Operationalizing an AWS ML Project

Fady Morris Ebeid [†]

January 20, 2022

[†]Email: fadymorris86@gmail.com

Contents

Project Directory Structure

```
doc
    writeup.pdf                               # Project writeup document.
ec2train1.py                                 # Python training script used inside EC2 instance.
hpo.py                                       # Hyperparameter tuning and model training script.
infernce2.py                                 # Endpoint inference script (entry point)
lambda-deployment-testing-security.ipynb        # Custom notebook to deploy and test lambda
   function.
lamdafunction.py                                # Lambda function entry point (modified to
   include the endpoint name)
README.md
screenshots                                  # Project screenshots.
train_and_deploy-solution.html               # HTML export of the solution notebook
train_and_deploy-solution.ipynb              # Solution notebook
```

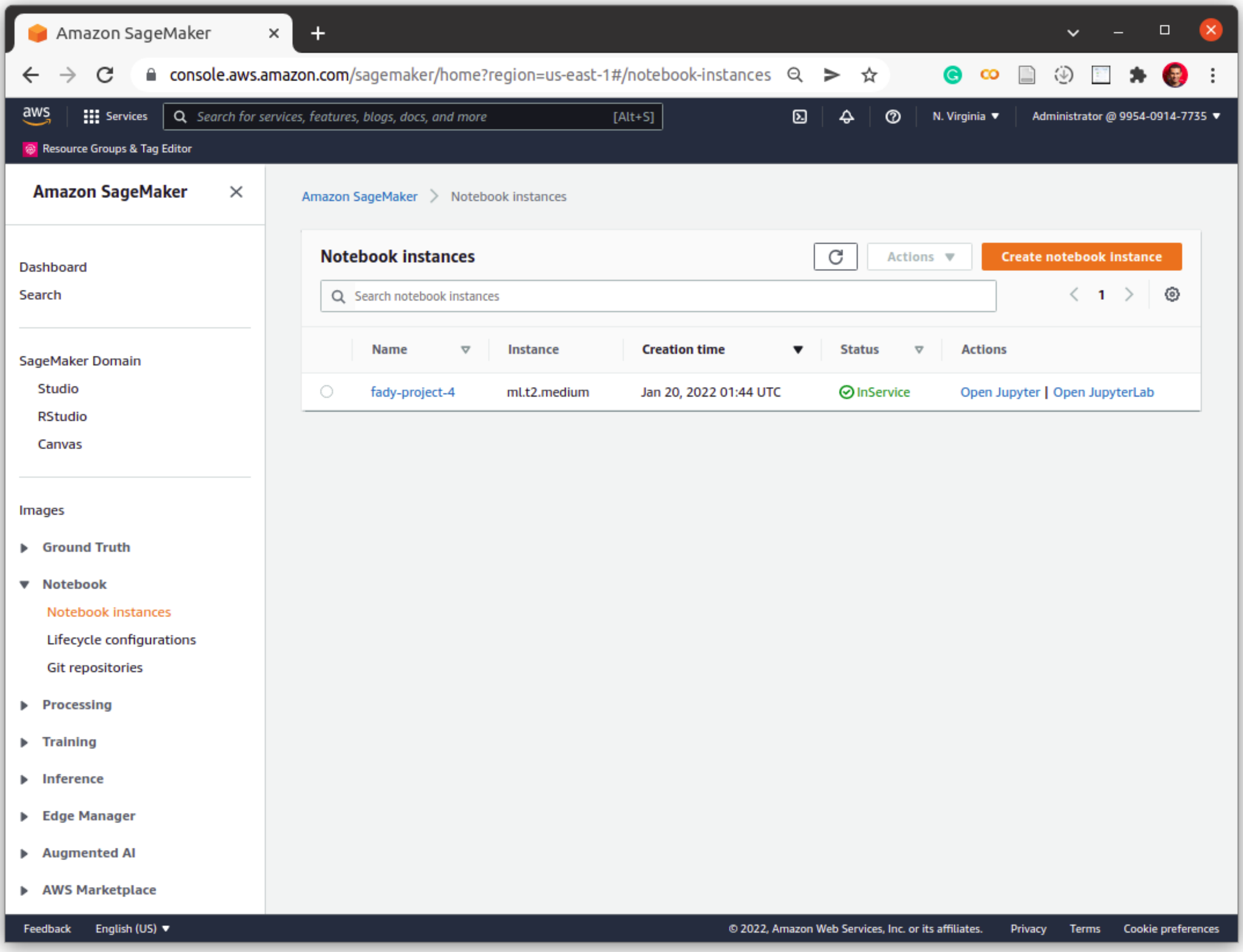
Listing 1: Project Directory Structure

Step 1: Initial setup, training and deployment

Initial Setup

First, we start by creating a sagemaker notebook instance. In this case I cose `ml.t2.medium` instance it is the most economic instance type in sagemaker. and we don't need powerful processing power or large RAM. This instance will be used for just running notebook code and will not be used for model training or inference.

A screenshot of the notebook instance:



Then upload code archive `starter.zip` to the notebook instance to run the experiment.

`starter.zip` archive contents:

```
starter.zip
 ec2train1.py
 hpo.py
 infernce2.py
 lab.jpg
 lamdafunction.py
 train_and_deploy-solution.ipynb
```

To upload and extract the source code files open jupyter notebook in the instance, then open the terminal and type the following commands:

```
cd /home/ec2-user/SageMaker/
wget -c https://video.udacity-data.com/topher/2021/September/613fd77f_starter/starter.zip
unzip starter.zip
```

Download data to an S3 bucket

The provided dataset is the dog breed classification dataset which can be downloaded from this link. It contains images of 133 dog breeds. divided into 6680 training images, 835 validation images, and 836 testing images.

