IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 1, 2021, accepted February 13, 2021, date of publication February 24, 2021, date of current version March 5, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3061890

Intelligent Autoscaling of Microservices in the Cloud for Real-Time Applications

ABEER ABDEL KHALEQ, (Member, IEEE), AND ILKYEUN RA, (Member, IEEE)
Department of Computer Science and Engineering, University of Colorado at Denver, Denver, CO 80204, USA
Corresponding author: Ilkyeun Ra (ilkyeun.ra@ucdenver.edu)

ABSTRACT Cloud applications are becoming more containerized in nature. Developing a cloud application based on a microservice architecture imposes different challenges including scalability at the container level. What adds to the challenge is that cloud applications impose quality of service (QoS) requirements and have various resource demands requiring a customized scaling approach. For example, real-time applications require near real time response time as a QoS. Existing autoscaling technologies such as Kubernetes offer some customization to a set of threshold values for autoscaling. The challenge is identifying the right values for the different autoscaling parameters that will guarantee QoS in a changing dynamic environment. Advancements in machine learning and reinforcement learning (RL) provides a means for autoscaling in cloud applications with no domain knowledge. In this article, we introduce an intelligent autonomous autoscaling system for microservices autoscaling in the cloud with QoS constraints. The system consists of two modules. The first module identifies the microservice resource demand via a generic autoscaling algorithm deployed on the Google Kubernetes Engine (GKE). Our algorithm adapts the Kubernetes autoscaling paradigm based on the application resource requirements. The second module uses reinforcement learning agents to learn and identify the autoscaling threshold values based on the resource demand and QoS. Experimental results show an enhancement in the microservice response time up to 20% compared to the default autoscaling paradigm. In addition, the RL agents can identify the autoscaling threshold values while maintaining a response time below the QoS constraint. Our proposed work provides a customized autoscaling solution for microservices in cloud applications while adhering to QoS constraints with minimum user interaction.

INDEX TERMS Autonomous autoscaling, Kubernetes, microservices autoscaling, real-time cloud applications, reinforcement learning.

I. INTRODUCTION

Microservices are the new norm for cloud applications for dynamic, scalable, and reliable solutions. Microservice architecture application consists of several fine-grained services that are independently scalable and deployable [26]. This provides benefits over a monolithic architecture in areas of agility, reliability, scalability, and domain specific development [2]. Many companies such as Twitter, Netflix, and others adopted this model. A primary challenge for a cloud-based microservice system is scaling the microservices under heavy load and various resource requests and consumptions. A scalable service should be able to handle increases in load without noticeable degradation in the system performance [10].

The associate editor coordinating the review of this manuscript and approving it for publication was M. Anwar Hossain.

While many tools can scale the microservices such as Kubernetes, there are still many scaling parameters at both the container and the auto scaler level that need to be adjusted. Those parameters require a very good understanding of the application domain and its behavior to be set up accordingly. The fact that most of the cloud applications need to adhere to some service level agreements under QoS constraints adds to the challenge [9]. Real time systems, for example, use response time as a QoS metric. Scalability is a crucial requirement for real-time systems including disaster management systems [1]. Although the available autoscaling methods might be useful for some basic types of cloud applications, their performance and resource utilization drop when various CPU, memory, and network intensive time-critical applications need to be used [3]. Research shows how log messages from containers are important for accurate monitoring of

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

compliance with service level agreement (SLA) requirements and can be adopted in the auto scaler to make decisions [4]. Motivated by the fact that microservices cloud log data can be used to infer such knowledge [20], [21], we aim to utilize machine learning techniques to better understand the behavior of the microservices and identify the auto scaling parameters to guarantee QoS.

Microservices autoscaling in cloud applications using machine learning research focuses on the use of machine learning algorithms, enhancing some existing algorithms, and working on supervised trained models based on simulation experiments and existing training data. In contrast to our work, these studies' limitations are that they only consider hardware level SLA instead of application-level requirements. They also work on the granularity of machines or VM not on microservices and docker containers. Few studies investigate how to use application metrics to meet SLA requirements. Research shows a dramatic decrease in SLA violation and resource efficiency as application-level metrics are incorporated into autoscaling algorithms [4]. We build on this model by identifying and incorporating the application-level metrics in the autoscaling algorithm.

In autoscaling, threshold values such as resource utilization and a maximum number of pods need to be identified. Threshold-based autoscaling is a widely available research area. Current approaches define observable metrics such as the application response time. Determining appropriate thresholds requires expert domain and application knowledge and must be updated based on the application workload changes, which might not give the optimal strategy. The advantage of RL approach is how it adapts to suit the environment based on its own experiences. Our work fills this gap by utilizing RL agents to be trained to identify the autoscaling threshold values within the required QoS time frame.

Our proposed work will auto scale microservices in the cloud based on their demands and QoS using machine learning techniques. The proposed module can be deployed as an extension to Kubernetes HPA to automatically auto scale the workload with minimum user interaction based on the workload log data. We will use Twitter Analytics for disaster management as our problem domain [2], but we believe that the system design is flexible and can be further adapted for other types of data.

The major contributions of this article can be summarized as follows:

- Propose a generic autoscaling algorithm to identify the microservice resource demand in cloud applications.
- Developing and implementing an intelligent autoscaling module that identifies the threshold values for autoscaling the microservice based on the resource demand and QoS constraint.
- Testing and evaluating our proposed module using RL agents on microservices log data for real time system with response time as a QoS metric.

The remaining structure of the paper is as follows: section II describes the related work in microservices autoscaling and the motivation for our study, section III provides the system model and architecture. The experiment setup is described in section IV followed by results analysis and evaluation in section V. The last section VI is for the conclusion and future work.

II. RELATED WORK AND MOTIVATION

In this section, we will look at the related work in research in autoscaling microservices in the cloud, QoS metrics and the use of machine learning. We will also look at the areas where our study is unique and different. We will address microservices autoscaling, existing research in machine learning for microservices autoscaling, and the need for new smart solutions to adapt based on QoS of real-time cloud systems such as disaster management.

A. MICROSERVICES AUTOSCALING AND QoS

Previous approaches investigated autoscaling metrics on CPU intensive and I/O intensive microservices. They found that CPU utilization might not be the best metric for non-CPU intensive ones. However, in those studies, response time was not measured as a performance metric or the application's nature was considered [11], [12].

Kho Lin *et al.* [11] showed that in order to produce a reliable autoscaling system, it is important to understand the target application. The authors presented Kubernetes autoscaling capabilities on a live defense system and suggested that auto scaling cannot meet the user performance demands by simply relying on CPU utilization and memory usage metrics alone. However, they did not investigate the effect of other metrics on a microservice under heavy load. Gotin *et al.* [12] investigated a set of performance metrics in a threshold-based rules autoscaling for scaling a SaaS cloud application based on message queue state. The authors concluded that CPU utilization is a suitable metric if the microservice exhibits constant characteristics, but it causes low performance and is considered unreliable for I/O intensive microservice as its characteristics change under various loads. They concluded that message queue metrics are much more resistant to changes in microservices characteristics. Casalicchio and Perciballi [13] investigated the effect of absolute versus relative metrics in microservices autoscaling. They suggested that for CPU intensive workloads, absolute metrics such as CPU utilization enable more accurate decisions than the relative metrics used by the default Kubernetes autoscaling algorithm. The focus of their work was on CPU intensive applications, and they did not investigate the effect of the metrics on a microservice with other requirements.

To overcome the limitations of existing methods, we adapt the Kubernetes autoscaling method based on the microservice requirements. We test the effect of other metrics on the response time of non-CPU intensive microservices.

