Data Science

Serverless Equation Solver

Cloud Computing

Professors:

Emiliano Casalicchio

Students:

Susanna Bravi 1916681

Gianluca Procopio 1942103

Academic Year 2023/2024

Contents

1 Overview

This project is about the design and deployment of an application using the serverless computing paradigm.
We chose to develop an image-to-text application with the goal of taking an image of a mathematical equation and returning the result to the user.
The application is able to solve simple equations and inequalities of the first degree.
We relied mainly on the *amazon rekognition service* and the python *SymPy* library. The former tool was used to extract the text from the image while the latter was used to solve the equation.

1.1 Frontend

Through a simple graphical interface and without the need for the user to authenticate themselves, the user can upload his photo and get their result quickly and easily. We created the User Interface with HTML with embedded JavaScript code to allow the interaction with out backend.

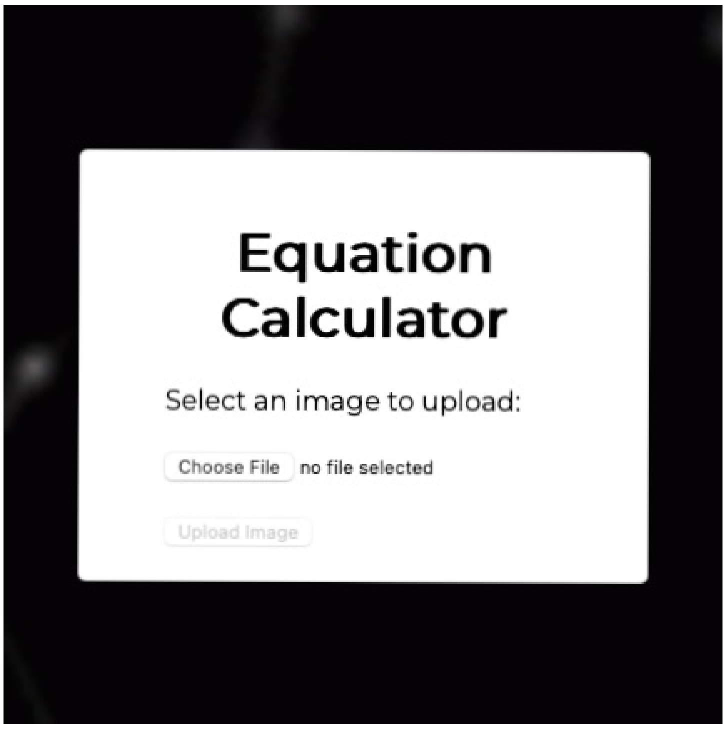


Figure 1: Screenshot of the application's interface

1.2 SymPy

SymPy is a Python library for symbolic mathematics, we took advantage of it for computing the solutions of our equations or inequalities. After a bit of processing of the retrieved text, Sympy is able to quickly and accurately compute the solution we are looking for.

1.3 AWS Services

We used different AWS services to develop the application

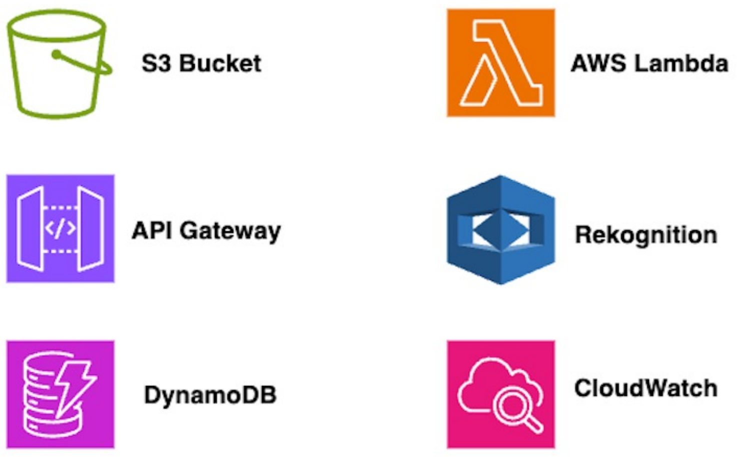


Figure 2: List of AWS services used

- **Amazon S3 (Simple Storage Service)**: Amazon S3 is an object storage service that offers scalability, data availability, security, and performance. We used it to host the static website of the applicarion and for storing the pictures.
- **AWS Lambda**: AWS Lambda is a serverless compute service that executes code in response to events, handling the compute resources. The lambda functions in the project have been used for uploading image into bucket, solving equation and querying the db.
- **Amazon API Gateway**: We designed and implemented RESTful APIs to handle HTTP requests from the frontend. This approach allows for a structured and standardized way of interacting with our backend services so to post and get request from frontend to lambda functions.

