Cloud project

**Project Document: Serverless Event-Driven Image Processing Application**

**1. Project Title**

**Serverless Event-Driven Image Processing Application**

**2. Project Overview**

This project is designed to create a **serverless web application** for uploading, storing, and processing images using AWS cloud services. The system allows users to upload images via a frontend interface, stores them in **S3**, triggers **AWS Lambda** functions for processing, and logs metadata in **DynamoDB**. The architecture ensures **scalability, cost-efficiency, and minimal server management**, making it suitable for dynamic image-based applications.

**3. Objectives**

* Provide a **web interface** for image uploads.
* Enable **serverless storage** of images in Amazon S3.
* Implement **automatic image processing** (e.g., resizing, format conversion) using AWS Lambda.
* Track and store **image metadata** in DynamoDB.
* Deploy **API endpoints** using API Gateway for frontend-backend communication.
* Ensure **security, scalability, and reliability** without managing servers.

**4. Tools & Technologies**

* **Frontend:** HTML, CSS, JavaScript
* **Backend/Serverless:** AWS Lambda, API Gateway
* **Storage:** Amazon S3 (Raw and Processed Buckets)
* **Database:** Amazon DynamoDB
* **Other Tools:** AWS IAM (Roles & Policies), Postman (for API testing), VS Code (IDE)

**5. Architecture**

**Serverless Event-Driven Architecture Diagram:**

[Frontend (HTML/JS)]

|

v

[API Gateway Endpoint] <---> [AWS Lambda Function]

|

v

[Amazon S3 - Raw Bucket]

|

v (Event Trigger)

[Lambda Function for Processing]

|

v

[Amazon S3 - Processed Bucket]

|

v

[Amazon DynamoDB (Metadata Storage)]

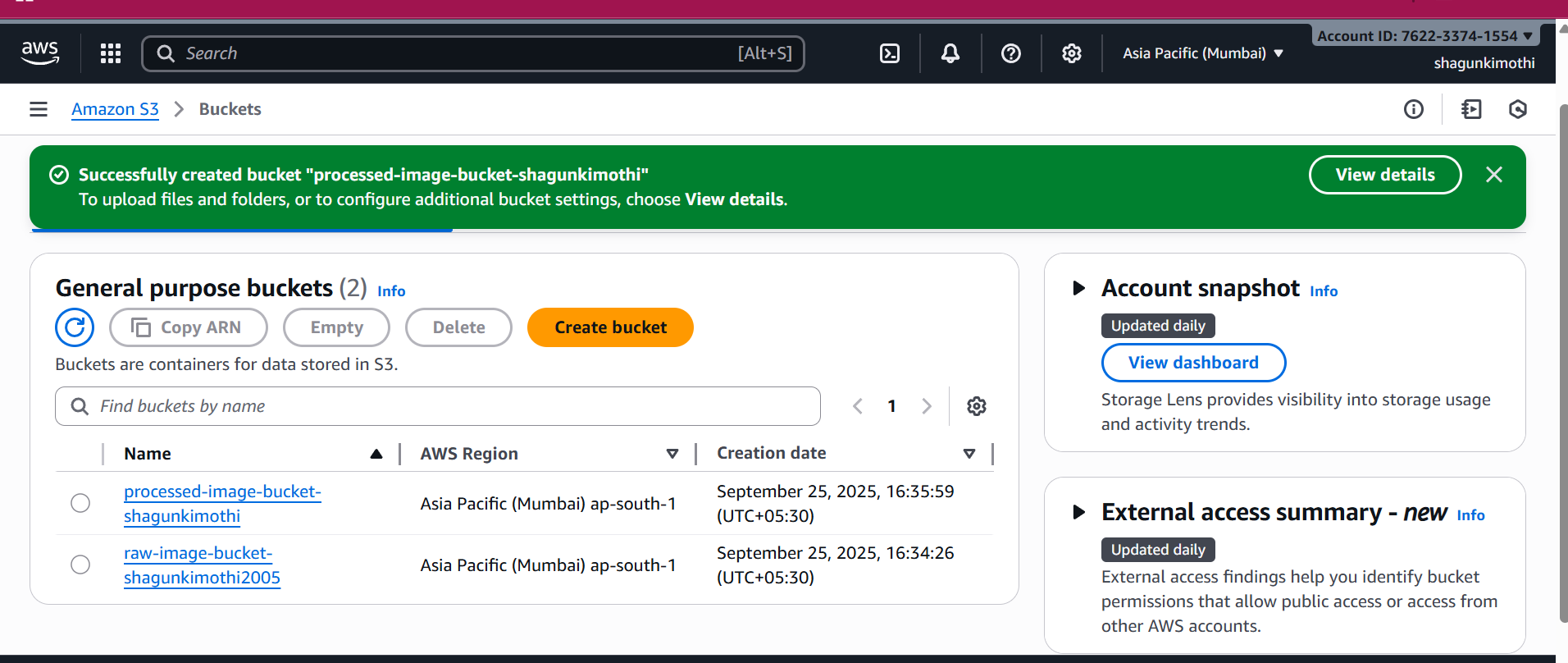
**Flow:**

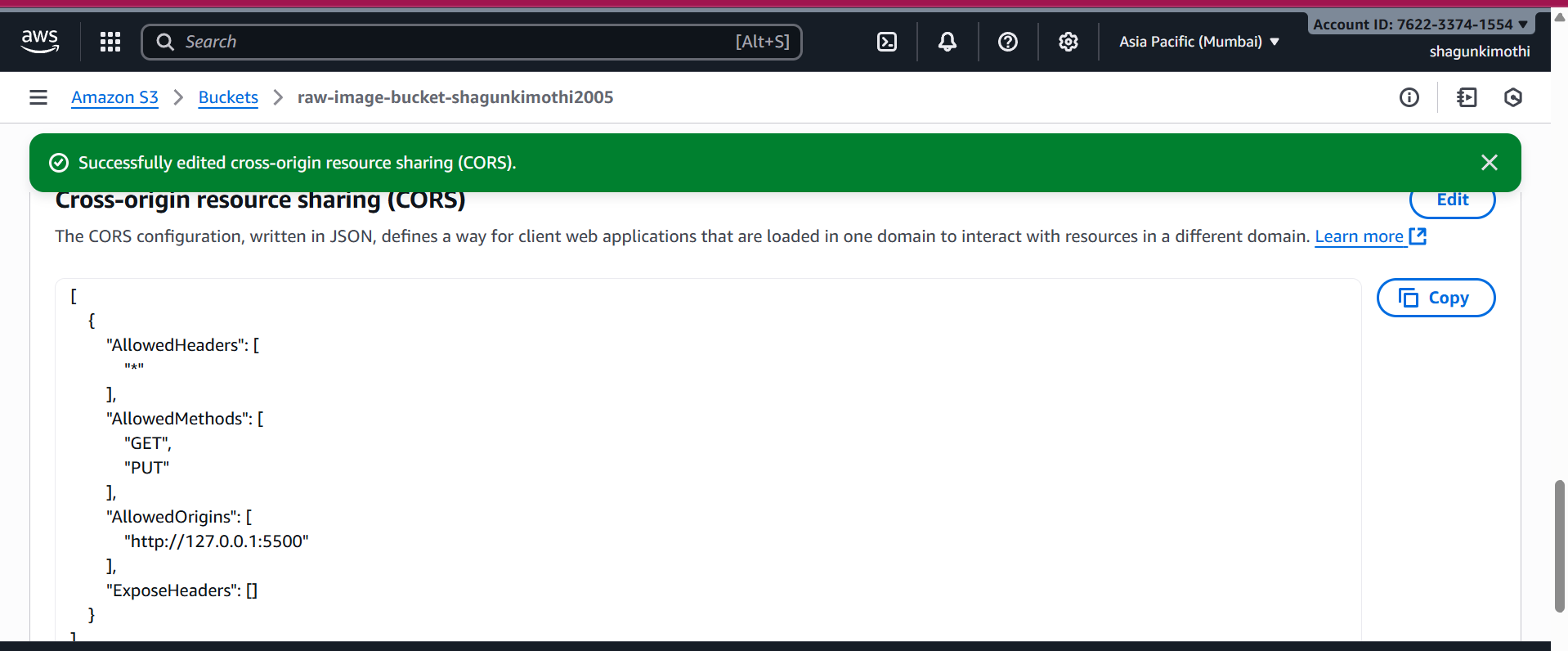
1. User uploads an image via the web interface.
2. Image is stored in **S3 raw bucket**.
3. **S3 event triggers Lambda** for processing (resize, compress, format change).
4. Processed image is stored in **S3 processed bucket**.
5. Image metadata (ID, timestamp, S3 URL) is stored in **DynamoDB**.
6. API Gateway serves as the interface for frontend-backend communication.

