Deployment of ML Models using Kubeflow on Different Cloud Providers

Aditya Pandey (ap6624), Maitreya Sonawane (mss9240),
Sumit Mamtani (sm9669)

Cloud and Machine Learning (CSCI-GA.3033)

May 19, 2022

1 Abstract

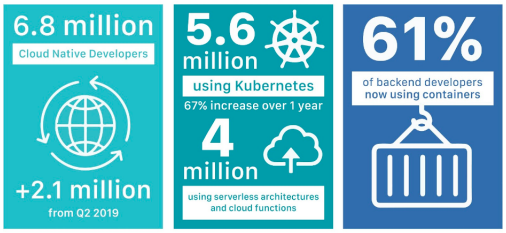
This project aims to explore the process of deploying Machine learning models on Kubernetes using an open-source tool called Kubeflow [1] - an end-to-end ML Stack orchestration toolkit. We create end-to-end Machine Learning models on Kubeflow in the form of pipelines and analyze various points including the ease of setup, deployment models, performance, limitations and features of the tool. We hope that our project acts almost like a seminar/introductory report that can help vanilla cloud/Kubernetes users with zero knowledge on Kubeflow use Kubeflow to deploy ML models. From setup on different clouds to serving our trained model over the internet - we give details and metrics detailing the performance of Kubeflow.

2 Background and Motivation

With the increase in use of Cloud Computing and the emergence of Distributed Systems due to the shear size of data and traffic over the internet, containers have become very popular due to their ease-of-use and scalability properties. A large number of companies have invested heavily in the management and deployments of such containerized applications. Kubernetes is one such open source system for automating deployment, scaling, and management of these containerized applications.

According to a 2021 report in CNCF [2], there are more than 5.6 million users of Kubernetes. Mix that with the crazy increase in companies investing in Machine Learning, the intersection of the two is where Kubeflow fits in perfectly.

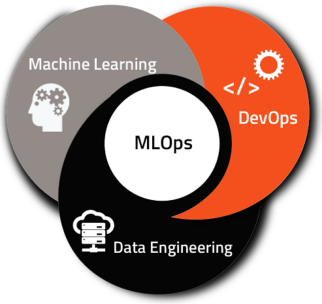
Figure 1: Rapidly increasing popularity of Kubernetes



Kubeflow lies in the intersection of Machine Learning, DevOps and Data Engineering - basically MLOps on the Cloud.

Kubeflow solves a number of pain points of MLOps -
(i) Inefficient tools and infrastructure
(ii) Lack of iterative deployment
(iii) Importance of Automated CI/CD pipelines and
(iv) Handling growth of data and computing power.

Figure 2: What is MLOps



Another important point to consider is the increasing complexity in deploying and maintaining large scale ML systems on the Cloud. While running ML models on developmental setups is straightforward and widely documented, productionizing them is a different ball game altogether.

In the 2015 paper by Sculley et. al [3], the authors talk about how the impression of considering the ease of building complex prediction systems using ML models is a free quick win is incorrect. The cost of being able to build such systems quickly is a large amount of **technical debt** in the form of maintanence costs of these real-world systems as well as several ML specific risk factors that need to be accounted for.

Kubeflow helps us "pay off" this debt to some extent by standardizing and containerizing ML workflows [4].

3 Related Work

In terms of MLOps and generaline pipelining and orchestration platforms, there has been some related work.

Airflow is an open-source workflow management platform for data engineering pipelines. It was initially created by AirBnb [5] to author, schedule and montior their workflows. Essentially, it is a generic task orchestration platform. While it is similar to Kubeflow in many ways, Airflow was not built with Kubernetes in mind and is more useful for a generic use case. In fact, Airflow was initially not intended for ML pipelines at all and usually only performs orchestration and workflow management.

Argo [6] is an open sourcer container-native workflow engine for Kubernetes. It is again more of a general task orchestration problems that runs on Kubernetes natively. A part of Kubeflow is actually built on Argo.

MLFlow [7] is very similar to Kubeflow in the fact that it is a ML-focused workflow and pipeline management tool. MLFlow is supported by Databricks and is not constrained to Kubernetes and runs where the user chooses. It is more of an ML-focused tool and solves experiment tracking and model versioning.

4 MLFlow and Cloud Flavours

4.1 History

TensorFlow [8] library was created by Google as an end-to-end platform for building and deploying ML models. To orchestrate end-to-end TensorFlow Extended pipeline and run TensorFlow jobs on Kubernetes, Kubeflow was developed as an internal project (initially named TensorFlow Extended).

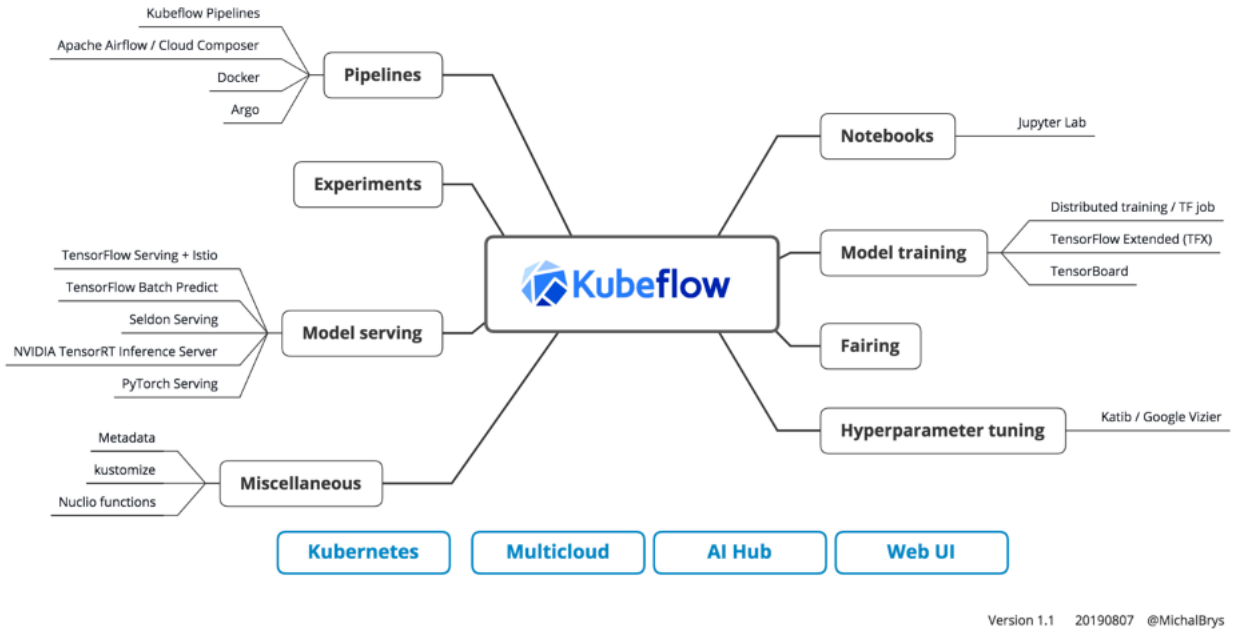
Kubeflow was open sourced at Kubecon in December 2017 and version 1.0 was announced on Febrauary 26, 2020 and since then, all the released versions are available on the its GitHub repository [9]. The idea was to offer ML model orchestration toolkit that can build on Kubernetes.

One big plus point of Kubeflow is that it is not locked into any Cloud Provider. Each provider provides similar services like Amazon SageMaker, IBM Watson, etc. but Kubeflow can abstract that to run on any cloud provider that supports Kubernetes.

4.2 Kubernetes Architecture & Components

Let's look at the main backbone and Kubeflow concepts we used -

Figure 3: Kubeflow Components and Architecture [10]



Pipelines are the backbone of Kubeflow. They are used to build end-to-end ML workflows. The main goal of Kubeflow pipelines is to achieve End-to-end orchestration, enable easy experimentation and enabling easy re-use of components and pipelines to quickly create end-to-end solutions without having to rebuild each time [11].

Notebooks provide a way to run web-based development environments inside your Kubernetes cluster by running them inside Pods. Kubeflow provides native support for JupyterLab, RStudio, and Visual Studio Code.

Hyperparameter Tuning in the form of **Katib** is a Kubernates-native project for automated machine learning (AutoML). Katib supports hyperparameter tuning, early stopping and neural architecture search (NAS) and can tune hyperparameters of many ML frameworks, such as TensorFlow, MXNet, PyTorch, XGBoost, and others. [12]

Model Serving is to host machine-learning models (on the cloud or on premises) and to make their functions available via API so that applications can incorporate AI into their systems. For this, we use **KServe** (formerly KFServing) which is natively supported.

4.3 Baseline Cloud Architectures

To establish a baselines to compare ease-of-use and performance aspects of Kubeflow, we run our training jobs on 2 types of platforms:

1. **NYU Greene Cluster + Server**:
   - This setup involved performing MNIST training on NYU Greene Cluster [13], a HPC cluster to support research across wide range of implementations and disciplines. The cluster supports a range of job types and sizes requiring multiple CPU cores, GPU cards, TBs of memory or even a single core job.
   - It gives a flavour of trainig the Machine Learning code on bare metal and storing the model weights for further use.
   - For inference hosting, we used linserv machine [14] that involves setting up an Apache Web Server. - As this is the most basic setup we had, this is setup exactly lacks what Kubeflow is good for - automated pipelines. As we everything including environment setup/resource requesting needs to be done manually, this is a good baseline to compare the pros and cons of integrating Kubeflow. Also there is no Docker or Kubenetes support, unlike our next baseline.

