Evaluating Horizontal Autoscalings on Minikube and Amazon Web Service

Fabio A. T. Madueno, Joseph A. P. Quino, Jonathan Nunes

***Abstract*—This study evaluates the performance of Horizontal Pod Autoscaling (HPA) in Kubernetes environments using Minikube and Amazon Elastic Kubernetes Service (EKS). Kubernetes is an open-source platform that enables the management and scaling of containerized applications. HPA is a key feature that dynamically adjusts the number of pod replicas to handle fluctuating workloads. The study compares the autoscaling behavior on Minikube, a local Kubernetes setup, with that on AWS EKS, a cloud-based service, focusing on their ability to scale resources in response to varying CPU loads. Experiments involved deploying a PHP-Apache server and applying artificial load to observe HPA's effectiveness. Results indicate that both environments successfully scale resources, but with variations in scalability limits, highlighting the trade-offs between local and cloud-based Kubernetes implementations.**

***Index Terms*—Kubernetes, Horizontal Pod Autoscaler, Minikube, Amazon EKS, Automatic Scaling, Microservices, Cloud Computing.**

I. Introduction

Cloud computing is a disruptive way of providing computing resources over the Internet to any other connected device. Cloud computing consists of offering on-demand access to shared resources, such as storage and applications, through data sharing. There are three types of services that the cloud can provide: Infrastructure-as-a-Service (IaaS), Platform- as-a-Service (PaaS) and Software-as-a-Service (SaaS)[1].

As the cloud computing services demand increases, the more efficient the process needs to be. In this context, clusters play a significant role. Instead of concentrating resources in a supercomputer, clusters a cluster links multiple computers via a high-speed network to distribute the workload. Clustering advantages include high availability, since if one machine breaks the system is not affected due the redundancy of nodes, and load balance, since we can control how the distribute the load between the nodes [2].

Another key element in cloud computing is the container. Container is known as an application package unit, or a virtualization on a operating system (OS) level [3]. It means that containers allow more than one application running in the same machine (node) with an isolated environment, as if it was a separate machine from the OS up, while still sharing the same hardware.

Many modern cloud applications use both conecpts to provision their resources. Instead of clustering the machines, they cluster containers, which increases flexibility on the resource allocation. Kubernetes is one example of container clustering—a popular open-source platform for managing, deploying and scaling container-based applications [4].

Kubernetes uses lightweight containers to build and decompose applications into small, independent modules known as micro-services, each responsible for a specific task of an application. This architecture allows the managing system to scale the resources for those micro-services based on the demand [4], a feature known as autoscaling. In Kubernetes, the smallest deployable unit, consisting of one or more co-located containers operating as a single unit, is called a pod.

There are three types of autoscaling methods [5]

- Horizontal scaling: Increases or decreases the number of pod replicas to share the load. This method defines the maximum load per pod and the minimum and maximum number of allowed pods.
- Vertical scaling: Adjusts resource allocation by redeploying containers with more or fewer resources as the load increases or decreases.
- Hybrid scaling: Combines horizontal and vertical scaling techniques, typically using vertical scaling to optimize resource allocation for a pod and horizontal scaling to replicate pods.

Given the importance of autoscaling, the main goal of this work is to evaluate the performance of Kubernetes horizontal autoscaling within two different frameworks:

- Amazon EKS: A cloud service offered from Amazon Web Services (AWS) to provide Elastic Container Service for Kubernetes (EKS).
- Minikube: A tool for creating and managing Kubernetes clusters in a single node or machine.

II. Methodology

This section outlines the tools, technologies, and procedures utilized to conduct the experiments to evaluate the Horizontal Pod Autoscaler. A GitHub repository was established to store the scripts and configurations used in the experiments https://github.com/fabiotorresmad/cluster-autoscaling. The evaluation is divided into two phases: the first involves experimentation on a local computer using Minikube, while the second focuses on experimentation within a cloud environment using AWS EKS.

A. Minikube Procedure

The methodology is based on the tutorial *Horizontal Pod Autoscaler Walkthrough* [6], which involves setting up a deployment on a local computer using Minikube. The Horizontal Pod Autoscaler (HPA) is then configured to scale the deployment based on CPU usage. An external script is employed to

simulate load on the deployment, and subsequently, the script is terminated. This subsection details the steps necessary to reproduce the experiment.

a) System Specifications.: The local experiments were conducted on a computer running macOS distribution, Sonoma 14.5, running in an Apple M1 processor featuring 8 cores and 16 GB of RAM. The versions of the software tools used are as follows: `minikube` v1.33.1, `kubectl` client v1.30.3, `kubectl` server v1.30.3, and `kustomize` v5.0.4.

