conf. Utility Cloud Comput.*, Dec. 2013, pp. 31–32.

[14] S. Singhal and A. Sharma, "Resource scheduling algorithms in cloud computing: A big picture," in *Proc. 5th Int. Conf. Inf. Syst. Comput. Netw. (ISCON)*, Oct. 2021, pp. 1–6.

[15] N. R. R. Mohan and E. B. Raj, "Resource allocation techniques in cloud computing—Research challenges for applications," in *Proc. 4th Int. Conf. Comput. Intell. Commun. Netw.*, Nov. 2012, pp. 556–560.

[16] S. Huaxin, X. Gu, K. Ping, and H. Hongyu, "An improved kubernetes scheduling algorithm for deep learning platform," in *Proc. 17th Int. Comput. Conf. Wavelet Act. Media Technol. Inf. Process. (ICCWAMTIP)*, Dec. 2020, pp. 113–116.

[17] L. Zhu, J. Li, Z. Liu, and D. Zhang, "A multi-resource scheduling scheme of kubernetes for IIoT," *J. Syst. Eng. Electron.*, vol. 33, no. 3, pp. 683–692, Jun. 2022.

[18] Z. He, "Novel container cloud elastic scaling strategy based on kubernetes," in *Proc. IEEE 5th Inf. Technol. Mechatronics Eng. Conf. (ITOEC)*, Jun. 2020, pp. 1400–1404.

[19] W. Guo, H. Huang, Z. Niu, and W. Liu, "A task priority-based resource scheduling algorithm for container-based clouds," in *Proc. IEEE Int. Conf. Emergency Sci. Inf. Technol. (ICESIT)*, Nov. 2021, pp. 268–273.

[20] A. Ning, "A customized kubernetes scheduling algorithm to improve resource utilization of nodes," in *Proc. 3rd Asia–Pacific Conf. Commun. Technol. Comput. Sci. (ACCTCS)*, Feb. 2023, pp. 588–591.

[21] H. Fu, Y. Fan, G. Pan, L. Shi, Q. Huang, Y. Tian, and C. Xiao, "Research on cloud computing resource allocation based on particle swarm optimization algorithm," in *Proc. IEEE Int. Conf. Adv. Electr. Eng. Comput. Appl. (AEECA)*, Aug. 2021, pp. 147–151.

[22] H. B. Abdallah, A. A. Sanni, K. Thummar, and T. Halabi, "Online energy-efficient resource allocation in cloud computing data centers," in *Proc. 24th Conf. Innov. Clouds, Internet Netw. Workshops (ICIN)*, Mar. 2021, pp. 92–99.

[23] L. Zuo, L. Shu, S. Dong, C. Zhu, and T. Hara, "A multi-objective optimization scheduling method based on the ant colony algorithm in cloud computing," *IEEE Access*, vol. 3, pp. 2687–2699, 2015.

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

[25] H. Gong, "Resource allocation in cloud computing environment based on NSGA-II," in *Proc. 2nd Int. Conf. Comput. Data Sci. (CDS)*, Jan. 2021, pp. 39–43.

[26] B. Tan, H. Ma, and Y. Mei, "A NSGA-II-based approach for multi-objective micro-service allocation in container-based clouds," in *Proc. 20th IEEE/ACM Int. Symp. Cluster, Cloud Internet Comput. (CCGRID)*, May 2020, pp. 282–289.

[27] K. D. Tran, "Elitist non-dominated sorting GA-II (NSGA-II) as a parameter-less multi-objective genetic algorithm," in *Proc. IEEE SoutheastCon*, Apr. 2005, pp. 359–367.

[28] R. Rajakumar, P. Dhavachelvan, and T. Vengattaraman, "A survey on nature inspired meta-heuristic algorithms with its domain specifications," in *Proc. Int. Conf. Commun. Electron. Syst. (ICCES)*, Oct. 2016, pp. 1–6.

[29] S. Sunaina, B. D. Shivahare, V. Kumar, S. K. Gupta, P. Singh, and M. Diwakar, "Metaheuristic optimization algorithms and recent applications: A comprehensive survey," in *Proc. Int. Conf. Comput. Intell., Commun. Technol. Netw. (CICTN)*, Apr. 2023, pp. 506–511.