Gandhi *et al.* [15] explained the challenges of autoscaling cloud applications where it requires expert knowledge of the

dynamics of the application including SLA and sophisticated modeling expertise to determine when and how to scale the deployment. They propose an automated cloud service that proactively and dynamically scales the application deployment based on user-specified performance requirements. It leverages resource-level and application-level statistics to infer the underlying system parameters and determines the required scaling actions to meet the performance goals in a cost-effective manner. They used response time as the metric for SLA. Their work was more focused on cost-effective measures with minimum user interaction. Our work is different where we work on autoscaling at the microservice level autonomously, on real log data, and on minimizing response time for cloud applications with real-time QoS constraints.

In more recent work, Salman *et al.* introduced a new dynamic multi-level auto-scaling method with dynamically changing thresholds that uses both infrastructure and application-level monitoring data [3]. They focus on adapting the threshold values based on the microservices behavior under different loads and maintaining the overall cluster utilization. Our work is different where we use machine learning techniques to identify the autoscaling threshold values dynamically. This will better serve the application based on its changing behavior and characteristics with minimum user interaction.

B. MACHINE LEARNING FOR MICROSERVICES AUTOSCALING

Microservices autoscaling in cloud applications using machine learning research focuses on the use of machine learning algorithms, enhancing some existing algorithms, and working on supervised trained models based on simulation experiments and existing training data. In our work, we are not aiming at creating new machine learning algorithms, but we aim at utilizing existing machine learning techniques including RL to identify the right threshold values to auto scale a cloud based microservice system adhering to QoS constraint. In AWS, Azure and Google, the user can 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. Threshold-based autoscaling is a widely available research area. Current approaches define observable metrics such as the application response time. Determining appropriate thresholds requires expert domain and application knowledge and must be updated based on the application workload changes which might not give the optimal strategy.

It is widely accepted that reinforcement learning (RL) is an excellent adaptive and robust solution for autoscaling problems where it ensures a stable system utilization under dynamic workload conditions [8], [14]. We are motivated in our work by the advantages of RL approach and how it adapts to suit the environment based on its own experiences. In RL, policies are transparent where they are not dependent on human intervention or deep domain knowledge but rather learned through interaction with the environment. They are dynamic as they determine the best adequate action based on the current state of the environment and the application. And policies are adaptable, where they can adapt to the dynamic changes that occur in the cloud environment [27]. This suits the nature of cloud applications for scalability. Our work fills this gap by utilizing RL agents to be trained to identify the autoscaling threshold values within the required QoS time frame.

Related works in microservices autoscaling and machine learning are focused on using q-theory [17], predicting time series, the use of machine learning to predict load and resource allocation [16], [18], and the use of fuzzy time series and genetic algorithms [4]. Some of those studies only consider hardware level SLA instead of application-level requirements. They also work on the granularity of machines or VM not on microservices and Docker containers. We utilize RL agents to auto scale microservices which is different from autoscaling VM. Few studies investigate how to use application metrics to meet SLA requirements. Zheng *et al.* [4] 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 research builds on this where we use microservices log data along with machine learning and reinforcement techniques to auto scale the cloud microservices while abiding with QoS constraints.

Research shows a need for more high-level approaches to address proper application scalability in a cloud context. Sukhpal and Chana [19] described how QoS is an important constraint for autonomous cloud computing systems where the system services will be able to execute, adapt and scale with minimum user interaction. More research is needed in autonomic cloud computing based on various QoS parameters. Optimized data mining or machine learning techniques can be used to improve immediate decision making and help the services to adapt to dynamic unpredicted conditions. To that end, our work is filling this gap to introduce an autonomous, intelligent way for autoscaling workloads in the cloud.

Rossi [22] and Rossi *et al.* [23] described that a threshold-based scaling policy like the default Kubernetes HPA is not well suited to satisfy QoS requirements of latency sensitive applications which requires identifying the relationship between a system metric such as utilization and application metric such as response time as well as to know the application bottlenecks. The author runs a comparison on the default threshold-scaling policy of Kubernetes against a model-free and model-based RL polices. Their work focuses on CPU utilization, in contrast to our work, we provide a generic learning module that can dynamically determine the resource metric of the application such as CPU, memory, traffic load, etc. In their solution, they are changing the Kubernetes autoscaling algorithm to do the scaling in or out based on RL decision, observed metrics and on the setup of upper

and lower bounds for the number of pods. This is different from our work, where we build an intelligent module that can identify the right threshold values for resource metrics based on the application behavior and the ultimate number of pods which will satisfy the QoS metric such as response time. They proposed RL solutions for controlling the horizontal and vertical elasticity of container-based applications with the goal to increase the flexibility to cope with varying workloads. They show how RL may suffer from a possible long learning phase especially when nothing is known about the system a-priori. Their proposed solutions exploit different degrees of knowledge about the system dynamics using q-learning, Dyna-q and model-based on simulation and prototype-based experiments.

Jamshidi *et al.* [24] show how it is hard to accurately identify the optimal set of scaling rules and reliance on users for defining cloud controllers is not optimal as users do not have enough knowledge about the workloads, infrastructure, or performance modeling. The authors used fuzzy logic and developed an online learning mechanism to adjust and improve autoscaling policies at run time by combining fuzzy control and fuzzy q-learning. They incorporated SLA, cost, and response time in their reward function.

A recent study from Google on workload autoscaling describes the system Autopilot which is the primary auto-scaler that Google uses on its internal cloud as it provides both horizontal and vertical autoscaling [25]. The paper focuses on Autopilot vertical autoscaling of memory as it is less commonly reported. They use two main algorithms for vertical autoscaling one that relies on the exponentially smoothed sliding window over historical usage and the other uses RL by running many variants of the algorithm and choosing the one resulting in the best performance for each job. Serving jobs with strict performance guarantees on query response time for SLA are the primary driver of Google infrastructure capacity. The study emphasizes how autoscaling is crucial for cloud efficiency, reliability and toil reduction and how manually setting the resource limits not only wastes resources but also leads to frequent limit violations. Our work builds on that area of research for autoscaling microservices with SLA adherence effectively, autonomously, and dynamically.

Table 1 provides a comparison of existing work in autoscaling microservices in cloud applications, the metric used, the ML technique and the application scope.

TABLE 1. A comparison of related work in cloud applications autoscaling.

Reference	Metric	ML	Scope	Application
Kho Lin *et al.* [11]	CPU, Memory	-	microservices	defense system
Gotin *et al.* [12]	CPU, queue	-	microservices	threshold-based
Casalicchio [13]	CPU	-	microservices	absolute vs. relative metrics
Gandhi *et al.* [15]	CPU, memory, response time, cost	-	cloud applications	
Zheng *et al.* [4]	CPU, SLA	-	Virtual machines	log data
Rossi *et al.* [22][23]	CPU, response time	RL	autoscaling policy	autoscaling Policy
Jamshidi *et al.* [24]	SLA, cost, response time	fuzzy logic	autoscaling Policy	autoscaling policy
Autopilot [25]	memory, SLA	historic and RL	autoscaling jobs	autoscaling jobs
Our study	CPU, memory, queue, response time (QoS)	RL	microservices, containers	real-time systems (disaster management)

III. SYSTEM MODEL AND ARCHITECTURE

At the very top level of our problem design, we look at our system as a black box optimization problem. Black box optimization algorithms can be used to find the best operating parameters for any system whose performance can be measured as a function of adjustable parameters [28]. For an objective function f: $X \rightarrow$ R the overall goal is for the system to generate a sequence of x that approaches the global optimum as rapidly as possible.

We can model the system as follows:

For a given microservice (pod) *ms* we have the following parameters:

$$ms = [r, rs, u]$$

where r is the response time, rs is the resource in demand, u is the current resource utilization.

