Scheduling of Containerized Resources for Microservices in Cloud

Kamalesh Karmakar[1,2](✉), Shramana Dey[2], Rajib K. Das[2], and Sunirmal Khatua[2]

[1] Future Institute of Technology, Kolkata, WB, India
k.karmakar.ju@gmail.com

[2] University of Calcutta, Kolkata, WB, India

Abstract. Most developers consider that microservice-based application design and development can improve scalability and maintainability. The microservices are developed as small independent modules and deployed in containers. The containers are deployed in virtual machines (VMs), which in turn run in hosts. Effective consolidation of the service requests to the containers may reduce the number of active hosts in a cloud environment, resulting in lesser power consumption of the cloud data centers. This research aims to maximize the resource utilization of the hosts by effectively allocating the containers to the VMs and VMs to the hosts. In this scheduling, a few additional containers and VMs are kept in the available resource pool so that during peak demand for services, the users get their service at the earliest (preferably without any delay). This paper presents a heuristic algorithm for microservice allocation in a containerized cloud environment to achieve these objectives. The performance of the proposed algorithm is validated and justified through the extensive experimental results. We have compared the performance of the proposed technique with the existing state-of-the-art. The number of container deployments in the proposed policy is reduced by 12.2–17.36% compared to the Spread policy and 6.13–10.57% compared to First-Fit and Best-Fit policies.

Keywords: Microservices · Cloud Computing · Container · Scheduling Technique · Resource Allocation

1 Introduction

With the fundamental shift in software development, service-oriented computing has become an attractive choice among software developers for its agility and efficiency. In a service-oriented architecture (SOA) [5,16], web services are small independent computing tasks (loosely coupled [16,20,23]) published as a service over the Internet. Web services make the business functionalities of one application accessible to other applications through web interfaces. These services can easily be integrated into client applications for low-cost development and deployment.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
S. Devismes et al. (Eds.): ICDCIT 2024, LNCS 14501, pp. 34–49, 2024.
https://doi.org/10.1007/978-3-031-50583-6_3

In a microservice architecture, an enterprise application is split into small self-contained modules, where a microservice is designated to perform a specific task. These microservices communicate among themselves using the *Simple Object Access Protocol (SOAP)*. Hence, instead of working on the shared codebase, the developers can be divided into many smaller groups who can work on developing business functions in different modules independently. It gives benefits for improving code re-usability and maintainability. More importantly, microservice-based applications should be scalable because of the highly fluctuating nature of demands.

The distributed deployment and fluctuating demand of the microservices require scalable resource provisioning for cost-effective deployment and execution. In this regard, cloud computing deals with the scenario by providing the backbone infrastructure as pay-per-use on-demand services. Furthermore, the Cloud Service Providers (CSPs) offer different pricing models for the end-users, depending on shared or dedicated resources. Hence, policy designing for resource allocation becomes a challenging task. Moreover, the cloud service providers want to reduce the energy consumption of the data centers and are facing challenges for dynamic resource provisioning of multiple clients within minimum infrastructural resources [11–14]. At the same time, service providers should ensure maintenance of the service level agreement of the users, like deadlines and quality of services [3,3,28]. This paper focuses on microservice deployment in Docker containers in Amazon EC2 on-demand instances to achieve these objectives.

Thus, this paper proposes a model for microservice-based task deployment and a heuristic algorithm for performance improvement. In this algorithm design, we try to minimize end-to-end delay for improved quality of services (QoS). Moreover, we ensure that the tasks are completed within their deadlines while keeping the monetary cost of task deployment low. Hence, the main contributions of this work are as follows:

- A microservice-based task deployment model.
- A heuristic algorithm to schedule the services to the containers.
- Extensive experimental results to analyze the performance of the proposed algorithm with existing techniques using benchmark data.

The rest of this paper is organized as follows. Section 2 illustrates a microservice architecture in a cloud environment. Later, we discuss the system model and formulate the problem in Sect. 3. Section 4 illustrates the proposed policy for task deployment. Section 5 presents and analyzes the experimental results. Finally, Sect. 6 concludes the paper with some future direction of research in this domain.

2 Literature Review

With the emergence of architectural patterns, microservice architecture has become necessary for achieving high flexibility in designing and implementing

modern applications. Hence, in the recent past, various microservice architectures have evolved and are standardized.

In [1], authors provide a benchmark analysis for master-slave and nested-container-based microservice architecture. They analyze the performance of the models in terms of CPU resource usage and network communication. In a keynote speech [8], Wilhelm Hasselbring discussed the importance of microservices over monolithic architecture. Several studies have shown that microservice architecture improves performance and DevOps development cycle over monolithic architecture [18,35].

However, SOA faces challenges using existing communication mechanisms with very high workload scenarios [2,9]. To overcome this challenge, a new technology, *Enterprise Service Bus* (ESB), was introduced [19,25]. It can achieve low latency and high scalability of the application. However, ESB is unsuitable for a cloud environment as virtual machine deployment is elastic, i.e., the number of virtual machines may vary based on service demand. Hence, to avoid this problem in SOA, a microservice architecture pattern has emerged that helps in developing lightweight services [4,21].

