Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Review article

A Kubernetes-based scheme for efficient resource allocation in containerized workflow

Danyang Liu [a], Yuanqing Xia [b,a,*], Chenggang Shan [c,*], Ke Tian [a], Yufeng Zhan [a]

[a] *School of Automation, Beijing Institute of Technology, Beijing, 100081, China*
[b] *Zhongyuan University of Technology, Zhengzhou, 450007, China*
[c] *School of Artificial Intelligence, Zaozhuang University, Zaozhuang, 277100, China*

ARTICLE INFO

Keywords:
Containerized workflow
Resource allocation
Kubernetes

ABSTRACT

In the cloud-native era, Kubernetes-based workflow engines simplify the execution of containerized workflows. However, these engines face challenges in dynamic environments with continuous workflow requests and unpredictable resource demand peaks. The traditional resource allocation approach, which relies merely on current workflow load data, also lacks flexibility and foresight, often leading to resource over-allocation or scarcity. To tackle these issues, we present a containerized workflow resource allocation (CWRA) scheme designed specifically for Kubernetes workflow engines. CWRA predicts future workflow tasks during the current task pod's lifecycle and employs a dynamic resource scaling strategy to manage high concurrency scenarios effectively. This scheme includes resource discovery and allocation algorithm, which are essential components of our containerized workflow engine (CWE). Our experimental results, across various workflow arrival patterns, indicate significant improvements when compared to the Argo workflow engine. CWRA achieves a reduction in total workflow duration by 0.9% to 11.4%, decreases average workflow duration by a maximum of 21.5%, and increases CPU and memory utilization by 2.07% to 16.95%.

Contents

* Corresponding authors.
E-mail addresses: liu_danyang@bit.edu.cn (D. Liu), xia_yuanqing@bit.edu.cn (Y. Xia), shanchenggang@uzz.edu.cn (C. Shan), tianke32@bit.edu.cn (K. Tian), yu-feng.zhan@bit.edu.cn (Y. Zhan).

https://doi.org/10.1016/j.future.2024.107699
Received 5 September 2024; Received in revised form 19 November 2024; Accepted 26 December 2024
Available online 2 January 2025

1. Introduction

In the era of cloud-native, Docker [1] and Kubernetes [2] have emerged as the leading virtualization solutions. Docker specializes in container packaging, while Kubernetes excels in orchestrating containers across multiple hosts. These technologies are pivotal in managing cloud resources and dominate the cloud-native ecosystem. In scientific computing, which encompasses fields such as astronomy, bioinformatics, materials science, and earth science, workflows [3] are essential. These workflows are typically represented as directed acyclic graphs (DAG), consisting of multiple tasks (nodes) and their interdependencies (directed edges). DAG provides an abstract representation of scientific processes by exchanging shared data files and predefined task dependencies. Utilizing cloud infrastructure powered by Docker and Kubernetes provides the benefits of scalability and high availability, making it an excellent platform for executing scientific workflows [4,5]. Scientific workflows, frequently employed in large-scale applications, require significant computational resources for execution. As a result, the efficient distribution of these resources becomes crucial to ensure optimal workflow performance. Currently, workflow management systems such as Volcano [6], Kubeflow [7], and Argo [8] are responsible for allocating computing resources to tasks within hundreds of workflows on cloud platforms. Computational resource usage in these workflows can vary significantly due to continuous requests and unexpected demand surges. This variability poses significant challenges in resource allocation management, potentially leading to reduced execution efficiency [9]. On the one hand, fixed allocations of computational resources may effectively manage peak demands in scenarios with intense resource usage, but can lead to underutilized and costly resources during quieter periods [10]. On the other hand, resource shortages can cause workflow execution failures, thereby reducing Quality-of-Service (QoS) [11].

Various existing approaches utilize models based on prediction, inference, feedback, heuristics, and learning to address the resource allocation challenge in Kubernetes-based workflow management engines [12–15]. However, these methods often rely on pre-existing knowledge of cloud systems, which limits their adaptability to the dynamic environments of Kubernetes. In addition, the training phase often requires multiple iterations, which can lead to significant computational complexity and resource consumption [16,17]. Additionally, these approaches may not always be compatible with the specific application platforms and technology stacks of Kubernetes-based workflow management engines. Therefore, an efficient and adaptable resource allocation scheme is needed to manage the continuous workflow demands and unexpected resource spikes in Kubernetes-based workflow management engines.

In our previous research, we introduced a Containerized Workflow Engine (CWE) [18] based on Kubernetes. This engine integrates workflow systems with Kubernetes to execute containerized workflows within a Kubernetes cluster. Another work in our research [19,20] explores a resource allocation scheme that does not explicitly specify target nodes but instead relies on the Kubernetes scheduler, making it difficult to accurately identify the optimal node based on task requests. This study presents a Kubernetes-based Containerized Workflow Resource Allocation (CWRA) scheme. CWRA dynamically addresses task pod resource demands through a two-part algorithmic approach: resource discovery and resource allocation. The resource discovery module tracks available and utilized resources across Kubernetes cluster nodes and running task pods. Concurrently, the resource discovery module assesses node resource sufficiency and scheduler workflow loads. The resource allocation module implements a dynamic scaling policy to meet active task pod resource needs, ensuring seamless workflow execution and effective peak demand management.

This study investigates resource allocation, focusing on the dynamic adjustment of resources in response to fluctuating workflow demands. The experimental analysis, which was compared to Argo, was conducted using four scientific workflows across three different workflow arrival scenarios. The results show that CWRA can reduce the total execution time across workflows by 0.9% to 11.4%, with individual workflows experiencing a decrease in processing time by a maximum of 21.5%. Furthermore, CWRA improves CPU and memory utilization by 2.07% to 16.95%, highlighting its effectiveness in resource management. The main contributions of this research are as follows:

- We propose a novel monitoring mechanism supported by a sophisticated resource discovery algorithm that leverages Kubernetes resource characteristics and the Informer component. Integral to the Resource Discovery module, this algorithm systematically collects and analyzes data within Kubernetes clusters.
- We introduce a novel resource allocation method. Unlike traditional static approaches, our mechanism dynamically adjusts resources based on real-time metrics and workload demands. This approach ensures optimal resource utilization by continuously monitoring performance indicators, minimizing waste, and enhancing Kubernetes cluster efficiency.
- We conducted extensive experiments using the CWRA algorithm across four distinct scientific workflows deployed on Kubernetes clusters. The results demonstrate that CWRA consistently outperforms the baseline algorithm in terms of performance metrics.

This paper is structured as follows: Section 2 reviews related work. Section 3 introduces the system model. Section 4 details the architecture. Section 5 illustrates the implementation of our resource allocation scheme. Section 6 provides the experimental evaluation. Section 7 concludes this paper.

2. Related work

Resource allocation policies in workflow management engines are heavily influenced by virtualization technologies in cloud infrastructures. This influence determines whether workflow tasks are managed by virtual machine (VM) instances or executed using container technologies. This section reviews the historical development of resource allocation policies and systematically discusses four primary approaches from the perspective of technology evolution: virtualized, containerized, cloud-native, and workflow/task scheduling algorithms. Although this discussion focuses on workflow management engines, the resource allocation policies discussed are broadly applicable across various cloud computing environments.

2.1. Virtualization resource allocation

Virtualization is crucial in cloud computing for efficient resource allocation and scalability. Various strategies have been studied to improve resource allocation in virtualized environments. Islam et al. [21] developed prediction-based resource provisioning strategies using neural networks and linear regression to address delays in hardware resource allocation. Their approach anticipates resource demands, thereby improving the efficiency of cloud resource allocation. Hoenisch et al. [22] introduced a self-adaptive resource allocation mechanism for business process management systems (BPMS) using cloud technologies. This mechanism dynamically adjusts resources based on real-time requirements, ensuring optimal performance and resource efficiency. Lee et al. [23] proposed the Maximum Effective Reduction (MER) algorithm to optimize the resource efficiency of workflow schedules. The MER algorithm minimizes idle time between tasks and consolidates resource usage, significantly reducing inefficiencies in virtualized environments. Deelman et al. [24] described Pegasus, a workflow management system that automates scientific simulations and data analysis. Pegasus coordinates data movement and task execution across various computational resources, optimizing the utilization of virtualized resources. Collectively, these studies advance the field of virtualization resource allocation by proposing predictive, adaptive, and efficient strategies. They highlight the importance of dynamic provisioning and efficient scheduling in virtualized environments.

2.2. Containerization resource allocation

Containerization has emerged as a vital technology for efficient resource allocation in cloud computing environments. The following studies offer significant insights and advancements in container-based resource allocation. For example, Xu et al. [25] introduced an innovative resource scheduling algorithm for container-based clouds, focusing on enhancing resource utilization and system performance. Their approach showed greater efficiency than traditional methods, laying a foundation for future research in container resource scheduling. Yin et al. [26] explored task scheduling and resource allocation in fog computing for smart manufacturing. They developed a container-based framework that optimized task execution and resource distribution, resulting in reduced latency and better resource utilization. This study underscored the benefits of container technology in industrial applications. Using the NSGA-II algorithm, Tan et al. [27] introduced a multi-objective optimization strategy for micro-service allocation in container-based clouds. Their method balanced performance, cost, and resource utilization, highlighting the advantages of multi-objective optimization in cloud environments. Hu et al. [28] addressed container scheduling in heterogeneous clusters with multi-resource constraints. They developed an algorithm to manage concurrent containerized workloads, enhancing scheduling efficiency and resource utilization across diverse hardware environments.

2.3. Cloud-native resource allocation

Cloud-native resource allocation has become a crucial area of research as organizations increasingly adopt containerized environments like Kubernetes to manage their applications. For example, Medel et al. [29] focused on modeling the performance and resource management aspects of Kubernetes environments. Their study addressed the challenges of resource allocation in containerized environments by proposing a framework that models the resource requirements and performance characteristics of applications. This approach offered valuable insights on efficient resource allocation strategies, demonstrating the potential to enhance Kubernetes resource management. Chang et al. [30] introduced a Kubernetes-based monitoring platform designed for dynamic cloud resource provisioning. The platform leverages Kubernetes to provide real-time monitoring and adaptive resource provisioning, ensuring the optimal performance and resource utilization of applications in dynamic cloud settings. Rattihalli et al. [31] explored non-disruptive vertical auto-scaling and resource estimation in Kubernetes. Their research investigated techniques for dynamically adjusting the resource allocations of running containers without causing disruption. By proposing algorithms and tools for seamless vertical scaling and accurate resource estimation, the study showed significant improvements in the scalability and resource efficiency of Kubernetes-managed applications. Kim et al. [32] examined resource management strategies within Kubernetes. They analyzed the existing resource management features of Kubernetes and identified areas for potential improvement. Their proposed enhancements to Kubernetes' resource scheduling and allocation processes were designed to improve the overall performance and reliability of containerized applications. Ding et al. [33] focused on microservice placement strategies in Kubernetes, emphasizing dynamic resource allocation. They presented a framework for optimizing the placement of microservices within a Kubernetes cluster, accounting for dynamic resource demands. The adaptive algorithm proposed in their study dynamically allocated resources based on real-time workload changes, ensuring efficient utilization and high performance. However, the resource allocation schemes in the above studies predominantly rely on open-source tools from the CNCF community, such as Prometheus, to construct resource monitoring systems. These tools are instrumental in monitoring cluster resource utilization and formulating resource provisioning policies.

