SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF COMPUTER SCIENCE

AWS Blog App

CLOUD COMPUTING

Professors:
Emiliano Casalicchio

Students:
Soykat Amin,
1985500
Cristian Di Iorio,
1983177

Academic Year 2024/2025

Contents

1 Introduction

This project aims to design, deploy, and evaluate a multitier web application on Amazon Web Services (AWS), using services available within the AWS Academy Learner Lab environment.

The deployment must span at least two availability zones (AZs) to ensure high availability and fault tolerance. Additionally, the application must be configured with an Auto Scaling Group (ASG) that uses step scaling policies, rather than target tracking, to respond dynamically to changes in demand. The policy will include at least two scale-out rules and two scale-in rules based on CloudWatch alarms and performance thresholds.

A critical component of this project is evaluating the performance and scalability of the deployment under different workload conditions. All activities will be performed with consideration for the AWS Learner Lab's service availability and usage restrictions to prevent overuse or account suspension.

This report documents the system architecture, service selection, scaling policy configuration, test design, and detailed analysis of results. It provides insights into how the application performs under load, how well it scales in response to demand, and the practical trade-offs encountered when working within a constrained cloud environment.

The application is a simple Python Flask blog we created. Users can upload and see posts with images.

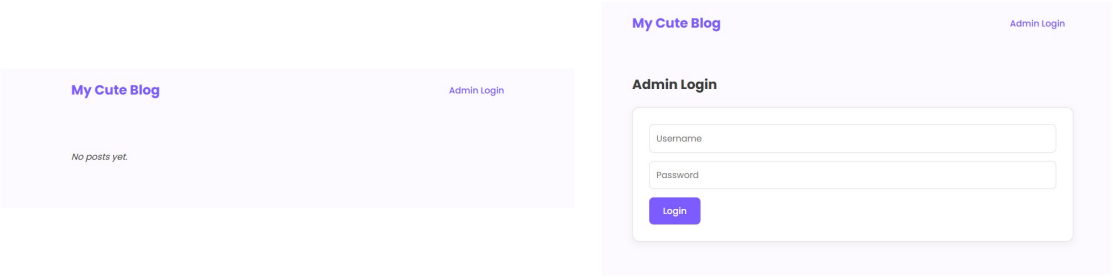


Figure 1: Blog with no posts and Login page

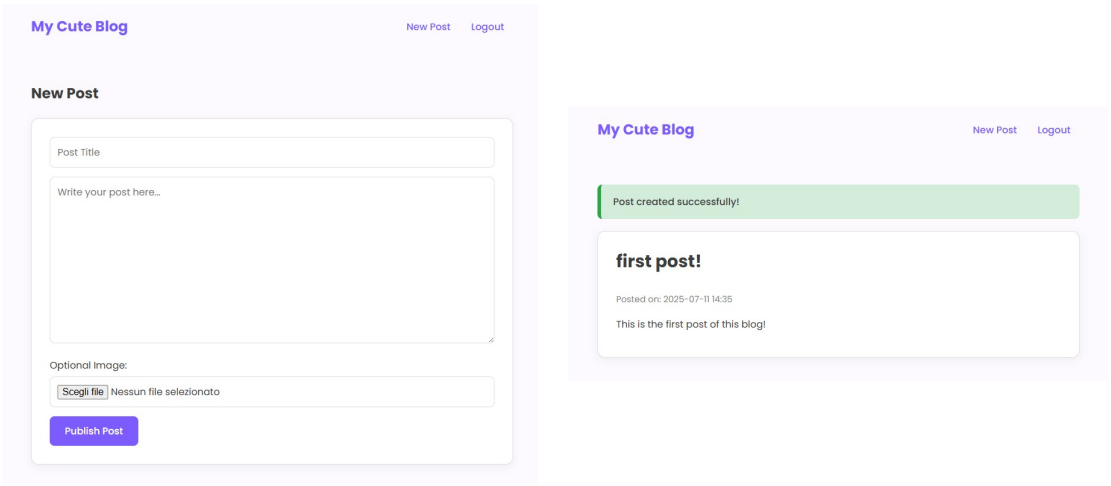


Figure 2: Post creation and First post

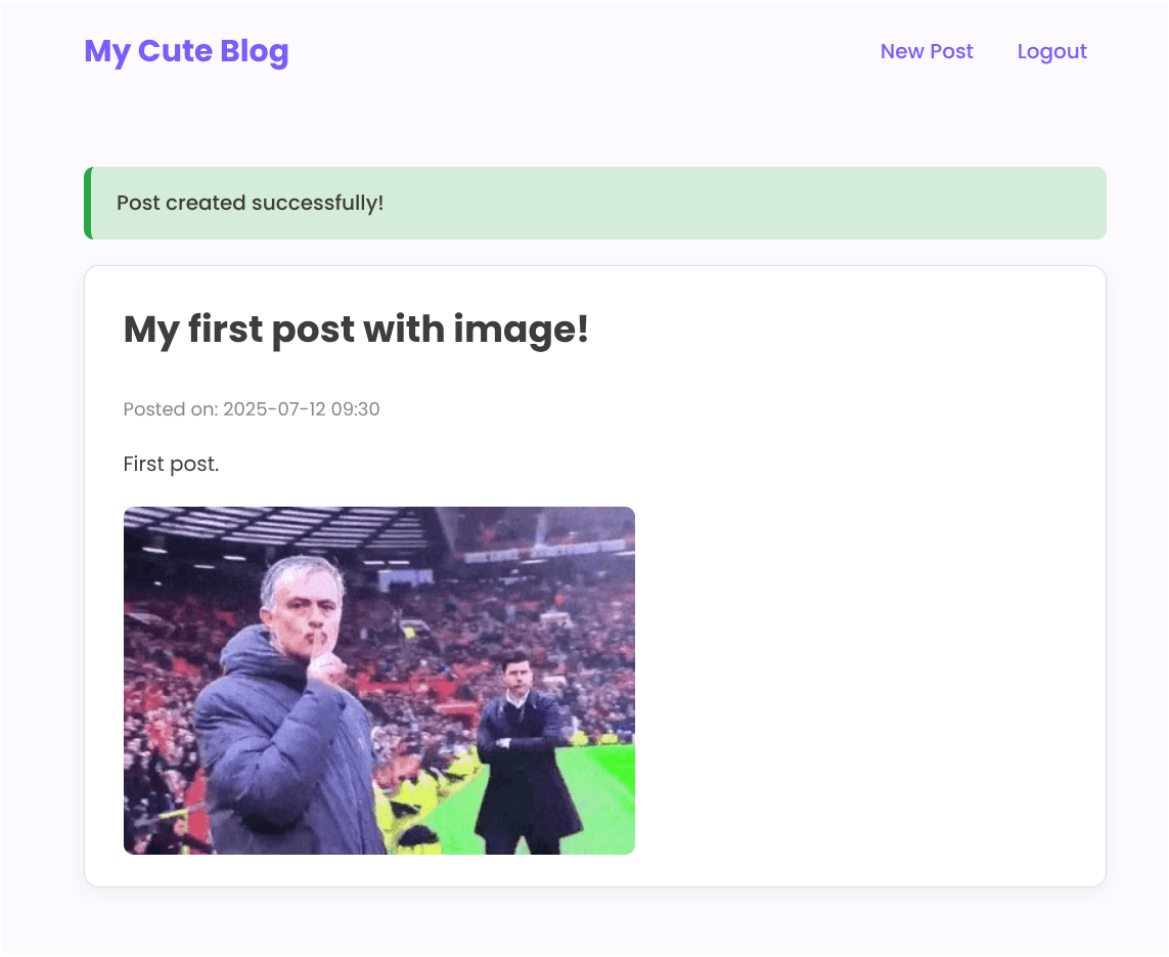


Figure 3: Post with image

2 Design and Architecture

2.1 Application Architecture

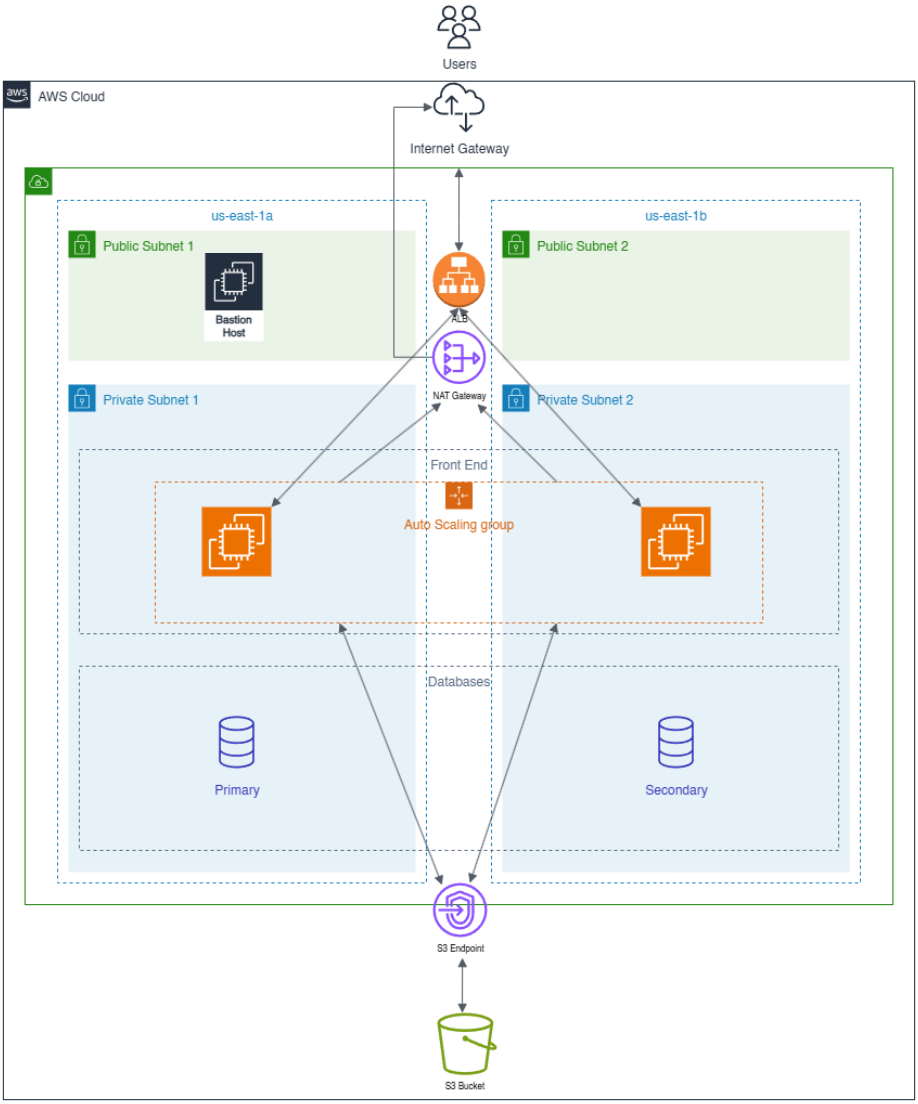


Figure 4: Application architecture

The application follows a classic three-tier model—comprising a Web Tier for traffic routing, an Application Tier for business logic, and a Data Tier for persistent storage—deployed over multiple Availability Zones to ensure high availability and fault tolerance.

2.1.1 Web Tier (Presentation Layer)

The Web Tier acts as the public-facing entry point for all user traffic. Its primary responsibility is to accept incoming HTTP/HTTPS requests from the internet and distribute them efficiently to the Application Tier. It does not contain any business

logic; it is purely a traffic management and distribution layer.

The core component is the Application Load Balancer (ALB). The ALB is deployed across two public subnets, one in each Availability Zone (AZ), ensuring high availability. It listens for user requests and forwards them to healthy instances in the Application Tier. It also performs health checks on the application servers, automatically routing traffic away from any that become unresponsive.

