

AWS EKS Shared Cluster

- [Architecture Diagram](#)
 - [Overview](#)
 - [Topics not covered in this document](#)
 - [Prepare AWS Access](#)
 - [IAM User / EC2 Instance Role and Permission](#)
 - [Prepare EC2 Jump host Local Environment](#)
 - [Prerequisites](#)
 - [Install AWS CLI](#)
 - [linux based system](#)
 - [Setup eksctl](#)
 - [Installation](#)
 - [Test](#)
 - [Setup kubectl](#)
 - [Installation](#)
 - [check kubectl](#)
 - [AWS EKS Cluster Setup](#)
 - [eksctl - The Imperative way](#)
 - [eksctl - The declarative way](#)
 - [Gather Facts](#)
 - [Create Cluster](#)
 - [Configure Cluster](#)
 - [Update Kubeconfig](#)
 - [K8s Autoscaler Deployment Steps:](#)
 - [Enable Cloudwatch logging](#)
 - [Enable Container insights with Cloudwatch Metrics](#)
 - [Setup and Configure Helm3](#)
 - [Setup and Configure Ingress-Nginx\(Kubernetes maintained\)](#)
 - [Setup K8s Dashboard](#)
 - [Federated IAM with RBAC](#)
 - [References](#)
 - [Support](#)
-

Architecture Diagram

<Diagram WIP>

Overview

On this page, one will find information on how to set up and configure **AWS EKS Private Cluster** using `eksctl` and `kubectl`. `eksctl` is a simple CLI tool for creating and managing clusters on EKS - Amazon's managed Kubernetes service for EC2. It is written in Go, uses CloudFormation, was created by Weaveworks and it welcomes contributions from the community. `eksctl` is also one among the two AWS-recommended ways of creating an AWS EKS Cluster. Once the Cluster is ready `kubectl` can be then used to configure the EKS cluster, like RBAC, Ingress, Monitoring, Logging etc

Topics not covered in this document

1. Basic understanding of Kubernetes
2. Basic understanding of AWS

Prepare AWS Access

Before starting, please make sure the below list of access/roles are granted to an IAM user or EC2 Instance Role using which the EKS cluster is created.

IAM User / EC2 Instance Role and Permission

To be able to run through this setup your IAM user / EC2 Instance Role needs to have certain privileges e.g. create all the required resources and objects. According to AWS Best Practices, you should *never* use your root account for working with AWS services.

There can be 2 approaches:

1. Provide admin access to the IAM user or the EC2 instance role, Basically, your user just requires *one* policy to be attached
 - `AdministratorAccess`

2. Provide a dedicated list of privileges/policies to cover all the required privileges, first, create additional policies
EKS-Admin-policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "eks:*"
      ],
      "Resource": "*"
    }
  ]
}
```

CloudFormation-Admin-policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "cloudformation:*"
      ],
      "Resource": "*"
    }
  ]
}
```

finally, assign the following policies to your IAM user / EC2 Instance Role:

- AmazonEC2FullAccess
- IAMFullAccess
- AmazonVPCFullAccess
- CloudFormation-Admin-policy
- EKS-Admin-policy

(where the last 2 policies are the ones you created above)

We have chosen the second approach with EC2 Instance Role. The EC2 instance with the right roles and policies are attached and provided to SC DevOps by the CloudOps team, for further clarification on the EC2 Instance Role it is advised to connect with the CloudOps team.

Prepare EC2 Jump host Local Environment

Prerequisites

- Python3 or Python2.7.9+
- Python Pip3 / Pip

Install AWS CLI

linux based system

```
pip install --user awscli
export PATH=$PATH:/home/$(whoami)/.local/bin
```

--user is used to installing the AWS CLI under your home directory, not to interfere with any existing libraries/installations

create file `~/.aws/credentials`

 Note: EC2 Instance Role only needs the region to be configured in the AWS credentials file

```
[default]
region=us-east-1
output=json
```

Setup eksctl

Installation

for non-Linux OS you can find a binary download here:

<https://github.com/weaveworks/eksctl/releases>

on Linux, you can just execute:

```
curl --silent --location "<https://github.com/weaveworks/eksctl/releases
/download/latest_release/eksctl_${uname> -s}_amd64.tar.gz" | tar xz -C
/tmp

sudo mv /tmp/eksctl /usr/local/bin
```

This utility will use the same *credentials* file as we explored for the AWS cli, located under '`~/.aws/credentials`'

