The Journal of Supercomputing (2023) 79:10300–10334
https://doi.org/10.1007/s11227-023-05077-7

Cost-efficient scheduling algorithms based on beetle antennae search for containerized applications in Kubernetes clouds

Hongjian Li[1] · Jie Shen[1] · Lei Zheng[2] · Yuzheng Cui[1] · Zhi Mao[1]

Accepted: 20 January 2023 / Published online: 5 February 2023

Abstract
With the development of cloud-native technologies, Kubernetes becomes the standard of fact for container scheduling. Kubernetes provides service discovery and scheduling of containers, load balancing, service self-healing, elastic scaling, storage volumes, etc. Although Kubernetes is mature with advanced features, it does not consider reducing the cost in Kubernetes clouds using the factor of communication frequent between pods while scheduling pods, nor does it have a rescheduling mechanism to save cost. Hence, we propose a cost-efficient scheduling algorithm and a rescheduling algorithm to reduce the cost of communication-intensive and periodically changing web applications deployed in Kubernetes, respectively. Network communication-intensive pods are scheduled to the same node by the scheduling algorithm based on Improved Beetle Antennae Search. According to the changing pod intimacy relationship, the rescheduling algorithm is completed through the replacement of new and old pods to reduce the cost. In addition, this paper evaluates the proposed algorithms in terms of cost and performance on a Kubernetes cloud. The result shows that the cost consumption of Kubernetes clusters in cloud environment is reduced by 20.97% on average compared with the default Kubernetes framework.

Keywords Kubernetes · Cost-efficient · Scheduling · Rescheduling · Containers

✉ Hongjian Li
lihj@cqupt.edu.cn

[1] Department of Computer Science and Technology, Chongqing University of Posts and Telecommunications, Chongqing 400065, China

[2] China Telecom Corporation Limited Chongqing Branch, Chongqing, China

1 Introduction

Docker container technology, as a new generation of virtualization technology, promotes the development of cloud computing. Compared with virtual machines, Docker containers have features such as the lightweight, portable deployment of containers across platforms, and component reuse [1–3]. However, the more the number of containers increases, the more difficult container management becomes [4–6].To solve this problem, cloud service providers (CSP) provide open-source management systems for containers, such as Google's Kubernetes [7], Docker's Swarm [8], and Apache's Mesos [9]. Among them, Kubernetes provides service discovery, load balancing, service self-healing, and elastic scaling for the entire life cycle of containers [7]. Mesos is complex and requires a customized framework when scheduling resources for specific jobs. [9]. Swarm is lightweight, architecturally simple, and suitable for small to medium-sized clusters. Compared with Kubernetes, it still has many limitations and instabilities [8]. These differences have led to the continuously rising popularity and adoption of Kubernetes in CSP and internet giants.

However, Kubernetes remains with limited cost consumption management policies that mainly focus on nodes' configuration, whereas pods' intimacy relationship is usually ignored. For instance, when the pod is scheduled, Kubernetes mostly considers nodes that satisfy the stable operation of the pods [10–12]. This may cause the nodes to generate fragmented resources, resulting in increased costs. In addition, Kubernetes clusters mostly consist of offsite nodes. The network communication between offsite nodes also increases the cost of Kubernetes clusters. Therefore, it is of great significance to get a cost-efficient scheduling strategy for cloud computing by optimizing the scheduling process of Kubernetes.

Recently, a lot of literature focusing on cost-efficient scheduling strategies have been widely studied on the deployment of containerized applications for Kubernetes [13–19]. For example, Zhong et al. [13] proposed a heterogeneous task allocation strategy (HTAS) for cost-efficient container orchestration through resource utilization optimization and elastic instance pricing. First, it supports heterogeneous job configurations to optimize the initial placement of containers into existing resources by task packing. Then it adjusts cluster size to meet the changing workload through autoscaling algorithms. Finally, it shuts down underutilized VM instances for cost saving and reallocates the relevant jobs without losing task progress. However, this strategy cannot start the closed nodes in time when the pods need to be expanded, which will affect the service experience. Rodriguez et al. [14] proposed a comprehensive container resource management algorithm. It optimizes the initial placement of containers and scales dynamically cluster resources. Ambati et al. [15] designed TR-Kubernetes. It optimizes the cost of executing mixed interactive and batch workloads on cloud platforms using transient VMs. Ding et al. [16] proposed a novel combined scaling method called COPA. Based on the collected microservice performance data, real-time workload, expected response time, and microservice instances scheme at runtime, COPA uses the queuing network model to calculate a combined scaling scheme that aims to minimize the default cost and resource cost. Zhang et al. [17] realized the model extraction of scheduling module of Kubernetes. And the K8S scheduling model is improved by combining ant colony algorithm and particle swarm optimization algorithm. Finally, it is scored, and the node with the smallest objective function is selected to deploy the pod. The experimental results show that the proposed algorithm reduces the total resource cost and the maximum load of the node and makes the task assignment more balanced. Zhang et al. [18] considered the container image pulling costs, the workload network transition costs from the clients to the container hosts and the host energy costs. To reduce cost, it employed the integer linear programming model to solve the native container scheduling problem. Zhu et al. [19] proposed a bi-metric approach to scaling pods by taking into account both CPU utilization and utilization of a thread pool. It addressed the problem that horizontal pod autoscaler (HPA) of Kubernetes may create more pods than actually needed.

However, these studies mainly focused on scheduling methods by reducing the number of worker nodes, adjusting the resource allocation of the cluster, introducing transient VMs, and allocating appropriate resources to containers. They did not consider cost-efficient scheduling of containerized applications from a perspective of the relationship of communication traffic between pods. And it is difficult to reduce the cost of Kubernetes clouds while ensuring service experience.

To address this issue in Kubernetes clouds, there are three key concerns: dynamicity of workload demand, the relationship of communication traffic between pods, and cost-efficient model. To minimize the costs under a Kubernetes cluster, it is required for the Kubernetes system to make proper decisions during scheduling and rescheduling. Therefore, this paper proposes a cost-efficient scheduling algorithm (CE-K8S) and a rescheduling algorithm using

seasonal autoregressive integrated moving average (SARIMA) based on the historical access log (RS-BHAL) for cost-efficient container orchestration to achieve cost optimization. Our work makes the following three key contributions:

- A cost-efficient model is built for Kubernetes, including the energy cost of CPU, memory, and network consumed while the pod is running and the cost of network communication between the offsite nodes.
- Based on the cost-efficient model, the scheduling algorithm is proposed based on the improved Beetle Antennae search (IBAS) and assigns pods with high network communication traffic between pods to the same node as much as possible.
- This paper describes a rescheduling algorithm that adopts the SARIMA technique to enable the replacement of new and old pods for the purpose of cost saving while ensuring service experience.

The rest of the paper is organized as follows. Section 2 describes the related work. Sections 3 and 4 introduce the proposed model and algorithm. Section 5 provides the evaluation and analysis, followed by the conclusion in Sect. 6.

2 Related work

There has been a significant amount of studies in the area of performance-oriented scheduling in Kubernetes, such as improving resource utilization [20–23] and ensuring cluster load balancing [24–26], etc.

Some studies focus on improving resource utilization of container scheduling [20–23]. The Ursa framework is proposed for container resource allocation, which makes use of the resource negotiation mechanism between job schedulers and executors [20]. To improve resource utilization and reduce task container runtime, it enables the scheduler to capture accurate resource demands dynamically from the execution runtime and to provide timely, fine-grained resource allocation based on monotasks. A new Docker controller is proposed for different I/O types, I/O access pattern, and I/O size of container applications. The controller decides the optimal batches of simultaneously operating containers in order to minimize total execution time and maximize resource utilization. It alleviates the problem of container resource contention and waiting when large I/O tasks and multiple containers write I/O at the same time [21]. A container-based resource management framework for data-intensive cluster computing, called BIG-C, is proposed. The framework devises two types of preemption strategies: immediate and graceful preemptions, and further develops job-level and task-level preemptive policies as well as a preemptive fair share cluster scheduler. It reduces queuing delays for short jobs and critical jobs and improves resource utilization [22]. An approach of managing container memory allocations dynamically is proposed to address the problem where resource utilization is low due to allocating large memory for containers to guarantee the demand at spike moments. By frequently adjusting the amount of memory reserved for each container during execution, this autonomous approach aims to increase the average number of containers that can be hosted on a server [23].

There are some studies focusing on load balancing of container scheduling [24–26]. Cai et al. [24] proposed a new priorities stage strategy, which takes the storage and network bandwidth of task nodes into account, then calculates the weight of each resource, and finally puts it into the scoring formula as the basis of resource scheduling. Thus, the load of the task nodes in the cluster is more balanced. Lin et al. [25] established a multi-objective optimization model for the container-based microservice scheduling, and proposed an ant colony algorithm to solve the scheduling problem. The algorithm considers not only the utilization of computing and storage resources of the physical nodes but also the number of microservice requests and the failure rate of the physical nodes. To improve the selection probability of the optimal path, a quality evaluation function is also used. Aruna et al. [26] proposed a new algorithm called ant colony optimization-based lightweight container (ACO-LWC) load balancing scheduling algorithm, which achieves load balancing of the cluster by scheduling various process requests.

There are some studies focusing on other aspects of container scheduling. An elastic scaling mechanism based on load prediction is proposed for PaaS cloud platforms. For periodic load changes, a series of time series of resource use are obtained by applying historical information of operation, and then the Fourier transform is used to synthesize each time period. Then the Fourier transform formulas of each time period are compared to find the pattern characteristics between different time series, so as to make a long-term prediction. Then the mechanism uses the obtained prediction to schedule containers elastically [27]. A scheduling approach called Caravel that provides better experience to stateful applications in dealing with load spikes is proposed. It allows the applications to overstep the resource request during a burst and use the resources on the same node while minimizing their evictions. Moreover, the scheduler provides a fair opportunity to all the stateful applications to use the spare resources in the cluster [28]. A new architecture for geographic orchestration of network intensive software components is proposed. It automatically selects the best geographically available computing resource within the SDDC according to the developed QoS model of the software component. It also uses both similarity matching of services and time-series nearest neighbor regression to predict resource demand to ensure the QoS of services [29].

A comparison of existing works is shown in Table 1. Most of existing resource scheduling methods do not reduce the cost of Kubernetes clouds while ensuring service experience. However, this paper considers cost consumption, service experience and workload prediction.

3 Problem formalization

This section enumerates assumptions for container-based applications and cloud resources, followed by the problem definition.

3.1 System model

It assumes a Kubernetes homogeneous cluster deployed on a cloud platform as a service provider. Cloud applications are multi-copy situation web-based systems and are charged by the configuration of the requested resources (CPU, memory,

Table 1 Comparison of related works

Works	Service experience	Workload prediction	Workload
Zhong et al. [13]	×	×	Long-running and batch jobs
Rodriguez et al. [14]	×	×	Long-running and batch jobs
Ambati et al. [15]	✓	×	Mixed interactive and batch jobs
Ding et al. [16]	✓	×	Long-running service
Zhang et al. [17]	×	×	Long-running service
Zhu et al. [19]	×	×	Long-running service
The proposed methods	✓	✓	Long-running service

network, disk I/O, etc.). The motivation is to ensure the high availability of multiple replicas and relatively optimal total costs for offsite nodes. Hence, our assumptions are listed next.

