Autoscaling Cloud-Native Applications using Custom Controller of Kubernetes.

MSc Research Project
Cloud Computing

Neha Deshpande
Student ID: x19203896

School of Computing
National College of Ireland

Supervisor: Sean Heeney

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Neha Deshpande
Student ID:	X19203896
Programme:	Cloud Computing
Year:	2021
Module:	MSc Research Project
Supervisor:	Sean Heeney
Submission Due Date:	16/08/2021
Project Title:	Autoscaling Cloud-Native Applications using Custom Controller of Kubernetes.
Word Count:	4958
Page Count:	20

Signature:	Neha Deshpande.
Date:	15th August 2021

Autoscaling Cloud-Native Applications using Custom Controller of Kubernetes.

Neha Deshpande
x19203896

Abstract

Cloud-native microservices-based applications are increasingly deployed using containers in the software engineering industry. When an application has massive workload, Kubernetes automatically scales the microservices. Kubernetes' default algorithm leads to improper resource allocation, appearing in performance degradation of cloud-native applications as well as increased maintenance costs. By using a custom controller, this paper concludes the appropriate number of instances for containers. By reducing maintenance costs, the proposed algorithm preserves the Quality of Service (QoS) of cloud-native applications. This study found that the default Kubernetes algorithm is more expensive than the custom controller. The maintenance cost of an application is reduced by almost 50% with custom controllers.

1 Introduction

Cloud computing is a favored technology for increasing efficiency and growing capacity. Cloud-native is a discipline that relies on how applications are run and developed to take an advantage of the cloud computing architecture. Cloud-native technologies enable the industry to deploy and develop scalable applications in dynamic and modern environments. Containers, service meshes, orchestrators, microservices, and declarative APIs allow loosely connected systems that are robust, observable, and manageable (Garces et al.; 2020).

Cloud-native appears to become the defacto standard for the industry now a days. According to the Cloud Native Computing Foundation (CNCF) recent study, "production practice of Cloud-Native projects has been increased higher than 200 percent in common from December 2017" (Aderaldo et al.; 2019).

The architectural procedure of microservices entails building the application as a set of tiny services. Each one is self-contained and performs atomic functions. Multiple microservices communicate with each other to respond to business service requests.

Microservices enables frequent and fast updates on production environment with a low impact on the end-user side because they are self-governing components that can be upgraded, deployed, restarted, scaled up, or scaled-down independently. Containerized microservices are more efficient and more durable than virtual machine-based ones. The instantiation of containers is as straightforward and agile as launching any operating system (Pahl et al.; 2019).

Scalability is the potential for an application or a resource to meet rising demand. The ability to scale up and down of the resources is one of the most highlighted features of cloud computing. The scaling system can be accomplished automatically. In the field of cloud computing, auto-scaling algorithms are becoming more and more prominent to maintain or increase the Quality of Service(QoS) (Jahan et al.; 2020).

Due to complex workload conditions, cloud computing poses difficult tasks such as the optimal allocation of resources for a given pod which is known as "autoscaling". In comparison to other container orchestration tools, Kubernetes provides the most automatic scaling mechanism, Horizontal and Vertical Pod Autoscaling (HPA and VPA). Since a pod contains one or a few tightly coupled containers, it establishes the smallest deployment entity in Kubernetes (Balla et al.; 2020). Below figure 1 shows the structure of Horizontal Pod Autoscaling by Kubernetes.

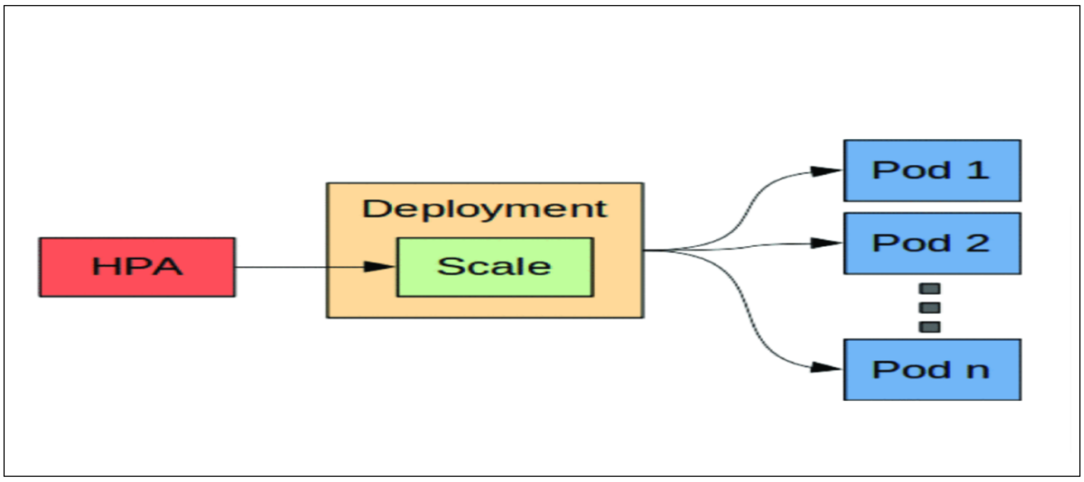
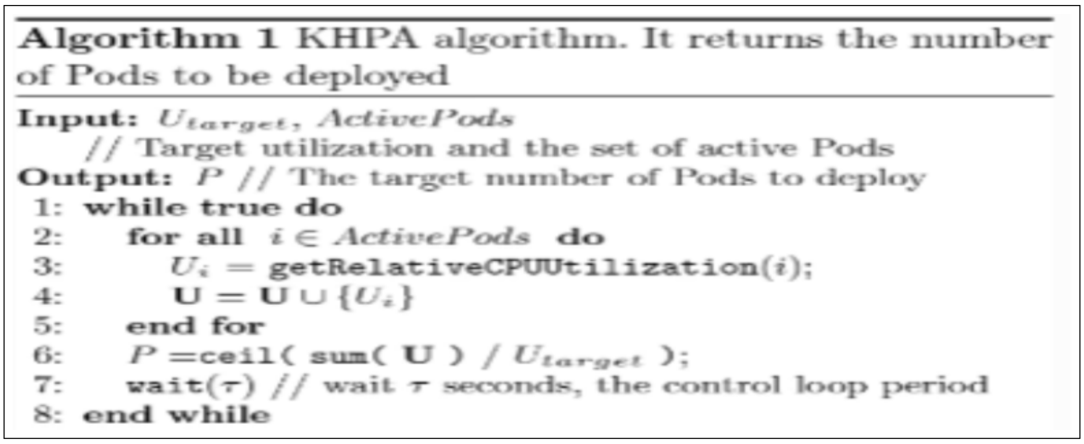


Figure 1: Horizontal Pod Auto-scaling by kubernetes. (Balla et al.; 2020)

As an input, Kubernetes' default algorithm measures the effectiveness of CPU utilization and the number of active pods on the cluster. Furthermore, its output returns several pods that are supposed to be deployed (Kubernetes (2021)). This algorithm calculates the CPU usage is dependent on other processes. The cAdvisor tool contains CPU usage, and it preserves the CPU usage in Vector U. Lastly, the algorithm determines how many pods need to be deployed. Figure 2 shows the default Kubernetes algorithm (KHPA) given by Kubernetes.



