AWS Deployment Project COMP4349_A2
510664228

1. Introduction

This serverless AWS image annotation platform delivers a robust, scalable, cloud-native image annotation platform on AWS, combining traditional compute and serverless approaches for optimal cost and performance. Users upload images via a web app using the ALB DNS endpoint; captions are automatically generated using Gemini AI, and thumbnails are produced in near-real time, with both stored and retrievable via a gallery interface. The architecture ensures high availability, strong decoupling, and auto scaling via managed AWS services and best-practice security isolation by integration of Lambda functions and ASGs with scaling policies.

2. Architecture Diagram

Integration between Components

<u>How the Web Application Triggers Lambda Functions</u>
The user uploads an image via a web app hosted on the private EC2 instance behind the ALB by uploading an image. The web app is accessible via the ALB DNS endpoint and is public to everyone. The web application uploads the image file to the S3 bucket under the uploads/ prefix, and since the S3 bucket is configured with an EventBridge rule that matches the S3 ObjectCreated events in the uploads/ prefix, it triggers the target groups, which are the 2 lambda functions and the SNS topic. Annotate lambda parses the event notification with the key and the bucket and reads the uploaded image, calls the Gemini API to generate a caption, and writes the caption to the shared RDS database. The thumbnail lambda also parses the event notification and gets the bucket and the key and reads the uploaded image, converts it to RGB for JPEG consistency, and writes a generated thumbnail to the thumbnails/ prefix in the same S3 bucket image-bucket-2508. It also sends the notification via SNS topic to my personal email.

<u>How Both Components Access Shared Resources</u>
S3 bucket: Both the web app private EC2 instances and Lambda functions read from and write to the same S3 bucket. The web app uploads images to uploads/ and displays images and thumbnails to users. The Lambda functions read images from uploads/ and write thumbnails to thumbnails.
RDS database: The web app writes metadata such as the filename, user, timestamp, etc., and displays captions in the gallery. At the same time, the Annotate Lambda, which has a security group configured with it that allows it to access the RDS, writes the Gemini AI-generated caption to the same RDS table, which is then read by the web app for display.
IAM Roles: Both the EC2 web app and Lambdas have specific IAM roles with the minimum necessary permissions to access only their required resources. Lambda uses LabRole, while EC2 instances use EMR_EC2_DefaultRole.
Networking and SGs: EC2 and Annotate Lambda use SGs and VPC settings to connect securely to the RDS. Both Lambda functions are granted access via LabRole to access the S3 bucket, but only Annotate has the SG to access the RDS.
<u>WALKTHROUGH:</u> User accesses app via ALB DNS, it uploads via web app form, and it stores the image in S3 bucket uploads/ and stores initial metadata in RDS. S3 generates an ObjectCreated event to trigger EventBridge, which invokes both Annotate and Thumbnail Lambdas to write the generated caption to RDS and the generated thumbnail to S3's thumbnails/ for the web app to display. Web app gallery page periodically queries RDS for updated captions and S3 for available thumbnails, displaying the results to the user. Finally SNS sends a notification.

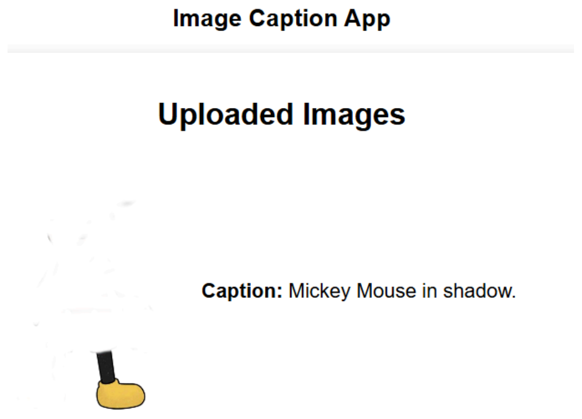


Image Caption App

Uploaded Images

Caption: Mickey Mouse in shadow.

Figure 1: Gallery route, showing thel (right_leg) and its generated captions, both resulting from Lambda functions.

Web Application Architecture

The web application diagram is rotated to fit the large picture. That way you could clearly see everything. Please have a good look at it. 🙂

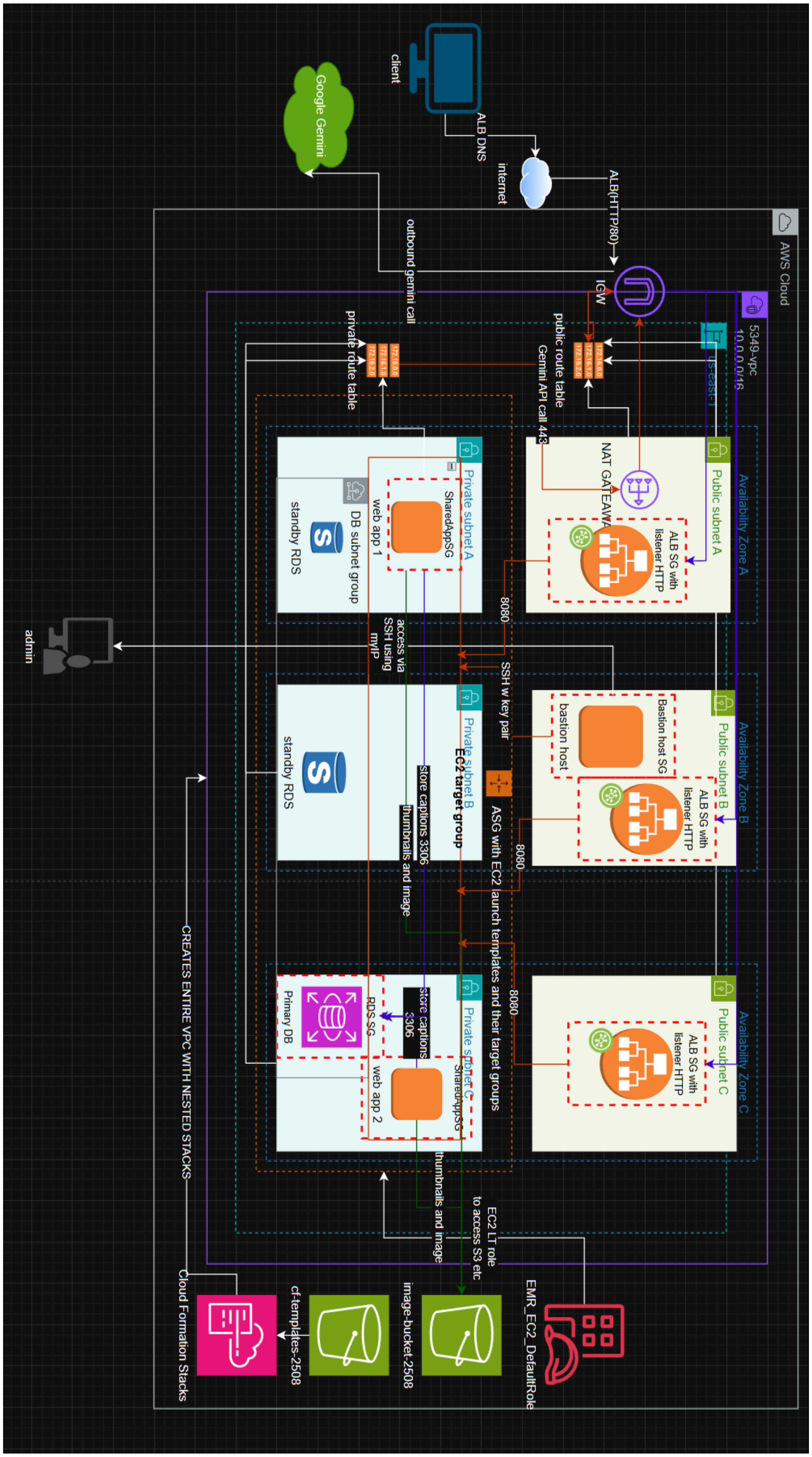


Diagram 1: Web Application Architecture with all routes and flows.