2.4. Workflow and task scheduling algorithms

Efficient scheduling of workflows and tasks is vital for enhancing performance and resource utilization in distributed computing environments. Numerous algorithms have been proposed to address the challenges of workflow scheduling, considering factors such as execution time, cost, resource heterogeneity, and QoS constraints. For example, Dynamic level scheduling [34] estimates processor availability, orders tasks, and selects processors using the earliest start time rule. For heterogeneous systems, Topcuoglu et al. [35] proposed the heterogeneous earliest finish time method, which sorts tasks in descending order based on the bottom-level value (b-level), representing the longest path from a DAG node to the bottom. It then uses the earliest finish time to insert tasks into idle processor slots. To address the heterogeneity of systems, Sakellariou et al. [36] studied the impact of using different computational methods (e.g., mean, median, worst-case, best-case) for DAG variables in list scheduling algorithms. In addition to DAG structural properties, some list scheduling algorithms [37] order tasks based on predicted processor states. Beyond single workflow scheduling, some studies have focused on multi-workflow scheduling, where multiple workflows are scheduled simultaneously. Hsu et al. [38] designed the OWM algorithm to solve the mixed-parallel workflow problem, achieving excellent results in reducing average workflow completion time. However, this algorithm only considers idle resources when matching tasks to processors, ignoring the possibility that tasks might complete earlier on busy processors than on idle ones, which could lead to excessive task waiting times and workflow completion delays. Hamid et al. [39] proposed the fairness dynamic workflow sharing algorithm, which addresses the flaw in the OWM algorithm by considering both idle and busy resources when assigning tasks to processors, thereby reducing overall multi-workflow completion time. Yang et al. [40] proposed the forward-ahead workflow scheduling algorithm, which aims to minimize workflow execution costs under deadline constraints. Ye et al. [41] designed a workflow scheduling scheme to meet reliability constraints in IaaS cloud environments while reducing energy consumption. The scheme uses an updating method to adjust task sub-reliability constraints in order to lower energy consumption. These algorithms highlight the importance of considering various factors such as heterogeneity, dynamism, energy efficiency, and QoS in designing effective scheduling algorithms.

Overall, the continuous evolution of cloud technologies necessitates ongoing research to refine resource allocation policies, aiming to achieve a balance between performance, cost-efficiency, and scalability in increasingly complex cloud infrastructures.

3. System model

This section outlines the application of our CWRA to effectively manage continuous workflow requests and mitigate the impact of unexpected spikes in resource demand. Aiming to optimize resource utilization and ensure adherence to predefined Service Level Objectives (SLO).

3.1. System description

We consider a distributed computing environment orchestrated by a Kubernetes cluster. This environment is composed of a set of nodes (virtual machines or physical servers), denoted by $N = \{N_1, N_2, \dots, N_m\}$, where m represents the total number of nodes within the cluster. Each node in the cluster is equipped with a set of available CPU cores $C = \{C_1, C_2, \dots, C_m\}$, and memory capacity $M = \{M_1, M_2, \dots, M_m\}$. The set of workflows submitted to the Kubernetes cluster for execution is represented by $W = \{W_1, W_2, \dots, W_z\}$, where z indicates the total number of workflows. Each workflow is defined as a sequence of tasks, $W_k = \{sla_{w_k}, t_{k,1}, t_{k,2}, \dots, t_{k,n}\}$, where sla_{w_k} represents the Service Level Agreement (SLA) associated with workflow. $t_{k,1}, t_{k,2}, \dots, t_{k,n}$ denotes the tasks of the workflow. Each workflow task is defined as:

$$\begin{aligned} t_{k,i} = \{&name, image, cpu, mem, dependencies, sla_{t_{k,i}}, \\ &env\{min_{cpu}, min_{mem}, \}\}, 1 \le k \le z \text{ and } 1 \le i \le n \end{aligned} \tag{1}$$

Herein, *name* represents the task name, *image* denotes the required Docker image, *cpu* and *mem* indicate the CPU and memory resources needed for the task, respectively. *dependencies* lists the prerequisite tasks that must be completed before this task can commence. The *env* field specifies the minimum CPU and memory required to execute the task, denoted by min_{cpu} and min_{mem}. The SLA for each workflow task is represented by $sla_{t_{k,i}}$, and the SLA for the entire workflow is represented by sla_{w_k}, as follows:

$$sla_k = \{slo_1, slo_2, \dots, slo_n\}, k \in \{w_k, t_{k,i}\} \tag{2}$$

The sole SLO in this workflow is the task deadline, which necessitates the completion of each task before its designated deadline. The workflow adheres to this standard uniformly.

$$\begin{aligned} sla_{w_k} &= deadline_{w_k} \\ sla_{t_{k,i}} &= deadline_{t_{k,i}} \end{aligned} \tag{3}$$

The deadline for the final task, denoted as $t_{k,last}$ in a workflow, coincides precisely with the overall deadline for the workflow.

$$deadline_{w_k} = deadline_{t_{k,last}} \tag{4}$$

3.2. Problem formulation

In this subsection, we delve into the optimization problem within our CWRA. For each task requested in workflow W_k, the allocated CPU and memory resources are defined as follows:

$$\begin{aligned} C &= \{c_{i,1}, c_{i,2}, \dots, c_{i,m}\} \\ M &= \{m_{i,1}, m_{i,2}, \dots, m_{i,m}\} \end{aligned} \tag{5}$$

Let $x^k_{i,j}$ be the decision variable for task placement in the ith workflow. Here, $1 \le i \le n$ and $1 \le j \le m$. Here, $1 \le i \le n$ and $1 \le j \le m$. The variable is defined as follows:

$$x^k_{i,j} = \begin{cases} 1 & \text{task } i \text{ in } W_k \text{ is on node } N_j \\ 0 & \text{task } i \text{ in } W_k \text{ is not on node } N_j \end{cases} \tag{6}$$

Each node N_j has limited CPU and memory capacities that cannot be exceeded. Let C^{max}_j and M^{max}_j denote the maximum CPU and memory capacities of node N_j, respectively. The constraints are expressed as:

$$\sum_{i=1}^{n} \sum_{k=1}^{z} c_{i,j} x^k_{i,j} \le C^{max}_j, \quad \forall j \in 1, \dots, m \tag{7}$$

$$\sum_{i=1}^{n} \sum_{k=1}^{z} m_{i,j} x^k_{i,j} \le M^{max}_j, \quad \forall j \in 1, \dots, m \tag{8}$$

Each task in the workflow must be assigned to exactly one node. This constraint is expressed as:

$$\sum_{j=1}^{m} \sum_{k=1}^{z} x^k_{i,j} = 1, \quad \forall i \in 1, \dots, n \tag{9}$$

Tasks within a workflow may have dependencies that dictate their execution order. Let $P(i)$ denote the set of predecessor tasks for task i. For any task i to start, all tasks in $P(i)$ must be completed This introduces the following constraint:

$$start^k_i \ge \max_{p \in P(i)} (finish^k_p + delay^k_{p,i}) \tag{10}$$

$$T = \max_{i \in 1,\dots,n} (finish^k_i - start^k_1) \tag{11}$$

where T is the makespan of the workflow, $start_i$ denotes the start time of task i, $finish_p$ denotes the finish time of task p, and $delay_{p,i}$ represents the communication delay between tasks p and i.

The objective function to be minimized combines resource utilization and task execution time. Let U_j denote the resource utilization of node N_j, defined as:

$$U_j = \frac{\sum_{i=1}^{n} \sum_{k=1}^{z} c_{i,j} x^k_{i,j}}{C^{max}_j} + \frac{\sum_{i=1}^{n} \sum_{k=1}^{z} m_{i,j} x^k_{i,j}}{M^{max}_j}, \quad \forall j \in 1, \dots, m \tag{12}$$

The overall objective function f can then be expressed as:

$$f = \alpha \sum_{j=1}^{m} U_j + \beta \sum_{k=1}^{z} T^k \tag{13}$$

where α and β are weighting factors that balance the trade-off between resource utilization and task execution time. To solve this optimization problem, we aim to achieve efficient resource allocation that maximizes resource utilization and minimizes the total execution time of the workflows.

4. Architecture

This section describes the system architecture of the Containerized Workflow Engine (CWE), focusing on its framework and primary modules.

4.1. CWE framework

Fig. 1 illustrates the CWE as a workflow management engine that manages, schedules, and executes containerized workflow tasks. The core functions are explained below:

- Workflow Management: Enables precise control of workflow tasks, allowing users to initiate, stop, and monitor tasks dynamically.
- Scheduling: Prioritizes the sequential ordering of tasks, ensuring that each is scheduled in its correct operational order.
- Resource Allocation: Optimizes the distribution of computing resources, such as CPU and memory, to meet task demands autonomously.
- Error Handling: Automatically detects and responds to task failures or system anomalies, minimizing downtime and maintaining continuous operation.

Overall, the CWE for CWRA provides a powerful and flexible solution for managing containerized workflow tasks, ensuring efficient and reliable execution in diverse and demanding environments.

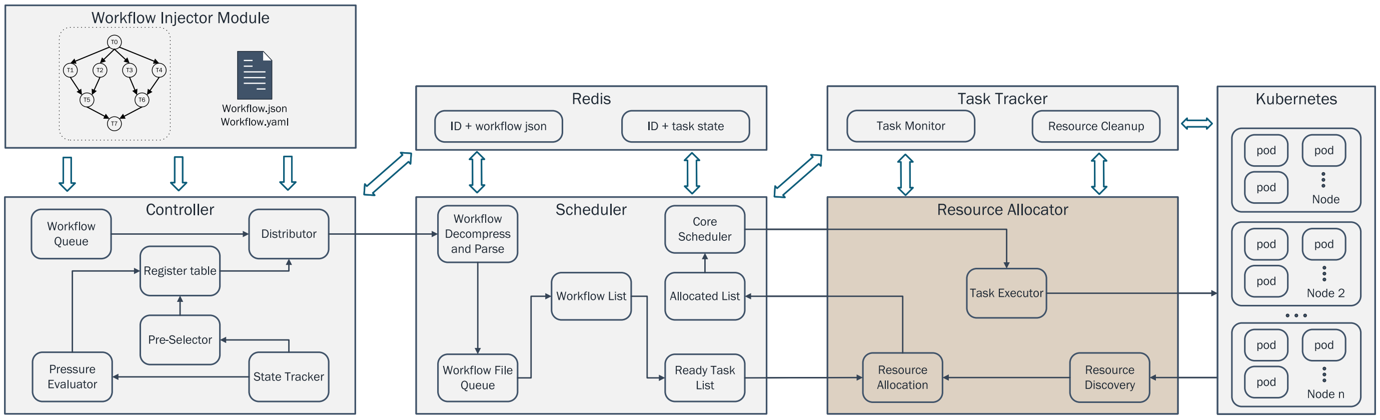


Fig. 1. CWE architecture.

4.2. CWE modules

As illustrated in Fig. 1, the CWE comprises five top-level entities: workflow injection module, controller module, scheduler module, resource allocation module, and task tracking module. This paper will particularly focus on the Resource Allocation Module as it relates to CWRA.

Workflow Injector Module: This module operates as independent pods, responsible for reading variable configuration information for workflow definitions from the mounted directory. It analyzes and packages the workflow to respond to interface requests, thereby generating subsequent workflow tasks.

