Universiteit Leiden

Master Computer Science

Would Rather be More than Less:
On-Demand Container Resizing

Name:	Yuxuan Zhao
Student ID:	S2258609
Date:	05/08/2021
Specialisation:	Computer Science and Advanced Data Analytics
1st supervisor:	Alexandru Uta
2nd supervisor:	Nele Mentens

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Universiteit Leiden
The Netherlands

Master Thesis

Would Rather be More than Less: On-Demand Container Resizing

Yuxuan Zhao
(S2258609)

Supervisors:
Alexandru Uta & Nele Mentens

MASTER THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

05/08/2021

Abstract

Containers are lightweight virtualization and isolation environments for running applications. Typically, the resource allocation of containers, in terms of CPU, memory, network bandwidth, is set before deployment. However, the actual application resource usage changes dynamically while the resource allocated to containers are kept constant. This may contribute to applications termination or throttling owing to insufficient resource or resource under-utilization due to improper sizing. To solve this problem, the concept of on-demand container resizing was proposed, which indicates the resource allocations are able to change with changes in actual resource usage automatically. In this thesis, we focus on vertical pod autoscaling, as named by the Kubernetes community.

We observe that the autoscaling mechanism in the current vertical pod autoscaler may perform poorly in fine-grained time intervals, where the CPU usage is highly variable per minute, or even per second. Thus, in this thesis, we propose a finer-grained vertical pod autoscaling mechanism for Kubernetes and integrate it into the vertical pod autoscaler component to make it plug and play. We compare our autoscaling strategy with the default autoscaling strategy in the current vertical pod autoscaler component, showing that our strategy performs closer recommendation values to the actual container CPU usage without trend delay. Then, we compare our autoscaling mechanism with another autoscaling mechanism based on Holt-Winters exponential smoothing and Long Short-term Memory. Our autoscaling mechanism shows a lower average slack in YCSB workloads. Moreover, our autoscaling mechanism presents higher usability than HW-based and LSTM-based autoscaling mechanisms at least in short-term workloads due to our algorithms inexpensive computational complexity.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing technology has become prevalent in the current industry. Among the Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS), elasticity [1] of the cloud is an essential feature in the actual use. Elasticity is the capability of a cloud to adjust the configurations to control the resource provision in an on-demand style. Virtualization [2, 3] techniques are typically applied at the layer of Infrastructure-as-a-Service (IaaS). One can change the configurations of virtual machines to rightsize the virtual machines, achieving a better utilization. However, it turns to a problem that recreating virtual machines with new configurations is very costly [4]. To avoid the cost of turning on and off virtual machines, a lighter-weight virtualization technique, containerizaion [5, 6], is leveraged. The emergence of containers greatly eliminates the overhead of restarting virtual machines.

For most cloud computing workloads, resource demands such as those for CPU, memory, network, and disk change significantly over time. In containers, we set resource requests to specify how many resources should be allocated to that container. The resource requests are usually used for the container scheduling in the container orchestration platform like Kubernetes [7] and they are always required to be set manually before the container's deployment. However, it is not easy to accurately estimate in advance how many resources an application in the container needs to run gracefully. If the resources allocated to the container are too small, the applications in the container would be throttled when it exceeds its CPU limit or terminated when it exceeds its memory limit. But if the resources allocated to the container are redundant, the spare resources would be wasted, namely, leading to resource under-utilization. Furthermore, the requests of the container are constant but the actual resource usage changes dynamically due to the usage patterns of applications change over time. For instance, an application may need a lot of CPU and memory resources at the initial stage of deployment but with time going by, the demand for CPU and memory may decrease gradually and more resources would be spare resulting in unworthy resources under-utilization. Operators can change the resource requests manually but the maintenance is time-consuming.

Thus, if the resource request of a container can change following the resource demand changes automatically, the utilization rate would increase, throttling and termination of applications would be less, and maintenance time would be reduced. This technique is called autoscaling for the con-

tainer resource changes in demand. In the Kubernetes community, a component called vertical pod autoscaler [8], abbreviated as VPA, is responsible for setting the container requests automatically. Vertical pod autoscaling has the following advantages. First, pods can use the amount of resources they need, which leads to the high utilization of cluster nodes. Second, pods will be allocated to the node that has available resources. Third, users do not need to run benchmarks to estimate the resource requests. Fourth, vertical pod autoscaling can change the requests of CPU and memory dynamically, resulting in less maintenance time. This thesis aims to figure out if we can design a mechanism and policies for on-demand resource resizing for existing containers.

In this thesis, without loss of generality, we only focus on the CPU usage of containers. We propose as our contribution a mechanism for autoscaling in short-term workloads that are sensitive to the CPU usage changes at a per-second granularity. Inspired by the load prediction models in [9], our design includes two types of recommenders. The recommenders involve two linear models, one is based on the simple moving average model and the other is based on the exponential moving average model. Both models are responsible for smoothing the trend of the actual CPU usage by calculating the unweighted or weighted average of the recent CPU usage. Then we make a prediction for the future CPU demands through the unweighted and weighted average of the recent CPU usage obtained in the previous step.

We integrated both of our methods into the vertical pod autoscaler component. We also compared these to the recommender in the current vertical pod autoscaler. The current vertical pod autoscaler recommends the resource through a maintained histogram for every container. Furthermore, we compare them with a state-of-the-art autoscaling mechanism [10]. This autoscaling mechanism is based on Holt-Winters exponential smoothing method [11] and Long Short-Term Memory neural networks [12] to make a prediction for future CPU usage. All of the comparisons in this thesis are in the loading and running phase of different workloads in the state-of-the-art YCSB [13] benchmark designed for key-value store workloads.

To tackle our fine-grained vertical pod autoscaling problem, we propose three research questions in this thesis:

- How does the current vertical pod autoscaler work? What problems does the current vertical pod autoscaler have?
- How can we design an autoscaling policy which enables good performance for the application and minimizes resource waste?
- Compared to the current vertical pod autoscaler and autoscaling mechanism based on HW and LSTM, what are the advantages of our autoscaling design?

In this thesis, we propose the following three main contributions:

- We give a detailed analysis for the current vertical pod autoscaling mechanism.
- We propose an autoscaling mechanism for the container CPU usage, which outperforms current vertical pod autoscaling and HW, LSTM autoscaling mechanism at least in short-term workloads, where the CPU usage is highly variable.

- We integrate our autoscaling mechanism into the current vertical pod autoscaler component and make it plug and play. Moreover, the implementation code of our design and all the analysis code for the performance comparison are open-sourced[1].

[1] `https://github.com/ZhaoNeil/On-Demand-Resizing`

Chapter 2

Background

This chapter involves the background of this thesis. Firstly, we make a general introduction of cgroup [14, 15] because it is the foundation of containerization [5, 6]. Containers are primarily implemented through cgroups. Then, we compare two types of virtualization techniques [16]: hypervisor-based architecture and container-based architecture. Virtual machines [2, 3] and containers are implementations of these two techniques. Compared with virtual machines, containers are more lightweight [6] and are what we mainly focus on in this thesis. After that, we include the introduction of some related knowledge about docker [17, 18] and Kubernetes [7, 19], which are tools used throughout the experiment section. Specifically, we use docker to package the images of revised vertical pod autoscaler [8] and we store our images on the remote registry, Dockerhub. Kubernetes serves as a container orchestration platform in our experiments. We deploy related containers on a simplified version of Kubernetes, minikube [20], which only runs a single-node Kubernetes cluster on our remote server. The difference between Kubernetes and minikube is presented in detail in the following explanation. Afterward, we involve the introduction of two databases: redis [21] and mongoDB [22]. One is a key-value database and the other is a document-oriented database. A redis cluster serves as the application in the container and is connected with the YCSB [13] container on minikube. MongoDB is used for storing the containers monitoring data in our experiments. Lastly, we present the background of the YCSB benchmark, which serves as an application in a container on minikube as well.

2.1 Cgroup

Cgroup [14, 15] is the abbreviation of control group, which is a mechanism for hierarchically organizing resources allocation. It was firstly merged in the Linux kernel in the 2.6.24 version. The UNIX group of cgroup is cgroups. Cgroup can be applied to any schedulable entity like virtual machines and containers [16], ensuring that none of them consume all of CPU, memory, I/O bandwidth, and network bandwidth. As a feature of Linux kernel, cgroup allows users to limit resources, isolate, and account for the Linux processes or collections of Linux processes. For instance, users can limit certain resource usage by cgroups like specifying the number of the CPU core and controlling the amount of memory. If the process occupies too many resources that exceed the limitation, it will be suspended

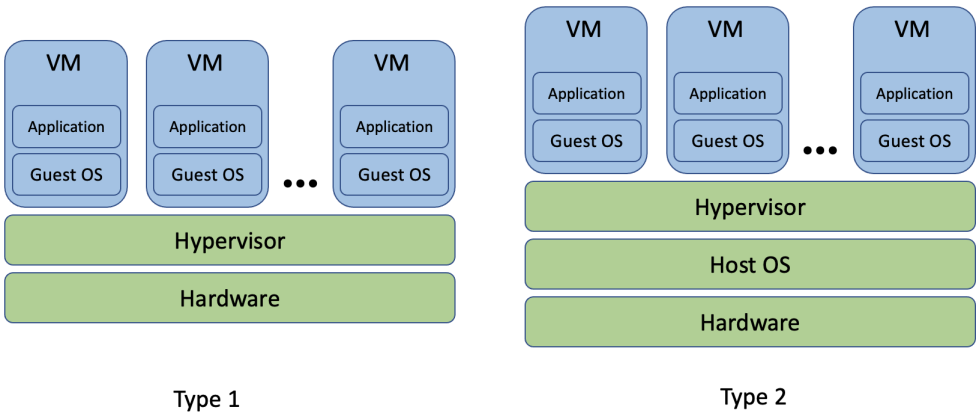


Figure 2.1: Two types of hypervisor-based virtualization architecture overviews [16, 17].

or killed. Cgroup mechanism is the foundation for Linux kernel to realize the resource virtualization technique. Similar to processes, cgroups are hierarchical and child cgroups can inherit attributes from their parent groups.

2.2 Virtual Machines & Containers

Virtualization [2, 3] is a technique allowing whole hardware elements like processor, memory, and storage to be divided into several parts for more efficient use. Each separated part is called a virtual machine (VM). A virtual machine can be regarded as a software-based computer, it has its own guest operating system and applications and runs like an independent physical computer. Since virtual machines are totally independent of one another, different operating systems can be run on different virtual machines. This feature makes the environment more flexible and more portable, at the same time enables the hardware utilization to be more efficient.

Hypervisor is an interface layer between the layer of physical hardware and virtual machines. It mainly functions as a resource manager for the virtual machines allocating the resources to them and ensuring there is no interference between them. There are two types of hypervisors, one is type 1 and the other is type 2 [16]. Type 1 hypervisors directly run on the top of physical hardware and they are also called native or bare-metal hypervisors. Type 2 hypervisors operate on top of a host operating system and are therefore called hosted hypervisors. Type 1 hypervisors are used more frequently and have a lower latency than type 2. Two types of hypervisor-based virtualization architectures are shown in Figure 2.1. As the Figure 2.1 shows, for type 1 hypervisor, independent virtual machines run on the layer of hypervisor which virtualizes the physical hardware. Each virtual machine serves as a physical computer, contains its own guest operating system and application, and it doesn't interfere with others. While, for the type 2 hypervisor, there is a host OS layer between hypervisor and physical hardware, which means virtual machines have to go through one more layer (host OS) to access the physical hardware.

Virtualization has a couple of benefits [3]. First, efficient use of resources. Since every virtual

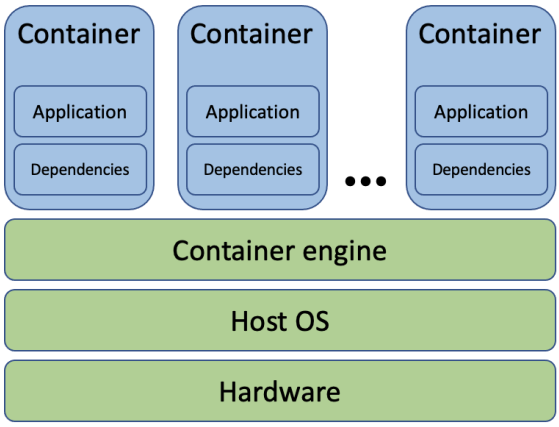


Figure 2.2: The container-based architecture overview [16, 17].

machine has its own operating system, this enables users to make a portable and flexible combination for their deployment, which would maximize the utilization of hardware resources. Second, improved fault tolerance. It's easy to backup and creates virtual machines, which makes the recovery after a severe down much simpler and allows to minimize the server downtime. Third, easier and faster deployment and more efficient management. Compared with setting up and configuring the physical hardware, deploying virtual machines is definitely simpler and faster.

Containerization [5, 6] is a technique that is always compared with virtualization but in a lighter-weight way, requiring less overhead. Container is an essential component of containerization, which is an isolated environment containing the needed elements, such as application code and its dependencies. Containers are mainly implemented through cgroups [14, 15]. Applications can run in containers in an isolated way, which means the applications are packed can be separated from their actual operating environment. Container supports share CPU, memory, storage, and network resource at the level of the operating system since containerization is an OS-level virtualization and containers share the host operating system instead of sharing the physical hardware. Moreover, containers can be deployed in any environment where the container engine is installed in a portable manner.

The container-based architecture is shown in Figure 2.2. Unlike type 1 hypervisor, there is a layer of host operating system on the top of the physical hardware layer. Container runtime engine is installed on the host operating systems to enable containers on top of it to share the host operating system. To reduce the overhead and the capacity of containers, some common dependencies can be shared among containers as well. On the layer of containers, each container encapsulates an application with its required dependencies like bins and libraries.