2.1.2 Application Tier (Logic Layer)

The Application Tier is the engine of the system, responsible for executing the business logic. It receives requests from the Web Tier, processes them, interacts with the Data Tier to read or write information, and generates the dynamic content that is sent back to the user.

The Application Tier consists of Amazon EC2 instances within an Auto Scaling Group (ASG). The EC2 instances are deployed in private subnets across two Availability Zones. This measure prevents them from being directly accessed from the internet. They can only receive traffic from the trusted Load Balancer. The Auto Scaling Group manages the fleet of EC2 instances, automatically increasing the number of servers during periods of high demand (scaling out) and decreasing the number during quiet periods (scaling in). Outbound internet access for software updates is provided securely via a NAT Gateway.

To securely administer the EC2 instances in the Application Tier, a bastion host was deployed in a public subnet. The bastion host acts as a jump server, allowing SSH access only from a trusted IP range while the application servers themselves remain in private subnets with no direct exposure to the internet. This setup adheres to the principle of least privilege and minimizes the attack surface, ensuring secure access for maintenance or debugging purposes without compromising the isolation of the private infrastructure.

2.1.3 Data Tier (Database Layer)

The Data Tier is responsible for the persistent storage and retrieval of application data. It is designed for high availability and data durability, ensuring that information is not lost in the event of a component failure.

The core components are Amazon Relational Database Service (RDS) for PostgreSQL and Amazon `S3`.

The RDS instance is deployed in a Multi-AZ configuration. This means a primary database instance is actively serving requests in one AZ, while a synchronous standby replica is maintained in a second AZ. All data is replicated in real-time between the two. If the primary instance fails, Amazon RDS automatically fails over to the standby replica with no data loss and minimal downtime. The database resides in private subnets and is protected by a dedicated security group that only allows connections from the instances in the Application Tier, ensuring the data is isolated and secure.

Amazon `S3` (Object Storage): Provides virtually unlimited, highly durable object storage for unstructured data. In this architecture, an S3 bucket is used to host user-uploaded images. A VPC Gateway Endpoint enables secure, private connectivity from the EC2 instances in the Application Tier (without traversing the public Internet) and bucket policies grant least-privilege access so that only the application's IAM role can read from or write to the bucket.

