Operationalizing an AWS ML Project

Kamar Abou Muharam

1. Initial Setup

First, we start by creating a sagemaker notebook instance. In this case I chose 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.

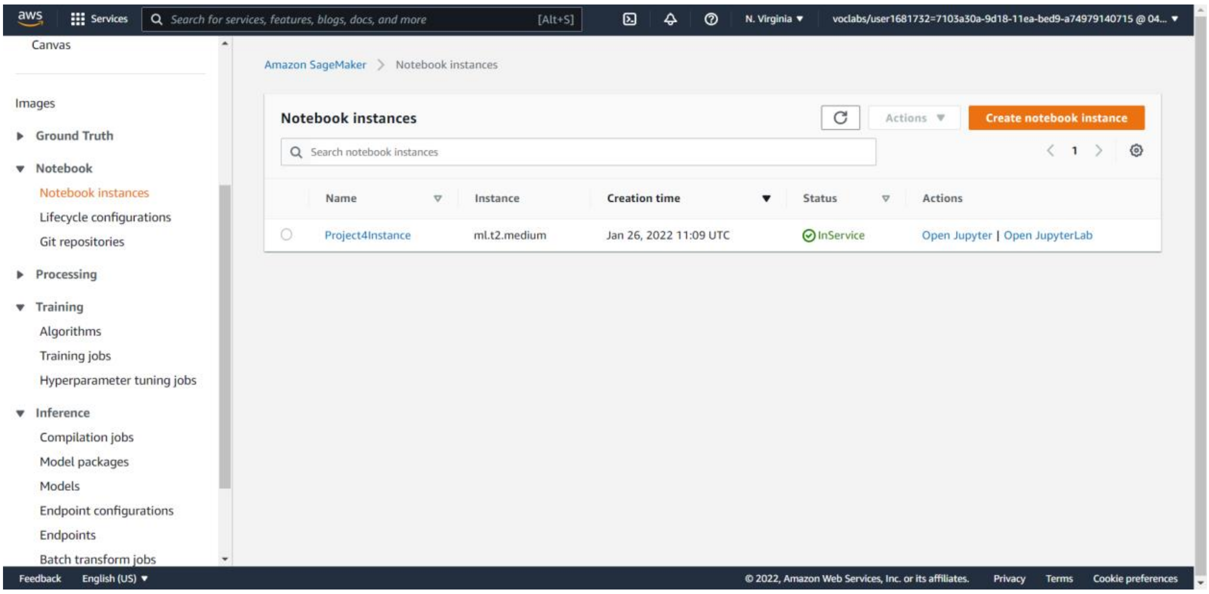


Figure 1 - Sagemaker Notebook Instance

Upload data to an S3 bucket

The dog breed dataset was uploaded to a newly created S3 bucket, successfully.
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:// 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://kamarproject4, then extracted the data into a subdirectory s3://kamarproject4 /data/. I did minor modifications on train_and_deploy-solution.ipynb to point the training script to the extracted dataset., then extracted the data into a subdirectory s3://kamarproject4 /data/. I did minor modifications on train_and_deploy-solution.ipynb to point the training script to the extracted dataset.

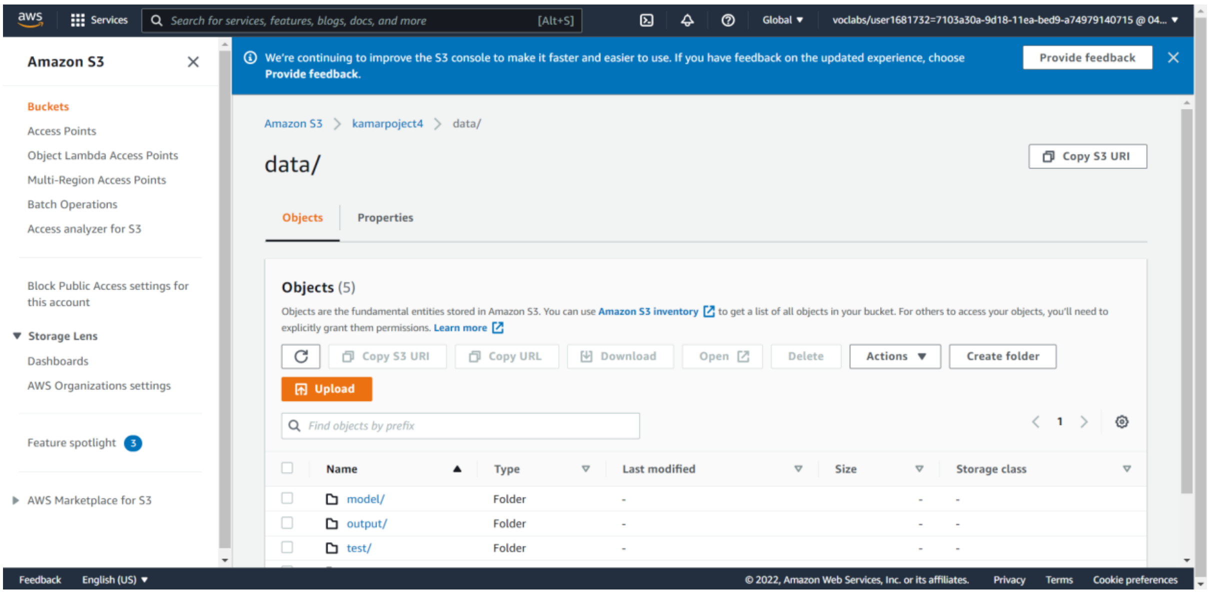


Figure 2 S3 Bucket snapshot

2. Sagemaker Training and Deployment

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" }
```

The endpint name is pytorch-inference-2022-01-27-12-34-34-327' and is shown in the following screenshot:

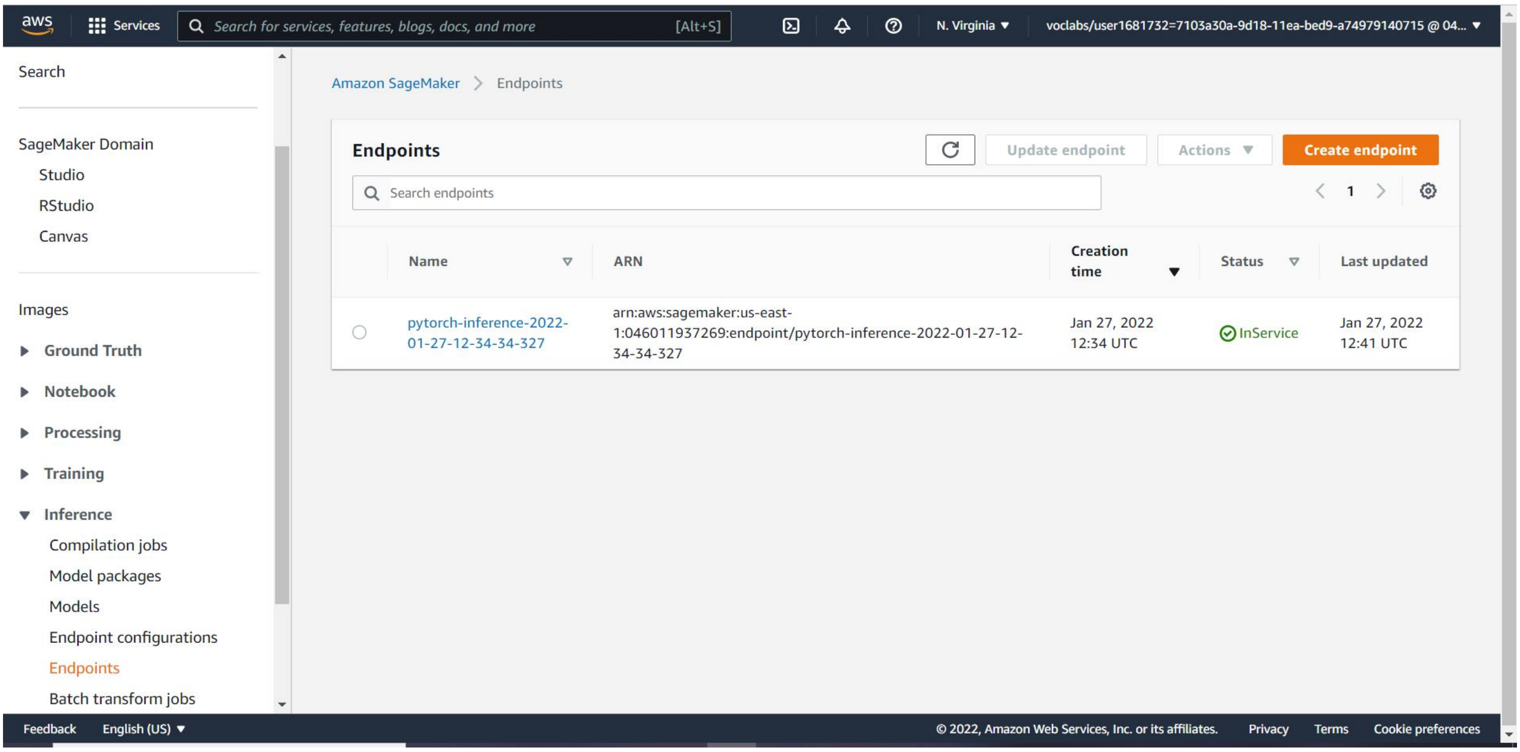


Figure 3 EndPoint

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.

However, upon training the model with the best parameters from above tuning, the model gave a 0 test accuracy! So increased the max_jobs = 6, max_parallel_jobs = 3 and also changed its instance_type = “ml.m5.xlarge” to speed up the computations a bit.
Reran the hyperparmeter jobs and it executed successfully:

3. EC2 Training

- We have utilized the t2.xlarge instance and the Deep Learning AMI (Amazon Linux 2) Version 55.0. This seems like a reasonable balance of performance and affordability.
- As per the documentation, T2 instances can sustain high CPU performance for as long as a workload needs it.
- For most general-purpose workloads, T2 instances will provide ample performance without any additional charges.

- Similarly, because we don't know the duration for which we might need to keep this EC2 instance running for training, it's better to go with a medium size instance so we don't have to pay for a large instance while we're doing setup, debugging and other tasks.

