Infra Project

DESCRIPTION

Create a DevOps infrastructure for an e-commerce application to run on high-availability mode.

**Background of the problem statement:**  
A popular payment application, **EasyPay**where users add money to their wallet accounts, faces an issue in its payment success rate. The timeout that occurs with  
the connectivity of the database has been the reason for the issue.  
While troubleshooting, it is found that the database server has several downtime instances at irregular intervals. This situation compels the company to create their own infrastructure that runs in high-availability mode.  
Given that online shopping experiences continue to evolve as per customer expectations, the developers are driven to make their app more reliable, fast, and secure for improving the performance of the current system.

**Implementation requirements:**

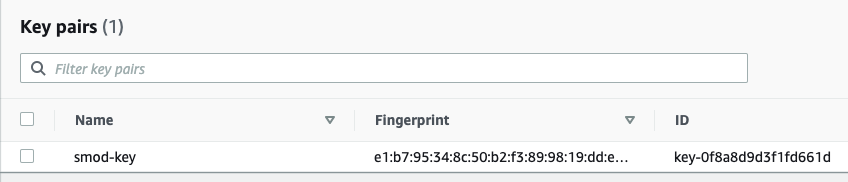
1. Create the cluster (EC2 instances with load balancer and elastic IP in case of AWS)
2. Automate the provisioning of an EC2 instance using Ansible or Chef Puppet
3. Install Docker and Kubernetes on the cluster
4. Implement the network policies at the database pod to allow ingress traffic from the front-end application pod
5. Create a new user with permissions to create, list, get, update, and delete pods
6. Configure application on the pod
7. Take snapshot of ETCD database
8. Set criteria such that if the memory of CPU goes beyond 50%, environments automatically get scaled up and configured

Refer to github.com/smodarressi/kube-infra-proj for all the files used and extra notes.

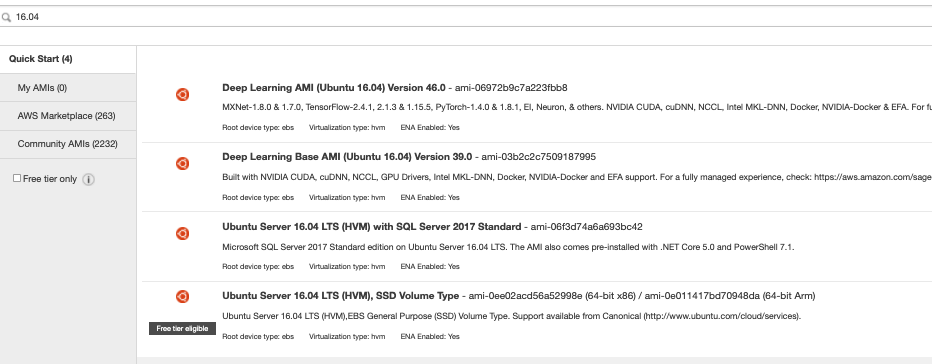
* 1. Create a cluster with EC2 instances and load balancer and elastic IP. Provision instances with ansible.

To create this, the AWS console automatically provisions default resources that can be used. It creates a default VPC, subnets, Internet Gateway, Security Group with network Ingress and Egress. Typically in a production environment, there will already be these resources and a VPC. The usage of ansible attach the new EC2 instances to existing resources. After the EC2 instances are created, network ingress and load balancer needs to be created for the resources. Ideally, you would use an application load balancer. But since the Kubernetes cluster will have its own metrics server, we only need to setup a network load balancer to direct incoming traffic.

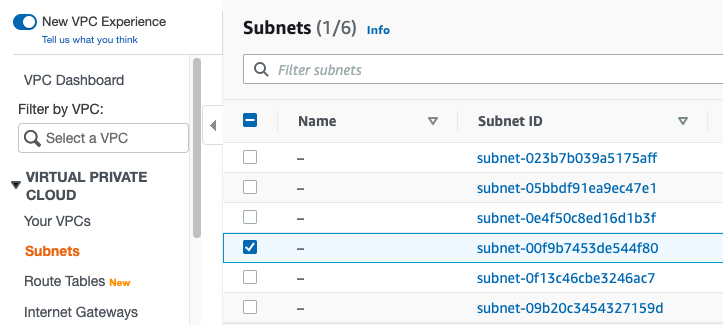
* 1. First python 3 and boto3, and ansible need to be present. Boto3 is a python module that allows ansible to interact with the AWS API.
     + Use: which python2; which python3 to see if existing python environment exist
     + Use: sudo apt install python2 python3 to install
     + Use: pip3 install boto3
     + Use: sudo apt install ansible
  2. Using ansible and ansible playbook, 3 EC2 instances can be made for the Kubernetes cluster. It will be provisioned in the subnet of your choosing.
     + Values needed from AWS: accesskey, secret key, token (if applicable), ami\_id, and subnet\_id
     + accesskey, secret key are from AWS user creation in IAM for programmatic access. Since we are using the lab, we need a token as well. A key from EC2 -> Keypairs needs to be created for the ansible yaml.



