Migrating Subsequent Model Versions to Production

Table of Contents

Introduction 3

Importance of Migrating the Subsequent Model Versions 3

Packaging of ML Models in Containers 4

What are the Containers? 4

Container Internal Image 4

Model Serialization 5

JSON Format (Java Script Object Notation) 6

Pickle 7

ONNX (Open Neural Network Exchange) 8

HDF5 (Hierarchical Data Format version 5) 10

Steps for Machine Learning Model Deployment 12

Subsequent Model Deployment Approaches 13

Recreate Strategy 13

Shadow Evaluation 14

A/B Testing 15

Blue-Green Deployment 16

Canary Testing 17

Choosing the Right Strategy 19

1. Infrastructure 19

2. Robust Validation 19

Active Monitoring and Alerting 19

Appendix A 21

Deploy Model with Blue-Green Strategy (Safe Roll-out Strategy) 21

Pre-requisites 21

1) Prepare the System 21

Define the Endpoint and Deployment 21

2) Create an Online Endpoint 21
3) Create 'Blue' Deployment 22
4) Scale Existing Model to Handle More Traffic 23
5) Deploy the New Model 23
6) Divert the Traffic to New Model 23
Canary Deployment Strategy for Kubernetes 25
Pre-requisites 25
1) Fork the Sample Code from Repository 25
2) Install Prometheus-Operator 26
3) Create Service Connection 26
4) Setup Continuous Integration 26
5) Edit Manifest File 27
6) Setup Continuous Deployment 27
7) Manual Intervention by Promoting or Rejecting Canary 30
8) Deploy a Stable Version 32
9) Initiate Canary Workflow 32
10) Simulate the Request 32
11) Setup Grafana Dashboard 32
Deploy Simple ML Model Using Docker Containers in Azure 34
Pre-requisites 34
1) Download The Source Code 34
2) Check if Liveness and Scoring Requests 34
3) Deploy Online End Point to Azure 35
4) Create an End Point and Deployment 36
5) Invoke the Endpoint 36
6) Delete the Endpoint 36
APPENDIX - B 37
Difference between A/B Testing and Blue-Green Deployment Strategy 37
Difference between Shadow Deployment Strategy and Blue-Green Deployment 37
Difference between Blue-Green Deployment and Canary Deployment 37

Introduction

Machine Learning model life cycle includes following major steps:

1. Data collection
2. Model creation and training
3. Testing and evaluation
4. Deployment and production

Migrating the ML models to production includes deploying the models and evaluating the models on production directly. Model deployment is a process that enables you to integrate machine learning models into production to make decisions on real-world data. It is essentially the second last stage of the ML life cycle before monitoring. Once deployed, the model further needs to be monitored to check whether the whole process of data ingestion, feature engineering, training, testing etc. are aligned properly so that no human intervention is required and the whole process is automatic.

But before deploying the model, one has to evaluate and test if the trained ML model is fit to be deployed into production. The model is tested for performance, efficiency, even bugs, and issues. There are various strategies one can use before deploying the ML model.

For the second or subsequent version, particular planned strategies should be adapted for model deployment based on the usage. As subsequent versions needs to be tested on Production directly to meet the outputs and efficiency of the current models deployed.

Importance of Migrating the Subsequent Model Versions

The importance of migrating model versions lies in the dynamic nature of machine learning models and the evolving requirements of production environments. As models progress through different iterations, incorporating improvements, feature updates, or addressing issues becomes common practice.

Migrating model versions to production ensures that the latest enhancements are deployed, enabling organizations to leverage the full potential of their machine learning investments. This process not only facilitates the adoption of cutting-edge algorithms and methodologies but also addresses the need for continuous improvement in model performance and adaptability to changing data distributions.

Effective model migration is crucial for maintaining model relevance, meeting business objectives, and staying competitive in the rapidly advancing field of machine learning. It allows organizations to harness the latest advancements in data science while maintaining the integrity and reliability of their deployed models.

Packaging of ML Models in Containers

What are the Containers?

Containers in the cloud are hosted in an online environment. Users access them from anywhere. However, application processes or microservices in cloud-based containers remain separate from cloud infrastructure.

Containers can be understood as a Virtual Operating System that runs your application so that it is compatible with any OS.

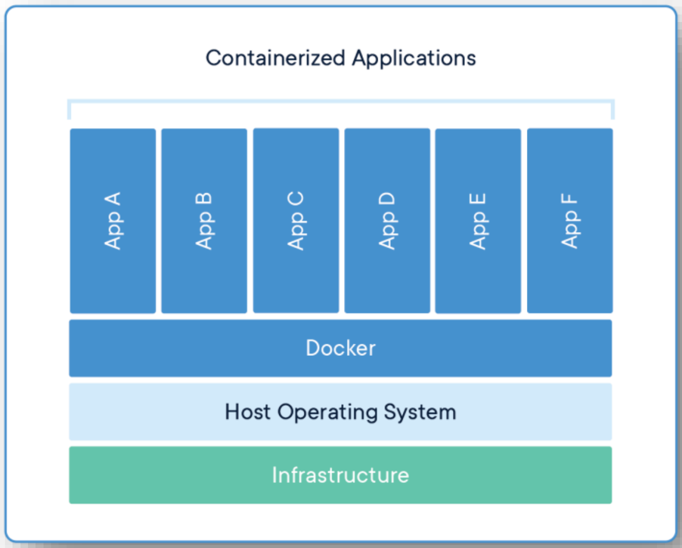
Because the application is not bound to any particular cloud, operating system, or storage space, containerized software can execute in any environment.

Containerization in cloud computing is the process of building software applications for containers. The final product of packaging and designing a container app is a **container image**.

These are the essential elements of a typical container image:

- The application code
- Configuration files
- Software dependencies
- Libraries
- Environment variables

Container Internal Image



Please refer to the section [Deploy Simple ML Model Using Docker Containers in Azure] and Microsoft documentation for detailed implementation of model deployment via Containers.

Model Serialization

Model serialization refers to the process of converting a machine learning model into a format that can be easily stored, transmitted, or reconstructed. Serialization is essential when you want to save a trained model to disk, transfer it over a network, or deploy it in a different environment. The serialized model can later be deserialized, allowing it to be used for predictions, analysis, or further training.

Advantages of model serialization

1. **Persistence:** Serialization allows you to save the state of a trained model to disk. This is crucial for reusing the model without having to retrain it every time. It enables the preservation of the learned patterns and parameters.

2. **Deployment:** Serialized models can be easily deployed in production environments. This is particularly important for web applications, mobile apps, or any system where the model needs to make real-time predictions.

3. **Interoperability:** Serialized models can be shared and used across different programming languages or platforms. This is important when, for example, training a model in Python and using it in a Java-based application.

4. **Scalability:** Serialized models can be distributed and deployed across a distributed computing environment or cloud infrastructure. This supports scalability when dealing with large datasets or complex models.

5. **Versioning:** Serialized models can be versioned, making it easier to manage different iterations or improvements of the model. This is valuable in collaborative or team-based machine learning projects.

