Utilizing Amazon S3 Triggers and Lambda Functions for Automated Data Processing and Storage

This document provides a comprehensive guide on leveraging the Amazon S3 trigger mechanism to seamlessly invoke a Lambda function, enabling the efficient processing of newly ingested data from the input bucket. The processed data is subsequently stored in the designated output bucket, ensuring a streamlined and automated data workflow. This integration allows for the transformation and enrichment of data as it traverses through the AWS ecosystem, facilitating seamless data management and enhancing overall system efficiency.

Steps Involved

To implement this solution, complete the following high-level steps:

1. Creating S3 buckets
2. Creating read-and-write policy
3. Creating lambda-specific roles and attaching the policy
4. Creating lambda function and setting trigger

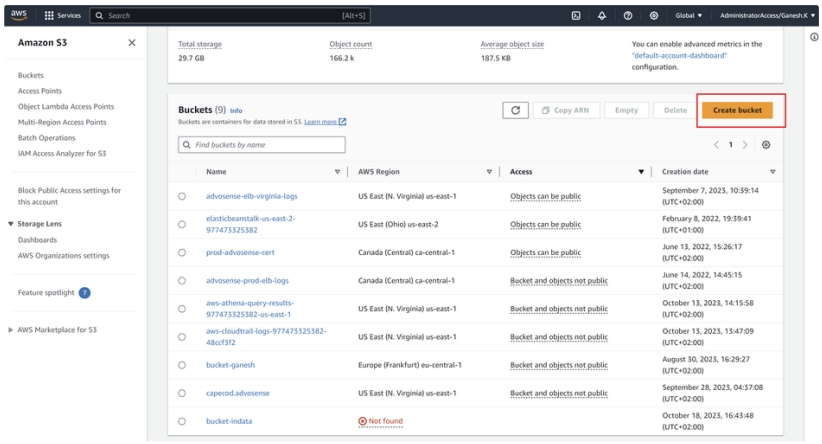
👀 Prerequisite

Not expertise level is required at least first-hand experience so that the following steps are ease to understand.

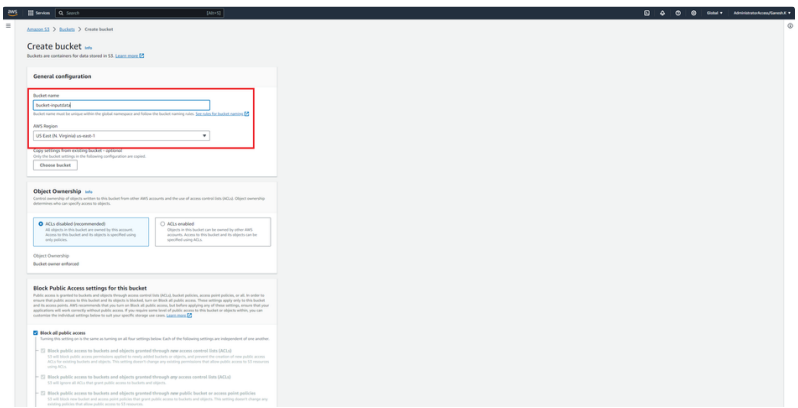
1. Python & pandas library
2. AWS

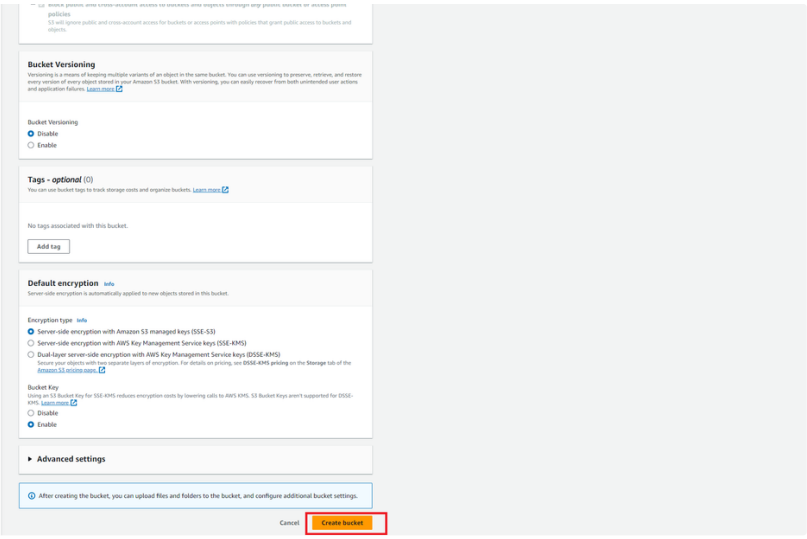
Step 1: Creating S3 buckets

1. Open the Amazon S3 console and select the **Buckets** page

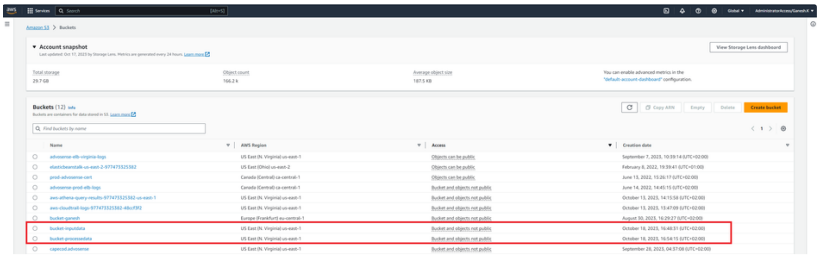


2. Choose **Create bucket** to load the input data.
3. Under **General configuration**, do the following:
   a. For the **Bucket name**, enter a globally unique name that meets the Amazon S3 Bucket naming rules. Bucket names can contain only lowercase letters, numbers, dots (.), and hyphens (-).
   b. For **AWS Region**, choose a Region. Later in the tutorial, you must create your Lambda function in the same Region.



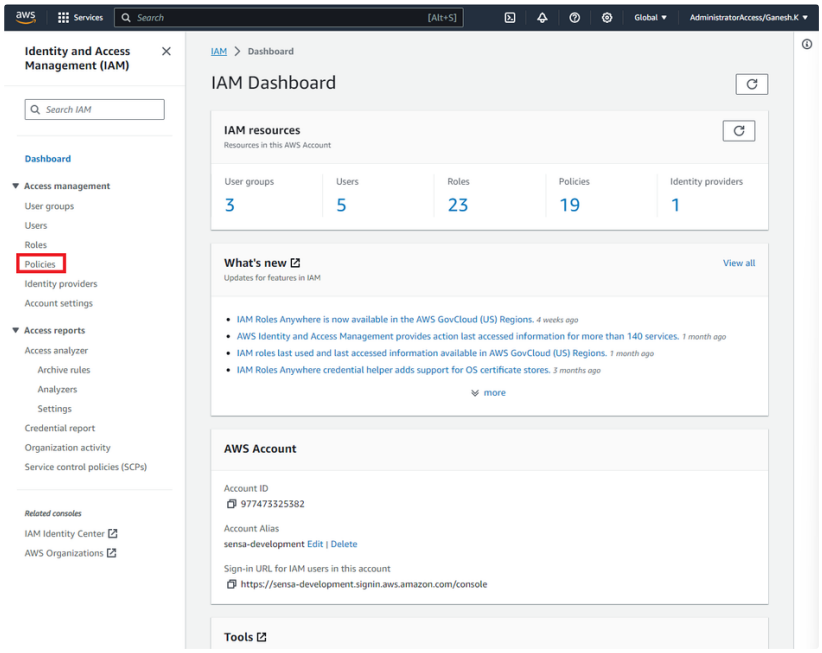


4. Leave all other options set to their default values and choose **Create bucket**.
5. Create **another bucket** to store the processed data as per the above procedure
6. Should have **two buckets** to store one for input data and one for processed data

