Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Swati Choudhary

Kubernetes-Based Architecture For An On-premises Machine Learning Platform

Master's Thesis
Espoo, September 3, 2021

Supervisor:	Professor Juho Rousu, Aalto University
Advisor:	Neelabh Kashyap, D.Sc (Tech.), CGI Suomi Oy

Aalto University
School of Science

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author:	Swati Choudhary		
Title: Kubernetes-Based Architecture For An On-premises Machine Learning Platform			
Date:	September 3, 2021	**Pages:**	105
Major:	Machine Learning, Data Science and Artificial Intelligence	**Code:**	SCI3044
Supervisor:	Professor Juho Rousu		
Advisor:	Neelabh Kashyap, D.Sc (Tech.), CGI Suomi Oy		

This thesis introduces an architecture where a machine learning platform can be set up on an unmanaged Kubernetes cluster in an on-premises environment, using Kubeflow and Kubeflow Pipelines. The proposed architecture allows the solution to be platform-agnostic and avoid vendor lock-in. The goal for providing an on-premises architecture is to be able to have control over the security of the platform and conform the machine learning service to custom Service Level Agreements (SLAs). While designing the architecture, open-source technologies are used which allow for review of the source code to identify potential vulnerabilities and weaknesses of the platform. This is especially vital in use-cases revolving around critical security and confidentiality. As part of experiments, it is shown how a machine learning model can be trained using Kubeflow Pipelines and deploy the model using Kubernetes in order to serve real-time user requests for prediction. Finally, the complexity of the proposed solution versus the existing solution is compared both with respect to the development of the solution as well as migration of the solution to other vendors.

Keywords: kubernetes, kubeflow, kubeflow pipelines, docker, flask, container registry, model training, model serving

Language: English

Acknowledgements

The journey to my Master's thesis has been no less than that of a roller coaster ride. Every moment of this journey was unique and memorable on its own. This however would not be possible without the constant support and assistance I have received.

I would like to thank my supervisor, *Professor Juho Rousu*, for his constant guidance and for providing me with the right direction for my thesis.

I would also like to thank my manager *Ville Suvanto* and my advisor *Neelabh Kashyap* for giving me an opportunity to be part of the *CGI Advanced Analytics Solutions* team and allowing me to work flexibly and freely on this research project. My master thesis journey became easier because of their constant guidance, motivation, and support whenever I needed it.

I would also like to acknowledge my parents, friends, colleagues, and my adorable cat, Nietos, who constantly reminded me to take breaks during my master thesis journey and play with him. I could not have completed this thesis without the support of my husband *Mayank Khandelwal*. He was always cheering me and motivating me to not give up and keep working hard till I achieve the desired results.

Finally, I would like to thank myself for doing all the hard work, taking no days off, working 7 days a week and being so dedicated.

Espoo, September 3, 2021

Swati Choudhary

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation & Scope

Machine learning operations (MLOps) [89] aims to integrate machine learning with *DevOps*. It comprises a set of best practices for developing machine learning solutions. It includes components of a machine learning life-cycle such as the development of the model, testing the predictions generated by the model, deploying a model for consumption, and maintaining the performance of the model. By using *MLOps*, automation and monitoring of the machine learning life-cycle becomes structured and allows for collaboration within teams.

Major cloud providers such as *Microsoft Azure*, *Amazon Web Services*, and *Google Cloud Platform* have support for a machine learning service based on MLOps. These services, however, require the use of cloud resources and do not provide advantages offered by an on-premises setup, such as custom security for sensitive data [34]. Using a machine learning platform that is built on-premises provides enterprises with full control over their application life-cycle and allows them to use their own set of customizations.

The primary advantage of using machine learning services offered by cloud providers is that they abstract the underlying platform setup and allow developers to focus on building the machine learning solution. To leverage a public cloud-like level of abstraction of the underlying platform on an on-premise environment for ML applications, a standardized architecture is needed. This can abstract the underlying infrastructure while retaining functionality in line with *MLOps* guidelines.

Having an on-premises machine learning solution has other advantages which enterprises may wish to leverage. Most solutions provided by cloud providers are not open-source and thus lack the functionality to accommodate customizations suitable for business needs. Using closed-source services enforces a vendor lock-in which makes migration to any other provider either infeasible or impractical. Applications built using open-source technologies can be packaged and deployed across infrastructure services offered by cloud providers. Open-source technologies provide full source code access to the developer, allowing identification of potential security issues & vulnerabilities, and understanding how the technology has been built. Such a platform is particularly useful in the banking and military sectors as well as with enterprises that have critical use-cases requiring confidentiality. In such an industry, a developer needs to be aware of potential issues arising as part of using the technology.

1.2 Contribution

The goal of this thesis is to present a standardized architecture for a machine learning platform that provides the functionality required to develop a portable and end-to-end machine learning solution in production.

The contribution of this thesis lies in three areas. The first contribution is to provide a solution to be able to deploy a machine learning workflow in an on-premises environment. A workflow, in this context, is defined as an integration of development stages when developing a machine learning solution, such as data pre-processing, training, inferencing, and deployment. This workflow when packaged for a business use-case is defined as a machine learning solution. The second contribution is to provide a solution that abstracts the underlying infrastructure the workflow is running on. The proposed solution uses open-source technologies, which due to access to the code-base, allows modifications in the solution which can be used to incorporate customizations. The third contribution focuses on portability, where the developed solution can be migrated to any platform such as a private cloud, hybrid cloud, public cloud or an on-premises environment. Portability is possible as part of the proposed solution because it abstracts the underlying infrastructure through the use of *(Docker) containerization* [67] and *Kubernetes* [74].

Through the contribution in these three areas, the objective is to be able to develop a machine learning solution, focused on an on-premises environment.

Kubernetes is chosen as the tool for deploying a machine learning solution as it provides portability, as Kubernetes can be run on any hardware setup provided that it can be installed. *Kubeflow* is used as the tool for abstracting the technicalities of the Kubernetes setup and enabling developers to focus on the machine learning solution. *Kubeflow pipelines* help modularize the different aspects of the solution by packaging different components of the solution. Packaging of components is done through the use of Docker which allows the components to be used and re-used on any platform. This setup allows the solution to be portable and compatible in a platform-agnostic way.

Due to the platform-agnostic nature of the solution, it can be deployed on an on-premises environment where an enterprise can leverage the advantages of having an on-premises setup while incorporating guidelines offered by *MLOps*.

1.3 Structure

Chapter 1 covers the scope of the thesis, listing the challenges of choosing and opting for a machine learning platform, a potential solution, and the goal of the thesis. Chapter 2 describes the need for such a platform and its associated advantages. This chapter also introduces the proposed technology stack and compares the proposed solution against existing solutions. Chapter 3 explains the conceptual setup of the proposed solution as well as the internal working of the components being used. Chapter 4 explains the procedure to setup the services on top of which the machine learning platform can be used. Chapter 5 covers the implementation aspect of the machine learning platform. The evaluation of the platform, related discussion, and future work is described in Chapter 6 and the conclusion is offered in Chapter 7. Scripts developed during experimentation can be found in the Appendix.

Chapter 2

Background

This chapter explains the need for on-premises solutions, the advantages of using a machine learning platform, MLOps, and its relation with DevOps, and the benefits of using a Kubernetes setup. The need for a Kubernetes-based machine learning platform based on the MLOps concept having the ability to run on an on-premises environment is explained. The technology stack used for setting up the platform and experimentation is explained. This chapter is concluded with alternate existing solutions, their advantages & dis-advantages; and how they stack up against the proposed setup.

2.1 Need for On-Premise Solutions

Using an on-premises solution means that everything is done internally within the enterprise, which includes an end-to-end implementation of the solution, including maintenance, security, and updates. Both the hardware and software are owned by the enterprise. Complete ownership of the solution is assumed when using on-premises solutions.

The major factors [9] that enterprises keep in mind while evaluating whether to use a public cloud solution or not are:

- **Security**: Enterprises that deal with sensitive data or user-privacy data, would not opt to have their data hosted in a public cloud for security reasons. Using an on-premises solution gives them complete control over the security of the data. For example, banking data would be highly secure data. While very uncommon, unauthorized data access between virtual devices hosted on the same physical server may be possible [1].

- **Network**: If a service can run in isolation or in conjunction with services within a private network, then an on-premises solution can be considered. Due to non-exposure to the internet, the probability of attacks such as *Denial of Service* is minimal.
- **Compliance**: Enterprises need to follow regulatory controls. To meet government-defined regulations, enterprises need to be compliant and have their data in order.
- **Deployment**: As both hardware and software are owned by the enterprise, it is the enterprise's responsibility to build and maintain the end-to-end solution. This requires to having deployment done on an in-house infrastructure setup.
- **Costs**: Enterprises can procure their own hardware or use hardware resources allocated for other purposes on a sharing basis, which can potentially be more economical than opting for a public cloud provider in order to minimize operational cost.
- **Control**: When using a cloud platform, a vendor lock-in may be implied when developing on top of or using the services offered by the provider. Not all software and services offered by the cloud provider are open-source and cannot be subject to change and customization. An on-premises solution allows enterprises to have complete control over what, how, and when software need to be set up.

2.2 Characteristics of a Machine Learning Platform

Machine learning (ML) platforms provide an environment for developing, deploying, monitoring, and maintaining a machine learning solution [11]. Through easy-to-use SDKs, ML Platforms provide support for common ML tasks such as data loading, transformation, pre-processing, and optimization. Some components such as data loading, and transformation can run autonomously, and some components such as hyper-parameter updating or training due to model drift [95] or concept drift, may require human intervention.

As the amount, quality, and characteristics of the data vary in an enterprise, there is a need to iterate over the ML model creation process which may include changes to the way data is pre-processed, transformed or trained.

As the business requirements vary, the deployment aspect, and the service design of the ML solution may vary. In order to track, organize, manage, maintain, and scale ML solutions, a ML platform should allow a mechanism to achieve that, and provide insights on the end-to-end solution.

A typical ML platform provides the following functionality [50]:

- **Data Ingestion**: This refers to the ability to be able to migrate data from one data source to another data source where it can be accessed and used.
- **Data Transformation & Cleaning**: On top of raw data, operations such as wrangling, joins (integration), normalization, standardization, pre-processing, balancing, filling missing values, augmentation, re-structuring, and re-formatting are performed. This allows the raw data to be transformed in a way that can be readily used for analysis. Some machine learning platforms such as *Azure Machine Learning* provide recommendations for transformations based on the dataset.
- **Feature Engineering**: *Feature Engineering* is the task of improving the performance of predictive modeling for a dataset by transforming the feature space [66]. In practice, this means the features are extracted from the raw data and transformed into formats that can then be used to train a machine learning model. *Feature Engineering* helps simplify the modeling process with ready-to-use features and results into a model which generalizes well over the dataset [100]. Machine learning platforms such as *Azure Machine Learning* perform basic methods such as scaling and normalization when using automated machine learning services with the option of enabling more methods.
- **Model Training**: Training a machine learning model means providing a machine learning algorithm with training data from which it can learn the characteristics of the training dataset. The resulting model artifact which is created during the training process is called a machine learning model [4]. A machine learning platform can provide a way to perform model training either through an interactive user interface or via the command line. The training can be conducted as part of experiments and machine learning platforms may offer to version the different pieces of training performed within or across experiments.
- **Experiment Tracking**: Experiment tracking is a core feature of a machine learning platform. It provides the ability to store the history

of model training runs and helps to track the improvements of the trained model over time.

- **Model Evaluation**: Model evaluation helps understand the performance of the model from a perspective. Model evaluations are conducted with respect to metrics and based on the metric score the performance of a model can be judged. Model evaluation techniques also help judge the robustness and stability of the machine learning model against unseen data. Machine learning platforms may provide the ability to compute multiple metrics through an interactive visual medium.
- **Model Deployment or Model Serving**: Model deployment or model serving refers to the act of exposing a machine learning process for it to be consumed by other services or by end-users to make business decisions. Once a model is deployed, user data can be run against the deployed model to generate predictions. Machine learning platforms typically provide model deployment functionality to host a model in a compute resource in two variants: real-time deployments where an API is exposed to generate predictions in real-time; or batch deployments where based on a trigger or schedule, an entire dataset is run against the model to generate predictions.
- **Model Monitoring & Tracking**: Model monitoring and tracking refers to the tracking of the predictions generated by the machine learning model based on the input data. Identification of potential issues such as data quality changes can be tracked before the business use-cases are impacted. While not considered as a core component of a machine learning platform, ancillary features are available which can be used alongside the machine learning setup, or a custom monitoring solution can be integrated.
- **Model Registry**: A *model registry* is a repository where trained models can be stored and be provided with model versioning. A model registry stores meta-data about the model and the content differs depending on the machine learning platform.

2.3 MLOps & Relation to DevOps

DevOps is a term formed by combining *Development* and *Operations*. *DevOps* consists of a set of practices which enables automation and integration between teams [31]. It allows different teams within an organization to develop, evaluate and release solutions through principles such as planning, building, continuous integration, and continuous delivery (CI/CD). Continuous integration consists of merging the application code to a centralized repository, whereas continuous delivery is the rollout of the automated software deployment.

MLOps is built on top of the concepts of *DevOps* and meant in the context of machine learning solutions [89]. *MLOps* uses *DevOps* principles and applies them alongside the functionality of a Machine learning platform. *MLOps* aims to reduce the time-to-deployment for a machine learning solution while retaining functionality to create and maintain the solution.

Both *DevOps* and *MLOps* aim at improving collaboration between team members who perform different tasks of a solution. Both focus on automation and the use of CI/CD. While *MLOps* is more focused on data checks and model drifts, *DevOps* focuses on getting the application ready. *MLOps* also includes the concept of continuous training of a model which is not present in *DevOps*.

In real-world applications, the data on which a ML model is trained changes constantly which means that the models need to be retrained to handle data drift or concept drift. This may also mean rebuilding the entire ML pipeline in cases where the end business use-case has evolved. The evolution of a machine learning platform manually leads to high operational costs as well as an increase in time-to-deployment. MLOps aims to solve both these problems through its guidelines revolving around automation.

2.4 Kubernetes Functionality

Kubernetes is an open-source system that allows deployment automation, scaling of the service, management, and orchestration of containerized applications [47]. Containers running within a cluster need to adhere to the pre-defined environment resource constraints and configuration available to the cluster, such as compute and storage. Containers that are not in the desired state must have an action performed on them to bring them to the desired state (for example: through restarting). This is handled by Kubernetes [81].

Kubernetes offers several advantages including (automated) bin-packing, load balancing, service discovery, storage orchestration, self-healing, secret management, configuration management; and automatic rollbacks & rollouts.

Kubernetes and its advantages are described in further detail in Section 2.6.4 and 3.1.

2.5 Features of a Kubernetes-based on-premises ML platform

A Kubernetes-based on-premises machine learning platform is a platform that is deployed on an on-premises environment through the use of a Kubernetes cluster. The proposed solution follows the principles offered by *MLOps* [101]. Such a platform allows leveraging the features provided by Kubernetes, the abstractions provided by using a machine learning platform, and the benefits arising from an on-premises setup.

Such a platform also allows customization and review of source code to identify vulnerabilities from a platform security perspective, because of the open-source tools proposed for the experiment. Adoption of this platform allows configuring its own level of security, compliance, and accesses for both users and services without the mandate of external network access. Such a setup can be migrated to a cloud provider, if needed, and can be deployed on cloud providers such as *Microsoft Azure*, *Amazon Web Services (AWS)* and *Google Cloud Platform (GCP)* due to the setup's platform-agnostic nature. When migrating to a cloud provider, managed Kubernetes services such as *Azure Kubernetes Services* by *Microsoft Azure*, *Elastic Container Service for Kubernetes* by *AWS* and *Google Kubernetes Engine* by *GCP*, can be used to deploy the machine learning platform on these services.

In order to create a machine learning platform for an on-premises environment, this thesis provides a way to build either customized containers from scratch or containers with the help of open-source extensions dedicated to providing functionality for the features as described in Section 2.2. *Containers* in-turn allow portability of the platform where it can be deployed on any environment supporting *Docker*. [75]. The proposed solution, however, does not have a model registry functionality because it is under development. An external model registry is used instead. In order to abstract the management of the Kubernetes cluster configuration, the proposed solution uses *Kubeflow*, allowing the developers to focus on the machine learning related tasks.

2.6 Technology Stack

2.6.1 Docker

Docker is an open-sourced platform that allows building, shipping, and running distributed applications using the Docker engine. A Docker engine is a portable and lightweight run-time packaging tool. Through the use of Docker, applications can be tested, run and deployed on any environment with Docker support available. The advantage of using Docker is that it combines a lightweight container virtualization platform with a workflow which reduces the time to deployment from a development perspective. Docker uses resource isolation features that allow multiple independent containers to run in a single node. Due to isolation, an application can securely run inside a Docker container with the resources allocated to it [75].

2.6.2 Flask

Flask is a web micro-framework based on the Python language. A micro-framework does not require specific libraries, packages, or tools without the inclusion of ancillary features or validation. *Flask* provides a library package that can typically be used to build websites without worrying about the abstraction of serving the web application. Typically, *Flask* uses another library called *Werkzeug* which is a web server gateway interface (WSGI) library, which takes care of processing and responding to API requests sent by the end-user [63]. The way a Flask application is designed violates the object-oriented programming principle of segregating the visual components (example: data display through HTML render) and the program logic of the application [8].

2.6.3 Container Registry

A container registry, in the context of Docker, is a versioned registry that allows compute environment to access the images and instantiate containers in the compute environment [82], eliminating the need to manually copy and update multiple iterations of images that would otherwise lead to inefficiencies, human-errors and place limitations on scalability. A container registry acts as a shared location where compute environments can access and pull images. These images can be stored in the compute environment's local image cache [15].

2.6.4 Kubernetes

The inner working, components, and terminologies of Kubernetes are described in detail in Section 3.1.

Kubernetes is an open-source cluster manager for managing and maintaining *Docker* containers. Kubernetes decouples the containers on which the application is running from the technicalities of the servers or the machines the containers are running on. Containers part of the Kubernetes cluster provide an abstraction, making every container its own self-contained unit for computation. A group of containers is termed as a *Pod.* Every pod can run one or more *Docker* containers, which are able to utilize services such as the file system, input and output operations associated with a pod [57]. Each pod is assigned a unique logical address which is accessible from other *Pods* within the cluster irrespective of their physical location [12]. One or more pods can run on one node and several nodes constitute a Kubernetes cluster.

There are **six** main advantages of using *Kubernetes* [83].

1. **(Automated) Bin Packing** [61] means that *Kubernetes* can automatically package an application and schedule running the containers based on how it is configured. The containers are assigned resources such as memory and disk, and then placed in the cluster depending on the resource requirement without hindrance to the availability of the service.

2. **Load Balancing and Service Discovery** [90]: Load balancing provides a way of ensuring maximum availability of resources for tasks to run on and provisioning scalability functionality. Network traffic is distributed amongst different compute resources depending on characteristics such as availability of the resource and resource requirement. Service discovery provides a mechanism for Pods to communicate to each other within a cluster and a way to actually connect to the desired service.

3. **Storage Orchestration** which enables automatic mounting of cloud or local storage system to a container. Kubernetes abstracts the storage layer from the technicalities of the storage resource. A Container Storage Interface (CSI) is a specification through which container orchestration tools can interact with storage solutions. A Kubernetes

storage class allows the configuration of pre-defined types of storage mounts that tasks can request and utilize. A persistent volume is a storage volume attached to the Kubernetes cluster [19].

