Computing and Informatics, Vol. 42, 2023, 1037–1059, doi: 10.31577/cai_2023_5_1037

AUTOSCALING METHOD FOR DOCKER SWARM TOWARDS BURSTY WORKLOAD

Qichen Huang, Song Wang, Zhijun Ding*

Department of Computer Science and Technology
Tongji University
No. 4800, Caoan Highway
Shanghai, China
e-mail: zhijun_ding@outlook.com

Abstract. The autoscaling mechanism of cloud computing can automatically adjust computing resources according to user needs, improve quality of service (QoS) and avoid over-provision. However, the traditional autoscaling methods suffer from oscillation and degradation of QoS when dealing with burstiness. Therefore, the autoscaling algorithm should consider the effect of bursty workloads. In this paper, we propose a novel AmRP (an autoscaling method that combines reactive and proactive mechanisms) that uses proactive scaling to launch some containers in advance, and then the reactive module performs vertical scaling based on existing containers to increase resources rapidly. Our method also integrates burst detection to alleviate the oscillation of the scaling algorithm and improve the QoS. Finally, we evaluated our approach with state-of-the-art baseline scaling methods under different workloads in a Docker Swarm cluster. Compared with the baseline methods, the experimental results show that AmRP has fewer SLA violations when dealing with bursty workloads, and its resource cost is also lower.

Keywords: Cloud computing, autoscaling, bursty workload, container, service-level agreement

Mathematics Subject Classification 2010: 68-W27

* Corresponding author

1 INTRODUCTION

In recent years, the rapid development of cloud computing has provided basic support for Big Data [1], Internet of Things [2], Artificial Intelligence [3], and other fields. Autoscaling is one of the important characteristics of cloud computing [4, 5], which automatically adjusts computing resources based on service requirements and preset policies. Appropriate computing resources can be allocated in a peak or a trough period of workload. Therefore, autoscaling further reflects the advantages of pay-as-you-go in cloud computing. Cloud vendors such as AWS[1], Google Cloud[2] and Microsoft Azure[3] have corresponding scaling strategies. According to the scaling policy, autoscaling can be divided into horizontal and vertical scaling [6]. Horizontal scaling refers to scaling in/out, adjusting only the number of containers/VMs (Virtual Machines). Vertical scaling refers to scaling up/down, which only adjusts the resources, such as CPU, memory, and network bandwidth. In Kubernetes [7, 8, 9], HPA [10] and VPA[4] are the horizontal and vertical scalers in the cluster, respectively. In addition, autoscaling can also be classified by scaling timing [11]. Reactive autoscaling uses the current service status and workload to make scaling decision and proactive autoscaling employs the future status of the service or workload.

Autoscaling can adjust the computing resource in real time as the workload changes. However, for bursty workload, whether it is reactive autoscaling or proactive autoscaling, there will be a period of QoS degradation. The impact of burstiness on the scaling algorithm is mainly due to two points. The first point is that the bursty workload usually fluctuates wildly, which brings oscillation to the scaling algorithm. That is, the resource provided is frequently changed; Another impact is that burstiness will cause a period of service degradation. For reactive scaling, resources are already under-provision when burstiness is detected. Similarly, proactive scaling presents a similar problem, as the quality of service (QoS) inevitably degrades when dealing with the bursty workload since it is hard to predict.

Most of the existing research about autoscaling focus on the prediction and resource provision models [12, 13, 14, 15]. For a non-bursty workload, optimizing the above two models can ensure the QoS and use fewer resources. However, autoscalers require additional optimization for bursty workloads. In this paper, we propose a novel burst-aware scaling method named AmRP (an autoscaling method that combines reactive and proactive mechanisms). AmRP can be divided into two main modules: the proactive module and the reactive module. The proactive module launches a part of the containers in advance. These containers mainly serve the reactive scaling modules. When the reactive module of AmRP performs scaling, vertical scaling can be performed on existing containers, which increases resources

[1] `https://aws.amazon.com/cn/autoscaling/`

[2] `https://cloud.google.com/compute/docs/autoscaler`

[3] `https://azure.microsoft.com/en-us/features/autoscale`

[4] `https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler`

more rapidly than horizontal scaling. In addition, burst detection is added to the reactive module. If the surge in the number of requests is detected, the scaling scheme will be further adjusted to alleviate the oscillation of the scaling algorithm and better ensure the QoS. In this paper, an AmRP prototype is designed and developed on Docker Swarm Cluster. Compared with the baseline methods, resource cost and SLA violation are lower when dealing with the bursty workload. The main contributions of this paper are as follows:

1. It designs a complete scaling system, which adjusts resources according to the real time workload to meet the expected response time.
2. It proposes a novel scaling algorithm combining reactive and proactive scaling to provide resources rapidly. The algorithm also adds burst detection to alleviate the impact of bursty workload on QoS.
3. It implements an AmRP prototype on Docker Swarm and evaluates it with the baseline methods under different workload types.

The paper is organized as follows. Section 2 introduces the work related to autoscaling in recent years. Section 3 describes in detail the AmRP scaling strategy proposed in this paper. The specific experimental design, including baseline methods and benchmark application, is described in Section 4. Section 5 analyzes the experimental results. Section 6 describes the conclusion and future work.

2 RELATED WORK

Autoscaling can be divided into reactive scaling and proactive scaling according to scaling timing. [16, 17, 18] are all about reactive scaling based on rules and analytical models. Using native Kubernetes HPA requires certain experience to set reasonable scaling rules. Therefore, [16] solves this problem employing two-stage scaling. Libra autoscaler is an autoscaler proposed by the authors. Libra first uses vertical scaling to find the optimal resource allocation for pods and then enters horizontal scaling to cope with fluctuating workloads. In [17], the author proposes a dynamic multi-layer indicator scaling method, adding application-level indicators based on native Kubernetes HPA, optimizing resource usage, and further ensuring QoS. [18] proposes a combined scaling method named COPA. When making scaling decisions, vertical and horizontal scaling are combined, and the rolling update parameter in Kubernetes is taken into account. While ensuring QoS, COPA reduces overall resource costs. The above work takes the current workload and service status as input when making scaling decisions, called reactive scaling. When proactive scaling makes scaling decisions, the input is the future workload or future service status. [19] proposes an autoscaler based on machine learning by using time series forecasting and queuing theory, which can accurately predict the workload of distributed servers, estimate resources required, optimize service response time, and meet SLA. In [20], authors use LSTM as a prediction model to dynamically scale horizontally and vertically to improve the end-to-end latency of the service.

