ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

ANALYSIS OF DEPLOYMENT OF SCALABLE SERVICE USING KUBERNETES IN CLOUD ENVIRONMENT

GARIBALDY HUMPHREY WATULINGAS[1], YANTO SETIAWAN[2]

[1]Computer Science, Bina Nusantara University, Jakarta, Indonesia

[2]Computer Science, Bina Nusantara University, Jakarta, Indonesia

E-mail: [1]garibaldy.watulingas@binus.ac.id, [2]yanto.setiawan@binus.ac.id

ABSTRACT

Cloud computing technology has developed very rapidly, and it is no longer exclusively being used by large companies but also by the individual level that everyone can this technology relatively easily. Some of the main advantages that drive the growth in the application and penetration of cloud computing technology in the market are cost efficiency, flexibility, reliability, and scalability. With the implementation of scalability, users can efficiently utilize the computing resources provided by cloud service providers to obtain maximum performance at minimal cost. This research will analyze the case where cloud computing technology can be used to apply a scalable service using Kubernetes running in a cloud environment and provide evidence for the performance improvement that can be achieved from implementing scalability.

Keywords: *Cloud Computing, Container Orchestration, Scalability*

1. INTRODUCTION

Cloud computing technology has experienced a very rapid development in the last two decades. Starting with Amazon Web Services which was launched in 2002, and then followed by other big companies that also launched their cloud service providers such as Google Cloud Platform in 2008, Microsoft Azure in 2010, Alibaba Cloud in 2009, IBM Cloud in 2011, Oracle Cloud in 2012, and many others [1]. According to a report issued by Fortune Business Insights in 2023, the market value of cloud computing services in 2022 is estimated to be 569.31 billion USD with a projected growth value of 677.95 billion USD in 2023, and 2,432.87 billion USD in 2030. The biggest factor for this significant increase in market penetration for cloud computing technology in recent years was none other than the impact of the COVID-19 pandemic that has pushed the market expansion for the use of cloud computing technology due to every industrial sector moving their business to cloud services [2].

One of the main advantages of utilizing a cloud computing technology is scalability, where users can use an unlimited computing resources provided by cloud computing services dynamically and efficiently based on demand. This way, cloud services can scale up computing resources whenever the needs arise to handle more requests from users and revert it back by scaling down the computing resources as usage decreases [3]. This scalability is especially useful to achieve cost efficiency due to the part where cloud service providers generally offer a pay as you go financing model where users need to pay based on the number and duration of the cloud computing resources used.

To achieve the maximum benefit of scalability implementation in a cloud environment, cloud service providers such as Amazon Web Services, Google Cloud Platform, and others have provided a feature called Auto Scaling Group (ASG) to perform automatic scaling on Virtual Machines (VM). The implementation of ASG in a cloud environment allows for scaling up and scaling down of computing resources based on CPU usage or other metrics that can be configured by users, such as whenever the load on one of the VMs reaches a preset threshold, the cloud service provider can then automatically create a new VM so then the load can be shared across multiple VMs [4]. However, scalability implementation directly on VMs is not recommended as it presents inefficiencies in existing computing resources. During a VM orchestration process, it is necessary for each VM to because during VM orchestration process it is necessary to run the respective operating system, because during VM orchestration process it is necessary to run the respective operating systems which can consume computing resources such as CPU and storage. Therefore, scalability implementation in cloud environments generally relies on other virtualization technologies, such as containers. Container provides an environment where processes can run independently and isolated both from the host and other containers. The virtualization technology in containers is pretty different than VMs, as containers do not require an operating system to be run on each instance so that they can save on the storage usage. Additionally, container technology other advantages, such as faster boot time (1.53 seconds compared to 11.48 seconds using a VM) and faster performance (4.546 seconds compared to 4.793 seconds using a VM) [5].

For all the reasons mentioned above, container technology may quickly replace VM-based virtualization technology as the computing instance of choice in cloud-based service. And to support the container-based deployment process, there are several container orchestration tool options available for managing clusters of computing instances for running containers, such as Kubernetes, Docker Swarm, Mesos, and OpenShift. Kubernetes, as the market leader of container orchestration tools thanks to it being an open-source application and supported by big community, makes it possible to scale pods based on the resource requirements of each pod, so that all applications running in Kubernetes can be easily scaled as needed [6].

Implementation of scalability using Kubernetes can be achieved by utilizing some of the features already provided in Kubernetes. Some of the scalability features provided in Kubernetes are Horizontal Pod Autoscaler, Vertical Pod Autoscaler, and Cluster Autoscaler [7], with each of them providing different approach to manage computing resources in the Kubernetes cluster by allowing containers to run on computing instances based on the resource requirements of each container and the overall computing resources availability [8]. And at the end of this research, we can obtain evidence on how these Kubernetes scalability features implemented in a cloud environment by running a series of tests conducted in this research, and how users can gain performance improvement by dynamically adding new computing resources when needed as well as cost efficiency by eliminating unused computing resources.

2. RESEARCH METHOD

This research carried out by running a series of experiments by applying various scalability configurations to an application running in a cloud environment to obtain the performance results of each configuration and make a reasonable comparison between the increase of computing resources and the application performance. In this research, the application used is Nginx, which is a popular HTTP server application used to provide static web content and capable of handling more than 10,000 connections simultaneously. Based on a survey conducted by Datadog in 2018, Nginx is the most used application running in containers, so for the purpose of this research, Nginx chosen as the application that will be run in Kubernetes and tested using a load test. Meanwhile, the container orchestration tool used is Elastic Kubernetes Service (EKS), which is a managed Kubernetes service provided in Amazon Web Services to run Kubernetes in a cloud environment. The implementation of scalability using Kubernetes in a cloud environment is not limited to the Elastic Kubernetes Service (EKS) provided by Amazon Web Services, but can also be applied to other managed Kubernetes services such as Google Kubernetes Engine (GKE) from Google, Azure Kubernetes Service (AKS) from Microsoft, Alibaba Cloud Container Service for Kubernetes (ACK) on Alibaba Cloud, and other services where users can immediately use Kubernetes services without managing the system themselves. However, this research will only focus on the implementation of Kubernetes on Amazon EKS because based on the Flexera 2021 State of the Cloud Report, Amazon EKS still dominates the cloud-based container orchestration tool with 51% market share compared to other services.