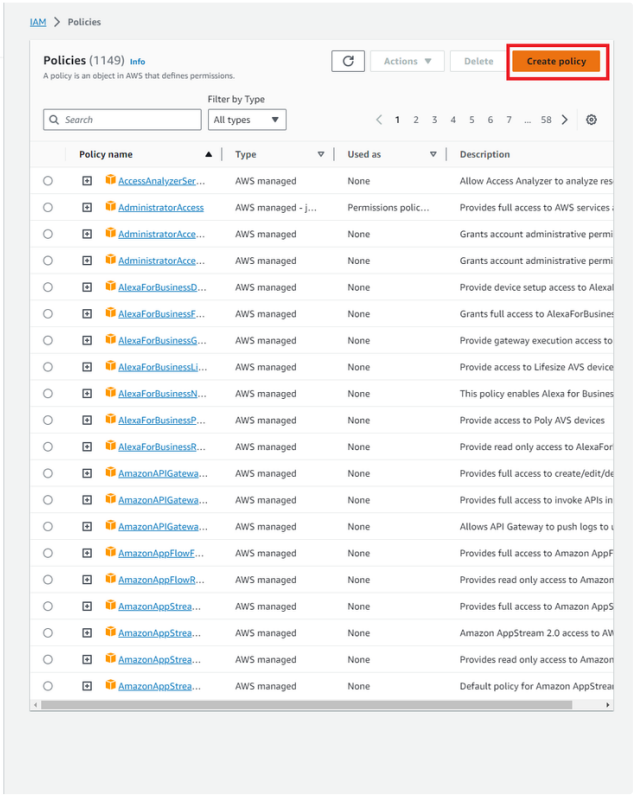


Step 2: Creating a read-and-write policy

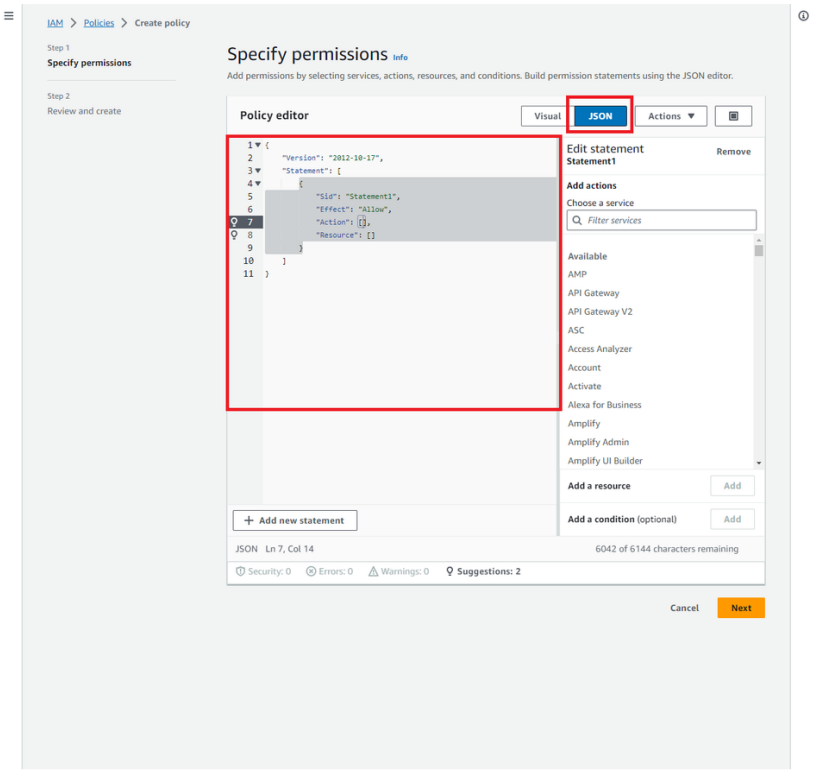
1. Open the **Management console** and select the **IAM** page
2. On the **IAM** page select **policies**



3. Choose to **create a policy**



4. Select **JSON**

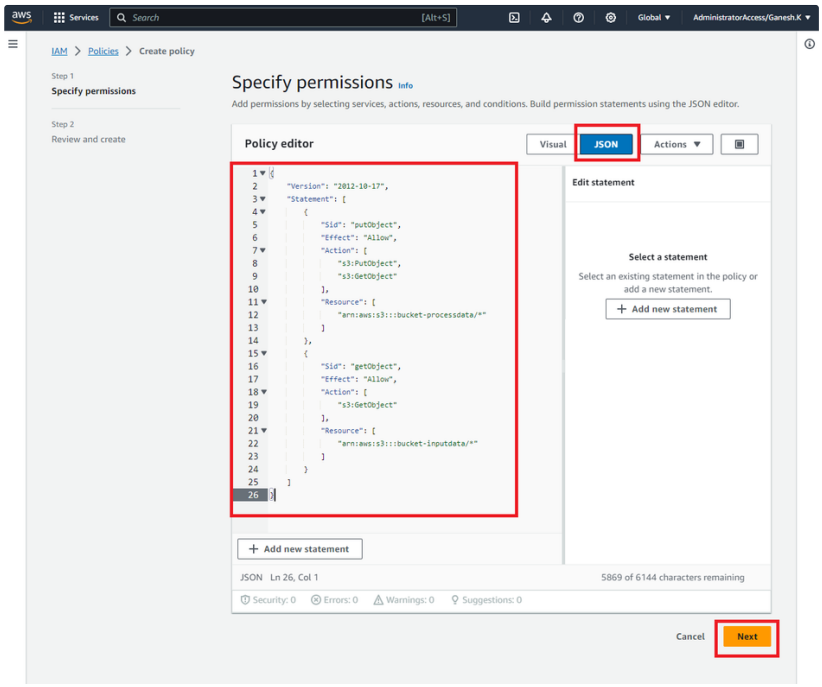


5. Paste the following code in the **policy editor field**

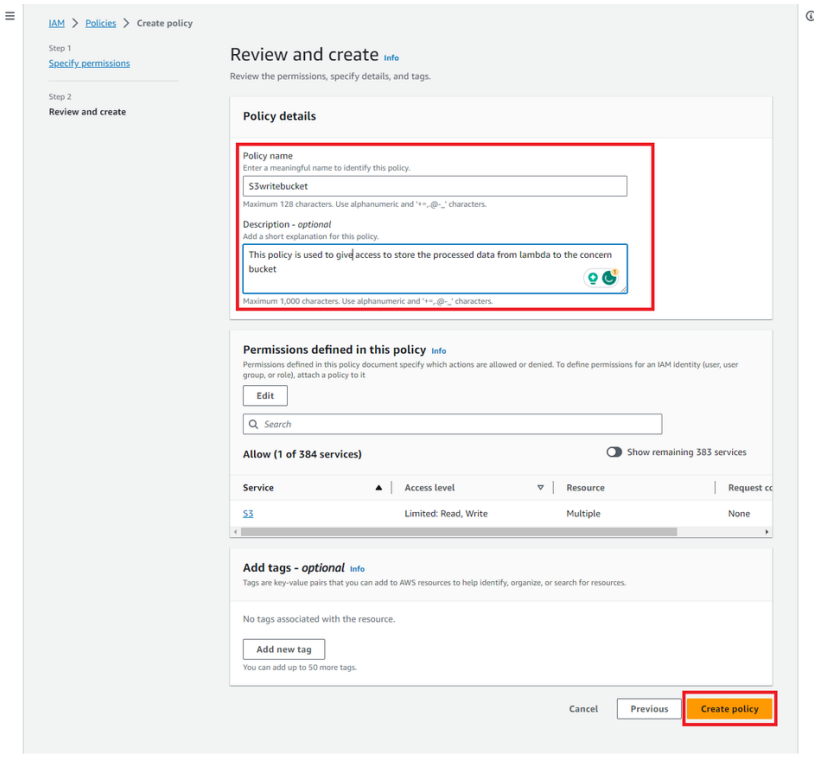
```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "putObject",
```

```
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::bucket-processedata/*"
            ]
        },
        {
            "Sid": "getObject",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::bucket-inputdata/*"
            ]
        }
    ]
}

