SAPIENZA
UNIVERSITÀ DI ROMA

Cloud Computing Report

Authors: *Student ID:*

Andrea Fede **1942562**
fede.1942562@studenti.uniroma1.it

Chiara Iannicelli **1957045**
iannicelli.1957045@studenti.uniroma1.it

Pietro Colaguori **1936709**
colaguori.1936709@studenti.uniroma1.it

Academic Year 2023/24

Contents

1 Introduction

Our project is focused on the development and deployment of a highly-available and scalable social media web application on the cloud, named **CloudyPics**. This web application aims to provide a robust platform where users can interact with each other and share photos of special moments seamlessly.
Key features of the web application include the ability for users to upload, like, and comment on pictures. Additionally, users have the option to follow other users to keep up with their content and ban users if necessary to maintain a safe and comfortable online environment.
The application also incorporates a dynamic stream feature, which displays images published by users whom the current user follows. This stream is designed to provide a real-time, personalized feed of updates, ensuring that users stay engaged with the latest content from their connections.
By leveraging cloud technologies, our application ensures high availability, scalability, and reliability, accommodating varying traffic loads and providing a seamless user experience.

2 Design

The diagram in Figure 1 presents our architecture, highlighting the technologies used, including programming languages and Amazon Web Services (AWS) products. This offers a detailed overview of the relationships between the various components. In the following sections, we will thoroughly examine each aspect.

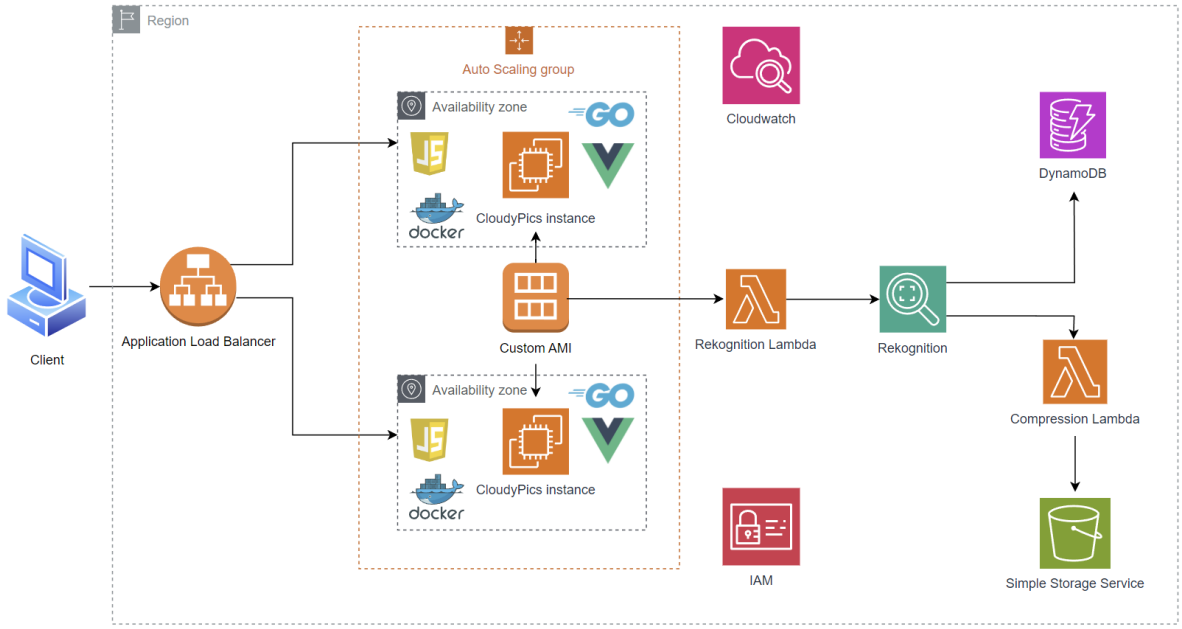

Figure 1: General overview of the architecture of our web application

2.1 Technologies used

To realize our project, we employed various technologies, including:

- **Docker** is a software platform that enables rapid building, testing, and deployment of applications. It packages software into standardized units called containers, which include all necessary components such as libraries, system tools, code and runtime. Docker ensures that applications run consistently across different environments. Deploying Docker on AWS offers developers a reliable, cost-effective solution to build, ship and manage distributed applications at any scale.

- **Amazon Simple Storage Service (Amazon S3)** is an object storage service renowned for its scalability, data availability, security and performance. It caters to customers of all sizes and industries, providing storage solutions for various use cases including websites, mobile applications, backup and restore, archiving, enterprise applications, IoT devices and big data analytics. Amazon S3 offers user-friendly management features, enabling you to organize your data and configure detailed access controls to meet business, organizational and compliance requirements.

- **Apache JMeter** is a tool for testing the performance of both static and dynamic resources and web dynamic applications. It simulates heavy loads on servers, groups of servers, networks, or objects to test their strength and analyze overall performance under different load conditions.

- **Amazon CloudWatch** is a monitoring and observability service designed for DevOps engineers, developers, site reliability engineers (SREs), and IT managers. CloudWatch provides actionable insights and data to monitor applications, respond to system-wide performance changes, optimize resource utilization, and gain a unified view of operational health. It collects logs, metrics, and events, offering a comprehensive view of AWS resources, applications, and services running on AWS and on-premises servers. CloudWatch helps in detecting anomalous behaviors, setting alarms, visualizing logs and metrics, automating actions, troubleshooting issues, and discovering insights to maintain smooth application operations.

- **Amazon DynamoDB** is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. It allows users to offload the administrative burdens of operating and scaling a distributed database, ensuring high availability and data durability. DynamoDB is designed to handle large-scale applications, providing consistent, rapid response times for reads and writes. It is ideal for web, mobile, gaming, ad tech, IoT, and many other applications that require low-latency data access.

- **Amazon Elastic Compute Cloud (Amazon EC2)** is a service that provides resizable compute capacity in the cloud. It is designed to make web-scale cloud computing easier for developers by offering a simple web service interface to obtain and configure capacity with minimal friction. Amazon EC2 offers complete control over your computing resources and lets you run on Amazon's proven computing environment.

- **AWS Lambda** is a serverless compute service that lets you run code without provisioning or managing servers. You pay only for the compute time you consume. With Lambda, you can run code for virtually any type of application or backend service, all with zero administration. Just upload your code, and Lambda takes care of everything required to run and scale your code with high availability.
- **GoLang**, also known as Go, is a statically typed, compiled programming language designed at Google. It is known for its simplicity, efficiency, and strong performance. GoLang is particularly well-suited for building scalable and high-performance applications, making it a popular choice for cloud-based services, distributed systems, and large-scale applications.
- **JavaScript** is a versatile, high-level programming language primarily used for frontend development. **Vue.js** is a progressive JavaScript framework used for building user interfaces. It is designed to be incrementally adoptable and focuses on the view layer only, making it easy to integrate with other projects and libraries. Vue.js is known for its simplicity and flexibility, enabling developers to create sophisticated single-page applications with ease.

2.2 Overview of CloudyPics

CloudyPics is a multi-tier web application consisting of a lightweight frontend developed in JavaScript using the Vue.js framework, a backend written in Golang that handles the business logic, and a NoSQL database, DynamoDB, for storing user and photo data. Additionally, our application utilizes AWS S3 storage buckets for storing user-uploaded pictures and employs serverless Lambda functions to perform content sanitization (to prevent the upload of violent, gore, or explicit pornographic images) and image compression (crucial for cost savings by reducing storage space on S3).

The communication between the backend and frontend of our web application adheres to a REST API specification. This specification defines all the requests and responses that the API supports, including endpoints, methods (`GET`, `POST`, `PUT`, `DELETE`, `PATCH`), request parameters, headers, and response formats. Serving as a contract between the frontend and backend, the API specification ensures consistent and predictable interactions. It details authentication, error handling, and data schemas, enabling developers to integrate and interact with the API efficiently and accurately. We used OpenAPI and Swagger to create and document these specifications, as partially illustrated in Figure 2.

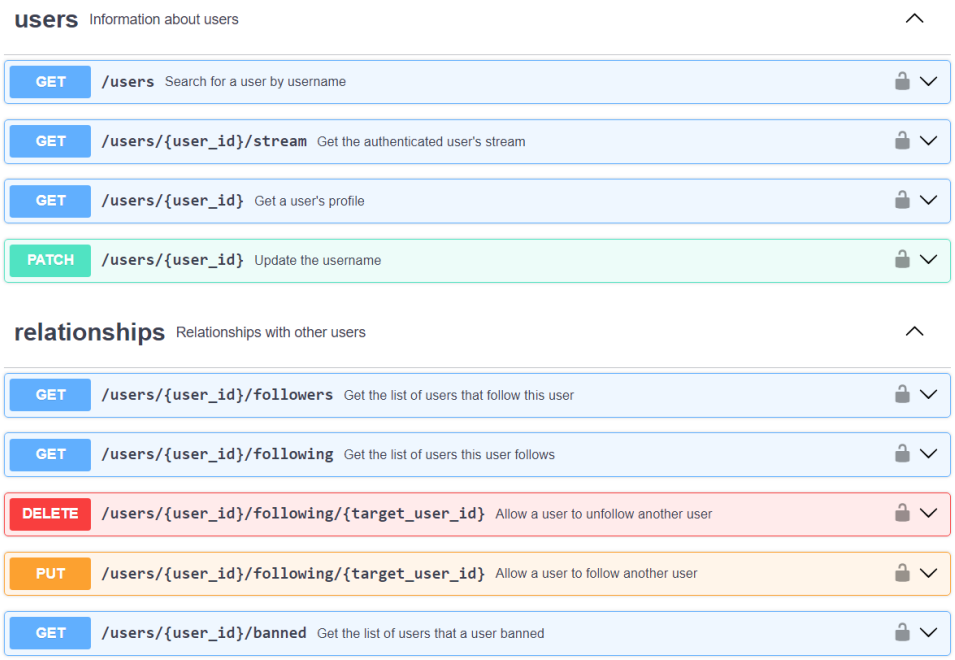

Figure 2: OpenAI document on the Swagger editor: methods and endpoints related to users and relationship between users

Navigating to our web application, we are welcomed by the login page, as shown in Figure 3.