Algorithm 1 KHPA algorithm. It returns the number of Pods to be deployed

```
Input: $U_{target}$, $ActivePods$
    // Target utilization and the set of active Pods
Output: $P$ // The target number of Pods to deploy
1: while true do
2:     for all $i \in ActivePods$ do
3:         $U_i$ = getRelativeCPUUtilization($i$);
4:         $\mathbf{U} = \mathbf{U} \cup \{U_i\}$
5:     end for
6:     $P$ =ceil( sum( U ) / $U_{target}$ );
7:     wait($\tau$) // wait $\tau$ seconds, the control loop period
8: end while
```

Figure 2: Kubernetes horizontal pod autoscaler(KHPA)

1.1 Motivation

Nowadays, service-oriented architecture (SOA) and microservice architecture are conventional methods of combining software applications. Developing these applications by different teams makes them easy to maintain. Applications based on microservices are more advantageous because their development method involves a well-defined structure of interface through which requests from a microservice can be made of another microservice (Suram et al.; 2018).

When autoscaling CPU computing resources, some methods may use static rules to increase or decrease the containers. If the cloud-based applications have dynamic workloads, then static rules do not support basic or static scenarios. The QoS (Quality of Service) of the cloud-native applications could be negatively affected. Due to the default algorithm of the Kubernetes(Horizontal Pod Autoscaling (KHPA)), large applications based on microservices regularly suffer from resource under-provisioning or over-provisioning that results in resource waste and performance degradation.

Due to the resource wastage, the maintenance cost of the application increases. According to the number of container instances, autoscaling must constantly allocate resources appropriately. Because of this issue, a novel autoscaling algorithm is proposed in this research to maintain the Quality of Service (QoS) of a cloud-native application by reducing the cost.

1.2 Research Question

Can the Quality of Service (QoS) of the microservice-based applications be maintained by reducing a cost using a custom controller for Kubernetes?

Cloud computing field is becoming more and more dependent on autoscaling algorithms to guarantee the Quality of Services (QoS) of the cloud-native applications. Due to complex workload conditions, it is difficult to determine how many pods a deployed application should get, as well as how much autoscaling to perform. Through this research an attempt is made to perform autoscaling by using kubernetes controllers. In this research Kubernetes deployments are controllers that are providing updates for pods and replica sets. Also, deployments describes a collection of identical, multiple pods with no individual status.

This research helps to make decision of scaling up and down according to the requests coming to the application. Depending on the target and current CPU utilization the next pod has been scheduled through the custom controller of the Kubernetes cluster.

1.3 Structure Of The Paper

The remaining paper is designed as follows. Section 2 outlines the Related work linked to the microservices autoscaling practicing Kubernetes. Section 3 manifests the methodology for achieving the intended approach. Section 4 gives the design specification of the proposed approach. Section 5 will give a detailed description of the implementation. and Section 6 and 7 give the evaluation results and conclusion respectively.

2 Related Work

The research on Kubernetes and microservices is very active at the moment. There is no denying that the use of cloud-native designs improves performance in almost every application. The cloud is nearly being adopted across all IT industries.

2.1 Microservices Background

2.1.1 Microservice Architecture

There are several characteristics that all cloud-native applications share, as Gannon et al. (2017) has stated. According to their research, the most recognized technique to build an application is to showcase the tendency of an application is microservice architecture. The computing standard splits the application into tiny pieces described as microservices, which interact with each other via RPC or by requesting web services. In their investigation, they describe the representation of cloud-native applications. They mention that microservices are a type of cloud-native application that can be containerized or dynamically orchestrated. Extending the investigation by Toffetti et al. (2015), they are examining how to combine administration abilities into cloud-based applications employing orchestration for achieving stable and stateless actions under functionalities that are management-related by attaching cloud-native design patterns. Nonetheless, the dynamic variations in the workload of the cloud-native application reduce the Quality of Service (QoS).

2.1.2 Microservice Performance

On AWS, Pelle et al. (2019) estimated the performance of a latency-sensitive cloud-native application. They converge on the components that cause a delay in cloud-native applications because they have approached latency-sensitive applications. Cloud platform latency increases as invocation times and data access times increase. Moreover, they quoted that picking the appropriate cloud services concerning cloud-native applications while they are being designed can reduce latency and improve overall performance. Likewise, For time-sensitive applications, Polona Štefanič et al. (2019) presented a framework as element of their SWITCH project. In SWITCH, you can immediately create software components using a docker-compose file and develop workflows and logic for applications. Notwithstanding the appropriate service provider or tool being chosen, cloud-native applications are affected by an increase in workload, which degrades the performance of the application.

2.2 Container state-of-the-art

2.2.1 Microservice Containerization

Virtual machines already controlled the cloud computing market, but Docker has accomplished prominence in the IT industry. Various investigations have been conducted to compare the performance of Docker and other technologies. Using genetic algorithms, Guerrero et al. (2018) implemented a microservice container allocation scheme in cloud environments. To balance the load, reduce the network overhead, and ensure service reliability, they have studied these methods. Also Christina Terese Joseph and Chandrasekaran (2020) advised that cluster stability and load balancing should be the purpose. These studies have proposed hybrid container allocation, which directs to the ineffective allocation of resource allocation among container instances.

A mathematical equation was developed that scales the cloud containers to maximize computing resource use and to reduce the response time by Zhang et al. Zhang et al. (2019). By continuing the work that is involved with Zhang et al. (2019), Pérez et al. (2018) developed an algorithm for determining a proper container for each task and furthermore migrated it to AWS Lambda. Response times were lowered while ensuring QoS requirements were met with their proposed approach. For task mapping to the appropriate containers, Pérez et al. (2018) also devised a container-based task scheduling policy. It reduced the processing time and response time for the tasks, while more efficiently utilizing resources. In this study, the primary objective is to minimize the processing time, which results in the delay, but the algorithms are limited to the AWS Lambda service provided by AWS.

2.2.2 Container Orchestration

Containerization provides the benefit of dividing an application into self-managed packages that can be managed anywhere, as well as dividing the dependencies of these packages. Development cycles become extra agile by using containers. Docker is the most extensively used container in cloud computing, despite the fact that there are many potential implementations. The building blocks for microservice systems are clusters of containerized service instances. fault-tolerant, distributed, and highly available are important aspects of the containers. In Khan (2017), they present abilities that container orchestration tools should incorporate. In their research, they outlined a design for recognizing key mechanisms for methods to implement container orchestration and orchestration frameworks. They have also described recommended and openly available orchestration platforms in their approach. The research in Khan (2017) and Jawarneh et al. (2019) designates that compared to other competing orchestration tools, Kubernetes outperforms Docker Swarm, Apache Mesos, and Mesosphere for the deployment of complex applications. These analyses tried to fulfill the gap in deciding an orchestration tool that is most suited for IT managers. The researchers conclude that Kubernetes works best for deploying complex applications, whereas other orchestration tools work best for simpler ones.

On the other hand, orchestration tools may be useful for applications that require a lot of computing power, allowing a quicker response time and improved QoS. We will orchestrate containers with Kubernetes for this study since it is more performing for complex applications. Using Kubernetes to automate cloud-native applications will be described in the following section.

2.3 Autoscaling Microservices

In auto-scaling, resources are dynamically provisioned based on changes in applications or environments. This is particularly useful during periods of high workload. There are several challenges in autoscaling, and the researcher proposes a taxonomy for web applications. An autoscaling survey has been proposed by Thai et al. (2018), the survey covered scaling approaches, resource estimation, threshold-based metrics, and container-based multi-tier application scaling and autoscaling. For building Bag-Of-Task applications in the public cloud, the Verma and Bala (2021) presents a resource optimization taxonomy and survey.