Furthermore, in recent days, microservice deployments in the cloud environment have gained attention to achieve the benefits of acquiring inherently scalable cloud resources whose billing is dependent on resource usage. High adaptation of cloud services is possible by deploying microservices in a VM instance, which can scale as per the user's demand. However, in recent days, containers have been preferred instead of VM instances as containers are lightweight, quickly migratable, and scalable. These containers, in turn, are deployed in VM instances. Microservices use container technology, which provides operating system-level virtualization [10,17,22]. One such container technology is Docker [6], developed as a lightweight virtualization platform [22,27].

In the SOA aspect, container-based application deployment has received considerable attention, and different companies have taken the initiative. *CoreOS* defined Application Container (*appc*)[1] specification by the image format, the runtime environment, and discovery protocol. Another notable runtime container engine project is *runC*[2], started by Docker. In 2015, Docker, Google, IBM, Microsoft, Red Hat, and many other partners formed *Open Container Initiative (OCI)*[3] for further enhancement and standardized container specification. OCI consists of two specifications, namely Runtime Specification (*runtime-spec*) and Image Specification (*image-spec*). In brief, an OCI image is downloaded and unpacked into an OCI runtime environment to create a container service. The Cloud Native Computing Foundation[4] designed an open-source software stack that allows the deployment of applications as microservices packed into containers. Furthermore, this software stack performs dynamic orchestration of the containers for optimizing resource utilization.

[1] https://coreos.com/rkt/docs/latest/app-container.html.
[2] https://blog.docker.com/2015/06/runc/.
[3] https://www.opencontainers.org/.
[4] https://www.cncf.io/.

In container-based microservices, each microservice, along with the required libraries and databases is packaged in a container deployed on a platform that supports container technology. Containerized microservices are isolated execution environments and can be instantiated at runtime based on service demand. These containers may run in one or more virtual machine(s) depending on the resource demand, virtual machine configuration, and availability of resources. Deployment of different containers on different virtual machines may improve performance when the resource demand is high. However, this incorporates more communication delay, as the virtual machines can be scattered within the data center or across data centers placed in different geographical areas.

2.1 Standard Data Formats, APIs and Protocols

Deployment of microservices in different containers, running in the same or different virtual machines, requires information exchange among the containers. Information exchange requires standard data representation and standard protocols. However, XML is a well-known and widely used data format for message exchange, JSON becomes more popular in recent days. *Data Format Description Language* (DFDL) supports a variety of data input, output formats and is used in both open source and proprietary tools [7,34]. Besides these, the Open API Initiative standardized RESTful API Markup Language (RAML), which can be used for a variety of data formats [30,31].

The microservice architecture uses standard HTTP and HTTPS protocols for information exchange. Along with standard TCP and UDP protocols, Stream Control Transmission Protocol (SCTP) is used for streaming services [24,26]. In the context of the machine to machine communication, the Organization for Advanced Structured Information Systems (OASIS)[5] standardized the Message Queuing Telemetry Transport (MQTT) protocol for publishing and subscribing messages at high speed [29,32].

3 System Model and Problem Formulation

Most software applications use microservices to deliver diverse functionalities to the end users. Hence, microservices are to be deployed in heterogeneous containers. Furthermore, the heterogeneity of underlying cloud data center resources and pricing models makes microservice allocation much harder. In this regard, the system is modeled based on the following assumptions:

- the containers can migrate to other VMs for consolidating them in a minimum number of VMs
- the VMs do not migrate as the time and monetary cost involved in VM migration is very high.

However, if the utilization of a container is too low and the environment is underutilized, newly arrived microservice requests are not allocated to it, which results in the termination of the container without any running service migration.

[5] https://www.oasis-open.org/.

3.1 System Model

Let us consider a set of microservice requests $S = \{s_1, s_2, s_3, \ldots, s_m\}$, where each request is associated with a deadline, are to be deployed in a containerized cloud environment. We consider a set of container images $CI = \{ci_1, ci_2, \ldots, ci_x\}$, where each container is configured for a specific type of microservice. The microservice containers $C = \{c_1, c_2, \ldots, c_n\}$ are instantiated using the container images. The containers are deployed into the VMs, $V = \{v_1, v_2, \ldots v_q\}$, which in turn are deployed into physical machines/hosts, $P = \{p_1, p_2, \ldots, p_r\}$.

The service requests are routed to the containers, where the corresponding microservice runs. An increase or decrease in the number of microservice requests causes the creation of new instances of containers or the termination of existing containers while keeping the resource utilization of containers within a desired range. The decision to scale the containers depends on the demand for container resources and the current utilization of the containers. Similarly, the VMs, that deploy the containers, are also scaled as per the current utilization of the running VMs. However, a container or VM is not shut down/stopped as soon as utilization becomes low. The scale-down occurs only if the containers or VMs are under-utilized for a certain period. The resource utilization of the containers and VMs is calculated as follows:

Resource Utilization of a Container. Let us represent service request allocation in a container using matrix $X_{i,j,t}$ where a request s_i is allocated to a container c_j at clock t. The utilization of the container c_j at time t is calculated as

$$u^c_{j,t} = \frac{\sum_{i=1}^{m}(mips_{s_i} \cdot X_{i,j,t})}{mips_{c_j}} \tag{1}$$

where $X_{i,j,t}$ is defined as

$$X_{i,j,t} = \begin{cases} 1, \text{if } s_i \text{ is allocated to } \textit{container } c_j \text{ at time } t \\ 0, \text{otherwise} \end{cases}$$

and $mips_{s_i}$ is the MIPS (million instructions per second) needed by the service request s_i, $mips_{c_j}$ is the capacity of the container in MIPS.

