# Operationalizing Machine Learning

## Overview

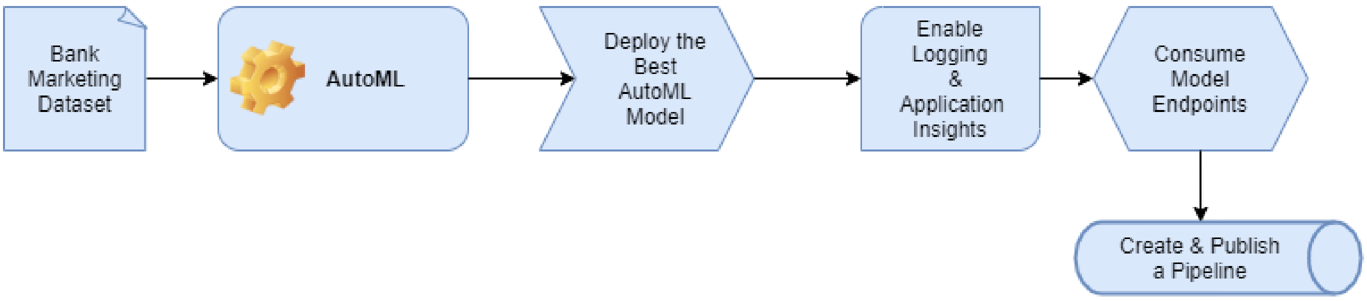
This project is part of the Udacity Azure ML Nanodegree.

In this project, we use Azure to **configure a cloud-based machine learning production model, deploy it, and consume it**. We will also create, publish, and consume a pipeline.

## Architectural Diagram

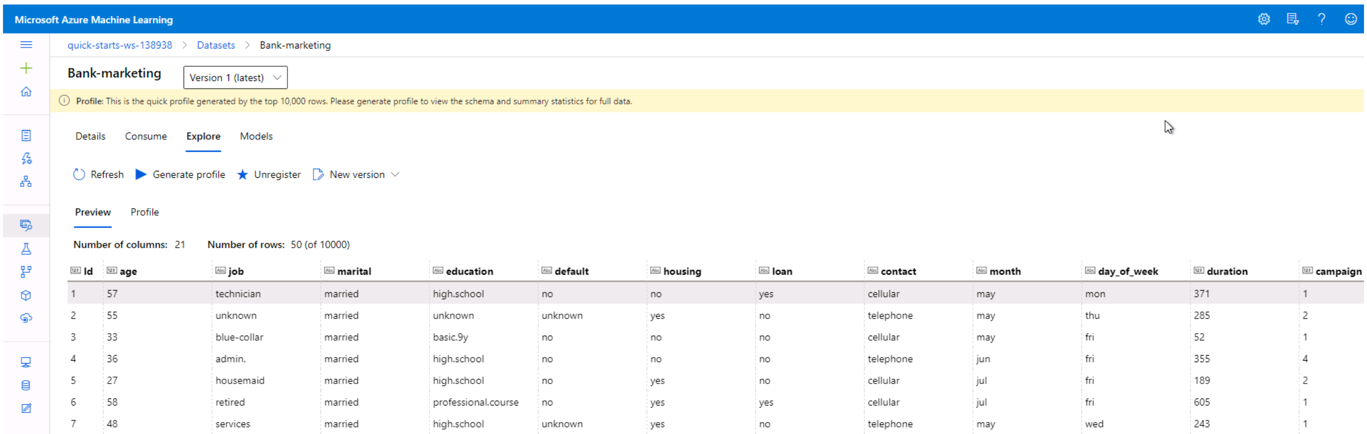
In this project, we explain briefly how to publish your best AutoML model and deploy it as a Web API.

- **Experiment Run**: Using MLStudio we manually upload data, select the compute target and the task we want to accomplish
- **Best Model Selection**: Comparing all the models, finally we choose the one with better primary metric, in this case, weighted AUC, according to the target unbalance.
- **Model Deployment**: The best model is deployed as an endpoint.
- **Application Insights Activation**: We enable this logs monitoring tool.
- **Display Swagger Documentation**: We make use of the *swagger.json* file given by Azure to visualize documentation in a more clear and professional way.
- **Consume Endpoint**: We do a test and a benchmark to check out that the endpoint works and to better know latency times.
- **Create, Publish and Consume a Pipeline using Python SDK**: In the top of Automation we have the pipelines. Processing data and retraining only with an HTTP request is possible.