The first three cells of `train_and_deploy-solution.ipynb` download the dog breed dataset to our AWS workspace. The third cell copies the data to the AWS S3 bucket.

I created a bucket and gave it the name `s3://fady-aws-mlnd-dog-images-classification`, then extracted the data into a subdirectory `s3://fady-aws-mlnd-dog-images-classification/data/`. I did minor modifications on `train_and_deploy-solution.ipynb` to point the training script to the extracted dataset.

A screenshot of the created bucket:

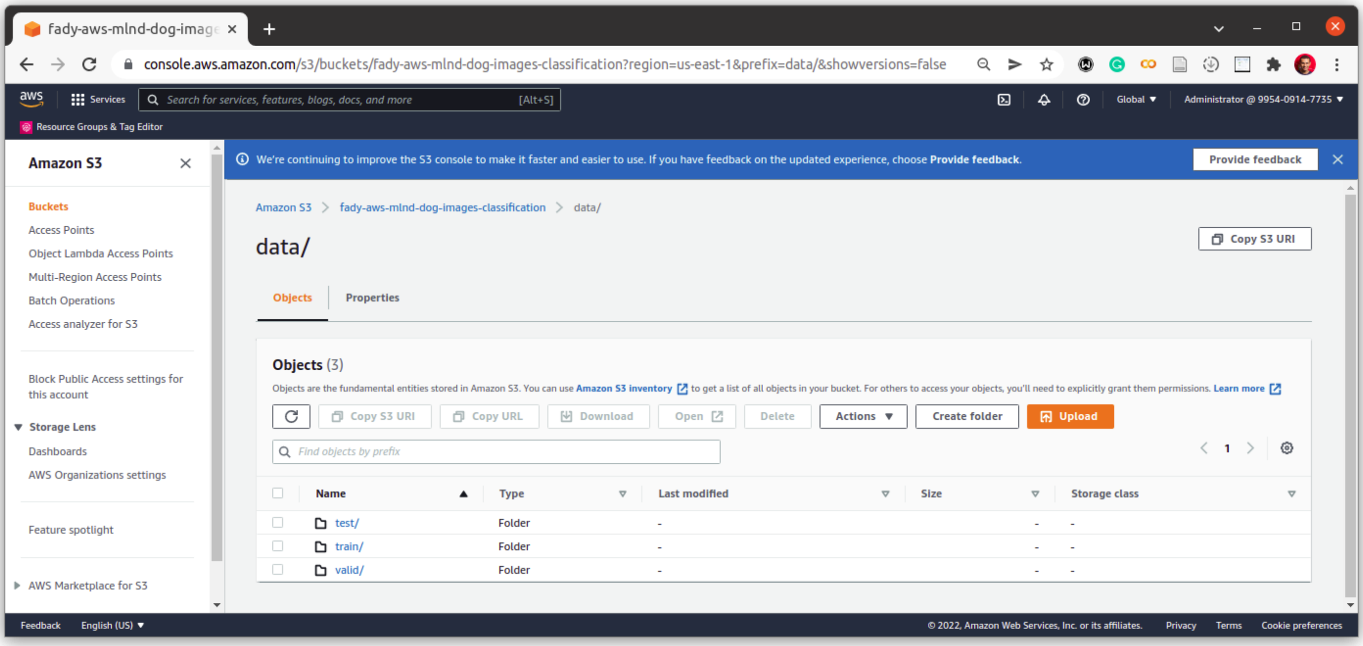


Figure 1: S3 bucket

Training and Deployment (Single Instance Training)

From the fourth to the sixteenth cell of the `train_and_deploy-solution.ipynb` notebook, I created a tuning job with an instance type `ml.m5.xlarge`, `max_jobs=2` and `max_parallel_jobs=1` it took approximately 41 minutes to complete. The best hyperparameters found were `{'batch_size': 32, 'learning_rate': '0.00834462420525608'}`

Then, I performed actual model training on the best hyperparameters found by the tuner. This time I used `ml.m5.2xlarge` instance as it has more processing power.

Then, I ran cells in the **Deployment** section of the notebook to run an endpoint. I chose `ml.t2.medium` as it was sufficent for the current inference task and I can run it for long hours to complete the next steps of the projects and test lambda functions without incurring too much charges.