Containers have a couple of advantages [6]. First, containers have high portability. Containers are able to run virtually in any environment where the container engine is installed, greatly lightening the workload of development and deployment. Second, containers are lighter-weight. Instead of containing a guest OS in every virtual machine, containers only encapsulate related dependencies they need, which significantly eliminates the overhead of guest OS and speeds up, driving a higher efficiency. Third, containers have security because containers isolate the applications, virtualize CPU,

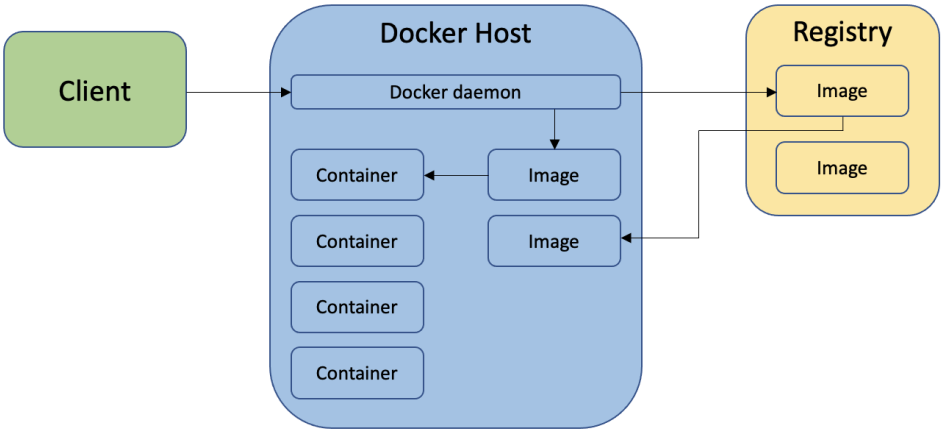


Figure 2.3: The basic architecture overview of docker [17, 18].

memory, storage, and network resources. In this way, an application in a container is prevented from interfering with other containers or the host OS. Fourth, more convenient management. A container orchestration platform like Kubernetes [7, 19] is able to create, deploy, and scale multiple containers in a very convenient manner.

In a nutshell overall, both virtualization and containerization are technologies that enable more efficient resource utilization but are different fundamentally. Virtual machines build isolated virtual environments by accessing the underlying physical hardware and sharing the resources of hardware. Containers are lighter-weight than virtual machines because containers pack applications and their required dependencies without a guest operating system. Containers virtualize at the operating system level, while virtual machines virtualize at the hardware level. Namely, containers share resources of the host operating system but require less setup time compared with virtual machines which share the resources of physical hardware. Virtualization tackles the pain point of the demand for using the entire server for an application. Containerization tackles the pain point of the demand for using the entire operating system for an application.

2.3 Docker & Kubernetes

Docker [17, 18] is a containerization software, which is used to create, deploy, and ship containers. Usually, the projects need various different technologies, for instance, node.js for font-end and mongoDB for database. Developers have to ensure these services are compatible with the version of the underlying operating system. Moreover, developers have to guarantee these different services are compatible with each other and libraries and dependencies on the operating system. The versions of the applications change over time, so the compatibility might change as well. However, docker can guarantee the applications that developers are going to deploy run in the same way and in the same environment by separating each application into containers with its own dependencies. In this way, developers do not need to pay attention to the software compatibilities.

Docker consists of docker client and docker host. Docker client is the main access for the users

to interact with docker. Docker host contains docker daemon and docker objects like images and containers [18] as Figure 2.3. Docker daemon is responsible for managing the containers, images, and volumes, etc. Docker daemon can pull docker images from a docker registry as well, such as Docker Hub. The basic architecture of docker is shown in Figure 2.3.

Kubernetes [7, 19] (called k8s as well), might be the most popular container orchestration platform, is from Borg [23] which is an internal large-scale cluster management system of Google and originally open-sourced by Google. If hundreds of or even thousands of containers communicate with each other or interact with each other and when multiple containers need to be deployed in different servers and different environments, it needs very much effort to handle these tedious problems manually in these situations. Moreover, some containers would crash or some containers would lose connection. Operators have to fix these issues one by one, which is time-consuming. As a result, Kubernetes came into being to manage multiple containers in a more efficient manner, in the meanwhile, guaranteeing high availability, scalability, and disaster recovery.

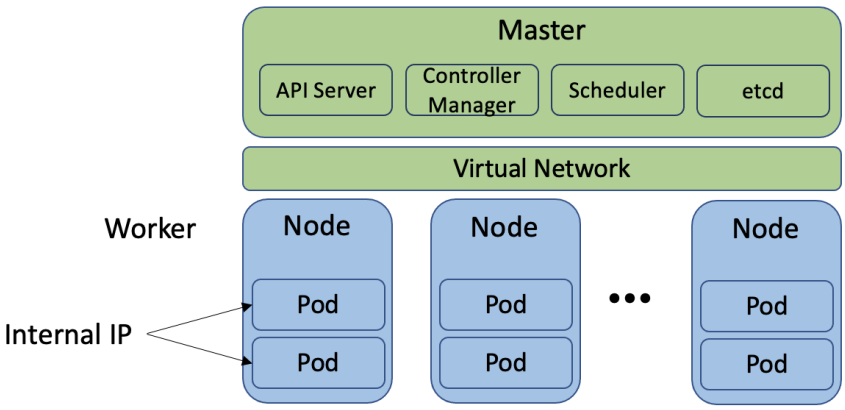


Figure 2.4: The basic architecture overview of Kubernetes [19].

The basic Kubernetes architecture is shown in Figure 2.4. Some necessary Kubernetes components [24] run on the master node, which are API server, controller manager, scheduler, and etcd. API server is the entrypoint to Kubernetes cluster. Controller manager is for controlling and managing the containers. Scheduler is responsible for container scheduling on different worker nodes. Etcd is a key-value storage storing the Kubernetes cluster states over time. Master node and worker nodes are connected through a virtual network. Pods are the smallest scheduling unit in Kubernetes and are an abstraction of containers, they run on the worker nodes. Pods are able to contain one or more containers but typically there is only one container in a pod. Each pod in Kubernetes has its own internal IP address and communicates with each other by addressing their internal IP. When the pods restart, their internal IP addresses would get changed. So another component of Kubernetes counts, which is service. Service actually is a constant IP address attached to pods. No matter how many times a pod restart, the IP would stay constant with the help of service. If users would like to expose the IP addresses of pods, the ingress component of Kubernetes can help to achieve that. If one does not have enough available resources to set up a Kubernetes cluster, minikube [20] is a good candidate.

Minikube can be regarded as a simplified version of Kubernetes, which is a one-node Kubernetes cluster whose master node process and worker node processes run on the same machine. The architecture of minikube is shown in Figure 2.5. The container runtime with docker engine is pre-installed on the minikube node.

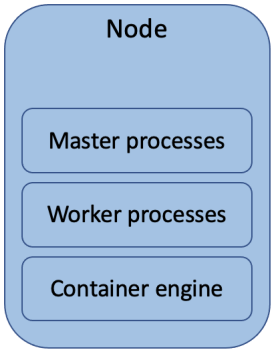


Figure 2.5: The basic architecture overview of minikube [20].

2.4 Redis & MongoDB

Redis [21] is an open-source key-value database. Data is stored in Redis in the form of key-value pairs. Redis supports multiple data structures to be the form of values, such as strings, hashes, etc. Leader follower replication [25] is an essential feature of redis cluster, it realizes backup of data, disaster recovery, and load balancing. Leader follower replication refers to copying data of one redis node to other redis nodes. The former one is called master node and the latter is called worker nodes correspondingly. Data replication is one-way in leader follower replication mode, which means data is only from the master node to the worker nodes. In this thesis, we deploy redis cluster benchmarked by YCSB [13] on minikube to test the resource usages fluctuate.

MongoDB [22] is classified as a NoSQL database. MongoDB is a kind of document-oriented database, which means it is similar to key-value storage but can be regarded as an upgraded version of key-value database. Data is stored in mongoDB in the form of documents. The documents look like JSON objects, consisting of multiple fields and value pairs. MongoDB supports other documents, arrays, etc. as values of the fields. In this thesis, we use mongoDB to store the monitoring data of containers resource usages, requests, and recommended values.

2.5 YCSB Benchmark

YCSB [13] is the abbreviation of Yahoo! Cloud Serving Benchmark, which is a standard benchmark framework to evaluate the performance of different cloud systems. It was developed by Yahoo! as its name indicates. YCSB consists of two main components, the YCSB client and the core workloads. The YCSB client is responsible for generating different workloads. The core workloads contain six different workloads for six scenarios. They are named in order from Workload A to Workload F and

each of these workloads has two phases, loading and running. In the loading phase, the database will be created and records will be loaded into the database. In the running phase, YCSB will operate the database as specified, the operations could be read, write, and modify. Different core workloads will have different proportions of operations. For example, workload A has 50% read operations and 50% write operations. Specifically, if we set record count to 2,500,000, which means 2,500,000 records will be loaded in database in the loading phase. In the running phase, if we set operation count to 2,5000,000 as well, which means there will be 1,250,000 read operations and 1,250,000 write operations due to the 50/50 proportion of workload A. The specifications of six core workloads are shown in Table 2.1. In this thesis, YCSB is used to make redis pod performance fluctuate.

Workload	**Specifications**
Workload A: Update heavy	50% read operations and 50% write operations
Workload B: Read mostly	95% read operations and 5% write operations
Workload C: Read only	100% read operations
Workload D: Read latest	Insert new records, the most recently inserted might read first
Workload E: Short ranges	Query small ranges of records
Workload F: Read-modify-write	Read a record, modify it, and write it back

Table 2.1: The specifications of six core workloads in YCSB.

Chapter 3

Methods

The recommender is an essential component in the vertical pod autoscaler. It provides the core resource usages (CPU and memory) estimation algorithm which recommends appropriate resource request values for pods in Kubernetes. As a result, the containers would resize according to the recommended resource values. Therefore, the quality of the recommendation algorithm largely determines the quality of the container resizing. An application would be throttled if it exceeds the specified CPU limit of the container and gets terminated if its memory exceeds the limited amount of the container. However, we only focus on container CPU demand variation in this thesis.

In this chapter, we introduce the resource estimation algorithm in the original vertical pod autoscaler in detail. The recommendation algorithm currently used in vertical pod autoscaler is deeply inspired by the moving window recommender in Google Borg Autopilot [26]. Then we include the autoscaling strategy proposed in [10], which primarily applies Holt-Winters exponential smoothing (HW) [11] and Long Short-Term Memory (LSTM) [12] algorithms to predict the future resource demands. Lastly, we present a resource estimation algorithm largely based on the ideas in CPU usage prediction models in web-based system [9] and bring forth new ideas into the algorithm. We have improved the algorithm to adapt to the usage scenarios of container resizing better. According to different CPU usage data processing methods, we divide our algorithm into two types: SMA-based and EMA-based. SMA represents simple moving average, responsible for getting the unweighted average of the recent CPU usage and the weights of observations are unchanged. EMA indicates exponential moving average, calculating the weighted average of the recent CPU usage while the weights of observations decay exponentially. Then, we make a prediction for the future CPU demands through the weighted and unweighted average of the recent CPU usage obtained in the previous step. To check whether they perform well in actual use, we implement our algorithms and integrate them into the vertical pod autoscaler component and evaluate the revised vertical pod autoscaler component in a real Kubernetes cluster. We show that our implementation works well and it is available in a plug-and-play fashion. Table 3.1 makes a summary of the methods used in this thesis and shows whether these methods are integrated into the vertical pod autoscaler component. Besides our newly implemented algorithms, the original VPA recommendation algorithm is already implemented into vertical pod autoscaler. As for HW and LSTM autoscaling strategy proposed in [10], they are not integrated into the vertical pod

autoscaler component, unfortunately.

Methods	Integrated into Vertical Pod Autoscaler
VPA recommender	Yes (existing)
HW recommender	No
LSTM recommender	No
SMA recommender	Yes (our contribution)
EMA recommender	Yes (our contribution)

Table 3.1: Methods considered and compared in this thesis.

3.1 Vertical Pod Autoscaler Recommender

The recommender of vertical pod autoscaler mainly borrows the ideas from moving window recommender in Google Borg Autopilot. Vertical pod autoscaler recommender creates a decaying histogram object for every container to store the CPU and memory usage. The recommender acquires the resource usage of all pods from Prometheus [27] regularly and writes the resource usage of containers into a maintained corresponding decaying histogram. The decaying histogram is composed of multiple buckets, which are used to store the weight of resource usage and the boundaries of buckets are the values of resource usage. The size of buckets in decaying histogram is growing exponentially with a ratio of 1.05. The first bucket stores the weights of resource usage in the range of $[0, firstBucketSize)$. Specifically, the first bucket size for CPU usage in decaying histogram in the current VPA recommender is set to 0.01 cores and for memory usage is set to 10MB. Since the bucket size grows exponentially, the starting value (left boundary) of the nth bucket follows

$$value(n) = firstBucketSize * \left(1 + ratio + ratio^2 + \ldots + ratio^{(n-1)}\right) = \frac{firstBucketSize * (ratio^n - 1)}{ratio - 1}.$$

The weight of every resource usage is stored in the bucket where the resource usage falls between the bucket boundaries (starting value and ending value of that bucket). Therefore, the index of the bucket where the weight of usage value was written is

$$index = int\left(log_{ratio}\left(\frac{value * (ratio - 1)}{firstBucketSize} + 1\right)\right),$$

value is the current usage value and the weight of current usage is

$$weight = Max(CPURequestCores, minSampleWeight) * 2^{\frac{time - begin}{CPUHistogramDecayHalfLife}},$$

the *minSampleWeight* here is set to 0.1 cores, the *begin* here refers to the time of the first recorded usage value, and $time$ means time of the current usage value. So it can be seen from the weight calculation formula that as time goes by, the weight of usage increases as well. Moreover, the default $CPUHistogramDecayHalfLife$ is set to 24h. It means under other conditions unchanged, the weight of the usage 24 hours ago will be halved.

VPA recommender maintains a decaying histogram composed of multiple buckets for each container as mentioned above. In the following, we explain how the VPA recommender uses the histograms to make predictions for future CPU demand. In terms of recommendation, VPA recommender involves three values: target value, lower bound value, and upper bound value. They are the starting values (left boundaries) of the bucket where the total weight of that bucket and the former buckets arrives at $0.9 * totalWeight$, $0.5 * totalWeight$, and $0.95 * totalWeight$ for the first time, correspondingly. For memory usage, VPA recommender watches the out of memory events in addition. VPA will increase the recommended values for the pods that had out of memory events. After that, VPA recommender adds some safety margin on target value, lower bound value, and upper bound value. The default safety margin is set to 15%, indicating these values have become 115% of the original so that containers can have some slack to breathe. Furthermore, VPA recommender applies a confidence multiplier to lower bound value and upper bound value. For lower bound value, it is multiplied by the factor $(1 + \frac{0.001}{history\_length\_in\_days})^{-2}$, so we get

$$lowerBound = lowerBound * (1 + \frac{0.001}{history\_length\_in\_days})^{-2}.$$

For upper bound value, the multiplier is $1 + \frac{1}{history\_length\_in\_days}$. Then we obtain the upper bound value after applying the confidence multiplier, which is

$$upperBound = upperBound * (1 + \frac{1}{history\_length\_in\_days}).$$

Vertical pod autoscaler evicts the pod as soon as its request value is beyond the range of upper bound and lower bound, then creates a pod with the current recommended value as its request. Table 3.2 and 3.3 display some sample multipliers yielded by the above lower bound and upper bound calculation formulas.

