Journal of Systems Engineering and Electronics
Vol. 34, No. 3, June 2023, pp.723 – 743

Adaptive resource allocation for workflow containerization on Kubernetes

SHAN Chenggang[1,2], WU Chuge[1], XIA Yuanqing[1,*], GUO Zehua[1], LIU Danyang[1], and ZHANG Jinhui[1]

1. School of Automation, Beijing Institute of Technology, Beijing 100081, China;
2. School of Artificial Intelligence, Zaozhuang University, Zaozhuang 277100, China

Abstract: In a cloud-native era, the Kubernetes-based workflow engine enables workflow containerized execution through the inherent abilities of Kubernetes. However, when encountering continuous workflow requests and unexpected resource request spikes, the engine is limited to the current workflow load information for resource allocation, which lacks the agility and predictability of resource allocation, resulting in over and under-provisioning resources. This mechanism seriously hinders workflow execution efficiency and leads to high resource waste. To overcome these drawbacks, we propose an adaptive resource allocation scheme named adaptive resource allocation scheme (ARAS) for the Kubernetes-based workflow engines. Considering potential future workflow task requests within the current task pod’s lifecycle, the ARAS uses a resource scaling strategy to allocate resources in response to high-concurrency workflow scenarios. The ARAS offers resource discovery, resource evaluation, and allocation functionalities and serves as a key component for our tailored workflow engine (KubeAdaptor). By integrating the ARAS into KubeAdaptor for workflow containerized execution, we demonstrate the practical abilities of KubeAdaptor and the advantages of our ARAS. Compared with the baseline algorithm, experimental evaluation under three distinct workflow arrival patterns shows that ARAS gains time-saving of 9.8% to 40.92% in the average total duration of all workflows, time-saving of 26.4% to 79.86% in the average duration of individual workflow, and an increase of 1% to 16% in centrol processing unit (CPU) and memory resource usage rate.

Keywords: resource allocation, workflow containerization, Kubernetes, workflow management engine.

DOI: 10.23919/JSEE.2023.000073

1. Introduction

With the advent of a cloud-native era, the most popular virtualization solution is using Docker for container encapsulation with Kubernetes (K8s) [1] for multi-host container orchestrating. Docker and K8s have become mainstream tools for cloud resource management and dominated the whole cloud-native technology ecosystem [2]. Workflows have been widely applied in scientific computing communities such as astronomy, bioinformatics, material science, and earth science [3]. A scientific workflow is commonly formulated as a directed acyclic graph (DAG), which consists of dozens of workflow tasks (represented by nodes) and dependencies among tasks (indicated by directed edges). A DAG abstracts a particular scientific computing process through shared data files between tasks and predefined task dependencies [4,5]. Powered by Docker and K8s, cloud infrastructure features the scalability and high availability of computational resources[6] and is especially suitable as a running platform for scientific workflows.

Scientific workflows usually serve large-scale applications and require a considerable amount of resources to execute. Efficient resource allocation is a key issue in workflow execution. Existing workflow management engines like Nextflow [7], Pegasus [8,9], Galaxy [10], and Argo workflow engine can execute hundreds of workflows on cloud infrastructure and be responsible for assigning computational resources to workflow tasks [11]. When encountering continuous workflow requests and unexpected resource request spikes, the computational resource requirements of workflows can be highly dynamic. The ever-changing resource requirements of workflows bring a great administrative burden to the workflow engines for resource allocation and seriously decrease the execution efficiency of workflows. First, the permanent provision of fixed computational resources will cope with peak loads in a resource-intensive scenario but incur high costs and resource over-provisioning, as resources are not fully utilized during off-peak times.

Manuscript received September 23, 2022.
*Corresponding author.
This work was supported by the National Natural Science Foundation of China (61873030, 62002019).

Second, some workflows may not be executed at all and suffer from a poor quality of service (QoS) due to insufficient resource provisions.

In order to avoid over and under-provisioning of resources, some existing works propose reasoning [12,13], feedback [14], heuristics [15], learning and prediction models [16−19] to cope with resource allocation in cloud environment. Although these solutions can partially address the cloud resource allocation problem, they commonly use prior knowledge of cloud systems to cope with resource allocation. As a result, these solutions might play to their strengths in a specific application scenario, but they are not fully adaptable to the K8s-based cloud environment with dynamic resource requirements. In addition, numerous iteration training may result in high computational complexity and resource overheads in learning and prediction models. Therefore, with the application platform and technology stack in mind, they do not fit with the K8s-based workflow management engines. The bottleneck here is the absence of a high-efficiency adaptive resource allocation scheme that can help the K8s-based workflow management engines to make appropriate resource provisions in response to continuous workflow requests and unexpected request resource spikes.

In [20,21], the customized K8s-based workflow management engine KubeAdaptor was proposed, which was able to integrate workflow systems with K8s and implement workflow containerization on K8s cluster. In this paper, we present an adaptive resource allocation scheme (ARAS) that follows the monitor-analyse-plan-execute knowledge (MAPE-K) model [22,23]. The ARAS periodically responds to the task pod's resource request and uses the resource discovery algorithm, resource evaluation algorithm, and resource allocation algorithm to complete the resource allocation of this round task by the resource scaling strategy. We reconstruct and extend KubeAdaptor, and implement ARAS as the resource manager component of KubeAdaptor, which consists of a resource discovery module, resource evaluator module, and allocator module. Three modules complement each other to achieve the adaptive resource allocation. First, the resource discovery module invokes the resource discovery algorithm to obtain the remaining resources (such as CPU and memory) of K8s cluster nodes and the resource usage of running task pods. Then the resource evaluator module integrates the remaining resources of the K8s cluster and workflow workloads from the Redis database and evaluates resource adequacy for the K8s cluster nodes. Finally, the allocator module uses a resource scaling strategy (i.e., vertical autoscaling) [24] to make resource provisions for current active task pods in response to continuous workflow requests and sudden request spikes. We have open-sourced the proposed ARAS. The source code is publicly available on GitHub [25].

This paper focuses on adaptation, that is, the adaptive adjustment of resource allocation in the context of changing workflow resource requirements. Compared with the baseline algorithm, experimental evaluation of running four scientific workflows under three different workflow arrival modes shows that ARAS gains time-saving of 9.8% to 40.92% in the average total duration of all workflows, time-saving of 26.4% to 79.86% in the average duration of individual workflow, and an increase of 1% to 16% in CPU and memory resource usage rate. The main contributions of this paper are summarized as follows.

(i) MAPE-K architecture. With the MAPE-K mechanism as a core, we decouple and reconstruct the KubeAdaptor and integrate our ARAS into four phases of the MAPE-K model to equip the KubeAdaptor with self-healing and self-configuration abilities.

(ii) A novel monitoring mechanism. We devise and develop a resource discovery algorithm through the K8s resource characteristics and informer component. The resource discovery module uses this algorithm to build a novel monitoring mechanism to collect all related data in K8s clusters.

(iii) Automation deployment. We modularize and implement the four steps of the proposed ARAS with loose coupling in mind so that the users can easily mount a newly designed algorithm module to replace an existing one with minimal intrusion into the workflow management engine.

(iv) Better performance. With the help of K8s and the MAPE-K mechanism, we use ARAS to conduct a wealth of experiments on four scientific workflows on K8s clusters. Our ARAS shows better performance compared to the baseline algorithm.

The rest of the paper is organized as follows. Section 2 introduces related work. Section 3 elaborates on the system model and problem formulation, while Section 4 further describes our system architecture and components. Section 5 illustrates the implementation of our adaptive resource allocation scheme. Section 6 describes the experimental setup and discusses the evaluation results. Finally, Section 7 concludes this paper.

2. Related work

The resource allocation scheme in the workflow management engine is influenced by the virtualization technology of cloud infrastructure, which is directly related to whether workflow tasks are hosted by virtual machine (VM) instances or containers. In this section, we review the development of resource allocation strategy and discuss three categories of resource allocation strategy from the perspective of the evolution of virtualization technology, namely VM-based, container-based, and cloud-native-based. Note that the analysis of each aspect is not completely limited to the scope of the workflow management engine.

2.1 VM-based resource allocation

In the VM-based era, Lee et al. [26] proposed an adaptive scheduling approach to adjust resource allocation and scheduling in the Pegasus workflow management system. This approach utilized batch queues to assign jobs to cluster's VMs and optimizes job scheduling across cluster's VMs through the average queue time of each available VM. Islam et al. [27] developed prediction-based resource measurement and provisioning strategies using neural networks and linear regression to satisfy upcoming resource demands. The sliding window approach for predicting resource usage in the cloud fits with dynamic and proactive resource management for interactive e-commerce applications. As for business process management system (BPMS) field, Hoenisch et al. [12] presented a self-adaptive resource allocation approach to automatically lease and release cloud resources for workflow executions based on knowledge (resource usage in VMs) about current and future process landscape. This approach has been implemented as part of ViePEP, a BPMS able to manage and schedule workflows in the cloud. By monitoring resource usage of VMs and the QoS of individual service invocations in VMs, ViePEP uses a prediction model to provide resource provisioning for elastic process execution of workflows. Subsequently, Hoenisch et al. [13] extended ViePEP by dynamic workflow scheduling and resource allocation algorithms. The proposed algorithm not only provided a complete schedule plan based on their former predicting model but also moved the service invocations (workflow task) from one timeslot to another to fully utilize the acquired resources.

Although these solutions present appropriate cloud resource allocation schemes to some extent, the predictive models commonly require the collection and modeling according to former data. These upfront preparations consume unnecessary resources and block the automatic operation flow of the workflow management engines. In addition, VMs-based resource allocation schemes are commonly limited to VM's features such as slow startup, clumsy deployment, and high resource consumption. Therefore, these schemes are not suitable for performing workflows with dynamic and ever-changing resource requirements in cloud infrastructures.

2.2 Container-based resource allocation

In the container era, container-based resource allocation schemes gradually become the mainstream of cloud resource management. Considering the absolute resource isolation and security features of VMs, most container-based resource allocation scenarios adopt the deployed model of VM hosting containers. For instance, Mao et al. [28] proposed a differentiated quality of experience scheduler to adjust resource provisioning for deep learning applications. This scheduler is implemented into Docker Swarm and can accept the targeted quality of experience specifications from clients and dynamically adjust resource limits of containers to approach performance targets. Abdullah et al. [16] introduced a new deep learning-based approach to estimate the execution time of the jobs through the collected performance traces. This approach also predicts the execution time for different CPU pins and uses the laws of diminishing marginal returns to provide optimal CPU allocation to Docker containers. In the fog computing community, Yin et al. [29] proposed a container-based task-scheduling algorithm with task delay constraint in mind. Herein, a resource reallocation mechanism works to achieve resource-utilization maximization on fog nodes by modifying the resource quota of task containers. Hu et al. [30] proposed containerised edge computing (CEC), a CEC framework for dynamic resource provisioning in response to multiple intelligent applications. The CEC first makes resource provisioning for containers in advance based on the workload prediction for the edge cluster formed by Docker Swarm and then uses the idea of control theory to achieve dynamic resource adjustments (meaning a sufficient number of containers) for hosted service applications.

Containers, an efficient and lightweight virtualization technology, bring significant technology change to VMs-based resource allocation strategies. However, it needs a container orchestration tool (e.g., Docker Swarm) to manage a wealth of containers across cluster nodes in several scenarios. In practice, resource limits' adjustment and reallocation of resource quotas in running containers have brought significant administrative burdens to Docker Swarm. Also, the adjustments to the number of containers will cause a delay in the startup of new containers. In addition, the preparation for predictive models is also not conducive to the automation of workflow management engines. Due to the shortcomings of the above solutions and the task dependencies and high concurrency of workflow, these resource allocation strategies cannot provide efficient ideas for container-based workflow management engines.

2.3 Cloud-native-based resource allocation

As the first hosted project by cloud native computing foundation (CNCF), K8s has become the defacto standard container orchestration system. Docker and K8s are reshaping resource management strategies for cloud infrastructures in the cloud-native era. For example, Chang et al. [31] proposed a generic platform to facilitate dynamic resource provisioning based on K8s. The platform employs open-source tools to retrieve all the resource utilization metrics (such as CPU and memory) while integrating the application QoS metrics into monitoring. The resource scheduler module in the platform makes dynamic resource provisioning by horizontal scaling of task pods according to the K8s cluster's workload. Mao et al. [32] investigated the performance of using cloud-native frameworks (Docker and K8s) for big data and deep learning applications from the perspective of resource management fields. Together with Prometheus and Grafana, the authors build a container monitoring system to keep tracking the resource usage of each job on worker nodes. To address massive aggregate resource wastage, Google uses Autopilot to configure resources automatically, adjusting both the number of concurrent tasks in a job (horizontal scaling) and the CPU/memory limits for individual tasks (vertical scaling) [24]. Subsequently, Bader et al. [11] proposed Tarema, a system for allocating task instances to heterogeneous K8s cluster resources during the execution of scalable scientific workflows. Using a scoring algorithm to determine the best match between a task and the available resources, Tarema provides the near-optimal task-resource allocation.

However, most of these resource allocation solutions in the cloud-native era use open source tools (from the CNCF community) to build resource monitoring systems, obtain the required resource utilization of the cluster, and provide corresponding resource provisioning strategies. It brings high deployment costs to the workflow management engine, which is inconsistent with the simple deployment and automatic system characteristics. In addition, these tools put too much pressure on the K8s cluster because of frequent access to kube-apiserver for acquiring cluster resources [33].

