**Cloud Computing and Big Data Systems - Assignment 2 Implementation Documentation**

**Part 2: Containerizing the Application using Docker**

1. Building a docker image for the flask app using Dockerfile: Use the command

$ docker buildx build --platform linux/arm64,linux/amd64 -t tejeshunyu/flask-todo-app:v1 .

to create the Docker image based on the Python 3.9-slim base image

A screenshot of a computer program

AI-generated content may be incorrect.

1. Test the image using a docker-compose file: Use the command below

$ docker compose up -d

to create docker containers for the flask app and mongodb

A close-up of a computer code

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

1. Push the flask app docker image to docker hub: Use the command below

$ docker push tejeshunyu/flask-todo-app:v1

to push the locally built docker image to docker hub

A screenshot of a computer program

AI-generated content may be incorrect.

**Part 3: Deploying the Application on Minikube**

* Start minikube by using the command:

$ minikube start

A screenshot of a computer program

AI-generated content may be incorrect.

* Create pods for the flask app and the mongodb using the Kubernetes deployment files and expose the deployment for the flask app using a service.

$ kubectl apply -f kmongo.yaml

$ kubectl apply -f flask-deployment.yaml

$ kubectl apply -f flask-service.yaml

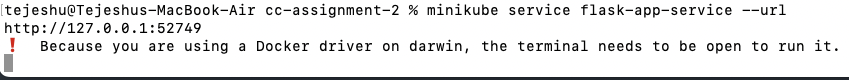
A screenshot of a computer

AI-generated content may be incorrect.

A close up of a computer code

AI-generated content may be incorrect.

By using the below command you can get the service external url for the flask app

$ minikube service flask-app-service --url 

A screenshot of a computer

AI-generated content may be incorrect.

**Part 4: Deploying the Application on AWS EKS**

After creating your AWS EKS cluster, you first need to setup AWS environment variables in your terminal for it to communicate with AWS EKS cluster

You need to setup AWS\_ACCESS\_KEY\_ID, AWS\_SECRET\_ACCESS\_KEY, AWS\_SESSION\_TOKEN

Then you can connect to your aws eks cluster using kubectl by using the below command:

$ aws eks update-kubeconfig --region region-code --name my-cluster



Then you can deploy your deployment on AWS EKS using the below commands:

$ kubectl apply -f kmongo.yaml

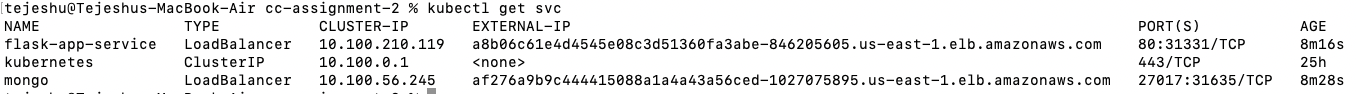
$ kubectl apply -f flask-deployment.yaml

$ kubectl apply -f flask-service.yaml

A black text on a white background

AI-generated content may be incorrect.

Then you can access your flask app using the external IP mentioned here:



A screenshot of a computer

AI-generated content may be incorrect.

**Part 5: Replication Controller Feature**

1. A replication controller was created using

$ kubectl apply -f flask-rc.yaml

which created 3 replicas as shown by

$ kubectl get rc

1. The replication controller's self-healing capability was tested by:

* Verifying that 3 Flask app pods were running
* Deliberately deleting one pod ($ kubectl delete pod flask-app-rc-dvf4c)
* Confirming that the replication controller automatically created a new replacement pod to maintain the desired count of 3

1. The ability to scale the application was tested by:

* Scaling up to 5 replicas using

$ kubectl scale rc flask-app-rc --replicas=5

* Verifying the successful scaling with

$ kubectl get rc

showing DESIRED=5, CURRENT=5

* Observing all 5 pods running with

$ kubectl get pods

1. Finally, scaling down was tested by:

* Reducing to 2 replicas using

$ kubectl scale rc flask-app-rc --replicas=2

* Confirming the scale-down with

$ kubectl get rc

Showing DESIRED=2, CURRENT=2

* Verifying only 2 Flask app pods remained running using

$ kubectl get pods

A screenshot of a computer program

AI-generated content may be incorrect.

1. We can also update the number of desired replicas by changing the replicas field in the yaml file and re applying the yaml file in Kubernetes

$ kubectl apply -f flask-rc.yaml

A screenshot of a computer program

AI-generated content may be incorrect.

**Part 6: Rolling Update Strategy**

1. **Preparing a new version**:

* Created a new version of the application image (v2) with

$ docker tag tejeshunyu/flask-todo-app:v1 tejeshunyu/flask-todo-app:v2

* Pushed the new image to Docker Hub with

$ docker push tejeshunyu/flask-todo-app:v2

A screenshot of a computer

AI-generated content may be incorrect.

