SAPIENZA
UNIVERSITÀ DI ROMA

Cloud Photos App

Cloud Computing

Professors:
Emiliano Casalicchio

Students:
Simone Sestito (1937764)
Alessandro Scifoni (1948810)

Academic Year 2023/2024

Contents

1 Introduction

The problem we wanted to address is the design and development of a Scalable and Highly-Available application for photo sharing. Users can access this application with a custom username, upload some photos they would like to share, search other users and their published photos.

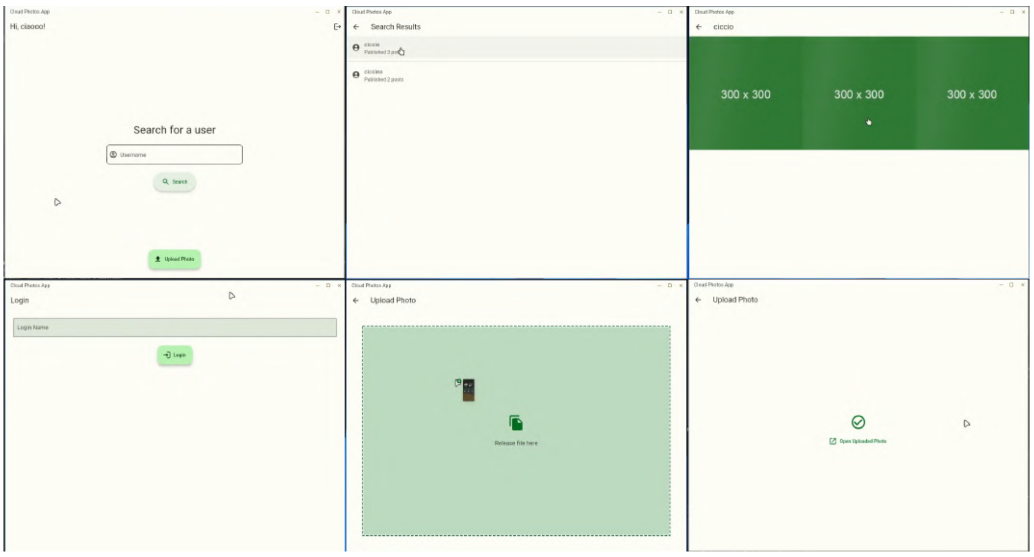


Figure 1: Screenshots of the real Flutter application (Desktop or Webapp)

2 Design

The main issues we want to take into consideration are:

- Since we are interested in scaling, we do not care about the security of the login
- We want not to let users publish bad images, such as violence, weapons or other stuff
- The load of the backend can vary but it is usually pretty stable: this is why we wanted to run it on EC2 instances
- AWS Lambdas are perfect for the Photo Upload process, since it is resource intensive (it invokes `imagemagick` locally to compress photos) and it's usage may be very *spiky* - there are probably way more occasions for it to be stopped with no traffic (= no one is uploading photos) when compared to the backend service (= no one at all is using the service, neither looking at any shared photo by any user)
- The Step Function created is *Express*: for our situation, we were able to observe a huge cost saving (calculated using `https://calculator.aws`) with respect to the Standard workflow, which is more indicated for long-running tasks
- The application is accessible from a webapp, and it is hosted on an S3 public bucket
- We needed a fast and simple database for this application, so the NoSQL solution proposed by DynamoDB is the perfect fit for our implementation
- The application, the backend, Lambdas and all the software was **built from scratch** by ourselves to have maximum flexibility during the project

