**Automating Image Processing with AWS Lambda and S3: A Detailed Guide**

**Blog By: Mohammad-Mujahid**

**Date: 4 July-2024**

**Aim:** In this project, we set up an AWS Lambda function triggered by S3 uploads to process images (e.g., resizing them). This blog post documents the entire process, including the challenges faced and how we resolved them.

**Project Overview**

The goal of this project is to create an automated image processing pipeline using AWS Lambda and S3. When an image is uploaded to an S3 bucket, a Lambda function is triggered to resize the image and save the processed image back to a specified S3 bucket.

**Step 1: Setting Up the S3 Bucket**

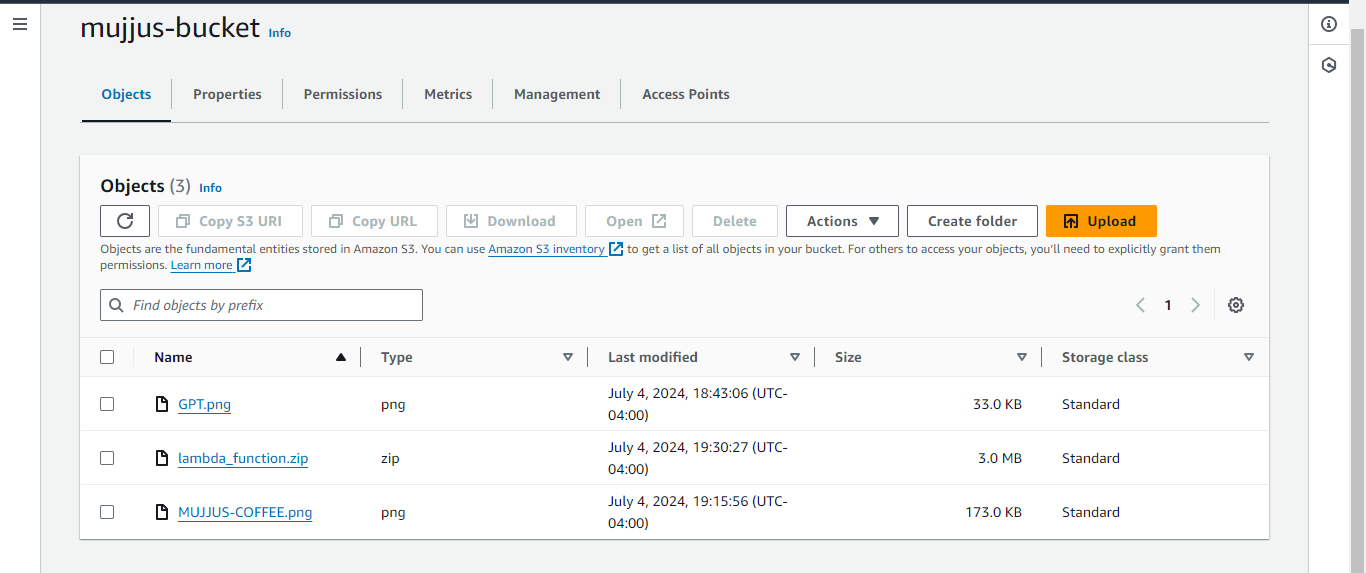
The first step is to create an S3 bucket to store the images. This bucket will trigger the Lambda function whenever a new image is uploaded.

Create an S3 Bucket:

Open the AWS Management Console.

Navigate to the S3 service.

Click "Create bucket" and follow the prompts to create a new bucket.



Step 2: Creating the Lambda Function

Next, we create a Lambda function that will process the images.

Create a Lambda Function:

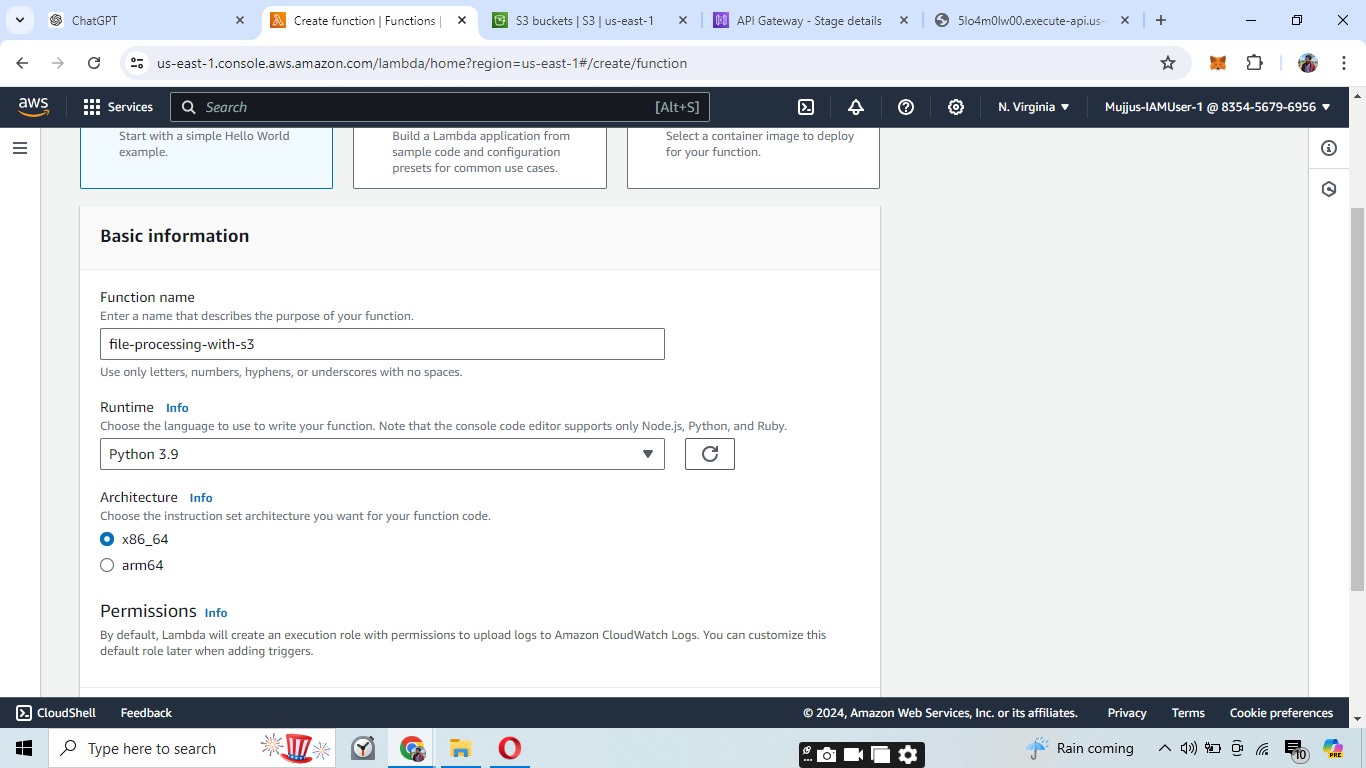
Open the AWS Management Console.

Navigate to the Lambda service.