For the pod auto scaler, we have the following parameters:

$$scale = [min, max, target, curU, curP]$$

where *min* is the minimum number of pods which is typically 1, *max* is the maximum number of pods to scale, *target* is the required target utilization for the resource *rs* of pods, *curU* is the current average resource utilization for the pods, *curP* is the current number of pods.

Our objective function will tune the input parameters to generate the threshold values needed to auto scale that will give an average response time less than or equal QoS response time. The function can be represented as follows:

$$f(ms, scale) = S \tag{1}$$

where $S = \{min,max,rs,target\}$ that will satisfy the constraint:

$$Avg(r) <= QoS$$

From there, the threshold values will be given to the auto scaler to scale.

Since the problem space is vast with many log data for many running microservices, we will utilize machine learning and RL techniques to fine tune and learn the parameters that

will yield the right threshold values for autoscaling while maintaining performance below or equal QoS. Fig. 1 illustrates the model.

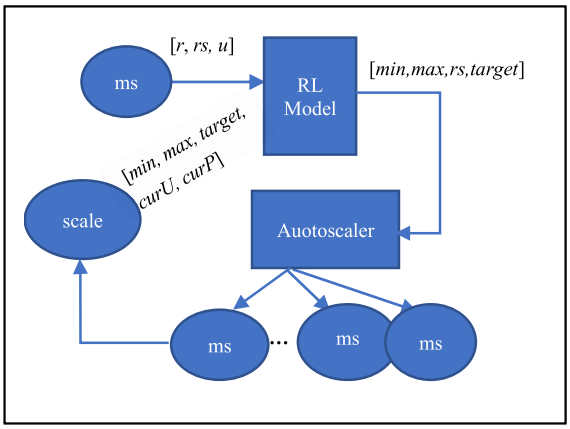

FIGURE 1. Autoscaling RL model illustrating input and output parameters.

We can define our problem as follows:

Application Domain: Disaster management.

Case Study: Twitter analytics for disaster phase discovery and disaster knowledge.

QoS Metric: Response time.

System Architecture: Cloud-based microservice architecture.

Goal: Autonomous autoscaling module at the application microservices container level satisfying the QoS constraint.

Methodology: Utilizing machine learning techniques to build an intelligent plug-in module on Kubernetes HPA using microservices log data.

Now we will describe the main areas of our system including the generic autoscaling model, the autonomous autoscaling model, and the machine learning autoscaling model.

A. GENERIC AUTOSCALING MODEL

We now present our microservices autoscaling model based on the Kubernetes HPA [5]. The architecture as illustrated in Fig. 2 is generic in nature, where we adapt the Kubernetes HPA based on the specific requirements of the microservice and the application QoS requirements. The system has a deployable controller module that will follow a customized algorithm to acquire the pod QoS metrics such as response time in addition to the resource requirements obtained from a log service such as Google Stack driver [6]. We will use the terms pod, microservice, and process interchangeably to refer to a small, identified task in the overall system.

The controller agnostic algorithm is based on the Kubernetes HPA. The algorithm which is presented in Fig. 3 starts by identifying the QoS metric and set of resource requirements for the microservice at the container level. The container will be deployed in one pod on

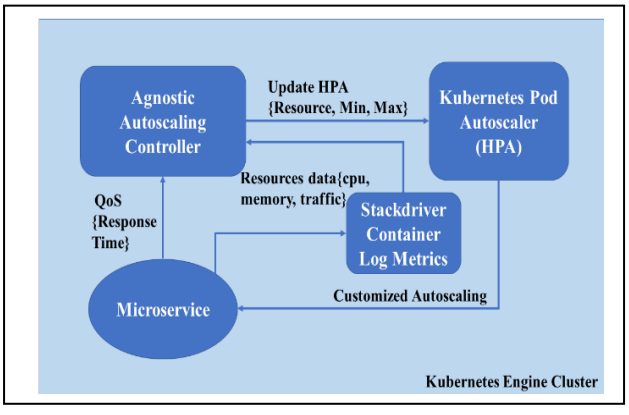

FIGURE 2. Agnostic autoscaling controller system.

Algorithm 1 Agnostic Controller Autoscaling Algorithm:

Let A = {ms_i: i=1,2,3,.# of microvervices} set of microservices for application A.
Let qs_i = QoS metric value for microservice ms_i
Let R_i = {rq_i^j: j = 1,2,3,# of resource requests} set of resources requests for ms_i

1: **For each** ms_i in set A **do**:
2: Deploy ms_i as pod_i on Kubernetes Cluster with resource request R_i
3: Start j=1 for number of replicas for pod_i
4: Add pod_i^j to replica set P_i
5: Get set of resources values for pod_i S_i = {cpu_i,ram_i,tr_i} from Stackdriver. (CPU, RAM, Traffic)

6: **Procedure** HIGHESTRESOURCEDEMAND(rs_j,S_i)
7: **For each** rs_j in S_i **do**:
8: If $rs_j > rq_i^j$ then
9: **return** rs_j

10:**Procedure** RESOURCETHRESHOLDS(rs_i,qs_i)
11: $min_i = 1$
12: $max_i = j$
13: **If** rs_i == *cpu* **then**
13: $target_i$ = AVERAGE_VALUE_CPU (*initial value for cpu*)
14: **else**
15: **if** rs_i == *ram* **then**
16: $target_i$ = AVERAGE_VALUE_RAM (*initial value for ram*)
17: **else**
18: **if** rs_i == *number_undelivered_messages* **then**
19: $target_i$=AVERAGE_VALUE_MESSAGES (*initial values for # of messages*)
20: **scale**(min_i,max_i,$target_i$,qs_i)

21:**Procedure** SCALE(min,max,rs,target,qs)
22: Let s_i represent response time for quality of
23: service qs.
24: Let t_f represent finish time of service.
25: Let t_s represent start time of service.
26: $s_i = (t_f - t_s)_i$
27: **do**
28: max++
29: Update HPA yaml file based on scale parameters min,max,target.
30: Deploy HPA
31: Find average response time a_i for service i:

$$a_i = \sum_{n=1}^{max} s_i^n / max$$

32: Get resource usage value $target_i$ for rs_i
33: **While**(a_i >= s_i)
34: **return** $target_i$,max_i

FIGURE 3. Generic autoscaling Algorithm.

Kubernetes Cluster. The pod usage will be obtained from the pod logs. The pod resource with the highest demand will be identified and used in adjusting Kubernetes HPA scaling. The threshold values for the resource, minimum, and a maximum number of pods will be identified through an optimization technique. The algorithm monitors the resources and dynamically adjusts the threshold parameters as the pods are deployed. Our algorithm scalability can be looked at based on the number of microservices to be deployed along with the number of instances per microservice to be scaled. The two numbers are finite, as we do not anticipate them to grow to be very large numbers. Assuming n is the maximum number of microservices deployed, and m is the maximum number of replicas per a microservice, then the scalability of our algorithm performance is $O(n^*m)$. Our focus is on testing the algorithm scalability at the container level.

B. AUTONOMOUS INTELLIGENT AUTOSCALING MODEL

Our focus in this work is to further improve the autoscaling algorithm with the use of machine learning and reinforcement learning to identify the microservices resource requirements and scalability thresholds satisfying the QoS agreements. As there are different parameters that need to be adjusted to guarantee the QSA, manually adjusting those values is tedious work that requires a good understanding of the application characteristics.