6. **Transfer Learning:** Serialized models can serve as a starting point for transfer learning. Transfer learning involves taking a pre-trained model and fine-tuning it on a new, related task. Serialization simplifies the process of using a pre-trained model.

7. **Reduced Training Time:** By serializing models, you can avoid the need to retrain them from scratch every time they are used. This can significantly reduce the overall time and computational resources required for model deployment.

Popular available formats for JSON, Pickle, and specialized formats like ONNX (Open Neural Network Exchange), HDF5, Keras for deep learning models. Other available formats are XML, dill etc. [Reference- Link]

JSON Format (Java Script Object Notation)

1. JSON is a lightweight data interchange format.
2. It is easy for humans to read and write and easy for machines to parse and generate.
3. JSON is commonly used for configuration files, data exchange between a server and a web application, and more.

Key Features

1. **Readability:** JSON is easy to read and write for both humans and machines.
2. **Simplicity:** It supports key-value pairs and nested structures, making it a straightforward data format.

Sample Code: Following code shows both, loading and saving of the model to and from a JSON file. (Reference: Link)

1. For saving the model in JSON format, to_json() method is used.
2. And Keras provide, model_from_json() method to load the model model from a JSON file.

```
# MLP for Pima Indians Dataset Serialize to JSON and HDF5
from tensorflow.keras.models import Sequential, model_from_json
from tensorflow.keras.layers import Dense
import numpy
import os
# fix random seed for reproducibility
numpy.random.seed(7)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Fit the model
model.fit(X, Y, epochs=150, batch_size=10, verbose=0)
# evaluate the model
scores = model.evaluate(X, Y, verbose=0)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))

# serialize model to JSON
model_json = model.to_json()
```

```
with open("model.json", "w") as json_file:
    json_file.write(model_json)
# serialize weights to HDF5
model.save_weights("model.h5")
print("Saved model to disk")



# load json and create model
json_file = open('model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)
# load weights into new model
loaded_model.load_weights("model.h5")
print("Loaded model from disk")

# evaluate loaded model on test data
loaded_model.compile(loss='binary_crossentropy', optimizer='rmsprop',
metrics=['accuracy'])
score = loaded_model.evaluate(X, Y, verbose=0)
print("%s: %.2f%%" % (loaded_model.metrics_names[1], score[1]*100))
```

Pickle

1. Pickle is a Python-specific serialization format.
2. It allows one to serialize and deserialize Python objects, including complex data structures and custom classes.
3. Pickle is commonly used for saving and loading trained machine learning models in Python.
4. Most preferred extension while using pickle modules is .pkl. However, other extensions like .pickle, .pk1, .p etc.

Key Features

1. **Python-Specific:** Pickle is specific to Python and may not be as interoperable as other formats across different programming languages.
2. **Object Serialization:** It supports the serialization of complex Python objects, preserving their structure.

Sample Code: Following code shows storing and loading of statistical model with pickle format. Pickle library modules now are by default provided in Python newer versions and can directly be imported with just import statement. [Reference: Link]

1. dump() method of Pickle module is used to save a model.
2. To load a model from serialized format using Pickle, load() method is used.

```python
from tensorflow.keras.models import Sequential, model_from_json
from tensorflow.keras.layers import Dense
import numpy
import os
import pickle

# fix random seed for reproducibility
numpy.random.seed(7)
# load pima indians dataset
dataset = numpy.loadtxt("/dbfs/mnt/ingestionpoc/PlayBookHelper/pima-indians-
diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Fit the model
model.fit(X, Y, epochs=150, batch_size=10, verbose=0)
# evaluate the model
scores = model.evaluate(X, Y, verbose=0)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))

filename = 'finalized_model.pkl'
pickle.dump(model, open(filename, 'wb'))

# some time later...

# load the model from disk
loaded_model = pickle.load(open(filename, 'rb'))
result = loaded_model.evaluate(X, Y,verbose=0)
print(result)
```

ONNX (Open Neural Network Exchange)

1. ONNX is an open format for representing deep learning models.
2. It was developed to provide a standard for interoperability between different deep learning frameworks.

3. ONNX allows you to export trained models from one deep learning framework and import them into another that supports the ONNX format.
4. It is developed by Microsoft and Facebook with aim to generalize different machine learning models usage.

Key Features

1. **Interoperability:** ONNX enables the exchange of models between various frameworks, such as PyTorch, TensorFlow, and others.
2. **Efficiency:** Since models can be trained in one framework and deployed in another, it can save time and resources.
3. **Broad Ecosystem:** ONNX has gained support from multiple deep learning frameworks and hardware vendors.

Sample Code: Following code shows the saving of tensorflow keras model in ONNX format.

1. Tensorflow model needs to be converted to ONNX format. Using tf2onnx.convert.from_keras() function, the model is converted to ONNX format.
2. To reload the model and start the inference, onnxruntime.InferenceSession() function can be used.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow.keras.models import Sequential,load_model
from tensorflow.keras.layers import Dense
import onnxruntime as rt
import numpy as np
import tf2onnx

df = pd.read_csv('/dbfs/mnt/ingestionpoc/PlayBookHelper/pima-indians-
diabetes.csv')


X = df.to_numpy()[:,0:8]
Y = df.to_numpy()[:,8]
seed = 42
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size = 0.25,
random_state = seed)
#print (f'Shape of Train Data : {X_train.shape}')
#print (f'Shape of Test Data : {X_test.shape}')

model = Sequential([
    Dense(24, input_dim = (8), activation = 'relu'),
    Dense(12, activation = 'relu'),
    Dense(1, activation = 'sigmoid'),
])

model.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics =
['accuracy'])
history = model.fit(X_train, y_train, epochs=150, batch_size=32, verbose = 1)
scores = model.evaluate(X_test, y_test)
print (f'{model.metrics_names[1]} : {round(scores[1]*100, 2)} %')


#Convert and save a model to onnx
tf2onnx.convert.from_keras(model, output_path='model.onnx')

#Load and infer from a saved onnx model

session = rt.InferenceSession('model.onnx')
x = np.array([[2,112,66,22,0,25,0.307,24]], dtype=np.float32)
inputDetails = session.get_inputs()
y = session.run(None, {inputDetails[0].name: x})
print(y[0])
```

HDF5 (Hierarchical Data Format version 5)

1. HDF5 is a file format and set of tools for managing complex data, widely used in scientific and machine learning applications.
2. It provides a hierarchical structure for organizing and storing large amounts of data, including metadata and attributes.
3. HDF5 is known for its efficiency in handling large datasets and supporting parallel I/O operations.

Key Features

1. Hierarchical Structure: HDF5 allows data to be organized in a hierarchical structure, making it suitable for complex data representations.
2. Compression and Chunking: HDF5 supports data compression and chunking, which helps in efficiently storing and retrieving large datasets.
3. Parallel I/O: HDF5 is designed to support parallel I/O operations, making it scalable for handling data in parallel computing environments.

Sample Code: Following code shows saving of model using HDF5 format.

1. Method .save() provided by Keras can directly be used to save a model.
2. To load the model load_model() method is used.

3. Note: .h5 format is considered as a legacy format and keras recommends to save the model in .keras format.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow.keras.models import Sequential,load_model
from tensorflow.keras.layers import Dense

df = pd.read_csv('/dbfs/mnt/ingestionpoc/PlayBookHelper/pima-indians-
diabetes.csv')


X = df.to_numpy()[:,0:8]
Y = df.to_numpy()[:,8]
seed = 42
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size = 0.25,
random_state = seed)
#print (f'Shape of Train Data : {X_train.shape}')
#print (f'Shape of Test Data : {X_test.shape}')

model = Sequential([
    Dense(24, input_dim = (8), activation = 'relu'),
    Dense(12, activation = 'relu'),
    Dense(1, activation = 'sigmoid'),
])

model.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics =
['accuracy'])
history = model.fit(X_train, y_train, epochs=150, batch_size=32, verbose = 1)
scores = model.evaluate(X_test, y_test)
print (f'{model.metrics_names[1]} : {round(scores[1]*100, 2)} %')


#Saving the model in Keras
model.save("model.h5")

#re-loading of model
model=load_model("model.h5")
scores = model.evaluate(X_test, y_test)
print (f'{model.metrics_names[1]} : {round(scores[1]*100, 2)} %')
```

