AWS Image App with Serverless Architecture

Table of Contents

I. Introduction

This report outlines the deployment of an image caption and thumbnail web application hosting with serverless computing paradigms on AWS. The system showcases cloud-native design principles, including auto-scaling capabilities, event-driven processing, and a three-tier security implementation. Key implementations include automatic horizontal scaling based on CPU utilization, event-driven serverless image processing triggered by S3 object storage events, secure a three-tier network topology with subnet isolation, and integration with external AI for content generation. The system also employs practices including load balancing, health monitoring and launch automation.

II. Architecture Diagram

The integration between web application and serverless components occurs through shared AWS services:

- **S3 Bucket:** Central storage repository accessed by both Flask (upload) and Lambda functions (processing).
- **RDS Database:** Shared data store containing metadata, captions, and thumbnail references.
- **IAM Roles:** Cross-service authentication enabling secure resource access.
- **VPC Networking:** Lambda functions deployed within the same VPC for database connectivity.

Web Application Architecture

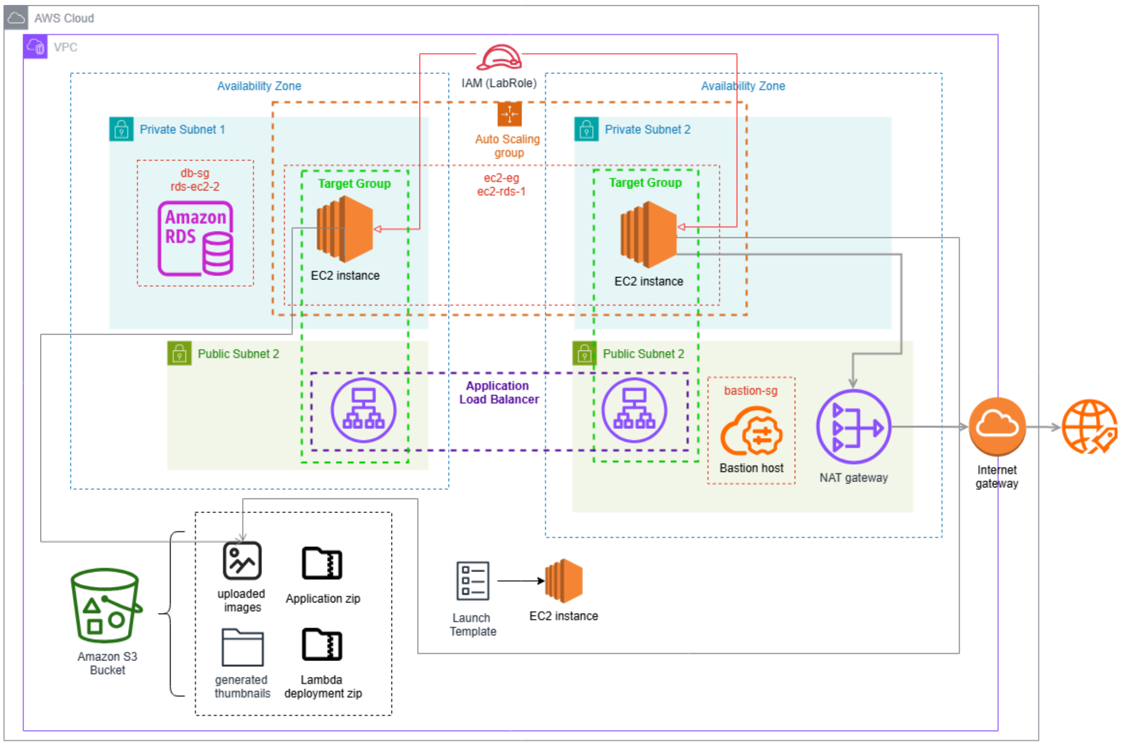


Figure 1 Web application architecture

- **Presentation layer:** Application Load Balancer (ALB) in public subnets across two Availability Zones, and the target group for health checks.
- **Application layer:** Flask web servers in private subnets with auto-scaling capabilities of EC2 instances, created with a launch template.
- **Data layer:** RDS MySQL instance in isolated private subnets.

Other components include Bastion host in public subnet for administrative database access. The entire deployment is in one VPC with custom route tables, a NAT gateway for private subnet internet access to the caption lambda function, and security groups implementing least-privilege access. The IAM role is the default `LabRole` for AWS lab.

