CPU resource co-op model using container metrics on microservice

Takamasa Iijima[1] Takayuki Kushida[1]

概要：Cloud applications such as Electronic Commerce sites have scalability. Microservice architecture is one of the architectures to make cloud applications. There is an increase in CPU usage due to the rapid increase in request rates in microservices. The method of adding an existing node adds a new node even though there is unused CPU on the existing node. When the request rate increases rapidly, the response time delay is caused by the addition of new nodes. This paper propose a CPU resource co-op model to reduce the frequency of node addition by improving the efficiency of CPU usage on existing nodes in order to maintain the response time when the request rate increases rapidly. The effect of request rate on CPU usage is examined by calculating the linear correlation between request rate and CPU usage for multiple microservices, and the microservices with the highest linear correlation values are selected for load balancing. Response time of requests generated based on the access logs of the article search service when accessed are evaluated at each request rate of 100[req/s] (the steady state) and 500[req/s] (the spike). Current scaling method requires about 50 seconds to recover from the response time of about 120[ms] at the time of a spike to about 35[ms] at the steady state, while the proposed method can recover in about 15 seconds. This proposal scaling method is 57.1% faster then kubernetes autoscalar method.

1. Introduction

Background

Users like smartphone users, TV listeners and PC users can upload own media to public internet ownself in 2022. Compare with legacy One-Way massmedia like radio station, TV stations migrate to new style. User Generated Contents(UGC) (e.g. Youtube, Twiiter and facebook) market is growing.

When users use these webservices, in actual they acceess to server to process their requests. Server architecure at cloud computing has two method. Monolitic Architecture and Micro Service Artchitecture(MSA). Monolitic architecture has integrated one code base for deloying service. On the other hands MSA has splited code base depends on developing groups. This paper use website as most simple example for cloud services. Website can't estimate number of requests will increase when massmedia introduce website like closeup TV program, news. Website have to increase number of server for increase number to respond to user at single time. MSA has mechanism named scale.

MSA, which divides the code base into different functions and allows for a high frequency of collaboration between business functions and development teams[1]. In the context of the use of cloud services, there is an expectation to flexibly adjust resources to dynamically changing traffic with as little human intervention as possible[2].

However, in the microservices used in cloud applications, in order to ensure scalability, it was necessary to explicitly formulate the specifications of each pod from the resources of each microservice, here the CPU usage.

In the case of microservices, code base of each service is independent of each other, and therefore, the container resources cannot be used in different ways. In conventional research and in the introduction of microservices in cloud applications, the priorities of services need to be predefined by humans. Therefore, human operations are also required for resource allocation in nodes.

Therefore, cloud applications at this stage need a mechanism to solve microservices issues. Cloud architecture using microservice has issues shows below. When the container has a "requests" parameter configuration of CPU threshold for scaling trigger, it cause waste for node CPUs.

[1] Tokyo University of Technology, Tokyo, Japan

When operator reduce ”requests” value, container scale more dynamically. For instance operator configure thousand milicore as one container can process thousand requests per seconds (req/s). When operator reduce service ”requests” value to ten milicore as one container, it can process 10 (req/s). Operator can prevent surplus provisioning of service when an access under the 10 (req/s). However when cloud application receive more than 10 (req/s) for example if receive requests as 100 (req/s), a container scale ten times. It makes ten times container creation delay and cause of network traffic jam.

Therefore, operator should take care of modify ”requests” parameter. This operator also should have heavy understanding about modify ”requests” and what it makes for cloud application.

Issue

When operator use default autoscaler, autoscaler deploy new application pod in current node. When cpu capacity of current node has no more free cpu resources, autoscaler add new node. Figure 1 shows current autoscale usecase. Issue is autoscaler add new node without consider other container free space.

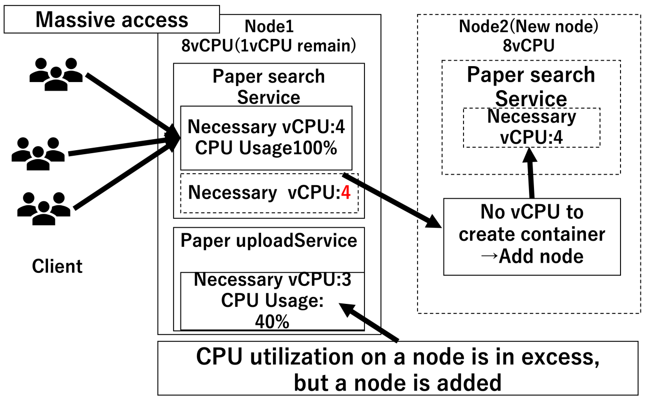


Fig. 1 Current scale method

2. References

In task scheduling by prioritizing queues in the cloud, tasks in the cloud environment are divided into queues and priorities are assigned to the queues using GA (Genetic Algorithm) [3]. This method calculated the priority of the tasks in a strict cloud environment. However, when this method is applied to microservices, it is difficult to schedule tasks according to this model because each function is connected to the network.