```

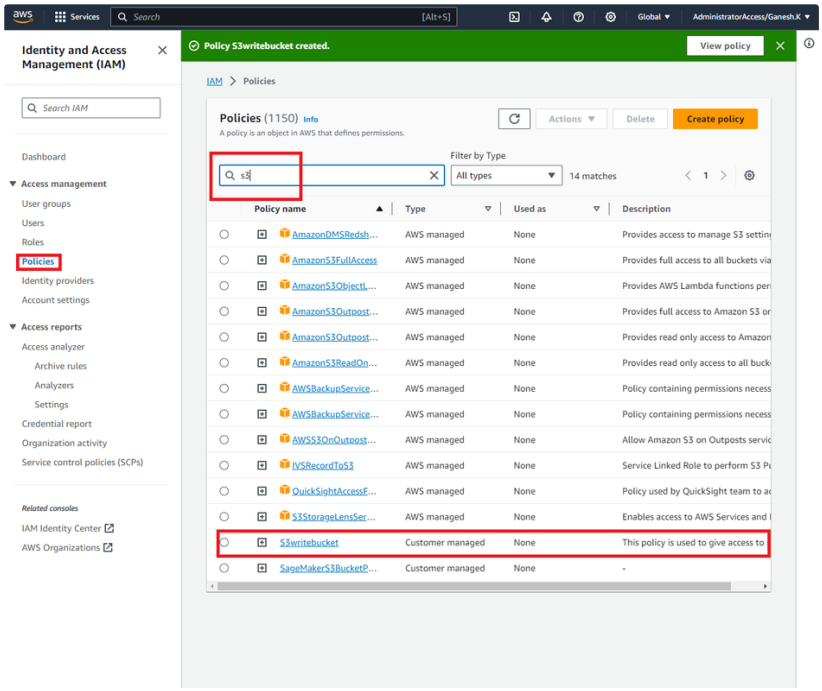
6. Select **Next**



7. In the next window, give the **policy name and description**, then select **Create policy.** In our case the name of the policy is **S3writebucket**

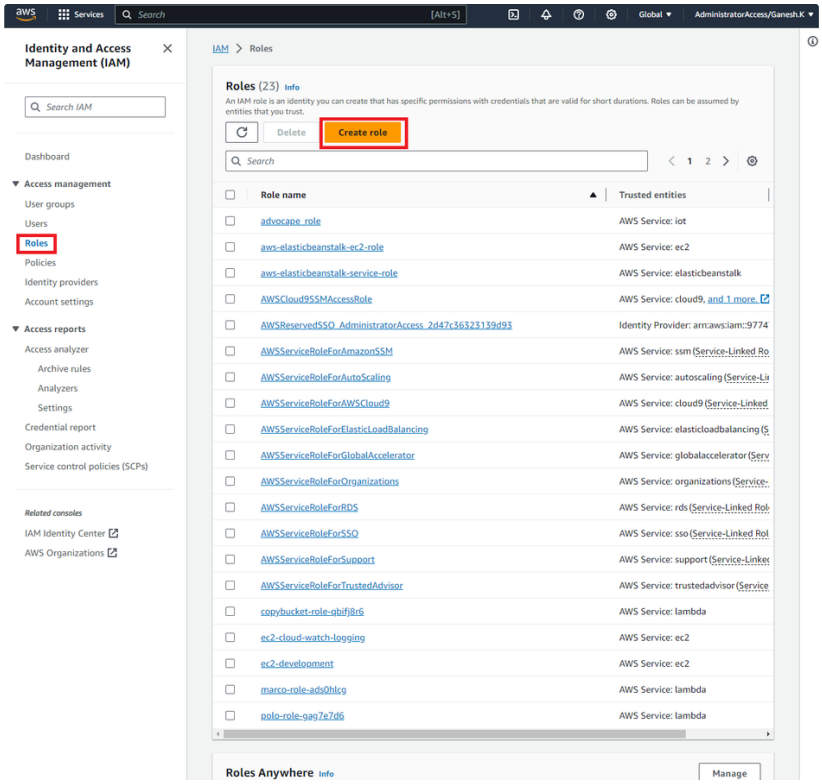


8. Once created, verify by searching the name in the field as shown below

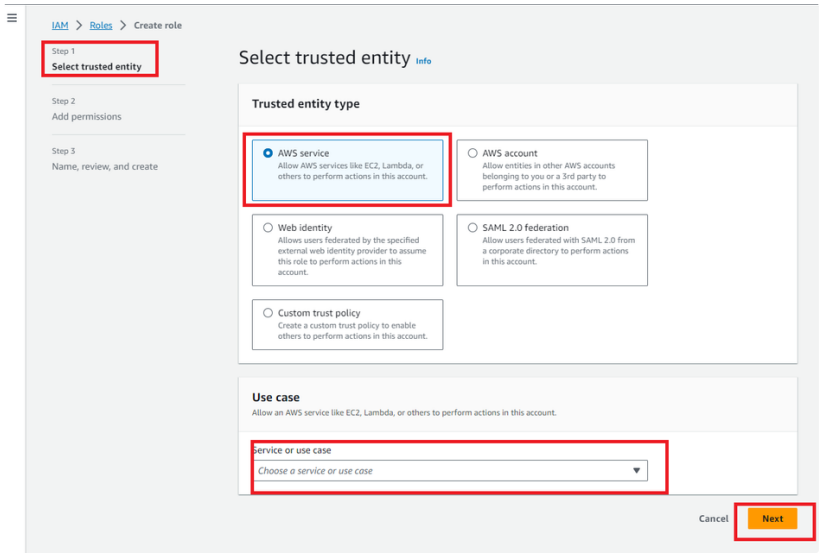


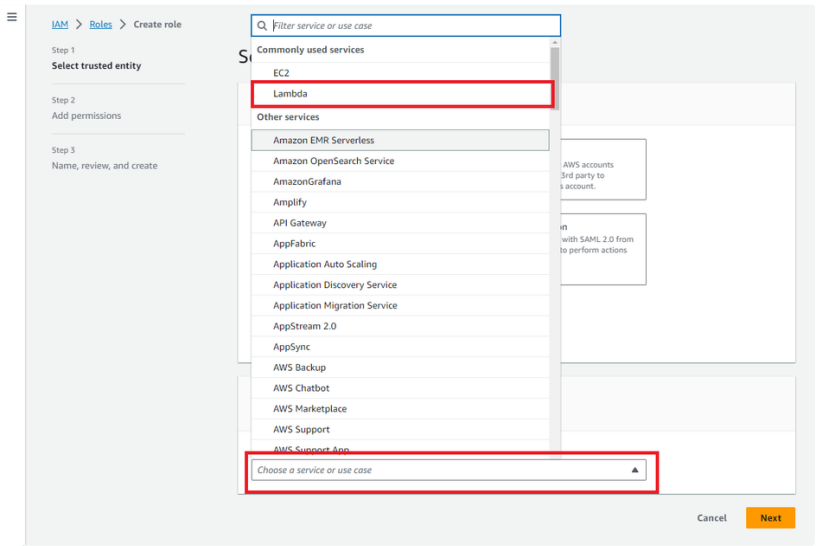
Step 3: Creating lambda-specific roles and attaching the policy

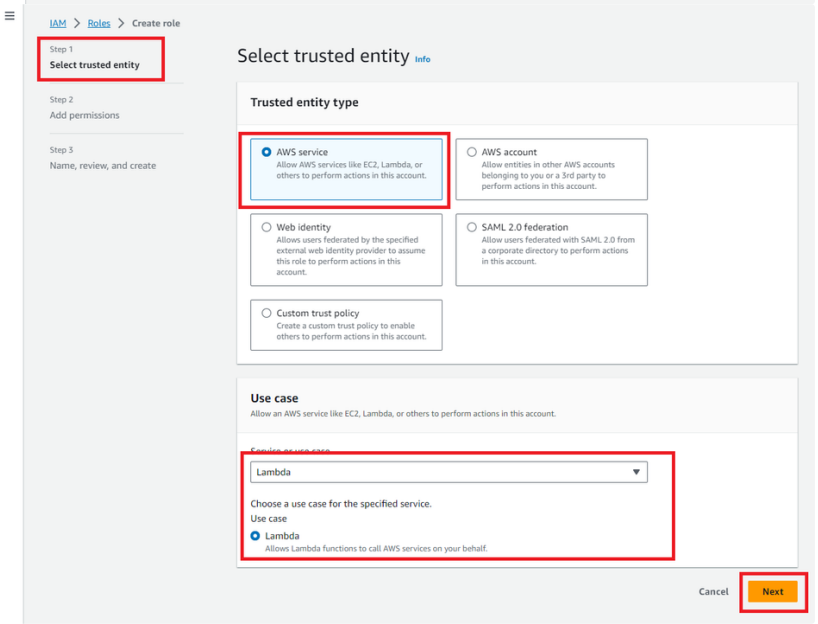
1. On the same **IAM** page, select **Roles**
2. Select **Create Role**