Serverless Architecture

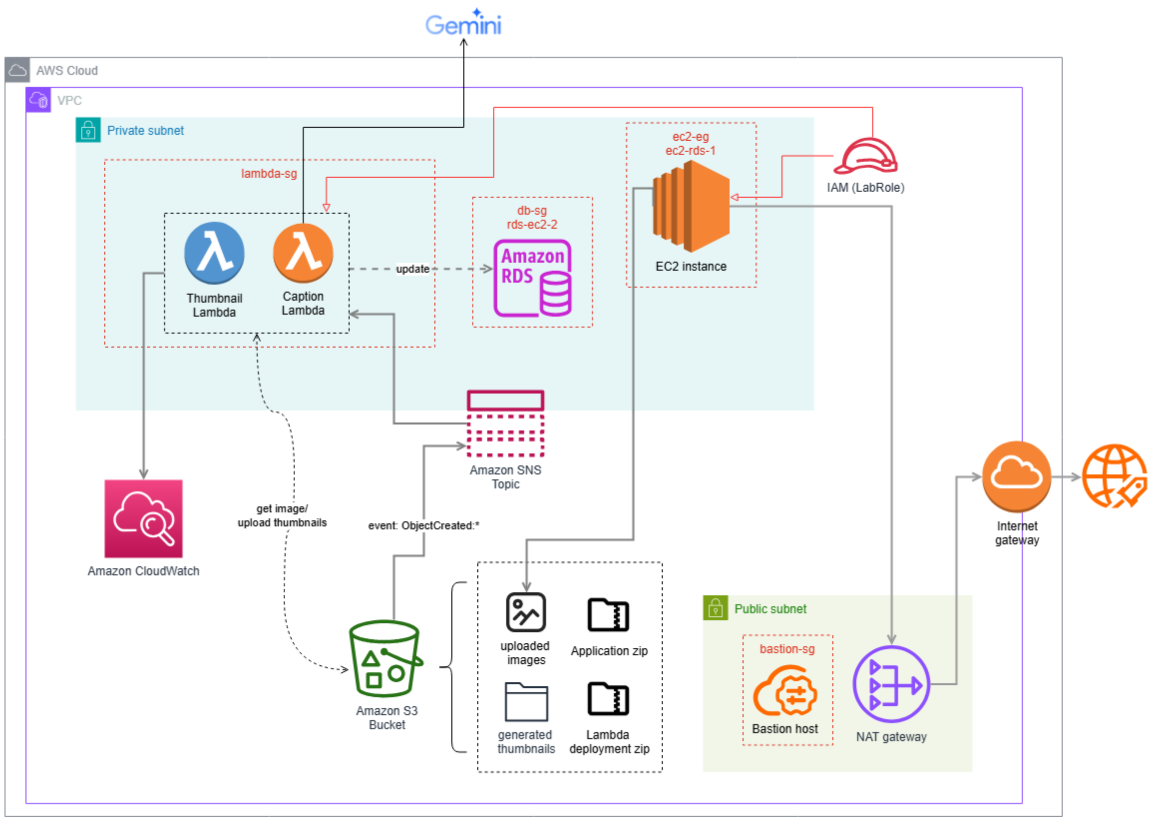


Figure 2 Serverless architecture

The event flow is as flows:

1. Each image uploaded to S3 bucket is given a unique key. This “image key” along with the “image file name” and “upload timestamp” are updated in RDS database.
2. Image upload triggers S3 ObjectCreated event.
3. SNS topic receives event and fans out to two subscriptions invoking Lambda functions: (1) Caption Lambda function; (2) Thumbnail Lambda function. CloudWatch is used for logging and monitoring them.
4. Caption Lambda retrieves image and calls external Gemini API.
5. Thumbnail Lambda processes image and generates resized thumbnail version.
6. Both functions update RDS database with processing results. The fields updated are “captions”, “generated thumbnails file name”, and “generation timestamp”.

III. Web Application Deployment

a. Compute Environment

The web application tier utilizes EC2 instances managed through an Auto Scaling Group (ASG) to provide elastic compute capacity that automatically adjusts to demand.

ASG Configuration

- **Launch Template**: Instance (t3.micro of 1 vCPU, 1 GB RAM) selected for cost optimization in development environment. Machine Image is Amazon Linux 2023 AMI. Every EC2 instance is attached 2 security group. The first security group is `ec2-rds-1`, allowing outbound MySQL traffic to RDS DB on port 3306 so it can update the DB entries upon image upload. The second security group is `ec2-sg` allowing inbound traffic only from ALB on port 5000, and inbound SSH from the bastion host (`bastion-sg`) on port 22 for testing. It does not limit outbound traffic.
- **IAM Instance Profile:** `LabRole` with permissions for S3 access and RDS connectivity.
- **Auto Scaling Policies:**
    - **Min and Max Capacity:** 1 instance minimum ensuring basic availability. 3 instances maximum preventing cost overruns while addressing traffic spikes. Note that this configuration is used for the auto scaling test, and is different in the video demo (set to 2-2).
    - **Desired Capacity:** 1 instance for normal operations.
    - **Scaling Policy:** Target tracking based on 70% average CPU utilization.
    - **Health Check:** ELB health checks ensuring traffic only reaches healthy instances, with a grace period of 300 seconds allowing sufficient startup time.
- **Network Configuration:**
    - **VPC:** Project VPC `as2-project-vpc` (10.0.0.0/16) with DNS hostnames and resolution enabled.
    - **Subnets:** Deployment across two private subnets (10.0.128.0/20, 10.0.144.0/20) in different Availability Zones.
    - **Internet Access:** NAT Gateway in public subnet 1 enabling outbound internet connectivity for VPC-connected lambda functions in the private subnets to connect to the internet, needed for external API call.
    - **Route Tables:** Private subnet route tables directing 0.0.0.0/0 traffic through NAT Gateway.
- **Security Configurations** (Three-Tier implementation):

The 3-tier implementation essentially only allows traffic to flow from ALB to EC2, and from EC2 to RDS.

- Tier 1 - Presentation layer:
  - **ALB Security Group:** The security group `LB-SG` only permits HTTP traffic on port 80 from internet (0.0.0.0/0). No limits on outbound.
  - **Public Subnet:** ALB deployed in public subnets 1 and 2 with internet gateway (IGW) access.
- Tier 2 - Application layer:
  - **EC2 Security Group:** The relevant rule is defined in the security group `ec2-sg` which restricts inbound traffic to port 5000 from ALB security group only. Other rules include inbound SSH from the bastion host security group (`bastion-sg`) on port 22 for testing, and outbound MySQL traffic defined in `ec2-rds-1` to RDS DB on port 3306.
  - **Private Subnet:** EC2 instances are in private subnets 1 and 2, and they have no direct internet access, only through NAT Gateway.
- Tier 3 - Data layer:
  - **RDS Security Group:** The security group `db-sg` and `rds-ec2-1` allows MySQL traffic (port 3306) only from Lambda security groups and EC2, respectively.
  - **Private Subnet:** RDS DB locate in private subnets 1 and 2 for security.

- **Administrative Access**

  A Bastion Host is configured to secure administrative access such as DB schema management and troubleshooting access to EC2 instances on private subnets. A keypair is created for this bastion host and must be used for SSH access. The bastion instance is created as a t2.micro instance in public subnet 1. It as a bastion security group (`bastion-sg`) SSH access (port 22) restricted to administrator IP addresses, such as my own IP address.

- **Application Load Balancer (ALB) Setup**

  The ALB ensures high availability by automatically detecting unhealthy instances and routing traffic only to healthy targets. Integration with ASG enables seamless instance replacement during failures.

  - **Scheme:** Internet-facing to accept traffic from external users. Round robin distribution ensuring even traffic distribution. We disable sticky sessions as application is stateless.

- **Public Subnets:** Deployed across public subnets 1 and 2 for high availability.
- **Listener:** ALB is listening for HTTP protocol on port 80, forwarding to target group, which selects a healthy EC2 instance exposing the app on port 5000. This port is set for development/debug purpose. Ideally in production, the flask app should be running on port 80 instead of 5000.
  - **Target Group:** `as2-target-group-5000` is used for HTTP health checks on "/health" endpoint. It is checking with an interval of 30 seconds. The time is 5 seconds. The healthy threshold is 2 consecutive successes, and the unhealthy threshold is 3 consecutive failures.

b. Database Environment (RDS MySQL)

The database tier implements a managed MySQL 8.0.35 instance providing ACID compliance, automated backups, and high availability options.

