Journal of Grid Computing (2024) 22:40
https://doi.org/10.1007/s10723-024-09745-7

RESEARCH

Cost-efficient Workflow as a Service using Containers

Kamalesh Karmakar · Anurina Tarafdar · Rajib K. Das · Sunirmal Khatua

Received: 3 October 2023 / Accepted: 12 January 2024 / Published online: 11 March 2024

Abstract Workflows are special applications used to solve complex scientific problems. The emerging Workflow as a Service (WaaS) model provides scientists with an effective way of deploying their workflow applications in Cloud environments. The WaaS model can execute multiple workflows in a multi-tenant Cloud environment. Scheduling the tasks of the workflows in the WaaS model has several challenges. The scheduling approach must properly utilize the underlying Cloud resources and satisfy the users' Quality of Service (QoS) requirements for all the workflows. In this work, we have proposed a heurisine-sensitive workflows in a containerized Cloud environment for the WaaS model. We formulated the problem of minimizing the MIPS (million instructions per second) requirement of tasks while satisfying the deadline of the workflows as a non-linear optimization problem and applied the Lagranges multiplier method to solve it. It allows us to configure/scale the containers' resources and reduce costs. We also ensure maximum utilization of VM's resources while allocating containers to VMs. Furthermore, we have proposed an approach to effectively scale containers and VMs to improve the schedulability of the workflows at runtime to deal with the dynamic arrival of the workflows. Extensive experiments and comparisons with other state-of-the-art works show that the proposed approach can significantly improve resource utilization, prevent deadline violation, and reduce the cost of renting Cloud resources for the WaaS model.

Keywords Workflow · Cloud computing · Container · Deadline · Resource utilization · Resource scheduling

Anurina Tarafdar, Rajib K. Das and Sunirmal Khatua contributed equally to this work.

K. Karmakar · R.K. Das · S. Khatua (✉)
Department of Computer Science and Engineering, University of Calcutta, JD-2, JD Block, Sector 3, Bidhannagar, Kolkata 700106, W.B., India
e-mail: skhatuacomp@caluniv.ac.in

K. Karmakar
e-mail: k.karmakar.ju@gmail.com

R.K. Das
e-mail: rajib.k.das@ieee.org

A. Tarafdar
Department of Computer Science and Engineering, Heritage Institute of Technology, Chowbaga Rd, Anandapur, Mundapara, Kolkata 700107, W.B., India
e-mail: anurinatarafdar@gmail.com

1 Introduction

A workflow consists of several tasks with data and control dependencies among them. Several fields like astronomy, biology, geophysics, and others extensively use Scientific workflows to solve complex scientific problems. These scientific workflows are for computing and data-intensive applications that need proper deployment to get results in a reasonable time. Cloud computing enables leasing any amount of resources on demand in a pay-per-use manner. The scalable and cost-effective nature of the Cloud makes it a preferred

option for deploying scientific workflows. The tasks of the workflows submitted by the users run in the virtual resources, considered as an infinite pool of resources, provided by the Cloud Service Providers (CSPs). The workflow deployment requests of the users are submitted to the Workflow Management System (WMS), which manages an infinite pool of virtual resources in CSP's infrastructure [1–3]. The tasks of the workflows are executed in Virtual Machines (VMs), which are deployed in CSP's physical infrastructure. Thus, the users have to pay only as long as they lease the resources.

A workflow consists of several types of tasks. Hence, a VM should be instantiated from the VM image containing the required software packages and dependencies to run the task. In traditional WMS, two approaches are used in VM image preparation: (i) one VM image containing the necessary software configurations for all tasks of the workflow, and (ii) multiple VM images for each type of task of the workflow. In the former case, the image size becomes too large, and executing multiple workflows for multiple users is almost unsolvable. Furthermore, it suffers from incompatibility issues with different software versions and configurations. A major challenge in the latter case is to prepare and register a large number of VM images for each distinct type of task of the workflows. Still, the latter technique is used in most of the WMS. However, keeping a set of dedicated VMs for each type of task of the workflows may lead to under-utilization of resources and increase resource renting costs [4].

Preparing a suitable schedule for an workflow involves deciding when to acquire/release the Cloud resources while satisfying the requirements at minimum cost. Most of the existing works in the literature consider the scheduling of a single workflow in the Cloud [5–7] where a single user submits a single workflow to the WMS. The WMS schedules the workflow in the Cloud to satisfy the user's QoS requirements. Frequent acquisition and release of the VMs results in non-negligible launching and termination time which is reflected as under-utilization of the resources. Intelligent allocation of multiple tasks one after another to same VM may enhance resource utilization, however, it causes security issue as tasks of different workflows of different users may run on same VM and if the VM is not properly cleaned after completion of each task. Thus, to avoid security issues, the pre-launched VMs should be terminated immediately after completion of the deployed task, even though resource utilization is compromised.

Containerized application deployment resolves the issue by creating isolated execution environments within the same VM and sharing the resources of the VM. Containers use virtualization at the operating system level. They allow the packaging of applications by providing a virtual environment that contains the necessary software, libraries, and other resources [8]. Each type of task in a workflow should have a corresponding container image, and its execution requires a container instance with the specified image. Due to the low overhead and minimum start-up delay, containers are becoming the preferred technology for task deployment in the Cloud. To prevent resource contention and workload interference, the "service instance per container" [9] deployment pattern has become popular. It allows only one task per container at a time, but a VM running on a host can have multiple containers. This model improves the resource utilization of the VMs and reduces the monetary cost of leasing resources.

The benefits of container-based virtualization motivate us to develop scheduling policies for workflow deployment on the Workflow as a Service (WaaS) platform [10,11]. Deployment of workflow applications on the WaaS platform reduces deployment effort as the platform is ready to deploy the workflows without any environment setup. The WaaS service providers take care of automated environment setup and scalability. The WaaS paradigm allows the execution of multiple workflows by different users and various structures with different QoS requirements in an isolated execution environment. It significantly reduces launch and release times. Furthermore, the size of a container image is significantly smaller than a VM image, which helps in quick replication and migration.

Scheduling workflows on a WaaS platform is more challenging than on a traditional WMS. The scheduling strategies should effectively utilize the Cloud resources and ensure the satisfaction of QoS requirements (like deadline satisfaction, reduction in makespan, and others). Scientific workflows generally differ in structure, number, and type of tasks [12]. Hence, configuration selection and management of the containers for proper scheduling of the workflows become quite challenging. Apart from the heterogeneous resource requirements of the workflows, they also have an unpredictable arrival rate. Thus, it is difficult to devise strategies for provisioning and auto-scaling resources for the tasks.

Containerized resources allow more flexibility in allocating resources to tasks, improve utilization, and reduce cost. Unlike VMs (which can only be scaled horizontally), containers allow vertical and horizontal scaling. Since a container can execute only one task at a time, we need horizontal scaling of containers depending on the demand for concurrent tasks. If the number of parallel tasks increases, more containers are needed to serve the users, and hence, more instances of the containers should be available in the system (containers get instantiated from different container images based on the type of task). On the other hand, if the demand decreases, we scale down the number of containers. For a task allocated to a container, we decide the MIPS and resource for the container depending on how quickly we need to complete the task, and accordingly, we vertically scale the container (increase or decrease its MIPS and memory resources).

In this paper, we have proposed a heuristic approach to schedule multiple deadline-constrained workflows in a containerized Cloud environment that addresses all the scheduling challenges discussed above. Our proposed heuristic maximizes the resource usage of the active VMs and thus reduces the resource rental cost. Our scheduling and auto-scaling approaches consider most factors affecting workflow scheduling, such as data transfer time, container image caching time, and provisioning delay of containers and VMs. The main contributions of this work are as follows:

1. We have proposed an algorithm that schedules the tasks of a workflow in containers deployed in on-demand VM instances. The computing resources of the containers are dynamically tuned to consolidate a maximum number of containers in a VM. We have formulated the problem of optimizing computational resources while completing a workflow within its deadline to a non-linear optimization and applied the Lagranges method to solve it. This solution helps to scale the resources of the containers dynamically. Also, to reduce the network delay, the schedule tries to allocate tasks on topologically close VMs if they are dependent (one is a predecessor or a successor of the other).
2. We have proposed an approach to effectively scale containers and VMs while accommodating newly arrived workflows at runtime.
3. We have conducted extensive experiments using benchmark data to validate the efficacy of our approach in comparison to other state-of-the-art techniques. Simulation results show that our method significantly improves resource utilization, reduces the resource renting cost of Cloud resources, and prevents deadline violation of the workflows.

The rest of this paper is organized as follows. Section 2 consists of the related works, followed by the problem definition and system model in Section 3. Section 4 illustrates our proposed workflow scheduling and auto-scaling approach, while experiments and analysis of the results appear in Section 5. Finally, Section 6 concludes the paper with a few future research directions.

2 Review of Literature

Workflow scheduling in the Cloud environment has drawn significant attention from researchers in the last few years. A few strategies have evolved to schedule workflows keeping different objectives under consideration like minimization of makespan, monetary cost, energy consumption, etc. In [13–15], the authors have proposed algorithms to minimize the makespan and monetary cost of the workflows. Qin et al. [6] have used reinforcement learning to reduce the energy consumption and makespan of the budget-constrained scientific workflows. Stavrinides et al. [5] have presented an energy-efficient, cost-effective, and QoS-aware scheduling strategy for scheduling workflows that use the DVFS technique to reduce energy consumption. In [16], authors presented a novel workflow scheduling approach for reducing the average makespan of the workflows and resource renting cost, and improving energy efficiency of the Cloud resources. However, these scheduling approaches are for the traditional WMS intended to schedule a single workflow.

Researchers have also investigated the use of containers to schedule scientific workflows. For example, Qasha et al. [17] and Liu et al. [18] have shown the docker containers to be highly suitable for deploying scientific workflows for their low overhead and high flexibility. Rodriguez et al. [8] have considered the emerging WaaS platform and have proposed an algorithm for scheduling multiple workflows using container-based virtualization. Similarly, in [4], the authors have considered a containerized Cloud environment and have proposed an approach to schedule

workflows to minimize their makespan while satisfying the budget constraint. Unlike our work, in [4,8], the authors have deployed only a single containerized task to a VM. It does not allow scaling of the container's resources depending on the task's requirements and leads to under-utilization of resources.