2. **Basic Kubernetes (on IBM Cloud)**:
   - This setup was replicated from Homework 5, which involved developing Container and Kubernetes artifacts to perform Deep Learning model's training and inference hosted on IBM Kubernetes cluster [15]. To exceute the tasks, we need to create and deploy a Docker image to Docker Hub [16] and then use the same image in our YAML files for creating jobs/containers. Hence, the MNIST model training and inference hosting was performed in the IBM Kubernetes cluster.
   - Unlike the previous setuop, here the environment and resource components are handled by Kubernetes and the setup does support generic containers, but we still needed to execute *kubectl apply -f* command to deploy each YAML file to Kubernetes cluster.
   - What could be better is an end-to-end automated system that manages every stage from data preprocessing and training to inference, exactly what we will try to achieve using Kubeflow.

With these baselines, we had the following two setups for testing Kubeflow:

1. Running MNIST on Kubeflow in Google Cloud (GCP) - Using E2E and Code Approach
2. Running MNIST on Kubeflow in IBM Cloud - Using E2E and Code Approach

4.4 Setup on Google Cloud

While exploring the online documentation on the Kubeflow's official website, there were several possible paths to setup Kubeflow using Google Cloud [17], one of the approach involved creating Kubernetes Cluster on Google Cloud Platform (GCP), creating Notebook instance on GCP's AI Platform which could open JupyterLab instance and then creating a new instance of pipeline on same platform to finally link both so that code in notebook can create a new pipeline using the instance spawned earlier.
We were able to perform model training and testing on using this method, but due to out of date documentation and inadequate online documentation, could not proceed with inference of the model. Below are the screenshots of the procedure described above:

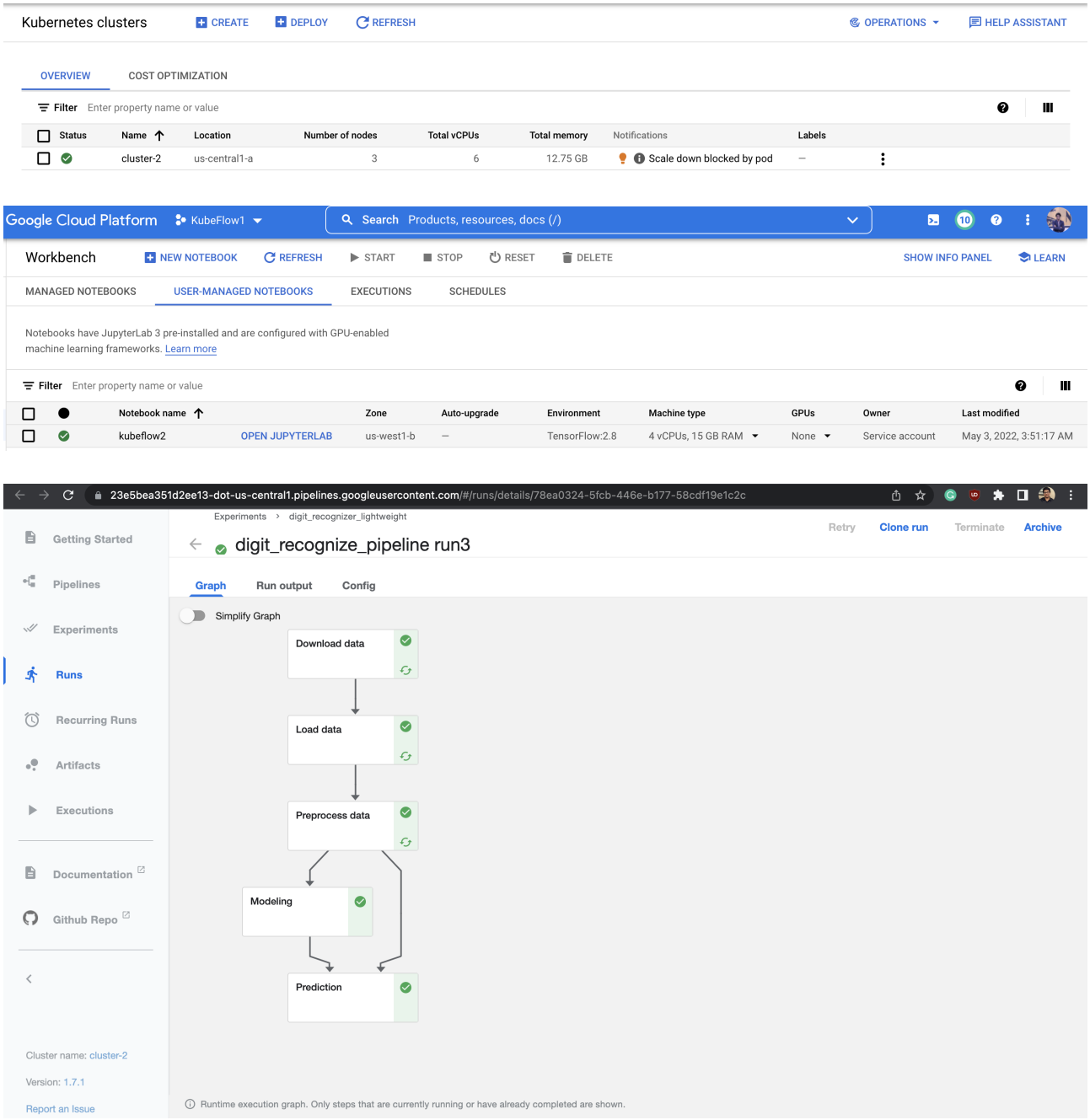


Figure 4: Creating KubeFlow pipeline on GCP - Attempt 1

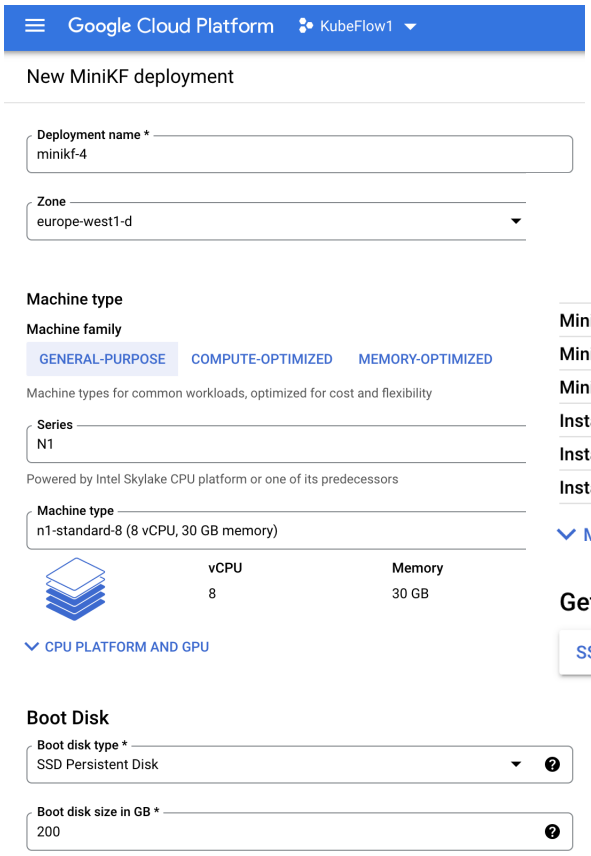
As one can notice in the pipeline image, there is no notebook option on the panel, which makes it difficult to setup the workflow as we need to create a notebook instance separately and then connect them via kfp.Client() - an API Client for Kubeflow Pipelines [18] by providing the pipeline URL. The file - *digit-recognizer-kfp-pipeline.ipynb* contains the code to to be executed on notebook instance to create Kubeflow pipelines on GCP. After facing several errors during inference stage, we decided to switch the methods.

Surprisingly, the most easiest and resourceful way to create Kubeflow pipelines on GCP was not mentioned in most of the documentation - MiniKF - a single node, full fledged KubeFlow deployment.
Basically, MiniKF is to Kubeflow what Minikube is to Kubernetes. It is a single Virtual Machine solution that has capability to install Kubernetes, Kubeflow, Kale, Katib, KFServe, etc, necessary to train and serve your model. And hence, there is no requirement to create a separate Kubernetes cluster, Notebook instance or even a Pipeline instance, as everything will be available in MiniKF itself.
This is the method we used for creating our end-to-end Kubeflow pipeline on GCP. The steps are as follows:

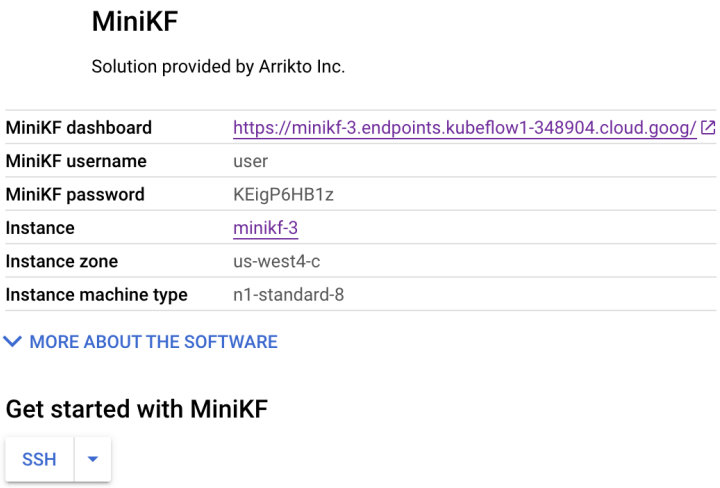
1. Create a project on GCP and ensure billing is enabled and IAM roles include editor privileges. Generally, if you are using your own GCP account and create a project, you would already be 'Owner' and have editor privileges for the project.