Difference between ec2train1.py (EC2 script) and train_and_deploysolution.ipynb + hpo.py (SageMaker scripts)

- There is no logic for calling any Estimator or Tuner functions in the EC2 script. The code in the EC2 script is responsible for saving the model to the local path. While in the sagemaker scripts this was handled internally by sagemaker where the model data was stored to a S3 location.
- In the EC2 training script, all the variables like hyperparameters and output locations, etc are already mentioned in the script itself and so there is no need for **argparse.** Meaning while running the EC2 script we do not need to mention any arguments.
- In the EC2 script the training happens on the same server on which the script is invoked/executed, however in the sagemaker scripts the training job that is invoked, it runs on a separate container than the one on which the sagemaker notebook is running.
- Another difference is that ec2train1.py lacks the main function
- For the EC2 Training, given that the training data and model, all are stored on the EC2 instance host itself it would be difficult to deploy the saved model to an endpoint in sagemaker. If we wish to do that then we might need to manually upload the model first to sagemaker and then use that to deploy an endpoint. This is not the case in models trained via the sagemaker notebook instances, as the model can be easily deployed to an endpoint.

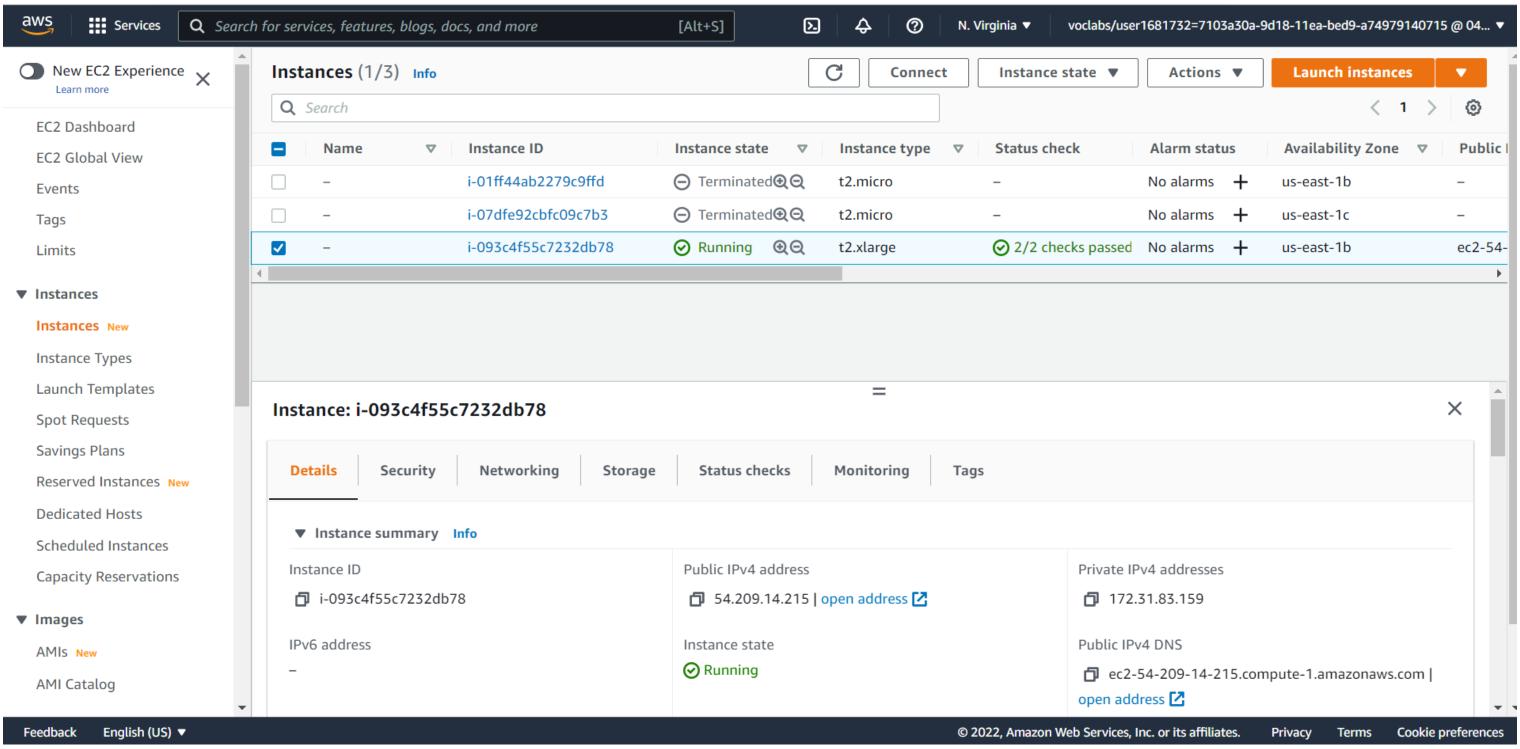


Figure 4 EC2 Instance snapshot

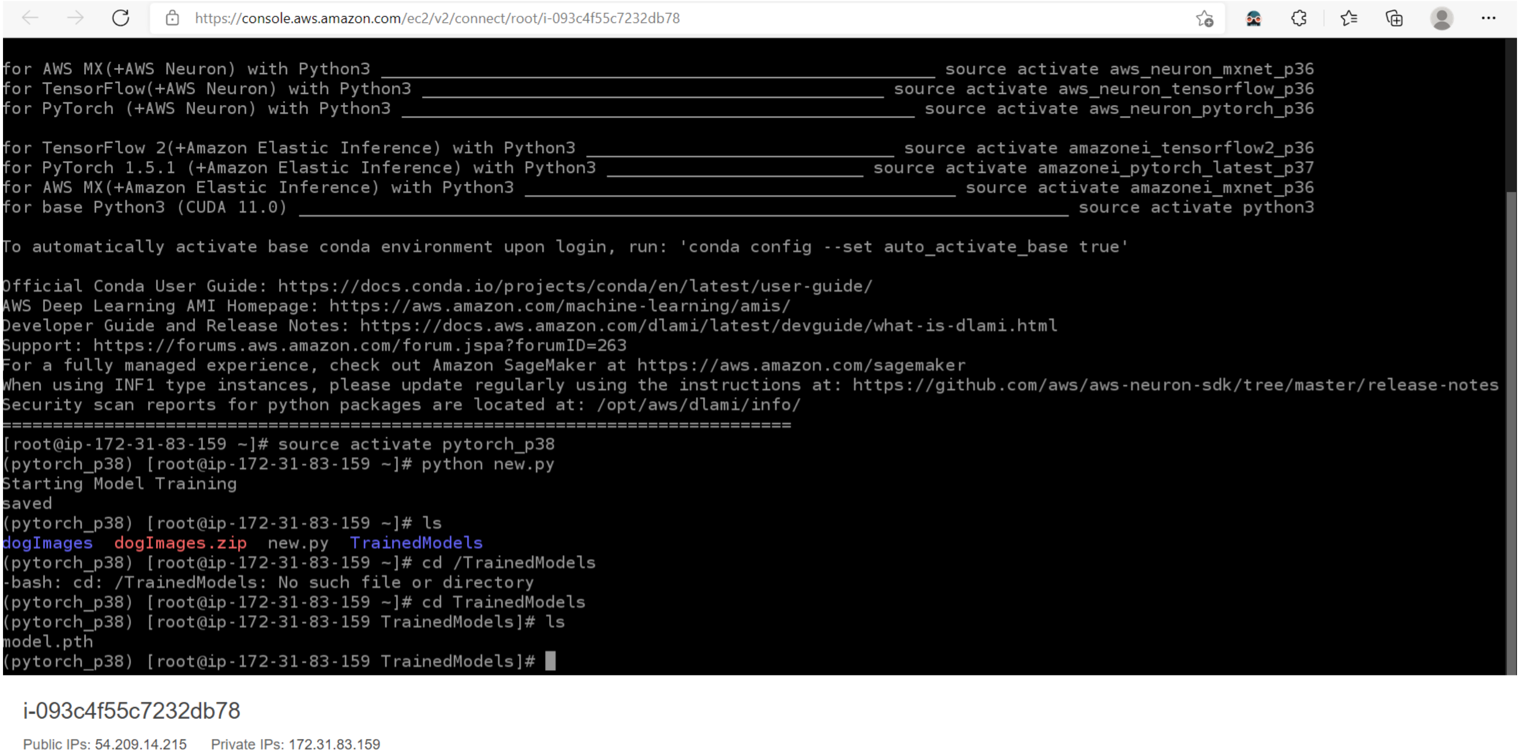


Figure 5 EC2 Training saved model.pth

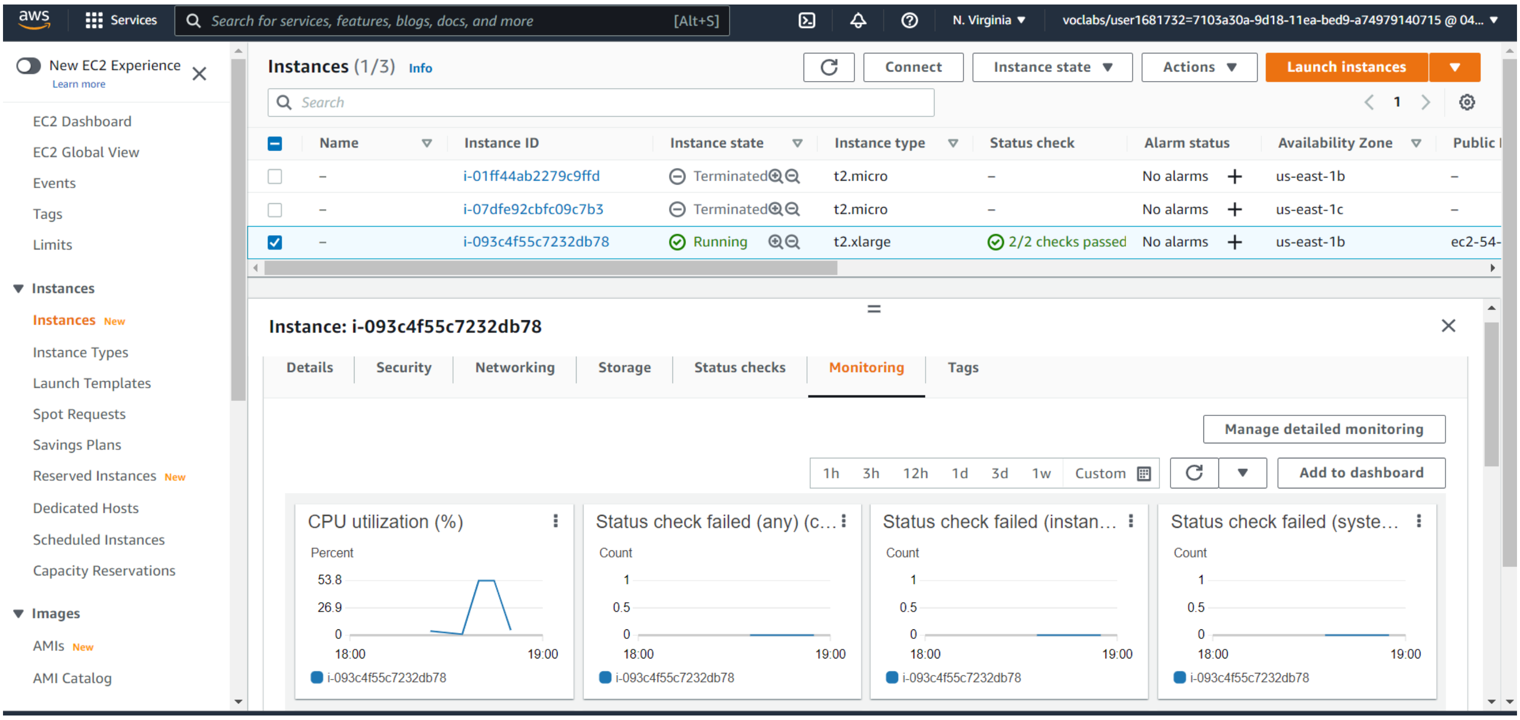


Figure 6 EC2 Instance Resource Metrics

4. Lambda functions

- The lambda functions will be used for invoking our deployed endpoints.
