Reducing instance acquisition lag to improve scaling out in the kubernetes cluster

MSc Research Project
Cloud Computing

Bharath Raj Kanthimathinathan
Student ID: 20225024

School of Computing
National College of Ireland

Supervisor: Shivani Jaswal

Student Name:	Bharath Raj Kanthimathinathan
Student ID:	20225024
Programme:	Cloud Computing
Year:	2022
Module:	MSc Research Project
Supervisor:	Shivani Jaswal
Submission Due Date:	15/08/2022
Project Title:	Reducing instance acquisition lag to improve scaling out in the kubernetes cluster
Word Count:	5969
Page Count:	19

Signature:	Bharath Raj
Date:	18th September 2022

Reducing instance acquisition lag to improve scaling out in the kubernetes cluster

Bharath Raj Kanthimathinathan
20225024

Abstract

Cloud infrastructure leverages containerization which allows developers to deploy application anywhere in a repeatable and consistent manner without library or package dependency failures. With kubernetes being the industry standard for container orchestration, there is still tremendous scope for improvement in kubernetes cluster autoscaling. Both vertical pod autoscaling and horizontal pod autoscaling is limited by the cluster node autoscaling which acts as a bottleneck. Hence, a new node autoscaling solution is required to overcome the limitation of cluster autoscaling. This paper proposes a kubernetes cluster autoscaling solution called ANA autoscaler which makes use of bash scripts which creates and adds a new cluster node in a dormant state, ready to use when needed during cluster scale out. Here, the instance acquisition time refers to the amount of time taken by the kubernetes cluster node to add to the cluster and make it usable. The results show 72% improvement in comparison with the reaction time taken for a node to be added in EKS amazon managed kubernetes cluster with a dynamic scaling setting. This will massively improve the scale out time and reduce the application performance degradation. This also indicates there is still scope for improvement in the reaction time of node autoscaling in kubernetes cluster.

1 Introduction

Containerization has become widely used in the industry over the past few years, by using the scalable features of the cloud and the application portability feature of the containers, applications are able to scale faster and reliably than ever before. There are many research papers done on many autoscaling techniques like horizontal pod autoscaling and vertical pod autoscaling. These research papers used a number of techniques which could be categorised into reactive and proactive autoscaling, but these were mostly concerned with pod autoscaling. So even though there were multiple techniques proposed, the amount of resources available will be limited to the total resources available in the cluster. There are very few research papers which addresses this issue, where node autoscaling techniques are explored, there is tremendous scope to improve on the scaling feature of the cluster to make resources available quickly. Faster node autoscaling will improve the scale out time in the clusters which are present in the production environment hence enabling to support more traffic without any outage or drop in performance of the application.

This research focuses on improving the scale out time, it aims at improving the node acquisition time by making use of a proposed custom autoscaling solution called ANA autoscaler. Unlike other methods this method configures the node for acquisition and uses them immediately when the resource shortage takes place. The objective of the research is to improve the node addition time in the kubernetes cluster. On improving the node addition time the other pod autoscalers will be able to leverage the resources that are added to the cluster. The pod autoscalers will use the resources and scale to enable faster scaling of application which in turn helps to support huge growing traffic. Through this research if we are able to achieve improvement in time in comparison to the AWS managed kubernetes cluster which uses node autoscaling, this research paper would prove there is still more scope in the context of node cluster autoscaling techniques.

1.1 Research Question

> *Can the instance acquisition lag be reduced in the kubernetes cluster during scaling out using a custom cluster auto-scaling solution called ANA autoscaler on the kubernetes cluster which is manually managed in AWS Cloud?*

The instance acquisition lag refers to the amount of time that is taken by the kubernetes cluster to add a new node to the kubernetes cluster and make the resources usable for the application scale out. The kubernetes cluster will be setup in the AWS cloud where the custom ANA autoscaler solution will be installed. The time taken to autoscale is compared with the amount of time taken by the EKS amazon cluster to autoscale the nodes in the cluster. The EKS kubernetes cluster will be setup with a dynamic scaling setting, which only autoscales reactively similar to the setup used in Ifrah (2019). In this manner the results will be compared and the results will be concluded.

The ANA autoscaler will have a bash script which will add the nodes to the cluster and set it in a power off state. Once the ANA autoscaler realises that the resources are running short, the node which was in the power off state is started up and utilized in the kubernetes cluster. This includes using a monitoring system to read the monitoring metrics and also involves using a load generator script to generate CPU load on the kubernetes. The research involves using an ideal setup which makes use of consistent load which only considers CPU utilization as the fixed parameter. In the setup only the CPU utilization intensive load will be generated, only this parameter will be considered in all tests performed across different setups.

The report section consists of the literature review which analyses the different research papers present in the area of kubernetes autoscaling. It analyses the solution as well as the advantages and disadvantages of the research papers. Also some discussion is done on which solutions and ideas can be used as inspiration for this research paper. Following this the methodology section consists of the steps followed in the research work, which discusses in detail the steps that are to be taken in the different stages of the research, the tools used and the experiments to be carried out. Then the design specification section consists of the specifications, architecture of the system and the autoscaler. Next, the implementation of the ANA autoscaler system is discussed in the implementation section. After this the evaluation section consists of the analysis of the experiments conducted in this research paper. Finally the conclusion and the future work consists of the findings and the analysis of the results along with discussion on further extension of the research.

2 Related Work

Research on kubernetes has been widely done in the area of autoscaling as containerization is used widely in different areas of the IT as seen in Table 1.

2.1 Monitoring based solutions

2.1.1 Resource Provisioning algorithm

In Chang et al. (2017), the MAPE model is used which is the Monitor-Analyze-Plan-Execute model. The system uses the monitoring, aggregation of the data, resource scheduling and the pod scaler. In monitoring system the system resource metrics and the application performance metrics are used which makes use of the heapster and the Apache JMeter. The influxdb is used to aggregate and store the data. Following this the resource provisioning algorithm is used which in turn provides the strategy to the pod scaler. The biggest advantage of using this method is that not only the resource utilization of the system is taken into consideration but also the QoS of the application metrics is considered. The problem of using this method is that there are too many different components which will cause version conflicts when one of them is updated over time. The monitoring and the control loop is used which could be used for reactive based metrics.

2.1.2 Auto-adjusting algorithm

Node exporter which is part of the prometheus monitoring system is used in Zheng and Yen (2018) as the monitoring component. Further an algorithm is used which determines the number of application instances that are required. An autoscaling broker is used which communicates with the API-server and the scheduler component of the kubernetes cluster. It ingests all the information of the nodes, the application and then makes adjustments in the application replicas. Port on each pod is used so that the autoscaling group can use a HTTP GET and gather the required information. The advantage of using the approach is that the response time for requests are improved whereas the drawback is that the resources are underutilized when the demand is not consistent and regular.

2.2 Horizontal Pod Autoscaling

Previously the monitoring based autoscaling systems were discussed which utilised different monitoring setups to control the application count. In Nguyen et al. (2020a) the selection of the scrapping period and the type of metrics for the application are selected based on the type of workload used in the application. The horizontal pod autoscaling is analysed with the prometheus monitoring metrics and the kubernetes resource metrics. Experiments on horizontal pod autoscaling with default kubernetes resource metrics are performed. Along with this the experiments with different scraping periods are performed with fifteen, thirty and sixty seconds period. This showed that with longer scraping periods only smaller number of replicas were created. This was helpful in efficient use of resources but could be detrimental when incoming traffic is too high.

Comparison of two node cluster and four node cluster is made to analyse the horizontal pod autoscaling performance. The comparison showed that there is longer waiting time in the two node cluster than the four node kubernetes cluster. Experiments with readiness probes were also discussed which showed the balance between resources and the QoS needs of the application. The advantage of using this method is that there is reduction in the latency and improvement in the bandwidth when the right kind of metrics is selected for the application. While the major drawback is that the solution does not have the ability to understand the behaviour of the application and choose the correct metrics for the application.