2.2 AWS Services Used

To construct this scalable and highly available architecture, the following key AWS services were utilized:

- **Amazon VPC (Virtual Private Cloud)**: Provided the foundational network isolation for the entire deployment. A custom VPC was configured with public and private subnets across two Availability Zones, along with route tables, an Internet Gateway for public traffic, and a NAT Gateway to allow private instances outbound access.
- **Amazon EC2 (Elastic Compute Cloud)**: Provided the virtual servers (instances) that run the web application code in the Application Tier. We used `t3.micro` instances to stay within budget while providing sufficient compute for the workload.
- **Application Load Balancer (ALB)**: Served as the single point of contact for clients. It distributed incoming application traffic across multiple EC2 instances in multiple AZs, increasing the fault tolerance and scalability of the application.
- **Auto Scaling Group (ASG)**: Automated the process of scaling the Application Tier. It was configured to maintain a minimum number of running EC2 instances and to automatically launch or terminate instances based on a set of defined Step Scaling policies.
- **Launch Template**: Served as a blueprint for the EC2 instances launched by the Auto Scaling Group. It specified the Amazon Machine Image (AMI), instance type, security groups, and a user data script to bootstrap the instances with the necessary software upon launch.

- **Amazon RDS (Relational Database Service)**: Provided a fully managed, Multi-AZ PostgreSQL database. Using RDS offloaded administrative tasks such as patching, backups, and failover management, allowing the focus to remain on the application.
- **Amazon S3 (Simple Storage Service)**: Provided highly durable, scalable object storage for static assets and backups. In this architecture, S3 is used to host static website content (images).
- **Amazon CloudWatch**: Acted as the monitoring and observability service. It was used to collect performance metrics (specifically `CPUUtilization`) from the EC2 instances. CloudWatch Alarms were configured to watch these metrics and trigger the Step Scaling policies when specific thresholds were breached.
- **AWS Security Groups**: Acted as a stateful virtual firewall for the instances and database. They were configured to enforce the principle of least privilege, ensuring that the ALB could only talk to the web servers on port 80, and the web servers could only talk to the database on port 3306.

3 Implementation

3.1 Setting Up the Network and Security

First, we created a secure and isolated network environment for the application.

3.1.1 Virtual Private Cloud

We created a Virtual Private Cloud (VPC) in 2 Availability Zones (AZ) with 2 Public Subnets and 2 Private Subnets. We also placed a NAT gateway in 1 AZ.

3.1.2 Security Groups

We defined five SGs to enforce least-privilege access.

Security Group	Inbound Rule	Source	Purpose
`tester-sg`	SSH (22)	My IP	Admin access to tester host
`bastion-sg`	SSH (22)	My IP	Admin access to bastion host
`alb-sg`	HTTP (80)	0.0.0.0/0	Public web access to ALB
`web-sg`	HTTP (80) SSH (22)	`alb-sg` `bastion-sg`	Web traffic from ALB Admin SSH from bastion host
`db-sg`	PostgreSQL (5432)	`web-sg`	DB access from web tier

3.2 Database, Bastion Host and Tester Host

Then from the RDS console we deployed a `PostgreSQL` database on a `db.t3.micro` instance. We made it available in both Availability Zones for maximum reliability.

We launched another EC2 instance, the `Bastion-Host`, which we will use as an SSH gateway. Since it does not require that many resources, we will host it on a `t3.micro` instance.

The last EC2 instance we manually launch is `Tester-host`, which we will use to carry out our performance tests later. Due to performance concerns we will explain later, this instance will need to be a `t3.small` one.

3.3 S3

An Amazon S3 bucket was provisioned via the AWS Management Console for storing user-uploaded images. Then in the instance template we added the IAM instance role "LabInstanceProfile".

3.4 Web Application Infrastructure

Now we need to set up the web tier, load balancer, and auto scaling.

First we created our **EC2 Launch Template**, which is a `t3.micro` EC2 instance running Amazon Linux 2. We will assign this to the `web-sg` security group. The `User Data` section is crucial, as it automates the instance initialization by pulling application code from GitHub and configuring the runtime environment. Then it creates a system service and runs our web application using `Gunicorn`.

Next we created a `Target Group` for our VPC, called `project-tg` of type `Instances` using Protocol/Port: HTTP/80.

Then we set up our internet facing Application Load Balancer; it is present in both public subnets of our VPC and is assigned to the `alb-sg` security group. Its role is to listen to `HTTP port 80` and forward it to the `project-tg` target group we just created.

3.4.1 Auto Scaling Group

Finally we created an Auto Scaling Group (ASG) in our VPC with our launch template (`project-web-template`). This ASG is active in both private subnets of our VPC and is attached to the `project-tg` target group. Here we set:

- Desired Instance Number = 2
- Minimum Instance Number = 2
- Maximum Instance Number = 6.

3.4.2 Finalizing Scaling Policies

To complete our setup we added CPU-based step scaling to respond to load. First we created two CloudWatch Alarms, which will be used by our scaling policies:

- `cpu-high-alarm`: `CPUUtilization` of `project-asg` $\geq 65\%$ for 1 minute.
- `cpu-low-alarm`: `CPUUtilization` of `project-asg` $\leq 30\%$ for 1 minute.

In the Auto-Scaling Group we added these two policies:

1. `scale-out`: when `cpu-high-alarm` is true, we:
    - add 1 capacity unit when $65 \leq$ `CPUUtilization` < 80
    - add 3 capacity units when $80 \leq$ `CPUUtilization`

2. `scale-in`: when `cpu-low-alarm` is true, we:

    - remove 1 capacity unit when $30 \geq$ `CPUUtilization` > 15
    - remove 3 capacity units when $15 \geq$ `CPUUtilization`

3.5 Addendum on EC2 instance type

At first we chose the `t2.micro` EC2 instance type. However after some tests, we could not guarantee a good level of service while the workload increased. Even while testing with around 1300 Transactions per second, JMeter detected 0,05% failures and the CPU always reached maximum usage (around 95%).

To avoid providing a low quality of service, we chose to use the `t3.micro` instance, which guarantees better performance and even costs less than `t2.micro`. The difference in price varies according to demand, but `t2.micro`'s cost is usually 0.0116$ per hour while `t3.micro`'s is around 0.0104$ per hour.

4 Performance Evaluation

4.1 JMeter Tests

To evaluate the performance of our system, we decided to use `Apache JMeter` tests. In particular, we used the `UltimateThreadGroup` plugin for maximum customizability of tests. All performance tests were executed from an EC2 instance within the same AWS region to avoid external network interference. JMeter was used to simulate concurrent user traffic, with a distribution of user roles representative of real-world usage patterns. Metrics were collected using Amazon CloudWatch and JMeter reports. Since `JMeter` has strict resource requirements, we will run them on a `t3.small` EC2 instance (`tester-host`). We used `JMeter Throughput Controllers` to simulate user behavior:

Sampler Step	**% of Users**
Viewers	70%
Text Posters	20%
Image Posters	10%

Here are the different actions for each category of user:

1. **Viewers:** HTTP `GET` to `/`, it simply visits the homepage.
2. **Text Posters:** First they login with a HTTP `POST` to `/login`. Then they perform a HTTP `POST` to `/create` with a simple string for both title and content.
3. **Create Image Post:** First they login with a HTTP `POST` to `/login`. Then they perform a HTTP `POST` to `/create`. Differently from before, they also have an image to post.

