Machine Learning Based Adaptive Auto-scaling Policy for Resource Orchestration in Kubernetes Clusters

Chapter · January 2022

DOI: 10.1007/978-3-030-94507-7_1

4 authors, including:

Abhishek Dixit
Indian Institute of Technology Patna
1 PUBLICATION **3** CITATIONS
SEE PROFILE

Rohit Kumar Gupta
Indian Institute of Technology Patna
15 PUBLICATIONS **83** CITATIONS
SEE PROFILE

Rajiv Misra
Indian Institute of Technology Patna
106 PUBLICATIONS **1,326** CITATIONS
SEE PROFILE

Machine Learning Based Adaptive Auto-scaling Policy for Resource Orchestration in Kubernetes Clusters

Abhishek Dixit(✉), Rohit Kumar Gupta, Ankur Dubey, and Rajiv Misra

Indian Institute of Technology Patna, Dayalpur Daulatpur, India
{adixit,1821cs16,ankur.cs17,rajivm}@iitp.ac.in

Abstract. The wide availability of computing devices has cleared the path for the development of a new generation of containerized apps that can run in a distributed cloud environment. Furthermore, the dynamic nature of workload demands elastic application deployment that can adapt to any scenario. One of the most popular existing container orchestration systems, Kubernetes, has a threshold-based scaling strategy that can be application-dependent and difficult to modify. Furthermore, its vertical scaling approach is disruptive, limiting deployment availability. The scaling decisions, instead of being proactive, are of reactive nature. In this work, our goal is to dynamically collect resource utilization of pods and predict future utilization for a period of time, and use the maximum utilization of that time window for proactive scaling, improving the overall resource utilization. We also contrast Kubernetes' built-in threshold-based scaling policy with a model-based reinforcement learning policy and the suggested LSTM Recurrent Neural Network-based prediction model. We demonstrate the benefits of data-driven rules, which can be combined with the Kubernetes container orchestrator.

what is deployment availability ?

1) dynamically collect " resource utilisation of pods

2) predict future utilization of that period (aggay kitna istamal karien gaye)

3) use "maximum utilization" of that time window (you are predicting)

> for proactive scaling
> improving overal resource utilization

1 Introduction

Container technology exploded in popularity with the debut of Docker in 2013. Docker is an essential component of the cloud ecosystem. Containerisation technology has swiftly become one of the hottest issues in the world of cloud computing due to its effective usage of computer resources and economic benefits. Shortly after the debut of Dockers in 2013, there was a flood of new container orchestrators aimed at reducing the complexity required in deploying, maintaining, and scaling containerised applications. One of these systems, the open source project Kubernetes, created by Google and now managed by the Cloud Native Computing Foundation (CNCF), has become the de facto standard for container management. Kubernetes, which offers container orchestration, deployment, and administration, is also crucial in cloud architecture. Because of its developer-centric container ecology features, Kubernetes has been the preferred option for container orchestration solutions as container technology has advanced. The main feature of Kubernetes is that it scales containerized applications up or down

R. Misra et al. (Eds.): ICIoTCT 2021, LNNS 340, pp. 1–16, 2022.
https://doi.org/10.1007/978-3-030-94507-7_1

based on the app's resource usage. The usage of resources changes based on the load demanded by the consumers. The load of the entire cluster is determined by the utilisation of each node. As a result, the advantages and disadvantages of the cluster's autoscaling method are critical [1–4].

2 Problem Statement

Threshold-based scaling strategies are application-dependent and difficult to fine-tune. The built-in autoscaler automatically suggests and modifies a pod's resource needs and limitations. Using the pods' usage data, it calculates a suggested value for the pod's resource demands and changes it to that number. The issue here is that if a pod is not running within the VPA's suggested range, it terminates the presently running version of the pod so that it may restart and go through the VPA admission procedure, which changes the CPU and memory demands for the pod before it can start. Because VPA only offers one suggested value for the time being, owing to the dynamic nature of load, one may wind up evicting pods too frequently, reducing availability and squandering many valuable resources. One of the reasons why a built-in vertical pod autoscaler is not utilised in production is because of this. We intend to forecast resource demands for application pods for a specific time period in the future. After analysing Kubernetes' general design and autoscaling approach, this predictive knowledge may be leveraged for vertical autoscaling that is more sustainable and reduces pod interruptions and waste of usable resources.

3 Theory and Related Work

3.1 Kubernetes Architecture

Kubernetes is made up of two parts: the master node and the worker node. The API Server, Kube-scheduler, and Controller manager are the three major fundamental components of the master node. The API Server component is in charge of replying to the user's management request, while the scheduler component connects the pod to the correct working node. The controller manager component, which consists of a group of controllers, is in charge of controlling and managing the corresponding resources. On the functioning node, two critical components, kubelet and kube-proxy, are operating. Kubelet is in charge of container life cycle management, as well as volume and network management. The Kube-proxy component is responsible for cluster-wide service discovery and load balancing [5,6].

