



Azure Machine Learning service: *A Technical Overview*

Florian Pydde 06/2020
Delivery Data Scientist
Florian.Pyddde@microsoft.com

Agenda

Part 1: Introduction to Azure AI

Part 2: Pre-built AI

Part 3: Custom AI

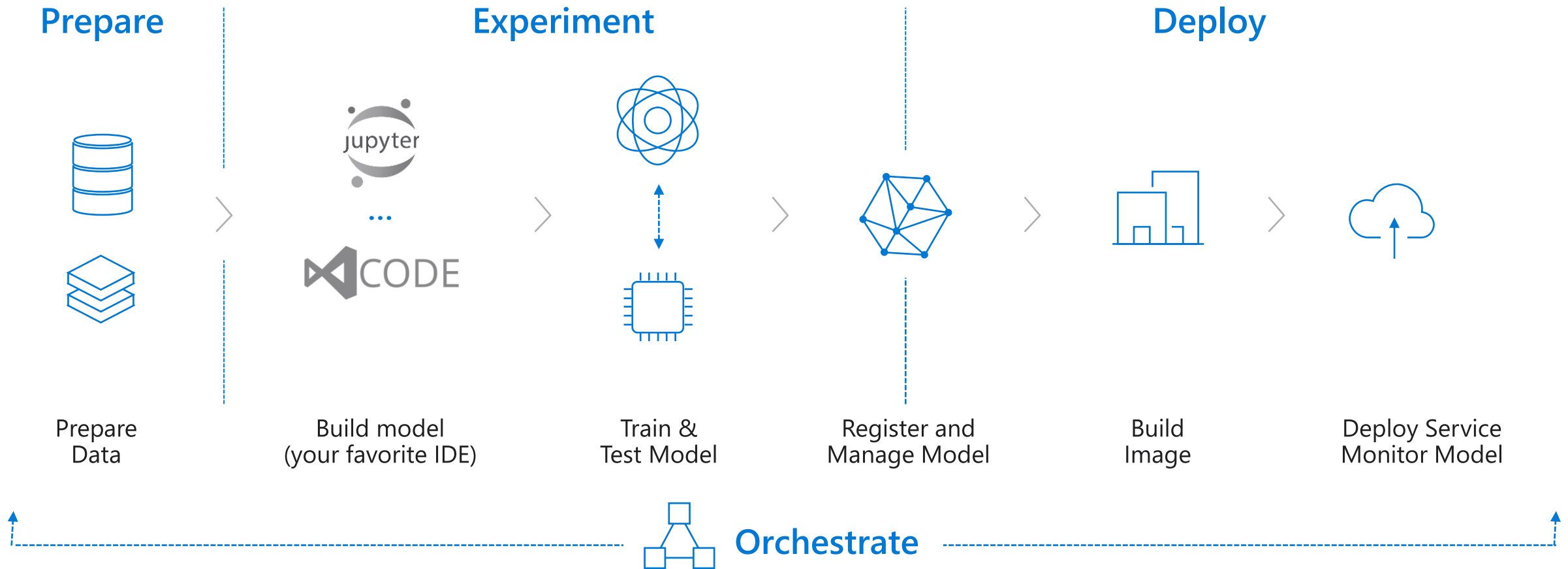
- Machine Learning Requirements
- Azure ML Platform
- Azure Machine Learning service
- E2E coding example
- Analytics Data Architecture
- Automated Machine Learning & Parameter tuning
- Live Demos: Pytorch training remotely & deploy



Requirements of an advanced ML Platform

Machine Learning

Typical E2E Process



Data Preparation

Requirements

Multiple Data Sources

SQL and NoSQL databases, file systems, network attached storage and cloud stores (such as Azure Blob Storage) and HDFS.

Multiple Formats

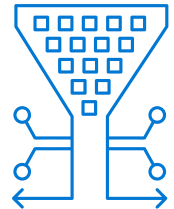
Binary, text, CSV, TS, ARFF, etc.

Cleansing

Detect and fix NULL values, outliers, out-of-range values, duplicate rows.

Transformation

General data transformation (transforming types) and ML-specific transformations (indexing, encoding, assembling into vectors, normalizing the vectors, binning, normalization and categorization).



Model Building

Requirements

Choice of algorithms

Choice of language

Python, R

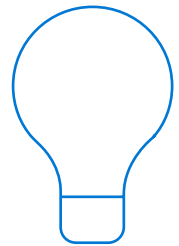
Choice of development tools

Browser-based, REPL-oriented, notebooks such as Jupyter, PyCharm and Spark Notebooks.

Desktop IDEs such as Visual Studio and R-Studio for R development.

Local Testing

To verify correctness before submitting to a more powerful (and expensive) training infrastructure.



Model Training

Requirements

Powerful Compute Environment

Choice should include scale-up VMs, auto-scaling scale-out clusters

Preconfigured

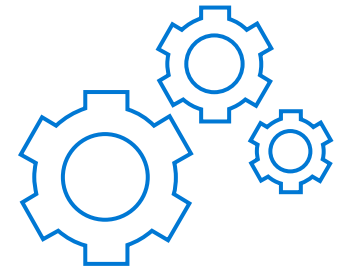
The compute environment should be pre-setup with all the correct versions ML frameworks, libraries, executables and container images.

Job Management

Data scientists should be able to easily start, stop, monitor and manage Jobs.

Automated Model and Parameter Selection

Solution should automatically select the best algorithms, and the corresponding best hyperparameters, for the desired outcome.



Model Registration and Management

Requirements

Containerization

Automatically convert models to Docker containers so that they can be deployed into an execution environment.

Versioning

Assign versions numbers to models, to track changes over time, to identify and retrieve a specific version for deployment, for A/B testing, rolling back changes etc.

Model Repository

For storing and sharing models, to enable integration into CI/CD pipelines.

Track Experiments

For auditing, see changes over time and enable collaboration between team members.



Model Deployment

Requirements

Choice of Deployment Environments

Single VM, Cluster of VMs, Spark Clusters, Hadoop Clusters, In the cloud, On-premises

Edge Deployment

To enable predictions close to the event source-for quicker response and avoid unnecessary data transfer.

Security

Even when deployed at the edge, the e2e security must be maintained. Models should be deployed, and data transmitted only to secure and authenticated devices.

Monitoring

Monitor the status, performance and security.





