# PART 1: INTRODUCTION

# Project Overview

This document outlines the automation of the Canada Vigilance Pipeline (CVP) system using AWS services and Python scripts. The project processes adverse drug reaction data from structured input files uploaded to an S3 bucket. It filters the data, converts it into structured formats (JSON, HTML, PDF), and emails the final report to clients. Additionally, it alerts clients if the specified drug name is not found.

## Objective

This project aims to build a scalable, fully automated system that extracts drug and reaction data from structured files in a ZIP archive. It filters the content based on specific drug names, converts the data into JSON, HTML, and PDF reports, and automatically emails the final reports using AWS SES. If any drug names are missing, clients are notified via AWS SNS. The entire pipeline is powered by AWS services including Lambda, S3, EventBridge, and SNS.

# PART 2: TECHNICAL REQUIREMENTS AND OBJECTIVES

## 2.1 AWS Environment Setup:

* Ensure the Lambda environment is properly set up with required libraries.
* Ensure the S3 bucket environment is properly set up with required folders and files.
* PDF File Format: Input and output PDF files must follow specified formats for smooth processing.

## 2.2 Data Integration:

Integrate AWS Lambda with the AWS S3 buckets to read and write data from source and target folders.

## 2.3 Data fetching

* Unzip and read specific files: reactions.txt, report\_drug.txt, report\_drug\_indication.txt, report\_links.txt, and reports.txt.
* Extract records related to the given drug name.
* Create a clean, filtered JSON report with the relevant data.

## 2.4 Data Filtering

* Filter based on the given drug names. Filter the corresponding report data of drug names.
* Filter based on “mah” in the report data

## 2.5 Sending Notification

Send the updated file with matching keywords as attachment to the email.

# PART 3: SCOPE DEFINITION

## 3.1 In-Scope Activities:

The scope includes the following activities and deliverables:

## 3.2 Setup AWS lambda Environments:

Configure Lambda functions.

## 3.3 Integration with AWS S3:

Integration of AWS Lambda with the S3 buckets for both (source) and target folders.

## 3.4 Data Integration:

Integrate AWS Lambda with the AWS S3 buckets to read and write data from staging and target folders.

## 3.5 Data fetching

* Fetch the input data from specified sources (e.g., text files, reports).
* Process the data by cleaning and validating it to ensure correct formatting.
* Remove irrelevant or unwanted data based on predefined rules.
* Structure the filtered data into a JSON format.

## 3.6 Data Filtering

* Convert the structured JSON data into an HTML template based on a predefined layout.
* Convert the HTML template into a PDF format.
* Store the output JSON, HTML, and PDF files in designated storage locations for further use.

## 3.7 Sending Notification

Send the updated file with matching keywords as attachment to the email.

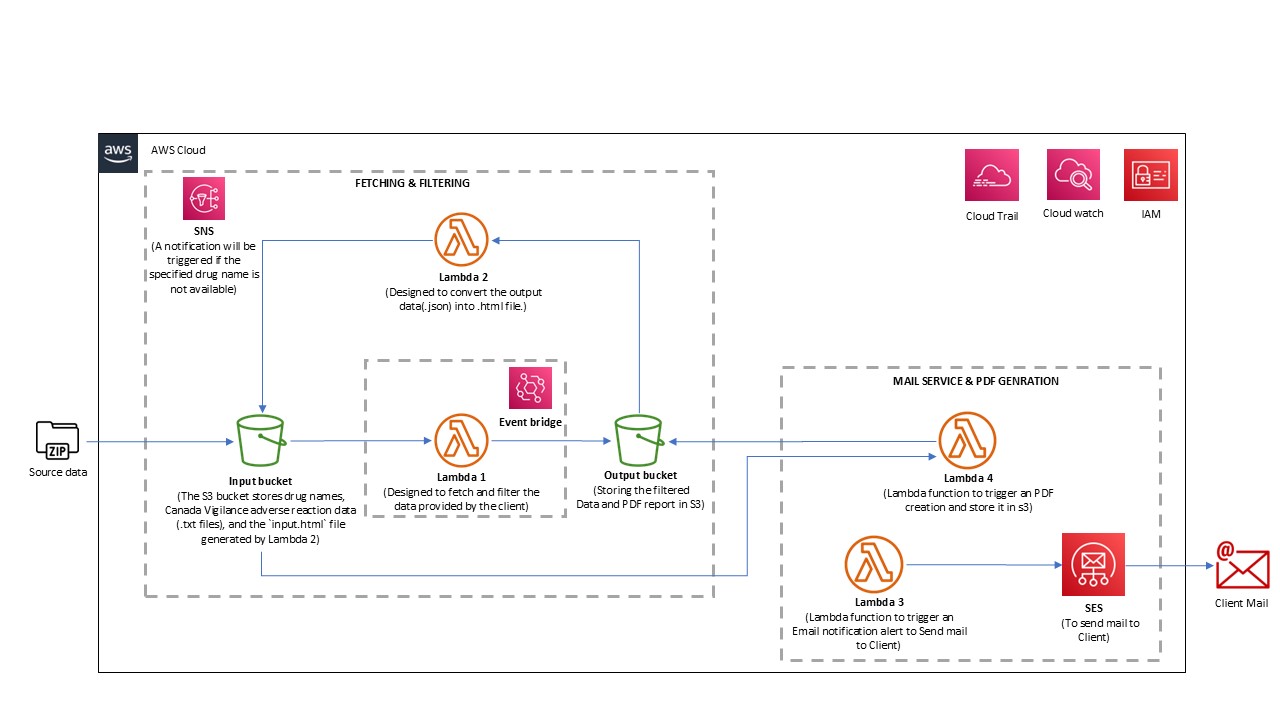
## 3.8 Assumptions:

* The required input data files will be available in the specified storage locations at the beginning of each processing stage.
* The AWS services (Lambda, SES, S3, etc.) will be properly configured and have the necessary permissions for execution.
* The email recipients and required notifications will be clearly identified before starting the process.

## 3.9 Out of Scope:

* Complex Data Filtering
* Manual Intervention for Data Processing
* Modifications to the HTML or PDF templates beyond the standard conversion process
* Customization of email content or format beyond including the PDF as an attachment:

# PART 4: ARCHITECTURE OVERVIEW



## 4.1 AWS Services Utilization:

### 4.1.1 Source Bucket AWS S3:

S3 Input Bucket: This bucket stores the input data files that will be processed, including text files and reports.

### 4.1.2 AWS Lambda:

* Lambda Functions 1, 2 ,3 & 4 are implemented for the processing of the files at different phases.
* These are invoked to trigger different processes. Specific functionalities include initiating the process.

**4.1.3 SES Simple Email Service**

Sending email notification using SES.

**4.1.4. CloudWatch:**

Used for logging and monitoring the AWS services.

**4.1.5. IAM (Identity and Access Management):**

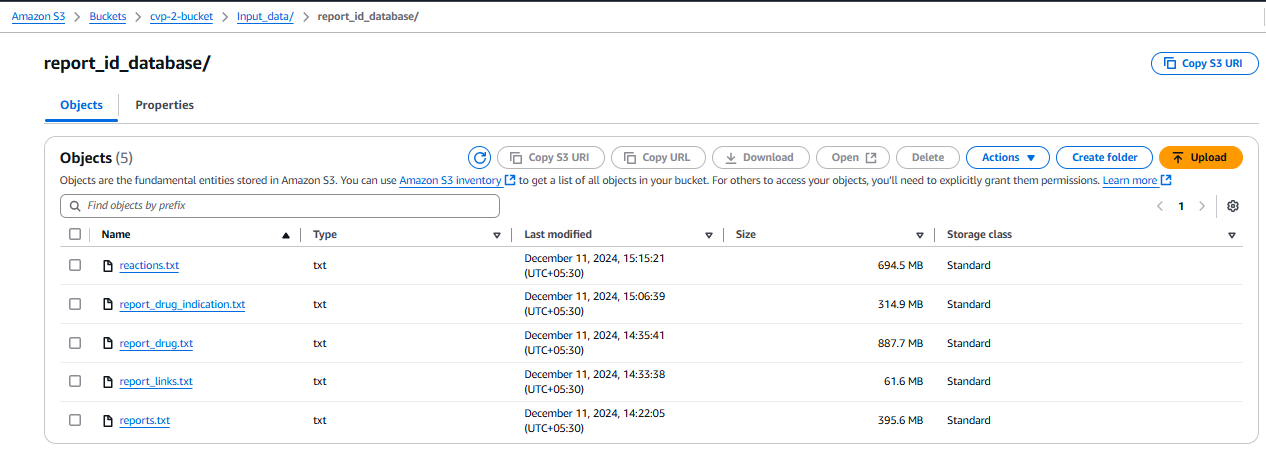
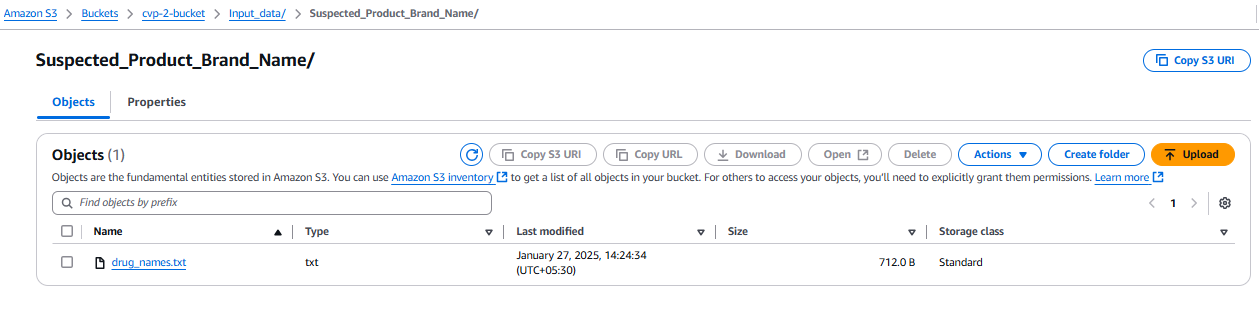
Provides permission controls for accessing AWS resources.

# PART 5: SOURCE DATA MANAGEMENT

This project utilizes multiple S3 buckets to manage the input and output data throughout the processing stages. The source bucket stores the raw data, while the output buckets store the results after processing. The main objective is to fetch, process, and convert the data into JSON, HTML, and PDF formats.

## 5.1 S3 -3 Bucket

* The source bucket stores the raw input data that will be processed.
* Input data includes text files such as reactions.txt, report\_drug\_indication.txt, report\_drug.txt, report\_links.txt, and reports.txt.  
  Also it stores the suspected drug names list (txt file).
* Data Type: Text files (TXT)
* Data Domain: Pharmaceutical industry
* Frequency: Data will be updated as per project needs
* File Size: Varies based on the size of the text files

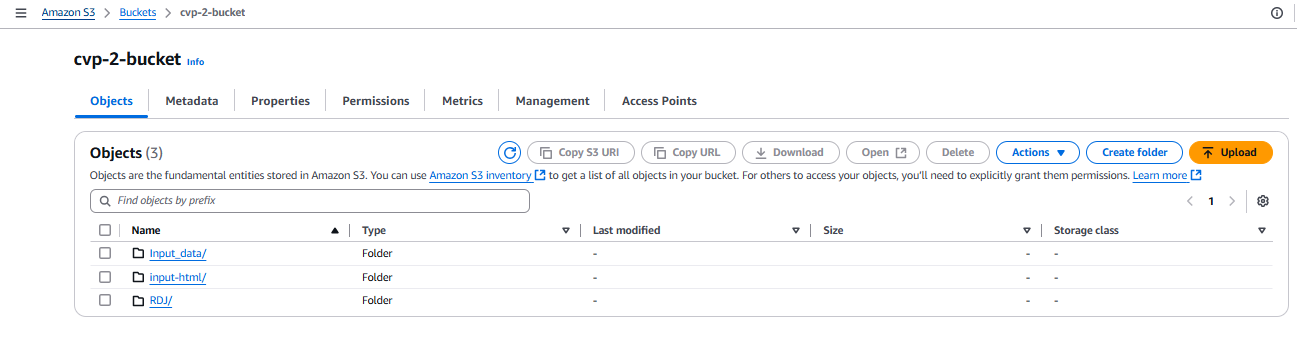
# PART 6: INFRASTRUCTURE SETUP

## 6.1 Create S3 Buckets

### 6.1.1 Create Bucket cvp-2-bucket:

**Bucket Name:** cvp-2-bucket

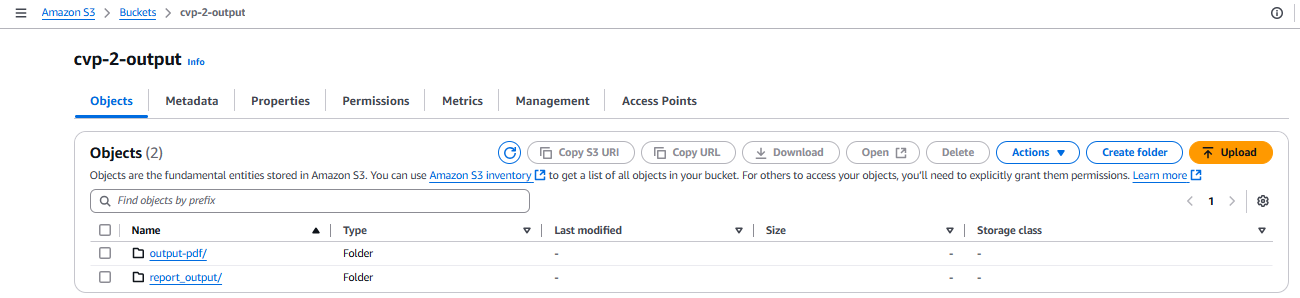
* **Purpose:** Stores input data files and intermediate HTML templates.
* **Folder Structure:**
  + Input\_data/report\_id\_database/ — Contains raw data files (e.g., reactions, reports).
  + Input\_data/Suspected\_Product\_Brand\_Name/ — Contains the drug names for filtering.
  + input-html/ — Stores generated HTML files from JSON reports.



### 6.1.2 Create Bucket cvp-2-output:

**Bucket Name:** cvp-2-output

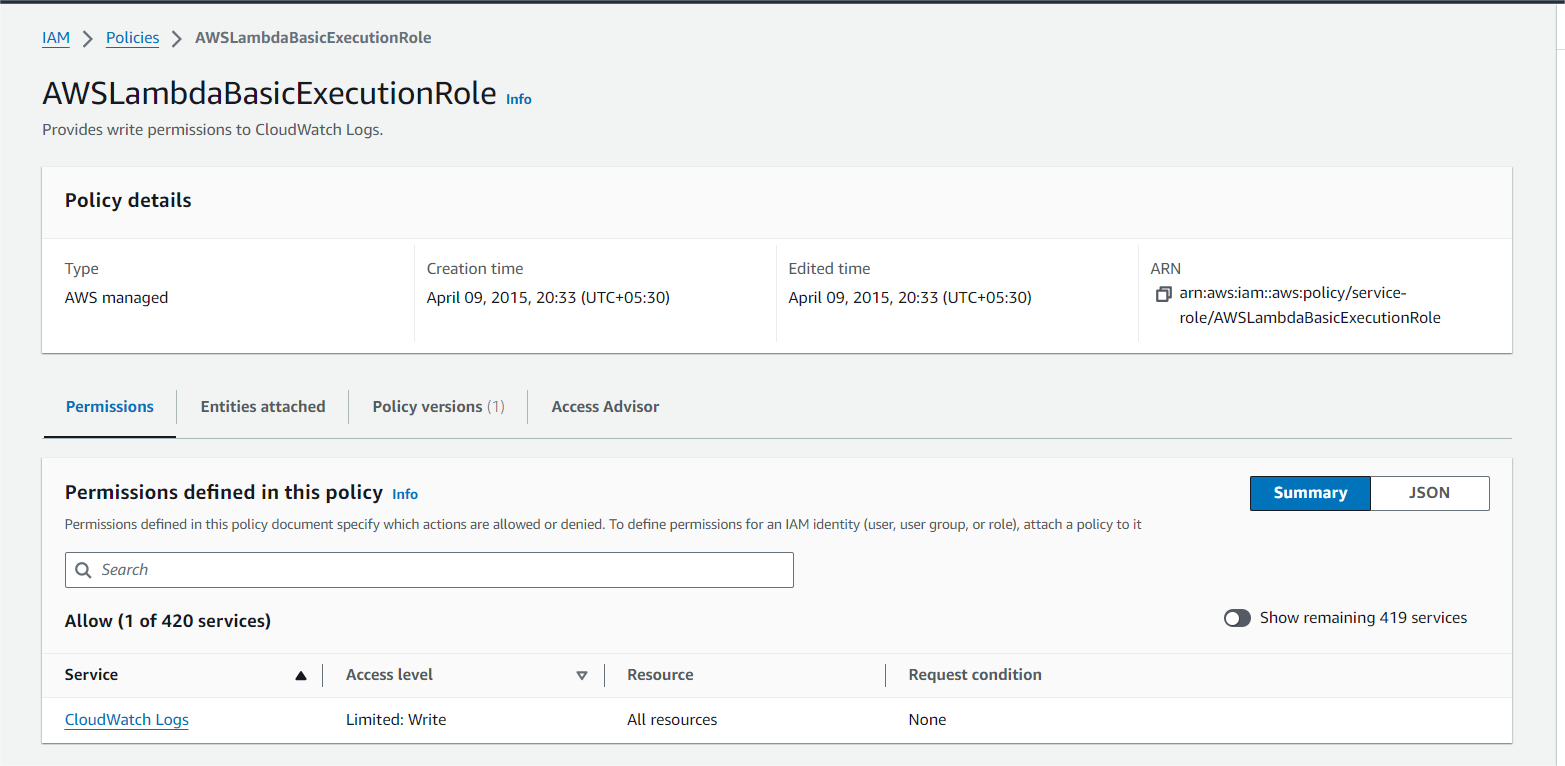
* **Purpose:** Stores final processed output (JSON reports and PDF files).
* **Folder Structure:**
  + report\_output/ — Stores filtered JSON reports.
  + output-pdf/ — Stores converted PDF reports.