Rossi et al. (2020) In their research, the author looked at autoscaling approaches, including the building of auto-scaling algorithms. As per their findings, Kubernetes Horizontal Pod autoscaling functions as a threshold-based reactive controller that adapts the necessary resources by an application via a closed-loop. Jindal et al. (2017) The possibilities of Kubernetes autoscaling were exhibited, and the researchers found that the autoscaling approach did not produce a higher performance based on CPU and memory use. They did not, however, explore the influence of other indicators on microservice-based applications under severe demand.

In this research conducted by Yin et al. (2018), the author looked at a number of performance measures for multi-layered cloud applications. According to the findings, CPU utilization is a suitable metric if the features of microservices are consistent, but it degrades the performance of input/output heavy microservices when the workload varies. The influence of absolute vs relative measurements in microservice autoscaling was discussed by the researcher. They stated that absolute indicators like CPU usage allow for more exact choices on CPU-intensive applications than the relative metrics used by the default Kubernetes autoscaling method. However, they only looked at the effect of CPU-intensive microservices and ignored the other metrics.

Using the containerization approach, created an efficient scheduling policy. Casalicchio (2019) policy's principal objective is to decrease processing time. Their policy is used as a monitoring approach to locate an acceptable resource for finishing the execution in the shortest amount of time and installing the microservices container. Their study aids in the distribution of load across all resources and the effective utilization of resources. This technique, on the other hand, can assist in load distribution, but only for a limited number of pods or resources. This method may minimize response time and enhance the Quality of Service if each pod has achieved its maximum CPU use.

2.4 Allocation of Pods using Kubernetes

Kubernetes is one of the most suited options for resource allocation in cloud-native applications to improve QoS. In Al-Dhuraibi et al. (2017), the author proposed a technique for allocating enough resources to satisfy the growing demands of cloud-native applications. Autonomic Cloud Computing resource scaling is the technique, which utilizes a constant value for a collection of resource-level measures including CPU usage. The workload can be categorized as light or heavy depending on whether the resource need exceeds the predetermined value. Only Virtual Machines are supported by this resource scaling architecture.

Complex cloud-native apps, on the other hand, cannot be created on tiny servers or workstations with low capabilities. In **?**, the ELASTICDOCKER framework was offered by the researchers as the first device to dynamically provide vertical elasticity to docker containers. Their strategy is based on IBM's MAPE-K guidelines (Monitor, Analyse, Planning, Execute, Knowledge). ELASTICDOCKER adjusts the amount of CPU and RAM allotted to each container based on the application workload. As vertical elasticity is restricted by the machine's capacity, ELASTICDOCKER executes live container migration when there are insufficient resources. Their quoted method surpasses Kubernetes' elasticity by 37.63%. Their solution, however, is reliant on particular functions, such as Linux's CRIU (Checkpoint Restore In Userspace) capability. The proposed method is not adaptable to all operating systems.

Pelle et al. (2019), the author described an auto-scaling technique for containerized applications that employs an adaptive feedback controller to dynamically scale necessary resources. EcoWare agent component should be loaded in each VM to complete this research. The Ecoware agent manages the container-specific data, such as CPU Utilization. This component is also in command of launching and terminating containers in VMs, as well as modifying resource allocations. This autoscaling approach, however, is only suitable for online apps. In addition, deploying an ECoWare agent for each container adds to the overhead. The challenge of scheduling containerized microservices across several Virtual Machines was explored in Al-Sharif et al. (2017). Their strategy was to minimize overall traffic and turnaround time for the full end-to-end operation.

Bhamare et al. (2017) In their technique, scheduling is accomplished by ranking, which focuses on optimal resource usage by ordering activities according to their computing needs. Both studies, although, failed to handle dynamic resource delivery. Static resource provisioning may result in unexpected failures.

2.5 Comparative Analysis Of The Algorithms

Reference	Algorithm/ Framework	Approach	Advantages	Limitations
Jindal et al. (2017)	APMT Framework	An innovative strategy for measuring autoscaler performance.	Measured the execution of auto scalers with a new solution	Not fitting for I/O concentrated microservices
Al-Sharif et al. (2017)	ACCRS Framework	Advanced fault disclosure for improving the Utilization level	Reduced Power and Cost consumption	This framework supports only Virtual Machines.
Casalicchio (2019)	KHPA-A	Introduced algorithm for auto-scaling	Early prediction of workload	Limited only for static workload
This Research	Custom Controller for auto-scaling	Autoscaling microservice-based application	Reduce cost for maintaining Quality of Service(QoS)	Supports only CPU intensive microservices

Table 2.1: Comparative analysis of the algorithms.

3 Methodology

The methodology of this research is discussed in this section. The process flow of research is described in 3.1 and 3.2 gives the overview of tools and technologies used in this research.

This study contributes to the state-of-the-art performance evaluation of autoscaling containers practicing CPU-intensive workloads. To eliminate resource waste or application failure, an autoscaling solution is provided in this study. For managing the containerized workloads Kubernetes an open-source platform is used in this study.

In this research, a custom controller is created to achieve better performance and maintain the Quality of Service of the microservice-based application. The custom controller is responsible for dynamically managing and scaling the pods depending on the containerized workload. Those pods will be scheduled depending on the requests coming towards the server. The controller acts upon the standard resources of Kubernetes. The below diagram shows the system architecture of the Kubernetes. Figure 3 shows all the standard components of kubernetes cluster.

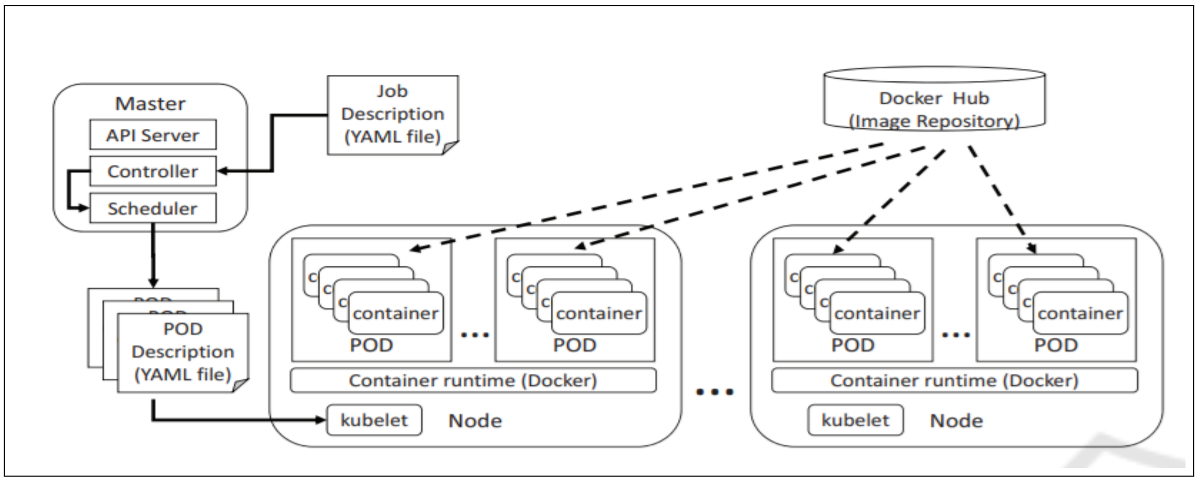

Master
API Server
Controller
Scheduler
Job Description (YAML file)
Docker Hub (Image Repository)
POD Description (YAML file)
container
POD
Container runtime (Docker)
kubelet
Node

Figure 3: Components of the kubernetes cluster. (Lin et al.; 2019)

3.1 Process flow of research

An Amazon Web Services t3.2xlarge instance was used for all operations carried in this study. This microservice-based cloud-native application has been downloaded from the Kubernetes website (Kubernetes; 2021). A docker container is used to deploy the application, which is developed in PHP. There is one microservice inside the application that performs computing-intensive computations for the application.

