AWS FSx for NetApp ONTAP (FSxN) for MLOps

NetApp Solutions

NetApp
April 26, 2024

Table of Contents

AWS FSx for NetApp ONTAP (FSxN) for MLOps

Author(s):
Jian Jian (Ken), Senior Data & Applied Scientist, NetApp

This section delves into the practical application of AI infrastructure development, providing an end-to-end walkthrough of constructing an MLOps pipeline using FSxN. Comprising three comprehensive examples, it guides you to meet your MLOps needs via this powerful data management platform.

These articles focus on:

1. Part 1 - Integrating AWS FSx for NetApp ONTAP (FSxN) as a private S3 bucket into AWS SageMaker
2. Part 2 - Leveraging AWS FSx for NetApp ONTAP (FSxN) as a Data Source for Model Training in SageMaker
3. Part 3 - Building A Simplified MLOps Pipeline (CI/CT/CD)

By the end of this section, you will have gained a solid understanding of how to use FSxN to streamline MLOps processes.

Part 1 - Integrating AWS FSx for NetApp ONTAP (FSxN) as a private S3 bucket into AWS SageMaker

Author(s):
Jian Jian (Ken), Senior Data & Applied Scientist, NetApp

Introduction

Using SageMaker as an example, this page provides guidance on configuring FSxN as a private S3 bucket.

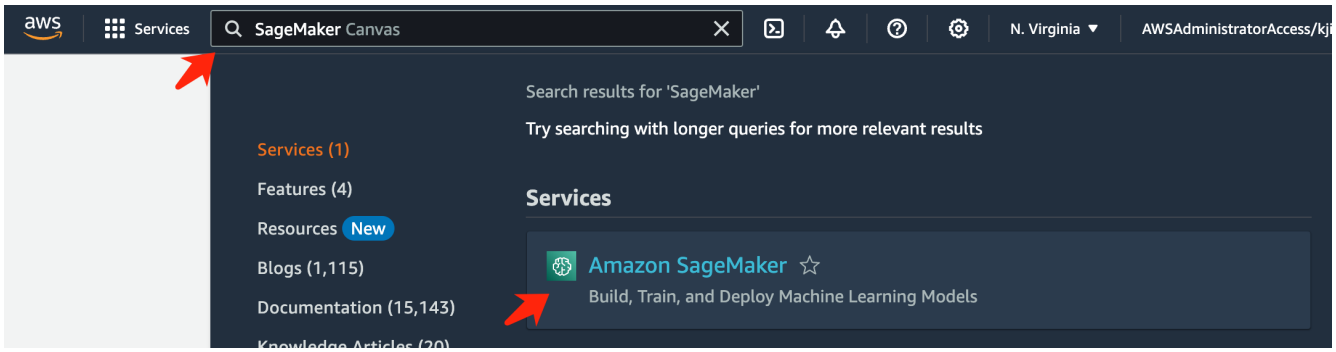
For more information about FSxN, please take a look at this presentation (Video Link)

User Guide

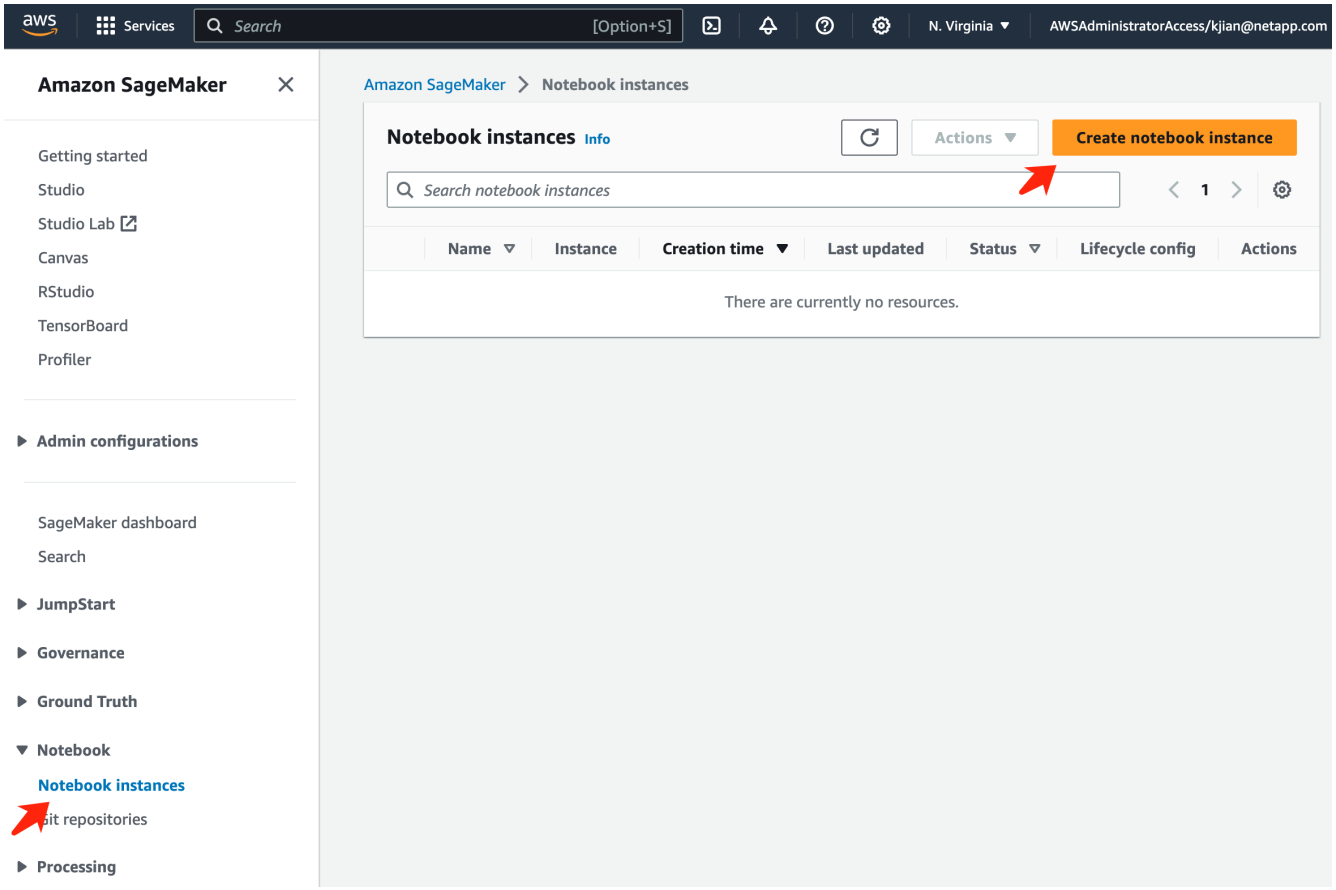
Server creation

Create a SageMaker Notebook Instance

1. Open AWS console. In the search panel, search SageMaker and click the service **Amazon SageMaker**.



2. Open the **Notebook instances** under Notebook tab, click the orange button **Create notebook instance**.



3. In the creation page,
   Enter the **Notebook instance name**
   Expand the **Network** panel
   Leave other entries default and select a **VPC**, **Subnet**, and **Security group(s)**. (This **VPC** and **Subnet** will be used to create FSxN file system later)
   Click the orange button **Create notebook instance** at the bottom right.

Amazon SageMaker > Notebook instances > Create notebook instance

Create notebook instance

Amazon SageMaker provides pre-built fully managed notebook instances that run Jupyter notebooks. The notebook instances include example code for common model training and hosting exercises. Learn more

Notebook instance settings

Notebook instance name

fsxn-demo

Maximum of 63 alphanumeric characters. Can include hyphens (-), but not spaces. Must be unique within your account in an AWS Region.

Notebook instance type

ml.t3.medium

Elastic Inference Learn more

none

Platform identifier Learn more

Amazon Linux 2, Jupyter Lab 3

▶ Additional configuration

Permissions and encryption

IAM role

Notebook instances require permissions to call other services including SageMaker and S3. Choose a role or let us create a role with the **AmazonSageMakerFullAccess** IAM policy attached.

AmazonSageMakerServiceCatalogProductsUseRole

Create role using the role creation wizard

Root access - *optional*

- (●) Enable - Give users root access to the notebook
- ( ) Disable - Don't give users root access to the notebook
  Lifecycle configurations always have root access

Encryption key - *optional*

Encrypt your notebook data. Choose an existing KMS key or enter a key's ARN.

No Custom Encryption

▼ Network - *optional*

VPC - *optional*

Default vpc-0df3956ab1fca2ec9 (172.31.0.0/16)

Subnet

Choose a subnet in an availability zone supported by Amazon SageMaker.

subnet-00060df0d0f562672 (172.31.16.0/20) | us-east-1a

Security group(s)

sg-0a39b3985770e9256 (default) ✕

Direct internet access

- (●) Enable — Access the internet directly through Amazon SageMaker
- ( ) Disable — Access the internet through a VPC
  To train or host models from a notebook, you need internet access. To enable internet access, make sure that your VPC has a NAT gateway and your security group allows outbound connections. Learn more

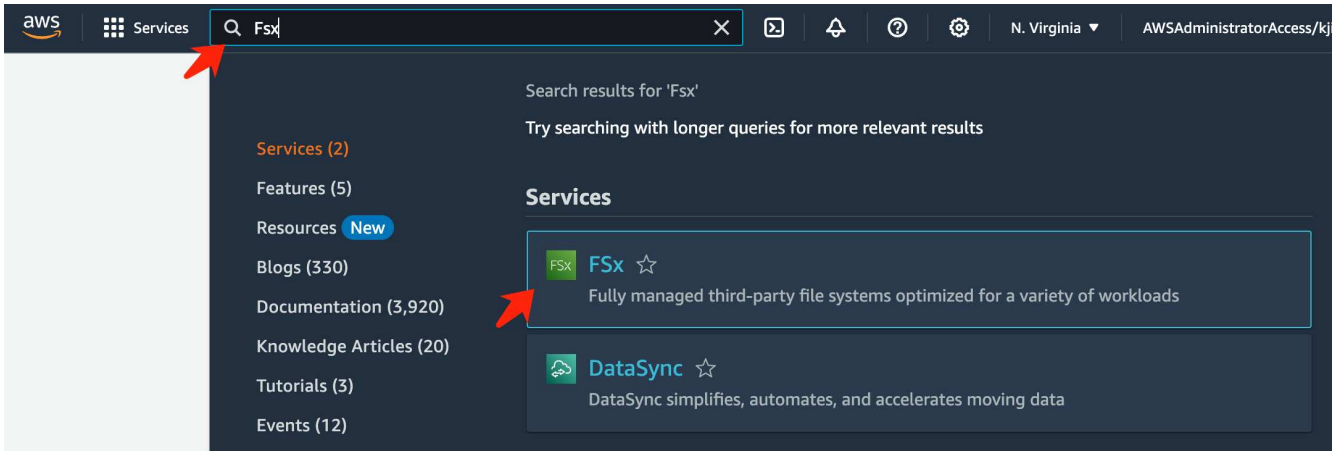
▶ Git repositories- *optional*

▶ Tags - *optional*

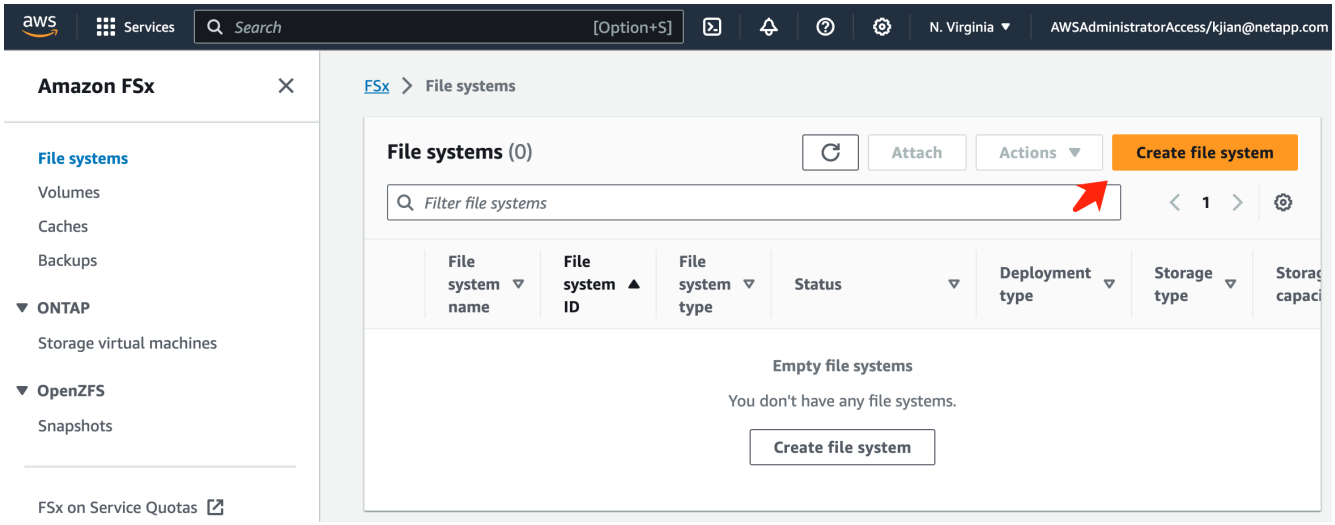
Cancel | Create notebook instance

Create an FSxN File System

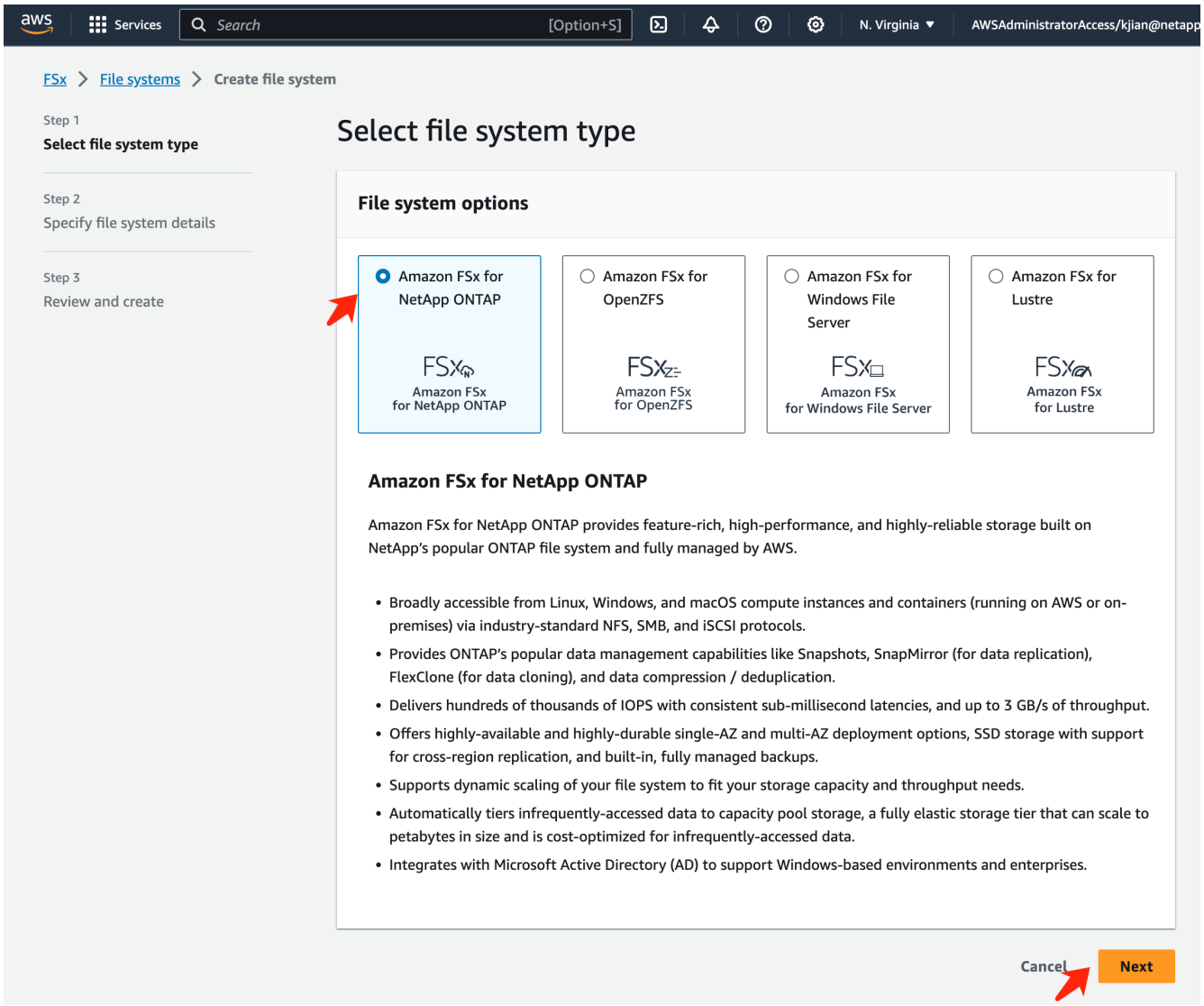
1. Open AWS console. In the search panel, search Fsx and click the service **FSx**.