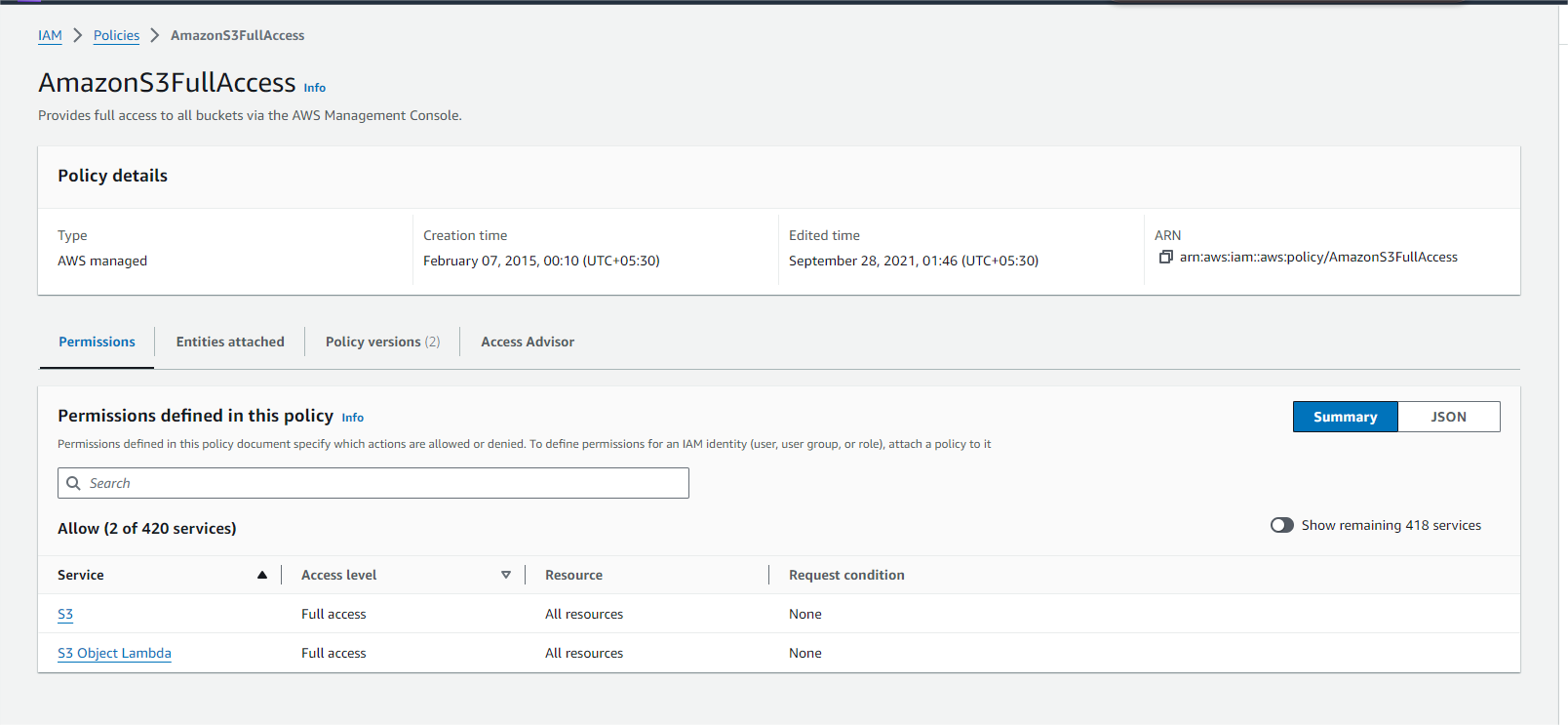


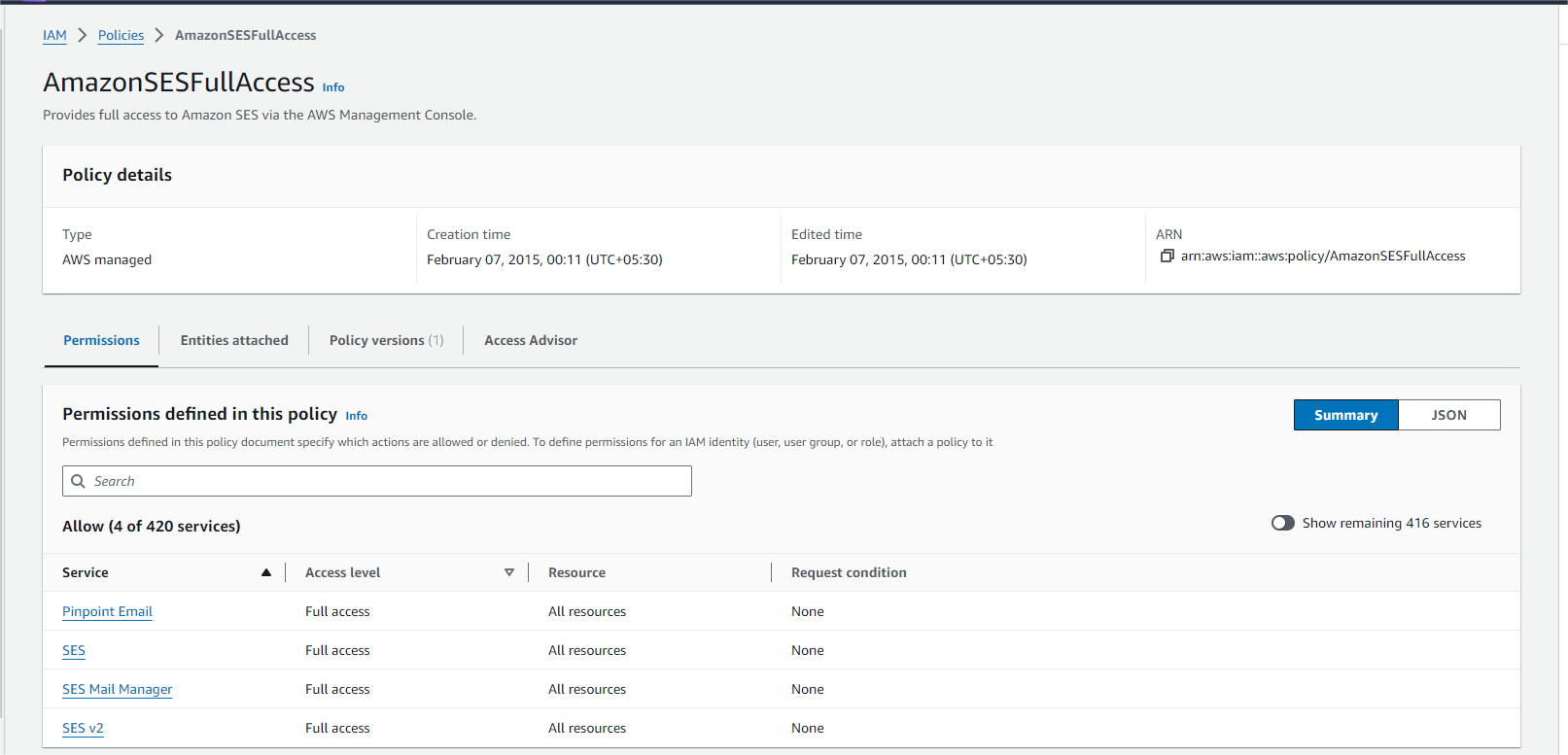
## 6.2 Create IAM Roles

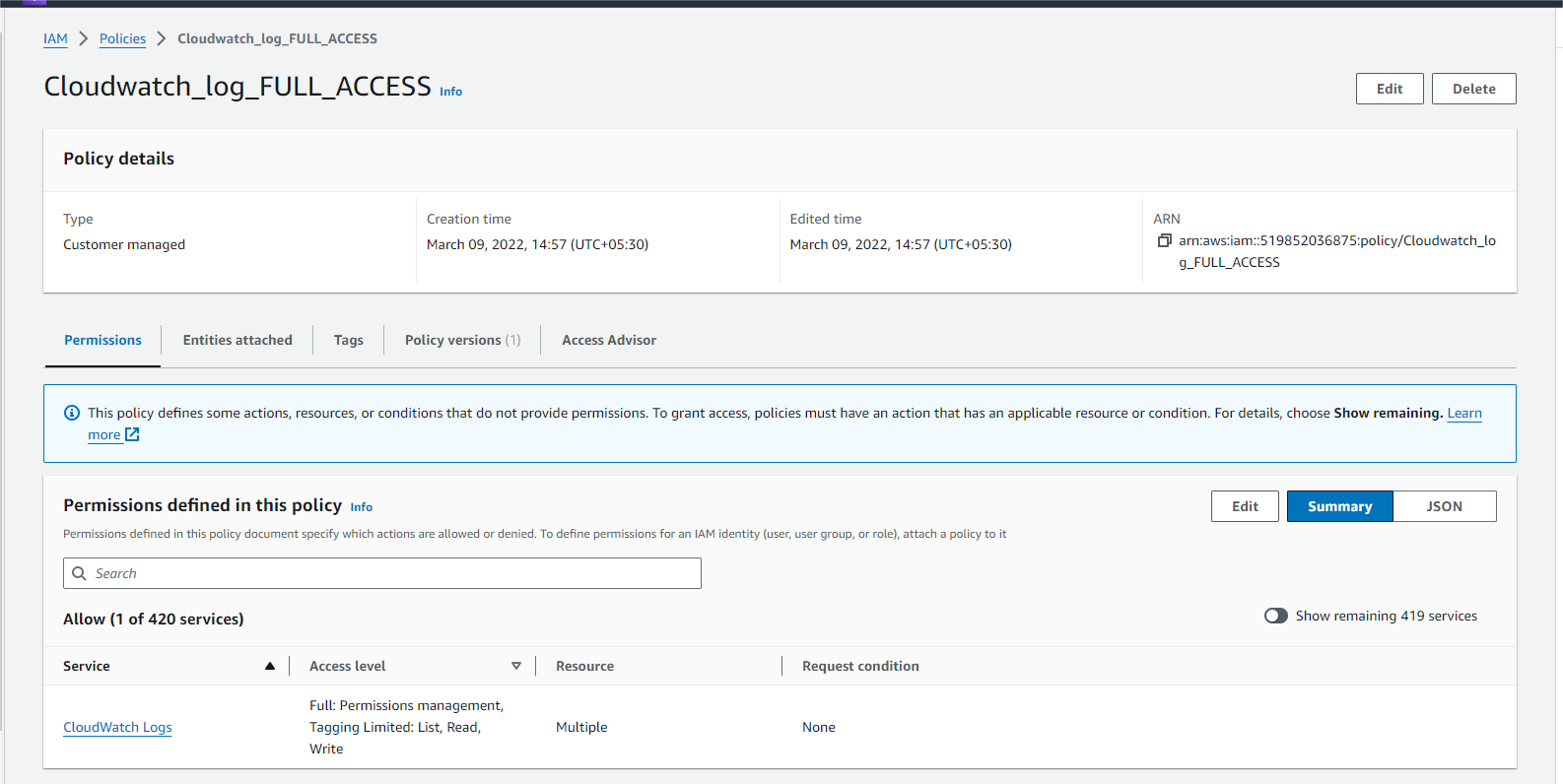
### 6.2.1 Create Role for Lambda Functions:

* Role Name: LambdaExecutionRole
* Permissions: Attach policies for S3 access, SES access, Lambda execution, and CloudWatch logs.









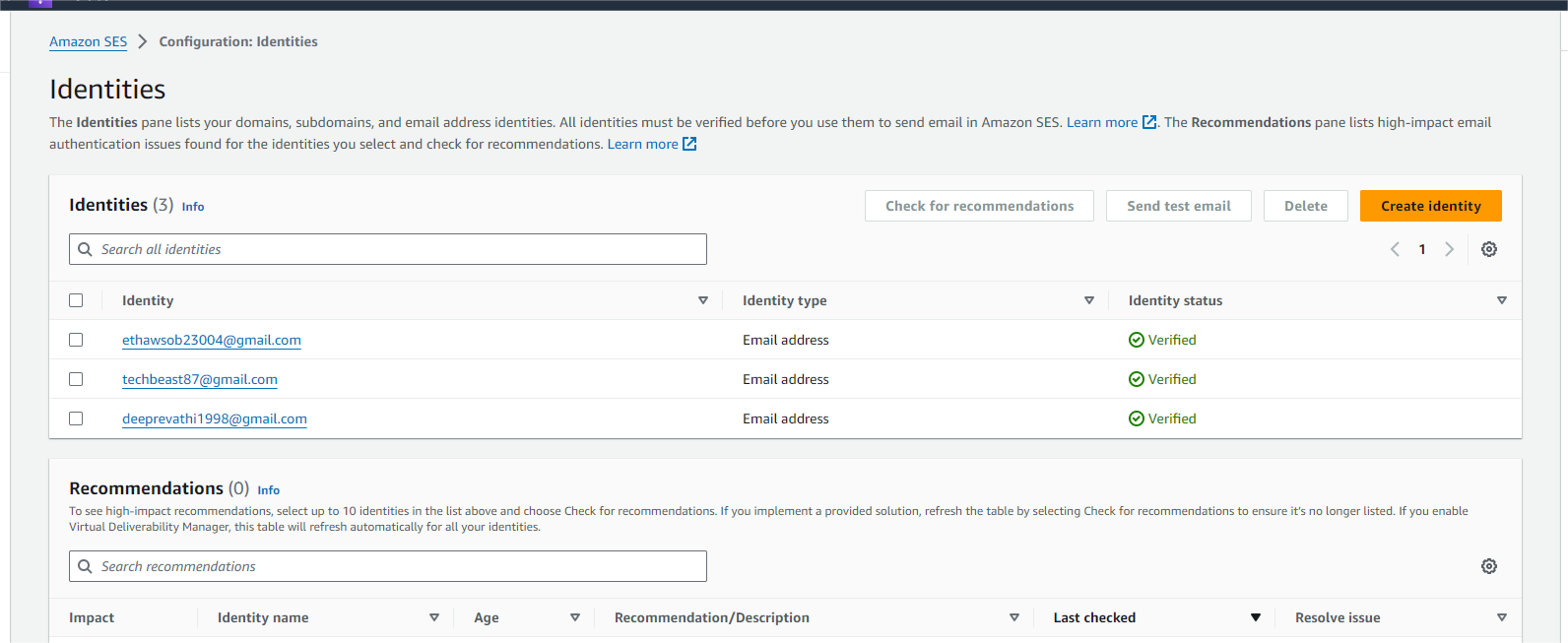
## 6.3 Configure SES

### 6.3.1 Verify Email Addresses:

* Sender Email: Verify the email address from which notifications will be sent.
* Recipient Email: Verify the email address to which notifications will be sent.

