2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C) | 978-1-6654-4393-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/ACSOS-C52956.2021.00025

Development of QoS-aware agents with reinforcement learning for autoscaling of microservices on the cloud

Abeer Abdel Khaleq
Department of Computer Science and Engineering
University of Colorado
Denver, CO USA
abeer.abdelkhaleq@ucdenver.edu

Ilkyeun Ra
Department of Computer Science and Engineering
University of Colorado
Denver, CO USA
ilkyeun.ra@ucdenver.edu

***Abstract*—Microservices play an essential role in cloud application scalability. When demand increases on a microservice-based application, the microservices need to be scaled to sustain the demand without degrading the application performance. At the same time, cloud platforms need to maintain Quality of Service (QoS) for deployed cloud applications. Current microservices autoscaling technologies such as Kubernetes Horizontal Pod Autoscaler (HPA) require identifying specific scaling metrics in addition to very good knowledge of the application resource usage. Those technologies do not provide a built-in autoscaling based on QoS constraints. In this work, we present an intelligent microservices autoscaling module using Reinforcement Learning (RL) agents. The RL agents are trained and validated on microservices for disaster management real time systems with response time as QoS constraint. Our RL agents deployed on Google cloud can identify the scaling metrics, provide microservices autoscaling, and enhance the response time compared to the default Kubernetes intelligently and autonomously. The RL agents serve as an extendible plug-in module to Kubernetes HPA for autoscaling microservices in the cloud while adhering to QoS constraints autonomously. The intelligent module is flexible to accommodate other types of QoS and provides a cost-effective solution to cloud applications autoscaling in areas of limited resources.**

Keywords—microservices autoscaling, Kubernetes, cloud applications autoscaling, qos, intelligent autoscaling

I. INTRODUCTION

Cloud applications developers utilize the microservices architecture for developing cloud applications based on the microservices foreseen advantages for scalability, agility, and reliability . A microservice is a small single task process that can be developed independently and deployed as a Docker container with its own runtime libraries. For cloud applications, scalability can be achieved at different levels from cluster to the container level. At the container level, a microservice needs to be able to scale horizontally under increased demand. Horizontal autoscaling is achieved by increasing the number of replicas for a microservice under demand to distribute the load and gain better performance. Available cloud technologies such as Kubernetes provide Horizontal Pod Autoscaling (HPA) based on certain scaling metrics such as resource utilization. However, the microservice resource demands need to be identified first to set the right scaling metric to achieve a better performance [1]. Cloud administrators need to have very good knowledge of the application resource usage and behavior to set the right scaling metrics in Kubernetes HPA.

Cloud providers also need to guarantee Quality of Service (QoS) for cloud applications based on certain Service level Agreements (SLA). Users negotiate SLAs to guarantee the QoS based on certain application characteristics. As a result, cloud computing services impose strict QoS constraints in terms of throughput and latency in addition to availability and reliability [2]. Designing architectures and algorithms for managing SLAs in clouds is still in its early stages, as there are profound challenges for enhancing traditional algorithms to satisfy SLAs [3]. Available cloud technologies such as AWS, Azure, and Google allow the user to set the threshold values for horizontal autoscaling. However, setting those values is challenging to allocate resources while maintaining the required SLA upon application of behavioral changes. QoS can be in the form of response time, latency, and others depending on the application domain requirements. In this paper we focus on real time systems where response time is a main QoS constraint. At the microservice level, the response time is calculated from the time when the microservice receives the request to the time when it finishes processing the request. When demand increases on a microservice, scalability becomes an important factor in guaranteeing QoS. This leads to the need for an autoscaling approach that will be able to learn the microservices resource usage and identify the set values for autoscaling while maintaining the QoS intelligently and autonomously.

Motivated by advancements in machine learning in general and reinforcement learning in particular, we present an intelligent, autonomous autoscaling module using RL agents. The advantage of the RL approach is, it will adapt to suit the environment based on its own experiences. In our work we adopt an RL agent model to learn the application resource usage behavior and identify autoscaling metrics that will guarantee QoS. Our goal is to utilize existing machine learning techniques including RL to aid in identifying the right autoscaling threshold values for a cloud based microservice system adhering to QoS constraints.

The RL agents are trained and validated using microservices log data for real-time application with response time as a QoS.