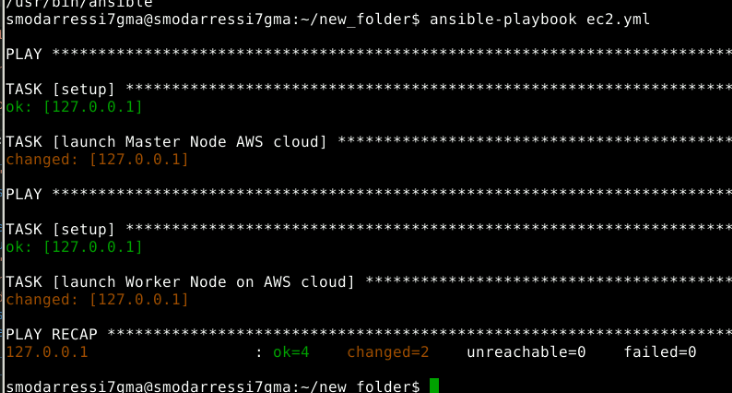
* + - We will use the ami\_id for Ubuntu 16.04 to provision the EC2 instances. To get that, search for it within the EC2 instance setup in AWS.



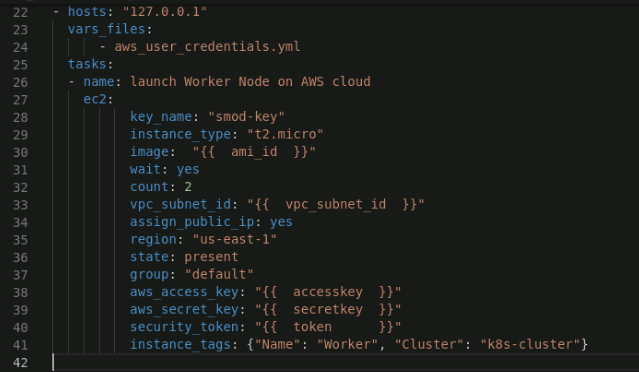
* + - For subnet\_id we check under the Subnets section in VPC.



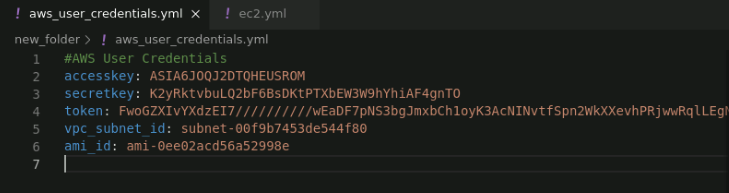
* + - Using the ansible yaml script, it will provision and label the Master and Worker nodes.



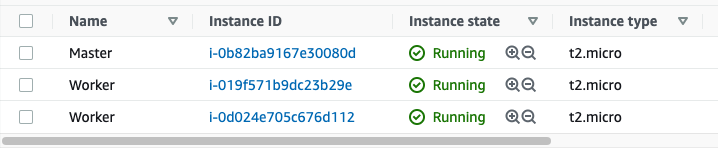
ec2.yml file example – same format used for master node



Aws\_user\_credential.yml (note: since these are temp credentials, it will be outdated by the time this gets posted. Ideally, you wouldn’t want to post this)



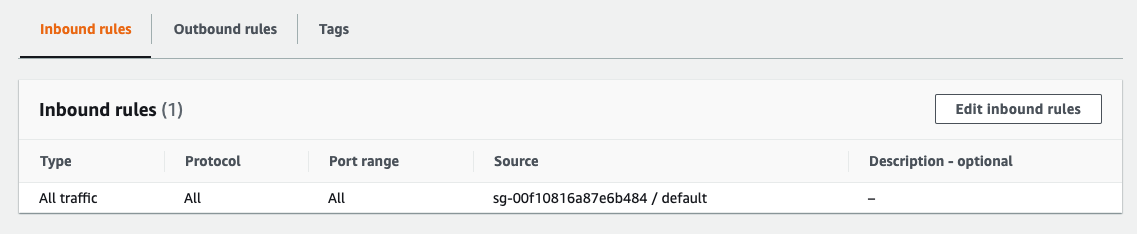
The result:



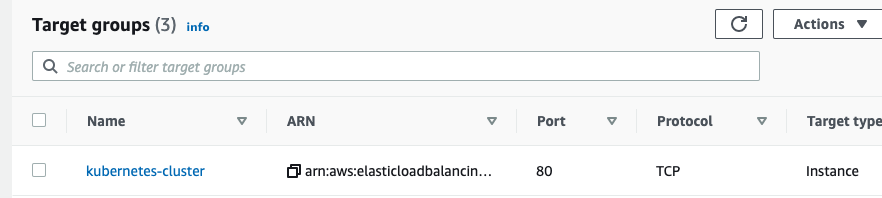
* + - If newer ansible playbooks are needed to setup the production environments within the nodes, then the ip addresses needs to be added within the /etc/ansible/host file
  1. Need to setup the security group of the Kubernetes cluster. Ingress rules for the Kubernetes cluster, weave networking, http, and https need to be created.

