CCF Transactions on High Performance Computing (2022) 4:165–181
https://doi.org/10.1007/s42514-022-00096-x

REGULAR PAPER

Adaptive processing rate based container provisioning for meshed Micro-services in Kubernetes Clouds

Hang Wu[1] · Zhicheng Cai[1] · Yamin Lei[1] · Jian Xu[1] · Rajkumar Buyya[2]

Received: 23 October 2021 / Accepted: 4 March 2022 / Published online: 6 April 2022

Abstract
More and more applications are organized in the form of meshed micro-services which can be deployed on the popular container orchestration platform Kubernetes. Designing appropriate container auto-scaling methods for such applications in Kubernetes is beneficial to reducing costs and guaranteeing Quality of Services (QoS). However, most existing resource provisioning methods focus on a service without considering interactions among meshed services. Meanwhile, synchronous calls among services have different impacts on the processing ability of containers as the proportion of different business type's requests changes which is not considered in existing methods too. Therefore, in this article, an adaptive queuing model and queuing-length aware Jackson queuing network based method is proposed. It adjusts the processing rate of containers according to the ratio of synchronous calls and considers queuing tasks when calculating the impact of bottleneck tiers to others. Experiments are performed on a real Kubernetes cluster, which illustrate that the proposal obtains the lowest percentage of Service Level Agreement (SLA)-violations (decreasing about 6.33%-12.29%) with about 0.9% additional costs compared with existing methods of Kubernetes and other latest methods.

Keywords Kubernetes · Container · Micro-services · Provisioning · Jackson queuing network

1 Introduction

There is a trend of providing complex intelligent functions in Web systems which usually contain multiple collaborative micro-services. The structure of micro-services' invoking topology is meshed as shown in Fig. 1. Since containers are more lightweight and portable than Virtual Machines (VMs), deploying micro-services in containers elastically rented from public Cloud vendors (Amazon Lightsail Containers Stormacq 2020) is able to provide elastic processing capacity by renting and releasing containers dynamically. Kubernetes (abbreviated as K8s) is a popular container application orchestration and management system. Many public Cloud vendors provide container-application developing and running platforms based on K8s such as Amazon Elastic Kubernetes Service (Amazon EKS) (Amazon 2021) and Alibaba Cloud Kubernetes (ACK) (AlibabaCloud 2021). The main goal of this article is to design container auto-scaling methods for meshed micro-services deployed on such K8s-based platforms to reduce resource costs while guaranteeing Quality of Services (QoS). The main challenges of developing container provisioning algorithms are complex interactions among meshed micro-services and degeneration processing ability caused by synchronous calls. Most of existing container scheduling algorithms (Abdullah et al. 2020; Zhong and Buyya 2020; Delnat et al. 2018; Kubernetes 2021; Toka et al. 2020; Cai and Buyya 2021) for K8s based platforms are designed for batch jobs or single micro-services. Separate batch jobs or hybrid of batch jobs and Web systems are scheduled to appropriate number of Pods (containers) to decrease resource costs (Abdullah et al. 2020; Zhong and Buyya 2020). In these works, it is assumed that batch jobs and Web systems consume the fixed amount of resources. However, resource consumed by Web systems usually changes as workload changes. For single micro-services, there are some K8s-embedded resource auto-scaling

✉ Zhicheng Cai
caizhicheng@njust.edu.cn

Hang Wu
hangwu@njust.edu.cn

1 School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing, China

2 Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Melbourne, Australia

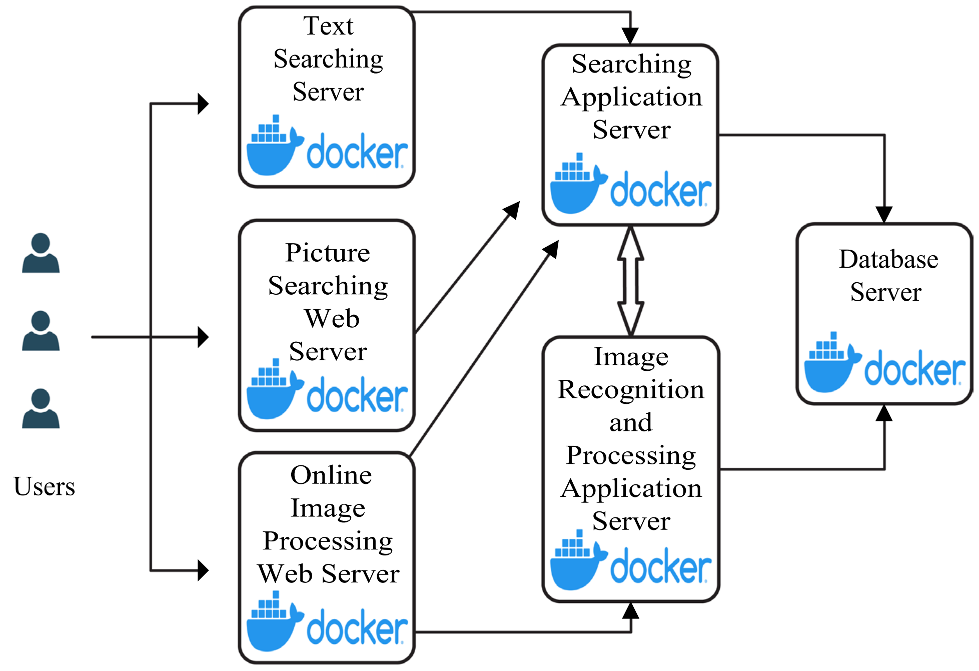
Text Searching Server
docker
Searching Application Server
docker
Picture Searching Web Server
docker
Database Server
docker
Users
Image Recognition and Processing Application Server
docker
Online Image Processing Web Server
docker

Fig. 1 An example of meshed micro-services

methods such as Horizontal and Vertical Pod Autoscaler (HPA and VPA) (Kubernetes 2021) or their extensions (Delnat et al. 2018; Toka et al. 2020) which dynamically scale resources based on threshold of Memory, CPU utilization or other metrics. However, pure threshold-based methods always lead to control delays. The hybrid of queuing and control theory is nearly able to make the average response time follow a given reference time by adjusting the number of allocated containers dynamically (Cai and Buyya 2021). These methods are all designed for single-tier systems without considering the impact among meshed multi-tiers. Allocating more containers to some bottleneck tiers will increase the speed of generating calls to other tiers leading to new bottlenecks. Bottlenecks must be eliminated by allocating more containers to multiple affected tiers for avoiding bottleneck shifting.

Synchronous invocations make the measuring of container's processing ability and identifying of bottleneck complex. Measuring processing rates appropriately is crucial to queuing models which have been widely used to determining the number of required containers for each micro-services (Melikov et al. 2018; Huang et al. 2016; Jiang et al. 2013; Lei et al. 2020). The processing rate of a micro-service's container without synchronous calls can be estimated by stress testing. However, micro-services in meshed systems need to call others to work collaboratively and synchronous invocations consume additional resources which decrease the processing ability. The request processing rate of a container usually degrades as the ratio of synchronous invocation to other micro-services increases. Therefore, it is not suitable to use a fixed estimated processing rate. Meanwhile, a meshed Web system usually supports multiple types of businesses with different invocation paths. From the perspective of a micro-service, it takes different time to wait for the returning of synchronous calls for requests from different business types, which has a great impact on the observed average response time of a micro-service's container. Consequently, it is easy to misjudge the bottleneck by comparing the average response time of the micro-service with a fixed Response Time Upper Limit (RTUL) specified by Service Level Agreement (SLA).

In this article, a hybrid of proactive and reactive method based on Jackson queuing network with adaptive processing rates and RTUL is proposed to elastically provision containers to meshed micro-services. The main contributions of our work are as follows.

1. An adaptive processing rate calculating method, which considers performance degeneration and proportion changes of different business types' requests, is developed to estimate the required number of containers accurately based on queuing models.
2. The RTUL of each micro-service is updated dynamically based on the proportion of different business types' requests to improve the accuracy of identifying bottlenecks.
3. The queuing length of each micro-service is considered to calculate impact on other services accurately.

The rest of this paper is organized as follows. Section 2 is the related work and Sect. 3 describes the problems of meshed Web systems in Kubernetes. The impact of synchronous calls on processing rate is analyzed in Sect. 4. The proposed method is introduced in Sect. 5. Sections 6 and 7 are performance evaluation on a real Kubernetes cluster, conclusion and future work.

2 Related work

The architecture of Web systems is becoming more and more complex which changes from single-tier (Cai and Buyya 2021; Toka et al. 2020) to linear multi-tier (Bi et al. 2015; Wang et al. 2019; Adam et al. 2017; Zhang et al. 2016) and then to meshed multi-tier (Lei et al. 2020; Xu and Buyya 2019) gradually. Resources allocated to Web systems might be VMs (Qu et al 2018; Lorido-Botran et al. 2014; Al-Dhuraibi et al. 2018) or containers (Adam et al. 2017; Xu and Buyya 2019; Cai and Buyya 2021; Zhong and Buyya 2020; Kho et al 2018; Toka et al. 2020; Delnat et al. 2018). For multi-tier systems, interactions among tiers can be categorized into synchronous and asynchronous calls (Chen et al 2017; Wang et al. 2019; Zhang et al. 2016; Lei et al. 2020).

Most of existing works are tailored for single-tier and linear multi-tier systems. For such simple structured systems, threshold (Kubernetes 2021; Toka et al. 2020), queuing theory (Melikov et al. 2018; Huang et al. 2016; Jiang et al. 2013), control theory (Cai and Buyya 2021; Pan et al. 2008; Cai et al. 2020; Baresi et al. 2016), and reinforcement learning (Barrett et al. 2013; Li and Venugopal 2011) have been used to allocate resources elastically. Kubernetes's embedded HPA (Kubernetes 2021) adjusts the number of Pods based on CPU or Memory utilization, which is slow in reacting to performance changes. HPA+ (Toka et al. 2020) is an extension of HPA which uses LSTM (Long Short-Term Memory) to predict the number of requests to amend the delay of reacting to performance changes. The M/M/N queuing model is used to predict minimum4 amount of resources required to obtain a mean response time smaller than a given upper limit (Huang et al. 2016). Threshold and queuing model based methods usually lack the ability of reacting to real-time deviations in terms of performance, therefore, control theory and reinforcement learning have been used to increase the reactive ability to real-time performance. For Web systems based on hourly charged VMs, UCM was proposed by Cai et al. (2020) which takes advantages of queuing model and feedback control to avoid over-control and react quickly to workload changes. For container based Web systems in Kubernetes, an inverse-queuing model based feedback control was developed by Cai and Buyya (2021) to provision containers to single-tiers separately. However, they are not designed for meshed multi-tier systems.

