Predicting CPU usage for proactive autoscaling

Thomas Wang
Ericsson AB

Simone Ferlin
Ericsson AB

Marco Chiesa
EECS and Digital Futures
KTH Royal Institute of Technology

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 resources to physical resources. If an application exceeds these values, it might be throttled or even terminated. As a consequence, requested values are often overestimated, resulting in poor resource utilization in the cloud infrastructure. Autoscaling is a technique used to overcome these problems.

We observed that Kubernetes Vertical Pod Autoscaler (VPA) might be using an autoscaling strategy that performs poorly on workloads that periodically change. Our experimental results show that compared to VPA, predictive methods based on Holt-Winters exponential smoothing (HW) and Long Short-Term Memory (LSTM) can decrease CPU slack 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.

1 Introduction

Private and public clouds such as Amazon Web Services or Microsoft Azure require users to specify requests for resources such as memory and CPU cores when deploying their applications. These values are generally not linked to the application's run-time requirements but serve more as an orientation for the application to be correctly mapped to physical resources available in the cloud infrastructure. For example, in Kubernetes, users are required to manually set CPU resource requests for their applications to guarantee 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. Therefore, users tend to overestimate resource requests, which results in poor overall utilization (below 50%) of physical resources [14, 19]. As such, improving resource utilization could significantly lower operating costs, for both users of cloud platforms and companies which host their own Kubernetes clusters.

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 autoscaling mechanism, to *proactively* increase CPU utilization efficiency while avoiding throttling. While LSTM short-term load prediction has shown to outperform season-based predictions [11, 12], these works do not *integrate* and *evaluate* the impact of the predictions into the operation of a proactive autoscaler, which is our core contribution. Moreover, we focus on CPU usage at the container level instead of the more coarse-grained cluster level.

We compare both algorithms to Kubernetes' default Vertical Pod Autoscaler (VPA). In our experiments, we use synthetic as well as real open-sourced cluster data to focus on analyzing vertical autoscaling (*i.e.*, adjusting resources requested for a single container) on CPU consumption, *i.e.*, the increase or reduction of CPU for an application instance in a Kubernetes cluster, where our results demonstrate the viability of proactive autoscaling. Results show that the proposed LSTM-based autoscaler reduces CPU waste by a factor of 2x without incurring more CPU throttling than the default over-provisioned Kubernetes approach.

2 Background on Kubernetes Autoscaling

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/rollbacks, etc. Next, we describe some Kubernetes components that are essential to understand our work.

2.0.1 Containers, Pods, and Nodes

Containers are the smallest unit in Kubernetes, packaging up code. Compared to VMs, containers are lightweight and do not require as much resources such as CPU, memory, and storage [17]. **Pods** are the most basic scheduling unit in Kubernetes, containing one or more containers. Kubernetes reserves a maximum amount of pre-configured resources to each pod and scales them in two possible ways: The Horizontal Pod Autoscaler (HPA) instantiates multiple replicas of the pod while the Vertical Pod Autoscaler (VPA) increases the resources (*e.g.*, CPU) assigned to a pod. **Nodes** are the virtual or physical machines where pods are deployed by Kubernetes. Each cluster has at least one master node and one or more worker nodes, see Figure 1. The master node manages the Kubernetes cluster, handling tasks such as scheduling

DOI: 10.1145/3437984.3458831

pods on the worker nodes, provisioning, controlling, and exposing API to the clients. In Figure 1, each worker node has a `Docker` container run-time to run containers. In each worker node, `Kubelet` is responsible for inspecting pod specifications, given by the Application Programming Interface (API) server in the master node, to ensure that pods run in a healthy state. `Kube-proxy` in each worker node is responsible for maintaining networking and exposing services to the outside.

