Cluster Computing (2023) 26:2789–2800
https://doi.org/10.1007/s10586-023-03999-8

Intelligent microservices autoscaling module using reinforcement learning

Abeer Abdel Khaleq[1] · Ilkyeun Ra[1]

Received: 14 March 2022 / Revised: 23 January 2023 / Accepted: 7 February 2023 / Published online: 28 April 2023

Abstract
Cloud applications developed using a microservices approach require a scaling policy that adapts dynamically to the changing resource demand of the microservice while satisfying the application's quality of service (QoS) agreement. Current cloud technologies, including Kubernetes Horizontal Pod Autoscaler (HPA), autoscale microservices based on predefined autoscaling metrics, which require a very good knowledge of the application's resource usage. There is no built-in QoS autoscaling mechanism. In this work, we introduce an intelligent autoscaling module for the horizontal autoscaling of microservices in the cloud. We develop, train, validate, and deploy reinforcement learning (RL) agents on a real-time application using response time as a QoS. The agents, deployed on Google Cloud, can learn the microservices autoscaling resource usage metrics from the cloud environment and customize the Kubernetes HPA while minimizing response time. Our results show a decrease in response time compared to the default Kubernetes HPA. The agents provide an extendible autoscaling plug-in module that can adapt to new QoS metrics.

Keywords Microservices · Autoscaling · Kubernetes · Cloud applications autoscaling · Qos · Intelligent autoscaling

1 Introduction

Note: This work is an extended version of the paper we presented at the AMGCC 2021 conference" [30].

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

✉ Abeer Abdel Khaleq
abeer.abdelkhaleq@ucdenver.edu

Ilkyeun Ra
ilkyeun.ra@ucdenver.edu

1 Department of Computer Science and Engineering, University of Colorado, Denver, CO, USA

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. 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:

- An intelligent autoscaling module using RL agents to autonomously and intelligently identify autoscaling threshold metrics for microservices in cloud applications.
- RL agents algorithm to identify microservices resource demand and autoscale while minimizing response time as a QoS reward.
- The development and deployment of the RL agents on Google cloud environment to identify the scaling metrics. Our agents deployed on real cloud environment for Twitter analytics achieve a better response time compared to the default HPA with minimum user involvement.
- We offer a comparison study on horizontal versus vertical autoscaling effect on response time as a QoS in cloud applications. We find that our RL agents achieve a better response time compared to vertical autoscaling, while vertical autoscaling provide better resource utilization.

The rest of the paper is organized as follows, section 2 lists the related work in the area of microservices autoscaling, QoS and the use of machine learning. Section 3 describes the autoscaling framework and model. Section 4 and 5 presents our experiment and results of the development and deployment of the RL agents on GKE and the effect of vertical autoscaling on QoS. Section 6 is for the conclusion and future work.

2 Related work and motivation

One of the main features of cloud providers is offering appropriate application scaling, preserving cloud computing main concepts of elasticity and dynamism [23]. 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]. Our work fills this gap, by offering an intelligent way for application scalability in cloud environment. We train and deploy RL agents to learn the right threshold values for autoscaling based on the application resource demand characteristics. 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]. These 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. Our work is different where we focus at the container level autoscaling. We deploy our agents on Google cloud to autoscale a Twitter analytics web-app microservice. Horizontal autoscaling at the microservice level is crucial for enhancing response time. Our RL agents can identify the microservices resource demand, and the autoscaling threshold values to autoscale while maintaining the QoS response time.

The authors in [27] developed a microservices autoscaler focusing on allocating virtual machines resources to meet end-to-end latency while minimizing dollar cost. While they focused on cluster level resource allocation, our work focuses on container level as it is the main component for horizontal autoscaling. Our work is different where we focus on identifying the microservice resource demand and horizontally autoscale using trained RL agents capable of identifying the threshold values. We do not use cost as a QoS, since response time is the major QoS to be maintained at the container level which affects the overall response time of the whole application. We focus on real-time applications where response time is the main QoS. Other studies focus on Kubernetes cluster-level resource utilization in maintaining QoS [28]. While those studies improve CPU utilization at the cluster level, they do not look at improving response time at the container level. Our work focuses on response time as a QoS and how to achieve it starting from the container level by identifying the right threshold values for HPA using intelligent RL agents in real-time.