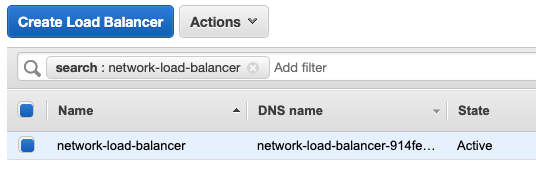
Refer to: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/ for ports that need to be opened.

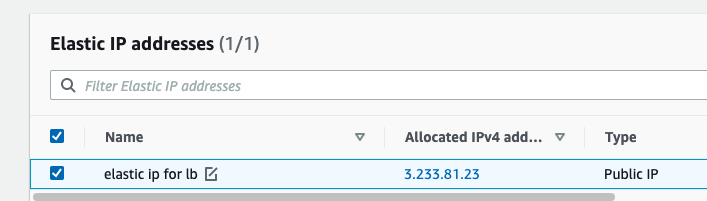
* + - Under the Security group of interest, inbound rules needs to have the referred ports, weave ports of tcp 6783, udp 6783-6784, http/https, and ssh



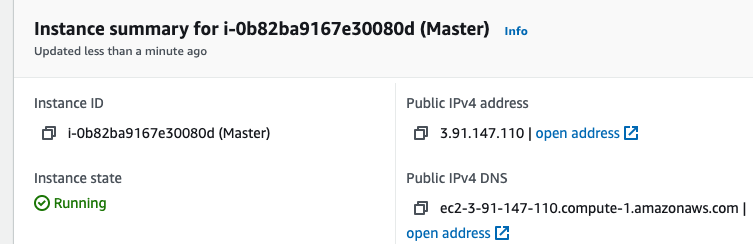
* + - Since it also asks for an elastic ip, we can allocate one and attach a network load balancer to it.
    - Under EC2 -> Elastic IPs provision 1 Elastic IP
    - For the Load balancer to work, we need a target group. This can refer to the EC2 instances where the Kubernetes cluster is located.
    - Under EC2 -> Load Balancer, provision a network load balancer.



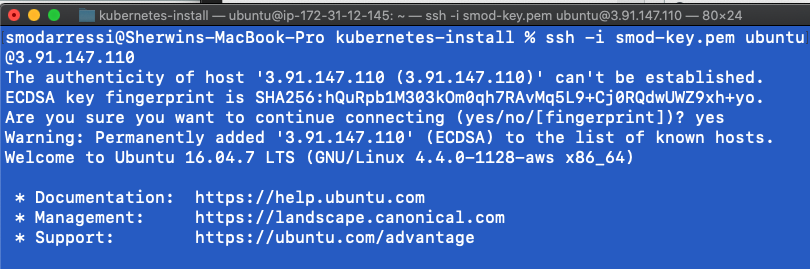




* + - To see if ssh is working and setup the Kubernetes cluster, we need the IP of all the nodes. This is in EC2 -> Instances (check details)



* + - ssh -i smod-key.pem ubuntu@<master-IP> for the next step

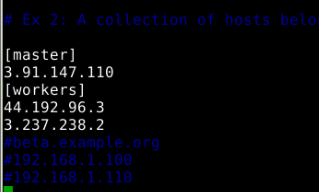


3. Setup Docker and Kubernetes on all nodes

Since ansible is used, we can use it to move and run the scripts I made to install Docker and Kubernetes. Ideally there would be python on these nodes, but it needs to be installed beforehand. If there is a production ami\_id, then this can be used to make it work.

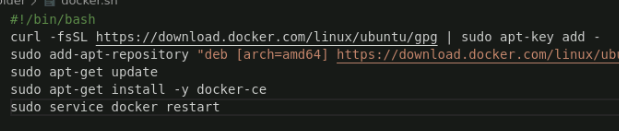
* + - We need to add the master and worker ip to the host file.

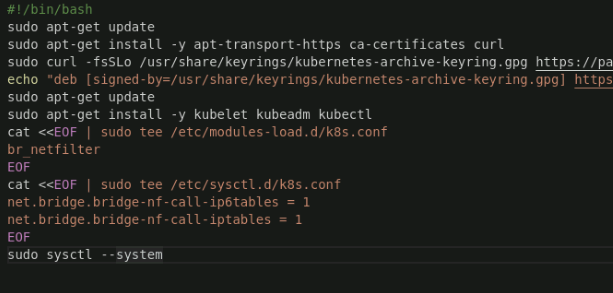
cd /etc/ansible/host

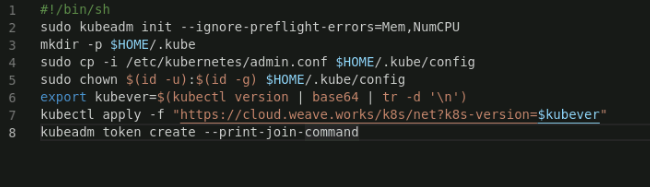


* + - Then we use 2 ansible-playbooks to move the scripts to the master and workers to run. It will then run the kubeadm setup to make the master node of the cluster. Once it is done we get the join command needed.