**6. Implementation Steps**

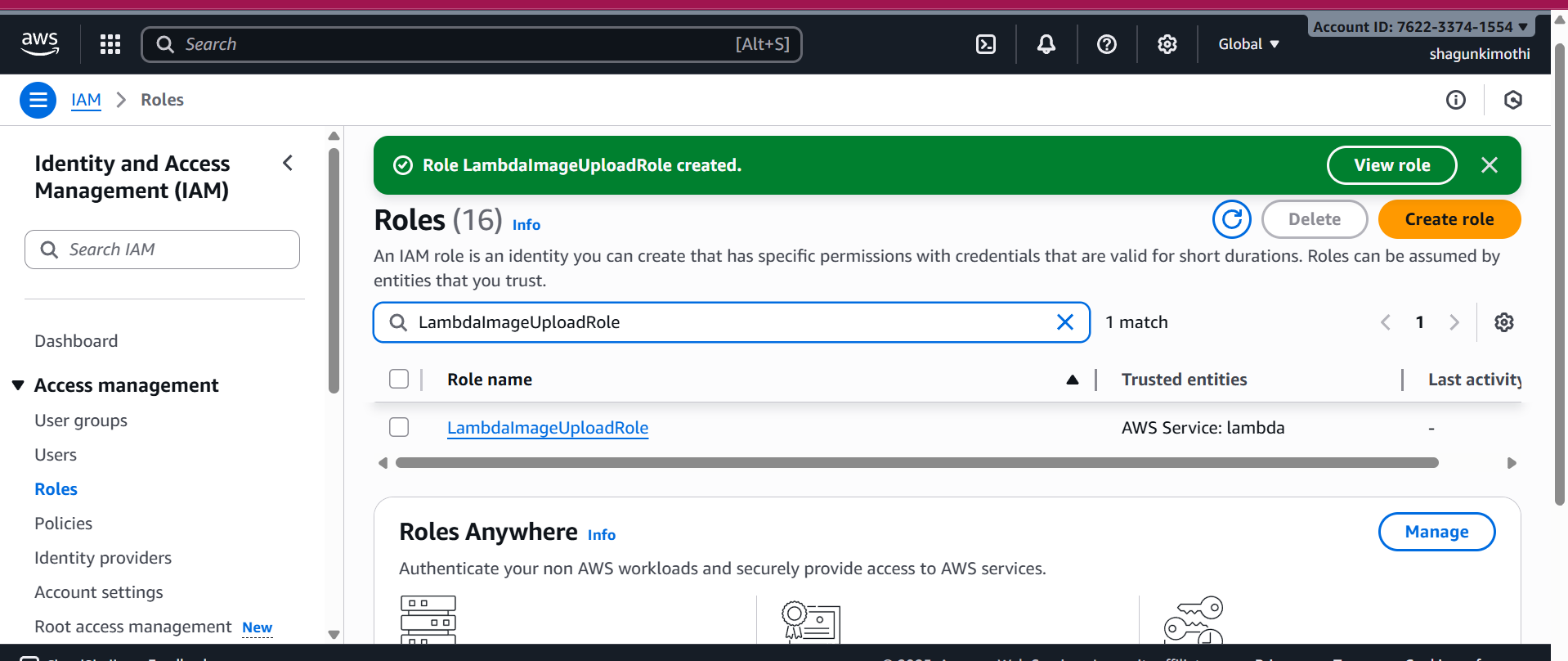
**6.1. AWS S3 Setup**

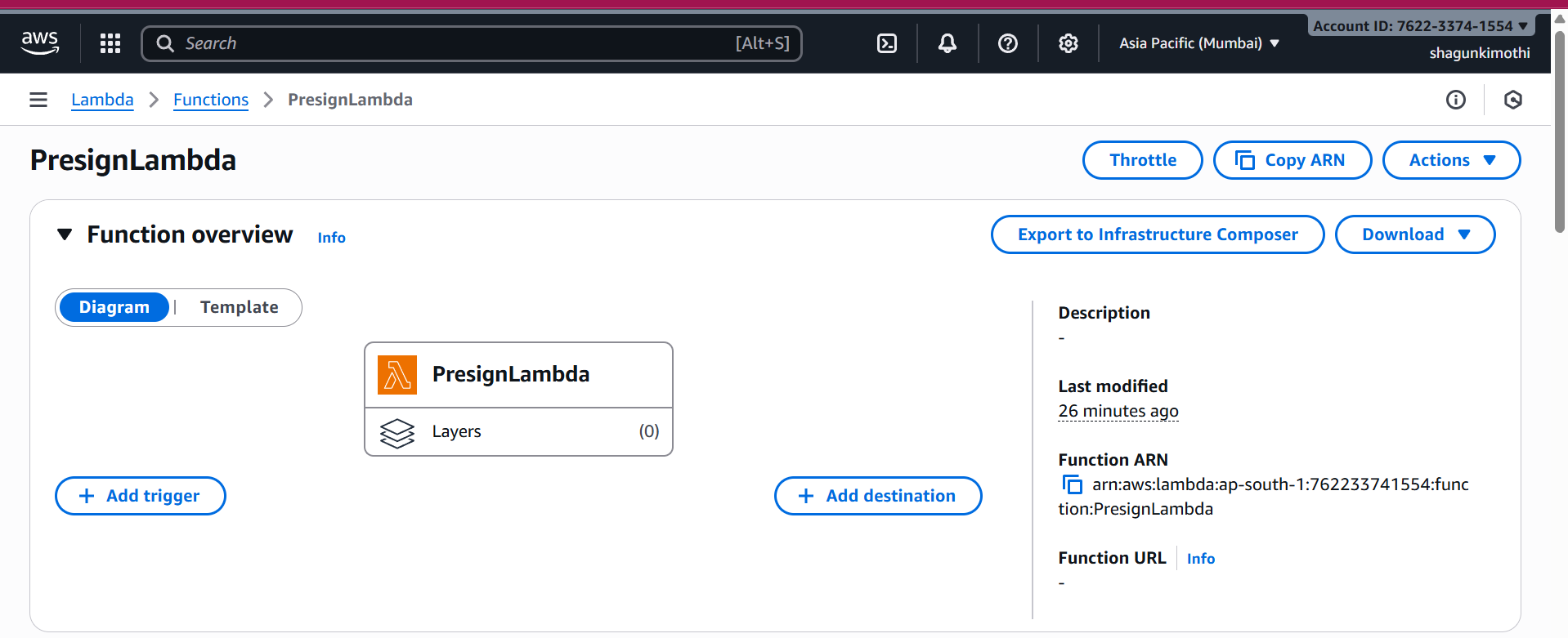
1. Create two S3 buckets:
   * raw-image-bucket (for original uploads)
   * processed-image-bucket (for processed images)
2. Enable **CORS** and proper **permissions** for uploads and Lambda triggers.