4. **Self Healing** which means that if a node of a cluster is unresponsive or killed, a copy of the node can be redeployed [91].

5. **Secret and Configuration Management** allows adding, modifying, and removing secrets without exposing secrets in the stack configuration and without the need of rebuilding the Docker image [83]. In Kubernetes, in order to secure sensitive information, a *Secret* object can be used [17].

6. **Automatic Rollbacks and Rollouts** can help check the status of a rollout, rollback to a previous deployment, scale up deployment, and pause a deployment [46].

Docker Swarm is seen as a potential alternative to *Kubernetes*. There are some differences in terms of functionality and usability, with some of the critical differences listed here. Kubernetes provides a graphical user interface [92] providing a more intuitive way to interact with the platform. Kubernetes provides native support for auto-scaling, unlike *Docker Swarm*. Auto-scaling provides a way of provisioning additional or de-allocating already provisioned resources, such as compute and storage, depending on the resources required to run a task. *Docker Swarm* is unable to perform roll-backs and must be done manually. Here, auto-scaling means either providing more resources or de-allocating resources based on the task. Kubernetes provides a native way of performing logging and monitoring [65]. In Kubernetes, however, storage volumes are restricted to the Docker container in the same *Pod*, unlike *Docker Swarm* where the storage volume can be shared across containers [83].

2.6.5 Kubeflow

Kubeflow is a platform that aims at simplifying the deployment of machine learning workflows on a Kubernetes cluster setup through the concept of distributed machine learning deployment. A distributed machine learning deployment constitutes components such as training, inferencing, scoring, and logging as part of the deployment pipeline. With the use of Kubeflow, technicalities of managing a Kubernetes cluster are abstracted [14].

Kubeflow provides a graphical user interface for intuitive interaction with the platform such as selection of desired resources and container images.

Kubeflow is still under active development at the time of writing and is far from a complete solution. For example: As part of *Kubeflow*, a machine learning workload [28] is considered as part of a workspace. A machine learning workload, in this context, is defined as a task that needs to be performed as part of the machine learning workflow. When a user makes a workspace for *Docker* image, a pod dedicated to the workspace is created which is managed by Kubernetes even though the pod performs no tasks, leading to inefficiencies with the pod management system. Another disadvantage is that a GPU un-mount has to be done manually even if the workload is finished. If there are more workloads than the number of worker nodes, then finished workloads need to be deleted manually [39].

While there are shortcomings, Kubeflow is constantly adding new functionality and improvements. Due to the open-source nature of the project, it allows changes in the source code enabling possibilities for customization and identification of vulnerabilities to prevent attacks against the technology being used. Using an open-source tool also allows usage of the tool without the need for subscription or licensing fees.

2.6.6 Kubeflow Pipelines

Kubeflow Pipelines is a component of Kubeflow which is responsible for managing and orchestrating machine learning workflows in a Kubernetes/Kubeflow setup. A typical machine learning workflow involves steps such as data wrangling, training, inferencing, monitoring, and logging. These steps are wrapped as components and packaged as *Docker* images for re-usability. The configuration of these components such as dependencies, input, and output relationships are defined in YAML configurational files which can be processed by Kubeflow [101]. Pipeline files are packaged into a single file and can be uploaded to *Kubeflow* using the user interface. Kubeflow pipelines can be triggered using an API call, or on a cron-based schedule.

2.7 Existing Solutions

This subsection describes the existing solutions available in the market, their functionalities/features, benefits, and shortcomings. While there are lots of different machine learning platforms, in this section we cover the most widely used (based on the number of users) machine learning platforms [73].

2.7.1 MLflow

MLflow is an open source platform for a machine learning life-cycle which is ML library agnostic as well as programming language agnostic [23]. The *MLflow* API allows integration of *MLOps* principles into machine learning projects [3]. This platform consists of four components.

- **MLflow Tracking**: This is an API used for recording experiment runs, the scripts being used, the parameters used to train the model, the input/output files, and metrics. This component allows logging and querying the experiment runs via user interface or via API (Python, REST, R and Java) [40].
- **MLflow Models**: This component provides a generic model packaging format that would include scripts and (data) dependencies. This can then be deployed to multiple environments such as Docker or Azure Machine Learning for serving Spark [7]. Through this component, it is possible to manage and deploy machine learning models through the user interface [87].
- **MLflow Projects**: This component allows code organization set with conventions describing the code. Code can be versioned and stored [87]. Reusable software components, modules, and environments can be created through the definition of YAML files [84].
- **MLflow Model Registry**: This component enables the model registry functionality. Model registry assists in model versioning and allows the definition of what stage a model is in [3].

MLflow allows abstraction of different components of a machine learning development setup such as training, inferencing, and deployment enabling the developer to focus on developing the machine learning solution without worrying about the underlying technicalities of deployment. It allows tracking of experiments which is useful in comparing results with other configurational runs and training a model based on learnings from previous outcomes. Reusability is an important functionality of *MLflow* which reduces redundancies across multiple environments. With the help of *MLflow*, it is possible to productionize the machine learning model in various ways such as real-time serving and batch predictions [99].

However, *MLflow* does not allow dataset versioning which would allow a machine learning developer to incorporate the dataset information associated with each model. It also does not offer a standardized framework-agnostic way of capturing model telemetry as part of model monitoring after deployment. Model telemetry is defined as the data describing the operational statistics of the model deployment such as usage and time to respond to user requests. At the time of writing, *MLflow* lacks the functionality of defining multiple sub-tasks and multi-step workflows. This functionality is particularly useful in cases where there are dependencies between different steps. For example, an output of one step is to be fed to the next step [23]. While orchestration is possible within *MLflow*, an external tool such as Apache Airflow can be used where the steps are represented as DAG (Directed Acyclic Graph) workflows [51].

2.7.2 Azure Machine Learning

Azure Machine Learning (Azure ML) is a cloud-based environment that allows training, deploying, automation, managing, and tracking a machine learning model in order to make predictions against the model [10]. This service supports multiple programming languages such as *R* and *Python.* This service also has a *Jupyter Notebook* integration and supports open-source libraries such as *PyTorch* and *TensorFlow*, *MLflow* and *Kubeflow.*

The primary advantage of using this service is that it has in-built integration into other services offered by *Microsoft Azure* such as *Azure Blob Storage* for data storage, *Azure Container Registry* for model tracking/versioning and *Azure DevOps* for source control [88]. This allows an end-to-end setup of a machine learning project within the *Microsoft Azure* environment. It also has a user interface through which machine learning models can be created using drag and drop with the need for limited programming skills. This service also provides a way of automatically building a machine learning model which requires a cleaned dataset and the type of machine learning use-case (classification/regression). It then automatically performs experiments and selects an algorithm along with hyper-parameters suited for the use case. *Azure ML* also includes MLOps capabilities which allow scalable creation and deployments of ML models with reproducible machine learning workflows. *Azure Machine Learning Pipelines* offers a way to create machine learning workflows with reusable steps. Trained models which are deployed can be run on Azure's own variant of Kubernetes called, *Azure Kubernetes Service (AKS).*

However, at the time of writing, support for running machine models written in the *R* language is limited with insufficient documentation available. Using features such as automatically building a machine learning model for production-level machine learning models is still pre-mature and should be used with extra caution. There is no native support available in order to mount external storage resources and migrating away from this service results in rebuilding the entire machine learning workflow.

2.7.3 Amazon SageMaker

Amazon SageMaker is a service offered by *Amazon Web Services* which helps implement and productize machine learning models. It offers Amazon's implementation of machine learning models along with support for open source models [2]. In-built machine learning models offered by this service can be used for training the models along with support for custom models packed in a *Docker* environment.

While *Amazon SageMaker* consists of a multitude of components, the primary components as part of the *Amazon SageMaker* service are as follows:

1. **SageMaker Studio**: This tool is an integrated development environment that allows step-based machine learning development through a web-based user interface [38]. The user interface allows uploading data, development via *Jupyter* notebooks, training machine learning models, hyper-parameter tuning, deploying models at production scale, and a way to visualize experiments and their results.

2. **SageMaker Autopilot**: This tool is an automatic machine learning solution that requires a tabular dataset (features) and a target column (label). *SageMaker Autopilot* then identifies the problem type, performs data analysis, auto-creates machine learning pipelines, and presents candidate machine learning models. The auto-created generated machine learning pipelines can further be customized [27].

3. **SageMaker Experiments**: This tool allows organizing, tracking, comparing, and evaluation of machine learning experiments. This tool also provides a model versioning service to track the developed machine learning models [5].

4. **SageMaker Model Monitor**: This is a solution that allows monitoring of machine learning models deployed in production. For example: alarms can be triggered to notify the data science team when shifts in data distributions are detected [13].

While different machine learning model versions can be deployed at the same end-point, re-usability of components or steps themselves in a machine learning workflow can not be intuitively used and manual intervention is required to create multiple environments. While constant functionality is constantly being added to this service, the level of abstraction with the underlying tech-stack is relatively lower than *Kubeflow*. Migration away from this service requires a rebuilding of the entire machine learning workflow. This can be mitigated by using SageMaker components for Kubeflow Pipelines [6].

2.7.4 Vertex AI

Vertex AI is a very recent managed machine learning platform, at the time of writing, and is offered by *Google*. Currently, there is minimal documentation and technical resources available due to the recent introduction of the service. This service aims at bringing automated machine learning and an API platform into a unified API, library, and user interface. It is possible to leverage the automated machine learning functionality as well as custom model training and save the model, deploy in production and perform real-time predictions [37]. This service offers the use of creating pipelines where each task of the machine learning workflow is identified as a step. Due to its recent introduction, this service is not described in further detail due to lack of extensive documentation and support materials, nor is it used for evaluation and comparison in Section 2.8.

2.7.5 Polyaxon

Polyaxon is an open-source enterprise-level machine learning platform for agile, reproducible, and scalable machine learning. The idea of *Polyaxon* revolves around deploying machine learning solutions on-premises and on top of Kubernetes. It allows development, training, and monitoring of machine learning applications [33].

Polyaxon has native integration of the *Kubeflow* components and can leverage *Kubeflow* operators such as *TFJob* (*TensorFlow* training jobs), PytorchJob (*PyTorch* training jobs) and MPIJob (Message Passing Interface for parallel computing). There are four components which *Polyaxon* consists of [72].

- **Core**: It is a set of tools for interaction with the platform; querying, filtering & sorting information; scheduling, and submission of operations.

- **Experimentation**: Allows data-preprocessing, training models, running notebooks & distributed jobs, launching the UI, tracking, logging and result visualization.
- **Automation**: This component allows the creation of DAGs in the workflow, performs hyper-parameter tuning, running of tasks and pipelines in parallel either as a trigger or on a schedule.
- **Management**: This component allows collaboration & sharing features with Role-Based Access Control (RBAC) functionality, running visualizations/media-renders, versioning, and a model registry for tracking machine learning models.

However *Polyaxon* lacks functionality for model serving or deployment and an external tool or service is needed to develop an end-to-end machine learning solution.

2.7.6 CNVRG.io

CNVRG.io is a *Kubernetes* based platform based on MLOps which provides a compute agnostic platform where data science tasks and workflow can be orchestrated and managed at scale. This service is based on *Docker* containers which allow integration of these containers into existing workflows. There are five primary features for *CNVRG.io* [24].

- **Management**: This feature allows collaboration between and within teams within an environment.
- **Tracking**: *Tracking* allows storing, managing, and tracking both models and machine learning experiments. It also allows real-time visualization and visual model tracking. The model repository allows storing of models and meta-data.
- **Deploy**: It allows deployment of machine learning models with the service having the capability of auto-scaling in order to optimize operational compute costs based on the workloads. This feature has monitoring capabilities and can track the status of deployments in real-time.
- **Automation using Flows**: This features provides a visual drag-and-drop way of creating machine learning pipelines that can be shared and reused in other workflows. It also consists of in-built hyper-parameter search capabilities.

- **Data Repository**: *Data Repository* allows data access through a central repository where datasets are tagged, versioned, and can be queried. This repository is data structure-agnostic and can be imported or integrated into any experiment.
- **Resource Management**: This allows visualization of the technical stack to analyze and optimize server utilization. It allows controlling and allocating resources to containers and cloud-native architecture in real-time.

However, to fully leverage the capabilities of this service, a paid subscription is needed. Without a subscription, there are limitations on functionality including parallelization, support for hybrid & multi-cloud, external object storage integration, and Spark with YARN.

2.8 Evaluation Of Existing Services

Summarization of the existing services described in Section 2.7 is shown in Table 2.1.

Experiment tracking is implemented differently across different services but provides support for tracking and versioning experiments. At the time of writing, the model registry component is under development for the setup proposed; and thus an external model registry service is used for experimentation. All services provide their version of developing machine learning pipelines, however, the difference lies when the setup is to be migrated to a different service. For example: AWS SageMaker pipelines would need significant rework to be able to migrate to Azure Machine Learning Pipelines.

Vendor lock-in refers to a lock-in where development is forced to continue with a particular service because migrating to a different service provider is either not practical or feasible. Deployment type refers to either a cloud deployment or an on-premise (or private) deployment. Machine learning platforms offered by cloud providers are only meant to run on the cloud, and not on an on-premise installation.

All services offer language support in multiple languages. *Polyaxon* does not offer capabilities to serve a model (for example: real-time predictions). Pipeline components in general are re-usable in all services, but with limitations on the extent and degree of re-usability. For example, with *Azure*

Machine Learning, a component needed in a separate pipeline needs a re-definition of the component.

CNVRG.io uniquely identifies its position in the market through the introduction of features such as a data repository for versioning of datasets. Another unique feature distinguishing this service from others is its ability to create machine learning pipelines using a visual interface.

Table 2.1: Comparison of Existing Services

SOLUTION	**Kubernetes + Kubeflow**	**MLFlow**	**Azure Machine Learning**	**Amazon SageMaker**	**Vertex AI**	**Polyaxon**	**CNVRG.io**
Experiment Tracking	**Yes**	Yes	Yes	Yes	Partial (Beta)	Yes	Yes
Model Registry	**Developing**	Yes	Yes	Yes	Developing	Yes	Yes
Pipeline Support	**Yes**	Yes	Yes	Yes	Yes	Yes	Yes
Vendor Lock-in	**No**	No	Yes	Yes	Yes	No	Partial
Deployment Type	**Any**	Any	Cloud	Cloud	Cloud	Any	Any
Client Library Language Support	**Python, Java, R and CLI**	Python, Java, R and CLI	Python, R and CLI	Python, R and CLI	Python, Java and Node.js	Python, R and CLI	Python, R
Model Serving	**Yes**	Yes	Yes	Yes	Yes (Beta)	No	Yes
Reusability of Pipeline Components	**Yes**	Yes	Semi-reusable	Semi-reusable	Semi-reusable	Yes	Yes

Chapter 3

Kubernetes & Kubeflow Pipelines

The goal of this chapter is to explain the working of Kubernetes and Kubeflow Pipelines [43]. First, the working of a Kubernetes cluster is explained along with the components associated with the cluster. Next, the Kubeflow Pipelines working is described and how the developer interacts with its *UI* which in turn interfaces with the Kubernetes cluster. Then the steps required to build and deploy a Kubeflow Pipeline is described. Finally how a model serving is done on Kubernetes is explained.

3.1 Kubernetes Cluster

Kubernetes is a container orchestration tool. In Figure 3.1 we can see how a kubernetes cluster is set up.

The main components of a Kubernetes cluster are:

- **Master node**: There are two objectives of a master node, with the first being to maintain the desired cluster state and the second being to maintain and manage the worker nodes [85].
- **Worker node(s)**: These are the machine(s) that run the containerized docker images that are scheduled on pods.

3.1.1 Master Node

The master node consists of:

1. **etcd**: Kubernetes stores the state of the cluster in *etcd*, which is a strongly consistent, distributed key-value store [41]. Kubernetes uses

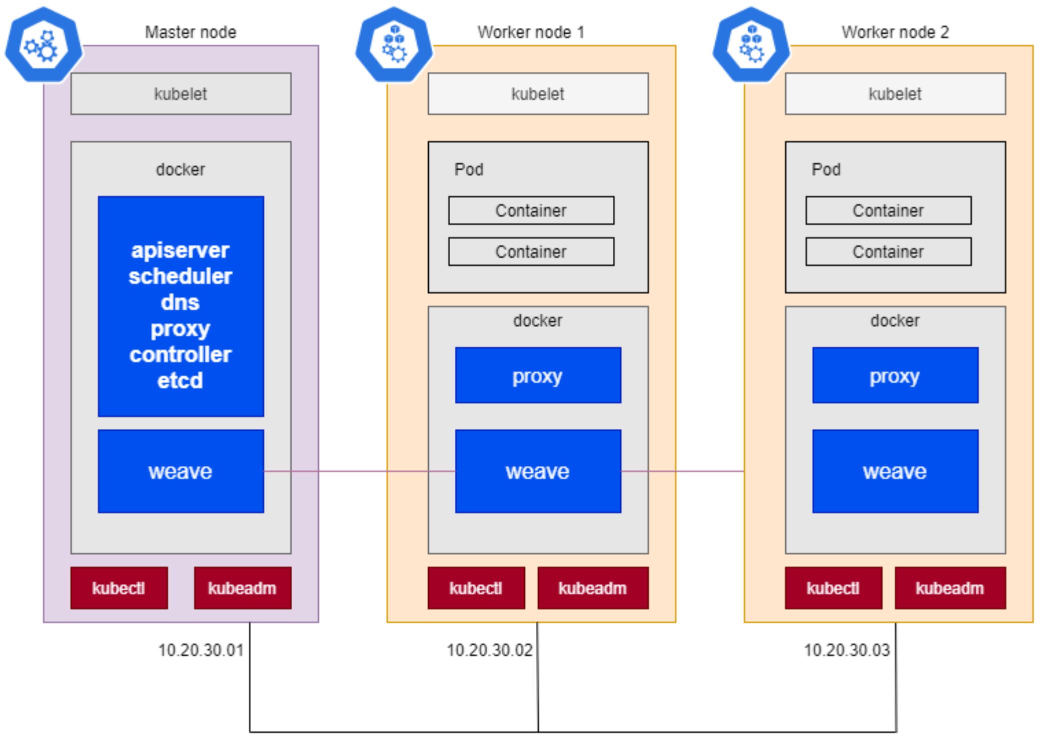


Figure 3.1: Kubernetes architecture

etcd to maintain and store configuration about containers. This service also enables communication amongst containers through the use of shared memory [68].

2. **kube-controller-manager**: A controller is responsible for the replication of pods, ensuring availability, performing node operations, and maintain correct number of pods. *kube-controller-manager* oversees the collection of smaller controllers. If a node fails, this can migrate the pods to other nodes [79].

3. **kube-apiserver**: This component is responsible for receiving and processing commands for Kubernetes objects (such as pods) in the Kubernetes cluster. The commands are sent to the API server using the Kubernetes command -line interface (CLI) called *kubectl* [20]. This API server exposes the underlying Kubernetes APIs and enables interaction of tools including the Kubernetes dashboard and performing the *kubectl* command [21]. This component serves the API with HTTP/REST endpoint and every HTTP call is made to the *kube-apiserver*. For example: if a new pod needs to be created using *kubectl*, a request is sent to this

server with the contents as the pod definition [98].

4. **kube-dns**: Kubernetes has its specific DNS (Domain Name Server) which is deployed as an add-on inside a pod [57]. This enables service discovery. Once a docker container has started, the service IP/port is available as environment variables. The service name is then resolved using *kube-dns* [52].

