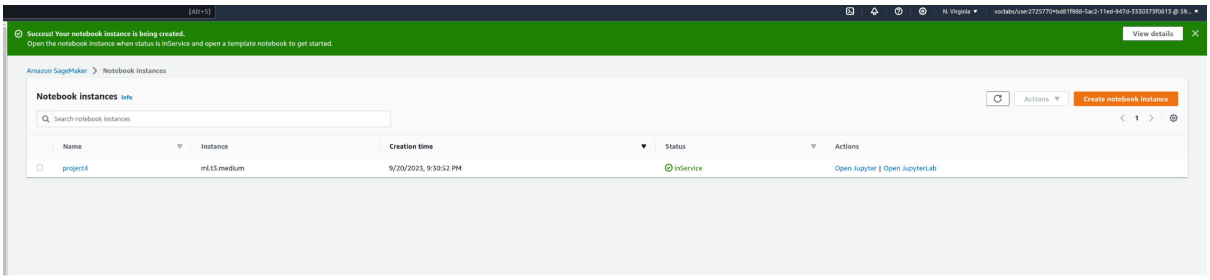
1. Write a short justification of why you chose the instance type you did.

I have chosen a 'ml.t3.medium' instance type. It only has vCPUs and 4 Gib memory, but that is more than enough to run the project's notebook, since all the heavy-weight work of training models will be performed by the instances that are specified when we create the estimators. The same logic applies for inference endpoints: the inferences are going to be run on dedicated instances that we specify when we deploy our models.

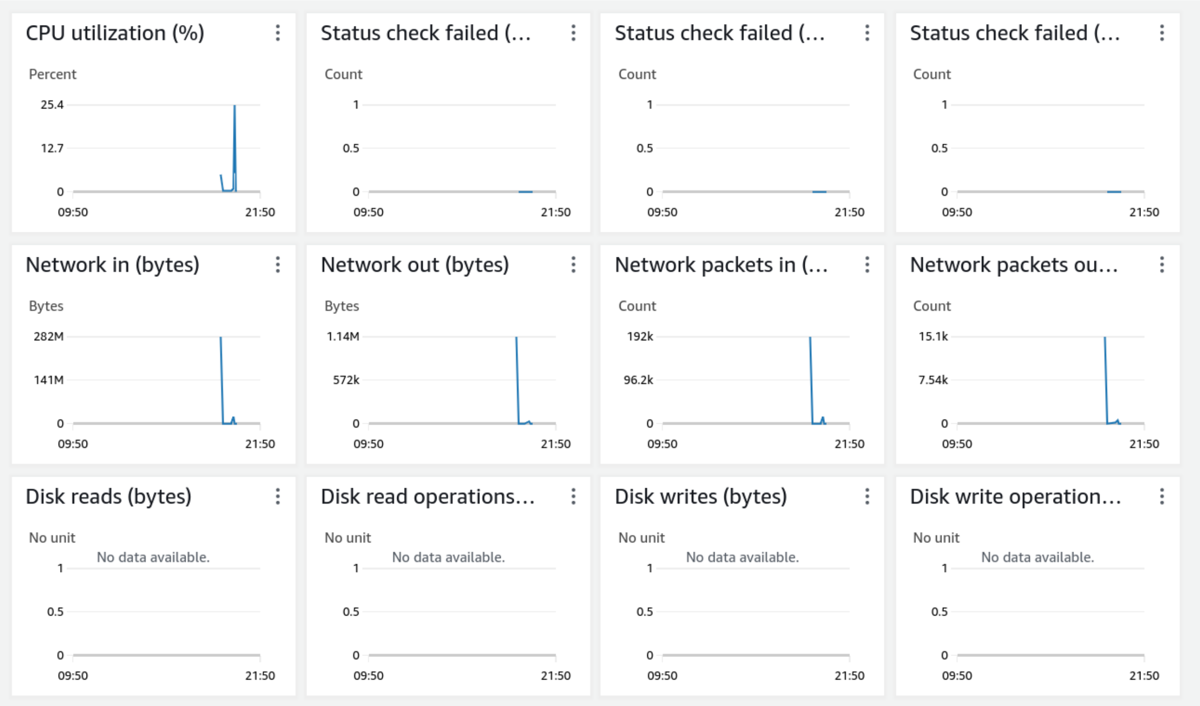
Another point of attention is the block storage size. Since we have to download the image dataset to the machine, and then upload it to S3, and we are dealing with a relatively large image dataset, I have increased the EBS size to 50Gb, which is more than enough. Taking a look at the hard drive utilization yields the following:

```
sh-4.2$ df  -h
Filesystem      Size  Used Avail Use% Mounted on
devtmpfs        1.9G     0  1.9G   0% /dev
tmpfs           1.9G  4.0K  1.9G   1% /dev/shm
tmpfs           1.9G  588K  1.9G   1% /run
tmpfs           1.9G     0  1.9G   0% /sys/fs/cgroup
/dev/nvme0n1p1  135G   96G   40G  71% /
/dev/nvme1n1     50G   12G   36G  25% /home/ec2-user/SageMaker
tmpfs           386M     0  386M   0% /run/user/1002
tmpfs           386M     0  386M   0% /run/user/1001
tmpfs           386M     0  386M   0% /run/user/1000
sh-4.2$ 
```