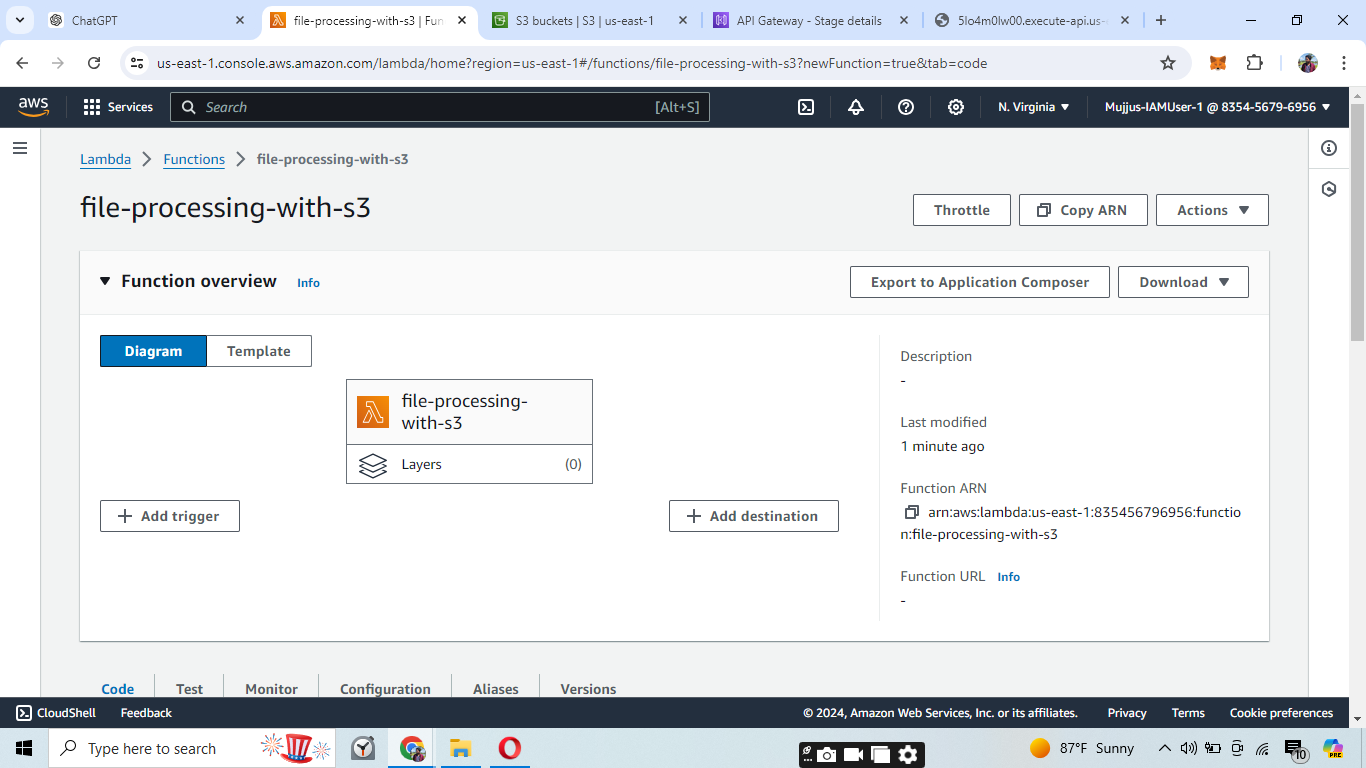
Click "Create function" and choose "Author from scratch".

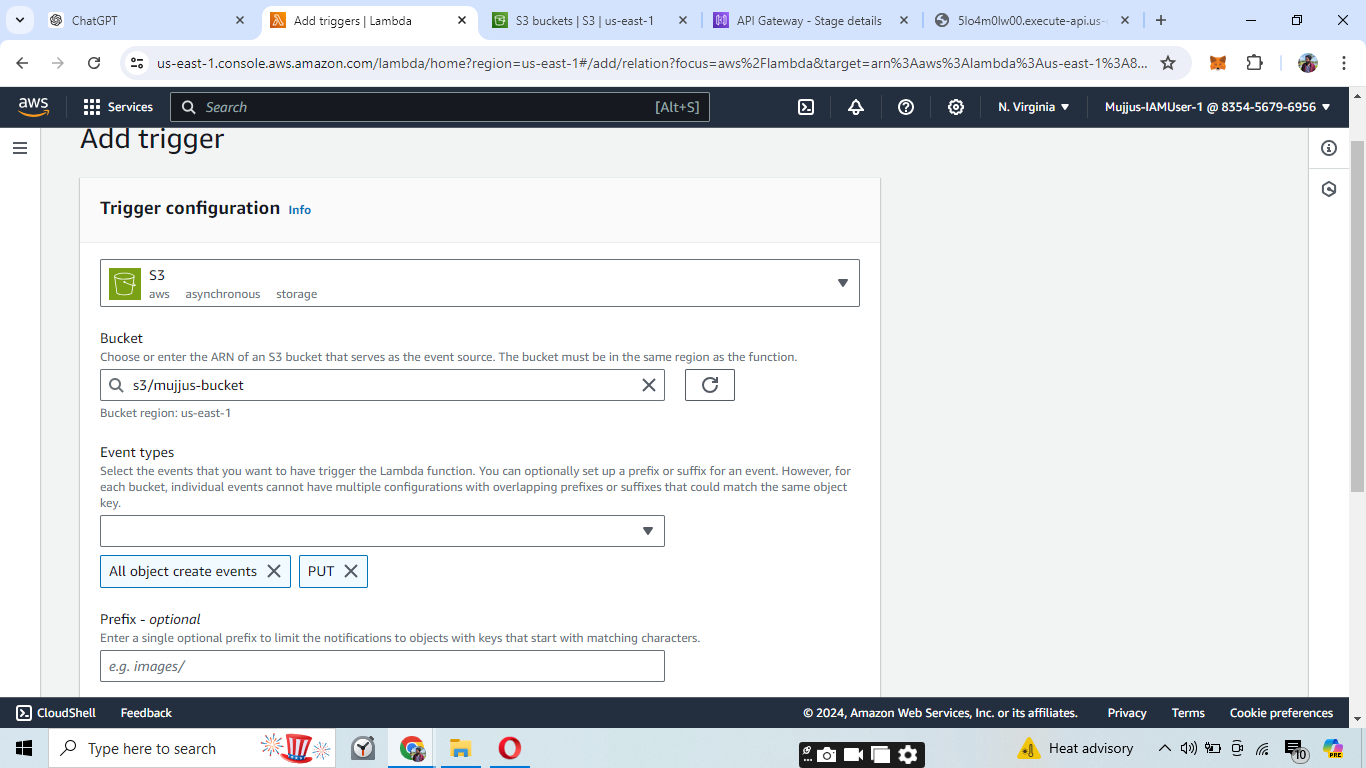
Provide a function name (e.g., file-processing-with-s3) and select the runtime (Python 3.9).

Choose "Create a new role with basic Lambda permissions".

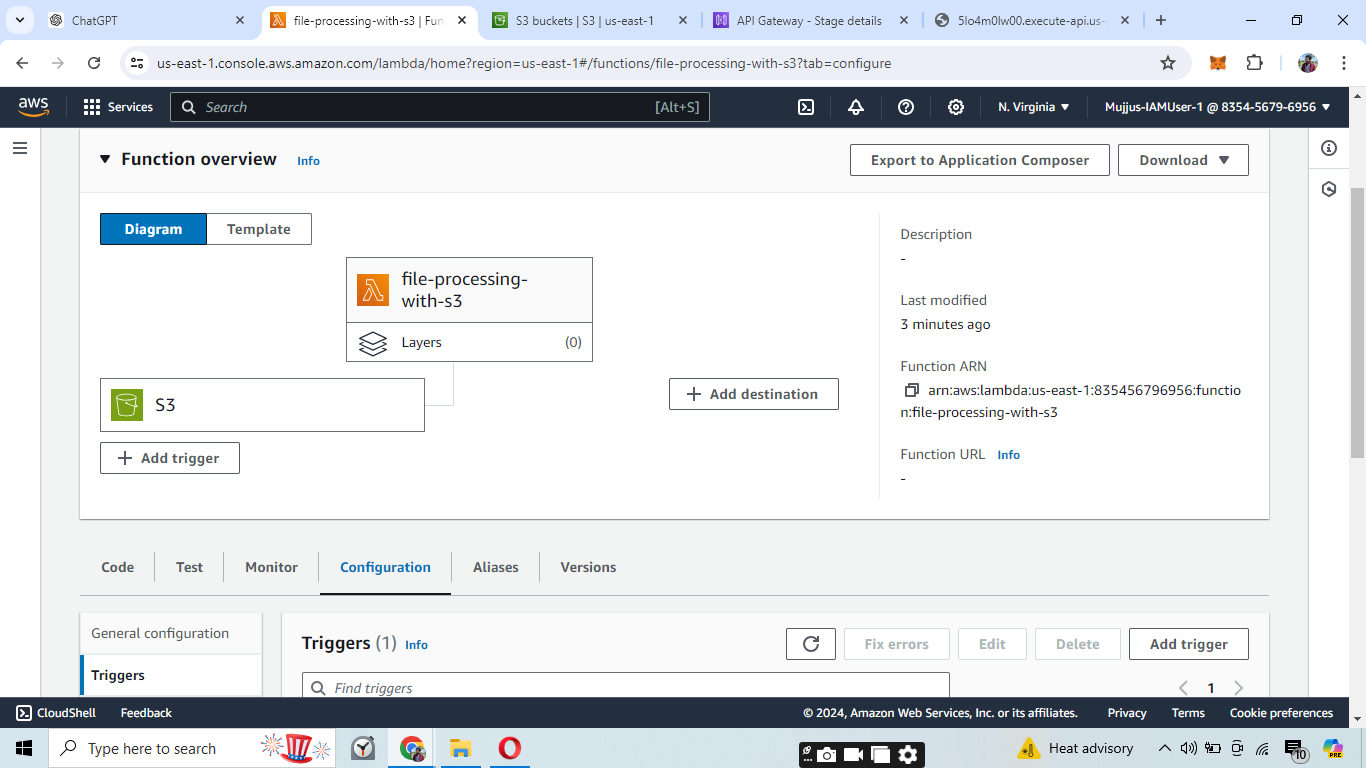


Then add trigger from Lambda function to S3 so that whenever user uploads something to the S3 bucket the lambda function will trigger.