### 6.3.2 Test Email Notifications:

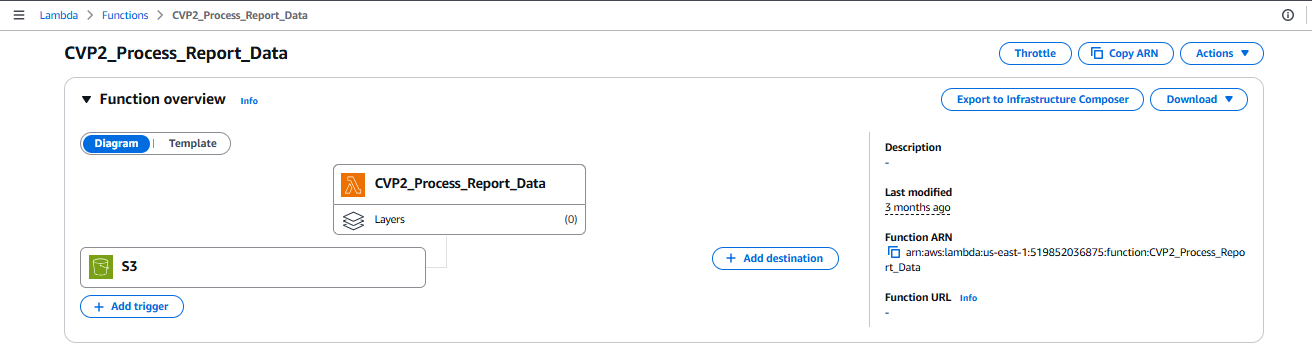
* Send Test Emails: Ensure that the configuration is correct and emails can be sent.



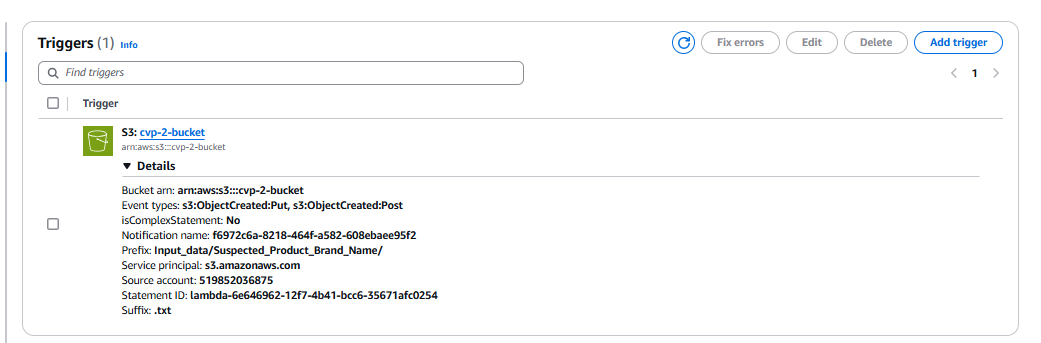
## 6.4 Configure Lambda Functions

### 6.4.1 CVP2\_Process\_Report\_Data:

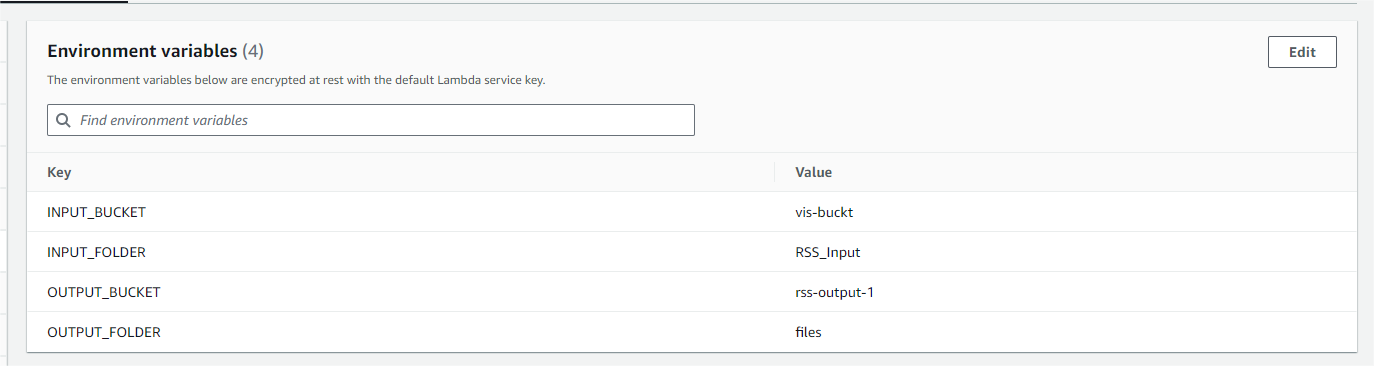
* Create Lambda Function:
* Function Name: CVP2\_Process\_Report\_Data
* Runtime: Python 3.13
* Role: default role
* Code: Upload the Python script that processes input files and filters data based on drug names.



* Set Triggers to the function.
* Trigger for CVP2\_Process\_Report\_Data:
* Navigate to S3 Service:
* Go to the S3 section in the AWS Management Console.
* Create Notification for Bucket:
* Select the cvp-2-bucket.
* Go to the Properties tab and scroll to Event notifications.
* Create Event Notification:
* Name: CVP-2.
* Event Type: Choose All object create events.
* Destination: Select Lambda Function and choose CVP2\_Process\_Report\_Data.
* Save the notification settings.



* Adding Environment Variables to Lambda Functions
* Navigate to AWS Lambda:
* Go to the AWS Management Console and open Lambda.
* Select Your Lambda Function:
* Go to the Environment Variables Section:
* Scroll down to the Environment variables section under your function's configuration.
* Add Environment Variables:
* Click Edit in the Environment variables section.
* Click Add environment variable and enter the key-value pairs for your environment variables.



* Detailed Explanation of Python Code
* Imports:

import boto3

import json

import logging

from collections import defaultdict

import time

from concurrent.futures import ThreadPoolExecutor

from datetime import datetime

import io

import os

* boto3: AWS SDK for Python, used to interact with S3 and SNS services.
* json: For working with JSON data formats.
* logging: Used to log messages for debugging and monitoring.
* collections.defaultdict: Simplifies dictionary value handling.
* time: Tracks execution durations and timestamps.
* concurrent.futures.ThreadPoolExecutor: Reads multiple S3 files in parallel.
* datetime: Handles date parsing and formatting.
* io: Reads in-memory file streams.
* os: Fetches environment variables.
* AWS S3 Client Initialization:

# Initialize S3 client

s3\_client = boto3.client('s3')

# Initialize SNS client

sns\_client = boto3.client('sns')

* s3\_client = boto3.client('s3'): Allows reading from and writing to S3.
* sns\_client = boto3.client('sns'): Sends notifications via AWS SNS.
* **Environment Variable Configuration:**

# SNS topic ARN (replace with your actual topic ARN)

sns\_topic\_arn = os.getenv("SNS\_TOPIC\_ARN")

# Input and output S3 buckets

input\_bucket = os.getenv("INPUT\_BUCKET")

output\_bucket = os.getenv("OUTPUT\_BUCKET")

# File paths in S3

drug\_names\_file = os.getenv("DRUG\_NAMES\_FILE\_PATH")

report\_drug\_file = os.getenv("REPORT\_DRUG\_FILE\_PATH")

reports\_file = os.getenv("REPORTS\_FILE\_PATH")

reactions\_file = os.getenv("REACTIONS\_FILE\_PATH")

report\_links\_file = os.getenv("REPORT\_LINKS\_FILE\_PATH")

report\_drug\_indication\_file = os.getenv("REPORT\_DRUG\_INDICATION\_FILE\_PATH")

* Loads file paths and topic ARNs for:
  + - Input S3 bucket
    - Output S3 bucket
    - SNS Topic ARN for alerts
    - File keys: drug names, reports, reactions, drug indications, etc.
* Helper Functions:

# Function to read files from S3

def read\_s3\_file(bucket, key):

try:

logging.info(f"Attempting to read S3 file {key} from bucket {bucket}...")

response = s3\_client.get\_object(Bucket=bucket, Key=key)

logging.info(f"Successfully read S3 file {key} from bucket {bucket}.")

return response['Body'].read().decode('utf-8').splitlines()

except Exception as e:

logging.error(f"Error reading S3 file {key} from bucket {bucket}: {e}")

return []

# converting date format

def convert\_date\_format(date\_str):

try:

# Convert the date string from 'DD-MMM-YY' to 'YYYY-MM-DD'

date\_obj = datetime.strptime(date\_str, "%d-%b-%y")

return date\_obj.strftime("%Y-%m-%d")

except ValueError:

# Return the original string if it doesn't match the expected format

return date\_str

# coverting 1 to yes, 2 to no

def convert\_to\_yes\_no(value):

if value == "1":

return "Yes"

elif value == "2":

return "No"

else:

return value # In case there are other values, return the original value

# cleaning data

def clean\_string(value):

"""Removes unwanted escape sequences and extra quotes from a JSON string."""

if not isinstance(value, str):

return "" # Return empty string if value is not a string

return value.strip('"').replace('\\"', '')

* + read\_s3\_file(bucket, key): Reads a text file from S3 and splits it into lines.
  + convert\_date\_format(date\_str): Converts dates from 'DD-MMM-YY' to 'YYYY-MM-DD'.
  + convert\_to\_yes\_no(value): Converts coded responses ('1' → Yes, '2' → No).
  + clean\_string(value): Removes quotes and escape sequences from text fields.
* Drug Name Parsing:

# Step 1: Parse drug names from file

def parse\_drug\_names(file\_content):

logging.info("Parsing drug names...")

drug\_names = set() # Use a set to store unique drug names

for line in file\_content:

drug\_name = line.strip().lower() # Normalize drug name to lowercase

if drug\_name: # Ignore blank lines

drug\_names.add(drug\_name) # Add to set (duplicates automatically removed)

logging.info(f"Parsed {len(drug\_names)} unique drug names.")

return list(drug\_names) # Convert back to list if needed

* + parse\_drug\_names(file\_content):
  + Removes whitespace, lowers case, filters empty lines.
  + Returns a list of unique drug names.
* Drug Matching and ID Extraction:

# Step 2: Locate REPORT\_IDs corresponding to drug names

def find\_report\_ids(drug\_names, report\_drug\_content):

logging.info(f"Finding REPORT\_IDs for {len(drug\_names)} drug names...")

report\_ids = defaultdict(list)

missing\_drug\_names = set(drug\_names) # Start by assuming all drug names are missing

# Convert the list of drug names to a set for faster lookup

drug\_names\_set = set(drug\_names)

# Process each line in the report

for line in report\_drug\_content:

fields = line.split('$')

if len(fields) > 1:

drug\_name = clean\_string(fields[3]).strip().lower() # Normalize drug name to lowercase

report\_id = clean\_string(fields[1]).strip()

# Check if any drug name is a substring in the field (fields[3])

for name in drug\_names\_set:

if name.lower() in drug\_name: # Check if the drug name is a substring of fields[3]

report\_ids[report\_id].append(fields)

# If this drug name matches, remove it from the missing list

if name in missing\_drug\_names:

missing\_drug\_names.remove(name)

break # Stop checking other drug names if a match is found

logging.info(f"Found {len(report\_ids)} report IDs matching the drug names.")

# If there are missing drugs, send SNS notification

if missing\_drug\_names:

send\_missing\_drug\_notification(missing\_drug\_names)

return report\_ids

* + find\_report\_ids(drug\_names, report\_drug\_content):
  + Loops through the report\_drug.txt file.
  + Matches each drug name with report entries (case-insensitive, substring match).
  + Extracts and maps matching REPORT\_IDs.
  + Sends SNS notification if some drug names are missing.
* Sending sns notification :

# Function to send SNS notification about missing drugs

def send\_missing\_drug\_notification(missing\_drug\_names):

# Create the message body

header\_message = "The following drugs from the provided list were not found in the report data:\n\n"

missing\_drug\_message = header\_message + "\n".join(missing\_drug\_names)

try:

# Publish to SNS

response = sns\_client.publish(

TopicArn=sns\_topic\_arn,

Message=missing\_drug\_message,

Subject="Missing Drug Names Notification"

)

logging.info(f"SNS Notification sent successfully. Message ID: {response['MessageId']}")

except Exception as e:

logging.error(f"Error sending SNS notification: {e}")

* Filter REPORT\_IDs by Source:

def filter\_report\_ids\_by\_source(report\_ids, reports\_content):

logging.info(f"Filtering REPORT\_IDs based on SOURCE\_ENG...")

# Only consider the REPORT\_IDs in the 374 found earlier

report\_ids\_set = set(report\_ids.keys()) # Convert 374 report IDs to a set

report\_ids\_to\_remove = set()

for line in reports\_content:

fields = line.split('$')

if len(fields) > 37: # Ensure fields[37] (SOURCE\_ENG) exists

report\_id = clean\_string(fields[0]).strip()

source\_eng = clean\_string(fields[37]).strip().lower()

# Check SOURCE\_ENG for "mah" only for relevant REPORT\_IDs

if report\_id in report\_ids\_set and "mah" in source\_eng:

report\_ids\_to\_remove.add(report\_id)

# Filter out REPORT\_IDs to remove

filtered\_report\_ids = {rid: details for rid, details in report\_ids.items() if rid not in report\_ids\_to\_remove}

logging.info(f"Initial REPORT\_IDs: {len(report\_ids)}")

logging.info(f"Excluded REPORT\_IDs: {len(report\_ids\_to\_remove)}")

logging.info(f"Remaining REPORT\_IDs: {len(filtered\_report\_ids)}")

return filtered\_report\_ids

* + filter\_report\_ids\_by\_source(report\_ids, reports\_content):
  + Excludes reports whose SOURCE\_ENG is 'mah'.
  + Logs counts before and after filtering.
* Data Extraction from All Files:

def extract\_report\_data(report\_ids, reports\_content, reactions\_content, report\_drug\_indication\_content,

report\_links\_content, report\_drug\_content):

logging.info("Extracting report data from reference files...")

report\_data = {}

# Step 1: Process reports.txt first

for line in reports\_content:

fields = line.split('$')

if len(fields) > 1:

report\_id = clean\_string(fields[0]).strip()

if report\_id not in report\_ids:

continue # Skip if the report\_id is not in the report\_ids

report\_data[report\_id] = {

'report\_no': clean\_string(fields[1]),

'version\_no': clean\_string(fields[2]),

'datreceived': convert\_date\_format(clean\_string(fields[3])),

'datintreceived': convert\_date\_format(clean\_string(fields[4])),

'mah\_no': clean\_string(fields[5]),

'report\_type\_eng': clean\_string(fields[7]),

'gender\_eng': clean\_string(fields[10]),

'age': clean\_string(fields[12]),

'age\_unit\_eng': clean\_string(fields[14]),

'outcome\_eng': clean\_string(fields[17]),

'weight': clean\_string(fields[19]),

'weight\_unit\_eng': clean\_string(fields[20]),

'height': clean\_string(fields[22]),

'height\_unit\_eng': clean\_string(fields[23]),

'seriousness\_eng': clean\_string(fields[26]),

'death': convert\_to\_yes\_no(clean\_string(fields[28])),

'disability': convert\_to\_yes\_no(clean\_string(fields[29])),

'congenital\_anomaly': convert\_to\_yes\_no(clean\_string(fields[30])),

'life\_threatening': convert\_to\_yes\_no(clean\_string(fields[31])),

'hospitalization': convert\_to\_yes\_no(clean\_string(fields[32])),

'other\_medically\_imp\_cond': convert\_to\_yes\_no(clean\_string(fields[33])),

'reporter\_type\_eng': clean\_string(fields[34]),

'source\_eng': clean\_string(fields[37])

}

# Step 2: Process reactions.txt

for line in reactions\_content:

fields = line.split('$')

report\_id = clean\_string(fields[1]).strip()

if report\_id not in report\_ids:

continue # Skip if the report\_id is not in the report\_ids

pt\_name\_eng = clean\_string(fields[5])

meddra\_version = clean\_string(fields[9])

duration = clean\_string(fields[2])

duration\_unit\_eng = clean\_string(fields[3])

if 'pt\_name\_eng' in report\_data[report\_id]:

report\_data[report\_id]['pt\_name\_eng'] += ', ' + pt\_name\_eng

report\_data[report\_id]['meddra\_version'] += ', ' + meddra\_version

report\_data[report\_id]['duration'] += ', ' + duration

report\_data[report\_id]['duration\_unit\_eng'] += ', ' + duration\_unit\_eng

else:

report\_data[report\_id]['pt\_name\_eng'] = pt\_name\_eng

report\_data[report\_id]['meddra\_version'] = meddra\_version

report\_data[report\_id]['duration'] = duration

report\_data[report\_id]['duration\_unit\_eng'] = duration\_unit\_eng

# Step 3: Process report\_links.txt

matched\_ids = set() # To track report\_ids found in report\_links.txt

for line in report\_links\_content:

fields = line.split('$')

report\_id = clean\_string(fields[1]).strip()

record\_type\_eng = clean\_string(fields[2]).strip()

report\_link\_no = clean\_string(fields[4]).strip()

