Project – 1

Serverless Image Processing

Table of contents

1. **Introduction**
   1.1. Setup Checklist for Mini Project
   1.2. Instructions
2. **Problem Statement**
   2.1. Objective
   2.2. Abstract of the Project
   2.3. Technology Used
3. **Implementation**
   3.1. Summary of the Functionality to be Built
   3.2. Guidelines on the Functionality to be Built
4. **Reports to be Built**

1 Introduction

This document outlines a mini-project for implementing an AWS serverless image processing system. The project utilizes AWS services such as S3, Lambda, and IAM. The aim is to automatically resize and optimize images uploaded to an S3 bucket using a Lambda function integrated with the Sharp image processing library.

Serverless Image Processing with AWS Lambda and S3:

AWS S3 (Simple Storage Service) is a cloud data storage service. It is one of the most popular services of AWS. It has high scalability, availability, security and is cost effective. S3 has different storage tiers depending on the use case. Some common use cases of AWS S3 are:

Storage: It can be used for storing large amounts of data.

Backup and Archive: S3 has different storage tiers based on how frequent the data is accessed which can be used to backup critical data at low costs.

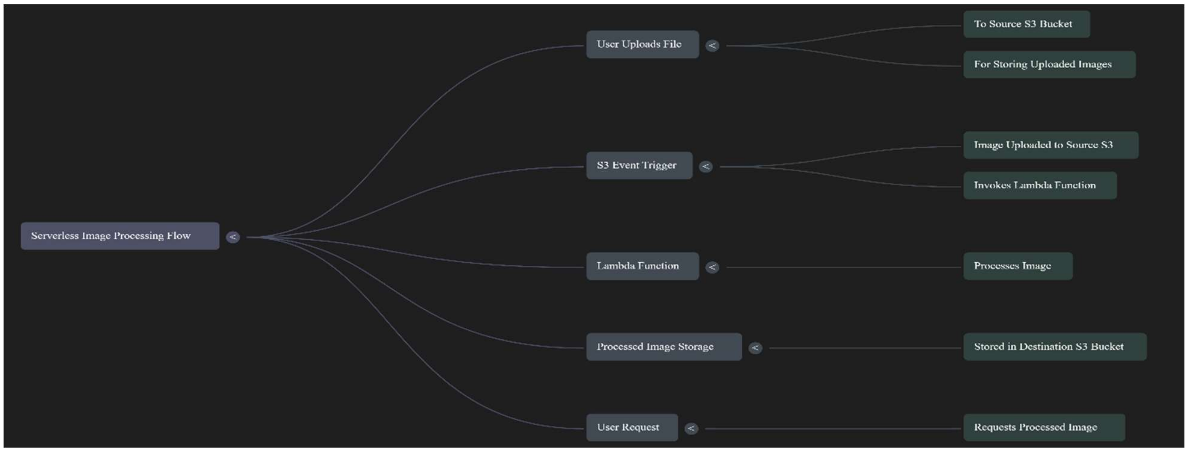
Static website: S3 offers static website hosting through HTML files stored in S3.

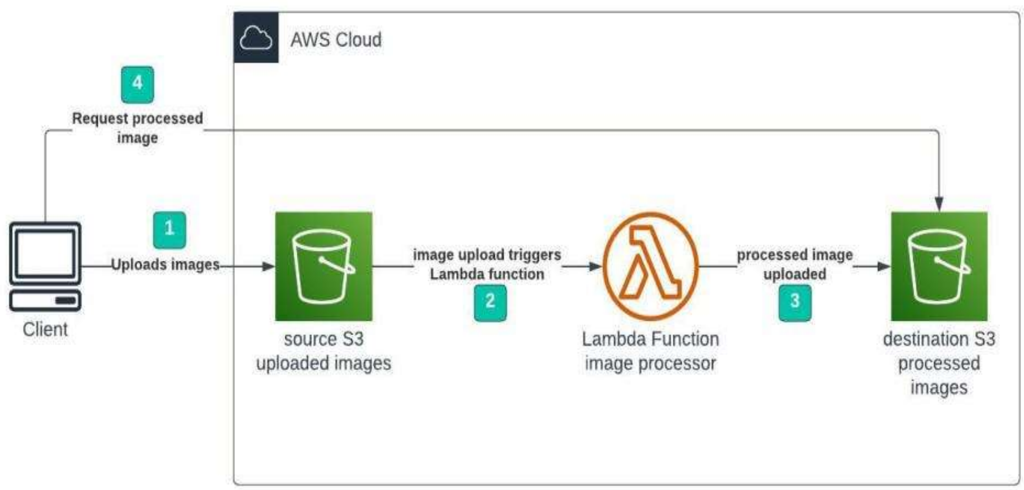
Data lakes and big data analytics: Companies can use AWS S3 as a data lake and then run analytics on it for getting business insights and take critical decisions.