The instance is configured as an db.t3.micro (2 vCPU, 1 GB RAM) for development. It has a storage of 20 GB General Purpose SSD with encryption at rest enabled. Multi-AZ is disabled for cost optimization. See Appendix A for the script for DB creation.

- **VPC:** Same VPC (`as2-project-vpc)` as application tier ensuring low-latency connectivity.
- **Private Subnets:** Spanning two private subnets 1 and 2 across AZs.
- **Security Group:** The security group `db-sg` and `rds-ec2-1` allows MySQL traffic (port 3306) only from Lambda security groups and EC2 application.
- **Access Policy**: master username (admin) with defined password. It also uses IAM (`LabRole`) DB authentication for enhanced security.

c. Storage Environment (S3 bucket)

The storage layer utilizes S3 for object storage with integrated event notification capabilities.

The bucket is configured as `as2-image-app-deploy-bucket`, in region us-east-1 for consistency with other services. Versioning is disabled for cost optimization, and server-side encryption (SSE-S3) is enabled by default. The bucket policy is set to block public access and restrict access to authenticated IAM principals (`LabRole`) only.

The zip files for application files and lambda functions are uploaded to the bucket, one to be downloaded by the EC2 instances and one to be accessed when configuring the lambda functions. The images will be uploaded to the root directory of the S3 bucket, and the Lambda-generated thumbnails will be put in a pseudo-folder “thumbnails/”. The trigger (event notification) for lambda functions is as follows:

- **Source:** S3 ObjectCreated events (s3:ObjectCreated:*). This can be extended to further limit to prefix (e.g., upload to specific folder) and suffix (e.g., jpg only).
- **Destination:** SNS Topic (`as2-image-processing`).
- **Fan-out:** SNS distributes events to 2 Lambda subscribers for Caption Lambda and Thumbnail Lambda.
- **Filter Rules:** This is important to exclude “thumbnails/” prefix to prevent recursive triggers.

IV. Serverless Component Deployment

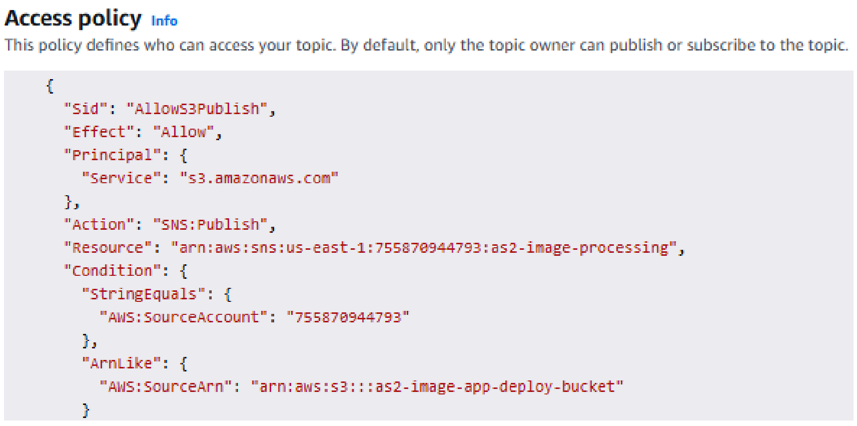
a. Event-Driven Architecture

The serverless architecture implements an event-driven pattern where S3 object creation events trigger downstream processing workflows through Amazon SNS topic fan-out.

As mentioned in the previous section, the event source is the S3 bucket which is configured with `ObjectCreated` event notifications. The event router is a SNS topic which receives S3 events and distributes to multiple subscribers. The event processors are the Caption Lambda and Thumbnail Lambda functions subscribed to the SNS topic for parallel processing. And the processing results are stored in the shared RDS DB for persistence.

The SNS topic `as2-image-processing` is configured as a standard topic for reliable delivery. There are two subscribers corresponding to the two Lambda functions for annotation and thumbnail generation. Dead Letter Queue is configured for failed message handling. And default retry mechanism is used with exponential backoff.

Note that within this SNS topic, an access policy for the S3 service needs to be defined for S3 to access the SNS.



Access policy Info

This policy defines who can access your topic. By default, only the topic owner can publish or subscribe to the topic.

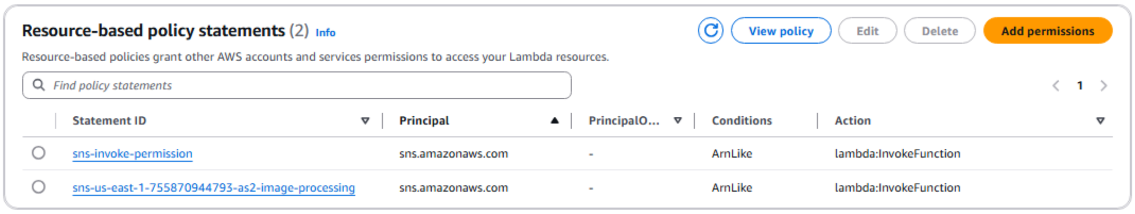
```
{
  "Sid": "AllowS3Publish",
  "Effect": "Allow",
  "Principal": {
    "Service": "s3.amazonaws.com"
  },
  "Action": "SNS:Publish",
  "Resource": "arn:aws:sns:us-east-1:755870944793:as2-image-processing",
  "Condition": {
    "StringEquals": {
      "AWS:SourceAccount": "755870944793"
    },
    "ArnLike": {
      "AWS:SourceArn": "arn:aws:s3:::as2-image-app-deploy-bucket"
    }
```

Figure 3 Access policy for S3 service

b. Annotation Function (Caption Lambda)

The annotation function is deployed as a Caption Lambda function with external dependencies packaged in a deployment artifact. It is packaged as a zip file containing function code with the handler and dependencies, and it is uploaded to the S3 bucket. See Appendix B for `caption_lambda.py` code.

For configuration, a memory of 512 MB is allocated for the Gemini API processing requirements. And timeout is set to 5 minutes to accommodate API response times. It uses the same project VPC as other application components. It is in private subnets with NAT Gateway for internet access, to call the external API. It has a `lambda-sg` security group that allows outbound MySQL traffic (port 3306) to the RDS DB. The IAM role attached is the `LabRole` which includes policies for CloudWatch Logs access for function logging, S3 GetObject permissions for image retrieval, RDS connectivity for database updates, and VPC management for network interface creation. However, this role lacks an additional resource-based policy that is the permission for the SNS topic (`as2-image-processing`) to invoke this lambda function.



