Final Project

DV1566 HT25 lp2 Introduction to Cloud Computing - Campus

December 17, 2024

Team Number	Team 14
Title	**Final project**
Supervisor(s)	Emiliano Casalicchio and Shahrooz Abghari

Student 1	Name	Yalamati Nikhil Souri
	Email	niya25@student.bth.se
Student 2	Name	Ajay Kumar Reddy Bindela
	Email	ajbi25@student.bth.se
Student 3	Name	Shaik Mahammed Sameen
	Email	mass25@student.bth.se

1 Introduction

This project demonstrates a scalable, serverless image-processing platform implemented using Amazon Web Services (AWS). The original goal was to deploy an image-processing application on AWS Lambda, invoke it via Amazon API Gateway and Amazon S3 events, and show the deployment scale by analysing performance metrics collected with Amazon CloudWatch.

The solution integrates Amazon S3, AWS Lambda, Amazon API Gateway, Amazon Rekognition, and Amazon CloudWatch into an end-to-end workflow. Images can be processed in two main ways:

- **Label detection:** Amazon Rekognition is used to identify objects and scenes in the image, returning labels and confidence scores.
- **Pixel-level transformations:** The Lambda function uses the Python Pillow library to perform grayscale conversion, dynamic resizing, and thumbnail generation before or in combination with label detection.

In addition to the backend, a lightweight browser-based user interface was implemented. The UI allows users to upload an image directly from their device, select an operation (labels only, grayscale, resize, or thumbnail), and immediately visualise the processed result along with the detected labels. Custom CloudWatch metrics (with dimensions for different operation types) are used to evaluate performance and basic scalability.

2 System Architecture

2.1 Overview of AWS Services

The solution uses the following AWS services:

- **Amazon S3**: Two buckets are used. The input bucket (`dv1566-image-input-nikhil`) preserves original files uploaded via S3 or via the HTTP API. The output bucket (`dv1566-image-output-r` contains the processed images under prefixes such as `processed/original/`, `processed/grayscale/`, `processed/resize/`, and `processed/thumbnail/`.
- **AWS Lambda**: The function `dv1566-image-processor` contains the core application logic. It retrieves images from S3, applies the requested transformation using Pillow, optionally invokes Amazon Rekognition, stores the processed image back into S3, and reports custom metrics to CloudWatch.
- **Amazon API Gateway (HTTP API)**: Provides an HTTP endpoint (`POST /process`) that external clients (including the web UI) can call to trigger image processing.
- **Amazon Rekognition**: Performs image analysis and provides labels with confidence scores when requested.
- **Amazon CloudWatch**: Stores custom application metrics and AWS Lambda metrics (invocations, duration, errors, concurrency). A dashboard visualises the performance and scalability of the application.

2.2 Enhanced Capabilities

Compared to a basic label-detection pipeline, the architecture was extended with the following capabilities:

- **Multiple operations per image**: The client can request one of several operations: labels only, grayscale, resize, or thumbnail.
- **Browser-based uploads**: The frontend sends images as base64-encoded data, so users can process local images without manually uploading them to S3 first.
- **Structured output layout**: Processed images are saved in S3 under prefixes that reflect the operation applied.
- **Operation-aware metrics**: Custom CloudWatch metrics include a dimension called `Operation`, allowing processing time and throughput to be compared across different modes.

2.3 Architecture Diagram

Figure 1 illustrates the high-level architecture of the system. Images can arrive via direct S3 uploads or via the HTTP API. The Lambda function applies the requested transformations, invokes Rekognition when enabled, and stores the resulting images and metrics. The browser UI communicates with API Gateway and visualises the results.

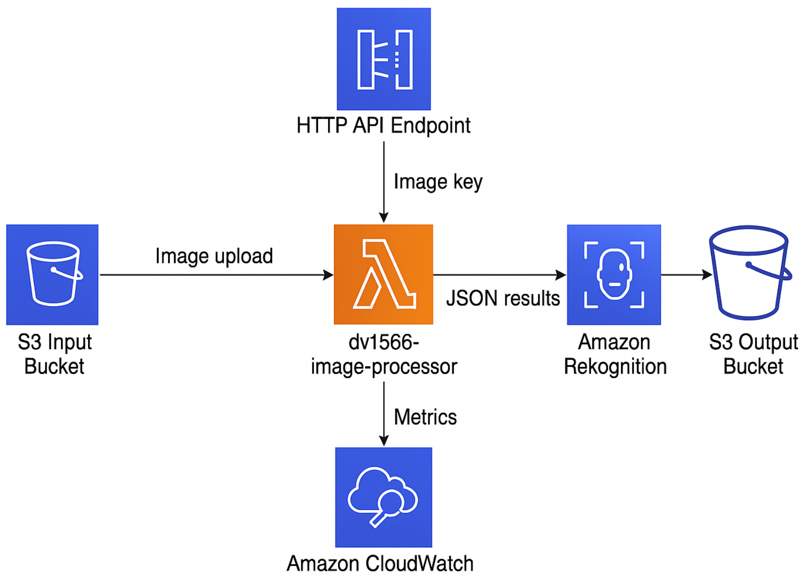


Figure 1: High-level architecture of the serverless image-processing system.

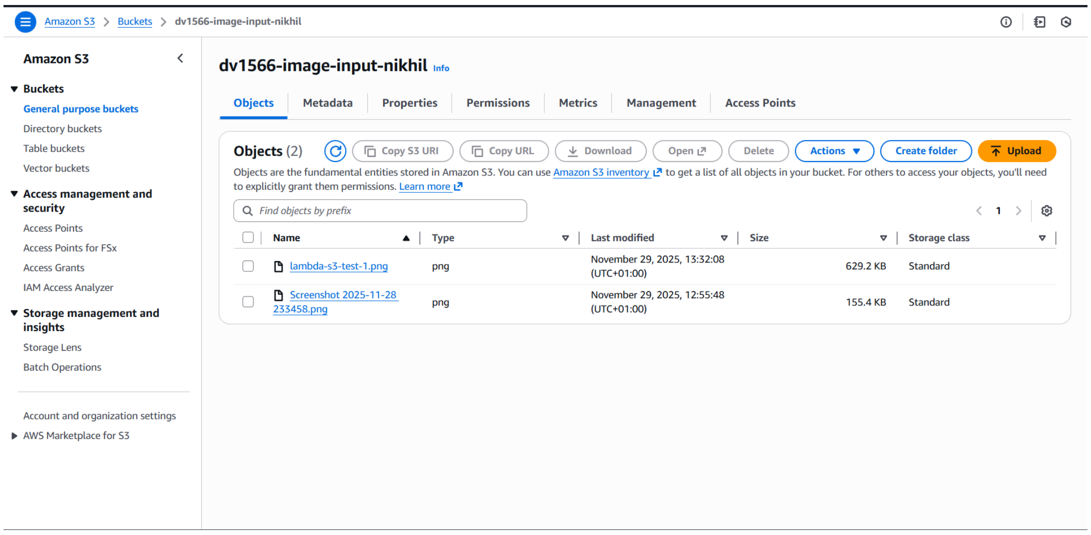
3 Implementation

3.1 S3 Buckets

Two S3 buckets were created in the `us-east-1` region:

- **Input bucket**: `dv1566-image-input-nikhil`, which stores original images uploaded via S3 or via the HTTP API (under a prefix such as `uploads/`).
- **Output bucket**: `dv1566-image-output-nikhil`, which stores processed images under operation-specific prefixes:
  - `processed/original/`
  - `processed/grayscale/`
  - `processed/resize/`
  - `processed/thumbnail/`

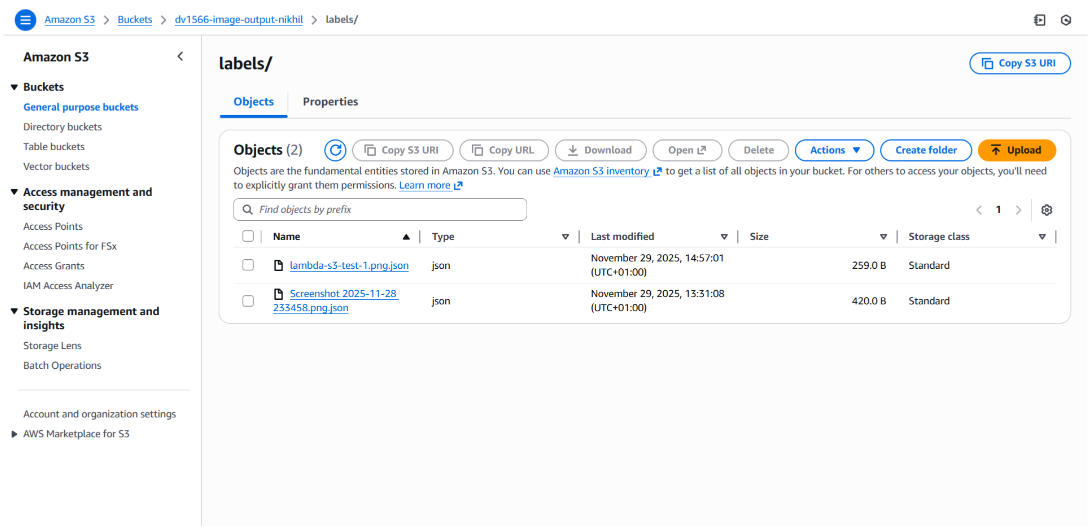
Figure 2 shows the input bucket contents with a set of test images. Figure 3 shows the output bucket with processed images organised by operation.



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

Name	Type	Last modified	Size	Storage class
lambda-s3-test-1.png	png	November 29, 2025, 13:32:08 (UTC+01:00)	629.2 KB	Standard
Screenshot 2025-11-28 233458.png	png	November 29, 2025, 12:55:48 (UTC+01:00)	155.4 KB	Standard

Figure 2: Input S3 bucket `dv1566-image-input-nikhil` with test images.



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

Name	Type	Last modified	Size	Storage class
lambda-s3-test-1.png.json	json	November 29, 2025, 14:57:01 (UTC+01:00)	259.0 B	Standard
Screenshot 2025-11-28 233458.png.json	json	November 29, 2025, 13:31:08 (UTC+01:00)	420.0 B	Standard

Figure 3: Output S3 bucket `dv1566-image-output-nikhil` with processed images under different prefixes.

3.2 Lambda Function

The `dv1566-image-processor` Lambda function is implemented in Python and assumes the IAM role `LabRole` provided by the AWS Academy Learner Lab. The environment variables `INPUT_BUCKET` and `OUTPUT_BUCKET` are used to store the names of the respective S3 buckets.

The extended Lambda function supports the following operations:

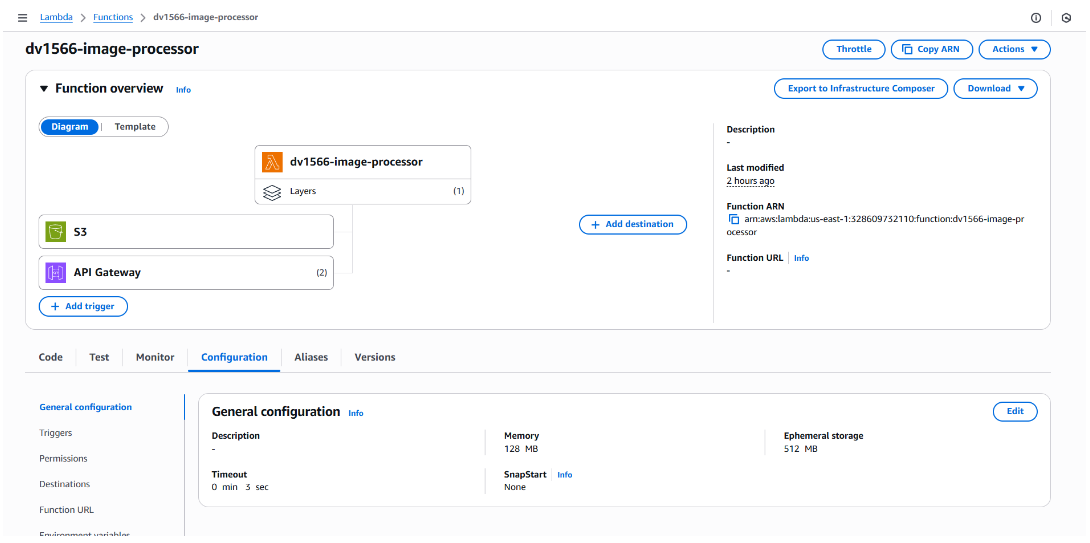
- **labels**: no pixel transformation; only Rekognition label detection.
- **grayscale**: converts the image to black-and-white.
- **resize**: resizes the image to the width and height provided by the client.
- **thumbnail**: generates a small thumbnail image (e.g., 256×256 pixels by default).

Internally, the function uses the Pillow library to perform the pixel-level operations:

1. Download the original image from S3.
2. Open it with Pillow and apply the requested transformation.
3. Store the processed image in the output bucket under `processed/<operation>/<filename>`.
4. Optionally call Amazon Rekognition on the processed image.
5. Measure processing time and count of labels.
6. Publish custom metrics to CloudWatch with an `Operation` dimension.
7. Return a JSON response containing metadata, labels, processing time, and the processed image as a base64 string for immediate display in the UI.

For S3 event triggers, the function currently defaults to the `labels` operation but still stores a copy of the image in `processed/original/` and publishes metrics.

Figure 4 shows the configuration of the Lambda function in the AWS console. Figure 5 shows a shorter excerpt of the extended code.



