DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2021

Predictive vertical CPU autoscaling in Kubernetes based on time-series forecasting with Holt-Winters exponential smoothing and long short-term memory

THOMAS WANG

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Predictive vertical CPU autoscaling in Kubernetes based on time-series forecasting with Holt-Winters exponential smoothing and long short-term memory

THOMAS WANG

Degree Programme in Computer Science and Engineering
Date: April 16, 2021
Supervisor: Marco Chiesa
Industrial supervisor: Simone Ferlin
Examiner: Cyrille Artho
School of Electrical Engineering and Computer Science
Host company: Ericsson AB
Swedish title: Prediktiv vertikal CPU-autoskalning i Kubernetes baserat på tidsserieprediktion med Holt-Winters exponentiell utjämning och långt korttidsminne

Abstract

Private and public clouds require users to specify requests for resources such as CPU and memory (RAM) to be provisioned for their applications. The values of these requests do not necessarily relate to the application's run-time requirements, but only help the cloud infrastructure resource manager to map requested virtual resources to physical resources. If an application exceeds these values, it might be throttled or even terminated. Consequently, requested values are often overestimated, resulting in poor resource utilization in the cloud infrastructure. Autoscaling is a technique used to overcome these problems.

In this research, we formulated two new predictive CPU autoscaling strategies for Kubernetes containerized applications, using time-series analysis, based on Holt-Winters exponential smoothing and long short-term memory (LSTM) artificial recurrent neural networks. The two approaches were analyzed, and their performances were compared to that of the default Kubernetes Vertical Pod Autoscaler (VPA). Efficiency was evaluated in terms of CPU resource wastage, and insufficient CPU percentage and amount for container workloads from Alibaba Cluster Trace 2018, and others.

In our experiments, we observed that Kubernetes Vertical Pod Autoscaler (VPA) tended to perform poorly on workloads that periodically change. Our results showed that compared to VPA, predictive methods based on Holt-Winters exponential smoothing (HW) and Long Short-Term Memory (LSTM) can decrease CPU wastage by over 40% while avoiding CPU insufficiency for various CPU workloads. Furthermore, LSTM has been shown to generate stabler predictions compared to that of HW, which allowed for more robust scaling decisions.

Keywords

Kubernetes, Docker, Container, Cloud Native, Cloud Computing, Resource Provisioning, Autoscaling, Predictive scaling, CPU Usage, Seasonality, Exponential Smoothing, Long short-term memory, Time-series Analysis

Sammanfattning

Privata och offentliga moln kräver att användare begär mängden CPU och minne (RAM) som ska fördelas till sina applikationer. Mängden resurser är inte nödvändigtvis relaterat till applikationernas körtidskrav, utan är till för att molninfrastrukturresurshanteraren ska kunna kartlägga begärda virtuella resurser till fysiska resurser. Om en applikation överskrider dessa värden kan den saktas ner eller till och med krascha. För att undvika störningar överskattas begärda värden oftast, vilket kan resultera i ineffektiv resursutnyttjande i molninfrastrukturen. Autoskalning är en teknik som används för att överkomma dessa problem.

I denna forskning formulerade vi två nya prediktiva CPU autoskalningsstrategier för containeriserade applikationer i Kubernetes, med hjälp av tidsserieanalys baserad på metoderna Holt-Winters exponentiell utjämning och långt korttidsminne (LSTM) återkommande neurala nätverk. De två metoderna analyserades, och deras prestationer jämfördes med Kubernetes Vertical Pod Autoscaler (VPA). Prestation utvärderades genom att observera under- och överutilisering av CPU-resurser, för diverse containerarbetsbelastningar från bl. a. Alibaba Cluster Trace 2018.

Vi observerade att Kubernetes Vertical Pod Autoscaler (VPA) i våra experiment tenderade att prestera dåligt på arbetsbelastningar som förändras periodvist. Våra resultat visar att jämfört med VPA kan prediktiva metoder baserade på Holt-Winters exponentiell utjämning (HW) och långt korttidsminne (LSTM) minska överflödig CPU-användning med över 40 % samtidigt som de undviker CPU-brist för olika arbetsbelastningar. Ytterligare visade sig LSTM generera stabilare prediktioner jämfört med HW, vilket ledde till mer robusta autoskalningsbeslut.

Nyckelord

Kubernetes, Docker, Container, Cloud Native, Cloud Computing, Resursförsörjning, Autoskalning, prediktiv skalning, CPU Användning, Säsongsmässighet, Exponentiell utjämning, långt korttidsminne, tidsserieanalys

Acknowledgements

I would like to express my deepest appreciation to my host company manager Lothar Wengerek for this valuable opportunity of writing my thesis at Ericsson AB. Also, I am extremely grateful to my supervisors Simone Ferlin at Ericsson AB and Marco Chiesa at KTH, for providing me with insightful feedback and inspiring guidance throughout the whole project.

Contents

Chapter 1

Introduction

In this research, we will be looking into how time-series analysis methods can help us predict CPU resource usage in Kubernetes. We will propose a strategy that makes use of time-series forecasting to achieve better autoscaling performance.

1.1 Background

Kubernetes is currently the most widely used container orchestration framework in the industry [1]. For many IT companies, Kubernetes has become an essential tool for managing container-based clusters and achieving cloud-native operations. This popular open-source framework is used for tasks such as automating containerized application deployment, scaling, and management. Kubernetes provides developers with numerous benefits such as load balancing, storage orchestration, automated roll-out/rollbacks, and traffic routing. As Kubernetes systems grow larger, elasticity becomes key. As the workload increases, maintaining performance requires sufficient resources to be provisioned to the applications. At the same time, it is important to free unused resources and minimize resource wastage, which is crucial in ensuring cost-efficient operation.

In Kubernetes, users are required to manually set CPU and memory resource requests for their applications to guarantee run-time performance. This is often done before the actual deployment, as part of the configuration file. However, the amount of resources required can depend on various factors, e.g., input parameters and files, workload, and traffic, which can be complex to estimate. Insufficient CPU can lead to throttling and insufficient memory leads to Out-Of-Memory (OOM) errors. As a result, many users take on a risk-averse approach, where they over-provision resources to guarantee performance. This approach can however result in poor overall utilization (below 50%) of physical resources in a cluster, and often leads to large overhead costs [2, 3, 4].

Due to the dynamic nature of cloud environments, optimal resource provisioning can be a challenging task which in many cases needs to be automated. Autoscaling is a technique that can help solve this problem.

1.2 Problem

Compute resource requests estimated manually by developers can be largely inaccurate. The authors in [5] point to the risks of provisioning too few resources, causing unacceptable performance degradation. For this reason, users tend to request more resources than what is actually needed, leading to resource wastage.

Kubernetes currently offers two default autoscaling mechanisms for container applications, *vertical* and *horizontal* autoscaling. These mechanisms adjust resource allocation and scale applications during run-time according to several measured metrics, such as CPU and memory usage. These existing mechanisms estimate the needed amount of resources by analyzing the current usage and applying moving average methods to historical usage. They are, however, unable to predict future changes in resource demand, even if these changes repeat periodically.

The currently existing mechanisms take a reactive approach - they only start to allocate more resources when they begin to run out, and similarly only start to deallocate resources when wastage is detected. This can cause quality-of-service (QoS) issues such as CPU throttling and insufficient memory, especially if reactions are slow [6]. Also, while the default mechanisms might work for stable workloads, they potentially perform poorly for workloads that display clear seasonality or change frequently. We shall later demonstrate that the existing vertical autoscaling strategy ignores reoccurring workload patterns, and takes a conservative approach that can lead to poor utilization of allocated resources.

1.3 Goals

Contrary to the current reactive approaches, a proactive approach predicts resource demands ahead of time. By preemptively scaling up or down based on these predictions, a proactive approach can achieve increased resource utilization efficiency, while meeting service-level objectives (SLOs) and QoS agreements [7].

In this paper, we apply methods such as Holt-Winters exponential smoothing (HW) and Long Short-Term Memory (LSTM) artificial neural networks for time-series analysis, to predict future CPU demand. We feed these predictions into a proposed proactive autoscaling mechanism, to increase CPU utilization while avoiding throttling. While LSTM short-term load prediction has shown to outperform season-based predictions [7, 8], these works do not *integrate* and *evaluate* the impact of the predictions into the operation of an autoscaler, which is our core contribution. Moreover, we focus on CPU usage at the container level instead of the more coarse-grained cluster level.

This research focuses on analyzing the less researched, vertical autoscaling on CPU consumption where there potentially exists a large room for improvement.

1.4 Research questions

The research questions that are presented in the section are centered around whether it is possible to achieve better autoscaling performance in Kubernetes by using a proactive rather than the current reactive approach.

The main research question is:

- *"By using a predictive autoscaling strategy, is it possible to increase CPU utilization without sacrificing performance, compared to the current Kubernetes VPA implementation?"*

Sub-questions are as follows:

1. *"By what percentage are we able to reduce CPU slack for various types of workloads, when comparing our implementation and the Kubernetes VPA?"*
2. *"Is a predictive autoscaling strategy more likely to cause CPU insufficiency?"*
3. *"What are the strengths and weaknesses of using a predictive autoscaling strategy based on exponential smoothing vs one based on an LSTM neural network?"*

1.5 Ethics and sustainability

This research may influence many of the UN Sustainable Development Goals (SDGs) in various direct and indirect ways. One of the goals that are directly affected includes SDG 12, "responsible consumption and production". The waste of computational resources such as CPU and memory directly corresponds to the ineffective power consumption of electricity-powered physical hardware. By allocating only the amount of resources that are actually needed, we may help ensure sustainable and effective consumption of energy. Indirectly affected goals include SDG 8, "decent work and economic growth". Ineffective consumption of hardware resources could potentially lead to significant increases in operational costs, which limit the chances for sustainable economic growth for companies. Advances in digital infrastructure in general also affect goals such as SDG 10, "reduced inequalities", providing more equal opportunities to people around the world independent of gender or race, to achieve prosperity.

1.6 Outline

The rest of this paper is organized as follows: Chapter 2 starts by introducing the background to the container management framework of Kubernetes and its components relevant to our research. Next, we introduce the background to exponential smoothing and LSTM networks, the key components for our proposed autoscaling strategy. Lastly, we also go through previous research on cloud resource autoscaling and CPU usage prediction. In Chapter 3 we introduce the proposed algorithms for CPU prediction and predictive autoscaling. Chapter 4 then goes through the setups used for our experiments. Chapter 5 presents the results of the experiments. Chapter 6 discusses some problems and limitations of our work, along with directions for future research. Chapter 7 provides a summary of the main findings of this research.

1.7 Summary

In this chapter, we have presented the main problem of resource provisioning within Kubernetes, and briefly introduced the strategy of predictive autoscaling we intend to employ to tackle this problem. We have also specified our research questions and discussed the potential ethical impacts our research may have on the UN Sustainable Development Goals.

Chapter 2

Background

This chapter covers a high-level overview of Kubernetes, existing autoscaling strategies, time-series forecasting based on Holt-Winters exponential smoothing and LSTM networks, and related research. Section 2.1 discusses the architecture of a Kubernetes cluster and introduces some of the most important components. We explain what container resources are, and how resource usage can be monitored. Section 2.2 introduces the currently existing autoscaling methods in Kubernetes, horizontal and vertical pod autoscaling. Sections 2.3 and 2.4 cover exponential smoothing and LSTM networks respectively. Lastly, section 2.5 discusses related work.

2.1 Kubernetes architecture

Containerized applications and container management frameworks such as Kubernetes enjoy widespread adoption with promising benefits such as flexibility, scalability, lower resource footprint, etc. The popularity comes from the management of applications to providing benefits such as load balancing, storage orchestration, automated roll-outs/roll-backs. Next, we describe some Kubernetes components that are essential to understand our work, see Figure 2.1.

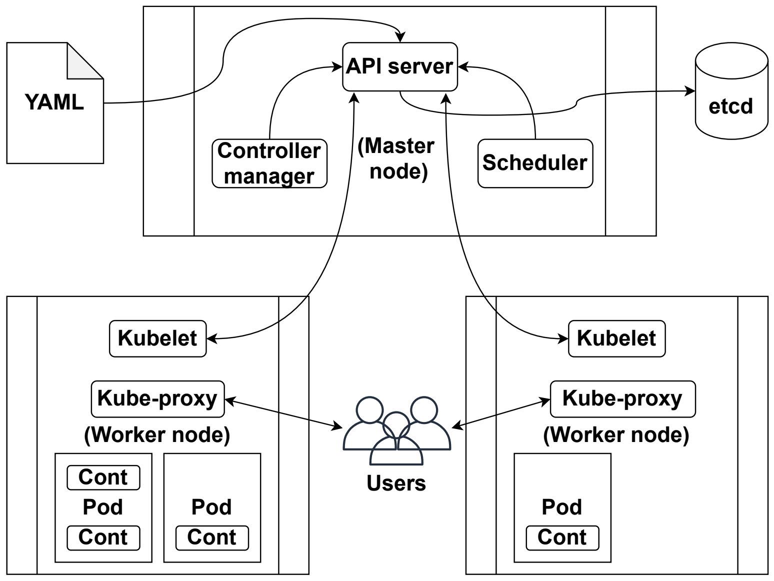


Figure 2.1 – Basic architecture of a Kubernetes cluster

2.1.1 Containers

A container can be seen as a standard unit of software, packaging up code, and all potential dependencies so that the containerized applications run reliably across different computing environments. Containers provide run-time environments for applications. They are designed to run microservices. One of the most popular types of containers used within Kubernetes is the `Docker` container[1].

Compared with virtual machines (VMs) that run a complete operating system including the kernel, containers virtualize and run the user-mode portion of an OS, which allows multiple workloads to execute on a single host OS instance. Compared to VMs, containers are lightweight and require fewer system resources such as CPU, memory, and storage [9]. On the other hand, VMs provide complete isolation between different VMs and the host OS and is capable of running a different OS than the host.

2.1.2 Pods

A pod is the most basic scheduling unit within Kubernetes, with each pod containing one or more containers [10]. Kubernetes automatically provides pods with their own cluster-internal IP addresses at the time of creation, which is used for all communication with the pods. Pods are defined through manifest files, specifying for example container images and resource requests. A set of pods belonging to the same application are identified through labels.

In Kubernetes, scaling applications according to load is done at the pod level. As load increases, we can increase the number of pods by creating replicas, and distribute the load evenly among the replicas, which is known as horizontal scaling. The alternative that this research focuses on, is to provide more resources to the containers running in every single pod, which is known as vertical scaling.