[30] B. Liu, P. Li, W. Lin, N. Shu, Y. Li, and V. Chang, "A new container scheduling algorithm based on multi-objective optimization," *Soft Comput.*, vol. 22, no. 23, pp. 7741–7752, Dec. 2018.

[31] C. Kaewkasi and K. Chuenmuneewong, "Improvement of container scheduling for Docker using ant colony optimization," in *Proc. 9th Int. Conf. Knowl. Smart Technol. (KST)*, Feb. 2017, pp. 254–259.

[32] N. Gupta, M. Batra, and A. Khosla, "Optimizing greedy algorithm to balance the server load in cloud simulated environment," in *Proc. 3rd Int. Conf. Inventive Res. Comput. Appl. (ICIRCA)*, Sep. 2021, pp. 351–356.

[33] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "Firm: An intelligent fine-grained resource management framework for slo-oriented microservices," in *Proc. 14th USENIX Conf. Oper. Syst. Design Implement. (OSDI)*. Berkeley, CA, USA: USENIX Association, 2020, pp. 1–22.

[34] F. Guo, B. Tang, M. Tang, and W. Liang, "Deep reinforcement learning-based microservice selection in mobile edge computing," *Cluster Comput.*, vol. 26, no. 2, pp. 1319–1335, Aug. 2022.

[35] W. Li, B. Liu, H. Gao, and X. Su, "Transfer learning based algorithm for service deployment under microservice architecture," in *Communications and Networking*, H. Gao, J. Wun, J. Yin, F. Shen, Y. Shen, and J. Yu, Eds. Cham, Switzerland: Springer, 2022, pp. 52–62.

[36] A. H. Ali, M. G. Yaseen, M. Aljanabi, S. A. Abed, and C. Gpt, "Transfer learning: A new promising techniques," *Mesopotamian J. Big Data*, vol. 2023, pp. 29–30, Feb. 2023.

[37] J. Han, Y. Hong, and J. Kim, "Refining microservices placement employing workload profiling over multiple kubernetes clusters," *IEEE Access*, vol. 8, pp. 192543–192556, 2020.

[38] M. E. Frincu and C. Craciun, "Multi-objective meta-heuristics for scheduling applications with high availability requirements and cost constraints in multi-cloud environments," in *Proc. 4th IEEE Int. Conf. Utility Cloud Comput.*, Dec. 2011, pp. 267–274.

[39] V. Sharma, "Managing multi-cloud deployments on kubernetes with istio, prometheus and grafana," in *Proc. 8th Int. Conf. Adv. Comput. Commun. Syst. (ICACCS)*, vol. 1, Mar. 2022, pp. 525–529.

[40] S. Lee, S. Son, J. Han, and J. Kim, "Refining micro services placement over multiple kubernetes-orchestrated clusters employing resource monitoring," in *Proc. IEEE 40th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Nov. 2020, pp. 1328–1332.

[41] I. Rocha, C. Göttel, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni, "Heats: Heterogeneity- and energy-aware task-based scheduling," in *Proc. 27th Euromicro Int. Conf. Parallel, Distrib. Netw.-Based Process. (PDP)*, Feb. 2019, pp. 400–405.

[42] I. M. Ali, K. M. Sallam, N. Moustafa, R. Chakraborty, M. Ryan, and K. R. Choo, "An automated task scheduling model using non-dominated sorting genetic algorithm II for fog-cloud systems," *IEEE Trans. Cloud Comput.*, vol. 10, no. 4, pp. 2294–2308, Oct. 2022.

[43] B. T. Hasan and D. B. Abdullah, "Real-time resource monitoring framework in a heterogeneous kubernetes cluster," in *Proc. Muthanna Int. Conf. Eng. Sci. Technol. (MICEST)*, Mar. 2022, pp. 184–189.

[44] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. 8th USENIX Conf. Netw. Syst. Design Implement. (NSDI)*. Berkeley, CA, USA: USENIX Association, 2011, pp. 323–336.

[45] T. Wang, H. Xu, and F. Liu, "Multi-resource load balancing for virtual network functions," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2017, pp. 1322–1332.

[46] D. Liu, G. Zhu, J. Zhang, and K. Huang, "Wireless data acquisition for edge learning: Importance-aware retransmission," in *Proc. IEEE 20th Int. Workshop Signal Process. Adv. Wireless Commun. (SPAWC)*, Jul. 2019, pp. 1–5.