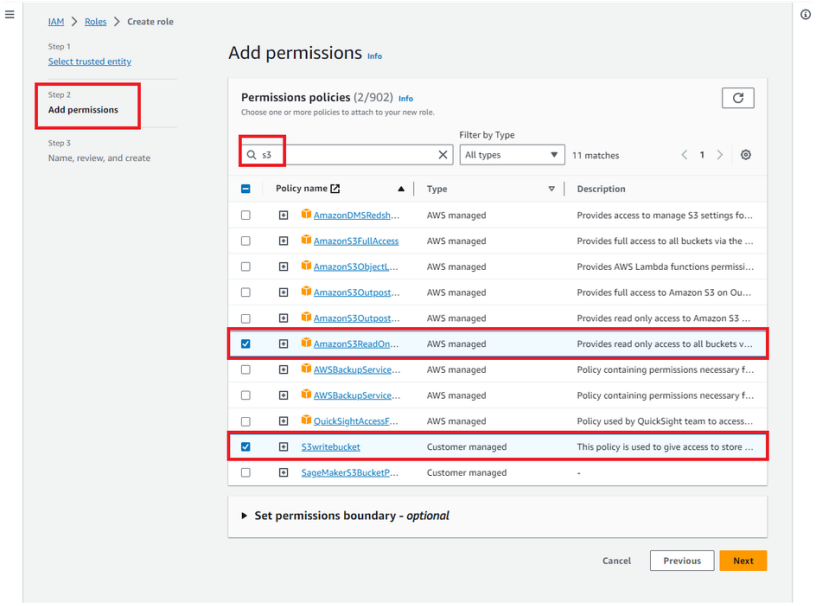
3. In the create role page, select **AWS service** and **choose service or use case as lambda,** then select **next**

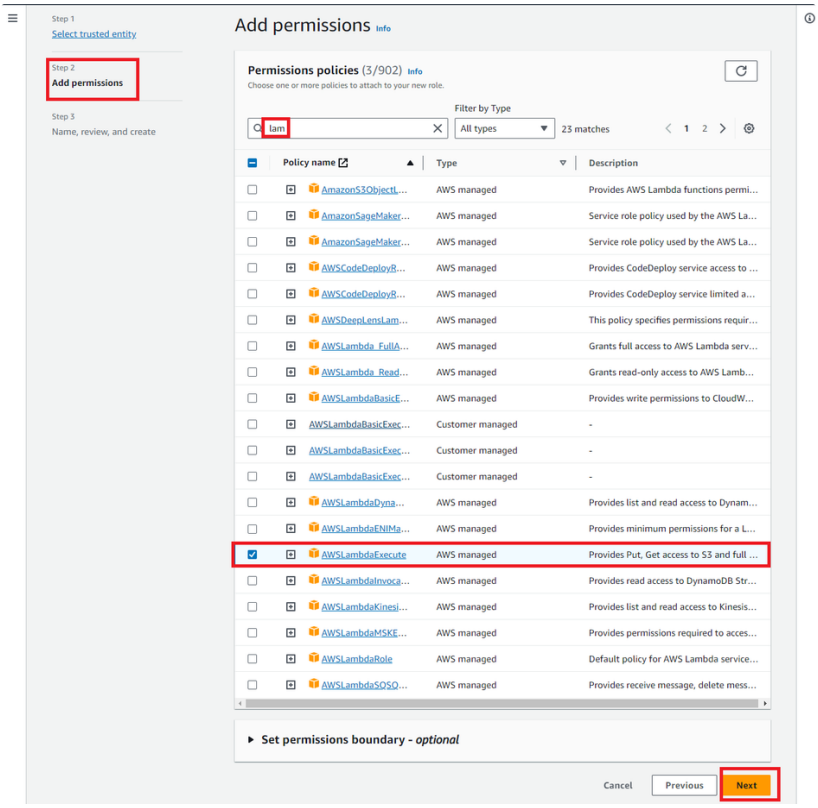




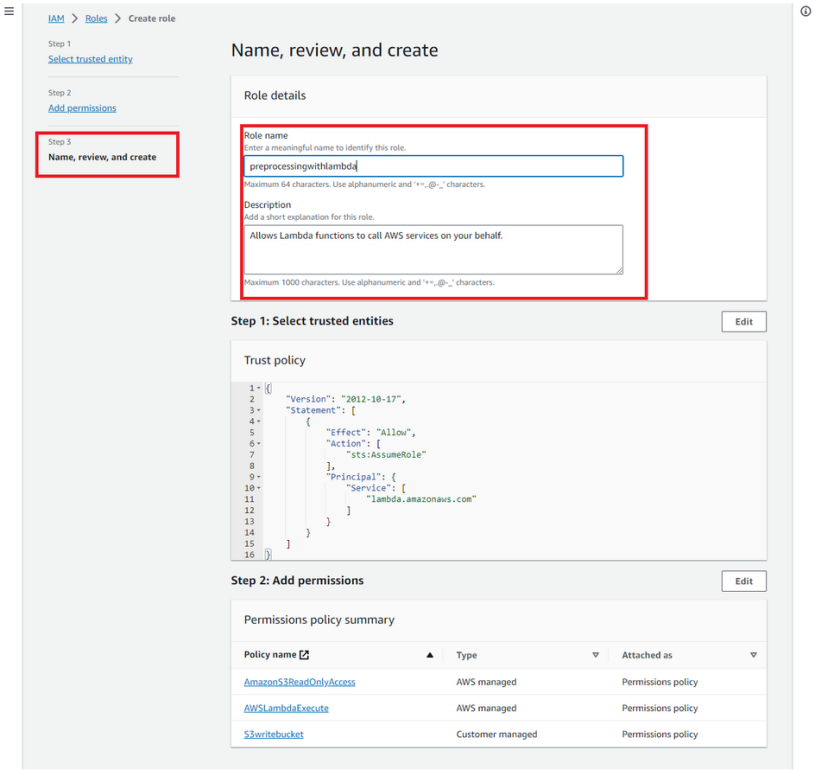


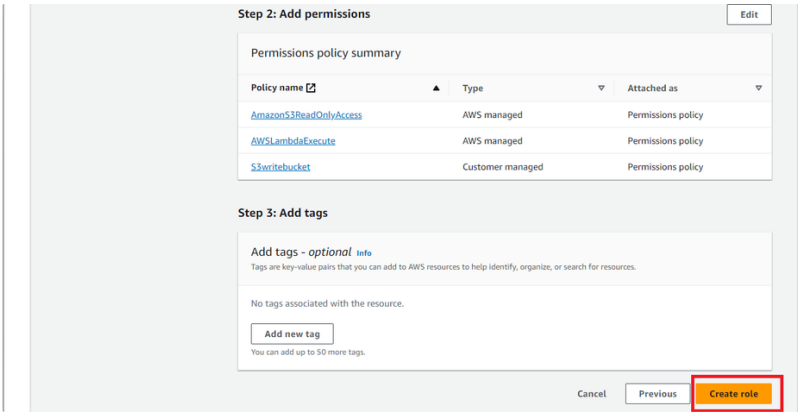
4. In the next add permission page, select three policies namely **S3ReadonlyAcess, S3writebucket and AWSLambdaExecute,** then click **Next**



