**Kubernetes Application Deployment with AWS EKS and ECR**

Deploying applications to Kubernetes using AWS EKS and AWS ECR images.



Photo by [Clément H](https://unsplash.com/@clemhlrdt?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText) on [Unsplash](https://towardsdatascience.com/s/photos/computer?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText" \t "_blank)

**Kubernetes**

Kubernetes is a container orchestration platform that is created by Google in 2014. It is an open-source platform where currently many organizations widely use for container deployment and management. Although there are other container orchestration tools are available in the community like Docker Swarm, Kubernetes remains in the top for container orchestration due to its features and flexible usability.

**AWS EKS**

Amazon Elastic Kubernetes Service is a service provided for Kubernetes on AWS infrastructure. Like any other service offered by AWS, Kubernetes resources will be fully managed by AWS themselves, which gives less overload for developers on maintaining them. AWS also make sure that these resources are highly available and reliable every time.

**AWS ECR**

Amazon Elastic Container Registry is a fully managed Docker registry provided by AWS. Just like the popular docker registry Dockerhub, ECR also supports private and public repositories which are very secure. We can either push or pull images to ECR using AWS CLI.



In this article, we are going to explore how we can deploy Kubernetes applications using AWS EKS and ECR services. Before going into complex details about how we are going to implement our Kubernetes solution below is the summary of tasks that we will be performing.

* Create a simple web application using Node.js
* Create a docker image of the web application
* Push our docker image to AWS ECR
* Create a VPC with public and private subnets for our EKS Cluster
* Create a Kubernetes Cluster
* Create Kubernetes workers(public and private workers)
* Deploy our web application on Kubernetes

Now I hope you have at least a little bit of an idea about what we are going to cover in this article. Before we start implementing we need to have the following prerequisites available in our development machines.

**Prerequisites**

* **Kubectl** — communication tool we will use to communicate between our Kubernetes cluster and our machine. Installation instructions available on <https://kubernetes.io/docs/tasks/tools/install-kubectl/>
* **AWS CLI**— AWS tool which we will use to issue commands related to AWS configurations. To install follow <https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html>
* **Aws iam authenticator** — Give permission with IAM roles to access our Kubernetes cluster. For installation <https://docs.aws.amazon.com/eks/latest/userguide/install-aws-iam-authenticator.html>
* **eksctl** — Official CLI tool for AWS EKS. For installation h[ttps://github.com/weaveworks/eksctl](https://github.com/weaveworks/eksctl)

After fulfilling our prerequisites first task will be to creating a simple server. I am using Node.js with express to create a very simple web application that will be listening on port 3000.

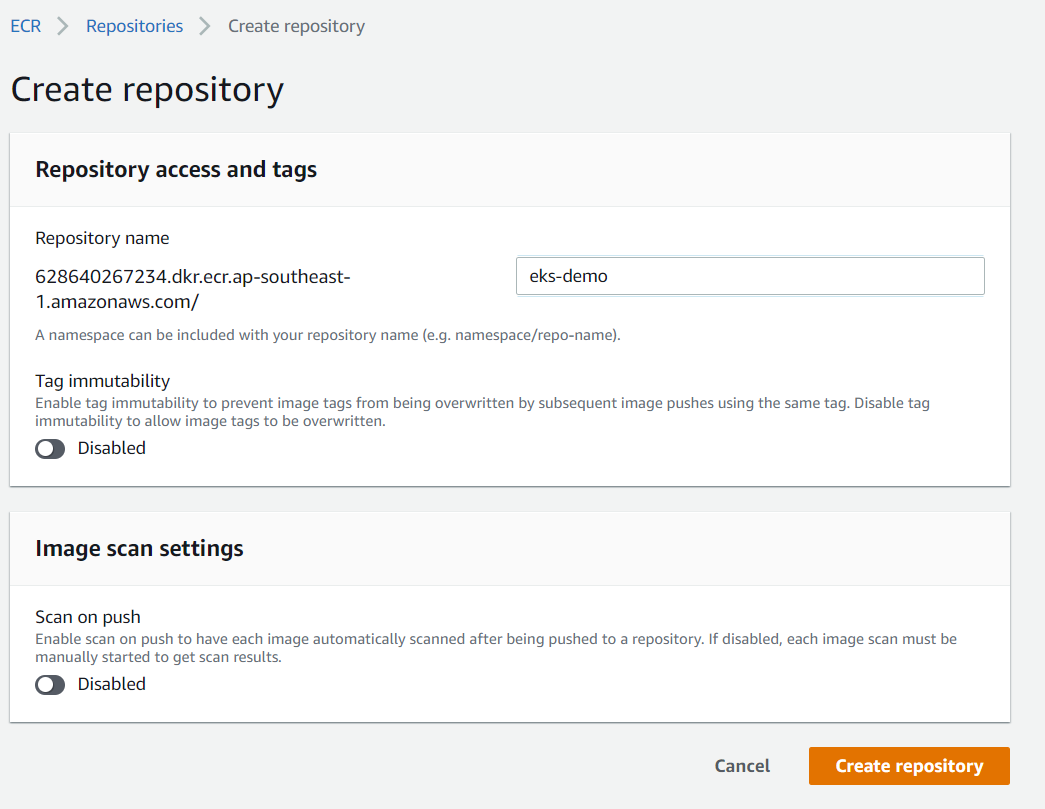
Next, let’s dockerize our web application. For that create a Dockerfile and issue docker build command.