According to the scaling policy, autoscaling can also be divided into horizontal and vertical scaling. [21] improves the VPA of Kubernetes. The problem with the native VPA is that the way to adjust resources is to start new pods and terminate old ones. Therefore, the author proposes RUBAS, which solves the VPA adjustment resource restart problem and improves resource utilization through container integration migration and checkpoint technology. The scenario in [22] is that container-based IoT applications in edge computing need to dynamically adjust resources according to the amount of IoT device requests. However, the native Kubernetes HPA evenly deploys pods on each node without considering the imbalance of resource demand among nodes in the edge computing environment. Therefore, the author proposed THPA, running on Kubernetes, to achieve real time traffic awareness and autoscaling pods for IoT applications in edge computing environments.

The impact of bursty workloads is rarely considered when scaling decisions are made. [23] uses Bi-LSTM to predict the number of HTTP requests and designs a proactive autoscaling approach in Kubernetes. Simple handling of the bursty workload is added to this method. The idea is to reserve part of pods each time when performing scaling in so that the autoscaler can have a better QoS and increase resources faster dealing with burstiness. [24] adds online burst detection into proactive autoscaling, uses standard deviation and sliding window to detect burstiness, and allocates a relatively stable amount of resources after detecting burstiness for the first time, which solves the problem of scaling oscillation, and a certain extent guaranteed QoS. The method in [25] is similar to that in [24], but the difference lies in that the information entropy method is used in [25] for burst detection. The existing scaling methods are quite simple to deal with the bursty workload. In scaling decisions, more resources are allocated to cope with burstiness that may occur at any time, resulting in higher resource costs than algorithms that do not consider bursty workload. In addition, because the bursty workload is hard to predict, the resources should be increased rapidly when burstiness is detected. However, this is rarely considered in the existing scaling strategies. Therefore, we propose a novel method combining proactive and reactive scaling. Proactive scaling performs workload prediction and launches containers in advance. Reactive scaling performs online burst detection and prioritizes vertical scaling. Compared with horizontal scaling, the vertical scaling strategy can increase resources more quickly and further reduce SLA violations.

3 PROPOSED SYSTEM

This section begins with an overview of AmRP. Section 3.2 describes the reactive module, including burst detection, resource provision model, and reactive scaling algorithm. Section 3.3 introduces the proactive module, including the time series forecasting model, an estimation of the maximum requests, and the proactive scaling algorithm.

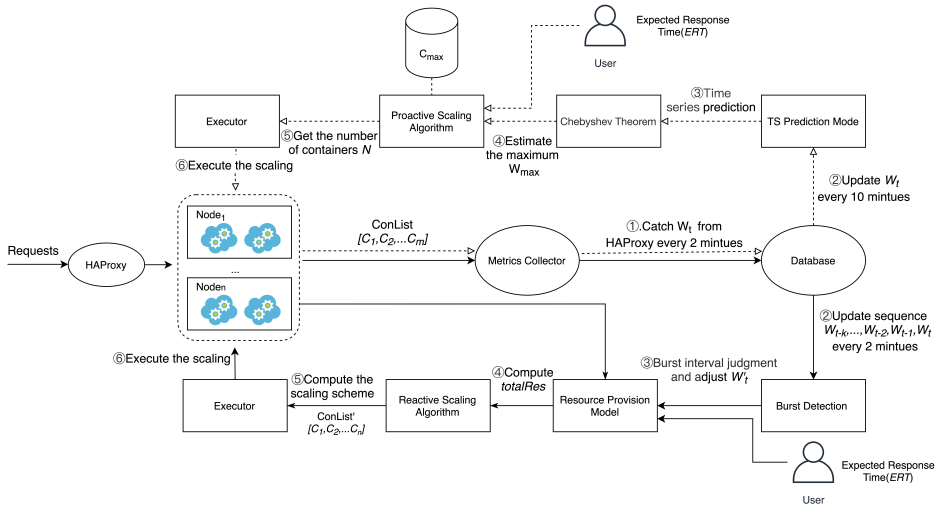


Figure 1. Overall architecture of AmRP

3.1 Overview of AmRP

The execution process of AmRP is shown in Figure 1. AmRP is an autoscaler that focuses on resource allocation at the container level. In Docker Swarm, the resources of containers are determined by the limit property. Therefore, the purpose of AmRP is to determine the number of containers and the resource setting. AmRP periodically executes the reactive module and the proactive module. The reactive module is implemented every short period, obtains the current number of requests from HAProxy in real time and performs burst detection. HAProxy[5] is a High-Performance TCP/HTTP Load Balancer. If it is in the burst interval at this time, the number of requests will be adjusted. The adjusted requests are used as input to the resource provision model. The resource provision model uses the number of requests and the expected response time from users as input to get the scaling scheme. If existing containers can be scaled vertically to meet the workload, AmRP will prioritize vertical scaling. If not, a combined scaling, that is, vertical and horizontal scaling is performed. The proactive module is executed every more extended period. It will perform time series prediction and estimation of the maximum number of requests used as one of the inputs to the proactive scaling algorithm. Only the number of containers is concerned in the proactive module.

[5] `https://www.haproxy.org`

3.2 Reactive Module

Reactive scaling module obtains the current number of requests W_t and the container list *ConList* containing resource settings for existing containers every 2 minutes and updates them to the database. The burst detection module obtains the requests sequence $W_{t-k}, \ldots, W_{t-1}, W_t$ from the database. The sequence will be judged, and the number of requests will be adjusted to better deal with the bursty workload if it is currently in the burst interval. The revised number of requests W'_t, the expected response time *ERT* set by users, and the container list *ConList* are used as the input of the resource provision model, and the output of the model is the total resource *totalRes*. The reactive module prefers vertical scaling when making scaling decisions because vertical scaling can increase resources faster than horizontal scaling.