2.3 Vertical and Horizontal elasticity

Previously solutions based on different monitoring setups were discussed whereas in Hoenisch et al. (2015) four dimensional scaling is considered where the virtual machines and the containers can be scaled horizontally and vertically. Here, the virtual machines and the containers are scaled horizontally by changing the number of instances where as the scaling in vertical way takes place by changing the resources being used. Multi-objective optimization model is used which interacts with different layers to control the number of virtual machines and the containers. In spite of additional complexity that is present with various degree of freedom in scaling, a reasonable distribution size is achieved. About twenty eight percent scaling is achieved with few scaling dimensions in two baseline scenarios.

The optimization model used in the solution focuses on reducing the over all cost for the virtual machines that are leased. It follows a two step approach where first the number of instances and the type that are needed are determined, following that the type of configuration for the container to be used is determined. The virtual machines are added and removed at any time, but the movement of containers could happen between the virtual machines. Leasing the right virtual machine by using the correct vertical scaling method could be useful in reducing the cost significantly with almost about twenty five percent savings. The major drawback of using this method is that the mechanism is unable to track the containers, that is in which virtual machine which container is present. This could lead to poor availability issues where all the containers could land up in the same virtual machine, so when there is a virtual machine failure the application could have a massive outage. Although there is poor visibility of scheduling of containers, the advantage is that the use of both scaling in virtual machines and containers leads to reduced costs.

2.4 Node autoscaling

In Thurgood and Lennon (2019) a Free and Open Source(FOSS) solution is used which will perform the cluster autoscaling in the kubernetes cluster. The solution was tested in the on-premises environment setup which could also be replicated in a cloud environment. Here, the virtual machine scaling is done on the basis of the foreman implementation. Foreman implementation proved to be a better choice as this was compatible with VMware products. Puppet modules is used for the installation of the foreman as well as the installation of the kubernetes cluster. For the ingress the HA-proxy design is used which is the industry based standard and apart from this a virtual machine is setup which behaves as the certificate authority. Three physical ESXi hosts are used with anti-affinity rules which separate the master nodes and the worker nodes. This provides redundancy in cases where node failures occur. Weave net is used which is the docker container network, this enables communication among containers present in different virtual machines. Foreman enables scaling by interacting with the VCenter API. Preconfigured templates are used which

Research paper	Advantage	Disadvantage
Chang et al. (2017)	The solution ingests details of both the utilization of resources in system and performance of application	Incompatability while upgrading due to many individual and seperate components
Zheng and Yen (2018)	Improvement in response time with respect to the traffic needs	When the traffic is uneven there is underutilization of resources
Nguyen et al. (2020a)	On selection of the suitable metrics, delay is improved	Requires user intervention for mapping the correct metrics with the application
Hoenisch et al. (2015)	Cost reduction is achieved by making use of scaling in containers and virtual machines	Provides no high availability for the containers
Thurgood and Lennon (2019)	Adding and deletion of node is done automatically based on load	Consumption of time is high, takes more than eight minutes which is very long
Current research	Huge improvement in time when adding new node to kubernetes cluster during scale-out of cluster	NA

Table 1: Research methods

uses the ubuntu image where the kubernetes packages and scripts are configured. For the setting and management of the IP, foreman tool takes care of the domain name system and the dynamic host configuration protocol. Consortium is used for the dynamic host configuration protocol and for the domain name system berkeley internet name domain version 9 is used.

The scaling of the nodes is initiated by a VCenter alarm which is started when the CPU or the memory utilization is increased beyond the set threshold. Locust is used for generating the application load on the system. A unique identifier is generated which is inputted to a command like tool called hammer, which is the command line tool for foreman. Secure copy protocol is used to copy files inside and outside the virtual machines when the draining or the scaling out of the kubernetes takes place. The advantage of this method is that the auto-addition and the auto-deletion of the node takes place automatically and reactively based on the application load, but the major disadvantage of using this method is that solution is very costly in terms of time used. The time taken for the virtual machine to build is around almost eight minutes which is very expensive in the world of on-demand cloud and containerization.

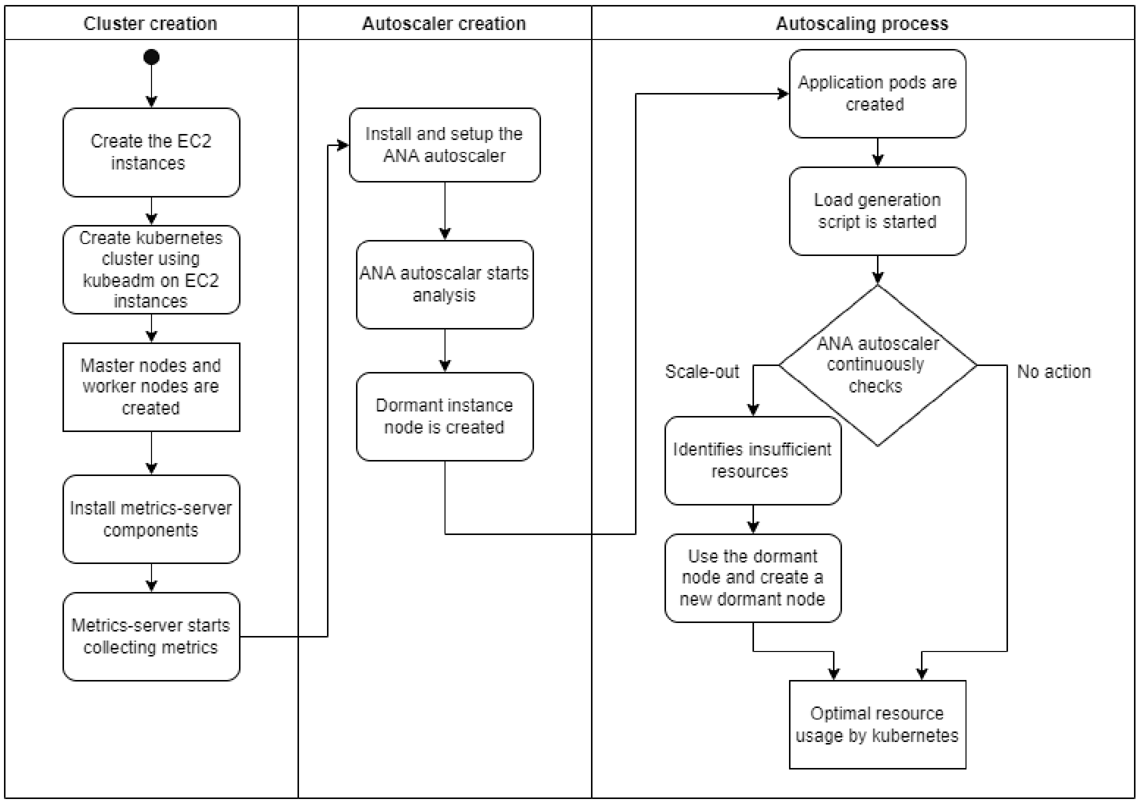

Cluster creation
Autoscaler creation
Autoscaling process
Create the EC2 instances
Create kubernetes cluster using kubeadm on EC2 instances
Master nodes and worker nodes are created
Install metrics-server components
Metrics-server starts collecting metrics
Install and setup the ANA autoscaler
ANA autoscalar starts analysis
Dormant instance node is created
Application pods are created
Load generation script is started
ANA autoscaler continuously checks
Scale-out
No action
Identifies insufficient resources
Use the dormant node and create a new dormant node
Optimal resource usage by kubernetes

Figure 1: Steps carried out during the research

3 Methodology

In this section the steps that are carried out in the research are explained, following that the tools used and the experiments to be carried out are mentioned as seen in Figure 1. This research focuses on whether the instance acquisition lag time can be reduced during scale out. To further improve the time taken for the addition of the kubernetes node during scale out an autoscaler solution called ANA autoscaler is used which will trigger and handle the addition of node for a kubernetes cluster with an application workload running.

