Operationalizing Machine Learning on SageMaker

-Mansi Chilwant

# Initial Setup

I opted ml.t2.medium (Figure: Sage Maker Instance) as the smallest SageMaker instance accessible for my notebook because I'll be working on the project for more hours and don't require a particularly strong instance in terms of CPU or RAM.

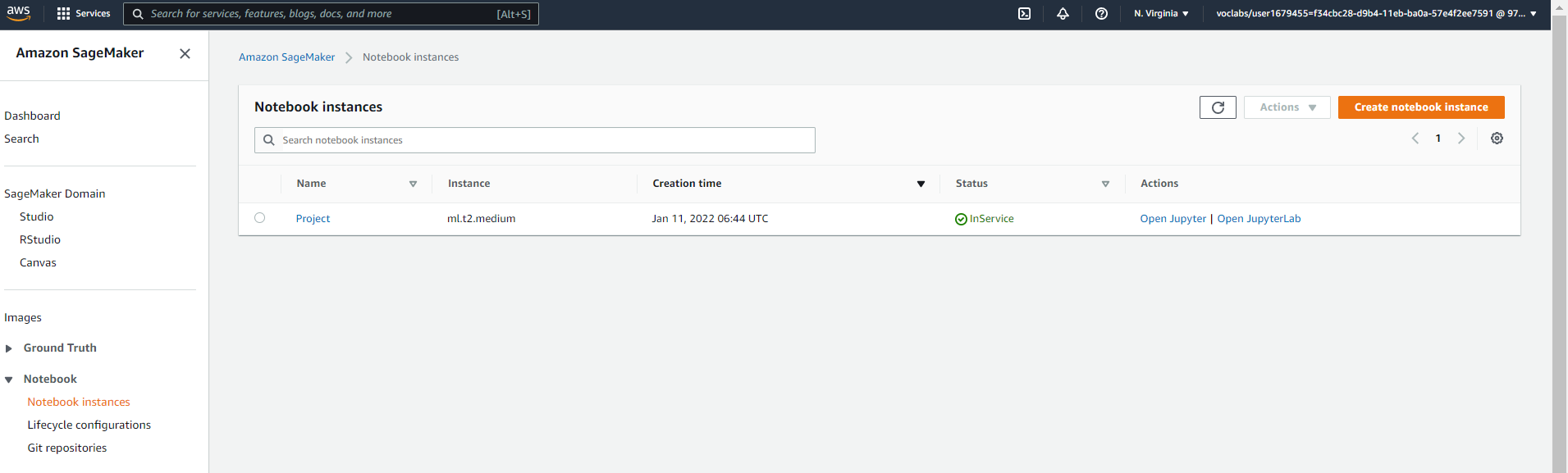


Figure 1: SageMaker Instance

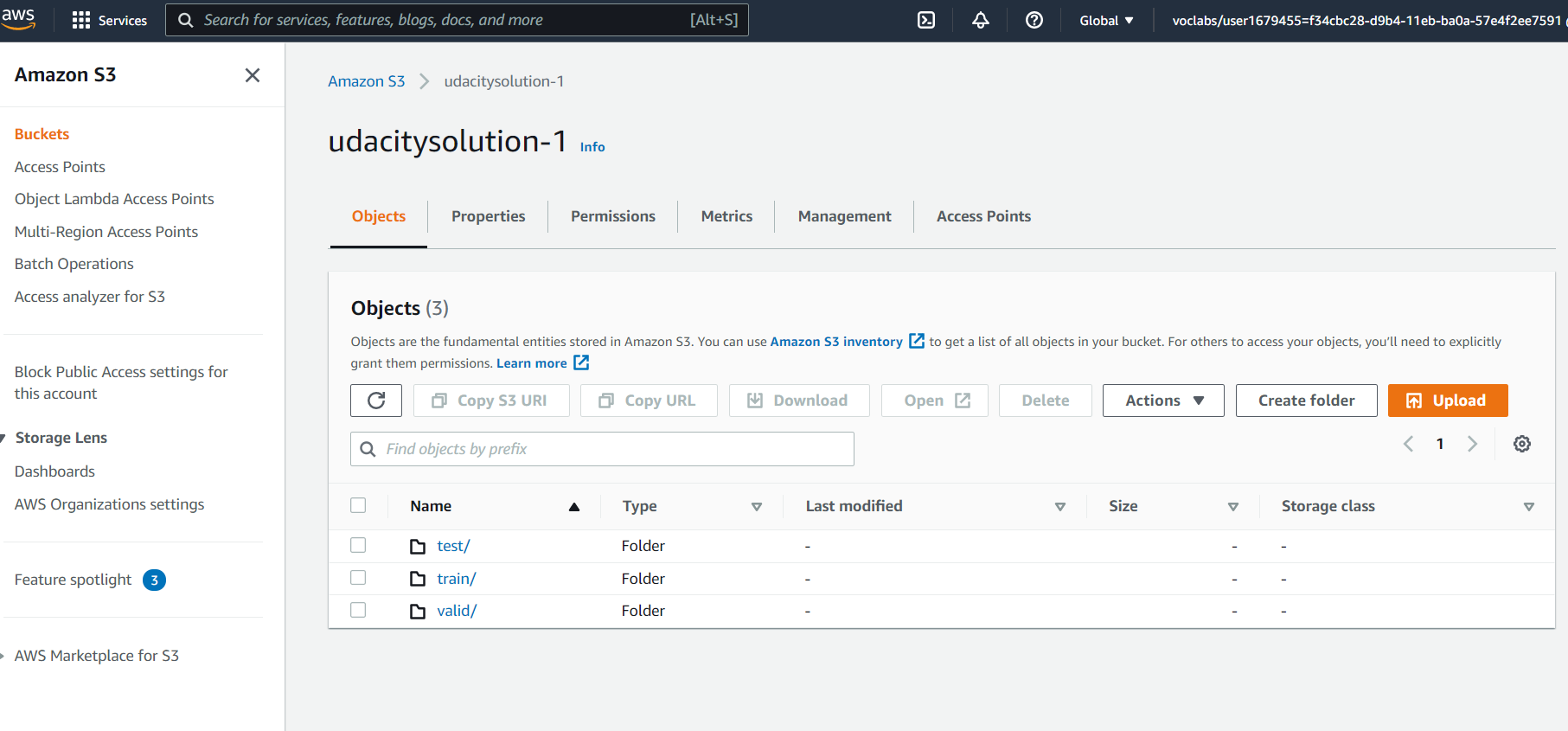


Figure 2: S3 Bucket Setup

Furthermore, I chose ml.m5.2xlarge for both tuning and training since it has more processing power, allowing the tuning and training operations to be completed faster and avoiding memory issues that I had previously seen with this dataset.

I increased the number of tuning tasks to 12, the number of parallel jobs to 3, and the early stopping type to "Auto" to speed up tuning and guarantee that better hyperparameters were picked.

To allow dispersed training, I made the following changes to the hpo.py file, including changes to the parameters necessary to start the training.