Only a few of existing works consider meshed multi-tier systems while most of them are not for systems with synchronous calls. Jackson queuing network (JQN) is widely used to model the impact among meshed multiple queues. For example, JPRM proposed by Lei et al. (2020) applied JQN to predict the arrival rate of each tier and impact of bottleneck tiers on others. However, the queuing length of each tier is not considered when calculating the impact of bottleneck tiers on others. Synchronous calls to other services consume some server threads and connections even if when they are waiting for the return of calls which has a great impact on the processing ability of underline Containers or VMs, and the ratio of Synchronous calls may lead to different degrees of performance degradation (Chen et al 2017; Wang et al. 2019). However, impact of synchronous calls among meshed multi-tier services is not considered in existing works.

Many algorithms are only evaluated on simulation platforms and there are deviations between real and simulation platforms. For example, JPRM (Lei et al. 2020) is only evaluated on a simulation environment established on CloudSim (Calheiros et al. 2011). UCM (Cai et al. 2020) is tested on both CloudSim and a real Kubernetes cluster. But it is for single-tier systems. ContainerCloudSim (Piraghaj et al. 2017) is a CloudSim extension which provides support for modeling and simulation of containerized Cloud Computing environments. These methods cannot simulate exactly the same as the actual production platforms. Therefore, it is necessary to design and evaluate resource provisioning algorithms for real production systems such as Docker Swarm (Xu and Buyya 2019; Magableh and Almiani 2020) and Kubernetes (Cai and Buyya 2021; Zhong and Buyya 2020; Kho et al 2018; Toka et al. 2020; Delnat et al. 2018).

A comparison between our approach and existing methods is shown in Table 1. As a whole, most of existing resource auto-scaling methods are designed for single or linear multi-tier Web systems without synchronous calls. In this article, our approach considers the queuing length of each tier when calculating impacts among meshed tiers using Jackson network, and performance degradation caused by synchronous calls. Meanwhile, our approach is evaluated on a real Kubernetes Cluster.

3 Problem description

Micro-service based architecture is beneficial to construct flexible software systems. Each micro-service provides a single function and is usually encapsulated as a RESTful or SOAP-based Web service (Cai and Buyya 2021). A Web system usually provides multiple types of businesses and each of which needs the collaboration of different combinations of micro-services. Requests of different business types have different accessing paths on the graph of meshed micro-services. Each micro-service usually has multiple containers to increase concurrent processing ability. Kubernetes is

Table 1 Comparison of our approach with existing resource provisioning algorithms for Web Systems

Works	Resource type	Objectives	Architecture	Resources provisioning strategy	Workload prediction	Platforms
JPRM (Lei et al. 2020)	VMs	VM rental costs and response time	Meshed multi-tier	M/M/N queuing-model	Holt-Winter's model	CloudSim
DCM (Chen et al 2017)	VMs	CPU-utilization and response time	Linear multi-tier	CPU utilization threshold	×	Private Datacenter
EcoWare (Baresi et al. 2016)	VMs containers	CPU-core rental time and response time	Single-tier	Control theory	×	EcoWare
BrownoutCon (Xu and Buyya 2019)	VMs containers	Power consumption	Meshed multi-tier	CPU utilization threshold	Sliding windows	Docker Swarm
HPA (Kubernetes 2021)	Containers	Response time	Single-tier	Threshold of diffrent kinds of metrics	×	Kubernetes
HPA+ (Toka et al. 2020)	Containers	Response time	Single-tier	Requests and CPU utilization threshold r	LSTM	Kubernetes
FeedBack_InverseQM (Cai and Buyya 2021)	Containers	Response time	Single-tier	M/M/N queuing model with control theory	×	Kubernetes
Proposed method	VMs containers	VM rental costs and response time	Meshed multi-tier	M/M/N queuing model with adaptive processing rate	Holt Winter's model	Kubernetes

a container-based application management platform which provides the function of creating a certain number of containers for each micro-service and deleting containers from the micro-service.

Figure 2 shows the architecture of meshed micro-services deployed in Kubernetes. The Master node is responsible for managing Kubernetes cluster. Each container is usually embedded in a Pod which is the minimum scheduling unit of K8s. Pods are deployed on Worker nodes which are VMs rented from public Clouds. Containers on the same Worker node have the same color and each micro-service have multiple containers from different Workers. TraefikLab (2021) is a front-end load balancer in charge of distributing requests to back-end containers. The Resource Auto-scaler scales resources through RESTful APIs. Logs of Traefik are collected and analyzed by the Logs Manager to obtain the mean response time, queuing length and so on for each micro-service.

Different business types usually have different mean response times because different combinations of micro-services are invoked. Figure 3 is an example of a group of business types with various accessing paths. Let $\mathbf{S} = \{S_i | i \in \{1, 2, 3 \ldots, n\}\}$ be the set of micro-services in the Web system. $b_l \in B$ is the $l-th$ business type provided by the system, and $b_l = (S_{l,1}, \ldots, S_{l,k}, \ldots)$ where $S_{l,k} \in \mathbf{S}$ is the $k-th$ accessed micro-service of b_l. For example, (S_1), (S_1, S_3, S_5) and (S_1, S_4, S_5) are three business types in Fig. 3. S_3 in $b_1 = (S1, S_3, S_5)$ means that S_3 is the second accessed micro-service in b_1. In SLA, the mean response time of each business type should be smaller than an upper limit. The objective of this article is to design container elastic provisioning algorithms for the Resources Auto-scaler to minimize resource costs while guaranteeing SLA.

4 Impact of synchronous calls

Asynchronous calls return immediately after being called so that other consequent operations can be performed while the asynchronous calls are performing simultaneously. On the contrary, consequent operations can not be executed before synchronous calls return. Connections of synchronous calls should be maintained before the return which consume additional resources and decrease the processing ability of each container.

In this article, the Jmeter (2021) is used to evaluate the impact of synchronous calls. Three same micro-services S_i, S_j and S_k which calculate the $n-th$ Fibonacci number in a recursive way, is selected as the test-bed. Three business types $b_1 = (S_i)$, $b_2 = (S_i, S_j)$ and $b_3 = (S_i, S_j, S_k)$ are tested which all need to access S_i. The objective is to obtain the Throughput Per Second (TPS) of one container of S_i under

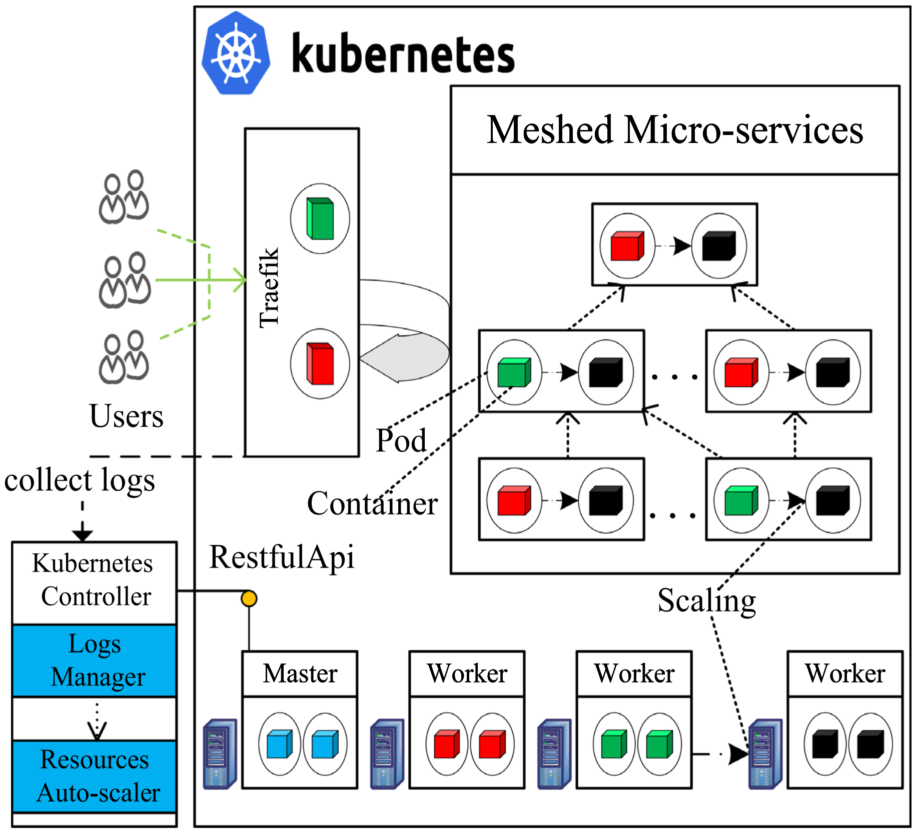

Fig. 2 Architecture of resources manager in Kubernetes

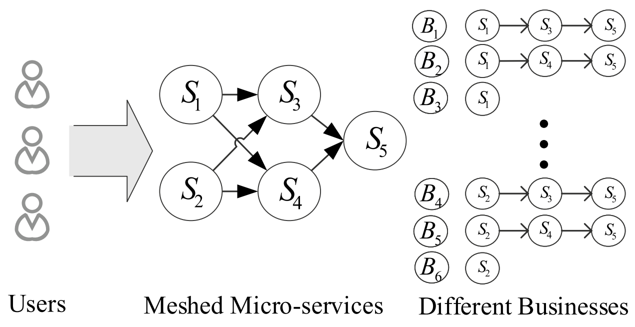

Fig. 3 An example of users acessing different types of businesses

different business types. In other words, S_i is only allocated one container with one CPU core and 500 Mi Memory, while S_j and S_k have sufficient number of containers. The speed of generating requests of each business type is increased gradually until the CPU utilization of S_i's container reaches 60%. Then, the TPS of one S_i's container of each business type is obtained. In order to evaluate the impact of the complexity of micro-services on processing ability, n of the Fibonacci micro-service takes different values from $\{23, 24, \dots, 30\}$.