2.1.3 Deployments

A deployment within Kubernetes acts as an abstraction layer for the pods. Pods within Kubernetes are generally deployed using deployments. The deployment object can be used to schedule multiple pod replicas. The fundamental purpose of the deployment object is to maintain resources declared in the deployment configuration in its desired state. The desired state of a deployment, such as the number of pod replicas, is defined in a configuration file commonly written in `YAML` format[11]. Such a configuration file also contains pod specifications such as container image, volumes, ports, and resource requests and limits. A Controller manager component, running in the Master node, is then responsible for changing the actual state to the desired state at a controlled rate. According to the specification of deployments, pods are created and destroyed dynamically. Pod belonging to the same deployment can run on different machines or nodes.

2.1.4 Nodes

At a high level, a Kubernetes cluster consists of at least one master node and one or more worker nodes, visualized in figure 2.1. Worker nodes are either virtual machines or physical servers, with each node containing the services required to run pods. Every worker node runs a container run-time such as `Docker`, which enables pods to run within the node. The `Kubelet` component contained within each worker node is responsible for inspecting pod specifications given by the Application Programming Interface (API) server contained in the master node, to ensure that those pods are running in a healthy state, and restart pods that have crashed. The other important component inside each worker node is the `Kube-proxy`, which is responsible for maintaining the distributed network within the cluster and exposes services to the outside world.

The master node is responsible for managing the Kubernetes cluster, handling tasks such as scheduling, provisioning, controlling, and exposing API to the clients. As stated above, the master node contains an API server, which is responsible for exposing Hypertext Transfer Protocol (HTTP) APIs that let end-users, cluster components, and external components communicate with one another. Some of the most important APIs are for example those for creating, deleting, modifying, and displaying resource components such as pods in the cluster. The Kubernetes API also allows one to query and manipulate the state of objects such as pods and deployments. The command line tool `kubectl` can be used to interact with the API server and perform operations in Kubernetes.

A master node also contains the Scheduler component, which is responsible for scheduling pods onto appropriate nodes, based on their configurations. The master node further contains a Controller manager component which is responsible for maintaining the health of the cluster, making sure the nodes and pods within the cluster are running correctly. For example, if a worker node goes down, the Controller manager will send commands to the Scheduler to reschedule the pods previously on the node to another one.

Configuration files are used for declarative management of objects such as pods and deployments within Kubernetes. When the controller manager detects differences between the current state of an object within the cluster and the defined state, it makes changes to fix the problem. The last important component in the master node is the `etcd` key-value database, which is responsible for storing information about the cluster state.

2.1.5 Container resource requests and limits

Container CPU and memory requests and limits can be specified in the pod's configuration file, where CPU is given in millicores and memory in bytes. The Kubernetes `Scheduler` in Figure 2.3 places pods onto nodes based on these definitions, where a pod is scheduled only if there are enough resources.

In the pod's configuration file, for each container, there are two key fields: "request" and "limit". The request field refers to the minimum amount of resources reserved for the container exclusively, see Figure 2.2. If there are resources on the node, applications can consume them beyond this specified minimum, however, bounded by the limit field. Decisions made by the `Sched-`

```
containers:
  - name: nginx
    image: nginx:1.18.0
    resources:
    requests:
      memory: "100Mi"
      cpu: "500m"
    limits:
      memory: "200Mi"
      cpu: "1000m"
```

Figure 2.2 – Container resource definitions

`uler` are based on the resource requests field and do not depend on the limits.

The purpose of the resource request field is to guarantee the application enough resources to execute normally even in the case of resource contention. Not defining the resource request field can lead to pods being scheduled onto nodes with insufficient resources to sustain the containers of the pod, which could lead to critical run-time issues. While insufficient CPU leads to CPU throttling, causing poor performance and increased latency, insufficient memory leads to Out-Of-Memory (OOM) errors which cause pods to terminate.

2.1.6 Metrics server

The Metrics Server component provides the most recent CPU and memory metrics of all pods and nodes on the cluster. This information is collected periodically from the `Kubelet` running on each node. The default collection rate is once every 60 seconds. This rate can be lowered to a minimum of 15 seconds per collection. The collected metrics are aggregated and stored in memory, ready to be served in Metrics API format. Only the most recent value of each metric is saved. The major drawback is that since metrics are stored in memory if the Metrics Server restarts, all data will be lost [12].

2.2 Kubernetes autoscaling

This section introduces the currently existing autoscaling techniques in Kubernetes: the Horizontal Pod Autoscaler and the Vertical Pod Autoscaler.

2.2.1 Horizontal Pod Autoscaler

The Kubernetes Horizontal Pod Autoscaler (HPA), dynamically scales the number of pods of a deployment by adding or removing pod replicas. As the load increases, replicas are created to share the workload. When the load decreases, replicas are removed when they become unnecessary.

HPA objects are created for and target specific deployments through name-attributes defined in a configuration file. In this configuration file, a single metric that is used to estimate load is also defined, along with the ideal value of this metric. This metric could for example be CPU/memory or other metrics such as incoming request rate or latency. By default, HPA makes use of CPU or memory metrics provided by the Kubernetes Metrics Server.

HPA can be configured to either use the direct values of the resource metrics or percentages of the requested value. It can also be configured to take the average of the given metric across all targeted pods.

To calculate the desired amount of pod replicas that should be running at a given time, HPA employs the simple algorithm shown in the equation below.

$$targetReplicas := ceil[currentReplicas \times \frac{currentMetricValue}{targetMetricValue}] \tag{2.1}$$

$targetReplicas$ indicates the recommended number of replicas HPA will try to maintain, which is based on the current number of replicas $currentReplicas$ and the ratio between the current metric value and ideal metric value. If this ratio is sufficiently close to 1.0, exactly how close is determined by the tolerance flag which defaults to 0.1, scaling will be skipped. HPA also records scale recommendations. Before HPA scales a deployment, the controller considers all recommendations within a set downscale stabilization period, choosing the highest recommendation for the scaling operation. The length of this period is configurable, and the default is set to 5 minutes [13]. Because of this stabilization period, although upscale operations are executed almost immediately, downscale operations will occur gradually, which helps to smooth out the impact of fluctuating metric values. It is also possible to define a limit for the rate at which pods are removed by the HPA. Furthermore, the minimum and the maximum number of replicas can also be specified.

2.2.2 Vertical Pod Autoscaler

The Kubernetes Vertical Pod Autoscaler (VPA) is used to scale CPU and memory resources provisioned to containers running inside the pods of a deployment. Instead of scaling the number of pod replicas, which is done during horizontal scaling, vertical scaling scales the amount of resources allocated to the containers within a single pod.

Running the VPA start-up script introduces a custom resource type called $VerticalPodAutoscaler$ and deploys the VPA application to the cluster. To scale the pods of a deployment using VPA, we have to define a $VerticalPodAutoscaler$ resource object that targets the deployment. By editing the $updateMode$ field in the definition, it is possible to toggle the automatic scaling function on and off for each deployment.

There are three main components to the VPA application, which are introduced to the cluster as deployments. These three components and their relationships are illustrated in figure 2.3.

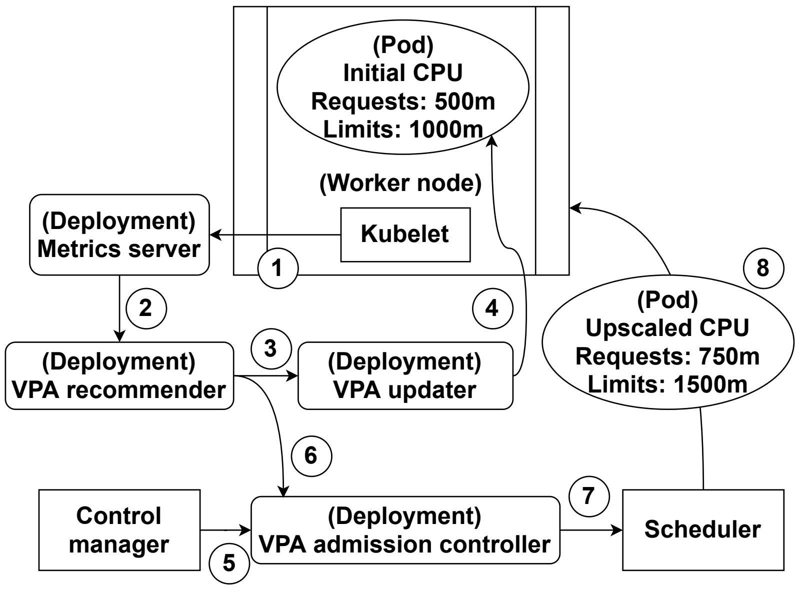


Figure 2.3 – Vertical Pod Autoscaler architecture

The first component is called the VPA Recommender, which is responsible for gathering pod CPU and memory consumption from the `Metrics Server`, collected from the `Kubelets` running on the worker nodes, see Figure 2.3 ($\textcircled{1}$) and ($\textcircled{2}$).

Based on these values, the `Recommender` provides CPU and memory recommendations for the monitored containers ($\textcircled{3}$). For each container, the recommendation implements a lower and upper bound, along with target values.

How the VPA bounds and target values are calculated will be explained in detail later on. Recommendations are produced for all deployments within the cluster that are targeted by a $VerticalPodAutoscaler$ resource object, even if the $updateMode$ is toggled off. Gathering resource recommendations while the $updateMode$ is toggled off could be potentially useful for understanding the behavior and performance of VPA on a specific pod.

The second component is the VPA Updater ($\textcircled{4}$), which evicts a pod when the requested CPU goes over the upper or under the lower bounds. The VPA Updater runs this check for all targeted pods every minute. Once evicted, it will be rescheduled by the cluster's `Control Manager` ($\textcircled{5}$).

The new pod then passes through the VPA `Admission Controller` ($\textcircled{6}$), which registers an admission *webhook* in the Kubernetes API. Every pod submitted to the cluster goes through this *webhook*, checking whether there is a $VerticalPodAutoscaler$ object referencing the deployment it belongs to. If there is, the Admission Controller will update the pod's container resource requests according to the target values calculated by the VPA Recommender. If resource limits are defined, they will also be updated to keep the same limit to request ratio as originally specified.

The pod is then rescheduled according to the updated values ($\textcircled{8}$). The original deployment specification is not changed by VPA. Due to the current design of VPA, pods need to be restarted for their resource requests and limits to be updated by the Admission Controller. As such, when pods are restarted with updated resource configurations, they may become scheduled on a different node than before. Because behavior may become unpredictable, HPA and VPA should not be used together when scaling on the same metrics [14].

VPA algorithm

The recommendation target, lower and upper bounds calculated by the VPA Recommender are based on a decaying histogram of weighted CPU usage samples from the metrics-server. A default half-life value of 24 hours controls the speed at which sample weights decrease. These samples are collected at a rate of one sample per minute.

As mentioned previously, the VPA updater evicts a pod when the requested CPU value goes outside of the upper or lower bounds. This can be seen in Figure 2.4 illustrating a newly started container, where we can see the lower bound slowly approaching the VPA target as we gather more samples. The VPA CPU target is calculated using the 90th percentile of all historical CPU samples, and the lower and upper bound values are the 50th and 95th percentiles, respectively. In general, VPA aims to keep CPU target recommendation above actual usage 90% of the time.

The lower and upper bounds indicate the confidence interval of the recommendation. Requested resources above the upper bound or below the lower bound, are seen as wasted or insufficient respectively. Once the requested CPU crosses the upper or lower bound, the pod is evicted and restarted with new requests set to the VPA target as seen in Figure 2.4. In the figure, we can see the CPU usage falling to zero after each re-scale. This is because the Metrics Server needs time to collect metrics from the newly restarted pod.

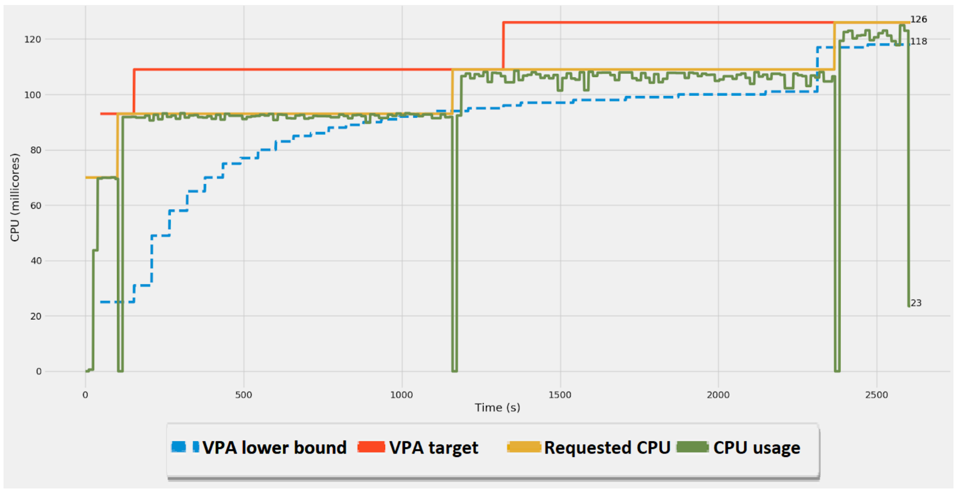
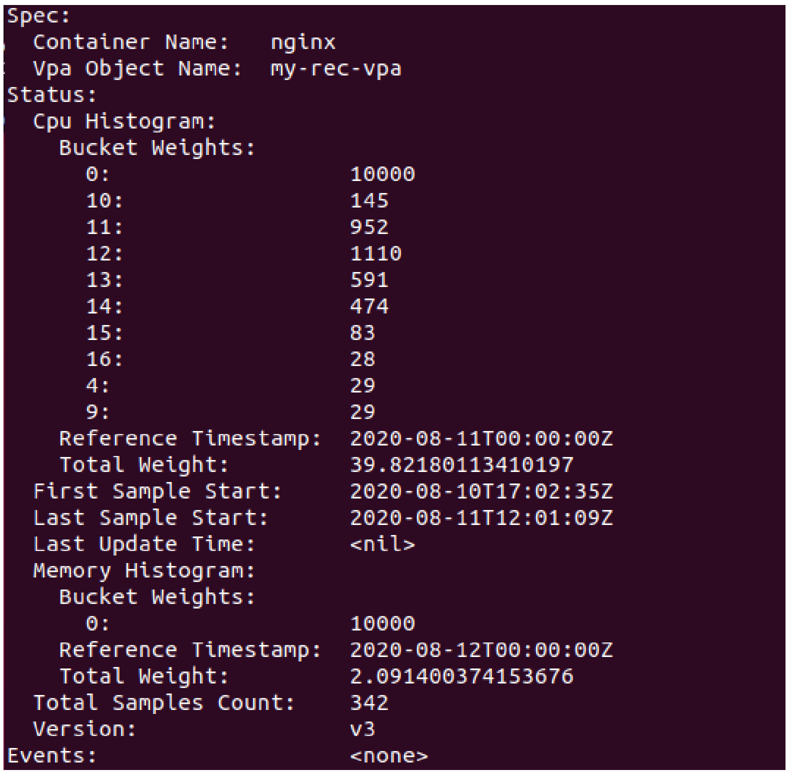


Figure 2.4 – Kubernetes VPA CPU scale up

The lower and upper bounds are also modified with confidence multipliers based on how long a deployment has been monitored or equivalently, how many samples have been collected for its pods. The fewer samples the pods have collected, the harder they will be to evict.