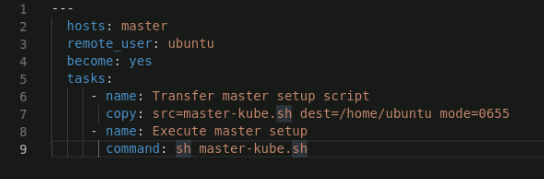
Scripts:

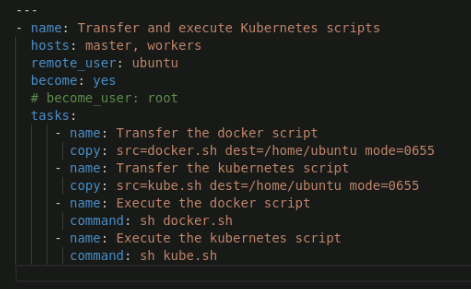


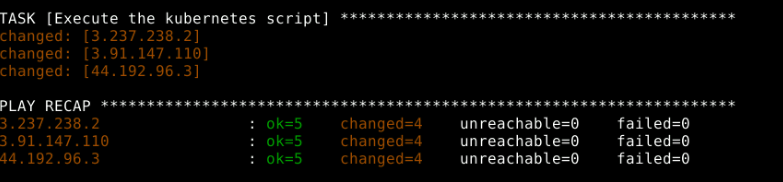




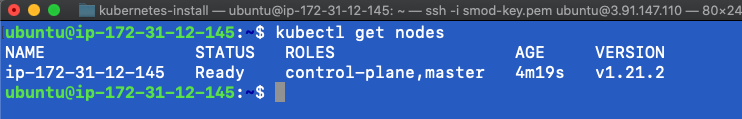
Yaml ansible playbooks:





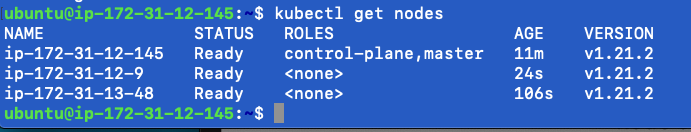


* + - Then the master-setup.yml playbook is ran
    - We then check to see if the master is setup and ready



* + - To attach the other nodes, we need the join command

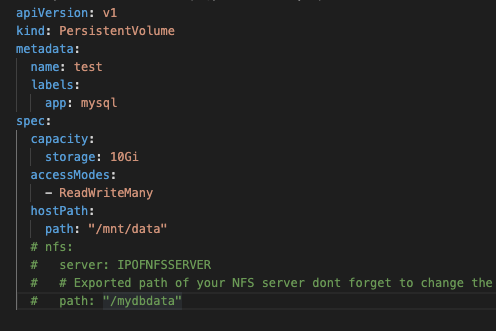
kubeadm token create --print-join-command

* + - ssh into those nodes and run the command that was printed to join (use sudo)
    - Once successful you should see 

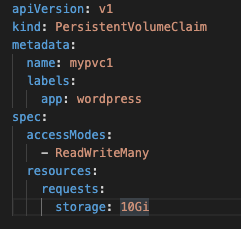
4. Implement the network policies at the database pod to allow ingress traffic from the front-end application pod

Since an example web app is needed, I will use wordpress for the web app pod and my-sql for the database pods. These will all be in yaml files. To separate confusion, I will put it in its own folder.

