Journal of Combinatorial Optimization (2025) 49:34
https://doi.org/10.1007/s10878-025-01265-8

Energy-efficient real-time multi-workflow scheduling in container-based cloud

Zaixing Sun[1] · Hejiao Huang[1] · Zhikai Li[1] · Chonglin Gu[1]

Accepted: 23 January 2025 / Published online: 22 February 2025
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2025

Abstract
Cloud computing has a powerful capability to handle a large number of tasks. However, this capability comes with significant energy requirements. It is critical to overcome the challenge of minimizing energy consumption within cloud service platforms without compromising service quality. In this paper, we propose a heuristic energy-saving scheduling algorithm, called Real-time Multi-workflow Energy-efficient Scheduling (RMES), which aims to minimize the total energy consumption in a container-based cloud. RMES schedules tasks in the most parallelized way to improve the resource utilization of the running machines in the cluster, thus reducing the time of the global process and saving energy. This paper also considers the affinity constraints between containers and machines, and RMES has the ability to satisfy the resource quantity and performance requirements of containers during the scheduling process. We introduce a re-scheduling mechanism that automatically adjusts the scheduling decisions of remaining tasks to account for the dynamic system states over time. The results show that RMES outperforms other scheduling algorithms in energy consumption and success rate. In the higher arrival rate scenario, the proposed algorithm saves energy consumption by more than 19.42%. The RMES approach can enhance the reliability and efficiency of scheduling systems.

Keywords Multi-workflow scheduling · Real time · Container cloud · Energy minimization

1 Introduction

Cloud Service Platforms (CSPs) are a potent tool for handling large-scale scientific applications that are frequently submitted by users in real-time in the form of workflows. These workflows are comprised of tasks with varying resource requirements, each with a unique structure and its own Quality of Service (QoS) requirements,

✉ Chonglin Gu
guchonglin@hit.edu.cn

[1] School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, China

including specific deadlines. CSPs provide consumers with the ability to access computing and storage resources on-demand, facilitating the execution of their workflows in a more streamlined and effective manner (Al-Dulaimy et al. 2022).

As a new virtualization technology, the container is more suitable for a real-time multi-workflow scheduling scenario than the traditional virtual machines (VMs) (Merkel 2014). Containers are lightweight, with lower overhead in terms of memory and CPU usage, allowing for faster deployment and more efficient resource utilization. Additionally, containers share the host system's operating system, which reduces redundant resource allocation, leading to lower energy consumption. These characteristics make containers more suitable for real-time, energy-constrained scenarios compared to VMs, where resource efficiency and rapid task execution are critical. Specifying container affinities enables more effective management and allocation of resources within a container cloud, ensuring that the diverse needs of multi-workflow environments are adequately addressed.

With the rapid growth of cloud computing, the energy consumption of data centers will rise significantly, from 200 terawatt hours (TWh) in 2016 to a staggering 2967 TWh by 2030 Katal et al. (2023). This surge in energy consumption not only increases data center operating costs, but also exacerbates environmental damage. With these challenges, it is crucial to minimize cloud service center energy consumption while ensuring timely task completion. Nowadays, there are many studies on energy-efficient task scheduling in data centers. Hussain et al. (2021) and Sun et al. (2023) proposed energy-saving task scheduling algorithms on cloud. However, the complex affinity relationship between containers and physical machines introduces more restrictions on energy-efficient scheduling decisions, so these methods cannot be directly applied to container cloud scenarios.

Recent research has largely focused on the development of scheduling algorithms to reduce energy consumption in data centers. In particular, studies have introduced energy-efficient scheduling algorithms tailored for traditional cloud environments. However, the intricate affinity relationships between containers and physical machines introduce additional complexities to making energy-efficient scheduling decisions in the container cloud environment, rendering these traditional methods less effective. Ding et al. (2020) proposed an energy-saving scheduling algorithm based on Q-Learning for independent task scheduling in cloud. However, they does not consider the important aspect of task dependencies, a key factor in workflow scheduling. Particle swarm optimization algorithm is used to solve the energy-saving scheduling problem for executing micro-service applications in container-based cloud (Adhikari and Srirama 2019). Despite their potential, these algorithms require significant computational overhead for decision making, which may not be compatible with the requirements of real-time scheduling scenarios where rapid response is essential.

To address the above limitations, we present the Real-time Multi-workflow Energy-efficient Scheduling (RMES) algorithm in a container-based cloud to minimize the energy consumption of the cluster while executing as many workflows as possible on time. In a cluster, executing tasks in parallel can improve resource utilization, reduce global process time, and reduce the number of machines running. Therefore, RMES uses a rule-based scheduling strategy to execute tasks as parallel as possible to save energy. RMES can also dynamically adjust the historical scheduling decisions

to adapt to the continuously changing system state in real-time scenarios. In addition, RMES takes into account the affinity between the container and the physical machine in scheduling, which makes it more suitable than other algorithms for solving the energy-saving scheduling problem under the container cloud platform. The main contributions of this work can be summarized as follows:

- Taking into account the affinity constraints between tasks and machines, we establish a real-time multi-workflow scheduling model on heterogeneous clusters.
- We present RMES, a heuristic scheduling algorithm that reduces cluster energy consumption by parallel execution of tasks.
- RMES efficiently minimizes energy consumption in the cluster by proactively identifying and shutting down underutilized physical machines.
- We evaluated the scheduling algorithm's performance using real-world workflows. Compared to existing algorithms, RMES reduces CSP energy consumption while satisfying container and physical machine affinity constraints.

2 Related work

The workflow scheduling problem is classified as an NP-Complete problem (Sun et al. 2023). At present, there are many researches in energy-saving scheduling of container cloud. There are four ways to solve these problems: accurate method, heuristics, meta-heuristics and machine learning methods (Ahmad et al. 2021).

Mathematical modeling is mainly to model problems as common problems (Kaur et al. 2020) (integer programming problem, mixed integer programming problem), and then solve it by using accurate algorithms (branch and bound method, cutting plane method, etc.). These accurate methods often fail to provide solutions solution in a reasonable time, as the problem's size increases.

Heuristics approximate the optimal scheduling of tasks on a cluster using predefined scheduling rules. These approaches are significantly faster at making scheduling decisions than other methods because they are based on rules designed from experience. As a result, heuristic techniques are well-suited for real-time scheduling scenarios. Havet et al. (2017) reduced energy consumption in clusters by using generational garbage collection principles, while striving to meet deadline constraints for all requests as closely as possible. Hu et al. (2022) proposed adjustments to task scheduling decisions that balance energy consumption with task execution times in real-time environments. However, these methods have limitations, such as not fully accounting for real-time constraints or overlooking specific resource constraints unique to container cloud environments, including container affinity.

The meta-heuristic method, which primarily uses genetic algorithms, simulated annealing, ant colony algorithms, and other techniques (Tan et al. 2019; Gan et al. 2010; Azad and Navimipour 2017), iterates to discover the best solution in the problem space. In Tan et al. (2019), the energy-saving scheduling problem is transformed into a two-level (task-VM, VM-PM) bin-packing problem and solved by genetic programming. However, this model can not be applied to the problem of using containers as resource allocation units. In Gan et al. (2010), the authors use simulated annealing algorithm

to deal with the parameters dimensionless in order to meet the different QoS needs of users. In Shi et al. (2018), the authors propose a two-stage multi-type particle swarm optimization approach, which reduces energy consumption during cluster operation. Although these meta-heuristic algorithms can solve large-scale scheduling problems, they focus on energy optimization in offline scenarios. In the real-time scenario, the cluster needs to make scheduling decisions for the incoming tasks in time. Iterative search requires a lot of calculation, and may not generate scheduling decisions quickly (Hu et al. 2022).