Description
-

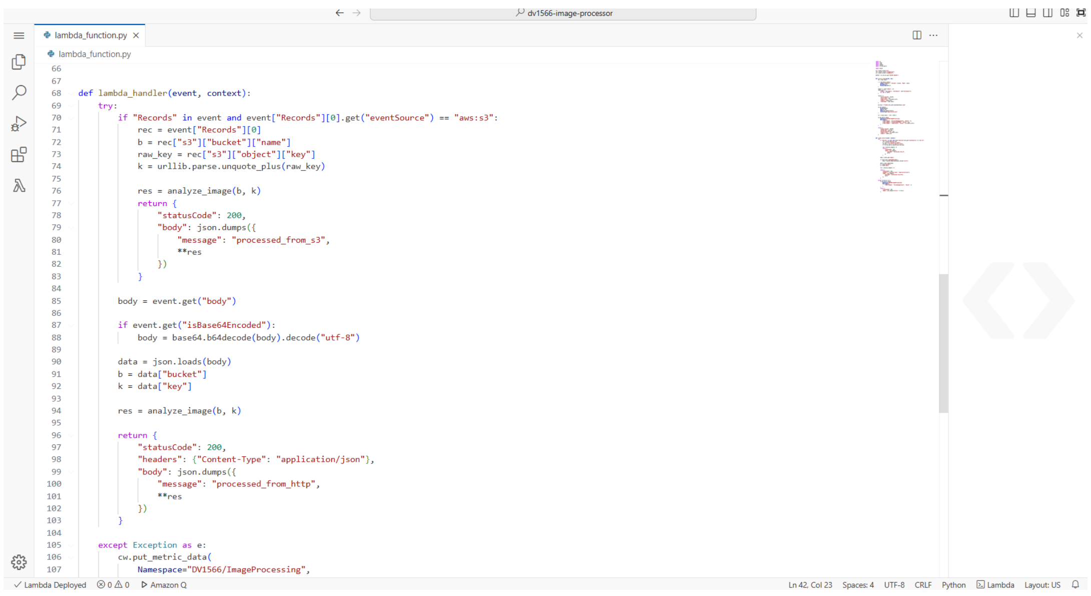
Last modified
2 hours ago

Function ARN
arn:aws:lambda:us-east-1:328609732110:function:dv1566-image-processor

Function URL
-

Description: -
Memory: 128 MB
Ephemeral storage: 512 MB
Timeout: 0 min 3 sec
SnapStart: None

Figure 4: Configuration of the Lambda function `dv1566-image-processor`.



```python


def lambda_handler(event, context):
    try:
        if "Records" in event and event["Records"][0].get("eventSource") == "aws:s3":
            rec = event["Records"][0]
            b = rec["s3"]["bucket"]["name"]
            raw_key = rec["s3"]["object"]["key"]
            k = urllib.parse.unquote_plus(raw_key)

            res = analyze_image(b, k)
            return {
                "statusCode": 200,
                "body": json.dumps({
                    "message": "processed_from_s3",
                    **res
                })
            }

        body = event.get("body")

        if event.get("isBase64Encoded"):
            body = base64.b64decode(body).decode("utf-8")

        data = json.loads(body)
        b = data["bucket"]
        k = data["key"]

        res = analyze_image(b, k)

        return {
            "statusCode": 200,
            "headers": {"Content-Type": "application/json"},
            "body": json.dumps({
                "message": "processed_from_http",
                **res
            })
        }

    except Exception as e:
        cw.put_metric_data(
            Namespace="DV1566/ImageProcessing",
```

Figure 5: Excerpt of the Lambda function source code in the AWS console.

3.3 API Gateway HTTP Endpoint

To allow external clients to trigger image processing, an HTTP API was created using Amazon API Gateway. The API consists of a single route:

- `POST /process`: accepts a JSON body describing the input and the desired operation.

The API supports two input modes:

1. **S3 mode** (legacy): the client sends:

```
{
  "bucket": "dv1566-image-input-nikhil",
  "key": "<image-filename>",
  "operation": "labels",
  "runRekognition": true
}
```

2. **Browser mode**: the client sends a base64-encoded image:

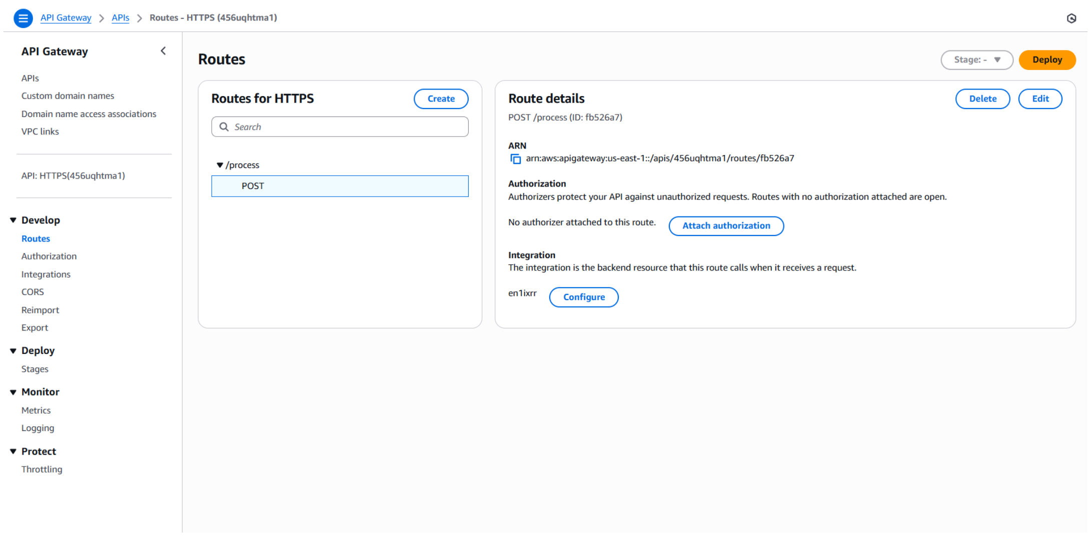
```
{
  "imageBase64": "<base64-encoded image>",
  "operation": "grayscale",
  "width": 800,
  "height": 600,
  "runRekognition": true
}
```

In the second case, the Lambda function decodes the base64 image, stores it into the input bucket under a generated key (e.g., `uploads/<timestamp>.png`), and then processes it.

The route is linked to the Lambda function `dv1566-image-processor`, and the API is available in the default stage. The complete URL of the endpoint is:

```
https://456uqhtma1.execute-api.us-east-1.amazonaws.com/process
```

Figure 6 shows the API Gateway configuration.



Routes

Routes for HTTPS

/process
POST

Route details

POST /process (ID: fb526a7)

ARN
arn:aws:apigateway:us-east-1::/apis/456uqhtma1/routes/fb526a7

Authorization
Authorizers protect your API against unauthorized requests. Routes with no authorization attached are open.

No authorizer attached to this route.

Integration
The integration is the backend resource that this route calls when it receives a request.

en1ixrr

Figure 6: API Gateway HTTP API route `POST /process` integrated with the Lambda function.