Chen et al. [19] have proposed an uncertainty-aware online scheduling algorithm (ROSA) to schedule multiple workflows having deadlines. ROSA controls the number of tasks waiting on each VM to prevent the propagation of uncertainties. However, their approach deploys tasks directly on VMs without using container-based virtualization. Moreover, they have not considered maximizing resource utilization as a scheduling objective. Burkat et al. [20] have conducted experiments to evaluate the efficacy of containers in deploying scientific workflows. Their experiments with container services like AWS Fargate and Google Cloud Run clearly show the viability of containers for executing scientific workflows. Ranjan et al. [21] have designed an energy-efficient container-based virtualized model for scheduling workflows. But [19,21] do not include strategies for auto-scaling of the Cloud resources.

Even though the above discussion underlines the presence of several workflow scheduling approaches, there is a lack of investigations on the problem of scheduling multiple workflows in a WaaS platform. We have proposed a scheduling and auto-scaling approach for multiple deadline-sensitive workflows in a containerized Cloud environment. To the best of our knowledge, this is the first approach to optimally scale the computing resource of every container while satisfying the deadline constraints. It improves the resource utilization of the active VMs, enables execution of the workflow tasks by leasing minimal resources, and consequently reduces the cost of renting resources.

3 Problem Definition and System Model

Our approach to processing multiple deadline-constrained scientific workflows in the containerized Cloud involves resource provisioning, scheduling, and auto-scaling. The tasks of the workflows run in containers that are isolated lightweight computing environments deployed in VMs. These tasks are of different types (as per the requirements [12]), each requiring a container instantiated from a specific container image. A container image contains the codebase for a task, and the users prepare and upload the container images to the service provider's infrastructure. The WaaS provider manages these container images characterized by their codebases and the runtime environments. Table 1 lists the symbols used in this manuscript for the problem formulation and illustration of the system model..

We consider a set of workflows $WF = \{ wf_1, wf_2, wf_3, \ldots, wf_z\}$ which are associated with deadlines $DL = \{dl^{wf_1}, dl^{wf_2}, dl^{wf_3}, \ldots, dl^{wf_z}\}$. Let $T_i, 1 \leq i \leq z$, denote the set of tasks in workflow wf_i. As these workflows come to the WaaS platform from different users, they may differ in structure, size, and input-output data. A directed acyclic graph, $wf_h(T_h, E_h)$ represents the workflow wf_h, where $T_h = \{\tau_1, \tau_2, \ldots, \tau_e\}$ is the set of tasks and E_h is the set of edges representing dependencies among the tasks. Dependency of a task τ_a on a task τ_i (represented by $\tau_i \rightarrow \tau_a$) indicates that τ_a can start only after receiving the output generated by τ_i. If τ_a is dependent on τ_i, the task τ_i is called the immediate predecessor of τ_a and τ_a is the immediate successor of τ_i. The immediate predecessors and successors of τ_i are represented by $pre(\tau_i)$ and $suc(\tau_i)$. The weight of the directed edge, $w_{i,a}$, denotes the volume of data to be transferred from τ_i to τ_a.

In a workflow, wf_h, the tasks, which are not dependent on any other tasks, are considered initial tasks represented by a set T_h^{in}. Thus, $T_h^{in} \leftarrow \{\tau_i \in T_h \mid pre(\tau_i) = \phi\}$. Similarly, the set of the tasks of the workflow, wf_h, on which none of the other tasks are dependent, is considered as the set of final tasks represented by a set T_h^{fin} and is defined as $T_h^{fin} \leftarrow \{\tau_i \in T_h \mid suc(\tau_i) = \phi\}$.

Let T denote the set of all the tasks of the workflows. Hence, $T = \{T_1 \cup T_2 \cup \ldots T_z\} = \{\tau_1, \tau_2, \tau_3, \ldots, \tau_m\}$ are to be deployed in the containerized Cloud environment. The set $CI = \{ci_1, ci_2, \ldots, ci_o\}$ is the set of container images and the set $C = \{c_1, c_2, \ldots, c_n\}$ represents the set of containers. Each container in the set C is instantiated with a particular container image from the set CI. Each container may require different resources based on the task it executes. The containers run on the VMs, $V = \{v_1, v_2, \ldots v_q\}$ deployed on the physical machines (hosts) of the data center.

Thus, the problem statement considered in this manuscript can be formally defined as follows:

Definition 1 *Given a set of deadline-sensitive workflows* $WF = \{wf_1, wf_2, \ldots, wf_z\}$, *allocate the tasks*

Table 1 List of symbols

Symbol	Description		
WF	Set of workflows	wf_h	The h^{th} workflow in the set WF
dl^{wf_h}	The deadline of the workflow wf_h	T_h	The set of tasks in the workflow wf_h
T	The set of tasks considering all the workflows in the set WF	T_h^{in}	The set of initial tasks of wf_h
T_h^{fin}	The set of final tasks of wf_h	τ_i	The i^{th} task in a workflow
$pre(\tau_i)$	The set of immediate predecessors of τ_i	$suc(\tau_i)$	The set of immediate successors of τ_i
CI	The set of container images	ci_g	The g^{th} container image
$ci_g.size$	The size of the container image ci_g	C	The set of containers
C_g	The set of containers deployed with the image ci_g	c_j	The j^{th} container
V	The set of virtual machines	v_k	The k^{th} virtual machine
R_{v_k}	The resources of the VM v_k	M_{v_k}	The per unit time monetary cost of the VM v_k
$M(V)$	The total monetary cost of execution of the workflow tasks	$st(v_k)$	Start time of VM v_k
$tt(v_k)$	Termination time of VM v_k	$f(\tau_i)$	The container c_j in which task τ_i is allocated
$g(c_j)$	The VM v_k in which container c_j is allocated	bw_{v_k}	Bandwidth of VM v_k
mi_{τ_i}	Length of task τ_i in million instructions	$mips_{\tau_i}$	Tuned MIPS requirement of task τ_i
$mips_{c_j}$	Tuned MIPS of container c_j	$mips_c^{\max}$	Maximum CPU capacity of a container in MIPS
$mips_{v_k}$	CPU capacity of VM v_k in million instructions per second		
$X_{i,j,t}$	Allocation of task τ_i to container c_j at the time t	$Y_{j,k,t}$	Allocation of container c_j to VM v_k at the time t
$ut_{v_k,t}$	The CPU utilization of VM v_k at time t		
$\Delta t_{a,i}$	Data transfer time from task τ_a to task τ_i	dv_{jb}	Volume of data to be transferred from c_j to c_b
dt_{c_j,c_b}^{wt}	Output data transfer time from container c_j to container c_b	dt_{c_b,c_j}^{rd}	Input data receive time from c_b to c_j
γ_{c_j,c_b}	comm. delay, for the distance between containers c_j and c_b	$et(\tau_i, c_j)$	Execution time of task τ_i in container c_j
$pt(\tau_i, c_j)$	Processing time of a task τ_i in a container c_j	$ic(ci_g, v_k)$	Time to cache container image ci_g to VM v_k
$est(\tau_i)$	Earliest start time of task τ_i	$eft(\tau_i)$	Earliest finish time of task τ_i
$lst(\tau_i)$	Latest start time of task τ_i	$lft(\tau_i)$	Latest finish time of task τ_i
ast_{τ_i}	Actual start time of τ_i	aet_{τ_i}	Actual end time of τ_i in c_j
ζ_{τ_i,c_j}	Container affinity of task τ_i for c_j		

into containers $f : \tau_i \rightarrow c_j$ and determine a container deployment schedule $g : c_j \rightarrow v_k$ subject to the constraints of resource availability, that maximizes the resource usage of the active VMs, reduces the resource renting cost of the Cloud, and satisfies the deadline of each of the workflows.

In the Cloud, each VM has a resource configuration and an associated monetary cost. The resources and the associated cost of a VM, v_k, is represented by $\{R_{v_k}, M_{v_k}\}$ where $R_{v_k} = (c, f, r, d)$ denotes the different resources associated with the VM v_k, and M_{v_k} represents the monetary cost of VM v_k per unit time [22]. Here c, f, r, and d stand for the number of CPU cores, frequency of each core, size of main memory, and volume of disk space, respectively. Generally, in the AWS Cloud[1], the VM instances are broadly classified as compute/memory/storage optimized or general purpose. As scientific workflows mostly involve high-performance computing, they are best suited for compute-optimized VM instances, which are leased on-demand in a pay-per-use manner. The tasks of the workflows are allocated to the containers running in these VMs.

To prepare the schedule of containers to the VMs and VMs to the hosts; the allocation of a task, τ_i to a container c_j at time t is represented by $X_{i,j,t}$ as

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

To prevent contention for resources, we restrict a container to execute only one task at a given time. The allocation of a container c_j to a VM v_k at time t is represented by $Y_{j,k,t}$ as

$$Y_{j,k,t} = \begin{cases} 1 & \text{if } c_j \text{ is allocated to } v_k \text{ at time } t \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

The container allocation to the VMs is limited by its resource availability. Even though the resources of a VM are immutable in nature, we can tune the resource capacity of a container during execution as containers are vertically scalable [23,24]. Moreover, the monetary cost is proportional to the number of reserved resources of the VMs and their reservation time. Thus, to maximize the resource utilization of a VM, it should efficiently consolidate its containers. The CPU utilization of a VM v_k at time t can be measured as

$$ut_{v_k,t} = \frac{\sum_{j=1}^{n}(mips_{c_j,t} \cdot Y_{j,k,t})}{mips_{v_k}} \tag{3}$$

where $mips_{v_k}$ is the CPU capacity of the VM v_k represented in Million Instructions per second (MIPS) and $mips_{c_j,t}$ is the tuned MIPS of the container c_j at time t. Here, $mips_{v_k}$ is directly proportional to the number of CPU cores c and the frequency f of each core (i.e. $mips_{v_k} \propto c \times f$). The memory and disk size of VMs constrain the availability of these resources for its containers.