AWS Lambda:

AWS Lambda is a serverless, event-driven compute service that lets you run code for virtually any type of application or backend service without provisioning or managing servers. Lambda functions run on demand i.e. they execute only when needed and you pay only for what you compute. Lambda is well integrated with may other AWS services. It supports a wide variety of programming languages.

Serverless Image Processing Flow

- User uploads a file to the source S3 bucket (which is used for storing uploaded images).
- When the image is uploaded to a source S3 bucket, it triggers an event which invokes the Lambda function. The lambda function processes the image.
- Processed image is stored in the destination S3 bucket.
- The processed image is requested by the user.
- 



Serverless Image Processor

4.1.

1.1 Setup Checklist for Mini Project

- **Hardware:**
  - Intel i3 or higher
  - 4 GB RAM or more
- **Software:**
  - AWS CloudShell (or terminal with AWS CLI)
  - Node.js (18.x runtime)
  - VS Code (optional)
  - AWS Console Access

1.2 Instructions

- Create all resources in the AWS Console or via CloudShell
- Follow IAM best practices
- Use CloudWatch for debugging Lambda logs
- ZIP structure for Lambda should be correct

2.1 Objective

To develop an image processing pipeline that automatically triggers upon file upload to an S3 bucket and resizes the image to predefined widths.

2.2 Abstract of the Project

The system will consist of:

- A **source S3 bucket** to upload original images
- A **Lambda function** that resizes the image
- An optional **destination S3 bucket** or folder where processed images are stored
- **IAM roles and permissions** to securely allow S3-Lambda interaction

This serverless solution will run without needing any EC2 or server provisioning, making it cost-effective and scalable.

2.3 Technology Used:

- AWS S3
- AWS Lambda
- Node.js (Sharp Library)
- IAM
- AWS CloudShell or CLI
- CloudWatch Logs

3.1 Summary of the Functionality to be Built:

- When an image is uploaded to the S3 source bucket
- An event triggers the Lambda function
- The Lambda function fetches the image, resizes it using sharp, and stores it back in another folder or bucket

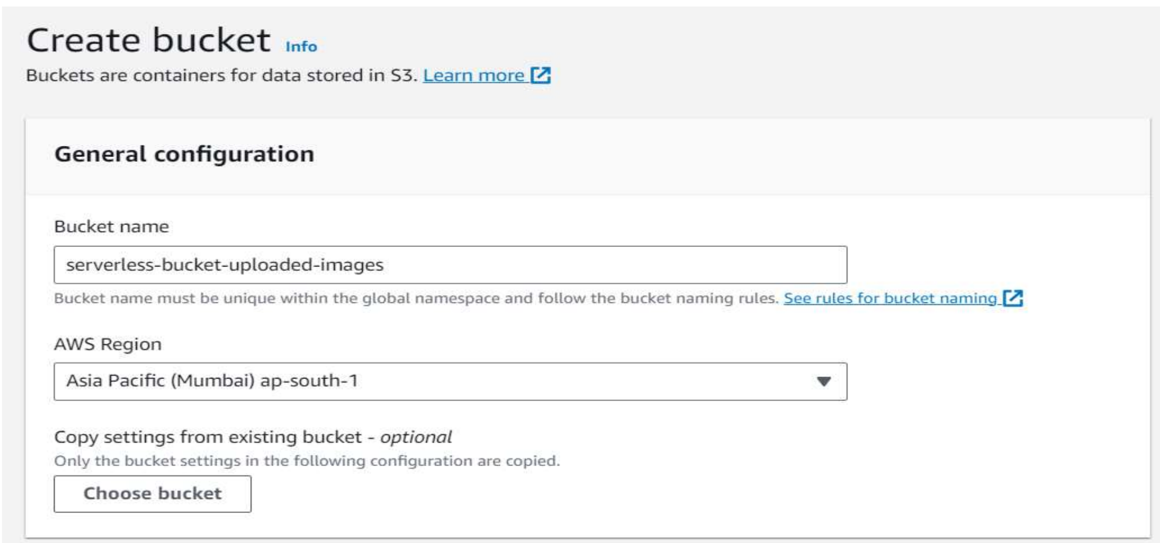
3.2 Guidelines on the Functionality to be Built:

Step 1 - Creating S3 buckets

We will use two S3 buckets:

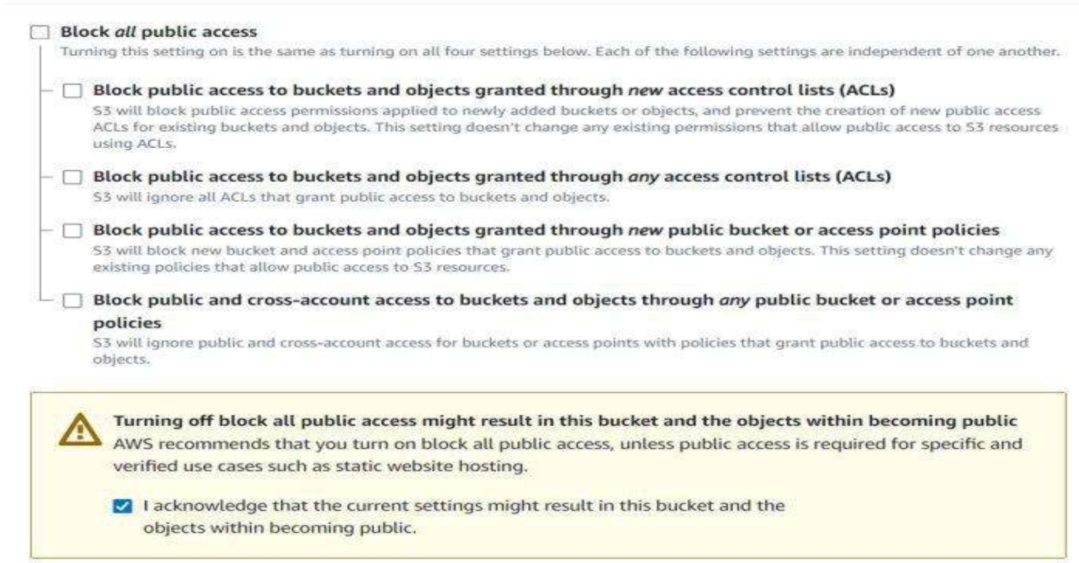
1. **source Bucket:** For storing uploaded images.
2. **destination Bucket:** For storing processed images.

Go to S3 console and click Create bucket. Enter bucket name as 'serverless-bucket-uploaded-images'. Choose any AWS region as 'ap-south-1'.



Step 2 - Configuring S3 bucket policy

In 'Block Public Access settings for this bucket' section disable "block all public access". You will get a warning that the bucket and its objects might become public. Agree to the warning**. (<u>Note:</u> we are making this bucket public only for this project, it is not recommended to make an S3 bucket public if not needed)**.



Leave all other settings as default and create bucket. Similarly, create another bucket named 'serverless-bucket-processed-images' with the same region. This bucket will be used to store the processed images. Although we enabled public access while creating the buckets, we still need to attach a bucket policy to access the objects stored in it. (Policies in AWS are JSON documents which defines the permissions for performing actions on a certain resource.)

Go to your source bucket and then click on Permissions tab. Scroll to Bucket Policy tab. Click Edit. You will be redirected to the policy editor. Click on policy generator.

Enter the following settings:

- Type of policy: [S3 Bucket Policy]
- Effect: Allow
- Principal: *
- Actions: GetObject
- Amazon Resource Name (ARN): arn:aws:s3:::SOURCE_BUCKET_NAME/*

SOURCE_BUCKET_NAME is the name of the bucket used for uploading the images.

Click Add Statement and then generate policy. Copy the JSON object.

Paste it in the policy editor and then save changes.

Follow same steps to attach a policy to the processed images S3 bucket. The policy settings for destination bucket are:

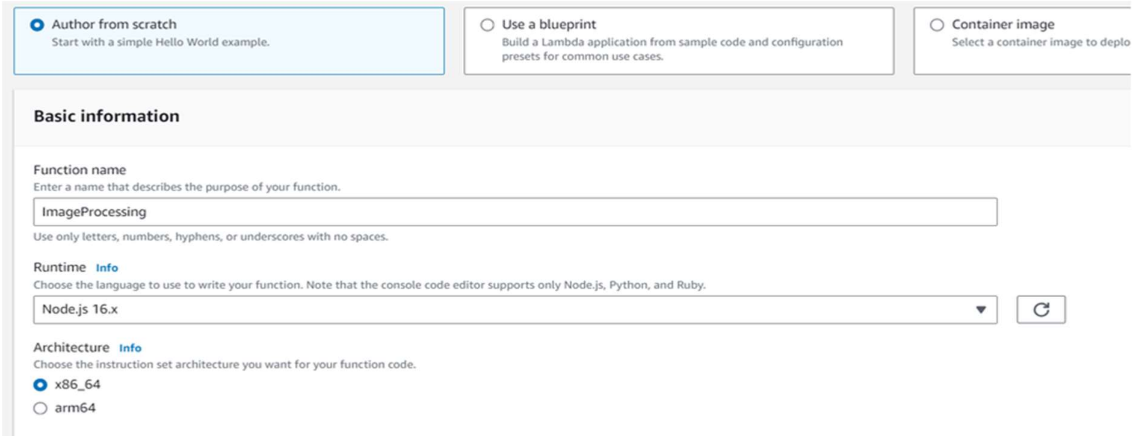
- Type of policy: S3 Bucket Policy
- Effect:Allow
- Principal: *
- Actions: GetObject, PutObject, and PutObjectAcl
- Amazon Resource Name (ARN): arn:aws:s3:::DESTINATION_BUCKET_NAME/*

DESTINATION_BUCKET_NAME is the name of the bucket used for storing processed images.

```
{
  "Version": "2012-10-17",
  "Id": "Policy1695244199034",
  "Statement": [
    {
      "Sid": "Stmt1695244197617",
      "Effect": "Allow",
      "Principal": "*",
      "Action": [
        "s3:GetObject",
        "s3:PutObject",
        "s3:PutObjectAcl"
      ],
      "Resource": "arn:aws:s3:::serverless-bucket-processed-images/*"
    }
  ]
}
```

Step 3 - Creating Lambda function

Go to AWS Lambda console. Navigate to Functions section. Click Create Function and name it "ImageProcessing". Select runtime as "NodeJS 16.x" and architecture as "x86_64". Leave all other settings as default. Create the function.



In the code editor on the Lambda function page paste the following code. This function is executed whenenver an image is uploaded to our source S3 bucket and creates two images (thumbnail (300x300) and coverphoto(800x800)) and stores it in the destination S3 bucket. **(Note: The value of processedImageBucket in the code should be set to the name of the destination bucket).**

```
const sharp = require("sharp");

const path = require("path");

const AWS = require("aws-sdk");


// Set the REGION

AWS.config.update({

  region: "ap-south-1",

});

const s3 = new AWS.S3();

const processedImageBucket = "serverless-bucket-processed-images";


// This Lambda function is attached to an S3 bucket. When any object is added in the S3
```

```
// bucket this handler will be called. When an image file is added in the S3 bucket, this function
// creates a square thumbnail of 300px x 300px size and it also creates a cover photo of
// 800px x 800px size. It then stores the thumbnail and coverphotos back to another S3 bucket
// at the same location as the original image file.
exports.handler = async (event, context, callback) => {
  console.log("An object was added to S3 bucket", JSON.stringify(event));
  let records = event.Records;
  // Each record represents one object in S3. There can be multiple
  // objects added to our bucket at a time. So multiple records can be there
  // How many records do we have? Each record represent one object in S3
  let size = records.length;

  for (let index = 0; index < size; index++) {
    let record = records[index];
    console.log("Record: ", record);
    // Extract the file name, path and extension
    let fileName = path.parse(record.s3.object.key).name;
    let filePath = path.parse(record.s3.object.key).dir;
    let fileExt = path.parse(record.s3.object.key).ext;

    console.log("filePath:" + filePath + ", fileName:" + fileName + ", fileExt:" + fileExt);
```

