**Cloud and Machine Learning**

**Homework 5**

-Avantika Singh

In this exercise I have trained and deployed an Image Recognition Model using Kubernetes on IBM Cloud. We begin with developing Kubernetes artifacts to train a Deep Learning Model and then use the trained model to provide an inference service. I have built a simple flask application with a front end that enables the user to upload an image and classify it into one of the four categories. The web interface provides the user with a train button which runs the model training job on the cloud and provides a prediction based on the input image uploaded by the user.   
Note: The focus of this experiment is on Kubernetes rather than the classification ability of the model hence the dataset used to train the model is limited.

**Part 1: Steps for execution:**

1. We begin with building docker images that are used containerize the source code and the environment required for the execution of the code. We run and build the applications locally before pushing the images on to docker hub.

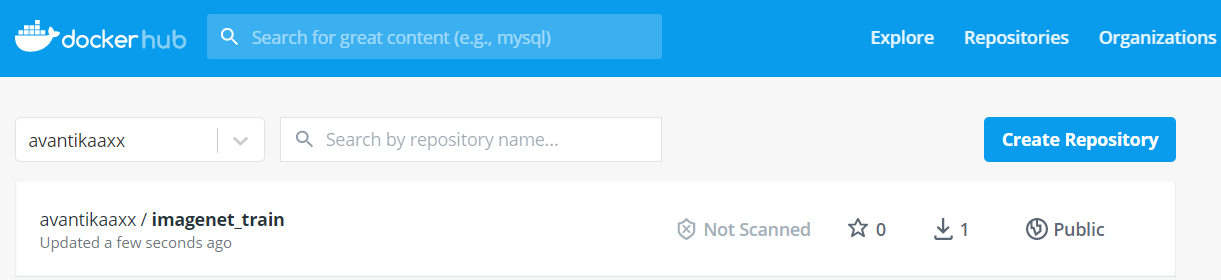
* Build the docker image:

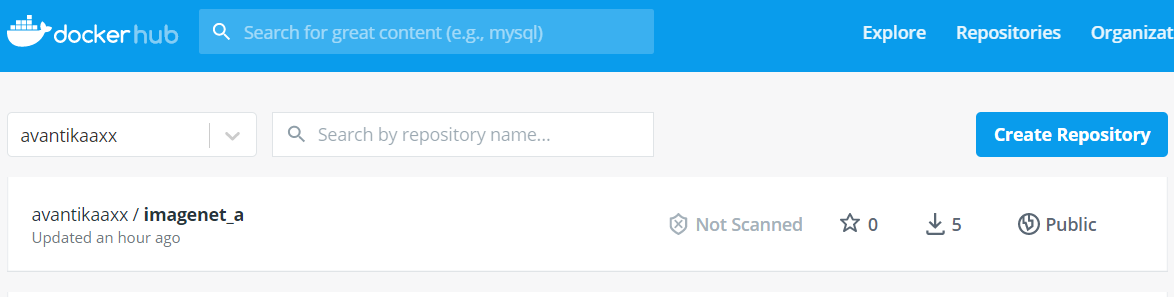
**docker build -t <image\_name> .**

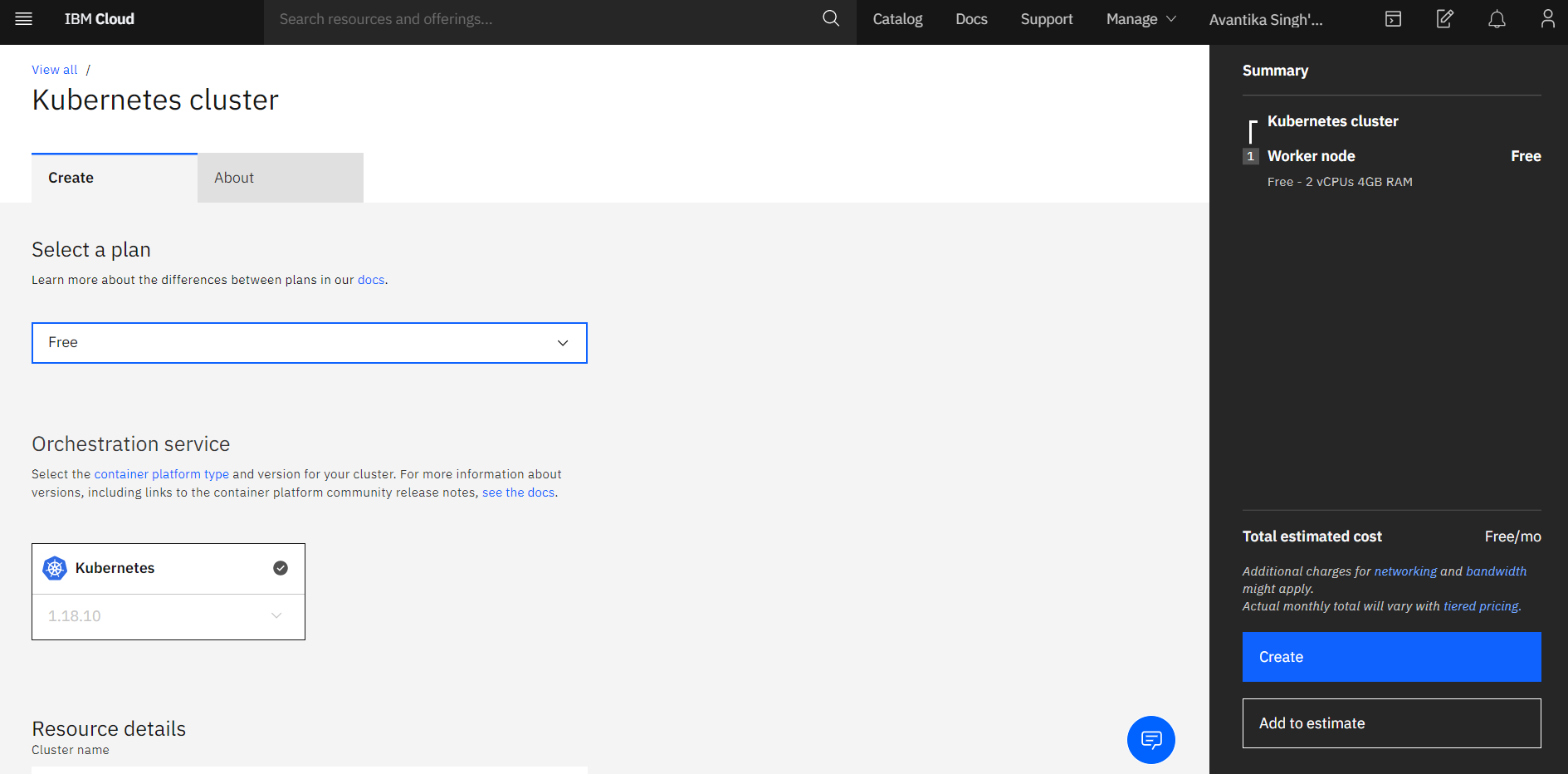
* Create an alias/tag for the id of the image.( Not necessary but good practice as per docker docs.)

**docker tag <image ID> <docker hub username>/<image name>**

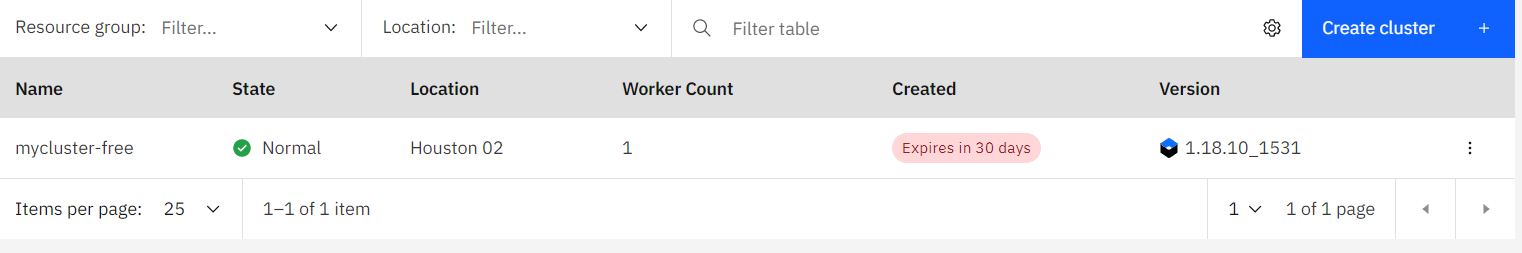
* Push the image onto docker hub.  
  **docker push <docker hub username>/<image name>**