5. **kube-proxy**: *kube-proxy* runs on all the nodes present in the cluster. This allows for cluster networking including tasks such as ensuring that each node is assigned a unique IP address, implementation of local *IPTABLES* or *IPVS* rules in the pod network which are used to handle routing and load-balancing [74].

6. **kube-scheduler**: *Kube-scheduler* is the default task scheduler in Kubernetes and it uses the pod as the smallest deployable unit [29]. It decides and opts for the most viable node for the placement of any incoming application. The *kube-scheduler* first determines the set of nodes that meet pre-defined constraints such as affinity and node availability. Then using a pre-defined rating function, the node viability is determined. These functions include a rating on image locality, affinity, and resource usage [93].

3.1.2 Worker Nodes

Each worker node consists of:

1. **kubelet**: The *kubelet* agent is executed on worker nodes. This agent connects the worker node to the **api-server** of the master node. *Kubelet* also ensures that all the pods running on the node are healthy [35].

2. **kube-proxy**: *kube-proxy* runs on all the nodes present in the cluster similar to the description in Section 3.1.1.

3. **pods**: *Pods* are the smallest deployable unit of *Kubernetes* and consist of one or more containers that share resources such as network and storage [85]. By default, all Kubernetes pods can communicate with other pods within the same Kubernetes cluster.

   Kubernetes supports two variants of pods [58].

   (a) **Service Pods**: These pods run permanently and are responsible for the background workload of the Kubernetes cluster. These pods track the availability of pods (fault, time to recover, etc.)

and utilization of the service (quality of service, response time, etc.). These pods are also responsible for providing system services such as networking, domain name service (DNS), logging, and monitoring of *Pods*.

(b) **Job/Batch Pods**: These pods are responsible for the actual execution of the tasks and termination on completion of a task. These pods track deployment and the time to execute a task. In case of a required restart of *Pod*, for example due to failure, these *Pods* are responsible for restarting *Pods* based on the restart policy.

3.2 Kubeflow Pipelines setup

Kubeflow Pipelines is a platform for building and deploying portable, scalable containerized machine learning workflows on top of Kubernetes. Kubeflow pipelines allow building of production-level machine learning pipelines without worrying about the low-level details of managing a Kubernetes cluster [14]. Kubeflow Pipelines is a core component of Kubeflow and is deployed at the same time when Kubeflow is deployed. For the purpose of the experimentation described here, the standalone *Kubeflow Pipelines* is deployed.

On a broad level, *Kubeflow Pipelines* consists of:

- A user interface (UI), *Kubeflow-Pipeline-UI* which enables the user to manage and track pipelines.
- An engine to schedule the workflows. It uses Argo Workflows [97] which is a container-native workflow engine for Kubernetes.
- Kubeflow Pipeline SDK installed on the developer machine which allows functionality for creating the pipelines and its components.

Figure 3.2 depicts how a Kubeflow Pipeline is being run. The application code is developed and compiled using the *Kubeflow Pipelines* SDK which creates a *YAML* definition file.

The *YAML* definition file consists of the definition of all the components and their interaction with each other. This definition file is uploaded through the *Kubeflow Pipelines* UI. In the *Kubeflow Pipeline UI* component shown in Figure 3.2, the five green boxes each depict a component.

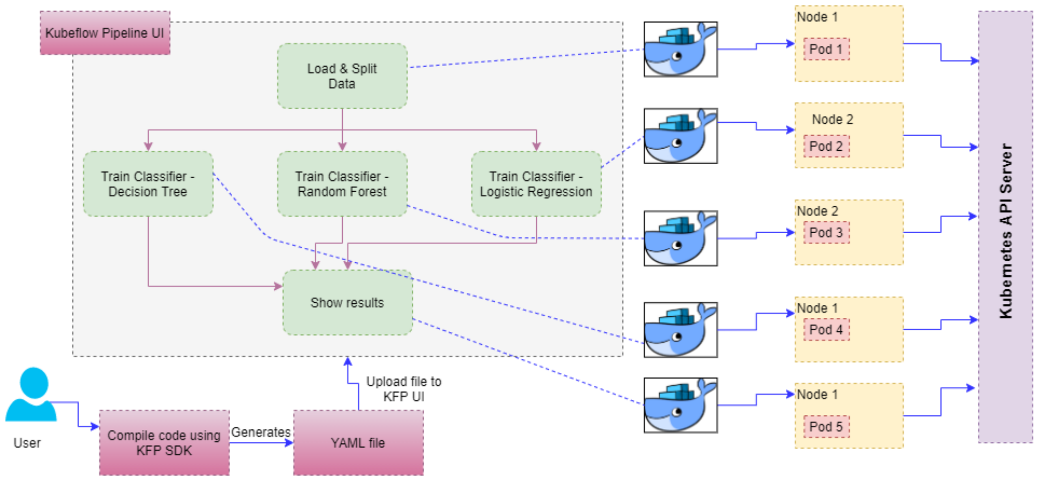


Figure 3.2: Kubeflow Pipelines deployment

Each of these components has a manifest YAML file running in the back-end of the service and is deployed as a docker container. *Each* container is assigned to a worker-node and runs as part of a *Pod*. These nodes perform an *API* call to the *Kubernetes API Server* and then the *API Server* executes the definition file and deploys the pipeline component.

3.3 Building a Kubeflow Pipeline

Figure 3.3 shows how to use the *Kubeflow Pipeline SDK* [43] to build pipelines and components.

Building a *Kubeflow Pipeline* requires the following steps to be referred in context with Figure 3.3:

1. Prepare the machine learning application script, entitled *my-app-code.py*.
2. Build a *Docker* image including the machine learning script and dependencies and push this image to a container registry.
3. A *Domain-Specific Language* (DSL) is used to define and interact with Kubernetes pipelines and its components. Write a *component* function which uses Kubeflow Pipeline DSL or write a *YAML* manifest file describing how the component is built. This description includes the inputs to the component, the output of the component, parameters, dependencies, the *Docker* image to be used, and the command to run the application.

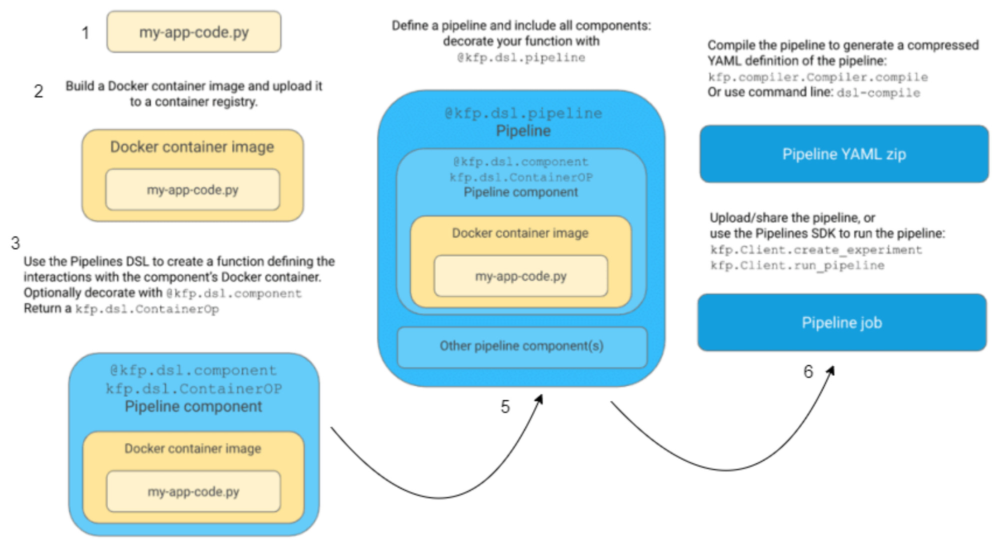

1
my-app-code.py
2
Build a Docker container image and upload it to a container registry.
Docker container image
my-app-code.py
3
Use the Pipelines DSL to create a function defining the interactions with the component's Docker container.
Optionally decorate with @kfp.dsl.component
Return a kfp.dsl.ContainerOp
@kfp.dsl.component
kfp.dsl.ContainerOP
Pipeline component
Docker container image
my-app-code.py
Define a pipeline and include all components: decorate your function with @kfp.dsl.pipeline
@kfp.dsl.pipeline
Pipeline
@kfp.dsl.component
kfp.dsl.ContainerOP
Pipeline component
Docker container image
my-app-code.py
Other pipeline component(s)
5
Compile the pipeline to generate a compressed YAML definition of the pipeline:
kfp.compiler.Compiler.compile
Or use command line: dsl-compile
Pipeline YAML zip
Upload/share the pipeline, or use the Pipelines SDK to run the pipeline:
kfp.Client.create_experiment
kfp.Client.run_pipeline
Pipeline job
6

Figure 3.3: Working of Kubeflow Pipelines [44]

4. Repeat the above steps for all components of the machine learning workflow.

5. Once the component setup is finished, then a pipeline file needs to be created, which will define how the created components interact with each other.

6. Compiling the above-created pipeline file will generate a *YAML* definition of the *Kubernetes* pipeline. This *YAML* definition file is uploaded through the *Kubernetes Pipelines* UI and a pipeline run can be triggered.

3.4 Model Serving in Kubernetes

Kubernetes works by creating resources in a declarative way. Resources are created using the *Kubernetes* API, where the resource definitions are stored in the *etcd* store and the *kube-controller-manager* ensures that these resources are deployed as per specification in the *Kubernetes* cluster.

Kubernetes provide a variety of resource types [47], and the ones used for the setup are described as follows. All these resources are uniquely defined and belong to a *Namespace.*

(a) **Namespace**: *Namespaces* allow a logical organization of clusters into virtual sub-clusters. *Namespaces* are useful when different teams or projects collaborate and share resources within the same *Kubernetes* cluster. Any number of namespaces is allowed within a cluster, each logically separated from the other but can still communicate with each other. Resources are deployed inside a *namespace* [70].

(b) **Replica Set**: A *Replica Set*, as the name suggests improves the reliability and availability of the deployed service by replicating the *Pods* by a specified number of identical *Pods* [17].

(c) **Deployment**: A Deployment provides a declarative update for *Pods* and *ReplicaSets.*

(d) **Service**: It is an abstract way to expose an application running on a set of Pods as a network service.

(e) **Endpoint**: A *Service* creates an endpoint which allows it to send traffic to a *Pod.* Endpoints track the logical addresses of the components or objects the service sends traffic to.

(f) **Service Account**: *Service Accounts* provide an identity to *Pods.* Pods that want to interact with the *kube-apiserver* will authenticate with a particular service account. By default, applications will authenticate as the default service account in the namespace they are running in [30].

(g) **Image Pull Secret**: A *Secret* is defined as a medium to authenticate against services and must be stored securely. Each *Service Account* can have any number of secrets defined. When a *Pod* is created, the *Pod* refers to the service account on which it is running to list *Secrets.* By default, if no image secret is set for the service account, it is unable to pull images from private registries. Hence, for a Pod running on a service account in the *Kubernetes* namespace, to be able to pull images, an image pull secret needs to be created and set as default for the service account namespace being used.

(h) **Storage class**: A *StorageClass* is a Kubernetes API for setting storage parameters using dynamic configuration to enable administrators to create new volumes as needed [96].

(i) **Persistent Volume (PV)**: It is a storage element that is either defined manually by an administrator or is defined dynamically by the *Storageclass*. A PV has a lifecycle that persists beyond the lifecycle of the container [19].

(j) **Persistent Volume Claim (PVC)**: This storage volume represents a user's storage request. The application running on a container can request a specific type of storage, size of the storage, or the way it needs to access the data. A pod needs to be able to claim a persistent volume [19].

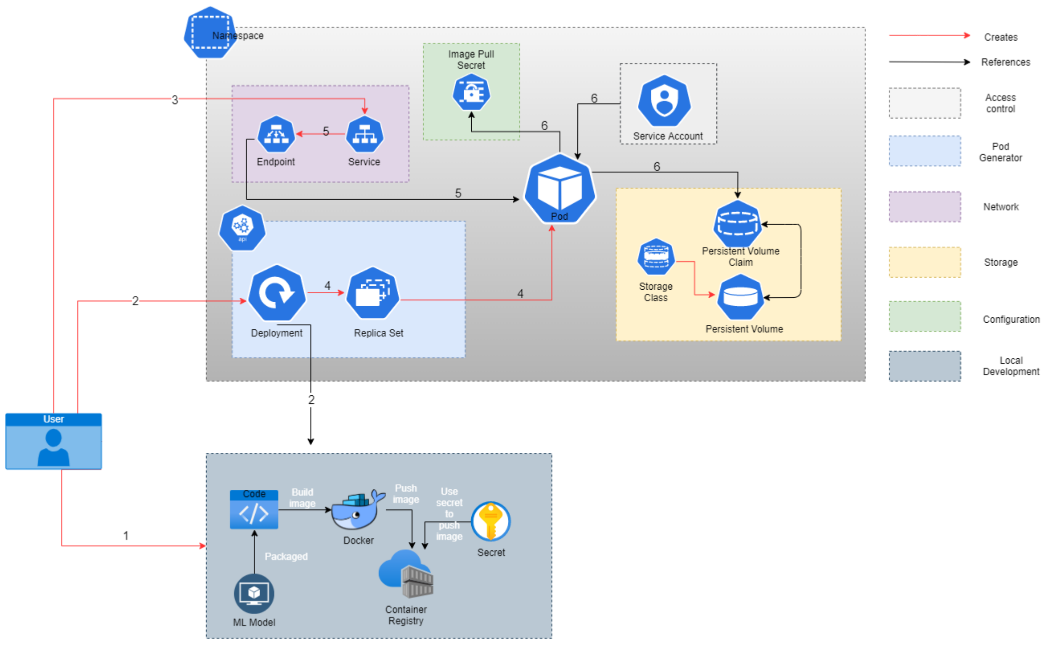


Figure 3.4: Model serving in Kubernetes

The explanation of model serving in the context of Figure 3.4 is as follows:

(a) The machine learning model trained on Kubeflow pipelines is used for making real-time inferences. A *Dockerfile* is created which includes the model serving script and dependencies to create a *Docker* image. The trained machine learning model is also packaged into the Docker image. The *Docker* image is then pushed and hosted by a container registry. If the container registry is private, a *Secret* is supplied.

(b) Definition for the *deployment manifest* is written which references the *Docker* image that needs to be deployed on the *Kubernetes* cluster from the container registry. The *deployment manifest* also contains the configuration including the number of replicas to ensure reliability and availability of the service.

(c) Definition for the *service manifest* is written which references the metadata defined in the deployment manifest. This is applicable when references are within the same namespace.

(d) The *deployment manifest* is executed and the requested pods as per configuration are provisioned.

(e) The *service manifest* is executed and an endpoint is generated by the platform for the *Pod* on which the application is running.

(f) Before a *Pod* is ready for use, we need to validate access to the container registry to ensure that the *Pod* is able to correctly pull the container image and that has the appropriate access. In order for the *Pod* to be able to identify itself, it must be running on a service account where it has access. The *Pod* then tries to claim storage by requesting for *Persistent Volume Claim.*

(g) Once the above steps are completed, a user can then send a request to the exposed service and get a response.

Chapter 4

Platform Setup

In order to build a solution built on top of open-source services, an analysis of compatibility and integration methodologies for those services need to be described. This allows the open-source services to be able to form the basis of the proposed solution and serve as the underlying technology.

In this chapter, the compatibility analysis has been performed empirically due to active ongoing development of services such as *Kubernetes*, *Kubeflow* and *Kubeflow Pipelines*. At the time of writing, it has been found that a combination of specific production versions of the services is required to ensure compatibility between different services and build the proposed solution.

The integration methodology analysis starts with how a Kubernetes cluster can be set up on servers installed with a vanilla-flavored (without customizations) *Ubuntu* operating system and how Kubeflow Pipelines can be integrated to run on the Kubernetes cluster. This setup allows the components of the machine learning workflow to be used as templates and be deployed on a *Kubernetes* cluster.

This chapter showcases how an unmanaged Kubernetes cluster can be set up on an on-premises environment and how on top of this cluster a machine learning platform can be implemented. An unmanaged Kubernetes cluster in this context means that everything starting from setting up the infrastructure, to providing the network configurations, the storage configurations, and setting up the communications between the infrastructure is managed by the enterprise itself and not by the cloud. A managed Kubernetes cluster on the other hand is such where everything will be managed by a service provider. This includes but is not limited to, providing automation tools to set up hosting the machines which will act as the nodes of a Kubernetes cluster, setting up storage, networking, and logging [78]. In this thesis, deployment of an unmanaged Kubernetes cluster is chosen because it is open-source, allows for vulnerability & security checks, can be used without relying on a service provider, and ability to perform customization on a source-code level.

Table 4.1 summarizes the different services that are used to set up this platform along with their versions.

4.1 Setting up Kubernetes

Kubeflow is built on top of Kubernetes. Hence, it is necessary to first have a Kubernetes cluster. A preferred choice for a Kubernetes cluster is to use a managed one which is provided by Azure, Amazon, or GCP. However, this thesis aims to deploy an unmanaged Kubernetes cluster where every component is managed by the user. By installing a self-managed cluster, the user gets control over which features to use and gets a private control-plane. An on-premises installation of Kubernetes is described that involves using virtual machines. Kubernetes can be installed in multiple ways. This thesis describes one such technique where we use virtual machines which are provisioned from *Microsoft Azure.*

4.1.1 Spinning up Virtual machines

In order to create an un-managed Kubernetes cluster, the first step is to create a set of virtual machines. A virtual machine in this context is a vanilla-flavored Ubuntu machine. A vanilla-flavored Ubuntu machine is the unaltered original Ubuntu setup without any customization or updates. These virtual machines act as the constituent elements that the cluster consists of. A minimalistic cluster setup consists of one master node and one worker node (also termed as minions).

The master node is responsible for tasks related to scheduling and the worker node is responsible for tasks related to running the applications. In such a minimalistic setup with one worker node, the node availability for process-intensive tasks is limited. This availability is defined by the specifications of the worker node itself. While any number of nodes can be used to create a cluster; in this setup, two worker nodes are used along with one master node. This brings the total number of virtual machines to three. Each of these nodes is configured with 40 Gigabytes of disk space. The choice of

40 Gigabytes is empirically derived and is guided by the disk-space requirements for the installation setup. Based on the components set up and the disk-space requirement, this figure may vary.

These separate three virtual machines have no inter-dependency but do require a way to communicate with each other. This communication is defined by having a way to orchestrate jobs, split tasks, and allocate resources for the jobs running on them. In order to achieve this, the virtual machines are set up in a single virtual network. This virtual network allows the machines to be connected securely within the same internal private network safe from outside the network communications and are identified by their physical and logical (IP) addresses. Another advantage of using a virtual network is to be able to have static IP addresses which are resistant to change during operational reboots or updates making the communication seamless with a static way of communicating to other machines. This process is demonstrated in A.1.

In order for the developer to be able to communicate to the machines from outside the virtual network, it is required to enable secure SSH connectivity to the machine from the developer's machine. This is achieved by enabling the default SSH port 22 on the virtual machine to accept incoming connections from the developer's machine and authentication using a private SSH key. The developer can then establish a SSH connection using the *public-ip-address-dns-name*. This *public-ip-address-dns-name* is established during the creation of the virtual machine and is used to identify the virtual machine outside the virtual network. There are numerous solutions to achieve a SSH connection. In this approach, it is achieved by using a bash terminal. One terminal allows one active SSH connection to the virtual machine. In order to achieve connection to all three virtual machines described in this setup, three parallel bash terminals can be spun up, as demonstrated in A.2.