Serverless Architecture

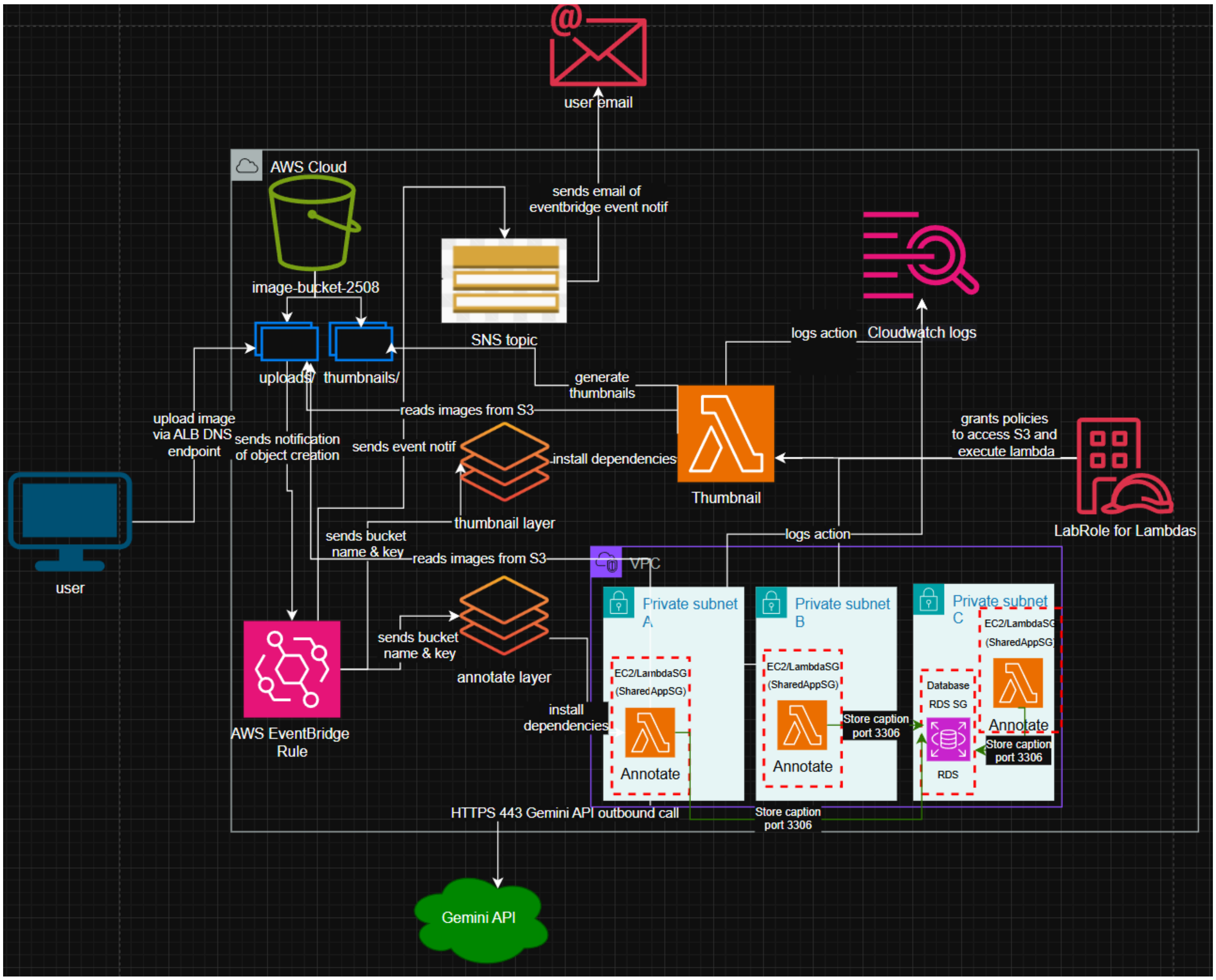


Diagram 2: Serverless Architecture Diagram with all workflows and routes.

1. Web Application Deployment

Compute Environment

All of these components are inside my newly created comp5349-vpc that's public with 3 public subnets and 3 private subnets and public/private route tables for each 3 public/private subnets, respectively. The public route tables are connected to the Internet Gateway, and private route tables are connected to the NAT gateway in the public subnet.