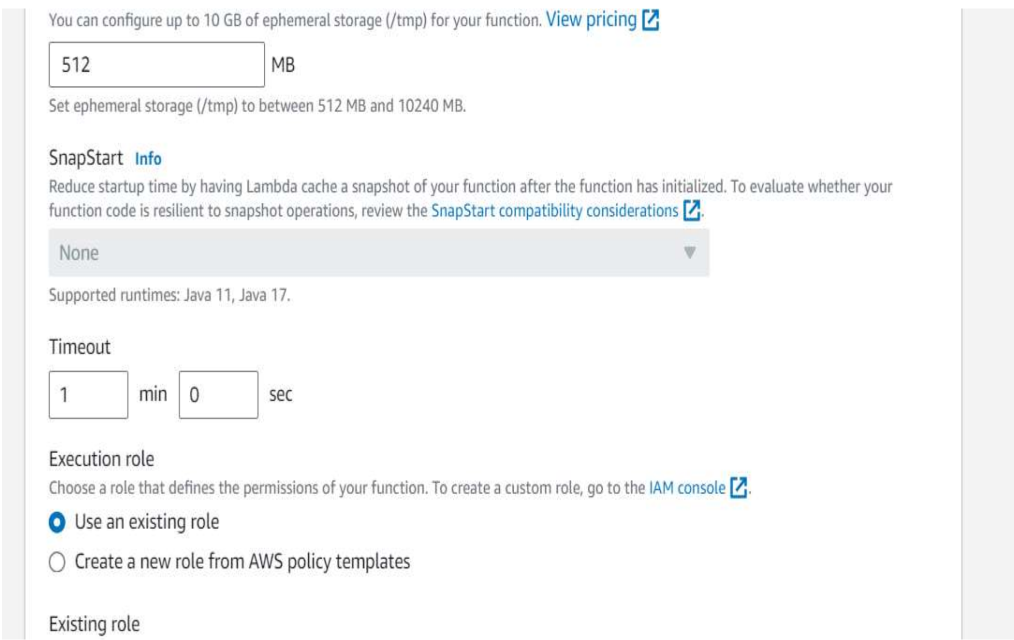
```
    // Read the image object that was added to the S3 bucket
    let imageObjectParam = {
      Bucket: record.s3.bucket.name,
      Key: record.s3.object.key,
    };

    let imageObject = await s3.getObject(imageObjectParam).promise();
    // Use sharp to create a 300px x 300px thumbnail
    // withMetadata() keeps the header info so rendering engine can read
    // orientation properly.
    let resized_thumbnail = await sharp(imageObject.Body)
      .resize({
        width: 300,
        height: 300,
        fit: sharp.fit.cover,
      })
      .withMetadata()
      .toBuffer();
    console.log("thumbnail image created");

    // Use sharp to create a 800px x 800px coverphoto
    let resized_coverphoto = await sharp(imageObject.Body)
      .resize({
```

```
      width: 800,
      height: 800,
      fit: sharp.fit.cover,
    })
    .withMetadata()
    .toBuffer();
  console.log("coverphoto image created");

  // The processed images are written to serverless-image-processing-bucket.
  let thumbnailImageParam = {
    Body: resized_thumbnail,
    Bucket: processedImageBucket,
    Key: fileName + "_thumbnail" + fileExt,
    CacheControl: "max-age=3600",
    ContentType: "image/" + fileExt.substring(1),
  };
  let result1 = await s3.putObject(thumbnailImageParam).promise();
  console.log("thumbnail image uploaded:" + JSON.stringify(result1));

  let coverphotoImageParam = {
    Body: resized_coverphoto,
    Bucket: processedImageBucket,
    Key: fileName + "_coverphoto" + fileExt,
    CacheControl: "max-age=3600",
    ContentType: "image/" + fileExt.substring(1),
```

```
    };
    let result2 = await s3.putObject(coverphotoImageParam).promise();
    console.log("coverphoto image uploaded:" + JSON.stringify(result2));
  }
};
```

Save the code and click Deploy to deploy the changes.

Go to Configuration tab and Edit the general configuration. There set the timeout to 1 min (timeout is the maximum time for which a Lambda function will run after which it stops running). We need to increase the timeout because the image can take time to process. Click on Save changes.



Step 4 - Creating Lambda layer and attaching it to Lambda function

Layers in Lambda is used to add dependencies to a Lambda Function. Lambda Layers reduces the code size of Lambda functions as we do not need to upload the dependencies with the function. It also useful for code reusability as we can reuse the layer with multiple functions if they require the same dependencies.