These virtual machines, as part of a Kubernetes cluster, are termed as *nodes* of the cluster.

4.1.2 Installing Services

In this step, the goal is to install services on all the three virtual machines created which can enable setting up functionality in the un-managed version of Kubernetes. Services are defined as modules aimed at providing functionality in Kubernetes. In this setup, four such services are installed and Figure

3.1 also shows these services.

The first service is *Docker*. The components of any machine learning application can be packaged with Docker in the form of Docker images. These docker images during run-time would constitute a Docker container. Each of these isolated Docker containers can then be orchestrated through the use of Kubernetes.

The second service is *Kubelet* which acts as the primary node agent running on every node. This service is responsible for managing docker containers according to the workflow defined by Kubernetes. This service registers the node with the API server. A single node may consist of one or more *pods* and in-turn docker containers run on the *pod*. *Kubelet* works through *PodSpec* which is a YAML/JSON object which describes the pod [49]. *Kubelet* uses *PodSpecs* (typically via API Server) and is responsible for ensuring the health and running status of the containers as described in the *PodSpec* [56].

Through *Kubelet*, for each node, this service can manage the pod, docker container, and docker images. When a client request, such as a REST or HTTP request is sent to the cluster, the request goes through the firewall which is then forwarded to the proxy at the node. This proxy then sends the request to *Kubelet* which is responsible for the load balancing and sends the request to the Docker container running in the pod which then processes the request [67]. If *Kubelet* probes unsuccessfully a pre-defined number of times, then the container is marked as unhealthy and a restart is performed within the same pod [53].

The third service is *Kubectl*. *Kubectl* is a command-line tool for Kubernetes which enables running commands against the Kubernetes cluster. This service is used to deploy an application, monitor and manage the cluster resources and investigate logs [18]. The *Kubectl* is the primary service that helps understand the reason an application is failing [55].

The fourth service is *Kubeadm*. *Kubeadm* enables monitoring nodes and service orchestration in a dynamic environment. The master-node of the *Kubernetes* cluster is responsible for orchestration and creation, addition, deletion, monitoring of nodes and services [80]. *Kubeadm* allows performing the necessary actions in order for a minimum-viable Kubernetes cluster. *Kubeadm* is also used to bootstrap the master node followed by integrating worker nodes to the cluster. The integration happens via tokens which creates bi-directional trust amongst nodes. [32].

At the time of writing, *Kubeadm* is under active development and is volatile in nature. For this reason, a specific version (v1.17) is chosen for the installation of the service. This version has been empirically tested to work with this platform setup. In the future, subsequent versions may be stable to use with this setup. This service is not required if using a managed Kubernetes cluster.

The installation of *Kubeadm*, *Kubectl* and *Kubelet* is demonstrated in A.3.

4.1.3 Configuring the master node

Kubernetes has multiple functional services packaged and running via Docker containers such as, but not limited to, controllers, schedulers, proxies and DNS. These services are installed via *Kubeadm.* In order to deploy docker images which can then run as docker container, the *'kubeadm init'* command is executed. This step also installs the *etcd*, *kube-controller-manager*, *kube-apiserver*, *kube-dns*, *kube-proxy* and *kube-scheduler* components.

Once *kubeadm* finishes installing the services, it will generate a *join* command which needs to run on the worker nodes. The purpose of this command is to attach the worker nodes to the master node. By default, root privileges are required to run *kubectl* commands. For convenience, the ownership can be changed using the *admin.conf* file which enables running commands on the nodes without needing root privileges. The output join command is showcased in A.4 and used in Section 4.1.5.

4.1.4 Installing a Network Plugin

Networking is a critical component of Kubernetes because it provides a way to have connectivity amongst different containers either within the same host or across hosts. Kubernetes uses the Container Network Interface (CNI) specification. In order to bootstrap networking in Kubernetes, Container Network Interface is used. This provides a unified interface for container interaction between runtimes [76]. This plugin maintains and orchestrates the entire pod network. For example, if a new pod is included then the *CNI* plugin would connect the container network namespace with the host network namespace. It will then assign a unique logical address to the newly created pod, apply network policies, and updates the new routing table to the entire cluster [77].

In this setup, a Container Network Interface (CNI) is installed before deploying containers on the nodes. Kubernetes uses a special model for networking that allows Docker to connect the containers with a virtual network. In other words, it is a pod network that allows pods to communicate with each other across the nodes. A *CNI* can be installed by using Weave, Flannel, or Calico. While any of these tools can be chosen, in this setup, *Weave* is chosen which creates a virtual network that will connect containers and enable automatic discovery of the containers.

This installation is shown in A.4.

4.1.5 Configuring the worker nodes

Now that the master node is set up, this step focuses on configuring the worker nodes. To join the worker nodes to the master node for them to be able to communicate, we run the output from the *kubeadm join* step mentioned in Section 4.1.3. After the command executes, a prompt would indicate *"This node has joined the cluster"* as shown in A.5.

4.2 Configuring Kubeflow Pipelines on the Kubernetes cluster

Once the Kubernetes platform has been set up, *Kubeflow Pipelines* is installed on top of the platform. Kubeflow Pipelines is a component running inside Kubeflow. At the time of writing, *Kubeflow* v1.3 is used for this setup. This version has been empirically tested to work with this platform setup and eliminates the need to use *kfctl* which is a command-line tool for running *Kubeflow* on *Kubernetes*. In the future, subsequent versions may be stable to use with this setup.

Kubeflow is a platform that simplifies the process of deploying machine learning (ML) workflows on Kubernetes. With the help of *Kubeflow*, a distributed ML deployment is managed by placing different components such as training, inferencing, monitoring, and logging as docker containers in the deployment pipeline. Through *Kubeflow*, the task of managing a Kubernetes cluster is abstracted away allowing the developer to focus on the development of machine learning applications [14].

There are three prerequisites in order to install and configure Kubeflow.

1. Kubernetes version 1.17 with a default Storageclass
2. kustomize version 3.2 (emprically chosen)
3. kubectl (already installed in Section 4.1.2)

4.2.1 Default Storageclass Configuration

The steps to install Kubernetes (v1.17) are described in Section 4.1. To provide storage for the pods, a separate storage provision needs to be done, and not provisioned by default. To achieve this, a *Storageclass* needs to be defined for the cluster. *Storageclass* is a construct in Kubernetes through which storage profiles can be created describing different storage options available for the setup. *Storageclass* can represent types of storage, policies for backup and encryption; and quality-of-service levels. A typical *Storageclass* configuration is a YAML file including the volume plugin, reclaim policies and optionally mount options [19].

There are multiple ways to create a *Storageclass* including, but not limited to Portworx [62], Native Azure Storage Class [25] and OpenEBS [22].

While any of *Storageclass* options can be used; in this setup, *OpenEBS* is used in order to create a *Storageclass* since it is open-source. OpenEBS is a cloud-native storage solution for applications on Kubernetes and enables the management of persistent storage. OpenEBS runs as Container Attached Storage(CAS) and allows for automated provisioning via dynamic persistent volumes [60], [19]. The primary advantages of using OpenEBS are that it is open-source and vendor-agnostic, and it supports provisioning both static and dynamic volumes.

OpenEBS can be installed using the Kubernetes command-line tool (kubectl). This installation and verification is shown in A.13.

When all the pods in the setup are running, a default storage class needs to be created which can instruct *Kubeflow* to route the pods to utilize the default storage. For the default *Storageclass*, the option entitled, *'openebs-hostpath'* is used because it provides support to provision local persistent volumes. Through a local persistent volume, data can be preserved in the storage object.

4.2.2 Installing Kustomize

Kustomize is a tool that helps create declarative customization of the base Kubernetes YAML manifests [69]. It enables the creation of a Kubernetes application without modifying the YAML files for individual components and ties together into a single configuration file. *Kustomize* allows reusing manifests to deploy an application in different environments. This is possible by defining a base manifest consisting of a common base for all environments. For each environment, overlays can be created to customize the environment as per need [54].

The installation is shown in A.13.

4.2.3 Setting up Kubeflow Pipelines

Kubeflow Pipelines is a platform that allows the construction and deployment of dockerized machine learning workflows on top of Kubernetes. It allows the implementation of machine learning pipelines while abstracting the management of the Kubernetes cluster [14]. *Kubeflow Pipelines* is a stand-alone component and can be installed without any other ancillary components offered by *Kubeflow*.

In this setup, *Kubeflow Pipelines* v1.5.1 is chosen (version chosen empirically based on the stability of the tool). Installation is shown in A.14 and takes approximately ten minutes. In order to verify that the pods required by Kubeflow are ready to use, run the commands shown in A.14. A *'Running'* state indicates that pods are ready to use.

Kubeflow Pipelines offers a graphical user interface (UI) through which a developer can visually interact with the platform. To be able to access this user interface, it is required to define the port on which the user interface can be accessed. This is achieved through port-forwarding of the *kubeflow-pipeline-ui* which dictates the port on which the UI can be accessed through the local developer machine. The port forwarding is also shown in A.14. Note that, in an enterprise setup, a port-forwarding would not be required, rather the service itself would be exposed such that the *kubeflow-pipeline-ui* can be accessed outside of the *Kubernetes* cluster.

Once the port forwarding is setup, SSH into the master node using the command shown in A.14 . The port "8080" is used since this is the port we used to port-forward the UI service above. Through the use of this command

Tool	**Version**
Ubuntu	20.04 Lts
Kubectl	1.17
Kubelet	1.17
Kubeadm	1.17
Kustomize	3.2
Kubeflow	1.3
Kubeflow Pipelines	1.5.1

Table 4.1: Tools and the versions used for the setup

for performing an SSH, the remote host IP address is changed to "127.0.0.1". This tells the remote host that the incoming packets need not be redirected outside of port "8080". Instead, the incoming packets are processed internally and then redirected to the inside of port 8080. This way a tunnel is created and it is not blocked by the firewall.

Once the command runs, the *Kubeflow Pipelines* UI can be locally accessed on the *localhost* of the developer machine at port 8080. This user interface enables uploading of machine learning pipelines through a visual interface.

Chapter 5

Experimentation

The goal of this chapter is to demonstrate how a machine learning application can be deployed in an on-premises environment using the proposed solution setup. It is demonstrated how different ML workflow components such as pre-processing and training are containerized and can be re-used. Through this solution, it is possible to understand the abstraction being created from the underlying compute resources allowing the developer to focus on the development of the ML workflow.

For this experimentation, it is shown how a machine learning model can be trained using *Kubeflow Pipelines*, how different steps such as pre-processing and batch inferencing are incorporated as part of a ML workflow, and how the solution can be deployed on *Kubernetes* to generate real-time predictions via an API end-point. To show-case how such a deployment would look in practice, the Iris dataset [71] is used in this Chapter.

5.1 Train a Machine Learning model using Kubeflow-Pipelines

The goal of this section is to be able to train a machine learning application packaged using Kubeflow Pipelines running on top of Kubernetes. Kubeflow pipelines enable building and deploying containerized machine learning workflows [59]. A pipeline consists of components, and each component is a self-contained set of developed code, packaged as a *Docker* image, that performs a single step in the entire pipeline. Kubeflow pipelines are configured using *Python* scripts and are compiled using the Kubeflow Pipelines SDK [45].

The experimentation explores procurement of both locally (within the cluster) and externally stored datasets when developing a ML workflow through Kubeflow Pipelines.

1. **Development With Local Datasets**: The first experiment uses the *Iris* dataset included in the *scikit-learn* Python package and trains two classification models using different algorithms. This experiment variant simulates the assumption that the data resides within the Kubernetes cluster.
2. **Development With External Datasets**: The second experiment uses the *Iris* dataset which is stored in a private storage resource outside the Kubernetes cluster and trains a model using the *Random Forest Classifier* algorithm.

5.1.1 Development With Local Datasets

In this experiment, there are four components, and each component is represented as a *Docker* container. *Docker* images are manually created for the following components and are hosted in ACR.

- **download_data** - Responsible for gathering the data
- **decision_tree** - Modelling using decision trees
- **logistic_regression** - Modelling using logistic regression

The fourth component called *show_results* is responsible for displaying the results of the workflow and will be generated through *Kubeflow Pipelines.*

In order to characterize each Docker container (or component), the definition is supplied through *YAML manifest* files. The pipeline graph including dependencies is configured in a separate Python script.

When the pipeline is compiled, a *YAML* file is generated by *Kubeflow Pipelines* which is then hosted to run through the user interface offered by *Kubeflow Pipelines.* Figure A.3 shows what the Kubeflow Pipeline UI and outputs look like when a run is successful.

5.1.1.1 Load and Split Iris data

The *download_data* component loads the in-built Iris dataset from the *scikit-learn* library in order to simulate how this would look like with data residing within the Kubernetes cluster. The data is then split into a training set and a testing set, as shown in A.15.

Kubeflow handles data differently than traditional machine learning development methods. Initially, a dictionary object is created for the datasets which need to be converted into a *JSON* object. This *JSON* object is then serialized into a binary object defined by command line arguments. The serialization is performed for the other components to be able to recognize the data. Once the *Python* script performing these steps is ready, a *Dockerfile* is created including the script and dependencies. A *Docker* image is created as shown in A.16 and pushed to ACR.

The next step is to define a *YAML manifest* file as shown in A.17, which describes how the component is built for use in the workflow. This manifest file is key to re-usability within *Kubeflow Pipelines*. This file includes information about the input to the component, the output of the component which can serve as input to other components, the location & key to access the *Docker* image hosted in *ACR*; and listing dependencies in order to create a DAG.

This step concludes the building of the *download_data* component.

5.1.1.2 Train and Validate Iris model using Decision Tree Algorithm

The *decision_tree* component is responsible for training a Decision Tree Classifier for the dataset and validating the predictions generated from the resulting machine learning model. Similar to the steps described above, a *Python* script as shown in A.18 is created that is responsible for performing the model training and evaluation. The next step is to build a *Docker* image as shown in A.19 and creation of a *YAML* manifest file. The *YAML* manifest is shown in A.20. This component is also responsible for logging and displaying the accuracy obtained on the test set.

5.1.1.3 Train and Validate Iris model using Logistic Regression Algorithm

The *logistic_regression* component is the similar to the one described in Section 5.1.1.2, except that instead of training a Decision Tree classifier we will train a Logistic Regression model. The Python file is shown in A.21, Dockerfile is shown in A.22 and the *YAML* manifest file is shown in A.23.

5.1.1.4 Show results of both models

The *show_results* component displays the accuracy obtained from both the *decision_tree* and *logistic_regression* components. This component also has a container definition, but need not be explicitly created as done for the other components. Rather, the component's container definition can be created by using the *@func_to_container_op* extension of *kfp.components* from the *Kubeflow Pipelines* SDK.

This way of creating a container definition is possible when the component does not possess any dependencies. Since this component does not have a container definition, we do not need to create a *YAML* manifest file. A.24 shows the code for this. This component is finally created when we generate the script in 5.1.1.5.

5.1.1.5 Build and Run Kubeflow pipeline

Once all four components have been defined, we create a script that defines the structure of the pipeline. This definition explains the interaction of components with each other. This script is shown in A.24. Here, we use the *@dsl.pipeline()* extension from the *Kubeflow Pipelines* SDK. A *decorator* in *Python* adds functionality to an existing code, and this extension is used as a decorator for our main function.

This main function defines the interaction between components. It loads the pre-compiled components built using YAML manifest files. Once the components are loaded, it runs each component by taking in required inputs from previous components.

We then generate a *YAML* manifest file which can then be submitted to the *Kubeflow Pipelines* UI to run the pipeline. The UI renders a visual graph showcasing the relationships and dependencies between components as shown in A.3.

5.1.2 Development With External Datasets

In this experiment, similar to the experiment defined in Section 5.1.1, we have four components where each component is a *Docker* container. The *Docker* images are hosted in *ACR*. The difference in this experiment is that instead of utilizing locally available data within the cluster (simulated by using datasets in *scikit-learn*), we access data from an external data source.

To demonstrate how data can be accessed from a private storage resource, we are using *Azure Blob Storage* to access the dataset. The choice of opting for *Azure Blob Storage* has no relation with the service being on the cloud, rather showcase the access to an enterprise-level private storage resource.

The four components used in this experiment are:

- **preprocess-data**: Responsible for loading data from Azure Blob Storage and splitting into training and test sets.
- **train-model**: Training using a Random Forest (RF) Classifier
- **test-model**: Evaluating the trained Classifier
- **deploy-model**: Serializing the machine learning model object into a *Python* pickle file. As part of this experiment, we add criteria for deployment based on the evaluation metric. The model is to be deployed only if the accuracy obtained from the *'test-model'* component meets a certain threshold.

In this experiment we will not use **YAML** manifest files to create a component, rather we will use the *kfp.dsl.ContainerOp* to return a component function. Figure A.5 shows how the pipeline will look when we have a condition criterion.

5.1.2.1 Load and Split Iris data

In order to access externally stored data, the *preprocess-data* must know the location of the stored data, a mechanism to access it, and a way to authenticate itself securely to the storage service. Importing data stored in Azure Blob Storage requires the following steps:

- Installing the *Azure Blob Service Client SDK.*
- Retrieve the Azure Storage Account URL (location of the resource).

- Retrieve the Azure Storage Account Key (*Secret* for authentication).
- Container name (Container inside *Azure Blob Storage*).
- Dataset file inside the container (example: iris.csv).
- Local file name: Once the data is loaded, it is accessible locally within the cluster with the local file name.

Once the data is loaded from the external storage resource (Blob), the labels for the classes of the dataset are encoded and then split into train and test sets. Once we have the train and test sets, we save these as a comma-separated-value file (CSV) on the container. In the main function, we pass the names to access the external storage resource as an argument that is entered during run-time. The code is shown in A.25. Figure A.4 shows the pipeline UI where we can enter these arguments at run time.

In order to containerize, we create a Dockerfile for this as shown in A.26 and host it in the container registry.

5.1.2.2 Train a Random Forest classifier

In the **train-model** component, we read the training data created in the *preprocess-data* component and create a Random Forest classifier model. The machine learning model object is serialized into a pickle file, which can then be de-serialized in the *test-model* component to evaluate the test set. Again, a *Dockerfile* is created for this and the *Docker* image is pushed into *ACR*.

The component is shown in A.27, and *Dockerfile* in A.28.

5.1.2.3 Evaluate trained model

In the *test-model* component, we de-serialize the machine learning model object and generate predictions for the test set. Based on the predictions from the model, we calculate the accuracy score and save the value to a file. A *Dockerfile* is created for this component and the *Docker* image is pushed into *ACR*.

The component is shown in A.29, and *Dockerfile* in A.30.

5.1.2.4 Deploy model

In the *deploy-model* component, the path of the saved serialized machine learning model object is saved onto this container. This path is used to get the trained model which will be used later when making real-time predictions. The component is shown in A.31, and *Dockerfile* in A.32.

5.1.2.5 Build and run the pipeline

Once the components are defined, the dependency and relationships between components need to be defined. In order to define our component function, we use the *dsl.ContainerOp* extension where we define the *Docker* image name, the input arguments, and the outputs for that component.

This is where the criterion of deploying the machine learning model based on the accuracy threshold is applied. To perform a baseline check, we use the *dsl.Condition* extension. If the accuracy is lower than the baseline then the model will not be deployed. If the accuracy meets the criteria then the model is deployed.