History Lengths	Lowerbound Multipliers
No history	0
5m	0.6
30m	0.9
60m	0.95

Table 3.2: Some sample multipliers of lower bound value for various history lengths.

History Lengths	Upperbound Multipliers
No history	INF
12h	3
24h	2
1 week	1.14

Table 3.3: Some sample multipliers of upper bound value for various history lengths.

For instance, when the history length is 1 hour, the vertical pod autoscaler will evict the pod when the pod request is lower than $0.95 * lowerbound$. Similarly, when the history length is 24 hours and the pod request is higher than $2 * upperbound$, the pod will be evicted by vertical pod autoscaler and a new pod will be created. From Table 3.2 and 3.3, we can see that both lower bound multiplier and upper bound multiplier are converging to 1.0 with time going by. However, lower bound multiplier converges much more rapidly than upper bound multiplier. For lower bound multiplier, it only needs 1 hour to converge around 1.0 while upper bound multiplier needs around 1 week to converge around 1.0. This is in line with the intuition that the recommendation value of resource usage would rather be more than less.

3.2 HW and LSTM Recommender

Thomas Wang et al. proposed an autoscaling mechanism in their paper [10], which applies Holt-Winters exponential smoothing (HW) [11] and Long Short-Term Memory (LSTM) [12] algorithms to increase the CPU utilization of the container. Their autoscaler takes target value, lower bound value, and upper bound value from HW and LSTM models as input and supplies 120 millicores as the error buffer. Millicores here is the same as the milliCPU. 1,000 millicores or milliCPU equals 1 CPU. In Kubernetes, 1 CPU refers to one vCPU on AWS or one vCore on Azure [28]. Their algorithm will give a new recommended value when the current CPU request is out of the range of bounds. They preset two values to avoid unnecessary rescaling. One is rescale cool-down value (18 time-steps), the other is minimum change check value (50 millicores). It means it will rescale only after at least 18 time-steps since the last time rescaling, in addition, the difference between the value of the current request and a new request must be more than 50 millicores.

Unfortunately, this autoscaling mechanism is not integrated into the vertical pod autoscaler. So its performance is only evaluated in a simulated way, out of the vertical pod autoscaler component. HW and LSTM models are implemented in Python with *Statsmodels* and *Keras*. We apply the actual CPU usage data obtained from the evaluation experiment of our recommenders as the input data of HW and LSTM recommenders. The input data are fed into these two models to refit or retrain the models and get the predicted values of future CPU usage. The final output of these models is the recommended value for the future CPU demands and is displayed in plot to compare with the actual CPU usage. The details of HW and LSTM recommenders can be seen in their paper [10]. All of the predictions are calculated out of the vertical pod autoscaling component through a python script. Thus, the results from HW and LSTM recommenders are not actually produced by a certain vertical pod autoscaler in real but are simulation results.

Moreover, HW and LSTM recommenders here are not plug-and-play because they need some time to ”warm up”. To be specific, HW and LSTM are only able to generate predictions after at least two seasons. Because HW and LSTM models need some data to initialize at the beginning. The season length is one day in their paper [10]. We present the general introduction of HW and LSTM algorithms in the following parts.

3.2.1 Holt-Winters (HW) exponential smoothing

Exponential smoothing methods [29] produce a smoothed time series data. Exponential smoothing methods smooth the historical data with exponentially decaying weights, which means recent data is attached a bigger weight than the older data. HW [11] is a triple exponential smoothing method involving three parameters to take care of not only the trend of data but also the seasonality of data. The formulas of Holt-Winters are shown as follow:

Overall smoothing:

$$S_t = \alpha \frac{y_t}{I_{t-L}} + (1-\alpha)(S_{t-1} + b_{t-1})$$

Trend smoothing:

$$b_t = \gamma(S_t - S_{t-1}) + (1-\gamma)b_{t-1}$$

Seasonal smoothing:

$$I_t = \beta \frac{y_t}{S_t} + (1-\beta)I_{t-L}$$

Forecast:

$$F_{t+m} = (S_t + mb_t)\, I_{t-L+m}$$

where y denotes the observation, S refers to the observation after smoothing, b indicates the factor of trend, I is the index of seasonality, F is the predicted value, and t refers to the time period. More details about HW implementation for autoscaling can be seen in the original paper [10].

3.2.2 Long short-term memory (LSTM)

Long short-term memory [12] is a special Recurrent Neural Network (RNN), which primarily aims at solving the problem of gradient vanishing and gradient explosion during the long-term sequence training process. Compared with vanilla RNN, LSTM can outperform in the longer sequences. A typical LSTM unit consists of a cell, an input gate, an output gate, and a forget gate. Cell is used for remembering values in any period. Three gates serve as regulators to control the input and output information flow of the cell. In other words, remember what needs to be remembered for a long time, forget the unimportant information. More details about LSTM implementation for autoscaling can be seen in the original paper [10].

3.3 Our Contribution: SMA-based and EMA-based recommender

Mauro Andreolini et al. [9] proposed load prediction models for CPU usage in web-based systems. Since the CPU usage workloads in Kubernetes are similar to web-based systems, both of them behave in a highly variable manner, we decide to adapt the models to better apply to vertical pod autoscaling. We innovate their algorithms while inheriting the ideas behind their prediction models. Moreover, we integrate the revised prediction models into the vertical pod autoscaler component and make it plug and play. The CPU measures are extremely variable at different time scales, especially in a short period. Mauro Andreolini et al. demonstrate in their paper [9] that it is almost impossible to predict future load well using the raw CPU usage measures. So they design load trackers, two linear

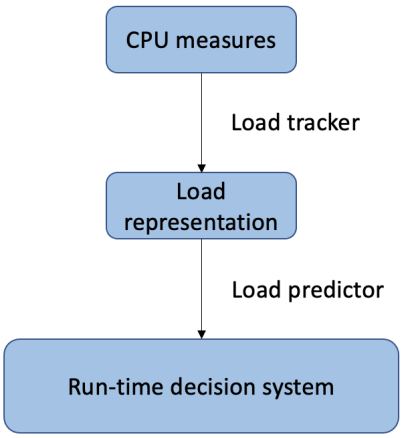


Figure 3.1: The basic framework of load prediction models [9].

functions to smooth the trend of CPU usage, representing the CPU load behavior of the system (see Section 3.3.1). Then, they make predictions of CPU usage through load representations processed by load trackers. The basic framework of the load prediction models looks like Figure 3.1.

3.3.1 Load trackers

As Figure 3.1 shows, the load prediction model achieves a two-step approach. In the first phase of the models, we still apply two linear load tracker functions presented in the original paper [9], which are simple moving average (SMA) load tracker and exponential moving average (EMA) load tracker. Given a CPU usage value s_i measured at time t_i and previously sampled n CPU usage values, they compose a set $S_n(t_i) = (s_{i-n}, \ldots, s_i)$. The load tracker function is defined as $LT(S_n(t_i)) : \mathbb{R}^{n+1} \to \mathbb{R}$, where the $S_n(t_i)$ is the input of the load tracker function and it returns a representation l_i to represent the set of $S_n(t_i)$ at time t_i. Simple moving average (SMA) is the unweighted average of $n+1$ CPU usage values in the set $S_n(t_i)$, the weights assigned to each observation are the same. So the SMA-based load tracker function at time t_i is defined as:

$$SMA(S_n(t_i)) = \frac{\sum_{i-n \le j \le i} s_j}{n+1}.$$

The problem of the SMA load tracker in theory is that it will involve a delay when it represents the workload trend, especially if the size of $S_n(t_i)$ is large. Whereas exponential moving average (EMA) load tracker function can decrease the delay effect well in theory. Exponential moving average (EMA) is the weighted average of $n+1$ CPU usage values in the set $S_n(t_i)$ and the weights of observations are exponentially decreasing. Thus, the EMA-based load tracker function at time t_i is defined as:

$$EMA(S_n(t_i)) = \begin{cases} \frac{\sum_{0 \le j \le n} s_j}{n+1} & \text{if } i \le n, \\ \alpha * s_i + (1-\alpha) * \text{EMA}(S_n(t_{i-1})) & \text{if } i > n. \end{cases}$$

The parameter α is called the smoothing constant. We conform with the constant value in the paper [9] and set the smoothing constant $\alpha = \frac{2}{n+1}$. For the load tracker based on EMA at time t_i, the recent observations contribute more to the representation l_i than the older observations due to the decaying weights.

3.3.2 Load predictor

In the second phase of the models, we adapt and innovate the load prediction in the paper [9] according to our usage scenarios. The load predictor function is defined as $LP(L_q(t_i)) : \mathbb{R}^{n+1} \to \mathbb{R}$, where $L_q(t_i) = (l_{i-q}, \ldots, l_i)$ is a set of $q+1$ representations obtained from load tracker function. $LP(L_q(t_i))$ returns the predicted future CPU usage value. $LP(L_q(t_i))$ follows

$$\begin{cases} LP(L_q(t_i)) = Max(1.5 * SMA(S_n(t_i)), \quad m*(i+k)+a), \\ SMA(S_n(t_i)) = \frac{\sum_{i-n \le j \le i} s_j}{n+1}, \\ m = \frac{l_i - l_{i-q}}{q}, \\ a = l_{i-q} - m*(i-q). \end{cases}$$

or

$$\begin{cases} LP(L_q(t_i)) = Max(1.5 * EMA(S_n(t_i)), \quad m*(i+k)+a), \\ EMA(S_n(t_i)) = \begin{cases} \frac{\sum_{0 \le j \le n} s_j}{n+1} & \text{if } i \le n, \\ \alpha * s_i + (1-\alpha) * \text{EMA}(S_n(t_{i-1})) & \text{if } i > n. \end{cases} \\ m = \frac{l_i - l_{i-q}}{q}, \\ a = l_{i-q} - m*(i-q). \end{cases}$$

According to the analysis in the paper [9] and our experiments, when k is equal to 2 times q, we can obtain a lower prediction error. Thus, we apply $k = 2*q$ here.

The fundamental idea behind the load predictor is conforming with the title of this thesis, which is the estimation for the future resource demands would rather be more than less. Because the application in a container will be throttled due to the insufficient CPU resource and be terminated due to the insufficient memory resource. Thus, in our prediction algorithm, we apply the bigger value between 1.5 times (unweighted or weighted) average of recent CPU usage values and linear extrapolation predicted value according to recent representations from the load tracker as our final predicted value. Since we introduce a multiplier $m = \frac{l_i - l_{i-q}}{q}$ in prediction, m will be close to 0 when the CPU load tends to be flat, which will lead to a "cliff" on the predicted values. Thus, we creatively introduce a "bottoming" mechanism to the algorithms. The adoption of 1.5 times (unweighted or weighted) average of recent CPU usage values prevents the predicted resource usage value from rapid decline unexpectedly and unreasonably. While linear extrapolation predicted value according to recent representations from load tracker ensures the predicted value to be capable to rise abruptly when CPU usage is peak. SMA-based and EMA-based recommenders have the advantage of their low computational complexity, so they are more suitable for a real-time processing system while keeping a promising prediction result.

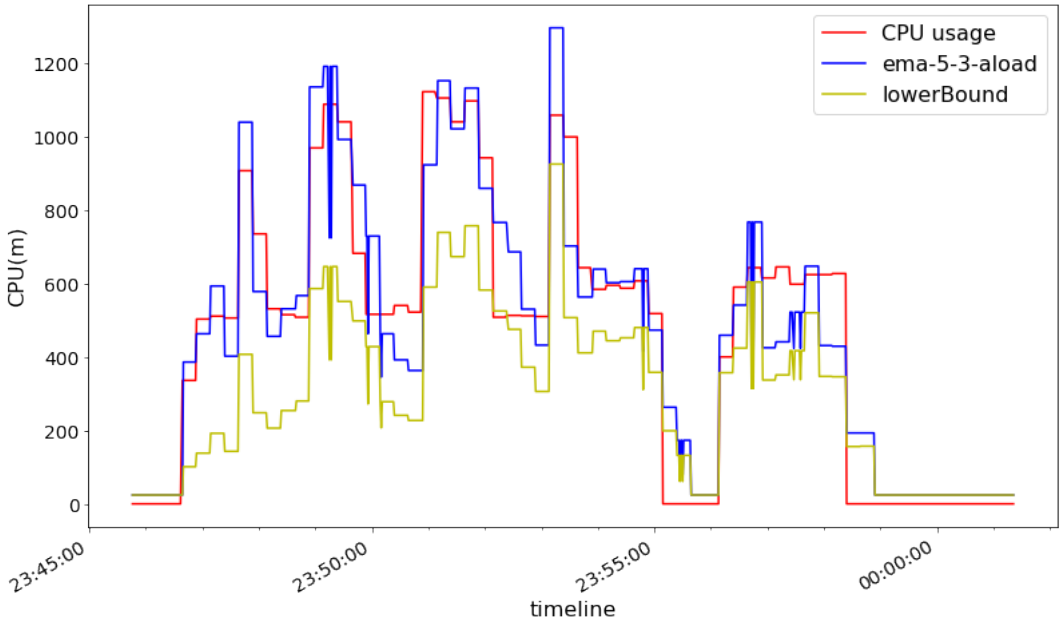


Figure 3.2: An example of the trends of predicted value and its lower bound.

