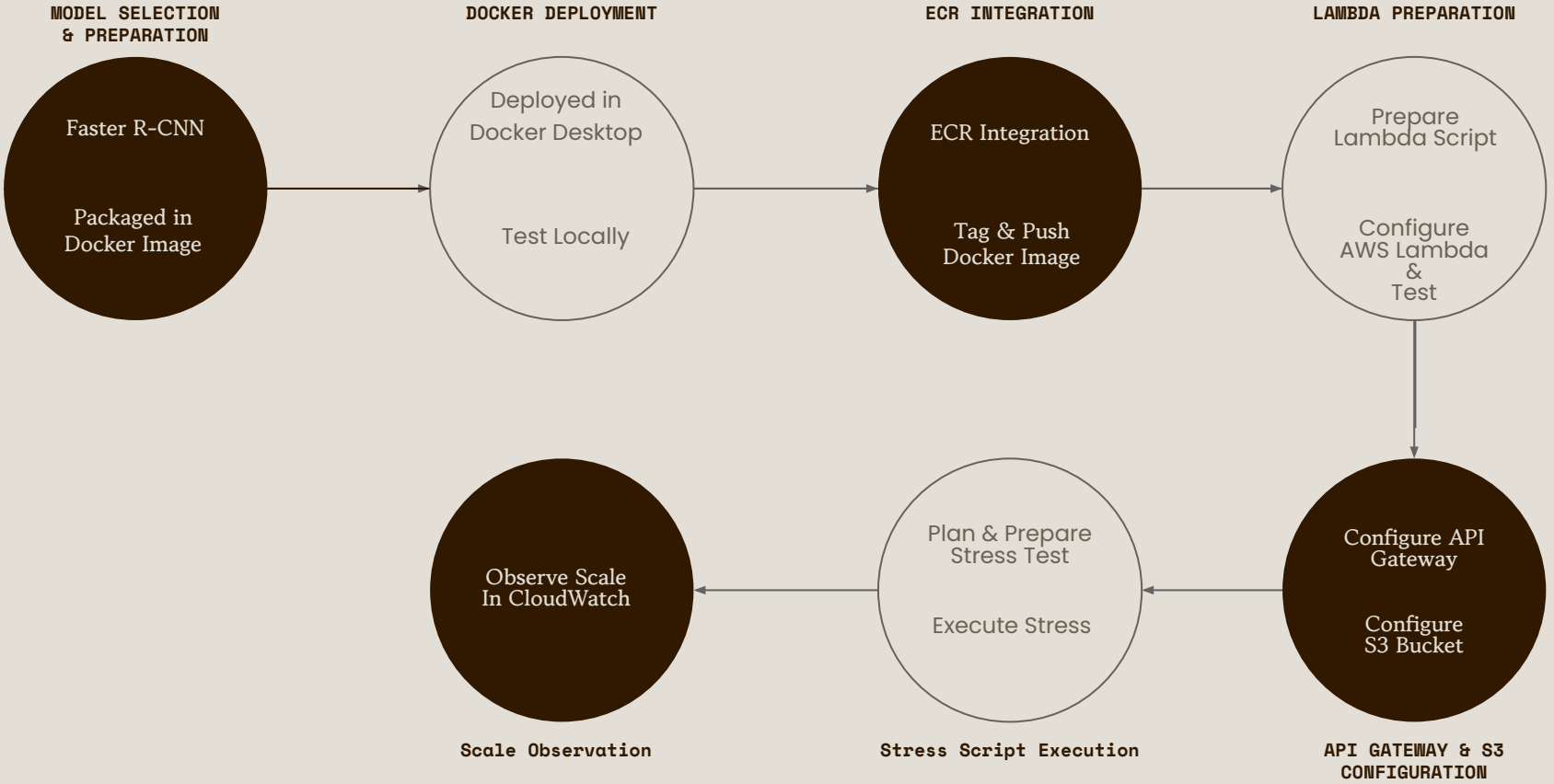
ML Model Deployment on AWS Lambda: Observing Scalability

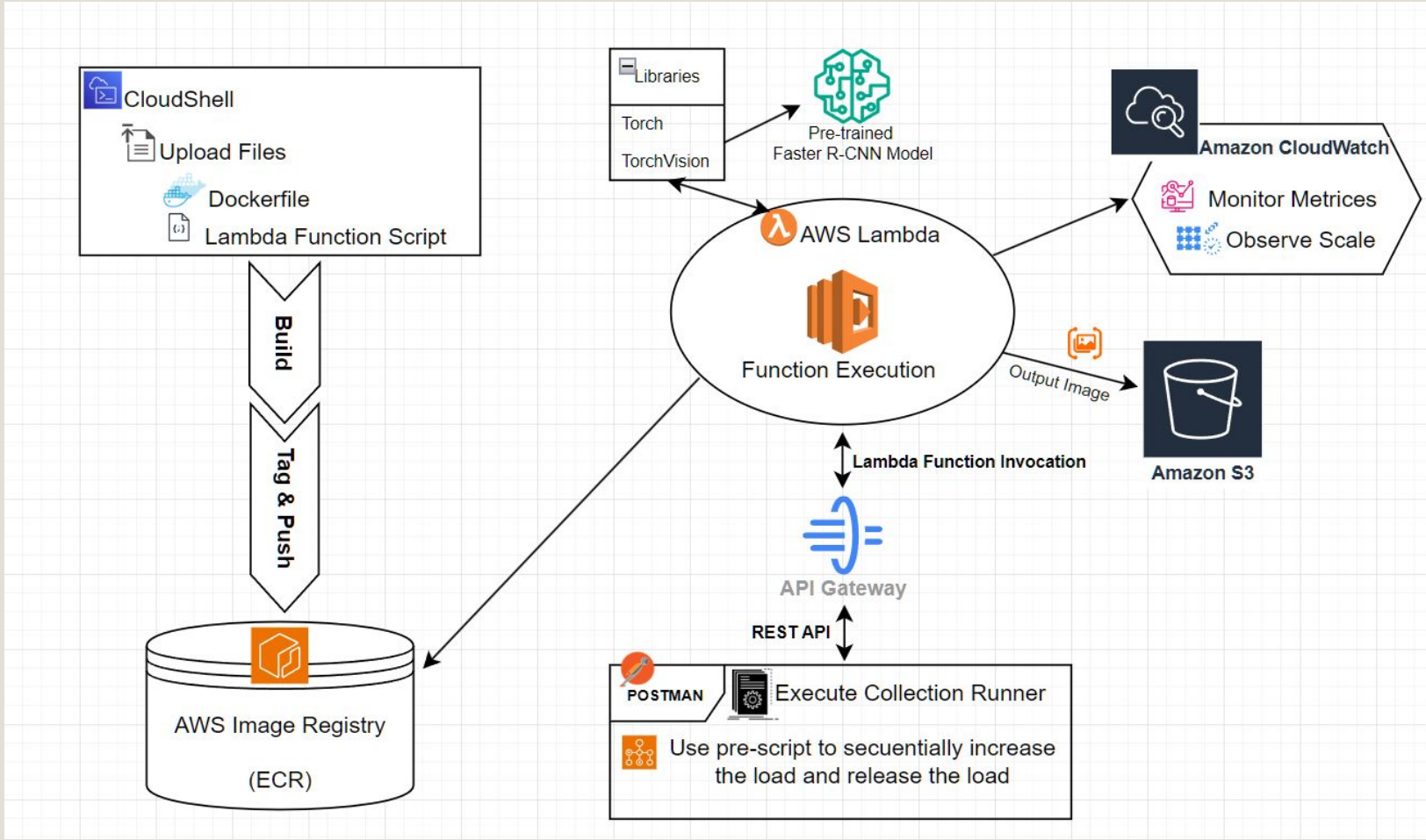
BOKHTIAR MEHEDY

Objective

- Deploy a pre-trained Faster R-CNN model for accurate **object detection** on AWS Lambda
- Observe **deployment scale** by analyzing performance metrics in CloudWatch