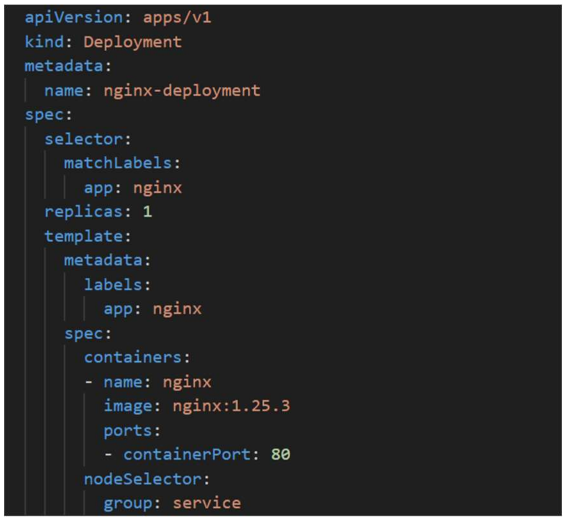
The scalability features in Kubernetes applied in this research are Horizontal Pod Autoscaler and Cluster Autoscaler. Horizontal Pod Autoscaler is a component in Kubernetes that can automatically increase the number of pods or the smallest unit in Kubernetes which represents a process running in a Kubernetes cluster based on the use of computing resources such as CPU or other metrics such as the number of requests from users. Meanwhile, the Cluster Autoscaler is a component that automatically adjusts the number of nodes or computing instances in a Kubernetes cluster so that a number of nodes are available to run pods and ensures that no computing resources that are left unused by removing unneeded nodes. By using these two features, the service can run with high performance and cost efficient, which is possible by adding computing resources only when needed and removing unused computing resources.

Experiment carried out by running load test on the Nginx application running in a Kubernetes cluster, where it is expected that it can increase the performance in handling requests from users by applying scalability. Nginx application will run using default configuration to send respond of the default page for each request from the client. Meanwhile, load test carried out using wrk application, which is an HTTP benchmarking tool capable of increasing web server performance loads by sending a large number of requests. Each wrk run will be using 40 threads and 400 open connections with a duration of 10 minutes as benchmark parameters, and monitored using kubectl top command.

And finally, the instance type used to run the load test will be using t3.xlarge with 4 vCPU and 16 GiB RAM, while the instance running the Nginx application will use 3.medium instance type with 2 vCPU and 4 GiB RAM. This ensures that the number of requests sent by load test instance is always higher than the actual number of requests that can be handled by Nginx, so the service ability to handle the maximum number of requests from users can be properly measured. In the experiment, scalability implementation is only applied based on monitoring of the CPU workload, while it is actually possible for Kubernetes to apply scalability based on other metrics such as workload of other computing resources or even custom metrics that is defined by the user.

3. RESULTS

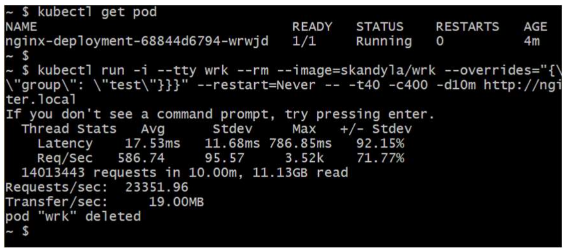
The first test carried out by running the Nginx web server in one pod, where all node resources were consumed by the application to handle requests sent by wrk so we can analyze how a process running only on one pod maximizes the usage of available computing resources. To run Nginx in Kubernetes, we need a Kubernetes manifest file to download the image from the container registry and deploy it to the Kubernetes cluster. This Kubernetes manifest file contains a simple configuration for running a process into a pod using just one replica.



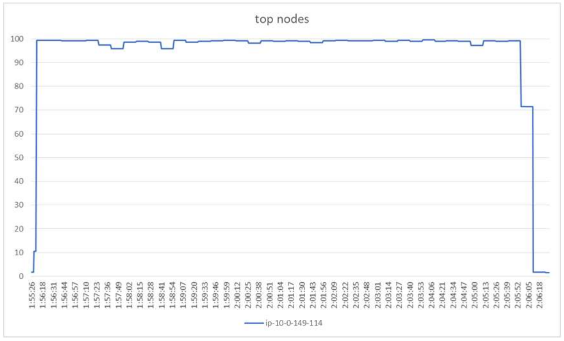
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.25.3
        ports:
        - containerPort: 80
      nodeSelector:
        group: service
```

Figure 1: Kubernetes Manifest for Nginx

During this test, the node CPU resources monitored at 98.77%, while the pod CPU usage 65.27%. By using this resources, wrk recorded that Nginx is capable of handling 14,013,443 requests sent within 10 minutes or 23,351.96 requests per second.