The master node is in charge of the Kubernetes cluster's operation. It serves as the entry point for all administrative procedures. The master node hosts the control processes required by the whole system [7,8].

3.2 Kubernetes Threshold-Based Auto-scaling Policies

One of the most important characteristics of Kubernetes as a container orchestration platform is its ability to scale containerized workloads in response to changing conditions. This is known as auto-scaling. In a Kubernets cluster, there are three popular techniques for auto-scaling [9].

Horizontal Pod Autoscaler (HPA). Horizontal Pod Autoscaler controls the amount of pod replicas. Horizontal scaling is another name for it.

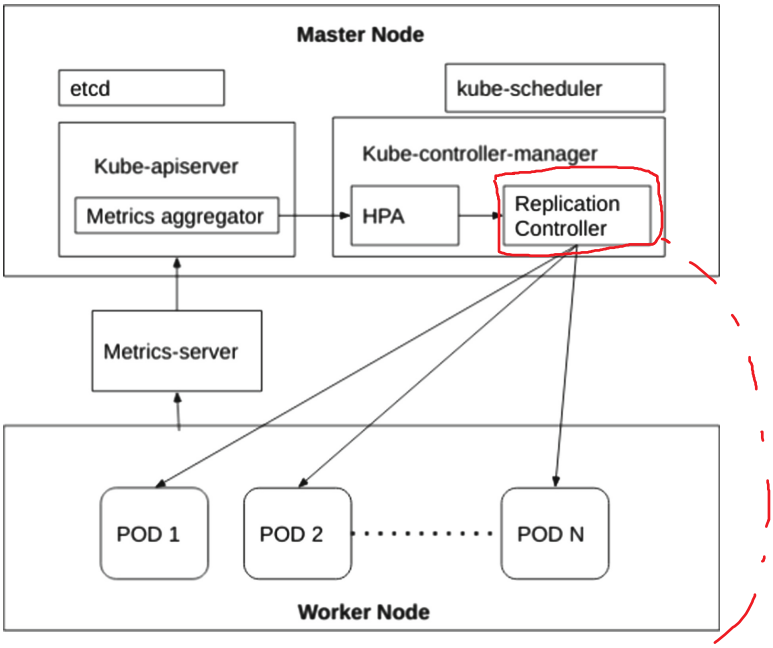

Fig. 1. Horizontal pod autoscaler (HPA)

The HPA controller collects utilisation information from the workload's pods' metrics-server and decides whether to change the number of copies of a running pod. In most situations, it does so by determining if adding or deleting a particular number of pod replicas will bring the current resource request value closer to the goal value. If there are n pods currently running and their individual CPU utilization is $U_1, U_2, ..., U_n$ then the average CPU utilization (U_{avg}) is the arithmetic mean of individual utilization.

how is U_i even measured ? is it a matrix or a variable, what is it ?

$$U_{avg} = \sum_{i=1}^{n} \frac{Ui}{n} \tag{1}$$

If the target CPU utilization is U_{target} then the HPA controller adjusts the number of pod replicas n such that U_{target} and U_{avg} is as close as possible.

For example, consider a deployment that have a target CPU utilization $U_{target} = 60\%$ and number of pods $n = 4$, and the mean CPU utilization $U_{avg} = 90\%$. Let n' be the number of pods need to be added to make $U_{target} \approx U_{avg}$.

$$\implies \frac{U_{avg} * n}{n + n'} \approx U_{target}$$
$$\implies \frac{90 * 4}{4 + n'} \approx 60$$
$$\implies n' = 2$$

that means 2 more pod replicas need to be added.

Cluster Autoscaler. It works in the same way as the Horizontal Pod Autoscaler (HPA), but instead of adjusting the number of replicas of a pod in the cluster, it changes the number of worker nodes based on the load.

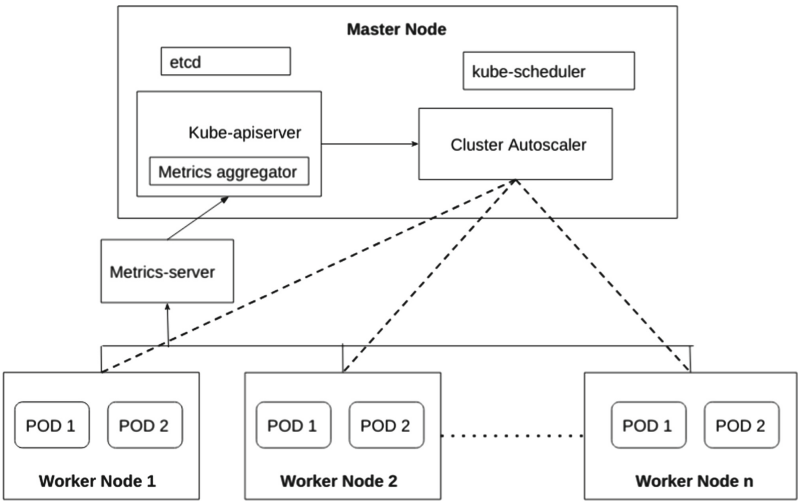

Fig. 2. Cluster autoscaler

The Cluster Autoscaler analyses the cluster to see whether there is a pod that cannot be scheduled on any of the available nodes owing to inadequate memory or CPU resources, or because the Node Affinity rule is in effect. Whether the Cluster Autoscaler discovers an unscheduled pod, it will examine its managed node pools to determine if adding any number of nodes can make this pod schedulable. If this is the case, it will add the necessary number of nodes to the pool if possible.