3.1 Flow of research

The steps to be carried out in the research can be categorized into three major sections. The first section is the creation of the kubernetes cluster, second is the autoscaler creation and the third is performing the autoscaling process. In the first stage the cloud that is selected is the amazon web services cloud, in this the virtual machine equivalent EC2 instance is created. The EC2 instances are created in the same subnet and the same VPC network to ensure that the connectivity between them are established. Additionally the same security group is also used for these created EC2 instances to ensure the inbound and outbound rules between them are able to connect with each other. Following this all the required kubernetes packages are installed in the EC2 instances. Once the required packages are installed in the EC2 instances, the kubeadm command is used to install the kubernetes cluster into the created nodes. Using the kubeadm tool the master nodes and the worker nodes are created with the EC2 instances. After this the metrics server component is installed which is used for monitoring by the ANA autoscaler. The metrics server api is already used by the kubernetes in horizontal pod autoscaling internally. Once the metrics server component is installed the metrics are collected from the nodes of the kubernetes cluster. This gives visibility of the cpu utilization of the pods and the nodes present.

In the second stage the autoscaler components are copied, which consist of bash scripts. Once the bash script is copied into the master node, these scripts are added in the crontab entry to be triggered repeatedly similar to Chang et al. (2006). Once the ANA autoscaler starts the analysis, a dormant node is created and then added to the kubernetes cluster. In the third stage the autoscaling process is done, here a PHP apache application is ran on the kubernetes cluster as gathered from Vohra (2017). Once the load generation script is started, the cpu utilization of the kubernetes cluster increases, this is recognised by the ANA autoscaler . There are two possible actions that can be done by the ANA autoscaler, first is the scale out and second is the no action taken. During the scale out once the ANA autoscaler realises that the resources are insufficient the dormant node is brought up and utilised for resource scheduling. Once the dormant node is used another dormant node is created to be used when required.

3.2 Tools used

The tools that are used for the study are mentioned below as seen in Table 2. The EC2 instances used in the AWS is t2.micro and the application which is used is a PHP apache which is deployed as a deployment kubernetes object in the cluster. For CPU load generation a load generation script is used which will hit the PHP endpoint in a loop.

- Docker: Docker version 20.10 is used which helps in deploying the application image and provides the support to run pods in kubernetes.

- Kubeadm: Kubeadm is used to install the kubernetes cluster, manage and upgrade the cluster similar to Sami et al. (2020). This handles all the low level actions like managing all the certification that are required for communication among the nodes present in the kubernetes cluster.

- Kubernetes: Kubernetes version used is 1.24 which helps in container orchestration of the pods. This provides high availability for the application by running multiple instances of the application in different nodes and providing load balancing to the application.

- Metrics-server: Metrics server is used to get the CPU utilization of the kubernetes worker nodes. Metrics server uses the in built APIs present in the kubernetes cluster to get the kubernetes metrics.

- kubectl: Kubectl is the command line tool that is used to interact with the kubernetes API. It is through kubectl by which the commands are interpreted and passed as json to the kubernetes server in the back-end.

- krew: Krew is a kubectl plugin installer tool. Different plugins are available which provides additional functionality to the kubectl command line tool.

Specification of the system		
Cloud instance	AWS	t2.micro
Operating system	Ubuntu	20.04
Container runtime	Docker	20.10
Container orchestration	Kubernetes	1.24
Cloud command line	awscli	2.4.5

Table 2: Specification

- resource-capacity: Resource capacity is a plugin that is used to get the CPU utilization of the kubernetes nodes. This is installed using the krew tool. This provides the output in JSON file which makes it easier to perform operations on.
- jq: This is JQuery command line tool which is useful in bash scripts to filter the json file outputs and perform filtering on.
- awscli: This is an amazon web services command line tool which enables to interact with the AWS API services. This allows to create new EC2 instance through command line when ever it is triggered as shown in Backes et al. (2019).

3.3 Experiments carried out

There are two experiments that are to be carried out in this study. Firstly, the time taken for the ANA autoscaler to scale out is determined in a kubernetes cluster based out of AWS EC2 instance. Secondly, to compare this value, the same tests are carried out in an AWS managed kubernetes cluster which provides the autoscaling automatically. Whether the time taken to scale out by the ANA autoscaler is less than the time taken by the AWS EKS kubernetes cluster answers the purpose of this study.

In the first setup, there are two worker nodes configured and then one master node is configured. Then the ANA autoscaler scripts are configured to scale out the kubernetes cluster. Then the load generation scripts are ran, the logging captures the timestamps of the scale out. Once done, in the second setup AWS EKS kubernetes managed cluster is setup which helps in autoscaling of the nodes based on a dynamic settings configuration. After this the same PHP apache application is ran on the EKS cluster to log the node scale out time. On comparison of the logs present from the two setups the conclusion could be drawn whether the node autoscaling can be improved in the kubernetes cluster by a custom autoscaling solution.

4 Design Specification

Design specification contains the design of the system which are configuration settings of the setup. This section also covers the architecture of the kubernetes system that is used for the experiment. Finally the design of the proposed ANA cluster autoscaler system is discussed.

4.1 Specification of the system

For the container runtime docker is used in the research. For the container orchestration kubernetes tool is used for handling the lifecycle of containers. The kubernetes cluster will