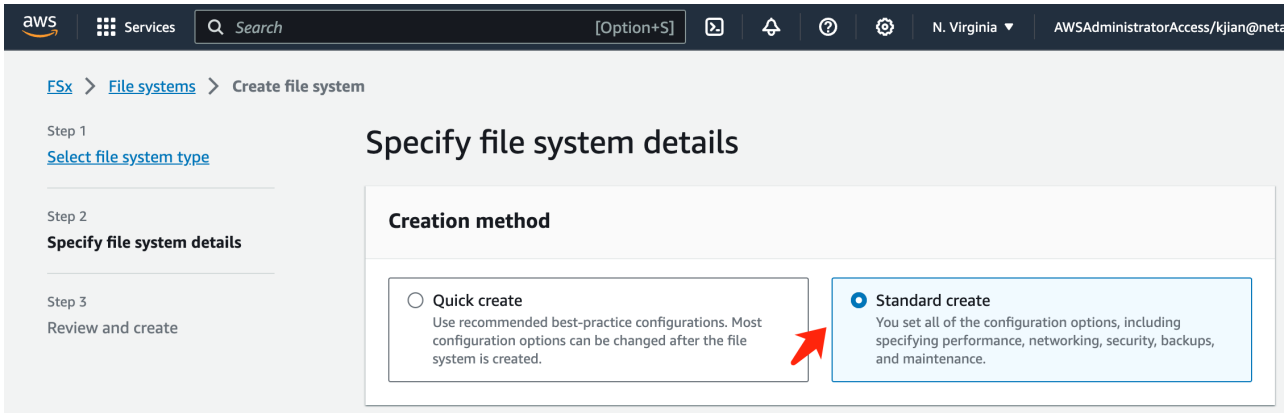
2. Click **Create file system**.



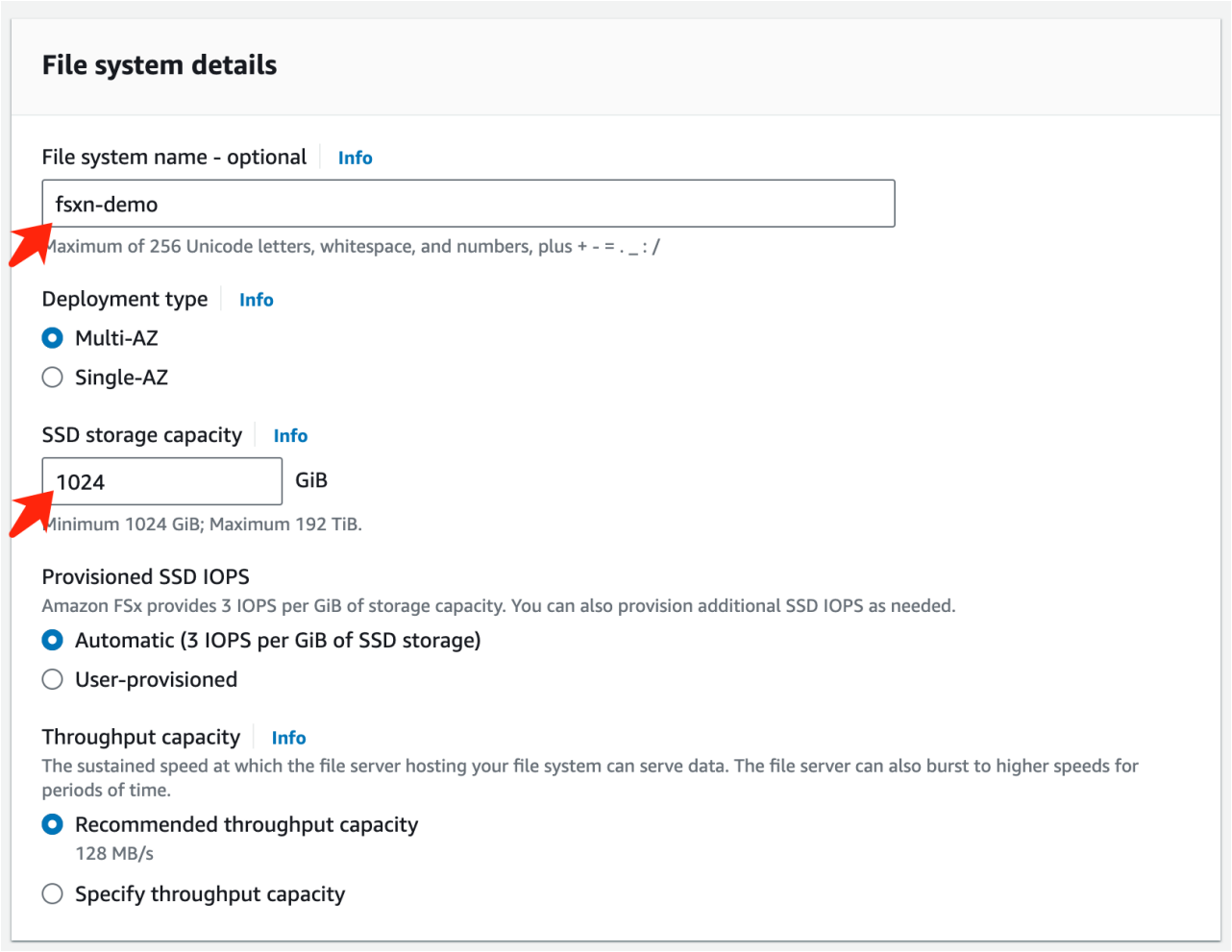
3. Select the first card **FSx for NetApp ONTAP** and click **Next**.



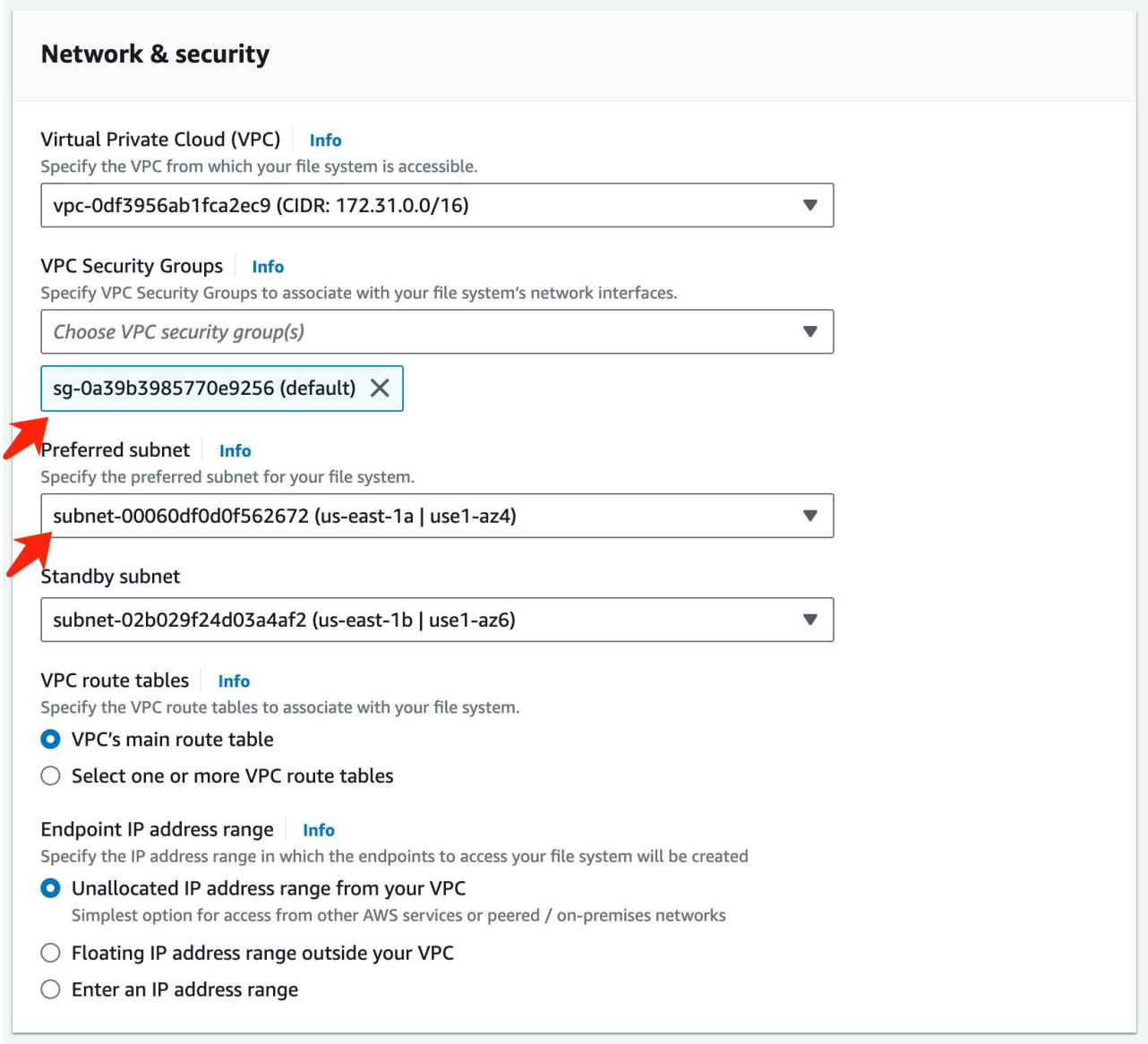
4. In the details configuration page.
   a. Select the **Standard create** option.



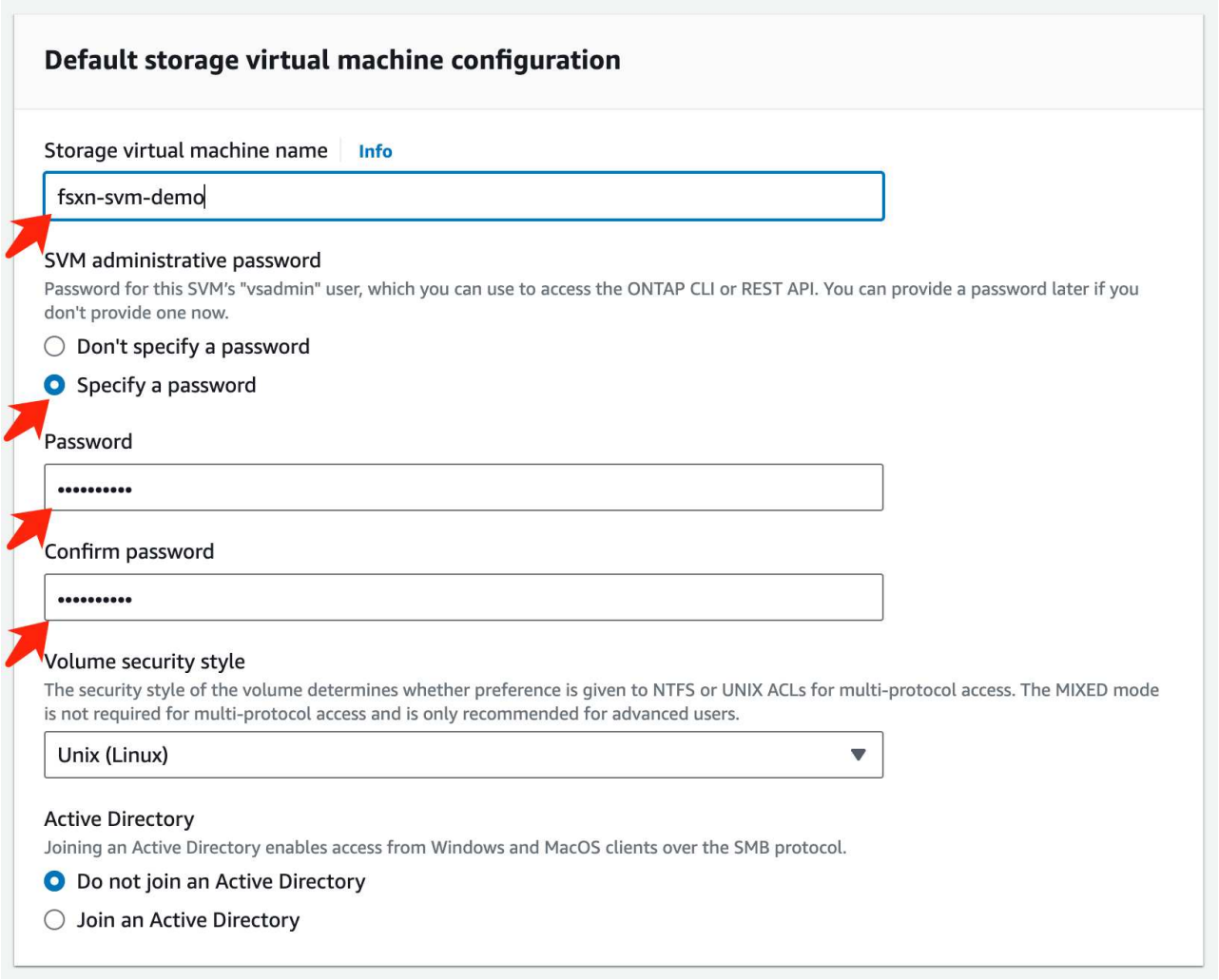
   b. Enter the **File system name** and the **SSD storage capacity**.



