DevOps Assignment 2

Eoin Fennessy - 30/04/2023

Table of Contents

0. Introduction and System Architecture

The system architecture diagram shown below (figure 0.0.1) represents some of the infrastructure and services that have been created for this assignment.

The system includes a VPC spanning three availability zones, each of which has a private and public subnet.

A simple web API application is deployed in the private subnets, and scaling of app instances is managed by an auto-scaling group. The app utilises a high-availability, multi-AZ RDS database. Application instances are registered with a load balancer's target group, and have been made publicly accessible using a load balancer endpoint.

A serverless, event-driven architecture using S3, SQS, and Lambda is also included in the system, in addition to other resources such as the NAT gateway and bastion host.

All infrastructural resources and services will be described in detail throughout this report.

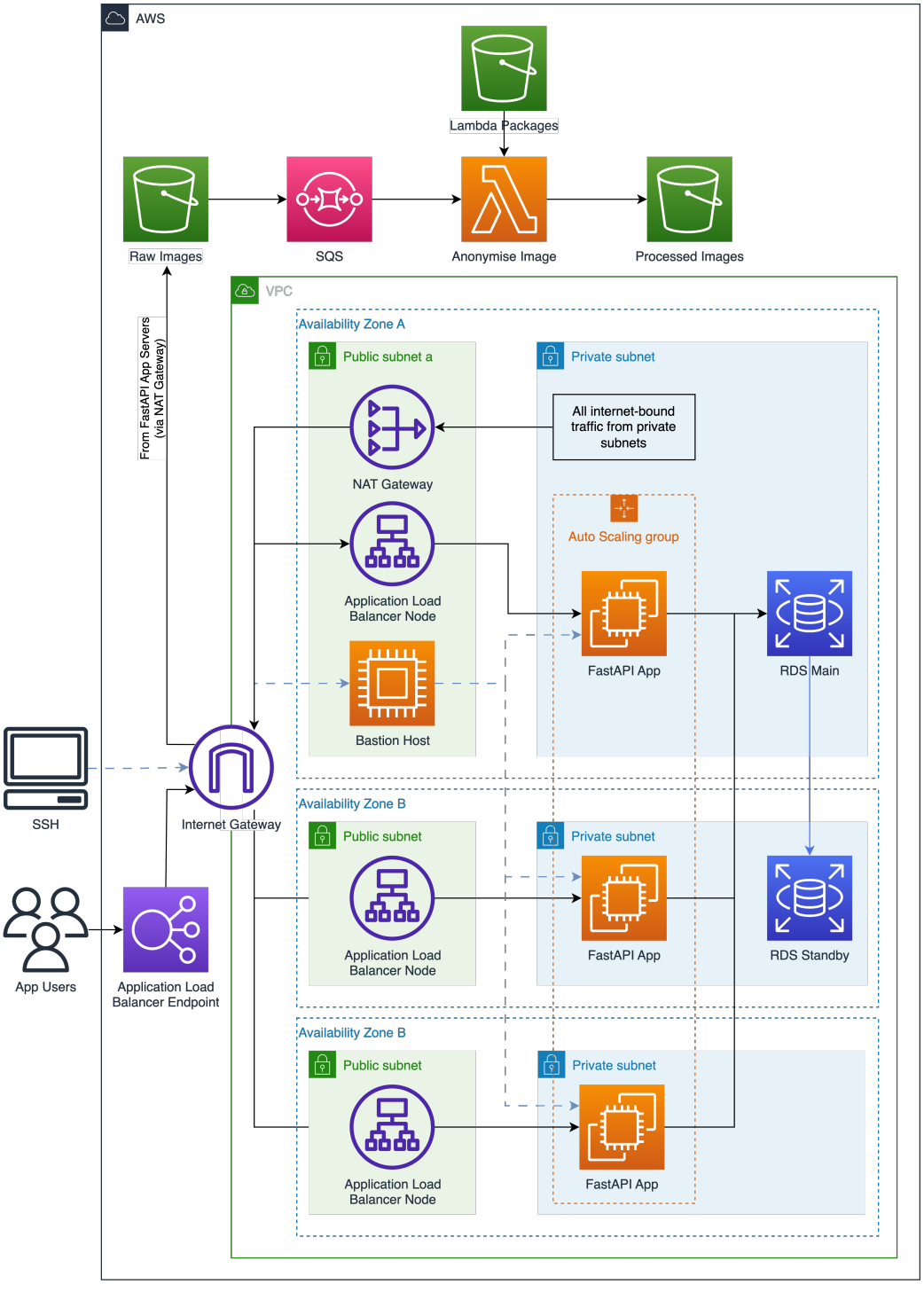

Figure 0.0.1: System Architecture

1. Web Application

MessageBoard is the name of the application deployed in this system and its source code can be found in the *messageboard* directory accompanying this report.

The application consists of a simple Python/FastAPI web API that connects to a PostgreSQL database and provides users endpoints to POST and GET messages.

Users can also upload images to the FastAPI app which then get stored in an S3 bucket. Each time an image is uploaded to the bucket an event-driven system anonymises the uploaded image by blurring the faces contained in it and then uploads the processed image to another bucket.

The FastAPI app serves an interactive Swagger UI documentation page (figure 1.0.1 below) that enables users to test the app's endpoints, and additional test endpoints are provided for monitoring instance health and tracing requests from client to instance via a load balancer.

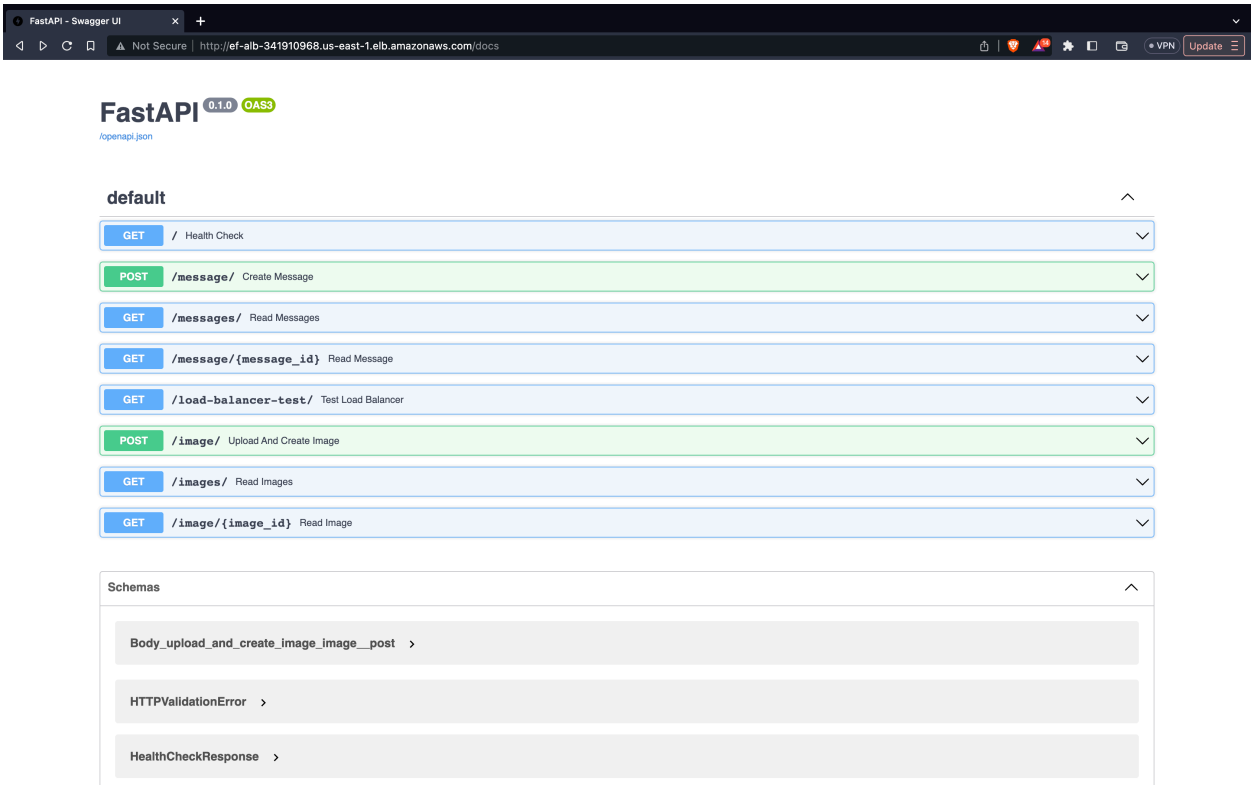

Figure 1.0.1: Swagger UI's Interactive API Documentation

At the time of writing the interactive documentation page can be accessed here.

1.1 APIs

The following web API endpoints are provided by the FastAPI application:

- **GET /**
  - A simple health check
- **POST /message/**
  - Creates a new message in the database using the required “author” and “body” fields provided in the request’s body
  - A successful response includes a JSON representation of the message, including its ID
- **GET /messages/**
  - Returns all messages
- **GET /messages/{message_id}**
  - Returns the message matching the ID provided
- **GET /load-balancer-test/[?trace_id=]**
  - Returns the ID of the instance that receives the request as well as the optional “trace_id” query parameter
  - The request, including the optional “trace_id”, is logged on the instance that receives it
- **POST /image/**
  - Takes an image file and returns the URLs for the raw image and the processed (anonymised) image
  - The raw image gets stored in an S3 bucket and triggers a process that anonymises the image and stores the result in another bucket
  - The image name, its URLs, and its ID are stored in the database
- **GET /images/**
  - Returns all image records stored in the database
- **GET /image/{image_id}**
  - Returns the image record matching the ID provided

1.2 Database

1.2.1 Availability and Scaling

The PostgreSQL database is managed by AWS's RDS service and has been configured for high availability and failover support by using a *Multi-AZ DB instance* deployment. A main instance of the database is used for all read/write access while a synchronous standby replica of the database is maintained in a different availability zone.

The standby instance does not assist in scaling the database's IOPS or throughput as it is only used for read/write access in the case of a failover event. To increase read capacity, RDS offers A *DB instance read replica* option, which asynchronously replicates data from the main DB instance to a single read-only DB instance. A *Multi-AZ DB cluster* deployment option is also available, which creates two readable standby DB instances in addition to the main read/write instance and offers semisynchronous data replication.

1.2.2 Application Integration

The FastAPI app interacts with the PostgreSQL database using the SQLAlchemy object-relational mapper. In `database.py`, the ORM connects to the database using environment variables that have been set on the instance by the user data script generated by Terraform.

database.py

```
...

DB_TYPE = os.environ.get("DB_TYPE", "postgresql")
DB_URL = os.environ.get("DB_URL")
DB_USER = os.environ.get("DB_USER")
DB_PASSWORD = os.environ.get("DB_PASSWORD")
DB_NAME = os.environ.get("DB_NAME")

SQLALCHEMY_DATABASE_URL = f"{DB_TYPE}://{DB_USER}:{DB_PASSWORD} \
@{DB_URL}/{DB_NAME}"

engine = create_engine(SQLALCHEMY_DATABASE_URL)
...
```

After connection, tables defined in `models.py` are created in the database if they do not yet exist. `crud.py` contains all the CRUD functionality used in the API endpoints found in `main.py`.

1.3 S3, SQS, and Lambda

When a user uploads an image to FastAPI it is stored in a "raw" bucket. When an object is uploaded to this bucket, an event message containing the bucket name, object key and other event data is sent to SQS, which then triggers Lambda by responding to a long-polling request with a message or a batch of messages from the queue.

Each batch of messages is processed by the `anonymise_image.lambda_handler` (see accompanying *lambda_function* directory). The function iterates through the batch of S3 messages, downloads each "raw" image into memory, detects and blurs faces using the OpenCV library, and puts the anonymised image into the "processed" bucket using boto3.

2. Custom AMI

A custom AMI for the FastAPI app was created using the code from the accompanying *messageboard* directory. This code was SCP-ed to a base instance created using an Amazon Linux 2 AMI, a virtual environment was then created using Python's `venv` module, and finally the app's dependencies listed in *requirements.txt* were installed in the virtual environment using `pip`.

After confirming that the app was working as expected (as well as setting up the CloudWatch monitoring described in section 8), an instance snapshot and custom AMI were created from the instance using the EC2 dashboard.

3. VPC

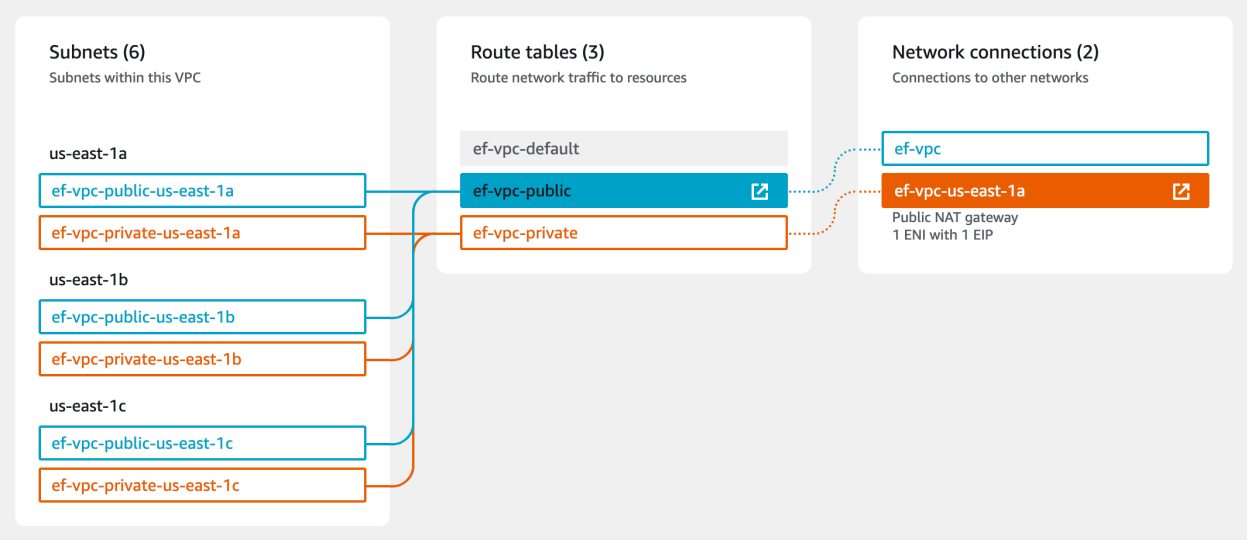

Figure 3.0.1: VPC Resource Map

Figure 3.0.1 shows an overview of the resources contained within the VPC. The VPC spans three availability zones (us-east-1a to us-east-1c), each of which contains a public subnet and a private subnet.