1) Kubernetes Cluster Configuration: The experiment was conducted on a local Kubernetes cluster using Minikube. The Minikube cluster is initiated by running the command in Listing 1. This command creates a cluster with three nodes: two worker nodes and one control plane node.

```
minikube start --nodes 3 -p multinode --cpus 3 --
    memory 2048
```

Listing 1: Starting the Minikube cluster

2) PHP Apache Server Deployment: The next step is to deploy a PHP Apache server on the Minikube cluster. The deployment is detailed in Appendix A. The command shown in Listing 2 deploys the PHP Apache server. The YAML file used describes two resources: a deployment that runs a PHP server from the image `registry.k8s.io/hpa-example` and a service that exposes the deployment.

```
kubectl apply -f https://k8s.io/examples/
    application/php-apache.yaml
```

Listing 2: Deploying the PHP Apache server

3) Horizontal Pod Autoscaler Configuration: The Horizontal Pod Autoscaler (HPA) is configured to scale the PHP Apache server deployment based on CPU usage. The command in Listing 3 creates an HPA that scales the deployment to maintain an average CPU utilization of 50%.

```
kubectl autoscale deployment php-apache --cpu-
    percent=50 --min=1 --max=10
```

Listing 3: Configuring the Horizontal Pod Autoscaler

4) Load Testing: To evaluate the HPA, a script is employed to simulate load on the PHP Apache server deployment. The command in Listing 4 runs the script in a loop to increase CPU usage.

```
kubectl run -i --tty load-generator --rm --image=
    busybox:1.28 --restart=Never -- /bin/sh -c "
    while sleep 0.01; do wget -q -O- http://php-
    apache; done"
```

Listing 4: Simulating the load

B. AWS EKS procedure

This subsection outlines the step-by-step procedure to implement a Horizontal Pod Autoscaler (HPA) in an Amazon Elastic Kubernetes Service (EKS) cluster, based on the official AWS tutorial *Horizontal Pod Autoscaler Walkthrough* [7]. All commands the was used to execute this procedure are described in Appendix B.

1) Environment Setup: To begin, ensure that the environment is correctly configured. The prerequisites include having the AWS Command Line Interface (CLI) and ‘kubectl‘ installed and properly configured. Additionally, ensure that the IAM user or role has the necessary permissions to interact with Amazon EKS.

First, verify that the EKS cluster is up and running by updating the kubeconfig. Next, it is essential to deploy the Kubernetes Metrics Server, which is a prerequisite for the Horizontal Pod Autoscaler to function correctly. The Metrics Server collects metrics about resource usage and exposes them to the Kubernetes API server.

Deploy the Metrics Server and verify the deployment by checking the status of the Metrics Server.

2) Deploying a Sample Application: With the environment set up, the next step is to deploy a sample application that will be scaled by the Horizontal Pod Autoscaler. The AWS tutorial uses a simple PHP-Apache web server application.

Deploy a sample PHP-Apache web server application:

```
kubectl apply -f https://k8s.io/examples/application
    /php-apache.yaml
```

Listing 5: Deploying PHP application

This command applies a YAML file that defines the deployment and service specifications for the PHP-Apache application.

3) Configuring the Horizontal Pod Autoscaler: Once the application is deployed, the next step is to create the Horizontal Pod Autoscaler. The HPA automatically adjusts the number of pods in a deployment based on observed CPU utilization or other select metrics.

Create the Horizontal Pod Autoscaler for the deployed application, the HPA is configured to maintain the average CPU utilization of the pods in the ‘php-apache‘ deployment at 50%. If the CPU utilization exceeds this target, the HPA will scale up the number of pods, with a minimum of 1 and a maximum of 10 replicas.

4) Testing the Autoscaler: To validate the functionality of the Horizontal Pod Autoscaler, generate a load on the application using the following command:

```
kubectl run -i --tty load-generator --rm --image=
    busybox --restart=Never -- /bin/sh -c "while
    sleep 0.01; do wget -q -O- http://php-apache;
    done"
```

Listing 6: Autoscaler test

This will continuously send requests to the PHP-Apache application, increasing its CPU usage and triggering the HPA to scale the pods. Observe the number of pods to ensure they scale appropriately:

```
kubectl get pods
```

Listing 7: Get the number of pods

5) Verification and Cleanup: Finally, verify that the Horizontal Pod Autoscaler is functioning as expected by analyzing the HPA status and the number of pods in the deployment. Once the testing is complete, it is important to clean up the resources to avoid unnecessary AWS charges.