In Min-Min scheduling algorithm with user priority guide for load balancing in cloud computing, the Min-Min scheduling algorithm is used to prioritize the available resources per user over the resources of the whole cloud (i.e., cluster) for load balancing of cluster resources such as CPU, memory, etc. in the whole cloud which has many heterogeneous environments. The Min-Min scheduling algorithm is used to prioritize the resources that can be used per user over the resources of the entire cloud (henceforth, the cluster) [4].This method can manage resources by user unit in clusters, and in many cases by company or group unit. However, this method still has problems when functions are highly intercommunicated through networks such as microservices. In addition, this method allows users to manually teach the system a priority level called VIP level, which enables load balancing considering the priority of each task. However, in the case of microservices, where the number of tasks and services increases exponentially, these priorities should be set automatically.

Online Machine Learning for Cloud Resource Provisioning in Microservice Backend Systems proposes a method to scale microservice-specific applications [5]. In this proposal, machine learning is used to properly provision and scale each service. However, as they state in their discussion, they currently use only one metric, CPU usage. Therefore, they are not able to deal with databases where storage bandwidth usage is more valuable than CPU usage as a metric.

3. Proposal method

This paper propose a method to maintain the response time when the request rate increases rapidly by increasing the efficiency of CPU usage on existing nodes, reducing the frequency of node additions, and using another microservice compatible with the program on the same node to perform processing on behalf of the node. The proposal consists of the following five steps.

(1) Selection of high-impact metrics in the application
(2) Calculation of priorities in microservices
(3) Measuring the similarity between two different microservices
(4) Execution of programs in other services by sharing programs
(5) Load balancing functionality by redirecting to other services

Algorithm 1 Sort metricś value order by request rate

```
Input:
    $ML$:List which contain kind of metrics
    $R$:Hashmap which contain (Time t(time_t): request rate(req_rate))
    $M$:Hashmap which contain all of Hashmap(Time(time_t):Value of each metrics)
Output: $I$:Hashmap which contain Hashmap((reqests_rate):metrics) in each metrics
1: function METRICS_MAPPING_BY_REQ_RATE($ML, R, M$)
2:     ▷ Value of metrics and request rate pair
3:     ▷ with time which container in $R$
4:     for all $time\_t, req\_rate \leftarrow R$ do
5:         for all $metrics \leftarrow ML$ do
6:             $I.insert(metrics, \{req\_rate\colon M[metrics][time\_t]\})$
7:         end for
8:     end for
9:     return $I$
```

3.1 Selection of Highly Influential Metrics in the Application

First, identify which metrics in the application are affected by the request rate. In this paper, the two metrics to be obtained and used are CPU usage and RAM usage in this paper. This ranks the metrics that are most affected when the application experiences a spike in request rate. We obtain the CPU usage, RAM usage, and the time at which the metrics are obtained for each microservice on a node every second. Based on this information, we map the request rate to the CPU usage and RAM usage for each microservice. Table 1 below shows a table of the metrics (CPU usage) available from the microservices and their mapping to each request rate.

Time (s)	CPU Usage (milicore)	Request rate ([req/s])
1	1400	100
2	700	300
3	110	150
4	400	600
5	1000	140

Table 1 Get metrics each seconds.

Algorithm 1 is the process of linking multiple metrics obtained from microservices to their respective request rates using the time time time_t from which the metrics and request rates were obtained.

2 The type of metrics as input, e.g., CPU usage, RAM usage as input, ML which is a list of metrics, e.g. CPU usage, RAM usage, R which is a hash map of time on the system and request rate, and M which is a hash map of time on the system and values of each metric. The output is a hash map I containing the request rate measured for each metric and the metrics obtained at the same time. The function $METRICS\_MAPPING\_BY\_REQ\_RATE$ first retrieves from R the time $time\_t$ and the request rate req_rate for each element of R. Then, for each metric, $TIME_T$ is used as a key, and the values obtained for each request rate metric from M are inserted into I. Table 2 below shows a table of metrics (CPU usage) mapped by request rate.

Request rate ([req/s])	CPU Usage (milicore)
100	1400
140	1000
150	110
300	700
600	400

Table 2 Get metrics each requestrate.

Ranking of metrics by calculating correlation coefficients and regression lines Algorithm 2 uses I obtained in Algorithm 1 to calculate the importance of metrics in each microservice using correlation coefficients and regression lines from multiple metrics in each microservice and stores them in a list PoM. Then, sort the list in ascending order based on the calculated importance and create a final list PoM_Final that is the final priority of each microservice metric.

