



SAPIENZA
UNIVERSITÀ DI ROMA

Project for Cloud Computing

Multimodal Interaction & Computer Vision

GAiLLERY

Deploying a scalable and highly available AI
application on AWS

June 2024

Ángel Sánchez Guerrero
Matricola 2117499
sanchezguerrero.2117499@studenti.uniroma1.it

1. Problem addressed

The project addresses the development of a scalable and highly available AI web application on Amazon Web Services (AWS). This application, an AI-enhanced photo gallery, integrates image and speech recognition to provide an interactive user experience. AWS plays multiple critical roles in this application, including web and container hosting, computational power, storage solutions, developing environment, content delivery network, but the most relevant for this report will be scalability and high availability, i.e., efficient handling of high traffic and data loads, maintaining performance and availability at scale.

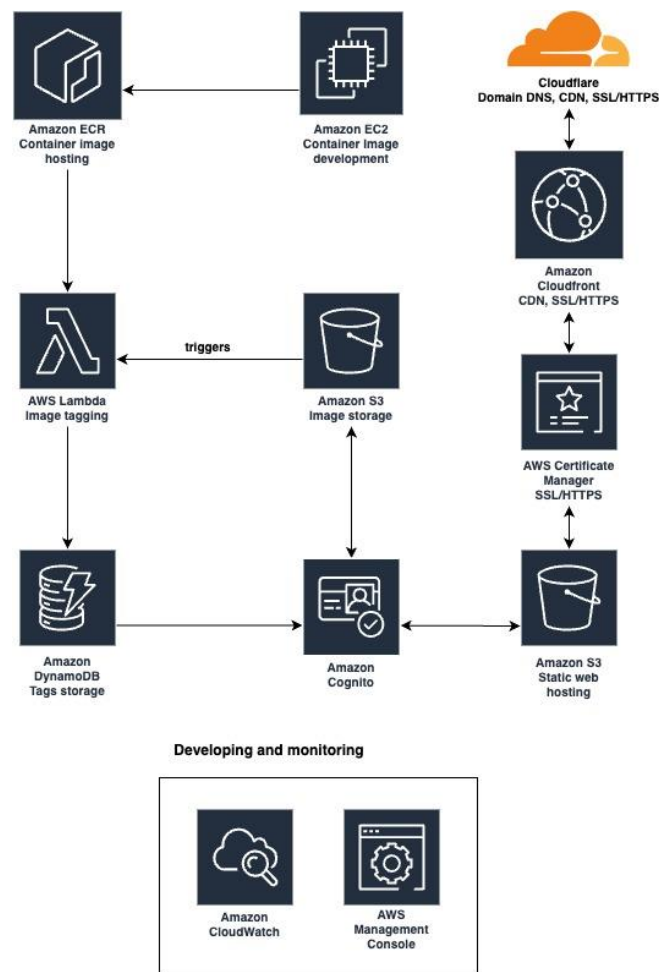
The decision to focus on this particular problem was influenced by my involvement in other projects from computer vision and multimodal interaction courses. Recognizing the overlap in skills and technologies these areas require, I saw a valuable opportunity to undertake a multidisciplinary project that combines these domains. This approach not only allow me to address the academic requirements of my coursework through a unified platform but also enhanced the practical utility and performance of the application. By merging these technologies, the project serves as a comprehensive showcase of integrating multiple disciplines in a real-world setting.

2. Designed solution

As this is a multi-course full-stack project there were a lot of issues to address. In this report I will be focusing on the Cloud Computing related aspects, and not for example in how the image recognition model was trained or what libraries were used for the UI.

- Static website hosting: AWS S3 Bucket
- Static website deploying from GitHub to S3: ASW CodePipeline
- Image files storage: AWS S3 Bucket
- Image files accessing: AWS Cognito
- Image tags database: AWS DynamoDB
- Image classification: AWS Lambda
- Container image hosting: AWS Elastic Container Registry
- Content Delivery Network: AWS CloudFront
- SSL/HTTPS: AWS Certificate Manager

This is a diagram for the whole infrastructure design:



3. Implementation process

At first, I wanted to utilize the AWS Learner Lab credits available from my cloud computing course. However, these credits were restricted to only certain services. To ensure full flexibility and control over the project, I decided to use my own AWS account and cover the minimal charges.