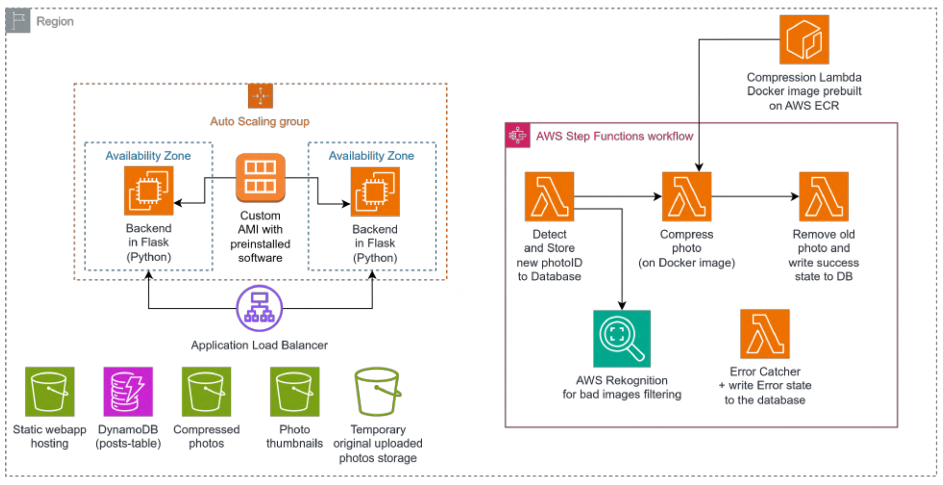

Region
Auto Scaling group
Availability Zone
Availability Zone
Backend in Flask (Python)
Custom AMI with preinstalled software
Backend in Flask (Python)
Application Load Balancer
Static webapp hosting
DynamoDB (posts-table)
Compressed photos
Photo thumbnails
Temporary original uploaded photos storage
Compression Lambda Docker image prebuilt on AWS ECR
AWS Step Functions workflow
Detect and Store new photoID to Database
Compress photo (on Docker image)
Remove old photo and write success state to DB
AWS Rekognition for bad images filtering
Error Catcher + write Error state to the database

Figure 2: AWS Architecture

3 Implementation

3.1 Software application implementation

This application is composed of several components.

The frontend is a multiplatform application built with Flutter. It can be compiled to run both as a web app and an Android and iOS native application, but also it has been tested to run as a native desktop Windows, Mac and Linux application.

The frontend, in whatever form it runs, will interact with the backend via a REST API. It exposes various endpoints for the functionalities of our application. CORS policies have been set to allow the communication from the browser to the REST API.

The backend REST API is implemented in Python using the Flask framework. It runs all the logic of the application and interacts with the other AWS services via the IAM Role "Lab" offered by the Learner Lab.
The photo upload Step Function is split into 2 parts:

- The first Lambdas in the Step Function workflow is a basic Python code, deployed using a simple ZIP, having the library Boto3 as a dependency to allow interaction with the rest of the AWS ecosystem
- The compression lambda function is, instead, a Docker image running inside the Lambda environment. It is based on the `public.ecr.aws/lambda/python` base image, allowing a flawless integration with the environment. Our custom deployed image is built using a Dockerfile, containing the instructions to also install the `imagemagick` famous Linux tool to manipulate images, which we'll use to compress and create the high quality photo and the thumbnail. This allows us to **save ton of space** on S3 buckets, also saving costs.

3.2 Scaling implementation

To implement the autoscaling of the EC2 instance (`t2.micro`), we followed a series of steps, summarized in short below:

1. Launch a simple EC2 instance, enabling SSH and HTTP access in its Security Group
2. Install all the necessary software on this instance, including Docker Engine, our container, a Nginx proxy and so on - also setting up the Nginx proxy and the Docker container containing the application to automatically start on boot
3. Stop the instance and create a custom AMI from it
4. Create a Target Group to automate the creation of new instances based on our custom AMI

5. Create a Load Balancer in 2 Availability Zones, to increase the fault tolerance not being located in a single datacenter

6. Configure the Security Groups of the new instances to allow only HTTP traffic (no SSH allowed at this time, for security reasons)

7. Our Launch Template would include all these points, and also a setting for *Detailed CloudWatch monitoring*, to make us able to react in a quicker way to increases in load demands - the AutoScaling group is set up to have at minimum 1 instance and at most 4

8. Finally, we set up the *Healthcheck* to the EC2 instances via a designated HTTP endpoint, to check if they work properly, and also a *Warm Pool* of 1 instance to make the scaling quicker.

The **Scaling policy**, for which we set the CloudWatch alarms to trigger autoscaling (scale up or scale down) reported that we had to wait for **1** datapoint above the 60% threshold to scale up, but **2 consecutive** data points below the 30% threshold to scale down, since a false positive in scaling down would cost us much more lost time to scale up again.