3.2.1 Burst Detection

Workload burstiness is usually detected and judged by entropy-based methods [26, 27], but these methods are generally offline models and require a complete workload trace. For autoscaling, it is necessary to detect business in real time. In this paper, we choose an online model for burst detection. Abdullah et al. [24] also chose online bustiness detection, which detection standard was the standard deviation of sliding windows. The method adopted in this paper is the strategy of combining sliding window and boxplot [28], which can be divided into the following two steps:

1. Conduct surge point detection and judgment.
2. Determine the burst interval according to whether it is a surge point.

Lines 1 to 13 of Algorithm 1 show the function of surge point judgment. The first loop is to calculate a new time series S. It takes the mean value of the sliding window as the reference value ref_i, then takes the value l_i behind the sliding window and makes a difference between the two values. Line 8, on the new sequence S, calculates the boxplot, where Q_3 is the third percentile, IQR is the Interquartile Range, and c is the coefficient of IQR. Through the boxplot, we can get the upper bound. If it exceeds the upper bound, W_t is the surge point. Lines 15 to the end of Algorithm 1 determine the burst interval. Line 15 calls the function to determine the surge point. If the current is the surge point, then update *burstLen* to the preset length *Len*, indicating that the time points from the current point to the following length are burst intervals, and adjust the number of requests to W'_t. The reason for using this strategy to determine the burst interval is that the burstiness usually occurs continuously. Lines 20 to 27 represent that another judgment is required if it is not a surge point. When *burstLen* is greater than zero, it means that it is in the burst interval at this time, decrement the value by one and adjust the number of requests W'_t; otherwise, *burstLen* is set to zero.

Algorithm 1 Burst detection

Input: Remaining burst interval length ($burstLen$), time series ($W_{t-k}, \ldots, W_t$), window size (ws)

Output: $burstLen$

function SURGE POINT JUDGMENT($ws, W_{t-k}, \ldots, W_t$)
  $Flag \leftarrow false$
  for each i *in* $k - ws$ **do**
    $ref_i \leftarrow$ the average of the sliding window
    $l_i \leftarrow$ next value of sliding window
    $S_i \leftarrow ref_i - l_i$ // S is a new sequence
  end for
  $UpperBound \leftarrow Q_3 + c * IQR$
  if $S_t > UpperBound$ **then**
    $Flag \leftarrow true$
  end if
  return $Flag$
end function

$Flag \leftarrow$ SURGE POINT JUDGMENT($ws, W_{t-k}, \ldots, W_t$)
if $Flag = true$ **then**
  $burstLen \leftarrow Len$
  $W'_t \leftarrow max(W_{t-k}, \ldots, W_t)$
else
  if $burstLen > 0$ **then**
    $burstLen \leftarrow burstLen - 1$
    $W'_t \leftarrow max(W_{t-k}, \ldots, W_t)$
  else
    $burstLen \leftarrow 0$
    $W'_t \leftarrow W_t$
  end if
end if
return $burstLen, W'_t$

3.2.2 Resource Provision Model

Containers can be deployed without setting resource allocation. However, it will bring about resource competition between containers. Therefore, whether in Kubernetes or Docker Swarm, it is best to give the resource setting of each container. In this paper, the resource limit for a single container ranges from C_{min} to C_{max}. When a single container is scaled vertically, resources are adjusted according to fixed step size, such as adding 0.25 vCPU or 0.5 vCPU. When the resource of a single container reaches C_{max}, horizontal scaling will be executed to increase the resources.

In Resource Provision Model, the total CPU resources required will be solved. Then the specific scaling scheme will be given by the reactive scaling algorithm. The model's input is the revised number of requests W'_t, and expected response time ERT, and the model's output is the total resource amount $totalRes$.

$$\varphi : (W'_t, ERT) \rightarrow totalRes. \tag{1}$$

To collect trace data, we deploy the benchmark application (see Section 4.2) in the Docker Swarm. Then, *Hey Load Generator*[6] is used to simulate users sending requests. Then, we increase the number of requests linearly while adding resources with a reactive scaling strategy. For example, 0.25 vCPU is incrementally added to the existing container when the response time exceeds ERT. If all existing containers have reached C_{max}, the new container with C_{min} will be started. Finally, we will filter the data whose response time exceeds ERT, and the remaining data will be used for model training. Table 1 shows part of the trace data. The first row in the table indicates that when the number of requests is 804 and the total CPU resources is 10.5 vCPU, the response time is 0.1751 s.

Request	**Response Time**	**totalRes**
804	0.1751	10.5
895	0.1904	10.75
956	0.1958	11.25
...	...	...

Table 1. Trace data

In AmRP, we choose Random forest as the resource provision model. Random forest is an ensemble learning method for classification and regression that operates by constructing a multitude of decision trees at training time [29, 30]. For classification tasks, the output of the random forest is the class selected by most trees. In this paper, we use it to solve regression tasks, so the mean or average prediction of the individual trees is returned. Random decision forests correct decision trees' habit of overfitting their training set. In addition, the advantage of random forest is that its training speed is relatively fast, and it can balance errors for imbalanced data sets.

3.2.3 Reactive Scaling Algorithm

In the reactive module of AmRP, when performing scaling decisions, if vertical scaling can meet the current workload, vertical scaling is preferred. Compared to horizontal scaling, the execution of vertical scaling has a shorter duration. Therefore when burstiness is detected, vertical scaling increases resources and restores the response time to the ERT more quickly. Two methods detect the surge at Point 9 in Figure 2. Response time for vertical scaling recovers below ERT faster than for

[6] `https://github.com/rakyll/hey/releases`

horizontal scaling. Therefore, AmRP tends to perform vertical scaling in the reactive module. AmRP uses a combination of vertical scaling and horizontal scaling to increase resources only when existing containers cannot be scaled vertically. In the proactive module of AmRP, by predicting and estimating the number of containers needed in the future, containers with C_{min} resources are started in advance to ensure that the reactive module can perform vertical scaling in most cases.

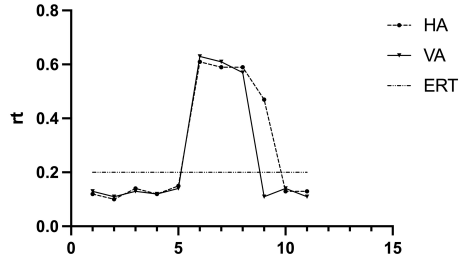


Figure 2. VA vs HA

Algorithm 2 describes the reactive scaling algorithm. The algorithm's input is the container information *ConList*, the adjusted number of requests W'_t, the expected response time *ERT*, and the output is the updated *ConList*'. Initially, the *ResourceProvisionModel* resolves the total current resource needs. Lines 2 to 24 of the algorithm are situations where it is necessary to increase the resources through vertical or combined scaling. $Count * (ConList) * C_{max}$ refers to the maximization of container resources that can reach through vertical scaling. Lines 3 to 14 indicate that adding resources to existing containers can meet the current workload. In order to adjust fewer containers when executing scaling, AmRP sorts *ConList* in ascending order according to the allocated resources and then increases the resources one by one. Lines 15 to 24 adopt a combined scaling strategy. First, set all existing containers to C_{max}. Then, the remaining required resource is added by starting new containers. In this paper, the new container's resource setting is C_{min}. Line 25 to the end of the algorithm is the case of reducing the resources. In the reactive module, only vertical scaling is used to reduce the resources. Since reserving a certain number of containers is conducive to vertical scaling, which increases resources more quickly. The operation of removing containers is performed in the proactive module. Similarly, to adjust only a tiny part of containers as much as possible, sort *ConList* in descending order and then reduce the resources one by one.