Fig. 4 provides the general architecture of the intelligent autoscaling Module. The general module workflow steps are described below:

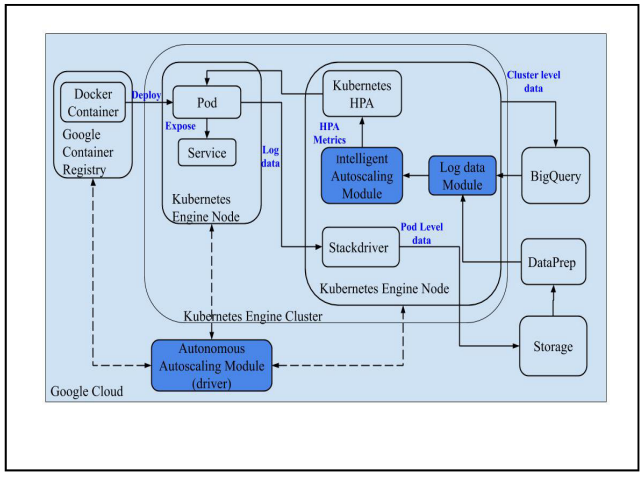

FIGURE 4. Intelligent autoscaling module system architecture.

1. The system is hosted in the Google cloud cluster.
2. The autonomous autoscaling module is the driver of the microservices autoscaling workflow, where it starts with the microservices to be deployed and auto scaled along with the QoS such as response time.
3. The microservice Docker container is pulled from the container registry to be deployed as pod.
4. The Kubernetes pod vertical autoscaling is run to get the recommended resource settings for running the microservice including CPU and memory.
5. The yaml file for the pod is set up based on the resource requests.
6. The pod is deployed on the Google cloud cluster, and it is exposed as a service.
7. Log data at the pod level is collected through Stackdriver to get the pod response time dynamically. The data is stored, cleaned, and prepared through DataPrep.
8. Cluster level log data showing the cluster resource consumption for CPU and memory at the pod level is collected and sent to Big Query dynamically.
9. All log data will be stored, processed, cleaned in the Log data module.
10. The log data will be fed into the intelligent autoscaling module that will identify through machine learning models the right threshold values for autoscaling the pods, including target resource, target utilization, the minimum and maximum number of pods satisfying QoS requirement for response time.
11. The threshold values are entered into Kubernetes HPA to auto scale the pod.
12. The cycle repeats from step 7 as more log data will be analyzed and the threshold values will be adjusted accordingly based on the intelligent module to scale the pods up or down to satisfy the QoS response time.

C. MACHINE LEARNING FOR MICROSERVICES AUTOSCALING

Fig. 5 describes the autoscaling model. The autoscaling is based on two machine learning modules deployed in the Google cloud cluster. As the system is dynamic in nature, we include a loop preserving the dynamic nature of microservices and how they can change their resource demand during deployment. We followed RL design principles where the system model is separated into sub models. This design will also help in providing a model where we can simulate the environment and get a model-based learning that expedites the learning process for exploitation instead of total exploration once the model is deployed in the real environment.

The first module, vertical resource demand learner (VRDL) identifies the resource demand for an incoming pod based on the accumulated log data of the cloud cluster and container. Our microservices are deployed as Docker containers on Google Kubernetes Engine (GKE). We send the log data to Big Query to be worked on. We run a query that looks at trends of resource request versus consumption. The second module, the horizontal reinforcement learning module (HRL) is where the dynamic learning is achieved using reinforcement learning based on trained agents deployed in the environment to learn the best threshold values for the HPA parameters. This hybrid approach in machine learning combines both historical trends and dynamic RL to better design the system to handle both vertical and horizontal autoscaling as the microservices can change their dynamics

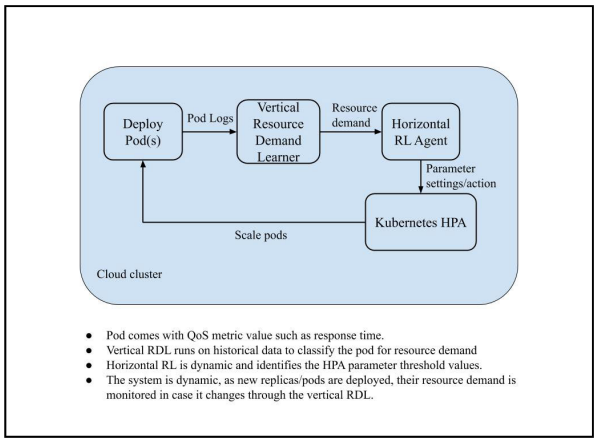

FIGURE 5. Machine learning autoscaling modules.

and resource demand as the system is running. Here is a description of the loop steps:

1. A microservice (pod) first comes in where its resource requirements need to be determined.

2. Run vertical scaling on the pod in GKE to get the initial recommended settings for the pod CPU and memory request.

3. Deploy the pod on the cluster, feed the GKE cluster logs into Big Query.

4. Run a query over log time-series (e.g. 1 hour) in Big Query to identify the pod trend consumption for resources and the ratio of consumed versus requested resources. This step can benefit from historical machine learning for an incoming pod based on its trend in resource usage or consumption and a new pod will be identified for resource type scaling. Also, this step will help in vertical autoscaling to adjust the resource request if needed.

4. Based on the result of step 4, identify the resource demand for the pod (e.g. CPU or memory).

5. Run the HPA based on identified resource from 4.

6. Collect pods and HPA logs from Stackdriver and run the horizontal RL agent to identify threshold values for the maximum number of pods and resource utilization that will minimize response time.

7. Continue from step 1 as long as the pods are deployed.

Here is the algorithm used in the step function of the RL agent Environment:

The algorithm represents the autoscaling environment where the RL agent will get the observation to perform the needed action. The algorithm starts by getting the current action from the agent along with the current system state for the minimum number of pods, the maximum number of pods, current resource utilization and current pods response time. We then repeat the autoscaling of pods until one of two conditions is met, either the current number of pods exceeds the maximum, or we achieve the target resource utilization. The algorithm calculates the new number of pods based on the Kubernetes HPA formula (2):

$$newPods = ceil\ (curPods * (curUtil/tgtUtil)) \qquad (2)$$

Input: Current Environment class (*Env*), current Agent Action (*Act*)
Output: Observation (*Obs*), reward (*r*), *isDone*
Get current agent action (*Act*).
Get *minPods*, *maxPods*, *curPods*, *curutil*, *curRsp* from current *Env.*
Apply current action *Act*, $maxPods = maxPods + Act$
$done = true$
While (*done*)
  $curPods = ceil(curPods * (curUtil / Env.target))$
  if($curPods > maxPods$)
    $done = false$
    $obs = [minPods,maxPods,curPods,curUtil,curRsp]$
  else
    collect Pods data for *curUtil* and *curRsp*
    if($curUtil < Env.target$)
      $done = false$
      $obs = [minPods,maxPods,curPods,curUtil,curRsp]$
    EndIf
  EndIf
EndWhile
$IsDone = abs(maxPods) > Env.PodsThreshold$
if $isDone = false$
  if $curRsp <= Env.SLA$
    Apply set reward *r*
  Else
    if $curRsp > Env.SLA$
      Apply set penalty
    EndIf
Else
  No state change
EndIf

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 *tgtUtil* 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 as long as the response time is less than or equal the *QoS* value.

Here is a description of the action and observation space for the RL agent:

Observation Space (S):

$$S = \{P_{min}, P_{max}, P_{cur}, U_{cur}, T_{cur}\}$$

where P_{min} is the minimum number of pods, P_{max} is the maximum number of pods, P_{cur} is the current number of pods, U_{cur} is the current resource utilization, T_{cur} is the current average response time.

Action Space (A):

$$A = \{A_{up}, A_{down}, A_{no-change}\}$$

where A_{up} is the action of increasing the number of pods by a given factor, A_{down} is the action of decreasing the number of pods and $A_{no-change}$ is taking no action for no state change.