Vertical Pod Autoscaler (VPA). Vertical Pod Autoscaler (VPA) guarantees that the resources of a container are not under or overutilized. It suggests optimum CPU and memory requests/limits settings and may also automatically update them if in auto update mode, ensuring that cluster resources are utilised efficiently.

Vertical Pod Autoscaler is made up of three parts: the Recommender, which monitors pod resource utilisation using metrics from the metrics server and recommends optimal target values; the Updater, which terminates pods that need to be updated with newly predicted values; and the Admission Controller, which uses admission Webhook to assign the recommended values to newly created pods.

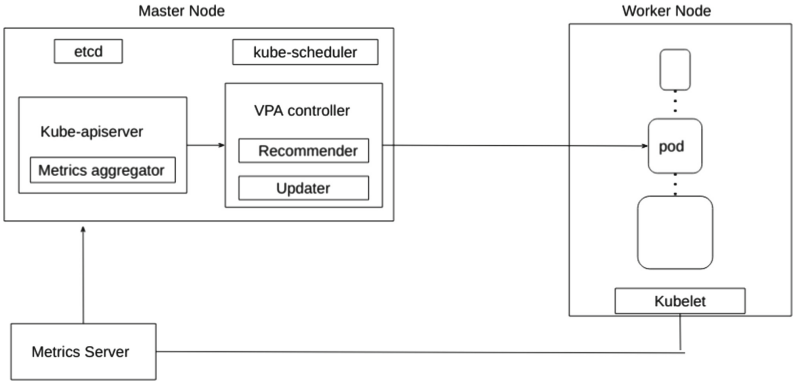

Fig. 3. Vertical pod autoscaler (VPA)

If a pod is specified with CPU request C_{req} and CPU limit C_{lim} and the recommended CPU request value is C_{newreq}, then the recommended CPU limit $C_{newlimit}$ is calculated proportionally:

$$C_{newlimit} = \frac{C_{lim}}{C_{req}} * C_{newreq} \tag{2}$$

For example, consider a pod p having $C_{req} = 50M$ and $C_{lim} = 200M$, then if the recommended request value $C_{newlimit} = 120M$, then:

$$C_{newlimit} = \frac{200}{50} * 120 = 480M$$

The main limitation of VPA is that A deployed pod's resource demands and limitations cannot be changed dynamically by the Kubernetes cluster. The VPA cannot add additional limitations to existing pods. It must evict pods that are not running within the anticipated range, and upon resuming, the VPA admission controller incorporates the suggested resource request and limit values into the specification of the newly formed pod.

3.3 Reinforcement Learning Scaling Policy

Many researchers have made significant contributions to enhancing the autoscaling policy [10,11] for various Kubernetes research topics. Fabiana Rossi [12] proposed a Reinforcement Learning based Kubernetes scaling policy to scale at run-time the number of conatinerized application pod instances. In Reinforcement Learning, an agent prefers activities that it found profitable in the past, which is referred to as exploitation. However, in order to uncover such rewarding behaviours, it must first investigate new activities, which is known as exploration. The RL agent determines the Deployment Controller state and updates the estimated long-term cost in the first phase, based on the received application and cluster-oriented metrics (i.e., Q-function) [13]. Furthermore, the Bellman equation replaces the simple weighted average of standard RL solutions (e.g., Q-learning) in updating the Q-function:

$$Q(s,a) = \sum_{s' \in S} p(s'|s,a)[c(s,a,s') + \gamma \min_{a' \in a} Q(s',a')] \tag{3}$$

Where γ is the discount factor c and p are cost function and transition probability respectively.

In the proposed autoscaler policy, Quality of Service requirements is expressed in terms of average response time with threshold $R_{max} = 80\,\text{ms}$.