To ensure that we have a mix of different requests, we used text of different sizes and images of different sizes. In particular:

- **Text Posters**: Post a random string, created using the `RandomString()` function; this string has random size between 100 and 2000 characters chosen using the `Random(100,2000)` function.
- **Image Posters**: Post an image chosen between the three we supplied: `small.jpg`, of size 5KB, `medium.jpg`, of size 250KB and `big.jpg`, of size 500KB.

We also employed Constant Throughput Timers for our tests. We created two different tests: a **lighter** one and a **heavier** one. The value of the timer is different for each case obviously.

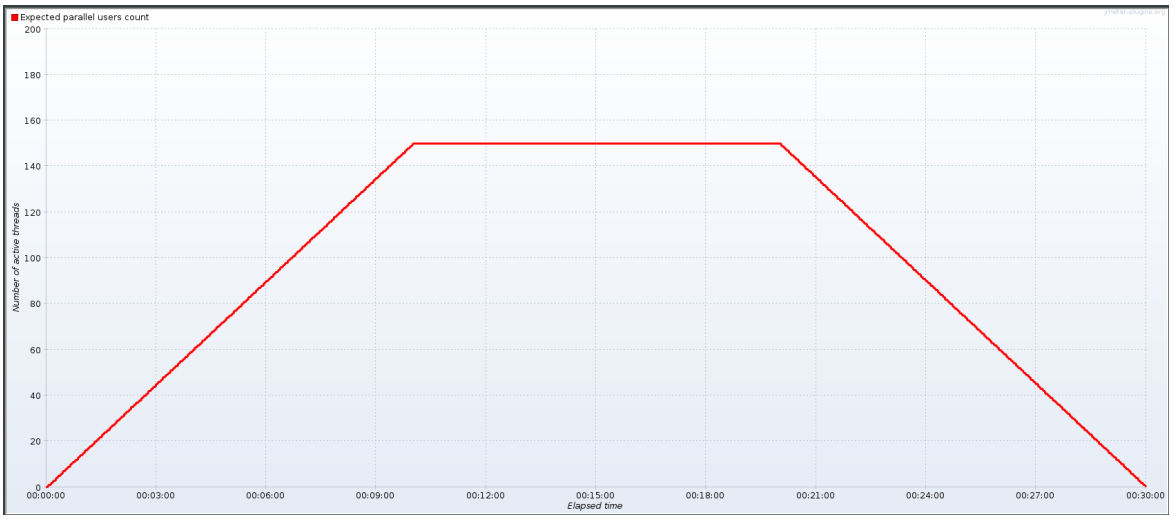
4.1.1 Heavy Test

The test lasts for 30 minutes and is structured like this:

# Threads	Delay (Δ_0) [s]	Ramp-Up (R) [s]	Hold (H) [s]	Ramp-Down (D) [s]
30	0	120	1080	120
30	120	120	1080	120
30	240	120	1080	120
30	360	120	1080	120
30	480	120	1080	120

There are five 30-thread groups start every 120 s; each ramps up over 120 s for the first 600 s, holds for 1080 s, then ramps down over 120 s.

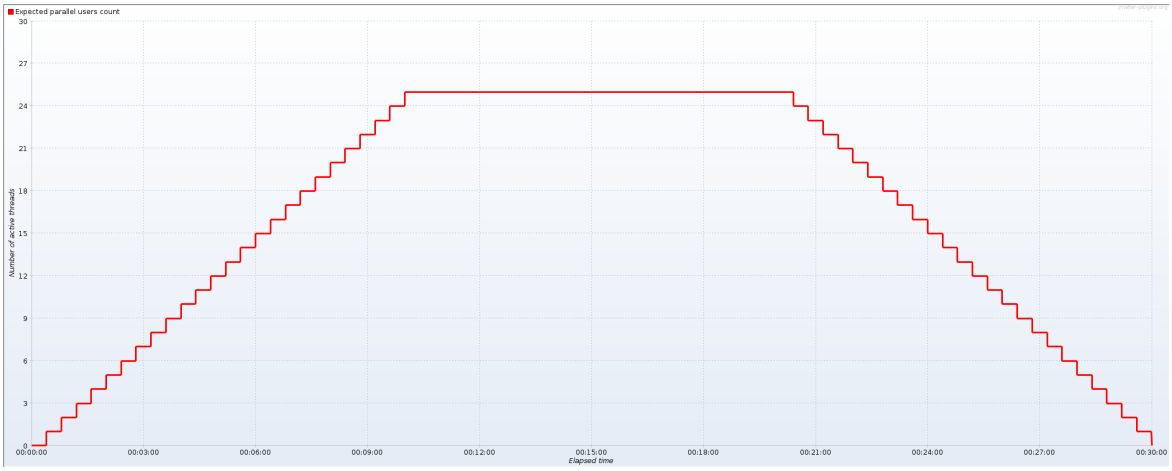
Here is the expected parallel users count computed by the `JMeter` GUI:



From Table 1 and the `JMeter` provided expected user count you can clearly see that there are distinct phases of Warm-Up, Ramp-Up which last 600 seconds then a Steady phase of 600 seconds and finally the Ramp-Down phase that lasts 600 seconds (it's hard to see from this image but it's not linear, there are multiple steps). In this case the constant throughput timer is aimed for 2000 TX/s.

4.1.2 Light Test

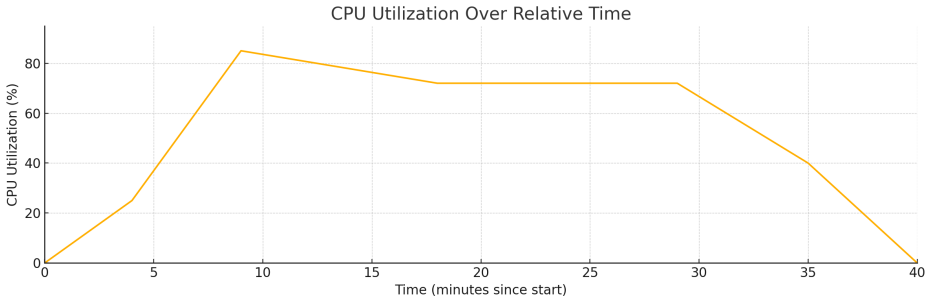
The light test uses the same statistics as the heavy test we just described in table 1, but instead of 30 threads per step it only dispatches 5 threads per step for a total of 25. The constant throughput timer is set to aim for 1000 TX/s.