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. Toka, L. et al. [23] used predictive AI models to autoscale web applications focusing on request loss and resource utilization as evaluation metrics. In their work, they propose a forecast-based approach for a proactive scaling policy trained on web traffic data using supervised and unsupervised neural networks. In our work, the RL agents are trained on microservices resource usage data, then deployed in the real cloud environment on a microservice-based application where response time is the main QoS. Many studies focused on autoscaling while maximizing resource utilization and minimizing cost. In our work, we focus on autoscaling while minimizing response time for real-time systems.

RL is a promising machine learning method that learns through trial and error [24]. Compared to other ML methods such as supervised and non-supervised learning which require a large amount of training data, RL does not require training data in advance and learns by obtaining rewards for actions applied to the environment. This is motivational for our study, as there is a lot of autoscaling parameters that need to be evaluated for microservices autoscaling. We are motivated by the RL agents' ability to learn from their environment and identify the right threshold values for microservices autoscaling. Being able to reward the agent on certain metrics can be used to hypertune the parameters on reducing response time.

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.

In this work we focus on horizontal microservices autoscaling. Horizontal scaling is replicating the microservice, or the deployed docker container, with its resources on the same machine or different ones. This is the most popular and less costly method that achieves high availability [25]. Vertical autoscaling, on the other hand, aims to maximize resource utilizations on a single machine by increasing the microservice CPU and memory allocated resources. A recent study from Google [12] focuses on Autopilot vertical autoscaling of memory as it is less commonly reported. 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. We test the effect of vertical autoscaling on the response time in our system later in the paper. Few studies looked at a hybrid approach combining horizontal and vertical autoscaling. The authors in [25]

Table 1 Related research work in autoscaling of cloud applications

Research paper	Scaling-level	QoS	Focus
[27] Sachidananda et al	Cluster-level	Cost, latency	Resource utilization
[28] Wu, Qiang, et al	Cluster-level	NA	Resource utilization
[22] S. Kho Lin et al	Kubernetes HPA	Latency	Defense systems
[23] Toka, L. et al	Resource provisioning	Request loss	Resource utilization
[9] Rossi, [10] Rossi et al	Kubernetes HPA	Response time	RL autoscaling
[12] Autopilot	N/A	Memory autoscaling	Vertical autoscaling
Our work	Container-level Horizontal	Response-time	Intelligent autoscaling

presented a hybrid autoscaling algorithm to dynamically allocate microservices resources preserving high availability and high resource utilization. They focused on the microservice needs at the container level instead of identifying the number of replicas. Table 1 summarizes some research work in the area of autoscaling in cloud applications highlighting our work and areas of contribution.

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.

3 RL agents framework model

In this section we present our RL agent framework for autoscaling microservices in cloud applications. Figure 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. Algorithm 1 provides the step function for the RL agent deployed in the environment. The algorithm starts by getting the current state of the deployed pods resources and response time. Then the agent identifies the scaling metric based on the current resource demand of CPU or memory. From there the HPA is called to update the number of pods based on the Kubernetes HPA formula:

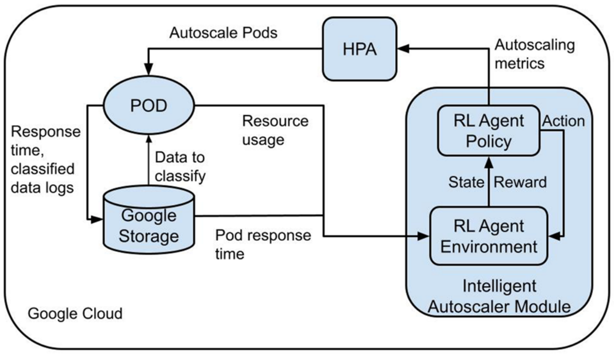

Fig. 1 Intelligent autonomous autoscaling model architecture

Algorithm 1 RL Agent Step Function Algorithm

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

```
1: Get current agent action (Act).
2: Get curPods, cpuUtil, memUtil, avgRsp from current State.
3: if episode ended then
4:     reset State to initial default state
5: end if
6: Identify if action Act is 1 (cpu) or 0 (memory) autoscaling
7: if $Act == 1$ then
8:     $curUtil = cpuUtil$
9:     $targetUtil = State.cpuTarget$
10:    $scaleMetric = 1$
11: else
12:    if $Act == 0$ then
13:        $curUtil = memUtil$
14:        $targetUtil = State.memoryTarge$
15:        $scaleMetric = 0$
16:    else
17:        Raise error, Act should be 0 or 1
18:    end if
19: end if
20: Apply Kubernetes HPA autoscaling formula based on identified scaling metric.
21: $curPods = ceil(curPods * (curUtil/targetUtil))$
22: Get deployed pods resource usage for memory and cpu
23: Get deployed pods average response time avgRsp
24: Record the current state observation
25: $State = [cpuUtil, memUtil, curPods, avgRsp,$
26: $scaleMetric, targetUtil]$
27: Calculate Reward
28: $Reward = -avgRsp$
29: if $curPods > thresholdValueORavgRsp < qos)$ then
30:     End Episode
31: end if
32: if Episode Ended then
33:     $ReturnState, Reward$
34: else
35:     Transition with Reward and discount factor
36: end if
37: Record the current state/observation:
38: $[avgcpu, avgMemory, curPods, avgRsp, scale]$ =0
```

$$newPods = ceil(curPods * (curUtil/targetUtil)) \quad (1)$$

Where newPods is the new number of pods to deploy for the autoscaling, curPods is the current number of pods deployed, curUtil is the current resource utilization value for the pods, and targetUtil is the target resource utilization value for the pods.

The scaled pods are deployed, and their resource utilization and response time are collected and averaged. We keep deploying and scaling the new pods for a set maximum threshold value for the maximum number of pods. The RL agent will receive a reward based on the current pods' response time aiming at maximizing the reward while maintaining the response time to be less than or equal to the QoS value.

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.

$$Reward = -avgRsp \quad (2)$$

We added a discount factor to better train the agent on future rewards. Our algorithm complexity can be measured based on the number of deployed pods (x) where the algorithm will find the average pod resource utilization and response time. The algorithm loop will stop when we reach the threshold value max for a maximum number of pods or achieved the required utilization. This can be represented as O(n * x) where n is the maximum number of pods threshold value.

As for the scalability, our algorithm focuses on scaling the pods horizontally, meaning adding more replicas for the pod under heavy load with the goal of keeping the resource utilization close to a target and the average response time below or equal to a target QoS value. The algorithm is scalable, where it will scale up the number of replicas per pod based on those two conditions. Furthermore, there is a set maximum threshold value for the number of pods that can be changed based on the system demand and available resources. As the agent will be deployed on a cloud cluster, scalability can also be achieved at the computing resource level.

The value of our work is in utilizing RL agents to identify the autoscaling threshold values based on resource demand in deployed cloud applications. In this study we focus on the response time as a QoS, however, the strength of our RL algorithm lies in its flexibility where it can adapt to different types of QoS. Since we utilize the QoS as a parameter to optimize the step and reward functions, we can see that other QoS parameters can be added, and the reward function parameters can be hyper tuned to achieve the QoS. Future research can experiment with multiple QoS constraints autoscaling. In the following section we present our experiment on training, validation, and deployment of the RL agents on GKE.

4 Horizontal autoscaling expirement

In this section we provide the details of our experiment on training, deployment, and validation of our RL agents on horizontal pods autoscaling.

4.1 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 (TF) for training the agents. Table 2 describes two of the developed environments, the CPU intensive and memory intensive.

A CPU-intensive environment 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 we also developed 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 in TensorFlow with 100 layers on the three environments. We trained for 20000 iterations, 1000 evaluation interval and a learning rate of 1e-3. Table 3 lists the DQN agent parameters we used in the training. We chose a DQN agent because it can be used in any environment with a discrete action space, our action space is based on scaling on CPU or memory represented as 0 and 1. We used the predefined AdamOptimizer from Keras, a loss function, squared-loss, and an integer step counter. AdamOptimizer is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments. The method is computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data and parameters [29].