3.3 Buckets configuration

Our application depends on 3 S3 buckets to work: one with compressed high-quality photos, one with thumbnail photos and one for the Static website.

In addition to this, we also use a Bucket for temporary storage of the original photos uploaded by users, but they get removed after the upload processing process is terminated, by the last Lambda in the workflow.

The 3 buckets we mentioned earlier must be publicly readable, and with the appropriate CORS policies (shown below) and the permissions to allow anyone to read the objects inside of it. In fact, we webapp computes the public URL to access the images it has to show to the user, without the need to let the backend generate pre-signed URLs.

From a security point of view, these buckets contain only public photos, so they are naturally meant to be publicly exposed (but still not listable).

```
[ {
  "AllowedHeaders": [ "*" ],
  "AllowedMethods": [ "GET", "HEAD" ],
  "AllowedOrigins": [ "*" ],
  "ExposeHeaders": [],
  "MaxAgeSeconds": 3000
} ]
```

3.4 Step Functions workflow configuration

The workflow of our application is based on a main step function used to upload the photos of each user. The step function is composed of two lambda functions:

1. The purpose of the first lambda is to store the operation of uploading the photo, putting in the DB the record, so that the backend of the application knows the status of the operation. In particular, it specifies that the operation is pending. Then the lambda function applies some filters on the file used as input, such as the detection of inappropriate content in images. Actually, this feature is used only to filter the content published by the users, so an image detected as bad will be only rejected from our workflow. But **AWS Rekognition service** can be used also for a more fine grained, allowing us to give an explanation for the failure of the operation and how much we are sure of the detection.

2. The purpose of the second lambda is to compress the image and create the thumbnail of the post. When the pipeline of compression reaches the end, the lambda updates the state in the DB to notify the application that the operation succeeded. Since we used ImageMagick, which is not supported natively, we had to choose between a lambda based on a layer or a containerized lambda. Our choice fell on the second one, because we are more familiarized with it and because we are not limited by the few runtimes supported.

4 EC2 scaling tests

The tests are made from another VM inside of the same availability zone in order to reduce the impact of factors not under our control (like Internet traffic worldwide).

The metric value is the number of requests per second, since this is a simple CRUD webservice, neither CPU-bound nor memory-bound.

It runs on EC2 (the backend), while a more intensive task is on AWS Step Functions running Lambdas.

When testing on a single VM, before defining the Load Balancer scaling strategies, we run our tests from another VM on the same availability zone, contacting the target VM using its own private IP address, without going outside the AWS network.

We wanted to observe what is the maximum number of requests per minute that a single VM instance can reasonably handle. To do that, we developed a custom tool in *Golang*, to be able to send a specified amount of requests, uniformly distributed in the time unit (a minute). This tool, every $\frac{60{,}000}{n}$ ms, asynchronously sent a new request to the application server. Here, the fact that it is asynchronous (using the `go` keyword in Golang) is crucial to ensure that, no matter how long the previous request took to execute, or if it is still running, the new one will be fired at the right time. Collected metrics are the number of failed requests and the average time they took (also calculating their standard deviation). A request is considered *failed* if the HTTP client library reported an error, the status code was not 200 or if it took more than the timeout, set to 1 second.

At each number of requests N, we run the benchmark and waited for 3 minutes between each run, not to let one test influence the others. The test was repeated 3 times, always returning almost the same result, which is summarized in the plot 3.

