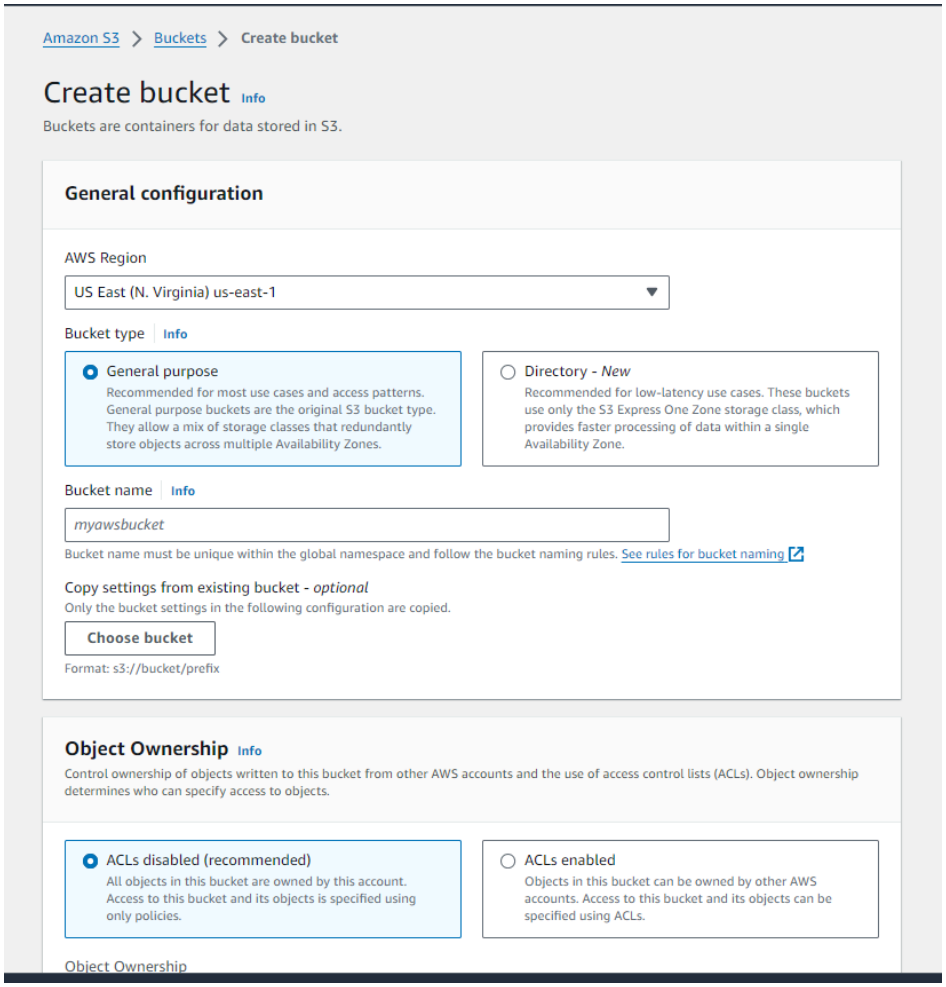
Work with RXNORM file,

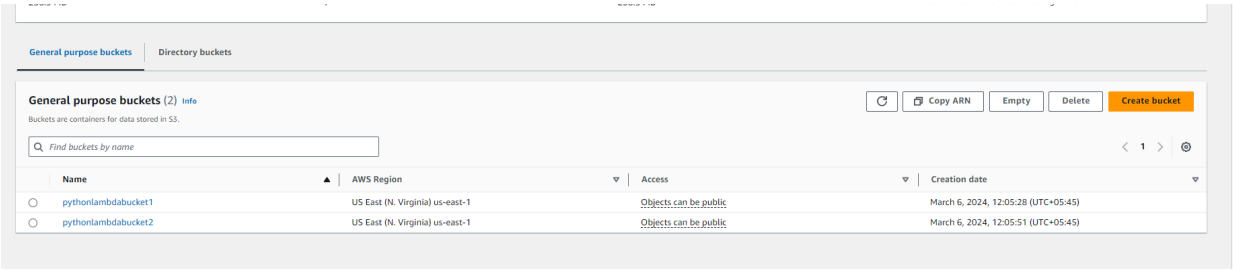
1. Scrap the latest RXNORM file from NLM webpage
2. Download the latest RXNORM file with api_key
3. Create a log file for the downloaded file
4. Add header into each rff from RXNORM.xlsx
5. Add CODE_SET & VERSION_MONTH column with default values RxNorm and version month from downloaded filename
6. Convert dates into YYYY-MM-DD
7. Save files as txt delimited by comma(,)c
8. Validate row_count between original and converted files

Steps Involved:

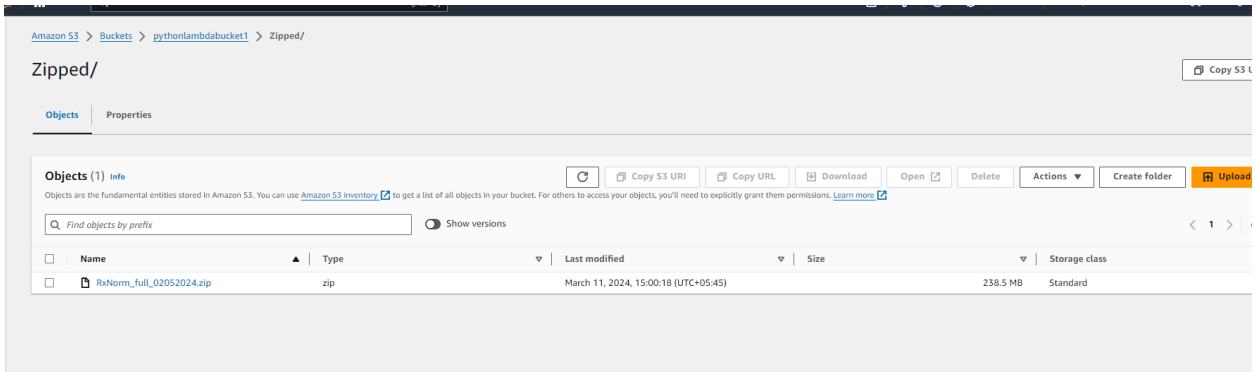
1) **A bucket is created to store the zip file containing the (.RFF) files, Excel file and the files created after transformations and headers are applied.**