Single instance jobs: 2 pytorch trainings

Multi instance jobs : 2 dog-pytorch trainings

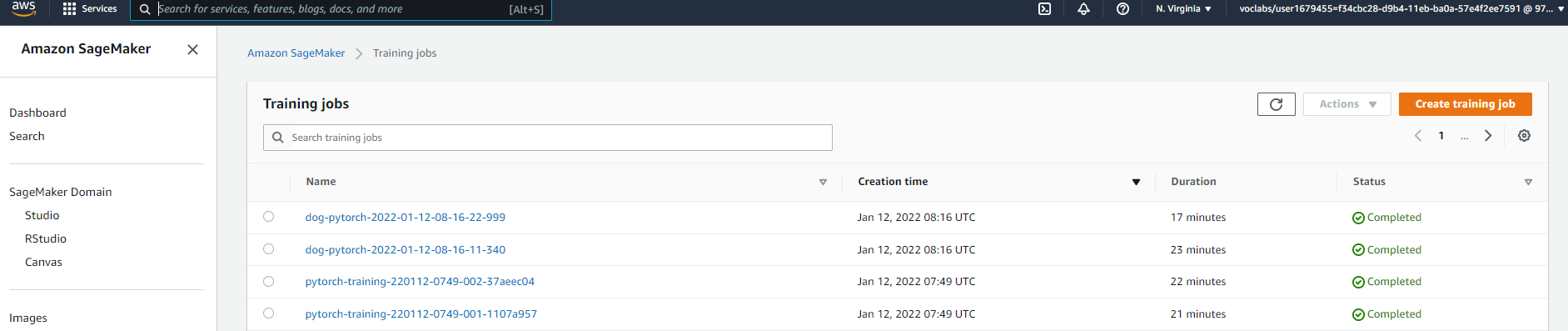


Figure 3: Training jobs both single and multi

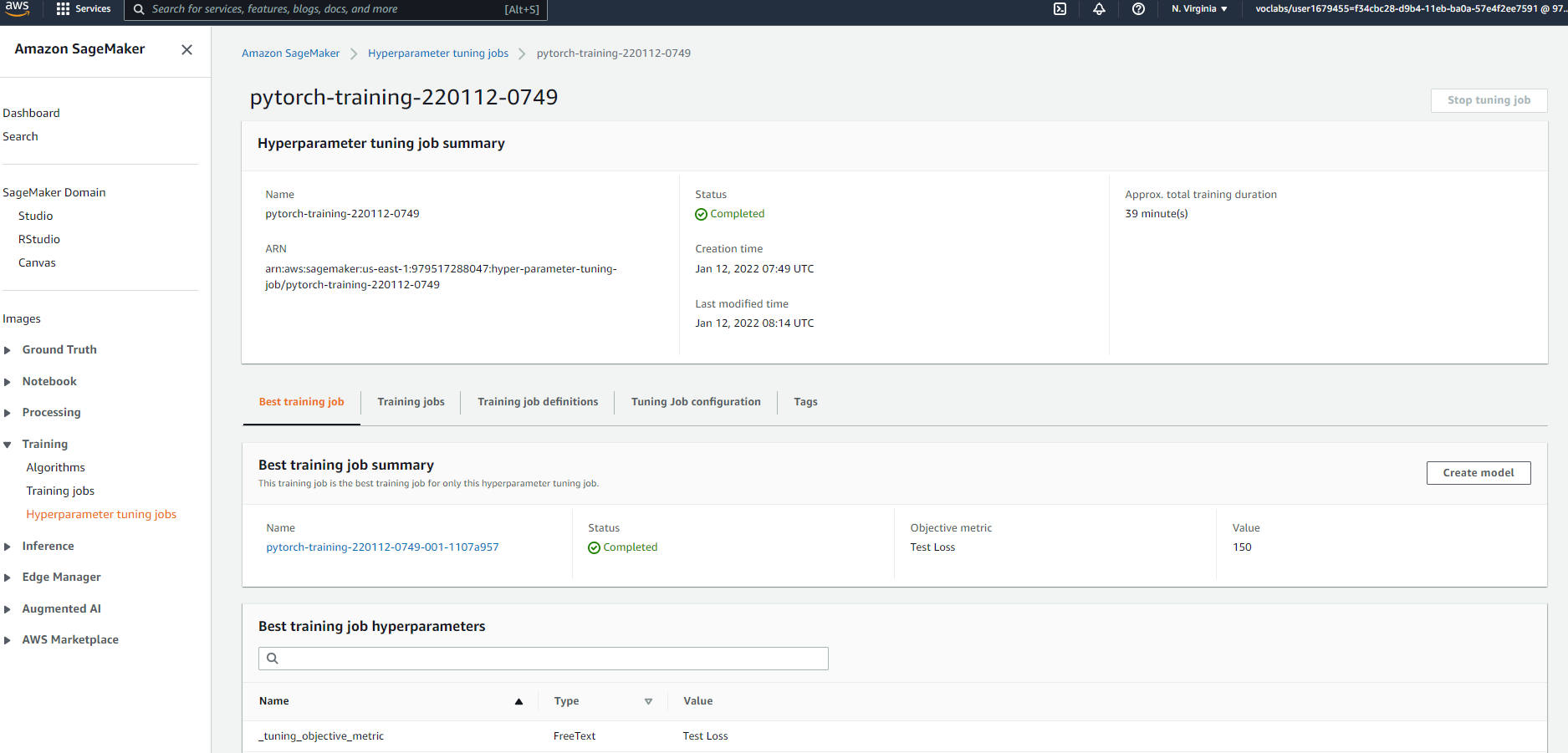


Figure 3: Best training job (single instance)

# EC2 Training

To save money, I used the t2.x micro instance and the Deep Learning AMI (Amazon Linux 2) Version 57.0. This struck me as a good compromise between performance and price. Because two end points are active while executing EC2 training, it's only natural to employ a more powerful instance type than is necessary.

Similarly, because we don't know how long it will take to set up and debug this EC2 training component, we should pick a smaller instance so we don't have to pay for a huge instance while we're performing setup, debugging, and so on.

As you can see in my screenshot, I've restricted access to my EC2 instance to only my IP address in order to avoid the possibility of a security breach.