To summarize, we can conclude that resource allocation policies change with the evolution of virtualization technologies to adapt to different application scenarios and technology platforms. The most typical example is ViePEP-C [34], which evolved from former work [12,13,35,36] in the VMs era to a container-based resilient BPMS platform in the container era, using containers instead of VMs for the execution of business process activities. Considering automation and flexible deployment of the integrated platform, resource allocation technology in the cloud-native era is more focused on Docker and K8s platforms. The design of our workflow management engine follows this idea. K8s, with its unique technical advantages and ecology in scheduling, automatic recovery, horizontal scalability, resource monitoring, and other aspects, makes its integration with workflow management engines far beyond the capabilities of container-based workflow management engines. Inspired by the work in [24,28,32], the proposed ARAS takes into account workflow loads in K8s clusters and uses vertical scaling technology of containers to cope with continuous workflow requests and sudden resource spikes.

3. System model

This section describes how to use the proposed ARAS to cope with continuous workflow requests and unexpected resource request spikes and maximize resource utilization while meeting workflow service level objectives (SLOs).

3.1 System description

For the clarity of presentation, we consider the scenario of a single K8s cluster with a set of nodes (VMs), donated by $V=\{v_1,v_2,\cdots,v_m\}$, where m represents the number of K8s cluster nodes. As for m nodes, we have a set of available CPU cores $C=\{c_1,c_2,\cdots,c_m\}$ and a set of availble memory capacity $M=\{\mathrm{mem}_1,\mathrm{mem}_2,\cdots,\mathrm{mem}_m\}$ correspondingly. The workflow set injected into the KubeAdaptor is represented as $W=\{w_1,w_2,\cdots,w_k\}$, where k indicates the amount of workflows. Herein, a workflow is abstractly defined as $w_i=\{\mathrm{sla}_{w_i},s_{i,1},s_{i,2},\cdots,s_{i,n}\}$, wherein i indicates the identification (ID) number of a workflow, sla_{w_i} represents a service level agreement (SLA) of a workflow and $s_{i,1},s_{i,2},\cdots,s_{i,n}$ indicates steps (i.e., tasks) of workflow w_i. Each workflow task is defined as

$$s_{i,j}=\{\mathrm{sla}_{s_{i,j}},\mathrm{id},\mathrm{image},\mathrm{cpu},\mathrm{mem},\mathrm{duration},\\ \mathrm{min}_{\mathrm{cpu}},\mathrm{min}_{\mathrm{mem}}\},\ 1\leqslant i\leqslant k;\ 1\leqslant j\leqslant n \tag{1}$$

where "id" is the unique identifier of this workflow task in workflow w_i, and "image" represents the Docker image address of this workflow task. The "cpu" is the amount of CPU Milli cores required by the users, and "mem" is the amount of memory capacity required by the users. "duration" indicates the duration of the task pod running, and "$\mathrm{min}_{\mathrm{cpu}}$" and "$\mathrm{min}_{\mathrm{mem}}$" represent a minimum of CPU and memory resources required to run the task container of $s_{i,j}$ in workflow w_i, respectively. Generally, a workflow can have an optional SLA (sla_i) composed of several SLOs expressed by $\mathrm{slo}_1,\mathrm{slo}_2,\cdots,\mathrm{slo}_n$ on workflow (sla_{w_i}) or workflow task ($\text{sla}_{s_{i,j}}$) as follows:

$$\text{sla}_i = \{\text{slo}_1, \text{slo}_2, \cdots, \text{slo}_n\},\ \ i \in \{w_i, s_{i,j}\}. \tag{2}$$

Herein, we only consider the deadline as the single SLO, meaning that each task in the workflow must be completed before its respective deadlines. Likewise, this workflow is no exception:

$$\begin{cases} \text{sla}_{w_i} = \text{deadline}_{w_i} \\ \text{sla}_{s_{i,j}} = \text{deadline}_{s_{i,j}} \end{cases}. \tag{3}$$

Note that the deadline for the last task $s_{i,\text{last}}$ in a workflow is identical to this workflow execution deadline:

$$\text{deadline}_{s_{i,\text{last}}} = \text{deadline}_{w_i}. \tag{4}$$

3.2 Problem formulation

We assume that SLAs and deadlines defined by users are valid and achievable, i.e., a properly completed workflow means that all of its tasks must be completed by the deadline. With maximizing the resource utilization in the K8s cluster as a goal, as long as a task request arrives, our ARAS uses the resource scaling method to provide computational resources to the task container. Here, the resource provision of the task container must not be less than a minimum of running resources to ensure the smooth operation of the task container.

Fig. 1 depicts an execution process of a small-scale Montage workflow.

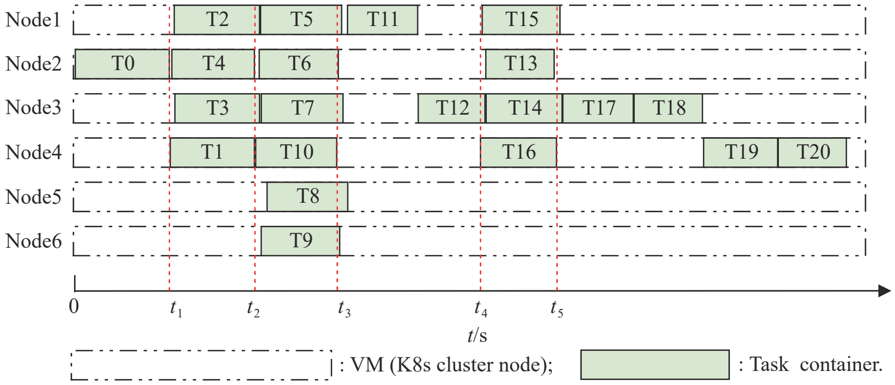


Fig. 1 Resource allocation example

At t_1 seconds, task request of T_1 arrives, and our ARAS has abilities to acquire concurrent tasks within its lifecycle (from t_1 to t_2) considering predefined deadlines. As can seen from Fig. 1, T_2, T_3, and T_4 will be launched within T_1's lifecycle and four workflow tasks will compete for computing resources each other. To ensure that four concurrent tasks have enough resources to run smoothly, our ARAS employs the resource scaling method to reasonably allocate resources, i.e., scaling down resource requirements of the current task T_1 according to the ratio of the total resource requirements of four tasks to the remaining resources in the K8s cluster (refers to (9)). Similarly, T_{10} executes between t_2 and t_3, T_{16} executes between t_4 and t_5. Their respective lifecycle all contains several concurrent tasks. The arrival of each task request also requires the resource scaling method to allocate resources in line with (9).

In the following, we elaborate on the optimal problem in our ARAS. The allocated CPU and memory resources for each requested task in workflow w_i are respectively defined as follows:

$$U = \{u_{i,1}, u_{i,2}, \cdots, u_{i,n}\}, \tag{5}$$

$$R = \{r_{i,1}, r_{i,2}, \cdots, r_{i,n}\}. \tag{6}$$

$x^i_{y,z} \in \{0,1\}$ with workflow identifier i is adopted as a decision variable for task placement, where $1 \leqslant y \leqslant n$ and $1 \leqslant z \leqslant m$ and defined as

$$x^i_{y,z} = \begin{cases} 1, & y\text{th task in } w_i \text{ is scheduled on node } v_z \\ 0, & y\text{th task in } w_i \text{ is not scheduled on node } v_z \end{cases}.$$

We assume that each node in the K8s cluster is always active and that workflows are continuously injected into our workflow management engine. $\text{Mem}_{\text{total}}$ indicates the total number of remaining memory resources of the K8s cluster. Since CPU is a compressible resource and memory is an incompressible resource, we only consider memory to maximize resource allocation in the optimization model. So our objective function is as follows:

$$\max \frac{\sum_{i=1}^{k} \sum_{j=1}^{n} r_{i,j}}{\text{Mem}_{\text{total}}}$$

$$\text{s.t.} \begin{cases} \sum_{z=1}^{m} x^i_{y,z} = 1 \\ \sum_{i=1}^{k} \sum_{y=1}^{n} x^i_{y,j} \cdot u_{i,y} \leqslant c_j \\ \sum_{i=1}^{k} \sum_{y=1}^{n} x^i_{y,j} \cdot r_{i,y} \leqslant \text{mem}_j \end{cases}. \tag{7}$$

Equation (7) maximizes the resource utilization for the remaining memory of the K8s cluster at each moment of the task request. The first constraint in (7) indicates that a task can be scheduled on only one cluster node. The last two constraints in (7) imply that the total CPU and memory resources consumed by all task pods on the hosted node v_j must be less than or equal to the amount of the respective available resources on that node.

4. Architecture

This section presents the system architecture of KubeAdaptor in detail, including its framework, design logic, and key modules. Subsequently, the MAPE-K model is elaborated around the resource allocation mechanism of KubeAdaptor.

4.1 KubeAdaptor framework

The KubeAdaptor for the ARAS is illustrated in Fig. 2.

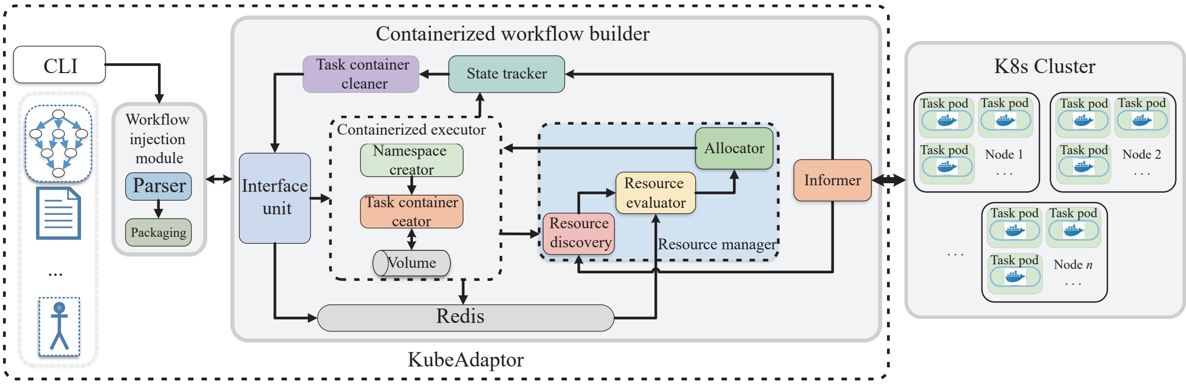


Fig. 2 KubeAdaptor architecture

As a workflow management engine, it works to administrate, schedule, and execute containerized workflow tasks. Its core functionalities are as follows:

(i) Provide an interface to the public or private cloud, allowing to customize workflows on demand.

(ii) Implement the containerized execution of workflows following the precedence and dependency relationships.

(iii) Adaptively allocate resource quotas for requested workflow tasks and maximize resource utilization when ensuring the SLAs of workflows.

(iv) Provide the ability of flexible deployment and the automatic operation flow and integrate into the K8s platform.

With the assistance of ARAS in this paper, KubeAdaptor equips with the functionalities of a cloud resource management system to elegantly manage a potentially highly volatile cloud workflow application scenario.

4.2 KubeAdaptor modules

As depicted in Fig. 2, KubeAdaptor consists of three main top level entities, including a command line interface (CLI), a workflow injection module and a containerized workflow builder. The containerized workflow builder comprises seven sub-components responsible for workflow reception, containerization, resource allocation, resource monitoring, and task container cleanup. We focus on the resource manager module related to ARAS.

CLI: it aims to define SLAs-based workflows and offer configuration files of one-key deployment. In addition, the users may request many workflows consecutively or even simultaneously through the CLI module.

Workflow injection module: its Parser and Packaging modules serve as an independent function pod and work to read variable configuration information of workflow definition from the mounted directory, parse and encapsulate workflows in response to generating request of the subsequent workflow from the Interface uint.

Interface uint: this module works on receiving the workflow generating request, decomposing the workflow tasks, watching the state changes of task pods or workflows from the task container cleaner, invoking the containerized executor to generate workflow namespaces and task pods, and writing workflow status into the Redis database. Once the creation of the task pod fails, this module turns to fault tolerance management [21], also known as self-healing, the ability of a system to detect and recover from potential problems and continue to operate smoothly.

Containerized executor: its two subcomponents work on generating workflow namespaces and task pods. This module creates task pods through allocated resources from the resource manager. In addition, the states of workflows and task pods are timely written into the Redis database.

Resource manager: it contains three subcomponents, such as resource discovery, resource evaluator and allocator. The resource discovery is responsible for acquiring the remaining resource of the overall K8s cluster from the Informer. The resource evaluator obtains workflow resource requirements and workflow execution states from the Redis database, assesses the adequacy of the current remaining resources of the K8s cluster, and launches corresponding countermeasures, if necessary. The allocator module uses resource scaling strategy to make resource allocation for current active task pods in response to continuous workflow requests and sudden request spikes. It is also known as self-configuration, the ability of a system to reconfigure itself under changing and unpredictable circumstances.

Informer: as a core toolkit in Client go, the informer is in charge of synchronizing resource objects and events between K8s core components and Informer local cache. It provides the resource discovery with the remaining resources of the K8s cluster and responds to the state tracker for the state changes of the resource objects.