The Kubernetes cluster has been created using "Kubeadm" and then the dockers are utilized to containerize the microservice. The microservices are deployed on the Kubernetes cluster and later the demanded number of pods has been generated using the custom controller. The load generator has been given by Kubernetes to generate the load on the application.

The following figure 4 shows the process flow of the research.

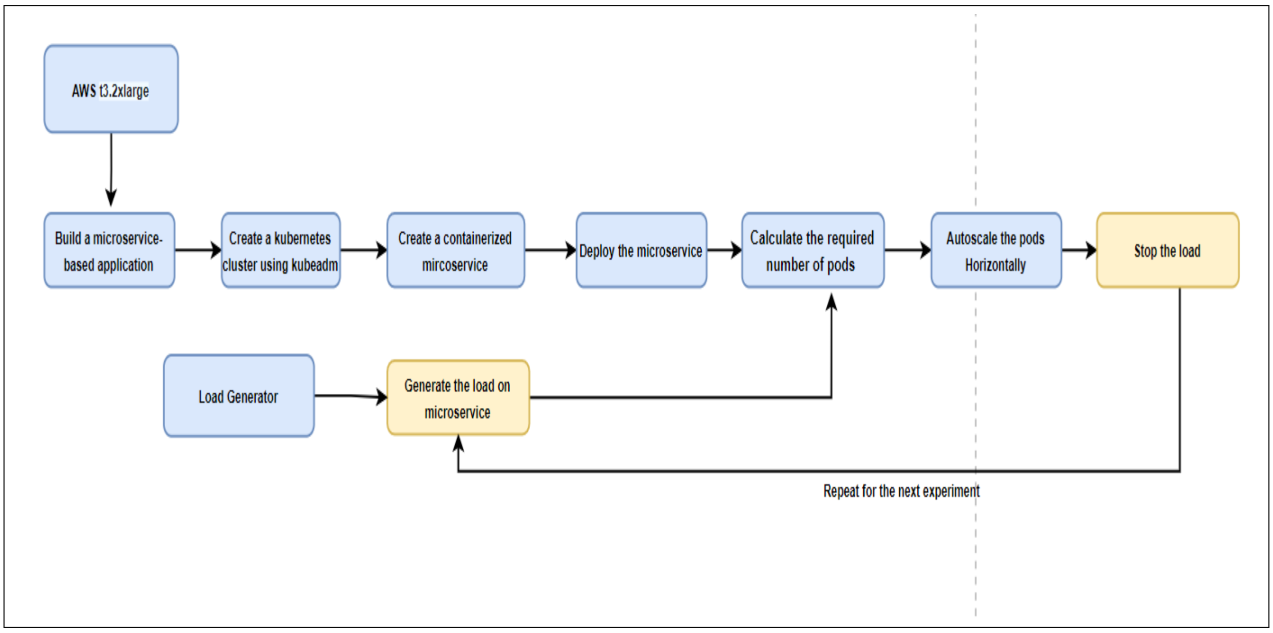

AWS t3.2xlarge
Build a microservice-based application
Create a kubernetes cluster using kubeadm
Create a containerized mircoservice
Deploy the microservice
Calculate the required number of pods
Autoscale the pods Horizontally
Stop the load
Load Generator
Generate the load on microservice
Repeat for the next experiment

Figure 4: Process Flow Of The Research

3.2 Tools and Technologies Used In Research

This study was conducted on an instance of **AWS EC2 t3.2xlarge (Ubuntu)**. Kubernetes **HPA(Horizontal Pod Autoscaler)** was used for deployment and autoscaling. The microservice-based application is developed in **PHP** language to perform the computing intensive operations.

To carry out the research in this study, following technologies and tools have been used:

- **Kubernetes :** This research uses the latest version of Kubernetes (1.18). There are one master node and two worker nodes in a kubernetes cluster.
- **Kubeadm :** A Kubernetes cluster is installed with Kubeadm in this study. A cluster can be bootstrapped using Kubeadm. All configuration files provided by Kubernetes can be accessed through Kubeadm..
- **Kubectl :** Kubernetes cluster commands can be given with the Kubectl tool. A deployment of the application has been made possible by Kubectl. In addition, Kubernetes logs are provided by Kubectl.
- **Docker :** Containerization of microservices is done with Docker. The version of Docker used in this study is 20.10.6. Containers are used to store microservices. Containers will be scheduled as pods.
- **PHP-Apache :** Apache is the web server that handles requests and process them via HTTP. PHP-Apache enables dynamic content creation with PHP.

3.3 Assessment Carried Out In This Research

A total of three experiments is conducted to obtain consistency in the results of the evaluation of the custom controller. Dedicated time frames of two minutes, eight minutes, and fifteen minutes are used for evaluating the 3 experiments after the workload is generated.

It has been examined here whether Kubernetes autoscaler is scheduling the necessary number of pods or not. Validation takes place against the default Kubernetes(KHPA) algorithm. Observations were made based on the current and target CPU utilization.

4 Design Specification

The specification used for designing this project is discussed in this section. proposed Specification of the system is described in 4.1. 4.2 gives the description of proposed architecture of the system and 4.3 gives the architecture of custom controller.

4.1 Proposed Specification Of The System

As part of this research, Docker lightweight containers are being used. When microservices are containerized, they can be restarted after failing or when they are updated. An orchestration tool named Kubernetes is used in this research to manage containers' operations and deployments. Containers inside the cluster are managed by Kubernetes. Three VMs are used to run a Kubernetes cluster on the AWS cloud. In this experiment, Network Time Protocol (NTP) has been used to achieve asynchronous communication between the nodes in the cluster. The following table 3 and table 2 shows the hardware specifications for kubernetes cluster and Virtual machine respecitvely.

Kubernetes Cluster		
Instance	AWS	Config=t3.2xlarge
Operating System	Ubuntu	Version = 20.04
Orchestration tool	Kubernetes	Version = 1.18
Container engine	Docker	Version = 17.09

Table 2: Required configuration for the cluster.

Virtual Machine	
vCPU	8
Memory	32GiB
Network Performance	upto 5 gbps
Cost	$0.3341/ hr

Table 3: Required configuration for the Virtual machine.

4.2 Proposed Architecture Of The System

In order to use a Kubernetes cluster, there is a need to install the kubeadm package. In this study, kubeadm 1.19 was used. The version of kubectl 1.21 is installed in order to run the command throughout the cluster. There are two worker nodes and one master node in a Kubernetes cluster. Containers have been used in the worker node so that pods can be organized. Component **etcd** maintains the cluster state between the master and the worker node.