In terms of update policy, we follow the design in vertical pod autoscaler. We keep the lower bound and upper bound for a recommendation value. However, what is different from vertical pod autoscaler is that we revise the calculation methods of lower bound and upper bound. In vertical pod autoscaler, the lower bound and upper bound are calculated as the starting values of the bucket where its accumulated weight achieves 50% and 95% of total weight. In our recommender, the upper bound and lower bound follow the trend of predicted value but with a multiplier like one in vertical pod autoscaler. Thus, the lower bound and upper bound will converge to the predicted value as time goes by like what they do in vertical pod autoscaler. The lower bound and upper bound follow as:

$$lowerBound = predictedValue * (1 + \frac{0.001}{history\_length\_in\_days})^{-2},$$

$$upperBound = predictedValue * (1 + \frac{1}{history\_length\_in\_days}).$$

When the request value of the pod is out of the range of lower bound and upper bound, the pod will be updated to a pod with a new request value that is the same as the recommendation value at that time. Figure 3.2 displays an example of the trend of predicted value and its lower bound.

Chapter 4

System Architecture

The basic architecture of our system is shown in Figure 4.1. Our experiment is conducted on a minikube cluster on Ubuntu 20.04. On this minikube cluster, three pods are deployed and we configure only one container running per pod. In this thesis, they correspond to the pod containing YCSB benchmark, the pod containing redis master node, and the pod containing redis worker node. The pods containing redis master and redis worker can establish a redis cluster in the leader follower replication mode [25] or in an isolated way without any connection. The details of deployment are demonstrated in A.2 and A.3. Out of the pods, a monitor is performed to monitor the resource usages of pods (or containers) in second granularity and write these resource usage values in an established mongoDB database (details in A.4). Furthermore, we enable vertical pod autoscaler to provide a mechanism re-scaling the containers resource request automatically. The main aims of vertical pod autoscaler are not only reducing the redundant resource wastage requested by containers but also reducing the probability of an application in the container being throttled or terminated due to insufficient resources. Vertical pod autoscaler primarily consists of three components, namely recommender, updater, and admission controller. In this thesis, we mainly focus on the recommender component. Our prediction algorithms are integrated into the recommender component and the performance of our algorithms is validated by configuring vertical pod autoscaler with replaced recommender component in a real minikube cluster. In the following sections, we demonstrate in detail how do we monitor the resource usage in Kubernetes and involve an overview of vertical pod autoscaler architecture [30].

4.1 Monitor

Resource metrics pipeline [31] in Kubernetes is shown as Figure 4.2. The values in the cgroup file are the ultimate sources of monitoring data. These monitoring data are collected by cAdvisor (container adsvisor) [32], which is a project open-sourced by Google. cAdvisor can collect the information of all the running containers on a machine, including CPU usage, memory usage, etc. Then, cAdvisor is integrated into kubelet [33]. Kubelet is an agent of each node in the cluster. Thus, metrics server [34] gets the resource metrics from kubelet and integrates the resource metrics into apiserver like metrics API [35]. Metrics API can also be accessed by kubectl commands as *'kubectl top'*.

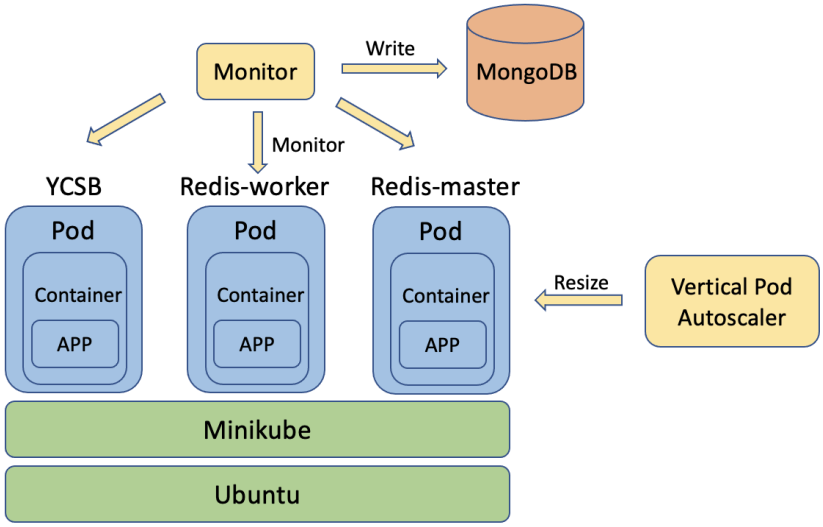


Figure 4.1: The architecture overview of our system.

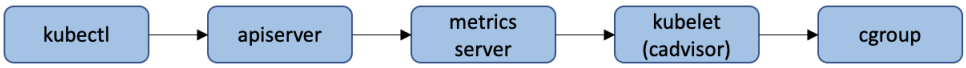


Figure 4.2: Resource metrics pipeline in Kubernetes [31, 34]

CPU and memory usages of containers are monitored through the *kubernetes.client.CustomObjectsApi* in the Kubernetes python client. Resource requests of pod and container are read via *kubernetes.client.CoreV1Api*. Resources recommended by vertical pod autoscaler are obtained from *kubernetes.client.ApiClient*. Table 4.1 lists three data types and their corresponding APIs where we monitor the resource values. All of the resource values mentioned above are monitored in second granularity and stored in a mongoDB database in real-time. CPU resource is measured in 'm', representing milliCPU. 100m CPU is equivalent to 0.1 CPU. In Kubernetes, 1 CPU refers to one vCPU on AWS, or one vCore on Azure [28]. Memory resource is measured in 'Ki' or 'Mi', 1 Mi equals 1024 Ki, 2^{20} bytes [36].

Data	API
Resource usages	client.CustomObjectsApi
Resource requests	client.CoreV1Api
VPA recommendations	client.ApiClient

Table 4.1: Data monitoring source.

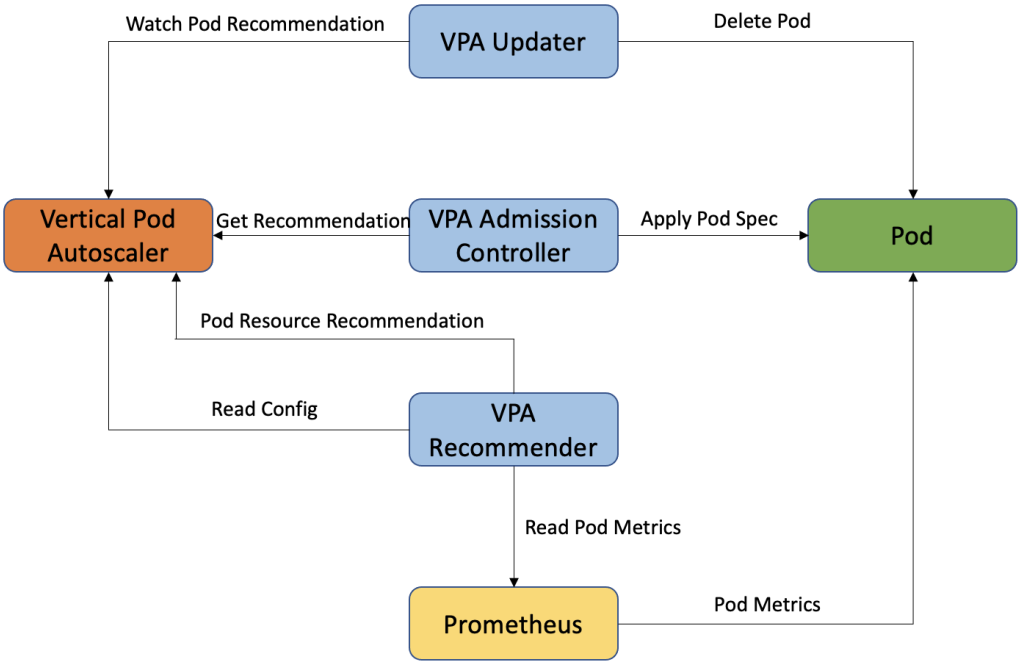


Figure 4.3: The architecture overview of vertical pod autoscaler [30].

4.2 Vertical Pod Autoscaler

The basic architecture overview of vertical pod autoscaler is illustrated in Figure 4.3. Vertical pod autoscaler is mainly composed of recommender, updater, and admission controller. Recommender watches all the pods in the Kubernetes cluster and calculates the recommendation values for each pod. The recommendation algorithm is included in section 3.1. Recommender reads the pod metrics from Prometheus [27] regularly. Updater is responsible for updating pods according to pod recommendations. When the current pod request is out of range between upper bound and lower bound, the pod will be updated by the VPA updater. Currently, the VPA updater only supports evicting old pods before recreating a new pod with the recommended resources. It means the service will be disrupted when a pod needs an update. Thus, in-place update [37] was proposed, which does not need to evict old pods anymore. But the in-place update is still under development. In terms of the VPA admission controller, it gets recommendations from the VPA recommender and sets it as the pod specifications.

Chapter 5

Experiments

In this chapter, we conduct the following experiments. Firstly, we set up our experiments environment, which are redis pods and YCSB pods on minikube. We connect redis pods with YCSB to make redis pods fluctuate in performance. We verify all of the pods on minikube work well and the connection between redis and YCSB is stable. Secondly, we validate the function of the current vertical pod autoscaler and its several limitations. We design the subsequent experiments according to the experience from this vertical pod autoscaling validation. Moreover, the current vertical pod autoscaling recommender is evaluated in the loading phase of workload A by three metrics, displaying there are still improvements to be done in vertical pod autoscaling. Thirdly, we evaluate the performance of our SMA-based and EMA-based recommenders in the real minikube cluster. Both of these two recommenders are integrated into the vertical pod autoscaler component and are plug and play. Three combinations of load track size and number for both SMA-based and EMA-based recommenders are evaluated. Three metrics are applied to evaluate performance and we pick the ema-5-3, which performs better, for the follow-up experiments. In the follow-up experiments, we verify if the ema-5-3 recommender is able to perform in the same way on other workloads of YCSB. For the poorly performing workload E, we adjust the multiplier in the algorithm and obtain better results. Lastly, we adjust the recommenders in the paper [10] slightly to make them suitable for our situation. We compare our ema-5-3 recommender with HW and LSTM recommenders. HW and LSTM recommenders are implemented out of vertical pod autoscaler to compare with our methods more conveniently.

5.1 Redis and YCSB Pods Validation

As mentioned in Chapter 4, we deploy a redis cluster consisting of one master node and one worker node on the minikube cluster. The redis master node is benchmarked by a YCSB pod. When the redis master pod loads or runs the workload in the YCSB pod, the CPU and memory usage of this redis master pod would fluctuate in theory. Thus, we want to validate the performance fluctuation of pods on the minikube cluster, at the same time, test if all the pods work well and if the connection between redis master pod and YCSB pod is stable.

We use command *'kubectl top'* to monitor CPU and memory usage. *'kubectl top'* needs to install

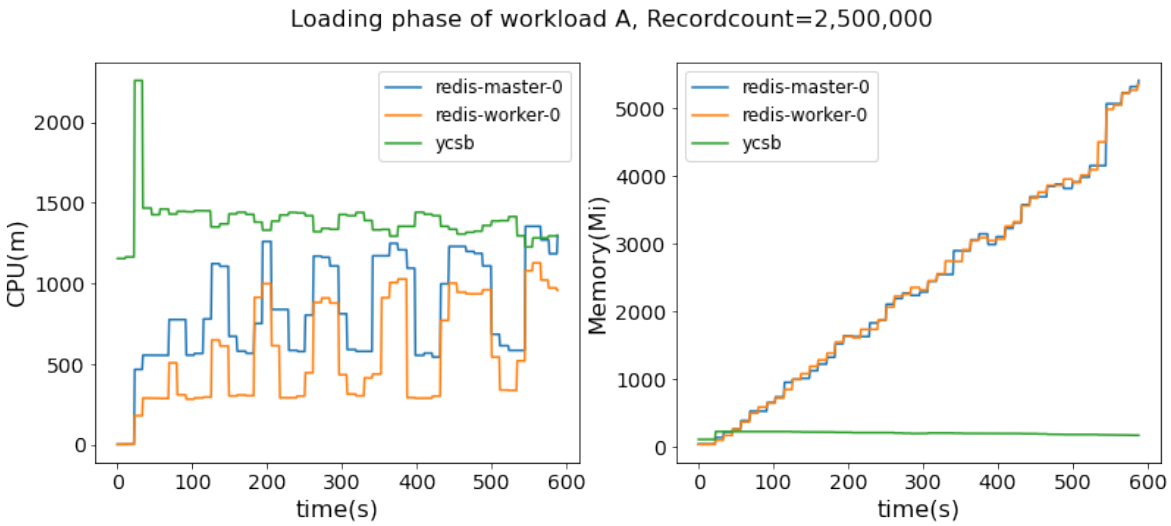


Figure 5.1: CPU and memory usage of pods in the loading phase, record count is set to 2,500,000.

a supporting addon, Kubernetes metrics server, to obtain the resource monitoring values. The same as units of container monitor, CPU and memory usages of pods are measured in milliCPU and Ki bytes. The CPU and memory usages of all pods are monitored in second granularity.

Figure 5.1 and Figure 5.2 show the CPU and memory usage of redis-master-0 pod, redis-worker-0 pod, and YCSB pod in the loading and running phases of workload A when the record count is set to 2,500,000 and operation count is set to 2,500,000 as well. Each pod contains a corresponding container. Redis-master-0 pod and redis-worker-0 pod compose a redis cluster. The master node is followed by a worker node and they are configured in leader follower replication mode [25]. The configuration for leader follower replication in the redis cluster is shown in A.3. From Figure 5.1 and Figure 5.2, we can see CPU usage fluctuates but memory usage presents a stepped increase which is in line with expectations. Moreover, it clearly shows that the trends of the redis master node and the redis worker node appear similar not only in the loading phase but also running phase. It indicates that the pods we deploy on minikube cluster work well and the connections between redis master pod and redis worker pod, as well as the YCSB pod are stable. Since the trends of redis cluster are the same, we will only focus on the pod containing redis master node in the following experiments for simplicity.

Conclusion-1: The pods deployed on minikube work well and the connections with each other are stable. Also, our monitoring framework is able to capture resource variability.