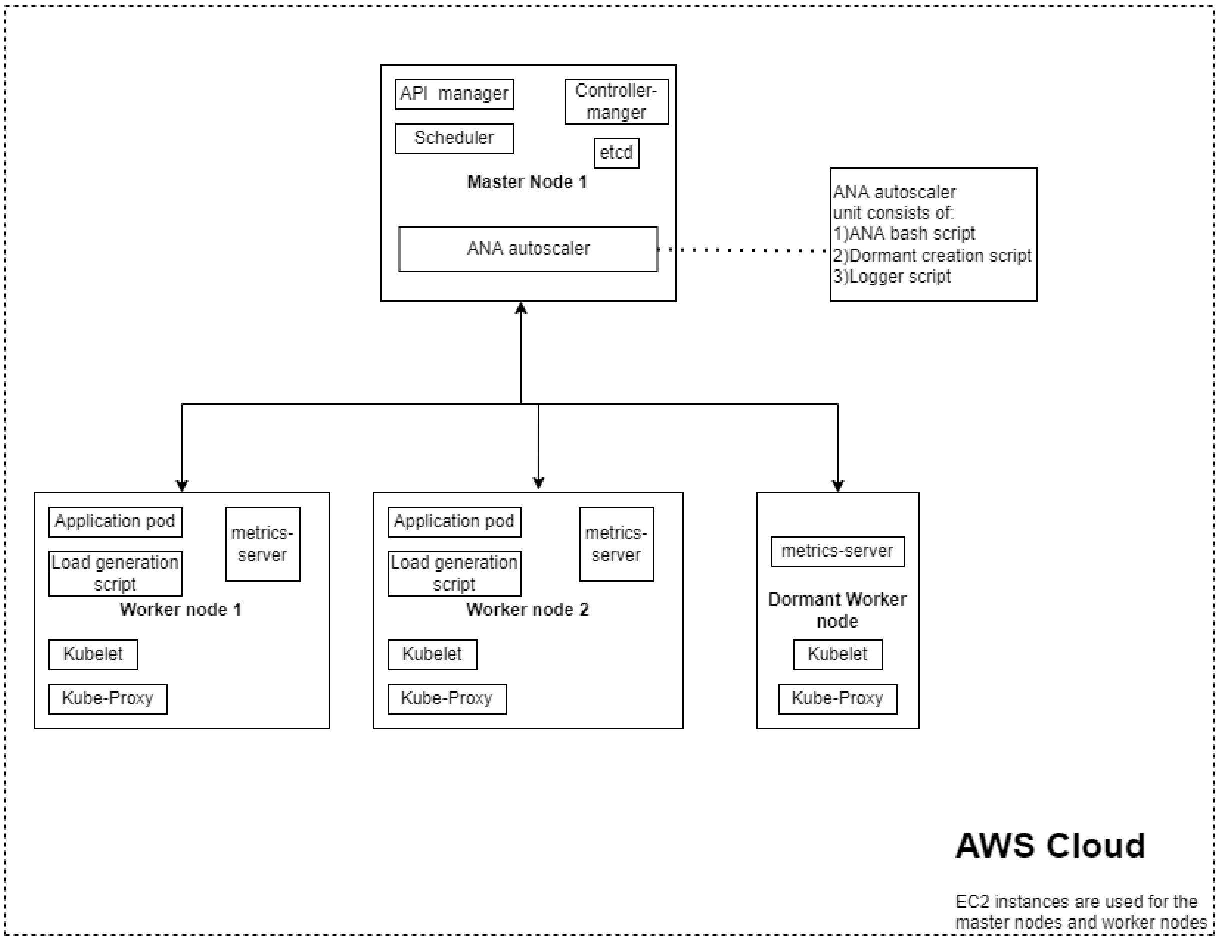


Figure 2: Architecture of the cluster

be installed in the EC2 instances present in the AWS cloud. The instance type version that is used is the t2.micro type. This consists of one virtual CPU and one GB of RAM. Each AWS instance costs about 0.0116 dollars per hour. Additionally AWS command line tool is installed in the master node to interact with the amazon api for triggering the launch and shutting down of the instance as required by the ANA autoscaler.

4.2 Architecture of the system

The kubernetes cluster consists of one master node and two worker nodes as seen in Figure 2. Worker nodes handles all the application traffic that is created by the application and the master node, also called as the control plane, manages all the pods and cluster components. As seen in paper Islam Shamim et al. (2020) in the master node the kube-apiserver, etcd, kube-scheduler, kube-controller-manager and cloud-controller manager are present. The control plane for the kubernetes is exposed by the API server, this also scales horizontally by creating replicas of the kube-apiserver. Etcd is used to store all the cluster information in the form of key value pairs, it also provides high availability and backing up feature for the stored data. Kube-scheduler schedules the pods based on the availability and rules set for scheduling. Various rules to be considered are policy constraints, resource requirements, locality of the data, anti-affinity rules and deadlines. Kube-controller-

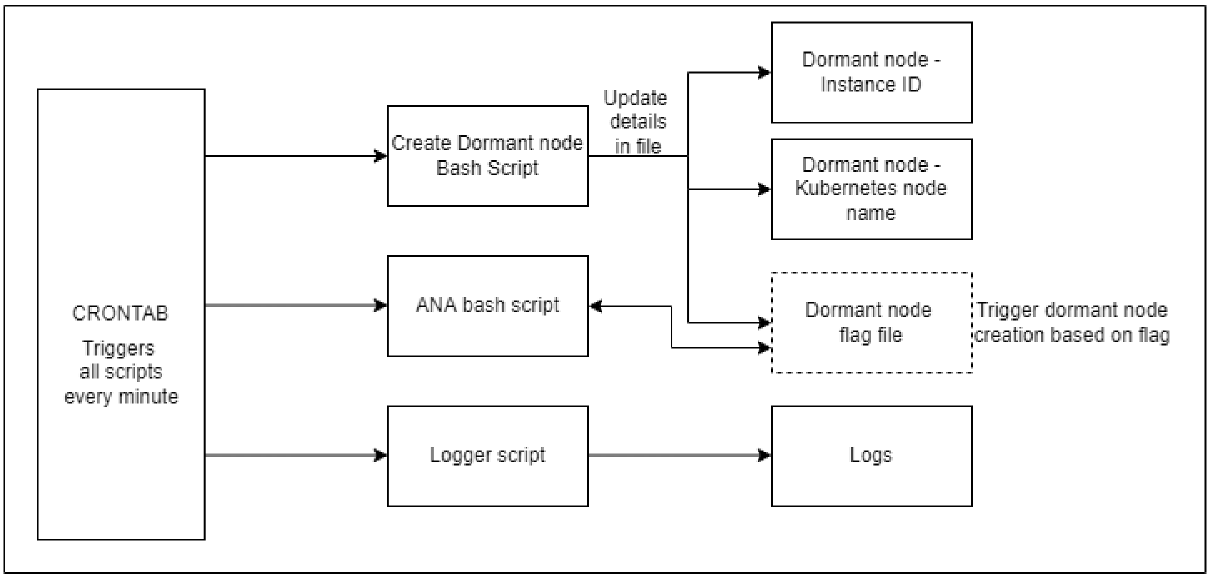

CRONTAB
Triggers all scripts every minute
Create Dormant node Bash Script
Update details in file
ANA bash script
Logger script
Dormant node - Instance ID
Dormant node - Kubernetes node name
Dormant node flag file
Trigger dormant node creation based on flag
Logs

Figure 3: ANA autoscaler architecture

manager runs different controller processes like job controller, node controller, endpoint controller and token controller. The cloud-controller-manager helps in interaction of the cluster with the cloud API. For cloud-controller-manager several controllers are used which run under the same process. In this setup cloud-controller-manager is not used.

The components that run on the kubernetes master are discussed so far, now the components that are present in every kubernetes node is discussed. Kubelet component runs on every node which ensures that containers run successfully in the pods present in the cluster. A network proxy called kube-proxy is installed in every node of the cluster which maintains the network rules for the kubernetes nodes. Container runtime docker is also installed in all nodes which are responsible for running containers same as the runtime used in Liu and Zhao (2014). Now after the cluster is setup, the ANA autoscaler components are installed into the master node of the cluster. The ANA autoscaler primarily consists of ANA bash script, dormant creation script and logger script. Apart from this the metrics-server component is also installed which is helpful in figuring out the CPU utilization of the cluster when testing with the generated load which is inspired from Rattihalli et al. (2019). The dormant node is a configured node in kubernetes which is put into the sleep state. Once the dormant node is created the configuration and addition to the kubernetes cluster is done, after that the kubernetes node is drained and then the node is shutdown. The dormant node is started from shutdown state when the node is required to be used in the kubernetes cluster. The ANA autoscaler always aims to maintain one dormant node at all times.

4.3 Architecture of the autoscaler

The ANA autoscaler is setup completely in the master node of the kubernetes cluster. It consists of three scripts which are create-dormant-node bash script, ANA bash script and logger script as seen in Figure 3. These three scripts are triggered periodically every minute through crontab settings present in the ubuntu operating system. Both create-dormant-node bash script and ANA bash script have lock mechanism which present multiple triggers, that is once it is triggered it does not allow additional re-trigger of the same script until the script is completed. The create-dormant script is responsible for interacting with the AWS API and spinning up an AWS instance as shown in Fernandez and Renjith (2021) using AWS command line tool. User-data is passed to the AWS API for the creation of the AWS instance, the user data contains all the commands required to install the packages and onboard the node to the kubernetes cluster. Once the node is created in the AWS, the output in the JSON form is collected, using JQuery the instance ID and the kubernetes node name is filtered similar to the way used in McCormick and De Volder (2004). These details are outputted into static files called dormant node-instance ID and dormant node-kubernetes node files. Once the output details are copied into the file, a flag file called a dormant node is created. This dormant node flag file indicates whether the dormant node is created or not. The create-dormant node script continuously checks for this flag for dormant node creation.