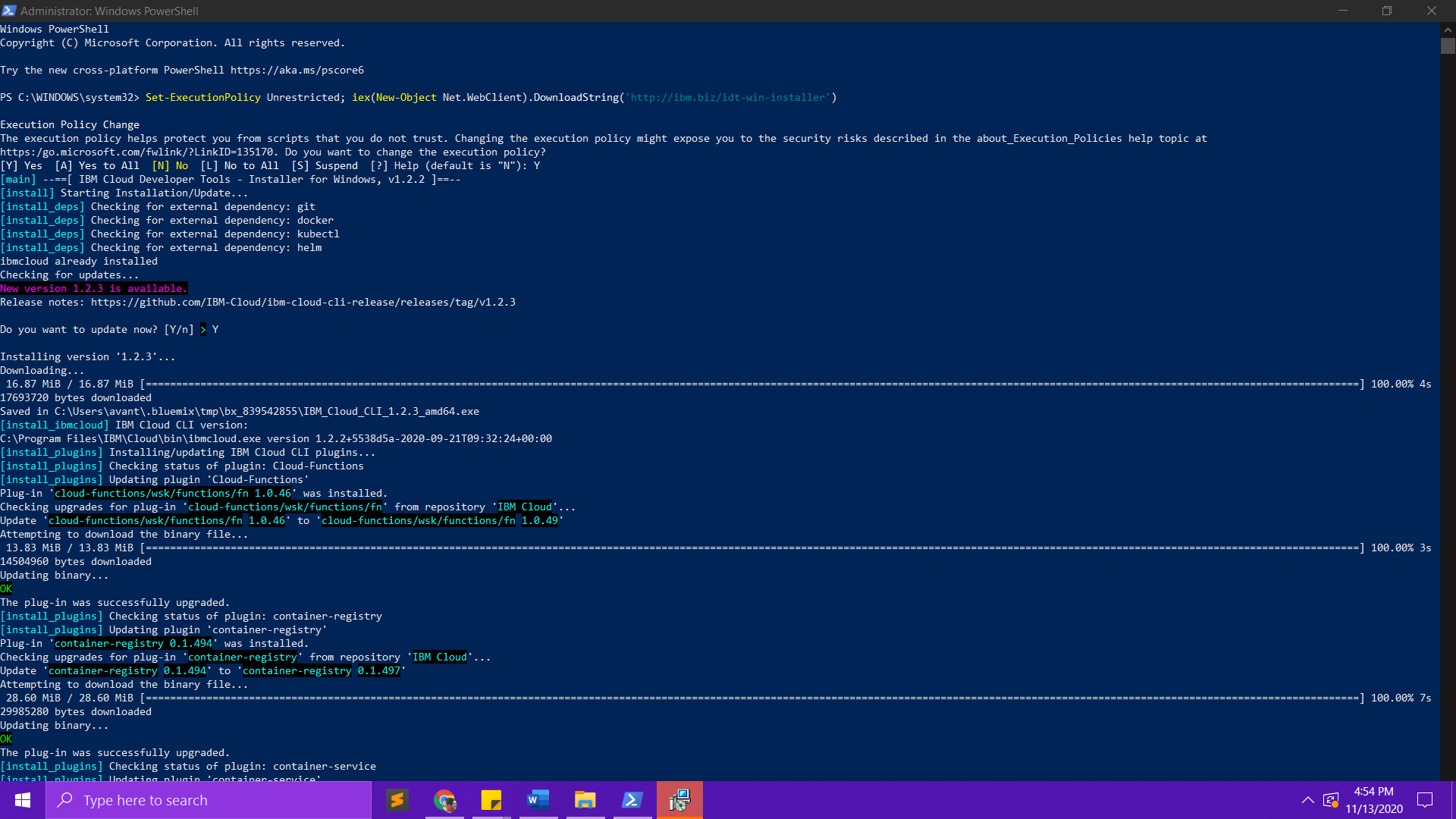
Two separate docker images were created and pushed on the docker hub one to train the model (imagenet\_train) and the other for the inference(imagenet\_a). 

