CLOUD COMPUTING
FINAL PROJECT REPORT
A.Y. 2021/2022

PROJECT REPORT

Giulio Smedile
smedile.1951902@studenti.uniroma1.it

Antonio Santamaria
santamaria.1999876@studenti.uniroma1.it

Emanuele Volanti
volanti.1761743@studenti.uniroma1.it

July 20, 2022

INTRODUCTION

This document serves as Project Report for the Cloud Computing course, held by Professor E. Casalicchio, from the Data Science course at the University of Rome Sapienza, during Academical Year 2021/2022.

1 General Project Description

The project the team developed is of an application that lets users hold some kind of online diary, and show them a trend of their mood across the various entries of their diary.

The application is available both as a web application and an Android application. The back-end is entirely built upon AWS services. More specifically:

- **AWS Cognito** for user authentication;
- **AWS Lambda** for the main computing operations;
- **AWS S3** for storing the web front-end and for storing the individual diary entries;
- **AWS DynamoDB** to store associations between users; entries and the sentiment value computed for each entry.

Additional components have been made with Python, HTML, JavaScript and Kotlin.

The system has been tested to verify its optimal functioning, and the various cloud component have been fine tuned to enhance their cost-to-performance ratio.

2 Design

The system is designed to take advantage of the cloud computing platform it is based on. More specifically, it is:

- *Loosely Coupled*: Each part of the application is a highly independent component that is able to work on its own. Each component could potentially be replaced or upgraded without necessarily having the need to modify the others.
- *Platform Agnostic*: The system is designed to work with simple HTTP calls, allowing users to choose the front-end solution of their choice, be it the web interface or the Android application.
- *Scalable*: The system takes advantage of the AWS platform capabilities to optimize cost and performance based on user load.

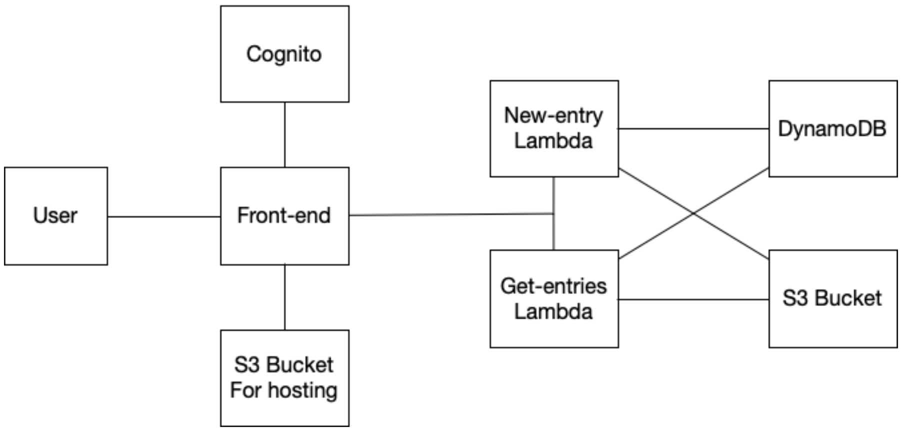


Figure 1: *A graphical overview of the application's components*

2.1 Front-end interfaces

Two front-end interfaces have been developed for the project: a web interface, powered by HTML and JavaScript, and an Android application, made with Kotlin.

The system achieves the intended functionalities as stated in the project proposal:

R.1. multiple users should be able to register and access the system
R.2. users should be able to create new diary entries
R.3. users should be able to access past diary entries
R.4. the system should be able to discern between entries of different users
R.5. the system should be able to perform sentiment analysis on each diary entry, and output a sentiment value
R.6. users should be able to access a trend of their calculated sentiment related to time

Both interfaces provide the same general functionalities, with obvious differences regarding the platform they were developed for (such as platform-dependent APIs to access AWS services), as they rely on HTTP calls to the back-end for communication purposes.

Security is ensured at all times as each request is paired with a Cognito session token, which is used to identify the user which is making the request.

The functionalities provided to the users by the front-end are:

- *Authentication*: Users can register and log in and out of the system by calling directly AWS Cognito. The users need to log in by their username and a password.
- *Diary Submission*: Users can submit a new diary entry by providing a title and the entry itself. This entry will be then processed by a Lambda function, and an indication of the sentiment analysis will then be returned by the back-end.