2. Using the search bar launch MiniKF from Marketplace. Here you will have to define Virtual Machine resources in 'Configure and Deploy' option. The creators of MiniKF recommend the following settings - 8 vCPU, 30 GB Memory, 200 GB SSD Boot Disk, 500GB SSD Data Disk.
   Although SSD memory is preferred to have, both for Boot and Data Disk, we experienced some problem while trying to create MiniKF instance with given settings (*quota 'ssd_total_gb' exceeded*) and hence, switched to Standard storage for Data Disk, although it is slower than SSD, it is cheaper and was good enough for our experimentation.

3. After the instance is created, a window with an SSH button will appear with message 'Get started with MiniKF'. Once this appears, you can ssh in and type *'minikf'* command and the rest setup will be carried out automatically. At last in the same window, a URL will be visible, were you can access the MiniKF dashboard using the username and password given. And that is it what all had to be done.



(a) VM settings on MiniKF



(b) MiniKF instance ready

Figure 5: Creating MiniKF instance on GCP

A summary of the steps followed can be found here - https://youtu.be/porxGAcWnq8. Visiting the URL for the MiniKF dashboard we see the UI which is very concise and easy to use. The left panel contains several options, one of which is 'Notebook' from where we can spawn a notebook instance directly into MiniKF instance and hence make use of the very functionality that was missing in the previous approaches.

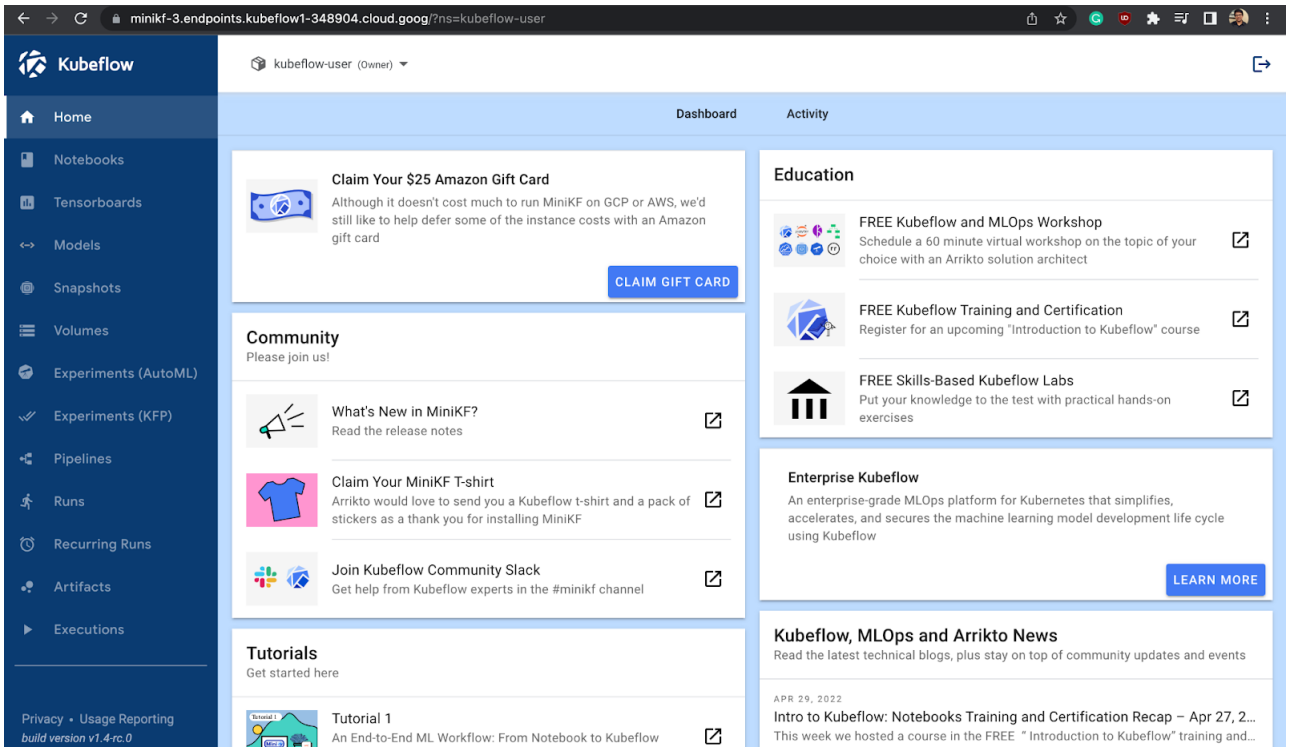


Figure 6: KubeFlow Dashboard on GCP

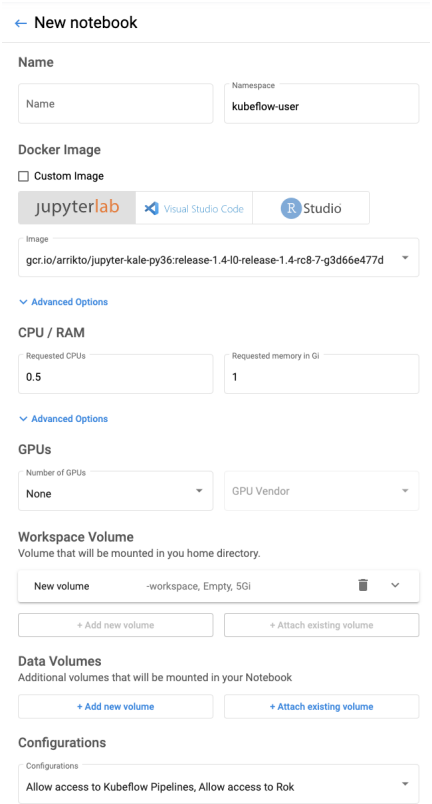


Figure 7: Notebook instance on Kubeflow Dashboard

4.5 Setup on IBM Cloud

To set up Kubeflow on IBM Cloud, there were essentially 2 sources of documentation for the process - (i) on the Kubeflow Docs site [19] (ii) Provided by IBMs development team [20]

There are 2 main configurations of Kubernetes on which Kubeflow can be installed on IBm Cloud -
(i) Classic IBM Cloud Kubernetes cluster
(ii) vpc-gen2 IBM Cloud Kubernetes cluster
Both of these have different setup methods - especially for storage, authentication, network acces, etc.

For this project, we have used a vpc-gen2 IBM Cloud Kubernetes cluster to deploy and setup Kubeflow.

The steps to setup Kubeflow are as follows -

1. The first step is to create our Kubernetes cluster.
   While the step seems straightforward, there are a number of things that we need to keep in mind.
   - We need to make sure that the kubernetes cluster version is compatible with Kubeflow as Kubeflow on IBM Cloud is not compatible with the latest versions.
   - As we are using the VPC approach, we need to make sure that our VPC is setup and we create the cluster in the right location.

2. Next, we need to create the block storage and attach it to our subnet. To create an effective Kubeflow platform, we set up/enable subnets, block storage and routing tables under the same region.

3. Now, we need to install **kustomize** [21] - which is a kubernetes native configuration management tool. Kustomize introduces a template-free way to customize application configuration that simplifies the use of off-the-shelf applications. This is on the same machine as the IBM CLI.
   - Note: Again the versions of kustomize need to be compatible.

4. Once we have the prerequisites, we can apply all the kubeflow configurations from the github link [20] above and our basic Kubeflow setup should be complete. The below three commands together will apply most of the configurations needed to install a basic flavour of Kubeflow.

```
$ git clone https://github.com/IBM/manifests.git
$ cd manifests
$ while ! kustomize build example | kubectl apply -f -;
do echo "Retrying to apply resources"; sleep 10; done
```

IBM Cloud Kubernetes Versions	Kubeflow 1.5.0
1.20	**Compatible**
1.21	**Compatible**
1.22	**Incompatible**

Figure 8: Compatibility of Kubeflow on IBM Cloud with Kubernetes

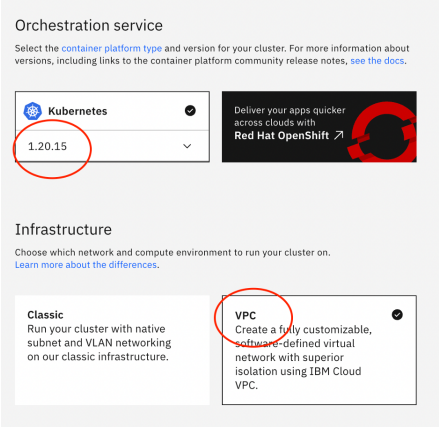


Figure 9: Choosing the Kubernetes version and Flavors correctly

Once we have our basic Kubeflow setup done, we can start playing with the features and access the dashboard.
But to have complete control over the platform, we need to do 2 further things -

1. To view the Kubeflow dashboard, we just need to activate the ingress service using -
   kubectl port-forward svc/istio-ingressgateway -n istio-system 8080:80
   The above command only works on the Kubernetes network. To expose it to the internet, we use -
   kubectl patch svc istio-ingressgateway -n istio-system -p '"spec":"type":"LoadBalancer"'

2. Although we now have the Kubeflow setup, the ingress gateways are only on HTTP. To access Jupyter or any other Notebooks, we need to secure the ingress gateway endpoints with HTTPS.
   For this, we follow the following steps -
   1. We create and setup a DNS for our Load Balanced endpoint
   2. We need to create a secret for the DNS and export it to the istio-ingress service
   3. Enable the kubeflow-gateway to use port 443 using the certificates
   The steps are mentioned in [22] and [23]

We can not spawn notebooks, create pipelines and setup experiments on Kubeflow on our IBM Cloud Kubernetes Cluster.