Steps for Machine Learning Model Deployment

1. **Choose Deployment Environment:** Select the deployment environment based on your requirements. Common options include cloud platforms (e.g., AWS, Azure, Google Cloud), on-premises servers, or edge devices.
2. **Model Serialization:** Serialize your trained model into a format that can be easily loaded by the deployment environment. Common serialization formats include HDF5, ONNX, or the native format of the machine learning framework you used (e.g., TensorFlow SavedModel, PyTorch model).
    - Please refer to the section [Packaging of ML Models in Containers] for detailed information.
3. **Create an Inference API or Endpoint:** Build an API or endpoint that exposes your model for making predictions. This could be implemented using a web framework (e.g., Flask, Django) or a cloud-based service (e.g., AWS Lambda, Azure Functions).
    - Please refer to the section [Appendix A] for the more details on creation of endpoint and deployment.
4. **Set up Model Monitoring:** Implement monitoring to track the performance of your deployed model over time. Monitoring can include tracking input data distributions, model accuracy, response times, and other relevant metrics.
5. **Handle Model Versioning:** Establish a versioning system for your models to facilitate updates and rollbacks. This is crucial for managing changes and improvements to the model without disrupting production services.
6. **Security Considerations:** Implement security measures to protect your deployed model. This may include encryption, authentication, and authorization mechanisms to control access to the model.
7. **Testing:** Thoroughly test your deployed model to ensure that it performs as expected in the production environment. Consider using a staging or testing environment before deploying to production.
8. **Documentation:** Create documentation that outlines how to use the deployed model, including information on input and output formats, API endpoints, and any other relevant details.
9. **Deploy to Production:** Deploy your model to the production environment. This may involve uploading your serialized model to a cloud service, deploying it on a server, or pushing it to edge devices.

Subsequent Model Deployment Approaches

Once the Machine Learning model is retrained with new/updated data, it can not be directly deployed to production as it may affect the traffic hitting to the current version of the models already deployed to Production. Specific strategies are required to be adopted to deploy and activate this model to production again.

Based on the traffic management done, following are the approaches available:

1. Recreate strategy
2. Shadow evaluation
3. A/B testing
4. Blue-green deployment
5. Canary deployment

Following are different deployment strategies which can be used to deploy the subsequent model versions,

Recreate Strategy

Approach/Method Details

1. The Recreate strategy is a fundamental approach where the current model is taken down, and a new/updated version of the model is deployed in its place.
2. This method involves a period of model downtime during the transition, which may have implications in scenarios with high user interaction.

Use Case

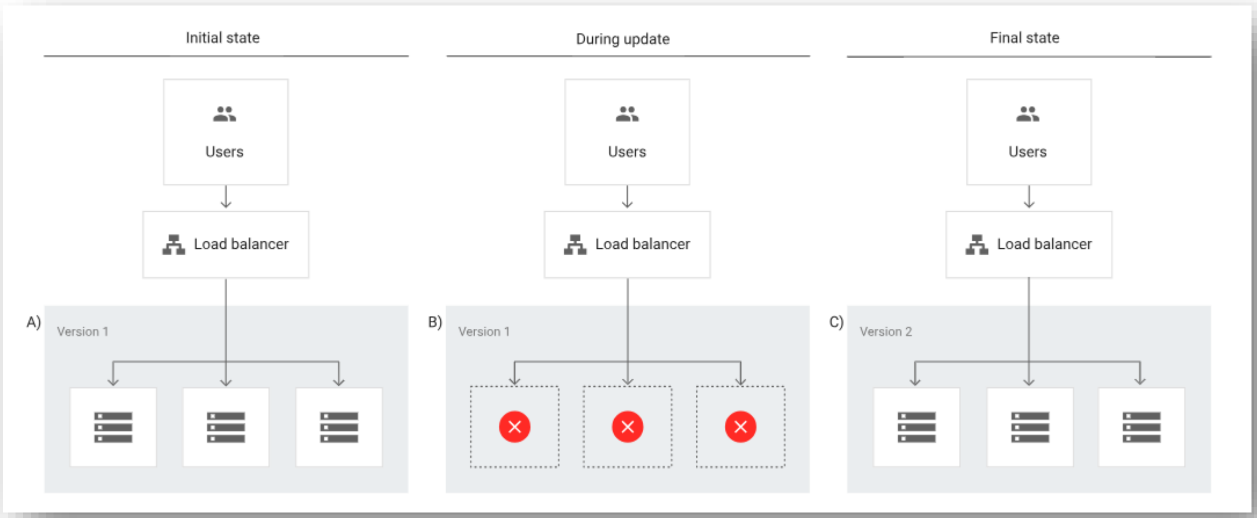
1. Suited for situations where a straightforward model replacement is acceptable and there is no requirement to maintain the existing model during the transition.
2. Best applied when the impact of model downtime on user interaction is minimal or manageable.

When to Use

1. Ideal for scenarios where a clean transition from the current model to the new version is acceptable, and temporary model downtime can be tolerated.
2. May not be suitable for applications with high user interaction where continuous model availability is critical.

Considerations

1. Users should be informed in advance about potential downtime during the model transition.
2. Assess the impact on user experience and system requirements to determine if temporary downtime aligns with the application's objectives.