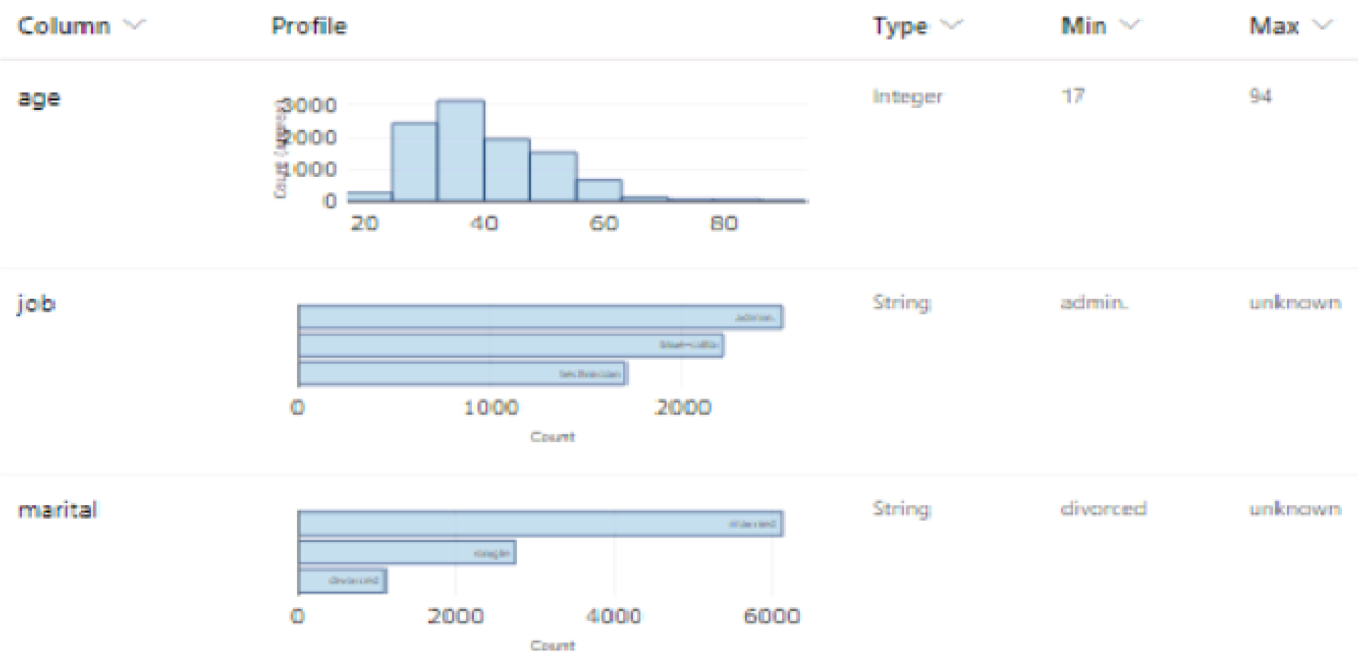



## Key Steps

### 1. Data Preparation: When AutoML run is created, *Bank-Marketing* data is uploaded and registered so that we can use it in our experiments.