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. Figures 2, 3, and 4 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.

4.2 RL agents deployment

In our previous work, we trained the agents in a simulated MATLAB environment to test the feasibility of the RL agent mechanism on identifying the scaling metrics [13]. In this work, we aim to deploy the RL agent in the real environment. We trained the RL agent policy in TF and created a Docker container to deploy on Google cloud. 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 response time as QoS [15] [16]. The deployed microservices perform disaster classification based on Twitter text data as part of a web application as depicted in Fig. 5 [1]. Horizontal autoscaling is crucial to web applications to adjust the microservices replicas based on resource demand while maintaining QoS. Our focus is on minimizing response time through horizontal autoscaling. The text analytics web application was developed in Python and allows us to increase traffic on the pod to increase resource demand and trigger the horizontal pod autoscaler. 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

Table 3 DQN hyperparameters used in training the TF agents

Parameter	Value
Number of iterations	20000
Initial collect steps	100
Collect steps per iteration	1
Replay buffer max length	100000
Batch size	64
Learning rate	1e-3
Log interval	200
Num-eval-episodes	10
Eval-interval	1000
Layer parameters	(100,)

Table 2 RL Agent Environments Parameters The different parameters for the TF agent custom environments CPU intensive and memory intensive. CPU and memory resources are in percentage, response time is in seconds

Parameter	Variables	Values	CPU intensive	Memory intensive
Observation Space	CPU Utilization	≥ 0	310–780	70–420
	Memory utilization	≥ 0	10–60	40–110
	Number of Pods	1 - Max	1–1000	1–1000
	Average response time	≥ 0	0.3–5.2	0.3–5.2
	Scale metric	0,1	[0,1]	[0,1]
	CPU target utilization	0–1	70	70
	Memory target utilization	0–1	10	10
Action Space	scaleMetric	[0,1]	[0,1]	[0,1]
QoS	Response time	Target response time	0.3	0.3
Reward	$-$avgRsp	≤ 0	$-$avgRsp	$-$avgRsp
Discount	Discount factor	1	1	1

for disaster relevance. The text classifier is built using Multi-Layer Perceptron (MLP) 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 4 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 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 5. 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.

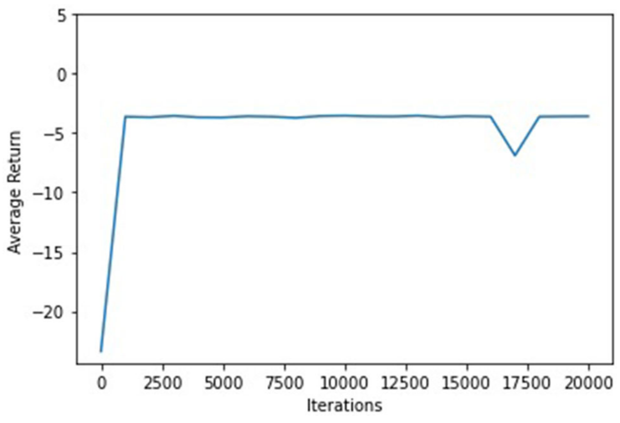

Fig. 3 Average reward of training RL agent on CPU intensive pod environment

4.3 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, see Table 6. 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 tweets. We got good accuracy around 98.4%. We deployed the pod on the GKE cluster

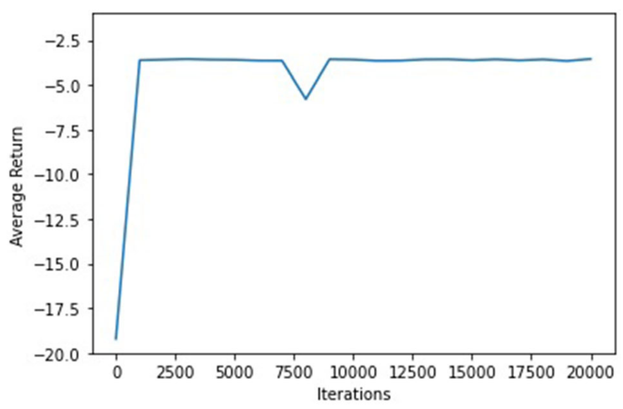

Fig. 2 Average reward of training the RL agent on a memory-intensive pod environment