2

The input is a list I containing the values of metrics for each request rate created in Algorithm 1. The output is a list PoM_Final containing the priorities of microservice metrics in ascending order. The function $SELECT\_METRICS$ in the first line is described below. 3 and 4 lines recursively retrieve the request rate for each metric from I as a key and the value of each metric as a value. The set of request rates during the processing of each metric is denoted by x and the set of values of the metric by y. The correlation coefficient ρ is calculated for each metric and request rate using x and y.

$$\rho = \frac{\mathrm{cov}(X, Y)}{\sigma_x \sigma_y} \tag{1}$$

This study of monitoring whitepaper and cloud metrics states that correlation coefficients below 0.8 should be ignored in order to obtain only strong correlations using the correlation coefficients.[6],[7]. We perform the same classification as previous studies that artificially classified correlation coefficients into five levels and consider ρ^2 equal to or greater than 0.64 as correlated and insert them into the correlation metrics list PoM. The PoM created here is

Algorithm 2 Calculate metric rankings within each microservice

```
Input:
    $I$:List containing the values of metrics per request rate created in Algorithm 1
Output: $PoM\_Final$:Ranked list of importance of metrics
 1: function SELECT_METRICS($I$)
 2:     ▷ Calculate the correlation coefficient with the request rate for each metric
 3:     for all $metrics \leftarrow I$ do
 4:         for all $req\_rate, metrics\_value \leftarrow metrics$ do
 5:             $x(array) \leftarrow req\_rate$
 6:             $y(array) \leftarrow metrics\_value$
 7:             ▷ Calculate the correlation coefficient
 8:             $\rho = (1)$
 9:             ▷ Correlation coefficients with a square of 0.64
10:             ▷ or greater are considered valid correlations
11:             if $\rho^2 > 0.64$ then
12:                 ▷ Calculate the regression line
13:                 $r = (2)$
14:                 $PoM.insert([metrics, r])$
15:             end if
16:         end for
17:     end for
18:
19:     function GET_METRICS_RANK($PoM$)
20:         $PoM\_Final = \{\}$
21:         $max = 0$
22:         for all $metrics, r\_value, index \leftarrow PoM\_Final$ do
23:             if $PoM\_Final.Length() is 0$ then
24:                 $PoM\_Final.insert([metrics, r\_value])$
25:             else $selected\_metrics\_r > r\_value$
    $PoM\_Final[index - 1].insert([metrics, r\_value])$
26:             end if
27:         end for
28:         return $PoM\_Final$
```

used to execute the $GET\_METRICS\_RANK$ function. The $GET\_METRICS\_RANK$ function sorts the metrics stored in PoM in order of the slope of the regression line.

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} \qquad (2)$$

This sorting ranks the metrics by the degree to which their values affect the request rate. 3 points are awarded for first place, 2 points for second place, 1 point for third place, and so on. Points based on this ranking are later used to prioritize metrics across microservices.

As soon as all microservices have ranked their metrics, we add up the points of those that have been ranked, rank the names of the metrics that affect the request rate in the application, and multiply the correlation coefficient of the 1 ranked metric by the average request rate of each microservice since the application started. The correlation coefficient of the first ranked metric is multiplied by the average request rate of each microservice since the start

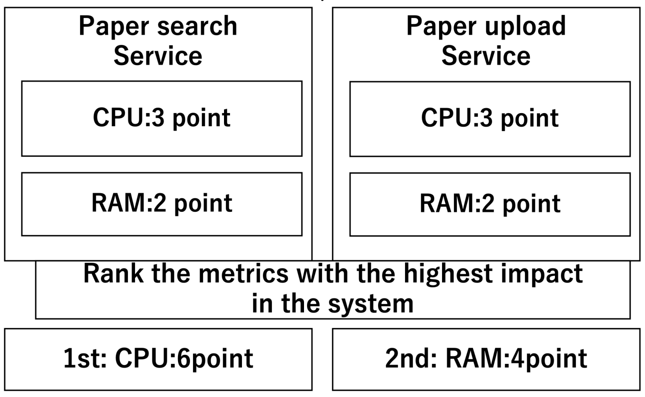


Fig. 2 ranking metrics across applications

of the application.