State tracker: it hosts the monitoring program based on the List-Watch mechanism and responds state queries of various resource objects to each module anytime.

Task container cleaner: it works on deleting the pods in a state of succeeded or failed or OOMKilled and workflow namespaces without uncompleted task pods. Once receiving successful feedback on the just-deleted workflow or task pods, this module proceeds to Interface unit and triggers the following workflow or subsequent task.

Redis: the Redis database is to be deployed within or outside the cluster in advance and is responsible for storing workflow execution status and predefined resource requirements of workflow tasks.

KubeAdaptor is implemented by the Go language and provides an CLI interface to K8s clusters. With just a few tweaks to the configuration file, the users can go out of the box and smoothly deploy the KubeAdaptor on K8s clusters. The deployment and uninstalling of KubeAdaptor are non-intrusive and clean to the cluster, and its workflow containerization execution works in an automated way. Further details about KubeAdaptor can be referred to [21].

4.3 MAPE-K model

The MAPE-K model [37], originated from the field of automatic computing, is an instrumental framework for systematic development of adaptive systems, including resource allocation and workflow adaptation. Adaptive strategy within the KubeAdaptor works to realize the self-optimization of resource utilization maximization. Herein, the self-optimization ability, along with self-healing and self-configuration (elaborated in Subsection 4.2), enables our KubeAdaptor to become a self-management system.

To deal with persistent workflow requests and ever-changing resource requirements, we use the MAPE-K model to retrofit with minimal intervention to the KubeAdaptor, which forms an adaptive execution cycle as depicted in Fig. 3.

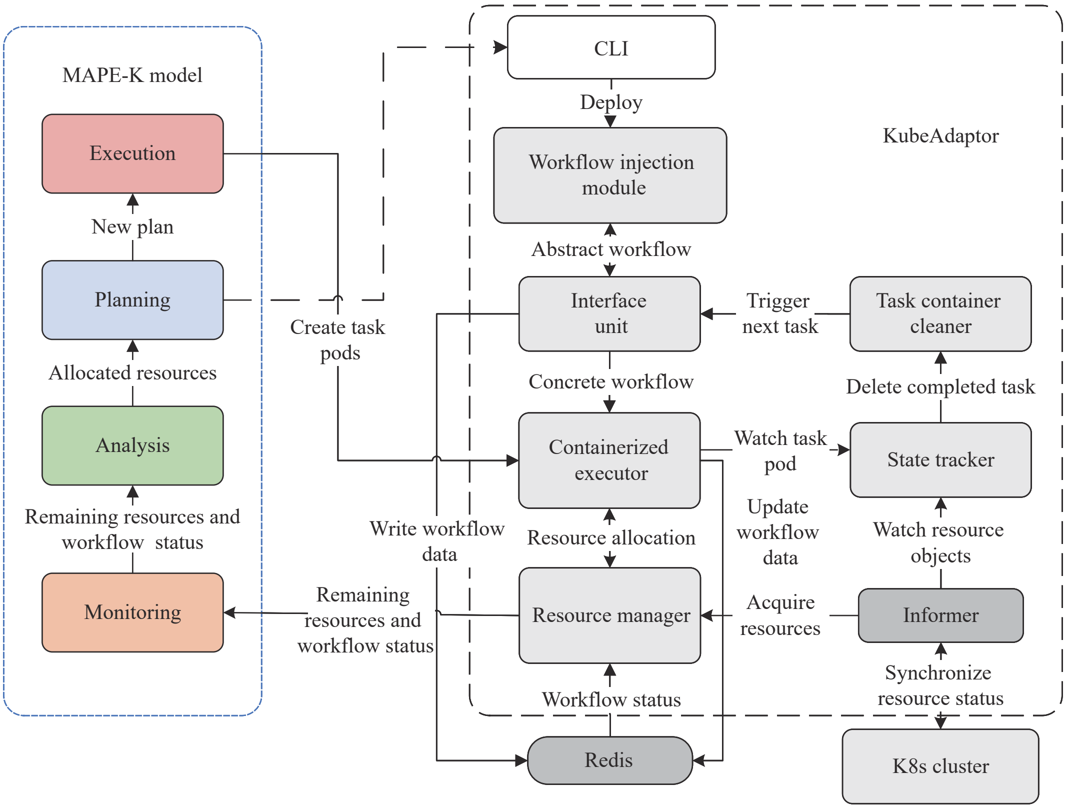


Fig. 3 Resource allocation scheme based on MAPE-K model

In the following, we will briefly discuss the four steps of this cycle and how they influence the self-management capabilities of KubeAdaptor.

Monitoring: the monitoring functionalities stem from the Informer and Redis database and work to provide workflow status and the remaining resources in K8s clusters to the next step. Workflow status data includes which workflows have been executed, which tasks have been completed, resource requirements of workflow tasks, as well as the SLOs of the workflow and task. The remaining resources refer to the residual CPU and memory resources of the K8s cluster and each node in the K8s cluster.

Analysis: the functionality of this step comes into play in the resource evaluator and allocator within resource manager. For adaptive resource allocation, it is necessary to analyze the monitored data and reason on the general knowledge about the system. It is done so that we can adapt according to countermeasures to deal with dynamic workflow requests and SLA violations.

Planning: the planning step takes full account of the resource requirements for future workflow tasks to be launched within the current task lifecycle and SLAs of workflows to carry out reasoning and generate a resource allocation plan. The planning results also provide sufficient prior knowledge for subsequent workflow input to the CLI module.

Execute: the execution steps are put into practice through containerized executor, aiming to finish the creation of a new round of task pods combined with the analysis results of the MAPE-K model.

Knowledge base: while not really a part of the cycle, the knowledge base stores the configuration information about the system and provides workflow execution status and the remaining resources in K8s clusters to MAPE-K model running.

In short, the KubeAdaptor needs to provide further analysis of the resource allocation scheme to self-optimize application scenarios in response to persistent workflow requests and sudden resource spikes.

5. Adaptive resource allocation scheme

Once resource manager receives a resource request of workflow task from containerized executor, its three subcomponents immediately launch resource discovery, resource evaluation, and resource allocation in turn. The entire execution process responds to the workflow task’s resource request iteratively. Next, we introduce the adaptive resource allocation algorithm, resource discovery algorithm, and resource evaluation algorithm. All notations used in our algorithm are illustrated in Table 1.

Table 1 Major notations used in adaptive resource allocation scheme

Notation	Meaning
v_i	A K8s node (VM), $v_i \in \mathcal{V}$
$s_{i,j}$	The jth task in workflow w_i $(1 \leqslant i \leqslant k,\ 1 \leqslant j \leqslant n)$
$\text{allocated}_{\text{cpu}}$	The allocated CPU resource number through the adaptive resource allocation algorithm
$\text{allocated}_{\text{mem}}$	The allocated memory resource number through the adaptive resource allocation algorithm
request.cpu	The accumulated CPU resource number for many task requests
request.mem	The accumulated memory resource number for many task requests
$\text{Re}_{\text{max}}^{\text{cpu}}$	The maximum remaining resource amount of CPU among K8s cluster nodes
$\text{Re}_{\text{max}}^{\text{mem}}$	The maximum remaining resource amount of memory among K8s cluster nodes
totalResidual.cpu	The total of residual CPU resource across K8s cluster nodes
totalResidual.mem	The total of residual memory resource across K8s cluster nodes
task^{req}	The current task request with respects to $s_{i,j}$
$\text{task}_{i,j}^{\text{redis}}$	A record of task-state data from Redis database
PodLister	An interface of acquiring pod list in Informer component
NodeLister	An interface of acquiring node list in Informer component
ResidualMap	A data dictionary for storing remaining resources (CPU and memory) of each node
nodeReq.cpu	The accumulated CPU resource requirements for all pods on a node
nodeReq.mem	The accumulated memory resource requirements for all pods on a node
allocatable.cpu	The accumulated allocatable CPU resource across K8s cluster nodes
allocatable.mem	The accumulated allocatable memory resource across K8s cluster nodes
residual.cpu	The residual CPU resource in K8s cluster
residual.mem	The residual memory resource in K8s cluster
pod_i	A data struct from Podlist, which contains many key fields about container’s features
cpu_{cut}	The allocated CPU resource amount for task request based on (9)
mem_{cut}	The allocated memory resource amount for task request based on (9)
α	A proportional value derived from experience, $\alpha \in (0,1)$
β	A constant value derived from experience, $\beta \geqslant 20$

In addition, workflow execution states are represented as a set of state data for all tasks of workflow w_i ($1 \leqslant i \leqslant k$) and a record of task-state data is defined as

$$\text{task}_{i,j}^{\text{redis}} = \{t_{\text{start}}, \text{duration}, t_{\text{end}}, \text{cpu}, \text{mem}, \text{flag}\},$$
$$1 \leqslant i \leqslant k;\ 1 \leqslant j \leqslant n;\ \text{flag} \in \{\text{false}, \text{true}\}. \qquad (8)$$

Note that as long as KubeAdaptor starts, $\text{task}_{i,j}^{\text{redis}}$ is stored in Redis database through Interface unit, and then is continuously updated by the containerized executor. Herein, t_{start} is the start time of the current task pod in K8s cluster, duration is similar to the definition of $s_{i,j}$.duration and represents the duration of the current task pod's running, t_{end} is the completed time of the current task in the K8s cluster, cpu and mem are equivalent to the definition of $s_{i,j}$.cpu and $s_{i,j}$.mem, and flag is a boolean variable that identifies the execution status of the current task pod. Herein, the boolean variable false indicates that the current task is not complete. We use Dictionary data structure Map < $\text{task}_{i,j}$.id, $\text{task}_{i,j}^{\text{redis}}$ > to indicate state data of the current task, and $\text{task}_{i,j}$.id is the unique identifier passed by task $s_{i,j}$ of workflow w_i (refer to (1)).

Because the pod is the minimum execution unit of the K8s container orchestrator, coupled with KubeAdaptor's non-invasive automated execution process, Resource manager allocates resources only once throughout the requested task pod's lifecycle in response to task pod's resource request. The users can initially set min_{cpu} and min_{mem} of the task pod in CLI module, by which this task pod ensures its hosted container runs smoothly.

5.1 Adaptive resource allocation algorithm

Algorithm 1 presents an adaptive resource allocation algorithm. It initializes these parameters to be 0 in line 1 and takes workflow task $s_{i,j}$ as input. Once the containerized executor sends a task pod's resource request, the Adaptive resource allocation algorithm performs the following process. Lines 4−13 work to access the Redis database and get the total of requested resources of all task pods to be launched within $s_{i,j}$'s lifecycle. These task pods have resource competition with the currently requested task pod. Subsequently, Algorithm 1 uses Resource discovery algorithm to obtain the remaining resources about the K8s cluster and each node in the K8s cluster. Lines 16−23 traverse the remaining resource structure ResidualMap and accumulate to get the total remaining resources of all nodes across the K8s cluster. Meanwhile, the proposed algorithm obtains the maximal remaining CPU and memory resources. Here, we assume that one node with the maximal remaining CPU resources also has the maximal remaining memory to facilitate the conditional comparison in Resource evaluation algorithm (prioritize CPU resource for allocation). Then, the proposed algorithm calls Resource evaluation algorithm to present the allocated resources (line 25).

Algorithm 1 Adaptive resource allocation algorithm

```
Input: $s_{i,j}$;
Output: $\text{allocated}_{\text{cpu}}$, $\text{allocated}_{\text{mem}}$;
1: Initialization: request.cpu, request.mem, $\text{Re}_{\text{max}}^{\text{cpu}}$, $\text{Re}_{\text{max}}^{\text{mem}}$,
   totalResidual.cpu, totalResidual.mem $\leftarrow 0$;
2: for each task pod's resource request do
3:   /*Access the Redis and get the total of requested
     resources of all pods to be launched within $s_{i,j}$'s lifecycle*/
4: Get $\text{task}^{\text{req}}$ with respects to $s_{i,j}$ of $w_i$ from Redis;
5: request.cpu $\leftarrow$ $\text{task}^{\text{req}}$.cpu;
6: request.mem $\leftarrow$ $\text{task}^{\text{req}}$.mem;
7: Get all $\text{task}_{i,j}^{\text{redis}}$ for all workflows from Redis;
8: for each task $\in \{\text{task}_{i,j}^{\text{redis}}\}$ do
9:   if task.$t_{\text{start}} \in [\text{task}^{\text{req}}.t_{\text{start}},\ \text{task}^{\text{req}}.t_{\text{end}})$ then
10:      request.cpu += task.cpu;
11:      request.mem += task.mem;
12:   end if
13: end for
14: /*Call the Resource discovery algorithm*/
15: ResidualMap $\leftarrow$ ResourceDiscoveryAlgorithm;
16: for each item $\in$ ResidualMap do
17:   totalResidual.cpu += item.residual.cpu;
18:   totalResidual.mem += item.esidual.mem;
19:   if item.residual.cpu > $\text{Re}_{\text{max}}^{\text{cpu}}$ then
20:      $\text{Re}_{\text{max}}^{\text{cpu}}$ = item.residual.cpu;
21:      $\text{Re}_{\text{max}}^{\text{mem}}$ = item.residual.mem;
22:   end if
23: end for
24: /*Call the Resource evaluation algorithm*/
25: $\text{allocated}_{\text{cpu}}$, $\text{allocated}_{\text{mem}}$ $\leftarrow$
26:          ResourceEvaluationAlgorithm;
27: if ($\text{allocated}_{\text{cpu}} \geqslant s_{i,j}.\text{min}_{\text{cpu}}$) and
28:   ($\text{allocated}_{\text{mem}} \geqslant s_{i,j}.\text{min}_{\text{mem}} + \beta$) then
29:   break;
30: end if
31: end for
32: Return $\text{allocated}_{\text{cpu}}$, $\text{allocated}_{\text{mem}}$.
```

To ensure the task pod run properly in our experimental testbed, with the running program via stress tool within the task pod in mind, we add a constant β to the minimum running resource for the task pod. It is because that Stress tool in the running program of task pod uses min_{mem} to allocate and release memory resources for resource loads. Resource amount $\text{min}_{\text{mem}} + \beta$ as a minimum of memory resource is just enough to run the task pods. Finally, the allocated resources returned by the proposed algorithm meet the conditions of minimum resource demands (line 27).