docker build -t webapp .

Now if you issue docker images we will see our webapp image. The next task is to push our image to AWS ECR.

**Creating an ECR repository**

Before we can push the image we need to create a repository on ECR. For that go to the ECR dashboard and click **Create Repository.**



Now we have a repository to push our image. But before that, we need to authenticate our AWS CLI to push images to our repository. For that issue below command.

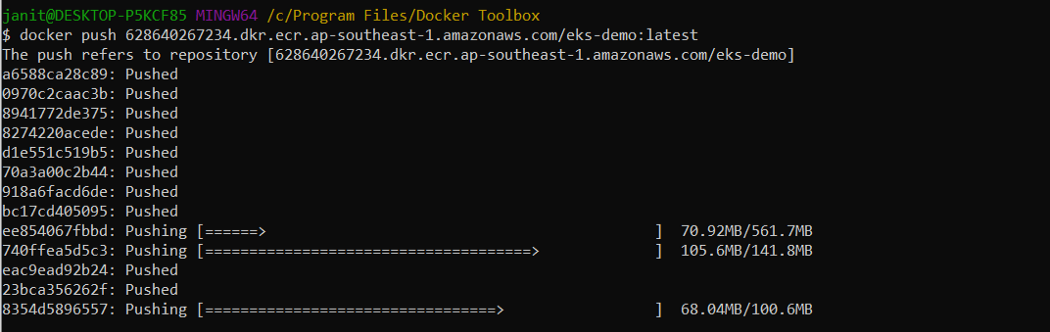
(Get-ECRLoginCommand).Password | docker login --username AWS --password-stdin 628640267234.dkr.ecr.ap-southeast-1.amazonaws.com

After that tag the image with our repository name. Here as the version, you can give any version, but in this instance, I am going to make the version as latest.

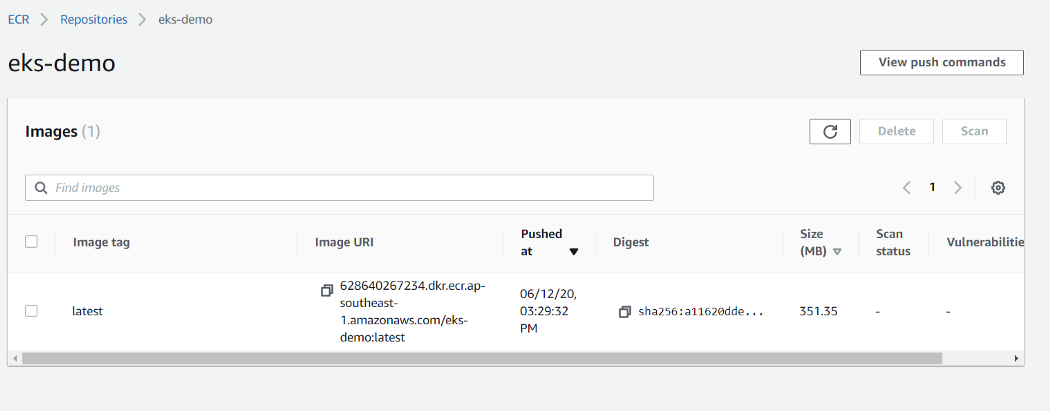
docker tag webapp:latest 628640267234.dkr.ecr.ap-southeast-1.amazonaws.com/eks-demo:latest

Now the last step, push our image to the ECR repository.

docker push 628640267234.dkr.ecr.ap-southeast-1.amazonaws.com/eks-demo:latest



If you get any permission issues make sure your AWS CLI role has permission [AmazonEC2ContainerRegistryFullAccess](https://console.aws.amazon.com/iam/home?region=ap-southeast-1#/policies/arn%3Aaws%3Aiam%3A%3Aaws%3Apolicy%2FAmazonEC2ContainerRegistryFullAccess).



