Name: Ishan Khanka

Net ID: IK1304

**Introduction**

For this assignment, I have trained and deployed an Image Recognition Model using Kubernetes on IBM Cloud.

Firstly, I develop Kubernetes artifacts to train a Deep Learning Model and then used the trained model to provide an inference service. For the 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.

Note: As the focus of this experiment is on Kubernetes rather than the classification ability  
of the model, hence it classifies into only 4 categories.

Additionally, for experimentation and exploration, I have created a checkbox button which allows you to either use newly trained model weights or pretrained weights.

File locations:

|  |  |
| --- | --- |
| Training: Dockerfile, K8S yaml file and Code | Train |
| Inference: Dockerfile, K8S yaml file and Code | Inference |
| Certificates | Certificates |

**Steps: Create Custom training dataset**

**Custom model**

Run the commands below to create a custom model definition, replacing <num-classes> with the number of classes in your dataset.

$ cd config/ # Navigate to config dir

$ bash create\_custom\_model.sh 3 # Will create custom model 'yolov3-custom.cfg'

**Classes**

Add class names to data/custom/classes.names. WBC, RBC and Platelets

**Image Folder**

Move the images of your dataset to data/custom/images/.

(To get file names use : dir /b /a-d)

**Annotation Folder**

Move your annotations to data/custom/labels/. The dataloader expects that the annotation file corresponding to the image data/custom/images/train.jpg has the path data/custom/labels/train.txt. Each row in the annotation file should define one bounding box, using the syntax label\_idx x\_center y\_center width height. The coordinates should be scaled [0, 1], and the label\_idx should be zero-indexed and correspond to the row number of the class name in data/custom/classes.names.

**Define Train and Validation Sets**

In data/custom/train.txt and data/custom/valid.txt, add paths to images that will be used as train and validation data respectively.

**Train**

To train on the custom dataset run:

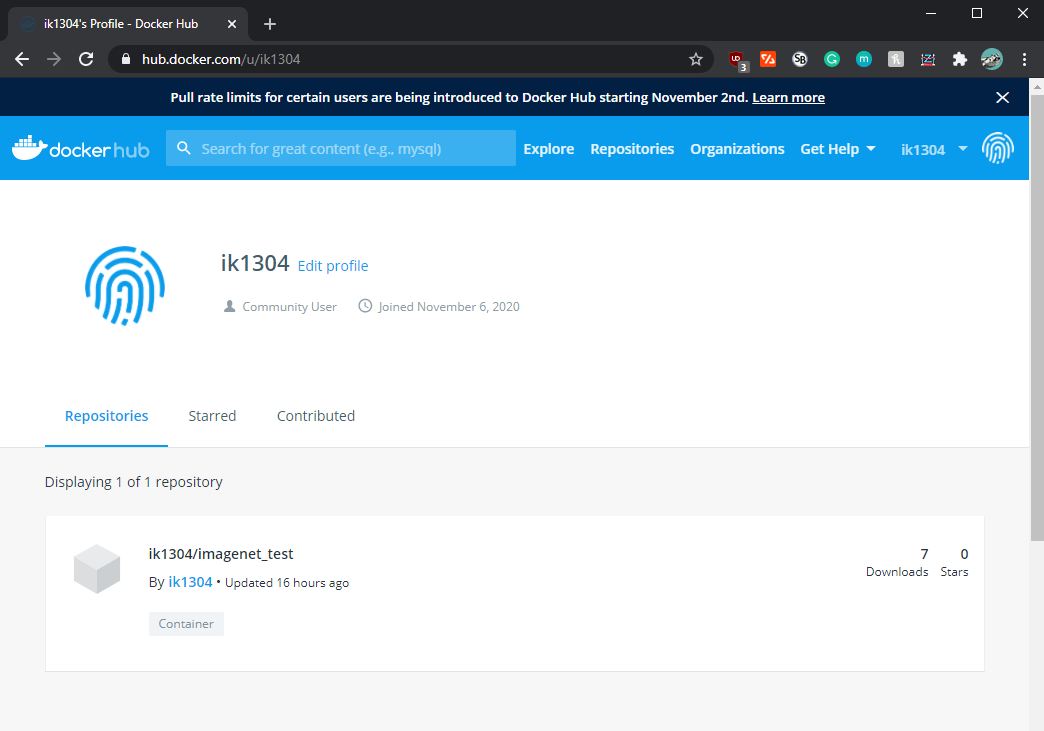
$ python3 train.py --model\_def config/yolov3-custom.cfg --data\_config config/custom.data

Add --pretrained\_weights weights/darknet53.conv.74 to train using a backend pretrained on ImageNet.