2. Decide the type of instance you want, create it in your workspace, and write a justification of why you chose the instance type you did.

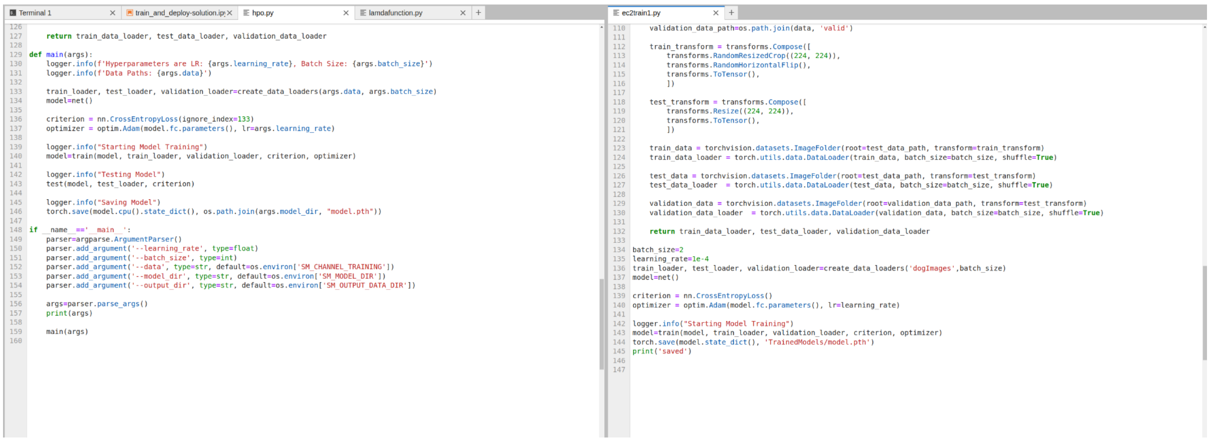
I have decided to use and ml.m5.xlarge instance type with an 'Deep Learning AMI GPU PyTorch 1.13.1' AMI. It is the same type of instance used during the sagemaker training jobs created in the notebook. The main rationale was to see how the instance's perform compared to the sagemaker job. To my surprise the training process itself was quicker than the equivalent sagemaker training job. Taking a look at the ec2 metrics, it can be seen that, at peak demand, only 25.4% of the cpu was used. So if we were running on a tight budget, we could try performing further training jobs in a smaller instance.



3. write at least one paragraph about the differences between the code in ec2train1.py and the code you used in Step 1.