[47] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang, "A survey on the edge computing for the Internet of Things," *IEEE Access*, vol. 6, pp. 6900–6919, 2018.

[48] S. Katoch, S. S. Chauhan, and V. Kumar, "A review on genetic algorithm: Past, present, and future," *Multimedia Tools Appl.*, vol. 80, no. 5, pp. 8091–8126, Feb. 2021.

[49] S. N. Deepa and S. N. Sivanandam, *Introduction to Genetic Algorithm*, 1st ed. Berlin, Germany: Springer-Verlag, 2008.

[50] K. Deb and H.-G. Beyer, "Self-adaptive genetic algorithms with simulated binary crossover," *Evol. Comput.*, vol. 9, no. 2, pp. 197–221, Jun. 2001.

[51] J. Chacón and C. Segura, "Analysis and enhancement of simulated binary crossover," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Jul. 2018, pp. 1–8.

[52] J. J. Durillo, A. J. Nebro, and E. Alba, "The jMetal framework for multi-objective optimization: Design and architecture," in *Proc. IEEE Congr. Evol. Comput.*, Jul. 2010, pp. 1–8.

[53] A. Benítez-Hidalgo, A. J. Nebro, J. García-Nieto, I. Oregi, and J. Del Ser, "JMetalPy: A Python framework for multi-objective optimization with metaheuristics," *Swarm Evol. Comput.*, vol. 51, Dec. 2019, Art. no. 100598.

[54] F. Biscani and D. Izzo, "A parallel global multiobjective framework for optimization: Pagmo," *J. Open Source Softw.*, vol. 5, no. 53, p. 2338, Sep. 2020.

[55] Z. Zhong and R. Buyya, "A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources," *ACM Trans. Internet Technol.*, vol. 20, no. 2, pp. 1–24, Apr. 2020.

[56] S. Prathiba and S. Sowvarnica, "Survey of failures and fault tolerance in cloud," in *Proc. 2nd Int. Conf. Comput. Commun. Technol. (ICCCT)*, Feb. 2017, pp. 169–172.

[57] C. Cérin, T. Menouer, W. Saad, and W. B. Abdallah, "A new Docker swarm scheduling strategy," in *Proc. IEEE 7th Int. Symp. Cloud Service Comput. (SC)*, Nov. 2017, pp. 112–117.

[58] A. Detti, L. Funari, and L. Petrucci, "μbench: An open-source factory of benchmark microservice applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 3, pp. 968–980, Mar. 2023.
[59] M. Hamilton, N. Gonsalves, C. Lee, A. Raman, B. Walsh, S. Prasad, D. Banda, L. Zhang, L. Zhang, and W. T. Freeman, "Large-scale intelligent microservices," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2020, pp. 298–309.

QI-HONG CHEN received the B.S. degree in computer science and information engineering from the National Chin-Yi University of Technology, Taichung, Taiwan, in 2021, and the M.S. degree in electrical engineering from the National Chung Hsing University, Taichung, in 2023. His main research interests include cloud computing, computer networks, and mobile telephony.

CHIH-YU WEN (Senior Member, IEEE) received the B.S.E.E. and first M.S.E.E. degrees from the National Cheng Kung University, Tainan, Taiwan, in 1995 and 1997, respectively, and the second M.S.E.E. and Ph.D. degrees from the University of Wisconsin-Madison, Madison, WI, USA, in 2002 and 2005, respectively, all in electrical engineering.

He joined the Department of Electrical Engineering, National Chung Hsing University, Taichung, Taiwan, in 2006, where he is currently a Lifetime Distinguished Professor. His current research interests include wireless communications, biomedical signal processing for health monitoring, and distributed networked sensing and control. He is a member of the Chinese Institute of Engineers. He was a recipient of the National Innovation Awards–Institute for Biotechnology and Medicine Industry, in 2016, 2018, 2019, 2020, and 2021, for contributions to remote rehabilitation and smart medical devices. He and coauthors were also a recipient of two Best Paper Awards, such as the 2020 Taiwan Telecommunications Annual Symposium Best Paper Award and the 2021 IEEE ECBIOS Best Paper Award. Since January 2018, he has been an Associate Editor of *IET Signal Processing*.