Reward Function (R) is resembled as a constrained optimization function R to maximize the reward:

$$\begin{aligned} &max_{cur,QoS}\ R(T_{cur}, T_{QoS}) \\ &\text{Subject to } Dif(T_{cur}, T_{QoS}) >= 0 \\ &\qquad Dif(T_{cur}, T_{QoS}) = T_{QoS} - T_{cur} \end{aligned} \tag{3}$$

where T_{cur} is the current average response time, T_{QoS} is the target QoS response time.

$$R(T_{cur}, T_{QoS}) = (T_{cur} <= T_{QoS})\ ?\ R_{reward} : R_{penalty}$$

where R_{reward} is the set reward value and $R_{penalty}$ is the set penalty value.

The complexity of the algorithm can be measured based on the number of deployed pods (x) where the algorithm will find the average resource utilization and response time of the pods. 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 threshold value for pods.

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, as this is a training phase, the pods cannot go indefinitely, their number and their replicas are both finite numbers. There is a set maximum threshold value for the number of pods that can be changed based on the system demand and available resources. Adding to the fact that the agent will be deployed on a cloud cluster where scalability can be achieved at the computing resource level.

IV. EXPERIMENTAL SETUP

In this section, we briefly describe the experiments performed for the proposed modules. We based our experiment on a Twitter analytics disaster management system where disaster data is extracted from tweets in real time [2], [7]. With incoming tweets ranging on average 6000 tweets per second, processing such a huge amount of data to extract useful information during a disaster becomes a challenge. The system consists of multiple microservices as illustrated in fig. 6. We implemented the microservices as Docker containers in Python and deployed them as pods on GKE cluster [5]. When the pod is deployed on GKE, the pods will share the cluster resources including CPU, memory, and disk. The job of the HPA is to replicate the pods up or down based on their average resource utilization.

We measured the average response time for the pods from the time it received the tweets information to the time it has processed it. We will use the average response time as the metric in our evaluation as we measure the performance of the autoscaling modules. We tested three different microservices each with different resource demand; tweets preprocessing which is high on input traffic as it gets the tweets in real time, and the disaster classification which is high on memory as it classifies the tweets for relevance. As our microservices are not high on CPU, we tested the model on a CPU-intensive microservice that performs heavy mathematical calculations. We ran multiple experiments on two types of clusters for over two hours of time. A standard cluster of 3 nodes, 1vcpu and 3.75G of memory each and another one of 3 nodes, 2vcpu and 3.75G memory each were used in the experiments. Fig. 7 shows the disaster classification pod resource consumption as it was deployed on GKE. We can see that the pod is high on memory demand compared to CPU.

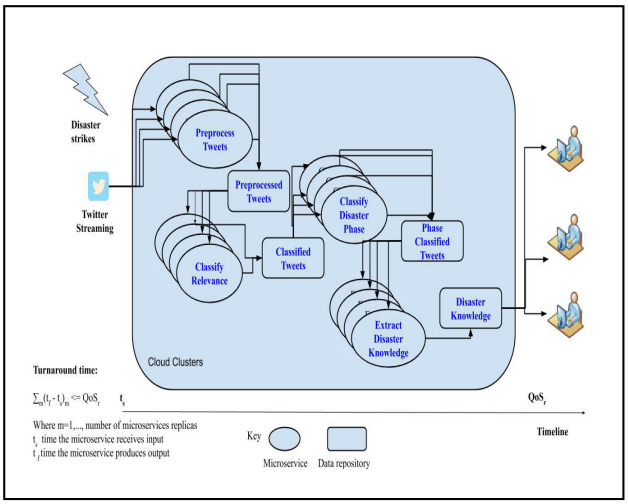

FIGURE 6. Twitter analytics disaster management system based on the microservices approach.

We tested the microservices under both normal load and heavy load with 110 maximum pods per node. We enabled Stackdriver monitoring, GKE usage and consumption logs in Big Query to send the GKE cluster log data.

For the RL agents, we developed a custom environment to simulate HPA in Matlab where the data for resource utilization and response time was taken from two pods log data for CPU intensive and memory intensive workloads. As we are simulating the HPA environment, we added a random factor on the resource usage and response time to increase the data set size and to allow the RL more exploration through the training process. The idea is to create a model locally based RL agent to allow for faster convergence instead of total open exploration in a real environment. Once we create model based RL agents, we will deploy them on real environment in GKE.

We have created and trained and validated multiple RL agents in Matlab on our simulated environment for comparison and evaluation purposes. We use the average pods response time as the metric for evaluation to validate the agents. The average response time value should be below or

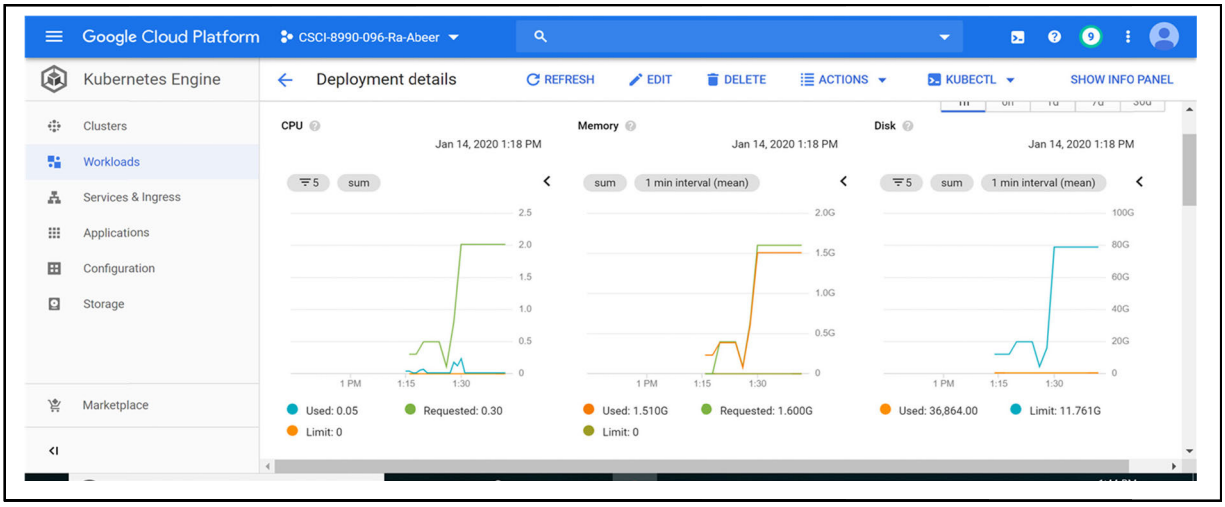

FIGURE 7. GKE deployment for disaster classification microservice showing the pod is high on memory consumption compared to CPU.

equal to the set QoS one. We chose the agents as our learning observation space is continuous, and our action space is discrete. We observe different values for the minimum number of pods, the maximum number of pods, the current number of pods, current resource utilization, and the current average response time. The action is increasing or decreasing the number of pods to achieve the autoscaling. Since our system has a continuous observation space, storing the observations and actions in lookup tables is impractical [29]. Based on that we represented the actor and critic in the agents using deep neural networks corresponding to the observation and action space dimensions of five and two, respectively.

Table 2 lists the RL agents used in the experiment.

TABLE 2. RL agents used in Matlab training along with their policy type.

RL Agent	Type
Policy Gradient	Function approximator
Q-Learning	Epsilon-greedy exploration
SARSA	Value-based
Deep Q-Network	Value-based
Actor-critic	Actor-critic
Proximal Policy Optimization	Actor-critic

V. RESULTS AND ANALYSIS

In this section, we present the results of our experiments on the different modules along with the analysis of the tests.

