Serverless Image Editor with AWS Lambda

Antonio Cordeiro, *1999975*
Lorenzo Ugolini, *1958654*

1 Introduction

For this project we developed a serverless Image Editor Web Application leveraging AWS Cloud Services. The users can upload an image to the website and make some edits to it like applying a grayscale filter, blurring or resizing. We implemented a simple frontend with HTML, CSS and Javascript. The backend is implemented with AWS Lambda functions written in Python 3.12. The API calls between frontend and backend are managed by API Gateway. The images the user uploads and modify are temporarily stored in Amazon S3 Buckets.

The main goal of the project is to test the scalability of an image processing serverless application. The workload to test the performance of the app was generated with a Selenium script in Python. We kept track of interesting metrics with Amazon Cloud Watch. The results show that our application ensures correct scalability and availability with multiple concurrent users.

2 The Tools

2.1 Frontend

The frontend is very simple. The user can upload a *.jpeg* image and there is a button for each edit the user can make to the picture, in our case apply a black and white filter, blur, sharpen, and resize. Once the user applied at least one modification to the image, he also has the possibility to revert the last operation. The GUI, Figure 1, is defined in HTML and styled with CSS, with Javascript code used to make the API calls to interact with the backend.

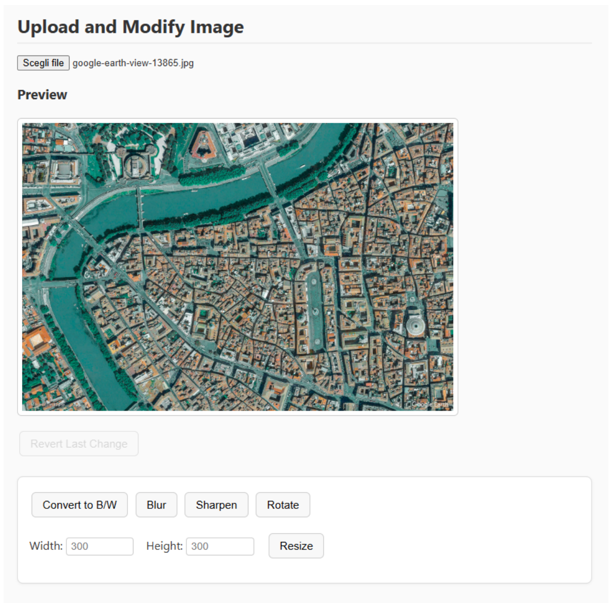

Fig. 1. The WebApp interface

2.2 OpenCV

OpenCV is an open-source computer vision library that provides a wide range of basic image processing functionalities, in our case:

- Grayscale conversion
- Blur: Gaussian blur with a 7x7 kernel
- Sharpen: 2D convolution with kernel [[0, -1, 0], [-1, 5,-1], [0, -1, 0]]
- Rotate: each rotation is of 90° clockwise
- Resize

These operations are of course only the foundation of more advanced computer vision tasks that can be implemented thanks to CV2.

2.3 AWS Services

The system uses different AWS Services:

1) **Amazon S3** (Simple Storage Service): Amazon S3 is an object storage service that offers scalability, data availability, security, and performance. In this work it has been used with various applications, like static website hosting, image storage and utility storage.

2) **Amazon API Gateway**: it is a fully managed service that makes it easy to create, publish, and manage APIs at any scale. It allowed us to implement a RESTful API to handle HTTP requests from the frontend and integrate the backend services, such as AWS Lambda functions. API Gateway simplified the process of exposing our application's functionalities while ensuring efficiency and scalability.
3) **AWS Lambda**: AWS Lambda is a serverless compute service that executes code in response to events or triggers. Overall we used four Lambda functions, triggered by various API endpoints requests.
4) **Amazon CloudWatch**: Amazon CloudWatch is a monitoring and observability service that helps tracking the performance and health of applications and infrastructure. It allows to collect and visualize metrics, set up alarms for critical events, and monitor log data in real time. It can be used to understand and respond to system-wide performance changes, optimize resource utilization, and get a unified view of operational health. We used it to keep track of the metrics of the system when testing its scalability performance.

As a final touch to our application we also considered implementing authorization service with AWS Cognito, but we weren't able to finalize the implementation because both our accounts have been banned due to testing activity.