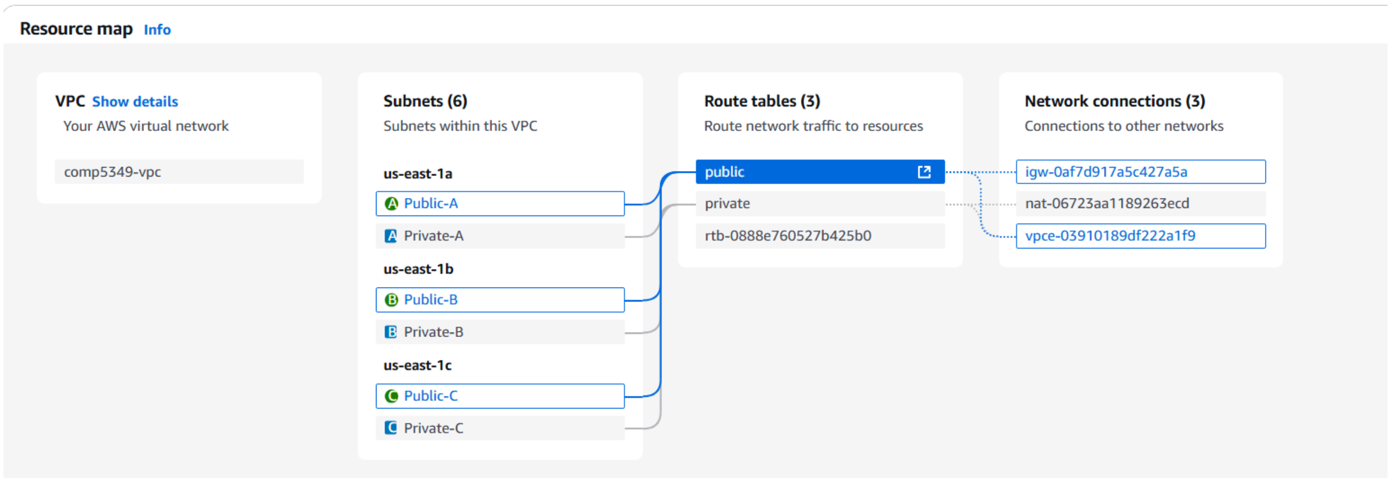


Figure 2: Resource map of VPC showing public routings

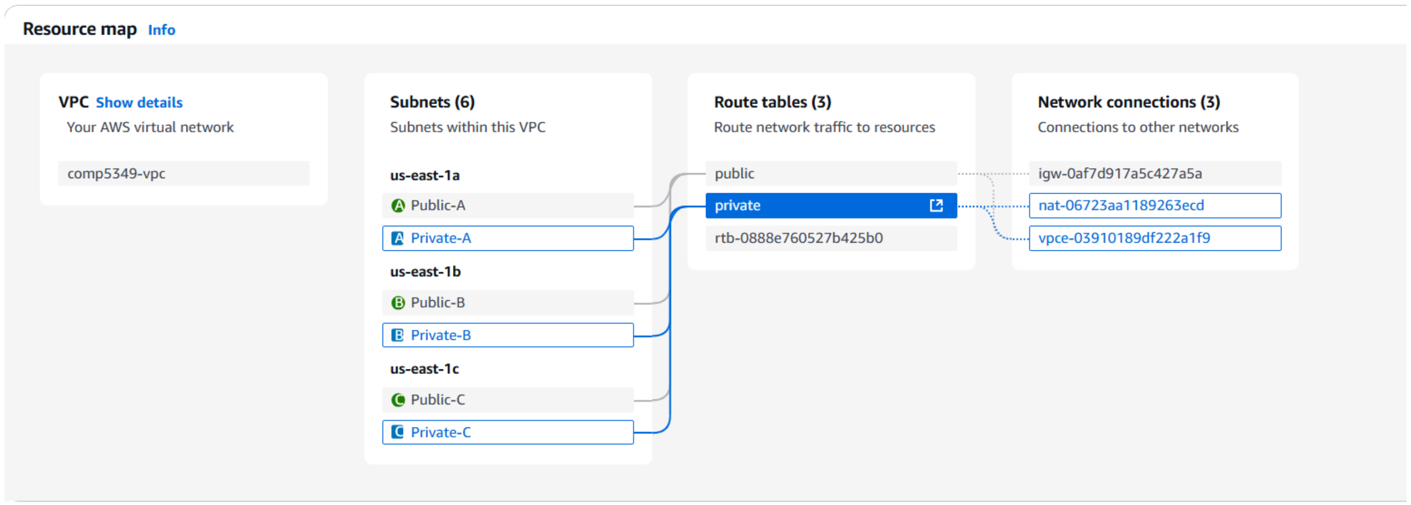
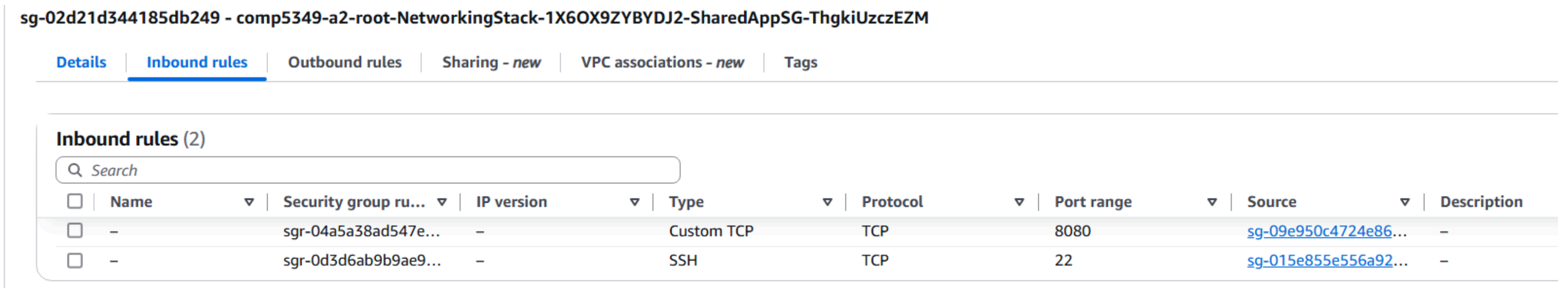


Figure 3: Resource map of VPC showing private routings

Security Groups

EC2/Lambda SG (SharedAppSG)

- Inbound: custom TCP on port 8080 (flask) from ALB SG
- Inbound: SSH on port 22 from Bastion SG
- Outbound: All traffic


sg-02d21d344185db249 - comp5349-a2-root-NetworkingStack-1X6OX9ZYBYDJ2-SharedAppSG-ThgkiUzczEZM

Inbound rules (2)

Name	Security group ru...	IP version	Type	Protocol	Port range	Source	Description
–	sgr-04a5a38ad547e...	–	Custom TCP	TCP	8080	sg-09e950c4724e86...	–
–	sgr-0d3d6ab9b9ae9...	–	SSH	TCP	22	sg-015e855e556a92...	–

Figure 4: Updated Inbound rules (the video contained the older version but these are the most updated rules)

RDS database SG

- Inbound: MySQL on port 3306 from EC2/Lambda SG
- Outbound: all traffic

ALB/DNS SG

- Inbound: HTTP on port 80 from Internet
- Outbound: all traffic

Bastion SG

- Inbound: SSH on port 22 from my personal IP
- Outbound: all traffic

IAM Roles and Secrets Manager

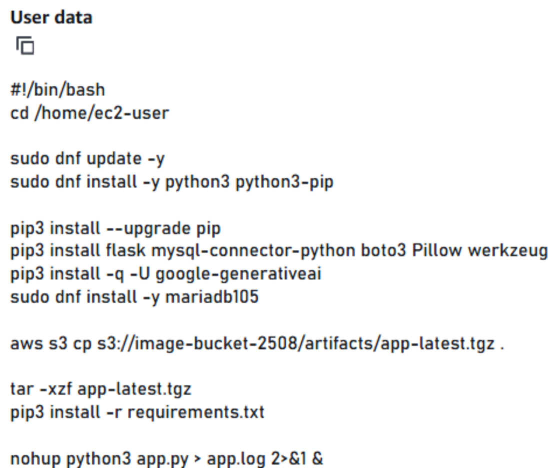
My launch template uses the **EMR_EC2_DefaultRole,** as it has the policy AmazonElasticMapReduceforEC2Role to allow it to access S3, CloudWatch, and other services like SNS, which I used here.

My Lambda roles use the **LabRole** as specified in Ed, so I could have the AWS Lambda basic execution policy and access the S3 bucket and all other ones. I couldn't list them down as I don't have access to the listed policies.

Secrets Manager: I also used Secrets Manager to manage my RDS. They are created once my nested database stack runs on CloudFormation, creating all the required variables as well, like the RDS MySQL instance, security groups, and all that. Its credentials are used in the web app code and to access the database via the private EC2 instance.