Shadow Evaluation

Approach/Method Details

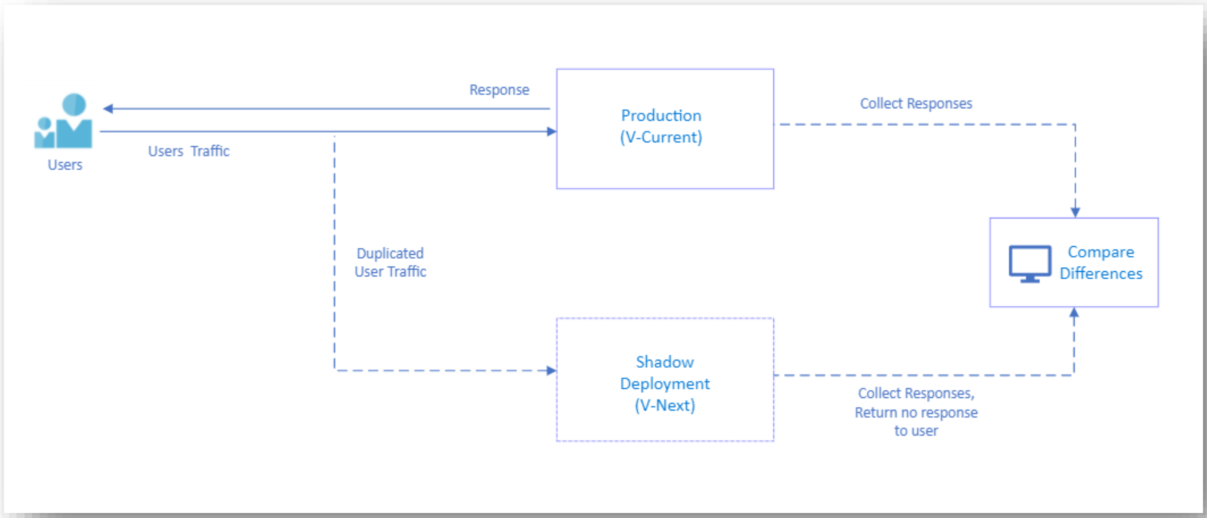
1. In the Shadow Deployment approach, a separate model is deployed parallel to the original deployment.
2. Incoming traffic requests are directed to both the newly deployed model and the old one. However, new model/shadow model is only considered for evaluation and response from the current model is returned to the user.
3. This strategy facilitates the evaluation of requests, model measurement metrics, and enables comparison with the output of the original model.

Use Case

1. This method allows real-time evaluation of the model using live data.
2. The output format can be validated and compared against the existing output structure and formats.
3. End-to-End pipelines can be measured and evaluated for effectiveness.

When to Use

1. This strategy is particularly effective when the model does not require user action for validation.
2. It is suitable for scenarios where comprehensive evaluation of requests, metrics, and comparison with an existing model's output are essential for ensuring deployment success.



[Source: Link]

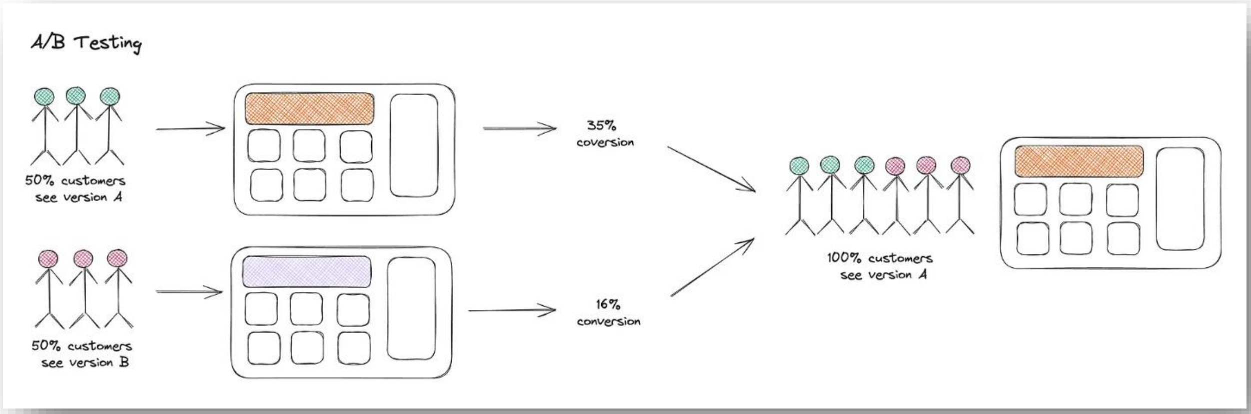
As given in the image above,

1. All user requests are sent to both the models in parallel manner.
2. Production (V-current) version of model is the old/deployed model and Shadow Deployment (V-Next) model is the updated version of model.
3. Output of the Production (V-current) model is returned to the user.
4. Output generated from the shadow models are compared with output generated from the current model for sanity, formatting, and sentiments etc.

A/B Testing

Approach/Method Details

1. A/B testing is a data-based method used to compare two models by setting them up in parallel to cater to different features on the same data.
2. In this approach, two models are concurrently deployed, each addressing distinct aspects of the data.
3. It is commonly employed in User Experience research to identify which model is more suitable for users and which model yields better performance.
4. It is also known as split testing. Users are divided into two or more groups, with each group experiencing a different version of the software. Metrics are then collected and analyzed to determine which version performs better in terms of user engagement, conversion rates, or other relevant key performance indicators.


A/B Testing
50% customers see version A
35% coversion
50% customers see version B
16% conversion
100% customers see version A

[Source: Link]

Use Case

1. A/B testing is widely used in Recommendation systems, especially for scenarios where multiple models offer different features.
2. For instance, one model may provide general recommendations, while another model is designed for localized regional details and languages.
3. Engineers can leverage A/B testing to identify the preferred model by users over time and take appropriate steps based on user preferences.
4. Example 1: To compare which product performs better.
5. Example 2: To identify which soil type supports better seed germination in agriculture.
6. Example 3: Setting price for a product, which one yields high profits or which one leads to more new customer.
7. **Real Life Example:** Bing conducted an A/B test in which they changed the way ad headlines were shown in the Bing search engine. This little experiment resulted in a revenue gain of 12% or more than $100 million per year in the United States alone.

When to Use:

1. A/B testing is particularly effective in situations where multiple models cater to different aspects of the same data, and user preferences need to be determined.
2. It is a valuable method when deploying models with varying features, such as language preferences or regional details, and the goal is to identify the most preferred model based on user interactions.

Blue-Green Deployment

Approach/Method Details

1. The Blue-Green Deployment method involves using two separate identical environments, referred to as Blue and Green.

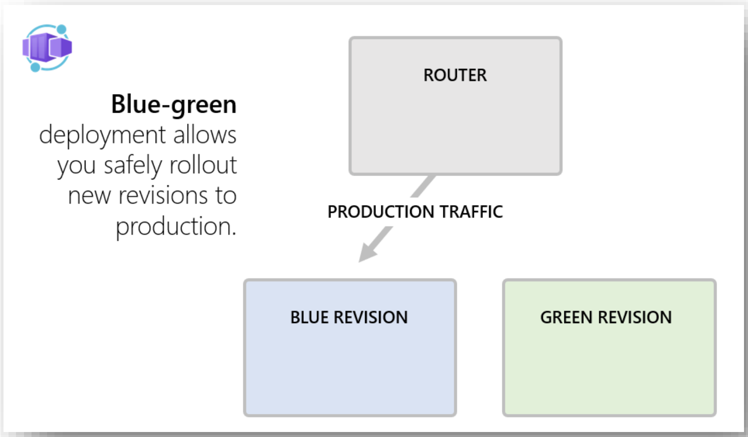
2. In this context, Blue represents the current model version, while Green represents the updated model version.
3. The method requires additional computational power and infrastructure for the deployment and maintenance of separate clusters.
4. One new deployment is tested, full traffic is diverted at once to the Green deployment.

