Resource-sharing Policy in Multi-tenant Scientific Workflow as a Service Platform

Muhammad H. Hilman, Maria A. Rodriguez, and Rajkumar Buyya, *Fellow, IEEE*

Abstract—Increasing adoption of scientific workflows in the community has urged for the development of multi-tenant platforms that provides these workflows execution as a service. Workflow as a Service (WaaS) concept has been brought up by researchers to address the future design of Workflow Management Systems (WMS) that can serve a large number of users from a single point of service. This platform differs from a traditional WMS in handling the workload of workflows at runtime. A traditional WMS is usually designed to execute a single workflow in a dedicated process while the WaaS platforms enhance the process by exploiting multiple workflows execution in a resource-sharing environment model. In this paper, we explore a novel resource-sharing policy to improve system utilization and to fulfill various Quality of Service (QoS) requirements from multiple users. We propose an **E**lastic **B**udget-constrained resource **P**rovisioning and **S**cheduling algorithm for **M**ultiple workflows designed for WaaS platforms that is able to reduce the computational overhead by encouraging resource-sharing policy to minimize workflows' makespan while meeting user-defined budget. Our experiments show that the EBPSM algorithm is able to utilize the resource-sharing policy to achieve higher performance in terms of minimizing the makespan compared to the state-of-the-art budget-constraint scheduling algorithm.

Index Terms—Resource-sharing policy, Workflow as a Service, Budget-constrained scheduling, Scientific workflows.

1 Introduction

Scientific workflows have accelerated the triumph of scientific missions on today's multi-discipline sciences [1]. This technology orchestrates and automates scientific applications in a way that reduces the complexity of managing scientific experiments. These workflows are composed of numerous interconnected tasks that represent data dependency and process flowing between them. Scientific workflows are widely known by their requirements of massive computational resources to run. Therefore, these resource-intensive applications are deployed in distributed systems with a high capacity of storage, network, and computing power to get a reasonable processing time.

Cloud computing has become a beneficial infrastructure for deploying scientific workflows. The cloud services, especially the Infrastructure as a Service (IaaS) offerings, provide a pseudo-infinite pool of resources that can be leased on-demand with a pay-per-use billing scheme. This pay-as-you-go model substantially eliminates the need for having an upfront investment for massive computational resources. IaaS provides the virtualized computational resource in the form of Virtual Machine (VM) with pre-defined CPU, memory, storage and bandwidth in various pre-configured bundling (i.e., VM types). The users then can elastically acquire and release as many VMs as they need and the providers generally charge the resource usage per time slot (i.e., billing period).

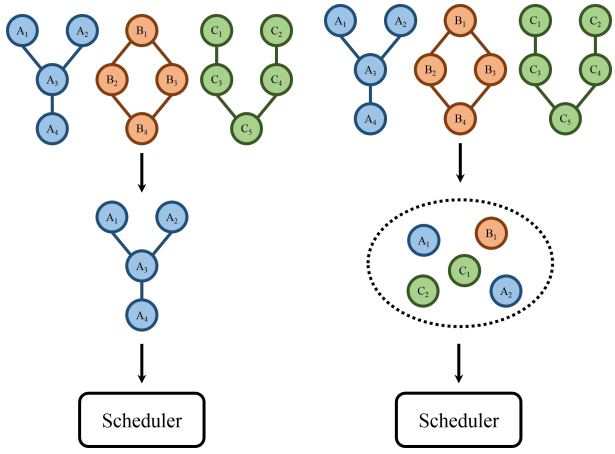
Designing algorithms for scheduling scientific workflows execution in clouds is not trivial. The clouds natural features evoke many challenges involving the strategy to decide what type of VMs that should be provisioned and when to acquire and release the VMs to get the most efficient scheduling result. The trade-off between having a faster execution time and an economical cost is something that must be carefully considered in leasing a particular cloud instance [2]. Other challenging factors are performance variation of VMs and uncertain overhead delays that might come from a virtualized backbone technology of clouds, geographical distribution, and its multi-tenancy [3].

These problems have attracted many computer scientists into cloud workflow scheduling research to fully utilize the clouds' capabilities for efficient scientific workflows execution [4] [5]. The majority of those studies focus on the scheduling of a single workflow in cloud environments. In this model, they assume a single user utilizes a Workflow Management System (WMS) to execute a particular workflow's job in the clouds. The WMS manages the execution of the workflow so that it can be completed within the defined Quality of Service (QoS) requirements. Along with the growing trend of scientific workflows adoption in the community, there is a need for platforms that provide scientific workflows execution as a service.

Workflow as a Service (WaaS) is an evolving idea which offers scientific workflows execution as a service to the scientific community. The platforms extend WMS technology that is commonly used for handling an individual execution of scientific workflow to serve a more significant number of users in a single point of service. Although some of the traditional cloud workflow scheduling algorithms can be extended for this problem, the specific nature of WaaS environments creates a potential obstruction. In the case of workload, WaaS platforms continuously receive many workflows' jobs from different users with their unique QoS requirements. The WaaS providers must be able to process these requests in a way that each of the requirements is fulfilled. A traditional WMS may process the workflows individually in a dedicated set of VMs as depicted in Fig. 1a. This approach, after all, is the simplest way to ensure the QoS fulfillment of each workflow.

In this dedicated service scenario, a WMS manages different types of tasks' execution by tailoring their specific software configurations and requirements to a VM image.

- *Muhammad H. Hilman is a PhD candidate at Cloud Computing and Distributed Systems (CLOUDS) Laboratory, The University of Melbourne, Australia.*
  E-mail: hilmanm@student.unimelb.edu.au
- *Maria A. Rodriguez is a Post-Doctoral Research Fellow at Cloud Computing and Distributed Systems (CLOUDS) Laboratory, The University of Melbourne, Australia.*
- *Rajkumar Buyya is a Director of Cloud Computing and Distributed Systems (CLOUDS) Laboratory, The University of Melbourne, Australia.*



(a) Scheduling workflows independently

(b) Scheduling workflows simultaneously

Fig. 1: Two approaches on scheduling multiple workflows

The VM containing this image then can be quickly deployed whenever a particular job of workflow is submitted. However, this model cannot easily be implemented in WaaS platforms where many users with different workflow applications are involved. We cannot naively simplify the assumption where every VM image can be shared between multiple users with different requirements. Various workflow applications may need different software configurations, which implies a possible dependency conflict if they are fitted within a VM image. This assumption also creates a more complex situation where, at any given time, a new workflow application type is needed to be deployed. This newly submitted job cannot reuse the already provisioned VMs as they may not contain its necessary software configurations. Furthermore, dedicating a single workflow execution in a set of VMs is considered inefficient as it leads to the inevitable schedule gaps from inter-dependent workflow tasks that result in the VMs being underutilized.

Therefore, adopting an appropriate resource-sharing policy and at the same time scheduling multiple workflows simultaneously, as shown in Fig. 1b, is considerably preferred for multi-tenant WaaS platforms. We argue that introducing this strategy creates a more efficient multi-tenant platform as it reduces the unnecessary overhead during the execution. The efficiency may be gained from (i) sharing the same workflow applications for different users by tailoring a specific software configuration in a container image instead of a VM image, so that (ii) sharing and reusing the already provisioned VMs between users to utilize the inevitable scheduling gaps from intra-workflow's dependency can be possible, and (iii) sharing local cached images and datasets within a VM that creates a locality, which eliminates the need for network transfer activity before the execution.

Based on these problems and requirements, we propose EBPSM, an **E**lastic **B**udget-constrained resource **P**rovisioning and **S**cheduling algorithm for **M**ultiple workflows designed for WaaS platforms, a budget-constrained version of EPSM algorithm [6]. Our proposed algorithm inherits the capabilities of the EPSM that can make fast decision to schedule the workflow tasks dynamically and empower the sharing of software configuration and reusing of already provisioned VMs between users using container technology. Our algorithm also considers inherent features of clouds that affect multiple workflows scheduling, such as performance variation of VMs [3] and VM provisioning delay [7] into a policy that determines the VMs provisioning decision. Furthermore, EBPSM implements an efficient budget distribution strategy that allows the algorithm to provision the fastest VMs possible to minimize the makespan and adopt the container images and datasets sharing policy to eliminate the need for network transfer between tasks' execution. Our extensive experiments show that EBPSM which adopts the resource-sharing policy can significantly reduce the overhead which implies the minimization of workflows' makespan.

The structure of the rest of the paper is organized as follows: Section 2 reviews works that are related to this proposal. Section 3 explains the problem formulation of multiple workflow scheduling in WaaS platforms including the assumption of application and resource models. The proposed algorithm is described in Section 4 followed by the performance evaluation in Section 5. Finally, the conclusion and future work are elaborated in Section 6.

2 Related Work

The majority of works in multiple workflows scheduling have pointed out the necessity of reusing already provisioned VMs instead of acquiring the new ones to reduce the idle gaps and increase system utilization. Examples of the works include the CWSA [8] algorithm that uses a depth-first search technique to find potential schedule gaps between tasks' execution. Another work is the CERSA [9] algorithm that dynamically adjusts the VM allocation for tasks in a reactive fashion whenever a new workflow job is submitted to the system. These works' idea to fill the schedule gaps between tasks' execution of a workflow to be utilized for scheduling tasks from another workflow is similar to our proposal. However, they merely assumed that different workflow applications could be deployed into any existing VMs available without considering possible complexity from software dependency conflicts. Our work differs in the way that we model the software configurations into a container image before deploying it to the VMs for execution.

