COSC 6376 - Cloud Computing

Team - Incognito

Production Ready Application Deployment Using Kubernetes

Viswanath Puppala
Computer Science, NSM
University of Houston systems
Houston, Texas
vpuppal2@cougarnet.uh.edu,2084700

Divyasri Padala
Computer Science, NSM
University of Houston systems
Houston, Texas
dpadala@cougarnet.uh.edu, 1991418

Sai Balasubrahmanya Dheeraj Gandikota
Computer Science, NSM
University of Houston systems
Houston, Texas
sgandiko@cougarnet.uh.edu, 2084645

ABSTRACT - Deploy a Dockerized applications onto Kubernetes cluster which is created by Terraform using Infrastructure-as-code (IAC) principles to enhance the scalability and to experience how applications are deployed to production using automation.

INTRODUCTION

A typical cloud deployment often entails manually installing code onto a virtual machine (VM), hosting services like a database and filesystems or on-premises and hosting the server that responds to application requests. However, in today's deployments, this is simplified using sophisticated frameworks such as Terraform, which contains provisions such as cloud infrastructure, container orchestration frameworks like Kubernetes, Docker, etc., which are used to improve capabilities like auto-scaling, load balancing, etc., The intent of the project is to deploy applications that auto scale based on user demand using Terraform for provisioning the infrastructure and EKS for container orchestration.

IMPROVEMENTS SUGGESTED

- When you dockerize your application. Also publish it in docker registry. It would do a demo where you can just docker pull it.
- Once you are done put everything in terraform. In your demo, deploy it, and make it do all the deployment from scratch before you demo. That truly will show the power of these automated deployments.

METHODOLOGIES

In this project, we will be hosting an application into a Kubernetes cluster for container orchestration and auto-scaling using the Kubernetes manifest files for deploying application into the cluster. We will be using Infrastructure-As-code (IaC) principle to spin up the infrastructure using Terraform and then we will create a Terraform project that will have the infrastructure code for EKS cluster, S3 bucket, VPC and subnets for the EKS cluster to consume. The application we have used is a face mask recognition application, which is developed using Python and Neural Networks and a data set obtained from Kaggle [1]. It will be having a Neural Network machine learning model that will be trained to recognize the images whether the person wears mask on the face or not. The application has a web layer which accepts the incoming images for prediction and a backend layer which contains the trained machine learning model. This application will do the dockerizing and the docker image will be stored into the ECR. The Kubernetes manifest files will consume this ECR image for deploying the application into the cluster. We have used S3 to store the incoming images into the cloud and CloudWatch for handling the logging. Since the application will be hosted on Kubernetes, the scaling of the application will be handled by the control plane of the cluster based on the incoming requests and scaling policy that which will be implemented on the cluster.

IAC using Terraform

. We have used terraform EKS blueprints module from AWS, which is a plug and play module for terraform to create EKS clusters with ease. This module will take care of creating all the resources like Security Groups, VPC, NAT gateways, routing table, the EKS cluster itself and the required node groups all together instead of us defining each resource required for a working EKS cluster.

```
module "eks_blueprints" {
  source = "github.com/aws-ia/terraform-aws-eks-blueprints?ref=v4.15.0"

  cluster_name    = local.cluster_name
  cluster_version = "1.23"

  vpc_id             = module.vpc.vpc_id
  private_subnet_ids = module.vpc.private_subnets

  managed_node_groups = {
    mg_5 = {
      node_group_name = "managed-ondemand"
      instance_types  = ["t3.medium"]
      min_size        = 1
      max_size        = 1
      desired_size    = 1
      subnet_ids      = module.vpc.private_subnets
    }
  }

  tags = local.tags
}
```

Figure (1): Terraform EKS module code

We created a folder named Deployments and under it there are 4 terraform files. This is an example code taken from direct AWS EKS blueprints module. The main.tf file has the module to create the cluster as above.