- **Amazon Rekognition**: Amazon Rekognition is a pre-trained computer vision API to extract insights from images and videos. We used it to extract the equations text from the images.
- **Amazon DynamoDB**: Amazon DynamoDB is a Serverless, NoSQL database service, we used it to store the equation's solutions.
- **Amazon CloudWatch**: : Amazon CloudWatch is a monitoring and observability service. It provides data and actionable insights to monitor applications, understand and respond to system-wide performance changes, optimize resource utilization, and get a unified view of operational health. We use it to keep track of the metrics of the system when testing scalability.

To improve our application we also considered using AWS Cognito for enabling authentication and CloudFront to enable broad network access to our website across the internet but we weren't able to implement such services because of limitations for the AWS student's account.

2 Project Implementation

In this section we explain the entire architecture of our system, in Figure 3 there is the entire workflow of the application.

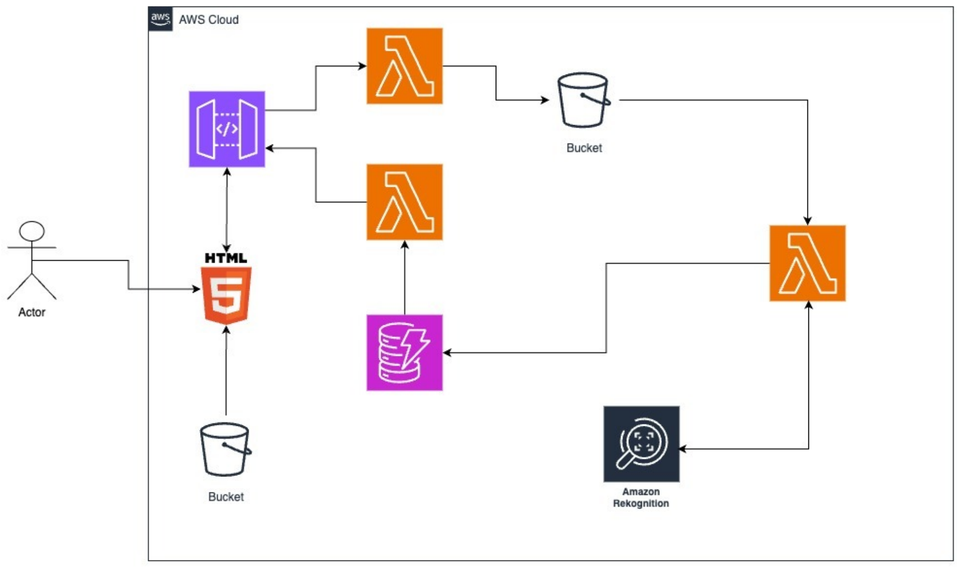


Figure 3: Workflow of the application

First of all, the user will connect to the frontend of our application and will upload a *.jpeg* image of an equation or inequality for which they are looking for the solution. This frontend is a simple static website hosted by an S3 Bucket.

After uploading the image, from the embedded JavaScript we extracted the Base64 encoding of the image (binary-to-text encoding with 64 characters) and we sent it to a Lambda function with a POST request through a RESTful API, by taking advantage of the AWS API Gateway. The Lambda function triggered by the POST request, will update such image to a S3 Bucket where all the updated image will be stored.