978-1-6654-4393-7/21/$31.00 ©2021 IEEE
DOI 10.1109/ACSOS-C52956.2021.00025

When deployed on Google Kubernetes Engine (GKE), our agents can identify the scaling metrics for microservices with different resource demand and enhance the response time compared to the default CPU-based Kubernetes HPA. The module design is flexible where it can be extended to include other QoS and scaling metrics. The module provides autoscaling for cloud applications at the container level intelligently and autonomously with minimum user involvement.

The contributions of our work are as follows:

- RL agent module for autoscaling microservices in cloud applications using response time as QoS. The module can be used as an extension to Kubernetes HPA to auto scale microservices intelligently and autonomously based on the microservice resource usage to minimize response time.
- The development and deployment of RL agents on cloud environment to identify the scaling metrics. The agents achieve a better response time compared to default HPA.

The rest of the paper is organized as follows, section II lists the related work in the area of microservices autoscaling, QoS and the use of machine learning. Section III describes the autoscaling framework and model. Section IV presents our experiment and results of the development and deployment of the RL agents on GKE. Section V is for the conclusion and future work.

II. Related Work

Research shows a need for more high-level approaches to address proper application scalability in a cloud context. It has been shown that QoS is an important constraint for autonomous cloud computing systems where the system services are able to execute, adapt and scale with minimum user interaction [4]. Threshold-based autoscaling is also a widely available research area where current approaches define observable metrics such as the application response time. Producing a reliable autoscaling system requires very good understanding of the target application, where autoscaling cannot meet the user performance demands by simply relying on CPU and memory utilization metrics alone [21]. Research in microservices autoscaling and machine learning is focused on using q-theory [5], predicting time series, the use of machine learning to predict load and resource allocation [6][7], and the use of fuzzy time series and genetic algorithms [8]. The studies consider hardware level SLA only instead of application-level requirements. They also work on the granularity of machines or VM not on microservices and Docker containers. In our study we focus on microservices autoscaling at the container level which is different from VM autoscaling.

Few studies investigate how to use application metrics to meet SLA requirements. The authors in [22] focused on service latency as QoS metric while reducing cost. T. Zheng et al. [8] show a dramatic decrease in SLA violation and resource efficiency as application-level metrics are incorporated into autoscaling algorithms. The authors looked at how log messages from containers are important for accurate monitoring of compliance with SLA requirements. Our work builds on this where we use microservices log data along with machine learning and reinforcement techniques for knowledge discovery to aid in microservices autoscaling with QoS constraints.

Rossi, Rossi et al. [9][10] described that a threshold-based scaling policy like the default Kubernetes HPA is not well suited to satisfy QoS requirements of latency sensitive applications. Such applications require identifying the relationship between a system metric such as utilization, an application metric such as response time, and the application bottlenecks. While their work focuses on CPU utilization, our work provides a generic learning module that can dynamically determine the resource metric of the application such as CPU, memory, or traffic load. Accurately identifying the optimal set of scaling rules is challenging. Also, relying on users for defining cloud controllers is not optimal, as users do not have enough knowledge about the workloads, infrastructure, or performance modeling [11]. This goes in hand with our strategy in using RL agents to acquire the system domain knowledge.

A recent study from Google [12] focuses on Autopilot vertical autoscaling of memory as it is less commonly reported. Vertical autoscaling is increasing or decreasing the resources' requests to accommodate the changing behavior of the service. We focus in this study on horizontal scaling of microservices as vertical autoscaling can be costly. In vertical autoscaling more resources need to be added, pods need to be stopped, throttled, moved, or restarted which can have a negative effect on the system performance.

In our previous work [13] we provided RL agents training and validation on simulated data for real time systems which showed that the agents, at least in the simulated environment, can identify the autoscaling metrics for resource utilization and the number of pods giving a response time below the QoS. In this study, we deploy the RL agents in the real cloud environment and test them on the deployed pods on GKE using response time as QoS. We show how our agents are able to identify the scaling metrics and satisfy the QoS with minimum user interaction.

III. RL Agents Framework Model

In this section we present our RL agent framework for autoscaling microservices in cloud applications. Fig. 1 provides the system architecture for the intelligent RL agent autoscaling module. The intelligent module holds the trained RL agents which will serve as an extension to Kubernetes HPA. The module allows the trained agents to access the pods resource usage log data, identify the resource demand that will minimize response time for QoS, and set the autoscaling metrics accordingly. The identified autoscaling metrics will then be fed into Kubernetes HPA to provide the autonomous autoscaling.