We simply need to run the following commands to **create the cluster** in an AWS environment:

```
terraform plan
terraform apply
```

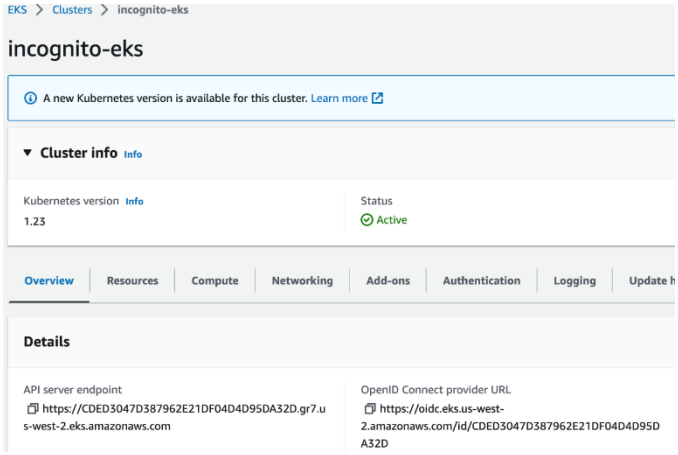


Figure (2): Image of EKS cluster after Terraform apply

Docker Image

We have chosen an existing machine application from the net which can detect if the person is wearing a mask of not. Over this code we have created a flask application for training and detecting the unseen data.

In this Flask application we have created two APIs names *"/train" and "/detect/<image_name>"*. But by default, when the flask server is started, the training of the application is also done, and the model artifacts are stored in the same directory. But when the detect API is called, which takes image name as a path variable, we read the image from a predefined S3 bucket which we have created to store the images to be predicted. In this API we have also hardcoded AWS Access and Secret key for the account and authenticated boto3 module to read the files from S3 bucket. This Image read from S3 is sent to the trained model for prediction and the result of the image will be shown in the browser as text.

This whole application which includes the machine learning code and flask application has been dockerized using a Dockerfile.

9 lines (9 sloc) | 223 Bytes

```
1  # syntax=docker/dockerfile:1
2  FROM python:3.8-slim-buster
3  WORKDIR /flask
4  COPY . .
5  RUN apt-get update
6  RUN apt-get install ffmpeg libsm6 libxext6  -y
7  RUN pip3 install -r requirements.txt
8  EXPOSE 3000
9  CMD ["python3", "main.py"]
```

Figure (3): Dockerfile of the application

We have created the docker image by running the following commands

```
docker build -t incognito.

aws ecr get-login-password --region us-west-2 | docker login --username AWS --password-stdin 237450111201.dkr.ecr.region.amazonaws.com

docker push 237450111201.dkr.ecr.region.amazonaws.com/incognito: latest
```

Once this docker image is pushed to ECR we can use that Image from cloud to deploy applications in to EKS cluster.

Kubernetes Deployment:

The EKS cluster created using Terraform, we can login into the cluster with the following command.

```
aws eks update-config --name incognito-eks --region us-west-2

#check if kube config is updated or not
kubectl create namespace flask
```

We have created a new namespace within the cluster named *"flask"*, this namespace will be used to deploy the application manifest we have created.

We have created a Kubernetes manifest file under Kubernetes folder, which has a yaml block for the deployment, that has the instructions to consume the docker image pushed in the ECR in the previous step and the port number for the application that needs to be exposed outside.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flaskapi-deployment
  labels:
    app: flaskapi
spec:
  replicas: 1
  selector:
    matchLabels:
      app: flaskapi
  template:
    metadata:
      labels:
        app: flaskapi
    spec:
      containers:
        - name: flaskapi
          image: 237450111201.dkr.ecr.us-west-2.amazonaws.com/incognito:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 3000
```

Figure (4): Kubernetes manifest for Deployment

The related service Yaml is also placed in the same file which has load balancer attached to the service so that the end points can be exposed to the internet and can be accessed by the user.

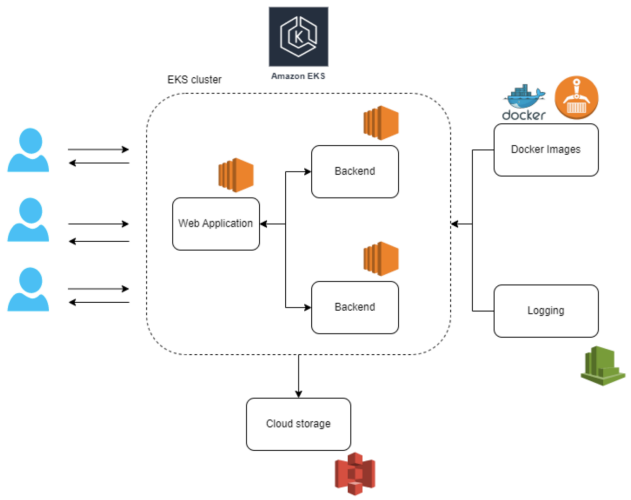
```yaml
apiVersion: v1
kind: Service
metadata:
  name: flask-service
spec:
  ports:
  - port: 3000
    protocol: TCP
    targetPort: 3000
  selector:
    app: flaskapi
  type: LoadBalancer
```

Figure (5): Kubernetes manifest for service and load balancer

We can apply this manifest file on the EKS cluster and into the *"flask"* namespace with the below command:

```
kubectl apply -f flaskservice.yml -n flask
```

ARCHITECTURE



RESULTS

Flask Services:

```
vidhyadharmusapeta@Vidhyadhars-MacBook-Pro kubernetes % k get all -n flask
NAME                                    READY   STATUS    RESTARTS   AGE
pod/flaskapi-deployment-74bbd44c98-2tmd6   1/1     Running   0          125m

NAME                    TYPE           CLUSTER-IP       EXTERNAL-IP
service/flask-service   LoadBalancer   172.20.171.187   ae81d41c7c3224dc3be83b3af210e795-1136611

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/flaskapi-deployment   1/1     1            1           125m

NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/flaskapi-deployment-74bbd44c98   1         1         1       125m
```

Figure (7): List of all resources in EKS cluster after deployment

Scaling Deployments:

Define fewer common abbreviations and acronyms the first

```
vidhyadharmusapeta@Vidhyadhars-MacBook-Pro kubernetes % kubectl scale deployment flaskapi-deployment -n flask  --replicas=2

deployment.apps/flaskapi-deployment scaled
vidhyadharmusapeta@Vidhyadhars-MacBook-Pro kubernetes % k get pods -n flask
NAME                                READY   STATUS    RESTARTS   AGE
flaskapi-deployment-74bbd44c98-2tmd6   1/1     Running   0          130m
flaskapi-deployment-74bbd44c98-6brbs   1/1     Running   0          15s
vidhyadharmusapeta@Vidhyadhars-MacBook-Pro kubernetes % kubectl scale deployment flaskapi-deployment -n flask  --replicas=5
deployment.apps/flaskapi-deployment scaled
vidhyadharmusapeta@Vidhyadhars-MacBook-Pro kubernetes % k get pods -n flask
NAME                                READY   STATUS    RESTARTS   AGE
flaskapi-deployment-74bbd44c98-2tmd6   1/1     Running   0          130m
flaskapi-deployment-74bbd44c98-64ll6   1/1     Running   0          9s
flaskapi-deployment-74bbd44c98-6brbs   1/1     Running   0          36s
flaskapi-deployment-74bbd44c98-bkkhk   1/1     Running   0          9s
flaskapi-deployment-74bbd44c98-wn9dd   1/1     Running   0          9s
vidhyadharmusapeta@Vidhyadhars-MacBook-Pro kubernetes %
```

Figure (8): Scaling the deployment pod with increase in replicas

API Results:

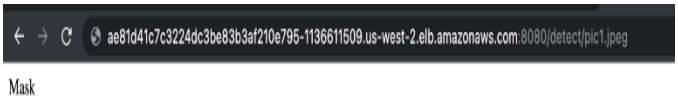


Figure (9): Predicting an Image using the external API from Load Balancer

FUTURE SCOPE

- In general, Dev-ops principles are implemented in production applications to automate the releases and deployments
- ***Automation using CI/CD pipelines*** - Instead of Manual implementation by pushing the code using Git, we propose to Integrate the architecture to a CI/CD pipeline with tools such as GitLab, Jenkins or code pipelines (AWS) to fully automate with GITOPS principles.
- ***Enabling live cam prediction*** – In this project, we are reading the images to the application from the S3 bucket which can be improvised by enabling live cam prediction

REFERENCES

1. https://www.clickittech.com/devops/terraform-kubernetes-deployment/
2. https://github.com/aws-ia/terraform-aws-eks-blueprints
3. https://www.kaggle.com/datasets/omkargurav/face-mask-dataset
4. https://towardsdatascience.com/build-and-run-a-docker-container-for-your-machine-learning-model-60209c2d7a7f
5. https://docs.aws.amazon.com/AmazonECR/latest/userguide/docker-push-ecr-image.html