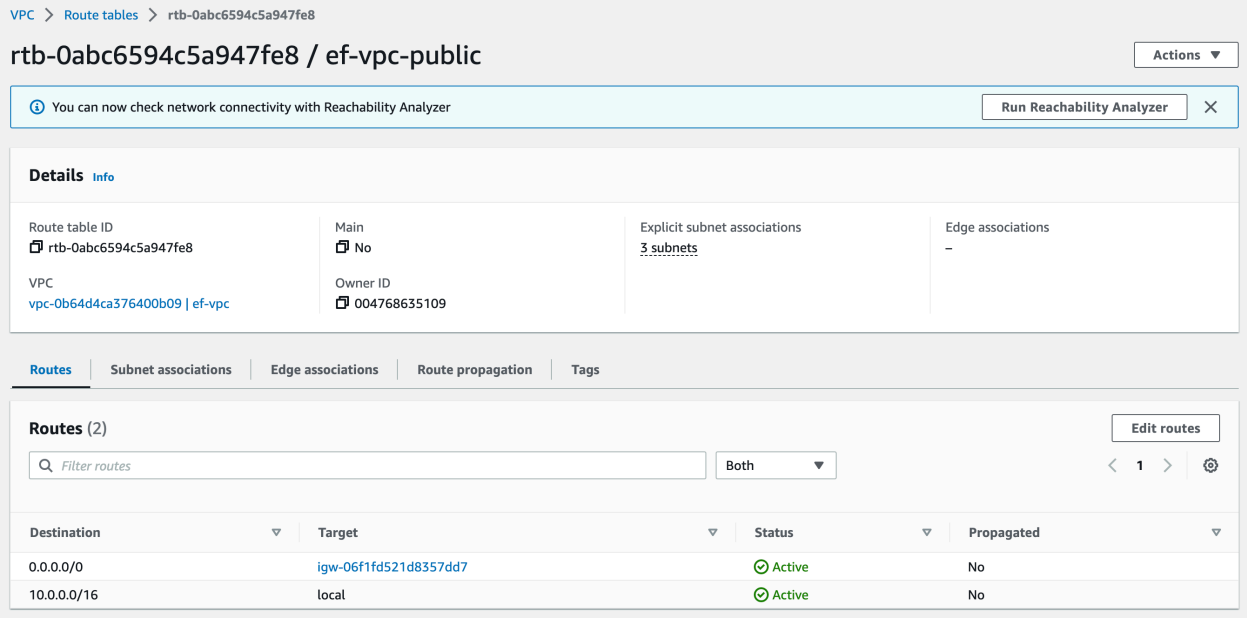

Figure 3.0.2: Public Routing Table

Figure 3.0.2 shows the public routing table used by the public subnets in the VPC. Traffic within the range of the VPC's CIDR block (10.0.0.0/16) remains within the VPC, while all other internet-bound traffic is routed through the VPC's internet gateway.

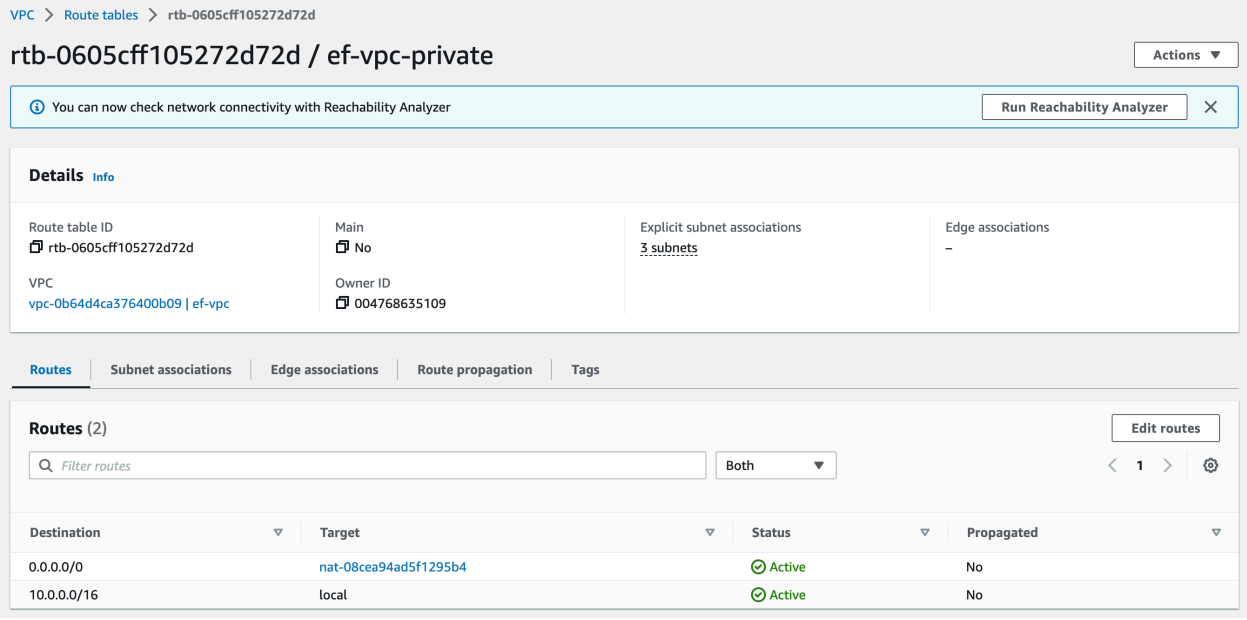

Figure 3.0.3: Private Routing Table

Figure 3.0.3 shows routing table used by the private subnets in the VPC. Again, traffic within the range of the VPC's CIDR block (10.0.0.0/16) remains within the VPC, but internet-bound traffic is routed to the a NAT gateway which is located within one of the VPC's public subnets.

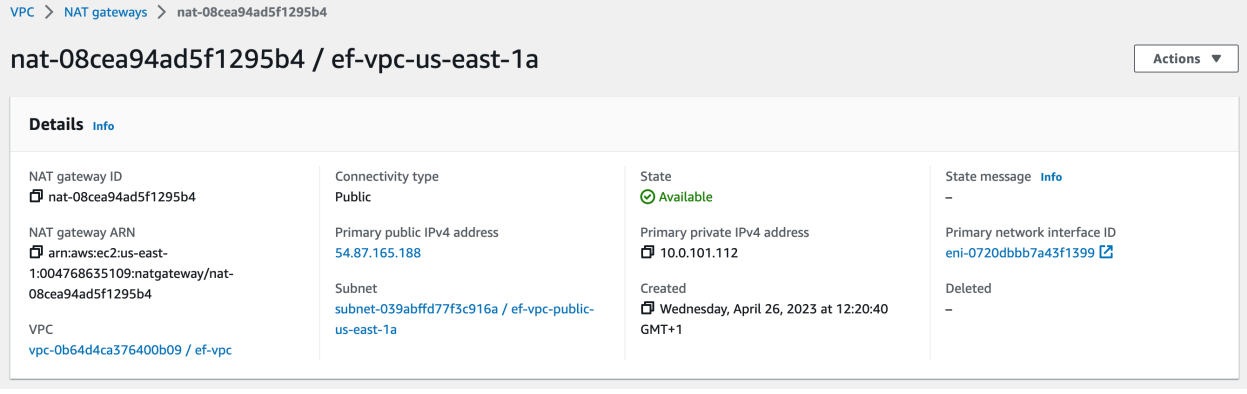

Figure 3.0.4: NAT Gateway Details

Traffic between private subnets and the NAT gateway is one-way. This means that private resources can connect to services external to the VPC using the NAT gateway, but cannot receive traffic from external sources. The NAT gateway accesses external services using the VPC's internet gateway.

3.1 Security Groups

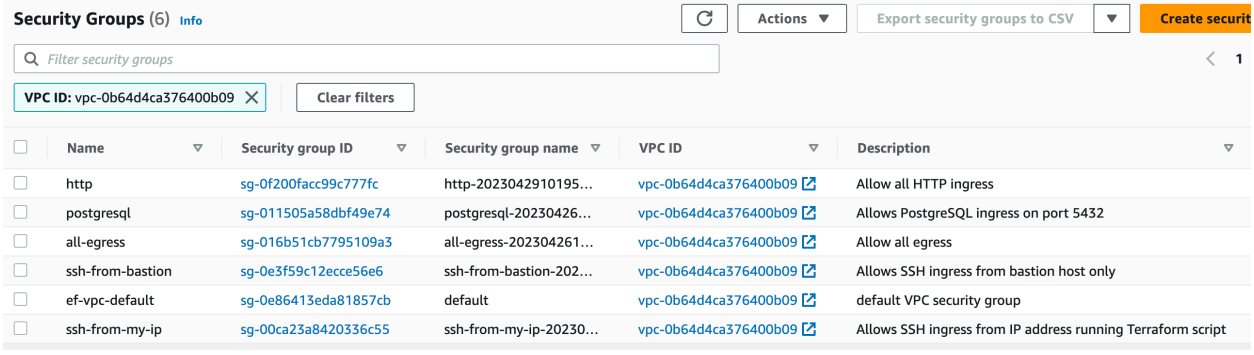

Figure 3.1.1: An Overview of the VPC's Security Groups

Figure 3.1.1 gives an overview of the security groups used by resources in the VPC. The *Description* column gives a good indication of the purpose of each. Below is a list of some of the key uses of these security groups.

- The *http* security group is used by both the application load balancer and the launch template for the FastAPI application.
- RDS uses the *postgresql* security group for database instances.
- *all-egress* is used wherever access to external services is required
- The *ssh-from-bastion* security group (figure 3.1.2 below) is used by the FastAPI launch template and allows SSH access from only the private IPv4 address of the bastion host. This is dynamically generated by Terraform
- *ssh-from-my-ip* (figure 3.1.3 below) is also dynamically generated by Terraform and uses the IP address of the host that runs the Terraform project. This security group is used by the bastion host.

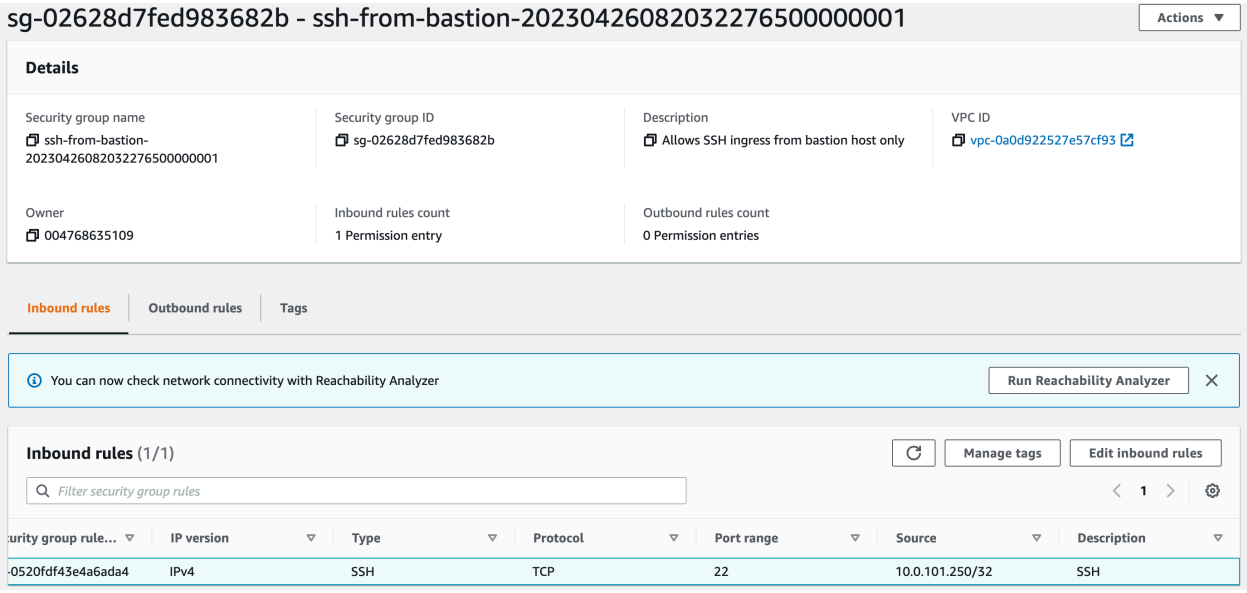

Figure 3.1.2: Security Group Allowing SSH Ingress from Bastion Host Only

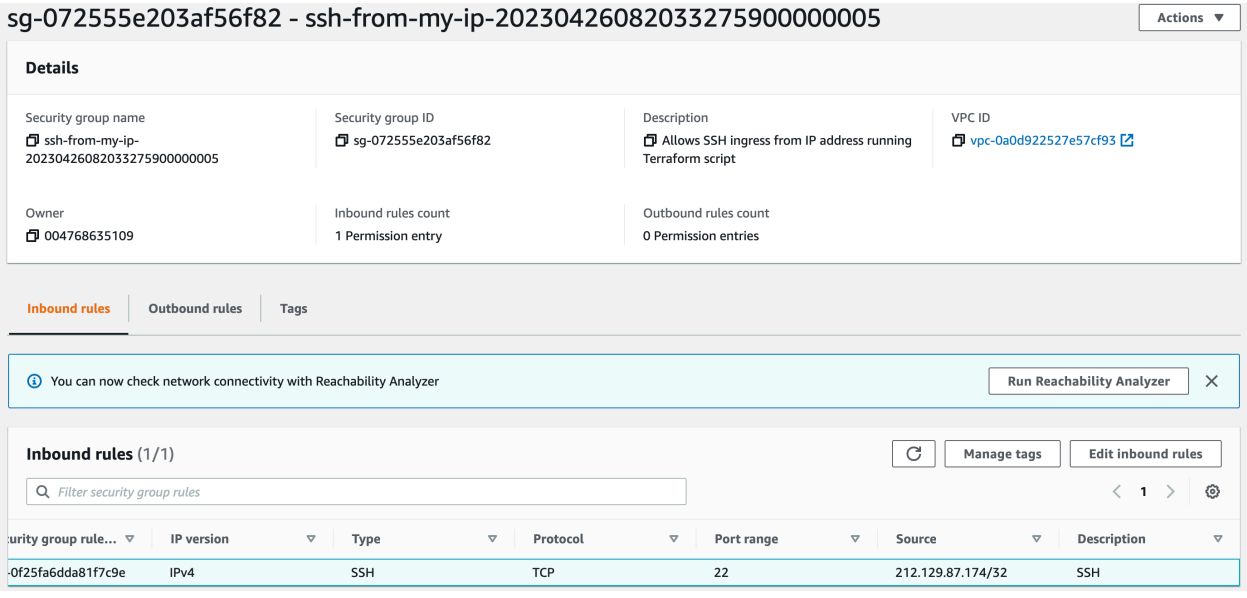

Figure 3.1.3: Security Group Allowing SSH Ingress from My IP Only

3.2 Bastion Host

A bastion host has been launched in a public subnet of the VPC. The purpose of this is to offer a single point of entry for SSH access to VPC resources. Resources in the VPC that require SSH access use a security group allowing SSH ingress from only the private IPv4 address of the bastion host, and the bastion host uses a security group allowing ingress from an allow list of trusted IPv4 addresses that require access (in this case, just my own IPv4 address).