**Azure offers a comprehensive
AI/ML platform that meets—and
exceeds—requirements**

Machine Learning on Azure

Domain specific pretrained models

To reduce time to market



Vision



Speech



Language



Search

Familiar Data Science tools

To simplify model development



PyCharm



Jupyter



Visual Studio Code



Command line

Popular frameworks

To build advanced deep learning solutions



Pytorch



TensorFlow



Scikit-Learn



Onnx

Productive services

To empower data science and development teams



Azure
Databricks



Azure Machine
Learning



Machine
Learning VMs

Powerful infrastructure

To accelerate deep learning



CPU



GPU



FPGA



From the Intelligent Cloud to the Intelligent Edge



(Custom) Model Creation with Azure ML Platform

Development Tools



Languages



Frameworks



ONNX



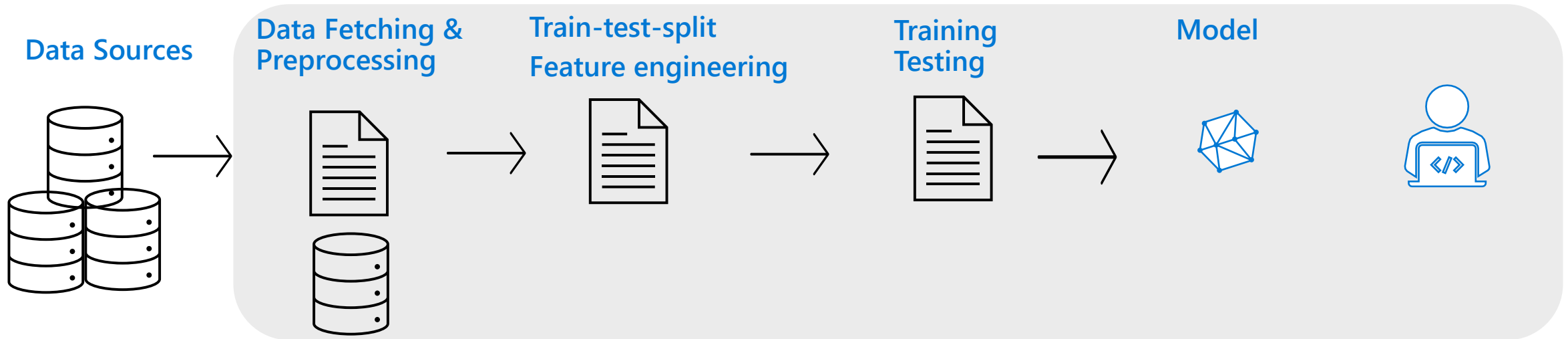
dmlc
XGBoost



Azure ML service

Lets you easily implement this AI/ML Lifecycle

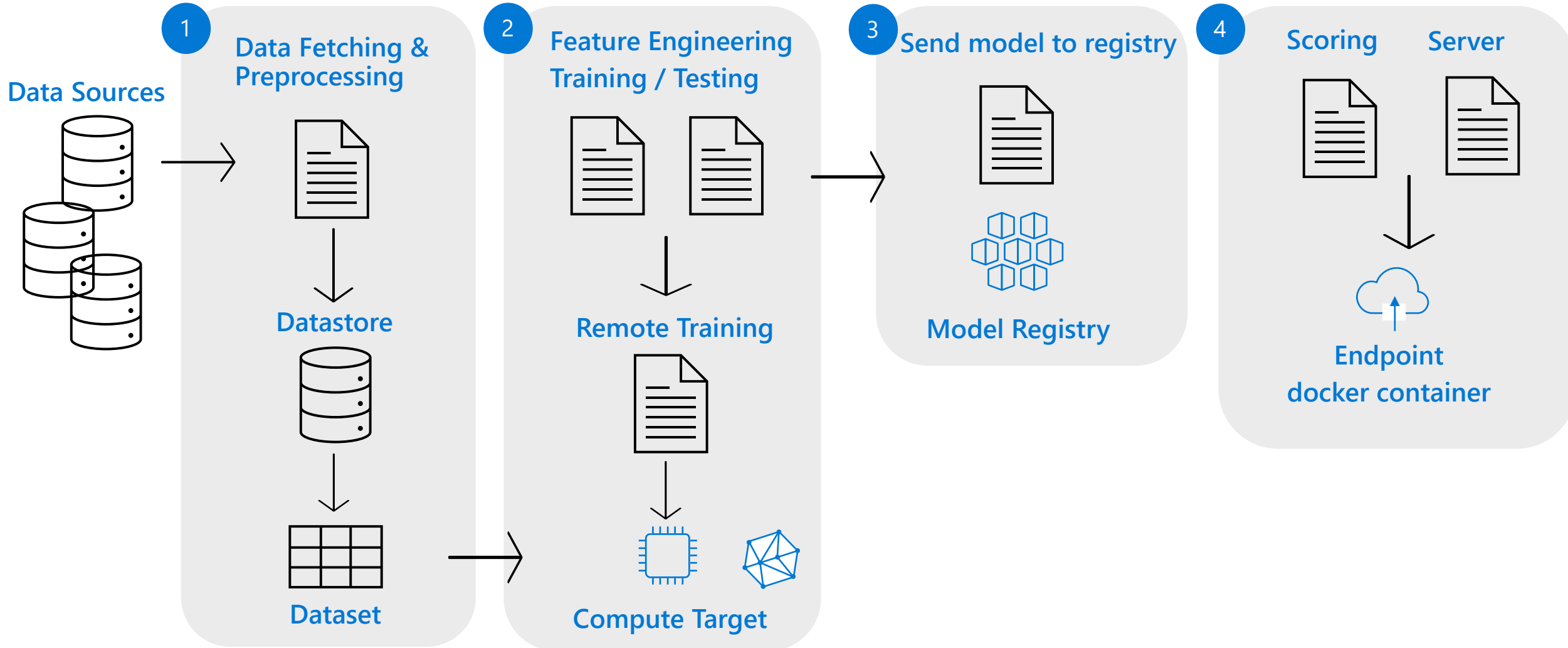
Experimentation Environment

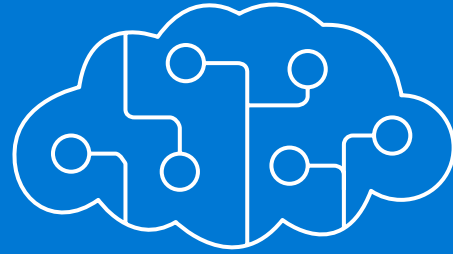


Azure ML service

Lets you easily implement this AI/ML Lifecycle

Operationalizing

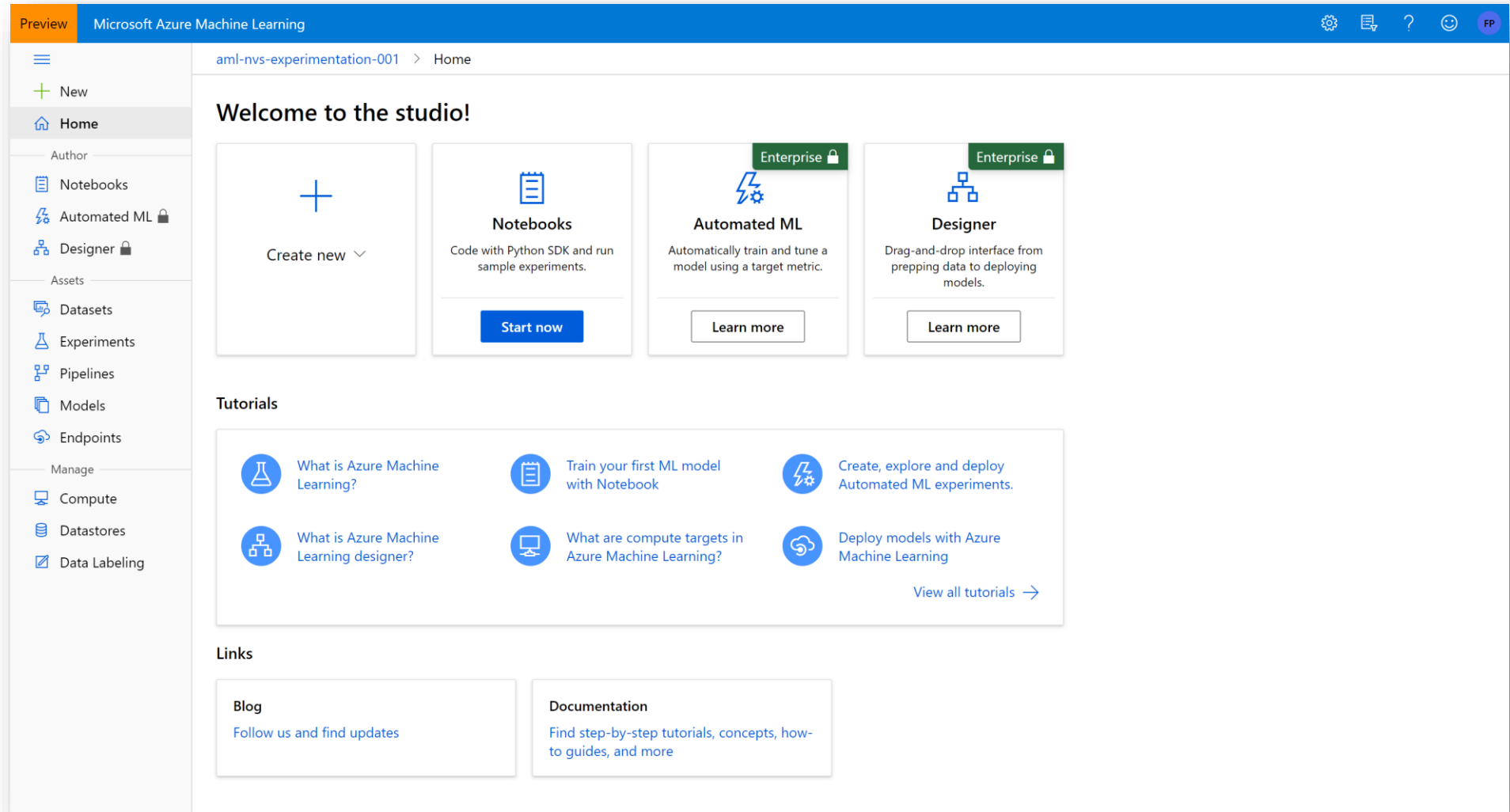




Azure Machine Learning: Technical Details

Azure ML service

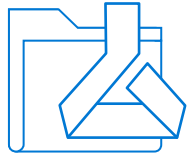
Key Artifacts



<https://ml.azure.com/>

Azure ML service

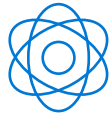
Key Artifacts



Workspace



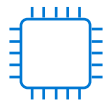
Models



Experiments



Pipelines



Compute Target



Images



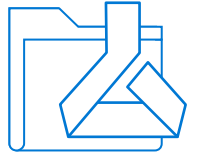
Deployment



Data Stores

Azure ML service Artifact

Workspace



The workspace is the **top-level resource** for the Azure Machine Learning service. It provides a centralized place to work with all the artifacts you create when using Azure Machine Learning service.

The workspace keeps a list of compute targets that can be used to train your model. It also keeps a history of the training runs, including logs, metrics, output, and a snapshot of your scripts.

Models are registered with the workspace.

You can create multiple workspaces, and each workspace can be shared by multiple people.

When you create a new workspace, it automatically creates these Azure resources:

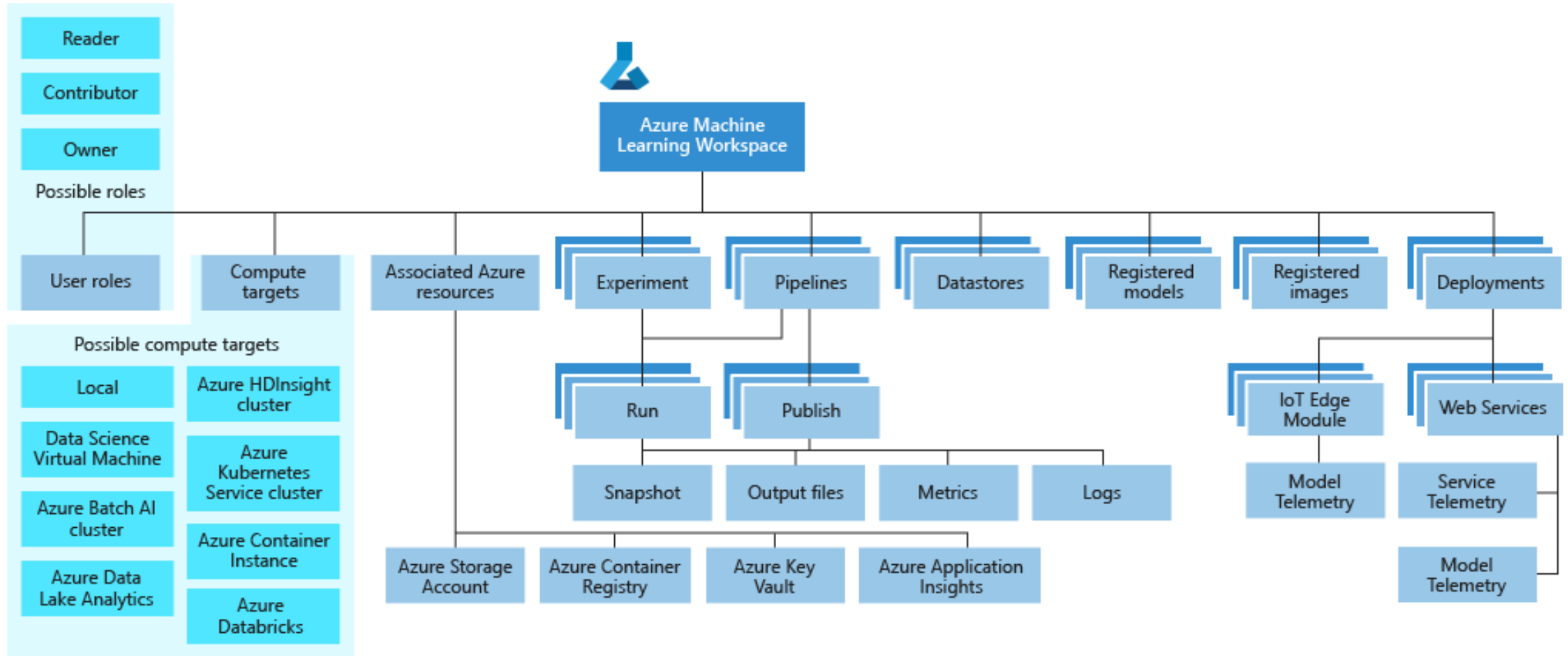
[Azure Container Registry](#) - Registers docker containers that are used during training and when deploying a model.

[Azure Storage](#) - Used as the default datastore for the workspace.