Controller Module: This module integrates several components to effectively manage and optimize workflow operations across multi-cluster Kubernetes environments. The Distributor dispatches workflows to the Scheduler Module within a multi-cluster Kubernetes environment, enhancing resource utilization and task execution through flexible allocation and load-balancing strategies. The Pre-Selector creates a register table using current Kubernetes resource data and scheduler pressure, assisting in accurately routing workflows to the most suitable scheduler module. It dynamically updates the table based on resource evaluation algorithms to maintain accuracy in workflow assignment and scheduler load balancing. Moreover, the Pressure Evaluator continuously monitors workload pressure across clusters and scheduler modules, utilizing both real-time and historical data to optimize workload distribution. This component aids in balancing workloads to maximize resource utilization and enhance overall system performance. Finally, the State Tracker manages the statuses of workflows and scheduler modules, monitors progress, and addresses any failures by rescheduling tasks as necessary. It plays a critical role in ensuring the reliability and effectiveness of workflow execution by maintaining an up-to-date state table and resolving potential issues promptly. Together, these components enable the Controller Module to achieve optimal performance and reliability in a distributed environment.

Scheduler Module: This module plays a pivotal role in implementing the core algorithm for workflow scheduling. It is responsible for the orchestration of workflow tasks and ensuring the efficient utilization of resources while meeting specific user demands. Upon receiving workflow tasks from the controller, the scheduler initially decompresses and parses the workflow. The parsed workflow information is then stored in the $workflowFileQueue$. To ensure independent management and tracking of each workflow, the scheduler assigns a unique identifier to each one and adds it to the $workflowList$. The scheduler updates the $readyTaskList$, containing all tasks $t_{k,i}$ that are ready and meet execution conditions. The $readyTaskList$ is then forwarded to the resource allocator for resource assignment. Upon completing this allocation, the resource allocator returns the $allocatedList$ to the scheduler. Based on the feedback, the scheduler sends specific task details to the resource allocator to facilitate task execution. Using the $allocatedList$ and the environment settings, the scheduler employs a combination of scheduling algorithms, such as the earliest deadline first or the shortest task priority algorithm, to determine the optimal task scheduling sequence. Once this sequence is established, the scheduler issues a task start request to the resource allocator in the determined order. Additionally, the scheduler continuously retrieves task execution statuses from the task tracker and logs this data in a Redis database. Should issues arise during task execution, such as resource bottlenecks or failures, the scheduler promptly adjusts or reschedules tasks to ensure continuous workflow operation and performance optimization.

Resource Allocator Module: This module is responsible for managing resource distribution for workflow tasks and consists of three key components: resource discovery, resource allocator, and task executor. The resource discovery component utilizes the Kubernetes informer API to continuously monitor the availability and status of resources within the Kubernetes cluster, including nodes and pods, with a specific focus on CPU and memory. The resource allocator component evaluates whether the current state of the Kubernetes cluster meet the resource demands of the workflow. It assesses the resource requirements specified by the workflow tasks and compares them with the available resources in the cluster. Based on this evaluation, the allocator implements predefined strategies to generate an optimized resource allocation plan, $allocatedList$. The task executor component creates isolated namespaces to ensure workflow task independence and utilizes the client-go library to create and manage task containers based on task information and resource allocation results. Additionally, it supports data sharing between tasks by creating Persistent Volumes (PV) and Persistent Volume Claims (PVC), thereby enhancing execution efficiency and data availability.

Task Tracker Module: This module effectively monitors the execution status of each task container within workflows by leveraging Kubernetes' List-Watch mechanism. It provides real-time feedback to the scheduler, enabling decisions based on the most current task execution data, thereby supporting the orderly and efficient execution of workflow tasks. Additionally, the task tracker is responsible for resource cleanup. It regularly inspects and removes Pods that are in a completed state (successful, failed, OOMKilled) and cleans up workflow namespaces that no longer contain unfinished tasks, thereby optimizing cluster resource use and allocation.

Redis: The Redis database, which can be deployed internally within the cluster or externally, is responsible for storing raw JSON workflow data and tracking the corresponding execution statuses. Furthermore, Redis supports a frontend system, which facilitates real-time user queries of workflow statuses.

Within the CWE framework, the Controller is developed in Rust, leveraging Rust's strong concurrency capabilities to support the operation of large-scale workflows across multiple Kubernetes environments.

The Scheduler is implemented in Java, leveraging its extensive library support to enhance workflow scheduling flexibility. The Resource Allocator and Task Tracker are written in Go, benefiting from Go's robust support within the Kubernetes ecosystem, which enhances the efficiency of resource allocation and task tracking. All modules utilize gRPC for effective cross-language communication. The deployment process of the CWE has been meticulously designed, allowing users to easily deploy by modifying the Kubernetes cluster's YAML configuration files. Additionally, workflow containerization within CWE is fully automated, greatly reducing the need for manual configuration and management, thereby facilitating more convenient and efficient system maintenance. For more details about CWE, please refer to [18]

5. Resource allocation scheme

This section systematically discusses two algorithms: resource discovery and resource allocation. These algorithms are crucial for efficiently handling resource requests in workflow tasks by the resource allocator. They operate sequentially to dynamically accommodate varying resource demands. This dynamic process enables flexible adaptation to changing task requirements. Table 1 clarifies these algorithms by defining the relevant notations. Furthermore, the workflow's execution status is indicated by a set of state data, reflecting the current conditions of all tasks within workflow. The state data for each task is recorded as follows:

$$s_{k,i} = \{w_k, t_{start}, t_{end}, duration, cpu, mem, state\},\quad 1 \le k \le z, 1 \le i \le n \tag{14}$$

Here, w_k represents the unique identifier for the workflow associated with the current task. t_{start} denotes the start time of the current task's pod within the Kubernetes cluster, while t_{end} indicates the task's completion time. $duration$ refers to the running time of the task pod. The cpu and mem variables represent the CPU resources and memory capacity allocated to the task, measured in milli-cores and megabytes, respectively. The $state$ is a string that describes the task pod's execution state, with possible states being "successful", "failed", "unexecuted", and "executed".

In the Kubernetes container orchestration system, the Pod serves as the essential runtime unit, seamlessly integrating with CWE's non-intrusive automatic execution process. This collaboration enables the resource allocator to efficiently meet resource demands by provisioning resources for the entire duration of the task Pod's lifecycle. Users can define the minimum CPU and memory requirements through the JSON configuration file. These parameters are set in the environment variables for the task Pod, ensuring that the containers within the task Pod operate stably throughout their runtime.

5.1. Resource discovery algorithm

Algorithm 1 presents a resource discovery algorithm that efficiently identifies and reports unused resources within a Kubernetes cluster, both at the overall cluster level and for each individual node. The results are represented by $total\_remain.cpu$, $total\_remain.mem$, and $remainMap$. Initially, the algorithm resets the cluster's total resource parameters to zero. It then retrieves the $PodList$ and $NodeList$ from the cluster using an Informer (lines 1–3). The algorithm iterates through each node, resetting the node resource parameters and then calculating the total resources consumed by "Running" and "Pending" Pods on that node (lines 4–11). Subsequently, the remaining CPU and memory resources for the current node are computed (lines 12–14). These values are aggregated to determine the total remaining CPU and memory resources for the entire cluster (lines 15–16). Upon completing the iteration through all nodes in the Kubernetes cluster, the algorithm returns the remaining resources as $total\_remain.cpu$, $total\_remain.mem$, and $remainMap$.

Table 1
Major notations.

Notation	Meaning
α	Scaling factor for remaining resources allocation
$allocated.cpu$	Allocated CPU resources for the task
$allocated.mem$	Allocated memory resources for the task
$allocatedList$	List of resources allocated for tasks
cpu_scal	Scaled CPU allocation based on Eq. (15)
mem_scal	Scaled memory allocation based on Eq. (15)
$NodeLister$	Kubernetes informer for listing nodes
$NodeList$	List of nodes obtained from $NodeLister$
$p\_request.cpu$	CPU resources requested by a pod
$p\_request.mem$	Memory resources requested by a pod
$PodLister$	Kubernetes informer for listing pods
$PodList$	List of pods obtained from $PodLister$
$readyTaskList$	List of tasks ready for execution
$remain.cpu$	Remaining CPU resources of a node
$remain.mem$	Remaining memory resources of a node
$remainMap$	Map of remaining resources for each node
$request.cpu$	CPU resources requested by pods on a node
$request.mem$	Memory resources requested by pods on a node
$task\_request.cpu$	CPU resources requested by the task
$task\_request.mem$	Memory resources requested by the task
$total.cpu$	Total CPU resources of a node
$total.mem$	Total memory resources of a node
$total\_remain.cpu$	Total remaining CPU resources in the cluster
$total\_remain.mem$	Total remaining memory resources in the cluster
$total\_request.cpu$	Total CPU resources requested by all tasks
$total\_request.mem$	Total memory resources requested by all tasks
w_c	Weight for CPU resources in node scoring
w_m	Weight for memory resources in node scoring

Algorithm 1 Resource Discovery Algorithm

```
Require: $PodLister$, $NodeLister$
 1: Initalize $total\_remain.cpu$, $total\_remain.mem$ to 0
 2: Get $PodList$ from $PodLister$ informer
 3: Get $NodeList$ from $NodeLister$ informer
 4: for each node $n$ in $nodeList$ do
 5:     Initalize $request.cpu$, $request.mem$, $remain.cpu$, $remain.mem$ to 0
 6:     for each pod $p$ in $podList$ do
 7:         if ($p\_phase$ == "Running" or "Pending") then
 8:             $request.cpu$ += $p\_request.cpu$
 9:             $request.mem$ += $p\_request.mem$
10:         end if
11:     end for
12:     $remain.cpu = total.cpu - request.cpu$
13:     $remain.mem = total.mem - request.mem$
14:     $remainMap[n] = \{remain.cpu, remain.mem\}$
15:     $total\_remain.cpu$ += $remain.cpu$
16:     $total\_remain.mem$ += $remain.mem$
17: end for
18: return $total\_remain.cpu$, $total\_remain.mem$, $remainMap$
```

5.2. Resource allocation algorithm

Algorithm 2 provides a detailed description of the resource allocation process. Upon receiving the $readyTaskList$ from the scheduler, the algorithm begins execution and returns the $allocatedList$, which includes the allocated CPU and memory resources, along with the scheduling nodes. There may be competition for computational resources among the ready tasks. To manage this contention effectively, we introduce a dynamic resource scaling method. This method dynamically adjusts CPU and memory resource allocation based on the following formulas:

$$\begin{aligned} cpu\_scal &= task\_request.cpu \cdot \frac{total\_remain.cpu}{total\_request.cpu} \cdot (1 - \frac{remain.cpu}{total.cpu}) \\ mem\_scal &= task\_request.mem \cdot \frac{total\_remain.mem}{total\_request.mem} \cdot (1 - \frac{remain.mem}{total.mem}) \end{aligned} \tag{15}$$

To ensure efficient resource allocation, the algorithm evaluates each node by assigning a score. The node with the highest score is selected for task placement. The node score is calculated using the following formula:

$$n\_score = \frac{remain.cpu}{task\_request.cpu} \cdot w_c + \frac{remain.mem}{task\_request.mem} \cdot w_m \tag{16}$$

In Algorithm 2, six comparison conditions labeled A1, A2, B1, B2, C1, and C2 are introduced (lines 2–7). The symbol ¬ denotes the negation of a condition (line 8). The algorithm iterates through each task in the $readyTaskList$ and calculates the required CPU and memory resources (lines 9–12). It then calculates the score of task on each node using Eq. (16) and selects the node with the highest score. Next, the algorithm calculates the CPU and memory resources to be allocated when scaling is needed, according to Eq. (15) (lines 14–15). Finally, the algorithm determines four distinct resource allocation schemes by comparing cumulative resource requests during the task lifecycle with the total available remaining resources.

