Final Project
Cloud Computing Track

Ala'a Abdelqader

Supervised by:
Eng.Yazan AlShannik
Soft Skills Instructor: Dr. Sawsan Sayeh
English Language Instructor: Dr. Basheer Mufleh

AlHussein Technical University (HTU)

June/2025

Table of Contents

Table of Figures

1.Introduction

The **ICT Upskilling Program** is a training initiative organized by the Centre of Professional Development and Community Outreach. It's designed to help participants build real, hands-on experience in cloud computing, manual software testing, soft skills, and business English. Under the guidance of industry instructors like Eng. Yazan Al-Shannik, Dr. Sawsan Alsayeh, and Dr. Basheer Mufleh, this program blends technical learning with communication and presentation skills to prepare us for real-world IT careers.

a. Objective

The main goal of the capstone is to prove that we can design, build, and explain cloud-based solutions using AWS services. We're expected to apply our technical skills (like creating secure infrastructure and automating tasks), demonstrate soft skills (like presenting clearly), and communicate professionally in English.

b. Overview of the Four Scenarios

1. **Hosting My Portfolio on AWS**
   I build and host a personal portfolio website using Amazon S3. It's delivered securely through CloudFront and linked to a custom domain using Route 53.
2. **Auto-Scaling Web Application with Monitoring**
   I create a scalable web app using EC2, Auto Scaling Groups, and a Load Balancer. Static content is served from S3, and I use CloudWatch and SNS to monitor and send alerts based on system activity.
3. **Secure Linux Server Setup**
   I set up a Linux server for three different departments (HR, Dev, and Ops), configure user and group permissions, and automate daily backups to a private GitHub repository using shell scripts and cron jobs.
4. **Serverless HealthBooking on AWS**
   I deploy a fully serverless application that simulates an Health booking system. It includes Lambda functions, API Gateway, DynamoDB, and other AWS services. The goal is to explore how serverless setups compare to traditional server-based systems.

2. Scenario #1 – Hosting My Portfolio on AWS

a. Objective

The goal of this scenario is to host a personal developer portfolio on AWS in a way that's secure, fast, and professional. This includes storing the website files in a private S3 bucket, delivering them using CloudFront, and connecting the site to a custom domain through Route 53. The final result should be a fully functional portfolio that reflects my work .

b. Architecture Diagram

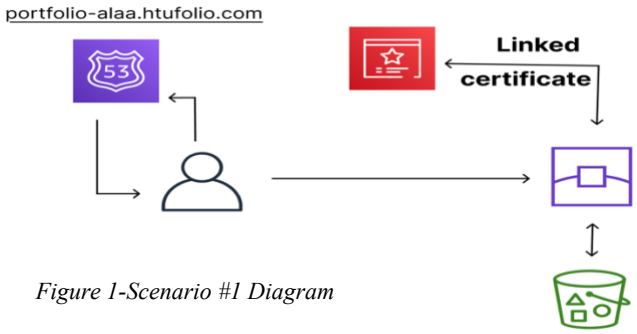


Figure 1-Scenario #1 Diagram

c. Implementation Steps

1) Created a Private S3 Bucket

- I named my bucket portfolio.alaaaa.com .
- To keep it secure, I enabled **Block All Public Access** for the bucket.
- I uploaded my website files, including index.html, images, CSS, and JS folders.

2) Configured Bucket Policy for CloudFront

- Since the bucket is private, I added a policy that allows **only CloudFront** to access its contents.
- This was done using a JSON policy that grants the CloudFront distribution permission to get objects from the bucket.

3) Set Up CloudFront Distribution

- I created a CloudFront distribution and set the S3 bucket as the origin.
- I used **Origin Access Control (OAC)** to securely connect CloudFront with the private S3 bucket.
- I added my custom domain portfolio-alaa.htufolio.com as an alternate domain name.
- Linked the distribution to a valid **SSL certificate** (issued by AWS Certificate Manager).
- I ran a **CloudFront cache invalidation** to make sure new changes appear immediately.

4) Created a Custom SSL Certificate

- I requested a new certificate for portfolio-alaa.htufolio.com using AWS Certificate Manager.
- To validate the domain, I added the provided **CNAME record** to Route 53.
- Once validated, the status showed **"Success"**, and I used it in CloudFront for HTTPS support.

5) Configured Route 53 for Domain Routing

- I used Route 53 to set up a **Hosted Zone** for my domain (htufolio.com).
- I created a **CNAME record** pointing portfolio-alaa.htufolio.com to the CloudFront domain (d1pedsexnd0a7l.cloudfront.net).
- This allows users to visit my portfolio using a clean and professional domain.

6) Tested the Website

- After completing all the setup, I tested the domain in the browser.
- The website loaded successfully using HTTPS, and the content was served via CloudFront from the private S3 bucket.

d. Screenshots

- **S3 Bucket**

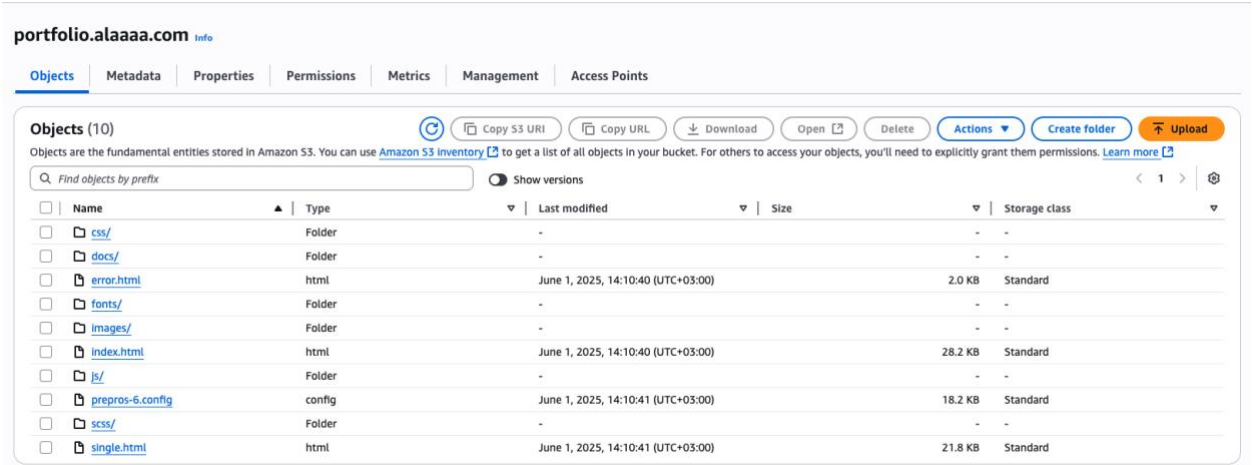


Figure 2-Created a Private S3 Bucket

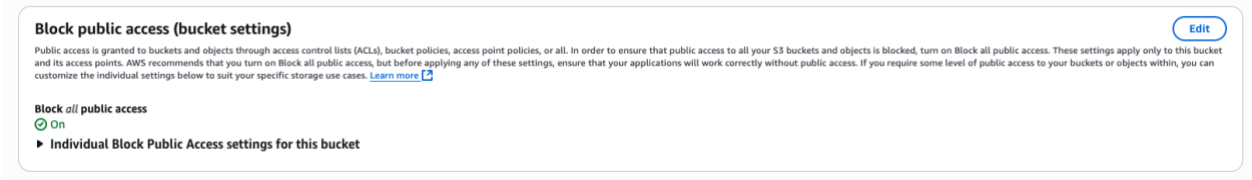


Figure4 -Block All Public Access

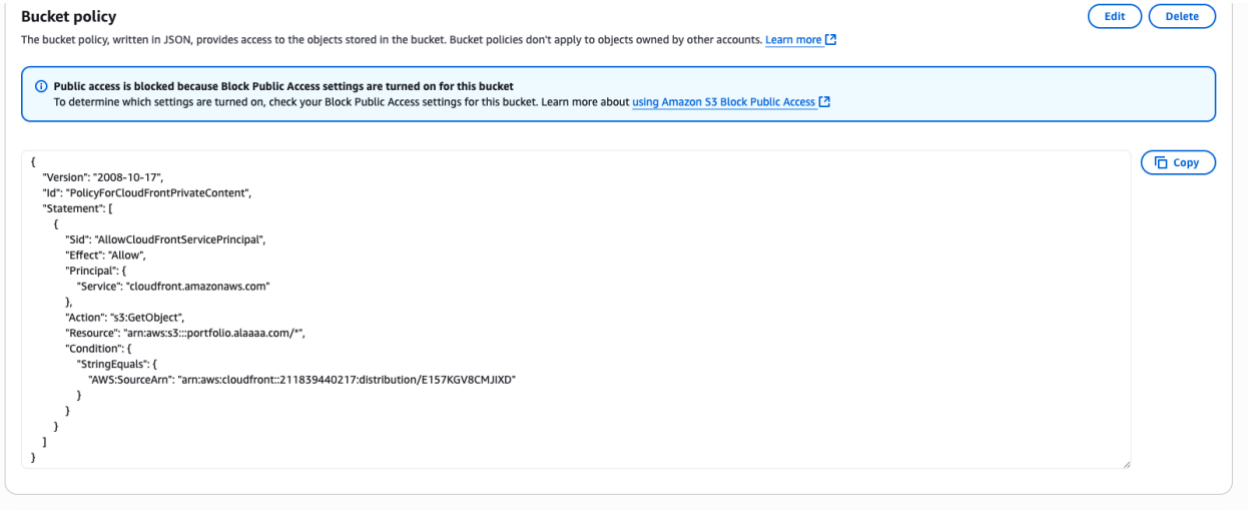


Figure3 -Configured Bucket Policy for CloudFront

- Cloudfront

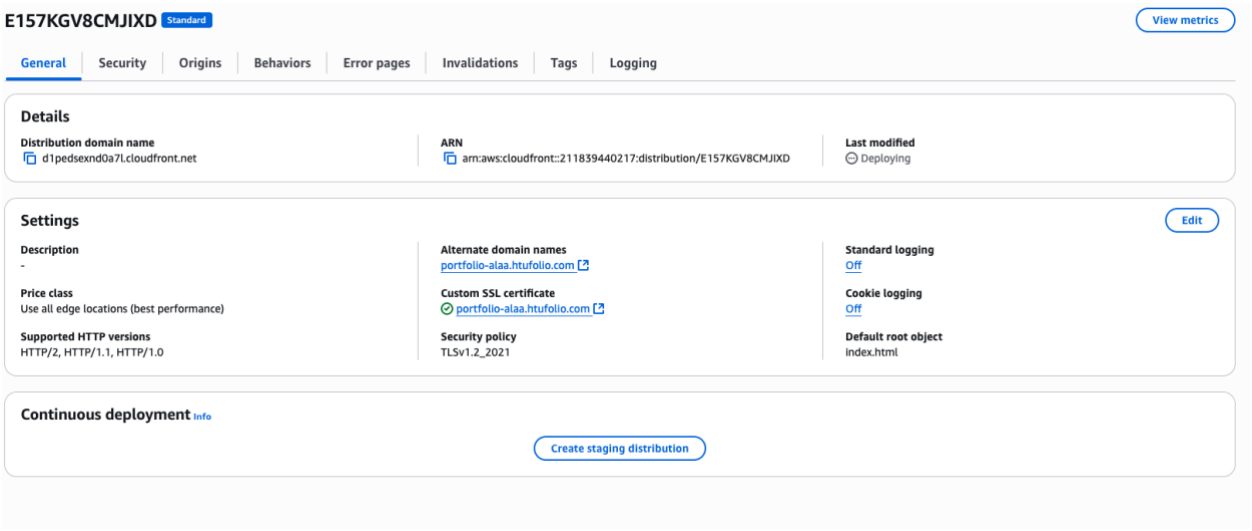


Figure 5 -Set Up CloudFront Distribution

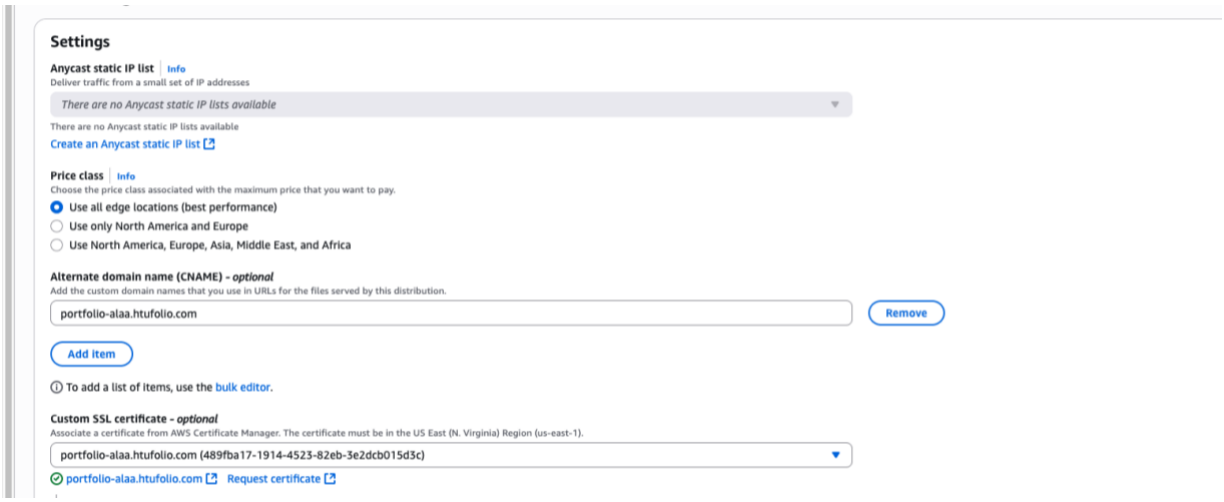


Figure 6 - added (custom domain + SSL certificate)