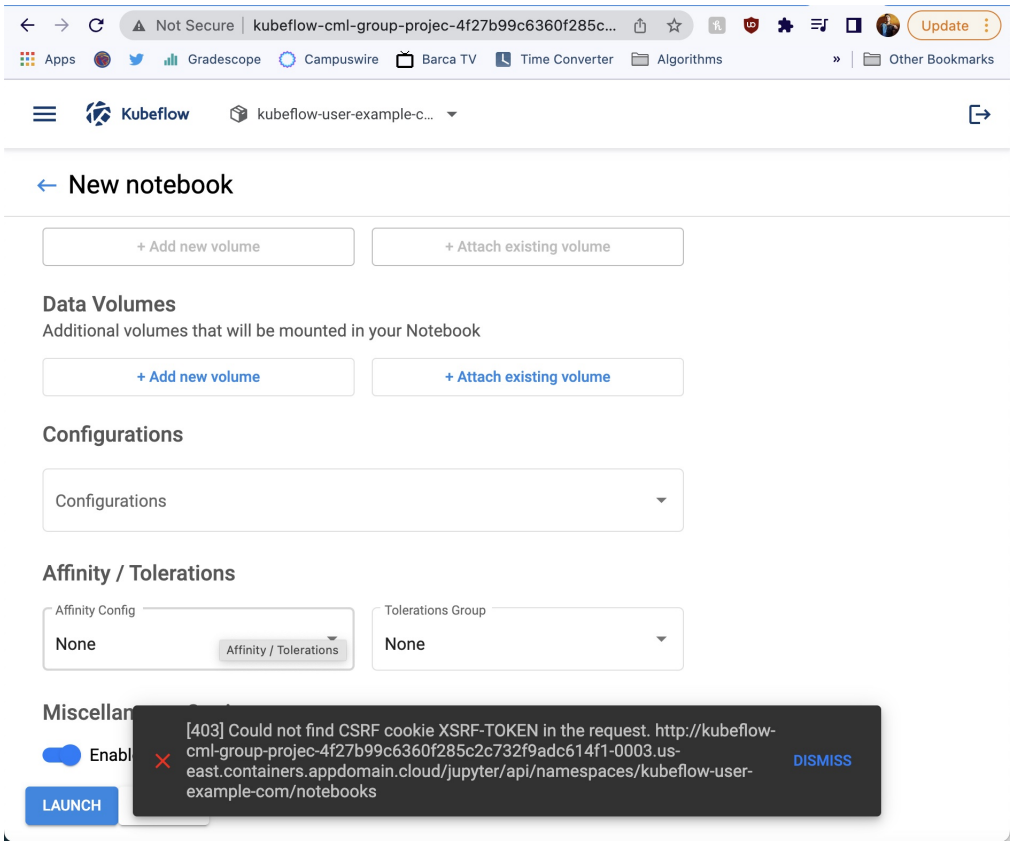


Figure 10: Notebook Creation failing on unsecured gateway

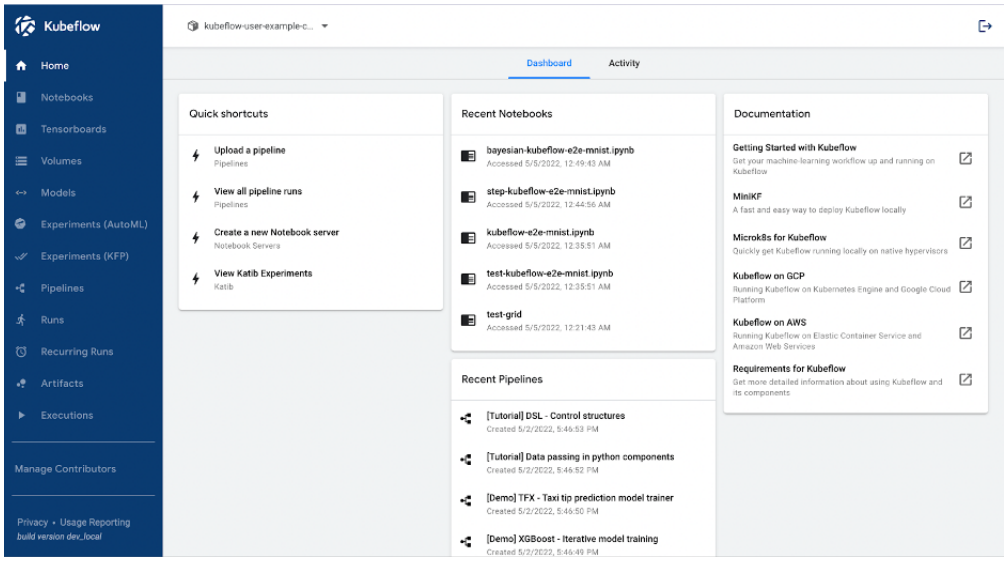


Figure 11: Kubeflow Dashboard on IBM Cloud

5 Experimental Setup

5.1 Dataset

For this project, we were more focused on the different features of Kubeflow and the end to end deployment of the ML Pipeline.

Therefore, we are using the MNIST dataset to experiment heavily with Kubeflow and different flavours of the same.

5.2 Code Approach vs E2E Approach

In terms of our code/pipelines we are creating and running on Kubeflow, we follow 2 main approaches -

1. Running a container image directly via a TFJob (End to End)
   In this, we run a Docker image containing a training code directly on Kubeflow and run a complete pipeline - right from data loading and preprocessing to model serving on KServe. This was on MNIST.
   The goal here is to showcase a complete end to end pipeling creation and execution on Kubeflow (including Hyperparameter tuning and AutoML)

```
},
"spec": {
    "containers": [
        {
            "name": "tensorflow",
            "image": "docker.io/liuhougangxa/tf-estimator-mnist",
            "command": [
                "sh",
                "-c"
            ],
            "args": [
                "python /opt/model.py --tf-export-dir=/mnt/export
            ],
            "volumeMounts": [
                {
                    "mountPath": "/mnt/export",
                    "name": "model-volume"
                }
            ]
        }
    ],
```

Figure 12: Running a direct image on Kubeflow

2. Creating a kubeflow pipeline by writing our own TF code over a base image.
   We also attempt to run a pipeline that contains a Neural network model defined by us by essentially converting our baremetal TF code to lightweight kubeflow component and launching it in a pipeline. This was also a Digit Recogoinzer on MNIST.

```
#initializing the classifier model with its input, hidden and outpu
hidden_dim1=56
hidden_dim2=100
DROPOUT=0.5
model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(filters = hidden_dim1, kernel_size =
                    activation ='relu'),
        tf.keras.layers.Dropout(DROPOUT),
        tf.keras.layers.Conv2D(filters = hidden_dim2, kernel_size =
                    activation ='relu'),
        tf.keras.layers.Dropout(DROPOUT),
        tf.keras.layers.Conv2D(filters = hidden_dim2, kernel_size =
                    activation ='relu'),
        tf.keras.layers.Dropout(DROPOUT),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(10, activation = "softmax")
        ])

model.build(input_shape=(None,28,28,1))
```

Figure 13: Custom Neural Network Model

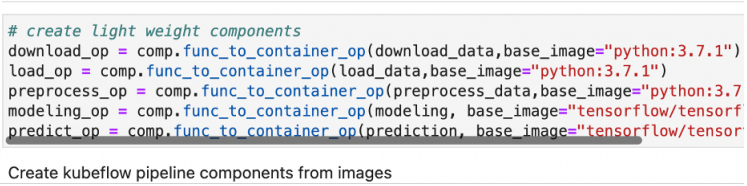


Figure 14: Using lightweight components to create a pipeline

Our goal here was to showcase the fact that users can indeed run their existing Machine Learning model on Kubeflow easily and without much modification while also having the flexibility of using Kubernetes's image pulll feature to train models.

5.3 Pipeline

Let's discuss the modelling of the pipeline that was used in the experimentation.

- Katib's Hyperparameter tuning task: To dive deep into components of Kubeflow, we decided to explore Katib [12] - a Kubernetes-native project that can be used for hyperparameter tuning, early stopping, etc. Here we inspected the Hyperparameter tuning task. Objective was to minimize the loss while training the MNIST model and the goal was to reach 0.001. The Docker image used in the code - *"docker.io/liuhougangxa/tf-estimator-mnist"* uses LeNet - an image classification ML model, to tune the hyperparameters. We decided to use random search over the hyperparamters which will search for the hyper-tuned parameters over a range of learning rate [0.01-0.05] and batch-size [80-100]. This algorithm will randomly choose over the values without replacement and hence will report the combination responsible for lowest loss.

- TFJob Training Task: This is a custom Kubenetes resource [24] that can facilitate running TensorFlow training jobs on Kubernetes instance created by MiniKF. This step will use the best hyperparameters found in Katib's experiment to train same model over which hyperparameters were tuned.

- KServe Inference: This resource [25] is a standard Model Inference platform and can serve ML models on frameworks like TF, PyTorch, etc. This creates a serving component URL that will be used in inference of the model. The biggest advantage to use this asset is its autoscaling and intelligent routing capabilities for load balancing. This step also involves utilizing volume (defined in next step) as PVC, a persitant data storage for our ML model.