# Process only if the report\_id is in report\_ids

if report\_id in report\_ids:

# Initialize the report\_data entry if it's not already present

if report\_id not in report\_data:

report\_data[report\_id] = {}

# Assign values from the line

report\_data[report\_id]['record\_type\_eng'] = record\_type\_eng

report\_data[report\_id]['report\_link\_no'] = report\_link\_no

# Mark this report\_id as matched

matched\_ids.add(report\_id)

# Handle report\_ids that were not matched

for report\_id in report\_ids:

if report\_id not in matched\_ids:

# Ensure only missing fields are updated without overwriting existing data

if report\_id not in report\_data:

report\_data[report\_id] = {} # Initialize if not present

report\_data[report\_id].setdefault('record\_type\_eng', 'No duplicate or linked report')

report\_data[report\_id].setdefault('report\_link\_no', 'No duplicate or linked report')

# Step 4: Process report\_drug.txt

drug\_names\_dict = {}

for line in report\_drug\_content:

fields = line.split('$')

if len(fields) > 1:

report\_id = clean\_string(fields[1]).strip()

if report\_id not in report\_ids:

continue # Skip if the report\_id is not in the report\_ids

drug\_name = clean\_string(fields[3])

drug\_involvement = clean\_string(fields[4])

route\_admin = clean\_string(fields[6])

unit\_dose\_qty = clean\_string(fields[8])

dose\_unit\_eng = clean\_string(fields[9])

freq\_time\_unit\_eng = clean\_string(fields[15])

therapy\_duration = clean\_string(fields[17])

therapy\_duration\_unit\_eng = clean\_string(fields[18])

dosageform\_eng = clean\_string(fields[20])

# Initialize drug\_names\_dict and report\_data

if report\_id not in drug\_names\_dict:

drug\_names\_dict[report\_id] = []

drug\_names\_dict[report\_id].append(drug\_name) # Add drug name to the list for this report\_id

if report\_id not in report\_data:

report\_data[report\_id] = {}

if 'drug\_name' in report\_data[report\_id]:

report\_data[report\_id]['drug\_name'] += ', ' + drug\_name

else:

report\_data[report\_id]['drug\_name'] = drug\_name

# Append or initialize for 'drug\_involvement'

if 'drug\_involvement' in report\_data[report\_id]:

report\_data[report\_id]['drug\_involvement'] += ', ' + drug\_involvement

else:

report\_data[report\_id]['drug\_involvement'] = drug\_involvement

# Append or initialize for 'route\_admin'

if 'route\_admin' in report\_data[report\_id]:

report\_data[report\_id]['route\_admin'] += ', ' + route\_admin

else:

report\_data[report\_id]['route\_admin'] = route\_admin

# Append or initialize for 'unit\_dose\_qty'

if 'unit\_dose\_qty' in report\_data[report\_id]:

report\_data[report\_id]['unit\_dose\_qty'] += ', ' + unit\_dose\_qty

else:

report\_data[report\_id]['unit\_dose\_qty'] = unit\_dose\_qty

# Append or initialize for 'dose\_unit\_eng'

if 'dose\_unit\_eng' in report\_data[report\_id]:

report\_data[report\_id]['dose\_unit\_eng'] += ', ' + dose\_unit\_eng

else:

report\_data[report\_id]['dose\_unit\_eng'] = dose\_unit\_eng

# Append or initialize for 'freq\_time\_unit\_eng'

if 'freq\_time\_unit\_eng' in report\_data[report\_id]:

report\_data[report\_id]['freq\_time\_unit\_eng'] += ', ' + freq\_time\_unit\_eng

else:

report\_data[report\_id]['freq\_time\_unit\_eng'] = freq\_time\_unit\_eng

# Append or initialize for 'therapy\_duration'

if 'therapy\_duration' in report\_data[report\_id]:

report\_data[report\_id]['therapy\_duration'] += ', ' + therapy\_duration

else:

report\_data[report\_id]['therapy\_duration'] = therapy\_duration

# Append or initialize for 'therapy\_duration\_unit\_eng'

if 'therapy\_duration\_unit\_eng' in report\_data[report\_id]:

report\_data[report\_id]['therapy\_duration\_unit\_eng'] += ', ' + therapy\_duration\_unit\_eng

else:

report\_data[report\_id]['therapy\_duration\_unit\_eng'] = therapy\_duration\_unit\_eng

# Append or initialize for 'therapy\_duration\_unit\_eng'

if 'dosageform\_eng' in report\_data[report\_id]:

report\_data[report\_id]['dosageform\_eng'] += ', ' + dosageform\_eng

else:

report\_data[report\_id]['dosageform\_eng'] = dosageform\_eng

# Step 5: Process report\_drug\_indication.txt after all other files

for line in report\_drug\_indication\_content:

fields = line.split('$')

if len(fields) > 4:

report\_id = clean\_string(fields[1]).strip()

drug\_name\_eng = clean\_string(fields[3]).strip().lower()

indication = clean\_string(fields[4]).strip()

if report\_id not in report\_ids:

continue

# Get the list of drug names for the current report\_id

drug\_names\_for\_report = drug\_names\_dict.get(report\_id, [])

# Initialize indication\_eng if it doesn't exist

if 'indication\_eng' not in report\_data[report\_id]:

# Placeholder for each drug: a space separated by commas

report\_data[report\_id]['indication\_eng'] = ' , ' \* (len(drug\_names\_for\_report) - 1) + ' '

# Find the drug index and assign the correct indication to that index

for index, drug\_name in enumerate(drug\_names\_for\_report):

# Match the drug name with its indication if it exists

if drug\_name\_eng == drug\_name.lower():

indication\_list = report\_data[report\_id]['indication\_eng'].split(', ')

indication\_list[index] = indication.strip() # Assign the indication to the correct drug

report\_data[report\_id]['indication\_eng'] = ', '.join(indication\_list)

return report\_data

* + extract\_report\_data():
  + From reports.txt: Basic patient and report info.
  + From reactions.txt: Adds PT name, MedDRA version, and reaction duration.
  + From report\_links.txt: Adds duplicate/linking report data.
  + From report\_drug.txt: Adds drug details (name, involvement, dosage, route, etc.).
  + From report\_drug\_indication.txt: Matches drug names to indications per report.
  + Combines all fields into a single dictionary per REPORT\_ID.
* Check for Already Processed Reports:

def get\_existing\_report\_ids\_from\_s3():

existing\_report\_ids = set()

try:

# List all objects in the 'report\_output/' folder

response = s3\_client.list\_objects\_v2(Bucket=output\_bucket, Prefix='report\_output/')

if 'Contents' in response:

for obj in response['Contents']:

file\_key = obj['Key']

if file\_key.endswith('.json'):

# Read the JSON file

file\_obj = s3\_client.get\_object(Bucket=output\_bucket, Key=file\_key)

file\_data = json.loads(file\_obj['Body'].read().decode('utf-8'))

# Extract report numbers from the JSON file

for record in file\_data:

if 'report\_no' in record:

existing\_report\_ids.add(str(record['report\_no']).strip().lower()) # Normalize to string (strip spaces, lowercase)

logging.info(f"Existing report numbers from S3: {existing\_report\_ids}")

except Exception as e:

logging.error(f"Error while retrieving existing report numbers from S3: {e}")

return existing\_report\_ids

* + get\_existing\_report\_ids\_from\_s3():
  + Lists all .json files in report\_output/.
  + Extracts and returns existing report\_nos.
* Filter Out Duplicate Reports:

def filter\_new\_report\_data(report\_data, existing\_report\_ids):

new\_report\_data = {}

# Iterate over the report data and check if the report\_no is already in the existing reports

for report\_id, data in report\_data.items():

report\_no = str(data.get('report\_no', '')).strip().lower() # Normalize report\_no to string (strip spaces, lowercase)

if report\_no not in existing\_report\_ids:

new\_report\_data[report\_id] = data # Add this report to new report data if it's not in the existing reports

logging.info(f"New report found: {report\_no}") # Log the new report number

else:

logging.info(f"Duplicate report found: {report\_no}") # Log duplicate report number

logging.info(f"New report data: {new\_report\_data.keys()}") # Log keys of new reports

return new\_report\_data

* + filter\_new\_report\_data(report\_data, existing\_report\_ids):
  + Compares new entries to previously processed ones.
  + Keeps only new, unprocessed reports.
* Generate and Upload JSON Output:

def generate\_json\_output(report\_data):

"""

Generate and upload the final JSON output to S3.

Only proceeds if there are new reports to upload.

"""

if not report\_data:

logging.info("No new reports found. Skipping JSON generation and upload.")

return

logging.info("Generating JSON output...")

final\_data = []

for report\_id, data in report\_data.items():

final\_data.append({

"report\_no": data.get('report\_no', ''),

"version\_no": data.get('version\_no', ''),

"datintreceived": data.get('datintreceived', ''),

"datreceived": data.get('datreceived', ''),

"source\_eng": data.get('source\_eng', ''),

"mah\_no": data.get('mah\_no', ''),

"report\_type\_eng": data.get('report\_type\_eng', ''),

"reporter\_type\_eng": data.get('reporter\_type\_eng', ''),

"seriousness\_eng": data.get('seriousness\_eng', ''),

"death": data.get('death', ''),

"disability": data.get('disability', ''),

"congenital\_anomaly": data.get('congenital\_anomaly', ''),

"life\_threatening": data.get('life\_threatening', ''),

"hospitalization": data.get('hospitalization', ''),

"other\_medically\_imp\_cond": data.get('other\_medically\_imp\_cond', ''),

"age": data.get('age', ''),

"age\_unit\_eng": data.get('age\_unit\_eng', ''),

"gender\_eng": data.get('gender\_eng', ''),

"height": data.get('height', ''),

"height\_unit\_eng": data.get('height\_unit\_eng', ''),

"weight": data.get('weight', ''),

"weight\_unit\_eng": data.get('weight\_unit\_eng', ''),

"outcome\_eng": data.get('outcome\_eng', ''),

"record\_type\_eng": data.get('record\_type\_eng', ''),

"report\_link\_no": data.get('report\_link\_no', ''),

"drug\_name": data.get('drug\_name', ''),

"drug\_involvement": data.get('drug\_involvement', ''),

"dosage\_form\_eng": data.get('dosageform\_eng', ''),

"route\_admin": data.get('route\_admin', ''),

"unit\_dose\_qty": data.get('unit\_dose\_qty', ''),

"dose\_unit\_eng": data.get('dose\_unit\_eng', ''),

"freq\_time\_unit\_eng": data.get('freq\_time\_unit\_eng', ''),

"therapy\_duration": data.get('therapy\_duration', ''),

"therapy\_duration\_unit\_eng": data.get('therapy\_duration\_unit\_eng', ''),

"indication\_eng": data.get('indication\_eng', ''),

"pt\_name\_eng": data.get('pt\_name\_eng', ''),

"meddra\_version": data.get('meddra\_version', ''),

"duration": data.get('duration', ''),

"duration\_unit\_eng": data.get('duration\_unit\_eng', '')

})

try:

json\_data = json.dumps(final\_data, indent=4)

timestamp = time.strftime('%d\_%b\_%Y\_%H\_%M\_%S')

output\_file = f"report\_output/reported\_adverse\_reaction\_{timestamp}.json"

s3\_client.put\_object(Bucket=output\_bucket, Key=output\_file, Body=json\_data)

logging.info(f"Successfully uploaded JSON file to S3: {output\_file}")

except Exception as e:

logging.error(f"Error generating or uploading JSON output: {e}")

* generate\_json\_output(report\_data):
* Creates structured JSON list with all extracted fields.
* Uploads the file to S3 with a timestamped filename.
* Skips upload if no new reports exist.
* Main Function Flow:

def main():

logging.info("Starting script execution...")

start\_time = time.time()

# Step 1: Retrieve existing report IDs from previous output files

existing\_report\_ids = get\_existing\_report\_ids\_from\_s3()

# Step 2: Read input files in parallel using ThreadPoolExecutor

with ThreadPoolExecutor() as executor:

# Submit S3 read tasks

futures = {

'drug\_names': executor.submit(read\_s3\_file, input\_bucket, drug\_names\_file),

'report\_drug': executor.submit(read\_s3\_file, input\_bucket, report\_drug\_file),

'reports': executor.submit(read\_s3\_file, input\_bucket, reports\_file),

'reactions': executor.submit(read\_s3\_file, input\_bucket, reactions\_file),

'report\_links': executor.submit(read\_s3\_file, input\_bucket, report\_links\_file),

'report\_drug\_indication': executor.submit(read\_s3\_file, input\_bucket, report\_drug\_indication\_file)

}

# Wait for all read tasks to finish

data = {key: future.result() for key, future in futures.items()}

# Step 3: Parse drug names

logging.info("Starting parsing drugnames...")

drug\_names = parse\_drug\_names(data['drug\_names'])

# Step 4: Find report IDs corresponding to drug names

filter\_report\_ids = find\_report\_ids(drug\_names, data['report\_drug'])

report\_ids = filter\_report\_ids\_by\_source(filter\_report\_ids, data['reports'])

# Step 5: Extract data based on report IDs

report\_data = extract\_report\_data(report\_ids, data['reports'], data['reactions'], data['report\_drug\_indication'],

data['report\_links'], data['report\_drug'])

# Step 6: Filter new report data that is not already in existing reports

new\_report\_data = filter\_new\_report\_data(report\_data, existing\_report\_ids)

# Step 7: Generate and save the JSON output to S3 (if there are new reports)

generate\_json\_output(new\_report\_data)

logging.info(f"Script execution completed in {time.time() - start\_time:.2f} seconds.")

* + main() orchestrates the following:
  + Retrieves previously processed report IDs.
  + Reads all input files in parallel.
  + Parses drug names and extracts report IDs.
  + Filters based on SOURCE\_ENG and compiles data.
  + Detects new reports and uploads their data.
* Lambda Handler:

def lambda\_handler(event, context):

logging.info("Lambda function started.")

# Simulate parallel S3 reading in AWS Lambda by calling main function (in a single thread for Lambda)

main()

return {

'statusCode': 200,

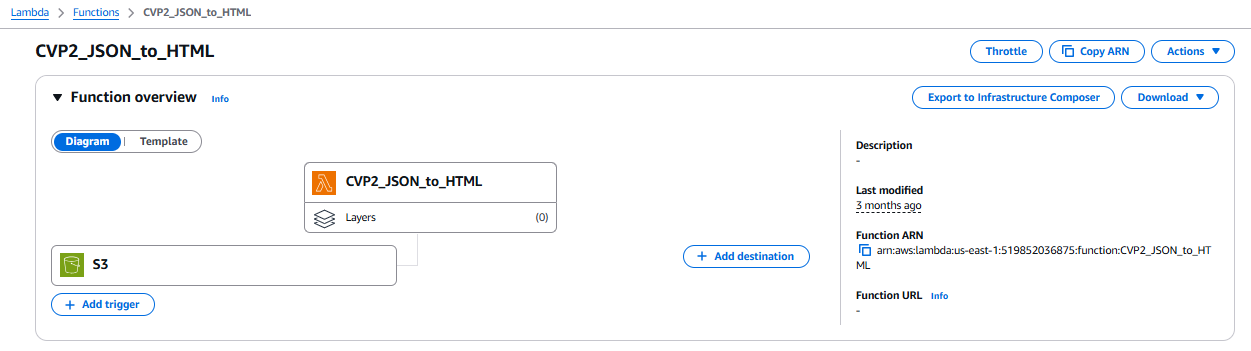
'body': json.dumps('Processing completed successfully.')