The ANA bash script continuously monitors the CPU utilization of the worker nodes. Once the ANA bash script realises that the CPU utilization is more than sixty percent, the dormant node is started from the shut down state and used for handling the additional load. When ever the ANA bash script uses the dormant node, the dormant flag is removed and this helps the creation of a new dormant node. Finally the logger script is trigger every minute to capture the details of the kubernetes cluster. Details like the time, the CPU utilization of the cluster and the nodes present in the cluster are written into log file. This is used to track and get the amount of time that is taken to create and use the dormant node. For the second experiment an EKS cluster with t2.micro is used with a dynamic scaling settings similar to the dynamic pool described in Ling et al. (2019), initially two worker nodes are created in the EKS setup.

5 Implementation

The implementation primarily consists of three scripts in ANA autoscaler. First is the dormant node creation script which is responsible for the creation of the dormant instance. The script is responsible for maintaining at least one dormant node in the kubernetes cluster. Second is the ANA autoscaler script which is responsible for consuming the dormant node to increase the overall resources of the kubernetes cluster. Finally a logger script is also used to get the logs of the utilization and the node information of the kubernetes cluster.

Algorithm 1 Dormant node creation

```
 1: procedure DORMANT_NODE_CREATION(dormantflag)            ▷ Create dormant node
 2:     if $dormantflag \neq 0$ then                        ▷ Check dormant node flag
 3:         Create_EC2_instance(User_Data)                  ▷ Pre-install packages
 4:         Store(Instance_ID)                              ▷ Store details in file
 5:         Store(NodeName)
 6:         Wait()                                          ▷ Wait for five minutes
 7:         Drain(NodeName)         ▷ Mark node as unschedulable in kubernetes
 8:         Shutdown(NodeName)
 9:         Create dormantflag        ▷ Set dormant flag as dormant node present
10:     else
11:         No Action required                              ▷ Dormant node present
```

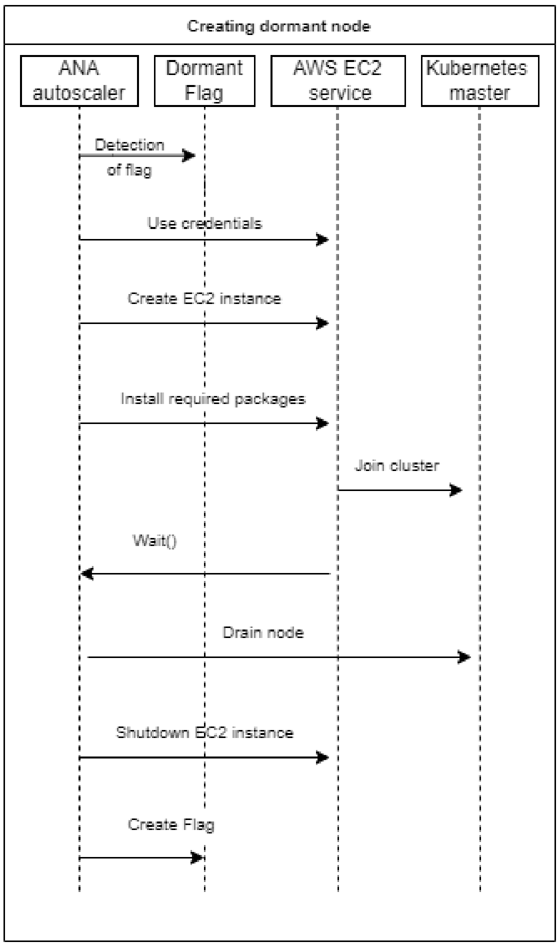
Algorithm 2 ANA Autoscaler

```
1: procedure ANA_Autoscaler($AvgCpuUtilization$)            ▷ Perform scale-out
2:     if $AvgCpuUtilization > 60\%$ then
3:         Start(NodeName)                                  ▷ Start up dormant node
4:         Wait()                                           ▷ Wait for 1 minute
5:         Schedule(NodeName)                 ▷ Schedule dormant node in kubernetes
6:         Delete(DormantFlag)                  ▷ Trigger creation of dormant node
7:     else
8:         No Action required                 ▷ Optimal utilization of resources
```

The dormant node creation script constantly checks for the dormant node flag. If the flag is present then it indicates that a single dormant node exists. If the dormant flag does not exist then the ec2 instance in aws is created along with user data. User data will provide all the details of the packages to be installed as mentioned in Vanmechelen et al. (2012) and the commands to join the kubernetes cluster. Once the ec2 instance is created the output present in the json format is filtered to get the instance id and the node name to store in files. Following this the node is drained and shutdown similar to the draining method used in Sarajlic et al. (2018). Once this is done the dormant node flag is created to indicate the completion of dormant node. The ANA autoscaler script on the other hand checks for the average cpu utilization that is used by the kubernetes cluster, once the utilization goes above sixty percent the dormant node that is in the shutdown state is started. After this the dormant node that is started is scheduled in the kubernetes cluster for the pods to be scheduled. Upon addition of the new node in the kubernetes cluster the utilization of the kubernetes cluster becomes optimum. Now the dormant flag is deleted, this will trigger the creation of the dormant node. In this manner it is ensured that always a dormant node is maintained in the kubernetes cluster by the ANA autoscaler.

6 Evaluation

The scale out time in cluster autoscaler is recorded in the experiment to understand whether the custom cluster autoscaler created could perform better than the autoscaler used in the AWS managed kubernetes cluster which is same as the autoscaler used in Liu et al. (2018). The cpu utilization is considered in both the cases of the experiment. For the generation of the load a PHP application is deployed into the kubernetes cluster using a deployment file. The deployment file will create a deployment object in the kubernetes cluster, this in turn will create replica set as utilised in Eidenbenz et al. (2020). This replica set will launch the pods into different nodes present in the kubernetes cluster. The deployment manifest file will also launch a service object which will create various endpoints in the cluster. A load generation bash script will be setup in the worker nodes, this bash script will be triggered to hit the endpoints of the PHP apache application similar to the method used in Johansson et al. (2022). Once the request reaches the endpoint, the kubernetes will load balance the application where the traffic will be sent to pods spread across worker nodes. To capture the logs a script is ran which will log the details of the cluster every minute. Through the logs it is possible to determine the time stamps and calculate the time that is taken for scale out.