3.1 Execution Time

As a container executes one task at a time, it can dedicate its entire resources to the task. Container technology like docker[2] enables us to dynamically scale and configure the resources that a container can access. In our approach, we tune a container at the start of a task and do not alter the configurations until it completes. However, estimating the length of a task in MI (million instructions) is nontrivial, but several models for this estimation exist [25,26]. Hence, this work assumes knowledge of the length of the tasks. The execution time of the task τ_i allocated to a container c_j, denoted by $et(\tau_i, c_j)$, is calculated as

$$et(\tau_i, c_j) = \frac{mi_{\tau_i}}{mips_{c_j}}. \tag{4}$$

where, mi_{τ_i} is the length of the task τ_i in MI and $mips_{c_j}$ is the tuned MIPS of the container c_j.

3.2 Container Image Caching Time

In Cloud, the container images reside in a repository. Before instantiating a container c_j in a VM v_k, the VM

[1] https://aws.amazon.com/ec2/instance-types/

[2] https://docs.docker.com/config/containers/resource_constraints/

must download the corresponding container image ci_g. A VM can also cache container images to avoid delay. Container image caching time in a VM v_k is calculated as

$$ic(ci_g, v_k) = \frac{ci_g^{size}}{bw_{v_k}} \tag{5}$$

where ci_g^{size} is the size of the container image ci_g and bw_{v_k} is bandwidth of the VM v_k. If a VM requires caching multiple images, it does it in a certain order. We calculate the average utilization of the containers with the container image ci_g as

$$\overline{ut_{ci_g}} = \frac{1}{\delta t} \cdot \frac{1}{|C'|} \sum_{t-\delta t}^{t-1} \sum_{i=1}^{m} \cdot \sum_{c_j \in C'} X_{i,j,t} \tag{6}$$

for a window $t - \delta t$ to t, where $C' = \{c_j | c_j$ is instantiated using $ci_g\}$. We cache the container images in the order of their utilization. We consider a time window ($t - \delta t$ to t) for calculating the utilization of a container because the average over a longer period of time does not properly reflect its demand. Rather, the data from the recent past can predict resource requirements more accurately.

3.3 Data Transfer Time

In a workflow, the output of the predecessor task goes as an input to the dependent successor task. Some of the existing works [4,8] consider the data transfer to take place using global storage like Amazon S3. However, this may unnecessarily increase the network traffic. Moreover, it does not seem appropriate to transfer data via global storage between the tasks allocated close to each other. It would be better to consider direct data transfer and not via a global storage system. But for that, we must instantiate the container having the successor task before the termination of the predecessor task. Figure 1 illustrates such a data transfer.

Although container overlapping may increase the number of active VMs, it provides higher security, lower network traffic, and lesser transfer time than the Amazon S3-based shared storage. The data transfer time from container c_j running in VM $g(c_j)$ to container c_b running on VM $g(c_b)$ depends on the bandwidth of the VMs $g(c_j)$, $g(c_b)$ and also the topological

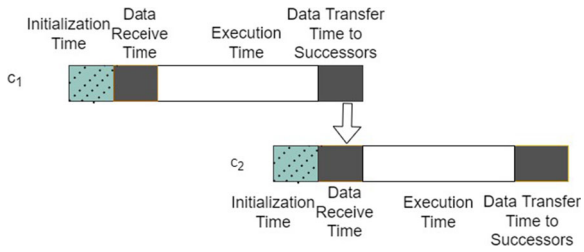


Fig. 1 Container overlapping for data transfer from container c_1 to container c_2

distance between the VMs. The topological distance introduces a multiplication factor γ_{c_j,c_b} defined as:

$$\gamma_{c_j,c_b} = \begin{cases} 0 & \text{if } g(c_j) = g(c_b) \\ \rho_1 & \text{if } g(c_j) \neq g(c_b) \text{ but } h(g(c_j)) = h(g(c_b)) \\ \rho_2 & \text{if } h(g(c_j)) \neq h(g(c_b)) \end{cases} \tag{7}$$

where $0 < \rho_1 < \rho_2$, and $h(g(c_j))$ and $h(g(c_b))$ represent the hosts in which the VMs $g(c_j)$ and $g(c_b)$ are placed.

The output data transfer time from a container c_j having the task τ_i to a container c_b having task $\tau_a = suc(\tau_i)$ is calculated as

$$dt_{c_j,c_b}^{wt} = \frac{dv_{jb} \cdot \gamma_{c_j,c_b}}{\min\{bw_{g(c_j)}, bw_{g(c_b)}\}} \tag{8}$$

where dv_{jb} is the volume of data to be transferred from c_j to c_b, $g(c_j)$ and $g(c_b)$ indicate the VMs in which the containers c_j and c_b are allocated, $bw_{g(c_j)}$ and $bw_{g(c_b)}$ indicate the bandwidth of the corresponding VMs.

If a task has multiple successor tasks, it forwards its output to all its successors. Hence, the total output data transfer time from a task τ_i allocated to a container c_j is

$$dt_{c_j,*}^{wt} = \sum_{c_b} \frac{dv_{jb} \cdot \gamma_{c_j,c_b}}{\min\{bw_{g(c_j)}, bw_{g(c_b)}\}} \quad \forall c_b \in suc(c_j) \tag{9}$$

Similarly, a task needs input data from all its predecessor tasks to start execution. The input data receive time of a container c_j from a container c_b is calculated as

$$dt_{c_b,c_j}^{rd} = \frac{dv_{bj} \cdot \gamma_{c_j,c_b}}{\min\{bw_{g(c_b)}, bw_{g(c_j)}\}} \tag{10}$$

The total input data receive time of a container c_j is

$$dt^{rd}_{*,c_j} = \sum_{c_b} \frac{dv_{bj} \cdot \gamma_{c_j,c_b}}{\min\{bw_{g(c_j)}, bw_{g(c_b)}\}} \quad \forall c_b \in pre(c_j) \tag{11}$$

During data transfer from a container to another, the bandwidth allocation for the data transfer is bounded by the available bandwidth of the enclosing VMs where the containers are deployed. If multiple containers send/receive data simultaneously and the bandwidth of the enclosing VM is shared in parallel transfer, the data transfer time is elongated for all the successors. Furthermore, the successor tasks can be started after completion of the data transfer operations as shown in Fig. 2. Due to parallel transfer, the tasks τ_2, τ_3 and τ_4 respectively deployed in containers c_2, c_3 and c_4 become ready to execute at the same time.

Hence, instead of performing parallel data transfer to the successors, sequential data transfer operations provide benefit in minimizing idle time of the containers, shown in Fig. 3. Here, a task can be started immediately after receiving data from the predecessors. In this example the successor tasks are ordered as τ_2, τ_3 and τ_4 which are deployed in containers c_2, c_3 and c_4 respectively.

It depicts that sequential transfer significantly reduces processing time of the containers. Thus, in this experiment, we have chosen sequential data transfer while send or receive operations involve inter-VM communication. If data is being transferred among a pair of containers residing in same VM, the bandwidth of the VM is not in use for the particular communication. Thus, before initiating data transfer to the successors, the transfer requests are ordered to ensure sequential transfer. Furthermore, a task can be started after receiving data from all the predecessors. The starting time of a container is determined depending on the scheduled data receive times of the container. The container needs to be started in advance by initialization time of the earliest scheduled data receive time.

3.4 Task Processing Time in a Container

The processing time of a task τ_i is dependent on many factors, like task initialization time $it(\tau_i)$, input data receive time dt^{rd}_{*,c_j}, execution time $et(\tau_i, c_j)$ and output data transfer time $dt^{wt}_{c_j,*}$. Thus, the processing time $pt(\tau_i, c_j)$ of a task τ_i in a container c_j is measured as

$$pt(\tau_i, c_j) = it(\tau_i) + dt^{rd}_{*,c_j} + et(\tau_i, c_j) + dt^{wt}_{c_j,*} \tag{12}$$

where initialization time is defined as

$$it(\tau_i) = \begin{cases} 0, & \text{if } c_j \in C \\ it(c_j), & \text{if } c_j \text{ is instantiated in an existing VM} \\ it(c_j) + it(v_k), & \text{if } c_j \text{ and } v_k \text{ are instantiated} \end{cases} \tag{13}$$

If a task is deployed on an existing container, then its initialization time is negligible. If the container is instantiated after the arrival of the task, then it will involve a container instantiation time of $it(c_j)$. This $it(c_j)$ will also include the container image caching time if the corresponding image is not present in the VM. If the VM is launched after the task arrival, then an additional VM instantiation time of $it(v_k)$ will be required.

Though task processing time in a container includes input read time (dt^{rd}_{*,c_j}) and output write time ($dt^{wt}_{c_j,*}$), the data transfer time between a pair of containers is counted once. The input data reading time of a container and the output data write time of its predecessor are overlapped, as the proposed system model considers direct data transfer from one container to another

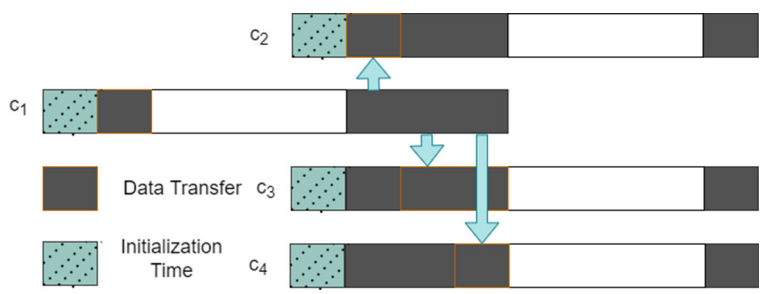


Fig. 2 Parallel data transfer to successor tasks

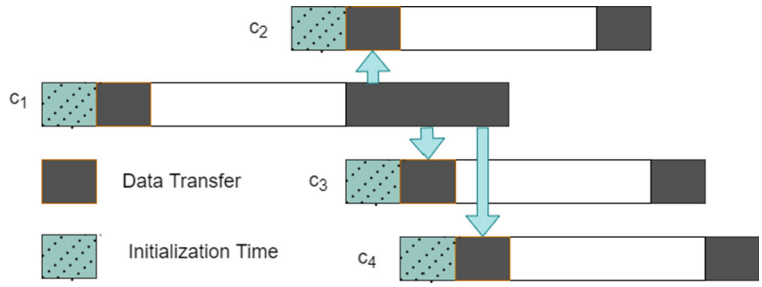


Fig. 3 Sequential data transfer to successor tasks

instead of writing the output of the predecessor to separate storage (like S3) and reading the data from that location afterward. Thus, even after a task is complete in a container, it should continue running until the completion of the transfer of data to the containers consisting of successor tasks. Also, containers with successor tasks should be running while the data transfer is in progress.