}

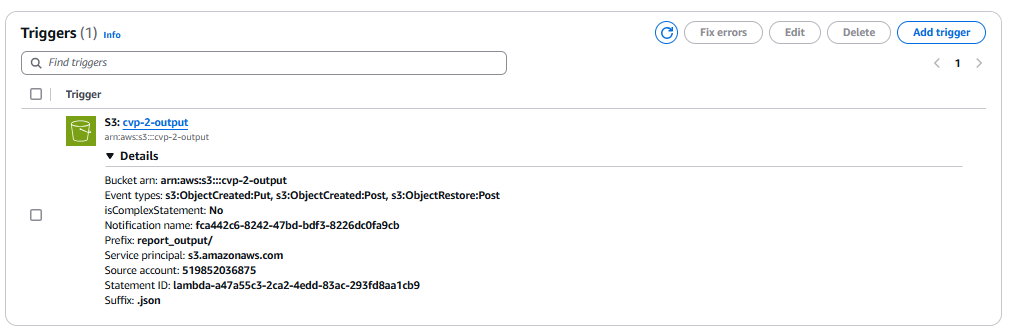
* + lambda\_handler(event, context):
  + Entry point for AWS Lambda execution.
  + Calls main().
  + Returns HTTP 200 with a success message.
* Function Execution:
  + Fetch Input Files: Read drug names, report drug info, reports, reactions, report links, and drug indications from cvp-2-bucket/Input\_data/... using parallel threads.
  + Parse Drug Names: Normalize, deduplicate, and prepare for matching.
  + Match REPORT\_IDs: Match drug names to entries in report drug data.
  + Filter Reports: Exclude records where the SOURCE\_ENG is mah.
  + Extract Full Data: Merge data from five reference files based on matched IDs.
  + Remove Duplicates: Compare new report\_nos with previously saved reports.
  + Generate JSON: Create structured JSON and save to cvp-2-output/report\_output/.
  + Send Alerts: Trigger SNS notification for unmatched drug names.

### 6.4.2 CVP2\_JSON\_to\_HTML:

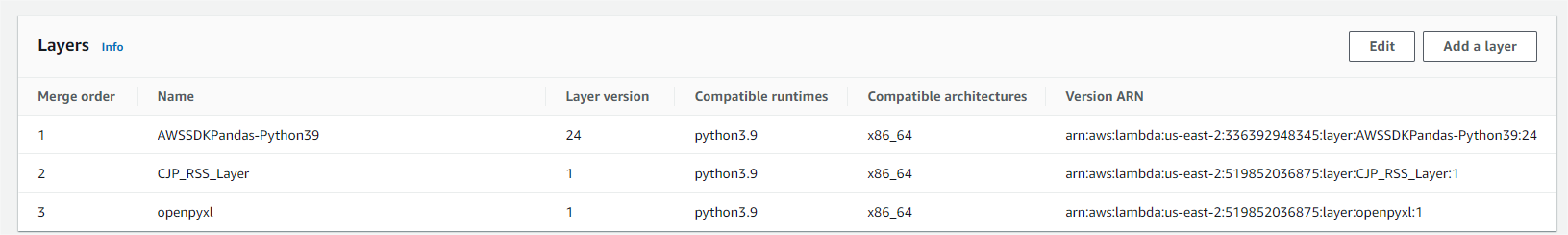
* Create Lambda Function:
* Function Name: CVP2\_JSON\_to\_HTML
* Runtime: Python
* Role: Attach LambdaExecutionRole
* Code: Upload Python script for converting the Json file into a html template.



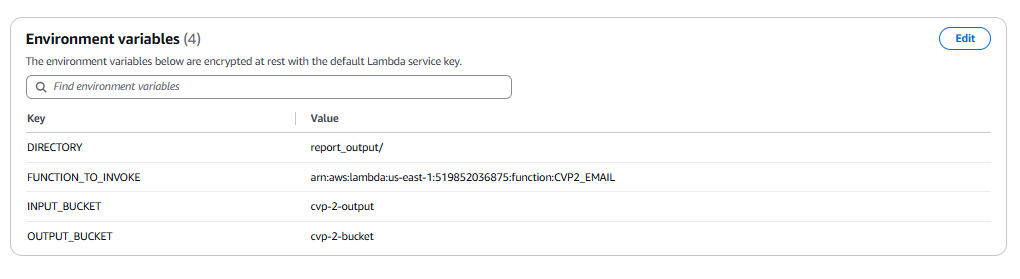
* Set Triggers to the function.
* Trigger for CVP2\_JSON\_to\_HTML2:
* Navigate to S3 Service:
* Go to the S3 section in the AWS Management Console.
* Create Notification for Bucket:
* Select the rss-output-1 bucket.
* Go to the Properties tab and scroll to Event notifications.
* Create Event Notification:
* Name: Trigger\_RSS\_LAM\_02.
* Event Type: Choose All object create events.
* Destination: Select Lambda Function and choose RSS\_LAM\_02.
* Save the notification settings.
* Upload Output: Upload the new file to rss-output-1/files/.



* Adding AWS-Provided Layers to Lambda Functions
* Go to the AWS Management Console and choose Lambda.
* From the list of functions, select the one you want to add the layer to (e.g., RSS\_LAM\_01).
* Scroll down to the Layers section of the Lambda function configuration.
* Example: To add the AWS SDK, search for AWSLambda-Python37-SciPy1x.
* Select the Layer Version: Choose the latest version available for the layer.
* Click Add to attach the AWS layer to your Lambda function.
* Attached Pandas layer from AWS.
* Adding Custom Layers to Lambda Functions
* Package the Custom Layer:
* Zip the Layer:
* Upload the Custom Layer to AWS:
* Create Layer:
* Add the Custom Layer to the Lambda Function:
* Add a Layer and Verify the Layer in the Function:
* Attached Openpyxl and feedparser dependency as a custom layer.



* Adding Environment Variables to Lambda Functions
* Navigate to AWS Lambda:
* Go to the AWS Management Console and open Lambda.
* Select Your Lambda Function:
* Go to the Environment Variables Section:
* Scroll down to the Environment variables section under your function's configuration.
* Add Environment Variables:
* Click Edit in the Environment variables section.
* Click Add environment variable and enter the key-value pairs for your environment variables.



* Detailed Explanation of Python Code
* Imports

import json

import boto3

import os

import time

from datetime import datetime

import logging

* + json: Used for parsing and dumping JSON data.
  + boto3: AWS SDK to interact with services like Lambda and S3.
  + os: Reads environment variables and file paths.
  + time: Generates timestamps.
  + datetime: Offers time and date utilities.
  + logging: Logs information or errors.
* AWS Client Initialization:

# Initialize the Lambda client to invoke other functions

lambda\_client = boto3.client('lambda')

* + lambda\_client = boto3.client('lambda'): Used to invoke another Lambda function (CVP2\_EMAIL).
  + S3 clients are initialized within functions as needed.
* Function: invoke\_cvp2\_email\_lambda()

def invoke\_cvp2\_email\_lambda():

"""Invoke the CVP2\_EMAIL Lambda function."""

try:

# You can pass an event or data to the second Lambda if required, modify as necessary

response = lambda\_client.invoke(

FunctionName=os.getenv("FUNCTION\_TO\_INVOKE"), # Replace with your function ARN

InvocationType="Event", # Asynchronous invocation

Payload=json.dumps({"message": "Triggering CVP2\_EMAIL Lambda function"})

)

# Log the response from Lambda invocation

print(f"Successfully invoked CVP2\_EMAIL Lambda: {response}")

except Exception as e:

print(f"Error invoking CVP2\_EMAIL Lambda: {str(e)}")

* + Invokes another Lambda function defined by the environment variable FUNCTION\_TO\_INVOKE (likely CVP2\_EMAIL).
  + Uses asynchronous invocation (InvocationType='Event').
  + Logs the response or catches any exception.
* Function: load\_json\_from\_s3(bucket\_name, directory)

def load\_json\_from\_s3(bucket\_name, directory):

"""Fetch the latest JSON file from the specified S3 directory."""

s3\_client = boto3.client('s3')

try:

# List all objects in the given directory

response = s3\_client.list\_objects\_v2(Bucket=bucket\_name, Prefix=directory)

files = response.get('Contents', [])

if not files:

print(f"No files found in {directory}.")

return None

# Sort files by last modified date, descending order

files.sort(key=lambda x: x['LastModified'], reverse=True)

# Get the most recent file

latest\_file = files[0]['Key']

print(f"Latest file: {latest\_file}")

# Fetch the latest file from S3

file\_obj = s3\_client.get\_object(Bucket=bucket\_name, Key=latest\_file)

file\_content = file\_obj['Body'].read().decode('utf-8')

# Parse JSON content

json\_data = json.loads(file\_content)

return json\_data

except Exception as e:

print(f"Error loading JSON from S3: {e}")

return None

* + Lists files in a given directory inside an S3 bucket.
  + Sorts them by LastModified date and picks the latest.
  + Loads the content of the selected JSON file.
  + Returns parsed JSON data or logs error if retrieval fails.
* Function: split\_comma\_values(value)

def split\_comma\_values(value):

"""Helper function to split comma-separated values and remove placeholders."""

placeholders = ["{{health\_product\_role}}", "{{dosage\_form}}", "{{route\_of\_administration}}",

"{{dose}}", "{{frequency}}", "{{therapy\_duration}}", "{{indication}}",

"{{meddra\_version}}", "{{reaction\_duration}}"]

values = [v.strip() for v in value.split(',') if v.strip() not in placeholders]

return values

* + Splits a comma-separated string and removes placeholders.
  + Used to clean input fields before formatting into HTML.
* Function: format\_combined\_values(quantity, unit)

def format\_combined\_values(quantity, unit):

"""Combine quantity and unit into a single string."""

return f"{quantity} {unit}" if quantity and unit else ""

* + Combines quantity and unit into a readable format (e.g., "5 mg").
  + Returns empty string if any field is missing.
* Function: generate\_html\_from\_template(item, formatted\_data, template\_html)

def generate\_html\_from\_template(item, formatted\_data, template\_html):

"""Generate HTML content for one report using the provided template."""

# Replace placeholders in the HTML template with dynamic values

html\_template = template\_html

try:

# Replace placeholders in the HTML template with dynamic values

html\_template = html\_template.replace('{{adverse\_reaction\_report\_number}}', item.get('report\_no', ''))

html\_template = html\_template.replace('{{latest\_aer\_version\_number}}', item.get('version\_no', ''))

html\_template = html\_template.replace('{{initial\_received\_date}}', item.get('datintreceived', ''))

html\_template = html\_template.replace('{{latest\_received\_date}}', item.get('datreceived', ''))

html\_template = html\_template.replace('{{source\_of\_report}}', item.get('source\_eng', ''))

html\_template = html\_template.replace('{{market\_authorization\_holder\_aer\_number}}', item.get('mah\_no', ''))

html\_template = html\_template.replace('{{type\_of\_report}}', item.get('report\_type\_eng', ''))

html\_template = html\_template.replace('{{reporter\_type}}', item.get('reporter\_type\_eng', ''))

html\_template = html\_template.replace('{{serious}}', item.get('seriousness\_eng', ''))

# Replace side-table (death, disability, etc.)

html\_template = html\_template.replace('{{death}}', item.get('death', ''))

html\_template = html\_template.replace('{{disability}}', item.get('disability', ''))

html\_template = html\_template.replace('{{anomaly}}', item.get('congenital\_anomaly', ''))

html\_template = html\_template.replace('{{life\_threatening}}', item.get('life\_threatening', ''))

html\_template = html\_template.replace('{{hospitalization}}', item.get('hospitalization', ''))

html\_template = html\_template.replace('{{other\_conditions}}', item.get('other\_medically\_imp\_cond', ''))

# Patient info

html\_template = html\_template.replace('{{age}}', item.get('age', '') + ' ' + item.get('age\_unit\_eng', ''))

html\_template = html\_template.replace('{{gender}}', item.get('gender\_eng', ''))

html\_template = html\_template.replace('{{height}}', item.get('height', '') + ' ' + item.get('height\_unit\_eng', ''))

html\_template = html\_template.replace('{{weight}}', item.get('weight', '') + ' ' + item.get('weight\_unit\_eng', ''))

html\_template = html\_template.replace('{{report\_outcome}}', item.get('outcome\_eng', ''))

html\_template = html\_template.replace('{{record\_type}}', item.get('record\_type\_eng', ''))

html\_template = html\_template.replace('{{link\_aer\_number}}', item.get('report\_link\_no', ''))

# Format product rows to prevent nesting

product\_rows = ""

max\_length = max(len(formatted\_data[key]) for key in['drug\_name', 'drug\_involvement', 'dosage\_form', 'route', 'dose', 'freq\_time', 'therapy\_duration', 'indication'])

for i in range(max\_length):

drug\_name = formatted\_data['drug\_name'][i] if i < len(formatted\_data['drug\_name']) else ""

drug\_involvement = formatted\_data['drug\_involvement'][i] if i < len(formatted\_data['drug\_involvement']) else ""

dosage\_form = formatted\_data['dosage\_form'][i] if i < len(formatted\_data['dosage\_form']) else ""

route = formatted\_data['route'][i] if i < len(formatted\_data['route']) else ""

dose = formatted\_data['dose'][i] if i < len(formatted\_data['dose']) else ""

freq\_time = formatted\_data['freq\_time'][i] if i < len(formatted\_data['freq\_time']) else ""

therapy\_duration = formatted\_data['therapy\_duration'][i] if i < len(formatted\_data['therapy\_duration']) else ""

indication = formatted\_data['indication'][i] if i < len(formatted\_data['indication']) else ""

# Add a row for the product information

product\_rows += f"<tr><td class='left-align'>{drug\_name}</td><td>{drug\_involvement}</td><td>{dosage\_form}</td><td>{route}</td><td>{dose}</td><td>{freq\_time}</td><td>{therapy\_duration}</td><td>{indication}</td></tr>"

if not product\_rows.strip():

product\_rows = "<tr><td colspan='8'>No product data available</td></tr>"

html\_template = html\_template.replace('{{product\_description}}', product\_rows)

# Format adverse reaction rows

adverse\_reaction\_rows = ""

for i in range(len(formatted\_data['pt\_name'])):

adverse\_reaction\_rows += f"<tr><td class='left-align'>{formatted\_data['pt\_name'][i]}</td><td>{formatted\_data['meddra\_version'][i]}</td><td>{formatted\_data['duration'][i]} {formatted\_data['duration\_unit'][i]}</td></tr>"

if not adverse\_reaction\_rows.strip():

adverse\_reaction\_rows = "<tr><td colspan='3'>No adverse reaction data available</td></tr>"

html\_template = html\_template.replace('{{adverse\_reaction\_terms}}', adverse\_reaction\_rows)

return html\_template

except Exception as e:

print(f"Error in generating HTML: {e}")

return ""

* + Replaces placeholders in the HTML template using values from the JSON data.
  + Fills in individual report details like:
  + Report metadata (dates, reporter type, seriousness, etc.)
  + Patient details (age, gender, height, weight, outcomes)
  + Product details (drug name, route, frequency, dose, etc.)
  + Reaction data (PT name, MedDRA version, duration)
  + Generates a table row for each product or reaction entry.
  + Returns the final HTML for one report.
* Function: format\_data(item)

def format\_data(item):

"""Formats the data and handles comma-separated values."""

fields = {

'drug\_name': split\_comma\_values(item.get('drug\_name', '')),

'drug\_involvement': split\_comma\_values(item.get('drug\_involvement', '')),

'dosage\_form': split\_comma\_values(item.get('dosage\_form\_eng', '')),

'route': split\_comma\_values(item.get('route\_admin', '')),

'unit\_dose': split\_comma\_values(item.get('unit\_dose\_qty', '')),

'dose\_unit': split\_comma\_values(item.get('dose\_unit\_eng', '')),

'freq\_time': split\_comma\_values(item.get('freq\_time\_unit\_eng', '')),

'therapy\_duration': split\_comma\_values(item.get('therapy\_duration', '')),

'therapy\_unit': split\_comma\_values(item.get('therapy\_duration\_unit\_eng', '')),

'indication': split\_comma\_values(item.get('indication\_eng', '')),

'pt\_name': split\_comma\_values(item.get('pt\_name\_eng', '')),

'meddra\_version': split\_comma\_values(item.get('meddra\_version', '')),

'duration': split\_comma\_values(item.get('duration', '')),

'duration\_unit': split\_comma\_values(item.get('duration\_unit\_eng', '')),

}

fields['dose'] = [format\_combined\_values(qty, unit) for qty, unit in zip(fields['unit\_dose'], fields['dose\_unit'])]

fields['therapy\_duration'] = [format\_combined\_values(dur, unit) for dur, unit in

zip(fields['therapy\_duration'], fields['therapy\_unit'])]