Use Case

1. Ideal for scenarios where minimizing system downtime and mitigating risks associated with deploying a new model to production are critical objectives.
2. Enables a seamless transition between the current and updated model versions, allowing thorough testing before full deployment.

When to Use

1. This approach is well-suited for situations where maintaining operational continuity is essential.
2. Effective when the deployment process involves substantial changes, ensuring that the new model is thoroughly tested and verified in the Green environment before directing full traffic to it.


Blue-green deployment allows you safely rollout new revisions to production.
ROUTER
PRODUCTION TRAFFIC
BLUE REVISION
GREEN REVISION

[Source: Link]

Canary Testing

Approach/Method Details

1. The Gradual Deployment method focuses on gradually enabling an increasing number of users to the new model version.
2. In this approach, the new version is deployed alongside the current versions, and partial real-user traffic is directed to the new model version.
3. Initially, a percentage of users, typically 10-20%, interact with the new model. Depending on successful testing, traffic to the new model can be gradually increased, or it can be taken down if it fails to meet testing requirements without additional downtime.

Use Case

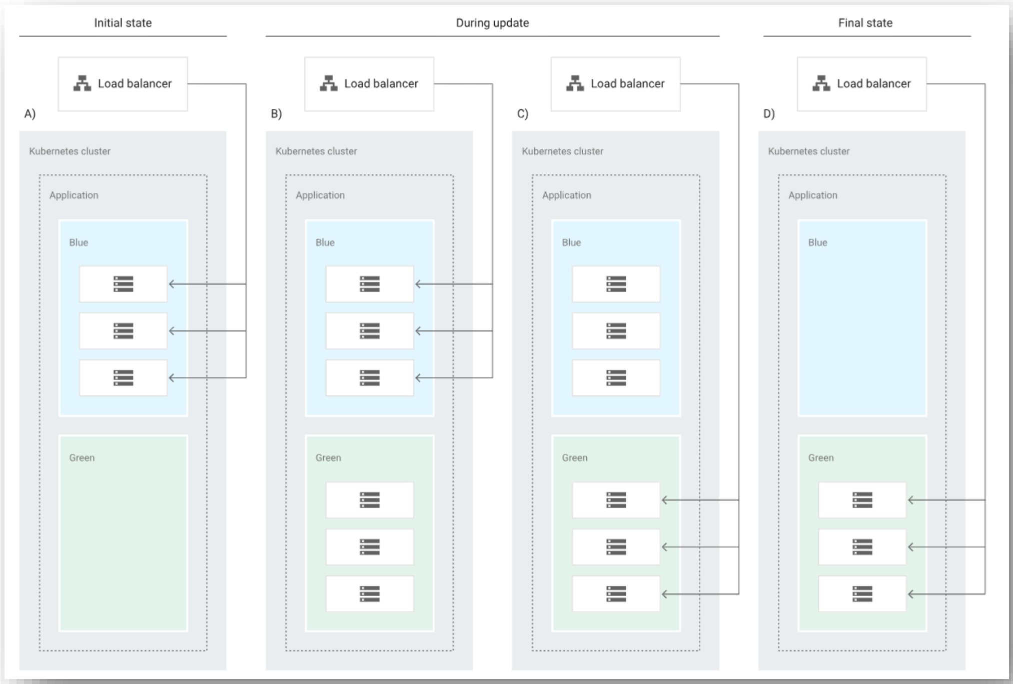
1. Suited for scenarios where a cautious, step-by-step deployment strategy is preferred, allowing for careful monitoring of the new model's performance and user impact.
2. Provides flexibility to scale the new model's user traffic based on successful testing outcomes and user feedback.

When to Use

1. Effective when the deployment process requires careful consideration of user impact and performance evaluation.
2. Particularly useful for scenarios where the risk of introducing a new model needs to be mitigated by observing its behavior with a subset of users before expanding to the entire user base.

Deployment Variations

- **Canary Rolling Deployment:** Involves replacing a small number of patches within a single cluster or instance.
- **Canary Parallel Deployment:** Deploys separate instances in parallel to the current one. The choice between these two depends on the available infrastructure and computational power.



Choosing the Right Strategy

1. Infrastructure

a. ML models have significant resource requirements, including computational power, storage, and data transfer speeds.

b. Failure to factor in these requirements during ML model deployment can pose a high risk and lead to project failure or issues later.

c. Efficient model deployment involves considering cloud platforms such as AWS, Azure, and Google Cloud, which provide scalable solutions adaptable to changing workloads.

d. Containerization technologies like Docker and orchestration tools like Kubernetes simplify deployment across diverse environments and should be considered in the deployment process.

e. It is crucial to align infrastructure with the model's requirements and organizational needs for long-term efficiency and scalability in model deployment.

2. Robust Validation

a. Thorough testing and validation are essential steps before deploying an ML model to ensure its effectiveness in real-world conditions.

b. Requirements and data models can cover various scenarios, necessitating the need for comprehensive testing to ensure the model behaves as expected.

c. Techniques such as cross-validation, exploratory data analysis, holdout testing, and A/B testing are employed to assess model performance and reliability.

d. The results of these tests provide insights for data engineers and MLOps teams to make crucial decisions on improving model robustness, maintaining output quality, and ensuring scalability during deployment.

Active Monitoring and Alerting

1. Beyond AI model deployment, effective management and monitoring post-deployment are crucial challenges in the machine learning lifecycle.
2. ML model management should include continuous monitoring and alerting systems to address ongoing concerns.
3. Continuous monitoring plays a key role in identifying deviations from expected behavior and capturing data shifts that impact the model's accuracy.
4. Establishing alerting mechanisms is recommended to notify relevant stakeholders promptly about any issues or deviations.

5. Proactive data governance through alerting enables timely interventions, ensuring model effectiveness, and facilitating retraining or fine-tuning as necessary.

Appendix A

Deploy Model with Blue-Green Strategy (Safe Roll-out Strategy)

Pre-requisites

1. Azure CLI should be installed on the System.

1) Prepare the System

1. Setup required Parameters for the Azure CLI.

```
az account set --subscription <subscription id>
az configure --defaults workspace=<Azure Machine Learning workspace name>
group=<resource group>
```

2. Clone the required repository.

```
git clone --depth 1 https://github.com/Azure/azureml-examples
cd azureml-examples
cd cli
```

Define the Endpoint and Deployment

2) Create an Online Endpoint

1. Set Endpoint's name using following code.
   a. Endpoint name should be unique within Azure region.

```
export ENDPOINT_NAME="<YOUR_ENDPOINT_NAME>"
```

2. Save the following code content in a file named endpoint.yml.

```
$schema:
https://azuremlschemas.azureedge.net/latest/managedOnlineEndpoint.schema.json
name: my-endpoint
auth_mode: key
```

3. Run following command to configure the YML file.

```
export ENDPOINT_NAME="<YOUR_ENDPOINT_NAME>"az ml online-endpoint create --name
$ENDPOINT_NAME -f endpoint.yml
```