3.5 Pricing of VMs

One of the primary objectives of workflow scheduling is the minimization of monetary cost, which is directly associated with the VM deployment cost. The cost of running a VM instance is the product of $(tt(v_k) - st(v_k))$ and cost per unit time for that instance where $st(v_k)$ and $tt(v_k)$ represents the start time and termination time of the VM v_k respectively. In this simulation, we calculate the cost of VM reservation assuming AWS EC2 instance pricing on Per-Second Billing[3]. To determine the long-term cost of running multiple workflows, we consider a window of 5 hours and compute the cost of all VMs active in that window.

4 Proposed Solution

In this section, we have discussed our proposed algorithms to solve the problem defined in the previous section. Our approach to minimize the cost of VM deployment while satisfying the deadline comprises three major steps:

1. To minimize the number of VMs while completing the workflow within its deadline, we need to maximize the number of containers in the VMs. Each VM has a capacity in MIPS, and the containers executing it had to be run at a specific MIPS to satisfy the deadline for the workflow. In this context, we try to minimize the total MIPS taken by all containers in the critical paths of the workflow. We solve this problem by formulating an NLP (nonlinear program) and get an approximate solution using the Lagrange Multiplier Method.
2. While solving the NLP, we assume data transfer occurs in parallel. During the actual schedule, we assume serial data transfer to calculate the start time of a task when the finish times of its precedent tasks are already known. If the start time changes from the previous estimated value, the MIPs of the container are scaled accordingly.
3. Finally, we have an algorithm to auto-scale the containers and VMs to optimize resource usage.

Initially, we have to deploy several running containers to serve the upcoming requests without delay. This process must ensure the deployment of enough containers in VMs to run all kinds of tasks. Algorithm 1 initiates this process.

The environment initialization process launches containers from each container image to enable the immediate scheduling of a new workflow and instantiates a few VMs to accommodate the containers. Since we do not know the average workload in advance, we instantiate only those many VMs to accommodate one container instance of every container image type. In this regard, the workflows initially arriving may be delayed maximally by the instantiation time of the VMs, container image caching time, and container instantiation time.

Algorithm 1 receives the set of container images (CI) and a set of demand of average container instances of each container image type (ID), assuming it knows or can predict the set ID. The function $isDeployable(c_j, v_k)$ checks if VM v_k has sufficient

[3] https://aws.amazon.com/ec2/pricing/

Algorithm 1: Environment initialization.

input : Container Images, $CI = \{ci_1, ci_2, ci_3, \ldots, ci_o\}$
Instance Demand, $ID = \{id_1, id_2, id_3, \ldots, id_o\}$
output: Containers, $C = \{c_1, c_2, \ldots, c_n\}$; VMs, $V = \{v_1, v_2, \ldots, v_q\}$; Allocation map, Y

```
/* Initialization                                   */
initialize set of VMs, V ← φ
initialize set of containers, C ← φ
foreach container image ci_g do
    for i = 1 to id_g do
        set flag isAllocated ← false
        foreach v_k ∈ V do
            if isDeployable(c_j, v_k) then
                update flag isAllocated ← true
                break;
        if isAllocated = false then
            instantiate a new VM, v_k
            update V ← V ∪ v_k
        create container c_j having maximum, mips_c^max,
        on v_k based on ci_g
        set Y_{j,k,t} ← 1
        C ← C ∪ c_j
```

resources required for the container c_j. The Algorithm launches a new VM if the container is not deployable in the existing VMs. During environment initialization, we consider every container to have a MIPS capacity equal to $mips_c^{\max}$, the maximum MIPS of the container. Afterward, the MIPS values of the containers are tuned using our proposed policy presented in the following subsection. Though during environment initialization, ignoring data transfer requirements may not place the cooperating containers at topological proximity, the information related to data transfer would be available during workflow scheduling and will be of use during auto-scaling of the resources (presented in Section 4.4).

4.1 Critical-Path based MIPS Tuning for the Containers

Dynamic tuning of the MIPS of the containers enables effective consolidation of the containers in the VMs and improves the resource utilization of the VMs. This subsection presents the proposed deployment policy based on critical paths for a deadline-constrained workflow. Here, we determine the optimum MIPS requirement of each task and ensure that the workflow completes within its deadline. The workflow spans till its deadline to reduce the peak resource demand of the VMs. The process of finding optimal MIPS for all tasks involves the following steps,

1. Identifying the critical paths.
2. Finding MIPS for all tasks in disjoint critical paths assuming maximum data transfer time.
3. Finding MIPS for all other tasks assuming maximum data transfer time.
4. Further tuning of MIPS taking into account actual data transfer time.

4.1.1 Identifying critical paths

Determination of the Critical Paths of a workflow requires finding the execution time of the individual tasks along with their earliest and latest start times and earliest and latest finish times. Equation (4) gives the execution time of each task τ_i assuming $mips_c^{\max}$ is the CPU capacity of the container. The earliest start time of τ_i is:

$$est(\tau_i) = \begin{cases} 0, & \text{if } \tau_i \in T_h^{in} \\ \max\limits_{\tau_a \in pre(\tau_i)} \{est(\tau_a) + et(\tau_a, c_{f(\tau_a)}) + dt^{rd}_{c_a,c_i}\}, & \text{otherwise} \end{cases} \tag{14}$$

where $c_{f(\tau_a)}$ represents the container where τ_a is allocated and $dt^{rd}_{a,i}$ represents data transfer time from task τ_a to task τ_i. The earliest finish time of the task τ_i is:

$$eft(\tau_i, c_j) = est(\tau_i) + et(\tau_i, c_j). \tag{15}$$

The latest start time of the task τ_i is:

$$lst(\tau_i) = \begin{cases} ms_h - et(\tau_i, c_j), & \text{if } \tau_i \in T_h^{fin} \\ \min\limits_{\tau_a \in suc(\tau_i)} \{lst(\tau_a) - \Delta t_{i,a} - et(\tau_i, c_j)\}, & \text{otherwise} \end{cases} \tag{16}$$

where ms_h is the minimum makespan of the workflow wf_h. The latest finish time $lst(\tau_i)$ of the task of the workflow is:

$$lft(\tau_i) = lst(\tau_i) + et(\tau_i, c_j). \tag{17}$$

A task τ belongs to a critical path if $lft(\tau) = eft(\tau)$. We find the set of critical paths ($CP =$

$\{cp_1, cp_2, \ldots, cp_y\}$). Then the problem of minimizing the total MIPs of all tasks satisfying the deadline constraint of the workflow wf is formulated as a non-Linear Program (NLP) where $mips_{\tau_i}$, the MIPs assigned to task τ_i are the decision variables.

4.1.2 Finding MIPS of all tasks in disjoint critical paths

Let $< \tau_a, \ldots, \tau_b >$ be a sequence of tasks, each one predecessor of the next. We call such a sequence a subpath. If there is another subpath $< \tau_c, \ldots, \tau_d >$, such that $\tau_x = pre(\tau_a) = pre(\tau_c)$ and $\tau_y = suc(\tau_b) = suc(\tau_d)$, the execution time of these subpaths should be same to minimize the total MIPS (completing one earlier than the other does not help reduce the makespan). Let $S_p(\tau_x, \tau_y)$ denote all subpaths going from τ_x to τ_y and let $\mathcal{S}$ denote the set of all such $S_p(\tau_x, \tau_y)$ for different (τ_x, τ_y). Then, the NLP to find optimal MIPs is as follows:

$$\textbf{NLP All tasks} : \text{minimize} \sum_{\tau \in wf} mips_\tau \tag{18}$$

subject to the following constraints

$$(i) \quad \sum_{\tau \in P} \frac{mi_\tau}{mips_\tau} \leq dl^{wf_h} \quad \forall \ P \in CP$$

(ii) for each pair of subpaths $(p, q) \in S_p$

$$\sum_{\tau \in q} \frac{mi_\tau}{mips_\tau} = \sum_{\tau \in p} \frac{mi_\tau}{mips_\tau} \forall S_p \in \mathcal{S}$$

$$(ii) \quad mips_\tau > 0 \quad \forall \ \tau \in wf$$

Since the above NLIP would take a very long time to solve, we try an approximate solution by applying Lagrange's multiplier for each disjoint (not sharing any common task) critical path P as follows:

$$\textbf{NLP single CP} : \text{minimize} \sum_{\tau \in P} mips_\tau \tag{19}$$

subject to the constraints

$$\sum_{\tau \in P} \frac{mi_\tau}{mips_\tau} \leq dl^{wf_h}$$

where the critical path P have tasks $\tau_1, \tau_2, \ldots, \tau_{m'}$ with deadline dl^{wf_h}.

But we need to include in the constraints the data transfer delay from container to container. As we do not know the exact delay (depends on the bandwidth of VMs on which containers run and the topological distance of the VMs) while formulating the problem, the delays in the constraints are as for parallel transfer among VMs that are farthest apart. Later, during the tuning of MIPS, we assume sequential transfer, as shown in Fig. 3.