Inside CloudShell:

mkdir sharp-layer && cd sharp-layer

mkdir nodejs && cd nodejs

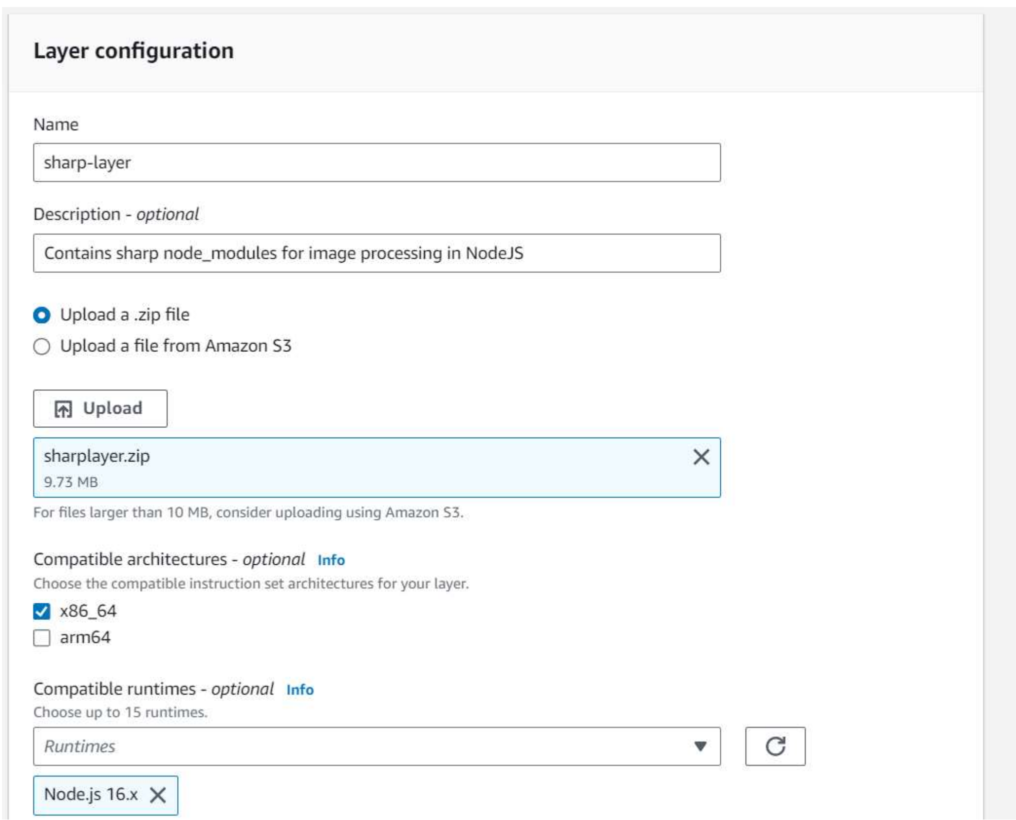
npm init -y

npm install sharp

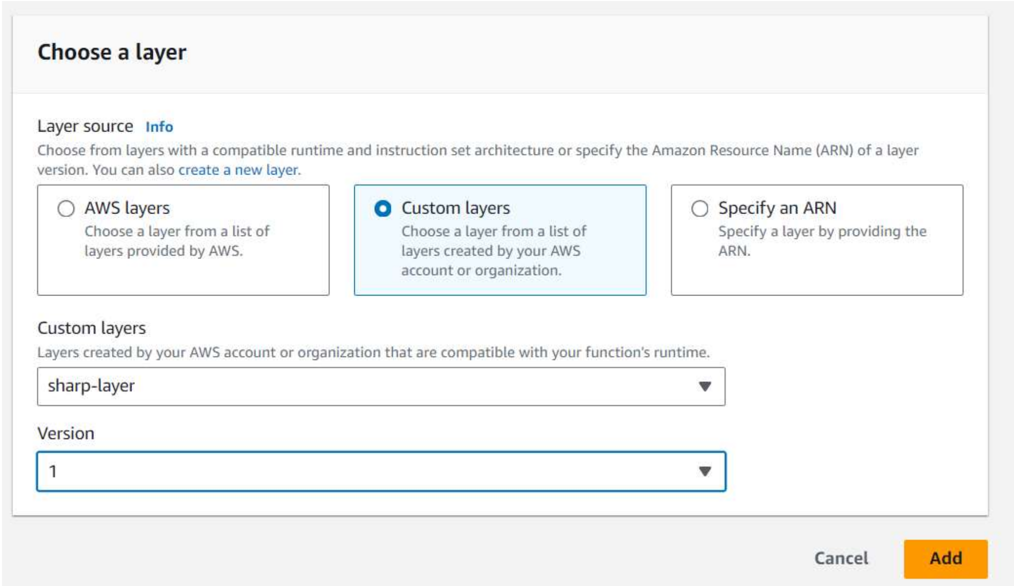
cd ..

zip -r sharp-layer.zip nodejs

Go to Layers in Lambda console. Click Create layer. Name it "sharp-layer". Upload your nodejs "sharplayer.zip" file here. Select x86_64 architecture. Select NodeJS 16.x in compatible runtimes. Click on Create Layer.