5.2 Vertical Pod Autoscaling Validation

In this section, we verify the function of the current vertical pod autoscaler and its several limitations. We configure the vertical pod autoscaler in ”Auto” mode on the redis master node. ”Auto” mode should have been an in-place update mode which avoids the restarts of the pod to be updated. But it has not been accomplished, the pods that need to be updated still have to be evicted first and then be

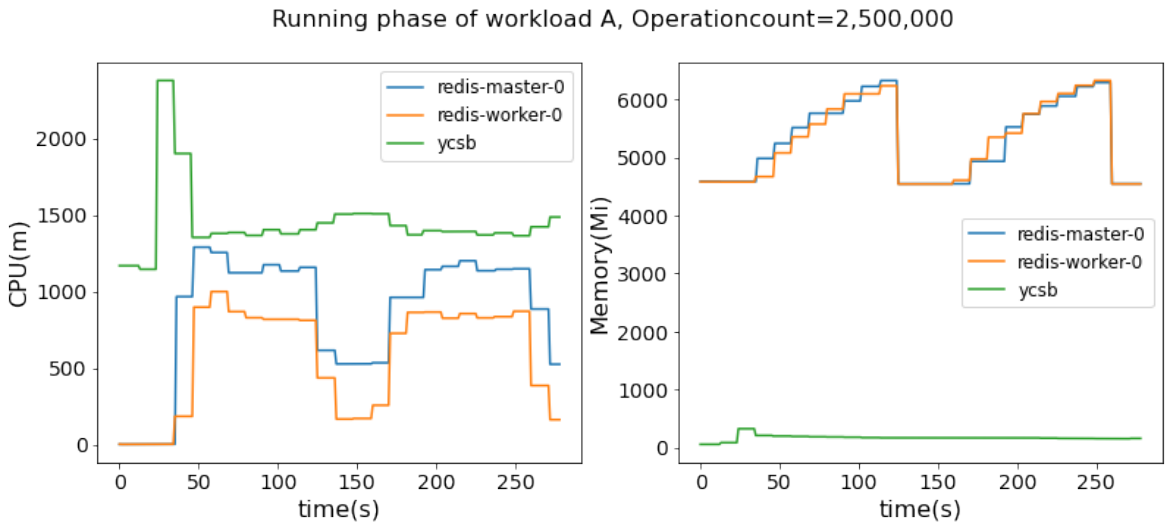


Figure 5.2: CPU and memory usage of pods in the running phase, operation count is set to 2,500,000.

created. Figure 5.3 obviously shows that there is a CPU request change on redis-master-0 pod and it works well after that change. We owe it to manually establishing the connection between redis master pod and the YCSB pod as soon as the redis master pod is recreated. Moreover, the figure presents the CPU usage and request of two redis master nodes. So now it brings out one of the limitations of vertical pod autoscaler, which is vertical pod autoscaler works normally only if the pod has at least two replications. It explains why there are two redis master pods in Figure 5.3, and we configure two replications of redis master pod to make sure vertical pod autoscaler works well. Despite there is one more replication for the pod, but the other pod replication does not show any changes at all. Another limitation of vertical pod autoscaler is that each pod can only correspond to one vertical pod autoscaler and vertical pod autoscaler can not be used with horizontal pod autoscaler at the same time. Thus, it implies we can not compare the recommenders on the same pod in one run, except the recommenders are not integrated into the vertical pod autoscaler, namely out of the VPA component.

According to the above facts, firstly, we decide to merely fix attention on the performance of the first replication that belongs to the target pod. Secondly, for the smoothness of our experiments, we decide to set VPA in ”Off” mode in the following comparisons of different recommenders. Vertical pod autoscaler in ”Off” mode only computes the recommended values but without automatic updates.

Following above-mentioned configurations, we deploy pods of the redis master node and YCSB on the minikube cluster and two pods connect with each other. We use the loading phase of workload A in YCSB to make the redis master pod fluctuates in CPU usage. The record count in workload A is set to 2,500,000. We monitor the actual CPU usage and recommendation values of the redis master pod in 900 seconds, which is long enough to complete the loading phase of workload A. Inspired by the three metrics in [10], the performance of the default vertical pod autoscaler is quantified by three metrics: average slack (m), percent of insufficient CPU observations (%), and average insufficient CPU (m). Slack is the amount of the resource that is overprovisioning. Insufficient CPU means the resource that is underprovisioning, namely, recommended values are lower than the actual CPU usage.

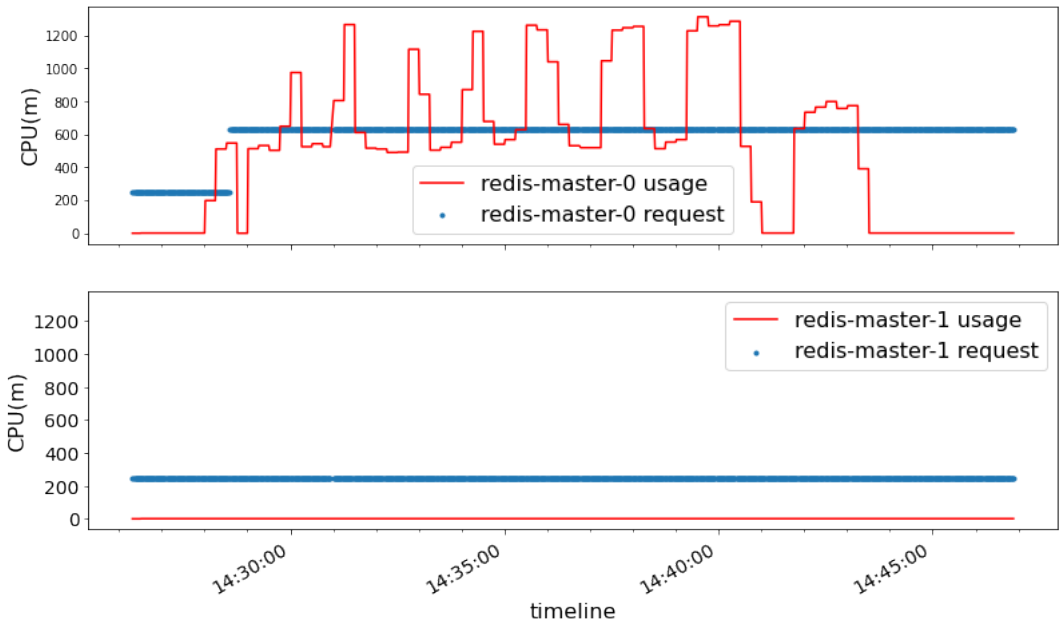


Figure 5.3: CPU usage and requests of redis-master node in the loading phase of workload A, record count is set to 2,500,000.

Recommenders	**Average slack** (m)	**Percent of insufficient CPU** (% observations)	**Average insufficient CPU** (m)
VPA	556.8	14.2	44.8

Table 5.1: The performance of vertical pod autoscaler in the loading phase of workload A.

Thus, the average slack in this thesis is defined as:

$$Average\_Slack = \frac{Total\_Slack}{Total\_Number\_of\_Observations}.$$

Percent of insufficient CPU observations follows:

$$Percent\_of\_Insufficient\_CPU = \frac{Number\_of\_Insufficient\_CPU\_Observations}{Total\_Number\_of\_Observations}.$$

Average insufficient CPU is defined as:

$$Average\_Insufficient\_CPU = \frac{Total\_Amount\_of\_Insufficient\_CPU}{Total\_Number\_of\_Observations}.$$

With the above metrics, we evaluate the performance of the current vertical pod autoscaling recommender in Kubernetes. Table 5.1 and Figure 5.4 present the performance of the current vertical pod autoscaling recommender. In spite of the VPA recommender has a low percentage of insufficient CPU, it obviously shows in the figure that the VPA recommender is not able to follow the trend of actual CPU usage very well at least in a short term, which contributes to too much slack between the actual CPU usage and recommendation. Furthermore, we can obtain the fact that the VPA recommender has

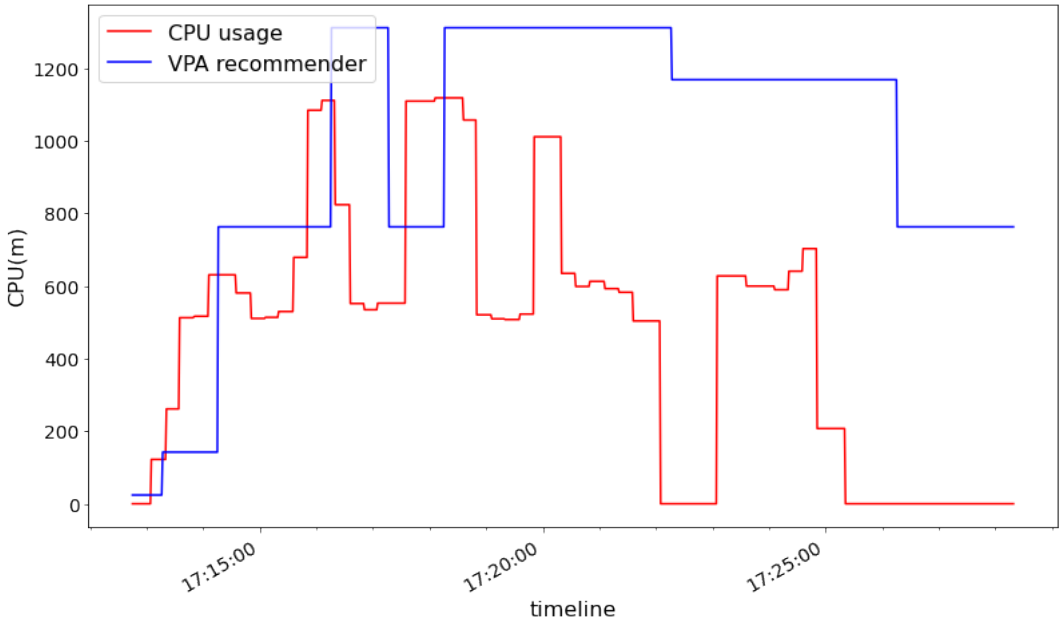


Figure 5.4: CPU usage of redis master pod and recommendations from vertical pod autoscaling recommender in the loading phase of workload A.

an obvious delay when the actual CPU usage increase suddenly. It may lead to the application has already been throttled before the container rescaling.

Conclusion-2: In the current vertical pod autoscaler, the pods that need to be updated still have to be evicted first and then be created. The pod that is applied on vertical pod autoscaling has to have at least two replications. Each pod can only correspond to one vertical pod autoscaler in one run. Moreover, the recommended value from the current vertical pod autoscaler can not follow the trend of CPU usage very well in a short-term workload and the recommended value from the current vertical pod autoscaler has an obvious delay in terms of the trend of CPU usage changes, resulting in a large slack and inefficiencies. It indicates there are still improvements to be done in vertical pod autoscaling.

5.3 SMA-based vs EMA-based recommender

In this section, we experiment three combinations of parameters for both SMA-based and EMA-based recommenders to verify the performance of these two recommenders. Two parameters are needed in both SMA-based and EMA-based recommenders, which are the size of the load tracker and the number of the load tracker. The size of the load tracker represents how many observations are in a load tracker to calculate the weighted or unweighted average. The number of the load tracker indicates how many representations l_i obtained from the load tracker function are used to make an extrapolation prediction. If the size of the load tracker and the number of it are two high values, it denotes that the recommender considers too many previous observations rather than recent observations, which will lead to trend delays in prediction. However, if the size of the load tracker and the number of it are two small values, it suggests that the recommender takes too many recent observations into account

rather than more previous ones. Namely, it usually indicates that the trend is very recent and is not fully smoothed in that situation. Both two situations mentioned above prevent a low prediction error.

Pods of the redis master node and YCSB are deployed on the minikube cluster and two pods are connected with each other. We use the loading phase of workload A in YCSB to make the redis master pod fluctuates in CPU usage. The record count in workload A is set to 2,500,000. We monitor the actual CPU usage and recommendation values of the redis master pod in 900 seconds, which is long enough to complete the loading phase of workload A. In addition, we set the VPA in "off" mode to ensure the smoothness of the experiments, which means the pod will not update automatically but still get recommendations. The recommenders are evaluated in three metrics, which are average slack (m), percent of insufficient CPU observations (%), and average insufficient CPU (m) as above-mentioned (see Section 5.2).

Recommenders	**Average slack** (m)	**Percent of insufficient CPU** (% observations)	**Average insufficient CPU** (m)
sma-3-2	340.9	30.4	79.7
ema-3-2	448.9	33.9	93.8
sma-5-3	53.0	43.3	76.7
ema-5-3	**51.8**	**34.4**	**42.5**
sma-10-5	71.9	34.7	87.6
ema-10-5	45.3	52.1	71.1

Table 5.2: The performance of different recommenders in the loading phase of workload A. The first number in the name of recommenders is the size of the load tracker and the second is the number of load trackers.

The performance of different recommenders in the loading phase of workload A is quantified in Table 5.2. The first number in the name of recommenders is the size of the load tracker and the second is the number of load trackers. For instance, ema-5-3 means EMA-based recommender with load tracker size as 5 and load tracker number as 3. In Table 5.2, it indicates ema-5-3 has the best performance among all recommenders, three metric values of it are relatively low. Figure 5.5 and 5.6 present the CPU usage of the redis master pod and recommended request values for that pod provided by recommender sma-5-3 and ema-5-3 correspondingly. Both two recommendations are able to follow the trend of actual CPU usage in general. However, compared with recommender sma-5-3, recommender ema-5-3 recommends the request values cover the actual CPU usage better despite the fact that there is some slack between ema-5-3 recommends and actual CPU usage.

To verify if the ema-5-3 recommender is able to perform in the same way on other workloads, we test the ema-5-3 recommender performance in the loading and running phase of workload A to D. Table 5.3 displays the performance of ema-5-3 recommender in the loading and running phase of workload A to F. Figure 5.7 and 5.8 present the CPU usage of the redis master pod and recommendations from ema-5-3 recommender in the loading and running phase of workload A to F. Recommender ema-5-3 achieves similar results in most workloads no matter loading and running phase, except for

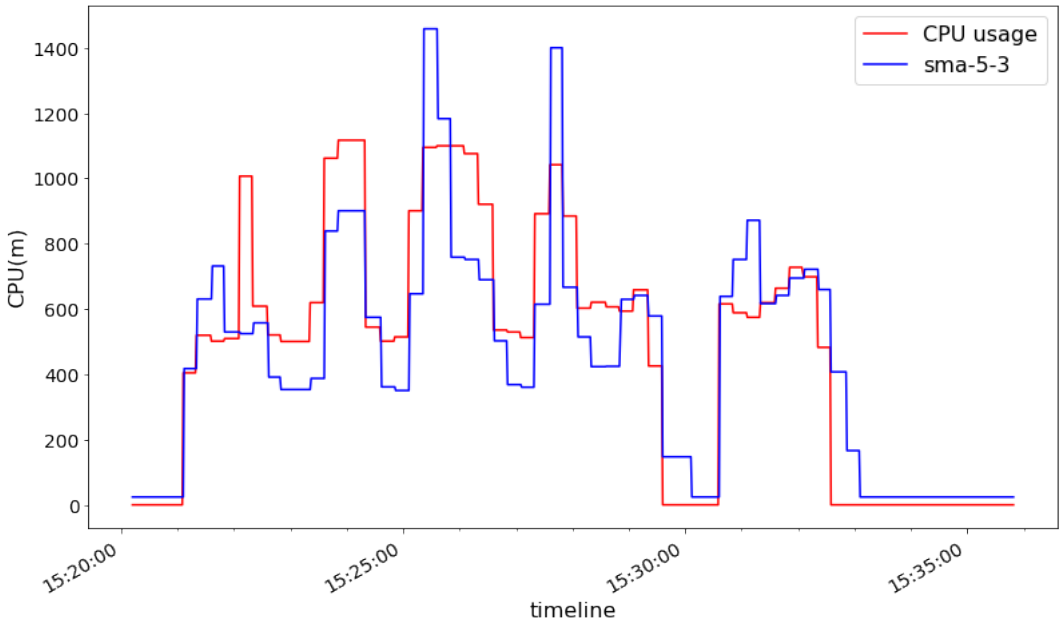


Figure 5.5: CPU usage of redis master pod and recommendations from recommender sma-5-3 in the loading phase of workload A.