**6.2. AWS Lambda Function**

1. Create a Lambda function using Python.
2. Attach the necessary **IAM role** with S3 and DynamoDB access.
3. Function workflow:
   * Receive S3 event.
   * Read uploaded image.
   * Process the image (resize, convert).
   * Store processed image in S3 processed bucket.
   * Save metadata in DynamoDB.
   * 



**Sample Lambda Function Skeleton (Python):**

import json

import boto3

import os

import time

import uuid

# S3 client

s3 = boto3.client('s3')

BUCKET = os.environ.get("RAW\_BUCKET", "raw-image-bucket-shagunkimothi2005")

# DynamoDB client

dynamodb = boto3.resource('dynamodb')

TABLE\_NAME = os.environ.get("DDB\_TABLE", "ImagesMetadata")

table = dynamodb.Table(TABLE\_NAME)

def lambda\_handler(event, context):

try:

        # Unique filename

        filename = f"upload-{int(time.time())}.jpg"

        imgid = str(uuid.uuid4())          # Unique imgid

        timestamp = str(int(time.time()))  # DynamoDB sort key

        # Generate presigned S3 URL

        url = s3.generate\_presigned\_url(

            'put\_object',

            Params={'Bucket': BUCKET, 'Key': filename, 'ContentType': 'image/jpeg'},

            ExpiresIn=300

        )

 # Get userid from event (matches GSI 'userindex')

        userid = event.get('userid', 'anonymous')

        # Save metadata to DynamoDB

        table.put\_item(

            Item={

                'imgid': imgid,                      # Partition key

                'timestamp': timestamp,              # Sort key

                'filename': filename,

                'userid': userid,                    # For userindex GSI

                'url': f"https://{BUCKET}.s3.amazonaws.com/{filename}"  # For urlindex GSI

            }

)

        # Return presigned URL

        return {

            'statusCode': 200,

            'headers': {"Access-Control-Allow-Origin": "\*"},

            'body': json.dumps({

                "uploadURL": url,

                "filename": filename,

                "imgid": imgid

})

        }

    except Exception as e:

        return {

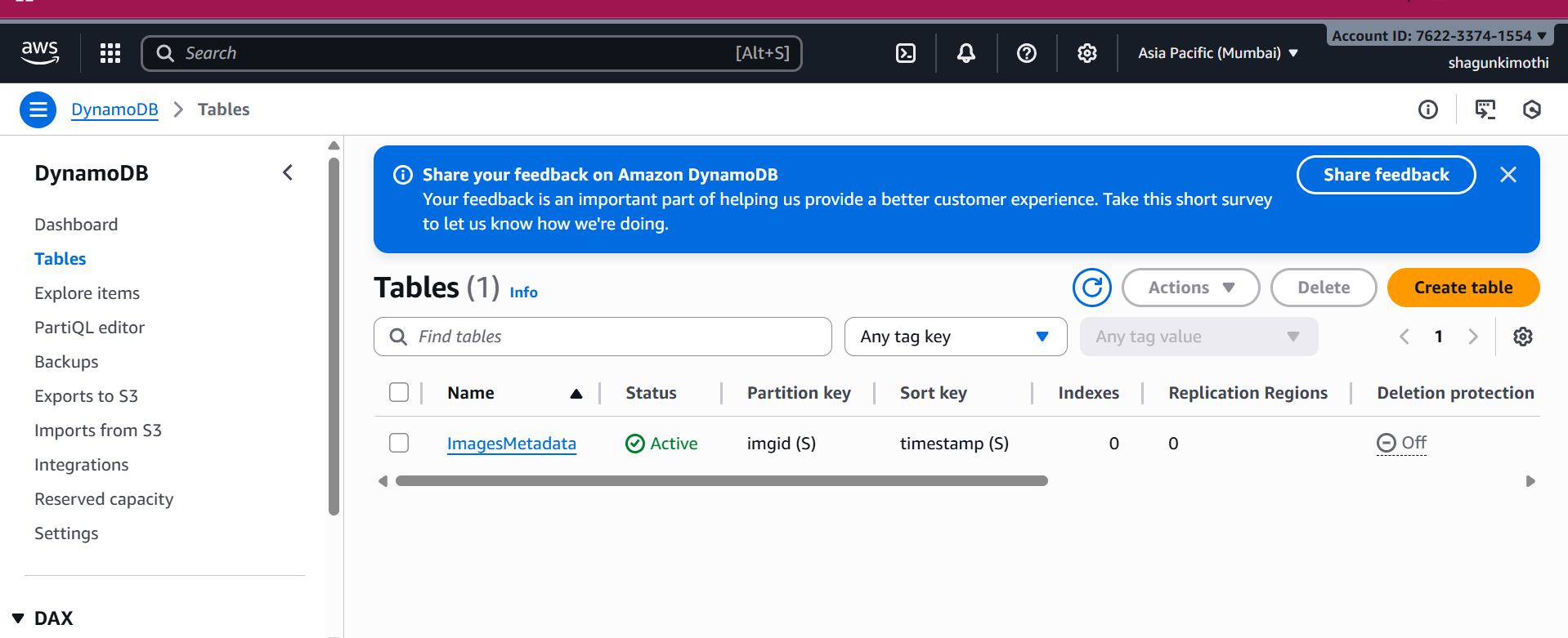
            'statusCode': 500,

            'body': json.dumps({"error": str(e)})