III. Results

A. Minikube Results

After executing the commands in Listing 1, Minikube successfully creates a three-node cluster, comprising two worker nodes and one control plane node, as illustrated in Figure 1.

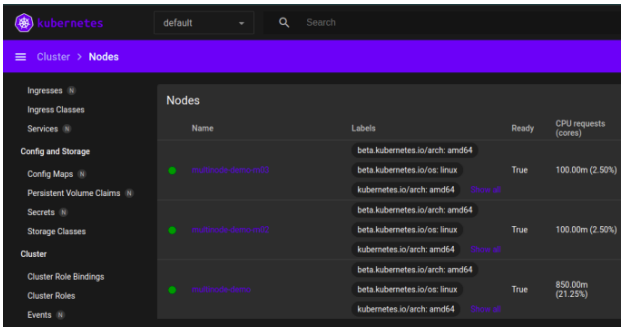

Fig. 1: Minikube cluster with three nodes.

Following the execution of the commands in Listing 2, the PHP Apache server is deployed on the Minikube cluster. The Kubernetes dashboard confirms the successful creation of the deployment and service, as depicted in Figures 2 and 3, respectively.

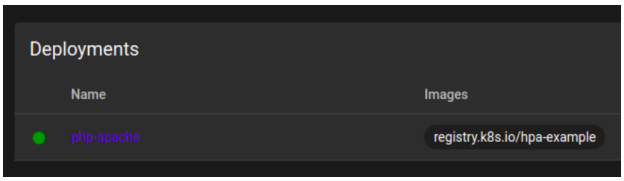

Fig. 2: PHP Apache server deployment.

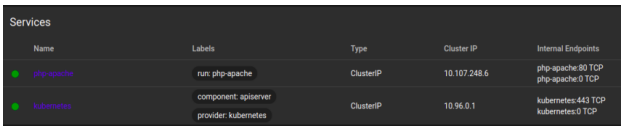

Fig. 3: PHP Apache server service.

Next, the Horizontal Pod Autoscaler (HPA) is configured to scale the PHP Apache server deployment based on CPU utilization. A load test is then performed to artificially increase the CPU usage of the server. As shown in Figure 4, the HPA increases the number of replicas from one to four when CPU usage exceeds 50%. In Figure 5, the HPA reaches its configured maximum of six replicas. Finally, after the load test is terminated, the CPU usage decreases, prompting the HPA to scale down the number of replicas from six to one, as illustrated in Figure 6.

```
❯ kubectl get hpa php-apache --watch
NAME         REFERENCE                  TARGETS         MINPODS   MAXPODS   REPLICAS
php-apache   Deployment/php-apache      cpu: 0%/50%     1         10        1
php-apache   Deployment/php-apache      cpu: 0%/50%     1         10        1
php-apache   Deployment/php-apache      cpu: 211%/50%   1         10        1
php-apache   Deployment/php-apache      cpu: 211%/50%   1         10        4
```

Fig. 4: HPA increasing the number of replicas.

```
❯ kubectl get hpa php-apache --watch
NAME         REFERENCE                  TARGETS         MINPODS   MAXPODS   REPLICAS
php-apache   Deployment/php-apache      cpu: 46%/50%    1         10        6
```

Fig. 5: HPA scaling to the maximum number of replicas.

```
❯ kubectl get hpa php-apache --watch
NAME         REFERENCE                  TARGETS         MINPODS   MAXPODS   REPLICAS
php-apache   Deployment/php-apache      cpu: 0%/50%     1         10        6
php-apache   Deployment/php-apache      cpu: 0%/50%     1         10        6
php-apache   Deployment/php-apache      cpu: 0%/50%     1         10        1
php-apache   Deployment/php-apache      cpu: 0%/50%     1         10        1
```

Fig. 6: HPA decreasing the number of replicas.