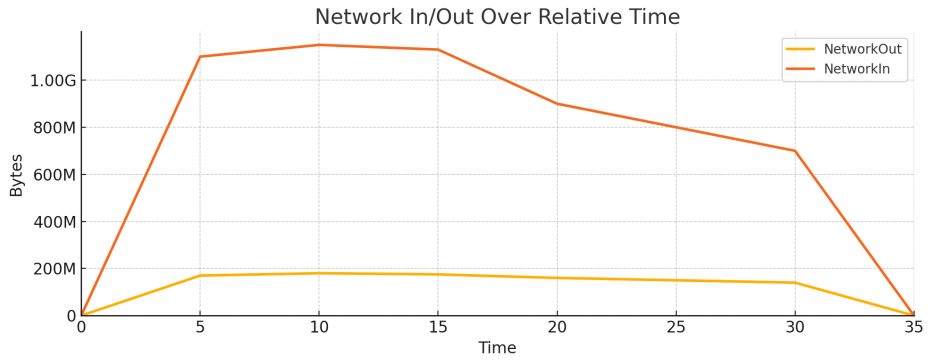
4.2 Heavy Load Test: Performance Evaluation

4.2.1 EC2 Metrics

- **CPUUtilization**, during the ramp-up phase (the first 600 seconds), CPU utilization rapidly increases as the number of active users grows. It reaches its peak during the steady phase. As the load ramps down and terminates (the final 600 seconds), CPU utilization decreases significantly.



- **NetworkIn** and **NetworkOut**, as the number of active users increases during the ramp-up phase, network traffic (both incoming and outgoing) rises proportionally. The incoming traffic is mainly made of the images that the users upload.



4.2.2 Autoscaling Events and Instance Scaling

As we already said in chapter 3, GroupMinSize is 2 and GroupMaxSize is 6. As you can see the scale out and scale in policies are employed to respond to the growing demand.

The activity log shows how all four scaling policies are used: aggressive scale-out, moderate scale-out and then aggressive scale-in, moderate scale-in at the end of the test:

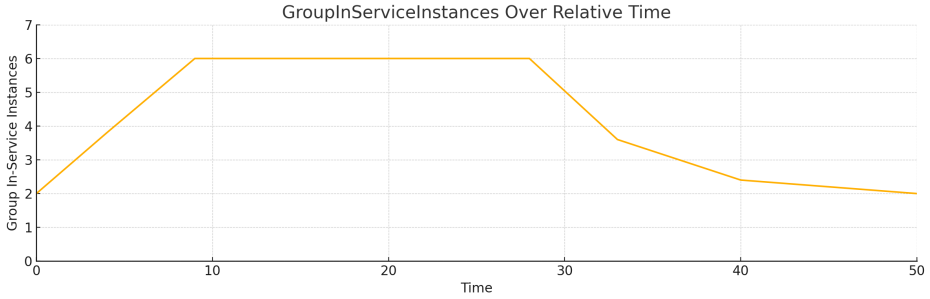
Activity history (10)

Filter activity history

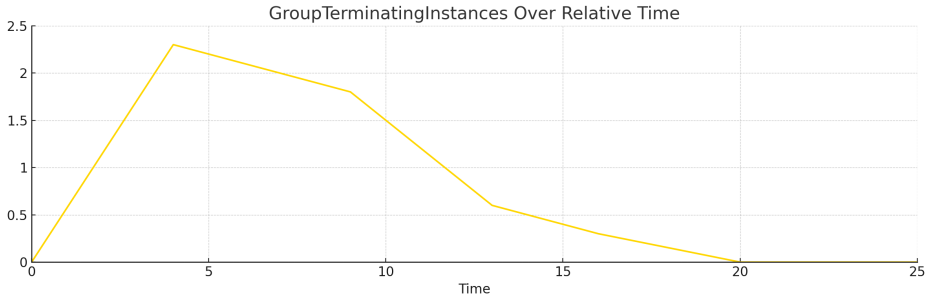
Status	Description	Cause	Start time	End time
Successful	Terminating EC2 instance: i-09d62dcd59b7cad2a	At 2025-07-11T15:32:38Z a monitor alarm cpu-low-alarm in state ALARM triggered policy scale-in changing the desired capacity from 3 to 2. At 2025-07-11T15:32:44Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 3 to 2. At 2025-07-11T15:32:44Z instance i-09d62dcd59b7cad2a was selected for termination.	2025 July 11, 05:32:44 PM +02:00	2025 July 11, 05:38:27 PM +02:00
Successful	Terminating EC2 instance: i-06b32e026a79f9824	At 2025-07-11T15:26:38Z a monitor alarm cpu-low-alarm in state ALARM triggered policy scale-in changing the desired capacity from 6 to 3. At 2025-07-11T15:26:45Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 6 to 3. At 2025-07-11T15:26:45Z instance i-03813c2017b850410 was selected for termination. At 2025-07-11T15:26:45Z instance i-09353f76310c9752b was selected for termination. At 2025-07-11T15:26:45Z instance i-06b32e026a79f9824 was selected for termination.	2025 July 11, 05:26:45 PM +02:00	2025 July 11, 05:32:48 PM +02:00
Successful	Terminating EC2 instance: i-09353f76310c9752b	At 2025-07-11T15:26:38Z a monitor alarm cpu-low-alarm in state ALARM triggered policy scale-in changing the desired capacity from 6 to 3. At 2025-07-11T15:26:45Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 6 to 3. At 2025-07-11T15:26:45Z instance i-03813c2017b850410 was selected for termination. At 2025-07-11T15:26:45Z instance i-09353f76310c9752b was selected for termination. At 2025-07-11T15:26:45Z instance i-06b32e026a79f9824 was selected for termination.	2025 July 11, 05:26:45 PM +02:00	2025 July 11, 05:32:48 PM +02:00
Successful	Terminating EC2 instance: i-03813c2017b850410	At 2025-07-11T15:26:38Z a monitor alarm cpu-low-alarm in state ALARM triggered policy scale-in changing the desired capacity from 6 to 3. At 2025-07-11T15:26:45Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 6 to 3. At 2025-07-11T15:26:45Z instance i-03813c2017b850410 was selected for termination. At 2025-07-11T15:26:45Z instance i-09353f76310c9752b was selected for termination. At 2025-07-11T15:26:45Z instance i-06b32e026a79f9824 was selected for termination.	2025 July 11, 05:26:45 PM +02:00	2025 July 11, 05:32:08 PM +02:00
Successful	Launching a new EC2 instance: i-09d62dcd59b7cad2a	At 2025-07-11T14:59:45Z a monitor alarm cpu-high-alarm in state ALARM triggered policy scale-out changing the desired capacity from 5 to 6. At 2025-07-11T14:59:49Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 5 to 6.	2025 July 11, 04:59:51 PM +02:00	2025 July 11, 05:02:08 PM +02:00
Successful	Launching a new EC2 instance: i-09353f76310c9752b	At 2025-07-11T14:56:45Z a monitor alarm cpu-high-alarm in state ALARM triggered policy scale-out changing the desired capacity from 2 to 5. At 2025-07-11T14:56:54Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 2 to 5.	2025 July 11, 04:56:56 PM +02:00	2025 July 11, 04:59:01 PM +02:00
Successful	Launching a new EC2 instance: i-06b32e026a79f9824	At 2025-07-11T14:56:45Z a monitor alarm cpu-high-alarm in state ALARM triggered policy scale-out changing the desired capacity from 2 to 5. At 2025-07-11T14:56:54Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 2 to 5.	2025 July 11, 04:56:56 PM +02:00	2025 July 11, 04:59:01 PM +02:00
Successful	Launching a new EC2 instance: i-03813c2017b850410	At 2025-07-11T14:56:45Z a monitor alarm cpu-high-alarm in state ALARM triggered policy scale-out changing the desired capacity from 2 to 5. At 2025-07-11T14:56:54Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 2 to 5.	2025 July 11, 04:56:56 PM +02:00	2025 July 11, 04:59:01 PM +02:00
Successful	Launching a new EC2 instance: i-05086e96a66115019	At 2025-07-11T14:25:30Z a user request created an AutoScalingGroup changing the desired capacity from 0 to 2. At 2025-07-11T14:25:33Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 0 to 2.	2025 July 11, 04:25:35 PM +02:00	2025 July 11, 04:25:41 PM +02:00
Successful	Launching a new EC2 instance: i-0c6ee668896486bde	At 2025-07-11T14:25:30Z a user request created an AutoScalingGroup changing the desired capacity from 0 to 2. At 2025-07-11T14:25:33Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 0 to 2.	2025 July 11, 04:25:35 PM +02:00	2025 July 11, 04:25:41 PM +02:00