The actual implementation of the decaying histogram is an exponential histogram. The histogram is made up of a set number of buckets with exponentially increasing bucket sizes. Each bucket corresponds to a range of CPU usage values, with the first bucket being no usage at all, the second bucket being slightly higher CPU usage, up until the last bucket with corresponds to very high CPU usage. With every CPU usage sample collected, weight is added to one of the buckets depending on the corresponding value of the sample. Also, the last bucket does not have an upper bound but instead covers CPU samples with values up until infinity. The weights of each bucket are periodically multiplied by a constant smaller than one, to simulate decaying weights.

A $VerticalPodAutoscalerCheckpoint$ resource object, see Figure 2.5, generated by VPA stores information about the histogram and weights. These objects are persistent in the cluster even after restarts. A checkpoint is always tied to a VPA object name.



```
Spec:
  Container Name:   nginx
  Vpa Object Name:  my-rec-vpa
Status:
  Cpu Histogram:
    Bucket Weights:
      0:                    10000
      10:                   145
      11:                   952
      12:                   1110
      13:                   591
      14:                   474
      15:                   83
      16:                   28
      4:                    29
      9:                    29
    Reference Timestamp:  2020-08-11T00:00:00Z
    Total Weight:         39.82180113410197
  First Sample Start:     2020-08-10T17:02:35Z
  Last Sample Start:      2020-08-11T12:01:09Z
  Last Update Time:       <nil>
  Memory Histogram:
    Bucket Weights:
      0:                    10000
    Reference Timestamp:  2020-08-12T00:00:00Z
    Total Weight:         2.091400374153676
  Total Samples Count:    342
  Version:                v3
Events:                   <none>
```

Figure 2.5 – Exponential histogram buckets implementation in *VerticalPodAutoscalerCheckpoint* object

Figure 2.5 shows the CPU and memory histograms saved within a *VerticalPodAutoscalerCheckpoint*. The figure shows the weights for the CPU histogram being distributed over several buckets, with the first bucket "0", having the most weight. We can also see that a similar histogram exists for memory usage and that its weight is all concentrated in the first bucket.

To get the VPA recommendation target, or upper and lower bounds, a $Percentile()$ function is used to move through the buckets until the asked for the percentile of total weight mass is reached. At this point, the upper boundary of the current bucket is returned. For the last bucket, however, the lower boundary is returned.

2.3 Exponential smoothing

This section will present exponential smoothing for time-series forecasting, which will be one of the main methods use for our proposed prediction algorithm. Exponential smoothing first introduced in the late 1950s has inspired some of the most successful forecasting methods that exist today [15]. Exponential smoothing methods make use of weighted averages of historical observations to generate forecasts [16, 17, 18]. The weights used decay exponentially with time, hence the name exponential smoothing. These methods can generate reliable forecasts quickly and apply to a large range of time series, which makes them favorable in many real-life applications.

In this section, we will first go through the most basic form of exponential smoothing, simple exponential smoothing. Building upon that, we introduce the concepts of trend and seasonality, along with the more advanced forms of exponential smoothing, including the Holt-Winters method which will be used for this research.

2.3.1 Simple exponential smoothing

Simple exponential smoothing is most suitable for forecasting data that displays no clear trend or seasonality. Similar to moving averages, the core idea behind simple exponential smoothing is that we base forecasts on previous observations, with more recent observations given higher weights. However, while moving averages usually consider a set number of historical observations, simple exponential smoothing considers all previous data points, while assigning exponentially smaller weights the further back we go. For each time-step t, we can obtain the smoothed value, or **level** l_t, by using the following **level equation** [19]:

$$l_t = \alpha \times y_t + (1 - \alpha) \times l_{t-1} \tag{2.2}$$

In this equation, α is the **smoothing factor**, which decides how much weight the most recent observed value is given and controls how fast we forget past values. The higher α is set, the faster past values lose significance. Note that $0 \leq \alpha \leq 1$. y_t is the observed value at time-step t. For simple exponential smoothing, at time-step t, a forecast h steps ahead of time $\hat{y}_{t+h}$, is given solely by the level l_t, of the last observation. The **forecast equation** is thus:

$$\hat{y}_{t+h} = l_t \tag{2.3}$$

From the above equation, we can see that simple exponential smoothing provides the same forecast value for all future values.

2.3.2 Double exponential smoothing

Simple exponential smoothing does not perform well when **trend** is present in the data. The trend or slope of a series at time t indicates the steepness at which the data is increasing or decreasing. Double exponential smoothing introduces a **trend equation** [19]:

$$b_t = \beta \times (l_t - l_{t-1}) + (1 - \beta) \times b_{t-1} \tag{2.4}$$

Here, b_t denotes an estimate of the trend of the series at time t, and β is the smoothing parameter for the trend, where $0 \leq \beta \leq 1$. Just like for α, β controls how much weight is put on the most recent trend and how fast historical trends lose significance. In the above equation, we are calculating the trend by subtracting the levels l_t and l_{t-1}. This is known as the **additive** trend method. Instead of subtracting l_{t-1} from l_t, using division would give us a ratio. This is known as the **multiplicative** trend method. In this research, however, we will only be focusing on data with additive trends.

For double exponential smoothing, the **level equation** is extended to include a trend element:

$$l_t = \alpha \times y_t + (1 - \alpha) \times (l_{t-1} + b_{t-1}) \tag{2.5}$$

Now, not only is the current level dependent on the past levels, but also past trends. This way we manage to capture the direction of movement of the data. Put together, we have the **forecast equation** for h steps into the future:

$$\hat{y}_{t+h} = l_t + h \times b_t \tag{2.6}$$

Compared to simple exponential smoothing, the forecast function for double exponential smoothing is no longer constant, but it is trending. The h steps ahead forecast is a linear function of h and depends on the last estimation of both the level and the trend of the series.

2.3.3 Holt-Winters method (Triple exponential smoothing)

Holt-Winters method extends double exponential smoothing to capture seasonality in data [20]. The method consists of a forecast equation and three smoothing equations for level l_t, trend b_t, and the seasonal component s_t. The corresponding smoothing parameters are α, β, and γ. We also introduce the parameter L to indicate the frequency of the seasonality, also known as season length.

Just as for trend, there are two variations to this method, depending on whether we use an **additive** or **multiplicative** seasonal component. The additive method is favored in situations where the seasonal variations are consistent throughout the series, while the multiplicative method is preferred in situations where the seasonal variations of the data are changing proportionally to the level of the series [19].

Building upon double exponential smoothing, in addition to applying exponential smoothing to the level and trend components, the Holt-Winters method also applies exponential smoothing to the seasonal components. This smoothing is applied across seasons, meaning that the seasonal component of nth step into the season is exponentially smoothed with regards to the corresponding nth step from the last season, last season, and so on.

Here is the additive version of the **seasonal component equation** for Holt-Winters method:

$$s_t = \gamma \times (y_t - l_{t-1} - b_{t-1}) + (1 - \gamma) \times s_{t-L} \tag{2.7}$$

We can see that the equation for the seasonal component consists of a weighted average between $(y_t - l_{t-1} - b_{t-1})$, the seasonal index for step t, and the seasonal index of the corresponding step $t - L$, last season.

Next we have the **level equation**:

$$l_t = \alpha \times (y_t - s_{t-L}) + (1 - \alpha) \times (l_{t-1} + b_{t-1}) \tag{2.8}$$

For the level, we take the weighted average between $(y_t - s_{t-L})$, the seasonally adjusted observation, and $(l_{t-1} + b_{t-1})$, the non-seasonal forecast for time t. By subtracting s_{t-L}, the last seasonal component from y_t, we are effectively removing any seasonality from the level component.

The **trend equation** is the exact same as the one for double exponential smoothing:

$$b_t = \beta \times (l_t - l_{t-1}) + (1 - \beta) \times b_{t-1} \tag{2.9}$$

Putting everything together, we have the **forecasting equation**:

$$\hat{y}_{t+h} = l_t + h \times b_t + s_{t+h-L(k+1)} \tag{2.10}$$

where k is $\frac{(h-1)}{L}$. $s_{t+h-L(k+1)}$ is the seasonal component of the corresponding step into the last observed season. This is needed because only past seasonal information should be accessible, not future information.

2.3.4 Optimizing parameters

For exponential smoothing, the smoothing parameters used for the calculation of the components need to be optimized to achieve good forecasting results. For Holt-Winters method, the parameters α, β, and γ are optimized. Although the initial states l_0, b_0, s_0, s_{-1}, ..., s_{-L+1}, can be estimated using general formulas, they can also be set through optimization.

One common way to estimate the parameters is by minimizing the Sum of Squared Errors (SSE) between the observed values and the smoothed values given by the model [21]. To find the best parameters, methods such as grid-search can be used. Another way to estimate the parameters is by maximizing the likelihood function. The likelihood is defined by the probability of the observed data arising from the specified model. A large likelihood indicates a good model.

2.4 Long-short term memory

LSTM networks are a special type of Recurrent Neural Networks (RNNs) used for identifying patterns in sequential data. Generally, LSTMs are suited for classifying and making predictions based on time series data as there may be long delays between important events. RNNs and specifically LSTM networks in recent years have led to many breakthroughs in natural language processing (NLP) research areas such as speech recognition, text-to-speech synthesis, and machine translation [22, 23].

Basic RNNs are built by chaining together RNN cells, containing the hidden states of the network that work as a memory mechanism. For each time-step, an RNN cell receives the input of the current time-step and combines it with the hidden state output from the previous cell. The result is then output as the next hidden state. This recurrence relationship effectively causes the hidden state output of the current cell to be dependent on all previous inputs of the sequence. The hidden states are also used to generate predictions, for example, the next output in a sentence.

Traditional RNNs, however, suffer from the vanishing/exploding gradient problem, which negatively affects their ability to capture long-term dependencies in data[24]. Because of this problem, RNNs are often unable to consider data outside of the recent history.

LSTM network cells, see Figure 2.6, build upon the basic RNN cell by adding a cell-state along with gates regulating the information of the hidden states. This allows LSTMs to avoid the vanishing/exploding gradient problem of RNNs and helps the network to recognize reoccurring patterns that may span over longer periods of time.

Mathematically, within each LSTM cell, the following computations are made[25]:

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \tag{2.11}$$

$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \tag{2.12}$$

$$g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hc}h_{t-1} + b_{hg}) \tag{2.13}$$

$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \tag{2.14}$$

$$c_t = f_t * c_{t-1} + i_t * g_t \tag{2.15}$$

$$h_t = o_t * \tanh(c_t) \tag{2.16}$$

where h_t is the hidden state, c_t is the cell state, x_t is the input at time t. Likewise, i_t, f_t, g_t, o_t denote the input, forget, cell, and output gates at time t respectively. The various Ws in the equations represent the weight matrices of the network, that are tuned during training.

The forget gate uses a sigmoid activation function which outputs values between 0 and 1, which is used to regulate what information is to be kept from the previous cell state c_{t-1}. Similarly, the input gate also uses a sigmoid function to regulate what information from the cell gate should be added to the cell state. The cell gate uses a tanh function that outputs values between -1 and 1. Squashing the values helps to mitigate the vanishing/exploding gradient problem. The output gate is used to regulate what information we should keep from the new cell state after going through a tanh function, to form the new hidden state.

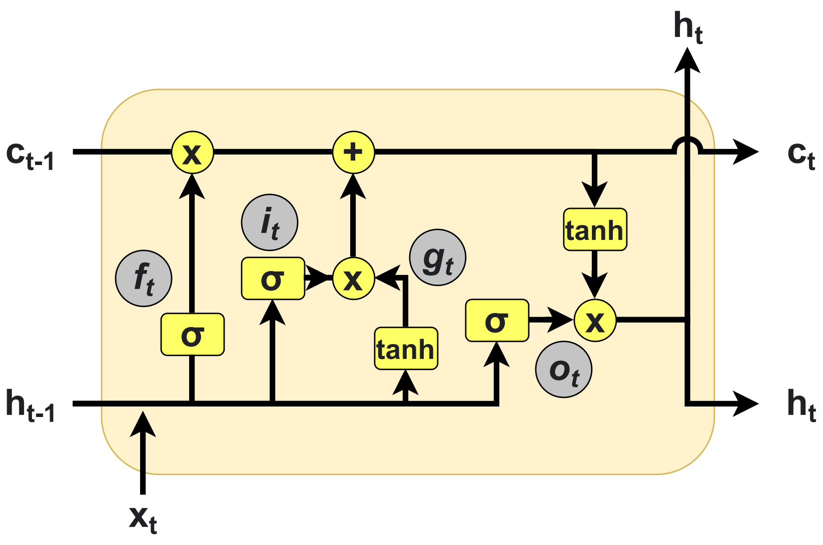


Figure 2.6 – Basic LSTM cell

2.5 Related work

We divide related work into research related to resource autoscaling in cloud environments and CPU usage prediction.

2.5.1 Cloud resource autoscaling

Cloud resource autoscaling is a research area that has gained considerable interest over recent years. Some of the previous surveys have analyzed the elasticity of public clouds and explored autoscaling techniques for elastic applications in cloud environments [26, 27]. However, most of the existing researches deal with horizontal autoscaling. In similar research, the authors analyze the performance of multiple horizontal autoscaling algorithms for various workflows in a cloud environment [28]. Another recent research evaluates the performance of horizontal autoscaling for VMs and Kubernetes pods in the public clouds of Amazon Web Services [29].

At the same time, recent researches into vertical autoscaling in cloud systems have analyzed the autonomic vertical elasticity of Docker containers [30]. Another survey analyzes the performance of vertical autoscaling models on stream joins in cloud infrastructures [31]. One significant recent research focuses on vertical scaling of memory resources for jobs running in Google's Borg cloud systems [32]. In this research, the authors present the autoscaler known as Autopilot, which manages to reduce memory slack from 46% to 23% compared to manually-managed jobs in Google's clusters. At the same time, it also managed to reduce the number of jobs severely impacted by OOM by a factor of 10.