The RL model aims to train an underlying model until the training policy produces a desired outcome [14]. The policy is the way the agent interacts with the environment to produce the desired goal. Our goal is to auto scale based on the pod identified resource metric while maintaining the average response time below QoS. Fig. 2 provides the step function algorithm for the RL agent deployed in the environment. The observation and action spaces are represented as follows:

State = {curPods,cpuUtil,memUtil,avgRsp}

Act = {cpuScale, memScale} for CPU and memory scaling.

In our previous work, we included a penalty in our reward function if the average response time is above the QoS. In this study, we enhance the reward function to reach a minimum response time over time to give the agent a better convergence. We represent the reward function as an inverse of the current response time.

$$Reward = -avgRsp \quad (1)$$

In the following section we present our experiment on training, validation, and deployment of the RL agents on GKE.

IV. AUTOSCALING EXPIREMENT

In this section we provide the details of our experiment on training, deployment, and validation of our RL agents on autoscaling pods. We conclude the section with a comparison on vertical versus horizontal autoscaling for QoS application.

A. RL Agents Training

We have shown in our previous work that autoscaling based on resource demand enhances response time at the container level [1]. As we are building a generic framework to train our RL agents on, we incorporate different environments resembling different pod resource usage. We focused on CPU and memory as they are the two main pod resources affected by increased demand. We developed three RL agents' environments in Python and Tensor Flow for training the agents. A CPU-intensive environment that simulates a pod high on CPU (random CPU utilization of 310% to 780%), a memory intensive environment that simulates a pod high on memory (random memory utilization between 40% and 110%), and a dynamic environment simulating a pod that has varying resource demands for both CPU and memory (random utilization for CPU 50% to 780% and memory 30% to 150%). We simulated an inverse relation between the resource demand and the response time where autoscaling based on resource demand reduces response time [1]. We added a discount factor of value 1 to better train the agent on future rewards.

We trained a Deep Q Network (DQN) agent with 100 layers on the three environments. We trained for 20000 iterations, 1000 evaluation interval and a learning rate of 1e-3. The most common metric used to evaluate a policy is the average return which is the sum of rewards obtained while running a policy in an environment for an episode. Several episodes are run, creating an average return. We first collect data from the environment to train the agent's neural network. Fig. 3, 4, and 5 show the results of training the agent on memory, CPU, and varying resource usage environments. We observe how the reward gets better during the training and ranges between -6 and -3. After training the agent we saved the three different policies to be deployed in the real environment on GKE. We now present the results of testing the agents in the real environment using three different pods.

B. RL Agents Deployment

We tested our RL agents on a real-time application with the goal of minimizing the response time. The log data is retrieved from the deployed pod where it is in raw json format. We process the data to extract the pod CPU and memory utilization along with the pod response time. The application performs Twitter analytics as part of a disaster management system with

