Cloud Computing and Big Data Systems - Fall 2023 Assignment 2 Instructions (rough documentation)

Objective:

In this assignment, you will learn how to deploy a web application on Kubernetes using Docker containers. You will start by containerizing the application using Docker, create a persistence volume, and push the image to Docker Hub. Then, you will deploy the container on a local Kubernetes cluster using Minikube and then deploy it on AWS EKS. You will also explore various Kubernetes features such as a replication controller, health monitoring, rolling updates, and alerting.

Prerequisites:

Before you start this assignment, you should have the following:

- A basic understanding of Docker and Kubernetes
- Docker installed on your computer
- Minikube and Kubectl installed on your computer
- An AWS account
- DockerHub account

Part 1: Creating an Application:

- For this assignment, you will create a simple **To-Do** web application using Flask and MongoDB. You can access the source code for it here.

python -m venv venv
source venv/bin/activate
Create virtual env and pip install -r requirements.txt and dependencies ,
Flask==2.1.3
pymongo==4.2.0
Werkzeug==2.2.2

Part 2: Containerizing the Application on Docker:

- Install Docker on your local machine by downloading and installing it from the official website.

- Use two containers for your application, one for the flask app and one for persisting the data in MongoDB.

- Write a Dockerfile for the flask application, which will contain instructions to build the Docker image. The Dockerfile will set the working directory, copy the application code into the container, and specify the command to run the application.

Install docker, create dockerfile and then docker build -t flask-app-trial3 image . we will use this image in container. Mongodb has an official image that we are going to reuse. Docker file has instructions such as command to run flask app, references documentation such as which port is exposed and volume.
docker login
Tag the image with 'docker tag your-flask-app-image-name dockerhub-username/your-repository-name:tag'
Push to docker hub and then run via docker-compose up(this works with the docker compose file we have created) 'docker push dockerhub-username/your-repository-name:tag'
At this point, to avoid port errors, map to 8080 (especially since mac has some processes always running at port 5000 due to sharing/airdrop). Python app.py file should have app.run(host='0.0.0.0', port=PORT)

- Build the Docker image using the Docker CLI (command-line interface). This will use the instructions in the Dockerfile to build the Docker image.

- To test the application locally, write a docker-compose file that defines the services for the flask app and MongoDB container. The docker-compose file should specify the images to use, the ports to expose, and a volume to mount to store the database information..

- Push the Docker image to Docker Hub to make it available for deployment to Kubernetes. This will require creating a Docker Hub account and using the Docker CLI to push the image to the Docker Hub registry.

Part 3: Deploying the Application on Minikube:

- Install Minikube on your local machine by following the official documentation.

- Start Minikube using the command-line interface.

- Create two pods: one for the flask app and one for the MongoDB to store data.

- Create a Kubernetes deployment for the application by specifying the Docker image from Docker Hub, the number of replicas, and the container port.

- Expose the deployment using a Kubernetes service to make the application accessible from outside the cluster. The service should expose the container port and provide load balancing to distribute traffic to the replicas.

- Test the application to ensure that it is running on Minikube by accessing it using the service URL.

minikube start
Yaml file for each pod,i.e., flask and mongodb.

kubectl apply -f flask-app-pod.yaml
kubectl apply -f mongodb-pod.yaml
kubectl apply -f flask-app-deployment.yaml
Create and start flask-service
kubectl apply -f flask-app-service.yaml

minikube service flask-app-service
kubectl get svc flask-app-service
Open external ip that is given

Part 4: Deploying the Application on AWS EKS:

- Create an AWS EKS cluster using the AWS Management Console or the AWS CLI. This will provide the Kubernetes infrastructure for running the application.

- Configure the Kubernetes CLI (kubectl) to connect to the EKS cluster. This will allow you to interact with the Kubernetes resources in the EKS cluster using kubectl commands.

- Create a Kubernetes deployment for the application in the EKS cluster. This deployment will specify the Docker image to use, the number of replicas of

the application to run, and the port number that the application listens on.