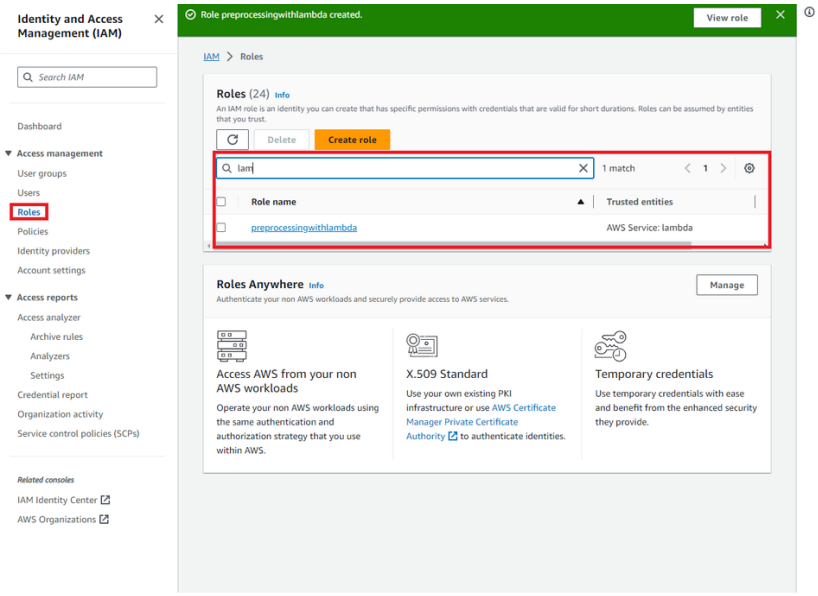


5. On the next page, give the role name, and description, then select **Create role**



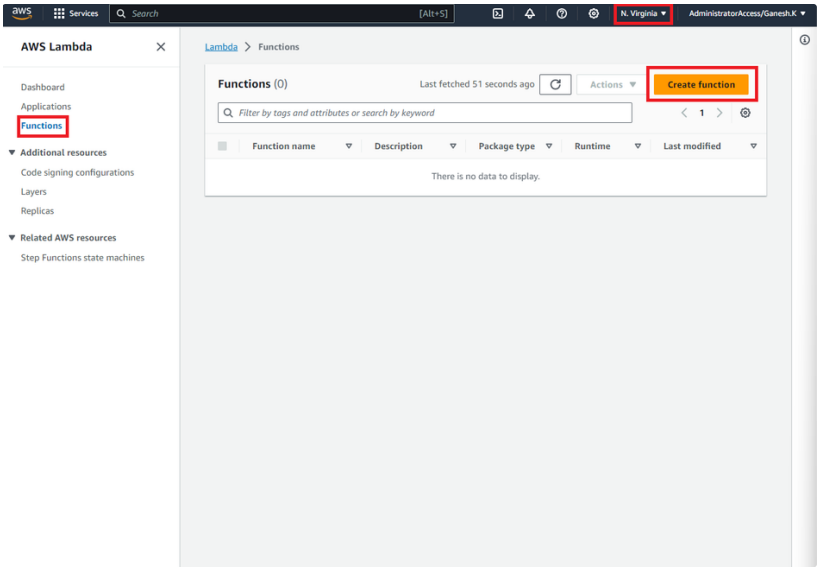


6. Once created, verify by searching the name in the field as shown below

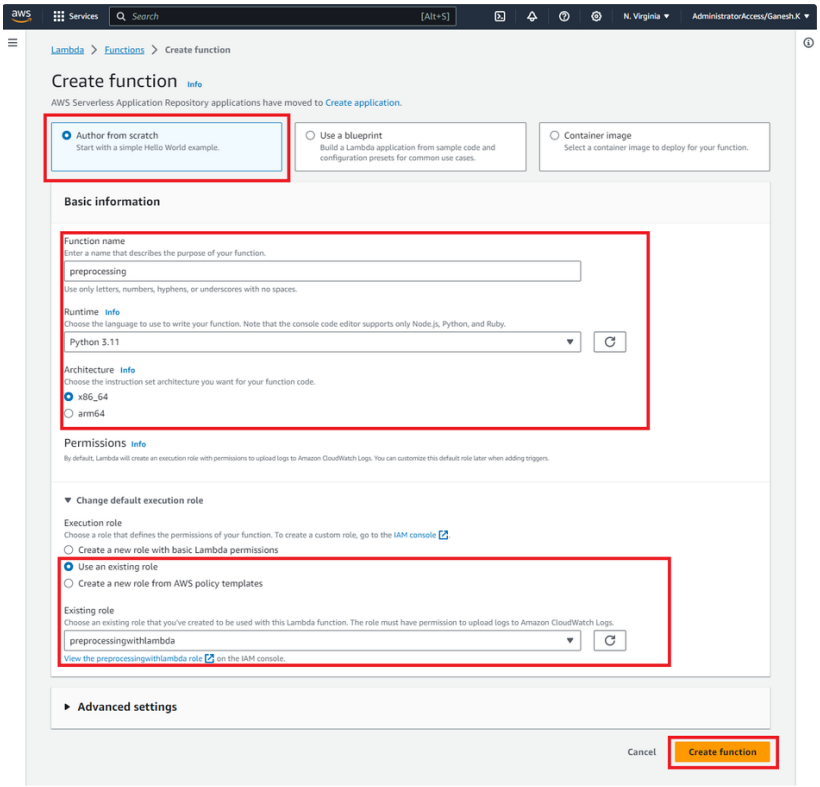


Step 4: Creating lambda function and setting trigger

1. Open the **Management Console** and select the **lambda** page
2. Select **Create Function**



3. In the creation function, choose an **author from scratch**, give a **function name**, select **runtime as python**, **architecture x86_64**, choose a role as **use an existing role** as created, then select **create function**

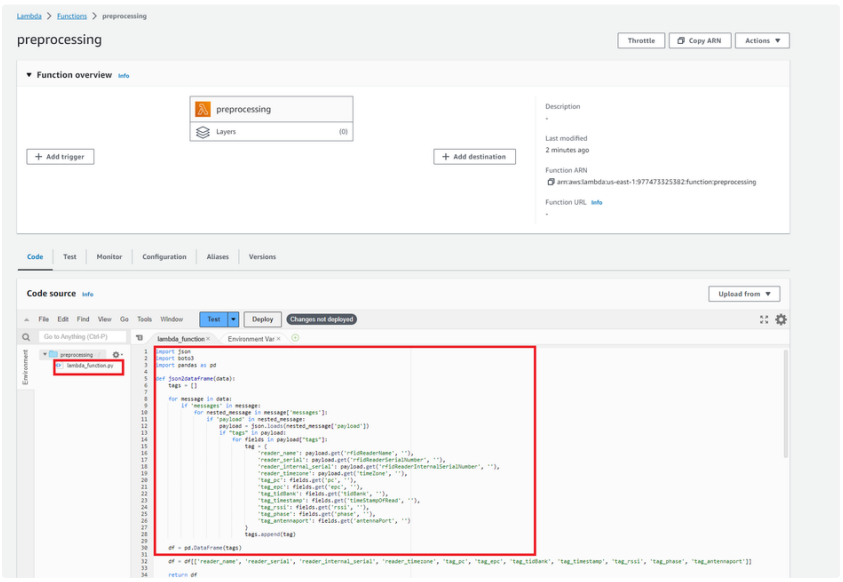


4. Paste the below code in the **code source on the next page**

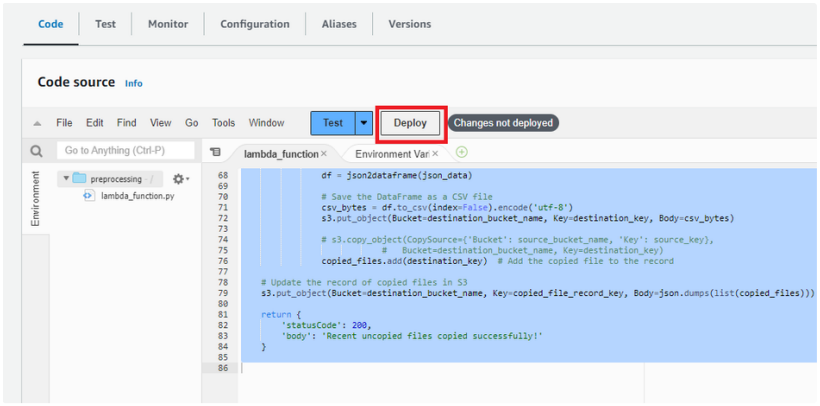
```
import json
import boto3
import pandas as pd

def json2dataframe(data):
    tags = []

    for message in data:
        if 'messages' in message:
            for nested_message in message['messages']:
                if 'payload' in nested_message:
```

```
                    payload = json.loads(nested_message['payload'])
                    if "tags" in payload:
                        for fields in payload["tags"]:
                            tag = {
                                'reader_name': payload.get('rfidReaderName', ''),
                                'reader_serial': payload.get('rfidReaderSerialNumber', ''),
                                'reader_internal_serial': payload.get('rfidReaderInternalSerialNumber', ''),
                                'reader_timezone': payload.get('timeZone', ''),
                                'tag_pc': fields.get('pc', ''),
                                'tag_epc': fields.get('epc', ''),
                                'tag_tidBank': fields.get('tidBank', ''),
                                'tag_timestamp': fields.get('timeStampOfRead', ''),
                                'tag_rssi': fields.get('rssi', ''),
                                'tag_phase': fields.get('phase', ''),
                                'tag_antennaport': fields.get('antennaPort', '')
                            }
                            tags.append(tag)

    df = pd.DataFrame(tags)

    df = df[['reader_name', 'reader_serial', 'reader_internal_serial', 'reader_timezone', 'tag_pc', 'tag_epc', 'tag_tidBank', 'tag_timestam

    return df

def lambda_handler(event, context):
    # Define the source and destination bucket names
    source_bucket_name = 'bucket-inputdata'
    destination_bucket_name = 'bucket-processedata'
    copied_file_record_key = 'copied_files.json'  # Key to store the record of copied files

    # Initialize the S3 client
    s3 = boto3.client('s3')

    # List objects in the source bucket
    response = s3.list_objects_v2(Bucket=source_bucket_name)

    # Load the record of copied files from S3 if it exists
    copied_files = set()
    try:
        copied_files_obj = s3.get_object(Bucket=destination_bucket_name, Key=copied_file_record_key)
        copied_files = set(json.loads(copied_files_obj['Body'].read().decode('utf-8')))
    except Exception as e:
        # If the record doesn't exist, it's okay; start with an empty set
        pass

    # Iterate through the files in the source bucket, copy only if not in the record
    if 'Contents' in response:
        for obj in response['Contents']:
            source_key = obj['Key']
            destination_key = source_key.rsplit('.', 1)[0] + '.csv'
            if destination_key not in copied_files:

                # Read the JSON data
                s3_response = s3.get_object(Bucket=source_bucket_name, Key=source_key)
                json_data = json.loads(s3_response['Body'].read().decode('utf-8'))

                df = json2dataframe(json_data)

                # Save the DataFrame as a CSV file
                csv_bytes = df.to_csv(index=False).encode('utf-8')
                s3.put_object(Bucket=destination_bucket_name, Key=destination_key, Body=csv_bytes)

                # s3.copy_object(CopySource={'Bucket': source_bucket_name, 'Key': source_key},
                            #   Bucket=destination_bucket_name, Key=destination_key)
                copied_files.add(destination_key)  # Add the copied file to the record

    # Update the record of copied files in S3
    s3.put_object(Bucket=destination_bucket_name, Key=copied_file_record_key, Body=json.dumps(list(copied_files)))

    return {
        'statusCode': 200,
        'body': 'Recent uncopied files copied successfully!'
```

