Deployment Report on Image Annotation Application with AWS

qsun0008
500709005

Contents

1 Introduction

The overall purpose of this project is to construct a cloud-native solution that integrates a user-facing web application with a serverless backend for automated image processing tasks. Key architectural features include the use of scalable EC2 instances for the web frontend, robust storage solutions with S3 and RDS, and event-driven serverless functions using AWS Lambda for tasks such as AI-based image annotation and thumbnail generation. This approach aims to demonstrate effective cloud resource utilization, scalability, and resilience.

2 Architecture Diagrams

2.1 Web Application Architecture

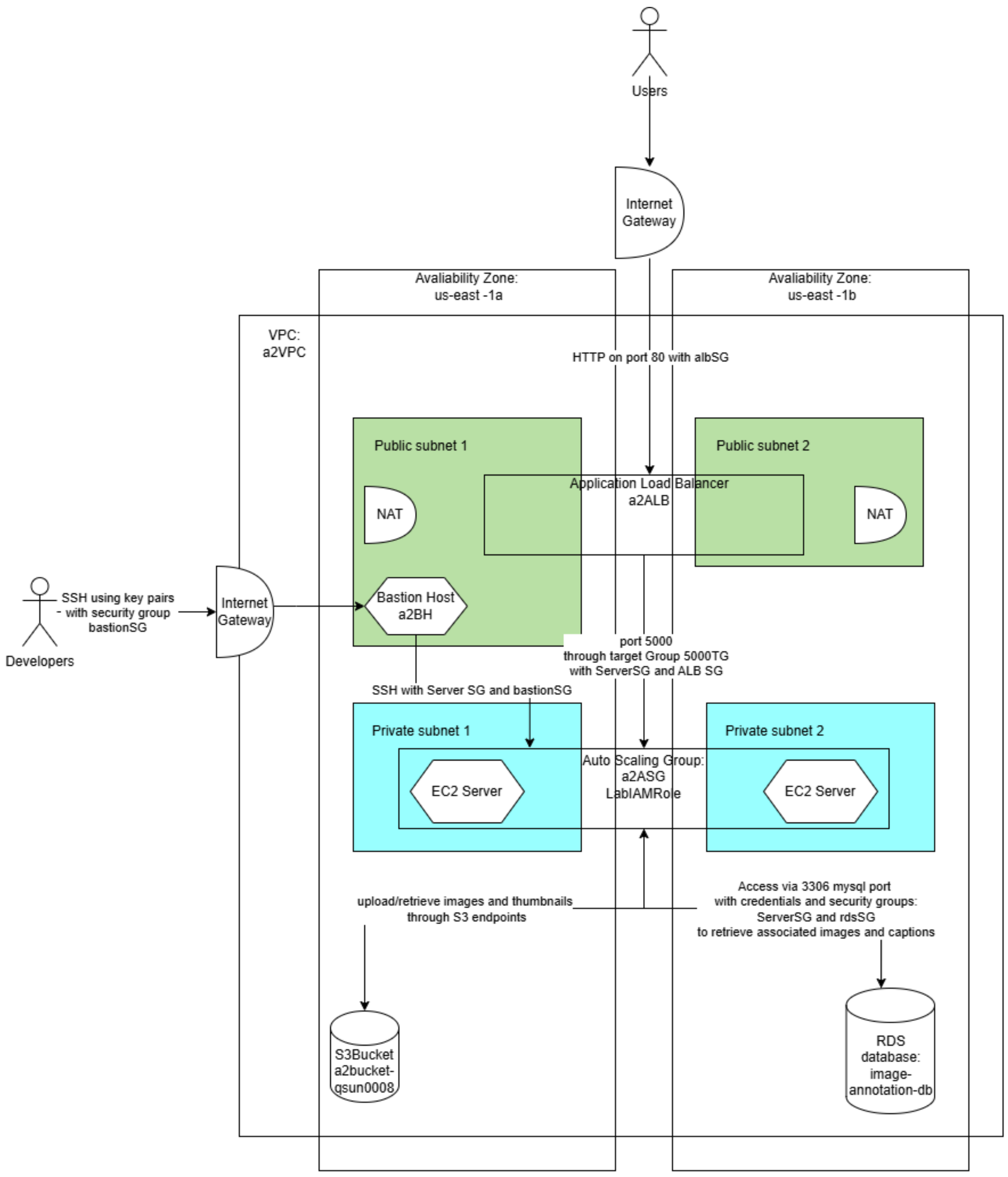

Users
Internet Gateway
Avaliability Zone: us-east -1a
Avaliability Zone: us-east -1b
VPC: a2VPC
HTTP on port 80 with albSG
Public subnet 1
Public subnet 2
Application Load Balancer a2ALB
NAT
NAT
SSH using key pairs - with security group bastionSG
Developers
Internet Gateway
Bastion Host a2BH
port 5000 through target Group 5000TG with ServerSG and ALB SG
SSH with Server SG and bastionSG
Private subnet 1
Private subnet 2
Auto Scaling Group: a2ASG LabIAMRole
EC2 Server
EC2 Server
upload/retrieve images and thumbnails through S3 endpoints
Access via 3306 mysql port with credentials and security groups: ServerSG and rdsSG to retrieve associated images and captions
S3Bucket a2bucket-qsun0008
RDS database: image-annotation-db

Figure 1: Web Application Architecture Diagram showcasing EC2 ASG, ALB, S3, RDS, and networking layers.