- Expose the deployment using a Kubernetes service. This service will provide load balancing for the application and expose the container port so that it can be accessed from outside the EKS cluster.

- Test the application to ensure that it is running on AWS EKS. This will verify that the deployment and service were configured correctly and that the application is accessible from outside the EKS cluster.

```
aws eks create-cluster --name your-cluster-name --role-arn your-eks-service-role-arn --resources-vpc-config
subnetIds=your-subnet-ids,securityGroupIds=your-security-group-ids

eksctl create cluster
--name eks-cluster-2
—version 1.28 \
—-region us-east-1 \
—-nodegroup-name assignment-nodes-1 \
—node-type t2.micro \
--nodes 2

aws eks update-kubeconfig --name your-cluster-name
aws eks describe-cluster --name Assignment2-Cluster
kubectl expose deployment flask-app-deployment --type=LoadBalancer --name=flask-app-service
```

Part 5: Replication controller feature:

- Create a replication controller to ensure that a specified number of replicas of the application are always running.

- Define the desired state of the application by setting the number of replicas in the replication controller configuration file.

- Use kubectl to create the replication controller and verify that the specified number of replicas are created and running.

- Test the replication controller by intentionally deleting one of the pods and verifying that a new pod is automatically created by the replication controller to maintain the desired number of replicas.

- Update the desired number of replicas in the replication controller configuration file and use kubectl to apply the changes to the running replication controller. Verify that the replication controller scales up or down the number of replicas as specified in the configuration file.

Create yaml file and apply
kubectl apply -f flask-app-replication-controller.yaml
kubectl get replicationcontroller

Part 6: Rolling update strategy:

- Configure the Kubernetes deployment to use a rolling update strategy.
- Set the update strategy to rolling Update and specify the maximum number of pods that can be unavailable during the update.
- Update the Docker image for the deployment to a new version.
- Trigger the rolling update by updating the deployment with the new Docker image version.
- Monitor the rolling update progress using the Kubernetes CLI (kubectl) or the Kubernetes Dashboard.
- Test the updated application to ensure that it is running with the new

Docker image version.

Update flask-app-deployment yaml file's strategy with rollingupdate
kubectl rollout status deployment flask-app-deployment

Part 7: Health monitoring:

- Configure Kubernetes to monitor the health of the application by setting up probes for the pods. There are two types of probes: liveness Probe and readiness Probe. A liveness Probe checks if the pod is alive and healthy, while a readiness Probe checks if the pod is ready to receive traffic. Specify the probe's configuration by setting parameters such as the probe type, the probe endpoint, and the probe interval.

- Configure Kubernetes to take action if a probe fails, such as by restarting the pod or marking it as unhealthy.

- Monitor the health of the pods using the Kubernetes CLI (kubectl) or the Kubernetes Dashboard.

- Test the health monitoring system by intentionally causing a failure, such as by adding a deliberate error to the application code, and verifying that Kubernetes takes the appropriate action.

Adding probes in deployment.yaml. We also add routes in app.py

Step 8: Alerting (Extra Credit 20 Points):

- Set up an alerting service such as Prometheus to receive alerts from Kubernetes.

- Configure Prometheus to generate alerts when certain conditions are met, such as the number of failing probes exceeding a certain threshold.

- Set up a notification service such as Slack to receive alerts from Prometheus.

- Test the alerting system by intentionally causing a failure, such as by deleting a pod, and verifying that an alert is generated and sent to the notification service.

brew install helm
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update

helm install prometheus prometheus-community/prometheus

helm install alertmanager prometheus-community/alertmanager
Set alert system and prometheus alerts yaml.
helm upgrade prometheus prometheus-community/prometheus -f prometheus-rules.yaml
helm install alertmanager prometheus-community/alertmanager -f values.yaml

Prometheus yaml contains api url of slack notification

Submission Requirements

- Dockerfile for the application - this is the file that defines the Docker image for the application.

- Kubernetes configuration files for the application deployment and service - these files specify how the application should be deployed and made available on Kubernetes.

- Document explaining the steps followed to complete the assignment - this should include screenshots of the application running on Minikube and AWS EKS