The first step was setting up the website. This process was very straightforward: I created a new S3 bucket and enabled static website hosting on the bucket, setting the correct permissions to make the website publicly accessible.

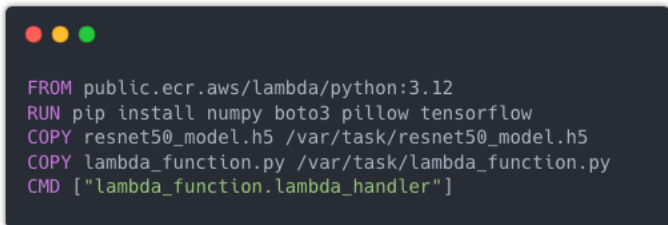
Next, I needed a method to deploy updates from my local environment to the bucket automatically while saving changes to my GitHub repository. For this, I set up an AWS CodePipeline. This was done by connecting my GitHub repository to CodePipeline, which automatically pulls the latest changes from the repository and deploys them to the S3 bucket whenever I push new code.

After creating a simple gallery website and testing it locally using images from my computer, I created another S3 bucket dedicated to storing these images.

To access these images from the website, I created an AWS Cognito Identity Pool with the correct IAM roles to ensure appropriate access permissions. Because this is just an educational purpose project, no further identification is required, anyone right now can upload and remove files from the bucket or database.

Once the functionalities for uploading, listing, and removing images were done, I focused on integrating a machine learning model using AWS Lambda. The objective was for each uploaded image to trigger a serverless function that would classify the image and store the results. However, the model environment required substantial storage, exceeding the capabilities of AWS Lambda. I resolved this by utilizing AWS Lambda's support for container images, allowing me to run the model within a container as a serverless function.

Here's how the *Dockerfile* is structured:

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays the content of a Dockerfile in a light-colored monospace font.

```
FROM public.ecr.aws/lambda/python:3.12
RUN pip install numpy boto3 pillow tensorflow
COPY resnet50_model.h5 /var/task/resnet50_model.h5
COPY lambda_function.py /var/task/lambda_function.py
CMD ["lambda_function.lambda_handler"]
```

The *lambda_function.py* script is designed to download the image that triggered the function, classify it, and then store the top three predictions along with their respective confidence percentages in a DynamoDB Table.

I encountered significant challenges in building the container due to the ARM CPU architecture of my computer, which was incompatible with the required processes. To resolve this, I spun up a *t2.small* EC2 instance (as *t2.micro* hadn't sufficient memory to handle the container build) that allowed me to download the model, build the container image, and then upload it to the AWS Elastic Container Registry. Then, in AWS Lambda I only had to choose the image as the main function and re-configure the timeout limit (3 min), the memory allocation (3GB) and ephemeral storage (3GB). Note that AWS allocates CPU power proportionally to the memory configured, so I would have chosen more memory in order to make the computing faster, but it is capped at 3GB for me. Some people on the internet say it depends on the region, others that new accounts have restricted quotas.

When all of the main functionality was working, the remaining thing to do was to link my custom domain and enable HTTPS. To do this I used AWS CloudFront (which also provides CDN) and AWS Certificate Manager to create the encrypting certificate.

4. High availability and scalability testing

High availability

It refers to the ability of a system to operate continuously and reliably over long periods of time, minimizing downtime and maintaining business continuity even in the face of failures and disasters. The core services used in this project are all architected to inherently manage it.

- AWS Lambda promotes high availability by automatically managing compute capacity across multiple Availability Zones within each region where it is deployed. This design means that Lambda functions are inherently fault tolerant. They operate across multiple zones and are engineered to withstand failures of individual servers or entire data centers within the region.
- AWS S3 offers exceptional durability and availability, with design specifications promising 99.999999999% durability and 99.99% availability. It achieves this by storing data across at least three geographically dispersed facilities within an AWS Region. S3 handles all the complexities involved in data replication, node failures, and other potential fault conditions autonomously.
- AWS DynamoDB also automatically replicates data across three Availability Zones within a region. It is built to offer fault tolerance and automatic data replication, which makes it an ideal solution for managing state information with high reliability and performance.