Solution Overview




CloudShell
Upload Files
Dockerfile
Lambda Function Script
Build
Tag & Push
AWS Image Registry
(ECR)
Libraries
Torch
TorchVision
Pre-trained
Faster R-CNN Model
AWS Lambda
Function Execution
Amazon CloudWatch
Monitor Metrices
Observe Scale
Output Image
Amazon S3
Lambda Function Invocation
API Gateway
REST API
POSTMAN
Execute Collection Runner
Use pre-script to secuentially increase
the load and release the load

Model Selection

Model Evaluation & Selection

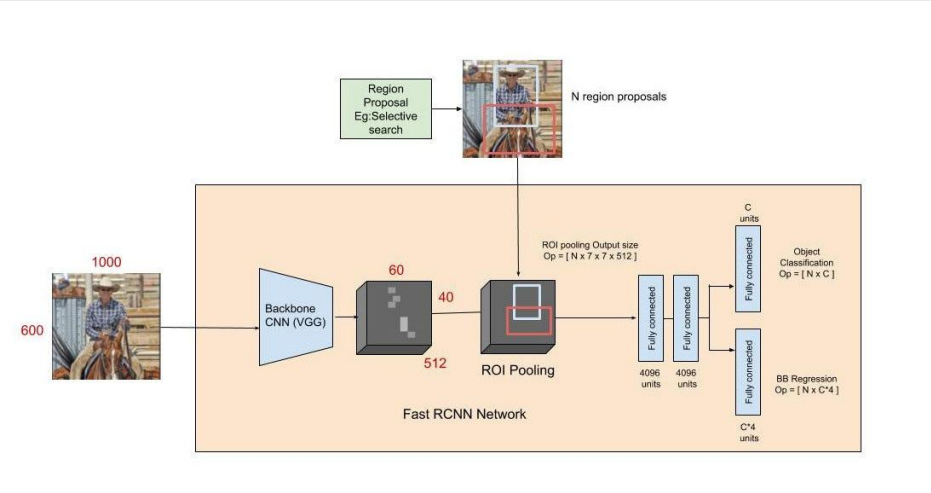
- Lightweight models like **SSDlite** were tried but observed poor detection performance.
- Selected **Faster R-CNN** for superior accuracy despite its heavier architecture.

Language and Library

- Decided to use **Python** as programming language
- PyTorch (Torch & TorchVision) as library
- Pre-trained model ***fasterrcnn_resnet50_fpn_v2*** and weights were used.

Challenge

- The size of the libraries are more than **700MB (170 MB CPU only)**
- Size of the pre-trained model in **168MB**
- Lambda has size constraints for **direct deployment** (250 MB limit for zipped packages) and **AWS Lambda Layers** (50 MB per layer)
- Necessitating the use of a **Docker image** for deployment (up to 10 GB)
- Using a Docker Image even considered standard for Lambda deployment



Preparing Docker Image

Dockerfile

- Configured **Dockerfile** with Lambda-compatible Python runtime to build the image.
- Packaged the Faster R-CNN model, dependencies (Torch and TorchVision), and the lambda_function.py script that implements the object detection.

Validation

- Locally validated the image using Docker Desktop

```
# Start with the official AWS Lambda Python runtime image
FROM public.ecr.aws/lambda/python:3.8

# Install necessary dependencies
RUN pip install torch==2.4.1 torchvision==0.19.1 --index-url https://download.pytorch.org/whl/cpu

# Copy the Lambda function code
COPY lambda_function.py .

# Set the handler to function
CMD ["lambda_function.lambda_handler"]

```

Preparing Docker Image

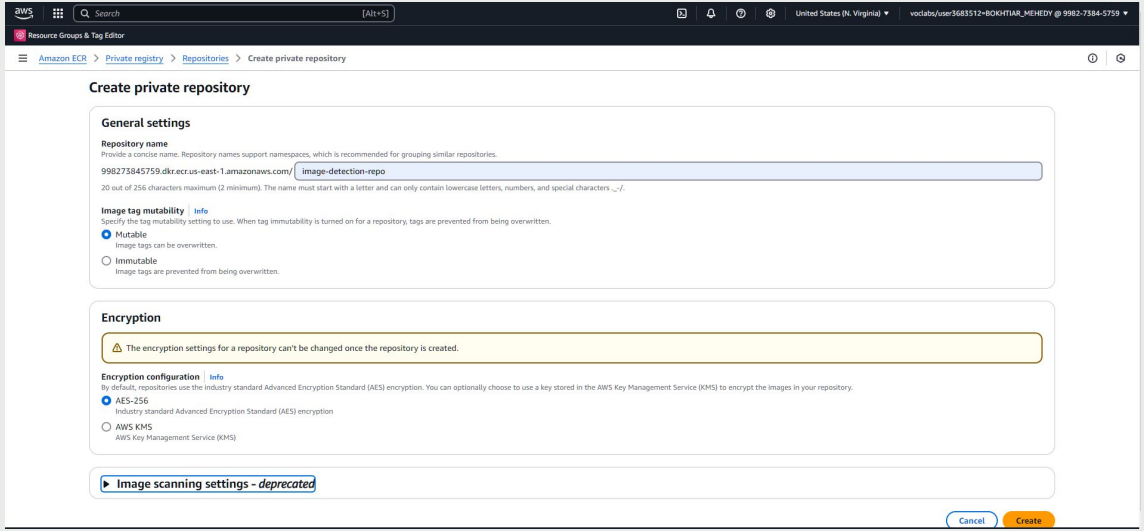
Lambda Function Script

```python
import os
import boto3
import torch
from torchvision.models.detection import fasterrcnn_resnet50_fpn_v2, FasterRCNN_ResNet50_FPN_V2_Weights
from torchvision.transforms import functional as F
from PIL import Image, ImageDraw
import io
import base64

# Set model weights to download in /tmp
os.environ["TORCH_HOME"] = "/tmp"

weights = FasterRCNN_ResNet50_FPN_V2_Weights.DEFAULT
model = fasterrcnn_resnet50_fpn_v2(weights=weights, box_score_thresh=0.9)
model.eval()

# Initialize S3 client
s3_client = boto3.client('s3')

# Lambda function handler
def lambda_handler(event, context):
    try:
        # Directly decode the base64-encoded image from the body
        image_data = base64.b64decode(event["body"])

        # Decode image
        image = Image.open(io.BytesIO(image_data))

        # Preprocess the image
        transform = weights.transforms()
        image_tensor = transform(image).unsqueeze(0)

        # Perform object detection
        with torch.no_grad():
            prediction = model(image_tensor)[0]

        # Extract predictions
        boxes = prediction["boxes"].cpu().numpy()
        labels = prediction["labels"].cpu().numpy()

        # Draw bounding boxes on the image
        draw = ImageDraw.Draw(image)
        for box in boxes:
            draw.rectangle(box.tolist(), outline="red", width=3)

        # Save the processed image to /tmp
        processed_image_path = "/tmp/processed_image.jpg"
        image.save(processed_image_path)

        # Upload the processed image to an S3 bucket
        bucket_name = "object-detected-images"
        object_key = "processed_image.jpg"
        s3_client.upload_file(processed_image_path, bucket_name, object_key)

        s3_url = f"https://{bucket_name}.s3.amazonaws.com/{object_key}"

        return {
            "statusCode": 200,
            "body": {
                "boxes": boxes.tolist(),
                "labels": labels.tolist(),
                "s3_url": s3_url
            }
        }
```