At last, after creating a volume to load and store data generated while training and serving our model, we run the Kubeflow Pipeline with end to end MNIST model with hyperparamter tuning, training and inference. The code from the file *gcp_kubeflow-e2e-mnist.ipynb* shows that there is no need of any parameter for *kfp.Client()* (no pipeline URL) as we are running notebook within the MiniKF instance. The run, when complete, is able to generate a single YAML file *'minikf_generated_gcp.yaml'* that will guide the pipeline creation and hence the user can just code naturally to generate pipelines compared to writing a tedious YAML file all by themselves. After running the notebook entirely we see the following outputted pipeline in our runs tab:

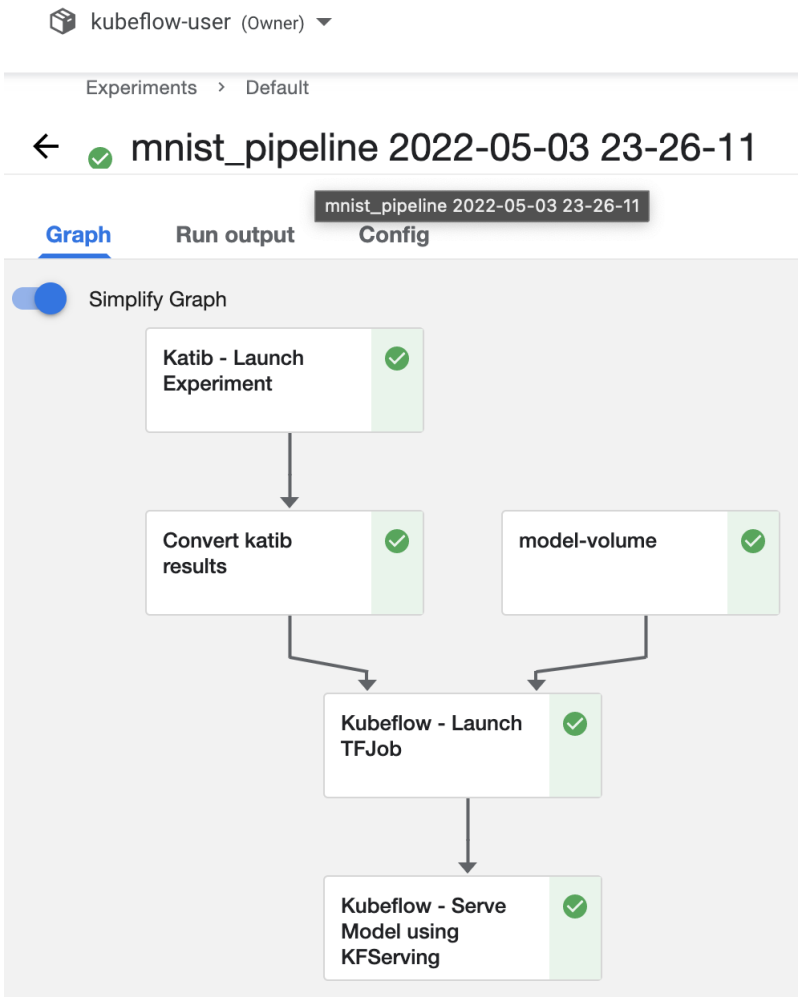


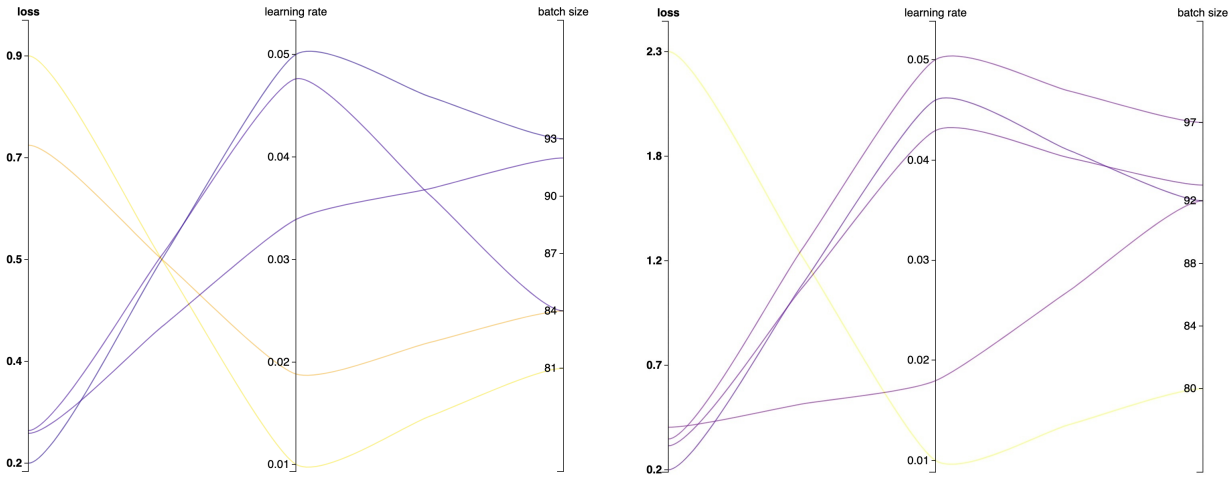
Figure 15: Successful pipeline completed after running the notebook

For tuning, we used AutoML in the form of Katib integration with Kubeflow. We used random, bayesian and step algorithms. Based on all the trials we did, we obtained the following results:

Platform	**Best Trail Performance Loss**	**Tuned Learning Rate**	**Tuned Batch-Size**
IBM	0.1876	0.453	92
GCP	0.2047	0.4980	93

Table 1: Table with hyperparameters tuned via Katib's experiments

Under the 'Experiments (AutoML)' tab, we can access the information about the Katib's hyperparameter tuning as well, and here are the results for the experiments we performed:



(a) Katib on GCP

(b) Katib on IBM Cloud

Figure 16: Katib's Hyperparmeter tuning on various Clouds

5.4 Model Serving

For model serving, we are using KServe to create our endpoints here. Kubeflow makes it extremely easy to create an enpoint and we can just pass our test values to the API endpoint using a CuRL request and get the result (as seen below).

```
maitreya@10-16-62-118 dataset % curl -X POST -F image=@zero.jpg 'http://52.118.148.144:5000/predict'
{"result": 0, "status": "Prediction Complete"}%
maitreya@10-16-62-118 dataset %
```

Figure 17: Serving through CuRL request

This can also be done programatically using any request module -

```
image_url = "https://i.imgur.com/6qsCz2W.png"
image = Image.open(requests.get(image_url, stream=True).raw)
data = np.array(image.convert('L').resize((28, 28))).astype(np.float).reshape(-1, 28, 28, 1)
data_formatted = np.array2string(data, separator=",", formatter={"float": lambda x: "%.1f" % x})
json_request = '{{ "instances" : {} }}'.format(data_formatted)

url = "http://mnist-e2e-4-predictor-default.kubeflow-user.svc.cluster.local/v1/models/mnist-e2e-4:predict"
response = requests.post(url, data=json_request)
print("Prediction for the image")
print(response.json())
display(image)
```

```
Prediction for the image
{'predictions': [{'predictions': [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0], 'classes': 7}]}
```


Figure 18: Serving through Python

For the experiment running on linserv, we created a basic Flask-based app which loads a saved PyTorch model and makes a prediction and hosted it on the NYU Server.

To showcase how an end-to-end ML based application might function, we also created a simple UI. The deployment for the UI Component is also very similar to any Kubernetes app deployment.
We create a simple static HTML page that lets users upload a photo and makes API requests to the inferencing API + displays the result.

We then dockerize the html file and host is using **nginx** in a very lightweight form.

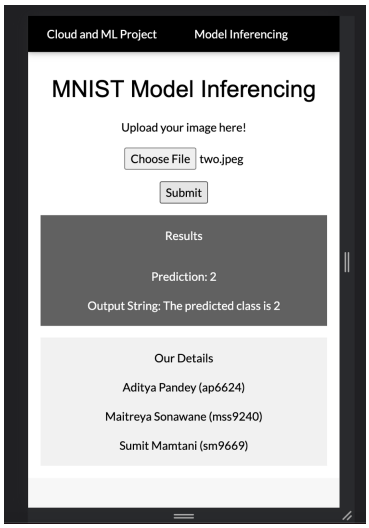


Figure 19: Simple Webpage making API requests

Once we have this docker image, we can now deploy our app to Kubernetes. We use a simple Pod to do this.
Now the last step is to expose this to the internet and we use a loadbalancer for this (similar to the previous part). Note that as there is only one pod, using a LoadBalancing Service is overkill but it provides a nice accessible endpoint for our webpage.

5.5 Kubeflow Addons Used

Let's discuss the additional tools that we integrated with a Kubeflow deployment.

- Istio:
  - Istio is an open source framework used by Kubeflow to enable end-to-end authentication and access control. It is a highly performant open-source implementation of service mesh used to describe the network of microservices and interactions between them. Kubeflow is a set of tools, frameworks, and services that work together to create machine learning workflows. These workflows are created by combining multiple services and components. Kubeflow provides the infrastructure that makes it possible to put these components together.
  - Kubeflow uses Istio for Securing service-to-service communication in a Kubeflow deployment with strong identity-based authentication and authorization. Its other requirements can include failure recovery, metrics, monitoring, and traces for traffic within the deployment including cluster ingress and egress.

- KServe:
  - KServe formerly known as KFServing provides an inferencing service on Kubernetes and it also provides performant, high abstraction interfaces for common machine learning (ML) frameworks like TensorFlow, scikit-learn, XGBoost, and PyTorch to solve production model serving use cases. It supports advanced deployments with canary rollout, experiments, ensembles and transformers.

- KServe also supports a modern serverless inference workload with autoscaling, networking, health checking, and server configuration including advanced serving features like GPU autoscaling, scale to zero, and canary rollouts to ML deployments.