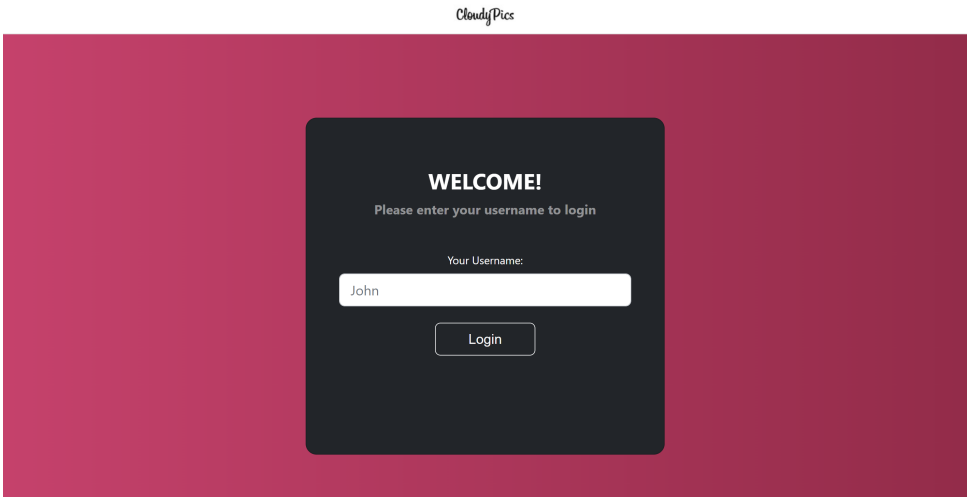

Figure 3: Login page

Since the security of our web application is not a primary goal for this project, we opted not to implement a third-party solution for secure login or multi-step user authentication.

Instead, we simplified the login process to require only a username. If the username already exists, the user's ID is returned and must be included as a Bearer token in the Authorization header of every subsequent HTTP request. If the username does not exist in the Users table of the database, a new user entry is created, and a new ID is generated and returned in the response.

After logging in as an existing user, *Oliver*, we can navigate to his profile page. As shown in Figure 4, the user profile page displays the current user's username, a button to edit the username, a button to upload a new photo, and various user statistics: the number of photos uploaded (displayed in a grid format below), the number of followers, the number of users the current user is following, and the users banned by the current user.

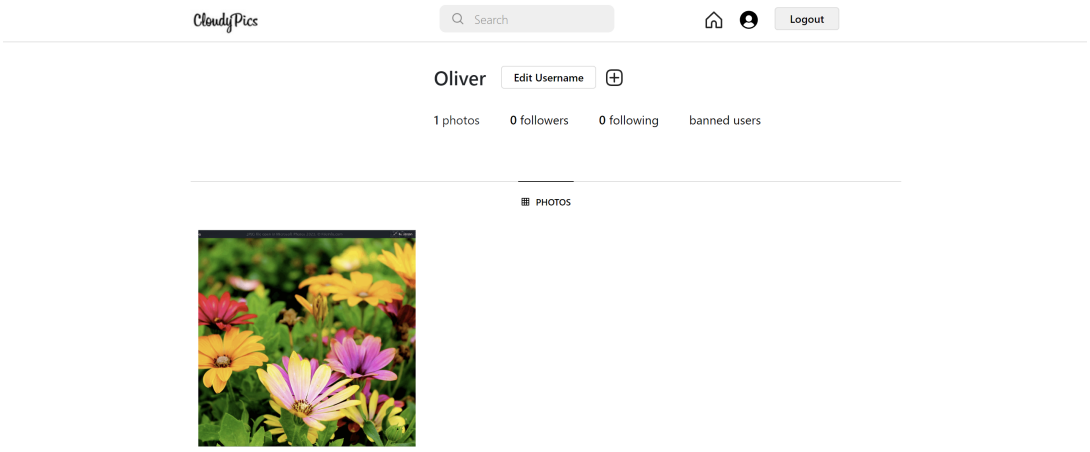

Figure 4: User profile page

Oliver's home page is currently empty because he is not following any other users. This page is intended to display all the photos published by the users he follows.

By entering a username in the search bar on the webpage's navbar, we can search for a specific user. If we just press enter without writing anything, we can see a list of all registered users, as shown in Figure 5.

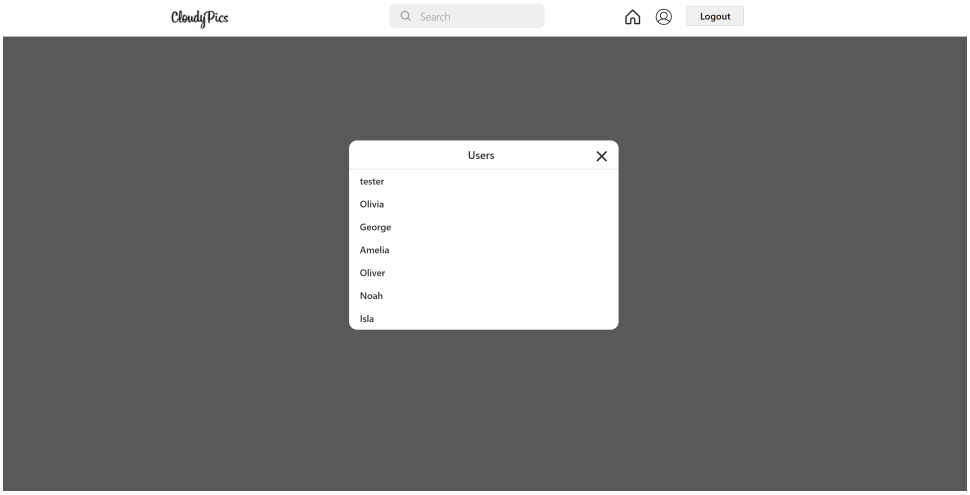

Figure 5: Search users

Clicking on any user will display their profile page. When viewing another user's profile, such as *Isla*'s profile in Figure 6, the layout is similar to the user's own profile page. However, there are a few differences: we cannot see the list of users they have banned,

and the buttons to edit the username and upload a new photo are absent. Instead, there are two buttons that allow us to follow or ban the user.

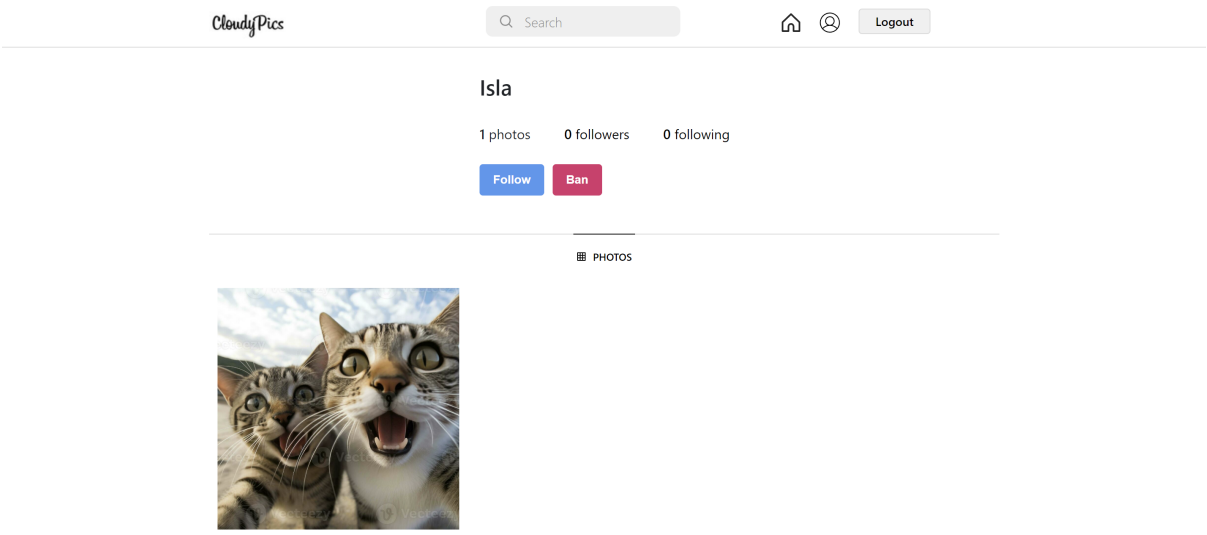

Figure 6: Profile page of another user

By clicking the `follow` button, we can follow this user. Once we do so, the number of followers for the user we are viewing increases by one (Figure 7), and we appear in their follower list (Figure 8).

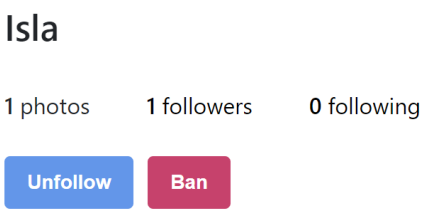

Figure 7: When we follow a user, the button text changes from `follow` to `unfollow`

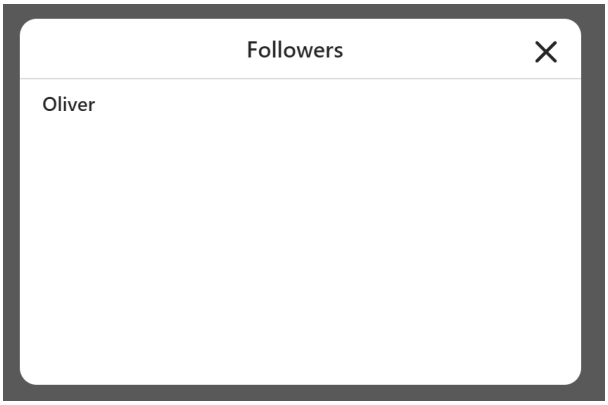

Figure 8: Get the followers of a user

If we navigate to our homepage, we can now see that the picture *Isla* uploaded is part of our stream. In Figure 9, we can also see options to like and comment on a picture.