Now go to our repository and the image we pushed should be available there.

**Creating AWS EKS Cluster**

The next step would be to create our EKS cluster. When we create our cluster, we need to specify the VPC subnets for our cluster to use. Amazon EKS requires subnets in at least two Availability Zones. When creating VPC we have two options.

* Create public and private subnets
* Create only public subnets

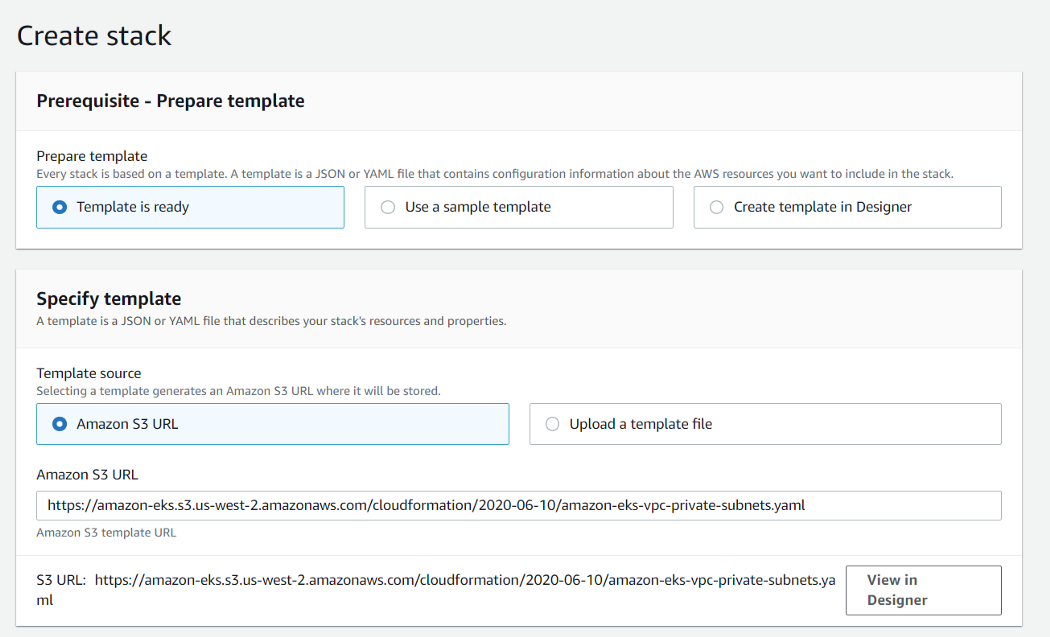
In this article, we are going to create a combination of public and private subnets. In that case, our web application can be externally accessed by using a public subnet, also if we need to deploy something like database then we can make them private which will be only accessible by our web application and any other application within the VPC.