A. GENERIC AUTOSCALING MODULE

Our goal is to show that autoscaling based on the pod resource demand enhances the overall response time compared to the default CPU-based autoscaling. The three different microservices of Tweets preprocessing which is high on traffic I/O, disaster classification, which is high on memory, and the CPU intensive mathematical one, were all deployed on GKE and auto scaled. Running a query to identify the resource consumption to request ratio on the log data collected in Big Query over an hour of time shows that the disaster classification pod is high on memory usage compared to CPU usage by a factor of 60%.

For the CPU intensive pod which runs a mathematical calculation that is CPU intensive, running the same experiment under the same controlled constraints shows that the pod is high on CPU usage compared to memory usage by about 48% factor. Fig. 8 shows our results. This confirms that we can identify the pod high resource demand in Big Query to be

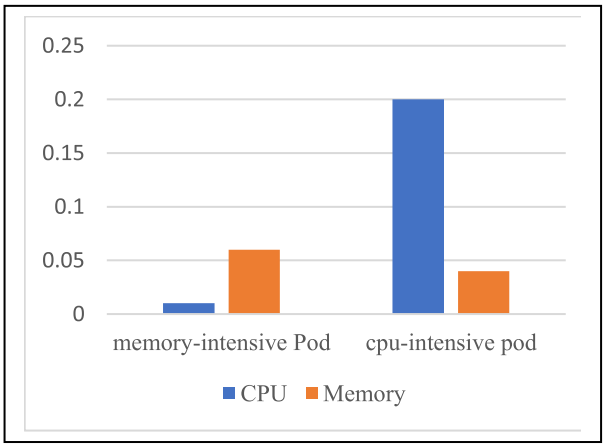

FIGURE 8. Generic pod resource demand identification shows CPU versus memory intensive pod resource consumption.

used in training our RL agent in the next step of the algorithm. This module can also be extended and worked on to provide a mean for vertical autoscaling at the container level, where once we identify the trends for resource utilization versus request, we can adjust the container resource request. Since the scope of this work is focused on horizontal autoscaling which is the most common and less costly where the number

of replicas is increasing or decreasing, we defer the vertical container autoscaling to future work.

After the resource demand is identified, we ran the generic autoscaling module by customizing the Kubernetes HPA based on the demand. Our preliminary results of implementing the algorithm on a disaster management system show that autoscaling a microservice based on its resource demand outperforms the default Kubernetes CPU-based autoscaling by a factor of 20% [7]. Table 3 presents the results of the experiment on the three different microservices with three different resource demands. Twitter disaster preprocessing which is high on the number of messages in the queue, disaster classification which is high on memory and a CPU-intensive generic microservice. We can see that the autoscaling tailored based on the microservice resource demand provides better response time compared to the default CPU-based autoscaling in Kubernetes.

TABLE 3. Average response time from the generic module for different resource-based microservices compared to default Kubernetes HPA.

Microservice Resource Demand	Model-based - Average Response time (micro sec)	Default - Average Response Time (micro sec.)
Memory	1.48	[1.5 – 1.63]
Traffic	[0.0125 – 0.02]	[0.01- 0.1]
CPU	0.139	0.179

B. RL AGENTS AUTOSCALING MODULE

We trained the Matlab RL agents for 1000 max episodes with 1000 maximum steps and an average reward of 480. After training the agents, we validated them on the simulated environment by changing some of the default values with 100 max steps and some revalidated with 1000 steps and we recorded the lowest response time we got as we noticed that after a certain time, the response time stabilizes. We will present our results on both a CPU-intensive microservice and the memory-intensive one for comparison on autoscaling on the different resources. We chose those two resources are they are the most common resources for autoscaling.

1) CPU-INTENSIVE RL AGENT AUTOSCALING

Table 4 provides the results for training the agents on the CPU intensive pod data. We can see from the results that the agents were able to minimize the response time below the QoS target while maintaining a target CPU utilization. The results can also give us the values for the other autoscaling parameters.

For example, the QValue agent was able to achieve a response time around 0.045 and a CPU utilization of 2.472 at 385 pods. This response time is below the default QoS set value of 0.3. These values can be used as parameters for the Kubernetes HPA.

Table 5 provides the results of validating the agents on the simulated environment where the default response time is changed to 0.199. We can see for example, that the Qvalue agent gave a response time during validation that is close to the trained one with a value of 0.07 at 494 pods where the default response time is 0.199 with a standard deviation of 0.01.

TABLE 4. RL agents training on CPU- intensive simulated HPA environment.

Agent	Max Pods	Current Pods	CPU utilization	Avg Response time	Standard deviation (0.01)
PG	1004	567	2.68	0.02	0.01
QValue with Deep NN	1004	385	2.472	0.045	0.035
SARA	1004	679	2.114	0.017	0.007
AC	1004	987	1.011	0.032	0.022
PPO	34	23	1.72	0.007	-0.003
DQN	1004	651	1.59	0.025	0.015

TABLE 5. RL agents validating results for CPU-intensive HPA environment.

Agent	Max Pods	Current Pods	CPU utilization	Avg Response time	Standard deviation (0.04)
PG	545	415	1.313	0.073	0.033
QValue with Deep NN	545	494	2.929	0.0781	0.038
SARSA	545	297	2.032	0.098	0.058
AC	535	371	1.02	0.02	-0.02
PPO	42	29	1.62	0.017	-0.023
DQN	545	316	1.369	0.129	0.109

We have compared the CPU utilization and the response time of the trained agent versus the validation environment.

As for CPU utilization, we can see from Fig. 9 that almost all the agents when validated got a CPU utilization value that is less or very close to the training one, and all of them got a utilization below the default one value of 3.1. Fig. 10 shows that validating the trained agent gave a response time average that is lower or close to the training one. Both the trained and validated agents gave a response time that is less than the default one of 0.199 with a standard deviation of 0.04. We can confirm that our RL agents in the simulated environment can achieve a better CPU utilization and can minimize the response time below a given set threshold value. In addition, the maximum number of pods required to get to that response time is identified.

2) MEMORY-INTENSIVE RL AGENT AUTOSCALING

For the memory intensive microservice, we deployed the disaster-classification pod, on GKE, which we showed in our previous work is memory intensive. We increased the traffic on the service and collected both the memory request utilization and the response time from Big Query and the service logs. We have trained the RL agents under the same constraints as we did for the CPU intensive service using

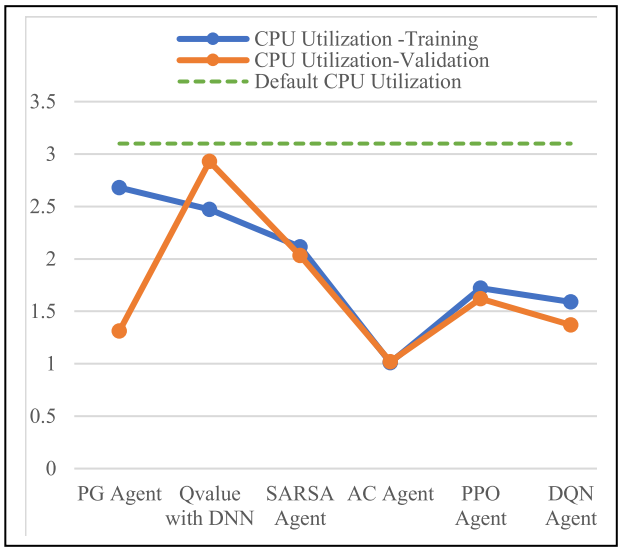

FIGURE 9. RL agents achieved pods CPU utilization for CPU intensive container.