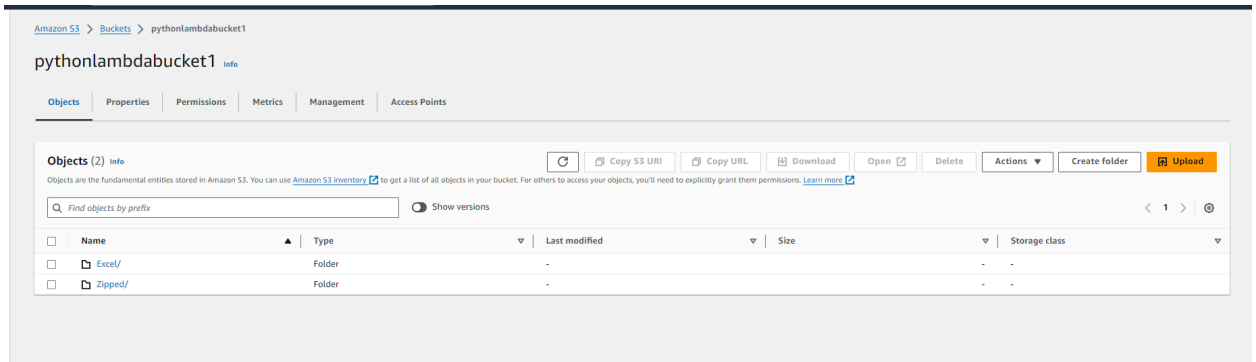


Buckets created successfully

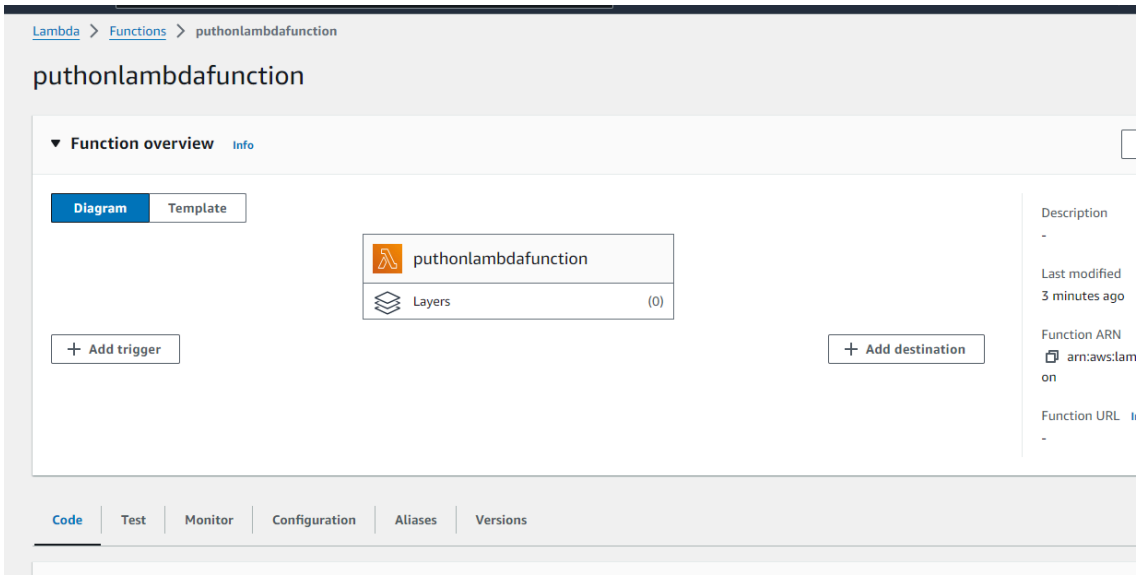


2) **Folders for zipped and excel sheets are created and respective file is uploaded in it.**