Similarly, the goal of our research is to reduce CPU slack and avoid at the same time insufficient CPU. However, instead of directly predicting future resource usage, Autopilot uses exponentially-smoothed sliding windows over historic usage to generate resource limits. Reinforcement learning techniques are then used to select the best performing window. Autopilot uses a reactive autoscaling strategy that sets resource limits based on past historical usage, which is different from our proactive strategy based on exponential smoothing and neural networks. For cases where a simple moving window does not manage to react quickly enough, a proactive autoscaling method could potentially help prevent service-level agreement (SLA) violations. Also, by recognizing and taking advantage of repeating patterns, a proactive strategy is potentially able to decrease slack even further.

Previous works directly related to vertical scaling in Kubernetes include the Kubernetes VPA [33], which sets container resource requests and limits using statistics over a moving window as described in 2.2.2. We will use the default Kubernetes VPA as the baseline when evaluating the performance of our implementation. Another research has studied the disruptive impacts of vertical scaling on the performance of containers running in Kubernetes [34]. In this research, the authors analyze performance metrics such as application latency and connection time, and the results obtained from experiments concluded that vertical scaling had no significant impact on performance. Other related works have explored the possibility of non-disruptive vertical autoscaling in Kubernetes [35]. The authors in this research incorporated container migration, to develop a prototype non-disruptive autoscaler called RUBAS, which managed to improve CPU and memory utilization of a test cluster by 10%.

2.5.2 CPU prediction

Many previous surveys have analyzed the prediction of CPU usage in cloud environments [36, 37, 38]. In one study, the authors use Long short-term memory (LSTM) neural networks and seasonal ARIMA (SARIMA) models to forecast cluster CPU usage [7]. These two methods were applied to CPU usage data sampled from a Microsoft Azure cluster. The original data with the granularity of 1 observation per minute was re-sampled with a frequency of one data point per 20 minutes. A strong daily seasonal component was identified by decomposing the data-set. Both models were evaluated by using the mean average percent error (MAPE) between the predictions and a test set. Results showed that the LSTM neural network was more robust and managed to perform better than the SARIMA model for the short-term task of predicting usage up to an hour into the future. However, for the long-term task of predicting usage over a period of three days, SARIMA was found to be superior. Furthermore, it was also concluded that the SARIMA model required the data to meet certain assumptions about seasonality.

This study, however, does not integrate the prediction into the operation of an autoscaler, which will be the key for our research. Rather than the MAPE value, we are more interested in autoscaling performance measured in CPU slack and insufficiency, etc. Also, we will be focusing on CPU usage on the container level rather than the cluster level.

2.6 Summary

This chapter has presented the basic architecture of Kubernetes clusters and discussed existing autoscaling mechanisms. We have also presented the backgrounds of exponential smoothing and LSTM networks in detail. We have seen how both methods differ vastly in their approaches to generating predictions. Lastly, we also introduced various related research on cloud resource autoscaling and CPU prediction.

Chapter 3

Methods

This chapter introduces our proposed algorithms for CPU prediction and vertical autoscaling.

Section 3.1 and 3.2 describe the details behind each prediction algorithm and discuss the motivation behind the various design choices. We will first talk about the prediction algorithm that is based on Holt-Winters method, and then LSTM. Following that, Section 3.3 will explain how autoscaling can be conducted based on the generated predictions.

3.1 Holt-Winters prediction algorithm

We use `Statsmodels` [39] to implement the Holt-Winters (HW) exponential smoothing prediction algorithm. The HW additive model requires a *Season length* and historical data of at least 2 $\times$ *Season length*, i.e., in all experiments, we start predicting after gathering container CPU usage data for at least two seasons. The season length is set to 144 time-steps, corresponding to 24 hours, with each time-step being 10 minutes. We assume that the season length is known beforehand.

For each time-step, we fit the model with the most recent CPU samples, dating up to *History length* time-steps into the past. We set the *History length* parameter to 8 seasons to take into consideration weekly patterns. By passing the *Optimized* = *True* parameter to the fit function, the model parameters were automatically optimized by maximizing the log-likelihood.

Using the fitted model, we generate a prediction window consisting of 24 future values starting from the current time-step. For this prediction window, we calculate the target value (90th percentile), lower (60th percentile), and upper (98th percentile) bounds. We choose these percentiles as they performed better than others in our experiments.

The reason a single prediction value is not used, but rather a high percentile of a window of predicted values and smoothed values, is because it reduces fluctuations in the prediction values. Rather than the exact values, we are more interested in the amount of CPU we need to request to accommodate for the majority of the upcoming usage. In this way, temporary sudden changes in container CPU usage will be less likely to have a big impact on the consistency of the predictions.

The prediction algorithm executes once per time-step, every time a new CPU usage observation is collected. This means that the Holt-Winters model must be recreated for every new data point. Every time the model is recreated, the last observation is added to the input data. By doing this, the accuracy of predictions is improved, as they will always be done on the latest available data.

Algorithm 1: Holt-Winters prediction

```
Input: Container CPU usage data up until now
Output: Prediction target, upper and lower bounds
if length(Past CPU usage) > Season length * 2 then
    $data \leftarrow$ last History length data points from input;
    $model \leftarrow$ create HW model with specified Season length
      using data;
    Fit the model to data;
    $window \leftarrow$ predict Window size future values using model;
    return 60, 90, 98th percentiles of window;
else
    return None;
end
```

3.2 LSTM prediction algorithm

We implement LSTM using `Keras` [1] 2.4.3 with two hidden layers. After experimenting with multiple values, we chose the dimension of the hidden states for both layers to be 50. Time-series CPU usage data was normalized and pre-processed to single-dimensional training features. The length of each input vector was determined by a *step_in* parameter, which corresponded to *step_in* past CPU usage values. After testing multiple values, we chose *step_in* = 96 for all tests. However, because of issues relating to execution time, *step_in* = 48 was used instead for the real-time tests, later explained in Section 4.3.

[1] `https://keras.io/api/`

Training labels contain three values: The lower (60th percentile), target (90th percentile), and upper (98th percentile) bounds for the 24 values following the input values. These parameter settings match the corresponding ones for the prediction window used for the HW implementation. A fully-connected layer, following the hidden layers, was used to generate an output vector containing the predictions for the three label values.

Algorithm 2: LSTM prediction one step

```
Input: Container CPU usage data up until now
Output: Prediction target, upper and lower bounds
if length(Past CPU usage) > Season length * 2 then
    $input$ ← last step_in data points from input;
    $output$ ←Feed input into model;
    return $output$;
else
    return None;
end
```

3.3 Predictive autoscaling algorithm

Our proposed autoscaling algorithm (Alg. 3) takes as input the prediction target, upper and lower bound values from either of the two prediction algorithms. Whenever the requested CPU is lower than the lower bound or higher than the upper bound, the requested CPU is re-scaled to the target. We also add an extra re-scaling buffer (120 millicores), to allow some room for error. Note that we have to subtract this buffer from the current requested CPU before doing the bounds check. A re-scale cool-down (18 time-steps), and minimum change check (50 millicores difference) is used to prevent unnecessary re-scaling. The cool-down condition prevents two re-scale events within less than 18 time-steps. The minimum change check prevents a re-scale event that attempts to adjust the requested CPU by less than 50 millicores.

Algorithm 3: Predictive autoscaling algorithm

```
Input: Prediction target, upper and lower bounds
Output: None
$new\_requested$ ← target + 120;
if Current requested CPU is outside of bounds then
    if Rescale cool-down $\leq$ 0 then
        if Abs(current requested $- new\_requested$) > 50 then
            Rescale to new_requested;
            Reset cool-down;
        end
    end
else
    Decrease cool-down;
end
```

3.4 Summary

This chapter introduced our proposed algorithms for CPU prediction and our algorithm for predictive autoscaling. We have discussed important design decisions and details of each algorithm. We have also shown how the proposed autoscaling algorithm works independently and can accept predictions from either prediction algorithm.

Chapter 4

Experimental Setup

This chapter discusses the experimental setups that were used to evaluate the performance of the algorithms.

We divide our experiments into three parts. First, in Section 4.1 we use the historical CPU usage of two containers from Alibaba's Open Cluster Trace 2018 [40] data-set to evaluate our algorithms. After that, in Section 4.2 we use synthetically generated CPU usage data outside of a Kubernetes cluster to assess the effects of varying seasonality and noise. Lastly, in Section 4.3 we run experiments inside of an actual Kubernetes cluster, scaling test containers in real-time with full control over the load generation.

4.1 Alibaba Open Cluster Trace 2018

First, we verify the proposed prediction algorithms on historical real-world container CPU usage gathered from Alibaba [40]. This data-set contains traces from containers running on 4000 machines over a period of 8 days. We select containers, c_1 and c_10235, which display seasonality of various degrees to test our algorithm. As many of the time-steps in the trace are spaced at irregular intervals of around 3, 5, and 10 minutes, we re-sampled the data of c_1 and c_10235 into 10 minutes per sample by linearly interpolating values between two data-points. Also, we removed all data points within the first 24 hours, as these were collected at highly irregular intervals.

These experiments run outside of Kubernetes without recommendations from VPA. Therefore, we use the 90th percentile of simulated CPU usage with an additional buffer of 50 as a reasonable estimate of the VPA target value. This buffer is motivated by the VPA target recommendation always slightly overshooting the historical 90th percentile usage [33].

4.2 Synthetic CPU workload generation

Thereafter, the performance of the predictive autoscaling is evaluated on artificially generated time-series simulating CPU usage. This way, we can have full control of various CPU loads with different degrees of seasonality and noise. The load is generated according to Equation 4.1, which models a sinusoidal load with a configurable amount of noise:

$$\text{CPU usage} = \alpha \times A \times sin(2\pi F \times x + C) + D + (1 - \alpha) \times e \tag{4.1}$$

The *sin* function has an amplitude A of 300 millicores, a frequency F equivalent to a period of one day (consisting of 144 points, one every ten minutes), a phase shift C of $0°$, and a vertical offset D of 200 millicores.

We add random noise e to simulate unpredictable CPU usage changes. We draw the noise component from a normal distribution with a mean of 0 and standard deviation of 300 matching the amplitude of the *sin* function.

The α value sets how much the workload reflects the sinusoidal function or an added noise. A value of $\alpha = 1$ represents a perfectly sinusoidal workload while a value of $\alpha = 0$ consists of a purely random signal. Note that negative values are set to 0. We also vary α from 0.1 to 1 in 0.1 steps. We also estimate the VPA target in the same way as for the Alibaba Cluster trace.

4.3 Real-time CPU workload generation

Now we evaluate both algorithms and our proposed autoscaler with controlled workloads in a real Kubernetes cluster. The purpose of the experiments is to verify that our algorithm brings tangible benefits in a real-world cluster, comparing against the default VPA autoscaler.

Just as in the synthetic experiments, the seasonality of the generated workloads is 144 time-steps, simulating 1 sample per 10 minutes for one day. We use the Kubernetes `Metrics Server` to collect CPU usage samples from a `NGINX` Web server application deployed on a pod. We collect metrics every 15 seconds and reduce the period to 2160 seconds to still handle 144 samples per period as in the synthetic experiments. We also lowered the default VPA half-life time from 24 hours to 2160 seconds accordingly.

The Kubernetes cluster version used is 1.17.0 with a container Docker run-time version 18.9.9, with a single master and worker node with 47900 millicores allocatable CPU and 158.3 GB of memory. Both nodes run on Ubuntu 18.04.4 LTS on top of a 5.3.0 Linux kernel.

The real-time experiments use the widely adopted NGINX web server [41]. This deployment contains a single pod with a single container, built using the nginx:1.18.0 Docker image.

We rely on Slowcooker[42] to send periodic HTTP GET requests to the NGINX server. The load on the NGNIX server is proportional to the number of requests from Slowcooker. Each season starts with a low request rate of 700 requests per second (RPS), which is slowly increased linearly up until a peak of 7000 RPS, over a period of 600 seconds. Thereafter, the request rate is lowered in the same way back to 700 RPS, where it stays until the beginning of the next season.

We set the initial CPU requests for NGINX to 700 millicores, which is sufficient to evaluate our predictive algorithms. During the experiments, we set the CPU limit constant at 1000 millicores, avoiding throttling. Our workload does not affect memory usage, so we do not set any limit for it.

We use Kubernetes VPA version 0.8.0 in recommendation mode. We disable auto-scaling so that we can use the VPA recommendation target solely as a comparison baseline when evaluating our prediction algorithms.

Note that the *step_in* for the LSTM prediction algorithm was set to 48 for the real-time tests, to limit execution time.

4.4 LSTM training

For the synthetic and real-time experiments, we train the model on the same two seasons ($2 * 144$ observations) of training data as the HW model, collected before we start generating any predictions. For these experiments, we train for 15 epochs using a batch size of 32, and for every new season, we re-train the model on the data collected up until that point. As for the Alibaba cluster trace experiments, we split the data-sets into training and validation using a 70/30 split. We train the model only once at the beginning with the training set. We use the validation set during training for early stopping. For all experiments, we use the Mean Squared Error (MSE) loss function. We set the maximum number of epochs to 30 with a "patience" value set to 3 for early stopping.

LSTM training and forecasting was done using CPU only.

4.5 Summary

This chapter has introduced the three types of experiments to be conducted. Each experiment was designed to test a specific aspect of our proposed algorithms. First, the Alibaba cluster experiments provide us with insight into how our methods would behave for real historical workloads. Next, the synthetic experiments allow us to understand how the change in noise and seasonality affects algorithm performance. Lastly, the real-time tests demonstrate how a real Kubernetes system responds to predictive autoscaling.

Chapter 5

Results

We now present the test results on the experiments from Chapter 4 for both prediction algorithms from Sections 3.1, 3.2, and the predictive autoscaling algorithm from Section 3.3. We quantify the performance by considering three main metrics for all workloads: average CPU slack, percentage observations with insufficient requested CPU, and the amount of insufficient CPU for these observations. Results show that the proposed strategies can generate predictions, which the autoscaling algorithm uses to make scaling decisions that reduce slack and insufficient CPU.

5.1 Alibaba cluster trace tests

5.1.1 Container c_1

As seen in Figure 5.1, the container c_1 workload displays both daily seasonality and irregularity in CPU usage. We compare the estimated recommendation target of VPA (green), and the prediction targets of Holt-Winters (red) and LSTM (blue). Note that the first two seasons are omitted, as we only start generating predictions from the third season, see Section 3.1. We also show the prediction bounds computed by HW and LSTM while we show the requested allocated CPU millicores by the three different autoscaling techniques in Figure 5.2. We remind the reader that the "requested" CPU value is re-scaled to the predicted target value, whenever the actual CPU usage goes above (below) the computed upper (lower) bound. We report the average CPU slack, insufficient CPU observations, and amount of insufficient CPU in Table 5.1.