Configuring ECR

Configure the ECR

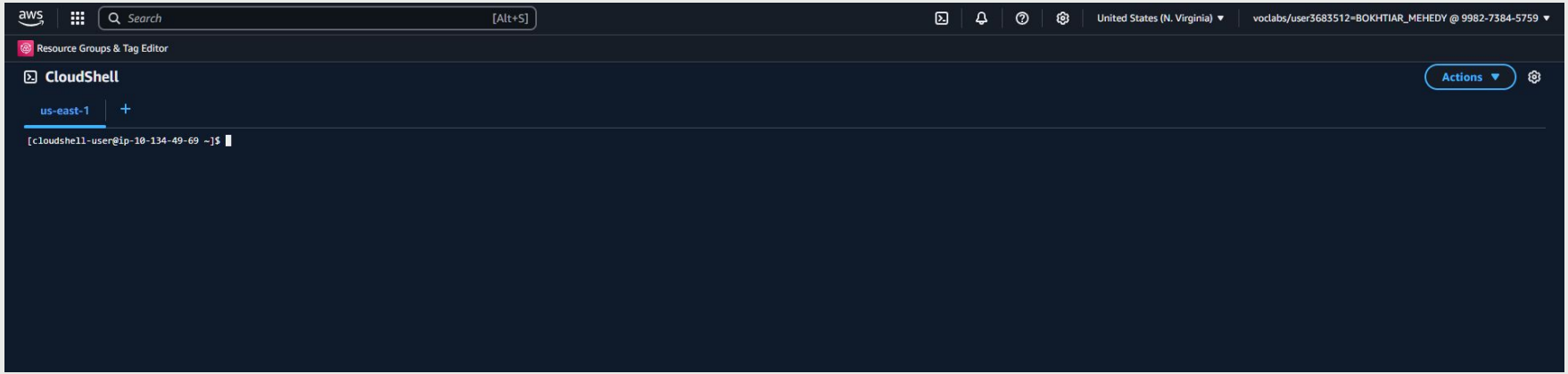
- To push our Docker Image we configured AWS Elastic Cloud Repository (ECR)
- Later we will create the Lambda Function from our image on the ECR



Using AWS CloudShell

AWS CLI
Unable to configure AWS CLI in local PC due to limited access (couldn't create a new IAM).

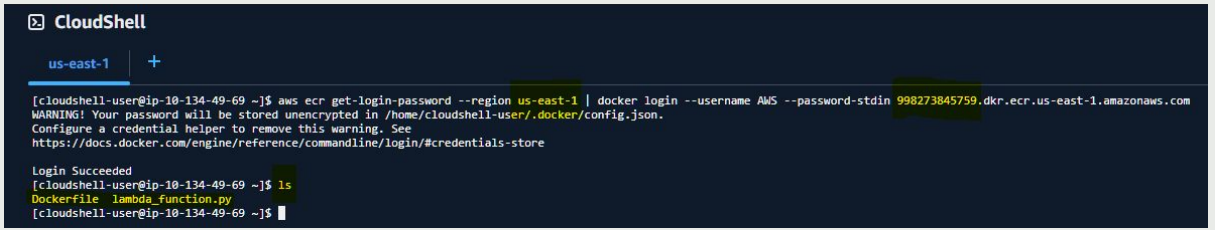
AWS CloudSell
Decided to use CloudShell to deploy the image in ECR



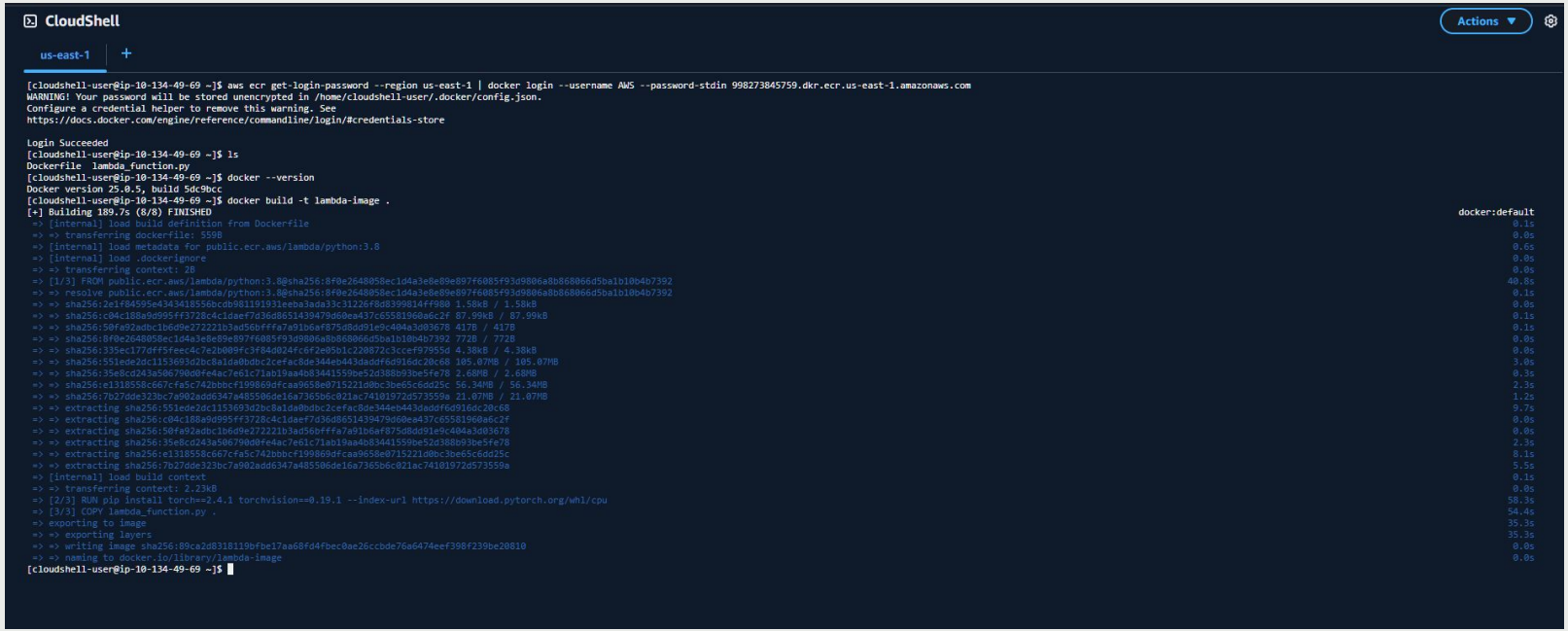
Build, Tag, & Push Image

Build, Tag, & Push the image to Repository

- We uploaded the **`Dockerfile`** and the **`lambda_function.py`** in CloudShell
- Then using CloudShell command window we ran commands to build the image and the Tag and Push it to the repository.