Scalability

Scalability is the capability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged to accommodate that growth. In AWS, scalability can be achieved through a variety of services designed to handle increased load by adjusting resources either automatically or on-demand. General methods of achieving scalability in AWS include:

- AWS Auto Scaling: Automatically adjusts the number of server or container instances based on the defined conditions and load.
- AWS Elastic Load Balancing: Distributes incoming traffic across multiple targets, in multiple Availability Zones, which increases the fault tolerance of applications.

In the context of this project, because we don't deal with EC2 or direct server management, scalability is also inherently managed by AWS. These services are designed to scale automatically, handling increases in demand by leveraging AWS's extensive infrastructure without the need for manual configuration.

- AWS Lambda: Scales automatically by running code in response to each trigger. The number of triggers (e.g., file uploads to an S3 bucket) can increase substantially, but Lambda can handle these increases by launching as many copies of the function as needed to process the events concurrently. On the other hand, there is a limit of 10 concurrent executions for new accounts. Although this is, in fact, a problem for scalability, a limit increase can be requested to Amazon.
- AWS S3: Automatically handles scaling through its distributed architecture. When demand increases, such as during peak loads for special events, S3 spreads the load across its vast network of servers and facilities. AWS minimizes latency, maximizes throughput, and avoids bottlenecks behind the scenes, ensuring users experience fast and reliable access to the website's content.
- AWS DynamoDB: It also automatically adjusts its capacity to handle varying loads through autoscaling, dynamically adapts read and write capacities based on actual usage. The only thing I had to do is switch the Capacity mode from "Provisioned" to "On-demand".

In conclusion, the project utilizes the scalable and highly available architecture of AWS out of the box. This approach simplifies operations and is likely more cost-effective than managing traditional server instances. These services automatically adjust to traffic and load, ensuring that we only pay for the resources we use while maintaining optimal performance.

Although these services are designed to scale effectively, it is critical to verify the system's capacity to handle unusually high loads to ensure that the architecture operates reliably under unexpected or extreme conditions.

4.1 Experimental design

The primary objective of these tests is to validate that the application can handle increased load without problems. One of the main challenges is how to reproduce user behavior and select which actions to reproduce. In my application there are two big main actions which will be tested:

- Task 1: Enter the website. Request the static files to the web S3 bucket via HTTP. Then, all the contents in the image bucket and the tags in the database are loaded.
- Task 2: Upload an image. Uploading files to the image bucket triggers the Lambda function, and therefore DynamoDB too.

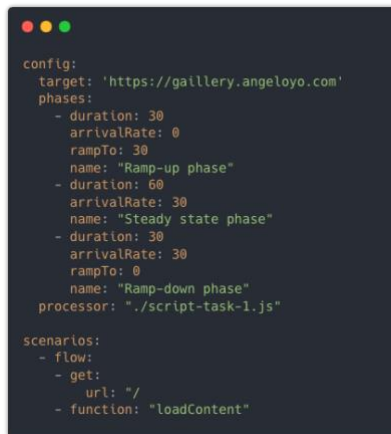
Tools and Resources:

- Load Testing Software: Artillery will be used to simulate web traffic and interactions with the AWS services.
- Monitoring Tools: AWS CloudWatch will be utilized to monitor system metrics.

The tests are composed of the following phases:

- Ramp-up (RU) period, load increase until a certain amount
- Steady (S) period, load is stable at the max value
- Ramp-down (RD) (to test scale down)

For the first task, because the images and tags on the website are retrieved using JavaScript, and Artillery doesn't execute it by default, I had to create code that simulates this behavior. This is how my artillery configuration file looks like for testing the first task:



```
config:
  target: 'https://gallery.angeloyo.com'
  phases:
    - duration: 30
      arrivalRate: 0
      rampTo: 30
      name: "Ramp-up phase"
    - duration: 60
      arrivalRate: 30
      name: "Steady state phase"
    - duration: 30
      arrivalRate: 30
      rampTo: 0
      name: "Ramp-down phase"
  processor: "./script-task-1.js"

scenarios:
  - flow:
    - get:
      url: "/"
      function: "loadContent"
```

The test flow consists of first getting the page via standard HTTP request, and then execute the custom function *loadContent* that also gets images and tags from S3 and DynamoDB.

