This is the part 2 of the blog series <<Name of the blog with link>>, we would be covering how HPE Ezmeral Unified analytics can be used to perform analytics and machine learning. HPE Ezmeral Unified Analytics gives a streamlined and optimized way to deploy, monitor and manage your machine learning models in any real-world applications. MLOPs on HPE Ezmeral Unified Analytics provides you enterprise grade security, efficient deployment and easy collaboration among various teams.

In our previous blog, we covered Data engineering aspect, streaming pipelines using Spark and data analytics using live Superset dashboards. In the part-2 of the blog we will be covering how we can build Machine Learning models, deploy these models and make them available for production machine learning pipelines. We will be continuing with the Stock price forecasting usecase from our previous blog.

Model Building and Training

Business problem: To forecast stock prices of different companies listed in the National Stock Exchange (NSE) of India.

Step1: Data Gathering

The data is streamed from external servers hosted publicly and then saved to Ezmeral Data Fabric Volume as explained in Blog-1

Step2: Data Preprocessing

The date feature along with open price and close price is taken from the data and the date field is set as the index. The stock market does not function on holidays and weekends, hence there would be no data on those dates. But the model expects continuous data and therefore the missing dates are imputed with the data corresponding to the previous working day.

```
def preprocess_data(df,start_date,end_date):
    #df=df[['Date','Open','Close']]
    df = df.loc[(df["Date"]>=start_date) & (df["Date"]<=end_date)]
    df['Date'] = pd.to_datetime(df['Date'],format = "%Y-%m-%d").dt.date
    df=df.set_index("Date")
    df.index.freq = 'D'

    date_range = pd.DataFrame(pd.date_range(start=start_date, end=end_date),columns=["Date"])
    date_range['Date'] = date_range['Date'].dt.date
    date_range=date_range.set_index("Date")
    date_range.index.freq = 'D'

    df = pd.merge(date_range, df, how='left',left_index=True, right_index=True).fillna(method = 'ffill')
    df.dropna(inplace=True)
    return df
```

Step3: Modelling

The data is divided into training and validation data and a Long Short Term Memory (LSTM) model is trained on the data. LSTM is a variety of recurrent neural networks (RNNs) that are capable of

learning long-term dependencies, especially in sequence prediction problems. The model is then evaluated on the error metric.

```
def model_definition(rows,cols):
    model = Sequential()
    model.add(LSTM(units=60, return_sequences=True, input_shape = (rows,cols)))
    model.add(Dropout(0.1))
    model.add(LSTM(units=60))
    model.add(Dense(2))
    model.compile(loss='mean_squared_error', optimizer='adam', metrics=['mean_absolute_error'])
    return model
```

Model Tracking and Monitoring using MLflow

MLflow is an open-source platform designed to manage the end-to-end machine learning lifecycle. It provides tools for tracking experiments, packaging code, sharing models, and managing deployment. MLflow was developed by Databricks, a company that specializes in big data and AI, and it has gained popularity within the machine learning community for its capabilities in streamlining and organizing various stages of the machine learning workflow.

MLflow consists of several key components:

Tracking: This component allows you to record and compare different experiments, including the parameters, metrics, and artifacts (such as models, visualizations, and data) associated with each experiment. It helps in keeping track of the different iterations and configurations tried during the model development process.

Projects: MLflow Projects provide a way to package code into a reproducible format, allowing you to define and share your machine learning projects with others. This ensures that the code and dependencies used for a particular experiment can be easily reproduced by others.

Models: MLflow's model management capabilities enable you to easily log, version, and deploy machine learning models. It supports various machine learning frameworks like TensorFlow, PyTorch, scikit-learn, etc. You can also deploy models to various deployment platforms directly from MLflow.

Registry: The MLflow Model Registry provides a centralized repository for storing, versioning, and managing models. It allows teams to collaborate on model development and helps maintain a history of different model versions.

UI and APIs: MLflow offers both a web-based user interface and APIs for interacting with its components programmatically. This makes it easy to use and integrate into various machine learning workflows.