- The lambda function implemented in this project expects the image inputs in json format, which is used to invoke the model's deployed endpoint
- Given we have two endpoints deployed, one for the single instance training and the other for the multi-instance training, we will only use the multi-instance training jobs endpoint and create a lambda function for invoking that endpoint.
- Multi instance trained endpoint that we will be using: "pytorch-inference-2022-01-27-12-34-34-327"

- We created the lambda function with the corresponding changes to invoke the endpoint:

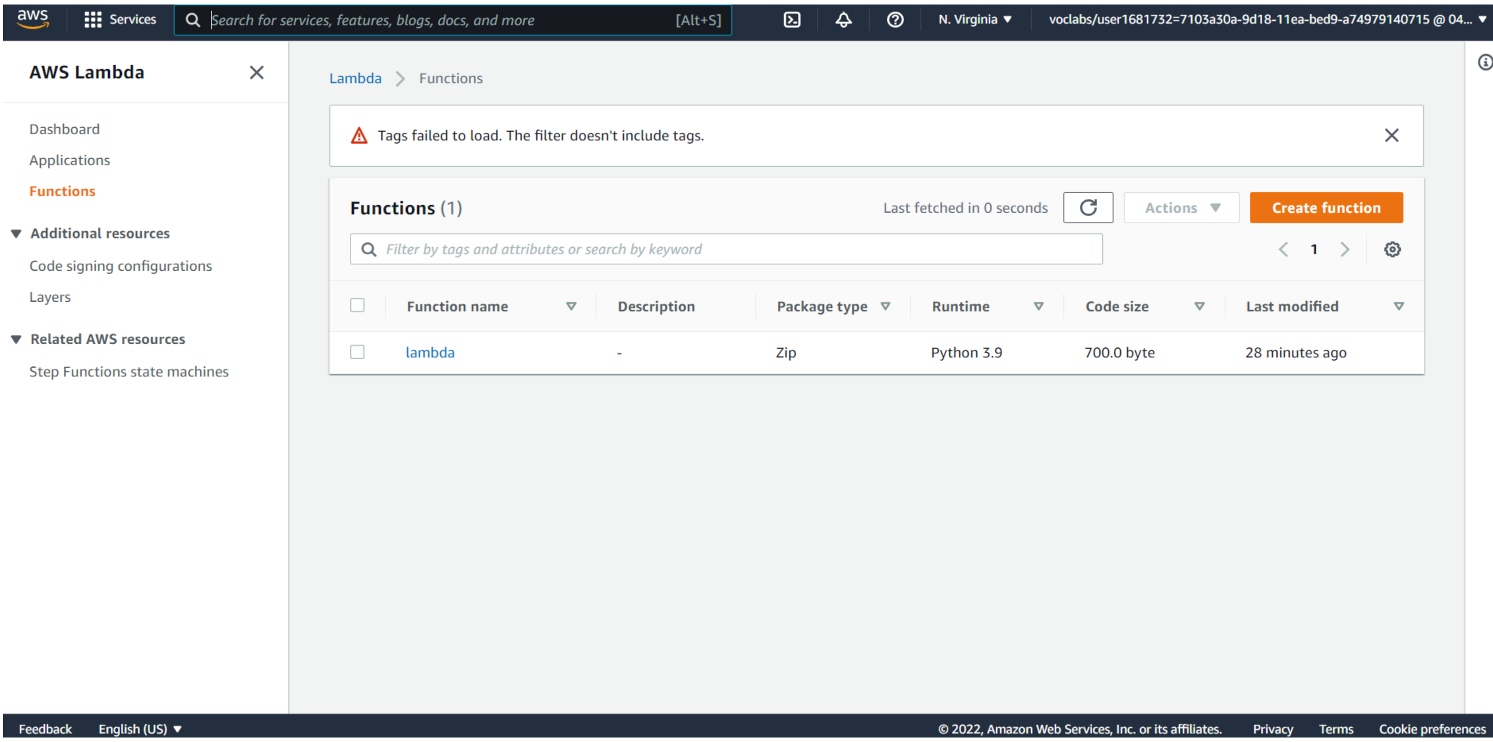


Figure 7 Lambda Function

5. Lambda Security and Testing

In this step we created an IAM execution role arn:aws:iam::046011937269:role/service-role/lambda-role-3yk1kea2 for our lambda function and attached AmazonSageMakerFullAccess security policy to it.