For web systems, pods are assumed to need to meet the following two requirements:

1. The same type of pod copy set cannot be placed on the same node.
2. The sum of pod affinity on all nodes must meet the maximum. The node resources can fully satisfy the needs of pods. The pods of all types are placed under the above first requirement. The resources required by pods on a node will not exceed the limit of the node.
3. There are no hard scheduling constraints defined in pods configurations.
4. Pods could be migrated without progress loss.
5. Each task may have dependencies on others. Task structure is directed acyclic graph.

The nodes are deployed offsite, i.e., different servers are selected between multiple regions to form a cluster. The multi-copy web system architecture of Kubernetes is shown in Fig. 1. All nodes are homogeneous, i.e., each node has the same hardware and software resources, and pods are only deployed on the worker nodes.

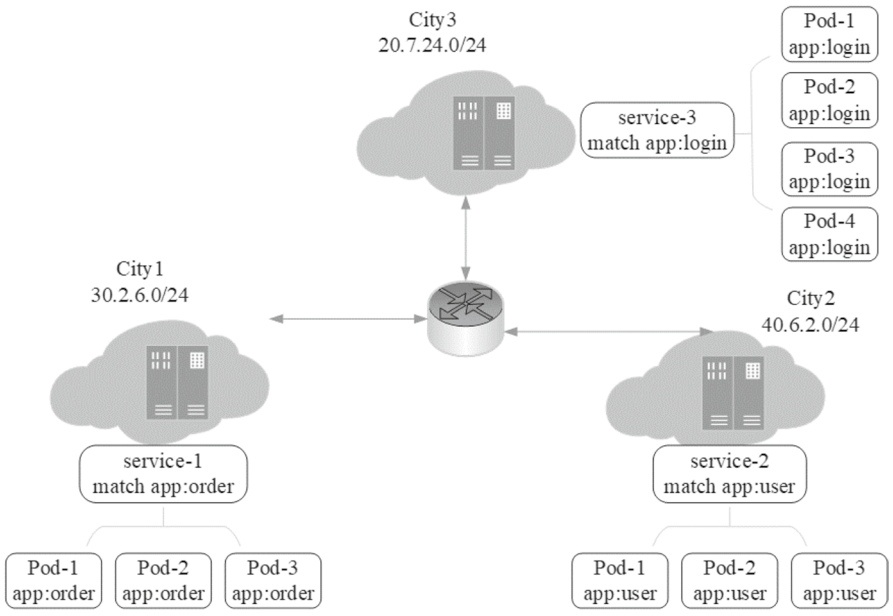

Fig. 1 The multi-copy web system architecture of Kubernetes

3.2 Problem definition

For the cost-efficient model, the set of nodes of the Kubernetes cluster is $Nodes = \{Node_1, Node_2, \cdots, Node_m\}$, the set of pod types is $Pods = \{Pod_1, Pod_2, \cdots, Pod_n\}$, where n and m represent the number of pod types and the number of nodes, respectively.

In the case of multi-copy deployment, the number of copies for each type of pods is not necessarily the same. Therefore, all pods are defined as the following formula 1:

$$Pods = \begin{bmatrix} Pod_{1,1} & Pod_{1,2} & \cdots & Pod_{1,k1} \\ Pod_{2,1} & Pod_{2,2} & \cdots & Pod_{2,k2} \\ \vdots & \vdots & \ddots & \vdots \\ Pod_{m,1} & Pod_{m,2} & \cdots & Pod_{m,kn} \end{bmatrix} \tag{1}$$

where $Pod_{m,kn}$ is the knth copy of the pod of type m. $k1$, $k2$, kn are the number of each type of pod, respectively. The value of ki needs to be greater than or equal to 2 in order to meet the high availability case. When different copies are located in different nodes, if one node fails, the other replica located in the other node can also guarantee to provide normal functions to users.

Service access to pods in Kubernetes clusters is done using Round-robin (RR). When the network traffic between pods of different types is to be counted, the traffic between each replica of pods of different types is counted. Thus, the network traffic between the pod of ith type and the pod of jth type is expressed as the following formula 2:

$$NetFlowPod(Pod_i, Pod_j) = \sum_{\lambda=1}^{ki} \sum_{\varphi=1}^{kj} NetFlow(Pod_{i,\lambda}, Pod_{j,\varphi}) \tag{2}$$

where $NetFlow$ represents the network traffic between two copies, $NetFlowPod$ is the network traffic between two pod types. $Podi$ is the pod of the type i, and $Podi, \lambda$ is λth copy of the pod of the type i. Because the network communication between pod on the same node is not forwarded by the network card, that is, the forwarding cost of the network card is not required, if $(Podi, \lambda, Podj, \varphi)$ on the same node, as shown in the following formula 3:

$$NetFlow(Pod_{i,\lambda}, Pod_{j,\varphi}) = 0 \tag{3}$$

The pod set with communication relationship of Pod_i is called $APod_i$, namely pod set with intimate relationship, then the total traffic $NetFlowPods(Pod_i)$ of pod of type i is the following formula 4:

$$NetFlowPods(Pod_i) = \sum_{Pod_\varsigma \in APod_i} NetFlowPod(Pod_i, Pod_\varsigma) \tag{4}$$

All pods traffic is expressed as the following formula 5:

$$NetFlow_{total} = \frac{1}{2} \sum_{p \in Pods} NetFlowPods(p) \tag{5}$$

The collection of CPU resources and memory resources consumed for pods is represented as the following formula 6:

$$RCM = \{(cpu_{b1}, mem_{b1}), (cpu_{b2}, mem_{b2}), \cdots, (cpu_{bm}, mem_{bm})\} \tag{6}$$

where *bi* is the number of one of the pods. The total CPU resources consumed by pods working are expressed as the following formula 7:

$$CPU_{total} = \sum_{r \in R\ C\ M} \zeta_r^{cpu} \tag{7}$$

The total memory resources consumed by pods are expressed as the following formula 8:

$$MEM_{total} = \sum_{r \in R\ C\ M} \zeta_r^{men} \tag{8}$$

According to the quantification standard of the energy consumption model in the data center [30], the total energy consumption cost is expressed as the following formula 9:

$$\begin{aligned} EneryCost_{total} = (C0 + C1 \cdot\ CPU_{total} + C2 \cdot\ MEM_{total} \\ +C3 \cdot\ NetFlow_{total}) \cdot\ Eprice \end{aligned} \tag{9}$$

where $C0$ is the constant, $C1$ is the coefficient of CPU cost, $C2$ is the cost coefficient of *MEM*, $C3$ is the cost coefficient of network communication, and *Eprice* is the price of electricity. The total energy cost is calculated by computing the utilization for each resource, including CPU usage CPU_{total}, memory usage MEM_{total}, and communication traffic usage $NetFlow_{total}$.

The cost of network communication between heterogeneous nodes is expressed in the following formula 10:

$$Nodecon = NetFlow_{total} \cdot\ Bprice \tag{10}$$

The total cost can be expressed as the following formula 11:

$$Cost_{total} = Nodecon + EneryCost_{total} \tag{11}$$

The price of power and network bandwidth are shown in Table 2:

Table 2 Price of electricity and network bandwidth

Name	Price
Electricity	0.34¥/kw.h
Network bandwidth	0.8¥/GB

4 Orchestration algorithms

The primary goal in this paper is to reduce the cost consumption of the Kubernetes cluster in two ways: (1) by considering pod intimacy relationship to optimize the initial placement of communication-intensive containers. (2) by rescheduling to enable replacement of new and old pods.

4.1 Scheduling algorithm

To solve the containerized application placement problem, we propose a cost-efficient scheduling algorithm (CE-K8S) based on the Improved beetle antennae search (IBAS) described in Sect. 4.1.1.

4.1.1 Improved beetle antennae search (IBAS)

(1) Beetle antennae search.

Inspired by the searching behavior of longhorn beetles, Jiang and Li [31] proposed a new algorithm called beetle antennae search algorithm (BAS), in 2017. It imitates the function of antennae and the random walking mechanism of beetles in nature, and then, two main steps of detecting and searching by considering the odors of food are implemented. The odors of food are an object function. The position of the beetle is a solution to the objective function. For long-running services, the scheduling process could be regarded as an offline version of the bin-packing problem. Solving the bin packing problem allows beetles to iterate and walk by the searching operation and the detecting operation.

First, to better describe the model of the BAS, x^t represents a vector of the position of the beetle at tth time instant (t = 1, 2,...,n), $f(x)$ represents a fitness function that describes the concentration of odors at position x. f_{best} denotes the denotes maximum of the concentration of odors. x_{best} denotes the position of beetle with f_{best}.

Second, to model the searching behavior, a random direction of beetle searching is expressed as follows formula 12:

$$\vec{b} = \frac{rand(h, 1)}{\|rand(h, 1)\|} \tag{12}$$

where $rand(\cdot)$ represents a random function that generates a vector of h-dimensional random values. Furthermore, the searching behaviors of both right-hand and left-hand sides, respectively, to imitate the activities of the beetle's antennae are proposed as follows formula 13:

$$\begin{cases} x_r = x^t + d^t \cdot \vec{b} \\ x_l = x^t - d^t \cdot \vec{b} \end{cases} \tag{13}$$

where x_r denotes a position lying in the searching area of right-hand side, and x_l denotes that of the left-hand side. d^t is the sensing length of antennae corresponding to the exploit ability at tth search.

Third, to formulate the behavior of detecting, iterative model as follows to associate with the odor detection by considering the searching behavior is generated as follows formula 14:

$$x^{t+1} = x^t + step^t \cdot \vec{b} \cdot sign(f(x_r) - f(x_l)) \tag{14}$$

where $sign(\cdot)$ represents a sign function. $step^t$ is the step size of tth search.

The update rules of $step^t$ and d^t are presented as follows formula 15:

$$\begin{cases} step^{t+1} = \alpha \cdot step^t \\ d^{t+1} = \beta \cdot d^t + 0.01 \end{cases} \tag{15}$$

where α and β are all variables which need to be set up for specific application scenarios.

The update rules of f_{best} are presented as follows formula 16:

$$f_{best} = min(f(x^{t+1}), f(x^t)) \tag{16}$$

where $min(\cdot)$ can be replaced by $max(\cdot)$ depending on the application scenario.

(2) Improved Beetle antennae search.