[Azure Application Insights](#) - Stores monitoring information about your models.

[Azure Key Vault](#) - Stores secrets used by compute targets and other sensitive information needed by the workspace.

Azure ML service Workspace Taxonomy



Azure ML service Artifacts

Models and Model Registry



Model

A machine learning model is an artifact that is created by your training process. You use a model to get predictions on new data.

A model is produced by a **run** in Azure Machine Learning.

Note: You can also use a model trained outside of Azure Machine Learning.

Azure Machine Learning service is framework agnostic — you can use any popular machine learning framework when creating a model.

A model can be registered under an Azure Machine Learning service workspace



Model Registry

Keeps track of all the models in your Azure Machine Learning service workspace.

Models are identified by name and version.

You can provide additional metadata tags when you register the model, and then use these tags when searching for models.

You cannot delete models that are being used by an image.

Azure ML Artifact

Compute Target

Compute Targets are the compute resources used to run training scripts or host your model when deployed as a web service.

They can be created and managed using the Azure Machine Learning SDK or CLI.

You can attach to existing resources.

You can start with local runs on your machine, and then scale up and out to other environments.

Currently supported compute targets

Compute Target	Training	Deployment
Local Computer	✓	
A Linux VM in Azure (such as the Data Science Virtual Machine)	✓	
Azure ML Compute	✓	
Azure Databricks	✓	
Azure Data Lake Analytics	✓	
Apache Spark for HDInsight	✓	
Azure Container Instance		✓
Azure Kubernetes Service		✓
Azure IoT Edge		✓
Field-programmable gate array (FPGA)		✓

Azure ML

Currently Supported Compute Targets

Compute target	GPU acceleration	Hyperdrive	Automated model selection	Can be used in pipelines
Local computer	Maybe		✓	
Data Science Virtual Machine (DSVM)	✓	✓	✓	✓
Azure ML compute	✓	✓	✓	✓
Azure Databricks	✓		✓	✓
Azure Data Lake Analytics				✓
Azure HDInsight				✓

Azure ML service Artifacts

Image and Registry



Image contains

1. A model.
2. A scoring script used to pass input to the model and return the output of the model.
3. Dependencies needed by the model or scoring script/application.

Two types of images

1. **FPGA image:** Used when deploying to a field-programmable gate array in the Azure cloud.
2. **Docker image:** Used when deploying to compute targets such as Azure Container Instances and Azure Kubernetes Service.

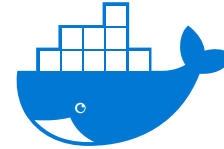


Image Registry

Keeps track of images created from models.

Metadata tags can be attached to images. Metadata tags are stored by the image registry and can be used in image searches

Azure ML Concept

Model Management

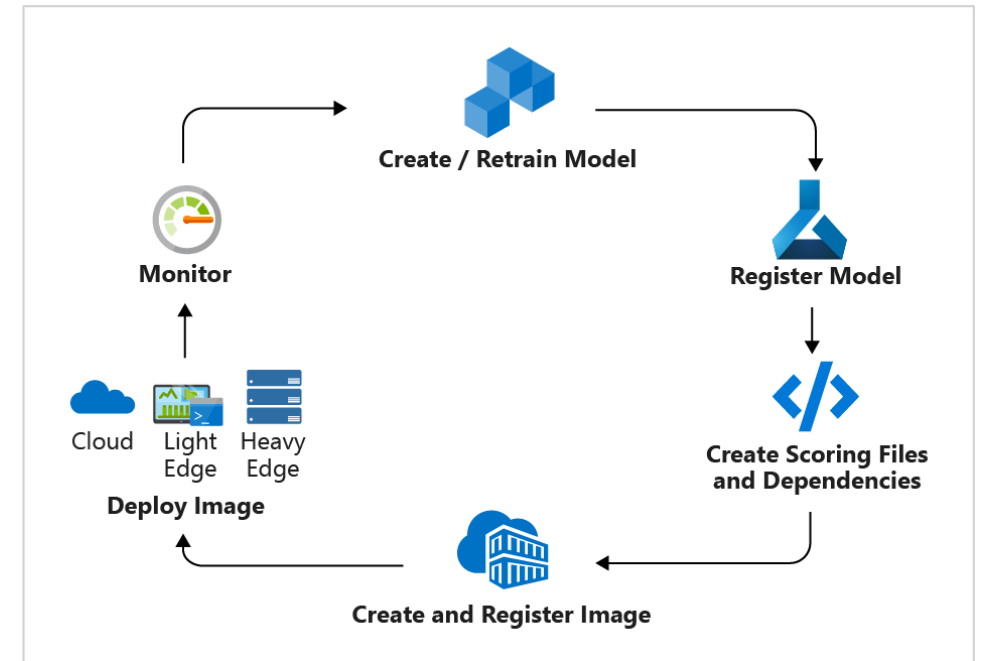
Model Management in Azure ML usually involves these four steps

Step 1: Register Model using the Model Registry

Step 2: Register Image using the Image Registry (the Azure Container Registry)

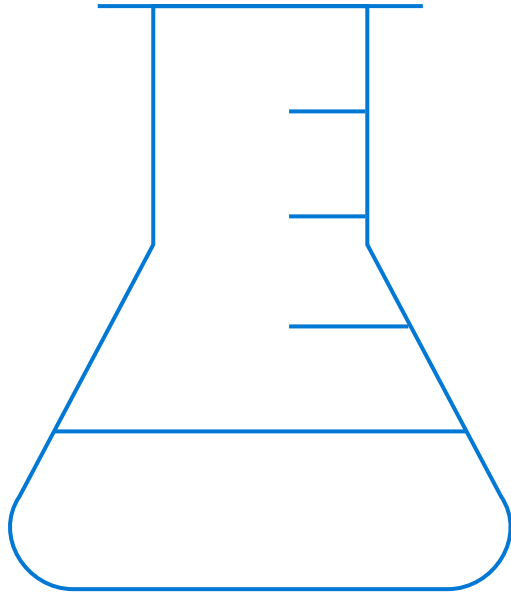
Step 3: Deploy the Image to cloud or to edge devices

Step 4: Monitor models—you can monitor input, output, and other relevant data from your model.



Azure ML Artifacts

Runs and Experiments



Experiment

Grouping of many runs from a given script.

Always belongs to a workspace.

Stores information about runs

Run

Produced when you submit a script to train a model. Contains:

Metadata about the run (timestamp, duration etc.)

Metrics logged by your script.

Output files autocollected by the experiment, or explicitly uploaded by you.

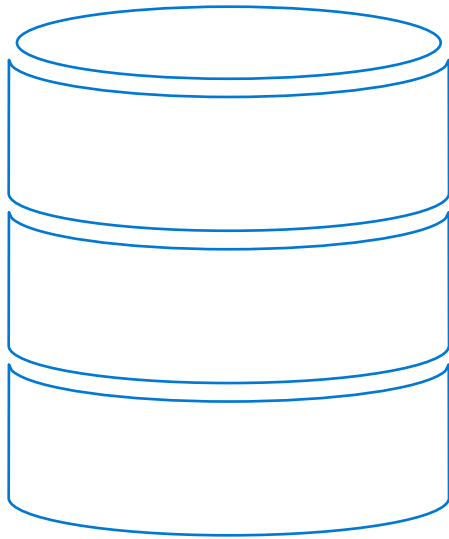
A snapshot of the directory that contains your scripts, prior to the run.

Run configuration

A set of instructions that defines how a script should be run in a given compute target.

Azure ML Artifact

Datastore



A datastore is a storage abstraction over an Azure Storage Account.

The datastore can use either an Azure blob container or an Azure file share as the backend storage.

Each workspace has a default datastore, and you may register additional datastores.

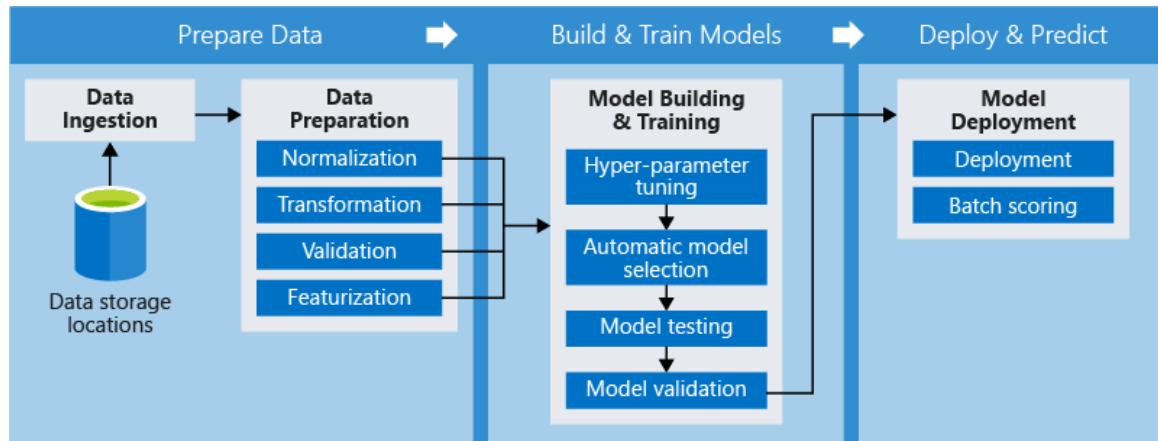
Use the Python SDK API or Azure Machine Learning CLI to store and retrieve files from the datastore.

Azure ML Artifact Pipeline

An Azure ML pipeline consists of several steps, where each step can be performed independently or as part of a single deployment command.

A [step](#) is a computational unit in the pipeline.

Diagram shows an example pipeline with multiple steps.



Azure ML pipelines enables data scientists, data engineers, and IT professionals to collaborate on the steps involved in: Data preparation, Model training, Model evaluation, Deployment

Azure ML Pipelines

Advantages

Advantage	Description
Unattended runs	Schedule a few steps to run in parallel or in sequence in a reliable and unattended manner. Since data prep and modeling can last days or weeks, you can now focus on other tasks while your pipeline is running.
Mixed and diverse compute	Use multiple pipelines that are reliably coordinated across heterogeneous and scalable computes and storages. Individual pipeline steps can be run on different compute targets, such as HDInsight, GPU Data Science VMs, and Databricks.
Reusability	Pipelines can be templated for specific scenarios such as retraining and batch scoring. They can be triggered from external systems via simple REST calls.
Tracking and versioning	Instead of manually tracking data and result paths as you iterate, use the pipelines SDK to explicitly name and version your data sources, inputs, and outputs as well as manage scripts and data separately for increased productivity

Azure ML Artifact

Deployment

Deployment is an instantiation of an image. Two options:



A dashed blue line originates from the text 'Deployment is an instantiation of an image. Two options:'. It extends horizontally to the right and then splits into two vertical dashed lines, each ending in a downward-pointing arrowhead. The left arrow points to the 'Web service' header, and the right arrow points to the 'IoT Module' header.