Resource Utilization of Virtual Machine. Similarly, we represent the container allocation to VMs using matrix $Y_{j,k,t}$, considering the container c_j is allocated to the VM v_k at time t. Thus, the utilization of a VM v_k at time t is measured as

$$u^v_{k,t} = \frac{\sum_{j=1}^{n}(mips_{c_j} \cdot u^c_{j,t} \cdot Y_{j,k,t})}{mips_{v_k}} \tag{2}$$

where $Y_{j,k,t}$ is defined as

$$Y_{j,k,t} = \begin{cases} 1, \text{if } c_j \text{ is allocated to } \textit{VM } v_k \text{ at time } t \\ 0, \text{otherwise} \end{cases}$$

and $mips_{v_k}$ is capacity of the VM v_k in terms of million instructions per second. The service requests allocated to a container are kept in the reservation table of the container.

Resource Utilization of Host. The VMs are deployed in hosts of a cloud data center. $Z_{k,l,t}$ represents the allocation of v_k in p_l at time t. That is,

$$Z_{k,l,t} = \begin{cases} 1, \text{ if } v_k \text{ is allocated to } \textit{host } p_l \text{ at time } t \\ 0, \text{ otherwise} \end{cases}$$

Hence, utilization of a host p_l at time t is measured as

$$u^p_{l,t} = \frac{\sum_{k=1}^{q}(mips_{v_k} \cdot u^v_{k,t} \cdot Z_{k,l,t})}{mips_{p_l}} \tag{3}$$

where q is number of VM instances, $mips_{v_k}$ and $mips_{p_l}$ are capacity of VM v_k and host p_l in terms of MIPS.

Energy Model of Host. The power consumption of a host is dependent on its utilization. This literature considers a linear power model shown below. The power consumption of a host p_l is calculated as:

$$En_{l,t} = \begin{cases} En^{idle}_{l,t} + (En^{busy}_{l,t} - En^{idle}_{l,t}) \cdot u^p_{l,t}, & \text{if } u^p_{l,t} \geq 0 \\ En^{off}_{l,t}, & \text{if host is shut down} \end{cases} \tag{4}$$

where $En^{idle}_{l,t}$ is the power consumption of an idle host and $En^{busy}_{l,t}$ is the power consumption of a fully utilized host.

3.2 Pricing Models of Different Platforms

Amazon AWS offers infrastructure services at different pricing schemes like *(i) reserved, (ii) on-demand, and (iii) spot instance* for the same instance type. The CSUs (cloud service users) can avail of on-demand instances at a fixed hourly rate with a guarantee of uninterrupted service. On the other hand, the price of spot instances varies with time, and availability depends on the bid price and overall demand for resources. The CSP can reclaim the spot instance whenever the price exceeds the bid. Nowadays, Amazon has changed its policy regarding the bid where the default bid is the on-demand price, and whenever the spot price is about to exceed the on-demand price, it issues a warning and revokes the spot instance after a short period (a few minutes). Thus, spot instances are generally cheaper than on-demand but come with a risk of revocation by CSPs (cloud service providers). We choose spot instances for microservice deployment as it is cost-effective and describe in Sect. 4 the techniques used to handle the problems arising out of revocation by CSPs.

3.3 Problem Formulation

Given a set of microservice requests $S = \{s_1, s_2, \ldots, s_m\}$, a set of container images $CI = \{ci_1, ci_2, \ldots, ci_x\}$, a set of virtual machines $V = \{v_1, v_2, \ldots v_q\}$ and a set of physical machines/hosts, $P = \{p_1, p_2, \ldots, p_r\}$, determine a schedule $f : s_i \rightarrow c_j$ where service request s_i is being allocated to container c_j, which are deployed in VMs $g : c_j \rightarrow v_k$, which in turn are deployed in physical hosts $h : v_k \rightarrow p_l$ in such a way that satisfies deadlines, minimizes the resource renting cost of the VMs, and achieves energy efficiency by minimizing the number of active hosts.