Figure 4 shows the TPS of S_i's container with different parameter n and business types which illustrates that TPS decreases when the length of tasks increases and the micro-service accesses other micro-services. The TPS of one S_i's container in different business types b_2 and b_3 is the same which proves that synchronous calls decreases the processing ability (decreased TPS) of containers, but the depth of micro-services in the synchronous calls has a very little impact on the processing ability. The degradation degree of processing ability decreases as the task-length of micro-services increases. The reason is that fewer synchronous calls are generated to others because the TPS of S_i decreased when task lengths of S_i become longer. Therefore, the degradation ratio of processing ability is mainly affected by the computation requirements of the service generating calls rather than those of calls.

5 Proposed methods

In this paper, a hybrid of proactive and reactive resource provisioning architecture similar with JPRM (Lei et al. 2020) is applied. In the proactive part, the request arrival rate of each tier(micro-service) is predicted by time series analysis methods and Jackson network. Then the minimum6 number of containers required by each tier is obtained by queuing

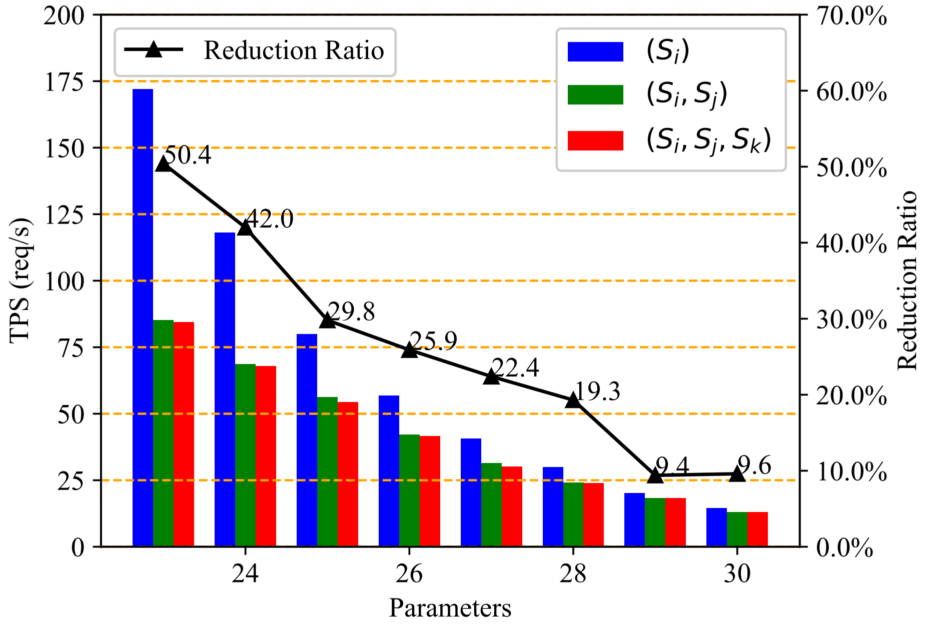

Fig. 4 The Pod TPS in parameter 23-30 under three types of business and the Pod TPS reduction ratio from business type (S_i) to business type (S_i, S_j)

models. In the reactive part, the impact of bottleneck tiers on other tiers is considered to eliminate bottlenecks quickly. In JPRM (Lei et al. 2020), queuing models with fixed processing rates are used. However, synchronous calls degrade the processing ability of containers. Therefore, in this article, an adaptive processing rate based queuing model is applied which adjust the processing rate of containers based on the ratio of synchronous calls dynamically. When a micro-service is a bottleneck, the resource is not sufficient, the number of requests passing through the tier per second is limited by the total processing rate (smaller than the arrival rate) and many requests are blocked in the queue. When the bottleneck is eliminated by providing more resources, it is assumed that the increased speed of calling other tiers is equal to the arrival rate minus the old processing rate. However, JPRM ignored that there are many tasks in queues of bottleneck tiers which generate calls to other tiers too. Therefore, in this article, the impact of queuing tasks is considered. When the increased speed of generating calls to other tiers are obtained, Jackson network is applied to obtain the request arrival rate of each tier.

5.1 Adaptive processing rate based Queuing Model

Queuing model has been used to model the relation among the processing rate $\boldsymbol{\mu}$, container number $\mathbf{N}$, request arrival rate λ and mean response time of one micro-service (Jiang et al. 2013; Lei et al. 2020). As mentioned above, the request processing rate of one container is affected by the ratio of synchronous calls. Using a fixed processing rate can not estimate the required number of Pods accurately.

In this article, the processing rate of each service S_i's container is calculated dynamically based on the ratio of requests from different business types. Let μ_i be the processing rate of S_i's container when there is no synchronous calls and ρ_i is the degradation ratio of processing ability when there are synchronous calls. The processing rate of b_l's requests is $\mu_{i,l} = \mu_i \times \alpha_l, b_l \in B_i$ where B_i is the set of business types containing S_i. $\alpha_l = \rho_i$ when b_l generates synchronous calls from S_i, otherwise, $\alpha_l = 1$. Therefore, the average processing time (not including waiting time) of b_l's requests is $\frac{1}{\mu_{i,l}}$. Let $\lambda_{i,l}$ be the arrival rate of b_l's requests to S_i and λ_i be the total arrival rate of all requests, the expected processing time of all requests is

$$\Gamma_i = \sum \frac{\lambda_{i,l}}{\lambda_i} \times \frac{1}{\mu_{i,l}} (b_l \in B_i) \tag{1}$$

Then the combined average processing rate $\hat{\mu}_i$ of S_i for all business types is

$$\hat{\mu}_i = \frac{1}{\Gamma_i} \tag{2}$$

Based on M/M/N queuing model (Cai and Buyya 2021; Lei et al. 2020; Jiang et al. 2013), the probability of no requests in S_i is

$$\mathbb{P}_0 = \left[\sum_{z=0}^{N_i-1} \frac{1}{(z)!} (\frac{\lambda}{\hat{\mu}_i})^z + \frac{\lambda_i^N}{N_i! \left(i - \frac{\lambda}{N_i \times \hat{\mu}_i} \right) \hat{\mu}_i^{N_i}} \right]^{-1} \tag{3}$$

The expected of response time of requests is

$$W_{S_i}(N_i, \lambda_i, \hat{\mu}_i) = \mathbb{P}_0 \frac{(\lambda_i/\hat{\mu}_i)^{N_i}}{N_i!(N_i\hat{\mu}_i)\left(1 - \left(\frac{\lambda_i}{N_i\hat{\mu}_i}\right)^2\right)} + \frac{1}{\hat{\mu}_i} \tag{4}$$

The minimum number of containers required by S_i to make the mean response time smaller than a given upper limit $\hat{W}_{S_i}$ is

$$N_i = \min_{N_i \in Z^+} \{N_i | W_{S_i}(N_i, \lambda_i, \hat{\mu}_i) \le \hat{W}_{S_i}\} \tag{5}$$

5.2 Prediction based proactive method

Algorithm 1 Proactive Algorithms

```
Input: Original processing rate $\mu_i$, processing rate degradation ratio $\rho_i$, reference response time $W_{S_i}, i \in \{1, 2, ..., n\}$
1:  for each micro-service $S_i$ do
2:      Obtain arrival rate $\lambda_i$, $\lambda_{i,l}$ from Logs Manager;
3:      Predicted arrival rate $\lambda_i^p$ of $S_i$;
4:      if $\lambda_i > \lambda_i^p$ then
5:          $\lambda_i^p \leftarrow \lambda_i$;
6:      end if
7:      Calculate real-time processing rate $\hat{\mu}_i$ by equation(2);
8:      Get $N_i$ based on Equation (5);
9:      Allocate $N_i$ containers to micro-service $S_i$;
10: end for
```

In order to deal with workload changes in advance, predicting workloads is very important. Multiplicative Holt-Winter's model (Lei et al. 2020; Balaji et al. 2014), which can predict quickly and precisely for time series data, has been applied. In case of the blockage of system , the prediction is mainly used to forecast workload increase.

The formal description of the proactive method is shown in Algorithm 1. As described in Sect. 4, For each micro-service S_i, the total arrival rate λ_i and the arrival rate $\lambda_{i,l}$ of each business type are first collected from Logs Manager. Then, the arrival rate λ_i^p of next time interval is predicted by the Multiplicative Holt-Winter, and the real-time processing rate $\hat{\mu}_i$ of S_i's container based on the current combination of different business types is calculated by Equation (2). Next, the required number of containers N_i for S_i is obtained by Equation (5) based on the predicted arrival rate λ_i^p and $\hat{\mu}_i$. However, if the predicted arrival rate is smaller than the current arrival rate, the current arrival rate is used to calculate the required number of containers. Finally, the number of containers allocated to S_i is adjusted to be N_i by allocating or releasing containers.

5.3 Queuing-length-aware reactive method

Algorithm 2 Reactive Algorithm

```
Input: $\hat{\mu}_i$, $W_{S_i}$, MRT $MRT_i$, current existing containers: $N_i^c$, $i \in \{1, 2, ..., n\}$, $\Psi$
1:  for micro-service $S_i$ do
2:      Obtain $\lambda_i$, $\lambda_{i,l}$, $Q$ from Logs Manager;
3:      Set $RUTL_i$ based on solving (7);
4:  end for
5:  for each micro-service $S_i$ do
6:      if $MRT_i > RTUL_i$ then
7:          Label $S_i$ as a bottlerneck tier;
8:          Calculate $\Delta\lambda_i$ based on solving(8);
9:      end if
10: end for
11: for micro-service $S_i$ do
12:     Calculate $\lambda_i'$ based on solving(10);
13:     Compute $N_i$ based on (5);
14:     if $N_i < N_i^c$ then
15:         $N_i \leftarrow N_i^c$ ;
16:     end if
17:     Allocate $N_i$ containers for micro-services $S_i$;
18: end for
```

Since workload changes dynamically, there might be deviations between the real-time request arrival rate and the predicted arrival rate during the proactive scheduling interval. Therefore, a reactive method is called more frequently than the proactive method to cope with the changes of workload. When the resource is not sufficient for some micro-services, these micro-services are called bottlenecks, and bottlenecks should be eliminated by considering the impact to other micro-services. JQN has been widely used to model the interactions among networked queuing systems which can be used to estimate the arrival rate of each micro-service after eliminating bottlenecks considering increased speeds of generating calls from bottleneck services. However, the queuing lengths of bottleneck services are not considered and the criteria of judging bottlenecks did not considered the impact of business types in existing method.