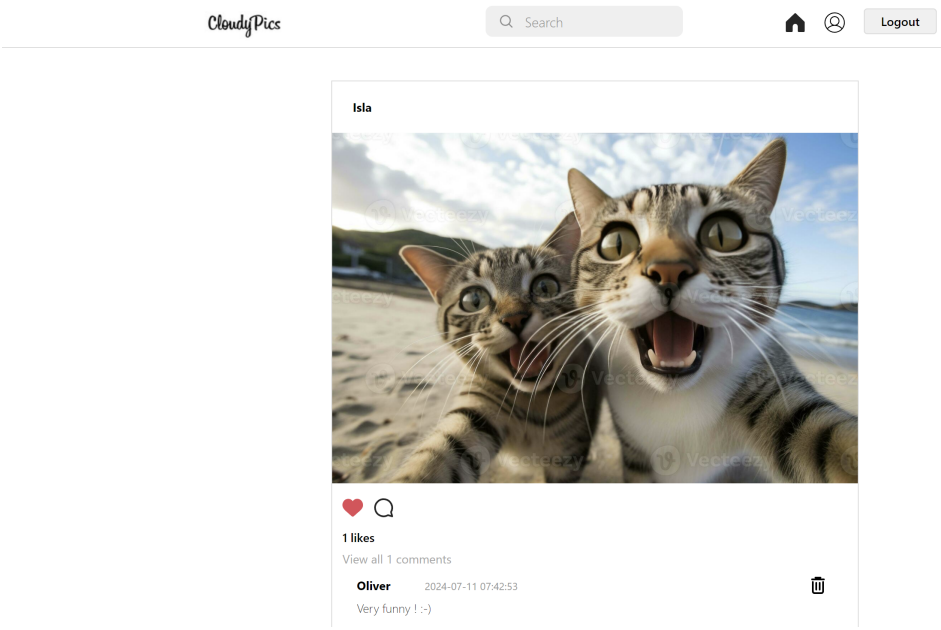

Figure 9: Home page with the stream of the current user

Returning to our profile page, we can click on the `Edit Username` button, which takes us to another page with a form to update our username (Figure 10).

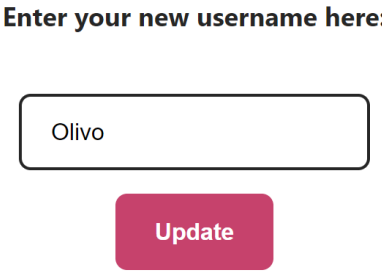

Figure 10: Edit username page

After clicking `Update`, we are redirected to our profile page, where we can see our updated username (Figure 11).

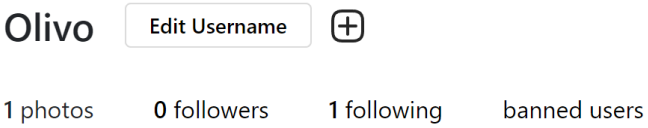

Figure 11: The user profile page shows the updated username

We can demonstrate the banning function by logging in as another user, *Isla*, navigating to *Olivo*'s profile, and clicking the `Ban` button (Figure 12).

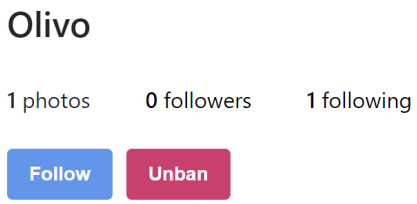

Figure 12: When we ban a user, the button text changes from `ban` to `unban`

If a user is banned by another user, the banned user cannot see the photos, comments, or profile of the user who banned them. In our case, if we log in as *Olivo*, we will now have an empty homepage because we cannot see *Isla*'s uploaded photos anymore (Figure 13).

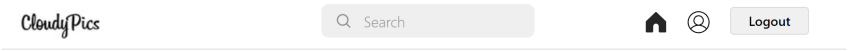

Figure 13: Empty homepage

If we try to open *Isla*'s profile, we are prompted with an error message: `Action forbidden` (Figure 14)

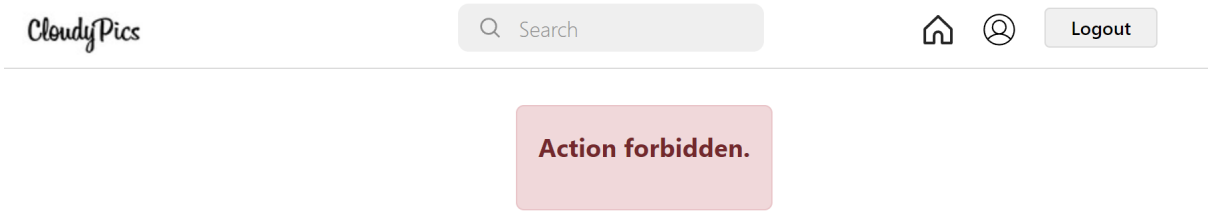

Figure 14: We cannot see the user profile of a user that banned us

3 Implementation

3.1 EC2

To deploy our web application, we utilized Amazon Web Services (AWS) EC2. We began by creating a `t2.small` EC2 instance and assigned it the `LabRole` IAM role. In our AWS Academy Learner Lab, this role has the highest privileges, allowing us to configure and set up various services and components without encountering permission issues.

For the Security Group, we permitted access to ports 22 (SSH), 80 (HTTP), and 3000 (the port on which the backend API listens). Next, we installed the necessary software on the instance, including the Docker Engine, configuring it to automatically start on boot and to launch the frontend and backend containers at startup.

This instance will serve as the basis for creating the Amazon Machine Image (AMI), which will be used to develop the launch template for auto scaling purposes.

3.2 DynamoDB

To store data about the users of our web application and the photos they upload, we use DynamoDB. We created two tables for this purpose: `User` and `Photo`.

The structure of the data in these tables is as follows:

`User` table:

Name	**Type**	**Key type**	**Description**
`id`	String	Partition key	Unique identifier of a user
`username`	String	None	Unique username of a user
`following`	List of strings	None	List containing the IDs of users followed by the current user
`banned`	List of strings	None	List containing the IDs of users banned by the current user

Table 1: User

`Photo` table:

Name	Type	Key type	Description
`id`	String	Partition key	Unique identifier of a photo
`owner`	String	Partition key of the secondary index	ID of the user who uploaded a photo
`created_at`	String	Sort key of the secondary index	Creation timestamp of a photo.
`likes`	List of strings	None	List containing the IDs of users who liked a photo
`comments`	List of objects	None	List containing the comments of a photo

Table 2: Photo

In particular, the `comments` object are structured as follows:

- `id`: unique identifier of a comment (string).
- `owner`: ID of the user who created the comment (string).
- `text`: content of the comment (string).
- `created_at`: creation timestamp of the comment (string).

3.3 Simple Storage Service (S3)

We used Amazon S3 to store the actual images loaded by the users. The implementation of this service is rather straightforward, we created a bucket and enabled public access from the Internet by writing the following policy:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PublicReadGetObject",
            "Effect": "Allow",
            "Principal": "*",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::cloudypicsbucket/*"
        }
    ]
}
```

Figure 15: S3 bucket policy

3.4 Lambda functions

As part of our project, we have developed two Lambda functions aimed at enhancing the functionality and security of our social media web application. These functions perform serverless computation and are triggered during the upload process of a photo. Both functions are written in Go.

3.4.1 Image-Rekognition

We have integrated Amazon Rekognition, a cloud-based image and video analysis service. This service provides easy-to-use APIs capable of swiftly analyzing any image or video file stored in Amazon S3. With this tool, we can effectively identify explicit nudity, violence, visually disturbing elements, drugs, hate symbols, and crime scenes within uploaded images. If an image matches any of these characteristics, it is promptly removed from S3 to prevent viewing, and the user who tried to upload it receives an error message. Below is a snippet of code highlighting the most interesting part of the Lambda function:

```
func isImageSafe(imageBytes []byte) (bool, error) {
    result, err :=
        ↪ rekognitionClient.DetectModerationLabels(context.TODO(),
        ↪ &rekognition.DetectModerationLabelsInput{
        Image: &rekognitionTypes.Image{
            Bytes: imageBytes,
        },
    })
    if err != nil {
        return false, fmt.Errorf("failed to detect moderation labels:
            ↪ %w", err)
    }

    for _, label := range result.ModerationLabels {
        log.Printf("Detected label: %s, Confidence: %f\n", *label.Name,
            ↪ *label.Confidence)
        if strings.Contains(*label.Name, "Explicit Nudity") ||
           strings.Contains(*label.Name, "Violence") ||
           strings.Contains(*label.Name, "Visually Disturbing") ||
           strings.Contains(*label.Name, "Drugs") ||
           strings.Contains(*label.Name, "Hate Symbols") ||
           strings.Contains(*label.Name, "Crime Scene") {
            return false, nil
        }
    }

