TIDE AI Deployment Guide for Beta Version

Tuesday, August 6, 2024
8:37 AM

By: K. Allen, PeopleTec Inc.

TABLE OF CONTENTS

INTRODUCTION

The TIDE AI Deployment Guide is designed to assist the MLOps deployment team in deploying the production system onto the customer's environment efficiently and effectively. This guide provides a comprehensive overview of the TIDE AI MLOps process, detailing the steps involved in data ingestion, processing, storage, and the lifecycle of machine learning models within the AWS ecosystem.

TIDE (TETRAs Intelligence Digital Ecosystem) is a cloud-based digital ecosystem hosted in the TETRA AWS GovCloud environment. It is accessible on any DoD-connected computer and leverages AI tools to deliver an application for tracking and predicting threat intelligence. This intelligence is incorporated into operational tests to inform future Test and Evaluation (T&E) investments and address T&E threat shortfalls.

The primary objective of this deployment guide is to provide clear, step-by-step instructions for setting up and deploying the TIDE AI system. The guide covers various aspects of the deployment process, including initial setup, permissions, AWS service configurations, and detailed instructions for deploying Lambda functions and other AWS resources. Additionally, it provides guidelines for testing the setup, troubleshooting common issues, and ensuring optimal performance and security.

By following this guide, the deployment team will be able to set up and maintain a robust, efficient, and secure TIDE AI system, ensuring that threat intelligence is effectively managed and utilized to support operational requirements.

TIDE AI MLOPS PROCESS OVERVIEW

DATA INGESTION

Our system integrates intelligence reports from JWICS or other sources like SIPR, handling various data types including textual documents, metadata, image, and quantitative data.

DATA PROCESSING & STORAGE

We employ AWS Lambda and SageMaker to process diverse data formats efficiently. Our system stores extracted text, images, and documents in a structured manner within the AWS S3 data lake. We ensure data segregation—raw PDFs and processed data reside in separate S3 buckets to comply with strict data governance principles. Additionally, automatic data tagging facilitates faster query and retrieval.

ML MODEL LIFECYCLE

We engage in continuous cycles of development, training, testing, and tuning of our ML models within the AWS ecosystem. Utilizing regularly updated datasets from S3, we maintain high model accuracy and relevance. Our current AI capabilities include:

- Text, image data, and metadata extraction from PDFs.
- Keyword analysis and document content summarization.

OVERVIEW OF AI ARCHITECTURE IN AWS

AMAZON S3 (SIMPLE STORAGE SERVICE)

Amazon S3 is a scalable object storage service used to store and retrieve any amount of data at any time. It provides durable and highly available storage infrastructure. S3 is commonly used for data backup, archiving, big data analytics, and content storage and distribution.

Key Features:

- Storage Classes: Different classes for different use cases including Standard, Intelligent-Tiering, Standard-IA, One Zone-IA, Glacier, and Glacier Deep Archive.
- Security: Supports data encryption at rest and in transit. Access control is managed using AWS Identity and Access Management (IAM) policies, bucket policies, and Access Control Lists (ACLs).
- Lifecycle Policies: Automate the transition of objects to different storage classes or delete them after a specified period.
- Versioning: Keep multiple versions of an object in the same bucket to protect against accidental deletions or overwrites.

AMAZON ECR (ELASTIC CONTAINER REGISTRY)

Amazon ECR is a fully managed Docker container registry that makes it easy to store, manage, and deploy Docker container images. It is integrated with Amazon Elastic Container Service (ECS) and AWS Fargate, simplifying the development to production workflow.

Key Features:

- Security: Integration with IAM to control access to repositories and images. Supports image encryption.
- Scalability: Automatically scales to meet the needs of your application.
- Integration: Seamlessly integrates with Amazon ECS, Kubernetes, AWS CodePipeline, and third-party CI/CD tools.

AWS LAMBDA FUNCTIONS

AWS Lambda is a serverless compute service that lets you run code without provisioning or managing servers. You pay only for the compute time you consume.

Key Features:

- Event-Driven: Can be triggered by events from other AWS services such as S3, DynamoDB, Kinesis, SNS, CloudWatch, and API Gateway.
- Scalability: Automatically scales by running code in response to each trigger.
- Resource Configuration: Allows configuration of memory, timeout, and concurrency settings for each function.
- Integration: Integrates with many AWS services and supports custom runtime using Lambda Layers.

AMAZON CLOUDWATCH LOGS AND DASHBOARDS

Amazon CloudWatch is a monitoring and observability service designed for DevOps engineers, developers, site reliability engineers (SREs), and IT managers. CloudWatch Logs enable you to monitor, store, and access log files from Amazon EC2 instances, AWS CloudTrail, and other sources.

Key Features:

- Logs: Collect and monitor logs from AWS resources and custom applications.
- Metrics: Collect and monitor metrics from AWS resources and custom applications.
- Alarms: Set alarms to automatically react to changes in metrics.
- Dashboards: Create custom dashboards to visualize and correlate metrics, logs, and alarms in a single view.

AWS STEP FUNCTIONS

AWS Step Functions is a serverless orchestration service that lets you coordinate multiple AWS services into serverless workflows so you can build and update apps quickly.

Key Features:

- Workflow Automation: Define workflows using state machines to coordinate AWS services such as Lambda, ECS, SageMaker, and more.
- Visual Editor: Create, visualize, and monitor workflows with a graphical editor.
- Integration: Seamlessly integrates with other AWS services and supports API integrations.
- Error Handling: Built-in error handling, retry logic, and state management.

AMAZON SNS (SIMPLE NOTIFICATION SERVICE)

Amazon SNS is a fully managed messaging service for both application-to-application (A2A) and application-to-person (A2P) communication. It allows you to decouple microservices, distributed systems, and serverless applications.

Key Features:

- Messaging: Supports pub/sub messaging to send messages to multiple subscribers.
- Delivery Methods: Supports multiple delivery methods including SMS, email, HTTP/S endpoints, and AWS Lambda functions.
- Filtering: Message filtering and message batching for efficient and cost-effective message delivery.
- Security: Supports encryption of messages in transit and at rest, along with access control through IAM policies.

INITIAL SETUP

The developer needs access to the following resources:

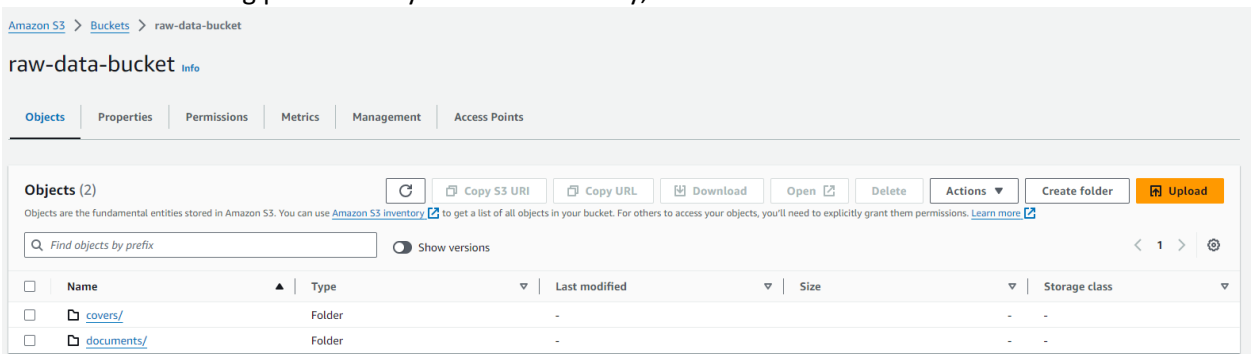
- AWS VPN
- AWS Account
- PowerShell on Local machine with Admin Privileges
- Docker Desktop on Local machine with Admin Privileges
- AWS S3 Access
- AWS ECR Access
- AWS Step Functions and Lambda Access
- Access to CloudWatch for Error logs

PERMISSIONS

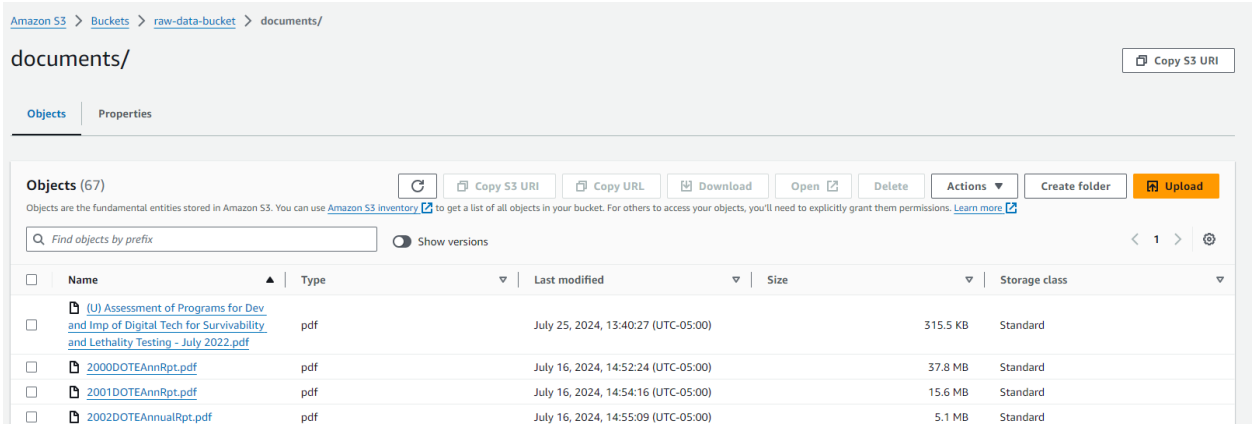
Ensuring the deployment team has the right level of permissions is critical. Without the appropriate permissions, developers will not be able to execute functions or connect to other pieces of the AWS architecture. The cloud services team will be able to setup the necessary permissions for successful deployment.

AMAZON S3 SETUP

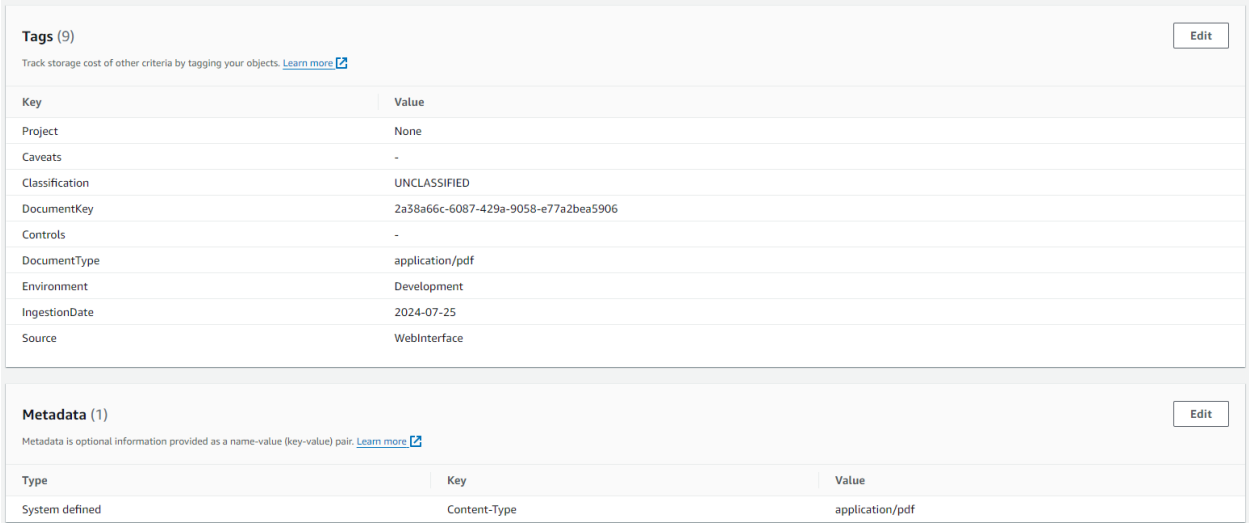
The TIDE AI Team uses two buckets in S3. The first is 'raw-data-bucket', which is the initial entry point for all raw data being processed by AI tools. Currently, there are two folders in the 'raw-data-bucket':



- **'raw-data-bucket/covers'**: stores the covers for the PDFs that are ingested from the UI.
- **'raw-data-bucket/documents'**: stores all the PDFs the users download via the UI and serves as a staging ground for further processing. For TIDE AI functions, we focus primarily on the documents subfolder.



Clicking on any document information specific to that document can be viewed including tags and metadata.



Tags (9) — Edit

Track storage cost of other criteria by tagging your objects. Learn more

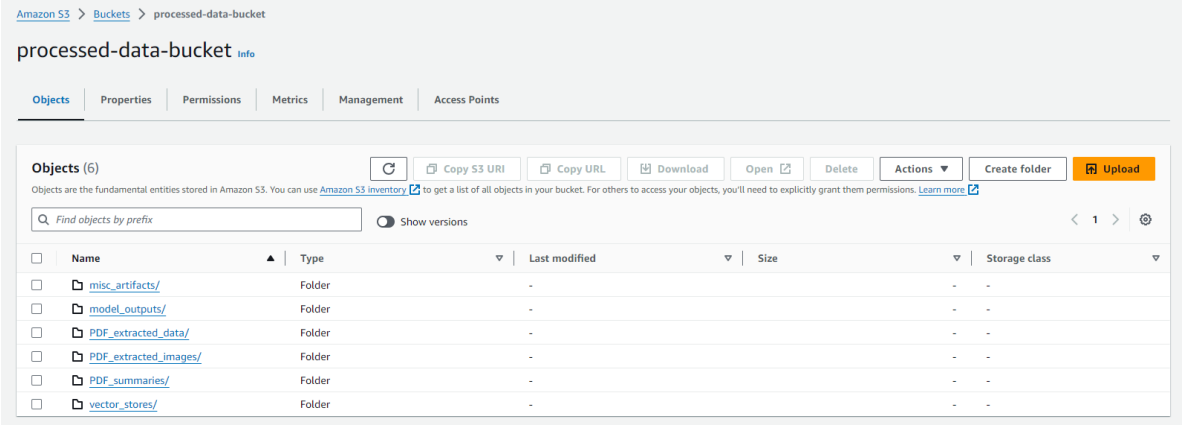
Key	Value
Project	None
Caveats	-
Classification	UNCLASSIFIED
DocumentKey	2a38a66c-6087-429a-9058-e77a2bea5906
Controls	-
DocumentType	application/pdf
Environment	Development
IngestionDate	2024-07-25
Source	WebInterface

Metadata (1) — Edit

Metadata is optional information provided as a name-value (key-value) pair. Learn more

Type	Key	Value
System defined	Content-Type	application/pdf

Both tags and metadata are automatically assigned to each document based on user interface with the TIDE UI. When each document is processed and AI summaries and table/image extraction occurs, the original tags are carried over to each derivative output. The source tag is updated during processing with the name of the function that processed it when derivative outputs are generated and saved to the 'processed-data-bucket' subfolders as shown below:



Amazon S3 > Buckets > processed-data-bucket

processed-data-bucket Info

Objects | Properties | Permissions | Metrics | Management | Access Points

Objects (6) — Copy S3 URI | Copy URL | Download | Open | Delete | Actions | Create folder | Upload