Sufficient remaining resources. When the remaining resources of the Kubernetes cluster can meet the demands of all concurrent tasks during their lifecycle, the algorithm further evaluates whether conditions B_1 and B_2 are fulfilled. If the CPU and memory resources requested by a task are less than the remaining resources on a node, the algorithm allocates resources according to the task's request (lines 18–20). However, if the node's maximum available CPU resources cannot satisfy the current task's CPU request, the algorithm introduces a scaling factor, α, to proportionally allocate the remaining CPU resources (lines 21–23). Similarly, if the node's maximum available memory resources are insufficient to meet the task's memory request, the algorithm allocates the scaled remaining memory resources (lines 24–26). If both the CPU and memory resources on the node are insufficient, the algorithm adjusts the allocation of both using the scaling factor α (lines 27–29).

Insufficient remaining resources. When the remaining CPU resources in the Kubernetes cluster are insufficient, condition $\neg A_1$ is met, prompting the algorithm to evaluate whether conditions C_1 and B_2 are met. If both C_1 and B_2 are satisfied, the algorithm allocates resources based on cpu_scal and the task's memory request (lines 33–35). However, if the maximum available CPU resources on the node are inadequate to meet cpu_scal, conditions $\neg C_1$ and B_2 hold, the algorithm allocates the remaining CPU resources using the scaling factor α. Given that the node has sufficient memory resources, the algorithm allocates memory as requested by the task (lines 36–38). If the node's maximum remaining memory resources are insufficient to meet the task's memory request, conditions C_1 and $\neg B_2$ hold, the algorithm allocates resources according to cpu_scal and the node's maximum remaining memory resources using the scaling factor α(lines 39–41). Lastly, when conditions $\neg C_1$ and $\neg B_2$ hold, the algorithm proportionally allocates the node's maximum remaining resources using the scaling factor α (lines 42–44).

When the remaining memory resources in the Kubernetes cluster are insufficient to meet the memory requirements of all tasks, the algorithm evaluates conditions B_1 and C_2 during resource allocation. In scenarios involving B_1 and C_2, B_1 and $\neg C_2$, $\neg B_1$ and C_2, and $\neg B_1$ and $\neg C_2$, the algorithm follows a similar strategy to the one previously mentioned, focusing specifically on memory resources (lines 47–60). Similarly, when both CPU and memory resources are insufficient in the Kubernetes cluster, conditions $\neg A_1$ and $\neg A_2$ are met. For scenarios involving C_1 and C_2, C_1 and $\neg C_2$, $\neg C_1$ and C_2, or $\neg C_1$ and $\neg C_2$, the algorithm follows a similar strategy as described earlier (lines 62–74). Finally, the algorithm checks if the resources allocated to the task meet the task's minimum requests. If minimum resources are insufficient, the task is placed at the top of the $readyTaskList$ for the next lifecycle. If the requirements are met, an allocation scheme is generated, detailing the allocated CPU, memory, and target nodes. This algorithm ensures efficient resource utilization and prevents task failures due to insufficient resources. Additionally, the use of scaling factors and resource adjustments allows the algorithm to dynamically allocate resources based on availability.

Our CWRA algorithm is a heuristic approach to the addresses NP-hard resource allocation problem. The time complexity of this algorithm is $O(T \times M)$, where T is the total number of tasks in the $readyTaskList$ and M is the number of nodes.

Algorithm 2 Resource Allocation Algorithm

```
Require: $readyTaskList$, $remainMap$, $total\_remain.cpu$, $total\_remain.mem$, $total\_request.cpu$, $total\_request.mem$
    /* Define conditions */
 2: $A_1 \leftarrow total\_request.cpu < total\_remain.cpu$
    $A_2 \leftarrow total\_request.mem < total\_remain.mem$
 4: $B_1 \leftarrow task\_request.cpu < remain.cpu$
    $B_2 \leftarrow task\_request.mem < remain.mem$
 6: $C_1 \leftarrow cpu\_scal < remain.cpu$
    $C_2 \leftarrow mem\_scal < remain.mem$
 8: ¬ denotes the negation of a condition
    for each task $t_{k,i}$in readyTaskList do
10:     $total\_request.cpu$ += $task\_request.cpu$
        $total\_request.mem$ += $task\_request.mem$
12: end for
    for each task $t_{k,i}$in readyTaskList do
14:     Compute $n\_score$ using Eq.(16) and select the Maximum score $n$
        Compute $cpu\_scal$ and $mem\_scal$ using Eq.(15)
    /* (1) The remaining resources are sufficient */
16:     if $A_1$ and $A_2$ then
            if $B_1$ and $B_2$ then
18:             $allocated.cpu = task\_request.cpu$
                $allocated.mem = task\_request.mem$
20:         else if $\neg B_1$ and $B_2$ then
                $allocated.cpu = remain.cpu \times \alpha$
22:             $allocated.mem = task\_request.mem$
            else if $B_1$ and $\neg B_2$ then
24:             $allocated.cpu = task\_request.cpu$
                $allocated.mem = remain.mem \times \alpha$
26:         else
                $allocated.cpu = remain.cpu \times \alpha$
28:             $allocated.mem = remain.mem \times \alpha$
            end if
    /* (2) The remaining CPU is insufficient */
30:     else if $\neg A_1$ and $A_2$ then
            if $C_1$ and $B_2$ then
32:             $allocated.cpu = cpu\_scal$
                $allocated.mem = task\_request.mem$
34:         else if $\neg C_1$ and $B_2$ then
                $allocated.cpu = remain.cpu \times \alpha$
36:             $allocated.mem = task\_request.mem$
            else if $C_1$ and $\neg B_2$ then
38:             $allocated.cpu = cpu\_scal$
                $allocated.mem = remain.mem \times \alpha$
40:         else
                $allocated.cpu = remain.cpu \times \alpha$
42:             $allocated.mem = remain.mem \times \alpha$
            end if
    /* (3) The remaining memory is insufficient */
44:     else if $A_1$ and $\neg A_2$ then
            if $B_1$ and $C_2$ then
46:             $allocated.cpu = task\_request.cpu$
                $allocated.mem = mem\_scal$
48:         else if $B_1$ and $\neg C_2$ then
                $allocated.cpu = task\_request.cpu$
50:             $allocated.mem = remain.mem \times \alpha$
            else if $\neg B_1$ and $C_2$ then
52:             $allocated.cpu = remain.cpu \times \alpha$
```

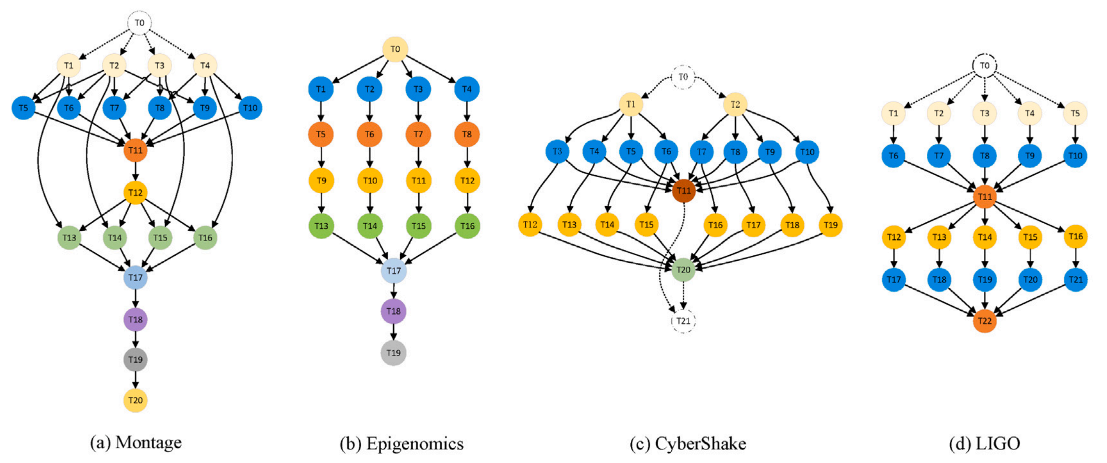


Fig. 2. The topology diagram of four scientific workflow applications.

```
                $allocated.mem = mem\_scal$
54:         else
                $allocated.cpu = remain.cpu \times \alpha$
56:             $allocated.mem = remain.mem \times \alpha$
            end if
    /* (4) The remaining resources are insufficient */
58:     else if $\neg A_1$ and $\neg A_2$ then
            if $C_1$ and $C_2$ then
60:             $allocated.cpu = cpu\_scal$
                $allocated.mem = mem\_scal$
62:         else if $C_1$ and $\neg C_2$ then
                $allocated.cpu = task\_request.cpu$
64:             $allocated.mem = remain.mem \times \alpha$
            else if $\neg C_1$ and $C_2$ then
66:             $allocated.cpu = remain.cpu \times \alpha$
                $allocated.mem = mem\_scal$
68:         else
                $allocated.cpu = remain.cpu \times \alpha$
70:             $allocated.mem = remain.mem \times \alpha$
            end if
72:     end if
        if $allocated.cpu < min_{cpu}$ or $allocated.mem < min_{mem}$ then
74:         Push $t_{k,i}$ to the next $readyTaskList$
        else
    $allocatedList[t_{k,i}] = \{allocated.cpu, allocated.mem, n\}$
76:     end if
    end for
78: return $allocatedList$
```

6. Experimental evaluation

We evaluate the performance of the proposed CWRA using various metrics and highlight its advantages by comparing its effectiveness across four specific arrival patterns with a baseline system.

6.1. Experimental setup and design

This subsection provides a concise overview of the experimental settings, covering scenarios, workflow examples, workflow instantiation, arrival patterns, evaluation metrics, and the baseline algorithm.

6.1.1. Experimental scenarios

The Kubernetes cluster deployed in our experiments is configured with one Master and five nodes. Each node is equipped with a 6-core AMD EPYC 7742 2.2 GHz CPU, 8 GB of RAM, and operates on Ubuntu 20.04 with Kubernetes v1.19.6 and Docker v18.09.6. We deployed Redis cluster version 6.2.7 across three virtual machines, each configured identically to the nodes. To evaluate the performance of the proposed CWE and compare it with the baseline, we executed four different scientific workflows and mixed workflows on this Kubernetes cluster.