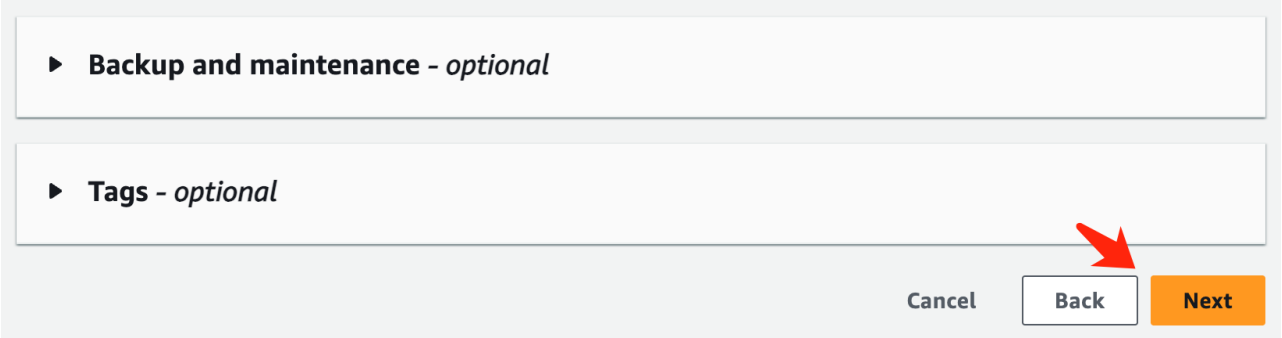
c. Make sure to use the **VPC** and **subnet** same to the **SageMaker Notebook** instance.



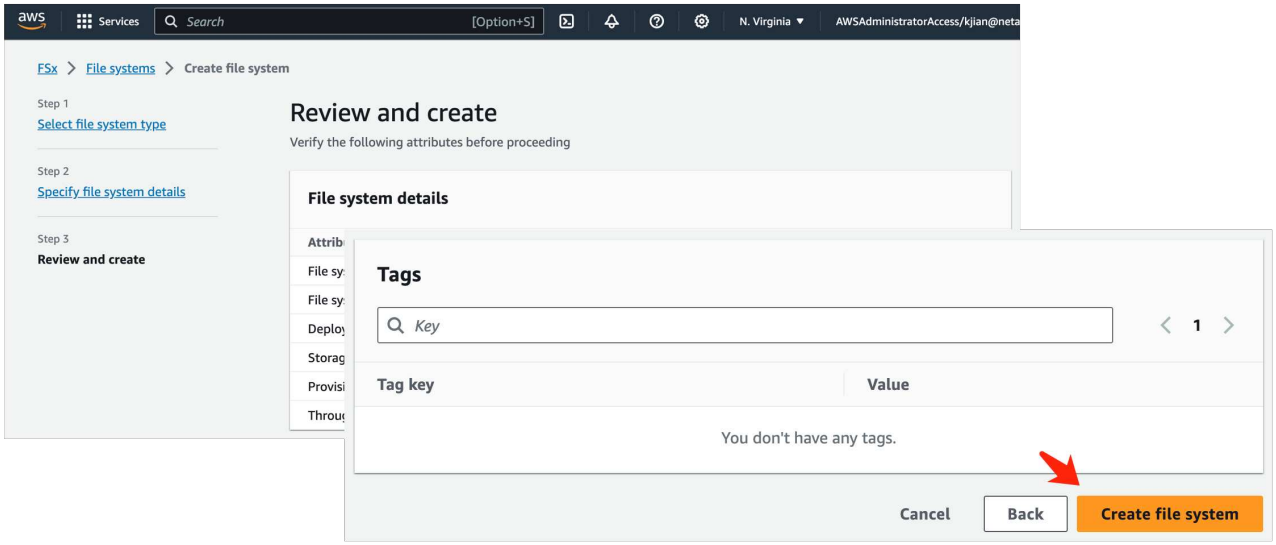
d. Enter the **Storage virtual machine** name and **Specify a password** for your SVM (storage virtual machine).



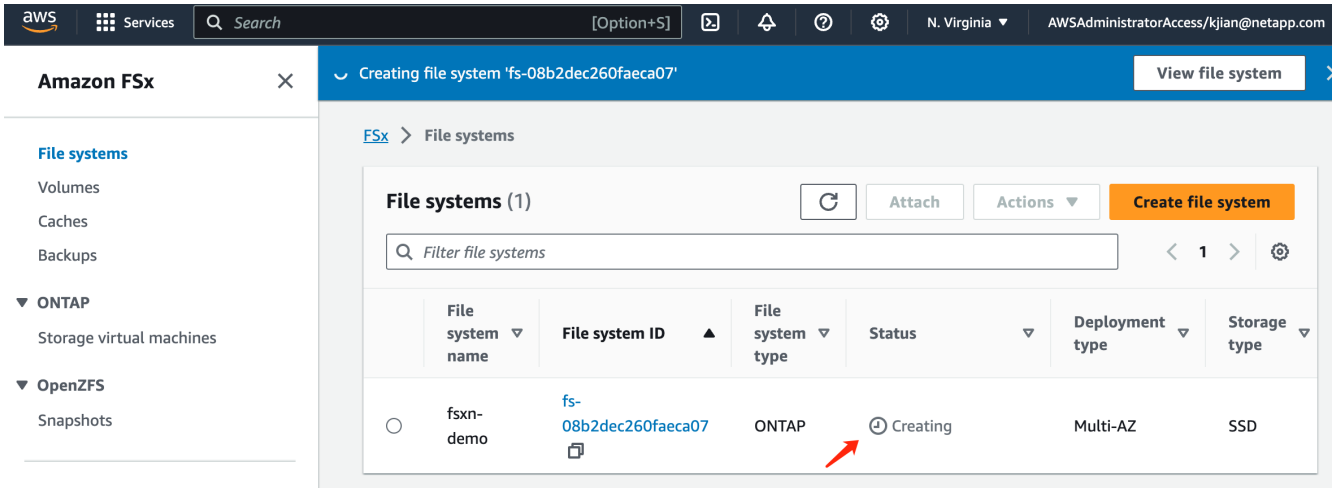
e. Leave other entries default and click the orange button **Next** at the bottom right.



f. Click the orange button **Create file system** at the bottom right of the review page.



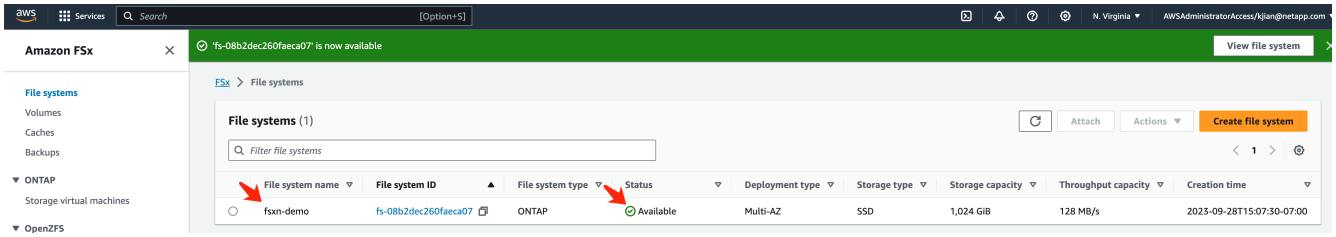
5. It may takes about **20-40 minutes** to spin up the FSx file system.



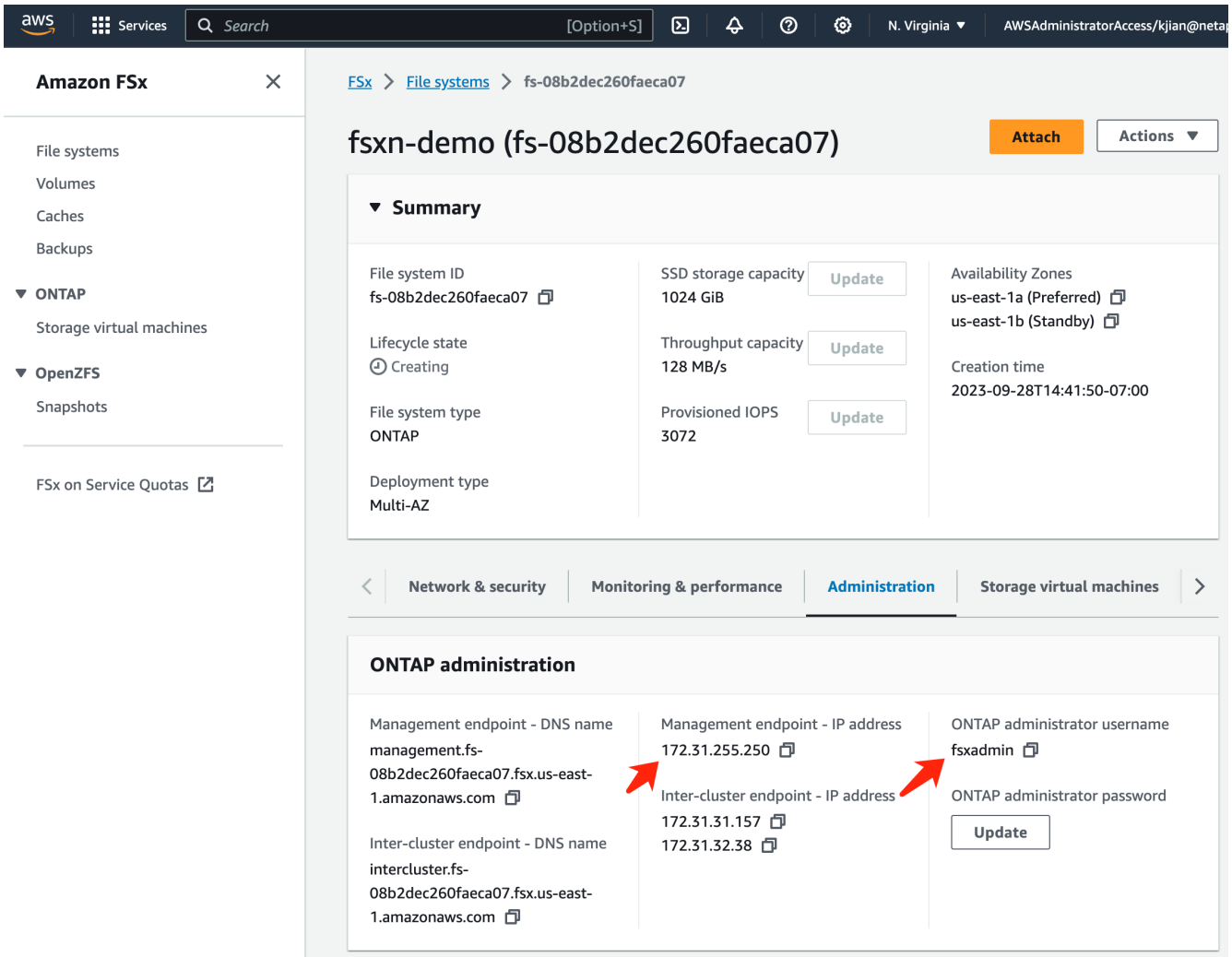
Server Configuration

ONTAP Configuration

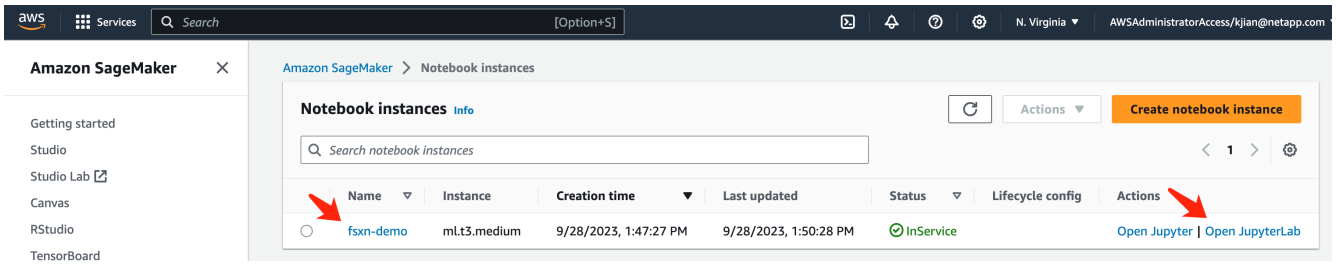
1. Open the created FSx file system. Please make sure the status is **Available**.



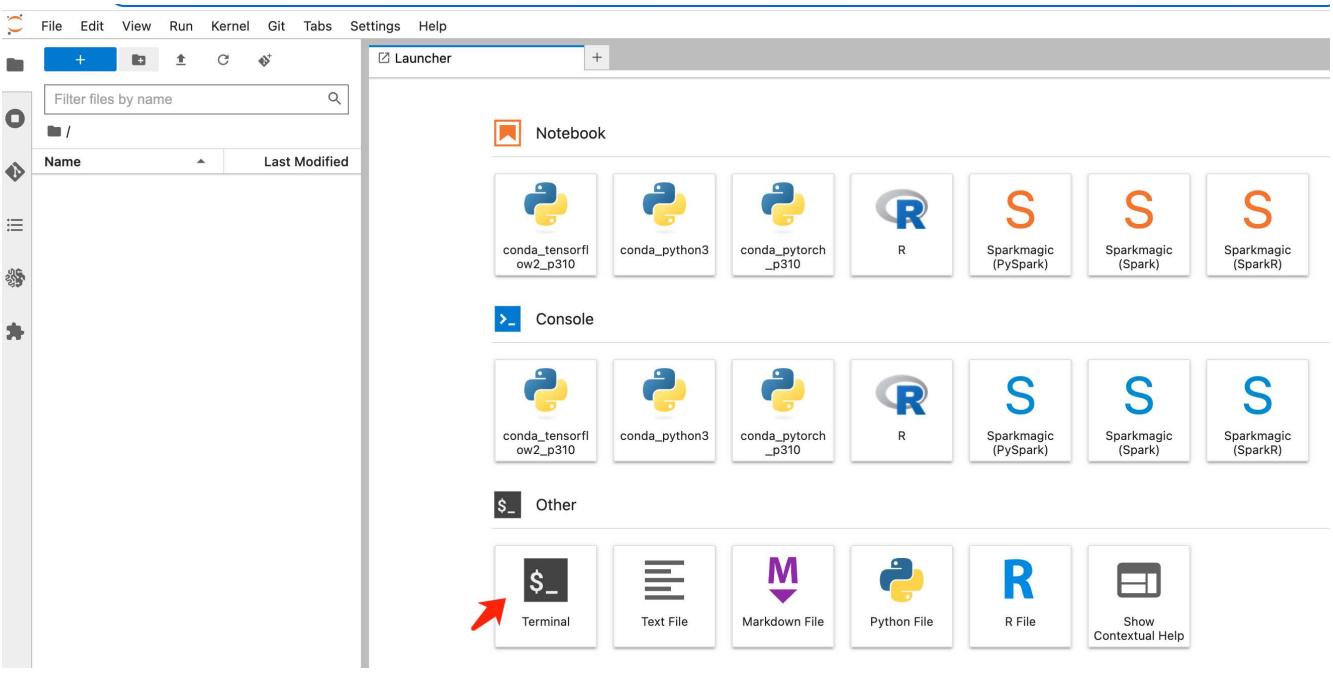
2. Select the **Administration** tab and keep the **Management endpoint - IP address** and **ONTAP administrator username**.