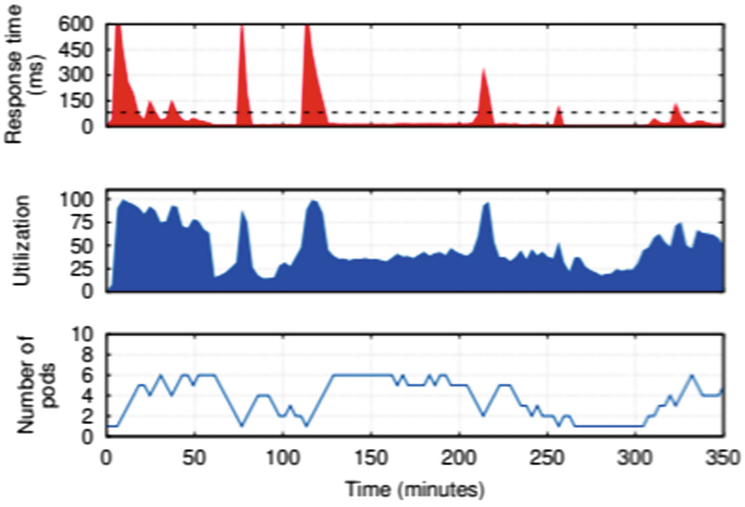

Fig. 4. Model-based RL autoscaler

4 Proposed Predictive Autoscaler

According to the circumstances described above, we need to dynamically gather resource consumption of pods and estimate future utilisation for a period of time, then scale based on the highest utilisation of that time frame.

We must gather historical resource consumption data and input it into our prediction module, which will offer resource recommendations for a period of time in the future. We shall then utilise the maximum resource request to avoid evicting pods too frequently.

The architecture of proposed predictive autoscaler is as follows:

Monitoring Module. Metrics Server is already present in an operational cluster. It is a scalable and efficient source for container resource measurements for Kubernetes' built-in autoscaling pipelines. These metrics can be used in our prediction model.

you can have Kalman filter do the job

LSTM Prediction Module. The use of resources on each working node varies with time. It cannot reflect the real usage of node resources based just on node information at the current moment. As a result of examining the link between the changing trend of monitoring nodes and time, as well as using previous resource consumption data supplied by the monitoring module, a prediction model is created to anticipate resource usage for a period of time in the future.

Vertical Pod Autoscaling Module. Evict pods that are not running within the expected range, and when the VPA admission controller restarts, it incorporates the suggested resource request and limit values into the newly formed pod's specification.

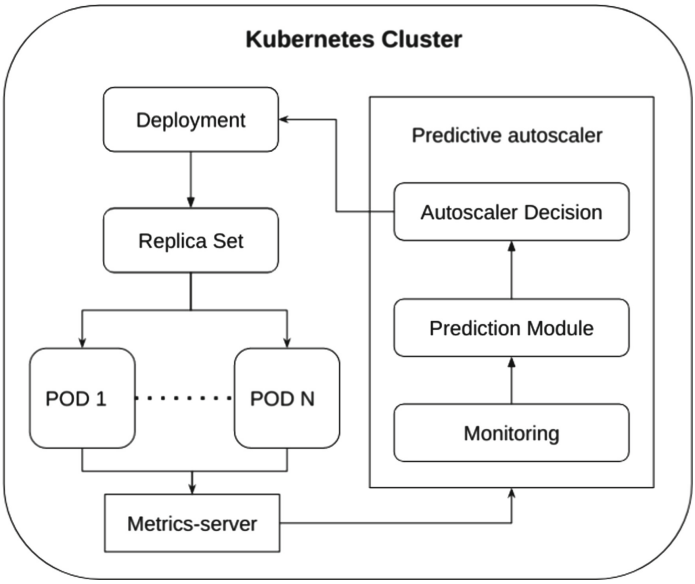

Fig. 5. Proposed predictive autoscaler architecture

5 Prediction Model

Long short-term memory (LSTM) [14] is a type of Recurrent Neural Network that has feedback that allows prior knowledge to be retained. When just short-term information is necessary to complete a task, traditional recurrent neural networks function effectively. Because of the vanishing/exploding gradients, RNN will struggle to represent a problem with long-term dependencies. The LSTM algorithm is a machine learning method designed to learn long-term dependencies. It keeps the information for a long time. In LSTM networks, memory blocks, rather than neurons, are connected across layers.

When compared to conventional neurons, an LSTM block contains distinct gates that allow it to store memory. A LSTM block comprises gates that determine the output and current state of the block. Each gate controls whether or not it is active by using the default sigmoid activation function units.

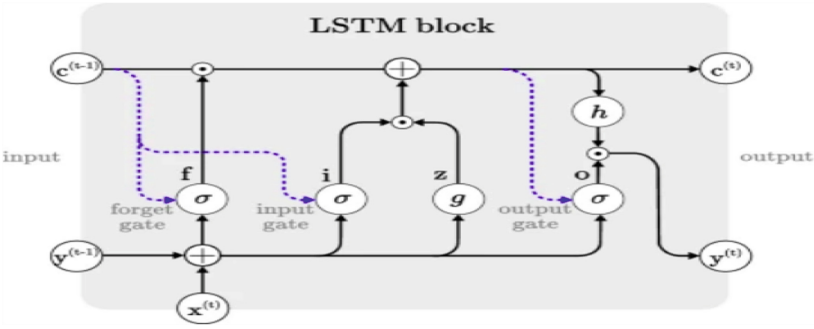

Fig. 6. LSTM architecture

There are three kinds of gates within a LSTM unit:

- **Forget Gate**: This gate determines information to be kept or thrown away from the block.
- **Input Gate**: This gate determines values from the input will be used update the memory state.
- **Output Gate**: This gate determines the next hidden state should be.