The master node contains three components, such as **Controller Manager, API server, Scheduler**. Managing Kubernetes core functions lies with the controller manager. Controlling Kubernetes pods and moving them from a shared state to a desired state is the responsibility of the controller. When changes are needed, the Kubernetes controller makes the necessary requests. Microservices and pods are configured and verified with the API server. Kubernetes policies are stored in a scheduler.

A worker node consists of **Docker, proxy system, and Kublet**. Kublet is the communication protocol between the master and worker nodes. By virtue of its role in managing the master node's state, the Kublet also administers pods according to the pod specification provided by the master node in **PodSpec**. Following figure 8 shows the high level architecture of the overall system.

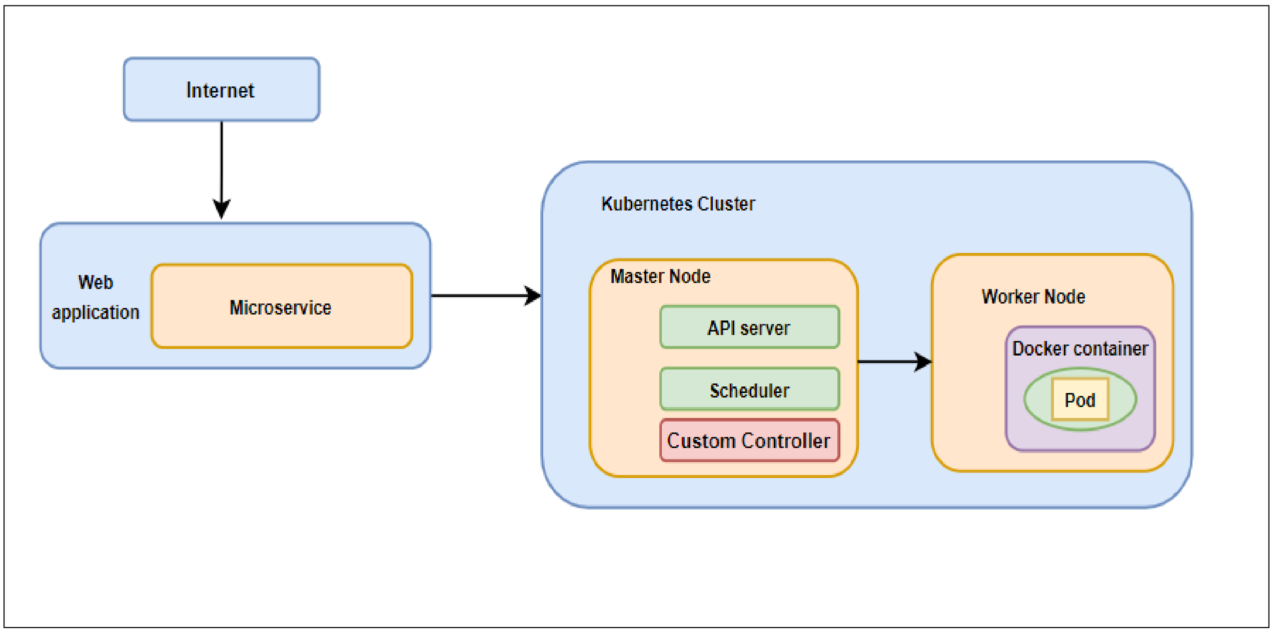


Figure 5: High Level Architecture Of System

4.3 Architecture of proposed custom controller

In this research, an attempt is made to write a custom controller to schedule the required number of pods to overcome resource wastage and website crashes. The system has been designed in a way that each pod contains only one container, and each container appears as a microservice. Based on the number of pods that are currently running inside the cluster, the custom controller takes action. There is a component called Kube-controller-manager.YAML in the Kubernetes cluster. Cluster state is monitored by Kube-controller-manager. The kube-controller-manager guarantees that the new replica of an application pod is built when one of the pods is missing from an application.

The Kube API server must be used to change the cluster's state if any changes need to be made by the custom controller. It enables interaction with other components as a fundamental management component. YAML files of Kube-API server communicates with the custom controller through API. To control replicas, the parameters in the YAML file are transferred to the Kube-API server, and then to the Kube-controller manager. Next, the Kube-API server informs the custom controller to start or create new pods.

Next, then Kube-proxy authorizes pods in a cluster to communicate internally and externally. Individually pod holds its IP address linked with it. Kube-proxy uses the IP address to shift the traffic to the pods from the cluster. Consequently, when the load on the application increases, the Horizontal Pod Autoscaler automatically scales up or down according to the actions taken by the custom controller.

5 Implementation

The implementation of the research is discussed in this section. It is described in 5.1 that the autoscaler is implemented in the custom controller, and in 5.2 that the control loops are implemented in the custom controller.

5.1 Implementation of Autoscaler

Algorithm 1 *Autoscaling Algorithm for resource allocation.*

```
Input: Total_Pods,Total_CPU_Usage_Value,Total_CPU_target_value;
Output: Total_Pods_n = Total number of pod to be scheduled.
Total_Pods; = sum(pod0,pod1,.....Podn));   // Calculates the total number of pods
  running in cluster
Size_of_cluster = Total_Pods.length;
Total_CPU_target_value = fetch_target_CPU();  // API call for fetching the target CPU
Total_CPU_Usage_Value = fetch_current_usage();   // API call for fetching the current
  CPU usage.
if Size_of_cluster > 0 && Total_CPU_Usage_Value > (Size_of_cluster
  *Total_CPU_target_value) then
    for i in Total_Pods do
        Total_Pods_n = Total_CPU_Usage_Value / Total_CPU_target_value   // Calculate
          the total number of pods.
    end
end
```

In many studies (Khazaei et al.; 2020),(Wang et al.; 2017) and (Cardellini et al.; 2016) its been observed that, even under predictable bursting workload scenarios when examining the dynamics of the underlying computing environment, microservice-based applications enter into an unstable state with slight inconstancy. One critical cause of this situation is taking numerous autoscaling operations during a heavy workload scenario this causes further auto-scaling burden, which renders the system into uncertain state.

In this research along with the adaption control loops(Explained in the 5.2), the autoscaler is designed in such a way that, if the CPU utilization is reached to the given Total_CPU_Usage_Value (In this study 50% of target CPU_utlization is used) such as 50% and if there are two containers running which has average CPU-Utilization of 40% and 45% respectively, so while autoscaling the pods will be calculating as follows :

$$= [(40\% + 45\%) \ / \ 50\%] = 50\%/85\% = 1.7 \approx 2.$$

So the 2 pods are required to scale up. The proposed autoscaler requires Total_Pods, Total_CPU_Usage_Value ,Total_CPU_target_value. The API from API_Server returns the total value of utilization of the CPU and the current value of the CPU. Algorithm 1 shows the pseudo code inside the custom controller.