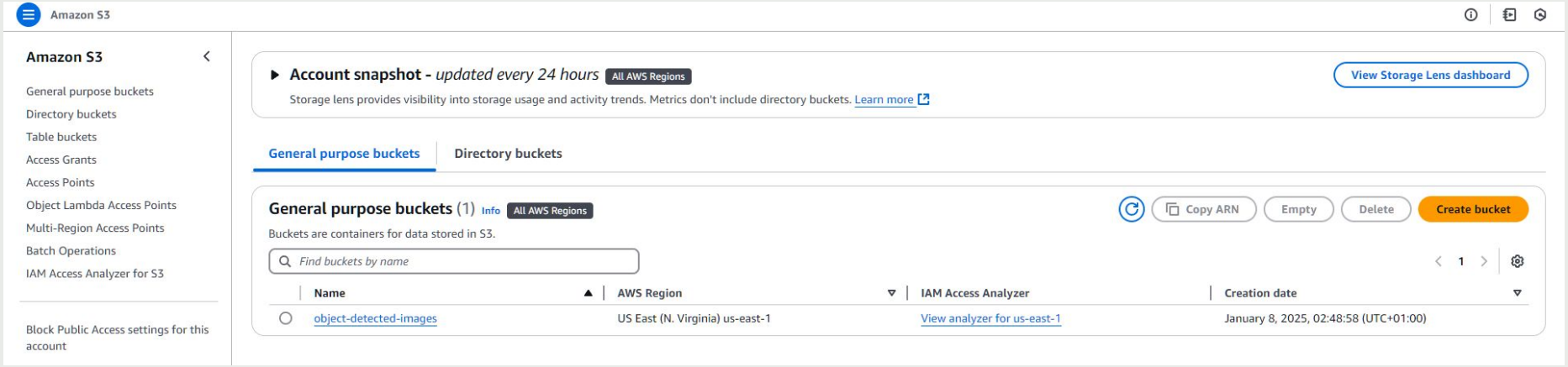


Build, Tag, & Push Image



Create & Configure S3 Bucket

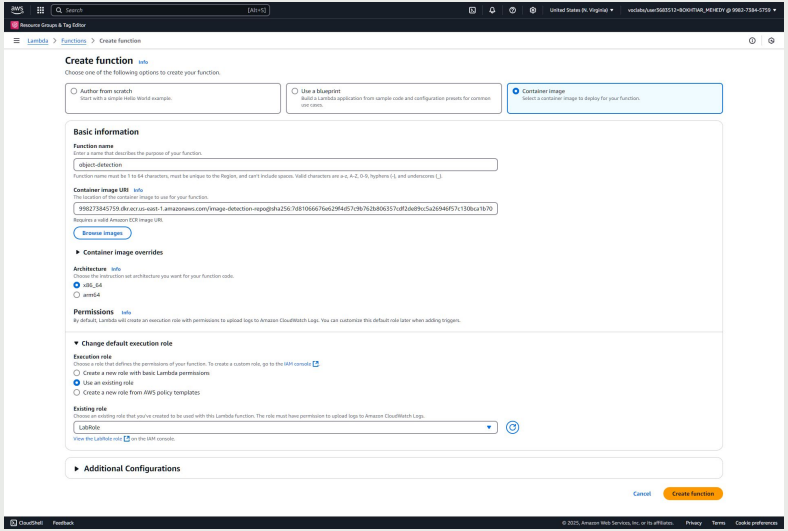
Create and Configure S3 Bucket
Along with JSON response (object coordinators: box) our function should store the object detected output image in a S3 bucket.

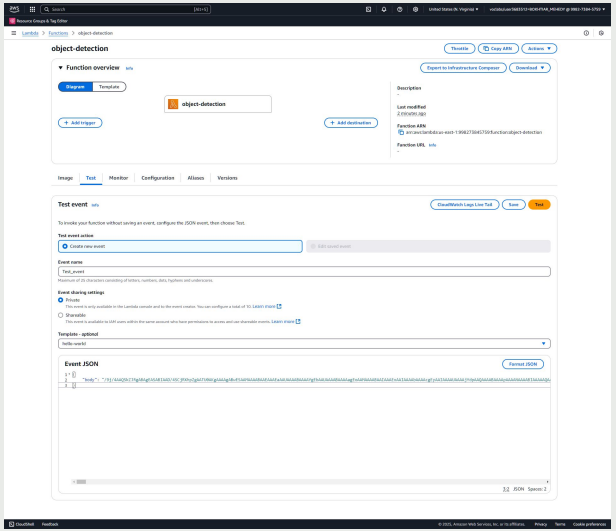


Create & Configure Lambda Function

Create and Configure AWS Lambda Function

- We created (selecting From Image) and configured a Lambda function to deploy our ML model that will detect objects in an image.
- We used an existing role to configure execution permission
- We tested the function using the Test option

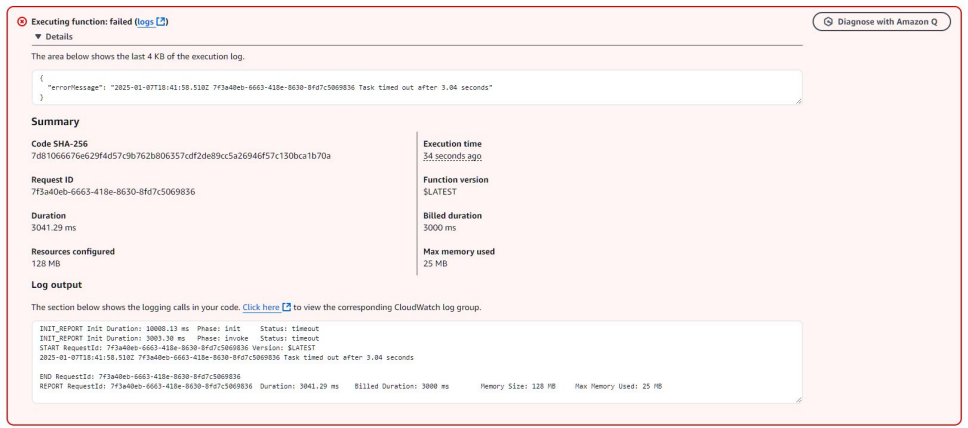


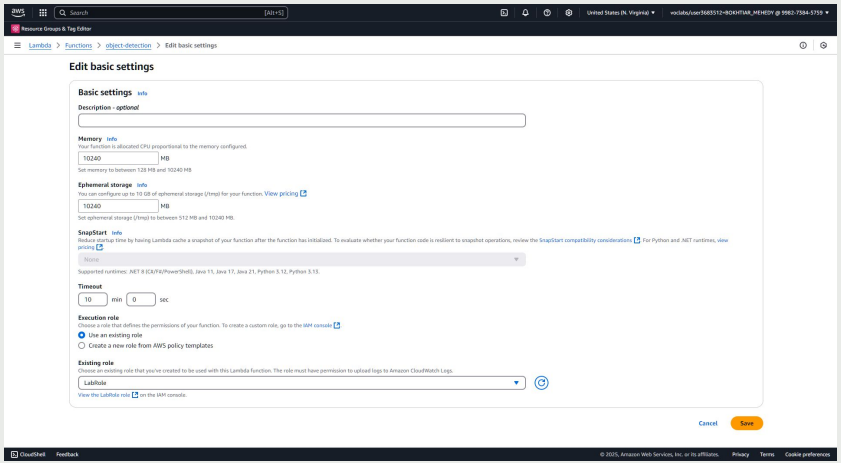


Create & Configure Lambda Function

Encountered an error and changed the configurations to fix

The test was failed because of the max execution timeout, so we configured and increased the limits