3.4 CloudWatch Metrics and Dashboard

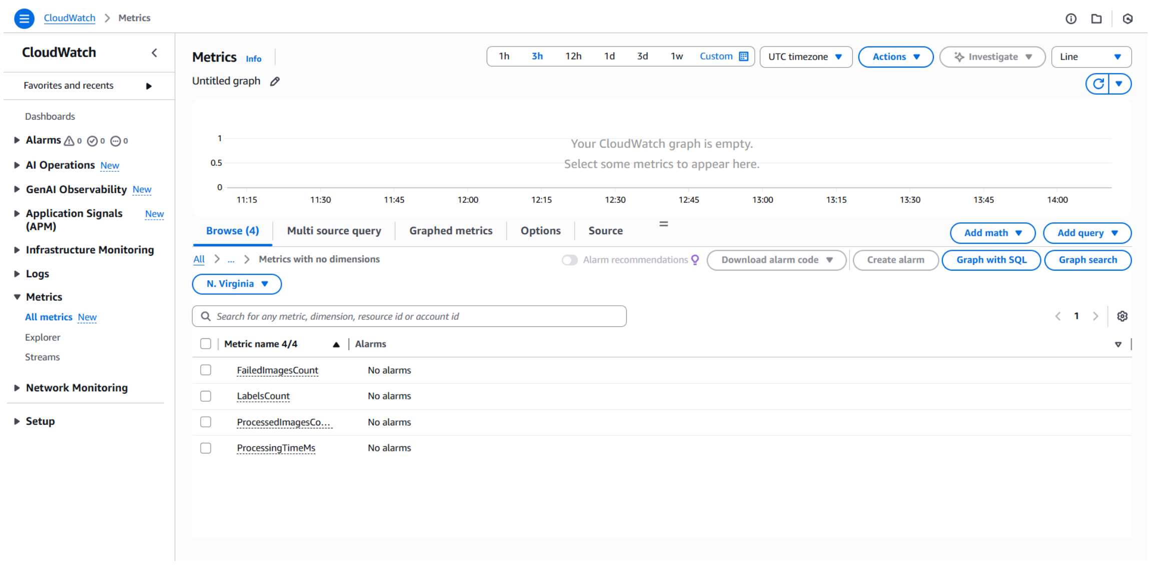
The Lambda function reports custom metrics to the CloudWatch namespace `DV1566/ImageProcessing`. The metrics include:

- **ProcessedImagesCount**: number of successfully processed images.
- **ProcessingTimeMs**: processing duration per image in milliseconds.
- **LabelsCount**: number of labels detected by Rekognition.
- **FailedImagesCount**: number of failed processing attempts.

Each metric is tagged with a dimension called `Operation`, which can take values such as `labels`, `grayscale`, `resize`, `thumbnail`, or `error`. This makes it possible to filter and compare metrics per operation type.

In addition, AWS automatically collects metrics for the Lambda function under the namespace `AWS/Lambda` (invocations, duration, errors, concurrent executions). A CloudWatch dashboard named `dv1566-image-dashboard` was created to visualise both the custom metrics and the Lambda metrics.

Figure 7 shows the list of custom metrics, and Figure 8 shows the dashboard after running tests.



Your CloudWatch graph is empty.
Select some metrics to appear here.

Metric name 4/4	Alarms
FailedImagesCount	No alarms
LabelsCount	No alarms
ProcessedImagesCo...	No alarms
ProcessingTimeMs	No alarms

Figure 7: Custom metrics under the `DV1566/ImageProcessing` namespace.

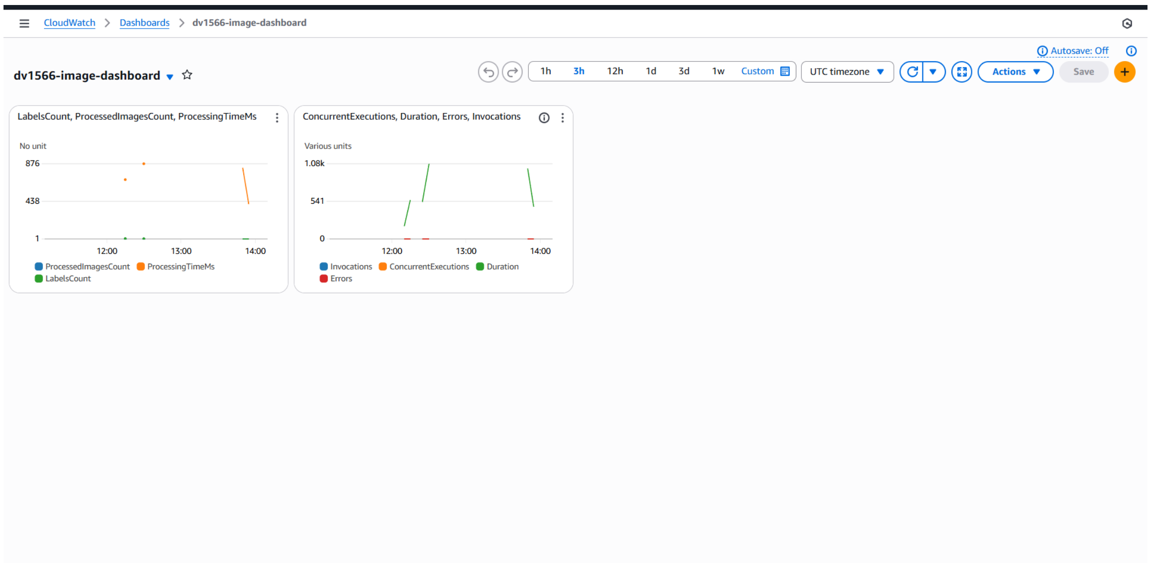


Figure 8: CloudWatch dashboard showing application and Lambda metrics after load tests.