3.3 Proactive Module

The proactive module is executed every 10 minutes. The module first uses the *ARIMA* model to perform a multi-step time series prediction. Then, the predicted data is combined with a part of the historical data to form a new time series. The *Chebyshev's Inequality* is used to estimate the maximum number of requests. The

Algorithm 2 Reactive Scaling Algorithm

Input: Current container list ($ConList$), Adjusted Request at time t (W'_t), Expected Response Time (ERT)

Output: $ConList$'

```
totalRes ← ResourceProvisionModel(W'_t, ERT)
if totalRes > sum(ConList) then
    if count(ConList) * C_max <= totalRes then
        sort ConList by ascending order
        addRes ← totalRes − sum(ConList)
        for i ∈ Conlist do
            diff ← C_max − C_i
            if addRes >= diff then
                C_i ← C_max
            else
                C_i ← C_i + addRes
                break
            end if
            addRes ← addRes − diff
        end for
    else
        for i ∈ ConList do
            C_i ← C_max
        end for
        addRes ← totalRes − count(ConList) * C_max
        num ← ⌈addRes/C_min⌉
        for 1 ... num do
            ConList.append(C_min)
        end for
    end if
else if totalRes < floor * sum(ConList) then
    sort ConList by descending order
    removeRes ← sum(ConList) − totalRes
    for i ∈ ConList do
        diff ← C_i − C_min
        if removeRes >= diff then
            C_i ← C_min
        else
            C_i ← C_i − removeRes
            break
        end if
        removeRes ← removeRes − diff
    end for
end if
ConList' ← ConList
return ConList'
```

proactive scaling algorithm takes the maximum number of requests W_{max}, expected response time ERT, and the container C_{max} as inputs to obtain the number of containers required. Therefore, when the workload fluctuates wildly, AmRP will start enough containers in advance and then use vertical scaling to adjust resources in the reactive module. In addition, this module is also responsible for removing containers. When the number of requests decreases and becomes stable, AmRP performs scaling in to remove part of the containers.

3.3.1 Prediction Model

The prediction model in the proactive module is the *ARIMA* model commonly used in statistics [31, 32, 33]. *ARIMA* is widely used in time series forecasting. In $ARIMA(p, d, q)$, AR is autoregression, parameter p is the number of autoregressive terms; MA is moving average, parameter q is the number of moving average terms; I represent difference, parameter d is the number of differences to convert the non-stationary sequence into a stationary sequence. In the model, the value of the time series at the next moment is predicted based on the value observed in the past and random error. The specific formula is as follows, y_t represents the value at time t, and ε_t represents the random error at time t; φ_1 $(i = 1, 2, \ldots, p)$ and θ_j $(j = 1, 2, \ldots, q)$ are the coefficients of the AR and MA models, respectively.

$$y_t = \theta_0 + \varphi_1 y_{t-1} + \cdots + \varphi_p y_{t-p} + \varepsilon_t - \theta_1 \varepsilon_{t-1} - \cdots - \theta_q \varepsilon_{t-q}. \tag{2}$$

3.3.2 Chebyshev's Inequality

AmRP uses *Chebyshev's Inequality* to estimate the maximum number of requests W_{max} in the proactive module [34, 35]. This inequality generally applies to data of various distributions and is called *Chebyshev's Theorem*. The portion of any dataset that lies within k standard deviations of its mean is always at least $1 - \frac{1}{k^2}$, where k is any positive number greater than 1. When $k = 3$, it means that at least 88.9 % of all data is within three standard deviations of the mean. Therefore *Chebyshev's Inequality* can estimate the probability of an event if the distribution of the random variable X is unknown. The formula follows, where μ is the mean and σ is the standard deviation. $\mu + k * \sigma$ will be used as the estimated maximum W_{max} as one of the inputs to the proactive scaling algorithm in Section 3.3.3.

$$P(|x - \mu| \geq k\sigma) \leq \frac{1}{k^2}, \tag{3}$$

$$W_{max} \approx \mu + k * \sigma. \tag{4}$$

3.3.3 Proactive Scaling Algorithm

Algorithm 3 describes the proactive scaling algorithm. The input is container information *ConList*, the estimated maximum number of requests W_{max}, the maximum

resources of a single container C_{max}, and the expected response time ERT. The output of the algorithm is the updated $ConList'$. The algorithm begins with calculating the required total resource $totalRes$ under W_{max}. Line 2 estimates the required number of containers, which guarantees that the resources can be increased through vertical scaling in the reactive module. In lines 3 to 6 of the algorithm, AmRP will start additional containers, and the containers' resource is C_{min}. Lines 7 to 12 of the algorithm are used to remove excess containers. Similarly, $ConList$ is sorted first, then AmRP will terminate containers with relatively few resources.

Algorithm 3 Proactive Scaling Algorithm

Input: $ConList$, W_{max}, C_{max}, ERT
Output: $ConList'$

```
1: totalRes ← ResourceProvisionModel(W_max, ERT)
2: num ← ⌈totalRes/C_max⌉
3: if num > count(ConList) then
4:     for i in 1 ... num − count(ConList) do
5:         ConList.append(C_min)
6:     end for
7: else if num < count(ConList) then
8:     sort ConList by ascending order
9:     for i in 1 ... num − count(ConList) do
10:        ConList.remove(C_i)
11:    end for
12: end if
13: ConList' ← ConList
14: return ConList'
```

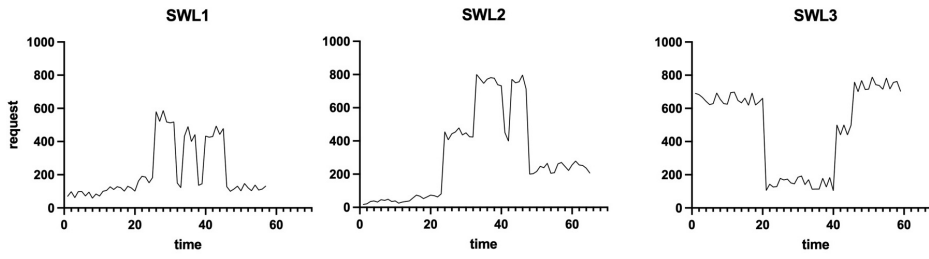
4 EXPERIMENT DESIGN

To evaluate our method, we design an AmRP prototype on the Docker Swarm platform and compare it with baseline scaling methods. All three scaling methods scale benchmark applications under various workloads. Methods are evaluated according to average response time, resource usage, and other indicators.