    return true, nil
}
```

Listing 1: Snippet of image-rekognition Lambda function

3.4.2 Compression

To optimize our application's storage and improve image delivery, we implemented a compression process. This is crucial for faster load times and reduced S3 storage costs, essential for maintaining smooth and cost-effective operations. Our compression function reduces the file size of uploaded images without compromising quality. We chose to use the image processing library `imaging`, which offers efficient compression algorithms. Unlike other libraries that rely on external C libraries, `imaging` is a pure Go library. This choice avoids the complexities associated with creating environments for the building of shared libraries with our Lambda function.

3.5 Auto Scaling Group

To ensure our web application high availability and scalability, we needed to set up an Auto Scaling Group (ASG). This process began with creating an Amazon Machine Image (AMI) from our running EC2 instance (as described in subsection 3.1), which captured the instance's configuration, software, and operating system.

First, we selected the running instance that hosted our web application. By choosing the `Image` option and then `Create Image`, we configured the AMI, providing a name and description and selecting the necessary volumes.

With our AMI ready, we proceeded to create a launch template. We accessed the `Launch Templates` section and started the creation of a new template. We provided a name and description, selected our custom AMI, and configured the instance type, key pairs and security groups. We also specified the storage configurations before finalizing the template.

Next, we set up the Auto Scaling group itself. We navigated to the `Auto Scaling Groups` section and started creating a new ASG. We named our ASG and selected the launch template we had prepared in the previous step. We configured the group size, setting both the initial and minimum number of instances to 1, and the maximum to 6, allowing our application to scale as needed.

We then configured our web application load balancer by adding a listener on ports 80 (frontend) and 3000 (backend). We created two target groups for these ports and associated them with the load balancer, as shown in Figure 16. Back in the ASG configuration, we linked the load balancer and target groups.

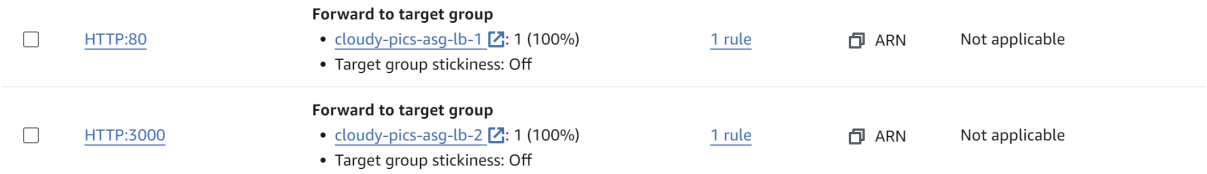

Figure 16: Load balancer listeners and target groups

To enhance scaling efficiency, we established a warm pool with one instance in stopped

state, ensuring rapid instance launches during peak demand periods.

Additionally, we defined a dynamic scaling policy to enable the Auto Scaling group to automatically adjust the number of instances based on traffic fluctuations. Specifically, we configured the policy to remove one instance from the ASG when the number of requests drops below 300 requests per minute (scale-in), and to add an instance to the ASG when the metric exceeds 600 requests per minute (scale-out), as outlined in Figure 17, 18. To do so, we defined two custom alarm on CloudWatch, based on the Application Load Balancer (ALB) metric `RequestCountPerTarget`.

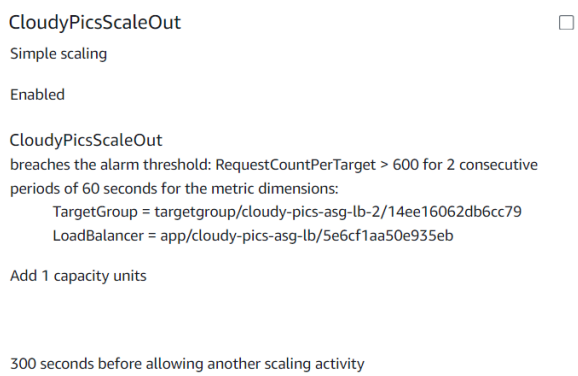

Figure 17: Dynamic scale-out policy

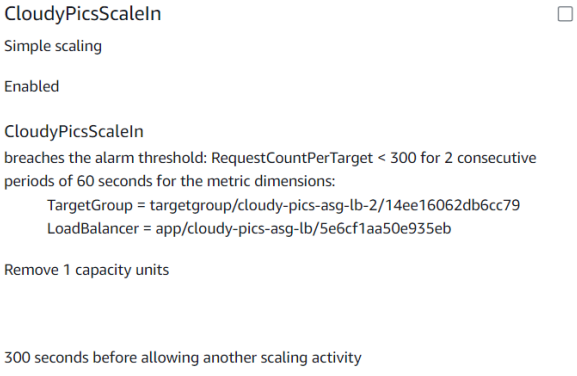

Figure 18: Dynamic scale-in policy

Scaling-out takes time because a new EC2 instance must boot up, even though the software is pre-installed in a custom AMI. Therefore, we scale-in only after observing two consecutive data points indicating overcapacity, while we scale-out with just one data point indicating too many requests.

The warm-up period for scaling-out is set to the default value of 240 seconds. Experiments showed this is a reasonable amount of time for a new EC2 instance to boot up and be ready to handle incoming requests. For scaling-in, the same warm-up period was sufficient for the system to adjust.

Notice: The values reported in this subsection (specifically, in Figures 17, 18) are different from the ones mentioned in subsection 4.1 (initial benchmarking) because we had to adjust them due to AWS banning our Learner Lab accounts. Further explanations are provided in subsection 4.2.

4 Scaling Tests

4.1 Initial Benchmarking

To conduct the testing phase, we used another EC2 instance within the same availability zone to minimize the impact of external factors, such as global internet traffic. By keeping all operations within the AWS environment, we aimed to reduce the risk of being banned during the tests. Unfortunately, despite these precautions, we were still banned.

The metric we focused on was the number of requests per second, given that this is a CRUD (Create, Read, Update, Delete) web service, which is neither CPU-bound nor memory-bound. Specifically, we aimed to explore how the average response time and success rate of requests changed as the number of requests increased.
Our objective was to determine the maximum number of requests per minute that an EC2 instance running our application could handle, in order to set an appropriate threshold for auto scaling purposes. To achieve this, we created a program in Golang, named `healthcheck.go`, which can send a specified number of requests distributed over a minute. Since we have already developed the backend in Golang, we chose to use it for our testing programs as well. This choice is advantageous due to Golang's powerful concurrency model, which leverages lightweight threads managed by the Go runtime, known as *goroutines*.
To run the program properly, it requires an argument `n`, representing the number of requests to be sent. For example, if $n = 100$, then `healthcheck.go` will send one request every 600 milliseconds. The HTTP requests are directed to the `/liveness` endpoint of our web application. The time taken for each request is measured, and at the end, the mean and standard deviation are calculated, as we can see from the graph in Figure 19. The test was repeated five times, consistently yielding the same average results.

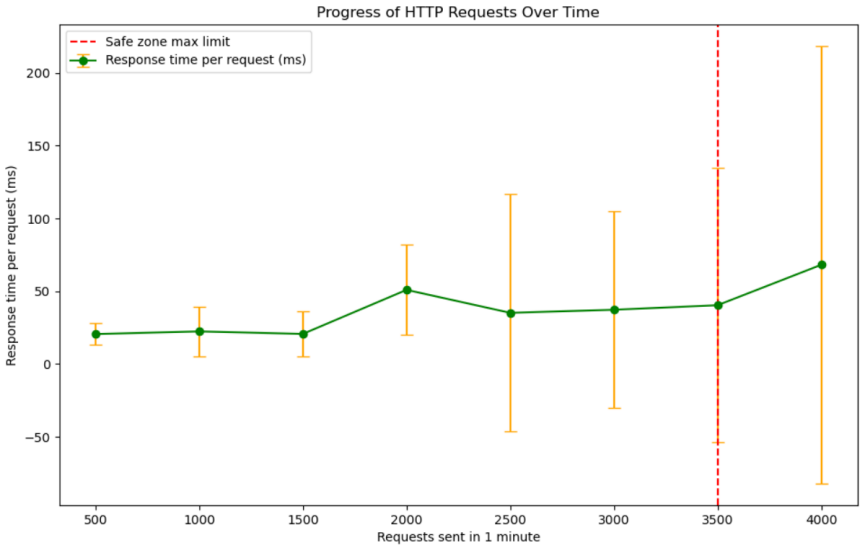

Figure 19: Graph of response time per request

We chose to perform our benchmark testing on the `/liveness` endpoint of our backend because we have seen that the average response time for such request is similar to the most frequent `GET` requests a user would normally perform on our application.

When the `/liveness` endpoint is requested, the backend performs the following actions:

- Checks connectivity with DynamoDB by listing the tables;
- Checks connectivity with S3 by listing the buckets;
- Checks connectivity with Lambda by listing the functions.

A request is considered failed if any of these connectivity checks fail. In such cases, a `500 Internal Server Error` status code is returned in the response, instead of the expected `200 OK` status. Additionally, a request is also considered failed if the response time exceeds the timeout of 2 seconds.

As we can see from Figure 20, the benchmark for our web application is **3500 requests** per minute.

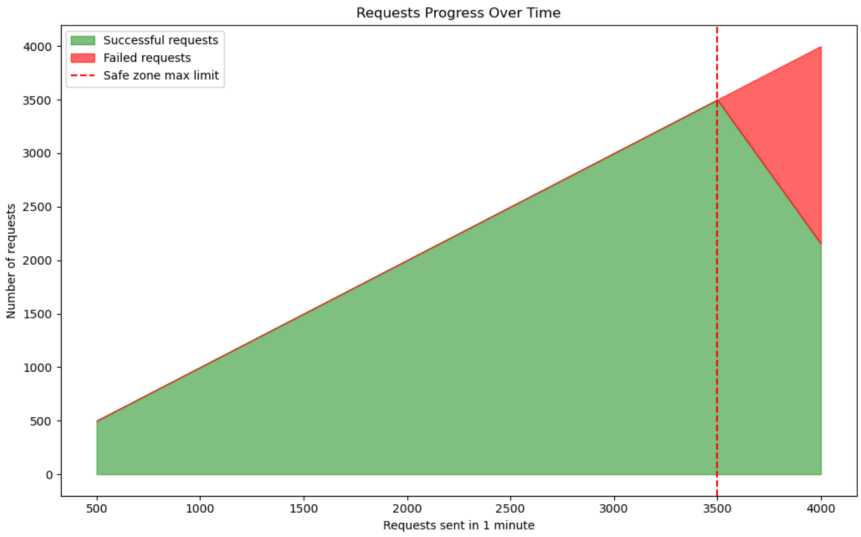

Figure 20: Graph of success rate over number of requests

Below is presented a snippet of our `healthcheck.go` program, showcasing the `benchmark` function. This function utilizes goroutines to send HTTP requests to the server concurrently.

```
func benchmark(url string, n int) {
    var wg sync.WaitGroup
    wg.Add(n)

    limiter := rate.NewLimiter(rate.Every(time.Minute/time.Duration(n)),
        ↪ 1)

    for i := 0; i < n; i++ {
        limiter.Wait(context.Background())
        go makeHTTPRequest(url, &wg)
    }