**Personal information**: Occupation, age, marital status and education.



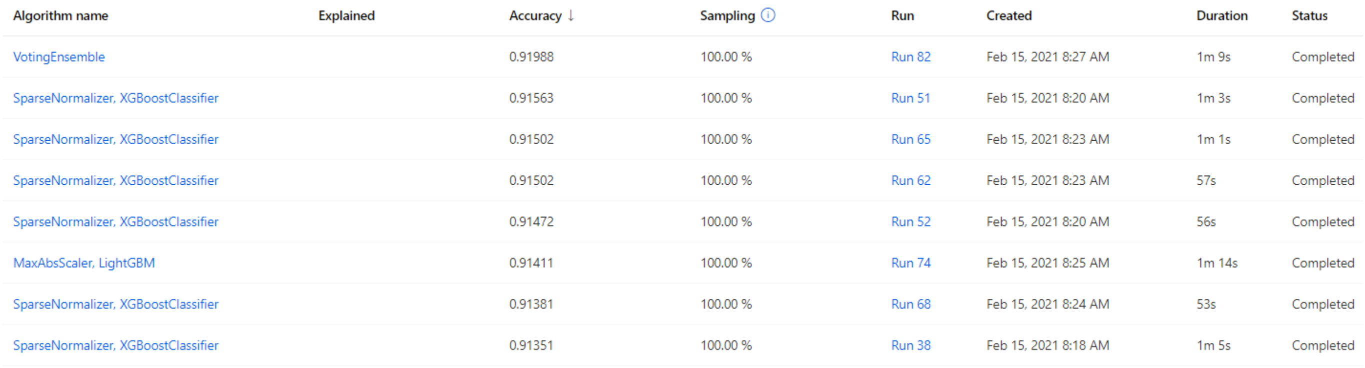

**Financial information**: Debts and other data about customer's financial health.

**Target**: The variable we seek to predict is the one that tells us if a given person is a potential client or not.

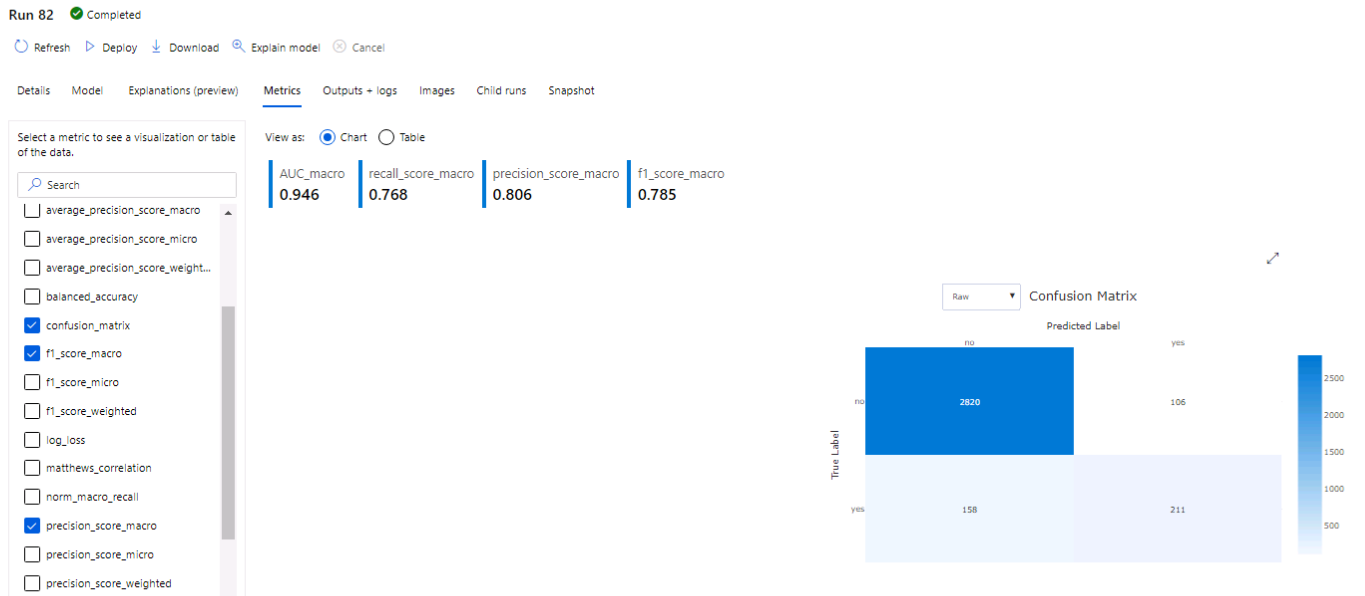

## 2. **Experiment Run**: AutoML experiment correctly run and submitted.


## 3. **Best Model**: The best performing model is the one using *VotingEnsemble*.

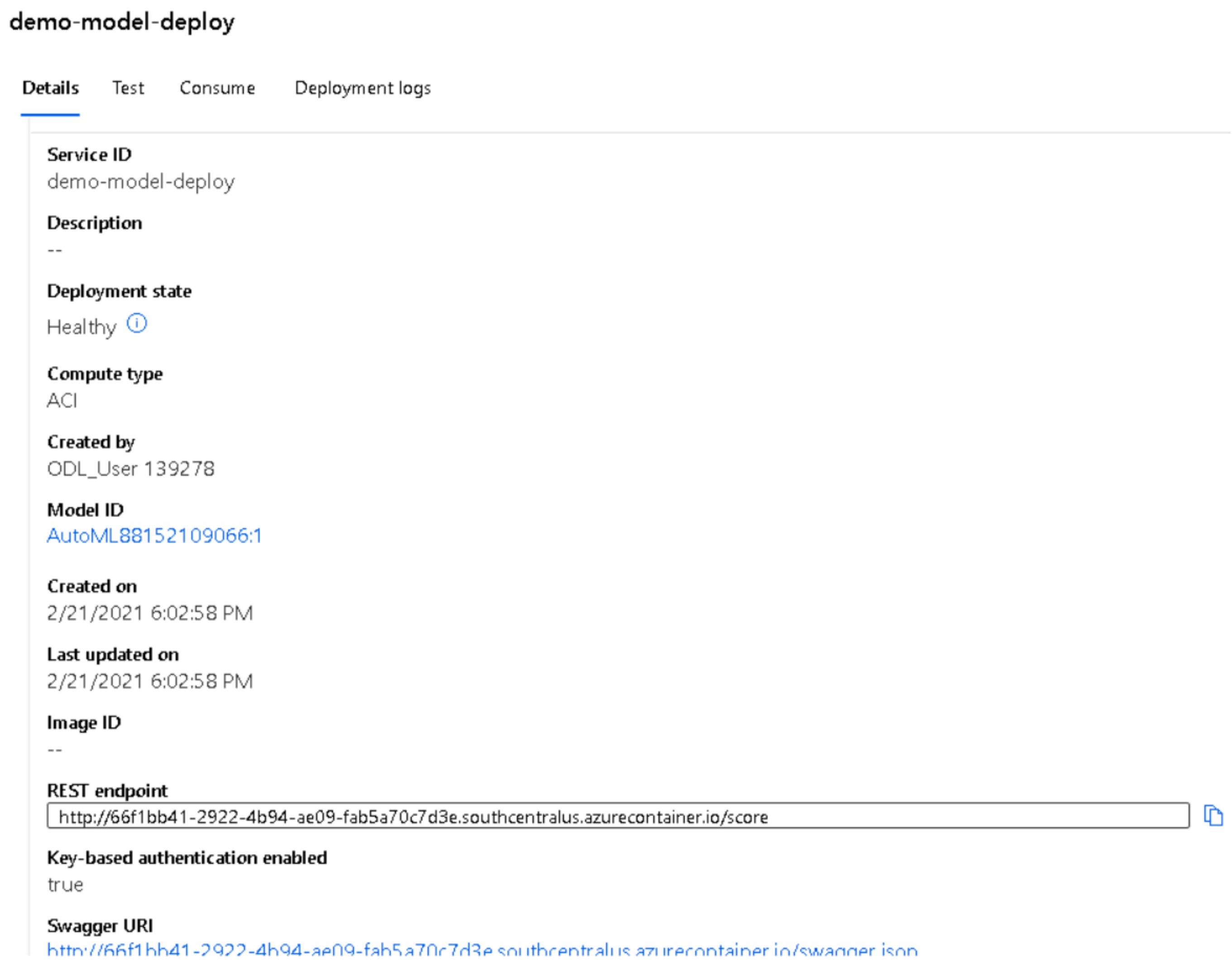
As it was mentioned in **ML-Pipeline** project, due to the high target unbalancement, we're going to focus on *macro* metrics.