The web application, illustrated in Figure 1, is architected for high availability and security within a multi-AZ VPC (a2VPC). User traffic enters via an Internet Gateway to an Application Load Balancer (a2ALB) situated in public subnets; this multi-AZ deployment of the ALB ensures fault tolerance and efficient distribution of requests to EC2 instances on port 5000. These instances, running the core application, are managed by an Auto Scaling Group (a2ASG) and reside in private subnets across two Availability Zones. This strategy enhances resilience by isolating them from direct internet exposure and allowing the ASG to dynamically scale resources based on demand while also replacing unhealthy instances. Secure, private communication with an S3 bucket (a2bucket-gsun0008) for image storage is achieved using S3 endpoints, while metadata is managed in an RDS MySQL database (image-annotation-db) also housed in private subnets to protect sensitive data. Administrative access to these secure EC2 instances is strictly controlled through a Bastion Host (a2BH) in a public subnet, minimizing the attack surface. NAT Gateways facilitate necessary outbound internet connectivity for instances in private subnets (e.g., for software updates) without compromising their inbound security. A multi-layered security approach is enforced using specific security groups (ServerSG, rdsSG, bastionSG, ALB SG) to apply the principle of least privilege at each communication interface.

2.2 Serverless Architecture

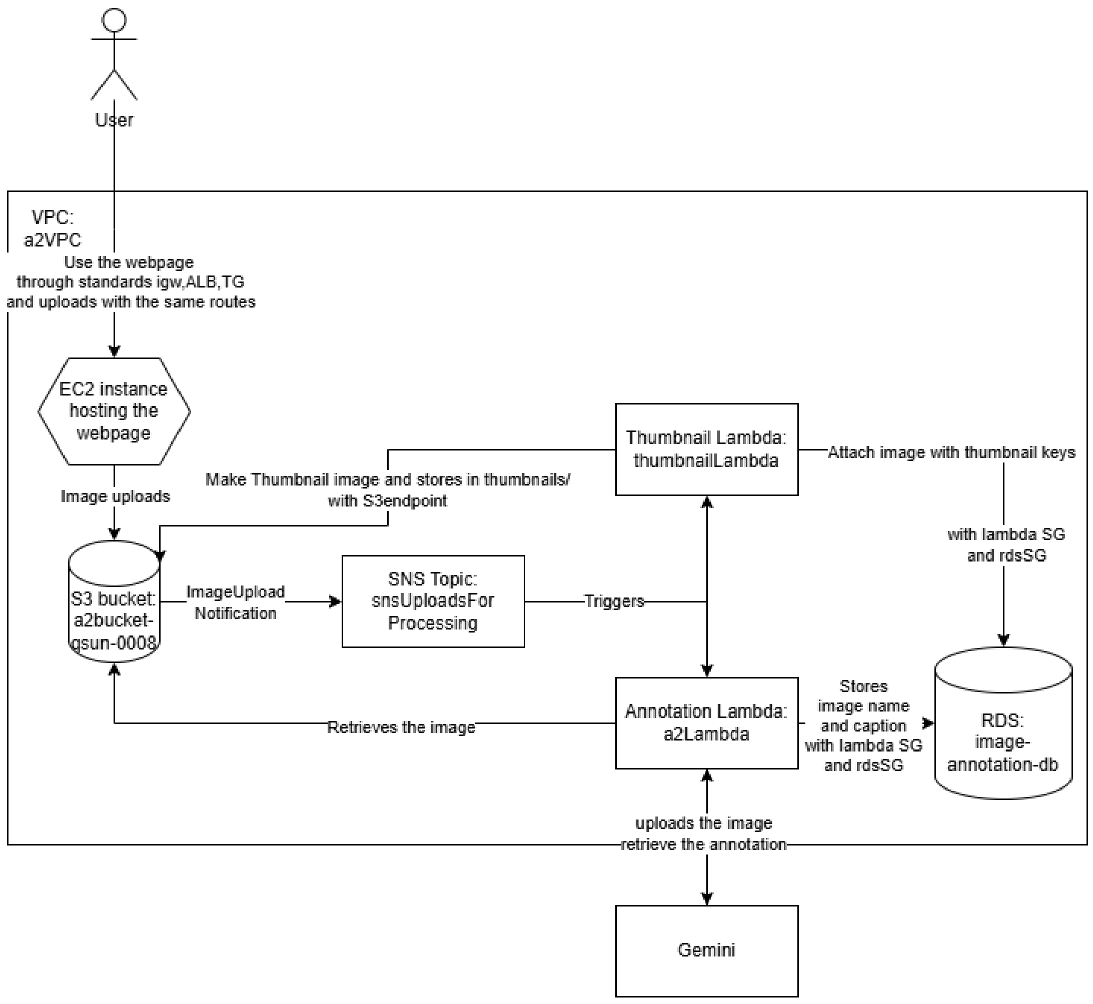

User
VPC: a2VPC
Use the webpage through standards igw,ALB,TG and uploads with the same routes
EC2 instance hosting the webpage
Image uploads
Make Thumbnail image and stores in thumbnails/ with S3endpoint
Thumbnail Lambda: thumbnailLambda
Attach image with thumbnail keys
with lambda SG and rdsSG
S3 bucket: a2bucket-gsun-0008
ImageUpload Notification
SNS Topic: snsUploadsForProcessing
Triggers
Retrieves the image
Annotation Lambda: a2Lambda
Stores image name and caption with lambda SG and rdsSG
RDS: image-annotation-db
uploads the image retrieve the annotation
Gemini

Figure 2: Serverless Architecture Diagram depicting Lambda functions, S3 event source via SNS, Gemini API interaction, and RDS/S3 targets.