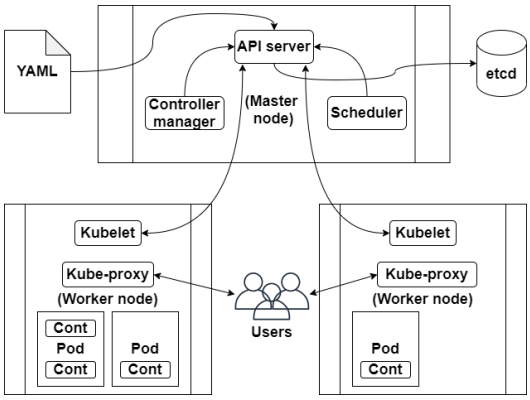

Figure 1. Basic architecture of a Kubernetes cluster

2.0.2 Resource requests and limits

Container CPU and memory requests and limits can be specified in the pod's deployment file, where CPU is given in millicores and memory in bytes. 1000 millicores is equivalent to 1 vCPU/Core for cloud providers and 1 hyperthread on bare-metal Intel processors. The Kubernetes `Scheduler` in Figure 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. If there are resources on the node, applications can consume them beyond this specified minimum, however, bounded the limit field. Decisions made by the `Scheduler` are based on the resource requests field and do not depend on the limits.

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

Figure 2. Container resource definition

The resource request field's purpose is to guarantee that applications have enough resources even under contention. Note that these values are estimates and not based on the application's run-time.

2.1 Kubernetes vertical pod autoscaler

The Vertical Pod Autoscaler (VPA) scales CPU and memory requests and limits of containers according to the measured loads. VPA contains three main components, which are introduced to the cluster as deployments. The relationships of these components are illustrated in figure 3.

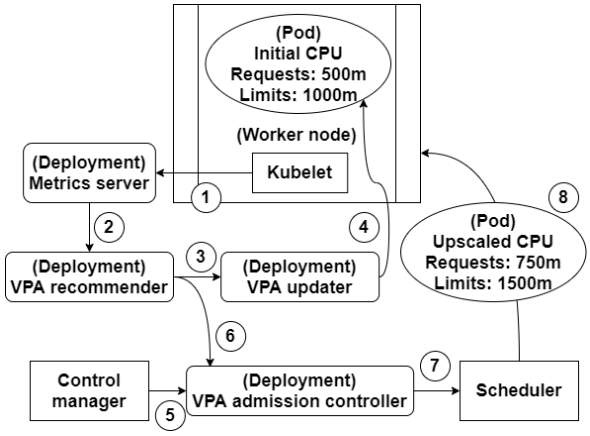

Figure 3. Vertical Pod Autoscaler (VPA): Architecture

The first component is the `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 3 (①) and (②).

Based on these values, the `Recommender` provides CPU and memory recommendations for the monitored containers (③). For each container, the recommendation implements a lower and upper bound, along with target values. The CPU recommendation target, lower, and upper bounds calculated by the VPA `Recommender` are based on a decaying histogram of weighted CPU samples collected once per minute. A default half-life value of 24 hours controls the speed at which sample weights decrease. The target value 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 second component is the `Updater` (④), which evicts a pod when the requested CPU goes over the upper or under the lower bounds. The `Updater` runs this check for all targeted pods every minute. Once evicted, it will be rescheduled by the cluster's `Control Manager` (⑤).

A new pod then passes through the `Admission Controller` component (⑥), which is responsible for updating its resource fields to the target value. If 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 (⑦ and 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`.

3 A Predictive and Dynamic Autoscaler

First, a prediction algorithm generates a prediction target with upper and lower bound values, which will be used by the autoscaling algorithm. To generate the predictions, we use HW exponential smoothing and LSTM, where both algorithms run with the container's CPU usage sampled at regular intervals. For each new sample, a new prediction is generated every 10 minutes.

Holt-Winters (HW) exponential smoothing. Exponential smoothing methods [6, 10, 21] use weighted averages of historical data to generate forecasts on time series with weights that decay exponentially. HW builds upon these methods, and further uses trend and seasonality in data to generate predictions. Exponential smoothing methods are known for their ability to generate reliable forecasts quickly and applicability to a large range of time series, which makes them favorable in many real-life applications. Compared to machine learning methods, HW has the advantage of being computationally inexpensive and is potentially easier to apply to real-world scenarios. The simplicity of the method however, could limit its ability to recognize more complex patterns. HW is for example unable to recognize multiple seasonal patterns.