The use of the container for deploying scientific workflows has been intensively researched. Examples include the work by Qasha et al. [10] that deployed a TOSCA-based workflow[1] using Docker[2] container on e-Science Central platform[3]. Although their work is done on a single host VM, the result shows promising future scientific workflows reproducibility that is made possible using container technology. A similar result is presented by Liu et al. [11] that convinces performance nativity and high flexibility of deploying scientific workflows using the Docker container. Finally, the adCFS [12] algorithm is designed to schedule containerized scientific workflows that encourage the CPU-sharing policy using a Markov-chain model to assign the appropriate CPU weight for containers. Those solutions are the early development of containerized scientific workflows on a single workflow execution. Their results show high feasibility to utilize container technology for efficiently bundling software configurations for workflows that are being proposed for WaaS platforms.

One of the challenges of executing scientific workflows in the clouds is related to the data-locality. Scientific workflows are known as data-intensive applications that involve a vast amount of data to be processed. Therefore, the communication overhead for transferring the data between tasks' execution may take a considerable amount of time that might impact the overall makespan. A work by Stavrinides and Karatza [13] shows that the use of distributed in-memory storage to

1. https://github.com/ditrit/workflows
2. https://www.docker.com/
3. https://www.esciencecentral.org/

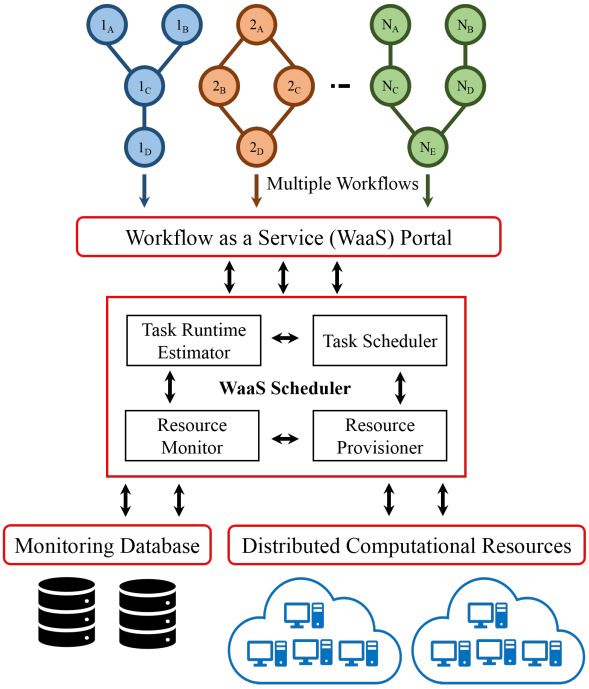


Fig. 2: Architecture of Workflow as a Service platform

store the datasets locally for tasks' execution can reduce the communication overhead. Our work is similar regarding the data-locality policy to minimize the data transfer between tasks' execution. However, we use the cached datasets in VMs local storage from tasks' execution to endorse the locality of data. We enhance this policy so that the algorithm can intelligently decide which task to be scheduled in particular VMs that can provide the minimum execution time given the available cached datasets.

Two conflicting QoS requirements in scheduling (e.g., time and cost) have been a significant concern of deploying scientific workflows in clouds. A more relaxed constraint to minimize the trade-off between these two requirements is shown in several works that consider scheduling the workflows within the deadline and budget constraints. They do not attempt to optimize one or both of the QoS requirements but instead maximizing the success rate of workflows execution within the constraints. Examples of these works include PAPS [14], MW-DBS [15], and MW-HBDCS [16] algorithms. Another similar work is the MQ-PAS [17] algorithm that emphasizes on increasing the providers' profit by exploiting the budget constraint as long as the deadline is not violated. Our work considers the user-defined budget constraint in the scheduling, but it differs in the way that the algorithm aims to optimize the overall makespan of workflows while meeting their budget.

Several works specifically focus on handling the real-time workload of workflows in WaaS platforms. This nature of workload raises the issue of uncertainties as the platforms have no knowledge of the arriving workflows. EDPRS [18] algorithm adopts the dynamic scheduling approach using event-driven and periodic rolling strategies to handle the uncertainties in real-time workloads. Another work, called ROSA [19] algorithm, controls the queuing jobs–which increase the uncertainties along with the performance variation of cloud resources–in the WaaS platforms to reduce the waiting time that can prohibit the uncertainties propagation. Both algorithms are designed to schedule the multiple workflows dynamically to minimize the operational cost while meeting the deadline. Our EBPSM can make a fast decision to schedule dynamically with the same purpose to handle the real-time workload and reduce the effect of uncertainties in WaaS environments. However, we differ in the way that our scheduling objectives are minimizing the workflows' makespan, while meeting the user-defined budget.

The majority of works in workflow scheduling that aim to minimize the makespan, while meeting the budget constraints, adopt a static scheduling approach. This approach finds a near-optimal solution of mapping the tasks to VMs–with various VM types configuration–to get a schedule plan before runtime. Examples of these works include CB-DT [20], MinMRW-MC [21], HEFT-Budg, and MinMin-Budg [22]. However, this static approach is considered inefficient for WaaS platforms as it might increase the waiting time of arriving workflows due to the intensive pre-processing computation time to generate a schedule plan. On the other hand, several works consider scheduling budget-constrained workflows dynamically driven by the available user-defined budget. BAT [23] algorithm distributes the budget of a particular workflow's job to their tasks by trickling down the available budget based on the depth of tasks from the entry task(s) in a workflow. Another work, called MSLBL [24], distributes workflow budget by calculating a proportion of the sub-budget to the task so that it can reduce the unused budget due to the inefficiency in budget distribution. However, those solutions are designed for a single cloud workflow scheduling scenario. To the best of our knowledge, none of the algorithms that aims to minimize the makespan, while meeting the budget for multiple workflows–that resembles the problem in WaaS platforms–has been proposed.

3 Problem Formulation

In this paper, we consider the resource provisioning and scheduling algorithm to schedule multiple workflows simultaneously. The workflows that are submitted for the execution may have the same workflow application type and require the same datasets, but they are not necessarily related to each other. Specifically, the workflows might belong to different users; therefore, they may have different QoS requirements. We assume that these workflows are being executed in a WaaS platform, and a proposed architecture for this particular system, based on a preliminary work [25], is depicted in Fig: 2.

3.1 Workflow as a Service (WaaS) Platform

The WaaS platforms can be placed either in the Platform as a Service (PaaS) or Software as a Service (SaaS) layer of the cloud service model. WaaS providers utilize distributed computational resources from Infrastructure as a Service (IaaS) services to serve the enormous need for computing power in the execution of scientific workflows. They provide an end-to-end service to scientists starting from the submission portal that allows users to define their requirements, automate their applications' installation and configuration based on the available or newly created templates, and stage input/output data from the system. Meanwhile, the rest of data and resource management including workflow scheduling and resource provisioning are transparent to the users. This platform is designed to process multiple workflows from different users as they arrive continuously. Therefore, WaaS platforms must handle a high level of complexity derived from their multi-tenancy nature, in contrast to a traditional WMS that is generally being deployed to manage a single workflow execution. In this work, we focus on the resource provisioner and task scheduler aspects that are designed to deal with the dynamic workload of workflows arriving into WaaS platforms.

3.2 Application and Resource Model

We consider a workload of workflows that are modelled as DAGs (Directed Acyclic Graphs). Furthermore, a workload $W = \{w_1, w_2, w_3, ..., w_n\}$ is composed of multiple workflows, where a workflow w consists of a number of tasks $T = \{t_1, t_2, t_3, ..., t_n\}$ and edges E. An edge $e_{ij}(t_i, t_j)$ represents the data dependency between task t_i and t_j where t_j, as a successor (i.e., child task), will only start the execution after t_i, as predecessor (i.e., parent task), is completed and output data $d_{t_i}^{out}$ from t_i is available on a VM allocated for t_j as input data $d_{t_j}^{in}$. We also assume that each workflow w is associated with a budget β_w that is defined as a soft constraint of cost representing users' willingness to pay for the execution of the workflows.

The task t is being executed within a container–that bundles software configurations for a particular workflow in a container image–which is then deployed on VMs. A container provisioning delays $prov_c$ is acknowledged to download the image, setup, and initiate the container on an active VM. We assume that only one container can run on top of a VM at a particular time. Therefore, the same host VM performance of CPU, memory, and bandwidth can be achieved in a particular container. Once the container is deployed, VM local storage maintains its image so it can be reused without the need to re-download the containers. We assume WaaS scheduler send custom signals by using commands in the container (e.g., Docker exec) to trigger tasks' execution within containers to avoid the necessity of container redeployment.

We consider a pay-as-you-go scheme in Iaas clouds, where VMs are provisioned on-demand and are priced per billing period bp (i.e., per-second, per-minute, per-hour). Hence, any partial use of the VM is being rounded up and charged based on the nearest bp. In this work, we assumed a fine-grained per-second bp as it is lately being adopted by the majority of IaaS clouds provider including Amazon EC2 [26], Google Cloud [27], and Azure [28]. We model a data center within a particular availability zone from a single IaaS cloud provider to reduce the network overhead and eliminate the cost associated with data transfer between zones. Our work considers a heterogeneous cloud environment model where VMs with different VM types $VMT = \{vmt_1, vmt_2, vmt_3, ..., vmt_n\}$ which have various processing power p_{vmt} and different cost per billing period c_{vmt} can be leased. We consider that all types of VM always have an adequate memory capacity to execute the various type of workflows' tasks. Finally, we can safely assume that the VM type with a higher p_{vmt} has more expensive c_{vmt} than the less powerful and slower ones.