Resource-based policy statements (2) Info

Resource-based policies grant other AWS accounts and services permissions to access your Lambda resources.

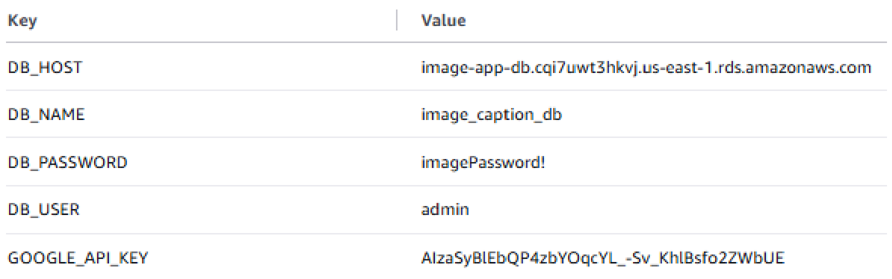
Statement ID	Principal	PrincipalO...	Conditions	Action
sns-invoke-permission	sns.amazonaws.com	-	ArnLike	lambda:InvokeFunction
sns-us-east-1-755870944793-as2-image-processing	sns.amazonaws.com	-	ArnLike	lambda:InvokeFunction

Figure 4 Caption Lambda SNS policy

The policy JSON is as follows:

```
{
  "Version": "2025-6-1",
  "Id": "default",
  "Statement": [
    {
      "Sid": "sns-invoke-permission",
      "Effect": "Allow",
      "Principal": {
        "Service": "sns.amazonaws.com"
      },
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-east-1:755870944793:function:caption-function",
      "Condition": {
        "ArnLike": {
          "AWS:SourceArn": "arn:aws:sns:us-east-1:755870944793:as2-image-processing"
        }
      }
    },
    {
      "Sid": "sns-us-east-1-755870944793-as2-image-processing",
      "Effect": "Allow",
      "Principal": {
        "Service": "sns.amazonaws.com"
      },
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-east-1:755870944793:function:caption-function",
      "Condition": {
        "ArnLike": {
          "AWS:SourceArn": "arn:aws:sns:us-east-1:755870944793:as2-image-processing"
        }
      }
    }
  ]
}
```

DB credentials and Gemini API key are defined as environment variables, as listed below:



Key	Value
DB_HOST	image-app-db.cqi7uwt3hkvj.us-east-1.rds.amazonaws.com
DB_NAME	image_caption_db
DB_PASSWORD	imagePassword!
DB_USER	admin
GOOGLE_API_KEY	AIzaSyBlEbQP4zbYOqcYL_-Sv_KhlBsfo2ZWbUE

Figure 5 Caption Lambda environment variables

c. Thumbnail Generator (Thumbnail Lambda)

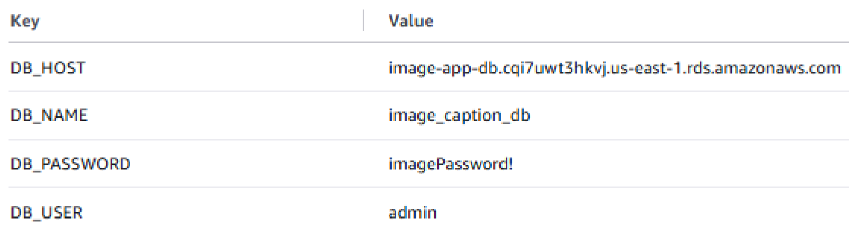
Similarly, the thumbnail function implements image resizing capabilities using the Python PIL library within the Thumbnail Lambda. It is also It is packaged as a zip file containing function code with the handler and dependencies, and it is uploaded to the S3 bucket. The function supports for JPEG, PNG, GIF formats. The thumbnail generated is set to be a JEPG file of size 200x200 pixels. The processing workflow is: (1) download image from S3, (2) open image using PIL library, (3) resize maintain aspect ratio, (4) optimize for web delivery, (5) upload thumbnail to S3 “thumbnails/” folder, and (6) update RDS DB with thumbnail metadata. See Appendix B for `Thumbnail_lambda.py` code.

For configuration, a memory of 512 MB is allocated for image processing, and timeout is set to 5 minutes. It uses the same project VPC, and it is in private subnets. It has a `lambda-sg` security group that allows outbound MySQL traffic (port 3306) to the RDS DB. The IAM role attached is the `LabRole`, and similarly we need an additional resource-based policy for the SNS topic to invoke this lambda function.

The policy JSON is same as the Caption Lambda, with the exception of “resource” section being the thumbnail function ARN. It is defined as follows:

```
{
  "Version": "2025-6-1",
  "Id": "default",
  "Statement": [
    {
      "Sid": "sns-invoke-permission",
      "Effect": "Allow",
      "Principal": {
        "Service": "sns.amazonaws.com"
      },
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-east-1:755870944793:function:thumbnail-function",
      "Condition": {
        "ArnLike": {
          "AWS:SourceArn": "arn:aws:sns:us-east-1:755870944793:as2-image-processing"
        }
      }
    },
    {
      "Sid": "sns-us-east-1-755870944793-as2-image-processing",
      "Effect": "Allow",
      "Principal": {
        "Service": "sns.amazonaws.com"
      },
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-east-1:755870944793:function:thumbnail-function",
      "Condition": {
        "ArnLike": {
          "AWS:SourceArn": "arn:aws:sns:us-east-1:755870944793:as2-image-processing"
        }
      }
    }
  ]
}
```

DB credentials are defined as environment variables, as listed below:

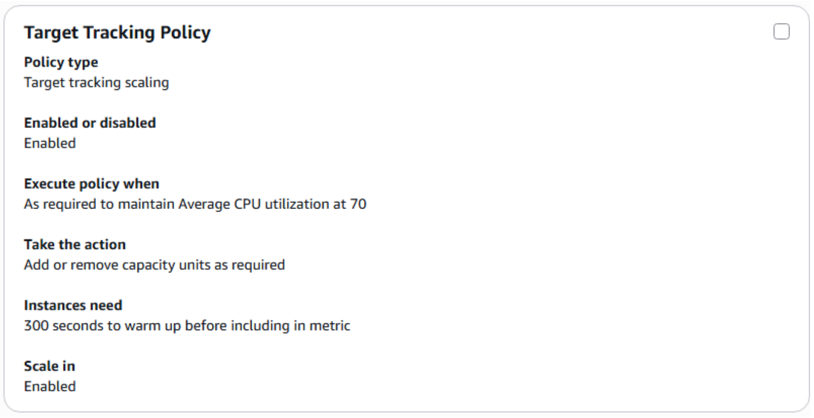


Key	Value
DB_HOST	image-app-db.cqi7uwt3hkvj.us-east-1.rds.amazonaws.com
DB_NAME	image_caption_db
DB_PASSWORD	imagePassword!
DB_USER	admin

Figure 6 Thumbnail Lambda environment variables

V. Auto Scaling Test

A load testing strategy was implemented to validate the auto-scaling behaviour of the EC2 Auto Scaling Group under sustained traffic load. Note that the desired capacity is set to 1, minimum as 1 and maximum as 3 (*This configuration is different to the video demo as the scaling test was performed after the video was recorded*). Since the target group is checking CPU utilization as shown below, we use a Python script using requests library and threading.