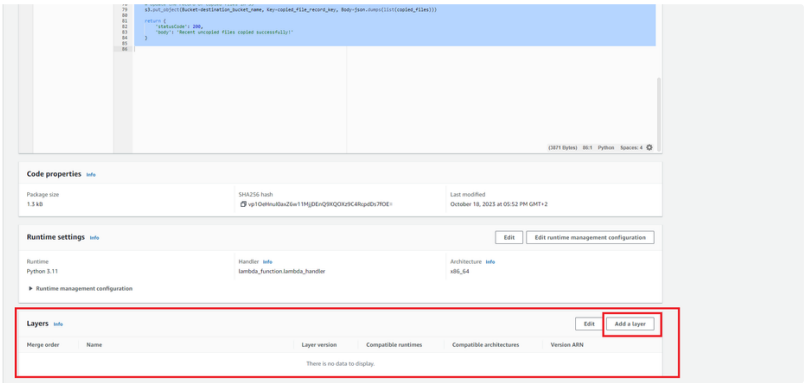
```
84      }
85
86
```



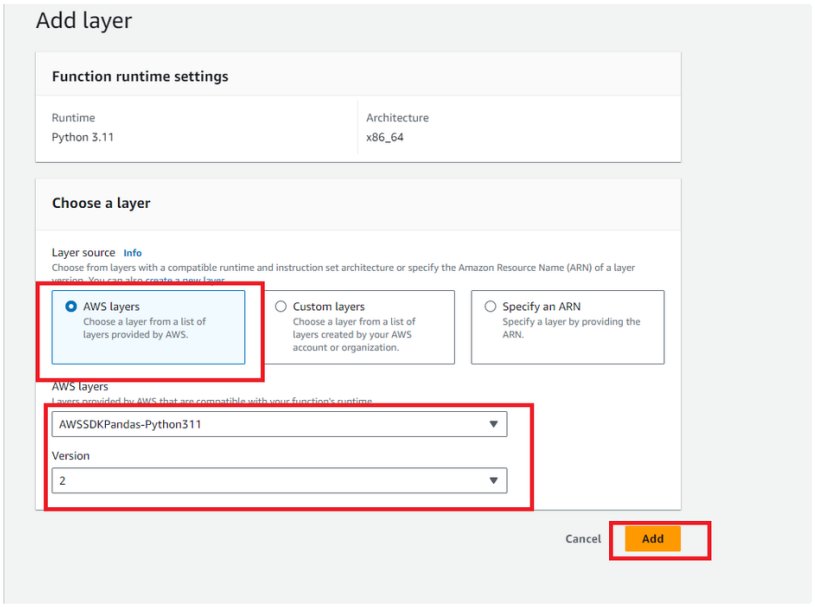
5. Once the code is pasted, click **Deploy**



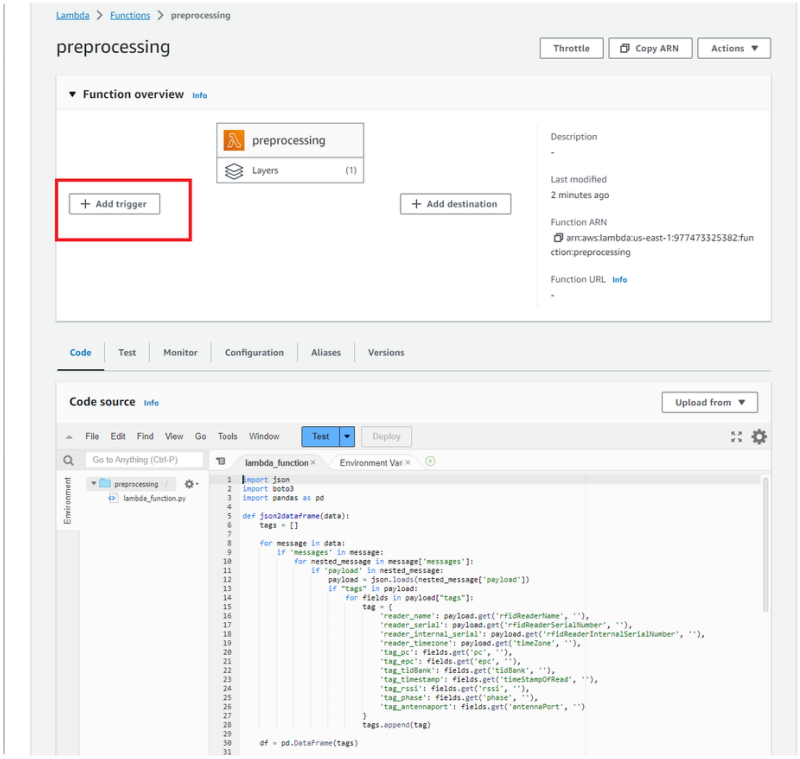
6. Now scroll down to the layers, click **add layer**



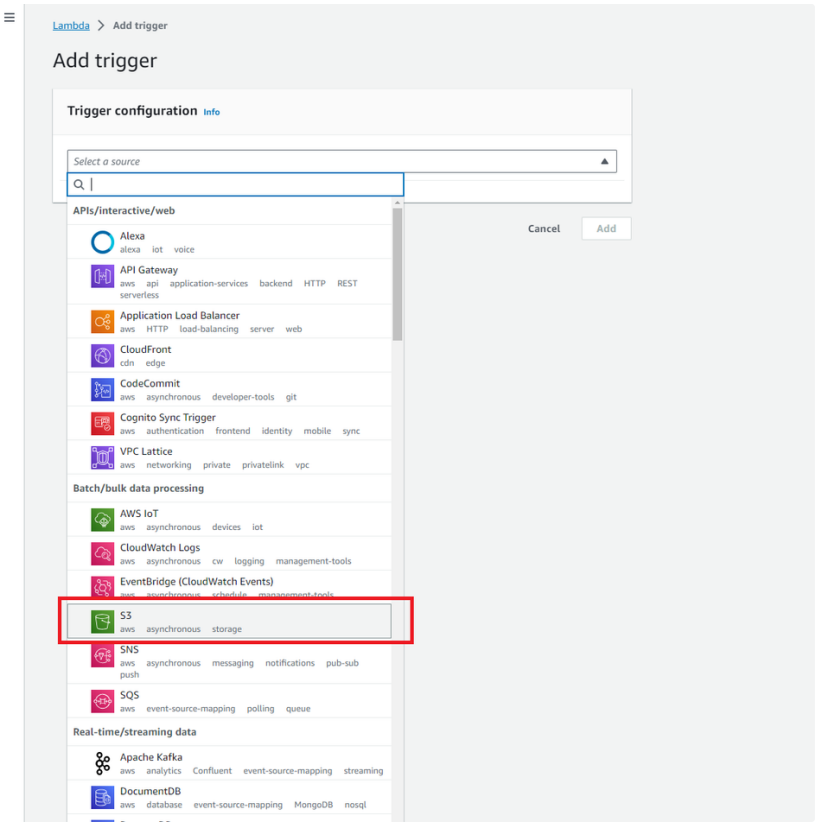
7. In the layer page, select **pandas** and version, then select **add**