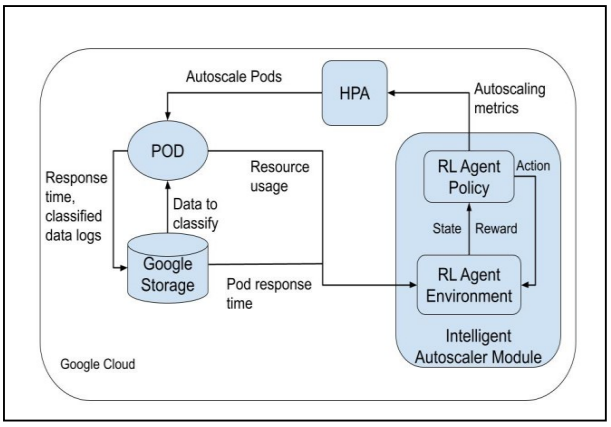

Fig. 1 Intelligent Autonomous Autoscaling Model Architecture

Input: Current Environment State (*State*), current Agent Action (*Act*)
Output: Observation new State(*State*), reward (*Reward*),

```
Get current agent action (Act).
Get curPods, cpuUtil, memUtil, avgRsp from current State.
if episode ended:
    reset State to initial default state.
Identify if action Act is 1 (cpu) or 0 (memory) autoscaling.
if(Act == 1)
    curUtil = cpuUtil
    targetUtil = State.cpuTarget
    scaleMetric = 1
else
if(Act == 0)
    curUtil = memUtil
    targetUtil = State.memoryTarget
    scaleMetric = 0
else
    Raise error, Act should be 0 or 1.
Apply Kubernetes HPA autoscaling formula based on identified scaling metric.
curPods = ceil(curPods * (curUtil / targetUtil))
Get deployed pods resource usage for memory and cpu.
Get deployed pods average response time avgRsp.
Record the current state observation.
State=[cpuUtil,memUtil,curPods,avgRsp,scaleMetric,
        targetUtil]
Calculate Reward.
Reward = - avgRsp
if(curPods > thresholdValue OR avgRsp < qos)
    End Episode
if(Episode Ended)
    Return State, Reward
else
Transition with Reward and discount factor.
    Record the current state/observation:
    [avgcpu,avgMemory, curPods,avgRsp,scale]
```

Fig. 2. RL Agent Step Function Algorithm

response time as QoS [15][16]. The deployed microservices perform disaster classification based on Twitter text data. The pod is high on memory and low on CPU demand [1]. Our microservice accesses the Google storage, gets the tweets data, classifies for disaster relevance, and returns the classified tweets. Response time is measured at the container level in seconds from the time the pod accesses the data, to the time it classifies it. The pod is deployed as a Web App that uses a tweet text classifier to classify tweets for disaster relevance. The text classifier is built using Multi-Layer Perceptron (MLP)

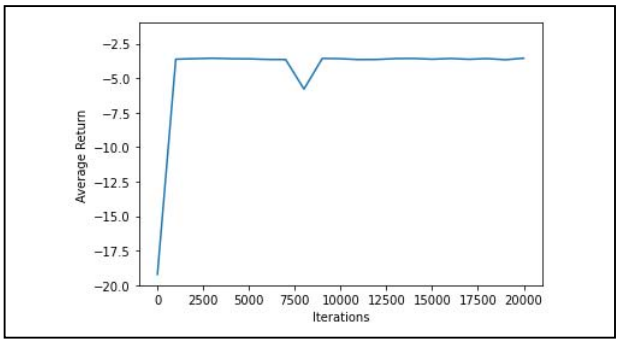

Fig. 3 Average reward of training the RL agent on a memory-intensive pod environment.

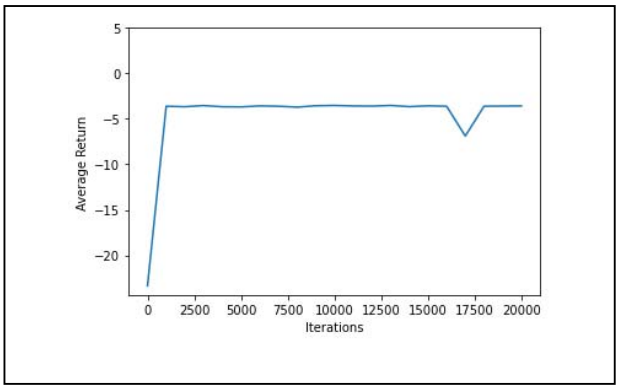

Fig. 4 Average reward of training RL agent on CPU intensive pod environment.

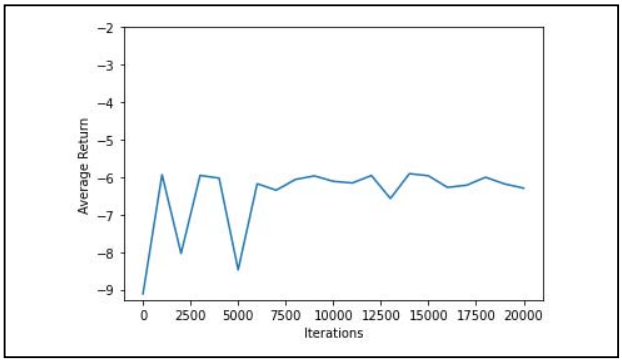

Fig. 5 Average reward of training RL agent on changing pod resource usage environment.