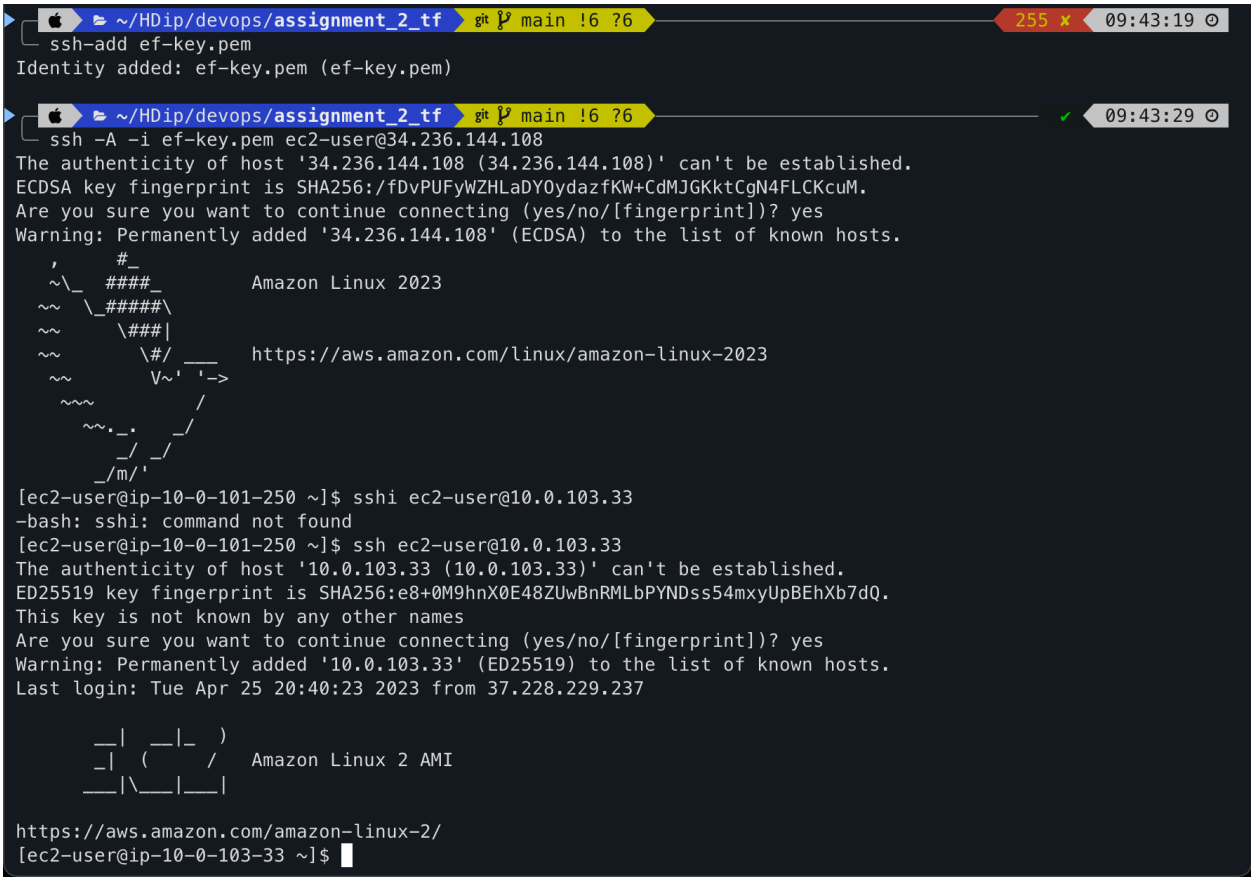

Figure 3.2.1: Bastion Host Usage

Figure 3.2.1 shows the process of SSHing into a private instance running in the VPC via a bastion host. The bastion host is SSH-ed into using its public IP address, and the private instance is then SSH-ed into from the bastion host using its private IP address. The process uses SSH agent forwarding to securely access the private instance without needing to store the private key on the bastion host.

4. Load-Balancing and Auto-Scaling

4.1 Application Load Balancer and Target Group

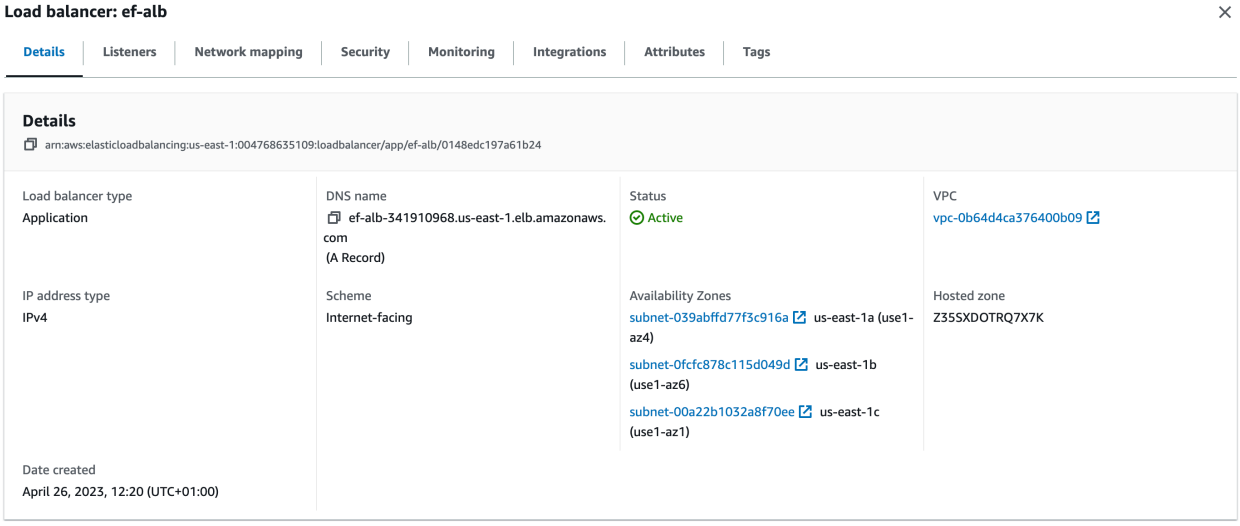

Figure 4.1.1: Load Balancer Overview

The application load balancer has been configured to listen for HTTP traffic on port 80. It distributes each request to port 80 of one of the FastAPI instances registered with the target group associated with the load balancer.

The load balancer offers an internet-facing endpoint that responds to each request with the IP address of one of the three load balancer nodes, each of which is located in the public subnet of one of three different availability zones and distributes traffic to the FastAPI instances within its AZ.

4.2 Launch Template

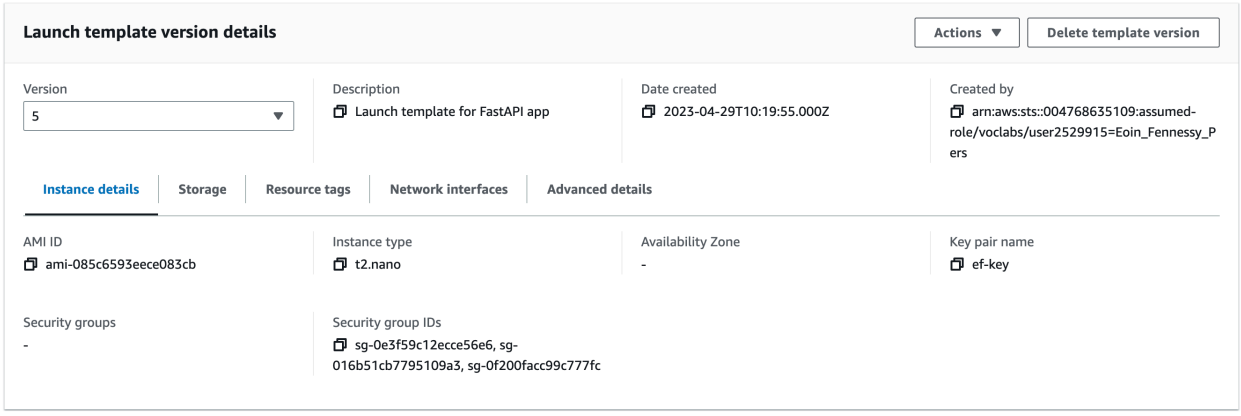

Figure 4.2.1: Launch Template Overview

- The launch template for the FastAPI app makes use of the custom AMI described in section 2
- A *t2.nano* instance type has been used for the launch template
- Security groups allowing all HTTP ingress, SSH ingress from bastion, and all egress are assigned to each instance created from the launch template
- A keypair for SSH access is used by the launch template
- Each instance is given the *LabInstanceProfile* IAM instance profile
- Detailed CloudWatch Monitoring is enabled

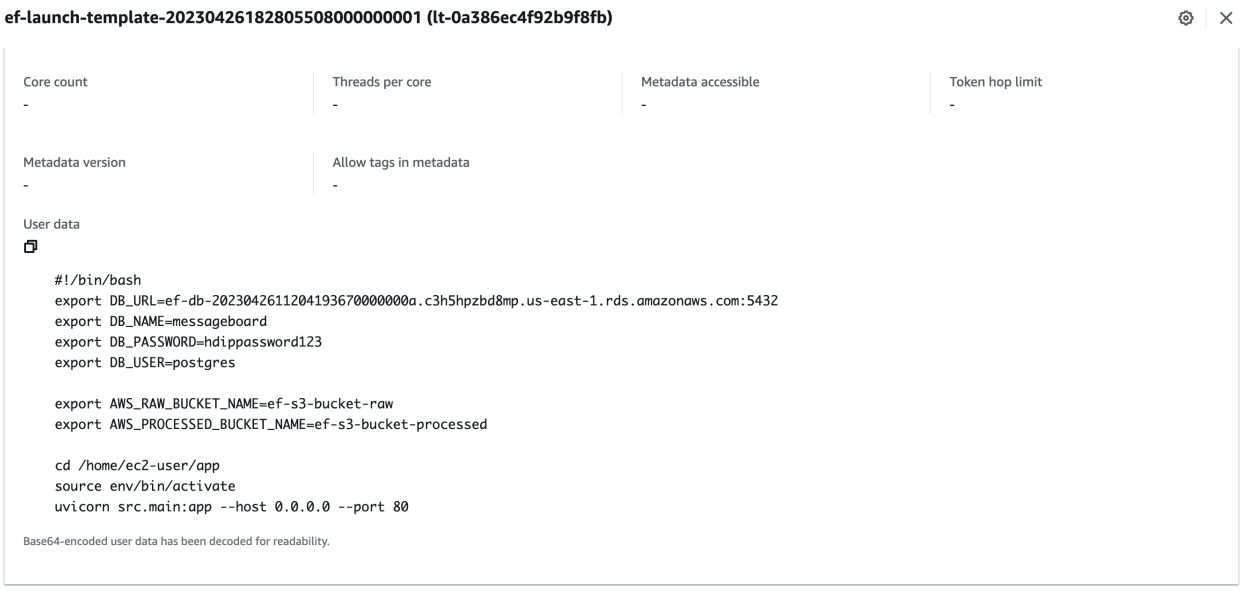

Figure 4.2.2: A Dynamically-Generated User Data Script

Figure 4.2.2 shows an example of a dynamically-created user data script that has been generated by Terraform. The script is used to set environment variables that will be used by the Python app before starting the FastAPI server. The environment variables include database details and secrets necessary for connection and S3 bucket details. The values of these variables are assigned by Terraform when creating the launch template (see snippet below).

```
user_data         = <<EOT
  #!/bin/bash
  export DB_URL=${module.db.db_instance_endpoint}
  export DB_NAME=${local.db_name}
  export DB_PASSWORD=${var.db_password}
  export DB_USER=${var.db_user}

  export AWS_RAW_BUCKET_NAME=${module.s3_bucket_raw.s3_bucket_id}
  export AWS_PROCESSED_BUCKET_NAME=$
{module.s3_bucket_processed.s3_bucket_id}

  cd /home/ec2-user/app
  source env/bin/activate
  uvicorn src.main:app --host 0.0.0.0 --port 80
EOT
```

Note that the approach taken here to manage secrets does not follow best security practices. More secure methods and tools for secret management, including AWS Secrets Manager, should be used in real-world projects.

4.3 Auto-Scaling Group

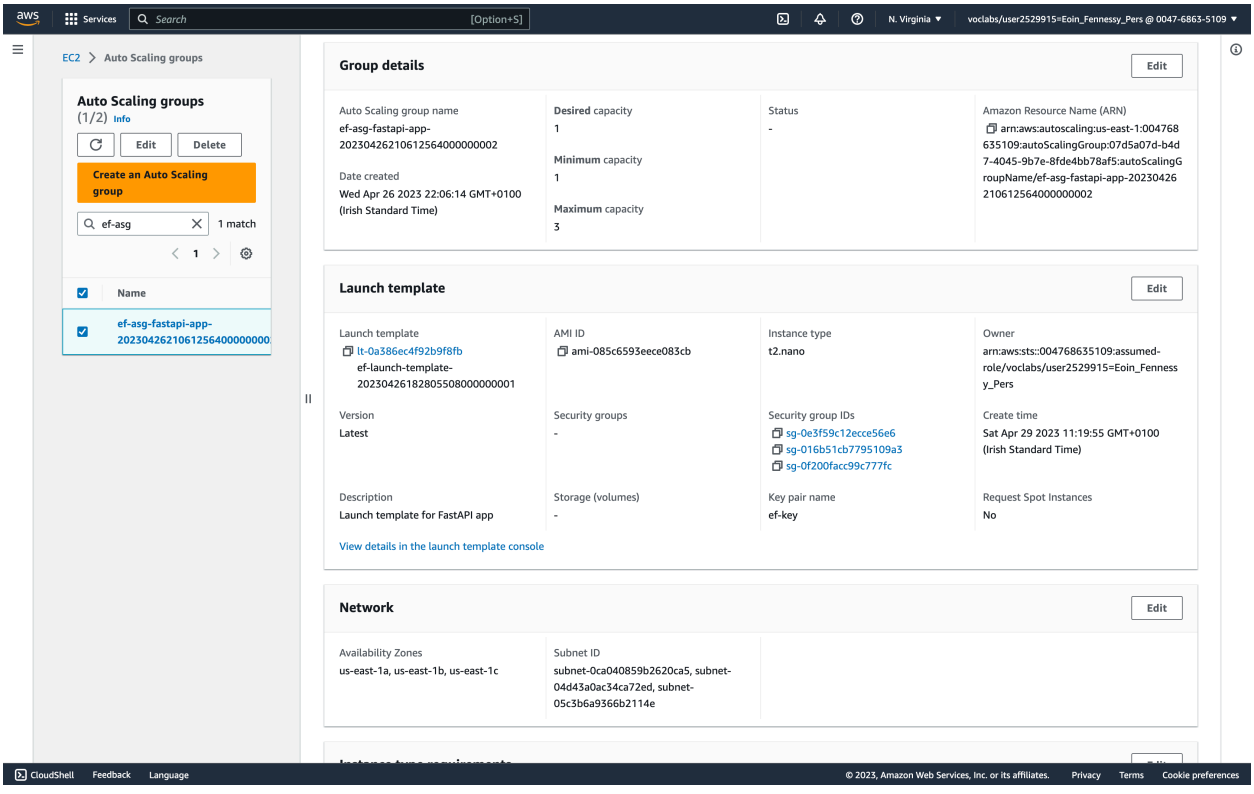

Figure 4.3.1: Auto-Scaling Group Overview

The auto-scaling group has been configured to use the launch template described in the previous subsection (4.2). Instances are launched into the private subnets of each of the three availability zones. The ASG has been attached to the load balancer described in section 4.1. All instances launched by the auto-scaling group are registered with the load balancer's target group, and the minimum and maximum capacities for the ASG are one and three respectively.