The updated constraint after taking this data transfer time into account is:

$$\sum_{\tau \in P} \left(\frac{mi_\tau}{mips_\tau} + dt_\tau \right) = dl^{wf_h} \tag{20}$$

where dt_τ is calculated as $dt_\tau = dt^{rd}_{*,c_j}$, considering τ is allocated to c_j. (we do not need to consider both read time and write time – when one task reads the data from its predecessor tasks, its predecessor tasks are also writing to it)

The objective function (19) and the constraint (20) together give the following Lagrangian :

$$L = \sum_{\tau \in P} mips_\tau + \lambda \left(\sum_{\tau \in P} \left(\frac{mi_\tau}{mips_\tau} + dt_\tau \right) - dl^{wf_h} \right) \tag{21}$$

where λ is Lagrange's multiplier. After differentiating L w.r.t each $\tau \in P$ and equating to 0, we get

$$mips_\tau = \sqrt{\lambda \cdot mi_\tau} \ \forall \tau \in P$$

$$\text{Now, } \sum_{\tau \in P} \frac{mi_\tau}{mips_\tau} = dl^{wf_h} - \sum_{\tau \in P} dt_\tau$$

$$\therefore \quad \sqrt{\lambda} = \frac{\sum_{\tau \in P} \sqrt{mi_\tau}}{dl^{wf_h} - \sum_{\tau \in P} dt_\tau}$$

Then, the MIPS of task τ is :

$$mips_\tau = \frac{\sum_{\tau \in P} \sqrt{mi_\tau}}{dl^{wf_h} - \sum_{\tau \in P} dt_\tau} . \sqrt{mi_\tau} \tag{22}$$

The MIPS of all tasks in disjoint critical paths are obtained by Algorithm 2 with start time 0 and finish time dl^{wf}.

Algorithm 2: MIPS distribution among the tasks of a path.

input : A path $p_g =< \tau_1, \tau_2, \ldots, \tau_{m'} >$;
start time st_g and finish time ft_g of p_g;
output: Tuned MIPS of the containers
/* calculate mips of the tasks on path p_g */
foreach $\tau_i \in p_g$ **do**
  $mips_{\tau_i} \leftarrow \frac{\sum \sqrt{mi_{\tau_i}}}{(ft_g - st_g) - \sum_{\tau_i \in p_g} dt_{\tau_i}} \cdot \sqrt{mi_{\tau_i}}$

4.1.3 MIPS of all tasks

To find the MIPS of all tasks, we execute the following steps:

1. Take from the list of critical paths one CP at a time, find the MIPs by applying Algorithm 2 with start time 0 and finish time dl^{wf_h}.
2. Remove the CPs from the list with tasks whose MIPS are already determined.
3. If the Q is empty, find the MIPS of tasks in subpaths.

But we observe that the total MIPS gets reduced if we process the CPs by sorting them in decreasing order of the number of tasks in them. Algorithm 3 does this by putting all CPs in a priority queue Q where the CP with the most number of tasks gets the highest priority.

4.1.4 Start time and Finish times of subpaths

A subpath sp_g is a sequence of tasks $< \tau_a \ldots, \tau_b >$ where we know the actual end time of predecessors of τ_a and actual start time of successors of τ_b. The start time st_g of subpath sp_g is:

$$st_g = \max_{\tau_x \in pre(\tau_a)} aet_{\tau_x}. \tag{23}$$

Similarly, the finish time ft_g of the path sp_g is calculated as follows:

$$ft_g = \min_{\tau_x \in suc(\tau_b)} ast_{\tau_x}. \tag{24}$$

Since the deadline for the tasks in subpath sp_g is $ft_g - st_g$, Algorithm 2 substitutes dl^{wf} in (22) by $ft_g - st_g$ to find the MIPs of the tasks $\tau_a \ldots, \tau_b$.

Algorithm 3: Path based MIPS tuning of the containers.

input : An workflow $wf_h = (T_h, E_h)$
output: Tuned MIPS of the containers
initialize $T' \leftarrow \phi$
calculate the set of critical paths, CP
sort the critical paths in the order of number of tasks in it, in descending order and put in Q
while Q *not empty* **do**
  take out from Q a critical path cp_g;
  Calculate MIPs of the tasks on path cp_g, using Algorithm 2
  foreach $\tau \in cp_g$ **do**
    calculate execution time $et(\tau)$ based on obtained mips
    update actual start time ast_τ and actual end time aet_τ
  update $T' \leftarrow T' \cup \tau \quad \in cp_g$
  remove from Q all critical paths having τ
while $|T'| < |T^h|$ **do**
  find a subpath sp_g having task $\tau \notin T'$
  determine start time st_g and finish time ft_g of path sp_g
  calculate mips of the tasks on the sub-path sp_g using Algorithm 2
  foreach $\tau \in cp_g$ **do**
    calculate execution time $et(\tau)$ based on obtained mips
    update actual start time ast_τ and actual end time aet_τ
  update $T' \leftarrow T' \cup \{\tau_i | \ \forall \ \tau_i \in sp_g\}$

4.1.5 Considering actual data transfer time

The initial computation of MIPs assumes parallel data transfer. As sequential data transfer is preferable, we find the aet (actual end time) considering a sequential transfer. Then the order in which a task τ's outputs go to its successor plays a role in determining the data transfer time. We divide the successors of a τ into two groups - those reading data from some other containers and those ready to accept data from τ. Data transfers to the latter group are ordered based on a priority value, where the priority value is:

$$\rho(\tau_i) = \frac{dt^{wt}_{c_a,c_j}}{\max\limits_{c_j \in suc(c_a)} dt^{wt}_{c_a,c_j}} \cdot \frac{mi_{c_j}}{\max\limits_{c_j \in suc(c_a)} mi_{c_j}} \tag{25}$$

This way, a successor task requiring a higher volume of input data and longer execution time gets higher priority. After completion of each transfer, any successor

in the first group may become ready. Then it comes into the latter group. After finding the time for sequential data transfer, if the successor task's actual start time differs from the one estimated by parallel transfer, the MIPs of the successor is scaled to accommodate the time gained.

4.2 Container Management

A task τ_i allocated to a container c_j reserves the container instance for its lifetime (represented by the task processing time). The task processing time consists of the initialization time $it(c_j)$, execution time $et(\tau_i, c_j)$ and the data transfer time $dt(c_j)$ as shown in (12). During this task processing time, the scheduling algorithm tunes the container's MIPS to the optimum MIPS requirement obtained using the Algorithm 3.

After the completion of a task, its output goes to its successors. A container c_j does not terminate immediately after completion as more than one task may be scheduled to execute in it. Each container uses a reservation schedule to manage all the tasks allocated to it. Each entry in the reservation schedule of a container c_j is denoted as $rs_{c_j} =< \tau_i, ast_{\tau_i}, aet_{\tau_i}, mips_{\tau_i}, dt_{\tau_i} >$, where ast_{τ_i} and aet_{τ_i} represent the actual start time and the actual end time of the task τ_i in c_j respectively, and dt_{τ_i} is the data transfer time. This data transfer time dt_{τ_i} has two parts, reading input data dt^{rd}_{*,c_j} and writing output data $dt^{wt}_{c_j,*}$ as shown in (11) and (9). Thus, the container c_j for task τ_i must be in a ready state at least dt^{rd}_{*,c_j} unit time before the starting of execution of task τ_i. In the same context, the container should exist at least $dt^{wt}_{c_j,*}$ unit time after the completion of execution of its task.

4.3 Online Multiple Workflow Deployment

The Algorithm 4 deals with the dynamic arrival of the workflow jobs. It is executed periodically to schedule all the workflows submitted in that scheduling clock. Given a set of workflows WF, we first determine the optimum MIPS requirement of each task in every workflow using Algorithm 3. The set T consists of all submitted workflows' tasks sorted in topological order. Then, taking them one at a time from the set T, we try to allocate them to a suitable container. A workflow has tasks of different types, each needing a specific container image with the necessary codebase and runtime environment. Thus, a task τ_i can be allocated only to a container deployed with a suitable container image ci_g. The set C_g represents the containers deployed with the image ci_g. A container c_j in C_g is eligible for allocation of the task τ_i if it satisfies the following two criteria. First, c_j must have a free slot from $est(\tau_i)$ to $lst(\tau_i) + et(\tau_i, c_j)$ in its reservation schedule. If not, the algorithm can shift the tasks in its reservation schedule to create a free slot. Second, the CPU capacity of the container must be greater than the MIPS requirement of the task, and if it is not, the underlying VM must have sufficient resources to enable tuning the con-

Algorithm 4: Workflow task allocation to containers.

input : Workflows, $WF = \{wf_1, wf_2, wf_3, \ldots, wf_z\}$; Container Images, $CI = \{ci_1, ci_2, ci_3, \ldots, ci_q\}$; Containers, $C = \{c_1, c_2, \ldots, c_n\}$; VMs, $V = \{v_1, v_2, \ldots, v_o\}$;
output: Task allocation to the containers, A

```
foreach workflow wf_h ∈ WF do
    determine the optimum MIPS requirement of each task in wf_h using Algorithm 3
initialize the set of tasks, T ← ⋃_{h=1}^{z} T_h and rearrange the tasks T using Topological ordering
foreach task τ_i ∈ T do
    select the containers C_g running in V and instantiated using the container image ci_g where τ_i can be deployed
    calculate container affinity of τ_i for all the containers in C_g using (26)
    rearrange containers C_g in descending order of affinity
    set c' ← null; set actual start time τ_i^ast ← τ_a^lst
    foreach c_j ∈ C_g do
        calculate execution time l_i ← mi_{τ_i} / mips_{c_j}
        if c_j has free slot l_i from τ_i^ast to τ_i^aft then
            calculate start time st' based on availability in reservation table of c_j
            if st' < τ_i^ast update c' ← c_j; τ_i^ast ← st';
    if c' ≠ null then
        allocate τ_i to c_j and update reservation table of c_j and update τ_a^est, τ_a^eft, τ_a^lst and τ_a^lft, ∀τ_a ∈ suc(τ_i)
    else
        failed to allocate
Auto-Scale Containers and VMs using Algorithm 5
```

tainer's MIPS to the required value. During task allocation, we give preference to the containers already having the necessary free slot and thus do not disturb the existing entries in the reservation schedules of the containers.