In terms of code, one of the main differences is the absence of the *"if __name__=='__main__':"* in the code that is run in the ec2, because it is not necessary to set parameters that the script should receive like, data location in s3 and hyperparameters configurations. All data is already loaded in the machine and the hyperparameter values are hard coded (See image below).

When training with sagemaker estimators, our code need to be parametrizable, and comply with certain standards (like receiving arguments via command line with the proper names). Besides that, the infrastructure of where our job is going to be run is also a parameter. When you train on ec2 instances, the hardware that you have is the hardware that you get. You cannot change that, so the focus is more on optimizing the code.




```python
    return train_data_loader, test_data_loader, validation_data_loader

def main(args):
    logger.info(f'Hyperparameters are LR: {args.learning_rate}, Batch Size: {args.batch_size}')
    logger.info(f'Data Paths: {args.data}')

    train_loader, test_loader, validation_loader=create_data_loaders(args.data, args.batch_size)
    model=net()

    criterion = nn.CrossEntropyLoss(ignore_index=133)
    optimizer = optim.Adam(model.fc.parameters(), lr=args.learning_rate)

    logger.info("Starting Model Training")
    model=train(model, train_loader, validation_loader, criterion, optimizer)

    logger.info("Testing Model")
    test(model, test_loader, criterion)

    logger.info("Saving Model")
    torch.save(model.cpu().state_dict(), os.path.join(args.model_dir, "model.pth"))

if __name__=='__main__':
    parser=argparse.ArgumentParser()
    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'])

    args=parser.parse_args()
    print(args)

    main(args)
```

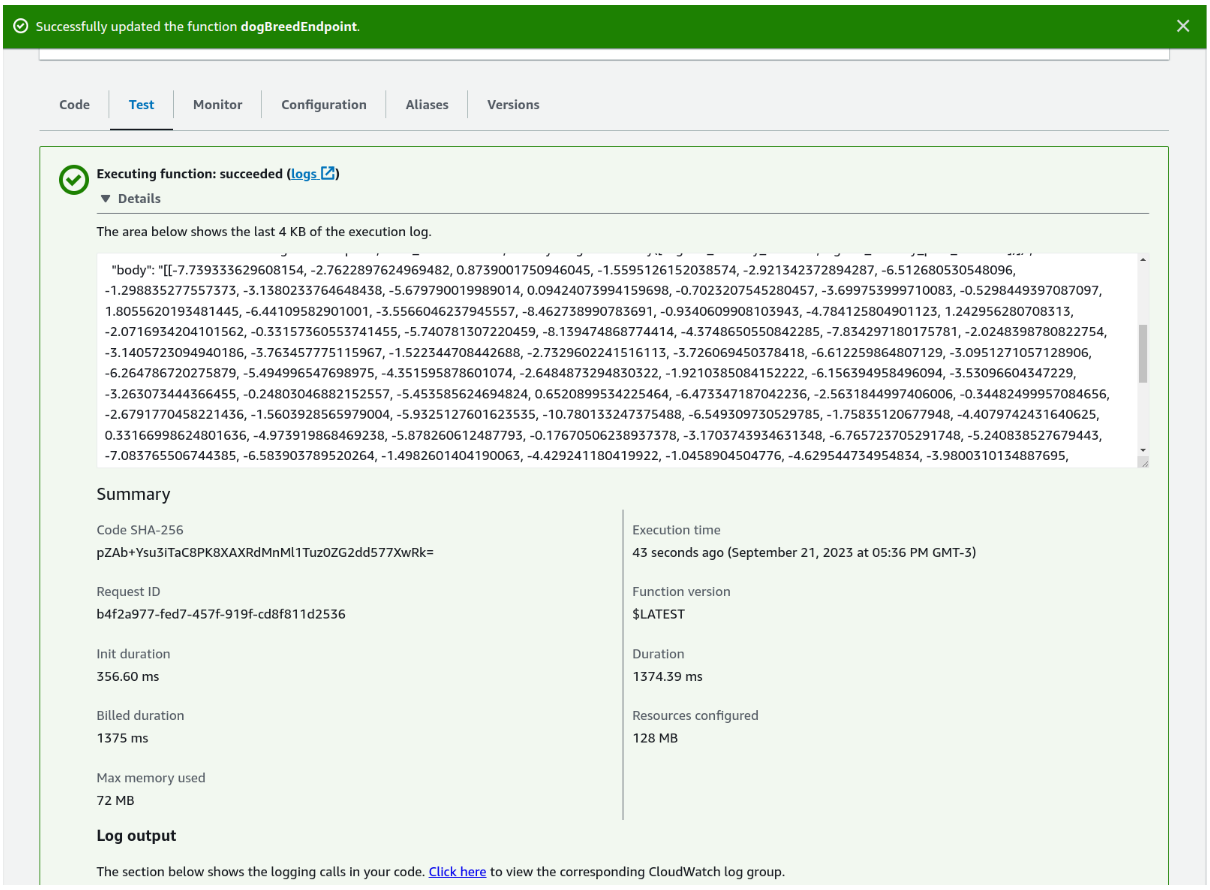
```python
    validation_data_path=os.path.join(data, 'valid')

    train_transform = transforms.Compose([
        transforms.RandomResizedCrop((224, 224)),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        ])

    test_transform = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        ])

    train_data = torchvision.datasets.ImageFolder(root=test_data_path, transform=train_transform)
    train_data_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True)

    test_data = torchvision.datasets.ImageFolder(root=test_data_path, transform=test_transform)
    test_data_loader  = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=True)

    validation_data = torchvision.datasets.ImageFolder(root=validation_data_path, transform=test_transform)
    validation_data_loader  = torch.utils.data.DataLoader(validation_data, batch_size=batch_size, shuffle=True)

    return train_data_loader, test_data_loader, validation_data_loader

batch_size=2
learning_rate=1e-4
train_loader, test_loader, validation_loader=create_data_loaders('dogImages',batch_size)
model=net()

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.fc.parameters(), lr=learning_rate)

logger.info("Starting Model Training")
model=train(model, train_loader, validation_loader, criterion, optimizer)
torch.save(model.state_dict(), 'TrainedModels/model.pth')
print('saved')
```