Web service

A deployed web service can run on Azure Container Instances, Azure Kubernetes Service, or field-programmable gate arrays (FPGA).

Can receive scoring requests via an exposed a load-balanced, HTTP endpoint.

Can be monitored by collecting Application Insight telemetry and/or model telemetry.

Azure can automatically scale deployments.

IoT Module

A deployed IoT Module is a Docker container that includes the model, associated script and additional dependencies.

Is deployed using **Azure IoT Edge** on edge devices.

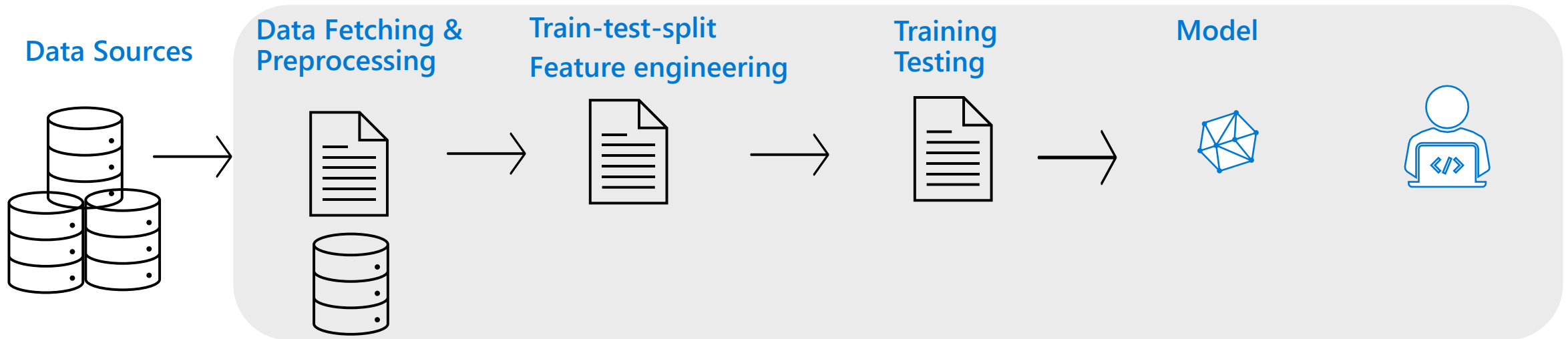
Can be monitored by collecting Application Insight telemetry and/or model telemetry.

Azure IoT Edge will ensure that your module is running and monitor the device that is hosting it.

Azure ML service

Lets you easily implement this AI/ML Lifecycle

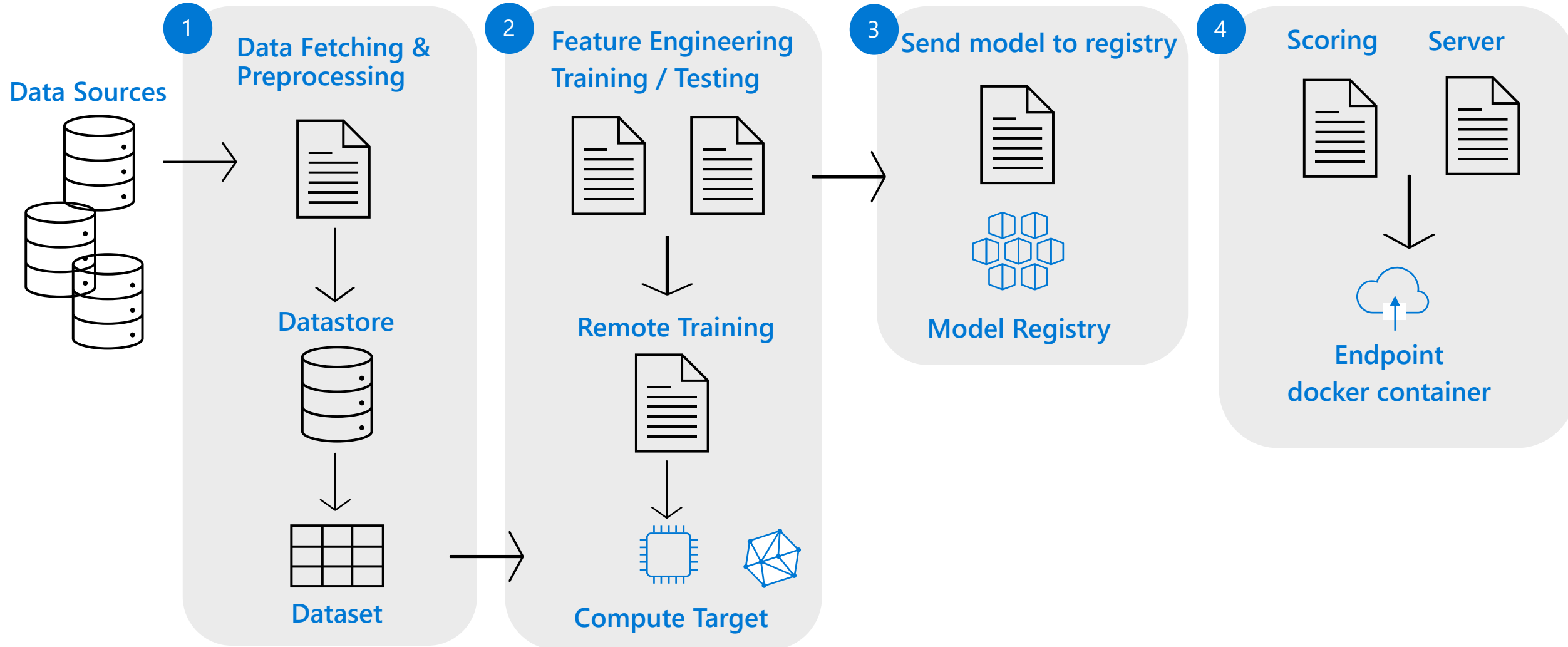
Experimentation Environment



Azure ML service

Lets you easily implement this AI/ML Lifecycle

Operationalizing





How to use the Azure Machine Learning service: E2E coding example using the SDK

Workspace Interaction

The artifacts creation and interaction with the workspace can be done in 3 different ways



User Interface
Azure Portal



Code
Python SDK



Command Line Interface (CLI)
Powershell

Setup for Code Example

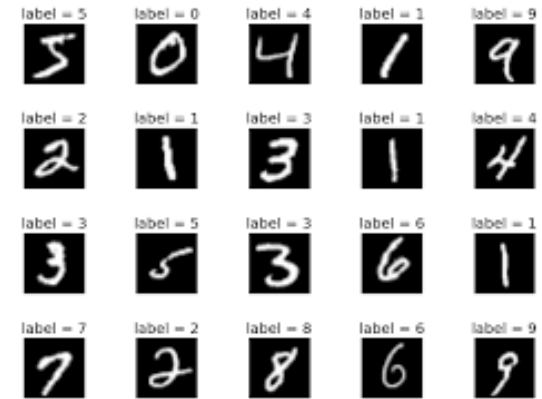
This tutorial trains a simple logistic regression using the [MNIST dataset](#) and [scikit-learn](#) with [Azure Machine Learning service](#).

MNIST is a dataset consisting of 70,000 grayscale images.

Each image is a handwritten digit of 28x28 pixels, representing a number from 0 to 9.

The goal is to create a multi-class classifier to identify the digit a given image represents.

[Github Link to Notebook](#)



Step 1 – Create a workspace

```
from azureml.core import Workspace
ws = Workspace.create(name='myworkspace',
                     subscription_id='<azure-subscription-id>',
                     resource_group='myresourcegroup',
                     create_resource_group=True,
                     location='eastus2' # or other supported Azure region
)

# see workspace details
ws.get_details()
```

Step 2 – Create an Experiment

Create an experiment to track the runs in the workspace. A workspace can have multiple experiments

```
experiment_name = 'my-experiment-1'

from azureml.core import Experiment
exp = Experiment(workspace=ws, name=experiment_name)
```

Step 3 – Create remote compute target

```
# choose a name for your cluster, specify min and max nodes
compute_name = os.environ.get("BATCHAI_CLUSTER_NAME", "cpucluster")
compute_min_nodes = os.environ.get("BATCHAI_CLUSTER_MIN_NODES", 0)
compute_max_nodes = os.environ.get("BATCHAI_CLUSTER_MAX_NODES", 4)

# This example uses CPU VM. For using GPU VM, set SKU to STANDARD_NC6
vm_size = os.environ.get("BATCHAI_CLUSTER_SKU", "STANDARD_D2_V2")

provisioning_config = AmlCompute.provisioning_configuration(
    vm_size = vm_size,
    min_nodes = compute_min_nodes,
    max_nodes = compute_max_nodes)

# create the cluster
print(' creating a new compute target... ')
compute_target = ComputeTarget.create(ws, compute_name, provisioning_config)

# You can poll for a minimum number of nodes and for a specific timeout.
# if no min node count is provided it will use the scale settings for the cluster
compute_target.wait_for_completion(show_output=True,
                                   min_node_count=None, timeout_in_minutes=20)
```

Zero is the default.
If min is zero then
the cluster is
automatically
deleted when no
jobs are running
on it.

Step 4 – Upload data to the cloud

First load the compressed files into numpy arrays. Note the '`load_data`' is a custom function that simply parses the compressed files into numpy arrays.

```
# note that while loading, we are shrinking the intensity values (X) from 0-255 to 0-1 so that the
model converge faster.
X_train = load_data('./data/train-images.gz', False) / 255.0
y_train = load_data('./data/train-labels.gz', True).reshape(-1)

X_test = load_data('./data/test-images.gz', False) / 255.0
y_test = load_data('./data/test-labels.gz', True).reshape(-1)
```

Now make the data accessible remotely by uploading that data from your local machine into Azure so it can be accessed for remote training. The files are uploaded into a directory named `mnist` at the root of the datastore.

```
ds = ws.get_default_datastore()
print(ds.datastore_type, ds.account_name, ds.container_name)

ds.upload(src_dir='./data', target_path='mnist', overwrite=True, show_progress=True)
```

We now have everything you need to start training a model.