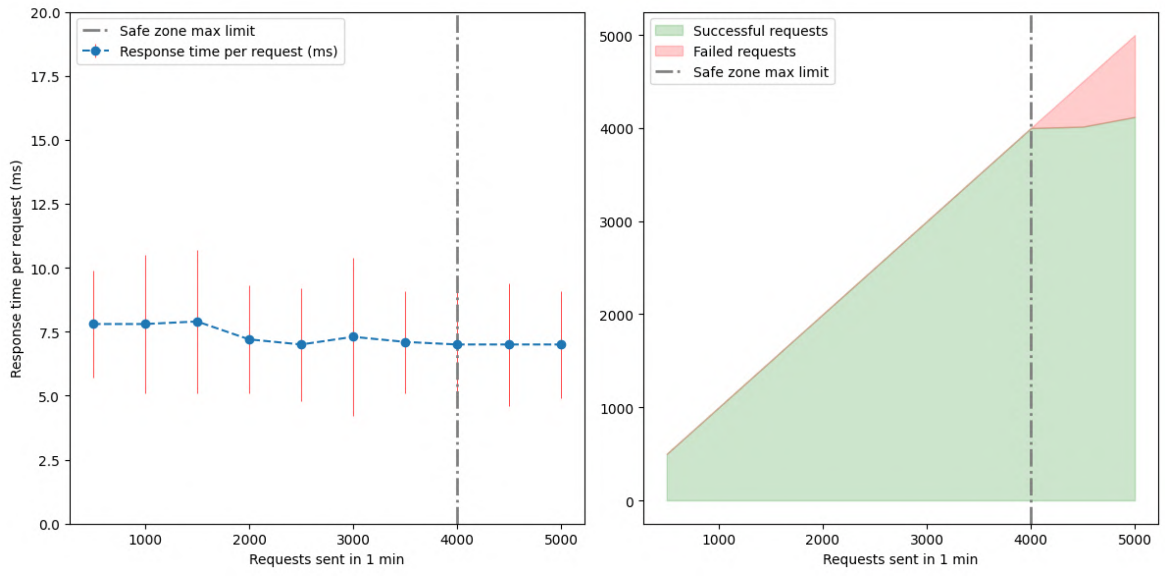


Figure 3: Benchmark of a single VM instance

It's clear how the limit is **4000 requests** per minute.

However, we would like to scale a bit earlier than reaching the maximum limit, ideally scaling up when reaching $> 60\%$ and scaling down when $< 30\%$. That being said, our policy is the following:

Metric value (req/min)	**Percentage adjustment**
Req <1200	-1 VMs
1200 <Req <2400	0 (ideal condition)
CPU >2400	+1 VMs

We have not set the policies for CPU and memory usage, since the main bottleneck of this application is the network and requests handling. We neither use the network traffic instead of the number of requests because we have observed that the only network-intensive operation (Photo Upload) does not represent a bottleneck: doing a test on this specific endpoint only, we observed that it flawlessly handles the same amount of parallel requests, and even many more.

Since the operation to scale up takes time (a new VM must be booted, even if all the software is already pre-installed in the custom AMI), we imposed that we scale down only after 2 consecutive data points indicating that we have a VM capacity that is too high, while we need just one data point reporting too many requests to scale up.

The *warm-up period* when scaling up was set to the default value of 240 seconds, since by experiments we observed that this is a reasonable amount of time to let the new VM boot up and be ready to serve incoming requests. When scaling down, a warm-up period of 180 seconds was enough to let the system react.

4.1 User activity as a Markov chain

We can create a custom piece of software in Golang, similar to the benchmark mentioned before, but this time we have several users emulated designing a **Markov Chain** to indicate the actions that a user may take, with probability p_{ij}, in screen i to state j.

The time between one state and the next one is 1 second, because it's reasonable for a user to actually wait that amount of time before doing another action (for instance, to read what appeared on the screen).

The *Upload Photo* state is particular because, as in the real application, the user is forced to do polling of the state of the upload processing every 5 seconds, until it goes to an error or a success state.

We only test the features that imply an HTTP request to our target backend, running on EC2. For example, the fetch of a single photo is done directly by getting the file from the appropriate S3 bucket (thumbnails or compressed high quality photos), so it is not object of our load tests.

The final behaviour of the typical user is summarized in the diagram 4.

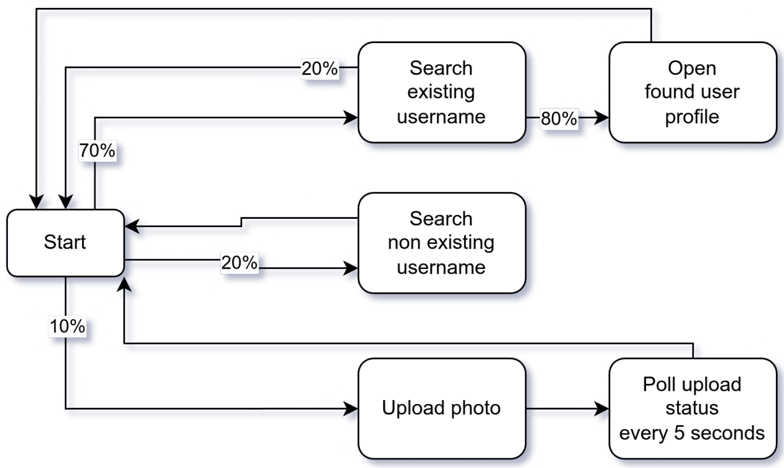


Figure 4: Markov chain representing user behaviour

4.2 Testing auto-scaling

Having the Markov model, it is simply a matter of implementing it in our favourite language for benchmarks (Golang) and running it.

