Journal of Network and Computer Applications 160 (2020) 102629

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Application deployment using containers with auto-scaling for microservices in cloud environment

Satish Narayana Srirama *, Mainak Adhikari, Souvik Paul

Mobile & Cloud Lab, Institute of Computer Science, University of Tartu, Estonia

ARTICLE INFO

Keywords:
Cloud computing
Microservices
Containers
Dynamic bin-packing strategy
Cold starts
Resource utilization

ABSTRACT

A microservice-based application is composed of a set of small services that run within their own processes and communicate with a lightweight mechanism. Processing the microservices efficiently with minimum processing time and cost, while utilizing the computing resources efficiently, is a challenging task in a cloud environment. To address this challenge, in this paper, we propose a new container-aware application scheduling strategy with an auto-scaling policy. The proposed strategy deploys the requested applications on the best-fit lightweight containers, with minimum deployment time, based on the resource requirements. Another important issue of the container-aware cloud environment is the cold start effect, which is solved using a rule-based policy in the proposed work for minimizing deployment time and cost of the applications. Furthermore, a dynamic bin-packing strategy is designed for deploying the applications to the minimum number of physical machines (PMs) with efficient utilization of the computing resources. Finally, a heuristic-based auto-scaling policy has been designed for minimizing the wastage of the computing resources in the cloud data center. Through numerical evaluation, we have shown the superiority of the proposed method over the existing state-of-the-art algorithms in terms of processing time, processing cost, resource utilization, and required numbers of PMs.

1. Introduction

Microservices is a software development technique that consists of a collection of small and autonomous loosely coupled services (Fowler and Lewis, 2014; Villamizar et al., 2015). Microservices have evolved from a service-oriented application development strategy where a big application is divided into various small independent service units and each service offers only one business need of the application (Yahia et al., 2016). Currently, most of the Industries like Spotify, LinkedIn, Amazon, SoundCloud, Netflix, and others have evolved their applications towards the microservice platform for increasing the efficiency and scalability of the computing resources. Flower and Lewis provided a promising definition of a microservice-based application as "*an approach for developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API*" (Fowler and Lewis, 2014). Microservices can provide significant benefits by developing, designing, realizing and testing the services with great agility. An illustrative example of a microservices-based application is Netflix ("Netflix" and Retrieved from). Netflix service consists of a set of small, independently scalable and deployable microservices which are used for the ecosystem. Netflix service makes a copy of each document (as a form of video) and stores it across the servers (i.e. Netflix cloud infrastructure) in the globe so that the Netflix orchestrator delivers the document from the closest server to the users at maximum quality and speed. Recently, processing the microservice-based applications efficiently in the cloud servers, while meeting various Quality-of-Services (QoS) parameters, has received significant attention from the research point of view (Yahia et al., 2016).

In the earlier days, the microservices are successfully deployed and executed in a traditional cloud server using virtual machines (VMs). The providers can deploy the VMs for executing the microservices in parallel, which makes the system more reliable and increases the parallelism among requested microservices. However, the fundamental limitation of VM-aware virtualization is the set-up methodology, where the VMs have an ability to consolidate the applications onto a single server with separate operating systems (OSs) image, which creates overhead in storage footprint. Recently, lightweight containers play an important role in virtualizing the computing resources with minimum deployment time and executing the microservices efficiently in a cloud environment, while meeting the service level agreements (SLAs). The containers are standalone and self-contained units that package software and its dependencies together (Adhikari and Srirama, 2019). Like the VMs, the

* Corresponding author.
E-mail addresses: satish.srirama@ut.ee (S.N. Srirama), mainak@ut.ee (M. Adhikari), souvik.paul@ut.ee (S. Paul).

https://doi.org/10.1016/j.jnca.2020.102629
Received 12 August 2019; Received in revised form 23 February 2020; Accepted 20 March 2020
Available online 2 April 2020

container aware virtualization technique enables the resources on a single server, which can be shared by multiple applications (Hightower et al., 2017; Medel et al., 2016). However, instead of virtualizing the resources at the hardware-level, containers do so at the process level. This makes the environment fast and flexible with fine-grained resource sharing. Container technologies such as Docker and Linux Containers deploy the microservices in a cloud environment for meeting the objectives of the microservices (DelValle et al., 2016; Docker, 2018).

One of the important challenging issues in the cloud environment is to minimize the total processing time and cost of the microservices, while auto-scaling the computing resources based on the requests from the users (Adhikari et al., 2018; Buyya et al., 2019). Over-provisioning of the computing resources leads to resource wastage, while under-provisioning causes the performance degradation of the cloud servers. Thus, an optimal auto-scaling mechanism identifies the resource capability of the active set of PMs/VMs in a data center in terms of their adoption of workload changes by generating a trigger for scaling-out/down the computing resources as per the predefined threshold values. In addition, due to the lightweight feature, containers are useful for high elasticity with more concurrent deployment on the cloud servers. Moreover, container-aware policy demands low computer resource overhead and performs better than the VMs. As a result, a container-aware auto-scaling policy needs to address two important research aspects in the cloud environment including, a) cost minimization, by assigning the microservices on the suitable PMs/VMs based on the requirements, and b) maximize resource utilization through proper auto-scaling mechanisms.

In this paper, we attempt to design a new scheduling strategy for processing microservices efficiently in a cloud environment with a heuristic auto-scaling policy. The contribution of the proposed strategy is four folded. The first contribution is to find a suitable container for each microservice using its multiple resource requirements with a greedy approach. This minimizes the total processing cost while meeting the SLA constraints. The second contribution is to minimize the cold start problem of the containers by reusing the active containers for a new set of microservices. This minimizes the deployment times of the containers with minimum completion time and overall processing cost. The third contribution is to deploy the containers on the suitable physical machines (PMs)/VMs based on the availability of the resources using the best-fit dynamic bin packing strategy. This utilizes computing resources efficiently by minimizing the number of PMs. Here, we consider both PMs and VMs, as in a regular public cloud, it is a lot more logical to have access to VMs, allowing to deploy the containers on the top of the VMs for processing microservices. In this regard, the remaining paper will be mostly mentioning PMs, occasionally interchanging it with VMs, however, we acknowledge applying the mapping to both PM/VMs. The fourth contribution is to auto-scale the cloud resources based on the utilization of the existing resources and further resources required by the incoming microservice requests, i.e. for the relevant containers. This minimizes the overall resource wastage of the PMs and balances the workloads among them. Finally, the proposed strategy is evaluated using various performance matrices including processing time, processing cost, resource utilization, and numbers of PMs/VMs over four popular Docker Swarm strategies. The major contributions of this work are summarized as follows.

- Designed a heuristic-based matching strategy to find the best-fit container for each microservice based on its multiple resource requirements, i.e. CPU and memory.
- Designed a new rule-based strategy for optimizing the cold start problem of container-based scheduling in the cloud environment.
- Developed the best-fit dynamic bin packing strategy for deploying the selected containers on the suitable PMs.
- Implemented a heuristic auto-scaling strategy on the containerized cloud platform to control the scaling out/down the resources by varying the number of PMs/VMs.
- Finally, we evaluate the performance of the proposed algorithm with four popular Docker Swarm strategies using real-time datasets over various performance metrics.

The remaining sections are organized as follows: Section 2 reviews the previous works on container-aware scheduling and auto-scaling policies in the cloud environment. Section 3 discusses the system model and problem statement of the work. The proposed strategy is elaborated in Section 4. Section 5 discusses the performance evaluation of the proposed method over the existing state-of-the-art algorithms. Finally, Section 6 concludes the work with future directions.

2. Related work

Lightweight containers have been used for efficiently utilizing cloud resources with more concurrent deployment. Container-aware scheduling strategies have been extensively studied in the literature with its merits and outcomes. In addition, the auto-scaling policy has been extensively focused by the researchers for minimizing the processing cost and efficient resource utilization in the data center. There have been multiple recent efforts to survey the state-of-the-art container-aware scheduling and auto-scaling policies in the cloud environment, which are reviewed as follows.

2.1. Container-based scheduling

There are few initiatives that have been tailored for processing the user-based applications on the containers in a cloud environment. Tang et al. have designed a container-aware auto-scaling strategy in a cloud environment with a mathematical model for scaling the containers for utilizing the computing resources efficiently and minimizing the overall processing time (Tang et al., 2018). Perez et al. have developed a task scheduling algorithm for finding an optimal Docker container for each task and deployed them on the AWS Lambda domain (Pérez et al., 2018). This reduced the overall processing time while meeting QoS constraints. Li et al. have designed a container-based scheduling scheme for optimizing loads of PMs (Li et al., 2015). This policy found an optimal loaded PM using a monitoring strategy for deploying the selected containers and completed the execution within minimum processing time. This helps to balance the load among the PMs and utilizes the resources efficiently. Yin et al. have developed an optimal scheduling strategy using the containerization technique (Yin et al., 2018). The main goal of this strategy is to minimize the total processing time with minimum delay.

Similarly, Xu et al. have developed a container-aware task scheduling strategy with an optimal scheduling strategy for mapping the requested tasks to suitable containers (Xu et al., 2014). This minimized the response time and processing time of the tasks while utilizing the resources efficiently. Zhang et al. have designed a scheduling platform for a novel video surveillance system (Zhang et al., 2016). This strategy found a suitable container based on future workload prediction theory and minimized the overall computation time with efficient resource utilization. Kaewkasi et al. have developed an ant colony optimization strategy for finding an optimal container using the availability of the resources (Kaewkasi and Chuenmuneewong, 2017). The main goal of this strategy was to improve the resource utilization of the PMs and minimized the processing time. Guerrero et al. have developed a resource provisioning strategy for finding an optimal container based on the Genetic algorithm in the Kubernetes platform (Guerrero et al., 2018). This strategy minimized the network overhead while improving the QoS of the resources.

2.2. Auto-scaling strategy

Researchers have used various types of auto-scaling strategies in a cloud environment for scaling out/down the resources based on the current load of the PMs (Qu et al., 2018). The researchers mainly used various types of swarm optimization techniques (Chen and Bahsoon,

2017; Frey et al., 2013), analytical modeling (Barna et al., 2018; Li et al., 2009; Gandhi et al., 2014) and application profiling (Qu et al., 2016; Gandhi et al., 2012) techniques for scaling. The existing swarm optimization techniques are used to find a suitable sample space for scaling operation and also suggest an optimal configuration of the computing resources. The analytical modeling is mainly focused on the performance model of the PMs inside the data center and minimizes the overall cost of the resources while meeting various QoS constraints. However, the application profiling approach makes a relationship between workload intensity, QoS parameters and the required number of resources by the applications. However, most of the existing strategies have used the capacity estimations of the resources which are not useful for the container-based cloud environment for processing microservices. The container-based matrices such as CPU and memory usage by the containers, number of warm containers (containers which are active in the PMs and are used for other microservices for further processing) and the Queue matrices such as length of the queue, number of microservices that are ready to execute, etc are not considered yet for the auto-scaling in the cloud environment.

Nowadays, the researchers have taken some initiative for rule-based autoscaling strategies for microservices (Gotin et al., 2018; Florio et al., 2016; Toffetti et al., 2017). Most of the works are focused on few container-level matrices such as CPU and memory usage instead of the cold start problem in the cloud environment. Zhang et al. have designed an auto-scaling strategy for the computing resources for the process-level containers (Zhang et al., 2019). This paper used current workload usage, workload duration and the cool-down period for scaling the containers in the PMs. Moreover, we have designed a new strategy for container-based scheduling and auto-scaling in the cloud environment. Here, the containers are assigned for each microservice based on its resource requirement and deploy the container to a suitable PM based on a dynamic bin packing strategy. This reduces the total number of PMs and minimizes the total cost of an application. Moreover, this strategy reduces the problem of cold starts of container-based scheduling by reusing the active containers for remaining microservices. Finally, we have designed a new auto-scaling strategy for scaling out/down the computing resources based on the resource utilization of the PMs and the resources requested by the non-warm containers for processing the microservices. The main aim of the proposed strategy is to minimize the overall cost of an application and to utilize the resources efficiently with a minimum number of PMs.

3. System model and problem formulation

This section describes an application deployment model using a container-based virtualization technique for processing microservices in PMs. The main aim of this model is to minimize the overall operational costs of the microservices while satisfying various QoS requirements. Next, a container-based auto-scaler framework has been designed for scaling out/down the physical resources horizontally based on the resource requirements of the microservices and the resource availability in the active PMs. Finally, the objectives of the work have been formulated mathematically at the end of this section.

3.1. Application deployment model

This section is devoted to introducing a container-based application model in the cloud environment for processing microservices in optimal time and minimum cost which is depicted in Fig. 1. The *Cloud Manager* is responsible for processing various interactivity before deploying the microservices to suitable PMs using containers. The *Admission Controller* of the cloud manager receives the microservices from the users and checks the availability of the resources in the PMs. The Application Scheduler receives the application from the admission controller and stores it in a queue. Further, the *Application Scheduler* finds a suitable scheduling strategy to deploy the microservices to the *Resource Provisioner* for finding the best-fit containers for processing. The *SLA Manager* checks the QoS constraints of the applications while processing in PMs such as CPU and memory requirements, budget constraints, etc. During processing, if any violation occurs, the *SLA Manager* immediately pauses the current microservice without affecting other applications. The *Resource Provisioner* is responsible for selecting the best-fit container from the container pool for each microservice based on its CPU and memory requirements. The *Resource Provisioner* is also responsible for finding the warm containers (the reusable containers for next microservices) in the PMs for the next set of microservices in the *Admission Controller*. The *Container-based Auto-Scaler* is mainly responsible for monitoring the current status of the resources in the PMs based on *Resource Monitor*. The *Auto-Scaler* is scaling out/down the PMs based on their resource availability and the availability of the warm containers. The components of the *Auto-Scaler* are discussed in the next section. Finally, the *Resource Management* finds suitable PMs for the containers and deploys the containers.