```
~ $ kubectl get pod
NAME                               READY   STATUS    RESTARTS   AGE
nginx-deployment-68844d6794-wrwjd   1/1     Running   0          4m
~ $
~ $ kubectl run -i --tty wrk --rm --image=skandyla/wrk --overrides="{\
\"group\": \"test\"}}}" --restart=Never -- -t40 -c400 -d10m http://ngi
ter.local
If you don't see a command prompt, try pressing enter.
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    17.53ms   11.68ms 786.85ms   92.15%
    Req/Sec   586.74     95.57     3.52k    71.77%
  14013443 requests in 10.00m, 11.13GB read
Requests/sec:  23351.96
Transfer/sec:     19.00MB
pod "wrk" deleted
~ $
```



ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

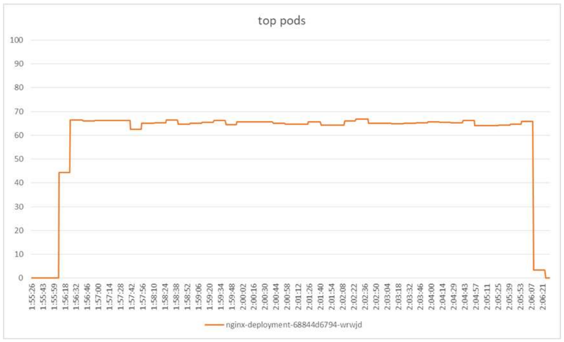
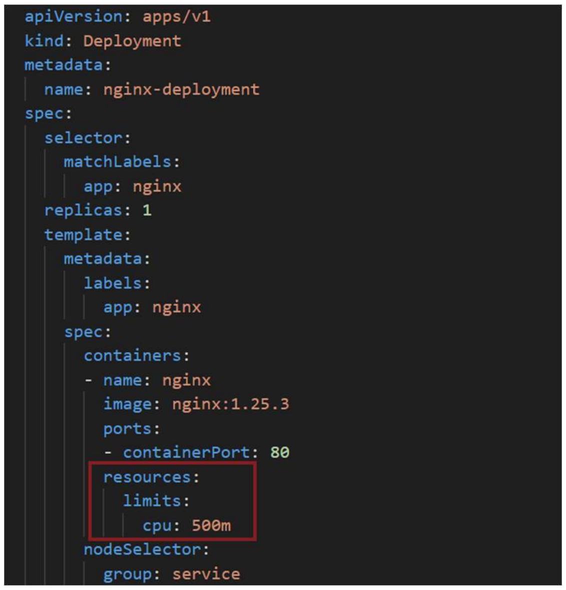


Figure 2: Load Test using 1 Pod

Based on the results of running tests using one pod, it can be ascertained that the maximum utilization of computing resources for the running process is only around 60 to 70% of the node CPU resource, although this figure will be consistent in all results obtained in subsequent tests. And while it is possible to run a process within a pod without specifying resource limit that the pod can use, it is not well recommended as it may interfere with other processes running on the same node. So generally, each pod will run with specifying resource request so that Kubernetes can ensure that the pod does not use resources that exceed the specified limits.

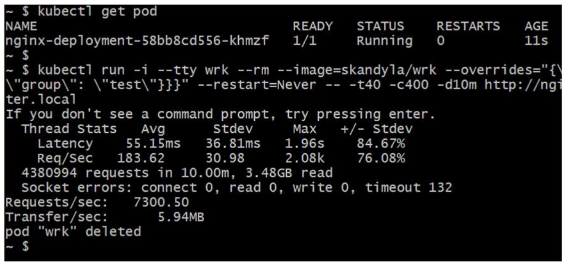
The second test still carried out by running the Nginx web server in one pod, but resource limit has been set, so now the pod is only allowed to use 0.5 vCPU or 25% of the total CPU in the t3.medium instance. In this test, we expect that each pod uses only a maximum of 25% CPU out of all the computing resources available for use. This can be done by setting the resource limit value in the Kubernetes manifest file, as in this test which specifies the CPU resource limit value using the unit of "millicore".

In the result for this test, the node CPU resources monitored at 39.18% with the pod CPU usage only peaking at 24.97%. By applying this limit, the number of requests sent by wrk that can be handled by Nginx reduced to only 4,380,994 requests in 10 minutes or 7,300.50 requests per second.

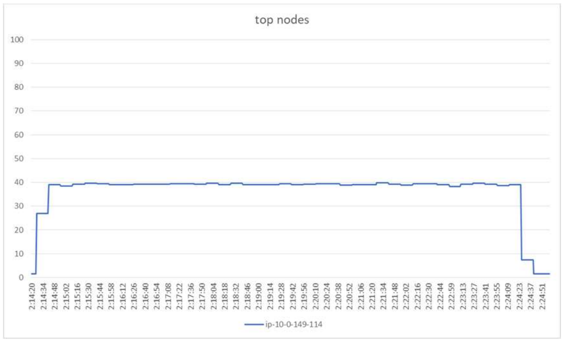


```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.25.3
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: 500m
      nodeSelector:
        group: service
```

Figure 3: Kubernetes Manifest with Resource Limit