3.2 similarity measurement of two different microservices

In order to calculate the priorities in microservices, this paper compare the similarities between two microservices using the layers of the containers to find the containers that can help each other in the aforementioned priorities. The goal here is to compare the containers that make up the two microservices so that it can be determined whether or not it is possible to take over the processing of the parts of the program that are interchangeable.

container layer comparison

When the container images are the same for different services, they can be processed on behalf of each other without sharing code or programs. Each container is assigned an ID. If the container ID is the same for all strings, the processing can be done on behalf of the container. However, in a production environment, since each developer executes his own source code and programs, it is unlikely that the IDs are the same for all containers. Here, this paper will focus on the layers that make up a container (hereinafter called container layers). The container layer separates the read-only image part of a container image from the read-write image part, so that the OS and kernel parts are common, and the developer's original source code can overwrite the image.

3.3 Execution of other microservice programs by sharing source code or programs

If the layers other than the user's original source code section are the same, it is highly likely that the program can be executed by sharing the source code or program. The method of sharing source code and programs is described in the next chapter.

In addition, if the layers other than the user's original source code part are the same, this paper will perform a verification to improve the accuracy of the similarity. This paper compare the source files of the programs with the user's original source code in a text-based format.

3.4 Load balancing function by redirecting to other services

Based on the results of searching similar microservices, copy the program file from the microservice with the highest priority to the microservice with the lowest priority. After that, a load balancer is installed in the pod to which the program copy is sent to distribute the load.

After that, requests are redirected from the higher priority to the lower priority.

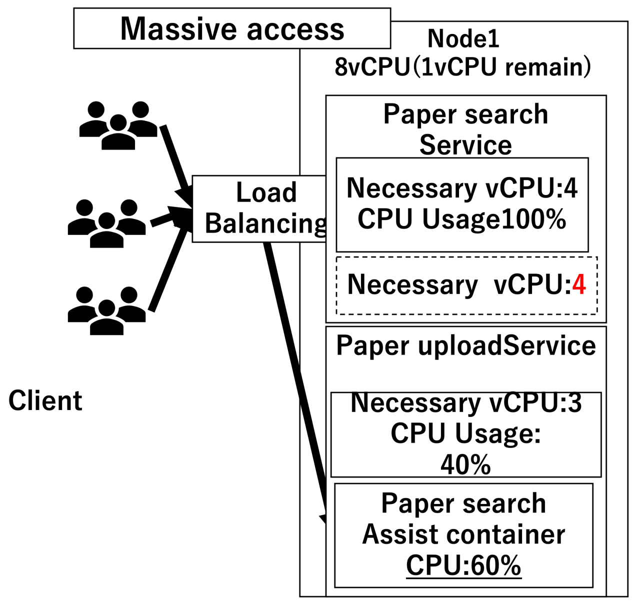


Fig. 3 Improves node CPU utilization by using unused CPUs within the same node

use-case scenario

As a use case scenario, this paper can confirm the effective utilization of resources and the acceleration of response time by using PoS in doktor, an article search site consisting of multiple microservices.

Doktor has two microservices. Suppose that a foreign researcher focuses on doktor, and the traffic increases rapidly. Then, the pdf service will be waiting for scale. In this paper, this paper propose PoS and program sharing to make the best use of the extra resources in the cluster.

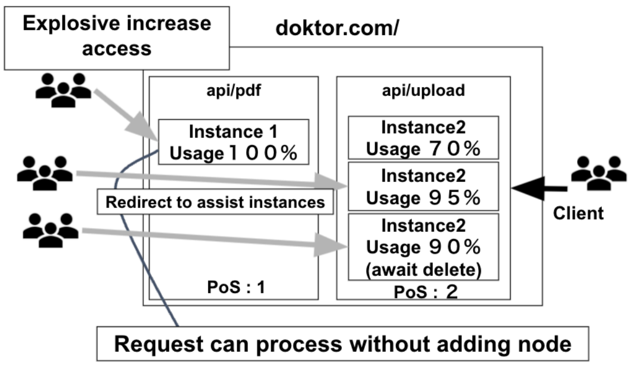


Fig. 4 Usecase

4. Implementation

As an experimental environment, we created a portal site of the thesis as an actual cloud service. In the experiment, we introduce the Mutual Aid Assistance Instance System to this service.

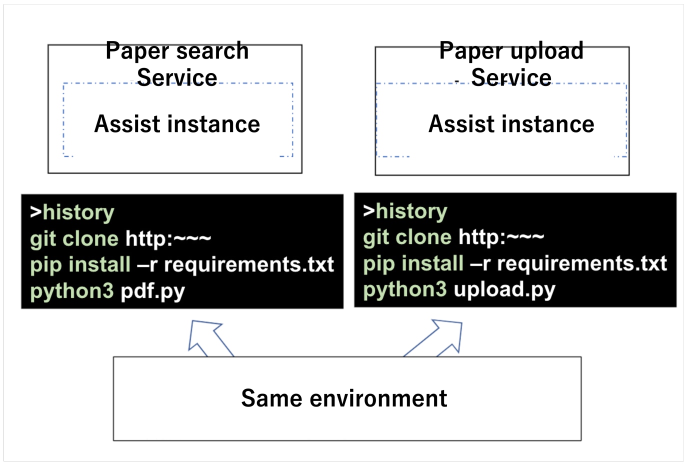
The environment in which this proposal is used is assumed to be based on a service mesh. All traffic in the microservice is monitored through Envoy, and metrics are obtained using datadog.