The machine learning method mainly uses reinforcement learning method and learning the feedback of historical scheduling decisions to constantly adjust the scheduling strategy of the cluster. In Ding et al. (2020), a Q-learning based algorithm is proposed to reduces energy consumption of task execution in container cloud. However, the algorithm does not consider the interdependence between tasks. In Cheng et al. (2018), a deep Q-learning-based two stage Resource Provisioning-Task Scheduling processor learns scheduling strategies according to historical electricity prices and requests to save energy. The authors of Kang et al. (2021) suggest an adaptive deep reinforcement learning framework for task scheduling that improves resource utilization to save energy consumption. The disadvantage of this machine learning method is that historical scheduling data need to be collected in advance for model training. Due to various constraints in the scheduling problem, many scenarios can not be well modeled using machine learning method.

Since the heuristic method can generate scheduling decisions in a short time, and some characteristics of container cloud can be used when setting heuristic rules, we utilize the heuristic method to solve the problem proposed in this paper.

3 Problem formulation

3.1 Workflow and service instance model

Efficiently managing workflow applications from users in real-time is critical in cloud computing. These applications consist of various requests, denoted as $\mathcal{W} = \{w_1, w_2, ..., w_m\}$, each of which can be divided into several tasks that can be executed either sequentially or in parallel. To represent a request, a directed acyclic graph (DAG) can be used. A DAG is a way of showing how different tasks are related to each other and the order in which they need to be executed to complete the request.

A request is modeled as $w_m = \{w_m^{at}, w_m^{trt}, w_m^d, G_m\} \in \mathcal{W}$, where w_m^{at} is the arrival time, w_m^{trt} is the maximum time a user can tolerate for a request to be executed, w_m^d is the deadline, and G_m is DAG structure of w_m. The deadline, w_m^d, can be calculated as $w_m^d = w_m^{at} + w_m^{trt}$. $G_m = (\mathcal{T}_m, E_m)$, where $\mathcal{T}_m$ is the set of tasks and E_m represents the dependency between tasks. Let $\{t_{m1}, t_{m2}, \cdots, t_{m|\mathcal{T}_m|}\}$ be the set of tasks $\mathcal{T}_m$ $t_{mi}(0 < i \leq |\mathcal{T}_m|)$ is the ith task of w_m. $|\mathcal{T}_m|$ is the total number of tasks in w_m. Furthermore, E_m is a 0-1 matrix of $T_m \times T_m$. $e_{u,v}^m = 1$ implies a data dependence between t_{mu} and t_{mv}, where t_{mu} is the immediate predecessor of t_{mv}. We model t_{mi} as $t_{mi} = \{t_{mi}^I, t_{mi}^{image}, t_{mi}^{type}, t_{mi^d}\}$, where:

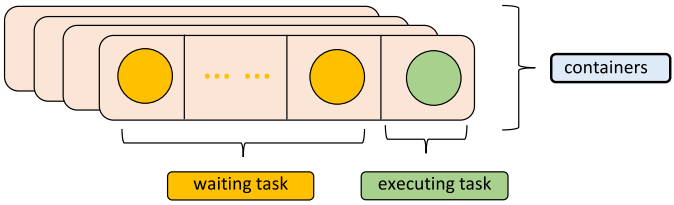

Fig. 1 Task queue in container

- t_{mi}^{I}: the number of instructions contained in t_{mi}.
- t_{mi}^{image}: the image of container executing t_{mi}.
- t_{mi}^{type}: the type of t_{mi}.
- t_{mi}^{d}: the sub-deadline of t_{mi}.

When a task needs to be executed, it is packaged with all the necessary software and configuration settings into a container image. This container image is then deployed on a physical machine (PM) for execution. The container image comprises all the software dependencies and libraries required to run the task, making it a self-contained and portable unit. The containerization approach enables the application to run in any computing environment without worrying about the underlying infrastructure. It also provides better resource utilization, enhances security, and enables faster deployment and scaling. Let $\mathcal{C}$ be the set of all containers in the cluster. We define container $c_j = \{c_j^{type}, c_j^{cpu}, c_j^{mem}, c_j^{cache}, c_j^{taints}\} \in \mathcal{C}$ $(0 < j \leq |\mathcal{C}|)$. The following are the properties of each container in the cluster:

- c_j^{type}: the type of task that container c_j runs.
- c_j^{cpu}: the number of CPU cores required by container c_j.
- c_j^{mem}: the memory size required by container c_j.
- c_j^{cache}: the set of tasks that have been assigned to container c_j.
- c_j^{taints}: taint nodes, i.e., the set of PMs that cannot run container c_j.

As illustrated in Fig. 1, each container is programmed to handle one task at a time, and these tasks in the cache are divided into two states: waiting and executing. The term "taint node" is used to identify a specific group of physical machines (PMs) that are not suitable to run a particular container due to factors determined by the user. These factors may include the absence of specific hardware components necessary for the container or the PM's hardware failing to meet the minimum performance standards required for running the container. There is a direct correspondence between containers and tasks, where each container is associated with a single task. Consequently, a container of a particular type is restricted to running tasks of the same type, and similarly, tasks of a particular type are restricted to running on containers of the corresponding type. Containers are instantiated from images associated with their respective tasks, ensuring that the container environment is configured precisely to meet the requirements of the task.

Let $\mathcal{P} = \{p_1, p_2, \ldots, p_{|\mathcal{P}|}\}$ be the set of PMs provided by CSP. The kth PM $p_k = \{p_k^{t\_cpu}, p_k^{t\_mem}, p_k^{\bar{t},u\_cpu}, p_k^{\bar{t},u\_mem}, p_k^{c}, p_k^{e\_total}, p_k^{e\_base}, p_k^{ips}\}$, where:

- $p_k^{t\_cpu}$: the number of CPU cores of the p_k.
- $p_k^{t\_mem}$: the memory resources of the p_k.
- $p_k^{\bar{t},u\_cpu}$: the number of CPU cores used in p_k at $\bar{t}$ moment.
- $p_k^{\bar{t},u\_mem}$: the memory resources used in p_k at $\bar{t}$ moment.
- p_k^{c}: the set of containers that are currently running in p_k.
- $p_k^{e\_total}$: the power consumption of the p_k under full load.
- $p_k^{e\_base}$: the power consumption of the physical machine when it is idle.
- p_k^{ips}: the number of instructions that a single core of p_k can process per second.