classifier [17] with hidden layers of size (100,100,100). We started the agents' deployment on the GKE cluster with 3 nodes of 6 VCPU and 12GB memory. We developed a script in Python to access the running pod resource usage for both memory and CPU using the Kubernetes API. We run the trained policy on the observed values in the real environment. The HPA will auto scale based on the identified resource values and scaling metric. Our goal is to observe how the trained agents will auto scale based on the resource demand. Table I shows the results we received for the disaster classification pod after running the three RL policies and increasing the traffic demand on the pod to trigger the HPA. CPU and memory utilizations are calculated by dividing the actual resource usage over the

TABLE I. RESULTS OF RUNNING THE RL AGENTS ON DISASTER CLASSIFICATION POD IN GKE SHOW HOW THE RL AGENTS CAN IDENTIFY THE SCALING METRICS WHILE PRESERVING RESPONSE TIME (QOS) .

Episode Number	RL Agent Trained Policy Environment	Max pods	Average CPU utilization	Average Memory utilization	Average Response time (sec.)	Scaling Metric (0 memory, 1 CPU)
Before traffic		1441	0.004	8.00	4.14	0
1	Memory Intensive	1665	0.004	8.6	4.03	0
2		1665	0.004	8.6	4.03	0
3		1665	0.004	8.6	4.03	0
Before traffic	CPU Intensive	3357	0.015	31.07	4.97	0
1		2583	0.05	11.57	4.75	0
2		2583	0.05	11.57	4.73	0
3		2584	0.05	11.58	4.72	0
Before traffic	Dynamic	4277	0.007	1.75	4.98	0
1		4277	0.008	1.75	4.8	0
2		4277	0.008	1.75	4.83	0
3		4277	0.008	1.75	4.82	0

request. We used 0.1 core for CPU request and 32MB for memory request. We can see that our agents picked the memory autoscaling which confirms that our agents can identify the autoscaling metric and threshold values to auto scale based on the pod resource usage. As for the response time, we can see that it did not increase due to autoscaling and even dropped slightly as the agents were trying to minimize the response time while identifying the threshold values.

Using the default HPA as a baseline, we repeated the experiment where we changed the pod CPU and memory requests, increased the traffic on the pod, and compared the achieved response time to the default HPA. The default HPA scales based on CPU utilization. The results are shown in Table II. There are extra factors outside the development environment that can affect the response time such as the network latency. As we run the pod as a Web App to send and receive requests, the network latency might increase the response time. On average, we can see that our RL agents achieved a response time that is less than the default HPA. Even though the enhancement is not significant, our agents were able to identify the autoscaling metrics intelligently and auto scale based on resource demand. Both the memory and the dynamic RL agents chose the memory autoscaling versus the CPU autoscaling as the pod is intensive on memory rather than CPU.

C. RL Agents Validation

To validate our results and the ability of our agents to identify threshold values over different pods, we repeated the experiment using two different pods. One that performs multiclassification using a different data set, and another pod that simulates a workload high on CPU and memory.

The multiclassification data set is based on hurricane Sandy Twitter data set [18]. We implemented a microservice which runs a multi-classifier and classifies the tweets into nine different categories of disaster information. We built the classifier using the MLP classifier with hidden layers of size (100,100,100) and 500 maximum iterations on a set of 646

TABLE II. RESULTS ACHIEVED BY RUNNING THE RL AGENTS ON THE DISASTER CLASSIFICATION POD ON GKE SHOWING THE RL AGENTS ACHIEVING A RESPONSE TIME BELOW THE DEFAULT HPA.

Auto scaler Policy	Average response time (sec.)	Number of pods	Max number of pods	Average CPU utilization	Average memory utilization	Target resource metric utilization	Scaling metric (0 memory, 1 CPU)
One pod	16.46	1	N/A	0.198	0.79		N/A
Memory RL agent	15.77	97	4912	0.18	0.77	0.1	0
	15.79	107	4912	0.36	0.87	0.1	0
CPU RL Agent	14.96 – 15.58	1	3547	0.03	0.75	0.7	1
	15.74	13	3547	0.36	0.82	0.7	1
Dynamic RL agent	15.4	1 – 96	3581	0.011	0.75	0.1	0
	15.63	106	3581	0.4	0.81	0.1	0
Default HPA	16.01	14	1000	0.45	0.82	0.7	1

TABLE III. RESULTS ACHIEVED BY RUNNING THE RL AGENTS ON THE CATEGORY MULTI-CLASSIFICATION POD ON GKE SHOWING THE RL AGENTS ACHIEVING A RESPONSE TIME BELOW THE DEFAULT HPA.