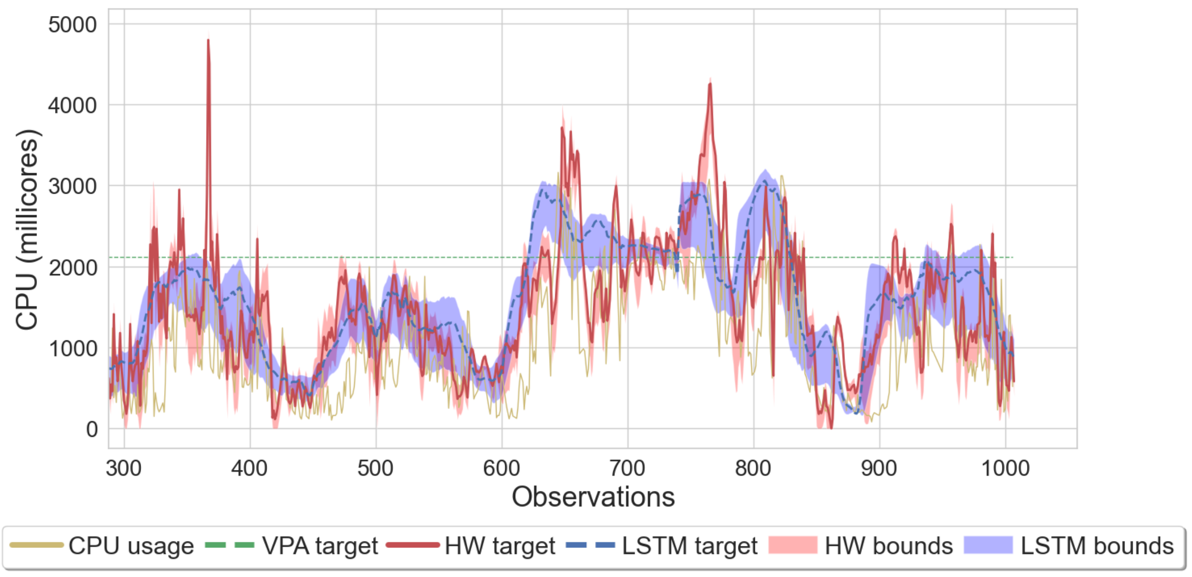


Figure 5.1 – Alibaba, c_1, prediction targets and bounds

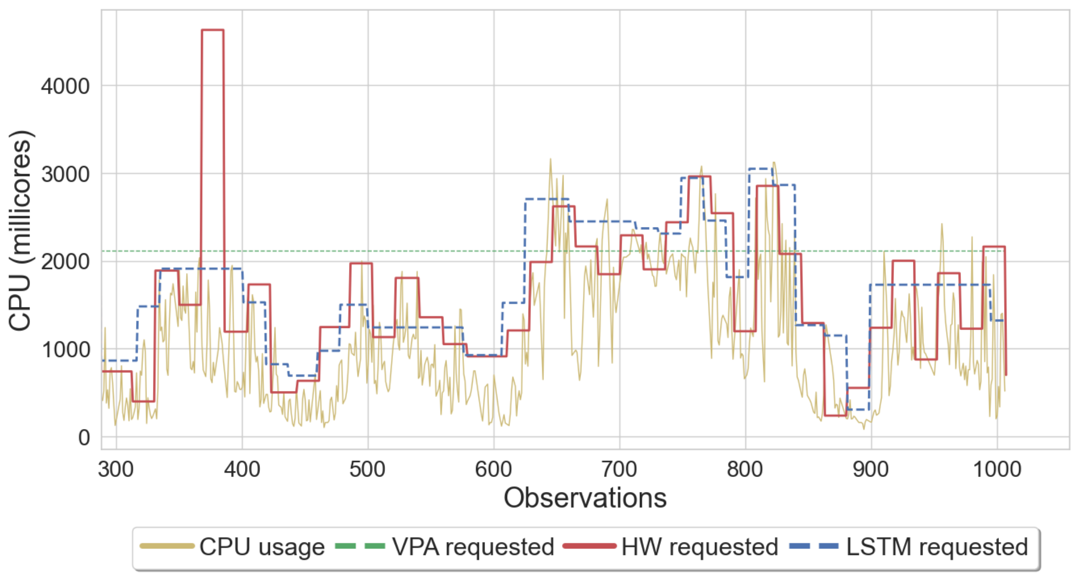


Figure 5.2 – Alibaba, c_1, scaling requested CPU

	Avg. slack (millicores)	**Insufficient CPU** (% observations)	**Insufficient CPU** (total millicores)
VPA	1,042	8.9	23,674
HW	533	18.5	43,790
LSTM	627	7.9	12,457

Table 5.1 – Performance summary for container c_1.

VPA does not adapt to dynamic workloads. We first observe that the estimated VPA target in Figure 5.1 is constant at around 2000 millicores despite the load showing some degrees of seasonality. We can see in Table 5.1 that the estimated VPA target achieves a relatively low ratio (8.9%) of insufficient CPU observations at the cost of a high average slack at 1042.7 millicores.

HW performs poorly due to irregular seasonality. Due to the irregularity in daily seasonality, HW generates a target prediction that often fluctuates aggressively. These sudden changes in the predicted values result sometimes in highly inaccurate scaling decisions, as shown in Figure 5.2 at for example X =390. We also see in Table 5.1 that although HW can achieve around 50% lower average slack than VPA, it has the highest percentage of insufficient CPU request observations at 18.5%.

LSTM learns to proactively scale, minimizing CPU insufficiency. In contrast to HW, LSTM has relatively wider prediction bounds, which gives the CPU usage more room for movement without triggering a re-scale. This makes the LSTM-based autoscaling strategy less reactive to smaller changes in the workload, which could be useful in avoiding unnecessary re-scaling. Indeed, Figure 5.2 shows that re-scales happen less frequently using the predictions from LSTM compared to those from HW.

LSTM predictions and scaling decisions demonstrate robustness. Figure 5.1 also shows that LSTM has a smoother prediction target curve, leading to less erratic scaling decisions. The robustness of LSTM is also reflected in Table 5.1. Not only does it have the lowest percentage of insufficient CPU observations, but it also has the lowest total amount of insufficient CPU at 12457 millicores (around 48% lower than VPA, 72% lower than HW), while saving around 40% slack compared to VPA. Compared to HW, LSTM has significantly fewer insufficient CPU observations and total insufficient millicores value, while only having a 15% higher slack.

Predictions are working as intended. Table 5.1 also shows that both HW and LSTM manage (to a certain extent) to predict future CPU usage, by increasing predictions before each uphill. As a result, as seen in Figure 5.2, the autoscaling algorithm manages to scale up preemptively thus avoiding insufficient CPU requests. We can also see that the predictions allow us to scale down promptly whenever the workload enters a calmer period.

5.1.2 Container c_10235

We now look at container c_10235 from the Alibaba trace, showing the predictions and scaling decisions in Figure 5.3 and Figure 5.4, respectively, and the corresponding performance summary in Table 5.2. We can see that the workload of this container displays much stabler daily seasonality, which allows for more accurate predictions from both LSTM and HW.

LSTM has overall best performance even with more regular seasonal patterns. The increase in prediction accuracy is reflected in Table 5.2, where both HW and LSTM manage to lower their % insufficient observations below that of VPA. Once again, LSTM stands out with the robustness of its predictions, being able to decrease the percentage of observations with insufficient CPU by over 50% when compared to the other methods. The last column of the table also indicates that LSTM can lower the total amount of insufficient millicores by 42% and 70% when compared to HW and VPA respectively.

Stable seasonality allows predictive autoscaling to lower slack effectively. Figure 5.4 shows how both HW and LSTM can decrease slack by scaling down when the workload enters its less intensive period, and scale up preemptively to avoid insufficient CPU requests during the peaks. Both predictive autoscaling methods manage to reduce average slack by around 40% when compared to VPA. Similar to before, HW achieved 7% lower slack than LSTM, at the cost of having more insufficient CPU.

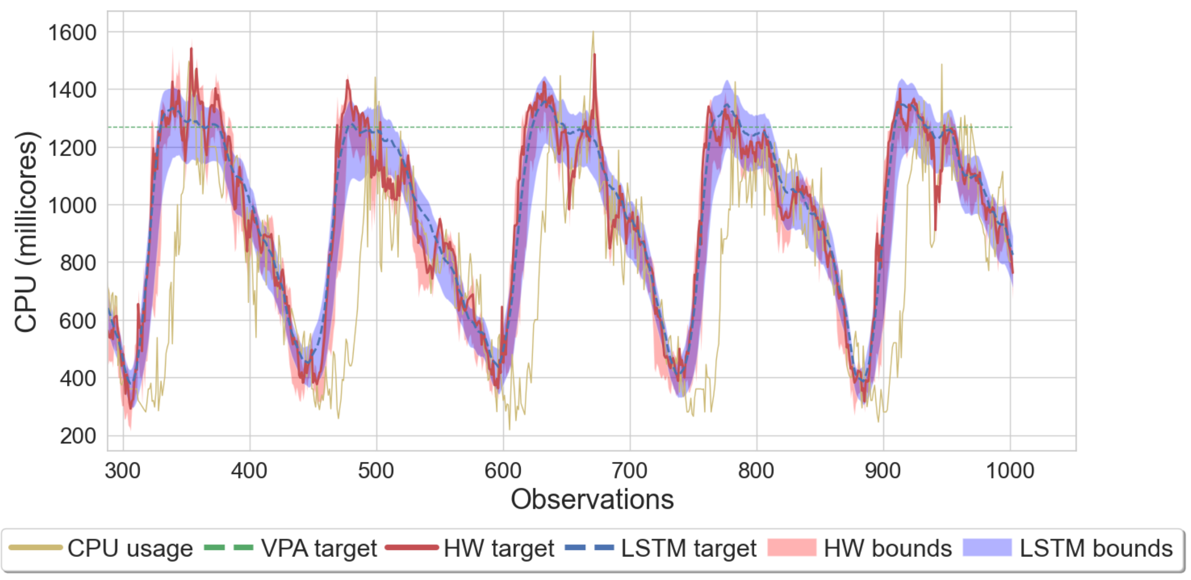


Figure 5.3 – Alibaba, c_10235, prediction targets and bounds

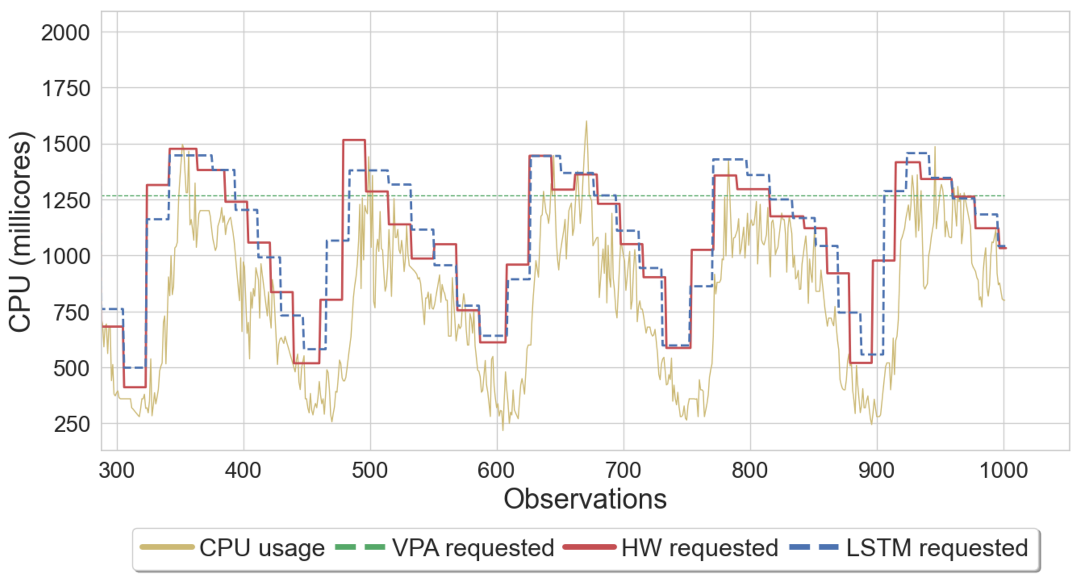


Figure 5.4 – Alibaba, c_10235, scaling requested CPU

	Avg. slack (millicores)	**Insufficient CPU** (% observations)	**Insufficient CPU** (total millicores)
VPA	468	5.3	3,558
HW	271	5.1	1,846
LSTM	291	2.3	1,075

Table 5.2 – Performance summary for container c_10235.

5.2 Synthetic test results

The synthetic test results give us a better understanding of how noise and seasonality intensity affects the proposed autoscaling strategies.

Starting with alpha = 0.1, the CPU usage data consists of almost pure noise. As alpha increases, noise diminishes and the seasonality of the sine curve intensifies gradually, see Figure 5.5 and 5.7. Just as in the Alibaba cluster tests, the predictions generated by the HW and LSTM prediction algorithms allow the autoscaler to proactively adjust the requested CPU, see Figure 5.6 and 5.8.