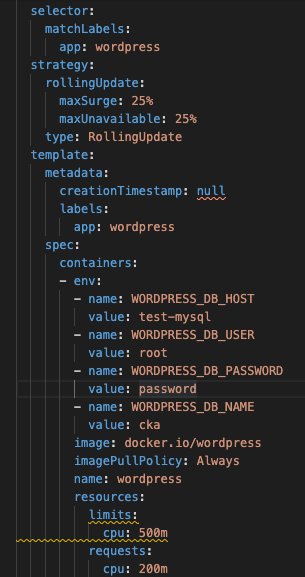
pv.yml



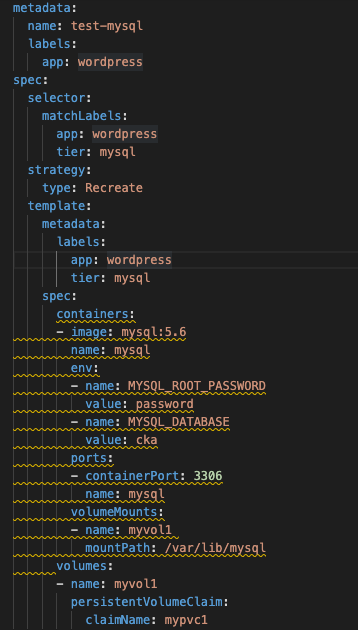
pvc.yml



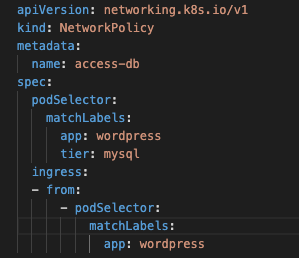
wordpress.yml



my-sql-deploy.yml



network-ingress.yml



Since my-sql is being used, it needs a persistant storage. First you need to allocate a persistant volume. In production, this could be a nfs mount, but in this case, we will use a local folder mount (seen under pv.yml). Then a persistant volume claim needs to be made to allocate the storage to the labeled pods with the (app:wordpress) in the label within file pvc.yml.

Under the mysql-deploy.yml, it will have the label (app:wordpress), selector (app: wordpress, tier: mysql) for the service resource, the env variables to setup the database, and the container to setup the pod

Under wordpress.yml, it will have the label (app:wordpress), selector (app: wordpress) for the service resource, the env variables to setup the database connection to mysql, the container to setup the pod, and allocated resources for the later step of scaling up.

All commands are run with: kubectl apply -f <name-of-yaml>.yml

To make sure everything is running, we run these commands:

#Setup and check mysql deployment

kubectl create -f mysql-deploy.yml

kubectl expose deployment test-mysql --port=3306

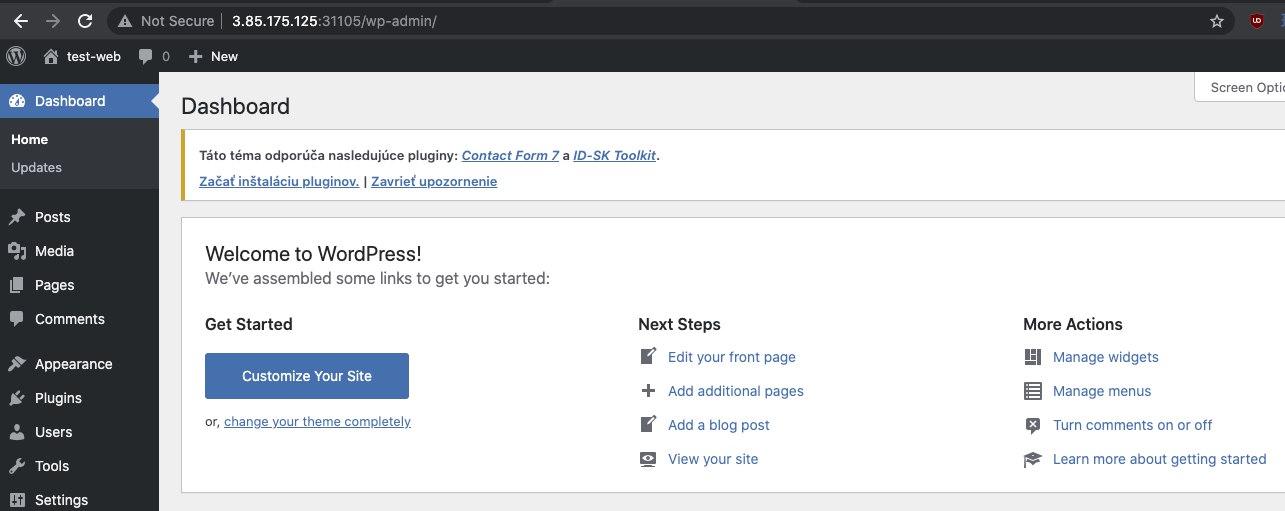
kubectl get svc

#Setup and check wordpress deployment

kubectl create -f wordpress.yml

kubectl expose deployment wordpress --port=80 --type=NodePort

These commands do two things: expose the mysql port for the other deployments to refer to within the cluster, and expose the wordpress app to the open port within the node it is in. The wordpress app should be working under the port specified when you try to connect to any of the nodes on the cluster.



Now we need a network policy that only allows wordpress web app connection to the mysql deployment.

In the network-ingress.yml file, we have the ingress from with the label app:wordpress (the wordpress pod), to the mysql pod selector(app: wordpress, tier: mysql). This makes sure that if another pod is provisioned in that namespace (default), it won’t allow a connection to the mysql server.

5. Create a new user with permissions to create, list, get, update, and delete pods

For this, a role and role-binding needs to be created.

Cluster Role - admin, view, cluster-admin

CRB = CR + SA

RBAC - Kubectl clients

Show kubectl client usage with a new user. This is determined by the set-context in the .kube/config file that includes the service accounts that are present with the role-bindings.