Target Tracking Policy

Policy type
Target tracking scaling

Enabled or disabled
Enabled

Execute policy when
As required to maintain Average CPU utilization at 70

Take the action
Add or remove capacity units as required

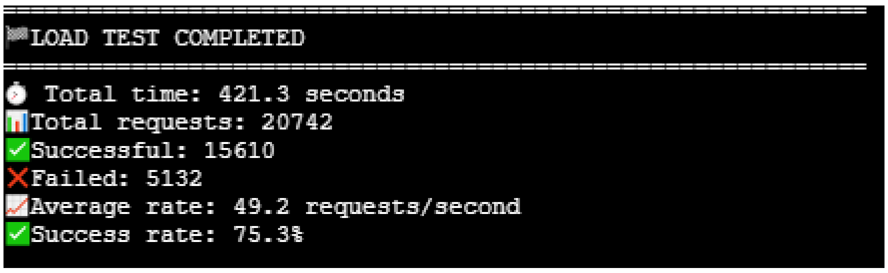
Instances need
300 seconds to warm up before including in metric

Scale in
Enabled

Figure 7 Target group CPU tracking policy

We run the load test in the bastion host. We expect 1 additional instance to launch in response to the CPU spike, then terminate when the load test finishes, and CPU drops back to a normal range. The load test configuration and testing results are as below:

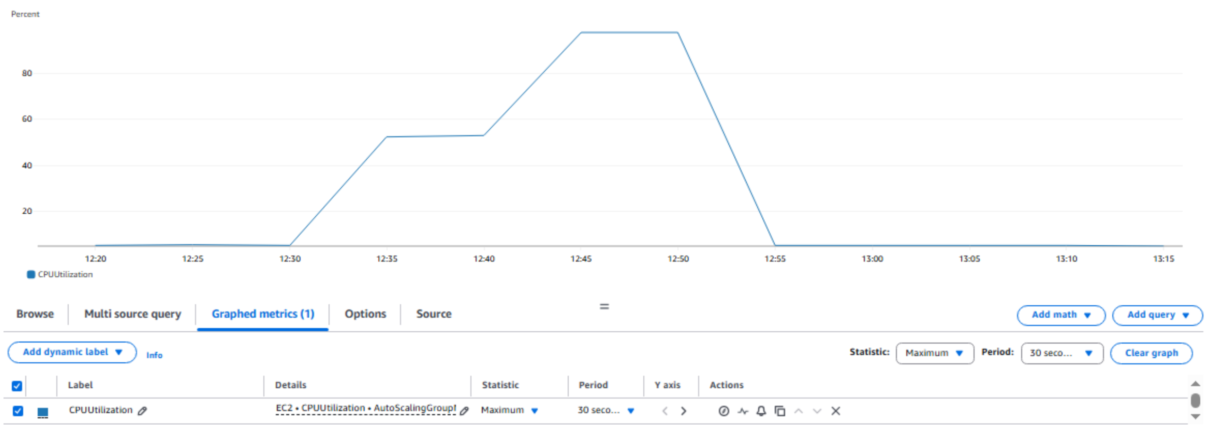
- **Target Endpoint**: /health; /gallery page to simulate realistic user traffic
- **Thread**: 8 burst threads (rapid fire); 6 sustained threads (50 req/sec each); 4 concurrent threads (10 concurrent per burst)
- **Request Rate**: ~500 requests per second
- **Duration**: 6 minutes (360 seconds) sustained load
- **Request Pattern**: Continuous GET requests with 0.5-second intervals


```
============================================================
🏁LOAD TEST COMPLETED
============================================================
⏱ Total time: 421.3 seconds
📊Total requests: 20742
✅Successful: 15610
❌Failed: 5132
📈Average rate: 49.2 requests/second
✅Success rate: 75.3%
```

We monitor the load test using CloudWatch Metrics (CPU utilization), ASG Activity including Instance launch and termination events, and LB target group, which monitors target health.

CloudWatch Metrics (CPU)

The CloudWatch graph shows that CPU utilization spike above 70% threshold during the load test, which is the trigger point for scale-in. The CPU then drops to normal when the load test ends for scale-out.



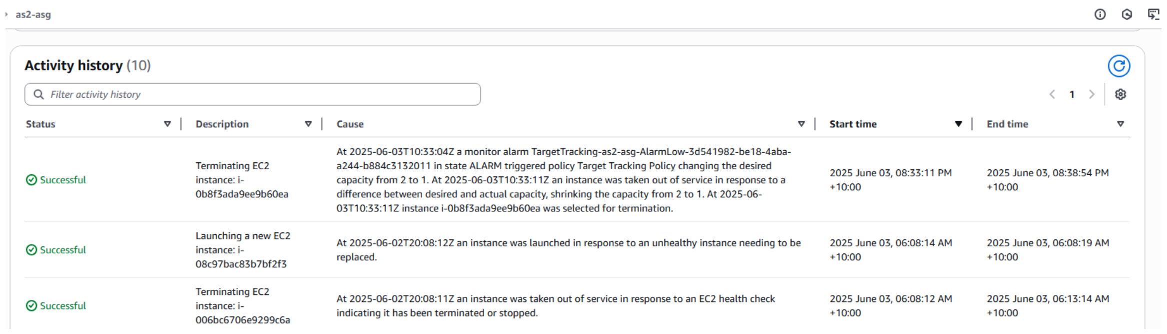
Statistic: Maximum Period: 30 seco...

Label	Details	Statistic	Period	Y axis	Actions
CPUUtilization	EC2 • CPUUtilization • AutoScalingGroup!	Maximum	30 seco...		