# Ensure that drug names are sorted alphabetically

fields['drug\_name'] = sorted(fields['drug\_name'])

fields['pt\_name'] = sorted(fields['pt\_name'])

return fields

* + Takes a report dictionary and prepares all values for HTML insertion.
  + Splits and formats data, combines dose/units, therapy/units.
  + Sorts drug and reaction names alphabetically.
  + Returns a dictionary of cleaned and structured lists.
* Function: generate\_input\_html(json\_data, template\_html)

def generate\_input\_html(json\_data, template\_html):

"""Generate the complete input.html file that contains all reports."""

report\_htmls = []

for index, item in enumerate(json\_data):

formatted\_data = format\_data(item)

report\_html = generate\_html\_from\_template(item, formatted\_data, template\_html)

# Wrap each report in <html></html> tags individually

report\_html = report\_html

report\_htmls.append(report\_html)

# Join all reports and return

return "\n".join(report\_htmls)

* + Loops through each report item in the JSON data.
  + Calls format\_data() and generate\_html\_from\_template() for each report.
  + Aggregates all report HTML sections into a single HTML string.
  + Returns full HTML content.
* Function: upload\_html\_to\_s3(html\_content, bucket\_name, file\_name)

def upload\_html\_to\_s3(html\_content, bucket\_name, file\_name):

"""Upload the generated HTML content to S3 bucket."""

s3\_client = boto3.client('s3')

s3\_client.put\_object(Body=html\_content, Bucket=bucket\_name, Key=file\_name, ContentType='text/html')

* + Uploads the final HTML string to the specified location in S3.
  + Uses content type text/html.
* Function: main()

def main():

try:

# S3 bucket details

input\_bucket = os.getenv("INPUT\_BUCKET") # Bucket containing the report\_output directory

output\_bucket = os.getenv("OUTPUT\_BUCKET") # Bucket to upload the generated HTML

directory = os.getenv("DIRECTORY") # Directory in the input bucket containing the JSON files

timestamp = time.strftime('%d\_%b\_%Y\_%H\_%M\_%S')

output\_html\_file\_key = f'input-html/reported\_adverse\_reaction\_{timestamp}.html' # Path in the output bucket where the file will be uploaded

# Load the latest JSON data from S3

json\_data = load\_json\_from\_s3(input\_bucket, directory)

if json\_data:

# Dynamically load the HTML template from the current script's directory

template\_path = os.path.join(os.path.dirname(\_\_file\_\_), 'template.html')

with open(template\_path, 'r') as file:

template\_html = file.read()

# Generate the input HTML

input\_html = generate\_input\_html(json\_data, template\_html)

# Upload the HTML file to S3

upload\_html\_to\_s3(input\_html, output\_bucket, output\_html\_file\_key)

print(f"HTML content successfully uploaded to {output\_bucket}/{output\_html\_file\_key}")

# Now invoke the CVP2\_EMAIL Lambda after successfully completing the tasks

invoke\_cvp2\_email\_lambda() # Trigger the second Lambda function

else:

print("Failed to load JSON data from S3.")

except Exception as e:

print(f"Error during execution: {e}")

* + Loads configuration from environment variables:
  + INPUT\_BUCKET, OUTPUT\_BUCKET, DIRECTORY
  + Generates a timestamped file path for the HTML file.
  + Loads the latest JSON report from S3.
  + Loads the HTML template file (template.html).
  + Generates the HTML content using the template and JSON data.
  + Uploads the resulting HTML to the specified output S3 path.
  + Triggers the CVP2\_EMAIL Lambda function.
  + Logs success or failure.
* Lambda Handler: lambda\_handler(event, context)

def lambda\_handler(event, context):

"""Lambda handler function."""

try:

result = main()

return result

except Exception as e:

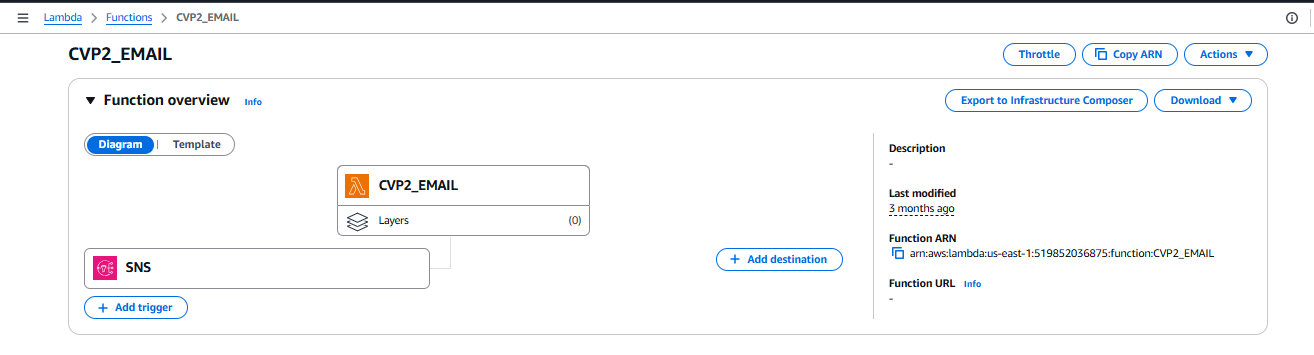
logger.error(f"Error in lambda handler: {e}")

return {'statusCode': 500, 'body': f"Error: {str(e)}"}

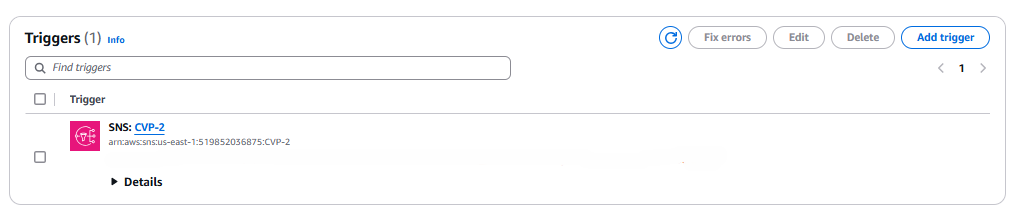
* + Serves as the Lambda entry point.
  + Calls main() and handles error logging.
  + Returns HTTP 500 on failure.
* Function Execution:
* Fetch Latest Report JSON: From cvp-2-output/report\_output/ using load\_json\_from\_s3().
* Parse JSON Reports: Extract and clean data per report.
* Generate HTML: Build formatted report HTML using template.html and placeholders.
* Upload HTML File: Save the final HTML content to cvp-2-bucket/input-html/ with a timestamp.
* Invoke Email Lambda: Automatically trigger CVP2\_EMAIL Lambda to send notifications.

### 6.4.3 CVP2\_EMAIL:

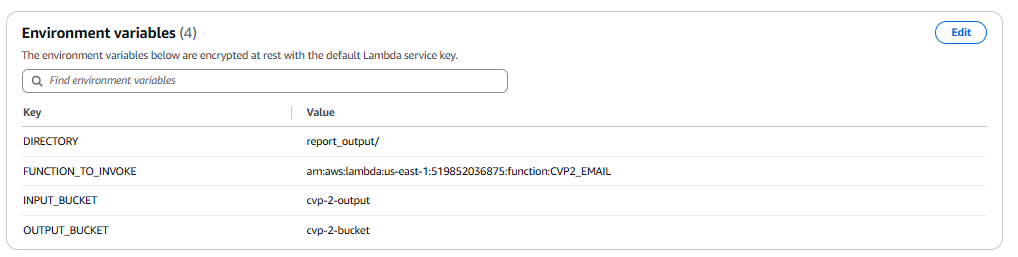
* Create Lambda Function:
* Function Name: CVP2\_EMAIL
* Runtime: Python
* Role: Attach LambdaExecutionRole
* Code: Upload Python script for sending emails.



* Set Triggers to the function.
* Trigger for CVP2\_EMAIL
* Navigate to S3 Service:
* Go to the S3 section in the AWS Management Console.
* Create Notification for Bucket:
* Select the email.
* Go to the Properties tab and scroll to Event notifications.
* Add few emails as subscriptions
* Save the notification settings



* Adding Environment Variables to Lambda Functions
* Navigate to AWS Lambda:
* Go to the AWS Management Console and open Lambda.
* Select Your Lambda Function:
* Go to the Environment Variables Section:
* Scroll down to the Environment variables section under your function's configuration.
* Add Environment Variables:
* Click Edit in the Environment variables section.
* Click Add environment variable and enter the key-value pairs for your environment variables.



* Detailed Explanation of Python Code
* Imports

import os

import json

import boto3

from botocore.exceptions import ClientError

from datetime import datetime

* + os: Accesses environment variables like sender/recipient email and S3 paths.
  + json: Parses the JSON file content.
  + boto3: AWS SDK used for interacting with S3 and SES.
  + ClientError: Catches client-side errors from AWS services.
  + datetime: Generates timestamp for email subject and body.
* AWS Client Initialization:

# Initialize Boto3 clients

s3\_client = boto3.client('s3')

ses\_client = boto3.client('ses')

* + s3\_client = boto3.client('s3'): For downloading report files.
  + ses\_client = boto3.client('ses'): For sending emails via AWS SES.
* Environment Variable Configuration:

# Email settings from environment variables

SENDER\_EMAIL = os.getenv("SENDER\_EMAIL")

RECIPIENT\_EMAIL = os.getenv("RECIPIENT\_EMAIL")

CC\_EMAIL = os.getenv("CC\_EMAIL")

# S3 and processing configuration

BUCKET\_NAME = os.getenv('BUCKET\_NAME')

FOLDER\_PREFIX = os.getenv('FOLDER\_PREFIX') # 'Adverse\_reaction\_reports/report\_details\_output\_'

* + SENDER\_EMAIL, RECIPIENT\_EMAIL, CC\_EMAIL: Set up email addresses for sending and receiving.
  + BUCKET\_NAME: S3 bucket that stores the JSON report.
  + FOLDER\_PREFIX: S3 path where the report files are saved.
* Function: fetch\_s3\_file(bucket\_name, file\_key)

def fetch\_s3\_file(bucket\_name, file\_key):

"""Fetches JSON file from S3 bucket and parses it."""

try:

response = s3\_client.get\_object(Bucket=bucket\_name, Key=file\_key)

body\_content = response['Body'].read().decode('utf-8')

if not body\_content:

print(f"Warning: Empty content retrieved from {file\_key}.")

return None

return json.loads(body\_content)

except ClientError as e:

print(f"Error fetching the file: {e}")

return None

except json.JSONDecodeError as e:

print(f"JSONDecodeError: {e}. Content: {body\_content}")

return None

* + Downloads a specified JSON file from S3.
  + Parses and returns the data using json.loads().
  + Handles and logs errors like missing files or decode failures.
* Function: generate\_email\_body(data, sent\_date)

def generate\_email\_body(data, sent\_date):

"""Generates HTML email body with a single table including all entries."""

html\_body = f"""

<html>

<body>

<h2 style="color: black; text-align: center; font-size: 24px;">Adverse Reaction Report - Alert</h2>

<p style="color: black;">This email contains the results from the extraction of Adverse Reaction Report.</p>

<p style="color: black;">The alert results cover the screening period up to <strong>{sent\_date}</strong>.</p>

<table border="1" cellpadding="5" cellspacing="0">

<tr>

<th style="color: black;">Sl.No</th>

<th style="color: black;">Adverse Reaction Report Number</th>

<th style="color: black;">Market Authorization Holder AER Number</th>

<th style="color: black;">Initial Received Date</th>

<th style="color: black;">Source of Report</th>

<th style="color: black;">Age</th>

<th style="color: black;">Gender</th>

<th style="color: black;">Suspected Product Brand Name</th>

<th style="color: black; width: 300px; padding-left: 10px; padding-right: 10px;">Adverse Reaction Terms</th> <!-- Increased width and padding -->

</tr>

"""

# Add each report as a row

for idx, report in enumerate(data, start=1):

# formatted\_reactions = ', '.join(part.strip().replace(' ', '') for part in report['pt\_name\_eng'].split(','))

html\_body += f"""

<tr>

<td style="color: black;">{idx}</td>

<td style="color: black;">{report['report\_no']}</td>

<td style="color: black;">{report.get('mah\_no', 'N/A')}</td>

<td style="color: black;">{report['datintreceived']}</td>

<td style="color: black;">{report['source\_eng']}</td>

<td style="color: black;">{report['age']} {report['age\_unit\_eng']}</td>

<td style="color: black;">{report['gender\_eng']}</td>

<td style="color: black;">{report['drug\_name']}</td>

<td style="color: black;">{report['pt\_name\_eng']}</td>

</tr>

"""

html\_body += """

</table>

</body>

</html>

"""

return html\_body

* + Constructs an HTML table format email body.
  + Includes:
  + Report number, MAH number, received date, source of report
  + Patient details (age, gender)
  + Drug name and adverse reaction terms
  + Adds styling to table and headers for a professional look.
  + Accepts sent\_date to mark the alert period in the message.
* Function: send\_email()

def send\_email(subject, body\_html):

"""Sends an email using SES."""

try:

response = ses\_client.send\_email(

Source=SENDER\_EMAIL,

Destination={

'ToAddresses': [RECIPIENT\_EMAIL],

'CcAddresses': [CC\_EMAIL] # Add CC email handling here

},

Message={

'Subject': {'Data': subject},

'Body': {'Html': {'Data': body\_html}}

}

)

print(f"Email sent! Message ID: {response['MessageId']}")

except ClientError as e:

print(f"Error sending email: {e}")

* Function: get\_latest\_file(bucket\_name, folder\_prefix)

def get\_latest\_file(bucket\_name, folder\_prefix):

"""Fetches the most recent file based on LastModified date from the specified folder in S3."""

try:

response = s3\_client.list\_objects\_v2(Bucket=bucket\_name, Prefix=folder\_prefix)

files = response.get('Contents', [])

# Sort files by last modified date and return the key of the most recent file

files.sort(key=lambda x: x['LastModified'], reverse=True)

if files:

return files[0]['Key']

return None

except ClientError as e:

print(f"Error fetching the file list: {e}")

return None

* + Lists all files in a given S3 folder prefix.
  + Sorts files by last modified time (descending).
  + Returns the key of the most recent file or None if empty.
* Lambda Handler: lambda\_handler(event, context)

def lambda\_handler(event, context):

"""Main Lambda handler."""

latest\_file = get\_latest\_file(BUCKET\_NAME, FOLDER\_PREFIX)

if not latest\_file:

print("No files found in the specified folder.")

return {'statusCode': 200, 'body': 'No files found in the specified folder.'}

data = fetch\_s3\_file(BUCKET\_NAME, latest\_file)

if not data:

print(f"Error retrieving or decoding content from {latest\_file}.")

return {'statusCode': 200, 'body': f"Error retrieving or decoding content from {latest\_file}."}

sent\_date = datetime.now().strftime('%Y-%m-%d %H:%M:%S')

email\_body = generate\_email\_body(data, sent\_date)

subject = f"Adverse Reaction Alert - {sent\_date}"

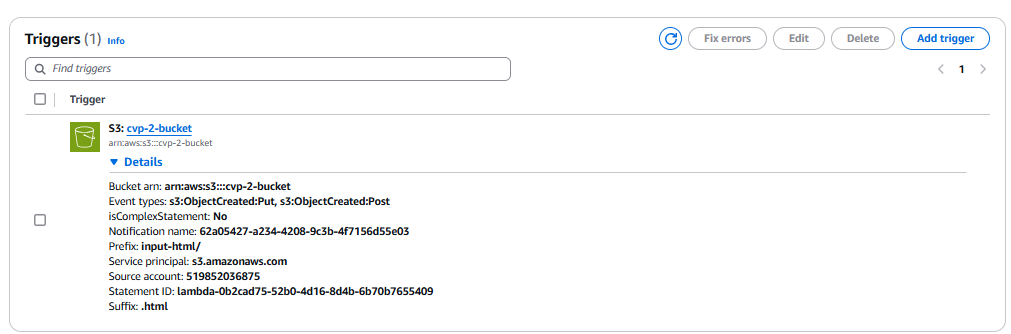
send\_email(subject, email\_body)

return {'statusCode': 200, 'body': 'Email sent successfully.'}

* + Orchestrates the execution:
  + Retrieves the latest report file from S3.
  + Parses the report content.
  + Generates the email body.
  + Sends the formatted report via email.
  + Returns HTTP 200 response whether or not data was found.
* Function Execution:
  + Find Latest Report File: From S3 folder Adverse\_reaction\_reports/report\_details\_output\_...
  + Load File Content: Parse JSON and prepare structured data.
  + Format Email: Build a styled HTML table with report highlights.
  + Send via SES: Use configured email addresses to send alert.