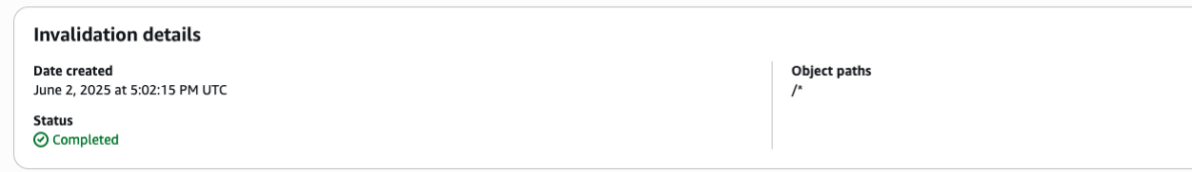


Figure 7-CloudFront cache invalidation

- **Certificate**

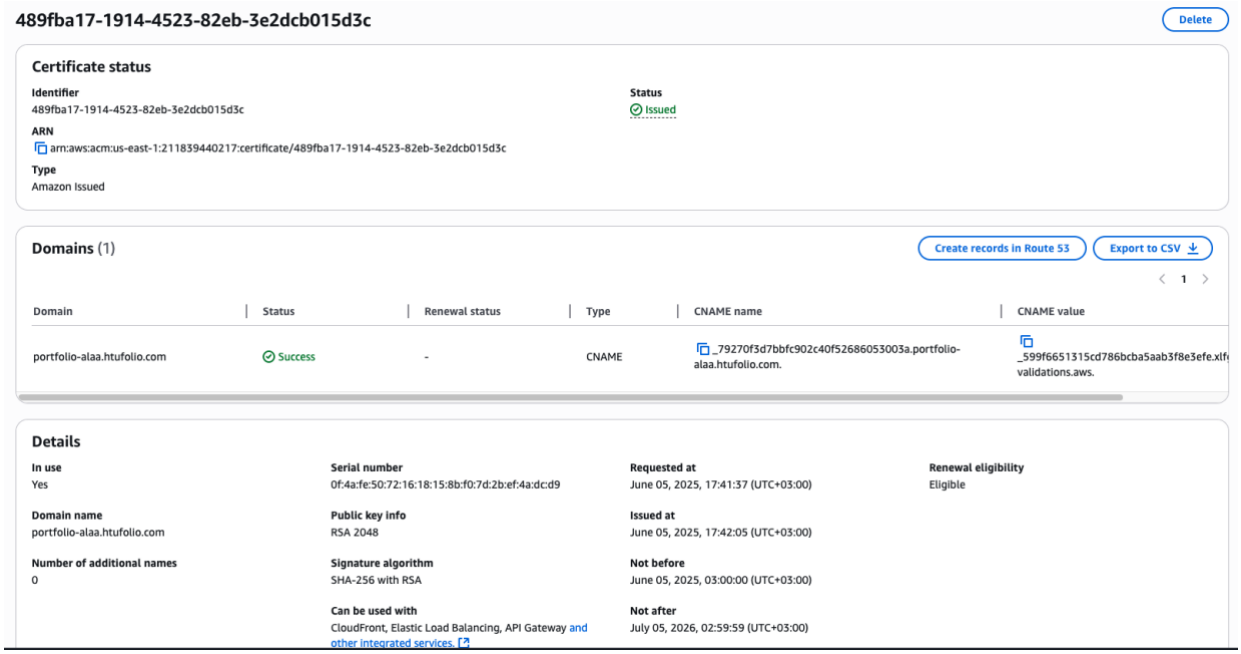


Figure 8-Created a Custom SSL Certificate

- **Route 53**

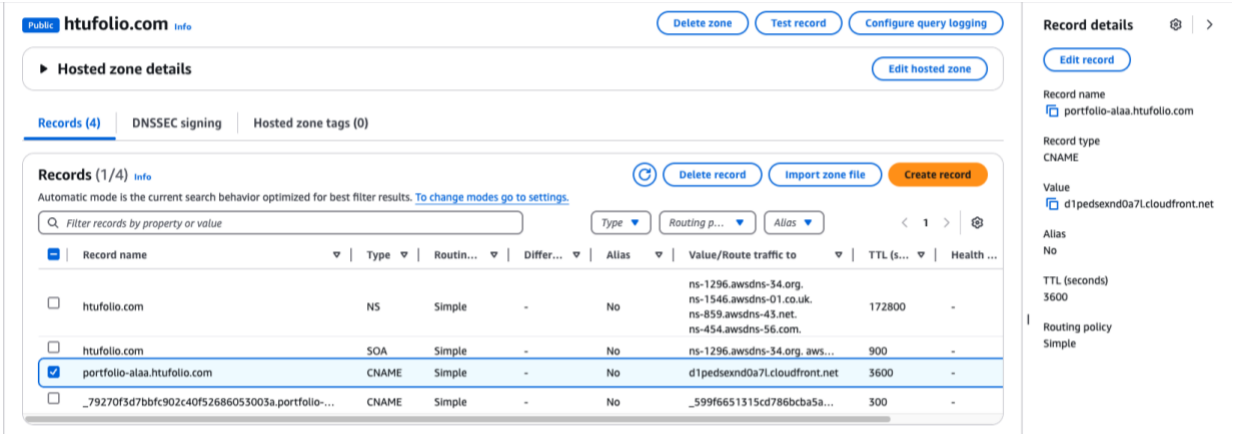


Figure 9-Configured Route 53 for Domain Routing

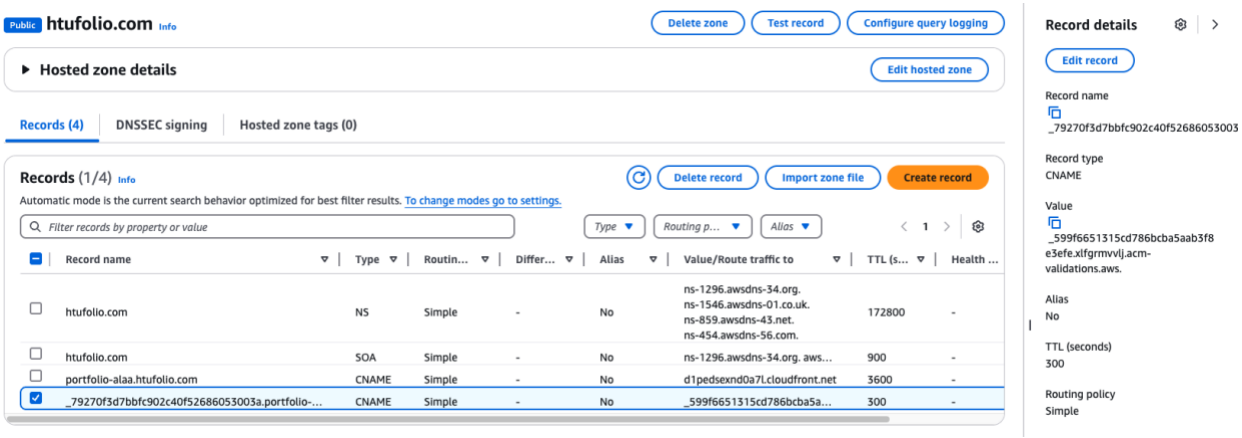


Figure 10- To validate the domain, I added the provided CNAME record to Route 53

- Tested The Website

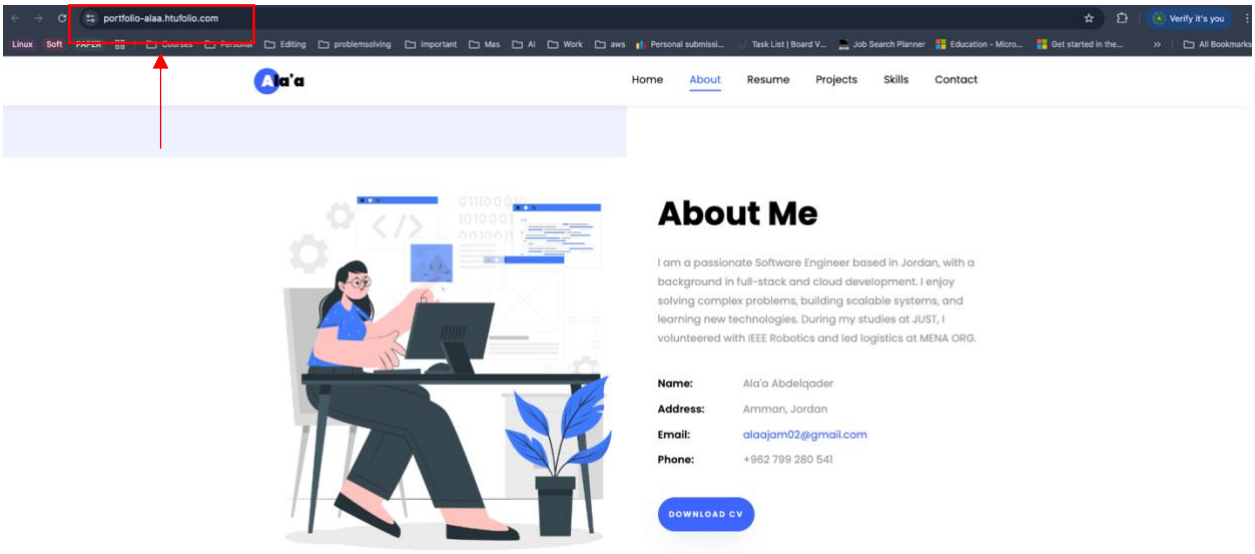


Figure 11-Tested the Website

e. Conclusion

The architecture was designed following AWS best practices to ensure **high availability and fault tolerance** by using **CloudFront** as a global content delivery network, which serves the website content from edge locations closest to the user, reducing latency and ensuring consistent performance. The content is hosted in an **S3 bucket with restricted public access**, and access is securely managed through **Origin Access Control (OAC)**. Additionally, **Route 53** was used for domain routing, providing DNS failover capability and improved availability.

In the future, I could improve the setup by:

- **Enabling versioning** on the S3 bucket to track and recover from accidental changes or deletions.
- **Adding a custom error page setup** in CloudFront for better user experience.
- **Implementing CloudWatch monitoring** to track access logs, performance, and detect issues early.
- **Setting up CI/CD pipelines** using AWS CodePipeline or GitHub Actions to automate future updates.

3. Scenario #2 – Auto-Scaling Web Application

a. Objective

The goal of this scenario is to build a web application that can **automatically scale up or down** based on how much traffic it's getting. This means the app can handle more users when needed and save costs when it's not busy. We used **EC2 instances** to run the web server and **S3** to store static content like the HTML file. To make sure the app stays available and performs well, we connected it to an **Auto Scaling Group** and a **Load Balancer**. We also added monitoring tools like **CloudWatch** to track how the system is doing, and **SNS notifications** to alert us by email if something changes (like launching new servers when traffic increases). The overall idea is to create a **resilient, scalable, and well-monitored** application infrastructure—just like real companies do when they want their services to stay online and responsive under any load.

b. Architecture Diagram

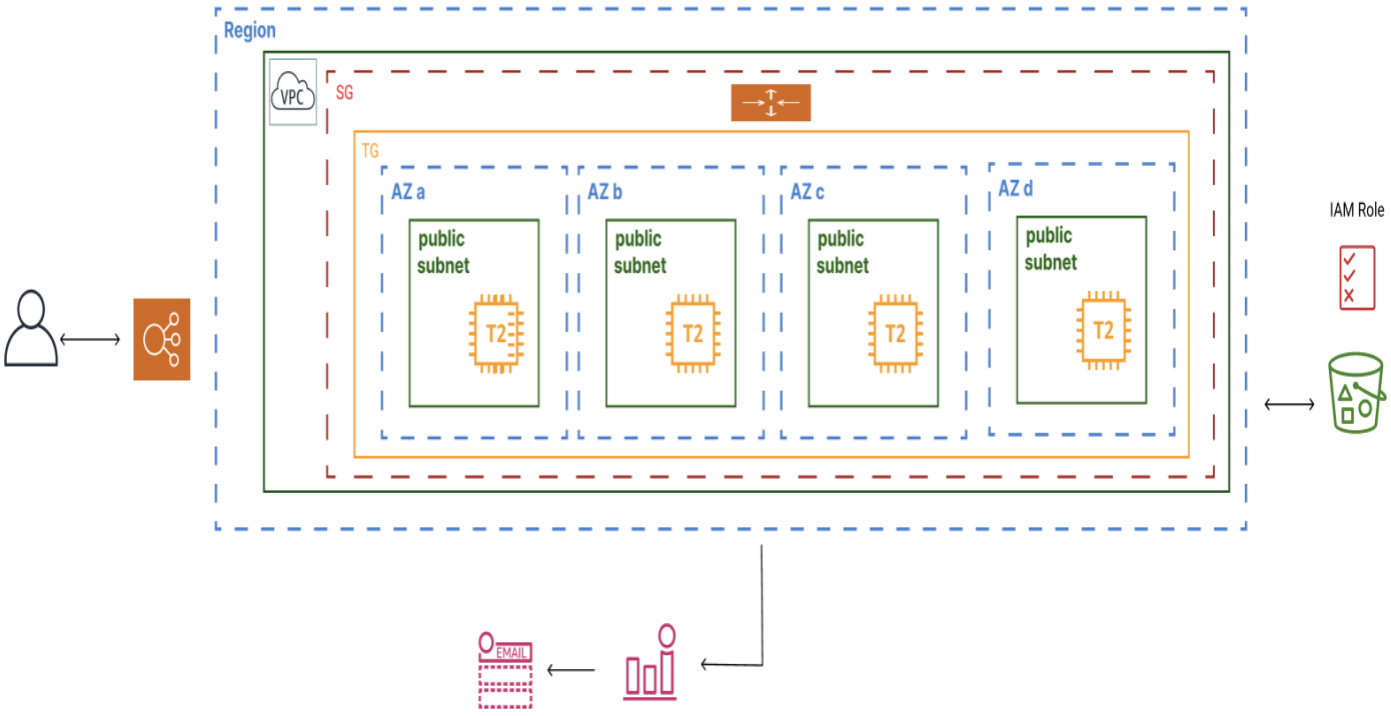


Figure 13-Scenario #2 Diagram

c. Implementation Steps

1) Created an S3 Bucket for Static Content.

- I created an S3 bucket named htu-webpage to host the static HTML content (index.html).
- Enabled S3 static website.
- Kept Block all public access **enabled** to improve security, and instead allowed access through EC2 using an IAM Role.

2) Built a Launch Template for EC2 Instances.

- Created a **Launch Template** named WebAppTemplate using the Amazon Linux 2 AMI.
- Attached a security group named web-security that allows:
    - HTTP traffic on port 80 from anywhere.
    - SSH access only from my IP for management
- In the **User Data** script:
    - Installed Apache
    - Started the HTTP service.
    - Pulled index.html from the S3 bucket and placed it in /var/www/html.

3) Created an Auto Scaling Group (ASG)

- I created an Auto Scaling Group named ASG-web using the above launch template.
- Set the minimum capacity to 1, desired capacity to 1, and maximum capacity to 4.
- Attached the group to four subnets in different Availability Zones for high availability.
- This ensures the application scales in or out based on actual demand.

4) Attached Scaling Policies

- Added **Target Tracking Scaling Policies** to the ASG: Policy to **scale out** when average CPU Utilization > 50%.
- Minimum number of healthy instances required for scaling out: 1.

- CloudWatch automatically triggers these policies when thresholds are breached.

5) Set Up an Application Load Balancer (ALB)

- Created an Application Load Balancer named ASG-web-LB.

  Scheme: **Internet-facing**, type: **Application**.

- Added the four public subnets to support high availability.
- Added a listener on port 80 (HTTP).
- Forwarded requests to a target group named WebApp-TG.
- Configured health checks for the target group to monitor /index.html.

6) Enabled CloudWatch Monitoring and Alarms

- Enabled detailed monitoring on the ASG and EC2 instances.
- Created CloudWatch Alarm for GroupDesiredCapacity = 4 to detect if max capacity is reached.
- Verified metrics visually via the **CloudWatch dashboard**.
- Used the SNS service to receive email notifications when alarms go off.

7) Performed Load Testing with stress Tool

- Connected to 2 EC2 instances via SSH.

- Installed the stress tool.
- Ran the following command to simulate high CPU usage for 10 minutes:
- Verified from CloudWatch and EC2 dashboard that:

    - CPU usage spiked above 50%.
    - Auto Scaling Group launched additional instances up to (3).
    - Received SNS email notifications about the scaling events.

8) Verified Notifications and Instance Behavior

- SNS notifications were triggered for launching new instances due to CPU spike and ASG reaching maximum configured capacity (4 instance).
- After stress test completed, idle instances were **automatically shut down**.
- Verified that EC2 instance count was scaled down again to match actual load.

9) Monitored with CloudWatch Dashboards

- Used CloudWatch to monitor: GroupDesiredCapacity
- All metrics confirmed correct scaling behavior throughout the test period.

d. Screenshots

- S3 Bucket

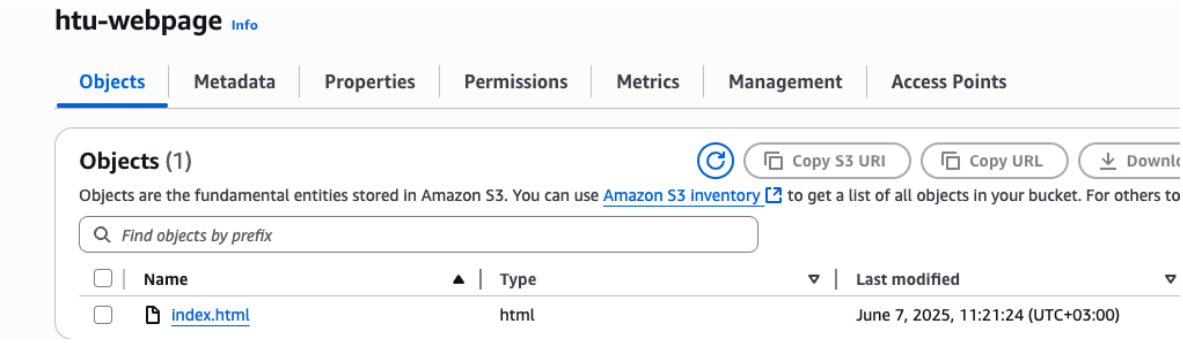


Figure 14- S3 Bucket for Static Content

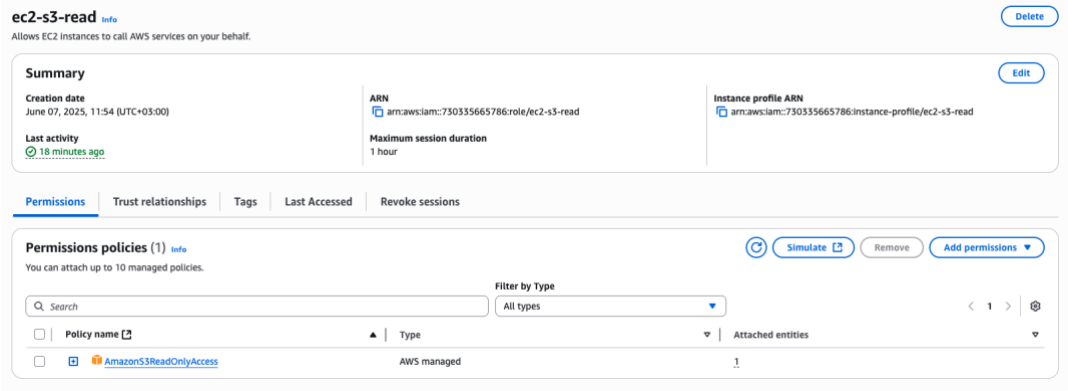


Figure 15-IAM Role to allowed access To S3 from EC2

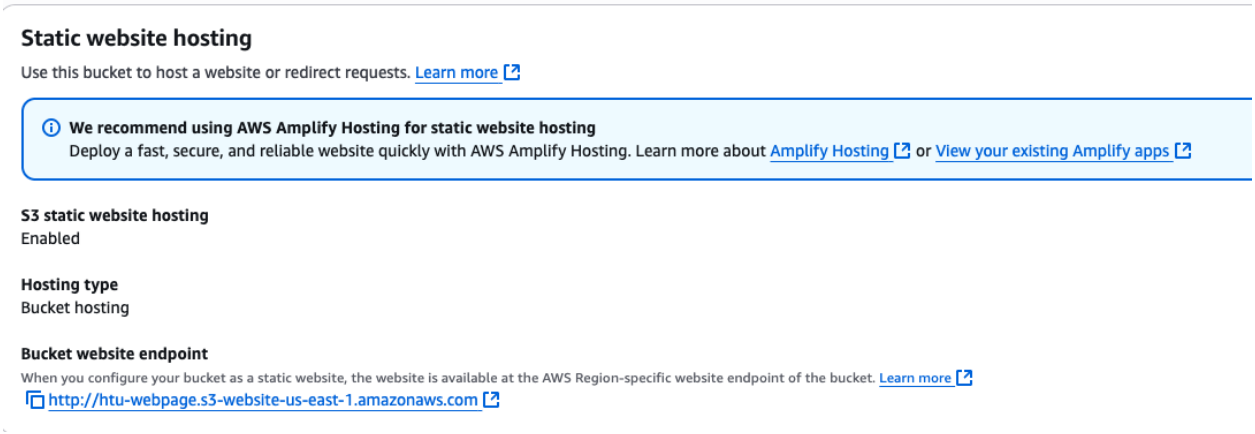


Figure 16- Enable Static Website

- Instance Template

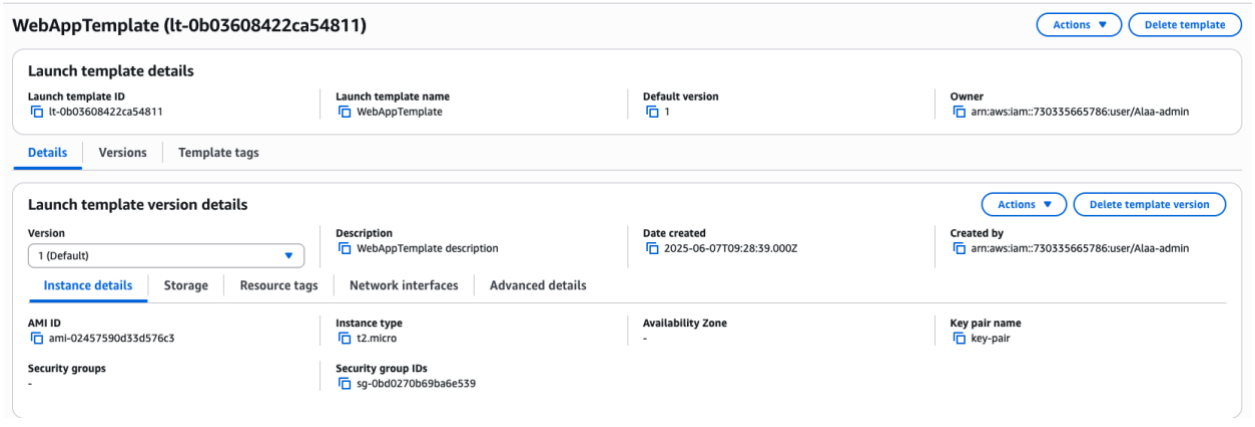


Figure 17-Instance Template

User data

```
#!/bin/bash

yum update -y

# Install Apache and AWS CLI
yum install -y httpd awscli

# Start and enable Apache
systemctl start httpd
systemctl enable httpd

# Download the HTML file from S3
aws s3 cp s3://htu-webpage/index.html /var/www/html/index.html
```