Figure 8 CPU utilization graph before, during and after load test

ASG Activity

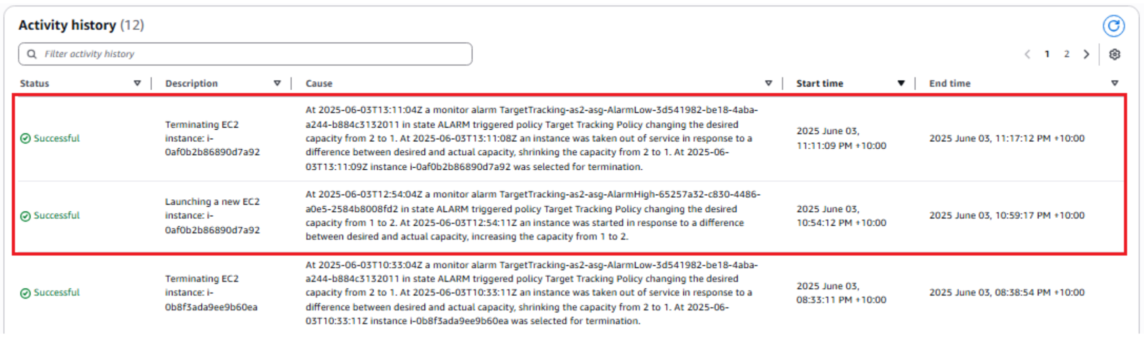
The ASG activity history shows the instance launch event when the CPU exceeds 70%, and the timestamp correlation with the CPU threshold breach. After the load test, it also shows the event of instance termination after cooldown period.


as2-asg

Activity history (10)

Status	Description	Cause	Start time	End time
✓ Successful	Terminating EC2 instance: i-0b8f3ada9ee9b60ea	At 2025-06-03T10:33:04Z a monitor alarm TargetTracking-as2-asg-AlarmLow-3d541982-be18-4aba-a244-b884c3132011 in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 2 to 1. At 2025-06-03T10:33:11Z 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 2025-06-03T10:33:11Z instance i-0b8f3ada9ee9b60ea was selected for termination.	2025 June 03, 08:33:11 PM +10:00	2025 June 03, 08:38:54 PM +10:00
✓ Successful	Launching a new EC2 instance: i-08c97bac83b7bf2f3	At 2025-06-02T20:08:12Z an instance was launched in response to an unhealthy instance needing to be replaced.	2025 June 03, 06:08:14 AM +10:00	2025 June 03, 06:08:19 AM +10:00
✓ Successful	Terminating EC2 instance: i-006bc6706e9299c6a	At 2025-06-02T20:08:11Z an instance was taken out of service in response to an EC2 health check indicating it has been terminated or stopped.	2025 June 03, 06:08:12 AM +10:00	2025 June 03, 06:13:14 AM +10:00

Figure 9 ASG activity log before load test

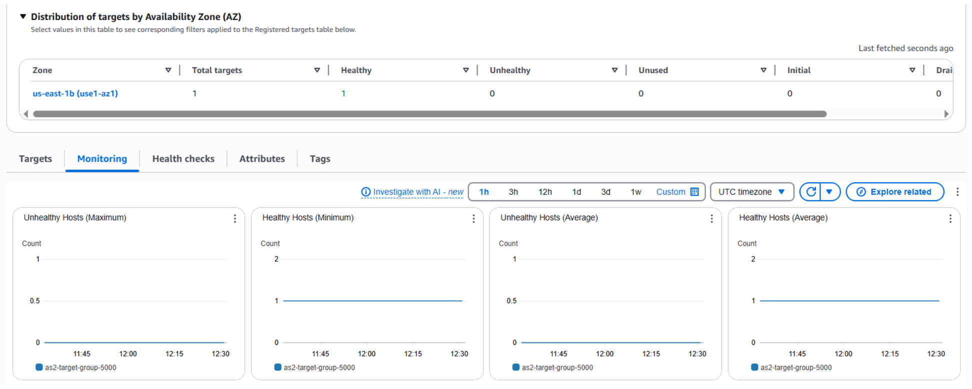

Activity history (12)

Status	Description	Cause	Start time	End time
✓ Successful	Terminating EC2 instance: i-0af0b2b86890d7a92	At 2025-06-03T13:11:04Z a monitor alarm TargetTracking-as2-asg-AlarmLow-3d541982-be18-4aba-a244-b884c3132011 in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 2 to 1. At 2025-06-03T13:11:08Z 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 2025-06-03T13:11:09Z instance i-0af0b2b86890d7a92 was selected for termination.	2025 June 03, 11:11:09 PM +10:00	2025 June 03, 11:17:12 PM +10:00
✓ Successful	Launching a new EC2 instance: i-0af0b2b86890d7a92	At 2025-06-03T12:54:04Z a monitor alarm TargetTracking-as2-asg-AlarmHigh-65257a32-c830-4486-a0e5-2584b8008fd2 in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 1 to 2. At 2025-06-03T12:54:11Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 1 to 2.	2025 June 03, 10:54:12 PM +10:00	2025 June 03, 10:59:17 PM +10:00
✓ Successful	Terminating EC2 instance: i-0b8f3ada9ee9b60ea	At 2025-06-03T10:33:04Z a monitor alarm TargetTracking-as2-asg-AlarmLow-3d541982-be18-4aba-a244-b884c3132011 in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 2 to 1. At 2025-06-03T10:33:11Z 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 2025-06-03T10:33:11Z instance i-0b8f3ada9ee9b60ea was selected for termination.	2025 June 03, 08:33:11 PM +10:00	2025 June 03, 08:38:54 PM +10:00

Figure 10 ASG activity log after load test (with 1 launch event and 1 termination event)

Load Balancer Target Health

The target group shows the health check status during scaling events. The first figure is the monitor log before the load test. The second figure shows the healthy instances count increase to 2, and the third figure shows an instance decrease (i.e., 1 draining instance) after the load test.