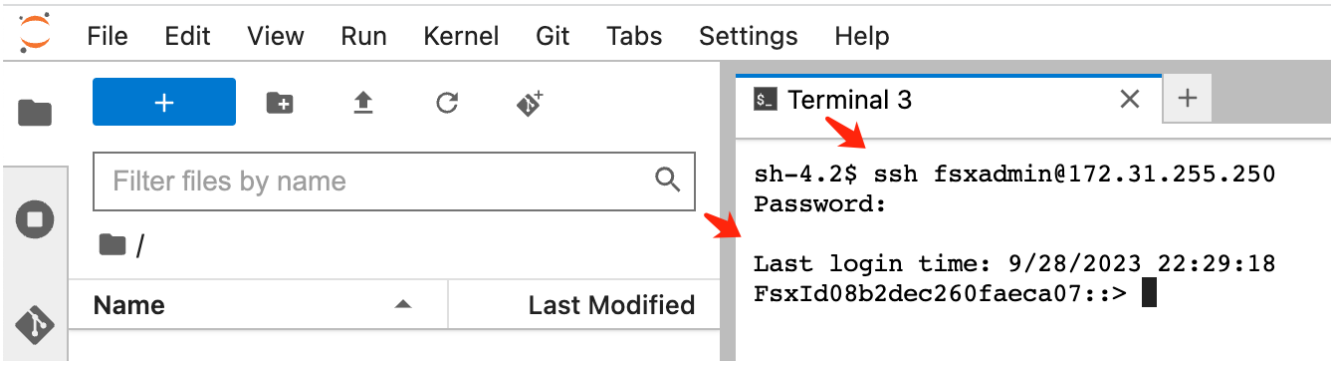
3. Open the created **SageMaker Notebook instance** and click **Open JupyterLab**.



4. In the Jupyter Lab page, open a new **Terminal**.

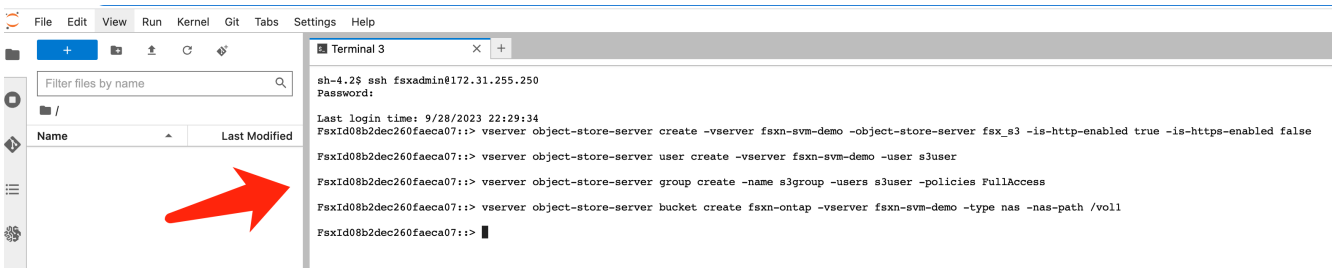


5. Enter the ssh command ssh <admin user name>@<ONTAP server IP> to login to the FSxN ONTAP file system. (The user name and IP address are retrieved from the step 2)
   Please use the password used when creating the **Storage virtual machine**.



6. Execute the commands in the following order.
   We use **fsxn-ontap** as the name for the **FSxN private S3 bucket name**.
   Please use the **storage virtual machine name** for the **-vserver** argument.

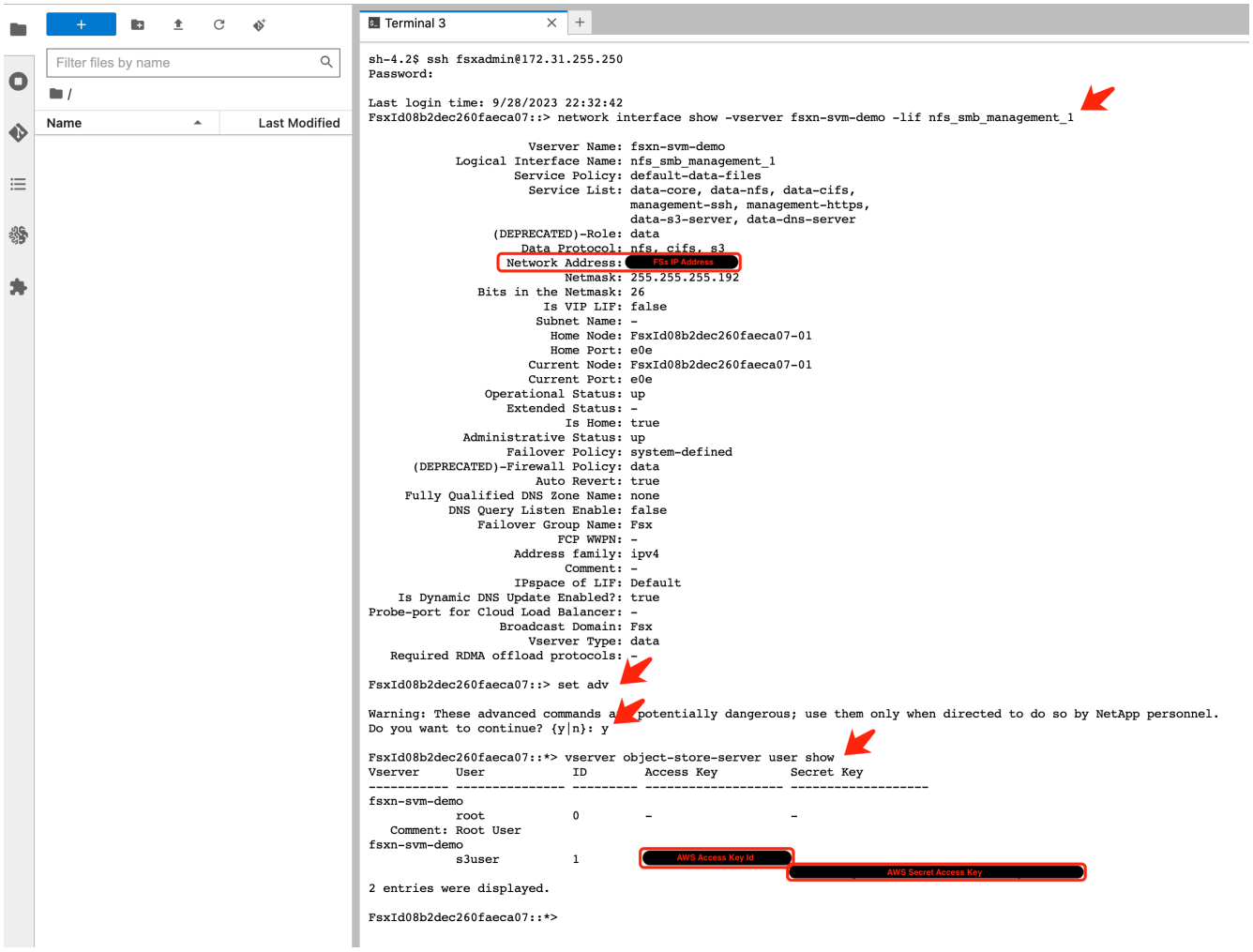
```
vserver object-store-server create -vserver fsxn-svm-demo -object-store
-server fsx_s3 -is-http-enabled true -is-https-enabled false

vserver object-store-server user create -vserver fsxn-svm-demo -user
s3user

vserver object-store-server group create -name s3group -users s3user
-policies FullAccess

vserver object-store-server bucket create fsxn-ontap -vserver fsxn-svm-
demo -type nas -nas-path /vol1
```



7. Execute the below commands to retrieve the endpoint IP and credentials for FSxN private S3.

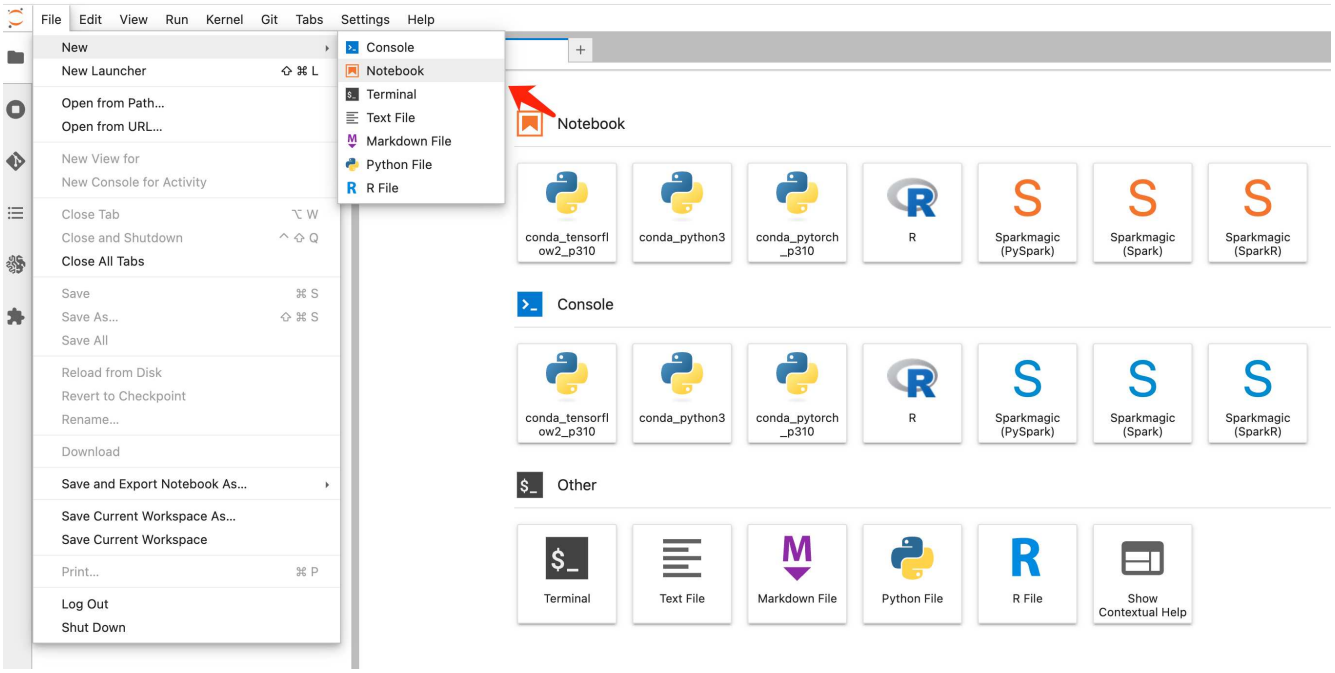
```
network interface show -vserver fsxn-svm-demo -lif nfs_smb_management_1

set adv

vserver object-store-server user show
```

8. Keep the endpoint IP and credential for future use.



Client Configuration

1. In SageMaker Notebook instance, create a new Jupyter notebook.



2. Use the below code as a work around solution to upload files to FSxN private S3 bucket.
For a comprehensive code example please refer to this notebook.
fsxn_demo.ipynb

```
# Setup configurations
# -------- Manual configurations --------
seed: int = 77                                                  # Random
seed
bucket_name: str = 'fsxn-ontap'                                 # The bucket
name in ONTAP
aws_access_key_id = '<Your ONTAP bucket key id>'                # Please get
this credential from ONTAP
aws_secret_access_key = '<Your ONTAP bucket access key>'        # Please get
this credential from ONTAP
fsx_endpoint_ip: str = '<Your FSxN IP address>'                 # Please get
this IP address from FSXN
# -------- Manual configurations --------

# Workaround
## Permission patch
!mkdir -p vol1
!sudo mount -t nfs $fsx_endpoint_ip:/vol1 /home/ec2-user/SageMaker/vol1
!sudo chmod 777 /home/ec2-user/SageMaker/vol1

## Authentication for FSxN as a Private S3 Bucket
!aws configure set aws_access_key_id $aws_access_key_id
```

```
!aws configure set aws_secret_access_key $aws_secret_access_key

## Upload file to the FSxN Private S3 Bucket
%%capture
local_file_path: str = <Your local file path>

!aws s3 cp --endpoint-url http://$fsx_endpoint_ip /home/ec2-user
/SageMaker/$local_file_path  s3://$bucket_name/$local_file_path

# Read data from FSxN Private S3 bucket
## Initialize a s3 resource client
import boto3

# Get session info
region_name = boto3.session.Session().region_name

# Initialize Fsxn S3 bucket object
# --- Start integrating SageMaker with FSXN ---
# This is the only code change we need to incorporate SageMaker with
FSXN
s3_client: boto3.client = boto3.resource(
    's3',
    region_name=region_name,
    aws_access_key_id=aws_access_key_id,
    aws_secret_access_key=aws_secret_access_key,
    use_ssl=False,
    endpoint_url=f'http://{fsx_endpoint_ip}',
    config=boto3.session.Config(
        signature_version='s3v4',
        s3={'addressing_style': 'path'}
    )
)
# --- End integrating SageMaker with FSXN ---

## Read file byte content
bucket = s3_client.Bucket(bucket_name)

binary_data = bucket.Object(data.filename).get()['Body']
```

This concludes the integration between FSxN and the SageMaker instance.

Useful debugging checklist

- Ensure that the SageMaker Notebook instance and FSxN file system are in the same VPC.
- Remember to run the **set dev** command on ONTAP to set the privilege level to **dev**.

FAQ (As of Sep 27, 2023)

Q: Why am I getting the error "**An error occurred (NotImplemented) when calling the CreateMultipartUpload operation: The s3 command you requested is not implemented**" when uploading files to FSxN?

A: As a private S3 bucket, FSxN supports uploading files up to 100MB. When using the S3 protocol, files larger than 100MB are divided into 100MB chunks, and the 'CreateMultipartUpload' function is called. However, the current implementation of FSxN private S3 does not support this function.

Q: Why am I getting the error "**An error occurred (AccessDenied) when calling the PutObject operations: Access Denied**" when uploading files to FSxN?

A: To access the FSxN private S3 bucket from a SageMaker Notebook instance, switch the AWS credentials to the FSxN credentials. However, granting write permission to the instance requires a workaround solution that involves mounting the bucket and running the 'chmod' shell command to change the permissions.

Q: How can I integrate the FSxN private S3 bucket with other SageMaker ML services?

A: Unfortunately, the SageMaker services SDK does not provide a way to specify the endpoint for the private S3 bucket. As a result, FSxN S3 is not compatible with SageMaker services such as Sagemaker Data Wrangler, Sagemaker Clarify, Sagemaker Glue, Sagemaker Athena, Sagemaker AutoML, and others.