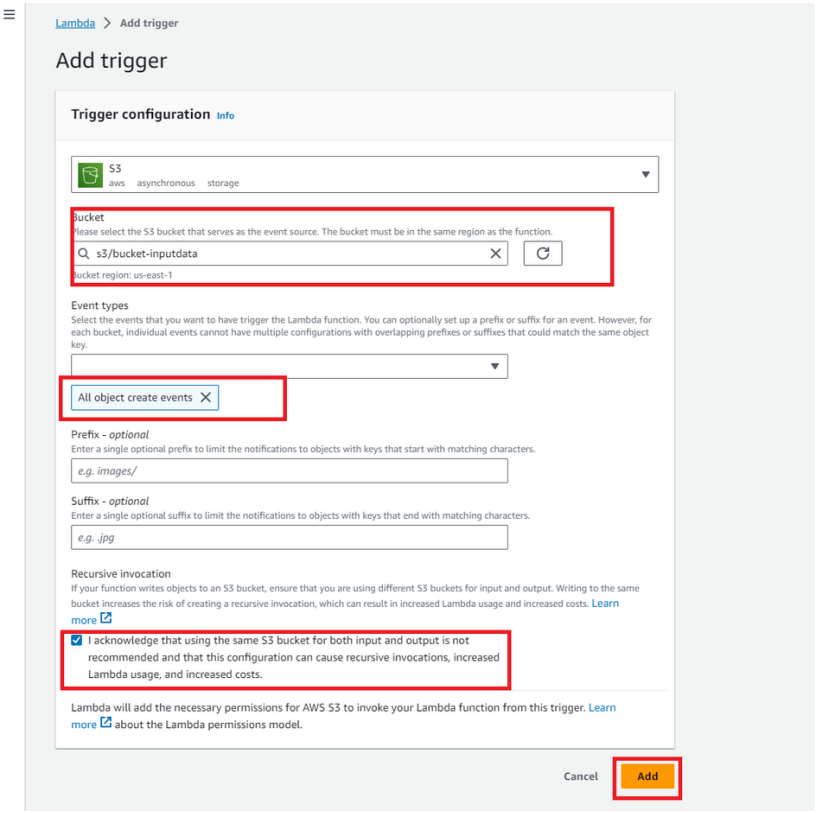
8. Now in the **function overview** click **Add trigger**



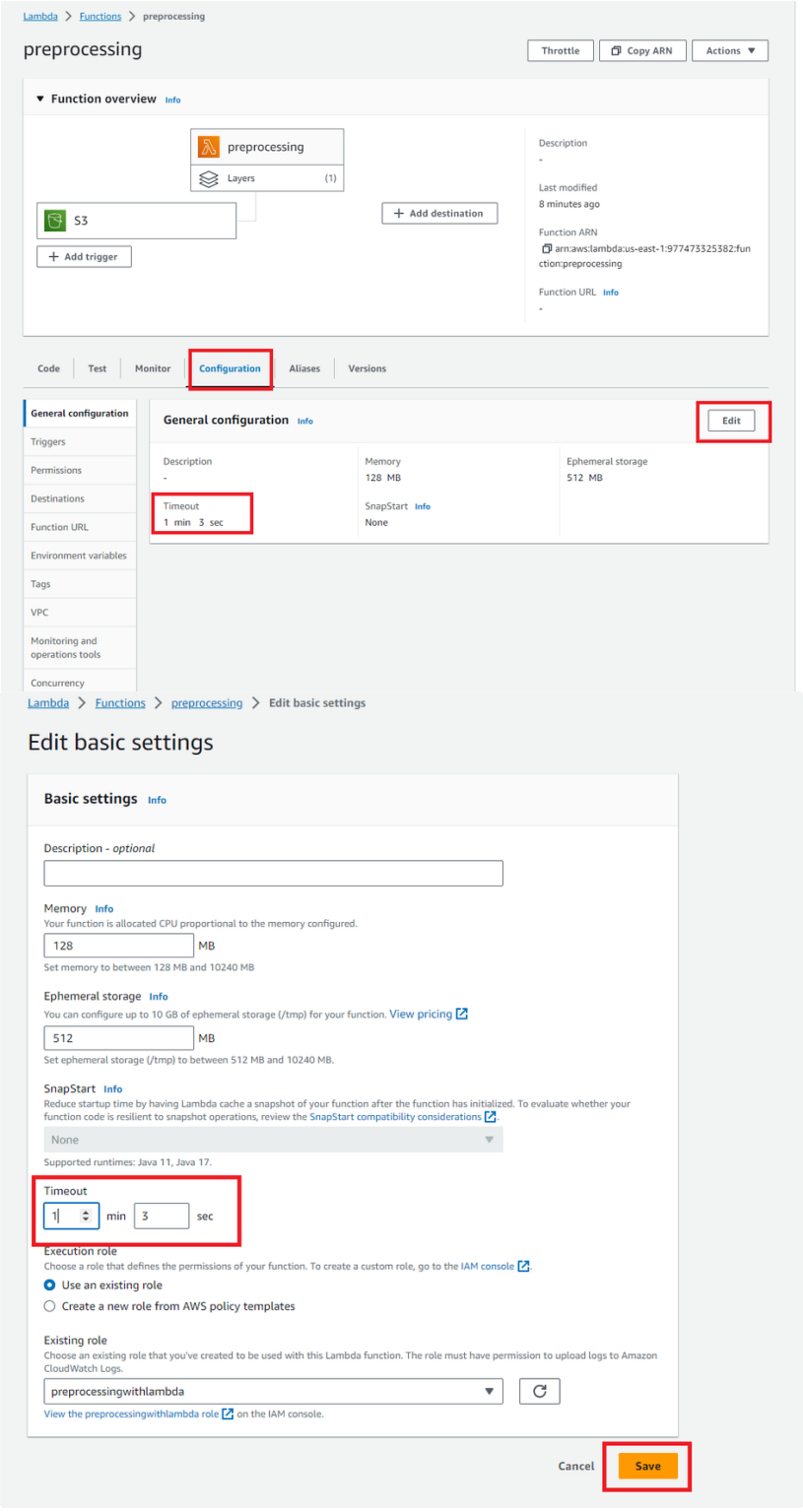
9. Select the source **S3** on the next page then click **next**



10. On the next page, select the **input bucket**, check the **acknowledge** then select **Add**



11. Goto **configuration tab on the lambda function** page and choose **edit in General configuration** as shown below

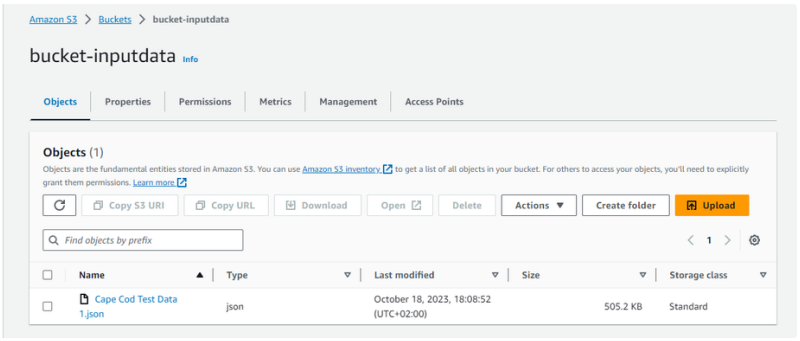


12. After changing the **time click save**

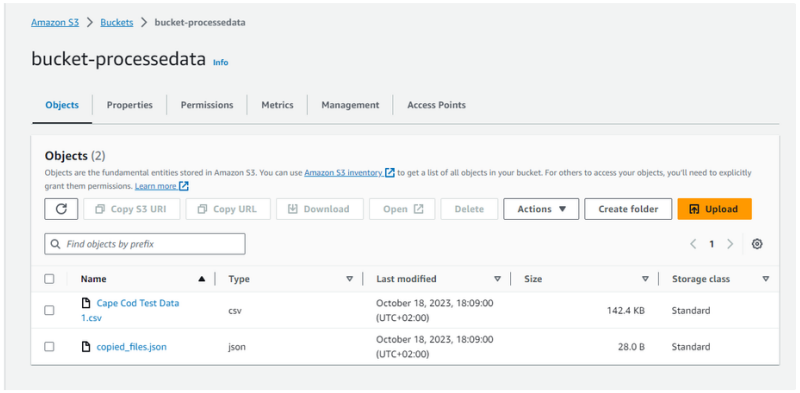
    If we don't change in case the execution will take more time than specified the function will stop due to timeout