    wg.Wait()
    fmt.Fprintf(os.Stderr, "\n\n")
}
```

Listing 2: `benchmark` function to make HTTP requests concurrently

4.2 EC2 Auto Scaling Testing

4.2.1 First attempt with Jmeter

As discussed in subsection 4.1, our web application can handle up to 3500 requests per minute. However, we aim to scale before reaching this threshold. Ideally, we will scale-out when the request rate exceeds 60% of this limit and scale-in when it falls below 30%, following the rule of thumb discussed in the lectures.

Our scaling policy is as follows:

Metric (Request / Minute)	Adjustment
req/min $<$ 1050	-1 EC2 instance
1050 $\leq$ req/min $\leq$ 2100	0
req/min $>$ 2100	+1 EC2 instance

Table 3: Auto Scaling policy

Initially, we decided to use **JMeter** for our performance tests. We installed JDK 17 on an EC2 instance running Debian in the same availability zone as the Auto Scaling group instance. Then, we installed JMeter on our tester VM. To run its GUI while connected via SSH, we set up PuTTY and enabled X11 forwarding. When launching the command `./jmeter`, the usual GUI of the application appeared on the screen, allowing us to customize and launch our test with ease.

The detailed configuration we used for JMeter is shown in Figure 21. We created a thread group to simulate all possible user actions on our web application (20 threads), leveraging the extensive customization capabilities provided by JMeter.

Specifically, we generated users with randomized usernames using JMeter's built-in functions. These users then logged in and retrieved their IDs, which they stored as the value of the bearer token for the Authorization header for the entire loop. During the loop, our threads interacted with pre-existing users in our application by randomly liking and unliking their pictures, uploading images of various sizes, following and unfollowing users, and banning and unbanning users.

We structured our test plan to have 100 concurrent threads with a ramp-up period of 30 seconds, aiming to reach an average of 2000 requests per minute in order to approach our scale-out threshold. Unfortunately, we were banned and the connection was closed, resulting in an abrupt end to our test and preventing us from obtaining any graphs.

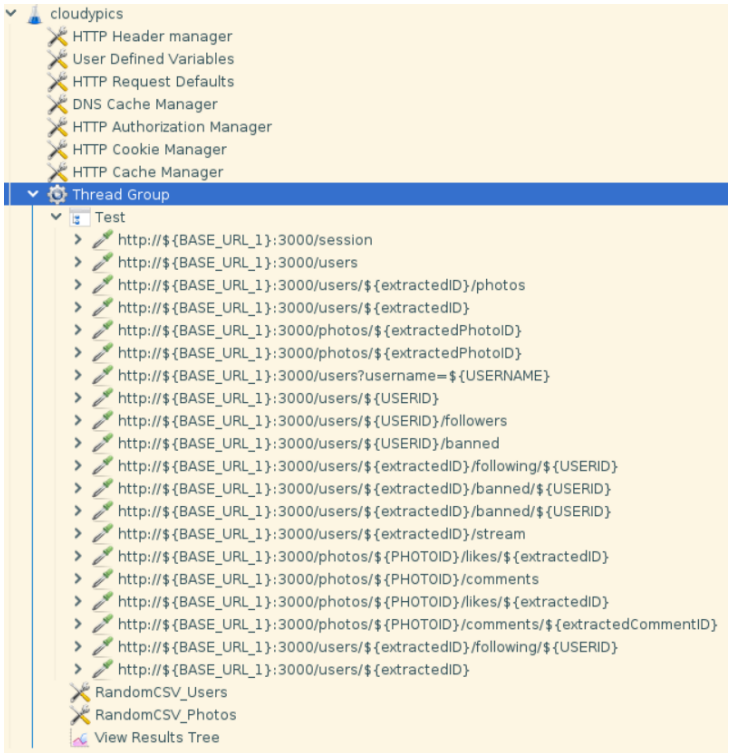

Figure 21: JMeter Test Plan

4.2.2 Custom tool for simulating user behavior using a Markov Chain

Due to the possibility that JMeter caused our ban, despite operating from the same Availability Zone (AZ) as the running instance of the Auto Scaling group, we decided to develop a custom tool.

We designed a Markov Chain in Golang to emulate user requests, as shown in Figure 22. This involved defining a specific set of actions, each with a probability between 0 and 1. Each action originates from a `Start` state, has a certain probability of occurring, and then returns to the `Start` state. The exceptions to this pattern are the actions `UploadPhoto`, `LikePhoto`, and `FollowUser`. These actions must transition through their respective opposite states (e.g., `UploadPhoto` to `DeletePhoto`, because a user cannot delete a photo that has not been uploaded yet) before returning to the `Start` state, accurately modeling most functionalities of our web application.

To simulate realistic user interactions on the web application, we introduced a 2-second waiting time between each request. This delay mimics the typical pause a user might experience between consecutive actions.

The `user-model.go` file contains the implementation of the Markov chain. We began by initializing an enum to represent the possible states (e.g., `SearchUser` or `LikePhoto`) and some global arrays holding the usernames, user IDs, and photo IDs of preexisting users in our application, which we will interact with during testing. We also created the `UserModel` structure to represent a user during a simulation, storing information such as the current state and the duration of requests made. Additionally, we developed the `UserModelStats` structure to gather statistical data about the user's activity.

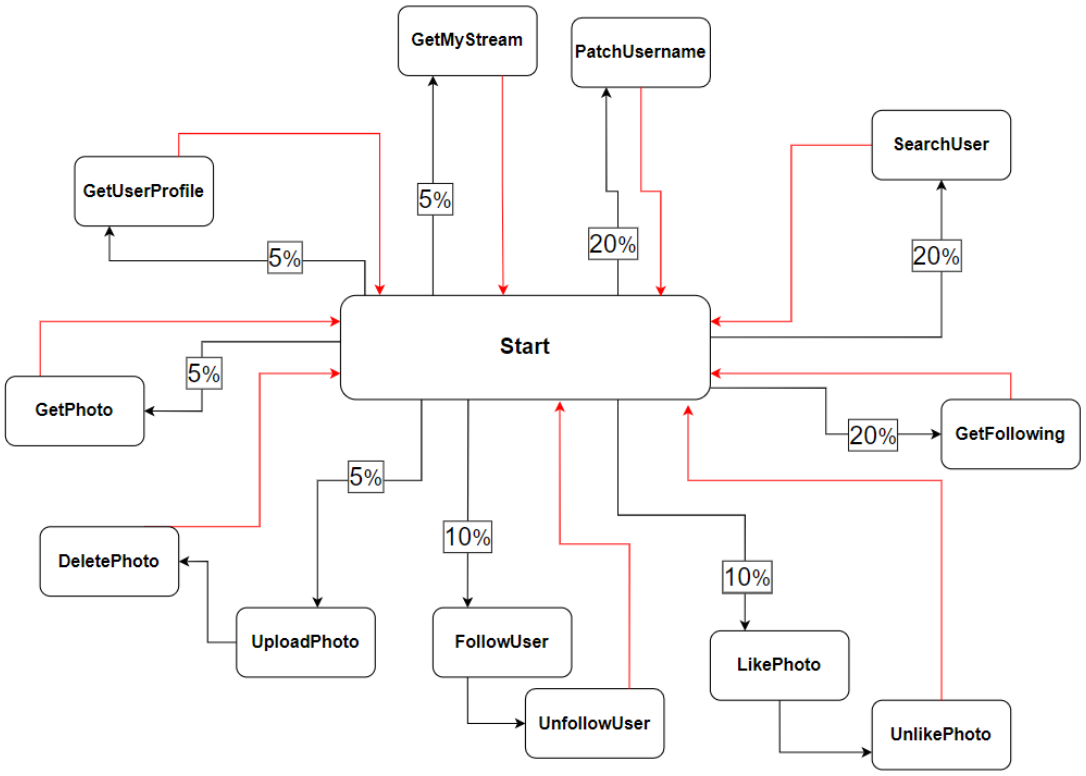

Figure 22: Markov Chain to simulate user behavior

To implement the different actions a user might take, we created functions such as `getUserProfile()`. The initial state corresponds to the `start()` function, which selects the next action based on the previously defined probabilities; this decision is made using a `switch` statement.

All these functions need to send HTTP requests to our applicataion, accomplished by invoking the `collectHTTPRequest` or `collectHTTPRequestWithBody` functions. These functions record the time related to the request for statistical purposes. The actual request are handled by functions defined in the `http-requests.go` utils library we created.

Below is a snippet of code demonstrating the `likePhoto()` function as an example.

```
func (u *UserModel) likePhoto() State {
log("Entering LikePhoto state...")

id := PhotoIDs[rand.Intn(len(PhotoIDs))]
err := u.collectHTTPRequest("/photos/" + id + "/likes/" + u.userId,
    ↪ u.userId, "PUT", "")