Part 2 - Leveraging AWS FSx for NetApp ONTAP (FSxN) as a Data Source for Model Training in SageMaker

Author(s):

Jian Jian (Ken), Senior Data & Applied Scientist, NetApp

Introduction

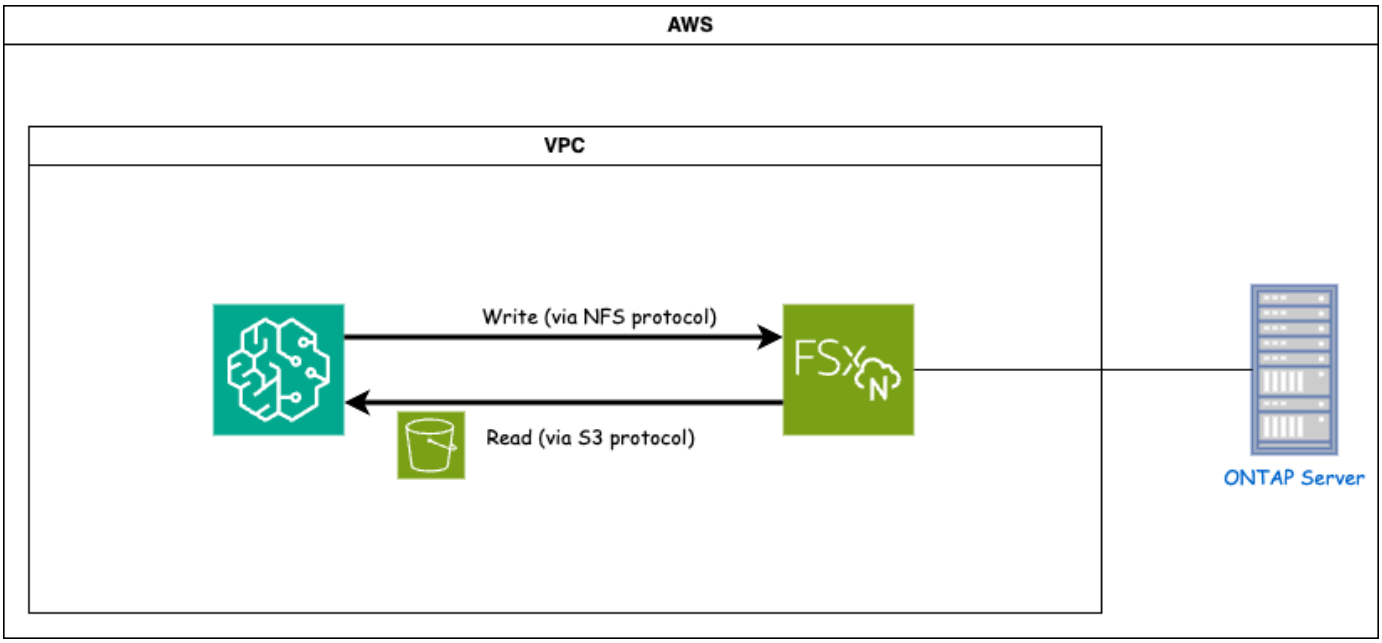
This tutorial offers a practical example of a computer vision classification project, providing hands-on experience in building ML models that utilize FSxN as the data source within the SageMaker environment. The project focuses on using PyTorch, a deep learning framework, to classify tire quality based on tire images. It emphasizes the development of machine learning models using FSxN as the data source in Amazon SageMaker.

What is FSxN

Amazon FSx for NetApp ONTAP is indeed a fully managed storage solution offered by AWS. It leverages NetApp's ONTAP file system to provide reliable and high-performance storage. With support for protocols like NFS, SMB, and iSCSI, it allows seamless access from different compute instances and containers. The service is designed to deliver exceptional performance, ensuring fast and efficient data operations. It also offers high availability and durability, ensuring that your data remains accessible and protected. Additionally, the storage capacity of Amazon FSx for NetApp ONTAP is scalable, allowing you to easily adjust it according to your needs.

Prerequisite

Network Environment



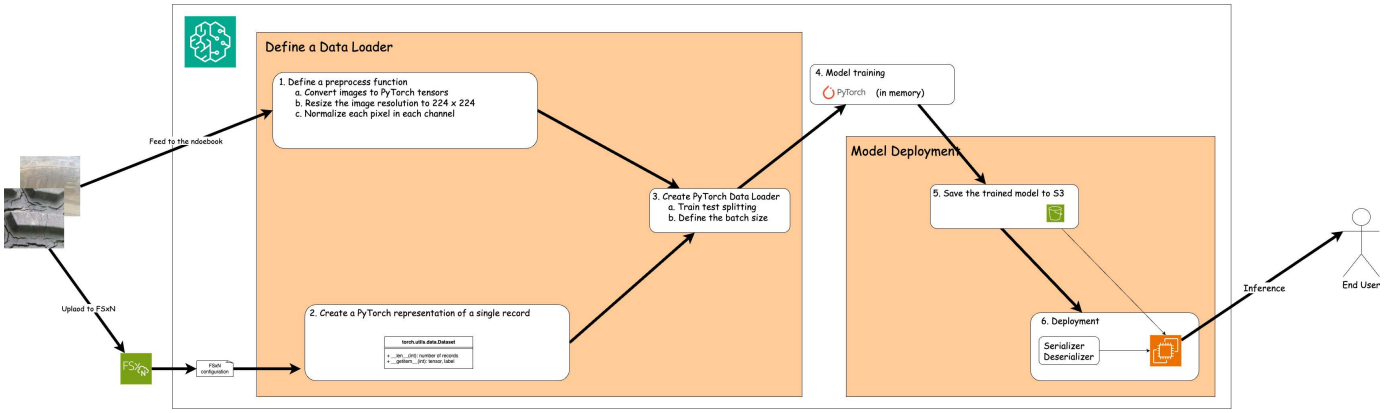
FSxN (Amazon FSx for NetApp ONTAP) is an AWS storage service. It includes a file system running on the NetApp ONTAP system and an AWS-managed system virtual machine (SVM) that connects to it. In the provided diagram, the NetApp ONTAP server managed by AWS is located outside the VPC. The SVM serves as the intermediary between SageMaker and the NetApp ONTAP system, receiving operation requests from SageMaker and forwarding them to the underlying storage. To access FSxN, SageMaker must be placed within the same VPC as the FSxN deployment. This configuration ensures communication and data access between SageMaker and FSxN.

Data Access

In real-world scenarios, data scientists typically utilize the existing data stored in FSxN to build their machine learning models. However, for demonstration purposes, since the FSxN file system is initially empty after creation, it is necessary to manually upload the training data. This can be achieved by mounting FSxN as a volume to SageMaker. Once the file system is successfully mounted, you can upload your dataset to the mounted location, making it accessible for training your models within the SageMaker environment. This approach allows you to leverage the storage capacity and capabilities of FSxN while working with SageMaker for model development and training.

The data reading process involves configuring FSxN as a private S3 bucket. To learn the detailed configuration instructions, please refer to Part 1 - Integrating AWS FSx for NetApp ONTAP (FSxN) as a private S3 bucket into AWS SageMaker

Integration Overview



The workflow of using training data in FSxN to build a deep learning model in SageMaker can be summarized into three main steps: data loader definition, model training, and deployment. At a high level, these steps form the foundation of an MLOps pipeline. However, each step involves several detailed sub-steps for a comprehensive implementation. These sub-steps encompass various tasks such as data preprocessing, dataset splitting, model configuration, hyperparameter tuning, model evaluation, and model deployment. These steps ensure a thorough and effective process for building and deploying deep learning models using training data from FSxN within the SageMaker environment.

Step-by-Step Integration

Data Loader

In order to train a PyTorch deep learning network with data, a data loader is created to facilitate the feeding of data. The data loader not only defines the batch size but also determines the procedure for reading and preprocessing each record within the batch. By configuring the data loader, we can handle the processing of data in batches, enabling training of the deep learning network.

The data loader consists of 3 parts.

Preprocessing Function

```
from torchvision import transforms

preprocess = transforms.Compose([
    transforms.ToTensor(),
    transforms.Resize((224,224)),
    transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    )
])
```

The above code snippet demonstrates the definition of image preprocessing transformations using the **torchvision.transforms** module. In this turtorial, the preprocess object is created to apply a series of transformations. Firstly, the **ToTensor()** transformation converts the image into a tensor representation. Subsequently, the **Resize 224,224** transformation resizes the image to a fixed size of 224x224 pixels. Finally, the **Normalize()** transformation normalizes the tensor values by subtracting the mean and dividing by the standard deviation along each channel. The mean and standard deviation values used for normalization are

commonly employed in pre-trained neural network models. Overall, this code prepares the image data for further processing or input into a pre-trained model by converting it to a tensor, resizing it, and normalizing the pixel values.

The PyTorch Dataset Class

```
import torch
from io import BytesIO
from PIL import Image


class FSxNImageDataset(torch.utils.data.Dataset):
    def __init__(self, bucket, prefix='', preprocess=None):
        self.image_keys = [
            s3_obj.key
            for s3_obj in list(bucket.objects.filter(Prefix=prefix).all())
        ]
        self.preprocess = preprocess

    def __len__(self):
        return len(self.image_keys)

    def __getitem__(self, index):
        key = self.image_keys[index]
        response = bucket.Object(key)

        label = 1 if key[13:].startswith('defective') else 0

        image_bytes = response.get()['Body'].read()
        image = Image.open(BytesIO(image_bytes))
        if image.mode == 'L':
            image = image.convert('RGB')

        if self.preprocess is not None:
            image = self.preprocess(image)
        return image, label
```

This class provides functionality to obtain the total number of records in the dataset and defines the method for reading data for each record. Within the ***getitem*** function, the code utilizes the boto3 S3 bucket object to retrieve the binary data from FSxN. The code style for accessing data from FSxN is similar to reading data from Amazon S3. The subsequent explanation delves into the creation process of the private S3 object **bucket**.

FSxN as a private S3 repository

```
seed = 77                                                    # Random seed
bucket_name = '<Your ONTAP bucket name>'                     # The bucket
name in ONTAP
aws_access_key_id = '<Your ONTAP bucket key id>'             # Please get
this credential from ONTAP
aws_secret_access_key = '<Your ONTAP bucket access key>'     # Please get
this credential from ONTAP
fsx_endpoint_ip = '<Your FSxN IP address>'                   # Please get
this IP address from FSXN
```

```
import boto3

# Get session info
region_name = boto3.session.Session().region_name

# Initialize Fsxn S3 bucket object
# --- Start integrating SageMaker with FSXN ---
# This is the only code change we need to incorporate SageMaker with FSXN
s3_client: boto3.client = boto3.resource(
    's3',
    region_name=region_name,
    aws_access_key_id=aws_access_key_id,
    aws_secret_access_key=aws_secret_access_key,
    use_ssl=False,
    endpoint_url=f'http://{fsx_endpoint_ip}',
    config=boto3.session.Config(
        signature_version='s3v4',
        s3={'addressing_style': 'path'}
    )
)
# s3_client = boto3.resource('s3')
bucket = s3_client.Bucket(bucket_name)
# --- End integrating SageMaker with FSXN ---
```

To read data from FSxN in SageMaker, a handler is created that points to the FSxN storage using the S3 protocol. This allows FSxN to be treated as a private S3 bucket. The handler configuration includes specifying the IP address of the FSxN SVM, the bucket name, and the necessary credentials. For a comprehensive explanation on obtaining these configuration items, please refer to the document at Part 1 - Integrating AWS FSx for NetApp ONTAP (FSxN) as a private S3 bucket into AWS SageMaker.

In the example mentioned above, the bucket object is used to instantiate the PyTorch dataset object. The dataset object will be further explained in the subsequent section.

The PyTorch Data Loader

```
from torch.utils.data import DataLoader
torch.manual_seed(seed)

# 1. Hyperparameters
batch_size = 64

# 2. Preparing for the dataset
dataset = FSxNImageDataset(bucket, 'dataset/tyre', preprocess=preprocess)

train, test = torch.utils.data.random_split(dataset, [1500, 356])

data_loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

In the example provided, a batch size of 64 is specified, indicating that each batch will contain 64 records. By combining the PyTorch **Dataset** class, the preprocessing function, and the training batch size, we obtain the data loader for training. This data loader facilitates the process of iterating through the dataset in batches during the training phase.

Model Training

```
from torch import nn


class TyreQualityClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Conv2d(3,32,(3,3)),
            nn.ReLU(),
            nn.Conv2d(32,32,(3,3)),
            nn.ReLU(),
            nn.Conv2d(32,64,(3,3)),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(64*(224-6)*(224-6),2)
        )
    def forward(self, x):
        return self.model(x)
```

```
import datetime

num_epochs = 2
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model = TyreQualityClassifier()
fn_loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)


model.to(device)
for epoch in range(num_epochs):
    for idx, (X, y) in enumerate(data_loader):
        X = X.to(device)
        y = y.to(device)

        y_hat = model(X)

        loss = fn_loss(y_hat, y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        current_time = datetime.datetime.now().strftime("%Y-%m-%d
%H:%M:%S")
        print(f"Current Time: {current_time} - Epoch [{epoch+1}/
{num_epochs}]- Batch [{idx + 1}] - Loss: {loss}", end='\r')
```

This code implements a standard PyTorch training process. It defines a neural network model called **TyreQualityClassifier** using convolutional layers and a linear layer to classify tire quality. The training loop iterates over data batches, computes the loss, and updates the model's parameters using backpropagation and optimization. Additionally, it prints the current time, epoch, batch, and loss for monitoring purposes.

Model Deployment

Deployment

```
import io
import os
import tarfile
import sagemaker

# 1. Save the PyTorch model to memory
buffer_model = io.BytesIO()
traced_model = torch.jit.script(model)
torch.jit.save(traced_model, buffer_model)

# 2. Upload to AWS S3
sagemaker_session = sagemaker.Session()
bucket_name_default = sagemaker_session.default_bucket()
model_name = f'tyre_quality_classifier.pth'

# 2.1. Zip PyTorch model into tar.gz file
buffer_zip = io.BytesIO()
with tarfile.open(fileobj=buffer_zip, mode="w:gz") as tar:
    # Add PyTorch pt file
    file_name = os.path.basename(model_name)
    file_name_with_extension = os.path.split(file_name)[-1]
    tarinfo = tarfile.TarInfo(file_name_with_extension)
    tarinfo.size = len(buffer_model.getbuffer())
    buffer_model.seek(0)
    tar.addfile(tarinfo, buffer_model)

# 2.2. Upload the tar.gz file to S3 bucket
buffer_zip.seek(0)
boto3.resource('s3') \
    .Bucket(bucket_name_default) \
    .Object(f'pytorch/{model_name}.tar.gz') \
    .put(Body=buffer_zip.getvalue())
```

The code saves the PyTorch model to **Amazon S3** because SageMaker requires the model to be stored in S3 for deployment. By uploading the model to **Amazon S3**, it becomes accessible to SageMaker, allowing for the deployment and inference on the deployed model.