        }

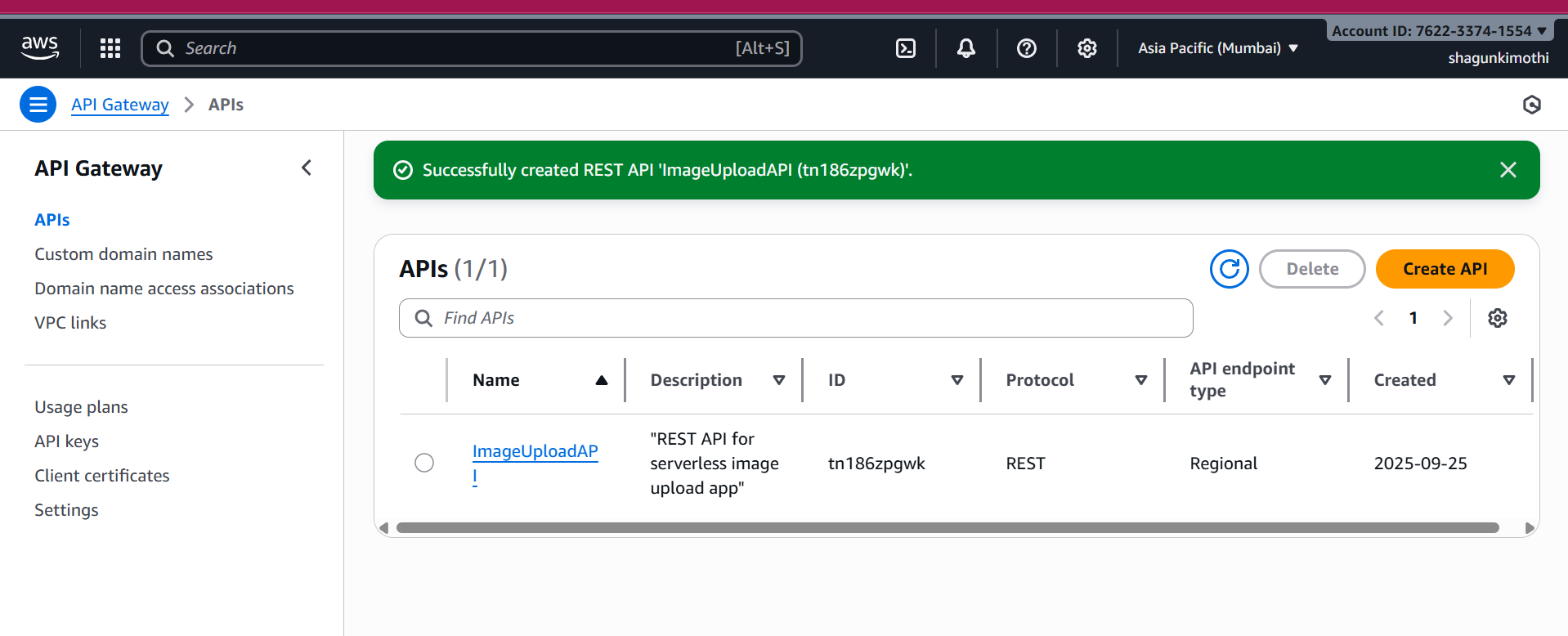
**6.3. DynamoDB Setup**

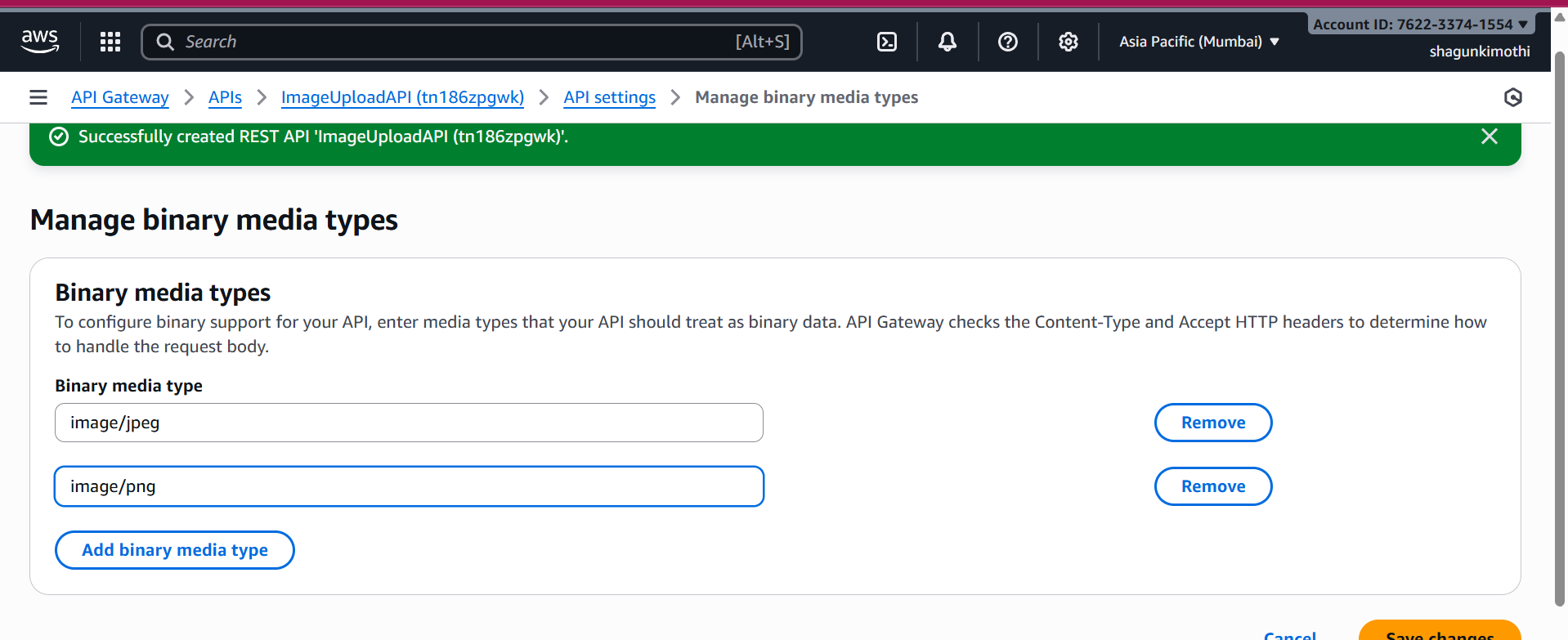
* Table Name: ImagesMetadata
* Partition Key: imageId (String)
* Attributes: imageName, timestamp, originalBucket, processedBucket
* Stores metadata of all uploaded and processed images.