If multiple containers are eligible to execute a task τ_i, the container c_j best selected for the task depends on its resource affinity. For container selection, our prime objective is to consolidate the containers into a minimum number of VMs that results in maximum resource utilization of the active VMs. The container affinity of task τ_i is calculated as:

$$\zeta_{\tau_i,c_j} = \alpha \cdot \frac{1}{\mu^{vh} - ut_{g(c_j),t}} + (1-\alpha) \cdot \frac{1}{\sum_b \gamma_{c_j,c_b}} \quad (26)$$

where $0 < \alpha < 1$, and μ^{vh} is the upper threshold of CPU utilization of a VM. Here $g(c_j)$ indicates the VM v_k on which the container c_j is allocated, and $ut_{g(c_j),t}$ is the CPU utilization of the VM at time t, and γ_{c_j,c_b} (7) determines the communication delay between c_j and c_b (c_b is the container where τ_i's predecessors would run). Finally, the container c_j having the highest affinity value ζ_{τ_i,c_j} gets the task τ_i and the MIPS of the container is tuned as discussed in Section 4.1.

4.4 Auto-Scaling of Containers and VMs

At the end of every scheduling period, the algorithm analyzes the CPU utilization of containers of different types (based on the container image type) and decides to scale (up/down) resources discussed as follows. The upper threshold of CPU utilization μ^{ch} and lower thresholds of CPU utilization μ^{cl} of the containers act as configurable parameters. Similarly, upper and lower thresholds of resources of the VMs are μ^{vh} and μ^{vl}. The containers and the VMs are scaled if their CPU utilization violates the upper or the lower threshold (Algorithm 5).

In our auto-scaling approach, if the average CPU utilization $\overline{ut_{ci_g}}$ of the containers deployed with the image ci_g exceeds the upper threshold μ^{ch}, then $\lceil(\overline{ut_{ci_g}} - \mu^{ch}) * |C_g|\rceil$ number of containers are instantiated in the VMs, and they form the set C_g. We first try to launch a container in one of the existing VMs in V. If none of the existing VMs have sufficient resources for the container, then only a new VM is launched and added to the set V. The new VM gets the container with image ci_g. Again, if $\overline{ut_{ci_g}}$ is lower than μ^{cl}, some idle containers are terminated by sorting them in decreasing order of their idle period. Similarly, if the average CPU utilization $\overline{ut_V}$ of the VMs in V exceeds μ^{vh}, then $\lceil(\overline{ut_V} - \mu^{vh}) * |V|\rceil$ number of VMs are to be instantiated and added to V. Again, if $\overline{ut_V}$ is lower than μ^{vl}, the idle VMs are terminated to reduce the resource renting cost. In this way, auto-scaling after every schedul-

Algorithm 5: Auto-scaling of containers and VMs.

input : VMs, V; Containers, C; Reservation table of containers, RT; Container allocation vector, B
output: VMs (V); Containers (C)

based on historical data, calculate t' as average length of workflows
`/* VMs shall be scaled up in 2 min. advanced as instantiation takes 1.5-2.0 min. */`
calculate average workload during $(t - t')$ to t where t is current clock
`/* auto scaling of containers */`
for $g = 1$ *to* o **do**
  select the containers C_g, intantiated using container image ci_g running on VMs V
  calculate average utilization $\overline{ut_{ci_g}}$ during $(t - t')$ based on the reservation tables
  if $\overline{ut_{ci_g}} \geq \mu^{ch}$ **then**
    `/* need to scale up if utilization exceeds high-threshold */`
    add $\lceil(\overline{ut_{ci_g}} - \mu^{ch}) * |C_g|\rceil$ number of containers in C'_g, which are to be deployed
    rearrange V in descending order of utilization
    foreach $c_j \in C'_g$ **do**
      set $flag \leftarrow false$ **for** $v_k \in V$ **do**
        if $ut_{v_k} + mips_{c_j}/mips_{v_k} < \mu^{vh}$ *and* v_k *has sufficient memory and storage* **then**
          instantiate c_j in v_k;
          update $flag \leftarrow true$
      if $flag = false$ **then**
        instantiate a new VM and add it to V
  else
    while $\overline{ut_{ci_g}} < \mu^{cl}$ *and more free containers in* C_g **do**
      remove the a container from C_g in which tasks are not allocated;
      update $\overline{ut_{ci_g}}$

ing period reduces the waiting time for the tasks and improves the schedulability of the workflows.

5 Performance Evaluation

For analysis of the results of our experiments, we focus on the following performance metrics: i) average makespan, ii) average number of active VMs, iii) resource wastage of active VMs, and iv) VM deployment cost. The first three are computed by taking their average over a window of a specific size, and the last one is the sum of costs for all VMs deployed during the window.

Average Makespan The makespan of a task is the time from its submission to its completion. We measure the make-span of all tasks completed within the window and then compute their average.

Resource Wastage The resource wastage of an active VM expressed in percentage is 100 minus its average utilization over its lifetime. For example, if the utilization is 90%, the resource wastage is 10%. We compute the average resource wastage as the average resource wastage of all VMs active for some time in the window.

The metrics - number of active VMs and cost of deployment are closely related to resource wastage. Running VMs at higher utilization allows the completion of tasks with fewer active VMs and reduces costs.

We compare the performance of the proposed policy with two other works dealing with workflow scheduling on WaaS platforms –*Elastic resource Provisioning and Scheduling (EPSM)* [8] and *Dynamic Deadline and Budget-aware Workflow Scheduling* (DDBWS) [27].

EPSM [8] presents a dynamic and scalable algorithm to schedule multiple multi-tenant Workflows on WaaS platforms to minimize the total cost of resource reservation while meeting the individual deadline of workflows. This scheduling technique uses a uniform ranking based on time and costs. The DWS [28] policy schedules multiple workflows dynamically submitted by different users in a WaaS while minimizing the execution cost and fulfilling the deadline constraints. As DWS does not consider the multi-resource packing scheme or the budget and does not take advantage of containerization, it is unsuitable for comparison with our work. DDBWS [27] is a heuristic-based scheduling algorithm that reduces execution costs by packing multiple tasks onto a single VM (like our proposed method) using containerized deployment.

5.1 Simulation

The simulation has been carried out on a Laptop having Intel(R) Core(TM) i3-7100U CPU @ 2.40GHz 2.40 GHz and 4 GB main memory. We use Java to design the simulator.

Due to the unavailability of the required dataset, we use a dataset synthesized from standard scientific workflows like Montage, Epigenomics, SIPHT, LIGO, and CyberShake. These workflows are heterogeneous - their makespans vary from 5 minutes to 2 hours. Inspired by Silva's work, [29], in the dataset preparation, we assume that the workflows arrive randomly following Poisson distribution.

Influenced by the works in [8,30], we take the average bandwidth for VMs as 500 Mb/s and the mean container size as 600 MB. We set the container provisioning delay to 10 seconds [8] and the VM provisioning delay to 100 seconds [31]. Pegasus's workflow generator tool[4] generates synthesized real-life traces [32,33] for workflows. A workload combines different types of workflows with different sizes. We assume the simulation environment has sufficient hosts to deploy VMs. These VMs are homogeneous in CPU and memory resources - all have 4-core CPUs and 4 GB of memory.

5.2 Analysis of Results

We ran the experiment in a long-running environment and measured the performance metrics over a window of 5 hours duration. The scheduling clock length is 1 minute (scheduling decisions are taken at 1-minute intervals).

We have compared the performance of the proposed algorithm with EPSM and DDBWS [27] in two different contexts. In one, we set the deadlines to values specified by the users. On the other, we use the makespans obtained by DDBWS [27] as deadlines. For both contexts, we take results by i) varying *the number of tasks of the workflows keeping the arrival rate fixed* and ii) varying *the arrival rate of the workflows keeping the size of workflows fixed.*

[4] https://workflowhub.eu/

The results plotted in the figures are the averages from several simulations, each with a window of 5 hours (long-running environment). We calculate the monetary cost assuming Amazon on-demand VM instances with per-second billing[5]. The experimental results are presented in Figs. 4, 5, 6 and 7

As mentioned in subsection 4.1.3, we get improved performance if we process the critical paths after prioritizing them according to their number of tasks. To underline this point, we present the result for both versions. The version *Proposed (B)* prioritizes the critical paths as per Algorithm 3, and the version *Proposed (A)* does not.

5.2.1 Satisfying user specified deadline

In workflow deployment, meeting deadlines is of utmost importance to ensure the successful completion of the goals as it critically impacts the overall effectiveness and efficiency of workflows. Here, we present the results where the proposed algorithm and others strictly satisfy the deadline specified by the users.

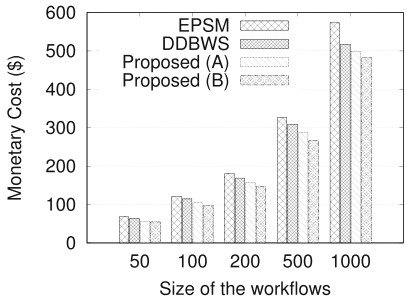
Varying size of the workflows Figure 4 depicts the results of the simulations where the arrival rate of workflows is 100 per scheduling clock, and the size of the workflow varies from 50 to 1000 tasks. Figure 4a shows the monetary cost of VM provisioning, and Fig. 4b, the average active VM instances. Both indicate an increasing trend with the increase in the size of workflows. Whereas the variation in workflow size has little impact on the makespan (Fig. 4d) and wastage of VM resources (Fig. 4c). As the workflows arrive at the same rate with more tasks per workflow, they require more resources, causing an increase in the number of active VMs and monetary costs. As the containers and VMs are scaled up to accommodate larger workflows, the makespan, and average utilizations almost remain the same. Among the existing policies, *DDBWS* [27] gives the best performance, and both proposed policies (A and B) have significantly lower monetary cost, average active VM instances, and resource wastage of VMs compared to *DDBWS*. However, the difference is more pronounced for *Proposed B* that process the critical paths in increasing order of their tasks. In terms of makespan), *Proposed A* and *Proposed B* give identical results and have values slightly more than those using *EPSM* or *DDBWS*. However, having longer makespans is not a demerit of the proposed policy. Rather, they utilize the full extent of the specified deadlines to optimize the number of active VMs and monetary costs.