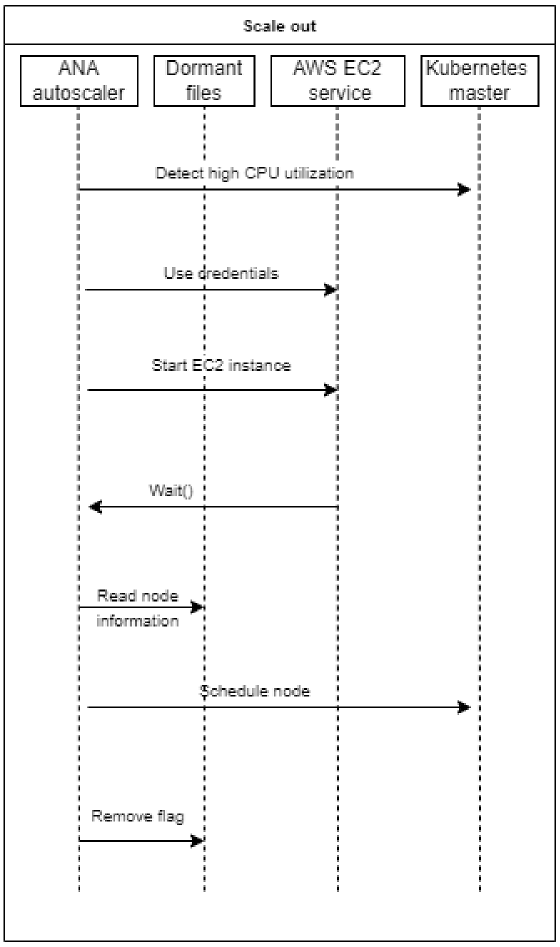
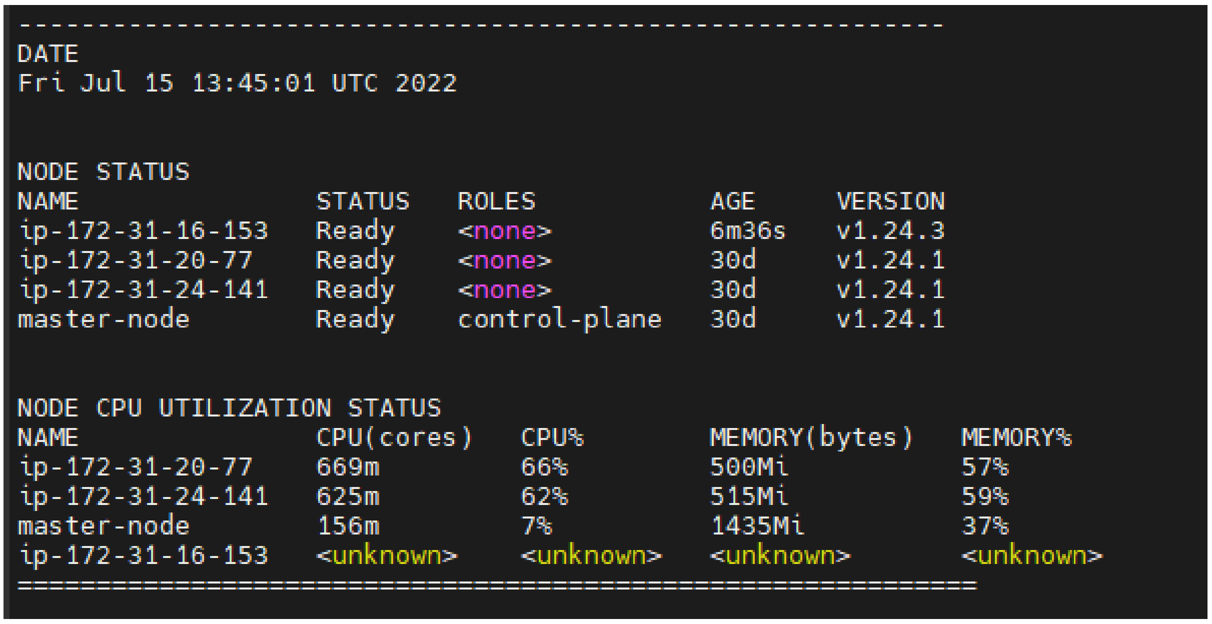


Figure 4: Detailed steps

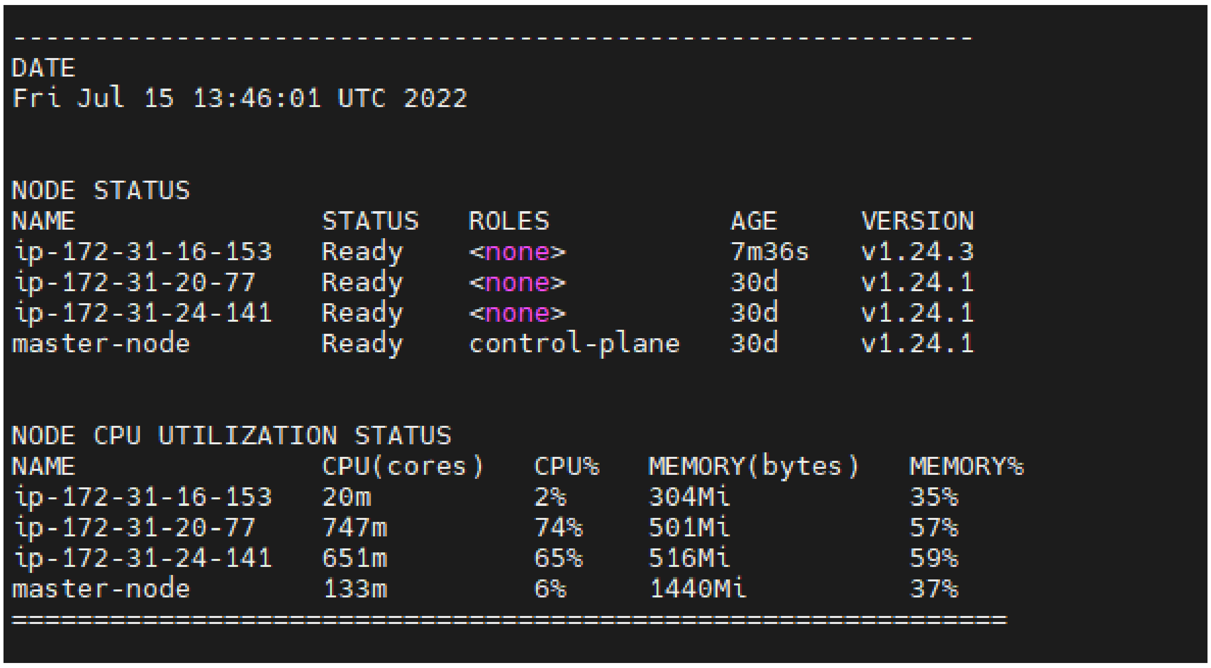


```
-----------------------------------------------------------
DATE
Fri Jul 15 13:45:01 UTC 2022


NODE STATUS
NAME               STATUS   ROLES           AGE     VERSION
ip-172-31-16-153   Ready    <none>          6m36s   v1.24.3
ip-172-31-20-77    Ready    <none>          30d     v1.24.1
ip-172-31-24-141   Ready    <none>          30d     v1.24.1
master-node        Ready    control-plane   30d     v1.24.1


NODE CPU UTILIZATION STATUS
NAME               CPU(cores)   CPU%        MEMORY(bytes)   MEMORY%
ip-172-31-20-77    669m         66%         500Mi           57%
ip-172-31-24-141   625m         62%         515Mi           59%
master-node        156m         7%          1435Mi          37%
ip-172-31-16-153   <unknown>    <unknown>   <unknown>       <unknown>
=============================================================
```

Figure 5: ANA autoscaler-before scale out test

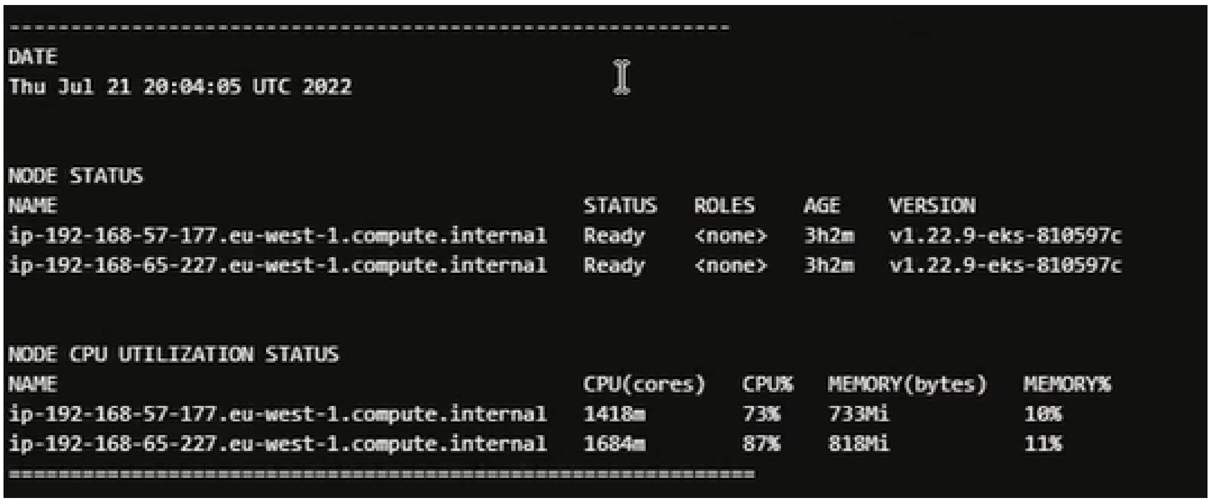


```
------------------------------------------------------------
DATE
Fri Jul 15 13:46:01 UTC 2022


NODE STATUS
NAME                STATUS   ROLES           AGE     VERSION
ip-172-31-16-153    Ready    <none>          7m36s   v1.24.3
ip-172-31-20-77     Ready    <none>          30d     v1.24.1
ip-172-31-24-141    Ready    <none>          30d     v1.24.1
master-node         Ready    control-plane   30d     v1.24.1


NODE CPU UTILIZATION STATUS
NAME                CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
ip-172-31-16-153    20m          2%     304Mi           35%
ip-172-31-20-77     747m         74%    501Mi           57%
ip-172-31-24-141    651m         65%    516Mi           59%
master-node         133m         6%     1440Mi          37%
==============================================================
```