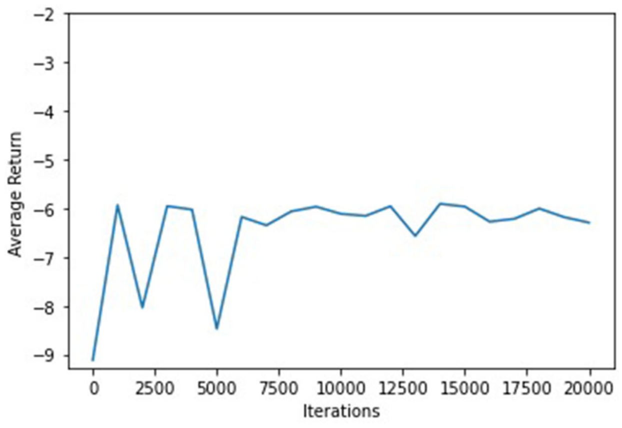

Fig. 4 Average reward of training RL agent on changing pod resource usage environment

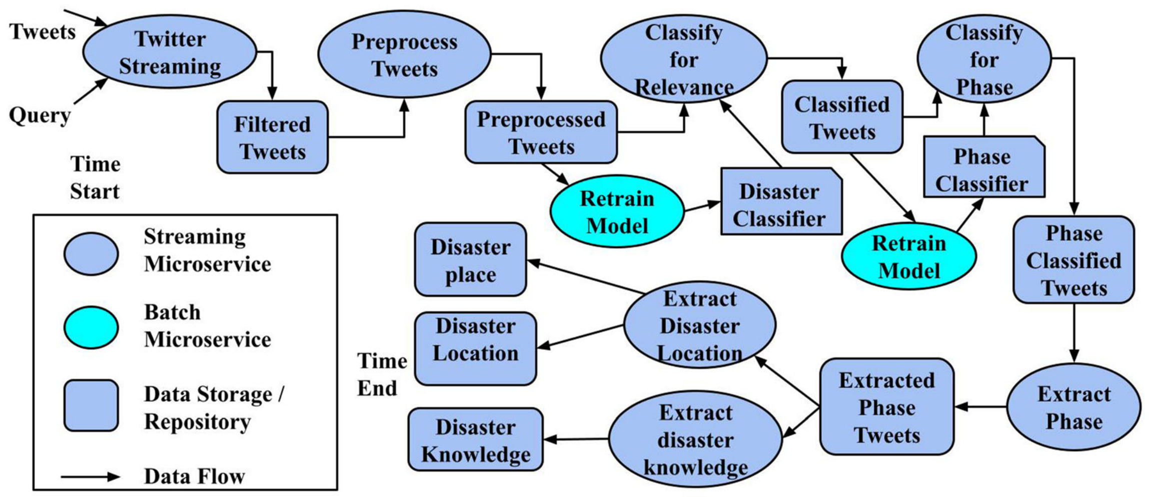

Fig. 5 Twitter analytics disaster management microservices workflow

and ran the RL agents auto scaler and the default HPA. Table 7 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 8. 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.

5 Vertical autoscaling effect On QoS

Our focus in this work is on horizontal scaling of microservices where the number of pods increases or decreases rapidly reacting to usage demand. Horizontal

Table 4 Results of running the RL agent on disaster classification pod in GKE show how the RL agents can identify the scaling metrics (0 for memory and 1 for CPU) while preserving response time (QOS).

Episode Number	Train Policy	Max Pods	CPU	Memory	Response Time	Scale
Before traffic	Memory Intensive	1441	0.004	8.00	4.14	0
1		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

CPU and Memory values shown are the average resource utilization

Table 5 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

Scaler policy One pod	Response time 16.46	Pods 1	Max Pods N/A	CPU 0.198	Memory 0.79	Target utilization N/A	Scale metric N/A
Memory Agent	15.77	97	4912	0.18	0.77	0.1	0
	15.79	107	4912	0.36	0.87	0.1	0
CPU 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 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 6 Hurricane Sandy data set labels used in the multi-classifier