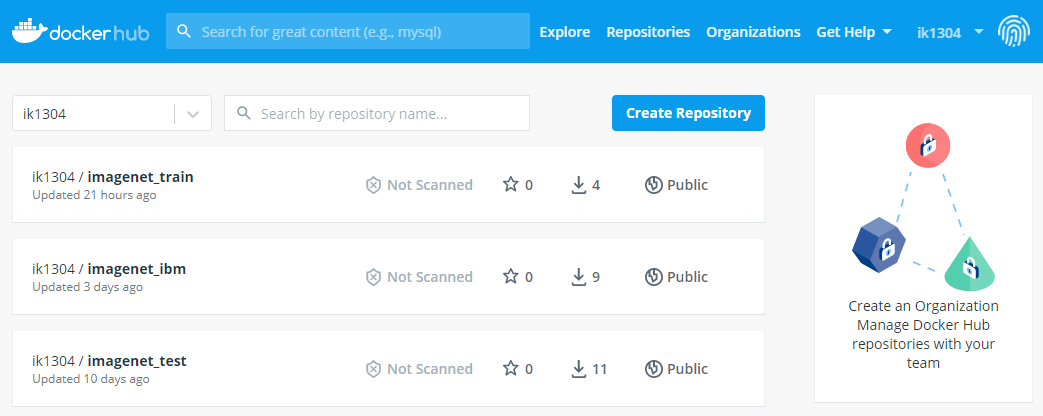
**Steps: Build and run the service on K8S on IBM Cloud**