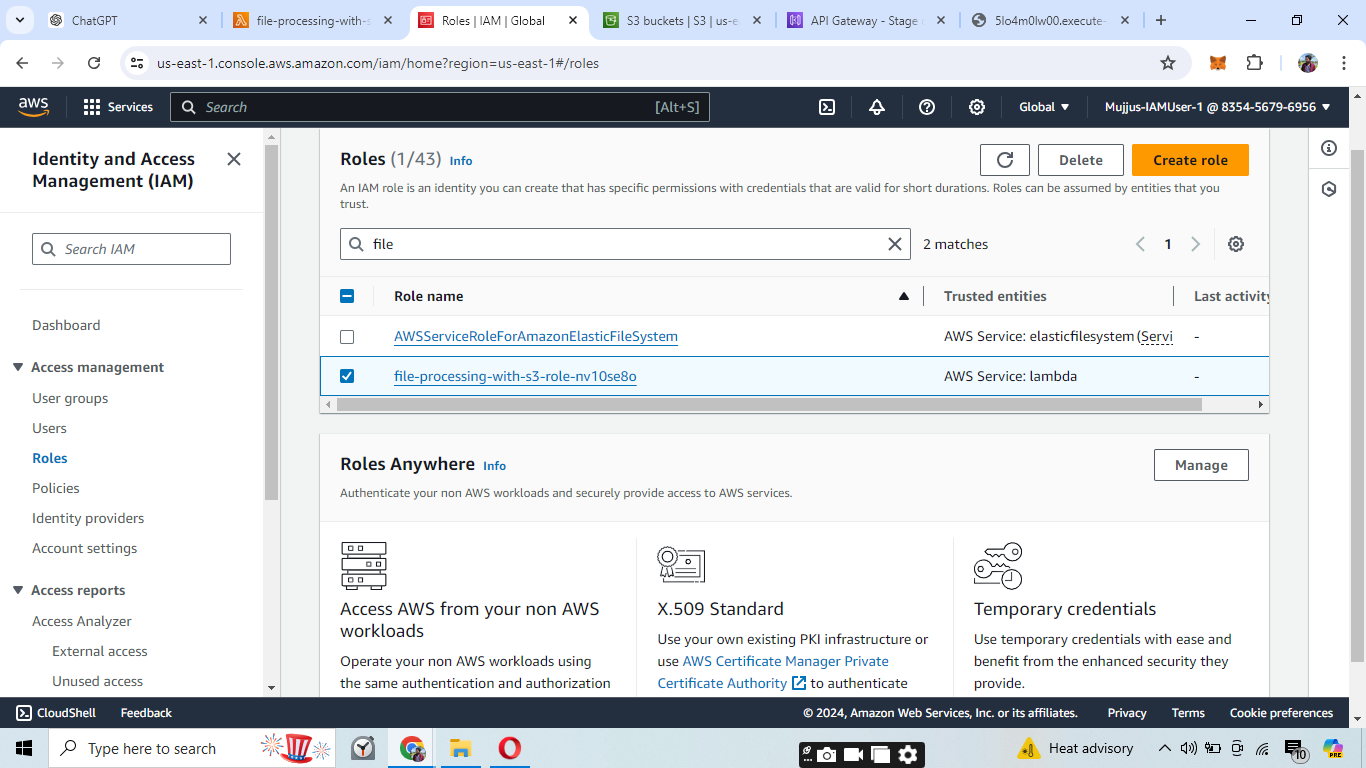


Select Event Type as Put when someone uploads anything to the S3-Bucket The lambda Function will Trigger

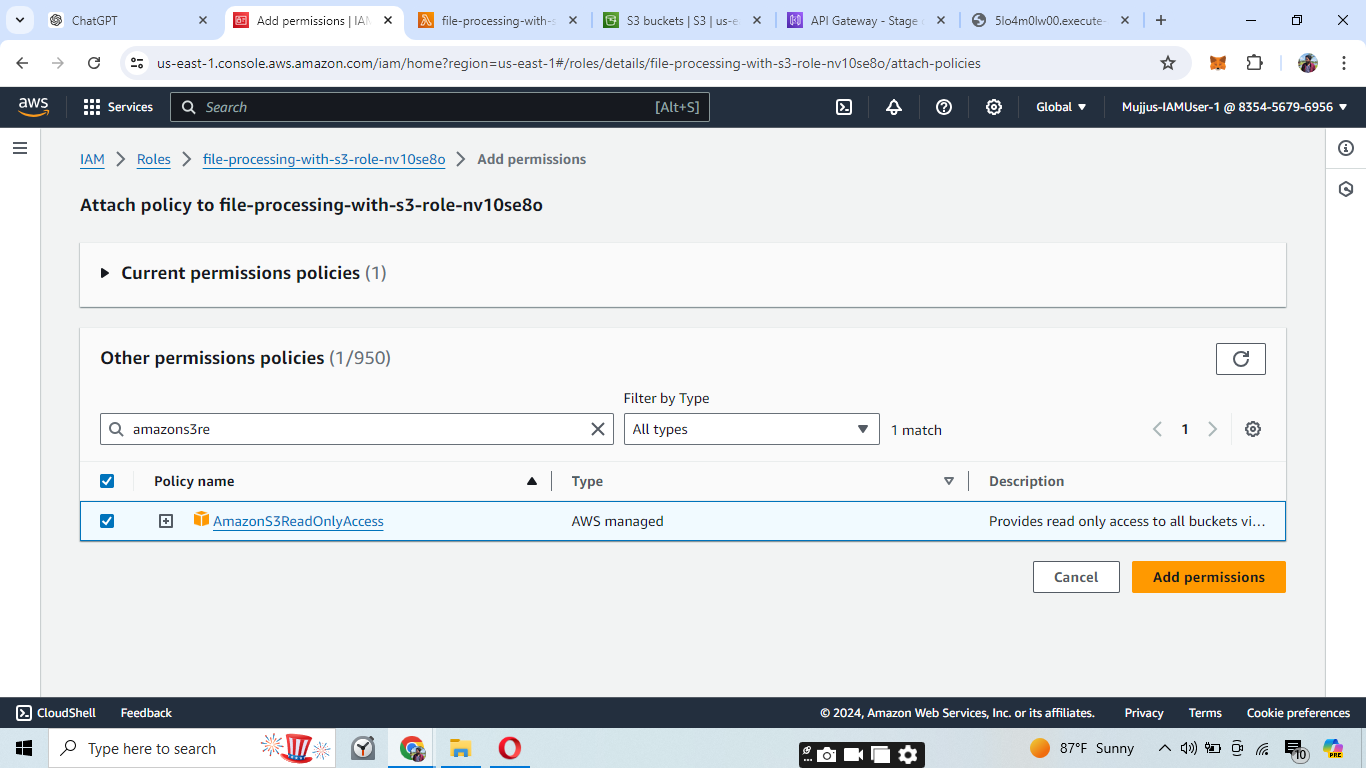
After Creating the S3-Bucket it looks like this:



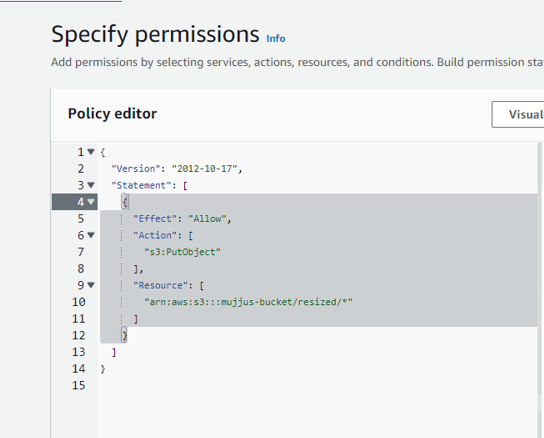
After that Attach Iam Roles to the Lambda Function so that it can access the S3 bucket.