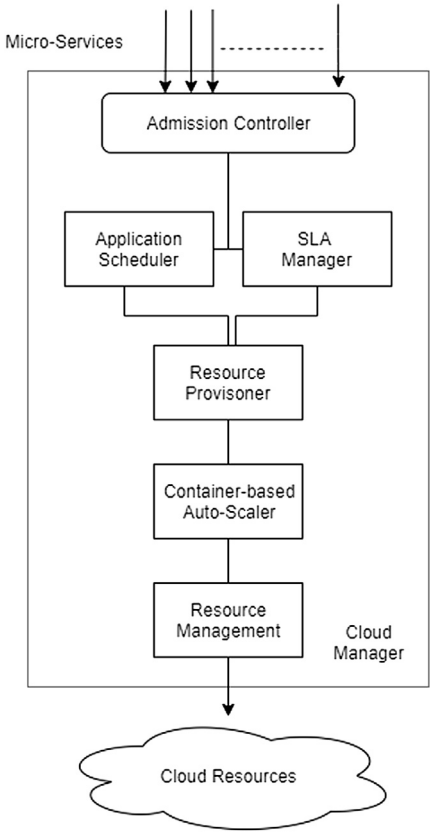

Fig. 1. Architecture for deploying microservice-based applications.

3.2. Container-based auto-scaler

This section describes an automatic tool for scaling the resources horizontally as a form of PMs/VMs in a data center. The components of the container-based auto-scaler are shown in Fig. 2. The data center contains n numbers of PMs which are represented as $P = \{P_1, P_2, P_3, \ldots, P_n\}$. Each PM consists of m number of containers based on the resource availability of the PMs which are denoted as $C = \{C_1, C_2, C_3, \ldots, C_m\}$. The *Container Manager* of each PM is responsible to host a lightweight container over the host operating system and shares the resources as per their requirements. The *Resource Monitor* of the auto-scaler receives information about the current usage of the resources (CPU and memory usage) in the PMs after each time interval and calculates the resource availability. The *Resource Monitor* also receives information from the resource provisioner of the *Cloud Manager* about the number of warm

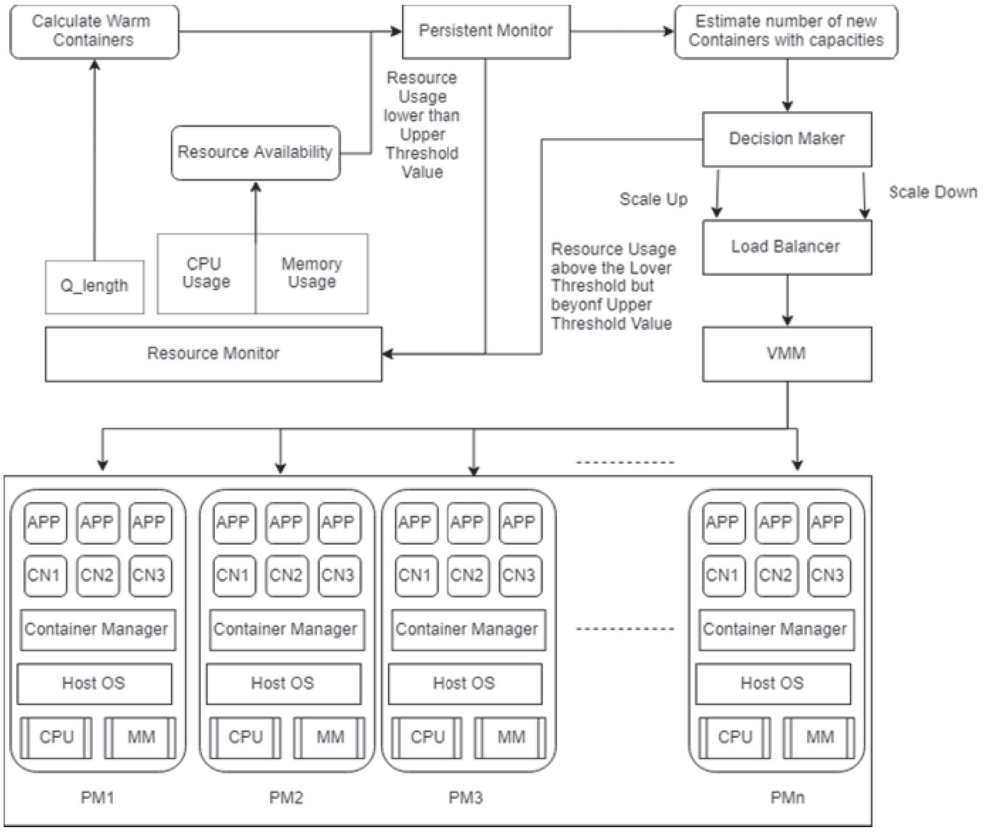

Fig. 2. The framework of the container-based Auto-Scaler.

containers. This helps to calculate the number of resources required further for processing the remaining microservices which are not using the warm containers. This should be done by the *Persistent Monitor* at a regular time interval and finds out the non-warm containers required for processing current microservices. This strategy is helpful for reducing the cold start problems for container-based scheduling. Based on the required resources of the non-warm containers and the current resource availability of the resources in the PMs, the *Decision Maker* decides whether scaling out/down the PMs. After fixing the PMs, the *Load Balancer* finds the optimal loaded PMs for the containers using a dynamic Bin Packing strategy and balances the loads among the active PMs. This policy reduces the number of PMs for processing microservices. Finally, the *Virtual Machine Manager* (VMM) deploys the containers on the selected PMs in the First Come First Serve basis.

3.3. Problem formulation

Let us consider that a PM/VM consists of a limited amount of computing resources as a form of CPU and memory. Here, we assume that the PMs have a heterogeneous amount of resources with a different cost which depends on the amount of resource usage by the applications. Moreover, the PMs can install different types of library functions in advance for processing the microservices. The admission controller receives a set of applications from the users where each application is represented by a set of a loosely coupled set of microservices, represented as $M = \{M_1, M_2, M_3, \ldots, M_k\}$. Each microservice of an application should complete a specific subfunction independently for that application and communicates with other microservices for data, if required. A microservice $l; l \in M$ considers a set of tuples ($CP^{CPU}{}_l$, $CP^{MM}{}_l$, BC_l), where $CP^{CPU}{}_l$ denotes the amount of requested CPU, $CP^{MM}{}_l$ denotes the amount of requested memory, and BC_l denotes the budget constraints of the microservice l. The resource provisioner finds a suitable container that should meet the CPU and memory requirements of the microservice.

The CPU and memory capacity of a container depends on the number of cores and blocks occupied by that container of a PM. The CPU capacity of the container i (CP_i^{CPU}) is defined as follows.

$$CP_i^{CPU} = |CR_i| \times sizeof(CR) \tag{1}$$

where ($|CR_i|$) is the number of cores and *sizeof*(CR) denotes the size of each core. Similarly, the memory capacity of the container i (CP_i^{MM}) is defined as follows.

$$CP_i^{MM} = |M_i| \times sizeof(M) \tag{2}$$

where ($|M_i|$) is the number of blocks in the memory and *sizeof*(M) denotes the size of each block. The processing time (PT_{li}) of microservice l is defined as the ratio between the size of the microservice and the CPU capacity of the container C_i which is given below.

$$PT_{li} = \frac{SZ_l}{CP_i^{CPU}} \tag{3}$$

where SZ_l has represented the size of the microservice. The processing cost of a microservice l for CPU usage depends on the CPU capacity of the selected containers of type-i and the processing time of the microservice l, which is represented as follows.

$$CO_{il}^{CPU} = CP_i^{CPU} \times \frac{PT_{li} \times CO_i^{CPU}}{\tau_1} \tag{4}$$

where CO_i^{CPU} represents the CPU usage cost of a container of type-i for a unit time interval τ_1. Similarly, the processing cost of a microservice l for memory usage depends on the memory capacity of the selected containers of type-i and the processing time of the microservice l, which is represented as follows.

$$CO_{il}^{MM} = CP_i^{MM} \times \frac{PT_{il} \times CO_i^{MM}}{\tau_2} \quad (5)$$

where CO_i^{MM} represents the memory usage cost of a container of type-i for a unit time interval τ_2. So, the total processing cost of a microservice l in a container of type-i is represented as follows.

$$CO_{il} = CO_{il}^{CPU} + CO_{il}^{MM} \quad (6)$$

Let, an application A_a contains z number of microservices for processing at a certain time interval t and the deployment time of each container is considered as DT_i. As a result, the cost of deploying each container of type-i is defined as follows.

$$CDT_i = \frac{DT_i \times CO_i^{DT}}{\tau_3} \quad (7)$$

where CO_i^{DT} represents the cost of deploying a container of type-i in a PM/VM for a unit time interval τ_3. Note that, $DT_i = 0$ for the warm containers type-i. So, the total processing cost of the application A_a is represented as follows.

$$CO_a = \sum_{l=1}^{z} CO_{il} + \sum_{i=1}^{k} CDT_i : where\ i = 1, 2, 3, \ldots, m \quad (8)$$

where we consider that k number of containers are required to process an application A_a. Another important issue of this work is to balance the load among the PMs and utilizes the resources efficiently. Here, we consider that the current load of a PM j depends on two computing resources i.e. CPU and memory. The current load of the PM j at a time frame t is defined as follows.

$$L_j(CPU, MM) = \beta \times \sum_{i=1}^{m} CP_i^{CPU}(t) + (1-\beta) \times \sum_{i=1}^{m} CP_i^{MM}(t) : where\ i = 1, 2, 3, \ldots, m \quad (9)$$

where β is the constant whose values lie in [0, 1]. The value of $CP_i^{CPU}(t)$ and $CP_i^{MM}(t)$ lie in the interval [0,100]. The available load of a PM j is represented as $RL_j(CPU,MM) = (1 - L_j(CPU,MM))$. For a given PM j, the resource utilization (RU_j) is calculated as follows.

$$RU_j = \frac{L_j(CPU, MM)}{L_j^{max}(CPU, MM)} \times 100 \quad (10)$$

where $L_j^{max}(CPU, MM)$ indicates the maximum resource capacity of the PM j. The main intention of this work is to minimize the overall processing cost of the microservice-based applications and utilize the resources efficiently while meeting various QoS constraints. The mathematical formulation of this work is given below.

Minimize CO_a
Maximize RU_j
Subject to
$CO_a \leq BC_a$ ..... (i)
$RU_j \leq 100$ ..... (ii)
$\sum_{i=1}^{m} RU_i \leq RU_j$ ..... (iii)
$L_j(\text{CPU}, \text{MM}) \leq L_j^{max}(CPU, MM)$ ..... (iv)
$x_{ik} \in \{0, 1\}; \quad i \in C_M, \quad k \in M_k$ ..... (v)
$\sum_{i=1}^{m} x_{ij}; \quad i \in C_l, \quad j \in P_n$ ..... (vi)

Constraint (i) indicates the budget constraint of an application a. This indicates that the overall processing cost of an application must not be beyond the budget constraint BC_a. Constraint (ii) indicates that the overall resource utilization of a PM j should not be beyond its maximum limit of 100%. Let us consider a PM j contains m number of containers. So, the overall resource utilization of m number of containers should not be beyond the resource utilization of the assigned PM j which is depicted in constraint (iii). Constraint (iv) indicates that the overall load of the PM j should not be beyond its load. Each microservice of an application should be assigned to a single container which is depicted in Constraint (v). Constraint (vi) indicates that multiple containers are assigned and run on a single PM j.

4. Proposed algorithm

This section describes a new container-based resource provisioning strategy for microservices with an automatic auto-scaling strategy. The proposed algorithm is divided into four stages, i.e. resource provisioning, reusability of containers, container-PM mapping and auto-scaling strategy. In the resource provisioning phase, a suitable container is selected for each microservice based on its resource requirements. In the second phase, the warm containers are selected for reusing the set of active and ideal containers in the PMs which should reduce the cold start problem. In the container-PM mapping phase, the set of selected containers are mapped to the suitable PMs based on the dynamic bin-packing strategy. Finally, the auto-scaling strategy is used to verify that whether the available resources are enough for processing the selected containers or the resources of the PMs should be scaled up/down. The detail discussions of those phases are given below.

4.1. Resource provisioning

This phase is used to find the perfect matching between microservices and containers using a heuristic strategy. Here, we consider that a pool of containers is available in the data center with a specific CPU and memory capacity. The heuristic strategy finds the suitable containers for each microservice such that the resource wastage of the container should be minimum and meets the requested load of that microservices. A load of each container type-i is calculated as follows.

$$L_i(CPU, MM) = \beta \times CP_i^{CPU}(t) + (1-\beta) \times CP_i^{MM}(t) \quad (11)$$

Similarly, the load requested by a microservice l is defined as follows.

$$L_l(CPU, MM) = \beta \times CP_l^{CPU}(t) + (1-\beta) \times CP_l^{MM}(t) \quad (12)$$

The given heuristic finds a container based on the residual load of each microservice over the available containers which is calculated as follows.

$$CM_{il} = \frac{L_i(CPU, MM)}{L_l(CPU, MM)} \quad (13)$$

Each microservice l deploys on a suitable container of type-i based on its residual load, i.e. a microservice l is assigned to the container of type-i whose value greater than or equal to one and the selected container has sufficient CPU and memory capacity for processing the microservice l. The selection of a container for each microservice is shown in Step 8 to 14 of Algorithm 1. For illustration, we consider eight microservices with a different set of resources and a pool of five containers with different resource usage. The load requested by the microservices and an available load of each container are shown in Table 1. Note that, we used $\beta = 0.5$ to simplify the calculation of the illustration. In the experimental evaluation, we first normalize the CPU and memory capacity of the requested applications and distribute the equal weight among the resources, i.e. $\beta = 0.5$ for finding the load capacity.