The serverless processing pipeline, detailed in Figure 2, employs an event-driven architecture to decouple backend tasks from the frontend application, enhancing scalability and maintainability. This pipeline is initiated when the web application uploads an image to the shared S3 bucket (a2bucket-gsun0008). This S3 object creation event triggers a notification to an SNS topic (”snsUploadsForProcessing”), which serves as a robust fan-out mechanism, enabling parallel processing by multiple subscribers. Two distinct AWS Lambda functions are subscribed: the Thumbnail Lambda (thumbnailLambda) and the Annotation Lambda (a2Lambda). This separation allows each function to be independently scaled, versioned, and resourced according to its specific needs. The Thumbnail Lambda generates an image thumbnail, stores it back into a designated prefix in the S3 bucket, and updates the shared RDS database (image-annotation-db) with the thumbnail's location. Concurrently, the Annotation Lambda retrieves the original image from S3, interacts with the external Gemini API for content analysis (requiring secure API key management and internet access via a NAT gateway as the Lambda is VPC-enabled), and then stores the derived annotation or caption in the shared RDS database. The necessity for both Lambda functions to interact with the RDS database, which is in private VPC subnets, mandates their configuration within the VPC, utilizing security groups (lambda SG and rdsSG) to ensure secure and controlled database access. This design ensures that intensive processing tasks do not impact frontend performance and can scale independently.

2.3 Integration Between Components

The integration points between the web application and serverless architectures are clearly defined. The web application triggers the Lambda functions indirectly by uploading images to an S3 bucket prefix (‘uploads/‘). This S3 event is configured to publish a notification to an SNS topic, which in turn invokes both the Annotation and Thumbnail Lambda functions. Both architectural components access shared resources: the S3 bucket for image storage and retrieval (web app writes to ‘uploads/‘, Annotation Lambda reads from ‘uploads/‘, Thumbnail Lambda reads from ‘uploads/‘ and writes to ‘thumbnails/‘, web app reads from both for the gallery), and the RDS database for metadata (web app writes initial metadata and reads for gallery, Lambdas update records with captions and thumbnail keys).

3 Web Application Deployment

The web application provides users with an interface to upload images and view a gallery of processed images. It is architected and deployed on AWS to prioritize scalability, high availability, resilience, and security, leveraging a suite of managed services to reduce operational overhead.

3.1 Compute Environment (EC2 Auto Scaling Group)

The core of the application's backend logic runs on Amazon EC2 instances. Specifically, `t2.micro` instances are utilized, offering a balance between computational capability and cost-effectiveness, suitable for an application with potentially variable but generally modest baseline resource requirements. The Amazon Linux 2023 AMI is selected as the operating system due to its optimization for the EC2 environment, long-term support, and regular security updates, providing a stable and secure foundation.

These instances are managed by an Auto Scaling Group (ASG), which is crucial for both scalability and resilience. The ASG dynamically adjusts the number of active EC2 instances based on demand, ensuring that performance remains consistent as user traffic fluctuates. It also contributes to high availability by automatically replacing any instances that fail health checks, thus maintaining the desired application capacity. A Launch Template defines the configuration for new instances, ensuring consistency. This template specifies the AMI ID, instance type, the IAM role granting necessary permissions, the security group for network traffic filtering, and a User Data script for instance bootstrapping.

The ASG is configured with a minimum of 1 instance to ensure the application is always available, a desired capacity of 1 for normal operation, and a maximum of 3 instances. This upper limit prevents uncontrolled scaling and associated costs while allowing the application to handle moderate surges in traffic. Scaling is governed by a target tracking scaling policy, which aims to maintain an average CPU utilization of 70%. This metric provides a good balance, allowing instances to absorb minor spikes without triggering scaling events too frequently, yet ensuring resources are added before performance degrades significantly.

Upon launch, each EC2 instance executes a `user_data.sh` script. This bootstrapping script automates the instance setup process, which is critical for rapid scaling and recovery. The script performs the following actions:

- Installs essential software packages: Python 3 (the application's runtime), pip (Python package installer), unzip (to extract the application code), and Gunicorn (a robust WSGI HTTP server for Python web applications).
- Downloads the Flask application code, packaged as `app.zip`, from a designated S3 bucket. Storing the application package in S3 allows for versioning and centralized management of the deployment artifact.
- Unzips the application code into the `/var/www/app/` directory.
- Installs Python dependencies listed in a `requirements.txt` file (included in the zip) using pip.
- Sets up and starts a systemd service named `image_app.service`. This ensures the Flask application, served by Gunicorn, runs reliably as a background process, is managed by the operating system, and restarts automatically if it crashes or the instance reboots. Gunicorn listens on port 5000 for incoming application traffic.

Network settings are designed for security and high availability. The entire deployment resides within a custom Virtual Private Cloud (VPC) with a CIDR block of `10.0.0.0/16`, providing a logically isolated section of the AWS cloud. EC2 instances are launched into private subnets distributed across at least two Availability Zones (AZs). This multi-AZ deployment strategy is fundamental for high availability, as it ensures the application can withstand the failure of a single AZ. Private subnets, by definition, do not have direct routes to the internet, enhancing security. Internet-bound traffic from these instances (e.g., for downloading software updates or accessing external APIs if necessary, though not explicitly stated for the EC2 instances themselves for this application's core function) is routed through a NAT Gateway residing in a public subnet. The NAT Gateway allows outbound internet connectivity while preventing unsolicited inbound connections. Furthermore, a VPC endpoint for S3 is utilized. This allows the EC2 instances to access the S3 bucket (e.g., to download the `app.zip` or for the application to interact with S3 if needed) via the AWS private network, which is more secure and can be faster than traversing the public internet.

Security configurations are multi-layered. An EC2 Security Group acts as a virtual firewall for the instances. It is configured with the principle of least privilege:

- Inbound TCP traffic on port 5000 (the Gunicorn port) is allowed only from the Application Load Balancer's security group. This ensures that application traffic must pass through the ALB.
- Inbound TCP traffic on port 22 (SSH) is allowed only from the Bastion Host's security group, restricting administrative access.
- Outbound HTTPS traffic on port 443 is permitted, enabling secure access to AWS services like S3 (for the user data script) and AWS Secrets Manager.
- Outbound TCP traffic on port 3306 is allowed specifically to the RDS database's security group, enabling the application to connect to the database.