Figure 18- User Data in instance Template

- Web Security Group

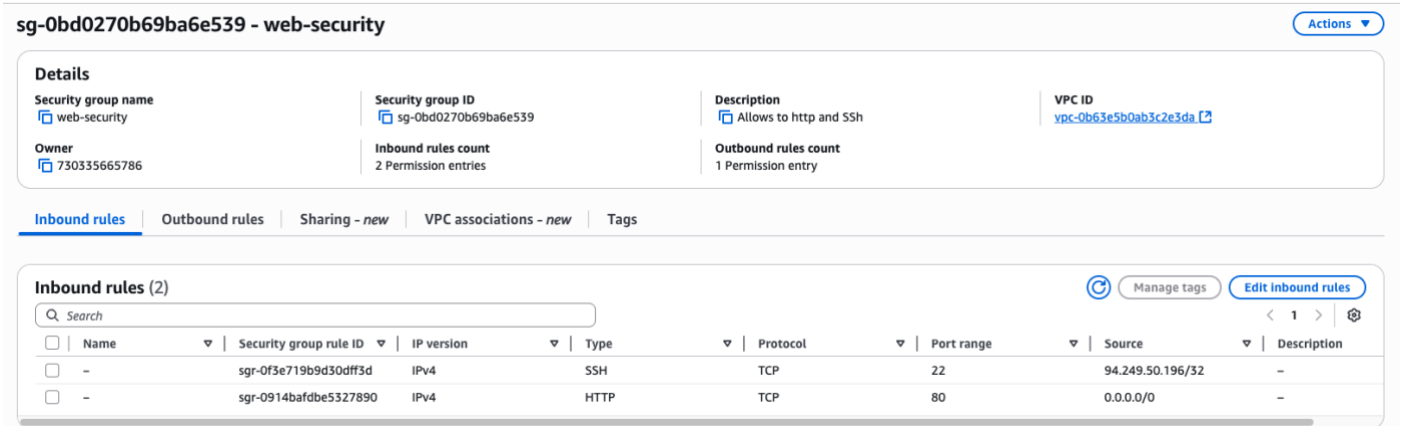


Figure 19-Attached a security group named web-security

- Auto Scaling Group

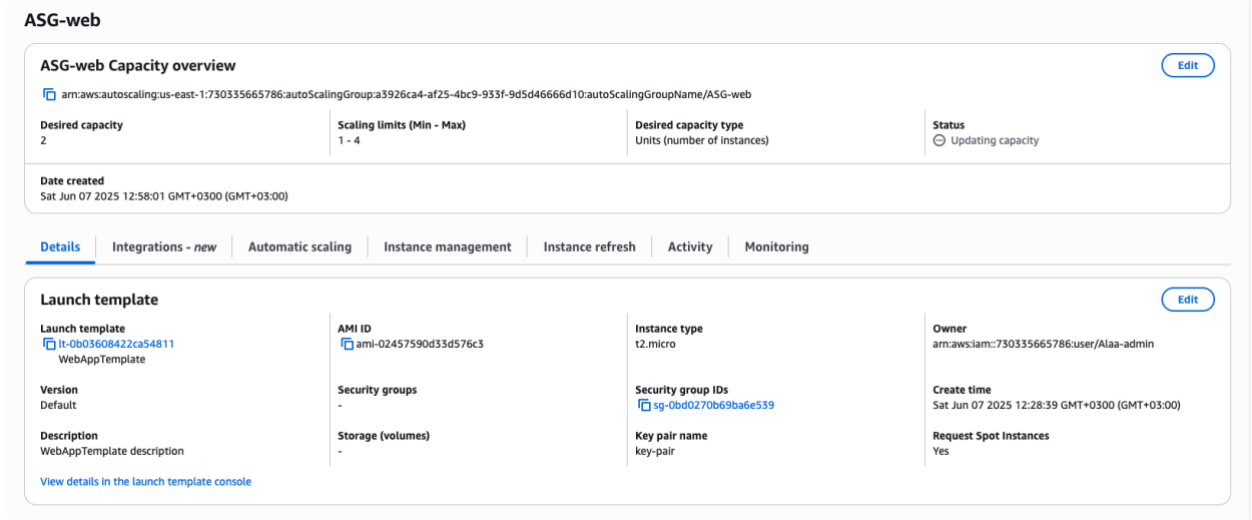


Figure 20-Auto Scaling Group

- Target Tracking

Figure 21-Added Target Tracking Scaling Policies to the ASG

- Application Load Balancer

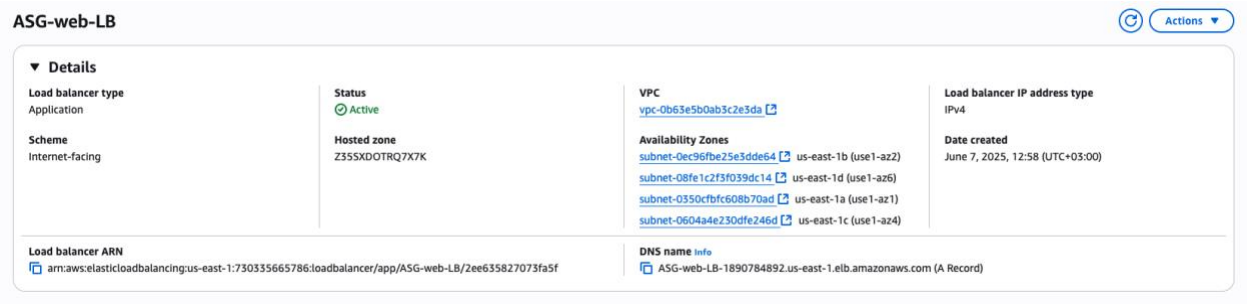


Figure 23-Application Load Balancer details

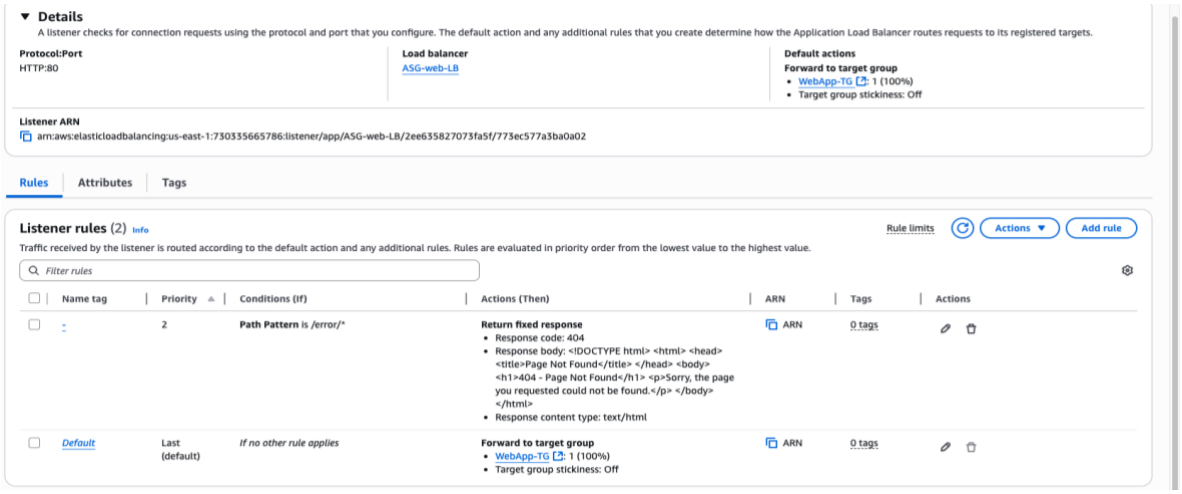


Figure 22 -Added a listener on port 80 (HTTP).

- Target group

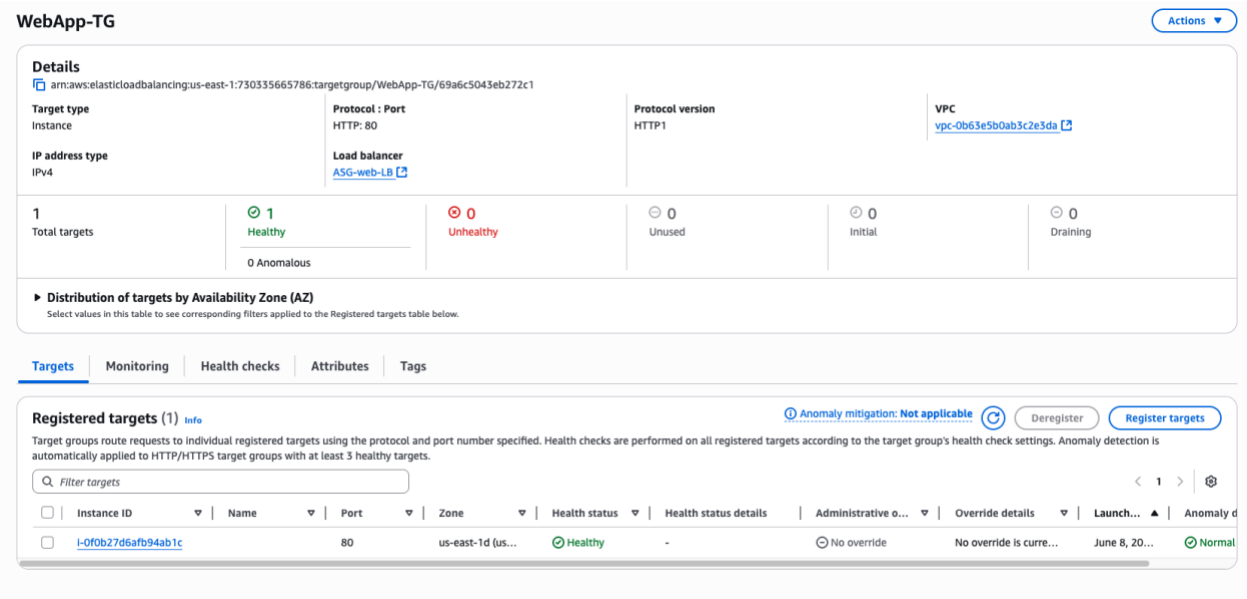


Figure 24- target group named WebApp-TG.

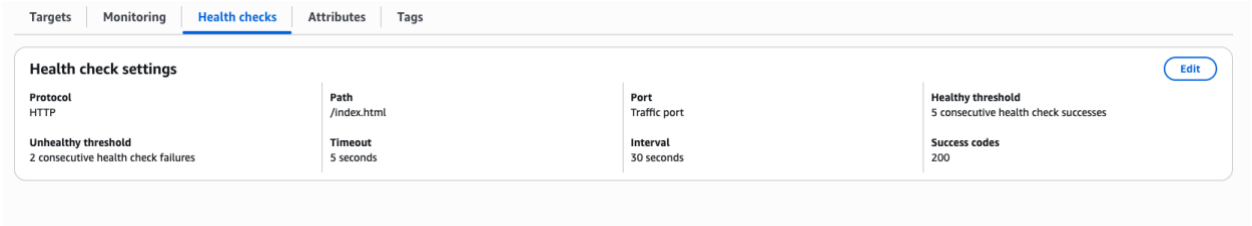


Figure 25-Configured health checks for the target group to monitor /index.html

- CloudWatch Monitoring and Alarms

Figure 26-GroupDesiredCapacity = 4 to detect if max capacity is reached.

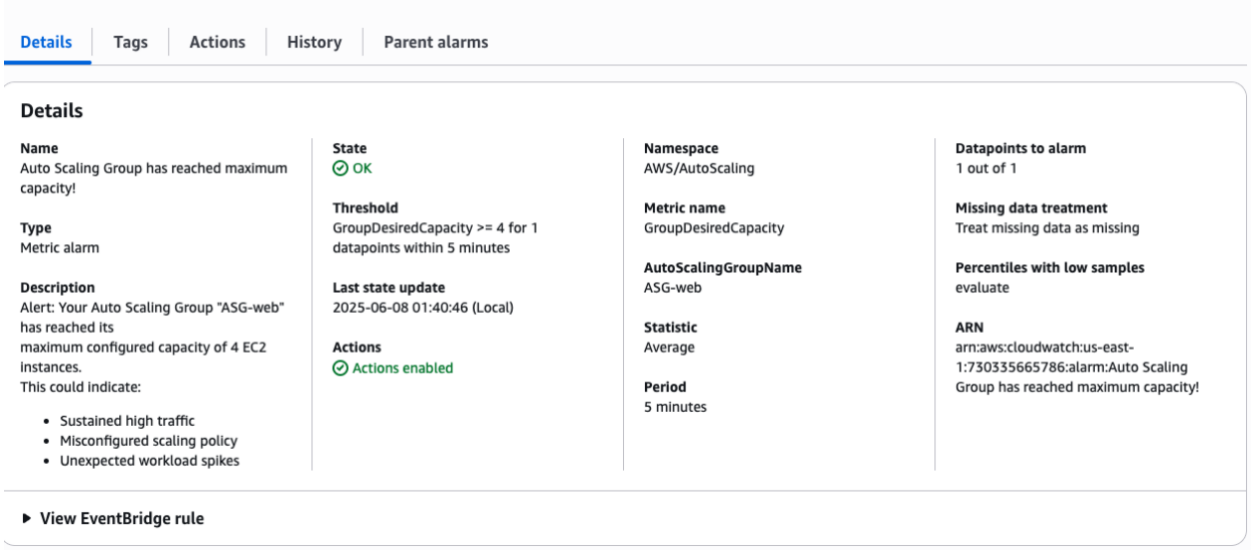


Figure 27- Enabled detailed monitoring

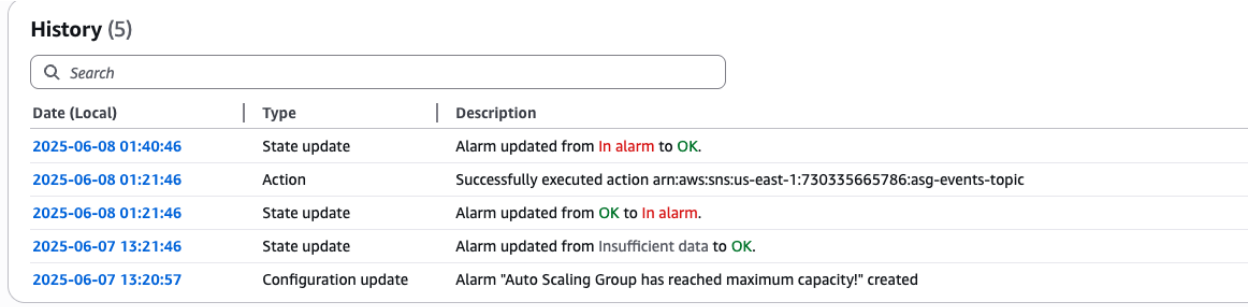


Figure 28-History CloudWatch

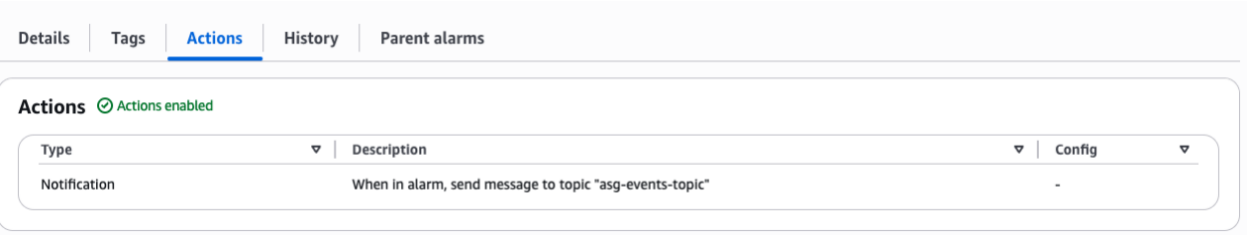


Figure 29 - Actions of CloudWatch

- SNS

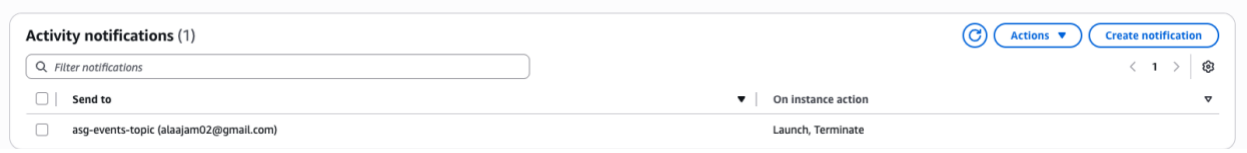


Figure 30- SNS To Email

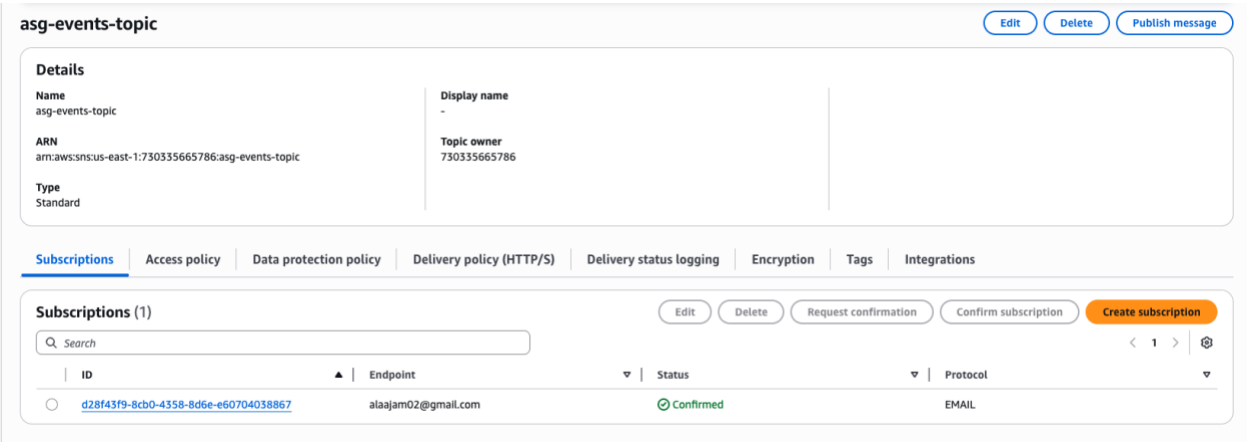


Figure 31- SNS Details

- Testing with stress Tool

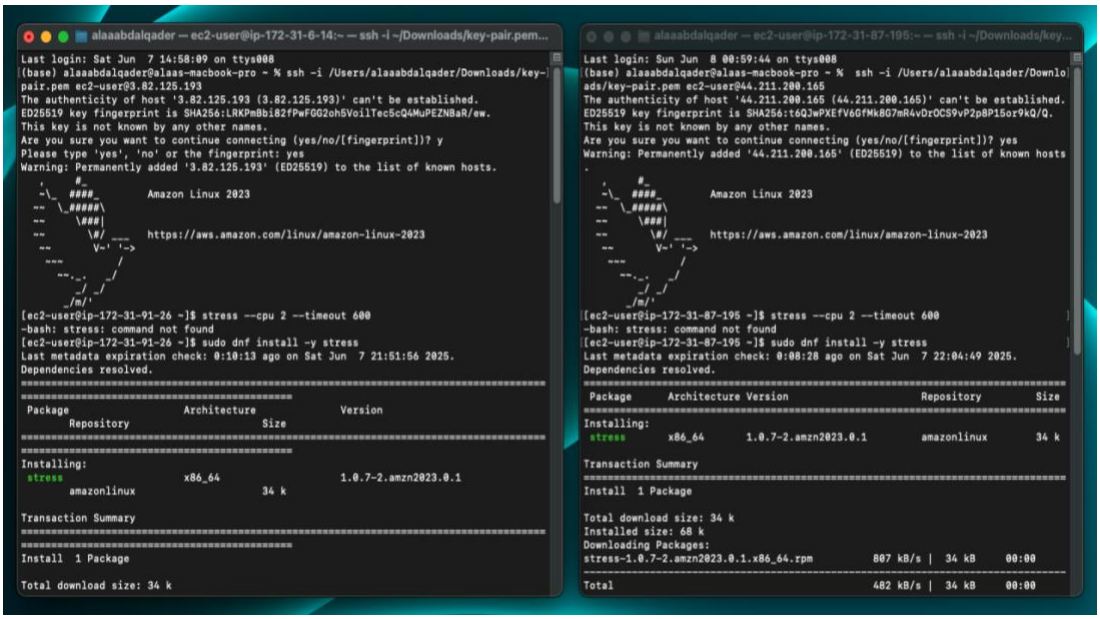


Figure 32-Connected to 2 EC2 instances via SSH and Installed the stress tool.