$p_k^{\bar{t},u\_cpu}$, $p_k^{\bar{t},u\_mem}$ can be calculate by Eqs. ((1), 2).

$$p_k^{\bar{t},u\_cpu} = \sum_{c_j \in \mathcal{C}} x_{j,k}^{\bar{t}} c_j^{cpu}, \tag{1}$$

$$p_k^{\bar{t},u\_mem} = \sum_{c_j \in \mathcal{C}} x_{j,k}^{\bar{t}} c_j^{mem}, \tag{2}$$

where $x_{j,k}^{\bar{t}} \in \{0, 1\}$. $x_{j,k}^{\bar{t}} = 1$ indicates that the container c_j is running in p_k at $\bar{t}$ moment. According to the widely-used model in cloud computing energy analysis (Ding et al. 2020; Beloglazov et al. 2011), we model the relationship between the power of the p_k and the CPU utilization at $\bar{t}$ moment by Eq. (3), where $\mathbf{P}_k^{\bar{t}}$ represent the power of p_k at $\bar{t}$ moment.

$$\mathbf{P}_k^{\bar{t}} = \begin{cases} 0, & p_k \, is \, off, \\ \frac{p_k^{\bar{t},u\_cpu}}{p_k^{t\_cpu}}(p_k^{e\_total} - p_k^{e\_base}) + p_k^{e\_base}, & p_k \, is \, on. \end{cases} \tag{3}$$

3.2 Cloud workflow scheduling model

This paper aims to minimize the total energy consumption for workflow execution, which can be expressed by Eq. (4). T denotes the total execution time of the system.

$$\mathbb{E} = \sum_{\bar{t}=0}^{T} \sum_{k=1}^{|\mathcal{P}|} \mathbf{P}_k^{\bar{t}}, \tag{4}$$

Task Dependency Constraints. All tasks can only be executed when their predecessors have finished or when there are no predecessors due to dependencies between tasks. t_{mu}^{EST} is the earliest start time of t_{mu}. t_{mv}^{FT} is the finish time of t_{mv}. $Pred(t_{mu}) = \{t_{mv} | e_{v,u}^m = 1, \forall \, t_{mv} \in \mathcal{T}_m\}$ is the set of immediate predecessors of t_{mu}.

$$\max_{t_{mv} \in Pred(t_{mu})} t_{mv}^{FT} \leq t_{mu}^{EST}, \tag{5}$$

Deadline constraint. We need to ensure that workflows can be finished on time when the scheduler makes decisions.

$$\max_{t_{mv} \in \mathcal{T}_m} t_{mv}^{FT} \leq w_m^d. \tag{6}$$

Task and Container placement constraint. A task can be deployed on a single container of the same type. The binary variable $y_{mi,j}^{\bar{t}}$ is either 0 or 1 in Eq. (7). If it is 1, it means that the task t_{mi} is deployed on the container c_j. If it is 0, it means that t_{mi} is not deployed on c_j. Resource level constraints must be met when deploying containers to PMs. Equation (9) and Eq. (10) ensure that the resources used by the container deployed on the PM do not exceed the total resources of the PM. Equation (11) ensures that a container can only be deployed on one PM and that this PM cannot be a taint node of c_j.

$$\sum_{j=1}^{|\mathcal{C}|} y_{mi,j}^{\bar{t}} = 1. \tag{7}$$

$$t_{mi}^{type} = c_j^{type}, y_{mi,j}^{\bar{t}} = 1. \tag{8}$$

$$\sum_{c_j \in \mathcal{C}} x_{j,k}^{\bar{t}} c_j^{cpu} \leq p_k^{t\_cpu}, \tag{9}$$

$$\sum_{c_j \in \mathcal{C}} x_{j,k}^{\bar{t}} c_j^{men} \leq p_k^{t\_mem}, \tag{10}$$

$$\sum_{j=1}^{|\mathcal{C}|} x_{j,k}^{\bar{t}} = 1, \ p_k \notin c_j^{taints}. \tag{11}$$

Based on the above discussion, we can formulate the constrained optimization problem as follows:

$$\textbf{Minimize } \mathbb{E}, \tag{12a}$$

$$\textbf{Subject to:} \tag{12b}$$

$$Eq.(5), \quad t_{mu} \in \mathcal{T}_m, w_m \in \mathcal{W}, \tag{12c}$$

$$Eq.(6), \quad w_m \in \mathcal{W}, \tag{12d}$$

$$Eq.(7), \quad t_{mi} \in \mathcal{T}_m, w_m \in \mathcal{W}, \tag{12e}$$

$$Eq.(8), \quad t_{mi} \in \mathcal{T}_m, w_m \in \mathcal{W}, c_j \in \mathcal{C}, \tag{12f}$$

$$Eqs.(9), (10), (11), \quad p_k \in \mathcal{P} \tag{12g}$$

$$x_{j,k}^{\bar{t}} \in \{0, 1\}, \quad p_k \in \mathcal{P} \tag{12h}$$

$$y_{mi,j}^{\bar{t}} \in \{0, 1\}, \quad t_{mi} \in \mathcal{T}_m, w_m \in \mathcal{W}, c_j \in \mathcal{C} \tag{12i}$$

where the decision variables are $x_{j,k}^{\bar{t}}$ and $y_{mi,j}^{\bar{t}}$.

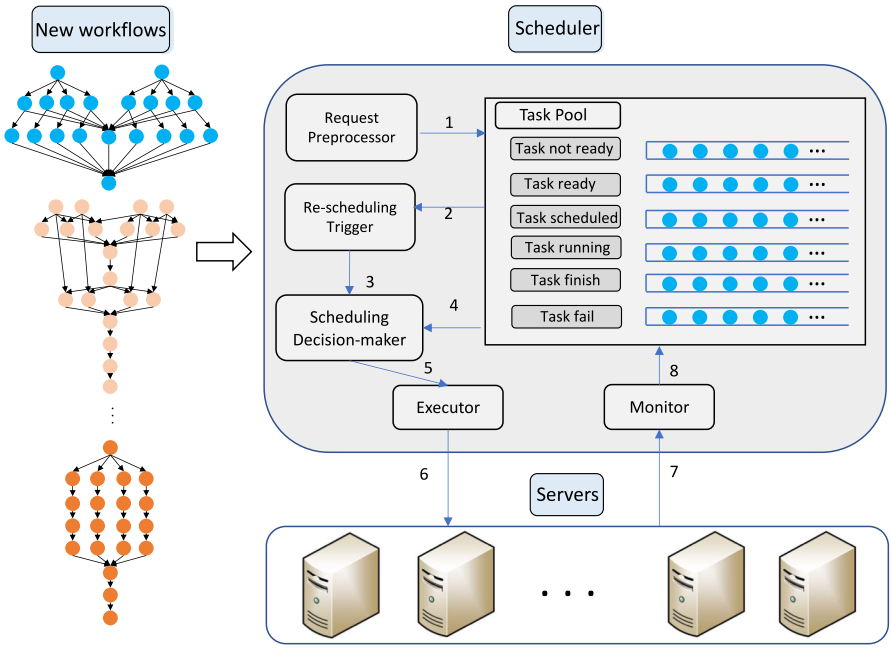

Fig. 2 Schedule System

4 Real-time multi-workflow energy-efficient scheduling algorithm

This section describes the RMES algorithm in detail. First, we introduce the structure of the scheduling system. Then, we introduce the algorithms of each phase of RMES.

4.1 Scheduling architecture

Fig. 2 shows the real-time workflow scheduling architecture. The system architecture is segmented into three primary components: end users, instance clusters, and the scheduler. This structure enables users to submit workflow requests at any given time. The scheduler plays a crucial role in allocating user-submitted workflows to instance clusters for optimal system operation.

A scheduler comprises various components, such as the request preprocessor, task pool, rescheduling trigger, scheduling decision maker, executor and monitor. When a workflow request is received by the scheduler, the request preprocessor takes the lead role by dividing the workflow into tasks. For each task, the preprocessor also sets a deadline and priority. The following are the steps involved in task scheduling and execution: Step 1: Tasks are added to the task pool. Step 2: The rescheduling trigger receives the status of the task pool and determines whether general scheduling or rescheduling is necessary. Step 3: Based on the trigger's decision, the scheduling decision-maker generates the scheduling plan. Step 4: The scheduler extracts the necessary task information from the task pool. Step 5: The scheduling decisions are sent

to the executor for task execution. Step 6: During the execution period, the cluster's running state is adjusted based on the received instructions. Step 7: The monitor constantly monitors the cluster's status. Step 8: The monitor updates the task pool with the latest information.