The residual load of the microservices over various containers are shown in Table 2. For example, the microservice MS_1 has a minimum residual load at container C_1 and the container meets the QoS constraints of the microservice such as CPU requirement, memory requirement and load requested. However, for microservice MS_6, the container C_2 provides minimum residual load and meets the memory requirement and

Table 1
Resource usage by microservices and containers at.$\beta = 0.5$

Microservices				Containers			
id	CPU requirements (in Cores)	Memory requirements (in Blocks)	Load Requested (Using Eq. (11))	id	CPU Usage (in Cores)	Memory Usage (in Blocks)	Load Usage (Using Eq. (10))
MS_1	1 EC2	2 B	1.5	C_1	2 EC2	4 B	3
MS_2	2 EC2	3 B	2.5	C_2	4 EC2	8 B	6
MS_3	1 EC2	4 B	2.5	C_3	6 EC2	14 B	10
MS_4	4 EC2	3 B	3.5	C_4	8 EC2	18 B	13
MS_5	3 EC2	8 B	5.5	C_5	12 EC2	24 B	18
MS_6	8 EC2	4 B	6				
MS_7	6 EC2	3 B	4.5				
MS_8	4 EC2	10 B	7				

Table 2
Residual load of microservices (Bold represents selected containers for the microservices; Normal represents other possible container types; Italic represents non suitable mapping).

	C_1	C_2	C_3	C_4	C_5
MS_1	**2**	4	6.67	8.67	12
MS_2	**1.2**	2.4	4	5.2	7.2
MS_3	**1.2**	2.4	4	5.2	7.2
MS_4	*.86*	**1.7**	2.86	3.7	5.1
MS_5	*.55*	**1.09**	1.82	2.36	3.27
MS_6	*.50*	*1*	*1.67*	**2.17**	3
MS_7	*.67*	*1.3*	**2.2**	2.89	4
MS_8	*.43*	*.86*	**1.43**	1.86	2.57

requested load capacity, but at the same time, the container C_2 fails to meet the CPU requirements. The same situation happens for container C_3. As a result, the microservice MS_6 should be deployed on container C_4.

The detailed matching between the microservices and containers is shown in Fig. 3. The pseudocode of the resource provisioning is depicted in Fig. 4. This strategy reduces the overall cost of an application by selecting the containers with minimum resource capacity for the microservices and meets the budget constraint.

Theorem 1. The total time complexity of the resource provisioning strategy is $O(mk)$.

Proof. The time complexity of the proposed resource provisioning strategy depends on the number of containers, *i.e.* m and the number of requested microservices, *i.e.* k at time t. Step 2 to 4 are responsible for finding the load capacity of m number of containers which makes the complexity $O(m)$. Similarly, Step 5 to 7 finds the requested load capacity of k number of microservices which makes the complexity $O(k)$. Finally, Step 8 to 16 is responsible for mapping k number of microservices to the suitable m number of containers which makes the complexity $O(mk)$. As a result, the overall complexity of the resource provisioning strategy is $O(m)+O(m)+O(mk) = O(m+m+mk) = O(2\,m+mk)$. Now, $mk \gg m$, so the overall time complexity of the resource provisioning strategy is $O(mk)$.

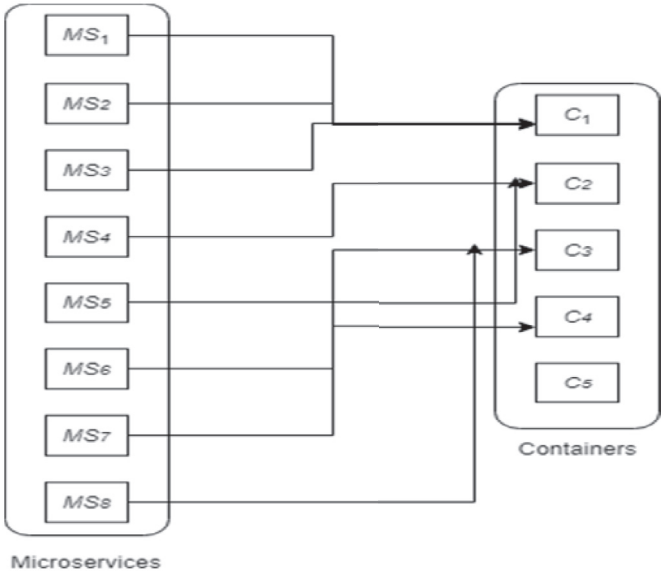

Fig. 3. Microservices to containers mapping.

4.2. Reusability of containers

This phase is used to find the reusable containers for processing further microservices waited in the queue, which are also called warm containers. Here, a rule-based strategy is used to find a set of warm containers. Let us consider there are m containers, where the CPU and memory capacity of the containers are represented by CP_i^{CPU} and CP_i^{MM}. The containers are arranged in a sorted order according to the CPU and memory capacity, in that order.

Rule1: if $((CP_l^{CPU} \leq CP_i^{CPU})\&\&(CP_l^{MM} \leq CP_i^{MM})\&\&(WT_i == Null) \parallel ((CP_l^{CPU} \leq CP_i^{CPU})\&\&(CP_l^{MM} \leq CP_i^{MM})\&\&(WT_i \leq \theta))$: If the running container C_i has the resource capacity less than or equal to the resource capacity is requested by the microservice l or the container is ideal and the waiting time (WT_i) of the container is less than or equal to a threshold time (θ) in a PM, then the container C_i is considered as a warm container and is assigned for microservice l for further processing.

Rule 2: if $((CP_l^{CPU} > CP_i^{CPU})\&\&(CP_l^{MM} \leq CP_i^{MM}))$: If the container C_i has the memory capacity less than or equal to the memory capacity is requested by the microservice l, but the container does not meet the CPU requirement then the container C_i is not considered as a warm container and is not assigned for microservice l for further processing.

Rule 3: if $((CP_l^{CPU} \leq CP_i^{CPU})\&\&(CP_l^{MM} > CP_i^{MM}))$: If the container C_i has the CPU capacity less than or equal to the CPU capacity is requested by the microservice l but the container does not meet the memory requirement then the container C_i is not considered as a warm container and is not assigned for microservice l, for further processing.

Rule 4: if $((CP_l^{CPU} \leq CP_i^{CPU})\&\&(CP_l^{MM} \leq CP_i^{MM})\&\&(WT_i > \theta))$: If the container C_i has the resource capacity less than or equal to the resource capacity is requested by the microservice l and the container is ideal, but the waiting time of the container is more than a threshold time (θ) in a PM, then the container C_i is destroyed and is not assigned for microservice l for further processing.

Rule 5: if $((CP_l^{CPU} \leq CP_i^{CPU})\&\&(CP_l^{MM} \leq CP_i^{MM})\&\&(WT_i == Null))$: If the container C_i has the resource capacity less than or equal to the resource capacity is requested by the microservice l, but the container is currently running for processing a microservice l, then the possible condition for selecting a container is a warm container or not is given below.

Rule 5.1: if $((RT_{gi} + PT_{li}) \leq (DT_{li} + PT_{li}))$ *where* $g \neq l$: If the summation of the remaining processing time (RT_{gi}) of the assigned

Algorithm 1: Resource Provisioning

Input: k number of microservices with CPU (CP_l^{CPU}) and memory (CP_l^{MM}) requirements
n number of containers with CPU (CP_i^{CPU}) and memory usage (CP_i^{MM})

Output: Find the best-fit Container i for each microservice l

1. **Begin**
2. **For** $i = 1$ to n
3. Calculate the resource load of each container:
$$L_i(CPU, MM) = \beta \times CP_i^{CPU}(t) + (1-\beta) \times CP_i^{MM}(t)$$
4. **End For**
5. **For** $l = 1$ to k
6. Calculate the resource requested by each microservice:
$$L_l(CPU, MM) = \beta \times CP_l^{CPU}(t) + (1-\beta) \times CP_l^{MM}(t)$$
7. **End For**
8. **For** $l = 1$ to k
9. **For** $i = 1$ to n
10. Calculate the residual load of each microservice:
$$CM_{il} = \frac{L_i(CPU,MM)}{L_l(CPU,MM)}$$
11. **If** $((CP_l^{CPU} \leq CP_i^{CPU})$ && $(CP_l^{MM} \leq CP_i^{MM})$ && $CM_{il} \geq 1)$
12. Find: ***MIN*** (CM_{il})
13. **End If**
14. **End For**
15. Select the best-fit Container i for each microservice l
16. **End For**
17. **End**

Fig. 4. Pseudocode of resource provisioning strategy.

microservice g and the processing time (PT_{li}) of microservice l in the container C_i is less than or equal to the total of the deployment time ((DT_{li}) and the processing time of microservice l in the container C_i, then the container C_i is considered as a warm container and reuse for microservice l.

Rule 5.2: if $((RT_{gi} + PT_{li}) > (DT_{li} + PT_{li}))$*where* $g \neq l$: If the total of the remaining processing time of the assigned microservice g and the processing time of microservice l in the container C_i is greater than to the total of the deployment time and processing time of microservice l in the container C_i, then the container C_i is not considered as a warm container and should not be reused for microservice l.

The remaining processing time (RT_{gi}) of the assigned microservice g in the container C_i is defined as the difference between the processing time of the microservice g and the running processing time (APT_{gi}) of that microservice g in the container C_i which is defined below.

$$RT_{gi} = \left(ET_{gi} - APT_{gi}\right) \tag{14}$$

The pseudocode of the reusability of the container is shown in Fig. 5. This phase finds the number of warm containers of reusable for processing the next set of microservices of an application. This reduces the deployment time of the container and the overall processing time of an application. So, this phase is used for optimizing the cold start problem of the containers in the cloud environment.

Theorem 2. The total time complexity of the reusability of container strategy is $O(mk)$.

Proof. The time complexity of the proposed reusability of container strategy also depends on the number of warm containers m available at time t for processing k number of microservices in PMs/VMs. Step 2 to 22 are responsible for finding the warm containers for k number of microservices, which makes the complexity is $O(mk)$. However, Step 23 to 29 calculates the total number of warm containers available at time t for processing k number of requested microservices which makes the complexity $O(1)$. As a result, the total time complexity of the container strategy is $O(mk) + O(1) = O(mk+1) = O(mk)$.

4.3. Container-PM/VM mapping

This phase is mainly responsible for deploying the tasks to the active PMs/VMs based on the dynamic bin packing strategy. The selected containers for processing the microservices wait in the global queue of the resource manager. The global queue of the data center uses a bounded queueing model with a size of N. Here, we consider time interval between the arrival rate of the microservices are exponentially distributed, *i.e.* Poisson Process and the arrival rate of two independent microservices depend on the density function $f(x) = \lambda e^{-\lambda x}$ for a fixed constant λ, also known as the arrival rate of the microservices. The resource utilization of the PM/VM is represented as, $\rho = \frac{\lambda}{n\mu}$, where n represents the number of active PMs and μ represents the service rate. The probability that all PMs are idle at certain timestamp is defined as follows.

$$P_0 = \frac{1}{\sum_{j=1}^{n} \frac{\left(\frac{\lambda}{\mu}\right)^m}{m!} + \frac{\left(\frac{\lambda}{\mu}\right)^n}{n!}\left(\frac{1}{1-\frac{\lambda}{n\mu}}\right)} \tag{15}$$

The probability for finding a PM is idle at a certain timestamp is defined as follows.

Algorithm 2: Reusability of containers

Input: k number of microservices with CPU (CP_l^{CPU}) and memory (CP_l^{MM}) requirements
m number of containers with CPU (CP_i^{CPU}) and memory usage (CP_i^{MM})

Output: Find the warm containers, Number of warm containers: $Count_1$

Begin
For $l = 1$ to k
For $i = 1$ to m
If$((CP_l^{CPU} \leq CP_i^{CPU})$ && $(CP_l^{MM} \leq CP_i^{MM})$ && $(WT_i == Null)$ || $((CP_l^{CPU} \leq CP_i^{CPU})$ && $(CP_l^{MM} \leq CP_i^{MM})$ && $(WT_i \leq \theta))$:
Match microservice l to container i
Update the container list
If$((CP_l^{CPU} > CP_i^{CPU})$ && $(CP_l^{MM} \leq CP_i^{MM}))$
Matching is not possible
If$((CP_l^{CPU} \leq CP_i^{CPU})$ && $(CP_l^{MM} > CP_i^{MM}))$
Matching is not possible
If$((CP_l^{CPU} \leq CP_i^{CPU})$ && $(CP_l^{MM} \leq CP_i^{MM})$ &&$(WT_i > \theta))$
Matching is not possible
If$((CP_l^{CPU} \leq CP_i^{CPU})$ && $(CP_l^{MM} \leq CP_i^{MM})$&& $(WT_i == Null))$
If$((RT_{gi} + PT_{li}) \leq (DT_{li} + PT_{li}))$ $where\ g \neq l$
Match microservice l to container i
Update the container list
Else
Matching is not possible
End If
End If
End For
End For
Generate the updated list of containers with microservices matching
If(All existing containers are matched with a new set of microservices)
Called as warm containers: No necessity to create new containers
Else
Create a new set of containers for remaining microservices
End If
End

Fig. 5. Pseudocode of reusability of container.

$$P_j = \begin{cases} \frac{\left(\lambda/\mu\right)^j}{j!} P_0 \ where\ j = 1, 2, 3, \ldots, n \\ \frac{\left(\lambda/\mu\right)^j}{n! n^{j-n}} P_0 \ where\ j = n+1, n+2, \ldots \end{cases} \quad (16)$$

The PMs are used for storing the microservices with the selected containers before processing. Here, we find a strategy to fix the length of the local queue (*lque_len*) of each PM after its activation in the data center. The $lque\_len_j$ of the PM j depends on the resource capacity of that PM and the capacity of the containers which is defined as follows.