if err != nil {
    log(err)
    return Start
}
```

```
    u.lastPhotoId = id

    return UnlikePhoto
}
```

Listing 3: Simulate a user liking a photo

Initially, we randomly choose an already existing picture to like from the globally provided list of IDs. Then, we provide the `collectHTTPRequest` function with the complete path, our ID (required for authorization to perform this action), the method, and the body. In this case, the body is empty as nothing else is needed.
We also need to keep track of the photo ID we chose so that in the `UnlikePhoto` state, we can unlike that specific picture and avoid causing an error.
In our chain, immediately after liking a picture, the simulated user also unlikes it. This is done to prevent errors since, without this check, a simulated user might attempt to unlike a photo that was never liked in the first place. In the code, this is implemented with `return UnlikePhoto`, which sets the next state. Normally, the next state would have been `Start`.

Finally, the `user-behavior.go` program is responsible for creating and running a specified number of user models in parallel. It collects and prints statistics about the performance of these simulations at regular intervals.
The number of parallel users can be manually set using the `parallelUsers` variable. The `main` function will start the simulations in the background and compute and print the current statistics every minute.

4.2.3 Second attempt with our custom tool

The most effective way to test our auto scaling policies is by gradually increasing and subsequently decreasing the number of concurrent users to stress the application. Given that each execution of our program simulates 10 parallel users, with each user making an average of 30 requests per second, we incrementally launched our program multiple times on our tester EC2 instance. This strategy aimed to approach our scale-out threshold. Unfortunately, we were banned by AWS once again. As this marks our second ban, it appears AWS might not appreciate our enthusiasm for rigorous testing.

4.2.4 Third attempt with lower thresholds and multiple tester instances

Since we had a last AWS Learner Lab account, we started being very cautious and we did some research to prevent further banning. We decided to lower the thresholds of our auto scaling policy to show that the application is able to scale correctly. Considering a limit of 1000 requests per minute, our new scaling policy is the following:

Metric (Request / Minute)	Adjustment
req/min $<$ 300	-1 EC2 instance
300 $\leq$ req/min $\leq$ 600	0
req/min $>$ 600	+1 EC2 instance

Table 4: New Auto Scaling policy

Furthermore, we discovered that generating a high volume of requests per minute from the same IP address, even within the AWS perimeter, could be a possible reason for our bans. To address this, we decided to create five tester EC2 instances and distribute the testing workload among them.

To conduct the testing, we deployed the five EC2 instances and initiated tests sequentially from each one. Our custom tool ran with 10 concurrent users on each instance, with each user executing one operation every 2 seconds. This configuration led to an average of between 250 and 300 requests per minute. The tests were conducted within the same availability zone to eliminate the negative effects of traversing the public Internet.

Before launching our simulation, we created a CloudWatch dashboard to monitor the metrics `RequestCountPerTarget` and `RequestCount`. This allowed us to verify whether the Auto Scaling group was functioning correctly and if the load was being adequately distributed among the automatically launched EC2 instances.

The graph resulting from our test is shown in Figure 23.

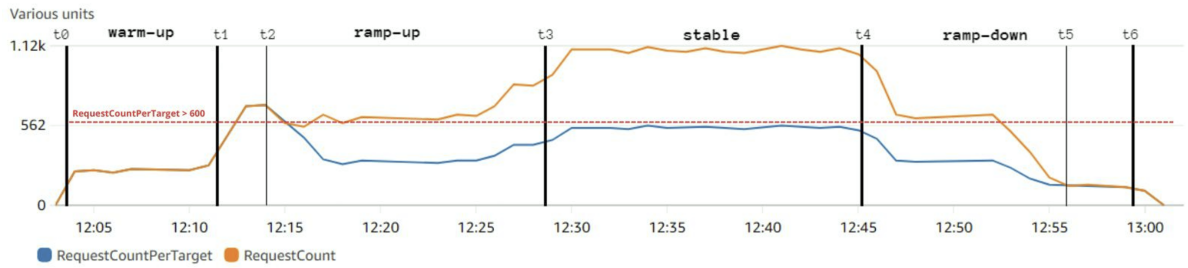

Figure 23: CloudWatch reported `RequestCount` and `RequestCountPerTarget` metrics during the various phases of our scaling test

In our graph, we can distinguish the various phases of our test, which span an interval of one hour:

1. The test started with 10 concurrent users by launching the first tester instance. The measured number of requests per minute averaged between 250 and 300.

2. After a while, the number of users increased from 10 to 20 by launching the second tester instance with an additional 10 concurrent users. The number of requests per minute now ranged between 500 and 600.

3. Immediately after, the number of users increased from 20 to 30 by launching the third tester instance with an additional 10 concurrent users. The threshold of 600 is exceeded and the scaling-out policy is triggered, adding one instance.

4. After a while, the users increased from 30 to 40 by launching the fourth tester instance with an additional 10 concurrent users. The amount of requests per minute is now between 800 and 900.

5. Immediately after, the number of users increased from 40 to 50 by launching the fifth and last tester instance with an additional 10 concurrent users. The amount of requests per minute is now between 1100 and 1200.

6. We maintained this maximum number of concurrent users for approximately 20 minutes.

7. We began terminating the programs, one by one, on the tester EC2 instances, waiting approximately five minutes before going down from 30 to 10 users.

8. After shutting down four instances, the number of requests per minute dropped below 300, triggering the scaling-in policy, which removed the previously added instance.

Let's dive deeper, analyzing the different phases of our test:

- **Warm-up**: Between t_0 and t_1, we had a stable load of 10 users.
- **Ramp-up**: Between t_1 and t_3, we started with 10 users, increased to 30 users, and ended with 50 users.
- **Stable**: Between t_3 and t_4, we had a stable high load of 50 concurrent users.
- **Ramp-down**: Between t_4 and t_6, we started with 50 users, decreased to 30 users, and finally reduced to 10 users.