Algorithm 3, optimally tunes the MIPS of containers, enabling more containers in a VM and reducing the average demand of active VMs. In comparison with *EPSM*, the *DDBWS* policy reduces average demand for VM resources by 5.48 to 10.92 %. Furthermore, the average demand for VM resources is reduced by 11.41 to 15.16 % using *Proposed (A)* policy and 13.95 to 18.56 % using Proposed (B) policy (Fig. 4b) in comparison with *EPSM*. Fewer active VMs result in lower monetary costs. From Fig. 4a, we observe that the monetary cost of VM provisioning, in comparison with *EPSM*, is lower by 5.43 to 9.85 for *DDBWS*, by 12.32 to 16.91% for *Proposed (A)* and by 15.52 to 19.85 % for *Proposed B*. Algorithm 5 performs auto-scaling of containers and VMs and keeps the utilization of VMs at high values close to the upper threshold μ^{vh}, thereby reducing the average resource wastage. Compared to EPSM, the average resource wastage of the VMs is lower by 2.29 to 3.81 % in DDBWS, 4.89 to 5.81 % in Proposed (A), and 5.49 to 6.81 % in Proposed (B).

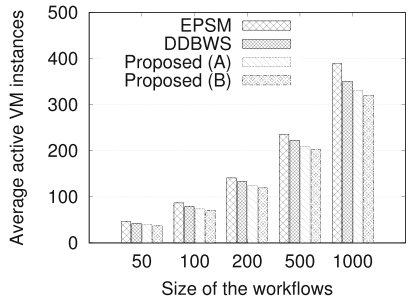
Varying arrival rate of the workflows The results taken by varying the arrival rate of the workflows (from 50 to 150 per scheduling clock) are presented in Fig. 5. It shows that the monetary costs (Fig. 5a), and average number of active instances (Fig. 5b) increase significantly with an increase in arrival rate but the makespan (Fig. 5d) and resource wastage (Fig. 5c) do not change that much. The reason is that to complete the tasks within their deadlines when the arrival rate increases (the workflows have fixed sizes), the scheduling algorithms have to deploy more VMs during the same window. As for the case with varying workflow sizes, the proposed policy has significantly lower monetary cost, average number of active VMs, and resource wastage than EPSM and DDBWS.

The results presented in Fig. 5 show that the monetary cost of VM reservation (Fig. 5a) decreases by

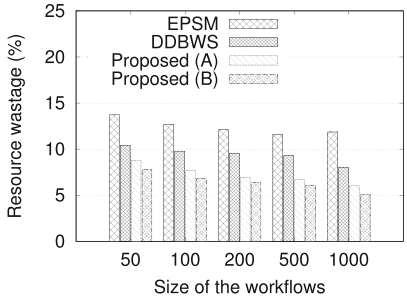
[5] https://cloud.google.com/compute/vm-instance-pricing#billingmodel



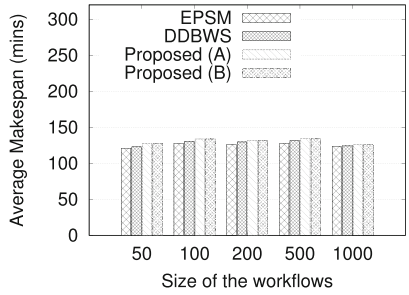
(a) Monetary cost of VM reservation.



(b) Average active VM instances.



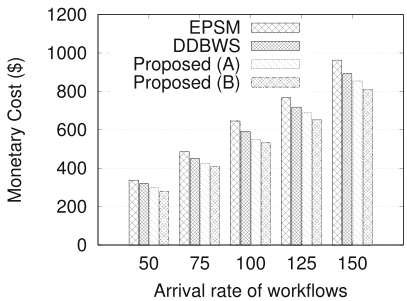
(c) Average resource wastage of active VM.



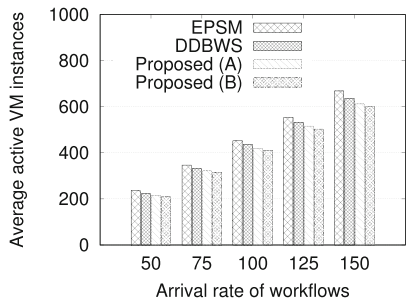
(d) Average makespan of the workflows.

Fig. 4 Performance of workflow deployment algorithm for user-specified deadline. The workflows vary in size but have a fixed arrival rate of 100 per scheduling clock

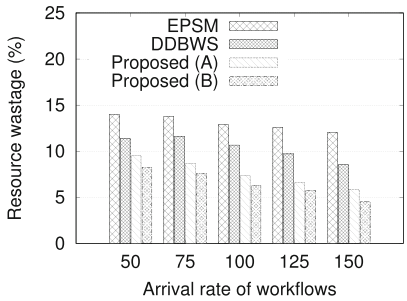
10.33 to 14.64% for *Proposed (A)* in comparison with EPSM. For *Proposed (B)*, the corresponding reduction is 15.03 to 17.75%. The average peak demand of active VMs (Fig. 5b) gets reduced by 6.84 to 8.40% for *Proposed (A)* and 9.19 to 10.69% for *Proposed (B)* (in comparison with EPSM). The resource wastage of the



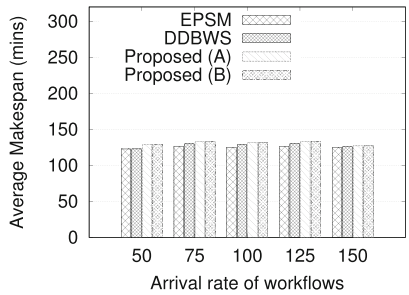
(a) Monetary cost of VM reservation.



(b) Average active VM instances.

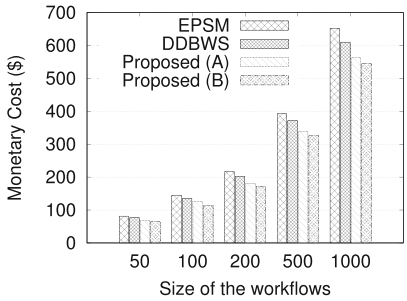


(c) Average resource wastage of active VM.

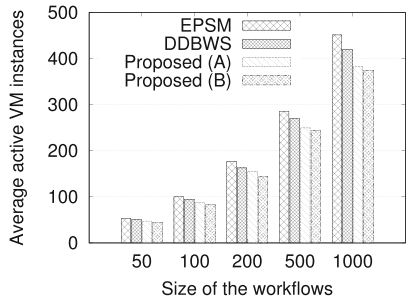


(d) Average makespan of the workflows.

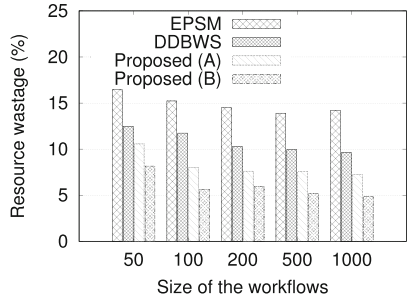
Fig. 5 Performance of workflow deployment algorithm for user-specified deadline. The arrival rate of workflows varies from 50 to 150 per scheduling clock



(a) Monetary cost of VM reservation.



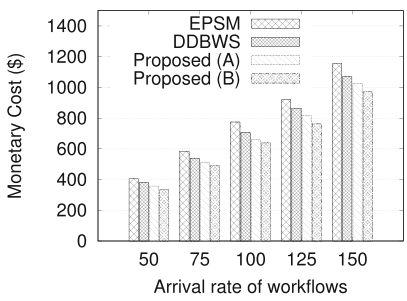
(b) Average active VM instances.



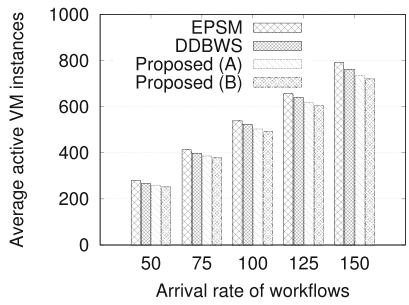
(c) Average resource wastage of active VM.

Fig. 6 Simulation results of workflows deployment in containerized cloud environment varying workflow size while arrival rate of workflows is 100 per scheduling clock. The simulation window size is 5 hrs

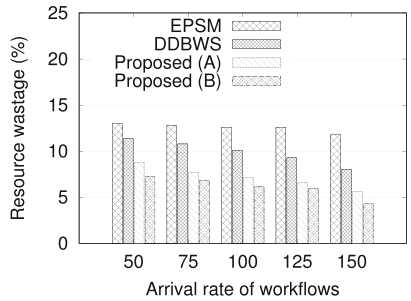
active VMs in *Proposed (A)* and *Proposed (B)* policies in comparison with EPSM are 4.51 to 6.21% and 5.78 to 7.55% respectively. The monetary cost, active VMs, and resource wastage for DDBWS are lower by 5.61 to 8.75 %, 3.40 to 5.59%, and 2.20 to 3.51% in comparison to EPSM.



(a) Monetary cost of VM reservation.



(b) Average active VM instances.



(c) Average resource wastage of active VM.

Fig. 7 Performance of workflow deployment algorithm varying arrival rate of the workflows between 50 to 150 per scheduling clock

5.2.2 Deadlines set to minimum makespans

The results with user-specified deadlines are somewhat biased in favor of the proposed algorithms as they give longer makespans, though within the deadline. Hence, to make a fairer comparison, we set the deadlines to the makespans given by *DDBWS* (it gives the shortest makespans among all algorithms) and study the other performance metrics like monetary cost, average active VMs and wastage of active VMs. As all the policies complete the workflows within the minimum makespan, we do not plot the makespan.

Varying size of the workflows Figure 6 shows that the monetary cost of VM reservation (Fig. 6a) decreases by 12.86 to 16.26% for *Proposed (A)* in comparison with EPSM. For *Proposed (B)*, the corresponding reduction is 16.51 to 21.94%. The average peak demand of active VMs (Fig. 6b) gets reduced by 11.31 to 15.07% using the Proposed (A) policy (in comparison with EPSM). The corresponding figures for Proposed (B) are 14.86 to 18.44%. The average number of active VMs, monetary cost and resource wastage of the active VMs, compared to EPSM, is lower nearly by 5.61 to 8.15%, 5.30 to 6.53%, and 3.48 to 4.58% for DDBWS, Proposed (A) and Proposed (B) policies respectively as shown in Fig. 6c.

We also ran many simulations considering different deadlines (from minimum makespan to user-specified deadline). However, we do not include those results in this manuscript as they depict similar graphs.