First the spot instance is created

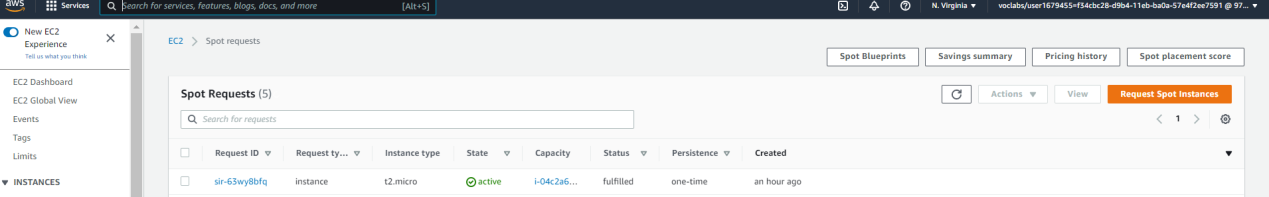


Figure 4: EC2 spot instance

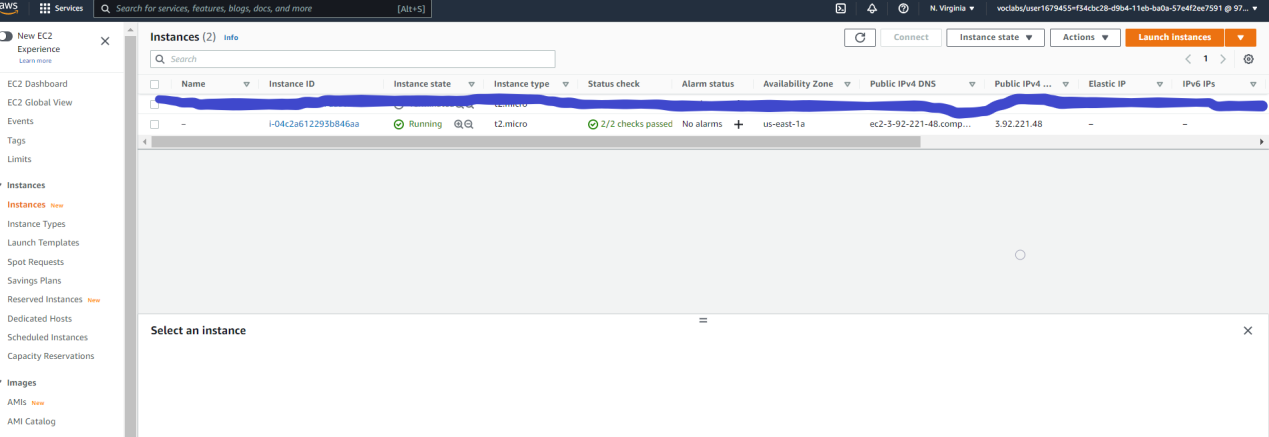


Figure 5: EC2 Instance

## Difference between ec2train1.py (EC2 script) and train\_and\_deploy

**solution.ipynb+hpo.py (SageMaker scripts)**

The biggest difference is that ec2train1.py doesn't have a main function, nor does it have the capabilities to handle argument parsing or optional main running. Similarly, ec2train1.py does not support multi-instance learning, like I implemented in my modified hpo.py. The most notable difference is that the ec2train1.py script uses test data to train, which means it uses a much smaller dataset and the same data for both training and testing.

This has been modified.

train\_data =

torchvision.datasets.ImageFolder(root=test\_data\_path,

transform=train\_transform)

To

train\_data =

torchvision.datasets.ImageFolder(root=train\_data\_path,

transform=train\_transform)