3 Implementation details

In this section there is a description of the system implementation, represented in Figure 2.

The system starts with the user connecting on the static website hosted in a dedicated S3 Bucket. On the page he uploads a *.jpeg/jpg* image and can perform the operations listed in Section 2.
To perform all the intended operations it is necessary to upload the original image to a dedicated S3 Bucket. We decided to silently upload the image in the background when the user selects its file. At the moment of the upload, the JavaScript code turns the picture into Base64, and issues a POST request at the `/images/upload-image` endpoint; the endpoint triggers the **imageUploader** AWS Lambda function, which decodes the image and puts it in the *uploads* Bucket, changing the name to a generated UUID.

While the picture is being uploaded the action buttons are kept unclickable. As soon as the Lambda confirms the upload, returning the assigned ID, the user can start modifying the picture. All the buttons issue a POST request to the `/images/modify-image` endpoint, with a body containing the picture ID and the type of action to perform. The request triggers the **imageModifier** Lambda function; it takes the stored picture, modifies it, and loads it into a *modified* Bucket attaching a timestamp on the original ID. Before this, it must check where to retrieve the picture from: if the image has been already modified, it takes the latest image with the same prefix from the *modified* Bucket, otherwise takes it from the *uploads* Bucket. For our Lambda to work correctly, we had to create a package Layer, which we stored in a dedicated Bucket.

After having performed at least one modification the user is able to revert the last one made. The "Revert Last Change" button triggers a GET request on the `/images/get-image` endpoint, executing the **imageGetter** Lambda function, which retrieves the second most recent picture with the given ID, and at the same time deletes the most recent one. If the user has performed one single modification, the Lambda deletes the only entry in the *modified* Bucket, and returns the original image stored in the *uploads* one.

We wanted to avoid filling both our images buckets with unused and old images. For this reason, in the JavaScript code we added a `addEventListener` function for the `beforeunload` event. In case the user refreshes or closes the page the event is triggered and the code sends a DELETE request to the `/images/delete-images` endpoint. The endpoint triggers the **imageRemover** Lambda function: it simply looks for images with the given ID in both *uploads* and *modified* Buckets and removes them.

4 Scalability

To evaluate the performance and scalability of the application under varying workload conditions we developed a dedicated Python script leveraging Selenium. This script leverages the Selenium library to simulate user behavior. We had to be very careful during this process to avoid being banned by the AWS platform. Initially, we unsuccessfully attempted to execute the script within a virtual machine on Amazon EC2, due to memory limitations of the free-tier instances. Consequently, we run