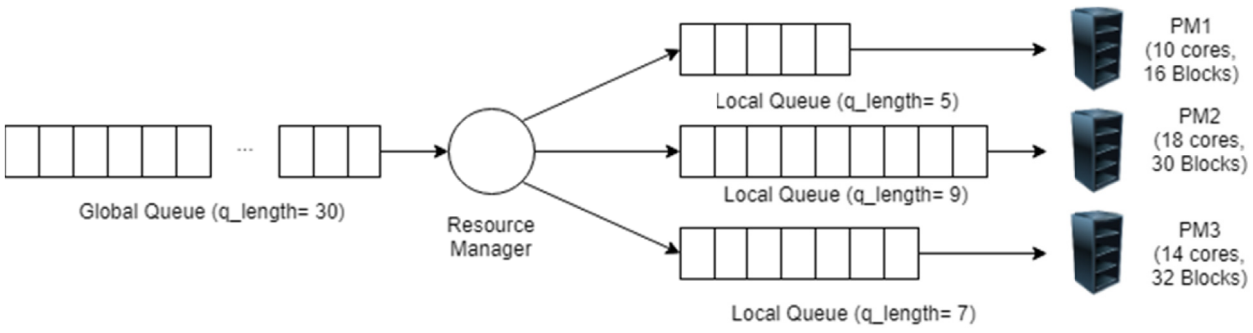

Fig. 6. Queueing model for Container-PM mapping.

$$lque\_len_j = MAX\left(\frac{CP_j^{CPU}}{MIN\left(CP_i^{CPU}\right)}, \frac{CP_j^{MM}}{MIN\left(CP_i^{MM}\right)}\right) \quad (17)$$

For example, let's consider, three PMs with resource capacity P_1= (10 cores CPU, 16 Blocks Memory), P_2= (18 cores CPU, 30 Blocks Memory), P_3= (14 cores CPU, 26 Blocks Memory) and the capacity of the containers are listed in Table 1. So, the local queue length of the PM_1 is, $lque\_len_1 = MAX\left(\frac{10}{MIN(2,\ 4,\ 6,\ 8,\ 12)}, \frac{16}{MIN(4,\ 8,\ 14,\ 18,24)}\right) = MAX\left(\frac{10}{2}, \frac{16}{4}\right) = 5$. Similarly, the local queue length of PM_2 and PM_3 are $lque\_len_2 = 9$ and $lque\_len_3 = 7$ respectively. The local queue lengths of the PMs are shown in Fig. 6. The dynamic bin-packing algorithm is used to minimize the total number of PMs for processing the microservices. So, the main objective of this phase is formulated below.Where,

$$P_j = \begin{cases} 1 & if\ \ PM\ \ j\ \ is\ \ used \\ 0 & Otherwise \end{cases}$$

$$X_{ij} = \begin{cases} 1 & if\ \ container\ \ i\ \ is\ \ assigned\ \ to\ \ PM\ \ j \\ 0 & Otherwise \end{cases}$$

To adopt the above-mentioned objective, we design a dynamic bin-packing strategy for finding a suitable PM/VM for the selected containers based on their resource requirements. Here, we consider the active set of PMs/VMs as bins with the weight of CPU and memory capacity and a modified version of the dynamic bin-packing strategy is designed for meeting the QoS parameters. Our modification is that once a bin, *i.e.* PM/VM is overloaded at time *t*, no more containers are further deployed on that bin until the capacity of the bin below the threshold value. Initially, all the active PMs/VMs are sorted in ascending order according to the available CPU and memory capacity. Each selected container *i* is deployed to a PM/VM *j* from the sorted list which has the available CPU and memory capacity less than or equal to the requested resource capacity of the selected container *i* and has minimum available resources to deploy. As a result, the proposed dynamic bin-packing strategy reduces the total number of PMs/VMs and minimizes resource wastage.

For example, the selected containers for eight microservices (in Table 2) should be mapped to the three PMs which are shown in Fig. 6. Let us container, C_1 can be assigned any of the three available PMs, however, the PM P_1 has enough CPU and memory capacity to hold the container C_1 and has minimum resources available as compared with the other two PMs. As a result, the container C_1 is mapped to PM P_1. Remaining containers are mapped to the suitable PMs by maintaining the same procedure. The detailed mapping of the containers and PMs are shown in Table 3. The pseudocode of the Container-PM mapping is shown in Fig. 7. In the pseudocode, ACP_j^{CPU} and ACP_j^{MM} represent the available CPU and memory capacity of PM *j*. Initially, the value of ACP_j^{CPU} and ACP_j^{MM} are set as the original CPU and memory capacity of the PM *j*.

Theorem 3. The total time complexity of the Container-PM Mapping strategy is $O(nk(log\ n))$.

Proof. The time complexity of the proposed Container-PM Mapping strategy depends on the number of selected containers, *i.e.* *k* for processing the requested *k* number of microservices and available sets of PMs/VMs in the data center, *i.e.* *n*. Step 5 sorts the PMs in ascending order according to their resource capacity, so the time complexity for this step is $O(n\ log\ n)$. Now, each microservice is deployed on the suitable PM from the sorted list. Thus, the total time complexity to deploy *k* number of microservices is $O(nk(log\ n))$ (refer to Step 3 to 13 of Algorithm 3).

4.4. Auto-scaling strategy

The container-based auto-scaling strategy requires two types of information for scaling out/down the resources in the PMs, i.e. the number of non-warm containers required for processing next set of microservices and the percentage of resources that are available in the active PMs/VMs. Taking all into consideration, the auto-scaler makes a decision about the scale-out or scale-down the resources based on the resource monitoring strategy. Here, the auto-scaling strategy consists of two types of threshold values including lower threshold value for CPU (LT_j^{CPU}) and memory (LT_j^{MM})) and the upper threshold value for CPU (UT_j^{CPU}) and memory (UT_j^{MM})). The threshold values of each PM/VM are calculated based on the percentage of the resource utilization such as the resource usage beyond the (UT_j^{CPU}) or (UT_j^{MM}) is considered as the overprovision of the PM/VM *j* and the resource usage lower than the (LT_j^{CPU}) or (LT_j^{MM}) is considered as the underprovision of the PM/VM *j*. The pictorial presentation of the resource utilization status and current loads of the PMs/VMs are shown in Fig. 8. The resource utilization status lower than underprovision requires scaling down the PMs, however, the resource utilization beyond the over-provision requires scaling out one or more PMs for processing the microservices with non-warm containers. The auto-scaling strategy consists of two phases, namely monitoring mechanism, and decision mechanism respectively which are elaborately discussed in the following subsections.

Similarly, memory utilization of *j*th PM/VM with $x, x \in C$ number of containers at a certain timestamp *t* is defined as follows

$$R_j^{MM}(t) = \frac{\sum_{i=1}^{x} CP_i^{MM}}{CP_j^{MM}} : x = 1, 2, 3, \ldots; x \in C; \quad (19)$$

The main responsibility of the monitoring strategy is to count the total number of active PMs/VMs which are either overprovisioned or underprovisioned at time *t* (refer to Step 9 to 14 of Algorithm 4). The monitoring strategy calls the decision mechanism phase of the proposed auto-scaling strategy at time *t* if half of the PMs/VMs loads are above the upper threshold values (presented in Step 15 to 16 of Algorithm 4) or below the lower threshold values (presented in Step 17 to 16 of Algorithm 18). The pseudocode of the monitoring mechanism is shown in Fig. 9.

Theorem 4. The total time complexity of the monitoring mechanism is $O(nx)$.

Proof. The time complexity of the proposed monitoring mechanism depends on the CPU and memory utilization of *x*, $x \in C$ number of containers, deployed on each PM/VM for processing microservices. Step 4 to

Table 3
Container-PM mapping.

Containers	Container to PM Mapping					
	P_1 (10 Cores, 16 Blocks)	Available Resources	P_2 (18 Cores, 30 Blocks)	Available Resources	P_3 (14 Cores, 32 Blocks)	Available Resources
C_1 (2 Cores, 4 Blocks)	C_1	(8 Cores, 12 Blocks)	–	(18 Cores, 30 Blocks)	–	(14 Cores, 32 Blocks)
C_1 (2 Cores, 4 Blocks)	C_1	(6 Cores, 8 Blocks)	–	(18 Cores, 30 Blocks)	–	(14 Cores, 32 Blocks)
C_1 (2 Cores, 4 Blocks)	C_1	(4 Cores, 4 Blocks)	–	(18 Cores, 30 Blocks)	–	(14 Cores, 32 Blocks)
C_2 (4 Cores, 8 Blocks)	–	(4 Cores, 4 Blocks)	C_2	(12 Cores, 22 Blocks)	–	(14 Cores, 32 Blocks)
C_2 (4 Cores, 8 Blocks)	–	(4 Cores, 4 Blocks)	C_2	(8 Cores, 14 Blocks)	–	(14 Cores, 32 Blocks)
C_4 (8 Cores, 18 Blocks)	–	(4 Cores, 4 Blocks)	–	(8 Cores, 14 Blocks)	C_4	(6 Cores, 14 Blocks)
C_3 (6 Cores, 14 Blocks)	–	(4 Cores, 4 Blocks)	C_3	(2 Cores, 0 Blocks)	–	(6 Cores, 14 Blocks)
C_3 (6 Cores, 14 Blocks)	–	(4 Cores, 4 Blocks)	–	(2 Cores, 0 Blocks)	C_3	(0 Cores, 0 Blocks)

Algorithm 3: Container-PM/VM Mapping

Input: k number of containers for k microservices with CPU (CP_i^{CPU}) and memory usage (CP_i^{MM})
m number of PMs with CPU (CP_j^{CPU}) and memory usage (CP_j^{MM})

Output: Find the best-fit PM j for each Container i

1. **Begin**
2. $ACP_j^{CPU} = CP_j^{CPU}$; $ACP_j^{MM} = CP_j^{MM}$;
3. **For** $i = 1$ to k
4. **For** $j = 1$ to n
5. Sort the PMs using the available CPU and memory capacity
6. **If** $((ACP_j^{CPU} \leq CP_i^{CPU})$ && $(ACP_j^{MM} \leq CP_i^{MM}))$
7. Assign Container i to PM j
8. $ACP_j^{CPU} = (ACP_j^{CPU} - CP_i^{CPU})$
9. $ACP_j^{MM} = (ACP_j^{MM} - CP_i^{MM})$
10. **Break**
11. **End If**
12. **End For**
13. **End For**
14. **End**

Fig. 7. Pseudocode of Container-PM Mapping strategy.

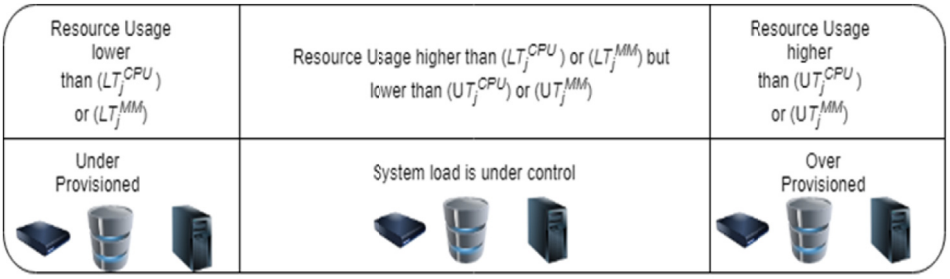

Fig. 8. Resource usage status and current load of PMs/VMs.

A. **Monitoring Mechanism:** The Monitoring mechanism receives information about the current resource usage of the PMs/VMs in a regular time interval and identifies the total number of PMs/VMs that are underprovisioned or overprovisioned. The CPU utilization of jth PM/VM with $x, x \in C$ number of containers at a certain timestamp t is defined as follows.

$$R_j^{CPU}(t) = \frac{\sum_{i=1}^{x} CP_i^{CPU}}{CP_j^{CPU}} : x = 1, 2, 3, \ldots; x \in C; \tag{18}$$

8 are responsible for finding the CPU and memory utilization of a PM/VM with x number of containers, which makes the complexity $O(x)$. However, the time complexity for finding the resource utilization of n number of PMs/VMs is $O(nx)$. However, Step 9 to 14 are responsible for counting the total numbers of PMs/VMs are overprovisioned or underprovisioned, which makes the complexity $O(n)$. Step 15 to 31 takes the decision whether to call the decision mechanism phase of the auto-scaling strategy or the load of the data center is under control, which makes the complexity $O(1)$. Thus, the total time complexity of monitoring mechanism is $O(nx) + O(n) + O(1) = O(nx + n + 1) = O(nx)$, as $nx \gg n$.

B. Decision Mechanism: The decision mechanism finally decides whether to scale out or scale down the PMs as per the resource requirements by the non-warm containers and the information received from the monitoring mechanism. The decision mechanism of the proposed auto-scaling strategy waits until $(t + \Delta t : where\ \Delta t \ll t)$ times before taking a scaling out/down decision. After $(t + \Delta t)$ time duration, the decision mechanism further triggers the resource utilization of the active set of PMs/VMs in terms of CPU and memory utilization. During this time period, if still more than $\frac{n}{2}$ a number of PMs/VMs are overprovisioned or underprovisioned, the decision mechanism, phase takes either scale-out or scale-down decision respectively. If more than $\frac{n}{2}$ number of PMs/VMs are overprovisioned, then the decision mechanism scale-out the PMs/VMs as per the requirements of the waiting containers and if more than $\frac{n}{2}$ a number of PMs/VMs are underprovisioned, then the decision mechanism scale down at least 1 p.m./VM at each timestamp t. The pseudocode of the decision mechanism is shown in Fig. 10.