In this article, a JQN based queuing-length-aware reactive method has been proposed. Before JQN can be used to estimate the final arrival rate of each micro-service, increased speed of generating calls to other services (called additional passing rate) from bottleneck services should be calculated. The basic of obtain additional passing rates is to determine which micro-services are bottlenecks. The intuitive way of finding bottlenecks is to compare the average response time with an RTUL, and a fixed RTUL of each micro-service is usually set based on the complexity of itself without considering impact of synchronous calls. However, synchronous calls of different business types have different waiting times making the total processing time diverse. A larger response time does not mean the service is a bottleneck because it may be the result of synchronous calls to other services. The proportion of different types of business changes dynamically leading to different response time as well.

In this article, the RTUL of each micro-service is adjusted dynamically based on the proportion of different business types. Let $p_{l,i}$ be the partial accessing path starting from S_i to the last service of the business type $b_l \in B_i$. When there is no synchronous calls, t_i denotes the basic processing time of S_i. Then the total processing time of the partial path $p_{l,i}$ is

$$T_{l,i} = \sum_{S_k \in p_{l,i}} t_k \tag{6}$$

Different partial paths have different processing time, and the expected processing time of all partial paths starting from S_i is adopted as the RTUL (called RTUL-b) of S_i as follows

$$RTUL_i = \sum_{b_l \in B_i} \frac{\lambda_{i,l}}{\lambda_i} \times T_{l,i} \tag{7}$$

RTUL-b is the theoretical response time of S_i under the current workload type with sufficient resource. Therefore, comparing the average response time with RTUL-b is able to identifying bottlenecks more accurately.

Queuing-length of each bottleneck tier also has a great impact on other tiers. When the average response time of S_i is larger than $RTUL_i$, S_i is a bottleneck. $\hat{\mu}_i \times N_i$ is the total processing ability of all S_i's containers. If $\lambda_i > \hat{\mu}_i \times N_i$, S_i is unstable and the queuing length will increase gradually until the timeout or maximum number of connection limit is reached. Otherwise, S_i is stable, but the number of containers is not sufficient and the queuing length q_i increases. When the bottleneck tiers are allocated more resources, the increased speed of generating calls to other services is affected not only by $\lambda_i - \hat{\mu}_i \times N_i$ (current speed), but by q_i too (accumulated requests). Therefore, the additional speed of generating calls to others from S_i (increased passing rate) is

$$\Delta\lambda_i = \begin{cases} \lambda_i - \hat{\mu}_i \times N_i + q_i & \hat{\mu}_i \times N_i < \lambda_i \\ q_i & otherwise \end{cases} \tag{8}$$

which is determined based on the stability of S_i separately.

After the increased passing rate is calculated for all bottleneck tiers, JQN is applied to calculate the final arrival rate of each tier. In JQN, the invoking probability among different micro-services is described a matrix Ψ

$$\Psi = \begin{pmatrix} \psi_{11} & \cdots & \psi_{1n} \\ \vdots & \psi_{ij} & \vdots \\ \psi_{n1} & \cdots & \psi_{nn} \end{pmatrix} \tag{9}$$

which can be obtained from logs. $\psi_{ij}(i,j \neq 0)$ is the ratio of S_i calling S_j. Ψ changes when the proportion of different business types changes. Therefore, Ψ is updated every proactive interval. Based on Ψ and Δ_i, the final arrival rate λ_i of S_i can be obtained by solving the following system of linear equations

$$\lambda'_i = \lambda_i + q_i + \sum_{j=1}^{n} \Delta\lambda_j \times \psi_{ji}, i \in \{1, 2, \dots, n\} \tag{10}$$

Finally, container numbers N_i for each service S_i can be obtained by Equation (5). Only when N_i is larger than the current number of containers N_i^c allocated S_i, additional containers are allocated to S_i, i.e., reactive method does not release resources because these resources allocated by the proactive method might be used in the left time of the interval.

5.4 Integer linear programming based VM provisioning

Previous methods are all about the allocating and releasing of containers. However, these containers are deployed in VMs which are rented from public Clouds. It is assumed that configurations of different micro-service's Pods are the same, all types of VMs can hold integer number of Pods. Let $H_k, k \in \{1, 2, \dots, M\}$ be the number of Pods (containers) can be hold by the $k-th$ type of VMs, $C^{pod} = \sum_{i \in \{1,2,\dots,n\}} N_i$ be the total number of Pods required by all services, and C_p^{pod} be the current number of Pods of all services. When $C^{pod} > C_p^{pod}$, $gap = C^{pod} - C_p^{pod}$ number of Pods should be rented. The following integer linear programming (ILP) model is build and solved by or-tools to get the optimal number of VMs.

$$\begin{aligned} \min \quad & \Delta C^{pod} \\ \text{s.t.} \quad & \Delta C^{pod} > gap \\ & \Delta C^{pod} = \sum H_k \times C_k^{vm} \quad k \in \{1, 2, \dots, M\} \end{aligned} \tag{11}$$

where C_k^{vm} is the number of newly rented VMs of type k.

When $C^{pod} < C_p^{pod}$, $gap = C_p^{pod} - C^{pod}$ number of Pods should be released. Another ILP model is used as follows

$$\begin{aligned} \max \quad & \Delta C^{pod} \\ \text{s.t.} \quad & \Delta C^{pod} < gap \\ & \Delta C^{pod} = \sum H_k \times C_k^{vm} \quad k \in \{1, 2, \dots, M\} \end{aligned} \tag{12}$$

where C_k^{vm} is the number of released VMs of type k. Whenever a VM is determined to be released, the action takes effect only at the next pricing interval of it.

6 Performance evaluation

The proposed method has been compared with existing algorithms on a Kubernetes cluster using Wikipedia traces (Urdaneta et al. 2009) and NASA-HTTP traces (Arlitt and Williamson 1996). Each worker node deployed on VMs has 8 virtual CPU cores and 4 GB Memory. One Pod is allocated 1 CPU core and 500Mi Memory. The meshed Web system consisting of multiple micro-services that all calculate Fibonacci numbers with $n = 28$ is applied. The Weighted Round Robin (WRR) is used as the workload balancing algorithm of TraefikLab (2021). Connection time-out of Traefik is 6s and request rate limit for each micro-service is 1000/s. User access traces of Wikipedia and NASA-HTTP shown in Fig. 5 are used to generate requests through Jmeter. The arrival rate of each business is proportional to the real load data.

Our approach has been compared with JPRM, JPRM with adaptive processing rate based queuing models (AQ_JPRM) and queue-length-aware JPRM (Q_JPRM) to verify the effectiveness of adaptive processing rate based queuing model and queue-length-aware reactive method using workload (called Workload1 and Workload2) based on Wikipedia traces. Because our approach QAQ-JPRM and AQ-JPRM have the capacity of calculating processing rate based on the ratio of synchronous calls. The initial processing rate of them is set to be the original processing rate without synchronous calls. On the contrary, the processing rates of micro-services under other algorithms without adaptive calculating ability need to be measured using Jmeter in advance like Fig. 4. And then QAQ_JPRM has been compared with JPRM under workloads (called Workload3 and Workload4) with more micro-services and longer accessing paths generated based on Wikipedia traces. For showing the generalization ability, QAQ_JPRM is also compared with JPRM under workloads (called Workload5 and Workload6) based on NASA-HTTP traces. The length of proactive scheduling intervals is 720 seconds and the length of reactive scheduling intervals is 180 seconds. The price per interval of VMs is 1. Our approach is also compared with the embedded algorithm HPA of Kubernetes. HPA-CPU is the HPA which uses the average CPU utilization of each Pod as the threshold. HPA-Request is another HPA algorithm which uses the request rate of each Pod as threshold.

6.1 Comparison between JPRM, AQ_JPRM, Q_JPRM and QAQ_JPRM

The proportions of different types of business of Workload1 and Workload2 are shown in Table 2. The average processing time of each micro-service without synchronous calls is about 30ms, and the reference response time W_s of micro-services used by the queuing model is set to be 50ms. Because the maximum length of accessing paths is 3, the upper limit of mean response time of each micro-service is defined to be 90ms in SLA. Using adaptive queuing model, AQ_JPRM and QAQ_JPRM only need the basic pod processing rate that there is no synchronous calls in each pod.

Table 2 Characteristic of Workload1 and Workload2 based on Wikipedia traces

Workload1		Workload2	
Business type	Proportion	Business type	Proportion
(S_1)	0.44	(S_3)	0.44
(S_1, S_3)	0.22	(S_3, S_1)	0.22
(S_2)	0.20	(S_2)	0.20
(S_2, S_4, S_5)	0.13	(S_2, S_4, S_5)	0.13
others	0.01	*others*	0.01

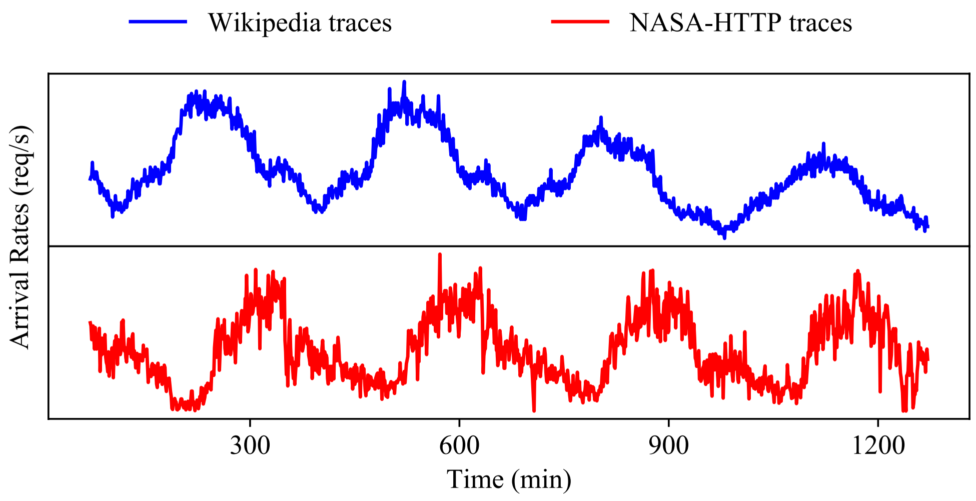

Fig. 5 The Arrival Rate of Wikipedia traces and NASA-HTTP traces

But for JPRM and Q_JPRM, it is crucial to get the pod processing rate before tests. The initial processing rate of each service is shown in Table 3.