EC2 Compute configs

Launch Template: I'm using the most updated version, 4, with AMI being Amazon Linux 2023 and the security group being SharedAppSG for EC2 and Lambda, as shown above. The instance type is t3.micro, and it has the key pair name "my-ec2-key-new" for bastion hosts to SSH into. The LT has user data as shown:



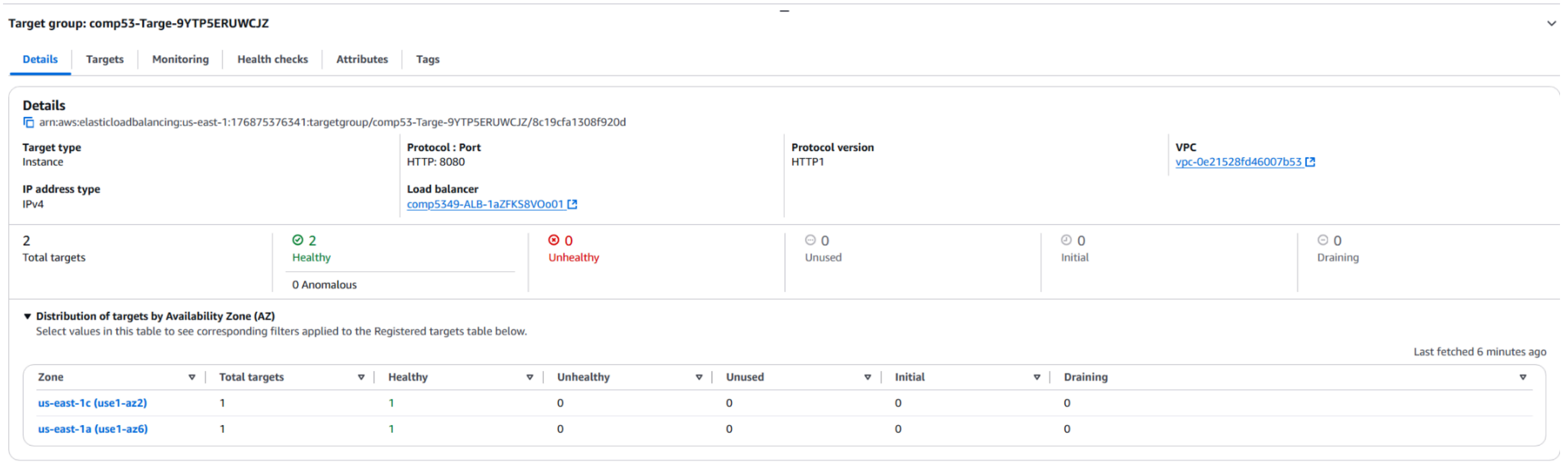
```
#!/bin/bash
cd /home/ec2-user

sudo dnf update -y
sudo dnf install -y python3 python3-pip

pip3 install --upgrade pip
pip3 install flask mysql-connector-python boto3 Pillow werkzeug
pip3 install -q -U google-generativeai
sudo dnf install -y mariadb105

aws s3 cp s3://image-bucket-2508/artifacts/app-latest.tgz .

tar -xzf app-latest.tgz
pip3 install -r requirements.txt

nohup python3 app.py > app.log 2>&1 &
```

Figure 5: User Data for Launch Template

What this user data does ultimately is configure all EC2 instances to have the proper dependencies and folders in order to run the web app in the background using nohup. It gets the folder by copying it from the S3 bucket called app-latest.tgz and unzipping it, installing the requirements inside the folder, and once the instance is initialized, then any user can access the web app via ALB DNS without any admin needing to start it by running python3 app.py.

EC2 ASG configs: It has a desired cap of 2, and scaling limits are 2-4 based on the traffic loads. It's attached to the load balancer target group, which controls the healthy/unhealthy instances. Its dynamic scaling policy is target tracking, and it's enabled to scale in with its trigger being an average CPU utilization metric above 30%. Actions are to scale out if it exceeds 30% for 3 minutes and 3 data points and scale in if it's below the alarm threshold, which is 21 for 15 minutes for 15 data points. Instances also need only 30 seconds to warm up before being shown in metrics for faster feedback. It also automatically launches instances whenever some of them are terminated using the Launch Template. The distribution uses balanced best effort with a prioritized on-demand allocation strategy and a capacity-optimized spot allocation strategy.

Target Group: Most of the times when the instances are idle, it contains only 2 instances based on the ASG desired cap of 2, with both of them being healthy. Instances are separated in different AZs, and health checks are conducted based on their protocol (HTTP), path (/), and port (8080). It has a healthy threshold of 5 successes (200 code), and if it fails twice, then it's unhealthy, and then the user can't use that instance, as it doesn't have proper routing, so they are redirected to healthy instances, with 1 being the minimum at all times. It uses the round robin algorithm to distribute loads so that no instances are idle when another is having a lot of traffic load.



Target group: comp53-Targe-9YTP5ERUWCJZ

Details | Targets | Monitoring | Health checks | Attributes | Tags

Details

arn:aws:elasticloadbalancing:us-east-1:176875376341:targetgroup/comp53-Targe-9YTP5ERUWCJZ/8c19cfa1308f920d

Target type Instance	**Protocol : Port** HTTP: 8080	**Protocol version** HTTP1	**VPC** vpc-0e21528fd46007b53
IP address type IPv4	**Load balancer** comp5349-ALB-1aZFKS8VOo01		

2 Total targets	✓ 2 Healthy / 0 Anomalous	⊗ 0 Unhealthy	0 Unused	0 Initial	0 Draining

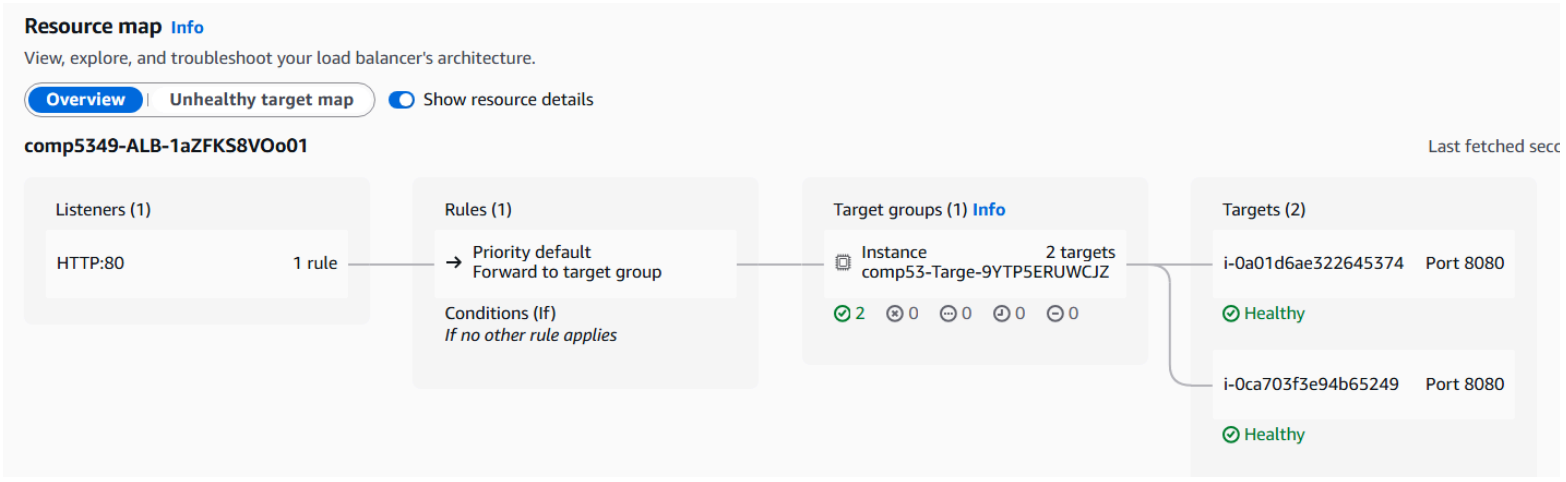
▼ **Distribution of targets by Availability Zone (AZ)**
Select values in this table to see corresponding filters applied to the Registered targets table below.