Varying arrival rate of workflows Figure 7 shows the results when the arrival rate of workflows varies between 50 to 150 per scheduling clock, and each workflow has 50 to 1000 tasks. The figures show that DDBWS performs better than EPSM, and the proposed policies (A and B) do better than DDBWS. Compared to EPSM, DDBWS gives average active VMs, monetary cost and average resource wastage of active VMs lower by 2.60 to 4.91%, 6.11 to 8.75% and 2.32 to 4.42%. The proposed (A) policy reduces the number of active VMs by 5.71 to 7.48%, the cost by 11.31 to 14.64% and the resource wastage of the active VMs by 4.88 to 7.21%. Corresponding reductions for the Proposed (B) policy are 8.08 to 10.05%, 15.98 to 17.75% and 6.70 to 8.76%.

We observe that even by setting the minimum makespans given by DDBWS, the proposed policies A and B perform better than EPSM and DDBWS but to a lesser extent compared to the case of the user-specified deadline.

6 Conclusion

This paper proposes resource-aware scheduling policies for deadline-sensitive workflows in the Cloud environment where containers in VMs execute workflow tasks. They use non-linear optimization using Lagrange's multiplier to minimize the MIPS of containers while ensuring the execution of workflows within the deadline (user-specified or minimum makespan). This way, VMs can accommodate more containers reducing energy and monetary costs. The proposed policy (B), which processes the critical paths in order of the number of tasks, performs better than policy (A). We also scale the containers and VMs to handle the dynamic situation where multiple workflows arrive and terminate. Experimental results validate the efficacy of the proposed policy compared to other state-of-art workflow scheduling techniques. We measure the performance metrics by experimenting with variations in the number of tasks and the arrival rate of the workflows. DDBWS performs best among the earlier works aiming to minimize cost and has a smaller makespan than the proposed policies. For a fair comparison, we have set the deadlines of workflows to the makespans obtained by DDBWS and measured the other performance metrics where we observe a decrease in cost in the range of around 15% for Proposed (A) and 20% for proposed (B). Under the same deadline constraints, the proposed policies fare better than the other existing ones in terms of the average peak demand of active VMs and the resource wastage of the active VMs.

As for future research, it would be interesting to investigate if the use of prediction of workflow arrivals gives improved schedules. We also intend to design workflow scheduling in a multi-cloud environment.

Funding No funding was obtained for this study.

Declarations

Competing interests I declare that the authors have no competing interests as defined by Springer, or other interests that might be perceived to influence the results and/or discussion reported in this paper.

References

1. Zhao, Y., Li, Y., Raicu, I., Lu, S., Lin, C., Zhang, Y., Tian, W., Xue, R.: A service framework for scientific workflow management in the Cloud. IEEE Trans. Serv. Comput. **8**(6), 930–944 (2014)
2. Zhao, Y., Li, Y., Raicu, I., Lu, S., Tian, W., Liu, H.: Enabling scalable scientific workflow management in the cloud. Futur. Gener. Comput. Syst. **46**, 3–16 (2015)
3. Ahmad, Z., Nazir, B., Umer, A.: A fault-tolerant workflow management system with quality-of-service-aware scheduling for scientific workflows in cloud computing. Int. J. Commun. Syst. **34**(1), 4649 (2021)
4. Hilman, M.H., Rodriguez, M.A., Buyya, R.: Resource-sharing policy in multi-tenant scientific workflow as a service platform (2019). Available from https://doi.org/10.48550/arXiv.1903.01113
5. Stavrinides, G.L., Karatza, H.D.: An energy-efficient, Qos-aware and cost-effective scheduling approach for real-time workflow applications in cloud computing systems utilizing DVFS and approximate computations. Futur. Gener. Comput. Syst. **96**, 216–226 (2019)
6. Qin, Y., Wang, H., Yi, S., Li, X., Zhai, L.: An energy-aware scheduling algorithm for budget-constrained scientific workflows based on multi-objective reinforcement learning. J. Supercomput. **76**(1), 455–480 (2020)
7. Tarafdar, A., Karmakar, K., Khatua, S., Das, R.K.: Energy-efficient scheduling of deadline-sensitive and budget-constrained workflows in the cloud. In: International conference on distributed computing and internet technology. Springer, pp. 65–80 (2021)
8. Rodriguez, M.A., Buyya, R.: Scheduling dynamic workloads in multi-tenant scientific workflow as a service platforms. Futur. Gener. Comput. Syst. **79**, 739–750 (2018)
9. Taibi, D., Lenarduzzi, V., Pahl, C.: Architectural patterns for microservices: a systematic mapping study. In: CLOSER 2018: Proceedings of the 8th international conference on cloud computing and services science. SciTePress, Funchal (2018)
10. Wang, J., Korambath, P., Altintas, I., Davis, J., Crawl, D.: Workflow as a service in the cloud: architecture and scheduling algorithms. Procedia Comput. Sci. **29**, 546–556 (2014)
11. Esteves, S., Veiga, L.: Waas: workflow-as-a-service for the cloud with scheduling of continuous and data-intensive workflows. Comput. J. **59**(3), 371–383 (2016)
12. Bharathi, S., Chervenak, A., Deelman, E., Mehta, G., Su, M.-H., Vahi, K.: Characterization of scientific workflows. In: 2008 third workshop on workflows in support of large-scale science. IEEE, pp. 1–10 (2008)
13. Rizvi, N., Ramesh, D.: Fair budget constrained workflow scheduling approach for heterogeneous clouds. Clust. Comput. **23**(4), 3185–3201 (2020)
14. Karmakar, K., Das, R.K., Khatua, S.: Resource scheduling of workflow tasks in cloud environment. In: 2019 IEEE international conference on advanced networks and telecommunications systems (ANTS). IEEE, pp. 1–6 (2019)
15. Karmakar, K., Das, R.K., Khatua, S.: Resource scheduling for tasks of a workflow in cloud environment. In: International conference on distributed computing and internet technology. Springer, pp. 214–226 (2020)
16. 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)
17. Qasha, R., Cala, J., Watson, P.: Dynamic deployment of scientific workflows in the cloud using container virtualization. In: 2016 IEEE international conference on cloud computing technology and science (CloudCom). IEEE, pp. 269–276 (2016)
18. Liu, K., Aida, K., Yokoyama, S., Masatani, Y.: Flexible container-based computing platform on cloud for scientific workflows. In: 2016 international conference on cloud computing research and innovations (ICCCRI). IEEE, pp. 56–63 (2016)
19. Chen, H., Zhu, X., Liu, G., Pedrycz, W.: Uncertainty-aware online scheduling for real-time workflows in cloud service environment. IEEE Trans. Serv. Comput. **14**(4), 1167–1178 (2018)
20. B Burkat, K., Pawlik, M., Balis, B., Malawski, M., Vahi, K., Rynge, M., da Silva, R.F., Deelman, E.: Serverless Containers–rising viable approach to Scientific Workflows. In: 17th International Conference on eScience (eScience). IEEE, pp. 40-49 (2021)
21. Ranjan, R., Thakur, I.S., Aujla, G.S., Kumar, N., Zomaya, A.Y.: Energy-efficient workflow scheduling using container-based virtualization in software-defined data centers. IEEE Trans. Ind. Inform. **16**(12), 7646–7657 (2020)
22. Bao, L., Wu, C., Bu, X., Ren, N., Shen, M.: Performance modeling and workflow scheduling of microservice-based applications in clouds. IEEE Trans. Parallel Distrib. Syst. **30**(9), 2114–2129 (2019)
23. Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., Merle, P.: Autonomic vertical elasticity of docker containers with Elasticdocker. In: 2017 IEEE 10th international conference on cloud computing (CLOUD). IEEE, pp. 472–479 (2017)
24. Paraiso, F., Challita, S., Al-Dhuraibi, Y., Merle, P.: Model-driven management of docker containers. In: 2016 IEEE 9th international conference on cloud computing (CLOUD). IEEE, pp. 718–725 (2016)
25. Wu, Q., Datla, V.V.: On performance modeling and prediction in support of scientific workflow optimization. In: 2011 IEEE world congress on services. IEEE, pp. 161–168 (2011)
26. Sadjadi, S.M., Shimizu, S., Figueroa, J., Rangaswami, R., Delgado, J., Duran, H., Collazo-Mojica, X.J.: A modeling approach for estimating execution time of long-running scientific applications. In: 2008 IEEE international symposium on parallel and distributed processing. IEEE, pp. 1–8 (2008)
27. Saeedizade, E., Ashtiani, M.: Ddbws: A dynamic deadline and budget-aware workflow scheduling algorithm in workflow-as-a-service environments. J. Supercomput. **77**(12), 14525–14564 (2021)
28. Arabnejad, V., Bubendorfer, K., Ng, B.: Dynamic multi-workflow scheduling: a deadline and cost-aware approach for commercial clouds. Futur. Gener. Comput. Syst. **100**, 98–108 (2019)
29. Silva, R.F., Pottier, L., Coleman, T., Deelman, E., Casanova, H.: Workflowhub: community framework for enabling scientific workflow research and development. In: 2020

IEEE/ACM workflows in support of large-scale science (WORKS). IEEE, pp. 49–56 (2020)
30. Piraghaj, S.F., Dastjerdi, A.V., Calheiros, R.N., Buyya, R.: Containercloudsim: an environment for modeling and simulation of containers in cloud data centers. Softw. Pract. Experience **47**(4), 505–521 (2017)
31. Mao, M., Humphrey, M.: A performance study on the VM startup time in the cloud. In: 2012 IEEE fifth international conference on cloud computing. IEEE, pp. 423–430 (2012)
32. Da Silva, R.F., Chen, W., Juve, G., Vahi, K., Deelman, E.: Community resources for enabling research in distributed scientific workflows. In: 2014 IEEE 10th international conference on e-science, vol. 1. IEEE, pp. 177–184 (2014)
33. Silva, R.F.d., Chen, W., Juve, G., Vahi, K., Deelman, E.: Community resources for enabling research in distributed scientific workflows. In: Proceedings of the 2014 IEEE 10th international conference on e-science-volume 01, pp. 177–184 (2014)