3) Create 'Blue' Deployment

1. Create a file named blue-deployment.yml.
2. Add the following code with updated/required configurations.

```
$schema:
https://azuremlschemas.azureedge.net/latest/managedOnlineDeployment.schema.json
name: blue
endpoint_name: my-endpoint
model:
  path: ../../model-1/model/
code_configuration:
  code: ../../model-1/onlinescoring/
  scoring_script: score.py
environment:
  conda_file: ../../model-1/environment/conda.yaml
  image: mcr.microsoft.com/azureml/openmpi4.1.0-ubuntu20.04:latest
instance_type: Standard_DS3_v2
instance_count: 1
```

3. Run the following command to create blue deployment.

```
az ml online-deployment create --name blue --endpoint-name $ENDPOINT_NAME -f
endpoints/online/managed/sample/blue-deployment.yml --all-traffic
```

4. Validate the current deployment.
5. Invoke the end point.
    a. Invoking can be done by specifying the deployment name or without specifying the deployment name.
    b. If we invoke the endpoint by specifying the deployment name, Azure ML will route full traffic to that.
    c. If we invoke the endpoint without specifying the deployment name, the request then will be handled by the Traffic Controller.
    d. Traffic Controller setting maintains percentage traffic that will be diverted towards one particular end point.
    e. We can list the status of existing endpoint and deployment by following code.

```
az ml online-endpoint show --name $ENDPOINT_NAME

az ml online-deployment show --name blue --endpoint $ENDPOINT_NAME
```

    f. Use following code to invoke the endpoint for testing the sample data.

```
az ml online-endpoint invoke --name $ENDPOINT_NAME --request-file
endpoints/online/model-1/sample-request.json
```

4) Scale Existing Model to Handle More Traffic

To scale the existing model to handle more traffic, we need to increase the instances allotted to that endpoint. Following code can be used to increase or change the number of instances associated with the endpoint.

```
az ml online-deployment update --name blue --endpoint-name $ENDPOINT_NAME --set
instance_count=2
```

5) Deploy the New Model

1. Create a new deployment named ‘green’.
    a. Create a file name green-deployment.yml.
    b. Refer to the format given in the section [Create an Online Endpoint].

```
az ml online-deployment create --name green --endpoint-name $ENDPOINT_NAME -f
green-deployment.yml
```

    c. Allotted/Diverted traffic for a deployment can be checked using following command.

```
az ml online-endpoint show -n $ENDPOINT_NAME --query traffic
```

2. Test the new deployment.
    a. Newly deployed endpoint can be tested by manually invoking it.

```
az ml online-endpoint invoke --name $ENDPOINT_NAME --deployment-name green --
request-file endpoints/online/model-2/sample-request.json
```

6) Divert the Traffic to New Model

1. Once manual invoking of the endpoint is successfully completed, we can transfer 10% of live traffic to the new deployment for testing and evaluation.
    a. This 10% traffic is sent to both endpoints. Response to the user is returned via original endpoint only, whereas the second endpoint is used for evaluation of the model.
    b. This is also known as Shadow deployment/Shadow testing.
    c. Such diverted traffic is known as mirror-ed traffic.

```
az ml online-endpoint update --name $ENDPOINT_NAME --mirror-traffic "green=10"
```

2. This traffic diversion to new deployment can be verified via logs.

```
az ml online-deployment get-logs --name blue --endpoint $ENDPOINT_NAME
```

Once testing is done command given in first step of this subsection can be used again and mirror traffic can be made 0% again to reduce the cost and system load. And full traffic can be shifted to the newly deployed endpoint.

3. Reduce the traffic to the Old deployment to 0% and fully divert traffic to new deployment.

```
az ml online-endpoint update --name $ENDPOINT_NAME --traffic "blue=0 green=100"
```

4. Remove the blue deployment.

```
az ml online-deployment delete --name blue --endpoint $ENDPOINT_NAME --yes --no-
wait
```

5. Delete the endpoint + deployment if the endpoint is not going to be used in future.

```
az ml online-endpoint delete --name $ENDPOINT_NAME --yes --no-wait
```

Canary Deployment Strategy for Kubernetes

This example uses one sample code provided by Microsoft.

Pre-requisites

- Azure account with Active subscription
- Azure Container Registry with Push privileges.
- Azure Kubernetes cluster.

1) Fork the Sample Code from Repository

Fork the repo by using the following code

```
https://github.com/MicrosoftDocs/azure-pipelines-canary-k8s
```

./app

1. app.p
    a. A simple, Flask-based web server that is instrumented by using the Prometheus instrumentation library for Python applications.
    b. A custom counter is set up for the number of good and bad responses given out, based on the value of the success_rate variable.
2. Dockerfile
    a. Used for building the image with each change made to *app.py*.
    b. With each change, the build pipeline is triggered, and the image gets built and pushed to the container registry.

./manifests

1. deployment.yml
    a. It contains the specification of the **sampleapp** deployment workload corresponding to the image published earlier.
    b. It can be used for stable version of deployment object along with deriving baselines and canary variants of the workloads.
2. service.yml
    a. Creates the sampleapp service. This service routes requests to the pods spun up by the deployments (stable, baseline, and canary) mentioned previously.

./misc

1. service-monitor.yml
    a. This file is used to setup a Service Monitor object. This object sets up Prometheus metric scraping.
2. fortio-deploy.yml

a. It is used to setup fortio deployment.
b. It is later used as a load-testing tool, to send the stream request to the **sampleapp** service deployed earlier.
c. The stream of requests sent to sampleapp are routed to pods under all three deployments (stable, baseline, and canary).

2) Install Prometheus-Operator

1. To install Prometheus on the cluster, use following command from the development machine.

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update # update local cache
helm install --name sampleapp  prometheus-community/kube-prometheus-stack
```

2. Kubectl and Helm should be pre-installed and the context to the cluster that is concerned for deployment should be set.
   - Helm will include the Grafanainstallation also.
3. Firstly, add the Prometheus Community Kubernetes Helm Charts repository to the Helm installation.
4. Then you'll install the kube-prometheus stack, a collection of Kubernetes manifests, Grafana dashboards, and Prometheus rules.

3) Create Service Connection

1. Go to **Project settings** > **Pipelines** > **Service connections** in the Azure DevOps menu.
2. Create a Docker registry service connection associated with your container registry. Name it **azure-pipelines-canary-k8s**.
3. Create a Kubernetes service connection for the Kubernetes cluster and namespace you want to deploy to. Name it **azure-pipelines-canary-k8s**.

4) Setup Continuous Integration

1. Go to **Pipelines** > **Create Pipeline** and select the repository.
2. On the **Configure** tab, choose **Starter pipeline**.
3. On the **Review** tab, replace the pipeline YAML with this code.