Then go inside the Lamda function and attach a policy AmazonS3ReadOnlyAccess to the lambda function.

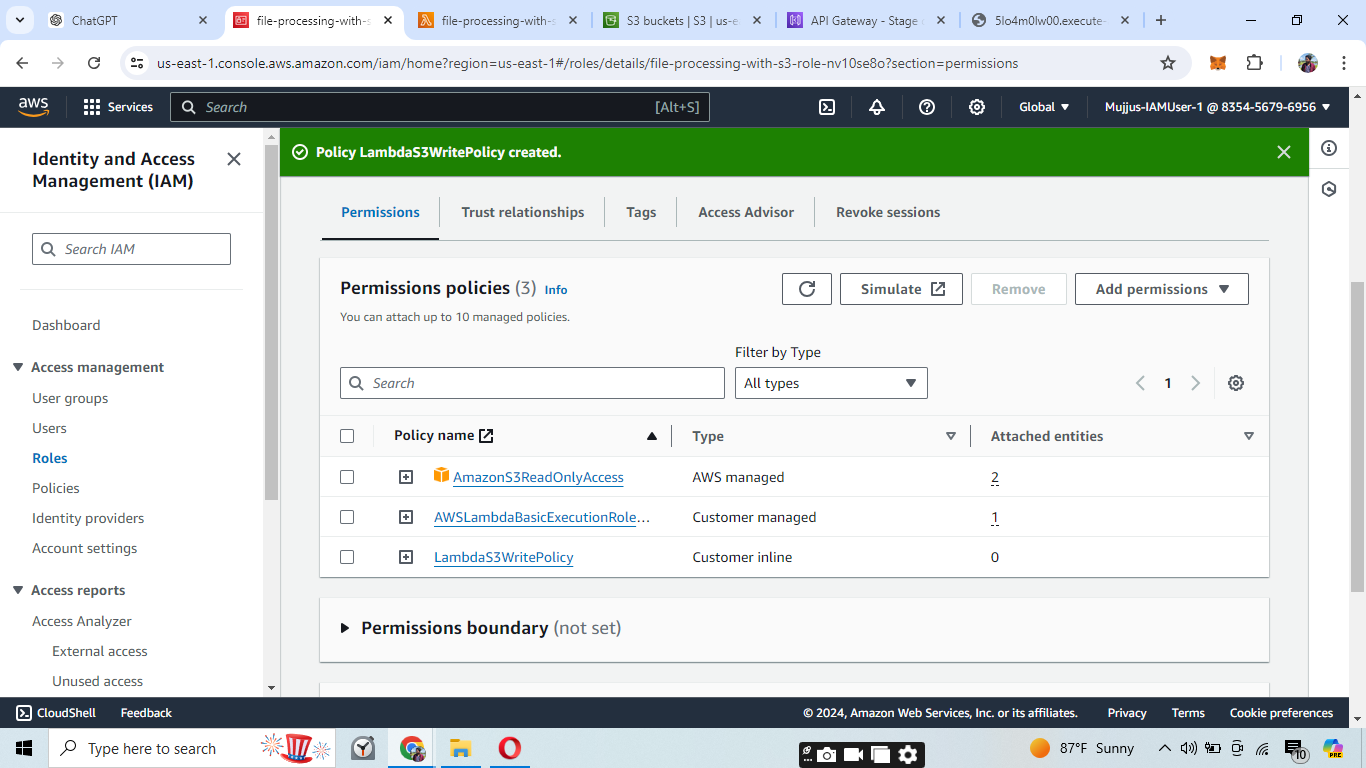


Then create an inline policy by clicking on Add permissions.

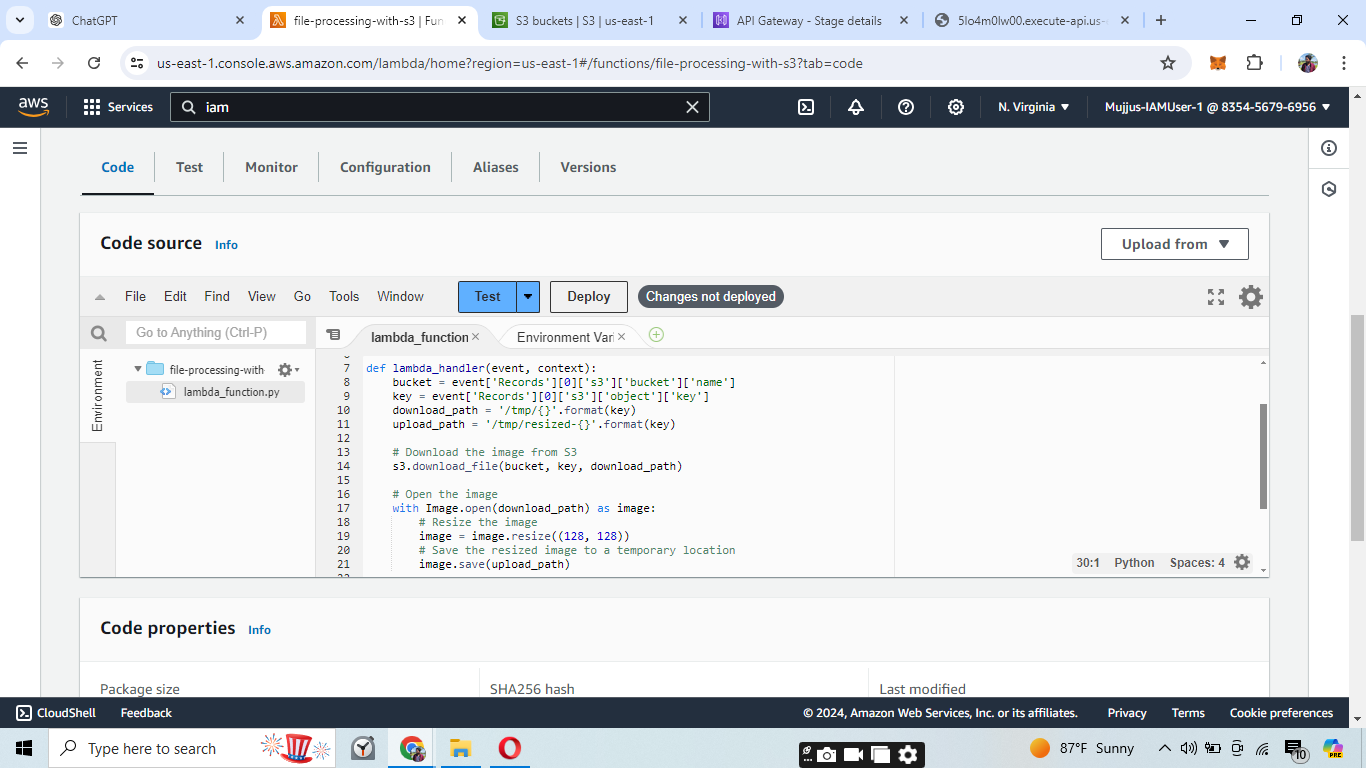


And this the Write Policy.

Now these are the policies attached to our Lambda function.