We increase the number of users gradually: first we simulate 20 concurrent users, then increase until a maximum point (**100 users**) and finally slowly decrease down to 20 users again. This is the simulation of a typical scaling situation.

Since we built the custom simulation tool to simulate 20 users per run, we can split the simulated audience on a pair of VMs in the same availability zone, so that the bottleneck won't become the VM running the benchmark itself. The statistics are of course summed up across all the instances.

A function models the user, and we can run it a few thousands of times, in parallel thanks to the `go` keyword provided by the language, and observe how the application responds.

If we assume that, on average, each user does 1 request every second (because every state represents a request), we will have $\approx$ **6,000 requests** per minute on the peak of our load test.

From these experiments, we want to calculate the same metrics as before, from the client side:

- Number of successful requests, *per minute*
- Mean of the time that each request took for a response to arrive

Additionally, from the Cloud side, we collect:

- Number of running VMs
- Number of stopped VMs [1]

[1]We keep 1 VM ready in the Warm Pools to make the scale-up much faster

- CloudWatch metric `RequestCountPerTarget`, on which alarms are based

The tests are, as always, run from another VM internal to the AWS network and **in the same availability zone** in order to avoid having the negative impact of the Internet.

Talking about the bandwidth usage on the VMs, we observed that it was still in acceptable ranges, even when the VM had to scale up. So this confirms that, for our specific application, this does not represent a bottleneck. Neither the disk utilization, since absolutely no data is stored on the disk, but only on S3 buckets and Amazon DynamoDB.

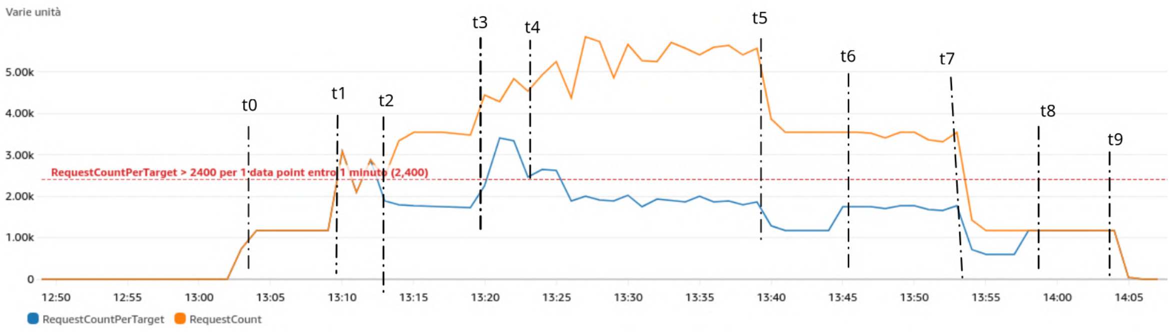


Figure 5: CloudWatch reported RequestCount metric during various phases of load test

In plot 5, we can observe the steps performed during the run of the load test, for a total of **an hour** of continuous requests, sent according to the user simulation discussed in section 4.1.