We use `Statsmodels` [16] 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 generating predictions 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.

Long short-term memory (LSTM).

LSTM is a special type of artificial Recurrent Neural Network (RNN) suited for identifying patterns in sequential data. Generally, LSTM is used for classifying and making predictions based on time series data as there may be long delays between important events. Although there exist other suitable machine learning methods such as attention-based RNNs and Transformers, we chose LSTM for its relative simplicity and its history of success in a wide range of applications such as speech recognition, text-to-speech synthesis, machine translation. In this paper, our goal is to demonstrate that proactive workload prediction is possible using LSTM. As such, we leave more detailed model optimization and alternative machine learning methods as future work. Compared to the much simpler HW exponential smoothing model, we expect LSTM to be able to recognize a wider range of patterns, at the cost of being more computationally expensive.

LSTM network cells build upon the basic RNN cell by adding a cell-state along with gates regulating the information of the hidden states. This allows LSTM to avoid the vanishing gradient problem of RNNs and it helps to recognize reoccurring patterns.

We implement LSTM using `Keras` [1] 2.4.3 with two hidden layers. Similar to our HW model, the design of the LSTM was also determined using a heuristic approach. 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. 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.

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

3.1 Predictive autoscaling

Our proposed autoscaling algorithm (Alg. 1) 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 1: 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 ≤ 0 then
        if Abs(current requested - new_requested) > 50
          then
            Rescale to new_requested;
            Reset cool-down;
        end
    end
else
    Decrease cool-down;
end
```

4 Experimental Setup

We divide our experiments into three parts. First, we use the historical CPU usage of two containers from Alibaba's Open Cluster Trace 2018 [1] data-set to evaluate our algorithms. After that, we use synthetically generated CPU usage data outside of a Kubernetes cluster to assess the effects of varying seasonality and noise. Lastly, 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 [1]. 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 90'th percentile usage [9].

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 1, which models a sinusoidal load with a configurable amount of noise:

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

The *sine* 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. 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 as described in the following. 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 sine function. 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 deployed at Ericsson. 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.

`NGINX` deployment. The real-time experiments use the widely adopted `NGINX` web server [3]. This deployment contains a single pod with a single container, built using the nginx:1.18.0 Docker image. We rely on `Slowcooker`[4] to send periodic HTTP requests to the `NGINX` server. The load

on the NGNIX server is proportional to the number of requests from Slowcooker. 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.