Figure 6: EC2 Training

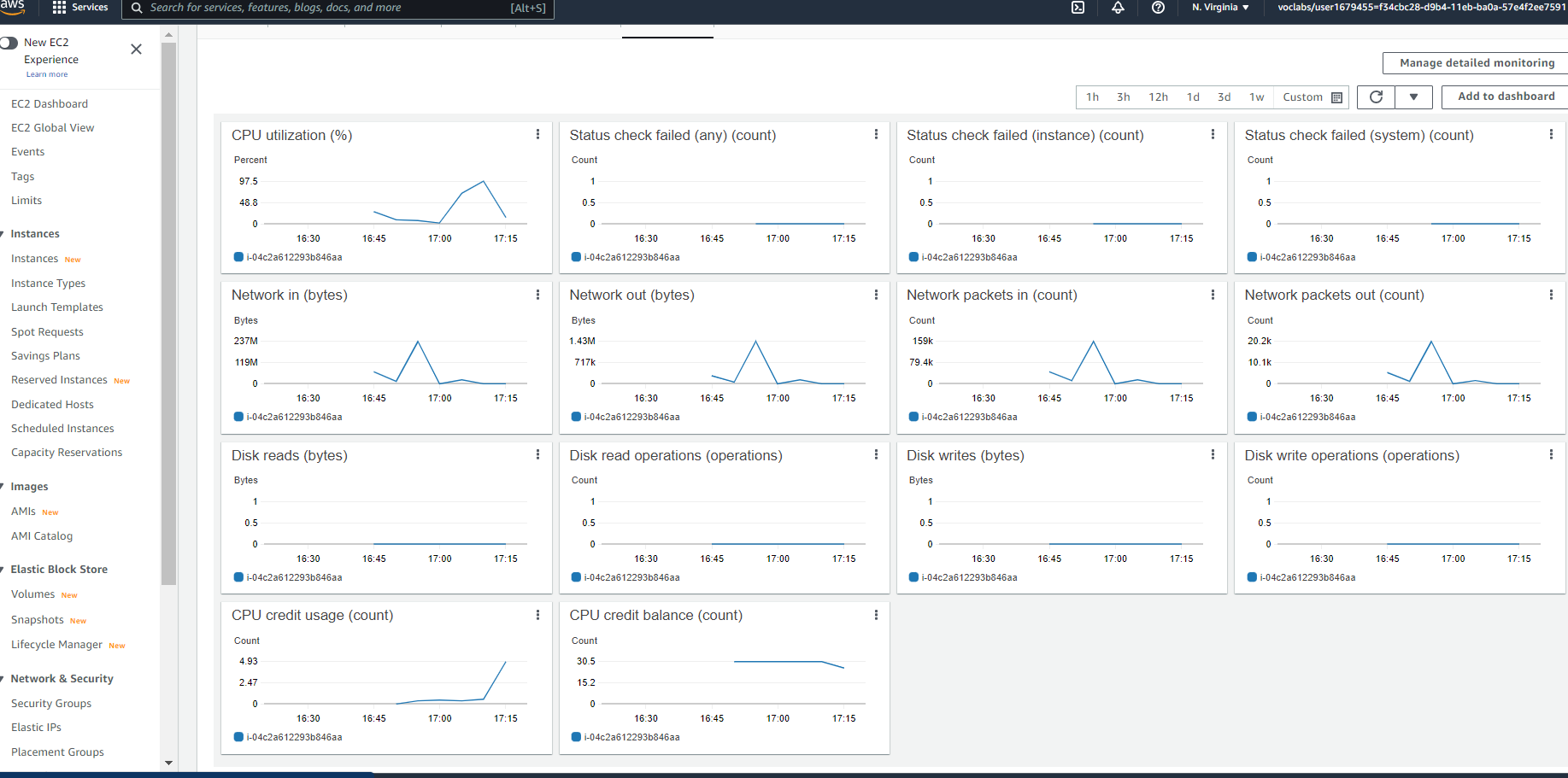


Figure 7: EC2 Training Resource Use

# Lambda functions

I created a Lambda function named dog-breed-prediction using the lambda function.py starter.

A Boto3 client is created using this function. This enables the function to communicate with other AWS services, such as SageMaker. The function utilises the client to contact the pytorch-inference-2022-01-12-08-25-47-451 endpoint from the multi-instance trained model. It expects a JSON-formatted input/image and returns a JSON. The prediction from the endpoint is in the key 'body' of this JSON, which has a status code of 200.

{

"errorMessage": "An error occurred (AccessDeniedException)

when calling the InvokeEndpoint operation: User:

arn:aws:sts::933869125920:assumed-role/testEndpointMTrainer-role-pr3dpwbk/testEndpointMTrainer

is not authorized to perform: sagemaker:InvokeEndpoint on

resource:

arn:aws:sagemaker:us-east-1:933869125920:endpoint/pytorch-inference-2021-11-28-17-26-20-083