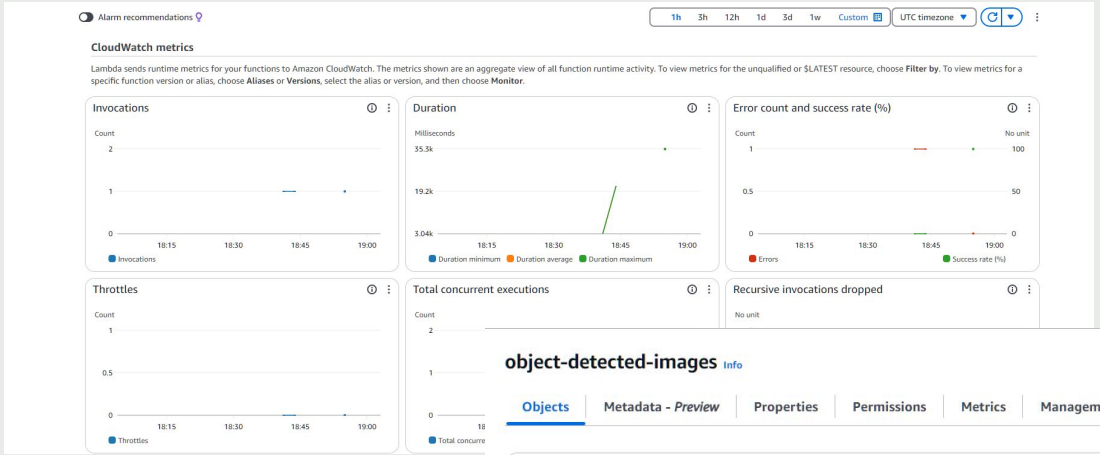


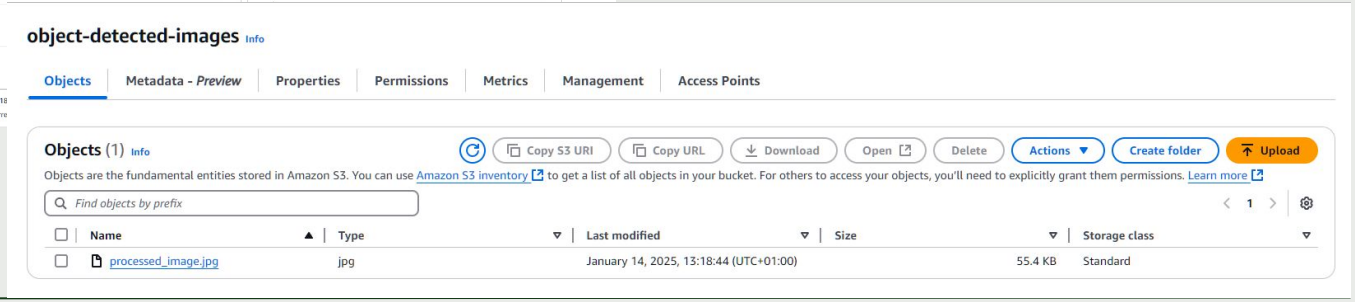


Test Lambda Function

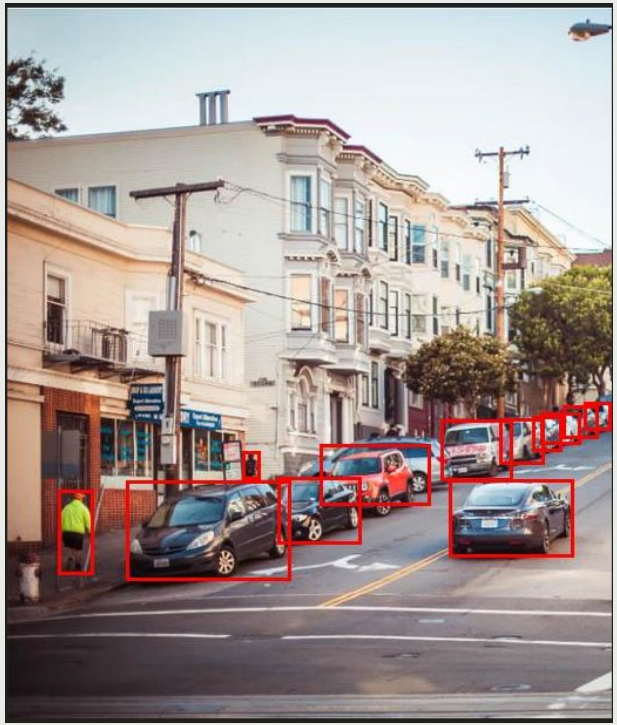
Test succeeded

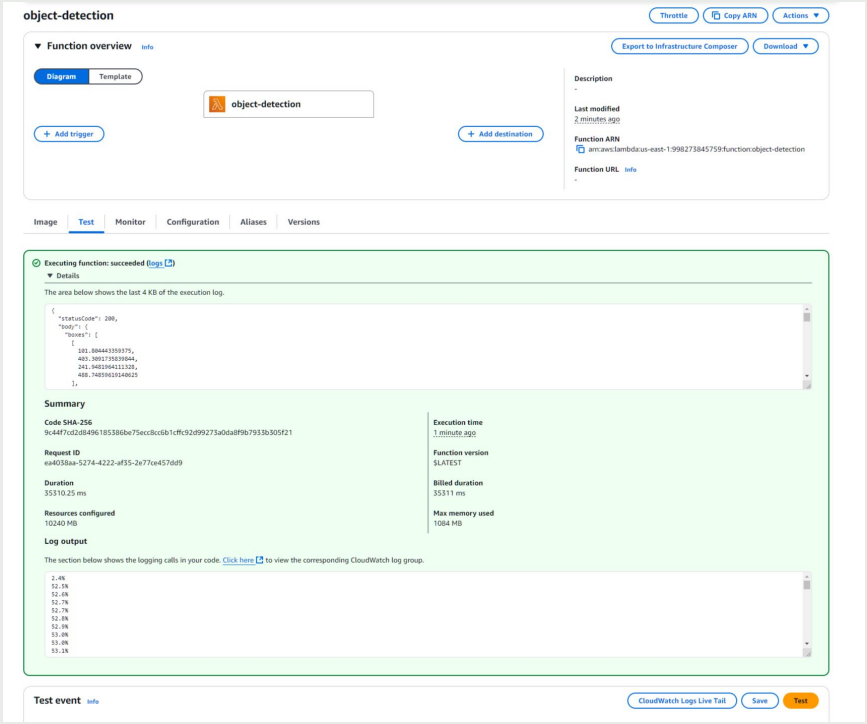
- The configuration changes fixed the issue.
- We got a response with status code 200 and expected payload.
- We can observer the Invocation, Error, and other metrics in CloudWatch. Also the object detected image is stored in the S3 bucket.