There are also other interesting metrics for autoscaling:

- **GroupInServiceInstances**, the instances rapidly scale out. As the number of requests drops at the end of the test, the ASG scales in, decreasing the number of active instances, although the scale-in phase appears slower than the scale-out phase.



- **GroupPendingInstances** and **GroupStandbyInstances** are always at 0
- **GroupTerminatingInstances**, as the load decreases during the ramp-down phase, the ASG begins to terminate instances that are no longer needed to meet the demand. The graph shows a peak in terminating instances after the peak load period, reflecting the ASG's scale-in activity.



4.2.3 User-side Metrics

Response times, Error Rates and Throughput by request provided by the `JMeter` report that is automatically generated at the end of a test:

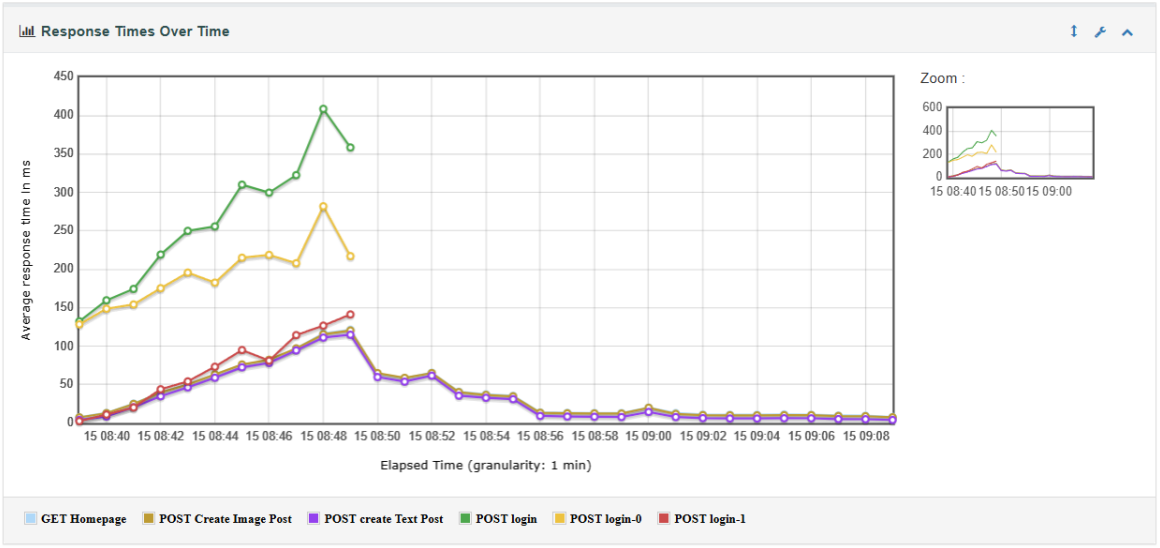
Label	Samples	FAIL	Err%	Avg (ms)	Min (ms)	Max (ms)	Med (ms)	90% (ms)	95% (ms)	99% (ms)	Tx/s	Recv (KB/s)	Sent (KB/s)
Total	2 696 261	0	0.00	31.1	1	1 082	6.0	9.0	10.0	12.0	1 501	7 913	55 702
GET Homepage	1 887 130	0	0.00	31.9	3	855	8.0	10.0	10.0	14.0	1 051	5 385	388
POST Create Image Post	404 332	0	0.00	31.4	2	1 082	8.0	12.0	14.0	20.0	225	1 264	54 977
POST create Text Post	404 349	0	0.00	27.2	1	825	4.0	6.0	8.0	12.0	225	1 264	344
POST login	150	0	0.00	271.4	112	696	252.5	424.6	512.0	675.1	0.25	1	0.16
POST login-0	150	0	0.00	198.0	109	468	163.0	312.7	378.5	439.4	0.25	0.16	0.07
POST login-1	150	0	0.00	73.2	1	374	41.5	195.9	219.9	356.7	0.25	1	0.09

Table 2: Heavy Test Performance Results

The `JMeter` report data in this table suggests that the system has excellent error handling, since there are no errors across 2.6 million requests. The login flow is the slowest path.

To investigate the login path, we can analyze the *Response Times Over Time* table. Here we can see that the `POST Login` requests exhibit the highest average

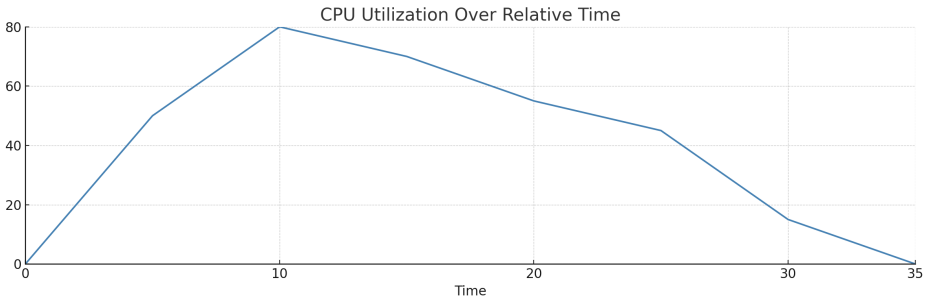
response time, likely due to the `flask-login`-based password hashing implemented for user authentication. This aligns with expectations, as cryptographic operations are CPU-bound and not easily parallelizable, making them more sensitive to load:



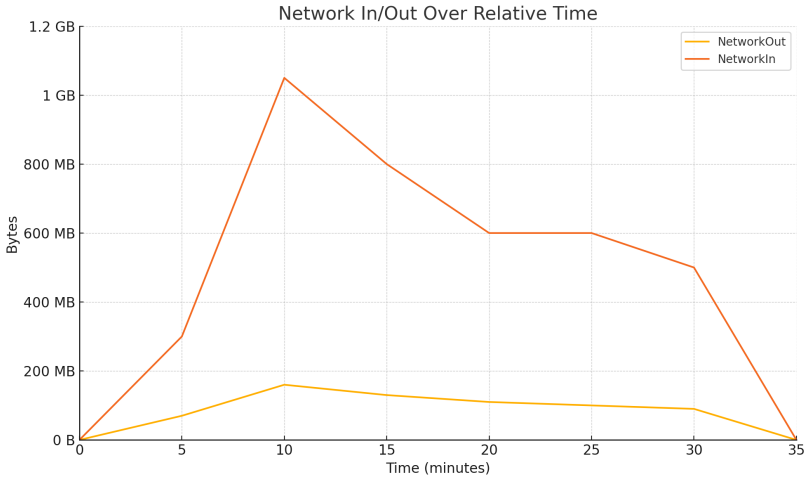
4.3 Performance Testing and Evaluation - Light Test

4.3.1 EC2 Metrics

- **CPUUtilization**, similar to the heavy test, CPU utilization increases during the ramp-up phase, reaches a peak during the steady phase, and then decreases (a bit slower than the scale out) as the load ramps down.