Theorem 5. The total time complexity of the decision mechanism is $O(n)$.

Proof. The time complexity of the proposed decision mechanism depends on the resource utilization of the active set of PMs/VMs at a time frame t. Similar to the persistent mechanism, the proposed decision mechanism finds the total number of PMs/VMs are overprovisioned or underprovisioned among the set of n number of PMs/VMs. So, the total time complexity of the proposed decision mechanism is $O(n)$.

The timeline of the proposed auto-scaling policy with a monitoring mechanism and decision mechanism is shown in Fig. 11. The monitoring mechanism monitors loads of the PMs/VMs and finds the total number of

Algorithm 4: Auto-Scaling- Monitoring Mechanism

Input: x number of containers with CPU (CP_i^{CPU}) and memory usage (CP_i^{MM})
n number of PMs with CPU (CP_j^{CPU}) and memory usage (CP_j^{MM})

Output: Check the status of the PMs

1. **Begin**
2. Set the lower and upper threshold value for CPU and Memory Utilization
   $LT_j^{CPU} = \emptyset_1$ and $LT_j^{MM} = \emptyset_2$; $UT_j^{CPU} = \theta_1$ and $UT_j^{MM} = \theta_2$
3. **Initialize:** $Count_2 = 0$; $Count_3 = 0$
4. **For** j = 1 to n
5. **For** i = 1 to x; $x \in C$
6. Calculate CPU Utilization at timestamp t
   $R_j^{CPU}(t) = \frac{\sum_{i=1}^{x} CP_i^{CPU}}{CP_j^{CPU}} : x = 1, 2, 3, \ldots; x \in C;$
7. Calculate memory Utilization at timestamp t
   $R_j^{MM}(t) = \frac{\sum_{i=1}^{x} CP_i^{MM}}{CP_j^{MM}} : x = 1, 2, 3, \ldots; x \in C;$
8. **End For**
9. **If** $((R_j^{CPU}(t) \geq \theta_1) \parallel (R_j^{MM}(t) \geq \theta_2))$
10. $Count_2$++ // Calculate total numbers of PMs/VMs are overprovisioned
11. **End If**
12. **If** $((R_j^{CPU}(t) < \emptyset_1) \parallel (R_j^{MM}(t) < \emptyset_2))$
13. $Count_3$++ // Calculate total numbers of PMs/VMs are underprovisioned
14. **End For**
15. **If** $((Count_2 \geq \lceil \frac{n}{2} \rceil) \&\& \textit{selected containers are not matched with warm containes})$
16. **Call** Decision Mechanism
17. **If** $(Count_3 \geq \lceil \frac{n}{2} \rceil)$
18. **Call** Decision Mechanism
19. **Else**
20. System Load is in Control
21. **End If**
22. **End**

Fig. 9. Pseudocode of monitoring mechanism.

PMs/VMs are overprovisioned or unprovisioned after each timestamp t. The monitoring mechanism calls the decision mechanism if more than $\frac{n}{2}$ a number of PMs/VMs are overprovisioned or underprovisioned otherwise the load of the overall data center is under control. The decision mechanism waits until $(t + \Delta t)$ before taking an optimal decision. After $(t + \Delta t)$, is still $\frac{n}{2}$ a number of PMs/VMs are overprovisioned or underprovisioned, then the decision mechanism Scaling out/down the PMs/VMs as per the requirements. This can utilize computing resources efficiently in the data center.

Theorem 6. The total time complexity of the proposed auto-scaling strategy is $O(nx)$.

Proof. The time complexity of the proposed auto-scaling strategy depends on a monitoring mechanism and decision mechanism. The complexity of the monitoring mechanism and decision mechanism are $O(nx)$, and $O(n)$ respectively. However, $nx \gg n$, so the overall time complexity of the proposed auto-scaling strategy is $O(nx) + O(n) = O(nx + n) = O(nx)$.

5. Performance analysis

In this section, we evaluate the performance of the proposed container-based auto-scaling strategy using a real trace from Google Cluster Tracelog (Reisset al., 2011; Wan et al., 2018). This file contains 25 million tasks that are submitted by 900 users over an observed period of a month. For the simulation purpose, four datasets have been generated from the Google Cluster Trace where each dataset consists of 5000 tasks (here, tasks are represented as microservices) with specific resource requirements and arrival rate for 5 h. To analyze the proposed method and existing state-of-the-art-algorithms over Google Cluster Trace, we further normalized the required CPU and memory capacity of the requested tasks and calculated the normalized load capacity of the tasks. The normalized CPU and memory capacity of the requested microservices along with their load is shown in Fig. 12.

From Fig. 12, it is shown that most of the microservices of Dataset 1, Dataset 2, and Dataset 3 requested 4 to 10 percent CPU capacity and memory capacity for processing, however, few microservices requested more CPU and memory usage which varies from 10 to 22 percent which increases the requested load capacity of the microservices. However, most of the microservices of Dataset 4 requested the CPU and memory capacity with the range of 5–7 percent which makes the requested load capacity of the microservices within the range of 5–7 percent as well. Here, each microservice should map to a suitable container for further processing using Algorithm 1. Finally, the selected containers are deployed on the available PMs/VMs using Algorithm 3. Initially, we considered 20 PMs/VMs for deploying the selected containers where the maximum resource capacity of the PMs/VMs are chosen randomly within the range of [40–100]. The simulation parameters for evaluating the

Algorithm 5: Auto-Scaling- Decision Mechanism

Input: n number of PMs with CPU ($R_j^{CPU}(t)$) and memory usage ($R_j^{CPU}(t)$)

Output: Check for Scaling out/down the PMs

1. **Begin**
2. **Initialize:** $Count_4 = 0;\ Count_5 = 0$
3. Wait until $(t + \Delta t)$
4. **For** j = 1 to n
5. **Calculate** $R_j^{CPU}(t)$ and $R_j^{MM}(t)$
6. **If** $((R_j^{CPU}(t) \geq \theta_1) \parallel (R_j^{MM}(t) \geq \theta_2))$
7. $Count_4$++ // Calculate total numbers of PMs/VMs are overprovisioned
8. **End If**
9. **If** $((R_j^{CPU}(t) \leq \emptyset_1) \parallel (R_j^{MM}(t) \leq \emptyset_2))$
10. $Count_5$++ // Calculate total numbers of PMs/VMs are underprovisioned
11. **End If**
12. **End For**
13. **If** $(Count_4 \geq \lceil \frac{m}{2} \rceil)$
14. **Scale-Out** the PMs/VMs as per the requirement
15. **End If**
16. **If** $(Count_5 \geq \lceil \frac{m}{2} \rceil)$
17. **Scale Down** a single PM/VM at a time
18. **End If**
19. **Else**
20. **Call** Monitoring Mechanism
21. **End**

Fig. 10. Pseudocode of decision mechanism.

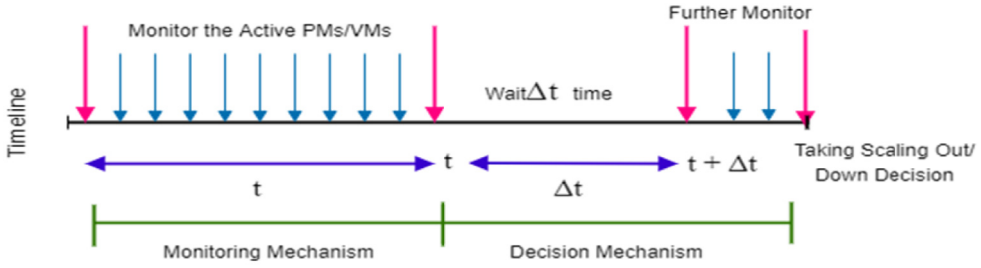

Fig. 11. Timeline of auto-scaling strategy.

proposed strategy over existing state-of-the-art algorithms are represented in Table 4.

The types of available containers of the data center with their resource usage and cost are depicted in Table 5. The scatteredness of different types of containers that are used for processing the microservices, in different datasets is shown in Fig. 13. The microservices are arranged in batches of 500, in sequential order. This shows how the selection of containers varies, as time progresses for the respective datasets.

To demonstrate the superiority of the proposed container-based auto-scaling strategy, we compared the proposed strategy with four popular Docker Swarm strategies, i.e. spread, round-robin, worst-fit and random, with respect to different aspects. The spread strategy deploys the new container to the PM which consists of minimum number of containers. The round-robin strategy deploys the selected containers in a circular manner to the available PMs. The worst-fit strategy selects the PMs for deploying the containers which have maximum number of running containers. The random strategy selects the PM randomly from the active PM pools for the containers. The existing Docker Swarm strategies select the PMs/VMs for the containers without considering their current resource usage and loads. Thus, the existing strategies minimize the resource utilization and maximize the processing time and cost of the microservices. However, the proposed container-based auto-scaling strategy deploys the microservices to the suitable containers based on their resource requirements and assigns the containers to the best-fit PMs/VMs based on their current loads. Moreover, the proposed strategy reuses the containers for further processing the remaining microservices and scale the PMs/VMs based on their resource availability. Thus, the proposed strategy utilizes computing resources efficiently while minimizing processing time and cost. The comparative results of the proposed strategy with the existing ones are discussed in the following sub-sections.

5.1. Resource utilization

Resource utilization of the containers and the active PMs/VMs for processing various types of microservices is one of the important parameters in the cloud environment. Here, we represent the resource utilization of multiple computing resources in terms of CPU and memory

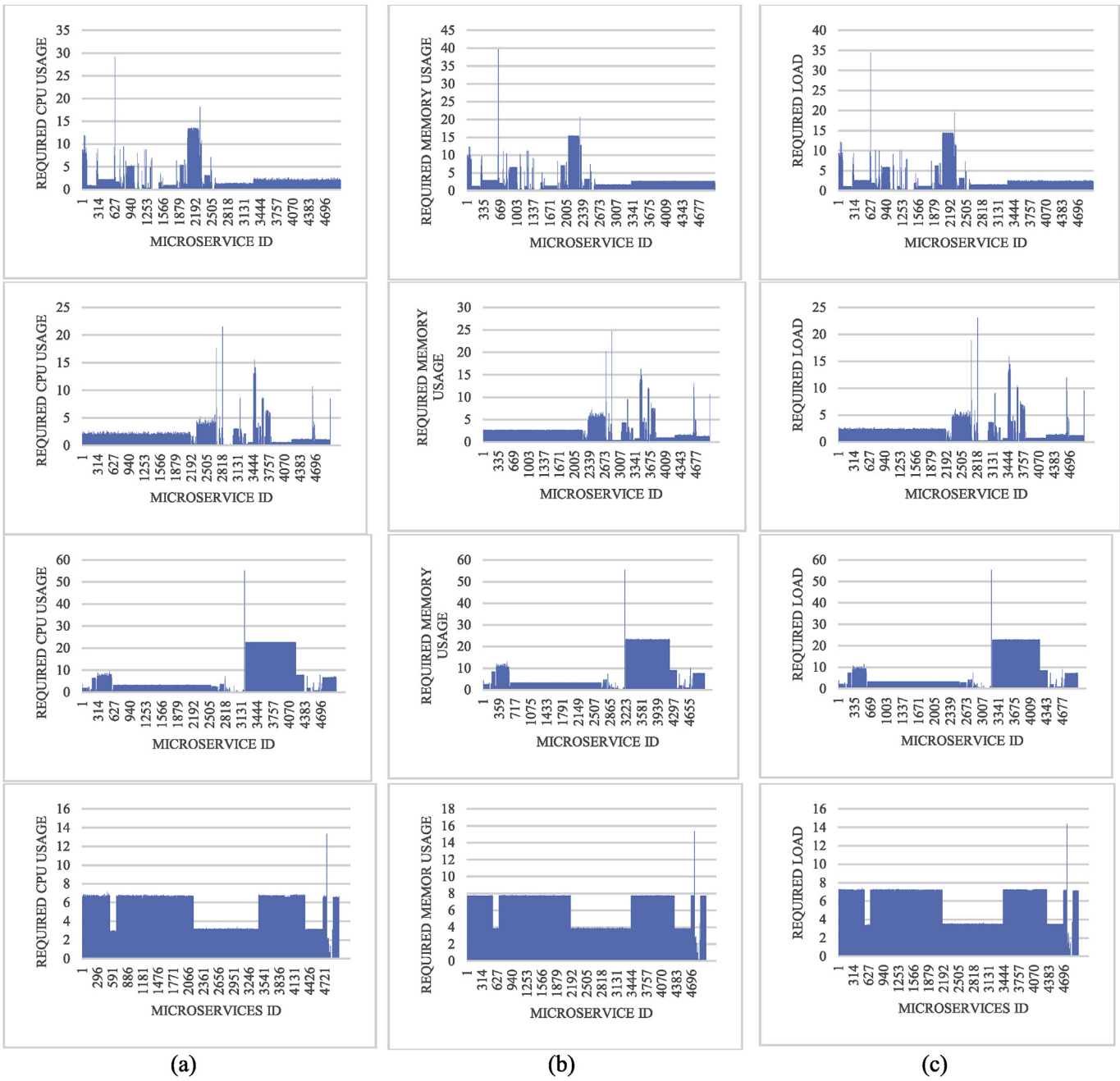

Fig. 12. Resource and load requested by microservices for various datasets of Google Trace log (a) Required CPU usage; (b) Required Memory Usage; (c) Required load for processing.

Table 4
Simulation parameters.

Simulation Parameters	Values
Number of real-time datasets	4
Requested number of microservices	5000/dataset
Total types of containers	10
Total number of PMs/VMs	20
Total number of CPU cores in PMs/VMs	$[40-100] \times 10000$ MIPS
Total number of CPU cores in containers	$[1-16] \times 10000$ MIPS
Total number of memory blocks in PMs/VMs	$[80-160] \times 2$ MB
Total number of CPU memory blocks in containers	$[1-16] \times 2$ MB
CPU usage cost	$2/CPU core
Memory usage cost	$0.5/memory block

Table 5
Types of containers.