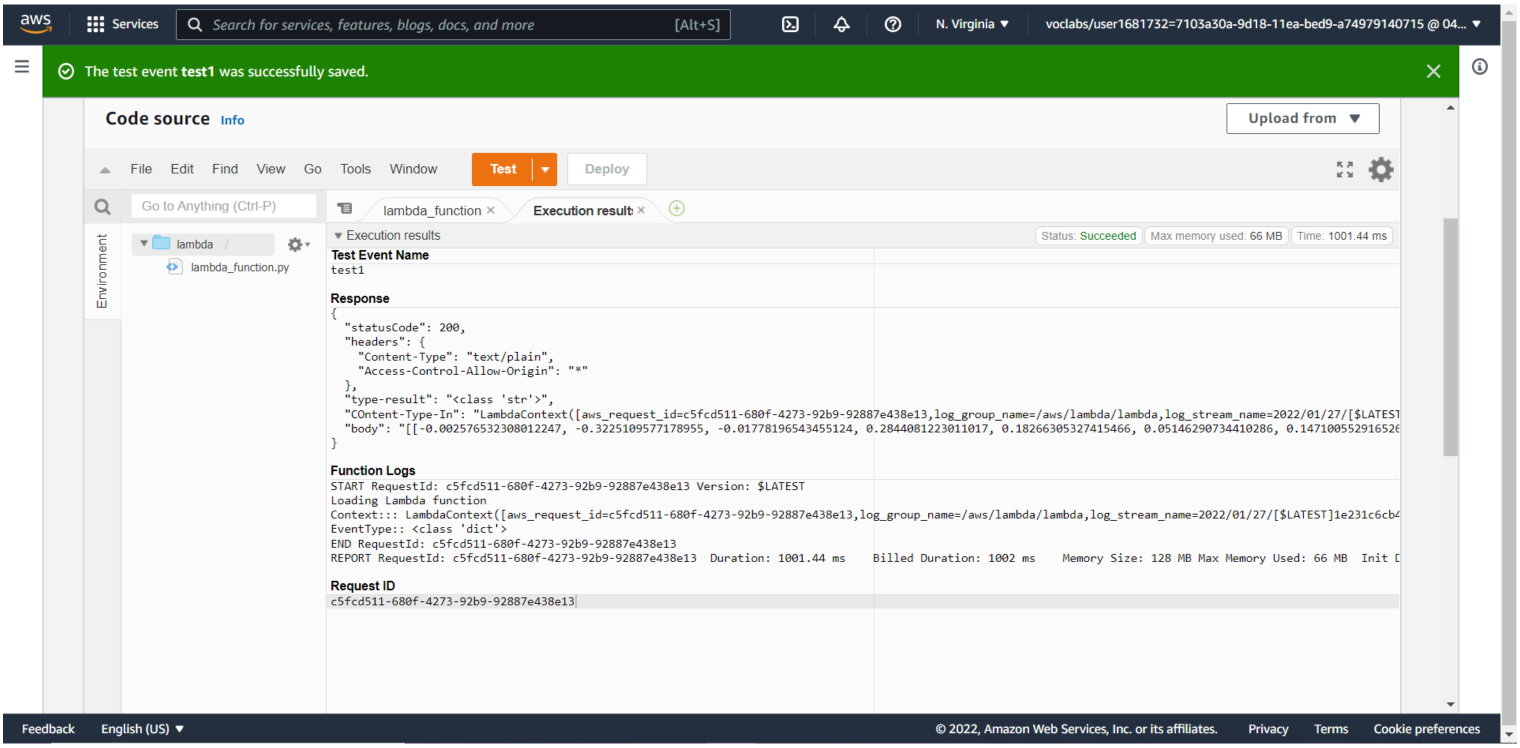
Lambda function testing

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" }

Figure 8 Lambda test



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.