  
 2. We then build a free Kubernetes cluster using the IBM cloud dashboard. We are provisioned a cluster that has one worker node which in turn has the following system specifications: 2 CPU’s and 4 GB RAM.



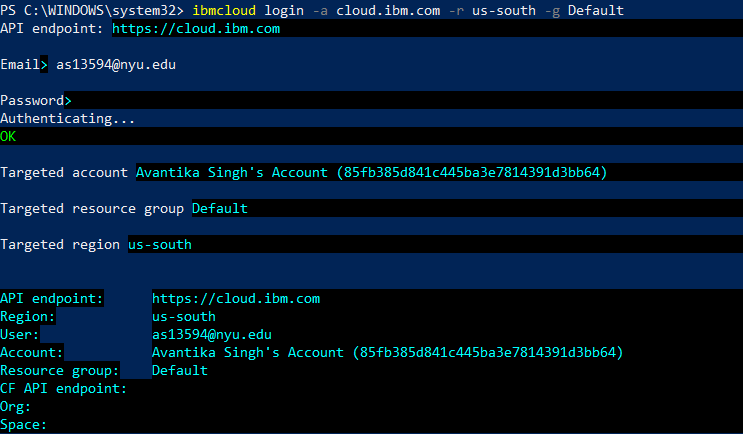
As can been observed from the screenshots attached below, a Kubernetes cluster(mycluster-free) has been created.



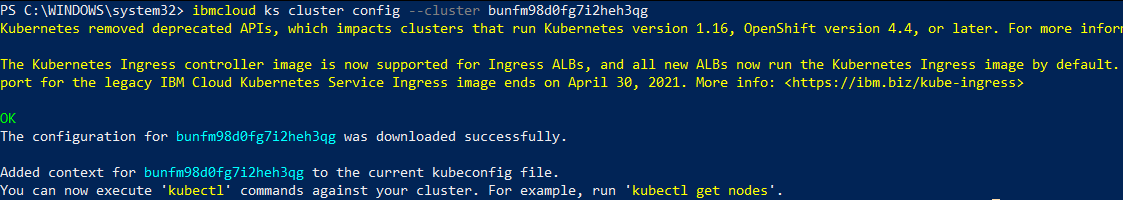
3. We move on to a one-time setup of the IBM CLI tools that we would be using to access the cluster.   


4. **Accessing the cluster :** We can then login to IBM cloud from our local machine using the command line interface. The commands have been attached for your reference.

* To login to your IBM Cloud account:

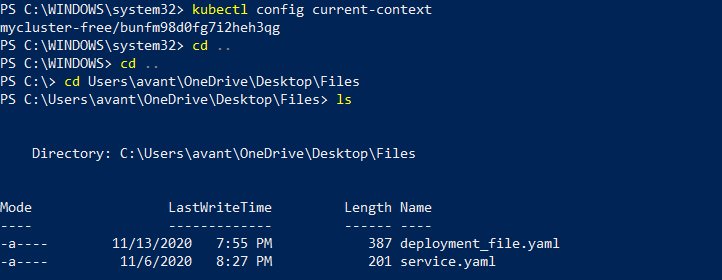
**ibmcloud login -a cloud.ibm.com -r us-south -g Default** 

* Set the Kubernetes context to your cluster for this terminal session

**ibmcloud ks cluster config --cluster bunfm98d0fg7i2heh3qg**

* Verify that you can connect to your cluster.