5.2 Resource discovery algorithm

Algorithm 2 shows how our resource discovery algorithm acquires the remaining resources of the K8s cluster and returns the remaining resource MapResidualMap. At first step, this algorithm initializes related parameters to be 0 in line 1 and respectively get the PodList and NodeList of K8s cluster from PodLister and NodeLister through informer component. Then the algorithm traverses all nodes in the K8s cluster and uses for-loop (lines 6-13) to acquire the total number of occupied resources for all pods with Running and Pending states on the current node v_i. Lines 15−17 obtain the residual CPU and memory resources on the current v_i. Next, the algorithm encapsulates the ResidualMap for each v_i. Once the iteration is complete for all the K8s cluster nodes, the algorithm returns the residual resource ResidualMap.

Algorithm 2 Resource discovery algorithm

```
Input: PodLister, NodeLister, ResidualMap;
Output: ResidualMap;
1: Initialization: nodeReq.cpu, nodeReq.mem, allocatable.cpu, allocatable.mem, residual.cpu, residual.mem ← 0;
2: Get the PodList from PodLister through Informer;
3: Get the NodeList from NodeLister through Informer;
4: for node $v_i \in V$ do
5:    /*Obtain the total resource requests of all pods on $v_i$*/
6:    for each pod $p_i$ in PodList do
7:       if $p_i$ is hosted in $v_i$ then
8:          if pod$_i$.phase $\in$ {Running, Pending} then
9:             nodeReq.cpu += pod$_i$.request.cpu;
10:            nodeReq.mem += pod$_i$.request.mem;
11:         end if
12:      end if
13:   end for
14:   /*Obtain the allocatable resources on each node $v_i$*/
15:   Obtain node$_i$ from NodeList corresponding to $v_i$.
16:   allocatable.cpu = node$_i$.allocatable.cpu;
17:   allocatable.mem = node$_i$.allocatable.mem;
18:   /*Acquire the remaining resources on node $v_i$*/
19:   residual.cpu = allocatable.cpu−nodeReq.cpu;
20:   residual.mem = allocatable.mem−nodeReq.mem;
21:   /*Encapsulate Dictionary ResidualMap*/
22:   ResidualMap[$v_i.ip$]={residual.cpu, residual.mem};
23: end for
24: Return ResidualMap.
```

5.3 Resource evaluation algorithm

Algorithm 3 elaborates the resource evaluation process in detail. The algorithm takes task_{req}, $\text{Re}_{\text{max}}^{\text{cpu}}$, $\text{Re}_{\text{max}}^{\text{mem}}$, totalResidual.cpu, totalResidual.mem, request.cpu and request.mem as input and finally return the allocated CPU and memory resources. As mentioned in Subsection 5.1, some task pods to be launched during the requested task pod's lifecycle will compete for computational resources with the current task request task^{req}, and we use a resource scaling method to provide resource allocation based on a proportional value of the total remaining resources over the total amount of resource requests, defined as follows:

$$\begin{cases} \text{cpu}_{\text{cut}} = (\text{task}^{\text{req}}.\text{cpu})\cdot \dfrac{\text{totalResidual.cpu}}{\text{request.cpu}} \\ \text{mem}_{\text{cut}} = (\text{task}^{\text{req}}.\text{mem})\cdot \dfrac{\text{totalResidual.mem}}{\text{request.mem}} \end{cases}. \tag{9}$$

Algorithm 3 Resource evaluation algorithm

```
Input: $\text{task}^{\text{req}}$, $\text{Re}_{\text{max}}^{\text{cpu}}$, $\text{Re}_{\text{max}}^{\text{mem}}$, totalResidual.cpu, totalResidual.mem, request.cpu, request.mem;
Output: allocated.cpu, allocated.mem;
1: Get $\text{cpu}_{\text{cut}}$ and $\text{mem}_{\text{cut}}$ through (9);
2: Define conditions request.cpu<totalResidual.cpu as $A_1$, request.mem<totalResidual.mem as $A_2$, $\text{task}^{\text{req}}$.cpu<$\text{Re}_{\text{max}}^{\text{cpu}}$ as $B_1$, $\text{task}^{\text{req}}$.mem<$\text{Re}_{\text{max}}^{\text{mem}}$ as $B_2$, $\text{cpu}_{\text{cut}}$<$\text{Re}_{\text{max}}^{\text{cpu}}$ as $C_1$, and $\text{mem}_{\text{cut}}$<$\text{Re}_{\text{max}}^{\text{mem}}$ as $C_2$;
3: Define the symbol $\neg$ as the negation of a condition and the symbol $\wedge$ as the logical and.
4: /*① The remaining resources are sufficient*/
5: if $A_1 \wedge A_2$ then
6:    if $B_1 \wedge B_2$ then
7:       allocated.cpu = $\text{task}^{\text{req}}$.cpu
8:       allocated.mem = $\text{task}^{\text{req}}$.mem
9:    else
10:      if $\neg B_1 \wedge B_2$ then
11:         allocated.cpu = $\text{Re}_{\text{max}}^{\text{cpu}} \cdot \alpha$
12:         allocated.mem = $\text{task}^{\text{req}}$.mem
13:      else
14:         if $B_1 \wedge \neg B_2$ then
15:            allocated.cpu = $\text{task}^{\text{req}}$.cpu
16:            allocated.mem = $\text{Re}_{\text{max}}^{\text{mem}} \cdot \alpha$
17:         else
18:            allocated.cpu = $\text{Re}_{\text{max}}^{\text{cpu}} \cdot \alpha$
19:            allocated.mem = $\text{Re}_{\text{max}}^{\text{mem}} \cdot \alpha$
20:         end if
21:      end if
22:   end if
23: end if
24: /*② The remaining cpu resource is unsufficient*/
25: if $\neg A_1 \wedge A_2$ then
26:   if $C_1 \wedge B_2$ then
27:      allocated.cpu = $\text{cpu}_{\text{cut}}$
28:      allocated.mem = $\text{task}^{\text{req}}$.mem
29:   else
```

```
30:         if $\neg C_1 \wedge B_2$ then
31:             allocated.cpu = $\mathrm{Re}_{\max}^{\mathrm{cpu}} \cdot \alpha$
32:             allocated.mem = $\mathrm{task}^{\mathrm{req}}$.mem
33:         else
34:             if $C_1 \wedge \neg B_2$ then
35:                 allocated.cpu = $\mathrm{cpu}_{\mathrm{cut}}$
36:                 allocated.mem = $\mathrm{Re}_{\max}^{\mathrm{mem}} \cdot \alpha$
37:             else
38:                 allocated.cpu = $\mathrm{Re}_{\max}^{\mathrm{cpu}} \cdot \alpha$
39:                 allocated.mem = $\mathrm{Re}_{\max}^{\mathrm{mem}} \cdot \alpha$
40:             end if
41:         end if
42:     end if
43: end if
44: /*③ The remaining memory resource is unsufficient*/
45: if $A_1 \wedge \neg A_2$ then
46:     if $B_1 \wedge C_2$ then
47:         allocated.cpu = $\mathrm{task}^{\mathrm{req}}$.cpu
48:         allocated.mem = $\mathrm{mem}_{\mathrm{cut}}$
49:     else
50:         if $\neg B_1 \wedge C_2$ then
51:             allocated.cpu = $\mathrm{Re}_{\max}^{\mathrm{cpu}} \cdot \alpha$
52:             allocated.mem = $\mathrm{mem}_{\mathrm{cut}}$
53:         else
54:             if $B_1 \wedge \neg C_2$ then
55:                 allocated.cpu = $\mathrm{task}^{\mathrm{req}}$.cpu
56:                 allocated.mem = $\mathrm{Re}_{\max}^{\mathrm{mem}} \cdot \alpha$
57:             else
58:                 allocated.cpu = $\mathrm{Re}_{\max}^{\mathrm{cpu}} \cdot \alpha$
59:                 allocated.mem = $\mathrm{Re}_{\max}^{\mathrm{mem}} \cdot \alpha$
60:             end if
61:         end if
62:     end if
63: end if
64: /*④ Both residual resources are unsufficient*/
65: if $\neg A_1 \wedge \neg A_2$ then
66:         allocated.cpu = $\mathrm{cpu}_{\mathrm{cut}}$
67:         allocated.mem = $\mathrm{mem}_{\mathrm{cut}}$
68: end if
69: Return allocated.cpu, allocated.mem.
```

In addition, we define a resource allocation factor α for each node with a maximum of residual resources (CPU or memory). Through lots of experimental evaluations, we use $\alpha = 0.8$, which means that the algorithm only allocates 80% of the remaining resources in response to insufficient residual resource scenario on the current node while ensuring 20% residual resources for its other loads.

Algorithm 3 first uses resource scaling to obtain allocated CPU and memory resources by (9) and defines six comparative conditions A_1, A_2, B_1, B_2, C_1, and C_2 (lines 1–2). The symbol $\neg$ denotes the negation of the condition and the symbol $\wedge$ denotes the logical and (line 3). Then the algorithm implements four resource allocation scenarios according to comparative conditions between accumulated resource requests (concurrent scenario) within the current task lifecycle and the total residual resources.

Sufficient residual resources. When the total residual resources across the K8s cluster are abundant for concurrent tasks within the current task lifecycle, we only think about B_1 and B_2. When the CPU and memory resource requests of the current $\mathrm{task}^{\mathrm{req}}$ are smaller than the maximum remaining resources of a node (meets B_1 and B_2), the algorithm allocates resources in the light of the current task request $\mathrm{task}^{\mathrm{req}}$ (lines 6–8). If the maximum remaining CPU resource on the cluster node fails to suffice for the current task request $\mathrm{task}^{\mathrm{req}}$, the algorithm allocates $\mathrm{Re}_{\max}^{\mathrm{cpu}} \cdot \alpha$ of the maximum remaining CPU resource on cluster node (lines 10–12). Conversely, so is memory (lines 14–16). When neither of the two remaining resources on cluster node (CPU and memory) can satisfy $\mathrm{task}^{\mathrm{req}}$, both types of allocated resources scale down by multiplying α (lines 17–19).

Insufficient residual CPU resource. When the total residual CPU resource across the K8s cluster cannot satisfy the total resource demand of concurrent tasks, conditions C_1 and B_2 are considered. The algorithm acquires the $\mathrm{cpu}_{\mathrm{cut}}$ through the resource scaling method (in (9)). In case of conditions C_1 and B_2, we allocate resources according to $\mathrm{cpu}_{\mathrm{cut}}$ and $\mathrm{task}^{\mathrm{req}}$.mem (lines 26–28). When the maximum CPU remaining resources of a node fail to accommodate $\mathrm{cpu}_{\mathrm{cut}}$, the algorithm adopts $\mathrm{Re}_{\max}^{\mathrm{cpu}} \cdot \alpha$ as the allocated CPU resource. Due to sufficient memory capacities, the algorithm suffices for the current task memory request $\mathrm{task}^{\mathrm{req}}$.mem (lines 30–32). In case of conditions $C_1 \wedge \neg B_2$, the algorithm allocates $\mathrm{cpu}_{\mathrm{cut}}$ CPU resource and $\mathrm{Re}_{\max}^{\mathrm{mem}} \cdot \alpha$ memory resource, which is due to that the current task memory request is greater than the maximum residual memory resource on cluster node (lines 34–36). Instead, the algorithm allocates resources (CPU and memory) according to the α scale factor of the largest node's remaining resources (lines 37–39).

Insufficient residual memory resource. When the total residual memory resource across the K8s cluster cannot satisfy the total resource demand of concurrent tasks, conditions B_1 and C_2 are considered. The algorithm acquires the $\mathrm{mem}_{\mathrm{cut}}$ through the resource scaling method (in (9)). The operations under conditions $B_1 \wedge C_2$, $\neg B_1 \wedge C_2$, $B_1 \wedge \neg C_2$ and $\neg B_1 \wedge \neg C_2$ are similar to the above, except that here we are talking about memory resource (lines 45–63).