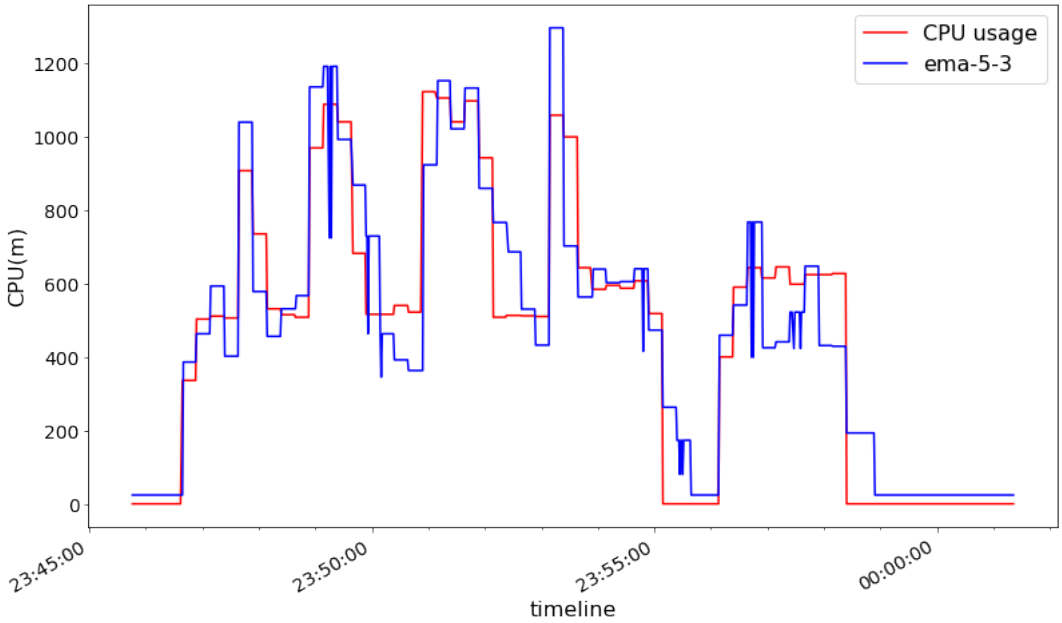


Figure 5.6: CPU usage of redis master pod and recommendations from recommender ema-5-3 in the loading phase of workload A.

running phase of workload E. In the workload E running phase, the CPU usage behaviors a flat but slightly fluctuating trend. Due to the characteristics of our extrapolation algorithm, the percent of insufficient CPU is high, at 48.7% in the workload E running phase. In this situation, our algorithm does not prevent the recommendations from dropping below the actual CPU usage very successfully but due to the "guarantee mechanism" in our design, the predicted value has not dropped too exaggeratedly. Thus, we can choose a larger multiplier for the average of recent CPU usage values to obtain a better result. The performance is displayed in Figure 5.9 when the multiplier is set to 2, we get 84.1m average slack, 26.8% insufficient CPU, and 12.6m average insufficient CPU.

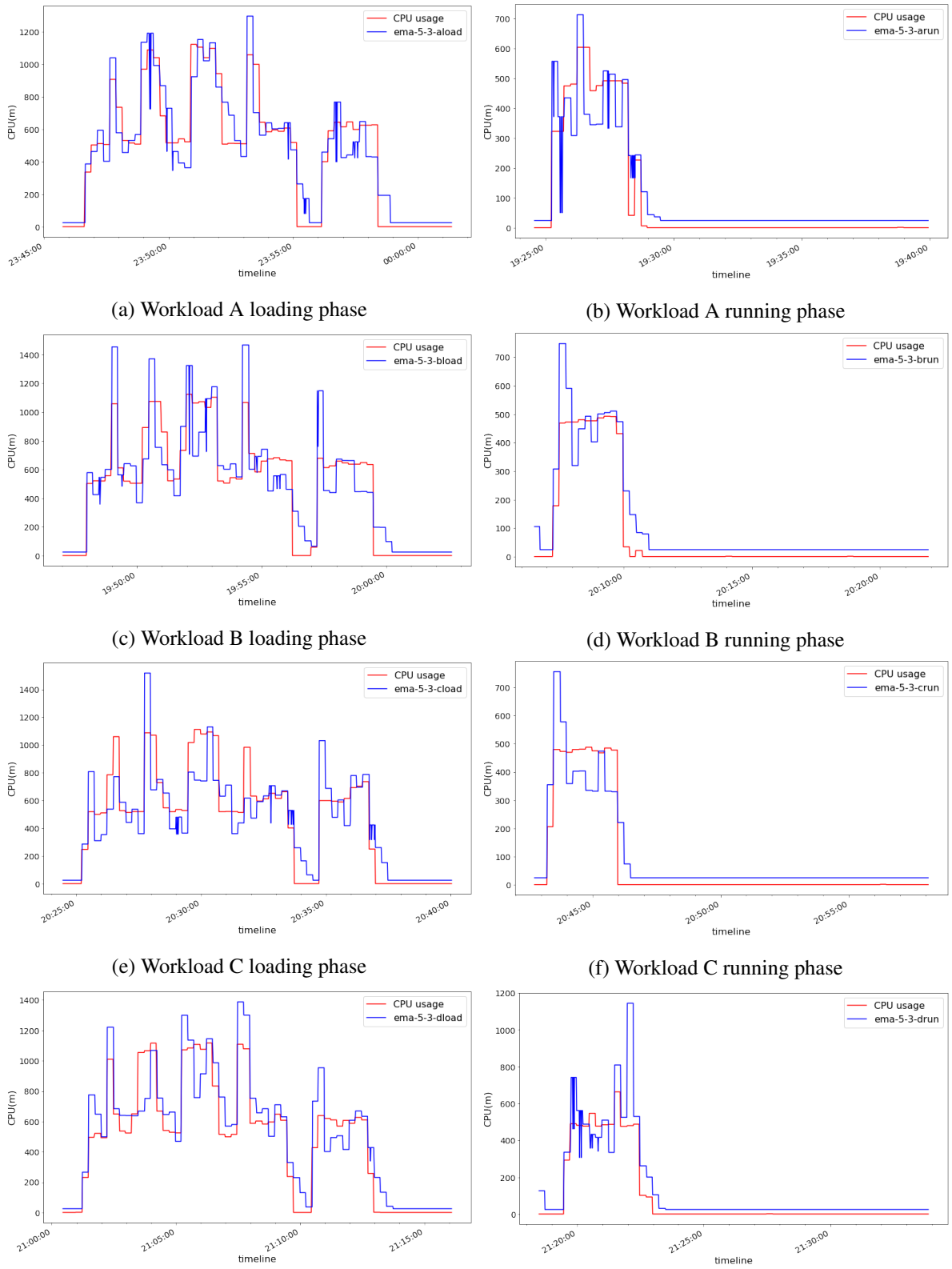
Conclusion-3: Both the recommended value from ema and sma recommenders can follow the trend of actual CPU usage closely in the loading phase of workload A when the size of the load tracker is set to 5 and the number of the load tracker is set to 3. The ema-5-3 has the best performance in the loading phase of workload A. In other workloads, ema-5-3 performs similarly except in the running phase of workload E. But when we enlarge the multiplier in our methods, it can get a better result in the running phase of workload E.

Workloads	**Average slack** (m)	**Percent of insufficient CPU** (% observations)	**Average insufficient CPU** (m)
A-loading	51.8	34.4	42.5
A-running	30.4	10.7	15.5
B-loading	71.5	31.2	57.7
B-running	37.4	5.1	4.4
C-loading	58.0	35.4	67.6
C-running	32.3	11.5	14.0
D-loading	81.7	19.0	32.8
D-running	46.9	5.5	6.2
E-loading	57.1	42.4	70.0
E-running	23.2	48.7	54.3
F-loading	49.5	41.6	78.2
F-running	33.0	12.8	12.1

Table 5.3: The performance of ema-5-3 recommender in the loading and running phase of workload A to F.

5.4 Comparison with HW and LSTM Recommender

In this section, we compare the performance of HW and LSTM recommenders with our recommender ema-5-3. Due to the computational complexity of HW and LSTM algorithms, they are not suitable to support vertical pod autoscaling per second because the models can not be trained in one second. For HW recommender, the prediction model refits when each new observation is collected. For LSTM recommender, the prediction model retrains every season. Thomas Wang et al. apply the HW and



(a) Workload A loading phase

(b) Workload A running phase

(c) Workload B loading phase

(d) Workload B running phase

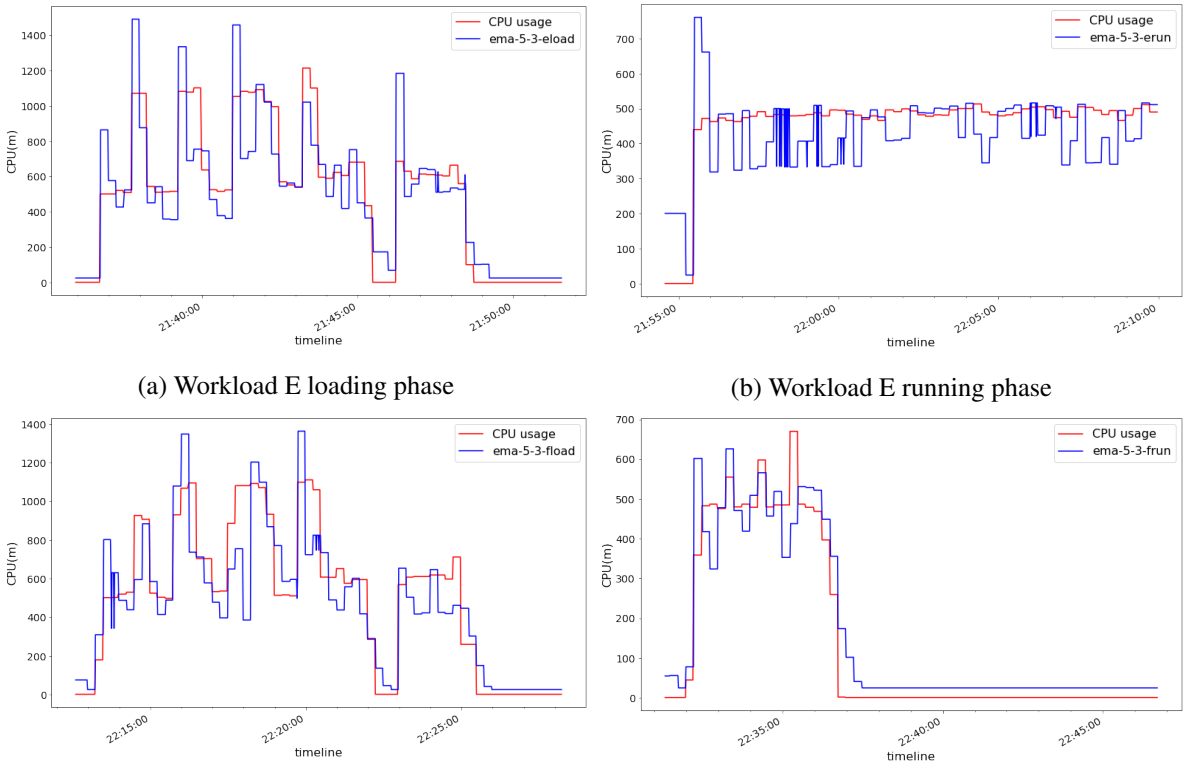
(e) Workload C loading phase

(f) Workload C running phase

(g) Workload D loading phase

(h) Workload D running phase

Figure 5.7: CPU usage of redis master pod and recommendations from ema-5-3 recommender in the loading and running phase of workload A to D.



(a) Workload E loading phase

(b) Workload E running phase

(c) Workload F loading phase

(d) Workload F running phase

Figure 5.8: CPU usage of redis master pod and recommendations from ema-5-3 recommender in the loading and running phase of workload E and F.

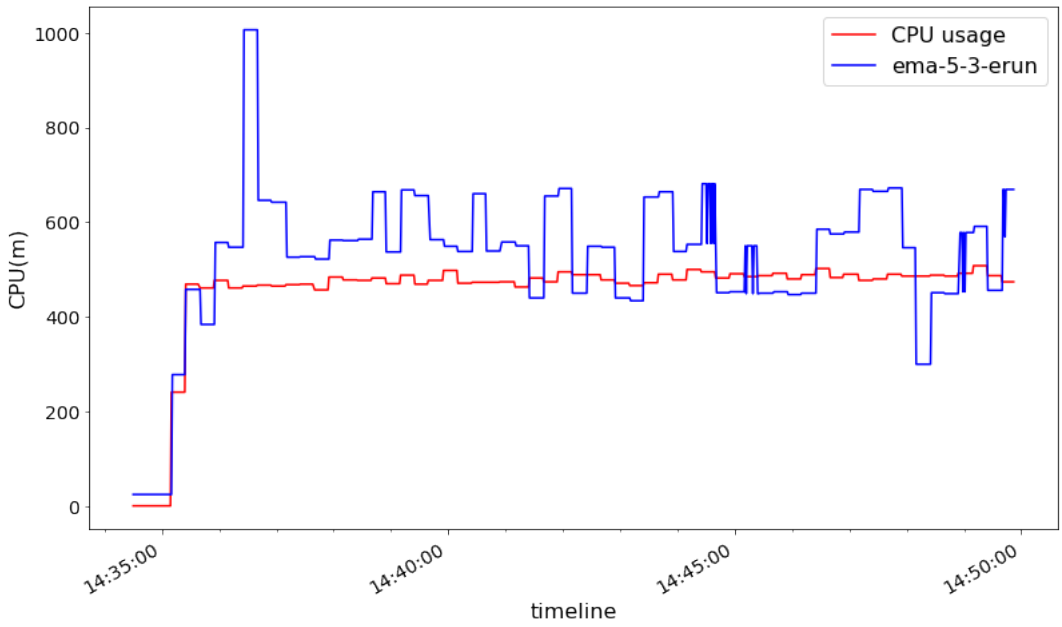


Figure 5.9: CPU usage of redis master pod and recommendations from recommender sma-5-3 in the running phase of workload E, multiplier in algorithm is set to 2.