```
~ $ kubectl get pod
NAME                               READY   STATUS    RESTARTS   AGE
nginx-deployment-58bb8cd556-khmzf  1/1     Running   0          11s
~ $
~ $ kubectl run -i --tty wrk --rm --image=skandyla/wrk --overrides="{\
\"group\": \"test\"}}}" --restart=Never -- -t40 -c400 -d10m http://ngi
ter.local
If you don't see a command prompt, try pressing enter.
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    55.15ms   36.81ms   1.96s    84.67%
    Req/Sec   183.62     30.98     2.08k    76.08%
  4380994 requests in 10.00m, 3.48GB read
  Socket errors: connect 0, read 0, write 0, timeout 132
Requests/sec:   7300.50
Transfer/sec:      5.94MB
pod "wrk" deleted
~ $
```



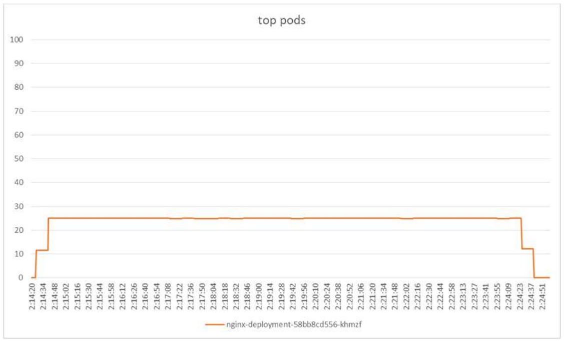
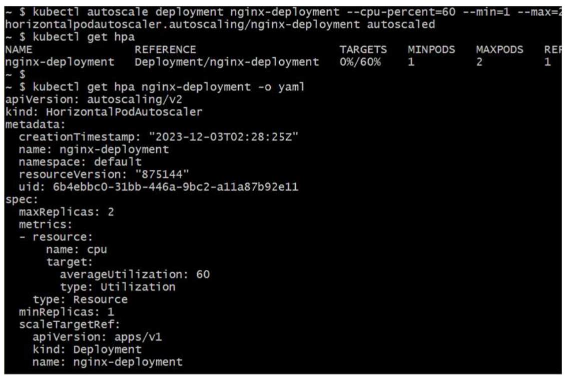


Figure 4: Load Test using 1 Pod with Resource Limit

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

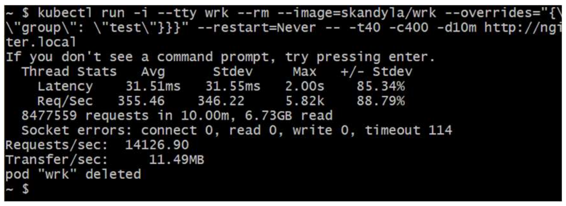
Next, load test carried out by applying scalability to increase the number of pods using the Horizontal Pod Autoscaler if the resource usage in the pod exceeds the specified limits. For this test, if CPU usage of a pod exceeds 0.3 vCPU or 15% of the total CPU in the t3.medium instance, then a new pod will be added to the cluster and requests from users will be distributed to all existing pods. Implementing scalability using Horizontal Pod Autoscaler is done by adding a Kubernetes object of the "HorizontalPodAutoscaler" kind to the Kubernetes manifest file, or can also be done by using the "kubectl autoscale" command and letting Kubernetes to automatically create the Kubernetes object without the user having to do it manually.



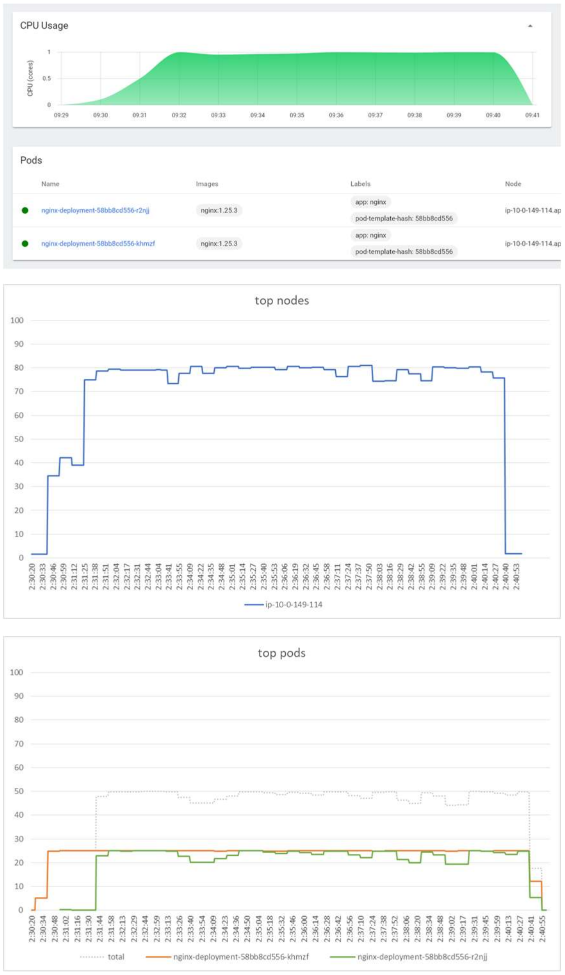
```
~ $ kubectl autoscale deployment nginx-deployment --cpu-percent=60 --min=1 --max=2
horizontalpodautoscaler.autoscaling/nginx-deployment autoscaled
~ $ kubectl get hpa
NAME               REFERENCE                     TARGETS   MINPODS   MAXPODS   REF
nginx-deployment   Deployment/nginx-deployment   0%/60%    1         2         1
~ $
~ $ kubectl get hpa nginx-deployment -o yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  creationTimestamp: "2023-12-03T02:28:25Z"
  name: nginx-deployment
  namespace: default
  resourceVersion: "875144"
  uid: 6b4ebbc0-31bb-446a-9bc2-a11a87b92e11
spec:
  maxReplicas: 2
  metrics:
  - resource:
      name: cpu
      target:
        averageUtilization: 60
        type: Utilization
    type: Resource
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx-deployment
```

Figure 5: Horizontal Pod Autoscaler

In this test, it can be observed at 2:30:55 that a new pod was added to the cluster to handle requests that cannot be handled by the existing pods. CPU resource usage on the node increased to 78.62% with CPU usage on each pod at 24.22%. With the addition of these pods, the number of requests sent by wrk that can be handled by Nginx instances increased to 8,477,559 requests in 10 minutes or 14,126.90 requests per second, which is approximately 2 times increase than before.