Each VM has a bandwidth capacity b_{vmt} that is relatively the same between different VM types as they come from a single availability zone. We do not restrict the number of VMs to be provisioned during multiple workflows execution, but we also acknowledge the delay in acquiring VMs $prov_{vmt}$ from the IaaS provider. We assume that the VMs can be eliminated immediately from the WaaS platform without additional delay. Furthermore, we consider performance variation of VMs that might come from a virtualized backbone technology of clouds, geographical distribution, and multi-tenancy [3] and that price of VM advertised by IaaS provider is the highest CPU capacity can be achieved by the VMs. We do not assume another performance degradation of using the containerized environments as reported by Kozhirbayev and Sinnott [29]. Finally, we expect that VMs are only able to process a single task at a particular time for the simplicity of scheduling purpose.

A global storage system GS is modelled for data sharing between tasks (e.g., Amazon S3) with unlimited storage capacity. This global storage has reading rates GS_r and writing rates GS_w respectively. In this model, the tasks transfer their outputs from the VMs to the storage and retrieve their inputs from the same place before the execution. Therefore, the network overhead is inevitable. It is one of the uncertainties in clouds as the networks performance degradation that can be observed due to the number of traffics and virtualized backbone [30]. To reduce the need for accessing storage GS for retrieving the data, VMs local storage LS_{vmt} is also modelled to maintain d_t^{in} and d_t^{out} after particular tasks' execution using FIFO policy. It means the earliest stored data will be deleted whenever the capacity of LS_{vmt} cannot accommodate a new data needing to be cached. Furthermore, the time taken to retrieve the input data for a particular task's execution from global storage is shown in Eq. 1.

$$T_{vmt}^{d_t^{in}} = (d_t^{in}/b_{vmt}) + (d_t^{in}/GS_r) \tag{1}$$

It is worth noting that there is no need to transfer the input data from the global storage whenever it is available in the VM as a cached data from previous tasks execution. Similarly, the time needed for storing the output data from a VM to the global storage is depicted in Eq. 2.

$$T_{vmt}^{d_t^{out}} = (d_t^{out}/b_{vmt}) + (d_t^{out}/GS_w) \tag{2}$$

The runtime RT_{vmt}^t of a task t in a VM of type vmt is assumed as available to the scheduler as part of the scheduling process. The fact is that this runtime can be estimated using various techniques, including machine learning approaches [31], but we simplify the assumption where it is calculated based on the task's size S_t in Million of Instruction (MI) and the processing capacity p_{vmt} of the particular VM type in Million of Instruction Per Second (MIPS) as shown in Eq. 3.

$$RT_{vmt}^t = S_t/p_{vmt} \tag{3}$$

It needs to be noted that this RT_{vmt}^t value is only an estimate and the scheduler does not depend on it being 100% accurate. Furthermore, a maximum processing time of a task in a VM type PT_{vmt}^t consists of reading the input data required from the global storage, executing the task, and writing the output to the storage which are depicted in Eq. 4.

$$PT_{vmt}^t = T_{vmt}^{d_t^{in}} + RT_{vmt}^t + T_{vmt}^{d_t^{out}} \tag{4}$$

From the previous equations, we can calculate the maximum cost C_{vmt}^t of executing a task t on a particular vmt as shown in Eq. 5.

$$C_{vmt}^t = \lceil (prov_{vmt} + prov_c + PT_{vmt}^t)/bp \rceil * c_{vmt} \tag{5}$$

The budget-constrained scheduling problem that is being addressed in this paper concerns on how to minimize the total makespan of workflow while meeting the user-defined budget as depicted in Eq. 6.

$$min \sum_{n=1}^{T} PT_{vmt}^{t_n} \quad \text{while} \quad \sum_{n=1}^{T} C_{vmt}^{t_n} \leq \beta_w \tag{6}$$

Intuitively, the budget β_w will be spent efficiently on $\sum_{n=1}^{T} PT_{vmt}^{t_n}$ if the overhead components $prov_{vmt}$ and $prov_c$ that burden the cost of a task's execution can be discarded. This implies to the minimization of $\sum_{n=1}^{T} PT_{vmt}^{t_n}$ as the fastest VMs can be leased based on the available budget β_w. Another important note is that, further minimization can be achieved when the task t is allocated to the VM with d_t^{in} available, so the need for $T_{vmt}^{d_t^{in}}$ which is related to the network factor that becomes one of the sources of uncertainties can be eliminated.

Algorithm 1 Budget Distribution

```
1: procedure DISTRIBUTEBUDGET(β, T)
2:     S = tasks' estimated execution order
3:     for each task t ∈ T do
4:         allocateLevel(t, l)
5:         initiateBudget(0, t)
6:     for each level l do
7:         T_l = set of all tasks in level l
8:         sort T_l based on ascending Earliest Finish Time (EFT)
9:         put(T_l, S)
10:    while β > 0 do
11:        t = S.poll
12:        vmt = chosen VM type
13:        allocateBudget(C^t_vmt, t)
14:        β = β − C^t_vmt
```

However, it needs to be noted that there still exist some uncertainties in RT^t_{vmt} as the estimate of S_t is not entirely accurate, and the performance of VMs depicted in PC_{vmt} can be degraded at any time. Hence, there must be a control mechanism to ensure that these uncertainties do not propagate throughout the tasks' execution that will cause a violation of β_w. This control can be done by evaluating the real value of a task's execution cost C^t_{vmt} right after the task is completed. In this way, its successors can adjust their sub-budget allocation so that the total cost will not violate the budget β_w.

4 THE EBPSM ALGORITHM

In this paper, we propose a dynamic heuristic resource provisioning and scheduling algorithm designed for WaaS platforms with the objective of minimizing the makespan while meeting the budget. The algorithm is developed to efficiently schedule scientific workflows in multi-tenant platforms that deal with dynamic workload heterogeneity, the potential of resource inefficiency, and uncertainties of overheads along with performance variations. Overall, the algorithm enhances the reuse of software configurations, computational resources, and datasets to reduce the overheads that become one of the critical uncertainties in cloud environments. This policy is implemented in a resource-sharing model by utilizing container technology and VMs local storage in the decision-making process to schedule tasks and auto-scale the resources.

When a workflow is submitted to the WaaS portal, its owner may define the software requirements by creating a new container image or choosing the existing template. Whenever the user selects the existing images, the platform will identify the possibly relevant information that is maintained from the previous workflows' execution including analyzing previous actual runtime execution and its related datasets. Furthermore, the user then defines the budget β_w that is highly likely different from various users submitting the same type of workflow.

Next step, the workflow is forwarded to WaaS scheduler and is preprocessed to assign the sub-budget for each task based on the user-defined β_w. This sub-budget along with the possible sharing of software configurations and datasets will lead the decision made at the runtime to schedule a task onto either an existing VMs in the resource pool or a new VM provisioned from the IaaS cloud provider. The first phase of the budget distribution algorithm is to estimate the potential tasks' execution order within a workflow. The entry task(s) in the first level of a workflow are assumed to be scheduled first, followed by their successors in the next level until it reaches the exit task(s). In this case, we assign every task to a level within a workflow's structure based on the Deadline Top Level (DTL) approach as seen in Eq. 7

$$level(t) = \begin{cases} 0 & \text{if} \quad pred(t) = \emptyset \\ \max\limits_{p \in pred(t)} level(p) + 1 & \text{otherwise} \end{cases} \tag{7}$$

Furthermore, to determine the tasks' priority within a level, we sort them based on their Earliest Finish Time (EFT) in an ascending order as shown in Eq. 8

$$eft(t) = \begin{cases} PT^t_{vmt} & \text{if} \quad pred(t) = \emptyset \\ \max\limits_{p \in pred(t)} eft(p) + PT^t_{vmt} & \text{otherwise} \end{cases} \tag{8}$$

After estimating the possible tasks' execution order, the algorithm iterates over the tasks based on this order and distributes the budget to each task. This budget distribution algorithm estimates the sub-budget of a task based on the cost c_{vmt} of particular VM types. At first, the algorithm chooses VMs with the cheapest types for the task, and whenever there is any extra budget left after all tasks get their allocated sub-budgets, the algorithm uses this extra budget to upgrade the sub-budget allocation for the faster VM type starting from the earliest tasks in the order. This approach is called the Slowest First Task-based Budget Distribution (SFTD). The detailed strategy is depicted in Algorithm 1 and its preliminary study can be referred to the work by Hilman et al. [32].