0. The test started with 20 users
1. The users increased from 20 to 60
2. The system responded, launching a new VM. We observe that now the requests (orange line) are divided by different targets (blue line)
3. The users increased from 60 to 100
4. A new VM was automatically launched by the Autoscaling group, reacting once again to our Cloudwatch Alarm (red line)
5. The users decreased from 100 to 60
6. A VM was stopped. We observe the requests handled by each VM to increase after the shutdown of one of them
7. The users decreased from 60 to 20
8. Now, only a single VM was still running. The orange and blue lines are matching again

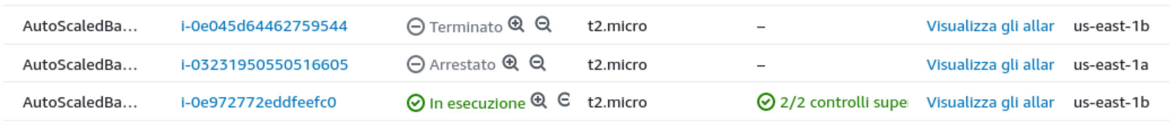
9. The test ended

Summarizing:

- From t_0 to t_1, we have the **warm-up** period
- From t_1 to t_3, we have the **ramp-up** period
- From t_3 to t_5, the load is **stable** at its peak of 100 concurrent users
- From t_5 to t_9, we finally have the **ramp-down** period, where we scale down to the minimum of 20 concurrent users

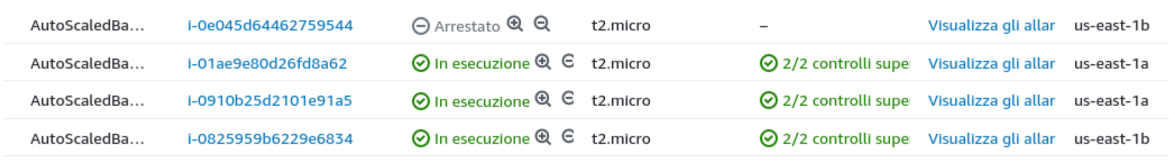
Finally, some key observations in conclusions are that:

- Since we created a Warm Pool in AWS Autoscaling, there was always a single VM in a Stopped state, ready to be turned on quicker
- The amount of failed requests reported by the client (simulator running in the same AWS Region), during the scaling-up period, was between 5% and 10%; more importantly, we observed that the response time increased from ≈ 20 ms to over 3 seconds in the load peak moments
- As seen in 6.2, the VMs were being created in the 2 Availability Zones, alternately



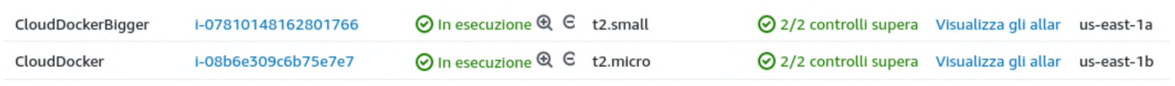
AutoScaledBa...	i-0e045d64462759544	Terminato	t2.micro	–	Visualizza gli allar	us-east-1b
AutoScaledBa...	i-03231950550516605	Arrestato	t2.micro	–	Visualizza gli allar	us-east-1a
AutoScaledBa...	i-0e972772eddfeefc0	In esecuzione	t2.micro	2/2 controlli supe	Visualizza gli allar	us-east-1b

Figure 6: State of EC2 machines after the test



AutoScaledBa...	i-0e045d64462759544	Arrestato	t2.micro	–	Visualizza gli allar	us-east-1b
AutoScaledBa...	i-01ae9e80d26fd8a62	In esecuzione	t2.micro	2/2 controlli supe	Visualizza gli allar	us-east-1a
AutoScaledBa...	i-0910b25d2101e91a5	In esecuzione	t2.micro	2/2 controlli supe	Visualizza gli allar	us-east-1a
AutoScaledBa...	i-0825959b6229e6834	In esecuzione	t2.micro	2/2 controlli supe	Visualizza gli allar	us-east-1b

Figure 7: State of EC2 machines during peak load of the test



CloudDockerBigger	i-07810148162801766	In esecuzione	t2.small	2/2 controlli supera	Visualizza gli allar	us-east-1a
CloudDocker	i-08b6e309c6b75e7e7	In esecuzione	t2.micro	2/2 controlli supera	Visualizza gli allar	us-east-1b

Figure 8: EC2 instances used to perform the tests (in different AZs)

5 Lambda scaling tests

More test were made, especially on the serverless part of the project, since we would like to demonstrate that the step function cannot be the bottleneck of the project. The test were conducted before using a custom tool from the same availability zone of the other resources and then using Jmeter from the same AZ. The change in the environment was due to the strict policies of AWS that banned our accounts. The goal of these tests is to collect metrics such as:

- The number of invocations of the step function, which aims to be better of the main test since we were limited by the maximum scaling set of EC2 instances in the main tests.
- The duration of the step function, so the average, minimum, and maximum amount of time our function code spends.
- Concurrent executions, so the number of function instances that are processing events

Since the test this time is simple from the point of view of the user, because we are not interested in choosing different operations (the step function is one) we decided to give the possibility of choosing the photo to upload because bad content could trigger the model provided by Amazon Rekognition. As previously said the request is not considered failed if the request detected bad content, but it is recovered by deleting the thumbnail and reporting the problem in the database to inform the user.