4 Browser-Based User Interface

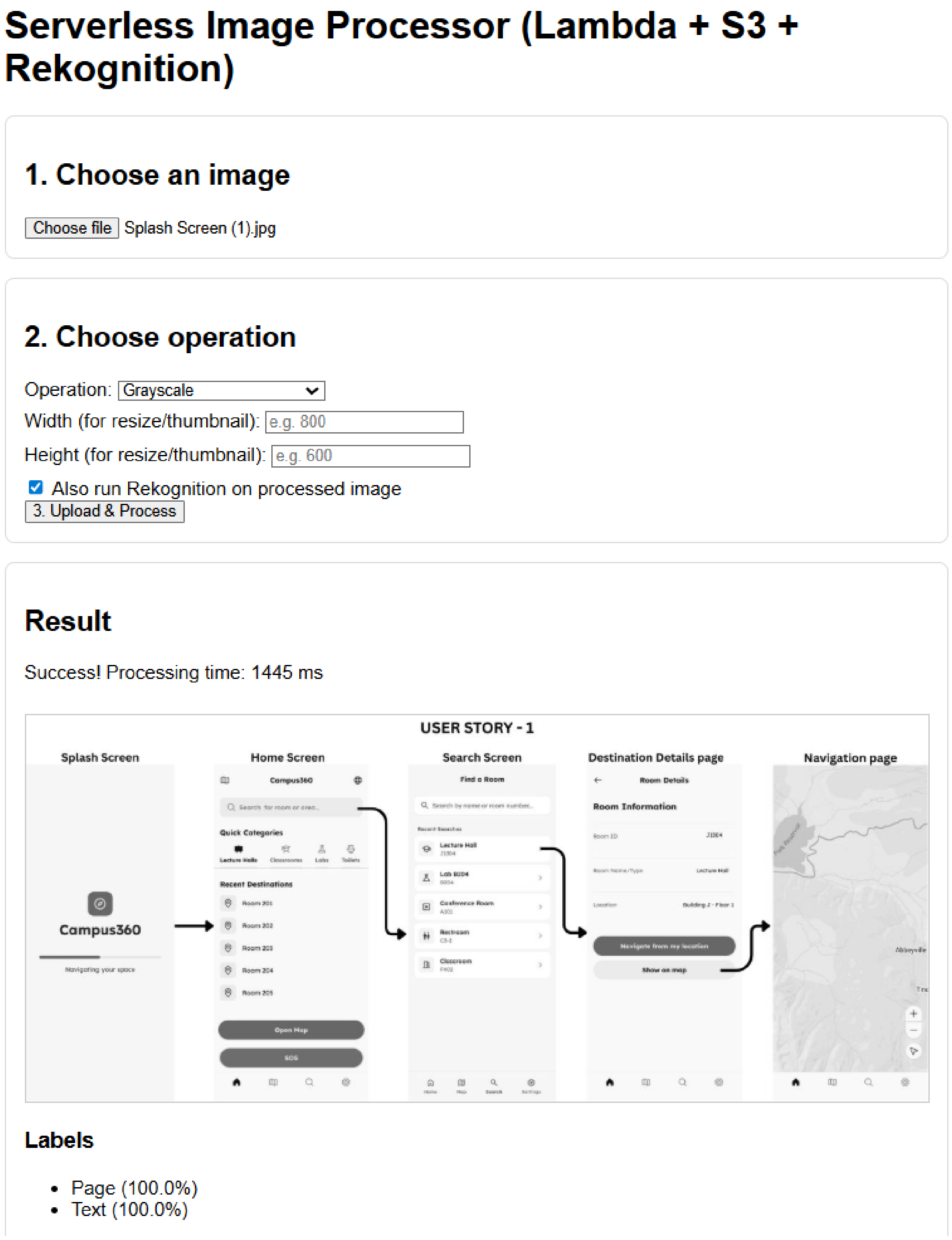
To make the application easier to use, a small browser-based frontend was implemented. This UI is a single HTML file that can be opened locally in the browser (e.g., Chrome). It uses JavaScript to:

1. Let the user choose an image file from the local machine.
2. Convert the file to a base64-encoded string.
3. Send a `POST` request to the API Gateway HTTP endpoint with the operation parameters.
4. Display the processed image and the returned labels.

5. Show the raw JSON response for debugging and verification.

The user can select one of the available operations (labels only, grayscale, resize, or thumbnail) and optionally specify width and height for resizing. The UI also includes a checkbox to enable or disable Rekognition for the processed image.

Figure 9 shows the UI after processing a grayscale image.



Serverless Image Processor (Lambda + S3 + Rekognition)

1. Choose an image

Choose file Splash Screen (1).jpg

2. Choose operation

Operation: Grayscale

Width (for resize/thumbnail): e.g. 800

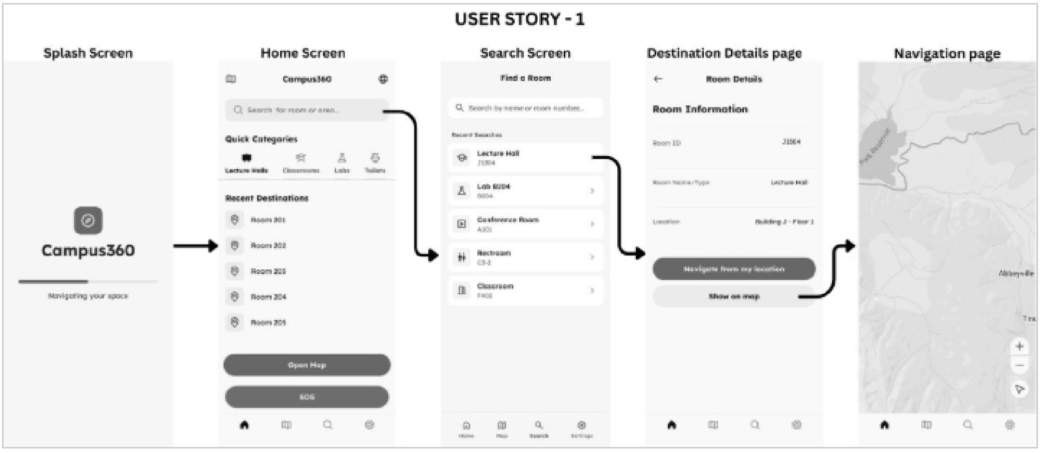
Height (for resize/thumbnail): e.g. 600

☑ Also run Rekognition on processed image

3. Upload & Process

Result

Success! Processing time: 1445 ms



Labels

- Page (100.0%)
- Text (100.0%)

Figure 9: User interface displaying a processed image and Rekognition labels.

5 Testing

5.1 Functional Testing via HTTP API (S3 Mode)

The first set of tests verified that the HTTP endpoint correctly triggers the Lambda function and that the system produces the expected outputs when using the legacy S3 mode. In this mode, an image is uploaded to the input S3 bucket, and a `POST` request is sent to the API endpoint with the following JSON body:

```
{
    "bucket": "dv1566-image-input-nikhil",
    "key": "<image-filename>",
    "operation": "labels",
    "runRekognition": true
}
```

The request can be issued from PowerShell using `Invoke-WebRequest`, for example:

```
Invoke-WebRequest `
  -Uri "https://456uqhtma1.execute-api.us-east-1.amazonaws.com/process" `
  -Method POST `
  -Headers @{ "Content-Type" = "application/json" } `
  -Body '{"bucket":"dv1566-image-input-nikhil",
          "key":"lambda-s3-test-1.png",
          "operation":"labels",
          "runRekognition":true }'
```

The response includes the output bucket, output key of the processed image, processing time, and the number of detected labels.

5.2 Functional Testing via Browser UI

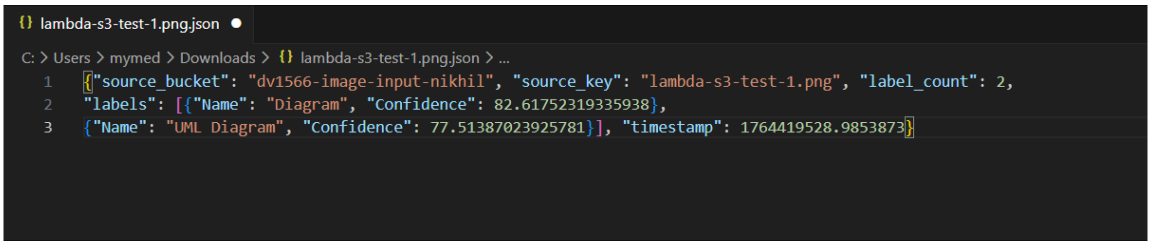
The main usage scenario is via the web UI. For each test case:

1. The user selects a local image file in the browser.
2. Chooses an operation mode.
3. Optionally enters width and height for resizing.
4. Clicks the "Upload & Process" button.