4.4 Training the LSTM

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.

5 Evaluations

We now describe the results including both prediction algorithms from Section 3 and the experiments from Section 4. We quantify the performance by considering three main metrics for all the workloads: average CPU slack (*i.e.*, amount of requested CPU minus actually utilized CPU), percentage observations with insufficient requested CPU, and the total amount of insufficient CPU for these observations. Our results show that the proposed strategies can generate predictions, which allows the autoscaling algorithm to make scaling decisions reducing both slack and insufficient CPU.

5.1 Alibaba cluster trace tests

As seen in Figure 4, the container c_1 workload displays both seasonality and irregularity in CPU usage. We compare the predicted targets of VPA (green), 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. We also show the bounds computed by HW and LSTM while we show the requested allocated CPU millicores by the three different autoscaling techniques in Figure 5. We remind the reader that the "requested" CPU millicores depend on whether the actual CPU usage goes above (below) the computed upper (lower) bound, and it is then reset to the predicted target value. We report the average CPU slack,

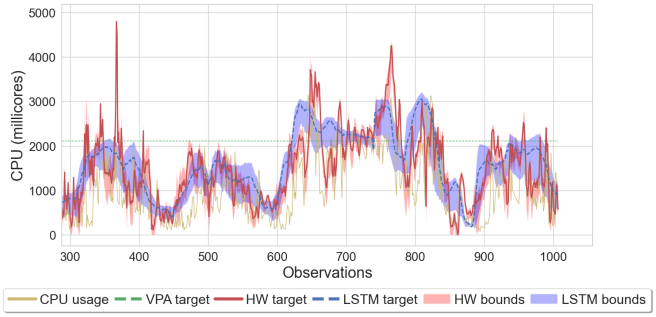

Figure 4. Alibaba, c_1, prediction targets and bounds

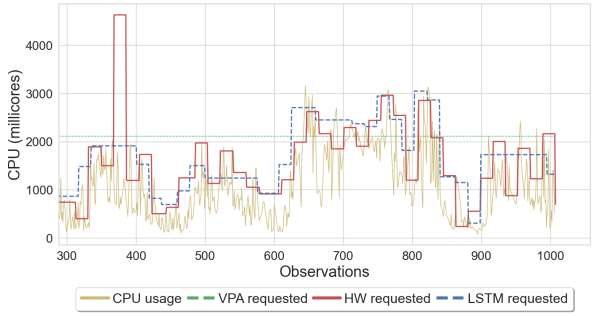

Figure 5. Alibaba, c_1, scaling requested CPU

	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 1. Performance summary for container c_1.

insufficient CPU observations, and amount of insufficient CPU in Table 1. We now discuss the two figures and the results in the table in detail.

VPA does not adapt to dynamic workloads. We first observe that the VPA target in Figure 4 is constant at around 2000 millicores despite the load showing some degrees of seasonality. We can see in Table 1 that the estimated VPA target achieves a relatively low 8.9 % 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 at for example X =390 and X =800. We also see in Table 1 that HW can achieve around 50% lower average slack than VPA, however, it has the highest percentage of insufficient CPU request observations (*i.e.*, 18.5%).

LSTM learns to proactively scale, minimizing CPU insufficiency. In contrast, LSTM has wider prediction bounds,

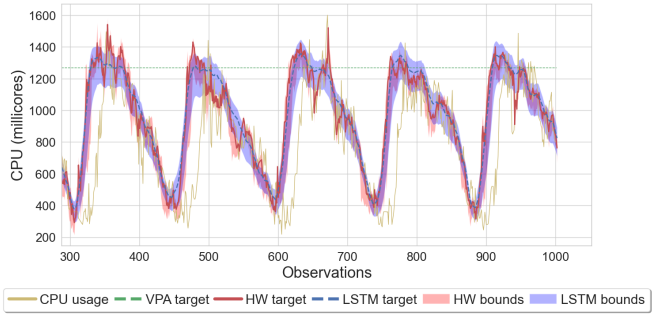

Figure 6. Alibaba, c_10235, prediction targets and bounds

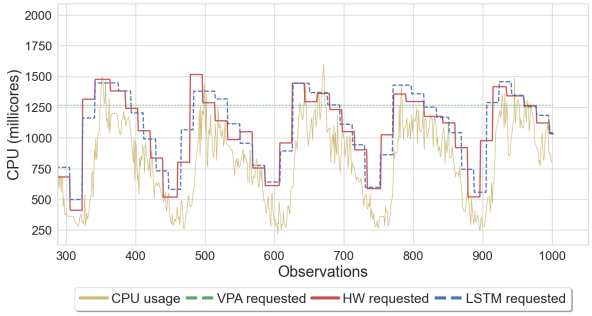

Figure 7. Alibaba, c_10235, scaling requested CPU

	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 2. Performance summary for container c_10235.

which makes it less likely to trigger a re-scale. This makes LSTM less reactive to smaller changes in CPU usage, which could be useful in avoiding unnecessary re-scaling. Indeed, Figure 5 shows that re-scales happen less frequently using the predictions from LSTM compared to HW.

Figure 4 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 1. Not only does it have the lowest % insufficient CPU observations, but it also has the lowest total amount of insufficient CPU at 12457 millicores while achieving superior slack savings compared to VPA. Compared to HW, LSTM has significantly fewer insufficient CPU observations and cumulative millicores value.

Predicting the future is key to avoid reactive scaling. Table 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 2, the autoscaling algorithm manages to scale up preemptively thus avoiding insufficient CPU requests.