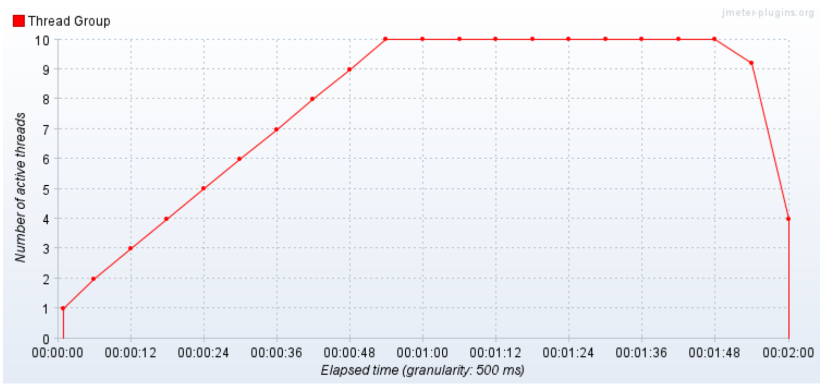


Figure 9: Active threads over time

The idea of the test was to start with 10 concurrent users and then scale reaching and then overtaking the result of the previous test. This time we expected to have fewer concurrent users since during the main tests they could choose between different actions. As we can see from 9, the test failed due to a ban, but actually, we recovered some information from the Jmeter output file, and the few taken from Cloudwatch before the page was actually unavailable.

# Samples	Avg	Min	Max	Error %	Throughput	Received Kb/s	Sent Kb/s	Avg. bytes
16	12,8	4,9	91,4	0.00%	0.135	0.59	0.65	448.9

In 5 we can see the summary report from Jmeter, which reports about:

- the average, minimum and maximum elapsed time for the request
- the throughput as request per second
- the Standard Deviation of the sample elapsed time
- Kilobytes received per second
- Kilobytes sent per second
- Average size of the sample response in bytes.

6 Availability tests

6.1 Step Function

Since all states in a step function, except Pass and Wait states, can encounter runtime errors, we implemented error handling to recover each possible fail.

6.1.1 Retriers

When a state encounters an error and includes a Retry field, Step Functions checks each retrier in the order they appear in the array. If the error name matches any listed in a retrier's ErrorEquals field, the state machine will attempt retries according to the specifications in the Retry field. The step function has a default retrier that matches errors such as TooManyRequestsException or AWSLambdaException and retries after some seconds where at each attempt increases the time of the next one.

6.1.2 Catchers

Step function allow to define a fallback state that when an error arise and either there is no Retry field, or if retries fail to resolve the error, tries to recover the error. As we can see in 10, the error that arose from the first lambda is solved in the fallback state. Since the protocol is to rollback the operation and store the error in the database to notify the user of the problem, each lambda has the same fallback state.