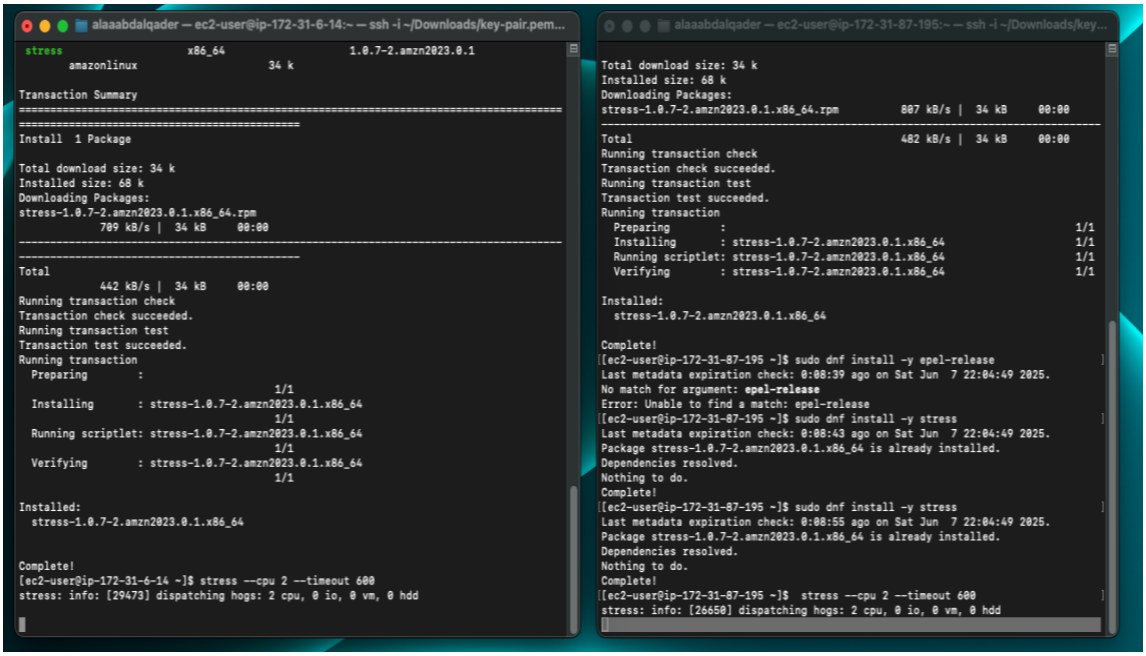


Figure 33-Ran command to simulate high CPU usage for 10 minutes.

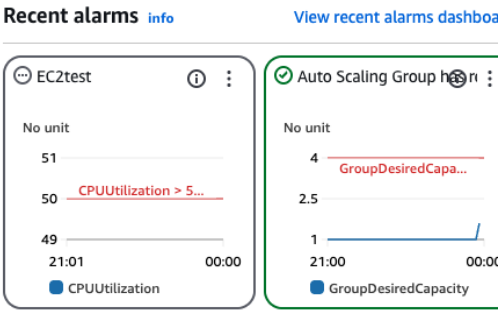


Figure 34-before stress

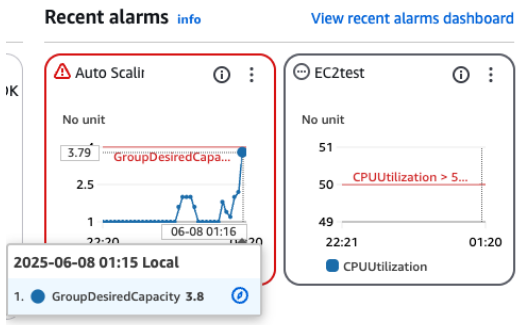


Figure 35-After stress on 2 instance

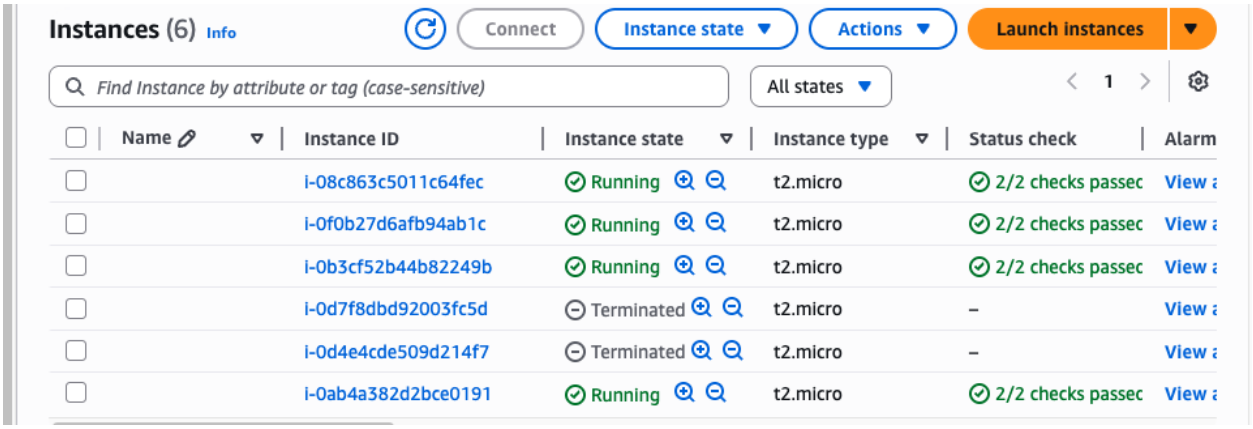


Figure 36-reaching maximum configured capacity

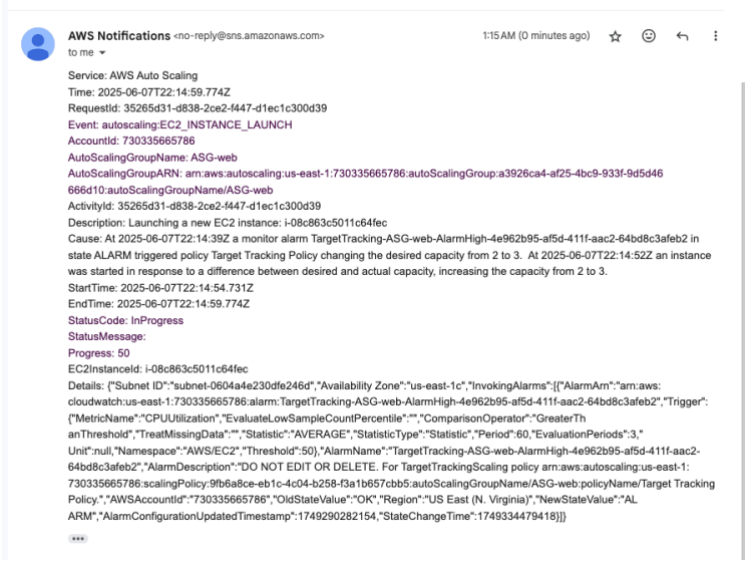
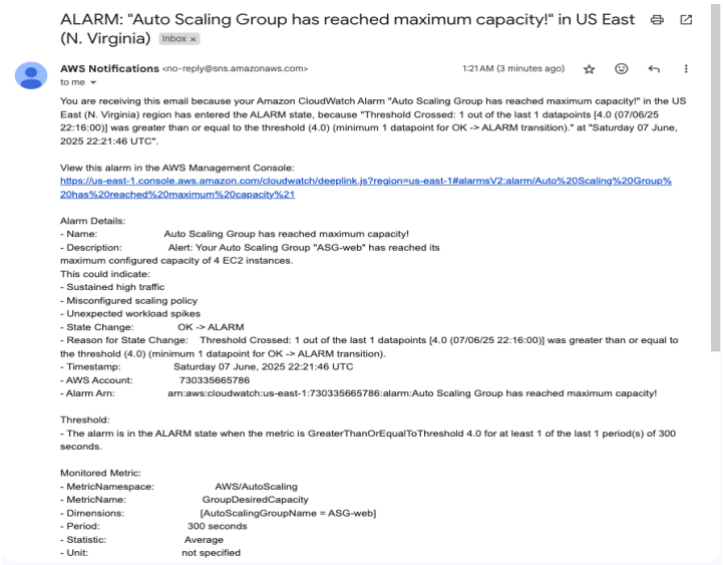


Figure 37-Auto Scaling Trigger: New EC2 Instance Launched Due to High CPU



ALARM: "Auto Scaling Group has reached maximum capacity!" in US East (N. Virginia) Inbox ×

AWS Notifications <no-reply@sns.amazonaws.com> 1:21 AM (3 minutes ago)
to me

You are receiving this email because your Amazon CloudWatch Alarm "Auto Scaling Group has reached maximum capacity!" in the US East (N. Virginia) region has entered the ALARM state, because "Threshold Crossed: 1 out of the last 1 datapoints [4.0 (07/06/25 22:16:00)] was greater than or equal to the threshold (4.0) (minimum 1 datapoint for OK -> ALARM transition)." at "Saturday 07 June, 2025 22:21:46 UTC".

View this alarm in the AWS Management Console:
https://us-east-1.console.aws.amazon.com/cloudwatch/deeplink.js?region=us-east-1#alarmsV2:alarm/Auto%20Scaling%20Group%20has%20reached%20maximum%20capacity%21

Alarm Details:
- Name: Auto Scaling Group has reached maximum capacity!
- Description: Alert: Your Auto Scaling Group "ASG-web" has reached its maximum configured capacity of 4 EC2 instances.
This could indicate:
- Sustained high traffic
- Misconfigured scaling policy
- Unexpected workload spikes
- State Change: OK -> ALARM
- Reason for State Change: Threshold Crossed: 1 out of the last 1 datapoints [4.0 (07/06/25 22:16:00)] was greater than or equal to the threshold (4.0) (minimum 1 datapoint for OK -> ALARM transition).
- Timestamp: Saturday 07 June, 2025 22:21:46 UTC
- AWS Account: 730335665786
- Alarm Arn: arn:aws:cloudwatch:us-east-1:730335665786:alarm:Auto Scaling Group has reached maximum capacity!

Threshold:
- The alarm is in the ALARM state when the metric is GreaterThanOrEqualToThreshold 4.0 for at least 1 of the last 1 period(s) of 300 seconds.

Monitored Metric:
- MetricNamespace: AWS/AutoScaling
- MetricName: GroupDesiredCapacity
- Dimensions: [AutoScalingGroupName = ASG-web]
- Period: 300 seconds
- Statistic: Average
- Unit: not specified

Figure 38- Auto Scaling Alert email Notification: Maximum Capacity Reached

Status	Description	Cause	Start time	End time
Successful	Launching a new EC2 instance: i-0f0b27d6afb94ab1c	At 2025-06-07T22:16:39Z a monitor alarm TargetTracking-ASG-web-AlarmHigh-4e962b95-af5d-411f-aac2-64bd8c3afeb2 in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 3 to 4. At 2025-06-07T22:16:53Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 3 to 4.	2025 June 08, 01:16:55 AM +03:00	2025 June 08, 01:22:27 AM +03:00
Successful	Launching a new EC2 instance: i-08c863c5011c64fec	At 2025-06-07T22:14:39Z a monitor alarm TargetTracking-ASG-web-AlarmHigh-4e962b95-af5d-411f-aac2-64bd8c3afeb2 in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 2 to 3. At 2025-06-07T22:14:52Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 2 to 3.	2025 June 08, 01:14:54 AM +03:00	2025 June 08, 01:19:59 AM +03:00
Successful	Launching a new EC2 instance: i-0ab4a382d2bce0191	At 2025-06-07T22:04:11Z a user request update of AutoScalingGroup constraints to min: 1, max: 4, desired: 2 changing the desired capacity from 1 to 2. At 2025-06-07T22:04:19Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 1 to 2.	2025 June 08, 01:04:20 AM +03:00	2025 June 08, 01:04:26 AM +03:00
Successful	Terminating EC2 instance: i-0d4e4cde509d214f7	At 2025-06-07T21:57:07Z a monitor alarm TargetTracking-ASG-web-AlarmLow-51415ad4-af53-40c1-b8a6-a880d4cda5e1 in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 2 to 1. At 2025-06-07T21:57:12Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 2 to 1. At 2025-06-07T21:57:12Z instance i-0d4e4cde509d214f7 was selected for termination.	2025 June 08, 12:57:12 AM +03:00	2025 June 08, 01:02:55 AM +03:00
Successful	Launching a new EC2 instance: i-0d4e4cde509d214f7	At 2025-06-07T21:51:12Z a user request update of AutoScalingGroup constraints to min: 1, max: 4, desired: 2 changing the desired capacity from 1 to 2. At 2025-06-07T21:51:22Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 1 to 2.	2025 June 08, 12:51:24 AM +03:00	2025 June 08, 12:51:41 AM +03:00

Figure 39- Auto Scaling Activity Log: EC2 Launch and Termination Events

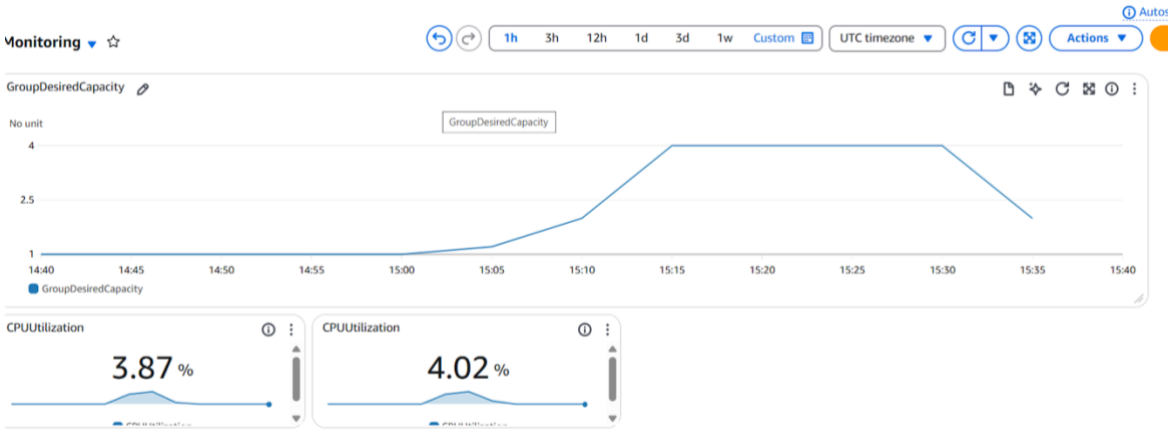


Figure 40-Analyze Scaling Behavior and Trends from CloudWatch Metrics

From the CloudWatch metrics, we can see how the Auto Scaling Group reacted to the increase in workload. At first, the desired capacity was set to 1 instance because everything was running smoothly. But once the CPU usage started to go up (around 4%), the Auto Scaling policy kicked in and started adding more instances — eventually reaching 4, which was the maximum allowed. After the stress test ended, the system automatically reduced the number of instances again. This shows that Auto Scaling is working well, but it also means the CPU threshold might be a bit too low. To make it work more efficiently, we could adjust the CPU threshold or add a cooldown period so the system doesn't scale up too quickly.

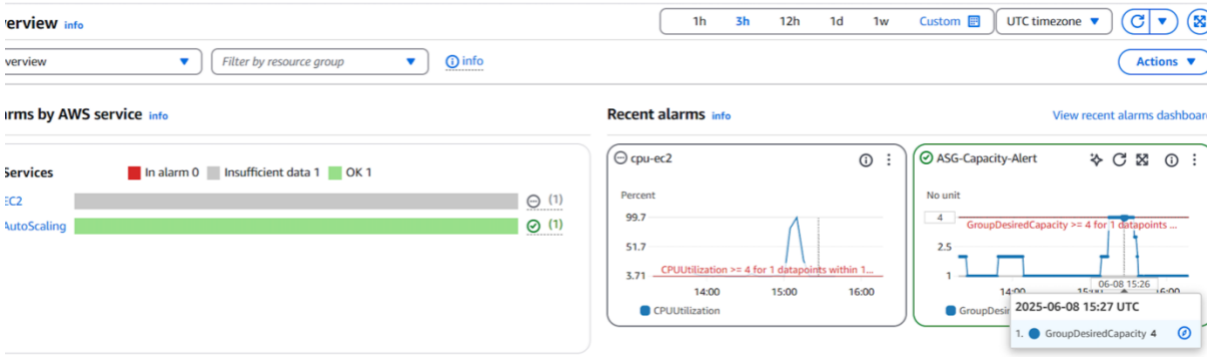


Figure 41-Analyzing Scaling Behavior Using CloudWatch Metrics

we can clearly see how the Auto Scaling Group responded to the CPU utilization. When the CPU usage spiked (as shown in the left graph), the system reacted by increasing the desired capacity (middle graph) to handle the higher load. It added instances until it reached the maximum configured limit of 4.

This trend confirms that the scaling policy worked correctly based on the defined CPU threshold (in this case, around 4%). Once the workload dropped again, the desired capacity began to decrease, showing that the system is also able to scale down automatically. This kind of visibility helps ensure that our infrastructure can react efficiently to real-time demand.

e. Conclusion

In this scenario, I successfully deployed a scalable and monitored web application using EC2, an Application Load Balancer (ALB), Auto Scaling Group (ASG), CloudWatch, and S3 integration. The setup dynamically adjusted the number of EC2 instances based on CPU utilization, ensuring high availability under stress conditions.