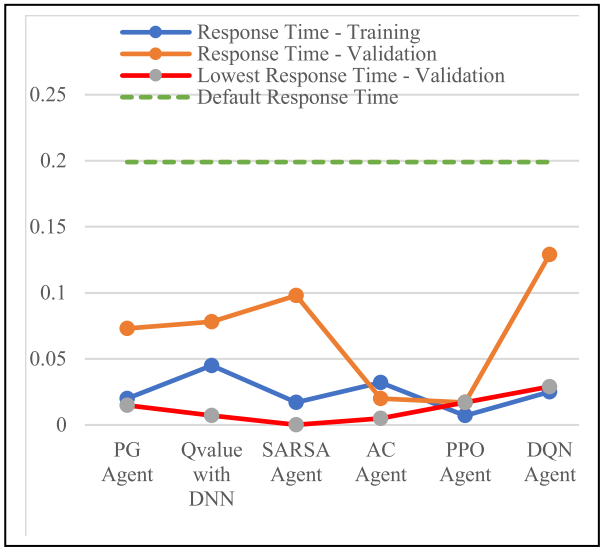

FIGURE 10. RL agents achieved average response time for CPU intensive container lower than the default.

the retrieved data for the simulated environment. We did the training on 100 steps then 1000 steps with the reward function to minimize the response time with a default QoS value of 1.5, while autoscaling based on memory utilization. Table 6 shows the results of the training with the average response time achieved, the memory request utilization and the HPA threshold values for current pods and the maximum number of pods. We can see that all the agents achieved a response time on average that is below the default one with 0.269 standard deviation.

We have then validated the agents on varying default values of response time, memory utilization, and the number of pods. We repeated the validation, and we recorded the response time as it was most stabilized with the current pods and

TABLE 6. RL agents autoscaling training results on memory-intensive service.

Agent	Max Pods	Current Pods	Memory utilization	Avg Response time	Standard deviation (0.269)
PG	1001	979	0.117	0.922	0.653
QValue with Deep NN	1001	480	0.289	1.155	0.886
SARSA	1001	575	0.188	1.483	1.214
AC	6	20	0.297	0.96	0.691
PPO	18	28	0.184	0.755	0.486
DQN	1001	550	0.206	0.665	0.396

memory utilization. Table 7 Shows the results of the validation with a standard deviation of 0.296.

TABLE 7. Validation results on RL agents for memory-intensive autoscaling.

Agent	Max Pods	Current Pods	Memory utilization	Lowest Avg Response time	Standard deviation (0.296)
PG	1001	758	0.131	0.88	0.584
QValue with Deep NN	1001	977	0.286	1.124	0.828
SARSA	1001	394	0.29	0.229	-0.067
AC	20	76	0.222	0.946	0.65
PPO	26	65	0.249	1.006	0.71
DQN	1001	472	0.227	0.642	0.346

We then made a comparison on the training and validation results for both the response time and the memory request utilization as illustrated in Fig. 11 and Fig. 12. We can see from Fig. 11 that all agents got training and validation results that are very close confirming that they can be trained to auto-scale based on memory utilization. As a result, they achieved a memory utilization that is slightly above the default target one, which is explained by the fact that our reward function was based on minimizing the response time as a main goal. Fig. 12 shows that the response time we got out of training and validation is maintained to be below the default QoS one, which is the main goal of this experiment.

The results of this experiment confirm that we can train the RL agent to auto-scale the pods based on memory utilization to reduce the response time to be less than or equal the QoS default set value. In comparison to manually setting those threshold values in the Kubernetes HPA, the value of the RL agents is apparent in identifying the autoscaling values autonomously and in an intelligent way. As an example, in this experiment, when the QoS response time was 1.5, we can see that we can train any of the agents to give a response time less than the default one and we can get the threshold values of autoscaling for the maximum number of

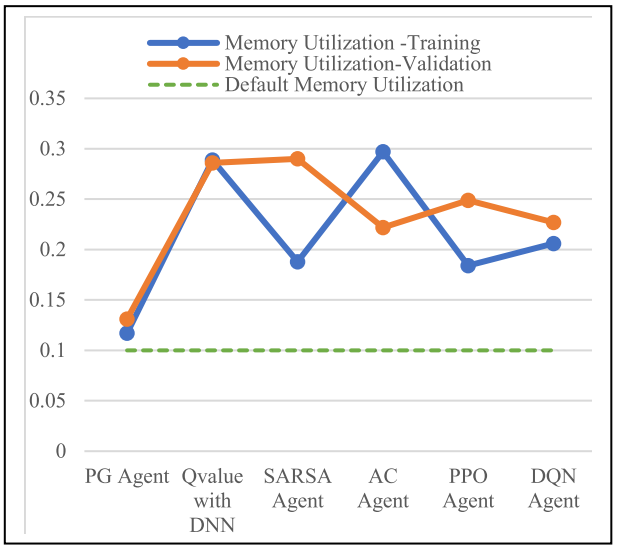

FIGURE 11. RL agents achieved pods' memory utilization for memory intensive container.

FIGURE 12. RL agents achieved average response time for memory intensive container below the default one.

pods and memory request utilization that will abide by the QoS set value.

The tests performed in the experiment are for training the agents and providing a model based RL training. The tests are not comprehensive and once we deploy the trained policy on the real GKE environment, we will gather more data and perform training in a more comprehensive manner. Initial outcomes from our experiment show promising results on being able to identify the scaling values while maintaining QoS. This satisfies the goal of our algorithm to be able to deliver those values in an intelligent and autonomous way.

VI. CONCLUSION AND FUTURE WORK

In this article, we presented a study on autoscaling microservices in the cloud for real-time applications where response time is the main QoS to be maintained. We first introduced a generic autoscaling model that will identify the highest resource demand for the microservice to be scaled. We showed that this generic model could lower the response time by a factor of 20% compared to the default Kubernetes HPA. We then presented our intelligent model based on RL agents to enhance identifying the threshold values of the autoscaling.

We utilized reinforcement learning for training and validating agents to auto-scale microservices horizontally based on resource consumption demand. As the microservices vary in their resource demand and utilization, their autoscaling needs to be performed in a smart way with a good understanding of their behavior.

In this study, we presented our findings on a disaster management application based on Twitter analytics with microservices that are memory intensive and others that are CPU intensive deployed on Google KE. Our findings on a simulated environment with real log data support the idea of training and validating RL agents to identify the threshold values for autoscaling with satisfying the QoS of response time. Our future work will focus on deploying the RL agents in the Google cloud and further evaluating and testing at a larger scale. As the focus of this work is on horizontal autoscaling, we will also be working on vertical scaling at the container level, which is more costly, but it is worth investigating the effect of vertical autoscaling versus horizontal autoscaling on the QoS such as response time.

Green computing is another important research area where reducing energy consumption is a main QoS constraint. Our focus in this study is on response time for real-time systems performance. As our autoscaling algorithm is generic in nature, different QoS constraints can be studied including workloads energy consumption. That can be another future research application for our proposed algorithm. More research can also be done on what resources to deploy on the cloud edge and how that can affect the system performance.

REFERENCES

[1] M. M. Shukla and J. Asundi, "Considering emergency and disaster management systems from a software architecture perspective," *Int. J. Syst. Eng*, vol. 3, no. 2, pp. 129–141, 2012.

[2] A. A. Khaleq and I. Ra, "Cloud-based disaster management as a service: A microservice approach for hurricane Twitter data analysis," in *Proc. IEEE Global Humanitarian Technol. Conf. (GHTC)*, Oct. 2018, pp. 1–8.