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

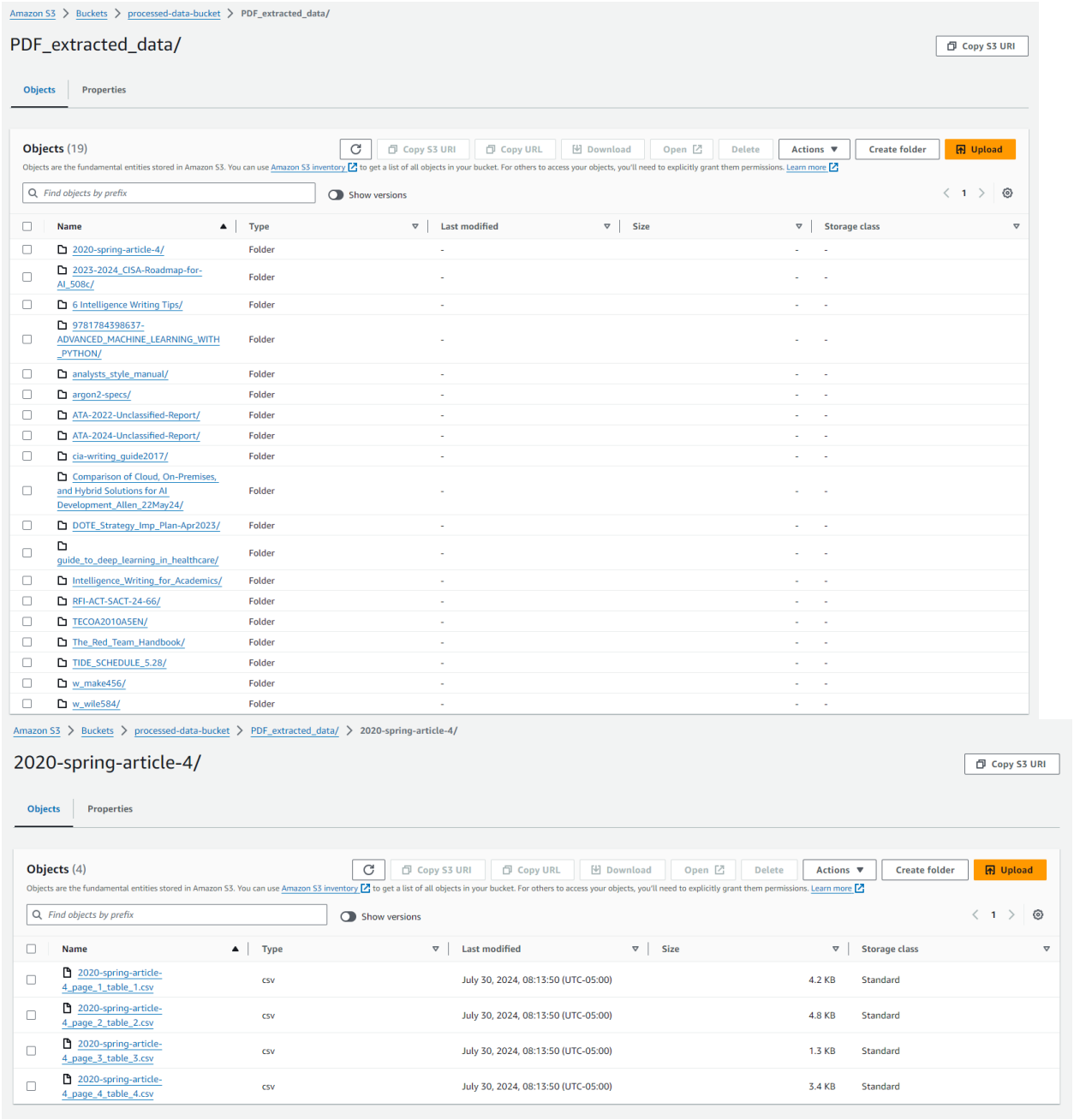
Find objects by prefix — Show versions

Name	Type	Last modified	Size	Storage class
misc_artifacts/	Folder	-	-	-
model_outputs/	Folder	-	-	-
PDF_extracted_data/	Folder	-	-	-
PDF_extracted_images/	Folder	-	-	-
PDF_summaries/	Folder	-	-	-
vector_stores/	Folder	-	-	-

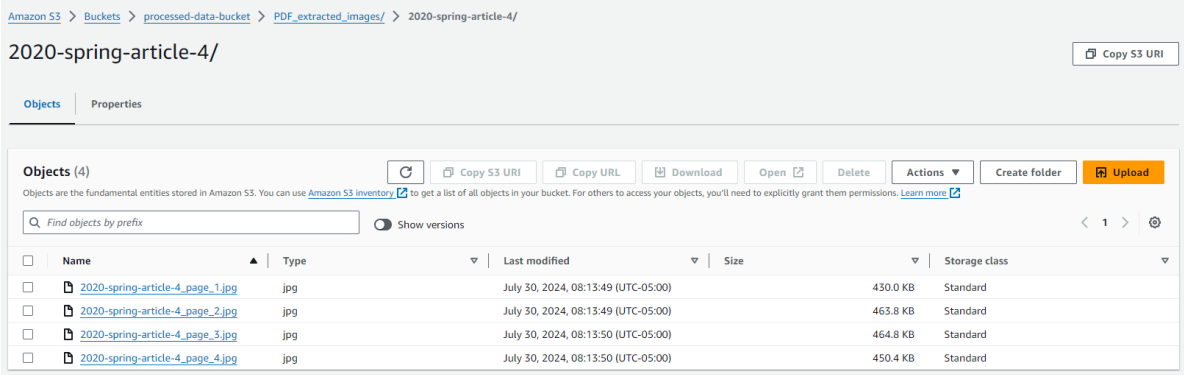
There are several tools under current development that will store artifacts, outputs for various models, and vector stores in the appropriate subfolder in this bucket. This is done to ensure data governance and documenting the lineage of all data as it is processed in the TIDE system. In this way, if there is ever an issue with a model, data source, or model output, the TIDE team can quickly map the lineage of the data in the system to determine how it was processed, by which function, and help determine why the system created that output.

Currently, there are six subfolders in the 'processed-data-bucket'. However, the PDF summary and image/table extraction tools focus on the following subfolders:

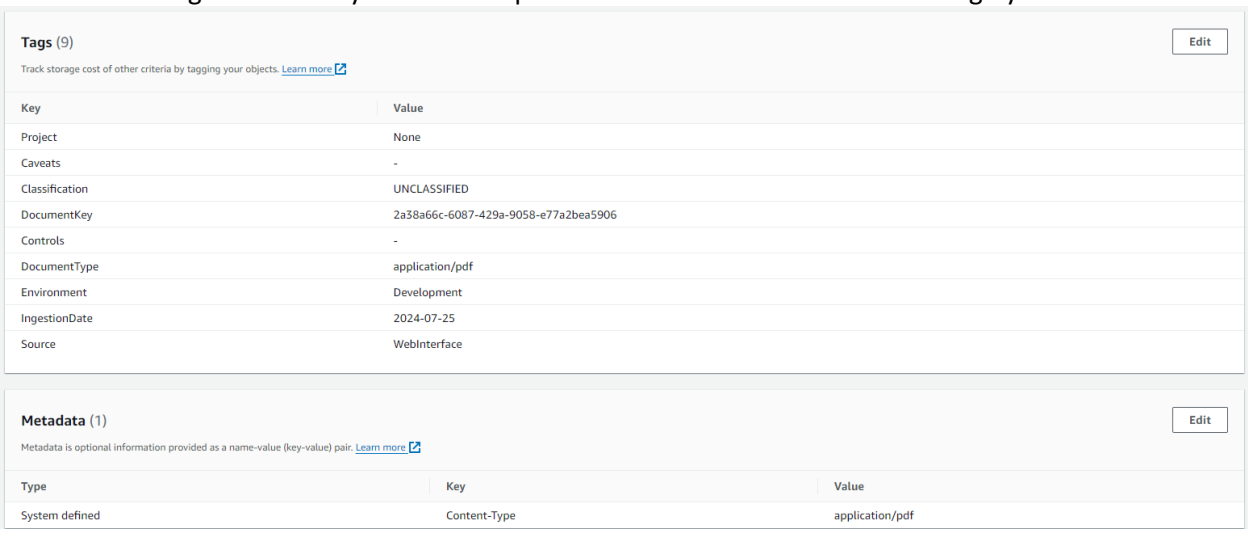
- **PDF_extracted_data:** The repository for all PDF tables extracted by the 'extract-pdf-image-tables' function. Each table is extracted, saved as a .csv, tagged, and organized separately by the page it is found in the PDF.



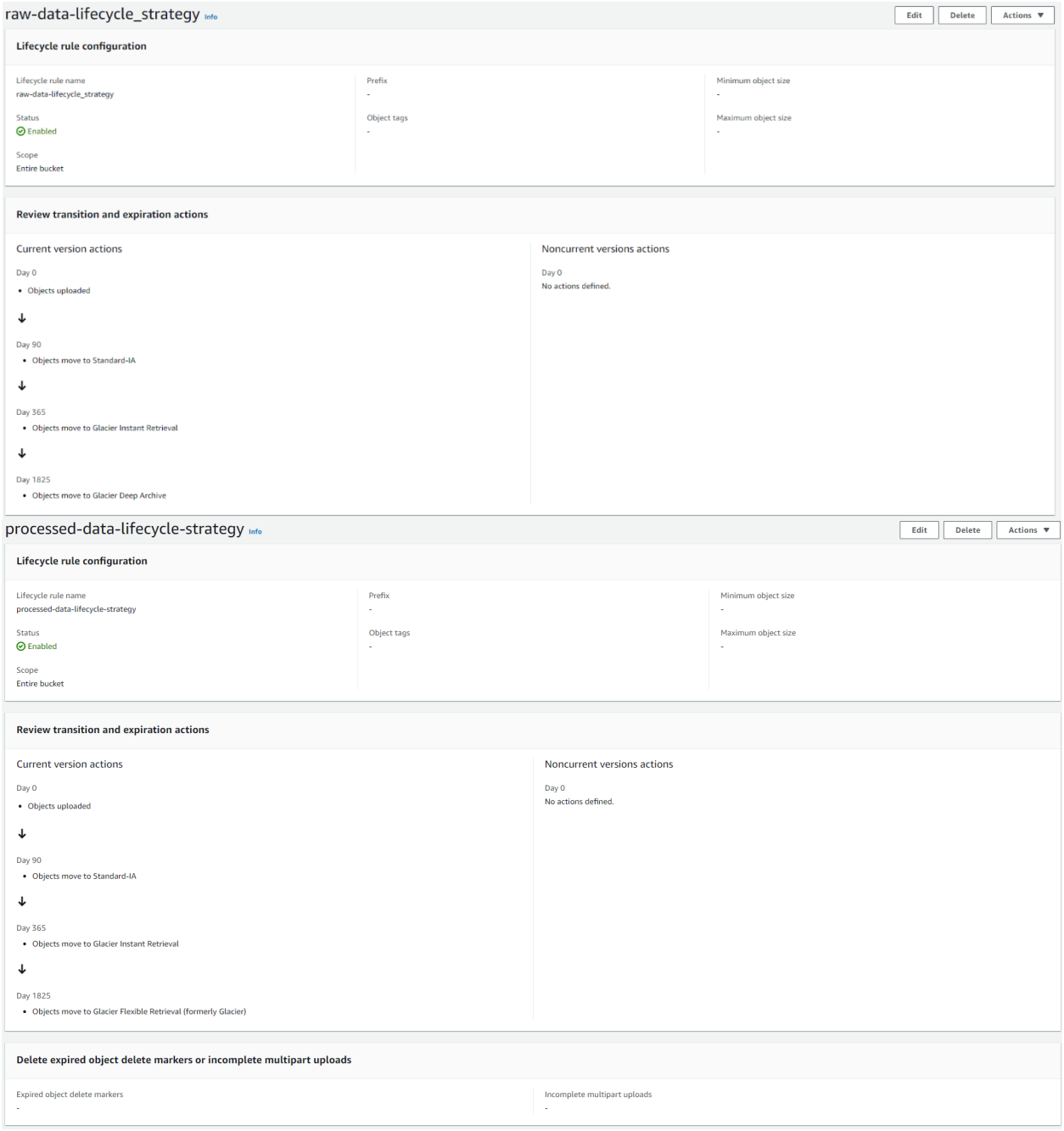
- **PDF_extracted_images:** Organized similarly but provides all the images extracted by the 'extract-pdf-image-tables' function in .jpg format.



- **PDF_summaries:** The repository for all PDF summary and keyword analysis produced by the 'process-pdf-documents-ecr' function. This is the function that provides all summaries and keywords. Note that when a summary is generated, the keywords appear in the document file as well as in the tags so that they can be incorporated into the UI for easier searching by the user.

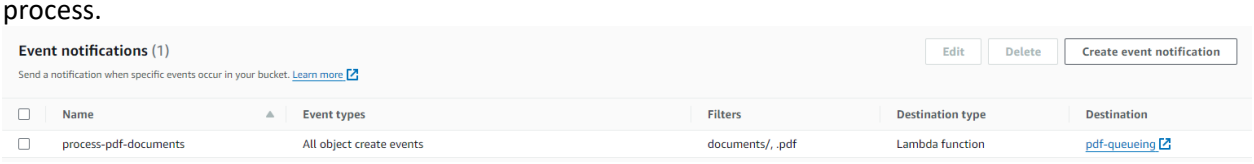


Ensure that the lifecycle rules are set up for each S3 bucket. This ensures that compute and storage costs are controlled as the system scales. The current lifecycle rules for both the 'raw-data-bucket' and 'processed-data-bucket' are shown below:



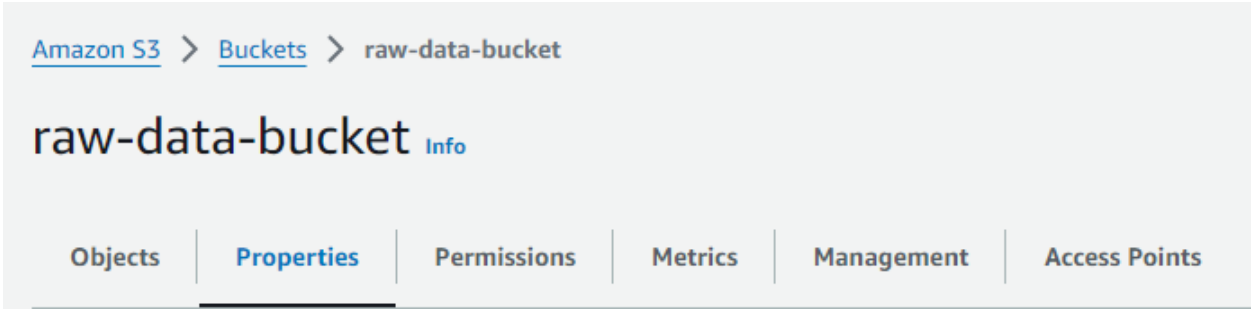
EVENT NOTIFICATION FOR 'RAW-DATA-BUCKET'

When a PDF is saved by the user in the TIDE UI, it is automatically uploaded to the 'raw-data-bucket/documents' bucket. This event triggers the entire PDF summary and image/table extraction process.

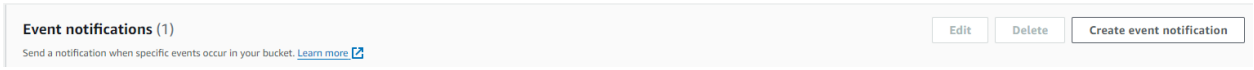


It is important to ensure this event notification is set up properly. Follow these steps:

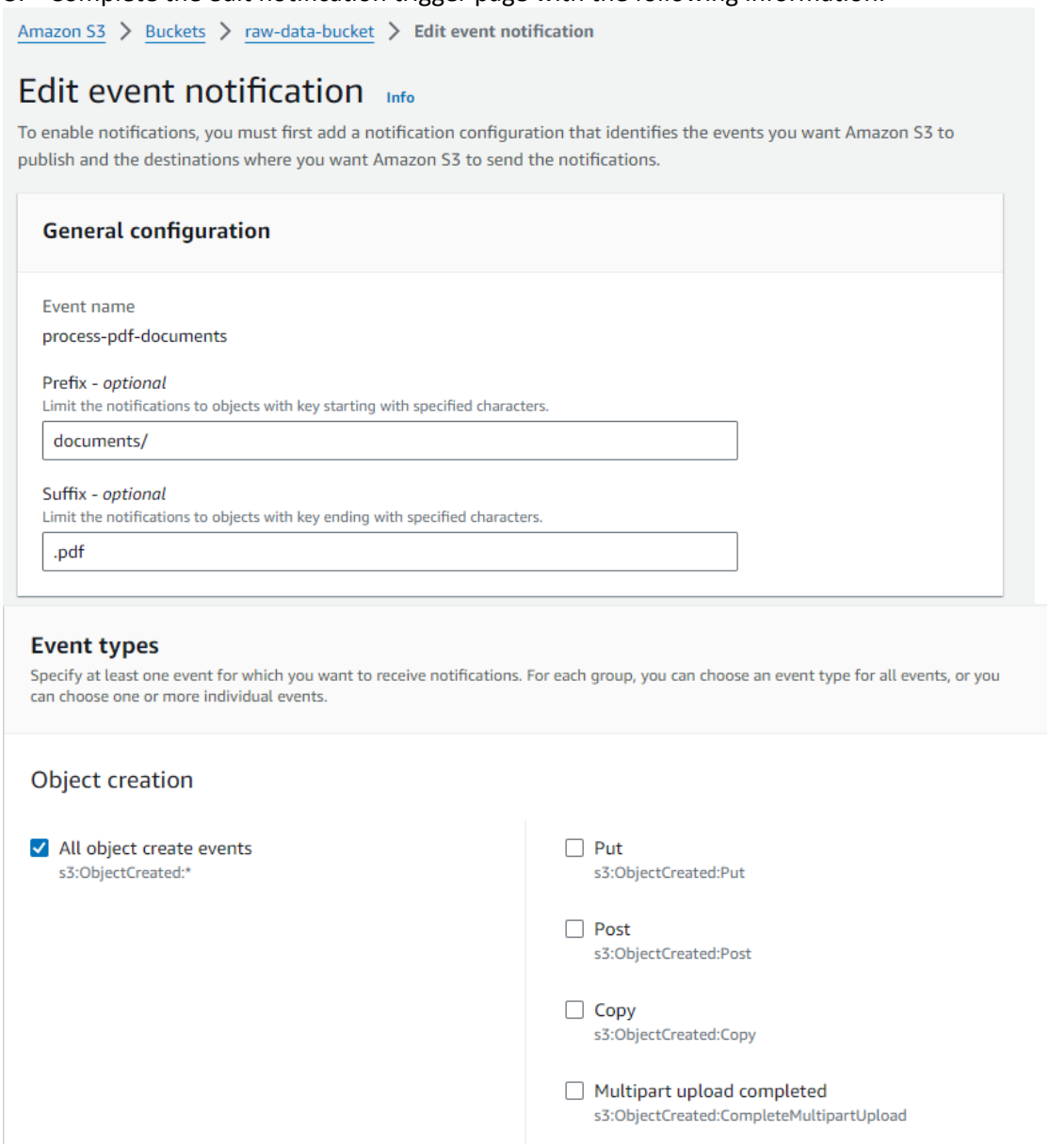
1. Select the "Properties" tab in the 'raw-data-bucket' console in AWS.