During testing, the Auto Scaling Group scaled out and in correctly, reacting to increased load and later terminating excess instances when demand dropped. Notifications via SNS confirmed each scaling event, proving the infrastructure was responsive and functioning as expected.

To further improve efficiency, I suggest fine-tuning the scaling thresholds. For example:

- **Adding additional CloudWatch metrics**, such as memory or network utilization, could improve decision-making beyond just CPU.

- **Monitoring the average response time** per instance in CloudWatch to complement CPU metrics and avoid premature scaling.

4) Scenario #3 – Secure Linux Server Setup on AWS

a. Objective

The goal of this scenario is to set up a secure Linux-based EC2 server that hosts isolated environments for three departments: Human Resources (HR), Development (Dev), and Operations (Ops). Each department should only access its own files. Additionally, the solution must include an automated backup process that archives each department's data and pushes it daily to a private GitHub repository using Bash scripts and cron jobs.

b. Implementation Steps

1) Launched the EC2 Instance

- I launched a new EC2 instance using **Amazon Linux 2** for its compatibility and long-term support.
- Chose instance type **t2.micro** to stay within the AWS free tier.
- I **generated a new key pair** to securely connect to the instance via SSH.
- I created a **custom security group** named SG-web that allows SSH access (port 22) **only from my IP address**.
- This security group was attached during instance launch to ensure limited and safe access.

2) Connected to EC2 Using SSH

- After launching the instance, I connected using the terminal command:

```
ssh -i ~/Downloads/secure-key.pem ec2-user@<EC2-Public-IP>
```

3) Created Folder Structure for Departments

- Once connected, I created isolated directories for each department under a main /company directory:

```
sudo mkdir -p /company/hr /company/dev /company/ops
```

4) Created User Groups and Users

- I created user groups for each department:

```
sudo groupadd HR
```

- Then I added two users per department and added them to their groups:

  `sudo useradd hr1 -G HR  sudo useradd hr2 -G HR`

5) Configured Directory Permissions

- I made each department folder owned by the root user but grouped to the corresponding department:

  `sudo chown root:HR /company/hr`

- Set strict permissions so only the corresponding group has access:

  `sudo chmod 770 /company/hr`

6) Wrote GitHub Backup Script

- I created a Bash script at `/usr/local/bin/github_backup.sh` to archive and push daily backups.

7) Set Up Cron Job for Daily Automation

- I used the `crontab` to schedule daily backups at 2 AM.

8) Validated the Automation

To verify that the backups were running and being pushed correctly:

- I reviewed the `/var/log/github_backup.log` file.
- I checked the private GitHub repository commit history to confirm daily uploads.

d. Screenshots

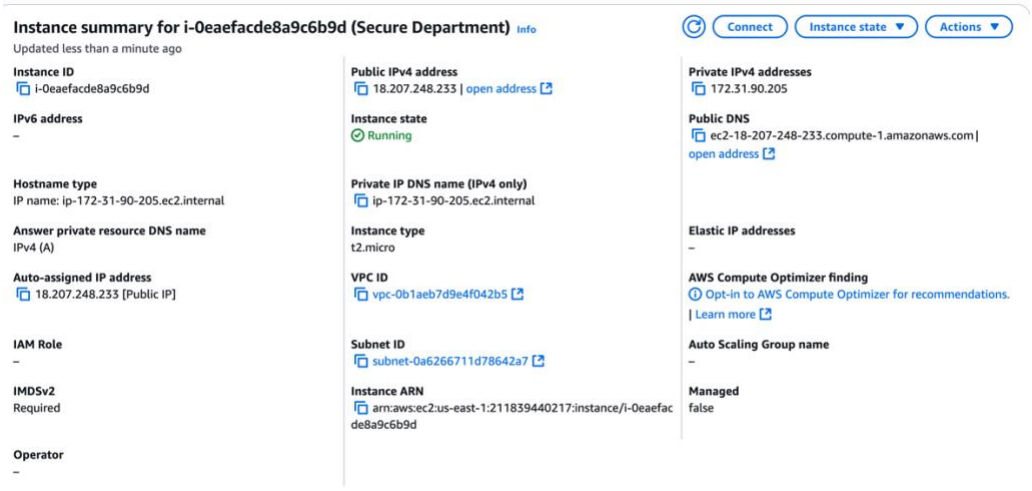


Figure 42-EC2 instance using Amazon Linux 2

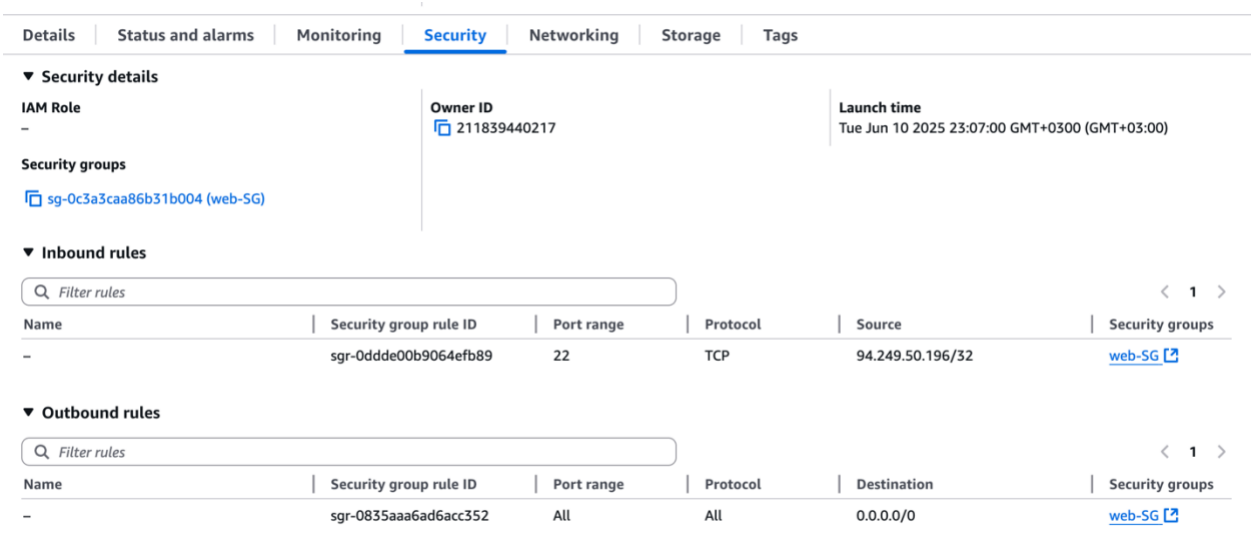


Figure 43-custom security group that allows SSH access (port 22) only from my IP address

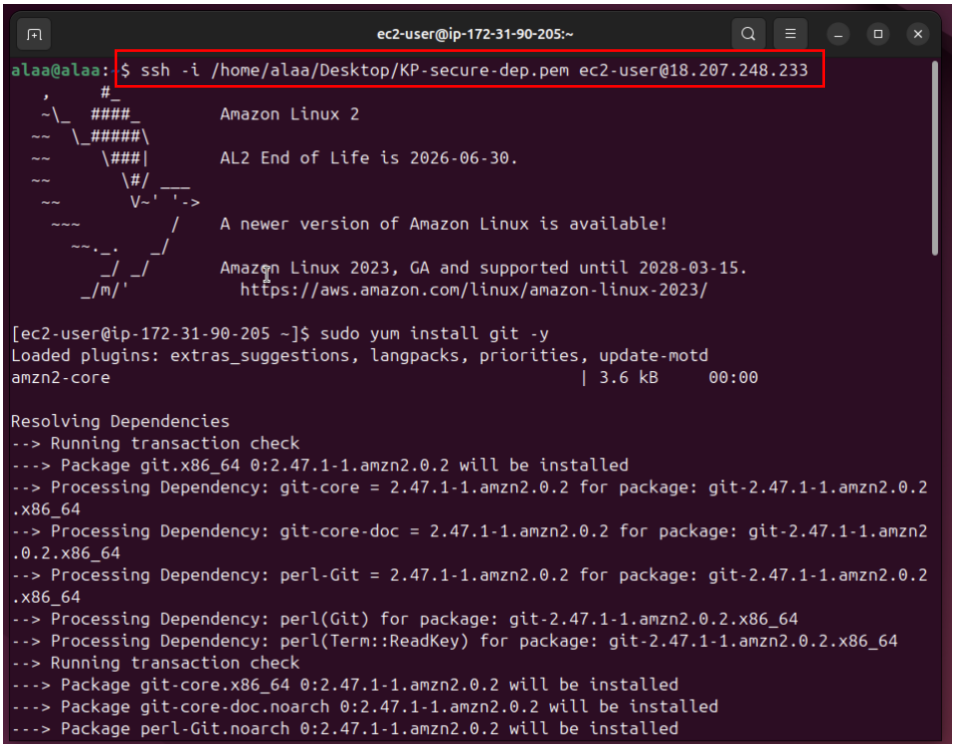


Figure 44-Connected to EC2 Using SSH

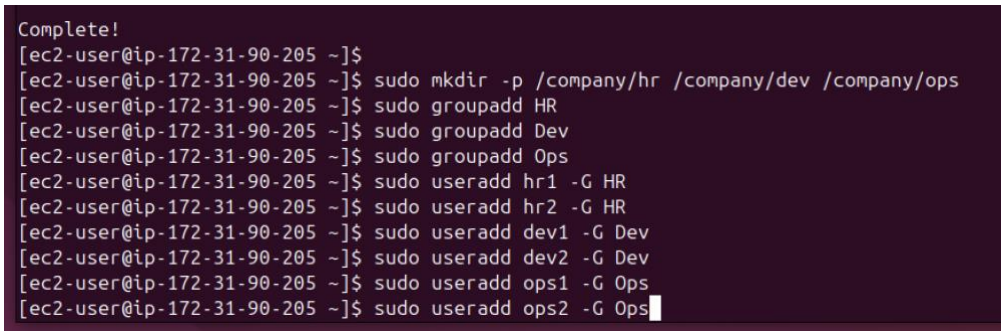


Figure 45-Create Groups and Users-assign them to their respective groups

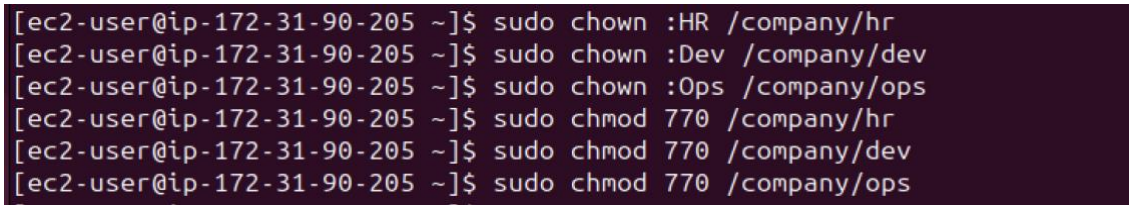


Figure 46-Ownership and Permissions

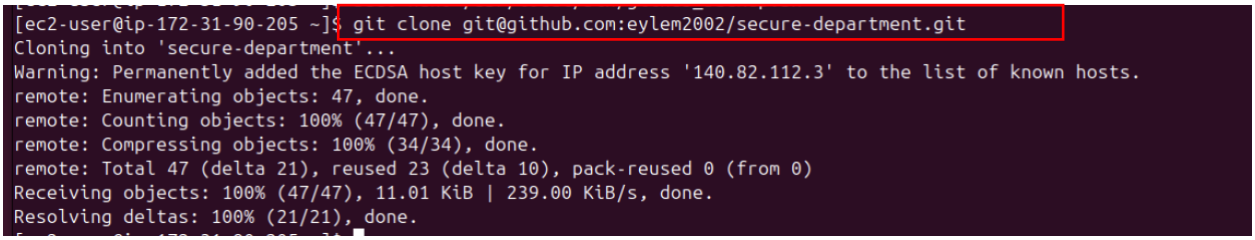


Figure 48-clone to my repo on GitHub

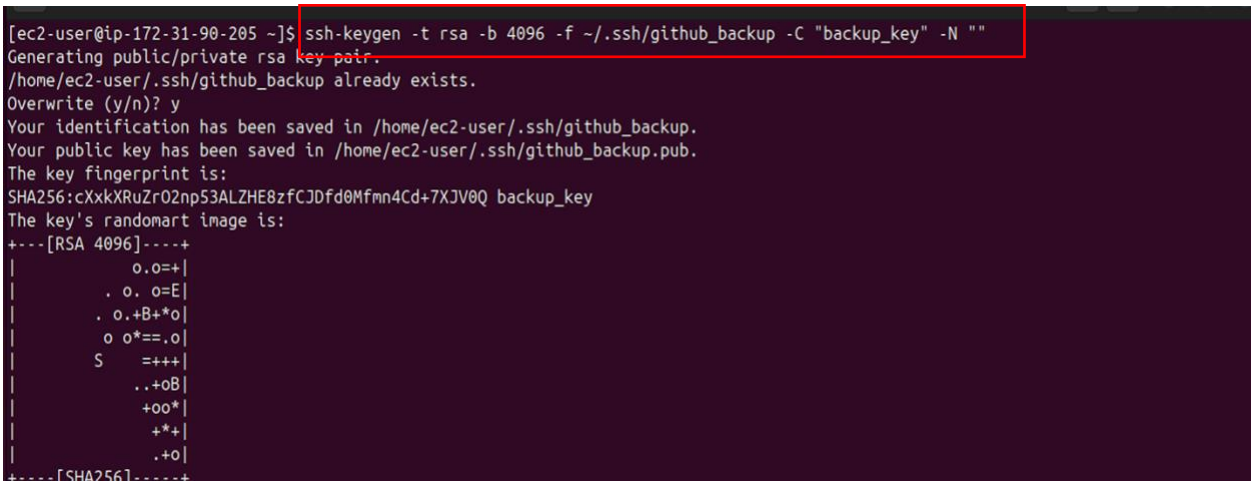


Figure 47-SSH Key Generation for GitHub Backup

Figure 49-Viewing and Configuring SSH Deploy Key

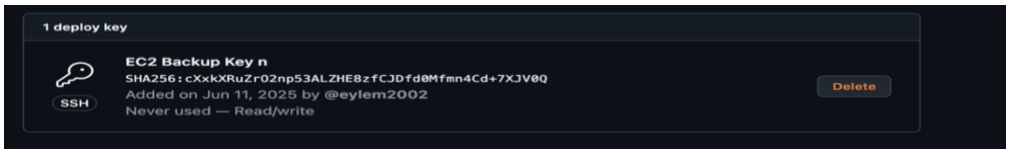


Figure 50-Add key on the repo

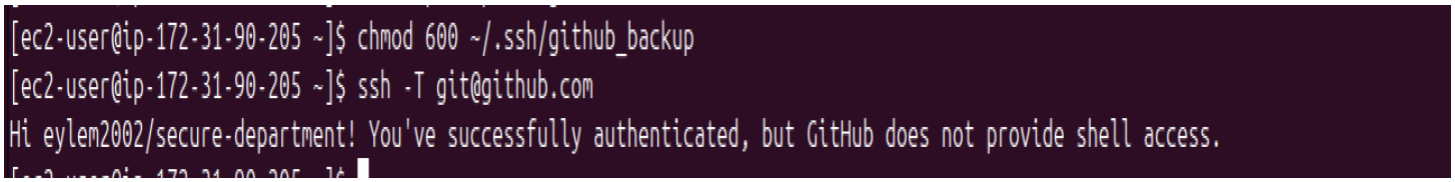


Figure 51-Connection to GitHub is Done

```
21 45 * * * /usr/local/bin/github_backup.sh >> /var/log/github_backup.log 2>&1
~
```

Figure 52 -Scheduled Cron Job for Daily GitHub Backup at 9:45 PM

```
  GNU nano 2.9.8                                          /usr/local/bi

#!/bin/bash

TODAY=$(date +%F)
BASE_DIR="/company"
BACKUP_DIR="$HOME/secure-department"

for dept in hr dev ops; do
  ARCHIVE_NAME="${dept}_${TODAY}.tar.gz"
  sudo tar -czf "/tmp/$ARCHIVE_NAME" "$BASE_DIR/$dept"
  mv "/tmp/$ARCHIVE_NAME" "$BACKUP_DIR/"
done

cd "$BACKUP_DIR"
git add .
git commit -m "Automated backup on $TODAY"
git push origin main
```

Figure 53-Backup script

```
[ec2-user@ip-172-31-90-205 ~]$ sudo /usr/local/bin/github_backup.sh
tar: Removing leading `/' from member names
tar: Removing leading `/' from member names
tar: Removing leading `/' from member names
[main 8cf6ea4] Automated backup on 2025-06-10
 Committer: EC2 Default User <ec2-user@ip-172-31-90-205.ec2.internal>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 3 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 dev_2025-06-10.tar.gz
 create mode 100644 hr_2025-06-10.tar.gz
 create mode 100644 ops_2025-06-10.tar.gz
Warning: Permanently added the ECDSA host key for IP address '140.82.114.4' to the list of known hosts.
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 769 bytes | 769.00 KiB/s, done.
Total 5 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To github.com:eylem2002/secure-department.git
```

Figure 54-Test the script manually.

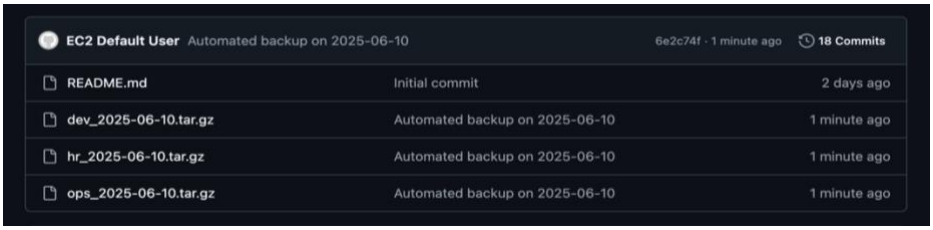


Figure 55 -Automated Daily Backup Files Uploaded to Private GitHub Repository

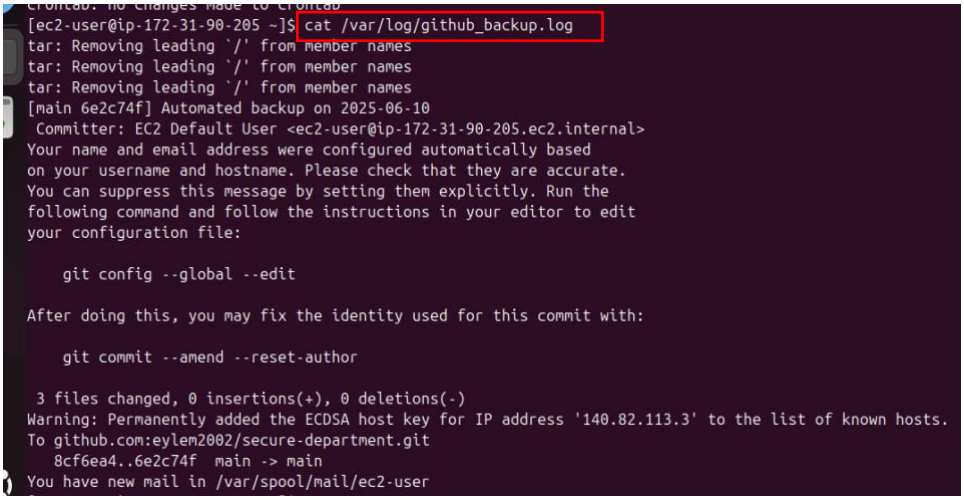


Figure 56-Log Output Showing Successful GitHub Backup Commit

e. Conclusion

This scenario successfully demonstrates the ability to configure a secure and isolated Linux environment for multiple departments using EC2. I also built a working automation flow that archives and backs up data daily to GitHub with zero manual intervention.