Test Lambda Function





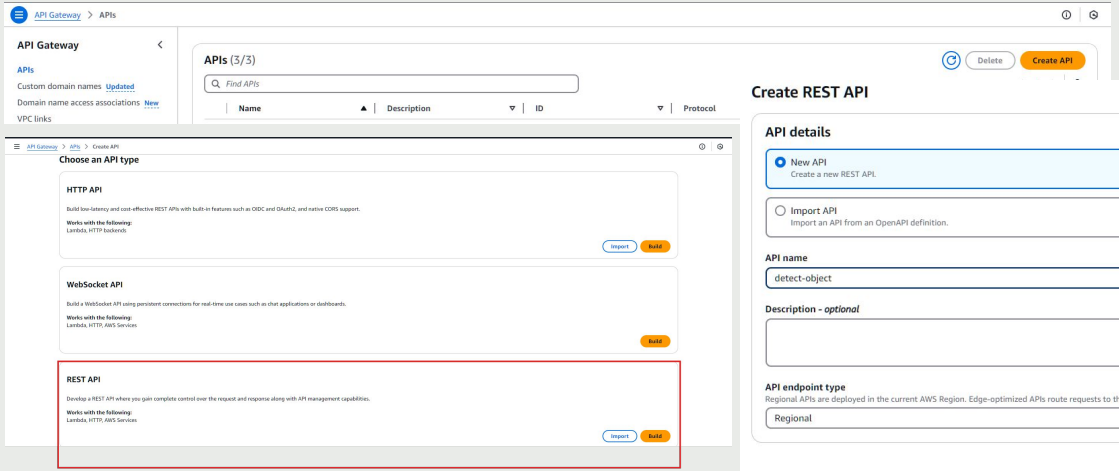
Create and Configure API Gateway

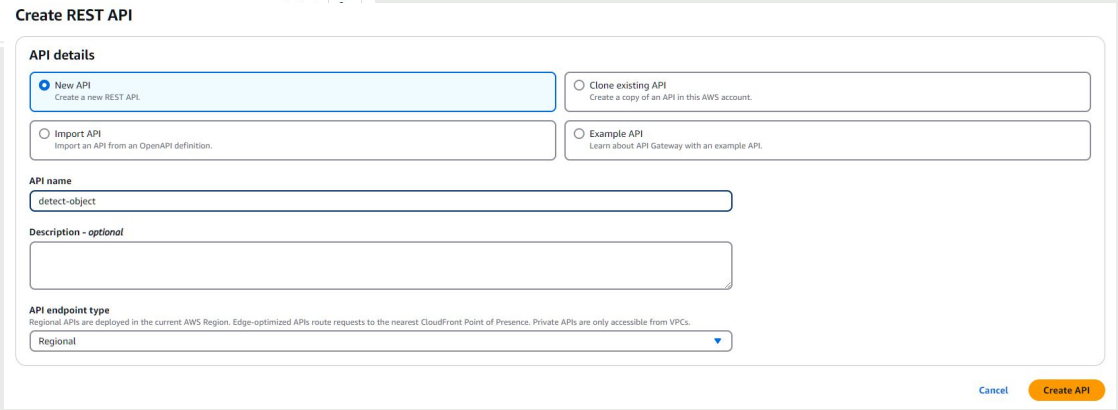
Planning Stress

- While planning to stress the Lambda function with multiple concurrent invocation we planned to expose the Lambda Function with REST API that we can call from a API agent like Postman.
- We'll call the REST API endpoint concurrently that will invoke the Lambda function.

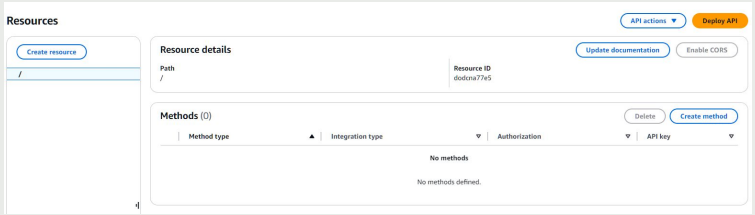
Create and Configure API Gateway

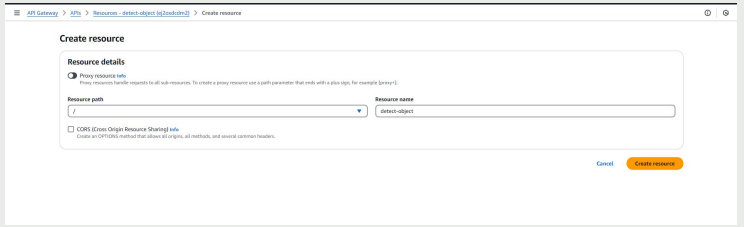
We created and configured API Gateway to expose and POST method REST API endpoint.