Data label	Percentage
damage (building, road, lines, etc.)	31%
caution or advice	15%
casualties (people injured or dead)	5.6%
requests donations of money, goods, or free services	6.5%
information source with extensive coverage	15%
other type of photos/videos (not in the above classes)	14%
people missing, or lost people found	0.47%
offers/gives donations of money, goods, or free services	6.5%
celebrities or authorities react to the event or visit the area	5.9%

autoscaling is the most common and least costly compared to vertical autoscaling [26]. Vertical autoscaling is increasing or decreasing the resources' requests to accommodate the changing behavior of the service. It can be either at the container level by adjusting the pod requests or the cluster level by increasing or decreasing the number of nodes. As we focus on autoscaling at the container level, we defer cluster level scaling to future work.

As for the vertical autoscaling at the container level, 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. 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. 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.

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

Table 7 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

Scaler policy	Response time	Pods	Max pods	CPU	Memory	Target utilization	Scale metric
One pod	5.03	1	N/A	0.017	0.74	N/A	N/A
Increased traffic	4.45	1	N/A	0.36	0.9	N/A	N/A
Memory Agent	4.52	13	2968	0.37	0.78	0.1	0
CPU Agent	4.42	13	1612	0.19	0.86	0.69	1
Dynamic 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

Table 8 Results achieved by running the RL agents and the default HPA on a CPU/Memory intensive pod on GKE

Scaler policy	Response time	Pods	Max pods	CPU	memory	Target utilization	Scale metric
One pod	10.48	1	N/A	0.2	0.403	N/A	N/A
Increased traffic	11.4	1	N/A	3.5	0.52	N/A	N/A
Memory Agent	9.65	55	1189	0.15	0.44	0.1	0
CPU Agent	8.63	12	1122	0.12–0.39	0.42	0.69	1
Dynamic 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

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. Figure 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 believe that our work lays a good foundation for other researchers in the field to explore autoscaling for real-time systems. Possible research direction might look at developing a hybrid approach in autoscaling that combines horizontal and vertical microservices autoscaling to satisfy QoS for real time systems. We defer the development of a hybrid approach utilizing both VPA and HPA on microservices with varying resource usage for future work.

6 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 metrics. 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. Vertical autoscaling requires adjusting the resources at the service replica level, where a service has a varying demand in resources. Future work will focus on utilizing RL and AI techniques to identify those demands in a more dynamic way to perform HPA, VPA, or a combination of both while preserving other metrics.

This work provides a cost-effective solution to HPA in real-time systems with limited resources. However, we can also look into using HPC and GPUs to conduct RL training for larger data sets and larger space dimension. In addition