Although the principle of BAS is simple and easy to understand, it relies heavily on the setting of the α. If α is set too large, beetle may quickly jump out of the local search, and the local extrema will not be explored sufficiently, and better solutions will be missed. If α is set too small, the local extrema will be explored excessively and the local optimum cannot be jumped out as well as the convergence speed is too slow to find the global optimum. AS a result, we improve the update rules Eq. 15 of $step^t$ of the BAS as follows formula 17 :

$$step = step + \alpha \cdot \frac{|gradf(x^{t+1})| - |gradf(x^t)|}{||gradf(x^{t+1})| - |gradf(x^t)||} \tag{17}$$

where $gradf(x)$ represents the gradient. The improved BAS is called IBAS.

4.1.2 Cost-efficient scheduling algorithm (CE-K8S)

The scheduling algorithm that incorporates the IBAS bin packing algorithm in this paper is named CE-K8S. In Kubernetes, the process of scheduling pods to nodes is a bin-packing problem with intimacy. The IBAS is a meta-heuristic algorithm. Compared with traditional solution methods, meta-heuristic algorithms have the advantages of fast convergence speed, higher average performance, and better results, and are more suitable for large-scale high-dimensional bin-packing problems [32]. The intimacy relationship between pods is reflected in the applications as the size of network traffic, and the IBAS bin packing algorithm is used to solve this intimacy packing problem. This paper represents the affinity set between pod types *PodsIntimacies* as the following formula 18:

$$PodsIntimacies = \begin{bmatrix} 0 & PI(pod_1, pod_2) & \cdots & PI(pod_1, pod_m) \\ PI(pod_2, pod_1) & 0 & \cdots & PI(pod_2, pod_m) \\ \vdots & \vdots & 0 & \vdots \\ PI(pod_m, pod_1) & PI(pod_m, pod_2) & \cdots & 0 \end{bmatrix} \tag{18}$$

where $PI(podi, podj)$ represents the affinity between pod of the type i and pod of the type j. Pods of the same type do not communicate with each other, so their closeness is 0. Pods of different types that do not communicate with each other also have a closeness of 0.

In solving NP-hard problems like the bin packing problem [33], the dimension of the algorithm search needs to be set to a two-dimensional matrix of the product of the number of pod types and the number of nodes, and in each update it is only necessary to update the coordinates of the algorithm search normally, and then convert the direction generated randomly each time to a two-dimensional matrix of the product of the number of pod types and the number of nodes.

The bin packing algorithm designed using IBAS is shown in Algorithm 1. IBAS bin packing algorithm first initializes the various parameters of the improved Beetle Antennae Search algorithm (line 1-2). Then the algorithm initializes the search dimension (called Dim) using a product of the number of nodes and the number of pod types (line 3). Then a one-dimensional matrix is created by combining the type name of pods and the number of each type (line 4). Then an initial position vector of the Dim dimension of beetles is created using random placements (line 5). After creating the initial position vector of beetles, an initial direction vector of beetles of the Dim dimension is created using Eq. 12 (line 7). Based on the direction vector, as shown in Eq. 13, the vector of the left and right beards is created. Then, the vector of left and right beards is converted into a two-dimensional matrix (line 8). Afterward, the matrix is sorted in descending order, and then the subscripts of the row elements in the matrix are output (line 9). Then, according to the number of copies of each pod, the corresponding elements are taken from the matrix (line 10). Then the elements are mapped to bags (line 11). If pods of the same type are on the same node, a new iteration is started (line 12–14). Then the intimacy between pods in each node in the cluster is calculated. The intimacy on each node is summed to get the total intimacy (called Maxfit) (line 15). Based on the total intimacy at tth iteration, the location of Beetle was updated using Eq. 14 (line 16). Then Maxfit is updated using Eq. 16 (line 17). Then depending on where the Maxfit is located, the placement strategy of the pods is updated (line 18). Then the step length of Beetle is updated using Eq. 17 (line 19). Then Iteration number minus one (line 20). Finally, the placement strategy with the optimal intimacy is obtained (line 22).

Algorithm 1 Pseudo-code of the IBAS bin packing algorithm

```
Input: Pods, nodes, PodsIntimacies, $x^0$, $d^0$, $\delta^0$ , t, nodesNum, PodsTypeNum,
    iter, Maxfit.
Output: The best solution to solve multiple knapsack problem.
 1: Initialize $x^0$, $d^0$, $\delta^0$ , t.
 2: Initialize nodesNum, PodsTypeNum, iter, Maxfit.
 3: Dim $\leftarrow$ nodesNum multiply PodsTypeNum
 4: pods $\leftarrow$ combine PodsType and the number of each type to generate a
    one-dimensional matrix
 5: randomly generate a dim dimension vector to represent the initial position
    of beetles.
 6: while iter $> 0$ do
 7:     randomly generate a vector of dim dimension to represent the direction
    of search
 8:     reshape the vector of the left and right beard into an array of
    PodsTypeNum multiplied by nodesNum
 9:     argosort each row of the array
10:     take out the corresponding elements from the array according to the
    number of pods of each type
11:     box by pods set and the results obtained above
12:     if same pod in the same node then
13:         continue
14:     end if
15:     calculate the Maxfit of total node
16:     determine the next position of beetle by the value of left and right
    longicorn.
17:     update Maxfit
18:     update pods placement policy
19:     change step of beetle
20:     iter = iter - 1
21: end while
22: Return the best initialization placement
```

The CE-K8S scheduling algorithm can be described by following Algorithm 2. CE-K8S algorithm first deploys web tasks using the default scheduling algorithm, and then user requests for web tasks are simulated (line 1). Then the communication traffic between each type of pod is fetched using Mizu (the API traffic viewer for Kubernetes) (line 2). Then the set of pod affinity is constructed by the communication traffic using Eq. 18 (line 2). Based on the set of pod affinity, the IBAS bin packing algorithm 1 is performed (line 3). Then the relative best deployment solution between pods and nodes regarding cost is obtained (line 3). Finally, pods scheduling starts to be implemented (line 4–14).

The architecture of CE-K8S is shown in Fig. 2. After obtaining the placement relationship with the highest closeness, the CE-K8S is implemented through the Kubernetes plug-in for the web system. The CE-K8S can be described as follows: First, the default scheduling algorithm is used to obtain the intimacy relationship between pods, and the cost-effective deployment scheme of pods is obtained through IBAS, and then the deployment of pods is realized through a custom scheduling plug-in, and if the deployment fails, it is deployed through the default scheduling algorithm.

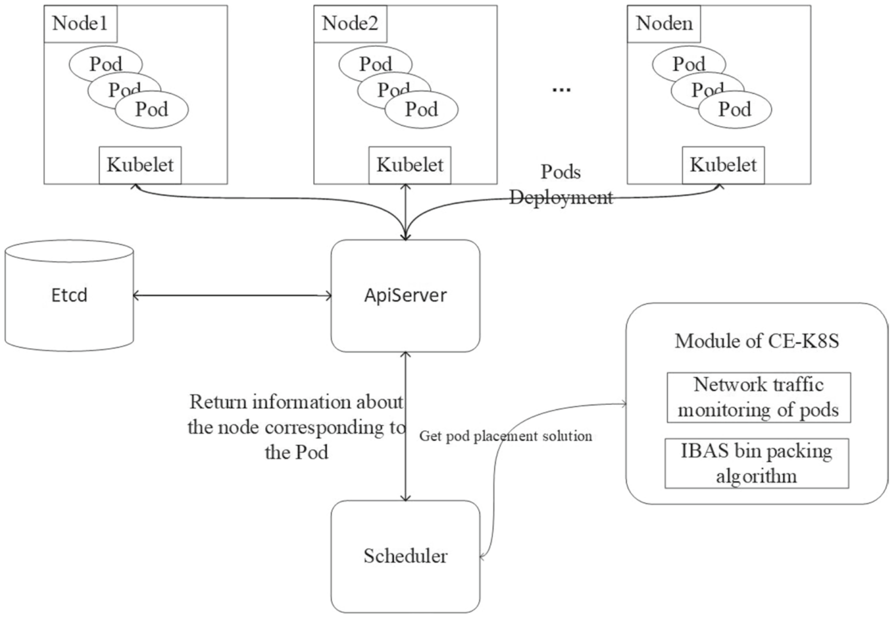
Node1
Pod
Pod
Pod
Kubelet
Node2
Pod
Pod
Pod
Kubelet
...
Noden
Pod
Pod
Pod
Kubelet
Pods Deployment
Etcd
ApiServer
Module of CE-K8S
Network traffic monitoring of pods
IBAS bin packing algorithm
Return information about the node corresponding to the Pod
Get pod placement solution
Scheduler

Fig. 2 CE-K8S working architecture

Algorithm 2 Pseudo code of CE-K8S algorithm

Input: Queue of pods to be scheduled, set of nodes
Output: Success or failure

```
1: Place pods by default scheduler and run to get the network traffic between
   each pod.
2: Get PodsIntimacies by network traffic
3: ce-Placement ← run IBAS Packing
4: for pod in pods do
5:     node ← get node that pod should be scheduled from ce-Placement
6:     if bind(pod, node) then
7:         Continue
8:     else
9:         schedule pod by default algorithm
10:        if schedule failed then
11:            return false
12:        end if
13:    end if
14: end for
15: return true
```

4.2 Rescheduling algorithm

The reason why pods are rescheduled is that after Kubernetes schedules the pods, the relationship between pods and nodes is bound, and this binding will continue until the Pods are deleted. However, as the business changes Kubernetes clusters may face a situation: if only the cluster business changes, the current resources can still meet the business needs. The deployment location of pods can be reasonably adjusted according to the business changes to further reduce the cost of using the cluster.

The architecture of RS-BHAL is shown in Fig. 3. First, the resource monitoring system and API traffic monitoring system are built in the cluster. This set of monitoring mainly consists of the resource data collection component Node Exporter, the monitoring component Prometheus, the visualization tool Grafana, the log visualization platform Kibana, the data storage platform ElasticSearch and the Kubernetes cluster API traffic monitoring tool Mizu, which monitors various resource changes in the Kubernetes cluster and communication between all pods in real time. This system uploads the data to the RS-BHAL rescheduling module, which

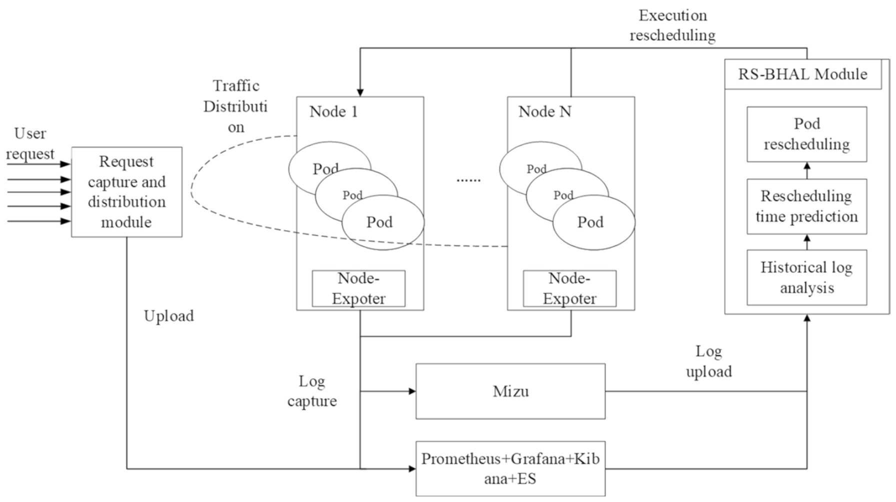

Fig. 3 RS-BHAL working architecture

analyzes the available data to decide whether and when to initiate the rescheduling algorithm.