Algorithm 2 Scheduling

```
1: procedure SCHEDULEQUEUEDTASKS(q)
2:     sort q by ascending Earliest Start Time (EST)
3:     while q is not empty do
4:         t = q.poll
5:         container = t.container
6:         vm = null
7:         if there are idle VMs then
8:             VM_idle = set of all idle VMs
9:             VM^input_idle = set of vm ∈ VM_idle that have
                   t's input data
10:            vm = vm ∈ VM^input_idle that can finish t within t.budget
                   with the fastest execution time
11:            if vm = null then
12:                VM_idle = VM_idle \ VM^input_idle
13:                VM^container_idle = set of vm ∈ VM_idle that have
                       container deployed
14:                vm = vm ∈ VM^container_idle that can finish t within
                       t.budget with the fastest execution time
15:                if vm = null then
16:                    VM_idle = VM_idle \ VM^container_idle
17:                    vm = vm ∈ VM_idle that can finish t within
                           t.budget with the fastest execution time
18:        else
19:            vmt = fastest VM type within t.budget
20:            vm = provisionVM(vmt)
21:        if vm != null then
22:            if vm.container != container then
23:                deployContainer(vm, container)
24:            scheduleTask(t, vm)
```

Once a workflow is preprocessed, and its budget is distributed, WaaS scheduler can begin the scheduling process, this step is illustrated in Algorithm 2. The primary objective of this scheduling algorithm is to reuse as much as possible the VMs that have datasets and containers–with software configurations available–in VMs local storage that may significantly reduce the overhead of retrieving the particular input data and container images from GS. This way, the algorithm avoids the provisioning of new VMs as much as possible, which reduces the VM provisioning delay and minimizes the network

Algorithm 3 Budget Update

```
1: procedure UPDATEBUDGET(T)
2:     t_f = completed task
3:     T_u = set of unscheduled t ∈ T
4:     β_u = total sum of t.budget, where t ∈ T_u
5:     sb = spare budget
6:     if C^{t_f}_{vmt} ≤ (t_f.budget + sb) then
7:         sb = (t_f.budget + sb) − C^{t_f}_{vmt}
8:         β_u = β_u + sb
9:     else
10:        debt = C^{t_f}_{vmt} − (t_f.budget + sb)
11:        β_u = β_u − debt
12:    DISTRIBUTEBUDGET(β_u, T_u)
```

Algorithm 4 Resource Provisioning

```
1: procedure MANAGERESOURCE
2:     VM_idle = all leased VMs that are currently idle
3:     threshold_idle = idle time threshold
4:     for each vm_idle ∈ VM_idle do
5:         t_idle = idle time of vm
6:         if t_idle ≥ threshold_idle then
7:             terminate vm_idle
```

communication overhead from data transfer and downloading container images that contribute to the uncertainties in the WaaS environments.

Furthermore, WaaS scheduler releases all the entry tasks (i.e., tasks with no parents) of multiple workflows into the queue. As the tasks' execution proceeds, the child tasks–which parents are completed–become ready for execution and are released into the scheduling queue. As a result, at any point in time, the queue contains all the tasks from different workflows submitted to the WaaS platform that is ready for execution. The queue is periodically being updated whenever one of the two events triggered the scheduling cycle. Those are the arrival of a new workflow's job and the completion of a task's execution.

In every scheduling cycle, each task in the scheduling queue is processed as follows. The first step is to find a set of VM_{idle} on the system that can finish the task's execution with the fastest time within its budget. The algorithm estimates the execution time by not only calculating PT^{t}_{vmt} but also considering possible overhead $prov_c$ caused by the need for initiating $container$ in case the available VM_{idle} does not have suitable $container$ deployed.

At first attempt, VM_{idle} with input datasets VM^{input}_{idle} are preferred. The VM^{input}_{idle} that have the datasets available in their local storage must also cache the $container$ image from the previous execution. In this way, two uncertain factors $T^{d^{in}_t}_{vmt}$ for retrieving datasets from GS and $prov_c$ for initiating the $container$ are eliminated. In this case, the sub-budget for this particular task can be spent well on leasing the fastest VM type to minimize its execution time. This scenario is always preferred since the retrieval of datasets from GS and downloading the $container$ images through networks has become a well-known overhead that poses a significant uncertainty as its performance also may be degraded over time [30].

If any VM has not been found in the previous set of VM^{input}_{idle}, the algorithm finds a set of VM_{idle} that have $container$ deployed. This set of $VM^{container}_{idle}$ may not have the input datasets available as it may have been cleared from VMs local storage due to its FIFO limited lifetime to cache data from the previous execution. If the set still does not contain the preferred VM, any VM from remaining set of VM_{idle} is chosen. In the last scenario, the overhead of provisioning delay $prov_{vm}$ can be eliminated. It is still better than having to acquire a new VM. Whenever a suitable VM in the resource pool is found, the task then is immediately scheduled on it. If the existing VMs are not available, then the algorithm provisions a new VM with the fastest VM type that can finish the task within its sub-budget. This approach is the last option to schedule a task on the platform.

For better adaptation to uncertainties that come from the performance variation and the unexpected overhead delays during execution, there is a control mechanism within the algorithm to dynamically adjust sub-budget allocation whenever a task is finished being executed. This mechanism defines a spare budget variable that stores the residual sub-budget calculated from the actual cost execution. Whenever a task is finished, the algorithm calculates the actual cost of execution using the Eq. 5 and redistributes the leftover sub-budget to the unscheduled tasks. If an actual cost is exceeded the allocated sub-budget, the shortfall will be taken from the sub-budget of unscheduled tasks. Therefore, the budget update (i.e., budget redistribution) will take place every time a task is finished. In this way, the uncertainties (e.g., performance variation, overhead delays) that befall to a particular task is not propagating throughout the rest of the following tasks. The details of this process are depicted in Algorithm 3.

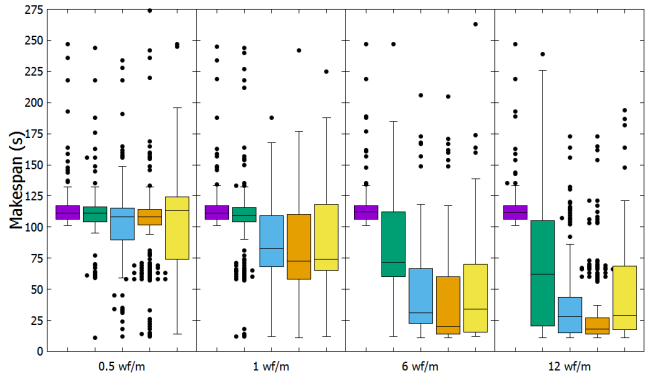
Regarding the resource provisioning strategy, the algorithm encourages a minimum number of VMs usage by reusing as much as possible existing VMs. The new VMs are only acquired whenever the idle VMs are not available due to the high density of the workload to be processed. In this way, the VM provisioning delays overhead can be reduced. As for the deprovisioning strategy, all of the running VMs are monitored every $prov_{int}$ and all VMs that have been idle for more than $threshold_{idle}$ time are terminated as seen in Algorithm 4. The decision to keep or terminate a particular VM is being taken considerably as the cached data-locality within its local storage is one of the valued factors that impact the whole performance of this algorithm. Therefore, the $prov_{int}$ and $threshold_{idle}$ are configurable parameters that can lead to a trade-off between performance in term of resource utilization and makespan along with the VMs local storage caching policy.

5 Performance Evaluation

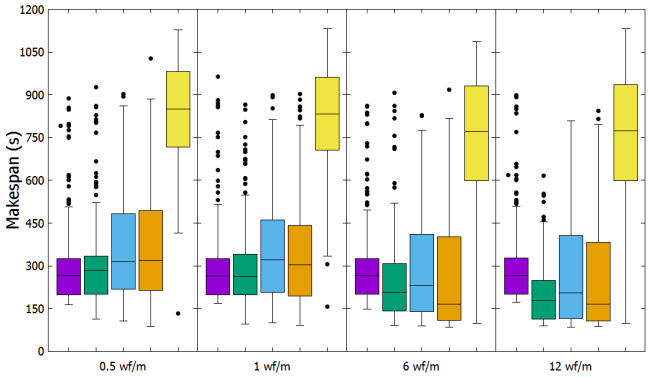
To evaluate our proposal, we used five synthetic workflows from the profiling of well-known workflows [33] from various scientific area generated using the WorkflowGenerator tool [34]. The Montage workflow is an astronomy application used to produce a sky mosaics image from several different angle sky observation images. Most of the Montage tasks are considered I/O intensive while involving less CPU processing. The second workflow is an astrophysics application called Inspiral–part of the LIGO project–that is used to analyze the data from gravitational waves detection. This workflow consists of CPU-intensive tasks and requires a high memory capacity. The third workflow is a Bioinformatics application used to encode sRNA genes called SIPHT. Its tasks are similar to the Inspiral LIGO, which have high CPU requirements

Table. 1: Configuration of VM types used in evaluation

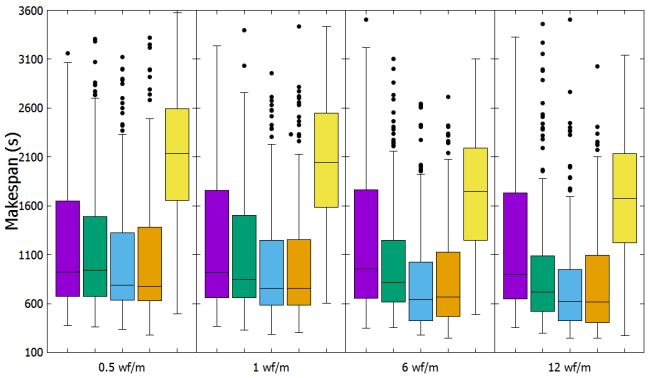
Name	CPU (MIPS)	Storage (GB)	Price per second
Small	2	20	¢1
Medium	4	40	¢2
Large	8	80	¢4
XLarge	16	160	¢8