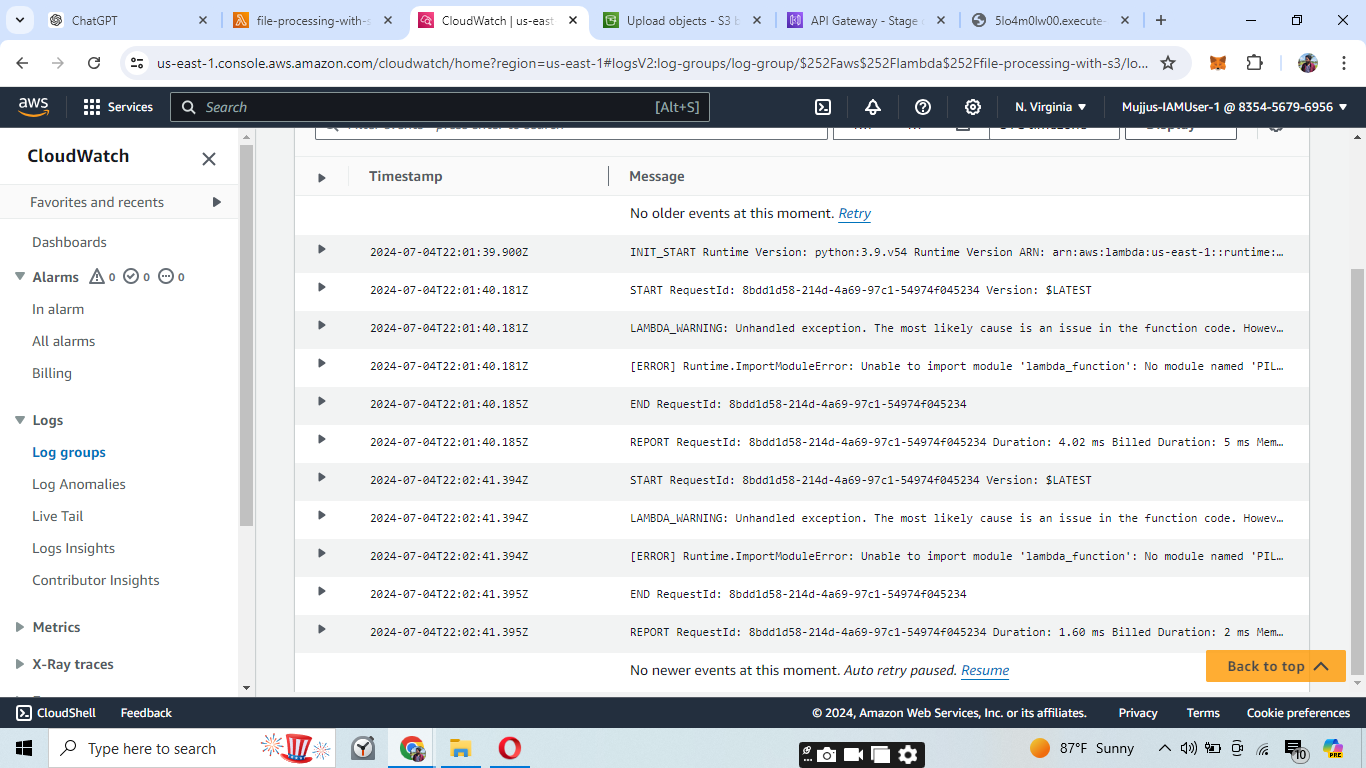


Now go to the Lambda Function and write the code:

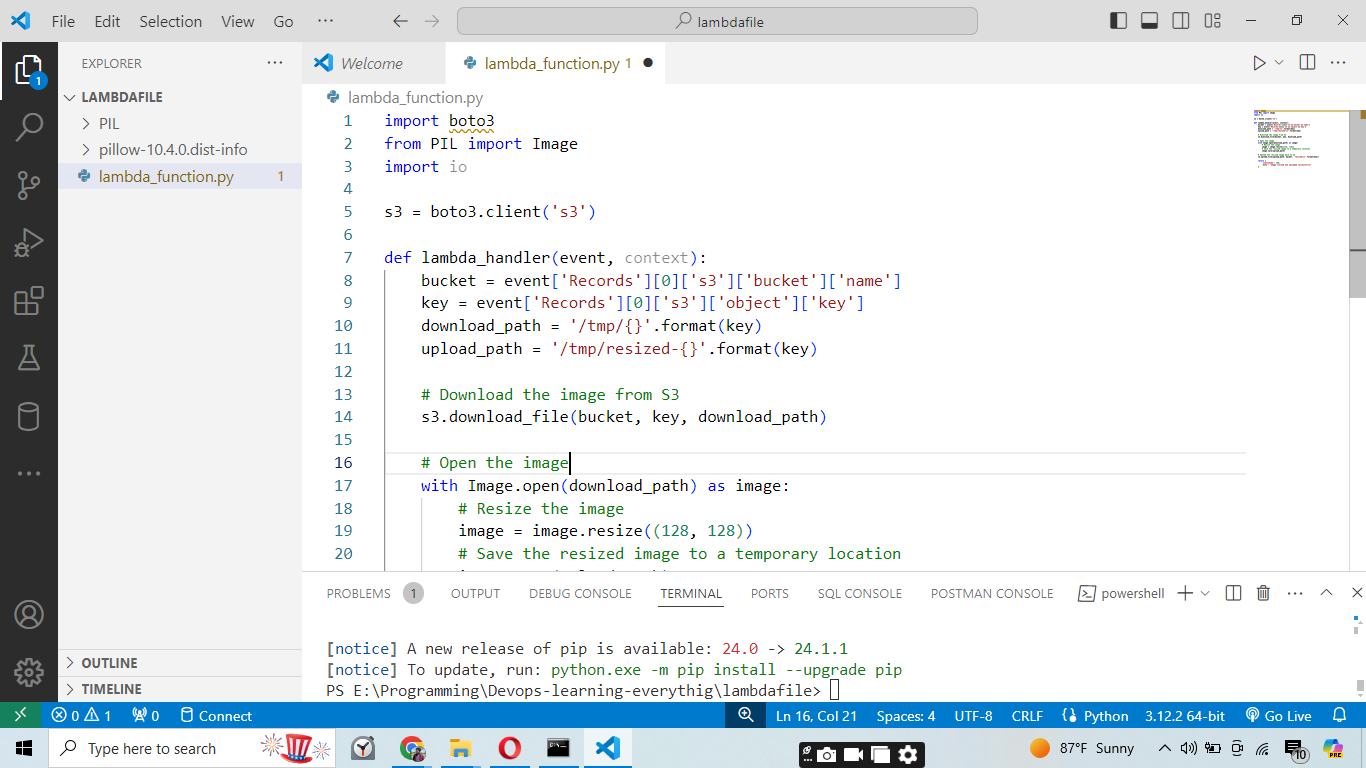


Now upload something to the S3 Bucket.

After uploading go to the CloudWatch to see the logs of the trigger.

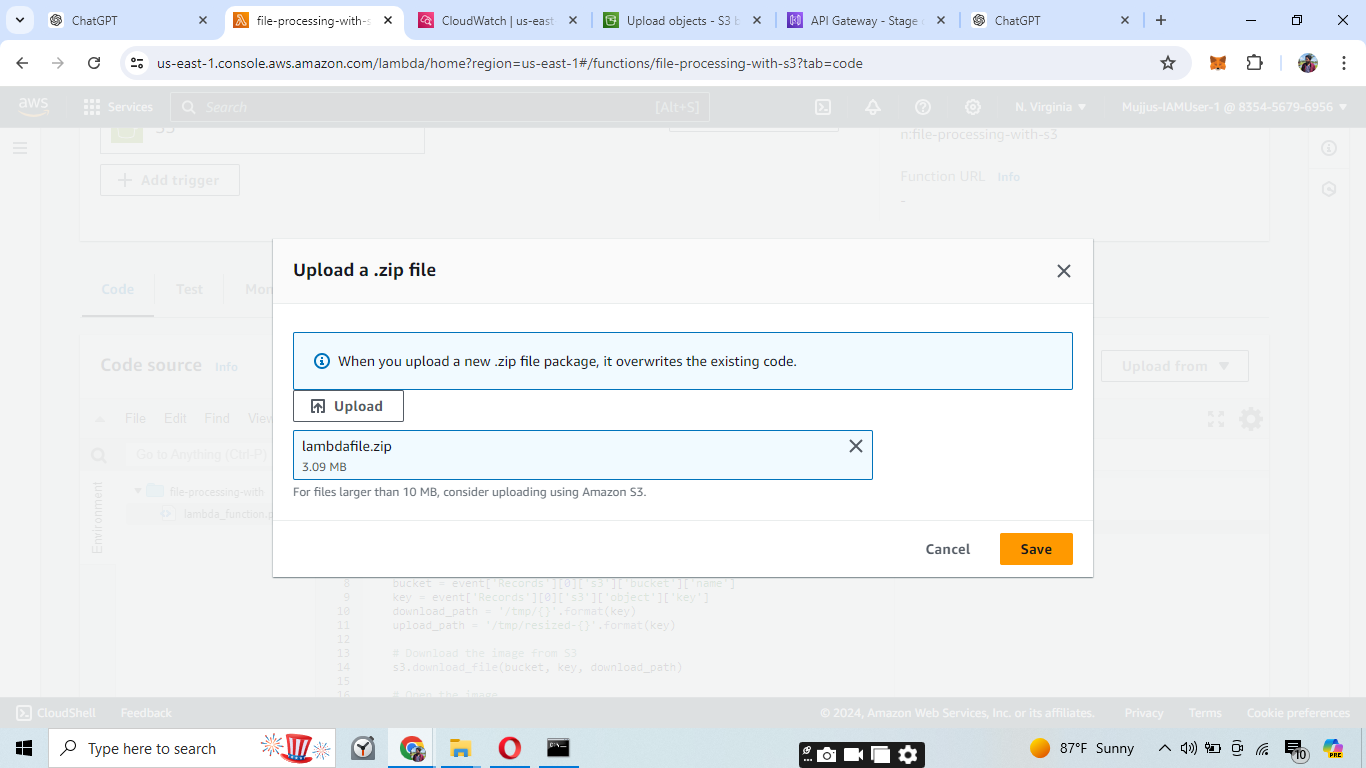


Now we are facing lot of issues here Bcoz PIL imports are not imported directly on the aws now here we have to package the whole code in a zip file and then we can upload it to the lambda function.