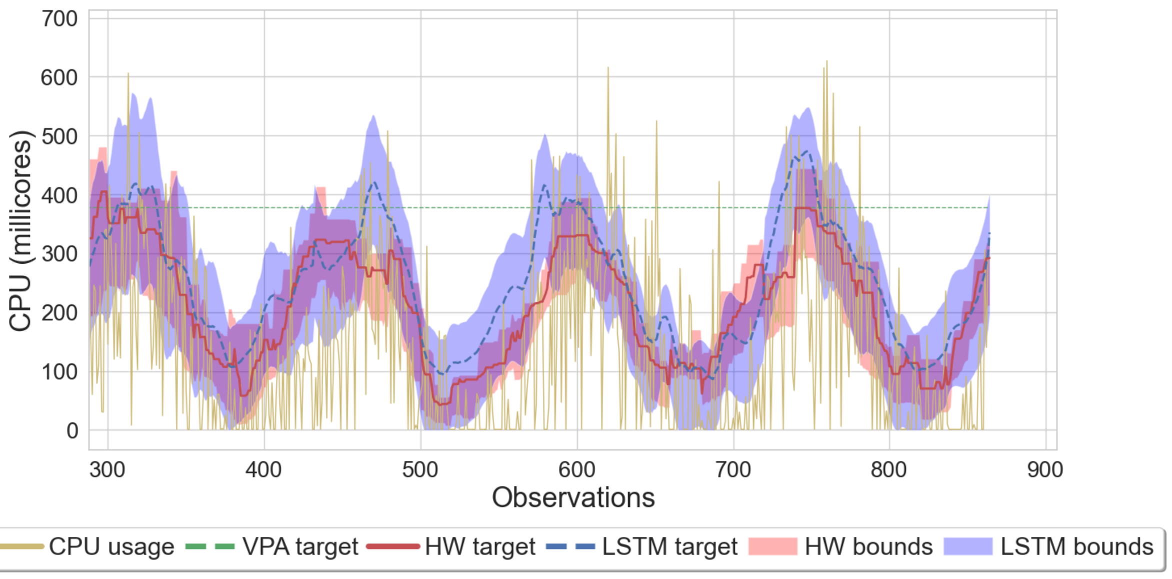


Figure 5.5 – Synthetic, alpha = 0.5, prediction targets and bounds

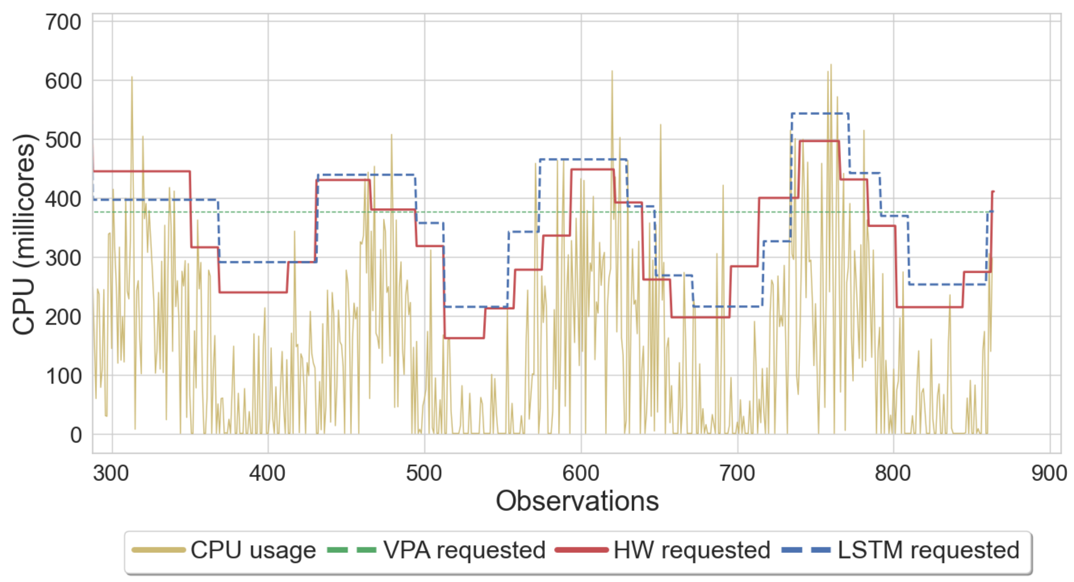


Figure 5.6 – Synthetic, alpha = 0.5, scaling requested CPU

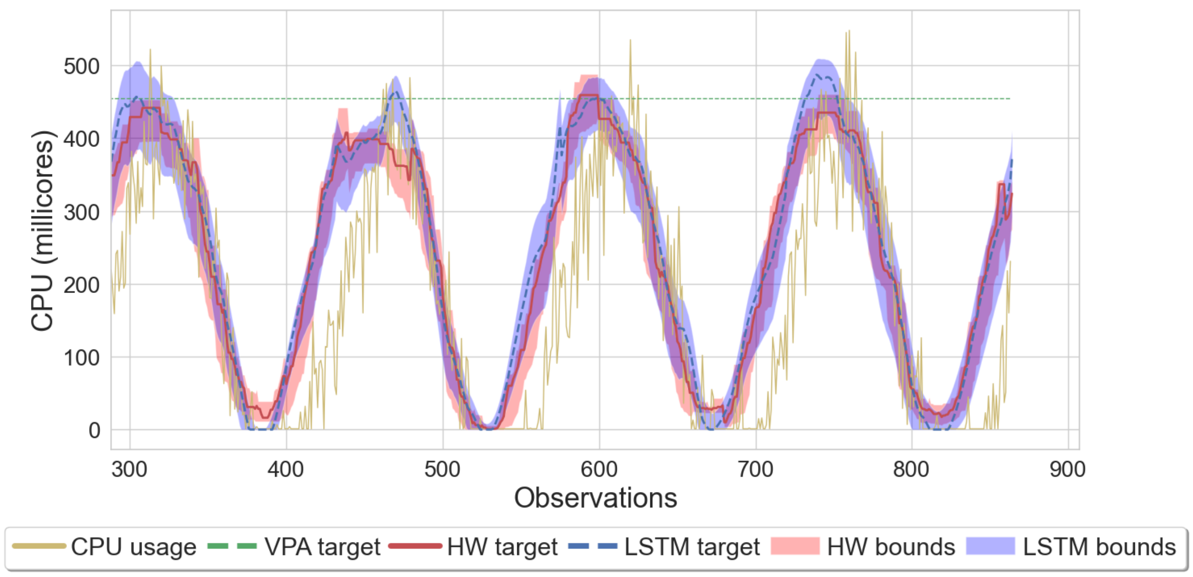


Figure 5.7 – Synthetic, alpha = 0.8, prediction targets and bounds

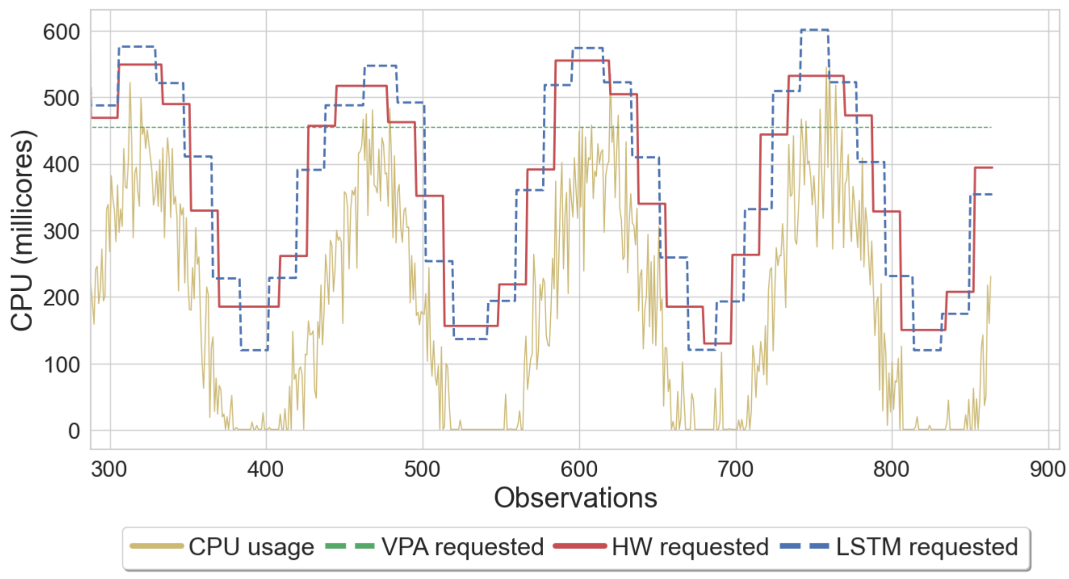


Figure 5.8 – Synthetic, alpha = 0.8, scaling requested CPU

Figure 5.9 shows the amount of slack generated by VPA, HW, and LSTM for different values of alpha using box plots where the whiskers indicate the 5th and 95th percentiles.

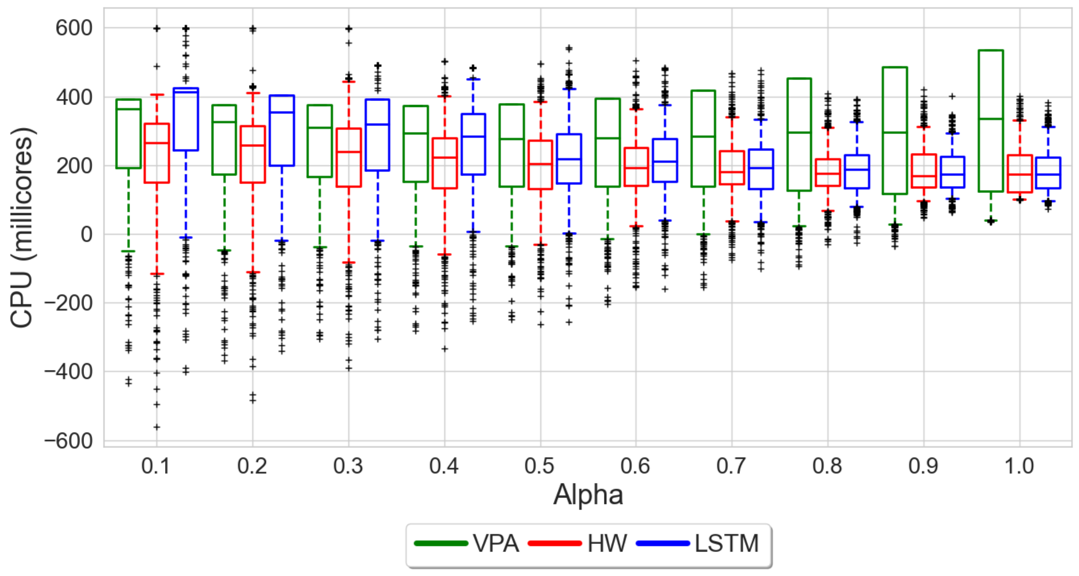


Figure 5.9 – Synthetic, alpha = 0.1 to 1.0 (lower value indicates more noise and less seasonality), CPU slack

Predictive autoscaling reduces slack. VPA has almost consistently higher overall slack for the tested alpha values, especially for alpha = 0.4 and above. For these alpha values, both predictive autoscaling methods can consistently achieve around 30 to 40% lower slack compared to VPA, as seen in Figure 5.9. These improvements are in line with what we observed for the Alibaba cluster tests.

The predictive methods perform better with clearer seasonality. Starting from alpha = 0.4, the average slack and spread of slack values for VPA also increase as alpha grows larger. The opposite can however be seen for HW and LSTM. This indicates that the stronger seasonality and weaker noise, the more resources we can save with a proactive approach and the more resources we waste by using the VPA approach.

HW is more easily affected by noise and relies on seasonality to perform well. HW and LSTM have similar performances in terms of slack until alpha drops below 0.4. At lower alpha values, we observe that LSTM starts to behave similarly to the estimated VPA target, with almost a flat prediction target, barely making any re-scales. However, similar to the container c_1 from the Alibaba cluster trace, we notice that the predictions of HW fluctuate far more, see Figure 5.10. Additionally, narrow prediction bounds make it easier to trigger re-scales at the cost of having a higher percentage of insufficient observations, as shown in Figure 5.11 and 5.12.

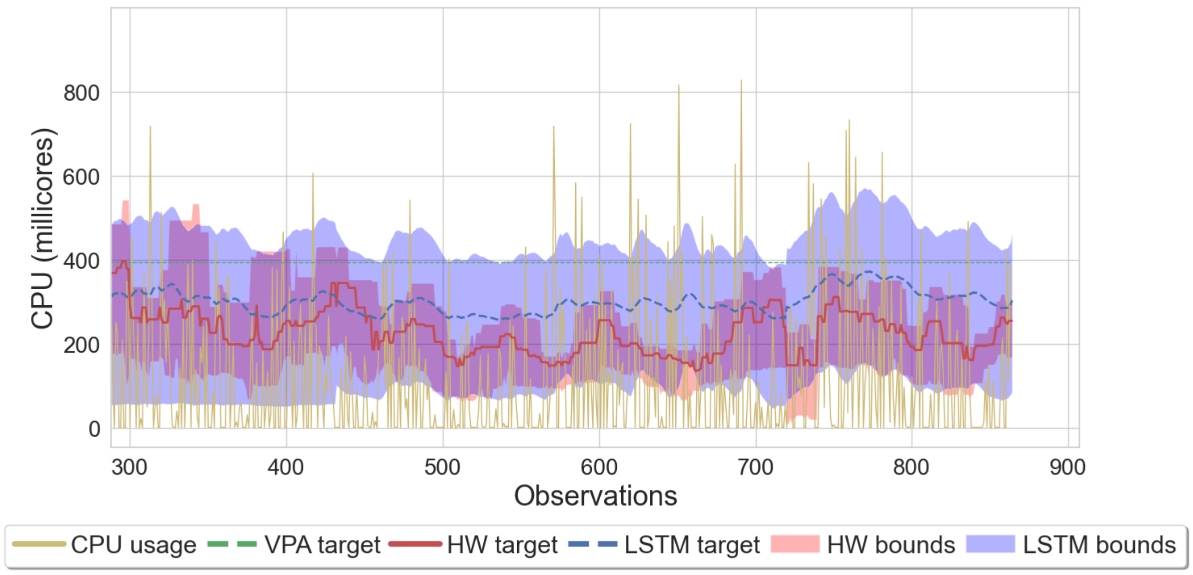


Figure 5.10 – Synthetic, alpha = 0.1, prediction targets and bounds

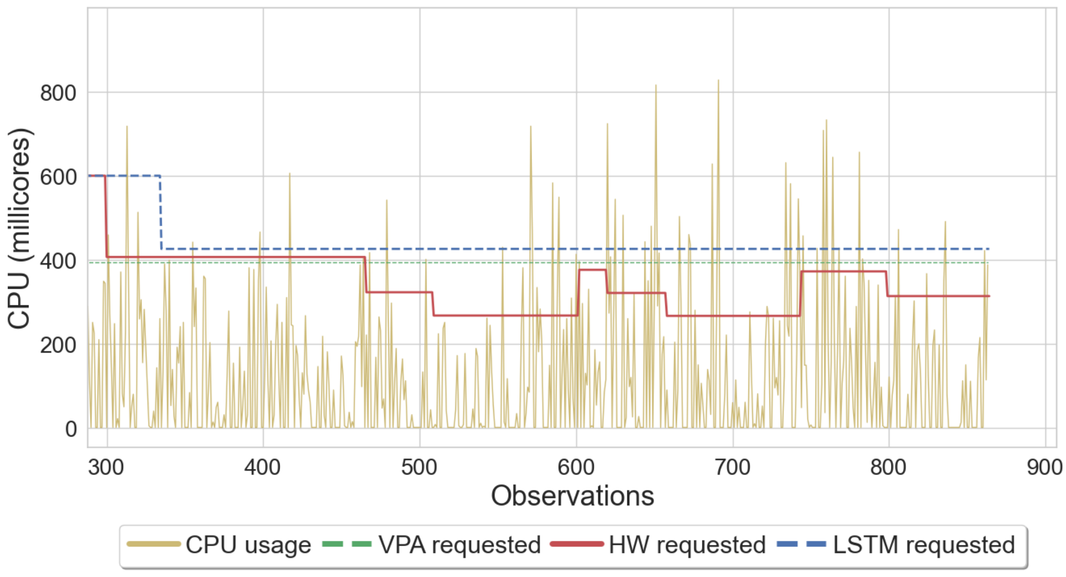


Figure 5.11 – Synthetic, alpha = 0.1, scaling requested CPU

LSTM-based predictive autoscaling demonstrates robustness. LSTM manages to maintain the lowest percentage of insufficient CPU requests for all alpha while keeping a relatively low slack. For alpha = 0.4 and below, it can achieve more than 40% fewer insufficient observations compared to HW, see Figure 5.12. Moreover, looking at Figure 5.13, we can see that the spread of insufficient CPU values for each alpha value does not differ much between the three methods. Once again, the whiskers indicate the 5th and 95th percentiles. In general, HW seems to have the most extreme values, which might be an indication that it is more prone to suffering from CPU throttling.