In Figure 24, we show a snapshot of our test execution during the **stable** phase, where the number of requests was at its maximum, with each of the five tester instances running the program. As seen in the statistics captured in the fifth terminal, there are no failed requests. This indicates that we have not yet reached the true limit of our web application, as we expected since we lowered the threshold.

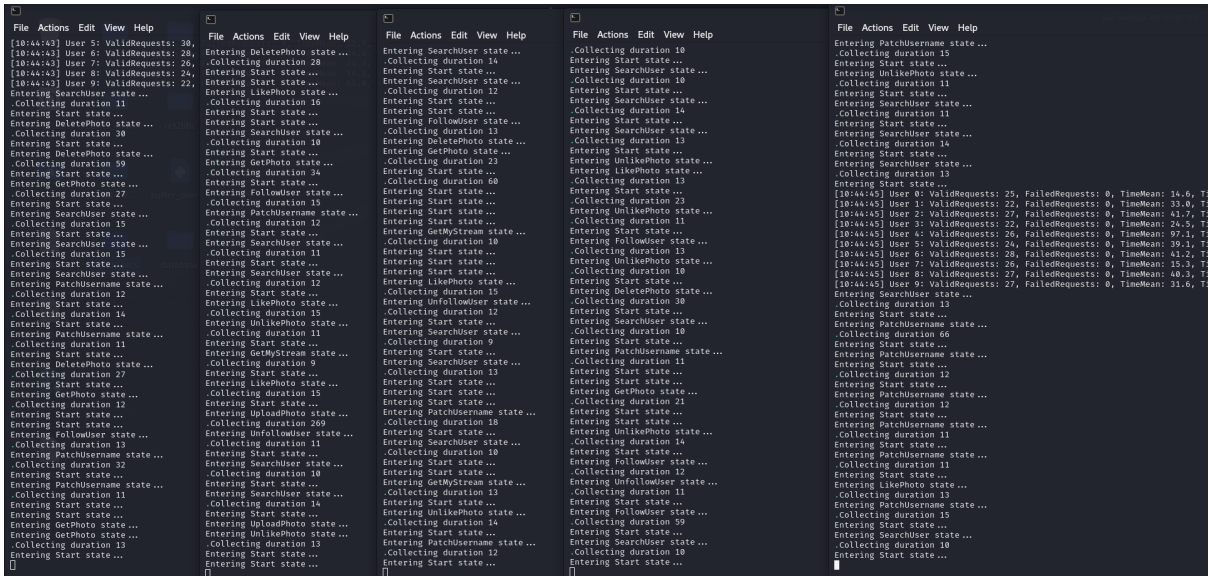

Figure 24: Workload distribution on the five tester instances during the **stable** phase

We monitored the entire test from the EC2 dashboard, where we observed in real-time the launch of a second instance during the scale-out process (Figure 25), as well as the termination of the second running instance during the scale-in process (Figure 26).

Name	Instance ID	Instance state
autoscaler-1	i-0e0ac91c6777a5b8d	Running
autoscaler-2	i-03ce7d1ef4b9c1c50	Running
autoscaler-warm	i-0bef3815c68d75073	Stopped
tester1	i-0e7995e15540177fb	Running
tester2	i-025c024094bc53a1a	Running
tester3	i-030879cd122f4bd8b	Running
tester4	i-0e10eeef08bd928de	Running
tester5	i-02734c3398ec22e4a	Running

Figure 25: Launch of second instance for the scale-out

Name	Instance ID	Instance state
autoscaler-1	i-0e0ac91c6777a5b8d	Running
autoscaler-2	i-03ce7d1ef4b9c1c50	Stopping
autoscaler-warm	i-0bef3815c68d75073	Stopped
tester1	i-0e7995e15540177fb	Running
tester2	i-025c024094bc53a1a	Running
tester3	i-030879cd122f4bd8b	Running
tester4	i-0e10eeef08bd928de	Running
tester5	i-02734c3398ec22e4a	Running

Figure 26: Second instance is stopping for scale-in

We conducted two tests, each lasting approximately one hour, and obtained consistent results across both tests, as illustrated in Figure 27.

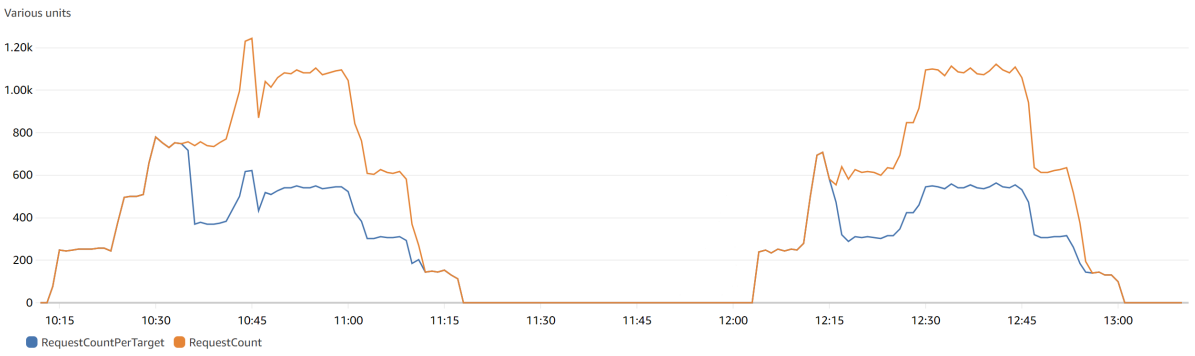

Figure 27: The two load tests performed, side-by-side

4.3 Bottleneck Stress Testing

We identified the `UploadPhoto` function as the most intensive operation of our application. This function uploads the photo to Amazon S3, creates the relevant entries in the appropriate DynamoDB tables, and then invokes the two Lambda functions described in subsection 3.4: `image-rekognition` and `compression`.

In our previous test, uploading a photo had a low probability of occurring in our Markov Chain, reflecting the realistic scenario where users seldom upload photos. However, we now want to stress test this functionality to determine how many requests this bottleneck can handle before it starts to fail and become unresponsive. Additionally, we are interested in investigating whether any potential failure in this function would be due to a failure in the Lambda function execution (e.g., too many concurrent invocations) or other relevant factors.

To perform the bottleneck stress testing, we used JMeter. We accessed the JMeter GUI using PuTTY from a tester EC2 instance, as before. By producing requests from an increasing number of concurrent users, we kept raising the number of threads until we encountered errors.

We began by setting the number of concurrent users to 15, with each user performing 10 loops invoking the Lambda functions by uploading a random picture. By appropriately setting the ramp-up time, JMeter automatically starts with a few samples, gradually increasing them step by step to the maximum, and then gradually decreases the number of threads.
The configuration we used for JMeter is shown in Figure 28. Notably, we added a `Constant Throughput Timer` set to 10 samples per minute per thread. This means that with the maximum number of concurrent threads, we will reach $\#Threads \cdot 10$ invocations per minute (e.g., with 75 users, 750 invocations per minute).

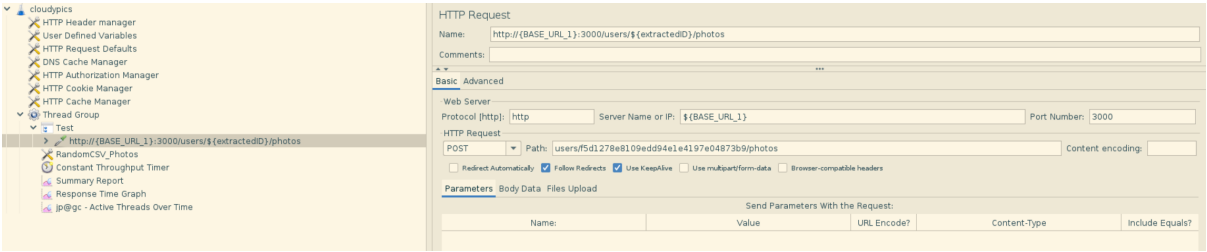

Figure 28: JMeter configuration for testing the photo upload process

We now proceed to present three test results, each consisting of a summary report (containing various statistics) and an "Active Threads Over Time" graph, obtained through a JMeter plugin.

As explained earlier, we start with 15 users, then increase to 75 users as a midpoint, and finally observe our application starting to fail due to being overloaded when we reach 200 concurrent users (resulting in a peak of 2000 requests per minute).

As we can see from the summary report in Figure 29, there were no failed requests. Figure 30 displays the variation in the number of active threads over time.

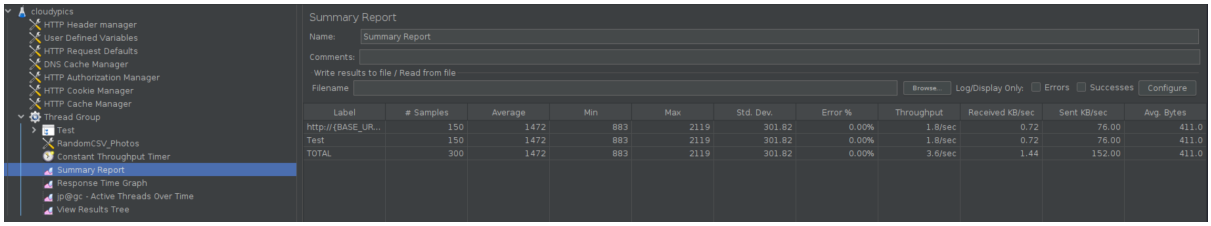

Figure 29: JMeter test summary with 15 users

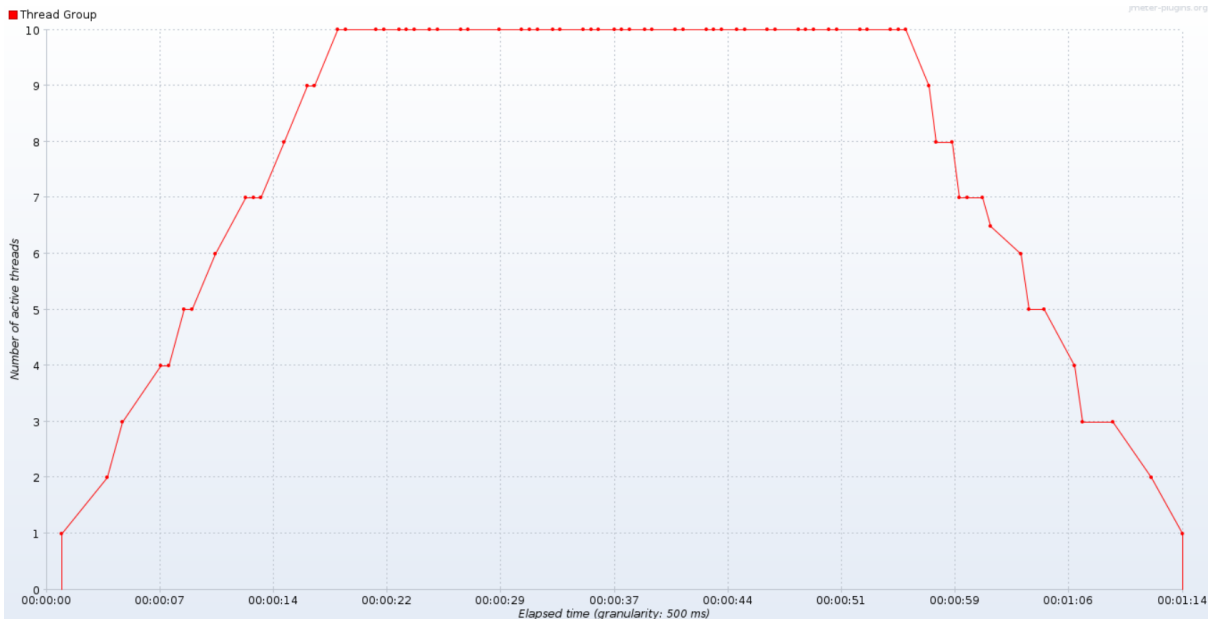

Figure 30: Active Threads Over Time graph with 15 users

Next, we increased the number of concurrent users to 75 and ran the same test, again encountering no errors, as shown in Figure 31. The corresponding graph of active threads over time is illustrated in Figure 32.

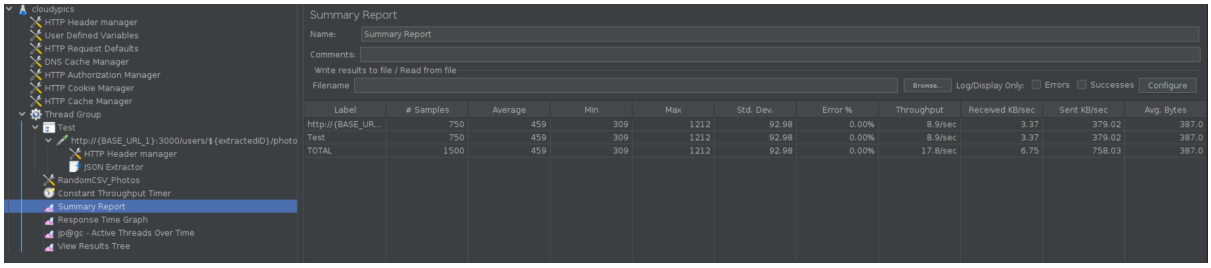

Figure 31: JMeter test summary with 75 users

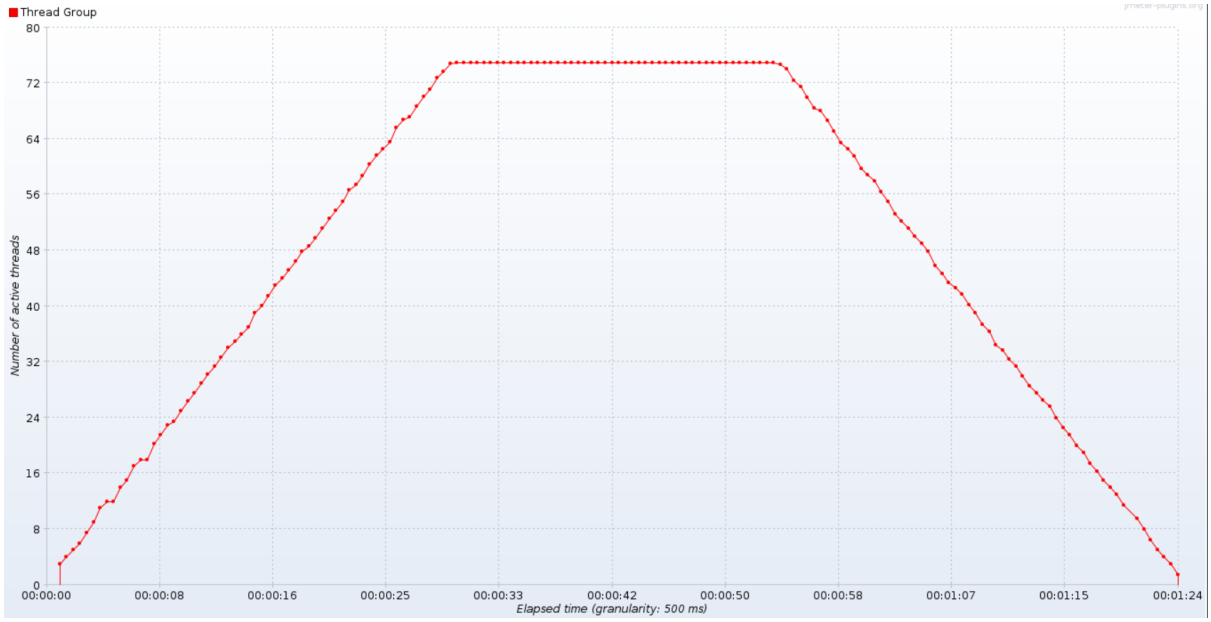

Figure 32: Active Threads Over Time graph with 75 users

After many attempts, we encountered a failure rate of 18.65% with 200 concurrent users, establishing our benchmark. The related summary and graph are shown in Figures 33 and 34, respectively.

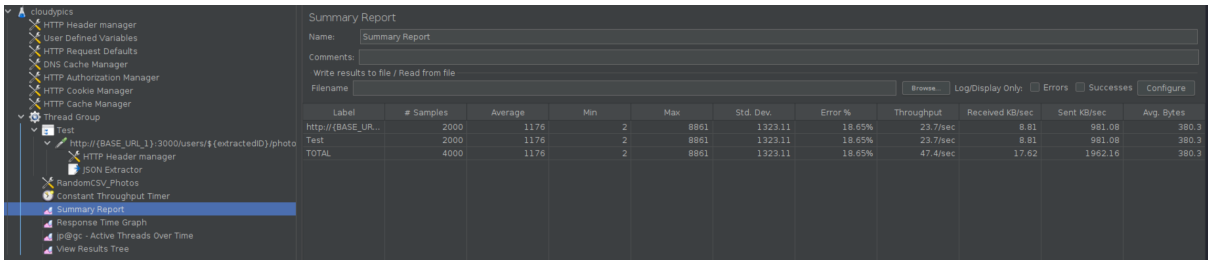

Figure 33: JMeter test summary with 200 users

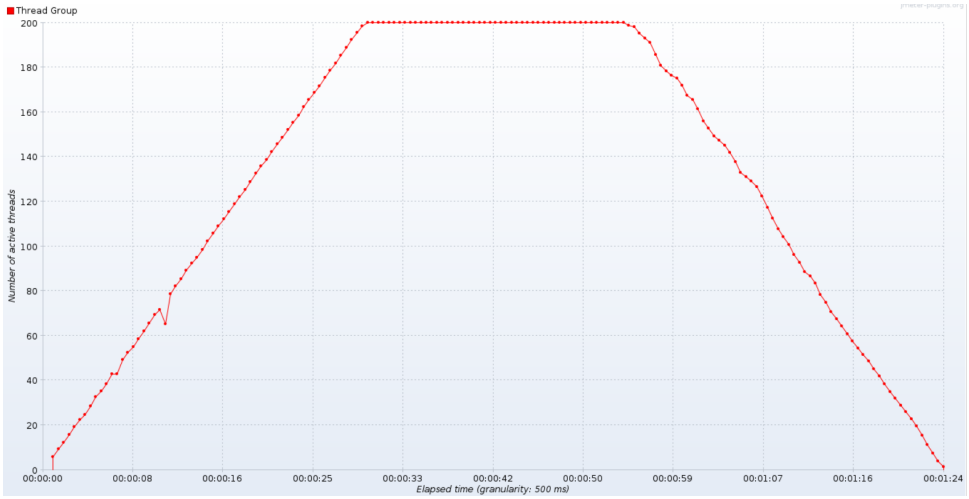

Figure 34: Active Threads Over Time graph with 200 users

At this point, we observed that the operations started to fail not because of the Lambda functions. Figures 35 and 36 show the monitoring graphs from AWS for the for the

`image-rekognition` and `compression` Lambda functions, respectively. These graphs indicate that the success rate remained at 100%, and provide other interesting information such as the number of invocations and the duration in milliseconds.

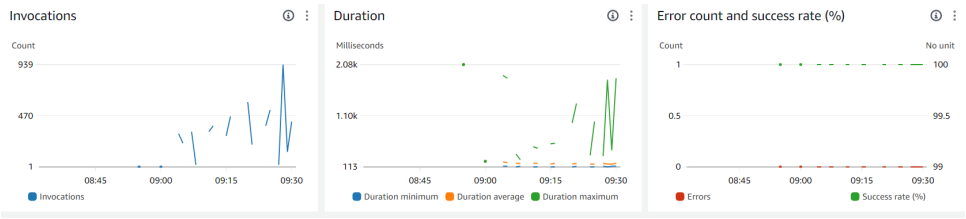

Figure 35: Monitoring graphs for `image-rekognition` lambda function

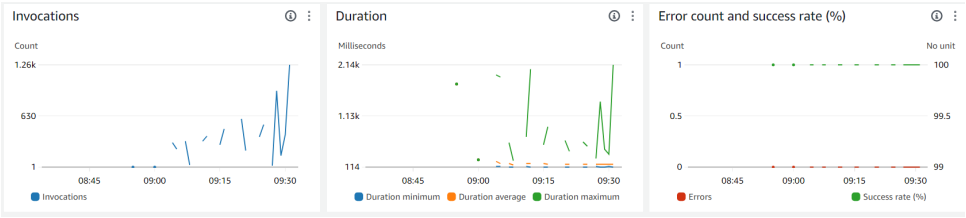

Figure 36: Monitoring graphs for `compression` lambda function

5 Availability Tests for EC2