Test

`eksctl version`

Setup kubectl

Installation

- on RH based Linux:

```
sudo dnf install kubernetes-clientcat <<EOF > /etc/yum.repos.d
/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=<https://packages.cloud.google.com/yum/repos/kubernetes-el7-
x86_64>
enabled=1
```

```
gpgcheck=1
repo_gpgcheck=1
gpgkey=<https://packages.cloud.google.com/yum/doc/yum-key.gpg>
<https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg>
EOF
yum install -y kubectl
```

- on Debian/Ubuntu:

```
sudo apt-get update && sudo apt-get install -y apt-transport-https
curl -s <https://packages.cloud.google.com/apt/doc/apt-key.gpg> | sudo
apt-key add -
echo "deb <https://apt.kubernetes.io/> kubernetes-xenial main" | sudo
tee -a /etc/apt/sources.list.d/kubernetes.list

sudo apt-get update
sudo apt-get install -y kubectl
```

check kubectl

- Linux:
kubectl version --short --client

AWS EKS Cluster Setup

i Note: The below sections speak about setting up AWS EKS Private Cluster Only, for creating a public cluster please refer official documentaiton <https://eksctl.io/>

To create an AWS EKS cluster using `eksctl` there are 2 ways, the imperative way, and then the declarative way.

eksctl - The Imperative way

Giving all the options over the command line interface is basically the imperative way, an example would like this:

```
eksctl create cluster --name EKS-demo --version 1.18 --region us-
east-1 --vpc-private-subnets=subnet-04c0095ee,subnet-0402d378fb --
node-private-networking --nodegroup-name eks-demo-ng1 --node-type t3.
medium --nodes 2 --nodes-min 2 --nodes-max 5 --ssh-access --ssh-
public-key eks-demo.pem --managed
```

i Note: For more details please refer <https://eksctl.io/usage/creating-and-managing-clusters/>

eksctl - The declarative way

All the above parameters mentioned in the Imperative way is now declared in an YAML file and this file is now used to create an EKS cluster. Additionally, this file can be version-controlled in BitBucket.

This is the approach chosen at SC DevOps.

```
eksctl create cluster -f eks-shared-cluster.yaml
```

Note: The content of the eks-shared-cluster.yaml is showed and explained later

Gather Facts

The AWS EKS Cluster needs to be installed in a pre-existing and predetermined VPC and private subnets.

Capture and make a note of the VPC ID and Subnet IDs from the AWS Console where the cluster's node group should reside.

In our case it is:

```
vpc:
  id: "vpc-03dd1dce314f88bc3"
subnets:
  private:
    us-east-1a:
      id: "subnet-0a06069f4b15e9090"
    us-east-1b:
      id: "subnet-02ffd1f9d45a266df"
```

Note: The actual VPC ID and Subnet IDs used are given above. This needs to be captured from AWS Console.

Create Cluster

1. Populate the eksctl config file

This file contains the configuration that is used to create the EKS Shared Cluster, The changes in this file can be tracked [here](#)

A copy of the eks-shared-cluster.yaml file is pasted below, Comments are used inline to explain each line.

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: EKS-shared                ## Name of the EKS cluster
  region: us-east-1              ## The AWS region of EKS cluster
  version: "1.18"                ## The Kubernetes version

privateCluster:                  ## Denotes, its a private EKS cluster
  enabled: true

  additionalEndpointServices:    ## Create internal/private vpc endpoints
  # For Cluster Autoscaler
  - "autoscaling"               ## Private VPC endpoint for AutoScaling
  # CloudWatch logging
  - "logs"                      ## Private VPC endpoint for Cloudwatch

  Logs
  # Cloudformation
  - "cloudformation"           ## Private VPC endpoint for
```

Cloudformation