[3] S. Taherizadeh and V. Stankovski, "Dynamic multi-level auto-scaling rules for containerized applications," *Comput. J.*, vol. 62, no. 2, pp. 174–197, Feb. 2019, doi: 10.1093/comjnl/bxy043.

[4] T. Zheng, X. Zheng, Y. Zhang, Y. Deng, E. Dong, R. Zhang, and X. Liu, "SmartVM: A SLA-aware microservice deployment framework," *World Wide Web*, vol. 22, pp. 275–293, Jan. 2019, doi: 10.1007/s11280-018-0562-5.

[5] A. Abdel Khaleq and I. Ra, "Agnostic approach for microservices autoscaling in cloud applications," in *Proc. Int. Conf. Comput. Sci. Comput. Intell. (CSCI)*, Dec. 2019, pp. 1411–1415, doi: 10.1109/CSCI49370.2019.00264.

[6] *Kubernetes Monitoring*. Accessed: May 2019. [Online]. Available: https://cloud.google.com/monitoring/kubernetes-engine/

[7] A. A. Khaleq and I. Ra, "Twitter analytics for disaster relevance and disaster phase discovery," in *Proc. Future Technol. Conf.* in Advances in Intelligent Systems and Computing, vol 880, K. Arai, R. Bhatia, and S. Kapoor, Eds. Cham, Switzerland: Springer, 2018.
[8] J. V. Bibal Benifa and D. Dejey, "RLPAS: Reinforcement learning-based proactive auto-scaler for resource provisioning in cloud environment," *Mobile Netw. Appl.*, vol. 24, no. 4, pp. 1348–1363, Aug. 2019.
[9] Y. Gan and C. Delimitrou, "The architectural implications of microservices in the cloud," 2018, *arXiv:1805.10351*. [Online]. Available: http://arxiv.org/abs/1805.10351
[10] Z. Ashktorab, C. Brown, M. Nandi, and A. Culotta, "Tweedr: Mining twitter to inform disaster response," in *Proc. ISCRAM*, S. R. Hiltz, M. S. Pfaff, L. Plotnick, P. C. Shih, Eds. Harrisburg, PA, USA: The PA State Univ., 2014, pp. 354–358.
[11] S. Kho Lin, U. Altaf, G. Jayaputera, J. Li, D. Marques, D. Meggyesy, S. Sarwar, S. Sharma, W. Voorsluys, R. Sinnott, A. Novak, V. Nguyen, and K. Pash, "Auto-scaling a defence application across the cloud using docker and kubernetes," in *Proc. IEEE/ACM Int. Conf. Utility Cloud Comput. Companion (UCC Companion)*, Dec. 2018, pp. 327–334.
[12] M. Gotin, F. Lösch, R. Heinrich, and R. Reussner, "Investigating performance metrics for scaling microservices in CloudIoT-environments," in *Proc. ACM/SPEC Int. Conf. Perform. Eng.*, Mar. 2018, pp. 157–167.
[13] E. Casalicchio and V. Perciballi, "Auto-scaling of containers: The impact of relative and absolute metrics," in *Proc. IEEE 2nd Int. Workshops Found. Appl. Self* Syst. (FAS*W)*, Sep. 2017, pp. 207–214.
[14] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, Jul. 2019, pp. 68–75.
[15] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, model-driven autoscaling for cloud applications," in *Proc. 11th ICAC*, 2014, pp. 57–64.
[16] Z. Yang, P. Nguyen, H. Jin, and K. Nahrstedt, "MIRAS: Model-based reinforcement learning for microservice resource allocation over scientific workflows," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2019, pp. 122–132.
[17] S. Horovitz and Y. Arian, "Efficient cloud auto-scaling with SLA objective using Q-learning," in *Proc. IEEE 6th Int. Conf. Future Internet Things Cloud (FiCloud)*, Aug. 2018, pp. 85–92.
[18] E. Barrett, E. Howley, and J. Duggan, "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud," *Concurrency Comput., Pract. Exper.*, vol. 25, no. 12, pp. 1656–1674, Aug. 2013.
[19] S. Singh and I. Chana, "QoS-aware autonomic resource management in cloud computing: A systematic review," *ACM Comput. Surveys*, vol. 48, no. 3, pp. 1–46, Feb. 2016.
[20] M. Cinque, R. Della Corte, and A. Pecchia, "Advancing monitoring in microservices systems," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Oct. 2019, pp. 122–123, doi: 10.1109/ISSREW.2019.00060.
[21] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2019, pp. 19–33.
[22] F. Rossi, "Auto-scaling policies to adapt the application deployment in Kubernetes," in *Proc. ZEUS*, 2020, pp. 30–38.
[23] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *Proc. IEEE 12th Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2019, pp. 329–338.
[24] P. Jamshidi, A. M. Sharifloo, C. Pahl, A. Metzger, and G. Estrada, "Self-learning cloud controllers: Fuzzy Q-learning for knowledge evolution," in *Proc. Int. Conf. Cloud Autonomic Comput.*, Sep. 2015, pp. 208–211, doi: 10.1109/ICCAC.2015.35.
[25] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, "Autopilot: Workload autoscaling at Google," in *Proc. 15th Eur. Conf. Comput. Syst.*, Apr. 2020, pp. 1–16.
[26] D. Idoughi, K. A. Abdelouhab, and C. Kolski, "Towards a microservices development approach for the crisis management field in developing countries," in *Proc. 4th Int. Conf. Inf. Commun. Technol. Disaster Manage. (ICT-DM)*, Dec. 2017, pp. 1–6.
[27] Y. Garí, D. A. Monge, E. Pacini, C. Mateos, and C. García Garino, "Reinforcement learning-based application autoscaling in the cloud: A survey," 2020, *arXiv:2001.09957*. [Online]. Available: http://arxiv.org/abs/2001.09957
[28] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," in *Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2017, pp. 1487–1495.
[29] MathWorks. *Create Policy and Value Function Representation*. Accessed: Jun. 15, 2020. [Online]. Available: https://www.mathworks.com/help/reinforcement-learning/ug/create-policy-and-value-function-representations.html

ABEER ABDEL KHALEQ (Member, IEEE) received the B.S. degree in computer science from Yarmouk University, Jordan, in 1991, and the M.S. degree in foundations of advanced information technology from the Imperial College London, London, U.K., in 1992. She is currently pursuing the Ph.D. degree in computer science and information systems with the University of Colorado at Denver, Denver, CO, USA.

From 1993 to 2007, she was a computer science Faculty at various universities and academic institutions in CO, USA. From 2002 to 2003, she was a Software Developer at a private company in CO, USA. From 2008 to 2015, she was the Computer Science Department Chair at the Arapahoe Community College, CO, USA, where she developed a new computer science degree curriculum. Since 2016, she has been working on her research in the Ph.D. program with the Department of Computer Science and Engineering and teaching some courses at the University of Colorado at Denver. She has published multiple conference papers. Her research interests include cloud computing, microservices architecture, and the development of real-time applications for disaster management in the cloud.

Dr. Abdel Khaleq's awards and honors include the Graduate Assistance in Areas of National Need (GAANN) Fellowship, the CRA-WP Grad Cohort for Women, the Google Cloud Research Grant, and multiple Honorary Awards for best Faculty.

ILKYEUN RA (Member, IEEE) received the B.S. and M.S. degrees in computer science from Sogang University, Seoul, South Korea, the M.S. degree in computer science from the University of Colorado at Boulder, CO, USA, and the Ph.D. degree in computer and information science from Syracuse University, USA. He is currently an Associate Professor with the Department of Computer Science and Engineering, University of Colorado at Denver. His main research interests include cloud computing, high-performance computing, and computer networks.