Algorithm 1: Service Allocation to Microservice Container

```
input  : S = {s_1, s_2, s_3, ..., s_m}; CI = {ci_1, ci_2, ci_3, ..., ci_x};
         C = {c_1, c_2, c_3, ..., c_n}; V = {v_1, v_2, v_3, ..., v_q}
output: Service request allocation to the Containers
1  keep un-allocated services of previous clock at the beginning of S.
2  foreach s_i ∈ S do
3      determine image type ci_g for service request s_i
4      determine containers C_a which can serve service request s_i
5      sort C_a in descending order of u^c_{j,t} · u^v_{k,t}, where u^c_{j,t} u^v_{k,t} are obtained
         from Eq. 1 and 2
6      set flag isServiceDeployed ← false
7      foreach c_j ∈ C_a do
8          status ← isDeployableContainer(s_i, c_j)
9          if status = true then
10             update reservation table of c_j; isServiceDeployed ← true
11             break;
12     if isServiceDeployed = false then
13         rearrange the VMs V in descending order of resource utilization
14         select a container configuration c for s_i based on ci_a;
15         set flag isContainerDeployed ← false
16         foreach v_k ∈ V do
17             if isDeployableVM(c, v_k) then
18                 instantiate container c in v_k based on container image ci_a
19                 update C ← C ∪ c; isContainerDeployed ← true
20         if isContainerDeployed = false then
21             instantiate a new VM v
22             instantiate container c in v based on container image ci_a
23             update V ← V ∪ v; C ← C ∪ c
24 call Algorithm 2 to manage pool of containers and VMs.
```

Algorithm 2: Resource Pooling of Containers and VMs

```
input  : service request, s_i; deadline of service request, dl_{s_i}; container, c_j;
         reservation table, rt_{c_j}
output: allocation status
1  rearrange the VMs V in descending order of resource utilization
2  for a = 1 to x do
3      select the containers C_a intantiated using container image ci_a
4      rearrange C_a in descending order of utilization
5      calculate utilization of microservice containers u^{ci}_{a,t} using Eq. 8
6      if overline(u^{ci}_{a.t}) > μ^{ch} then
7          add ⌈(overline(u^{ci}_a) − μ^{ch}) ∗ |C_a|⌉ number of containers, say C'_a, are to be
           deployed
8          foreach c_j ∈ C'_a do
9              set flag ← false
10             for v ∈ V do
11                 if u^v_{k,t} + mips_{c_j}/mips_{v_k} < μ^{vh} as per Eq. 9 then
12                     instantiate c_j in v_k; update flag ← true
13             if flag = false then
14                 instantiate a new VM v and add it to V
15     else if u^{ci}_{a,t} < μ^{cl} then
16         while u^{ci}_{a,t} < μ^{cl} do
17             remove the containers from C_a in which service is not allocated
               and the container is running in a most underutilized VM
```

4 Proposed Algorithm

The proposed algorithm deals with independent microservice requests for allocation in a set of containers C to run on a set of virtual machines V. A container, by design, can run only a specific type of microservice. Hence, the service requests S can be classified based on their attributes and mapped to the respective containers. Based on users' service demand, multiple containers instantiated from the same container image may provide the same microservice. On the other hand, we must instantiate at least one container for each type of microservice. These containers and VMs are to be scaled to reduce monetary costs. Thus, the proposed policy has two phases: *(i) Service request allocation, (ii) Containers and VMs Pool management*, as presented below.

4.1 Service Request Allocation

In a scheduling clock, the resource provisioner receives a set of service requests S and routes the requests to the containers. Before allocating any service request, the containers are to be rearranged in descending order of resource utilizations so that the resource provisioner allocates service requests to maximally utilized

but not overloaded containers. If we combine with this, the policy of freeing underutilized containers when demand for services reduces, we can decrease the number of container deployments. However, the monetary cost is not associated with the number of container deployments but with the number of VM deployments. Also, the resource utilization of the VMs impacts the energy consumption of the data center. Hence, to ensure the concentration of the containers in a lower number of VMs, the containers are sorted in descending order of the product of container and VM utilization ($u^c_{j,t} \cdot u^v_{k,t}$) as shown in line no. 4 of Algorithm 1. Here, $u^c j, t$ and $u^v_{k,t}$ are computed by Eqs. 1 and 2. Using line no 7–11, a container is selected for the service request s_i if $u^c_{j,t}$ is less than the upper threshold of container utilization μ^{ch} and if s_i is deployable in the container c_j, i.e., $mips_{s_i} + u_{c_j} \cdot mips_{c_j} \leq mips_{c_j}$. If the service request is not deployable in any existing container, a container must be instantiated using the container image ci_a on a suitable VM. A container is deployable in a VM if it has enough resources available. The resource availability of a VM v_k at time t is given by:

$$R^v_{k,t} = mips_{v_k} \cdot \mu^{vh} - \sum_j u^c_{j,t}.mips_{c_j}.X_{j,k,t} \tag{5}$$

where μ^{vh} is upper threshold of VM utilization.

To avoid resource contention among the containers, we assume that resources are fully reserved for a container irrespective of its utilization. Thus, the resource availability of v_k is redefined by substituting 1 for $u^c_{j,t}$ as

$$R^v_{k,t} = mips_{v_k} \cdot \mu^{vh} - \sum_j mips_{c_j}.X_{j,k,t}. \tag{6}$$

The function $isDeployableVM(c_j, v_k)$ returns true if the VM v_k can satisfy the resource requirement of the new container, i.e. $mips_{c_j} \leq R^v_{k,t}$.

Each container maintains a reservation table for the service requests. The reservation table of container c_j at time t (rt_{c_j}) has the tuples $<s_i, st_{s_i} l_{s_i}, at_{s_i}> | \quad \forall \quad X_{i,j,t} = 1$. Here, st_{s_i} is the start time, l_{s_i} is the service execution time, and at_{s_i} is the arrival time. In every scheduling clock, the reservation table is updated according to the arrival of new service requests and completion of existing service requests.