```
import time
from sagemaker.pytorch import PyTorchModel
from sagemaker.predictor import Predictor
from sagemaker.serializers import IdentitySerializer
from sagemaker.deserializers import JSONDeserializer


class TyreQualitySerializer(IdentitySerializer):
```

```
    CONTENT_TYPE = 'application/x-torch'

    def serialize(self, data):
        transformed_image = preprocess(data)
        tensor_image = torch.Tensor(transformed_image)

        serialized_data = io.BytesIO()
        torch.save(tensor_image, serialized_data)
        serialized_data.seek(0)
        serialized_data = serialized_data.read()

        return serialized_data


class TyreQualityPredictor(Predictor):
    def __init__(self, endpoint_name, sagemaker_session):
        super().__init__(
            endpoint_name,
            sagemaker_session=sagemaker_session,
            serializer=TyreQualitySerializer(),
            deserializer=JSONDeserializer(),
        )

sagemaker_model = PyTorchModel(
    model_data=f's3://{bucket_name_default}/pytorch/{model_name}.tar.gz',
    role=sagemaker.get_execution_role(),
    framework_version='2.0.1',
    py_version='py310',
    predictor_cls=TyreQualityPredictor,
    entry_point='inference.py',
    source_dir='code',
)

timestamp = int(time.time())
pytorch_endpoint_name = '{}-{}-{}'.format('tyre-quality-classifier', 'pt',
timestamp)
sagemaker_predictor = sagemaker_model.deploy(
    initial_instance_count=1,
    instance_type='ml.p3.2xlarge',
    endpoint_name=pytorch_endpoint_name
)
```

This code facilitates the deployment of a PyTorch model on SageMaker. It defines a custom serializer, **TyreQualitySerializer**, which preprocesses and serializes input data as a PyTorch tensor. The **TyreQualityPredictor** class is a custom predictor that utilizes the defined serializer and a **JSONDeserializer**. The code also creates a **PyTorchModel** object to specify the model's S3 location, IAM role, framework version, and entry point for inference. The code generates a timestamp and constructs an endpoint name based on the

model and timestamp. Finally, the model is deployed using the deploy method, specifying the instance count, instance type, and generated endpoint name. This enables the PyTorch model to be deployed and accessible for inference on SageMaker.

Inference

```
image_object = list(bucket.objects.filter('dataset/tyre'))[0].get()
image_bytes = image_object['Body'].read()

with Image.open(with Image.open(BytesIO(image_bytes)) as image:
    predicted_classes = sagemaker_predictor.predict(image)

    print(predicted_classes)
```

This is the example of using the deployed endpoint to do the inference.

Part 3 - Building A Simplified MLOps Pipeline (CI/CT/CD)

Author(s):
Jian Jian (Ken), Senior Data & Applied Scientist, NetApp

Introduction

In this tutorial, you will learn how to leverage various AWS services to construct a simple MLOps pipeline that encompasses Continuous Integration (CI), Continuous Training (CT), and Continuous Deployment (CD). Unlike traditional DevOps pipelines, MLOps requires additional considerations to complete the operational cycle. By following this tutorial, you will gain insights into incorporating CT into the MLOps loop, enabling continuous training of your models and seamless deployment for inference. The tutorial will guide you through the process of utilizing AWS services to establish this end-to-end MLOps pipeline.

Manifest

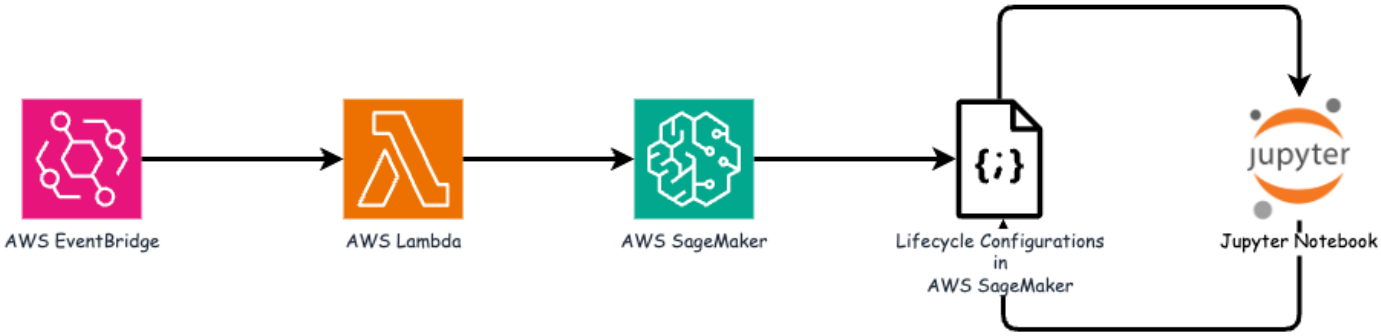
Functionality	Name	Comment
Data storage	AWS FSxN	Refer to Part 1 - Integrating AWS FSx for NetApp ONTAP (FSxN) as a private S3 bucket into AWS SageMaker.
Data science IDE	AWS SageMaker	This tutorial is based on the Jupyter notebook presented in Part 2 - Leveraging AWS FSx for NetApp ONTAP (FSxN) as a Data Source for Model Training in SageMaker.
Function to trigger the MLOps pipeline	AWS Lambda function	-
Cron job trigger	AWS EventBridge	-
Deep learning framework	PyTorch	-

Functionality	Name	Comment
AWS Python SDK	boto3	-
Programming Language	Python	v3.10

Prerequisite

- An pre-configured FSxN file system. This tutorial utilizes data stored in FSxN for the training process.
- A **SageMaker Notebook instance** that is configured to share the same VPC as the FSxN file system mentioned above.
- Before triggering the **AWS Lambda function**, ensure that the **SageMaker Notebook instance** is in **stopped** status.
- The **ml.g4dn.xlarge** instance type is required to leverage the GPU acceleration necessary for the computations of deep neural networks.

Architecture



This MLOps pipeline is a practical implementation that utilizes a cron job to trigger a serverless function, which in turn executes an AWS service registered with a lifecycle callback function. The **AWS EventBridge** acts as the cron job. It periodically invokes an **AWS Lambda function** responsible for retraining and redeploying the model. This process involves spinning up the **AWS SageMaker Notebook** instance to perform the necessary tasks.

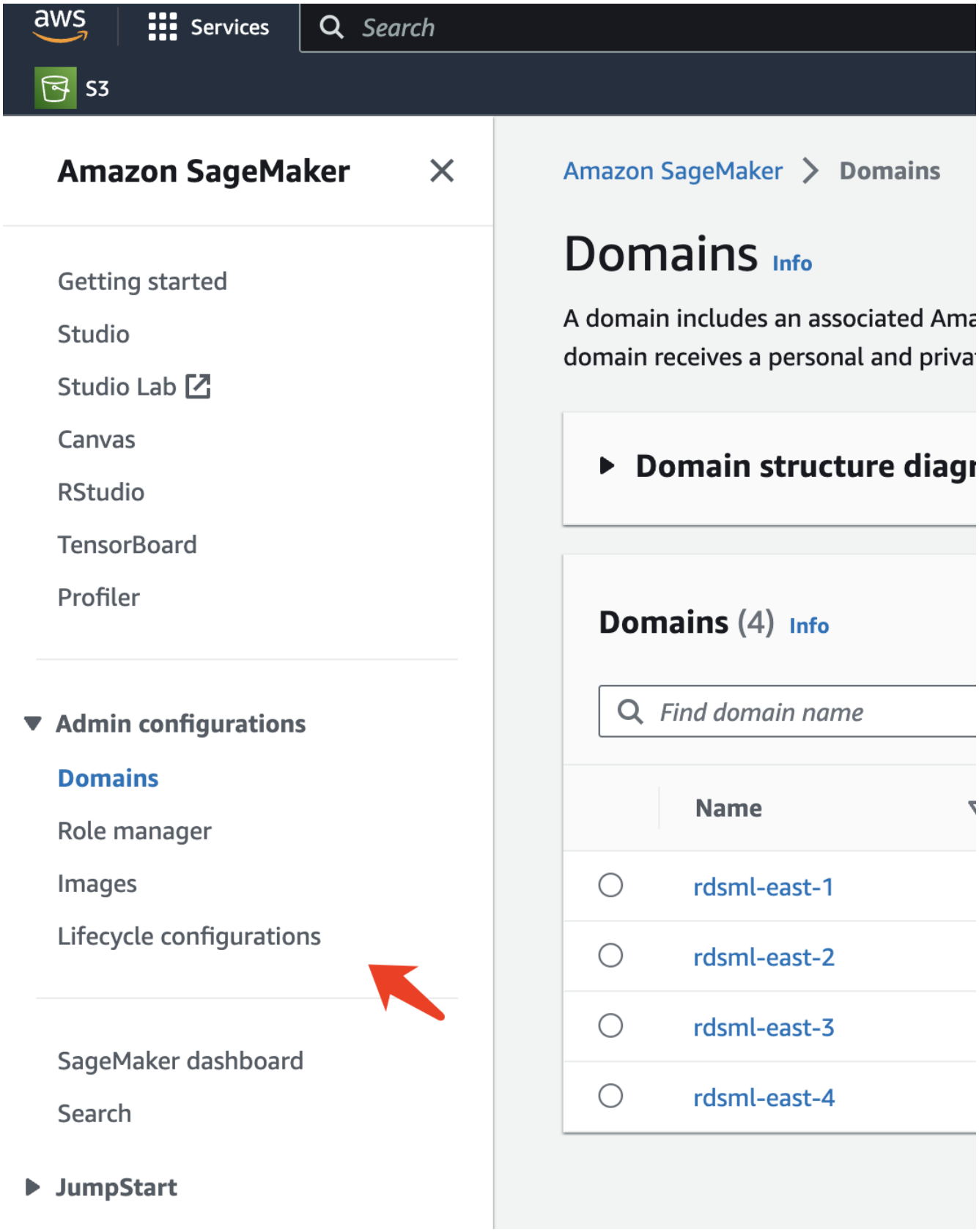
Step-by-Step Configuration

Lifecycle configurations

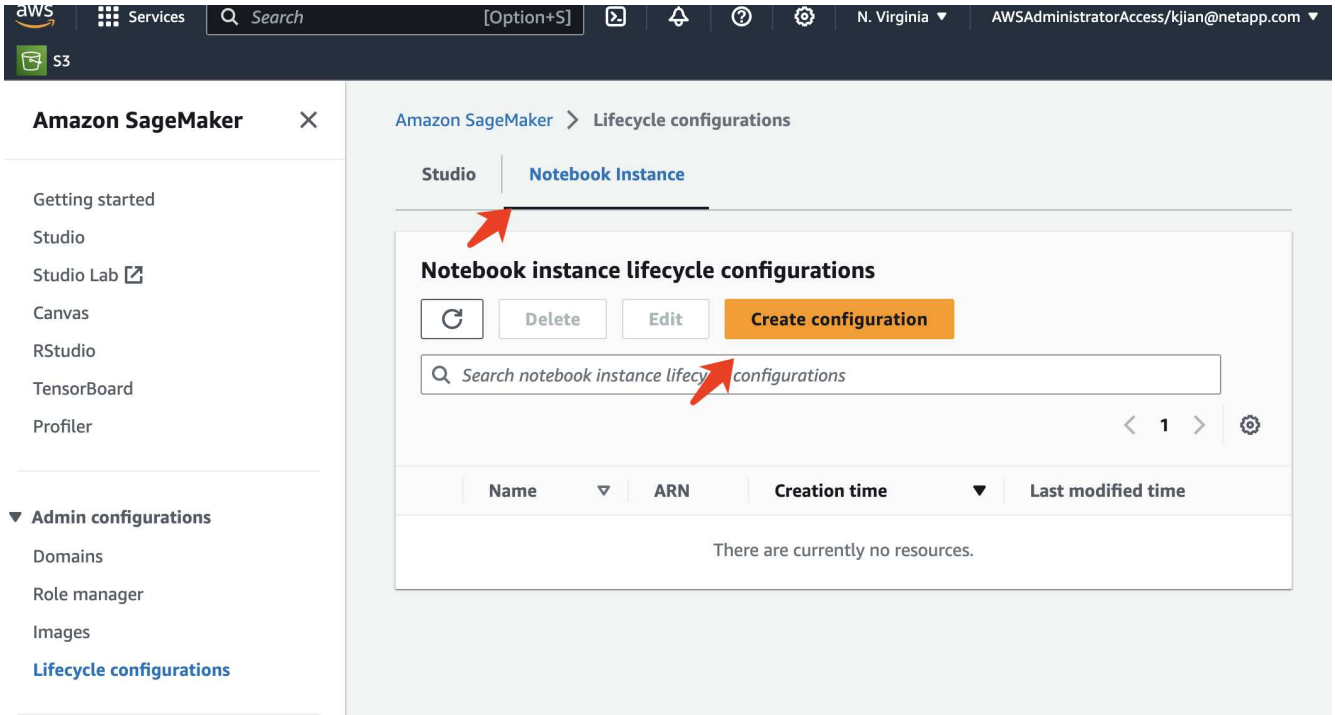
To configure the lifecycle callback function for the AWS SageMaker Notebook instance, you would utilize **Lifecycle configurations**. This service allow you to define the necessary actions to be performed during when spinning up the notebook instance. Specifically, a shell script can be implemented within the **Lifecycle configurations** to automatically shut down the notebook instance once the training and deployment processes are completed. This is a required configuration as the cost is one of the major consideration in MLOps.

It's important to note that the configuration for **Lifecycle configurations** needs to be set up in advance. Therefore, it is recommended to prioritize configuring this aspect before proceeding with the other MLOps pipeline setup.

1. To set up a Lifecycle configurations, open the **Sagemaker** panel and navigate to **Lifecycle configurations** under the section **Admin configurations**.



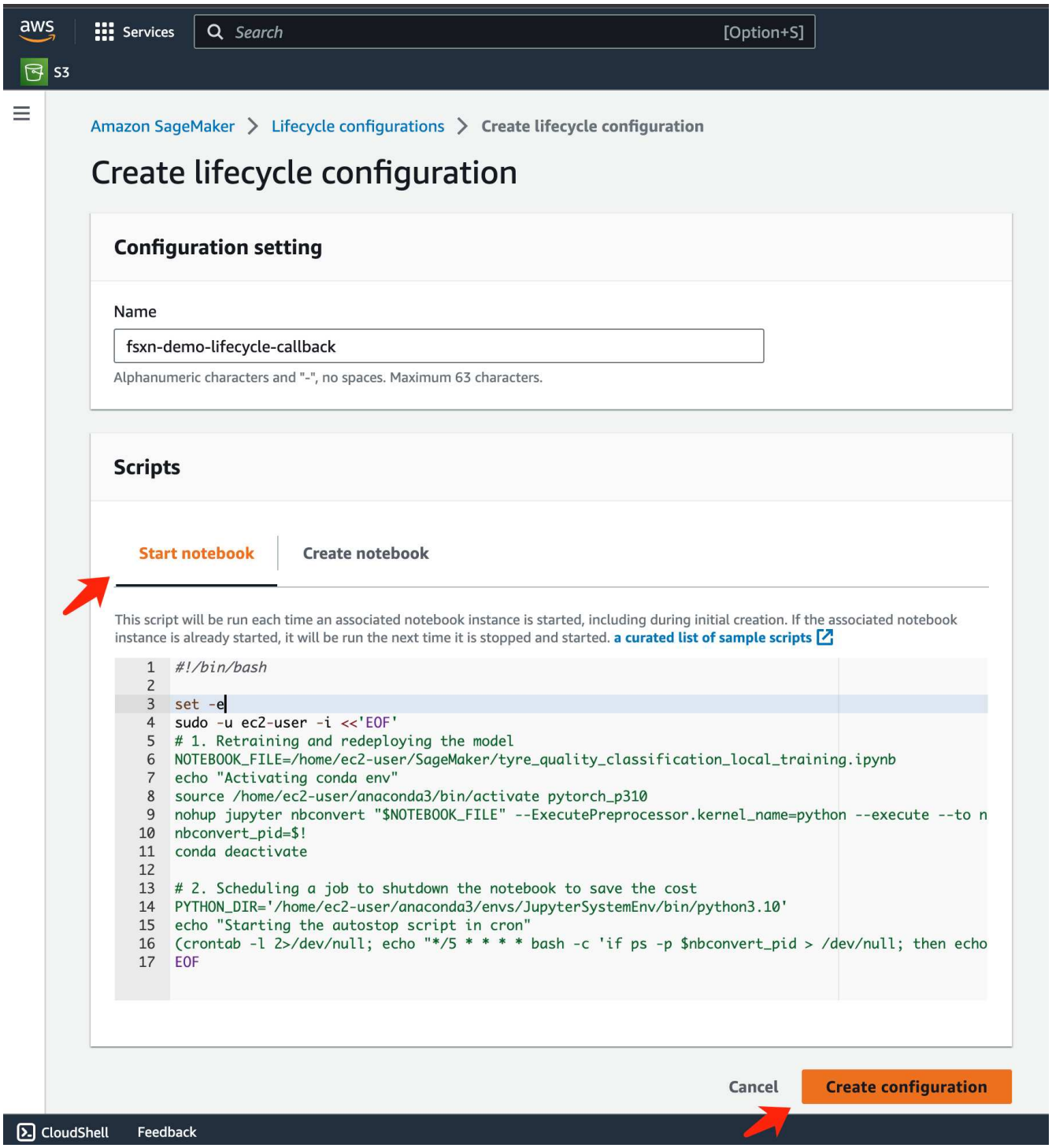
2. Select the **Notebook Instance** tab and click the **Create configuration** button



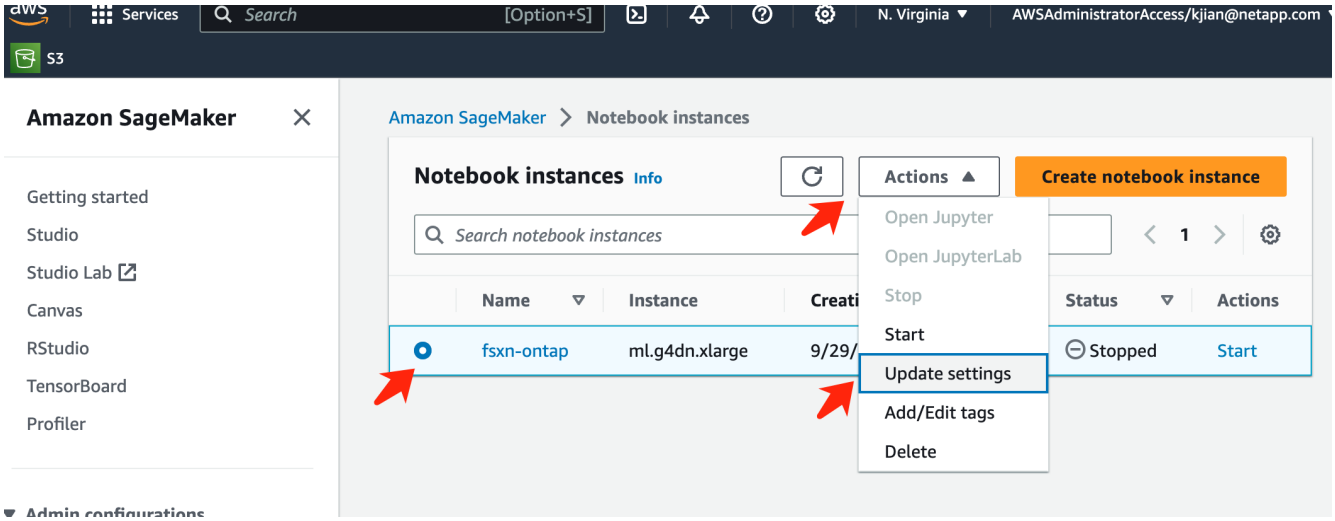
3. Paste the below code to the entry area.