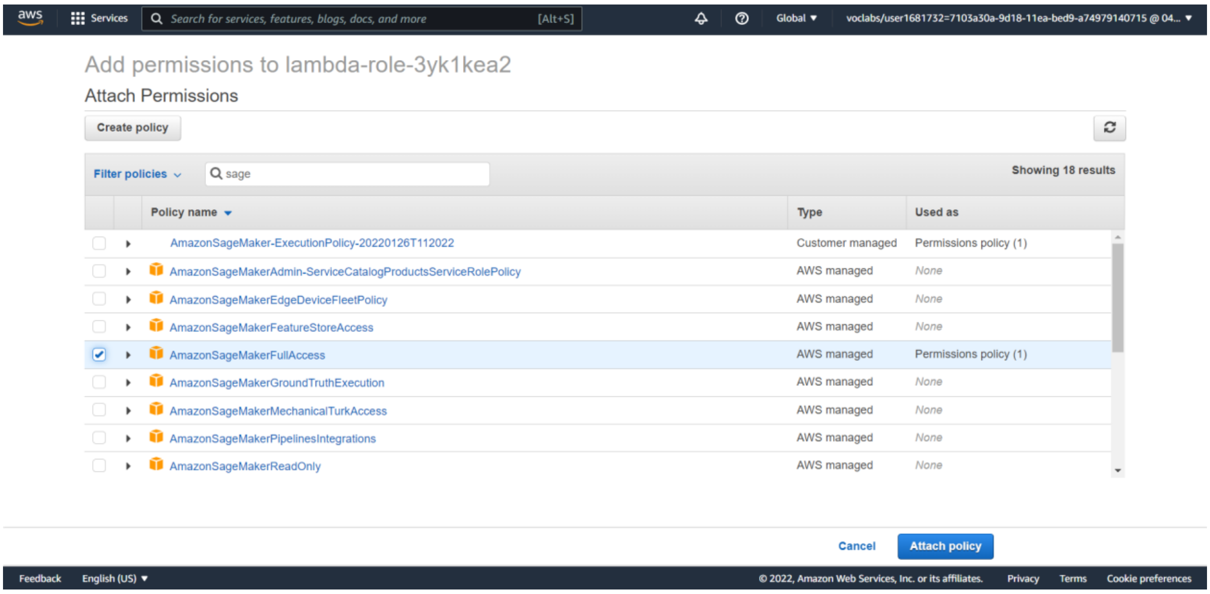


Figure 9 Lambda permission

I'm concerned about the “Full Access” type permission policies that are available.

• For example, for this lambda functions we have provided the lambda function a Full Access to Sagemaker resources, but this does not seem to follow the concept of least privilege access.
• Ideally, one should only allow these lambda functions to query the endpoints that they're intended and allowed to query.
• We will have to do some more analysis to figure out if there's anything we could do about it.
• Furthermore, another concern is that the account's root user does not employ MFA
