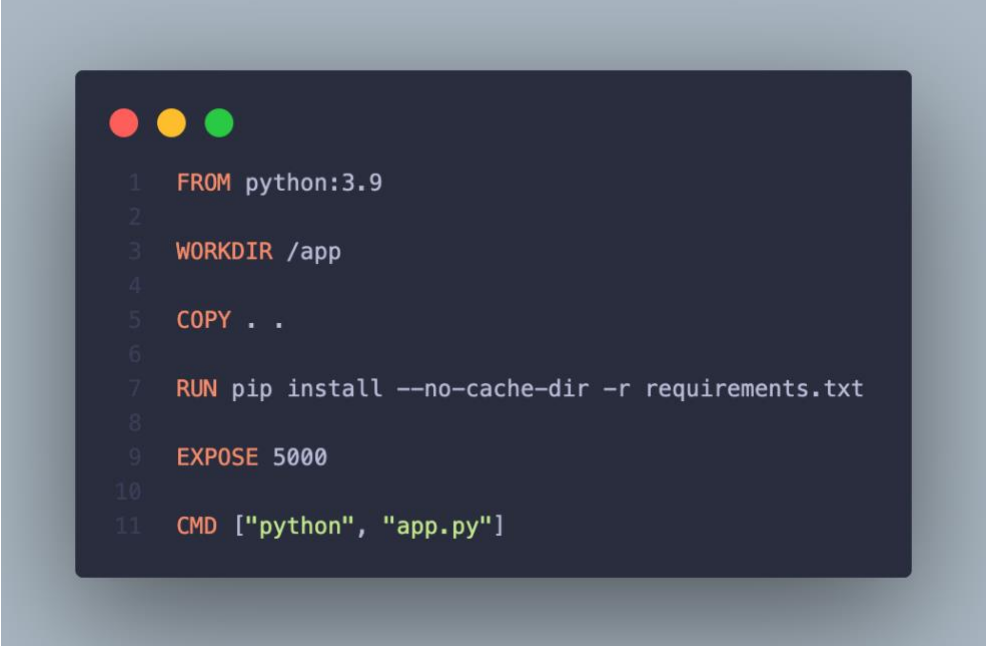


## Assignment 2

### 1. Containerizing the Application on Docker:

- a. Requisites: Docker, KubectL, and Minikube are already installed
- b. First, the source code for the flask web application is downloaded locally.  
The idea is to containerize this application and run it both locally using Minikube and then in AWS EKS.
- c. The first thing to do is write the Dockerfile which will have the instructions to make the Docker image (these is used to make containers)



```
1 FROM python:3.9
2
3 WORKDIR /app
4
5 COPY . .
6
7 RUN pip install --no-cache-dir -r requirements.txt
8
9 EXPOSE 5000
10
11 CMD ["python", "app.py"]
```

- i.
  - ii. The Dockerfile copies the files into the container, installs dependencies and runs the app when the container starts
- d. Then we wrote the docker compose file to run many containers at the same time.

```

1  services:
2    mongo:
3      image: mongo:latest
4      container_name: mongo_container
5      restart: always
6      ports:
7        - "27017:27017"
8      volumes:
9        - mongo_data:/data/db
10     environment:
11       MONGO_INITDB_ROOT_USERNAME: root
12       MONGO_INITDB_ROOT_PASSWORD: example
13     networks:
14       - app_network
15
16   flask-app:
17     build: .
18     container_name: flask_app_container
19     restart: always
20     ports:
21       - "5000:5000"
22     environment:
23       MONGO_HOST: mongo
24       MONGO_PORT: 27017
25       MONGO_INITDB_ROOT_USERNAME: root
26       MONGO_INITDB_ROOT_PASSWORD: example
27     depends_on:
28       - mongo
29     networks:
30       - app_network
31
32   volumes:
33     mongo_data:
34
35   networks:
36     app_network:
37

```

- i.
  - ii. The mongo service runs the mongo db container that persists the data in the mongo\_data volume. The flask service uses mongo and runs the flask app in the container.
- e. Next, we build the containers and start them. To do this we run the command:
- i. `docker-compose up --build`
    1. This takes the docker-compose.yaml and for the services it sees if it takes an image or has to build it. The mongo service takes the latest image while the flask service has **"build: ."**,

which means it builds the image if no available image is found.  
If there is no image then it uses the Dockerfile to build the image

2. First the mongo container starts and then after the docker image for the flask app is built, the flask app starts

- ii. SCREENSHOT FLASK MONGODB LOCALLY

- f. Finally, the flask app image is pushed on Docker Hub. To do this first we login using **docker login**. Then we tag the image created using **docker tag flask-app username/flask-app:v1** for Kubernetes to have the image available.