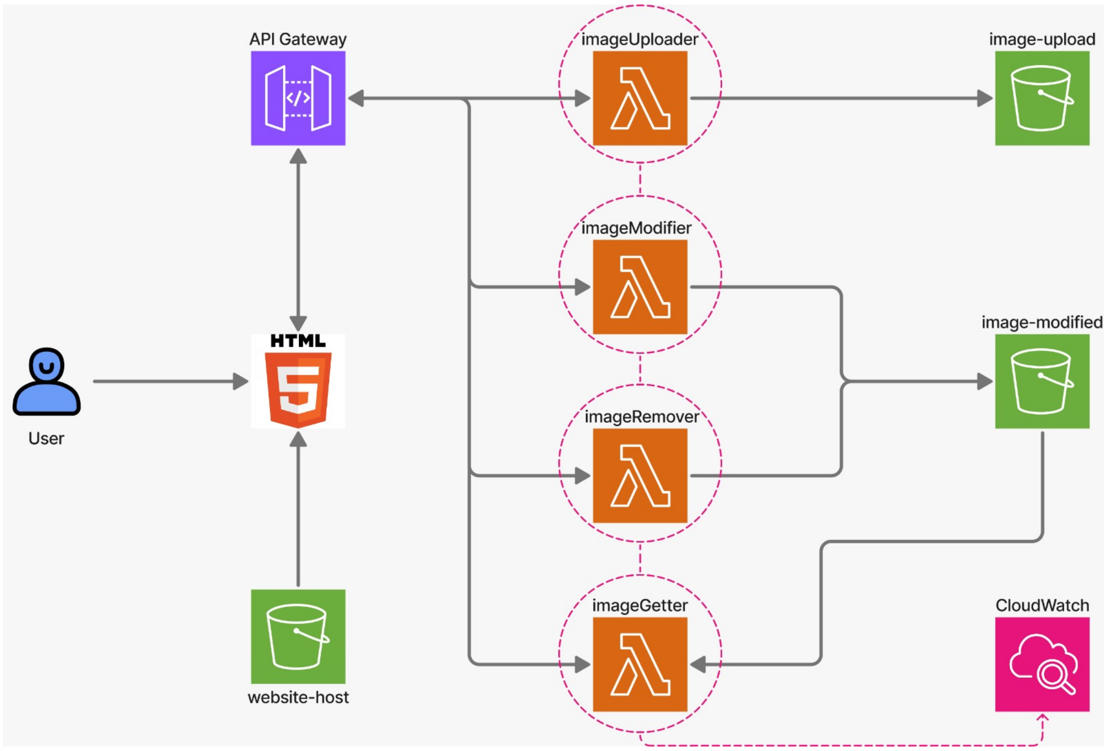

Fig. 2. The WebApp interface

the script locally on our machines to ensure sufficient resources for the simulated workloads. The testing methodology was divided into two distinct approaches:

1) Workload bursts
2) Incremental workload

In both settings, the simulated user workflow was the following: each user first uploads a single image and then performs 5 additional actions. The editing actions are chosen randomly with an equal probability. To accurately simulate real-world parallelism the script initiates a separate thread for each concurrent user. For the workload burst tests, the web app was tested with 20 concurrent users. The primary metric observed in this test was the average time required for each user to complete their full workflow, going from 5 seconds with 5 concurrent users, to around 11 when the users are more than 10.

For the incremental tests, we structured the workload to simulate a more gradual increase in user traffic. The test began with a warm-up period of 5 minutes, during which we maintained a steady load adding 5 new users per minute. Following this, a ramp-up phase started, where the number of new concurrent user per minute was increased every 5 minutes until it reached a peak of 15 users. The system then entered a steady period, adding 15 concurrent users every minute for 10 minutes to assess long-term high-load stability. Finally, the load was gradually scaled down until arriving back to 5 users per minute for 5 minutes. This detailed approach allowed us to observe the application's performance as it adapts to changing load conditions.

Our system showed to be able to easily undergo this type of workload, ensuring scalability and availability for the whole duration of the test. The most interesting insight are given by **CloudWatch** plots on our Lambda functions. Figures 3 and 4 accurately reflects the incremental workload: we can clearly see the warm-up, ramp-up and steady periods. Looking at the overall Lambda functions integration, Figure 6 the computing duration stays quite steady during the test, with the *imageModifier* being the most computationally expensive one. Focusing on the latter, Figure 7 also show changes in number of concurrent executions of the function. Looking at the average execution time of *imageModifier*, the orange line in Figure 5, we can deduce that our system was able to correctly distribute the bursts and steady workload instantiating a variable number of concurrent executions without significant degradation, arriving at peaks of 13 concurrent executions at the end of the steady period.