1. **Dockerize the application for training and inference**

* Make account on <https://hub.docker.com/>

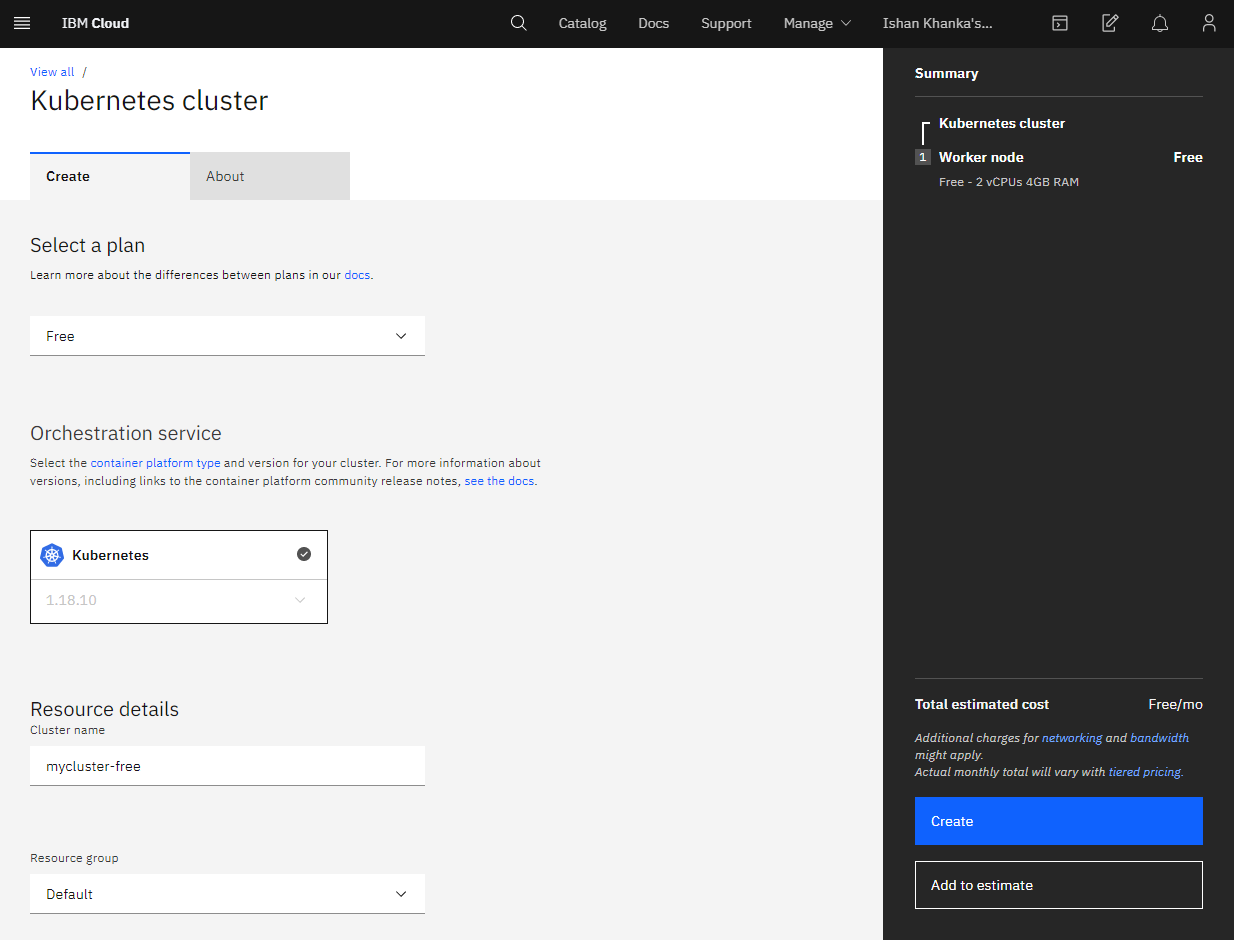


* Move to the directory containing Dockerfile for training or inference (train / inference)
* Build Docker (for training): “docker build -t yolo .”
* Build Docker (for inference): “docker build -t imagenet\_inference .”
* Run: docker tag imagenet\_train ik1304/imagenet\_train
* Run: docker tag imagenet\_inference ik1304/imagenet\_inference
* Push the training image: docker push ik1304/imagenet\_train
* Push the inference image: docker push ik1304/imagenet\_inference