**Creating VPC**

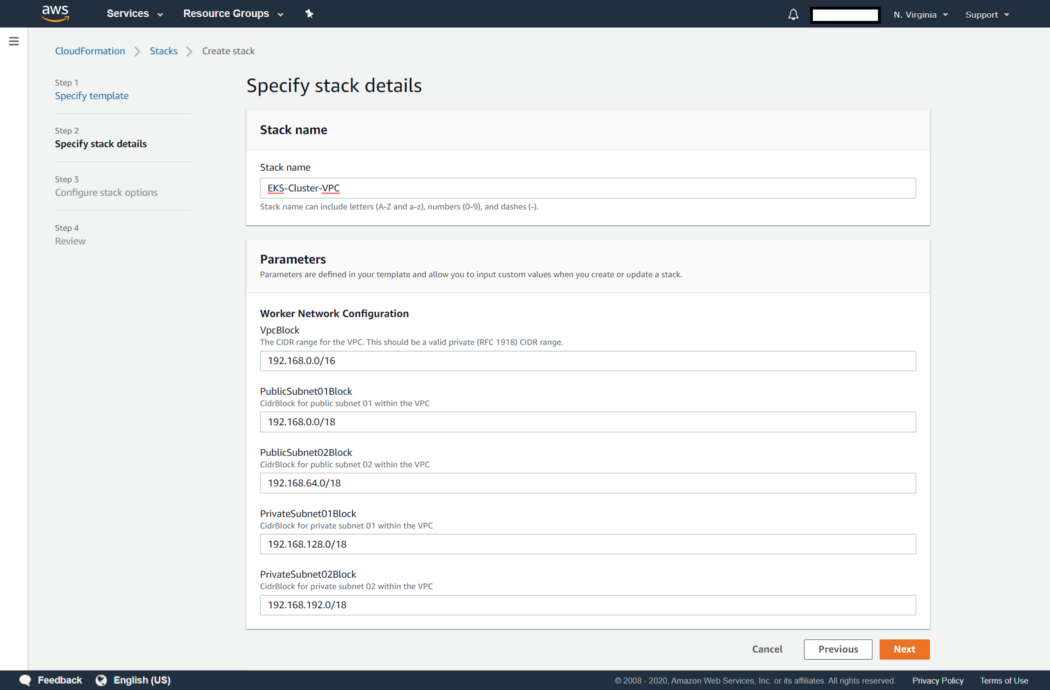
VPC for our cluster can be created manually if we want. But let’s create our VPC using AWS Cloudformation because AWS already has a template for creating a public and private subnet VPC.

* Go to CloudFormation dashboard and select Create Stack
* Here select template source as Amazon S3 URL and provide the following template already created by AWS.

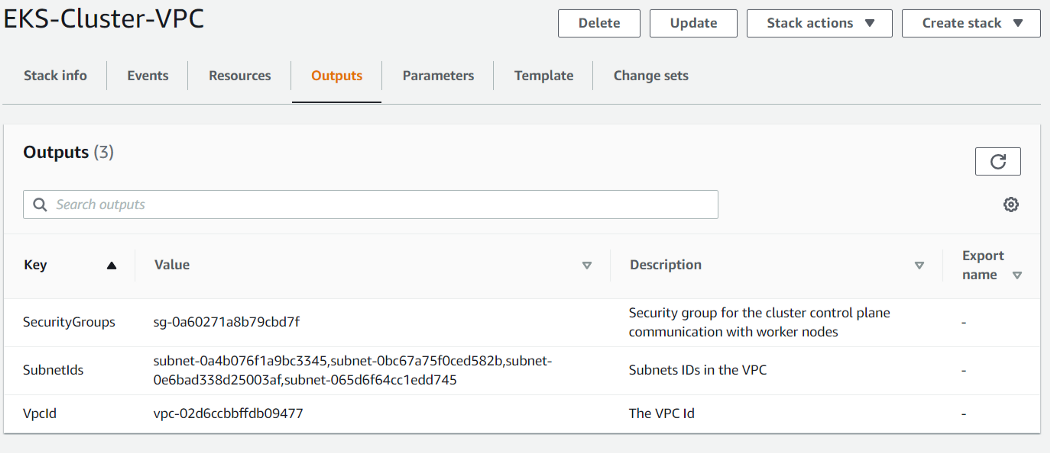
<https://amazon-eks.s3.us-west-2.amazonaws.com/cloudformation/2020-06-10/amazon-eks-vpc-private-subnets.yaml>



* VPC will have CIDR addresses of 192.168.0.0/16
* Create two public subnets with CIDR blocks 192.168.0.0/18 and 192.168.64.0/18
* Create two private subnets with CIDR blocks 192.168.128.0/18 and 192.168.192.0/18



In the end, select Create and wait until the stack is created. At the end of the stack creation, it will give 3 outputs.



* SecurityGroups — this is the security group created for our VPC.
* SubnetIds — Ids of the 4 subnets we have created. A
* VpcId — Id of the VPC created.