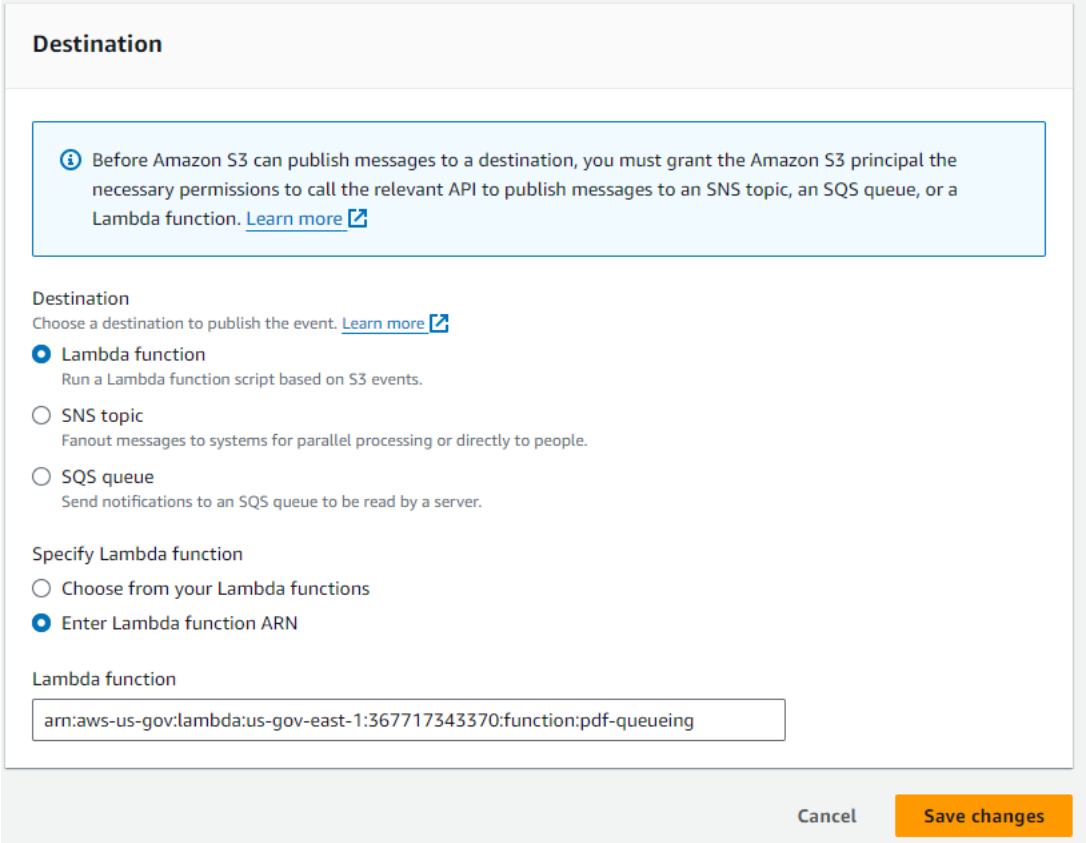


2. Select 'Create event Notification'.



3. Complete the edit notification trigger page with the following information:





4. When complete, click 'Save Changes', and the event notification trigger should be successfully created.

Note: For the current set of tools, there is no event notification for the 'processed-data-bucket' as the Step Functions process orchestrates the entire workflow through the system when new data is uploaded to the raw-data-bucket after being triggered by the initial event notification.

LAMBDA FUNCTION TO INITIATE PDF QUEUEING PROCESS 'PDF-QUEUEING'

The primary purpose is to handle events triggered by Amazon S3 when new objects are uploaded. It checks if the uploaded object exists in the S3 bucket and then initiates an AWS Step Functions state machine to process the object.

Dependencies:

- **AWS Lambda Environment:**
  - Ensure the Lambda function has execution role permissions to interact with S3 and Step Functions.
  - The Lambda function should be set up in the AWS GovCloud (US) region (e.g. us-gov-east-1).
- **Boto3 Library:** The AWS SDK for Python (Boto3) is required for interacting with AWS services. This is typically included in the AWS Lambda Python runtime.

Key Points:

1. **Step Functions State Machine ARN:** The ARN (arn:aws-us-gov:states:us-gov-east-1:367717343370:stateMachine:PDF_Summary_State_Machine) must be valid, and the Lambda function must have permission to start executions on this state machine.
2. **Error Handling:** The code includes error handling for checking the existence of the S3 object and starting the Step Functions execution.
3. **Event Structure:** The function expects the event to have a specific structure, particularly that generated by S3 bucket notifications.

Example Usage: The Lambda function is triggered by an S3 event when a new object is uploaded. It validates the object's existence in the bucket and then starts a Step Functions state machine to process the object. This setup is useful for workflows where S3 objects (e.g. PDFs) need to be processed by a series of tasks defined in a state machine.

CURRENT BUILD FILES AND PROCESS: The 'pdf-queueing' function is small enough to upload to Lambda via S3. We do not currently use a Docker image for its deployment. Follow the steps below for building and uploading this function to AWS:

Run this script in PowerShell on a local machine:

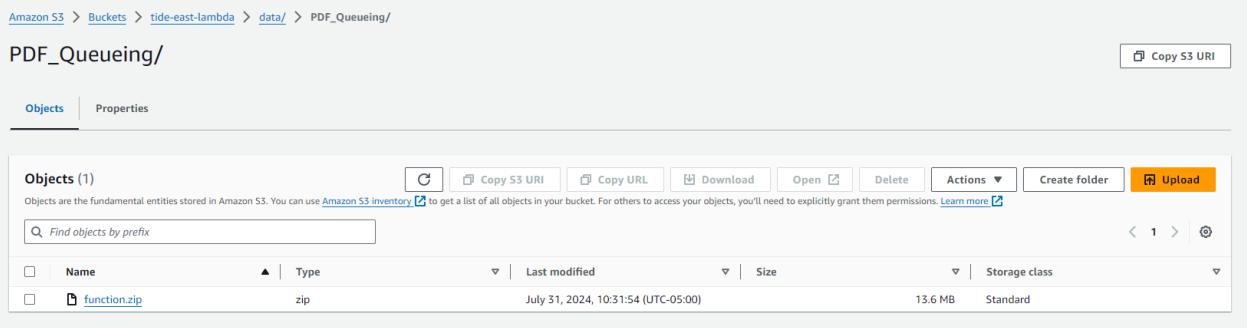
```
# Define variables
$functionName = "function"
$zipFileName = "$functionName.zip"
$requirementsFileName = "requirements.txt"
$buildDir = Join-Path -Path (Get-Location) -ChildPath "build"
$srcDir = Join-Path -Path (Get-Location) -ChildPath "src"

# Create source directory if it doesn't exist
if (-Not (Test-Path $srcDir)) {
    New-Item -ItemType Directory -Path $srcDir
}

# Create the Lambda function script
$lambdaFunctionContent = @"
import json
import boto3
import urllib.parse
import time
from botocore.exceptions import ClientError

# Initialize the Step Functions client
stepfunctions_client = boto3.client('stepfunctions', region_name='us-gov-east-1')
s3_client = boto3.client('s3', region_name='us-gov-east-1')

def fix_object_key(object_key):
    """
    Ensure the object key is properly formatted for the process.
```

```
    This function can include any necessary corrections to the object key.
    """
    return urllib.parse.unquote(object_key)

def check_s3_object_exists(bucket, key, retries=3, delay=2):
    for attempt in range(retries):
        try:
            s3_client.head_object(Bucket=bucket, Key=key)
            return True
        except ClientError as e:
            if e.response['Error']['Code'] == '404':
                time.sleep(delay)
            else:
                raise e
    return False

def lambda_handler(event, context):
    state_machine_arn = 'arn:aws-us-gov:states:us-gov-east-
1:367717343370:stateMachine:PDF_Summary_State_Machine'

    print(f"Received event: {json.dumps(event, indent=2)}")

    if 'Records' not in event:
        print("Error: No Records found in the event.")
        return {
            'statusCode': 400,
            'body': json.dumps('Error: No Records found in the event.')
        }

    for record in event['Records']:
        if 's3' not in record or 'bucket' not in record['s3'] or 'name' not in record['s3']['bucket'] or 'key' not in
record['s3']['object']:
            print("Error: Invalid record structure.")
            continue

        bucket_name = record['s3']['bucket']['name']
        object_key = record['s3']['object']['key']
        corrected_object_key = fix_object_key(object_key)
        encoded_object_key = urllib.parse.quote(corrected_object_key, safe='')

        print(f"Original Bucket: {bucket_name}, Original Key: {object_key}, Corrected Key:
{corrected_object_key}, Encoded Key: {encoded_object_key}")

        if check_s3_object_exists(bucket_name, corrected_object_key):
```

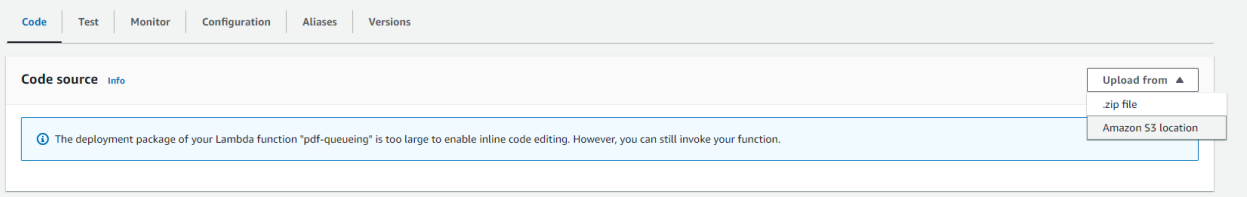
```
        step_function_input = {
            'Records': [{
                's3': {
                    'bucket': {'name': bucket_name},
                    'object': {'key': corrected_object_key}
                }
            }]
        }

        try:
            response = stepfunctions_client.start_execution(
                stateMachineArn=state_machine_arn,
                input=json.dumps(step_function_input)
            )
            print(f"Started Step Function execution: {response['executionArn']}")
        except Exception as e:
            print(f"Error starting Step Function execution: {str(e)}")
            raise
    else:
        print(f"Error: S3 object not found - Bucket: {bucket_name}, Key: {corrected_object_key}")

    return {
        'statusCode': 200,
        'body': json.dumps('State machine execution started successfully for all documents')
    }
"@

# Save the Lambda function script to the source directory
Set-Content -Path "$srcDir\lambda_function.py" -Value $lambdaFunctionContent

# Create a clean build directory
if (Test-Path $buildDir) {
    Remove-Item $buildDir -Recurse -Force
}
New-Item -ItemType Directory -Path $buildDir

# Copy source code to build directory
Copy-Item -Path "$srcDir\*" -Destination $buildDir -Recurse

# Create requirements.txt for dependencies
$requirementsContent = @"
boto3
botocore
"@
```

```
Set-Content -Path "$buildDir\$requirementsFileName" -Value $requirementsContent

# Install dependencies to the build directory
pip install -r "$buildDir\$requirementsFileName" -t $buildDir

# Create the Lambda deployment package
if (Test-Path $zipFileName) {
    Remove-Item $zipFileName -Force
}
Add-Type -AssemblyName System.IO.Compression.FileSystem
[System.IO.Compression.ZipFile]::CreateFromDirectory($buildDir, (Join-Path -Path (Get-Location) -
ChildPath $zipFileName))

# Clean up build directory
Remove-Item $buildDir -Recurse -Force

Write-Host "Lambda package $zipFileName created successfully."
```

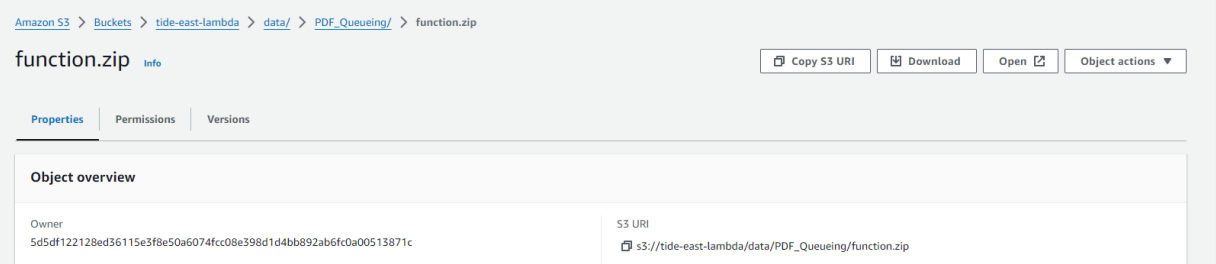
This process builds a .zip folder on the local machine called 'function.zip' that needs to be uploaded manually to this bucket and subfolder in S3:



Once the 'function.zip' file is uploaded to 'tide-east-lambda', call it from the 'pdf-queueing' Lambda function from the code source dialogue by selecting 'Upload from' > 'Amazon S3 location'.



This will prompt you to supply the URI of the 'function.zip fil, which can be found in S3 as shown:



Successful upload will be noted by a green box indicating the .zip file was properly uploaded to S3.

LAMBDA FUNCTION FOR PDF SUMMARY: 'PROCESS-PDF-DOCUMENTS-ECR'

This function is an AWS Lambda function designed to process PDF files stored in an S3 bucket. It performs text extraction, keyword extraction, summarization, and then uploads a modified version of the PDF back to S3 with additional metadata and tags.

Dependencies:

- **AWS Lambda Environment:** Ensure the Lambda function has the necessary execution role permissions to interact with S3 and CloudWatch.
- **The Lambda function should be set up in the appropriate AWS region.**
- **Python Libraries:**
  - boto3
  - fitz (PyMuPDF)
  - nltk
  - transformers
  - reportlab

Example Usage: The Lambda function is triggered by an S3 event when a new PDF is uploaded. It performs several steps to process the PDF including text extraction, keyword extraction, summarization, and re-uploading the processed PDF to S3 with updated metadata and tags. This setup is useful for workflows that require automated processing and summarization of PDF documents.