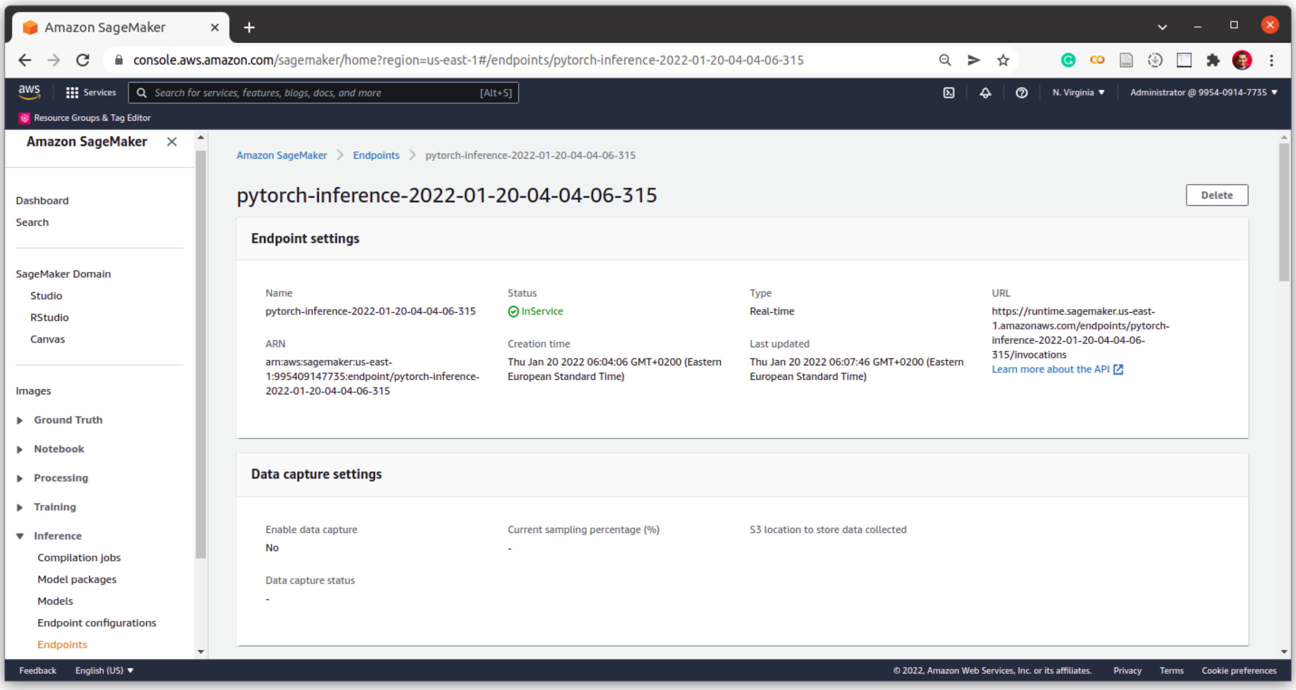
Then, I tested it using the supplied request dict

```
{"url": "https://s3.amazonaws.com/cdn-origin-etr.akc.org/wp-content/uploads/2017/11/20113314/Carolina-Dog-standing-outdoors.jpg"}
```

and I got the following ineference vector:

```
[ 0.13380046,  0.12406865,  0.01930925,  0.04459066,  0.31491399,
  0.14133964, -0.06225839,  0.15845889, -0.15713356, -0.06818745,
  0.12937857,  0.147229   , -0.06525478,  0.12964083,  0.19601114,
  0.07686417,  0.09087689, -0.02894698, -0.01715944,  0.16697152,
  0.1283621 , -0.01064654,  0.10620853,  0.17278288, -0.03866604,
 -0.15799397,  0.14493701, -0.18795769,  0.27061909,  0.05165472,
  0.07323486,  0.11265536, -0.07641914,  0.14794479,  0.01756929,
  0.13642494, -0.03507356,  0.11343399,  0.1752239 ,  0.06675729,
  0.21560508,  0.11155353,  0.01593112,  0.14728004,  0.01010097,
  0.19000889,  0.01708234,  0.07144631, -0.0570593 , -0.04598607,
  0.08687741,  0.00667302, -0.09806156,  0.07812651,  0.01943411,
  0.11979307,  0.12735154, -0.01926402, -0.07421727,  0.08052468,
  0.08618335,  0.05832509,  0.04668785, -0.14725739, -0.10800982,
 -0.22411092, -0.23320678,  0.14338717,  0.03731703, -0.01098941,
  0.16383903, -0.1022775 , -0.10918213, -0.17303845, -0.11920816,
  0.08246608, -0.10248563, -0.12475339,  0.07674296, -0.03876449,
  0.0273919 ,  0.09122247, -0.06179177, -0.01053959, -0.19576041,
  0.0418855 ,  0.15541904,  0.02792337,  0.01690321,  0.06227571,
  0.0740654 , -0.05714193, -0.21430534, -0.12310754, -0.09441458,
 -0.09090099, -0.02984259, -0.01214801, -0.09557887, -0.21917576,
 -0.08656652, -0.29427898,  0.05461352, -0.11696375, -0.25783783,
 -0.00612676, -0.02330022, -0.40888697, -0.08250632, -0.23096839,
 -0.0869923 ,  0.06188012, -0.14508837, -0.21565518,  0.07465033,
 -0.30452806, -0.04589951,  0.03223781, -0.31530797, -0.15733883,
 -0.37445912, -0.14996621, -0.07589076,  0.04984585, -0.25771111,
 -0.28103814, -0.13811049, -0.26339793, -0.03755928, -0.11867087,
 -0.28322217, -0.40221205, -0.3020235 ]
```

The endpint name is `'pytorch-inference-2022-01-20-04-04-06-315'` and is shown in the following screenshot:

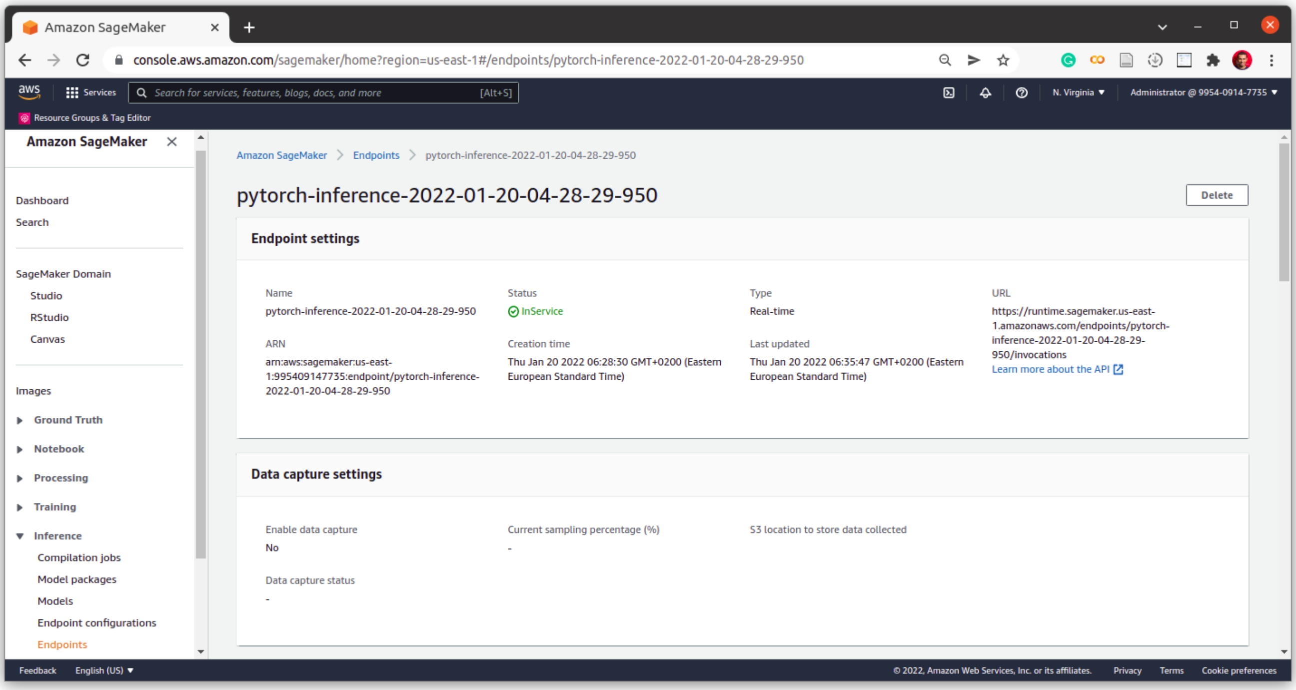


Training and Deployment (Multi-instance training)

I created a multi-instance training job by modifying the parameter `insance_count=4` to run 4 instances simultanously for training.

```
estimator_multi_instance = PyTorch(
  ... ,
  instance_count = 4,
  ...
)
```

Then I deployed another endpoint, the endpint name is `'pytorch-inference-2022-01-20-04-28-29-950'` and is shown in the following screenshot:



Step 2: EC2 Training

I used m5.2xlarge. I ran multiple experiments in Project 03 - image-classification-aws-sagemaker with training 1 epoch of the dataset and I found out that this instance has a decent value per dollar. The results can be shown in the following table:

compute instance	billing time	cost/hour	Epoch training time(sec)	Epoch testing time(sec)	setup time(sec)	training cost/epoch	total cost
ml.m5.large	1980	0.115	1648.37	156.01	175.62	0.053	0.063
ml.m5.xlarge	1164	0.23	894.19	92.23	177.58	0.057	0.074
ml.p2.xlarge	601	1.125	163.52	16.18	421.3	0.051	0.188
ml.m5.4xlarge	538	0.922	336.38	46.23	155.39	0.086	0.138
ml.c4.4xlarge	648	0.955	398.55	52.32	197.13	0.106	0.172
ml.m5.2xlarge	742	0.461	518.4	60.69	162.91	0.066	0.095
ml.g4dn.12xlarge	473	4.89	111.42	9.67	351.91	0.151	0.642
ml.p3.2xlarge	495	3.825	108.59	11.17	375.24	0.115	0.526

As a training image, I used Deep Learning AMI (Amazon Linux 2) Version 57.0 - ami-06ada98f5d02a2d2d to train the model.

The command that is used to create the instance with the deep learning image is:

```
aws ec2 run-instances --image-id ami-06ada98f5d02a2d2d --count 1 --instance-type m5.2xlarge --
  key-name <kms-key-name> --security-groups <security-group-name>
```

Screenshot of the created instance:

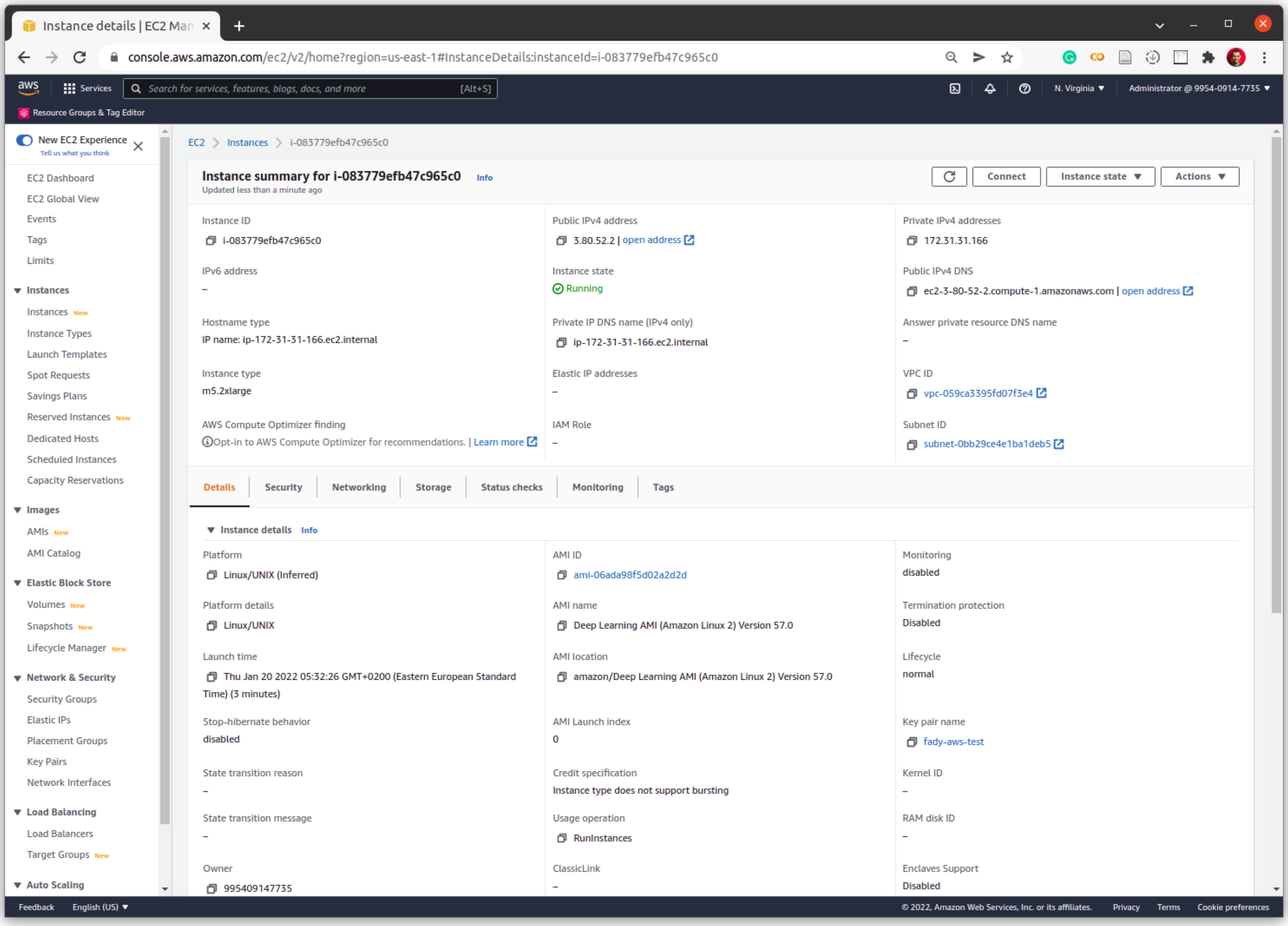


Figure 2: EC2 Instance

Then I used ssh to connect to the instance:

```
ssh -i "fady-aws-test.pem" ec2-user@ec2-3-80-52-2.compute-1.amazonaws.com
```

Download the data and create model output directory:

```
wget https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip
unzip dogImages.zip
mkdir TrainedModels
```

Paste the contents of ec2train1.py inside `solution.py` on the machine

```
vim solution.py
```

Activate the `pytorch` environment that we will use to train our model:

```
conda activate aws_neuron_pytorch_p36
```

Train the model

```
python solution.py
```

Screenshot of final model training step in terminal:

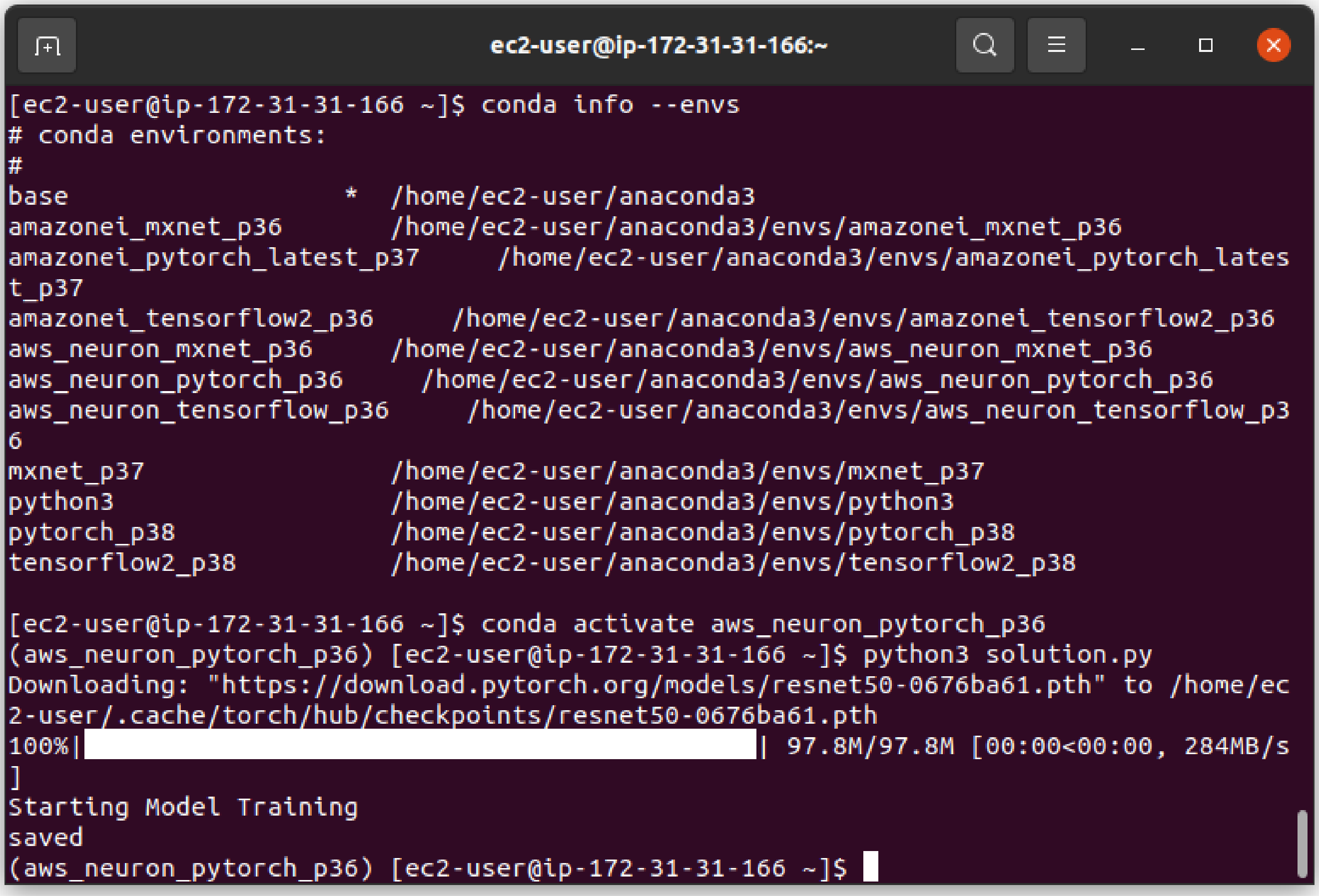


Figure 3: EC2 Terminal

Difference Between EC2 Training Code and the Code used in Sagemaker

For EC2 Training we executed `ec2train1.py` directly inside the instance. While in Sagemaker we executed deployment code from cells in train_and_deploy-solution.ipynb that created a new training job instance and copied the training script `hpo.py` to it to do the training.

Major differences:

- Executing `ec2train1.py` directly from the command line performs the training locally on the same compute machine. While in the Sagemaker notebook train_and_deploy-solution.ipynb it spawns another compute instance and pass all the training parameters to it.
- The hyperparameters in Sagemaker starter script `hpo.py` are passed explicitly to the script and recorded in the instance environment variables.
  `hpo.py` execution command:

```
/opt/conda/bin/python3.6 hpo.py --batch_size 32 --learning_rate 0.00834462420525608
```

  `hpo.py` argument parsing:

```
parser.add_argument('--learning_rate', type=float)
parser.add_argument('--batch_size', type=int)
parser.add_argument('--data', type=str, default=os.environ['SM_CHANNEL_TRAINING'])
parser.add_argument('--model_dir', type=str, default=os.environ['SM_MODEL_DIR'])
parser.add_argument('--output_dir', type=str, default=os.environ['SM_OUTPUT_DATA_DIR'])
```

  Instance environment variables:

```
SM_USER_ARGS=["--batch_size","32","--learning_rate","0.00834462420525608"]
SM_HPS={"batch_size":32,"learning_rate":"0.00834462420525608"}
SM_HP_BATCH_SIZE=32
SM_HP_LEARNING_RATE=0.00834462420525608
SM_CHANNEL_TRAINING=/opt/ml/input/data/training
SM_MODEL_DIR=/opt/ml/model
SM_OUTPUT_DIR=/opt/ml/output
SM_USER_ENTRY_POINT=hpo.py
```

  In `ec2train1.py`, the hyperparameters are included in the script.

```
batch_size=2
learning_rate=1e-4
```

  `ec2train1.py` execution command:

```
python ec2train1.py
```

- In Sagemaker, the output trained model `model.pth` is saved to the training job compute instance `SM_MODEL_DIR=/opt/ml/model` , then it is compressed with the source code `hpo.py` and the model artifact is transferred automatically to an output directory in the S3 bucket. While in EC2 training the model is saved locally inside `./TrainedModels` and the user is responsible of saving the model to S3 using `aws s3 cp` command.
- Sagemaker can use the training script to spawn multiple instances and perform distributed training. While EC2 instance is just limited to the one instance that the script is run on.

Step 3: Setting up a Lambda function

The supplied lambda function `lamdafunction.py` was patched with the deployed endpoint that we created in previous steps `endpoint_Name='pytorch-inference-2022-01-20-04-28-29-950'`

The function only accepts image URL passed as a request dictionary with content type `application/json`. The request dictionary has the format `{'url':'http://website.com/image-url.ext'}`

It invokes the `pytorch-inference-2022-01-20-04-28-29-950` endpoint, passing the request dictionary (`{'url': 'http://....'}`) in the body and setting the content type to `application/json`. The endpoint returns the predictions to the lambda function and the lambda function returns the predictions to the user in the body of the response, with a status code 200.

Step 4: Lambda Security and Testing

In this step we created an IAM execution role `arn:aws:iam::995409147735:role/fady-project-4-lambda-execution-role` for our lambda function and attached `AmazonSageMakerFullAccess` security policy to it.