4.2 Request preprocessor

This particular component has two functions. Firstly, it sets a sub-deadline for tasks that are included in a user's request. Secondly, it sorts these tasks based on their priority. The sub-deadline for each task is determined by its topological level in the request. In this context, the topological level of a task (t_{mi}) is calculated using a specific formula (Eq.(13)). For each level, we take the task with the most instructions as the critical task of that level. Then, we determine the execution time of that level ($level_l^{time}$) by considering the duration of the critical task on the fastest machine.

$$Lev(t_{mi}) = \begin{cases} 1, & Pred(t_{mi}) = \emptyset \\ \max\limits_{t_{mj} \in Pred(t_{mi})} Lev(t_{mj}) + 1, & other. \end{cases} \tag{13}$$

$$Level_l^t = \{t_{mi} | Lev(t_{mi}) = l)\} \tag{14}$$

$$\hat{t}_l = \arg\max_{t_{mi} \in Level_l^t} (t_{mi}^I) \tag{15}$$

$$\hat{p} = \arg\max_{p_k \in \mathcal{P}} (p_k^{ips}) \tag{16}$$

$$level_l^{time} = c_j^{load} + \frac{\hat{t}_l}{\hat{p}^{ips} \cdot c_j^{cpu}}, c_j^{type} = \hat{t}_l^{type}, \tag{17}$$

where $\hat{t}_l$ is the task with highest number of instructions in level l, $\hat{p}$ is the fastest single core PM, and c_j^{load} is the duration required for preparing the container for handling tasks after deployment. We can model the estimated processing time for w_m as Eq. (18). Then we can set the sub-deadline of t_{mi} as Eq. (19).

$$w_m^{et} = \sum_{l=1}^{L} level_l^{time}, \tag{18}$$

$$t_{mi}^d = \frac{level_l^{time}}{w_m^{et}} \cdot w_m^{trt} + w_m^{at}, \ Lev(t_{mi}), \tag{19}$$

where L is the maximum level in w_m.

The prioritization of tasks is determined by a ranking method that considers the number of subsequent tasks associated with each task, including immediate and mediate successors. We define $d(t_{mu})$ as the set of tasks dependent on t_{mu}, including all tasks directly or indirectly dependent on it.

$$d(t_{mu}) = (\bigcup_{t_{mv} \in Sub(t_{mu})} d(t_{mv})) \cup t_{mu}, \tag{20}$$

$$Rank(t_{mu}) = |d(t_{mu})|, \tag{21}$$

$Sub(t_{mv}) = \{t_{mu} | e^m_{u,v} = 1, \forall\, t_{mu} \in \mathcal{T}_m\}$ represents the set of immediate successors of t_{mv}. A task's rank is a crucial factor in determining its priority level for scheduling. Tasks with a higher rank are given priority over others at the same level in the topology. It is essential to assign the correct rank to a task to ensure efficient and timely completion of the workflow.

4.3 Task pool

The task pool is a comprehensive record of the current status of tasks within the system. These tasks can be categorized into various types within the cluster, such as:

- *Task not ready* Tasks cannot be initiated until their dependencies have been fulfilled.
- *Task ready* These tasks have completed their preceding tasks and are waiting for scheduling by the system based on priority and resource availability. They are currently pending execution on a node.
- *Task scheduled* These tasks have been assigned to a container, but have not yet commenced execution.
- *Task running* These tasks are currently being processed by the container and are actively consuming resources.
- *Task finished* Tasks that have been finished on time and are no longer consuming resources.
- *Task fail* Tasks that were not completed within the given time.

In summary, the task pool system provides a detailed categorization of tasks in a cluster, which is essential for efficient task management. The system enables users to keep track of the progress of each task and provides valuable insights into resource allocation and utilization.

4.4 Re-scheduling trigger

In real-time scenarios, it is essential to optimize task scheduling to ensure efficient resource utilization. However, most scheduling strategies only schedule tasks once, which may lead to suboptimal results. When the system makes scheduling decisions on tasks, it generates an approximate optimal strategy according to the system state when scheduling. Real-time systems can receive requests at any time, leading to fluctuations in the state of tasks within the system. Sometimes, a task that has been scheduled could have a better scheduling decision in the current system state, even before its container starts execution. However, rescheduling all tasks in the system for each new task arrival can significantly increase the scheduling cost and adversely affect the quality of service. To resolve this issue, the state of unprocessed tasks in the system at time t is described by θ_t.

$$\theta_t = \frac{|Newtask_t|}{|Alltask_t|}, \tag{22}$$

where $Newtask_t$ is the set of new executable tasks at t moment and $Alltask_t$ is the set of tasks that can be excuted but have not started. $\alpha(0 < \alpha < 1)$ is re-schedule factor. When θ_t is greater than α, it indicates that the task state in the system has changed significantly, and rescheduling decisions would be a better choice. By balancing the need for rescheduling with the cost of doing so, the system can achieve optimal performance and maintain a high quality of service.

4.5 Scheduling decision-maker

After completing the aforementioned stages, the system has screened and sorted tasks for scheduling. These tasks will be scheduled based on their priority. When selecting the target container and machine, it is essential to consider schemes that prioritize energy efficiency while ensuring timely task completion. The versatility of the machine should also be taken into account. However, it is crucial to note that excessive deployment of containers on highly versatile machines may result in resource shortages for more demanding tasks that require more resources. Therefore, a comprehensive approach is recommended while selecting scheduling objectives, weighing the energy consumption caused by machine deployment and the universality of the target machine. Such an approach will help us determine the optimal scheduling objectives.

To effectively calculate new energy consumption after deployment, we can categorize the scenarios into the following cases:

- $\mathcal{A}$ In this case, a container has been deployed, and its running time won't exceed the maximum running time of the container deployed on the physical machine it was assigned to.
- $\mathcal{B}$ This scenario occurs when a deployed container's running time exceeds the maximum running time of the container deployed on the physical machine it was assigned to.
- $\mathcal{C}$ In this case, a new container needs to be deployed on a physical machine that has already been powered on, and the running time of the machine won't change.
- $\mathcal{D}$ This scenario occurs when a new container needs to be deployed on a physical machine that has already been powered on, which will increase the running time of the machine.
- $\mathcal{E}$ This is when a new physical machine needs to be deployed to accommodate the new container.
- $\mathcal{F}$ This scenario arises when there are insufficient resources in the cluster to complete the task within the required time.