5. Scaling Policies and CloudWatch Alarms

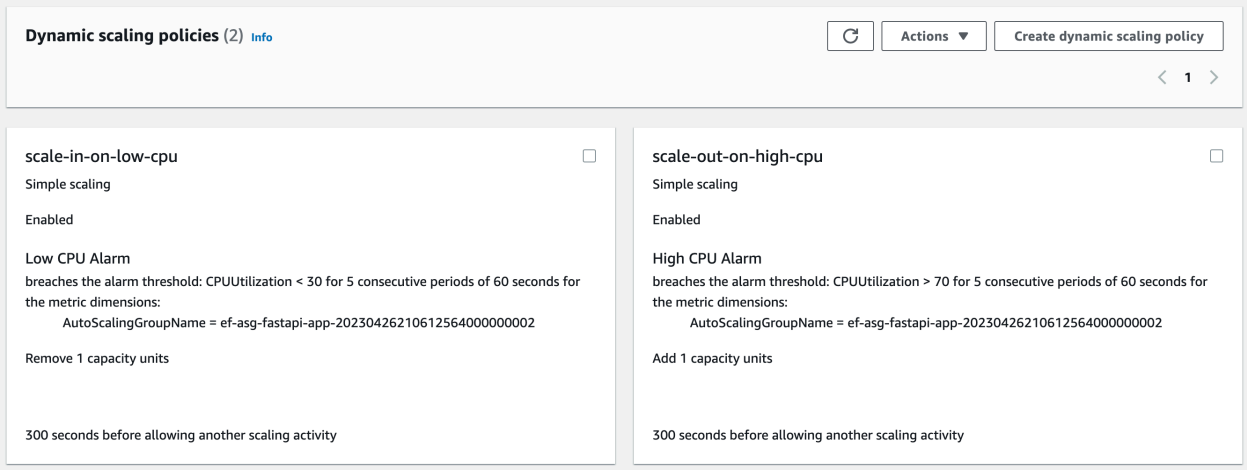

Figure 5.0.1: Simple Scaling Policies

Two simple scaling policies have been set up for use with the auto-scaling group, *scale-in-on-low-cpu* and *scale-out-on-high-cpu*. Both are based on CloudWatch alarms that use the average CPU utilisation of all instances in the auto-scaling group as the metric to determine whether or not scaling actions should be taken. A static scaling quantity of (plus or minus) one instance has been set for both policies, and scaling events can occur at most once every five minutes.

The CloudWatch alarms both evaluate the average CPU utilisation of the ASG every 60 seconds and go into a state of alarm when four out of the five most recent datapoints gathered cross the specified threshold.

5.1 Testing: Scaling Out

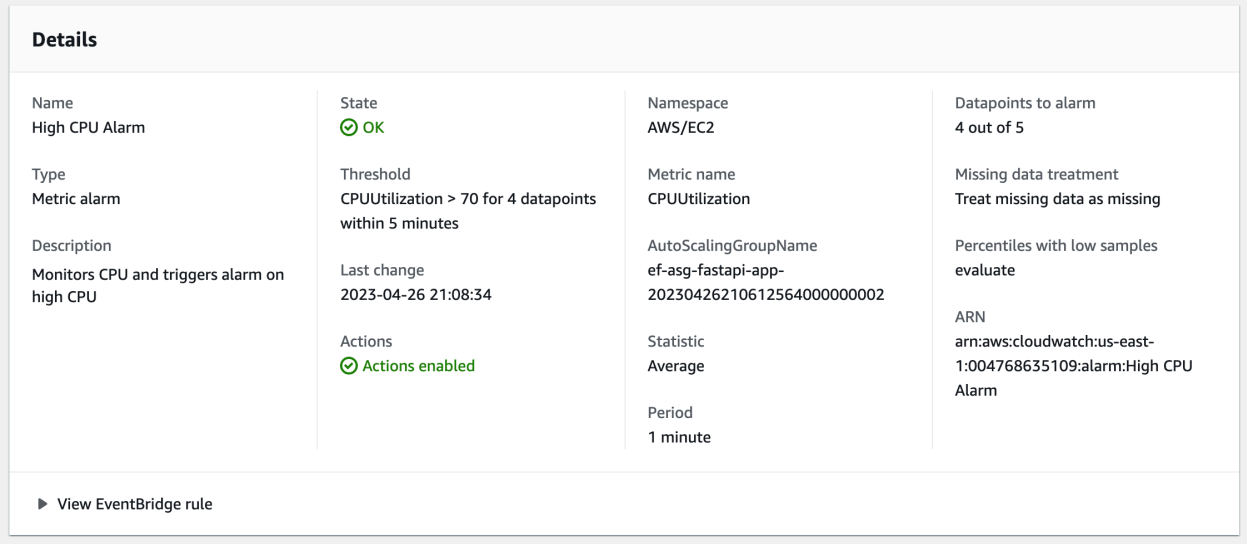

Figure 5.1.1: The High CPU Alarm used by the "scale-out-on-high-cpu" Scaling Policy

In order to trigger a scale out event the average CPU utilisation of the ASG needs to be greater than 70% for 4 out of the 5 most recent datapoints gathered.

We will now simulate CPU load on an instance running in the target group in order to trigger the high CPU alarm and scale out the number of instances. Figure 5.1.2 shows the load balancer's target group before beginning the test. There is one instance registered with the group.

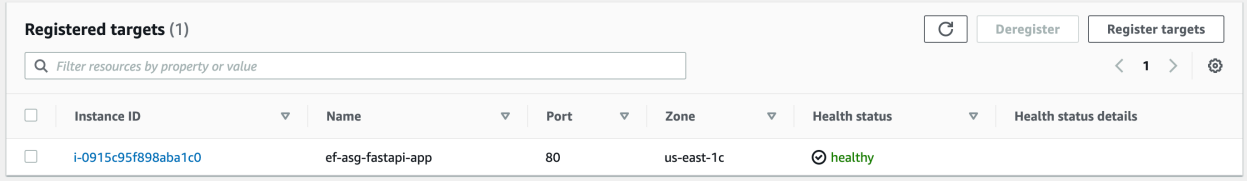

Figure 5.1.2: Load Balancer's Target Group Prior to ASG Test

We will now SSH into this instance and simulate CPU load by running `while true; do x=0; done`. This will keep CPU utilisation for the instance at 100% until it is stopped.

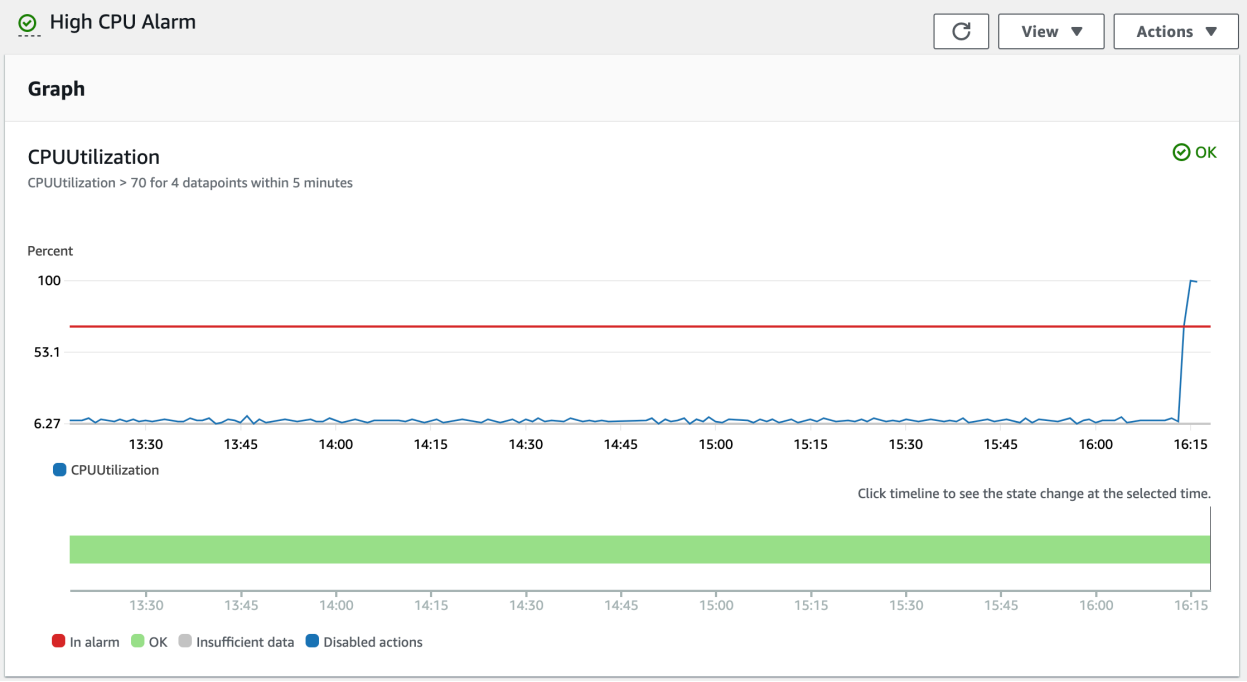

Figure 5.1.3: Initial Datapoints Above Threshold

After leaving the command running for a few minutes we can see that some datapoints have exceeded the 70% threshold, but not enough to trigger the alarm.

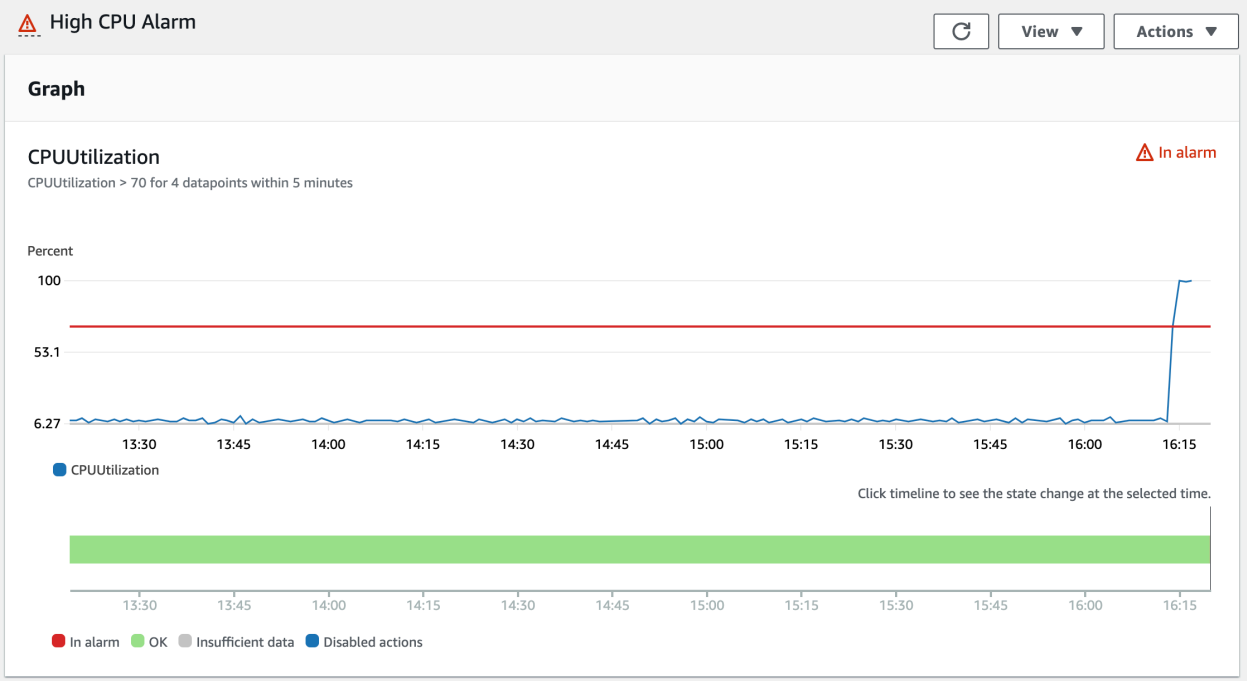

Figure 5.1.4: High CPU Alarm in State of Alarm

When four datapoints within five minutes exceed the threshold the alarm is put into a state of "in alarm".

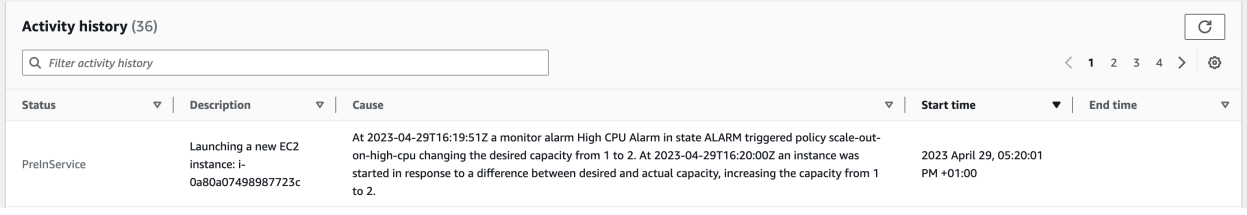

Figure 5.1.5: The Alarm Triggers Auto-Scaling Activity

In the auto-scaling group's activity we can see that this alarm has triggered a *scale-out-on-high-cpu* policy and the desired capacity has been increased from one to two instances, which in turn has caused another instance to be launched.

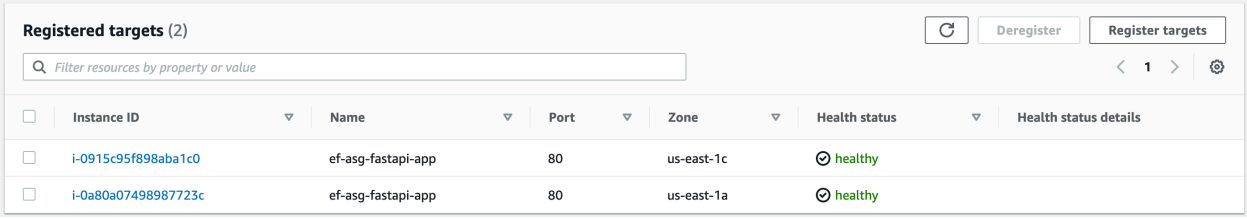

Figure 5.1.6: An Increase to two Registered Targets

The second instance can now be seen in the load balancer's registered targets view.