Now go to your lambda function page. In Layers section click on Add layer button. Select Custom Layer. Choose "sharp-layer". Select version 1

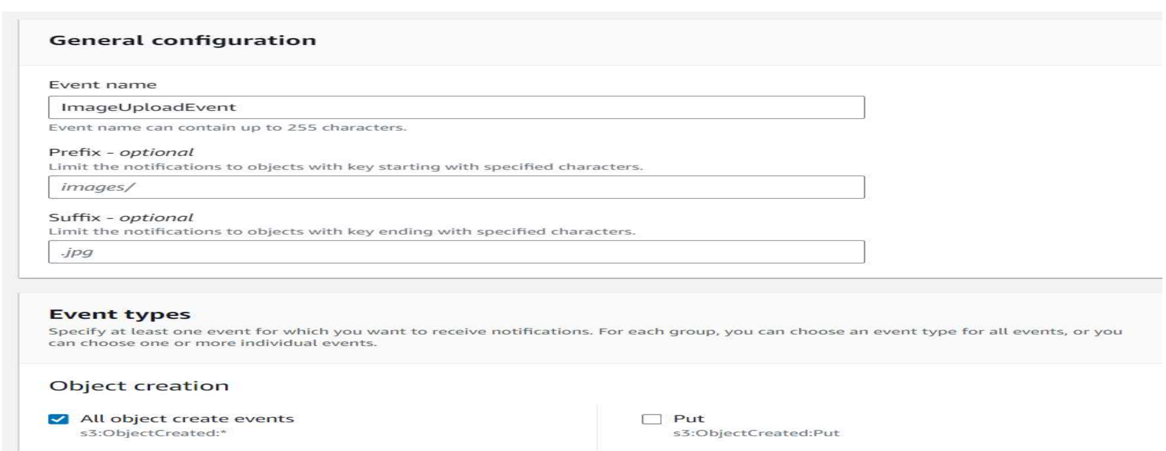


Step 5 - Creating S3 trigger

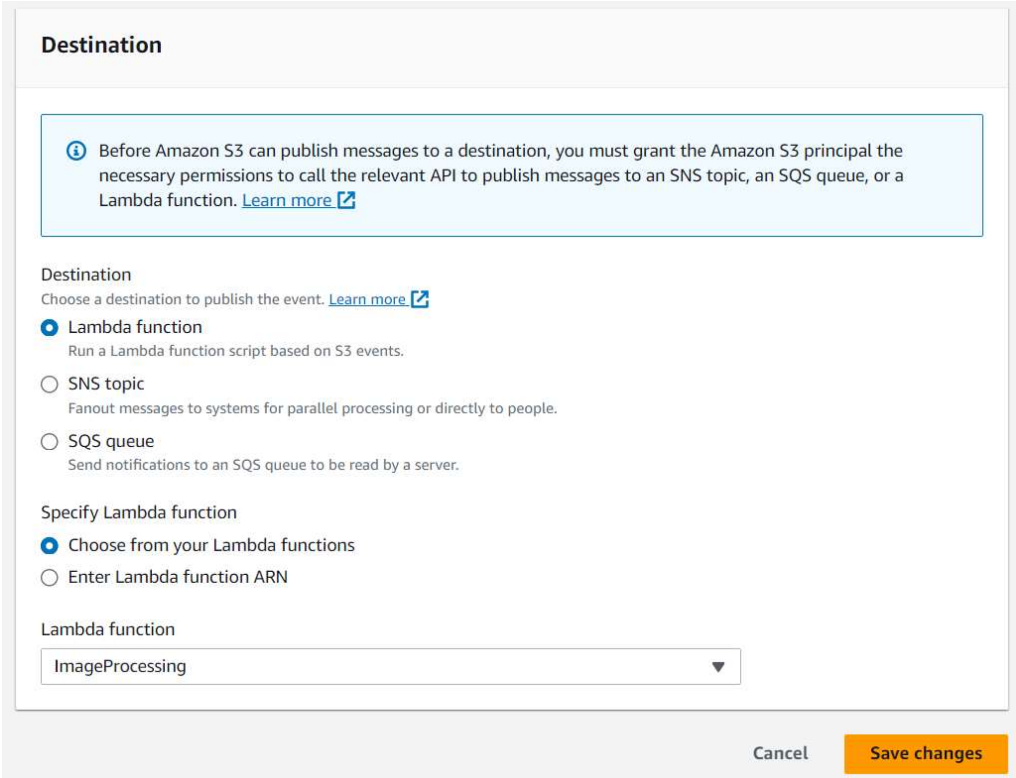
Now we need our Lambda function to know when an image is uploaded to the source bucket. We can do this by adding an event to the source S3 bucket and configure it to get triggered when an image is uploaded to the bucket which in turn invokes the Lambda function.