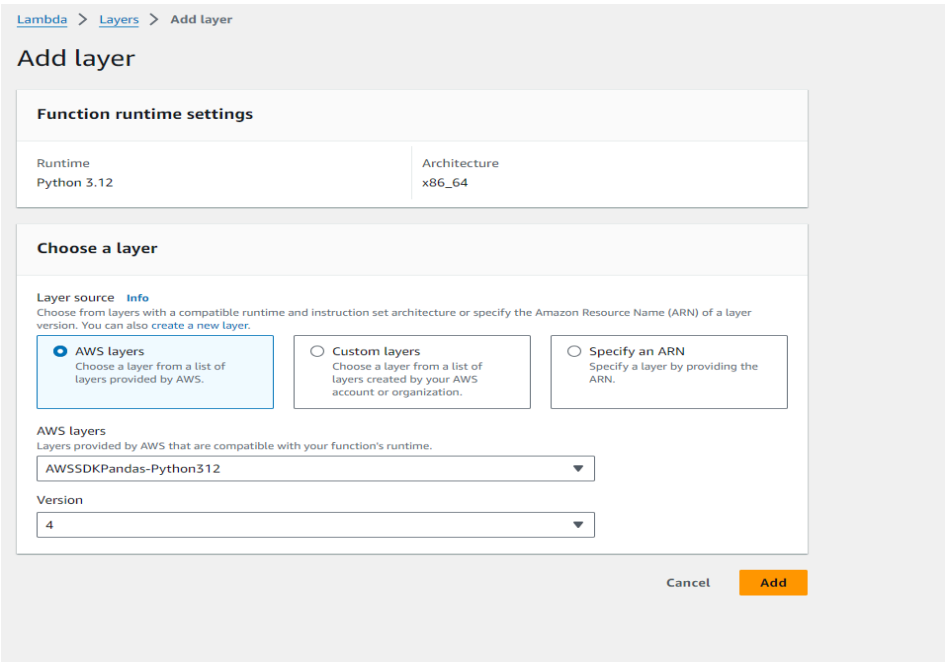


3) Creation of AWS Lambda Function



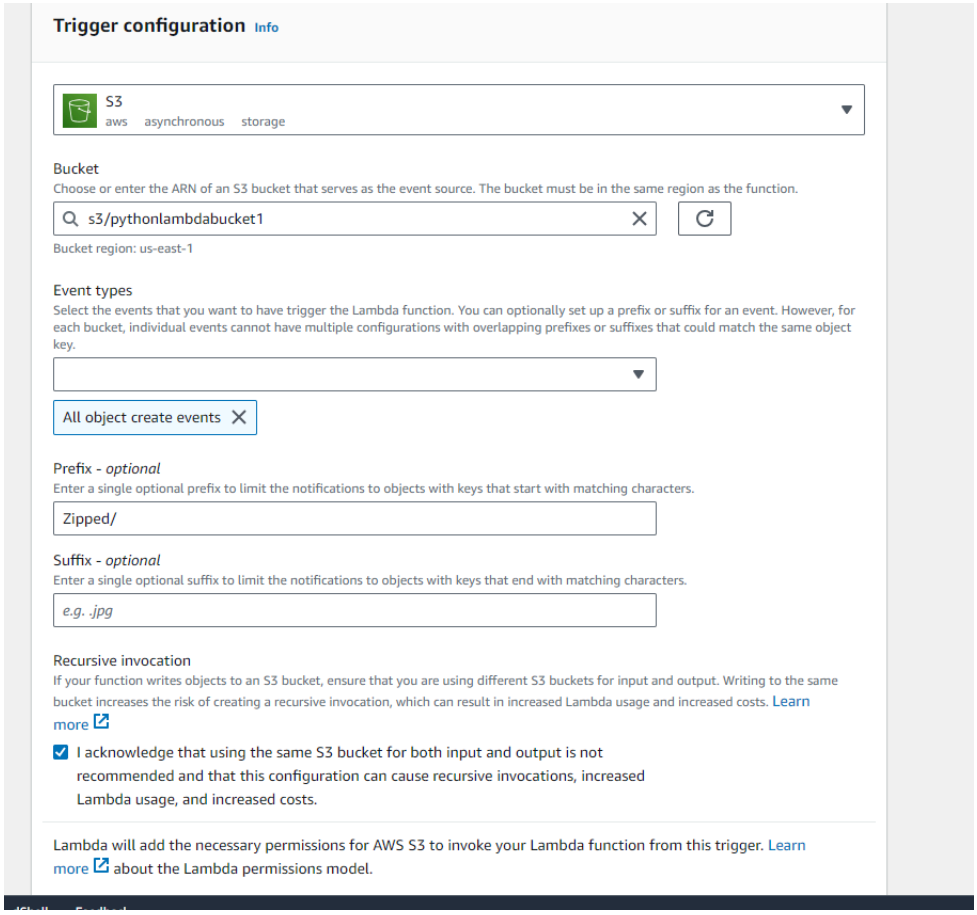
4) Adding Lambda Layers

Pandas Layer is added to the Lambda for processing the files per task requirements

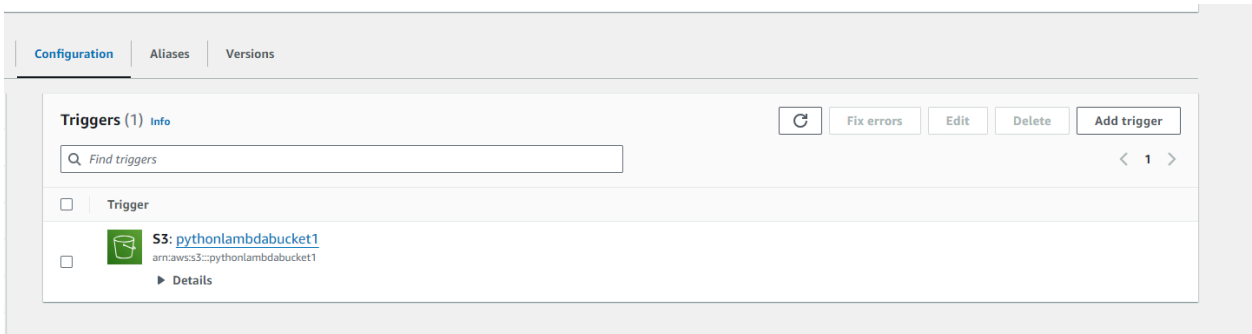


5) Adding Trigger

Choose Event types – all
S3 bucket made previously is added as a trigger to process files.



Triggered added successfully.



6) The complete code is written:

Go Tools Window Test Deploy Changes not deployed

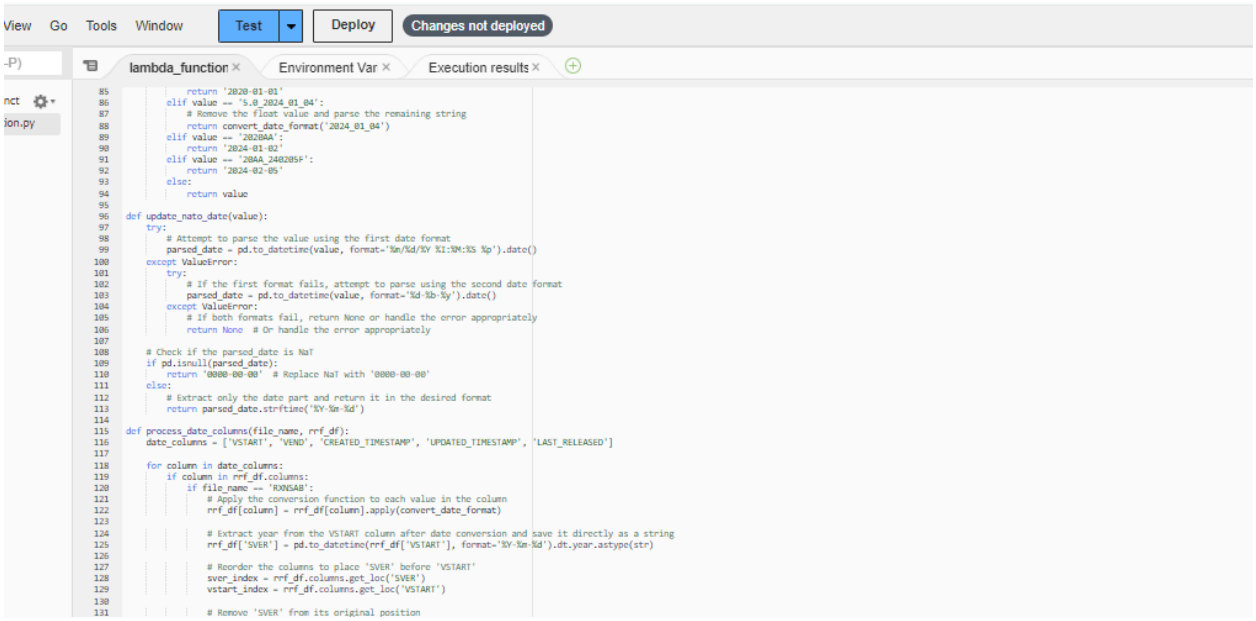
lambda_function × Environment Var × Execution results ×

```python
import boto3
import zipfile
import io
import os
import pandas as pd
from io import BytesIO

s3 = boto3.client('s3')
excel_headers = {}

def read_excel_from_s3(bucket):
    try:
        folder_path = 'excelfiles/'
        excel_file_name = 'RxNorm_Header.xlsx'
        key = folder_path + excel_file_name

        # Download the Excel file to the /tmp directory
        local_excel_file = '/tmp/RxNorm_Header.xlsx'
        s3.download_file(bucket, key, local_excel_file)

        # Check if the Excel file exists
        if os.path.exists(local_excel_file):
            print(f"Excel file downloaded to: {local_excel_file}")

        # Read the Excel file into an ExcelFile object
        excel_file = pd.ExcelFile(local_excel_file)

        # Get the sheet names
        sheet_names = excel_file.sheet_names
        print("Sheet names:", sheet_names)

        for sheets in sheet_names:
            # Read the data from the sheet into a DataFrame
            sheets_data = excel_file.parse(sheets, header=None)
            headers_data = sheets_data.iloc[:, 0].tolist()
            excel_headers[sheets] = headers_data

        print(f'excel_headers dictionary for sheet {sheet_names[0]}: {excel_headers[sheet_names[0]]}')

    except Exception as e:
        print(f"Error occurred: {e}")

def code_set_and_version_month(zip_filename, rrf_df):
    try:
        # Extract version month from the filename
        version_month = os.path.splitext(zip_filename)[0].split('_')[-1]
```

Go Tools Window Test Deploy Changes not deployed

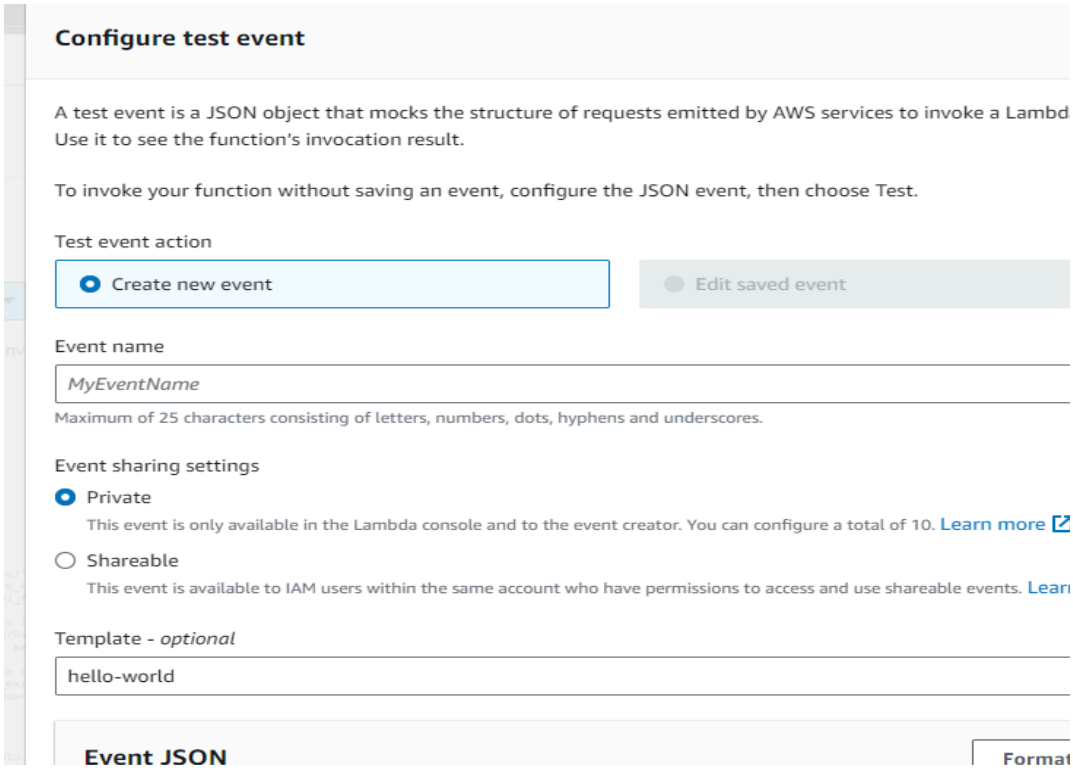
lambda_function × Environment Var × Execution results ×

```python
def code_set_and_version_month(zip_filename, rrf_df):
    try:
        # Extract version month from the filename
        version_month = os.path.splitext(zip_filename)[0].split('_')[-1]

        # Convert version month to a more readable format
        version_month = pd.to_datetime(version_month, format='%m%d%Y').strftime('%Y-%m-%d')
        print(f"Version month: {version_month}")

        # Add 'Code Set' and 'Version Month' columns to the DataFrame
        rrf_df['Code Set'] = 'RxNorm'
        rrf_df['Version Month'] = version_month

    except Exception as e:
        print(f"Error occurred while extracting version month: {e}")

    return rrf_df

def apply_header_to_rrf(file_name, rrf_df):
    # Check if the corresponding Excel sheet exists
    if file_name in excel_headers:
        # Get the headers from the Excel sheet
        excel_headers_list = excel_headers[file_name]
        excel_headers_list = [header for header in excel_headers_list if header != 'SVER']

        # Take names from the excel header list up to the length of the split DataFrame
        excel_headers_list = excel_headers_list[:len(rrf_df.columns)]

        # Set the correct header for the DataFrame
        rrf_df.columns = excel_headers_list

    return rrf_df

def convert_date_format(value):
    try:
        # Try to parse the value into datetime format
        parsed_date = pd.to_datetime(value, format='%Y_%m_%d').date()

        # Extract only the date part
        return parsed_date.strftime('%Y-%m-%d')
    except ValueError:
        if value == '2020':
            return '2020-01-01'
        elif value == '5.0_2024_01_04':
            # Remove the float value and parse the remaining string
            return convert_date_format('2024_01_04')
        elif value == '2020AA':
```