Container Type	CPU Usage	Memory Usage	Load	Cost/Unit
C_1	1	1	1	2
C_2	1	2	1.5	3
C_3	2	2	2	4
C_4	2	4	3	6
C_5	2	8	5	10
C_6	4	4	4	8
C_7	4	8	6	12
C_8	4	16	10	20
C_9	8	8	8	16
C_{10}	16	16	16	32

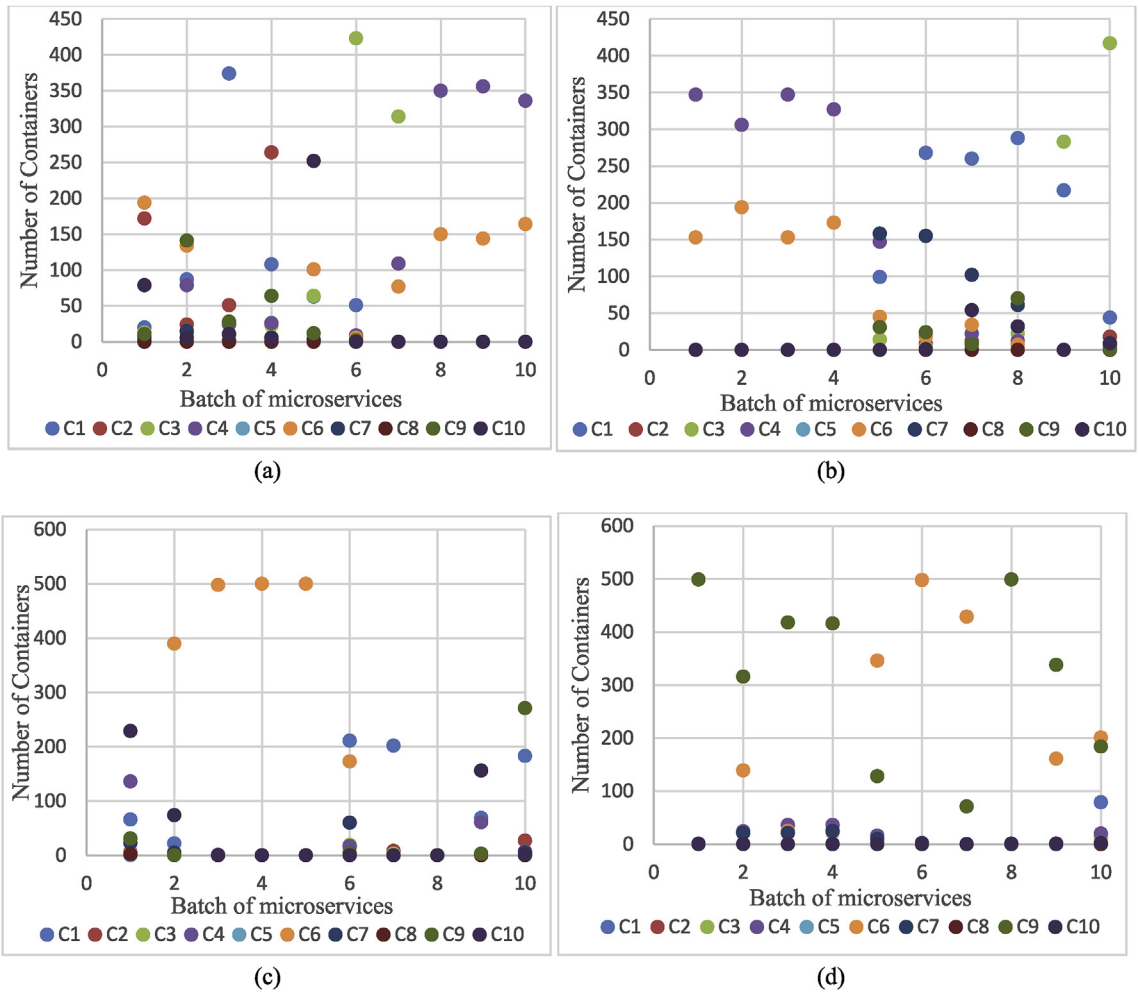

Fig. 13. Scattered off the containers for different microservices, arrived in every 500 s: (a) Dataset 1, (b) Dataset 2, (c) Dataset 3, (d) Dataset 4.

utilization. The proposed container-based auto-scaling strategy finds the best-fit containers based on the resource requirements and loads of the microservices which minimizes the resource wastage of the containers with efficient utilization of the resources. The CPU and memory utilization shares of the selected containers for processing microservices in different datasets are shown in Fig. 14. For an illustration, the Container types C_4 and C_6 are mostly used for processing microservices in Dataset 1, however, the Container types C_6 and C_9 are mostly used for processing microservices in Dataset 4.

The comparative results of the utilization of the computing resources of the proposed container-based auto-scaling strategy with the existing Docker Swarm strategies are shown in Fig. 15. The proposed strategy deploys the containers to the active PMs/VMs based on the current availability of the multiple computing resources and loads of those PMs/VMs using a dynamic bin packing strategy. This utilizes the computing resources efficiently as compared with the existing strategies. From Fig. 15, it is shown that in each time interval, the CPU utilization of the active set of PMs/VMs varies from 80 to 95 percent for all four datasets using the proposed strategy. Similarly, the memory utilization of the active set of PMs/VMs varies from 75 to 92 percent for all four datasets. This makes the data center more stable with efficient utilization of the computing resources. The existing strategies have deployed the microservices to the PMs/VMs randomly or based on the number of active containers on the PMs/VMs without considering the resource availability of loads. This reduces the utilization of the computing resources and the performance of the system. Moreover, the proposed strategy deploys an auto-scaling strategy for processing the microservices with minimum numbers of PMs/VMs which maximizes the resource utilization and the performance of the data center. The percentage of improvements of the proposed container-based auto-scaling strategy in terms of CPU utilization over the random, worst-fit, round-robin and spread are 15%, 12%, 10%, and 9% respectively. Similarly, the percentage of improvements of the proposed container-based auto-scaling strategy in terms of memory utilization over the random, worst-fit, round-robin and spread are 18%, 16%, 13%, and 10% respectively.

5.2. Processing time

The processing time of a microservice depends on the resource capacity of the selected container and the deployment time of the container in a suitable PM/VM. Thus, finding a suitable container for each microservice based on its resource requirements and reusing that container for remaining microservices can minimize the processing time. The existing Docker Swarm strategies find a container for each microservice based on their requirements and delete the container after processing. This increases the overall processing time of the microservices due to the extra deployment time of the containers. However, the proposed strategy selects the container for each microservices based on its resource requirements and reuses the containers (warm containers) for remaining microservices using Algorithm 2. This reduces the overhead of the container deployment time and minimizes the overall processing time. The processing time of the microservices of various Google Cluster Trace datasets on the selected containers is shown in Fig. 16. From Fig. 16, it is shown that the processing time of most of the microservices using the

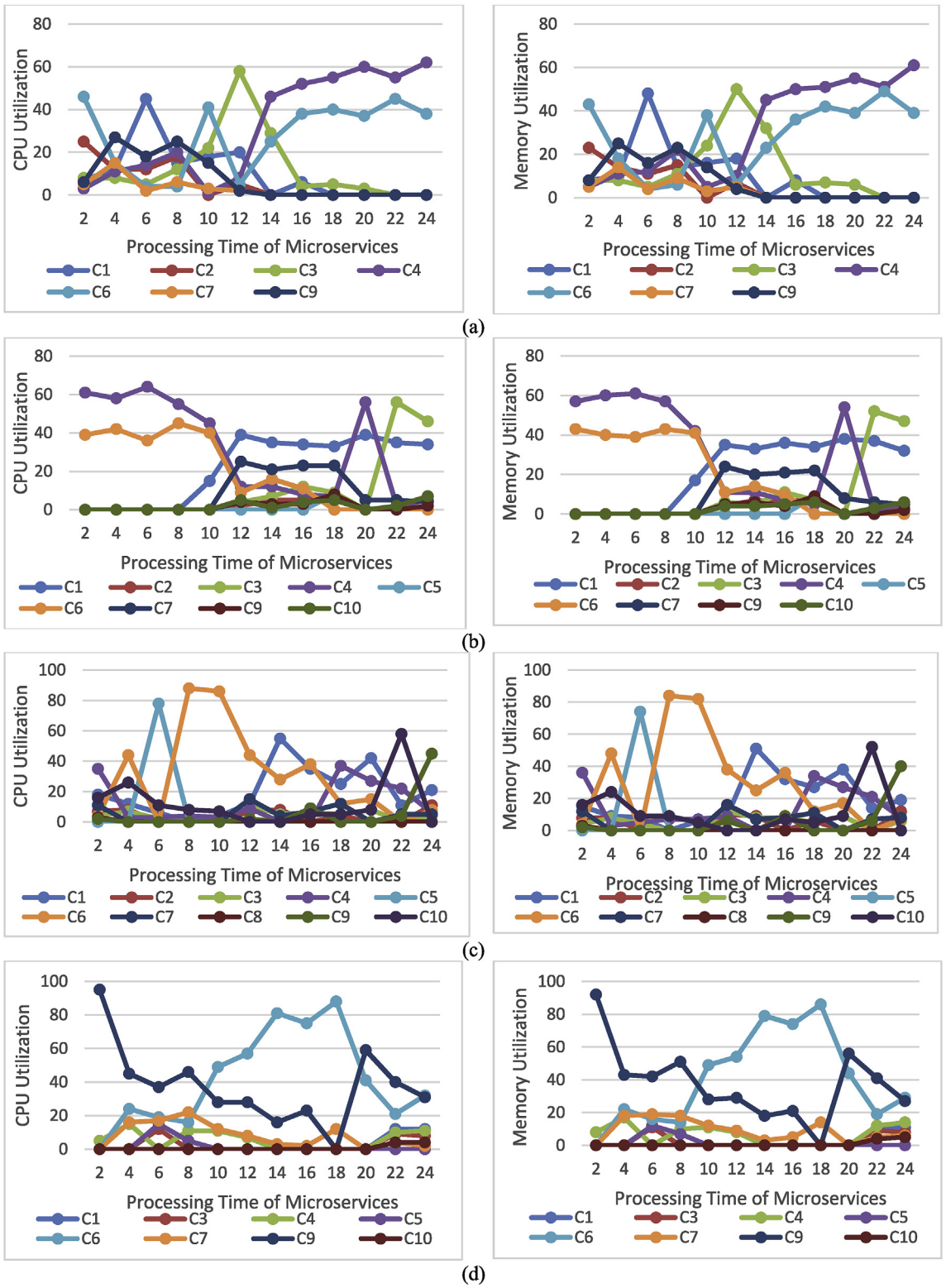

Fig. 14. Resource sharing (CPU and memory) of the containers: (a) Dataset 1, (b) Dataset 2, (c) Dataset 3, (d) Dataset 4.

proposed strategy varies from 0.6 to 1 s due to the minimum deployment time of the containers and reuse the warm containers on the active PMs/VMs.

The completion time of a set of microservices depends on their arrival time and their processing time on the selected containers. Moreover, the completion time of the microservices depends on the resource availability and allocation strategy on the active PMs/VMs. An optimal allocation strategy of the containers should increase the parallelism and minimize the overall completion time of the microservices. The proposed strategy finds suitable PMs/VMs for the containers using a dynamic bin packing strategy based on the current resource availability and loads them. Moreover, an auto-scaling strategy also designed for initiating a new set of PMs/VMs for processing remaining microservices when most of the PMs/VMs are overloaded. Thus, the proposed strategy increases

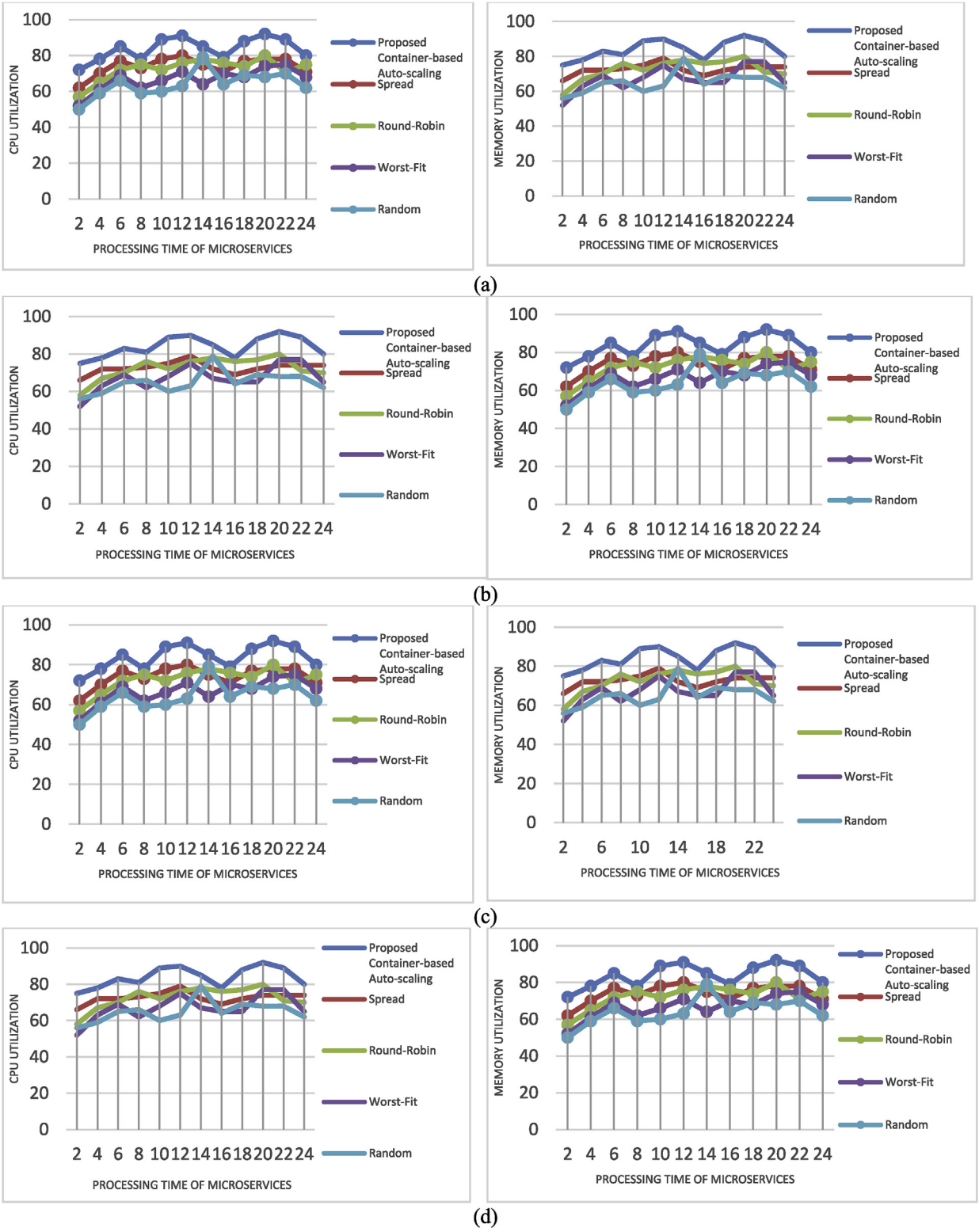

Fig. 15. Average resource utilization (CPU and memory) of the PMs/VMs: (a) Dataset 1, (b) Dataset 2, (c) Dataset 3, (d) Dataset 4.