Subsequently, as soon as new *.jpeg* file is being uploaded on this Bucket, a new Lambda Function will be triggered. This is the most important function because here happen all the computation of the input equation/inequality. As a matter of fact, in this Lambda function the main operations of our application are performed:

- **Amazon Rekognition**: to extract the text from the input images;
- **Pre-processing**: after extracting the text, some pre-processing is needed to transform the extracted text in a format suitable for SymPy;
- **SymPy**: to actually compute the solution of the equation/inequality. We needed to attach a personalized layer to this Lambda function with Python 3.9 and SymPy;
- **Database**: store the result in a database for a subsequent query;

The result will be stored in a DynamoDB table, where each entry (solution) will be stored with 4 fields. The key will be the file name, then there are also a timestamp, the equation text and the equation solution.

After all these computation, from the frontend will be sent a GET request, always through the same RESTful API, to another Lambda function, which is responsible for querying the database and retrieving the solution of the uploaded equation. In the end, this solution will be easily displayed in the user interface:

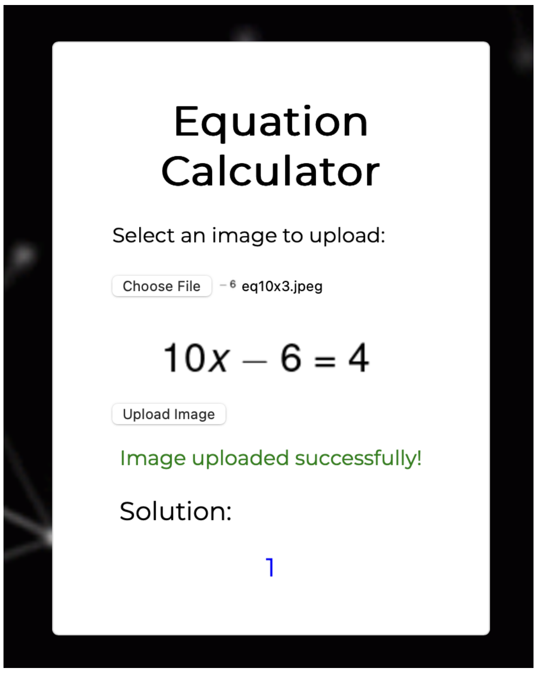


Figure 4: Screenshot of the solution of the eqaution

3 Scalability

To ensure the scalability of the system we have been careful with our design choices, especially during the service discovery part. Since ours is a **server-less** application, it is easy to scale. As a matter of fact, the logic of our application is mainly handled by Lambda functions, because of its ability to quickly absorb traffic peaks through managed scalability and simultaneous code startup. Beyond the Lambda functions, we also used S3 Bucket and API Gateway, which do not suffer from scalability issues just like the Lambda function. As a matter of fact, by looking at the Lambda documentation we read that for a normal user there is a total concurrency limit of 1,000 concurrent executions across all functions in an AWS Region. However, such limit has been decreased to 400 for a student account.

In order to test availability and scalability we tried to stress our application under different scenario, or rather we simulated different amounts of user simultaneously using the application.

To stress our application we took advantage of the **Selenium** library in a multi-thread script. In this script essentially we open a new browser and access the application and then we upload an image to test whether it is working; we also added a timer that starts after the click and ends as soon as the solution is being displayed: such a timer helps us keep track of the time taken in executing operations to see if performance remains the same even if many users are connected at the same time.

This is the code we used to test our application:

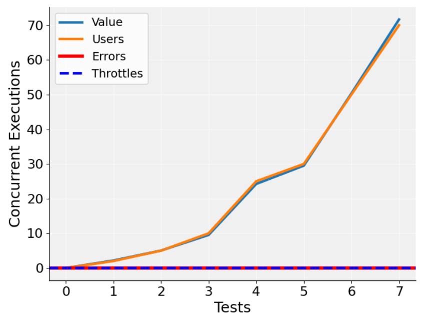
```python
def upload_test():
    # Set up Chrome options for headless mode
    driver = webdriver.Chrome(options=chrome_options)
    driver.get("http://127.0.0.1:5500/frontend.html")
    # upload an image
    driver.find_element(By.ID, 'imageUpload').send_keys("pceqal.jpeg")
    # click on the submit button
    driver.find_element(By.CSS_SELECTOR,"input[type='submit']").click()
    # start timer
    start = time.time()
    # wait for the result to be displayed
    while True:
        try:
            result = driver.find_element(By.CLASS_NAME,"solution-text")
            # end timer
            end = time.time()
            break
        except:
            pass
    times.append(end - start)
    # close the browser
    driver.quit()
# Number of concurrent users to simulate
num_users = 10 # 2,5,10,25,30,50,70
# Create and start threads
threads = []
times = []
for _ in range(num_users):
    thread = threading.Thread(target=upload_test)
    threads.append(thread)
    thread.start()
# Wait for all threads to complete
for thread in threads:
    thread.join()
```