```
vpc:
  id: "vpc-03ddldce314f88bc3"  ## Shared VPC ID
  subnets:
    private:
      us-east-1a:  ## Private Subnet 1
        id: "subnet-0a06069f4b15e9090"
      us-east-1b:  ## PPrivate Subnet 2
        id: "subnet-02ffd1f9d45a266df"

managedNodeGroups:  ## EKS Managed Node Group
- name: autoscale-ondemand  ## Name of the node group
  instanceTypes: ["t2.medium"]  ## Instance types used for the Node
  Group
  desiredCapacity: 3  ## EC2 Capacity to be running all the
  time
  maxSize: 10  ## Autoscale upto 10 instances
  iam:
    withAddonPolicies:
      autoScaler: true  ## For Autoscaling until maxSize is
  reached when required
  labels:  ## Labels which can be used later for
  nodeSelector
    nodegroup-type: stateless-workload
    instance-type: onDemand
    role: workers
  ssh:  ## The name of the public key to login
  to nodegroup EC2s
    publicKeyName: shared-eks
  privateNetworking: true  ## Creates nodegroup nodes in a private
  network

cloudWatch:
  clusterLogging:  ## Enable and forward logs to
  cloudwatch logging
  enableTypes: ["audit", "authenticator"]
```

2. Create a cluster using eksctl

```
eksctl create cluster -f eks-shared-cluster.yaml
```

 Note: If the above command fails to create node group then below section can be referred to create node groups separately

▼ Create Nodegoups separately

- The above config file can be used to create the node groups separately, use the below command to create the node group using the same above config file

```
eksctl create nodegroup --config-file=eks-shared-cluster.yaml
```


Configure Cluster

Update Kubeconfig

pull or Update new EKS cluster kubeconfig using aws cli command

```
aws eks --region us-east-1 update-kubeconfig --name EKS-shared
```

K8s Autoscaler Deployment Steps:

 Note: This is a K8s Add-On functionality and not specific to AWS EKS. The release details for the same can be found here: <https://github.com/kubernetes/autoscaler/releases>

The cluster autoscaler automatically launches additional worker nodes if more resources are needed, and shutdown worker nodes if they are underutilized. The autoscaling works within a nodegroup, hence make sure the nodegroup is created with this feature enabled by following the AWS EKS Cluster Setup section

- Deploy the autoscaler,

✓ [Expand to see the steps involved](#)

☒ The autoscaler itself

```
kubectl apply -f <https://raw.githubusercontent.com/kubernetes/autoscaler/master/cluster-autoscaler/cloudprovider/aws/examples/cluster-autoscaler-autodiscover.yaml>
```

The noodle version of the file can be found [here](#)

☒ put required annotation to the deployment

```
kubectl -n kube-system annotate deployment.apps/cluster-autoscaler cluster-autoscaler.kubernetes.io/safe-to-evict="false"
```

☒ get the autoscaler image version: open <https://github.com/kubernetes/autoscaler/releases> and get the latest release version matching your Kubernetes version, e.g. Kubernetes 1.18 => check for 1.18.n where "n" is the latest release version

☒ edit deployment and set your EKS cluster name:

```
kubectl -n kube-system edit deployment.apps/cluster-autoscaler
```

- set the image version at property `image=k8s.gcr.io/cluster-autoscaler:vx.yy.z`
- set your EKS cluster name at the end of property - `--node-group-auto-discovery=asg:tag=k8s.io/cluster-autoscaler/enabled,k8s.io/cluster-autoscaler/EKS-shared`

☒ View cluster autoscaler logs


```
kubectl -n kube-system logs deployment.apps/cluster-autoscaler
```

- Test the autoscaler

▼ [Expand to see the steps involved](#)

- ☐ create a deployment of nginx, that can be used for testing autoscaling functionality

```
kubectl apply -f test-autoscaler.yaml
```

 Note: the content of the test-autoscaler.yaml can be found [here](#)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-autoscaler
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        service: nginx
        app: nginx
    spec:
      containers:
        - image: nginx
          name: test-autoscaler
          resources:
            limits:
              cpu: 300m
              memory: 512Mi
            requests:
              cpu: 300m
              memory: 512Mi
      nodeSelector:
        instance-type: onDemand
```

- ☐ scale the deployment

```
kubectl scale --replicas=21 deployment/test-autoscaler
```


Now, the pod is configured to run 21 replicas, this is only for testing the autoscaling feature on the nodegroup. Please remember to remove this deployment after testing!