Need namespace - we will use the default namespace

In the master node:

mkdir kubelet-client-user

cd kubelet-client-user/

Make the credential keys in the kubelet-client-user directory

Only important to use these values when creating the keys.

Organization name → namespace (default)

common name → user-name (new-user)

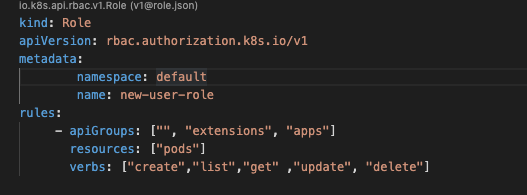
sudo openssl genrsa -out new-user.key 2048

sudo openssl req -new -key new-user.key -out new-user.csr

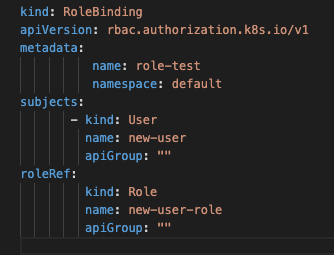
sudo openssl x509 -req -in new-user.csr -CA /etc/kubernetes/pki/ca.crt -CAkey /etc/kubernetes/pki/ca.key -CAcreateserial -out new-user.crt -days 500

Refer to role-binding and role yaml

role.yml



rolebinding.yaml



These will create the user in the role.yml with the commands specified in the problem statement and role-binding.yml

kubectl create -f role.yaml

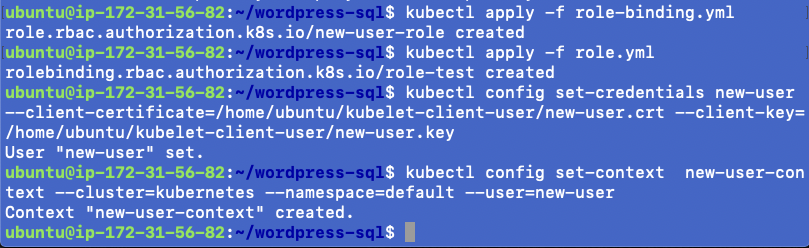
kubectl create -f rolebinding.yaml

Set credentials (this will add into .kube/config file)

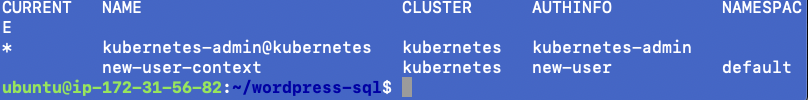
kubectl config set-credentials new-user --client-certificate=/home/ubuntu/kubelet-client-user/new-user.crt --client-key=/home/ubuntu/kubelet-client-user/new-user.key

Set context to default Namespace:

kubectl config set-context new-user-context --cluster=kubernetes --namespace=default --user=new-user



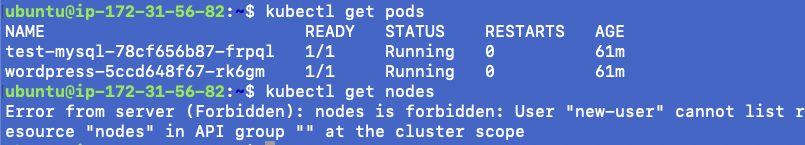
kubectl config get-contexts



To change the user, go into $HOME/.kube/config and change “current-context” value to the user you want to control the kubectl. In this case it is “new-user-context”



Now if you use kubectl, you can only interact with pods.