The UI then sends the image as base64 and receives a JSON response similar to the following:

```
{
  "message": "processed_from_http",
  "source_bucket": "dv1566-image-input-nikhil",
  "source_key": "uploads/1766169104053.png",
  "operation": "labels",
  "output_image_bucket": "dv1566-image-output-nikhil",
  "output_image_key": "processed/original/1766169104053.png",
  "processing_time_ms": 1096.14,
  "labels_count": 10,
  "labels": [...],
  "output_image_base64": "..."
}
```

The UI decodes `output_image_base64` and shows the processed image on the page. Labels are listed with their confidence scores. Figure 10 shows an example JSON response for one of the test images.



lambda-s3-test-1.png.json
C: > Users > mymed > Downloads > lambda-s3-test-1.png.json > ...

```
{"source_bucket": "dv1566-image-input-nikhil", "source_key": "lambda-s3-test-1.png", "label_count": 2,
"labels": [{"Name": "Diagram", "Confidence": 82.61752319335938},
{"Name": "UML Diagram", "Confidence": 77.51387023925781}], "timestamp": 1764419528.9853873}
```

Figure 10: Example JSON response / output for a processed image triggered via the UI.

5.3 Functional Testing via S3 Event Triggers

To verify event-driven behaviour, several images were uploaded directly to the input bucket. Each upload generated an S3 `ObjectCreated` event, which invoked the Lambda function. In this scenario, the Lambda function uses the default `labels` operation and stores a processed copy of the image in `processed/original/` in the output bucket. Labels are returned in the function response (visible in CloudWatch logs), and metrics are published.

Table 1 summarises a few S3-triggered test cases.

Table 1: Example functional test cases using S3 upload triggers.

Test case	**Uploaded image**	**Processed output key**
S1	`lambda-s3-test-1.png`	`processed/original/lambda-s3-test-1.png`
S2	`burst1.png`	`processed/original/burst1.png`
S3	`burst2.png`	`processed/original/burst2.png`

6 Scalability Testing

6.1 Steps to Generate Load and View Metrics

To study scalability using CloudWatch, the following steps were executed:

1. Open the CloudWatch dashboard `dv1566-image-dashboard` with widgets for:
    - `DV1566/ImageProcessing`: `ProcessedImagesCount`, `ProcessingTimeMs`, `LabelsCount`, `FailedImagesCount`.
    - `AWS/Lambda`: `Invocations`, `Duration`, optionally `ConcurrentExecutions`.

2. Set the time range of the dashboard to “Last 15 minutes”.
3. Send a single HTTP request from the UI or PowerShell to confirm that metrics appear.
4. Run a small load test with multiple HTTP requests (e.g., a PowerShell loop with ten consecutive requests).
5. Upload 5–10 images at once to the input bucket to generate a burst of S3 events.
6. Observe spikes in `ProcessedImagesCount`, `Invocations`, and `ProcessingTimeMs`, and optionally filter by the `Operation` dimension (labels vs grayscale vs resize).

6.2 API Load Test

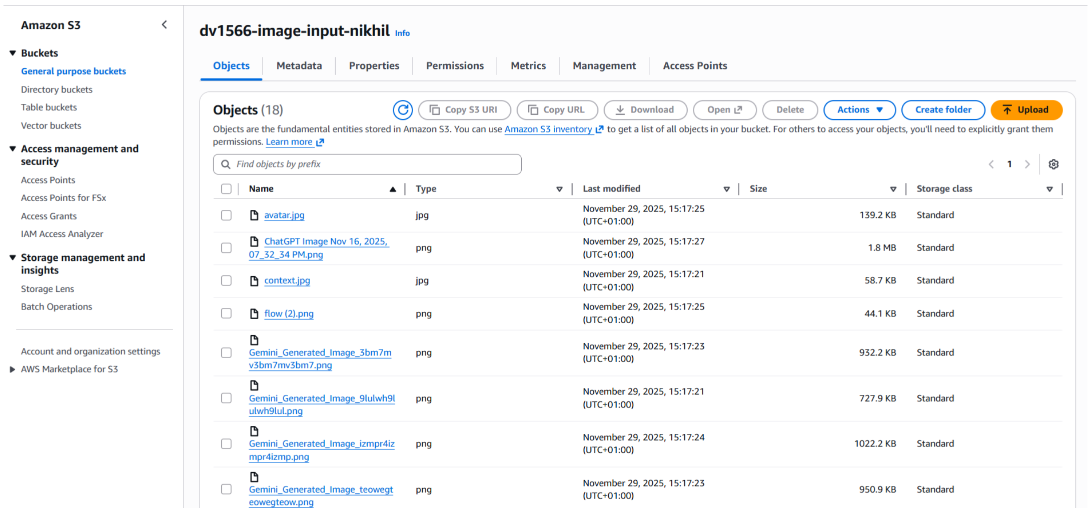
Sending ten consecutive HTTP requests results in ten Lambda invocations. In the CloudWatch dashboard, a clear increase in both the number of invocations and the `ProcessedImagesCount` metric can be seen. The average processing time per image remains stable, demonstrating that the serverless application can handle this level of load without manual scaling.

7 Test Results

This section summarises the outputs of the system after running the final set of tests. New test images were uploaded and processed both via the HTTP API and via the web UI. The processed images in the output bucket and the CloudWatch metrics on the dashboard were inspected to verify correctness and behaviour.

7.1 S3 Buckets After Test Run

Figure 11 shows the input bucket after uploading several new test images. The corresponding processed images produced by the Lambda function can be seen in Figure 12 under the `processed/` prefixes.



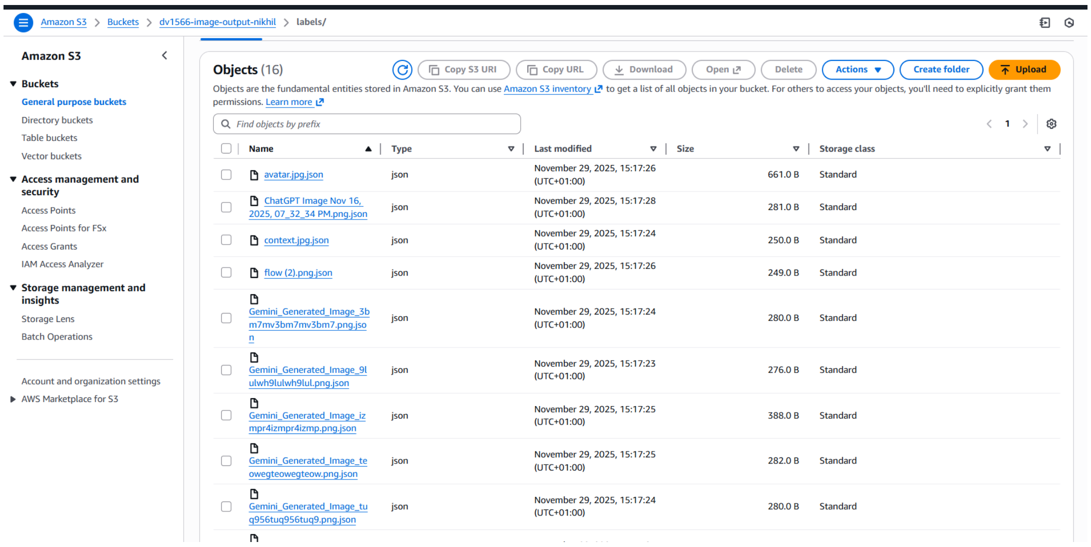
dv1566-image-input-nikhil

Objects (18)