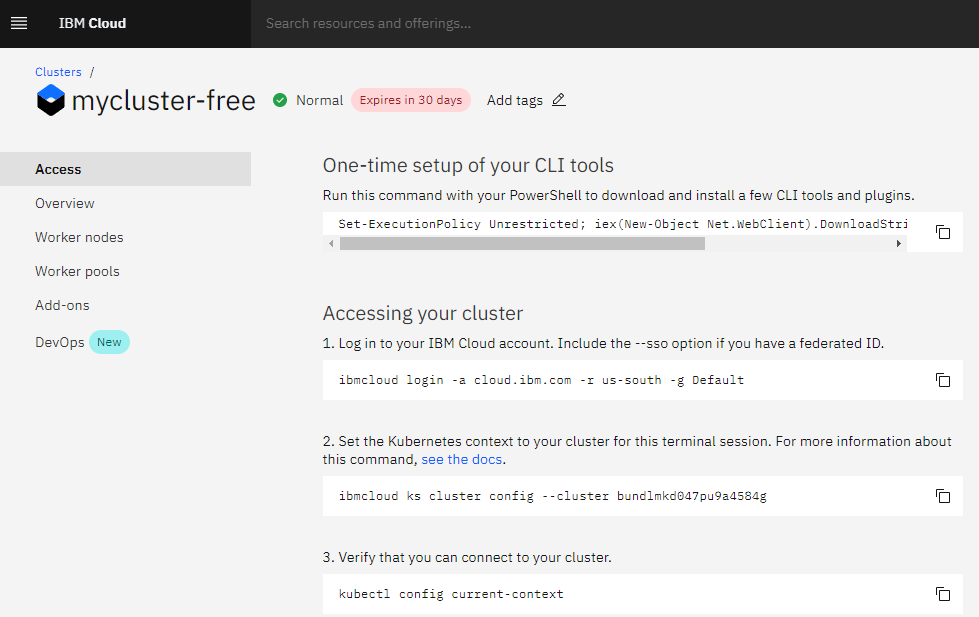


1. **Deploy and run the service on K8S on IBM Cloud**

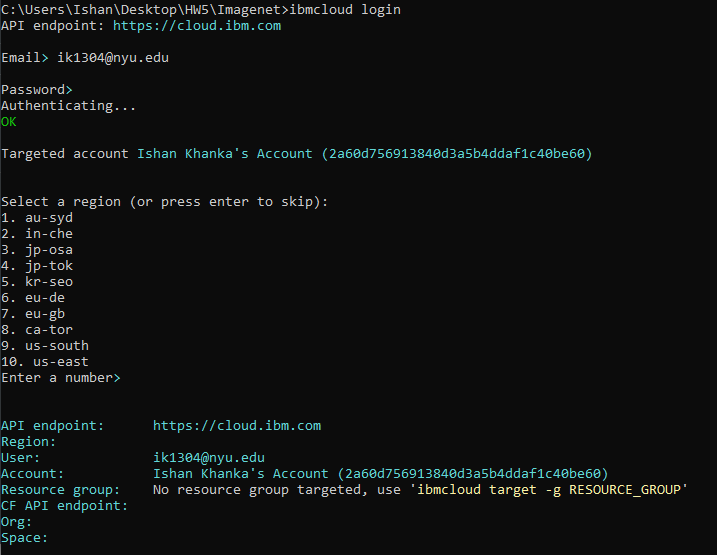
* Create IBM cluster by going to: <https://cloud.ibm.com/kubernetes/clusters>



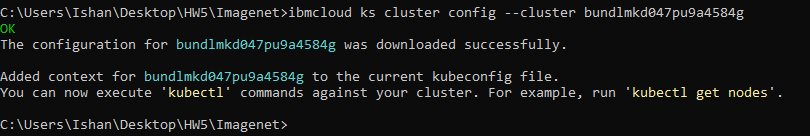
* Follow the CMD on [https://cloud.ibm.com/kubernetes/clusters/<your\_cluster](https://cloud.ibm.com/kubernetes/clusters/%3cyour_cluster)> to access your cluster



* Login to ibm cloud: “ibmcloud login”



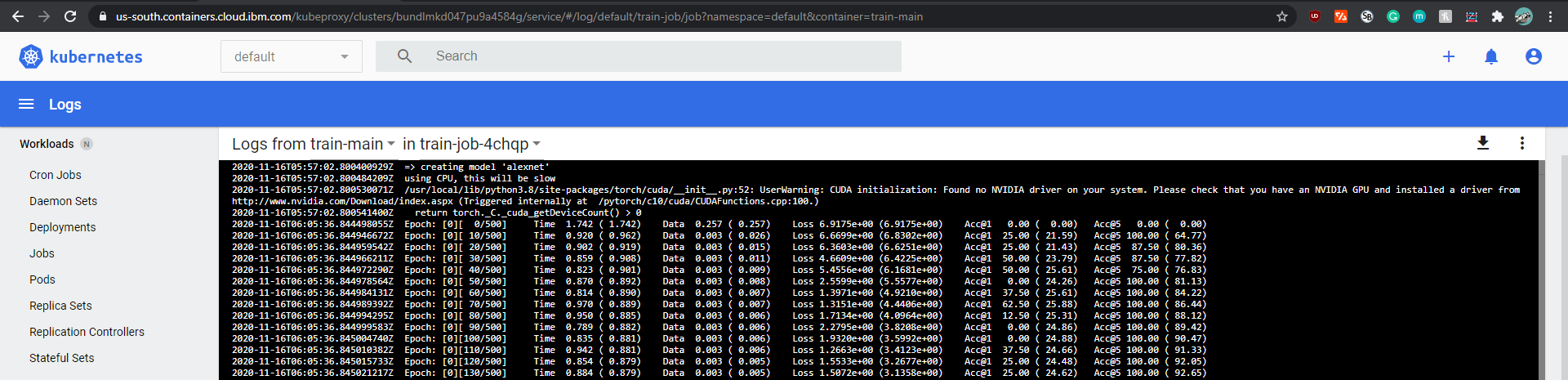
* Set the Kubernetes context to your cluster for this terminal session



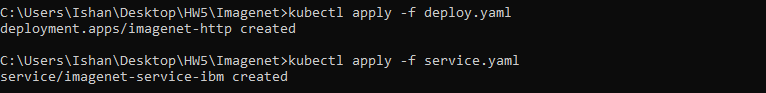
* Verify the connection: “kubectl config current-context”



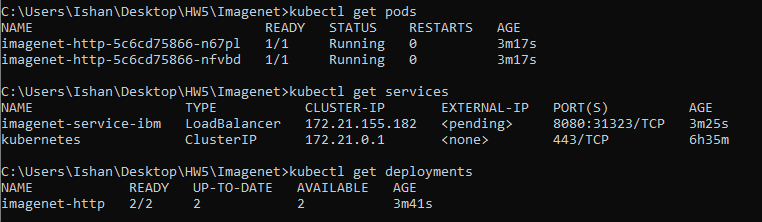
* Run the training job: ‘’kubectl apply -f train.yaml”
* Logs after successful training:

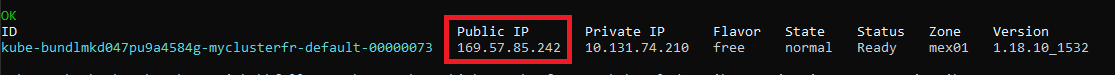


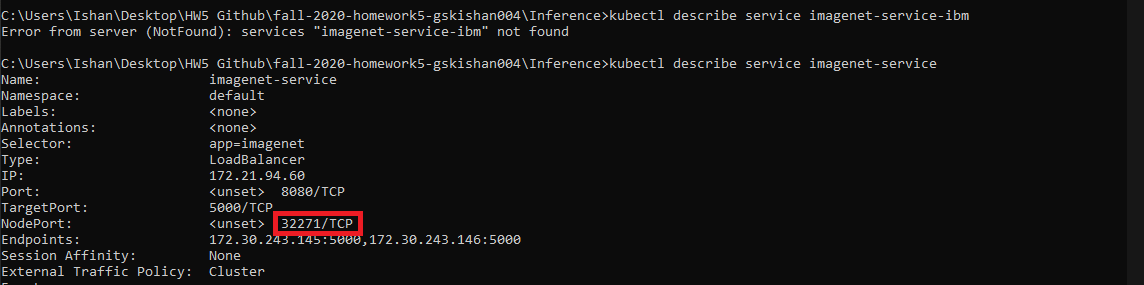
* Provisioning front end for inference: “kubectl apply -f deploy.yaml” and “kubectl apply -f service.yaml”



* Check pods: “kubectl get pods”
* Check services: “kubectl get services”
* Check deployments: “kubectl get deployments”



* Note down Public IP: “ibmcloud ks workers --cluster mycluster-free”
* Note down PORT: “kubectl describe service imagenet-service”



* To access the application go to the link: Public IP:PORT in our case <http://169.57.85.242:32271/>

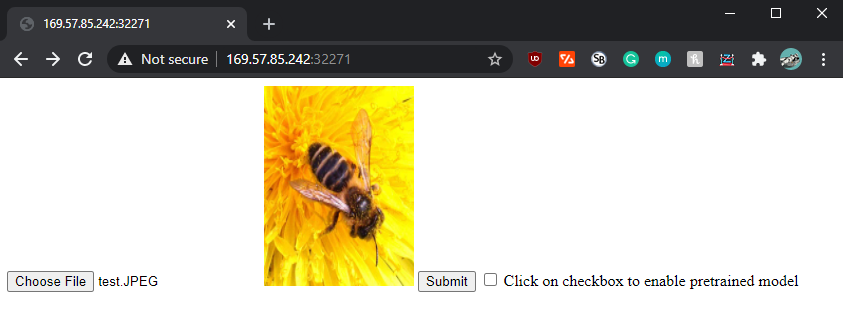
1. **Instructions for training and inference**

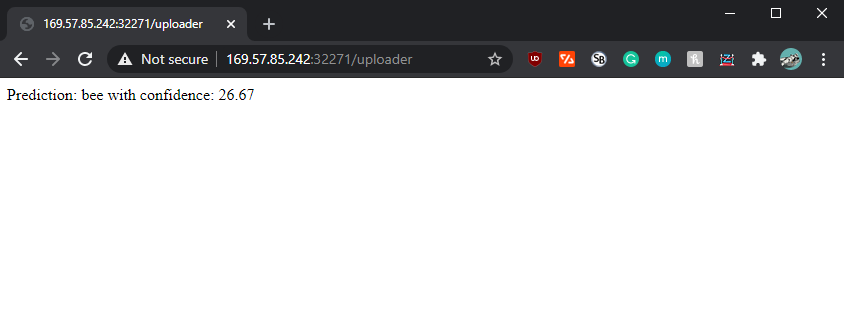
Training:

* It is done automatically when user runs ‘‘’kubectl apply -f train.yaml” and the weights are stored in Host storage.