☐ check pods

```
kubectl get pods -o wide --watch
```

At this stage few pods will be in pending stage until the auto scaler spins up the new nodes in the nodegroup and makes them available for the new pods to be placed

☐ Check nodes


```
kubectl get nodes
```

It can be seen that the number of nodes in the nodegroups have increased to accommodate new pods

☐ View cluster autoscaler logs

```
kubectl -n kube-system logs deployment.apps/cluster-autoscaler | grep -A5 "Expanding Node Group"

kubectl -n kube-system logs deployment.apps/cluster-autoscaler | grep -A5 "removing node"
```

 Please remember to remove the test autoscaler deployment after testing

Enable Cloudwatch logging

eksctl will take care of the cloudwatch logging via cloudformation template at the backend, all we need to do to enable it is provide the below config in the cluster configuration file

```
cloudWatch:
  clusterLogging:
    enableTypes: ["audit", "authenticator"]
```

Enable Container insights with Cloudwatch Metrics

In order to send node or pod metrics to Cloudwatch there has to be an additional IAM policy that needs to be attached to the nodegroup role name, follow the steps for setting up AWS Cloudwatch Container Insights for AWS EKS Cluster

- Add a policy to your nodegroup(s)
- ▼ [Expand to see the steps involved](#)
- add policy *CloudWatchAgentServerPolicy* to nodegroup(s) role

Can be done via command line as well:


```
aws iam attach-role-policy --policy-arn arn:aws:iam::aws:policy/CloudWatchAgentServerPolicy --role-name eksctl-EKS-shared-nodegroup-autos-NodeInstanceRole-BFZL4VLQRT3S
```

- Deploy the cloudwatch agent

▼ [Expand to see the steps involved](#)

The cloudwatch agent runs as kubernetes daemonset, means one per node

```
curl <https://raw.githubusercontent.com/aws-samples/amazon-cloudwatch-container-insights/master/k8s-yaml-templates/quickstart/cwagent-fluentd-quickstart.yaml> | sed "s/{{cluster_name}}/EKS-shared;/s/{{region_name}}/us-east-1/" | kubectl apply -f -
```

 The noodle version of the file can be found [here](#)

- Load generation

▼ [Expand to see the steps involved](#)

- Below commands will run a webserver container (php-apache) and then a busybox image container (load-generator) that runs a simple load generator to hit the webserver in an infinite loop

```
kubectl run php-apache --image=k8s.gcr.io/hpa-example --requests=cpu=200m --limits=cpu=500m --expose --port=80
kubectl run --generator=run-pod/v1 -it --rm load-generator --image=busybox /bin/sh
OR
kubectl run -it --rm load-generator --image=busybox /bin/sh
```

Hit enter for command prompt

```
while true; do wget -q -O- <http://php-apache.default.svc.cluster.local;> done
```

Notice the metrics being populated in the AWS console under Cloudwatch >> Container Metrics.

 Note: Please make sure to stop the load generation loop given above and kill the container

Setup and Configure Helm3

- Install Helm3

```
curl https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 | bash
```

- Adding the K8s official helm repository and name it as "stable"

```
helm repo add stable <https://charts.helm.sh/stable>
```

- List the current helm repository

```
helm repo list
```

- Update Helm repo

```
helm repo update
```

- List / Search all the charts that are contained in a repository

```
helm search repo
```

- Test helm by installing a helm chart for Redis from the stable repository

```
helm install redis-test stable/redis
```

- List Charts deployed

```
helm ls
```

- Delete / Remove a deployed helm chart

```
helm uninstall redis-test
```

- Install a chart from the present directory

```
helm install --set name=prod myredis ./redis
```

Setup and Configure Ingress-Nginx(Kubernetes maintained)

- Deploy Ingress
- ▼ [Expand to see the steps involved](#)
 - Deploy ingress-nginx

Use this [file](#) to deploy the ingress-nginx, please note this file is customized and tested as per our requirement.

```
kubectl apply -f aws-ingress-nlb-inginx.yaml
```

Note: The above command will create an ingress controller and an internal AWS NLB. The subnet needs the below tags for the above step to complete successfully.

tags:

[kubernetes.io/role/internal-elb](#) : 1

- Configure custom TCP ports, if required.

✓ [Expand to see the steps involved](#)

- Configuring TCP port 3306 for MySQL to listen on NLB

For any TCP ports that need to listen on NLB, it should be added or edited through a config map available called `tcp-services` in the namespace `ingress-nginx`. The complete config map is available [here](#). Then apply the changes

```
kubectl apply -f tcp-services.yaml
```

A copy of the `tcp-services` config map is pasted below for a quick reference.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: tcp-services
  namespace: ingress-nginx
data:
  3306: "default/mysql:3306"          ## read-as: namespace/service-name
```

- Create a K8s service called `mysql` in default namespace listening on port 3306

- Validate Ingress

✓ [Expand to see the steps involved](#)

- List the NLB and copy the External IP address of the `ingress-nginx-controller`