```
trigger:
- main

pool:
  vmImage: ubuntu-latest

variables:
  imageName: azure-pipelines-canary-k8s

steps:
- task: Docker@2
  displayName: Build and push image
  inputs:
    containerRegistry: azure-pipelines-canary-k8s #replace with name of Docker 
registry service connection
    repository: $(imageName)
    command: buildAndPush
    Dockerfile: app/Dockerfile
    tags: |
      $(Build.BuildId)
```

5) Edit Manifest File

- In manifest/deployment.yml, replace the <example> with the container registry's URL.
    - i.e., contosodemo.azurecr.io/azure-pipelines-canary-k8s

6) Setup Continuous Deployment

Deploy Canary Stage

Refer to the following steps for YAML deployment.

1. Go to **Pipelines** > **Environments** > **Create environment**.
2. Create a new environment.
    - **Name**: *akscanary*
    - **Resource**: Choose Kubernetes.
3. Select **Next**, and configure Kubernetes resource as follows:
    - **Provider**: Azure Kubernetes Service
    - **Azure subscription**: Choose the subscription that holds Kubernetes cluster.
    - **Cluster**: Choose your cluster.
    - **Namespace**: Create a new namespace, with the name *canarydemo*.
4. Select **Validate and Create**.

5. Go to **Pipelines**. Select the pipeline you created and select **Edit**.

6. Change the step you created previously to now use a stage.

    - Add two more steps to copy the manifests and *misc* directories as artifacts for use by consecutive stages. Following is sample YAML once completed.

```
trigger:
- main

pool:
  vmImage: ubuntu-latest

variables:
  imageName: azure-pipelines-canary-k8s
  dockerRegistryServiceConnection: dockerRegistryServiceConnectionName
#replace with name of your Docker registry service connection
  imageRepository: 'azure-pipelines-canary-k8s'
  containerRegistry: example.azurecr.io #replace with the name of your
container registry, Should be in the format example.azurecr.io
  tag: '$(Build.BuildId)'

stages:
- stage: Build
  displayName: Build stage
  jobs:
  - job: Build
    displayName: Build
    pool:
      vmImage: ubuntu-latest
    steps:
    - task: Docker@2
      displayName: Build and push image
      inputs:
        containerRegistry: $(dockerRegistryServiceConnection)
        repository: $(imageName)
        command: buildAndPush
        Dockerfile: app/Dockerfile
        tags: |
          $(tag)

    - publish: manifests
      artifact: manifests

    - publish: misc
      artifact: misc
```

7. Add a stage at the end of the file to deploy the Canary version.

```yaml
- stage: DeployCanary
  displayName: Deploy canary
  dependsOn: Build
  condition: succeeded()

  jobs:
  - deployment: Deploycanary
    displayName: Deploy canary
    pool:
      vmImage: ubuntu-latest
    environment: 'akscanary.canarydemo'
    strategy:
      runOnce:
        deploy:
          steps:
          - task: KubernetesManifest@0
            displayName: Create imagePullSecret
            inputs:
              action: createSecret
              secretName: azure-pipelines-canary-k8s
              dockerRegistryEndpoint: azure-pipelines-canary-k8s

          - task: KubernetesManifest@0
            displayName: Deploy to Kubernetes cluster
            inputs:
              action: 'deploy'
              strategy: 'canary'
              percentage: '25'
              manifests: |
                $(Pipeline.Workspace)/manifests/deployment.yml
                $(Pipeline.Workspace)/manifests/service.yml
              containers: '$(containerRegistry)/$(imageRepository):$(tag)'
              imagePullSecrets: azure-pipelines-canary-k8s

          - task: KubernetesManifest@0
            displayName: Deploy Forbio and ServiceMonitor
            inputs:
              action: 'deploy'
              manifests: |
                $(Pipeline.Workspace)/misc/*
```

8. Save the pipeline by directly committing to the Main branch.

7) Manual Intervention by Promoting or Rejecting Canary

Please refer to the following steps for YAML.

1. Go to **Pipelines** > **Environments** > **New environment**.
2. Configure the new environment.
   - **Name**: *akspromote*
   - **Resource**: Choose Kubernetes.
3. Select **Next**, and configure your Kubernetes resource as follows:
   - **Provider**: Azure Kubernetes Service
   - **Azure subscription**: Choose the subscription that holds your Kubernetes cluster.
   - **Cluster**: Choose your cluster.
   - **Namespace**: Choose the namespace, *canarydemo*, that you created earlier.
4. Select **Validate and Create**.
5. Select your new akspromote environment from the list of environments.
6. Select **Approvals and checks** > **Approvals**. Then select the ellipsis icon (the three dots).
7. Configure your approval as follows:
   - **Approvers**: Add your own user account.
   - **Advanced**: Make sure that the **Allow approvers to approve their own runs** box is selected.
8. Select **Create**.
9. Go to **Pipelines** and select the pipeline that you created. Then select **Edit**.
10. Add another stage, PromoteRejectCanary, at the end of your YAML file, to promote the changes.

```
- stage: PromoteRejectCanary
  displayName: Promote or Reject canary
  dependsOn: DeployCanary
  condition: succeeded()

  jobs:
  - deployment: PromoteCanary
    displayName: Promote Canary
```

```yaml
    pool:
      vmImage: ubuntu-latest
    environment: 'akspromote.canarydemo'
    strategy:
      runOnce:
        deploy:
          steps:
          - task: KubernetesManifest@0
            displayName: promote canary
            inputs:
              action: 'promote'
              strategy: 'canary'
              manifests: '$(Pipeline.Workspace)/manifests/*'
              containers: '$(containerRegistry)/$(imageRepository):$(tag)'
              imagePullSecrets: '$(imagePullSecret)'
```

11. Add another stage, RejectCanary, at the end of your YAML file, to roll back the changes.

```yaml
- stage: RejectCanary
  displayName: Reject canary
  dependsOn: PromoteRejectCanary
  condition: failed()

  jobs:
  - deployment: RejectCanary
    displayName: Reject Canary
    pool:
      vmImage: ubuntu-latest
    environment: 'akscanary.canarydemo'
    strategy:
      runOnce:
        deploy:
          steps:
          - task: KubernetesManifest@0
            displayName: reject canary
            inputs:
              action: 'reject'
              strategy: 'canary'
              manifests: '$(Pipeline.Workspace)/manifests/*'
```

12. Save this YAML pipeline by selecting **Save**, and then commit it directly to the main branch.

8) Deploy a Stable Version

For the first run of the pipeline the stable version of the workloads, and their baseline or canary versions don't exist in the cluster. To deploy the stable version:

1. In *app/app.py*, change success_rate = 5 to success_rate = 10. This change triggers the pipeline, leading to a build and push of the image to the container registry. It will also trigger the DeployCanary stage.
2. Because we configured an approval on the akspromote environment, the release will wait before running that stage.
3. In the summary of the run, select **Review** > **Approve**. This deploys the stable version of the workloads (the sampleapp deployment in *manifests/deployment.yml*) to the namespace.

9) Initiate Canary Workflow

The stable version of the workload sampleapp now exists in the cluster. Next, make the following change to the simulation application:

In *app/app.py*, change success_rate = 10 to success_rate = 20.

This change triggers the build pipeline, resulting in the build and push of the image to the container registry. This process in turn triggers the release pipeline and begins the **Deploy Canary** stage.

10) Simulate the Request

On your development machine, run the following commands, and keep it running to send a constant stream of requests at the sampleapp service.

- sampleapp routes the requests to the pods spun by the stable sampleapp deployment, and to the pods spun up by the sampleapp-baseline and sampleapp-canary deployments.
- The selector specified for sampleapp is applicable to all of these pods.