Inference:

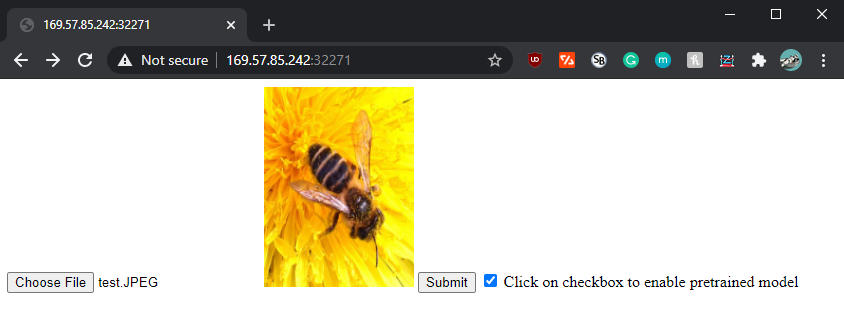
* Chose the image by clicking on Choose Files button and you will get the preview of Image
* Click on Submit to get prediction
* Here Prediction is Bee with 27% confidence.

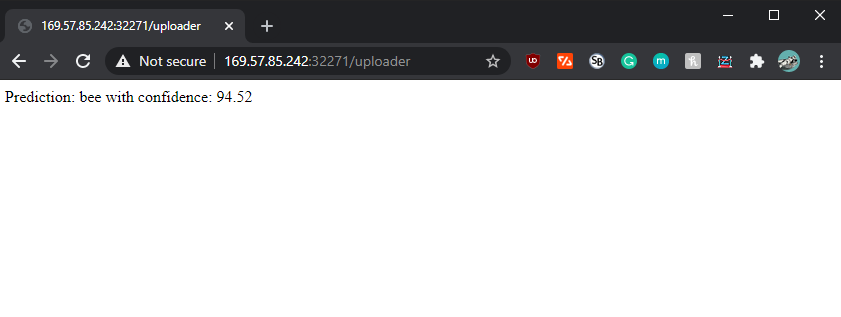




Inference (with pretrained weights):

* To use pertained AlexNet model weights, tick the checkbox before submitting
* Here you will get higher confidence (95%) as we are using pretrained model weights (which were trained for hours on hundreds of GPUs on ImageNet)





**Lessons learned and experience**

What Kubernetes controllers did you use for training and inference and why? How did you get the trained model into the inference service?

This assignment helped me to explore Kubernetes framework. It 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 and objects that I used for training and inference part are as follows:

* **Job**:
  + It creates one or more Pods and ensures that a specified number of them successfully terminate after completion. When a specified number of successful completions is reached, the Job is complete.
  + The Job object will start a new Pod if the first Pod fails or is deleted.
  + I have created a Kubernetes Job object to reliably run one Pod to completion for training the Deep Learning model. It is used to train the model and it stores the model in the host storage space after training.
* **Deployment**:
  + Deployment helps describe a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate.
  + A deployment’s primary purpose is to declare how many replicas a pod will be running and by using a deployment. In my case, I have defined Deployments to create new 2 Replica Sets.
  + When a deployment is added to the cluster, it will automatically create the requested number of pods and in case a pod dies, the deployment will re-create it automatically.
  + Since inference serving a front end for the inference, it always needs to be active to be able to handle all requests. Hence deployment is most suitable for this case as it allows to set up multiple replicas of the application.
* **Service**:
  + Service is an abstraction which defines a logical set of Pods and a policy by which to access them.
  + Kubernetes Pods are created and destroyed to match the state of your cluster. So, after using Deployment to run the app, it can create and destroy Pods dynamically.
  + But each Pod gets its own IP address, and the set of Pods running in one moment in time could be different from the set of Pods running that application a moment later.
  + Hence, to solve the above issues with respect to how pods connect to each other and keep track of which IP address to connect to, I have used service.
  + Service provides the abstraction and the policy by which Pods can interact with each other.
* **Pods**
  + They are automatically generated when we create jobs and deployments.
  + For training, pods were used as the task of training a machine learning model is not repeated as often as inferencing.
  + Hence, for training using a pod would be enough.
* **Host storage/host volume**
  + I used Host storage/host volume as a shared space/storage between containers for the train and inference part.
  + This allows the trained model to be stored on the shared space which then the inference service (flask application) utilizes for the prediction.
* **Two separate docker images**
  + One was created and pushed on the docker hub to train the model and the other for the flask application (inference).