Table 4 shows percentages of SLA-violations under Workload1 and Workload2 and total VM rental costs of all JPRM based algorithms. As a whole, our proposed approach (QAQ_JPRM) gets the lowest percentages of SLA-Violations with only 0.9% higher VM rental costs than existing JPRM.

Figures 6 and 7 show Pod numbers, response times and VM numbers of JPRM and AQ_JPRM. Because of space limitation, only results of the most representative micro-services are displayed. When the workload changes from Workload1 to Workload2, the proportion of business types changes as Table 2 shows, the processing rate of Pods in S_3 has decreased but S_1 has increased. Because the initial processing rates of JPRM is tested by JMeter based on Workload1, JPRM can not estimate the number of required containers accurately when the workload changes from Workload1 to Workload2. For example, the resource of S_3 is not sufficient while S_1 has excessive number of Pods. The degree of SLA-violations in Workload2 is serious than that in Workload1. On the contrary, AQ_JPRM is able to adjust the processing rate of each service when the workload changes. The Pod number of S_3 is increased while the Pod number of S_1 is decrease by AQ_JPRM. The degrees of SLA-Violations in Workload1 and Workload2 are nearly the same and smaller than those of JPRM which are not affected by the changes of workload types. However, the results also show that AQ_JPRM reactive method is not able to restore

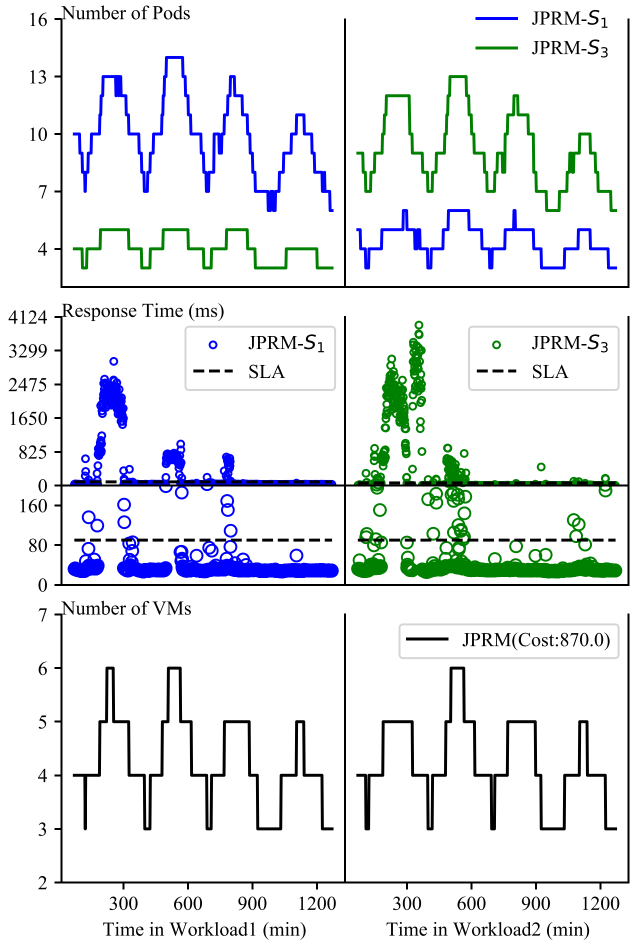

Fig. 6 Pod numbers, VM numbers and response time of JPRM

the system from congestion to normal whenever there is a surge in workload.

Figure 8 shows Pod numbers, response times and VM numbers of Q_JPRM. When there is a surge in workload, many requests might be blocked in waiting queues. Providing resources only based on the current arrival rate (in AQ_JPRM) is not sufficient to restore the system to normal state because it does not consider blocked requests. On the

Table 3 The initial processing rate of each micro-service

Methods	Micro-services				
	S_1	S_2	S_3	S_4	S_5
QAQ-JPRM	30.0	30.0	30.0	30.0	30.0
AQ-JPRM	30.0	30.0	30.0	30.0	30.0
Q-JPRM	27.7	26.6	30.0	25.0	30.0
JPRM	27.7	26.6	30.0	25.0	30.0

Table 4 Percentages of SLA-violations and VM rental costs in Workload1 and Workload2

Micro-services	JPRM		AQ_JPRM		Q_JPRM		QAQ_JPRM	
	Workload1	Workload2	Workload1	Workload2	Workload1	Workload2	Workload1	Workload2
S_1	18.4%	1.1%	13.1%	1.1%	6.2%	**0.0%**	**3.4%**	0.9%
S_2	11.1%	7.2%	4.2%	5.2%	3.5%	**2.8%**	**1.4%**	3.2%
S_3	6.4%	20.1%	1.5%	14.6%	0.6%	14.4%	**0.3%**	**6.7%**
S_4	8.1%	6.1%	1.5%	2.4%	1.6%	1.2%	**0.6%**	**0.6%**
S_5	1.0%	2.0%	0.5%	1.2%	0.7%	**0.3%**	**0.2%**	**0.3%**
VM costs	870		**860**		901		878	

Bold values indicate better results

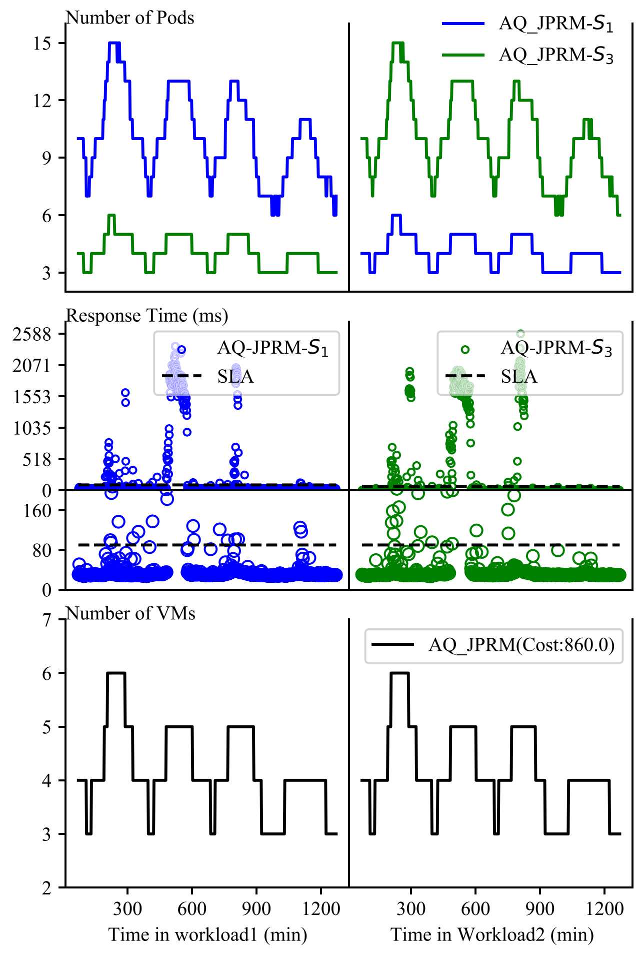

Fig. 7 Pod numbers, VM numbers and response time of AQ_JPRM

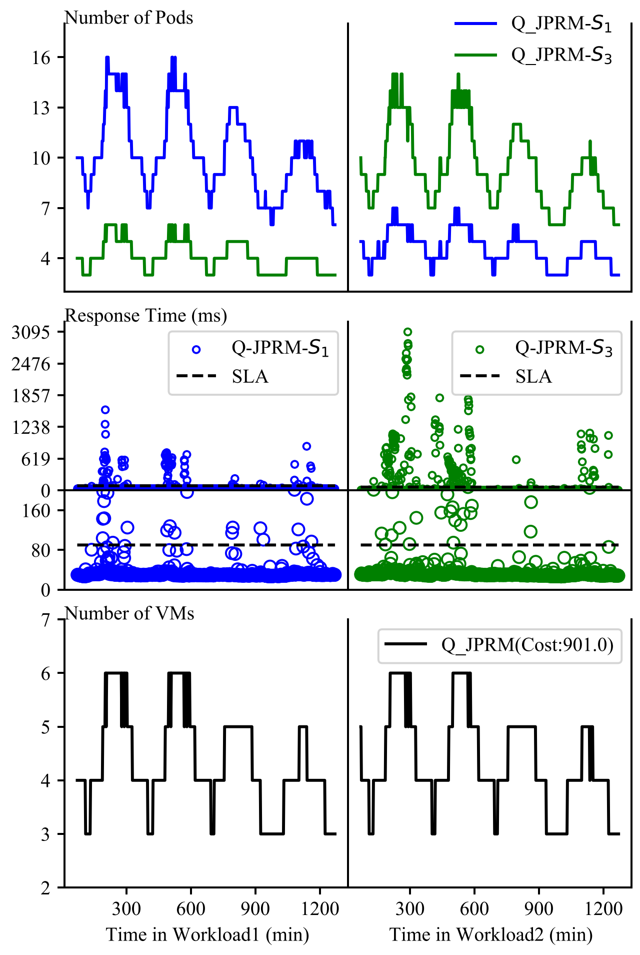

Fig. 8 Pod numbers, VM numbers and response time of Q_JPRM

contrary, Q_JPRM's reactive method is able to allocate necessary containers to all related services based on queuing lengths. As shown in Figs. 6, 7 and 8 within 1200 min in Workload1, the Q_JPRM provides more Pods in the intervals when request arrival rates surge. And the more stable system in Q_JPRM provides enough Pods to handle users' requests. As for JPRM and AQ_JPRM, without considering the queuing requests, systems are easy to break down when facing workload surge and VMs scaling. For the first two peaks in Workload1, Q_JPRM can prevent the system from all congestions but AQ_JPRM and JPRM cannot. Therefore, Q_JPRM gets lower degree of SLA-violations than JPRM and AQ_JPRM. However, when the workload changes from Workload1 to Workload2, Q_JPRM's performance in Workload2 becomes worse than that in Workload1. The reason is that Q_JPRM can not adjust processing rate when the ratio of synchronous calls changes which is similar with JPRM.