LSTM algorithms in their paper [10] to estimate the CPU usage every 10 minutes. So there are 144 samples per day. Samples in one day compose a season. Furthermore, at least two season data are needed to generate the future CPU usage prediction from HW and LSTM recommenders because these two models need to be initialized at the beginning. That explains why there is a fixed recommendation value in the initial stage for both HW and LSTM recommenders.

In this experiment, we apply these two recommenders out of the vertical pod autoscaler and use the same CPU usage as in the ema-5-3 run for a more equitable performance comparison between recommender ema-5-3 and them. We modify the season length to one minute and these two recommenders present the preset value in the first two seasons due the models initialization. From Table 5.4 and Figure 5.10, we can obtain the fact that HW recommender is not capable to predict the CPU usage trend very well. The average slack of HW recommender is pretty high, ten times more than the slack of recommender ema-5-3. To have a clearer demonstration of the comparison between LSTM recommender and ema-5-3, we remove the prediction from HW recommender and show the comparison again in Figure 5.11. As seen in Figure 5.11, the prediction value from LSTM recommender has more slack than the value predicted by ema-5-3. Moreover, compared with LSTM recommender, the prediction from ema-5-3 follows the CPU usage trend much more closely and the prediction from LSTM recommender displays delays in terms of changes in the CPU usage trend.

Conclusion-4: HW recommender can not perform well in the loading phase of workload A due to its random seasonality. Compared with LSTM recommender, the prediction from ema-5-3 follows the CPU usage trend much more closely, namely, with less slack and without delays in terms of changes in the CPU usage trend. Moreover, our method has much lower computational complexity than HW and LSTM methods.

Recommenders	**Average slack** (m)	**Percent of insufficient CPU** (% observations)	**Average insufficient CPU** (m)
ema-5-3	**51.8**	**34.4**	**42.5**
LSTM	269.3	22.1	40.6
HW	588.5	26.3	111.4

Table 5.4: Compared with the performance of HW and LSTM recommenders in the loading phase of workload A.

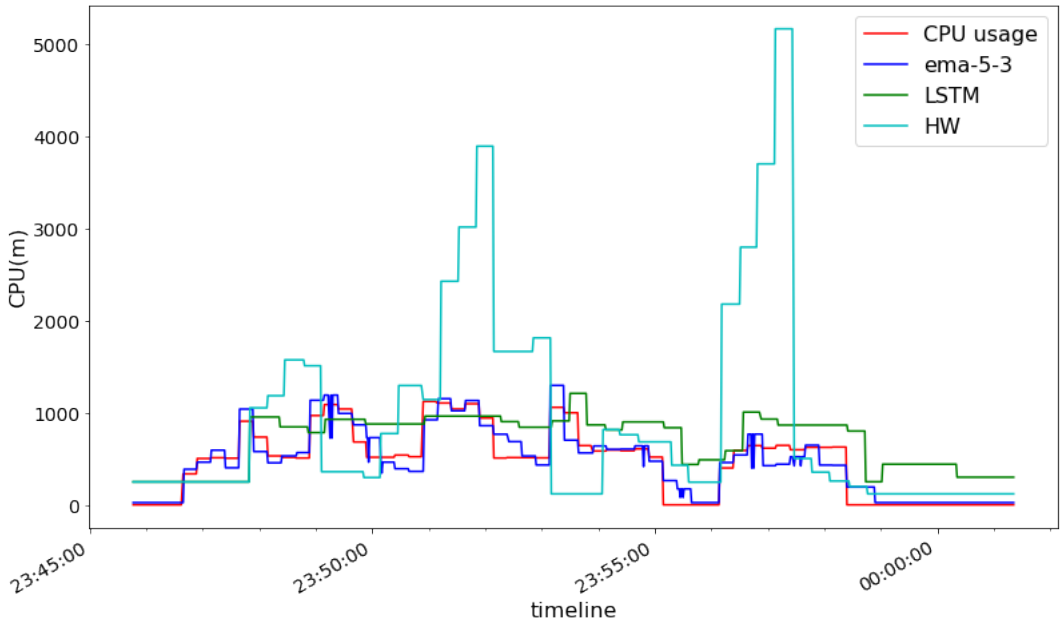


Figure 5.10: Performance comparison of different recommenders in the loading phase of workload A.

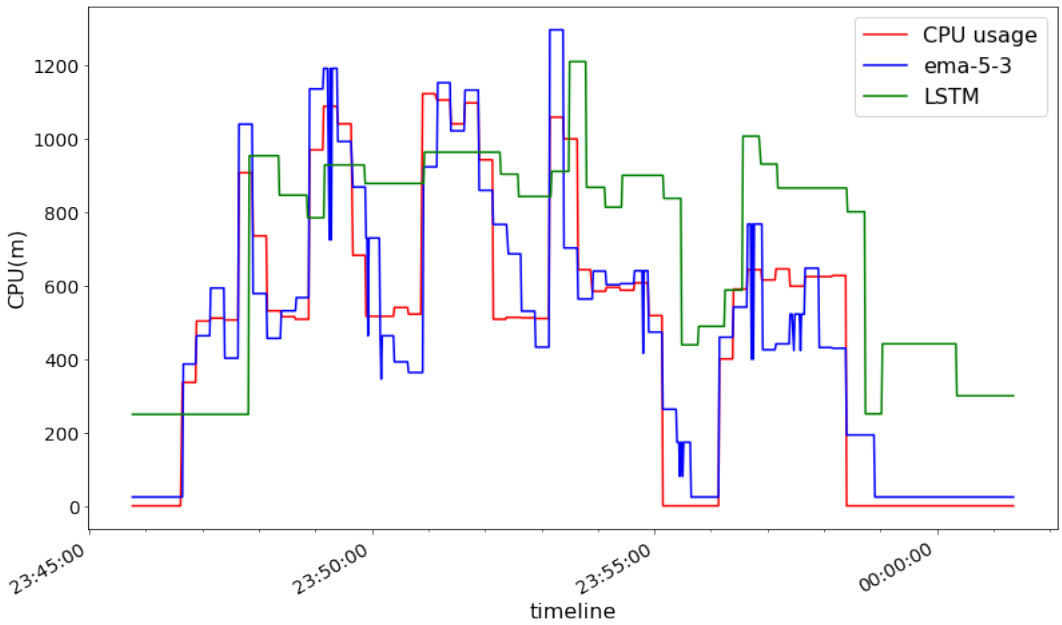


Figure 5.11: Performance comparison of different recommenders (without HW recommender) in the loading phase of workload A.

Chapter 6

Related Work

In this chapter, we mainly introduce some works related to our thesis. We introduce them in four aspects: prediction of workloads, autoscaling in the cloud, autoscaling containers, and vertical autoscaling virtual machines.

6.1 Predicting trend of workloads

Prediction algorithm for the future resource demands plays an essential role in container autoscaling in this thesis. Thus, we describe several related works in the aspect of predicting the trend of workloads in this section. As the comparison of our prediction methods, a recent work [10] about the CPU usage prediction for autoscaling is based on Holt-Winters exponential smoothing (HW) and Long Short-Term Memory (LSTM) methods. These two methods have more expensive computational complexity than our methods. We have analyzed them in detail in the previous chapter. Our prediction methods are based on the two-step CPU usage prediction model from [9]. This CPU usage prediction model is introduced in detail in the previous chapter of this thesis as well. Similar to this model, Sara Casolari and Mauro Andreolini [38] proposed another trend-aware regression model. The idea of the trend-aware regression model is very similar to that of the two-step prediction model from [9] because these two models are proposed by the same authors. Both of these two models mainly use linear extrapolation to predict future CPU usage but with some differences in the representation of recent CPU usage values.

Apart from these, I.J. Davis et al. [39] proposed a regression-based hybrid method which composed of Fourier analysis, autocorrelation, multivariate linear regression. It chooses the best prediction strategy from these methods according to the recent prediction accuracy. Nilabja Roy et al. [40] used an autoregressive moving average method to predict the trend of workloads.

6.2 Autoscaling in the cloud

Some works specifically investigate the performance study of the state-of-art autoscalers. Therefore, we firstly involve several works related to the autoscalers performance study in this section. For

instance, Laurens Versluis et al. [41] analyzed the impact of several factors on the performance of autoscalers. These factors include the application domain, sudden peaks, allocation policy, and diversity of datacenter environments. Alexey Ilyushkin et al. [42] demonstrated a detailed performance comparison of seven state-of-the-art autoscaling strategies. Anshul Jindal et al. [43] developed a tool for the performance measurement of autoscaling.

While the other works research on the autoscaling strategy itself. For instance, Tao Chen et al. [44] investigated the self-aware and self-adaptive cloud autoscaling system and gave a complete taxonomy in this field. They listed a lot of autoscaling models in their paper. Zhiheng Zhong and Rajkumar Buyya [45] proposed a task allocation strategy to optimize resource utilization from three aspects. One of them is to change the cluster size to increase the utilization through autoscaling strategies. Brandon Thurgood and Ruth G. Lennon [46] offered a software solution to autoscale the Kubernetes cluster itself. Hamoun Ghanbari et al. [47] provided a solution for autoscaling mechanisms from an optimization perspective. They proposed an autoscaling strategy that regards the problem as a model predictive control problem, minimize the cost of resource usage while meeting with the application provider's requirements.

6.3 Autoscaling containers

In this thesis, we only focus on the autoscaling mechanism at the container level. Our work in this thesis only provides a solution for the vertical autoscaling on CPU usage. The same is to address the containers rightsizing problem in Kubernetes, Gourav Rattihalli et al. [48] designed an autoscaling mechanism called RUBAS to estimate the CPU and memory resources of containers through the sum of the median of observations and the absolute deviation of observations. RUBAS can reschedule the containers based on the above-mentioned estimation method. They also showed the effectiveness of their estimation method in their previous work [49, 50].

However, some works focus on the horizontal autoscaling as well. Horizontal autoscaling is an autoscaling mechanism to adjust the number of concurrent running containers automatically. Autopilot [26] from Google is a complete autoscaling system, it not only takes into account vertical autoscaling but also horizontal autoscaling on both CPU and memory usage. The current vertical pod autoscaling recommender is directly inspired by the moving window recommenders in Autopilot. Thanh-Tung Nguyen et al. [51] investigated the horizontal pod autoscaling and gave suggestions on optimizing the performance of horizontal pod autoscaling.

6.4 Vertical autoscaling of VMs

Different from focusing on the vertical autoscaling of the containers like what we do in this thesis, some works focus on the vertical autoscaling of virtual machines. For instance, Abdul R. Hummaida et al. [52] investigated the veritical autoscaling of virtual machines and they defined it as cloud systems adaptation. Mina Sedaghat et al. [4] proposed a vertical autoscaling mechanism for virtual machines, which considers the cost and benefit trade-off of replacing the virtual machines.

Chapter 7

Conclusions

In this thesis, we have uncovered the autoscaling mechanism of the vertical pod autoscaler component in Kubernetes. Based on the fundamental of the current vertical pod autoscaler, we design an autoscaling mechanism that is sensitive to the CPU usage changes. Our proposed autoscaling mechanism is able to follow the container CPU usage trend very closely. Due to its inexpensive computational complexity, it is suitable to be applied in actual scenarios. Moreover, our autoscaling mechanism outperforms the current vertical pod autoscaling mechanism and autoscaling mechanism based on HW and LSTM in short-term workloads, where the CPU usage is highly variable.

7.1 Answers to the Research Questions

- How does the current vertical pod autosclaer work? What problems does the current vertical pod autoscaler have?

  The detailed mechanism of the current vertical pod autoscaler is included in Section 3.1. Currently, vertical pod autoscaler can not solve the problem of recommendations for sudden CPU usage increases very well. The recommendations from the current vertical pod autoscaler usually have lots of slack, which means it results in resource under-utilization. On the contrary, vertical pod autoscaler is able to provide a reliable recommendation in a long-term workload.

- How can we design an autoscaling policy which enables good performance for the application and minimizes resource waste?

  Would rather be more than less is the essential idea of autoscaling. Because insufficient resources will lead to the termination or throttle of the application, which is more serious than the resource under-utilization. In the current vertical autoscaler, it is reflected that the lower bound converges much more rapidly than the upper bound. In the autoscaling mechanism based on HW and LSTM, it is reflected in the extra rescaling buffer for some error rooms. As for our design, it is reflected in two following aspects. First, due to the characteristic of extrapolation prediction, we adopt a ”bottoming” mechanism for the future CPU demands prediction. The final prediction value is the bigger one between the several times the weighted or unweighted average of recent CPU usage and the extrapolation prediction value.

- Compared to the current vertical pod autoscaler and autoscaling mechanism based on HW and LSTM, what are the advantages of our autoscaling design?

  Our autoscaling mechanism has the following advantages. First, it fits the CPU usage trend well and almost has no delay to the trend change. Second, due to its inexpensive computational complexity, it is easier to be applied in actual scenarios. Third, it is integrated into the vertical pod autoscaler component which is plug and play. Moreover, unlike the autoscaling mechanism based on HW and LSTM, our mechanism does not need any time to ”warm-up”, namely, train the model in the initial stage.