**Create EKS Cluster with Workers**

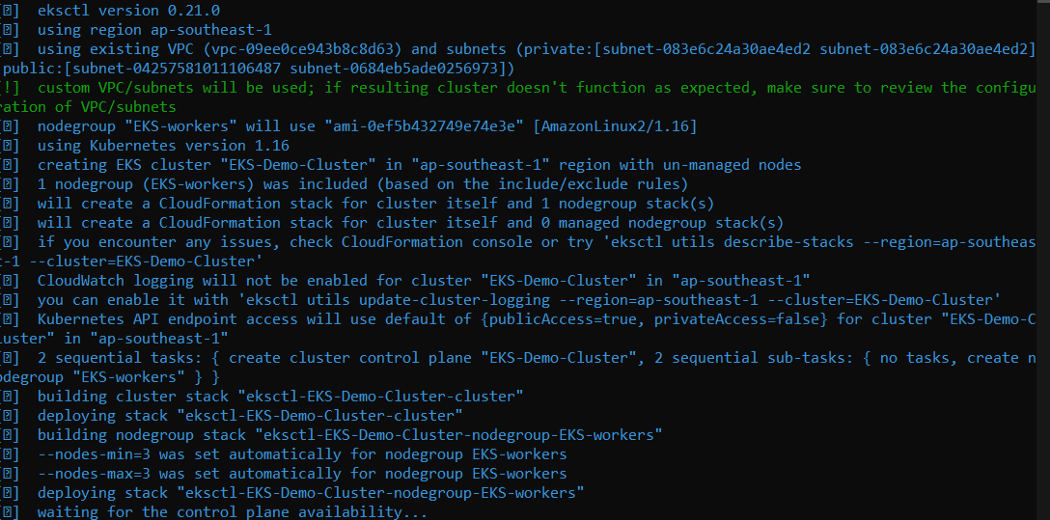
We can create clusters easily by giving **eksctl create** cluster command. But let’s create a YAML file with additional configurations below.

In the above cluster.yaml file, we define the following configurations for our cluster.

* Cluster name will be **EKS-Demo-Cluster**
* In vpc section, we provide the already created VPC earlier
* In node group, we create 3 workers with t2.meduim instances. Out of 3 workers 2 will be created as public workers while one will be private.

Now issue below command to create our cluster on EKS.

eksctl create cluster -f cluster.yaml **--kubeconfig**=C:\Users\{user}\.kube\config



After that eksctl will start creating our cluster according to our YAML file. Creating the cluster and nodes will take several minutes. At the get issue following command to check whether our cluster is deployed.

kubectl get svc

https://miro.medium.com/max/990/1*eP_hFBeY5W5JXgdoRQFWSQ.png

Sometimes you may get the following error when you issue the kubectl command. This might mean that in our kubectl config file, credentials and users required to access our cluster is not defined.

error: no configuration has been provided, try setting KUBERNETES\_MASTER environment variable

To write these configuration details to config file issue following command.

aws eks --region {region} update-kubeconfig --name EKS-Demo-Cluster

**Create Deploy Manifest**

Now let’s start to deploy our application on the created Kubernetes cluster. First, to deploy our application on pods, we need to create a deployment. Below is the deployment manifest that will be used for deployment.

* Our application label will be **app:web**
* In spec:template:spec:containers set image for the AWS ECR image we pushed
* Number of replicas for the application is 2

Issue following command to create our deployment.

kubectl apply -f deployment.yaml

To check whether our deployment created, issue below command.

kubectl get deployments

https://miro.medium.com/max/701/1*MgBqyD-O_MeUzfEfeitHZw.png

https://miro.medium.com/max/696/1*p8t5K4NRu3dnRqN-E5YrHw.png

Now we can see that our deployment is created and is running on two pods.

**Create service manifest**

Now to access our application, we need to create a service. Our service type will be **Nodeport** because we need our application to access from outside.

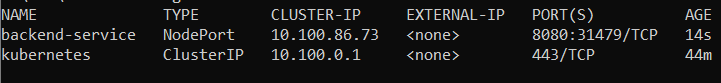
* service type is **Nodeport**
* targetPort is **3000** since that is our container exposed port
* the selector will be **app:web** since that is the label we defined in our deployment

To create our service issue below command

kubectl apply -f service.yaml

To check whether our service created, issue below command.