To enhance the availability of our service, we configured the AWS Auto Scaling group to utilize multiple Availability Zones (AZs). By distributing EC2 instances across different AZs, we aim to ensure that our application server remains operational even if one AZ experiences an outage or failure. This redundancy is a critical component of achieving high availability. For our web application, we employed three Availability Zones: us-east-1a, us-east-1b, us-east-1c.

Another test of the application server's availability involved simulating an EC2 instance failure. We manually stopped an instance to observe how the AWS Auto Scaling group would respond to this simulated crash. We verified that the Auto Scaling group successfully identified the stopped instance and promptly launched a replacement instance within our EC2 Target Group, as shown in Figures 37, 38, 39, 40, 41.

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS
autoscaler-1	i-0e0ac91c6777a5b8d	Running	t2.small	2/2 checks passed	View alarms	us-east-1c	ec2-52-54-188-115.co...
autoscaler-warm	i-03ce7d1ef4b9c1c50	Stopped	t2.small	–	View alarms	us-east-1b	–

Figure 37: Initial state: one running instance and one stopped instance (warm pool)

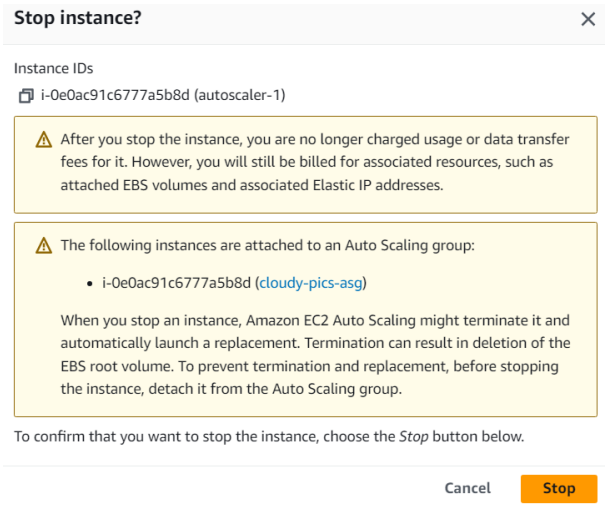

Figure 38: As expected we receive an alert when trying to stop the running instance, since it is part of an Auto Scaling group

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS
autoscaler-1	i-0e0ac91c6777a5b8d	Stopping	t2.small	–	View alarms	us-east-1c	ec2-52-54-188-115.co…
autoscaler-warm	i-03ce7d1ef4b9c1c50	Stopped	t2.small	–	View alarms	us-east-1b	–

Figure 39: The running instance is stopping

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS
autoscaler-new	i-038005aa6605cada8	Pending	t2.small	–	View alarms	us-east-1a	ec2-54-204-209-124.co…
autoscaler-1	i-0e0ac91c6777a5b8d	Stopped	t2.small	–	View alarms	us-east-1c	–
autoscaler-warm	i-03ce7d1ef4b9c1c50	Pending	t2.small	–	View alarms	us-east-1b	ec2-54-163-58-249.co…

Figure 40: The previously running instance is now stopped, and the instance in the warm pool is now starting, while a new instance is being created to go in a stopped state in the warm pool

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS
autoscaler-1	i-0e0ac91c6777a5b8d	Stopped	t2.small	–	View alarms	us-east-1c	–
autoscaler-new	i-038005aa6605cada8	Stopped	t2.small	–	View alarms	us-east-1a	–
autoscaler-warm	i-03ce7d1ef4b9c1c50	Running	t2.small	Initializing	View alarms	us-east-1b	ec2-54-163-58-249.co…

Figure 41: Final state: one running instance, one stopped instance in the warm pool, and one stopped instance that will be terminated

Internally, our backend and frontend services run within two separate Docker containers in the same EC2 instance, to simplify the configuration process and to enhance portability. The containers were initiated with the `--restart unless-stopped` flag, ensuring that the Docker daemon automatically restarts them in the event of a crash.

Additionally, our application backend server includes an HTTP endpoint, `/liveness` (as described in subsection 4.1), which is integrated into the configuration of the AWS Elastic Load Balancer (ELB) (Figures 42, 43). This endpoint allows the Application Load Balancer used by the Auto Scaling group to avoid directing traffic to any instance registered in the specified Target Group that fails the healthcheck. If an error occurs, the response status of the `GET` request to the `/liveness` endpoint will return a HTTP status code `500 Internal Server Error`, signaling the ELB to redirect traffic away from the faulty instance.

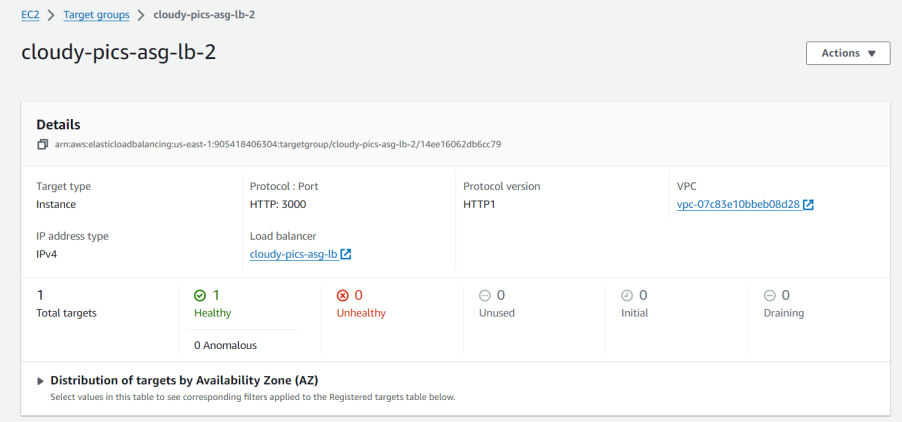

Figure 42: Target Group for the backend

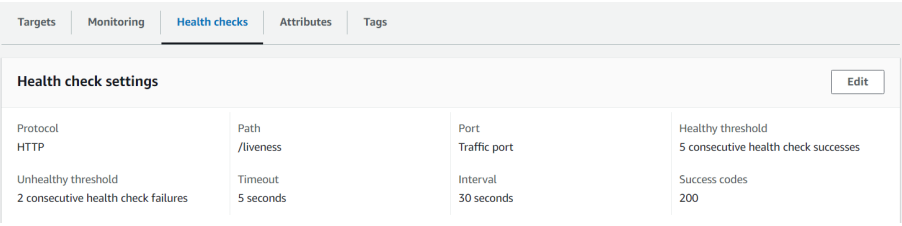

Figure 43: Target Group healthcheck configuration

6 Conclusions

6.1 Future Developments

To create a better platform for our user base, we propose several potential additions and improvements. Time constraints for project delivery, the unfortunate bans of our AWS Academy Lab accounts, and the limitations of free-tier AWS accounts prevented us from utilizing some beneficial services for our project.

An architectural enhancement would be to integrate AWS Cognito for authentication. Cognito offers valuable features such as seamless integration with other AWS services like Amplify, AWS IAM, and AWS API Gateway, and it is also very cost-effective.
We chose DynamoDB for storing users and photos data. However, it would be beneficial to compare its performance with that of a traditional relational database such as Amazon RDS.
Currently, our infrastructure is almost entirely server-based, with the backend hosted on AWS EC2. Analyzing the behavior of a completely serverless architecture could provide interesting insights.

Additional testing activities should be considered, such as extending the testing duration beyond 60 minutes and collecting a larger number of samples. Creating a distributed load testing platform using AWS could prove advantageous. Automating application testing would facilitate future scaling efforts and could be easily achieved using services such as EC2, Lambda, and Amplify.

6.2 Final Remarks

Overall, this project was challenging, requiring significant time to understand and apply the theoretical concepts learned during the lectures. However, through this process, we gained substantial knowledge and experience. One of the most difficult, time-consuming, and tiresome aspects of the project was dealing with AWS bans on our Learner Lab accounts, which necessitated setting up the entire environment on AWS three times. This was particularly frustrating given that we followed the recommendations to conduct our tests within the perimeter of AWS. The uncertainty regarding the rates and parameters that could lead to account bans added to the stress, raising concerns about potentially being unable to complete our project tests. Despite these obstacles, we persevered and managed to conduct the necessary tests, ultimately overcoming these challenges.