Suggestions for improvement:

- Add automated email alerts if the backup script fails.
- Encrypt backup files before pushing to GitHub for additional security.
- Use AWS Systems Manager instead of manual SSH to improve access control and logging.
- Expand the solution to sync with an S3 bucket or centralized logging system in future enterprise environments.

5. Scenario #4 – Serverless HealthBooking on AWS

a. Objective

The goal of this project was to design and deploy a serverless web-based medical appointment booking system using AWS-native tools. The application enables patients to book appointments and allows doctors to manage them. All backend components were built using AWS serverless services, and the frontend was deployed using AWS Amplify.

b. Implementation Steps

1) Designed and Created DynamoDB Tables

- I began by designing the data model for the system. The solution relies on two DynamoDB tables , Both tables were created through the DynamoDB Console using on-demand capacity mode to ensure cost-efficiency and scalability.

- The **Appointments** table stores appointment details including patient name, symptoms, status, and time slot.
- The **Slots** table maintains the status of available time slots to prevent overbooking.

2) Developed Lambda Functions and APIs

- I developed five Lambda functions to manage the backend logic of the application:
  - **GET Time Slots**: Retrieves available (unbooked) time slots.
  - **POST Appointments**: Creates a new appointment, books the time slot, and triggers an SNS email notification.
  - **GET Appointments**: Retrieves all appointment records for the doctor/admin view.
  - **PATCH Appointment Status**: Allows status updates (e.g., Pending, In Progress, Completed) for any appointment.
  - **Reset Daily Bookings**: Scheduled using EventBridge to run daily at 5 PM UTC. It clears all current day bookings and resets slot availability.
    • Each Lambda function was integrated with API Gateway and tested thoroughly before frontend integration.

3) Configured Notifications Using Amazon SNS

- I created an SNS topic to notify the doctor whenever a new appointment is made or the daily reset occurs.

- My email was subscribed to the topic to receive real-time notifications, ensuring visibility into system activity without manual checking.

4) Set Up Daily Automation with EventBridge

- A scheduled rule was set up using Amazon EventBridge to invoke the **Reset Daily Bookings** Lambda function daily at 5 PM UTC.
- The function deletes all appointments for the current day and updates slot statuses to available.
- A confirmation email is sent via SNS after every reset operation to ensure that the system's availability is maintained for the next day.

5) Built and Deployed the Frontend Using AWS Amplify

- The frontend application was built using Vue.js with Vite as the build tool and PrimeVue as the UI framework.

- I forked the provided GitHub repository and updated all API URLs to match my deployed API Gateway endpoints.
- The repository was then connected to AWS Amplify, which was configured for automatic CI/CD deployment triggered on every push to the main branch.

6) Implemented Two Main Web Pages

- **Book Appointment Page**
  - Allows users to enter their name, describe symptoms, and choose a time slot.
  - Submits the booking information to the **POST Appointments** API for processing.
- **Appointments Dashboard**
  - Enables doctors or admins to view all appointments.
  - Includes options to update the status of any appointment using the PATCH API.

7) Set Up Monitoring and Logging Using CloudWatch

- I enabled logging for all Lambda functions through AWS CloudWatch to ensure proper visibility into errors and execution logs.
- Additionally, I configured CloudWatch metrics to monitor API response times and function invocations, helping me debug and improve performance throughout the project lifecycle.