B. AWS EKS results

After the execution of the commands to test the autoscaler, as outlined in Listing 6, The output of this command is presented in Figure 8.

```
→  src git:(main) ✗ kubectl run -i \
    --tty load-generator \
    --rm --image=busybox \
    --restart=Never \
    -- /bin/sh -c "while sleep 0.01; do wget -q -O- http://php-apache; done"
If you don't see a command prompt, try pressing enter.
E0807 23:34:37.976020   24564 websocket.go:296] Unknown stream id 1, discarding message
                                                                                OK!OK!OK!OK!O
K!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK
!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!
OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!O
```

Fig. 7: Test autoscaler on EKS.

Next, we allowed approximately 2.5 minutes to elapse before proceeding with the command that displays the number of replicas managed by the autoscaler, as shown in Listing 7. The output of this command is presented in Figure 8, where it is evident that nine replicas of the server were generated in response to the load induced by the test command.

```
→  ~ kubectl get hpa
NAME         REFERENCE                  TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache      cpu: 72%/50%    1         10        9          2m36s
→  ~
```

Fig. 8: PHP Apache server metrics.

IV. Conclusion

This study evalueted the Horizontal Pod Autoscaling (HPA) from Kubernetes on two different scenarios: Minikube and Amazon EKS. While both environments effectively manage dynamic workloads, Minikube is better suited for local development due to its limited scalability, since it is limited to the machine resources, whereas EKS excels in production scenarios that demand high performance and reliability. The choice between these platforms should align with the specific needs of the application. Minikube is ideal for rapid testing, development and learning, while EKS is preferable for applications requiring robust, scalable, and consistent performance in production. Future work could explore optimizing HPA configurations for different workloads, as well as investigating other Kubernetes environments and scaling strategies to further refine the decision-making process for deployment environments.

References

[1] A. G. Prajapati, S. J. Sharma, and V. S. Badgujar, "All about cloud: A systematic survey," in *2018 International Conference on Smart City and Emerging Technology (ICSCET)*. IEEE, Jan. 2018.

[2] N. Sadashiv and S. M. D. Kumar, "Cluster, grid and cloud computing: A detailed comparison," in *2011 6th International Conference on Computer Science & Education (ICCSE)*. IEEE, Aug. 2011.

[3] T. Siddiqui, S. A. Siddiqui, and N. A. Khan, "Comprehensive analysis of container technology," in *2019 4th International Conference on Information Systems and Computer Networks (ISCON)*. IEEE, Nov. 2019.

[4] K. Senjab, S. Abbas, N. Ahmed, and A. u. R. Khan, "A survey of kubernetes scheduling algorithms," *Journal of Cloud Computing*, vol. 12, no. 1, Jun. 2023.

[5] M.-N. Tran, D.-D. Vu, and Y. Kim, "A survey of autoscaling in kubernetes," in *2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN)*. IEEE, Jul. 2022.

[6] Jul 2024. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/

[7] Jul 2024. [Online]. Available: https://docs.aws.amazon.com/eks/latest/userguide/horizontal-pod-autoscaler.html

Appendix A
Kubernetes Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
spec:
  selector:
    matchLabels:
      run: php-apache
  template:
    metadata:
      labels:
        run: php-apache
    spec:
      containers:
      - name: php-apache
        image: registry.k8s.io/hpa-example
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: 500m
          requests:
            cpu: 200m
---
apiVersion: v1
kind: Service
metadata:
  name: php-apache
  labels:
    run: php-apache
spec:
  ports:
  - port: 80
  selector:
    run: php-apache
```

Appendix B
EKS Deployment Command List

```
% Update kubeconfig for EKS cluster
aws eks --region us-east-1 update-kubeconfig --name
    my-cluster
% Deploy Metrics Server
kubectl apply -f https://github.com/kubernetes-sigs/
    metrics-server/releases/latest/download/
    components.yaml
% Check Metrics Server status
kubectl get deployment metrics-server -n kube-system
% Deploy PHP-Apache Application
kubectl apply -f https://k8s.io/examples/application
    /php-apache.yaml
% Create Horizontal Pod Autoscaler
kubectl autoscale deployment php-apache --cpu-
    percent=50 --min=1 --max=10
% Simulate load
kubectl run -i --tty load-generator --rm --image=
    busybox --restart=Never -- /bin/sh -c "while
    sleep 0.01; do wget -q -O- http://php-apache;
    done"
% Check HPA status
kubectl get hpa
% Get Pods
kubectl get pods
% Delete deployment and HPA
kubectl delete deployment php-apache
kubectl delete hpa php-apache
% Delete Metrics Server
kubectl delete -f https://github.com/kubernetes-sigs
    /metrics-server/releases/latest/download/
    components.yaml
```