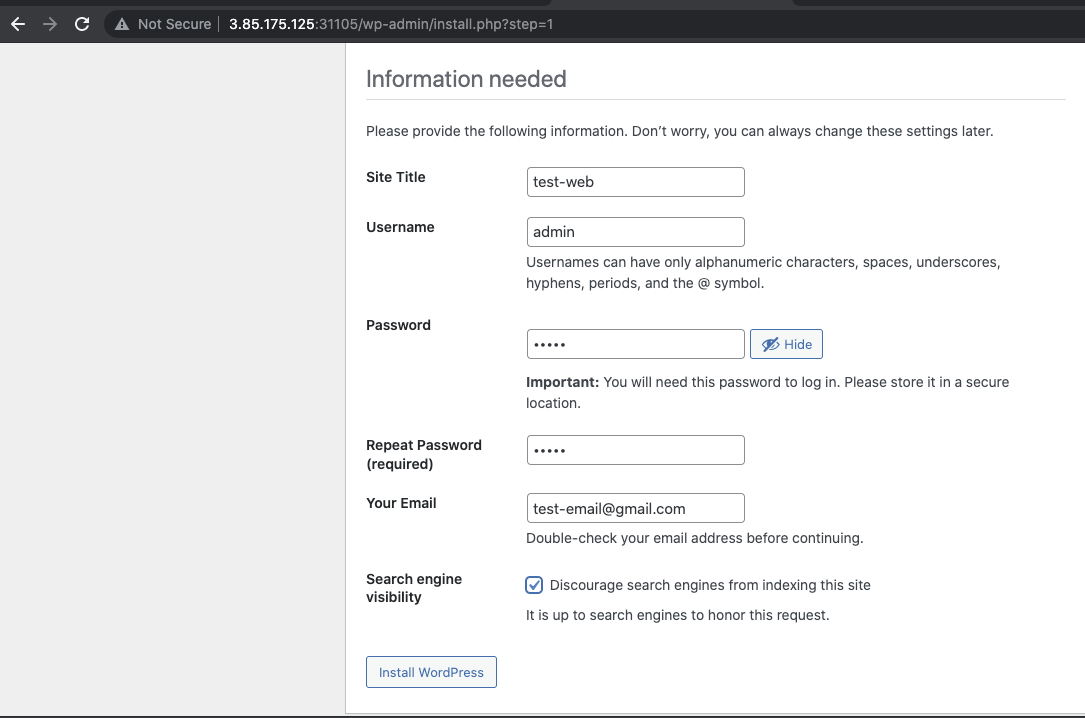


Using the master config, you can make a configuration for the worker nodes that uses this user only to control those kubelets clients.

1. ssh it a worker node
2. mkdir $HOME/.kube
3. cd $HOME/.kube
4. copy the config from the master to this node
5. Create the folder where your certificates are for the new-user
6. Remove the admin user from the users: section leaving the new-user left.
7. Change current-context to new-user-context

6. Configure application on the pod

Up to this point, many configurations have happened within the application pod like database associations. For further configuration of the pod, you can go inside and change configurations as needed. Wordpress configuration can happen within the admin area of the application: ipaddress:port/wp-admin.



But if the web app was something else, you would do these steps

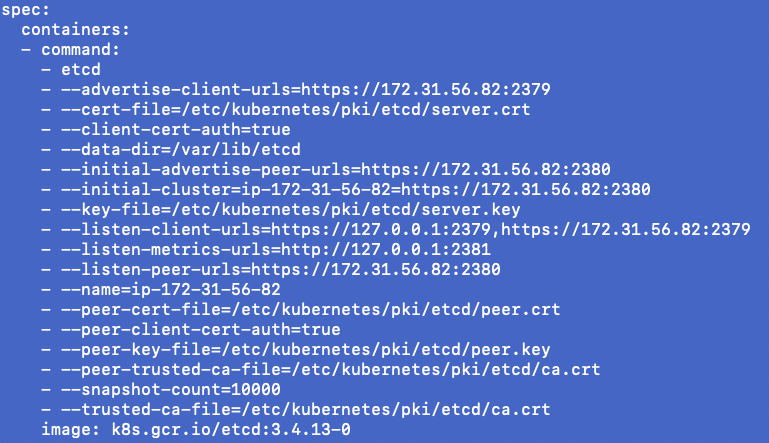
1. To go into the pod run command to get into its terminal:

Kubectl exec -it <pod-name> -- /bin/bash

1. Read the documentation for the application installed to find the directory of the config files for the specific application.

7. Take snapshot of ETCD database

Etcd pod config: etcd.yml



First check cat /etc/kubernetes/manifests/etcd.yaml

/etc/kubernetes/manifests is location where kubernetes save control plane/static pods yaml file

Information from this file is required to get etcd configuration

Important info:

--advertise-client-urls

--cert-file

--peer-cert-file

--peer-key-file

--data-dir

--initial-advertise-peer-urls

--initial-cluster

# Backup certificates

sudo cp -r /etc/kubernetes/pki backup/

# Make etcd snapshot

sudo docker run --rm -v $(pwd)/backup:/backup \

--network host \

-v /etc/kubernetes/pki/etcd:/etc/kubernetes/pki/etcd \

--env ETCDCTL\_API=3 \

k8s.gcr.io/etcd:3.4.13-0 \

etcdctl --endpoints=https://127.0.0.1:2379 \

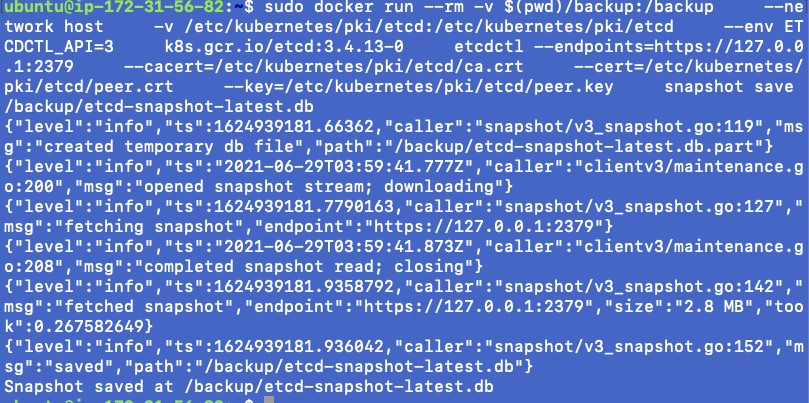
--cacert=/etc/kubernetes/pki/etcd/ca.crt \