It is essential to note that the energy consumption estimation post-deployment is critical in ensuring the efficient and effective operation of the cluster. By categorizing the scenarios, it becomes feasible to determine the energy consumption for each scenario and optimally allocate resources. Based on the aforementioned cases, the energy consumption resulting from the deployment tasks can be calculated using the Eq. (23).

$$\Delta E = \begin{cases} \frac{c_j^{cpu}}{p_k^{cpu}} \cdot (p_k^{e\_total} - p_k^{e\_base}) \cdot \bar{t}_r, & \mathcal{A} \\ \frac{c_j^{cpu}}{p_k^{cpu}} \cdot (p_k^{e\_total} - p_k^{e\_base}) \cdot \bar{t}_r + \bar{t}_e \cdot p_k^{e\_base}, & \mathcal{B} \\ \frac{c_j^{cpu}}{p_k^{cpu}} \cdot (p_k^{e\_total} - p_k^{e\_base}) \cdot (\bar{t}_r + \bar{t}_p), & \mathcal{C} \\ \frac{c_j^{cpu}}{p_k^{cpu}} \cdot (p_k^{e\_total} - p_k^{e\_base}) \cdot (\bar{t}_r + \bar{t}_p) + \bar{t}_e \cdot p_k^{e\_base}, & \mathcal{D} \\ \frac{c_j^{cpu}}{p_k^{cpu}} \cdot (p_k^{e\_total} - p_k^{e\_base}) \cdot (\bar{t}_r + \bar{t}_p) + (\bar{t}_r + \bar{t}_p + \bar{t}_s) \cdot p_k^{e\_base}, & \mathcal{E} \end{cases} \tag{23}$$

$$t_r = \frac{t_{mi}^I}{p_k^{ips} \cdot c_j^{cpu}}, \tag{24}$$

where c_j is the container to deploy task t_{mi} and p_k is the PM to deploy task t_{mi}. $\bar{t}_r$ is the execution time of the task. $\bar{t}_e$ is the extended execution time of PM. $\bar{t}_p$ is the start time of container. $\bar{t}_s$ is the start time of PM. The universality of each PM (p_k^u, Eq. (25)) is calculated based on the taints node information submitted by the user.

$$p_k^u = \frac{|\mathcal{C}_{avail}|}{|\mathcal{C}|}, \tag{25}$$

where $\mathcal{C}_{avail}$ is a set of containers which may be deployed to p_k. The higher p_k^u means that p_k is more versatility.

To achieve the scheduling goals, the system is designed to distribute tasks based on minimal energy consumption and more exclusive use of PMs. The energy consumption of deploying t_i to container c_j and PM p_k is calculated by $Q_{i,j,k} \in \mathcal{Q}$ as represented by Eq. (26). The system is programmed to prioritize the deployment of tasks with a higher Q scheme, in order to ensure that the system runs efficiently and effectively.

$$Q_{i,j,k} = \beta \Delta E_{i,j,k} + (1 - \beta) p_k^u, \tag{26}$$

where β ($0 < \beta < 1$) is the weight of energy consumption in scheduling decision.

The detailed procedure is given in Algorithm 1. $task_{ready}$ and $task_{scheduled}$ represent task sets of type task ready and task scheduled in the task pool, respectively. $< Q_{n,i,j}, c_i, p_j >$ means assign $task_n$ to c_i running in p_j. To ensure efficient task scheduling, the system first evaluates the current state of tasks and decides whether to reschedule them or not (lines 2-10 of Algorithm 1). Then, the system gets the Q value of the task that needs to be scheduled and deploys it to the container with the lowest Q value (lines 11-24 of Algorithm 1). If there is no suitable container available, the system creates a new one and selects the PM with the lowest Q value to run the container (lines 25-41 of Algorithm 1). This ensures that tasks are executed in a timely and efficient manner, leading to better overall system performance.

Algorithm 1: RMES

Input: $task_{ready}, task_{scheduled}, \mathcal{C}, \mathcal{P}$
Output: $\{x_{n,j}\}$,$\{y_{i,j}\}$

$scheduletask \leftarrow \emptyset$;
Calculate ,θ_t according Eq. (22);
if $\theta_t > \alpha$ **then**
  foreach $t_n \in task_{scheduled}$ **do**
    $scheduletask \leftarrow scheduletask \cup \{t_n\}$;
  end
end
foreach $t_n \in task_{ready}$ **do**
  $scheduletask \leftarrow scheduletask \cup \{t_n\}$;
end
foreach $t_n \in scheduletask$ **do**
  $target \leftarrow \emptyset$;
  foreach $c_i \in \mathcal{C}$ **do**
    if $c_i^{type} = task_n^{type}$ **then**
      Calculate $Q_{n,i,j}$ according Eq. (26);
      $target \leftarrow < Q_{n,i,j}, c_i, p_j >$;
    end
  end
  if $target \neq \emptyset$ **then**
    select c_i with minimum Q;
    $x_{n,j} \leftarrow 1$;
    $scheduletask \leftarrow scheduletask - \{task_n\}$;
  end
end
if $scheduletask \neq \emptyset$ **then**
  foreach $t_n \in scheduletask$ **do**
    $target \leftarrow \emptyset$;
    create container c_i;
    foreach $pm_j \in \mathcal{P}$ **do**
      Calculate $Q_{n,i,j}$ according Eq. (26);
      $target \leftarrow < Q_{n,i,j}, c_i, p_j >$;
    end
    if $target \neq \emptyset$ **then**
      select c_i with minimum Q;
      $x_{n,j} \leftarrow 1$;
      $y_{i,j} \leftarrow 1$;
      $scheduletask \leftarrow scheduletask - \{task_n\}$;
      $\mathcal{C} \leftarrow \mathcal{C} \cup \{c_i\}$;
    end
  end
end

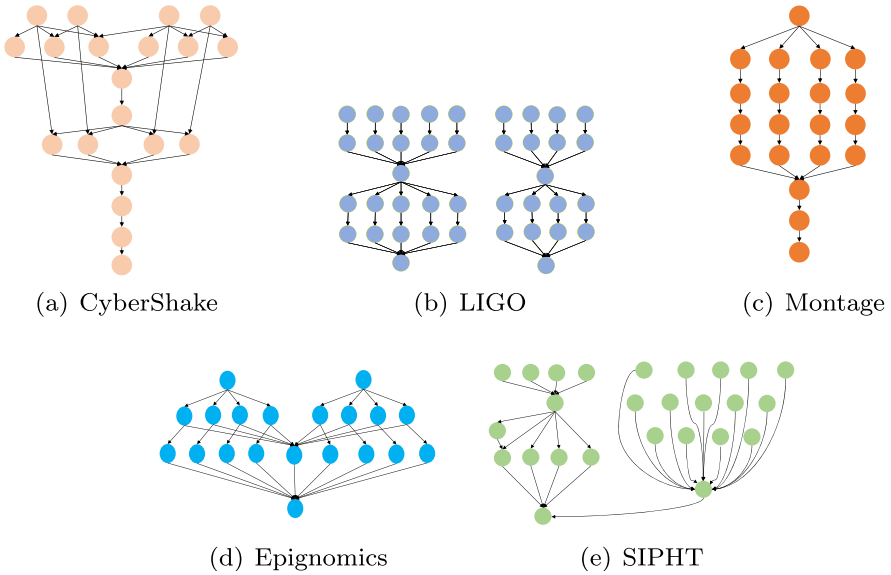

Fig. 3 Sample structure of five scientific workflows used in the evaluation of the algorithms

5 Performance evaluation

5.1 Workflow applications and resource environment

We will evaluate the algorithm using five workflows that were widely used in previous work Juve et al. (2013); Deng et al. (2019): Epigenomics, CyberShake, LIGO, Montage, and SIPHT[1] Those workflows are shown in Fig. 3. Workflow arrival is modeled using the Poisson distribution with an arrive rate (Deng et al. 2019). To solve the deadline-constrained workflow scheduling problem in the actual scenario will not be strictly consistent, we set different deadline factors for workflows, denoted as w_m^{min}. It takes the longest critical path in the workflow to run on the fastest PM.

$$w_m^{trt} = \gamma \cdot w_m^{min}, 2 \leq \gamma \leq 8 \tag{27}$$

The deadline factor γ ranges from 2 (representing very tight deadline constraints) to 8 (relatively loose deadline constraints). When assigning deadline to each workflow, we select γ according to the uniform distribution within the range given in Eq. (27), and then assign the deadline calculated by the selected γ to the workflow.