Figure 6: ANA autoscaler-after scale out test

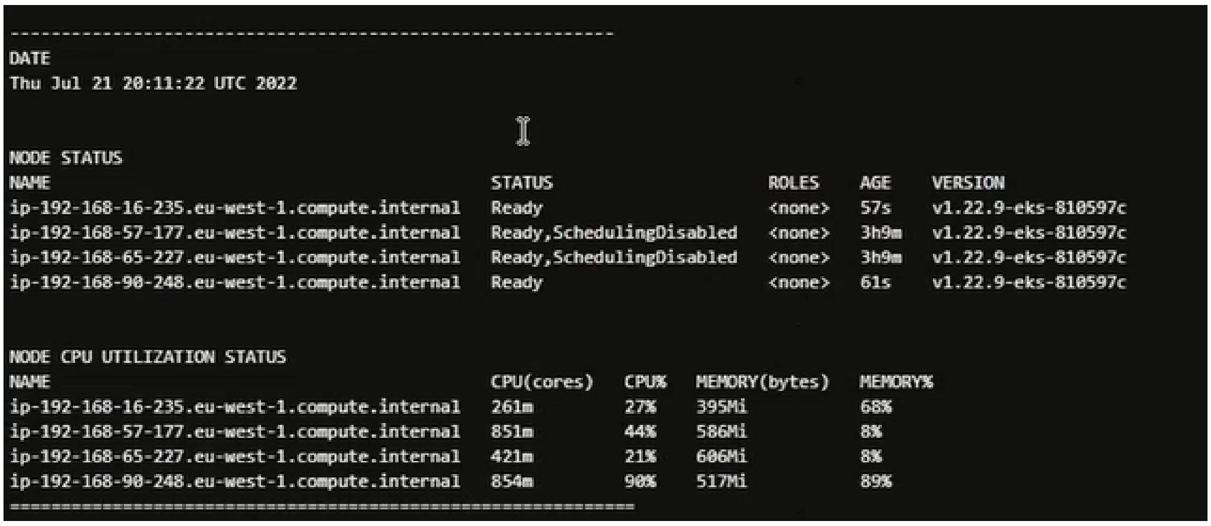


```
------------------------------------------------------------
DATE
Thu Jul 21 20:04:05 UTC 2022


NODE STATUS
NAME                                          STATUS   ROLES    AGE    VERSION
ip-192-168-57-177.eu-west-1.compute.internal  Ready    <none>   3h2m   v1.22.9-eks-810597c
ip-192-168-65-227.eu-west-1.compute.internal  Ready    <none>   3h2m   v1.22.9-eks-810597c


NODE CPU UTILIZATION STATUS
NAME                                          CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
ip-192-168-57-177.eu-west-1.compute.internal  1418m        73%    733Mi           10%
ip-192-168-65-227.eu-west-1.compute.internal  1684m        87%    818Mi           11%
==============================================================
```

Figure 7: AWS autoscaler-before scale out



```
------------------------------------------------------------
DATE
Thu Jul 21 20:11:22 UTC 2022


NODE STATUS
NAME                                          STATUS                      ROLES    AGE    VERSION
ip-192-168-16-235.eu-west-1.compute.internal  Ready                       <none>   57s    v1.22.9-eks-810597c
ip-192-168-57-177.eu-west-1.compute.internal  Ready,SchedulingDisabled    <none>   3h9m   v1.22.9-eks-810597c
ip-192-168-65-227.eu-west-1.compute.internal  Ready,SchedulingDisabled    <none>   3h9m   v1.22.9-eks-810597c
ip-192-168-90-248.eu-west-1.compute.internal  Ready                       <none>   61s    v1.22.9-eks-810597c


NODE CPU UTILIZATION STATUS
NAME                                          CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
ip-192-168-16-235.eu-west-1.compute.internal  261m         27%    395Mi           68%
ip-192-168-57-177.eu-west-1.compute.internal  851m         44%    586Mi           8%
ip-192-168-65-227.eu-west-1.compute.internal  421m         21%    606Mi           8%
ip-192-168-90-248.eu-west-1.compute.internal  854m         90%    517Mi           89%
==============================================================
```

Figure 8: AWS autoscaler-after scale out

6.1 CPU load testing with ANA autoscaler

In the first experiment the cpu load generation is performed with the custom ANA autoscaler setup. As shown in figure 5, the logs shows the state of the cluster before the load generation is performed. So as shown in the logs the node ip-172-31-16-153 is the dormant node, the ANA autoscaler realises that the average load has reached above 60 percent. Once the ANA autoscaler realises that the average CPU utilization is above threshold, the dormant node in the shutdown state is started up and consumed. At timestamp of 13:44 the dormant node was started up, by 13:45 the dormant node is in ready state. As shown in 6 the dormant node in ready state is used. The node ip-172-32-16-153 is being used in the cluster which shows a CPU utilisation of 2 percent, now the cpu load is being load balanced among the cluster. So autoscaler checks every minute in the crontab setup at the system level, once the load has reached above sixty percentage of cpu load the dormant node is started up and scheduled in the kubernetes cluster to take the cpu load. This only takes about 2 minutes to react to the CPU load increase in the traffic.

6.2 CPU load testing with EKS-AWS autoscaler

To compare the results of the ANA autoscaler the cluster autoscaler in the EKS is used as deployed in Poniszewska-Marańda and Czechowska (2021). The dynamic settings in the EKS cluster is set to autoscale when sixty percentage is reached. As seen in figure 7 the cpu load generation is started by running scripts in the worker node. At timestamp 20:04, there are two nodes present as the worker nodes. The cpu utilization is about 73 percent and 87 percent in the worker nodes respectively. The same logger script is present in the worker node to capture the logs. Once the the cpu utilization is increased the AWS cluster autoscaler reactively scales out to meet the increasing cpu utilization demand. In timestamp 20:11, four nodes are present in the EKS kubernetes cluster. The CPU utilization is balanced in the cluster and traffic is getting distributed evenly among the nodes. Now even after the nodes are utilised the cluster has started to scale in due to excess resources available. In EKS the scale out takes almost 7 minutes and 7 seconds for

Timestamp	**Node Name**	**Autoscaling type**	**Time taken**
13:44-13.46	ip-173-31-16-153	ANA autoscaler	120 seconds
20:04-20:11	ip-192-168-57-177 ip-192-168-65-227	AWS autoscaler	437 seconds
Result			**72.54 % improvement**

Table 3: Results

the cluster to react when the nodes CPU load average has reached above sixty percentage as seen in Table 3.