Last fetched 6 minutes ago

Zone	Total targets	Healthy	Unhealthy	Unused	Initial	Draining
us-east-1c (use1-az2)	1	1	0	0	0	0
us-east-1a (use1-az6)	1	1	0	0	0	0

Figure 6: Target group configs

ALB: It's internet-facing with all 3 public subnets inside the 5349- vpc and it's covering 3 availability zones, which instances can be placed in. It contains the DNS endpoint where any user can access the web app from. It has a listener on port 80 and will forward the connection request to the target group above on port 8080, where the app is hosted on (Flask). It has the target group stickiness off so that any instance can handle any request, so each new request from a user can go to any healthy target in the target group, even if it's the same client, and it also allows the ALB to pick a target for each request based on the round robin algorithm. Uses ALB SG



Resource map Info

View, explore, and troubleshoot your load balancer's architecture.

comp5349-ALB-1aZFKS8VOo01

Last fetched secc

Listeners (1): HTTP:80 — 1 rule

Rules (1): Priority default — Forward to target group; Conditions (If) — *If no other rule applies*

Target groups (1) Info: Instance comp53-Targe-9YTP5ERUWCJZ — 2 targets; ✓ 2, 0, 0, 0, 0

Targets (2):
i-0a01d6ae322645374 Port 8080 — ✓ Healthy
i-0ca703f3e94b65249 Port 8080 — ✓ Healthy

Figure 7: Resource map for ALB with listener rules and target groups

How it ensures high availability: My ALB setup is deployed across 3 public subnets, each in a different AZ within the VPC, to ensure that traffic is always routed to at least one healthy EC2 instance even if the other AZs fail. The ASG also launches instances in multiple private subnets in different AZs so there is always redundancy and no single point of failure. The ALB also continuously monitors all EC2 instances with health checks on the path, port, and protocol, ensuring unhealthy instances don't get used by users so they would never experience downtime due to a failed instance. The scaling policy integration also makes use of the CPU utilization to scale in and out whenever CPU usage exceeds 30% for some time. New instances registered with the ALB also only start receiving traffic after they pass the health checks in the target group config. This setup ensures high availability.

How it ensures traffic distribution: My ALB forwards all HTTP connection requests on port 80 straight to the target group of EC2 instances on port 8080. It's configured with round robin load balancing to distribute incoming requests evenly among all healthy instances to prevent any instance from being a bottleneck and ensure optimal resource usage. Target group stickiness is also off, so each new client request can be routed to any available healthy instance. This maximizes the ALB's ability to balance the load, especially as instances are added or removed during scaling events. Combining this with my launch template, which configures each instance with all dependencies and files on startup, makes them register faster for health checks with the ALB and start handling user traffic properly.

Administrative Access

Bastion Host: Deployed in a public subnet that is accessible only via SSH from my IP, and I could SSH into my private EC2 instance via key pair and its private IP address for private subnet management. T3.micro, Amazon Linux 2023 with Bastion SG.

Database Environment

Created in the VPC 5349-vpc in a private subnet. It's using the database SG shown above so that only private EC2 instances can connect to it. Its credentials are stored in the Secrets Manager as well for better security. The RDS has the retain deletion policy so that it's not accidentally deleted with t2.micro storage, MySQL engine version 8.0. It's multi-AZ configured with encrypted storage and is not publicly accessible. It's only accessed via the SharedAppSG shown above.
It's accessed by the web app by placing the RDS credentials in app.py and create-database.sh to create the MySQL RDS instance with the database image_caption_db and table captions.

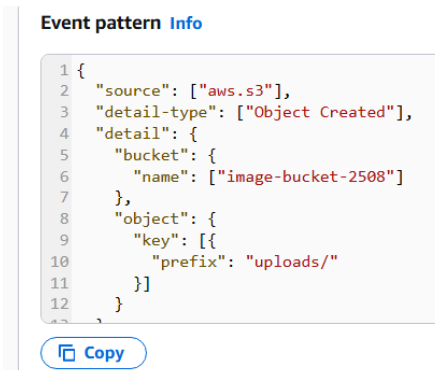
Storage Environment

I have 2 buckets for clearer isolation. One of them is cf-templates-2508, which stores my CloudFormation templates that I use to run the stacks in. The other one is image-bucket-2508 for the uploads, thumbnails, and artifacts for lambda layers and app code. I don't have a direct S3 trigger, but I've modified the bucket to use EventBridge notifications so all events that happen in the bucket could be sent to the AWS EventBridge rule on prefix uploads/ to trigger the 2 lambdas, which will store thumbnails in the image-bucket-2508 and store captions in the RDS.

2. Serverless Component Deployment

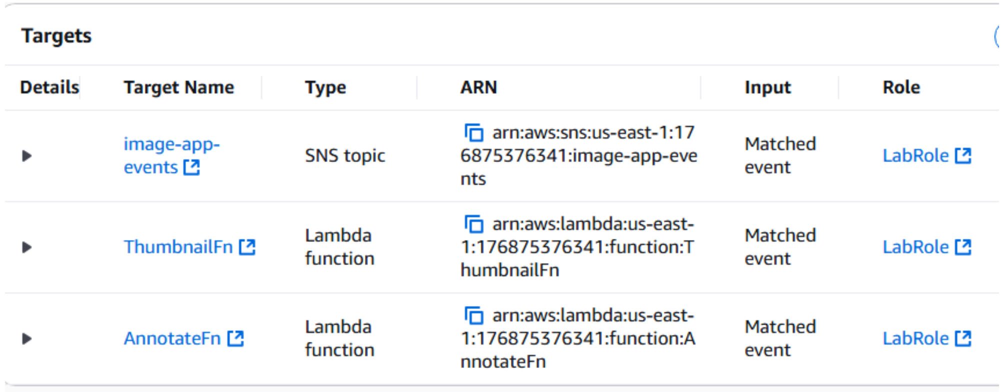
Event Driven Architecture

S3 object created in uploads/ in bucket image-bucket-2508 is the rule for my EventBridge called “ImageAppS3UploadEvents”. Its targets include Annotate, Thumbnail, and SNS topic on my email.