As shown in Table 1, We use 8 types of PMs (Third quarter 2021). According to the results of parametric experiments, re-schedule factor α is set to 0.05. We establish the simulation in Python 3.8.5 on a Ubuntu 20.04.2 LTS with i5-9500 3.0GHz CPU and 32 GB RAM.

We create six scenarios - five homogeneous ones, each with only one workflow type, and one heterogeneous scenario with all workflow types. Then, we conduct several

[1] The Pegasus project: https://download.pegasus.isi.edu/misc/SyntheticWorkflows.tar.gz.

Table 1 Real-world PM types

Type	CPU cores	Mem(GB)	Basic power (w)	Full load power (w)
PM1	4	4	43	115
PM2	4	8	63	115
PM3	8	8	89.4	173
PM4	8	16	155	269
PM5	8	18	173	334
PM6	8	32	226	294
PM7	16	16	299	521
PM8	32	32	260	748

experiments to assess the performance of each algorithm with different workloads, workload patterns, and workflow types. In the experiment, we use six kinds of workflows, including five homogeneous hybrid workflows and one heterogeneous hybrid workflow. We generate homogeneous hybrid workflows from workflow requests of different sizes belonging to the same structure. Heterogeneous hybrid workflow is composed of five workflow structures: Montage, LIGO, Epigenomics, SIPHT and CyberShake. We randomly select the above five workflows according to equal probability to generate hybrid workflows. In order to conduct our experiments, we have identified three key parameters: arrival rate λ, workflow scale $scale$, and compatibility δ. We have built upon previous studies, such as Deng et al. (2019) and Arabnejad et al. (2019), by using four Poisson distributions to represent λ with arrival rates of 1, 5, 10, and 15 workflows per second. We have also categorized workflow scale as *small*, *medium*, and *large*, with each scale consisting of a combination of multiple workflows. The total number of tasks in each scale is 1000, 2000, and 3000 respectively, and larger scales contain more tasks. By carefully selecting these parameters, we aim to conduct comprehensive experiments that will yield valuable insights into the performance of our system. We set compatibility to 0.2, which represent 20% of the PMs in the cluster as taints of the task. Compatibility δ has three parameters: 0, 0.2 and 0.5, which respectively represent 0%, 20% and 50% of the PMs in the cluster as taints of the task. To ensure the best performance, we assessed the degree of selectivity required for the PMs in the cloud service provider cluster. For each workflow structure, we conducted a series of experiments with varying values for three key parameters. In total, we conducted 36 groups of experiments for each workflow structure.

5.2 Comparison algorithm

To verify the effectiveness of our proposed algorithm, we conduct a comparative analysis with four existing algorithms: Random and Round-Robin, ROSA (Chen et al. 2018) and DWS (Arabnejad et al. 2019). Random is a scheduling strategy that randomly assigns tasks to containers and schedules them to run on random machines. Round Robin is a default scheduling strategy in Kubernetes that assigns tasks to appropriate containers and PMs based on polling rules. ROSA is an uncertainty-aware online

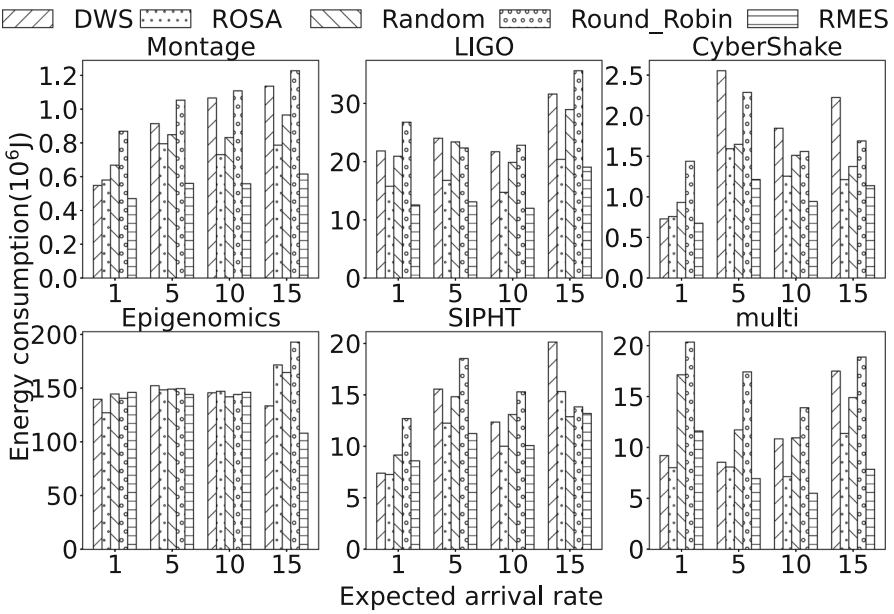

Fig. 4 The energy consumption of each workflow with DWS, ROSA, Round-Robin, Random and RMES

scheduling algorithm for dynamic and multiple workflow scheduling with deadlines. The algorithm first estimates the task completion time, and then schedules the task to minimize the cost. DWS is an online heuristic algorithm that aims to minimize the cost of renting service instances under the deadline. When a new workflow is received, the system sets heuristic rules based on the cost deadline to schedule the task on a more suitable instance.

To ensure a fair experiment, we replace the optimization objective function in ROSA and DWS with the same energy consumption objective function as RMES. This approach allows us to compare the algorithms based on the same criteria and provides a constructive evaluation of their performance.

5.3 Simulation results

5.3.1 Arrival rate

Figures 4 and 5 show the energy consumption and success rate results of the algorithms under different workflow structures. The arrival rate is increased from 1 to 15 in the case of medium workflow scale and compatibility is 0.2. The success rate is the proportion of workflows that meet their scheduling deadline. The experimental results demonstrate that RMES performed exceptionally well, surpassing other algorithms in most cases. Specifically, as the arrival rate increased, RMES algorithm's performance improved significantly. For an arrival rate of 1 workflow per second, RMES algorithm outperformed DWS, ROSA, Random and Round-Robin by −1%, −10%, 18.45%,

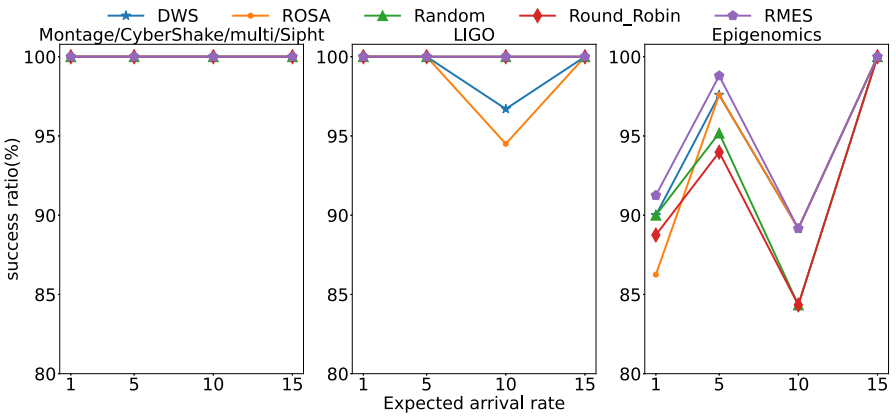

Fig. 5 The Success Rate of each workflow with DWS, ROSA, Round-Robin, Random and RMES