the parallelism among the containers and minimizes the overall completion time. However, the existing strategies deploy the containers on the PMs/VMs without considering their current load, which increases the waiting time in the local queue of the microservices and minimizes the parallelism among the containers. The comparative analysis among the proposed strategy and existing Docker Swarm strategies over various datasets are depicted in Fig. 17. The percentage of improvements of the proposed container-based auto-scaling strategy in terms of the processing time over the random, worst-fit, round-robin and spread are 15%, 14%, 11%, and 9% respectively.

5.3. Processing cost

The processing cost of the microservices depends on the type of

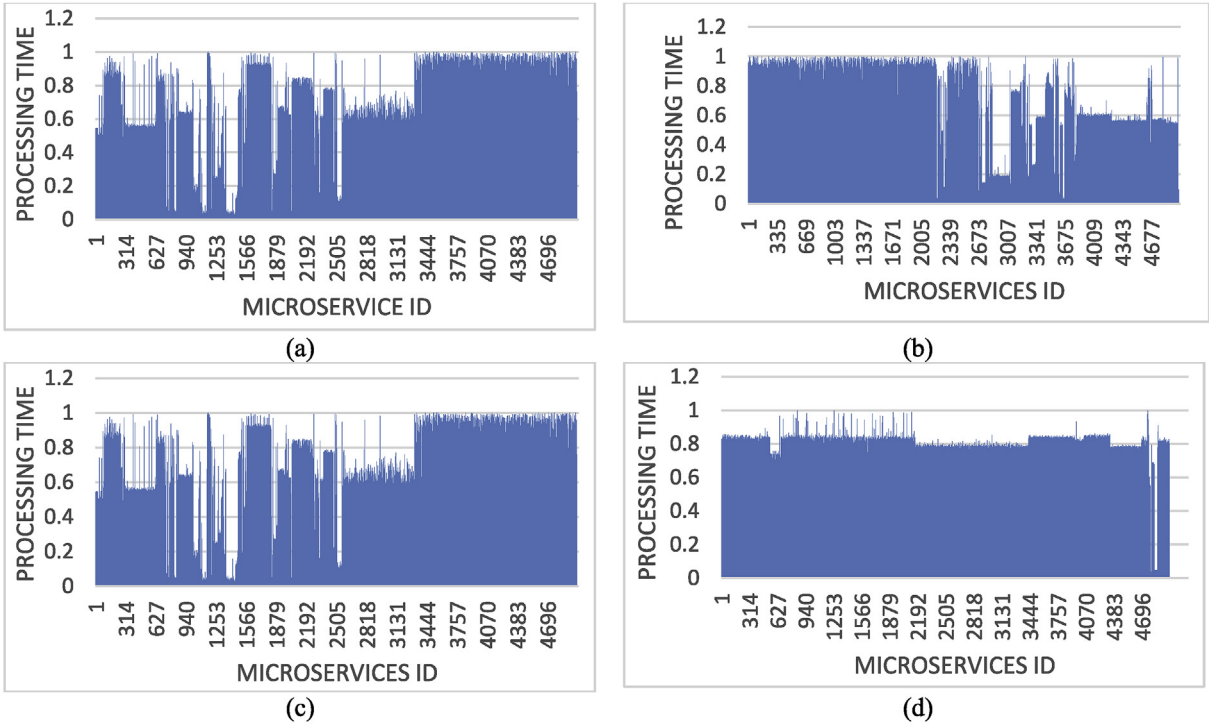

Fig. 16. Processing time of the microservices: (a) Dataset 1, (b) Dataset 2, (c) Dataset 3, (d) Dataset 4.

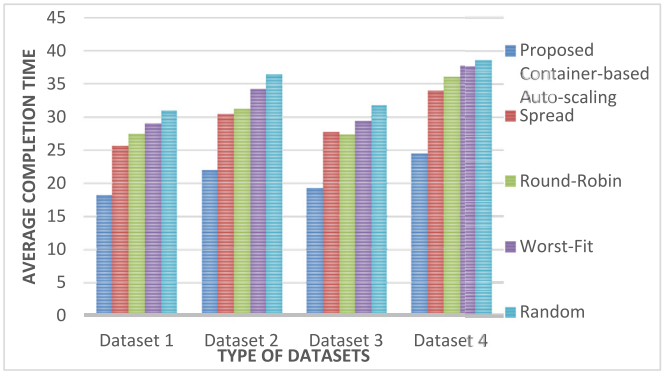

Fig. 17. Average completion time of the microservices on various datasets.

container being used for processing a microservice. The proposed strategy finds the best-fit container for each microservice based on their resource requirements using Algorithm 1. For example, a microservice MS_1 deploys on a suitable container from the container pool which is shown in Table 1. However, the proposed strategy deploys the microservice to container C_1 which meets the resource requirements of the microservices and contains minimum resources as compared with other available containers. This reduces the processing cost of the container. Moreover, the proposed strategy uses the dynamic bin packing strategy for deploying the containers on the PMs/VMs using Algorithm 3 and reuses the active containers using Algorithm 2 which minimizes the waiting time of the microservices and minimizes the processing cost. The processing cost of each microservice on the selected container over various datasets using the proposed strategy is shown in Fig. 18. From Fig. 18, it is shown that the processing cost of most of the microservices using the proposed strategy varies from $3 to $5 due to the minimum deployment time of the containers and reuse the warm containers on the active PMs/VMs. However, few of the microservices requested more CPU and memory capacity for processing, which increase the processing cost and the cost varies within the range of $10 to $27.

The overall processing cost of the microservices depends on the processing costs of the set of microservices arrive at a certain time instance. The proposed strategy deploys the microservices on suitable containers and minimizes the waiting time on the local queue of the PMs/VMs. This reduces the processing cost of each microservices which reflects the overall processing cost of the set of microservices. However, the existing Docker Swarm strategies deploy the microservices on the randomly selected containers and deploy the containers on the PMs/VMs without considering their current load, which increases the waiting time of the microservices and the processing cost. The comparative analysis of the proposed strategy and the existing Dock Swarm strategies over various datasets is shown in Fig. 19. The percentage of improvements of the proposed container-based auto-scaling strategy in terms of the processing cost over the random, worst-fit, round-robin and spread are 13%, 12%, 10%, and 8% respectively.

5.4. Number of PMs/VMs

The number of PMs/VMs depend on the number of microservices that have arrived in the global queue for processing. The cloud providers need to use an optimal allocation strategy for deploying the selected containers on the PMs/VMs and auto-scale the PMs/VMs horizontally based on the further resource requirements. Here, the proposed strategy uses the dynamic bin packing algorithm for deploying the microservices on the PMs/VMs based on their current load. Moreover, an auto-scaling strategy is designed for scaling the PMs/VMs horizontally using resource monitoring. This reduces the total number of PMs/VMs for processing the set of microservices with efficient resource utilization. Fig. 20(a) represents the variation of the number of PMs/VMs with respect to the processing time of the microservices. On the other hand, the existing static Docker Swarm strategies such as random, worst-fit, and round-robin deploy the microservices on the active set of PMs/VMs (initially it is considered as 20) without considering an auto-scaling policy. As a result, the static strategies use the same set of PMs/VMs without considering the resource wastage or overutilization of the resources. However, the spread algorithm selects a suitable PM/VM for each microservices dynamically but fails to find the optimal one. As a result, the existing Docker Swarm

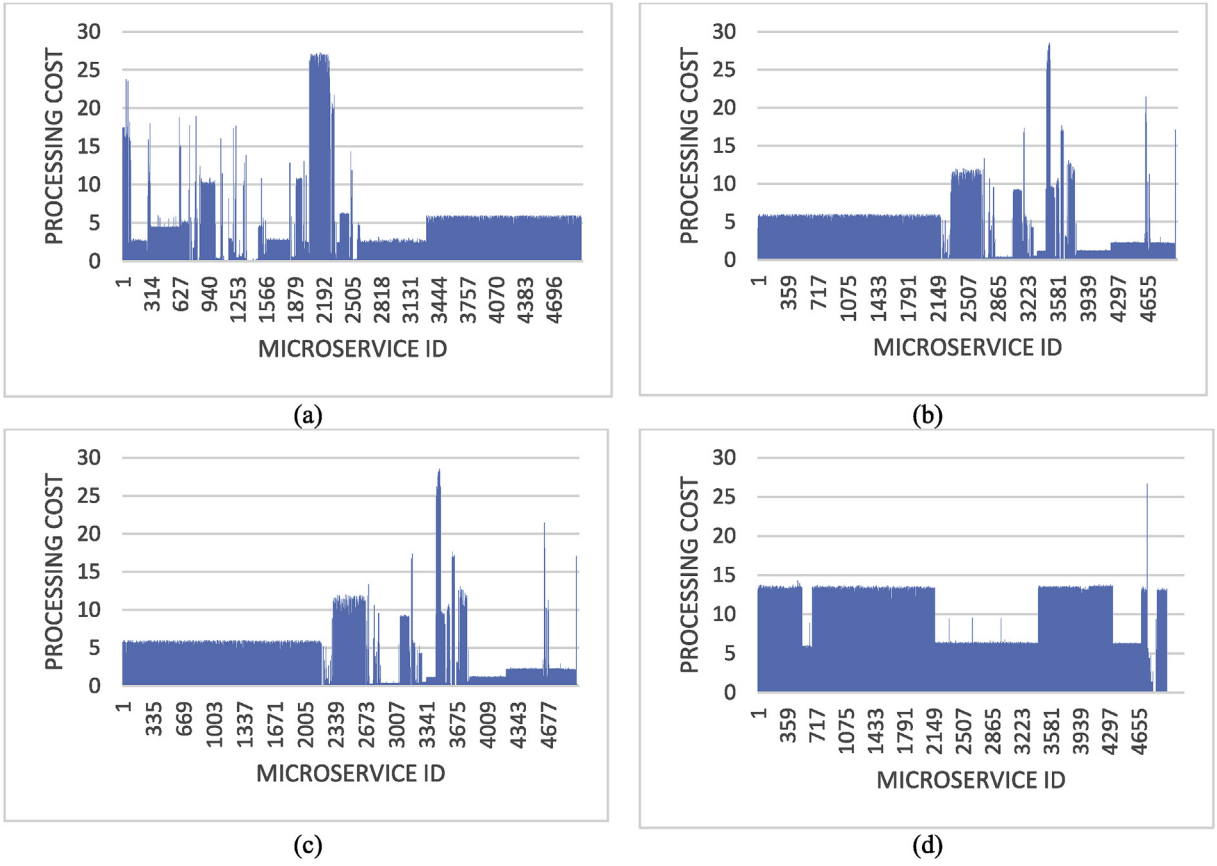

Fig. 18. Processing cost of the microservices: (a) Dataset 1, (b) Dataset 2, (c) Dataset 3, (d) Dataset 4.

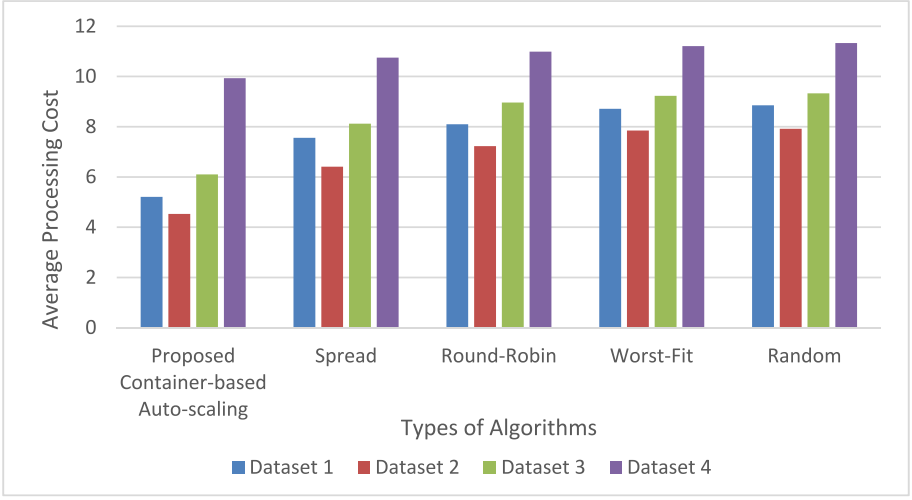

Fig. 19. Average processing cost of the microservices on various datasets.

strategies fail to vary the number of PMs/VMs efficiently as per the requirements. The comparative analysis of the existing Docker Swarm strategies with the proposed container-based auto-scaling strategy for variation of the active set of PMs/VMs in a data center while processing microservices is shown in Fig. 20(b). The percentage of improvements of the proposed container-based auto-scaling strategy in terms of the number of PMs/VMs over the random, worst-fit, round-robin and spread are 18%, 18%, 18%, and 10% respectively.