6 Results & Comparisons

6.1 Katib Hyperparameter Tuning

We performed experiments on Katib, it is a Kubernetes-native project used for hyperparameter tuning, early stopping and neural architecture search (NAS). Experiments were done on Katib to find the optimum values of hyperparameters using three techniques - Grid Search, Random Search and Bayesian optimization search.

- Grid Search: Grid Search exhaustively searches this space in a sequential manner and trains a model for every possible combination of hyperparameter values. Grid search is not very often used in practice because the number of models to train grows exponentially as the number of hyperparameters is increased. This is very inefficient in time. We can also see from the Fig. 20 as we increase the number of runs the grid search is taking most time compared to Random and Bayesian optimization search.

- Random Search: The key difference between random and grid search is that in a random search, not all the values are tested and values tested are selected at random. The advantage of randomized search is that we can extend our search limits without increasing the number of iterations (time-consuming) as we can see from the Fig. 20. And the main point is that we can also use it to find narrow limits to continue a thorough search in a smaller region.

- Bayesian optimization Search: Bayesian optimization is a sequential model-based optimization technique that uses the results from the previous iteration to decide the next hyperparameter values of the model. Bayesian optimization search is efficient because they select hyperparameters in an informed manner. When the model complexity is less Bayesian Optimization search takes less iteration to get to the global optima while Random and Grid search might take a high amount of iterations to get there. In our case, We have a somewhat complex model to train so that's why random search is taking less time than Bayesian and Grid Search.

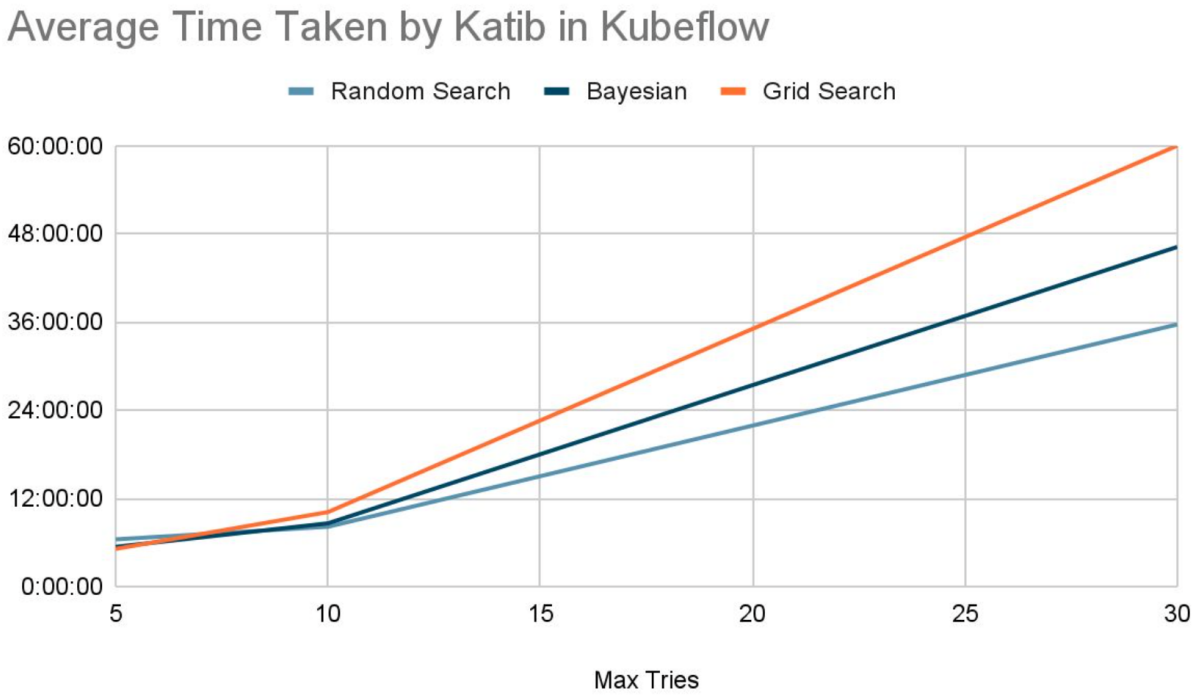


Figure 20: Average Time taken by Katib in Kubeflow for three Hyperparameter Tuning Algorithms

Max Tries	Random Search	Bayesian Search	Grid Search
5	6:25:00	5:24:00	5:07:00
10	8:08:00	8:36:00	10:08:00
15	35:40:00	46:13:00	60:00:00

Table 2: This table shows the average time taken by Katib in Kubeflow for three hyperparameter tuning algorithms for different number of tries

6.2 Comparison of Inference Time across all platforms

We performed four different Experiments here.
Each experiment consisted of a model training instance and then an inference service hosted on the cloud platform.

1. We ran MNIST code on NYU Greene Cluster and deployed the model on Linserv for inference.

2. Running a Basic MNIST Image on Kubernetes (on IBM Cloud) and then hosting a simple API with a LoadBalancing service

3. Running MNIST on Kubeflow in IBM Cloud - Served using KServe

4. Running MNIST on Kubeflow in Google Cloud (GCP) - Served using KServe

To test the inference performance of each approach, we performed a sort of stress-test where we repeatedly sent predict requests consisting of one test image to the serving endpoint on the same network and noted the total time taken to respond to each request. Here are our results while comparing inference times on different platforms:

No of Req	w/o KF baremetal	w/o KF K8 cluster	w KF GCP	w KF IBM
1	0.2415	0.1955	0.1194	0.0603
4	2.8616	0.6382	0.3362	0.1974
8	5.9860	1.4011	0.7692	0.4774
16	10.0755	2.4590	1.4794	0.7617
32	16.4755	5.9566	3.1525	1.4677
64	27.7447	14.8050	6.4079	2.6950
100	44.0205	20.6780	9.4831	4.1400
128	64.4582	24.9734	12.1261	5.2376
256	104.3657	54.2752	24.3457	10.6503
512	178.9776	114.6778	47.9780	22.6391

Table 3: Table of time required for inference on different platforms mentioned in subsection 4.3

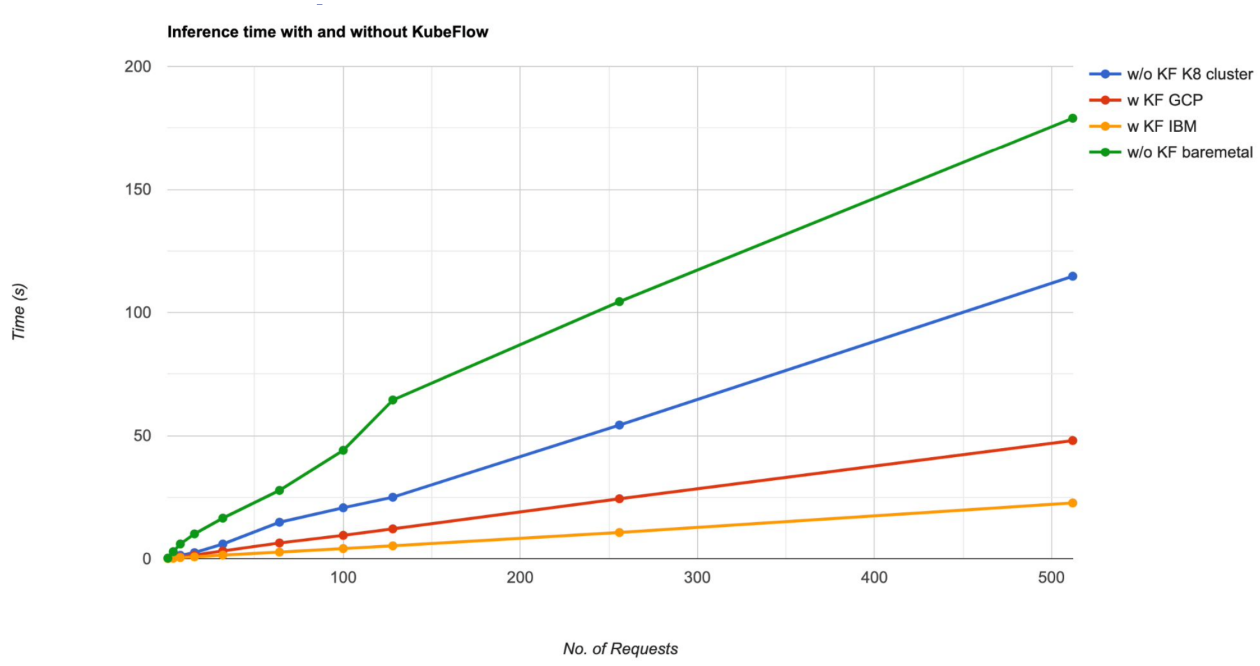


Figure 21: Inference Time Comparison with and without Kubeflow

We can see here that both the experiments run using the Kubeflow approach perform much better on the Stress test further showing us that Kubeflow is a great choice for model serving.

6.3 Comparing the performance of Kubeflow across clouds

Our next experiment was to compare the performances of Kubeflow on the 2 different cloud providers - IBM Cloud and Google Cloud Platform.
We tested the running time of our 2 approaches -
(i) our custom model running a Digit recognizer

	Kubeflow on GCP	Kubeflow on IBM Cloud
Total Pipeline Time	745	908
Model Running Time	429	583

Table 4: We compute the time required to run Kubeflow models on GCP vs IBM Cloud

(ii) our E2E pipeline with Katib and Model Serving.