An IAM Role is attached to the EC2 instances via the Launch Template. This role grants the application the necessary permissions to interact with other AWS services without needing to embed credentials directly into the application code. Permissions include access to specific S3 buckets (e.g., for application deployment artifacts), AWS Secrets Manager (to retrieve database credentials), and CloudWatch Logs (for logging application and system events, crucial for monitoring and troubleshooting). Database credentials, API keys, and other sensitive information are securely stored in AWS Secrets Manager, which provides robust encryption and access control.

Administrative access to the EC2 instances for troubleshooting or maintenance is facilitated by a Bastion Host (or jump host). This is a separate EC2 instance located in a public subnet. Its security group is tightly controlled, allowing SSH access (port 22) only from a specific, trusted IP address range (e.g., the administrator's office network). Administrators first connect to the Bastion Host and then, from the Bastion Host, can SSH into the application instances in the private subnets. This approach significantly reduces the attack surface of the application instances.

3.2 Load Balancer Setup (Application Load Balancer)

An internet-facing Application Load Balancer (ALB) serves as the single point of entry for all incoming HTTP traffic to the web application. The ALB is chosen because it operates at the application layer (Layer 7) and offers features like content-based routing, SSL/TLS termination (though not explicitly configured here with an HTTP listener, it's a key capability), and integration with AWS WAF, making it ideal for web applications.

The ALB is configured with an HTTP listener on port 80. This listener receives incoming requests from users and forwards them to a target group. This target group is configured to route traffic to the EC2 instances within the Auto Scaling Group, specifically to port 5000 using the HTTP protocol, which matches the port Gunicorn is configured to listen on within each instance.

Health checks are a critical component of the ALB setup. The ALB periodically sends requests to a specified path on each registered instance (in this case, the '/' path on port 5000) to verify its health. If an instance fails these health checks, the ALB stops sending traffic to it and the ASG will eventually terminate and replace the unhealthy instance. This ensures that user traffic is only directed to healthy, responsive instances, thereby contributing significantly to the application's overall availability and reliability.

To ensure high availability for the load balancing tier itself, the ALB is deployed across the public subnets in at least two Availability Zones. This mirrors the multi-AZ deployment of the EC2 instances. If one AZ becomes unavailable, the ALB instances in the other AZ(s) can continue to handle traffic. The ALB's security group is configured to allow inbound HTTP traffic on port 80 from anywhere on the internet (`0.0.0.0/0`) and outbound traffic on port 5000 to the EC2 instances' security group, restricting its communication path.

3.3 Database Environment (RDS MySQL)

The application relies on a relational database to store image metadata, such as annotations and image paths. An Amazon RDS for MySQL instance, specifically a `db.t3.micro` instance class, is provisioned for this purpose. The `db.t3.micro` offers a cost-effective solution for the expected database load, while RDS itself provides significant operational benefits. Using RDS abstracts away much of the database management overhead, including patching, backups, software updates, and provides options for easy scaling and high availability configurations (e.g., Multi-AZ deployments, read replicas) if needed in the future.

The database is named `image_annotations_db`. Critically, it is provisioned within the private subnets of the VPC, ensuring it is not directly accessible from the public internet. This is a fundamental security best practice. Access to the database is strictly controlled by an RDS Security Group. This security group is configured to allow inbound TCP traffic on port 3306 (the default MySQL port) only from the security groups associated with the EC2 application instances and the Lambda functions. This precise rule minimizes the exposure of the database.

The Flask application, running on the EC2 instances, connects to this RDS instance using credentials (username and password). These credentials are not hardcoded into the application code. Instead, they are securely retrieved at runtime from AWS Secrets Manager. This practice enhances security by centralizing secret management, enabling easier credential rotation, and providing fine-grained access control to the secrets themselves.

3.4 Storage Environment (S3 Bucket)

Amazon S3 (Simple Storage Service) is used as the primary storage solution for image files due to its high durability, scalability, availability, and cost-effectiveness for object storage. A single S3 bucket is configured to serve multiple purposes within the application architecture, organized using prefixes for clarity and access control:

- `uploads/`: Original images uploaded by users via the Flask web application are stored under this prefix.
- `thumbnails/`: Smaller, resized versions (thumbnails) of the uploaded images, generated by a Lambda function, are stored under this prefix. These are likely used in the gallery view for faster loading.

This S3 bucket is also integrated into the event-driven processing pipeline. Object creation events specifically within the `uploads/` prefix are configured to trigger notifications. Instead of directly triggering Lambda functions, these S3 event notifications are first sent to an Amazon SNS (Simple Notification Service) topic. This use of SNS decouples S3 from the Lambda functions, allowing for greater flexibility, such as having multiple subscribers (different Lambda functions or other services) react to the same event independently, or for queuing and retrying if a downstream service is temporarily unavailable.

The IAM role attached to the EC2 instances (and similarly, roles for Lambda functions) grants the necessary S3 permissions (e.g., `s3:PutObject` for uploads, `s3:GetObject` for retrieval). To ensure data security, Block Public Access settings are enabled at the bucket level, preventing accidental public exposure of the stored images. Access to images for display in the web application (e.g., in the gallery) is handled by generating presigned URLs. Presigned URLs provide temporary, time-limited access to specific S3 objects, allowing users to view images without making the S3 objects themselves public.

4 Serverless Component Deployment

The serverless backend components consist of two AWS Lambda functions. These functions are responsible for post-upload processing: annotating images using an external API and generating thumbnails. Their deployment leverages an event-driven architecture, primarily triggered by image uploads to S3.