Applications deployed on Kubernetes have periodic workloads. The periodic history of accesses to the cluster APIs by external traffic can be queried from ElasticSearch. This record is analyzed to obtain the cyclical changes of the business and the periodic access to the APIs and the cyclical access. The specific steps are.

1. Get the set of periodic changes in pod closeness about time from the historical log records (line 1).
2. Get the mapping of the current pod and node deployment and the set of pod affinities (line 1).
3. The pod-node mapping relationship is obtained by the default scheduling algorithm (line 2).
4. The current set of pod affinity cycle changes is used to obtain the new pod-node mapping relationship by IBAS (line 3).
5. Get the cost of the default and new pod to node mapping relationships by pod affinity (line 4-5).
6. Compare the cost of the two to decide whether to perform rescheduling. If executed, it returns the deployment mapping relationship between pod and node (line 6–10).

The pseudo code of historical log analysis algorithm is shown in Algorithm 3.

Algorithm 3 Pseudo code of historical log analysis algorithm

```
Input: History Logging
Output: Whether to perform rescheduling, the new set of periodic mapping
    relations between pod and node
1: PodsIntimacies $\leftarrow$ The set of periodic changes in closeness between pods
2: defaultRe $\leftarrow$Get the mapping relationship between pod and node of the
   default scheduling algorithm
3: NewRe = CE-K8S(PodsIntimacies)
4: defaultCost = Cost effectiveness about defaultRe
5: NewCost = Cost effectiveness about NewRe
6: if NewCost $>$ defaultCost then
7:     return true, NewRe
8: else
9:     return false, NULL
10: end if
```

To ensure that the impact on the cluster services is minimized during rescheduling, the rescheduling timing should be chosen at the point when the periodic changes occur, because this is the time when the user requests are the smallest, otherwise it will cause damage to the services experience.

SARIMA is one of the time series forecasting analysis methods. Because the periodic variation of a web system is also a kind of time series variation by nature, the seasonal autoregressive integrated moving average model (SARIMA) is used to predict the periodic variation of the web system.

The main steps of the SARIMA(p,d,q)(P,D,Q,s) model can be summarized as follows.

1. Time series sample smoothness test and processing, this process is mainly to determine whether the data sample is with smoothness, if not, then use the difference processing to smooth the data.
2. Model selection and parameter estimation, a process used to determine the corresponding values of each parameter in the SARIMA(p,d,q)(P,D,Q,s) model, which mainly uses ACF and PACF.
3. Model accuracy assessment, this process is mainly used to assess the accuracy of the model.

Therefore, the algorithm for predicting the rescheduling time using the SARIMA(p,d,q)(P,D,Q,s) model can be described as the following pseudo-code shown in Algorithm 4. Rescheduling time prediction algorithm first gets the history log records (line 1). Then the variable (called times) is created to represent the set of rescheduling time points (line 2). Then request data is fetched from the history log (line 3). Afterward, request data is cleaned and filtered in order to eliminate non-user requests (line 4). Then the data is checked for compliance with smoothness requirements (line 5). Based on the data, SARIMA is used to get the predicted time points (line 6). If the data do not meet the smoothness requirement, the data are differenced (line 9).

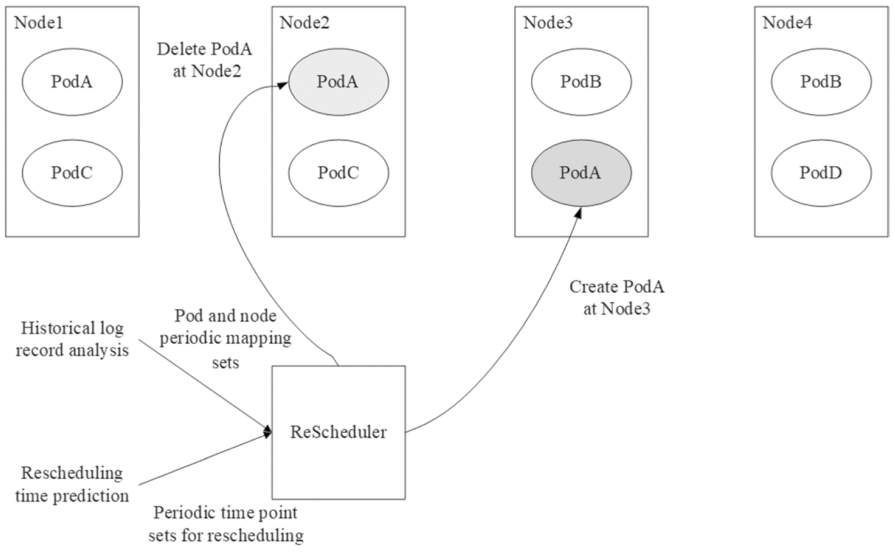
Node1
PodA
PodC
Delete PodA at Node2
Node2
PodA
PodC
Node3
PodB
PodA
Node4
PodB
PodD
Create PodA at Node3
Historical log record analysis
Pod and node periodic mapping sets
ReScheduler
Rescheduling time prediction
Periodic time point sets for rescheduling

Fig. 4 Working mode of rescheduling

Algorithm 4 Pseudo code of rescheduling time prediction algorithm

```
Input: History Logging
Output: Redispatch time point set
1:  logs ← History logs
2:  times ← Redispatch time point collection
3:  predata = extract_History_req(logs)
4:  data = clear_filter(predata)
5:  if check_Stationarity(data) then
6:      times = SARIMA_Predict(data)
7:      return times
8:  else
9:      data = differential_process(data)
10:     times = SARIMA_Predict(data)
11: end if
12: return times
```

After the historical log analysis and the rescheduling time prediction algorithm, the periodic mapping relationship between pod and node and the periodic change time point of the next phase are obtained. With these two points, a specific rescheduling algorithm can be executed, which first creates a backup of the pod at the destination node and then deletes the old pod at the old node. The way it works is shown in Fig. 4. To avoid damaging application performance, the old pod can serve users normally while the new one is being created.

The pseudo-code of rescheduling algorithm is shown in Algorithm 5.

Algorithm 5 The pseudo-code of rescheduling algorithm

```
Input: The pseudo-code of rescheduling algorithm
1: newRes ← pod to node periodic mapping relationship
2: times ← Rescheduling periodic time point sets
3: for res in newRes do
4:     reschedulerlists = get_per_reflect_of_pod(res)
5:     for list in reschedulerlists do
6:         pod_node_times = getargetNodeandtime(list, times)
7:         for reschepod in pod_node_times do
8:             if createNewPod(reschepod) then
9:                 deleteOldPod(reschepod)
10:            else
11:                continue
12:            end if
13:        end for
14:    end for
15: end for
16: Return
```

5 Performance evaluation

To compare the cost and performance of the proposed algorithms (CE-K8S and RS-BHAL) with related algorithmic works, including the default K8s framework, Tabu, BFD and Descheduler, we implemented the proposed algorithms and carried out the empirical evaluations by deploying experiments on a Kubernetes cloud.

CE-K8S Scheduling Evaluation: Sect. 5.2 compares CE-K8S with three other containerized application scheduling approaches in terms of the cost and performance using Workload (called Workload 1). These approaches are selected to solve the containerized application scheduling problem within Kubernetes clouds.

1. The default K8s framework—Containerized application scheduling problem is solved from the perspective of balanced resource allocation.
2. BFD—One of containerized application scheduling approaches proposed in [13], where containerized application scheduling (initializing placement of pods within Kubernetes clouds) and cost are the focus of scheduling decision-making.
3. Tabu—Algorithm which is meta-heuristic is the same as the type of BAS (Beetle Antennae search). We use this to demonstrate how the incorporation of improved BAS(IBAS) results in better placement decisions.

The default K8s framework, BFD, and Tabu are the works that can be adapted and applied to the scheduling of containerized applications addressed in our work. So, they are chosen for the comparison of CE-K8S in terms of cost and performance.

RS-BHAL Rescheduling Evaluation: Rescheduling is live migration. Sect. subsec5.3 compares RS-BHAL with two other containerized application rescheduling approaches in terms of the cost using Workload (called Workload 2). These approaches are selected to solve the containerized application rescheduling problem within Kubernetes clouds.

1. The default K8s framework—We use the prediction model proposed in this paper to change the scheduling algorithm of the default K8s framework to a rescheduling algorithm. It gets a containerized application migration solution from the perspective of balanced resource allocation using our workload prediction. We use this to demonstrate how the consideration of the relationship of communication traffic between pods and workload prediction results in better migration decisions in terms of cost.
2. Descheduler—It is a popular Kubernetes sub-project, where balanced resource allocation is the focus of rescheduling decision-making.

The default K8s framework and Descheduler are the works that can be adapted and applied to the rescheduling of containerized applications addressed in our work. So, they are chosen for the comparison of RS-BHAL in terms of the cost.

5.1 Experimental setup

Workload 1: A company's communication-intensive web system, consisting of seven microservices, has a set of copies of each service and the required resources as shown in Table 3.

Workload 2: A company's web system with periodicity. Because of user habits, the system faces an increase in the number of users' requests every day from around 8:00 am to 1:00 pm. In order to reduce the size of the table, entries of the table represent the number of HTTP requests received by the web application every three hours from Monday to Friday (requests-per-three hour), as shown in Table 4.

This experiment relies on the cloud platform to build a Kubernetes homogeneous cluster, which consists of one master node and four worker nodes. All the nodes in the cluster are considered offsite deployments, and the network communication between them needs to be costed. Each worker node deployed on VMs has 8 G CPU cores and 16GB RAM. The operating system of each worker node is Centos 6.5. Since they are connected to one network backbone, the number of hops a packet traverses from source to destination is between 12 and 14 hops [34]. Therefore, different network distances do not affect site selection.

Other software and hardware parameters are shown below.

Software: GoLand2019, PyCharm2021, Go1.16, Python3.6.

Network plug-in: Flannel.