Experiments

An experiment is registered in the mlflow tracking service, and different runs care carried out with different parameters and features. The parameters can then be logged in mlflow which can then be viewed on the UI. Different runs are then compared and the model with the best metric is chosen as the model to be moved to production

Firstly, set the experiment name to access MLflow from the jupyter notebook. Then as we have used tensorflow LSTM as our ML model, we have used mlflow.tensorflow.autolog() to log automatically the training parameters, hyperparameters and model metrics.

```
experiment_name = 'stock-exp-1'
mlflow.set_experiment(experiment_name)
mlflow.tensorflow.autolog()
```

Fig 1a: Creating an experiment with auto logging

Once the model is ready, the run details are available via MLflow APIs like search_run(). Best run can be auto-selected as per our requirements.

```
best_run_df = mlflow.search_runs(order_by=['metrics.mean_absolute_error ASC'], max_results=1)
if len(best_run_df.index) == 0:
    raise Exception(f"Found no runs for experiment '{experiment_name}'")

best_run = mlflow.get_run(best_run_df.at[0, 'run_id'])
best_model_uri = f"{best_run.info.artifact_uri}/model"
best_model = mlflow.tensorflow.load_model(best_model_uri)
```

Fig 1b: Selecting the best run of the experiment and loading the model

On the Web UI for MLflow, we can access multiple runs of the ML experiment for different combinations of parameters. The run details can be accessed by clicking on the corresponding Run Name.

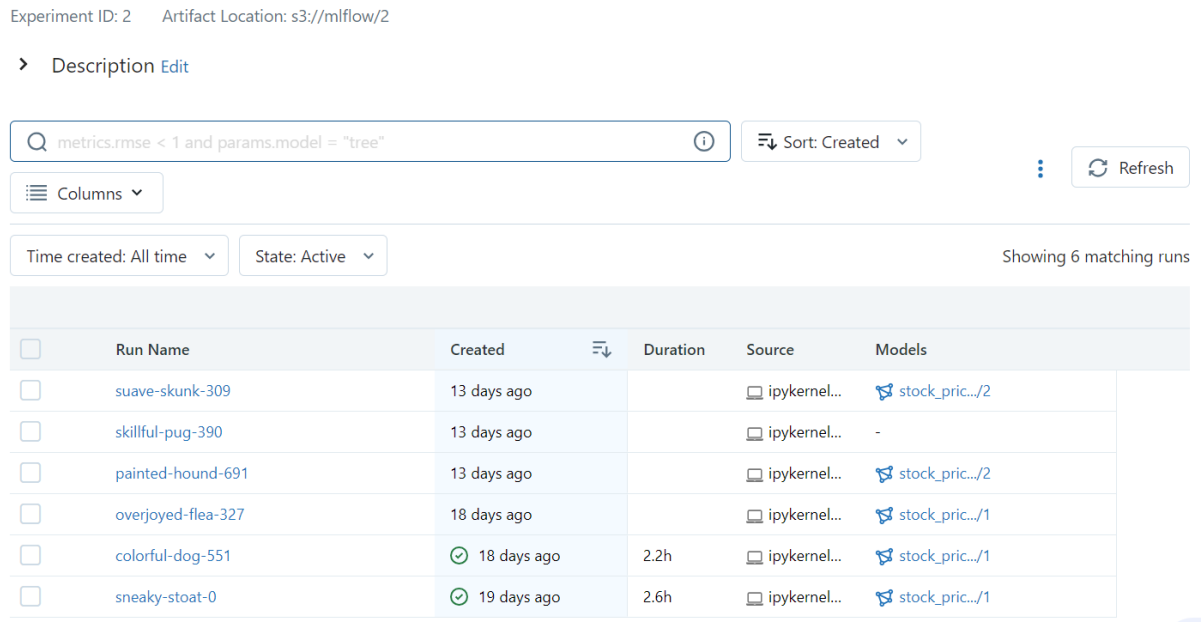


Fig2: Multiple runs are made on the experiment with different parameters in each run

We can also compare various runs, to see how the models perform. Here all the 6 runs in the experiment have been selected and we can see, model with epochs=20, performs best with least loss and MAE.