- **NetworkIn** and **NetworkOut**, the traffic volumes are significantly lower than those observed during the heavy test due to the fewer concurrent users:



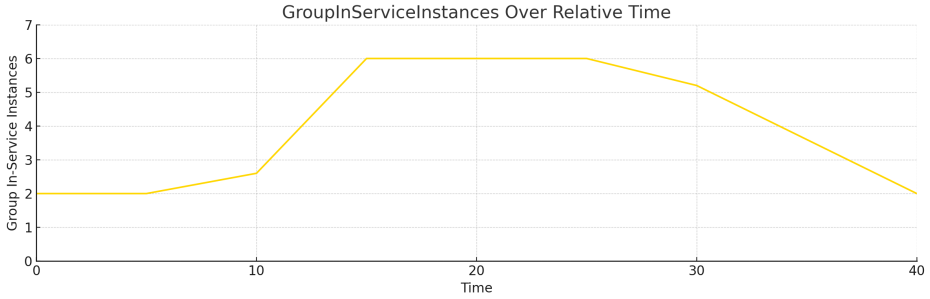
4.3.2 Autoscaling Metrics

The activity log shows that for this test the moderate scale-out rule is used first. Then the aggressive scale-out rule is used. Then when the Steady phase of the test ends the scale-in rules are employed by the system:

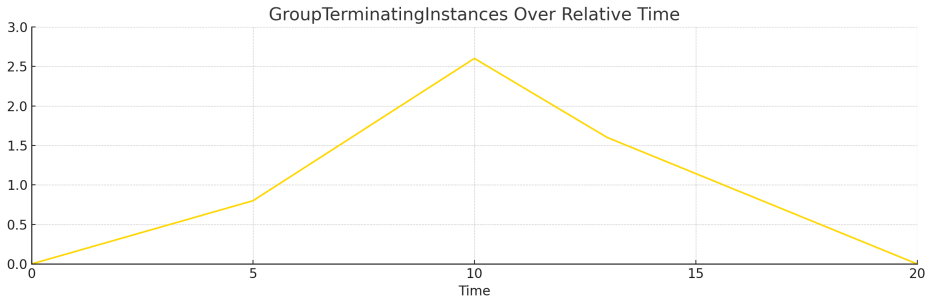
Status	Description	Cause	Start time	End time
Successful	Terminating EC2 instance: i-01cca3563965211f7	At 2025-07-11T17:22:38Z a monitor alarm cpu-low-alarm in state ALARM triggered policy scale-in changing the desired capacity from 3 to 2. At 2025-07-11T17:22:39Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 3 to 2. At 2025-07-11T17:22:40Z instance i-01cca3563965211f7 was selected for termination.	2025 July 11, 07:22:40 PM +02:00	2025 July 11, 07:28:22 PM +02:00
Successful	Terminating EC2 instance: i-01424ea6c9f65d45b	At 2025-07-11T17:21:38Z a monitor alarm cpu-low-alarm in state ALARM triggered policy scale-in changing the desired capacity from 5 to 3. At 2025-07-11T17:21:43Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 5 to 3. At 2025-07-11T17:21:43Z instance i-006ed4168255420be was selected for termination. At 2025-07-11T17:21:43Z instance i-01424ea6c9f65d45b was selected for termination.	2025 July 11, 07:21:43 PM +02:00	2025 July 11, 07:28:06 PM +02:00
Successful	Terminating EC2 instance: i-006ed4168255420be	At 2025-07-11T17:21:38Z a monitor alarm cpu-low-alarm in state ALARM triggered policy scale-in changing the desired capacity from 5 to 3. At 2025-07-11T17:21:43Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 5 to 3. At 2025-07-11T17:21:43Z instance i-006ed4168255420be was selected for termination. At 2025-07-11T17:21:43Z instance i-01424ea6c9f65d45b was selected for termination.	2025 July 11, 07:21:43 PM +02:00	2025 July 11, 07:27:46 PM +02:00
Successful	Terminating EC2 instance: i-0319b8554c8b08e3c	At 2025-07-11T17:16:38Z a monitor alarm cpu-low-alarm in state ALARM triggered policy scale-in changing the desired capacity from 6 to 5. At 2025-07-11T17:16:49Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 6 to 5. At 2025-07-11T17:16:49Z instance i-0319b8554c8b08e3c was selected for termination.	2025 July 11, 07:16:49 PM +02:00	2025 July 11, 07:22:31 PM +02:00
Successful	Launching a new EC2 instance: i-062490e1003f6df1b	At 2025-07-11T16:59:45Z a monitor alarm cpu-high-alarm in state ALARM triggered policy scale-out changing the desired capacity from 3 to 6. At 2025-07-11T16:59:49Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 3 to 6.	2025 July 11, 06:59:51 PM +02:00	2025 July 11, 07:01:58 PM +02:00
Successful	Launching a new EC2 instance: i-0f4e93926b0fad12d	At 2025-07-11T16:59:45Z a monitor alarm cpu-high-alarm in state ALARM triggered policy scale-out changing the desired capacity from 3 to 6. At 2025-07-11T16:59:49Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 3 to 6.	2025 July 11, 06:59:51 PM +02:00	2025 July 11, 07:01:58 PM +02:00
Successful	Launching a new EC2 instance: i-01cca3563965211f7	At 2025-07-11T16:59:45Z a monitor alarm cpu-high-alarm in state ALARM triggered policy scale-out changing the desired capacity from 3 to 6. At 2025-07-11T16:59:49Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 3 to 6.	2025 July 11, 06:59:51 PM +02:00	2025 July 11, 07:01:56 PM +02:00
Successful	Launching a new EC2 instance: i-01424ea6c9f65d45b	At 2025-07-11T16:56:45Z a monitor alarm cpu-high-alarm in state ALARM triggered policy scale-out changing the desired capacity from 2 to 3. At 2025-07-11T16:56:54Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 2 to 3.	2025 July 11, 06:56:56 PM +02:00	2025 July 11, 06:59:03 PM +02:00

We also want to analyze the other autoscaling metrics:

- **GroupInServiceInstances**, the graph shows the ASG scaling out moderately initially, and then aggressively, in response to the light test load. When the steady phase ends, the scale-in rules are employed, reducing the number of active instances.