### 6.4.4 CVP2\_HTML\_to\_PDF

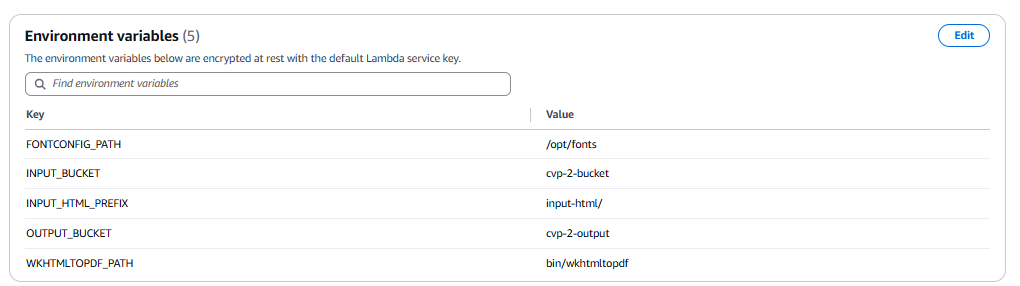
* Create Lambda Function:
* Function Name: CVP2\_HTML\_to\_PDF
* Runtime: Python
* Role: Attach LambdaExecutionRole
* Code: Upload Python script for sending emails.
* Set Triggers to the function.
* Trigger for CVP2\_HTML\_to\_PDF:
* Navigate to S3 Service:
* Go to the S3 section in the AWS Management Console.
* Create Notification for Bucket:
* Select the cvp-2-bucket bucket.
* Go to the Properties tab and scroll to Event notifications.
* Create Event Notification:
* Name: Trigger\_RSS\_LAM\_02.
* Event Type: Choose All object create events in folder input-html/.
* Save the notification settings.



* Adding Custom Layers to Lambda Functions
* Package the Custom Layer:
* Zip the Layer:
* Upload the Custom Layer to AWS:
* Create Layer:
* Add the Custom Layer to the Lambda Function:
* Add a Layer and Verify the Layer in the Function:
* Attached PDFKIT\_and\_BOTO3 , PyPDF2 and wkhtmltopdf dependency as a custom layer.



* Adding Environment Variables to Lambda Functions
* Navigate to AWS Lambda:
* Go to the AWS Management Console and open Lambda.
* Select Your Lambda Function:
* Go to the Environment Variables Section:
* Scroll down to the Environment variables section under your function's configuration.
* Add Environment Variables:
* Click Edit in the Environment variables section.
* Click Add environment variable and enter the key-value pairs for your environment variables.



* Detailed Explanation of Python Code
* Imports:

import json

import boto3

import pdfkit

import PyPDF2

from concurrent.futures import ThreadPoolExecutor

from io import BytesIO

import time

from datetime import datetime

import logging

import os

* + json: Formats and encodes response data.
  + boto3: AWS SDK used to interact with S3.
  + pdfkit: Converts HTML to PDF using wkhtmltopdf.
  + PyPDF2: Merges multiple PDFs into a single file.
  + ThreadPoolExecutor: Runs concurrent PDF generation tasks.
  + BytesIO: In-memory byte stream used to avoid writing to disk.
  + time, datetime: Used to generate timestamped file names.
  + logging: Logs messages (not configured explicitly here).
  + os: Reads environment variables like S3 paths and wkhtmltopdf path.
* AWS S3 Client Initialization:

# Initialize the S3 client

s3\_client = boto3.client('s3')

* + s3\_client = boto3.client('s3'): Provides access to S3 for reading and writing files.
* Function: get\_latest\_file\_from\_s3(bucket\_name, prefix)

def get\_latest\_file\_from\_s3(bucket\_name, prefix):

"""

Retrieve the latest file from a specific directory in the S3 bucket.

:param bucket\_name: The name of the S3 bucket

:param prefix: The directory or folder within the bucket

:return: The key of the most recently modified file

"""

try:

# List objects in the S3 bucket with the specified prefix (directory)

response = s3\_client.list\_objects\_v2(Bucket=bucket\_name, Prefix=prefix)

# Check if there are any files in the directory

if 'Contents' not in response or len(response['Contents']) == 0:

raise Exception("No files found in the specified directory.")

# Sort the files by 'LastModified' in descending order (most recent first)

files = sorted(response['Contents'], key=lambda x: x['LastModified'], reverse=True)

# Get the most recently modified file's key

latest\_file\_key = files[0]['Key']

return latest\_file\_key

except Exception as e:

print(f"Error retrieving the latest file: {str(e)}")

return None

* + Lists files in the specified S3 prefix (folder path).
  + Sorts files by LastModified in descending order.
  + Returns the key of the most recently uploaded HTML file.
  + Handles and logs errors if no files are found or listing fails.
* Lambda Handler: lambda\_handler(event, context)

def lambda\_handler(event, context):

# Source S3 bucket and key for the input HTML

input\_bucket\_name = os.getenv("INPUT\_BUCKET") # Replace with your input bucket name

input\_html\_prefix = os.getenv("INPUT\_HTML\_PREFIX") # Directory or folder in the S3 bucket

timestamp = time.strftime('%d\_%b\_%Y\_%H\_%M\_%S')

# Destination S3 bucket and key for the generated PDF

output\_bucket\_name = os.getenv("OUTPUT\_BUCKET") # Replace with your output bucket name

output\_pdf\_key = f'output-pdf/reported\_adverse\_reaction\_{timestamp}.pdf' # Path in the bucket where the PDF will be stored

# Path to the wkhtmltopdf binary

wkhtmltopdf\_path = os.getenv("WKHTMLTOPDF\_PATH") # Adjust this path as needed (use Lambda Layer for wkhtmltopdf)

try:

# Get the key of the latest HTML file in the specified directory

input\_html\_key = get\_latest\_file\_from\_s3(input\_bucket\_name, input\_html\_prefix)

# If no file is found, return an error

if not input\_html\_key:

return {

'statusCode': 500,

'body': json.dumps("No files found in the specified S3 folder.")

}

# Fetch the HTML file from the S3 bucket

response = s3\_client.get\_object(Bucket=input\_bucket\_name, Key=input\_html\_key)

html\_content = response['Body'].read().decode('utf-8') # Decode the content to string

# Split the HTML content wherever a new <html> tag appears

html\_parts = html\_content.split('<html>')

# Ensure each part is reconstructed properly

formatted\_html\_parts = [f"<html>{part.strip()}" for part in html\_parts if part.strip()]

# Add page breaks between parts

html\_with\_page\_breaks = "<html>".join(

[f'{part}<div style="page-break-after: always;"></div>' for part in formatted\_html\_parts]

)

# Specify the wkhtmltopdf executable in pdfkit configuration

config = pdfkit.configuration(wkhtmltopdf=wkhtmltopdf\_path)

options = {

'orientation': 'Landscape',

'page-size': 'A4'

}

# Function to generate PDF from a string (HTML)

def generate\_pdf\_from\_html(html\_string):

return pdfkit.from\_string(html\_string, False, configuration=config, options=options)

# Use ThreadPoolExecutor to handle HTML parts concurrently

with ThreadPoolExecutor(max\_workers=5) as executor:

pdf\_parts = list(executor.map(generate\_pdf\_from\_html, formatted\_html\_parts))

# Initialize PyPDF2 PdfMerger

pdf\_merger = PyPDF2.PdfMerger()

# Merge each PDF part into the final PDF

for pdf\_part in pdf\_parts:

pdf\_reader = PyPDF2.PdfReader(BytesIO(pdf\_part))

pdf\_merger.append(pdf\_reader)

# Write the combined PDF to a BytesIO object

combined\_pdf = BytesIO()

pdf\_merger.write(combined\_pdf)

combined\_pdf.seek(0) # Reset the stream position

# Upload the merged PDF to the S3 bucket

s3\_client.put\_object(

Bucket=output\_bucket\_name,

Key=output\_pdf\_key,

Body=combined\_pdf,

ContentType='application/pdf'

)

return {

'statusCode': 200,

'body': json.dumps(f"PDF generated and uploaded to S3 at {output\_pdf\_key}")

}

except Exception as e:

return {

'statusCode': 500,

'body': json.dumps(f"Error generating PDF: {str(e)}")

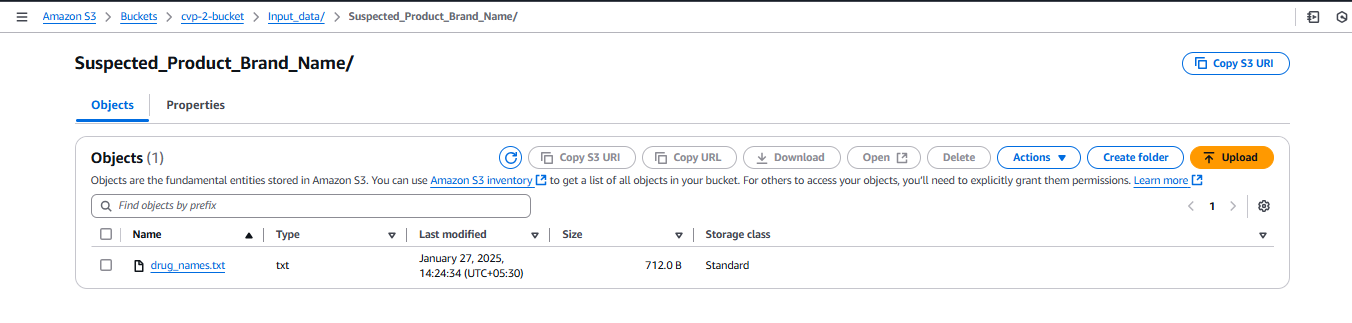
}

* + Loads environment variables:
  + INPUT\_BUCKET, INPUT\_HTML\_PREFIX: Where the HTML is stored.
  + OUTPUT\_BUCKET: Where to upload the PDF.
  + WKHTMLTOPDF\_PATH: Location of wkhtmltopdf binary, configured via Lambda Layer.
  + Generates a timestamped file name for the output PDF.
  + Calls get\_latest\_file\_from\_s3() to find the newest HTML report.
  + Retrieves the HTML content and splits it at every new <html> tag.
  + Reconstructs each split into a proper HTML document.
  + Adds page breaks between HTML parts to separate them in PDF.
  + Configures pdfkit with options:
  + Orientation: Landscape
  + Page size: A4
  + Uses ThreadPoolExecutor to convert multiple HTML segments concurrently to PDFs.
  + Merges the resulting PDF files using PyPDF2.PdfMerger().
  + Uploads the final PDF to the output bucket at output-pdf/ path.
  + Returns a status message or error depending on success.
* PDF Processing Flow:
  + Split multi-record HTML into chunks.
  + Convert each HTML segment to a PDF (in parallel).
  + Merge all individual PDFs into a single document.
  + Save and upload the combined file to S3.
* Function Execution:
  + Retrieve HTML File: Identify the latest HTML file in S3 at input-html/.
  + Split & Clean HTML Content: Use <html> tags to isolate multiple reports.
  + Generate PDFs in Parallel: Render each HTML part into PDF using pdfkit.
  + Merge PDFs: Combine all PDFs into one using PyPDF2.
  + Upload Output: Store the final PDF file in cvp-2-output/output-pdf/ bucket with a timestamped name.

# PART 7: Detailed Workflow

## 7.1 Input Phase:

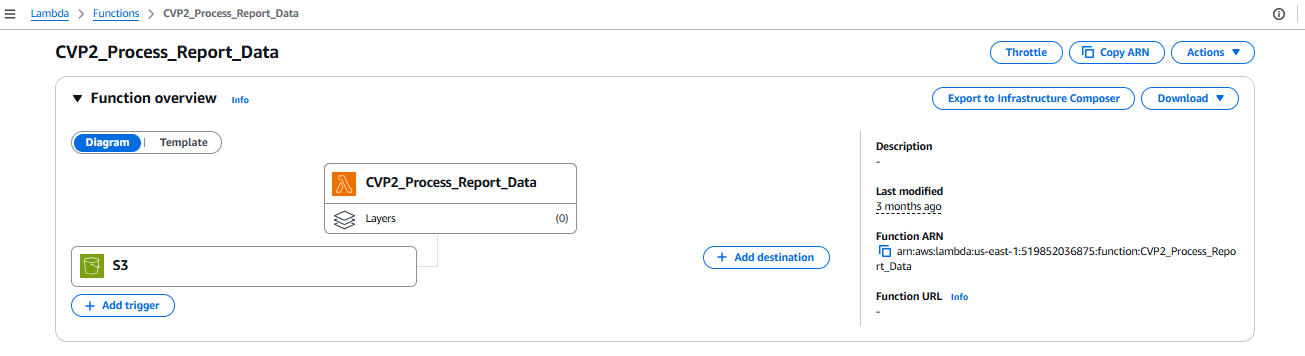
The process begins when the drug names list is updated and uploaded to the S3 bucket cvp-2-bucket, under the path Input\_data/Suspected\_Product\_Brand\_Name/drug\_names.txt.



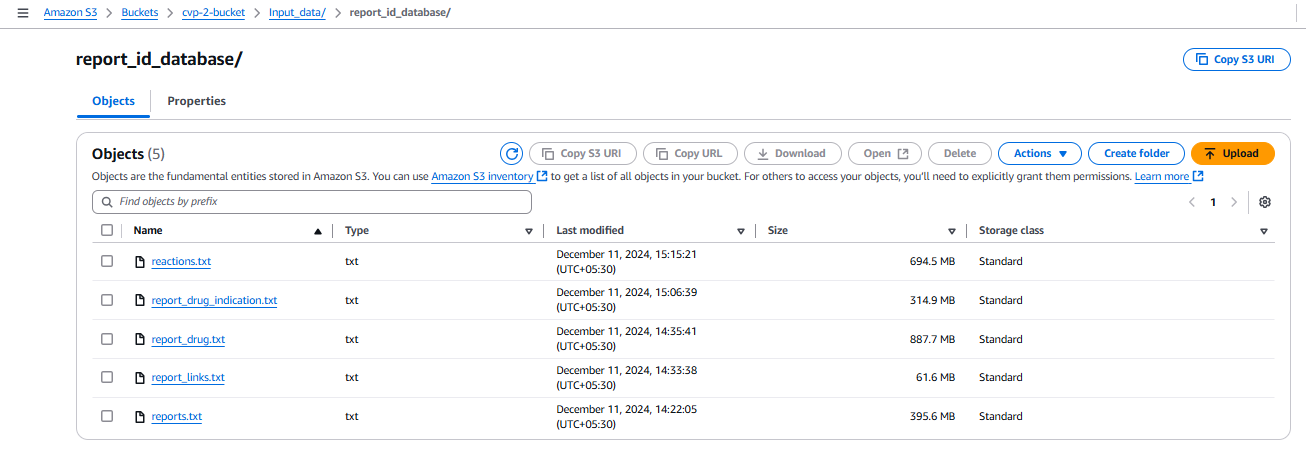
## 7.2 Triggering Lambda Function-01 (CVP2\_Process\_Report\_Data):

An S3 event or EventBridge rule triggers Lambda Function-01 after the drug\_names.txt is updated or on a scheduled basis.