Checking

1. Now upload a JSON file in the bucket-inputdata

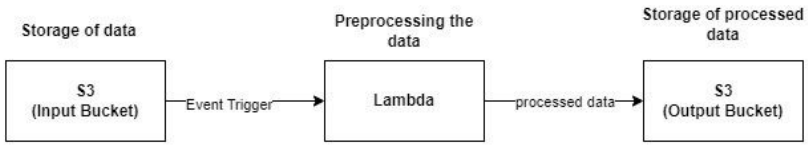


2. You should be able to see the same file in the CSV format and a copied_files.json that stores the files that have been processed to avoid duplication.



3. In case it didn't give the required output see the error in Cloudwatch (the code given in the document works fine)

Advosense Pipeline in AWS



Improvisation of lambda code

This session discusses the improvement of the lambda function code. The previous code only converted the data from JSON format to CSV format. However, the updated code adds an extra step by converting the timestamps into the proper format of date and time in a separate column. To incorporate the improved functionality, replace the code in STEP 4, the 4th procedure, with the following code. If you wish to retain the old functionality of just data conversion, set the variable "**preprocess_data**" to "**False**" on line 104 of the code below. The code lines 64 and 65 remove the column which contains all values as "**NAN**".

```
1  import json
2  import boto3
3  import pandas as pd
4  import datetime
5  import pytz
6
7  # Function to convert microseconds to datetime objects
8  def convert_microseconds_to_datetime(timestamp_microseconds):
9      """
10     Converts a timestamp in microseconds to a datetime object in a specified timezone.
11
12     Args:
13         timestamp_microseconds (int): A timestamp in microseconds.
14
15     Returns:
```

```
        str: A datetime string in the format 'YYYY-MM-DD HH:MM:SS' with the desired timezone applied.
             Microseconds are removed.


    """

    timestamp_seconds = int(timestamp_microseconds) / 1_000_000
    timestamp_datetime = datetime.datetime.fromtimestamp(timestamp_seconds)
    desired_timezone = pytz.timezone('US/Eastern')  # Replace with your desired time zone, currently it's -5
    timestamp_datetime = timestamp_datetime.astimezone(desired_timezone)
    timestamp_datetime = timestamp_datetime.replace(microsecond=0)  # Remove microseconds
    timestamp_datetime = timestamp_datetime.strftime('%Y-%m-%d %H:%M:%S')  # Format without milliseconds
    return timestamp_datetime

# Function to split date and time
def formatting_datetime(df, remove_microseconds = True, split_datetime = True):
    """
    Formats a DataFrame with timestamp data, optionally removing microseconds and
    splitting datetime into separate Date and Time columns.

    Args:
        df (DataFrame): A DataFrame containing a 'tag_timestamp' column with timestamp data.
        remove_microseconds (bool, optional): A boolean flag to remove microseconds
                                              from the timestamp (default is True).
        split_datetime (bool, optional): A boolean flag to split the timestamp into separate
                                         'Date' and 'Time' columns (default is True).

    Returns:
        DataFrame: A modified DataFrame with 'Date' and 'Time' columns based on the specified flags.


    """

    if remove_microseconds:
        # Apply the conversion function to the DataFrame column
        df['tag_timestamp'] = df['tag_timestamp'].apply(convert_microseconds_to_datetime)
    if split_datetime:
        # Split 'Data and Time' column into 'Date' and 'Time' columns
        df[['Date', 'Time']] = df['tag_timestamp'].str.split(pat=' ', n=1, expand=True)

        # Drop the original 'Data and Time' column
        df.drop(columns=['tag_timestamp'], inplace=True)

        # Reorder columns with 'Date' and 'Time' at the start
        new_column_order = ['Date', 'Time'] + [col for col in df.columns if col not in ['Date', 'Time']]
        df = df[new_column_order]
        df['Time'] = df['Time'].str.split('.').str[0].str[:8]

    dropped_columns = df.columns[df.isnull().all()]
    df = df.dropna(axis=1, how='all')
    print("\nDropped Columns:")
    print(dropped_columns)
    return df


#Fucntion to convert data from
#JSON to CSV
def json2dataframe(data):
    tags = []

    for message in data:
        if 'messages' in message:
            for nested_message in message['messages']:
                if 'payload' in nested_message:
                    payload = json.loads(nested_message['payload'])
                    if "tags" in payload:
                        for fields in payload["tags"]:
                            tag = {
                                'reader_name': payload.get('rfidReaderName', ''),
                                'reader_serial': payload.get('rfidReaderSerialNumber', ''),
                                'reader_internal_serial': payload.get('rfidReaderInternalSerialNumber', ''),
                                'reader_timezone': payload.get('timeZone', ''),
```

```
                                'tag_pc': fields.get('pc', ''),
                                'tag_epc': fields.get('epc', ''),
                                'tag_tidBank': fields.get('tidBank', ''),
                                'tag_timestamp': fields.get('timeStampOfRead', ''),
                                'tag_rssi': fields.get('rssi', ''),
                                'tag_phase': fields.get('phase', ''),
                                'tag_antennaport': fields.get('antennaPort', '')
                            }
                            tags.append(tag)

    df = pd.DataFrame(tags)

    df = df[['reader_name', 'reader_serial', 'reader_internal_serial', 'reader_timezone', 'tag_pc', 'tag_epc', 'tag_tidBank', 'tag_timestamp

    return df

def lambda_handler(event, context):
    # Define the source and destination bucket names
    source_bucket_name = 'bucket-inputdata'
    destination_bucket_name = 'bucket-processedata'
    copied_file_record_key = 'copied_files.json'  # Key to store the record of copied files
    preprocess_data = True

    # Initialize the S3 client
    s3 = boto3.client('s3')

    # List objects in the source bucket
    response = s3.list_objects_v2(Bucket=source_bucket_name)

    # Load the record of copied files from S3 if it exists
    copied_files = set()
    try:
        copied_files_obj = s3.get_object(Bucket=destination_bucket_name, Key=copied_file_record_key)
        copied_files = set(json.loads(copied_files_obj['Body'].read().decode('utf-8')))
    except Exception as e:
        # If the record doesn't exist, it's okay; start with an empty set
        pass

    # Iterate through the files in the source bucket, copy only if not in the record
    if 'Contents' in response:
        for obj in response['Contents']:
            source_key = obj['Key']
            destination_key = source_key.rsplit('.', 1)[0] + '.csv'
            if destination_key not in copied_files:

                # Read the JSON data
                s3_response = s3.get_object(Bucket=source_bucket_name, Key=source_key)
                json_data = json.loads(s3_response['Body'].read().decode('utf-8'))

                if preprocess_data:
                    df = json2dataframe(json_data)
                    df = formatting_datetime(df)
                else:
                    df = json2dataframe(json_data)

                # Save the DataFrame as a CSV file
                csv_bytes = df.to_csv(index=False).encode('utf-8')
                s3.put_object(Bucket=destination_bucket_name, Key=destination_key, Body=csv_bytes)

                # s3.copy_object(CopySource={'Bucket': source_bucket_name, 'Key': source_key},
                            #   Bucket=destination_bucket_name, Key=destination_key)
                copied_files.add(destination_key)  # Add the copied file to the record

    # Update the record of copied files in S3
    s3.put_object(Bucket=destination_bucket_name, Key=copied_file_record_key, Body=json.dumps(list(copied_files)))

    return {
        'statusCode': 200,
        'body': 'Recent uncopied files copied successfully!'
    }
```