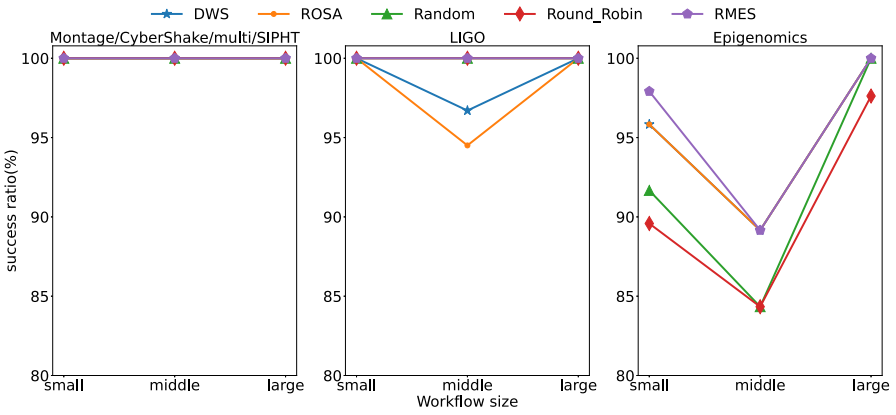

Fig. 6 The Success Rate of each workflow with DWS, ROSA, Round-Robin, Random and RMES

and 34.11%, respectively. When the arrival rate reached 15 workflows per second, RMES algorithm improved to 40.57%, 19.42%, 27.89%, and 39.40%, respectively. With an increased arrival rate, the proportion of newly arrived tasks in the task pool also increased, leading to significant changes in the task pool's status. Therefore, RMES's rescheduling mechanism allowed the unexecuted tasks in the system to be rescheduled promptly. This ensured that the scheduling results of tasks in the task pool were more in line with the current state of the task pool.

In Fig. 6, we can find that under the four workflow structures of CyberShake, Montage, LIGO and multi, the success rate of all algorithms is 100%, but under the two workflow structures of Epigenomics and LIGO, the success rate of all algorithms fluctuates. Nevertheless, RMES still maintains a higher completion rate, and in the worst case, the success rate still reaches the highest 89.2%.

Table 2 The energy consumption of each workflow with DWS, ROSA, Round-Robin, Random and RMES

Workflow, size	Random	Round-robin	DWS	ROSA	RMES
Montage, S	0.47	0.79	0.64	0.52	**0.37**
Montage, M	0.83	1.10	1.06	0.73	**0.55**
Montage, L	1.62	1.77	1.72	**1.17**	1.22
LIGO, S	13.12	18.96	9.95	**9.16**	9.64
LIGO, M	19.86	22.81	21.68	14.71	**11.98**
LIGO, L	23.17	24.69	27.85	19.74	**17.46**
CyberShake, S	0.85	0.95	0.80	0.75	**0.67**
CyberShake, M	1.51	2.55	1.84	1.25	**0.94**
CyberShake, L	1.85	2.44	2.17	1.60	**1.30**
Epigenomics, S	208.48	**205.63**	216.23	216.22	206.24
Epigenomics, M	**141.98**	144.15	145.72	147.13	146.19
Epigenomics, L	163.81	162.89	**160.84**	172.21	162.00
SIPHT, S	12.92	13.14	17.49	12.08	**11.18**
SIPHT, M	13.09	15.20	12.37	**10.02**	10.08
SIPHT, L	30.89	46.99	44.93	28.90	**23.99**
multi, S	6.21	8.06	6.01	4.44	**3.86**
multi, M	10.95	13.91	10.85	10.95	**5.51**
multi, L	27.49	34.94	24.80	17.12	**12.44**

5.3.2 Workflow scale

Table. 2 and Fig. 6 show the energy consumption and success rate results of the algorithm under different sizes and different workflow structures when the arrival rate is 10 workflows per second and the compatibility is 0.2. In Table. 2, 'Montage, S' means Montage workflow small-scale test example. It's worth noting that except for two workflow structures exhibiting fluctuating completion rates, RMES outperforms other algorithms significantly. The results suggest that RMES can be a promising choice for managing workflows with high success rates and low energy consumption. Under the three workflow structures of multi, SIPHT and CyberShake, the improvement of RMES and the comparison algorithm with the best performance increases from 13.06%, 7.45% and 10.66% on a small scale to 27.33%, 16.98% and 18.75% on a large scale. We find that RMES algorithm has a greater improvement under the condition of large-scale workflow. Under Epignomics and LIGO, the success rate of all algorithm has not reached 100% in all workflow scales. Nevertheless, RMES still maintains the relatively highest completion rate.

5.3.3 Compatibility

In order to test the impact of the different compatibility of the submitted workflow with the physical machine on the scheduling results, we conducted several groups of experiments under the condition of medium-scale workflows and arrival rate of 10

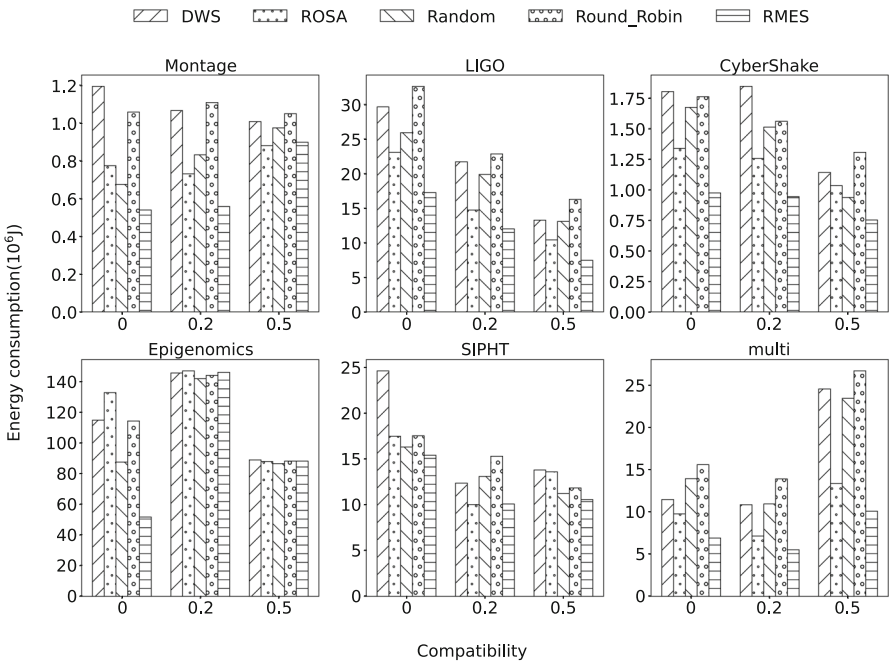

Fig. 7 The energy consumption of each workflow with DWS, ROSA, Round-Robin, Random and RMES

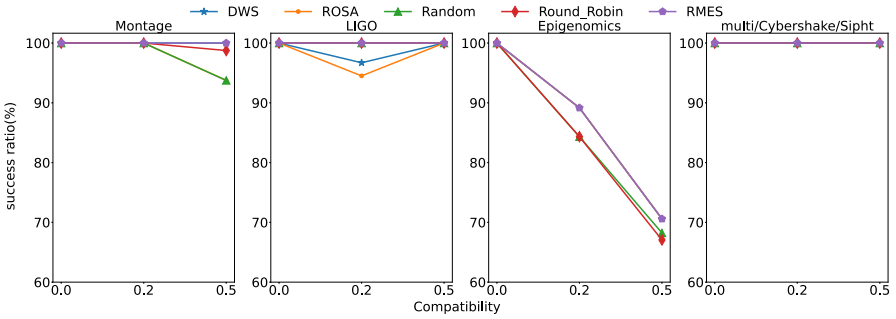

Fig. 8 The Success Rate of each workflow with DWS, ROSA, Round-Robin, Random and RMES

workflows/second, and the results are shown in Figs. 7 and 8. With the improvement of compatibility requirements, under the three workflow structures of Epigenomics, LIGO and Montage, the success rate of all algorithms has decreased to varying degrees. This is because higher compatibility requirements mean that the same task can only be run by fewer machines. As fewer machines can run the task, the completion rate decreases. Since RMES considers the limitation of compatibility when making scheduling decisions, RMES still maintains the highest completion rate among all algorithms as the compatibility becomes more and more stringent. In the experiment of Epigenomics, the success rate can only reach about 70%. However, compared with

other algorithms, RMES algorithm still maintains a higher success rate under more stringent compatibility requirements.