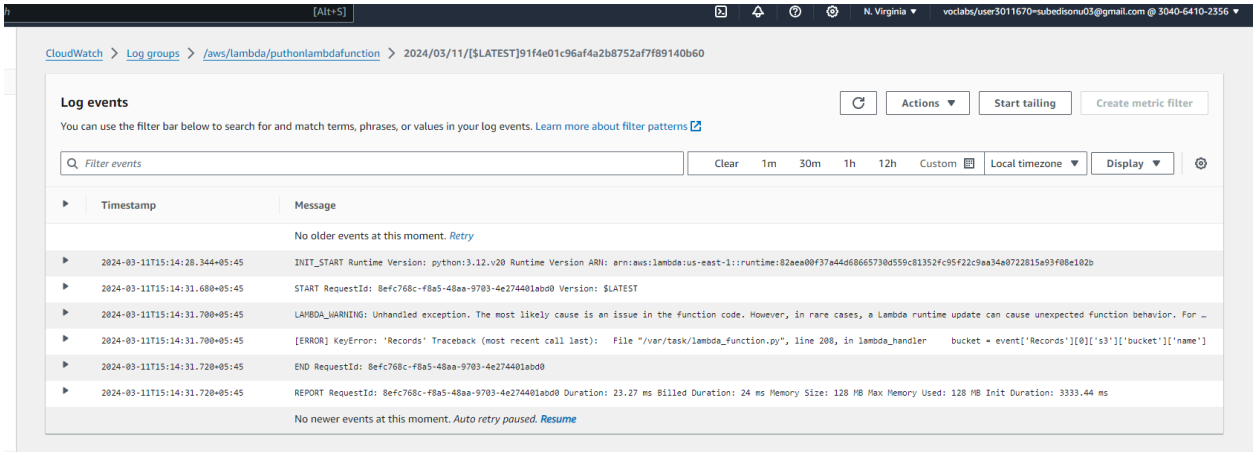



```
            return '2020-01-01'
        elif value == '5.0_2024_01_04':
            # Remove the float value and parse the remaining string
            return convert_date_format('2024_01_04')
        elif value == '2020AA':
            return '2024-01-02'
        elif value == '20AA_240205F':
            return '2024-02-05'
        else:
            return value

def update_nato_date(value):
    try:
        # Attempt to parse the value using the first date format
        parsed_date = pd.to_datetime(value, format='%m/%d/%Y %I:%M:%S %p').date()
    except ValueError:
        try:
            # If the first format fails, attempt to parse using the second date format
            parsed_date = pd.to_datetime(value, format='%d-%b-%y').date()
        except ValueError:
            # If both formats fail, return None or handle the error appropriately
            return None  # Or handle the error appropriately

    # Check if the parsed_date is NaT
    if pd.isnull(parsed_date):
        return '0000-00-00'  # Replace NaT with '0000-00-00'
    else:
        # Extract only the date part and return it in the desired format
        return parsed_date.strftime('%Y-%m-%d')

def process_date_columns(file_name, rrf_df):
    date_columns = ['VSTART', 'VEND', 'CREATED_TIMESTAMP', 'UPDATED_TIMESTAMP', 'LAST_RELEASED']

    for column in date_columns:
        if column in rrf_df.columns:
            if file_name == 'RXNSAB':
                # Apply the conversion function to each value in the column
                rrf_df[column] = rrf_df[column].apply(convert_date_format)

                # Extract year from the VSTART column after date conversion and save it directly as a string
                rrf_df['SVER'] = pd.to_datetime(rrf_df['VSTART'], format='%Y-%m-%d').dt.year.astype(str)

                # Reorder the columns to place 'SVER' before 'VSTART'
                sver_index = rrf_df.columns.get_loc('SVER')
                vstart_index = rrf_df.columns.get_loc('VSTART')

                # Remove 'SVER' from its original position
```