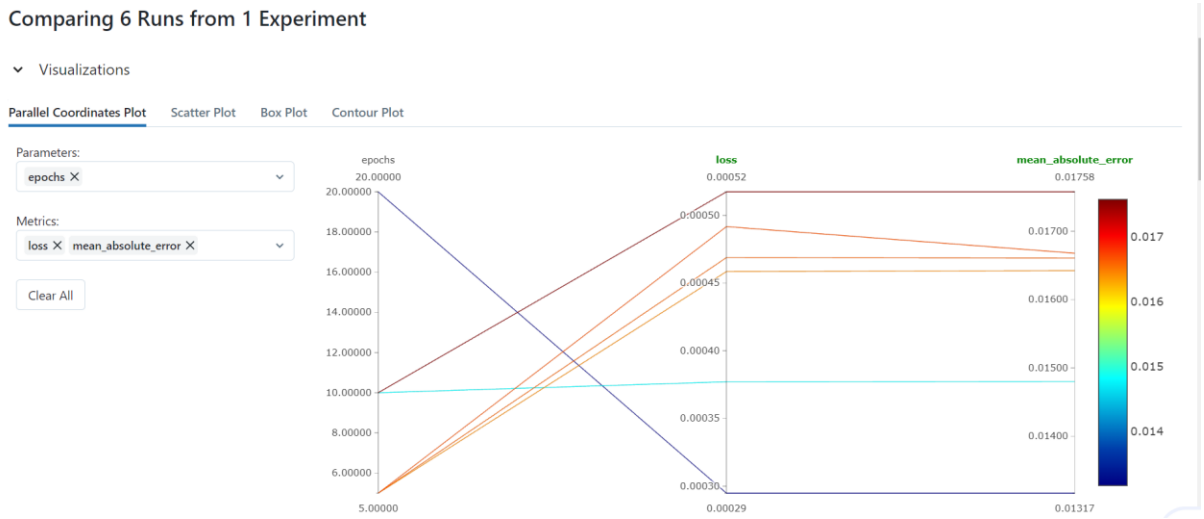


Fig3: All runs are compared with each other based on the relevant metric

The best model can be registered on the MLFlow Web UI or programmatically using mlflow API. The model can be deployed to Staging/Production region based on the model performance.

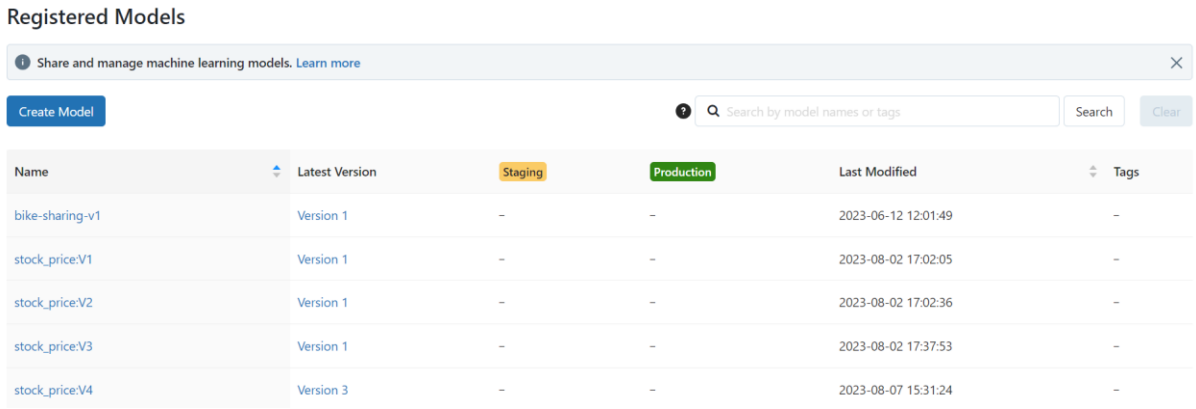
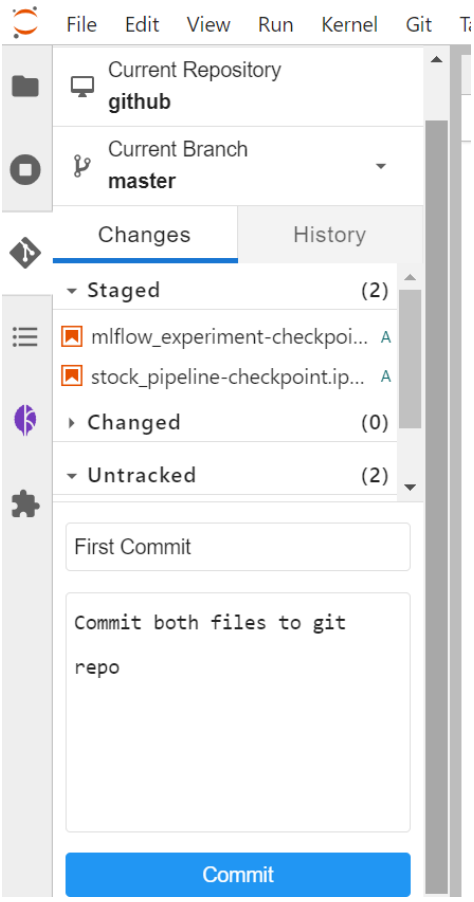