6.1.2. Workflow examples

To evaluate the performance of our CWE, we utilized four scientific workflows and mixed workflows in a containerized setup on a Kubernetes cluster. These workflows include Montage (astronomy), Epigenomics (genome sequencing), CyberShake (earthquake science), and LIGO (gravitational physics). The mixed workflows consist of four distinct types of scientific workflows, each chosen with equal probability for every submission, to simulate a multi-user environment. Each workflow type was selected with equal probability for each submission, ensuring a balanced distribution across the different types. Modifications were made to the workflow structures by introducing virtual entry and exit nodes, shaping the workflows into DAG. These workflows, as illustrated in Fig. 2, were sourced from the Pegasus Workflow repository. Each workflow, containing approximately 20 tasks, was selected to demonstrate a broad spectrum of structural characteristics such as in-tree, out-tree, fork-join, and pipeline patterns, highlighting the versatility and complexity inherent in scientific workflows. Our focus was primarily on the workflow topologies rather than the specific data processing tasks, to assess the adaptability of our CWE without the influence of real-world data handling. To facilitate a fair comparison of resource allocation strategies, we standardized the tasks across all four scientific workflow categories. The tasks do not involve processing large datasets or transferring significant amounts of data between nodes. This design choice ensures that data communication costs are minimized and do not affect the resource allocation and scheduling decisions in our experiments. In our experimental setup, each workflow node simulated task execution by monitoring resource consumption (CPU and memory utilization) and service runtime, using a top-down task scheduling approach based on dependencies within the workflow DAG.

As for simulating resource loads for workflow tasks, we integrate multiple parameters alongside Stress to model scientific workflow tasks. For each task, the Stress tool is configured to initiate multiple CPU forks, allocate a memory of 500Mi, and fixed duration. The total duration of each task pod is fixed at 13 s. Then we pack the Python application along with the Stress program into a Docker image and upload it to Docker Hub repository. Regarding resource allocation in the task pod, we uniformly apply resource requests and limits at 1000 milli-cores (1000 m) for CPU and 1000Mi for memory. It is important to note that setting identical values for the requests and limits fields ensures the task pod receives the highest scheduling priority, classified as Guaranteed.

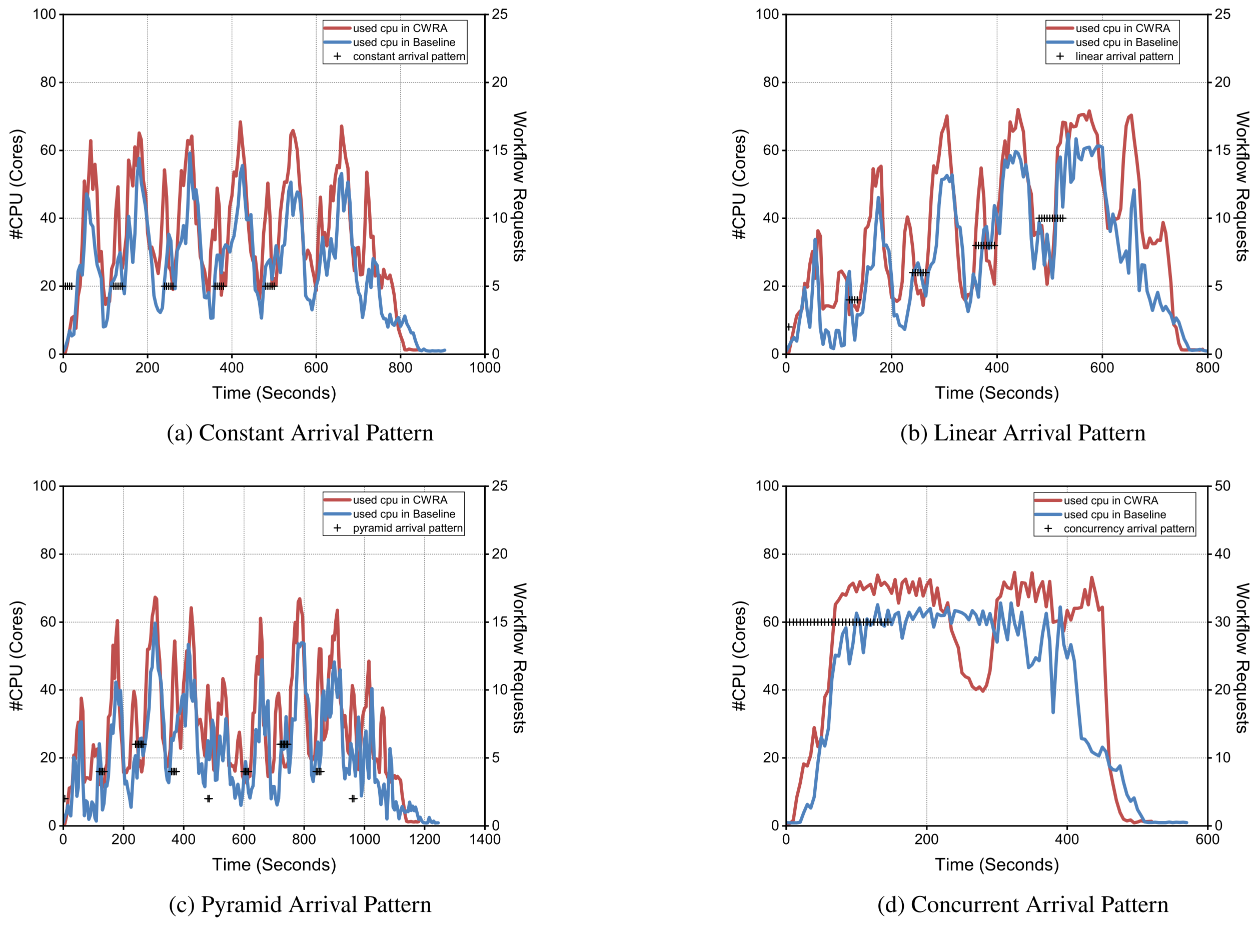


Fig. 3. The CPU resource usage rate under four distinct arrival patterns for Montage workflows.

6.1.3. *Workflow arrival patterns*

We employ four distinct workflow request patterns:

Constant Arrival Scenario: In this scenario, the workflow requests arrive in a constant manner. The Workflow Injector Module, in conjunction with the CLI, dispatches a batch of 5 workflow requests to the CWE every 120 s. This process is repeated six times, cumulatively sending 30 workflows.

Linear Arrival Scenario: In this scenario, the workflow requests follow a linearly increasing pattern. The Workflow Injector Module, in conjunction with the CLI, dispatches workflow requests as defined by the equation$y = kx + d$. Here, 'y' denotes the number of concurrent workflow requests, 'k' represents the increase rate set at 2, and 'd' signifies the initial value, also set at 2. The frequency of workflow requests escalates at a rate of 'k' every 120 s. Fig. 3(b) illustrates this linear increase pattern, depicting the process repeated five times to achieve a total of 30 workflows.

Pyramid Arrival Scenario: In this scenario, workflow requests are directed to the CWE, following a pyramid-like pattern. Initially, the system handles a small volume of concurrent workflow requests, starting at 2. This number incrementally increases to a larger, predetermined figure (6 for each workflow type) as illustrated in Fig. 3(c). The frequency of concurrent workflow requests escalates by 2 every 120 s until a peak is achieved. Upon reaching the peak, the number is reverted to the initial count in a similar fashion, and this cycle repeats until the aggregate number of workflow requests attains the predetermined total of 34.

Concurrent Arrival Scenario: In this scenario, the system is challenged with a high initial load where 30 workflow requests are simultaneously injected at the start. Unlike the other scenarios, there are no subsequent increases in the number of workflows; the system instead focuses on processing this initial batch without additional inputs. This scenario is critical to test the CWE's ability to handle a sudden surge in demand and to assess its immediate response capabilities. Fig. 3(d) illustrates this scenario, capturing the initial burst of activity and the CWE's ability to manage such an influx effectively.

The deployment of these four scenarios is designed to comprehensively accommodate the dynamic resource demands and unexpected surges in workflow requests encountered in a production environment. Although the Constant and Linear Arrival scenarios exhibit a degree of predictability, the Pyramid scenario introduces an element of unpredictability with its variable arrival pattern.

6.1.4. *Evaluation metrics*

To evaluate the effectiveness of our CWE, we conduct a comprehensive analysis using the following quantitative metrics:

Total Workflows Duration: This metric represents the average duration of all workflows. It is defined as the time span from the receipt of the first workflow request to the completion of the last request.

Average Workflow Duration: This metric calculates the average duration of each workflow, measuring the time from the start of the first task to the completion of the last task.

CPU Usage: The CPU usage indicates the average utilization of CPU throughout all workflows in the Kubernetes cluster. Higher resource utilization indicates better alignment with our optimization objectives.

When analyzing CPU usage, we also observed memory usage. The results show that the memory utilization curves closely overlap with the CPU utilization curves, indicating similar usage patterns. Given that cluster nodes have sufficient memory resources, we believe that

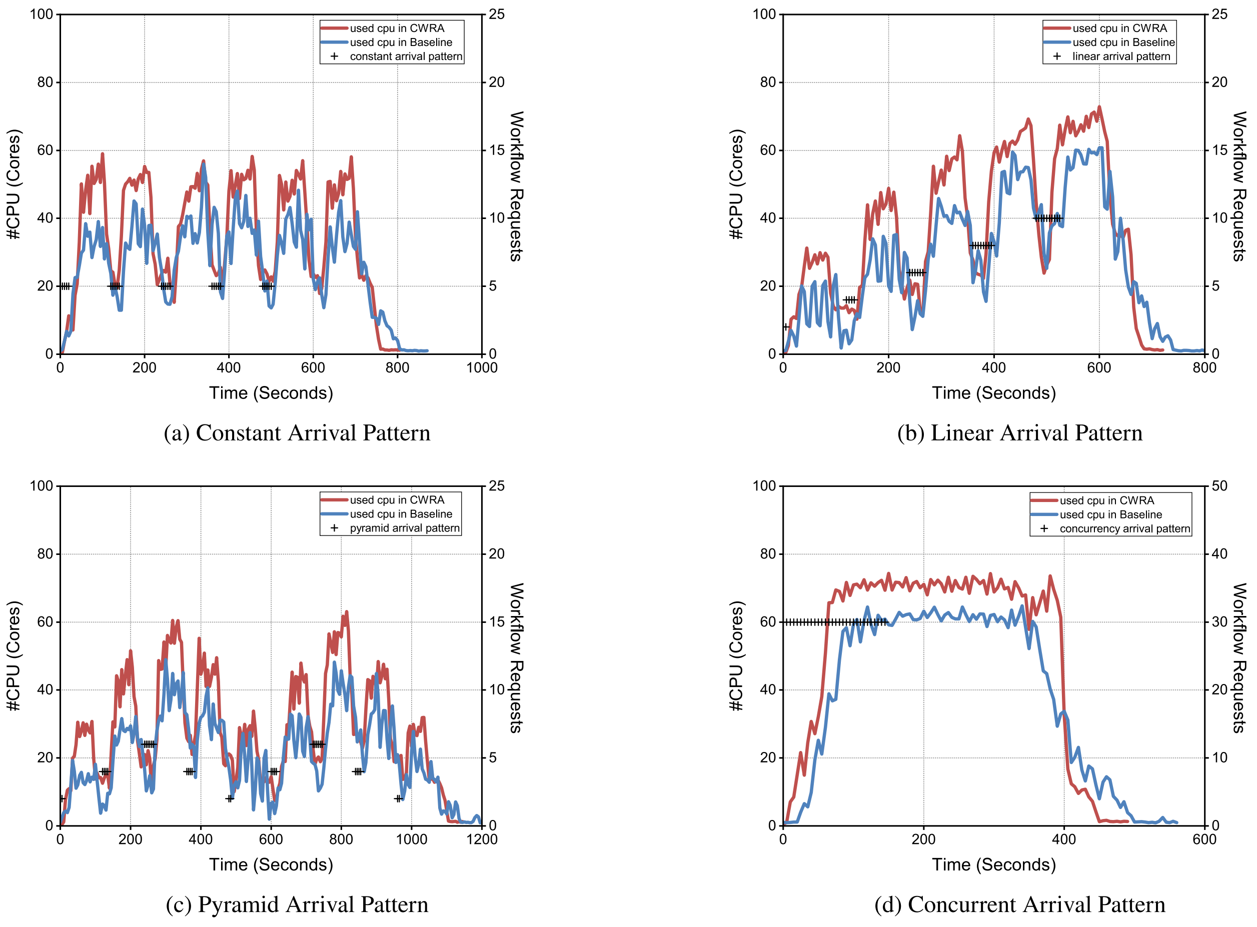


Fig. 4. The CPU resource usage rate under four distinct arrival patterns for Epigenomics workflows.

memory limitations do not significantly affect system performance in most cases. Therefore, we chose to present only the CPU utilization image and did not further analyze the memory utilization in detail.