--cert=/etc/kubernetes/pki/etcd/peer.crt \

--key=/etc/kubernetes/pki/etcd/peer.key \

snapshot save /backup/etcd-snapshot-latest.db

You will see something like this



To elaborate the command used:

Since we are using docker containers, docker command is used for the etcd container

--network host: use host network to access

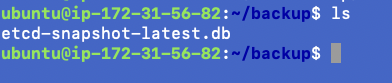
-v $(pwd)/backup:/backup --> mount the /backup folder

--env ETCDCTL\_API=3 k8s.gcr.io/etcd:3.4.13-0 --> use etcd version that was pulled with kubeadm

--endpoint --> use localhost:2379 to access the exposed etcd port

--cert, --cacert, --key for etcd specified certificates

etcdctl snapshot save /backup/etcd-snapshot-latest.db --> base command for creating a snapshot



8. Set criteria such that if the memory of CPU goes beyond 50%, environments automatically get scaled up and configured

In reference to the wordpress creation, resources values were setup. To use this we need a metrics server. Then we need a scaling yaml to horizontal scale the pod if it exceed a certain value.

kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml

kubectl get pods -n kube-system

wget -c https://gist.githubusercontent.com/initcron/1a2bd25353e1faa22a0ad41ad1c01b62/raw/008e23f9fbf4d7e2cf79df1dd008de2f1db62a10/k8s-metrics-server.patch.yaml

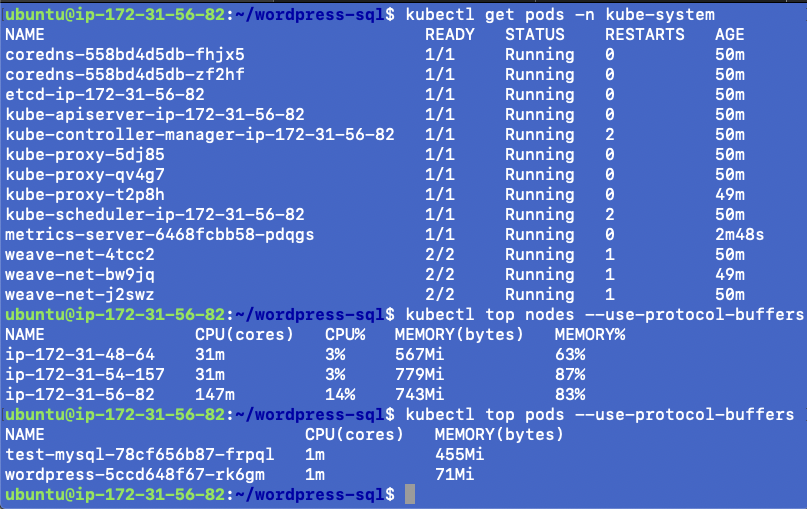
kubectl patch deploy metrics-server -p "$(cat k8s-metrics-server.patch.yaml)" -n kube-system

To check to see if the metric server is working

kubectl get pods -n kube-system

kubectl top nodes --use-protocol-buffers

kubectl top pods --use-protocol-buffers



First it needs a base allocation of cpu for the pod

This was added beforehand for this step in the wordpress deployment yaml

Specific part is below:

resources:

limits:

cpu: 500m

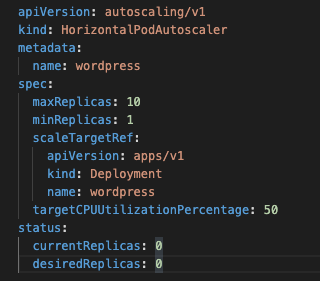
requests:

cpu: 200m

Now we need a horizontalpod autoscaler for scale up at the threshold value of 50.

kubectl autoscale deployment wordpress --cpu-percent=50 --min=1 --max=10

#this is seen in hpa-scaling.yml



Conclusion:

Overall this project shows many usage of Kubernetes, Docker, Ansible, and AWS. Kubernetes was used to create a highly available cluster which would autoscale based on the usage load and limit the connection from other pods and deployments to the my-sql pod/deployment. Docker was the main container provisioner for the Kubernetes pods and allows access into it to backup parts of the cluster like etcd to restore a problematic master node/control plane to a previous state. Ansible was used to provision EC2 instances into an existing infrastructure in AWS to create the initial nodes for the Kubernetes setup. AWS provides a load-balancer attached to an elastic-ip to create a singular entry point into the cluster and balance the network activity that comes to it. AWS also provides its own provisioning of a Kubernetes cluster (EKS) but making this cluster in EC2 allows you to create control the master/control plane to your needs.