```
~ $ kubectl run -i --tty wrk --rm --image=skandyla/wrk --overrides="{\
\"group\": \"test\"}}}" --restart=Never -- -t40 -c400 -d10m http://ngi
ter.local
If you don't see a command prompt, try pressing enter.
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    31.51ms   31.55ms   2.00s    85.34%
    Req/Sec   355.46    346.22     5.82k    88.79%
  8477559 requests in 10.00m, 6.73GB read
  Socket errors: connect 0, read 0, write 0, timeout 114
Requests/sec:  14126.90
Transfer/sec:     11.49MB
pod "wrk" deleted
~ $
```

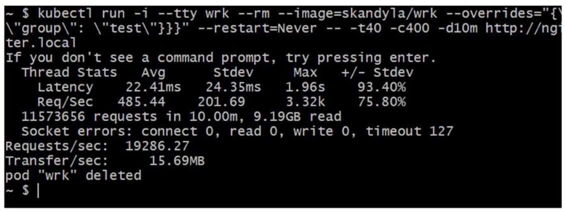


Name	Images	Labels	Node
● nginx-deployment-58bb8cd556-r2njj	nginx:1.25.3	app: nginx pod-template-hash: 58bb8cd556	ip-10-0-149-114.ap
● nginx-deployment-58bb8cd556-khmzf	nginx:1.25.3	app: nginx pod-template-hash: 58bb8cd556	ip-10-0-149-114.ap

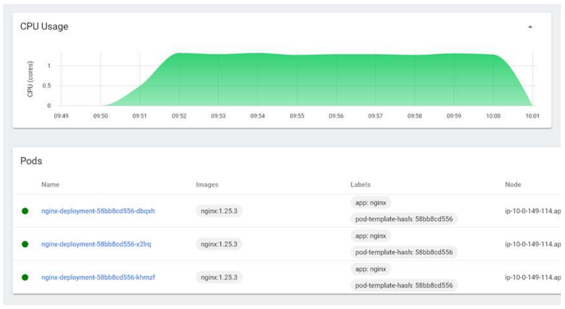
Figure 6: Load Test using 2 Pods

The next test carried out by applying scalability to use a maximum number of 3 pods. Based on observation at 2:51:43, there are 2 pods added to the cluster and the node CPU resources usage increased to 99.95%. However, we can also notice that the CPU usage of each pod decreased to 21.57%, with the first pod CPU usage decreased from 24.75% to 22.75% at 2:51:43 mark. This decrease in pod CPU usage was caused by the node CPU usage already peaked at almost 100%, so there were no more computing resources available for use by the pod. While the number of requests still increased to 11,573,656 requests in 10 minutes or 19,286.27 requests per second or 2.6 times than when using 1 pod, it is noticeable that the performance increase is not linear with the addition of pods.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

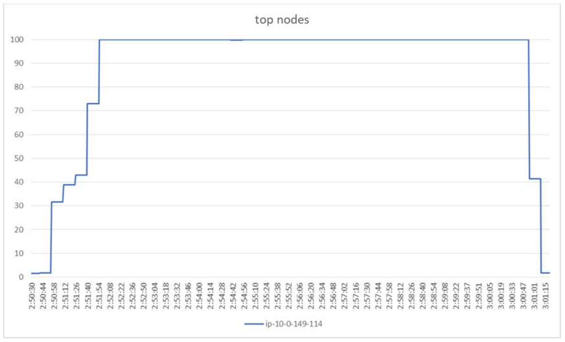


```
~ $ kubectl run -i --tty wrk --rm --image=skandyla/wrk --overrides="{\
\"group\": \"test\"}}}" --restart=Never -- -t40 -c400 -d10m http://ngi
ter.local
If you don't see a command prompt, try pressing enter.
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    22.41ms   24.35ms   1.96s    93.40%
    Req/Sec   485.44    201.69     3.32k    75.80%
  11573656 requests in 10.00m, 9.19GB read
  Socket errors: connect 0, read 0, write 0, timeout 127
Requests/sec:  19286.27
Transfer/sec:     15.69MB
pod "wrk" deleted
~ $ |
```



Pods

	Name	Images	Labels	Node
●	nginx-deployment-58bb8cd556-dbqxh	nginx:1.25.3	app: nginx pod-template-hash: 58bb8cd556	ip-10-0-149-114.ap
●	nginx-deployment-58bb8cd556-x2lrq	nginx:1.25.3	app: nginx pod-template-hash: 58bb8cd556	ip-10-0-149-114.ap
●	nginx-deployment-58bb8cd556-khmzf	nginx:1.25.3	app: nginx pod-template-hash: 58bb8cd556	ip-10-0-149-114.ap



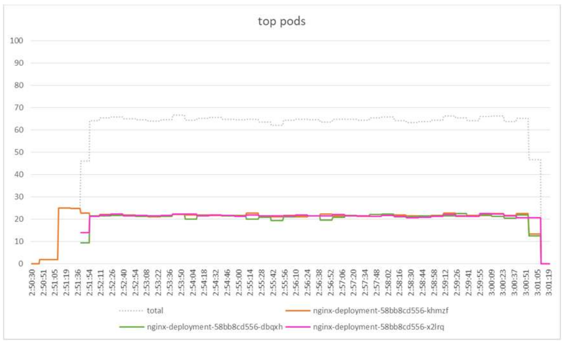
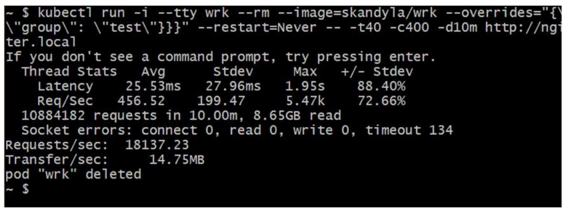
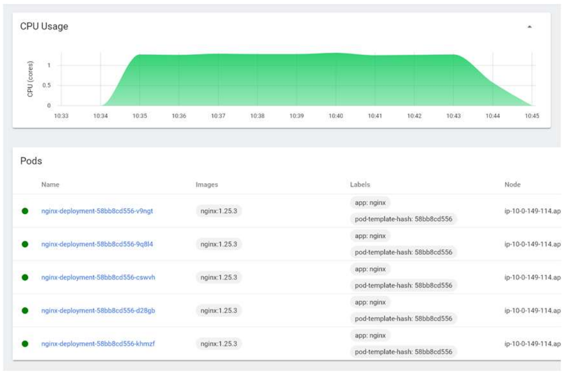


Figure 7: Load Test using 3 Pods

Further test by increasing the maximum number of pods to 5 shows proof that the addition of pods that were not accompanied by adding more computing resources resulted in no performance improvement, and in this case even cause a performance degradation where the node CPU resources usage stays at 99.96% while the average CPU usage of each pod is only at 12.54%. The requests also decreased to only 10,884,182 requests in 10 minutes, or 18,137.23 requests per second.

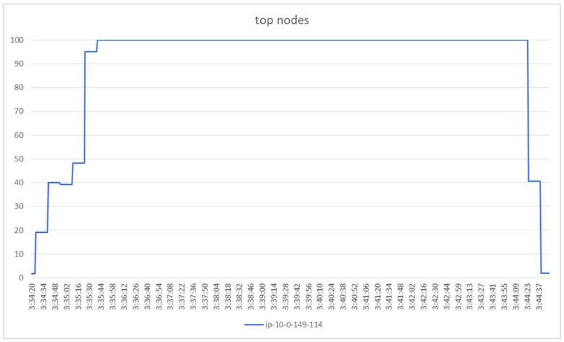