Insufficient residual CPU and memory resources. In the case of $\neg A_1 \wedge \neg A_2$, meaning that the total remaining CPU and memory resources across the K8s cluster fail to suffice for the CPU and memory resource requests of concurrent tasks, the algorithm allocates CPU and memory resources according to cpu_{cut} and mem_{cut} obtained by resource scaling method (lines 65−67). Finally, Resource evaluation algorithm returns allocated resources allocated.cpu and allocated.mem.

6. Experimental evaluation

In the following subsections, we evaluate the proposed ARAS for different evaluation metrics and discuss the benefits of the proposed ARAS under three distinct arrival patterns compared with the baseline.

6.1 Experimental setup and design

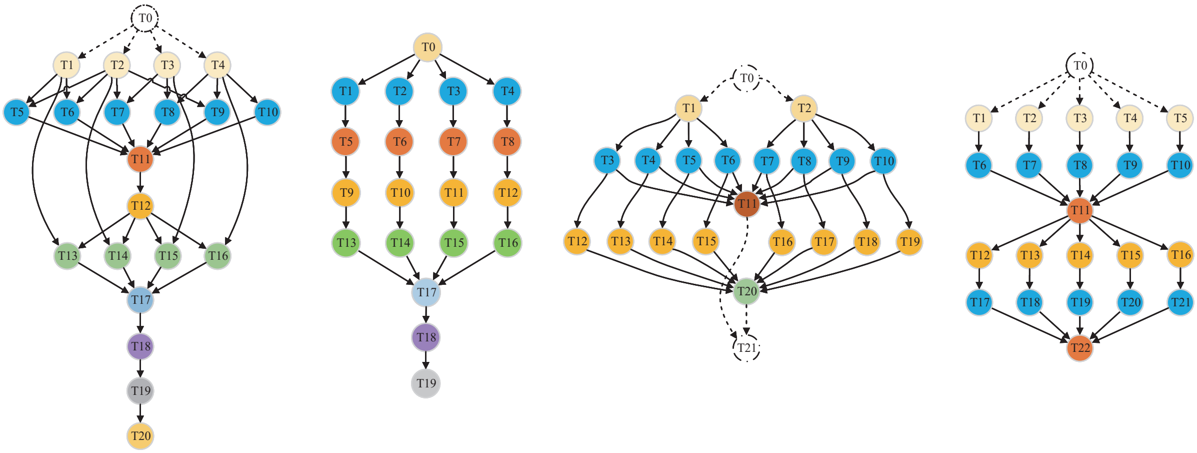
For the evaluation, we apply a setting employed in our former work [21] and make adaptations for the proposed ARAS within KubeAdaptor discussed here. In this subsection, we briefly introduce experimental scenarios, workflow examples, workflow instantiation, workflow arrival patterns, evaluation metrics, and baseline algorithm.

6.1.1 Experimental scenarios

The K8s cluster used in our experiments consists of one Master node and six nodes. Each node equips with an 8-core AMD EPYC 7742 2.2 GHz CPU and 16 GB of RAM, running Ubuntu 20.4 and K8s v1.19.6 and Docker version 18.09.6. The Redis database v5.0.7 is installed on the Master node. Workflow injector module and Containerized workflow builder are containerized and deployed into the K8s cluster through service and deployment. We explore the performances of the proposed ARAS and baseline by running four scientific workflows on the K8s cluster.

6.1.2 Workflow examples

To verify the adaptations of our proposed ARAS within KubeAdaptor, four scientific workflows, such as montage (astronomy), epigenomics (genome sequence), CyberShake (earthquake science), and laser interferometer gravitation-wave observatiory (LIGO) inspiral (gravitational physics), are used to run on K8s cluster in a containerized manner [3]. We make a few tweaks to the workflow structure and add virtual entrance and exit nodes of workflows to form the workflow structure with the DAG diagram. As for each type of scientific workflow, we uniformly adopt a small-scale workflow (about 20 tasks) in our experiments, as shown in Fig. 4, derived from the Pegasus workflow repository [38]. Structurally, four types of workflows cover all the structural features regarding composition and components (in-tree, out-tree, fork-join, and pipeline) that serve to illustrate the universality and complexity of workflows. Here, we only consider the topologies of four scientific workflows and do not focus on the real-world data processing of the tasks, which does not affect verifying the adaptability of our ARAS. For ease of performance comparisons among resource allocation solutions, we assume that four classes of scientific workflows consist of the same tasks. Each node of workflow DAGs uses resource load (CPU and memory utilization) and service runtime to simulate workflow tasks in the experiments. Note that the KubeAdaptor schedules workflow tasks topologically in a top-down fashion according to task dependencies.



(a) Montage (b) Epigenomics (c) CyberShake (d) LIGO

Fig. 4 Topological diagram of four scientific workflow applications

6.1.3 Workflow instantiation

As for resource load in workflow tasks, we employ several parameters together with Stress to work on simulating scientific workflow tasks. In each task program, we use the Stress tool to set several CPU forks, a memory of

1 000Mi (is equal to $\text{mem}_{\min}$ of (1)), and random duration (user's definition ahead of time in (1)). CPU forking and memory allocation operations in the task pod last twice as long as duraion. The total duration of each task pod is random and falls between 10 s and 20 s. Then we pack the Python application with stress program into a task Image file through Docker Engine, store the task Image file in local Harbor or remote Docker Hub repository. We can import container parameters (refer to (1)) defined in the ConfigMap file in Workflow injection module into the task container hosted in the task pod. For resource setting within task pod, we uniformly set the resource requests and limits to 2 000 Milli cores (i.e., 2 000 m) CPU and 4 000Mi memory. Note that the requests field has the same parameter as the limits field, which ensures that this task pod has the highest priority, namely Guaranteed [39].

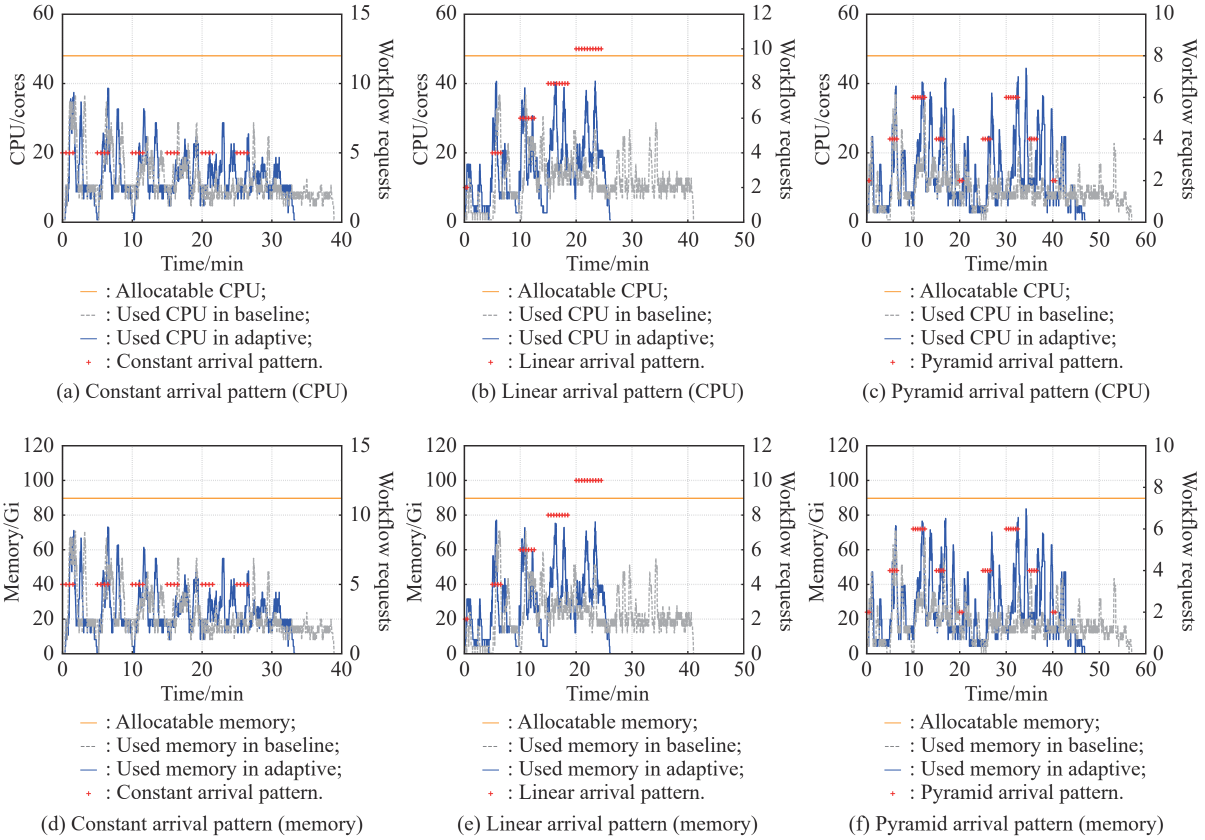
6.1.4 Workflow arrival patterns

We make use of three distinct workflow request arrival patterns.

Constant arrival scenario: in this scenario, workflow requests arrive in a constant manner. Workflow injector moduler together with CLI sends five workflow requests simultaneously to the containerized workflow builder every 300 seconds, i.e., $y = 5$. Send six times for a total of 30 workflows. A graphical depiction of this arrival curve is depicted in Fig. 5(a).

Linear arrival scenario: in this scenario, the workflow requests are injected into Containerized workflow builder in a linear rising function, i.e., $y = kx + d$, where y is the amount of concurrent workflow request and d is the initial value 2. Concurrent workflow requests increase by $k = 2$ every 300 seconds. This linear arrival curve is depicted in Fig. 5(b). Send five times for a total of 30 workflows.

Pyramid arrival scenario: in this scenario, workflow requests are sent to containerized workflow builder in line with a pyramid-like function. We start with a small number of concurrent workflow requests (is equal to 2), till it grows to a randomly selected large number (is equal to 6 for each type of workflow), which can be seen in Fig. 5(c). Concurrent workflow requests grow every 300 seconds by 2 until the peak is reached. Once the peak reaches, we immediately reduce this number to the small initial value in the same manner and repeat this process until the total number of workflow requests is reached (here the number is 34).



(a) Constant arrival pattern (CPU)

(b) Linear arrival pattern (CPU)

(c) Pyramid arrival pattern (CPU)

(d) Constant arrival pattern (memory)

(e) Linear arrival pattern (memory)

(f) Pyramid arrival pattern (memory)

Fig. 5 CPU and memory resource usage rate under three distinct arrival patterns for Montage workflows

The deployment of these three scenarios aims to maximize coverage of the ever-changing resource needs and sudden peaks of workflow requests in a production environment. Even if there is some predictability in the constant and linear arrival scenarios, the Pyramid function follows an unpredictable arrival pattern.

6.1.5 Evaluation metrics

To evaluate our ARAS, we conduct each arrival pattern three times and analyze the results against the following quantitative metrics.

Total duration of all workflows (in minutes): this metric is the average total duration of all injected workflows, i.e., the elapsed time from the arrival of the first workflow request to the moment when the last workflow request is complete.

Average workflow duration (in minutes): this metric reflects the average execution time of individual workflow, which is the time that each workflow takes from the start of the first task to the end of the last.

Resource usage: resource usage contains CPU and memory resource utilization, reflecting the average resource utilization throughout the total duration of all injected workflows across the K8s cluster. The greater the resource utilization, the closer to our optimization goal. The resource usage comparison covering four types of scientific workflows with the baseline algorithm further verifies the better performance of our ARAS solution.

6.1.6 Baseline

In experiments, we used our recent resource allocation strategy [21] as a baseline method, which does not take into account the potential future task requests throughout the current task's lifecycle. It means that the resource allocation strategy in the baseline follows the first come first serve (FCFS) and relies on the adequacy of residual resources on cluster nodes. If enough, the resource allocation is complete. Otherwise, wait for other task pods to complete and release resources to meet the resource reallocation for the current task request.

6.2 Results and analysis

To fully evaluate KubeAdaptor together with our ARAS, we present a general evaluation and the evaluation of resource allocation failure, and discuss the evaluation results. In order to minimize external influences, our K8s cluster has no other application load, and we execute each evaluation three times at different times of one day.

6.2.1 General evaluation

In the following, we use the KubeAdaptor with our ARAS and the baseline to run four scientific workflows against three distinct workflow arrival patterns three times and compare our ARAS with the baseline on experimental results. We calculate the mean value and the standard deviation δ for all metrics.

Table 2 presents the resulting mean values and the standard deviation from the conducted evaluation runs. In general, the observed standard deviation is low and therefore indicates a low dispersion in the results of the different evaluations. As presented in Table 2, "adaptive" denotes our ARAS, while "Baseline" marks the application of the baseline algorithm (mentioned in Subsection 6.1.6). The interval between two workflow request bursts is set to 300 seconds for three arrival patterns, and the amount of injected workflows for three arrival patterns is set to 30, 30, and 34, respectively.