Event pattern Info

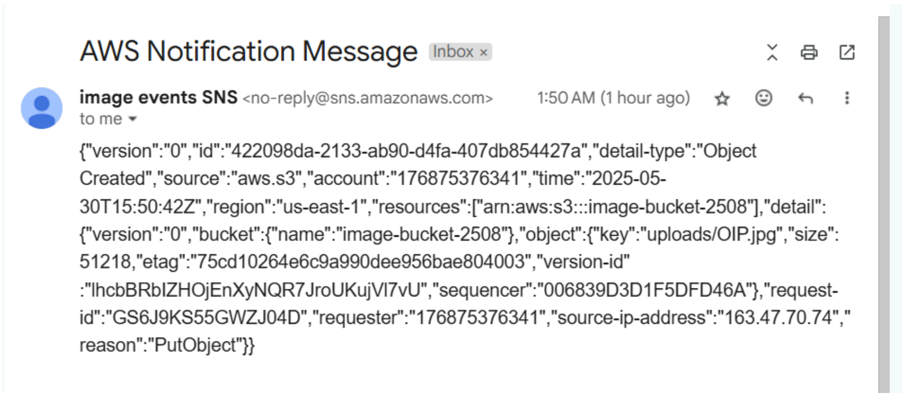
```
{
  "source": ["aws.s3"],
  "detail-type": ["Object Created"],
  "detail": {
    "bucket": {
      "name": ["image-bucket-2508"]
    },
    "object": {
      "key": [{
        "prefix": "uploads/"
      }]
    }
```



Targets

Details	Target Name	Type	ARN	Input	Role
▸	image-app-events	SNS topic	arn:aws:sns:us-east-1:176875376341:image-app-events	Matched event	LabRole
▸	ThumbnailFn	Lambda function	arn:aws:lambda:us-east-1:176875376341:function:ThumbnailFn	Matched event	LabRole
▸	AnnotateFn	Lambda function	arn:aws:lambda:us-east-1:176875376341:function:AnnotateFn	Matched event	LabRole

Figures 8 & 9: EventBridge Configurations (pattern and targets) for S3Object created events.



AWS Notification Message Inbox ×

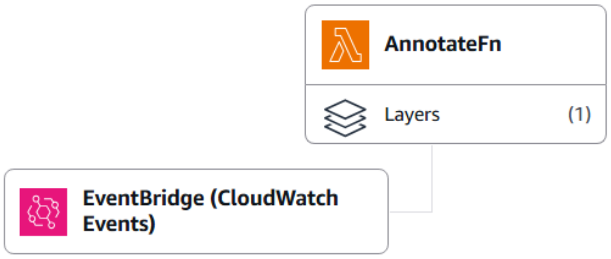
image events SNS <no-reply@sns.amazonaws.com> 1:50 AM (1 hour ago)
to me

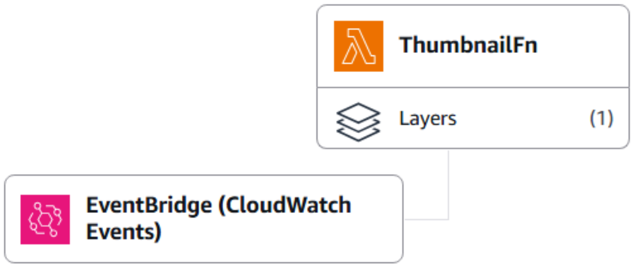
{"version":"0","id":"422098da-2133-ab90-d4fa-407db854427a","detail-type":"Object Created","source":"aws.s3","account":"176875376341","time":"2025-05-30T15:50:42Z","region":"us-east-1","resources":["arn:aws:s3:::image-bucket-2508"],"detail":{"version":"0","bucket":{"name":"image-bucket-2508"},"object":{"key":"uploads/OIP.jpg","size":51218,"etag":"75cd10264e6c9a990dee956bae804003","version-id":"lhcbBRbIZHOjEnXyNQR7JroUKujVl7vU","sequencer":"006839D3D1F5DFD46A"},"request-id":"GS6J9KS55GWZJ04D","requester":"176875376341","source-ip-address":"163.47.70.74","reason":"PutObject"}}

Figure 10: SNS topic triggered by EventBridge notification to my email.

Lambda Layering

Both Lambda functions are deployed as zip files with the dependencies being on a Lambda layer above the lambda_function code. It has the same runtime, and the way that it works is that it takes the layer-a.zip or layer-t.zip based on the lambda used, and inside it includes a Python directory that contains the required dependencies needed for each lambda function. Both lambda functions and layers use Python 3.12 runtime for compatibility.





Figures 11 & 12: Annotate & Thumbnail Lambda Function

Lambda Annotation Function

It runs on a layer with installed dependencies to run the code, which is triggered by the EventBridge rule shown above.
This function uses the hard-coded environmental variables in the Lambda environment to access the S3 bucket and RDS. The function firstly handles the event bridge to get the bucket and key and uses the Gemini API with a valid API key to generate captions based on the object uploaded and generates an image caption to be written to the RDS by connecting to it using RDS credentials. Deployed through the AWS console manually.

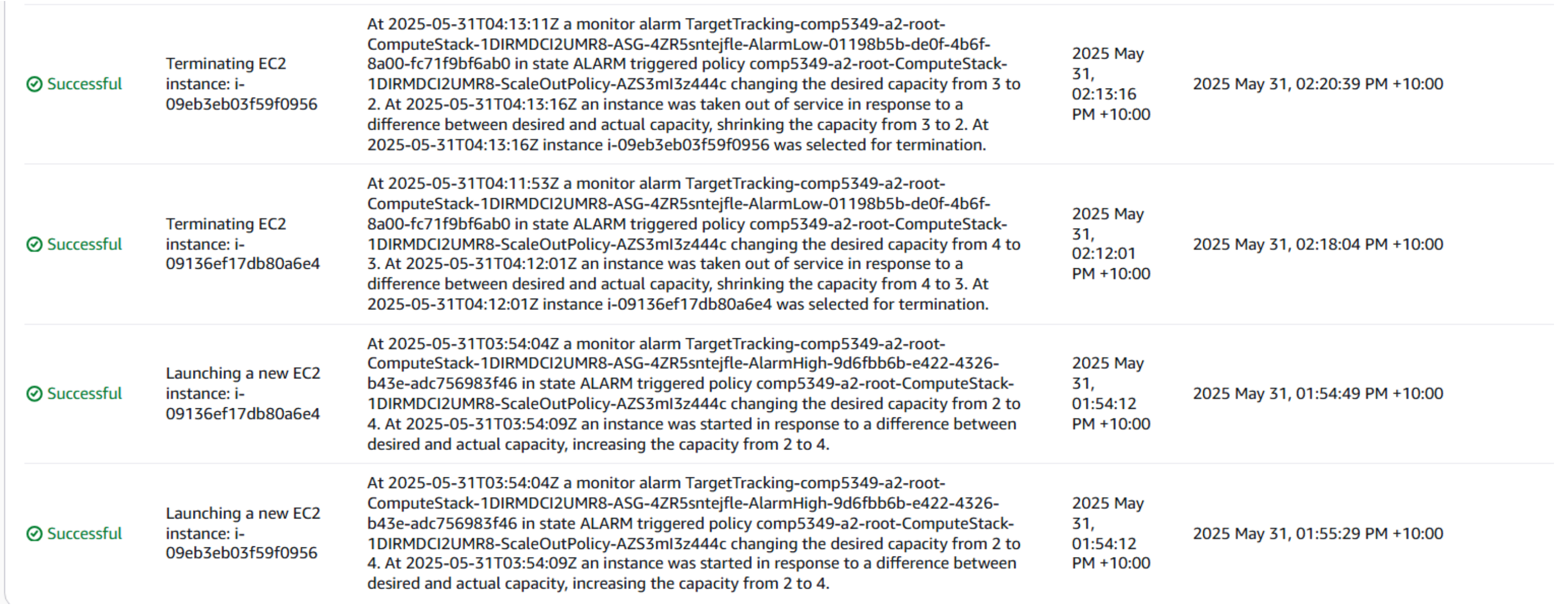
Since this lambda has to connect to the RDS, it's inside the private subnets of the VPC with the security group SharedAppSG to write to the RDS.