CURRENT BUILD FILES AND PROCESS: To successfully build and push Docker image containers to ECR to enable proper execution of this function, the developer must utilize several scripts shown below from their local system, build the image with Docker Desktop, and push to an authenticated ECR instance. The files and build commands are shown below:

Dockerfile (text document):

```
# Use the Amazon Linux 2 as the base image
FROM public.ecr.aws/lambda/python:3.11

# Set environment variables for Hugging Face and NLTK data paths
```

```
ENV HF_HOME=/tmp/transformers_cache
ENV NLTK_DATA=/opt/python/nltk_data

# Copy the function code and requirements
COPY lambda_function.py ${LAMBDA_TASK_ROOT}
COPY requirements.txt .
COPY download_nltk_data.py .

# Install the dependencies
RUN pip install -r requirements.txt -t ${LAMBDA_TASK_ROOT}

# Download NLTK data
RUN python download_nltk_data.py

# Command to run the Lambda function
CMD ["lambda_function.lambda_handler"]
```

download_nltk_data.py:

```
import ssl
import nltk

# Disable SSL certificate verification
try:
    _create_unverified_https_context = ssl._create_unverified_context
except AttributeError:
    # Legacy Python that doesn't verify HTTPS certificates by default
    pass
else:
    ssl._create_default_https_context = _create_unverified_https_context

# Download the NLTK data
nltk.download('punkt', download_dir='/opt/python/nltk_data')
nltk.download('stopwords', download_dir='/opt/python/nltk_data')
nltk.download('averaged_perceptron_tagger', download_dir='/opt/python/nltk_data')
```

lambda_function.py (this is the main function script):

```
import os
import boto3
import fitz  # PyMuPDF
import re
import nltk
from datetime import datetime
```

```
from reportlab.lib.pagesizes import letter
from reportlab.pdfgen import canvas
from reportlab.lib.utils import simpleSplit
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize, sent_tokenize
from nltk import pos_tag
from collections import Counter
import logging
import time
from transformers import pipeline, AutoModelForSeq2SeqLM, AutoTokenizer
from concurrent.futures import ThreadPoolExecutor
from botocore.exceptions import NoCredentialsError, PartialCredentialsError, ClientError
import urllib.parse
import json

# Set environment variables for Hugging Face and NLTK data paths
os.environ['HF_HOME'] = '/tmp/transformers_cache'
os.environ['NLTK_DATA'] = '/opt/python/nltk_data'

# Set the NLTK data path to point to the correct directory within the Lambda layer
nltk.data.path.append('/opt/python/nltk_data')

# Initialize the logger
logging.basicConfig(level=logging.INFO)

# Initialize AWS clients
s3_client = boto3.client('s3')
cloudwatch_client = boto3.client('cloudwatch')
sns_client = boto3.client('sns')

def log_to_cloudwatch(namespace, metric_name, value, dimensions=[]):
    try:
        cloudwatch_client.put_metric_data(
            Namespace=namespace,
            MetricData=[
                {
                    'MetricName': metric_name,
                    'Dimensions': dimensions,
                    'Value': value,
                    'Unit': 'Count'
                }
            ]
        )
        logging.info(f"Successfully logged {metric_name} to CloudWatch")
```

```
    except Exception as e:
        logging.error(f"Failed to log {metric_name} to CloudWatch: {str(e)}")

# Load model and tokenizer separately
model_name = "facebook/bart-large-cnn"
model = AutoModelForSeq2SeqLM.from_pretrained(model_name,
cache_dir="/tmp/transformers_cache")
tokenizer = AutoTokenizer.from_pretrained(model_name, cache_dir="/tmp/transformers_cache")
summarizer = pipeline("summarization", model=model, tokenizer=tokenizer)

def add_page_header(c, width, height, header_text):
    c.setFont("Helvetica-Bold", 12)
    c.drawCentredString(width / 2, height - 50, header_text)
    c.setFont("Helvetica", 10)

def get_object_with_retries(bucket_name, key, context, retries=10, delay=5):
    for attempt in range(retries):
        try:
            response = s3_client.get_object(Bucket=bucket_name, Key=key)
            logging.info(f"Successfully retrieved object {key} from bucket {bucket_name}")
            return response
        except s3_client.exceptions.NoSuchKey as e:
            logging.error(f"Attempt {attempt + 1}: File {key} does not exist in bucket {bucket_name}: {e}")
            if attempt < retries - 1:
                logging.info(f"Retrying in {delay} seconds...")
                time.sleep(delay)
            else:
                return None
        except (NoCredentialsError, PartialCredentialsError) as e:
            logging.error(f"Credentials error: {str(e)}")
            return None
        except ClientError as e:
            if e.response['Error']['Code'] == 'AccessDenied':
                logging.error(f"Access denied when trying to retrieve {key} from {bucket_name}: {str(e)}")
                return None
            logging.error(f"Client error: {str(e)}")
            if attempt < retries - 1:
                logging.info(f"Retrying in {delay} seconds...")
                time.sleep(delay)
            else:
                return None

def download_pdf_from_s3_and_get_tags(bucket_name, file_name, context):
    local_path = f"/tmp/{os.path.basename(file_name)}"
```

```
    response = get_object_with_retries(bucket_name, file_name, context)
    if response is None:
        return None, None, None, None

    try:
        with open(local_path, 'wb') as f:
            f.write(response['Body'].read())

        tagging_info = s3_client.get_object_tagging(Bucket=bucket_name, Key=file_name)
        tags = {item['Key']: item['Value'] for item in tagging_info['TagSet']}

        metadata = response.get('Metadata', {})
        content_type = response.get('ContentType', None)
        return local_path, tags, metadata, content_type

    except s3_client.exceptions.NoSuchKey as e:
        logging.error(f"File {file_name} does not exist in bucket {bucket_name}: {e}")
        return None, None, None, None
    except Exception as e:
        logging.error(f"Error retrieving object {file_name} from bucket {bucket_name}: {str(e)}")
        raise

def extract_text_from_pdf(pdf_path):
    try:
        doc = fitz.open(pdf_path)
        text = ''
        for page in doc:
            page_text = page.get_text("text")
            lines = page_text.split('\n')
            filtered_lines = [line for line in lines if not re.match(r'^\d+(\.\d+)*\s+', line) and len(line.split()) >
4]
            text += ' '.join(filtered_lines) + ' '
        doc.close()
        logging.info(f"Extracted text: {text[:500]}...")  # Log the first 500 characters of the extracted text
        return text
    except Exception as e:
        logging.error(f"Failed to extract text from PDF {pdf_path}: {str(e)}")
        return None

def clean_text(text):
    text = re.sub(r'\[\s*\d+\s*\]', '', text)
    text = re.sub(r'[\u25A0-\u25FF]', '', text)
    text = re.sub(r'[^\x00-\x7F]+', ' ', text)
```

```
    text = re.sub(r'\s+', ' ', text)
    return text

def extract_key_nouns(text, num_keywords=10):
    if text is None:
        return []
    text = clean_text(text)
    words = word_tokenize(text)
    words_pos = pos_tag(words)

    # Extract nouns and proper nouns
    nouns = [word for word, pos in words_pos if pos in ['NN', 'NNS', 'NNP', 'NNPS']]

    # Use a set of stopwords
    stop_words = set(stopwords.words('english'))

    # Filter out stopwords and single characters
    filtered_nouns = [word for word in nouns if word.lower() not in stop_words and len(word) > 1]

    # Count the frequency of nouns
    noun_counts = Counter(filtered_nouns)

    # Get the most common nouns
    most_common_nouns = [word for word, _ in noun_counts.most_common(num_keywords)]

    # Capitalize proper nouns
    proper_nouns = [word.capitalize() if pos in ['NNP', 'NNPS'] else word for word, pos in
pos_tag(most_common_nouns)]

    return proper_nouns  # Return the list of keywords

def summarize_text(text, max_length=300, min_length=100, chunk_size=512):
    if text is None:
        return "No text available."

    text = clean_text(text)

    sentences = nltk.sent_tokenize(text)
    chunks = []
    current_chunk = []
    current_length = 0

    for sentence in sentences:
        sentence_length = len(nltk.word_tokenize(sentence))
```

```
        if current_length + sentence_length <= chunk_size:
            current_chunk.append(sentence)
            current_length += sentence_length
        else:
            chunks.append(" ".join(current_chunk))
            current_chunk = [sentence]
            current_length = sentence_length

    if current_chunk:
        chunks.append(" ".join(current_chunk))

    if not chunks:
        chunks.append(text)

    logging.info(f"Total chunks created: {len(chunks)}")

    summaries = []
    with ThreadPoolExecutor() as executor:
        futures = [executor.submit(summarizer, chunk, max_length=max_length, min_length=min_length,
do_sample=False) for chunk in chunks]
        for future in futures:
            try:
                summary = future.result()
                logging.info(f"Summary chunk: {summary}")
                summaries.append(summary[0]['summary_text'])
            except Exception as e:
                logging.error(f"Error summarizing chunk: {str(e)}")

    summary_text = "\n".join(["• " + summary for summary in summaries])

    logging.info(f"Final Summary: {summary_text}")

    return summary_text

def print_section(c, title, content, start_y, width, height, indent=40, bullet_point=True):
    c.setFont("Helvetica-Bold", 12)
    c.drawString(indent, start_y, title)
    start_y -= 24

    c.setFont("Helvetica", 10)
    text_object = c.beginText(indent, start_y)
    points = content.split('\n') if bullet_point else [content]
    for point in points:
        wrapped_lines = simpleSplit(point, "Helvetica", 10, width - 2 * indent)
```

```
        first_line = True
        for line in wrapped_lines:
            if text_object.getY() < 50:
                c.drawText(text_object)
                c.showPage()
                text_object = c.beginText(indent, height - 100)
                c.setFont("Helvetica", 10)
                add_page_header(c, width, height - 25, "Continued...")
            if first_line:
                text_object.textLine('• ' + line if bullet_point else line)
                first_line = False
            else:
                text_object.textLine('  ' + line)
    c.drawText(text_object)
    return text_object.getY() - 20

def save_to_pdf(file_name, keywords, general_summary, file_path):
    c = canvas.Canvas(file_path, pagesize=letter)
    width, height = letter
    current_time = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    header_text = f"{file_name} - Processed on {current_time}"
    add_page_header(c, width, height, header_text)

    y_position = height - 100
    keywords_text = '; '.join(keywords)
    y_position = print_section(c, "Keywords", keywords_text, y_position, width, height,
bullet_point=False)
    y_position = print_section(c, "General Summary", general_summary, y_position, width, height,
bullet_point=False)

    c.save()

def clean_tag_value(value):
    value = re.sub(r'[^a-zA-Z0-9\s_\.\:/=\+\-@]', '', value)
    return value[:256]

def upload_file_to_s3_with_modified_tags(bucket_name, file_path, s3_path, tags, content_type,
keywords):
    tags['Source'] = 'PDF_Summary_v4.ipynb'
    tags['Keywords'] = clean_tag_value(" ".join(keywords))
    tag_set = [{'Key': key, 'Value': clean_tag_value(value)} for key, value in tags.items()]

    s3_client.upload_file(
        Filename=file_path,
```

```
        Bucket=bucket_name,
        Key=s3_path,
        ExtraArgs={
            'ContentType': content_type if content_type else 'application/pdf',
            'Metadata': {}
        }
    )

    s3_client.put_object_tagging(
        Bucket=bucket_name,
        Key=s3_path,
        Tagging={'TagSet': tag_set}
    )

def lambda_handler(event, context):
    file_name = None
    try:
        logging.info(f"Event: {event}")

        # Log the entire event for debugging
        logging.debug(f"Event structure: {json.dumps(event, indent=2)}")

        if 'Records' in event and event['Records']:
            bucket_name = event['Records'][0]['s3']['bucket']['name']
            key = event['Records'][0]['s3']['object']['key']
        elif 'bucket_name' in event and 'object_key' in event:
            bucket_name = event['bucket_name']
            key = event['object_key']
        else:
            raise KeyError("Event does not contain 'bucket_name' or 'object_key' key")

        file_name = os.path.basename(key)
        output_bucket_name = "processed-data-bucket"
        subfolder_path = "PDF_summaries/"

        logging.info(f"Bucket: {bucket_name}, Key: {key}")

        local_pdf_path, tags, metadata, content_type =
download_pdf_from_s3_and_get_tags(bucket_name, key, context)
        if local_pdf_path is None:
            logging.error(f"File {file_name} does not exist in bucket {bucket_name}. Skipping processing.")
            return {
                'statusCode': 404,
                'body': json.dumps(f"File {file_name} does not exist in bucket {bucket_name}.")
```

```
        }

    pdf_text = extract_text_from_pdf(local_pdf_path)
    logging.info(f"Extracted PDF text: {pdf_text[:500]}...")

    keywords = extract_key_nouns(pdf_text, num_keywords=10)
    general_summary = summarize_text(pdf_text)

    logging.info(f"Keywords: {keywords}")
    logging.info(f"General Summary: {general_summary}")

    local_pdf_output_path = f"/tmp/{file_name}"
    save_to_pdf(file_name, keywords, general_summary, local_pdf_output_path)

    output_s3_file_name = f"{subfolder_path}{os.path.basename(local_pdf_output_path)}"
    upload_file_to_s3_with_modified_tags(output_bucket_name, local_pdf_output_path,
output_s3_file_name, tags, content_type, keywords)

    logging.info(f"Processing and uploading completed successfully for {file_name}")

    # Publish to SNS Topic
    sns_topic_arn = os.environ['SNS_TOPIC_ARN']
    s3_url = f"https://{output_bucket_name}.s3.amazonaws.com/{output_s3_file_name}"
    message = {
        "bucket_name": output_bucket_name,
        "file_key": output_s3_file_name,
        "s3_url": s3_url
    }
    sns_client.publish(
        TopicArn=sns_topic_arn,
        Message=json.dumps({'default': json.dumps(message)}),
        MessageStructure='json'
    )

    log_to_cloudwatch(
        namespace='MyLambdaFunction',
        metric_name='SuccessfulProcessing',
        value=1,
        dimensions=[
            {'Name': 'FunctionName', 'Value': 'SummarizePDF'},
            {'Name': 'FileName', 'Value': file_name}
        ]
    )
    return {
```

```
            'statusCode': 200,
            'body': json.dumps('Processing and uploading completed successfully')
        }

    except KeyError as e:
        logging.error(f"KeyError: {str(e)}")
        logging.debug(f"Event structure at error: {json.dumps(event, indent=2)}")
        log_to_cloudwatch(
            namespace='MyLambdaFunction',
            metric_name='KeyError',
            value=1,
            dimensions=[
                {'Name': 'FunctionName', 'Value': 'SummarizePDF'}
            ]
        )
        return {
            'statusCode': 400,
            'body': json.dumps(f"KeyError: {str(e)}")
        }
    except Exception as e:
        logging.error(f"Error processing {file_name if file_name else 'unknown file'}: {str(e)}")
        log_to_cloudwatch(
            namespace='MyLambdaFunction',
            metric_name='ProcessingError',
            value=1,
            dimensions=[
                {'Name': 'FunctionName', 'Value': 'SummarizePDF'}
            ]
        )
        return {
            'statusCode': 500,
            'body': json.dumps(f"Error processing {file_name if file_name else 'unknown file'}: {str(e)}")
        }
```

requirements.txt file

```
Boto3
PyMuPDF
nltk
reportlab
rake-nltk
transformers
torch
```

Build and Push Commands using Windows Powershell to Authenticate Docker and PowerShell with AWS Account:

Get the Access key information from your AWS Account home page:

#PowerShell Commands to Authenticate AWS and ECR

$Env:AWS_ACCESS_KEY_ID="Access Key from Account Page"
$Env:AWS_SECRET_ACCESS_KEY="Secreet Access Key from Account Page"
$Env:AWS_SESSION_TOKEN="Session Token from Access Page - This will be long"
$env:AWS_REGION="us-gov-east-1"

aws configure

AWS Access Key ID [****************AOFT]: Provide the access key again
AWS Secret Access Key [****************Py1f]: Provide the secret access key again
Default region name [us-gov-east-1]: us-gov-east-1
Default output format [json]: json

Get-STSCallerIdentity

(Get-ECRLoginCommand -Region us-gov-east-1).Password | docker login --username AWS --password-stdin 367717343370.dkr.ecr.us-gov-east-1.amazonaws.com

Output should read 'Login Successful"

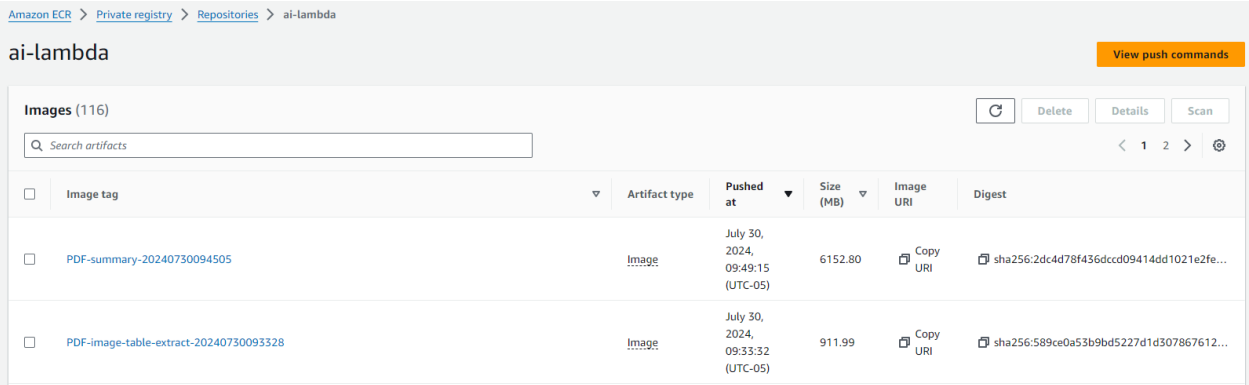
Push Commands for building and pushing function Docker images to ECR:

#Build and Push Commands General format
cd C:\<local directory to files>
docker build -t lambda_function .
$TAG = Get-Date -Format "yyyyMMddHHmmss"
docker tag lambda_function:latest 367717343370.dkr.ecr.us-gov-east-1.amazonaws.com/ai-lambda:lambda_function-$TAG
docker push 367717343370.dkr.ecr.us-gov-east-1.amazonaws.com/ai-lambda:lambda_function-$TAG