4.2 Resource Pooling of Containers and VMs

At the end of each scheduling clock, the algorithm investigates the health of container instances of each service type (container image type) and analyzes resource utilization to make scaling decisions.

The utilization of the containers $u^{ci}_{a,t}$ of container image type ci_a at time t is calculated as

$$u^{ci}_{a,t} = \frac{\sum_j u^c_{j,t}.mips_{c_j}}{\sum_j mips_{c_j}} \quad \forall c_j \in C_a \tag{7}$$

where $u^c_{j,t}$ is utilization of container c_j and C_a is the set of containers instantiated from container image ci_a. If the average utilization for a certain period t' to t is higher than the upper threshold μ^{ch}, we should increase the size of C_a (a scale-up). The average utilization for a certain period is calculated as

$$\overline{u^{ci}_{a,t}} = \frac{1}{t - t'} \int_{t'}^{t} u^{ci}_{a,t} \delta t \tag{8}$$

If the average utilization $\overline{u^{ci}_{a,t}}$ is greater than μ^{ch}, $\lceil(\overline{u^{ci}_{a,t}} - \mu^{ch}) * |C_a|\rceil$ number of containers are to be instantiated. While instantiating a container in an existing VM v_k, we must check that its new utilization does not exceed utilization threshold μ^{vh} as per the following equation:

$$u^v_{k,t} + \frac{mips_{c_j}}{mips_{v_k}} < \mu^{vh} \tag{9}$$

Other resource constraints such as memory etc., are also considered while deploying a container in v_k. If the container does not fit in any existing VM, we must instantiate a new VM to allocate the container. As the VM instantiation takes 1–2 min, to avoid any delay in launching the container, at the end of every scheduling clock, VMs are also scaled up to keep them in a reserved pool to allow immediate deployment of containers in them.

The containers are scaled down if the average utilization of the containers $\overline{u^{ci}_{a,t}}$ is less than the lower threshold μ^{cl}. The idle containers (maximum $\mu^{cl} - \overline{u^{ci}_{a,t}}.|C_a|$ of them are selected for shutting down). Similarly, the VMs are scaled down if resource average utilization $\overline{u^v_t}$ is less than the lower threshold μ^{vl}. The VMs without any allocated containers are selected for shutting down (maximum $\mu^{vl} - \overline{u^v_t}.|V|$ number of VMs).

The VMs are deployed in hosts using a polynomial-time heuristic technique, Resource Affinity-based VM Placement (RAbVMP), aiming for a reduction of active hosts (presented in [15]).

The rearrangement of containers and VMs according to the descending order of resource utilization ensures that during scale-up, allocation of microservices to the containers is such that utilization of the containers is highest at the front of the list and least at the tail. This approach ensures that the VMs at the tail of the list become idle. Similarly, the containers at the tail of the list, if idle, can be considered for shutting down. A priority queue based implementation for the lists of VMs and containers minimizes computational time.

5 Experimental Results and Discussion

In our experiment, we simulated the microservice deployment environment as presented in this article and obtained the results. In the simulation, different parameters act as input to the algorithms. The simulation environment assumes the continuous arrival of microservice requests from multiple users at runtime. Thus, the algorithm works as an online algorithm to manage dynamic arrivals of microservice requests.

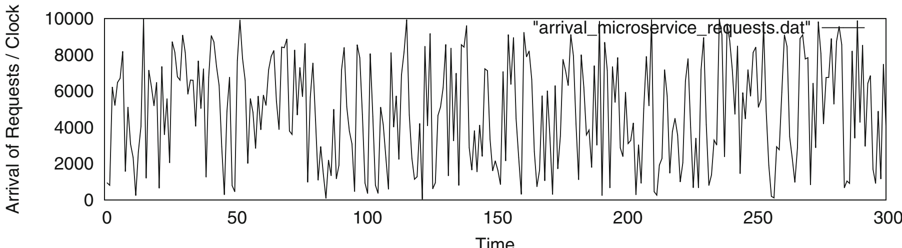


Fig. 1. Arrival of Microservice Requests in a Simulation, where the average arrival rate of microservices is 5000 per scheduling clock.

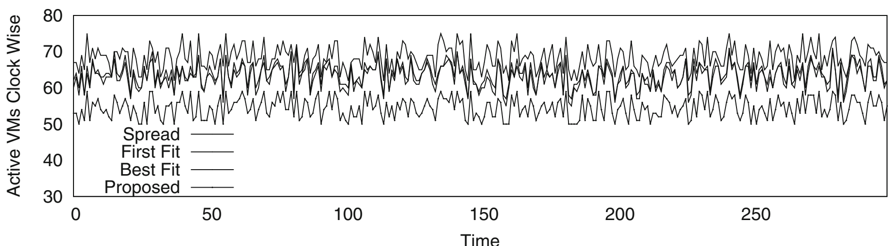


Fig. 2. Number of active VMs clock-wise, during a simulation where the arrival rate of microservice requests is 25000 in a minute.