because no identity-based policy allows the

}

## Permissions

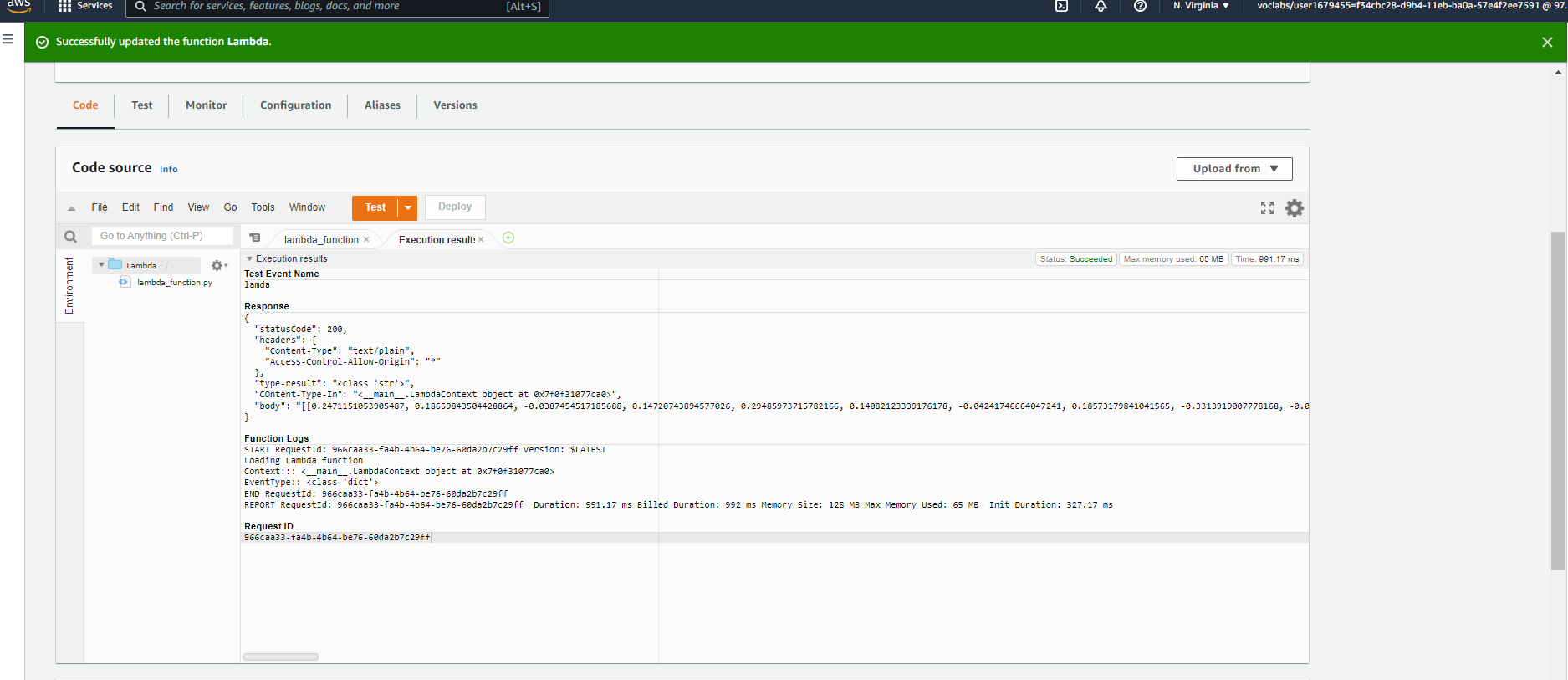
I'm worried about the permissions we've granted these lambda functions because they don't seem to follow the least privilege principle. In a perfect world, we'd only allow these lambda functions to query the endpoints for which they were created. I'll have to investigate further to see if there's anything I can do about it. These lambda functions might also be used as a second line of defence against DOS attacks on endpoints.