| Algorithm name | Explained | Accuracy ↓ | Sampling | Run | Created | Duration | Status |
|---|---|---|---|---|---|---|---|
| VotingEnsemble | | 0.91988 | 100.00 % | Run 82 | Feb 15, 2021 8:27 AM | 1m 9s | Completed |
| SparseNormalizer, XGBoostClassifier | | 0.91563 | 100.00 % | Run 51 | Feb 15, 2021 8:20 AM | 1m 3s | Completed |
| SparseNormalizer, XGBoostClassifier | | 0.91502 | 100.00 % | Run 65 | Feb 15, 2021 8:23 AM | 1m 1s | Completed |
| SparseNormalizer, XGBoostClassifier | | 0.91502 | 100.00 % | Run 62 | Feb 15, 2021 8:23 AM | 57s | Completed |
| SparseNormalizer, XGBoostClassifier | | 0.91472 | 100.00 % | Run 52 | Feb 15, 2021 8:20 AM | 56s | Completed |
| MaxAbsScaler, LightGBM | | 0.91411 | 100.00 % | Run 74 | Feb 15, 2021 8:25 AM | 1m 14s | Completed |
| SparseNormalizer, XGBoostClassifier | | 0.91381 | 100.00 % | Run 68 | Feb 15, 2021 8:24 AM | 53s | Completed |
| SparseNormalizer, XGBoostClassifier | | 0.91351 | 100.00 % | Run 38 | Feb 15, 2021 8:18 AM | 1m 5s | Completed |