4.1 Event-Driven Architecture

The system employs an event-driven architecture, initiated by user actions. When a new image is successfully uploaded to the `uploads/` prefix in the S3 bucket, S3 automatically publishes an event notification. This notification is sent to a pre-configured SNS topic. Using SNS as an intermediary offers several advantages:

- **Decoupling**: S3 does not need to know about its consumers. It simply publishes an event to SNS.
- **Fan-out**: Multiple services can subscribe to the same SNS topic and react to the event concurrently. In this case, both the Annotation Lambda and the Thumbnail Lambda are subscribed to this SNS topic.

- **Resilience**: SNS provides durable message delivery and retry mechanisms if subscribers are temporarily unavailable.
- **Flexibility**: It's easy to add new subscribers (e.g., another Lambda for different processing, or a queuing system) in the future without modifying the S3 event configuration.

Upon receiving a message from SNS, both Lambda functions are invoked. The SNS message wraps the original S3 event payload. The Lambda functions are programmed to parse this incoming SNS message to extract the details of the S3 event, particularly the bucket name and the object key (the full path to the newly uploaded image). This information allows them to locate and process the correct image. This parallel invocation of Lambda functions for annotation and thumbnail generation allows these tasks to be performed independently and efficiently.

4.2 Annotation Function

The Annotation Lambda function is responsible for analyzing the uploaded image and generating descriptive annotations, presumably by interacting with an external machine learning API .

Packaging and Deployment: The function is written in Python 3.10, a widely supported and mature runtime for Lambda. It is packaged as a ZIP file. This archive includes the handler script , which contains the core logic, and any necessary dependencies. Key dependencies include mysql-connector-python for interacting with the RDS database (to store annotations) and the google-generativeai SDK (along with its own dependencies like `grpcio`) for making calls to the Gemini API. To ensure compatibility with the Lambda execution environment, this ZIP package was built in an Amazon Linux 2 environment. This precaution mitigates issues related to compiled dependencies that might arise if built on a different operating system. The function is deployed via the AWS Lambda Management Console, though for more robust and repeatable deployments, Infrastructure as Code tools like AWS CloudFormation or AWS SAM would be recommended in a production setting.

Configuration and Permissions: The Lambda function's IAM execution role is configured with the principle of least privilege:

- `s3:GetObject` permission is granted for the specific S3 bucket to allow the function to download the uploaded image for analysis.
- Permissions to write logs to CloudWatch Logs are automatically included for monitoring and debugging.

To access the RDS database (which resides in private subnets), the Lambda function is configured to run within the same VPC as the RDS instance. It is associated with the private subnets and a specific security group. This Lambda security group allows outbound TCP traffic on port 3306 to the RDS instance's security group.

For external API access (Gemini API, which is on the internet), the Lambda function, while running in private subnets, requires a route to the internet. This is achieved by configuring the VPC with a NAT Gateway, and ensuring the Lambda's subnets have a route to this NAT Gateway for internet-bound traffic (0.0.0.0/0). This allows the Lambda to make outbound HTTPS calls to the Gemini API. The API key for the Gemini API is managed securely, ideally via AWS Secrets Manager. If using Secrets Manager, the Lambda's IAM role would also need `secretsmanager:GetSecretValue` permission for the specific secret. Alternatively, it could be passed as an encrypted environment variable, though Secrets Manager offers better management and rotation capabilities.

Environment variables are configured for the Lambda function to provide runtime configuration, such as database connection details (host, username, potentially the secret ARN if using Secrets Manager for the password) and the Gemini API key or its ARN in Secrets Manager. The function is configured with a timeout of 2 minutes and 30 seconds (150 seconds) and a memory allocation of 512 MB. These settings are chosen based on the expected duration of the Gemini API call, image processing, and database interaction, as well as the memory footprint of the Python runtime and its libraries. The timeout needs to be sufficient to handle potential retries or longer API response times, while memory should be adequate to prevent out-of-memory errors.

4.3 Thumbnail Generator Function

The Thumbnail Generator Lambda function is responsible for creating a smaller, optimized version (thumbnail) of the uploaded image.

Packaging and Deployment: Similar to the Annotation Lambda, this function is written in Python 3.10 and packaged as a ZIP file. The archive contains its handler script (`thumbnail_lambda.py`) and its dependencies. The primary dependency for image manipulation is the Pillow library (`Pillow`). The `mysql-connector-python` library is also included if the function needs to update the database with thumbnail information (e.g., path to the thumbnail), though the primary interaction described is with S3. As with the Annotation Lambda, this package was built in an Amazon Linux 2 environment for runtime compatibility and deployed via the AWS Lambda Management Console.

Configuration and Permissions: The IAM execution role for this Lambda function includes:

- `s3:GetObject` permission for the `uploads/` prefix in the source S3 bucket to retrieve the original image.
- `s3:PutObject` permission for the `thumbnails/` prefix in the target S3 bucket (which is the same bucket but could be different) to store the generated thumbnail. It's crucial that this also includes `s3:PutObjectAcl` if objects need specific ACLs, though typically relying on bucket policies and IAM roles is preferred.
- Standard CloudWatch Logs permissions.

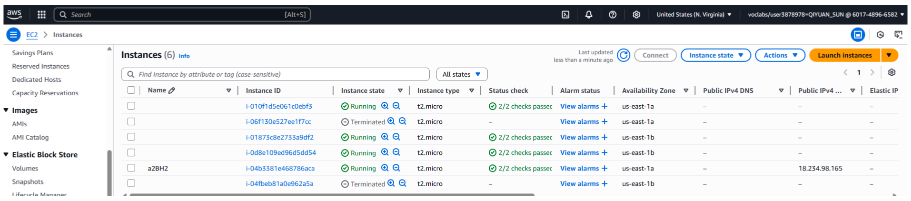
If database interaction is required (e.g., to update a record with the thumbnail S3 key), this Lambda function would also share similar VPC, subnet, security group (allowing outbound TCP on 3306 to RDS), and IAM configurations (for Secrets Manager if used) as the Annotation Lambda.