Each gate has default sigmoid function and a multiplication operation. Sigmoid function outputs to a range between 0 to 1 that is decided by the combination of $h(t-1)$ and $x(t)$. The output of the sigmoid function is then multiplied with the input to determine the gate's output. For example, if the output of sigmoid function is 1, then the gate's output will be equal to the input since input is multiplied by the result of sigmoid function. Each input vector to the unit is processes as following:

- The input vector $x(t)$ and the preceding hidden state vector $h(t-1)$ is chained to form another vector. The resultant vector will be used as the input in the three gates along with the tanh function.
- The forget gate uses the following equation to control what information to be kept within the unit:

$$f(t) = \text{sig}(W_f * [h(t-1), x(t)] + b_f) \tag{4}$$

where W is the weight and b is the bias.
- $\overline{C}(t)$, a new candidate values vector is calculated using:

$$\overline{C}(t) = \tanh(W_C * [h(t-1), x(t)] + b_C) \tag{5}$$

- The $\overline{C}(t)$ to be added to current cell is determined by multiplying it with $i(t)$ The equation of $i(t)$ is:

$$i(t) = \text{sig}(W_i * [h(t-1), x(t)] + b_i) \tag{6}$$

- The equation for the final current cell state $C(t)$ is:

$$C(t) = i(t) * \overline{C}(t) + f(t) * C(t-1) \tag{7}$$

- Output gate controls the amount the candidate value transferred into the next cell using:

$$o(t) = \text{sig}(W_o * [h(t-1), x(t)] + b_o) \tag{8}$$

- The equation for the final hidden state $h(t)$ is:

$$h(t) = \tanh(C(t)) * o(t) \tag{9}$$

6 Experimental Evaluation

Minikube, a tool for creating a local Kubernetes cluster, is being used for our experiment. For testing purposes, we installed a containerized Nginx web server on our Kubernetes cluster. We utilised Siege, a benchmarking programme, to generate traffic on our cluster, causing it to begin using cluster resources.

```
[ankur@vivobook]11:15 AM ~/k8s
-> kubectl apply -f nginx.yaml
deployment.apps/nginx configured
[ankur@vivobook]11:15 AM ~/k8s
-> kubectl get pods
NAME                    READY   STATUS    RESTARTS   AGE
nginx-7b8bd68b7-95ls8   1/1     Running   0          27s
```

Fig. 7. Deploying nginx on minikube

Once the containerized app is deployed, we generated traffic using Siege tool as discussed above.

```
[ankur@vivobook]11:22 AM ~/k8s
-> minikube service nginx-service --url
http://192.168.99.100:31856
```

Fig. 8. Exposing app URL

```
[ankur@vivobook]11:26 AM ~/k8s
-> siege http://192.168.99.100:31856
New configuration template added to /home/ankur/.siege
Run siege -C to view the current settings in that file
** SIEGE 4.0.4
** Preparing 25 concurrent users for battle.
The server is now under siege...
```

Fig. 9. Generating traffic using siege

Once our cluster starts getting traffic, we can see in our Kubernetes dashboard that CPU and Memory utilization starts increasing for our pod. We collected these values at regular time intervals.