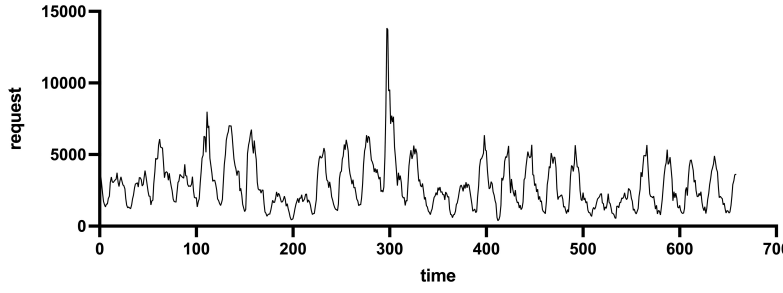
4.1 Baseline Methods

4.1.1 Base-Aware Predictive Autoscaling (BPA)

Abdullah et al. [24] proposed a burst-aware method based on proactive autoscaling, which is referred to as BPA in the rest of this paper. BPA adds burst detection based on standard deviation and sliding windows to traditional proactive autoscaling to measure burstiness in real time. If burstiness is detected, the number of instances is modified to the maximum number of instances in a nearby period. The regression



a) synthetic workloads



b) NASA workload

Figure 3. Workloads

model is used to predict the workload and the decision tree model is used to solve the scaling scheme. BPA provides a relatively stable resources when dealing with the workload with frequent fluctuations, which can better ensure the QoS and alleviate oscillation.

4.1.2 Reactive Method Based Queuing Theory (RMQ)

This baseline method is reactive autoscaling based on the Queuing Theory. In the remaining chapters of this paper, this method is referred to as RMQ. RMQ takes the processing capacity of a single service and the current number of requests as the input of the Queuing Theory model to analyze and solve and then obtains a reasonable number of instances in real time. For a non-bursty workload, this method can guarantee QoS and use fewer resources.

4.2 Benchmark Application

With the rise of microservices architecture, cloud services are becoming more and more fine-grained, such as image search, image recognition, document translation, video or audio decoding tasks, etc. Benchmark application selected in this paper

is a matrix operation, which is implemented based on PHP and packaged into containers to run in clusters. This application is a typical CPU-intensive task. In the cloud environment, most microservices are more sensitive to CPU resources, so this benchmark application is quite representative.

4.3 Simulation of Users

In the experiment, it is necessary to simulate the continuous and concurrent requests of users. In this paper, we choose to use the stress testing tool to simulate requests from users. *Hey Load Generator*[7] is an open-source stress-testing tool developed based on the Go language. To better verify the performance of the scaling algorithm under different workloads, this paper extracts some workloads showing burstinesses from the real data set, and also generates some SWL (Synthetic Workload). Figure 3 shows three SWLs and NASA workload [36]. It can be seen that burstiness occurs multiple times.

4.4 Evaluation Criteria

The selected evaluation indicators are resource usage, average request response time, SLA violations. The resource usage mainly refers to the usage of vCPU. The reason for not considering the memory is that the benchmark application is a CPU-intensive task with low memory requirements. Average request response time and SLA violations can reflect the overall QoS.

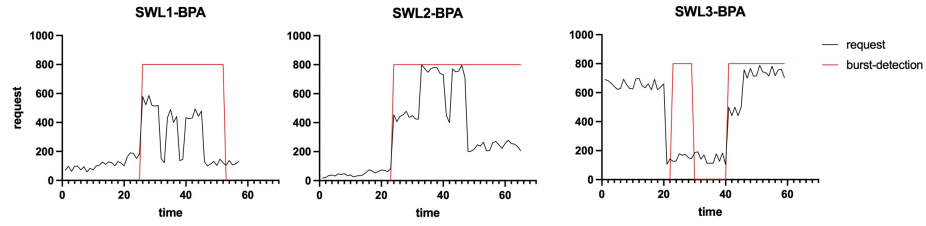
4.5 Experiment Platform

The experiment is carried out on Docker Swarm cluster, Docker version 20.10.12, where the CPU of the node is Intel(R) Xeon(R) Gold 6230 CPU @ 2.10 GHz, a total of 16 vCPU and 16 GB memory. In addition, since the stress testing tool consumes a lot of resources when sending requests concurrently, *Hey Load Generator* is independent of the experimental cluster and occupies a node with 4 vCPU and 4 GB exclusively.

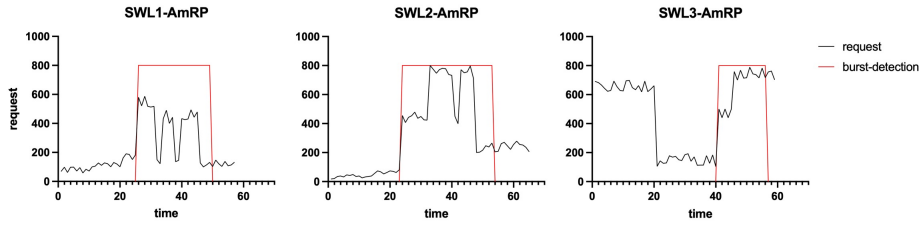
4.6 Experimental Parameters

Table 2 lists the key parameters in the experiment. There are three critical parameters in Burst Detection. First, we set k to 10, which means that we take the number of requests data adjacent to ten points as the detection input, ws is the window size of the surge point, and the setting of *Len* will affect the resource cost and QoS. If *Len* is too large, there will be some redundancy in resources, but the service quality can be better guaranteed, and vice versa. After some trial and error, we finally set this parameter to 10. Regarding container resources, we set the minimum and

[7] `https://github.com/rakyll/hey/releases`



a) Burst interval of BPA



b) Burst interval of AmRP

Figure 4. Burst detection

maximum resources of the container to 0.25 vCPU and 2 vCPU, respectively, and the step size of adjusting its resource amount is 0.25 vCPU. Since our application is a CPU-intensive service, there is very little demand for memory, and the impact of memory is not considered in this experiment. The last parameter in the table is the parameter of the *ARIMA* model.