Figure 9 shows Pod numbers, response times and VM numbers of QAQ_JPRM which illustrates that most of response time in QAQ_JPRM are below SLA, especially after the prediction method collected sufficient data to forecast the arrival rate accurately. The reason is that collaboration of processing rate adjusting method and queuing-length-aware JQN is able to increase the accuracy of queuing models and restore from serious congestion simultaneously. QAQ_JPRM performs well no matter how the proportion of different business types changes. QAQ_JPRM has the similar performance with Q_JPRM in Workload1 and the best performance in Workload2. There are still SLA-violations in some periods inevitably, because the migration of Pods from a released VM to another VM takes about 20s. In total, the VM rental costs of all JPRM based methods are similar which means that QAQ_JPRM obtains much lower degrees of SLA-violation with a little bit additional cost.

6.2 Effectiveness under more micro-services and longer accessing paths

The characteristics of Workload3 and Workload4 are shown in Table 5 which is based on Wikipedia traces. The initial processing rate of each micro-service is shown in Table 6. Because QAQ_JPRM is able to calculate adaptive

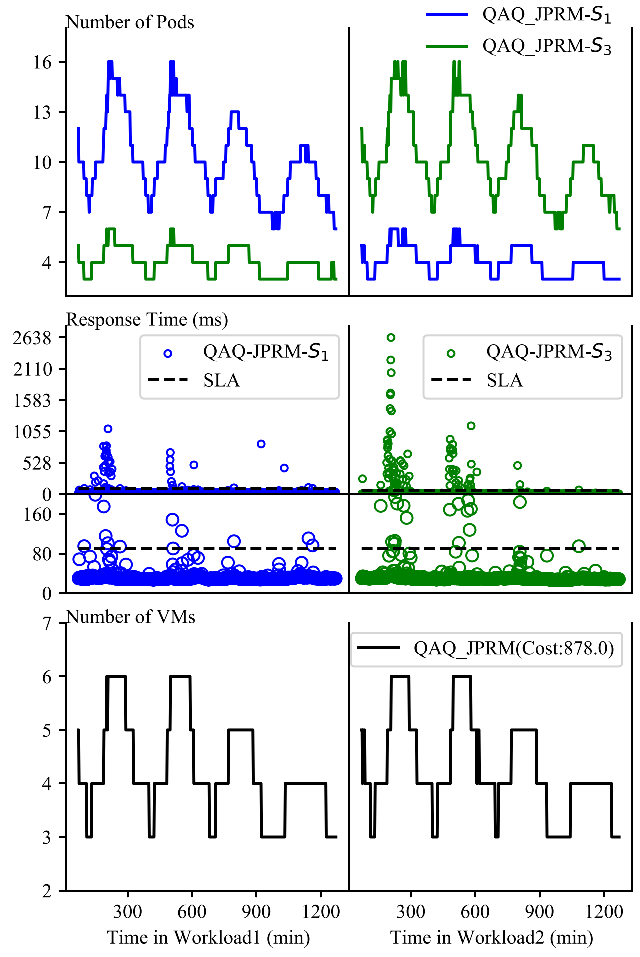

Fig. 9 Pod numbers, VM numbers and response time of QAQ_JPRM

Table 5 Characteristic of Workload3 and Workload4

Workload3		Workload4	
Business type	Proportion	Business type	Proportion
(S_3)	0.10	(S_3)	0.10
(S_2, S_5, S_7)	0.55	(S_2, S_8, S_4)	0.55
$(S_1, S_4, S_6, S_7, S_8)$	0.35	$(S_1, S_5, S_6, S_7, S_8)$	0.35

processing rate based on the proportion of different business types, the processing rate in QAQ_JPRM is the same. As the maximum length of accessing paths is 5 in Workload3 and Workload4, the upper limit of mean response time of each micro-service is changed to 150 ms. The percentages of SLA-Violations of each micro-services on Workload3 and Workload4 have been shown in Table 7. Because of longer access paths, one micro-service blockage could influence the performance of more other related micro-services. Although both JPRM and QAQ_JPRM obtained worse performance on Workload 3 and Workload4 than on Workload1 and Workload2, QAQ_JPRM is still better than JPRM. JPRM's

Table 6 The initial processing rate of each micro-service in Workload3 and Workload4

Micro-services	Methods	
	QAQ-JPRM	JPRM
S_1	30.0	24.0
S_2	30.0	24.0
S_3	30.0	30.0
S_4	30.0	24.0
S_5	30.0	24.0
S_6	30.0	24.0
S_7	30.0	27.3
S_8	30.0	30.0

Table 7 Percentages of SLA-Violations in Workload3 and Workload4

Micro-services	JPRM		QAQ_JPRM	
	Workload3	Workload4	Workload3	Workload4
S_1	11.6%	16.5%	**10.8%**	**11.4%**
S_2	8.7%	14.1%	**7.0%**	**7.2%**
S_3	0.0%	0.0%	0.0%	0.0%
S_4	10.1%	3.8%	**8.8%**	**1.4%**
S_5	7.7%	14.4%	**5.6%**	**7.5%**
S_6	9.3%	13.3%	**6.8%**	**5.6%**
S_7	8.1%	12.6%	**4.2%**	**2.6%**
S_8	7.8%	11.8%	**1.7%**	**2.0%**
VM costs	**1012**		1022	

Bold values indicate better results

degrees of SLA-Violations increase greatly when workloads changed from Workload3 to Workload4. On the contrary, with the aid of the adaptive processing rate and the JQN based queuing-length-aware reactive method, QAQ_JPRM's performance nearly kept unchanged. And during 200 pricing intervals (2400 minutes), QAQ_JPRM only consumes 10 additional VM pricing intervals.

6.3 Comparison with JPRM using NASA-HTTP traces

More experiments under Workload5 and Workload6 with the same business characteristics of Table 2 based on NASA-HTTP traces have been used to prove the effectiveness of QAQ_JPRM further. Compared with Wikipedia traces, the workload based on NASA-HTTP changes more frequently as shown in Fig. 5.

Table 8 shows the SLA-violations on Workload5 and Workload6 which illustrates that QAQ_JPRM is better than JPRM with lower degrees of SLA-violations and consuming only 5 additional VM pricing intervals. After the proportion of business types changes from Workload5 to Workload6, the percentage of SLA-Violations in S_3 for JPRM is up to

82.0% which means that the system has broken down. Figure 10 shows a comparison about Pod numbers, response time and VM numbers between JPRM and QAQ_JPRM. In Fig. 11a, JPRM's response times of S_3 become higher than 5 seconds after the workload changes from Workload5 to Workload6. That is because the queuing-length continues to worse the blockage of system. Using fixed Pod processing capacity in JPRM, the Auto-scaler has supplied insufficient Pods for S_3 in Workload6. And without considering the queuing length, that smaller arrival rates have been measured by JPRM when there are serious blockages makes the condition worse. As for QAQ_JPRM, the system remains to work well and enough Pods have been scheduled to cope with varying request arrival rates.

6.4 Comparison with embedded auto-scalers of K8s

In this experiment, QAQ_JPRM, HPA-CPU and HPA-Request are compared using Workload1 and Workload2.

Table 8 Percentages of SLA-Violations in Workload5 and Workload6 based on NASA-HTTP traces

Micro-services	JPRM		QAQ_JPRM	
	Workload5	Workload6	Workload5	Workload6
S_1	16.0%	**0.2%**	**5.4%**	0.6%
S_2	8.9%	**0.2%**	**1.4%**	4.0%
S_3	6.1%	82.0%	**1.6%**	**5.1%**
S_4	3.8%	**0.1%**	**0.7%**	2.3%
S_5	0.1%	**0.0%**	**0.0%**	0.1%
VM Costs	**895**		900	

Bold values indicate better results

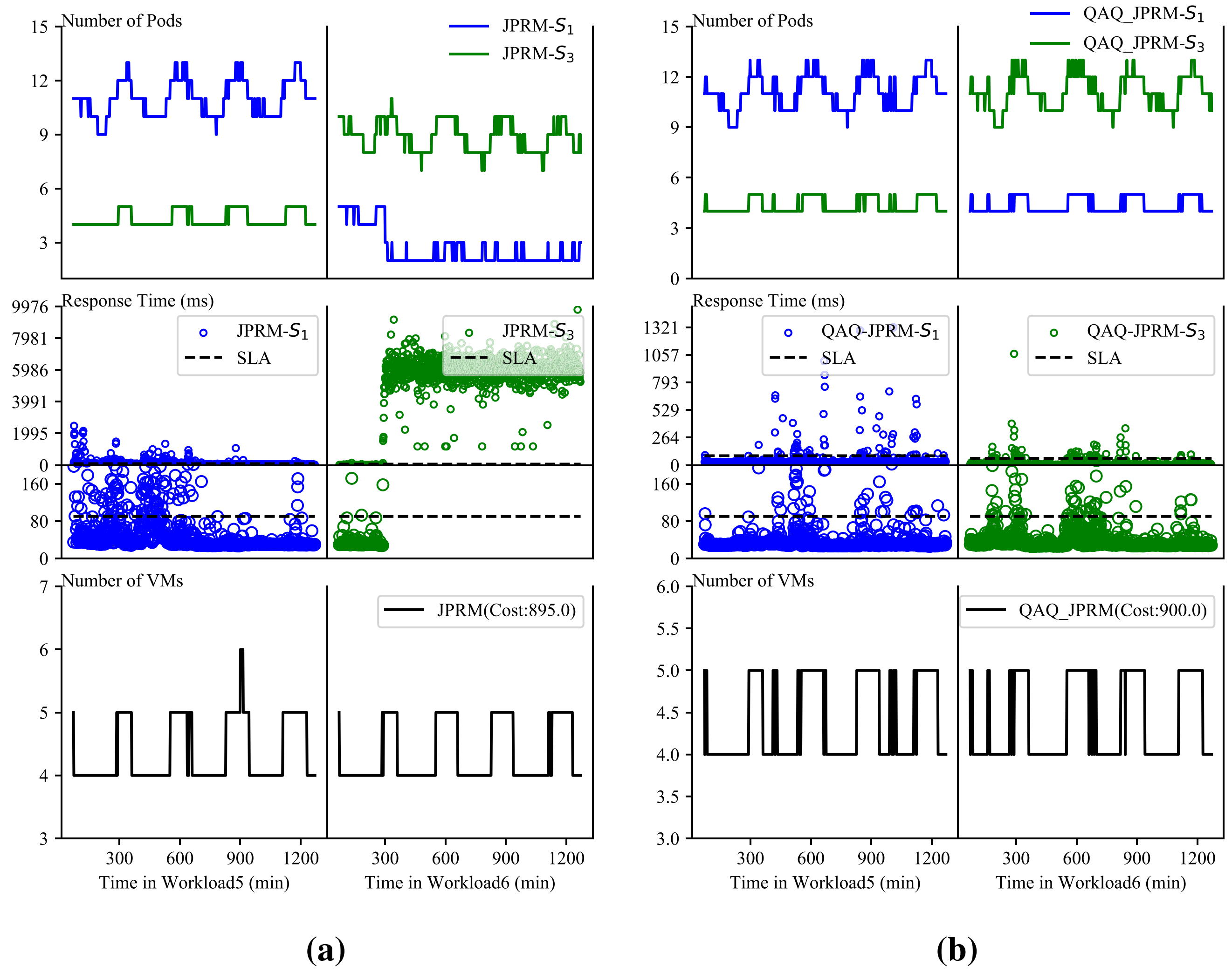

Fig. 10 Pod numbers, response time and VM numbers of JPRM and QAQ_JPRM in Workload5 and Workload6