Then we can push it using **docker push username/flask-app:v1**

- i. SCREENSHOT DOCKERHUB REPO

2. Deploying the application on Minikube

- a. Then we used Minikube to run the app locally using Kubernetes (Minikube)
  - b. First we start Kubernetes locally using **minikube start** which starts a local Kubernetes cluster. Then we create the two pods: one for flask and one for mongo (to do this we use the deployment files for flask and mongo.
    - i. We create both the flask-deployment.yml and the mongo-deployment.yml that will create the pods



```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: mongo
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: mongo
10   template:
11     metadata:
12       labels:
13         app: mongo
14     spec:
15       containers:
16       - name: mongo
17         image: mongo:latest
18         ports:
19         - containerPort: 27017
20         env:
21         - name: MONGO_INITDB_ROOT_USERNAME
22           value: root
23         - name: MONGO_INITDB_ROOT_PASSWORD
24           value: example
25         volumeMounts:
26         - name: mongo-storage
27           mountPath: /data/db
28       volumes:
29       - name: mongo-storage
30         emptyDir: {}
31   ---
32   apiVersion: v1
33   kind: Service
34   metadata:
35     name: mongo-service
36   spec:
37     selector:
38       app: mongo
39     ports:
40     - protocol: TCP
41       port: 27017
42       targetPort: 27017
43

```

1. The mongo deployment file states the type of Kubernetes (deployment), the image to use that is taken from the Docker Hub, the ports to use inside the Kubernetes cluster, the volumes to use. The service creates a ClusterIP service so only pods inside the Kubernetes can access it.

```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: flask-app
5  spec:
6    replicas: 3
7    strategy:
8      type: RollingUpdate
9      rollingUpdate:
10        maxUnavailable: 1
11        maxSurge: 1
12    selector:
13      matchLabels:
14        app: flask-app
15    template:
16      metadata:
17        labels:
18          app: flask-app
19      spec:
20        containers:
21          - name: flask-app
22            image: navasanti19/flask-app:latest
23            ports:
24              - containerPort: 5000
25            env:
26              - name: MONGO_HOST
27                value: "mongo-service"
28              - name: MONGO_PORT
29                value: "27017"
30            livenessProbe:
31              httpGet:
32                path: /health
33                port: 5000
34              initialDelaySeconds: 10
35              periodSeconds: 5
36              failureThreshold: 3
37            readinessProbe:
38              httpGet:
39                path: /ready
40                port: 5000
41              initialDelaySeconds: 5
42              periodSeconds: 5
43              failureThreshold: 3
44    ---
45    apiVersion: v1
46    kind: Service
47    metadata:
48      name: flask-service
49      annotations:
50        service.beta.kubernetes.io/aws-load-balancer-scheme: internet-facing
51    spec:
52      selector:
53        app: flask-app
54      ports:
55        - protocol: TCP
56          port: 5000
57          targetPort: 5000
58      type: LoadBalancer
59

```

1. The flask deployment file takes the image from Docker Hub and creates 3 replicas (pods) for load balancing. The RollingUpdate strategy ensures that pods are always available. Also, there is livenessProbe and readinessProbe to monitor the pods' health every 5 seconds, the time to failure and the initial delay. Moreover, the service is a LoadBalancer type that will expose the container as a loadbalancer service
- c. Then we expose these deployments
  - i. We deploy the mongo deployment using **kubectl apply -f mongo-deployment.yml** so that Kubernetes reads the file and creates and updates resources inside the Kubernetes. It first reads the mongo deployment which creates the mongo pod container and exposes its service for other pods inside the Kubernetes. Then, the pods for the flask node are created with the LoadBalancing service that routes network traffic to the flask pods. Moreover, the probes check the pods constantly to see if they fail.
  - d. Now that the Kubernetes created the deployments and services we checked that the pods and services were running using the **kubectl get pods** and **kubectl get services** commands
    - i. SCREENSHOT pods running
    - ii. SCREENSHOT services running
  - e. Also, as the Flask service is a LoadBalancer it has an external IP to access it so we use the **minikube service flask-service --url**
    - i. SCREENSHOT FLASK RUNNING BROWESER
3. Deploying the app on AWS EKS
  - a. Now that the app worked locally, we deployed it to AWS EKS using AWS CLI and kubectl
  - b. First we create a cluster in EKS