```
kubectl get svc -n ingress-nginx
```

The output should look like this:

```
[shebin.babu@ip-172-19-22-164 ~]$ kubectl get svc -n ingress-nginx
NAME                                TYPE                CLUSTER-IP      EXTERNAL-IP      PORT(S)                                     AGE
ingress-nginx-controller            LoadBalancer        10.100.192.216   a8f6d644ac04a44e7ae5b32cfdb84fc7-e0a6e7720456742d.elb.us-east-1.amazonaws.com   80:32203/TCP,443:32653/TCP,3306:31661/TCP   6d4h
ingress-nginx-controller-admission  ClusterIP            10.100.204.156   <none>           443/TCP                                     6d4h
```


- Try connecting the external IP which is basically the DNS endpoint name of the NLB via telnet on port 3306, using the command below,

```
telnet a8f6d644ac04a44e7ae5b32cfdb84fc7-e0a6e7720456742d.elb.us-east-1.amazonaws.com 3306
```

The output should look like this:

```
[shebin.babu@ip-172-19-22-164 ~]$ telnet a8f6d644ac04a44e7ae5b32cfdb84fc7-e0a6e7720456742d.elb.us-east-1.amazonaws.com 3306
Trying 172.19.22.172...
```

```
Connected to a8f6d644ac04a44e7ae5b32cfdb84fc7-e0a6e7720456742d.elb.us-east-1.amazonaws.com.
Escape character is '^]'.
J
vbf`fy13e<mysql_native_passwordquit
!#08S01Got packets out of orderConnection closed by foreign host.
```

 By default NLB listens only on 443 and 80, the custom TCP ports are configured later

Setup K8s Dashboard

- Deploy the K8s Metrics Server

▼ [Expand to see the steps involved](#)

```
sudo apt install jq curl

DOWNLOAD_URL=$(curl --silent "<https://api.github.com/repos
/kubernetes-sigs/metrics-server/releases/latest"> | jq -r .
tarball_url)

DOWNLOAD_VERSION=$(grep -o '^[^/]*$' <<< $DOWNLOAD_URL)

curl -Ls $DOWNLOAD_URL -o metrics-server-$DOWNLOAD_VERSION.tar.gz

mkdir metrics-server-$DOWNLOAD_VERSION

tar -xzf metrics-server-$DOWNLOAD_VERSION.tar.gz --directory metrics-
server-$DOWNLOAD_VERSION --strip-components 1

kubectl apply -f metrics-server-$DOWNLOAD_VERSION/deploy/1.8+//
```

- Verification

```
kubectl get deployment metrics-server -n kube-system
```

- Deploy the K8s dashboard

▼ [Expand to see the steps involved](#)

- grab the latest release here: <https://github.com/kubernetes/dashboard/releases>

```
export DASHBOARD_RELEASE=v2.0.3

kubectl apply -f <https://raw.githubusercontent.com/kubernetes
/dashboard/$DASHBOARD_RELEASE/aio/deploy/recommended.yaml>
```

- Create an admin user account

```
kubectl apply -f admin-service-account.yaml
```

- access the dashboard
- ▼ Expand to see the steps involved
 - Create an ingress in the default namespace to access K8s dashboard using NLB External DNS

ingress-dashboard.yaml

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/backend-protocol: "HTTPS"
  name: ingress-dashboard
  namespace: kubernetes-dashboard
spec:
  rules:
  - host: dashboard-eks-shared.noodle.ai
    http:
      paths:
      - backend:
          serviceName: kubernetes-dashboard
          servicePort: 443
        path: /
```

- Apply the ingress config

```
kubectl apply -f ingress-dashboard.yaml
```

- Get the security Token and save it to login to the dashboard,

```
kubectl -n kube-system describe secret $(kubectl -n kube-system get secret | grep eks-course-admin | awk '{print $1}')
```

- On FRAI-Kel-Pod01 Desktop in Citrix Workspace, run the following command in the terminal to tunnel a connection between FRAI POD VM and EKS Shared Jump host,