6.1.5. Baseline

In our experiments, we selected the industry-recognized Argo Workflow as a baseline for comparison with our proposed CWE. Argo Workflow is an open-source, cloud-native workflow engine that schedules parallel tasks on Kubernetes. It is widely used in automated deployment, machine learning model training, and batch processing tasks. We deployed Argo Workflow on a Kubernetes cluster using its official YAML configuration file and submitted identical workflow tasks to both Argo Workflow and our CWE for evaluation.

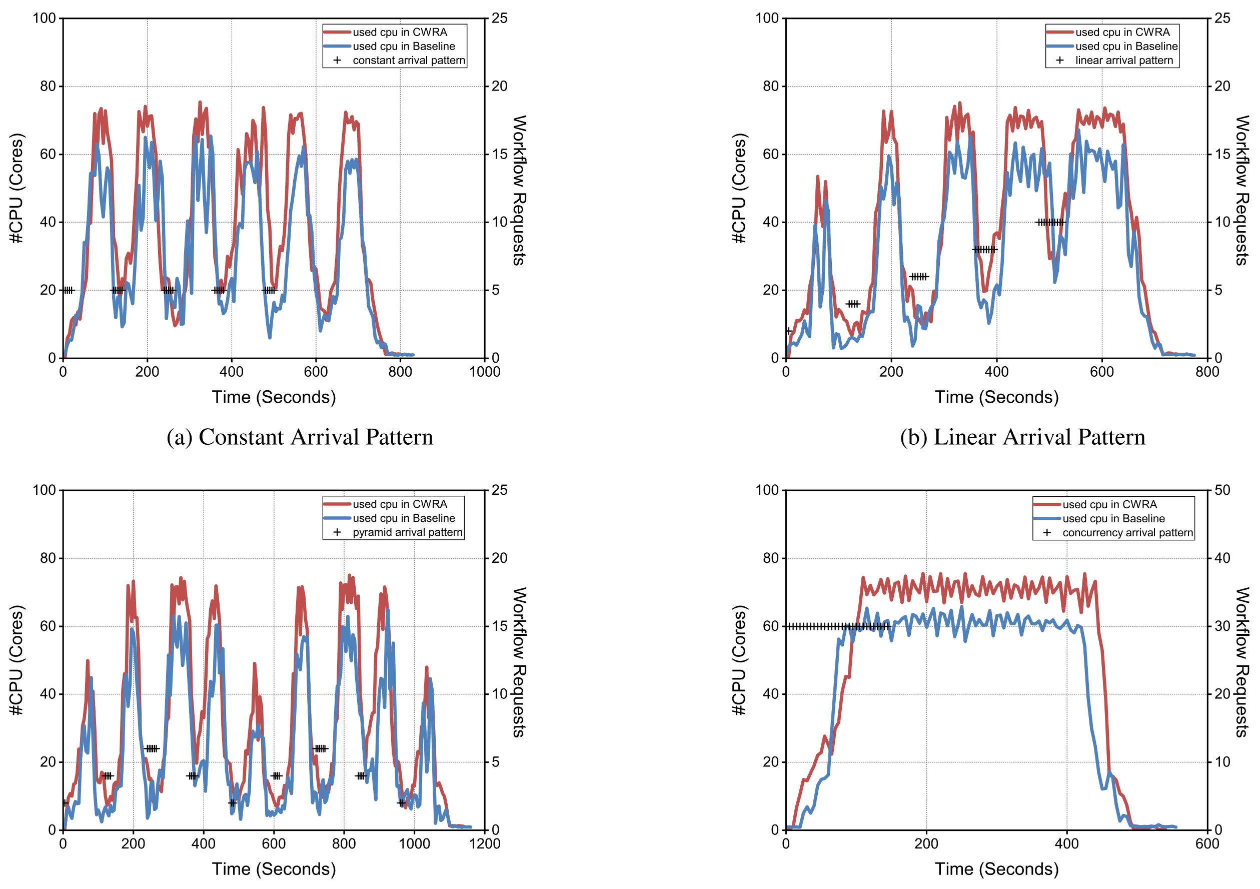
6.2. Results and analysis

In order to fully evaluate CWE and CWRA, we conducted an overall performance and reliability assessment, and we discussed the key findings in detail. To minimize external effects such as network latency and resource contention, our Kubernetes cluster was run without any other application load. Four different workflows and mixed workflows were evaluated ten times for each arrival type to ensure the reliability and consistency of the results.

6.2.1. General evaluation

Below, we describe our experiment in which we ran four different scientific workflows ten times, each with different arrival patterns, using both CWE and our CWRA algorithm, along with a baseline for comparison. We evaluated our CWRA against this baseline and calculated the average values for all metrics. Table 2 presents the average values obtained from the assessment runs. The term CWRA refers to our containerized workflow resource allocation, while 'Baseline' refers to the use of Argo. For the first four arrival patterns, the interval between of workflow requests is set at 120 s. The fourth pattern is concurrent and, therefore, has no interval. The number of workflows injected in each of the four patterns is 30, 30, 34, and 30, respectively. Overall, across various workflow arrival patterns and types, our CWRA consistently outperforms the baseline in most metrics observed.

Montage: The experimental setup of this study involved a small Montage workflow comprising 21 tasks (Fig. 2(a)). CWRA was found to save total time across various arrival modes compared to the baseline method. Specifically, it saved 7.1% in constant arrival, 4.7% in linear arrival, 5.5% in pyramid arrival, and 7.1% in concurrent arrival modes, as detailed in Table 2. CWRA achieved a reduction in average workflow duration of 11.7% under constant arrival, a 2.2% increase under linear arrival, a 10.0% reduction under pyramid arrival, and a 14.1% increase under concurrent arrival. Fig. 3 demonstrates the consistency of these results with the total duration of all injected workflows. Analysis of the results showed that in scenarios with linear and concurrent arrivals, CWRA outperformed the baseline in total duration but not in average duration. The baseline's first-come, first-served scheduling strategy was effective in reducing average durations under high concurrency but performed poorly in terms of total duration. Consequently, although CWRA was less effective in managing average durations for high-concurrency tasks, its performance in total duration was notably better. CWRA efficiently handled multiple tasks arriving simultaneously, thereby reducing overall processing time. Considering CWE's objective to schedule large-scale workflows across multiple clus-



(a) Constant Arrival Pattern

(b) Linear Arrival Pattern

(c) Pyramid Arrival Pattern

(d) Concurrent Arrival Pattern

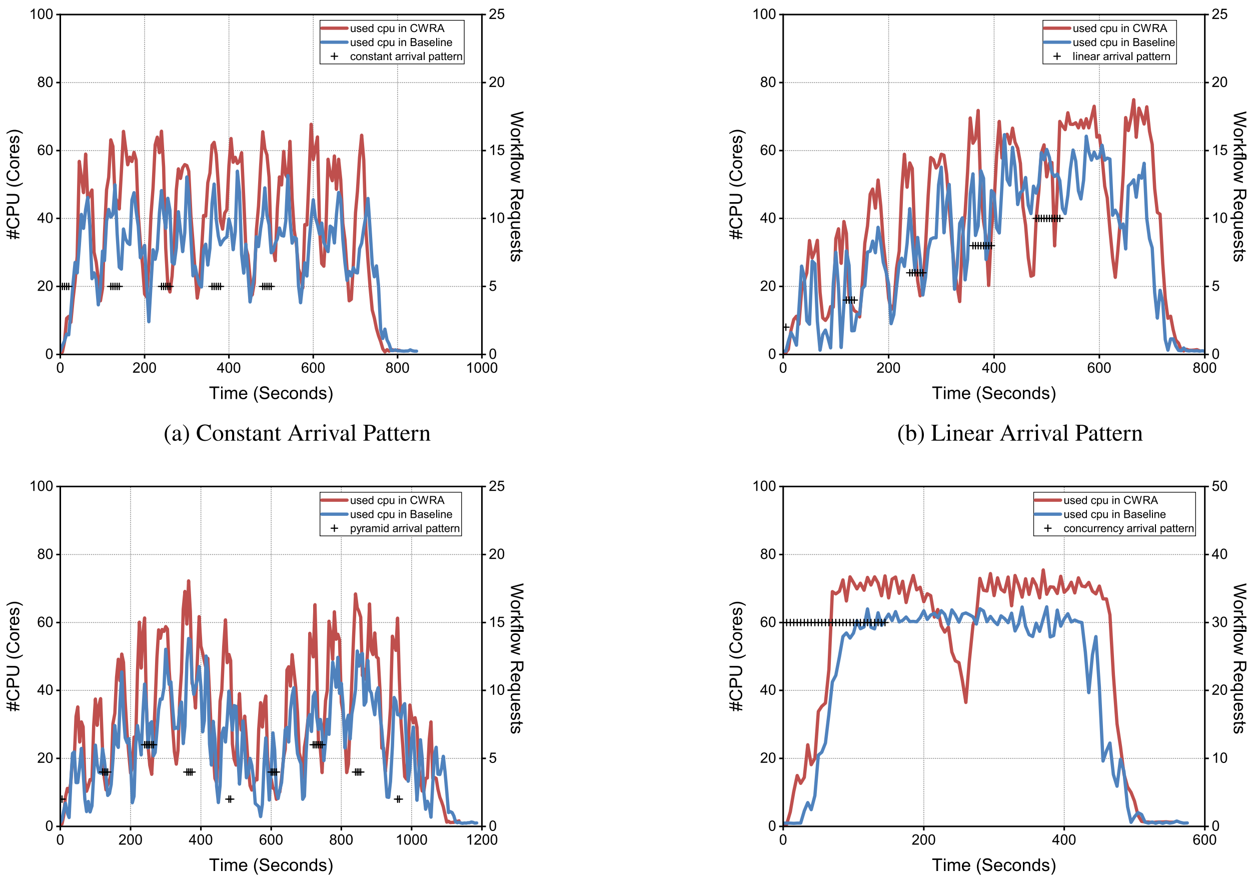
Fig. 5. The CPU resource usage rate under four distinct arrival patterns for CyberShake workflows.

Table 2
Evaluation results.