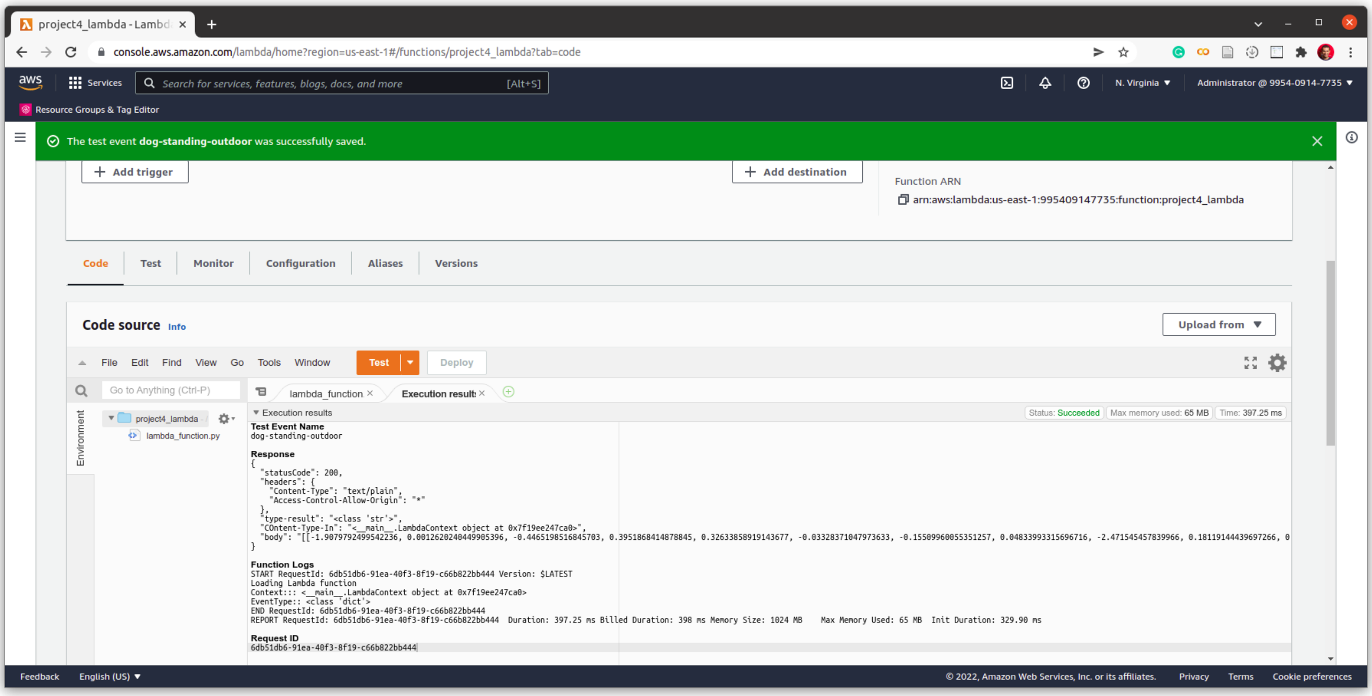
Lambda function testing

I tested the lambda function on the following dog image:

using the supplied request dict

`{"url": "https://s3.amazonaws.com/cdn-origin-etr.akc.org/wp-content/uploads/2017/11/20113314/Carolina-Dog-standing-outdoors.jpg"}`

the result is shown in the following screenshot:



Security considerations

The `AmazonSageMakerFullAccess` policy may be too much for our lambda function that only executes endpoints from sagemaker. Perhaps restricting it to endpoints only would be a better practice.

Also care should be taken to delete unused lambdas and roles and give the least priviliges to resources in use to pervent vulnerabilities.

Screenshot of the IAM role used to execute the lambda function:

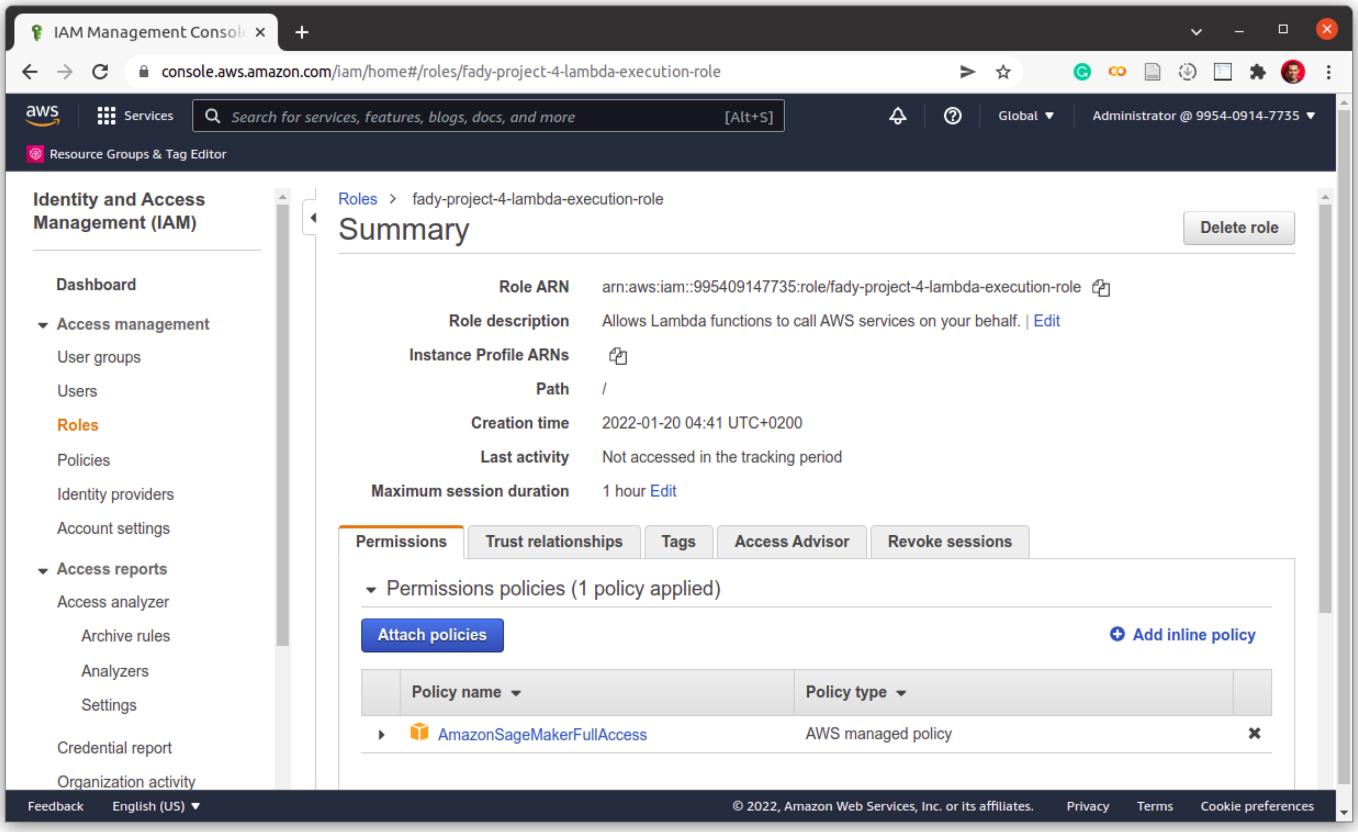


Figure 4: IAM role

Step 5: Concurrency and auto-scaling

Concurrency

oncurrency refers to the ability of Lambda functions to service multiple requests at once

We can use either reserved or provisioned concurrency for our function. Provisioned concurrency is more responsive, but leads to higher costs.

Since we don't expect very high volumes on these functions, it's not necessary to choose very high concurrency. I set the provisioned concurrency to 3 and it is enough for our load, and 100 for reserved concurrency.

Screenshot of lambda concurrency settings:

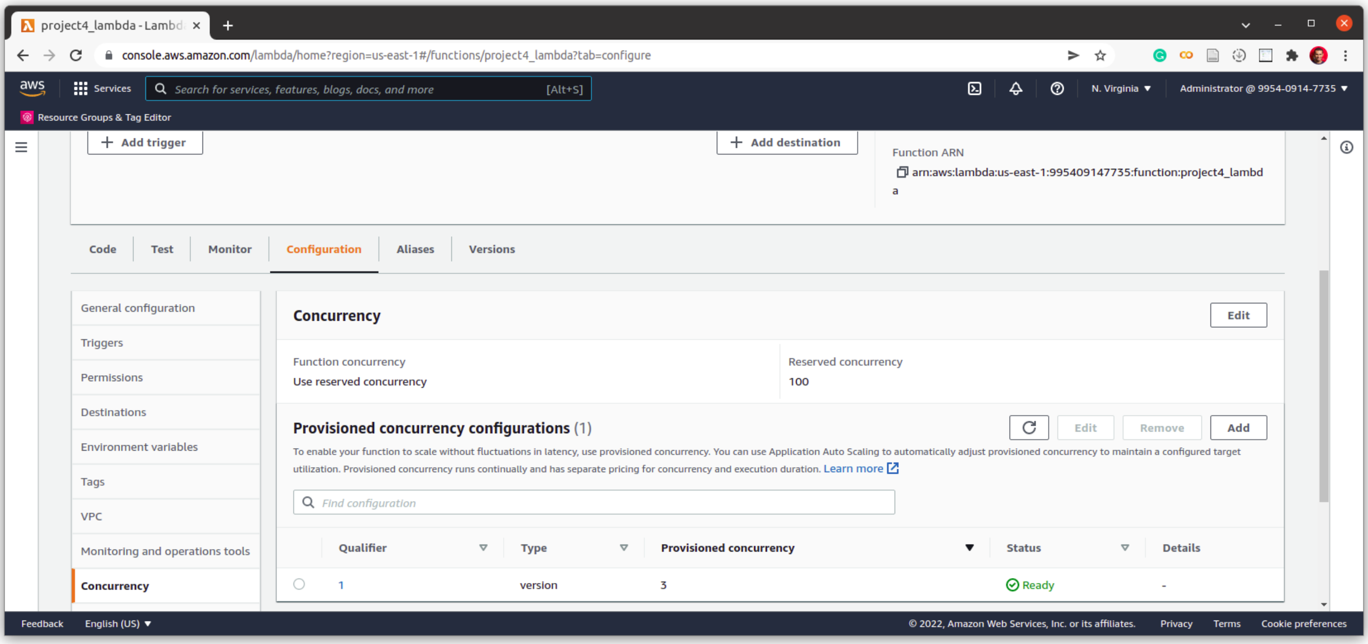


Figure 5: Lambda function concurrency settings

Auto-scaling

Auto-scaling refers to the ability of endpoints to service multiple lambda function requests at once. I chose to autoscale endpoints to 4 instances maximum, with scale in coold down time of 30 seconds and scale out cool down time of 300 seconds. These settings are sufficient for our project needs and workload.