d. Screenshots

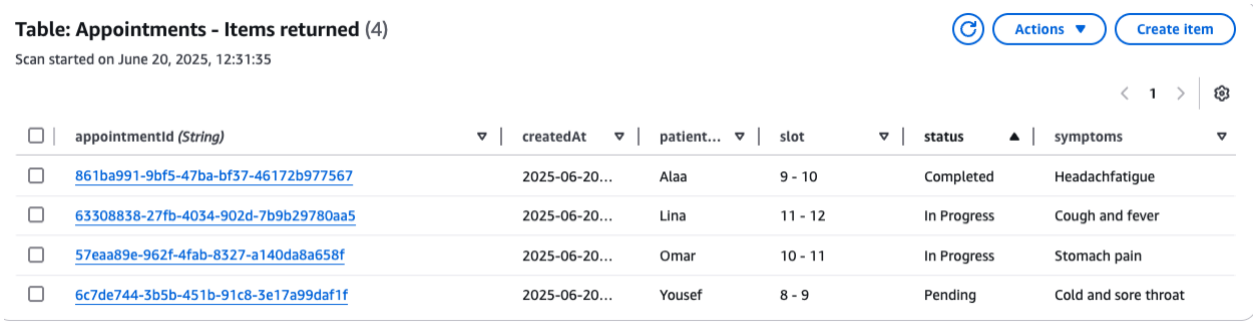


Figure 57-Appointment table

Figure 58-slots table

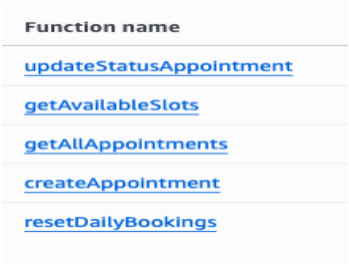


Figure 59-All Lambda Functions

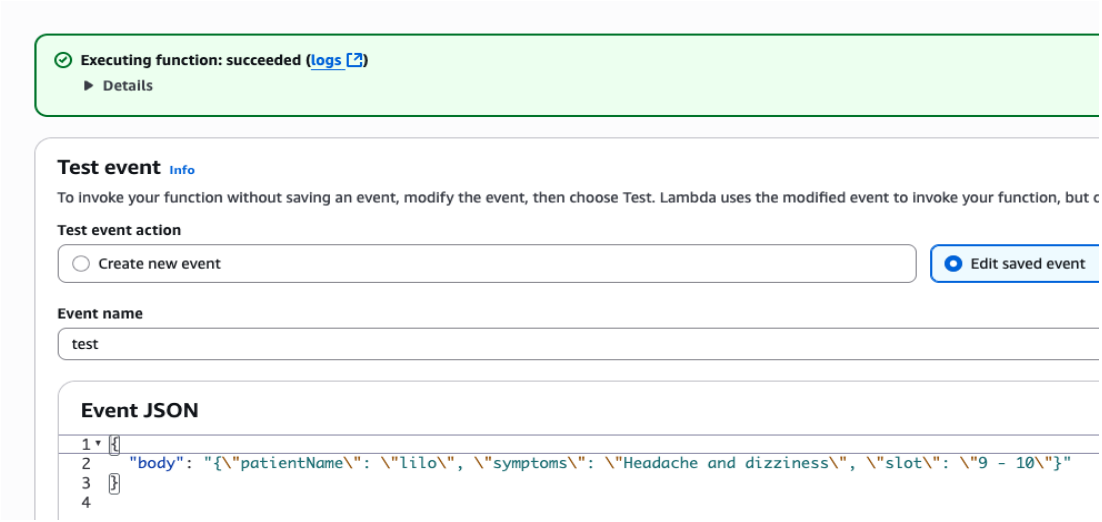


Figure 60- Test create Appointment Function

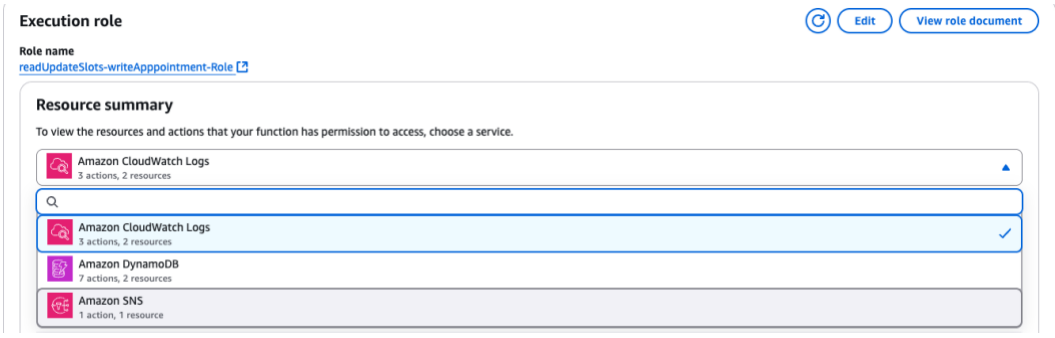


Figure 61-All permissions for create Appointment

Figure 62-Cloudwatch permission for create Appointment

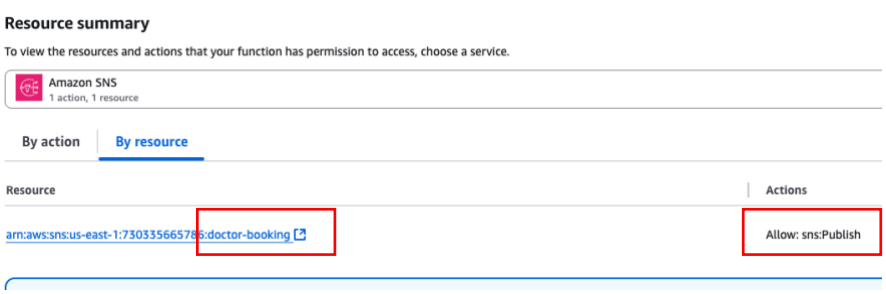


Figure 63-SNS permission for the create Appointment

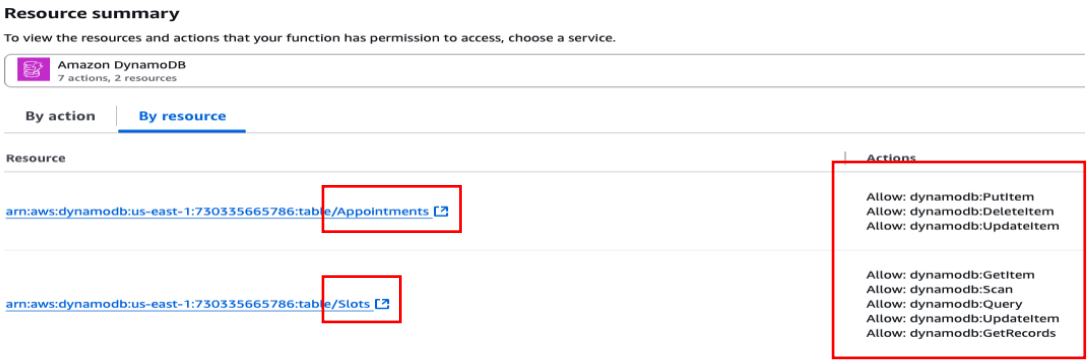


Figure 64-DynamoDB permission for create Appointment

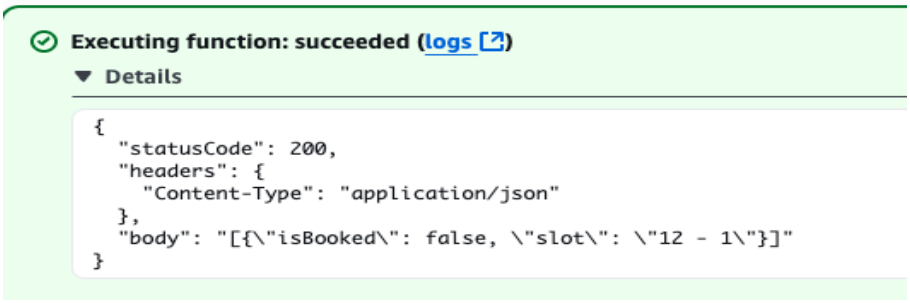


Figure 65 - Test get Available Slots function

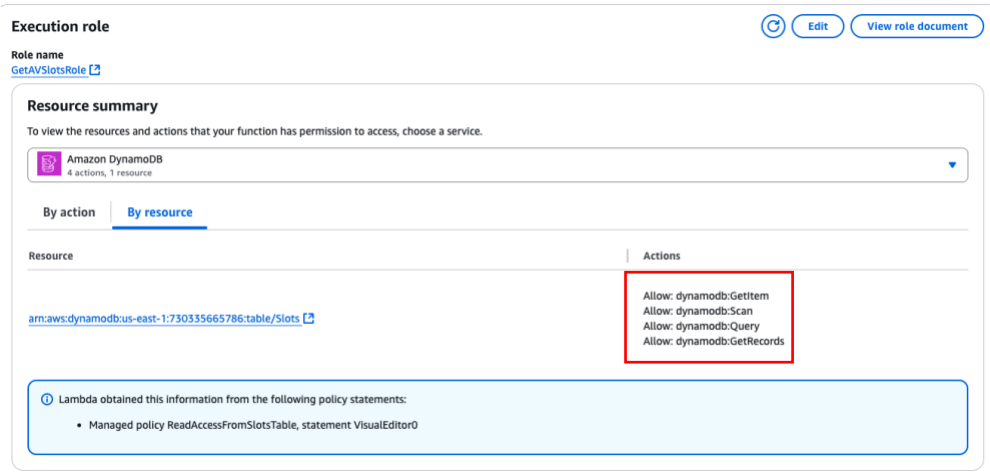


Figure 66-Dynamo Permission for get Available Slots

Figure 67 -Test get All Appointments function

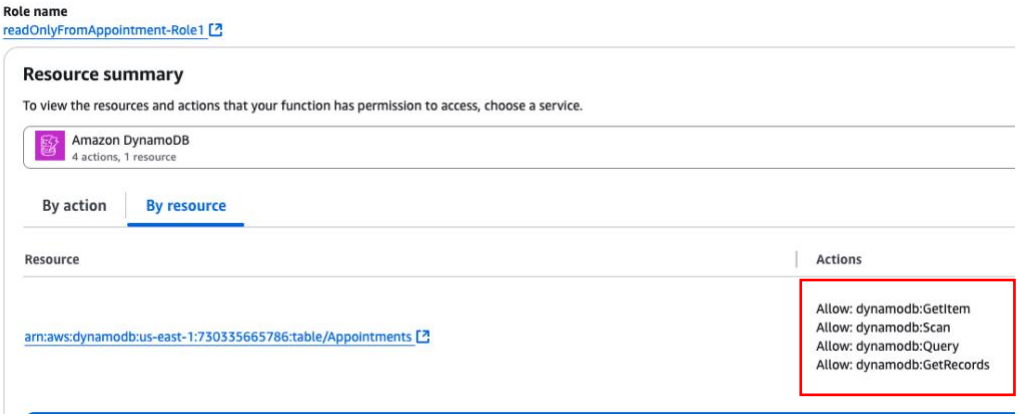


Figure 68 - DyanmoDB permission for get All Appointments

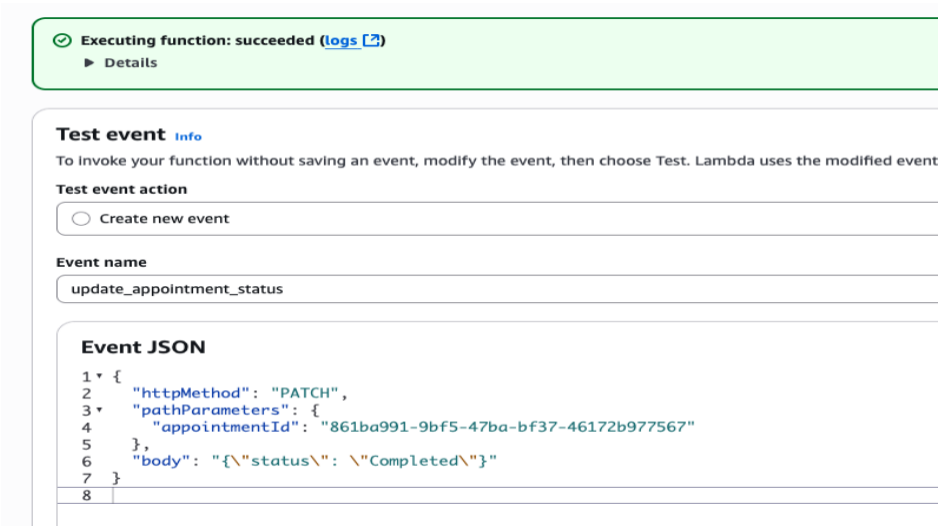


Figure 69-Test update Status Appointment function

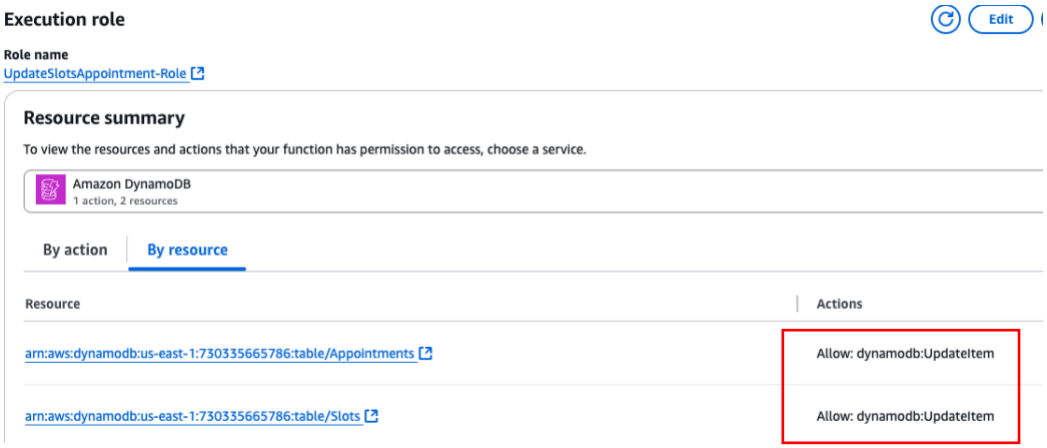


Figure 70 -DynamoDB permission for update Status Appointment

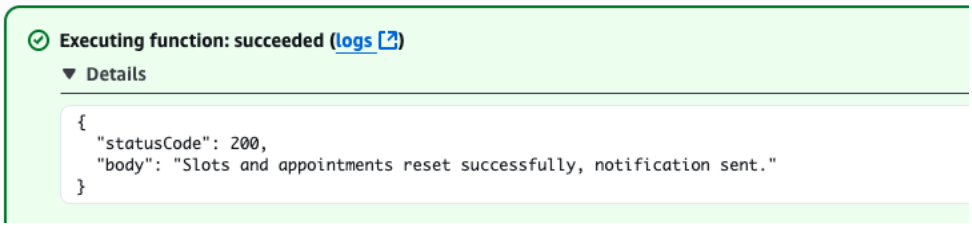


Figure 71-Test for reset daily bookings

Figure 72-All permission for reset daily bookings

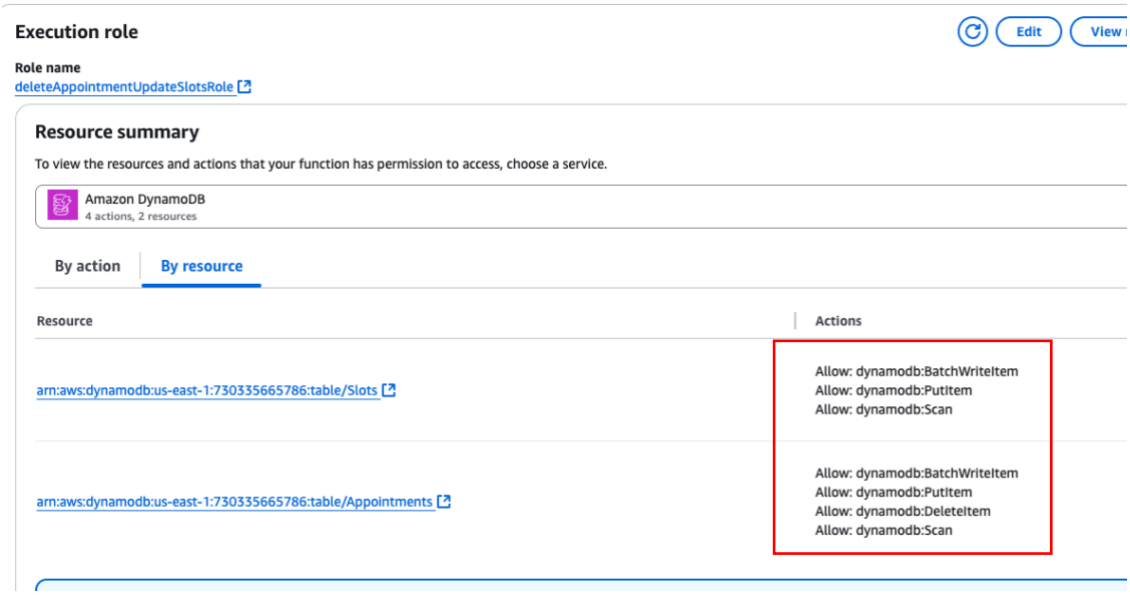


Figure 73-DynamoDB permission for reset daily bookings

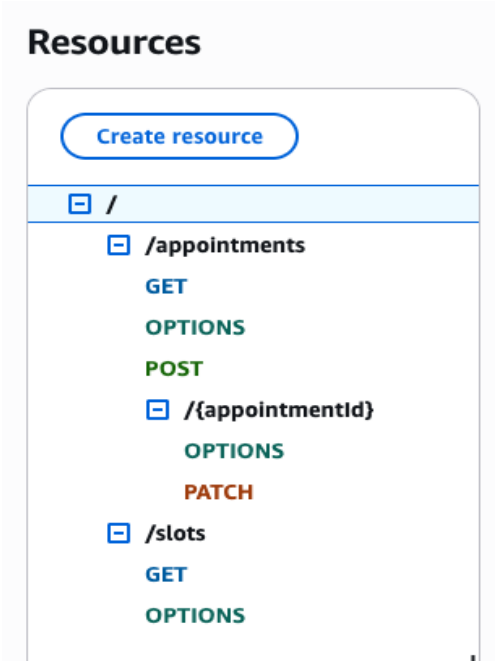


Figure 74-All my APIs

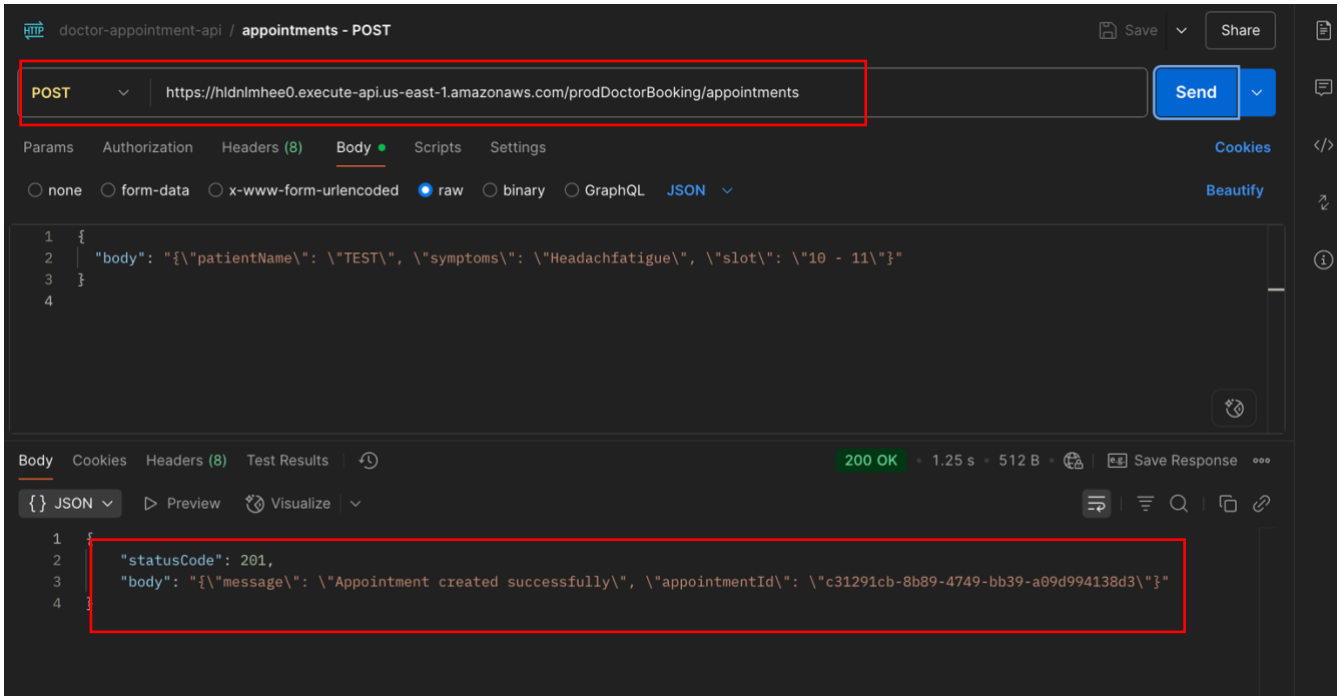


Figure 75-Book an Appointment API Test Postman

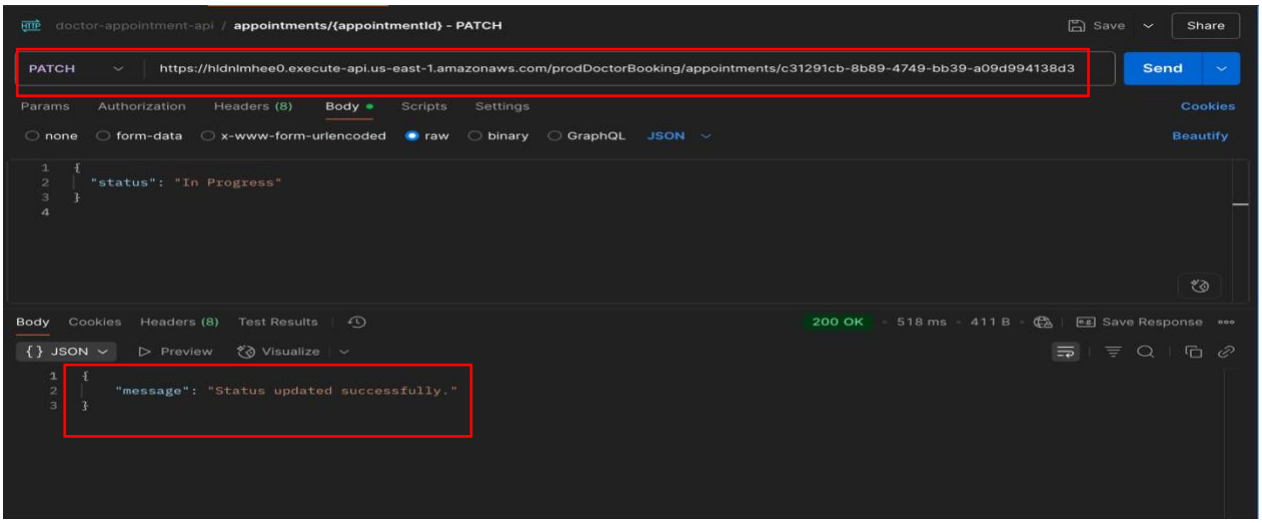


Figure 76-Update Appointment Status API Test Postman

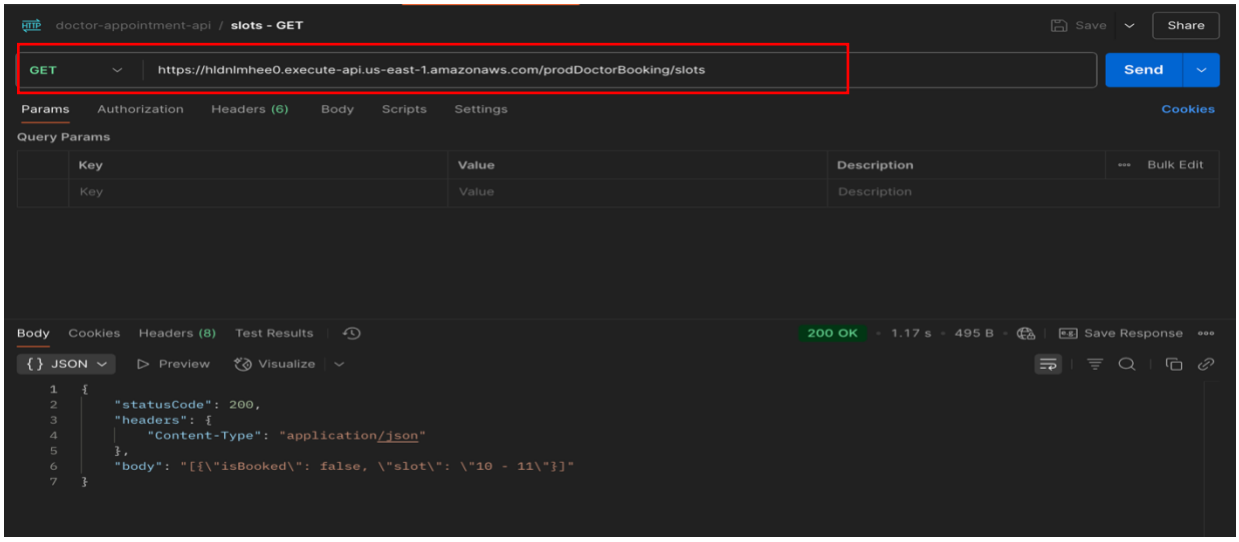


Figure 77-Test Get Available Time Slots API Postman

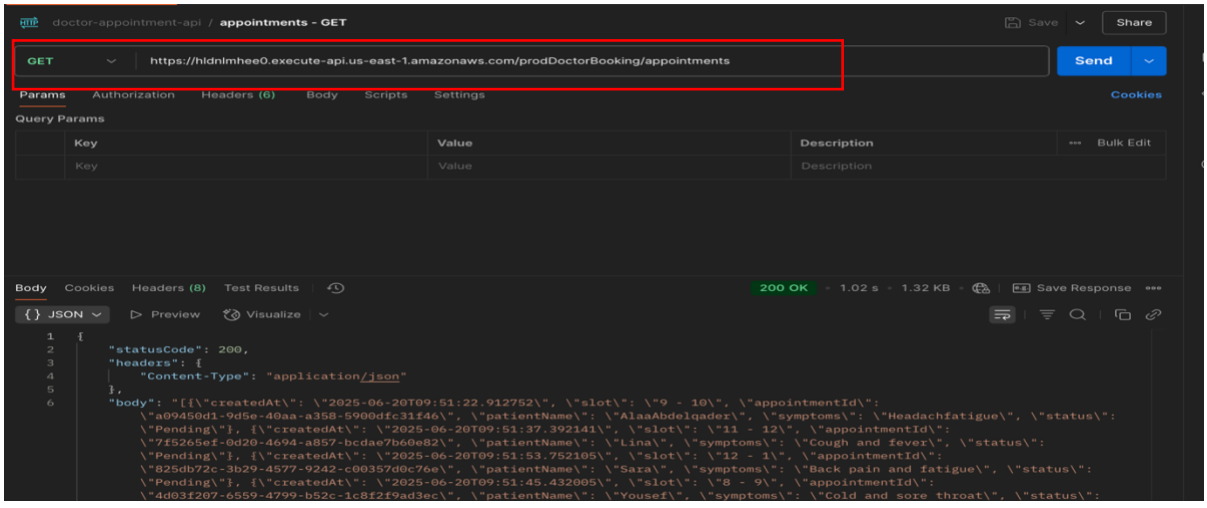


Figure 78-Get Appointments API test Postman

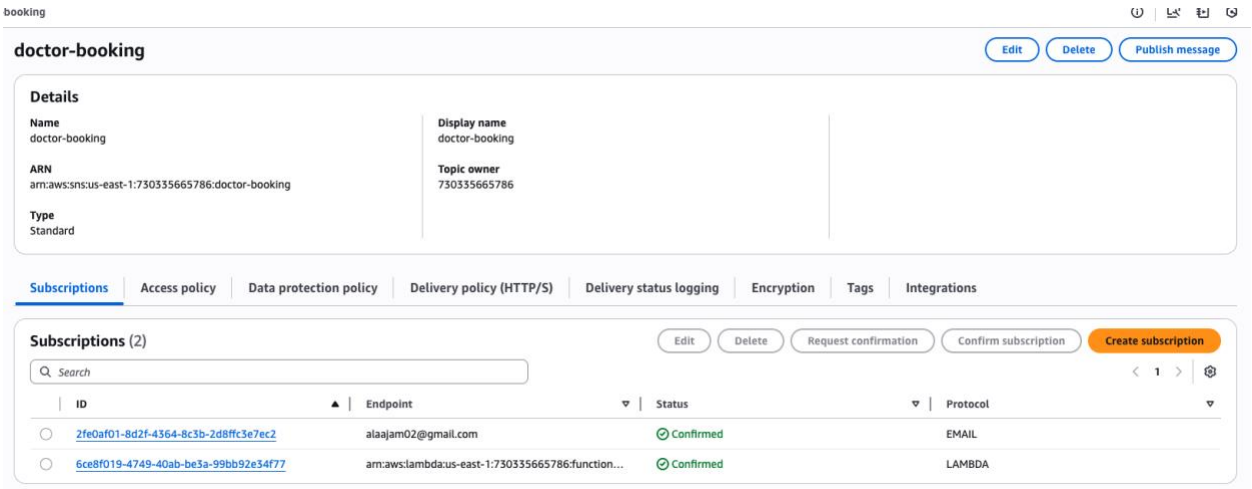


Figure 79 - SNS doctor booking

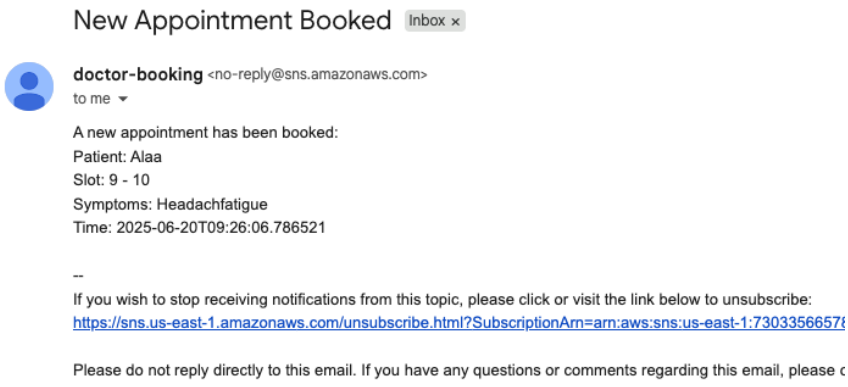


Figure 80 -A new appointment Notification (SNS TEST)

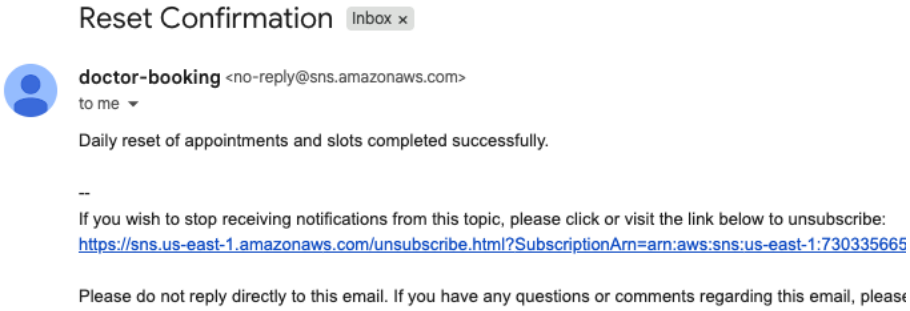


Figure 81 -Reset Confirmation Notification (SNS Test)