For the second task another similar Artillery configuration will be used, with a different custom function that uploads a random number of images to the S3 Bucket. Each one of the uploads will trigger the lambda function that tags them and store the results in DynamoDB.

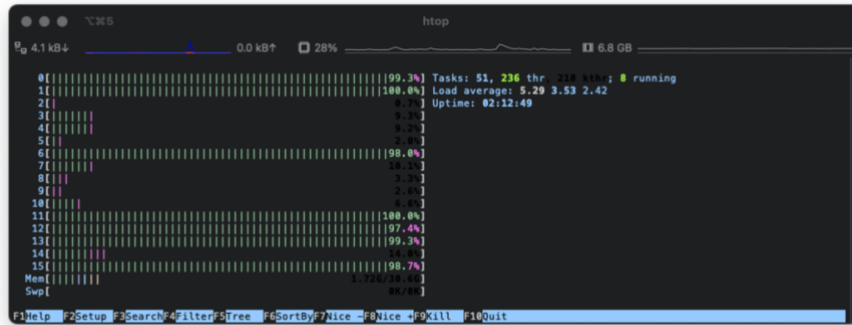
All the used code is available in the [GitHub repository](#).

4.2 Experimental results

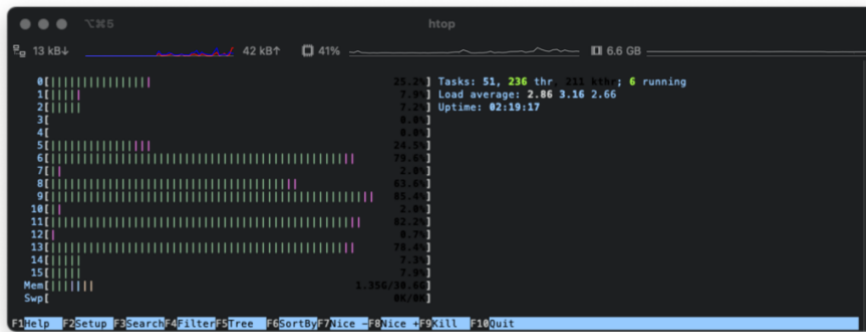
Task 1

To avoid potential bans due to high traffic, the tests were run on an EC2 instance. At first, because the testing involved handling a large amount of image files in a short time, the CPU usage of the machine was a bottleneck problem. I attempted to enhance performance by increasing the number of cores of the instance, but unfortunately, I was limited by an account restriction to 16 vCPUs, so I used a *c5.4xlarge* instance. With this drawback I had to find the bottleneck point and not surpass it. Checking the CPU usage all the time, I discovered the bottleneck was between 5 and 6 virtual users per second. In this

image, CPU usage for a test with 7 virtual users per second. We can clearly see the bottleneck with some cores hitting 100%.



This is CPU usage with 5 virtual users per second. Although the load is still high it doesn't bottleneck.



The difference is clear when you look at the obtained test metrics.

TESTS	1	2	3	4	5	6	7	8	9	10
http.codes.200	686	594	594	500	500	500	500	500	500	1000
http.request_rate	2/sec	2/sec	2/sec	2/sec	2/sec	2/sec	2/sec	2/sec	2/sec	1/sec
http.requests	686	594	594	500	500	500	500	500	500	1000
http.response_time_min	0	0	0	0	0	0	0	0	0	0
http.response_time_max	56	13	10	8	15	19	11	46	7	12
http.response_time_mean	3,9	1,2	1,2	1,2	1,3	1,2	1,2	1,4	1,2	1,3
http.response_time_median	1	1	1	1	1	1	1	1	1	1
http.response_time_p95	19,1	2	2	2	2	2	2	2	2	2
http.response_time_p99	40,9	3	3	3	4	3	3	7,9	3	6
http.responses	686	594	594	500	500	500	500	500	500	1000
vusers.completed	686	594	594	500	500	500	500	500	500	1000
vusers.created	686	594	594	500	500	500	500	500	500	1000
vusers.created_by_name.0	686	594	594	500	500	500	500	500	500	1000
vusers.failed	0	0	0	0	0	0	0	0	0	0
vusers.session_length_min	378,5	390	375,1	381,7	376,4	378	380	395,3	390,7	380
vusers.session_length_max	3866,1	1033,7	1772,1	908,7	1561	1629,7	1646	868	848,2	1163,4
vusers.session_length_mean	1923,2	880,5	835,9	735,7	715,9	778,3	739,6	737,8	733,4	722,3
vusers.session_length_median	2143,5	925,4	871,5	772,9	742,6	820,7	772,9	772,9	757,6	757,6
vusers.session_length_p95	3072,4	982,6	944	820,7	820,7	871,5	820,7	837,3	820,7	804,5
vusers.session_length_p99	3534,1	1022,7	982,6	837,3	907	907	854,2	854,2	837,3	820,7
Warmup										
duration	30	30	30	30	30	30	30	30	30	60
arrivalRate	0	0	0	0	0	0	0	0	0	0
rampTo	7	6	6	5	5	5	5	5	5	5
Steady state phase										
duration	60	60	60	60	60	60	60	60	60	120
arrivalRate	7	6	6	5	5	5	5	5	5	5
Ramp-down phase										
duration	30	30	30	30	30	30	30	30	30	60
arrivalRate	7	6	6	5	5	5	5	5	5	5
rampTo	0	0	0	0	0	0	0	0	0	0