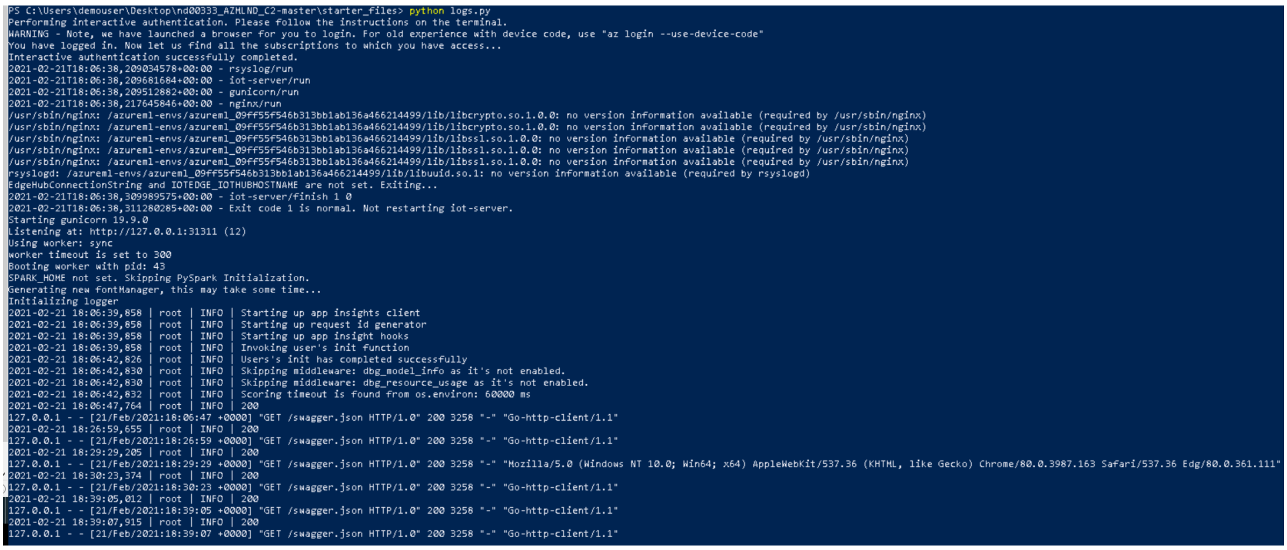



**4. Deploy Model**: Create an endpoint associated to an ACI.

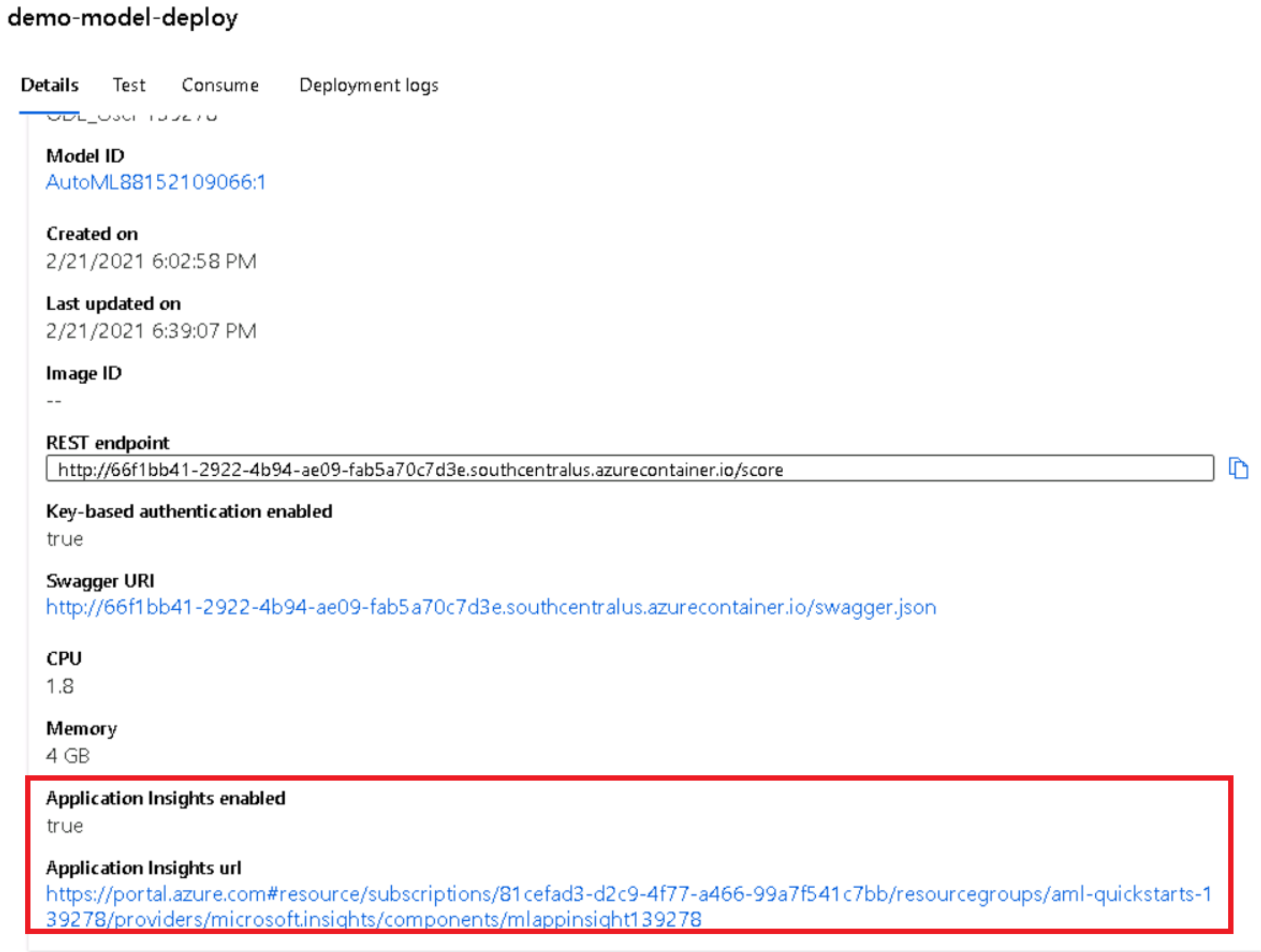


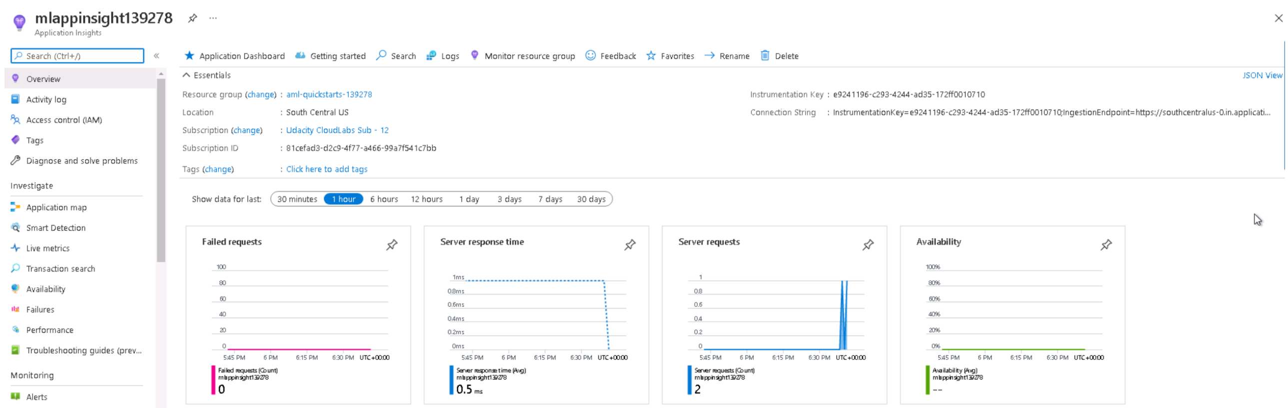

## 5. **Activate Application Insights**:

1. Check out *az* extension is installed with `az version` and `az extension add -n azure-cli-ml`.
2. Create a python virtual environment with `virtualenv venv`.
3. Edit and run `logs.py` writing the endpoint name (`demo-model-deploy` in this case) and check output in console:



4. Check that **Application Insights** is enabled and wotking:

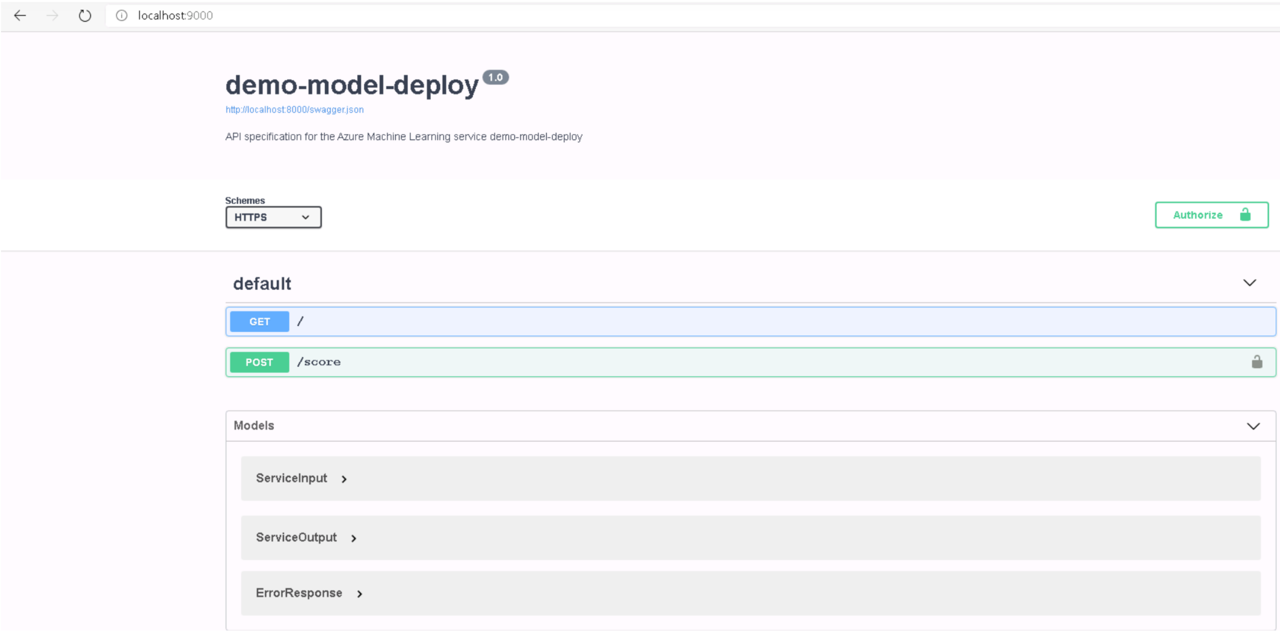



## 6. Display Swagger Documentation

1. Download swagger.json from *Details* tab inside the endpoint.
2. Change port from recommended `80` to `9000` in `swagger.sh` because 80 is not available and run it to start the `swagger-ui` docker container. Now it is displayed in `http://localhost:9000/`.
3. Run server.py in 8000 port

4. Check `http://localhost:8000/swagger.json` in `Swagger`.




## 7. Consume Endpoint and benchmarking

1. Edit *endpoint.py* with the URL and the key necessary to authorize the HTTP request. Some data in a json format is used to make the request and see what the best model predicts from it.

```
  GNU nano 4.9.3
import requests
import json

# URL for the web service, should be similar to:
# 'http://8530a665-66f3-49c8-a953-b82a2d312917.eastus.azurecontainer.io/score'
scoring_uri = 'http://66f1bb41-2922-4b94-ae09-fab5a70c7d3e.southcentralus.azurecontainer.io/score'
# If the service is authenticated, set the key or token
key = 'UQTwrNWHmpVlcGdwxwYXWdJNYId9jfW8'

# Two sets of data to score, so we get two results back
data = {"data":
        [
          {
            "age": 17,
            "campaign": 1,
            "cons.conf.idx": -46.2,
            "cons.price.idx": 92.893,
            "contact": "cellular",
            "day_of_week": "mon",
            "default": "no",
            "duration": 971,
            "education": "university.degree",
            "emp.var.rate": -1.8,
            "euribor3m": 1.299,
            "housing": "yes",
            "job": "blue-collar",
            "loan": "yes",
            "marital": "married",
            "month": "may",
            "nr.employed": 5099.1,
            "pdays": 999,
            "poutcome": "failure",
            "previous": 1
          },
          {
            "age": 87,
            "campaign": 1,
            "cons.conf.idx": -46.2,
            "cons.price.idx": 92.893,
            "contact": "cellular",
            "day_of_week": "mon",
            "default": "no",
            "duration": 471,
            "education": "university.degree",
            "emp.var.rate": -1.8,
            "euribor3m": 1.299,
            "housing": "yes",
            "job": "blue-collar",
            "loan": "yes",
            "marital": "married",
            "month": "may",
            "nr.employed": 5099.1,
            "pdays": 999,
            "poutcome": "failure",
            "previous": 1
          },
      ]
    }
# Convert to JSON string
input_data = json.dumps(data)
with open("data.json", "w") as _f:
    _f.write(input_data)

# Set the content type
headers = {'Content-Type': 'application/json'}
# If authentication is enabled, set the authorization header
headers['Authorization'] = f'Bearer {key}'

# Make the request and display the response
resp = requests.post(scoring_uri, input_data, headers=headers)
print(resp.json())
```

```
PS C:\Users\demouser\Desktop\nd00333_AZMLND_C2-master\starter_files> python .\endpoint.py
{"result": ["yes", "no"]}
```

2. Now we want to know the average time necessary to receiver our responses. For that we use **Apache Benchmark**. First of all we check that it is installed displaying the help `ab -h`. Once we see that it works, Authorization key and URI can be introduced into *benchmark.sh* and run the script.

```
  GNU nano 4.9.3                                                    benchmark.sh
# HTTP/1.0 200 OK
# Content-Length: 33
# Content-Type: application/json
# Date: Thu, 30 Jul 2020 12:33:34 GMT
# Server: nginx/1.10.3 (Ubuntu)
# X-Ms-Request-Id: b48dd8da-0b4e-44fd-a1e5-04043bfa77f1
# X-Ms-Run-Function-Failed: False
#
# "{\"result\": [\"yes\", \"no\"]}"
# LOG: Response code = 200
# ..done
#
#
# Server Software:        nginx/1.10.3
# Server Hostname:        8530a665-66f3-49c8-a953-b82a2d312917.eastus.azurecontainer.io
# Server Port:            80
#
# Document Path:          /score
# Document Length:        33 bytes
#
# Concurrency Level:      1
# Time taken for tests:   1.599 seconds
# Complete requests:      10
# Failed requests:        0
# Total transferred:      2600 bytes
# Total body sent:        10560
# HTML transferred:       330 bytes
# Requests per second:    6.25 [#/sec] (mean)
# Time per request:       159.918 [ms] (mean)
# Time per request:       159.918 [ms] (mean, across all concurrent requests)
# Transfer rate:          1.59 [Kbytes/sec] received
#                         6.45 kb/s sent
#                         8.04 kb/s total
#
# Connection Times (ms)
#               min  mean[+/-sd] median   max
# Connect:       21   23   0.8     23      24
# Processing:    92  137  28.3    151     176
# Waiting:       92  137  28.3    151     176
# Total:        114  160  28.0    172     199#

 ab -n 10 -v 4 -p data.json -T 'application/json' -H 'Authorization: Bearer UQTwrNWHnpVlcGdwxwYxWdJNYId9jfW8' http://66f1bb41-2922-4b94-ae09-fab5a70c7d3e.southcentralus.azurecontainer.io/score
```

```
Server Software:        nginx/1.10.3
Server Hostname:        66f1bb41-2922-4b94-ae09-fab5a70c7d3e.southcentralus.azurecontainer.io
Server Port:            80

Document Path:          /score
Document Length:        33 bytes

Concurrency Level:      1
Time taken for tests:   2.269 seconds
Complete requests:      10
Failed requests:        0
Total transferred:      2600 bytes
Total body sent:        10640
HTML transferred:       330 bytes
Requests per second:    4.41 [#/sec] (mean)
Time per request:       226.853 [ms] (mean)
Time per request:       226.853 [ms] (mean, across all concurrent requests)
Transfer rate:          1.12 [Kbytes/sec] received
                        4.58 kb/s sent
                        5.70 kb/s total

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        2    3   0.5      3       4
Processing:   111  224  46.4    238     270
Waiting:      111  224  46.4    238     270
Total:        114  227  46.6    241     274

Percentage of the requests served within a certain time (ms)
  50%    241
  66%    252
  75%    261
  80%    266
  90%    274
  95%    274
  98%    274
  99%    274
 100%    274 (longest request)

demouser@labvm MINGW64 ~/Desktop/nd00333_AZMLND_C2-master/starter_files
$ |
```