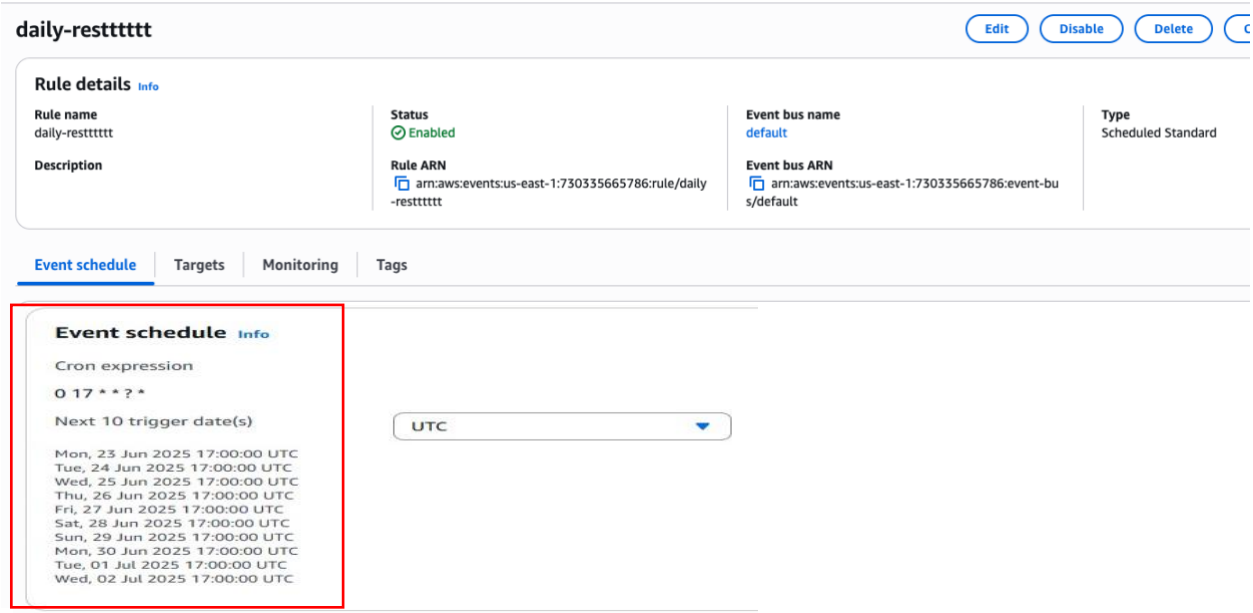


Figure 82- Daily Automation with Event Bridge

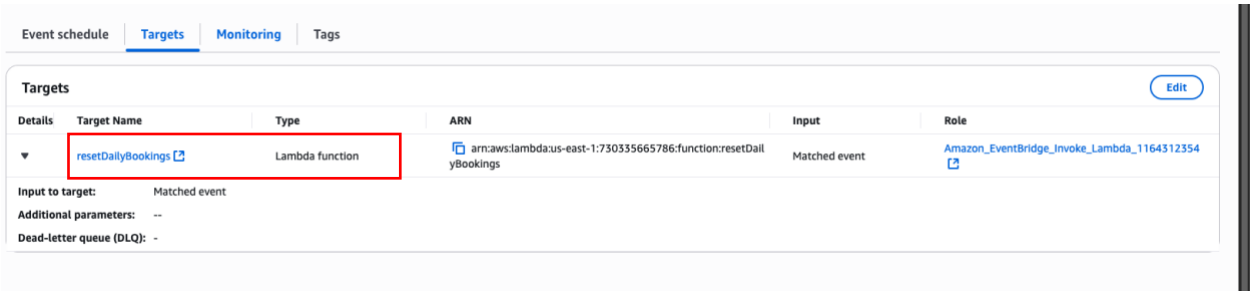


Figure 83 - EventBridge Target

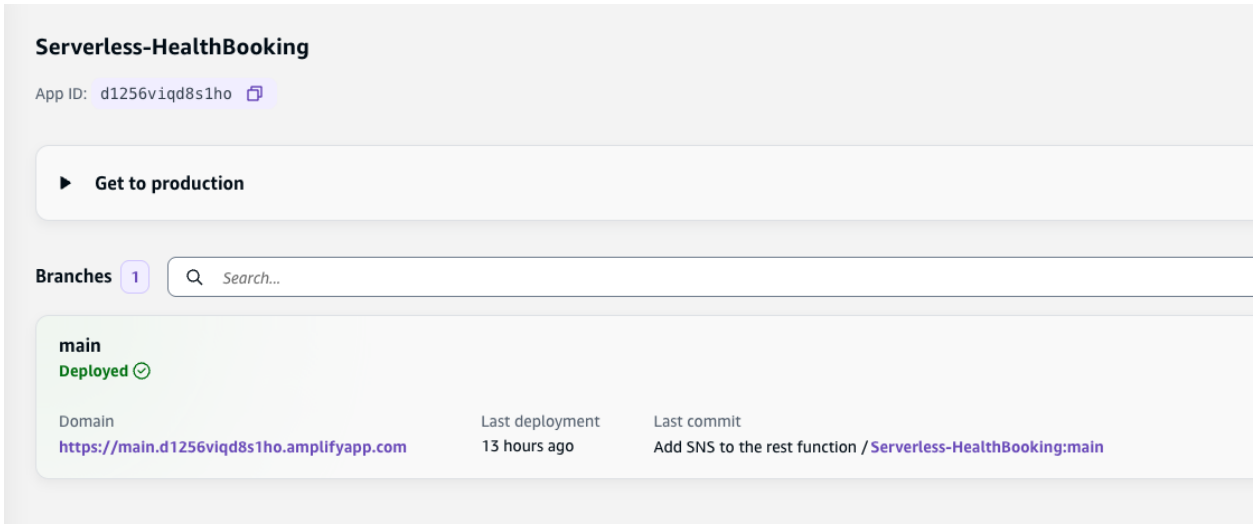


Figure 84-Deployed the Frontend Using AWS Amplify

Figure 85 -Book an Appointment Page

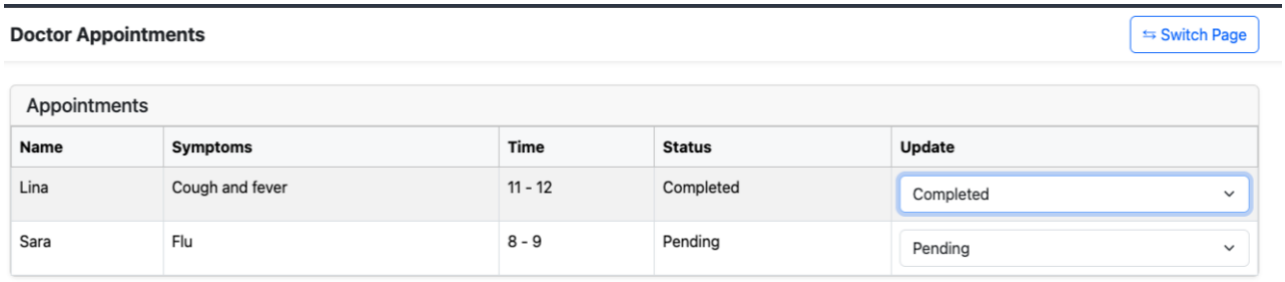


Figure 86-Doctor Appointments page

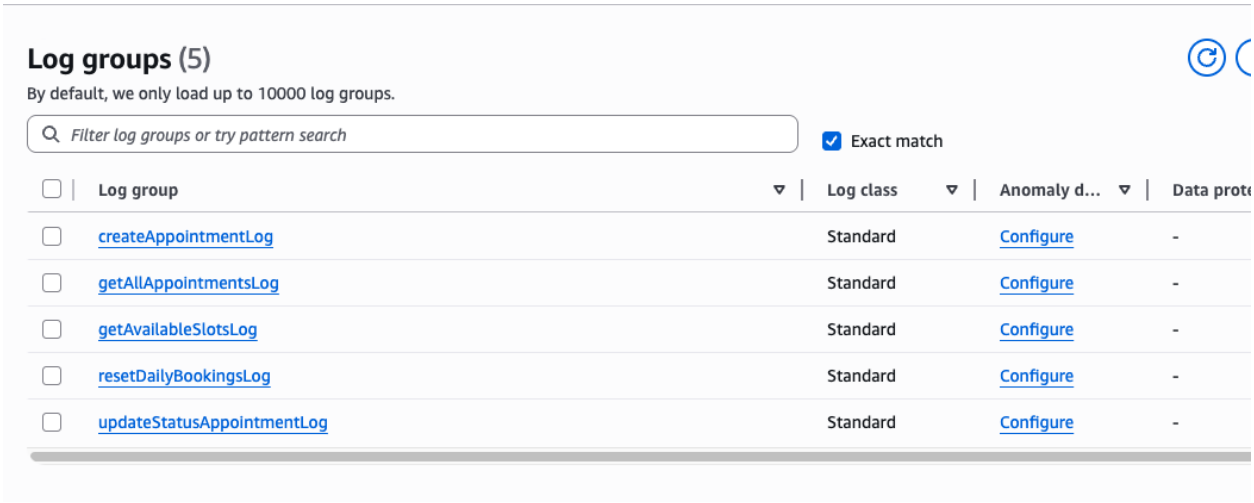


Figure 87 -Enabled logging for all Lambda functions

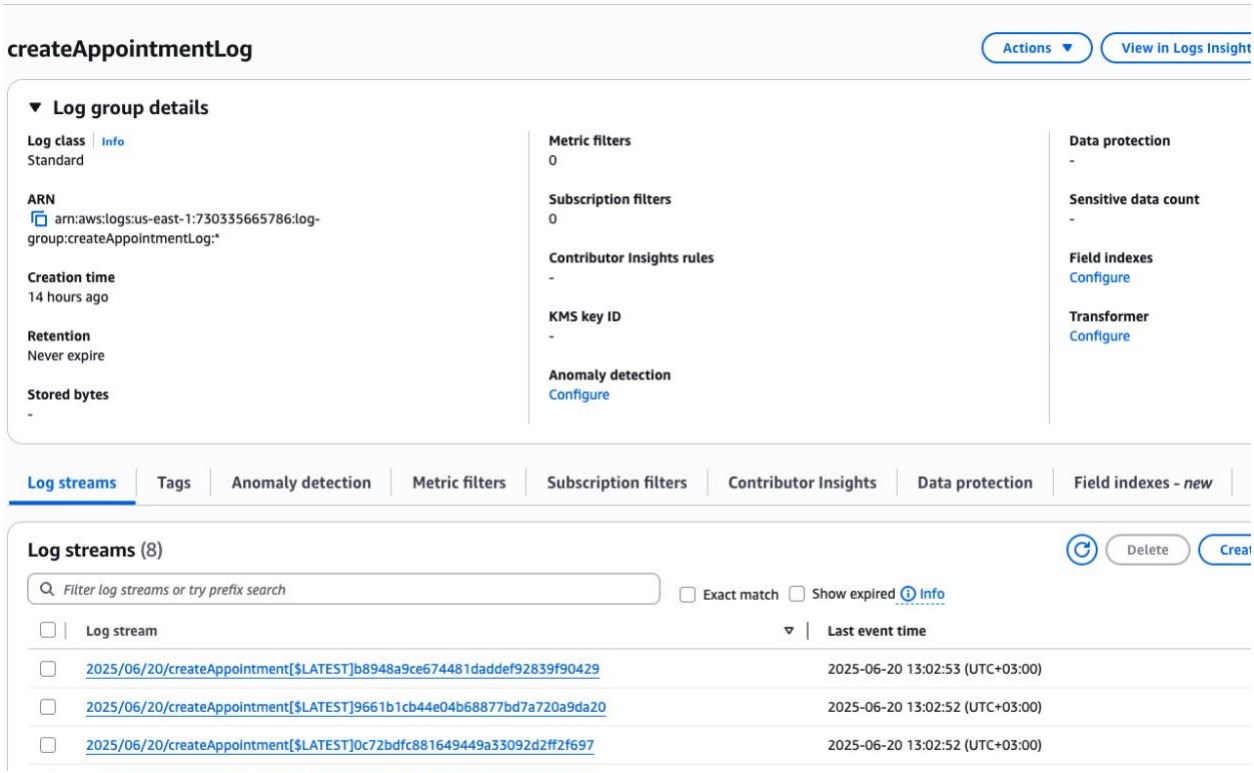


Figure 88 -create Appointment Logging Page

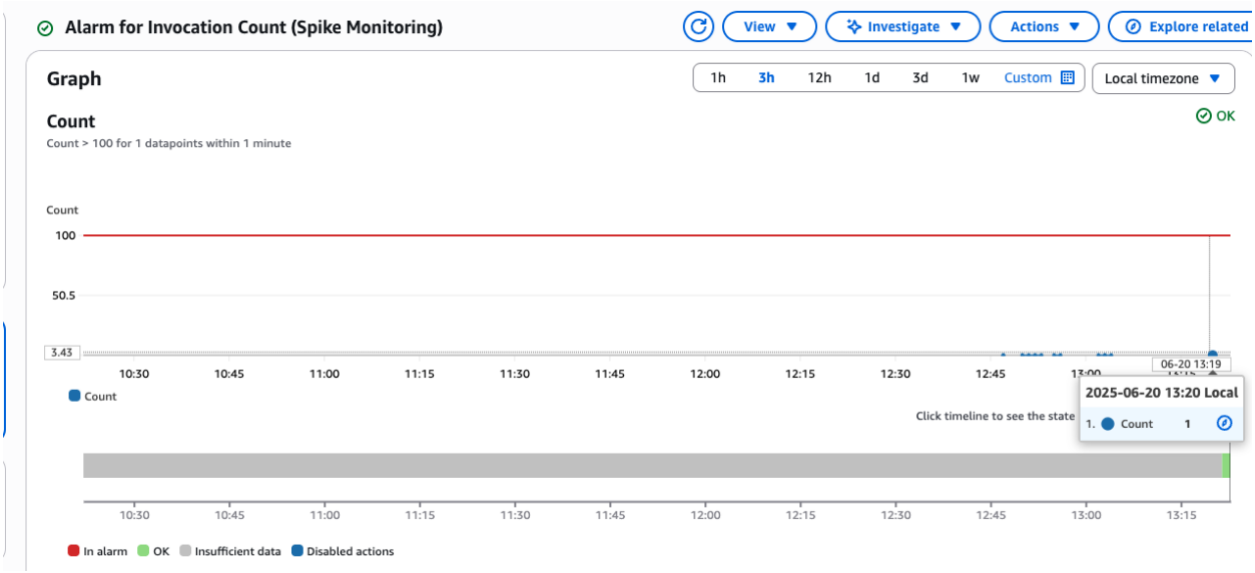


Figure 89- Alarm for Invocation Count (Spike Monitoring)

Alarm for Invocation Count (Spike Monitoring)

This graph monitors the number of API invocations to detect sudden traffic spikes. The red line represents the threshold, set at 100 requests per minute. The blue dots show the actual number of invocations. At 13:20, for example, the count was only 1 request, which is far below the threshold. This confirms that the traffic to the API is currently normal. If the number of invocations suddenly spikes above 100, the alarm will be triggered, alerting me to potential issues such as bot attacks or system errors.

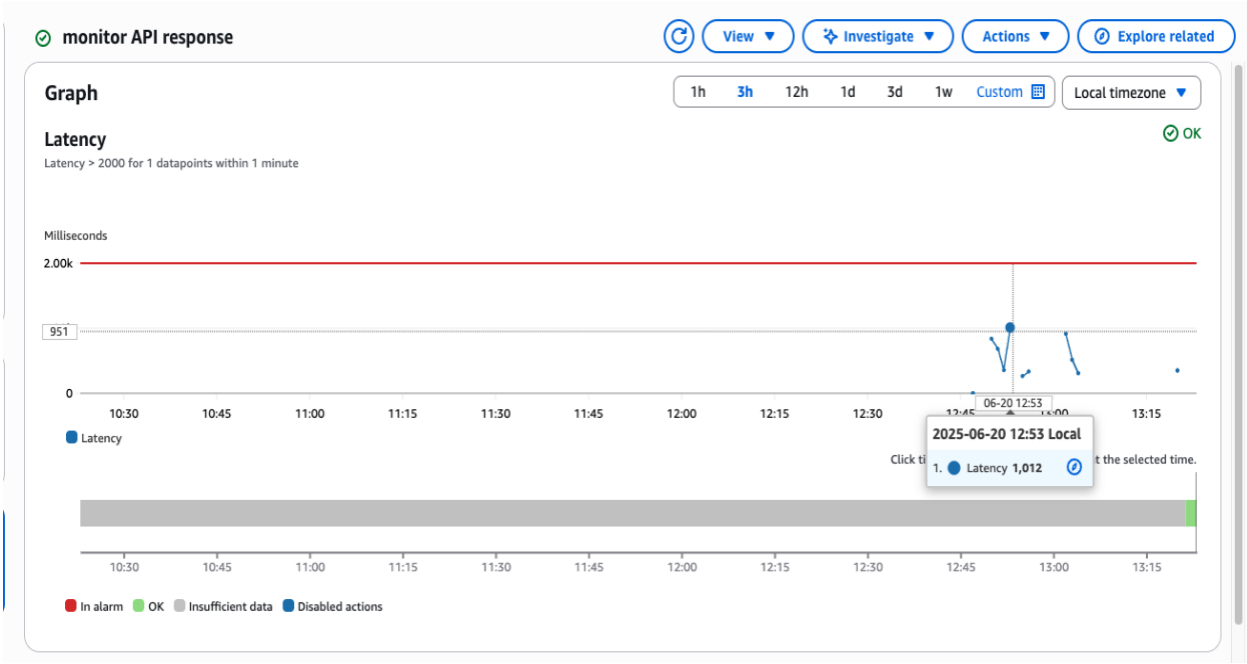


Figure 90 -Alert if Latency > 2s (Cloudwatch)

Alert if Latency > 2s

This graph shows the latency monitoring setup for my API. The red line indicates the threshold set at 2000 milliseconds (2 seconds). The blue dots represent the actual latency recorded during each minute. As seen in the data point at 12:53, the latency was 1012 milliseconds, which is well below the threshold. This indicates that the API is currently performing well and responding within an acceptable time. If latency ever exceeds 2 seconds, CloudWatch will trigger an alarm to help me investigate and take action quickly.

e. Conclusion

This scenario successfully demonstrates the ability to build and deploy a fully serverless medical appointment booking system using AWS-native services. By integrating Lambda, DynamoDB, API Gateway, SNS, EventBridge, CloudWatch, and AWS Amplify, I created a scalable and maintainable architecture with no traditional server management required. The system is secure, cost-effective, and offers a seamless experience for both patients and doctors.

Suggestions for improvement:
• Add user authentication using Amazon Cognito to restrict admin access.
• Implement input validation and security layers such as WAF for enhanced protection.
• Enable detailed metrics and alerts in CloudWatch for real-time monitoring of system health.
• Introduce integration with S3 or RDS to support uploading medical files or patient history in future expansions.

• References

1. Amazon Web Services. (2024). *Amazon S3 Documentation*. Retrieved from https://docs.aws.amazon.com/s3/
2. Amazon Web Services. (2024). *CloudFront Developer Guide*. Retrieved from https://docs.aws.amazon.com/AmazonCloudFront/
3. Amazon Web Services. (2024). *AWS Lambda Developer Guide*. Retrieved from https://docs.aws.amazon.com/lambda/
4. Amazon Web Services. (2024). *Auto Scaling User Guide*. Retrieved from https://docs.aws.amazon.com/autoscaling/
5. Amazon Web Services. (2024). *Amazon Route 53 Documentation*. Retrieved from https://docs.aws.amazon.com/route53/
6. Bash, L. (2022). *Serverless Architectures on AWS*. O'Reilly Media.