3. Write at least 1 paragraph describing how this function is written and how it works.

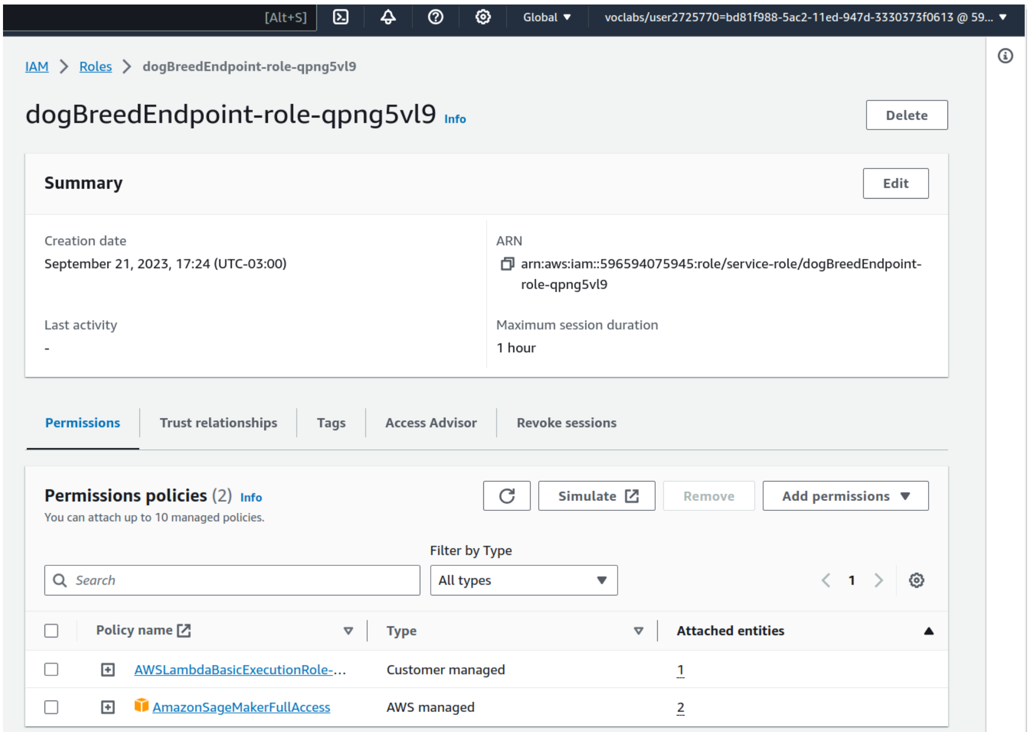
The function works by receiving a payload with the data that should go through our trained model. This payload is called and event. In this specific case, the event is a json object which contains the url of the image against which we should run our predictions. We use a sagemaker runtime to invoke the endpoint, passing the endpoint name, type of payload, type of accepted response, and the payload itself.