To simulate user requests, we send GET requests to a microservice built from Locust, a testing tool, in this case a payment service. When the payment service is accessed intensively and the CPU utilization exceeds a threshold, in this case 90

PoS-Calc measures the similarity between the services and checks if the program of another microservice can be executed. Here is an example of a cloud application I developed that mimics an article search site. We focus on the following two different services.

- Article search service A microservice processed by Python that displays articles according to the requests received by the API.
- Article submission service A microservice that stores uploaded article files in storage. It is processed by Python.

This section describes how these two services compare commands and improve the accuracy of similarity. In Figure 6 below, we assume that the code executed by the article search service and the article submission service are all identical when compared with the history command. In other words, all commands used in the program are identical. Therefore, we can say that these two containers have a high possibility of execution by sharing source code and programs. they have a high degree of similarity.



```
>history
git clone http:~~~
pip install –r requirements.txt
python3 pdf.py
```

```
>history
git clone http:~~~
pip install –r requirements.txt
python3 upload.py
```

Fig. 5 Ex:Co-op Available

5. Experiments

experimental environment

As an experimental environment, this paper have created a portal site for our thesis as an actual cloud service. In the experiment, this paper introduce a mutual aid assistance instance system into this service.

This paper assume that a service mesh is used as the premise of the environment for using this proposal. This paper assume that all traffic in the microservices is monitored via Envoy, and that metrics can be obtained using datadog.

The configuration table of these services is shown in Table 2 below. These services are composed of the following microservices (denoted as μS in the figure).

5.1 Building a Kubernetes cluster

First, install ESXi on the machine in table 3 as a virtual environment for creating node VMs.

CPU	AMD Ryzen 3950X
RAM	128GB
SSD	NVMe 2TB
NIC	Intel 1Gbps

Table 3 Spec: ESXi Server

Next, to build a Kubernetes cluster, we built three nodes with the specifications shown in the following table The OS used was microk8s, which comes with Kubernetes and Docker modules preinstalled by default.

5.2 Implementation of each service

The experiment was performed on a Kubernetes system with the following architecture.

- Python/Locust: Tool to automatically send requests

CPU	4vCPU
RAM	16GB
Storage	200GB
OS	Ubuntu 20.04

Table 4 Spec: each node

as a substitute for users and API users
- Flask(Scalable max3): Paper search Service
- Flask(Scalable max3):Paper upload Service

Set up a replica for each service, deploy a Kubernetes Service that performs load balancing, and access each Kubernetes Service. Table 2 below shows a schematic diagram of the experiments in this paper. To simulate user requests, we send GET requests to a microservice built from Locust, a testing tool, in this case, the search service. When the search service is accessed intensively and the CPU utilization exceeds a threshold value, in this case 90%, PoS-LB fires an event to the gateway API and changes the state of the gateway API so that the request to the search service is sent to PoS-LB once. Thereafter, requests for the search service are forwarded through PoS-LB to the search service and the article submission service that provides the resources.

Figure 7 below shows a diagram of the experimental configuration created based on the above configuration.