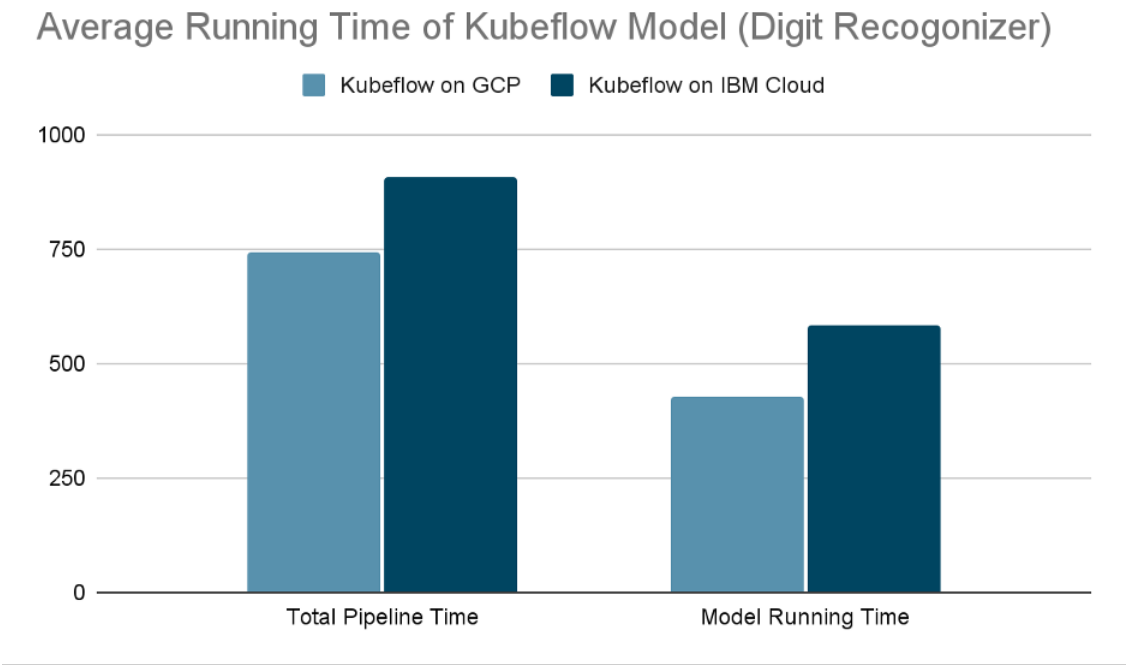


Figure 22: Running Time Comparison of our Custom Model

Time in sec	GCP	IBM
Total Time	547	685
Katib Experiment	338	385
TFJob	126	108
Model Serving	43	111

Table 5: Here we analyse the time required in every step for running Kubeflow E2E pipeline on GCP vs IBM Cloud

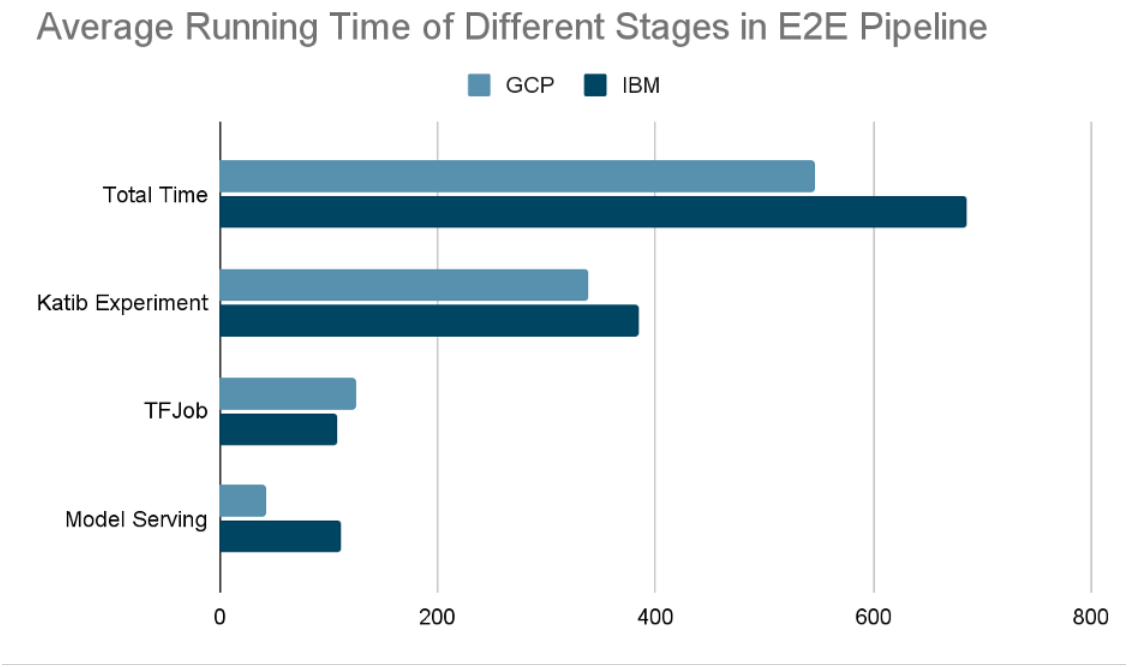


Figure 23: Running Time Comparison of different stages in the E2E Pipeline

We see that in both these approaches, Kubeflow on Google Cloud performs a bit faster than IBM Cloud on average.

7 Insights

Based on the results above, we came up with the following insights -

1. Based on the sort of stress test on the inference endpoint for both clouds, we noticed that the MNIST inferentce model running on the Kubeflow on IBM cloud has the least inference time among all the models.
   One reason we think that IBM Cloud has a lower inference time is because all our K8s components on IBM Cloud are defined inside the same VPC in the same region. As we are making the network calls from the same network, a dedicated VPC would give us better network performance.

   The inference model API hosted on linserv is the slowest. This makes sense as the API is hosted on the public linserv server where we do not have any loadbalancing service and we are loading a new PyTorch model everytime a new request comes in.

   The API model hosted on Kubernetes makes use of loadbalancing features but still suffers from the same problem as above.

2. The Duration for running E2E pipeline is less for Kubeflow on GCP
   We feel that the total duration for running the pipeline on GCP is lower as the cluster is more powerful and the contention of resources is lower.
   While we tried to make the flavours of base compute the same for both clouds, the MiniKF setup on GCP is more idealistic and does the steup with minimal fuss.

Another point here could be the fact that Kubeflow is a Google-based product due to which it's performance of GCP is slightly better.

3. The overall process of creating a cluster and using Kubeflow on it was easier on GCP for a few reasons - easier availability of documentation, automatic HTTPS endpoint securing, easy access to KF pipelines from Notebooks, etc.

   While IBM cloud has all these same features (as Kubeflow is platform agnostic), it is more challenging to enable/find resources or documentaion for the same.

4. While Kubeflow has a lot of advantages, especially during Model training and inferencing, there are a few pitfalls that we feel prevent a more widespread adoption of the framework.

   The difficulty with the initial installation and authentication setup makes it a pain oint to start development with Kubeflow.
   As Kubeflow uses different versions of different components (such as Istio), upgrading individual components is a risky task.
   In addition, the presence of out of date documentation + Broken Links is a big challenge (as we talk about below).

8 Challenges

Among the various challenges we faced during project, the most prominent ones were related to inadequate and outdated documentation of Kubeflow. The website https://www.kubeflow.org is mainly a documentation website that provides instructions on setting up Kubeflow and using it to achieve various tasks. Here are the major challenges we faced:

1. **Deploying Kubeflow using UI**
   There were so many resources that guide on setup and configuration of Kubeflow which start with the first step of Deploying Kubeflow using UI for example [26]. Here the problem is Deployment using UI is no longer supported for Kubeflow https://www.kubeflow.org/docs/distributions/gke/deploy/deploy-ui. The resources might be helpful enough to create Kubeflow environment but if the first step itself is broken, the resource is of no use. One should also note the inconsistency in the documentation as this website - https://v0-6.kubeflow.org/docs/gke/deploy/deploy-ui about Kubeflow v0.6 actually mentions that deployment can happen with URL, but takes us to a broken link when trying to open the documented URLs.

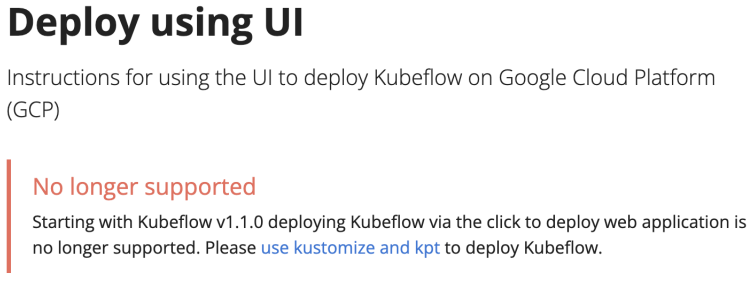


Figure 24: Deployment using UI no longer supported

2. **Component Specifications**
The Kubeflow Pipeline requires container component specification to describe data model. This will be ultimately serialized to a file in YAML format for sharing, similar to what can be found in the attached file *'minikf_generated_gcp'*. As one can imagine, the step is one of the most important in the pipeline building phase, and yet the documentation if Out of Date - https://www.kubeflow.org/docs/components/pipelines/reference/component-spec.