Lambda Thumbnail Generator Function

It runs on a layer with installed dependencies to run the code, which is triggered by the EventBridge rule shown above.
This function uses just one bucket environmental variable to access the S3 bucket to place the thumbnails in the thumbnails/ prefix. It firstly gets the bucket and key from the EventBridge event structure to process the valid images uploaded and uses Pillow in the layer with boto3 to create the thumbnail, convert it to RGB for JPEG compatibility, and save it to the bucket's thumbnails/ prefix. This lambda doesn't have a VPC connected, as its functions are not required to be within the VPC. Deployed through the AWS console manually.

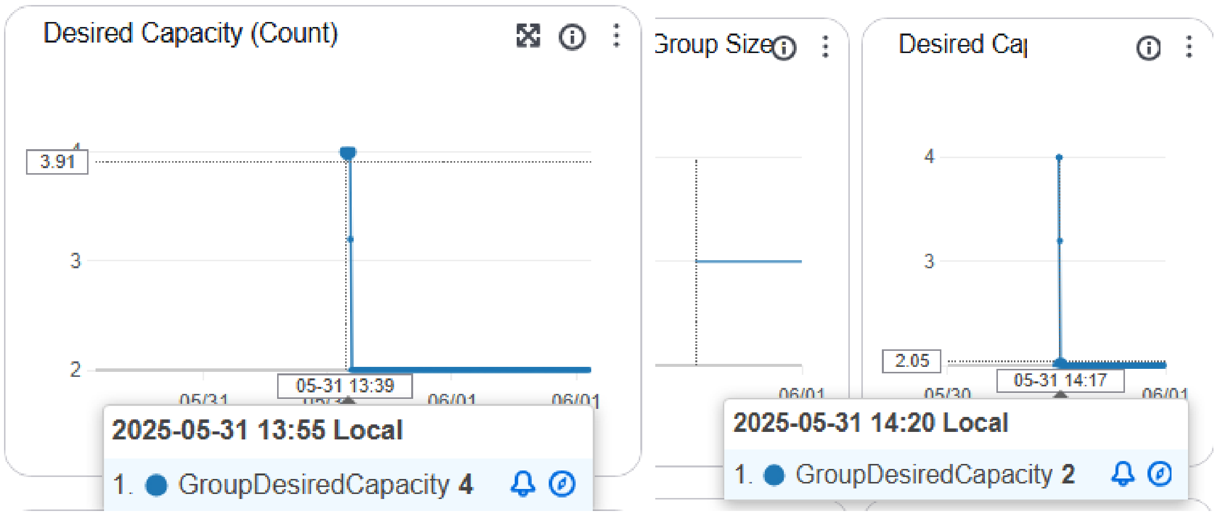
3. Auto Scaling Test Observation

The Auto Scaling test is conducted using ApacheBench on my local PC, using this command, "ab -n 5000 -c 100 http://<alb-dns-endpoint>/gallery", to send out 5000 requests in total with 100 concurrent requests on the busiest route, /gallery, where it has to access the RDS every time the route is visited. It increases the CPU utilization up to more than 30%, which then triggers the ASG scaling policy to scale out to 4 instances. I did it a few times, but at first it didn't seem to trigger a lot of the CPU at the /uploads route, so I changed it to /gallery for more CPU usage.