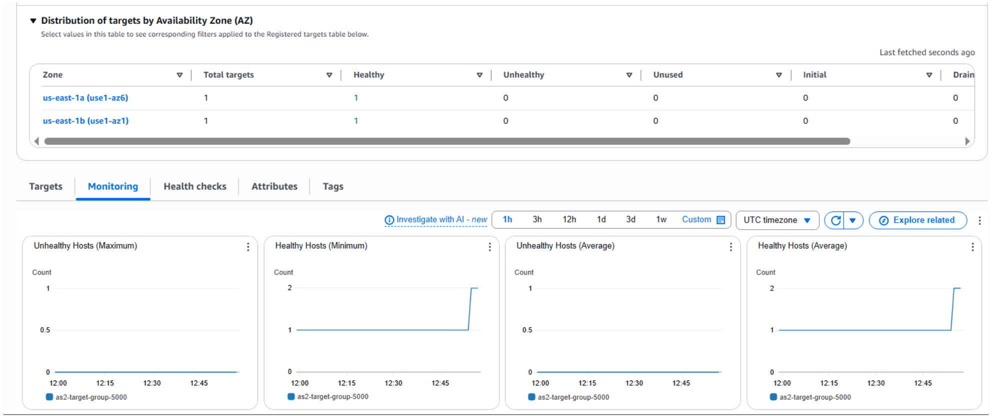


Distribution of targets by Availability Zone (AZ)
Select values in this table to see corresponding filters applied to the Registered targets table below.

Last fetched seconds ago

Zone	Total targets	Healthy	Unhealthy	Unused	Initial	Drai
us-east-1b (use1-az1)	1	1	0	0	0	0

Figure 11 Target health log before load test

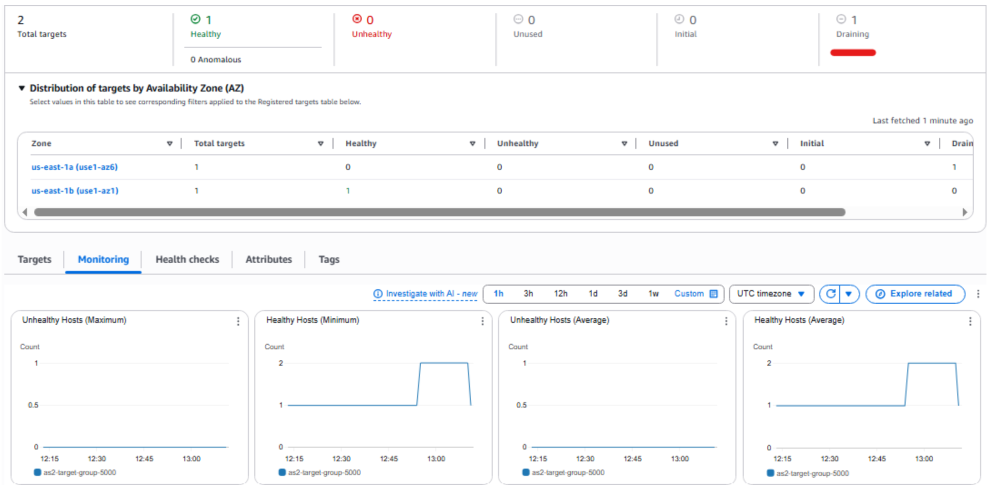


Distribution of targets by Availability Zone (AZ)
Select values in this table to see corresponding filters applied to the Registered targets table below.

Last fetched seconds ago

Zone	Total targets	Healthy	Unhealthy	Unused	Initial	Drain
us-east-1a (use1-az6)	1	1	0	0	0	0
us-east-1b (use1-az1)	1	1	0	0	0	0

Figure 12 Target health log during load test



2 Total targets	1 Healthy (0 Anomalous)	0 Unhealthy	0 Unused	0 Initial	1 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 1 minute ago

Zone	Total targets	Healthy	Unhealthy	Unused	Initial	Drain
us-east-1a (use1-az6)	1	0	0	0	0	1
us-east-1b (use1-az1)	1	1	0	0	0	0

Figure 13 Target health log after load test

Conclusion

In this project, I learnt how to successfully deploy a cloud-native web application leveraging AWS's serverless and auto-scaling capabilities. I learnt how to set up a three-tier security model with Application Load Balancer in public subnets, EC2 instances in private subnets, and RDS database in isolated private subnets, while integrating event-driven Lambda functions for automated image processing. I observed how the serverless component decoupled caption generation and thumbnail creation from the web application, demonstrating the effectiveness of event-driven architecture where S3 uploads automatically trigger Lambda functions to process images calling external API and store the processing results back to the shared RDS database.

The primary challenges I encountered was noticing that the given IAM role lack access to SNS policy, failing to invoke the lambda function. Additionally, there were some difficulties debugging the three-tier security architecture due to the coordination of security groups, subnets, and IAM roles. It is also important to only define proper S3 event that should trigger Lambda functions without creating recursive loops. The final architecture shows great improvements in scalability and fault tolerance, with successful load balancing across multiple instances in different AZs.

Appendix

A. DB creation code

```
#!/bin/bash

# RDS connection details
DB_HOST="image-app-db.cqi7uwt3hkvj.us-east-1.rds.amazonaws.com"
DB_USER="admin"                # Your RDS admin username
DB_PASSWORD="imagePassword!"       # Your RDS password
SQL_COMMANDS=$(cat <<EOF
/*
  Database Creation Script for the Image Captioning App
*/

DROP DATABASE IF EXISTS image_caption_db;
CREATE DATABASE image_caption_db;
USE image_caption_db;

/* Create captions table with thumbnail support */
CREATE TABLE captions (
    id INT AUTO_INCREMENT PRIMARY KEY,
    image_key VARCHAR(255) NOT NULL,
    caption TEXT,
    thumbnail_key VARCHAR(255),
    uploaded_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    caption_generated_at TIMESTAMP NULL,
    thumbnail_generated_at TIMESTAMP NULL
);

/* Create index for faster queries */
CREATE INDEX idx_uploaded_at ON captions(uploaded_at);
CREATE INDEX idx_image_key ON captions(image_key);
EOF
)

# Execute SQL commands
echo "Creating database and table..."
mysql -h $DB_HOST -u $DB_USER -p$DB_PASSWORD -e "$SQL_COMMANDS"

# Check if the previous command was successful
if [ $? -eq 0 ]; then
    echo "Database and table created successfully!"
else
    echo "Error: Failed to create database and table. Please check the connection details and try
again."
    exit 1
fi
```