We wanted to push our system a little more,

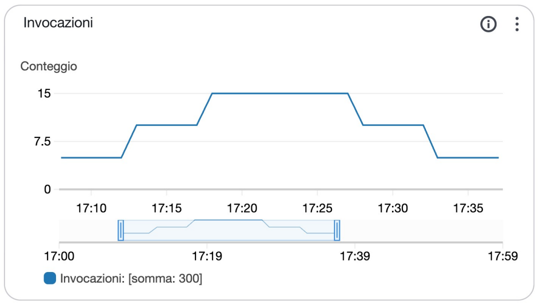

Fig. 3. Invocations of imageUploader

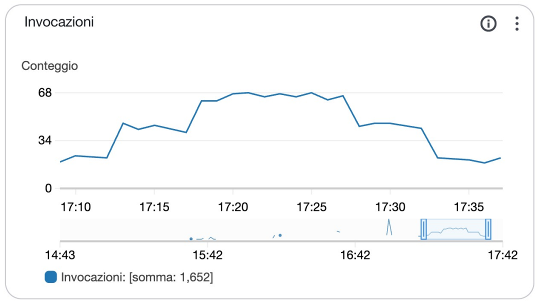

Fig. 4. Invocations of imageModifier

modifying the incremental test to have more than 20 new users per minute at its peak. Unfortunately, we were banned for this reason but still, we wanted to try out this test. Our idea was to launch an **EC2** instance in which we would run our code. We launched an instance of type *t3.large*, the biggest possible within the student Lab account, using the Ubuntu Amazon Machine Image. Unfortunately, our test is based on launching multiple concurrent threads, and this causes the EC2 instance, with limited resources, to crash. After a couple tries we understood this was not a feasible way.
Another option we explored was to implement our testing with **Locust**. Differently from Selenium, it works by directly issuing API requests. This is not ideal for our use case: the issue of specific API requests is dependent on other actions, so the testing method has to be able to correctly simulate the user behavior, interacting with UI elements.

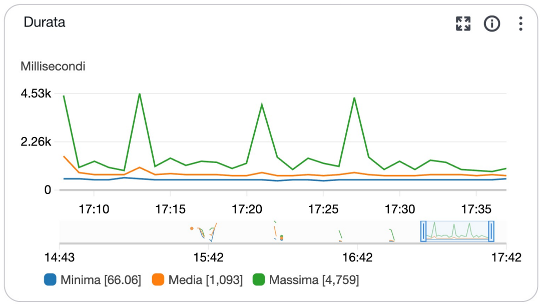

Fig. 5. Execution time of imageModifier

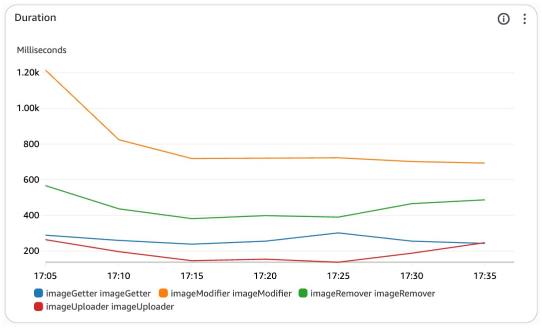

Fig. 6. Overall computing times of Lambda functions

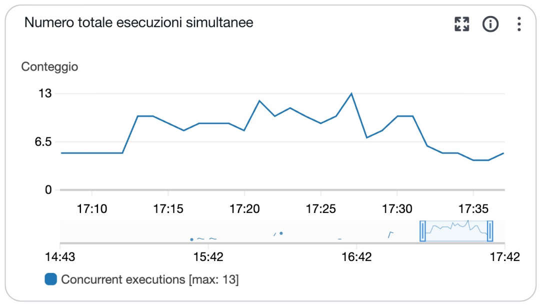

Fig. 7. Concurrent executions of imageModifier

5 Conclusions

In this project we developed and evaluated a serverless image editor web application using a cloud architecture. By leveraging key AWS services such as Lambda, API Gateway, and S3, we were able to create a scalable solution for image processing. The application's frontend, built with HTML, CSS, and JavaScript, provides a user-friendly interface for various image manipulations, including grayscale conversion, blurring, and resizing, exploiting the OpenCV library.

The core objective of this work was to test the scalability of our serverless system under different load conditions. Our testing, conducted with a Python and Selenium-based script, confirmed the ability of this type of system of handling multiple concurrent users effectively, ensuring availability. Monitoring with Amazon CloudWatch the workload tests highlighted how services quickly adapt to fulfill the computing necessities.

Finally, the adoption of a serverless architecture with pay-as-you-go model, ensuring efficient resource utilization without incurring in unnecessary expenses, allowed us to minimize the costs, saving the already limited Student Lab budget.

5.1 Possible Improvements

Looking away from the good results obtained by our systems, there are some possible improvements that can further enhance the performances.

First of all we can of course add new image processing functionalities, both from the OpenCV library and from other sources, for example simple AI models.

The most critic aspect of our project is that it is based on *synchronous* operations. All the actions take place at the moment they get requested. This is the major source of bottlenecks, in particular, for the great number of synchronous accesses to S3 Buckets. A possible improvement could be to insert an intermediary step between the request and the actual issuing of the computation; a possible option would include the use of **DynamoDB**: after the user chooses a modification, its details are uploaded in JSON format on Dynamo, and this is what triggers the `imageModifier` Lambda function. In this way we would have higher performance removing the synchronous operations, but of course it introduces big downsizes in the user experience.