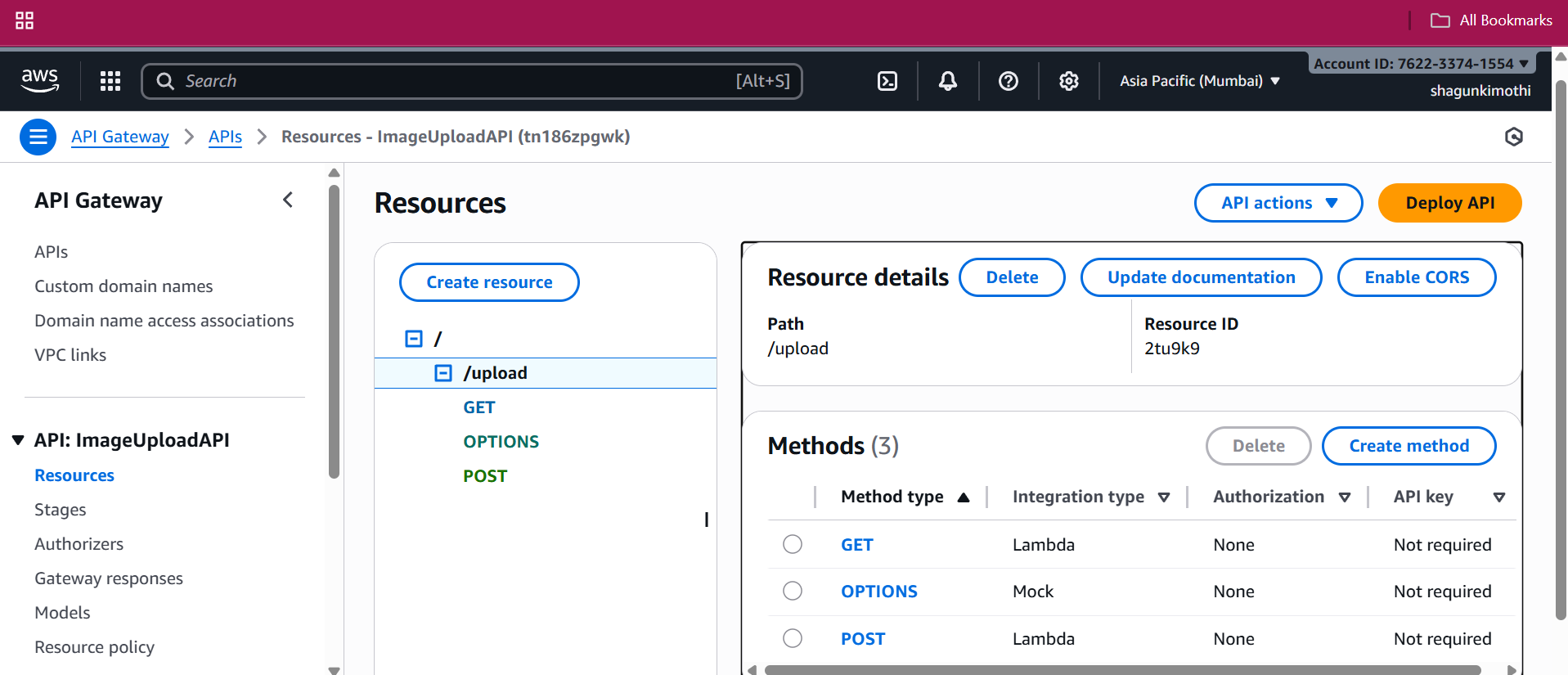


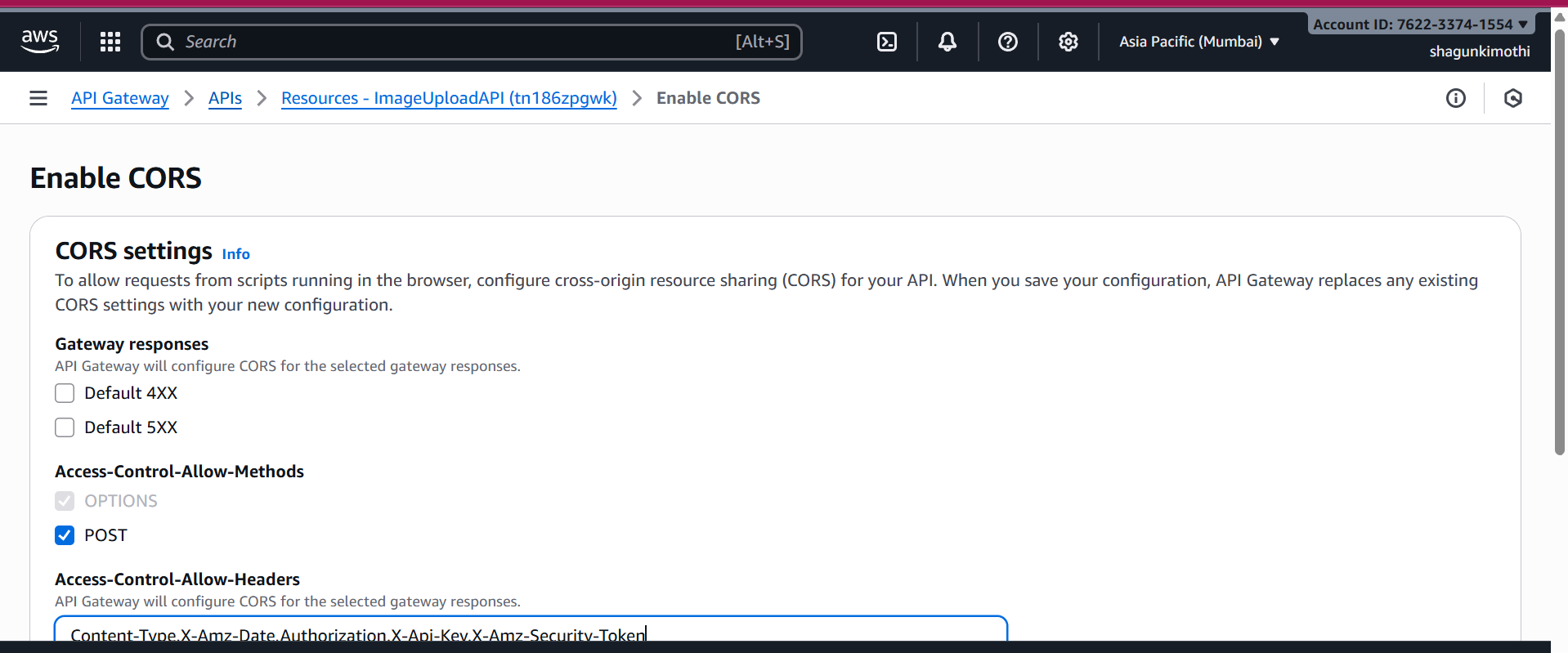
**6.4. API Gateway Setup**

* Create REST API.
* Create **POST endpoint** for image upload.
* Enable **CORS** for frontend access.
* Integrate endpoint with Lambda function.
* Deploy API to generate a **public URL**.