5.2 Implementation of control loops for the custom controller

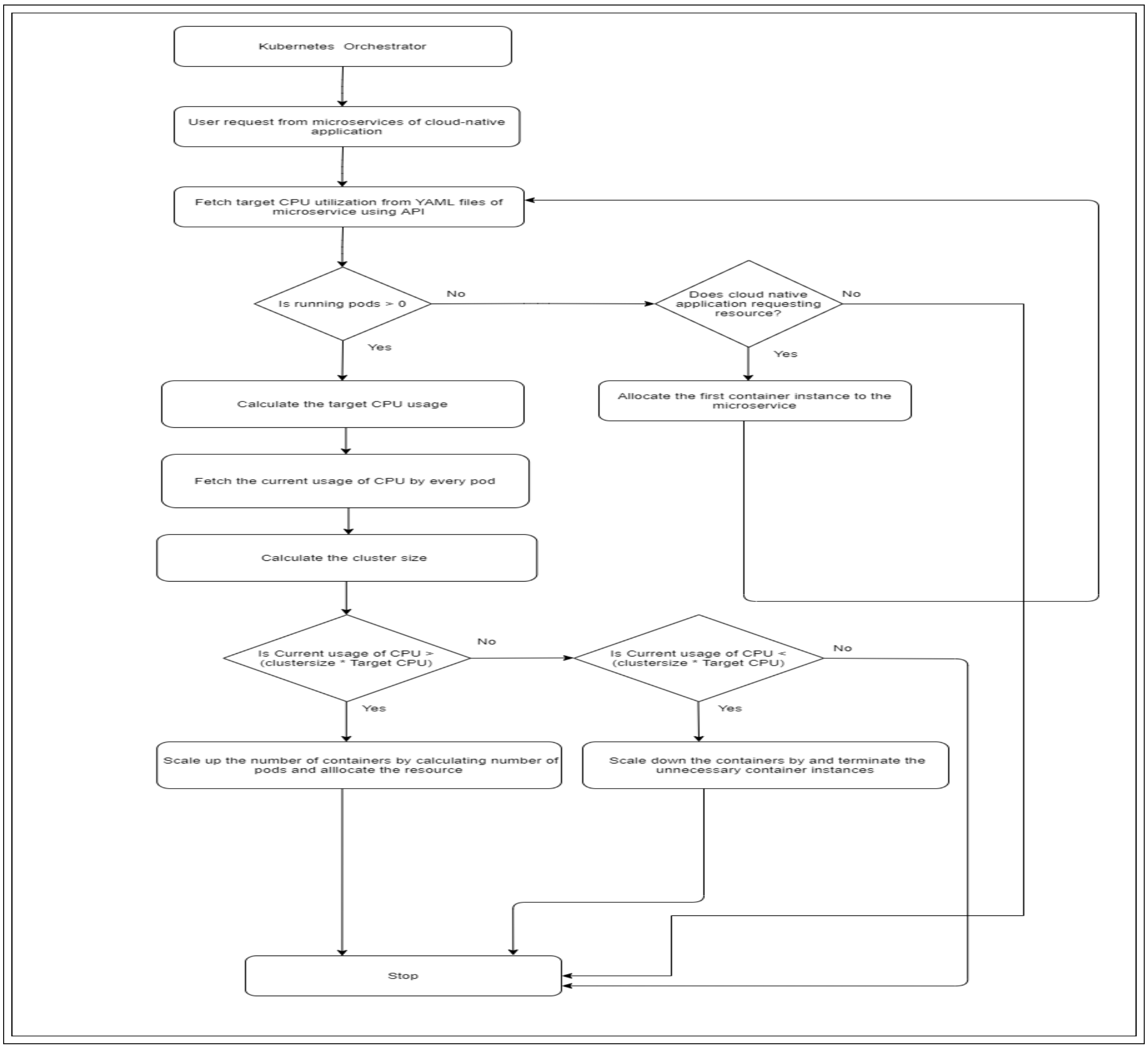

Kubernetes Orchestrator
User request from microservices of cloud-native application
Fetch target CPU utilization from YAML files of microservice using API
Is running pods > 0
No
Yes
Does cloud native application requesting resource?
No
Yes
Calculate the target CPU usage
Allocate the first container instance to the microservice
Fetch the current usage of CPU by every pod
Calculate the cluster size
Is Current usage of CPU > (clustersize * Target CPU)
No
Yes
Is Current usage of CPU < (clustersize * Target CPU)
No
Yes
Scale up the number of containers by calculating number of pods and alllocate the resource
Scale down the containers by and terminate the unnecessary container instances
Stop

Figure 6: Flow diagram of custom controller

Adaption of the control loops plays a major role in implementation of this research. A controller's adaptation interval is defined as the minimum period within two consecutive adaptation operations over time. It is also called as 'Control Loops' of the controller. In the execution environment, the adaptation interval is kept longer than the time taken to start up a container instance. Adaptation decisions were made when a system is quite steady. The entire system continues to run smoothly, even in the heavy workload scenario.

6 Evaluation

6.1 Performance Evaluation

The performance of the custom controller is evaluated through a series of experiments. The values and comparisons is done against the default horizontal pod autoscaler algorithm. During the experimentation process, each experiment was repeated three times to obtain the exact number of pods and to verify the results in order to have greater validity. As a result, each experiment was run three times. A t3.2xlarge instance and a Kubernetes cluster running 1.18 were used in the evaluations and experiments. Calculations are then performed after the load has been generated. After generating the load, all observations were taken after two, eight, and fifteen minutes.

A load is generated on the application once it is ready to handle requests in all the experiments. Observations have been noted after generating the workload. The target CPU utilization was set to 50%, which means when the CPU utilization of one pod reaches 50%, an autoscaling decision should be made, and an extra pod should be scheduled to handle the further workload. To schedule the pods, the minimum replica count is set to was one, and the maximum count was set to ten. It has been verified that the replica count of the autoscaler is zero during each experiment since Kubernetes replicas take time to stabilize. Furthermore, the command 'kubectl get pod' has confirmed that there should only be one pod running before the generation of the workload, both after each experiment has been run and before each experiment has been run. Since the current deployment (application) is running on only one pod, 'kubectl get hpa' shows the output that one pod is running.

6.2 Experiment 1

The first experiment is observed after the two minutes of workload generation. After two minutes the CPU usage was reached to the 18% and the target cpu usage was 50%. At this time only one pod was running since the application is deployed on the same pod. according the algorithm inside the controller. The comparison of the results between default kubernetes algorithm and custom controller is shown in below table 4.

Using the algorithm inside the custom controller, the number of pods required to handle the workload should be as follows:

$$\mathbf{Total_{Pods}} = \mathbf{(18\% \ / \ 50\%)} = \mathbf{0.36} \approx 0.$$

Number of pods scheduled after 2 minutes	
Default Kubernetes HPA	**Custom Controller**
5	**0**

Table 4: Result comparison after experiment one

6.3 Experiment 2

After eight minutes of workload generation, the second experiment is observed. After eight minutes, the CPU usage reached 106%, while the target CPU usage was 50%. Below table 5 compares the results of default Kubernetes algorithm and custom controller.

Number of pods scheduled after 8 minutes	
Default Kubernetes HPA	**Custom Controller**
7	**2**

Table 5: Result comparison after experiment two

$$\mathbf{Total_{Pods} = (106\% \ / \ 50\%) = 2.12} \approx 2.$$

6.4 Experiment 3 (Complex one)

After fifteen minutes of workload generation, the third experiment is observed. The CPU usage has reached to 250% after fifteen minutes and the target CPU usage was 50%. Below table 6 compares the results of the default Kubernetes algorithm and custom controller. .

Number of pods scheduled after 15 minutes	
Default Kubernetes HPA	**Custom Controller**
1	**5**

Table 6: Result comparison after experiment fifteen

$$\mathbf{Total_{Pods} = (250\% \ / \ 50\%) = 5\ .}$$

For the application to run and handle this complex workload, five pods were needed. According to the default Kubernetes algorithm, only 1 pod was scheduled and 5 pods were scheduled by the custom controller, as expected.