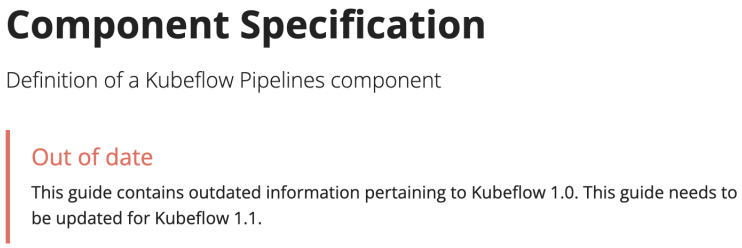


Figure 25: Component Specifications Outdated

3. **Official Documentation obsolete**
The model used in our experimentation for training, testing and inference was a ML model for MNIST dataset. The official repository of Kubeflow does provide an example of notebook to setup and run the same model. The problem - the documentation is outdated and the notebook is full of errors, possibly cause lots of component used are either removed or changed - https://github.com/kubeflow/examples/tree/master/pytorch_mnist. Remember, Kubeflow has been open sourced just in recent years. There have been several version rolled out in very short duration of time, for example, Kubeflow 1.0 in February 2020, Kubeflow 1.1 in June 2020, Kubeflow 1.2 in November 2020 and Kubeflow 1.3 in April 2021. Currently we are on version 1.5, and yet the documentation for the notebook is not updated since July of 2019.

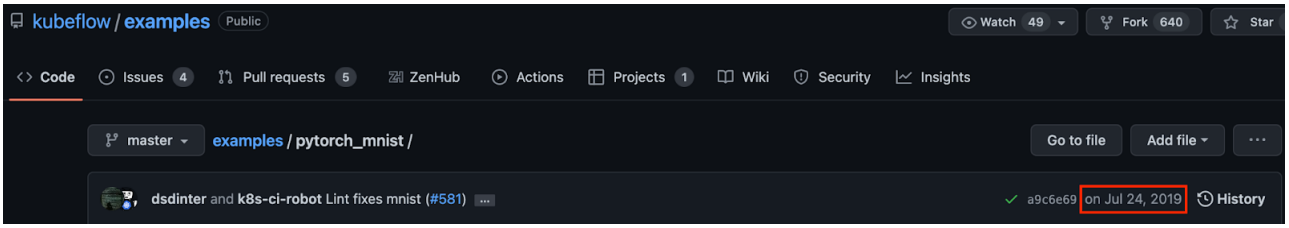


Figure 26: MNIST notebook obsolete

4. **Challenge of Setting up Kubeflow on IBM Cloud**
Setting up Kubeflow on IBM Cloud has a number of additional challenges than the above. The biggest one is the compatibility issues of installing Kubeflow on IKS. As the latest versions of Kubernetes and Kustomize are not supported, using existing clusters is a pain point as there is no clear way to downgrade the Kubernetes version.

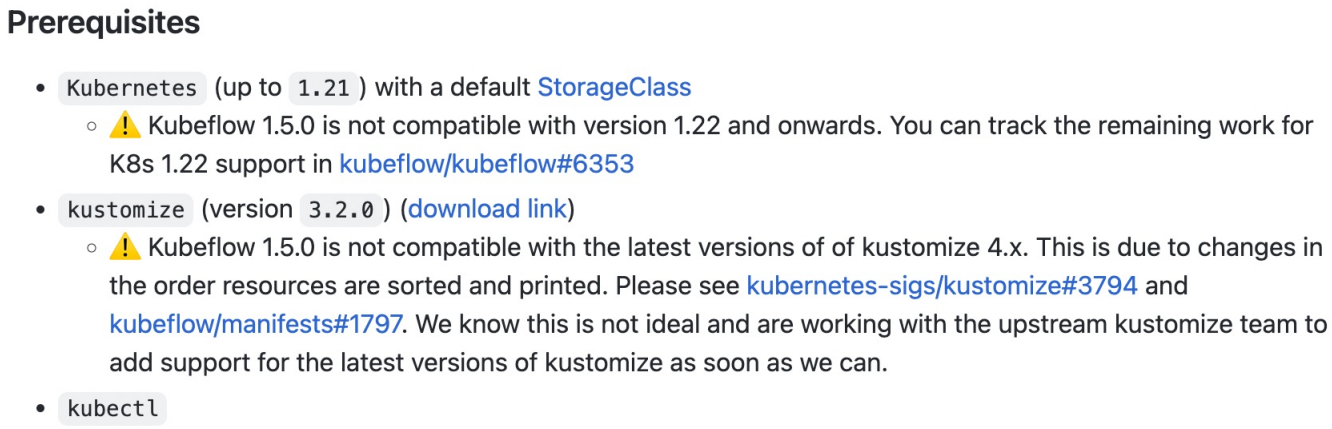


Figure 27: Lack of Support for latest versions

In addition, there are a number of broken links and the fact that there are not many public users of IBM Cloud definitely reduces the number of resources/support pages available to debug issues on Kubeflow on IBM Cloud.

9 Conclusion and Discussion

In this project, we presented a deep dive into integrating Kubeflow on both IBM Cloud and GCP, while comparing them with similar models deployed on K8s cluster and also models trained on NYU HPC without Kubeflow.
- We found that while duration of E2E run for Kubeflow on GCP was the least, IBM Cloud surpassed every other model to give fastest inference time. These results can be explained in the slight differences in setup and architectures of these 2 cloud providers.
- Kubeflow is a great tool and platform for users and developers to test and productionize end to end Machine Learning solutions. the ease of creating a pipeline - right from Data Ingestion/Preprocessing to Parameter tuning to Model serving makes it a valuable tool for any organizing wanting to deploy an ML solution.
- In addition, the presence of Kubernetes as a base makes use of all the advantages that Kubernetes provides in terms of Scalability and Orchestration.
But all this comes with a lot of effort needed to setup Kubeflow and replying on different services such as IStio, KServe, etc. that do have integration issues with the framework. - We must also note that While Kubeflow is great for running ML Jobs on Kubernetes; environments such as HPCs and Big Data Systems have their own flavours of MLOps

We strongly believe that with better documentation and community support, Kubeflow has the right tools to be a successful framework.

10 Future Work

We believe that there are many interesting avenues that can be explored with Kubeflow.
- For this project, we only explored CPU based execution of ML models due to limitations wth basic accounts on GCP and IBM Cloud. With more access (and credits), we could explore the integration of Kubeflow with GPU enabled Clusters and Notebooks.

- With GPU addition, we could also increase the depth of the neural network to see even more improvement over bare metal and increase number of trials in terms of Katib hyper-parameter tuning with different model architectures.
- One point that was mentioned was how Kubeflow was an extension of TensorFlow. But Kubeflow also supports PyTorch and other frameworks and this compatibility could be explored.
- We can also try an ML problem from a different domain (such as Speech/Text) and check if the advantages offered to us here also hold for those domains.
- Finally, we encourage anyone interested in this framework to try their hand at becoming an open-source contributor - https://v1-5-branch.kubeflow.org/docs/about/contributing/

References

[1] Kubeflow. https://www.kubeflow.org/docs/distributions/.

[2] New SlashData report: 5.6 million developers use Kubernetes, an increase of 67% over one year. https://www.cncf.io/blog/2021/12/20/new-slashdata-report-5-6-million-developers-use-kubernetes-an-increase-of-67-over-one-year/.

[3] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, page 2503–2511, Cambridge, MA, USA, 2015. MIT Press.

[4] Machine Learning Technical Debt: Using Kubeflow to Pay It Off Quickly. https://www.arrikto.com/blog/machine-learning-technical-debt-using-kubeflow-to-pay-it-off-quickly/.

[5] Airflow: a workflow management platform - AirBnB Tech Blog. https://medium.com/airbnb-engineering/airflow-a-workflow-management-platform-46318b977fd8.

[6] Introducing Argo — A Container-Native Workflow Engine for Kubernetes. https://blog.argoproj.io/introducing-argo-a-container-native-workflow-engine-for-kubernetes-55c0b4b76fac.

[7] MLFlow. https://mlflow.org/docs/latest/index.html.

[8] TensorFlow. https://www.tensorflow.org.

[9] Kubeflow - GitHub repository. https://github.com/kubeflow/kubeflow.

[10] Kubeflow Architecture and Components. https://medium.com/@michal.brys/kubeflow-a-machine-learning-toolkit-for-kubernetes-d8686f6c91b6.

[11] What is Kubeflow Pipelines? . https://www.kubeflow.org/docs/components/pipelines/introduction/.

[12] Introduction to Katib. https://www.kubeflow.org/docs/components/katib/overview/.

[13] Getting Started on Greene. https://sites.google.com/nyu.edu/nyu-hpc/hpc-systems/greene/getting-started.

[14] The Linux Servers. https://cims.nyu.edu/webapps/content/systems/resources/computeservers/linserv.

[15] Slide 65 and 66 of lecture 9 slides - Cloud and Machine Learning (CSCI-GA.3033) (Spring 22).

[16] Docker Hub. https://hub.docker.com/.

[17] Kubeflow on Google Cloud. https://www.kubeflow.org/docs/distributions/gke/.

[18] kfp.Client class. https://kubeflow-pipelines.readthedocs.io/en/latest/source/kfp.client.html.

[19] What is Kubeflow Pipelines? . https://www.kubeflow.org/docs/distributions/ibm/.

[20] What is Kubeflow Pipelines? . https://github.com/IBM/manifests.

[21] Kustomize. https://kustomize.io/.

[22] Enabling HTTPS om Kubeflow - IBM Cloud. https://www.civo.com/learn/get-up-and-running-with-kubeflow-on-civo-kubernetes#step-4-enable-https-to-access-kubeflow.

[23] IBM Cloud Authentication. https://www.kubeflow.org/docs/distributions/ibm/deploy/authentication/.

[24] TensorFlow Training (TFJob). https://www.kubeflow.org/docs/components/training/tftraining/.

[25] KServe. https://github.com/kserve/kserve.

[26] Kubeflow Examples - Named Entity Recognition. https://github.com/kubeflow/examples/blob/master/named_entity_recognition/documentation/step-1-setup.md.