<https://tn186zpgwk.execute-api.ap-south-1.amazonaws.com/prod/upload>

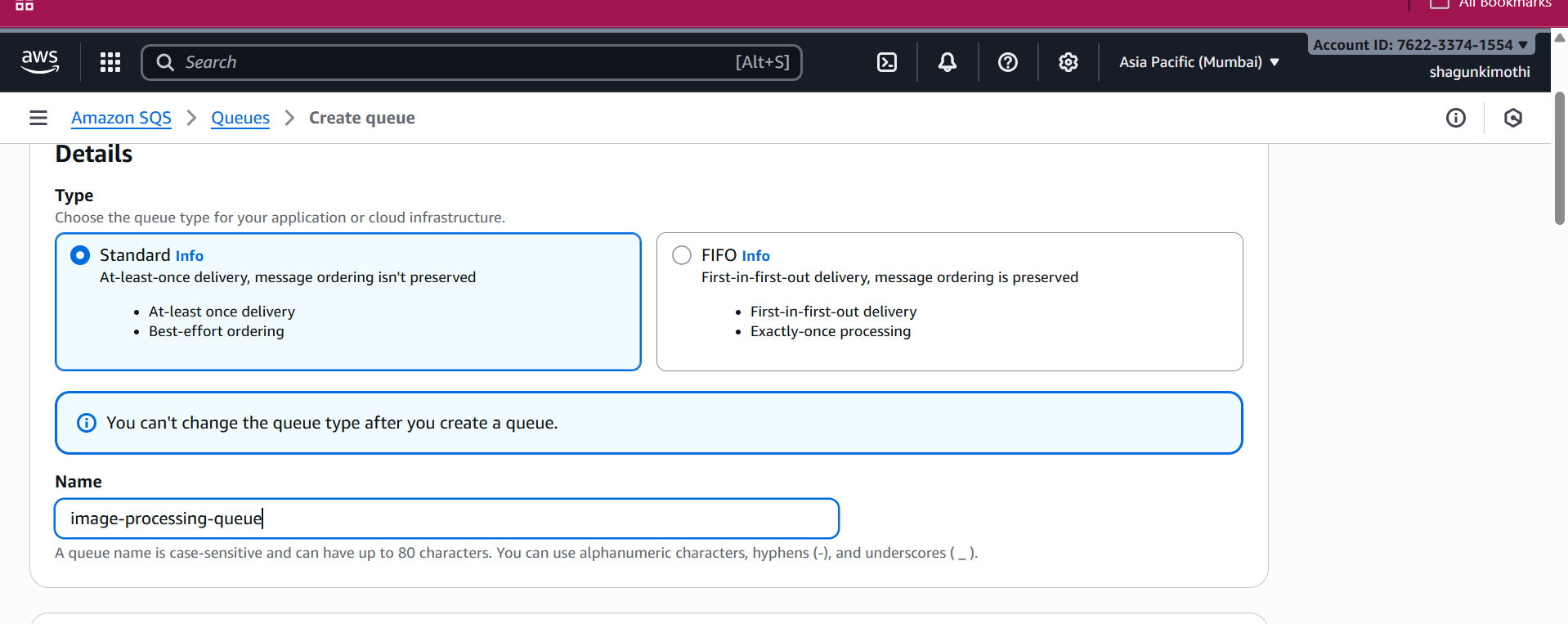
**6.5. Frontend Implementation**

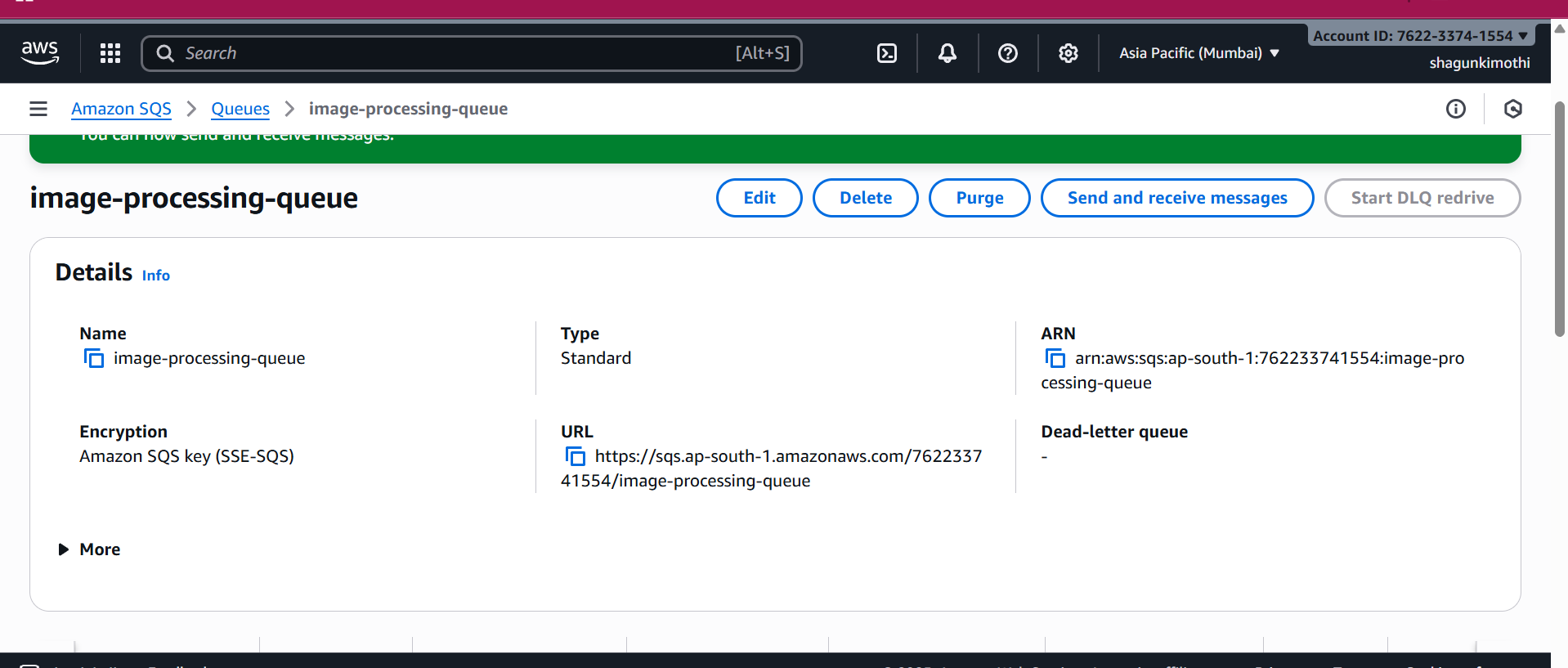
* Simple HTML form for uploading images.
* JavaScript to call the **API Gateway endpoint** with fetch() or XMLHttpRequest.
* Display uploaded image and metadata on the web page.