```
~ $ kubectl run -i --tty wrk --rm --image=skandyla/wrk --overrides="{\
\"group\": \"test\"}}}" --restart=Never -- -t40 -c400 -d10m http://ngi
ter.local
If you don't see a command prompt, try pressing enter.
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    25.53ms   27.96ms   1.95s    88.40%
    Req/Sec   456.52    199.47     5.47k    72.66%
  10884182 requests in 10.00m, 8.65GB read
  Socket errors: connect 0, read 0, write 0, timeout 134
Requests/sec:  18137.23
Transfer/sec:     14.75MB
pod "wrk" deleted
~ $
```



Pods

	Name	Images	Labels	Node
●	nginx-deployment-58bb8cd556-v9ngt	nginx:1.25.3	app: nginx pod-template-hash: 58bb8cd556	ip-10-0-149-114.ap
●	nginx-deployment-58bb8cd556-9q8l4	nginx:1.25.3	app: nginx pod-template-hash: 58bb8cd556	ip-10-0-149-114.ap
●	nginx-deployment-58bb8cd556-cswvh	nginx:1.25.3	app: nginx pod-template-hash: 58bb8cd556	ip-10-0-149-114.ap
●	nginx-deployment-58bb8cd556-d28gb	nginx:1.25.3	app: nginx pod-template-hash: 58bb8cd556	ip-10-0-149-114.ap
●	nginx-deployment-58bb8cd556-khmzf	nginx:1.25.3	app: nginx pod-template-hash: 58bb8cd556	ip-10-0-149-114.ap



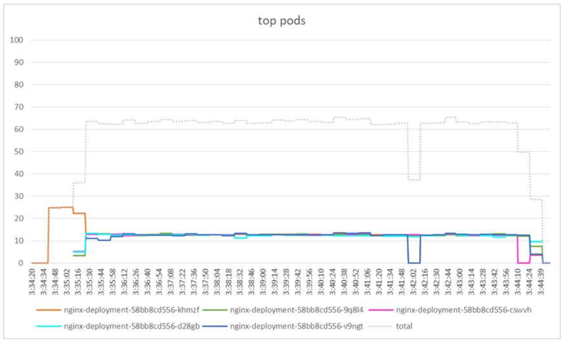
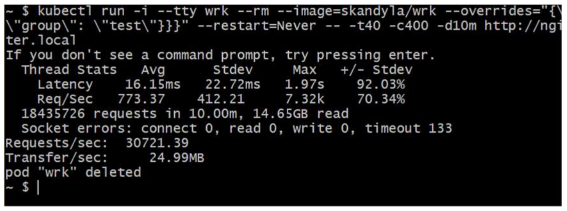


Figure 8: Load Test using 5 Pods

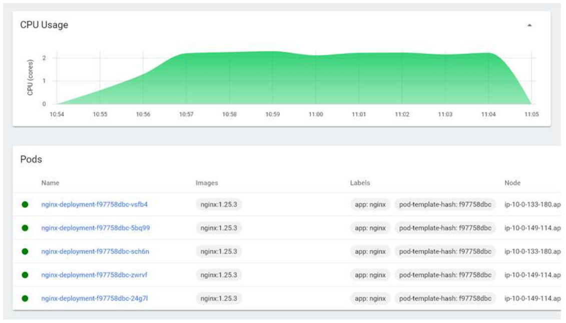
Therefore, applying scalability by simply increasing the number of pods is not enough and must be balanced by increasing the computing resources. To rectify the issue the number of nodes can be adjusted to accommodate the number of pods. Cluster Autoscaler is a Kubernetes component that ensures that no pods fail to run on the cluster due to insufficient resources, and improves efficiency on nodes that are not fully utilized for a long time by moving pods to other nodes and deleting unused nodes.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

In this last test, it can be observed at 3:56:40 there is a new node added to the cluster because all the computing resources on the old node had been consumed by the 3 pods running on that node. So a new node was needed to run the remaining 2 pods. In this test, the pod CPU usage stabilize at 21.98%, with CPU usage of the first node at 99.7% and the second node at 73.84%. The number of requests that can be handled using 5 pods on 2 nodes increased to 18,435,726 requests in 10 minutes or 30,721.39 requests per second or almost 4.2 times of the number of requests using 1 pod.

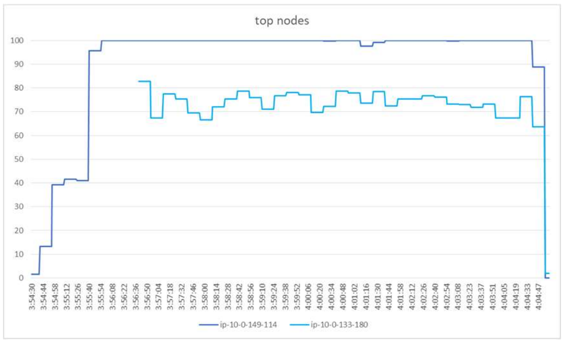