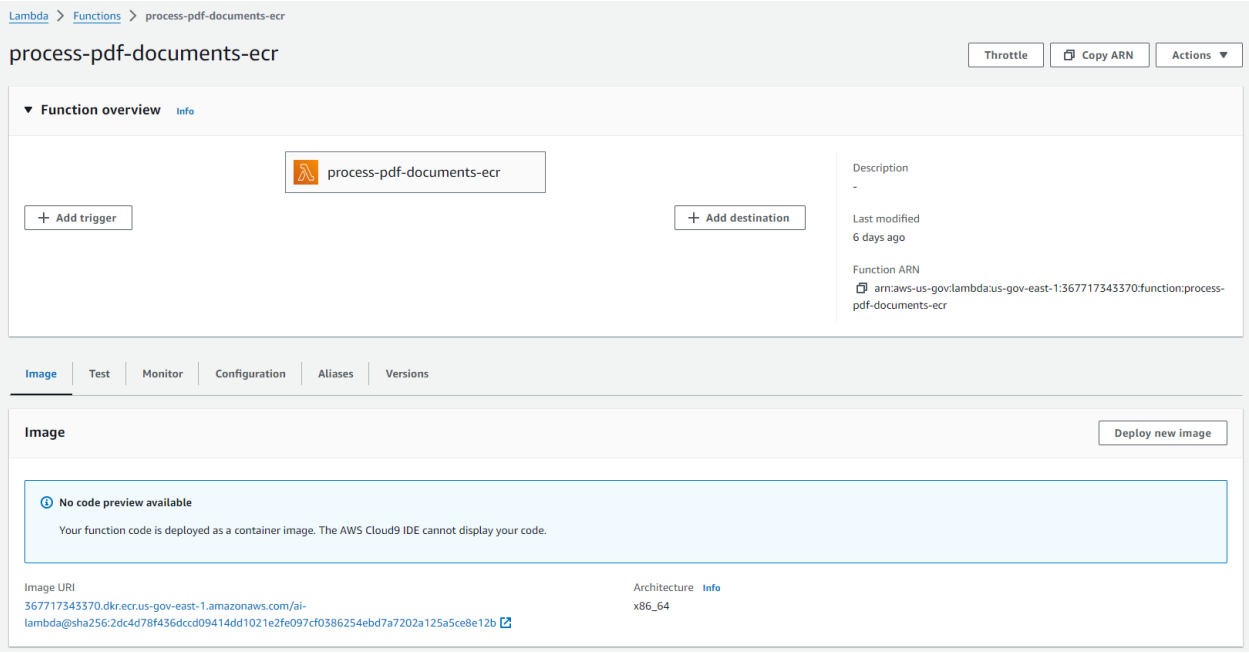
Build and push Docker image for PDF Summary
cd C:\<local directory to files>

docker build -t lambda_function .
$TAG = Get-Date -Format "yyyyMMddHHmmss"
docker tag lambda_function:latest 367717343370.dkr.ecr.us-gov-east-1.amazonaws.com/ai-lambda:PDF-summary-$TAG
docker push 367717343370.dkr.ecr.us-gov-east-1.amazonaws.com/ai-lambda:PDF-summary-$TAG

Update Image from ECR: Once the image is successfully built and pushed to the ECR repo 'ai-lambda', the Lambda function must be updated so that it accesses the correct image. Go to 'ai-lambda' repo in ECR and select the appropriate image for the function. All images are labeled by their purpose and tagged with the time and date as shown below:



In the Lambda function, select 'deploy new image' and search for the updated image . A green successful image update message should be depicted at the top of the screen in Lambda.



LAMBDA FUNCTION FOR PDF IMAGE AND TABLE EXTRACTION

This function is an AWS Lambda function designed to process PDF files stored in an S3 bucket. It extracts images and tables from the PDF, performs OCR on the images, and saves the extracted data as CSV files. The extracted images and tables are then uploaded back to S3 with modified tags.

Dependencies:

- **AWS Lambda Environment:**
  - Ensure the Lambda function has the necessary execution role permissions to interact with S3 and CloudWatch.
  - The Lambda function should be set up in the appropriate AWS region.
- **Python Libraries:**
  - boto3: AWS SDK for Python to interact with AWS services.
  - pandas: For handling tabular data.
  - camelot: For extracting tables from PDF files.
  - pytesseract: For OCR processing of images.
  - pdfminer.six: For text extraction from PDF files.
  - PyPDF2: For PDF file handling.
  - pdf2image: For converting PDF pages to images.
  - Pillow: For image handling.

Example Usage: The Lambda function is triggered by an S3 event when a new PDF is uploaded. It performs several steps to process the PDF, including text extraction, image extraction, table extraction, and re-uploading the processed data to S3 with updated metadata and tags. This setup is useful for workflows that require automated processing and extraction of data from PDF documents.

Key Points:

- **Helper Functions:**
  - log_to_cloudwatch: Logs metrics to AWS CloudWatch.
  - patched_save_page and patched_read_pdf: Patches Camelot to use PdfReader instead of PdfFileReader.
  - download_pdf_from_s3_and_get_tags: Downloads a PDF from S3 and retrieves its tags and metadata.
  - describe_image_content: Uses OCR to describe the content of an image.
  - is_valid_image: Checks if a byte stream is a valid image.
  - extract_images_from_pdf: Extracts images from a PDF file.
  - extract_tables_from_pdf: Extracts tables from a PDF file using Camelot.
  - save_tables_to_csv: Saves extracted tables to CSV files.
  - clean_tag_value: Cleans tag values to be uploaded to S3.
  - upload_files_to_s3_with_modified_tags: Uploads files to S3 with modified tags and metadata.

Steps Involved:

1. **Imports and Initialization:**

- Imports necessary libraries including boto3 for AWS services, camelot for table extraction, pytesseract for OCR, pdfminer.six for text extraction, and pdf2image for converting PDF pages to images.
- Initializes AWS clients for S3 and CloudWatch.
- Configures logging.

2. **Lambda Handler:**
   - Processes the event triggered by S3 when a new object is uploaded.
   - Downloads the PDF from S3 and extracts its text.
   - Extracts images and tables from the PDF.
   - Saves extracted images and tables to the local file system.
   - Uploads the extracted images and tables back to S3 with additional metadata and tags.
   - Logs processing metrics to CloudWatch.
   - Handles errors and logs relevant information.

This function ensures that all images and tables within a PDF are effectively extracted, processed, and stored, facilitating the automated handling and analysis of PDF content within the TIDE AI system.

BUILD AND PUSH COMMANDS

Build these files on your local machine:

Dockerfile

```
# Use Amazon Linux 2 as the base image
FROM amazonlinux:2

# Install system dependencies
RUN yum -y update && \
    amazon-linux-extras install epel && \
    yum install -y \
    gcc \
    gcc-c++ \
    make \
    openssl-devel \
    bzip2-devel \
    libffi-devel \
    wget \
    git \
    tar \
    xz \
    zlib-devel \
    ghostscript \
    poppler-utils \
    pdftohtml \
    tesseract \
    leptonica \
```

```
    lcms2 \
    lcms2-devel \
    libjpeg-devel \
    libpng-devel \
    openjpeg2 \
    openjpeg2-devel \
    cmake \
    perl \
    xz-devel \
    sqlite-devel

# Install OpenSSL from source
RUN curl -O https://www.openssl.org/source/openssl-1.1.1l.tar.gz && \
    tar -xzf openssl-1.1.1l.tar.gz && \
    cd openssl-1.1.1l && \
    ./config --prefix=/usr/local/ssl --openssldir=/usr/local/ssl shared zlib && \
    make && \
    make install && \
    cd .. && \
    rm -rf openssl-1.1.1l.tar.gz openssl-1.1.1l

# Install Python 3.11 from source
RUN curl -O https://www.python.org/ftp/python/3.11.0/Python-3.11.0.tgz && \
    tar -xzf Python-3.11.0.tgz && \
    cd Python-3.11.0 && \
    ./configure --enable-optimizations --with-ensurepip=install --with-openssl=/usr/local/ssl --with-
openssl-rpath=auto && \
    make altinstall && \
    cd .. && \
    rm -rf Python-3.11.0.tgz Python-3.11.0

# Upgrade pip and install awslambdaric and other pip packages
COPY requirements.txt .
RUN python3.11 -m pip install --upgrade pip && \
    python3.11 -m pip install awslambdaric && \
    python3.11 -m pip install -r requirements.txt

# Set the working directory
WORKDIR /app

# Copy the application code
COPY lambda_function.py .

# Define the entry point for the Lambda function
```

```
CMD ["python3.11", "-m", "awslambdaric", "lambda_function.lambda_handler"]
```

Requirements.txt file

```
boto3==1.24.4
pandas==1.3.5
numpy==1.23.4
pillow==8.4.0
pdfminer.six==20211012
pytesseract==0.3.8
ghostscript==0.7
camelot-py==0.10.1
opencv-python==4.5.5.64
PyPDF2==2.10.0
pdf2image==1.16.0
fpdf==1.7.2
```

Lambda_function (main lambda function)

```
import json
import boto3
import os
import re
import pandas as pd
import camelot
import logging
import time
from datetime import datetime
from io import BytesIO
from PIL import Image, UnidentifiedImageError
import pytesseract  # for OCR on images
from pdfminer.high_level import extract_text
from PyPDF2 import PdfReader, PdfWriter
from pdf2image import convert_from_path

# Handle import for importlib.metadata
try:
    from importlib.metadata import version
except ImportError:
    from importlib_metadata import version

# Initialize the logger
logging.basicConfig(level=logging.INFO)
```

```
# Initialize AWS clients
s3_client = boto3.client('s3')
cloudwatch_client = boto3.client('cloudwatch')

def log_to_cloudwatch(namespace, metric_name, value, dimensions=[]):
    try:
        cloudwatch_client.put_metric_data(
            Namespace=namespace,
            MetricData=[
                {
                    'MetricName': metric_name,
                    'Dimensions': dimensions,
                    'Value': value,
                    'Unit': 'Count'
                }
            ]
        )
        logging.info(f"Successfully logged {metric_name} to CloudWatch")
    except Exception as e:
        logging.error(f"Failed to log {metric_name} to CloudWatch: {str(e)}")

# Patch camelot to use PdfReader instead of PdfFileReader
def patched_save_page(self, filepath, page, temp):
    with open(filepath, "rb") as fileobj:
        infile = PdfReader(fileobj)
        writer = PdfWriter()
        writer.add_page(infile.pages[page - 1])
        with open(os.path.join(temp, f"page-{page}.pdf"), "wb") as fp:
            writer.write(fp)

camelot.handlers.PDFHandler._save_page = patched_save_page

original_read_pdf = camelot.read_pdf

def patched_read_pdf(filepath, **kwargs):
    return original_read_pdf(filepath, **kwargs)

camelot.read_pdf = patched_read_pdf

def download_pdf_from_s3_and_get_tags(bucket_name, file_name, retries=3, delay=2):
    local_path = f"/tmp/{os.path.basename(file_name)}"

    for attempt in range(retries):
```

```
        try:
            response = s3_client.get_object(Bucket=bucket_name, Key=file_name)
            with open(local_path, 'wb') as f:
                f.write(response['Body'].read())

            tagging_info = s3_client.get_object_tagging(Bucket=bucket_name, Key=file_name)
            tags = {item['Key']: item['Value'] for item in tagging_info['TagSet']}

            head_response = s3_client.head_object(Bucket=bucket_name, Key=file_name)
            metadata = head_response['Metadata']

            logging.info(f"Successfully retrieved tags and metadata for object {file_name} from bucket
{bucket_name}")
            return local_path, tags, metadata
        except s3_client.exceptions.NoSuchKey as e:
            logging.error(f"File {file_name} does not exist in bucket {bucket_name}: {e}")
            log_to_cloudwatch(
                namespace='MyLambdaFunction',
                metric_name='NoSuchKeyErrors',
                value=1,
                dimensions=[
                    {'Name': 'FunctionName', 'Value': 'PDF_Extraction'}
                ]
            )
            raise
        except Exception as e:
            logging.error(f"Attempt {attempt + 1} failed with error: {str(e)}")
            if attempt < retries - 1:
                logging.info(f"Retrying in {delay} seconds...")
                time.sleep(delay)
            else:
                logging.error(f"All {retries} attempts failed.")
                raise

def describe_image_content(image):
    try:
        text = pytesseract.image_to_string(image).strip()
        if text:
            return " ".join(text.split()[:5])
        else:
            return "No recognizable content"
    except Exception as e:
        logging.error(f"OCR processing failed: {str(e)}")
        return "OCR processing failed"
```

```
def is_valid_image(image_bytes):
    try:
        image = Image.open(BytesIO(image_bytes))
        image.verify()
        return True
    except (UnidentifiedImageError, OSError):
        return False

def extract_images_from_pdf(pdf_path, output_folder, pdf_name):
    images = convert_from_path(pdf_path)
    image_files = []
    for i, image in enumerate(images):
        try:
            image_filename = f"{output_folder}/{pdf_name}_page_{i+1}.jpg"
            image.save(image_filename, 'JPEG')
            image_content = describe_image_content(image)
            image_files.append((image_filename, image_content))
        except UnidentifiedImageError as e:
            logging.error(f"UnidentifiedImageError: {str(e)} for page {i+1}")
        except OSError as e:
            logging.error(f"OSError: {str(e)} for page {i+1}")
        except Exception as e:
            logging.error(f"Exception: {str(e)} for page {i+1}")
    return image_files

def extract_tables_from_pdf(pdf_path, pdf_name):
    table_dfs = []
    table_counter = 0
    doc = PdfReader(pdf_path)
    num_pages = len(doc.pages)
    for page_num in range(num_pages):
        try:
            tables = camelot.read_pdf(pdf_path, pages=str(page_num + 1), flavor='stream')
            for table_index, table in enumerate(tables):
                if len(table.df.columns) > 1 and len(table.df) > 1:
                    table_counter += 1
                    header = " ".join(table.df.iloc[0].values)
                    sanitized_content = clean_tag_value(header.strip())
                    table_dfs.append((table.df, sanitized_content, page_num + 1, table_counter))
        except IndexError as e:
            logging.error(f"IndexError: {str(e)} while processing page {page_num+1} of {pdf_path}")
        except Exception as e:
            logging.error(f"Exception: {str(e)} while processing page {page_num+1} of {pdf_path}")
```

```
    return table_dfs

def save_tables_to_csv(tables, output_folder, pdf_name):
    csv_files = []
    for table, content, page_num, table_num in tables:
        csv_filename = f"{output_folder}/{pdf_name}_page_{page_num}_table_{table_num}.csv"
        table.to_csv(csv_filename, index=False)
        csv_files.append((csv_filename, content if content else "No table content"))
    return csv_files

def clean_tag_value(value):
    value = re.sub(r'\s+', ' ', value)
    value = re.sub(r'[^a-zA-Z0-9\s_\.\:/=\+\-@]', '', value)
    return value[:256]

def upload_files_to_s3_with_modified_tags(bucket_name, files, s3_path, tags, content_type):
    for file_path, content in files:
        s3_key = f"{s3_path}/{os.path.basename(file_path)}"

        # Update specific tags
        tags['Source'] = 'PDF_extract_image_tables.py'
        tag_set = [{'Key': key, 'Value': clean_tag_value(value)} for key, value in tags.items()]

        # Upload the file with the correct Content-Type
        s3_client.upload_file(
            Filename=file_path,
            Bucket=bucket_name,
            Key=s3_key,
            ExtraArgs={
                'ContentType': content_type,
                'Metadata': {}
            }
        )

        # Set the tags on the uploaded file
        s3_client.put_object_tagging(
            Bucket=bucket_name,
            Key=s3_key,
            Tagging={'TagSet': tag_set}
        )

def lambda_handler(event, context):
    file_name = "unknown"  # Initialize file_name to a default value
    try:
```