Table 2 Evaluation results

Workflow type	Metrics	Constant arrival		Linear arrival		Pyramid arrival	
		Adaptive	Baseline	Adaptive	Baseline	Adaptive	Baseline
	Number of workflow rquests	30		30		34	
	Interval between two requests bursts/s	300		300		300	
Montage	Total duration of all workflows/min (standard deviation)	33.18 ($\delta = 0.21$)	36.79 ($\delta = 2.26$)	26.95 ($\delta = 0.38$)	36.45 ($\delta = 6.31$)	49.31 ($\delta = 2.46$)	54.69 ($\delta = 1.74$)
	Average workflow duration/min (standard deviation)	5.74 ($\delta = 0.49$)	7.80 ($\delta = 0.36$)	5.41 ($\delta = 0.26$)	11.33 ($\delta = 4.28$)	7.22 ($\delta = 1.36$)	11.73 ($\delta = 0.88$)
	CPU resource usage (standard deviation)	0.28 ($\delta = 0.00$)	0.27 ($\delta = 0.02$)	0.35 ($\delta = 0.01$)	0.31 ($\delta = 0.07$)	0.26 ($\delta = 0.03$)	0.20 ($\delta = 0.01$)
	Memory resource usage (standard deviation)	0.28 ($\delta = 0.00$)	0.27 ($\delta = 0.13$)	0.35 ($\delta = 0.01$)	0.31 ($\delta = 0.07$)	0.26 ($\delta = 0.03$)	0.20 ($\delta = 0.01$)
Epigenomics	Total duration of all workflows/min (standard deviation)	30.55 ($\delta = 0.19$)	39.06 ($\delta = 1.84$)	34.3 ($\delta = 7.29$)	43.66 ($\delta = 4.37$)	51.42 ($\delta = 4.28$)	62.12 ($\delta = 4.32$)
	Average workflow duration/min (standard deviation)	4.24 ($\delta = 0.05$)	9.35 ($\delta = 1.56$)	9.81 ($\delta = 5.11$)	16.53 ($\delta = 4.41$)	9.65 ($\delta = 3.33$)	19.41 ($\delta = 6.04$)
	CPU resource usage (standard deviation)	0.34 ($\delta = 0.02$)	0.27 ($\delta = 0.01$)	0.32 ($\delta = 0.06$)	0.25 ($\delta = 0.00$)	0.21 ($\delta = 0.00$)	0.20 ($\delta = 0.00$)
	Memory resource usage (standard deviation)	0.34 ($\delta = 0.02$)	0.27 ($\delta = 0.01$)	0.32 ($\delta = 0.06$)	0.25 ($\delta = 0.00$)	0.21 ($\delta = 0.01$)	0.20 ($\delta = 0.01$)

Continued

Workflow type	Metrics	Constant arrival		Linear arrival		Pyramid arrival	
		Adaptive	Baseline	Adaptive	Baseline	Adaptive	Baseline
	Number of workflow rquests	30		30		34	
	Interval between two requests bursts/s	300		300		300	
CyberShake	Total duration of all workflows/min (standard deviation)	38.30 ($\delta = 5.77$)	50.29 ($\delta = 5.29$)	34.06 ($\delta = 6.16$)	49.46 ($\delta = 1.18$)	46.76 ($\delta = 4.02$)	66.41 ($\delta = 6.56$)
	Average workflow duration/min (standard deviation)	9.19 ($\delta = 3.72$)	17.29 ($\delta = 2.89$)	9.41 ($\delta = 4.27$)	20.61 ($\delta = 0.86$)	4.94 ($\delta = 2.07$)	19.47 ($\delta = 6.50$)
	CPU resource usage (standard deviation)	0.26 ($\delta = 0.03$)	0.24 ($\delta = 0.02$)	0.27 ($\delta = 0.04$)	0.24 ($\delta = 0.01$)	0.22 ($\delta = 0.03$)	0.19 ($\delta = 0.01$)
	Memory resource usage (standard deviation)	0.26 ($\delta = 0.03$)	0.24 ($\delta = 0.02$)	0.27 ($\delta = 0.04$)	0.23 ($\delta = 0.01$)	0.22 ($\delta = 0.03$)	0.19 ($\delta = 0.01$)
LIGO	Total duration of all workflows/min (standard deviation)	30.82 ($\delta = 0.38$)	52.17 ($\delta = 3.99$)	44.02 ($\delta = 10.9$)	53.87 ($\delta = 4.20$)	45.26 ($\delta = 0.52$)	63.56 ($\delta = 1.60$)
	Average workflow duration/min (standard deviation)	4.26 ($\delta = 0.09$)	21.15 ($\delta = 0.44$)	16.21 ($\delta = 7.68$)	28.05 ($\delta = 7.88$)	4.20 ($\delta = 0.15$)	14.07 ($\delta = 1.33$)
	CPU resource usage (standard deviation)	0.40 ($\delta = 0.00$)	0.24 ($\delta = 0.02$)	0.28 ($\delta = 0.08$)	0.23 ($\delta = 0.02$)	0.31 ($\delta = 0.01$)	0.23 ($\delta = 0.00$)
	Memory resource usage (standard deviation)	0.40 ($\delta = 0.00$)	0.24 ($\delta = 0.02$)	0.28 ($\delta = 0.08$)	0.23 ($\delta = 0.02$)	0.31 ($\delta = 0.01$)	0.23 ($\delta = 0.00$)

Generally, our ARAS is superior to the baseline algorithm on each observation metric against four different workflow types under distinct workflow arrival patterns. In addition, the CPU and memory resources set in the task pod are constant, the allocatable cluster resources are fixed, and the resource scaling method scales down resources according to (10). So in each arrival pattern and each resource allocation algorithm, the utilization rate of CPU and memory resources are the same, and both resource usage curves in each workflow arrival pattern are similar. In the following, we elaborate on the evaluation metrics of each workflow arrival pattern in the light of workflow types.

Fig. 5 to Fig. 8 illustrate the presentation of the average evaluation results by depicting the arrival patterns (workflow requests) over time and the number of used computational resources (CPU and memory) until all workflow requests have been served. Note that the used resource curve in each workflow arrival pattern usually ends later than the workflow request curve. It can be traced to the fact that each workflow has a deadline in the future, and some workflows are still waiting in queue for execution.

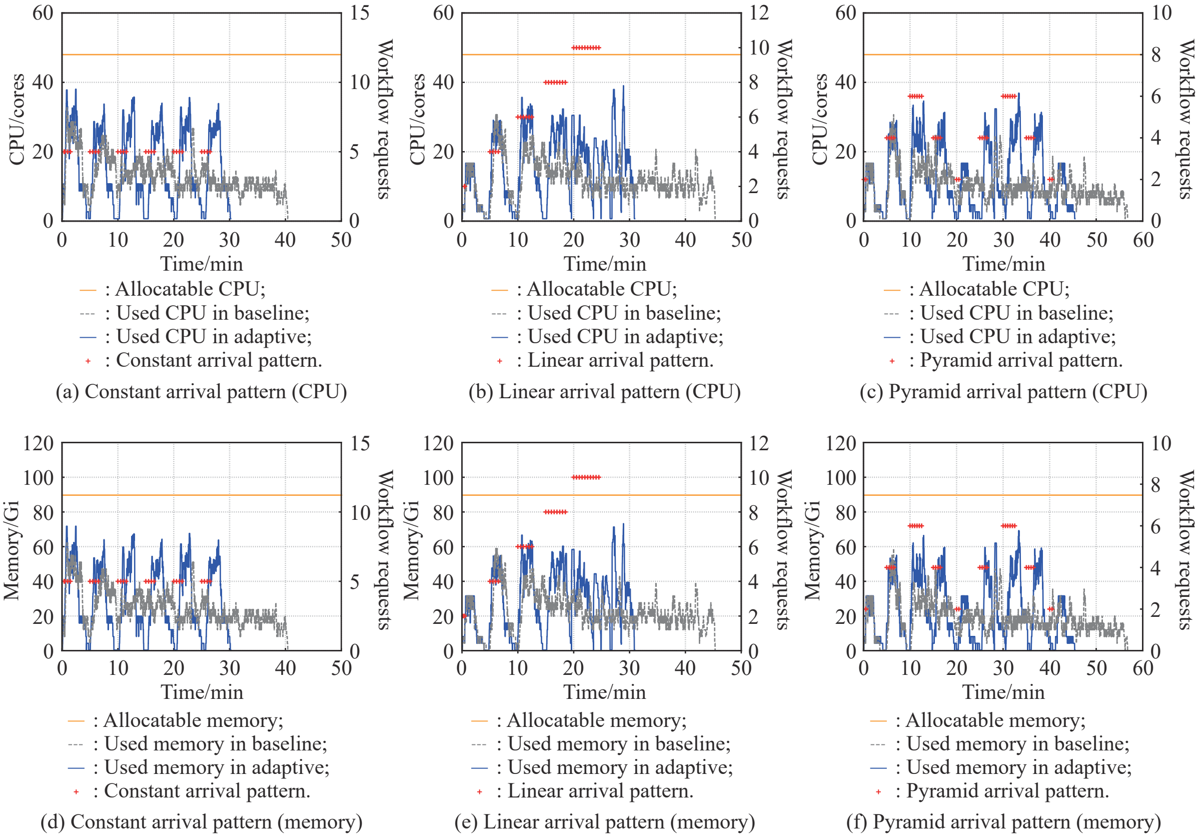
Montage: in our experimental setup, a small-scale Montage workflow consists of 21 tasks (refer to Fig. 4(a)). Compared with the baseline, as for the total duration of all workflows in Table 2, our ARAS leads to time savings of 9.8% for the constant arrival pattern and time savings of 26.06% for the linear arrival pattern, while in the pyramid arrival pattern, time savings amounts to 9.8%. Similarly, as for average workflow duration, our ARAS, in comparison to the baseline, gains a time saving of 26.4%, time savings of 52.3%, and time savings of 38.5% for three different workflow arrival patterns from left to right, respectively. Fig. 5 broadly reflects the consistency of the above data with the total duration of all injected workflows, even with a set of evaluation data. Our ARAS achieves better performances in total workflow duration and average workflow duration in linear arrival patterns. It can be deduced that the concurrent degree of received workflow requests is directly related to the total duration of all injected workflows and the average duration of a single workflow. The higher the concurrent degree of received workflow requests, the more workflow tasks are executed per unit time, so the shorter the total duration of all injected workflows and the average duration of individual workflow.

Regarding the CPU and memory resource usage in Fig. 5, our ARAS outperforms the baseline for each arrival pattern. Herein, the linear arrival pattern features a maximum value of 35% for our ARAS, 4% higher than the baseline algorithm. Our ARAS outperforms the baseline algorithm by 1% and 6%, respectively, in the other two patterns. It can be traced back to the fact that over time the linear arrival pattern requests more task pods to be performed in parallel in response to more and more workflow requests and gains a maximum resource usage rate.

Looking at the resource usage curves (CPU and memory) of three workflow arrival patterns in Fig. 5, the resource usage peak of our ARAS is higher than that of the baseline algorithm for most of the time. It can be further observed that the peak of the resource usage curve is consistent with the centralized arrival of workflow requests. It is because our ARAS can use a resource scaling strategy to adjust the resource limits of potential future task requests within the current task's lifecycle. This scheme launches task pods as many as possible on the premise of the smooth operation of task pods, thus speeding up the execution efficiency of workflows. However, the baseline algorithm depends on the adequacy of residual resources on cluster nodes. In high concurrency scenarios, the insufficient remaining resources of nodes will make the baseline algorithm lead to endless waiting and much time-wasting and prolong the total duration of workflows and the average duration of a single workflow.

Epigenomics: we adopt a small-scale Epigenomics workflow with 20 tasks in experimental evaluations. As can be seen from Fig. 4(b), the topology of Epigenomics workflows is mostly pipeline structure. As for the total duration of all workflows, our ARAS obtains time savings of 21.8% for the constant arrival pattern, time savings of 21.4% for the linear arrival pattern, and time savings of 17.2% for the pyramid arrival pattern compared with the baseline. As for average workflow duration, our ARAS, in comparison to the baseline, gains time savings of 54.65%, time savings of 40.65%, and time savings of 50.28% for three arrival patterns from left to right, respectively. Note that the Epigenomics workflow is substantially more significant in performance improvement than the Montage workflow in terms of average total workflow duration and average duration of individual workflow for three arrival patterns. Because the pipeline topology in the Epigenomics workflow is better suited for high concurrency scenarios, our ARAS scheme saves more time than the baseline in response to continuous workflow requests.

Regarding the resource usage in Fig. 6, the constant arrival pattern features a maximum value of 34% for our ARAS, 7% higher than the baseline. The linear arrival pattern features a value of 32% for our ARAS, which is also 7% higher than the baseline. The higher resource utilization of these two patterns can be attributed to the fact that 30 workflows are injected in the first 25 min, totaling more than 600 tasks. The higher density of workflow requests results in higher CPU and memory resource utilization. In addition, the resource scaling method enables our ARAS to adjust the resource limits of the task pods in time and cope with the continuous workflow requests on the premise of the normal execution of workflow tasks. It also shows that the peak time of resource usage in our ARAS is longer than that of the baseline. The baseline algorithm waits for resource release in response to resource shortage on cluster nodes, so it consumes too much time and results in a longer total workflow duration and average duration of a single workflow.