Figure 8: Lambda Functions success

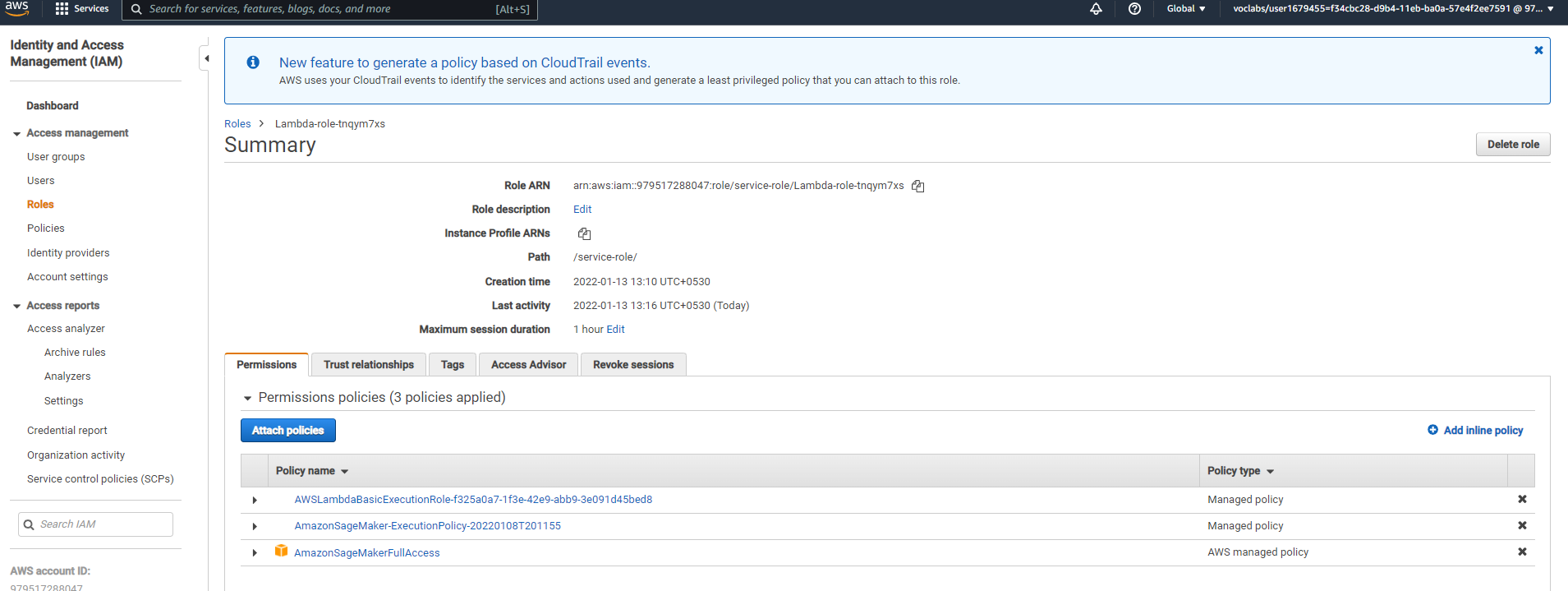


Figure 9: Lambda Functions Role Update

# Concurrency and auto-scaling

Because reserved concurrency is free and meets our present needs, I chose it over shared concurrency. Furthermore, we are unlikely to handle more than a few requests per endpoint instance at any given time, as this would indicate that the latter is overloaded, so a value that is a multiple of the number of endpoint instances makes sense, so I chose 100, which allows for 20 requests per endpoint instance. I choose to scale the endpoint to between one and five instances. While this is unlikely to ever be an issue, because each forecast takes about 0.25 seconds, this should allow for about 20 requests per second, which, along with appropriate request throttling, should allow for a large number of individuals to utilise the service.