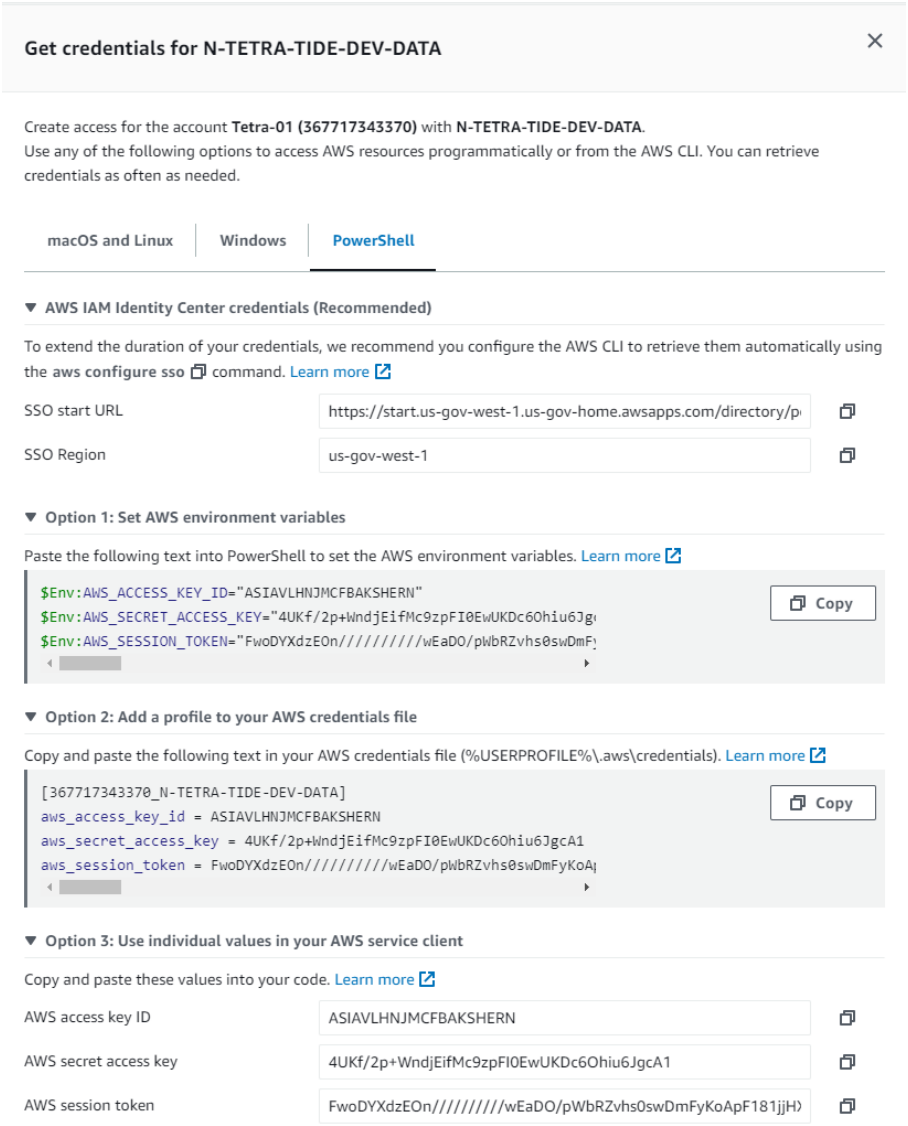
```
    logging.info(f"Event: {event}")

    # Check if the event is in the expected format
    if 'Records' in event and event['Records']:
        bucket_name = event['Records'][0]['s3']['bucket']['name']
        key = event['Records'][0]['s3']['object']['key']
    elif 'bucket_name' in event and 'object_key' in event:
        bucket_name = event['bucket_name']
        key = event['object_key']
    else:
        raise KeyError("Event does not contain 'bucket_name' or 'object_key' key")

    file_name = os.path.basename(key)
    output_bucket_name = 'processed-data-bucket'
    images_subfolder = 'PDF_extracted_images'
    data_subfolder = 'PDF_extracted_data'

    pdf_name = os.path.splitext(file_name)[0]
    images_output_folder = f"/tmp/{pdf_name}_images"
    data_output_folder = f"/tmp/{pdf_name}_data"
    os.makedirs(images_output_folder, exist_ok=True)
    os.makedirs(data_output_folder, exist_ok=True)

    logging.info(f"Bucket: {bucket_name}, Key: {key}")

    local_pdf_path, tags, metadata = download_pdf_from_s3_and_get_tags(bucket_name, key)

    # Extract text from PDF using pdfminer.six
    text = extract_text(local_pdf_path)

    extracted_images = extract_images_from_pdf(local_pdf_path, images_output_folder, pdf_name)
    extracted_tables = extract_tables_from_pdf(local_pdf_path, pdf_name)
    csv_files = save_tables_to_csv(extracted_tables, data_output_folder, pdf_name)

    images_s3_path = f"{images_subfolder}/{pdf_name}"
    data_s3_path = f"{data_subfolder}/{pdf_name}"

    upload_files_to_s3_with_modified_tags(output_bucket_name, extracted_images, images_s3_path,
tags, 'image/jpeg')
    upload_files_to_s3_with_modified_tags(output_bucket_name, csv_files, data_s3_path, tags,
'text/plain')

    logging.info(f"Processing and uploading completed successfully for {file_name}")
    log_to_cloudwatch(
```

```
            namespace='MyLambdaFunction',
            metric_name='SuccessfulProcessing',
            value=1,
            dimensions=[
                {'Name': 'FunctionName', 'Value': 'ExtractImagesAndTables'},
                {'Name': 'FileName', 'Value': file_name}
            ]
        )
        return {
            'statusCode': 200,
            'body': json.dumps('Processing and uploading completed successfully')
        }

    except KeyError as e:
        logging.error(f"KeyError: {str(e)}")
        log_to_cloudwatch(
            namespace='MyLambdaFunction',
            metric_name='KeyError',
            value=1,
            dimensions=[
                {'Name': 'FunctionName', 'Value': 'ExtractImagesAndTables'}
            ]
        )
        return {
            'statusCode': 400,
            'body': json.dumps(f"KeyError: {str(e)}")
        }
    except Exception as e:
        logging.error(f"Error processing {file_name}: {str(e)}")
        log_to_cloudwatch(
            namespace='MyLambdaFunction',
            metric_name='ProcessingError',
            value=1,
            dimensions=[
                {'Name': 'FunctionName', 'Value': 'ExtractImagesAndTables'}
            ]
        )
        return {
            'statusCode': 500,
            'body': json.dumps(f"Error processing {file_name}: {str(e)}")
        }
```

Build and Push Commands using Windows Powershell to Authenticate Docker and PowerShell with AWS Account:

Get the Access key information from your AWS Account home page:

Choose ‘Access keys’, which will display this dialogue:



The access key information will change daily, therefore you must reauthenticate daily.

#PowerShell Commands to Authenticate AWS and ECR

```
$Env:AWS_ACCESS_KEY_ID="Access Key from Account Page"
$Env:AWS_SECRET_ACCESS_KEY="Secreet Access Key from Account Page"
$Env:AWS_SESSION_TOKEN="Session Token from Access Page - This  will be long"
$env:AWS_REGION="us-gov-east-1"

aws configure

AWS Access Key ID [****************AOFT]: Provide the access key again
AWS Secret Access Key [****************Py1f]: Provide the secret access key again
Default region name [us-gov-east-1]: us-gov-east-1
Default output format [json]: json

Get-STSCallerIdentity

(Get-ECRLoginCommand -Region us-gov-east-1).Password | docker login --username AWS --password-
stdin 367717343370.dkr.ecr.us-gov-east-1.amazonaws.com

Output should read 'Login Successful"

Push Commands for building and pushing function Docker images to ECR:

#Build and Push Commands General format
cd C:\<local directory to files>
docker build -t lambda_function .
$TAG = Get-Date -Format "yyyyMMddHHmmss"
docker tag lambda_function:latest 367717343370.dkr.ecr.us-gov-east-1.amazonaws.com/ai-
lambda:lambda_function-$TAG
docker push 367717343370.dkr.ecr.us-gov-east-1.amazonaws.com/ai-lambda:lambda_function-$TAG

# Build and push Docker image for PDF Image/Table Extraction
cd C:\<local directory to files>
docker build -t lambda_function .
$TAG = Get-Date -Format "yyyyMMddHHmmss"
docker tag lambda_function:latest 367717343370.dkr.ecr.us-gov-east-1.amazonaws.com/ai-
lambda:PDF-image-table-extract-$TAG
docker push 367717343370.dkr.ecr.us-gov-east-1.amazonaws.com/ai-lambda:PDF-image-table-extract-
$TAG
```

Update Image from ECR: Once the image is successfully built and pushed to the ECR repo 'ai-lambda', the Lambda function must be updated so that it accesses the correct image as shown above.

BUILD AND PUSH COMMANDS USING WINDOWS POWERSHELL

Get the Access key information from your AWS Account home page.

PowerShell Commands to Authenticate AWS and ECR:

powershell
Copy code

```
$Env:AWS_ACCESS_KEY_ID="Access Key from Account Page"
$Env:AWS_SECRET_ACCESS_KEY="Secret Access Key from Account Page"
$Env:AWS_SESSION_TOKEN="Session Token from Access Page - This will be long"
$env:AWS_REGION="us-gov-east-1"

aws configure

AWS Access Key ID [****************AOFT]: Provide the access key again
AWS Secret Access Key [****************Py1f]: Provide the secret access key again
Default region name [us-gov-east-1]: us-gov-east-1
Default output format [json]: json

Get-STSCallerIdentity

(Get-ECRLoginCommand -Region us-gov-east-1).Password | docker login --username AWS --password-
stdin 367717343370.dkr.ecr.us-gov-east-1.amazonaws.com
Output should read 'Login Successful'.
```

Push Commands for building and pushing function Docker images to ECR:

powershell
Copy code

```
cd C:\<local directory to files>
docker build -t lambda_function .
$TAG = Get-Date -Format "yyyyMMddHHmmss"
docker tag lambda_function:latest 367717343370.dkr.ecr.us-gov-east-1.amazonaws.com/ai-
lambda:lambda_function-$TAG
docker push 367717343370.dkr.ecr.us-gov-east-1.amazonaws.com/ai-lambda:lambda_function-$TAG
```

SETUP AWS STEP FUNCTIONS STATE MACHINE: 'PDF_SUMMARY_STATE_MACHINE'

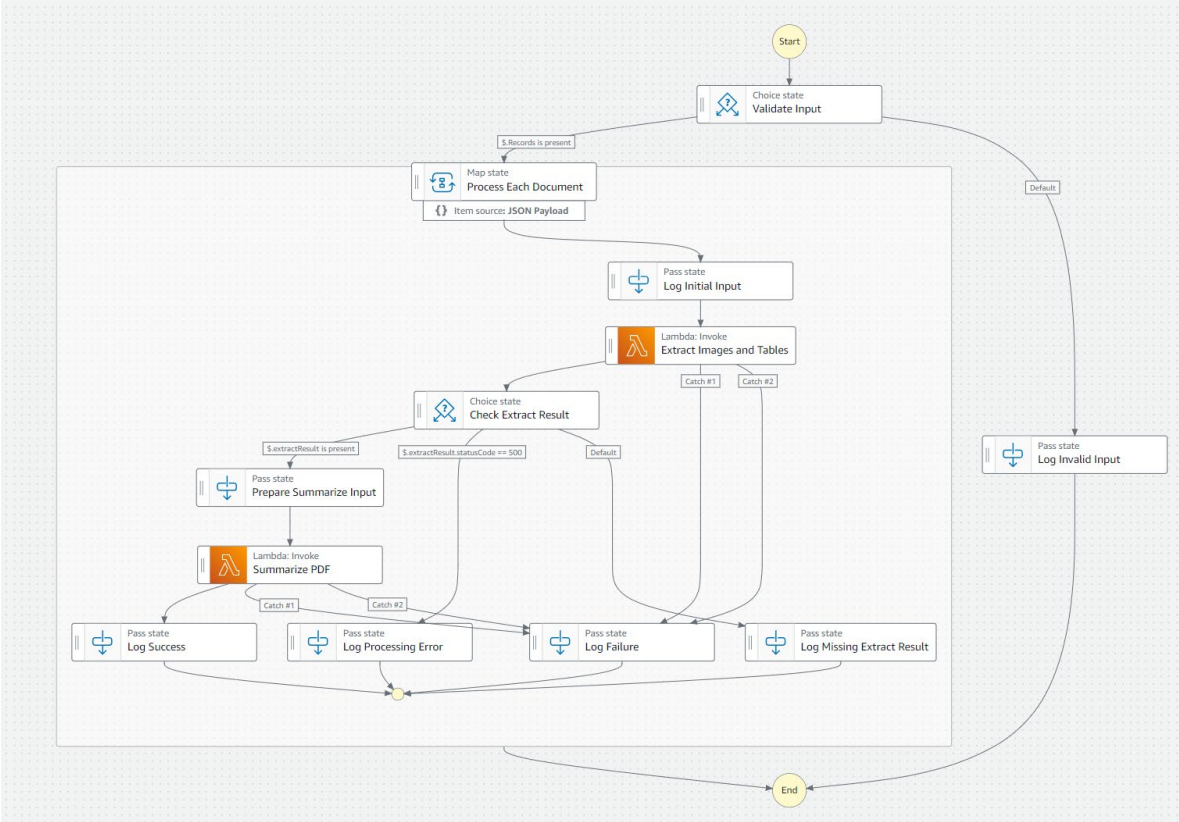
The AWS Step Functions state machine orchestrates a comprehensive PDF processing workflow with built-in retry logic and error handling mechanisms. This state machine processes each document by first logging the initial input details such as the bucket name and object key. It then extracts images and tables from the PDF using the extract-pdf-image-tables Lambda function. The state machine checks the results of this extraction and if successful, prepares the input for summarizing the PDF content. The summarization is handled by the process-pdf-documents-ecr Lambda function.

To ensure reliability, the state machine includes retry logic, attempting each task up to three times with exponential backoff if an error occurs. Specific errors such as missing keys are caught and logged with

the workflow proceeding to failure states if necessary. Successes are also logged for tracking purposes. The state machine requires appropriate IAM permissions to invoke the specified Lambda functions, ensuring secure and authorized execution. This setup provides a robust and automated solution for processing PDF documents, handling various potential issues gracefully while maintaining a reliable workflow.

Placeholders for Images:

1. **Diagram showing the architecture of the AWS Step Functions state machine setup.**



2. **State Machine code defition (JSON):**

```
{
  "Comment": "State machine to orchestrate PDF processing workflow with retry logic and handling within Lambda functions",
  "StartAt": "Validate Input",
  "States": {
    "Validate Input": {
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$.Records",
          "IsPresent": true,
          "Next": "Process Each Document"
        }
      ],
      "Default": "Log Invalid Input"
```

```
    },
    "Process Each Document": {
      "Type": "Map",
      "ItemsPath": "$.Records",
      "Iterator": {
        "StartAt": "Log Initial Input",
        "States": {
          "Log Initial Input": {
            "Type": "Pass",
            "Parameters": {
              "bucket_name.$": "$.s3.bucket.name",
              "object_key.$": "$.s3.object.key"
            },
            "Next": "Extract Images and Tables"
          },
          "Extract Images and Tables": {
            "Type": "Task",
            "Resource": "arn:aws-us-gov:lambda:us-gov-east-
1:367717343370:function:extract-pdf-image-tables",
            "Next": "Check Extract Result",
            "InputPath": "$",
            "ResultPath": "$.extractResult",
            "Retry": [
              {
                "ErrorEquals": [
                  "States.TaskFailed",
                  "NoSuchKey"
                ],
                "IntervalSeconds": 5,
                "MaxAttempts": 3,
                "BackoffRate": 2
              },
              {
                "ErrorEquals": [
                  "States.ALL"
                ],
                "IntervalSeconds": 5,
                "MaxAttempts": 3,
                "BackoffRate": 2
              }
            ],
            "Catch": [
              {
                "ErrorEquals": [
                  "NoSuchKey"
                ],
                "ResultPath": "$.errorInfo",
                "Next": "Log Failure"
```

```
            },
            {
              "ErrorEquals": [
                "States.ALL"
              ],
              "ResultPath": "$.errorInfo",
              "Next": "Log Failure"
            }
          ]
        },
        "Check Extract Result": {
          "Type": "Choice",
          "Choices": [
            {
              "Variable": "$.extractResult",
              "IsPresent": true,
              "Next": "Prepare Summarize Input"
            },
            {
              "Variable": "$.extractResult.statusCode",
              "NumericEquals": 500,
              "Next": "Log Processing Error"
            }
          ],
          "Default": "Log Missing Extract Result"
        },
        "Prepare Summarize Input": {
          "Type": "Pass",
          "Parameters": {
            "bucket_name.$": "$.bucket_name",
            "object_key.$": "$.object_key",
            "extractResult.$": "$.extractResult"
          },
          "ResultPath": "$.summarizeInput",
          "Next": "Summarize PDF"
        },
        "Summarize PDF": {
          "Type": "Task",
          "Resource": "arn:aws-us-gov:lambda:us-gov-east-
1:367717343370:function:process-pdf-documents-ecr",
          "InputPath": "$.summarizeInput",
          "ResultPath": "$.summarizeResult",
          "Retry": [
            {
              "ErrorEquals": [
                "States.TaskFailed",
                "NoSuchKey"
              ],
```

```
        "IntervalSeconds": 5,
        "MaxAttempts": 3,
        "BackoffRate": 2
      },
      {
        "ErrorEquals": [
          "States.ALL"
        ],
        "IntervalSeconds": 5,
        "MaxAttempts": 3,
        "BackoffRate": 2
      }
    ],
    "Catch": [
      {
        "ErrorEquals": [
          "NoSuchKey"
        ],
        "ResultPath": "$.errorInfo",
        "Next": "Log Failure"
      },
      {
        "ErrorEquals": [
          "States.ALL"
        ],
        "ResultPath": "$.errorInfo",
        "Next": "Log Failure"
      }
    ],
    "Next": "Log Success"
  },
  "Log Success": {
    "Type": "Pass",
    "Result": {
      "Status": "Success",
      "Message": "PDF processing succeeded"
    },
    "End": true
  },
  "Log Missing Extract Result": {
    "Type": "Pass",
    "Result": {
      "Status": "Failure",
      "Message": "Extract result is missing"
    },
    "End": true
  },
  "Log Processing Error": {
```

```
            "Type": "Pass",
            "Result": {
              "Status": "Failure",
              "Message": "Error processing PDF",
              "ErrorDetails.$": "$.extractResult.body"
            },
            "End": true
          },
          "Log Failure": {
            "Type": "Pass",
            "Result": {
              "Status": "Failure",
              "Message": "PDF processing failed",
              "ErrorDetails.$": "$.errorInfo"
            },
            "End": true
          }
        }
      },
      "End": true
    },
    "Log Invalid Input": {
      "Type": "Pass",
      "Result": {
        "Status": "Failure",
        "Message": "Invalid input data"
      },
      "End": true
    }
  }
}
```