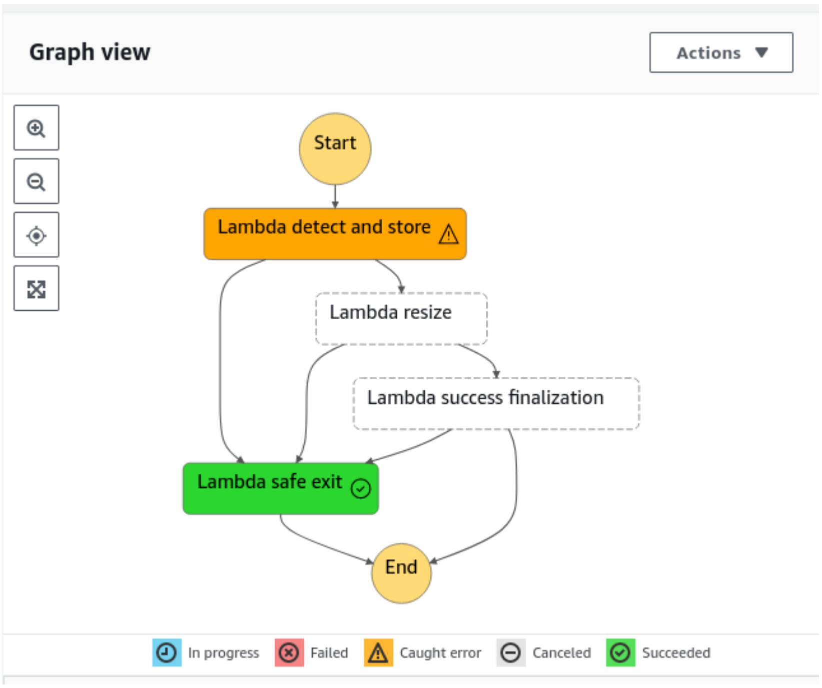


Figure 10: State machine of an errored step function

6.2 EC2 Backend

To make the service more available, we decided to assign to the AWS Autoscaling group more than just one Availability zone.

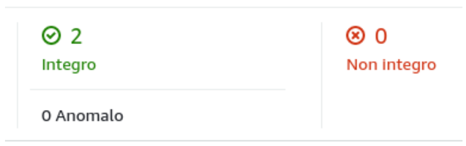
This way, by spreading EC2 instances across multiple availability zones, we wanted to ensure that, if one AZ experiences an outage or failure, our application server remains online in other AZs. This redundancy is an important aspect of being highly available.

More specifically, just for the sake of testing, we used 2 Availability Zones (*us-test-1a* and *us-test-1b*). The EC2 instances were all of type **t2.micro** to save costs.

Another different test for the Availability of the application server, is that in case of a failure of a VM, we should be back online as fast as we can. In this situation, we **manually stopped** a VM, simulating a crash, and observed how the AWS Autoscaling group will proceed in recreating another one in our *EC2 Target Group*, because one was reported as faulty.

Also, internally to the VM, the backend service runs inside a Docker container for simplicity in its configuration. This container was launched using the `--restart unless-stopped` flag. In this way, in the case of a crash specifically inside the Docker container, the Docker daemon proceeds to restarting it.

Finally, in addition to this checks, our application server exposes an HTTP Endpoint `/healthcheck` that, set up in the configuration of the AWS Load Balancer, makes possible for ELB (Elastic Load Balancer) **not to redirect requests** to our faulty instance. In fact, the healthcheck endpoint, performs a standard search in the Database, making sure that it works fine, and returning HTTP status code 200 OK. In case of failure, it will propagate the exception up to returning HTTP status code 500 Internal Server Error to ELB.

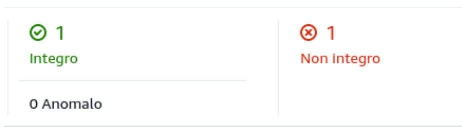


✓ 2
Integro

0 Anomalo

⊗ 0
Non integro

Figure 11: Target Group status before the test

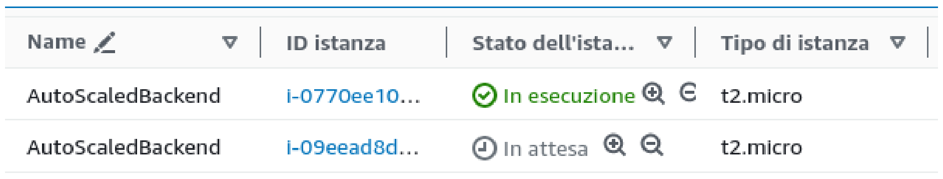


✓ 1
Integro

0 Anomalo

⊗ 1
Non integro

Figure 12: Target Group status after manually stopping HTTP proxy on the VM



Name	ID istanza	Stato dell'ista...	Tipo di istanza
AutoScaledBackend	i-0770ee10...	✓ In esecuzione	t2.micro
AutoScaledBackend	i-09eead8d...	In attesa	t2.micro

Figure 13: Target Group automatically created another healthy VM