We can see that, calling 10 times the API with a *data.json* POST (the same used for testing the endpoint) we see that all of them are done without issues in 226 ms in average. This is a very acceptable responsing time.

## 8. Create, Publish and Consume a Pipeline

1. Upload the Jupyter Notebook that contains the process of the Pipeline creation and change the experiment name so that it's the same used for AutoML.
2. Upload the `config.json` with the info about the workspace.
3. Using Python SDK, run the AutoML step and publish the Pipeline so that we can find it in MLStudio Pipelines section.

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

**Run 353e8550-331c-4521-973b-227bb9086440** Completed

```
[2021-02-21 20:34:53Z] Submitting 1 runs, first five are: b50ad284:7985a0dc-9d5d-40dd-92c6-c4bd0f5601e3
[2021-02-21 21:07:48Z] Completing processing run id 7985a0dc-9d5d-40dd-92c6-c4bd0f5601e3.

Run is completed.
```

quick-starts-ws-139278 > Pipelines

**Pipelines**

**Pipeline runs** Pipeline endpoints Pipeline drafts

+ New pipeline Refresh

Add filter

| Run | Run ID | Experiment | Status | Description | Submitted time ↓ | Duration | Submitted by | Tags |
|---|---|---|---|---|---|---|---|---|
| Run 12 | 876fa943-80d3-49c7-84d2-183925cd... | pipeline-rest-endpoint | Running | | Feb 21, 2021 9:13 PM | 2m 55s | ODL_User 139278 | azureml.pipelineid : 66990b07-cd9a-4921-a49f-6 |
| Run 1 | c0152fde-b659-4779-b1f6-7c5b4b0c... | pipeline-rest-endpoint | Running | | Feb 21, 2021 9:08 PM | 7m 56s | ODL_User 139278 | azureml.pipelineid : e518dbd4-3e89-40f6-85d0-4 |
| Run 74 | 353e8550-331c-4521-973b-227bb90... | p2_ml-ops-udacity | Completed | pipeline_with_automlst... | Feb 21, 2021 8:34 PM | 33m 1s | ODL_User 139278 | azureml.pipelineComponent : pipelinerun |

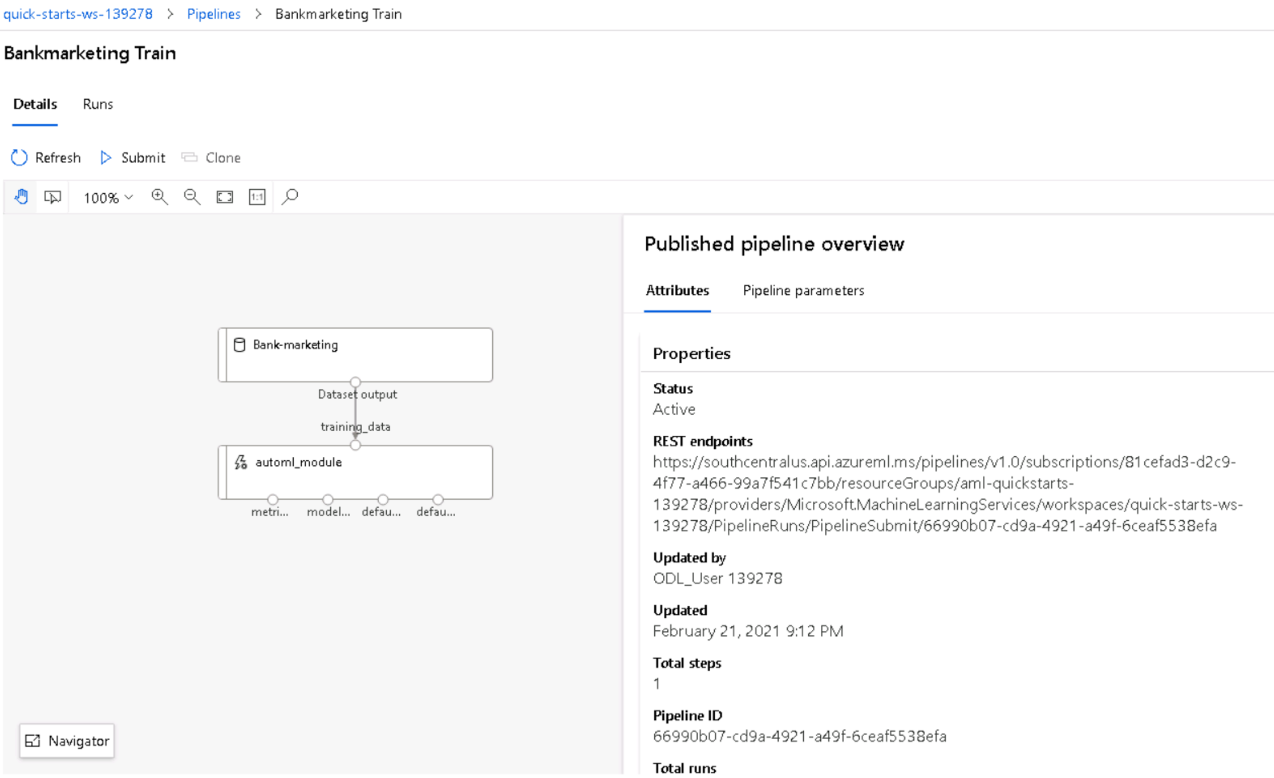
4. Deploy it creating an HTTP endpoint.