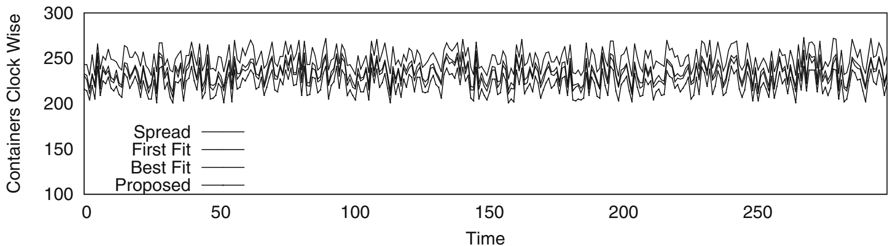


Fig. 3. Number of running Containers clockwise during a simulation where the arrival rate of microservice requests is 25000 in a minute.

5.1 Simulation Environment Setup

The simulation environment is designed based on the Fat-Tree data center network topology that uses 8-ary switches. The environment consists of 128 physical machines (hosts), where each host can accommodate 8 medium-sized VMs. The average number of container deployments in a VM is 4; this number may vary based on the resource availability of the VMs. Hence, the environment consists of approximately 1024 VMs in which nearly 4096 containers may run simultaneously.

In this simulation, the containers are deployed on c7g.xlarge (a compute-intensive VM type) with 4 vCPU and 8 GiB memory that supports up to 12.5

Gbps network bandwidth. As per the Microsoft Azure reference manual, one vCPU supports nearly 150 MIPS[6]. We consider that a container provides support upto 120 MIPS. Thus a VM may contain 4 containers at a time if the containers run at their maximum capacity because a VM is permitted to run at a maximum 85% utilization. If the containers are tuned to a lower MIPS, a VM can deploy more of them.

Furthermore, we consider that the average execution time of a microservice request in a container (running in maximum capacity) is 490 ms where microservice execution time varies within the range of 0.05 s to 2.0 s. Thus, every minute, an average of 122 microservices can be executed. Then, on average, a VM can complete $122 \times 4 = 488$ microservice requests per minute. In our simulation, we vary the number of microservice requests from 5000 to 25000 per minute.

For evaluating the performance of the proposed scheduling policy, we compared with the well-known common industrial strategies –Spread, BinPack, Random[7,8], and some state-of-the-art strategies– First-Fit and Best-Fit [33].

5.2 Result Analysis

In this simulation, the average arrival rate of microservice requests varies between 5000 and 25000 per minute. The arrival of the microservice requests is dynamic and follows the Poisson distribution. In this regard, Fig. 1 depicts the arrivals of microservice requests for a 300-min period where the arrival rate of requests is 5000. Figure 2 shows the number of active VMs using the proposed algorithm in a simulation where the arrival rate of the microservices is 25000. Figure 3 depicts the number of containers running in the VMs during a simulation period using the proposed algorithmic policy.

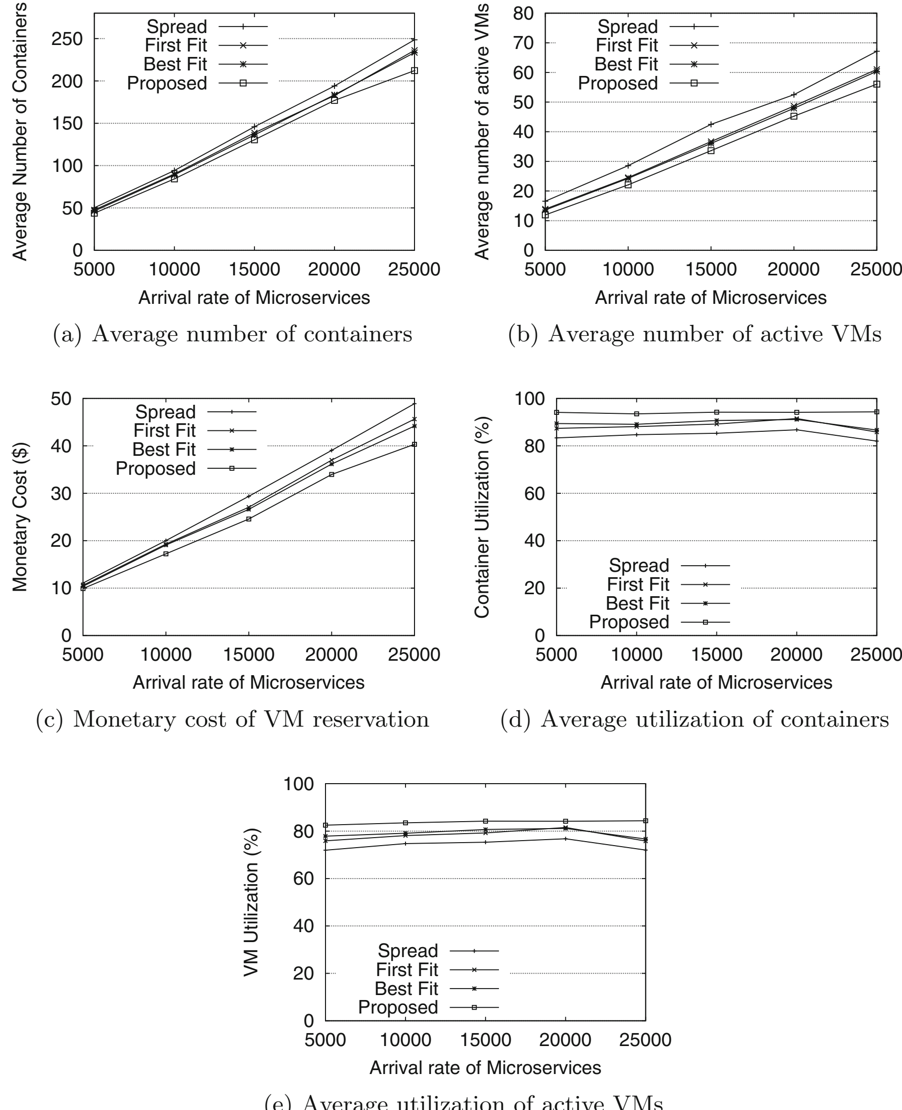
The results, shown in Fig. 4a, depict that the proposed policy significantly reduces the number of containers. The number of containers required for the execution of the microservices is a little high in the Spread policy, and the performances of the First-Fit/Best-Fit policies are close. The number of container deployments in the proposed method is lower than that for the Spread policy by 12.2–17.36%, and that for First-Fit/Best Fit policies by 6.13–10.57%.