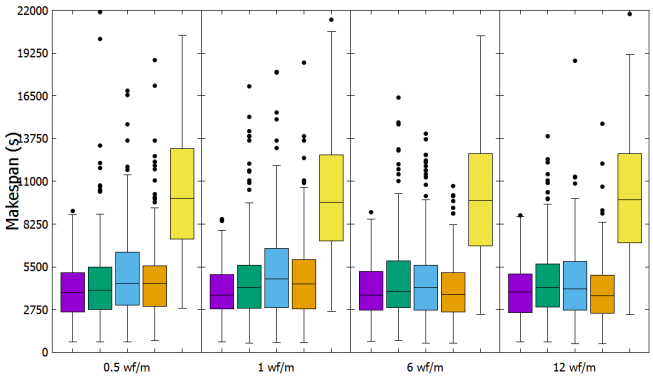
(a) Makespan of Montage



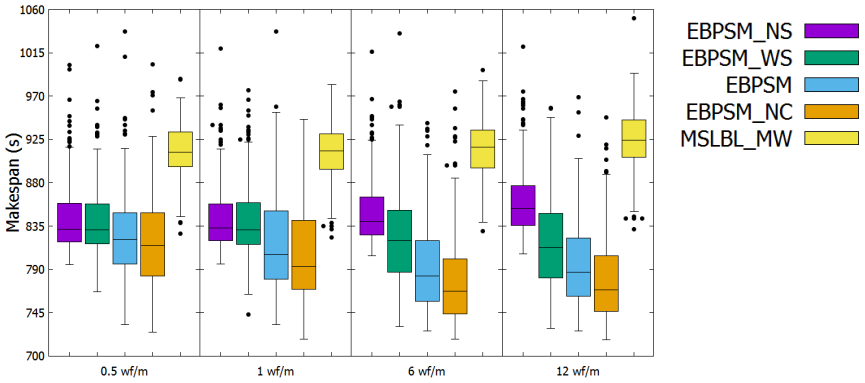
(b) Makespan of LIGO



(c) Makespan of SIPHT



(d) Makespan of Epigenomics



(e) Makespan of CyberShake

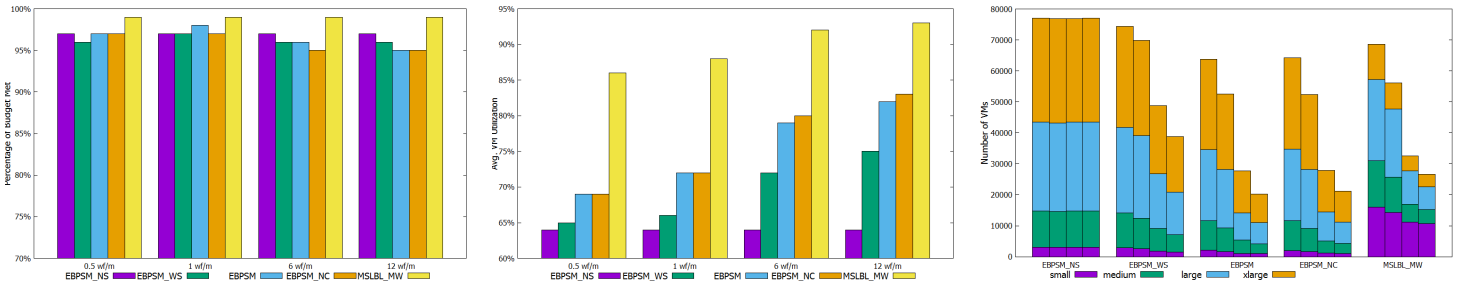
Fig. 3: Makespan of workflows on various workloads with different arrival rate

with relatively low I/O utilization. The next workflow is Epigenomics, another Bioinformatics application with CPU-intensive tasks for executing operations that are related to the genome-sequencing research. Finally, we include the CyberShake workflow that generates synthetic seismograms to differentiate various earthquakes hazards. This earth-science workflow is data-intensive with large CPU and memory requirements.

The experiments were conducted with various workloads composed of a combination of workflows mentioned above in three different sizes: around 50 tasks (small), 100 tasks (medium), and 1000 tasks (large). Each workload contains a different number and various type of workflow that was randomly selected based on the uniform distribution, and the arrival rate was modeled based on the Poisson distribution. Every workflow in a workload was assigned a budget that is always assumed sufficient. Budget insufficiency can be managed by rejecting the job from the platform or renegotiating the budget with the users. This budget was randomly generated based on the uniform distribution from a range of minimum and maximum cost of executing the workflow. The minimum cost was estimated from simulating the execution of all its tasks in sequential order on a single slowest VM type, while the maximum cost was estimated based on the execution of each task on a dedicated fastest VM. In this experiment, we used the runtime generated from the WorkflowGenerator for the size measurement of the task.

We extended CloudSim [35] to support the simulation of WaaS platforms. Using CloudSim, we modeled a single IaaS cloud provider that offers a data center within a single availability zone with four VM types that are shown in Table 1. These four VM types configurations are the simplification of the compute optimized (c4) instance types offered by the Amazon EC2 where the CPU capacity has a linear relationship with its respective price. We modeled the per-second billing period for leasing the VMs, and for all VM types, we set the provisioning delay to 45 seconds based on the latest study by Ulrich et al. [36]. On the other side, the container provisioning delay was set to 10 seconds based on the average container size of 600 MB, a bandwidth 500 Mbps, and a 0.4 seconds delay in container initialization [37].

The CPU and network performance variation were mod-



(a) Percentage of budget met (b) Avg. VM utilization (c) Number of VMs

Fig. 4: Percentage of Budget Met and VM usage on various workloads with different arrival rate

eled based on the findings by Leitner and Cito [3]. The CPU performance was degraded by a maximum 24% of its published capacity based on a normal distribution with a 12% mean and a 10% standard deviation. Furthermore, the bandwidth available for a data transfer was potentially degraded by at most 19% based on a normal distribution with a 9.5% mean and a 5% standard deviation. In this experiment, as mentioned earlier, each VM was modeled to maintain an LS_{vmt} that stores the cached data produced during the execution based on FIFO policy. The design for a more intelligent strategy to maintain or terminate the lifetime of cached datasets within an LS_{vmt} is left for future work.

To create a baseline for EBPSM, we extended the MSLBL [24] algorithm for multiple workflows scheduling (MSLBL_MW). MSLBL was designed for a single workflow execution, so we added a function to handle multiple workflows by creating a pool of arriving workflows where the algorithm then dispatched the ready tasks from all workflows for scheduling. Furthermore, MSLBL assumed that a set of computational resources are available in a fixed quantity all over the scheduling time. Therefore, to cope up with a dynamic environment in WaaS platforms, we added a simple dynamic provisioner for MSLBL_MW that provisions a new VM whenever there is no existing VMs available. This newly provisioned VM is selected based on the fastest VM type that can be afforded by the sub-budget of a particular task being scheduled. This dynamic provisioner also automatically terminates any idle VMs to ensure the optimal utilization of the system. Finally, for MSLBL_MW, we assumed that every VM can contain software configurations for every workflow application type and can be shared between any users in WaaS platforms.

To demonstrate the benefits of our resource-sharing policy, we implemented three additional versions of EBPSM, which are EBPSM_NS, EBPSM_WS, and EBPSM_NC. EBPSM_NS does not implement any sharing policy; it is a version of EBPSM that executes each workflow submitted into the WaaS platform in dedicated service. EBPSM_WS tailors the software configuration of workflow applications in a VM image instead of containers. Therefore, the algorithm allows only tasks from the same workflow application type (e.g., SIPHT with 50 tasks and SIPHT with 100 tasks) that are able to share the provisioned VMs during the execution. Meanwhile, EBPSM_NC ignores the use of containers to store the configuration template and assumes that each VM can be shared between many users with different requirements. This version was a direct comparable case for MSLBL_MW. Finally, the $threshold_{idle}$ for EBPSM_WS, EBPSM, and EBPSM_NC was set to 5 seconds. It means the vm_{idle} is not immediately terminated whenever it goes idle to accommodate the further utilization of the cached data sets within the VM.

5.1 Sharing Policy Evaluation

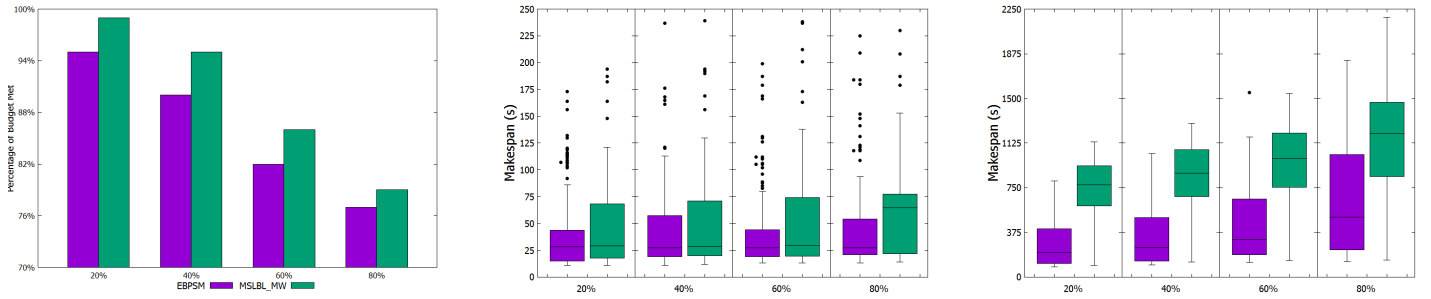
The purpose of this experiment is to evaluate the effectiveness of our proposed resource-sharing policy regarding its capability to minimize the workflows' makespan while meeting the soft limit budget. We evaluated EBPSM and its variants against MSBLBL_MW under four workloads with the different arrival rate of 0.5, 1, 6, and 12 workflows per minute. Each workload consists of 1000 workflows with approximately 170 thousand tasks with various size (e.g., small, medium, large) and different workflow's type generated randomly based on a uniform distribution. The arrival rate for these four workloads represents the density of workflows' arrival in the WaaS platform. The arrival of 0.5 workflows per minute represents the less occupied platform, while the arrival of 12 workflows per minute models the busiest system in handling the workflows.