Environment variables are set for this function, including database connection details (if applicable) and crucially, `TARGET_S3_BUCKET` and potentially a `TARGET_S3_PREFIX` for thumbnails. The function is configured with a timeout of 2 minutes and 30 seconds (150 seconds) and a memory allocation of 128 MB. The memory requirement for thumbnail generation is typically lower than for functions involving ML model inference or large data processing, as image resizing is generally CPU and memory-efficient for common image sizes with libraries like Pillow. The timeout is set generously to accommodate variations in image size and processing time.

By using separate Lambda functions for annotation and thumbnail generation, triggered by the same S3 event via SNS, the system achieves parallel processing, improving overall throughput and responsiveness. Each function can be scaled and configured independently based on its specific resource needs and execution characteristics.

5 Auto Scaling Test Observation

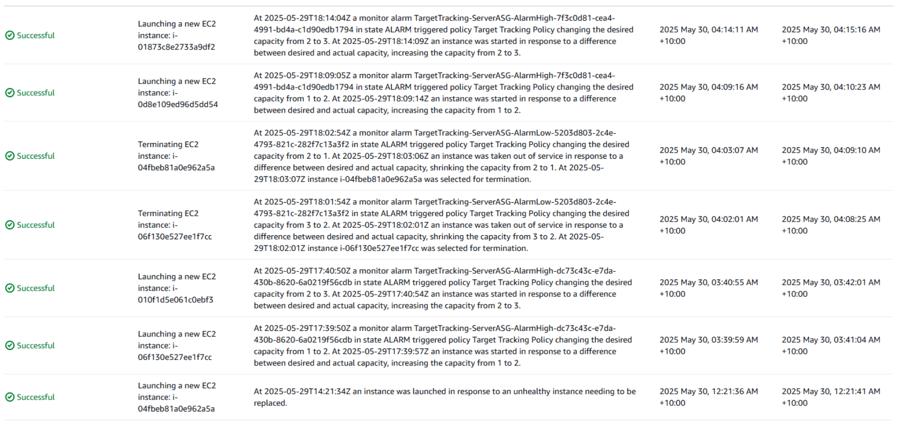
Below is screenshots for Evidence of AutoScaling:



Instances (6) Info

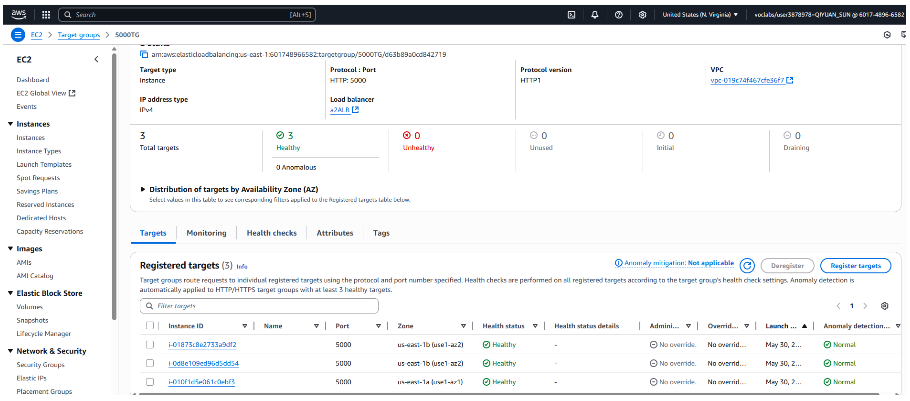
Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv4 ...	Elastic IP
	i-010f1d5e061c0ebf3	Running	t2.micro	2/2 checks passec	View alarms +	us-east-1a	–	–	–
	i-06f130e527ee1f7cc	Terminated	t2.micro	–	View alarms +	us-east-1a	–	–	–
	i-01873c8e2733a9df2	Running	t2.micro	2/2 checks passec	View alarms +	us-east-1b	–	–	–
	i-0d8e109ed96d5dd54	Running	t2.micro	2/2 checks passec	View alarms +	us-east-1b	–	–	–
a2BH2	i-04b3381e468786aca	Running	t2.micro	2/2 checks passec	View alarms +	us-east-1a	–	18.234.98.165	–
	i-04fbeb81a0e962a5a	Terminated	t2.micro	–	View alarms +	us-east-1b	–	–	–

Figure 3: EC2 instances snapshot reflecting scaling activity.