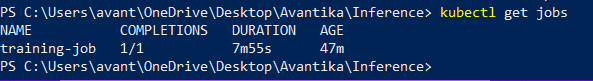
**kubectl config current-context**

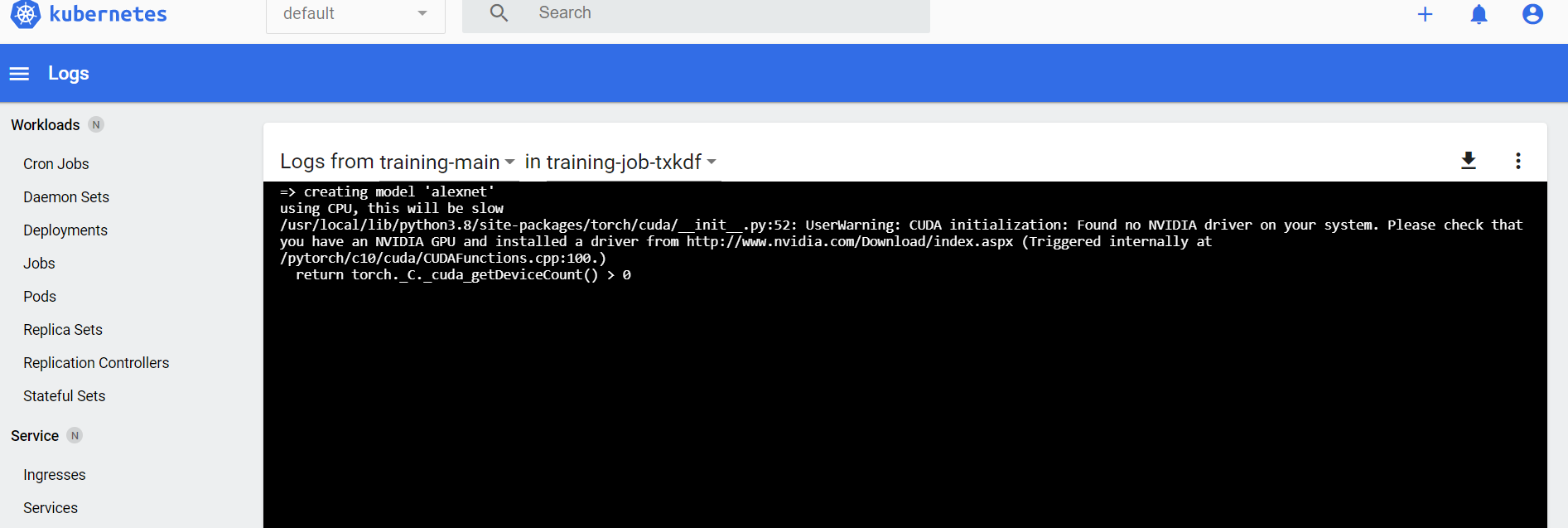


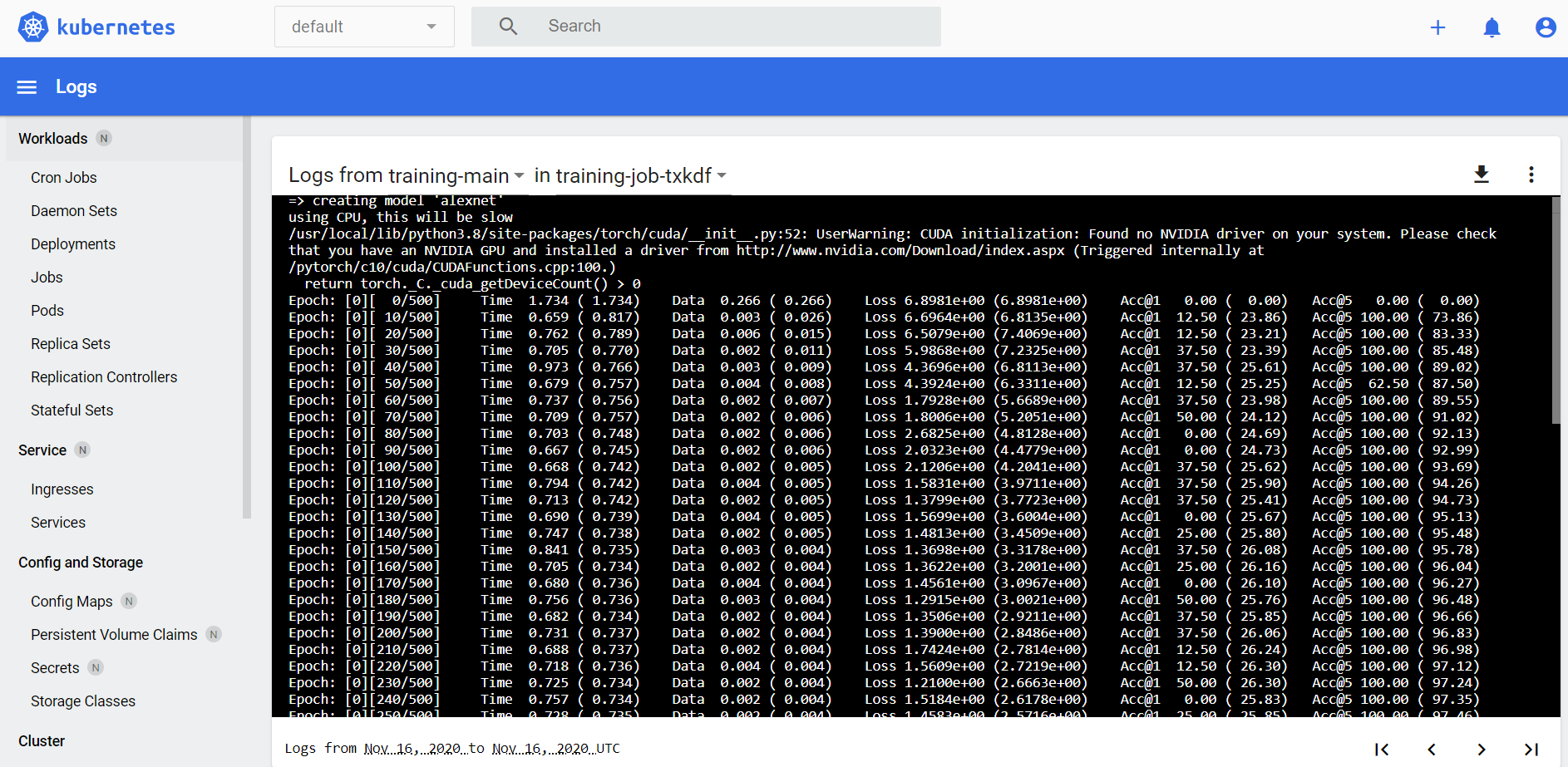
5. **Training the Image Recognition model:**

We use the command **​“kubectl apply -f job.yaml”** to run the job file for training the model. To check the status of the job we can use the following command :

**​“kubectl get jobs”**   
What this does is it runs the model and stores the trained model ​ in the host storage that will be used for inference. This uses the image from my docker hub under my repository avantikaaxx/imagenet\_train

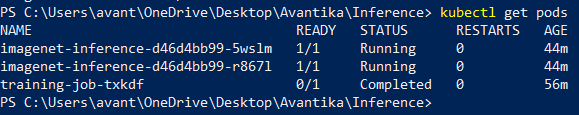






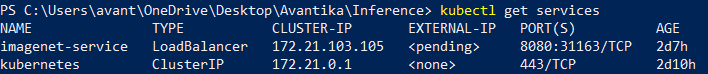
6. **Model Inference :**

Use command “**kubectl apply -f deployment\_jobs.yaml**” to create the deployment.   
  
Use the command “**kubectl get deployments**” to check the status of the deployment.  
  
This will make use of the trained model to deploy the application on IBM cloud using Kubernetes artifacts and will use the image from docker hub under my repository.  
We use the following command for Kubernetes services configuration  
**kubectl apply -f service.yaml**  
and the below command to verify the status of the services   
**kubectl get services**

7. Use the command “**kubectl get pods**” to check the pod status. Now that the application has been deployed on the cloud, we can access it from the local machine.  


8. Try finding out the external IP/public IP and node port to access the application/service from the local machine. There are 2 ways one can find out their external IP by logging into the Kubernetes dashboard or using the following command :“kubectl get nodes-o yaml | grep External IP -C 1“ this command gets the external IP from the node , the other way is to login to the dashboard and get into the worker nodes as shown below to find out the external-IP. Get the nodePort from “kubectl get services” now use the external-IP and the nodePort to access the application from the web browser. We found the public IP to be 169.57.85.82





And the node port to be 31163.