Table 3 Web system

Microservice name	Number of copies	Request CPU	Request mem	Limit CPU	Limit men
A	3	500m	400Mi	800m	500Mi
B	2	300m	300Mi	500m	400Mi
C	3	400m	400Mi	600m	500Mi
D	3	400m	400Mi	600m	500Mi
E	2	300m	300Mi	500m	400Mi
F	2	300m	300Mi	500m	400Mi
G	3	500m	400Mi	800m	500Mi

Table 4 History log

	0 o'clock	3 o'clock	6 o'clock	9 o'clock	12 o'clock	16 o'clock	19 o'clock	21 o'clock
Monday	101492	55622	228896	914642	647972	731655	492698	261837
Tuesday	92841	57676	247896	1197639	782043	1088065	531711	266626
Wednesday	115460	88990	366659	1296082	1060030	1121163	606155	293199
Thursday	113074	87180	335241	1313634	815463	1125282	573913	253010
Friday	102002	81394	308106	1294758	829717	1150607	537227	231938

5.2 Workload 1

In this section, 100, 500, 1000, 2000, 3000, 5000, 10000, 15000, 20000 concurrent accesses to the web interface are performed using the Kubernetes default scheduling algorithm, BFD bin packing algorithm, forbidden search algorithm (Tabu), and CE-K8S, respectively. Ten experiments are performed for each request, and the average CPU usage, average memory usage, average request time, and average network communication traffic between nodes are counted, and finally the cost of each algorithm is calculated according to the cost formula defined above.

The average CPU usage of the cluster under different concurrent requests is shown in Fig. 5. The reason why the CPU utilization of the different algorithms is compared is that the total energy cost (Eq. 9) is calculated. When the number of concurrent requests is 100, 300 and 1000, the difference between the four algorithms is not very obvious, because the number of concurrent requests is not very high at this time and the overall CPU consumption is not large. The CE-K8S algorithm can save up to 0.9% CPU cost and down to 0.2% CPU cost compared to other algorithms. As the number of concurrent requests increases, the effect of the CE-K8S algorithm starts to show up, but the overall difference is still not very large. At 20,000 concurrent requests, the CPU utilization of the BFD starts to be higher than that of the default scheduling algorithm, probably because the deployment strategy of BFD at this point causes most of the frequently communicating pods to belong to different nodes, which cannot withstand the high concurrency scenario. CE-K8S has the lowest average CPU usage compared to other algorithms, i.e., it can complete the same task with lower resource consumption, saving energy costs.

The average memory usage for each algorithm of the cluster at different concurrent request volumes is shown in Fig. 6. The reason why the memory utilization of the different algorithms is compared is that the total energy cost (Eq. 9) is calculated. At the concurrent request volumes of 100, 300, and 1000, the difference in the effectiveness of the four algorithms is not very obvious, and the CE-K8S saves an average of 0.8% memory overhead compared to the other algorithms, with the lowest saving of 0.4% memory overhead. As the number of concurrent requests increases, the effect of the CE-K8S algorithm starts to show up, thanks to the fact that the CE-K8S algorithm schedules pods with higher network traffic on the same node and reduces the data sent to another node, thus reducing the memory buffer usage and memory.

Figures 7 and 8 represent the average request time and the average inter-node traffic for each algorithm of the cluster under different concurrent request volumes, respectively. Formula 5 is used in Fig. 8. As can be seen from the figures, the CE-K8S algorithm is able to complete the specified number of concurrent requests in the shortest time compared to other scheduling strategies, while minimizing the communication traffic between nodes. Compared to the default scheduling algorithm, CE-K8S reduces the request time by 5.3% on average, and the inter-node network traffic by 35.3%. This is due to the fact that CE-K8S crates the pods reasonably according to their communication volume, so that the pods with high network traffic are on the same node.

Figure 9 shows the number of reboots for each algorithm at different concurrency levels, and the number of reboots reflects the robustness of the service, and the smaller the number of reboots, the higher the robustness of the service. As can

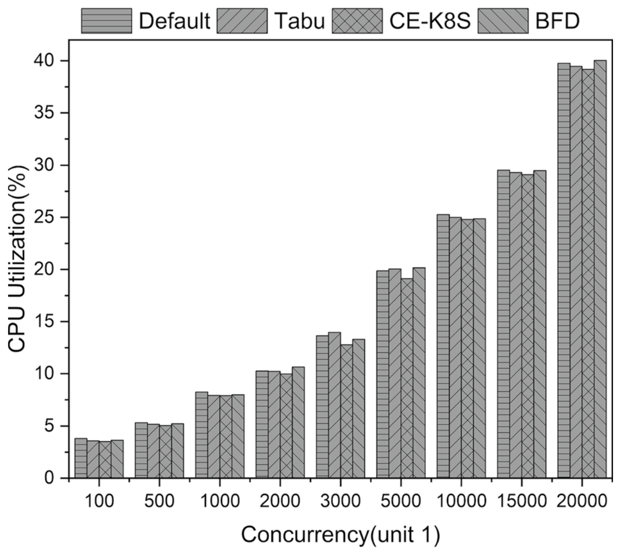

Fig. 5 Comparison of average CPU usage

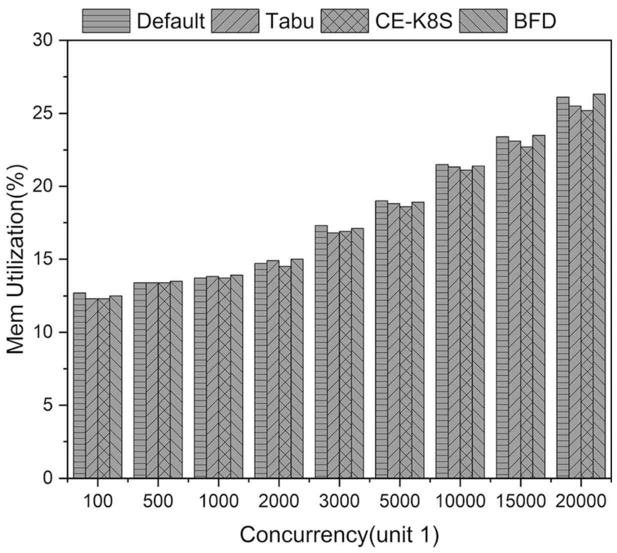

Fig. 6 Comparison of average memory usage

be seen from Figure 9, before the concurrency is 5000, the number of reboots of pods for each algorithm is 0, and the service is equally robust. After the concurrency is greater than 5000, the advantages and disadvantages of the four algorithms start to emerge, where the default scheduling algorithm is the highest at different subsequent concurrency, the CE-K8S algorithm is the lowest at all. Therefore, the CE-K8S scheduling algorithm can improve the stability of the system by scheduling closely communicating pods to a node.

Figure 10 represents the number of request failures for each algorithm at different concurrent volumes. As can be seen from the figure, the CE-K8S algorithm has the lowest number of request failures under different concurrent request volumes. This is due to the fact that the CE-K8S schedules the frequently communicating pods on the same node, which enables them to complete network requests faster. Another reason is that the CE-K8S algorithm has the lowest number of pod restarts for different request concurrency, this is known from Fig. 9, some requests may just happen during the process of pod hang restarts, so the CE-K8S algorithm has the lowest number of request failures for different concurrency. This reflects that the CE-K8S algorithm can improve the stability of the cluster service from the other side.

Figure 11 shows the energy cost of the cluster at different concurrent request volumes. When the concurrent request volume is less than 1000, the difference in energy cost of the four algorithms is not very obvious, because the cluster can withstand the concurrent volume at this time and the overhead on CPU and memory is relatively small. When the volume of concurrent requests reaches 3000, the reason why CE-K8S has a higher energy overhead than Tabu search is that as shown in Fig. 6, when the volume of concurrent requests reaches 3000, CE-K8S has a higher memory usage than Tabu search. As the number of concurrent

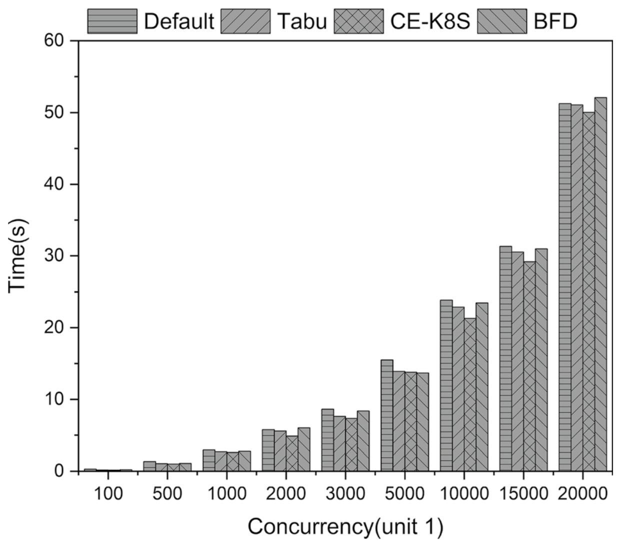

Fig. 7 Comparison of average request completion time

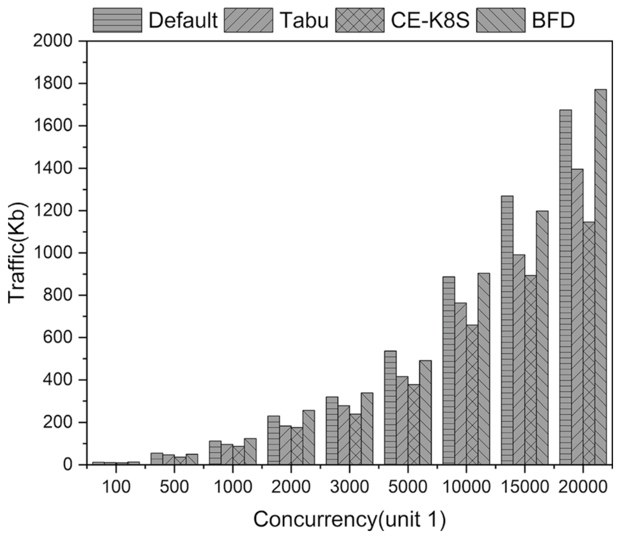

Fig. 8 Comparison of average network traffic between nodes

requests increases, the effect of CE-K8S starts to appear gradually, and compared with the default scheduling algorithm, Tabu and BFD algorithms, CE-K8S is able to save 4%, 2.7%, and 3% of energy cost overhead on average. Figure 12 represents the network bandwidth cost of communication between offsite nodes for different amounts of concurrent network requests, which is proportional to the previous inter-node network traffic and saves an average of 35.3% network bandwidth