LSTM outperforms HW even with more regular seasonal patterns. We now look at container c_10235 from the Alibaba trace, showing the predictions and scaling decisions in Figure 6 and Figure 7, respectively, and the corresponding summary in Table 2. The workload of this container displays much stabler seasonality, which allows for more accurate predictions for both LSTM and HW. This increase in accuracy is reflected in Table 2, where both HW and LSTM manage to lower their % insufficient observations below that of VPA. At the same time, the proposed methods also manage to reduce average slack by around 40%. Figure 7 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. Compared to HW, LSTM achieves over 50% lower insufficient CPU observations and the total amount of insufficient millicores is reduced by 42%.

5.2 Synthetic test results

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

Figure 8 shows the amount of slack achieved by VPA, HW, and LSTM for different values of α using a box plot where the whiskers indicate the 5th and 95th percentiles. We see that as α increases starting from 0.4, the average slack and spread of slack values increase for VPA. The opposite can however be seen for HW and LSTM. HW and LSTM have similar performances in terms of slack until α is at least 0.4. At lower α 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, as for the Alibaba trace, we notice that the predictions of HW fluctuate far more (not shown in the graph), triggering re-scales at the cost of having a higher % insufficient observations. LSTM manages to maintain the lowest % insufficient CPU requests for all α while keeping a relatively low slack, which once again demonstrates its robustness. For $\alpha = 0.4$ and below, it can achieve more than 30% fewer insufficient observations, see Figure 9.

5.3 Real-time test results

We observe similar results running the real-time experiments in our Kubernetes cluster. However, due to lack of space, we opt to not show them in the paper. 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 (our granularity). Having more than a single pod replica could potentially avoid any disruption. We also noticed that VPA's target behavior was very similar to our estimations.

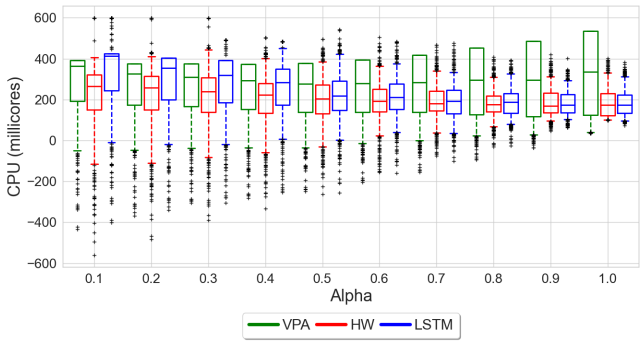

Figure 8. Synthetic, alpha = 0.1 to 1.0 (lower value indicates more noise and less seasonality), CPU slack

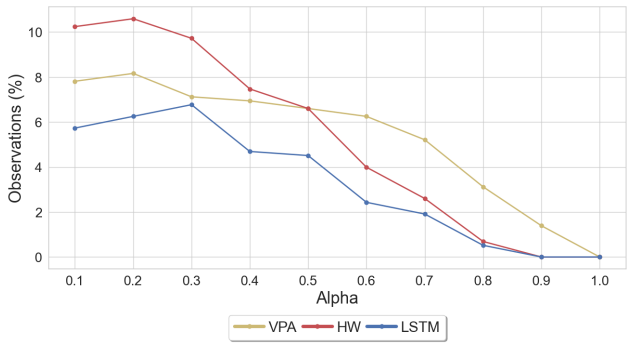

Figure 9. Synthetic, alpha = 0.1 to 1.0 (lower value indicates more noise and less seasonality), % observations with insufficient CPU

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.

6 Discussion and Future Work

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. There is on-going work for in-place updates of pod resources, that aims to make restarts unnecessary [2]. If in-place resource updates became possible, our proposed autoscaling strategies would also become more viable.

7 Related Work

Broad literature exists on general load predictions [7, 18]. Here, we focus on container-level predictions and vertical autoscaling and do not discuss horizontal autoscalers [13]. LSTM short-term load predictions have been shown to outperform season-based predictions [11, 12]. However, all 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 more coarse-grained cluster level.