Figures 3a, 3b, 3c, 3d, and 3e depict the makespan achieved for Montage, LIGO, SIPHT, Epigenomics, and CyberShake workflow respectively. EBPSM_NS that does not adopt any sharing policy shows almost no difference in the algorithm's performance across different arrival rate. This version of the algorithm serves each of the workflows in dedicated and isolated resources. Therefore, it can maintain a similar performance for all four workloads. However, on the other hand, EBPSM_NC shows the lowest percentage of average VM utilization due to this non-sharing policy as seen in Figure 4b.

In contrast to EBPSM_NS, the other three versions (e.g., EBPSM_WS, EBPSM, EBPSM_NC) exhibit the performance improvement along with the increasing density of the workloads. This further makespan minimization is the result of (i) the elimination of data transfer overhead between tasks' execution and (ii) the utilization of inevitable scheduling gaps between tasks' execution. In the (i) case, we can observe that the improvement is relatively not significant for Epigenomics workflows where the CPU processing takes the biggest portion of the execution time instead of I/O and data movement. On the other hand, the highest improvement can be observed from the data-intensive workflows such as Montage and CyberShake applications. Furthermore, the superiority of EBPSM and EBPSM_NC over EBPSM_WS both in makespan and average VM utilization shows a valid argument for the (ii) case. From the following result, we can conclude that the

Table. 2: Cost/budget ratio for budget violated cases for EBPSM on various workloads with different arrival rate

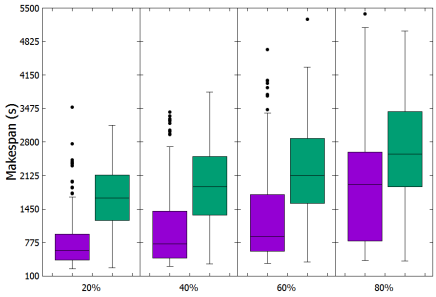
Percentile	0.5 wf/m	1 wf/m	6 wf/m	12 wf/m
10th	1.005	1.004	1.017	1.005
30th	1.017	1.018	1.032	1.023
50th	1.026	1.030	1.051	1.052
70th	1.046	1.053	1.065	1.069
90th	1.072	1.083	1.121	1.107



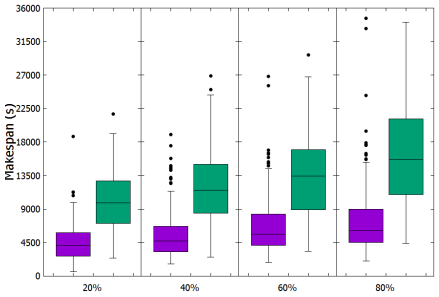
(a) Percentage of budget met

(b) Makespan of Montage

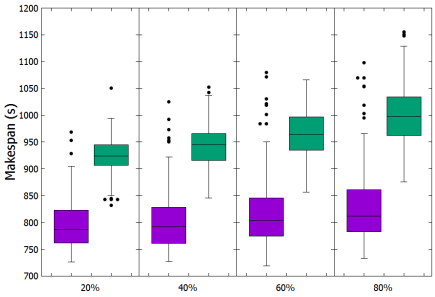
(c) Makespan of LIGO



(d) Makespan of SIPHT



(e) Makespan of Epigenomics



(f) Makespan of CyberShake

Fig. 5: Percentage of budget met and makespan of workflows on various CPU performance degradation

utilization of idle gaps between users from different workflow type can further minimize the makespan.

The assumption that every VM can be shared between any users in the platform explains the lower makespan and the higher utilization produced by EBPSM_NC compared to EBPSM. Container usage generates additional initialization delays that affect EBPSM performance. However, the difference between them is marginal, and EBPSM still exhibits its superiority to the other versions.

We observe that in four out of five workflows cases, all versions of EBPSM overthrow MSLBL_MW regarding the makespan achievement. This result comes from the different strategy of both algorithms in distributing the budget and avoiding the violation which implies the type of VMs they provisioned. EBPSM prioritizes the budget allocation to the earlier tasks and leases the fastest VM type as much as possible based on the idea that the following children tasks can utilize these already provisioned VMs while maintaining the capability of meeting the budget by updating the allocation based on the actual tasks' execution.

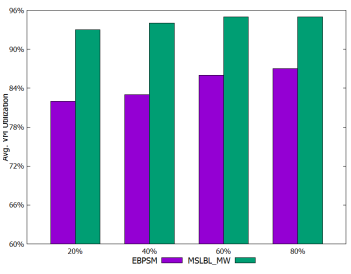
On the other hand, MSLBL_MW allocates the budget based on the budget level factor which creates a safety net by provisioning the VM that costs somewhere between the minimum and maximum execution cost of a particular task. In this way, MSLBL_MW reduces the possibility of budget violation at the budget distribution phase. These two different approaches result in the different number of VM types used during the execution as can be seen from Figure 4c. MSLBL_MW leases a lower number of faster VM types compared to EBPSM for all cases. The only case where the performance of MSLBL_MW is relatively equal to EBPSM is in Montage workflow where the tasks are relatively short in CPU processing time, while the significant portion of their execution time takes place in the data movement. In this Montage case, the decision to lease which kind of VM type does not significantly affect the total makespan.

If we zoom-in Figure 4a, we can see that all of the algorithms are able to achieve at least 95% of the budget meeting for all cases. The margin between MSLBL_MW and four versions of EBPSM was never wider than 4%. MSLBL_MW is superior to EBPSM regarding the average VM utilization. This result is caused by the difference in the VM deprovisioning policy. MSLBL_MW eliminates any VMs as soon as they become idle, while EBPSM delays the elimination in the hope of further utilization and cached data for the following tasks on that particular idle VM. In this case, the configurable settings of $threshold_{idle}$ value may affect the VM utilization. However, from these two different approaches, a significant margin of makespan between MSLBL_MW and EBPSM can be observed in most cases.

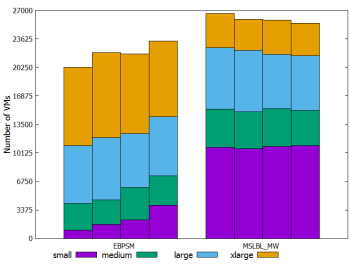
We captured the cases where EBPSM failed to meet the budget and show the result in Table 2 to see the actual EBPSM performance. From the table, we can see that the cost/budget ratio for 90% of the EBPSM budget violation cases are lower than 1.12. It means that the additional cost produced from these violations are never higher than 12%. This percentage is relatively marginal and may be caused by the extreme case of CPU and network performance degradation. It is not possible to completely eliminate the negative impact of these uncertainties in such a dynamic environment.

5.2 Performance Degradation Sensitivity

Adapting to performance variability is an essential feature for scheduler in multi-tenant dynamic environments. This ability ensures the platform to quickly recover from unexpected events that may occur at any given time without giving a chance to produce a snowball impact to the later execution.

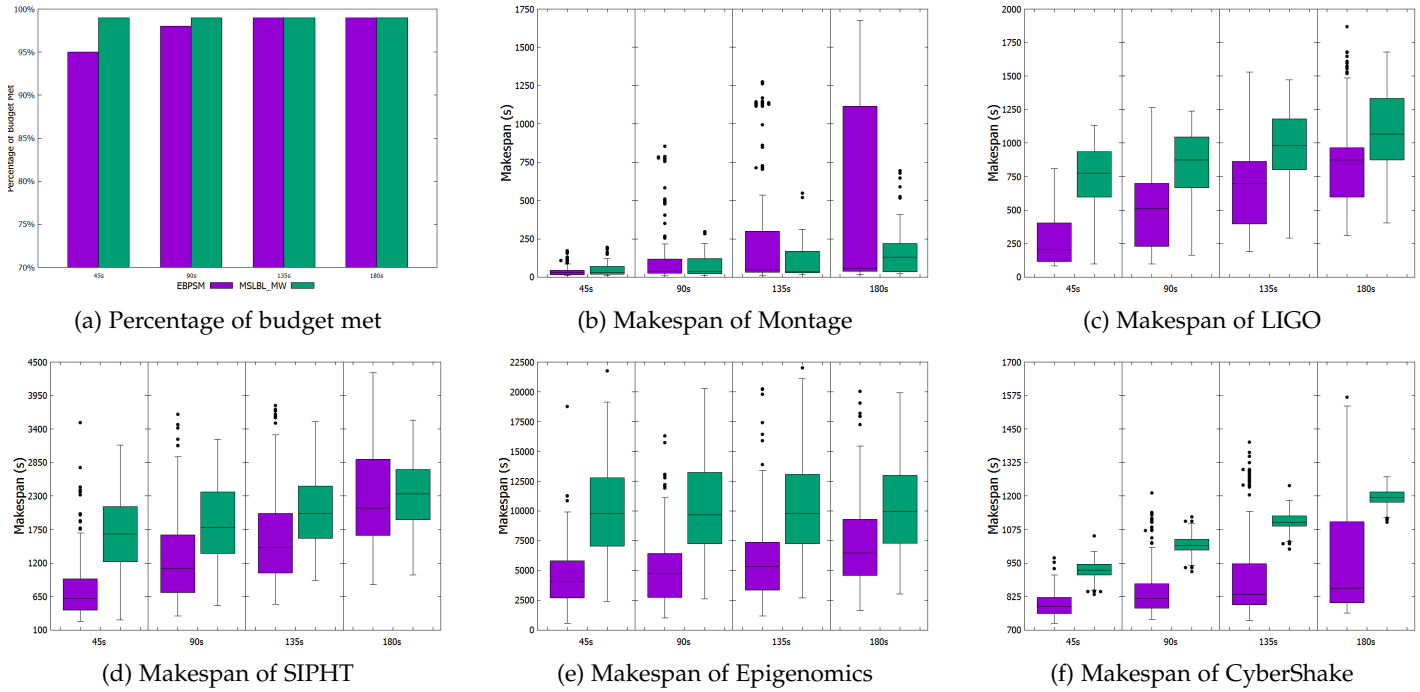


(a) Avg. VM Utilization



(b) Number of VMs

Fig. 6: VM usage on various CPU performance degradation



(a) Percentage of budget met (b) Makespan of Montage (c) Makespan of LIGO

(d) Makespan of SIPHT (e) Makespan of Epigenomics (f) Makespan of CyberShake

Fig. 7: Percentage of budget met and makespan of workflows on various VM provisioning delay

The sensitivity of the algorithms toward CPU performance degradation was evaluated by analyzing the percentage of budget met, makespan, average VM utilization, and the number of VMs used on various performance degradation scenarios. We model the CPU performance degradation using a normal distribution with 1% variance and different average and maximum values. The average value is defined as half of the maximum of the CPU performance degradation which ranges from 20% to 80%.