Auto scaler Policy	Average response time (sec.)	Number of pods	Maximum number of pods	Average CPU utilization	Average memory	Target resource metric utilization	Scaling metric (0 memory, 1 CPU)
One pod	5.03	1	N/A	0.017	0.74	N/A	N/A
Increase traffic	4.45	1	N/A	0.36	0.9	N/A	N/A
Memory RL agent	4.52	13	2968	0.37	0.78	0.1	0
CPU RL agent	4.42	13	1612	0.19	0.86	0.69	1
Dynamic RL agent	4.48	12	1625	0.43	0.79	0.69	1
Default HPA	5.17	12	1000	0.29	0.81	0.7	1

tweets. We got good accuracy around 98.4%. We deployed the pod on the GKE cluster and ran the RL agents auto scaler and the default HPA. Table III shows the achieved results. We can see that our RL agents were able to identify the scaling metrics and maintain a response time below the default HPA. Even though the response time enhancement is not significant, but it is not over the default HPA which is important as the agents identify the scaling metrics autonomously.

For our second pod, since our Twitter classification domain pods are mainly low on CPU, we developed a pod that runs some CPU intensive mathematical formulas that is high on CPU, memory, and disk usage. We deployed the pod on the GKE cluster and recorded our results in Table IV. We can see that our agents performed well compared to the default HPA. They picked CPU metrics for scaling except for the memory RL agent. The dynamic RL agent chose CPU autoscaling and achieved a response time less than the default HPA. All agents achieved a response time around or below the HPA. Since the default HPA is based on CPU metrics, we expect it to give a good response time for a CPU intensive pod. We conclude that our RL agents can identify the scaling metric, the threshold values, and maintain a response time below the default HPA on applications that exhibit consistent behavior in CPU or memory resource usage. For applications that exhibit varying resource demand, our RL agent detects one scaling metric. Adjusting the scaling metric based on resource demand's changes will be deferred to future work as the agents can dynamically learn how to adjust in the real environment.

D. RL Agents Horizontal versus Vertical Autoscaling on QoS

Google provides its own Vertical Pod Autoscaler (VPA) along with Autopilot [12][19], which provides recommendations for the pod resource requests and adjusts the resources dynamically while the pod is running. We utilize the Google VPA to give recommendations for the pod resource requests. Unlike the HPA, the VPA observes pods over time and gradually finds the optimal CPU and memory resources required by the pods. The VPA is meant for workloads not handled by the HPA [20]. It is not recommended to mix the HPA with the VPA on CPU and memory resources.

TABLE IV. RESULTS ACHIEVED BY RUNNING THE RL AGENTS AND THE DEFAULT HPA ON A CPU/MEMORY INTENSIVE POD ON GKE.

Auto scaler Policy	Average response time (sec.)	Number of pods	Maximum number of pods	Average CPU utilization	Average memory utilization	Target resource metric utilization	Scaling metric (0 memory, 1 CPU)
One pod	10.48	1	N/A	0.2	0.403	N/A	N/A
Increase traffic	11.4	1	N/A	3.5	0.52	N/A	N/A
Memory RL	9.65	55	1189	0.15	0.44	0.1	0
CPU RL agent	8.63	12	1122	0.12 – 0.39	0.42	0.69	1
Dynamic RL agent	8.26	9	3408	0.39	0.37	0.69	1
Default HPA	8.65	9	1000	0.41	0.42	0.7	1

Since the Twitter analytics application exhibits consistent behavior, we do not anticipate a huge spike in memory or CPU to consider the vertical pod autoscaling to adjust the resources dynamically. To study the effect of VPA on response time and compare it to the RL agents autoscaling, we ran the VPA on a pod that exhibits varying CPU and memory usage. The VPA gave the recommendation for the resources' requests and limits. We increased traffic on the pod, but we got errors and noticed that the pod was not schedulable, and the response time increased rather than decreased. This confirms that HPA works better with applications under heavy load/traffic compared to VPA.