2. **Performing the rolling update**:

* Deploying the v1 of the image for rolling update

$ kubectl apply -f flask-ru-deployment.yaml

* Updated the deployment to use the new image version with

$ kubectl set image deployments/flask-app-ru-deployment flask-app-ru=tejeshunyu/flask-todo-app:v2

* Monitored the rollout status with

$ kubectl rollout status deployment/flask-app-ru-deployment

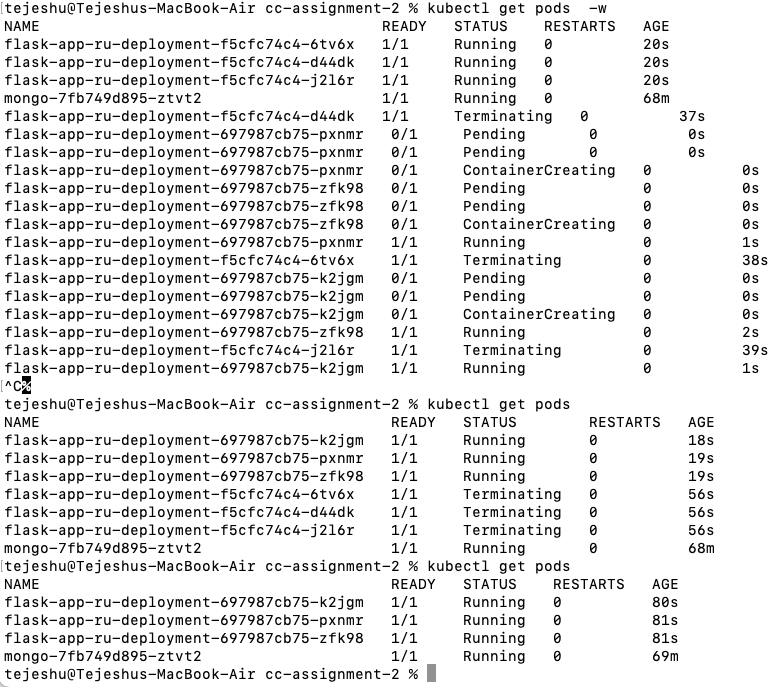
* Successfully completed the rolling update as indicated by ‘deployment "flask-app-ru-deployment" successfully rolled out’

3. **Verifying the update**:

* Confirmed all pods are running the new version with

$ kubectl get pods -w

$ kubectl get pods



* We can see the history of rollout for the deployment using

$ kubectl rollout history deployments/flask-app-ru-deployment

* We can then see the version of each revision using

$ kubectl rollout history deployments/flask-app-ru-deployment --revision=1

$ kubectl rollout history deployments/flask-app-ru-deployment --revision=2

* The output shows that the deployments have different versions v1 and v2 and the above showed that the update was done gracefully.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**Part 7: Health monitoring:**

The health monitoring has been configure for the yaml files flask-hm-deployment.yaml and flask-hm-service.yaml

For testing purposes, the endpoints for the liveness and readiness probes in app.py are given as:

* For testing the liveness probe, I have written the code for app.py as below:

start\_time = time.time()

@app.route("/healthz")

def healthz():

elapsed\_time = time.time() - start\_time

if 30 < elapsed\_time < 60:

return jsonify(error="Failure"), 500

else:

return "OK", 200

@app.route("/readyz")

def readyz():

# return "OK", 200

try:

client.admin.command('ping')

return "OK", 200

except ConnectionFailure:

return jsonify(error="MongoDB unavailable"), 503

which are such that after 30s the pods should restart, we can verify as below

A close up of a text

AI-generated content may be incorrect.

We can see that the pods are restarting due to the failure of the liveness probe, we can also see that in the logs for the pods

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Since I have set failure threshold to 3, it will restart after 3 retries

* For testing the readiness probe, I have written the code for app.py as below:

start\_time = time.time()

@app.route("/healthz")

def healthz():

# elapsed\_time = time.time() - start\_time

# if 30 < elapsed\_time < 60:

# return jsonify(error="Failure"), 500

# else:

return "OK", 200

@app.route("/readyz")

def readyz():

elapsed\_time = time.time() - start\_time

if 30 < elapsed\_time < 60:

return jsonify(error="Failure"), 500

else:

try:

client.admin.command('ping')

return "OK", 200

except ConnectionFailure:

return jsonify(error="MongoDB unavailable"), 503

We can see that the readiness probe is getting the expected error from the endpoint

A screenshot of a computer

AI-generated content may be incorrect.

Then we can see that the pods are getting marked as unhealthy

A computer screen with white text

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.