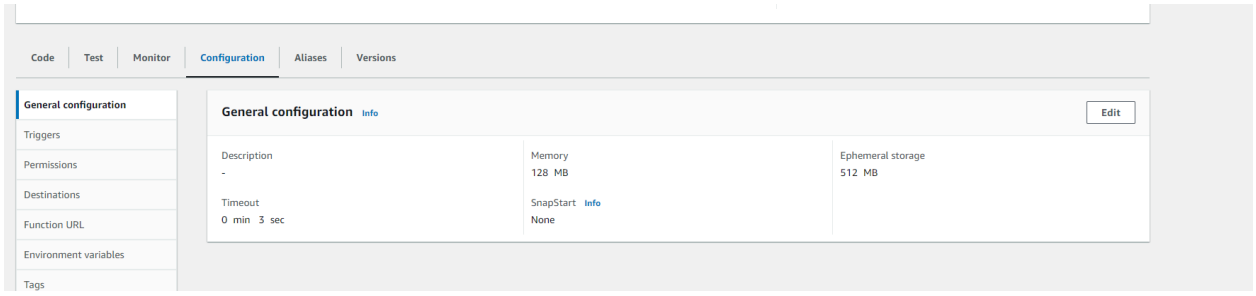
7) **Configuring Test Event**
A simple test event is made to check for correct functioning of the code.



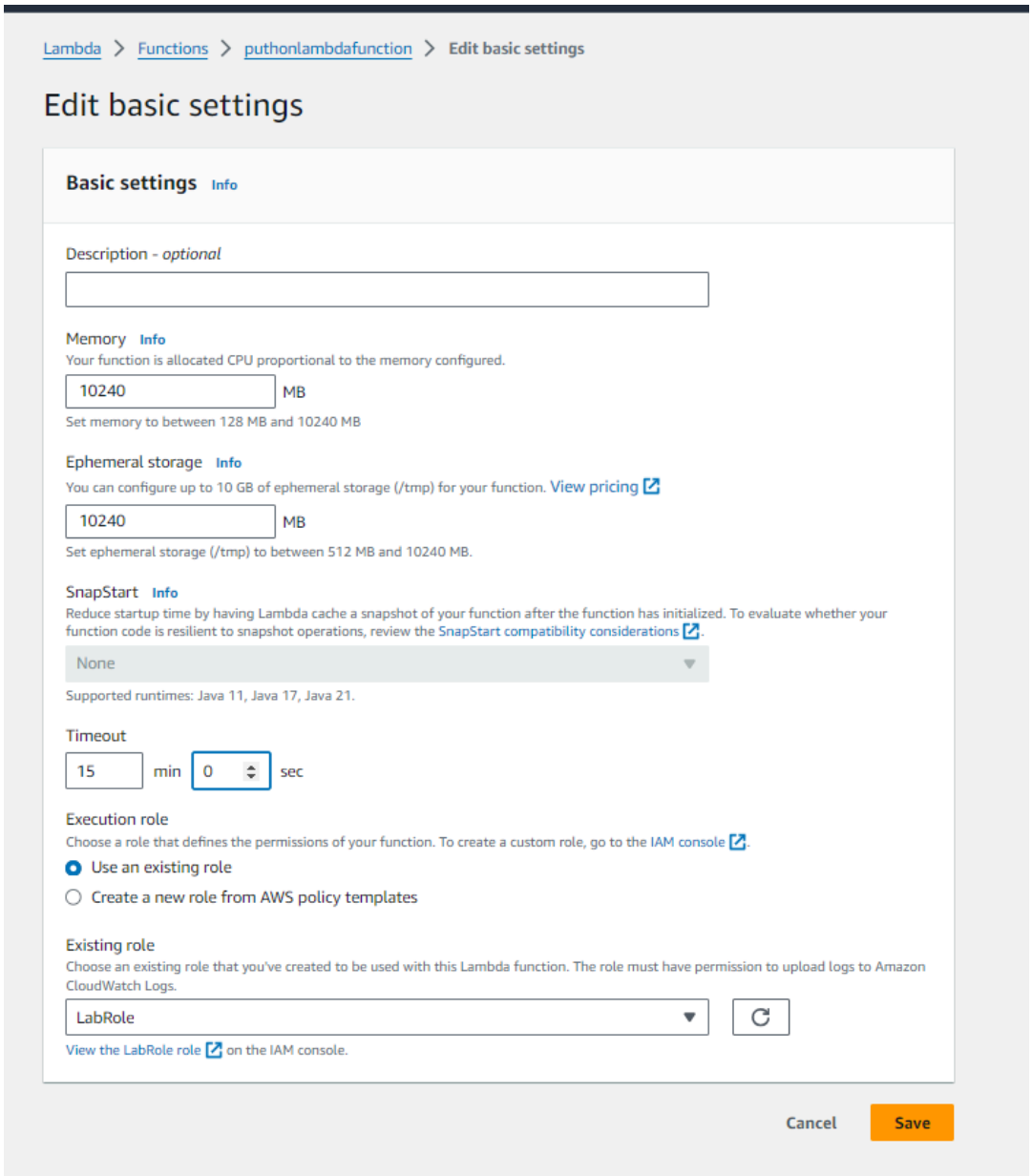
8) Checking log events.. The issues persist and are solved accordingly.