We repeated the experiment by deploying the pod with varying CPU and memory on a Google Autopilot cluster which will automatically configure the cluster based on the deployed workload resource usage. In Autopilot, vertical autoscaling is enabled by default. After the pod is deployed, the Autopilot configured the pod resource limit and request for both CPU and memory. We increased the demand for the pod and recorded the results. Fig. 6 shows the response time and resource utilization achieved using Autopilot compared to RL agents and the default HPA. We can see that the response time increased in Autopilot compared to the horizontal auto scalers. However,

Autopilot gave better CPU and memory utilization which is explained by Autopilot value in optimizing resource allocation and cost reduction.

We conclude that for microservices with consistent resource usage, HPA works better under heavy usage demand. Our RL agents achieve a better response time and can auto scale horizontally with the correct threshold values and scaling metrics. Vertical autoscaling helped in understanding the microservice resource requests and the cluster resource allocation but did not improve response time. We defer the development of a hybrid approach utilizing both VPA and HPA on microservices with varying resource usage for future work.

V. CONCLUSION AND FUTURE WORK

In this work we presented an intelligent autoscaling module for microservices in cloud applications. The module is built using RL agents trained on microservices log data to identify the autoscaling metrics while preserving QoS constraints. The module provides an extension to Kubernetes HPA to auto scale the microservices intelligently and autonomously.

We deployed the RL agents on the Google cloud and analyzed the RL agents autoscaling on a real-time cloud application that performs Twitter analytics as part of a disaster management system. Where response time is the main QoS constraint, our agents were able to identify the scaling metric based on resource demand and achieve a better average response time compared to the default Kubernetes HPA. The module design is flexible where the RL agents can be trained on different types of QoS and on different cloud applications. The module will have the added benefit of less expert knowledge of the application resource usage and lower maintenance. Being hosted in the cloud and on top of Kubernetes open-source model provides a cost-effective solution for areas with limited resources.

We also provided a comparison on the effect of vertical versus horizontal autoscaling on the response time. We concluded that for microservices that exhibit consistent behavior in resource usage, horizontal autoscaling gives a better response time compared to vertical. However, vertical autoscaling achieves better resource utilization especially when the microservices exhibit dynamic resource changes.

Future work will be focused on studying other types of applications along with different QoS. For example, high-performance computing, training machine learning modules and AI solutions in the cloud, or on edge. We are also interested in developing a hybrid module for microservices autoscaling that will allow for horizontal and vertical autoscaling based on the application resource demand while maintaining the system performance in a more dynamic fashion. Another area of study is the effect of other parameters on the overall system performance such as the microservices intercommunication and latency. Future work can benefit from studying the effect of network latency on response time with cloud and edge deployment.

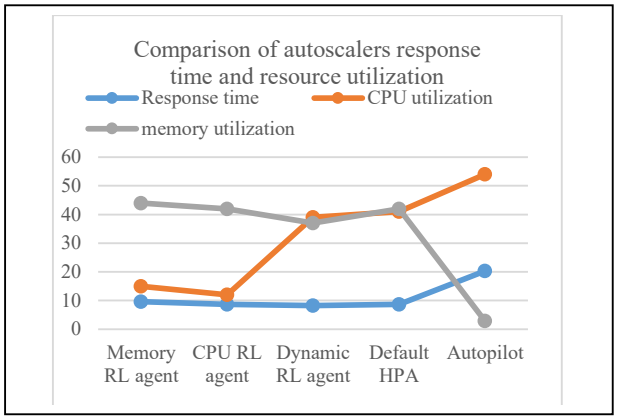

Fig. 6 Comparison of RL agents, default Kubernetes HPA, and Autopilot VPA on average response time and resource utilization for a pod with varying CPU and memory usage.

ACKNOWLEDGMENT

This research was developed with support from Google cloud for research and the department of Computer Science at the University of Colorado, Denver.

REFERENCES

[1] A. A. Khaleq, I. Ra, "Agnostic approach for microservices autoscaling in cloud applications," In Proc. CSCI, Las Vegas, NV, USA, 2019, pp. 1411-1415, DOI: 10.1109/CSCI49370.2019.00264.

[2] Y. Gan, and C. Delimitrou, 2018. "The architectural implications of microservices in the cloud," arXiv preprint arXiv:1805.10351Y.