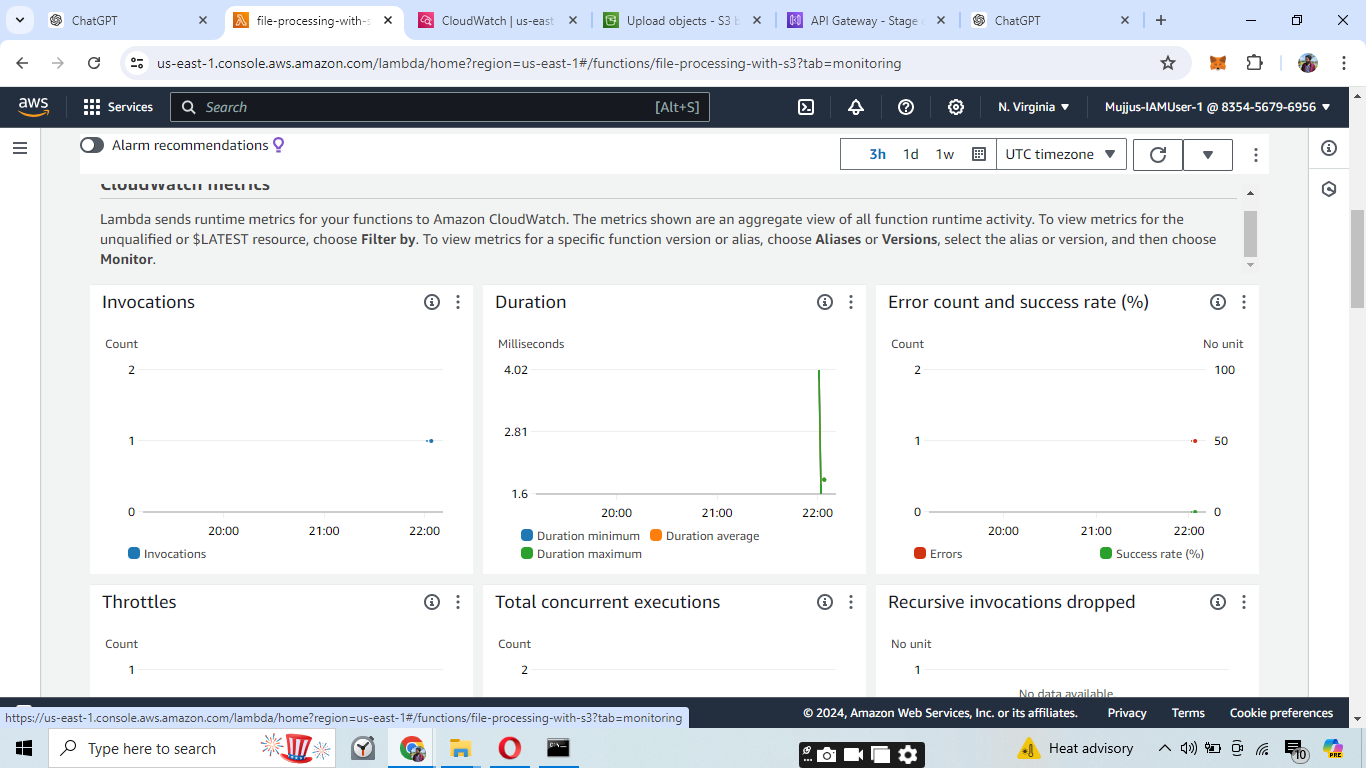


In this code we have pillow library and packaged it using a 7Zip.

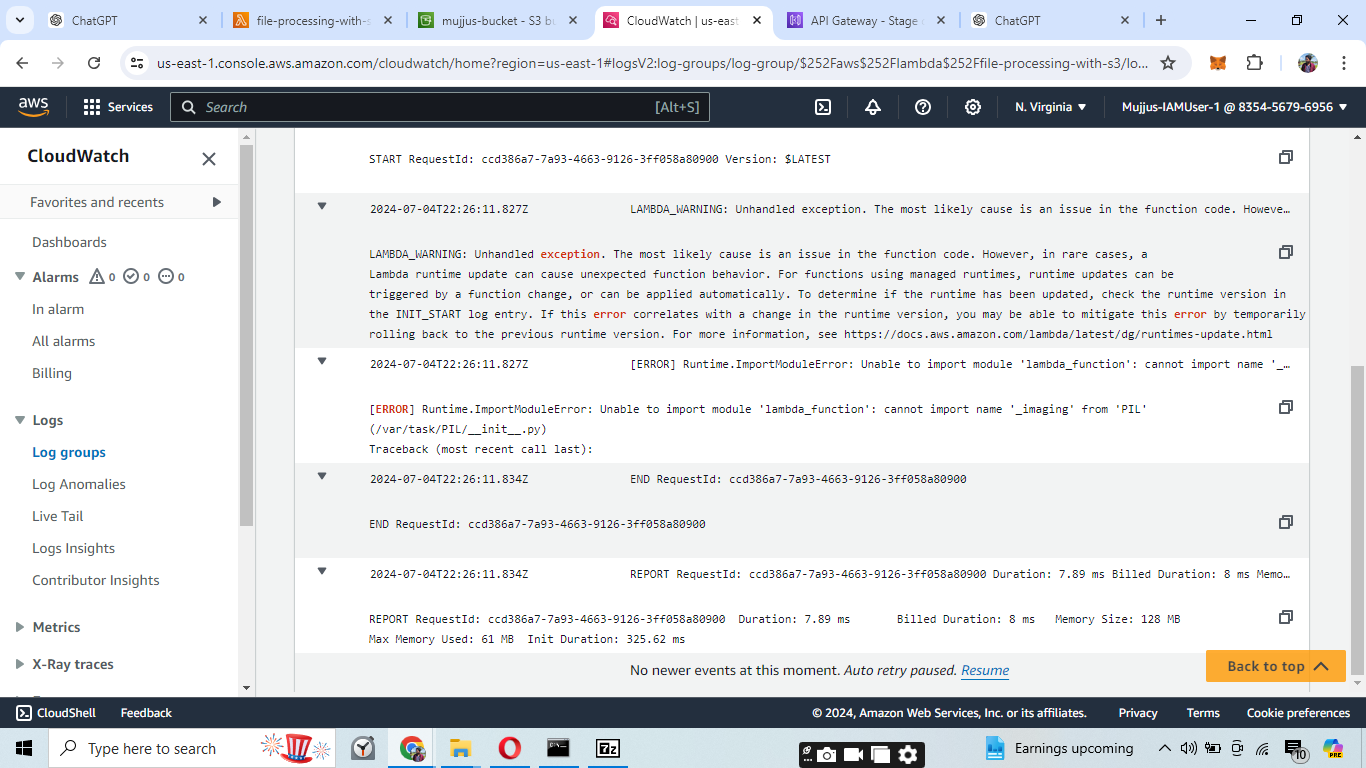
Now upload the Zip file.



After uploading the file we uploaded an image to the S3 bucket and we can see the lambda sends the run time metrics to cloud Watch.