All algorithms are significantly affected by CPU performance degradation as their percentage of budget met decreases along with the increased maximum degradation value. However, MSLBL_MW suffers the most as its performance margin with EBPSM is getting smaller as seen in Figure 5a. This suffering also can be observed from Figures 5b, 5c, 5d, 5e, 5f. The increasing makespan as a response to the performance degradation for EBPSM was relatively lower than its effect on MSLBL_MW. EBPSM can survive better than MSLBL_MW because of its capability to adapt the changes by evaluating a particular task's execution right after it was finished.

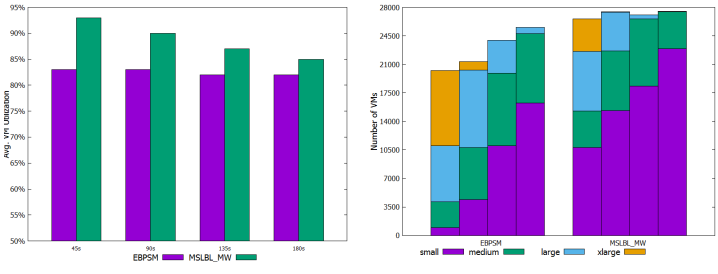
Meanwhile, MSLBL_MW only relies on the spare budget from its safety net of budget allocation that limits the number of faster VM type at the budget distribution phase. When the maximum degradation value increases, there is no extra budget left from this safety net. Hence, MSLBL_MW performance of meeting the budget drops faster than EBPSM. This reason is also in line with the average VM utilization results where MSLBL_MW cannot increase their VMs utilization as it reaches the top limit of its capabilities as seen in Figure 6a.

Another perspective can be observed from the number of VMs used by each algorithm as depicted in Figure 6b. EBPSM recovers from the CPU performance degradation by continually increasing the number of slower VM type, while maintaining as much as possible the faster VM type. This adaption is being made through the budget update process after a task finished being executed. Therefore, the earlier tasks are still able to be scheduled onto faster VMs, while the children tasks in the tail of the workflow are inevitably allocated to the slower ones due to the budget left from the recovering process. Different behavior is observed from MSLBL_MW usage of VMs. We cannot see any significant effort to recover from the number of VM type used. We argue that MSLBL_MW way to survive the CPU performance degradation is by highly relying upon the safety net of budget allocation from the budget distribution phase.

5.3 VM Provisioning Delay Sensitivity

A large volume of arriving workflows results in a vast number of tasks in the scheduling queue to be processed at a given time. However hard it is the effort to minimize the number of VMs usage by sharing and re-using already provisioned VMs, provisioning a massive VMs is inevitable in WaaS platform. This provisioning will become a problem if the scheduler is not designed to handle the uncertainties from delays in acquiring the VMs. We study the sensitivity of the algorithms under four different VM provisioning delays ranging from 45 to 180 seconds. We analyze this sensitivity from the percentage of budget met, makespan, average VM utilization, and the number of VMs used in various VM provisioning delays scenarios.

An interesting point can be observed from Figure 7a that shows the percentage of budget met with the algorithms. All of the algorithms are able to perform better facing increasing VM provisioning delays. However, it comes with a greater trade-off of the workflows' makespan as seen in Figures 7b, 7c, 7d, 7e, 7f especially for the workflow's application that highly depends on CPU processing. We can observe that the spectrum



(a) Avg. VM Utilization (b) Number of VMs

Fig. 8: VM usage on various CPU provisioning delay

for Montage and CyberShake makespan in EBPSM results for 180 seconds delays are quite wide although its overall performance is still superior to MSLBL_MW. In contrast to this situation, the makespan spectrum for MSLBL_MW for those two workflows is very narrow.

The number of shared VMs and how varied the VM type used during the execution may cause the reason for the different makespan spectrum generated in higher VM provisioning delays scenarios. MSLBL_MW adopts a simple sharing policy. This algorithm does not care whether a particular VM contains a cached data for a future scheduled task or not; it just merely scheduled the task onto any idle VM available. In this way, the algorithm can reduce the variation of the VM type used during the execution.

When EBPSM leases faster VMs for the earlier tasks, VM provisioning delays increase the cost for those VMS. Therefore, the algorithm cannot afford those already provisioned VMs if the budget left for the following tasks is not enough to provision the same VM type. In this case, EBPSM must provision a new VM with a slower type. Hence, a workflow with a tight budget will suffer unexpected longer makespan as the algorithm tries to allocate a faster VM type for the earlier tasks but turns out that the budget left is not sufficient and automatically recovers during the budget update by provisioning a new slower VM type for that particular task. Figure 8b confirms this situation where EBPSM uses a broader variety of VM type compared to MSLBL_MW. However, in general, the average VM utilization does not affect by the VM provisioning delays as the VM initiating process is not counted through the total utilization, but still, this delay is being charged. So that is why the delay profoundly affects the total budget. Finally, this analysis does not hinder the fact that EBPSM in general, is still superior to MSLBL_MW in terms of makespan for most of the cases.

6 Conclusions and Future Work

The growing popularity of scientific workflows deployment in clouds drives the research on multi-tenant platforms that provides utility-like services for executing workflows. As well as any other multi-tenant platform, this workflow as a service (WaaS) platform faces several challenges that may impede the system effectiveness and efficiency. These challenges include the handling of a continuously arriving workload of workflows, the potential of system inefficiency from inevitable idle time slots from workflows' tasks dependency execution, and the uncertainties from computational resources performances that may impose significant overhead delays.

WaaS platforms extend the capabilities of a traditional Workflow Management System (MWS) to provide a more comprehensive service for larger users. In a traditional WMS, a single workflow job is processed in dedicated service to ensure its Quality of Service (QoS) requirements. However, this approach may not be able to cope up with the WaaS environments where a significant deficiency may arise from its conventional way of tailoring workflows' software configurations into a VM image, intra-dependent workflows' tasks inevitable schedule gaps, and possible overhead delays from workflow pre-processing and data movement handling.

To achieve more efficient multi-tenant WaaS platforms, we proposed a novel resource-sharing policy that (i) utilizes container technology to wrap software configurations required by particular workflows to (ii) enable the idle slots VMs sharing between users and (iii) further makespan minimization by sharing the datasets and container images cached within VMs local storage. We implemented this policy on EBPSM, a dynamic heuristic scheduling algorithms designed for multiple workflows scheduling in WaaS. Our extensive experiments show that the sharing scenarios overthrow a traditional dedicated approach in handling workflows' jobs. Furthermore, our proposed algorithm is able to surpass a modified state-of-the-art budget-constrained dynamic scheduling algorithm in terms of minimizing makespan of workflows and meeting the soft limit budget defined by WaaS platforms' users.

There are several aspects of our experiments that need to be further investigated. First, the budget distribution phase takes a vital role in budget-constrained scheduling. The decision to either allocate more budget for the earlier tasks so they can lease faster computational resources or maintain a safety net allocation to ensure the budget compliance must be carefully taken into account. A trade-off between having a faster execution time and meeting the allocated budget is always inevitable. In this way, defining the nature of execution including strictness of the budget constraints can help to design an appropriate configuration between two approaches.

Second, the resource provisioning (and deprovisioning) strategy must consider the *quid pro quo* between having a higher system utilization (i.e., lower idle VM times) and an optimal data sharing and movement which utilizes the VM local storage. We observed that delaying a particular VM termination in the deprovisioning phase may improve the performance when the cached data stored within the VM is intelligently considered. In this work, we did not consider task failure either caused by the software or the infrastructure (i.e., container, VMs). Incorporating a fault-tolerant strategy into both EPSM and EBPSM algorithms is necessary to address the task failure that highly likely impacting the WaaS platforms performance.

Finally, further investigation on how multiple container instances can be run and scheduled on top of a single VM is another to do list. The delay in initiating a container image has reduced our algorithm's performance. There must be a way to counterbalance this issue by exploiting the swarming nature of container to gain benefits from this predetermined condition to enhance the WaaS platforms efficiency further.

Acknowledgments

This research is partially supported by LPDP (Indonesia Endowment Fund for Education) and ARC (Australia Research Council) research grant.

References

[1] E. Deelman, T. Peterka, I. Altintas, C. D. Carothers, K. K. van Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, and J. Vetter, "The Future of Scientific Workflows," *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 159–175, 2018.