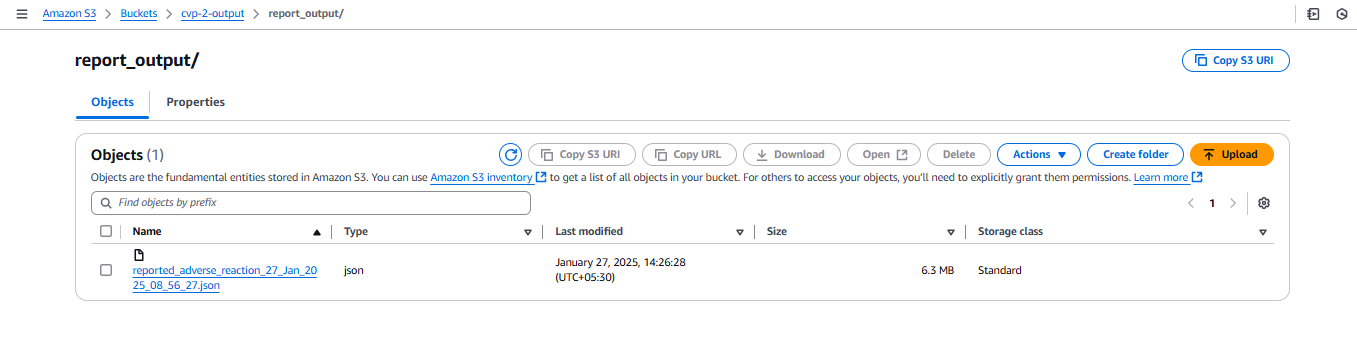
## 7.3 Processing with Lambda Function-01:



* Lambda-01 fetches the following input files from S3:
  + Drug name list: Input\_data/Suspected\_Product\_Brand\_Name/drug\_names.txt
  + Report files: Input\_data/report\_id\_database/ (including reports.txt, reactions.txt, report\_drug.txt, report\_drug\_indication.txt, and report\_links.txt)



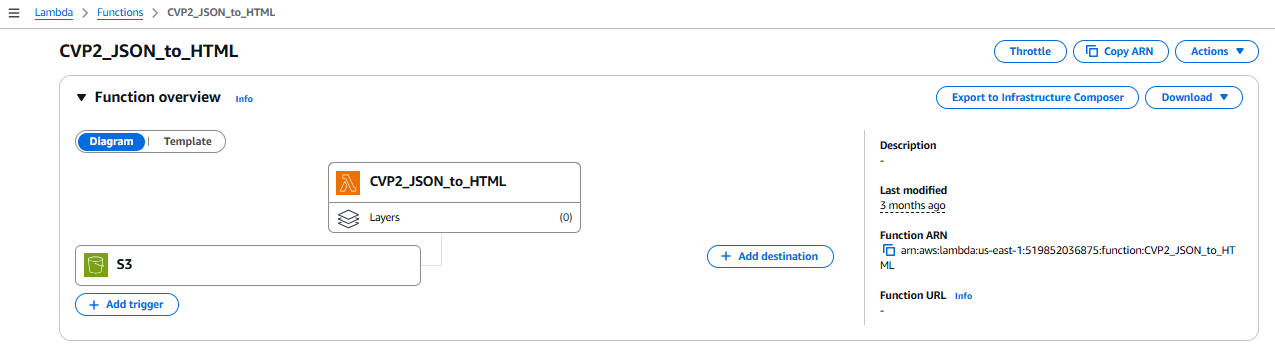
* Drug names are normalized and compared with the report drug file to extract matching REPORT\_IDs.
* Filters out any reports with the SOURCE\_ENG marked as 'mah'.
* Aggregates report data across files (drug info, reactions, patient details).
* Avoids duplicate processing by comparing against previously generated reports.
* Generates a final filtered output as JSON.
* Uploads the JSON to: cvp-2-output/report\_output/reported\_adverse\_reaction\_<timestamp>.json



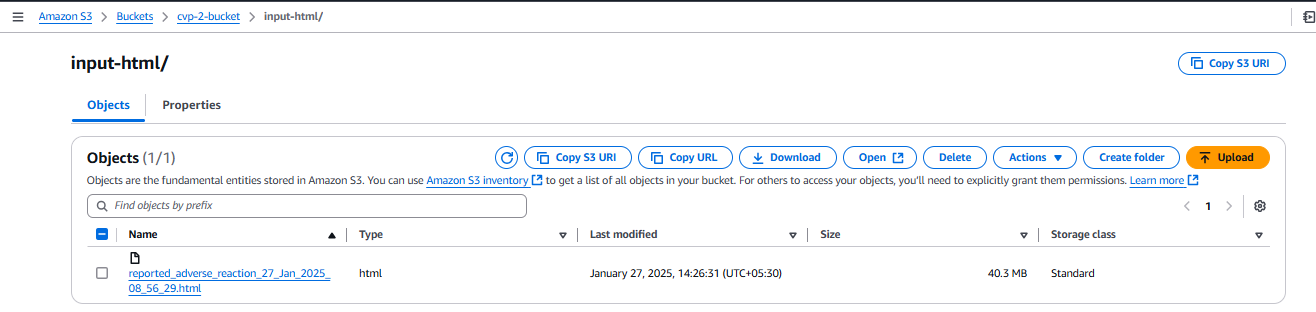
## 7.4 Triggering Lambda Function-02 (CVP2\_JSON\_to\_HTML):

Lambda Function-02 is triggered after the output JSON is uploaded to cvp-2-output/report\_output/.

## 7.5 Converting JSON to HTML with Lambda Function-02:



* Lambda-02 fetches the latest JSON file from cvp-2-output/report\_output/.
* Loads the template.html from the Lambda deployment package.
* Iterates through each report in the JSON:
  + Formats drug data, reactions, patient info.
  + Injects data into placeholders within the HTML template.
* Combines all reports into a single HTML document.
* Uploads the generated HTML to cvp-2-bucket/input-html/reported\_adverse\_reaction\_<timestamp>.html

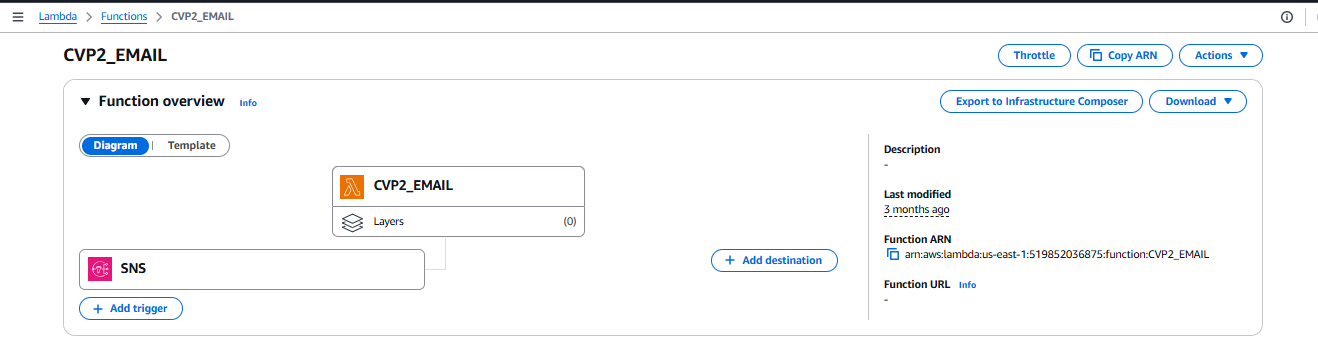


* Once the HTML file is uploaded, Lambda Function-03 is invoked.

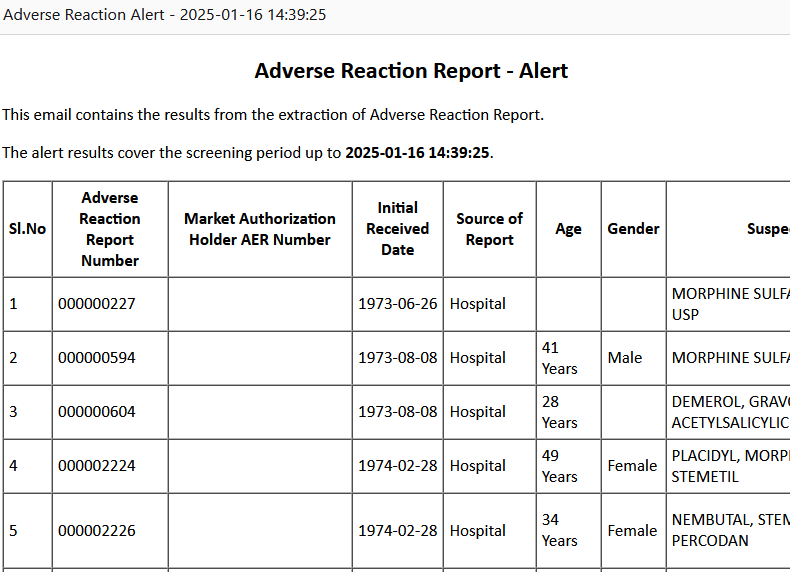
## 7.6 Triggering Lambda Function-03 (CVP2\_EMAIL):

Lambda-03 is triggered immediately after Lambda-02 completes and uploads the HTML file, via asynchronous invocation inside the Lambda code.

## 7.7 Email Notification with Lambda Function-03:



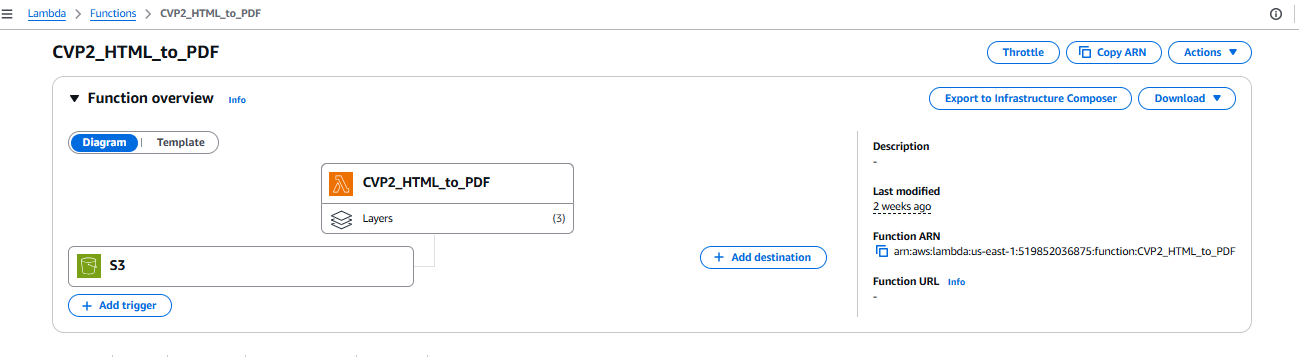
* Lambda-03 determines the latest report data from cvp-2-output/report\_output/.
* Formats the report content into a structured HTML email table.
* Uses Amazon SES to send the email.
  + Sender, recipient, and CC addresses are configured via environment variables.
* The email includes all relevant details:
  + Report number, MAH number, reaction terms, product names, etc.
* Confirms success via response from SES.



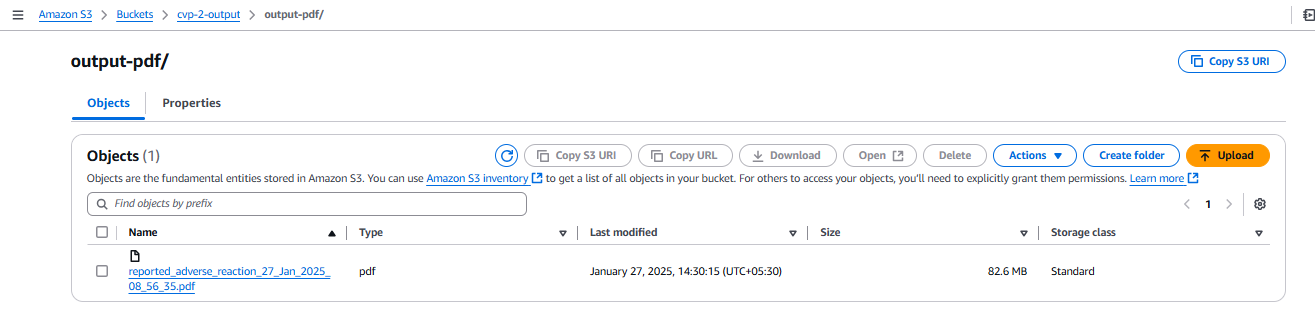
## 7.8 Triggering Lambda Function-04 (CVP2\_HTML\_to\_PDF):

Lambda Function-04 is triggered automatically (can be scheduled or invoked manually).

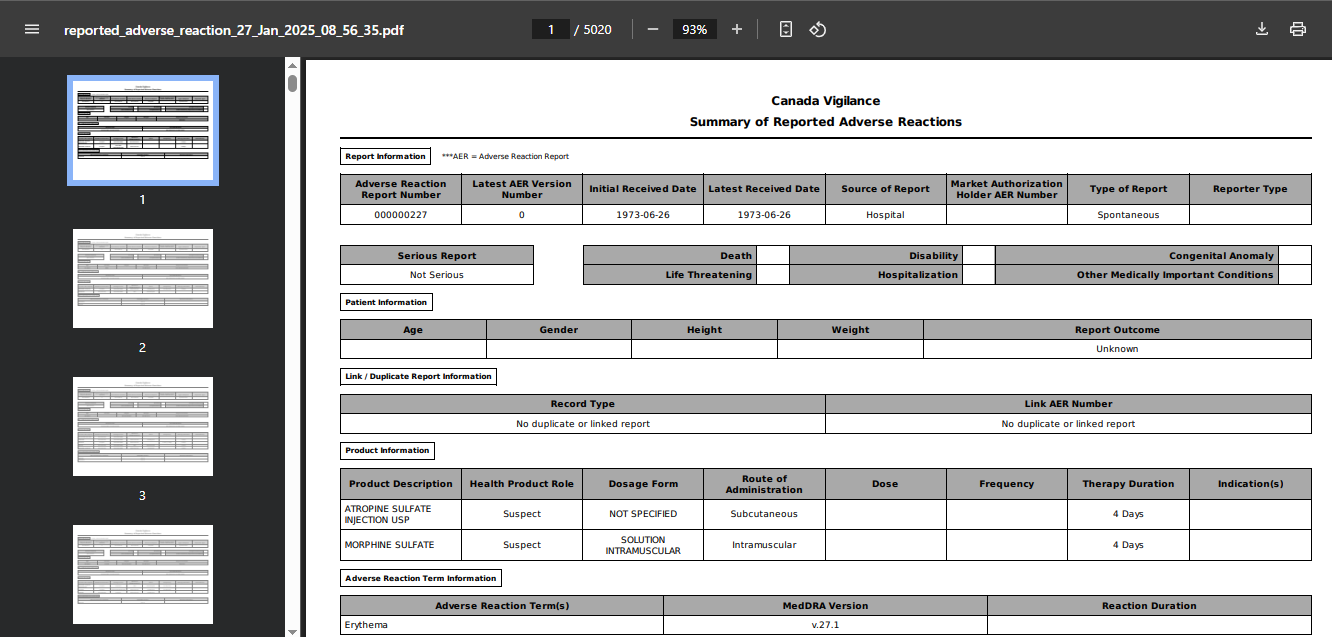
## 7.9 HTML to PDF Conversion with Lambda Function-04:



* Lambda-04 fetches the latest HTML report from cvp-2-bucket/input-html/.
* Splits HTML content into multiple <html> blocks for each report.
* Converts each block into individual PDFs using pdfkit and wkhtmltopdf binary (provided via Lambda Layer).
* Merges all PDFs using PyPDF2.
* Uploads the final PDF to cvp-2-output/output-pdf/reported\_adverse\_reaction\_<timestamp>.pdf



* Returns status of generation and upload.



# PART 8: CONCLUSION

This documentation presents a comprehensive and automated solution for processing adverse reaction report data using AWS serverless technologies. The system follows a modular pipeline approach, consisting of the following key components:

Input & Data Handling:

* Drug names are updated and uploaded to an S3 bucket, serving as the starting point for data extraction.
* Structured report files are accessed from S3 and processed using efficient multi-threaded Lambda functions.

## AWS Lambda Functions:

* The pipeline is divided into four distinct Lambda functions, each with a focused responsibility:
  + Data extraction and filtering (Lambda-01)
  + HTML report generation (Lambda-02)
  + Email notification with formatted content (Lambda-03)
  + PDF report creation from HTML (Lambda-04)
* Functions are triggered via S3 events or asynchronously within the pipeline.

## S3 Buckets & Output Management:

* Separate buckets and structured folders manage inputs, intermediate files, and final outputs.
* Timestamped filenames ensure traceability and prevent overwriting of data.

## AWS SES Integration:

* The system leverages AWS SES to deliver email alerts to configured recipients.
* Report summaries are formatted as rich HTML tables to ensure readability.

## IAM Roles and Permissions:

* Secure and controlled access to AWS resources using IAM roles and policies.
* Specific permissions for Lambda, S3, and SES to ensure smooth operation of the pipeline.

## EventBridge Integration:

* Scheduling and automation of the Lambda functions based on specific triggers or intervals.
* Ensures the timely execution of data extraction, processing, and notification tasks.

## Error Handling and Monitoring:

* Implementation of error handling mechanisms in Lambda functions to ensure robust processing.
* Use of CloudWatch Logs and Alarms for monitoring and alerting on errors or performance issues
* By utilizing AWS-native services such as Lambda, S3, and SES, the pipeline offers a scalable, reliable, and maintainable architecture for public health reporting. This documentation serves as a complete reference for understanding, deploying, and extending the solution for future reporting and automation needs
* This documentation provides a step-by-step guide to the architecture, configuration, and execution of the pipeline, offering valuable insights and best practices for similar projects in the future.

Thank You

Submitted by

Ashwiny Menon

Cloud Engineer