We then generate a *YAML* manifest file which can then be submitted to the *Kubeflow Pipelines* UI to run the pipeline. This process shown in A.33.

5.2 Serving Machine Learning Model on Kubernetes

The goal of this section is to be able to serve a machine learning model packaged using Flask which will run on Kubernetes. Flask is a web micro-framework based on Python and provides a way of implementing a minimalistic web server [94]. A REST API is created which can be utilized by users to make real-time predictions. The workflow involves using the machine learning model trained on Kubeflow Pipelines, building the flask application, containerization of the flask application using Docker, creation of deployment and service files; followed by deployment on Kubernetes.

5.2.1 Training a Machine Learning Model

In this experimentation, we use the trained *Random Forest* classifier model, which was trained in section 5.1.2 to predict the Iris species from the user input consisting of flower characteristics.

5.2.2 Flask Application Implementation

Once the machine learning model is ready, the next step is to enable a way for end-users to consume the machine learning model. Consuming a model refers to the act of making predictions against a machine learning model. In order to achieve this, there are three steps required to be performed.

The first step is to implement a web-server setup which is enabled through the use of *Flask*. *Flask* is responsible for handling API requests made by the end-user and sending an optional API response back to the end-user. Setting up a *Flask* application requires careful handling of open ports on which the web application can be accessed, and is beyond the scope of the discussion in this context. The *Flask* application can be operationally tested by sending a valid JSON response against the logical IP and port on which the application is deployed.

For the machine learning model to be able to make predictions, there may be pre-processing of the user input as part of API request which is taken care of by the second step, called a *scoring function*. An example of the pre-processing could be de-serializing the JSON request into a Pandas dataframe, running the prediction, and serializing the response into a JSON object.

The third step is containerization of the *Flask* application using Docker which can then be run in a platform-agnostic environment. This step is described in further detail in 5.2.3. The logic for the scoring function and instantiating the *Flask* class is performed in the conventionally titled Python script called *app.py*. The serialized model object is built into the *Docker* image itself. When running the *Docker* container, this object is then deserialized and loaded into memory which is then accessed by the request handlers when traffic flows into the application.

The implementation for the *Flask* application is listed in A.6.

5.2.3 Containerize Flask Application

Once the *Flask* application is ready, the application is containerized using Docker. To create a Docker image consisting of the operational environment configuration for the *Flask* application to run, the list of dependencies for the application to run must be listed. A vanilla-flavored *Ubuntu* machine along with the application and Python packages needed to run the application are sufficient to define the operational configuration. To list down the Python packages needed to run the application, a conventional way of defining the dependencies is through creating a *'requirements.txt'* file consisting of the package name and specific versions of the packages. In this setup, this file looks like A.7.

A *Dockerfile* is a file consisting of the sequential list of commands executed on a command line that can be used to create a Docker image. This file is shown in A.8 and consists of the vanilla-flavored *Ubuntu* base docker image on top of which the image is built, along with the *Flask* application files, *entrypoint* (the file to run) and the Python dependencies listed in *'requirements.txt'*. This setup assumes that *Docker* is already installed, configured, and operational in the developer machine.

Once the *Docker* image is created, the image needs to be hosted in a repository from which it can be retrieved. In an on-premise or private cloud setup, there could be an additional resource where such a repository is hosted. In a developer machine, the repository can be locally hosted. In the public domain, it can be hosted on *DockerHub*. In this setup, the choice of the repository has been on yet another service offered by *Microsoft* called *Azure Container Registry* (ACR). Hosting and retrieving a docker image on/from ACR has the most additional configuration needed, and thus chosen for this setup. The choice for ACR has no relevance with respect to utilizing cloud capabilities, rather demonstrate how an enterprise-level secured and private repository can be configured to work with this setup.

As shown in A.9, the command builds a docker image and pushes it to a private ACR. Validation of successful push of Docker image can be run by pulling the pushed image and verifying the same result from the pre-containerized Flask application.

The flask application has been containerized and can run in a platform-agnostic environment supporting Docker. However, simply running this application on a *Docker* container is unable to leverage the functionality and features offered by Kubernetes for scalability. The next step describes deploying this on a Kubernetes setup which can then automate the container deployment process.

5.2.4 Create a Kubernetes Deployment Manifest

A Kubernetes deployment refers to a set of identical pods which have no unique identities. This deployment would run multiple replicas of the application and replaces unhealthy instances. Unhealthy, in this context refers to failure or unresponsiveness. It ensures that at least one instance is available to serve requests from the end-user [46].

A Kubernetes deployment configuration is described in the form of a YAML configuration file and executed using the *kubectl apply* command. In this setup, this file defines the desired state for the *Pods*, the number of replicas needed, the unique label name for the pod, the Docker container registry location (in this setup, it is ACR) from where the *Docker* image is to be pulled, the secret (access key) to the private container registry and the inbound port for the container. The deployment manifest is shown in A.10.

5.2.5 Create a Kubernetes Service Manifest

A Kubernetes Service refers to an abstract way to expose an application that runs on *Pods* as a network service. Kubernetes performs load-balancing and assigns *Pods* their own logical address and a unified DNS name is assigned to the set of *Pods* [48].

The Kubernetes Service manifest defines a logical group of pods which would be a policy enforced across them defining access definitions. In this setup, for demonstration purposes, the outbound port is defined in a way that the endpoint which is created from this manifest can be accessed from both inside and outside the Kubernetes cluster. We also use the unique label name used in the deployment manifest to associate which container runs where. The service manifest is shown in A.11.

5.2.6 Serving an application on Kubernetes

In Figure 3.4, it is shown how a model can be served on Kubernetes. In order to be able to pull the *Docker* image from ACR which will run on the Kubernetes setup, *image secret* must be supplied to the Kubernetes namespace where the application will run. Providing this secret is not relevant for public container registries such as *DockerHub*, but must be supplied for private container registries such as ACR in order to be able to authenticate a valid image pull request.

Once the *image secret* is supplied, we can run the deployment and service files created in subsection 5.2.4 and 5.2.5 using the *kubectl apply* command. Once the deployment is succeeded, the deployment can be tested by sending valid JSON data to the *Kubernetes* API endpoint. There are multiple ways of performing an API request including the use of *curl* bash command. This step is showcased in A.12. Figure A.1 shows the output when the endpoint is accessed from inside the Kubernetes cluster, and Figure A.2 shows the output when the endpoint is accessed from outside the Kubernetes cluster.

Chapter 6

Discussion

In this chapter, an evaluation of the proposed solution is performed and compared with the existing cloud machine learning solutions in the market. This chapter also discusses the advantages and disadvantages of using the proposed solution. To measure the performance, we compare the complexity of developing a solution and the complexity required to migrate to another vendor. Finally, the future work which can be done as part of developing this platform further is described.

6.1 Evaluation of Proposed Solution

Kubeflow makes the deployment of machine learning workflows simple, portable, and scalable. Each of these three characteristics is explained below.

6.1.1 Simplicity

Kubeflow pipelines allow a developer to decompose a machine learning workflow into small components. Decomposition allows the developer to focus on the functionality of that particular component. It also allows collaboration, where different developers can work on different components independently.

In the experiments conducted, the machine learning workflow was decomposed into loading, pre-processing, training, evaluation, and model deployment for batch inferencing.

Decomposition allows the possibility of sequencing the running of components, for example, through the use of directed acyclic graphs and listing the dependencies (input/output) of each component.

In the experiment, the input to the *loading and pre-processing* step was the raw dataset and the output was the processed data. The input to the training component was the processed data generated from the *loading and pre-processing* step, and the output was the serialized object of the machine learning model (pickle file). Finally, the serialized object of the model acted as the input for both the evaluation and the deployment components. The output was an accuracy metric for the evaluation component and inferencing for the deployment component. All these components were configured to run sequentially.

Simplicity also refers to the fact that tasks unrelated to the actual problem being solved are abstracted. Through the use of the presented solution, the developer can be focused on building the actual machine learning solution while the technicalities of the underlying setup and management of the *Kubernetes* cluster is abstracted.

6.1.2 Portability

Portability refers to the fact that the created solution is platform-agnostic. This allows the migration of the solution to an on-premises setup or a public, private, and hybrid cloud environment. Portability allows avoiding vendor lock-in, where migration of the solution to another vendor is either impractical or infeasible.

In this experiment, the entire solution is presented through the pure use of open-source technologies such as *Flask*, *Docker*, *Kubernetes*, *Kubeflow* and its related extensions including *Kubeflow Pipelines*. Using open-source technologies is the first step towards portability.

To use the solution, first, a *Kubernetes* cluster must be set up. On top of this *Kubernetes* cluster, *Kubeflow* can be installed which includes the *Kubeflow Pipelines* component. For a solution to be portable, each of these components must not have any dependencies which would enforce a vendor lock-in.

In this experiment only Kubeflow Pipelines (which is a core component of Kubeflow) is installed since installing all components of Kubeflow is out of the scope of this thesis.

The hardware pre-requisite of installing Kubernetes is a compute instance with command-line capabilities, such as a Linux instance. Adding multiple compute instances, allow the solution to have a master-worker setup, and handle intensive loads with reliability depending on the resource configuration available for the tasks. Linux-based instances can be configured or installed in an on-premises setup or a public, private and hybrid cloud environment. This allows Kubernetes to run in any of these environments.

In order to install *Kubeflow*, the pre-requisite is a *Kubernetes* setup. Since *Kubernetes* can be run in any environment (on-premises or cloud), therefore *Kubeflow* (and *Kubeflow Pipelines*) can be installed on top of *Kubernetes* in any environment.

6.1.3 Scalability

Scalability refers to the capability of a solution to perform as per service quality guidelines when a change in the workload resource demand is required to be handled by the solution. A scalable solution can thus maintain or increase performance when workload demands vary.

Scalability does not simply mean providing more (or less for scale-out) resources to a solution such as compute, memory, storage, and throughput. It also refers to the workload management and efficiency in handling varying workload demands.

For example: if a solution is designed to process data sequentially on a single node, a larger amount of data (than what the solution expects) will still process data sequentially on that node. This sequential processing on one node can act as a bottleneck in the system even if additional compute resources are provided. Instead, a parallel approach utilizing multiple nodes may prove to be more efficient in handling larger tasks because each node gets only a subset of the data to process.

Thus, scalability of a solution means the modification of resources such as memory and storage to handle the varying workload, as well as an architecture capable of handling that workload.

In this experiment, three compute nodes were provisioned which includes one master node and two worker nodes. In order to scale up, the quantity and/or configuration of the worker nodes can be provisioned to meet the demands of the workload. This can be done by joining an additional provisioned resource to the cluster. Similarly, a scale-out can be performed by de-coupling resources from the cluster.

On a *Kubeflow* setup deployed on a *Kubernetes* cluster, it can dynamically scale up or scale out according to the resource demand by the task being executed in a container by changing the resource allocation configuration of the container. The scale-up or scale-out is abstracted and handled by the Kubernetes engine.

For example: if a task running on a container had requested 1GB memory, but on execution actually requires 2GB or 500MB memory, then additional resources can be provisioned or released, depending on the availability of resources on the node where the container is running.

Due to the component re-usability functionality of the proposed setup, it allows migration of the solution to a different environment with a scaled-up or scaled-out configuration. Reusable components and a standardized programming model allow developers to develop and deploy solutions without the need for modifications based on the environment it is developed or trained on.

6.1.4 Configuring Proposed Solution on Cloud Providers

The platform setup and experimentation presented in this thesis have been focused on an on-premises environment. The underlying compute instances can either be on-premises or used from a cloud service provider as *Infrastructure as a Service* (IaaS).

If the compute instances, are provisioned from a cloud service provider, the setup is exactly the same as described in this thesis. Note that this thesis is configured to run on an *Ubuntu* machine. In this scenario, the responsibility of the entire solution development lies with the enterprise.

An alternative scenario is a hybrid development approach, where the Kubernetes setup is offered by the cloud service provider, and the enterprise is responsible for the Kubeflow aspect. In such a scenario, a fully-managed Kubernetes service is used from a cloud provider. Azure Kubernetes Service (AKS) from Azure, Elastic Kubernetes Service (EKS) from AWS, and Google Kubernetes Engine (GKE) for GCP are examples of fully-managed Kubernetes services. In a fully-managed Kubernetes service, the Kubernetes setup is pre-configured in a ready-to-use state.

Kubeflow or only Kubeflow Pipelines can then be installed on top of the fully-managed Kubernetes service, with the steps described in Section 4.2.

6.2 Proposed Solution Versus Existing Cloud Solutions

This section aims to evaluate the proposed solution against existing machine learning platforms offered by cloud service providers. The cloud service providers against which the proposed solution is compared are *Azure Machine Learning* (AzureML) service by *Microsoft Azure* and *AWS SageMaker* (SageMaker) by *Amazon*. The reason both these services are chosen is that both were used before the introduction of *Kubeflow* in the commercial market [64], [73]. Both these services offer additional services such as in-built model versioning and in-built integration to container registry which are currently not fully supported in *Kubeflow*.

The evaluation is done on two aspects and in order to create a baseline for the comparison, the evaluation is done considering developing the same machine learning model described in Chapter 5.

- Complexity for building a solution
- Complexity for migration

6.2.1 Solution Development Complexity

Complexity to build a solution, in the context of this evaluation, refers to the different amount of tasks needed to be performed to build an end-to-end machine learning workflow. For the purposes of this evaluation, it is assumed that a training script, a dataset, and the scoring script are tested and readily available. It is also assumed that the developer has equal expertise in working with all the mentioned technology stacks and services used in the evaluation.

Azure ML and *SageMaker* do not require any platform setup from scratch due to their nature of being managed services. However, with these services, additional resources still need to be provisioned such as the platform itself, storage, networking, and container registries. These resources can be provisioned in an interactive way or through the use of Infrastructure-as-code

(IaC). Ancillary services such as (Docker) container registries and model registries have native integration within their own cloud services. This factor impacts and reduces the complexity to develop a machine learning solution.

However, in order to use ancillary services or tools outside the cloud service provider, in most cases (depending on the service), manual development work needs to be done in order to provide the integration.

For example: *Azure ML* has native integration to *Azure Container Registry* which is an Azure managed Docker container registry, and Azure's own implementation of a model registry for model versioning & model tracking. If in an Azure ML environment, integration of Amazon ECR (Elastic Container Registry) is needed, then that integration needs to be manually done through the use of libraries with no abstraction provided as in the case of native integration.

The proposed solution has no managed service components and requires the platform to be set up. Additional ancillary services such as hyperparameter tuning and data store require manual development or integration to the solution. This, in turn increases the complexity for a machine learning solution when compared to services offered by cloud providers.

Components of a machine learning workflow, such as training and evaluation, in the proposed solution, are containerized into *Docker* images. This allows the components to be independently re-usable across several workflows without any modifications. The *YAML* configuration file for each workflow is listed with the *Docker* images needed for the workflow and is the only step required in order to re-use components. This makes the re-usability of components possible due to the lack of changes that need to be made to be able to use the component.

With respect to complexity for a machine learning solution, cloud services are better when compared to the proposed solution. However, if there are multiple workflows in the solution with a lot of components that can be re-used, the proposed solution is better suited for development because the complexity to re-develop components of the workflow is not required when using the proposed solution.

A detailed summarization of the evaluation is presented in Table 6.1 with the complexity and tasks to be performed when building and deploying a classifier model from the Iris dataset used in Chapter 5.

Table 6.1: Proposed vs. Cloud Solutions Complexity

TASK DEFINITION	**PROPOSED SOLUTION**	**AZURE ML**	**AWS SAGEMAKER**
Provisioning compute & storage resources	Compute instance and networking	Azure ARM Templates	CloudFormation
Platform Setup	Installing Kubernetes	Managed Platform	Managed Platform
	Installing dependencies before Kubeflow		
	Installing Kubeflow Pipelines		
Pre processing script	No modification required	Encapsulation in AzureML Pipeline	Encapsulation in SageMaker pipeline
Training script			
Evaluation script			
Model serving script			
Configuration files	YAML files	Not required	Not required
Build & push images to container registry	Integration to external registry	Native integration	Native integration
Deploying to Kubernetes	Not required	AKS	EKS
Encapsulation into pipeline	Kubeflow Pipelines	AzureML Pipelines	SageMaker Pipelines

6.2.2 Complexity for migration

Complexity for migration refers to the different amount of steps taken to migrate an established machine workflow from one vendor to another. Migration may include the way inputs are passed into the workflow, how outputs are processed from the workflow and how the components interact with other components within the workflow. Migration may be driven due to factors such as cost, functionality, reliability, service level agreements (SLA), and business goals.

Building components within Azure ML, requires an integration of the *Azure Machine Learning SDK* which provides support for interaction with its managed ML service and native integration with its ancillary services. Similarly, building components within AWS SageMaker and interacting with its ancillary services requires an integration of the *Amazon SageMaker SDK*. This enforces a vendor lock-in where migration may be infeasible or impractical.

Consider a scenario, where a machine learning workflow developed using the *Azure ML* service needs to be migrated to *AWS SageMaker.* In a typical migration of a ML workflow, the following steps need to be done:

1. Provision of resources on *AWS* such as SageMaker.
2. Modification of integration of data source.
3. Modification of logging and monitoring components (AWS CloudWatch)
4. Modification of pre-processing, training, and evaluation scripts using the *SageMaker* SDK in order to be encapsulated within a *SageMaker* pipeline.
5. Modification of the output generated through the deployment

All the above steps are eliminated with the proposed solution. When migrating to a new environment, only the infrastructure resources are provisioned and the creation of the new cluster (including joining and configuration of nodes), which can be provisioned through a code template. *Kubernetes* and *KubeFlow* setup procedure is followed for the new environment and the ML workflow containerized components work without modifications due to no dependency on any particular vendor.

Thus, complexity for migration is lower for the proposed solution as compared to the cloud services.

6.3 Future Work

While the proposed solution is presented as an approach to deploy machine learning solutions on an on-premises environment, there are many functionalities that can enrich the platform. This section covers what those additional functionalities are, and how they can be incorporated into this setup.

In this thesis, *KubeFlow Pipelines* being used is just one component from the *KubeFlow* component offerings and extensions. KubeFlow has support for integration of *Jupyter Notebooks* which provide a way to code, present visualizations, marked-up text, and equations. This functionality can be enabled through the *Kubeflow Notebook Server* component and can improve collaboration between developers or teams. *Jupyter Notebooks* already exist as part of cloud machine learning services in the market.

KubeFlow offers a component called *Katib* [36], which allows hyper-parameter tuning for machine learning models. It is a machine learning framework agnostic and can tune hyper-parameters irrespective of the programming language the ML application is written in. This component also provides many AutoML algorithms such as *Bayesian optimization* and random search. At the time of writing, this component is in the beta stage and not stable for production use.

In order to add the functionality for logging and monitoring the workflow, the *Prometheus* component can be used [39]. *Prometheus* is an open-source toolkit used for system monitoring and alerting. Logging and monitoring components native to *KubeFlow* are under development at the time of writing.

The Kubeflow experimentation presented in this thesis is conducted on a batch-prediction variant of deployment. In order to use real-time as well as batch predictions in Kubeflow, the *KFServing* [42] component can be used. This allows serving machine learning models, is framework-agnostic, and abstracts the management of the deployment such as auto-scaling, health checks, networking, and configuration of the server.