Affiliation	**Parameters**	**Values**
Burst Detection	k	10
	ws	2
	Len	10
Container Resource	C_{min}	0.25
	C_{max}	2
	$stepSize$	0.25
ARIMA	(p, d, q)	$(2, 0, 0)$

Table 2. Part of experimental parameters

5 RESULTS

5.1 Burst Interval

AmRP and BPA both contain the burst detection module. Figure 4 shows the detection of burstiness under three SWLs by two methods. For SWL1, AmRP and BPA were almost identical in determining burst interval. AmRP set $[23, 33]$ ($Len = 10$) as the burst interval after detecting the surge point at the 23^{rd} point. Then AmRP detected two surge points, which updated the end of the burst interval twice. Therefore, for SWL1, AmRP detected the burst interval as $[23, 50]$. Similarly, BPA detected that the standard deviation of SWL1 exceeded the preset threshold at the 23^{rd} point. The standard deviation was lower than the preset threshold at the 53^{rd} point, and the number of requests at this time was lower than the moment before the start of the burst interval, so 53 was the end of the burst interval. SWL2 and SWL3 are processed differently by two methods. The determination of the burst interval by AmRP depends on the surge point and the preset length. Therefore, for SWL2, its burst interval detection is $[23, 54]$, while the detection of SWL2 by BPA belongs to burstiness from time point 23, and there is no end point. The reason is that at all time points after 23, the number of requests is higher than the moment before the first burstiness, so it is impossible to exit the burst interval. For SWL3, there is a sudden drop in the number of requests around time point 20, and since BPA uses the standard deviation, it is determined as a burstiness. The starting point of another burst interval detected by BPA is 40. Similar to SWL2, this interval also cannot be exited. The result of AmRP for SWL3 detection is that $[40, 57]$ is the burst interval. It can be seen that the burst detection of AmRP is guided by the surge point. When a surge point is detected, it enters the burst interval, and if there is no burstiness point for a long time, it exits the burst interval. The burst detection of BPA is oriented by the standard deviation, and its algorithm also marks the number of requests before the surge to exit the burst interval. For some workloads that have been stable for a long time after the burstiness, it will lead to certain waste of resources.

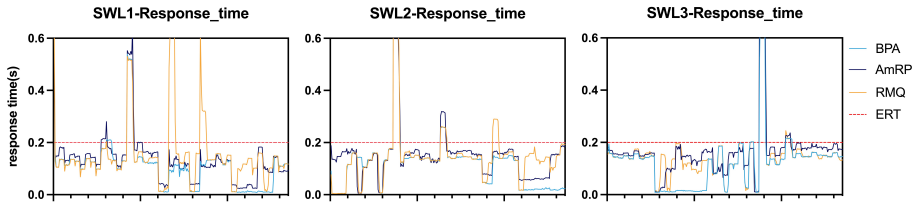


Figure 5. RT comparison under SWL

5.1.1 Response Time, SLA Violation

Figure 5. shows the response time comparison between AmRP and baseline scaling methods under SWL. To display the content of the figure more clearly, the upper limit of the response time is 0.6 s, which means that the 0.6 s in the figure may be greater than 0.6 s. There were three surges in SWL1. All scaling methods experienced a period of service degradation because the initial burstiness was difficult to predict and deal with. While AmRP and BPA benefit from burst detection, they will provide relatively stable resources once burstiness is detected. Therefore, the response time of AmRP and BPA did not exceed *ERT* in the subsequent two surges. However, the RMQ method does not consider bursty workload, so resources are frequently added/removed during burstiness, which makes its response time fluctuation obvious, and its quality of service is the worst among the three scaling methods.

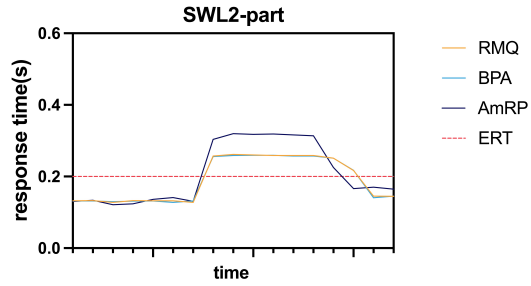


Figure 6. SWL2-Part

Similarly, in SWL2, all methods experience a drop in QoS for a while when the first surge occurs. After AmRP and BPA detected the burstiness, they were marked as surge status. A subsequent surge occurred shortly after that, and the number of requests exceeded the previous surge. At this time, it can be seen that the QoS of the three scaling algorithms will still decline because the strategy of AmRP and BPA in burst interval is to take the maximum resource amount near the time. Figure 6 shows the details of the second surge. AmRP restores the response time to *ERT* fastest among all methods. This benefits from the proactive module of AmRP that starts the container in advance. When the reactive module detects a surge, it increases resources through vertical scaling on existing containers. By comparison, the baseline method uses horizontal scaling to increase resources, so it takes more time to reduce the response time to the desired value. The three scaling ways in SWL3 and NASA (see Figure 7) are similar to SWL1 and SWL2. Table 3 summarizes the SLA violation of different methods. Except for SWL1, AmRP has the lowest SLA violation rate.

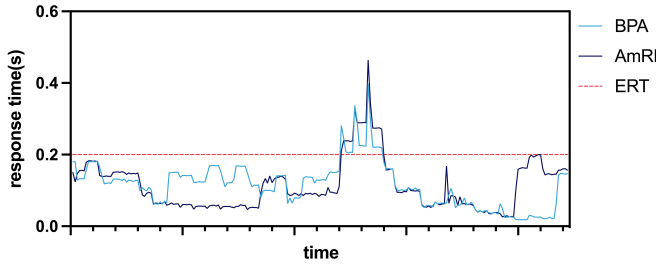


Figure 7. RT comparison under NASA

5.1.2 Resource Usage

Table 4 shows the total resource usage under various workloads. RMQ does not take bursty workload into account, and its resource usage is directly related to request trends. So RMQ is generally the scaling method that uses the least resources. AmRP and BPA give the benchmark application relatively stable resources in the burst interval, so these two methods use more resources than RMQ. The total resource usage of AmRP is lower than that of BPA, which benefits from the heterogeneity of the scaling scheme in AmRP and the finer granularity of resources. In addition, burst detection of AmRP can determine the burst interval more accurately than BPA.