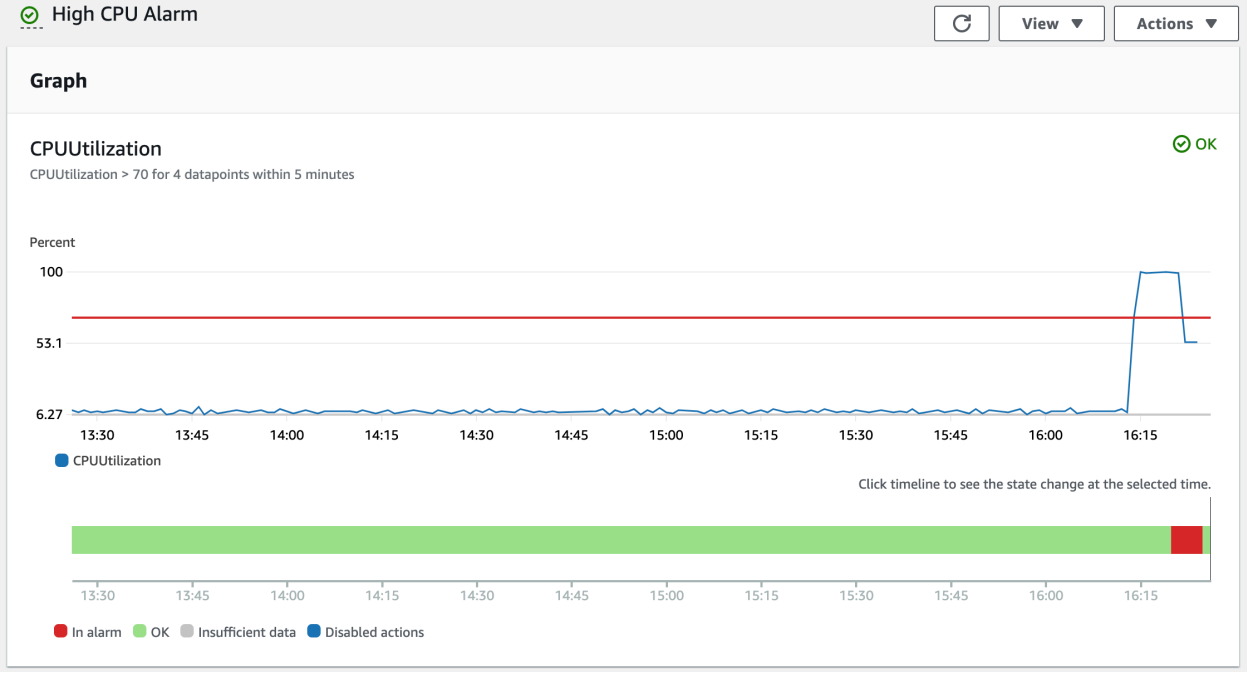

Figure 5.1.7: Average CPU Utilisation Drops to ~50%

Due to an additional instance being added to the ASG, the average CPU utilisation drops to ~50%.

Let's now run SSH into the new instance and run the command to increase the CPU utilisation to 100%…

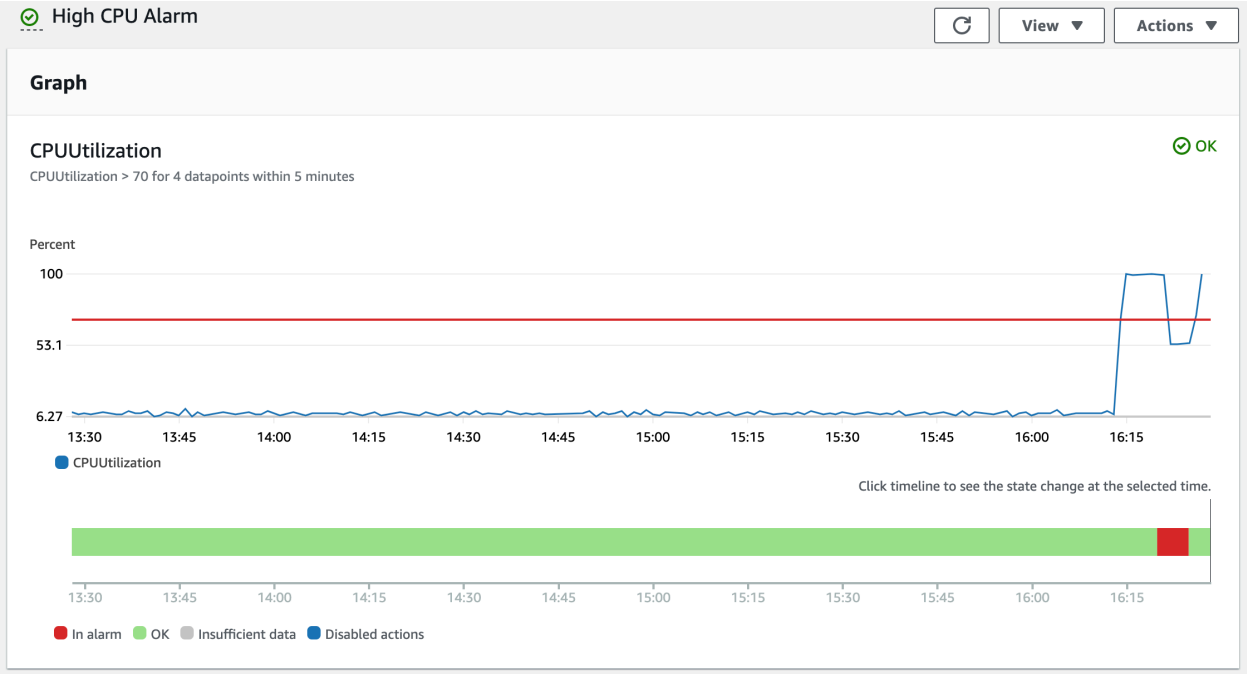

Figure 5.1.8: Average CPU Utilisation Rises to ~100% Again

This causes the average CPU utilisation for the ASG to rise to ~100% once again.

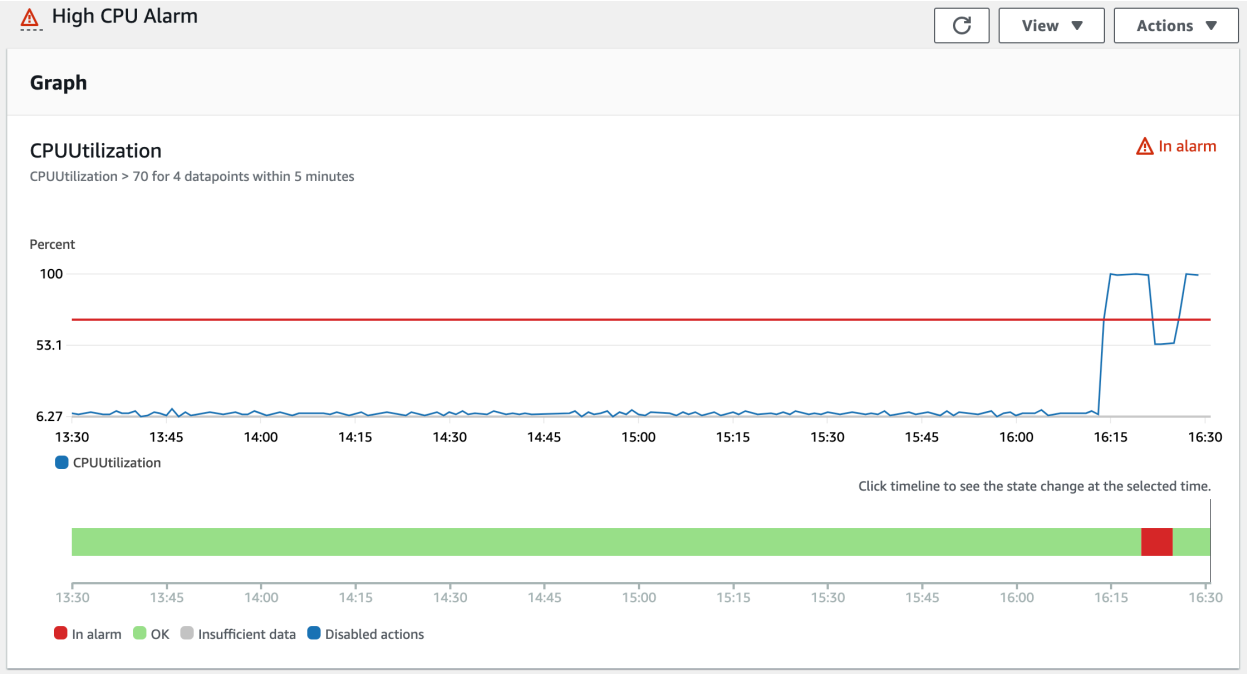

Figure 5.1.9: Another High CPU Alarm

After four datapoints exceed the threshold, the alarm is put in “in alarm” state once again.

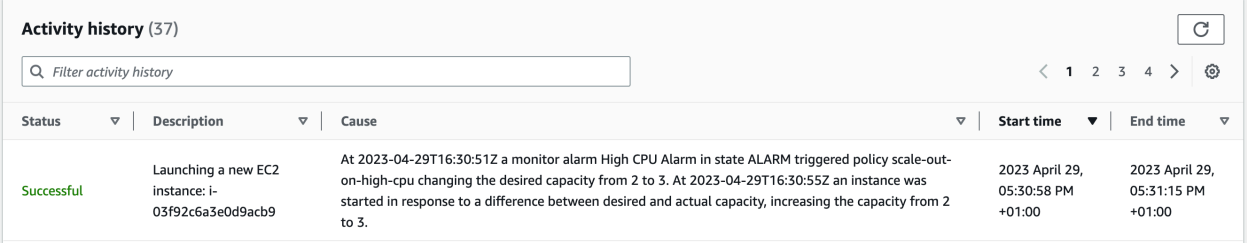

Figure 5.1.10: A Third Instance is Launched

The alarm triggers the creation of a third instance.

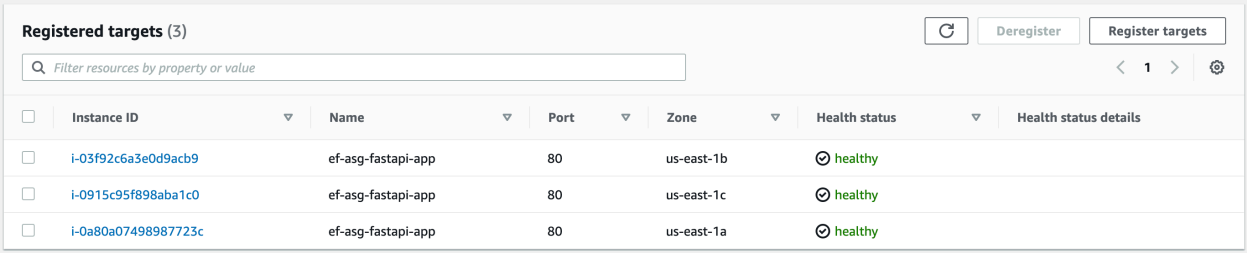

Figure 5.1.11: A Third Target is Registered

The count of registered targets in the target group rises to three (one instance in each AZ).

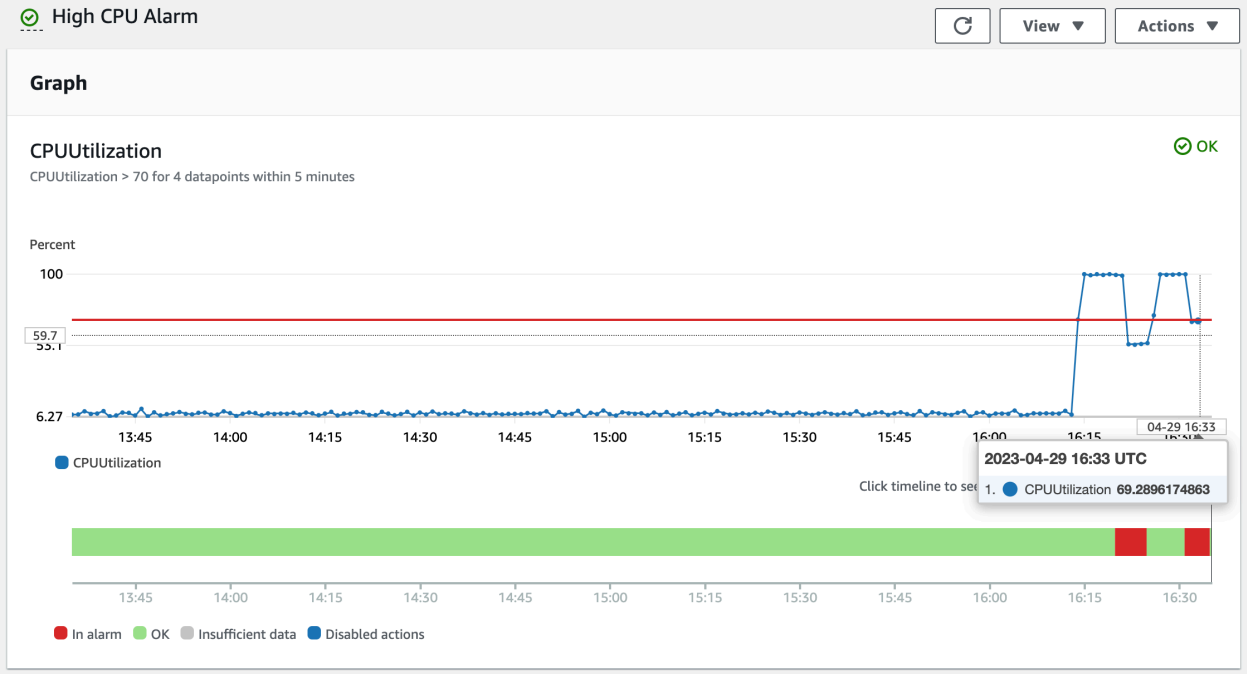

Figure 5.1.12: Average CPU Utilisation Drops Again

Finally, the average CPU utilisation for the ASG drops to 69% after the third instance is launched.