- **GroupPendingInstances** and **GroupStandbyInstances** are always at 0
- **GroupTerminatingInstances**, this graph shows the number of instances being terminated. This reflects the ASG's scale-in process after the load has decreased, showing a peak in terminations during the ramp-down phase:



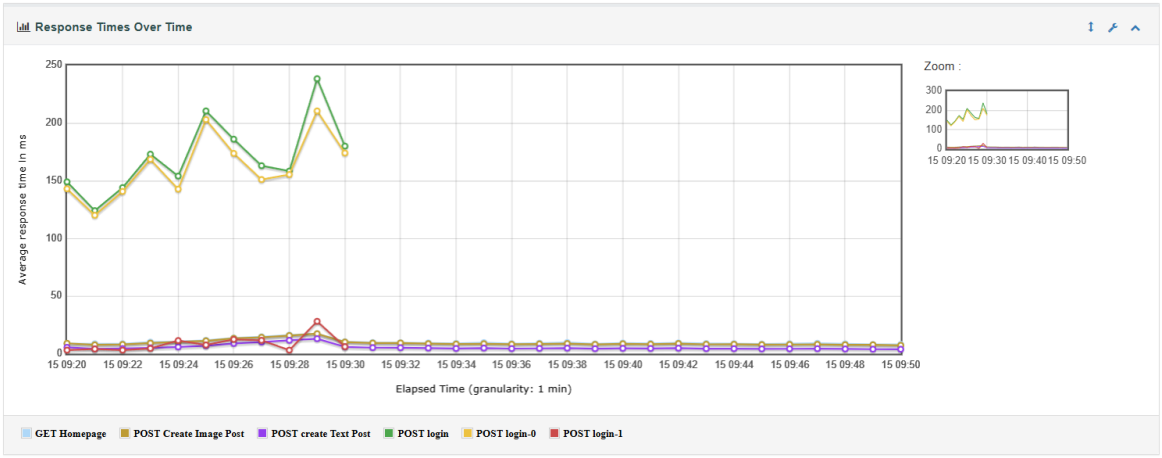
4.3.3 User Side Metrics

Response times, Error Rates and Throughput by request provided by the `JMeter` report that is automatically generated at the end of a test. In this case the number of transactions per second is halved compared to the heavy test; there are no errors too:

Label	Samples	FAIL	Err%	Avg (ms)	Min (ms)	Max (ms)	Med (ms)	90% (ms)	95% (ms)	99% (ms)	Tx/s	Recv (KB/s)	Sent (KB/s)
Total	1 539 663	0	0.00	9.3	2	1 078	8.0	9.0	10.0	13.0	867	4 570	32 251
GET Homepage	1 077 722	0	0.00	10.1	4	1 065	8.0	9.0	10.0	13.0	607	3 110	224
POST Create Image Post	230 930	0	0.00	9.5	3	1 034	7.0	11.0	12.0	17.0	130	730	31 833
POST create Text Post	230 936	0	0.00	5.6	2	1 078	4.0	5.0	6.0	9.0	130	730	198
POST login	25	0	0.00	170.2	113	260	166.0	238.4	253.7	260.0	0.04	0.26	0.03
POST login-0	25	0	0.00	161.4	110	248	158.0	220.6	241.4	248.0	0.04	0.03	0.01
POST login-1	25	0	0.00	8.6	2	34	4.0	20.2	30.4	34.0	0.04	0.23	0.02

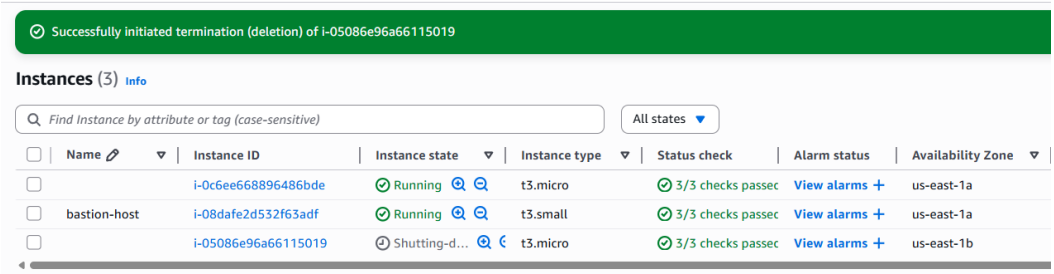
Table 3: Light Test Performance Results

As with the heavy test, we can see that the `POST Login` requests are computational hotspots in this case as well:

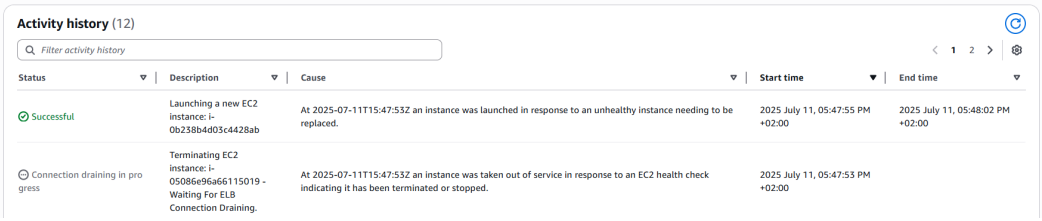


4.4 Availability Tests

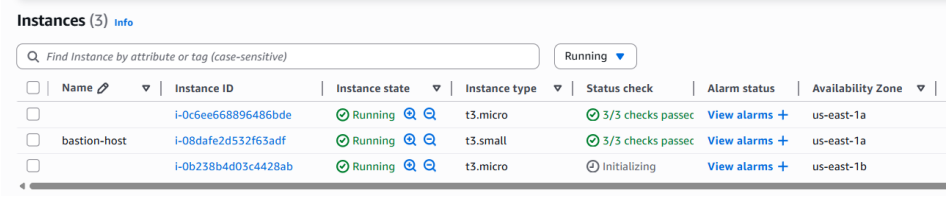
To test availability we completed this simple test:



(a) Killed an EC2 instance.



(b) The log shows that a new EC2 instance is launched in response to the killed one.



(c) Finally, on the instances tab a new healthy EC2 instance appears.

5 Conclusions

To have a rough estimate of the cost for our system, we used AWS Pricing Calculator to estimate our costs. Interestingly enough, most of the cost is due to the PostgreSQL relational database:

Service Name	**Upfront cost**	**Monthly cost**	**Description**
Amazon EC2	0.00 USD	8.22 USD	project-web
Amazon RDS for PostgreSQL	0.00 USD	146.66 USD	project-db
Amazon Virtual Private Cloud (VPC)	0.00 USD	32.85 USD	project-vpc
Amazon EC2	0.00 USD	3.80 USD	bastion-host
Elastic Load Balancing	0.00 USD	17.25 USD	project-alb
Amazon Simple Storage Service (S3)	0.03 USD	2.30 USD	project-s3

Table 4: AWS service costs and descriptions

In summary, the project successfully demonstrated the deployment of a scalable, fault-tolerant blog application using AWS services. The use of a three-tier architecture combined with Auto Scaling and multi-AZ deployment ensured both high availability and dynamic scalability under varying workloads. Performance evaluations confirmed that the system maintained low error rates and acceptable response times, even under heavy load. While the PostgreSQL database emerged as the primary cost driver, the architecture remains cost-effective for educational and small-scale production scenarios. This project highlights the practical considerations and trade-offs involved in deploying cloud-native applications within constrained environments.