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

* Different hyperparameters

Try to achieve better accuracy for the trained model through relevant hyperparameter tuning in the training phase

* Front end with option to train with user provided hyperparameters

Currently, the training was done with the train.yaml file, but for the ease of use for the end user – a train button on the front-end flask application would be helpful. Additionally, we can add form fields where the user can key in hyperparameters like batch size, learning rate, epochs etc. This would allow user to train and then immediately test out the performance of the model.

* Persistent storage

Change 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 does not 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.

* More artifacts

I would also try to explore more Kubernetes artifacts controllers and objects. And additionally, change the number of replicas set for the application.

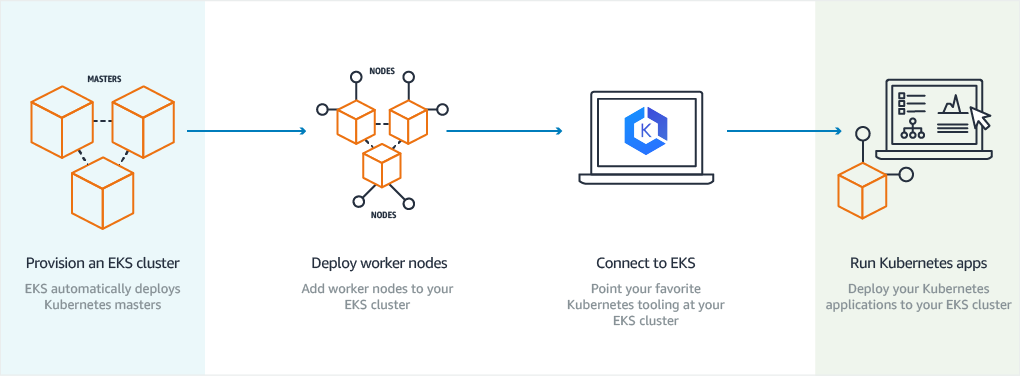
* Load balancing

Test out the workings of load balancing on a Kubernetes cluster. In our case we used Node port as free / student plan of IBM cloud does not offer Load Balancer type. But under standard or other plans, Load Balancer can be  
used which allows to directly expose a service and forward all traffic on a specified port to the service.

**Bonus: Create a Kubernetes cluster in AWS or GCP**

Kubernetes cluster in AWS and compare your experience

To create Kubernetes clusters in AWS, we used Elastic Kubernetes Service (Amazon EKS). This is a fully managed Kubernetes service.