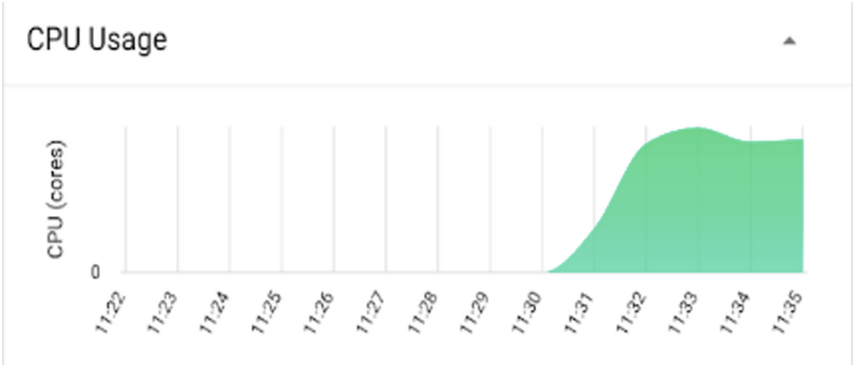

Fig. 10. CPU utilization after siege testing

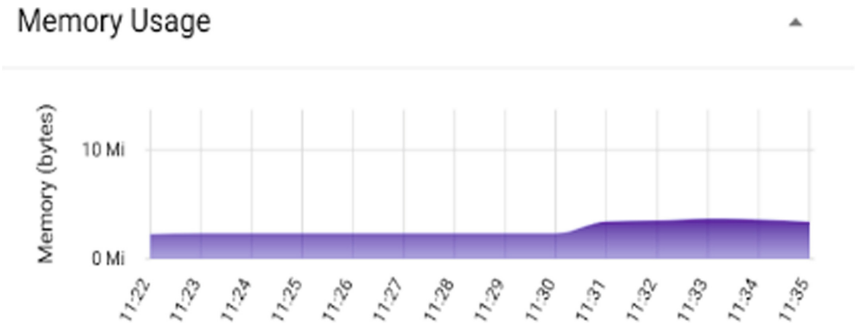

Fig. 11. Memory utilization after siege testing

6.1 Dataset

Real-time CPU and memory use data is required to develop the prediction model. The monitoring module, as previously mentioned, is used to create the dataset. We utilised Metrics Server, the most commonly used built-in open-source Kubernetes monitoring tool, which is already present in any Kubernetes cluster in production. We may send the measurements in whatever format needed and feed them into InfluxDB, an open-source time-series database. It receives metrics from Metrics Server and generates a time-series database with columns such as CPU Utilization percentage in Node, Number of pods in cluster, and so on, as well as the timestamp.

Plotting the complete CPU and Memory use dataset gathered using the aforementioned approaches, as well as its seasonal breakdown, for visualisation:

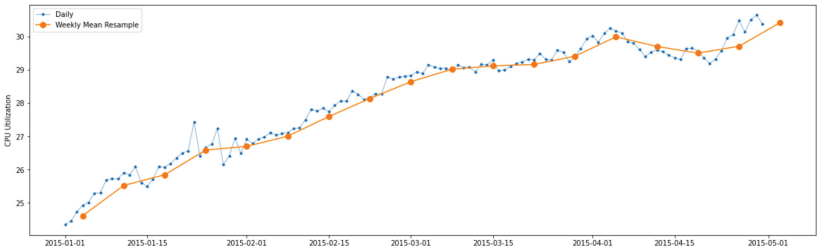

Fig. 12. CPU utilization

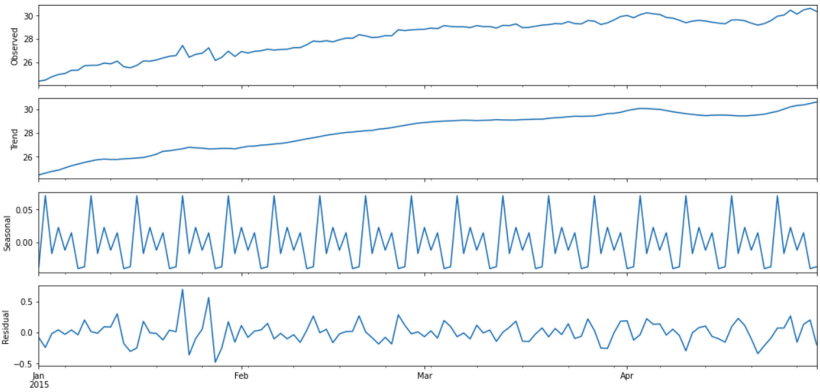

Fig. 13. CPU utilization components

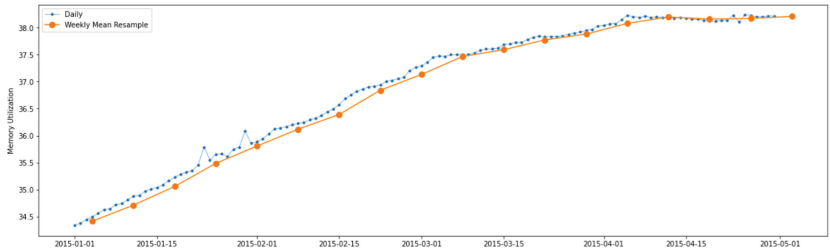

Fig. 14. Memory utilization

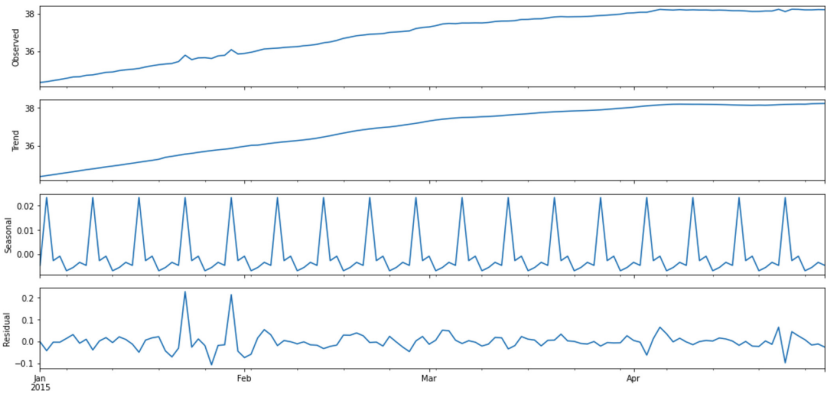

Fig. 15. Memory utilization components

The dataset has been split into training dataset that is 67% of the data and will be used for training the model, and the rest 33% data will be used for testing the model.