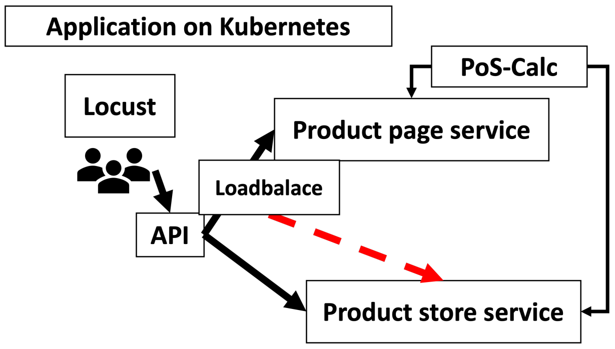


Fig. 6 Diagram of the Experiment

In this section, this paper calculates the expected number of accesses for the service used in this experiment. IEEE, the most famous academic society in the world, has about 5 million documents on IEEE Xplore, its paper search site. It also says that 25,000 documents are added every month. Assuming that each paper has a co-author, at least the author, and the co-author should access the website and check it. Therefore, this paper assumes that the number of accesses to the search portal is at least twice as many as the number of documents uploaded per month. In this case, 25,000 documents have been uploaded, so 50,000 is the base number of users browsing the portal each month. This translates to 0.019 requests per second. (50000/30(monthly)*24(hours)*3600(minutes and seconds)=0.019). In addition to this, if the conference was held every week, there is a possibility that users who attended the conference would view the papers. At the IEEE CCEM conference I attended, there were about 150 participants. If a user did this every week, an additional 600 people would be using the search portal for papers. Also, let's assume that the person who uploads a paper reads the references in the paper (in this paper, assumes about 20 references). In addition to that, for papers with more than 10 citations, this paper sets the number of monthly views to 2000. The reason for this is that the number of monthly views has been hovering around 2000 in the example of the IEEE metrics explanation site. Therefore in this paper define the average page view access rate to 386.22 request per second. Access rate of upload service is 0.009 request per second. It is almost once every 34 minutes.

evaluation method As an evaluation, we prepared an API in an environment where two microservices are running and accessed it according to the following request rate conditions. We compare the response times of the existing method and the proposed method when scale-out occurs. In this experiment, the constant request rate was set to 100 [req/s], and the peak request rate was set to 500 [req/s]. The target requests were limited to POST requests to the Web server. The experiment time was 180 seconds for the node utilization calculation experiment, and 100 [req/s] was sent for 60 seconds to remove unnecessary spikes when the first request was sent to the service, and the experiment was conducted from a stable state. After 60 seconds, the request rate was increased to 500 [req/s] to simulate an increase in requests. Each service was assumed to be able to run only one pod on each node, requiring expansion of another node when scaling out. The auto-scale threshold was set at 80The number of trials was set to 50 and 1000 to minimize measurement variability.

6. Evaluation

This figure shows the relationship between the CPU and the calculated PoS value.

The above ranking is also used to check whether CPUs are efficiently allocated among the nodes. In this study, we compared the existing and proposed methods by adding up the amount of CPUs in all nodes as the total CPU usage.

The response time from the API is compared before and after the introduction of this research model. It is assumed that the response time for services with high PoS will not increase compared to conventional clusters, even if there is a rapid increase in requests. During this experiment, the response times of services with low PoS will also be checked and discussed. We will also confirm that the frequency of node additions decreases as cluster utilization becomes more efficient.

7. Evaluation

Efficient use of nodes when PoS system is deployed Figure 9 shows the CPU utilization on the nodes prepared from the beginning of the experiment. The blue line shows the CPU utilization of the nodes with the stock Kubernetes autoscale, and the red line shows the CPU utilization of the nodes resulting from the implementation of our PoS system.

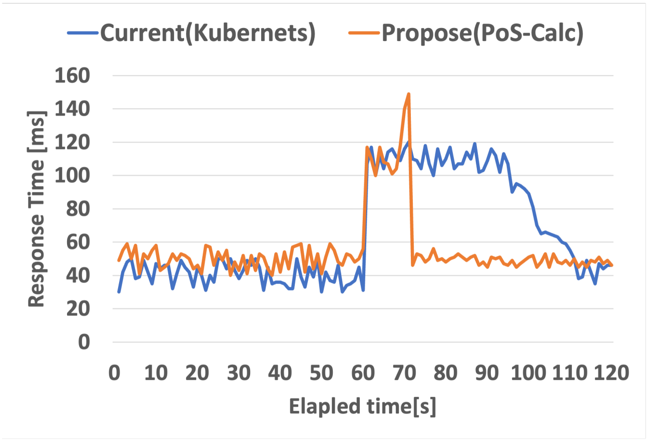


Fig. 7 CPU utilization measured at the first node in the two methods

During the first 60 seconds of the evaluation, requests reached the service at 100 [req/s], indicating that there was almost no difference between the two systems until 60 seconds. After that, Kubernetes' autoscaling functionality is activated, resulting in a significant drop in CPU utilization at around 70 seconds. The decrease in CPU utilization here is due to the fact that Kubernetes deployed the same service on a different node and distributed the load, and at 70 seconds, the system was scaling at a stage where about 10 percent of CPU utilization remained, indicating that the node resources were not being used up effectively. In comparison, the PoS system was deployed at 70 seconds, and the original service was no longer able to process all the requests, so the load was distributed to similar containers, which used the CPU utilization to the maximum extent before scaling. In this respect, the PoS system proposed in this study contributes to highly efficient utilization of the node for about 40 seconds compared to the Kubernetes autoscaler.

However, the CPU utilization after 130 seconds is not as low as that of the Kubernetes auto-scaler, even though the system is scaling. This is because the number of requests received by the system is higher when the PoS system scales to another node as a result of exhausting the system's resources to the limit, and when Kubernetes scales with room to spare.