Successful	Launching a new EC2 instance: i-01873c8e2733a9df2	At 2025-05-29T18:14:04Z a monitor alarm TargetTracking-ServerASG-AlarmHigh-7f3c0d81-cea4-4991-bd4a-c1d90edb1794 in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 2 to 3. At 2025-05-29T18:14:09Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 2 to 3.	2025 May 30, 04:14:11 AM +10:00	2025 May 30, 04:15:16 AM +10:00
Successful	Launching a new EC2 instance: i-0d8e109ed96d5dd54	At 2025-05-29T18:09:05Z a monitor alarm TargetTracking-ServerASG-AlarmHigh-7f3c0d81-cea4-4991-bd4a-c1d90edb1794 in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 1 to 2. At 2025-05-29T18:09:14Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 1 to 2.	2025 May 30, 04:09:16 AM +10:00	2025 May 30, 04:10:23 AM +10:00
Successful	Terminating EC2 instance: i-04fbeb81a0e962a5a	At 2025-05-29T18:02:54Z a monitor alarm TargetTracking-ServerASG-AlarmLow-5203d803-2c4e-4793-821c-282f7c13a3f2 in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 2 to 1. At 2025-05-29T18:03:06Z 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 2025-05-29T18:03:07Z instance i-04fbeb81a0e962a5a was selected for termination.	2025 May 30, 04:03:07 AM +10:00	2025 May 30, 04:09:10 AM +10:00
Successful	Terminating EC2 instance: i-06f130e527ee1f7cc	At 2025-05-29T18:01:54Z a monitor alarm TargetTracking-ServerASG-AlarmLow-5203d803-2c4e-4793-821c-282f7c13a3f2 in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 3 to 2. At 2025-05-29T18:02:01Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 3 to 2. At 2025-05-29T18:02:01Z instance i-06f130e527ee1f7cc was selected for termination.	2025 May 30, 04:02:01 AM +10:00	2025 May 30, 04:08:25 AM +10:00
Successful	Launching a new EC2 instance: i-010f1d5e061c0ebf3	At 2025-05-29T17:40:50Z a monitor alarm TargetTracking-ServerASG-AlarmHigh-dc73c43c-e7da-430b-8620-6a0219f56cdb in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 2 to 3. At 2025-05-29T17:40:54Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 2 to 3.	2025 May 30, 03:40:55 AM +10:00	2025 May 30, 03:42:01 AM +10:00
Successful	Launching a new EC2 instance: i-06f130e527ee1f7cc	At 2025-05-29T17:39:50Z a monitor alarm TargetTracking-ServerASG-AlarmHigh-dc73c43c-e7da-430b-8620-6a0219f56cdb in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 1 to 2. At 2025-05-29T17:39:57Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 1 to 2.	2025 May 30, 03:39:59 AM +10:00	2025 May 30, 03:41:04 AM +10:00
Successful	Launching a new EC2 instance: i-04fbeb81a0e962a5a	At 2025-05-29T14:21:34Z an instance was launched in response to an unhealthy instance needing to be replaced.	2025 May 30, 12:21:36 AM +10:00	2025 May 30, 12:21:41 AM +10:00

Figure 4: ASG activity history showing scale-out and scale-in events.



arn:aws:elasticloadbalancing:us-east-1:601748966582:targetgroup/5000TG/d63b89a0cd842719

Target type: Instance
Protocol : Port: HTTP: 5000
Protocol version: HTTP1
VPC: vpc-019c74f467cfe36f7
IP address type: IPv4
Load balancer: a2ALB

3 Total targets | 3 Healthy | 0 Anomalous | 0 Unhealthy | 0 Unused | 0 Initial | 0 Draining

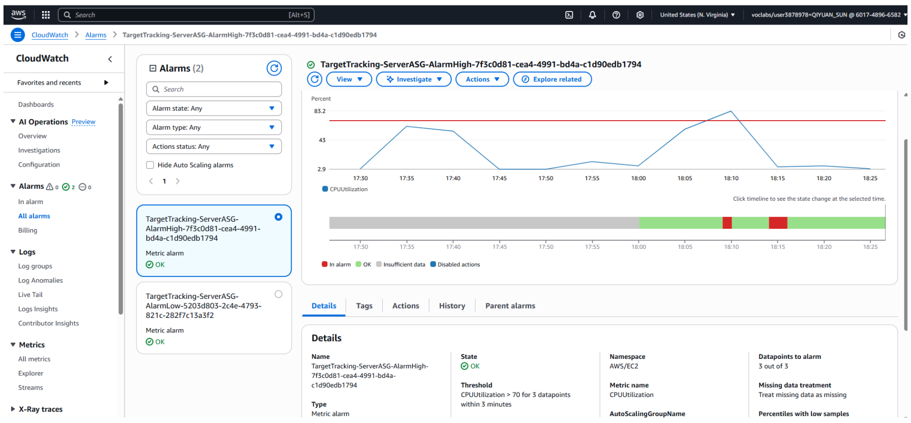
Distribution of targets by Availability Zone (AZ)
Select values in this table to see corresponding filters applied to the Registered targets table below.

Registered targets (3) Info

Target groups route requests to individual registered targets using the protocol and port number specified. Health checks are performed on all registered targets according to the target group's health check settings. Anomaly detection is automatically applied to HTTP/HTTPS target groups with at least 3 healthy targets.

Instance ID	Name	Port	Zone	Health status	Health status details	Admini...	Overrid...	Launch ...	Anomaly detection...
i-01873c8e2733a9df2		5000	us-east-1b (use1-az2)	Healthy	-	No override.	No overrid...	May 30, 2...	Normal
i-0d8e109ed96d5dd54		5000	us-east-1b (use1-az2)	Healthy	-	No override.	No overrid...	May 30, 2...	Normal
i-010f1d5e061c0ebf3		5000	us-east-1a (use1-az1)	Healthy	-	No override.	No overrid...	May 30, 2...	Normal

Figure 5: ALB Target Group showing three healthy registered targets.

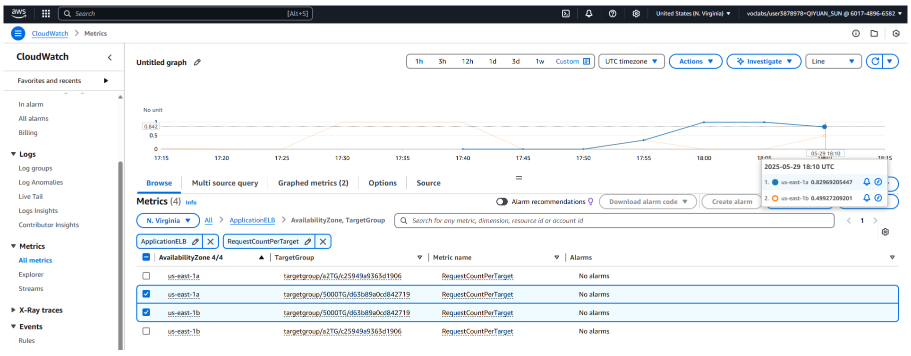


TargetTracking-ServerASG-AlarmHigh-7f3c0d81-cea4-4991-bd4a-c1d90edb1794

Details

Name	State	Namespace	Datapoints to alarm
TargetTracking-ServerASG-AlarmHigh-7f3c0d81-cea4-4991-bd4a-c1d90edb1794	OK	AWS/EC2	3 out of 3
Type	**Threshold**	**Metric name**	**Missing data treatment**
Metric alarm	CPUUtilization > 70 for 3 datapoints within 3 minutes	CPUUtilization	Treat missing data as missing

Figure 6: CloudWatch Alarm for high CPU utilization triggering scale-out.