A different line of research works [5, 8, 20] have looked at the scheduling problem in a cluster. Autopilot [15] is an autoscaling system built for Google's Borg system. Its vertical autoscaler manages to reduce memory slack from 46% to 23% compared to manually-managed jobs. 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. Borg uses a reactive autoscaling strategy that sets resource limits based on past historical usage, which is completely 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 SLA violations. We leave comparison with Autopilot as future work.

8 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.

References

[1] 2018. Alibaba Inc. 2018. Alibaba Open Cluster Trace. https://github.com/alibaba/clusterdata.

[2] 2018. KEP: in-place update of pod resources #686. https://github.com/kubernetes/enhancements/pull/686#

[3] 2020. NGINX Web Server (GitHub). https://github.com/nginx/nginx

[4] 2020. Slowcooker project (GitHub). https://github.com/BuoyantIO/slow_cooker

[5] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI) 14)*. 285–300.

[6] Robert Goodell Brown. 1959. *Statistical forecasting for inventory control*. McGraw/Hill.

[7] Sheng Di, Derrick Kondo, and Walfredo Cirne. 2012. Host load prediction in a Google compute cloud with a Bayesian model. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[8] Yuqi Fu, Shaolun Zhang, Jose Terrero, Ying Mao, Guangya Liu, Sheng Li, and Dingwen Tao. 2019. Progress-based container scheduling for short-lived applications in a Kubernetes cluster. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 278–287.

[9] Krzysztof Grygiel and Marcin Wielgus. 2018. Vertical Pod Autoscaler: design proposal. https://github.com/kubernetes/community/blob/master/contributors/design-proposals/autoscaling/vertical-pod-autoscaler.md

[10] Charles C Holt. 2004. Forecasting seasonals and trends by exponentially weighted moving averages. *International journal of forecasting* 20, 1 (2004), 5–10.

[11] D. Janardhanan and E. Barrett. 2017. 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)*. 55–60. https://doi.org/10.23919/ICITST.2017.8356346

[12] Langston Nashold and Rayan Krishnan. 2020. Using LSTM and SARIMA Models to Forecast Cluster CPU Usage. *arXiv preprint arXiv:2007.08092* (2020).

[13] Thanh-Tung Nguyen, Yu-Jin Yeom, Taehong Kim, Dae-Heon Park, and Sehan Kim. 2020. Horizontal pod autoscaling in Kubernetes for elastic container orchestration. *Sensors* 20, 16 (2020), 4621.

[14] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the third ACM symposium on cloud computing*. 1–13.

[15] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. 2020. Autopilot: workload autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[16] Skipper Seabold and Josef Perktold. 2010. statsmodels: Econometric and statistical modeling with python. In *9th Python in Science Conference*.

[17] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and YC Tay. 2016. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*. 1–13.

[18] Binbin Song, Yao Yu, Yu Zhou, Ziqiang Wang, and Sidan Du. 2018. Host load prediction with long short-term memory in cloud computing. *The Journal of Supercomputing* 74, 12 (2018), 6554–6568.

[19] Xiaoyang Sun, Chunming Hu, Renyu Yang, Peter Garraghan, Tianyu Wo, Jie Xu, Jianyong Zhu, and Chao Li. 2018. Rose: Cluster resource scheduling via speculative over-subscription. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 949–960.

[20] Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. 2012. DejaVu: Accelerating Resource Allocation in Virtualized Environments. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[21] Peter R Winters. 1960. Forecasting sales by exponentially weighted moving averages. *Management science* 6, 3 (1960), 324–342.