Fig4: Model registry with different registered models

Code Repository

A code repository is a storage location for code and other software development assets, such as documentation, tests, and scripts. They are often used to manage and organize a software project's codebase and collaborate with other project developers.

The source code for the model is stored on GitHub in a private repository. It can then be seamlessly pulled into the IDE using git commands and the updated version can be pushed to different branches of the source code repo.



Model Deployment

The best model which was chosen from the mlflow experiment will be deployed as the next step. Here we are using KServe which is a standard Model Inference Platform on Kubernetes, built for highly scalable use cases.

KServe, also known as "Kubeflow Serving," is a component of the Kubeflow ecosystem designed to facilitate serving machine learning models in Kubernetes environments. Kubeflow itself is an open-source platform for deploying, monitoring, and managing machine learning models on Kubernetes.

KServe provides several features and benefits:

1. Scalability: It is designed to handle serving machine learning models at scale. You can easily scale up or down based on the traffic and load requirements.
2. Multi-Framework Support: KServe supports serving models trained in various machine learning frameworks, such as TensorFlow, PyTorch, scikit-learn, and others.
3. Advanced Deployment Strategies: KServe supports various deployment strategies, including Canary deployments, Blue-Green deployments, and more, allowing you to roll out new model versions gradually and monitor their performance.
4. Monitoring and Metrics: It provides metrics and monitoring capabilities, allowing you to keep track of model performance, errors, and other relevant statistics.
5. Customization: KServe is highly customizable, allowing you to define how you want to serve your models, handle preprocessing and post-processing tasks, and even add custom logic to your serving containers.

6. Integration with Kubeflow Pipelines: If you're using Kubeflow Pipelines for end-to-end machine learning workflows, KServe can be integrated seamlessly to deploy models as part of your pipeline.

Visit the Kserve documentation for more information.

The first step of model deployment is to define the service account with minio object store secret. Then the kserve yaml file for the inference service has to be scripted.

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: kserve-minio-secret
  namespace: user-namespace
secrets:
- name: user-namespace-objectstore-secret
```

Fig 1a: Defining service account with minio secret

```yaml
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  annotations:
    sidecar.istio.io/inject: 'false'
  name: stock-exp-1
  namespace: hpedemo-user01
spec:
  predictor:
    serviceAccountName: kserve-minio-sa
    tensorflow:
      resources: null
      storageUri: s3://mlflow/2/594c725e3ab441e4a5d126589d98d9b9/artifacts/model/data/model
```

Fig 1b: Defining the inference service

Next, a new model server must be created in Kubeflow and this yaml file has to be uploaded. The inference service will now be created and once it is running, we can send our test data through REST APIs and get the response.

```
URL = f"http://stock-exp-1-predictor-default.{os.getenv('JUPYTERHUB_USER')}.svc.cluster.local/v1/models/stock-exp-1:predict"
print(URL)
headers = {"Authorization": f"Bearer {os.environ['AUTH_TOKEN']}"}
inputs = {"inputs": test_seq.tolist()}
response = requests.post(URL, json=inputs, headers=headers)
print(response)
test_predicted = response.json()["outputs"]
```

Model Retraining

The models that are deployed in production are to be constantly monitored to check for inconsistencies and model performance. The models over time can be skewed to the old data on which it was trained and could underperform when the fresh data develops new patterns and characteristics. Hence the models should be retrained whenever the performance drops below a particular threshold. For model retraining, a pipeline is developed which could then be triggered with ease and all the steps of model retraining would happen in a sequential manner.

Kubeflow pipelines are used to carry out this process. Each step of the Kubeflow pipeline runs as a containerised module. The python program needs to be modified to run as a Kubeflow pipeline. The whole cycle of data collection, preprocessing, modelling, etc are divided into separate components. Each process would be isolated from each other and would be running as independent containers inside individual Kubernetes pods. All the dependencies of each task have to be mentioned inside the task due to the isolated nature of the environment. The artifacts that are used in the process like input data files, model object, etc are saved in specific locations and are then referenced by the tasks.

In order to prepare the Kubeflow pipelines, we have to define each task as a function. Firstly, read_data() is defined to read the input file from the data volume on the Kubeflow pipeline.