5.2 Testing: Scaling In

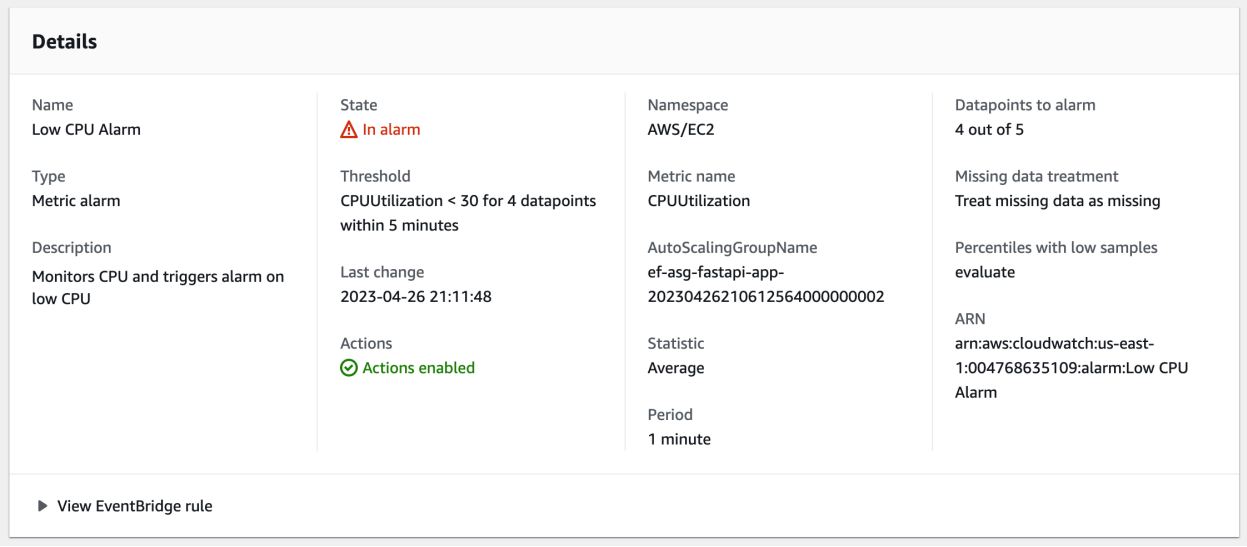

Figure 5.2.1: The Low CPU Alarm used by the "scale-in-on-low-cpu" Scaling Policy

In order to trigger a scale out event the average CPU utilisation of the ASG needs to be less than 30% for 4 out of the 5 most recent datapoints gathered.

Starting from where we left off in the previous section, we'll now scale the ASG back in. We'll do this by stopping the process that is creating CPU load on two of the three running instances.

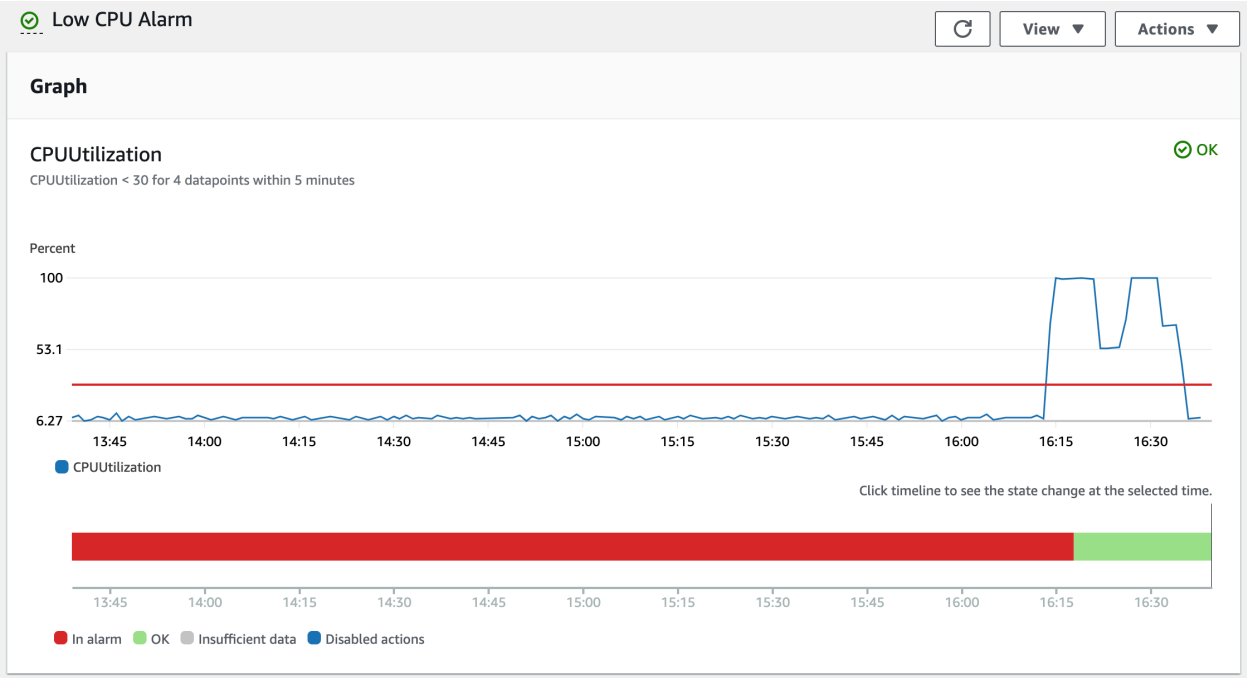

Figure 5.2.2: CPU Load Near Zero

After stopping the process the CPU load drops to zero, and a few datapoints are registered below the threshold before the alarm is triggered.

Figure 5.2.3: Low CPU Alarm in Alarm

After 4 out of the 5 most recent datapoints fall below the alarm threshold the alarm is put into the "in alarm" state.

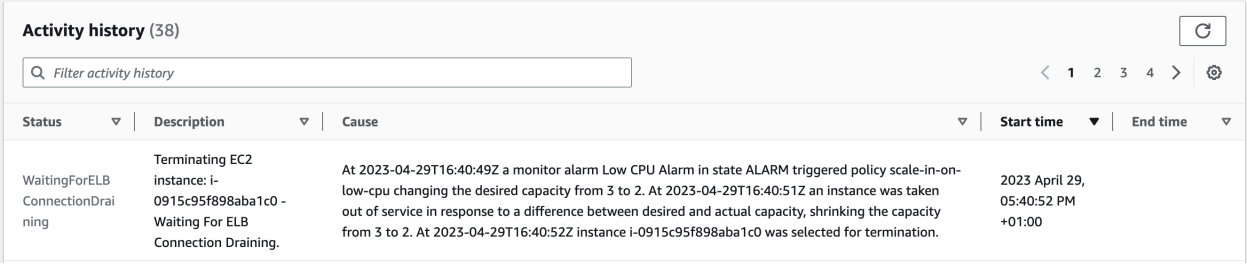

Figure 5.2.4: ASG Scale In Activity

This triggers the *scale-in-on-low-cpu* policy and a change in the desired capacity of the ASG from 3 to 2. Because of this, an instance is terminated.

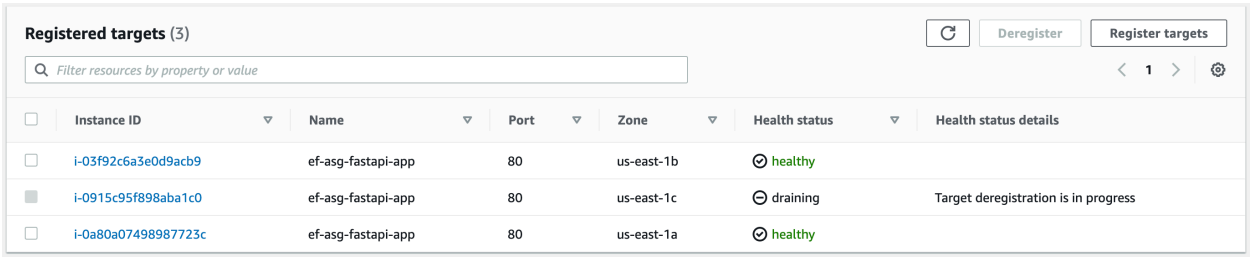

Figure 5.2.5: A Target is Deregistered

A target is then deregistered from the load balancer's target group.

Figure 5.2.6: Delayed ASG Activity

Even though the low CPU alarm remains in alarm, it takes over five minutes for another scaling activity to be triggered because of the cooldown period of 300 seconds that has been configured for each scaling policy. Just after six minutes before the last change in desired capacity, the desired capacity is again reduced by one, from two to one.

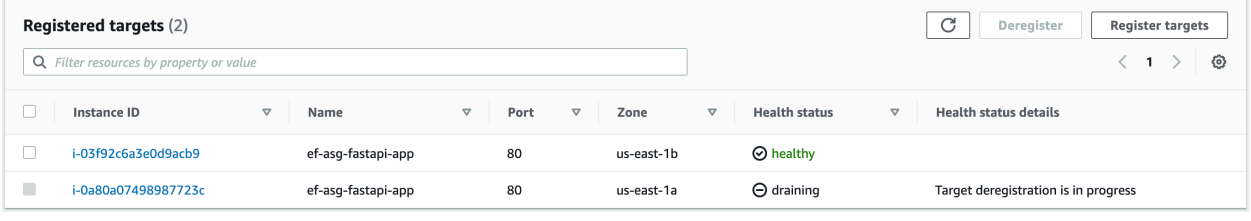

Figure 5.2.7: A Second Target is Deregistered

Finally, a second target is deregistered from the target group, leaving just one registered target.

5.3 Justification and Reasoning

Three primary considerations were taken into account when configuring the scaling policies.

1. Finding an optimum balance between application performance and cost

Average CPU utilisation of the ASG was chosen as the metric to use for informing scaling decisions as it is a good general indicator of how much load an instance is under. I felt that 30 to 70% was a reasonable range to aim to keep the average CPU utilisation within, so for this reason I used these figures as thresholds for the low and high cpu alarms. The figures are somewhat arbitrary, and I think that more in-depth stress testing of the application could help refine these figures by determining how CPU load affects the degradation in application performance.

2. Ensuring that scaling is both graceful and responsive

The five minute cooldown period between scaling events helps prevent errant scaling changes being made to the system before the full effect of the previous change is known. Additionally, I feel the 40% gap between alarm thresholds is wide enough that it would help prevent the system becoming unstable and constantly scaling up and down. Introduction of step scaling policies that determine the number of instances to scale by as a function of the measured metric may help improve the balance between gracefulness and responsiveness.

3. Outlier data measurements should not trigger or prevent scaling activities

Ensuring that four out of five consecutive datapoints are needed to trigger an alarm prevents outlier data from errantly triggering or preventing the triggering of scaling activity.

5.4 Alternative Scaling Metrics

Average CPU utilisation has been used in this system as the metric to determine when scaling activities should be triggered. However, some services or applications may not be particularly CPU-intensive, and it may make more sense to scale them based on an alternative metric that is more closely tied to their perceived performance. Such metrics may include the quantity of incoming or outgoing network traffic to or from instances in the ASG, the average memory usage of ASG instances, the number of messages in a queue that ASG instances need to process, or the total count of disk I/O operations within the ASG.

6. Test Traffic Generation Script

A Python script used to generate test traffic to send to the FastAPI instances via the load balancer can be found in the accompanying *test_load_balancer* directory.

The script makes use of Python's `asyncio` and `httpx` libraries to asynchronously send a user-specified number of requests to the **GET /load-balancer-test/** endpoint served by the FastAPI app.

The script allows the user to specify some parameters as command line arguments:

- `--endpoint` or `-e` - The URL of the load balancer including the endpoint to which requests will be sent
- `--base_trace_id` or `-b` - A string that will be used as the prefix for all trace IDs. Defaults to "load-balancer-test" if not specified
- `--request_count` or `-c` - The number of test requests to be sent. Defaults to 10 if not specified
- `--log_file` or `-l` - The filename to log to. Defaults to "./test_load_balancer.log" if not specified

The script generates a simple trace ID for each request. The trace ID is made up of a combination of the `--base_trace_id` and the request number (from zero to the specified count of requests minus one). Each trace ID is sent in the URL of a request as a query parameter named `trace_id`.

The **GET /load-balancer-test/** endpoint creates a log entry containing the trace ID on the instance that received the request, it then responds with the trace ID as well as the ID of the instance that handled the request. This information is then logged to a file on the client, along with some aggregate test data once all requests have been completed.

The following section describes a test setup and displays examples of logs gathered on both the client and server.

7. Load Distribution

Using the script described in section 6.1, the following command was executed to send 100 requests in parallel to the load balancer endpoint. Three instances were registered with the load balancer's target group at the time this test was carried out.

```
python test_load_balancer.py -e http://ef-alb-341910968.us-
east-1.elb.amazonaws.com/load-balancer-test/ -c 100
```

The logs resulting from running the script are described in the following two subsections.

7.1 Client Logs

Running the above script generated the following logs on the client side. These have been edited here for brevity. Each log entry displays the request's trace ID, one of three instance IDs, and the total time it took to complete the request. Additionally, some aggregated test data is displayed at the end of the main body of logs, including the count of requests & average request duration grouped by each of the three instance IDs, and the total time taken to complete the test.

```
2023-04-28 11:25:03,497    [INFO]    Request with trace ID "load-
balancer-test-1" was recieved by instance with ID
"i-086830d016d82cae5" and took a total of 0.2968 seconds to complete
2023-04-28 11:25:03,515    [INFO]    Request with trace ID "load-
balancer-test-0" was recieved by instance with ID
"i-08a886a4b13fdd1f6" and took a total of 0.3376 seconds to complete
2023-04-28 11:25:03,562    [INFO]    Request with trace ID "load-
balancer-test-9" was recieved by instance with ID
"i-08a886a4b13fdd1f6" and took a total of 0.3600 seconds to complete
2023-04-28 11:25:03,571    [INFO]    Request with trace ID "load-
balancer-test-3" was recieved by instance with ID
"i-06ac5c80bf334d932" and took a total of 0.3701 seconds to complete

...

2023-04-28 11:25:03,801    [INFO]    Request with trace ID "load-
balancer-test-63" was recieved by instance with ID
"i-086830d016d82cae5" and took a total of 0.5923 seconds to complete
2023-04-28 11:25:03,803    [INFO]    Request with trace ID "load-
balancer-test-96" was recieved by instance with ID
"i-086830d016d82cae5" and took a total of 0.5907 seconds to complete
```

```
2023-04-28 11:25:03,804    [INFO]    Request with trace ID "load-
balancer-test-89" was recieved by instance with ID
"i-086830d016d82cae5" and took a total of 0.5928 seconds to complete
2023-04-28 11:25:03,805    [INFO]    Instance i-06ac5c80bf334d932
completed 32 requests with an average request time of 0.4560 seconds
2023-04-28 11:25:03,805    [INFO]    Instance i-086830d016d82cae5
completed 35 requests with an average request time of 0.5436 seconds
2023-04-28 11:25:03,805    [INFO]    Instance i-08a886a4b13fdd1f6
completed 33 requests with an average request time of 0.5017 seconds
2023-04-28 11:25:03,809    [INFO]    Completed 100 requests in 0.6462
seconds
```

Note that the load was quite evenly distributed between the three instances, with each receiving between 32 and 35 requests each. This is due to the load balancer's use of the static *round robin* algorithm for load distribution. Use of a dynamic load balancing algorithm based on resource usage, request time, or number of connections may have resulted in a less even distribution of requests received, but perhaps more closely matched average request times.