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

Name	Type	Last modified	Size	Storage class
avatar.jpg	jpg	November 29, 2025, 15:17:25 (UTC+01:00)	139.2 KB	Standard
ChatGPT Image Nov 16, 2025, 07_32_34 PM.png	png	November 29, 2025, 15:17:27 (UTC+01:00)	1.8 MB	Standard
context.jpg	jpg	November 29, 2025, 15:17:21 (UTC+01:00)	58.7 KB	Standard
flow (2).png	png	November 29, 2025, 15:17:25 (UTC+01:00)	44.1 KB	Standard
Gemini_Generated_Image_3bm7mv3bm7mv3bm7.png	png	November 29, 2025, 15:17:23 (UTC+01:00)	932.2 KB	Standard
Gemini_Generated_Image_9lulwh9lulwh9lul.png	png	November 29, 2025, 15:17:21 (UTC+01:00)	727.9 KB	Standard
Gemini_Generated_Image_izmpr4izmpr4izmp.png	png	November 29, 2025, 15:17:24 (UTC+01:00)	1022.2 KB	Standard
Gemini_Generated_Image_teowegteowegteow.png	png	November 29, 2025, 15:17:23 (UTC+01:00)	950.9 KB	Standard

Figure 11: Input bucket `dv1566-image-input-nikhil` with the new test images.



Objects (16)

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

Name	Type	Last modified	Size	Storage class
avatar.jpg.json	json	November 29, 2025, 15:17:26 (UTC+01:00)	661.0 B	Standard
ChatGPT Image Nov 16, 2025, 07_32_34 PM.png.json	json	November 29, 2025, 15:17:28 (UTC+01:00)	281.0 B	Standard
context.jpg.json	json	November 29, 2025, 15:17:24 (UTC+01:00)	250.0 B	Standard
flow (2).png.json	json	November 29, 2025, 15:17:26 (UTC+01:00)	249.0 B	Standard
Gemini_Generated_Image_3bm7mv3bm7mv3bm7.png.json	json	November 29, 2025, 15:17:24 (UTC+01:00)	280.0 B	Standard
Gemini_Generated_Image_9lulwh9lulwh9lul.png.json	json	November 29, 2025, 15:17:23 (UTC+01:00)	276.0 B	Standard
Gemini_Generated_Image_izmpr4izmpr4izmp.png.json	json	November 29, 2025, 15:17:25 (UTC+01:00)	388.0 B	Standard
Gemini_Generated_Image_teowegteowegteow.png.json	json	November 29, 2025, 15:17:25 (UTC+01:00)	282.0 B	Standard
Gemini_Generated_Image_tuq956tuq956tuq9.png.json	json	November 29, 2025, 15:17:24 (UTC+01:00)	280.0 B	Standard

Figure 12: Output bucket `dv1566-image-output-nikhil/processed/` with processed images grouped by operation.

7.2 CloudWatch Dashboard After Testing

Figure 13 shows the CloudWatch dashboard after running the tests. The graphs illustrate increases in the number of processed images, Lambda invocations, and processing times during the load periods. Filtering by the `Operation` dimension makes it possible to compare the cost of different operations (e.g., labels only vs grayscale vs resize).

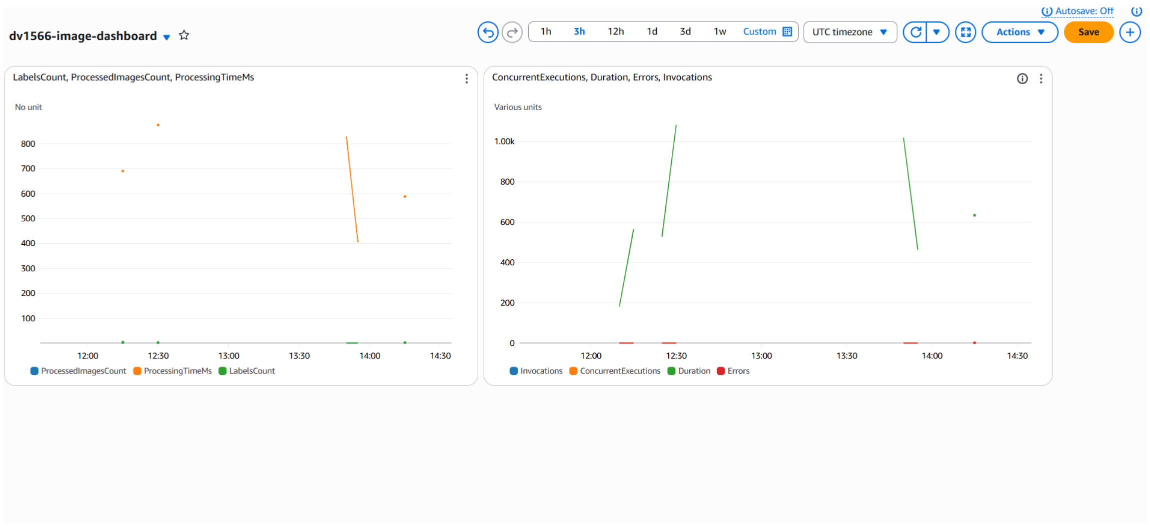


Figure 13: CloudWatch dashboard after running the HTTP API and S3 tests.

8 Results and Discussion

The functional tests confirm that the system correctly processes images from both the HTTP API and S3 upload triggers. For each input image, a processed copy is written to the output bucket, and when Rekognition is enabled, labels with confidence scores are returned in the JSON response. The labels are reasonable for the test images (e.g., books, diagrams, pages).

The CloudWatch metrics and dashboard show that the system can process a sequence of requests and a small burst of S3 events without errors. Individual images are typically processed in hundreds of milliseconds, including both image transformation and Rekognition. This performance is acceptable for the intended lab setting.

The introduction of additional operations (grayscale, resize, thumbnail) demonstrates how lightweight image processing tasks can be integrated into a fully serverless pipeline. Using a single Lambda function to coordinate both pixel-level operations and cloud-based computer vision simplifies deployment while still providing flexibility.

The main limitation of the experiments is that they were conducted at a relatively small scale due to the resource constraints of the AWS Academy Learner Lab. However, the serverless design can be scaled out by adjusting concurrency limits and by introducing more advanced dashboards and alarms if deployed in a production environment.