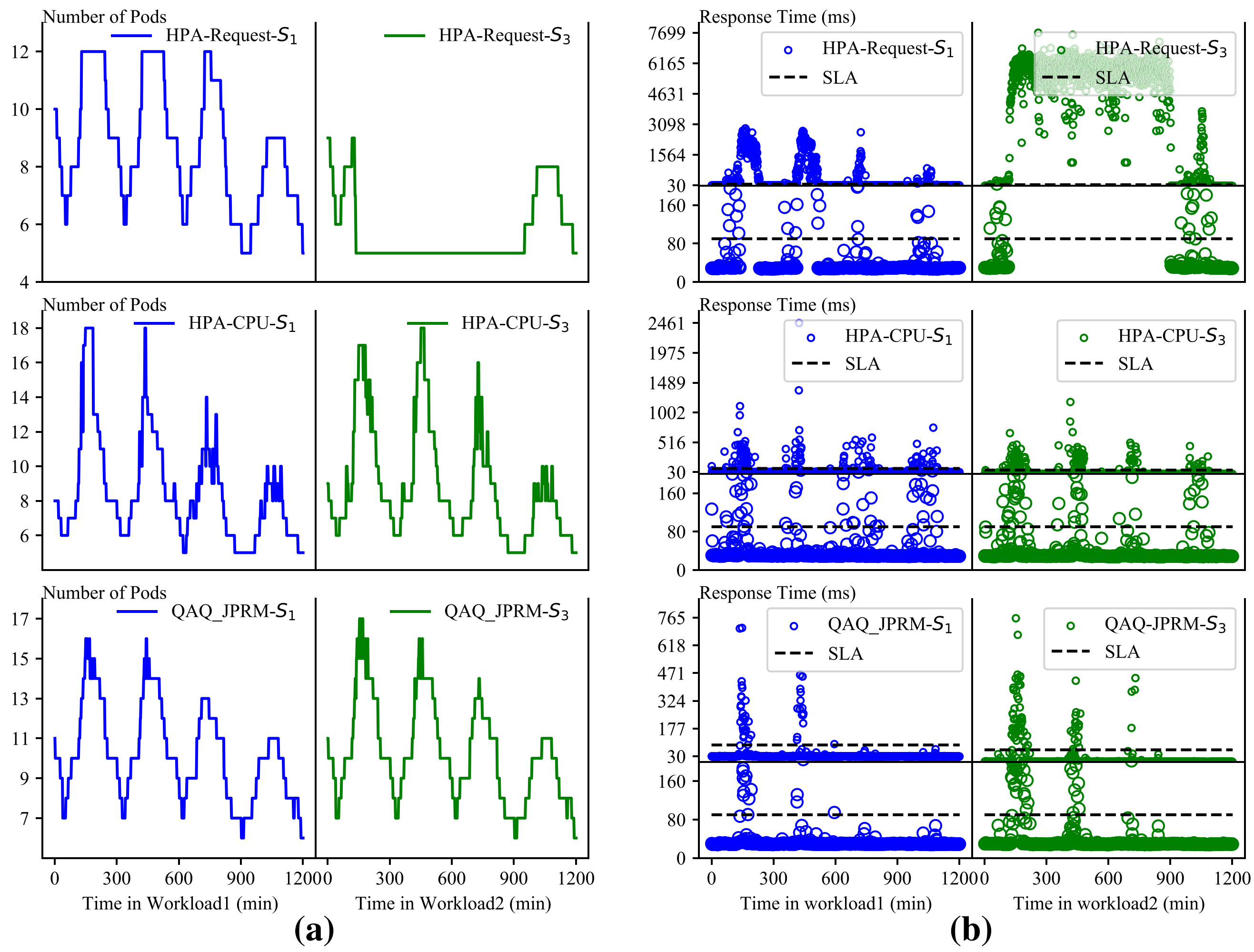

Fig. 11 Pod numbers and response time of QAQ_JPRM , HPA-CPU and HPA-Request

Table 9 The initial processing rate of each micro-service

Methods	Micro-services				
	S_1	S_2	S_3	S_4	S_5
QAQ-JPRM	30.0	30.0	30.0	30.0	30.0
HPA-Request	25.8	25.6	28.0	24.0	28.0
HPA-CPU	CPU utilization threshold : 60%				

Based on the test in Sect. 4, CPU utilization threshold is set to be 60%. For HPA-Request, it is necessary to get the pod processing rate before tests, and the initial processing rate of each micro-service is shown in Table 9. For HPA-CPU and HPA-Request, the minimal number of Pods is set to be 2-5.

Table 10 shows each micro-service's SLA-Violations of QAQ_JPRM, HPA-CPU and HPA-Request. It is because HPA-CPU and HPA-Request are only designed for Pod

Table 10 Percentages of QAQ_JPRM SLA-Violations compared with HPA

Micro-services	QAQ_JPRM		HPA-CPU		HPA-Request	
	Workload1 (%)	Workload2 (%)	Workload1 (%)	Workload2 (%)	Workload1 (%)	Workload2 (%)
S_1	**3.7**	**0.6**	10.1.	1.2	22.6	0.8
S_2	**2.7**	6.3	4.5	6.7	16.8	**3.2**
S_3	**0.0**	**6.1**	0.9	10.1	12.7	71.4
S_4	**0.2**	**0.0**	2.0	4.0	12.2	3.0
S_5	0.2	**0.0**	0.7	1.2	**0.0**	**0.0**

Bold values indicate better results

auto-scaling without considering the renting of VMs. For fair comparison, our approach QAQ_JPRM is given a fixed number of VMs similar with HPAs. Only Pods are auto-scaled while VMs can not be released and newly rented. In total, QAQ_JPRM gets the lowest degrees of SLA-violation compared with HPA algorithms.

Figure 11a and b show the number of Pods and response times of QAQ_JPRM, HPA-CPU and HPA-Request. For HPA-Request, the request arrival rate is lower than the real value when there is congestion. The reason lies at that some requests can not be received by the Web container when the port is blocked, and the arrival rate is underestimated. Meanwhile, insufficient resource will make the congestion serious which misleads the arrival rate in turn, and the system is blocked completely as shown in right part of Fig. 11a. HPA-CPU uses a linear model based on the CPU utilization to determine the required number of Pods which can not estimate the actual requirement accurately. And when there is congestion, the CPU utilization is always 100% which can not describe the degree of congestion and decreases the performance of HPA-CPU's linear model greatly. Therefore, the scale of adjusting is not rational (too large or small) leading to serious SLA-violations. The response times of QAQ_JPRM are almost all below the SLA after the prediction takes effect. QAQ_JPRM has allocated suitable number of Pods for each micro-service by the aid of the adaptive queuing model and queuing-length-aware JQN.

7 Conclusion

In this paper, a container auto-scaling algorithm is proposed for meshed micro-services in Kubernetes which takes the advantage of adaptive-processing-rate based queuing model and queuing-length-aware Jackson queuing network. Algorithms are compared on a real Kubernetes cluster using the workload with changing proportions of business types. Experimental results illustrate that adaptive processing rate calculating method considering the changes of business type's proportion is helpful to increasing the accuracy of queuing models. Meanwhile, queuing-length-aware JQN is able to restore from serious congestion by considering the impact of newly arrived and accumulated requests together. Our approach obtains the lowest percentage (decreasing about 6.33% - 12.29%) of SLA-violations compared with both of existing JPRM and embedded HPA algorithms in Kubernetes, and the VM rental cost of our approach is only about 0.9% higher than that of JPRM. Designing resource auto-scaling algorithms for meshed micro-services in Edge Computing considering the collaboration of multiple edge nodes is promising future work.

Acknowledgements This work is supported by the National Natural Science Foundation of China (Grant No. 61972202, 61872186, 61973161, 61991404), the Fundamental Research Funds for the Central Universities (No. 30919011235).