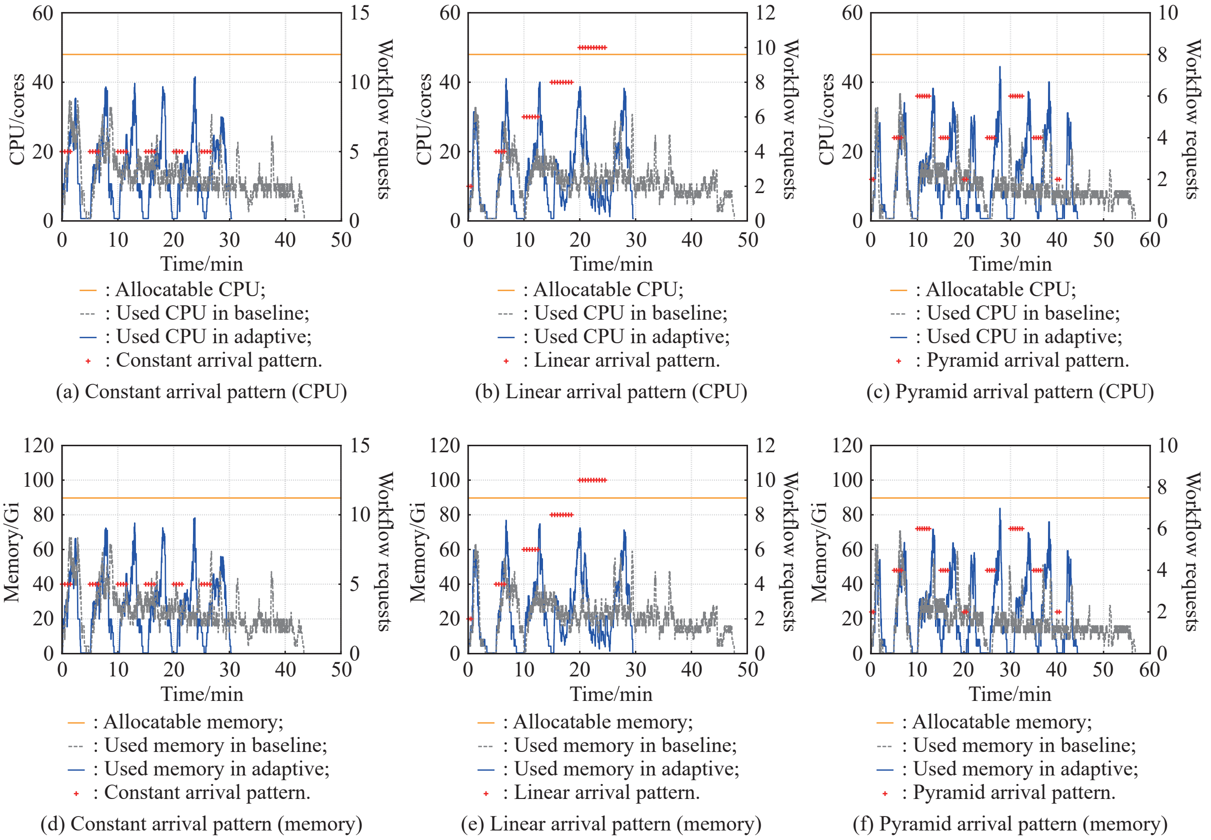
(a) Constant arrival pattern (CPU) (b) Linear arrival pattern (CPU) (c) Pyramid arrival pattern (CPU)

(d) Constant arrival pattern (memory) (e) Linear arrival pattern (memory) (f) Pyramid arrival pattern (memory)

Fig. 6 CPU and memory resource usage rate under three distinct arrival patterns for Epigenomics workflows

CyberShake: a small-scale CyberShake workflow in our experiments comprises 22 tasks (refers to Fig. 4(c)). Our ARAS, in comparison to the baseline, leads to time savings of 23.8% (for the constant arrival pattern), time savings of 31.1% (for the linear arrival pattern), and time savings of 29.6% (for the pyramid arrival pattern) for the total duration of all workflows. Similarly, for average workflow duration, our ARAS, in comparison to the baseline, gains time savings of 46.85%, time savings of 54.34%, and time savings of 74.63% for three arrival patterns from left to right, respectively.

Due to the topology structure of the CyberShake workflow with smaller depth and greater width, the CyberShke workflow features a higher degree of inherent parallelism, which is easier to take advantage of our ARAS in response to continuous workflow request arrivals. Compared with the baseline algorithm, the ARAS has prominent performance advantages on metrics of total workflow duration and duration of a single workflow. As for CPU and memory resource usage, the proposed ARAS obtains 26%, 27%, and 22% for three distinct arrival patterns, respectively, and slightly higher than the baseline. Combined with the resource utilization curve in Fig. 7, it can be observed that our ARAS benefiting from the resource scaling method outperforms the baseline in all performance metrics under three different workflow arrival patterns.



(a) Constant arrival pattern (CPU)

(b) Linear arrival pattern (CPU)

(c) Pyramid arrival pattern (CPU)

(d) Constant arrival pattern (memory)

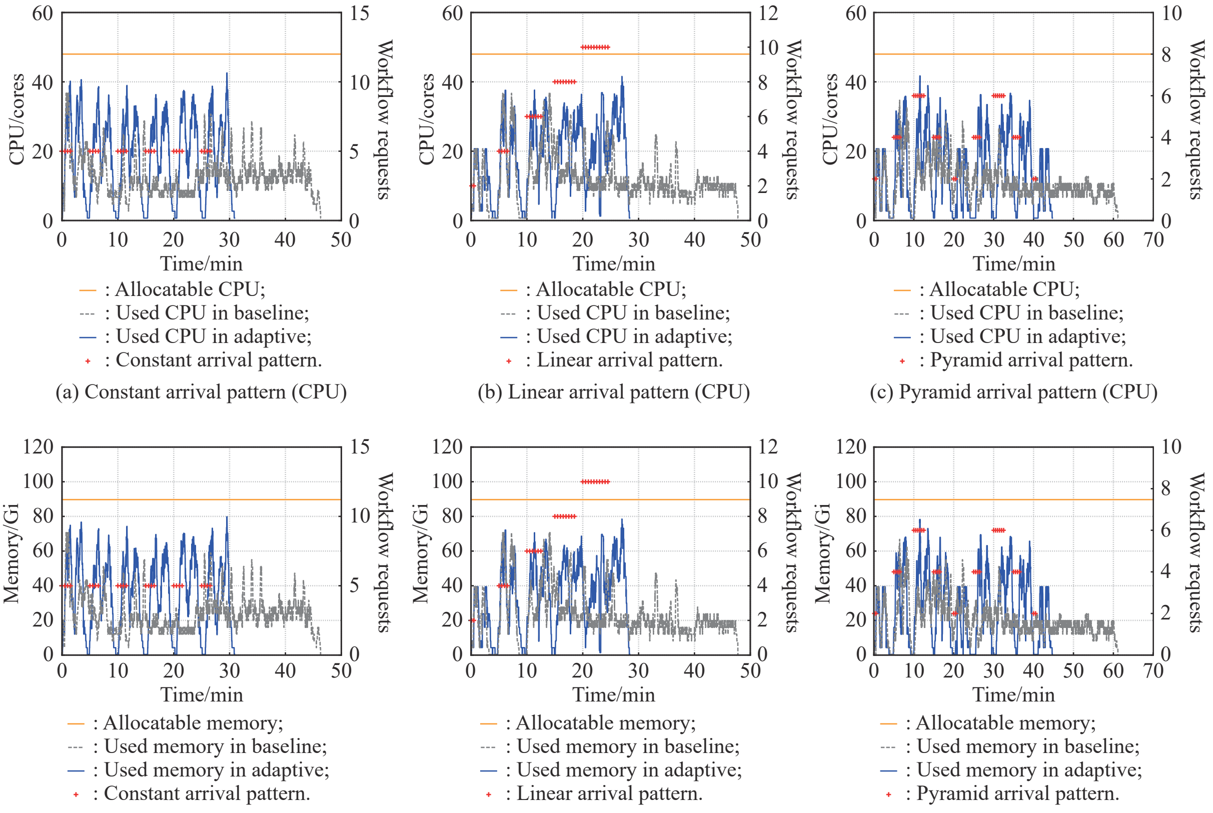
(e) Linear arrival pattern (memory)

(f) Pyramid arrival pattern (memory)

Fig. 7 CPU and memory resource usage rate under three distinct arrival patterns for Cybershake workflows

LIGO: a small-scale LIGO workflow in our experiments consists of 23 tasks (refers to Fig. 4(d)). Compared with the baseline, our ARAS gains time savings of 40.92% (for the constant arrival pattern), time savings of 18.28% (for the linear arrival pattern), and time savings of 28.79% (for the pyramid arrival pattern) for the total duration of all workflows. Similarly, for average workflow duration, our ARAS, in comparison to the baseline, gains time savings of 79.86%, time savings of 42.21%, and time savings of 70.15% for three arrival patterns from left to right, respectively. With unique concurrent topology, LIGO workflows, like epigenomics and CyberShake workflows, enable our ARAS to perform better in the total workflow duration and individual workflow duration metrics than the baseline algorithm under three different arrival patterns.

As for resource usage in Fig. 8, our ARAS obtained 40% for the constant arrival pattern, 28% for linear arrival pattern and 31% for pyramid arrival pattern, respectively, much higher than the baseline algorithm. In combination with the resource utilization curve trend, the resource scaling strategy and workflow's unique concurrency topology once again help the proposed ARAS outperform the baseline under three different workflow arrival patterns.



(a) Constant arrival pattern (CPU) (b) Linear arrival pattern (CPU) (c) Pyramid arrival pattern (CPU)

(d) Constant arrival pattern (memory) (e) Linear arrival pattern (memory) (f) Pyramid arrival pattern (memory)

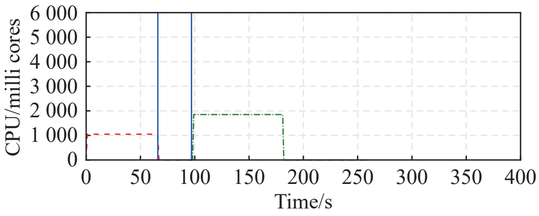
Fig. 8 CPU and memory resource usage rate under three distinct arrival patterns for LIGO workflows

6.2.2 Evaluation of resource allocation failure

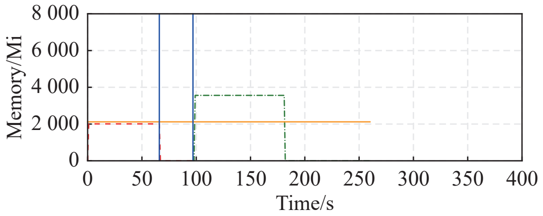
In this evaluation, we analyze the behavior of the KubeAdaptor in a failure situation of resource allocation. This situation means that our ARAS allocates resource quotas less than $\min_{\rm mem}+\beta$ through the resource scaling method against a high-concurrency scenario. So the task pods cannot smoothly execute and turn to OOMKilled status due to insufficient memory resources. Accordingly, the OOMKilled task pods make the workflow running get stuck. The source code of evaluation of resource allocation failture is available [40].

In the following, we investigate how KuberAdaptor responds to OOMKilled task pods, reallocates resources to execute task pods, and resumes workflow execution under our ARAS. For this evaluation, we inject 10 montage workflows into our K8s cluster (mentioned in Subsection 6.1.1) at a time under the constant arrival pattern. We fine-tune $\min_{\rm cpu}$ and $\min_{\rm mem}$ to be less than the amount of memory required by the Stress tool in the task pod. Subsequently, our ARAS tries to reduce the allocated resource quota by the resource scaling method in response to continuous workflow requests. When the allocated resource is less than $\min_{\rm mem}+\beta$, OOMKilled task pods will appear due to running resource shortage.

Fig. 9 depicts the results of this evaluation. The first vertical blue line labeled OOMKilled indicates that this task pod encounters OOMKilled due to insufficient allocated memory resources. Another vertical blue line labeled Reallocation indicates that the task pod previously OOMKilled is recreated due to sufficient allocated memory resources.



(a) CPU



(b) Memory

- - - : Allocated memory of OOMKilled task pod;
-·-· : Reallocated memory of this task pod;
— : Minimum memory for the task pod to run.

Fig. 9 Evaluation results of resource allocation failure

In Fig. 9 the first annotation marker, labeled OOMKilled, signalizes when the current task pod encounters the out of memory (OOM) event, and the other annotation marker, labeled Reallocation, signalizes when the current task pod is regenerated by using the reallocated computational resources. As can be seen in Fig. 9, at the beginning (second 0), our ARAS uses the resource scaling method to allocate CPU of 1 048 milli cores and memory of 2 009 Mi. In this evaluation, the minimum memory for a task pod to run, i.e., the amount of memory operated by the Stress tool in the task pod is set to 2 000 Mi. Here, we only focus on memory resources because memory resources are incompressible resources, and insufficient memory resources will trigger the task pod OOMKilled, while CPU resources as compressible resources do not. Once the allocated memory resource fails to reach $\min_{\text{mem}}+\beta$ (i.e., 2 000Mi + 20Mi), the current task pod turns to OOMKilled at 66 s. Meanwhile, the workflow with the current OOMKilled task pod also terminates execution. KubeAdaptor can capture the OOMKilled task pod and delete the task pod at 66 s. In our experimental evaluation, up to 210 task pods (10 Montage workflows) possess frequent created and deleted operations, leading to operation delay of deleting OOMKilled task pod. At 97 s, KubeAdaptor triggers the regeneration of the current task pod, reallocates computational resources, and launches the task pod. Since the second allocation of resources is sufficient for the smooth execution of the task pod (1 849 m CPU and 3 560 Mi memory), the task pod is completed at 181 s. At 258 s, KubeAdaptor deletes the completed task pod.