9 Conclusion

In this laboratory work, a serverless image-processing application was designed, implemented, and extended using Amazon Web Services. The system integrates Amazon S3, AWS Lambda, Amazon API Gateway, Amazon Rekognition, and Amazon CloudWatch to provide an end-to-end workflow for analysing and transforming images, storing results, and monitoring performance.

The initial version focused on label detection using Rekognition, while the extended version also performs grayscale conversion, image resizing, and thumbnail generation using the Pillow library inside AWS Lambda. A small web UI was developed to allow users to upload images directly from the browser and visualise processed outputs immediately.

The results demonstrate that the solution works correctly for individual images and small batches and that the serverless approach can automatically scale with the number of requests and S3 events without manual server management. As future work, the application could be enhanced with additional processing features, more detailed performance analysis, or a richer frontend for end users.

Appendix A: Lambda Function Source Code

Listing 1: Full source code of the Lambda function `dv1566-image-processor`.

```python
import os
import json
import time
import base64
import urllib.parse
import io

import boto3
from PIL import Image

s3 = boto3.client("s3")
rek = boto3.client("rekognition")
cw = boto3.client("cloudwatch")

INPUT_BUCKET = os.environ.get("INPUT_BUCKET", "dv1566-image-input-nikhil")
OUTPUT_BUCKET = os.environ.get("OUTPUT_BUCKET", "dv1566-image-output-nikhil")

def put_metrics(success=True, processing_ms=None,
               labels_count=0, operation="labels"):
    metric_data = []

    if success:
        metric_data.append({
            "MetricName": "ProcessedImagesCount",
            "Value": 1,
            "Unit": "Count"
        })
    else:
        metric_data.append({
            "MetricName": "FailedImagesCount",
            "Value": 1,
            "Unit": "Count"
        })

    if processing_ms is not None:
        metric_data.append({
            "MetricName": "ProcessingTimeMs",
            "Value": processing_ms,
            "Unit": "Milliseconds"
        })

    metric_data.append({
        "MetricName": "LabelsCount",
        "Value": labels_count,
        "Unit": "Count"
    })
    cw.put_metric_data(
        Namespace="DV1566/ImageProcessing",
        MetricData=[
            {
                **m,
                "Dimensions": [
                    {"Name": "Operation", "Value": operation}
                ]
            }
            for m in metric_data
        ]
    )

def run_rekognition(bucket, key):
    r = rek.detect_labels(
        Image={"S3Object": {"Bucket": bucket, "Name": key}},
        MaxLabels=10,
        MinConfidence=70.0
    )
    labels = r.get("Labels", [])
    labels_out = [
        {"Name": lab["Name"], "Confidence": lab["Confidence"]}
        for lab in labels
    ]
    return labels_out

def apply_operation(image_bytes, operation, width=None, height=None):
    if operation == "labels":
        return image_bytes, "original"

    img = Image.open(io.BytesIO(image_bytes))

    if operation == "grayscale":
        img = img.convert("L")
    elif operation == "resize":
        if width is None or height is None:
            width, height = img.size
        img = img.resize((int(width), int(height)))
    elif operation == "thumbnail":
        if width is None or height is None:
            width, height = 256, 256
        img.thumbnail((int(width), int(height)))
    else:
        operation = "labels"
        return image_bytes, "original"

    buf = io.BytesIO()
    img.save(buf, format="PNG")
    buf.seek(0)
    return buf.read(), "processed"
```

```python
def process_image_from_s3(bucket, key, operation="labels",
                          width=None, height=None,
                          run_rekognition_flag=True):
    t0 = time.time()

    obj = s3.get_object(Bucket=bucket, Key=key)
    original_bytes = obj["Body"].read()

    processed_bytes, kind = apply_operation(
        original_bytes, operation, width, height
    )

    base_name = os.path.basename(key)

    if kind == "original":
        out_key = f"processed/original/{base_name}"
    else:
        out_key = f"processed/{operation}/{base_name}"

    s3.put_object(
        Bucket=OUTPUT_BUCKET,
        Key=out_key,
        Body=processed_bytes,
        ContentType="image/png"
    )

    labels_out = []
    if run_rekognition_flag:
        labels_out = run_rekognition(OUTPUT_BUCKET, out_key)

    t1_ms = (time.time() - t0) * 1000.0

    put_metrics(
        success=True,
        processing_ms=t1_ms,
        labels_count=len(labels_out),
        operation=operation
    )

    processed_b64 = base64.b64encode(processed_bytes).decode("utf-8")

    result = {
        "source_bucket": bucket,
        "source_key": key,
        "operation": operation,
        "output_image_bucket": OUTPUT_BUCKET,
        "output_image_key": out_key,
        "processing_time_ms": t1_ms,
        "labels_count": len(labels_out),
        "labels": labels_out,
        "output_image_base64": processed_b64
    }
    return result

def lambda_handler(event, context):
    try:
        if ("Records" in event and
            event["Records"][0].get("eventSource") == "aws:s3"):
            rec = event["Records"][0]
            b = rec["s3"]["bucket"]["name"]
            raw_key = rec["s3"]["object"]["key"]
            k = urllib.parse.unquote_plus(raw_key)

            res = process_image_from_s3(
                bucket=b,
                key=k,
                operation="labels",
                width=None,
                height=None,
                run_rekognition_flag=True
            )

            return {
                "statusCode": 200,
                "body": json.dumps({
                    "message": "processed_from_s3",
                    **res
                })
            }

        body = event.get("body")
        if body is None:
            raise ValueError("Missing body in HTTP event")

        if event.get("isBase64Encoded"):
            body = base64.b64decode(body).decode("utf-8")

        data = json.loads(body)

        operation = data.get("operation", "labels")
        run_rekognition_flag = bool(data.get("runRekognition", True))
        width = data.get("width")
        height = data.get("height")

        if "imageBase64" in data:
            img_b64 = data["imageBase64"]
            img_bytes = base64.b64decode(img_b64)

            key = f"uploads/{int(time.time() * 1000)}.png"
            s3.put_object(
                Bucket=INPUT_BUCKET,
                Key=key,
                Body=img_bytes,
                ContentType="image/png"
            )
            source_bucket = INPUT_BUCKET
        else:
            source_bucket = data["bucket"]
            key = data["key"]

        res = process_image_from_s3(
            bucket=source_bucket,
            key=key,
            operation=operation,
            width=width,
            height=height,
            run_rekognition_flag=run_rekognition_flag
        )

        return {
            "statusCode": 200,
            "headers": {
                "Content-Type": "application/json",
                "Access-Control-Allow-Origin": "*"
            },
            "body": json.dumps({
                "message": "processed_from_http",
                **res
            })
        }

    except Exception as e:
        try:
            put_metrics(
                success=False,
                processing_ms=None,
                labels_count=0,
                operation="error"
            )
        except Exception:
            pass

        return {
            "statusCode": 500,
            "headers": {
                "Content-Type": "application/json",
                "Access-Control-Allow-Origin": "*"
            },
            "body": json.dumps({"error": str(e)}) }
```