• Looking at the IAM roles that are currently active, all the roles seems to be necessary and also most of the roles have been added on a per need basis.
• However, we need to keep an eye on the roles dashboard to make sure only relevant roles absolutely necessary for currently active projects , are the only roles that are in active state to prevent unauthorized accesses.

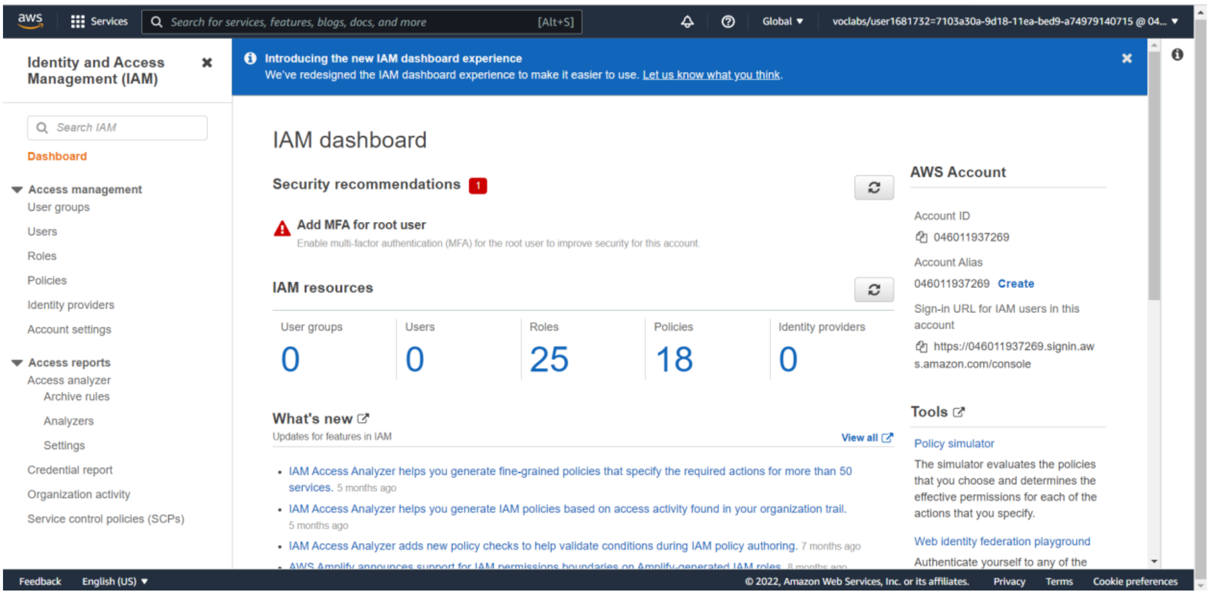


Figure 10: IAM dashboard

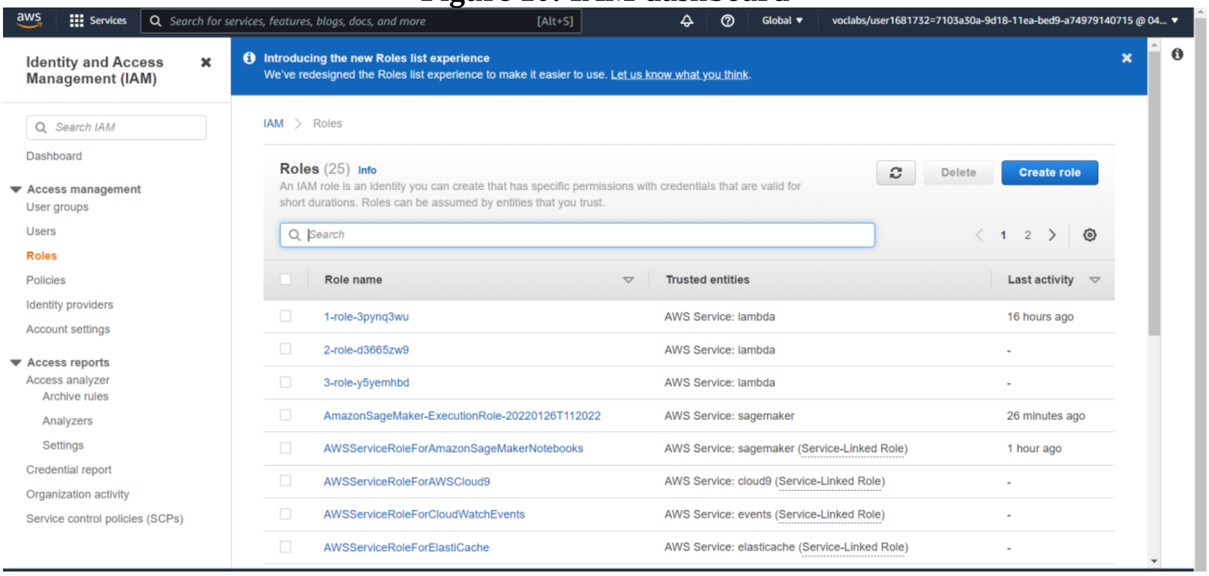


Figure 11: IAM Roles

6. 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:

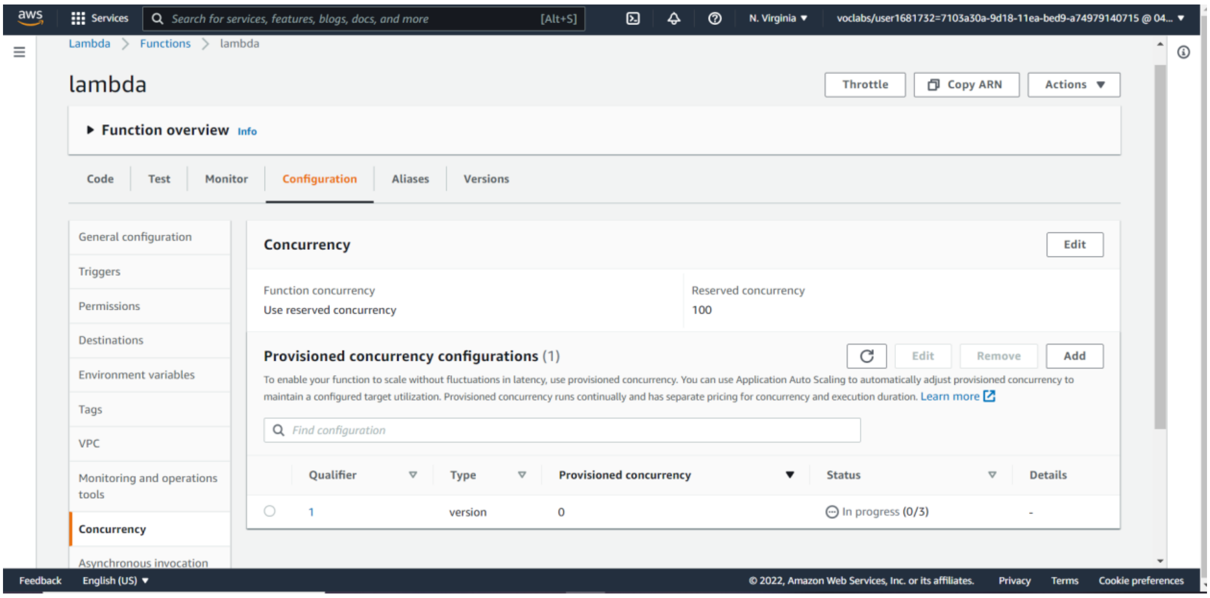


Figure 12: Lambda function concurrency settings

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.