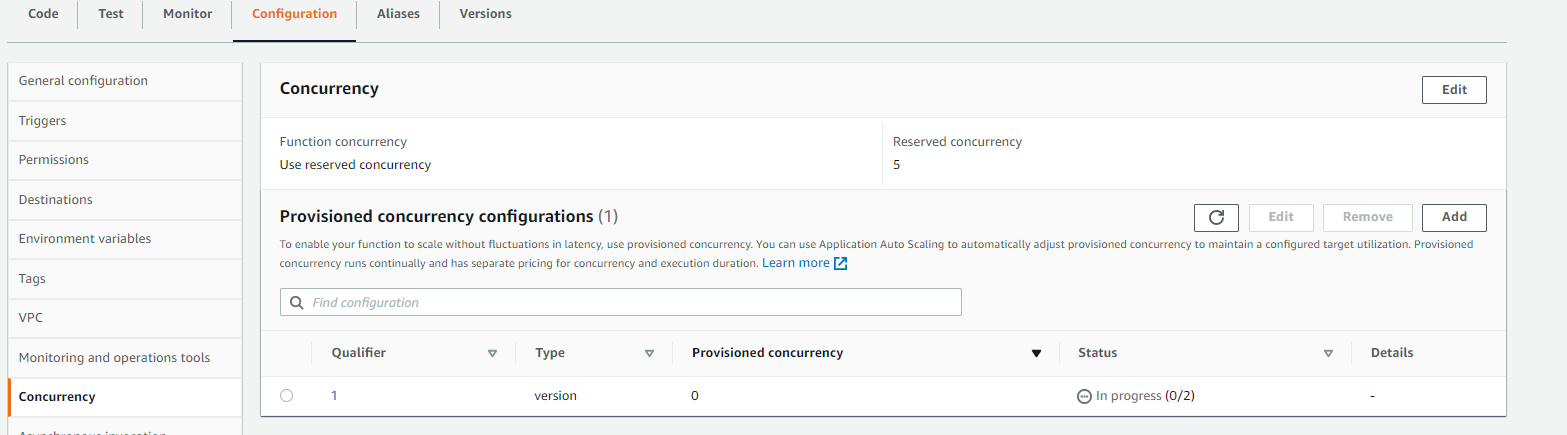


Figure 10: Configuring Concurrency

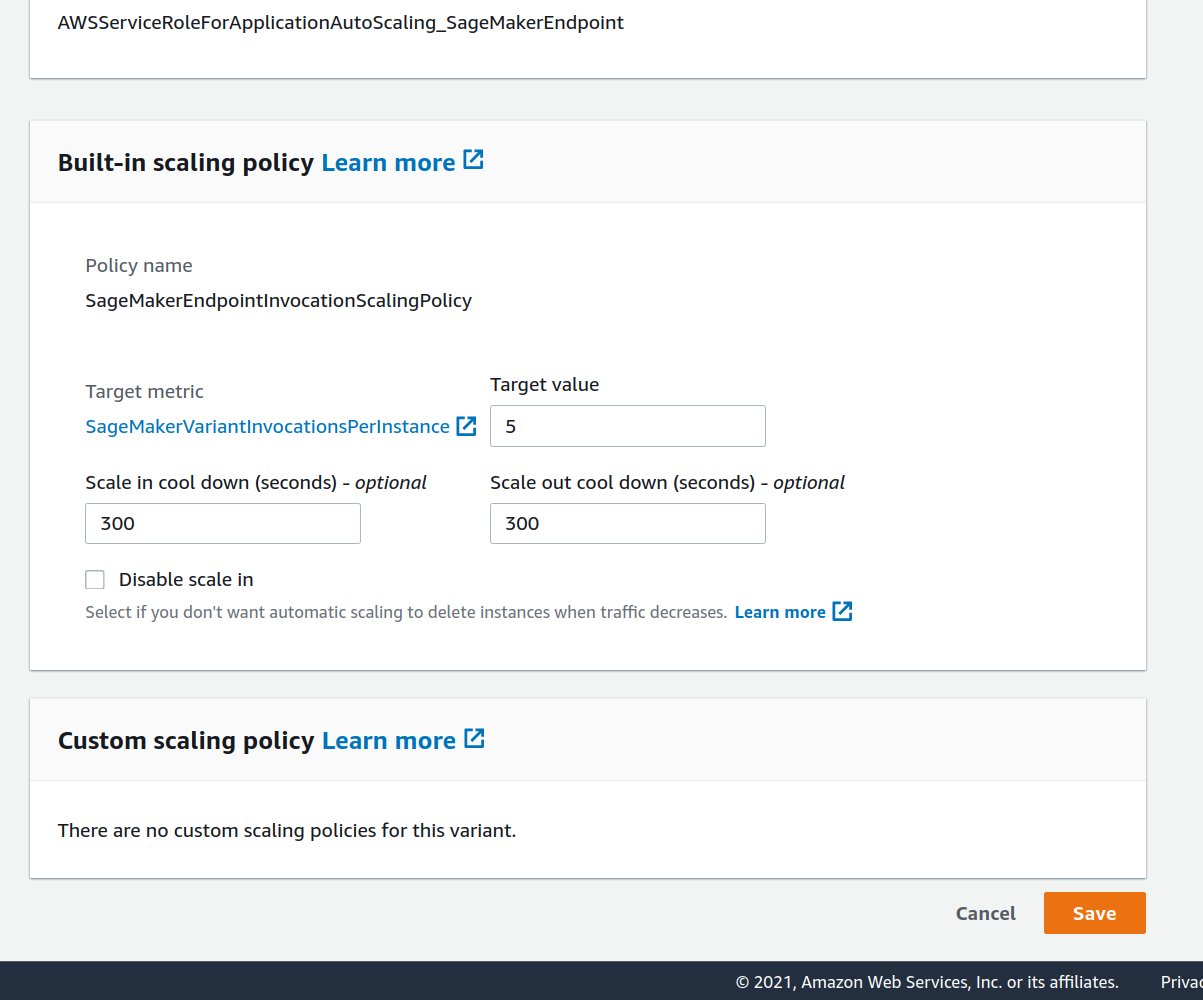


Figure 11: Configuring Concurrency