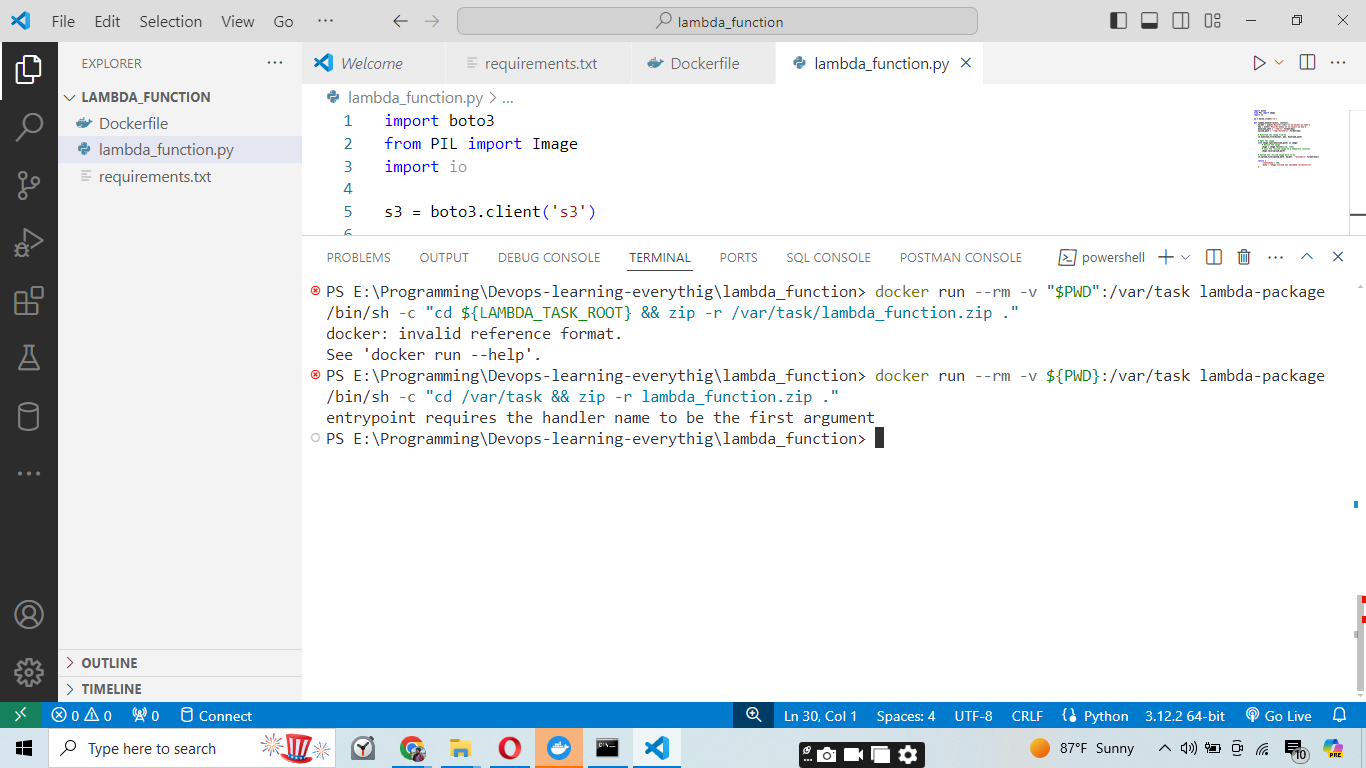


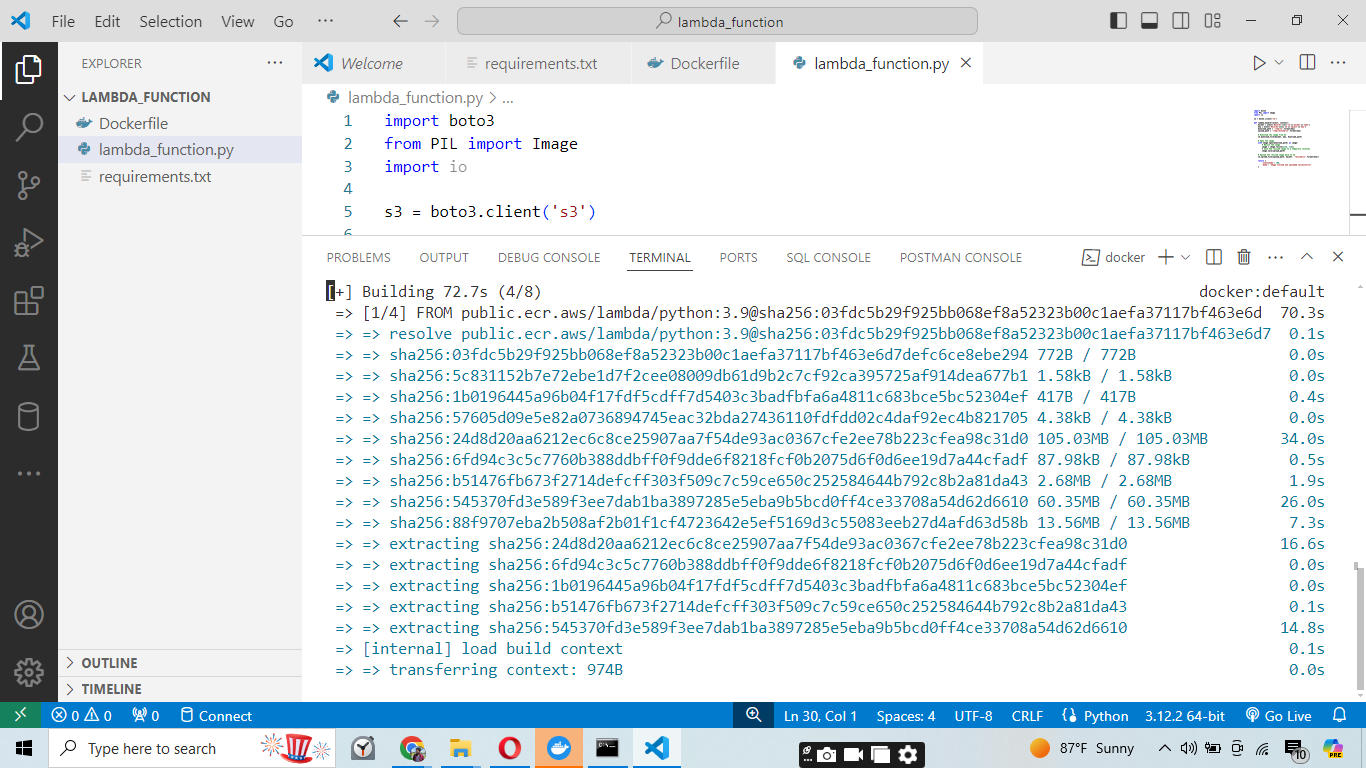
And we have Faced lot of issues as we can see in the CloudWatch.



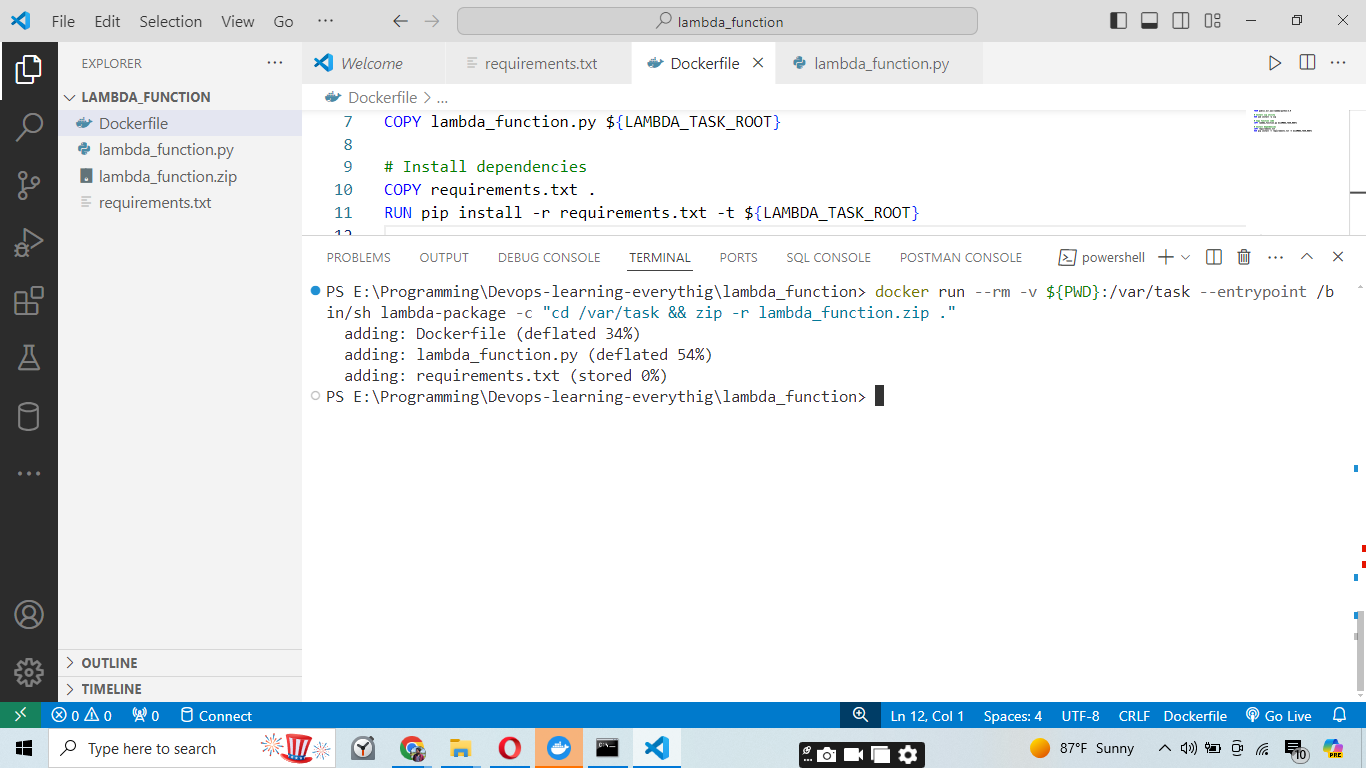
The error cannot import name '\_imaging' from 'PIL' typically occurs when there is an issue with the Pillow library not being packaged correctly for the AWS Lambda environment.