We chose Selenium because it allows us to simulate the user behaviour, and by launching many different threads we simulate the concurrent access to the appication.

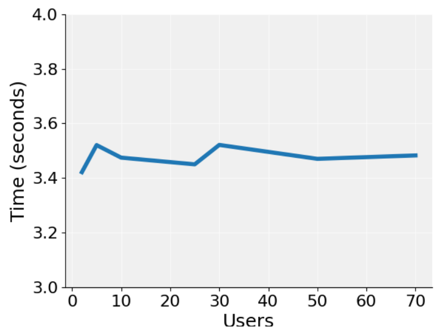
We tried different workloads, until eventually we got blocked (twice) when we tried to simulate the concurrent access of 50 users. Nevertheless, we continued our tests by recreating the application on a unrestricted account, despite the risk of being charged for using a certain amount of resources. We have been able to carry on our planned test up to 70 concurrent users.

To check whether the application is scaling or not we used AWS CloudWatch metrics related to the Lambda functions, like the number of Concurrent Invocations and the number of Invocation errors and throttle. Moreover, we are also keeping track of the mean execution time among all the users by using the *time* library in python which essentially allows us to simulate how much time elapses from when selenium uploads the photo to when the result is obtained, essentially measuring the time it takes to complete all the operations in the cloud.

Obviously between different runs of a test there are different results but by running the same test several times we got a more reliable value



(a) Concurrent lambda invocations



(b) Average time taken for users

Figure 5: Testing

As shown in the plots above, we can see that the time elapsed from when a user send the request to when it gets the results remains overall constant, regardless of the number of users that are simultaneously connected to the application. Elasticity and scalability of the Lambda functions is confirmed by the plot on the right. In the left plot there are four elements: the number of concurrent lambda functions, the number of users in the test, the number of invocation errors and the number of throttles.

It is clear how the application is able to scale without any problem, since we are exploiting the intrinsic scalability properties of Lambda and RestAPI.

4 Conclusions

This project focused on developing a scalable cloud-based system for processing mathematical equations and inequalities from user-uploaded images. By utilizing Lambda functions for serverless computing, DynamoDB for data storage, and API Gateway for endpoint management, we designed a robust solution to efficiently handle varying user loads, ensuring *scalability*, *availability* and *responsivness.*

To validate the scalability of our architecture, we conducted load tests simulating concurrent access by 2 to 70 users. The results demonstrate our system's capability to dynamically scale resources in response to increasing demands while maintaining constant performance.

One of the key strengths was evident in the implementation of Lambda's auto-scaling feature, which dynamically adjusted computing resources based on fluctuating user demands. This capability ensured that our system maintained optimal performance levels, smoothly handling simultaneous user requests without experiencing delays or performance degradation.

Another critical aspect of our solution was the use of DynamoDB for data management. DynamoDB consistently delivered low-latency access for storing and retrieving data, even under peak loads. This reliability was crucial for maintaining responsive user interactions and ensuring the integrity of data operations throughout the system.

Furthermore, API Gateway played a key role in managing incoming requests effectively. By efficiently routing requests to Lambda functions, API Gateway minimized latency and errors, contributing to a reliable and streamlined user experience.

From a cost perspective, adopting a serverless architecture proved beneficial. By leveraging Lambda's pay-as-you-go model, we optimized resource allocation and minimized idle resource costs. This cost-effective strategy allowed us to scale resources dynamically in line with actual usage patterns, ensuring efficient resource utilization without incurring unnecessary expenses.

References

[1] Aaron Meurer, Christopher P. Smith, and Paprocki et all. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.

[2] Amazon Web Services. Amazon web services documentation, 2024.