Method / WL	AmRP	BPA	RMQ
SWL1	92.3 %	95.3 %	90.5 %
SWL2	95.3 %	95.1 %	92.9 %
SWL3	95.1 %	93.1 %	92.7 %
NASA	91.5 %	91 %	–

Table 3. SLA violations

Method / WL	AmRP	BPA	RMQ
SWL1	1 495	1 584	1 227
SWL2	2 227	2 400	1 830
SWL3	2 182	2 934	1 986
NASA	1 882	1 986	–

Table 4. Total resource usage

6 CONCLUSION AND FUTURE WORK

We propose a novel scaling method named AmRP for bursty workloads. AmRP is mainly divided into two parts, a proactive scaling module and a reactive scaling module. In proactive scaling, multi-step time series prediction and the maximum number of requests estimation are performed, and horizontal scaling is performed. This module aims to start enough containers in advance so that in the reactive module, vertical scaling can be executed on existing containers. The reactive module includes real time burst detection, and when calculating scaling solutions, vertical scaling is preferred. Experimental results show, compared with the baseline scaling algorithm, that for bursty workload, the AmRP scaling method further alleviates the QoS degradation caused by the surge of requests compared with BPA. As a result, AmRP can increase resources more rapidly and simultaneously provide stable resources within the burst interval. In terms of resource usage, since AmRP is a heterogeneous scaling solution, more fine-grained resource allocation further reduces the cost of AmRP resources.

Experimental results also show that the AmRP and baseline scaling methods are ineffective in dealing with the first surge or continuous surge because AmRP and BPA essentially do not predict the burstiness but only deal with them accordingly after detecting the surge. When the scaling method detects burstiness, the QoS declines, and the average response time exceeds the ERT. The solution in this paper is how to increase the resources more quickly after detecting a surge. Therefore, in future work, we will consider introducing burstiness prediction which can further improve the QoS of the scaling method in dealing with bursty workloads. However, its effect depends more on the accuracy of the burstiness prediction model.

REFERENCES

[1] Das, M.—Dash, R.: Role of Cloud Computing for Big Data: A Review. In: Mishra, D., Buyya, R., Mohapatra, P., Patnaik, S. (Eds.): Intelligent and Cloud Computing (ICICC 2019, Volume 2). Springer, Singapore, Smart Innovation, Systems and Technologies (SIST), Vol. 153, 2021, pp. 171–179, doi: 10.1007/978-981-15-6202-0_18.

[2] Sadeeq, M. M.—Abdulkareem, N. M.—Zeebaree, S. R. M.—Ahmed, D. M.—Sami, A. S.—Zebari, R. R.: IoT and Cloud Computing Issues, Challenges and Opportunities: A Review. Qubahan Academic Journal, Vol. 1, 2021, No. 2, pp. 1–7, doi: 10.48161/qaj.v1n2a36.

[3] Saadia, D.: Integration of Cloud Computing, Big Data, Artificial Intelligence, and Internet of Things: Review and Open Research Issues. International Journal of Web-Based Learning and Teaching Technologies (IJWLTT), Vol. 16, 2021, No. 1, pp. 10–17, doi: 10.4018/IJWLTT.2021010102.

[4] Coutinho, E. F.—de Carvalho Sousa, F. R.—Rego, P. A. L.—Gomes, D. G.—de Souza, J. N.: Elasticity in Cloud Computing: A Survey.

Annals of Telecommunications – Annales des Télécommunications, Vol. 70, 2015, No. 7, pp. 289–309, doi: 10.1007/s12243-014-0450-7.

[5] Chen, T.—Bahsoon, R.: Toward a Smarter Cloud: Self-Aware Autoscaling of Cloud Configurations and Resources. Computer, Vol. 48, 2015, No. 9, pp. 93–96, doi: 10.1109/MC.2015.278.

[6] Singh, P.—Gupta, P.—Jyoti, K.—Nayyar, A.: Research on Auto-Scaling of Web Applications in Cloud: Survey, Trends and Future Directions. Scalable Computing: Practice and Experience, Vol. 20, 2019, No. 2, pp. 399–432, doi: 10.12694/scpe.v20i2.1537.

[7] Burns, B.—Grant, B.—Oppenheimer, D.—Brewer, E.—Wilkes, J.: Borg, Omega, and Kubernetes. Communications of the ACM, Vol. 59, 2016, No. 5, pp. 50–57, doi: 10.1145/2890784.

[8] Casalicchio, E.: Container Orchestration: A Survey. In: Puliafito, A., Trivedi, K. S. (Eds.): Systems Modeling: Methodologies and Tools. Springer, Cham, EAI/Springer Innovations in Communication and Computing, 2019, pp. 221–235, doi: 10.1007/978-3-319-92378-9_14.

[9] Brewer, E. A.: Kubernetes and the Path to Cloud Native. Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15), 2015, pp. 167–167, doi: 10.1145/2806777.2809955.

[10] Nguyen, T. T.—Yeom, Y. J.—Kim, T.—Park, D. H.—Kim, S.: Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration. Sensors, Vol. 20, 2020, No. 16, Art. No. 4621, doi: 10.3390/s20164621.

[11] Qu, C.—Calheiros, R. N.—Buyya, R.: Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey. ACM Computing Surveys (CSUR), Vol. 51, 2018, No. 4, Art. No. 73, doi: 10.1145/3148149.

[12] Etemadi, M.—Ghobaei-Arani, M.—Shahidinejad, A.: A Cost-Efficient Auto-Scaling Mechanism for IoT Applications in Fog Computing Environment: A Deep Learning-Based Approach. Cluster Computing, Vol. 24, 2021, No. 4, pp. 3277–3292, doi: 10.1007/s10586-021-03307-2.

[13] Schuler, L.—Jamil, S.—Kühl, N.: AI-Based Resource Allocation: Reinforcement Learning for Adaptive Auto-Scaling in Serverless Environments. 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), 2021, pp. 804–811, doi: 10.1109/CCGrid51090.2021.00098.

[14] Golshani, E.—Ashtiani, M.: Proactive Auto-Scaling for Cloud Environments Using Temporal Convolutional Neural Networks. Journal of Parallel and Distributed Computing, Vol. 154, 2021, pp. 119–141, doi: 10.1016/j.jpdc.2021.04.006.

[15] Toka, L.—Dobreff, G.—Fodor, B.—Sonkoly, B.: Machine Learning-Based Scaling Management for Kubernetes Edge Clusters. IEEE Transactions on Network and Service Management, Vol. 18, 2021, No. 1, pp. 958–972, doi: 10.1109/TNSM.2021.3052837.

[16] Balla, D.—Simon, C.—Maliosz, M.: Adaptive Scaling of Kubernetes Pods. NOMS 2020 – 2020 IEEE/IFIP Network Operations and Management Symposium, 2020, pp. 1–5, doi: 10.1109/NOMS47738.2020.9110428.