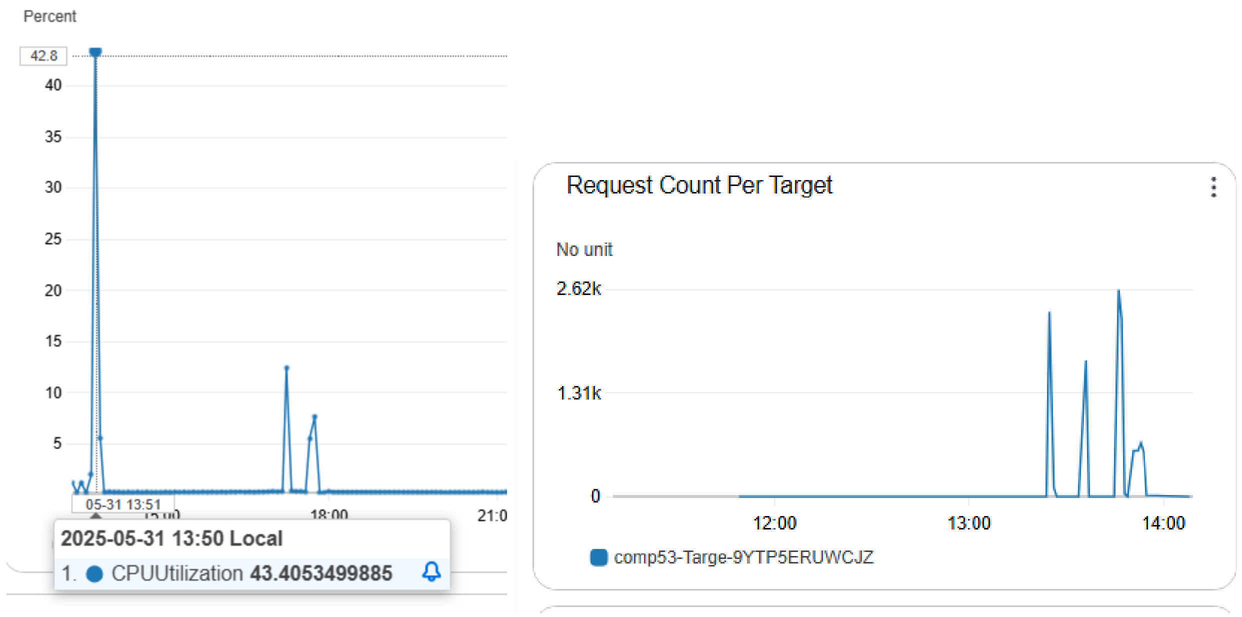


✓ Successful	Terminating EC2 instance: i-09eb3eb03f59f0956	At 2025-05-31T04:13:11Z a monitor alarm TargetTracking-comp5349-a2-root-ComputeStack-1DIRMDCI2UMR8-ASG-4ZR5sntejfle-AlarmLow-01198b5b-de0f-4b6f-8a00-fc71f9bf6ab0 in state ALARM triggered policy comp5349-a2-root-ComputeStack-1DIRMDCI2UMR8-ScaleOutPolicy-AZS3mI3z444c changing the desired capacity from 3 to 2. At 2025-05-31T04:13:16Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 3 to 2. At 2025-05-31T04:13:16Z instance i-09eb3eb03f59f0956 was selected for termination.	2025 May 31, 02:13:16 PM +10:00	2025 May 31, 02:20:39 PM +10:00
✓ Successful	Terminating EC2 instance: i-09136ef17db80a6e4	At 2025-05-31T04:11:53Z a monitor alarm TargetTracking-comp5349-a2-root-ComputeStack-1DIRMDCI2UMR8-ASG-4ZR5sntejfle-AlarmLow-01198b5b-de0f-4b6f-8a00-fc71f9bf6ab0 in state ALARM triggered policy comp5349-a2-root-ComputeStack-1DIRMDCI2UMR8-ScaleOutPolicy-AZS3mI3z444c changing the desired capacity from 4 to 3. At 2025-05-31T04:12:01Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 4 to 3. At 2025-05-31T04:12:01Z instance i-09136ef17db80a6e4 was selected for termination.	2025 May 31, 02:12:01 PM +10:00	2025 May 31, 02:18:04 PM +10:00
✓ Successful	Launching a new EC2 instance: i-09136ef17db80a6e4	At 2025-05-31T03:54:04Z a monitor alarm TargetTracking-comp5349-a2-root-ComputeStack-1DIRMDCI2UMR8-ASG-4ZR5sntejfle-AlarmHigh-9d6fbb6b-e422-4326-b43e-adc756983f46 in state ALARM triggered policy comp5349-a2-root-ComputeStack-1DIRMDCI2UMR8-ScaleOutPolicy-AZS3mI3z444c changing the desired capacity from 2 to 4. At 2025-05-31T03:54:09Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 2 to 4.	2025 May 31, 01:54:12 PM +10:00	2025 May 31, 01:54:49 PM +10:00
✓ Successful	Launching a new EC2 instance: i-09eb3eb03f59f0956	At 2025-05-31T03:54:04Z a monitor alarm TargetTracking-comp5349-a2-root-ComputeStack-1DIRMDCI2UMR8-ASG-4ZR5sntejfle-AlarmHigh-9d6fbb6b-e422-4326-b43e-adc756983f46 in state ALARM triggered policy comp5349-a2-root-ComputeStack-1DIRMDCI2UMR8-ScaleOutPolicy-AZS3mI3z444c changing the desired capacity from 2 to 4. At 2025-05-31T03:54:09Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 2 to 4.	2025 May 31, 01:54:12 PM +10:00	2025 May 31, 01:55:29 PM +10:00

Figure 13: CloudWatch Alarm trigger to scale out to 4 and scale in back to 2.



Figures 14 & 15: Scaling out to 4 from 2 instances at 1:55pm, then scaling back in to 2, just like Figure 10 shows.



Figures 16 & 17: CPU utilization metrics triggering the ASG scaling policy, and load is being distributed so each instance has some traffic distributed.

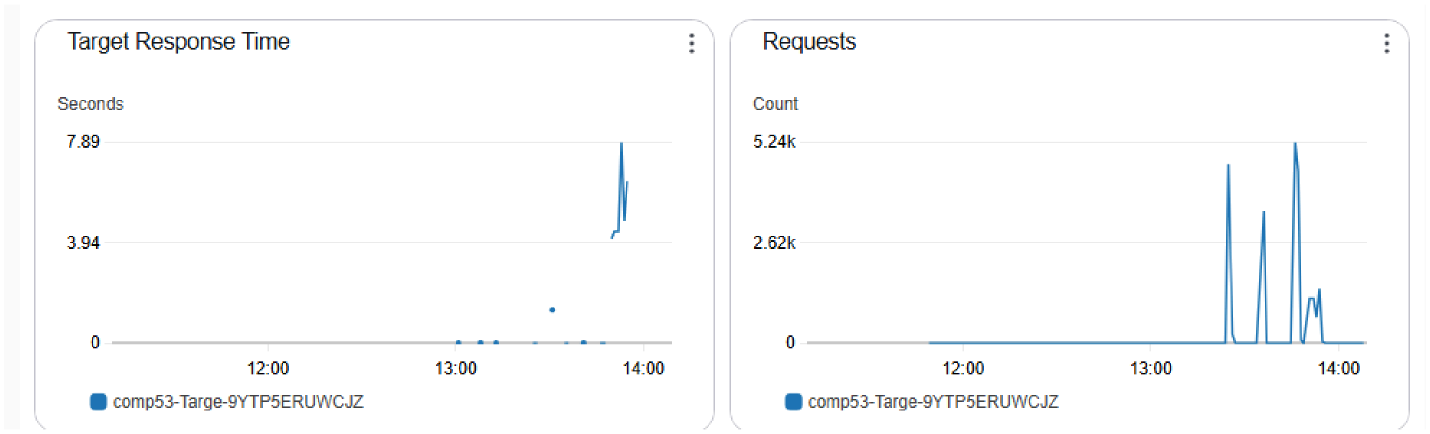


Figure 18: ASG EC2 metrics on the long response times and the number of requests.

4. Summary and Lessons Learned

Key Findings and Lessons Learnt

During this assignment, I built and deployed a fully integrated AWS-based serverless image annotation platform. Throughout the process, I encountered several technical and operational challenges that helped deepen my understanding of AWS services and cloud-native architecture.

1. One of the toughest parts was learning how to design and troubleshoot **CloudFormation** stacks. Syntax errors, missing dependencies, and strict parameter requirements led to repeated stack failures. Through trial and error (and about 20 iterations!), I learned to modularize templates, use outputs for passing resource references, and leverage stack policies to avoid accidental deletions (especially with RDS). Nested stacks and "commenting out" resources for testing became essential.

2. I learned the hard way that RDS resources with a "**retain**" **deletion policy** are not removed when stacks are deleted. This led to leftover resources and stack errors. The solution was to manually clean up retained resources before stack recreation and to document which resources had persistent state.

3. As my Learner Lab account **restricted custom roles and policies**, I struggled to assign correct permissions for Lambda and EC2. Attempts to create roles failed, so I switched to using pre-existing LabRole and EMR_EC2_DefaultRole, even though documentation was sparse. I learned to read environmental constraints, adapt to what's available, and not waste time fighting the platform.

4. **Networking** was a big challenge, requiring proper isolation for security and connectivity for integration. I had to repeatedly test and adjust security group rules, NAT Gateway, and route tables to enable Lambda-to-RDS communication, ALB-to-EC2 access, and Bastion connectivity. Least privilege and separation of duties guided my decisions.

5. **CloudWatch Logs** and metrics became essential for debugging failed Lambda executions, stack events, and Auto Scaling policies. I learned to set alarms for CPU utilization and to analyze logs for permission or connectivity issues.

6. **Lambda configurations** were also new to me. I did know that I have to use a lambda layer, but it always has to be in the same runtime as the code, which is new to me. Finding out that I have to package the zip layer with a Python directory instead of just the dependencies was also new to me. Setting up environmental variables in Lambda was also new to me and the VPC as well.

7. Once both my lambda functions were configured, I had to find a way to trigger both of them with the same event, which was tricky until I found out about **EventBridge**, which was a lifesaver, as all I had to do was set targets, and it triggers it based on objects created on S3 notification. I also found out how to parse the EventBridge structure to get the bucket and the key for my function to work as well.

8. Simulating high traffic with **ApacheBench** helped verify scaling behavior. Watching new EC2 instances launch and terminate in response to real-time metrics reinforced the value of infrastructure as code and automation.

9. **Decoupling** via **S3** and **EventBridge** simplified troubleshooting, scaling, and future extensibility. Modular design and least-privilege access were crucial to achieving a production-ready, secure system.

Summary

This project demonstrated E2E deployment of a secure, available, and automated web app by combining EC2 with Lambda for performance and cost. ALB, ASG, RDS, S3, Lambdas, and EventBridge were properly configured for the best user experience and auto-scaling policies. This hands-on experience reinforced the value of infrastructure as code, modular design, and the least access privilege principle. Real-world problems like IAM policy limits and VPC networking challenges forced my creative problem-solving skills and boosted my confidence in architecting cloud apps using AWS.