Introduction to FLASK for Model Deployment:

Step-by-Step Guide for Containerised Deployment

Use Case: Text Sentiment Analysis

<u>Prerequisites:</u>

1) **Python 3.7 and above with pip installed**
2) **Access to a code editor (e.g. Visual Studio Code & Notepad++)**
   Note: The following guide will be using Visual Studio Code.
3) **Access to terminal console**
   Note: The following guide will be mainly using Ubuntu 20.04.

Contents Page

A. Installation of Docker

Note: This guide was referenced from the following link: DigitalOcean – Docker Installation for Ubuntu 20.04. If you are a Mac user and would like to install docker, you can refer to Mac Docker.

1. Update the existing list of packages and install a few prerequisite packages which let apt use packages over HTTPS:

```
>> sudo apt-get update
>> sudo apt install apt-transport-https ca-certificates curl software-properties-common
```

2. Add the GPG key for the official Docker repository to your system:

```
>> curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

3. Add the Docker repository to APT sources:

```
>> sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu
focal stable"
```

4. Next, update the package database with the Docker packages from the newly added repo:

```
>> sudo apt update
```

5. Make sure you are about to install from the Docker repo instead of the default Ubuntu repo:

```
>> apt-cache policy docker-ce
```

You will see an output like this, although the Docker version may be different. Notice that docker-ce is not installed, but the candidate for installation is from the Docker repository for Ubuntu 20.04 (focal).

```
vinniechu@[redacted]:/mnt/c/Users/Vinne/Desktop/HTX/Workshop/Flask$ apt-cache policy docker-ce
docker-ce:
  Installed: (none)
  Candidate: 5:20.10.7~3-0~ubuntu-focal
  Version table:
     5:20.10.7~3-0~ubuntu-focal 500
        500 https://download.docker.com/linux/ubuntu focal/stable amd64 Packages
     5:20.10.6~3-0~ubuntu-focal 500
```

6. Finally, install docker:

```
>> sudo apt install docker-ce
```

7. Docker should now be installed, the daemon started, and the process enabled to start on boot. Check that it is running using the following command:

```
>> sudo systemctl status docker
```

The output should be like the screenshot below, which indicates that the service is active & running:

```
● docker.service - Docker Application Container Engine
     Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset>
     Active: active (running) since Thu 2021-05-20 13:52:21 UTC; 1 months 2 day>
TriggeredBy: ● docker.socket
       Docs: https://docs.docker.com
   Main PID: 719 (dockerd)
      Tasks: 10
     Memory: 124.4M
     CGroup: /system.slice/docker.service
             └─719 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/conta>
```

Note: For Windows users using Ubuntu on WSL 2, you can use the following command to make sure that your docker has started and is running:

```
>> sudo service docker start
```

B. Creating Docker Image

1. Create a file called Dockerfile that contains what you need to run the application. Place the file in the same folder as your application (i.e. app.py). The Dockerfile to run our Flask app looks like the following:

```
FROM python:3.9

COPY . /
RUN pip install -r requirements.txt

EXPOSE 5000
CMD ["python", "-u", "/app.py"]
```

Note: For the endpoint to be successfully accessed, the flask host must be initialised to '0.0.0.0' instead of the default 'localhost' (i.e. app.run(host='0.0.0.0')).

2. Build the Docker image on your computer, using the following command:

```
>> cd ./Workshop  #go into the workshop folder in terminal
>> sudo docker -t ballchuuuu/mloe_flask_workshop_xx1 .
```

Note:

- *The path in grey should be your path to the Flask application that you want to build.*
- *The path in blue should be the username of your Docker Hub account. If you do not have an account, create a free Docker Hub account at https://hub.docker.com/.*

If Docker is built successfully, you would see a similar output as shown below:

```
 ---> 150f268f1ed9
Step 4/4 : CMD ["python", "-u", "/app.py"]
 ---> Running in 57c6bc6734f4
Removing intermediate container 57c6bc6734f4
 ---> a970e93a8bf8
Successfully built a970e93a8bf8
Successfully tagged mloe_flask_workshop_xx1:latest
```

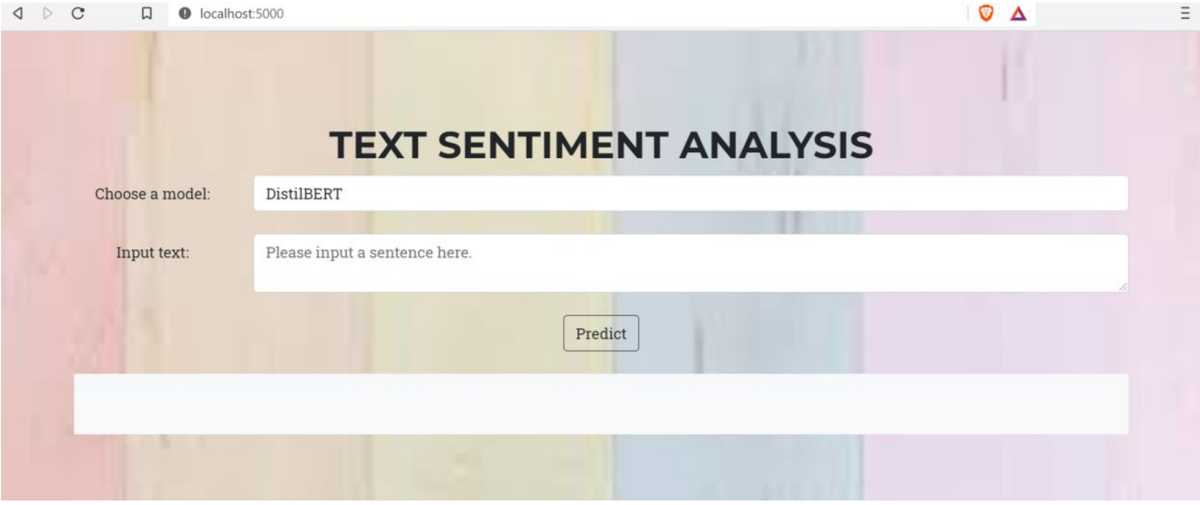
3. Run your Docker image locally:

```
>> sudo docker run --rm -d --name t mloe_flask_workshop_xx1 -p 5000:5000
ballchuuuu/mloe_flask_workshop_xx1
```

Breaking down the above command:

- `run`: runs a Docker container
- `--rm:` removes the container after it exits
- `-d`: runs the container in the background, without this the docker run command will wait in your shell
- `--name` *`mloe_flask_workshop_xx1`*: gives the container a name
- `-p 5000:5000:`sets the port
- *`balllchuuuu/mloe_flask_workshop_xx1:`* is the tag of the image that is to be started in the container
    - Note: *The path in blue should be the username of your Docker Hub account.*

4. Use your web browser to go to http://localhost:5000 and make sure your web application is working in its containerized environment. For this guide, your web browser should show the following output:



Note: To check the logs of the container, the following command can be used:

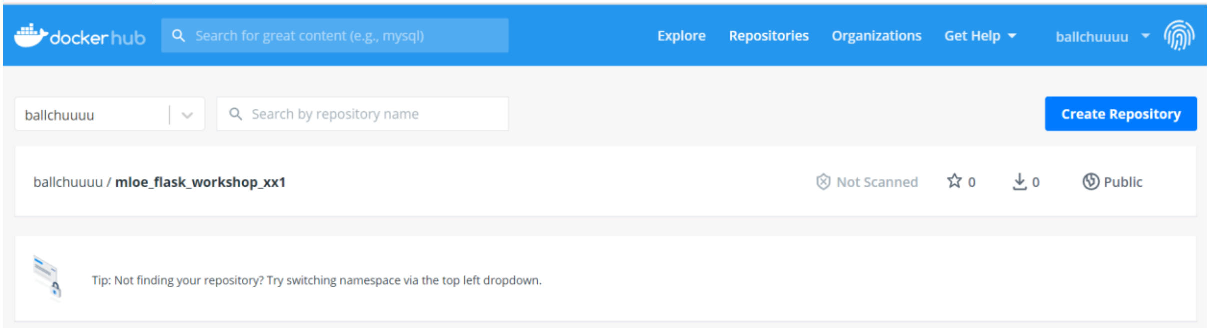
```
>> sudo docker container logs mole_flask_workshop
```

C. Hosting the Docker Image on Docker Hub

1. Once everything is working, log in to your Docker Hub account from your terminal using the following command:

```
>> sudo docker login
```

2. Create a new repository that mirrors the same name as your image tag (i.e. ballchuuuu/mloe_flask_workshop_xx1)



Note: The path in blue should be the username of your Docker Hub account.

3. Run the following command to push your image to Docker Hub:

```
>> sudo docker push ballchuuuu/mloe_flask_workshop_xx1
```

Note: The path in blue should be the username of your Docker Hub account.

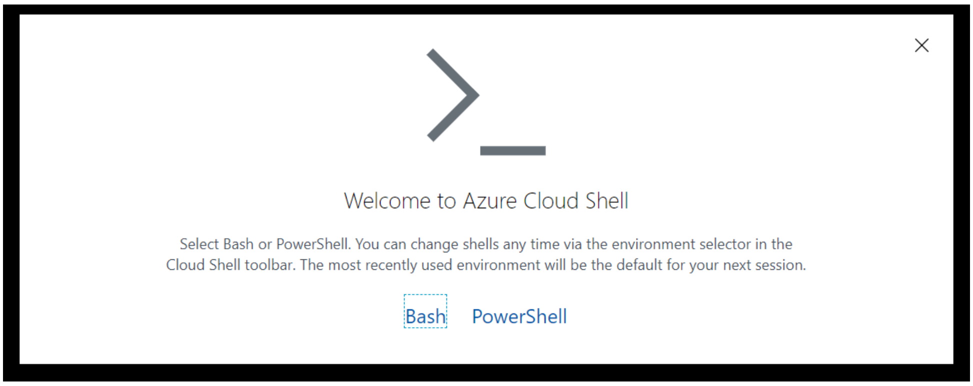
If the process is a success, you would see a similar output:

```
vinniechu@[redacted]:/mnt/c/Users/Vinne/Desktop/HTX/Workshop/Flask_Full_Code$ sudo docker push ballchuuuu/mloe_flask_workshop_xx1
Using default tag: latest
The push refers to repository [docker.io/ballchuuuu/mloe_flask_workshop_xx1]
b12371eba427: Pushed
a2f1fcb9b070: Pushed
126ba0c24224: Pushed
001ade22e15c: Pushed
97e852d9107e: Pushed
1591bf7ec708: Pushed
dd3097cd7909: Pushed
685934357c89: Pushed
ccb9b68523fd: Pushed
00bcea93703b: Pushed
688e187d6c79: Pushed
latest: digest: sha256:26fd2d822d758f53a8c9a4be507c958643c69e4e4855bd98a659884bf817030a size: 2644
```

D. API Deployment on Azure Kubernetes Service (AKS) for Scalability

Note: For more details on AKS, please refer to the following link: [AKS Documentation](#).

1. Log in to your Azure portal account and go to https://portal.azure.com/#cloudshell/. Select Bash as shown in the following screenshot:



2. Create a new resource group (if you do not have one) using the following command in the Bash console:

```
>> az group create --name rs_mloe_flask_workshop --location eastus
```

If the creation is sucessful, a similar output as below should be displayed:

```
vinnie@Azure:~$ az group create --name rs_mloe_flask_workshop --location eastus
{
  "id": "/subscriptions/2d8c4f52-6b66-4a85-90a5-c7913e3840c8/resourceGroups/rs_mloe_flask_workshop",
  "location": "eastus",
  "managedBy": null,
  "name": "rs_mloe_flask_workshop",
  "properties": {
    "provisioningState": "Succeeded"
  },
  "tags": null,
  "type": "Microsoft.Resources/resourceGroups"
}
```

3. Create a single-node Kubernetes cluster:

```
>> az aks create \
    --resource-group rs_mloe_flask_workshop \
    --name mloe-flask-workshop \
    --node-vm-size Standard_NC6  \
    --node-count 1 \
    --generate-ssh-keys
```

If the creation is sucessful, a similar output as below should be displayed:

```
vinnie@Azure:~$ az aks create \
>     --resource-group rs_mloe_flask_workshop \
>     --name mloe-flask-workshop \
>     --node-vm-size Standard_NC6  \
>     --node-count 1 \
>     --generate-ssh-keys

SSH key files '/home/vinnie/.ssh/id_rsa' and '/home/vinnie/.ssh/id_rsa.pub' have been generated under
ing machines without permanent storage like Azure Cloud Shell without an attached file share, back up
{
  "aadProfile": null,
  "addonProfiles": null,
```

Note: The VM-size was chosen to be Standard_NC6 (GPU-enabled) due to DistilBERT being a larger model. To see the different types of VMs, please refer to: [Azure link](#). You could also check out the readings on Pytorch & TensorFlow Serving readings for alternative deployment routes.

4. Add the credentials for that Kubernetes cluster to your local kubectl configuration:

```
>> az aks get-credentials --resource-group rs_mloe_flask_workshop --name mloe-flask-workshop
```

If the addition is successful, a similar output as below should be displayed:

```
vinnie@Azure:~$ az aks get-credentials --resource-group rs_mloe_flask_workshop --name mloe-flask-workshop
Merged "mloe-flask-workshop" as current context in /home/vinnie/.kube/config
```

5. Install NVDIA Device Plugin (For GPU-enabled VMs)

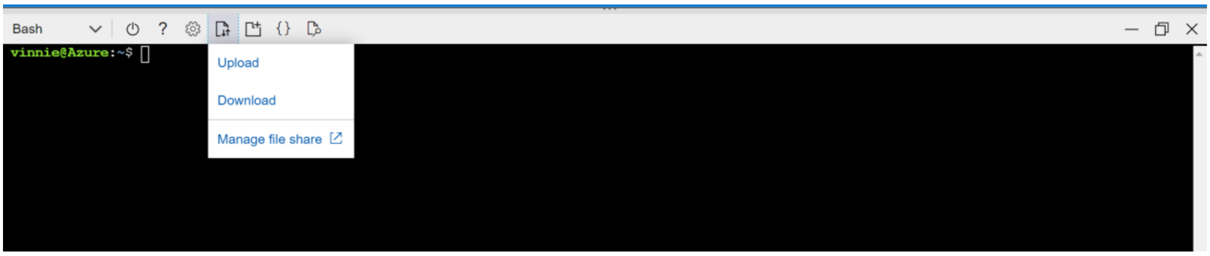
a) Create a namespace using the following command:

```
>> kubectl create namespace gpu-resources
```

b) Create a file named nvidia-device-plugin-ds.yaml locally and paste the following YAML manifest. This manifest is provided as part of the NVIDIA device plugin for Kubernetes project.

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: nvidia-device-plugin-daemonset
  namespace: gpu-resources
spec:
  selector:
    matchLabels:
      name: nvidia-device-plugin-ds
  updateStrategy:
    type: RollingUpdate
  template:
    metadata:
        scheduler.alpha.kubernetes.io/critical-pod: ""
      labels:
        name: nvidia-device-plugin-ds
    spec:
      tolerations:
      - key: CriticalAddonsOnly
        operator: Exists
      - key: nvidia.com/gpu
        operator: Exists
        effect: NoSchedule
      containers:
      - image: mcr.microsoft.com/oss/nvidia/k8s-device-plugin:1.11
        name: nvidia-device-plugin-ctr
        securityContext:
          allowPrivilegeEscalation: false
          capabilities:
            drop: ["ALL"]
        volumeMounts:
          - name: device-plugin
            mountPath: /var/lib/kubelet/device-plugins
      volumes:
        - name: device-plugin
          hostPath:
            path: /var/lib/kubelet/device-plugins
```

c) Upload the yaml file (i.e. nvidia-device-plugin-ds.yaml) to the Azure storage as shown in the screenshot below:



d) To create the DaemonSet, run the following command:

```
>> kubectl apply -f nvidia-device-plugin-ds.yaml
```

If the creation is successful, a similar output as below should be displayed:

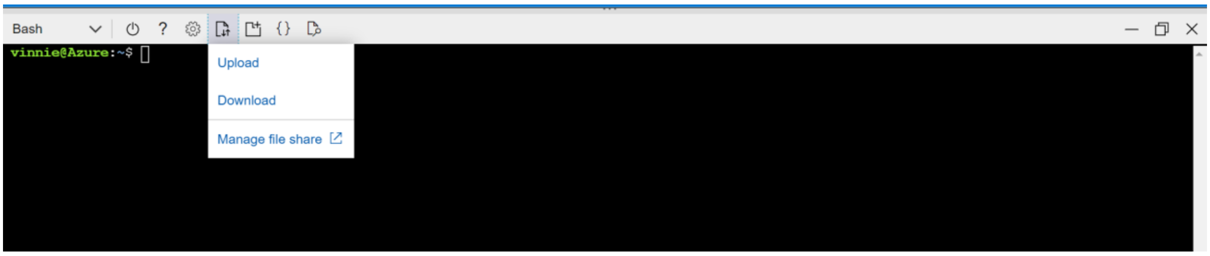
```
vinnie@Azure:~$ kubectl apply -f nvidia-device-plugin-ds.yaml
daemonset.apps/nvidia-device-plugin-daemonset created
```

7. Create the Kubernetes manifest yaml file locally (i.e. mloe_flask_workshop.yaml) to state the configurations. For this guide, the yaml file can be written as follows:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mloe-flask-workshop
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mloe-flask-workshop
  template:
    metadata:
      labels:
        app: mloe-flask-workshop
    spec:
      containers:
      - name: mloe-flask-workshop
        image: ballchuuuu/mloe_flask_workshop_xx1
        resources:
          limits:
            nvidia.com/gpu: 1
        ports:
        - containerPort: 5000