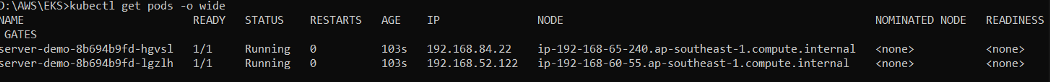
kubectl get services



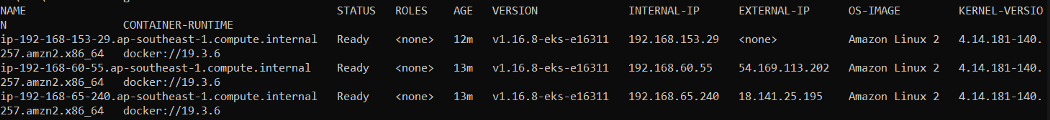
Now let’s try to access our web application externally. From the service, we know that our application is listening on port 31479. Next, we need to acquire the public IP address of our application nodes. Let’s first try to identify where are the pods of our application are running.

kubectl get pods -o wide

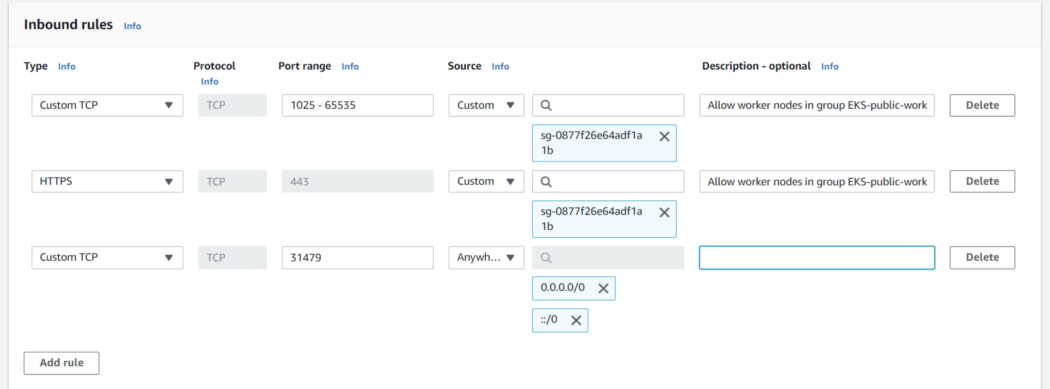
From that, we can identify the nodes of the pods that our application is running. To get the external IP addresses of those nodes, issue the get nodes command.



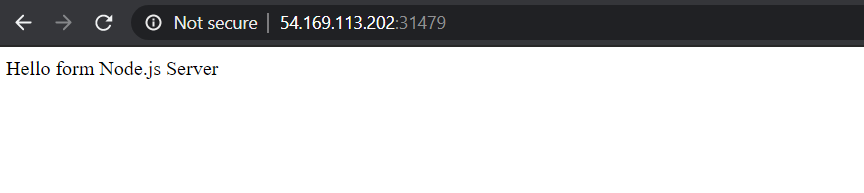
kubectl get nodes -o wide



In the above nodes list, we can see two of our nodes have external IPs while one does not have because we configured it as a private worker node. Now we have our IP addresses as well as the port it is listening. The next task will be to add this port in the node’s security group to allow traffic in. For that identify security group created for nodes and add an inbound rule to allow traffic in port 31479.



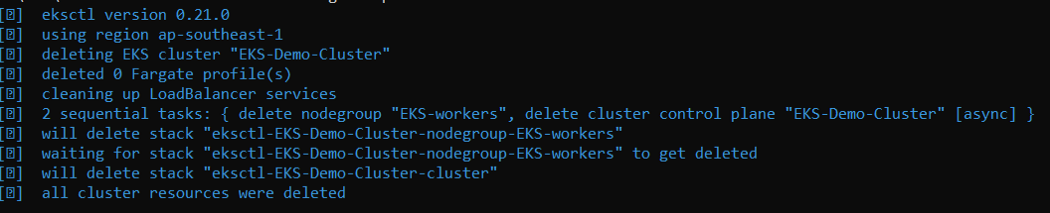
After that, we can get a public node IP address and call to it with port 31479.



We can also do the same with other IP address and the result should be the same.

The next task would be to deploy a database into our Kubernetes cluster. But I will leave that task for you to try out. After that make sure to delete the cluster by giving below command to avoid charges on EC2 instances we created.

eksctl delete cluster --region=ap-southeast-1 --name=EKS-Demo-Cluster



That is it for how to create and deploy applications to Kubernetes using AWS EKS and ECR. There are so many other concepts inside Kubernetes as well as on EKS that we can learn. So make sure to learn more and more until you feel the confidence to deploy and manage applications. Thank you.