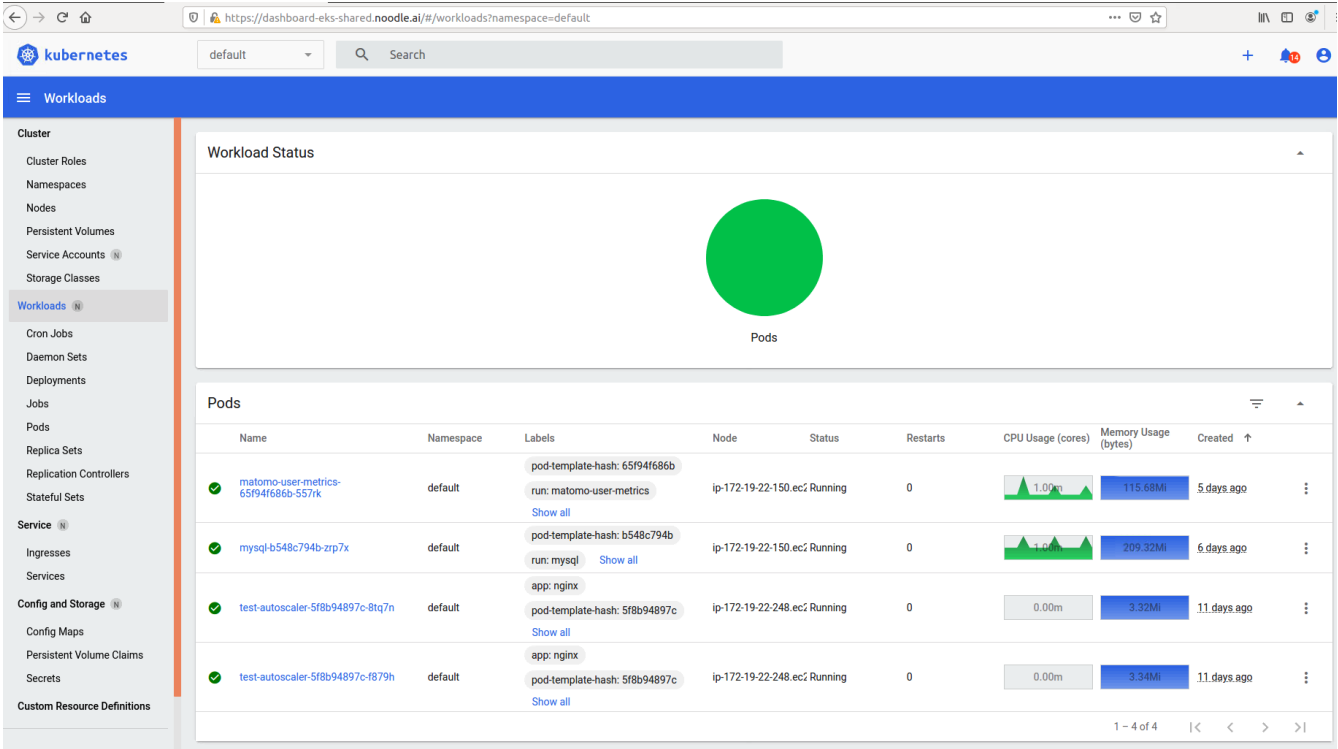
```
ssh -D 1337 -q -C -N username@172.19.22.164
```

- Then From Citrix Workspace access FRAI Host and then Google Chrome and enter the URL given below, choose *token* as a login method, where you have to provide the token.

```
https://dashboard-eks-shared.noodle.ai/#/overview?namespace=default
```

 /etc/hosts - should contain the IP/DNS endpoint address for the NLB in the EKS-shared-jumphost machine.

The dashboard should look like this:



Federated IAM with RBAC

WIP

References

- [EKS User Guide](#)
- [Production Readiness Guide](#)
- [EKS Private cluster](#)
- [EKS Cluster insights](#)
- [EKS Networking](#)
- [EKS Sec Group Considerations](#)
- [EKS CNI Calico](#)
- [EKS CNI Pod Networking\(default\)](#)
- [Max pods EC2 templates](#)
- [EKS Cluster endpoint](#)
- [EKS VPC Endpoints](#)
- [eksctl](#)
 - [eksctl Config File Schema](#)
 - [eksctl Getting Started](#)
 - [Example config 1](#)
 - [Example config 2](#)
- [EKS ALB Load Balancer Controller](#)
- [Kubernetes Ingress Nginx Controller](#)
 - [Official Documentation](#)
 - [Examples1](#)
 - [Examples2](#)
 - [NLB with Ingress-Nginx, How it works](#)
 - [V1.18 Ingress Config](#)
 - [Ingress Controller](#)

Support