B. caption_lambda.py and thumbnail_lambda.py

```
# Triggered by S3 ObjectCreated events to generate captions with Gemini
```

```python
import json
import boto3
import mysql.connector
import google.generativeai as genai
import base64
from urllib.parse import unquote_plus
import os

# Configure Gemini API, REPLACE with your Gemini API key
# GOOGLE_API_KEY = "AIzaSyBlEbQP4zbYOqcYL_-Sv_KhlBsfo2ZWbUE"
GOOGLE_API_KEY = os.environ['GOOGLE_API_KEY']  # set as environment variable
genai.configure(api_key=GOOGLE_API_KEY)

# Choose a Gemini model for generating captions
model = genai.GenerativeModel(model_name="gemini-2.5-flash-preview-04-17")

# db config from environment variables
DB_HOST = os.environ['DB_HOST']
DB_NAME = os.environ['DB_NAME']
DB_USER = os.environ['DB_USER']
DB_PASSWORD = os.environ['DB_PASSWORD']

# Initialize the S3 client outside of the handler
s3_client = boto3.client('s3')

def get_db_connection():
    # connect to RDS
    try:
        connection = mysql.connector.connect(
            host=DB_HOST,
            database=DB_NAME,
            user=DB_USER,
            password=DB_PASSWORD
        )
        return connection
    except mysql.connector.Error as e:
        print(f'Database connection error: {str(e)}')
        return None

def generate_caption(image_data):
    # generate caption using GEmini
    try:
        encoded_image = base64.b64encode(image_data).decode("utf-8")
        response = model.generate_content(
            [
                {"mime_type": "image/jpeg", "data": encoded_image},
                "Caption this image, in a paragraph, focusing on the main subjects, setting, and notable 
features.",
            ]
        )
        return response.text if response.text else "No caption generated."
    except Exception as e:
        print(f"Caption generation error: {str(e)}")
        return f"Caption generation error: {str(e)}"

def lambda_handler(event, context):
    # lambda handler for S3 ObjectCreated events
    # download img, generate caption, update RDS DB
    for record in event['Records']:
        # sns message
        sns_message = json.loads(record['Sns']['Message'])
        # s3 event from SNS
        for s3_record in sns_message['Records']:
            bucket = s3_record['s3']['bucket']['name']
            key = unquote_plus(s3_record['s3']['object']['key'])  # unquoting HTML form values

            if key.startswith('thumbnails'):
                print(f'skipping thumbnail: {key}')
                continue

            print(f'Processing image: {bucket}/{key}')

            try:
                # download img from S3
                response = s3_client.get_object(Bucket=bucket, Key=key)
                image_data = response['Body'].read()

                # generate caption
                caption = generate_caption(image_data)
                print(f'Caption: {caption}')

                # update DB with caption
                connection = get_db_connection()
                if connection:
                    cursor = connection.cursor()
                    cursor.execute(
                        """UPDATE captions
                        SET caption = %s, caption_generated_at = NOW()
                        WHERE image_key = %s""",
                        (caption, key)
                    )
                    connection.commit()
                    connection.close()
                    print(f'DB updated for {key}')

                else:
                    print(f'DB failed to connect')
            except Exception as e:
                print(f'Error processing {key}: {str(e)}')

    return {
        'statusCode': 200,
        'body': json.dumps('Caption generated')
    }
```

```
# Triggered by S3 ObjectCreated to generate thumbnail
import json
import boto3
import mysql.connector
from PIL import Image
from io import BytesIO
from urllib.parse import unquote_plus
import os

# db config from environment variables
DB_HOST = os.environ['DB_HOST']
DB_NAME = os.environ['DB_NAME']
DB_USER = os.environ['DB_USER']
DB_PASSWORD = os.environ['DB_PASSWORD']

s3_client = boto3.client('s3')

# thumbnail settings
THUMBNAIL_SIZE = (200, 200)
THUMBNAIL_QUALITY = 85

def get_db_connection():
    # connect to RDS
    try:
        connection = mysql.connector.connect(
            host=DB_HOST,
            database=DB_NAME,
            user=DB_USER,
            password=DB_PASSWORD
        )
        return connection
    except mysql.connector.Error as e:
        print(f'Database connection error: {str(e)}')
        return None

def generate_thumbnail(image_data):
    # generate thumbnail using GEmini
    try:
        image = Image.open(BytesIO(image_data))
        if image.mode in ('RGBA', 'LA', 'P'):
            image = image.convert('RGB')

        image.thumbnail(THUMBNAIL_SIZE, Image.Resampling.LANCZOS)

        thumbnail_buffer = BytesIO()
        image.save(thumbnail_buffer, format='JPEG', quality=THUMBNAIL_QUALITY, optimize=True)
        thumbnail_buffer.seek(0)

        return thumbnail_buffer.getvalue()

    except Exception as e:
        print(f"Thumbnail generation error: {str(e)}")
        return f"Thumbnail generation error: {str(e)}"

def lambda_handler(event, context):
    # lambda handler for S3 ObjectCreated events
    # download img, generate thumbnail, upload to S3, update RDS DB
    for record in event['Records']:
        # sns message
        sns_message = json.loads(record['Sns']['Message'])
        # s3 event fron SNS
        for s3_record in sns_message['Records']:
            bucket = s3_record['s3']['bucket']['name']
            key = unquote_plus(s3_record['s3']['object']['key'])  # unquoting HTML form values

            if key.startswith('thumbnails'):
                print(f'skipping thumbnail: {key}')
                continue

            print(f'Processing thumbnail: {bucket}/{key}')

            try:
                # download img from S3
                response = s3_client.get_object(Bucket=bucket, Key=key)
                image_data = response['Body'].read()

                # generate caption
                thumbnail_data = generate_thumbnail(image_data)

                if thumbnail_data:
                    thumbnail_key = f'thumbnails/thumb_{key}'

                    # upload to S3
                    s3_client.put_object(
                        Bucket=bucket,
                        Key=thumbnail_key,
                        Body=thumbnail_data,
                        ContentType='image/jpeg',
                        Metadata={
                            'original-key': key,
                            'thumbnail-size': f'{THUMBNAIL_SIZE[0]}x{THUMBNAIL_SIZE[1]}'
```

```
                                    }
                                )

                            print(f'Thumbnail uploaded: {thumbnail_key}')

                            # update DB with thumbnail key
                            connection = get_db_connection()
                            if connection:
                                cursor = connection.cursor()
                                cursor.execute(
                                    """UPDATE captions
                                    SET thumbnail_key = %s, thumbnail_generated_at = NOW()
                                    WHERE image_key = %s""",
                                    (thumbnail_key, key)
                                )
                                connection.commit()
                                connection.close()
                                print(f'DB updated with thumbnail for {key}')

                            else:
                                print(f'DB failed to connect')

                        except Exception as e:
                            print(f'Error processing thumbnail for {key}: {str(e)}')

        return {
            'statusCode': 200,
            'body': json.dumps('Thumbnail generated')
        }
```