Under the three workflow structures of CyberShake, SIPHT and multi, although the completion rate of all algorithms has reached 100%, RMES is still more advantageous in terms of energy consumption. The overall energy consumption decreases with the increase of compatibility requirements. The main reasons are the following two points: under the condition of high compatibility requirements, more tasks are not executed due to the timeout default of predecessor tasks. This requirement leads to the need for the cluster to open more machines to perform tasks at the same time. It uses more machines, but reduces the execution time of a single machine.

6 Conclusion

In this paper, we focus on optimizing the energy-efficient real-time scheduling of multiple workflows in a container-based cloud. First, we establish a cloud-based workflow scheduling model, which considers resource quantity and performance constraints of container deployment. We then propose an real-time multi-workflow energy-efficient scheduling (RMES) algorithm. RMES can compress the global process time to reduce the base energy consumption by executing tasks in parallel on the running PM. In addition, RMES integrates a rescheduling mechanism to dynamically adjust task scheduling decisions in response to changes in system state. To evaluate the effectiveness of RMES, we conducted a series of experiments under realistic workflow conditions. The results show that RMES significantly outperforms existing algorithms in reducing the energy consumption of cloud service providers (CSPs). In future work, we intend to consider resource competition of containers deployed on the same physical machine in our model to further refine the efficiency and effectiveness of our scheduling approach.

Acknowledgements This work is financially supported by Shenzhen Science and Technology Program under Grant No. JCYJ20210324132406016 and National Natural Science Foundation of China under Grant No. 61732022.

Funding The authors have not disclosed any funding.

Data Availability Enquiries about data availability should be directed to the authors.

Declarations

Conflict of interest The authors have not disclosed any competing interests.

References

Adhikari M, Srirama SN (2019) Multi-objective accelerated particle swarm optimization with a container-based scheduling for internet-of-things in cloud environment. J Netw Comput Appl 137:35–61

Ahmad I, AlFailakawi MG, AlMutawa A, Alsalman L (2021) Container scheduling techniques: a survey and assessment. J King Saud Univ-Comput Inf Sci 34(7):3934–3947

Al-Dulaimy A, Taheri J, Kassler A, HoseinyFarahabady MR, Deng S, Zomaya A (2022) Multiscaler: a multi-loop auto-scaling approach for cloud-based applications. IEEE Trans Cloud Comput 10(4):2769–2786
Arabnejad V, Bubendorfer K, Ng B (2019) Dynamic multi-workflow scheduling: a deadline and cost-aware approach for commercial clouds. Futur Gener Comput Syst 100:98–108
Azad P, Navimipour NJ (2017) An energy-aware task scheduling in the cloud computing using a hybrid cultural and ant colony optimization algorithm. Int J Cloud Appl Comput(IJCAC) 7(4):20–40
Beloglazov A, Buyya R, Lee YC, Zomaya A (2011) A taxonomy and survey of energy-efficient data centers and cloud computing systems. Adv Comput 82:47–111
Chen H, Zhu X, Liu G, Pedrycz W (2018) Uncertainty-aware online scheduling for real-time workflows in cloud service environment. IEEE Trans Serv Comput 14(4):1167–1178
Cheng M, Li J, Nazarian S (2018) DRL-cloud: deep reinforcement learning-based resource provisioning and task scheduling for cloud service providers. In: 2018 23rd Asia and South Pacific design automation conference. pp. 129–134
Deng F, Lai M, Geng J (2019) Multi-workflow scheduling based on genetic algorithm. In: 2019 IEEE 4th International conference on cloud computing and big data analysis (ICCCBDA). pp. 300–305. IEEE
Ding D, Fan X, Zhao Y, Kang K, Yin Q, Zeng J (2020) Q-learning based dynamic task scheduling for energy-efficient cloud computing. Futur Gener Comput Syst 108:361–371
Gan GN, Huang TL, Gao S (2010) Genetic simulated annealing algorithm for task scheduling based on cloud computing environment. In: 2010 International conference on intelligent computing and integrated systems. pp. 60–63. IEEE
Havet A, Schiavoni V, Felber P, Colmant M, Rouvoy R, Fetzer C (2017) Genpack: a generational scheduler for cloud data centers. In: 2017 IEEE International conference on cloud engineering (IC2E). pp. 95–104
Hu B, Cao Z, Zhou M (2022) Scheduling real-time parallel applications in cloud to minimize energy consumption. IEEE Trans Cloud Comput 10(1):662–674
Hussain M, Wei LF, Lakhan A, Wali S, Ali S, Hussain A (2021) Energy and performance-efficient task scheduling in heterogeneous virtualized cloud computing. Sustain Comput: Inf Syst 30:100517
Juve G, Chervenak A, Deelman E, Bharathi S, Mehta G, Vahi K (2013) Characterizing and profiling scientific workflows. Futur Gener Comput Syst 29(3):682–692
Kang KX, Ding D, Xie HM, Yin Q, Zeng J (2021) Adaptive DRL-based task scheduling for energy-efficient cloud computing. IEEE Trans Netw Serv Manag 19(4):4948–4961
Katal A, Dahiya S, Choudhury T (2023) Energy efficiency in cloud computing data centers: a survey on software technologies. Cluster Comput. 26(3):1845–1875
Kaur K, Garg S, Kaddoum G, Ahmed SH, Atiquzzaman M (2020) KEIDS: Kubernetes-based energy and interference driven scheduler for industrial IOT in edge-cloud ecosystem. IEEE Internet Things J 7(5):4228–4237
Merkel D et al (2014) Docker: lightweight Linux containers for consistent development and deployment. Linux J 2014(239):2
Shi T, Ma H, Chen G (2018) Energy-aware container consolidation based on PSO in cloud data centers. In: 2018 IEEE Congress on evolutionary computation (CEC). pp. 1–8
Sun Z, Huang H, Li Z, Gu C, Xie R, Qian B (2023) Efficient, economical and energy-saving multi-workflow scheduling in hybrid cloud. Expert Syst Appl 228:120401
Sun Z, Zhang B, Gu C, Xie R, Qian B, Huang H (2023) ET2FA: a hybrid heuristic algorithm for deadline-constrained workflow scheduling in cloud. IEEE Trans Serv Comput 16(3):1807–1821
Tan B, Ma H, Mei Y (2019) A hybrid genetic programming hyper-heuristic approach for online two-level resource allocation in container-based clouds. In: 2019 IEEE Congress on evolutionary computation (CEC). pp. 2681–2688. IEEE
Third quarter 2021 specpower_ssj2008 results. https://www.spec.org/power_ssj2008/results/res2021q3/ (2021)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.