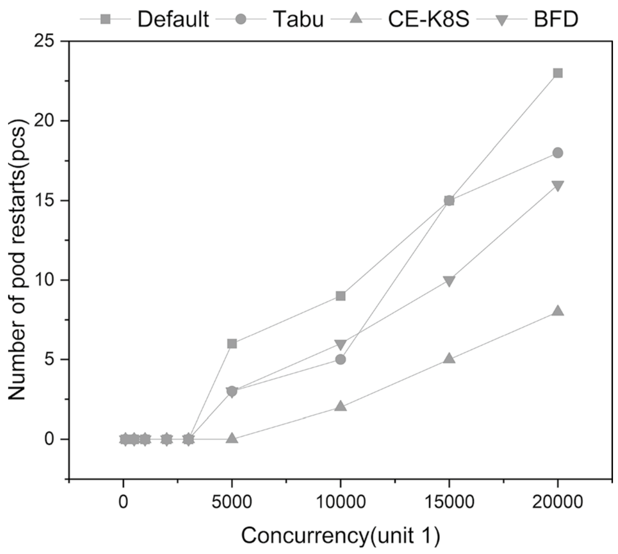

Fig. 9 Comparison of number of pod restarts

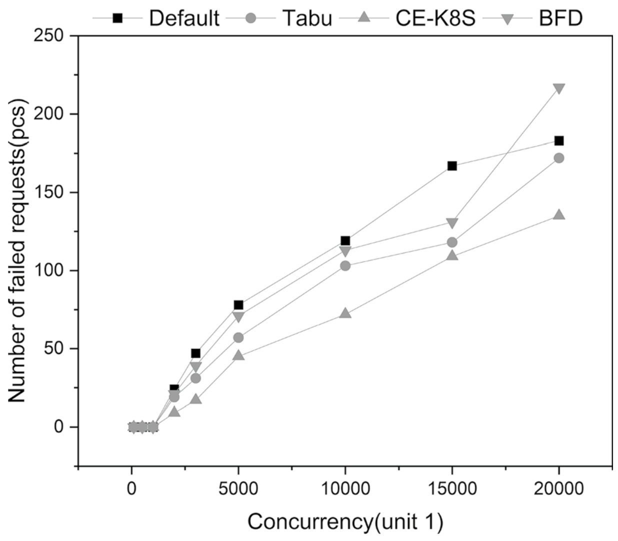

Fig. 10 Comparison of number of request failures

cost compared to the default scheduling algorithm. Formula 11 is used in Fig. 13. Figure 13 represents the total cost of the cluster, and CE-K8S reduces the cluster cost by 20.97%, 8.82%, and 21.13% on average compared to the other three algorithms. It shows that the CE-K8S proposed in this paper can save the energy cost and network bandwidth cost of the cluster by scheduling the frequently communicating pods to one node when targeting the web system on a Kubernetes cluster composed of offsite nodes.

5.3 Workload 2

5.3.1 Prediction of rescheduling time

To verify whether the system meets the requirements of SARIMA time series forecasting, it is necessary to make a test on the periodic log change data of the system.

(1) Data stability test and parameter determination.

The stability test was first performed on the data from July 1 to 5, 2021, and the original data is shown in Fig. 14. The unit root test is shown in Fig. 15. If the pvalue is large, the data need to be differenced, and the first-order differencing process is shown in Fig. 16. By analyzing the correlation between ACF and PACF in Fig. 15, we can only roughly determine the range of p and q within the second order, so we need to further determine the values of p and q by the AIC information criterion, and the smaller the value of AIC information criterion means the more accurate the model is fitted.

Because it can be considered smooth after first-order differencing, $d = 1$, the value of s is set to 24 according to the period, and the value of D is 0. Therefore, only the parameter values of p, q, P, and Q need to be determined. Here, the four parameters are combined in ranges, and the best combination of parameters is determined by AIC through the SARIMAX package to exclude the combinations that cannot converge. The AIC values for each combination are shown in Table 5.

As can be determined from Table 5, Akaike information criterion (called AIC) is a measure of the degree of adaptation of a statistical mode. The smaller the AIC, the

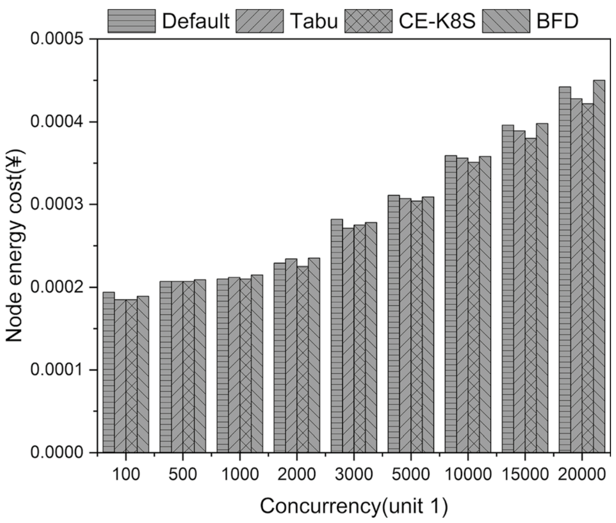

Fig. 11 Comparison of cluster energy cost

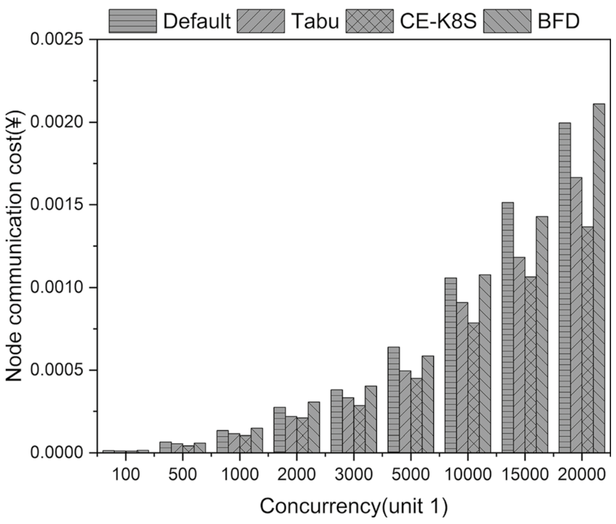

Fig. 12 Comparison of communication cost of offsite nodes

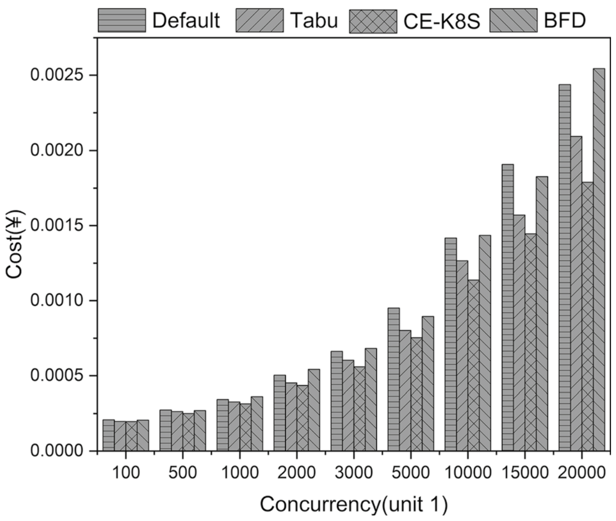

Fig. 13 Comparison of total cost of cluster

better the model. The model with the smallest AIC is usually chosen. The minimum AIC value for the combination of parameters (0,0,1,0) is 1051.950286. Therefore, it is determined that the best fitting parameter is (0,0,1,0). At this point, all the parameters required for the SARIMA model are all determined, and the model will be tested below.

(2) Model Testing and Forecasting.

As shown in Fig. 17, the residual terms conform to a normal distribution, indicating that the model meets the requirements.

Using the data from July 1 to 5 to make predictions, the results are shown in Fig. 18. July 6 data is the predicted data, and the trend of the predicted data is in line with the overall change of the data and meets the requirements. In this paper, the predicted data is compared with the real data on July 6, and the error of the two data is about 50,000, which is about 1% and acceptable compared with the daily data

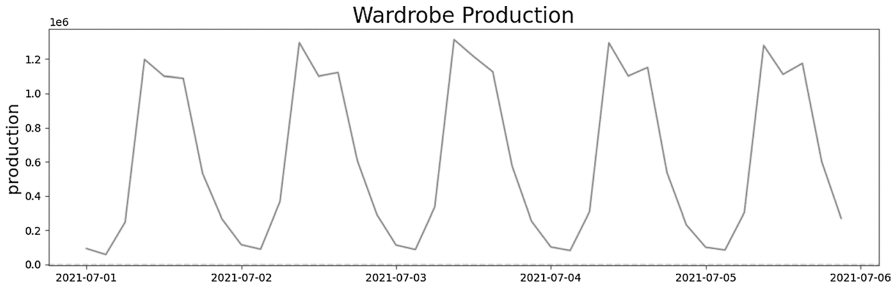

Fig. 14 Original data

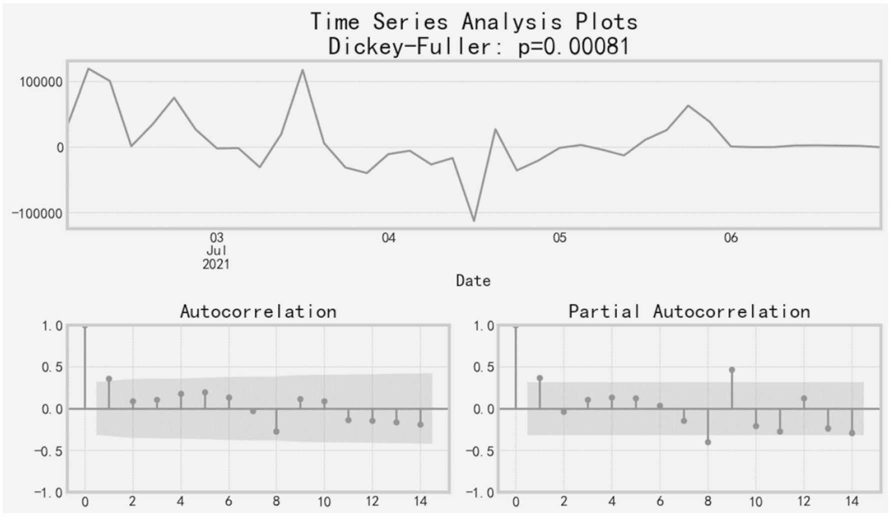

Fig. 15 Unit root test and ACF and PACF

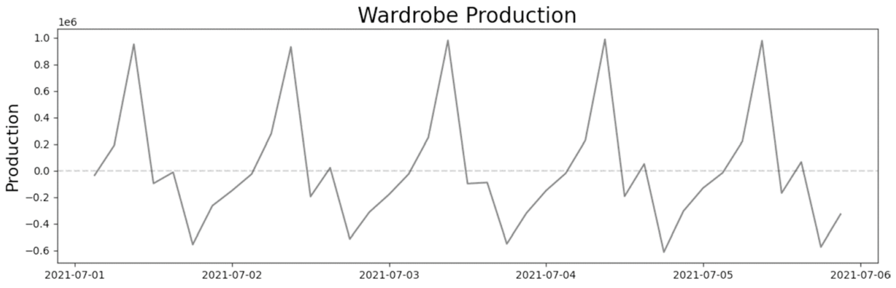

Fig. 16 First-order difference

records of nearly 5 million. Meanwhile, the difference of cycle change time is within 1 minute, which is acceptable. It indicates that SARIMA is suitable for this system to predict the cycle change.