KubeFlow lacks a *Feature Store* functionality which acts as a central repository to store features for machine learning. In order to incorporate this functionality, an open-source feature store such as *Feast* can be used to simplify the process of feature engineering [26]. In order to orchestrate multiple workflows within *KubeFlow*, an open-source platform such as *Apache Airflow* can be used to create, schedule, orchestrate and monitor the workflows [86]. Airflow is a generic task orchestration platform, while Kubeflow focuses specifically on machine learning workflows. This is useful to enable job orchestration and perform jobs such as Extract-Transform-Load (ETL) for the input data.

Finally, in order to facilitate migrations of the proposed solution, *Helm Charts* can be used [16]. A *Helm Chart* is a collection of files that describes a related set of *Kubernetes* resources. At the time of writing this thesis, *Helm Charts* for *Kubeflow* is under development.

At the time of writing, *KubeFlow* is at an evolving stage and continues to develop. There is a multitude of functionality which is currently under development and can compete with other solutions to be a feature-rich end-to-end solution.

Chapter 7

Conclusion

This thesis described the need for an on-premises solution and how portability of a solution can be maintained. A standardized architecture for a machine learning platform is presented built using open-source technologies and ensuring ease of migration to a different environment. Kubeflow is a platform-agnostic, open-source tool chosen to run machine learning workflows as containers, and Kubernetes is chosen for the orchestration of these containers. This allows the proposed solution to be configured in any environment where Kubernetes can be run, i.e anywhere from on-premises data centres, simulation of Kubernetes on local machine or on any of the cloud providers environment.

The thesis also discusses the advantages of using open-source technologies over cloud solutions, the functionalities of a machine learning platform provided by cloud providers which can be built by using combination of open-source tools for on-premises solutions. We also discussed why a Kubernetes based *MLOps* platform is needed and how it simplifies the production of a machine learning solution.

The different components offered by Kubeflow is discussed, and how Kubeflow Pipelines can be a potential solution to make an end-to-end machine learning solution simple, portable and scalable. The complexity of development on Azure ML and Amazon Sagemaker versus the proposed Kubeflow solution is discussed, and a Kubeflow based solution has more steps in the development process. However, when we compare the complexity to migrate a Kubeflow based solution to any cloud provider environment or to another environment where Kubernetes runs, then this solution will require lesser amount of development that the solution provided by cloud providers.

Bibliography

[1] Abdalla, P. A., and Varol, A. Advantages to disadvantages of cloud computing for small-sized business. In *2019 7th International Symposium on Digital Forensics and Security (ISDFS)* (2019), IEEE, pp. 1–6.

[2] Alla, S., and Adari, S. K. Deploying in aws. In *Beginning MLOps with MLFlow*. Springer, 2021, pp. 229–252.

[3] Alla, S., and Adari, S. K. Introduction to mlflow. In *Beginning MLOps with MLFlow*. Springer, 2021, pp. 125–227.

[4] Amazon. Amazon machine learning. `https://docs.aws.amazon.com/machine-learning/latest/dg/training-ml-models.html`. Accessed: 2021-08-03.

[5] Amazon. Sagemaker experiments. `https://aws.amazon.com/blogs/aws/amazon-sagemaker-experiments-organize-track-and-compare\-your-machine-learning-trainings/`. Accessed: 2021-08-18.

[6] Amazon. Sagemaker kubeflow. `https://aws.amazon.com/blogs/machine-learning/introducing-amazon-sagemaker-components-for-kubeflow-pipelines/`. Accessed: 2021-08-18.

[7] Anoshin, D., Shirokov, D., and Strok, D. Snowflake and data science. In *Jumpstart Snowflake*. Springer, 2020, pp. 213–228.

[8] Ashley, D. Using flask and jinja. In *Foundation Dynamic Web Pages with Python*. Springer, 2020, pp. 159–181.

[9] Avram, M.-G. Advantages and challenges of adopting cloud computing from an enterprise perspective. *Procedia Technology 12* (2014), 529–534.

[10] Barnes, J. Azure machine learning. *Microsoft Azure Essentials. 1st ed, Microsoft* (2015).

[11] Baylor, D., Breck, E., Cheng, H.-T., Fiedel, N., Foo, C. Y., Haque, Z., Haykal, S., Ispir, M., Jain, V., Koc, L., et al. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2017), pp. 1387–1395.

[12] Bernstein, D. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing 1*, 3 (2014), 81–84.

[13] Biessmann, F., Golebiowski, J., Rukat, T., Lange, D., and Schmidt, P. Automated data validation in machine learning systems. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* (2021).

[14] Bisong, E. Kubeflow and kubeflow pipelines. In *Building Machine Learning and Deep Learning Models on Google Cloud Platform*. Springer, 2019, pp. 671–685.

[15] Buchanan, S., Rangama, J., and Bellavance, N. Container registries. In *Introducing Azure Kubernetes Service*. Springer, 2020, pp. 17–34.

[16] Buchanan, S., Rangama, J., and Bellavance, N. Helm charts for azure kubernetes service. In *Introducing Azure Kubernetes Service*. Springer, 2020, pp. 151–189.

[17] Buchanan, S., Rangama, J., and Bellavance, N. Inside kubernetes. In *Introducing Azure Kubernetes Service*. Springer, 2020, pp. 35–50.

[18] Buchanan, S., Rangama, J., and Bellavance, N. kubectl overview. In *Introducing Azure Kubernetes Service*. Springer, 2020, pp. 51–62.

[19] Caban, W. Storage. In *Architecting and Operating OpenShift Clusters*. Springer, 2019, pp. 77–98.

[20] Chang, C.-C., Yang, S.-R., Yeh, E.-H., Lin, P., and Jeng, J.-Y. A kubernetes-based monitoring platform for dynamic cloud resource provisioning. In *GLOBECOM 2017-2017 IEEE Global Communications Conference* (2017), IEEE, pp. 1–6.

[21] Chawla, H., and Kathuria, H. Azure kubernetes service. In *Building Microservices Applications on Microsoft Azure*. Springer, 2019, pp. 151–177.

[22] Chazapis, A., Pinto, C., Gkoufas, Y., Kozanitis, C., and Bilas, A. A unified storage layer for supporting distributed workflows in kubernetes. In *Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems* (2021), pp. 1–9.

[23] Chen, A., Chow, A., Davidson, A., DCunha, A., Ghodsi, A., Hong, S. A., Konwinski, A., Mewald, C., Murching, S., Nykodym, T., et al. Developments in mlflow: A system to accelerate the machine learning lifecycle. In *Proceedings of the fourth international workshop on data management for end-to-end machine learning* (2020), pp. 1–4.

[24] CNVRG. Cnvrg. `https://cnvrg.io/`. Accessed: 2021-08-09.

[25] Copeland, M., Soh, J., Puca, A., Manning, M., and Gollob, D. Understanding azure storage and databases. In *Microsoft Azure*. Springer, 2015, pp. 177–201.

[26] Cox, C., Sun, D., Tarn, E., Singh, A., Kelkar, R., and Goodwin, D. Serverless inferencing on kubernetes. *arXiv preprint arXiv:2007.07366* (2020).

[27] Das, P., Ivkin, N., Bansal, T., Rouesnel, L., Gautier, P., Karnin, Z., Dirac, L., Ramakrishnan, L., Perunicic, A., Shcherbatyi, I., et al. Amazon sagemaker autopilot: a white box automl solution at scale. In *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning* (2020), pp. 1–7.

[28] Derakhshan, B., Rezaei Mahdiraji, A., Abedjan, Z., Rabl, T., and Markl, V. Optimizing machine learning workloads in collaborative environments. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), pp. 1701–1716.

[29] Douhara, R., Hsu, Y.-F., Yoshihisa, T., Matsuda, K., and Matsuoka, M. Kubernetes-based workload allocation optimizer for minimizing power consumption of computing system with neural network. In *2020 International Conference on Computational Science and Computational Intelligence (CSCI)* (2020), IEEE, pp. 1269–1275.

[30] D'Silva, D., and Ambawade, D. D. Building a zero trust architecture using kubernetes. In *2021 6th International Conference for Convergence in Technology (I2CT)* (2021), IEEE, pp. 1–8.

[31] Ebert, C., Gallardo, G., Hernantes, J., and Serrano, N. Devops. *Ieee Software 33*, 3 (2016), 94–100.

[32] Galloway, J., Henson, B., Kirby, A., Thomas, J., and Armstrong, M. On deployment of a local bioinformatics cloud. In *2019 IEEE Cloud Summit* (2019), IEEE, pp. 92–98.

[33] García, Á. L., De Lucas, J. M., Antonacci, M., Zu Castell, W., David, M., Hardt, M., Iglesias, L. L., Moltó, G., Plociennik, M., Tran, V., et al. A cloud-based framework for machine learning workloads and applications. *IEEE access 8* (2020), 18681–18692.

[34] Gastermann, B., Stopper, M., Kossik, A., and Katalinic, B. Secure implementation of an on-premises cloud storage service for small and medium-sized enterprises. *Procedia Engineering 100* (2015), 574–583.

[35] Gatev, R. Getting up to speed with kubernetes. In *Introducing Distributed Application Runtime (Dapr)*. Springer, 2021, pp. 51–67.

[36] George, J., Gao, C., Liu, R., Liu, H. G., Tang, Y., Pydipaty, R., and Saha, A. K. A scalable and cloud-native hyperparameter tuning system. *arXiv preprint arXiv:2006.02085* (2020).

[37] Google. Vertex ai. `https://cloud.google.com/vertex-ai/docs`. Accessed: 2021-08-01.

[38] Hardt, M., Chen, X., Cheng, X., Donini, M., Gelman, J., Gollaprolu, S., He, J., Larroy, P., Liu, X., McCarthy, N., et al. Amazon sagemaker clarify: Machine learning bias detection and explainability in the cloud. *Amazon SageMaker Clarify* (2021).

[39] Hong, Y., and Kim, J. Workflow improvement for kubeflow dl performance over cloud-native smartx ai cluster. In *International Conference on Emerging Internetworking, Data & Web Technologies* (2020), Springer, pp. 522–531.

[40] Janardhanan, P. Project repositories for machine learning with tensorflow. *Procedia Computer Science 171* (2020), 188–196.

[41] Jeffery, A., Howard, H., and Mortier, R. Rearchitecting kubernetes for the edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking* (2021), pp. 7–12.

[42] Klaise, J., Van Looveren, A., Cox, C., Vacanti, G., and Coca, A. Monitoring and explainability of models in production. *arXiv preprint arXiv:2007.06299* (2020).

[43] Kubeflow. Kubeflow pipeline documentation. `https://kubeflow-pipelines.readthedocs.io/en/stable/source/kfp.html`. Accessed: 2021-07-31.

[44] Kubeflow. Kubeflow pipeline working. `https://v0-6.kubeflow.org/docs/pipelines/sdk/sdk-overview/`. Accessed: 2021-07-31.

[45] Kubeflow. Kubeflow pipelines. `https://www.kubeflow.org/docs/components/pipelines/overview/pipelines-overview/`. Accessed: 2021-08-13.

[46] Kubernetes. Deployment. `https://cloud.google.com/kubernetes-engine/docs/concepts/deployment`. Accessed: 2021-08-14.

[47] Kubernetes. Kubernetes documentation. `https://kubernetes.io/docs/home/`. Accessed: 2021-08-10.

[48] Kubernetes. Service. `https://kubernetes.io/docs/concepts/services-networking/service/`. Accessed: 2021-08-14.

[49] Kubernetes. Kubelet. Tech. rep., Google, 2021.

[50] Lee, K. M., Yoo, J., Kim, S.-W., Lee, J.-H., and Hong, J. Autonomic machine learning platform. *International Journal of Information Management 49* (2019), 491–501.

[51] López, P. G., Arjona, A., Sampé, J., Slominski, A., and Villard, L. Triggerflow: trigger-based orchestration of serverless workflows. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems* (2020), pp. 3–14.

[52] Manaouil, K., and Lebre, A. *Kubernetes and the Edge?* PhD thesis, Inria Rennes-Bretagne Atlantique, 2020.

[53] Martin, P. Application self-healing. In *Kubernetes*. Springer, 2021, pp. 79–87.

[54] Martin, P. Command-line tools. In *Kubernetes*. Springer, 2021, pp. 193–217.

[55] Martin, P. Observability. In *Kubernetes*. Springer, 2021, pp. 175–183.

[56] Martin, P. Scheduling pods. In *Kubernetes*. Springer, 2021, pp. 89–100.

[57] Medel, V., Rana, O., Bañares, J. Á., and Arronategui, U. Modelling performance & resource management in kubernetes. In *Proceedings of the 9th International Conference on Utility and Cloud Computing* (2016), pp. 257–262.

[58] Medel, V., Tolosana-Calasanz, R., Bañares, J. Á., Arronategui, U., and Rana, O. F. Characterising resource management performance in kubernetes. *Computers & Electrical Engineering 68* (2018), 286–297.

[59] Melenli, S., and Topkaya, A. Real-time maintaining of social distance in covid-19 environment using image processing and big data. In *2020 Innovations in Intelligent Systems and Applications Conference (ASYU)* (2020), IEEE, pp. 1–5.

[60] Mercl, L., and Pavlik, J. Public cloud kubernetes storage performance analysis. In *International Conference on Computational Collective Intelligence* (2019), Springer, pp. 649–660.

[61] Modak, A., Chaudhary, S., Paygude, P., and Ldate, S. Techniques to secure data on cloud: Docker swarm or kubernetes? In *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)* (2018), IEEE, pp. 7–12.

[62] Mohamed, M., Engel, R., Warke, A., Berman, S., and Ludwig, H. Extensible persistence as a service for containers. *Future Generation Computer Systems 97* (2019), 10–20.

[63] Mufid, M. R., Basofi, A., Al Rasyid, M. U. H., Rochimansyah, I. F., et al. Design an mvc model using python for flask framework development. In *2019 International Electronics Symposium (IES)* (2019), IEEE, pp. 214–219.

[64] Mund, S. *Microsoft azure machine learning*. Packt Publishing Ltd, 2015.

[65] Mustafa, B., Tao, W., Lee, J., and Thorpe, S. Kubernetes from the ground up deploy and scale performant and reliable containerized applications with kubernetes. *Kubernetes from the ground up deploy and scale performant and reliable containerized applications with Kubernetes* (2018).

[66] Nargesian, F., Samulowitz, H., Khurana, U., Khalil, E. B., and Turaga, D. S. Learning feature engineering for classification. In *Ijcai* (2017), pp. 2529–2535.

[67] Netto, H. V., Luiz, A. F., Correia, M., de Oliveira Rech, L., and Oliveira, C. P. Koordinator: A service approach for replicating docker containers in kubernetes. In *2018 IEEE Symposium on Computers and Communications (ISCC)* (2018), IEEE, pp. 00058–00063.

[68] Netto, H. V., Lung, L. C., Correia, M., Luiz, A. F., and de Souza, L. M. S. State machine replication in containers managed by kubernetes. *Journal of Systems Architecture 73* (2017), 53–59.

[69] Orzechowski, M., Baliś, B., Słota, R. G., and Kitowski, J. Reproducibility of computational experiments on kubernetes-managed container clouds with hyperflow. In *International Conference on Computational Science* (2020), Springer, pp. 220–233.

[70] Packard, M., Stubbs, J., Drake, J., and Garcia, C. Real-world, self-hosted kubernetes experience. In *Practice and Experience in Advanced Research Computing*. Kubernetes, 2021, pp. 1–5.

[71] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[72] Polyaxon. Polyaxon documentation. `https://polyaxon.com/docs/`. Accessed: 2021-08-01.

[73] Posoldova, A. Machine learning pipelines: From research to production. *IEEE Potentials 39*, 6 (2020), 38–42.

[74] Poulton, N. *The Kubernetes Book*. Amazon, 2021.

[75] Preeth, E., Mulerickal, F. J. P., Paul, B., and Sastri, Y. Evaluation of docker containers based on hardware utilization. In *2015 international conference on control communication & computing India (ICCC)* (2015), IEEE, pp. 697–700.

[76] Qi, S., Kulkarni, S. G., and Ramakrishnan, K. Assessing container network interface plugins: Functionality, performance, and scalability. *IEEE Transactions on Network and Service Management 18*, 1 (2020), 656–671.

[77] Qi, S., Kulkarni, S. G., and Ramakrishnan, K. Understanding container network interface plugins: Design considerations and performance. In *2020 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN* (2020), IEEE, pp. 1–6.

[78] Riti, P. *Pro DevOps with Google Cloud Platform: With Docker, Jenkins, and Kubernetes.* Springer, 2018.

[79] Sabharwal, N., and Pandey, P. Container overview. In *Monitoring Microservices and Containerized Applications.* Springer, 2020, pp. 1–20.

[80] Sami, H., and Mourad, A. Towards dynamic on-demand fog computing formation based on containerization technology. In *2018 International Conference on Computational Science and Computational Intelligence (CSCI)* (2018), IEEE, pp. 960–965.

[81] Sayfan, G. *Mastering kubernetes.* Packt Publishing Ltd, 2017.

[82] Schmidt, R. F. Chapter 3 - software architecture. In *Software Engineering*, R. F. Schmidt, Ed. Morgan Kaufmann, Boston, 2013, pp. 43–54.

[83] Shah, J., and Dubaria, D. Building modern clouds: using docker, kubernetes & google cloud platform. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)* (2019), IEEE, pp. 0184–0189.

[84] Shahoud, S., Gunnarsdottir, S., Khalloof, H., Duepmeier, C., and Hagenmeyer, V. Facilitating and managing machine learning and data analysis tasks in big data environments using web and microservice technologies. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XLV.* Springer, 2020, pp. 132–171.

[85] Shamim, M. S. I., Bhuiyan, F. A., and Rahman, A. Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices. In *2020 IEEE Secure Development (SecDev)* (2020), IEEE, pp. 58–64.

[86] Singh, P. Airflow. In *Learn PySpark*. Springer, 2019, pp. 67–84.

[87] Soh, J., and Singh, P. Apache spark, big data, and azure databricks. In *Data Science Solutions on Azure*. Springer, 2020, pp. 201–223.

[88] Soh, J., and Singh, P. Hands-on with azure machine learning. In *Data Science Solutions on Azure*. Springer, 2020, pp. 149–199.

[89] Soh, J., and Singh, P. Machine learning operations. In *Data Science Solutions on Azure*. Springer, 2020, pp. 259–279.

[90] Song, M., Zhang, C., and Haihong, E. An auto scaling system for api gateway based on kubernetes. In *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)* (2018), IEEE, pp. 109–112.

[91] Stack, P., Xiong, H., Mersel, D., Makhloufi, M., Terpend, G., and Dong, D. Self-healing in a decentralised cloud management system. In *Proceedings of the 1st International Workshop on Next generation of Cloud Architectures* (2017), pp. 1–6.

[92] Tesliuk, A., Bobkov, S., Ilyin, V., Novikov, A., Poyda, A., and Velikhov, V. Kubernetes container orchestration as a framework for flexible and effective scientific data analysis. In *2019 Ivannikov Ispras Open Conference (ISPRAS)* (2019), IEEE, pp. 67–71.

[93] Tzenetopoulos, A., Masouros, D., Xydis, S., and Soudris, D. Interference-aware orchestration in kubernetes. In *International Conference on High Performance Computing* (2020), Springer, pp. 321–330.

[94] Vogel, P., Klooster, T., Andrikopoulos, V., and Lungu, M. A low-effort analytics platform for visualizing evolving flask-based python web services. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)* (2017), IEEE, pp. 109–113.