Figure 4b shows the number of active VMs, and we observe that a decrease in the number of containers deployed results in fewer active VMs. The proposed policy reduces the number of active VMs by 11.23–15.42% compared to the Spread policy and 6.43–8.76% compared to First-Fit and Best-Fit policies. Reduction in the number of active VMs results in significant improvement in monetary cost, shown in Fig. 4c. This lower cost results from very high VM utilization caused by efficient consolidation of the containers. In this simulation, the upper threshold of VM utilization is 85% with a relaxation of 5%.

[6] https://docs.microsoft.com/en-us/azure/virtual-machines/workloads/mainframe-rehosting/concepts/mainframe-compute-azure.

[7] https://docs.docker.com/engine/swarm/.

[8] https://github.com/docker-archive/classicswarm/tree/master/scheduler/strategy.



(a) Average number of containers (b) Average number of active VMs

(c) Monetary cost of VM reservation (d) Average utilization of containers

(e) Average utilization of active VMs

Fig. 4. Results of microservice deployment in containers with varying arrival rate of microservice requests.

6 Conclusion and Future Scope

This paper presented a policy for allocating the microservice requests to the containerized cloud environment and auto-scaling the containers and underlying VMs. The objective is to reduce the monetary cost of task execution and minimize resource usage by maximizing resource utilization of the active VMs. More

specifically, the microservice requests are categorized based on their types and consolidated in a minimum number of containers that are deployed in a minimum number of VMs. We compared the performances of the proposed algorithm with Spread, First-Fit, and Best-Fit policies. The results of extensive experiments show that the proposed method significantly reduces the number of containers and active VMs.

Though the proposed policy improves the utilization of the resources in cloud infrastructure, there is a scope for further improvement in microservice deployment. Moreover, we plan to analyze workflow-based microservice deployment considering communication overhead among the containers. Furthermore, there is a need to develop a policy that allocates dependent microservices at proximity to improve performance.

References

1. Amaral, M., Polo, J., Carrera, D., Mohomed, I., Unuvar, M., Steinder, M.: Performance evaluation of microservices architectures using containers. In: 2015 IEEE 14th International Symposium on Network Computing and Applications, pp. 27–34. IEEE (2015)
2. Becker, A., Buxmann, P., Widjaja, T., et al.: Value potential and challenges of service-oriented architectures-a user and vendor perspective. In: ECIS, pp. 2085–2096 (2009)
3. Tarafdar, A., Karmakar, K., Khatua, S., Das, R.K.: Energy-efficient scheduling of deadline-sensitive and budget-constrained workflows in the cloud. In: Goswami, D., Hoang, T.A. (eds.) ICDCIT 2021. LNCS, vol. 12582, pp. 65–80. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-65621-8_4
4. Dragoni, N., et al.: Microservices: yesterday, today, and tomorrow. In: Mazzara, M., Meyer, B. (eds.) Present and Ulterior Software Engineering, pp. 195–216. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67425-4_12
5. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Pearson Education India (1900)
6. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and Linux containers. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 171–172. IEEE (2015)
7. Gao, T.T., Huang, F.W., Zhai, X.D., Zhu, P.: Generating data format description language schema. US Patent App. 14/724,851 (2015)
8. Hasselbring, W.: Microservices for scalability: keynote talk abstract. In: Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering, pp. 133–134 (2016)
9. Hutchinson, J., Kotonya, G., Walkerdine, J., Sawyer, P., Dobson, G., Onditi, V.: Evolving existing systems to service-oriented architectures: Perspective and challenges. In: IEEE International Conference on Web Services (ICWS 2007), pp. 896–903. IEEE (2007)
10. Kang, H., Le, M., Tao, S.: Container and microservice driven design for cloud infrastructure devops. In: 2016 IEEE International Conference on Cloud Engineering (IC2E), pp. 202–211. IEEE (2016)