[3] Ghahramani, Mohammad Hossein, MengChu Zhou, and Chi Tin Hon. "Toward cloud computing QoS architecture: Analysis of cloud systems and cloud services," IEEE/CAA Journal of Automatica Sinica 4.1 (2017): 6-18.

[4] Singh, Sukhpal, and Inderveer Chana. "QoS-aware autonomic resource management in cloud computing: a systematic review," ACM Computing Surveys (CSUR) 48.3 (2015): 1-46.

[5] Horovitz, S. and Arian, Y., 2018, August. "Efficient cloud auto-scaling with SLA objective using Q-Learning," in 2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud) (pp. 85-92). IEEE.

[6] Z. Yang, P. Nguyen, H. Jin and K. Nahrstedt, "MIRAS: Model-based reinforcement learning for microservice resource allocation over scientific workflows," 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), Dallas, TX, USA, 2019, pp. 122-132.

[7] Barrett, Enda, Enda Howley, and Jim Duggan. "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud," Concurrency and Computation: Practice and Experience 25, no. 12 (2013): 1656-1674

[8] Zheng, T., Zheng, X., Zhang, Y., Deng, Y., Dong, E., Zhang, R., & Liu, X. (2019). "SmartVM: a SLA-aware microservice deployment framework," World Wide Web, 22(1), 275-293.

[9] F. Rossi, "Auto-scaling Policies to Adapt the Application Deployment in Kubernetes," ZEUS, pp. 30-38, 2020.

[10] F. Rossi, M. Nardelli, V. Cardellini. "Horizontal and vertical scaling of container-based applications using reinforcement learning," In Proc. IEEE CLOUD, pp. 329-338, 2019.

[11] P. Jamshidi, A. M. Sharifloo, C. Pahl, A. Metzger and G. Estrada, "Self-learning cloud controllers: fuzzy q-learning for knowledge evolution," International Conference on Cloud and Autonomic Computing, Boston, MA, USA, 2015, pp. 208-211, DOI: 10.1109/ICCAC.2015.35.

[12] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak et al. "Autopilot: workload autoscaling at Google," In Proc. of the Fifteenth European Conference on Computer Systems, pp. 1-16, 2020.

[13] A. A. Khaleq and I. Ra, "Intelligent autoscaling of microservices in the cloud for real-time applications," in IEEE Access, doi: 10.1109/ACCESS.2021.3061890.

[14] "Train a deep Q-Network with TF-Agents," The TF-Agents Authors, [Online], Available: https://www.tensorflow.org/agents/tutorials/1_dqn_tutorial Last accessed 3/17/2021.

[15] A. Abdel Khaleq and I. Ra, 20018, November. "Twitter analytics for disaster relevance and disaster phase discovery," In Proceeding of the Future Technologies conference (pp. 401-417). Springer, Cham.

[16] A. A. Khaleq and I. Ra, "Cloud-based disaster management as a service: A microservice approach for hurricane Twitter data analysis," 2018 IEEE Global Humanitarian Technology Conference (GHTC), San Jose, CA, 2018, pp. 1-8.

[17] "Multi-layer perceptron classifier," Scikit Learn, [Online], Available: https://scikit-learn.org/stable/modules/neural_networks_supervised.html Last accessed 3/17/2021.

[18] Imran, M., Elbassuoni S., Castillo, C., Diaz, F., Meier, P., "Practical extraction of disaster-relevant information from social media," In: 22nd International Conference on World Wide Web, pp. 1021-1024. ACM, Rio de Janeiro, Brazil (2013).

[19] "Vertical pod autoscaling," Google cloud, [Online], Available: https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler. Last accessed 3/17/2021.

[20] "Best practices for running cost-optimized Kubernetes applications on GKE," Cloud Architecture Center, [Online], Available: https://cloud.google.com/solutions/best-practices-for-running-cost-effective-kubernetes-applications-on-gke. Last accessed 3/21/2021.

[21] "Cloud computing QoS for real-time data applications", European commission, [Online], Available: https://cordis.europa.eu/article/id/124056-cloud-computing-qos-for-realtime-data-applications, Last accessed 4/15/2021.

[22] S. Kho Lin et al., "Auto-scaling a defence application across the cloud using Docker and Kubernetes," 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), Zurich, 2018, pp. 327-334.