---
apiVersion: v1
kind: Service
metadata:
  name: mloe-flask-workshop
spec:
  type: LoadBalancer
  ports:
  - port: 5000
  selector:
    app: mloe-flask-workshop
```

8. Similar to Step 6(c), upload the yaml file (i.e. mloe_flask_workshop.yaml) to the Azure storage as shown in the screenshot below:



8. Send the Kubernetes manifest yaml file (i.e. mloe_flask_workshop.yaml) to the cluster:

```
>> kubectl apply -f mloe_flask_workshop.yaml
```

If the creation is successful, a similar output as below should be displayed:

```
vinnie@Azure:~$ kubectl apply -f mloe_flask_workshop.yaml
deployment.apps/mloe-flask-workshop created
service/mloe-flask-workshop created
```

9. Check the state of the created Kubernetes cluster, using the following command

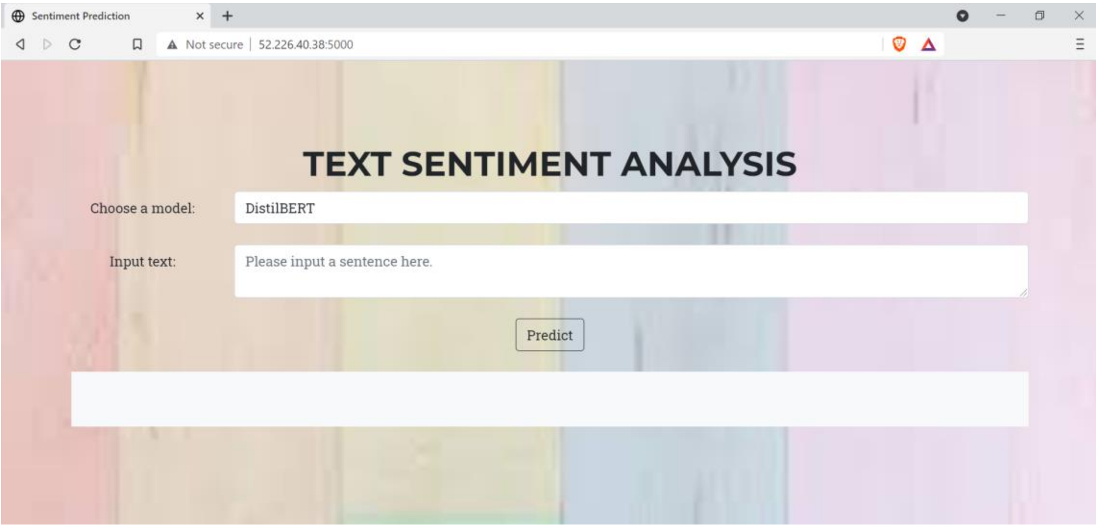
```
>> kubectl get pods,svc -o wide
```

If the deployment is successful, the status should be running as seen in the output below. This might take up to 5 minutes The external IP would be the access point used to reach our deployed application.

```
vinnie@Azure:~$ kubectl get pods,svc -o wide
NAME                                    READY   STATUS    RESTARTS   AGE     IP           NODE                                NOMINATED NODE
READINESS GATES
pod/mloe-flask-workshop-75785f7df-25z9q   1/1     Running   0          6m41s   10.244.0.8   aks-nodepool1-80122740-vmss000000   <none>
<none>
pod/mloe-flask-workshop-75785f7df-kxzb9   0/1     Pending   0          6m41s   <none>       <none>                              <none>
<none>

NAME                          TYPE           CLUSTER-IP     EXTERNAL-IP    PORT(S)          AGE    SELECTOR
service/kubernetes            ClusterIP      10.0.0.1       <none>         443/TCP          14m    <none>
service/mloe-flask-workshop   LoadBalancer   10.0.192.208   20.81.12.127   5000:31320/TCP   7m9s   app=mloe-flask-workshop
```

10. Following the use case of this guide, the [http:// 52.226.40.38:5000](http://52.226.40.38:5000) will thus show the following model inference webpage.



E. References & Extra Readings

1. Introduction to Docker
2. Introduction to Kubernetes
3. Deploying Machine Learning Models on Kubernetes
4. Using TensorFlow Serving with Kubernetes
5. PyTorch libraries for serving and training models at scale
6. Considerations for large node clusters in Kubernetes

--THE END---