11. Karmakar, K., Banerjee, S., Das, R.K., Khatua, S.: Utilization aware and network I/O intensive virtual machine placement policies for cloud data center. J. Netw. Comput. Appl. **205**, 103442 (2022)
12. Karmakar, K., Das, R.K., Khatua, S.: Minimizing communication cost for virtual machine placement in cloud data center. In: TENCON 2019–2019 IEEE Region 10 Conference (TENCON), pp. 1553–1558. IEEE (2019)
13. Karmakar, K., Das, R.K., Khatua, S.: Balanced graph partitioning based I/O intensive virtual cluster placement in cloud data center. In: 2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT), pp. 01–06. IEEE (2021)
14. Karmakar, K., Das, R.K., Khatua, S.: An ACO-based multi-objective optimization for cooperating VM placement in cloud data center. J. Supercomput. 1–29 (2022)
15. Karmakar, K., Khatua, S., Das, R.K.: Efficient virtual machine placement in cloud environment. In: 2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI), pp. 1004–1009. IEEE (2017)
16. Krafzig, D., Banke, K., Slama, D.: Enterprise SOA: Service-oriented Architecture Best Practices. Prentice Hall Professional (2005)
17. Kratzke, N.: About microservices, containers and their underestimated impact on network performance. arXiv preprint arXiv:1710.04049 (2017)
18. Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., Pallickara, S.: Serverless computing: an investigation of factors influencing microservice performance. In: 2018 IEEE International Conference on Cloud Engineering (IC2E), pp. 159–169. IEEE (2018)
19. Luo, M., Goldshlager, B., Zhang, L.J.: Designing and implementing enterprise service bus (ESB) and SOA solutions. In: 2005 IEEE International Conference on Services Computing (SCC 2005), vol. 1, 2, p. xiv-vol. IEEE (2005)
20. McGovern, J., Sims, O., Jain, A., Little, M.: Enterprise Service Oriented Architectures: Concepts, Challenges, Recommendations. Springer, Dordrecht (2006). https://doi.org/10.1007/1-4020-3705-8
21. Nadareishvili, I., Mitra, R., McLarty, M., Amundsen, M.: Microservice Architecture: Aligning Principles, Practices, and Culture. O'Reilly Media, Inc. (2016)
22. Pahl, C., Jamshidi, P.: Microservices: a systematic mapping study. In: CLOSER (1), pp. 137–146 (2016)
23. Papazoglou, M.P., Georgakopoulos, D.: Service-oriented computing. Commun. ACM **46**(10), 25–28 (2003)
24. Schlechtendahl, J., Kretschmer, F., Lechler, A., Verl, A.: Communication mechanisms for cloud based machine controls. Procedia CiRp **17**, 830–834 (2014)
25. Schmidt, M.T., Hutchison, B., Lambros, P., Phippen, R.: The enterprise service bus: making service-oriented architecture real. IBM Syst. J. **44**(4), 781–797 (2005)
26. Sill, A.: Standards at the edge of the cloud. IEEE Cloud Comput. **4**(2), 63–67 (2017)
27. Stubbs, J., Moreira, W., Dooley, R.: Distributed systems of microservices using docker and serfnode. In: 2015 7th International Workshop on Science Gateways, pp. 34–39. IEEE (2015)
28. Tarafdar, A., Karmakar, K., Das, R.K., Khatua, S.: Multi-criteria scheduling of scientific workflows in the workflow as a service platform. Comput. Electr. Eng. **105**, 108458 (2023)
29. Thangavel, D., Ma, X., Valera, A., Tan, H.X., Tan, C.K.Y.: Performance evaluation of MQTT and CoAP via a common middleware. In: 2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), pp. 1–6. IEEE (2014)

30. Verborgh, R., et al.: Survey of semantic description of REST APIs. In: Pautasso, C., Wilde, E., Alarcon, R. (eds.) REST: Advanced Research Topics and Practical Applications, pp. 69–89. Springer, New York (2014). https://doi.org/10.1007/978-1-4614-9299-3_5
31. Wilde, E., Pautasso, C.: REST: From Research to Practice. Springer, New York (2011). https://doi.org/10.1007/978-1-4419-8303-9
32. Xu, Y., Mahendran, V., Radhakrishnan, S.: Towards SDN-based fog computing: MQTT broker virtualization for effective and reliable delivery. In: 2016 8th International Conference on Communication Systems and Networks (COMSNETS), pp. 1–6. IEEE (2016)
33. Zhang, R., Zhong, A., Dong, B., Tian, F., Li, R.: Container-VM-PM architecture: a novel architecture for docker container placement. In: Luo, M., Zhang, L.-J. (eds.) CLOUD 2018. LNCS, vol. 10967, pp. 128–140. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94295-7_9
34. Zhao, Y., Dobson, J., Foster, I., Moreau, L., Wilde, M.: A notation and system for expressing and executing cleanly typed workflows on messy scientific data. ACM SIGMOD Rec. **34**(3), 37–43 (2005)
35. Zhou, X., et al.: Poster: benchmarking microservice systems for software engineering research. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), pp. 323–324. IEEE (2018)