```
~ $ kubectl run -i --tty wrk --rm --image=skandyla/wrk --overrides="{\
\"group\": \"test\"}}}" --restart=Never -- -t40 -c400 -d10m http://ngi
ter.local
If you don't see a command prompt, try pressing enter.
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    16.15ms   22.72ms   1.97s    92.03%
    Req/Sec   773.37    412.21     7.32k    70.34%
  18435726 requests in 10.00m, 14.65GB read
  Socket errors: connect 0, read 0, write 0, timeout 133
Requests/sec:  30721.39
Transfer/sec:     24.99MB
pod "wrk" deleted
~ $ |
```



Pods

	Name	Images	Labels	Node
●	nginx-deployment-f97758dbc-vsfb4	nginx:1.25.3	app: nginx pod-template-hash: f97758dbc	ip-10-0-133-180.ap
●	nginx-deployment-f97758dbc-5bq99	nginx:1.25.3	app: nginx pod-template-hash: f97758dbc	ip-10-0-149-114.ap
●	nginx-deployment-f97758dbc-sch6n	nginx:1.25.3	app: nginx pod-template-hash: f97758dbc	ip-10-0-133-180.ap
●	nginx-deployment-f97758dbc-zwrvf	nginx:1.25.3	app: nginx pod-template-hash: f97758dbc	ip-10-0-149-114.ap
●	nginx-deployment-f97758dbc-24g7l	nginx:1.25.3	app: nginx pod-template-hash: f97758dbc	ip-10-0-149-114.ap



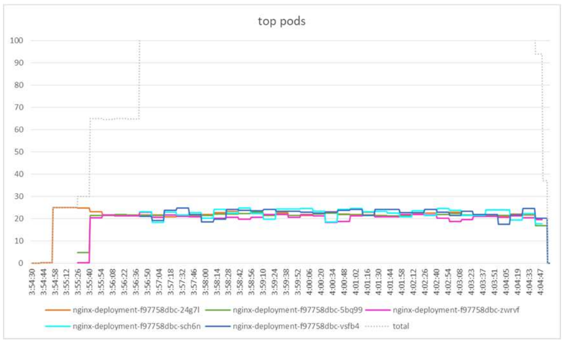
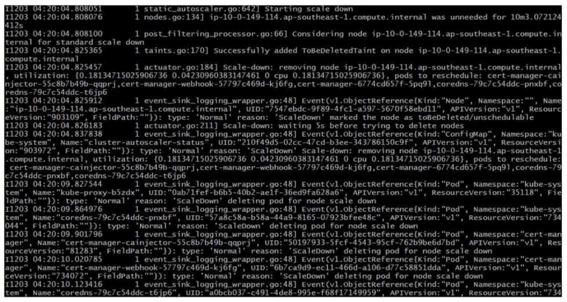


Figure 9: Load Test using 5 Pods on 2 Nodes

To maximize the efficiency of added resources in scalability implementation, all pods and nodes added by the Horizontal Pod Autoscaler and Cluster Autoscaler will be scheduled to be deleted when they are no longer needed. Pods with less than 10% CPU workload for more than 5 minutes can be scheduled for deletion, while nodes with low usage will also be deleted after 10 minutes. Therefore, when there are no more requests from wrk, the newly added node in this last test will be scheduled for deletion by Cluster Autoscaler.