Figure 10 shows a line graph of cumulative CPU usage when deploying up to a maximum of three nodes.

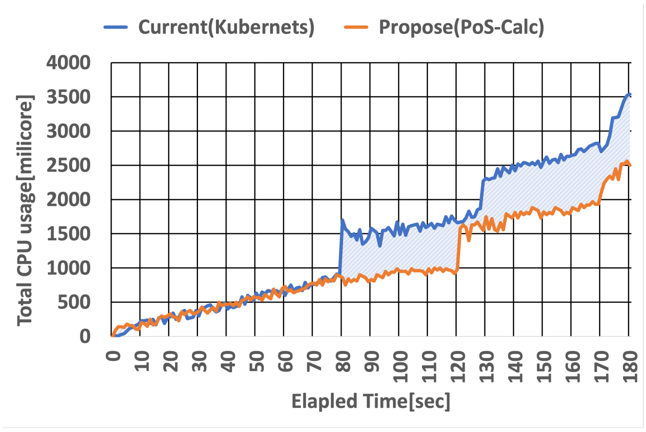


Fig. 8 Total CPU usage on all nodes

The shaded areas indicate areas where the PoS system reduces CPU utilization more than the existing Kubernetes Scaling. Figure 11 shows the CPU utilization of the first node when the number of trials is increased to 1000.

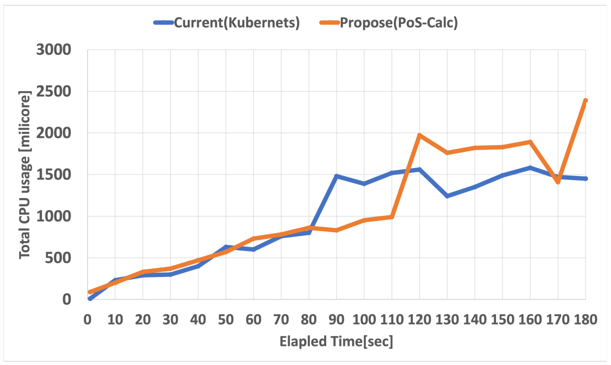


Fig. 9 CPU utilization of the first node when the number of attempts is set to 1000

Even when the number of trials is increased, the existing method and the proposed method significantly reduce CPU utilization by adding a new node at about 70 seconds from the start of the experiment. The difference between the timing of node addition by the existing method and the proposed method is 39.6 seconds on average, and this difference is expected to increase as the number of compatible microservices in the same application increases.

Maintaining response time when PoS system is deployed

Figure 13 shows the response time of the Kubernetes autoscaler when the request rate is increased from the start of the request, or when the PoS system's co-op process is started.

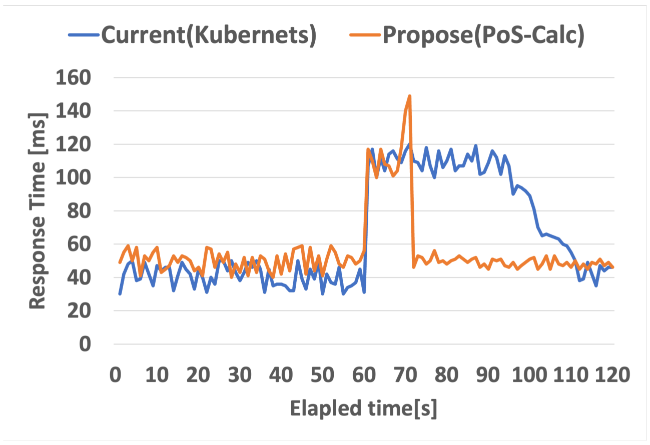


Fig. 10 Maintain response time when using PoS system

When the request rate increased from 100 [req/s] to 500 [req/s] after 60 seconds, the response time increased dramatically for both systems. However, when the PoS system was used, the load was distributed among the services in about 10 seconds and the response time decreased as the request rate approached 100 [req/s], while the Kubernetes autoscaler showed a dramatic increase in response time due to the time lag before the service was deployed on a new node. In contrast, the Kubernetes autoscaler showed a difference of up to 75 [ms] from the time when the PoS system started helping, which is about 40 seconds due to the time lag before the service is deployed to new nodes.

The following figure 12 shows Total CPU usage which testtime set to 1000. It can be seen that even when the number of trials is increased, the proposed method adds nodes less frequently than the existing methods.

Generally, a website is accessed rapidly immediately after the service is launched. Here, we simulate that 10 times as many requests are made in the first 10 seconds as in the steady state. Figure 13 reproduces a situation where the request rate is higher than the steady state for a certain period of time after a sudden increase in requests for a service with a steady state of 100[req/s].