Step 5 – Train a local model

Train a simple logistic regression model using scikit-learn locally. This should take a minute or two.

```
%%time from sklearn.linear_model import LogisticRegression
clf = LogisticRegression()
clf.fit(X_train, y_train)

# Next, make predictions using the test set and calculate the accuracy
y_hat = clf.predict(X_test)
print(np.average(y_hat == y_test))
```

You should see the local model accuracy displayed. [It should be a number like 0.915]

Step 6 – Train model on remote cluster

To submit a training job to a remote you have to perform the following tasks:

- 6.1: Create a directory
- 6.2: Create a training script
- 6.3: Create an estimator object
- 6.4: Submit the job

Step 6.1 – Create a directory

Create a directory to deliver the required code from your computer to the remote resource.

```
import os
script_folder = './sklearn-mnist'
os.makedirs(script_folder, exist_ok=True)
```

Step 6.2 – Create a Training Script (1/2)

```
%%writefile $script_folder/train.py

# parse script arguments

parser = argparse.ArgumentParser()

parser.add_argument('--data-folder', type=str, dest='data_folder', help='data folder mounting point')

args = parser.parse_args()

data_folder = args.data_folder

X_train = load_data(os.path.join(data_folder, 'train-images.gz'), False) / 255.0
X_test  = load_data(os.path.join(data_folder, 'test-images.gz'), False) / 255.0
y_train = load_data(os.path.join(data_folder, 'train-labels.gz'), True).reshape(-1)
y_test  = load_data(os.path.join(data_folder, 'test-labels.gz'), True).reshape(-1)

# get hold of the current run
run = Run.get_context()

#Train a logistic regression model

clf = LogisticRegression(C=1.25, random_state=42)
clf.fit(X_train, y_train)
```


Step 6.2 – Create a Training Script (2/2)

```
print('Predict the test set')
y_hat = clf.predict(X_test)

# calculate accuracy on the prediction
acc = np.average(y_hat == y_test)
print('Accuracy is', acc)

run.log('regularization rate', np.float(1.25))
run.log('accuracy', np.float(acc)) os.makedirs('outputs', exist_ok=True)

# The training script saves the model into a directory named 'outputs'. Note files saved in the
# outputs folder are automatically uploaded into experiment record. Anything written in this
# directory is automatically uploaded into the workspace.
joblib.dump(value=clf, filename='outputs/sklearn_mnist_model.pkl')
```

Step 6.3 – Create an Estimator (Deployment Script)

An estimator object is used to submit the run.

```
from azureml.train.estimator import Estimator

script_params = { '--data-folder': ds.as_mount() }

est = Estimator(source_directory=script_folder,
                script_params=script_params,
                compute_target=compute_target,
                entry_script='train.py',
                conda_packages=['scikit-learn'])
```

Name of
estimator

Python Packages
needed for training

Training Script
Name

Compute
target (Batch AI
in this case)

Parameters required
from the training script

The directory that contains the
scripts. All the files in this
directory are uploaded into
the cluster nodes for execution

Step 6.4 – Submit the job to the cluster for training

```
run = exp.submit(config=est)
```

What happens after you submit the job?

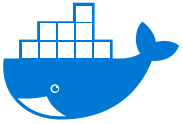


Image creation

A Docker image is created matching the Python environment specified by the estimator. The image is uploaded to the workspace. Image creation and uploading takes about 5 minutes.

This happens once for each Python environment since the container is cached for subsequent runs. During image creation, logs are streamed to the run history. You can monitor the image creation progress using these logs.



Scaling

If the remote cluster requires more nodes to execute the run than currently available, additional nodes are added automatically. Scaling typically takes about 5 minutes.



Running

In this stage, the necessary scripts and files are sent to the compute target, then data stores are mounted/copied, then the entry_script is run. While the job is running, stdout and the ./logs directory are streamed to the run history. You can monitor the run's progress using these logs.



Post-Processing

The ./outputs directory of the run is copied over to the run history in your workspace so you can access these results.

Step 7 – Monitor a run

You can watch the progress of the run with a Jupyter widget. The widget is asynchronous and provides live updates every 10-15 seconds until the job completes.

```
from azureml.widgets import RunDetails
RunDetails(run).show()
```

Here is a still snapshot of the widget shown at the end of training:

Run Properties

Status	Completed
Start Time	8/10/2018 12:11:42 PM
Duration	0:07:20
Run Id	sklearn-mnist_1533921100384
Arguments	N/A
regularization rate	0.01
accuracy	0.9185

Output Logs

Uploading experiment status to history service.
Adding run profile attachment azureml-logs/80_driver_log.txt

Data folder: /mnt/batch/tasks/shared/LS_root/jobs/gpucluster225c81517743bf5/azureml/sklearn-mnist_1533921100384/mounts/workspacefilestore/mnist
(60000, 784)
(60000,)
(10000, 784)
(10000,)
Train a logistic regression model with regularizaion rate of 0.01
Predict the test set
Accuracy is 0.9185
The experiment completed successfully. Starting post-processing steps.

[Click here to see the run in Azure portal](#)

Step 8 – See the results

As model training and monitoring happen in the background. Wait until the model has completed training before running more code. Use [wait_for_completion](#) to show when the model training is complete

```
run.wait_for_completion(show_output=False)
```

-----> Specify 'True' for a verbose log

```
# now there is a trained model on the remote cluster
```

```
print(run.get_metrics())
```

-----> Displays the accuracy of the model. You should see an output that looks like this.

```
{'regularization rate': 0.8, 'accuracy': 0.9204}
```

Step 9 – Register the model

Recall that the last step in the training script is:

```
joblib.dump(value=clf, filename='outputs/sklearn_mnist_model.pkl')
```

This wrote the file `'outputs/sklearn_mnist_model.pkl'` in a directory named `'outputs'` in the VM of the cluster where the job is executed.

- `outputs` is a special directory in that all content in this directory is automatically uploaded to your workspace.
- This content appears in the run record in the experiment under your workspace.
- Hence, the model file is now also available in your workspace.

```
# register the model in the workspace
model = run.register_model (
    model_name='sklearn_mnist',
    model_path='outputs/sklearn_mnist_model.pkl')
```

The model is now available to query, examine, or deploy

Step 9 – Deploy the Model

Deploy the model registered in the previous slide, to Azure Container Instance (ACI) as a Web Service

There are 4 steps involved in model deployment

Step 9.1 – Create scoring script

Step 9.2 – Create environment file

Step 9.3 – Create configuration file

Step 9.4 – Deploy to ACI!

Step 9.1 – Create the scoring script

Create the scoring script, called `score.py`, used by the web service call to show how to use the model. It requires two functions – `init()` and `run(input data)`

```
from azureml.core.model import Model

def init():
    global model
    # retrieve the path to the model file using the model name
    model_path = Model.get_model_path('sklearn_mnist')
    model = joblib.load(model_path)

def run(raw_data):
    data = np.array(json.loads(raw_data)['data'])
    # make prediction
    y_hat = model.predict(data)
    return json.dumps(y_hat.tolist())
```

The `init()` function, typically loads the model into a global object. This function is run only once when the Docker container is started.

The `run(input_data)` function uses the model to predict a value based on the input data. Inputs and outputs to the run typically use JSON for serialization and de-serialization, but other formats are supported

Step 9.2 – Create environment file

Create an environment file, called `myenv.yml`, that specifies all of the script's package dependencies. This file is used to ensure that all of those dependencies are installed in the Docker image. This example needs `scikit-learn` and `azureml-sdk`.

```
from azureml.core.conda_dependencies import CondaDependencies

myenv = CondaDependencies()
myenv.add_conda_package("scikit-learn")

with open("myenv.yml", "w") as f:
    f.write(myenv.serialize_to_string())
```

Step 9.3 – Create configuration file

Create a deployment configuration file and specify the number of CPUs and gigabyte of RAM needed for the ACI container. Here we will use the defaults (1 core and 1 gigabyte of RAM)

```
from azureml.core.webservice import AciWebservice

aciconfig = AciWebservice.deploy_configuration(cpu_cores=1, memory_gb=1,
                                              tags={"data": "MNIST", "method": "sklearn"},
                                              description='Predict MNIST with sklearn')
```

Step 9.4 – Deploy the model to ACI

```
%%time
from azureml.core.webservice import Webservice
from azureml.core.image import ContainerImage
```

```
# configure the image
```

```
image_config = ContainerImage.image_configuration(
    execution_script = "score.py",
    runtime = "python",
    conda_file = "myenv.yml")
```

```
service = Webservice.deploy_from_model(workspace=ws, name='sklearn-mnist-svc',
    deployment_config=aciconfig, models=[model],
    image_config=image_config)
```

```
service.wait_for_deployment(show_output=True) -----> Start up a container in ACI using the image
```

Build an image using:

- The scoring file (score.py)
- The environment file (myenv.yml)
- The model file

Register that image under the workspace and send the image to the ACI container.

Step 10 – Test the deployed model using the HTTP end point

Test the deployed model by sending images to be classified to the HTTP endpoint

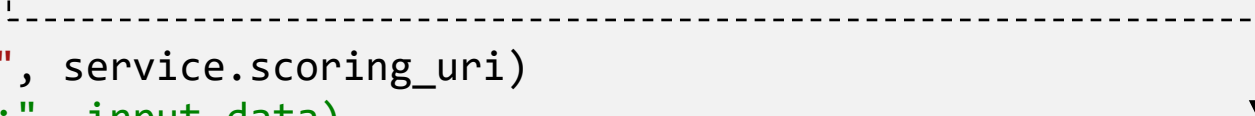
```
import requests
import json

# send a random row from the test set to score
random_index = np.random.randint(0, len(X_test)-1)
input_data = "{\"data\": [" + str(list(X_test[random_index])) + "]}"

headers = {'Content-Type': 'application/json'}

resp = requests.post(service.scoring_uri, input_data, headers=headers)

print("POST to url", service.scoring_uri)
#print("input data:", input_data)
print("label:", y_test[random_index])
print("prediction:", resp.text)
```