Here is how we can package the Lambda function with all its dependencies using Docker to ensure compatibility with the Lambda environment.

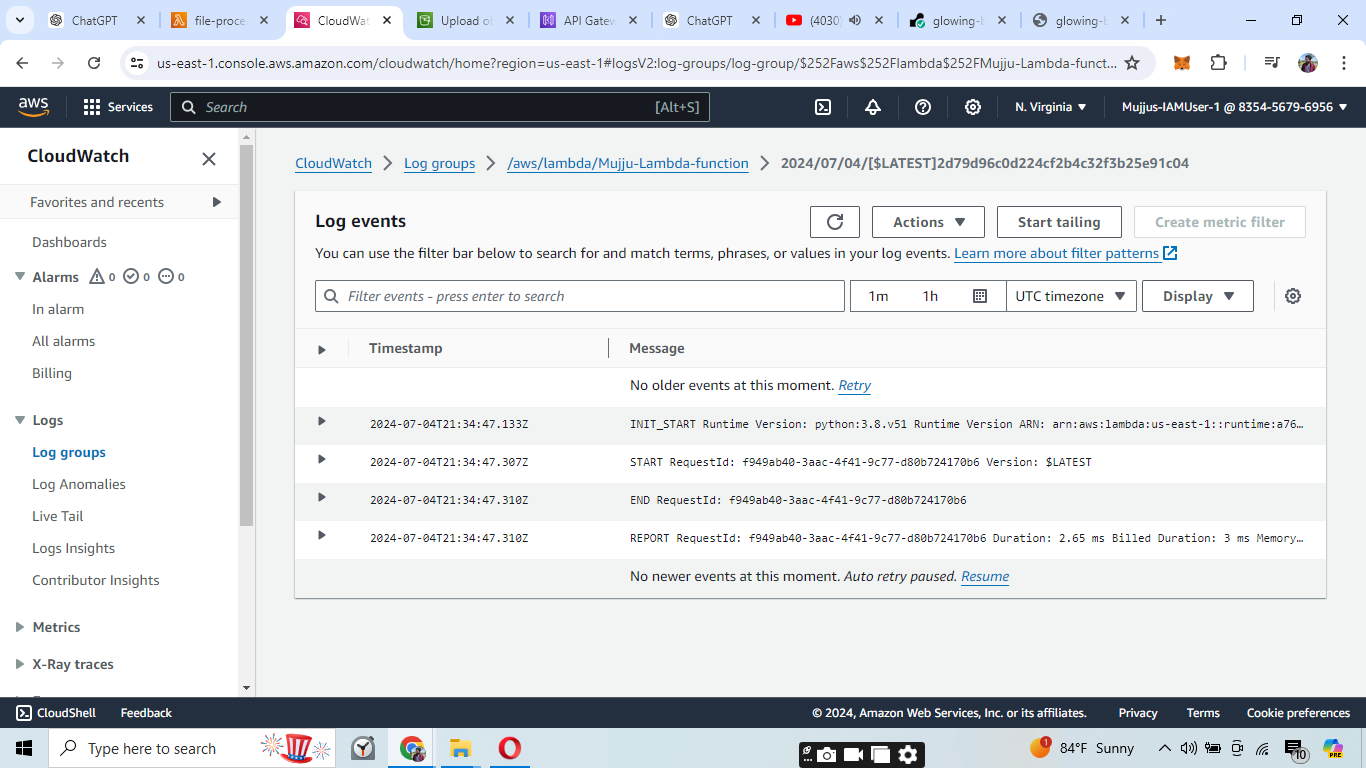




Now we can use Docker command to package the whole content into a zip file.



Now at last after uploading the Zip file to the Lambda Function and uploading the image to the S3 bucket it worked.



As we can see the logs in the image it worked.

**Conclusion**

Through this project, we successfully automated image processing using AWS Lambda and S3. The key challenges were related to packaging dependencies and ensuring correct permissions, which were resolved by manually packaging the Lambda function with a virtual environment and adjusting IAM roles.

**Problems faced:**

**Lambda Permissions Issues:**

Problem: Initially, the Lambda function lacked the necessary permissions to read from and write to the S3 bucket.

Solution: Updated the IAM role to include AmazonS3ReadOnlyAccess and a custom policy allowing read/write access to the specific S3 bucket.

**Missing Dependencies (Pillow Library):**

Problem: Encountered the error No module named 'PIL' because the Pillow library was not included in the Lambda deployment package.

Solution: Manually packaged the Lambda function with all dependencies using a virtual environment.

**Lambda Deployment Package Size:**

Problem: The deployment package exceeded the inline editing size limit in the AWS Lambda console.

Solution: Uploaded the deployment package via S3 to handle larger file sizes.

**Error with \_imaging Module in PIL:**

Problem: Encountered the error cannot import name '\_imaging' from 'PIL' due to missing binary dependencies in the PIL module.

Solution: Ensured that the virtual environment correctly installed all necessary dependencies and included them in the deployment package.

**Docker Packaging Issues:**

Problem: The initial attempt to package the Lambda function with Docker failed due to various issues, including missing zip utility and incorrect path references.

Solution: Corrected the Dockerfile and packaging commands to ensure all dependencies were included.

**Log Monitoring and Troubleshooting:**

Problem: Difficulties in identifying issues without detailed logs.

Solution: Utilized CloudWatch logs to monitor the Lambda function’s execution and troubleshoot errors

effectively.

**Learnings**

**Importance of Proper IAM Permissions:**

Ensuring the Lambda function has the correct permissions to access required AWS services is crucial for its successful execution.

**Packaging Dependencies Correctly:**

Dependencies not included in the default Lambda environment must be packaged manually to avoid import errors.

**Handling Large Deployment Packages:**

For deployment packages exceeding the inline editing limit, using S3 to upload the package is a viable solution.

**Using Virtual Environments for Dependency Management:**

Virtual environments help manage and package dependencies accurately, ensuring all necessary libraries are included.

**Effective Use of Docker for Packaging:**

Docker can be a powerful tool for packaging Lambda functions, but it requires careful configuration to ensure all dependencies are correctly installed.

**Monitoring and Troubleshooting with CloudWatch:**

CloudWatch logs are invaluable for diagnosing issues and verifying the successful execution of Lambda functions.

**Understanding AWS Service Interactions:**

Familiarity with how different AWS services (Lambda, S3, IAM, CloudWatch) interact is essential for creating robust cloud-based solutions.

**Documentation and Process Tracking:**

Keeping detailed documentation of each step and challenge helps in troubleshooting and provides a reference for future projects.

**This project demonstrates how to leverage AWS services to create a scalable and automated image processing pipeline.**