6.3 Discussion

The table summarises the results of the experiments that is conducted in the research. The cpu load was generated in the kubernetes cluster using the same load generation method. The workload scripts are triggered from the worker nodes in both cases where the php application endpoints are hit. In comparing the custom created ANA autoscaler with the AWS autoscaler, the cluster autoscaler can improve around 72.54 percent. The experiments are primarily focused on the cpu utilization load and only on the area of improvement in the reactive strategies of the cluster autoscaling in kubernetes cluster. The load is only applied in a linear method and the method could get complex when dealing with scale in feature as well. The method has no predictive analysis, this could also be integrated to improve the efficiency of the cluster scaling process as seen in Zhao et al. (2019). There are other metrics like memory also which requires to be tested as shown in Pereira Ferreira and Sinnott (2019). Another additional feature is the ability to add and integrate even application metrics. These things can be added to make a comprehensive solution which could be capable of choosing the best method and take the right decisions based on the best metrics. Although the research has many improvements that could be added, the results show that the instance acquisition lag in reaction to the scale out could be improved.

7 Conclusion and Future Work

The research was carried out in the area of autoscaling in kubernetes with respect to the resources available in the cluster. A custom solution called ANA autoscaler was created to improve the node acquisition time during scale out of the kubernetes cluster. To compare this, the same load generation scripts are used in the AWS managed EKS-kubernetes service. The results showed that the node acquisition time can be improved using the ANA autoscaler solution compared to the AWS cluster autoscaler. Results showed 72 percent improvement in the time taken by the kubernetes node to add the node to the cluster. The limitation of the work is that only the cpu utilisation of the kubernetes cluster is taken as the consideration for the research. This research shows that there is still scope for improvement in the area of cluster autoscaling, this will encourage to create solutions which will further improve the reactive time and compliment the horizontal pod autoscaling as seen in Nguyen et al. (2020b) and vertically autoscaling as seen in Balla et al. (2020).

For the future work there are many enhancements that could be made. The first enhancement could be to test different metrics other than the cpu metrics of the kubernetes cluster, additionally adding the ability to add custom metrics for application workload could make the solution comprehensive. The other enhancement that could be performed is that the scaling in feature could also be incorporated in addition to the scaling out feature, this would lead to the best optimal use of resources. Another extension of the research could be to add a predictive scaling in addition to the reactive scaling present, in the predictive scaling a machine learning algorithm for the best resources usage could be utilized as seen in Khaleq and Ra (2021). In an ideal world creating a complete solution with all the features would provide the best possible use of resources and in turn provide the best performance to cost ratio. Although realistically this would be very complicated and difficult to achieve, the objective of reaching the ideal solution would help in discovering different ideas and observations which are yet to be uncovered in the area of autoscaling.

References

Backes, J., Bayless, S., Cook, B., Dodge, C., Gacek, A., Hu, A. J., Kahsai, T., Kocik, B., Kotelnikov, E., Kukovec, J. et al. (2019). Reachability analysis for aws-based networks, *International Conference on Computer Aided Verification*, Springer, pp. 231–241.

Balla, D., Simon, C. and Maliosz, M. (2020). Adaptive scaling of kubernetes pods, *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–5.

Chang, C.-C., Yang, S.-R., Yeh, E.-H., Lin, P. and Jeng, J.-Y. (2017). A kubernetes-based monitoring platform for dynamic cloud resource provisioning, *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pp. 1–6.

Chang, Y.-H., Lee, C.-J. and Lin, B. (2006). Design and implementation of an automation service system based on crontab, *International Journal of Services and Standards* **2**(2): 203–214.

Eidenbenz, R., Pignolet, Y.-A. and Ryser, A. (2020). Latency-aware industrial fog application orchestration with kubernetes, *2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC)*, IEEE, pp. 164–171.

Fernandez, I. G. and Renjith, J. A. (2021). A novel approach on auto-scaling for resource scheduling using aws, *International Virtual Conference on Industry 4.0: Select Proceedings of IVCI4. 0 2020*, Vol. 355, Springer Nature, p. 99.

Hoenisch, P., Weber, I., Schulte, S., Zhu, L. and Fekete, A. (2015). Four-fold auto-scaling on a contemporary deployment platform using docker containers, *International Conference on Service-Oriented Computing*, Springer, pp. 316–323.

Ifrah, S. (2019). Deploy a containerized application with amazon eks, *Deploy Containers on AWS*, Springer, pp. 135–173.

Islam Shamim, M. S., Ahamed Bhuiyan, F. and Rahman, A. (2020). Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices, *2020 IEEE Secure Development (SecDev)*, pp. 58–64.

Johansson, B., Rågberger, M., Nolte, T. and Papadopoulos, A. V. (2022). Kubernetes orchestration of high availability distributed control systems, *Proc. ICIT*.

Khaleq, A. A. and Ra, I. (2021). Intelligent autoscaling of microservices in the cloud for real-time applications, *IEEE Access* **9**: 35464–35476.

Ling, W., Ma, L., Tian, C. and Hu, Z. (2019). Pigeon: A dynamic and efficient serverless and faas framework for private cloud, *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, IEEE, pp. 1416–1421.

Liu, B., Buyya, R. and Nadjaran Toosi, A. (2018). A fuzzy-based auto-scaler for web applications in cloud computing environments, *International Conference on Service-Oriented Computing*, Springer, pp. 797–811.

Liu, D. and Zhao, L. (2014). The research and implementation of cloud computing platform based on docker, *2014 11th international computer conference on wavelet actiev media technology and information processing (ICCWAMTIP)*, IEEE, pp. 475–478.

McCormick, E. and De Volder, K. (2004). Jquery: finding your way through tangled code, *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pp. 9–10.

Nguyen, T.-T., Yeom, Y.-J., Kim, T., Park, D.-H. and Kim, S. (2020a). Horizontal pod autoscaling in kubernetes for elastic container orchestration, *Sensors* **20**(16).
URL: *https://www.mdpi.com/1424-8220/20/16/4621*

Nguyen, T.-T., Yeom, Y.-J., Kim, T., Park, D.-H. and Kim, S. (2020b). Horizontal pod autoscaling in kubernetes for elastic container orchestration, *Sensors* **20**(16): 4621.

Pereira Ferreira, A. and Sinnott, R. (2019). A performance evaluation of containers running on managed kubernetes services, *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 199–208.

Poniszewska-Marańda, A. and Czechowska, E. (2021). Kubernetes cluster for automating software production environment, *Sensors* **21**(5): 1910.

Rattihalli, G., Govindaraju, M., Lu, H. and Tiwari, D. (2019). Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes, *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, IEEE, pp. 33–40.

Sami, H., Mourad, A. and El-Hajj, W. (2020). Vehicular-obus-as-on-demand-fogs: Resource and context aware deployment of containerized micro-services, *IEEE/ACM Transactions On Networking* **28**(2): 778–790.

Sarajlic, S., Chastang, J., Marru, S., Fischer, J. and Lowe, M. (2018). Scaling jupyterhub using kubernetes on jetstream cloud: Platform as a service for research and educational initiatives in the atmospheric sciences, *Proceedings of the Practice and Experience on Advanced Research Computing*, pp. 1–4.

Thurgood, B. and Lennon, R. G. (2019). Cloud computing with kubernetes cluster elastic scaling, *Proceedings of the 3rd International Conference on Future Networks and Distributed Systems*, pp. 1–7.

Vanmechelen, K., De Munck, S. and Broeckhove, J. (2012). Conservative distributed discrete event simulation on amazon ec2, *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, IEEE, pp. 853–860.

Vohra, D. (2017). Using autoscaling, *Kubernetes Management Design Patterns*, Springer, pp. 299–308.

Zhao, A., Huang, Q., Huang, Y., Zou, L., Chen, Z. and Song, J. (2019). Research on resource prediction model based on kubernetes container auto-scaling technology, *IOP Conference Series: Materials Science and Engineering*, Vol. 569, IOP Publishing, p. 052092.

Zheng, W.-S. and Yen, L.-H. (2018). Auto-scaling in kubernetes-based fog computing platform, *International Computer Symposium*, Springer, pp. 338–345.