```
def read_data(output_csv:OutputPath(),
              csv_url: str) -> None:
    import pandas as pd
    df=pd.read_csv(csv_url)
    print(df.head())
    with open(output_csv,"w") as f:
        df.to_csv(f,index=False)
```

Fig1: Defining each task as a function with all the necessary packages inside

Once function is defined its converted into Kubeflow component using create_component_from_func().

```
step_read_data=create_component_from_func(
    func = read_data,
    base_image='python:3.8',
    packages_to_install=['pandas==1.2.4']
)
```

Fig2: Converting each task into a component along with the base image and the packages used

Once each of the tasks are defined, the Kubeflow pipelines are defined to call each of the steps using DSL (Domain Specific Language)

```
@dsl.pipeline(name='Stock Price Prediction Pipeline',description='Stock Price Prediction Pipeline for POC')
def stock_pipeline(csv_url:str,start_date:str,end_date:str,train_percent:float,loss: str,optimizer: str,metrics: str,epochs:int):

    env_var_http = V1EnvVar(name='HTTP_PROXY', value='[redacted]')
    env_var_https = V1EnvVar(name='HTTPS_PROXY', value='[redacted]')

    container_op1 = logg_env_function_op().add_env_variable(env_var_https).add_env_variable(env_var_http)

    #STEP1: Task to read the file from source
    read_data_task = step_read_data(csv_url=csv_url).add_env_variable(env_var_https).add_env_variable(env_var_http)
    #STEP2: Task to preprocess the data
    preprocess_data_task = step_preprocess_data(read_data_task.outputs['output_csv'],
                                            start_date=start_date,
                                            end_date=end_date).add_env_variable(env_var_https).add_env_variable(env_var_http)
    preprocess_data_task.after(read_data_task)
    #STEP3: Task to split the data to training and test
    train_test_task = step_get_train_test_data(preprocess_data_task.outputs['output_csv'],
                                           train_percent=train_percent).add_env_variable(env_var_https).add_env_variable(env_var_http)
    train_test_task.after(preprocess_data_task)
    #STEP4 - a: Task to create the input sequence and output label from the training data
    train_seq_task = step_create_sequence(train_test_task.outputs['train_data']).add_env_variable(env_var_https).add_env_variable(env_var_http)
    train_seq_task.after(train_test_task)
    #STEP4 - b: Task to create the input sequence and output label from the test data
    test_seq_task = step_create_sequence(train_test_task.outputs['test_data']).add_env_variable(env_var_https).add_env_variable(env_var_http)
    test_seq_task.after(train_test_task)
    #STEP5: Task to define and train the model
    task_model_training = step_model_training(train_seq_task.outputs['seq_data'],
                                          train_seq_task.outputs['label_data'],
                                          test_seq_task.outputs['seq_data'],
                                          test_seq_task.outputs['label_data'],
                                          loss = loss,
                                          optimizer = optimizer,
                                          metrics = metrics,
                                          epochs = epochs
                                         ).add_env_variable(env_var_https).add_env_variable(env_var_http)
    task_model_training.after(train_seq_task)
    task_model_training.after(test_seq_task)
    #STEP6: Task to evaluate the model on different metrices
    model_evaluate_task = step_model_evaluate(task_model_training.outputs['trained_model'],
                                          test_seq_task.outputs['seq_data'],
                                          test_seq_task.outputs['label_data'],
                                          preprocess_data_task.outputs['scale_factor']
                                         ).add_env_variable(env_var_https).add_env_variable(env_var_http)
    model_evaluate_task.after(task_model_training)
```

Fig3: Defining the pipeline with all the tasks involved in the appropriate sequence

Once the Kubeflow pipeline is set up, Kubeflow can run the pipeline using Kubeflow Client where you can pass the required arguments and parameters for the specific run.