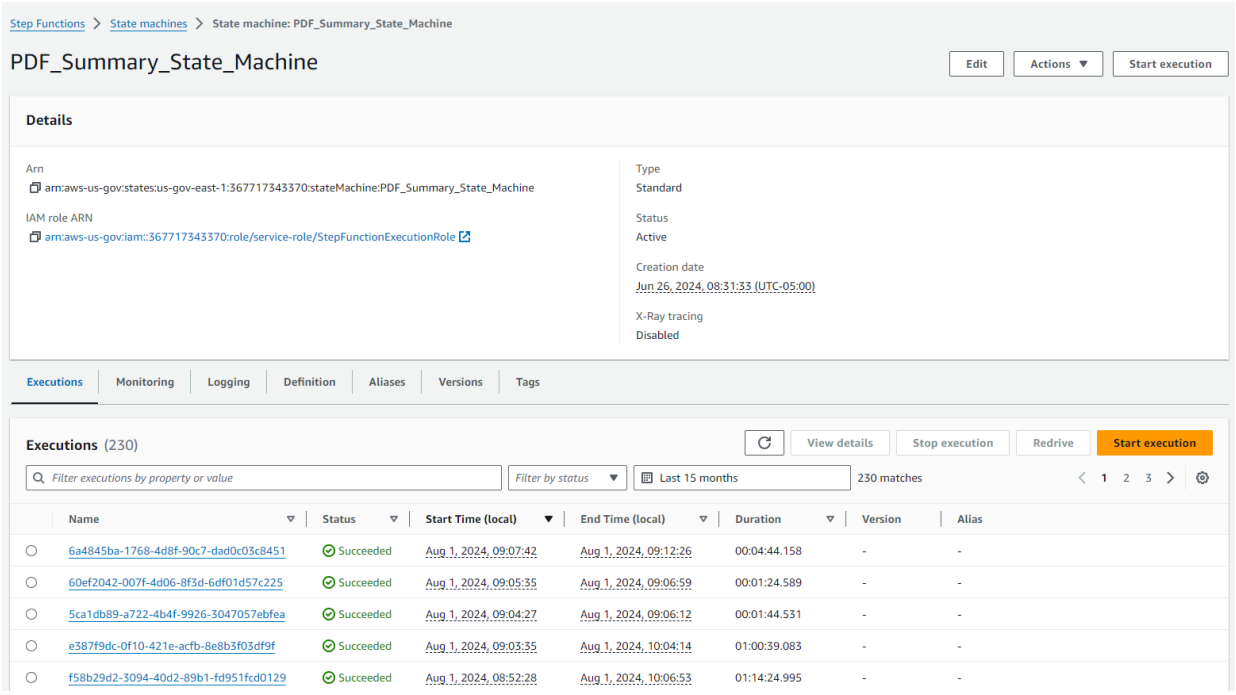
CLOUDWATCH AND STATE MACHINE LOGS

All activity that flows through the state machine, including all PDF processing, is logged to include errors and non-completed processes. These logs can be accessed either in the State Machine itself or in CloudWatch.

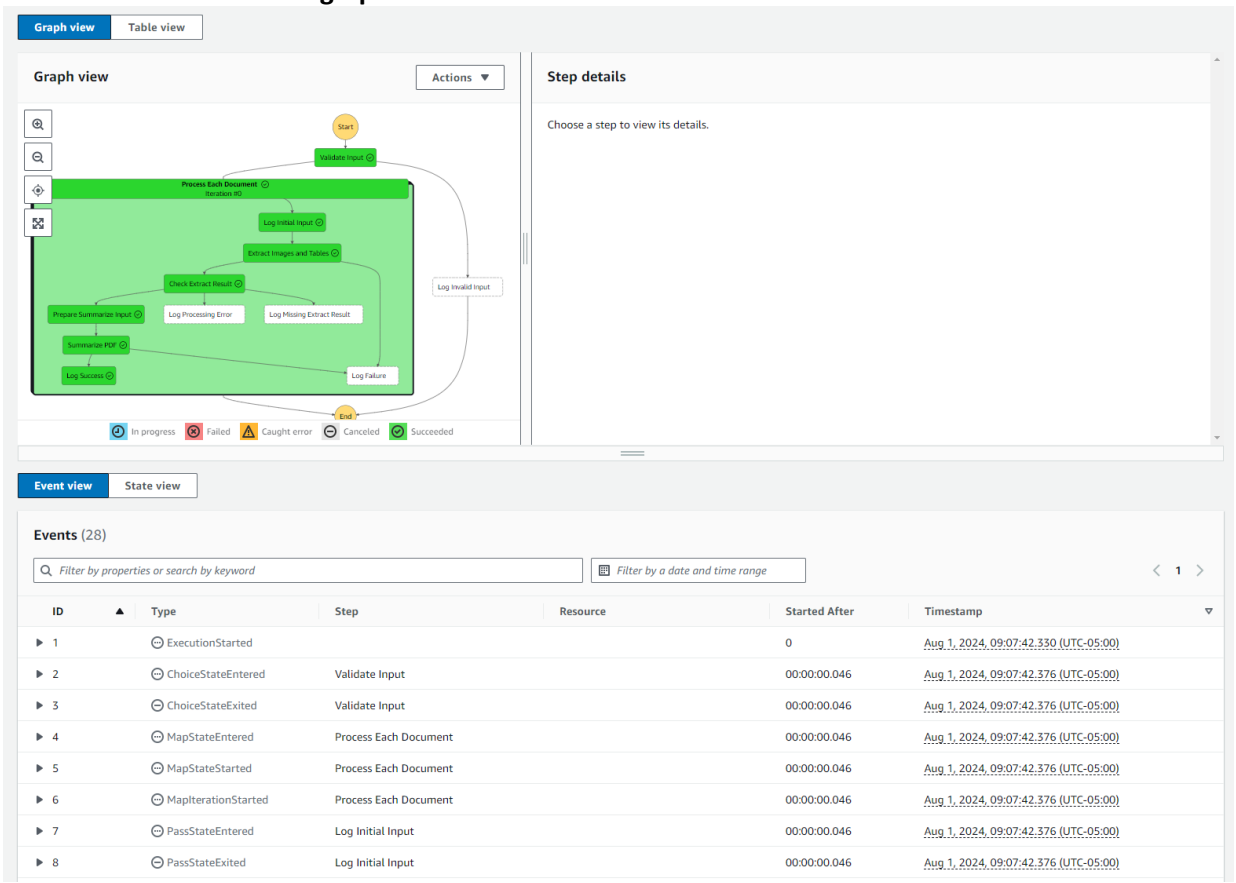
From within the State Machine: Select the executions links for the process in question. This will show a graphical and step-by-step view of all actions that occurred during that particular run. Events can be further expanded to reveal any errors and successful activities.

Placeholders for Images:

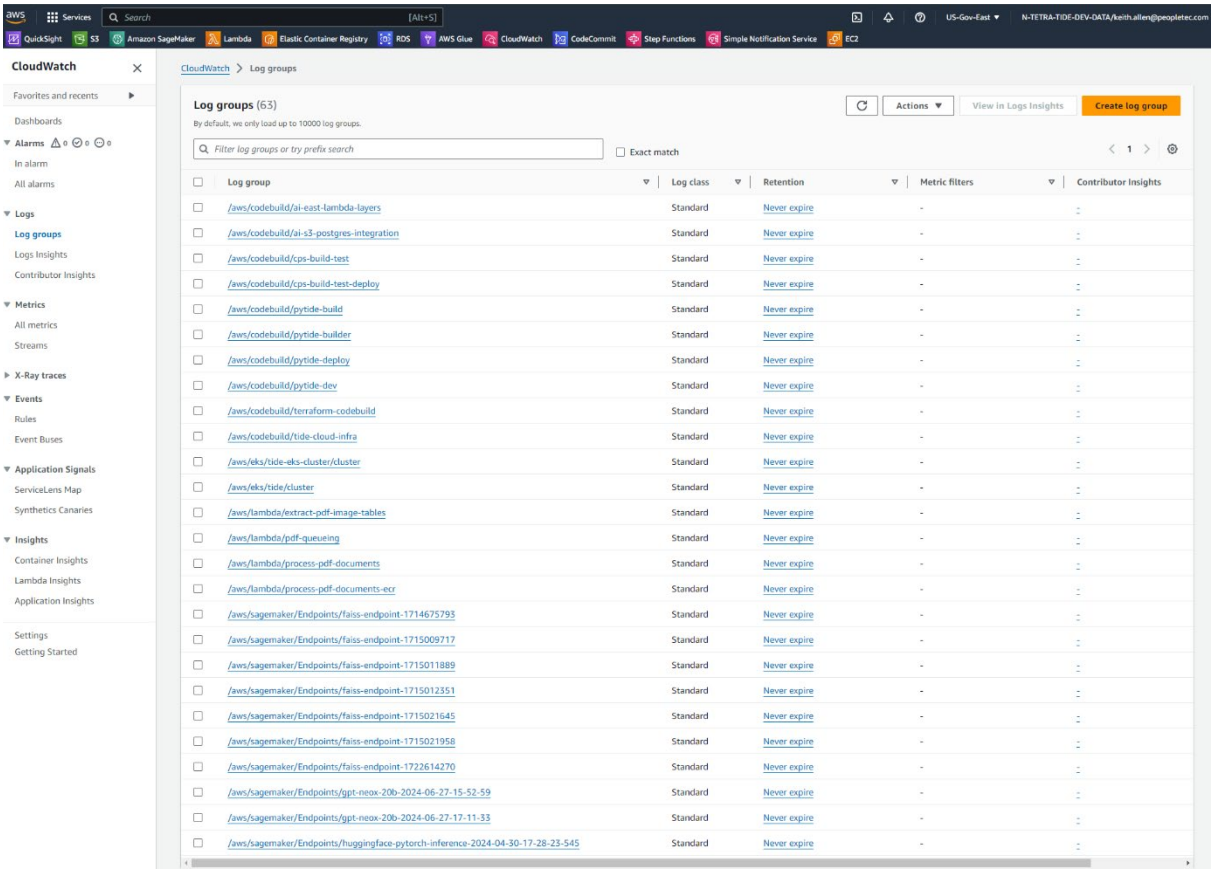
1. **Screenshot showing the State Machine executions in the AWS Management Console.**



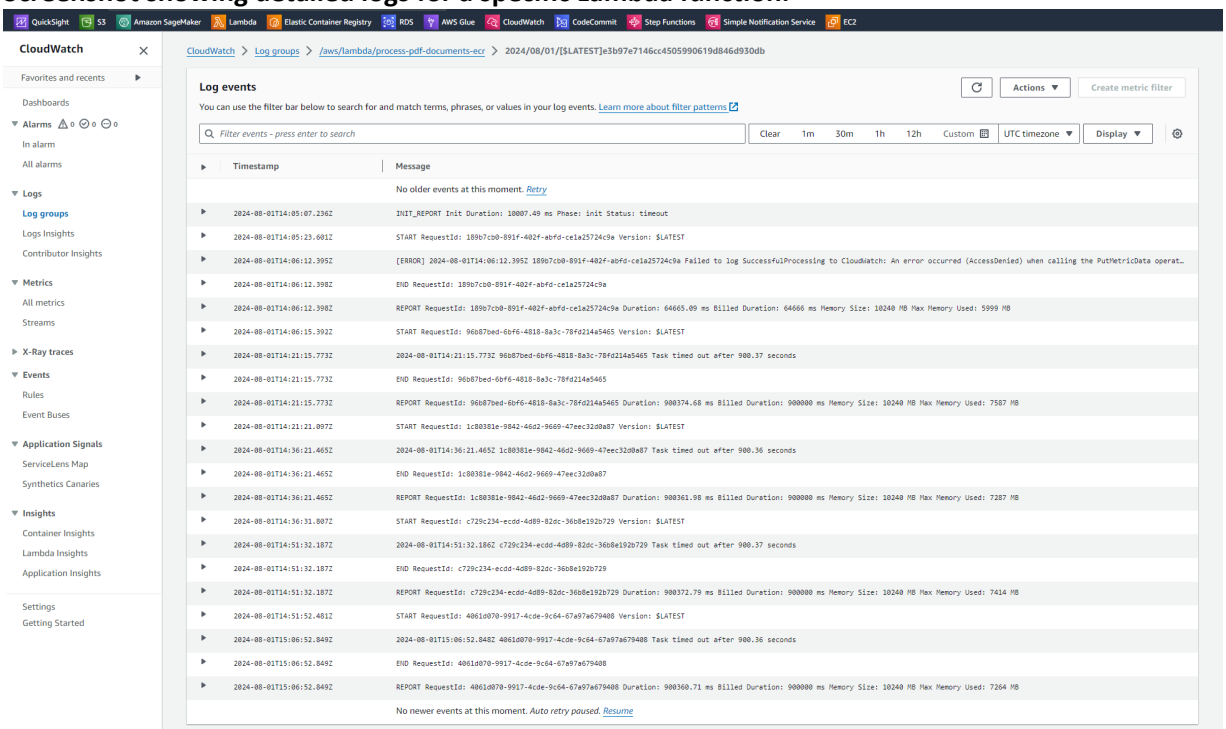
2. **Screenshot of the graphical view of a State Machine execution.**



More detailed logs are available in CloudWatch log groups by function: Selecting the individual log group by function allows detailed logs to be viewed and downloaded.

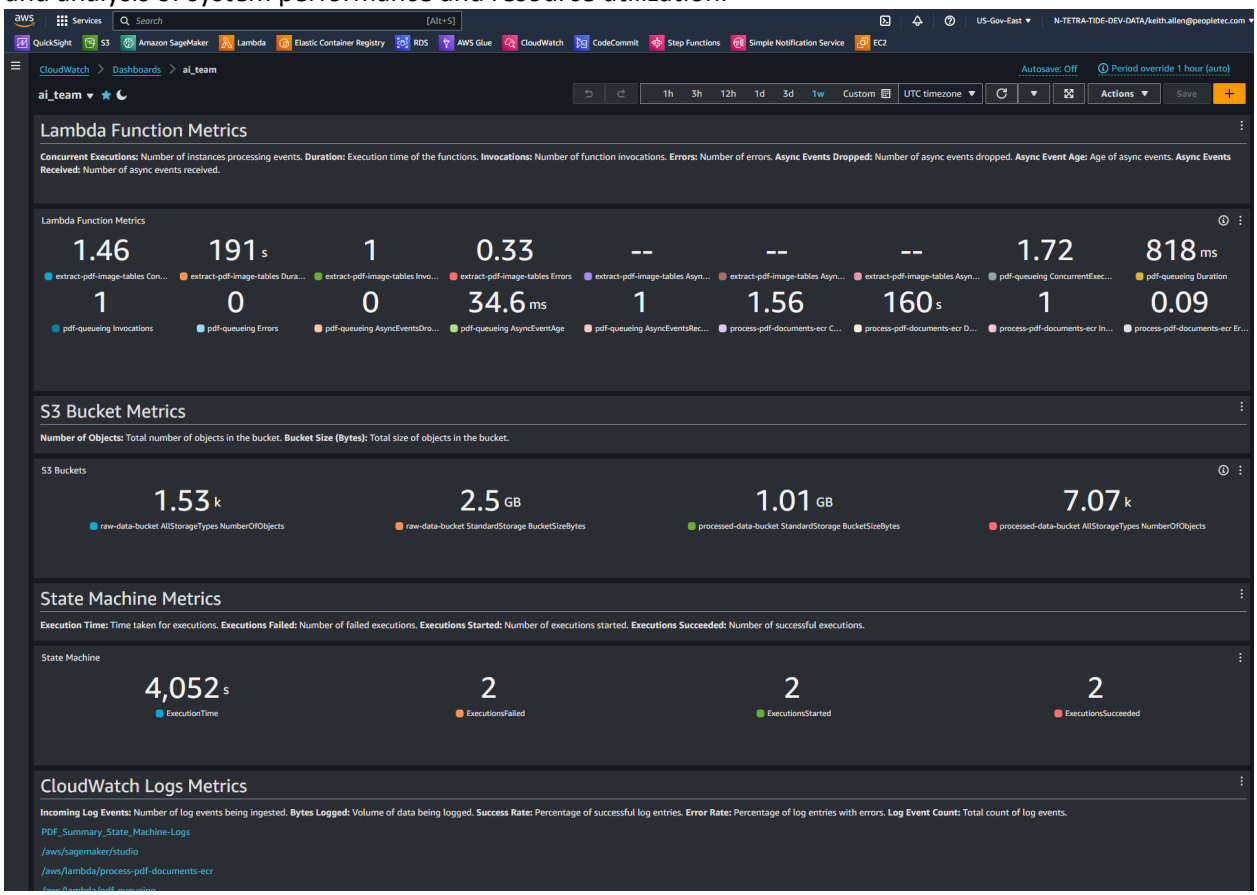


Screenshot showing detailed logs for a specific Lambda function.