```
FORTIO_POD=$(kubectl get pod | grep fortio | awk '{ print $1 }')
kubectl exec -it $FORTIO_POD -c fortio /usr/bin/fortio -- load -allow-initial-
errors -t 0 http://sampleapp:8080/
```

11) Setup Grafana Dashboard

1. Run the following port-forwarding command on the local development machine to be able to access Grafana.

```
kubectl port-forward svc/sampleapp-grafana 3000:80
```

2. In a browser, open the following URL.

```
http://localhost:3000/login
```

3. When you're prompted for credentials, unless the adminPassword value was overridden during the prometheus-operator Helm chart installation, you can use the following values:
    a. username: admin
    b. password: prom-operator
4. From the menu on the left, choose + > Dashboard > Graph
5. Select anywhere on the newly added panel, and type e to edit the panel.
6. On the Metrics tab, enter the following query:

```
rate(requests_total{pod=~"sampleapp-.*", custom_status="good"}[1m])
```

7. On the General tab, change the name of this panel to All sampleapp pods.
8. In the overview bar at the top of the page, change the duration range to Last 5 minutes or Last 15 minutes.
9. To save this panel, select the save icon in the overview bar.
10. The preceding panel visualizes success rate metrics from all the variants. These include stable (from the sampleapp deployment), baseline (from the sampleapp-baseline deployment), and canary (from the sampleapp-canary deployment). You can visualize just the baseline and canary metrics by adding another panel, with the following configuration:
    a. On the General tab, for Title, select sampleapp baseline and canary.
    b. On the Metrics tab, use the following query:

```
rate(requests_total{pod=~"sampleapp-baseline-.*|sampleapp-canary-.*",
custom_status="good"}[1m])
```

Deploy Simple ML Model Using Docker Containers in Azure

Pre-requisites

- Azure Machine Learning Workspace
- Azure CLI and the m1 extension/Azure ML Python SDK V2
- Contributor access to personal/ Service Principal for the resource group.
- Docker Engine preinstalled in the local system.

1) Download The Source Code

1. Download the source code from the link below.
   a. Open Azure CLI and run the following command.

```
git clone https://github.com/Azure/azureml-examples --depth 1
cd azureml-examples/cli
```

2. Initialize Environment Variables
   a. Define the environment variables.

```
BASE_PATH=endpoints/online/custom-container/tfserving/half-plus-two
AML_MODEL_NAME=tfserving-mounted
MODEL_NAME=half_plus_two
MODEL_BASE_PATH=/var/azureml-app/azureml-models/$AML_MODEL_NAME/1
```

3. Download a Tensorflow model.
   a. Download and unzip a model that divides an input by 2 and adds 2 to the result.

```
wget https://aka.ms/half_plus_two-model -O $BASE_PATH/half_plus_two.tar.gz
tar -xvf $BASE_PATH/half_plus_two.tar.gz -C $BASE_PATH
```

4. Run a Tensorflow Serving image locally to test working of it.

```
docker run --rm -d -v $PWD/$BASE_PATH:$MODEL_BASE_PATH -p 8501:8501 -e
MODEL_BASE_PATH=$MODEL_BASE_PATH -e MODEL_NAME=$MODEL_NAME --name="tfserving-
test" docker.io/tensorflow/serving:latest
sleep 10
```

2) Check if Liveness and Scoring Requests

1. Check that the container is "alive," meaning that the process inside the container is still running. You should get a 200 (OK) response.

```
curl -v http://localhost:8501/v1/models/$MODEL_NAME
```

2. Check that you can get predictions about unlabeled data.

```
curl --header "Content-Type: application/json" --request POST --data
@$BASE_PATH/sample_request.json
http://localhost:8501/v1/models/$MODEL_NAME:predict
```

Stop the Image

```
docker stop tfserving-test
```

3) Deploy Online End Point to Azure

Create one YAML file for the Endpoint and Deployment

1. Create a file named **tfserving-endpoint.yml** using following code.

```
$schema:
https://azuremlsdk2.blob.core.windows.net/latest/managedOnlineEndpoint.schema.jso
n
name: tfserving-endpoint
auth_mode: aml_token
```

2. Create a file **tfserving-deployment.yml** using following code template.

```
$schema:
https://azuremlschemas.azureedge.net/latest/managedOnlineDeployment.schema.json
name: tfserving-deployment
endpoint_name: tfserving-endpoint
model:
  name: tfserving-mounted
  version: {{MODEL_VERSION}}
  path: ./half_plus_two
environment_variables:
  MODEL_BASE_PATH: /var/azureml-app/azureml-models/tfserving-
mounted/{{MODEL_VERSION}}
  MODEL_NAME: half_plus_two
environment:
  #name: tfserving
  #version: 1
  image: docker.io/tensorflow/serving:latest
  inference_config:
    liveness_route:
      port: 8501
      path: /v1/models/half_plus_two
    readiness_route:
      port: 8501
      path: /v1/models/half_plus_two
    scoring_route:
      port: 8501
      path: /v1/models/half_plus_two:predict
instance_type: Standard_DS3_v2
instance_count: 1
```

4) Create an End Point and Deployment

1. Run following commands in Azure CLI to create deployment with both the YAML files.

```
az ml online-endpoint create --name tfserving-endpoint -f
endpoints/online/custom-container/tfserving-endpoint.yml
```

```
az ml online-deployment create --name tfserving-deployment -f
endpoints/online/custom-container/tfserving-deployment.yml --all-traffic
```

5) Invoke the Endpoint

Run following command to invoke the Endpoint.

```
RESPONSE=$(az ml online-endpoint invoke -n $ENDPOINT_NAME --request-file
$BASE_PATH/sample_request.json)
```

6) Delete the Endpoint

```
az ml online-endpoint delete --name tfserving-endpoint
```

APPENDIX - B

Difference between A/B Testing and Blue-Green Deployment Strategy

In the A/B testing, users are split into two different groups and exposed to one of the model versions between A & B. This strategy measures majorly the user experience with two different models having different features consideration. Based on the user feedback, one of the model is chosen as the final version.

In Blue-Green deployment strategy, Green/Updated version of the model is first kept for testing. Once proper testing is done by the developer, only then it is exposed to the users. Here, full traffic of user is switched to Green deployment at once.

Reference: Link

Difference between Shadow Deployment Strategy and Blue-Green Deployment

In Shadow deployment, user requests replicated and sent to both, current version and a shadow version of model. It is majorly used for extensive testing on user inputs.

In Blue-Green deployment strategy, testing is done from developer side, either manual or in automated manner. This strategy is majorly used to reduce the downtime while deploying the model.

Difference between Blue-Green Deployment and Canary Deployment

In Blue-Green deployment, full environment is replicated and provisioned where new model version is deployed, whereas in canary deployment one of the nodes of the environment is switched with new version.

Once testing for Green model deployment is done, traffic is fully switched to the new deployment version at once in Blue-Green deployment. In Canary deployment, some percentage of the traffic is initially diverted to the new model version and the percentage is slowly increased over time and testing.

Blue-Green deployment provides zero-downtime whereas Canary allows easy toggle between new and old features of the models.