[17] Taherizadeh, S.—Stankovski, V.: Dynamic Multi-Level Auto-Scaling Rules for

Containerized Applications. The Computer Journal, Vol. 62, 2019, No. 2, pp. 174–197, doi: 10.1093/comjnl/bxy043.

[18] Ding, Z.—Huang, Q.: COPA: A Combined Autoscaling Method for Kubernetes. 2021 IEEE International Conference on Web Services (ICWS), 2021, pp. 416–425, doi: 10.1109/ICWS53863.2021.00061.

[19] Moreno-Vozmediano, R.—Montero, R. S.—Huedo, E.—Llorente, I. M.: Efficient Resource Provisioning for Elastic Cloud Services Based on Machine Learning Techniques. Journal of Cloud Computing, Vol. 8, 2019, No. 1, Art. No. 5, doi: 10.1186/s13677-019-0128-9.

[20] Marie-Magdelaine, N.—Ahmed, T.: Proactive Autoscaling for Cloud-Native Applications Using Machine Learning. GLOBECOM 2020 – 2020 IEEE Global Communications Conference, 2020, pp. 1–7, doi: 10.1109/GLOBECOM42002.2020.9322147.

[21] Rattihalli, G.—Govindaraju, M.—Lu, H.—Tiwari, D.: Exploring Potential for Non-Disruptive Vertical Auto Scaling and Resource Estimation in Kubernetes. 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), 2019, pp. 33–40, doi: 10.1109/CLOUD.2019.00018.

[22] Phuc, L. H.—Phan, L. A.—Kim, T.: Traffic-Aware Horizontal Pod Autoscaler in Kubernetes-Based Edge Computing Infrastructure. IEEE Access, Vol. 10, 2022, pp. 18966–18977, doi: 10.1109/ACCESS.2022.3150867.

[23] Dang-Quang, N. M.—Yoo, M.: Deep Learning-Based Autoscaling Using Bidirectional Long Short-Term Memory for Kubernetes. Applied Sciences, Vol. 11, 2021, No. 9, Art. No. 3835, doi: 10.3390/app11093835.

[24] Abdullah, M.—Iqbal, W.—Berral, J. L.—Polo, J.—Carrera, D.: Burst-Aware Predictive Autoscaling for Containerized Microservices. IEEE Transactions on Services Computing, Vol. 15, 2022, No. 3, pp. 1448–1460, doi: 10.1109/TSC.2020.2995937.

[25] Tahir, F.—Abdullah, M.—Bukhari, F.—Almustafa, K. M.—Iqbal, W.: Online Workload Burst Detection for Efficient Predictive Autoscaling of Applications. IEEE Access, Vol. 8, 2020, pp. 73730–73745, doi: 10.1109/ACCESS.2020.2988207.

[26] Lassnig, M.—Fahringer, T.—Garonne, V.—Molfetas, A.—Branco, M.: Identification, Modelling and Prediction of Non-Periodic Bursts in Workloads. 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, 2010, pp. 485–494, doi: 10.1109/ACCESS.2020.2988207.

[27] Ali-Eldin, A.—Seleznjev, O.—Sjöstedt-de Luna, S.—Tordsson, J.—Elmroth, E.: Measuring Cloud Workload Burstiness. 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, 2014, pp. 566–572, doi: 10.1109/UCC.2014.87.

[28] Frigge, M.—Hoaglin, D. C.—Iglewicz, B.: Some Implementations of the Boxplot. The American Statistician, Vol. 43, 1989, No. 1, pp. 50–54, doi: 10.1080/00031305.1989.10475612.

[29] Biau, G.—Scornet, E.: A Random Forest Guided Tour. Test, Vol. 25, 2016, No. 2, pp. 197–227, doi: 10.1007/s11749-016-0481-7.

[30] Liu, Y.—Wang, Y.—Zhang, J.: New Machine Learning Algorithm: Random Forest. In: Liu, B., Ma, M., Chang, J. (Eds.): Information Computing and Applications.

Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 7473, 2012, pp. 246–252, doi: 10.1007/978-3-642-34062-8_32.

[31] Liu, C.—Hoi, S. C. H.—Zhao, P.—Sun, J.: Online ARIMA Algorithms for Time Series Prediction. Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 30, 2016, No. 1, pp. 1867–1873, doi: 10.1609/aaai.v30i1.10257.

[32] Ho, S. L.—Xie, M.—Goh, T. N.: A Comparative Study of Neural Network and Box-Jenkins ARIMA Modeling in Time Series Prediction. Computers and Industrial Engineering, Vol. 42, 2002, No. 2-4, pp. 371–375, doi: 10.1016/S0360-8352(02)00036-0.

[33] Mahalakshmi, G.—Sridevi, S.—Rajaram, S.: A Survey on Forecasting of Time Series Data. 2016 International Conference on Computing Technologies and Intelligent Data Engineering (ICCTIDE '16), 2016, pp. 1–8, doi: 10.1109/ICCTIDE.2016.7725358.

[34] Saw, J. G.—Yang, M. C. K.—Mo, T. C.: Chebyshev Inequality with Estimated Mean and Variance. The American Statistician, Vol. 38, 1984, No. 2, pp. 130–132, doi: 10.1080/00031305.1984.10483182.

[35] Amidan, B. G.—Ferryman, T. A.—Cooley, S. K.: Data Outlier Detection Using the Chebyshev Theorem. 2005 IEEE Aerospace Conference, 2005, pp. 3814–3819, doi: 10.1109/AERO.2005.1559688.

[36] Arlitt, M. F.—Williamson, C. L.: Internet Web Servers: Workload Characterization and Performance Implications. IEEE/ACM Transactions on Networking, Vol. 5, 1997, No. 5, pp. 631–645, doi: 10.1109/90.649565.

Qichen Huang received his B.Sc. degree in computing science and technology from Shanghai Ocean University, Shanghai, China, in 2019. He is currently pursuing the M.S. degree in the Department of Computer Science and Technology, Tongji University, Shanghai, China. His current research interests include services computing.

Song Wang received his M.Sc. degree from the Shandong University of Science and Technology, Qingdao, China, in 2016. He is currently pursuing his Ph.D. degree with the Department of Computer Science and Technology, Tongji University, Shanghai, China. His research interest is services computing.

Zhijun Ding received his M.Sc. degree from the Shandong University of Science and Technology, Tai'an, China, in 2001, and the Ph.D. degree from Tongji University, Shanghai, China, in 2007. He serves currently as Professor with the Department of Computer Science and Technology, Tongji University. He has published over 100 papers in domestic and international academic journals and conference proceedings. His research interests are in formal engineering, Petri nets, services computing, and mobile internet.