CLOUDWATCH DASHBOARD

The TIDE AI team monitors all processes in and out of the system via a CloudWatch Dashboard called 'ai-team'. The provided dashboard is a comprehensive monitoring tool for AWS Lambda functions, S3 buckets, State Machines, and CloudWatch logs, offering real-time insights into system performance and resource usage.
Key metrics tracked for Lambda functions include concurrent executions, execution duration, invocations, errors, and async event statistics for functions like extract-pdf-image-tables, pdf-queueing, and process-pdf-documents-ecr.
Additionally, the dashboard monitors S3 buckets (raw-data-bucket and processed-data-bucket) for the number of objects and total bucket size. State Machine metrics include execution time, failed executions, started executions, and successful executions for the PDF_Summary_State_Machine. CloudWatch logs metrics track incoming log events, data volume, success rate, error rate, and total log event count with links to specific log groups for detailed inspection. This setup ensures efficient tracking and analysis of system performance and resource utilization.



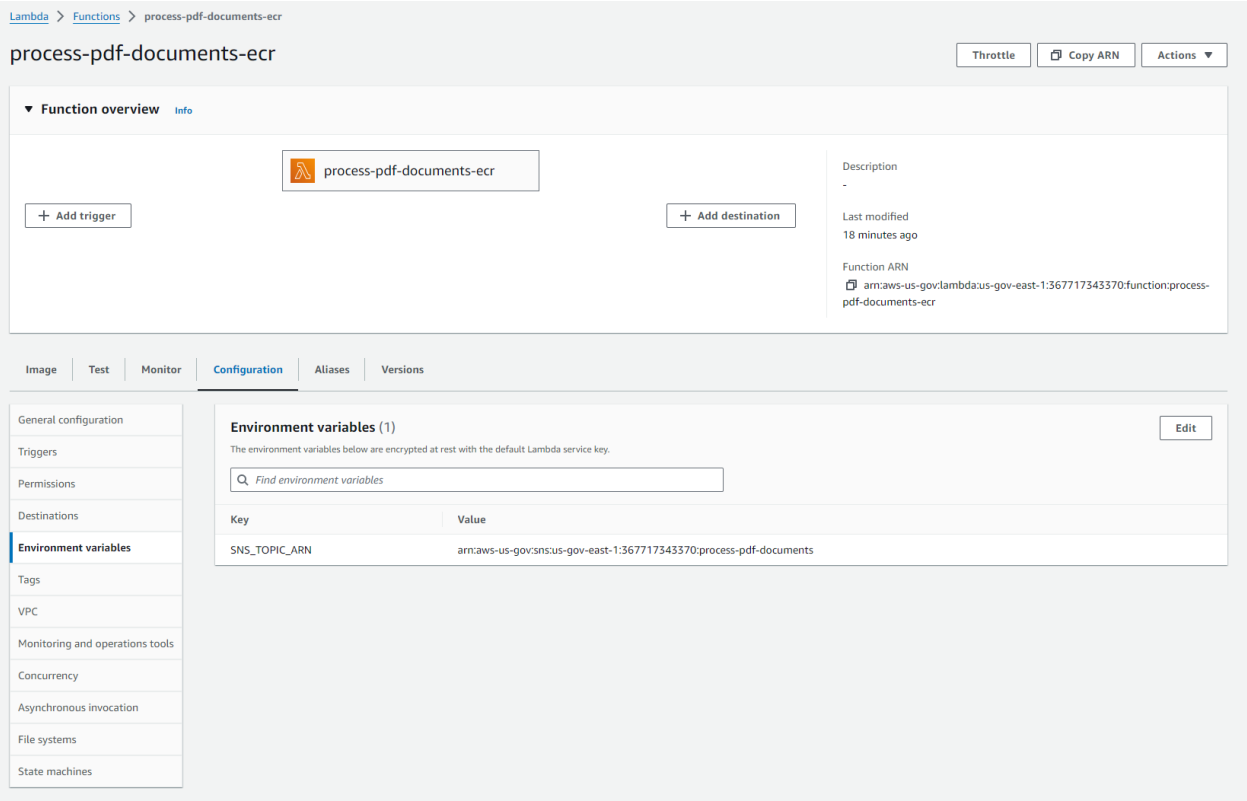
SNS TOPIC: PDF READY MESSAGE FOR UI

Embedded in the process-pdf-documents-ecr Lambda function is code that sends a notification to the SNS Topic process-pdf-documents. A unique message is sent to SNS every time a PDF summary is generated by the process and includes the unique URL for the processed PDF Summary in S3. This notifies the UI every time a PDF is processed and enables the UI to call the PDF into the UI.

Notifications will be displayed in the SNS topic ‘subscriptions’ page in the following format:

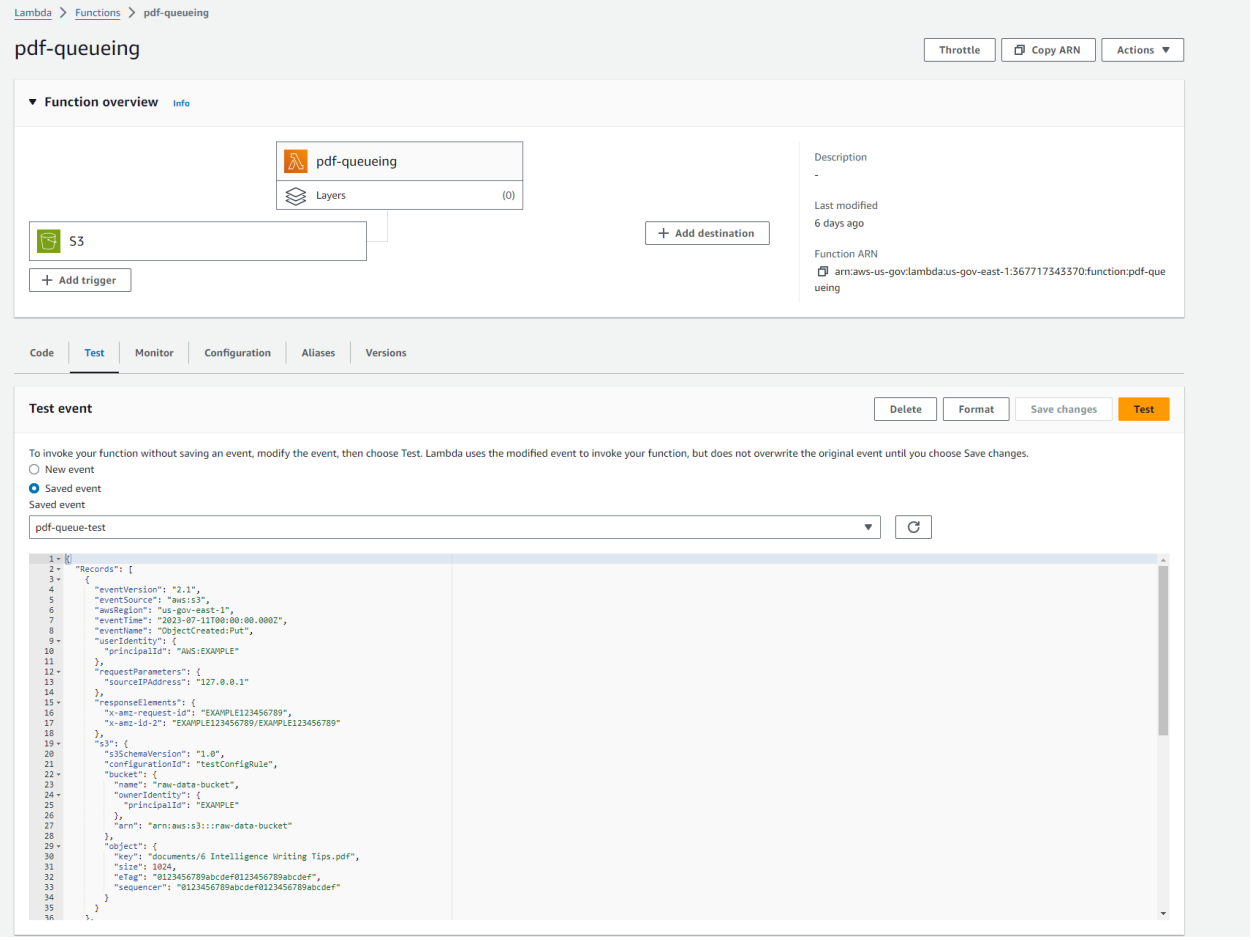
{
"default": "{\"bucket_name\": \"processed-data-bucket\", \"file_key\": \"PDF_summaries/processed_file.pdf\", \"s3_url\": \"https://processed-data-bucket.s3.amazonaws.com/PDF_summaries/processed_file.pdf\"}"
}

Ensure that environment variables in the process-pdf-documents-ecr Lambda function are updated with the following information to enable this process.



TESTING THE SETUP

Once the entire system is configured, it is important to test the entire workflow. There are two ways to test the system: either by simply uploading a set of documents to the raw-data-bucket/documents folder or by initiating a test script. To initiate the test script, a document must already exist in the raw-data-bucket/documents that will be processed by the system.
Go to the 'test' tab under the pdf-queueing function as shown:



This test function simulates AWS S3 events triggering a Lambda function in the us-gov-east-1 region. It includes two records indicating objects being created (via Put operation) in the raw-data-bucket. Each record provides details about the event such as the event version (2.1), event source (aws:s3), event time, event name (ObjectCreated:Put), and the AWS region. The user identity, request parameters (source IP address), and response elements (request ID and ID 2) are also included. The S3 object information such as schema version (1.0), configuration ID, bucket name, bucket ARN, object key, size, eTag, and sequencer is detailed for both objects: "6 Intelligence Writing Tips.pdf" (size 1024 bytes) and "Comparison of Cloud On-Premises and Hybrid Solutions for AI Development_Allen_22May24.pdf" (size 2048 bytes).

Note: The specific document or documents being tested can easily be changed within Lambda in the above console screen.

Once a test occurs, be sure to check the output folders in the processed-data-bucket to ensure the correct outputs are rendered, as well as the CloudWatch logs to ensure no errors have occurred.

GENERAL TIPS, ADVICE, AND RULES OF THUMB FOR PDF PROCESSING WORKFLOW SETUP

Initial Setup:

1. **Access and Permissions:**
    - Ensure you have access to AWS VPN, AWS Account, and administrative privileges on your local machine.
    - Verify access to necessary AWS services including S3, ECR, Step Functions, Lambda, and CloudWatch.
2. **Authentication:**
    - Use PowerShell to authenticate with AWS by setting environment variables for AWS access keys and region.
    - Authenticate Docker with AWS ECR for pushing Docker images.
3. **Building and Pushing Docker Images:**
    - Use Docker commands to build and tag Lambda function images.
    - Push the Docker images to AWS ECR using appropriate tags for different Lambda functions.

Lambda Functions:

1. **Lambda Function to Initiate Queueing Process:**
    - This function handles S3 events, checks object existence, and starts the Step Functions state machine.
    - Ensure proper error handling and logging for troubleshooting.
    - Dependencies include AWS Lambda environment setup, Boto3 library, and valid Step Functions ARN.
2. **Lambda Function for PDF Summary:**
    - Processes PDF files to extract text, keywords, and summaries then uploads the modified PDF back to S3.
    - Dependencies include libraries like Boto3, PyMuPDF, NLTK, transformers, and reportlab.
    - Ensure environment variables for Hugging Face and NLTK data paths are set.
3. **Lambda Function for PDF Image and Table Extraction:**
    - Extracts images and tables from PDFs, performs OCR, and saves the extracted data as CSV files.
    - Dependencies include libraries like Boto3, Camelot, Pytesseract, pdfminer.six, PyPDF2, pdf2image, and Pillow.
    - Use retry logic and proper error handling to ensure reliability.

AWS Step Functions State Machine:

1. **Workflow Orchestration:**
    - The state machine logs initial input, extracts images and tables, and summarizes PDF content.
    - Includes retry logic with exponential backoff to handle errors gracefully.
    - Logs success and failure states for tracking purposes.
2. **Dependencies and Permissions:**
    - Ensure the state machine has IAM roles with permissions to invoke the specified Lambda functions.
    - Validate that all ARNs and resource names are correct.

General Advice:

1. **Testing and Debugging:**
    - Thoroughly test each Lambda function independently before integrating them into the state machine. Test scripts are available for each Lambda function.
    - Use CloudWatch logs for debugging and monitoring the workflow.
2. **Security:**
    - Follow AWS security best practices including least privilege access for IAM roles and encrypting sensitive data.
    - Regularly rotate access keys and update environment variables accordingly.
3. **Performance Optimization:**
    - Use efficient code practices to optimize the performance of Lambda functions.
    - Monitor execution times and optimize memory and timeout settings as needed.
4. **Documentation:**
    - Maintain clear documentation for all setup steps, commands, and configurations.
    - Document any customizations or changes made to the workflow for future reference.

GENERAL BEST PRACTICES FOR TIDE AI DEPLOYMENT SETUP

Ensure you have the following prerequisites and configurations in place before starting the deployment process:

Initial Setup:

1. **Access and Permissions:**
    - **AWS VPN:** Ensure you have access to the AWS VPN for secure connectivity.
    - **AWS Account:** You need an AWS account with the necessary permissions.
    - **Local Machine Setup:**
        - **PowerShell:** Install PowerShell with administrative privileges.
        - **Docker Desktop:** Install Docker Desktop with administrative privileges.
2. **AWS Service Access:**
    - **AWS S3 Access:** Ensure you have the necessary permissions to create and manage S3 buckets.
    - **AWS ECR Access:** Ensure you have permissions to create and manage repositories in Amazon ECR.
    - **AWS Step Functions and Lambda Access:** Ensure you have permissions to create and manage Step Functions and Lambda functions.
    - **CloudWatch Access:** Ensure you have access to CloudWatch for monitoring and error logging.

Commands to Authenticate Docker and PowerShell with AWS Account:

1. **Retrieve Access Keys:**
    - Obtain your AWS access key, secret access key, and session token from your AWS account home page.
2. **PowerShell Commands to Authenticate AWS and ECR:**

```
powershell
Copy code
$Env:AWS_ACCESS_KEY_ID="Access Key from Account Page"
$Env:AWS_SECRET_ACCESS_KEY="Secret Access Key from Account Page"
$Env:AWS_SESSION_TOKEN="Session Token from Access Page - This will be long"
$env:AWS_REGION="us-gov-east-1"
```

aws configure

AWS Access Key ID [**********AOFT]: Provide the access key again
AWS Secret Access Key [**********Py1f]: Provide the secret access key again
Default region name [us-gov-east-1]: us-gov-east-1
Default output format [json]: json

3. **Authenticate Docker with AWS ECR:**
   powershell
   Copy code
   (Get-ECRLoginCommand -Region us-gov-east-1).Password | docker login --username AWS --password-stdin 367717343370.dkr.ecr.us-gov-east-1.amazonaws.com

Building and Pushing Docker Images:

1. **Build Docker Image:**

   cd C:\<local directory to files>
   docker build -t lambda_function .
   $TAG = Get-Date -Format "yyyyMMddHHmmss"
   docker tag lambda_function:latest 367717343370.dkr.ecr.us-gov-east-1.amazonaws.com/ai-lambda:lambda_function-$TAG
2. **Push Docker Image to ECR:**
   docker push 367717343370.dkr.ecr.us-gov-east-1.amazonaws.com/ai-lambda:lambda_function-$TAG

General Advice:

1. **Testing and Debugging:**
   - Thoroughly test each Lambda function independently before integrating them into the state machine. Test scripts are available for each Lambda function.
   - Use CloudWatch logs for debugging and monitoring the workflow.
2. **Security:**
   - Follow AWS security best practices including least privilege access for IAM roles and encrypting sensitive data.
   - Regularly rotate access keys and update environment variables accordingly.
3. **Performance Optimization:**
   - Use efficient code practices to optimize the performance of Lambda functions.
   - Monitor execution times and optimize memory and timeout settings as needed.
4. **Documentation:**
   - Maintain clear documentation for all setup steps, commands, and configurations.
   - Document any customizations or changes made to the workflow for future reference.