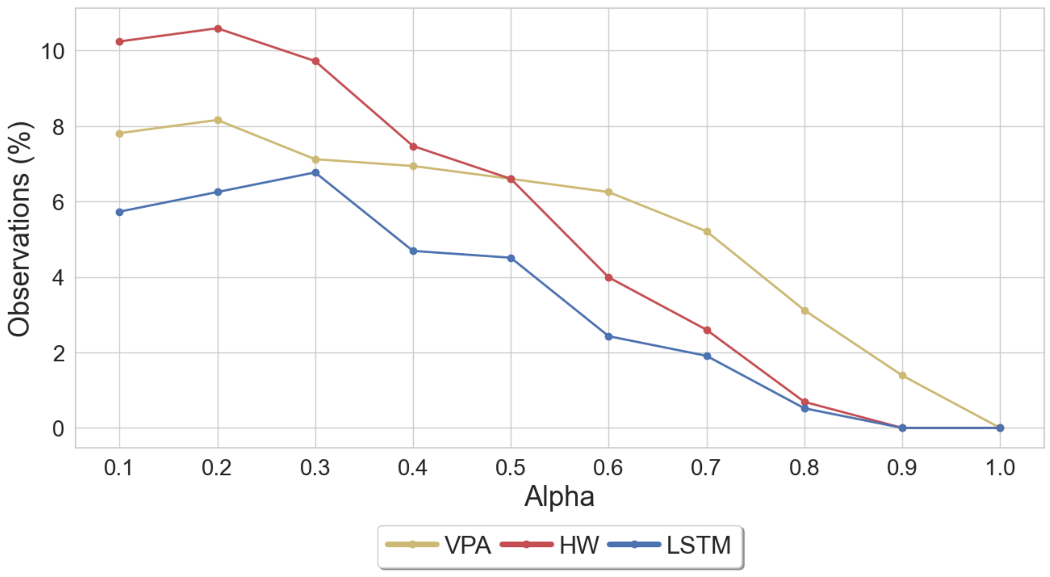


Figure 5.12 – Synthetic, alpha = 0.1 to 1.0 (lower value indicates more noise and less seasonality), % observations with insufficient CPU

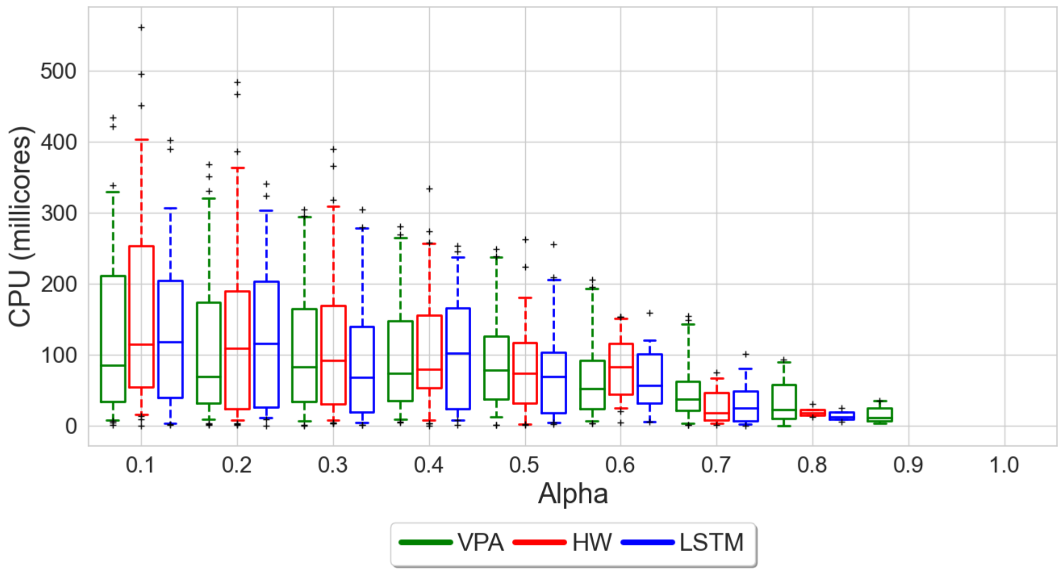


Figure 5.13 – Synthetic, alpha = 0.1 to 1.0 (lower value indicates more noise and less seasonality), insufficient CPU amount

5.3 Real-time test results

We observe similar results running the real-time experiments in our Kubernetes cluster. The workload that we are simulating with the `NGINX` container and `Slowcooker` setup displays strong seasonality, similar to the synthetic tests with high alpha values. As we saw previously in Figure 5.9, for workloads with strong seasonality, HW and LSTM have similar overall performance.

Instead of estimating the VPA target, we now receive the VPA target directly from the VPA Recommender component. As seen in Figures 5.14 and 5.15, the VPA target's behavior is very similar to our estimations, keeping a nearly constant high value. We can see the VPA target fluctuates slightly at the beginning of each new season. This is due to the decay of the weights corresponding to the higher CPU usage values of the workload, which happens during the resting phase of each season. When the new season begins, however, the decay stops, and the VPA target is restored to its initial value.

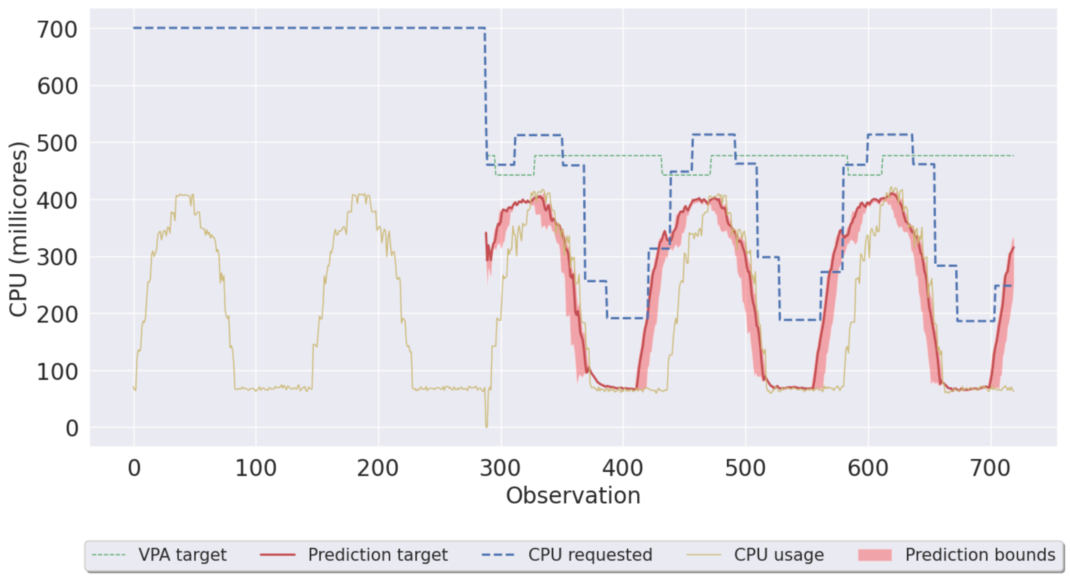


Figure 5.14 – Real-time, HW prediction and scaling

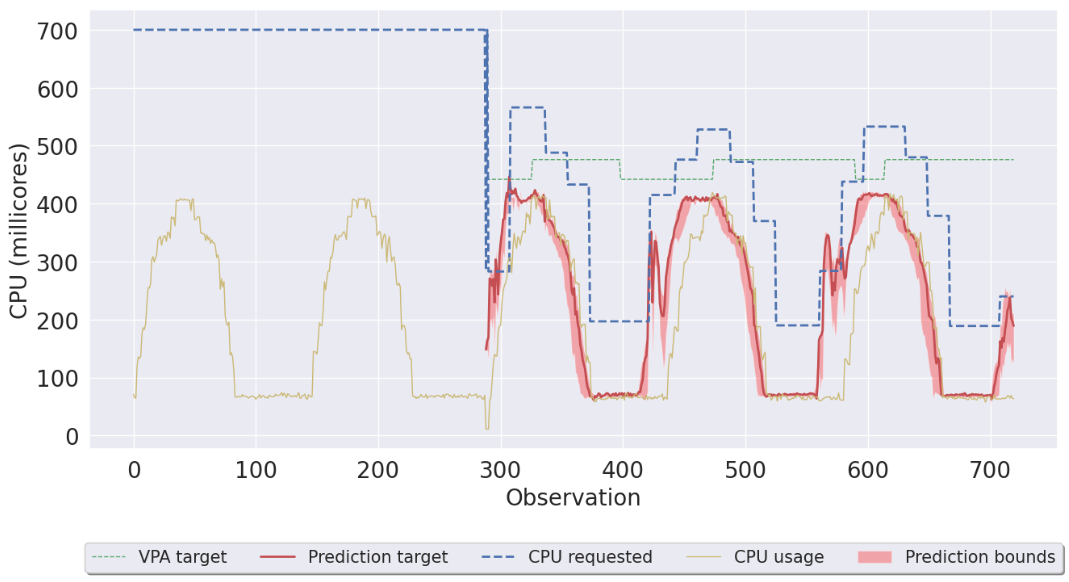


Figure 5.15 – Real-time, LSTM prediction and scaling

During the real-time tests, we notice that upon a re-scale operation, `NGINX` actually displayed a few milliseconds of downtime, waiting for the new pod to start up. This caused a temporary loss of around 0.7‰ requests over a period of 10 seconds. Having more than a single pod replica could potentially avoid disruptions.

Moreover, throughout all tests, it was noticed that it took significantly longer to re-train the LSTM models, compared to re-fitting the HW models. This is due to the computational complexity of the LSTM algorithm and the fact that no GPU was used. In general, re-training one of our LSTM models required around 10 seconds depending on the size of the training set, while it took less than 1 second to re-fit the HW model using the same data-set. However, the LSTM model is only re-trained every new season (trained only once for the Alibaba cluster tests), while the HW model is re-fitted every new time-step. This frequent re-fit is necessary for the HW model, as it otherwise has no way to take into consideration new observations. LSTM does not need to be re-trained as often, as it accepts new observations as input by default.

In contrast to training, generating predictions with the models took much less time. This could done in a few milliseconds for both the LSTM and HW models.

Also, recall that the *step_in* parameter for the LSTM algorithm was reduced from 96 to 48 for the real-time tests. The *step_in* parameter controls the length of each input feature to the network, which directly determines the amount of

LSTM units present in each layer. Increasing this parameter thus increases the number of trainable parameters of the LSTM model, which may lead to increased training time. With *step_in* = 96, some predictions took more than 15 seconds to generate, which caused overlapping prediction samples. Having a smaller *step_in* value, however, may cause certain longer spanning patterns to be unrecognizable for the LSTM. This is illustrated in Figure 5.15. The low workload intensity period spans longer than 48 time-steps, which causes inaccurate predictions at observations $X = 420$ and $X = 570$.

5.4 Summary

In this chapter, we have compared the performance of the proposed autoscaling strategies against that of VPA. In general, for all tests, our algorithms consistently achieved lower slack and reduced insufficient CPU. The LSTM-based prediction strategy demonstrated particularly promising results.

Chapter 6

Discussion

Up until now, we have demonstrated the promising aspects of predictive autoscaling, and we have been focusing on the potential performance improvements it is possible to gain by using such a strategy. However, there are still many issues that we need to consider. This chapter discusses our results along with some of the limitations, weaknesses, and problems faced by predictive autoscaling. We start by presenting the answers to our research questions.

6.1 Answers to research questions

Based on the test results, for workloads that demonstrate stronger seasonality, we can expect a reduction in slack up to around 30 to 40% by using the proposed predictive autoscaling strategy instead of the current Kubernetes VPA. Although we can achieve the highest decrease in slack by basing predictions on HW, LSTM demonstrates higher robustness and can achieve a similar decrease in slack while being able to maintain up to 40% fewer insufficient CPU observations.

However, our results also demonstrate that, for workloads that display weaker seasonality and higher noise, a predictive autoscaling strategy is unable to achieve much improvement in slack without causing an increase in insufficient CPU.

Furthermore, we have shown that a predictive autoscaling strategy is not necessarily more likely to cause CPU insufficiency, and in fact, can lower CPU insufficiency in many cases.

6.2 Results discussion

In the previous chapter, we have seen how time-series analysis based on both HW and LSTM can recognize patterns in historical CPU usage and use these patterns to predict future usage. We have also shown how these predictions can be used by an autoscaling algorithm to proactively scale CPU resources. Compared to a reactive method, we have shown how the proposed methods are capable of lowering CPU slack at the same time reducing insufficient CPU, for various seasonal workloads.

Another major finding was that the underlying prediction algorithm can greatly affect autoscaling performance. Although the amount of slack decreased was nearly the same, the LSTM-based algorithm was much better at avoiding insufficient CPU and is perhaps more suitable than HW for making predictions.

Fundamentally, LSTM and HW use widely different strategies to generate predictions. HW relies heavily on seasonality and as we have seen, irregular changes in seasonality between seasons can cause predictions to fluctuate greatly. As the seasonal component of HW is calculated by exponential smoothing over past seasonal components exactly "season length" steps back in time, HW can be greatly affected by changes in season length. For example, if a season shifts and starts a few time-steps earlier or later, predictions could suddenly become erratic. This could perhaps be the explanation for the fluctuations in the predictions seen in the Alibaba cluster test results, see Section 5.1. Contrary to this, the synthetic tests had seasons that started perfectly on time, which could be why we did not see the same kind of instability. Furthermore, HW relies on exponential smoothing which always treats more recent historical data with higher significance. HW has no way of adjusting for seasonal patterns that appear across seasons, for example, only every two seasons.