```
experiment_name = "stock-pipeline "+str(datetime.datetime.now().date())
run_name = stock_pipeline.__name__+'_run'
namespace='hpedemo-user01'

arguments = {"csv_url":"https://raw.githubusercontent.com/snairharikrishnan/test/main/ASIANPAINT.csv",
             "start_date":"2014-01-01",
             "end_date":"2019-12-31",
             "train_percent":"0.80",
             "loss":"mean_squared_error",
             "optimizer":"adam",
             "metrics":"mean_absolute_error",
             "epochs":"5"
            }

client = kfp.Client()
run_result = client.create_run_from_pipeline_func(pipeline_func=stock_pipeline,
                                                  experiment_name=experiment_name,
                                                  run_name=run_name,
                                                  arguments=arguments
                                                 )
```

Fig4: Creating a Kubeflow client and executing the pipeline with appropriate arguments

On the Kubeflow UI, stock_pipeline_run would be triggered. Each step would turn to Green/Red based on the completion status of each step.

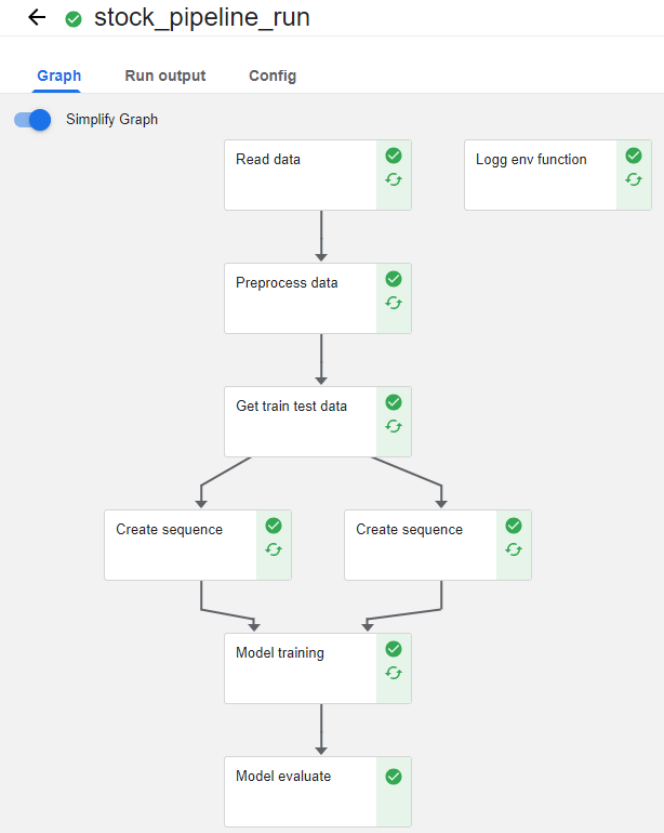


Fig5: Simplified pipeline run

Conclusion

From the above example, HPE Ezmeral Unified Analytics has significantly enhanced the end-to-end pipeline of FSI use cases by providing a comprehensive and Kubernetes-native platform tailored to the specific needs of ML practitioners and data scientists. Here is a conclusion highlighting how HPE Ezmeral Unified Analytics has made a positive impact:

Seamless Integration: HPE Ezmeral Unified Analytics offers seamless integration of various data engineering and ML components like Spark cluster, EzPrestoDB, Superset, MLflow, Kubeflow, etc. This integration streamlines the entire ML workflow, reducing friction between different stages and enabling more efficient development.

Data Exploration and Visualization: By seamlessly integrating Apache Superset into the Ezmeral ecosystem, organizations can harness the power of data-driven insights. Teams can easily access and visualize data, track experiment results, and make data-informed decisions to improve model quality and performance.

Reproducibility: Kubeflow ensures reproducibility by allowing users to define and version their entire ML pipelines. This means that every step, from data preparation to model deployment, can be tracked, versioned, and replicated easily, leading to better model accountability and auditability.

Scalability: Kubernetes, the underlying platform for Kubeflow, excels at scaling applications. Kubeflow leverages this capability to scale ML workloads as needed, handling large datasets and complex models with ease. This scalability is crucial for training and serving models in real-world, production environments.

Model Serving: Kubeflow Serving (KServe) simplifies model deployment and serving. It provides features like advanced deployment strategies, model versioning, and monitoring, making it easier to roll out new models, manage production deployments, and ensure model performance and reliability.

Automated Workflow Orchestration: Kubeflow Pipelines simplifies the orchestration of ML workflows as code. This automation reduces manual effort, enhances reproducibility, and accelerates experimentation, resulting in more efficient model development.