Figure 13 shows that the scaling method using PoS-Calc

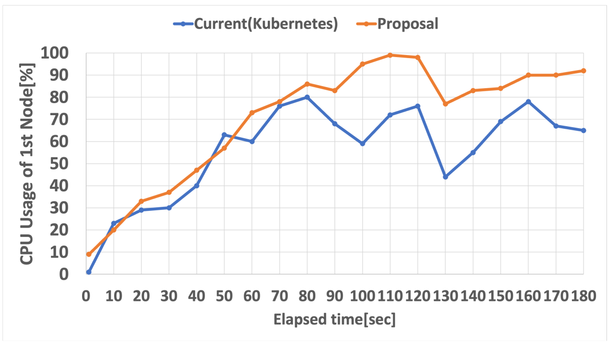


Fig. 11 Total CPU usage when the number of trials is set to 1000

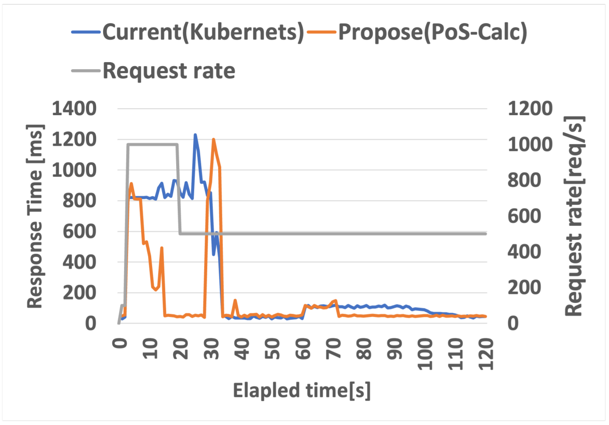


Fig. 12 Simulation peak

is faster than the existing auto-scale method in terms of response time recovery. This contributes to maintaining response time for large scale accesses compared to conventional methods.Proposed method spikes after 30 seconds is that the node has been added because the processing limit of the existing node has been exceeded.

7.1 Discussions

This proposal does not implement the timing for terminating the assist container when the number of requests processed decreases. The assist container should be terminated when the number of user accesses decreases, however since the number of user accesses and the metrics of the container are always variable, it is necessary to determine on the basis of which interval metrics the container should be terminated. In this case, the range of metrics to be acquired can be experimentally increased from 1 second (the shortest acquisition frequency) to 1 day or 1 week, and the scaling method can be made leaner by predicting the actual user accesses based on these models.

8. Conclusions

In this paper, we have considered the issue of increasing response time when scaling with additional nodes when adopting a MSA. We measured the response time of requests generated based on the access logs of the article search service when accessed at the request rate of 100[req/s] (steady state) and 500[req/s] (rapid increase), respectively. The existing method requires about 50 seconds to recover from the response time of about 120[ms] at the time of a spike to about 50[ms] at the steady state, whereas the proposed method can recover in about 10 seconds. In this study, we proposed a mechanism to automatically calculate priorities from running microservices. The proposed mechanism can suppress the response time delay when add nodes.

References

[1] Dragoni, N., Lanese, I., Larsen, S. T., Mazzara, M., Mustafin, R. and Safina, L.: Microservices: How to make your application scale, *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, Springer, pp. 95–104 (2017).

[2] Lehrig, S., Eikerling, H. and Becker, S.: Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics, *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*, pp. 83–92 (2015).

[3] Keshanchi, B., Souri, A. and Navimipour, N. J.: An improved genetic algorithm for task scheduling in the cloud environments using the priority queues: formal verification, simulation, and statistical testing, *Journal of Systems and Software*, Vol. 124, pp. 1–21 (2017).

[4] Chen, H., Wang, F., Helian, N. and Akanmu, G.: User-priority guided Min-Min scheduling algorithm for load balancing in cloud computing, *2013 National Conference on Parallel Computing Technologies (PARCOMPTECH)*, pp. 1–8 (online), DOI: 10.1109/ParCompTech.2013.6621389 (2013).

[5] Alipour, H. and Liu, Y.: Online machine learning for cloud resource provisioning of microservice backend systems, *2017 IEEE International Conference on Big Data (Big Data)*, pp. 2433–2441 (online), DOI: 10.1109/BigData.2017.8258201 (2017).

[6] Evans, J. D.: *Straightforward statistics for the behavioral sciences.*, Thomson Brooks/Cole Publishing Co (1996).

[7] Wang, T., Xu, J., Zhang, W., Gu, Z. and Zhong, H.: Self-adaptive cloud monitoring with online anomaly detection, *Future Generation Computer Systems*, Vol. 80, pp. 89–101 (online), DOI: https://doi.org/10.1016/j.future.2017.09.067 (2018).