6.5 Discussion

After generating the workload, three experiments were conducted on various durations, and at every stage, the results invariably vary.

During the first experiment, there was no need to schedule an additional pod, and auto-scaling was not needed; one pod was sufficient to handle the workload. Kubernetes' default scheduling algorithm has scheduled four extra pods which were unnecessary in this scenario. In this scenario, if the Quality of Service (QoS) of the application needs to be maintained, there is a need to pay the extra cost of four extra instances. However, the custom controller does not make a decision about autoscaling since the current CPU usage is below the target CPU usage. Due to the adaptability of the control loops, this is possible with custom controllers.

During the second experiment, the CPU utilization was 106%, which meant the third pod was necessary to handle the extra workload, yet Custom Controller only calculated two pods, and default Kubernetes calculated seven. In both cases, the results are not ideal since the Kubernetes algorithm requires the cost of four additional instances when there was only a need for three. By not calculating the third pod, the custom controller is unable to handle the workload.

During the third experiment, CPU usage reached 250%. This means that this complex workload condition required a total of 5 pods. Default Kubernetes schedules only one pod, which was definitely causing the performance of the application to degrade due to it being incapable of handling the workload. However, the custom controller has scheduled the precise number of required pods so that the application's Quality of Service(QoS) remains intact.

The below figure shows the comparison between pods scheduled by Custom controller and Kubernetes horizontal pod autoscaler algorithm.

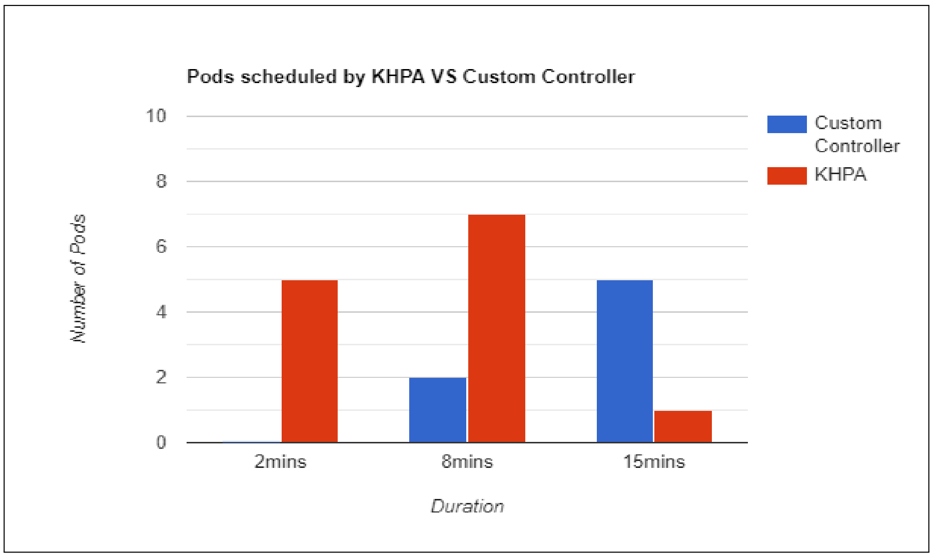


Figure 7: Pods scheduled by custom controller Vs. KHPA

One instance(t3.2xlarge) used in this project costs \$0.3341/hr. If we consider the application is up for 8 hours in a day, then ideally in normal workload condition there is a need to pay 3 dollars. The calculation is as follows.

$$\$0.3341 * 8 = \$2.6728 \approx \$3$$

The below graph shows the cost comparison graph of KHPA VS Custom Controller.

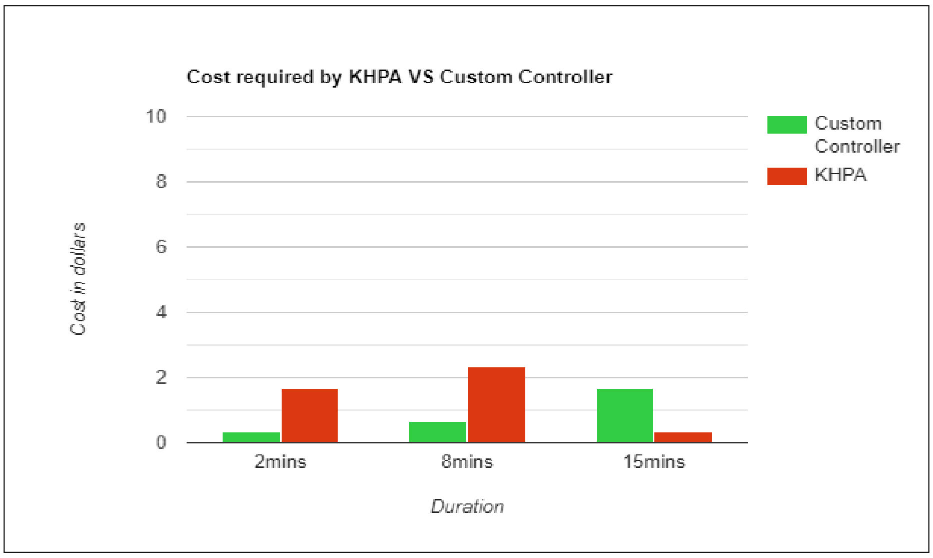


Figure 8: Cost Comparison between custom controller Vs. KHPA

This graph compares the costs between KHPA and Custom controller when workloads are generated for two minutes, eight minutes, and 15 minutes, respectively. KHPA incurred a higher cost than what is incurred with a custom controller.

7 Conclusion and Future Work

An application can be scaled up or down automatically based on the amount of traffic it generates. In heavy workload conditions, autoscaling makes the application scalable and sustainable. Currently, cloud computing is booming with microservice-based cloud applications. Docker can be used to containerize microservices. Containerized microservices can now be autoscaled with Kubernetes.

Kubernetes Horizontal Pod Autoscaling (KHPA), the default algorithm proposed by Kubernetes, causes resource overprovisioning and underprovisioning, leading to high maintenance costs and performance degradation. By using a custom controller which makes the autoscaling decisions and schedules the pods, this research attempts to solve the issue of inefficient pod allocation to applications. Pods in this application can be calculated using the algorithm in the custom controller, which lowers maintenance costs. Default Kubernetes algorithm has been observed to be more expensive than custom controllers in this study.

Since replicas take some time to stabilize, this problem can be considered in the future, and thus accurate pod calculations in every scenario will be possible. In addition, the algorithm inside the custom controller can be enhanced for memory and storage intensive microservices so that it will support all type of microservices in the future.

References

Aderaldo, C. M., Mendonça, N. C., Schmerl, B. and Garlan, D. (2019). Kubow: An architecture-based self-adaptation service for cloud native applications, *Proceedings of the 13th European Conference on Software Architecture - Volume 2*, ECSA '19, Association for Computing Machinery, New York, NY, USA, p. 42–45. Core Rank = A.
URL: *https://doi.org/10.1145/3344948.3344963*

Al-Dhuraibi, Y., Paraiso, F., Djarallah, N. and Merle, P. (2017). Autonomic vertical elasticity of docker containers with elasticdocker, *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, Honolulu, HI, USA, pp. 472–479. Core Rank = B.