7.2 Server Logs

Below are the corresponding logs for this test gathered on the instance with ID "i-06ac5c80bf334d932". Each log entry includes the request's trace ID corresponding to the above requests.

```
2023-04-28 10:25:03,529  [INFO    ]   Received request with trace ID:
"load-balancer-test-3"
2023-04-28 10:25:03,538  [INFO    ]   Received request with trace ID:
"load-balancer-test-21"
2023-04-28 10:25:03,540  [INFO    ]   Received request with trace ID:
"load-balancer-test-8"

...

2023-04-28 10:25:03,608  [INFO    ]   Received request with trace ID:
"load-balancer-test-94"
2023-04-28 10:25:03,613  [INFO    ]   Received request with trace ID:
"load-balancer-test-90"
2023-04-28 10:25:03,623  [INFO    ]   Received request with trace ID:
"load-balancer-test-86"
```

8. CloudWatch Monitoring of Custom Metrics

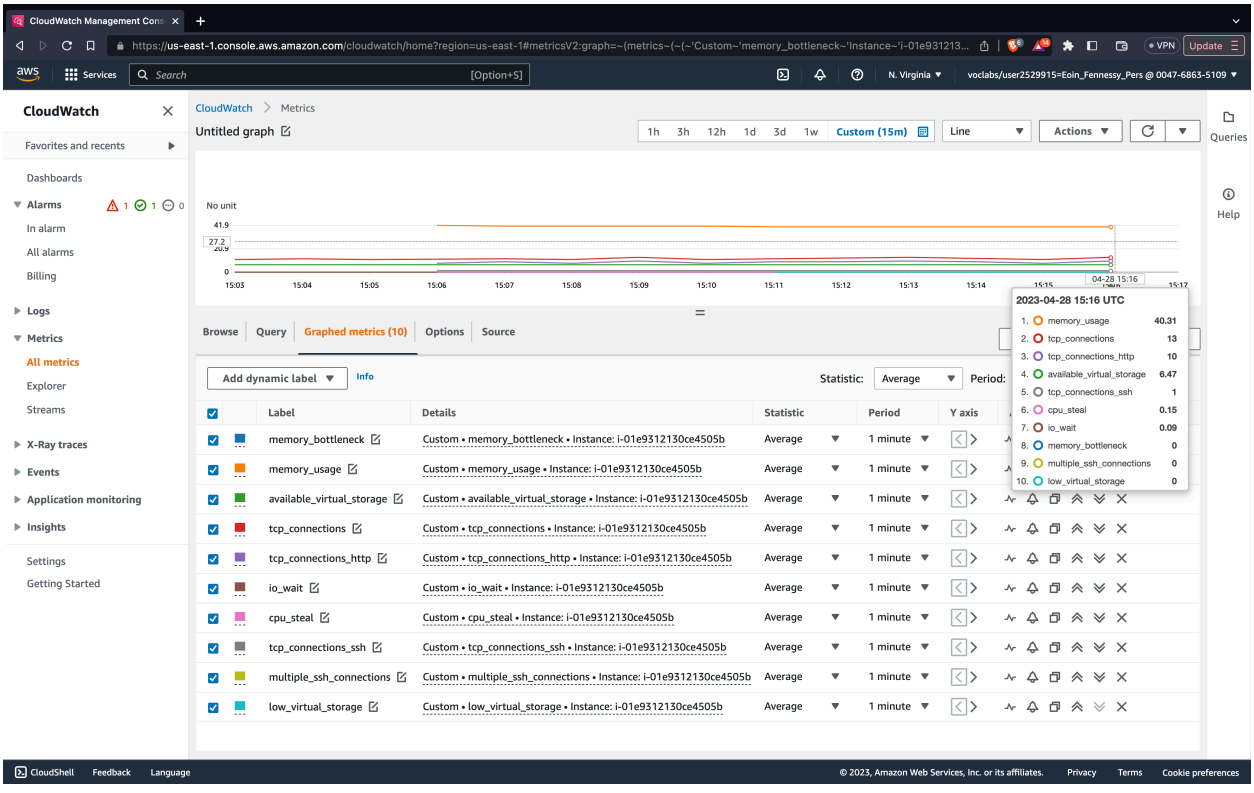

Figure 8.0.1: Custom Instance Metrics Displayed in the CloudWatch Dashboard

The *monitor.sh* file provided with this report has been set up to run once a minute on each FastAPI instance launched. The script first gathers various instance performance metrics and then pushes each to CloudWatch. An example of the custom metrics gathered from an instance and displayed in the CloudWatch dashboard can be seen in figure 8.0.1.

Here are the metrics each instance pushes to CloudWatch:

- `memory_usage` - The percentage of memory used
- `tcp_connections` - The total count of TCP connections
- `tcp_connections_http` - The count of HTTP connections
- `tcp_connections_ssh` - The count of HTTP connections
- `io_wait` - The IO wait percentage
- `cpu_steal` - The CPU steal percentage

- `available_virtual_storage` - The quantity of remaining virtual storage in gigabytes
- `memory_bottleneck` - A boolean value calculated using a combination of `memory_usage` and `io_wait`
- `multiple_ssh_connections` - A simple boolean value indicating more than one SSH connection is active
- `low_virtual_storage` - A simple boolean value that is true if the quantity of virtual storage is less than two gigabytes

9. Additional Functionality

9.1 Automation with Terraform

The primary additional functionality I explored in this assignment was the use of Terraform for automating the provisioning of infrastructure and deployment of applications on AWS. The *terraform* directory contains all the code used, and the *main.tf* file contains all the modules and resources used in the deployment.

Everything in the system excluding the creation of the FastAPI AMI has been automated, including the creation of the following.

- A three-AZ VPC with public and private subnets in each AZ
- Security groups
- Keypair
- Application load balancer & target group
- The FastAPI app's launch template
- Auto-scaling group
- Auto-scaling policies
- CloudWatch alarms
- RDS database
- Bastion host
- S3 buckets
- S3 notifications
- SQS queue
- Lambda package

- Lambda function
- Lambda-SQS event source mapping

The automation uses many of the high-level Terraform AWS modules, in addition to some lower-level AWS resources.

I don't have time to go into all the details of the automation, but here are some of the features of Terraform that I found useful.

Plans

Terraform works out the order resources declared in a project need to be created in and creates a plan to create them - the explicit `depends_on` meta-argument can be used for this too, but I found no need to use it in this assignment.

Idempotence

A state file is maintained for each Terraform project containing the state of all resources that are being managed by it. When a TF project is applied (run) multiple times, TF uses the state file to fetch the actual state of resources, and then creates a new plan to create/update only the resources that do not match the desired state. This makes it very easy to make gradual, iterative changes to the infrastructure, as well as tear down/destroy all infrastructure at once.

Lambda Packages

The Lambda module makes it very easy to package Lambda functions by using the Serverless Application Model (SAM) CLI. You provide it with a path to your source code and it figures out how to package the code using the conventional method for the runtime provided. For Python, it installs all dependencies listed in the *requirements.txt* file and places them in the root of the build directory, it then zips the package up and uploads it to Lambda or an S3 bucket.

You can also specify a `build_in_docker` boolean value to build your package in a locally-running container with an environment that matches the specified runtime - this remedied an issue I encountered where OpenCV was attempting to use binaries compiled for macOS in a Lambda function running in a Linux environment.

Dynamic Creation of Resources

With Terraform, resource/object attributes can be used anywhere within the project - even above where they have been declared in a file or in a different file entirely. Using these resource attributes to dynamically create things like environment variables, user data scripts, security group ingress rules, etc. really helped with the automation.

9.2 RDS

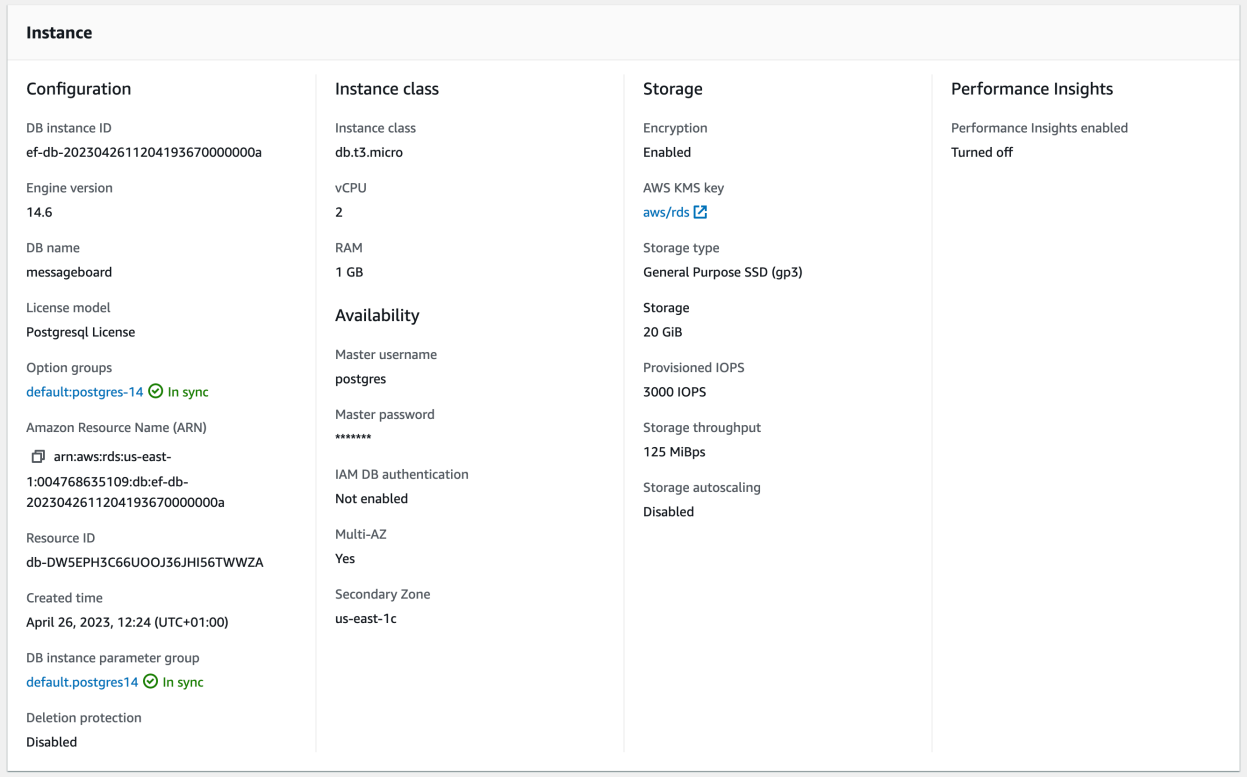

Figure 9.2.1: RDS Overview

RDS has been used to provide the PostgreSQL DB used by the app. It is a *Multi-AZ DB instance* deployment that maintains a replica of the main DB in a separate availability zone. This allows for high availability and failover support. Some additional configuration has been done to enable automatic maintenance of the DB at an off-peak time window.

Below is a quick demonstration of persistence using the FastAPI interactive Swagger docs that shows an entry being created and then read.

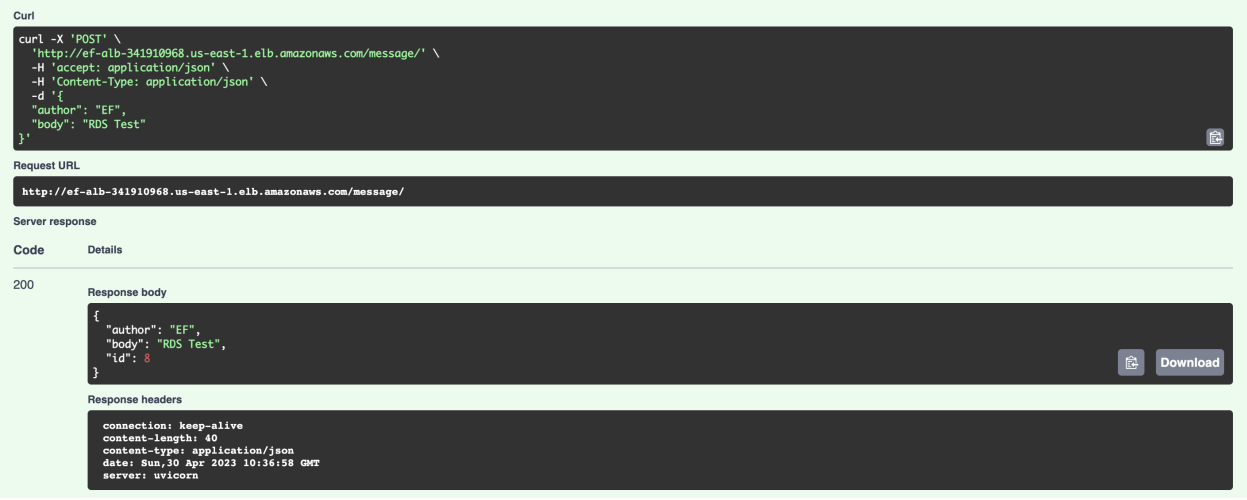

Figure 9.2.2: RDS Demonstration 1: Create a Message

Figure 9.2.2 shows a message being created using FastAPI. This will be persisted on the PostgreSQL DB.

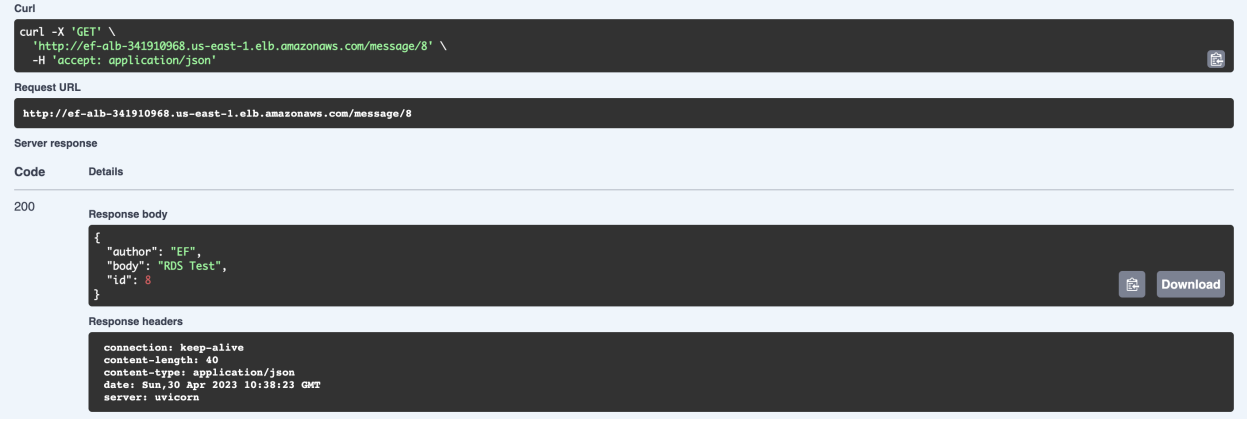

Figure 9.2.3: RDS Demonstration 2: Get the Message using its ID

Figure 9.2.3 shows the same message being retrieved from the database via FastAPI.

9.3 S3, SQS, and Lambda

A serverless, event-driven architecture was explored using S3, SQS, and Lambda. This is described in further detail in section 1.3, and the *lambda_function* directory contains the code used for the Lambda function.