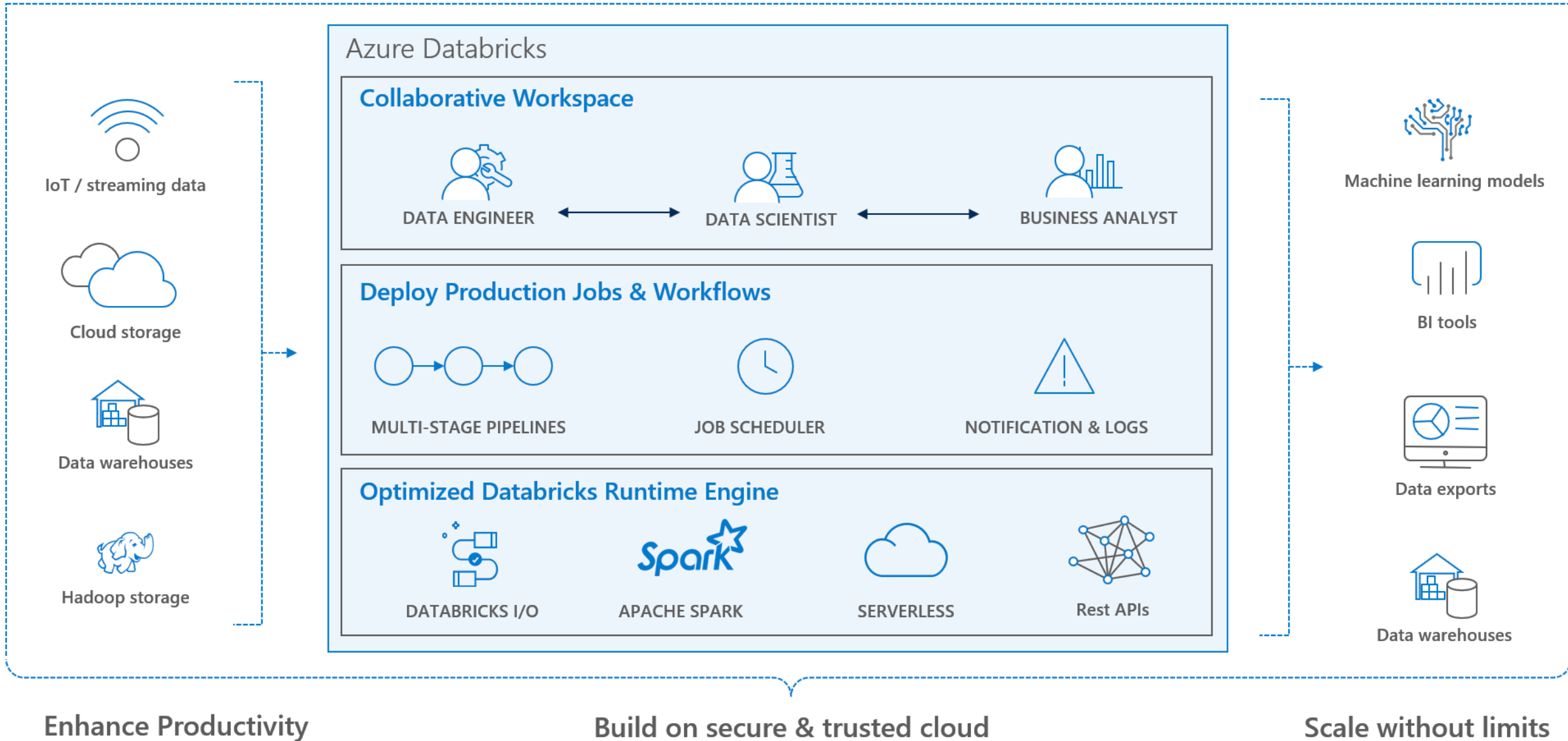


Send the data to the HTTP end-point for scoring



Databricks

Azure Databricks



Azure Databricks

Microsoft Azure

erikzwi@microsoft.com

PORTAL

Azure Databricks

Home

Workspace

Recent

Data

Clusters

Jobs

Search

taxi-demo-file-loading (Python)

Attached: training File View: Code Permissions Stop Execution Clear

Schedule Comments Revision history

Cmd 1

NYC Taxi Commission Dataset

A demo of Databricks functionality using the publically available dataset from the [NYC Taxi Commission](#)

Cmd 2

Mount blob ADLS in DBFS applicationId = '974d7063-4897-453f-967d-c022c6c4fa52' ...

Cmd 3

Create Schema Object from pyspark.sql.types import * greenSchema = StructType ...

▶ greenDF: pyspark.sql.dataframe.DataFrame = [vendor_id: integer, pickup_datetime: timestamp ... 18 more fields]

Cmd 4

```
filteredGreenDF = (greenDF.filter("pickup_longitude<0")
                          .filter("pickup_longitude>-78")
                          .filter("pickup_latitude>0")
                          .sample(False, 0.3, 11234)
                          )

latitudes = filteredGreenDF.rdd.map(lambda x: x['pickup_latitude']).collect()
longitudes = filteredGreenDF.rdd.map(lambda x: x['pickup_longitude']).collect()
```

▼ (2) Spark Jobs Cancel

▼ Job 40 View (1 stages)

Stage 41: 22/22 (0 running) ⓘ

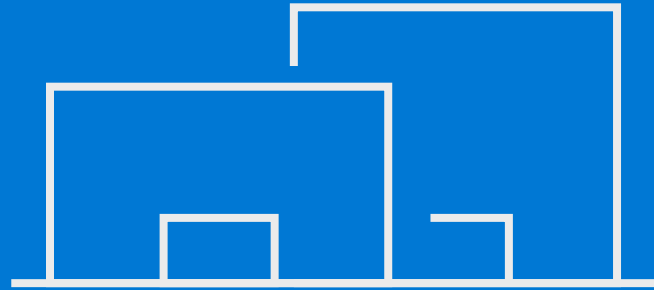
▶ Job 41 View (1 stages)

▶ filteredGreenDF: pyspark.sql.dataframe.DataFrame = [vendor_id: integer, pickup_datetime: timestamp ... 18 more fields]