Appendix A

Environment Setup

A.1 Minikube Setup

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
minikube start --cpus 8 --memory 8192
minikube addons enable metrics-server
```

A.2 YCSB Setup

```
vim ycsb.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: ycsb
  labels:
    app: test
spec:
  containers:
    - name: ycsb
      image: 0track/ycsb:latest
      imagePullPolicy: Always
      ports:
        - name: ycsb
          containerPort: 80
          protocol: TCP
```

```
kubectl apply -f ycsb.yaml
kubectl exec -it ycsb -- sh
apt-get update
apt-get install vim
apt-get install maven
cd YCSB
mvn -pl com.yahoo.ycsb:redis-binding -am clean package
exit
```

A.3 Redis Setup

```
vim redis-statefulset.yaml
```

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis-master
spec:
  serviceName: redis
  replicas: 2
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis-master
        image: redis:6.0-alpine
        ports:
        - containerPort: 6379
          name: redis-master
---
apiVersion: v1
kind: Service
metadata:
  name: redis-service
spec:
  clusterIP: None
  ports:
    - port: 6379
      targetPort: 6379
  selector:
      app: redis
```

```
vim redis-worker.yaml
```

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis-worker
spec:
  serviceName: redis
  replicas: 2
  selector:
    matchLabels:
```

```
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis-worker
        image: redis:6.0-alpine
        ports:
        - containerPort: 6379
          name: redis-worker

kubectl exec -it redis-worker -- sh
redis-cli
slaveof 172.17.0.4 6379
# 172.17.0.4 is the ip of redis-master
```

A.4 MongoDB Setup

```
wget https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-ubuntu2004-4.4.5.tgz
tar -zxvf mongodb-linux-x86_64-ubuntu2004-4.4.5.tgz
sudo mv mongodb-linux-x86_64-ubuntu2004-4.4.5 /usr/local/mongodb4
export PATH=/usr/local/mongodb4/bin:$PATH
sudo mkdir -p /var/lib/mongo
sudo mkdir -p /var/log/mongodb
sudo chown `whoami` /var/lib/mongo
sudo chown `whoami` /var/log/mongodb
mongod --dbpath /var/lib/mongo --logpath /var/log/mongodb/mongod.log --fork
```

A.5 Vertical Pod Autoscaler Setup

```
git clone https://github.com/kubernetes/autoscaler.git
cd autoscaler/vertical-pod-autoscaler
./hack/vpa-down.sh
./hack/vpa-up.sh

vim redis-vpa.yaml

apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: redis-vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:       StatefulSet
    name:       redis-master
```

```
  updatePolicy:
    updateMode: "Auto"
```

A.6 Docker Image

```
docker build -t vpa-recommender:lastest .
docker tag vpa-recommender:lastest username/vpa-recommender:lastest
```

References

[1] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. “Elasticity in Cloud Computing: What It Is, and What It Is Not”. In: *10th International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX Association, June 2013, pp. 23–27. ISBN: 978-1-931971-02-7. URL: https://www.usenix.org/conference/icac13/technical-sessions/presentation/herbst.

[2] Keith Adams and Ole Agesen. “A Comparison of Software and Hardware Techniques for X86 Virtualization”. In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XII. San Jose, California, USA: Association for Computing Machinery, 2006, pp. 2–13. ISBN: 1595934510. DOI: 10.1145/1168857.1168860. URL: https://doi.org/10.1145/1168857.1168860.

[3] IBM Cloud Education. *IBM Cloud Learn Hub/What is Virtualization?/Virtualization*. Accessed: 2021-05-01. URL: https://www.ibm.com/cloud/learn/virtualization-a-complete-guide.

[4] Mina Sedaghat, Francisco Hernandez-Rodriguez, and Erik Elmroth. “A Virtual Machine Re-Packing Approach to the Horizontal vs. Vertical Elasticity Trade-off for Cloud Autoscaling”. In: *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*. CAC ’13. Miami, Florida, USA: Association for Computing Machinery, 2013. ISBN: 9781450321723. DOI: 10.1145/2494621.2494628. URL: https://doi.org/10.1145/2494621.2494628.

[5] Claus Pahl et al. “Cloud Container Technologies: A State-of-the-Art Review”. In: *IEEE Transactions on Cloud Computing* PP (May 2017), pp. 1–1. DOI: 10.1109/TCC.2017.2702586.

[6] IBM Cloud Education. *IBM Cloud Learn Hub/Containerization Explained/Containerization*. Accessed: 2021-05-01. URL: https://www.ibm.com/cloud/learn/containerization.

[7] Brendan Burns et al. “Borg, Omega, and Kubernetes”. In: *ACM Queue* 14 (2016), pp. 70–93. URL: http://queue.acm.org/detail.cfm?id=2898444.

[8] K. Grygiel and M. Wielgus. *Kubernetes Vertical Pod Autoscaler: Design Proposal*. Accessed: 2021-04-28. URL: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/autoscaling/vertical-pod-autoscaler.md.

[9] Mauro Andreolini and Sara Casolari. "Load Prediction Models in Web-Based Systems". In: *Proceedings of the 1st International Conference on Performance Evaluation Methodolgies and Tools*. valuetools '06. Pisa, Italy: Association for Computing Machinery, 2006, 27–es. ISBN: 1595935045. DOI: 10.1145/1190095.1190129. URL: https://doi.org/10.1145/1190095.1190129.

[10] Thomas Wang, Simone Ferlin, and Marco Chiesa. "Predicting CPU Usage for Proactive Autoscaling". In: *Proceedings of the 1st Workshop on Machine Learning and Systems*. EuroMLSys '21. Online, United Kingdom: Association for Computing Machinery, 2021, pp. 31–38. ISBN: 9781450382984. DOI: 10.1145/3437984.3458831. URL: https://doi.org/10.1145/3437984.3458831.

[11] Rong Pan. "Holt–Winters Exponential Smoothing". In: *Wiley Encyclopedia of Operations Research and Management Science*. American Cancer Society, 2011. ISBN: 9780470400531. DOI: https://doi.org/10.1002/9780470400531.eorms0385. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470400531.eorms0385. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470400531.eorms0385.

[12] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[13] Brian F. Cooper et al. "Benchmarking Cloud Serving Systems with YCSB". In: SoCC '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 143–154. ISBN: 9781450300360. DOI: 10.1145/1807128.1807152. URL: https://doi.org/10.1145/1807128.1807152.

[14] Red Hat. *CHAPTER 1. INTRODUCTION TO CONTROL GROUPS (CGROUPS)*. Accessed: 2021-05-02. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01.

[15] Christoph Lameter Paul Jackson. *CGROUPS*. Accessed: 2021-07-10. URL: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[16] Roberto Morabito, Jimmy Kjällman, and Miika Komu. "Hypervisors vs. Lightweight Virtualization: A Performance Comparison". In: *2015 IEEE International Conference on Cloud Engineering*. 2015, pp. 386–393. DOI: 10.1109/IC2E.2015.74.

[17] Thanh Bui. *Analysis of Docker Security*. 2015. arXiv: 1501.02967 [cs.CR].

[18] Docker. *Docker Overview*. Accessed: 2021-05-03. URL: https://docs.docker.com/get-started/overview/.

[19] 2021 The Kubernetes Authors. *What is Kubernetes?* Accessed: 2021-05-03. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[20] 2021 The Kubernetes Authors. *Minikube Documentation*. Accessed: 2021-05-04. URL: https://minikube.sigs.k8s.io/docs/.

[21] Redislabs. *Introduction to Redis*. Accessed: 2021-07-10. URL: https://redis.io/topics/introduction.

[22] MongoDB Inc. *Introduction to MongoDB*. Accessed: 2021-07-10. URL: https://docs.mongodb.com/manual/introduction/.

[23] Abhishek Verma et al. "Large-scale cluster management at Google with Borg". In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015.

[24] 2021 The Kubernetes Authors. *Kubernetes Components*. Accessed: 2021-07-10. URL: https://kubernetes.io/docs/concepts/overview/components/.

[25] Redislabs. *Replication*. Accessed: 2021-05-03. URL: https://redis.io/topics/replication.

[26] Krzysztof Rzadca et al. "Autopilot: Workload Autoscaling at Google Scale". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020. URL: https://dl.acm.org/doi/10.1145/3342195.3387524.

[27] 2021 The Linux Foundation. *What is Prometheus?* Accessed: 2021-07-12. URL: https://prometheus.io/docs/introduction/overview/.

[28] 2021 The Kubernetes Authors. *Assign CPU Resources to Containers and Pods*. Accessed: 2021-05-05. URL: https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-resource/.

[29] Charles C. Holt. "Forecasting seasonals and trends by exponentially weighted moving averages". In: *International Journal of Forecasting* 20.1 (2004), pp. 5–10. ISSN: 0169-2070. DOI: https://doi.org/10.1016/j.ijforecast.2003.09.015. URL: https://www.sciencedirect.com/science/article/pii/S0169207003001134.

[30] Toader Sebastian. *Vertical Pod Autoscaler*. Accessed: 2021-07-16. URL: https://banzaicloud.com/blog/k8s-vertical-pod-autoscaler/.

[31] 2021 The Kubernetes Authors. *Kubernetes Monitoring Architecture*. Accessed: 2021-05-05. URL: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/instrumentation/monitoring_architecture.md.

[32] Google. *cAdvisor*. Accessed: 2021-05-05. URL: https://github.com/google/cadvisor.

[33] 2021 The Kubernetes Authors. *Kubelet*. Accessed: 2021-07-10. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/.

[34] kubernetes-sigs. *Kubernetes Metrics Server*. Accessed: 2021-05-05. URL: https://github.com/kubernetes-sigs/metrics-server.

[35] 2021 The Kubernetes Authors. *Metrics-API*. Accessed: 2021-05-05. URL: https://github.com/kubernetes/metrics.

[36] 2021 The Kubernetes Authors. *Assign Memory Resources to Containers and Pods*. Accessed: 2021-05-05. URL: https://kubernetes.io/docs/tasks/configure-pod-container/assign-memory-resource/.

[37] Vinay Kulkarni. *In-Place Update of Pod Resources*. Accessed: 2021-07-16. URL: https://github.com/kubernetes/enhancements/issues/1287.

[38] Sara Casolari, Mauro Andreolini, and Michele Colajanni. "Runtime prediction models for Web-based system resources". In: *2008 IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems*. 2008, pp. 1–8. DOI: 10.1109/MASCOT.2008.4770556.

[39] I. J. Davis et al. "Regression-Based Utilization Prediction Algorithms: An Empirical Investigation". In: *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*. CASCON '13. Ontario, Canada: IBM Corp., 2013, pp. 106–120.

[40] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. "Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting". In: *2011 IEEE 4th International Conference on Cloud Computing*. 2011, pp. 500–507. DOI: 10.1109/CLOUD.2011.42.

[41] Laurens Versluis, Mihai Neacşu, and Alexandru Iosup. "A Trace-Based Performance Study of Autoscaling Workloads of Workflows in Datacenters". In: *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. CCGrid '18. Washington, District of Columbia: IEEE Press, 2018, pp. 223–232. ISBN: 9781538658154. DOI: 10.1109/CCGRID.2018.00037. URL: https://doi.org/10.1109/CCGRID.2018.00037.

[42] Alexey Ilyushkin et al. "An Experimental Performance Evaluation of Autoscalers for Complex Workflows". In: *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3.2 (Apr. 2018). ISSN: 2376-3639. DOI: 10.1145/3164537. URL: https://doi.org/10.1145/3164537.

[43] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. "Autoscaling Performance Measurement Tool". In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE '18. Berlin, Germany: Association for Computing Machinery, 2018, pp. 91–92. ISBN: 9781450356299. DOI: 10.1145/3185768.3186293. URL: https://doi.org/10.1145/3185768.3186293.

[44] Tao Chen, Rami Bahsoon, and Xin Yao. "A Survey and Taxonomy of Self-Aware and Self-Adaptive Cloud Autoscaling Systems". In: *ACM Comput. Surv.* 51.3 (June 2018). ISSN: 0360-0300. DOI: 10.1145/3190507. URL: https://doi.org/10.1145/3190507.

[45] Zhiheng Zhong and Rajkumar Buyya. "A Cost-Efficient Container Orchestration Strategy in Kubernetes-Based Cloud Computing Infrastructures with Heterogeneous Resources". In: *ACM Trans. Internet Technol.* 20.2 (Apr. 2020). ISSN: 1533-5399. DOI: 10.1145/3378447. URL: https://doi.org/10.1145/3378447.

[46] Brandon Thurgood and Ruth G. Lennon. "Cloud Computing With Kubernetes Cluster Elastic Scaling". In: *Proceedings of the 3rd International Conference on Future Networks and Distributed Systems*. ICFNDS '19. Paris, France: Association for Computing Machinery, 2019. ISBN: 9781450371636. DOI: 10.1145/3341325.3341995. URL: https://doi.org/10.1145/3341325.3341995.

[47] Hamoun Ghanbari et al. “Optimal Autoscaling in a IaaS Cloud”. In: *Proceedings of the 9th International Conference on Autonomic Computing*. ICAC ’12. San Jose, California, USA: Association for Computing Machinery, 2012, pp. 173–178. ISBN: 9781450315203. DOI: 10.1145/2371536.2371567. URL: https://doi.org/10.1145/2371536.2371567.

[48] Gourav Rattihalli et al. “Exploring Potential for Non-Disruptive Vertical Auto Scaling and Resource Estimation in Kubernetes”. In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 2019, pp. 33–40. DOI: 10.1109/CLOUD.2019.00018.

[49] Gourav Rattihalli et al. “Two stage cluster for resource optimization with Apache Mesos”. In: *CoRR* abs/1905.09166 (2019). arXiv: 1905.09166. URL: http://arxiv.org/abs/1905.09166.

[50] Gourav Rattihalli. “Exploring Potential for Resource Request Right-Sizing via Estimation and Container Migration in Apache Mesos”. In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. 2018, pp. 59–64. DOI: 10.1109/UCC-Companion.2018.00035.

[51] Thanh-Tung Nguyen et al. “Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration”. In: *Sensors* 20.16 (2020). ISSN: 1424-8220. DOI: 10.3390/s20164621. URL: https://www.mdpi.com/1424-8220/20/16/4621.

[52] Abdul R. Hummaida, Norman W. Paton, and Rizos Sakellariou. “Adaptation in Cloud Resource Configuration: A Survey”. In: *J. Cloud Comput.* 5.1 (Dec. 2016). ISSN: 2192-113X. DOI: 10.1186/s13677-016-0057-9. URL: https://doi.org/10.1186/s13677-016-0057-9.