[95] Webb, G. I., Hyde, R., Cao, H., Nguyen, H. L., and Petitjean, F. Characterizing concept drift. *Data Mining and Knowledge Discovery 30*, 4 (2016), 964–994.

[96] Weissman, B., and Nocentino, A. E. Deploying a data controller. In *Azure Arc-Enabled Data Services Revealed*. Springer, 2021, pp. 67–83.

[97] Workflows, A. Argo workflows. `https://github.com/argoproj/argo-workflows`. Accessed: 2021-07-31.

[98] Yilmaz, O. Extending the kubernetes api. In *Extending Kubernetes*. Springer, 2021, pp. 99–141.

[99] Zaharia, M., Chen, A., Davidson, A., Ghodsi, A., Hong, S. A., Konwinski, A., Murching, S., Nykodym, T., Ogilvie, P., Parkhe, M., et al. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull. 41*, 4 (2018), 39–45.

[100] Zheng, A., and Casari, A. *Feature engineering for machine learning: principles and techniques for data scientists.* " O'Reilly Media, Inc.", 2018.

[101] Zhou, Y., Yu, Y., and Ding, B. Towards mlops: A case study of ml pipeline platform. In *2020 International Conference on Artificial Intelligence and Computer Engineering (ICAICE)* (2020), IEEE, pp. 494–500.

Appendix A

Appendix

```
#Login to azure using your active directory credentials
az login

#List all subscriptions as a table
az account list -o table

#Set the subscription to be used
az account set -s <your-subscription-id>

#Create a resource group
az group create -n <your-resource-group-name> -l<location-of-resource-group>

#Create a VNet (Virtual Network)
az network vnet create -g <your-resource-group-name> -n <your-vnet-name> \
--address-prefix 172.0.0.0/16 --subnet-name MySubnet --subnet-prefix 172.0.0.0/24

#Create 3 Virtual Machines

##Create the VM which will act as the Kubernetes Master node
az vm create -n kube-master -g <your-resource-group-name> --image ubuntults
--size Standard_DS2_v2 \
--data-disk-sizes-gb 50 --generate-ssh-keys \
--public-ip-address-dns-name <your-public-dns-name>

##Create 2 worker nodes
az vm create -n kube-worker-1 -g <your-resource-group-name> --image ubuntults
--size Standard_DS2_v2 \
--data-disk-sizes-gb 50 --generate-ssh-keys \
--public-ip-address-dns-name <your-public-dns-name>
```

```

az vm create --n kube-worker-2 -g <your-resource-group-name> --
    image ubuntults
--size Standard_DS2_v2 \
--data-disk-sizes-gb 50 --generate-ssh-keys \
--public-ip-address-dns-name <your-public-dns-name>
```

Code Listing A.1: Creation of Virtual Machines

```
#Open 3 terminals and login to each VM using a separate terminal

#Connecting to master node
ssh -i <private key path> <public-dns-name-of-master-node>.<
    resource-group-location>.cloudapp.azure.com

#Connecting to worker nodes
ssh -i <private key path> <public-dns-name-of-worker-node-1>.<
    resource-group-location>.cloudapp.azure.com

ssh -i <private key path> <public-dns-name-of-worker-node-2>.<
    resource-group-location>.cloudapp.azure.com
```

Code Listing A.2: Login to the Virtual machines using ssh

```
#These steps need to be performed on all nodes
sudo apt update

#Install Docker
sudo apt install docker.io -y
sudo systemctl enable docker

#Get the gpg keys for Kubeadm
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg |
    sudo apt-key add
sudo apt-add-repository "deb http://apt.kubernetes.io/
    kubernetes-xenial main"

#Install Kubeadm, Kubelet and Kubectl.
#NOTE: specific versions as mentioned are to be used.
sudo apt install -y kubeadm=1.17.1-00 kubectl=1.17.1-00 kubelet
    =1.17.1-00
```

Code Listing A.3: Install services on all virtual machines

```
#Initialize kubeadm only on the master node
#This will output a join command which is needed later
sudo kubeadm init

#Copy admin.conf so that kubectl can be used normally from the
    master node
```

```
mkdir $HOME/.kube
sudo cp /etc/kubernetes/admin.conf $HOME/.kube/config

#Change ownership of file to current user and group
sudo chown $(id -u):$(id -g) $HOME/.kube/config

#Install weave on the master node
kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=
    $(kubectl version | base64 | tr -d '\n')"
```

Code Listing A.4: Initialize kubeadm and weave

```
#Run kubeadm join command from each of the worker nodes
sudo kubeadm join 172.0.0.4:6443 --token <your-token> \
    --discovery-token-ca-cert-hash <your-sha256-hash>

#If the join is successful, the prompt will show: "Node has
    joined the cluster"
```

Code Listing A.5: Kubeadm join command to attach the worker nodes to the master ndoe

```
#Imports and definitions
from flask import Flask, request
import pickle
from numpy import array2string

#path to the pickled model
model_path = "rf-model.pkl"

#Unpickle the random forest model
with open(model_path, "rb") as file:
    unpickled_rf = pickle.load(file)

#Define the app
app = Flask(__name__)

#Use decorator to define the /score input method and define the
    predict function
@app.route("/score", methods=["POST"])
def predict_species():
    #List to append inputs
    flower = []

    #Print input values
    print(request.get_json()["petal_length"])
    print(request.get_json()["petal_width"])
    print(request.get_json()["sepal_length"])
    print(request.get_json()["sepal_width"])

    flower.append(request.get_json()["petal_length"])
    flower.append(request.get_json()["petal_width"])
    flower.append(request.get_json()["sepal_length"])
    flower.append(request.get_json()["sepal_width"])

    print(flower)

    #Return the prediction
    return array2string(unpickled_rf.predict([flower]))


#Run the app
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5005)
```

Code Listing A.6: Create a flask application for Iris dataset

```
flask==2.0.1
numpy==1.19.3
scikit-learn==0.24.1
```

Code Listing A.7: Requirements file for running a Flask application

```
#set base image (host OS)
FROM python:3.8

#set the working directory in the container
WORKDIR /code

#expose port - use the same port as in the flask app
EXPOSE 5005

#copy the dependencies file to the working directory
COPY requirements.txt .

#install dependencies
RUN pip install -r requirements.txt

#copy the content of the directory to the working directory
COPY . .

# command to run on container start - use the same port as in
    the flask app
ENTRYPOINT FLASK_APP=app.py flask run --host=0.0.0.0 --port=5005
```

Code Listing A.8: Dockerfile for Flask application

```
#Build the docker image container on your local machine
```

```
docker build -t iris-demo .

#Pushing the container to  docker registry
az login

#Login to your Azure container registry
az acr login --name <your-registry-name>

#Create an alias of the image
docker tag iris-demo <your-container-image-path>

#Service principal must be created in Azure portal with roles of
    'acrpull' and 'acrpush'
#Push the image to your registry
docker push <your-container-image-path>
```

Code Listing A.9: Build and push docker image to Azure container Registry

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: iris-demo-deployment
spec:
  selector:
    matchLabels:
      app: iris-demo-deployment
  replicas: 1
  template:
    metadata:
      labels:
        app: iris-demo-deployment
    spec:
      containers:
      - name: webservice
        image: <your-container-image-name-path>
        ports:
        - containerPort: 5005
        resources:
            limits:
              cpu: 500m
              memory: 1Gi
      imagePullSecrets:
      - name: acr-secret
```

Code Listing A.10: Deployment manifest

```
apiVersion: v1
kind: Service
metadata:
  name: iris-demo-svc
spec:
  ports:
  - port: 5005
    protocol: TCP
    targetPort: 5005
  selector:
    app: iris-demo-deployment
  type: LoadBalancer
```

Code Listing A.11: Service manifest

```
#Create an image pull secret to authenticate to your registry
kubectl create secret docker-registry <secret-name> \
    —namespace <namespace> \
    —docker-server=<container-registry-name> \
    —docker-username=<service-principal-ID> \
    —docker-password=<service-principal-password>

#Run the iris deployment file
kubectl apply -f deployment.yaml

#Run the iris service file
kubectl apply -f service.yaml

#Check if pod is created & deployment is successful
kubectl get pods
kubectl get deployments

#Get the port for the above exposed service
kubectl get svc

#Test from inside the Kubernetes cluster if the deployment is
    working
curl -i -H "Content-Type:␣application/json" -X POST -d"{\"
    petal_length\":␣7,␣\"sepal_length\":␣4.7,␣\"petal_width\":␣
    1.5,␣\"sepal_width\":␣0.2}" http://<exposed-cluster-ip>:5005/
    score

#To test if the deployment works outside the kubernetes cluster
curl -i -H "Content-Type:␣application/json" -X POST -d"{\"
    petal_length\":␣7,␣\"sepal_length\":␣4.7,␣\"petal_width\":␣
    1.5,␣\"sepal_width\":␣0.2}" http://<public-ip-of-master-node
    >:<exposed-port-service>/score
```

Code Listing A.12: Kubectl commands to deploy iris container and test results

```
#Install kustomize version 3.2 for linux distribution
```

```
wget https://github.com/kubernetes-sigs/kustomize/releases/
    download/v3.2.0/kustomize_3.2.0_linux_amd64
chmod +x kustomize_3.2.0_linux_amd64
mv kustomize_3.2.0_linux_amd64 /usr/local/bin/kustomize

#Verify Kustomize installation
kustomize version

#Install OpenEBS
kubectl apply -f https://openebs.github.io/charts/openebs-
    operator.yaml

#Verify OpenEBS installation
kubectl get storageclass
kubectl get pods -n openebs
kubectl get blockdevice -n openebs
kubectl get sp

#Make a default storage class
kubectl patch storageclass openebs-hostpath -p '{"metadata":{"
    annotations":{"storageclass.kubernetes.io/is-default-class":"
    true"}}}'
```

Code Listing A.13: Install kustomize and OpenEBS

```
#Install standalone Kubeflow pipelines
export PIPELINE_VERSION=1.5.1
kubectl apply -k "github.com/kubeflow/pipelines/manifests/
    kustomize/cluster-scoped-resources?ref=$PIPELINE_VERSION"
kubectl wait --for condition=established --timeout=60s crd/
    applications.app.k8s.io
kubectl apply -k "github.com/kubeflow/pipelines/manifests/
    kustomize/env/platform-agnostic-pns?ref=$PIPELINE_VERSION"

#Installation takes about 10 minutes.
#Verify kubeflow pipelines is installed

#All pods should be running
kubectl get pods -n kubeflow

#All deployments should be ready
kubectl get deployments -n kubeflow

#All services should be exposed
kubectl get svc -n kubeflow

#Persistent volume(PV) and Persistent volume claim(pvc) should
    be in 'Bound status'
kubectl get pv -n kubeflow
kubectl get pvc -n kubeflow

#Add the acr-secret to the service account where the kubeflow
    pipeline experiments will run
kubectl patch serviceaccount pipeline-runner -p '{"
    imagePullSecrets": [{"name": "acr-secret"}]}' -n kubeflow

#Do a port-forwarding for kubeflow pipelines UI, so that it is
    accessible outisde the kubernetes cluster
kubectl port-forward -n kubeflow svc/ml-pipeline-ui 8080:80

#Open a new terminal and ssh to the master node
ssh -L 8080:127.0.0.1:8080 -i <private key path> <public-dns-
    name-of-master-node>.<resource-group-location>.cloudapp.azure
    .com
```

Code Listing A.14: Install standalone Kubeflow Pipelines

```
import json

import argparse
from pathlib import Path

from sklearn import datasets
from sklearn.model_selection import train_test_split

def _download_data(args):

    #Get iris dataset and split into train, test sets
    x, y = datasets.load_iris(return_X_y=True)
    x_train, x_test, y_train, y_test = train_test_split(x, y,
        test_size=0.2)

    #Creates a `iris_data` structure to save and share train and
         test datasets.
    iris_data = {'x_train' : x_train.tolist(),
                 'y_train' : y_train.tolist(),
                 'x_test' : x_test.tolist(),
                 'y_test' : y_test.tolist()}

    # Creates a json object based on `iris_data`
    data_json = json.dumps(iris_data)

    # Saves the json object into a file
    with open(args.data, 'w') as out_file:
        json.dump(data_json, out_file)

if __name__ == '__main__':

    #This component does not receive any input and it only
        outputs one artifact which is `data`.
    parser = argparse.ArgumentParser()
    parser.add_argument('--data', type=str)

    args = parser.parse_args()

    #Creating the directory where the output file will be
        created
    Path(args.data).parent.mkdir(parents=True, exist_ok=True)
    _download_data(args)
```

Code Listing A.15: Load data from a local source - Sklearn

```
FROM python:3.8-slim
WORKDIR /pipeline
COPY requirements.txt /pipeline
RUN pip install -r requirements.txt
```

```
COPY download_data.py /pipeline
```

Code Listing A.16: Dockerfile for loading local data

```
name: Download Data Function
description: Download iris data from sklearn datasets

outputs:
- {name: Data, type: LocalPath, description: 'Path where data will be stored.'}

implementation:
  container:
    image: <your-container-image-path>
    command: [
      python, download_data.py,

      --data,
      {outputPath: Data},
    ]
```

Code Listing A.17: YAML manifest file for loading local data

```python
import json

import argparse
from pathlib import Path

from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier

def _decision_tree(args):

    #Open and reads file "data"
    with open(args.data) as data_file:
        data = json.load(data_file)

    #The excted data format is dictionary, but since the file is
        loaded as a json object, first it needs to be loaded as
        a string and then used as dictionary
    data = json.loads(data)

    x_train = data['x_train']
    y_train = data['y_train']
    x_test = data['x_test']
    y_test = data['y_test']

    #Initialize and train the model
    model = DecisionTreeClassifier(max_depth=6)
    model.fit(x_train, y_train)

    #Get predictions
    y_pred = model.predict(x_test)

    #Get accuracy
    accuracy = accuracy_score(y_test, y_pred)

    #Save output into file
    with open(args.accuracy, 'w') as accuracy_file:
        accuracy_file.write(str(accuracy))

if __name__ == '__main__':

    # Defining and parsing the command-line arguments
    parser = argparse.ArgumentParser(description='Iris training
        description')
    parser.add_argument('--data', type=str)
    parser.add_argument('--accuracy', type=str)

    args = parser.parse_args()

    #Creating the directory where the output file will be
        created (the directory may or may not exist).
    Path(args.accuracy).parent.mkdir(parents=True, exist_ok=True
        )
    _decision_tree(args)
```

Code Listing A.18: Train & Evaluate Decision Tree Model

```
FROM python:3.8-slim
WORKDIR /pipelines
COPY requirements.txt /pipelines
RUN pip install -r requirements.txt
COPY decision_tree.py /pipelines
```

Code Listing A.19: Dockefile for Train & Evaluate Decision Tree Model

```
name: Decision Tree classifier
description: Trains a decision tree classifier

inputs:
- {name: Data, type: LocalPath, description: 'Path where data is
    stored.'}
outputs:
- {name: Accuracy, type: Float, description: 'Accuracy metric'}

implementation:
  container:
    image: <your-container-image-path>
    command: [
      python, decision_tree.py,

      --data,
      {inputPath: Data},

      --accuracy,
      {outputPath: Accuracy},

    ]
```

Code Listing A.20: YAML manifest file for Train & Evaluate Decision Tree Model

```python
import json

import argparse
from pathlib import Path

from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression

def _logistic_regression(args):

    #Open and reads file "data"
    with open(args.data) as data_file:
        data = json.load(data_file)

    data = json.loads(data)

    x_train = data['x_train']
    y_train = data['y_train']
    x_test = data['x_test']
    y_test = data['y_test']

    #Initialize and train the model
    model = LogisticRegression()
    model.fit(x_train, y_train)

    #Get predictions
    y_pred = model.predict(x_test)

    #Get accuracy
    accuracy = accuracy_score(y_test, y_pred)

    #Save output into file
    with open(args.accuracy, 'w') as accuracy_file:
        accuracy_file.write(str(accuracy))

if __name__ == '__main__':

    # Defining and parsing the command-line arguments
    parser = argparse.ArgumentParser(description='Iris LogisticRegression training description')
    parser.add_argument('--data', type=str)
    parser.add_argument('--accuracy', type=str)

    args = parser.parse_args()

    Path(args.accuracy).parent.mkdir(parents=True, exist_ok=True)
    _logistic_regression(args)
```

Code Listing A.21: Train & Evaluate Logistic Regression Model

```
FROM python:3.8-slim
WORKDIR /pipelines
COPY requirements.txt /pipelines
RUN pip install -r requirements.txt
COPY logistic_regression.py /pipelines
```

Code Listing A.22: Dockefile for Train & Evaluate Logistic Regression Model

```
name: Logistic Regression Classifier
description: Trains a Logistic Regression Classifier

inputs:
- {name: Data, type: LocalPath, description: 'Path where data is stored.'}
outputs:
- {name: Accuracy, type: Float, description: 'Accuracy metric'}

implementation:
  container:
    image: <your-container-image-path>
    command: [
      python, logistic_regression.py,

      --data,
      {inputPath: Data},

      --accuracy,
      {outputPath: Accuracy},

    ]
```

Code Listing A.23: YAML manifest file for Train & Evaluate Logistic Regression Model

```
import kfp
from kfp import dsl
from kfp.components import func_to_container_op

@func_to_container_op
def show_results(decision_tree: float, logistic_regression : float) -> None:
    #Given the outputs from decision_tree and logistic regression components the results are displayed.

     print(f"Decision tree metrics: {decision_tree}")

     print(f"Logistic regression accuracy: {logistic_regression}")


@dsl.pipeline(name='Iris Reusable Components Pipeline', description='Applies Decision Tree and Logistic Regression for Iris dataset.')
def iris_pipeline():

    # Loads the yaml manifest for each component
    download = kfp.components.load_component_from_file('download_data/download_data.yaml')
     decision_tree = kfp.components.load_component_from_file('decision_tree/decision_tree.yaml')
     logistic_regression = kfp.components.load_component_from_file('logistic_regression/logistic_regression.yaml')

    #Run download_data task
    download_task = download()

    #Run tasks "decison_tree" and "logistic_regression" given the output generated by "download_task".

     decision_tree_task = decision_tree(download_task.output)

     logistic_regression_task = logistic_regression(download_task.output)

    #Given the outputs from "decision_tree" and "logistic_regression" the component "show_results" is called to print the results.
    show_results(decision_tree_task.output, logistic_regression_task.output)

if __name__ == '__main__':
    kfp.compiler.Compiler().compile(iris_pipeline, '
        IrisResuablePipeline.yaml')
```

Code Listing A.24: Python pipeline for Iris local data

```
from azure.storage.blob import BlobServiceClient
import pandas as pd
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
import numpy as np
import time
import argparse

def _preprocess_data(storage_account_url, storage_account_key,
    container_name,blob_name, local_file):

    """
    Parameters that define from where the data is fetched.
    Here the parameters are for fetching data from a Azure Blob
        Storage
    """
    STORAGEACCOUNTURL= storage_account_url
    STORAGEACCOUNTKEY = storage_account_key
    LOCALFILENAME= local_file
    CONTAINERNAME = container_name
    BLOBNAME = blob_name