6.2 Evaluation Metric

We have used the Root Mean Squared deviation (RMSD) as a performance metric. The reason for choosing it over a strategy like Mean Absolute Error (MSE) because we do not desire to have large errors. RMSD uses the difference is squared terms; hence errors will reflect better in RMSD. RMSD is the difference between predicted and observed values first squared and then taken the mean of and lastly taken the square root of.

$$RMSD = \sqrt{\frac{\sum_{t=1}^{T}(x_{1,t} - x_{2,t})^2}{T}} \tag{10}$$

where $x_{1,t}$ and $x_{2,t}$ are actual observations series and estimated series respectively.

6.3 Prediction Results

We trained the LSTM RNN model with the collected utilisation data. With one visible input layer, one hidden layer and an output layer. The hidden layer has 4 LSTM blocks, and the prediction of output layer is a single value. The activation function used is the sigmoid function and Mean Squared Error was used as the loss function. After testing, the following results were obtained.

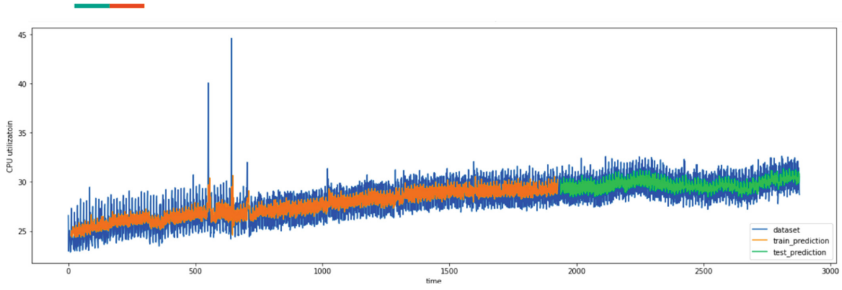

Fig. 16. CPU prediction

Our model performed well and the predictions are shown in the above diagram. Blue line represents the original data, green is the prediction on training data, and red is the prediction for test data. The RMSD error value of our CPU utilization forecasts is 1.49 for train data and 1.29 for test data, and for Memory utilization it is 0.29 for train data and 0.21 for test data.

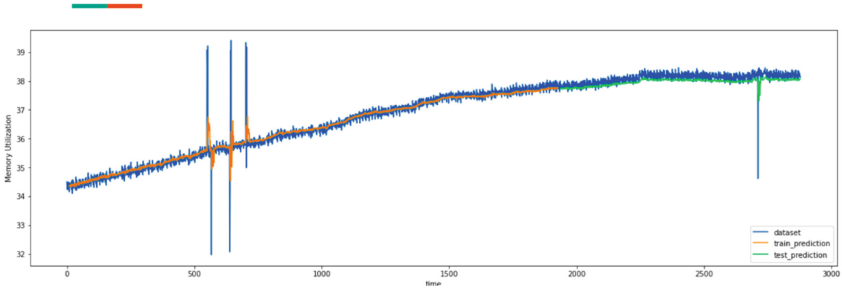

Fig. 17. Memory prediction

6.4 Comparison of Predictive Autoscaler with Default Autoscaler

The Kubernetes Vertical Pod Autoscaler uses statistical method for resource request recommendation. It does so by using weight buckets for historical resource usage and the weights diminishes exponentially. This is similar to the equation:

$$\hat{x}_{T+1\,T} = \alpha x_T + \alpha(1-\alpha)x_{T-1} + \alpha(1-\alpha)^2 x_{T-2} + \cdots \tag{11}$$

with $\alpha = 0.5$

Using above prediction equation, the RMSD Error value our CPU utilization forecasts is 1.31, and for Memory utilization it is 0.27.

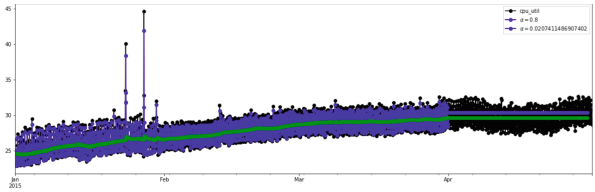

Fig. 18. CPU prediction using built-in VPA

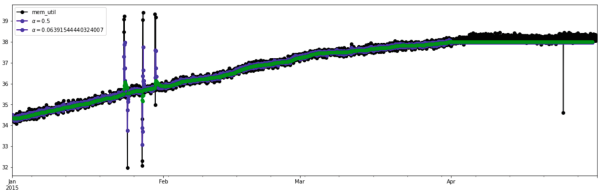

Fig. 19. CPU prediction using built-in VPA

As we can see in Fig. 16, 17, 18 and 19, the predictive autoscaler has performed better than the built-in autoscaler by slight margin. However, since predictive autoscaler uses data-driven approach, as it learns from more and more

data, it's performance is expected to improve further. Also, the built-in VPA can only recommend memory values above 250Mi. So for apps that need lesser resource, our predictive model will ensure that resources are not getting under-utilized. The predictive LSTM Recurrent Neural Network model has no such lower or upper limit of prediction that makes it more flexible in real-life use cases.