Go to S3 console. Select the source bucket ("serverless-bucket-uploaded-images"). Go to the Properties tab. Navigate to "Event Notifications". Click "Create Event Notifications".

Give an appropriate name to the event. Check the "All object create events



Navigate to the "Destination" and select your lambda function. Save changes.



Step 6 - Testing the application

Upload an image file to source S3 bucket ("serverless-bucket-uploaded-images"). Wait for few seconds and check the destination bucket ("serverless-bucket-processed-images"). There you will see two images (thumbnail and coverphoto).

Congratulations, you just built a serverless Image processing application.

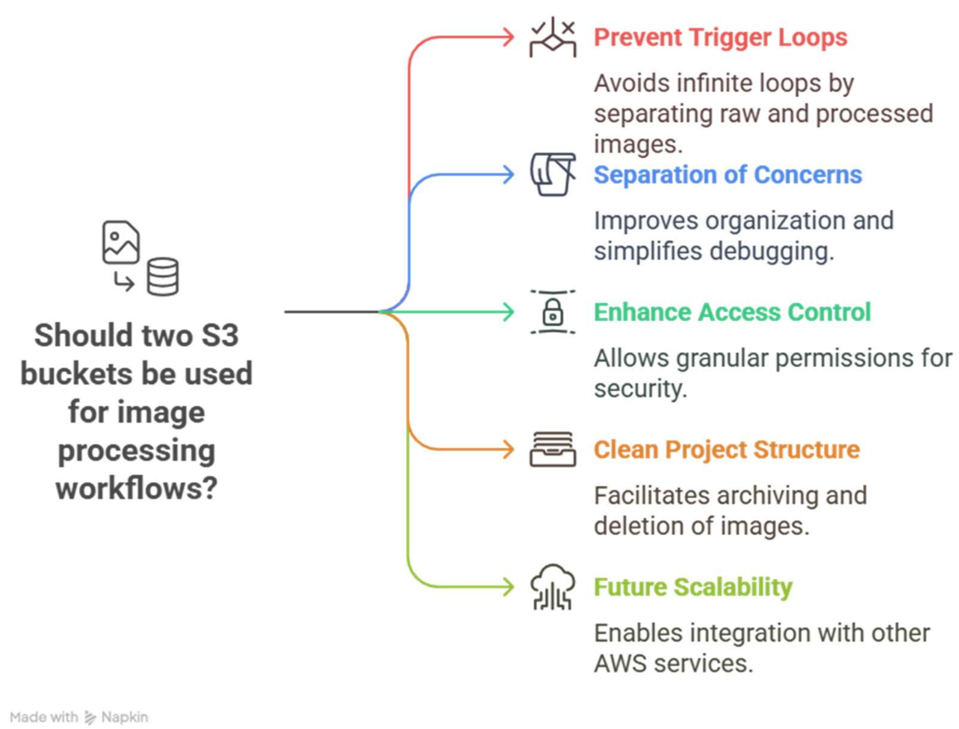
- Upload an image to source-image-bucket/uploads/
- Check the resized image in resized/ folder

4 Reports to be Built

- CloudWatch Logs: Verify image process logs
- S3: Check for resized image output
- Lambda Monitoring: Success/failure count

. Why Use Two S3 Buckets

- **Avoid Trigger Loops**: Writing resized images to the same source bucket might re-trigger the Lambda unintentionally. Using a separate destination bucket or folder prevents this loop.
- **Separation of Concerns**: One bucket handles raw uploads, and another stores final, processed images.
- **Better Access Control**: Different IAM permissions can be applied to source and destination buckets, improving security.
- **Clean Project Structure**: Makes it easy to manage, archive, or delete processed vs original files.
- **Future Scalability**: Allows integration with CDNs (like CloudFront) for processed bucket or adding analytics to uploads.



Note: This project simulates a real-world use case where serverless architecture is used for automatic image resizing on upload — scalable, cost-effective, and maintenance-free.