```
I1203 04:20:04.808051       1 static_autoscaler.go:642] Starting scale down
I1203 04:20:04.808076       1 nodes.go:134] ip-10-0-149-114.ap-southeast-1.compute.internal was unneeded for 10m3.072124
412s
I1203 04:20:04.808100       1 post_filtering_processor.go:66] Considering node ip-10-0-149-114.ap-southeast-1.compute.in
ternal for standard scale down
I1203 04:20:04.825365       1 taints.go:170] Successfully added ToBeDeletedTaint on node ip-10-0-149-114.ap-southeast-1.
compute.internal
I1203 04:20:04.825457       1 actuator.go:184] Scale-down: removing node ip-10-0-149-114.ap-southeast-1.compute.internal
, utilization: {0.1813471502590736 0.04230960383147461 0 cpu 0.1813471502590736}, pods to reschedule: cert-manager-cai
njector-55c8b7b49b-qqprj,cert-manager-webhook-57797c469d-kj6fg,cert-manager-6774cd657f-5pq91,coredns-79c7c54ddc-pnxbf,co
redns-79c7c54ddc-t6jp6
I1203 04:20:04.825912       1 event_sink_logging_wrapper.go:48] Event(v1.ObjectReference{Kind:"Node", Namespace:"", Name
:"ip-10-0-149-114.ap-southeast-1.compute.internal", UID:"7547ebdc-9f89-4fc1-a597-5670f58ebd11", APIVersion:"v1", Resourc
eVersion:"903109", FieldPath:""}): type: 'Normal' reason: 'ScaleDown' marked the node as toBeDeleted/unschedulable
I1203 04:20:04.826183       1 actuator.go:211] Scale-down: waiting 5s before trying to delete nodes
I1203 04:20:04.837838       1 event_sink_logging_wrapper.go:48] Event(v1.ObjectReference{Kind:"ConfigMap", Namespace:"ku
be-system", Name:"cluster-autoscaler-status", UID:"210f49d5-02cc-47cd-b3ee-343786150c9f", APIVersion:"v1", ResourceVersi
on:"903972", FieldPath:""}): type: 'Normal' reason: 'ScaleDown' Scale-down: removing node ip-10-0-149-114.ap-southeast-1
.compute.internal, utilization: {0.1813471502590736 0.04230960383147461 0 cpu 0.1813471502590736}, pods to reschedule:
 cert-manager-cainjector-55c8b7b49b-qqprj,cert-manager-webhook-57797c469d-kj6fg,cert-manager-6774cd657f-5pq91,coredns-79
c7c54ddc-pnxbf,coredns-79c7c54ddc-t6jp6
I1203 04:20:09.827544       1 event_sink_logging_wrapper.go:48] Event(v1.ObjectReference{Kind:"Pod", Namespace:"kube-sys
tem", Name:"kube-proxy-b5zdx", UID:"0ab71fef-b6b5-40b2-ae1f-36ed9fa628a6", APIVersion:"v1", ResourceVersion:"35118", Fie
ldPath:""}): type: 'Normal' reason: 'ScaleDown' deleting pod for node scale down
I1203 04:20:09.864976       1 event_sink_logging_wrapper.go:48] Event(v1.ObjectReference{Kind:"Pod", Namespace:"kube-sys
tem", Name:"coredns-79c7c54ddc-pnxbf", UID:"57a8c58a-b58a-44a9-8165-07923bfee48c", APIVersion:"v1", ResourceVersion:"734
044", FieldPath:""}): type: 'Normal' reason: 'ScaleDown' deleting pod for node scale down
I1203 04:20:09.901796       1 event_sink_logging_wrapper.go:48] Event(v1.ObjectReference{Kind:"Pod", Namespace:"cert-man
ager", Name:"cert-manager-cainjector-55c8b7b49b-qqprj", UID:"50197933-5fcf-4543-95cf-762b9be6d7bd", APIVersion:"v1", Res
ourceVersion:"81283", FieldPath:""}): type: 'Normal' reason: 'ScaleDown' deleting pod for node scale down
I1203 04:20:10.020785       1 event_sink_logging_wrapper.go:48] Event(v1.ObjectReference{Kind:"Pod", Namespace:"cert-man
ager", Name:"cert-manager-webhook-57797c469d-kj6fg", UID:"6b7ca9d9-ec11-466d-a106-d77c58851dda", APIVersion:"v1", Resour
ceVersion:"734072", FieldPath:""}): type: 'Normal' reason: 'ScaleDown' deleting pod for node scale down
I1203 04:20:10.123416       1 event_sink_logging_wrapper.go:48] Event(v1.ObjectReference{Kind:"Pod", Namespace:"kube-sys
tem", Name:"coredns-79c7c54ddc-t6jp6", UID:"a0bcb037-c491-4de8-995e-f68f17149959", APIVersion:"v1", ResourceVersion:"734
```

Figure 10: Cluster Autoscaler Scale-Down

Below is a comparison table of CPU usage and the number of requests that can be handled by Nginx web server for each load test configuration executed in this research.

Table 1: Performance Comparison for each Load Test

	Node CPU usage	Pod CPU usage	Req/10 min	Req/sec
1 pod (resource limit)	39.18%	24.97%	4,380,994	7,300.50
1 pod (no resource limit)	98.77%	65.27%	14,013,443	23,351.96
2 pod	78.62%	24.22%	8,477,559	14,126.90
3 pod	99.95%	21.57%	11,573,656	19,286.27
5 pod 1 node	99.96%	12.54%	10,884,182	18,137.23
5 pod 2 nodes	173.54%	21.98%	18,435,726	30,721.39

These numbers are quite comparable to the findings presented in prior studies where the performance improvement obtained from applying scalability is more or less proportional to the number of pods [9], or even in the reverse scenario where scalability is applied to a service with statically defined number of requests and it has shown evidence that the CPU workload is being distributed proportionally among multiple pods [8].

4. CONCLUSIONS

Based on the test results carried out in this research, it is proven that applying scalability to applications can improve the performance of the application, such as increasing the number of requests sent by wrk that can be handled by the Nginx web server. And at the same time, the performance improvement introduced by scaling up the computing resources is balanced with the scaling down of unused computing resources so users can gain the benefit of maximum performance with minimal cost.

So when designing scalable and efficient services, especially for those running in a cloud environment, Kubernetes with its various scalability features can be one of the options that can be chosen by users. In addition, apart from scalability implementation based on monitoring of CPU workload only as carried out in this research, there are other metrics that can be used, such as based on the number of requests from users, which can be considered for next research. And since the implementation of scalability by Cluster Autoscaler dependent on the cloud service provider, a research to compare the performance improvement on scaling up and cost efficiency on scaling down between multiple cloud environments is also worth considering as an idea for further research.

REFERENCES

[1] P. Dutta and P. Dutta, "Comparative Study of Cloud Services Offered by Amazon, Microsoft and Google," *International Journal of Trend in Scientific Research and Development,* vol. 3, pp. 981-985, 2019.

[2] "Cloud Computing Market Size, Share & COVID-19 Impact Analysis," [Online]. Available: https://www.fortunebusinessinsights.com/cloud-computing-market-102697.

[3] N. K. Sehgal, P. C. P. Bhatt and J. M. Acken, Cloud Computing with Security Concepts and Practices, Second ed., Springer, 2020.

[4] D. ND, A. Deshmukh, G. M, A. P. Chavan, H. V. R. Aradhya and N. N. N, "Experimental Analysis of Performance Metrics for Configuration of Auto Scaling Groups," *International Research Journal of Engineering and Technology (IRJET),* vol. 7, no. 5, 2020.

[5] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon and B.-J. Kim, "Performance Comparison Analysis of Linux Container and Virtual Machine for Building Cloud," *Advanced Science and Technology Letters,* vol. 66, pp. 105-111, 2014.

[6] V. Medel, O. Rana, J. Á. Bañares and U. Arronategui, "Modelling Performance & Resource Management in Kubernetes," *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing,* 2016.

[7] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park and S. Kim, "Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration," *Sensors,* vol. 20, no. 16, 2020.

[8] L. P. Dewi, A. Noertjahyana, H. N. Palit and K. Yedutun, "Server Scalability Using Kubernetes," *The 2019 Technology Innovation Management and Engineering Science International Conference (TIMES-iCON2019),* pp. 1-4, 2019.

[9] D. Balla, C. Simon and M. Maliosz, "Adaptive scaling of Kubernetes pods," *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium,* pp. 1-5, 2020.