Below is a demonstration of how these services work together and with the rest of the app.

Figure 9.3.: Starting Image

The above image will be used for the demonstration. All faces in this image will be blurred by the end of the demonstration.

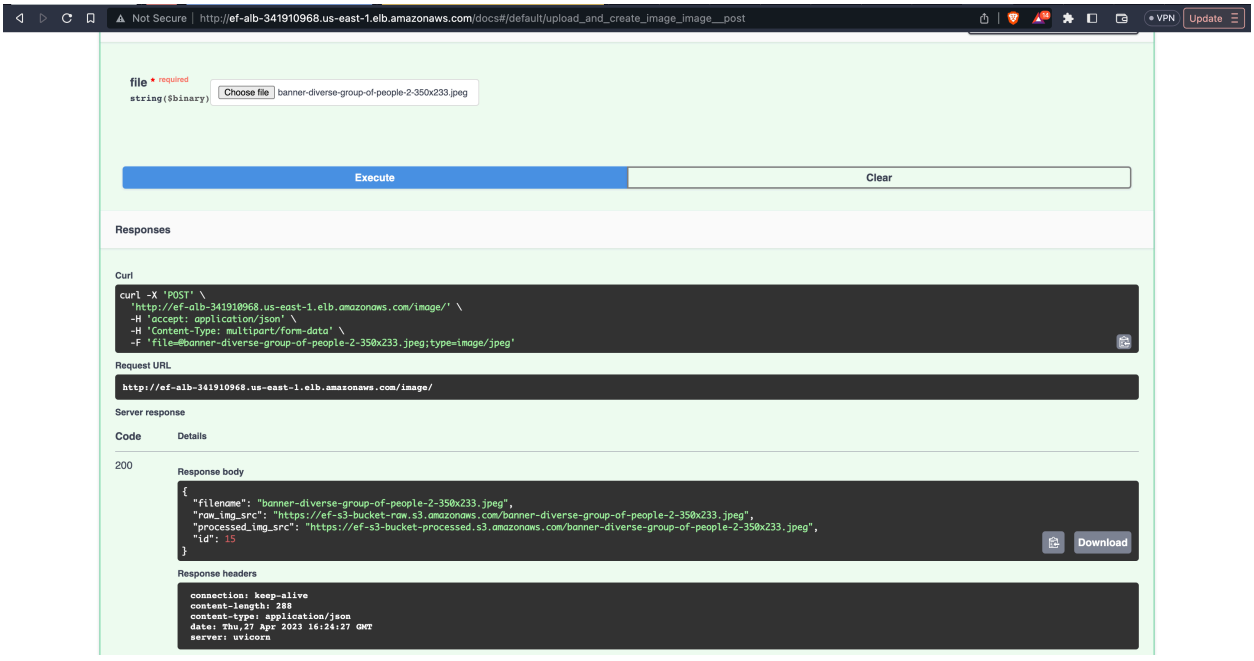

Figure 9.3.: Image Upload using Swagger Docs

The image is sent to the FastAPI `POST /image/` endpoint using the Swagger UI docs.

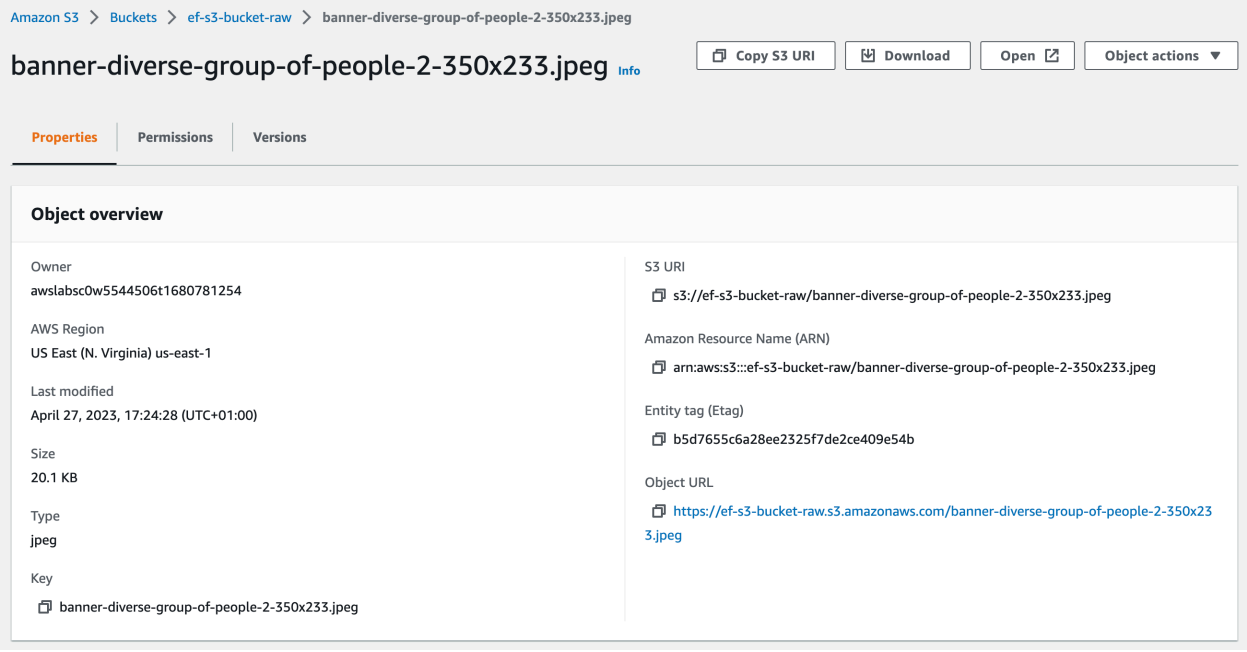

Figure 9.3.: The Image Object in the Raw Bucket

The image object can then be seen in the raw S3 bucket after the FastAPI app uploads it using boto3.

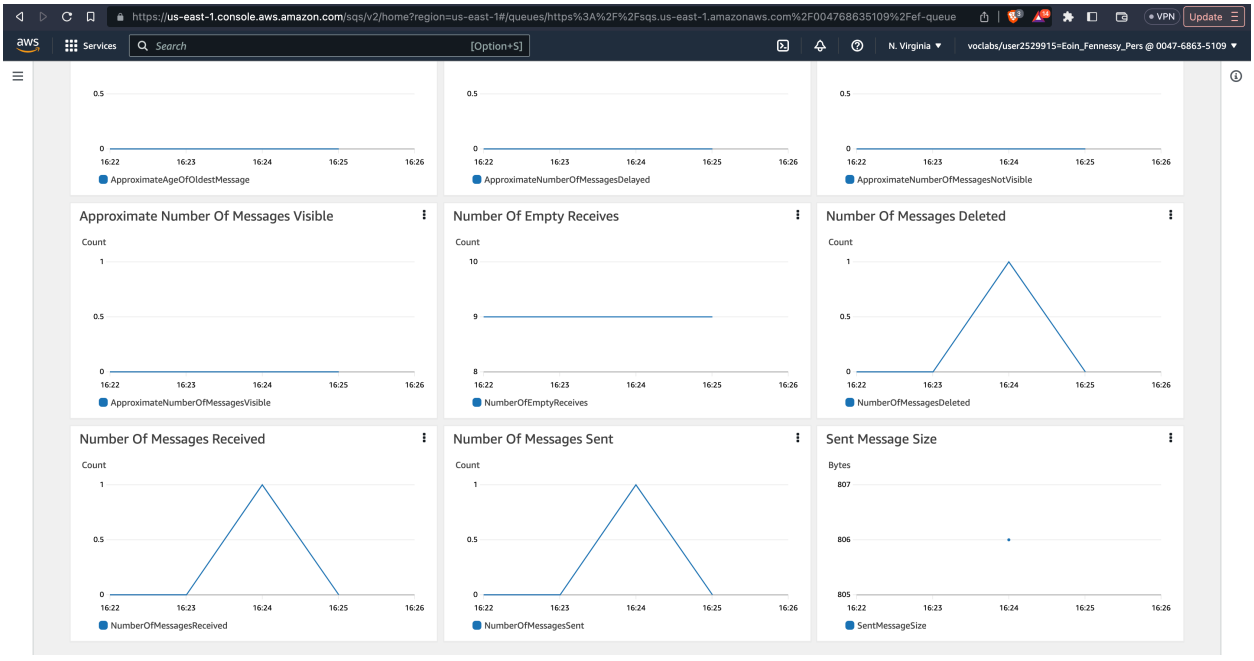

Figure 9.3.: The SQS Dashboard Showing Messages Received, Sent, and Deleted

S3 sends a notification to SQS, which triggers Lambda to receive and process a batch of messages (one message) from the queue. The above figure shows that the message has been received, sent, and deleted.

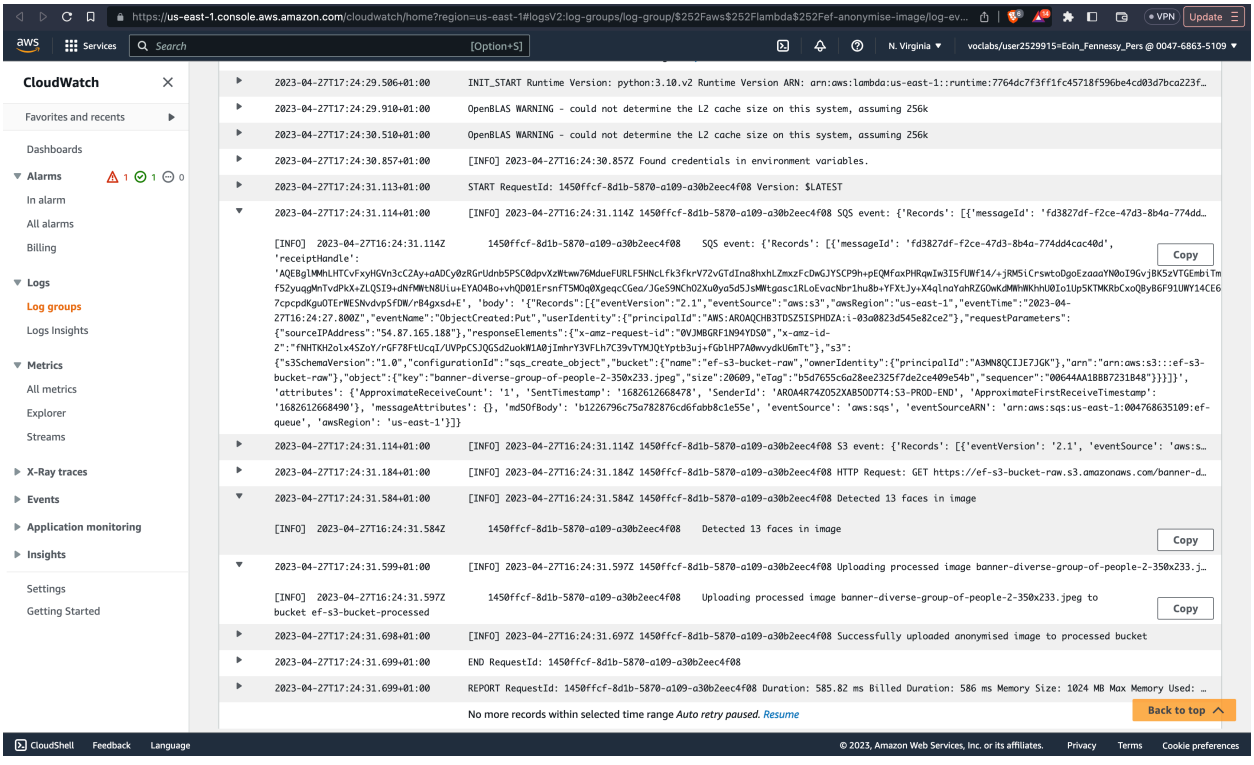

Figure 9.3.: Lambda Function Invocation Log

The Lambda function invocation logs show the SQS event received, which contains a body of records containing the single S3 message that was sent to SQS. The third last log entry indicates that the image was successfully uploaded to the processed images bucket.

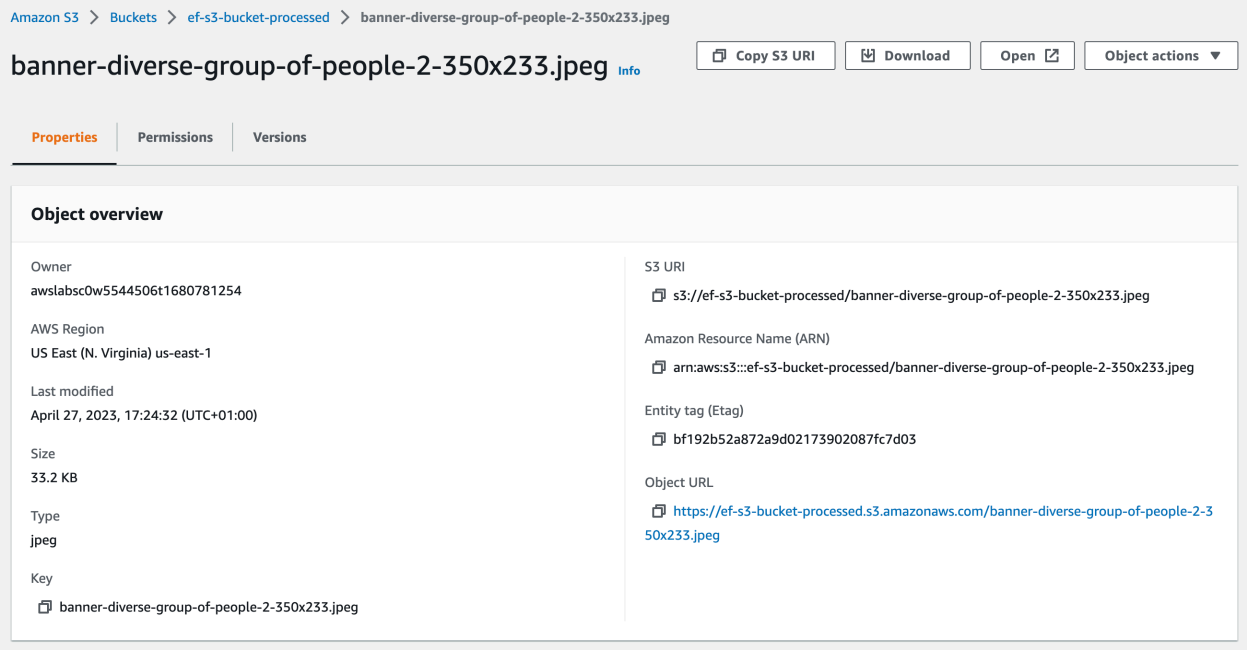

Figure 9.3.: The Image Object in the Processed Bucket

The anonymised image object can then be seen in the processed S3 bucket.

Figure 9.3.: The Resulting Image

And finally, the anonymised image can be obtained using the object URL.