Cmd 5

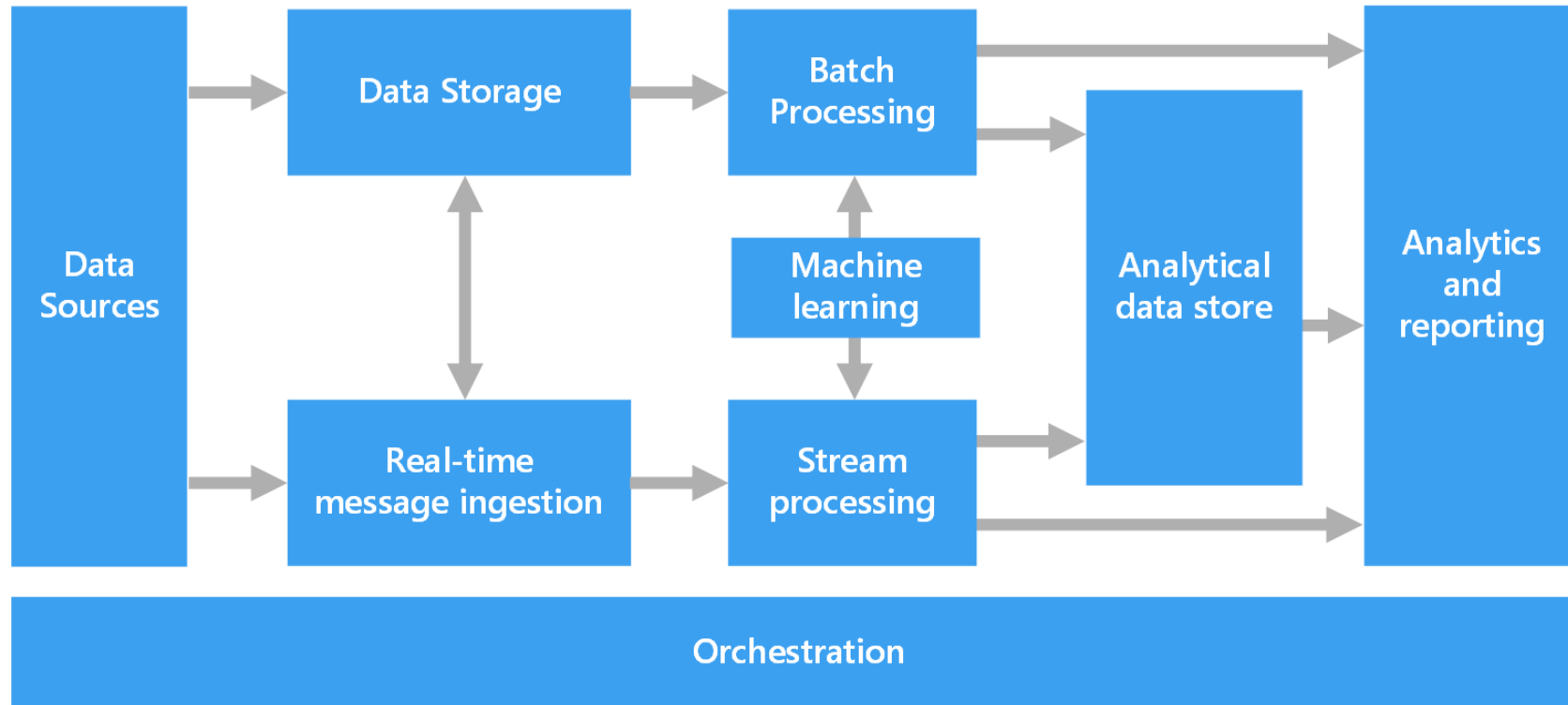
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.scatter(longitude, latitude)
```



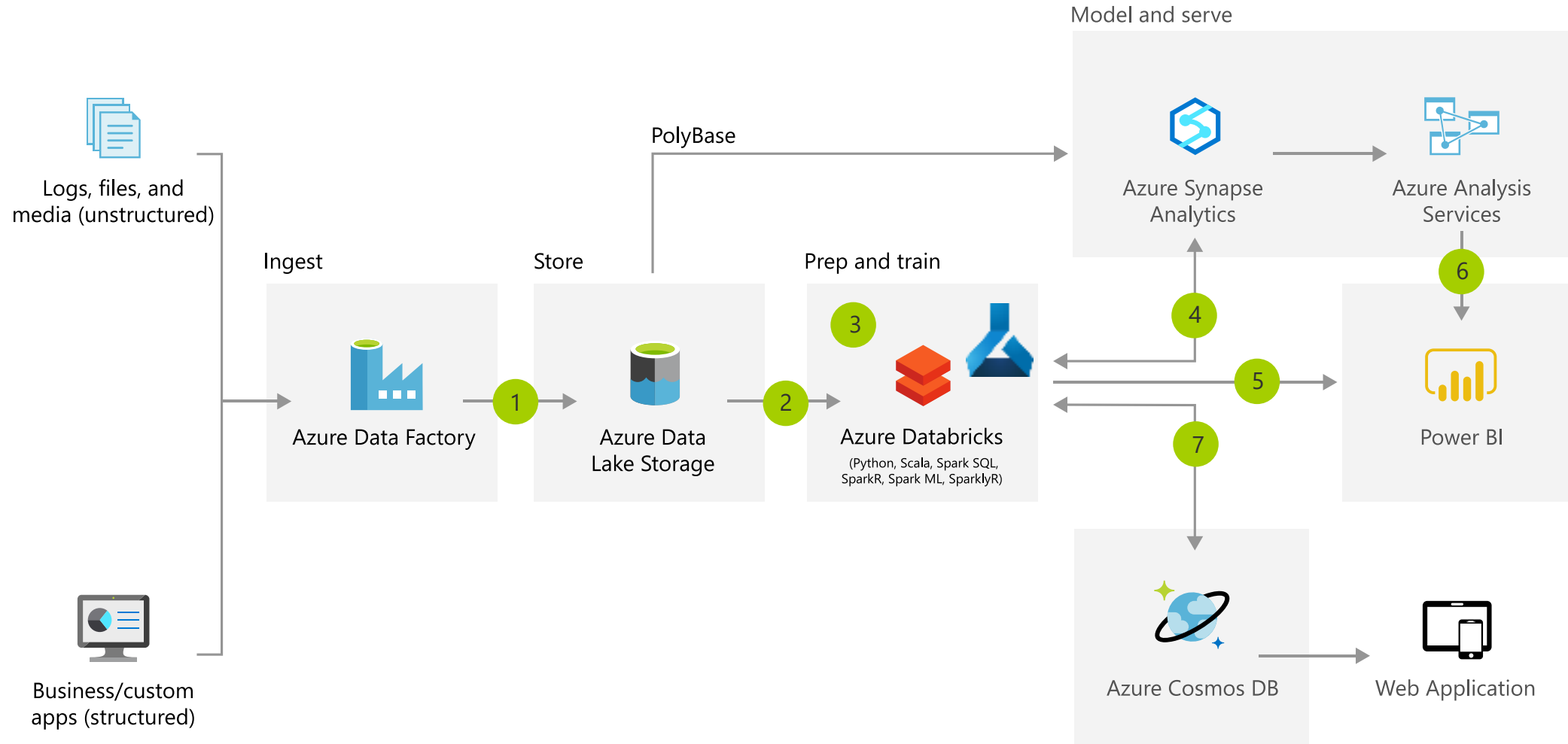
Analytics Data Architecture

Analytics Data Architecture



[Link to Big Data Architecture Documentation](#)

Analytics Data Architecture





[Link to Azure Architecture Documentation](#)

eSmart architecture

Data Sources


Drone collected images
Batch upload of drone images




Ingest


Azure Blob

Cosmos DB

Prepare


Azure Machine Learning

Analyze


TensorFlow

Azure ML compute

Docker Image
DNN contained in a Docker image

Publish


Cosmos DB
Contain inventory results and state changes

Intelligent Edge
Models deployed to drones for accelerated inferencing

Consume

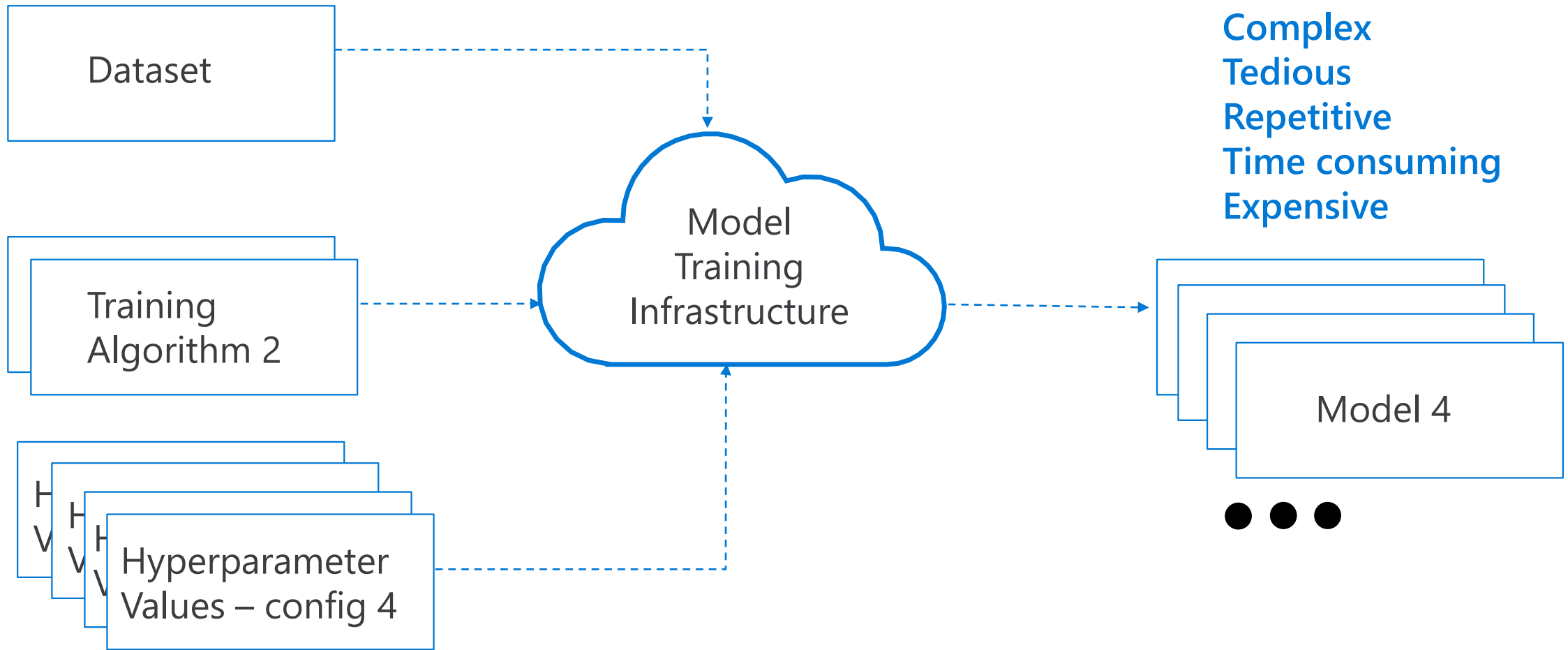
On-prem command center


Azure ML service



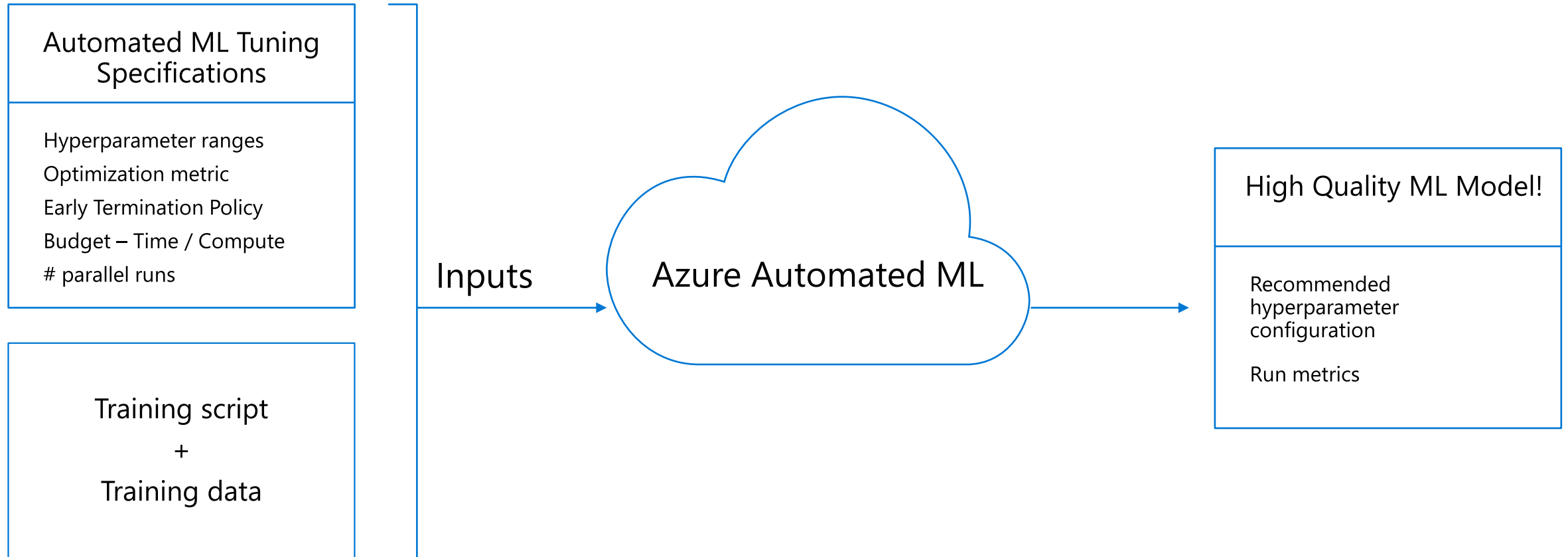
**Azure Automated Machine Learning
'simplifies' the creation and selection
of the optimal model
(Bonus)**

Typical 'manual' approach to hyperparameter tuning



Automated ML

Conceptual Overview



Automated ML

Current Capabilities

Category		Value
ML Problem Spaces		Classification Regression Forecasting
Frameworks		Scikit Learn, Lightgbm
Languages		Python
Data Type and Data Formats		Numerical Text Scikit-learn supported data formats (Numpy, Pandas)
Data sources		Local Files, Azure Blob Storage
Compute Target	Automated Hyperparameter Tuning	Azure ML Compute (Batch AI), Azure Databricks
	Automated Model Selection	Local Compute, Azure ML Compute (Batch AI), Azure Databricks

Automated ML

Algorithms Supported

Classification

`sklearn.linear_model.LogisticRegression`

`sklearn.linear_model.SGDClassifier`

`sklearn.naive_bayes.BernoulliNB`

`sklearn.naive_bayes.MultinomialNB`

`sklearn.svm.SVC`

`sklearn.svm.LinearSVC`

`sklearn.calibration.CalibratedClassifierCV`

`sklearn.neighbors.KNeighborsClassifier`

`sklearn.tree.DecisionTreeClassifier`

`sklearn.ensemble.RandomForestClassifier`

`sklearn.ensemble.ExtraTreesClassifier`

`sklearn.ensemble.GradientBoostingClassifier`

`lightgbm.LGBMClassifier`

Regression

`sklearn.linear_model.ElasticNet`

`sklearn.ensemble.GradientBoostingRegressor`

`sklearn.tree.DecisionTreeRegressor`

`sklearn.neighbors.KNeighborsRegressor`

`sklearn.linear_model.LassoLars`

`sklearn.linear_model.SGDRegressor`

`sklearn.ensemble.RandomForestRegressor`

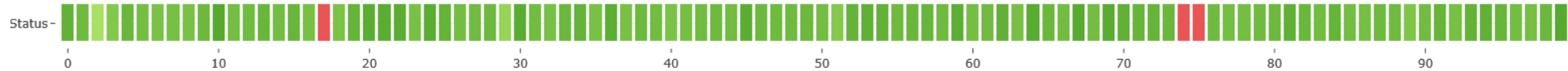
`sklearn.ensemble.ExtraTreesRegressor`






`lightgbm.LGBMRegressor`

Azure Automated ML – Sample Output

AutoML_ab755820-4bfd-4e8a-8b4b-9e0a2446b1c2:

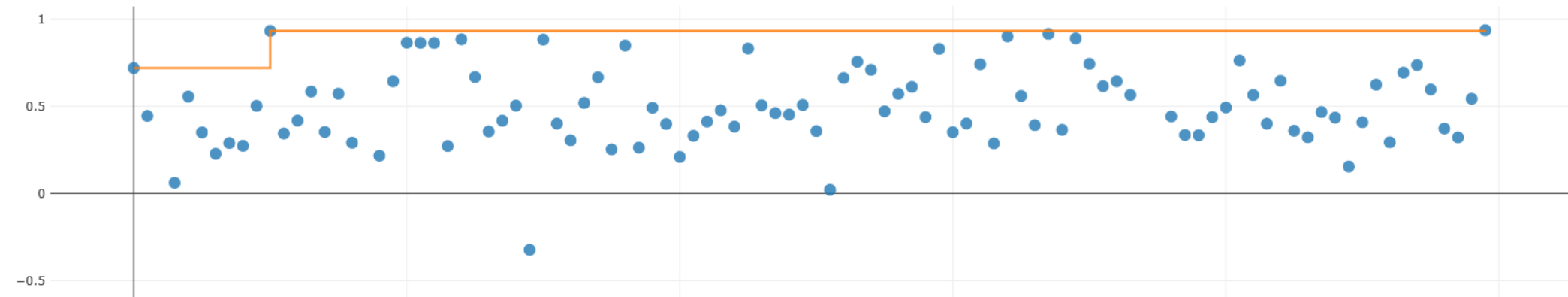
Status: Completed



Iteration	Pipeline	Iteration metric	Best metric	Status	Duration	Started	Run Id
99	Ensemble	0.93702349	0.93702349	Completed	0:02:18	Dec 4, 2018 12:18 AM	
10	MaxAbsScaler, LightGBM	0.93289307	0.93289307	Completed	0:01:22	Dec 3, 2018 7:49 PM	
67	SparseNormalizer, LightGBM	0.9154763	0.93289307	Completed	0:01:31	Dec 3, 2018 10:19 PM	
64	MaxAbsScaler, LightGBM	0.90148724	0.93289307	Completed	0:01:24	Dec 3, 2018 10:09 PM	
69	MaxAbsScaler, LightGBM	0.88975241	0.93289307	Completed	0:00:55	Dec 3, 2018 10:22 PM	

Pages: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ... Next Last 5 per page

AutoML Run with metric : r2_score



Automated ML

Use via the Python SDK

jupyter102.auto-ml-regression(unsaved changes)

FileEditViewInsertCellKernelWidgetsHelp

Not TrustedPython [default]

+

⌂

⌕

📄

⬆️

⬆️

⏮️

⏪️

⏩️

⏭️

Code

🗨️

Instantiate Auto ML Regressor

Instantiate a AutoML Object This creates an Experiment in Azure ML. You can reuse this objects to trigger multiple runs. Each run will be part of the same experiment.

Property	Description
primary_metric	This is the metric that you want to optimize. Auto ML Regressor supports the following primary metrics spearman_correlation normalized_root_mean_squared_error r2_score
max_time_sec	Time limit in seconds for each iterations
iterations	Number of iterations. In each iteration Auto ML Classifier trains the data with a specific pipeline
num_cross_folds	Cross Validation split