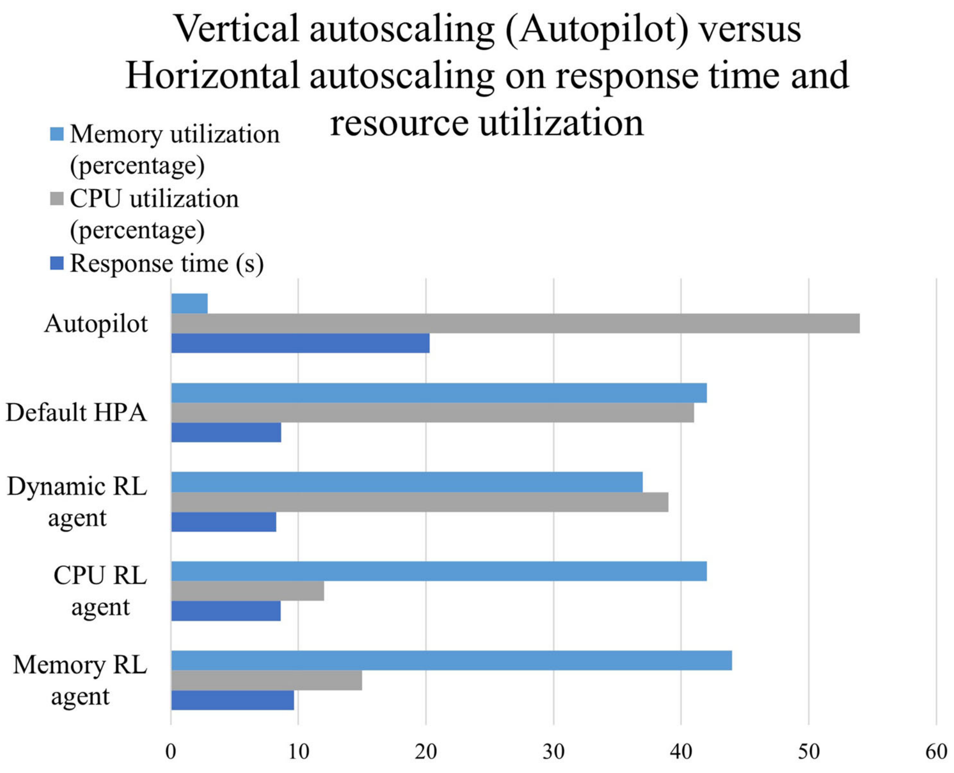

Fig. 6 Comparison of RL agents, default Kubernetes HPA, and Autopilot VPA on average response time and resource utilization for a pod with varying CPU and memory usage

to how using those resources affect the autoscaling QoS metrics. Some of the limitations of this work is in evaluating the effect of other parameters on the overall system performance such as the microservices intercommunication and latency. Some applications might have different SLA and might prioritize latency over response time depending on the application requirements. Future work can benefit from studying different applications domains and the effect of network latency versus response time as a QoS while incorporating cloud edge deployment.

Funding The authors have not disclosed any funding.

Data availability Enquiries about data availability should be directed to the authors.

Declarations

Competing interests The authors have not disclosed any competing interests.

References

1. Khaleq, A. A., Ra, I.: Agnostic approach for microservices autoscaling in cloud applications. In :Proc. CSCI. pp. 1411–1415. Las Vegas, NV, USA (2019). https://doi.org/10.1109/CSCI49370.2019.00264.
2. Gan, Y., Delimitrou, C.: The architectural implications of microservices in the cloud, arXiv preprint arXiv:1805.10351Y (2018)
3. Ghahramani, Mohammad Hossein, Zhou, MengChu, Hon, Chi Tin: Toward cloud computing QoS architecture: analysis of cloud systems and cloud services. IEEE/CAA J. Autom. Sinica **4**(1), 6–18 (2017)
4. Singh, Sukhpal, Chana, Inderveer: QoS-aware autonomic resource management in cloud computing: a systematic review. ACM Comput. Surv. (CSUR) **48**(3), 1–46 (2015)
5. Horovitz, S., Arian, Y.: 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 (2018 August)
6. Yang, Z., Nguyen, P., Jin, H., Nahrstedt, K.: MIRAS: Model-based reinforcement learning for microservice resource allocation over scientific workflows. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS). pp. 122–132. Dallas, TX, USA (2019)
7. Barrett, Enda, Howley, Enda, Duggan, Jim: Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. Concurr. Comput.: Pract. Exp. **25**(12), 1656–1674 (2013)
8. Zheng, T., Zheng, X., Zhang, Y., Deng, Y., Dong, E., Zhang, R., Liu, X.: SmartVM: a SLA-aware microservice deployment framework. World Wide Web **22**(1), 275–293 (2019)
9. Rossi, F.: Auto-scaling Policies to Adapt the application deployment in kubernetes, pp. 30–38. ZEUS, Olympia (2020)
10. Rossi, F., Nardelli, M., Cardellini, V.: Horizontal and vertical scaling of container-based applications using reinforcement learning. In: Proc. IEEE CLOUD. pp. 329–338. (2019)
11. Jamshidi, P., Sharifloo, A. M., Pahl, C., Metzger, A., Estrada, G.: Self-learning cloud controllers: fuzzy q-learning for knowledge evolution. In: International Conference on Cloud and Autonomic