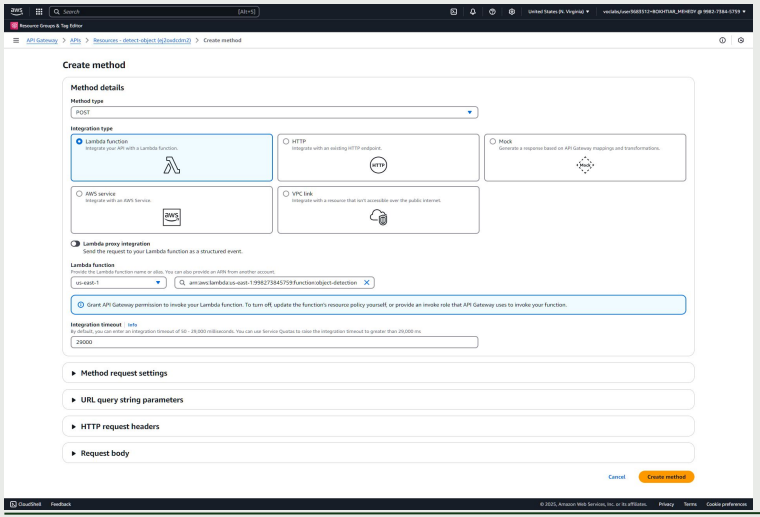


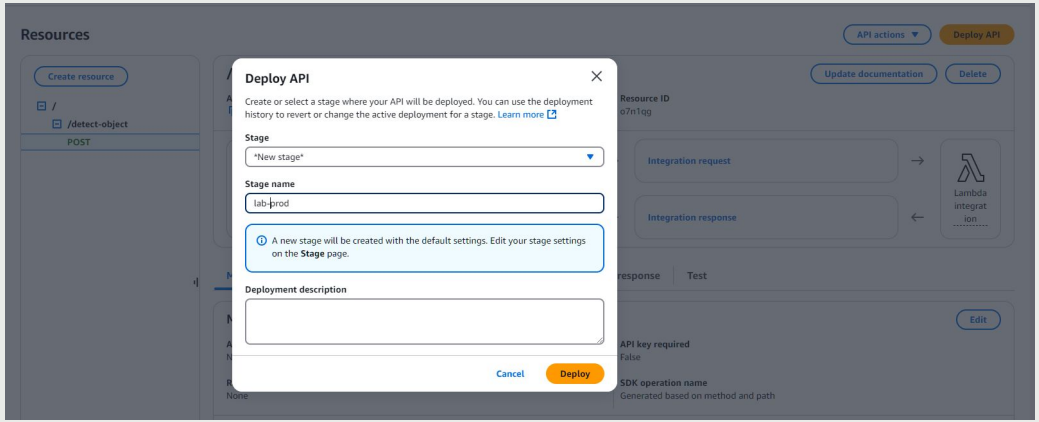


Create and Configure API Gateway



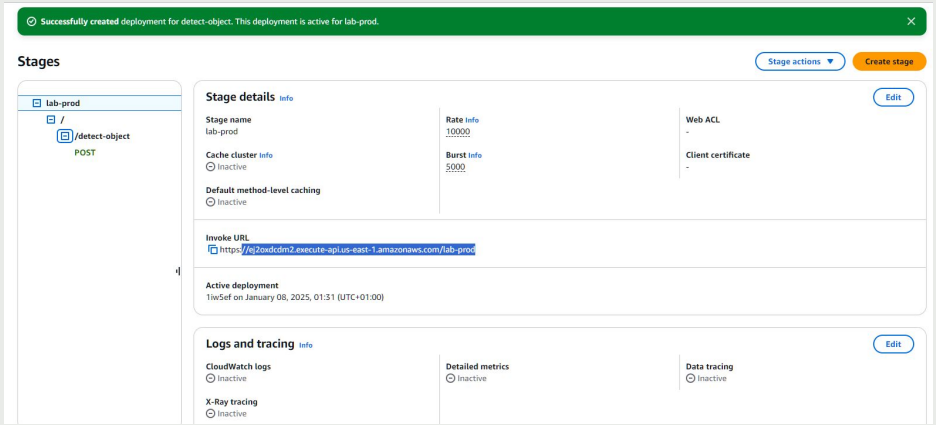


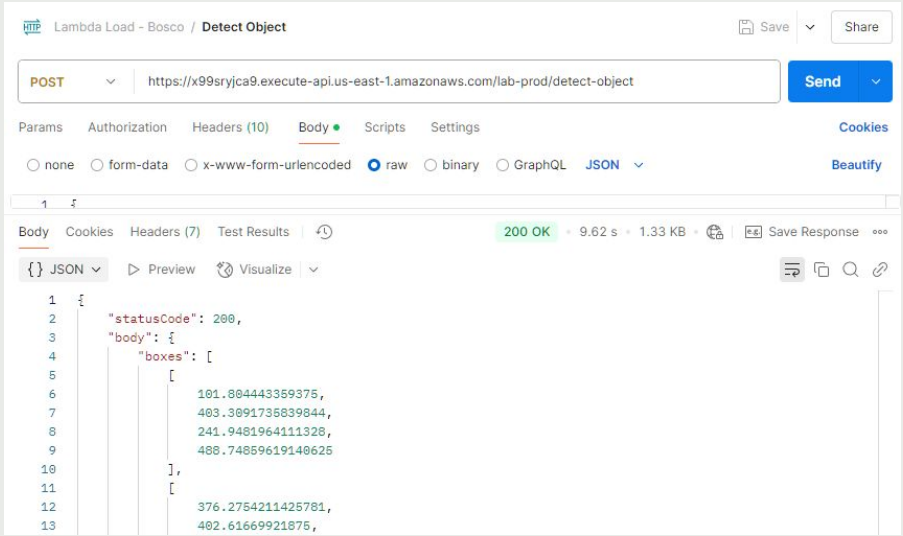




Test API

Test API Endpoint from POSTMAN
We sent request from **Postman** to verify the API endpoint exposed in the API Gateway with the **Invoke URL** and configured resource





Stress Lambda Function

Stress Using Collection Runner in POSTMAN

- We decided to use Postman Collection Runner to send concurrent API request that will subsequently invoke the lambda function.
- We'll monitor mainly the (Concurrent Execution) metrics along with the Invocation, Duration and Error Count.
- Our goal was to increase the stress gradually (increasing the number of concurrent API call) and then release the stress (reducing concurrent API call number).
- We planned to the concurrent number of API: 2, 4, 8, 16 then 8, 4, 2.

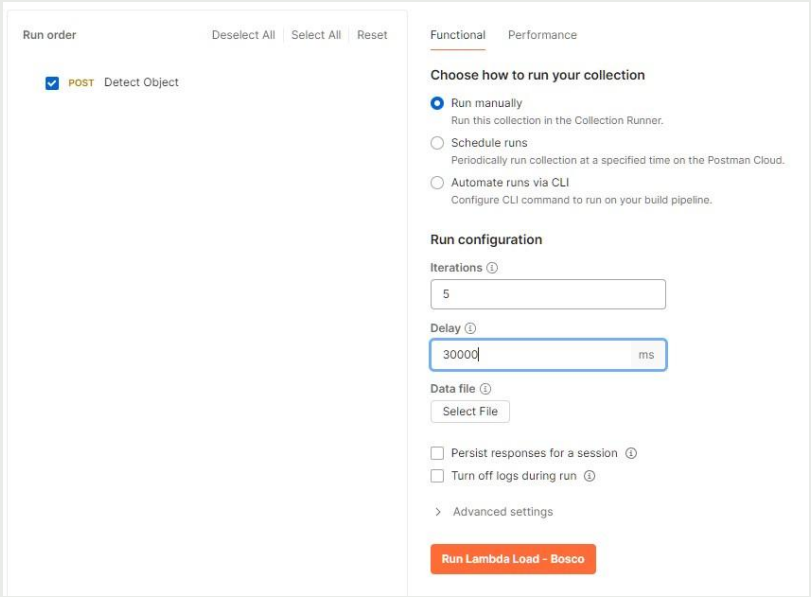
Unfortunate Event

- Unfortunately we lost our AWS account twice (once not knowing the consequence of large concurrent API call, then trying to gradual increase and decrease.
- We also lost one of our group member unfortunately in the last minute because of schedule mismatch.
- **So we kept the number very conservative during the stress: 1, 2, 4, 2, 1 concurrent requests 30 seconds apart**

Stress Lambda Function

Postman Collection Runner

- We ran the postman Collection Runner to execute the pre-script

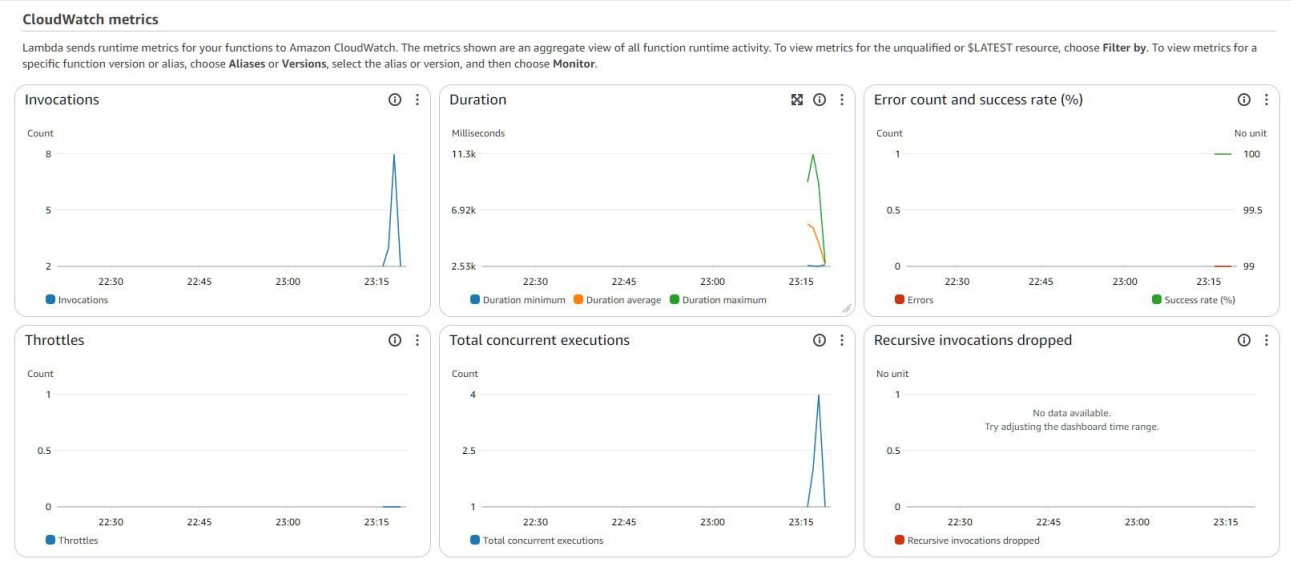



```
// Get the current iteration and calculate the request count
const currentIteration = pm.info.iteration;
const totalIterations = pm.info.iterationCount; // Automatically retrieves the total iterations

// Calculate the number of requests
let requestCount;
if (currentIteration <= Math.floor(totalIterations / 2)) {
    // Increasing in the first half
    requestCount = Math.pow(2, currentIteration);
} else {
    // Decreasing in the second half
    requestCount = Math.pow(2, totalIterations - currentIteration - 1);
}

console.log(`Iteration ${currentIteration + 1} of ${totalIterations}: Sending ${requestCount}
    concurrent requests.`);

// Send multiple requests
for (let i = 0; i < requestCount; i++) {
    pm.sendRequest({
        url: apiUrl,
        method: "POST",
        header: headers,
        body: {
            mode: "raw",
            raw: requestBody,
        },
    }, (err, response) => {
        if (err) {
            console.error(`Request ${i + 1} failed:`, err);
        } else {
            console.log(`Response from request ${i + 1}:`, response);
        }
    });
}
```