```
In [5]: from azureml.train.automl import AutoMLConfig

automl_config = AutoMLConfig(task = 'regression',
                             debug_log = 'automl_errors.log',
                             primary_metric = 'spearman_correlation',
                             max_time_sec = 12000,
                             iterations = 10,
                             n_cross_validations = 3,
                             verbosity = logging.INFO,
                             X = X,
                             y = y,
                             path=project_folder)
```

Training the Model

You can call the fit method on the AutoML instance and pass the run configuration. For Local runs the execution is synchronous. Depending on the data and number of iterations this can run for while. You will see the currently running iterations printing to the console.

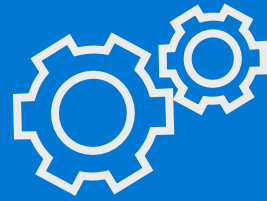
fit method on Auto ML Regressor triggers the training of the model. It can be called with the following parameters

Parameter	Description
X	(sparse) array-like, shape = [n_samples, n_features]
y	(sparse) array-like, shape = [n_samples,], [n_samples, n_classes] Multi-class targets. An indicator matrix turns on multilabel classification.
compute_target	Indicates the compute used for training. /local indicates train on the same compute which hosts the jupyter notebook. For DSVM and Batch AI please refer to the relevant notebooks.
show_output	True/False to turn on/off console output

```
In [6]: local_run = experiment.submit(automl_config, show_output=True)
```

```
Parent Run ID: AutoML_e7a4236e-8935-4e93-888d-1ea8310a6b22
*****
ITERATION: The iteration being evaluated.
PIPELINE: A summary description of the pipeline being evaluated.
DURATION: Time taken for the current iteration.
METRIC: The result of computing score on the fitted pipeline.
BEST: The best observed score thus far.
*****
```

ITERATION	PIPELINE	DURATION	METRIC	BEST
0	Normalize extra trees regressor	0:00:12.069893	0.688	0.688
1	Normalize lightGBM regressor	0:00:11.192919	0.597	0.688
2	Normalize Elastic net	0:00:09.866233	0.689	0.689
3	Scale 0/1 lightGBM regressor	0:00:10.069764	0.656	0.689
4	Robust Scaler kNN regressor	0:00:09.090668	0.598	0.689
5	Normalize lightGBM regressor	0:00:12.562876	0.649	0.689
6	Robust Scaler kNN regressor	0:00:09.361137	0.600	0.689
7	Normalize SGD regressor	0:00:09.010672	0.070	0.689
8	Scale 0/1 extra trees regressor	0:00:10.442752	0.685	0.689
9	Robust Scaler Gradient boosting regres	0:00:09.567582	0.651	0.689



Automated Hyperparameter Tuning (Bonus)

Automated Hyperparameter Tuning

Sampling to generate new runs

Define hyperparameter search space

```
{  
  "learning_rate": uniform(0, 1),  
  "num_layers": choice(2, 4, 8)  
  ...  
}
```

Sampling
algorithm



```
Config1= {"learning_rate": 0.2,  
  "num_layers": 2, ...}
```

```
Config2= {"learning_rate": 0.5,  
  "num_layers": 4, ...}
```

```
Config3= {"learning_rate": 0.9,  
  "num_layers": 8, ...}
```

...

Supported sampling algorithms:

Grid Sampling

Random Sampling

Bayesian Optimization

Automated Hyperparameter Tuning

Manage Active Jobs

Evaluate training runs for specified primary metric

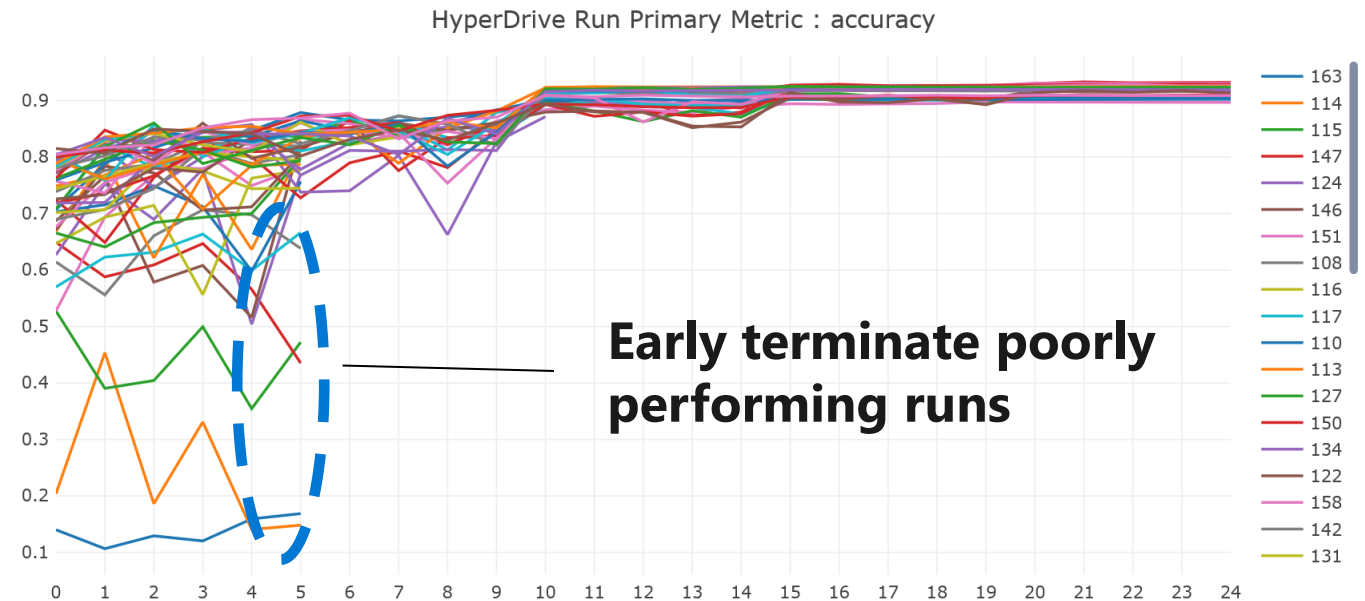
Use resources to explore new configurations

Early terminate poor performing training runs. Early termination policies include:

- Bandit policy

- Median Stopping policy

- Truncation Selection policy



Learn more

[Link to demo repo](#)

<https://docs.microsoft.com/en-us/azure/machine-learning/>

<https://github.com/Azure/MachineLearningNotebooks>

<https://docs.microsoft.com/en-gb/azure/machine-learning/team-data-science-process/overview>

<https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/advanced-analytics-on-big-data>

<https://github.com/interpretml/interpret>