5.3.2 RS-BHAL rescheduling

The six hours of the 9:00–15:00 time period with more obvious cycle changes were selected, while the business in the morning and the business in the afternoon were adjusted, so the time period selected was reasonable. In terms of data volume size, 1% of the requests within each minute were randomly selected as the experimental replay. As for the statistics of the experimental results, the statistics are conducted every minute, and the comparison algorithm is chosen from the rescheduling algorithm (Descheduler) of the open source community and the default scheduling algorithm of Kubernetes.

For the web system, the resource utilization of the three scheduling policies from 9:00 am to 3:00 pm is shown in Figs. 19 and 20. The reason why the resource utilization of the different algorithms is compared is that the total energy cost (Eq. 9) is calculated. Before 10:25 a.m, CPU and memory utilization starts to increase as application accesses increase, but the difference in utilization between the three policies is not significant because no rescheduling is triggered. After 10:25, as shown in Fig. 4, the business volume starts to increase, and the RS-BHAL starts to execute rescheduling based on the rescheduling time predicted by the previous history logs. The CPU and memory utilization of RS-BHAL algorithm suddenly increases around 10:20, indicating that pod rescheduling is being executed at this time. However, the increase in the CPU and the memory utilization of the RS-BHAL algorithm is temporary, because the backups of pods are created at the destination node during rescheduling, and then the old pods at the old node are deleted. After rescheduling around 10:25 to 12:00, the CPU and memory utilization of RS-BHAL algorithm is obviously smaller than the default scheduling and Descheduler algorithm. At around 1:30 p.m. when the workload is adjusted again, the RS-BHAL is triggered again and the pods are rescheduled according to the affinity. The CPU and memory utilization thereafter is smaller than the default scheduler algorithm most of the time. At the same time, the CPU and memory utilization suddenly increase during rescheduling, which is due to the fact that the backups of pods are created during rescheduling, which causes the CPU and memory overhead to increase. Compared with the default scheduling algorithm and Descheduler, RS-BHAL reduces CPU utilization by

Table 5 Parameter combinations and their corresponding AIC

Parameters	AIC
(0, 0, 1, 0)	1051.950286
(2, 3, 1, 0)	1051.950286
(3, 3, 1, 0)	1118.461490
(2, 3, 1, 1)	1119.191850
(1, 3, 1, 0)	1121.378024

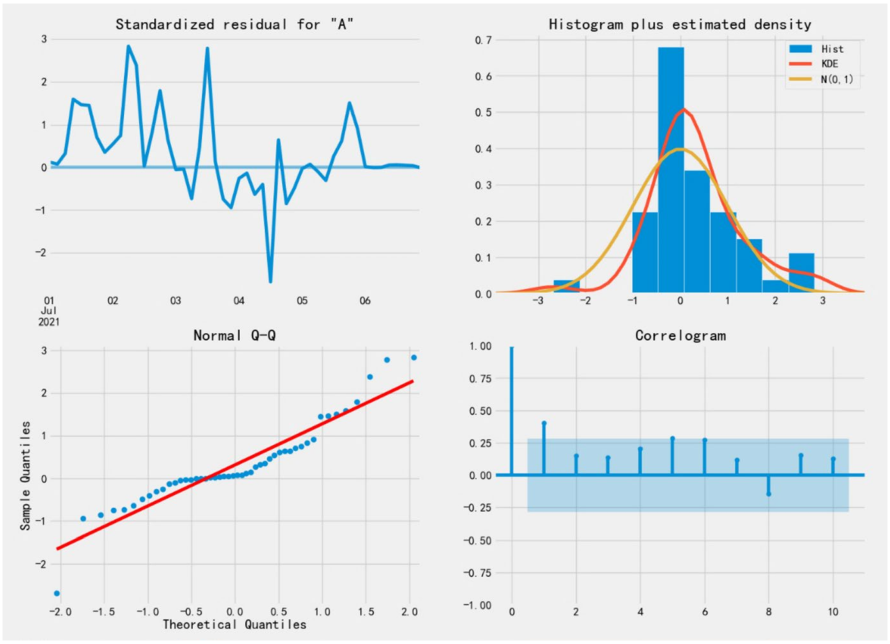

Fig. 17 Model diagnosis

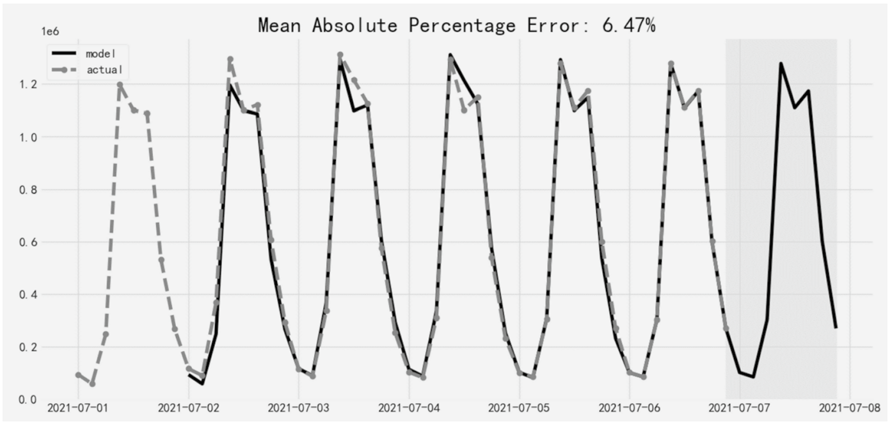

Fig. 18 Model prediction

7.08% and 4.66%, and memory by 7.49% and 5.1%, respectively, during the entire request period.

The network changes between pods corresponding to the three algorithms throughout the historical request rescheduling period are shown in Fig. 21. Formula 5 is used in Fig. 21. It can be seen that the RS-BHAL can reduce the network communication between pods after the execution of rescheduling, but a surge

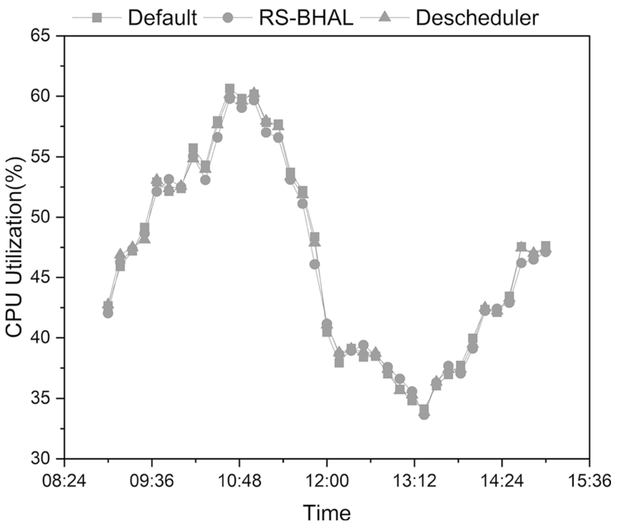

Fig. 19 Comparison of CPU utilization

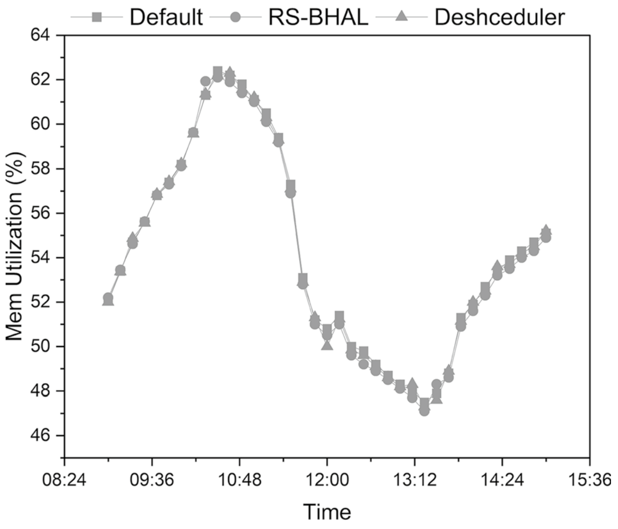

Fig. 20 Comparison of memory utilization

of communication traffic between pods occurs when the rescheduling is specifically executed. This is due to the fact that when the old pods are deleted during the rescheduling, the new pods have not yet taken over the traffic transferred from the old pods, resulting in a decrease in the number of available pods and an increase in the communication traffic between pods. Once the rescheduling is completed and the new pods start to take over the traffic, the communication traffic between pods will start to decrease. Through data comparison, RS-BHAL reduces the inter-node network traffic by 5.65% and 4.7% compared with the default scheduling algorithm and Descheduler, respectively, during the whole experimental period, which laterally reflects that the intimacy gap between pods of this system is not large, i.e., the network traffic between each other is roughly equivalent. It indicates that the RS-BHAL rescheduling strategy can reduce the network communication traffic between cluster business pods.

The average energy cost incurred by the RS-BHAL and the other two algorithms during the whole experimental period is shown in Fig. 22. The RS-BHAL reduces the energy cost by 5.8% and 4.8%, respectively, compared to the other two algorithms. Figure 23 represents the network bandwidth cost of the three strategies, from which it can be seen that RS-BHAL reduces 5.7% and 3.67%, respectively, compared to the other two algorithms, although the network communication volume of this system is relatively high, the network bandwidth cost saved by RS-BHAL algorithm for this system is not very high, the reason is that the affinity between pods of this system is approximately the same, i.e., the network traffic between each type of pod is approximately the same. Formula 11 is used in Fig. 24. Figure 24 represents the total cost of the three algorithms, and RS-BHAL reduces 5.59% and 4.7%, respectively, compared to the other two algorithms. The difference shows that the RS-BHAL proposed in this paper can reduce the cost by periodically rescheduling the pod in conjunction with the business.