```
#!/bin/bash

set -e
sudo -u ec2-user -i <<'EOF'
# 1. Retraining and redeploying the model
NOTEBOOK_FILE=/home/ec2-
user/SageMaker/tyre_quality_classification_local_training.ipynb
echo "Activating conda env"
source /home/ec2-user/anaconda3/bin/activate pytorch_p310
nohup jupyter nbconvert "$NOTEBOOK_FILE"
--ExecutePreprocessor.kernel_name=python --execute --to notebook &
nbconvert_pid=$!
conda deactivate

# 2. Scheduling a job to shutdown the notebook to save the cost
PYTHON_DIR='/home/ec2-
user/anaconda3/envs/JupyterSystemEnv/bin/python3.10'
echo "Starting the autostop script in cron"
(crontab -l 2>/dev/null; echo "*/5 * * * * bash -c 'if ps -p
$nbconvert_pid > /dev/null; then echo \"Notebook is still running.\" >>
/var/log/jupyter.log; else echo \"Notebook execution completed.\" >>
/var/log/jupyter.log; $PYTHON_DIR -c \"import boto3;boto3.client(
\'sagemaker\').stop_notebook_instance(NotebookInstanceName=get_notebook_
name())\" >> /var/log/jupyter.log; fi'") | crontab -
EOF
```

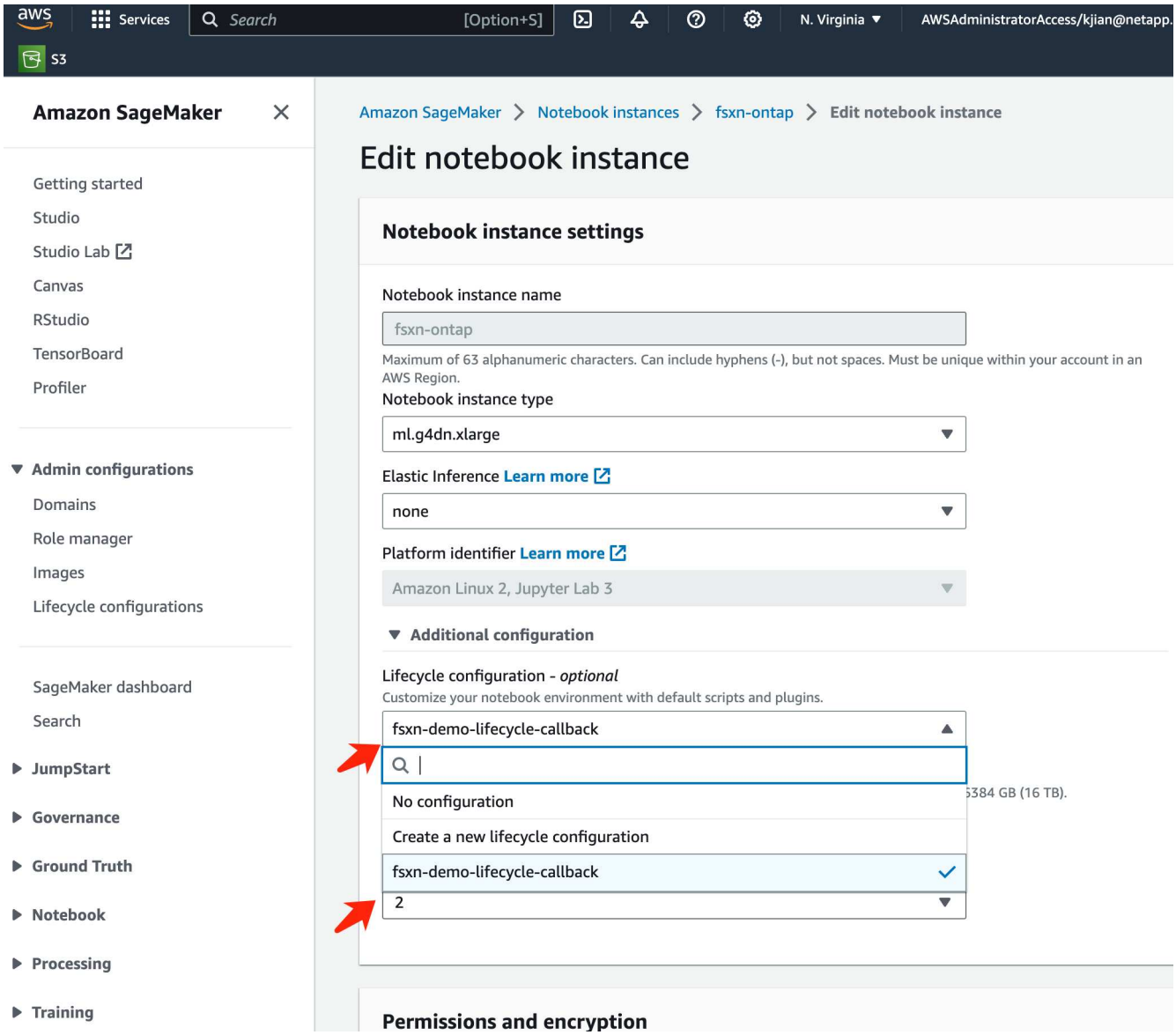
4. This script executes the Jupyter Notebook, which handles the retraining and redeployment of the model for inference. After the execution is complete, the notebook will automatically shut down within 5 minutes. To learn more about the problem statement and the code implementation, please refer to Part 2 - Leveraging AWS FSx for NetApp ONTAP (FSxN) as a Data Source for Model Training in SageMaker.



5. After the creation, navigate to Notebook instances, select the target instance, and click **Update settings** under Actions dropdown.



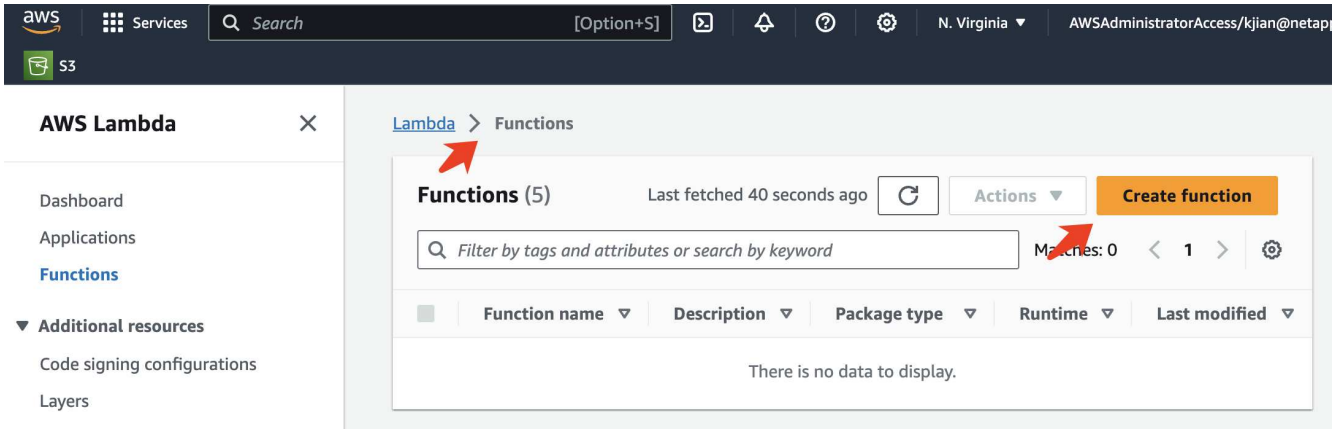
6. Select the created **Lifecycle configuration** and click **Update notebook instance**.



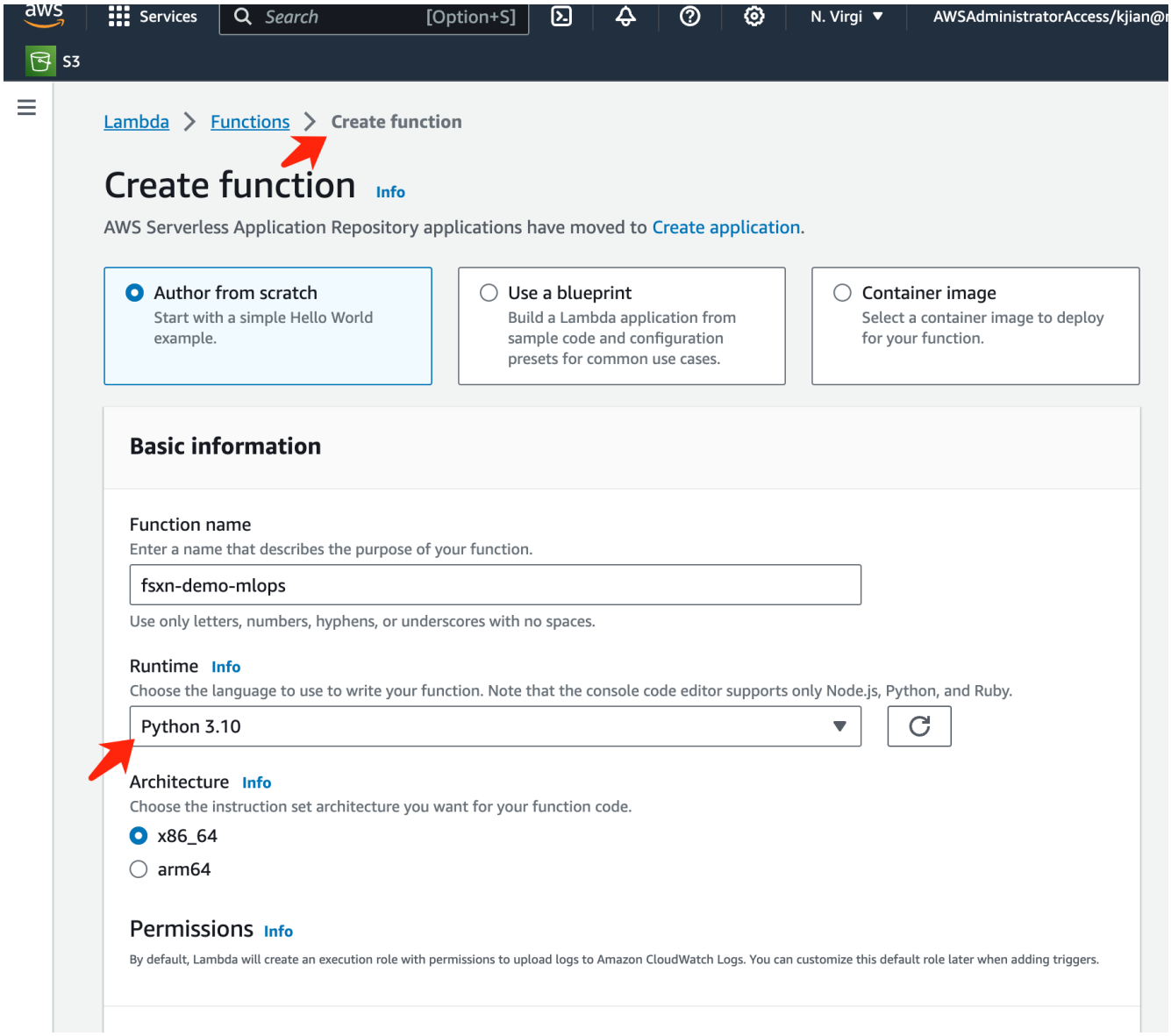
AWS Lambda serverless function

As mentioned earlier, the **AWS Lambda function** is responsible for spinning up the **AWS SageMaker Notebook instance**.

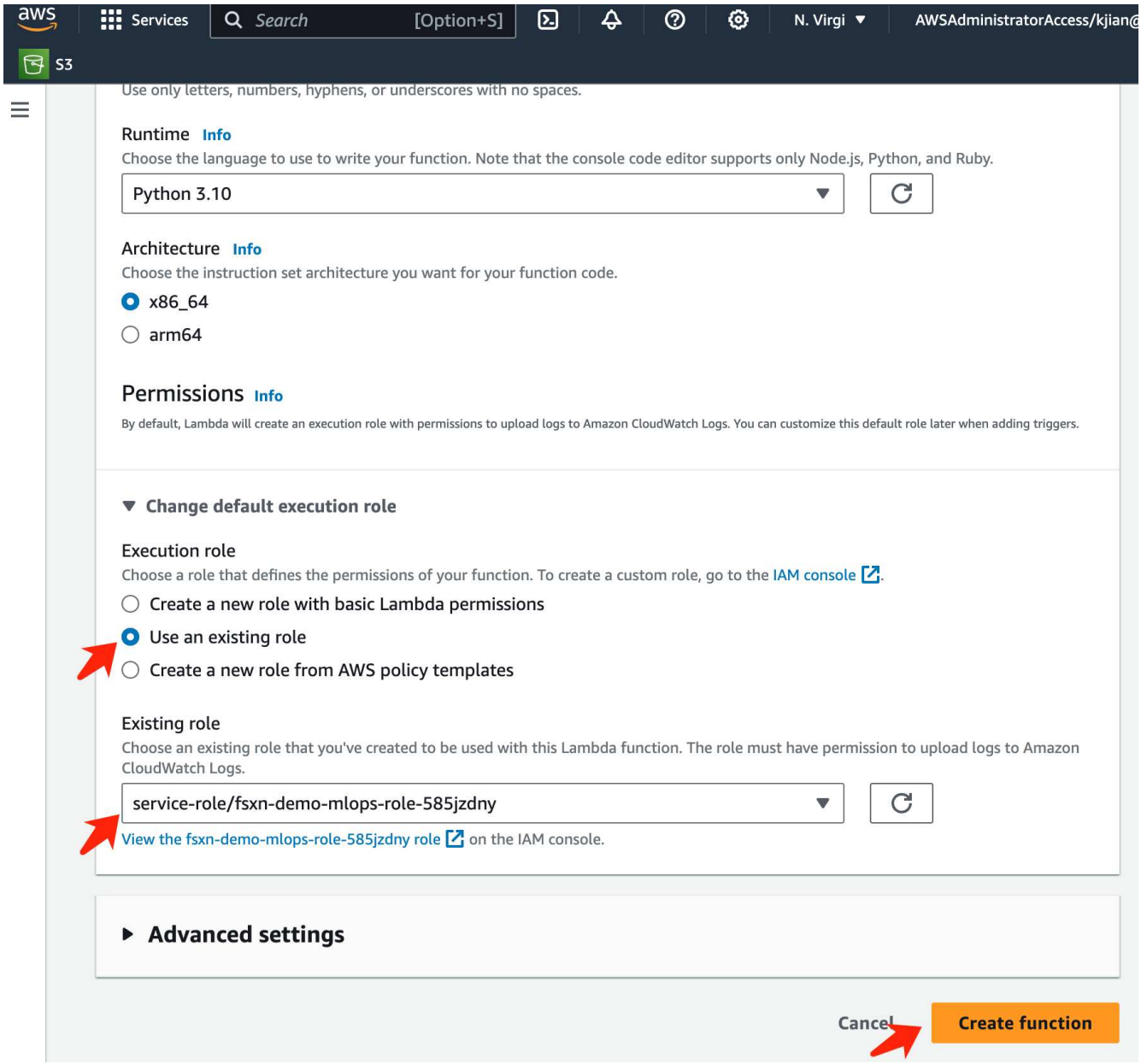
1. To create an **AWS Lambda function**, navigate to the respective panel, switch to the **Functions** tab, and click on **Create Function**.



2. Please file all required entries on the page and remember to switch the Runtime to **Python 3.10**.



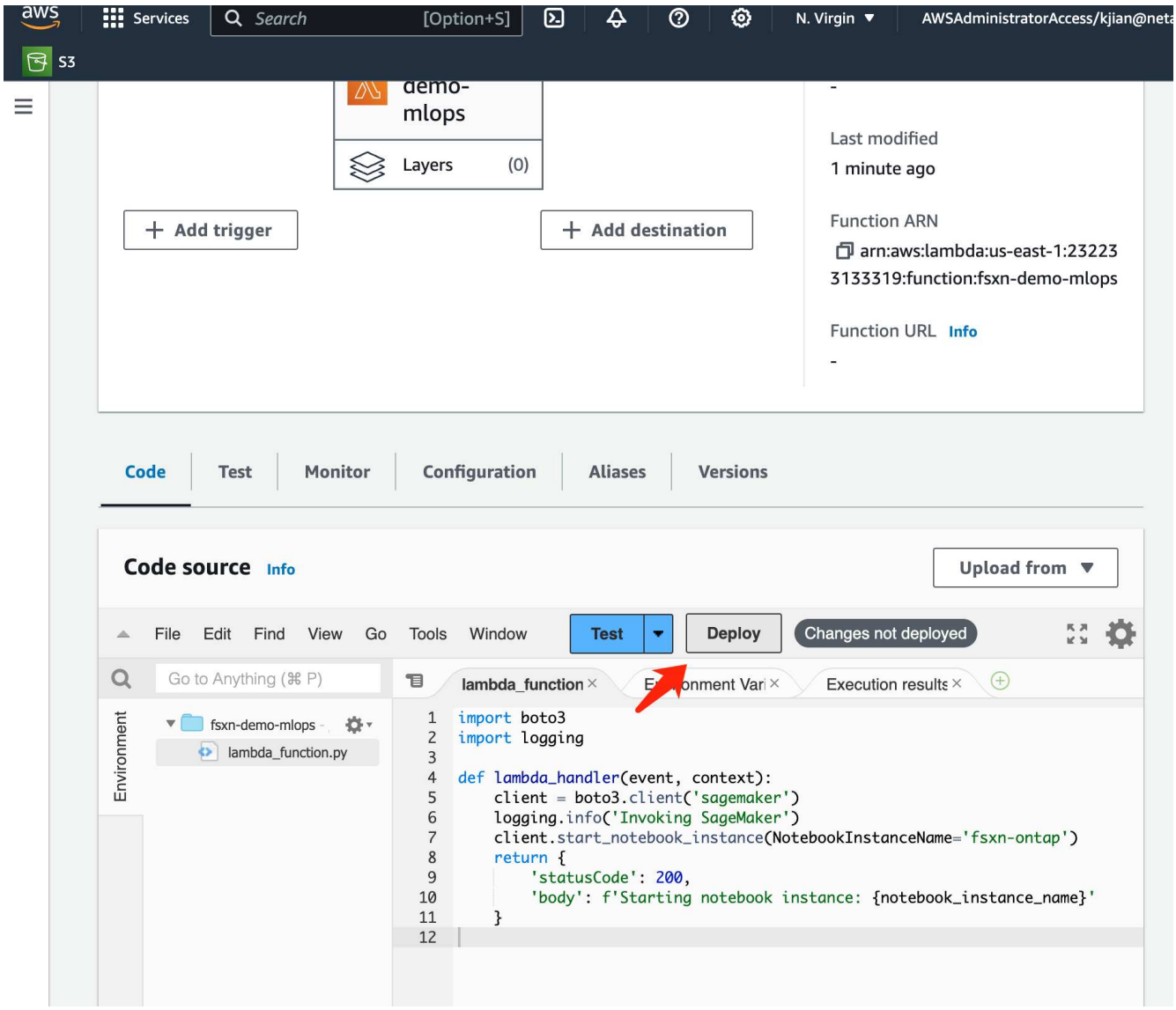
3. Please verify that the designated role has the required permission **AmazonSageMakerFullAccess** and click on the **Create function** button.



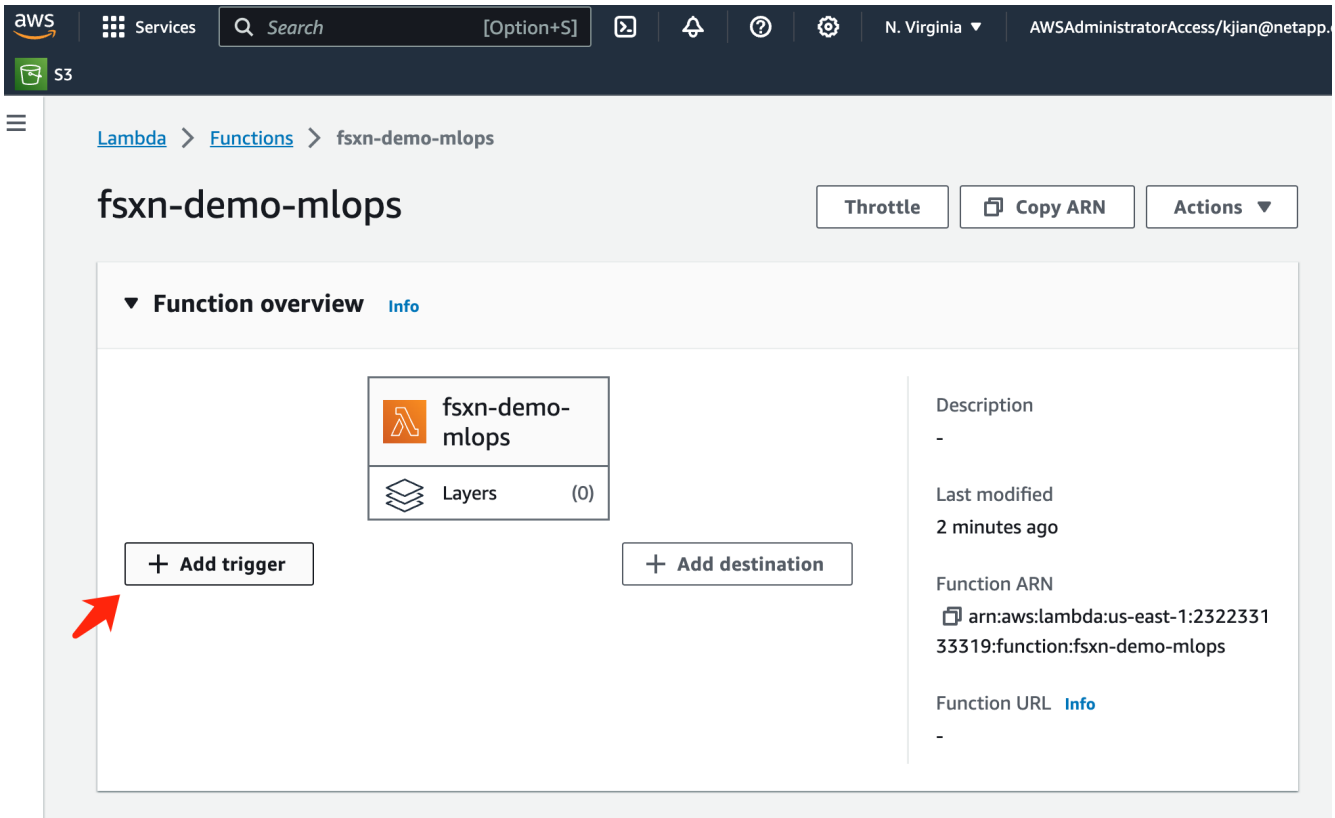
4. Select the created Lambda function. In the code tab, copy and paste the following code into the text area. This code starts the notebook instance named **fsxn-ontap**.