On the other hand, predictions based on LSTM do not rely on seasonality in the same way. LSTM does not care about where in a season we are when making a prediction. Instead of relying on fixed seasonal components, LSTM attempts to capture patterns in historical usage, wherever they might appear. Because of this, predictions based on LSTM are much less vulnerable to inconsistencies in seasonality, as seen with the Alibaba cluster tests in Section 5.1 and the high noise synthetic tests in Section 5.2.

However, there are cases where HW manages to generate more stable predictions than LSTM, as we saw with the real-time test in Section 5.3. To generate each prediction, LSTM relies on an input vector of length *step_in*, as described in 3.2. If this length is too short, the LSTM network will have difficulties with differentiating between different patterns that have overlapping characteristics. This may cause fluctuations in the predictions as we saw with the real-time test results.

6.3 Threats to validity

6.3.1 Historical data

One of the weaknesses of our predictive autoscaling strategies is that a certain amount of historical usage needs to be collected before we can start scaling. Depending on the seasonality and patterns present in the workload, collecting this data could take days, weeks, or even longer. Furthermore, before we have collected this data, it is difficult to know even if these patterns exist in the first place. Fundamentally, a predictive strategy is only able to achieve good performance if there are repeating patterns in the workload.

A reactive method such as VPA may not provide optimal performance, however, it can provide recommendations and autoscale already from the very start. VPA also does not rely on repeating patterns. For this reason, predictive autoscaling would perhaps be more suitable for established workloads with a recorded resource usage history.

For the scope of this research, we have not further explored the amount of historical data needed for various workloads, to effectively employ a predictive autoscaling strategy. In all experiments, we have assumed that enough historical data is available that allows our algorithms to at least identify some useful patterns, which they can use to their advantage. Further analysis is required to understand how the amount of historical data affects the viability of the proposed strategy.

6.3.2 Computational complexity

Another issue is increased computational complexity. Generating predictions using neural networks can be computationally expensive. Depending on the workload and the model that is being used, we might end up using more resources to generate predictions than what we are saving by reducing slack. For workloads where there is no potential for large improvement, VPA or other strategies should perhaps be considered first.

For the scope of this research, we have not considered the amount of computational resources required to run the tested autoscaling strategies. Also, we have not thoroughly optimized any of the algorithms. Moreover, we have limited ourselves to using CPU to execute the LSTM-based algorithm. Having a

GPU would potentially decrease the training and execution time of the algorithm significantly.

To understand the viability of predictive autoscaling for real use-cases, one would need to further optimize and analyze the performance of the underlying algorithms.

6.3.3 Advanced seasonal patterns

In this research, we have only been considering daily seasonality. For workloads that contain other seasonal patterns such as weekly seasonality, or multiple seasonal patterns, the presented algorithms would need to be tuned or extended accordingly.

The basic HW algorithm is built to recognize only a single seasonal component. If we were working with a workload that has both daily and weekly seasonality, we would need to extend the model to be able to handle multiple seasonal patterns. Another alternative would be to focus on a single seasonal component, such as the weekly one.

For patterns that span over a longer time duration, some of the parameters of the LSTM-based prediction algorithm would need to be adjusted. For example, the *step_in* parameter needs to be large enough for the model to be able to distinguish between different usage patterns. Note that increasing the number of parameters could have an additional impact on computational complexity.

To effectively apply the proposed autoscaling strategy to real use-cases, more advanced seasonal patterns would potentially need to be considered.

6.3.4 Pod restarts

At the end of the last Chapter, we only briefly mentioned the effects of restarting a pod to rescale its resource requests. To effectively utilize our proposed autoscaling strategy or any other vertical autoscaler in Kubernetes, we have to make sure the running application can handle restarts with minimal disruption.

Currently, changing container resource requests in Kubernetes requires restarting the pod. This restart can be undesirable for certain stateful applications, as moving or copying existing state information could be expensive or unfeasible depending on the underlying implementation. Service state information may become temporarily inaccessible, and users trying to access the service may encounter non-trivial delays before getting back to expected performance. This was an issue that was also identified during our real-time tests. Also, excessive scaling might generate excessive load on Kubernetes components that are responsible for handling the pod restart.

As we have seen from the test results, the LSTM-based predictive autoscaling strategy manages to achieve similar or superior performance in terms of slack and insufficient CPU when compared to VPA. However, more frequent re-scaling is inevitable as we try to adapt to the workload in a fine-grained manner. We have, however, implemented parameters that can limit the frequency of re-scaling and avoid unnecessary restarts, as described in Section 3.3. By tuning these parameters, it is potentially possible to optimize the algorithms for the constraints on re-scaling frequency.

As such, to effectively apply predictive autoscaling in real Kubernetes applications, further research will be needed to understand how the frequency of restarts can affect pods.

6.4 Future work

Currently, there is ongoing work for in-place updates that aims to make restarts unnecessary when modifying pod resource requests [43]. The issue of pod restarts is one of the major drawbacks to predictive autoscaling and Kubernetes pod autoscaling in general. If in-place resource updates became possible, our proposed autoscaling strategies would also become more viable.

Another interesting continuation would be to further optimize the LSTM network used for generating predictions. Both tuning parameters and model modifications such as adding drop-out, peep-hole connections, etc., could potentially yield better performance.

Additionally, it would also be interesting to investigate the combination of reactive and proactive scaling strategies, and shifting between strategies depending on the situation. By combining strategies, it could potentially be possible to avoid some of the weaknesses of predictive autoscaling, such as the initial need for long historical data.

6.5 Summary

Our experiments demonstrated the viability of predictive autoscaling for various synthetic and real-life workloads. We have seen how LSTM and HW generate predictions in vastly different ways, and how both methods have their strengths and weaknesses. We have further discussed how predictive autoscaling can be reliant on historical data and is potentially limited by other factors such as computational complexity and pod restarts.

Chapter 7

Conclusions

Our work shows that for dynamic real-world workloads, LSTM neural networks reduce the wasted CPU resources by a factor of 40% compared to traditional VPA without making the applications suffer more from sudden CPU resource limitations. Our LSTM-based autoscaler is more robust than exponentially smoothing techniques such as HW, which suffers from irregularity in the seasonal patterns.

We have demonstrated how Kubernetes VPA uses a strategy that generates excessive CPU slack for many seasonal workloads, which can be avoided by using our proposed prediction and autoscaling methods. Moreover, we also show how time-series analysis using HW and LSTM allows us to identify reoccurring patterns in historical CPU usage and use these patterns to predict future usage. These predictions enabled our autoscaling algorithm to scale proactively, not only managing to decrease slack but also minimize CPU insufficiency.

Further, HW predictions can be heavily affected by irregularities in seasonality, while LSTM predictions demonstrate robustness and manage to adapt well even to workloads with high noise. As a result, LSTM could decrease slack up to around 40 %, while achieving a similar or even lower percentage of insufficient CPU compared to VPA. Although HW was able to achieve even slightly lower slack, its instability causes significantly more insufficient observations, especially for noisy workloads. Moreover, generating predictions using exponential smoothing and especially neural networks can be computationally costly. This is an important factor that we need to consider when attempting to apply predictive autoscaling to real use-cases.

References

[1] J. Shah and D. Dubaria, "Building modern clouds: using Docker, Kubernetes & Google Cloud platform," in *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 2019, pp. 0184–0189.

[2] X. Sun, C. Hu, R. Yang, P. Garraghan, T. Wo, J. Xu, J. Zhu, and C. Li, "Rose: Cluster resource scheduling via speculative over-subscription," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 949–960.

[3] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the third ACM symposium on cloud computing*, 2012, pp. 1–13.

[4] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards network-aware resource provisioning in Kubernetes for fog computing applications," in *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019, pp. 351–359.

[5] M. F. Aktaş, C. Wang, A. Youssef, and M. G. Steinder, "Resource profile advisor for containers in cognitive platform," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 506–506.

[6] F. Tahir, M. Abdullah, F. Bukhari, K. M. Almustafa, and W. Iqbal, "Online workload burst detection for efficient predictive autoscaling of applications," *IEEE Access*, vol. 8, pp. 73 730–73 745, 2020.

[7] L. Nashold and R. Krishnan, "Using LSTM and SARIMA models to forecast cluster CPU usage," *arXiv preprint arXiv:2007.08092*, 2020.

[8] D. Janardhanan and E. Barrett, "CPU workload forecasting of machines in data centers using LSTM recurrent neural networks and ARIMA models,"

in *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*, 2017. doi: 10.23919/ICITST.2017.8356346 pp. 55–60.

[9] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay, “Containers and virtual machines at scale: A comparative study,” in *Proceedings of the 17th International Middleware Conference*, 2016, pp. 1–13.

[10] K. Authors, “Kubernetes documentation,” Jan. 2021. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/

[11] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, “Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade,” *Queue*, vol. 14, no. 1, pp. 70–93, 2016.

[12] M. Siarkowicz, D. G. Qian Chenglong, and T. Junchen, “Metrics server FAQ,” Dec. 2020. [Online]. Available: https://github.com/kubernetes-sigs/metrics-server/blob/4fdff44db13551696f2ddcd5f6ed9c88cb859ecd/FAQ.md

[13] T. Bannister and G. Templeton, “Horizontal pod autoscaler,” Nov. 2020. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale

[14] B. Lach, K. G. Marcin Wielgus, and G. Templeton, “Vertical pod autoscaler readme,” Oct. 2020. [Online]. Available: https://github.com/kubernetes/autoscaler/blob/master/vertical-pod-autoscaler/README.md

[15] P. Goodwin *et al.*, “The Holt-Winters approach to exponential smoothing: 50 years old and going strong,” *Foresight*, vol. 19, no. 19, pp. 30–33, 2010.

[16] R. G. Brown, *Statistical forecasting for inventory control.* McGraw/Hill, 1959.

[17] P. R. Winters, “Forecasting sales by exponentially weighted moving averages,” *Management science*, vol. 6, no. 3, pp. 324–342, 1960.

[18] C. C. Holt, “Forecasting seasonals and trends by exponentially weighted moving averages,” *International journal of forecasting*, vol. 20, no. 1, pp. 5–10, 2004.

[19] R. J. Hyndman and G. Athanasopoulos, *Forecasting: principles and practice*. OTexts, 2018.

[20] C. Chatfield, "The Holt-Winters forecasting procedure," *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, vol. 27, no. 3, pp. 264–279, 1978.

[21] J. V. Segura and E. Vercher, "A spreadsheet modeling approach to the Holt-Winters optimal forecasting," *European Journal of Operational Research*, vol. 131, no. 2, pp. 375–388, 2001.

[22] S. Fernández, A. Graves, and J. Schmidhuber, "An application of recurrent neural networks to discriminative keyword spotting," in *International Conference on Artificial Neural Networks*. Springer, 2007, pp. 220–229.

[23] M. Sundermeyer, R. Schlüter, and H. Ney, "LSTM neural networks for language modeling," in *Thirteenth annual conference of the international speech communication association*, 2012.

[24] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 02, pp. 107–116, 1998.

[25] C. P. Rosé, R. Martínez-Maldonado, H. U. Hoppe, R. Luckin, M. Mavrikis, K. Porayska-Pomsta, B. McLaren, and B. Du Boulay, *Artificial Intelligence in Education: 19th International Conference, AIED 2018, London, UK, June 27–30, 2018, Proceedings, Part I*. Springer, 2018, vol. 10947.

[26] G. Galante, L. C. E. De Bona, A. R. Mury, B. Schulze, and R. da Rosa Righi, "An analysis of public clouds elasticity in the execution of scientific applications: a survey," *Journal of Grid Computing*, vol. 14, no. 2, pp. 193–216, 2016.

[27] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of grid computing*, vol. 12, no. 4, pp. 559–592, 2014.

[28] A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. V. Papadopoulos, B. Ghit, D. Epema, and A. Iosup, "An experimental performance evaluation of autoscaling policies for complex workflows," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 2017, pp. 75–86.

[29] V. Podolskiy, A. Jindal, and M. Gerndt, "IaaS reactive autoscaling performance challenges," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 954–957.

[30] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of Docker containers with Elasticdocker," in *2017 IEEE 10th international conference on cloud computing (CLOUD)*. IEEE, 2017, pp. 472–479.

[31] H. Najdataei, V. Gulisano, A. V. Papadopoulos, I. Walulya, M. Papatriantafilou, and P. Tsigas, "Performance modeling and vertical autoscaling of stream joins," *arXiv preprint arXiv:2005.04935*, 2020.

[32] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand *et al.*, "Autopilot: workload autoscaling at Google," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[33] K. Grygiel and M. Wielgus, "Vertical pod autoscaler: design proposal," Dec. 2018. [Online]. Available: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/autoscaling/vertical-pod-autoscaler.md

[34] D. Midigudla, "Performance analysis of the impact of vertical scaling on application containerized with docker: Kubernetes on Amazon Web Services-EC2," 2019.

[35] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, "Exploring potential for non-disruptive vertical auto scaling and resource estimation in Kubernetes," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 33–40.

[36] A. A. Shahin, "Using multiple seasonal Holt-Winters exponential smoothing to predict cloud resource provisioning," *arXiv preprint arXiv:1701.03296*, 2017.

[37] D. Janardhanan and E. Barrett, "CPU workload forecasting of machines in data centers using LSTM recurrent neural networks and ARIMA models," in *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*. IEEE, 2017, pp. 55–60.

[38] S. N. Rao, G. Shobha, S. Prabhu, and N. Deepamala, "Time series forecasting methods suitable for prediction of CPU usage," in *2019 4th International Conference on Computational Systems and Information Technology for Sustainable Solution (CSITSS)*, vol. 4. IEEE, 2019, pp. 1–5.

[39] S. Seabold and J. Perktold, "statsmodels: Econometric and statistical modeling with python," in *9th Python in Science Conference*, 2010.

[40] "Alibaba inc. 2018. Alibaba open cluster trace," https://github.com/alibaba/clusterdata, 2018.

[41] "NGINX web server (github)," Feb. 2020. [Online]. Available: https://github.com/nginx/nginx

[42] "Slowcooker project (github)," Feb. 2020. [Online]. Available: https://github.com/BuoyantIO/slow_cooker

[43] "KEP: in-place update of pod resources #686," Dec. 2018. [Online]. Available: https://github.com/kubernetes/enhancements/pull/686#

TRITA-EECS-EX-2021:104