After sending the request, we receive another json object, called response. In the response body, we find the content of our prediction. This content is then desserialized(converted from bytes to python object). In final part, the function return the success status code, along with other response parameters and the serialized and parsed response.

4. Add the result that your lambda function returns (the list of 33 numbers) to your solution writeup



5. Write about the security of your AWS workspace in your writeup.



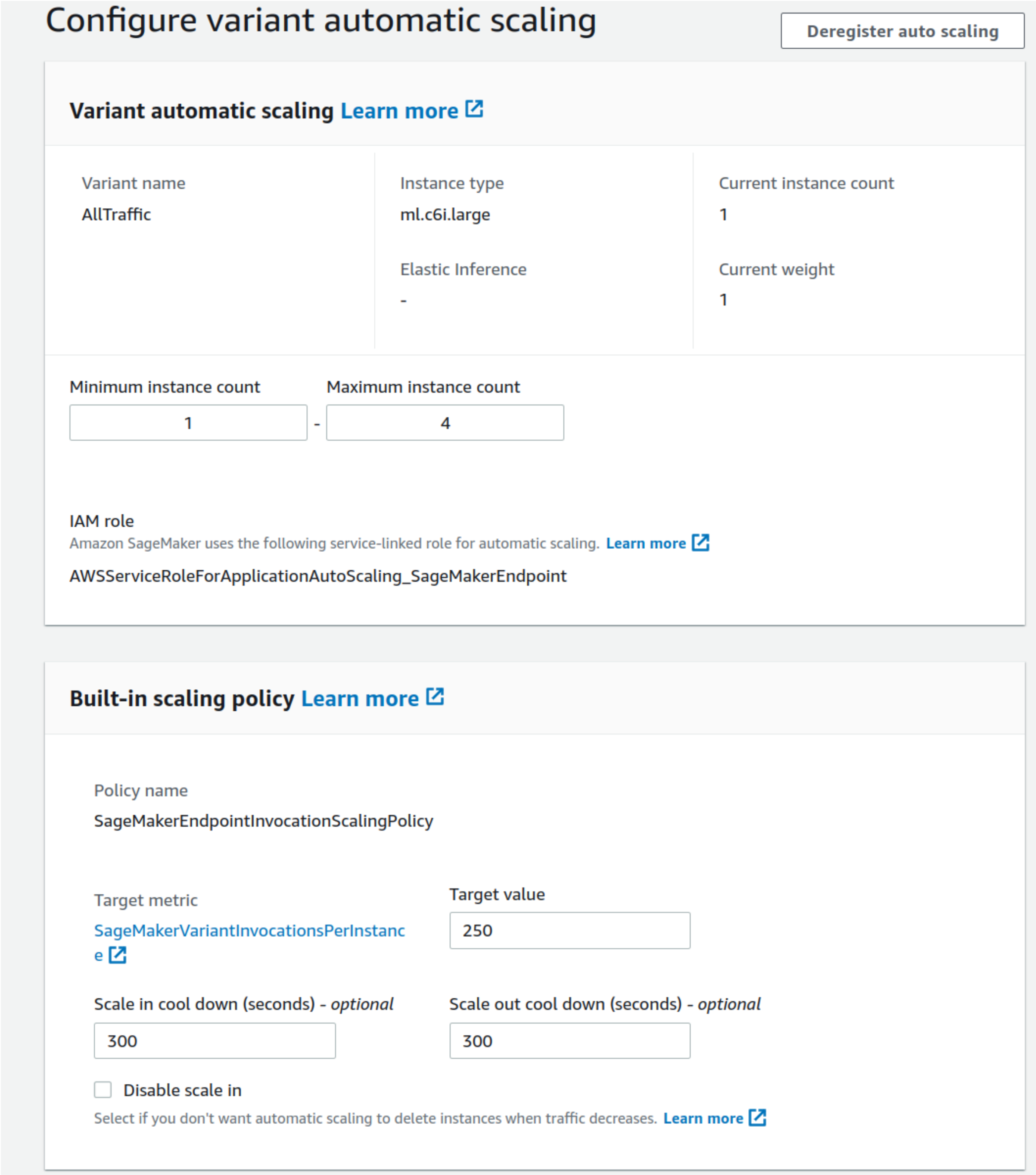
Due to the absence of a better suited AWS managed policy, I have used policy called "AmazonSagemakerFullAccess", which, as the name suggest, grands full access to sagemaker services. This is against AWS *Principle of Least Priviledge*, which states that a resource should not be given any more privileges than necessary. Since our function is only going to invoke endpoints on sagemaker, the ideal would be to go to IAM, customize this policy, and remove all access that are not relevant for this lambda function.