```
import boto3
import logging

def lambda_handler(event, context):
    client = boto3.client('sagemaker')
    logging.info('Invoking SageMaker')
    client.start_notebook_instance(NotebookInstanceName='fsxn-ontap')
    return {
        'statusCode': 200,
        'body': f'Starting notebook instance: {notebook_instance_name}'
    }
```

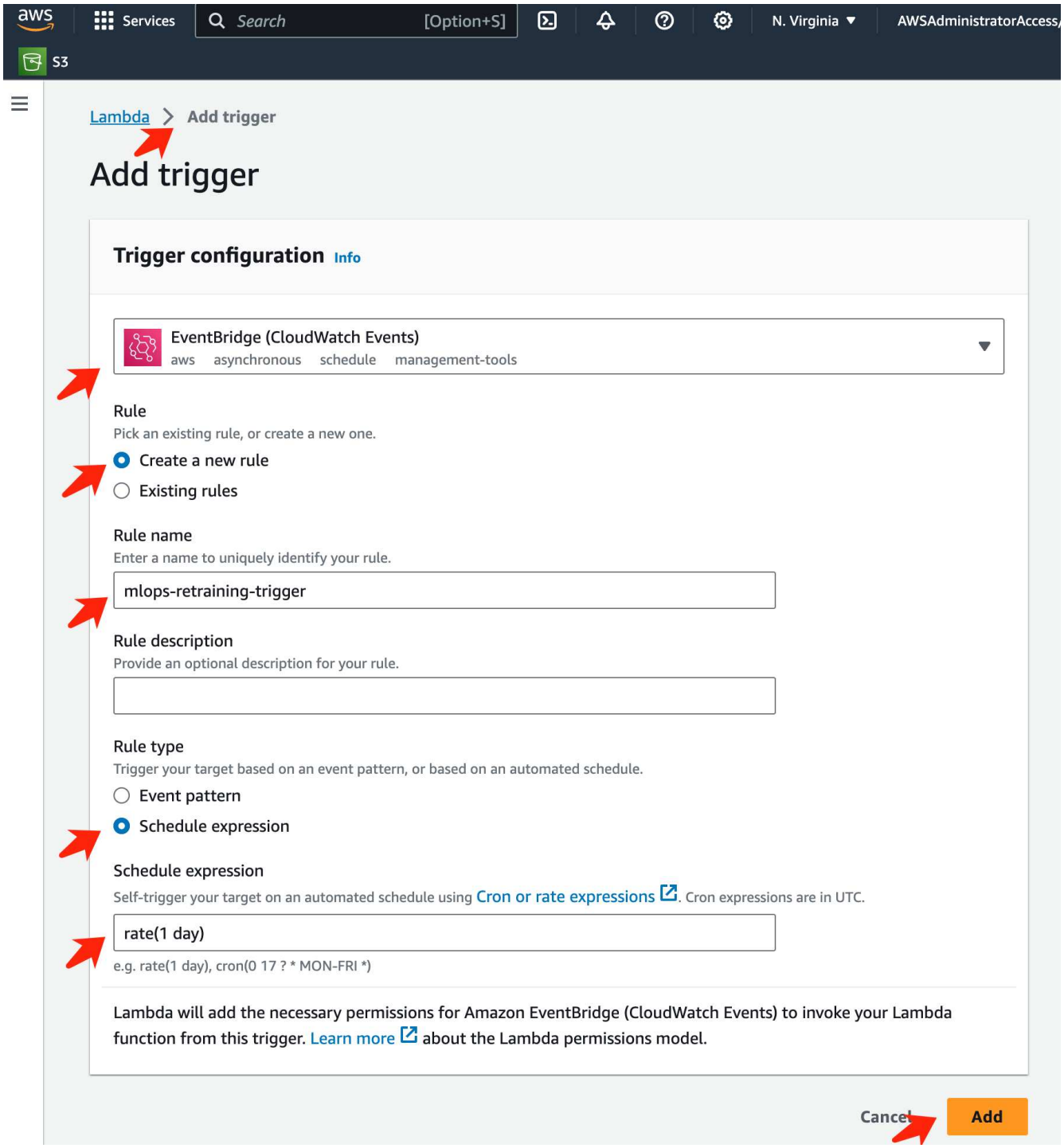
5. Click the **Deploy** button to apply this code change.



6. To specify how to trigger this AWS Lambda function, click on the Add Trigger button.



7. Select EventBridge from the dropdown menu, then click on the radio button labeled Create a new rule. In the schedule expression field, enter `rate(1 day)`, and click on the Add button to create and apply this new cron job rule to the AWS Lambda function.



After completing the two-step configuration, on a daily basis, the **AWS Lambda function** will initiate the **SageMaker Notebook**, perform model retraining using the data from the **FSxN** repository, redeploy the updated model to the production environment, and automatically shut down the **SageMaker Notebook instance** to optimize cost. This ensures that the model remains up to date.

This concludes the tutorial for developing an MLOps pipeline.