2025-05-29 18:10 UTC
1. us-east-1a 0.82969205447
2. us-east-1b 0.49927209201

Metrics (4)

	AvailabilityZone 4/4	TargetGroup	Metric name	Alarms
☐	us-east-1a	targetgroup/a2TG/c25949a9363d1906	RequestCountPerTarget	No alarms
☑	us-east-1a	targetgroup/5000TG/d63b89a0cd842719	RequestCountPerTarget	No alarms
☑	us-east-1b	targetgroup/5000TG/d63b89a0cd842719	RequestCountPerTarget	No alarms
☐	us-east-1b	targetgroup/a2TG/c25949a9363d1906	RequestCountPerTarget	No alarms

Figure 7: CloudWatch metrics demonstrating load distribution across targets.

To validate the Auto Scaling and Load Balancing, a load test was conducted on the '/gallery' page of the web application using ApacheBench with parameters '-n 500000 -c 1500'. Two of such load test where run on 17:45 UTC and peaking around 18:05-18:10 UTC respectively to monitor scaling out and scaling in.

EC2 Instances Scaling Out: Evidence of EC2 instances scaling out in response to increased load was clearly observed. The Auto Scaling Group was initially operating with one active instance, as indicated by the ASG activity history prior to the scale-out event (see Figure 4, showing capacity increasing from 1 to 2). During the load test, the average CPU utilization for the ASG surpassed the configured threshold of 70%. This is demonstrated in Figure 6, where the 'CPUUtilization' metric for the 'TargetTracking-ServerASG-AlarmHigh-...' alarm is shown exceeding the threshold and triggering an "In alarm" state around 18:08-18:10 UTC on May 29, 2025.

Consequently, the ASG scaling policy initiated the launch of additional instances. The ASG activity history (Figure 4) records these scale-out events, showing instance 'i-0d8e109ed96d5dd54' launching at 18:09:16 UTC to increase capacity from 1 to 2, followed by instance 'i-01873d8e2733a9df2' launching at 18:14:11 UTC, which further increased capacity from 2 to 3. This increased the total number of active instances in the target group to three, which is also corroborated by Figure 5 showing three healthy targets registered with the load balancer. The EC2 instances view (Figure 3) provides a snapshot reflecting multiple running instances resulting from this scaling activity.

EC2 Instances Scaling In: The system's ability to scale in by terminating surplus instances as load decreased was also demonstrated. While the primary focus of the 'cloudwatch_alarm.png' (Figure 6) is a scale-out event, the ASG activity history in Figure 4 shows earlier scale-in activities triggered by a low CPU utilization alarm ('TargetTracking-ServerASG-AlarmLow-...'). Specifically, instance 'i-06f130e527ee17cc' was terminated at 18:02:01 UTC (May 29, 2025), which reduced the desired capacity from 3 to 2, and subsequently, instance 'i-04fbeb81a0e962a5a' was terminated at 18:03:07 UTC , reducing the desired capacity from 2 to 1. These actions illustrate the ASG's response to periods of lower CPU utilization, automatically adjusting capacity to match demand. The EC2 Instances console view (Figure 3) also shows instance 'i-04fbeb81a0e962a5a' with a "Terminated" status, consistent with this scale-in activity. This confirms the ASG's capability to revert to a smaller number of instances when the increased load subsides.

Load Distribution: The Application Load Balancer effectively distributed the incoming requests across all healthy instances within the '5000TG' target group. Figure 5 shows three instances registered and in a "Healthy" state, distributed across Availability Zones 'us-east-1a' and 'us-east-1b'. Evidence of active load distribution is presented in Figure 7, which displays the 'RequestCountPerTarget' CloudWatch metric. The graph shows multiple target group entities handling concurrent requests, with an increase starting around 17:45 UTC. However, this initial load test ended before generating sufficient requests. A subsequent test, between 18:05-18:10 UTC, confirmed the load was distributed and not concentrated on a single instance.

6 Summary and Lessons Learned

This project demonstrated the deployment of a scalable and resilient web application with a serverless backend on AWS. Key findings highlighted the effectiveness of decoupling the web tier from back-end processing through the use of AWS Lambda and SNS, and underscored the critical role of Auto Scaling and Load Balancing in achieving frontend elasticity. For secure and functional inter-service communication, proper IAM roles, security groups, and meticulous network configuration, including NAT Gateways and S3 Endpoints, proved paramount.

Several challenges were encountered and addressed during the assignment. Initial Lambda trigger mis-configurations were resolved by implementing an SNS fan-out pattern, which provided a more robust and flexible event-handling mechanism. Lambda execution issues, such as timeouts during S3 downloads and segmentation faults related to the 'grpcio' library (a dependency for the Gemini API), were overcome by utilizing S3 Gateway Endpoints for Lambdas operating within a VPC, ensuring adequate Lambda timeout settings, and building deployment packages in Lambda-compatible Linux environments to maintain binary compatibility of dependencies. Database schema errors, including "Table doesn't exist," "Specified key was too long," and "Unknown column," necessitated careful execution of Data Definition Language statements and schema adjustments, such as reducing VARCHAR lengths for indexed UTF8MB4 columns. Furthermore, Application Load Balancer health check failures, which occurred despite the application running correctly on EC2 instances, were typically traced back to incorrect Target Group port configurations or EC2 security group rules that did not accurately source the ALB's security group. These experiences underscored the value of an iterative debugging process and comprehensive testing across all layers of a multi-service cloud application.