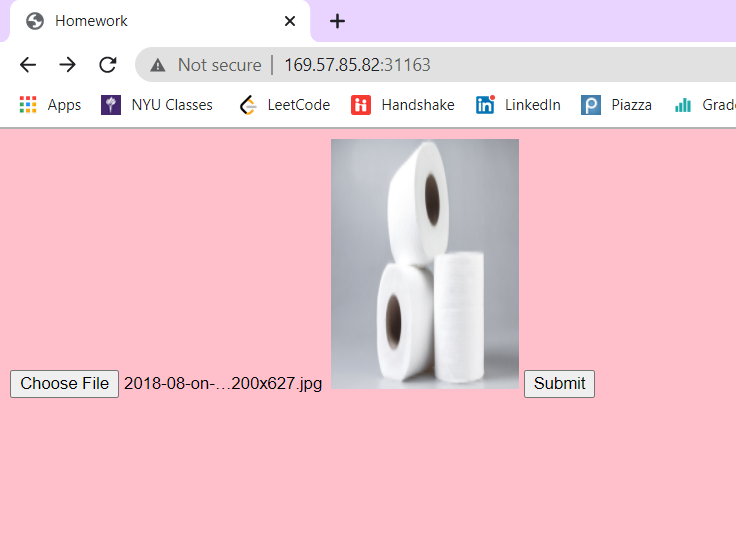
8. Go into your web browser and type http://external-IP:nodePort . You can now access the application that’s been deployed on the IBM cloud.

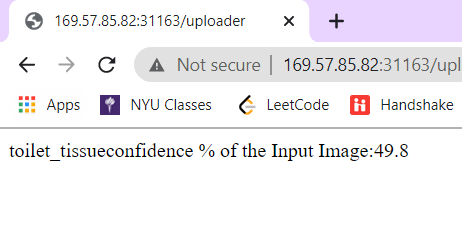
We use the following URL to access our service :

<http://169.57.85.82:31163/>

**Test case1:**

**We upload an image and click on Submit .The application takes the input image and returns a prediction indicating the object identified in the image as well as a confidence percentage for the corresponding analysis.**