Workflow types	Metrics	Constant arrival		Linear arrival		Pyramid arrival		Concurrency arrival	
		CWRA	Baseline	CWRA	Baseline	CWRA	Baseline	CWRA	Baseline
	Number of workflow requests	30		30		34		30	
	Interval between two requests bursts	120		120		120		0	
Montage	Total workflows duration in seconds	785.2	844.8	728.4	764.6	1119.5	1184.7	460.3	495.7
	Average workflow duration in seconds	209.7	237.6	243.8	238.6	200.2	222.6	446.7	391.5
	Cpu resource usage	36.54	26.76	38.18	29.23	30.40	21.87	56.09	45.17
Epigenomics	Total workflows duration in seconds	742.8	808.3	660.2	742.9	1084.8	1141.7	430.1	485.7
	Average workflow duration in seconds	149.6	187.5	166.3	195.0	141.5	180.3	392.3	355.2
	Cpu resource usage	37.07	28.09	39.93	28.91	30.12	21.64	56.92	43.39
CyberShake	Total workflows duration in seconds	739.6	771.8	687.7	715.4	1080.6	1101.9	486.8	480.6
	Average workflow duration in seconds	141.6	152.6	170.5	170.6	131.8	148	455.6	375.3
	Cpu resource usage	41.34	32.10	41.63	32.50	32.36	24.34	55.97	47.96
LIGO	Total workflows duration in seconds	752.8	785.5	741.5	747.9	1095.3	1125.3	503.9	495.1
	Average workflow duration in seconds	159.4	162.1	201.2	186.3	150.4	160.7	466.6	364.6
	Cpu resource usage	41.42	31.11	41.63	33.99	33.38	24.99	56.80	48.27
Mixed workflows	Total workflows duration in seconds	740.7	815.8	685.8	739.5	1077.2	1177.2	449.8	528.7
	Average workflow duration in seconds	142.8	167.3	175.7	180.8	160.3	191.3	397.1	387.5
	Cpu resource usage	33.16	26.92	33.50	29.83	27.73	21.69	53.62	41.13

ters, CWRA's design is particularly well-suited for scenarios involving a significant number of concurrent tasks.

As shown in Fig. 3, our CWRA system CPU usage consistently outperforms the baseline method across all request arrival scenarios. In the constant arrival mode, CWRA achieves a maximum CPU usage of 68.40%, which is 9.13% higher than the baseline's 59.27%. In the linear arrival mode, CWRA reaches 71.99%, surpassing the baseline's 64.82% by 7.17%. The CPU usage for CWRA in the pyramid arrival mode peaks at 67.34%, 7.66% higher than the baseline's 59.68%. Similarly, in the concurrent arrival mode, CWRA records a maximum CPU usage of 74.60%, 8.95% greater than the baseline's 65.65%. In the constant arrival mode, CPU usage remains moderate due to the steady frequency of requests. In the linear arrival mode, as requests increase linearly over time, CPU usage progressively rises, peaking at



(a) Constant Arrival Pattern

(b) Linear Arrival Pattern

(c) Pyramid Arrival Pattern

(d) Concurrent Arrival Pattern

Fig. 6. The CPU resource usage rate under four distinct arrival patterns for Ligo workflows.

the end of the period. In the pyramid arrival mode, request volume initially rises and then declines, leading to a CPU usage pattern that peaks midway, resulting in a more balanced overall distribution. In the concurrent arrival mode, a large influx of simultaneous requests causes a sharp spike in CPU usage over a short duration.

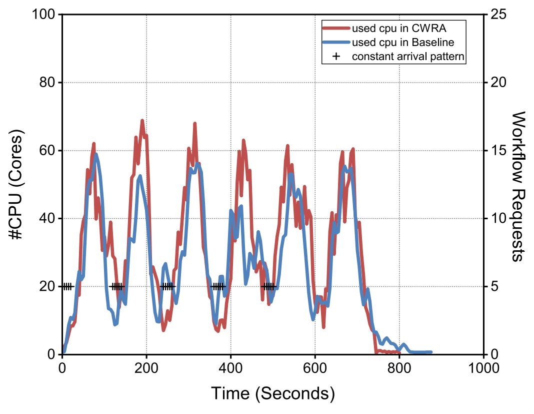
Epigenomics: The experimental setup consisted of a 20-task Epigenomics workflow (Fig. 2(b)), comparing its total duration under CWRA and a baseline across different arrival modes. CWRA achieved time savings of 8.1%, 11.1%, 5.0%, and 16.4% in constant, linear, pyramid, and concurrent arrival modes, respectively. In terms of average workflow duration, CWRA reduced time by 20.2%, 14.7%, and 21.5% in constant, linear, and pyramid modes, but showed a 5.9% increase in concurrent mode. Further analysis shows that the Epigenomics workflow outperforms Montage in all test scenarios for total and individual durations. This is particularly evident in the first three arrival modes. This advantage arises from the linear and pipelined topology of the Epigenomics workflow, which is well-suited for high concurrency scenarios. CWRA shows greater efficiency and time savings than the baseline in managing continuous workflow requests.

Fig. 4 presents a comparison of CPU resource usage between the CWRA algorithm and a baseline across various arrival patterns. Under constant arrival mode, CWRA achieves 59.02% CPU usage, slightly higher than the baseline by 2.07%. In linear, pyramid, and concurrent modes, CWRA's CPU usage is 72.87%, 63.11%, and 74.32%, respectively, outperforming the baseline by 12.12%, 14.17%, and 9.54%. The higher resource utilization in these modes is due to peak workloads, where more than ten large-task workflows are processed simultaneously, leading to elevated CPU demands. These results illustrate CWRA's improved efficiency during peak resource usage compared to the baseline algorithm.

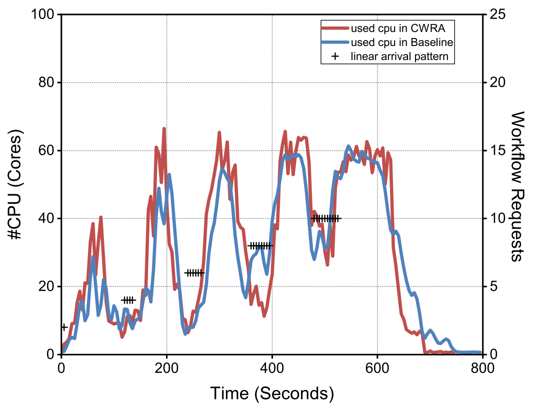
CyberShake: The experimental setup involved a small CyberShake workflow with 22 tasks (Fig. 2(c)). CWRA achieved time savings of 4.2%, 3.9%, and 2.0% in constant, linear, and pyramid arrival modes, respectively, compared to the baseline, though it was 1.3% slower in concurrent mode. The average workflow duration in constant arrival mode was reduced by 7.2%, while the pyramid mode 10.9% reduction. Conversely, the concurrent mode resulted in a 16.7% increase in duration, with no significant change in the linear mode.

Fig. 5 illustrates the CPU resource usage of the CWRA algorithm compared to the baseline under different arrival scenarios. CWRA consistently maintained around 75% CPU usage across all modes, exceeding the baseline by approximately 10%. The CyberShake workflow's shallow depth and wide topology, along with its high parallelism, enable the CWRA algorithm to demonstrate superior resource utilization efficiency across various arrival patterns.

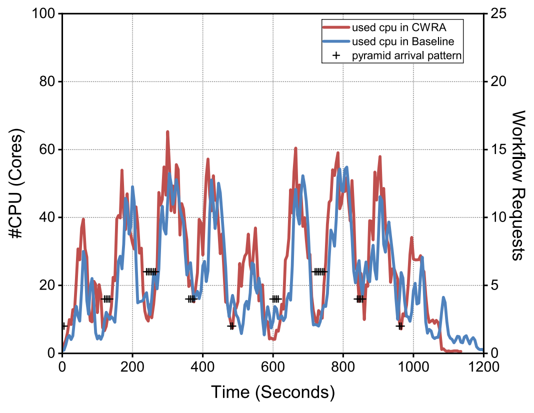
LIGO: The experimental setup included a small-scale LIGO workflow comprising 23 tasks, as shown in Fig. 2(d). Compared to the baseline, CWRA reduced the total workflow duration by 4.2%, 0.9%, and 2.7% in constant, linear, and pyramid arrival modes, respectively, but increased by 0.16% in concurrent mode. For average workflow durations, CWRA showed a 1.7% reduction in constant mode, a 6.4% reduction in pyramid mode, but increased by 8.0% in linear mode and



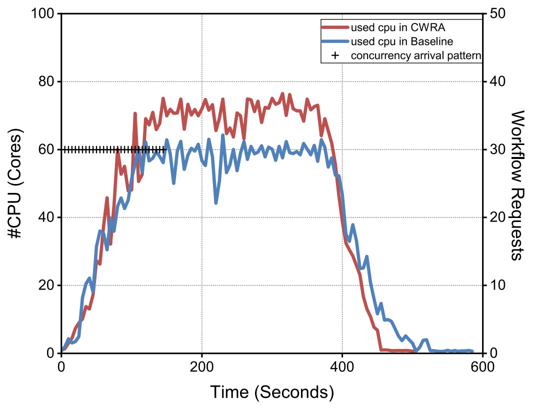
(a) Constant Arrival Pattern



(b) Linear Arrival Pattern



(c) Pyramid Arrival Pattern



(d) Concurrent Arrival Pattern

Fig. 7. The CPU resource usage rate under four distinct arrival patterns for mixed workflows.

19.4% in concurrent mode.

Fig. 6 presents the CPU utilization results, showing that CWRA consistently achieved higher CPU usage across all arrival patterns. Under constant, linear, pyramid, and concurrent arrival patterns, CWRA's maximum CPU usage was 67.71%, 74.96%, 72.26%, and 75.47%, respectively, exceeding the baseline by 13.78%, 10.27%, 16.95%, and 10.88%. These findings indicate that CWRA consistently demonstrates higher CPU utilization, ensuring high efficiency in task processing.

Mixed workflows: The experimental setup for the mixed workflows involved the submission of workflows that were randomly selected from four types (Montage, Epigenomics, CyberShake, and LIGO) with an equal probability for each submission. This approach simulates a multi-user environment, reflecting real-world scenarios where diverse scientific applications are processed concurrently. Under different arrival patterns, CWRA demonstrated improvements over the baseline. Specifically, CWRA achieved total time savings of 10.1% in the constant arrival pattern, 7.8% in the linear arrival pattern, 9.3% in the pyramid arrival pattern, and 17.5% in the concurrent arrival pattern. For average workflow durations, CWRA reduced times by 17.2% under constant arrival, 2.9% under linear arrival, and 19.3% under pyramid arrival patterns. However, in the concurrent arrival pattern, the average workflow duration increased by 2.5%.

Regarding CPU resource utilization, CWRA consistently demonstrated higher CPU usage rates across all arrival patterns compared to the baseline, as shown in Fig. 7. For the constant arrival pattern, CWRA reached a peak CPU usage of 68.8%, 9.9% higher than the baseline's 58.9%. Under the linear arrival pattern, CWRA's CPU usage peaked at 66.5%, 5.2% above the baseline. For the pyramid arrival pattern, CWRA achieved a CPU usage of 65.3%, 10.5% higher than the baseline. Under the concurrent arrival pattern, CWRA's CPU usage peaked at 76.5%, 12.2% higher than the baseline. These results highlight CWRA's adaptability and scalability across diverse scenarios, demonstrating its ability to maintain optimal performance under mixed workflows conditions.