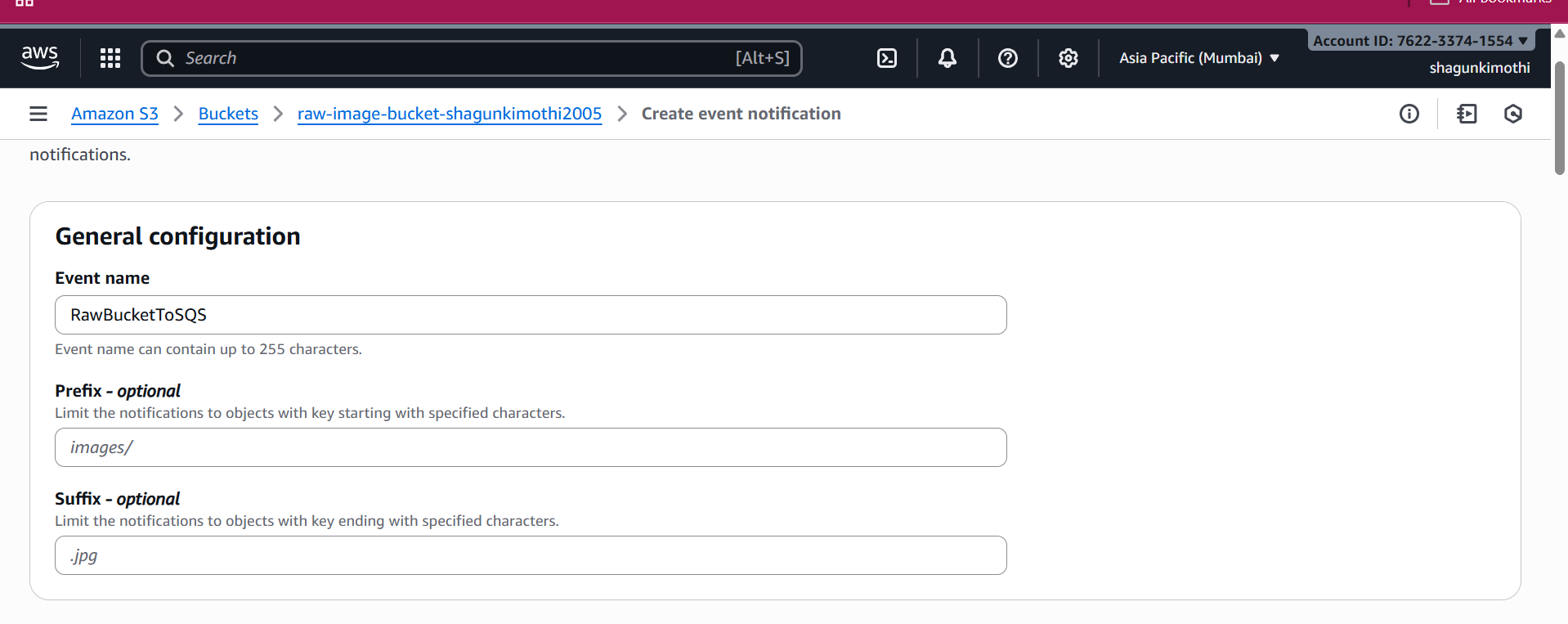
**Sample Frontend Snippet:**

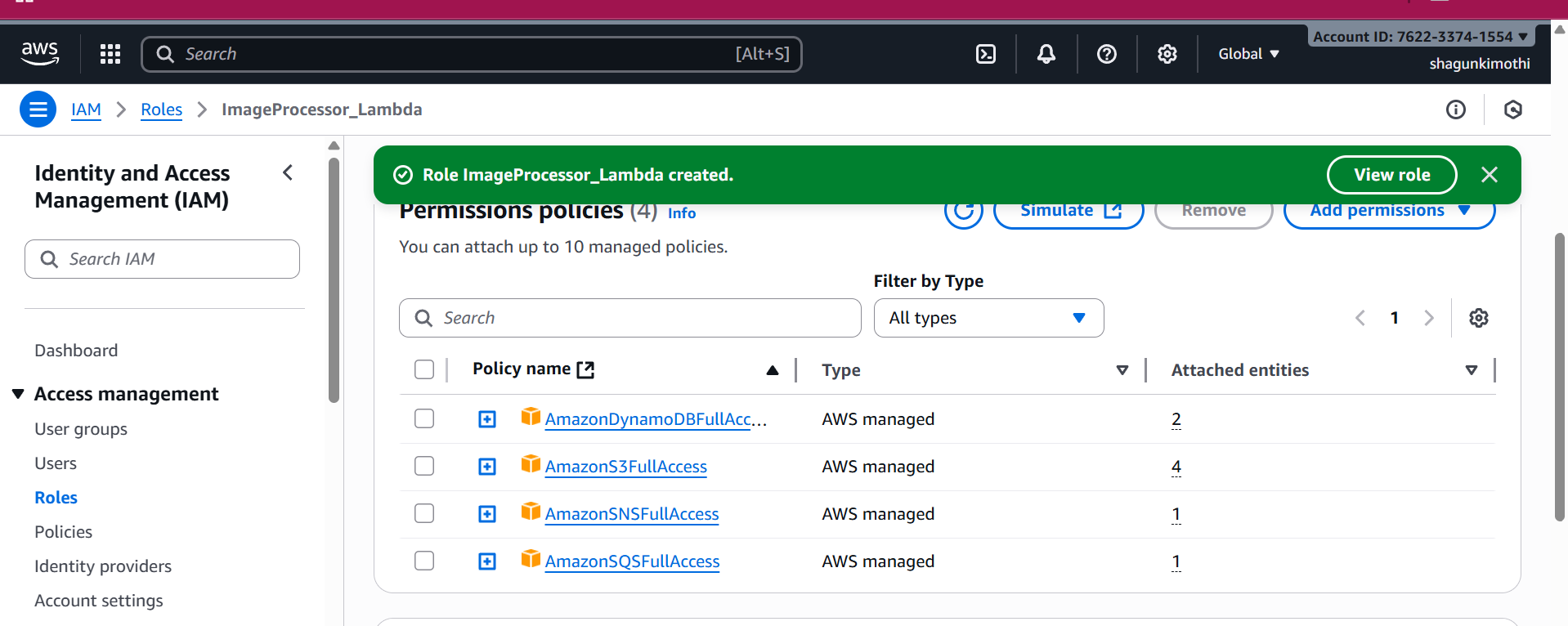
**Lambda Purpose**

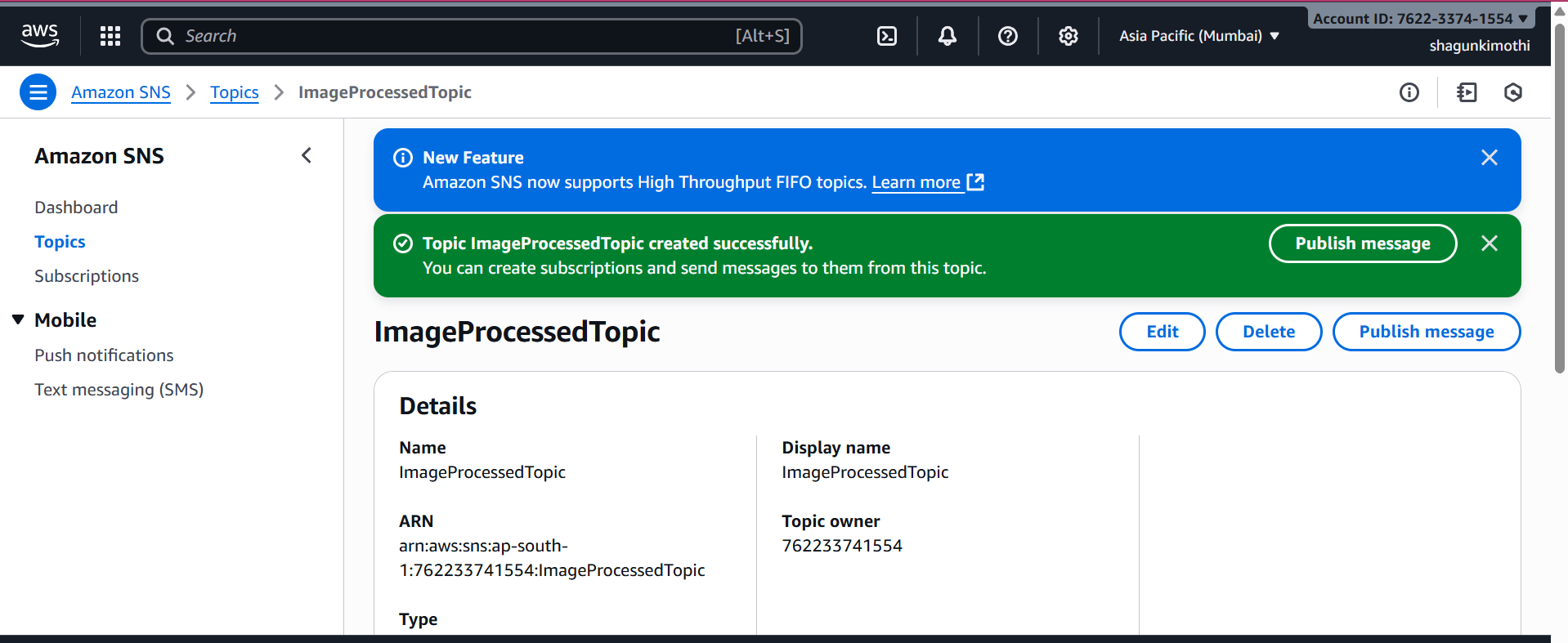
* **Name/Role:** Processed Image Lambda
* **Purpose:** Triggered by new images in the **raw S3 bucket**, this Lambda uploads images to the **processed S3 bucket**.
* **Trigger:** Connected via **SNS/SQS** notifications from the raw bucket upload events.
* **Metadata:** Saves relevant information in **DynamoDB**, including imgid, raw image key, and processed image key.











import boto3

import os

from datetime import datetime

s3 = boto3.client('s3')

dynamodb = boto3.resource('dynamodb')

TABLE\_NAME = os.environ['DDB\_TABLE']

PROCESSED\_BUCKET = os.environ['PROCESSED\_BUCKET']

table = dynamodb.Table(TABLE\_NAME)

def lambda\_handler(event, context):

    print("Incoming event:", event)

    # If no Records key, return safely

    if 'Records' not in event:

        return {"status": "no S3 event", "event": event}

    for record in event['Records']:

        source\_bucket = record['s3']['bucket']['name']

        object\_key = record['s3']['object']['key']

        # Copy image to Processed bucket

        s3.copy\_object(

            Bucket=PROCESSED\_BUCKET,

            Key=object\_key,

            CopySource={'Bucket': source\_bucket, 'Key': object\_key}

        )

        # Save metadata

        table.put\_item(Item={

            'imageId': object\_key,

            'timestamp': datetime.utcnow().isoformat()

        })

        print(f"✅ {object\_key} copied to {PROCESSED\_BUCKET} and metadata stored.")

    return {"status": "done"}

**2. Workflow**

1. User uploads an image to the **raw bucket**.
2. Raw bucket triggers an **SQS/SNS event**.
3. **Processed Lambda** receives the event.
4. Lambda uploads the file to the **processed bucket**.
5. Lambda writes metadata to **DynamoDB** for tracking.
6. Frontend/gallery can fetch processed image URLs from DynamoDB.