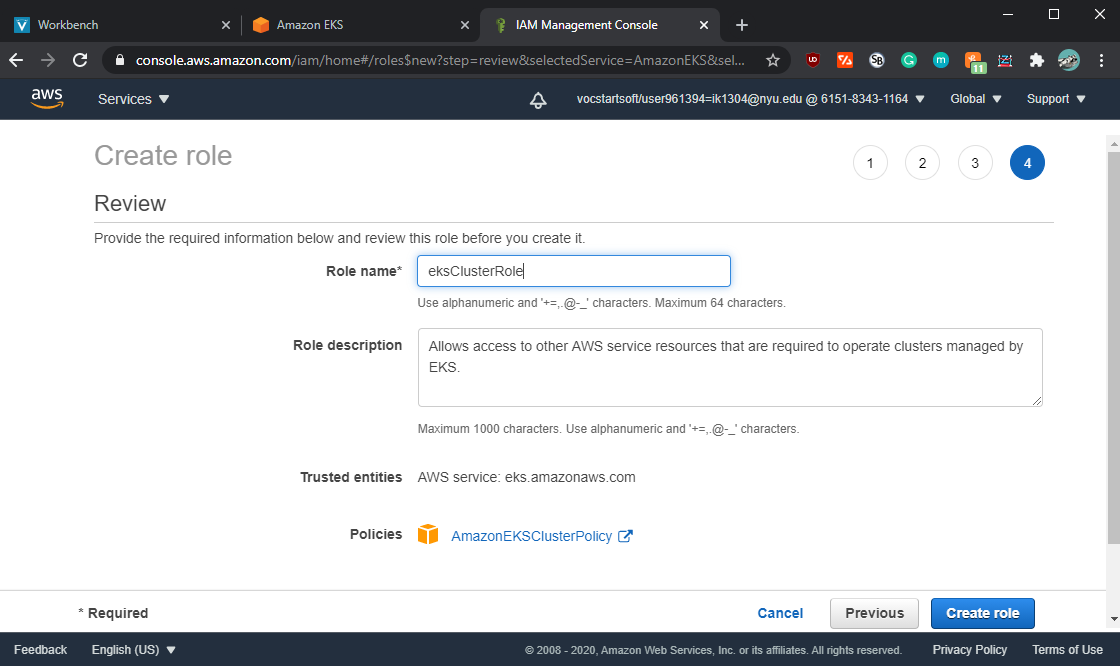


This tool is relatively simple to start using as it comes with good documentation and UI interface to create Kubernetes cluster. The hardest part of getting started is getting a docker image setup for the first time. Getting workflows and tasks setup is simple to understand and implement. AWS provides better reliability due to large number of available regions for cluster deployment.

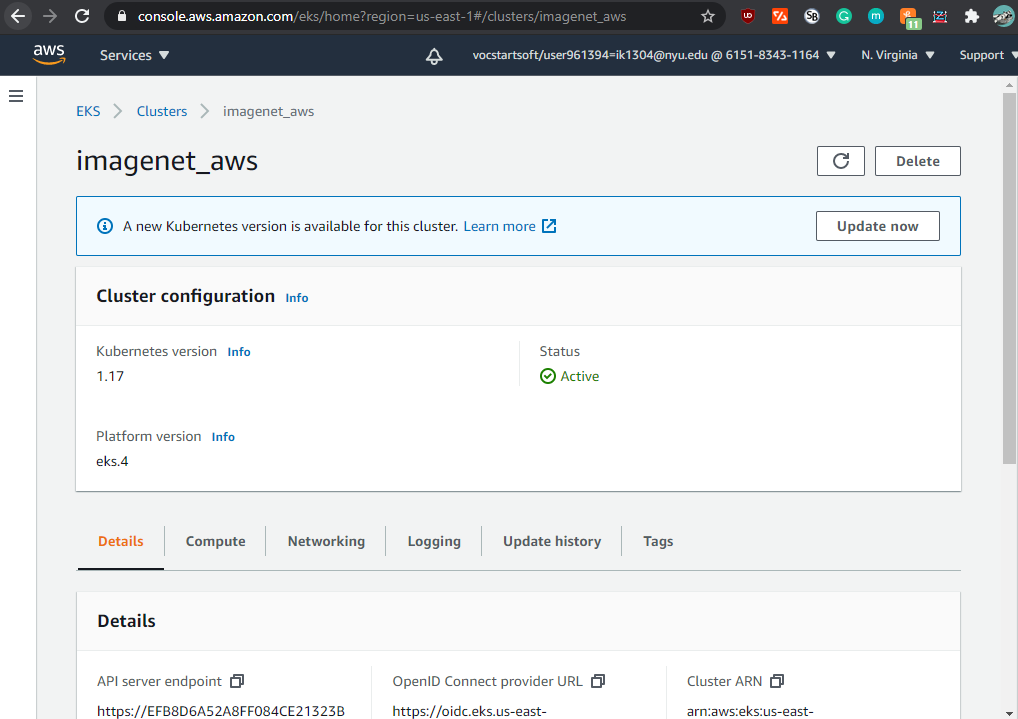
Steps to create Kubernetes Cluster in AWS:

The steps are straight forward with step by step instructions in the following link: <https://docs.aws.amazon.com/eks/latest/userguide/create-cluster.html>

* Create IAM role for EKS



* Go to Amazon ECS to create imagenet\_aws Kubernetes cluster:



**Note:** If you get the following error**: “**Cannot create cluster 'imagenet\_aws' because us-east-1e, the targeted availability zone, does not currently have sufficient capacity to support the cluster.” 🡪 Try changing VPC or create a New VPC in different zone.

|  |  |  |
| --- | --- | --- |
|  | IBM Cloud | AWS EKS |
| Kubernetes Versions | IBM has the largest availability of Kubernetes versions | AWS has lesser availably of versions especially the latest releases |
| Cluster Creation | It is faster compared to EKS | It is slower compared to IBM |
| Time to Provision an application | Again, application provisioning time is faster than EKS | Has slower provision time compared to IBM Cloud |
| Cost (used online prices, as I had student account for both) | Cheaper | More expensive |
| Ease of use and features | Easier to use but lesser features and low granularity | Steep learning curve as you must understand IAM roles, VPC etc., before even creating cluster. Additional knowledge of integrating docker hub images is required |