7. Conclusion

In this paper, we present the CWRA approach, specifically designed for containerized workflow management engines. Leveraging the innovative architecture of CWE and its integration with Kubernetes, our CWRA approach maximizes resource utilization through dynamic scaling method, effectively addressing complex and evolving workflow demands. Experimental evaluations demonstrate that CWRA outperforms baseline algorithms in key performance metrics, including resource utilization, total workflow duration, and average task duration, across various workflow arrival patterns and categories (Montage, Epigenomics, CyberShake, LIGO, and Mixed workflows). In future work, we intend to further analyze and refine various resource allocation algorithms within the CWE framework to optimize the balance between resource usage efficiency and workflow performance. Specifically, we will explore deep reinforcement learning methods for resource allocation in cloud workflows to achieve intelligent and automated resource management. Additionally, we will explore extending CWRA capabilities in distributed environments to support larger-scale and more complex workflow management.

CRediT authorship contribution statement

Danyang Liu: Writing – original draft, Software, Methodology, Investigation, Conceptualization. **Yuanqing Xia:** Writing – review & editing, Supervision, Methodology. **Chenggang Shan:** Writing – review & editing, Data curation. **Ke Tian:** Writing – review & editing. **Yufeng Zhan:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant No. 61836001).

Data availability

Data will be made available on request.

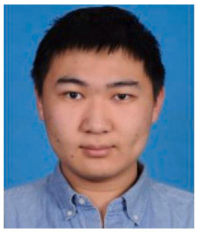
References

[1] D. Merkel, et al., Docker: lightweight linux containers for consistent development and deployment, Linux j 239 (2) (2014) 2.
[2] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, J. Wilkes, Borg, omega, and Kubernetes, Commun. ACM 59 (5) (2016) 50–57.
[3] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, et al., Pegasus: A framework for mapping complex scientific workflows onto distributed systems, Sci. Program. 13 (3) (2005) 219–237.
[4] B. Howe, Virtual appliances, cloud computing, and reproducible research, Comput. Sci. Eng. 14 (4) (2012) 36–41.
[5] K. Liu, K. Aida, S. Yokoyama, Y. Masatani, Flexible container-based computing platform on cloud for scientific workflows, in: 2016 International Conference on Cloud Computing Research and Innovations, ICCCRI, IEEE, 2016, pp. 56–63.
[6] Volcano, Cloud native batch scheduling system for compute-intensive workloads. [online], 2024, Available: https://volcano.sh/.
[7] Kubeflow, The machine learning toolkit for Kubernetes. [online], 2024, Available: https://www.kubeflow.org/.
[8] Argo, The workflow engine for Kubernetes. [online], 2024, Available: https://github.com/argoproj/argo-workflows.
[9] M. Malawski, G. Juve, E. Deelman, J. Nabrzyski, Algorithms for cost-and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds, Future Gener. Comput. Syst. 48 (2015) 1–18.
[10] R. Prodan, T. Fahringer, Dynamic scheduling of scientific workflow applications on the grid: a case study, in: Proceedings of the 2005 ACM Symposium on Applied Computing, 2005, pp. 687–694.
[11] Z. Ahmad, B. Nazir, A. Umer, A fault-tolerant workflow management system with quality-of-service-aware scheduling for scientific workflows in cloud computing, Int. J. Commun. Syst. 34 (1) (2021) e4649.
[12] M. Mao, M. Humphrey, Auto-scaling to minimize cost and meet application deadlines in cloud workflows, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011, pp. 1–12.
[13] R.N. Calheiros, R. Ranjan, A. Beloglazov, C.A. De Rose, R. Buyya, CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, Softw. - Pract. Exp. 41 (1) (2011) 23–50.
[14] M. Malawski, K. Figiela, M. Bubak, E. Deelman, J. Nabrzyski, Scheduling multilevel deadline-constrained scientific workflows on clouds based on cost optimization, Sci. Program. 2015 (2015) 5–5.
[15] A. Alsarhan, A. Itradat, A.Y. Al-Dubai, A.Y. Zomaya, G. Min, Adaptive resource allocation and provisioning in multi-service cloud environments, IEEE Trans. Parallel Distrib. Syst. 29 (1) (2017) 31–42.
[16] Q. Zhang, L. Cheng, R. Boutaba, Cloud computing: state-of-the-art and research challenges, J. Internet Serv. Appl. 1 (2010) 7–18.
[17] J. Xu, J.A. Fortes, Multi-objective virtual machine placement in virtualized data center environments, in: 2010 IEEE/ACM Int'L Conference on Green Computing and Communications & Int'L Conference on Cyber, Physical and Social Computing, IEEE, 2010, pp. 179–188.
[18] D. Liu, Y. Xia, C. Shan, G. Wang, Y. Wang, Scheduling containerized workflow in multi-cluster Kubernetes, in: CCF Conference on Big Data, Springer, 2023, pp. 149–163.
[19] C. Shan, Y. Xia, Y. Zhan, J. Zhang, KubeAdaptor: a docking framework for workflow containerization on Kubernetes, Future Gener. Comput. Syst. 148 (2023) 584–599.
[20] C. Shan, C. Wu, Y. Xia, Z. Guo, D. Liu, J. Zhang, Adaptive resource allocation for workflow containerization on Kubernetes, J. Syst. Eng. Electron. 34 (3) (2023) 723–743.
[21] S. Islam, J. Keung, K. Lee, A. Liu, Empirical prediction models for adaptive resource provisioning in the cloud, Future Gener. Comput. Syst. 28 (1) (2012) 155–162.
[22] P. Hoenisch, S. Schulte, S. Dustdar, S. Venugopal, Self-adaptive resource allocation for elastic process execution, in: 2013 IEEE Sixth International Conference on Cloud Computing, IEEE, 2013, pp. 220–227.
[23] Y.C. Lee, H. Han, A.Y. Zomaya, On resource efficiency of workflow schedules, Procedia Comput. Sci. 29 (2014) 534–545.
[24] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P.J. Maechling, R. Mayani, W. Chen, R.F. Da Silva, M. Livny, et al., Pegasus, a workflow management system for science automation, Future Gener. Comput. Syst. 46 (2015) 17–35.
[25] X. Xu, H. Yu, X. Pei, A novel resource scheduling approach in container based clouds, in: 2014 IEEE 17th International Conference on Computational Science and Engineering, IEEE, 2014, pp. 257–264.
[26] L. Yin, J. Luo, H. Luo, Tasks scheduling and resource allocation in fog computing based on containers for smart manufacturing, IEEE Trans. Ind. Inform. 14 (10) (2018) 4712–4721.
[27] B. Tan, H. Ma, Y. Mei, A NSGA-II-based approach for multi-objective micro-service allocation in container-based clouds, in: 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID, IEEE, 2020, pp. 282–289.
[28] Y. Hu, H. Zhou, C. de Laat, Z. Zhao, Concurrent container scheduling on heterogeneous clusters with multi-resource constraints, Future Gener. Comput. Syst. 102 (2020) 562–573.
[29] V. Medel, O. Rana, J.Á. Bañares, U. Arronategui, Modelling performance & resource management in Kubernetes, in: Proceedings of the 9th International Conference on Utility and Cloud Computing, 2016, pp. 257–262.
[30] C.-C. Chang, S.-R. Yang, E.-H. Yeh, P. Lin, J.-Y. Jeng, A Kubernetes-based monitoring platform for dynamic cloud resource provisioning, in: GLOBECOM 2017-2017 IEEE Global Communications Conference, IEEE, 2017, pp. 1–6.
[31] G. Rattihalli, M. Govindaraju, H. Lu, D. Tiwari, Exploring potential for non-disruptive vertical auto scaling and resource estimation in Kubernetes, in: 2019 IEEE 12th International Conference on Cloud Computing, CLOUD, IEEE, 2019, pp. 33–40.
[32] E. Kim, K. Lee, C. Yoo, On the resource management of Kubernetes, in: 2021 International Conference on Information Networking, ICOIN, IEEE, 2021, pp. 154–158.
[33] Z. Ding, S. Wang, C. Jiang, Kubernetes-oriented microservice placement with dynamic resource allocation, IEEE Trans. Cloud Comput. (2022).
[34] G.C. Sih, E.A. Lee, A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures, IEEE Trans. Parallel Distrib. Syst. 4 (2) (1993) 175–187.
[35] H. Topcuoglu, S. Hariri, M.-Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distrib. Syst. 13 (3) (2002) 260–274.
[36] R. Sakellariou, H. Zhao, A hybrid heuristic for DAG scheduling on heterogeneous systems, in: 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings, IEEE, 2004, p. 111.
[37] L.F. Bittencourt, R. Sakellariou, E.R. Madeira, Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm, in: 2010 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing, IEEE, 2010, pp. 27–34.
[38] C.-C. Hsu, K.-C. Huang, F.-J. Wang, Online scheduling of workflow applications in grid environments, Future Gener. Comput. Syst. 27 (6) (2011) 860–870.
[39] H. Arabnejad, J. Barbosa, Fairness resource sharing for dynamic workflow scheduling on heterogeneous systems, in: 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, IEEE, 2012, pp. 633–639.
[40] L. Yang, L. Ye, Y. Xia, Y. Zhan, Look-ahead workflow scheduling with width changing trend in clouds, Future Gener. Comput. Syst. 139 (2023) 139–150.
[41] L. Ye, Y. Xia, S. Tao, C. Yan, R. Gao, Y. Zhan, Reliability-aware and energy-efficient workflow scheduling in iaas clouds, IEEE Trans. Autom. Sci. Eng. 20 (3) (2022) 2156–2169.

Danyang Liu received the M.S. degree in mathematics from HeBei University of Science and Technology in 2020. He is currently working toward the Ph.D. degree in the School of Automation, Beijing Institute of Technology. His current research interests include cloud computing and Data Center Network.

Yuanqing Xia (Fellow, IEEE) received his M.S. degree in Fundamental Mathematics from Anhui University, China, in 1998 and his Ph.D. degree in Control Theory and Control Engineering from Beijing University of Aeronautics and Astronautics, Beijing, China, in 2001. During January 2002-November 2003, he was a Postdoctoral Research Associate with the Institute of Systems Science, Academy of Mathematics and System Sciences, Chinese Academy of Sciences, Beijing, China. From November 2003 to February 2004, he was with the National University of Singapore as a Research Fellow, where he worked on variable structure control. From February 2004 to February 2006, he was with the University of Glamorgan, Pontypridd, U.K., as a Research Fellow. From February 2007 to June 2008, he was a Guest Professor with Innsbruck Medical University, Innsbruck, Austria. Since 2004, he has been with the Department of Automatic Control, Beijing Institute of Technology, Beijing, first as an Associate Professor, then, since 2008, as a Professor. His current research interests are in the fields of cloud control systems, networked control systems, robust control and signal processing, active disturbance rejection control, unmanned system control, and flight control.

Chenggang Shan received the M.S. degree in computer applied technology from Qiqihr University, China, in 2007. He received Ph.D. degree with the School of Automation, Beijing Institute of Technology, Beijing, China, in 2023. He was an associate professor with the School of Artificial Intelligence, Zaozhuang University, China, in 2017. His research interests include networked control systems, cloud computing, cloud–edge collaboration, wireless networks.



Ke Tian received the M.S. degree from George Washington University in 2021. He is currently working toward the Ph.D. degree in the School of Automation, Beijing Institute of Technology. His current research interests include artificial intelligence and medical image recognition.

Yufeng Zhan received his Ph.D. degree from Beijing Institute of Technology, Beijing, China, in 2018. He is currently an assistant professor in the School of Automation with BIT. Prior to join BIT, he was a post-doctoral fellow in the Department of Computing with The Hong Kong Polytechnic University. His research interests include networking systems, game theory, and machine learning.