6. Write about the choices you made in the setup of concurrency and auto-scaling, and why you made each of those choices.

I have set the function to use the unreserved account concurrency of 900 invocations plus a provisioned concurrency of 100 invocations. Under this setting, there is 100 execution environments that are initialized and prepared to respond immediately to function invocations. If the throughput goes beyond this threshold, there is still 900 execution environments available, but they will not be as performant, mainly in terms of latency, as the ones in from the provisioned concurrency.

For autoscaling I have configured a minimum of 1 instance and a maxium of 4. The number 4 was chosen thinking about a scenario of peak demand, when some sort of event like a promotion, or our application goes viral, makes the amount of request to our model increase substantially. The scaling policy was set in a way that, as soon as the instance get 250 "simultaneous"(short time-spaced) invocations per instance, the number of instances will be increased. Both the scale in and out periods are of 300 seconds.

In a real world scenario, to estimate the optimal concurrency and auto-scaling configuration to minimize costs, latency and satisfy other business requirements, we would need to analyze historical invocation behavior and decide the appropiate amounts of each type of concurrency.


Configure variant automatic scaling
Deregister auto scaling
Variant automatic scaling Learn more
Variant name
AllTraffic
Instance type
ml.c6i.large
Elastic Inference
-
Current instance count
1
Current weight
1
Minimum instance count
1
-
Maximum instance count
4
IAM role
Amazon SageMaker uses the following service-linked role for automatic scaling. Learn more
AWSServiceRoleForApplicationAutoScaling_SageMakerEndpoint
Built-in scaling policy Learn more
Policy name
SageMakerEndpointInvocationScalingPolicy
Target metric
SageMakerVariantInvocationsPerInstance
Target value
250
Scale in cool down (seconds) - optional
300
Scale out cool down (seconds) - optional
300
Disable scale in
Select if you don't want automatic scaling to delete instances when traffic decreases. Learn more