**Endpoints**

Real-time endpoints **Pipeline endpoints**

Refresh Disable Enable Include disabled

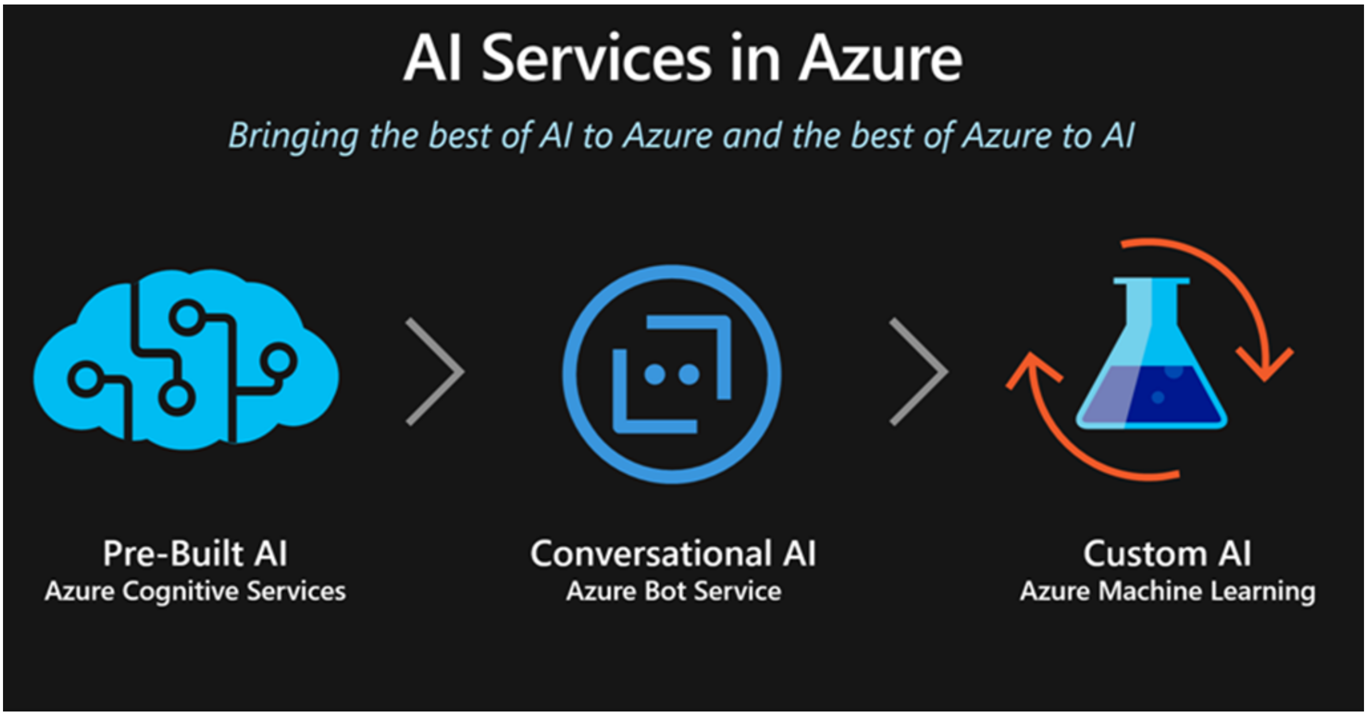
| Name ↓ | Description | Date updated | Updated by | Last run submit time | Last run status | Status |
|---|---|---|---|---|---|---|
| Bankmarketing Train | Training bank... | February 21, 2021 9:12 PM | ODL_User 139... | February 21, 2021 9:13 PM | Running | Active |
| Bankmarketing Train | Training bank... | February 21, 2021 9:08 PM | ODL_User 139... | February 21, 2021 9:08 PM | Running | Active |




5. Authenticate using interactive login and make a POST request with the new experiment name so that a new Pipeline run is executed.

## Screen Recording

Clicking on the following picture you can watch a **5 minutes video** with explanations about the content of the project.




## Standout Suggestions

For future work, I would suggest making further tweaks to the AutoML step of the pipeline. There are a lot of settings involved and making changes to them could improve the search space and help find an even better model solution. There are also additional steps that could be added to the pipeline, perhaps doing some dataset cleanup or feature engineering before the AutoML step, or doing additional steps after the AutoML step has completed.

In addition, I propose connecting Application Insights to our Pipeline API and Scheduling periodical runs or each time Banking Dataset updates (trigger).