CloudWatch Report

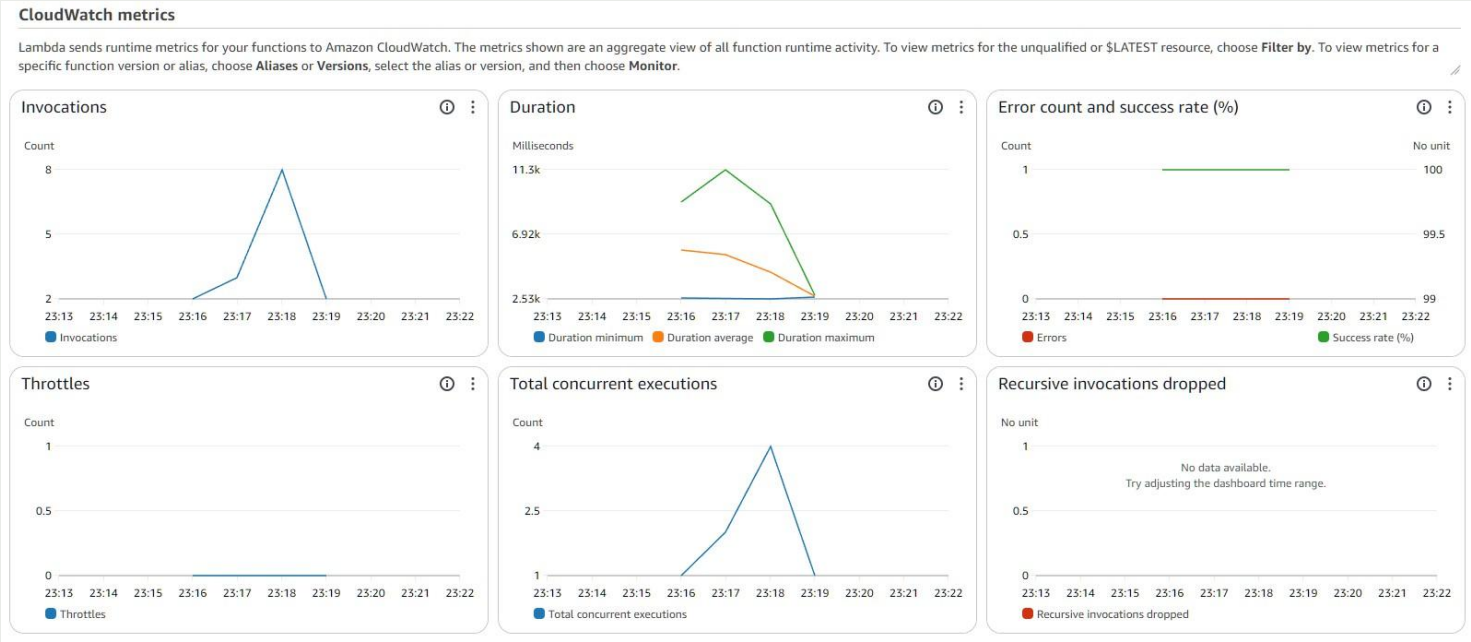
CloudWatch Metrics to Observe the Deployment Scale

- We observed that the number of **Total Concurrent Execution** increases with the higher number of Lambda Function invocation.
- Also when the invocation is decreased the **Total Concurrent Execution** number is decreased.
- Iteration delay in-between was **30 seconds**



CloudWatch Report

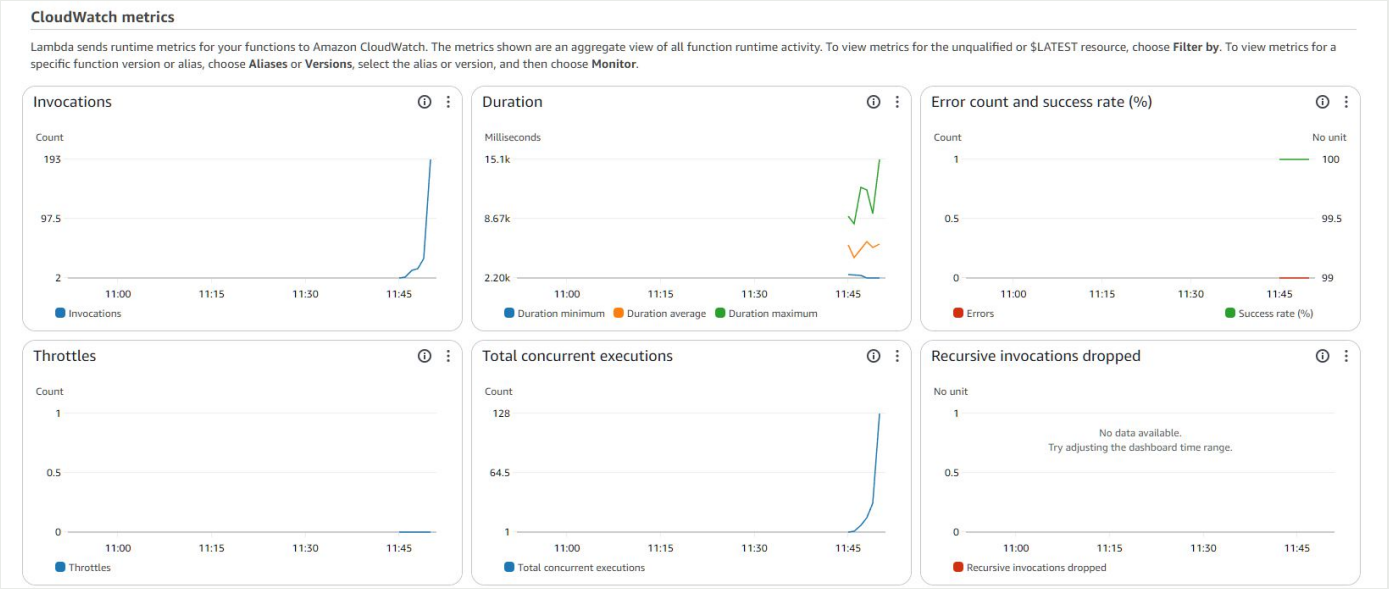
Closer Look of Deployment Scale Out and In



CloudWatch Report

CloudWatch Metrics to Observe the Deployment Scale

- Account crashed before decrement could be observed. But we can compare with the previous run.



Thank You!