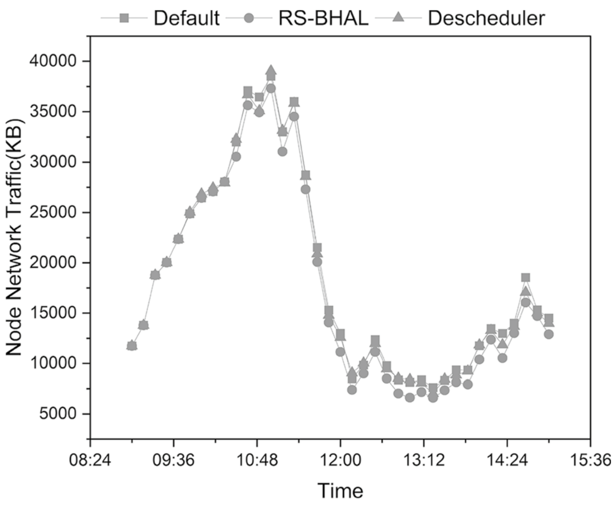

Fig. 21 Comparison of network communication traffic

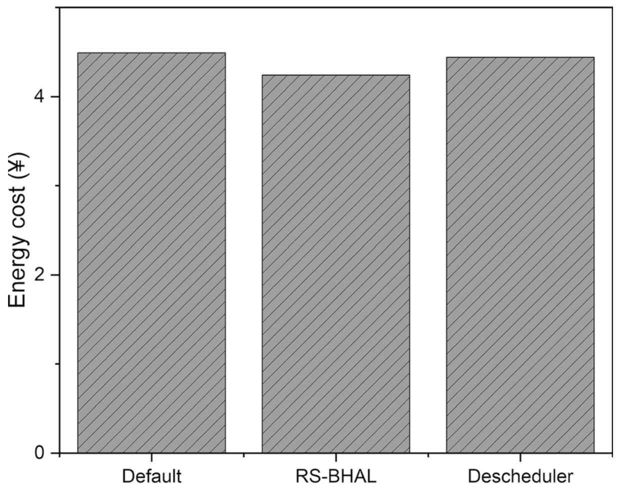

Fig. 22 Comparison of energy cost of different algorithms

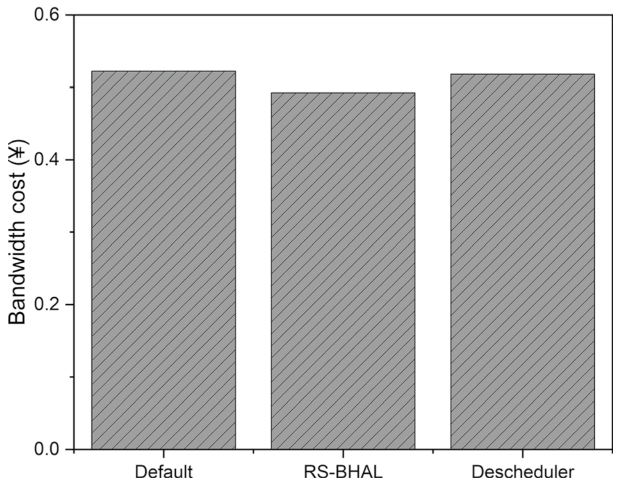

Fig. 23 Comparison of bandwidth cost of different algorithms

6 Conclusion and future directions

In this paper, a cost-efficient scheduling algorithm (CE-K8S) and a cost-efficient rescheduling algorithm (RS-BHAL) for Kubernetes are proposed based on cloud. A new cost-efficient model is built by integrating the energy cost of CPU, memory, and network consumed while the pod is running and the cost of network communication between the offsite nodes. Based on this model, our two scheduling algorithms minimized the total cost of Kubernetes cluster while satisfying the service experience. The experiment of Kubernetes cluster using communication-intensive workload and cycle workload on the cloud shows that the proposed two algorithms can

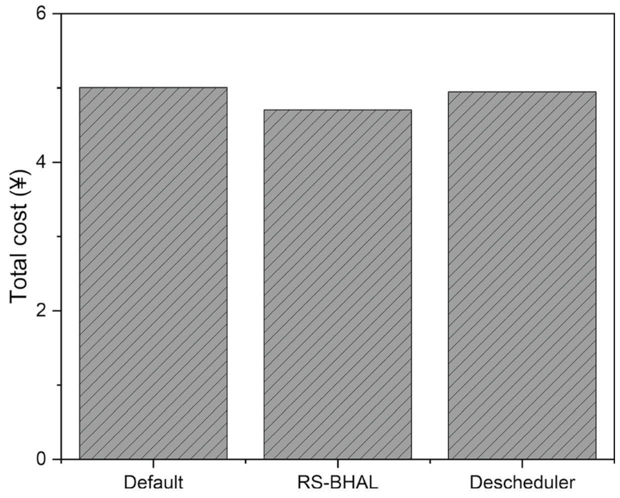

Fig. 24 Comparison of total cost of different algorithms

efficiently reduce the total cost consumption compared with the traditional scheduling algorithms. The Kubernetes scheduling framework designed in this paper still has some limitations. Due to the pursuit of lower cluster cost, compared with other performance-oriented scheduling frameworks, the algorithm pervasiveness and cluster resource balancing may not be improved too much. In the future work, we will consider the balance optimization and locality factor of cluster resources. And we will also try to improve the applicability of the algorithm on the expansion, such as focus on MySQL, Redis, and other databases.

Acknowledgements This work was supported by Chongqing science and Technology Commission Project (Grant No.: cstc2018jcyjAX0525), Key Research and Development Projects of Sichuan Science and Technology Department (Grant No.: 2019YFG0107).

Author Contributions HL: Proposed an idea, Experiment, Wrote the manuscript. JS: Proposed an idea, Experiment, Wrote the manuscript. LZ: Experiment, Helped to write also several sections of the manuscript, Proofreading. YC: Helped to write also several sections of the manuscript, Proofreading. ZM: Helped to wrote also several sections of the manuscript, Proofreading.

Data availability The datasets generated during the current study are available from the corresponding author on reasonable request.

Declarations

Conflict of interest None. The authors declare that they have no known conflict financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

1. Boettiger C (2015) An introduction to docker for reproducible research. ACM SIGOPS Oper Syst Rev 49(1):71–79
2. Wan X, Guan X, Wang T, Bai G, Choi B-Y (2018) Application deployment using microservice and docker containers: Framework and optimization. J Netw Comput Appl 119:97–109
3. Bugnion E, Devine S, Rosenblum M, Sugerman J, Wang EY (2012) Bringing virtualization to the x86 architecture with the original vmware workstation. ACM Trans Comput Syst (TOCS) 30(4):1–51
4. Kaur K, Garg S, Kaddoum G, Ahmed SH, Atiquzzaman M (2019) Keids: Kubernetes-based energy and interference driven scheduler for industrial iot in edge-cloud ecosystem. IEEE Internet Things J 7(5):4228–4237
5. Toka L, Dobreff G, Fodor B, Sonkoly B (2021) Machine learning-based scaling management for Kubernetes edge clusters. IEEE Trans Netw Serv Manage 18(1):958–972
6. Zheng S, Huang F, Li C, Wang H (2021) A cloud resource prediction and migration method for container scheduling. In: 2021 IEEE Conference on Telecommunications, Optics and Computer Science (TOCS). IEEE, pp 76–80
7. Burns B, Grant B, Oppenheimer D, Brewer E, Wilkes J (2016) Borg, omega, and Kubernetes. Commun ACM 59(5):50–57
8. Soppelsa F, Kaewkasi C (2016) Native docker clustering with swarm. Packt Publishing Ltd
9. Dubhashi D, Das A (2016) Mastering Mesos. Packt Publishing Ltd
10. Wojciechowski Ł, Opasiak K, Latusek J, Wereski M, Morales V, Kim T, Hong M (2021) Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh. In: IEEE INFOCOM 2021-IEEE Conference on Computer Communications. IEEE, pp 1–9
11. Carrión C (2022) Kubernetes scheduling: taxonomy, ongoing issues and challenges. ACM Computing Surveys (CSUR)
12. Burns B, Beda J, Hightower K (2019) Kubernetes: up and running: dive into the future of infrastructure. O'Reilly Media
13. Zhong Z, Buyya R (2020) A cost-efficient container orchestration strategy in Kubernetes-based cloud computing infrastructures with heterogeneous resources. ACM Trans Internet Technol (TOIT) 20(2):1–24
14. Rodriguez M, Buyya R (2020) Container orchestration with cost-efficient autoscaling in cloud computing environments. In: Handbook of research on multimedia cyber security. IGI global, pp 190–213
15. Ambati P, Irwin D (2019) Optimizing the cost of executing mixed interactive and batch workloads on transient VMS. Proc ACM Measur Anal Comput Syst 3(2):1–24
16. Ding Z, Huang Q (2021) Copa: a combined autoscaling method for kubernetes. In: 2021 IEEE International Conference on Web Services (ICWS). IEEE, pp 416–425
17. Wei-guo Z, Xi-lin M, Jin-zhong Z (2018) Research on Kubernetes' resource scheduling scheme. In: Proceedings of the 8th International Conference on Communication and Network Security, pp 144–148
18. Zhang D, Yan B-H, Feng Z, Zhang C, Wang Y-X (2017) Container oriented job scheduling using linear programming model. In: 2017 3rd International Conference on Information Management (ICIM). IEEE, pp 174–180
19. Zhu C, Han B, Zhao Y (2022) A bi-metric autoscaling approach for n-tier web applications on Kubernetes. Front Comp Sci 16(3):1–12
20. Jin T, Cai Z, Li B, Zheng C, Jiang G, Cheng J (2020) Improving resource utilization by timely fine-grained scheduling. In: Proceedings of the Fifteenth European Conference on Computer Systems, pp 1–16
21. Bhimani J, Yang Z, Mi N, Yang J, Xu Q, Awasthi M, Pandurangan R, Balakrishnan V (2018) Docker container scheduler for i/o intensive applications running on nvme ssds. IEEE Trans Multi-Scale Comput Syst 4(3):313–326
22. Chen W, Zhou X, Rao J (2019) Preemptive and low latency datacenter scheduling via lightweight containers. IEEE Trans Parallel Distrib Syst 31(12):2749–2762
23. Nicodemus CH, Boeres C, Rebello VE (2020) Managing vertical memory elasticity in containers. In: 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC). IEEE, pp 132–142
24. Zhiyong C, Xiaolan X (2019) An improved container cloud resource scheduling strategy. In: Proceedings of the 2019 4th International Conference on Intelligent Information Processing, pp 383–387

25. Lin M, Xi J, Bai W, Wu J (2019) Ant colony algorithm for multi-objective optimization of container-based microservice scheduling in cloud. IEEE Access 7:83088–83100
26. Aruna K, Pradeep G (2021) Development and analysis of ant colony optimization-based light weight container (aco-lwc) algorithm for efficient load balancing
27. Zhong C, Yuan X (2019) Intelligent elastic scheduling algorithms for paas cloud platform based on load prediction. In: IEEE 8th Joint International Information Technology and Artificial Intelligence Conference (ITAIC). IEEE, pp. 1500–1503
28. Deshpande U (2019) Caravel: Burst tolerant scheduling for containerized stateful applications. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS). IEEE, pp 1432–1442
29. Paščinski U, Trnkoczy J, Stankovski V, Cigale M, Gec S (2018) Qos-aware orchestration of network intensive software utilities within software defined data centres. J Grid Comput 16(1):85–112
30. Luo L, Wu W-J, Zhang F (2014) Energy modeling based on cloud data center. J Softw 25(7):1371–1387
31. Jiang X, Li S (2017) Bas: beetle antennae search algorithm for optimization problems. CoRR arXiv:abs/1710.10724
32. Hopper E, Turton BC (2001) An empirical investigation of meta-heuristic and heuristic algorithms for a 2d packing problem. Eur J Oper Res 128(1):34–57
33. Medvedeva MA, Katsikis VN, Mourtas SD, Simos TE (2021) Randomized time-varying knapsack problems via binary beetle antennae search algorithm: emphasis on applications in portfolio insurance. Math Methods Appl Sci 44(2):2002–2012
34. Van Mieghem P (2009) Performance analysis of communications networks and systems. Cambridge University Press