References

Abdullah, M., Iqbal, W., Bukhari, F., Erradi, A.: Diminishing returns and deep learning for adaptive CPU resource allocation of containers. IEEE Trans. Netw. Serv. Manag. **17**(4), 2052–2063 (2020). https://doi.org/10.1109/TNSM.2020.3033025

Adam, O., Lee, Y.C., Zomaya, A.Y.: Stochastic resource provisioning for containerized multi-tier web services in clouds. IEEE Trans. Parallel Distrib. Syst. **28**(7), 2060–2073 (2017). https://doi.org/10.1109/TPDS.2016.2639009

Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., Merle, P.: Elasticity in cloud computing: state of the art and research challenges. IEEE Trans. Serv. Comput. **11**(2), 430–447 (2018). https://doi.org/10.1109/TSC.2017.2711009

AlibabaCloud.: Container service for kubernetes. https://www.alibabacloud.com/product/kubernetes. Accessed 22 Oct 2021

Amazon.: Amazon elastic kubernetes service. https://aws.amazon.com/eks/. Accessed 22 Oct 2021

Arlitt, M.F., Williamson, C.L.: Web server workload characterization: the search for invariants. In: Proceedings of the 1996 ACM SIGMETRICS international conference on measurement and modeling of computer systems, association for computing machinery, New York, NY, USA, SIGMETRICS '96, pp. 126–137, (1996). https://doi.org/10.1145/233013.233034

Balaji, M., Rao, G.S.V., Kumar, C.A.: A comparitive study of predictive models for cloud infrastructure management. In: 2014 14th IEEE/ACM international symposium on cluster, cloud and grid computing, pp. 923–926, (2014). https://doi.org/10.1109/CCGrid.2014.32

Baresi, L., Guinea, S., Leva, A., Quattrocchi. G.: A discrete-time feedback controller for containerized cloud applications. In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering, association for computing machinery, New York, NY, USA, FSE 2016, pp. 217–228, (2016). https://doi.org/10.1145/2950290.2950328

Barrett, E., Howley, E., Duggan, J.: Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. Concurr. Comput. Pract. Exp. **25**(12), 1656–1674 (2013). https://doi.org/10.1002/cpe.2864

Bi, J., Yuan, H., Tie, M., Tan, W.: Sla-based optimisation of virtualised resource for multi-tier web applications in cloud data centres. Enterp. Inf. Syst. **9**(7), 743–767 (2015). https://doi.org/10.1080/17517575.2013.830342

Cai, Z., Buyya, R.: Inverse queuing model based feedback control for elastic container provisioning of web systems in Kubernetes. IEEE Trans. Comput. (2021). https://doi.org/10.1109/TC.2021.3049598

Cai, Z., Liu, D., Lu, Y., Buyya, R.: Unequal-interval based loosely coupled control method for auto-scaling heterogeneous cloud resources for web applications. Concurr. Comput. Pract. Exp. **32**(23), e5926 (2020). https://doi.org/10.1002/cpe.5926

Calheiros, R.N., Ranjan, R., Beloglazov, A., De Rose, C.A.F., Buyya, R.: Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw. Pract. Exp. **41**(1), 23–50 (2011). https://doi.org/10.1002/spe.995

Chen, H., Wang, Q., Palanisamy, B., Xiong, P.: DCM: dynamic concurrency management for scaling n-tier applications in cloud. In: Lee, K., Liu, L. (eds.) 2017 IEEE 37th international conference on distributed computing systems (ICDCS 2017), pp. 2097–2104, (2017). https://doi.org/10.1109/ICDCS.2017.22

Delnat. W., Truyen, E., Rafique, A., Van Landuyt, D., Joosen, W.: K8-scalar: a workbench to compare autoscalers for container-orchestrated database clusters. In: Proceedings of the 13th international conference on software engineering for adaptive and self-managing systems, association for computing machinery, New York, NY, USA, SEAMS '18, pp. 33–39, (2018). https://doi.org/10.1145/3194133.3194162

Huang, G., Wang, S., Zhang, M., Li, Y., Qian, Z., Chen, Y., Zhang, S.: Auto scaling virtual machines for web applications with queueing theory. In: 2016 3rd International conference on systems and informatics (ICSAI), pp. 433–438, (2016). https://doi.org/10.1109/ICSAI.2016.7810994

Jiang, J., Lu, J., Zhang, G., Long, G.: Optimal cloud resource auto-scaling for web applications. In: Proceedings of the 13th IEEE/ACM international symposium on cluster, cloud, and grid computing, IEEE Press, CCGRID '13, pp. 58–65, (2013). https://doi.org/10.1109/CCGrid.2013.73

Jmeter, A.: Apache jmeter: workload generator. https://jmeter.apache.org/. Accessed 22 Oct 2021

Kho Lin, S., Altaf, U., Jayaputera, G., Li, J., Marques, D., Meggyesy, D., Sarwar, S., Sharma, S., Voorsluys, W., Sinnott, R., Novak, A., Nguyen, V., Pash, K.: Auto-scaling a defence application across the cloud using docker and Kubernetes. In: 2018 IEEE/ACM international conference on utility and cloud computing companion (UCC companion), pp. 327–334, (2018). https://doi.org/10.1109/UCC-Companion.2018.00076

Kubernetes.: Horizontal pod autoscaler. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Accessed 22 Oct 2021

Lei, Y., Cai, Z., Wu, H., Buyya, R.: Cloud resource provisioning and bottleneck eliminating for meshed web systems. In: 2020 IEEE 13th international conference on cloud computing (CLOUD), pp. 512–516, (2020). https://doi.org/10.1109/CLOUD49709.2020.00076

Li, H., Venugopal, S.: Using reinforcement learning for controlling an elastic web application hosting platform. In: Proceedings of the 8th ACM international conference on autonomic computing, association for computing machinery, New York, NY, USA, ICAC '11, pp. 205–208, (2011). https://doi.org/10.1145/1998582.1998630

Lorido-Botran, T., Miguel-Alonso, J., Lozano, J.A.: A review of auto-scaling techniques for elastic applications in cloud environments. J. Grid Comput. **12**(4), 559–592 (2014). https://doi.org/10.1007/s10723-014-9314-7

Magableh, B., Almiani, M.: A self healing microservices architecture: a case study in docker swarm cluster. In: Barolli, L., Takizawa, M., Xhafa, F., Enokido, T. (eds.) Advanced Information Networking and Applications, pp. 846–858. Springer, Cham (2020)

Melikov, A.Z., Rustamov, A.M., Sztrik, J.: Queuing management with feedback in cloud computing centers with large numbers of web servers. In: Vishnevskiy, V.M., Kozyrev, D.V. (eds.) Distributed Computer and Communication Networks, pp. 106–119. Springer, Cham (2018)

Pan, W., Mu, D., Wu, H., Yao, L.: Feedback control-based qos guarantees in web application servers. In: 2008 10th IEEE international conference on high performance computing and communications, pp. 328–334, (2008). https://doi.org/10.1109/HPCC.2008.106

Piraghaj, S.F., Dastjerdi, A.V., Calheiros, R.N., Buyya, R.: Container-cloudsim: an environment for modeling and simulation of containers in cloud data centers. Softw. Pract. Exp. **47**(4), 505–521 (2017). https://doi.org/10.1002/spe.2422

Qu, C., Calheiros, R.N., Buyya, R.: Auto-scaling web applications in clouds: a taxonomy and survey. ACM Comput. Surv. **51**, 4 (2018). https://doi.org/10.1145/3148149

Stormacq, S.: Lightsail containers: an easy way to run your containers in the cloud. https://aws.amazon.com/cn/blogs/aws/lightsail-containers-an-easy-way-to-run-your-containers-in-the-cloud/. Accessed 22 Oct 2021

Toka, L., Dobreff, G., Fodor, B., Sonkoly, B.: Adaptive ai-based auto-scaling for Kubernetes. In: 2020 20th IEEE/ACM international symposium on cluster, cloud and internet computing (CCGRID), IEEE Computer Society, Los Alamitos, CA, USA, pp. 599–608, (2020). https://doi.org/10.1109/CCGrid49817.2020.00-33

TraefikLab.: Traefik : Edge router. (2021). https://doc.traefik.io/traefik/

Urdaneta, G., Pierre, G., van Steen, M.: Wikipedia workload analysis for decentralized hosting. Comput. Netw. **53**(11), 1830–1845 (2009). https://doi.org/10.1016/j.comnet.2009.02.019

Wang, Q., Chen, H., Zhang, S., Hu, L., Palanisamy, B.: Integrating concurrency control in n-tier application scaling management in the cloud. IEEE Trans. Parallel Distrib. Syst. **30**(4), 855–869 (2019). https://doi.org/10.1109/TPDS.2018.2871086

Xu, M., Buyya, R.: Brownoutcon: a software system based on brownout and containers for energy-efficient cloud computing. J. Syst. Softw. **155**, 91–103 (2019). https://doi.org/10.1016/j.jss.2019.05.031

Zhang, W., Shi, Y., Liu, L., Zhang, S., Zheng, Y., Cui, L., Yu, H.: CTP: a scheduling strategy to smooth response time fluctuations in multi-tier website system. Microprocess. Microsyst. **47**, 198–208 (2016). https://doi.org/10.1016/j.micpro.2016.05.017

Zhong, Z., Buyya, R.: A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources. ACM Trans. Internet Technol. (2020). https://doi.org/10.1145/3378447

Hang Wu is currently working toward the M.Sc. degree in School of Computer Science and Engineering from Nanjing University of Science and Technology, China. He is currently working as a cloud native development engineer. His main research interests focus on resource scheduling, Fog Computing, Cloud Computing.

Zhicheng Cai (Member, IEEE) received the Ph.D. degree in Applied Computer Science from Southeast University, Nanjing, China, in 2015. He is an associate professor with the Nanjing University of Science and Technology, China. His research interests focus on resource scheduling in Cloud, Fog and Edge computing. He is the author of more than 20 publications in journals such as IEEE Transactions on Computers, IEEE Transactions on Services Computing, IEEE Transactions on Cloud Computing, IEEE Transactions on Automation Science and Engineering and Future Generation Computer Systems and at conferences such as ICSOC, ICPADS, ISPA, ICA3PP, CLOUD, HPCC, SMC, CBD, and CASE.

Yamin Lei received the M.Sc. degree in School of Computer Science and Engineering from Nanjing University of Science and Technology, China in 2021. She is currently working toward the Ph.D. degree in Southeast University, Nanjing, China. Her main research interests focus on resource scheduling of meshed Web systems in Cloud Computing.

Jian Xu received a Ph.D. in Computer Science in 2007 from Nanjing University of Science and Technology, Nanjing, China. Now he holds the position of a professor at Nanjing University of Science and Technology. His research interests are event mining, log mining and their applications to complex system management, and he has published about 30 papers in journals and refereed conference proceedings in those areas.

Rajkumar Buyya (Fellow, IEEE) is a Redmond Barry distinguished professor and director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, University of Melbourne, Australia. He has authored more than 625 publications and seven text books. He is one of the highly cited authors in computer science and software engineering worldwide (h-index=153, g-index=324, more than 121,200 citations). Microsoft Academic Search Index ranked him as \#1 author in the world (2005-2016) for both field rating and citations evaluations in the area of distributed and parallel computing. He is recognized as a “Web of Science Highly Cited researcher” during 2016-2021 by Thomson Reuters.