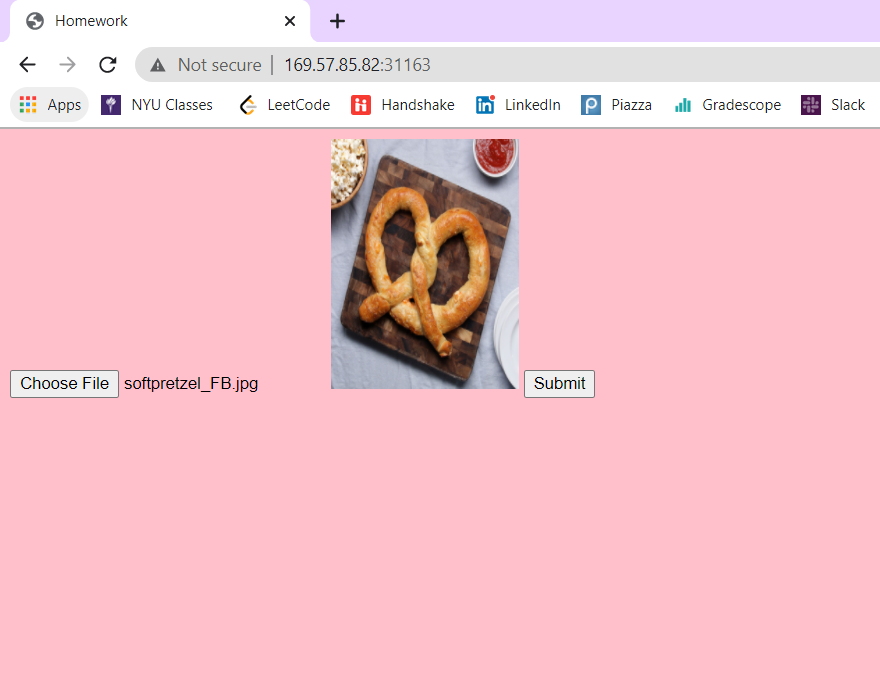


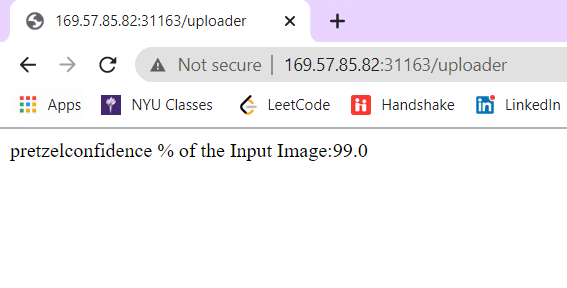


Our model accurately predicts the presence of toilet tissue in the input image with a confidence percentage of 49.8%.

**Test Case 2:**

For test case 2, we provide a pretzel as the input image to the application. The model begins training when the user clicks on the train button and returns an accurate inference(Pretzel) with a much higher confidence of 99%.





# Part 2: Conclusion :

**Write a small report on your experiences: What Kubernetes controllers did you use for training and inference and why? How did you get the trained model into the inference service?**(\*\*Mentioned under host volume)

Through the experiment I got a chance to explore the Kubernetes framework. Kubernetes provides support for the management and  
orchestration of different workloads and services on the containers. I used my Kubernetes cluster to train a deep learning model and run an Image Recognition service on the cluster.

The Kubernetes controllers/artifacts that I used for training the DL model and for the inference part are as follows:

* **Job:** To train my model and store the model in the host storage space. A Job creates one or more pods and ensures that a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (ie, Job) is complete. Deleting a Job will clean up the Pods it created.
* **Deployment and service :** Used to use the trained model stored in the host storage and deploy it on the cloud as an inference service. Even though pods are the basic unit of computation in Kubernetes, they aren’t usually launched on a cluster directly. Instead, they are typically managed by a “**deployment**” which serves as one more layer of abstraction. A deployment’s primary purpose is to declare how many replicas a pod will be running and by using a deployment, one doesn’t have to manually manage the pods. When a deployment is added to the cluster, it will automatically create the requested number of pods and in case a pod dies (is rendered inactive), the deployment will re-create it automatically. Since inference is nothing but serving a trained machine learning model to end users for use, it needs to be active at all times to be able to handle any and all requests. Hence deployment is most suitable for this case as it allows to set up multiple replicas of the application and thus will ensure that our model is up running even if one or more pods become inactive.  
  In Kubernetes, a **Service** is an abstraction to expose an application running on a set of [Pods](https://kubernetes.io/docs/concepts/workloads/pods/) as a network service. It defines a logical set of Pods and a policy by which to access them (sometimes this pattern is called a micro-service).
* **Pods** are automatically generated when we create jobs and deployments. For training, pods were used. Usually, the task of training a machine learning model is not repeated as often as inferencing from said model would be. Therefore, deployments would not be necessary in this case and using a pod would be enough.
* **Host storage/host volume** was used as a shared space/storage between containers for the trained model to be stored and enable its use as an inference service. So, the trained model was saved on the shared space and the same space was shared by the inference service to make use of it and deploy it on the web.

**What would you do differently if you are given more time?**

1. If given more time for the execution of the assignment I would like to change from host volume storage to persistent volumes. Local storage associated with each node in a Kubernetes environment is used as a temporary cache, but any data saved locally doesn’t persist. For this reason, persistent volumes could have been used which would allow permanent data storage in Kubernetes to ensure that the trained model is stored and is accessible by the inference code at any time.
2. Moreover, I would try to achieve better accuracy with the model through relevant hyperparameter tuning in the training phase.
3. I would also try to explore namespaces and other Kubernetes artifacts, to change number of replicas sets for the application.
4. I would also explore how the load balancing works on a Kubernetes cluster. Since IBM cloud doesn’t offer LoadBalancer type under the free plan, NodePort had to be used. But under standard or other plans, LoadBalancer can be used which allows to directly expose a service and forward all traffic on a specified port to the service regardless of protocol (HTTP, TCP, UDP, etc).