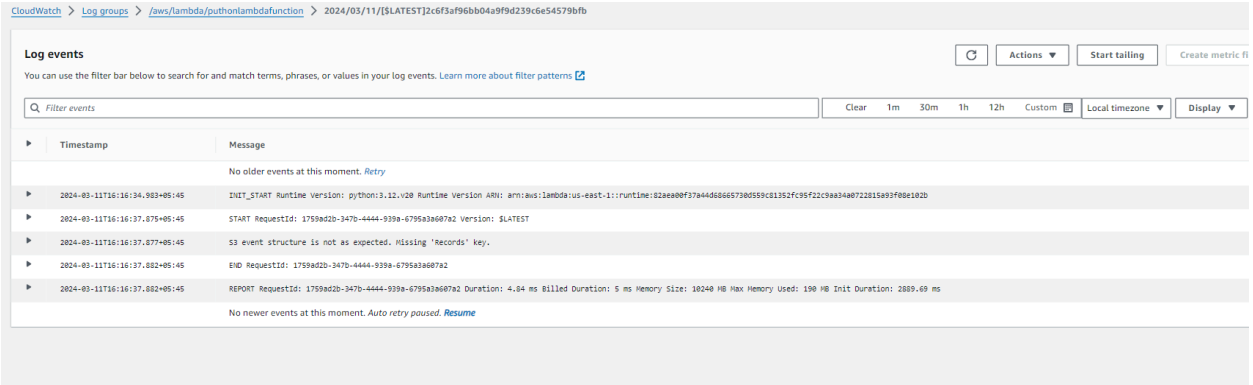
9) Go to the General Configuration of lambda function and change its basic settings.



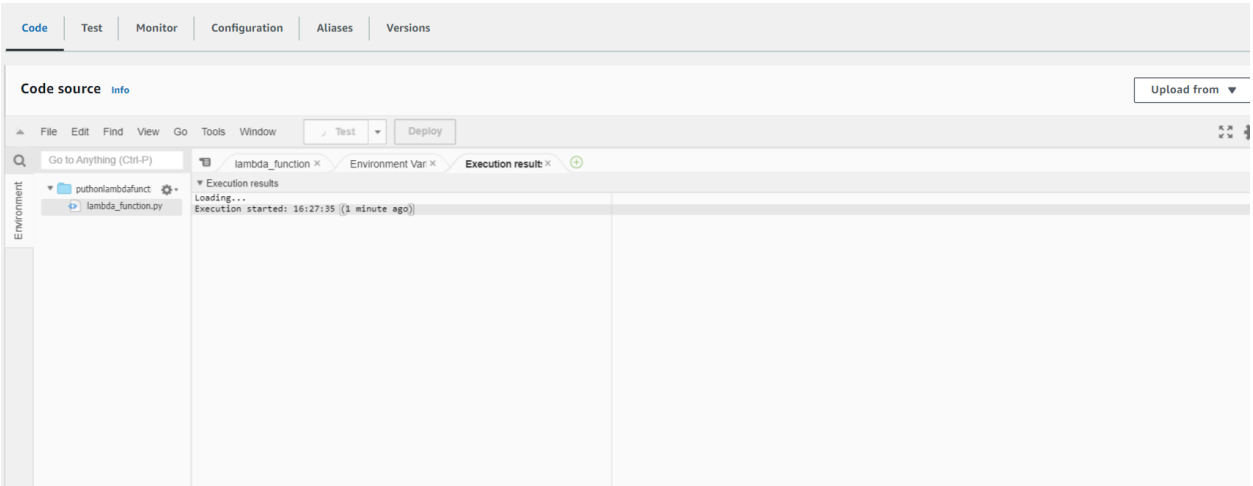
Changing Basic Settings. Edit Memory configuration andTimeout as required.



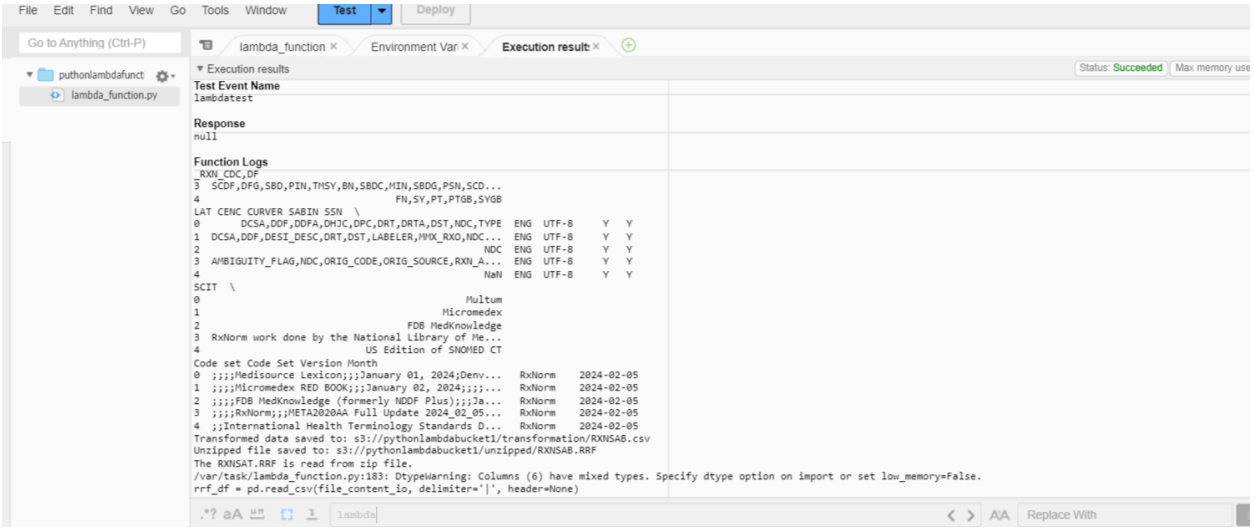
Some of the errors seen in logs of CloudWatch:



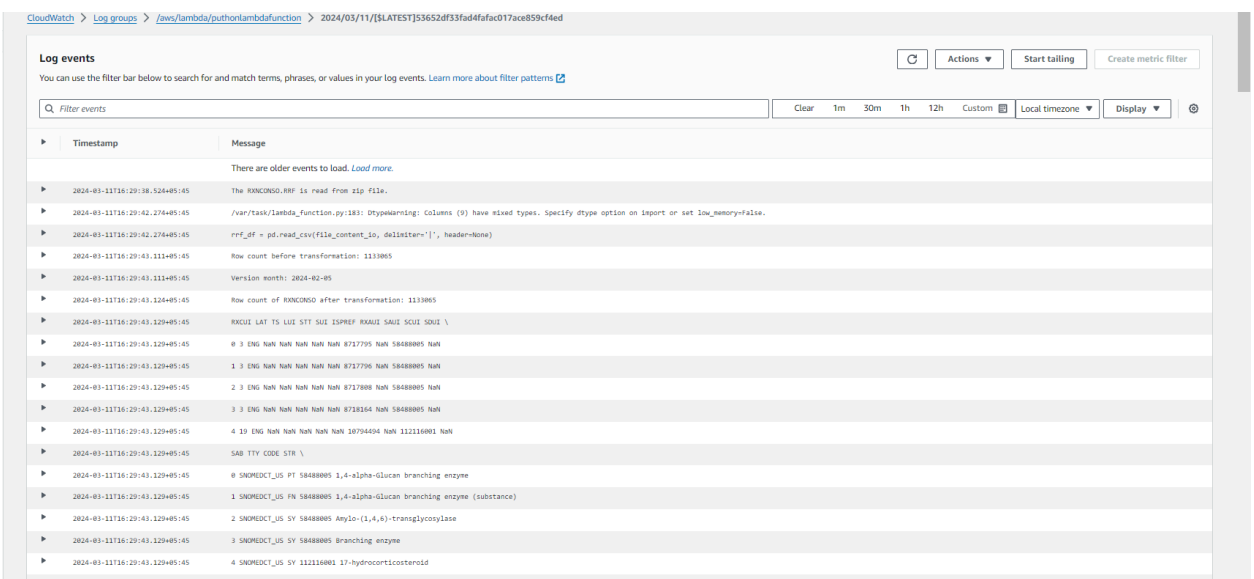
10) Resolving errors and testing the code.



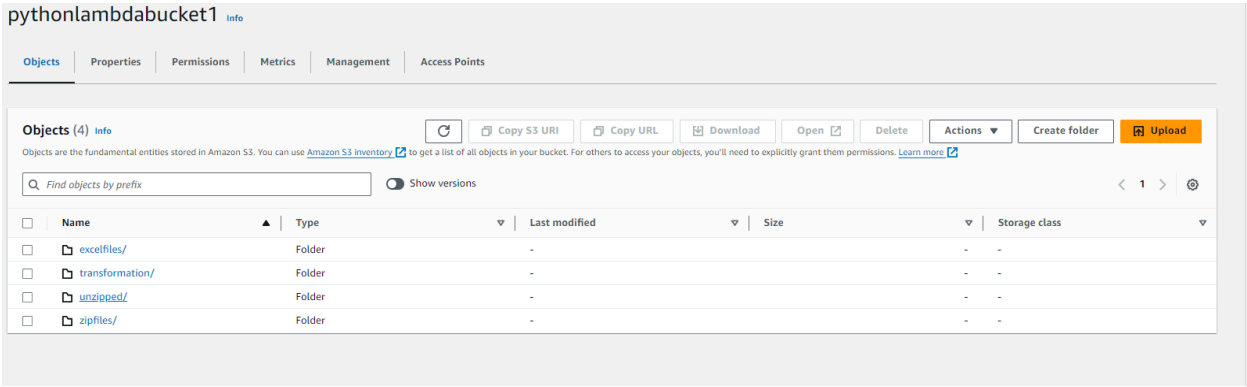
11) Test done successfully.



12) Logs Events in Cloudwatch

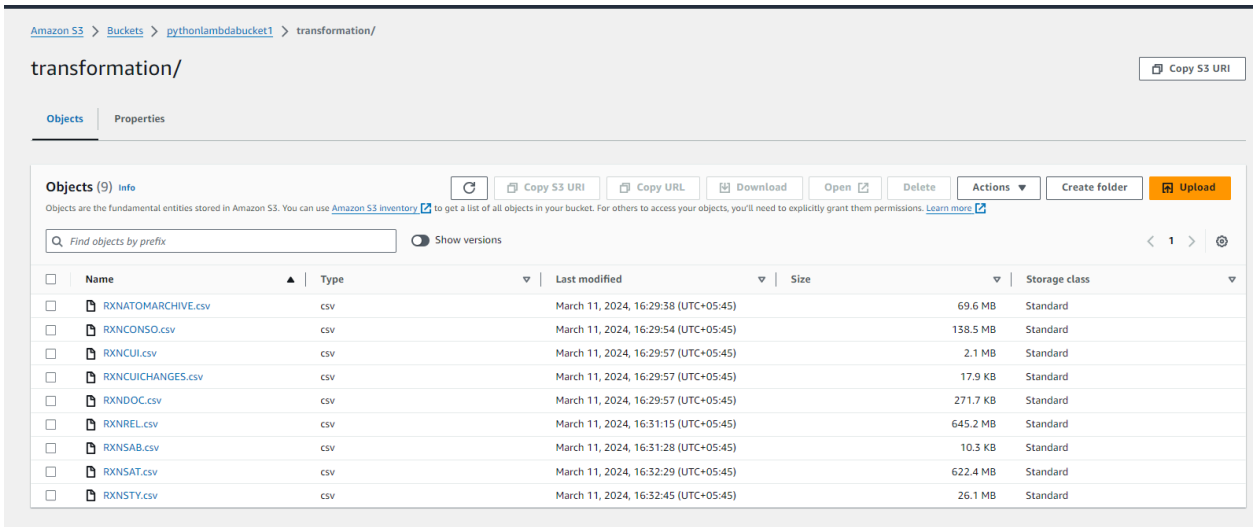


13) The two files transformed and unzipped created successfully.



14) Transformation

After execution, headers are added, delimiter is changed to comma(,), and date format is changed.



15) Unzipped

The zipped file is unzipped successfully.