KubeAdaptor equipped with our ARAS in this paper can watch OOMKilled events, delete these OOMKilled task pods, reallocate computational resources, and regenerate these OOMKilled task pods, ensuring continuous execution of workflows. In production practice, the users inevitably misestimate the resource quota of the main program inside workflow tasks, resulting in a large number of OOMKilled task pods and termination of workflow execution. These countermeasures ensure continuous executions of workflows and keep the KubeAdaptor stable and robust. It also reflects the self-healing and self-configuration abilities of the KubeAdaptor (mentioned in Subsection 4.3).

6.2.3 Concluding discussion

Finally, it can be observed that the KubeAdaptor with our ARAS always achieves better results regarding each metric (Subsection 6.1.5). From Montage workflows to LIGO workflows, our ARAS outperforms the baseline against different metrics under three distinct workflow arrival patterns.

Most of the time savings from the total workflow duration and average duration of individual workflow result from the fact that the resource scaling method enables our ARAS to maximize resource utilization on cluster nodes according to our optimized functions while ensuring the smooth running of workflow pods. In addition, the workflow topology with concurrent characteristics also plays a positive role.

Concerning resource allocation failure and workflow recovery after termination, we have shown in Subsection 6.2.2 that the KubeAdaptor has abilities to watch the state changes of task pods in real-time, delete the OOMKilled task pods, and reallocate computational resources for the task pod, followed by the re-creation of this OOMKilled task pod and recover of workflow executing.

7. Conclusions

In this paper, we propose an ARAS for our tailored workflow management engine. With the novel architecture of KubeAdaptor and the integration between KubeAdaptor and K8s, our ARAS enables the KubeAdaptor to maximize resource utilization through the resource scaling method in response to complex and changing workflow requests. Experimental evaluations show that our ARAS, ranging from Montage to LIGO workflows, obtain better performances than the baseline algorithm for various metrics under different workflow arrival patterns. Furthermore, we have shown that the KubeAdaptor detects and handles failure situations of resource allocation. The self-healing and self-configuration abilities of the KubeAdaptor are also fully verified. In our future work, we intend to use KubeAdaptor to analyze different resource allocating algorithms and try to use deep reinforcement learning method to investigate cloud resource allocation for cloud workflows. In addition, we will study resource allocation strategies suitable for a cloud-edge cooperation environment and provide a practical solution for cloud-edge task scheduling.

References

[1] BURNS B, GRANT B, OPPENHEIMER D, et al. Borg, omega, and kubernetes. Communications of the ACM, 2016, 59(5): 50–57.

[2] BERNSTEIN D. Containers and cloud: from LXC to Docker to Kubernetes. IEEE Cloud Computing, 2014, 1(3): 81–84.

[3] JUVE G, CHERVENAK A, DEELMAN E, et al. Characterizing and profiling scientific workflows. Future Generation Computer Systems, 2013, 29(3): 682–692.

[4] LEE Y C, ZOMAYA A Y. Stretch out and compact: workflow scheduling with resource abundance. Proc. of the IEEE/ACM 13th International Symposium on Cluster, Cloud, and Grid Computing, 2013: 219–226.

[5] ZHENG C, TOVAR B, THAIN D. Deploying high throughput scientific workflows on container schedulers with makeflow and mesos. Proc. of the IEEE/ACM 17th International Symposium on Cluster, Cloud and Grid Computing, 2017: 130–139.

[6] SILVER A. Software simplified. Nature , 2017, 546(7656): 173–174.

[7] DI T P, CHATZOU M, FLODEN E W, et al. Nextflow enables reproducible computational workflows. Nature Biotechnology, 2017, 35(4): 316–319.

[8] DEELMAN E, VAHI K, JUVE G, et al. Pegasus, a workflow management system for science automation. Future Generation Computer Systems, 2015, 46: 17–35.

[9] DEELMAN E, VAHI K, RYNGE M, et al. The evolution of the pegasus workflow management software. Computing in Science & Engineering, 2019, 21(4): 22–36.

[10] JALILI V, AFGAN E, GU Q, et al. The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2020 update. Nucleic Acids Research, 2020, 48(W1): W395–W402.

[11] BADER J, THAMSEN L, KULAGINA S, et al. Tarema: adaptive resource allocation for scalable scientific workflows in heterogeneous clusters. Proc. of the IEEE International Conference on Big Data, 2021: 65–75.

[12] HOENISCH P, SCHULTE S, DUSTDAR S, et al. Self-adaptive resource allocation for elastic process execution. Proc. of the IEEE 6th International Conference on Cloud Computing, 2013: 220–227.

[13] HOENISCH P, SCHULTE S, DUSTDAR S. Workflow scheduling and resource allocation for cloud-based execution of elastic processes. Proc. of the IEEE 6th International Conference on Service-oriented Computing and Applications, 2013. DOI: 10.1109/SOCA.2013.44.

[14] WITT C, WAGNER D, LESER U. Feedback-based resource allocation for batch scheduling of scientific workflows. Proc. of the International Conference on High Performance Computing & Simulation, 2019: 761–768.

[15] KHATUA S, SUR P K, DAS R K, et al. Heuristic-based resource reservation strategies for public cloud. IEEE Trans. on Cloud Computing, 2014, 4(4): 392–401.

[16] ABDULLAH M, IQBAL W, BUKHARI F, et al. Diminishing returns and deep learning for adaptive CPU resource allocation of containers. IEEE Trans. on Network and Service Management, 2020, 17(4): 2052–2063.

[17] CHEN Z Y, HU J, MIN G, et al. Adaptive and efficient resource allocation in cloud datacenters using actor-critic deep reinforcement learning. IEEE Trans. on Parallel and Distributed Systems, 2021, 33(8): 1911–1923.

[18] CHEN M S, HUANG S, FU X, et al. Statistical model checking-based evaluation and optimization for cloud workflow resource allocation. IEEE Trans. on Cloud Computing, 2016, 8(2): 443–458.

[19] SCHULER L, JAMIL S, KUHL N. AI-based resource allocation: reinforcement learning for adaptive auto-scaling in serverless environments. Proc. of the IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing, 2021: 804–811.

[20] SHAN C G, WANG G, XIA Y Q, et al. Containerized workflow builder for Kubernetes. Proc. of the IEEE 23rd International Conference on High Performance Computing and Communications, 2021: 685–692.

[21] SHAN C G, WANG G, XIA Y Q, et al. KubeAdaptor: a docking framework for workflow containerization on Kubernetes. https://arxiv.53yu.com/abs/2207.01222.

[22] IGLESIA D G D L, WEYNS D. MAPE-K formal templates to rigorously design behaviors for self-adaptive systems. ACM Trans. on Autonomous and Adaptive Systems, 2015, 10(3): 15.

[23] ARCAINIP, RICCOBENE E, SCANDURRA P. Modeling and analyzing MAPE-K feedback loops for self-adaptation. Proc. of the IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, 2015: 13–23.

[24] RZADCA K, FINDEISEN P, SWIDERSKI J, et al. Autopilot: workload autoscaling at google. Proc. of the 15th European Conference on Computer Systems, 2020: 16.

[25] GitHub-source code. https://github.com/CloudControlSystems/ResourceAllocation.

[26] LEE K, PATON N W, SAKELLARIOU R, et al. Adaptive workflow processing and execution in Pegasus. Concurrency and Computation: Practice and Experience, 2009, 21(16): 1965–1981.

[27] ISLAM S, KEUNG J, LEE K, et al. Empirical prediction models for adaptive resource provisioning in the cloud. Future Generation Computer Systems, 2012, 28(1): 155–162.

[28] MAO Y, YAN W F, SONG Y, et al. Differentiate quality of experience scheduling for deep learning inferences with docker containers in the cloud. IEEE Trans. on Cloud Computing, 2022. DOI: 10.1109/TCC.2022.3154117.

[29] YIN L, LUO J, LUO H B. Tasks scheduling and resource allocation in fog computing based on containers for smart manufacturing. IEEE Trans. on Industrial Informatics, 2018, 14(10): 4712–4721.

[30] HU S H, SHI W S, LI G H. CEC: a containerized edge computing framework for dynamic resource provisioning. IEEE Trans. on Mobile Computing, 2022. DOI: 10.1109/TMC.2022.3147800.

[31] CHANG C C, YANG S R, YEH E H, et al. A Kubernetes-based monitoring platform for dynamic cloud resource provisioning. Proc. of the IEEE Global Communications Conference, 2017. DOI: 10.1109/GLOCOM.2017.8254046.

[32] MAO Y, FU Y Q, GU S W, et al. Resource management schemes for cloud-native platforms with computing containers of docker and Kubernetes, 2020. https://arxiv.53yu.com/abs/2010.10350.

[33] CHAKRABORTY J, MALTZAHN C, JIMENEZ L. Enabling seamless execution of computational and data science workflows on hpc and cloud with the popper container-native automation engine. Proc. of the 2nd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC, 2020: 8−18.

[34] WAIBEL P, HOCHREINER C, SCHULTE S, et al. ViePEP-C: a container-based elastic process platform. IEEE Trans. on Cloud Computing, 2019, 9(4): 1657–1674.

[35] SCHULTE S, HOENISCH P, VENUGOPAL S, et al. Introducing the vienna platform for elastic processes. Proc. of the

International Conference on Service-Oriented Computing, 2013: 179–190.

[36] HOENISCH P, SCHULLER D, SCHULTE S, et al. Optimization of complex elastic processes. IEEE Trans. on Services Computing, 2015, 9(5): 700–713.

[37] KEPHART J O, CHESS D M. The vision of autonomic computing. Computer, 2003, 36(1): 41–50.

[38] Pegasus. Workflow gallery. https://pegasus.isi.edu/workflow_gallery.

[39] Kubernetes. Configure quality of service for pods. https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod.

[40] GitHub-source code. https://github.com/CloudControlSystems/OOM-Test.

Biographies

SHAN Chenggang was born 1982. He received his M.S. degree in computer applied technology from Qiqihr University, China, in 2007. He is working toward his Ph.D. degree with the School of Automation, Beijing Institute of Technology, Beijing, China. 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.

E-mail: uzz_scg@163.com

WU Chuge was born in 1993. She received her B.E. degree in automatic control from Tsinghua University, Beijing, China, in 2015, and her M.S. and Ph.D. degrees in control theory and its applications from Tsinghua University, Beijing, China, in 2021. She was a visiting scholar with the University of Sydney, NSW, Australia, in 2018. She is currently a assistant processor for the School of Automation, Beijing Institute of Technology. Her current research interests include the scheduling and optimization theory and algorithms for cloud computing, fog computing systems, real-time scheduling, and evolutionary algorithms.

E-mail: wucg@bit.edu.cn

XIA Yuanqing was born in 1971. He received his Ph.D. degree in control theory and control engineering from Beijing University of Aeronautics and Astronautics, Beijing, China, in 2001. From January 2002 to 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 National University of Singapore as a research fellow, where he worked on variable structure control. From February 2004 to February 2006, he was with 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 School of Automation, Beijing Institute of Technology, Beijing, first as an associate professor, then, since 2008, as a professor. His research interests include networked control systems, robust control and signal processing, and active disturbance rejection control.

E-mail: xia_yuanqing@bit.edu.cn

GUO Zehua was born in 1985. He received his B.S. degree from Northwestern Polytechnical University, Xi'an, China, M.S. degree from Xidian University, Xi'an, China, and Ph.D. degree from Northwestern Polytechnical University. He was a research fellow at the Department of Electrical and Computer Engineering, Tandon School of Engineering, New York University, New York, NY, USA, and a research associate at the Department of Computer Science and Engineering, University of Minnesota Twin Cities, Minneapolis, MN, USA. His research interests include programmable networks (e.g., software-defined networking, network function virtualization), machine learning, and network security.

E-mail: guo@bit.edu.cn

LIU Danyang was born in 1993. He received his B.S. degree in mathematics and information science from Shijiazhuang University, Shijiazhuang, China, in 2016, and M.S. degree from Hebei University of Science and Technology University, Shijiazhuang, China, in 2020. He is currently pursuing his Ph.D. degree in control science and engineering from the School of Automation, Beijing Institute of Technology, Beijing, China. His research interests include cloud computing and data center networks.

E-mail: liudanyang093@163.com

ZHANG Jinhui was born in 1982. He received his Ph.D. degree in control science and engineering from Beijing Institute of Technology, Beijing, China, in 2011. He was a research associate in the Department of Mechanical Engineering, University of Hong Kong, from February 2010 to May 2010, a senior research associate in the Department of Manufacturing Engineering and Engineering Management, City University of Hong Kong, from December 2010 to March 2011, and a visiting fellow with the School of Computing, Engineering & Mathematics, University of Western Sydney, Sydney, Australia, from February 2013 to May 2013. He was an associate professor in the Beijing University of Chemical Technology, Beijing, from March 2011 to March 2016, a professor in the School of Electrical and Automation Engineering, Tianjin University, Tianjin, from April 2016 to September 2016. He joined Beijing Institute of Technology in October 2016, where he is currently an tenured professor. His research interests include networked control systems and composite disturbance rejection control.

E-mail: zhangjinh@bit.edu.cn