Despite these limitations and constraints, the test results demonstrated that my solution could scale up to meet increased demands, showing no errors and a good response time.

Task 2

In the initial test run, each virtual user uploaded five images to verify the correct creation of objects in S3 and DynamoDB. Once confirmed, I modified the script to upload a random number of images between 1 and 5 to simulate user behavior more accurately. Unlike the tests for the previous task, these did not consume as much CPU capacity. I began with five virtual users per second and gradually increased this number until I encountered the bottleneck. Here are the results of these experiments:

TESTS	1	2	3	4	5	6	7	8	9	10	11
vusers.completed	500	500	594	686	950	1196	2250	4500	9000	13500	18000
vusers.created	500	500	594	686	950	1196	2250	4500	9000	13500	18000
vusers.created_by_name.0	500	500	594	686	950	1196	2250	4500	9000	13500	18000
vusers.failed	0	0	0	0	0	0	0	0	0	0	0
vusers.session_length_min	75,9	38	33,9	33,7	37,1	35,4	32,5	32,4	33,5	34,7	32,6
vusers.session_length_max	934,8	406	225,5	637	233,1	249,9	276	909,5	330,7	625,9	1942,5
vusers.session_length_mean	128,5	96,4	90,4	88,2	87	85,6	85,1	83,5	92,3	141,3	569
vusers.session_length_median	122,7	96,6	90,9	87,4	87,4	87,4	85,6	83,9	92,8	133	632,8
vusers.session_length_p95	172,5	138,4	133	120,3	125,2	117,9	122,7	120,3	135,7	242,3	925,4
vusers.session_length_p99	247,2	172,5	165,7	194,4	172,5	165,7	179,5	165,7	190,6	320,6	1130,2
Warmup											
duration	30	30	30	30	30	30	30	30	30	30	30
arrivalRate	0	0	0	0	0	0	0	0	0	0	0
rampTo	5	5	6	7	10	13	25	50	100	150	200
Steady state phase											
duration	60	60	60	60	60	60	60	60	60	60	60
arrivalRate	5	5	6	7	10	13	25	50	100	150	200
Ramp-down phase											
duration	30	30	30	30	30	30	30	30	30	30	30
arrivalRate	5	5	6	7	10	13	25	50	100	150	200
rampTo	0	0	0	0	0	0	0	0	0	0	0

It is evident that at a rate of 200 virtual users per second, the CPU was unable to keep up. But the big issue was the limit of 10 concurrent Lambda executions for new accounts, leading to a queue that prevented a clear view of the system's scalability.

Despite these challenges, the test results confirmed that my solution was capable of scaling effectively to meet increased demands. Even when pushing the system to the current setup's limits, the system remained operational and responsive. The Artillery tests concluded without any errors from virtual users, CloudWatch metrics verified a 100% success rate for Lambda executions, and DynamoDB reported no errors either.

GitHub repository: <https://github.com/angeloyo/sapienza-project>
 Project website: <https://gaillery.angeloyo.com/>