    #download from blob
    t1=time.time()
    blob_service_client_instance = BlobServiceClient(account_url
        =STORAGEACCOUNTURL, credential=STORAGEACCOUNTKEY)
    blob_client_instance = blob_service_client_instance.
        get_blob_client(CONTAINERNAME, BLOBNAME, snapshot=None)
    with open(LOCALFILENAME, "wb") as my_blob:
         blob_data = blob_client_instance.download_blob()
         blob_data.readinto(my_blob)
    t2=time.time()
    print(("It takes %s seconds to download "+BLOBNAME) % (t2 -
        t1))

    # LOCALFILENAME is the local file name where the data from
        Azure is copied into
    iris = pd.read_csv(LOCALFILENAME, index_col=False)
    print(iris.head())

    #Encode the Species Label
    encode = LabelEncoder()
    iris.Species = encode.fit_transform(iris.Species)

    train , test = train_test_split(iris, test_size=0.2,
        random_state=0)
    print('shape of training data : ', train.shape)
```

```python
    print('shape of testing data', test.shape)

    X_train = train.drop(columns=['Species'], axis=1)
    y_train = train['Species']
    X_test = test.drop(columns=['Species'], axis=1)
    y_test = test['Species']

    print("Loaded data, now saving it onto the container")
    X_train.to_csv('x_train.csv', index=False)
    X_test.to_csv('x_test.csv', index=False)
    y_train.to_csv('y_train.csv', index=False)
    y_test.to_csv('y_test.csv', index=False)
    print("Data saved")

if __name__ == '__main__':
    print('Preprocessing data...')
     # preprocess_data()

    argument_parser = argparse.ArgumentParser()

    argument_parser.add_argument(
        '--storage_account_url',
        type=str,
        help="Azure URL for storage account"
    )

    argument_parser.add_argument(
        '--storage_account_key',
        type=str,
        help="Storage account key"
    )

    argument_parser.add_argument(
         '--container_name',
        type=str,
        help="Input container name"
    )

    argument_parser.add_argument(
        '--blob_name',
        type=str,
        help="Name of file inside blob container to be loaded"
    )

    argument_parser.add_argument(
        '--local_file',
        type=str,
        help="File name where blob data will be copied into"
    )

    args = argument_parser.parse_args()

    _preprocess_data(args.storage_account_url, args.
        storage_account_key, args.container_name, args.blob_name,
         args.local_file)
```

Code Listing A.25: Load data from an External source

```

FROM python:3.7-slim

WORKDIR /app

RUN pip install -U scikit-learn pandas numpy azure-storage-blob

COPY load_data.py ./load_data.py

ENTRYPOINT [ "python", "load_data.py" ]
```

Code Listing A.26: Dockerfile for loading external data

```python
import argparse
import joblib
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier

def train_model(x_train, y_train):
    print("Entered␣func")
    x_train_data = pd.read_csv(x_train)
    y_train_data = pd.read_csv(y_train)
    print("data␣training␣loaded")

    #define the model
    n_trees = 100
    model = RandomForestClassifier(
        n_estimators=n_trees, oob_score=True, random_state=123
    )
    model.fit(x_train_data, y_train_data)

    joblib.dump(model, 'iris_rf_model.pkl')
    print("Model␣pkl␣made")


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--x_train')
    parser.add_argument('--y_train')
    args = parser.parse_args()
    train_model(args.x_train, args.y_train)
```

Code Listing A.27: Train Random Forest Classifier model

```dockerfile
FROM python:3.7-slim

WORKDIR /app

RUN pip install -U scikit-learn numpy pandas

COPY train.py ./train.py

ENTRYPOINT [ "python", "train.py" ]
```

Code Listing A.28: Dockerfile for training model

```python
import argparse
import joblib
import numpy as np
import pandas as pd
from sklearn.metrics import accuracy_score, recall_score,
    f1_score, precision_score
from sklearn.ensemble import RandomForestClassifier


def test_model(x_test, y_test, model_path):
    x_test_data = pd.read_csv(x_test)
    y_test_data = pd.read_csv(y_test)
    print("Data loaded")
    model = joblib.load(model_path)
    print("Model pkl loaded")
    y_pred = model.predict(x_test_data)
    print("Prediction made")
    print(y_pred)

    print("accuracy : ", accuracy_score(y_test_data, y_pred))

    accuracy = accuracy_score(y_test_data, y_pred)

    metrics = {
        'metrics': [{
            'name': 'accuracy-score',
            'metric_value': accuracy,
            'format': "%",
        }]
    }

    with open('rf_accuracy.txt', 'a') as f:
        f.write(str(accuracy))

    with open('rf_metrics.txt', 'a') as f:
        f.write(str(metrics))

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--x_test')
    parser.add_argument('--y_test')
    parser.add_argument('--model')
    args = parser.parse_args()
    test_model(args.x_test, args.y_test, args.model)
```

Code Listing A.29: Evaluate Random Forest Classifier model

```
FROM python:3.7-slim

```

```
WORKDIR /app

RUN pip install -U scikit-learn numpy pandas

COPY test.py ./test.py

ENTRYPOINT [ "python", "test.py" ]
```

Code Listing A.30: Dockerfile for evaluating model

```
import argparse

def deploy_model(model_path):
    print(f'deploying model {model_path}...')


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--model')
    args = parser.parse_args()
    deploy_model(args.model)
```

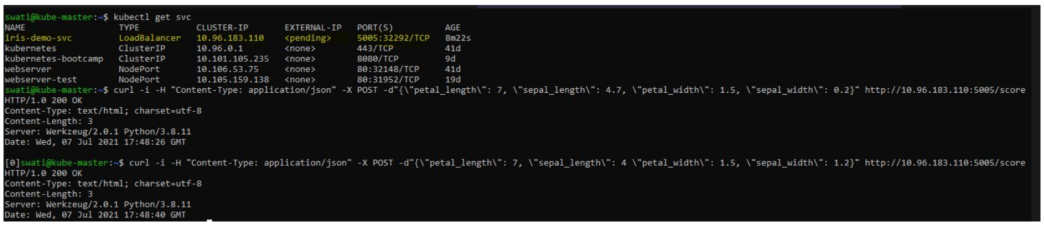
Code Listing A.31: Deploy Random Forest Classifier model

```
FROM python:3.7-slim

WORKDIR /app

COPY deploy_model.py ./deploy_model.py

ENTRYPOINT [ "python", "deploy_model.py" ]
```

Code Listing A.32: Dockerfile for deploying model

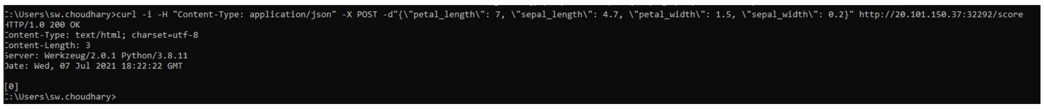
```

import kfp
import kfp.components as comp
from kfp import dsl

def preprocess_op(storage_account_url, storage_account_key,
   container_name,blob_name, local_file):

    return dsl.ContainerOp(
        name='Preprocess Data',
        image=<your-container-image-path>,
        arguments=[
        '--storage_account_url', storage_account_url,
            '--storage_account_key', storage_account_key,
            '--container_name', container_name,
            '--blob_name', blob_name,
            '--local_file', local_file
        ],
        file_outputs={
            'x_train': '/app/x_train.csv',
            'x_test': '/app/x_test.csv',
            'y_train': '/app/y_train.csv',
            'y_test': '/app/y_test.csv',
        }
    )

def train_op(x_train, y_train):

    return dsl.ContainerOp(
        name='Train Model',
        image=<your-container-image-path>,
        arguments=[
            '--x_train', x_train,
            '--y_train', y_train
        ],
        file_outputs={
            'model': '/app/iris_rf_model.pkl'
        }
    )

def test_op(x_test, y_test, model):

    return dsl.ContainerOp(
        name='Test Model',
        image=<your-container-image-path>,
        arguments=[
            '--x_test', x_test,
            '--y_test', y_test,
            '--model', model
        ],
        file_outputs={
            'accuracy': '/app/rf_accuracy.txt',
            'metrics' : '/app/rf_metrics.txt'
        }
    )

def deploy_model_op(model):

    return dsl.ContainerOp(
        name='Deploy Model',
        image=<your-container-image-path>,
        arguments=[
            '--model', model
        ]
    )


def print_op(msg):
    """Print a message."""
    return dsl.ContainerOp(
        name='Print',
        image='alpine:3.6',
        command=['echo', msg],
)


@dsl.pipeline(
    name='Iris dataset Pipeline using RF classifier',
    description='An example pipeline that trains and logs a Random Forest classification model.'
)
def iris_pipeline(storage_account_url, storage_account_key, container_name,blob_name, local_file):
    _preprocess_op = preprocess_op(storage_account_url, storage_account_key, container_name,blob_name, local_file)

    _train_op = train_op(
        dsl.InputArgumentPath(_preprocess_op.outputs['x_train']),
        dsl.InputArgumentPath(_preprocess_op.outputs['y_train'])
    ).after(_preprocess_op)

    _test_op = test_op(
        dsl.InputArgumentPath(_preprocess_op.outputs['x_test']),
        dsl.InputArgumentPath(_preprocess_op.outputs['y_test']),
        dsl.InputArgumentPath(_train_op.outputs['model'])
    ).after(_train_op)


    baseline = 0.7
    with dsl.Condition(_test_op.outputs['accuracy'] > baseline ) as check_condition:
        print_op(f"accuracy is {_test_op.outputs['accuracy']} greater than accuracy baseline {baseline}")
        deploy_model_op(
        dsl.InputArgumentPath(_train_op.outputs['model'])
    ).after(_test_op)

    with dsl.Condition(_test_op.outputs['accuracy'] < baseline) as check_condition:
        print_op(f"accuracy is {_test_op.outputs['accuracy']} less than accuracy baseline {baseline}").after(
                    _test_op)

kfp.compiler.Compiler().compile(iris_pipeline, 'IrisPipeline_RF.
    yaml')
```

Code Listing A.33: Python pipeline for Iris external data



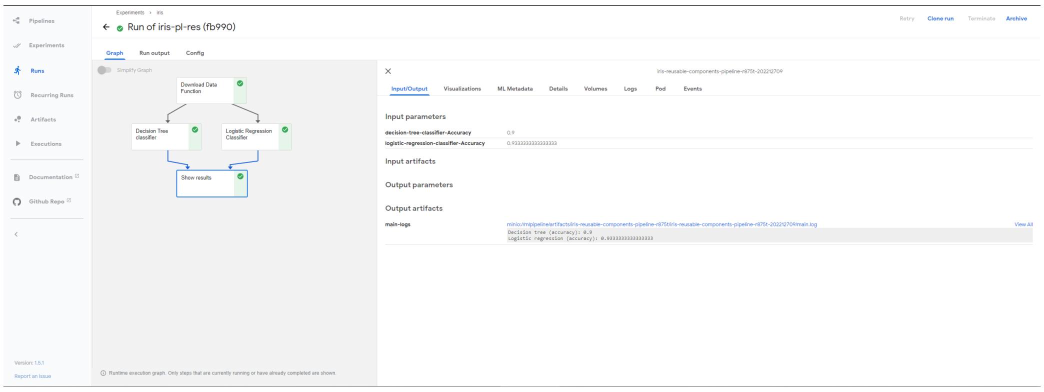
```
swati@kube-master:~$ kubectl get svc
NAME                   TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
iris-demo-svc          LoadBalancer   10.96.183.110    <pending>     5005:32292/TCP   8m22s
kubernetes             ClusterIP      10.96.0.1        <none>        443/TCP          41d
kubernetes-bootcamp    ClusterIP      10.101.105.235   <none>        8080/TCP         9d
webserver              NodePort       10.106.53.75     <none>        80:32148/TCP     41d
webserver-test         NodePort       10.105.159.138   <none>        80:31952/TCP     19d
swati@kube-master:~$ curl -i -H "Content-Type: application/json" -X POST -d"{\"petal_length\": 7, \"sepal_length\": 4.7, \"petal_width\": 1.5, \"sepal_width\": 0.2}" http://10.96.183.110:5005/score
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 3
Server: Werkzeug/2.0.1 Python/3.8.11
Date: Wed, 07 Jul 2021 17:48:26 GMT

[0]swati@kube-master:~$ curl -i -H "Content-Type: application/json" -X POST -d"{\"petal_length\": 7, \"sepal_length\": 4 \"petal_width\": 1.5, \"sepal_width\": 1.2}" http://10.96.183.110:5005/score
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 3
Server: Werkzeug/2.0.1 Python/3.8.11
Date: Wed, 07 Jul 2021 17:48:40 GMT
```

Figure A.1: Output of curl command when testing inside the Kubernetes cluster



```
:\Users\sw.choudhary>curl -i -H "Content-Type: application/json" -X POST -d"{\"petal_length\": 7, \"sepal_length\": 4.7, \"petal_width\": 1.5, \"sepal_width\": 0.2}" http://20.101.150.37:32292/score
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 3
Server: Werkzeug/2.0.1 Python/3.8.11
Date: Wed, 07 Jul 2021 18:22:22 GMT

[0]
:\Users\sw.choudhary>
```

Figure A.2: Output of curl command when testing outside the Kubernetes cluster



Run of iris-pl-res (fb990)

iris-reusable-components-pipeline-r875t-202212709

Input parameters

decision-tree-classifier-Accuracy 0.9
logistic-regression-classifier-Accuracy 0.9333333333333333

Input artifacts

Output parameters

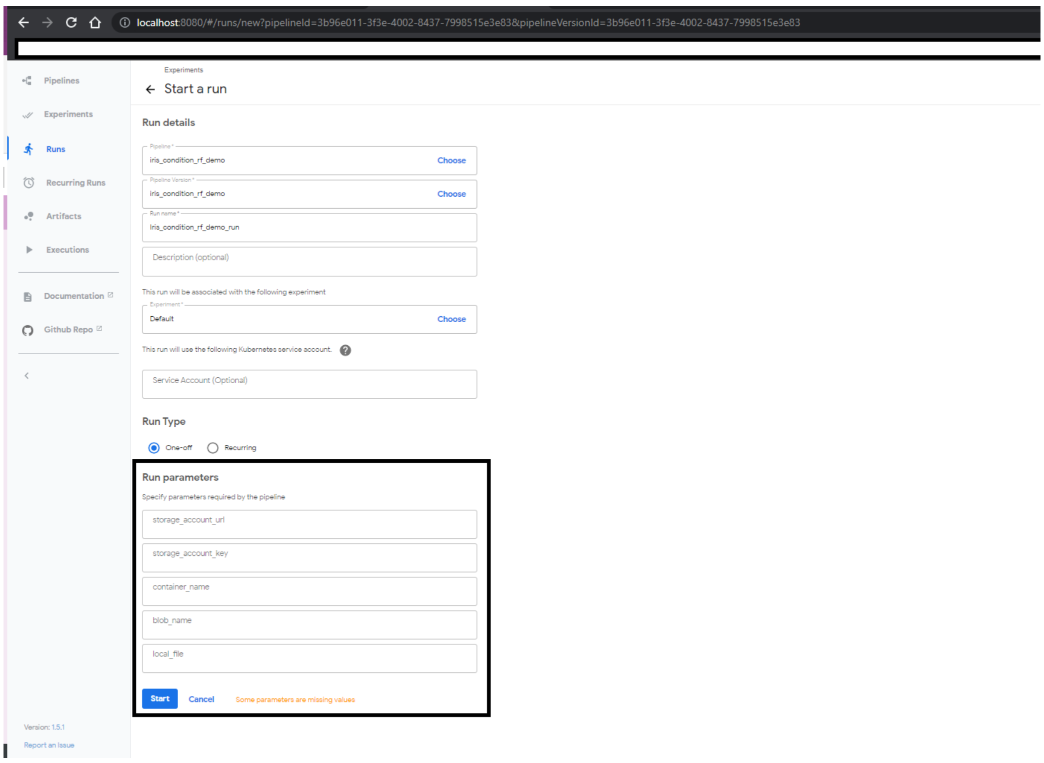
Output artifacts

main-logs minio://mlpipeline/artifacts/iris-reusable-components-pipeline-r875t/iris-reusable-components-pipeline-r875t-202212709/main.log

```
Decision tree (accuracy): 0.9
Logistic regression (accuracy): 0.9333333333333333
```

Runtime execution graph. Only steps that are currently running or have already completed are shown.

Figure A.3: Running Iris (local data) Pipeline with Kubeflow Pipeline UI



localhost:8080/#/runs/new?pipelineId=3b96e011-3f3e-4002-8437-7998515e3e83&pipelineVersionId=3b96e011-3f3e-4002-8437-7998515e3e83

Experiments

Start a run

Run details

Pipeline *	iris_condition_rf_demo
Pipeline Version *	iris_condition_rf_demo
Run name *	Iris_condition_rf_demo_run
Description (optional)	

This run will be associated with the following experiment

Experiment *	Default

This run will use the following Kubernetes service account.

Service Account (Optional)

Run Type

[✓ One-off] Recurring

Run parameters

Specify parameters required by the pipeline

storage_account_url

storage_account_key

container_name

blob_name

local_file

Some parameters are missing values

Figure A.4: Parameters to be entered at run-time

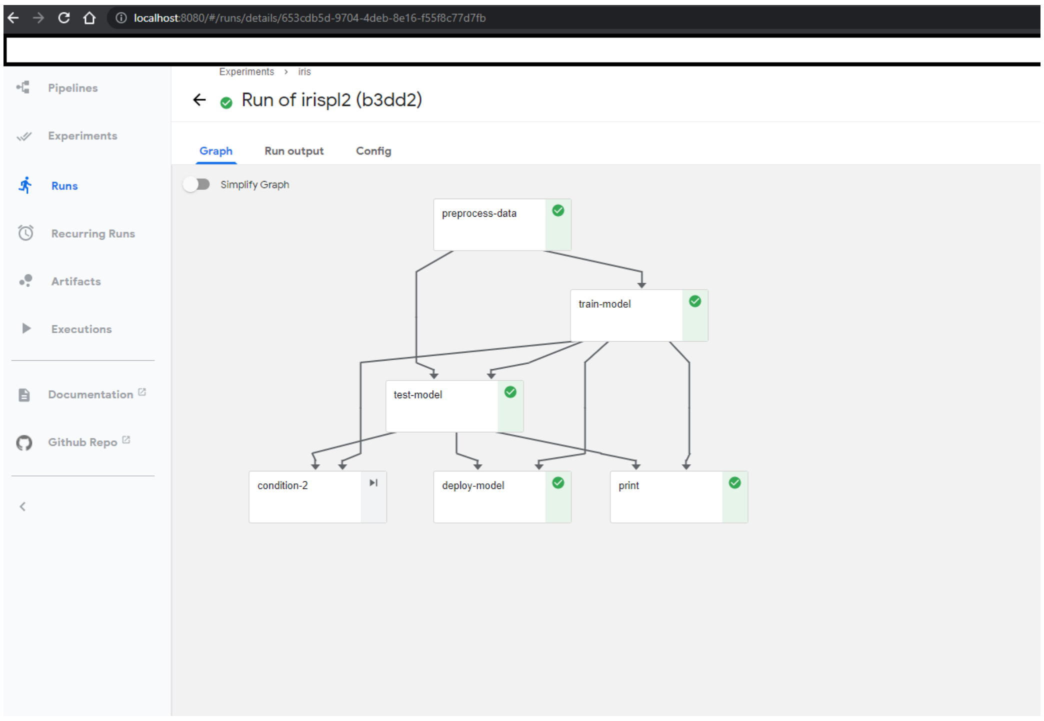


Figure A.5: Running Iris Pipeline with a condition criteria on Kubeflow Pipeline UI