The above-mentioned comparative analysis shows that the proposed container-based auto-scaling strategy performs better than the existing Docker Swarm strategies for processing various types of microservices in a cloud environment.

6. Conclusion

In this paper, we have presented a container-based auto-scaling strategy in a cloud environment for processing the microservices efficiently. The contribution of the proposed method is four folded. The first contribution is to find a suitable container for each microservice based on its multiple resource requirements which minimize the processing time

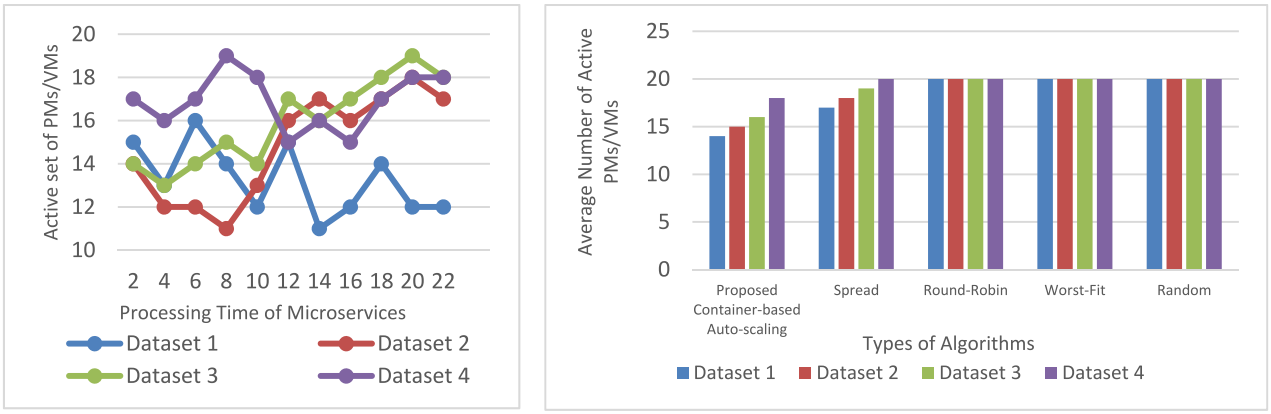

Fig. 20. Number of PMs/VMs (a) used by the proposed container-based auto-scaling strategy for different datasets; (b) the average number of active PMs/VMs used by different strategies.

and cost. The second contribution is to reuse the active warm containers for processing the remaining microservices. This minimizes the deployment times of the containers with minimum completion time and overall processing cost. The third contribution is to deploy the containers on the suitable PMs/VMs based on the availability of the resources using the best-fit bin packing strategy. This utilizes computing resources efficiently by minimizing the number of PMs/VMs. The fourth contribution is to auto-scale the resources based on the availability of the resources and the resources requested by the containers for processing remaining microservices. This minimizes the overall resource wastage of the PMs and balances the workloads among them. Finally, the proposed strategy is evaluated with existing Docker Swarm strategies using various performance matrices over different real-time Google Cluster traces. Overall, the container-based auto-scaling strategy minimizes 12–20% processing cost of the microservices and maximizes the CPU and memory utilization of the cloud servers by 9–15% and 10–18%, respectively, over the existing Docker Swarm strategies.

In our current experiments, we have used the Google Cluster Trace-logs for evaluation. Regarding our future work, we will try to have the experiments performed on the real infrastructure, also considering the warm containers. Moreover, we will try to apply the proposed strategy to real case studies as well. Currently, we are participating in the EU H2020 RADON project (Casale et al., 2019), where different microservices will be migrated to the cloud following the standardization approach using Topology and Orchestration Specification for Cloud Applications (TOSCA). Once the microservices are automatically deployed, they can also be scaled out or down using TOSCA in a cloud environment.

Declaration of competing interest

There are no potential conflicts of interest.

CRediT authorship contribution statement

Satish Narayana Srirama: Conceptualization, Methodology, Formal analysis, Resources, Writing - original draft, Writing - review & editing, Supervision, Project administration, Funding acquisition. **Mainak Adhikari:** Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Writing - original draft, Writing - review & editing, Visualization. **Souvik Paul:** Software, Validation, Investigation, Visualization.

Acknowledgment

This work has been partially supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 825040 (RADON).

References

Adhikari, Mainak, Nandy, Sudarshan, Amgoth, Tarachand, 2018. Meta heuristic-based task deployment mechanism for load balancing in IaaS cloud. J. Netw. Comput. Appl. 128, 64–77.

Adhikari, Mainak, Srirama, Satish Narayana, 2019. Multi-objective accelerated particle swarm optimization with a container-based scheduling for Internet-of-Things in cloud environment. J. Netw. Comput. Appl. 137, 35–61.

Barna, C., Litoiu, M., Fokaefs, M., Shtern, M., Wigglesworth, J., 2018. Runtime performance management for cloud applications with adaptive controllers. In: Proc. of Int'l. Conference on Performance Engineering. ACM, pp. 176–183.

Buyya, Rajkumar, Srirama, Satish Narayana, Casale, Giuliano, et al., 2019. A manifesto for future generation cloud computing: research directions for the next decade. ACM Comput. Surv. 51 (5), 105, 38 (ACM Press, New York, USA).

Casale, G., Artac, M., van den Heuvel, W.-J., van Hoorn, A., Jakovits, P., Leymann, F., Long, M., Papanikolaou, V., Presenza, D., Russo, A., Srirama, S.N., Tamburri, D.A., Wurster, M., Zhu, L., 2019. RADON: Rational Decomposition and Orchestration for Serverless Computing. Software-Intensive Cyber-Physical Systems (SICS), ISSN: 2524-8510. Springer (Accepted for publication).

Chen, T., Bahsoon, R., 2017. Self-adaptive trade-off decision making for autoscaling cloud-based services. IEEE Trans. Serv. Comput. 10 (4), 618–632.

DelValle, R., Rattihalli, G., Beltre, A., Govindaraju, M., Lewis, M.J., 2016. Exploring the design space for optimizations with Apache aurora and mesos. In: Proceedings of the 2016 IEEE 9th International Conference on Cloud Computing. CLOUD), *San Francisco*, CA, pp. 537–544.

Docker, 2018. https://www.docker.com.

Florio, L., Di Nitto, E., Gru, ‘‘, 2016. An approach to introduce decentralized autonomic behavior in microservices architectures. In: Proc. of Int'l. Conference on Autonomic Computing. IEEE, pp. 357–362.

Fowler, M., Lewis, J., 2014. Microservices a Definition of This New Architectural Term. http://martinfowler.com/articles/microservices.html.

Frey, S., Fittkau, F., Hasselbring, W., 2013. Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In: Proc. of Int'l. Conference on Software Engineering. IEEE, pp. 512–521.

Gandhi, A., Harchol-Balter, M., Raghunathan, R., Kozuch, M.A., 2012. AutoScale: dynamic, robust capacity management for multi-tier data centers. ACM Trans. Comput. Syst. 30 (4), 14.

Gandhi, A., Dube, P., Karve, A., Kochut, A., Zhang, L., 2014. Adaptive, model-driven autoscaling for cloud applications. In: Proc. of Int'l. Conference on Autonomic Computing, pp. 57–64.

Gotin, M., Lösch, F., Heinrich, R., Reussner, R., 2018. Investigating performance metrics for scaling microservices in cloudiot-environments. In: Proc. of Int'l. Conference on Performance Engineering. ACM, pp. 157–167.

Guerrero, C., Lera, I., Juiz, C., 2018. Genetic algorithm for multi-objective optimization of container allocation in cloud architecture. J. Grid Comput. 16 (1), 113–135.

Hightower, K., Burns, B., Beda, J., 2017. In: Kubernetes: up and Running: Dive into the Future of Infrastructure. O'Reilly Media, Inc.

Kaewkasi, C., Chuenmuneewong, K., 2017. Improvement of container scheduling for docker using ant colony Optimization. In: Proceedings of 2017 9th IEEE International Conference on Knowledge and Smart Technology (KST), pp. 254–259.

Li, J., Chinneck, J., Woodside, M., Litoiu, M., Iszlai, G., 2009. Performance model driven QoS guarantees and optimization in clouds. In: Proc. of ICSE Workshop on Software Engineering Challenges of Cloud Computing. IEEE Computer Society, pp. 15–22.

Li, W., Kanso, A., Gherbi, A., 2015. Leveraging Linux containers to achieve high availability for cloud services. In: Proceeding in IEEE International Conference on Cloud Engineering (IC2E), pp. 76–83.

Medel, V., Rana, O., Bañares, J.Á., Arronategui, U., 2016. Modelling performance & resource management in Kubernetes. In: Proceedings of the 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing. UCC), Shanghai, pp. 257–262.

Netflix. Retrieved from 1. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

Pérez, Alfonso, Moltó, Germán, Caballer, Miguel, Calatrava, Amanda, 2018. Serverless computing for container-based architectures. Future Generat. Comput. Syst. 83, 50–59.

Qu, C., Calheiros, R.N., Buyya, R., 2016. A reliable and cost-efficient autoscaling system for web applications using heterogeneous spot instances. J. Netw. Comput. Appl. 65, 167–180.

Qu, C., Calheiros, R.N., Buyya, R., 2018. Auto-scaling web applications in clouds: a taxonomy and survey. ACM Comput. Surv. 51 (4), 73.

Reiss, C., et al., 2011. Google Cluster-Usage Traces: Format + Schema. Google Inc., White Paper.

Tang, X., Zhang, F., Li, X., 2018. Quantifying cloud elasticity with container-based auto-scaling. Future Generat. Comput. Syst. https://doi.org/10.1016/j.future.2018.09.009.

Toffetti, G., Brunner, S., Blochlinger, M., Spillner, J., Bohnert, T.M., 2017. Self-managing cloud-native applications: design, implementation, and experience. Future Generat. Comput. Syst. 72, 165–179.

Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., Gil, S., 2015. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: Computing Colombian Conference (10CCC), 2015 10th. IEEE, pp. 583–590.

Wan, Xili, Guan, Xinjie, Wang, Tianjing, Bai, Guangwei, Choi, Baek-Yong, 2018. Application deployment using Microservice and Docker containers: framework and optimization. J. Netw. Comput. Appl. 119, 97–108.

Xu, X., Yu, H., Pei, X., 2014. A novel resource scheduling approach in container-based clouds". In: Proceedings of 2014 IEEE 17th International Conference on Computational Science and Engineering (CSE), pp. 257–264.

Yahia, E.B.H., Réveillère, L., Bromberg, Y.-D., Chevalier, R., Cadot, A., 2016. Medley: an event-driven lightweight platform for service composition. In: International Conference on Web Engineering. Springer, pp. 3–20.

Yin, L., Luo, J., Luo, H., 2018. Tasks scheduling and resource allocation in fog computing based on containers for smart manufacturing. IEEE Trans. Indust. Inform. 14 (10), 4712–4721.

Zhang H, Ma H, Fu, G, Yang, X, Jiang, Z, and Gao Y, "Container-based video surveillance cloud service with fine-grained resource provisioning", in Proceedings of 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), pp. 758-765.

Zhang, Fan, Tang, Xuxin, Li, Xiu, Khan, Samee U., Li, Zhijiang, 2019. Quantifying cloud elasticity with container-based autoscaling. Future Generat. Comput. Syst. 98, 672–681.

Satish Narayana Srirama is a Research Professor and the head of the Mobile & Cloud Lab at the Institute of Computer Science, University of Tartu, Estonia and a Visiting Professor at University of Hyderabad, India. He received his PhD in computer science from RWTH Aachen University, Germany. His current research focuses on cloud computing, mobile web services, mobile cloud, Internet of Things, fog computing, migrating scientific computing and enterprise applications to the cloud and large scale data analytics on the cloud. He is an IEEE senior member, is an Editor of Wiley Software: Practice and Experience, a 50 year old Journal, was an Associate Editor of IEEE Transactions in Cloud Computing and a program committee member of several international conferences and workshops. Dr. Srirama has co-authored over 140 refereed scientific publications in international conferences and journals. Dr. Srirama has successfully managed several national, international and enterprise collaborative research grants and projects. For further information of Prof. Srirama, please visit: http://kodu.ut.ee/ srirama/

Mainak Adhikari is currently working as a Post Doctorate Research Fellow at University of Tartu, Estonia. He has completed his Ph.D in Cloud Computing from IIT(ISM) Dhanbad, India in 2020. He has obtained his M.Tech.From Kalyani University in the year 2013. He earned his B.E.Degree from West Bengal University of Technology in the year of 2011. His area of research includes Internet of Things, Fog Computing, Cloud Computing, Serverless Computing and Evolutionary algorithm. He has Contributed numerous research Articles in various national and inter-national journal and Conference.

Souvik Paul currently joined as a Master's Program at University of Tartu, Estonia. He has completed his B. Tech Degree from Maulana Abul Kalam Azad University of Technology, India, in the year of 2019. His area of research includes Internet of Things, Fog Computing, Cloud Computing, and Machine Learning.