Table 1. Comparison of predictive VPA with built-in VPA using RMSD as evaluation metric

Data	Predictive VPA (RMSD)	Built-in VPA (RMSD)
CPU utilization	1.29	1.31
Memory utilization	0.21	0.27

7 Conclusion

In this paper, we addressed autoscaling policy of the original Kubernetes cluster by analyzing the overall framework, aiming at the limitations of threshold-based scaling, vertical scaling and it's disruptive nature. A proactive LSTM Recurrent Neural Network based predictive autoscaler is proposed to optimize the prediction combined with the autoscaling strategy. However, this method may increase the computational burden due to its complexity. Considering the simple linear relationship between the increased computational burden and the number of nodes, it is completely acceptable. Optimized proactive autoscaling policy can improve the solution to a certain extent, and ultimately improve the resource utilization and Quality of Service (QoS) effectively. By collecting historical resource usage data of applications running on the Kubernetes platform, a combined prediction model is established to predict resource usage for a period of time in the future. This prediction data can be applied to the dynamic autoscaling module, so that we can improving the cluster resource utilization and Quality of Service.

References

1. Hu T, Yannian W (2021) A kubernetes autoscaler based on pod replicas prediction. In :2021 Asia-Pacific conference on communications technology and computer science (ACCTCS), IEEE
2. Imdoukh M, Ahmad I, Alfailakawi MG (2019) Machine learning-based auto-scaling for containerized applications. Neural Comput. Appl. 32(13):9745–9760. https://doi.org/10.1007/s00521-019-04507-z
3. Gupta RK, Bibhudatta S (2018) Security issues in software-defined networks. IUP J. Inf. Technol. 14(2):72–82
4. Ticherahine A, et al (2020) Time series forecasting of hourly water consumption with combinations of deterministic and learning models in the context of a tertiary building. In: 2020 international conference on decision aid sciences and application (DASA), IEEE

5. Gupta RK, Ranjan A, Moid MA, Misra R (2021) Deep-learning based mobile-traffic forecasting for resource utilization in 5G network slicing. In: Misra R, Kesswani N, Rajarajan M, Bharadwaj V, Patel A (eds) ICIoTCT 2020, vol 1382. AISC. Springer, Cham, pp 410–424. https://doi.org/10.1007/978-3-030-76736-5_38
6. Meng Y, Ra, R, Zhang X, Hong P (2016) CRUPA: a container resource utilization prediction algorithm for auto-scaling based on time series analysis. In: 2016 international conference on progress in informatics and computing (PIC), Shanghai, pp 468–472
7. Xie Y, et al (2020) Real-time prediction of docker container resource load based on a hybrid model of ARIMA and triple exponential smoothing. IEEE Trans. Cloud Comput. https://doi.org/10.1109/TCC.2020.2989631
8. Zhao H, Lim H, Hanif M, Lee C (2019) Predictive container auto-scaling for cloud-native applications. In: 2019 international conference on information and communication technology convergence (ICTC), Jeju Island, Korea (South), pp 1280–1282
9. Toka L, Dobreff G, Fodor B, Sonkoly B (2020) Adaptive AI-based auto-scaling for kubernetes. In: 2020 20th IEEE/ACM international symposium on cluster, cloud and internet computing (CCGRID), pp 599–608. https://doi.org/10.1109/CCGrid49817.2020.00-33
10. Gupta RK, Choubey A, Jain S, Greeshma RR, Misra R (2021) Machine learning based network slicing and resource allocation for electric vehicles (EVs). In: Misra R, Kesswani N, Rajarajan M, Bharadwaj V, Patel A (eds) ICIoTCT 2020, vol 1382. AISC. Springer, Cham, pp 333–347. https://doi.org/10.1007/978-3-030-76736-5_31
11. Gupta RK, Raji, M (2019) Machine learning-based slice allocation algorithms in 5G networks. In: 2019 international conference on advances in computing, communication and control (ICAC3), IEEE
12. Rossi F (2020) Auto-scaling Policies to Adapt the Application Deployment in Kubernetes
13. Fabiana R et al (2020) Geo-distributed efficient deployment of containers with Kubernetes. Comput. Commun. 159:161–174
14. Van Houdt G, Mosquera C, Nápoles G (2020) A review on the long short-term memory model. Artif. Intell. Rev. 53:5929–5955. https://doi.org/10.1007/s10462-020-09838-1