Al-Sharif, Z. A., Jararweh, Y., Al-Dahoud, A. and Alawneh, L. M. (2017). Accrs: autonomic based cloud computing resource scaling, *Cluster Computing* **20**(3): 2479–2488. JCR Impact Factor: 3.458(2019).

Balla, D., Simon, C. and Maliosz, M. (2020). Adaptive scaling of kubernetes pods, *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, Budapest, Hungary, pp. 1–5. Core Rank = B.
URL: *https://doi.org/10.1109/NOMS47738.2020.9110428*

Bhamare, D., Samaka, M., Erbad, A., Jain, R., Gupta, L. and Chan, H. A. (2017). Multi-objective scheduling of micro-services for optimal service function chains, *2017 IEEE International Conference on Communications (ICC)*, Paris, France, pp. 1–6. Core Rank = A.

Cardellini, V., Grassi, V., Lo Presti, F. and Nardelli, M. (2016). Optimal operator placement for distributed stream processing applications, *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pp. 69–80.

Casalicchio, E. (2019). A study on performance measures for auto-scaling cpu-intensive containerized applications, *Cluster Computing* **22**(3): 995–1006. JCR Impact Factor: 3.458 (2019).

Christina Terese Joseph and Chandrasekaran, K. (2020). Intma: Dynamic interaction-aware resource allocation for containerized microservices in cloud environments, *Journal of Systems Architecture* **111**: 101785. JCR Impact Factor: 2.552.
URL: *https://www.sciencedirect.com/science/article/pii/S1383762120300758*

Gannon, D., Barga, R. and Sundaresan, N. (2017). Cloud-native applications, *IEEE Cloud Computing* **4**(5): 16–21. JCR Impact Factor:4.393.

Garces, L., Martinez-Fernandez, S., Graciano Neto, V. V. and Nakagawa, E. Y. (2020). Architectural solutions for self-adaptive systems, *Computer* **53**(12): 47–59. JCR Impact Factor: 4.419.
URL: *https://doi.org/10.1109/MC.2020.3017574*

Guerrero, C., Lera, I. and Juiz, C. (2018). Genetic algorithm for multi-objective optimization of container allocation in cloud architecture, *Journal of Grid Computing* **16**(1): 113–135. JCR Impact Factor: 2.095 (2019).

Jahan, S., Riley, I., Walter, C., Gamble, R. F., Pasco, M., McKinley, P. K. and Cheng, B. H. (2020). Mape-k/mape-sac: An interaction framework for adaptive systems with security assurance cases, *Future Generation Computer Systems* **109**: 197–209. JCR Impact Factor: 6.125.
URL: *https://www.sciencedirect.com/science/article/pii/S0167739X19320527*

Jawarneh, I. M. A., Bellavista, P., Bosi, F., Foschini, L., Martuscelli, G., Montanari, R. and Palopoli, A. (2019). Container orchestration engines: A thorough functional and performance comparison, *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, Shanghai, China, pp. 1–6. Core Rank = A.

Jindal, A., Podolskiy, V. and Gerndt, M. (2017). Multilayered cloud applications autoscaling performance estimation, *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2)*, Kanazawa, Japan, pp. 24–31. Core Rank = A.

Khan, A. (2017). Key characteristics of a container orchestration platform to enable a modern application, *IEEE Cloud Computing* **4**(5): 42–48. JCR Impact Factor: 4.393.

Khazaei, H., Mahmoudi, N., Barna, C. and Litoiu, M. (2020). Performance modeling of microservice platforms, *IEEE Transactions on Cloud Computing* .

Kubernetes (2021). Kubernetes. [Online]. Available: https://kubernetes.io. [Accessed: 09- Aug- 2021].
URL: *https://kubernetes.io/*

Lin, C.-Y., Yeh, T.-A. and Chou, J. (2019). Dragon: A dynamic scheduling and scaling controller for managing distributed deep learning jobs in kubernetes cluster., *CLOSER*, pp. 569–577.

Pahl, C., Brogi, A., Soldani, J. and Jamshidi, P. (2019). Cloud container technologies: A state-of-the-art review, *IEEE Transactions on Cloud Computing* **7**(3): 677–692.
URL: *https://doi.org/10.1109/TCC.2017.2702586*

Pelle, I., Czentye, J., Dóka, J. and Sonkoly, B. (2019). Towards latency sensitive cloud native applications: A performance study on aws, *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, Milan, Italy, pp. 272–280. Core Rank = B.

Polona Štefanič, Matej Cigale, Jones, A. C., Knight, L., Taylor, I., Istrate, C., Suciu, G., Ulisses, A., Stankovski, V., Taherizadeh, S., Salado, G. F., Koulouzis, S., Martin, P. and Zhao, Z. (2019). Switch workbench: A novel approach for the development and deployment of time-critical microservice-based cloud-native applications, Vol. 99, pp. 197–212. JCR Impact Factor: 6.125.
URL: *https://www.sciencedirect.com/science/article/pii/S0167739X1831094X*

Pérez, A., Moltó, G., Caballer, M. and Calatrava, A. (2018). Serverless computing for container-based architectures, *Future Generation Computer Systems* **83**: 50–59. JCR Impact Factor: 6.125.
URL: *https://www.sciencedirect.com/science/article/pii/S0167739X17316485*

Rossi, F., Cardellini, V., Lo Presti, F. and Nardelli, M. (2020). Geo-distributed efficient deployment of containers with kubernetes, *Computer Communications* **159**: 161–174. JCR Impact Factor: 2.816.
URL: *https://www.sciencedirect.com/science/article/pii/S0140366419317931*

Suram, S., MacCarty, N. A. and Bryden, K. M. (2018). Engineering design analysis utilizing a cloud platform, *Advances in Engineering Software* **115**: 374–385. JCR Impact Factor: 3.884.
URL: *https://www.sciencedirect.com/science/article/pii/S0965997817303733*

Thai, L., Varghese, B. and Barker, A. (2018). A survey and taxonomy of resource optimisation for executing bag-of-task applications on public clouds, *Future Generation Computer Systems* **82**: 1–11. JCR Impact Factor: 3.458(2019).
URL: *https://www.sciencedirect.com/science/article/pii/S0167739X17305071*

Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F. and Edmonds, A. (2015). An architecture for self-managing microservices, *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, AIMC '15, Association for Computing Machinery, New York, NY, USA, p. 19–24. Core Rank = B.
URL: *https://doi.org/10.1145/2747470.2747474*

Verma, S. and Bala, A. (2021). Auto-scaling techniques for iot-based cloud applications: a review, *Cluster Computing* pp. 1–35. JCR Impact Factor: 3.458(2019).

Wang, N., Varghese, B., Matthaiou, M. and Nikolopoulos, D. S. (2017). Enorm: A framework for edge node resource management, *IEEE transactions on services computing* .

Yin, L., Luo, J. and Luo, H. (2018). Tasks scheduling and resource allocation in fog computing based on containers for smart manufacturing, *IEEE Transactions on Industrial Informatics* **14**(10): 4712–4721. JCR Impact Factor: 9.112.

Zhang, F., Tang, X., Li, X., Khan, S. U. and Li, Z. (2019). Quantifying cloud elasticity with container-based autoscaling, *Future Generation Computer Systems* **98**: 672–681. JCR Impact Factor: 6.125.
URL: *https://www.sciencedirect.com/science/article/pii/S0167739X18307842*