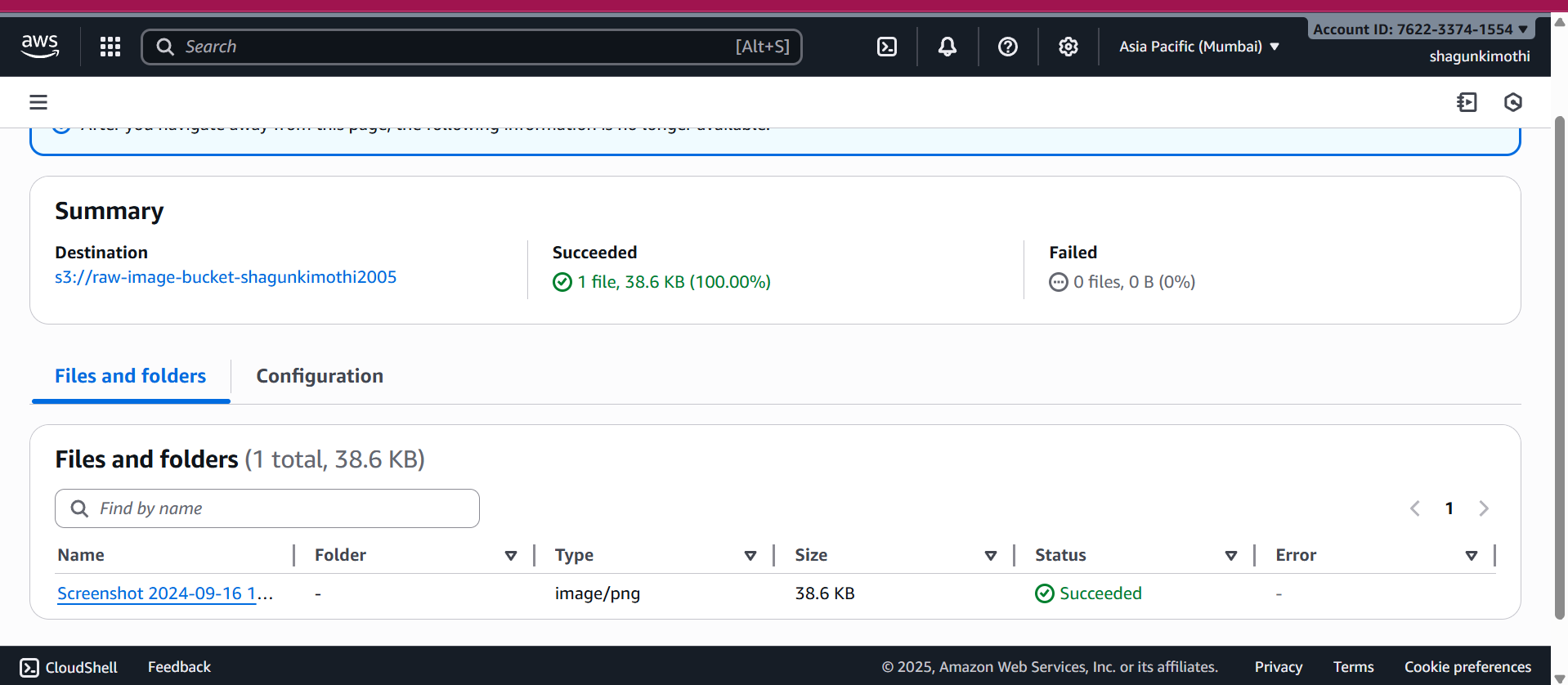
**3. Current Behavior**

* The Lambda currently **does not perform any image transformation** (resize, filter, watermark, etc.).
* The “processed” image is **identical** to the raw image.
* The gallery displays images from the processed bucket.

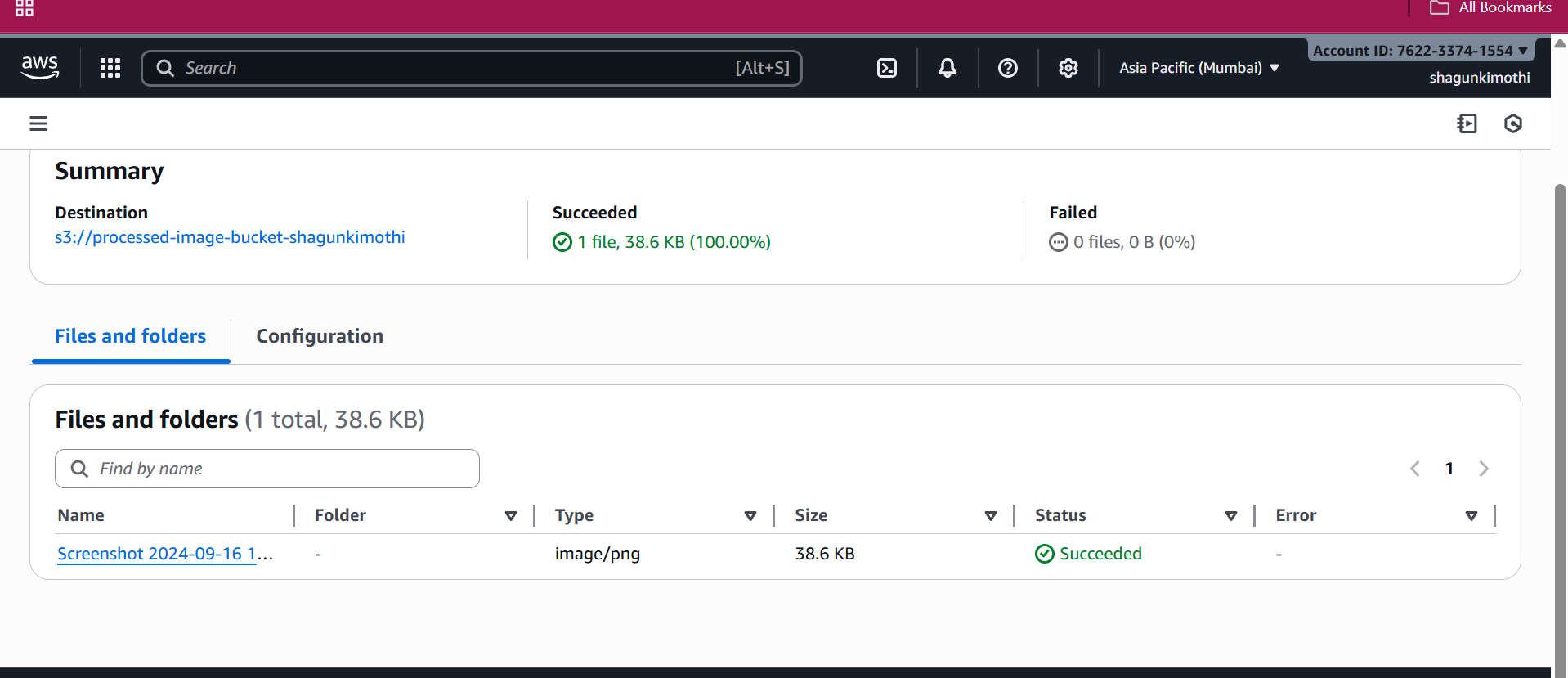
1. **Notes**

* Lambda uses **temporary storage** (/tmp/) if any file manipulation is done.
* Integration with SNS/SQS ensures **scalable, event-driven architecture**.
* DynamoDB metadata ensures images can be **queried by user or timestamp**.

Raw image



Processed image



Final