[2] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The Cost of Doing Science on The Cloud: The Montage Example," in *Proceedings of The ACM/IEEE Conference on Supercomputing*, Nov 2008, pp. 1–12.

[3] P. Leitner and J. Cito, "Patterns in the Chaos: A Study of Performance Variation and Predictability in Public IaaS Clouds," *ACM Transaction on Internet Technology*, vol. 16, no. 3, pp. 15:1–15:23, Apr. 2016.

[4] E. N. Alkhanak, S. P. Lee, R. Rezaei, and R. M. Parizi, "Cost Optimization Approaches for Scientific Workflow Scheduling in Cloud and Grid Computing: A Review, Classifications, and Open Issues," *Journal of Systems and Software*, vol. 113, no. Supplement C, pp. 1–26, 2016.

[5] M. A. Rodriguez and R. Buyya, "A Taxonomy and Survey on Scheduling Algorithms for Scientific Workflows in IaaS Cloud Computing Environments," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 8, pp. e4041–n/a, 2017.

[6] ——, "Scheduling Dynamic Workloads in Multi-tenant Scientific Workflow as a Service Platforms," *Future Generation Computer Systems*, vol. 79, no. Part 2, pp. 739–750, 2018.
[7] M. Jones, B. Arcand, B. Bergeron, D. Bestor, C. Byun, L. Milechin, V. Gadepally, M. Hubbell, J. Kepner, P. Michaleas, J. Mullen, A. Prout, T. Rosa, S. Samsi, C. Yee, and A. Reuther, "Scalability of VM Provisioning Systems," in *Proceedings of The IEEE High Performance Extreme Computing Conference*, Sept 2016, pp. 1–5.
[8] B. P. Rimal and M. Maier, "Workflow Scheduling in Multi-tenant Cloud Computing Environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 290–304, 2017.
[9] H. Chen, J. Zhu, G. Wu, and L. Huo, "Cost-efficient Reactive Scheduling for Real-time Workflows in Clouds," *The Journal of Supercomputing*, Sep 2018.
[10] R. Qasha, J. Cala, and P. Watson, "Dynamic Deployment of Scientific Workflows in The Cloud Using Container Virtualization," in *Proceedings of The IEEE International Conference on Cloud Computing Technology and Science*, 2016, pp. 269–276.
[11] K. Liu, K. Aida, S. Yokoyama, and Y. Masatani, "Flexible Container-based Computing Platform on Cloud for Scientific Workflows," in *Proceedings of The International Conference on Cloud Computing Research and Innovations*, 2016, pp. 56–63.
[12] E. J. Alzahrani, Z. Tari, Y. C. Lee, D. Alsadie, and A. Y. Zomaya, "adCFS: Adaptive Completely Fair Scheduling Policy for Containerised Workflows Systems," in *Proceedings of The 16th IEEE International Symposium on Network Computing and Applications*, 2017, pp. 1–8.
[13] G. L. Stavrinides, F. R. Duro, H. D. Karatza, J. G. Blas, and J. Carretero, "Different Aspects of Workflow Scheduling in Large-scale Distributed Systems," *Simulation Modelling Practice and Theory*, vol. 70, no. Supplement C, pp. 120–134, 2017.
[14] J. Shi, J. Luo, F. Dong, J. Zhang, and J. Zhang, "Elastic Resource Provisioning for Scientific Workflow Scheduling in Cloud Under Budget and Deadline Constraints," *Cluster Computing*, vol. 19, no. 1, pp. 167–182, Mar 2016.
[15] H. Arabnejad and J. G. Barbosa, "Maximizing The Completion Rate of Concurrent Scientific Applications Under Time and Budget Constraints," *Journal of Computational Science*, vol. 23, no. Supplement C, pp. 120–129, 2017.
[16] N. Zhou, F. Li, K. Xu, and D. Qi, "Concurrent Workflow Budget- and Deadline-constrained Scheduling in Heterogeneous Distributed Environments," *Soft Computing*, June 2018.
[17] H. Arabnejad and J. G. Barbosa, "Multi-QoS Constrained and Profit-aware Scheduling Approach for Concurrent Workflows on Heterogeneous Systems," *Future Generation Computer Systems*, vol. 68, no. Supplement C, pp. 211–221, 2017.
[18] H. Chen, J. Zhu, Z. Zhang, M. Ma, and X. Shen, "Real-time Workflows Oriented Online Scheduling in Uncertain Cloud Environment," *The Journal of Supercomputing*, vol. 73, no. 11, pp. 4906–4922, 2017.
[19] H. Chen, X. Zhu, G. Liu, and W. Pedrycz, "Uncertainty-Aware Online Scheduling for Real-Time Workflows in Cloud Service Environment," *IEEE Transactions on Services Computing*, pp. 1–1, 2018.
[20] R. Ghafouri, A. Movaghar, and M. Mohsenzadeh, "A Budget Constrained Scheduling Algorithm for Executing Workflow Application in Infrastructure as a Service Clouds," *Peer-to-Peer Networking and Applications*, pp. 1–28, June 2018.
[21] H. Cao and C. Q. Wu, "Performance Optimization of Budget-constrained MapReduce Workflows in Multi-clouds," in *Proceedings of The 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2018, pp. 243–252.
[22] Y. Caniou, E. Caron, A. K. W. Chang, and Y. Robert, "Budget-Aware Scheduling Algorithms for Scientific Workflows with Stochastic Task Weights on Heterogeneous IaaS Cloud Platforms," in *Proceedings of The IEEE International Parallel and Distributed Processing Symposium Workshops*, May 2018, pp. 15–26.
[23] V. Arabnejad, K. Bubendorfer, and B. Ng, "A Budget-Aware Algorithm for Scheduling Scientific Workflows in Cloud," in *Proceedings of The 18th IEEE International Conference on High Performance Computing and Communications; 14th IEEE International Conference on Smart City; 2nd IEEE International Conference on Data Science and Systems*, Dec 2016, pp. 1188–1195.
[24] W. Chen, G. Xie, R. Li, Y. Bai, C. Fan, and K. Li, "Efficient Task Scheduling for Budget Constrained Parallel Applications on Heterogeneous Cloud Computing Systems," *Future Generation Computer Systems*, vol. 74, pp. 1–11, 2017.
[25] M. H. Hilman, M. A. Rodríguez, and R. Buyya, "Task Runtime Prediction in Scientific Workflows Using an Online Incremental Learning Approach," in *Proceedings of The 11th IEEE/ACM International Conference on Utility and Cloud Computing*, Dec 2018, pp. 93–102.
[26] "Announcing Amazon EC2 per second billing," Oct 2017. [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2017/10/announcing-amazon-ec2-per-second-billing/
[27] P. Nash, "Extending per second billing in Google Cloud — Google Cloud Blog," Sep 2017. [Online]. Available: https://cloud.google.com/blog/products/gcp/extending-per-second-billing-in-google
[28] "Windows Virtual Machines Pricing." [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/virtual-machines/windows/
[29] Z. Kozhirbayev and R. O. Sinnott, "A Performance Comparison of Container-based Technologies for The Cloud," *Future Generation Computer Systems*, vol. 68, no. Supplement C, pp. 175–182, 2017.
[30] R. Shea, F. Wang, H. Wang, and J. Liu, "A Deep Investigation Into Network Performance in Virtual Machine Based Cloud Environments," in *Proceeding of The IEEE Conference on Computer Communications*, April 2014, pp. 1285–1293.
[31] T. P. Pham, J. J. Durillo, and T. Fahringer, "Predicting Workflow Task Execution Time in The Cloud Using A Two-Stage Machine Learning Approach," *IEEE Transactions on Cloud Computing*, vol. PP, no. 99, pp. 1–1, 2017.
[32] M. H. Hilman, M. A. Rodríguez, and R. Buyya, "Task-Based Budget Distribution Strategies for Scientific Workflows with Coarse-Grained Billing Periods in IaaS Clouds," in *Proceedings of The 13th IEEE International Conference on e-Science*, 2017, pp. 128–137.
[33] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and Profiling Scientific Workflows," *Future Generation Computer Systems*, vol. 29, no. 3, pp. 682–692, 2013.
[34] "Pegasus Workflow Management System." [Online]. Available: https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator
[35] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: a Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011.
[36] M. Ullrich, J. Lässig, J. Sun, M. Gaedke, and K. Aida, "A Benchmark Model for The Creation of Compute Instance Performance Footprints," in *Internet and Distributed Computing Systems*. Springer International Publishing, 2018, pp. 221–234.
[37] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya, "ContainerCloudSim: An Environment for Modeling and Simulation of Containers in Cloud Data Centers," *Software: Practice and Experience*, vol. 47, no. 4, pp. 505–521, 2017.

Muhammad H. Hilman is a Ph.D. candidate in the Cloud Computing and Distributed Systems (CLOUDS) Laboratory in the School of Computing and Information Systems, The University of Melbourne, Australia. His research interests include resource management and scheduling in clouds, parallel computing, and high-performance computing.

Maria A. Rodriguez is a Post-Doctoral Research Fellow in the Cloud Computing and Distributed Systems (CLOUDS) Laboratory in the School of Computing and Information Systems, The University of Melbourne, Australia. Her research interests include resource management and scheduling in clouds and scientific computing.

Rajkumar Buyya is a Professor of Computer Science and Software Engineering and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He is also the founding CEO of Manjrasoft, a spin-off company of the University, commercializing its innovation in Cloud Computing. He has authored over 400 publications and four textbooks. He is one of the highly cited authors in computer science and software engineering worldwide.