Computing. pp. 208–211. Boston, MA, USA (2015) https://doi.org/10.1109/ICCAC.2015.35
12. Rzadca, K., Findeisen, P., Swiderski, J., Zych, P., Broniek, P., Kusmierek, J., Nowak, P., et al.: Autopilot: workload autoscaling at Google. In: Proc. of the Fifteenth European Conference on Computer Systems. pp. 1–16. (2020)
13. Khaleq, A.A., Ra, I.: Intelligent autoscaling of microservices in the cloud for real-time applications. IEEE Access (2021). https://doi.org/10.1109/ACCESS.2021.3061890
14. Train a deep Q-Network with TF-Agents. The TF-Agents Authors. https://www.tensorflow.org/agents/tutorials/1_dqn_tutorial. Last Accessed 17 Mar 2021
15. Abdel Khaleq, A., Ra, I.: 20018, November. Twitter analytics for disaster relevance and disaster phase discovery. In: Proceeding of the Future Technologies conference. pp. 401–417. Springer, Cham
16. Khaleq, A.A., Ra, I.: Cloud-based disaster management as a service A microservice approach for hurricane Twitter data analysis. In: IEEE Global Humanitarian Technology Conference (GHTC). San Jose, CA **2018**. 1–8 (2018)
17. Multi-layer perceptron classifier, Scikit Learn. https://scikit-learn.org/stable/modules/neural_networks_supervised.html. Last Accessed 17 Mar 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. https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler. Last Accessed 17 Mar 2021
20. Best practices for running cost-optimized Kubernetes applications on GKE, Cloud Architecture Center. https://cloud.google.com/solutions/best-practices-for-running-cost-effective-kubernetes-applications-on-gke. Last Accessed 21 Mar 2021
21. Cloud computing QoS for real-time data applications, European commission. https://cordis.europa.eu/article/id/124056-cloud-computing-qos-for-realtime-data-applications. Last Accessed 15 Apr 2021
22. Kho Lin, S. et al.: Auto-scaling a defence application across the cloud using Docker and Kubernetes. In: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). Zurich, pp. 327–334 (2018)
23. Toka, L., Dobreff, G., Fodor, B., Sonkoly, B.: Adaptive AI-based auto-scaling for Kubernetes, p. 599. IEEE, Piscataway (2020)
24. Lee, Do-Young., et al.: Deep Q-network-based auto scaling for service in a multi-access edge computing environment. Int. J. Netw. Manag. **31**(6), e2176 (2021)
25. Kwan, Anthony, et al.: Hyscale: Hybrid and network scaling of dockerized microservices in cloud data centres. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS). IEEE, (2019)
26. Khaleq, Abeer Abdel: Design and Evaluation of QoS Aware Intelligent Autoscaling Services for Microservices in Cloud Computing Environments. University of Colorado at Denver, PhD diss. (2021)
27. Sachidananda, Vighnesh, Sivaraman, Anirudh: Learned Autoscaling for Cloud Microservices with Multi-Armed Bandits. arXiv preprint arXiv:2112.14845 (2021)
28. Wu, Qiang, et al.: Dynamically adjusting scale of a kubernetes cluster under qos guarantee. In: 2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS). IEEE, (2019)
29. Kingma, Diederik P., Ba, Jimmy: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
30. Khaleq, A. A., Ra, I.: Development of QoS-aware agents with reinforcement learning for autoscaling of microservices on the cloud. In: IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C). pp. 13–19. DC, USA (2021) https://doi.org/10.1109/ACSOS-C52956.2021.00025

Abeer Abdel Khaleq received a Ph.D. degree in Computer science and information systems from University of Colorado, Denver, USA, and M.S. degree in Foundations of advanced information technology from Imperial College, London, UK, and BS in Computer science from Yarmouk University, Jordan. She held multiple positions as a computer science Faculty at various universities and academic institutions in CO, USA. She recently finished a post-doctorate fellowship in computational bioscience at the University of Colorado-Denver / Anschutz. She has published multiple conference and journal papers. Her research interests include cloud computing, microservices architecture and the development of real-time applications in the cloud using AI and ML for emergent technology.

Ilkyeun Ra received a Ph.D. degree in Computer and Information Science from Syracuse University, USA, MS degree in Computer Science from University of Colorado Boulder, Colorado, USA, and BS degree and MS degree in Computer Science from Sogang University, Seoul, Korea. Currently, he is an associate professor in the department of Computer Science and Engineering at the University of Colorado Denver. His main research interests include cloud computing, high performance computing, and computer networks.