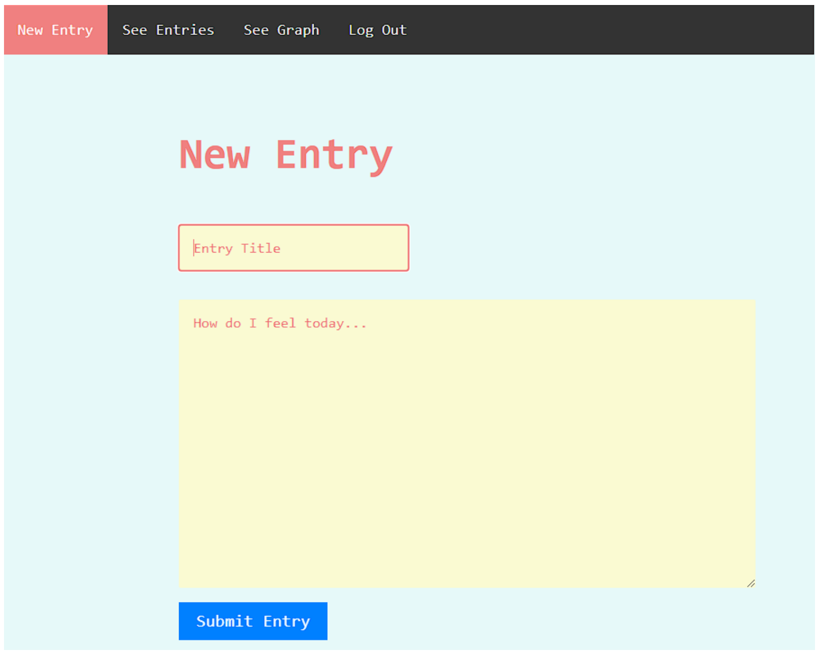
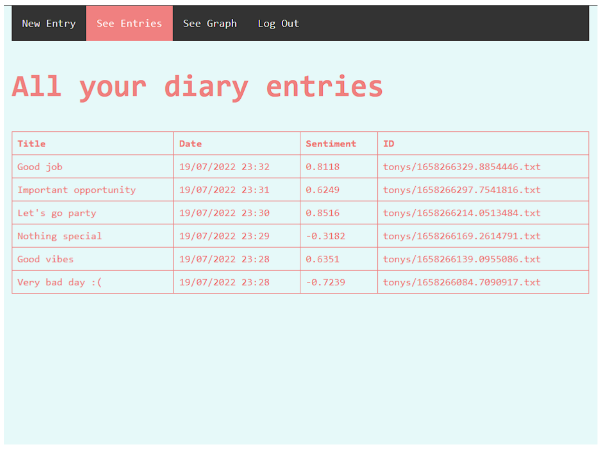


Figure 2: *The new entry page*

- *Diary Visualization*: Users can visualize their diary in one of two ways, as a list of entries, sorted in chronological order, or as a two-dimensional chart which plots the sentiment value over time. The data requested to the server is the same (give all the entries related to a single user) for each visualization, and the front-end is tasked with interpreting and showing the data. The data is visualized in tabular form with the title of the diary entry, the timestamp of the submission, the cumulative sentiment score as calculated from the application, and the ID of the entry as stored in the S3 bucket.



All your diary entries

Title	Date	Sentiment	ID
Good job	19/07/2022 23:32	0.8118	tonys/1658266329.8854446.txt
Important opportunity	19/07/2022 23:31	0.6249	tonys/1658266297.7541816.txt
Let's go party	19/07/2022 23:30	0.8516	tonys/1658266214.0513484.txt
Nothing special	19/07/2022 23:29	-0.3182	tonys/1658266169.2614791.txt
Good vibes	19/07/2022 23:28	0.6351	tonys/1658266139.0955086.txt
Very bad day :(	19/07/2022 23:28	-0.7239	tonys/1658266084.7090917.txt

Figure 3: *The "see entries" page*

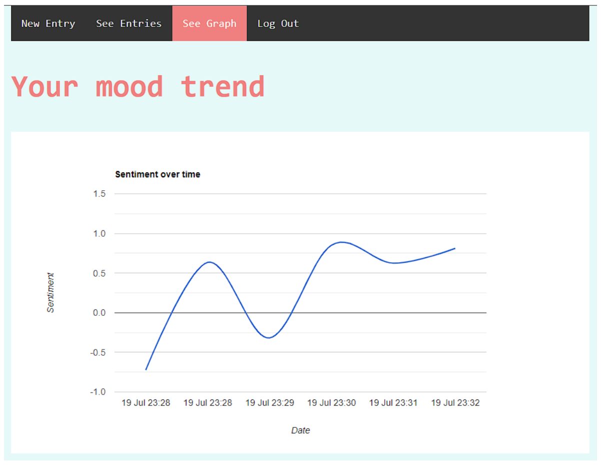
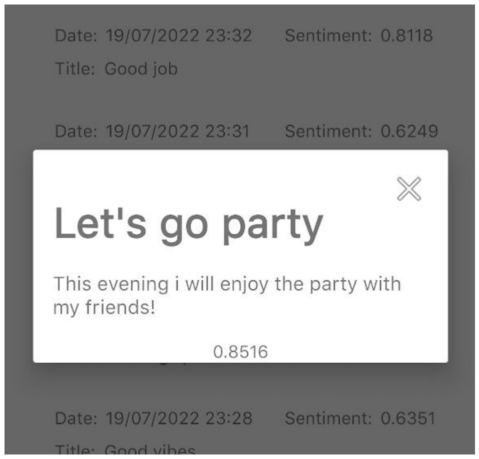


Figure 4: *The "mood graph" page*

- *Single Entry Visualization*: A user can choose to visualize the information of a single entry. By selecting the single item in the tabular view, the user is shown a more in-depth view of the entry: both the title and body of the diary entry are shown, together with the cumulative sentiment score.



Date: 19/07/2022 23:32 Sentiment: 0.8118
Title: Good job
Date: 19/07/2022 23:31 Sentiment: 0.6249

Let's go party

This evening i will enjoy the party with my friends!

0.8516

Date: 19/07/2022 23:28 Sentiment: 0.6351
Title: Good vibes

Figure 5: *A single entry, as viewed from the Android application.*

Communication to the back end is carried through a series of HTTP requests to the Lambda functions, which will carry on the computation directly and will use the HTTP request parameters as input data.

2.2 Lambda Functions

Two AWS Lambda functions have been created for the project, which serve as the main computational units for the read and write operations of the back-end, respectively.

The first Lambda function, named `sentiment`, is tasked with validating the data submitted from the user, carry on the sentiment analysis through the NLTK Python API, and store the result in the databases. The entry is firstly stored in the S3 bucket, with a filename matching the user and the timestamp it has been submit to the server, precise to the millionth of a second, and then the meta data about the entry, such as the overall sentiment data and the title, are stored in DynamoDB.

The second lambda function, named `getDiaryEntry` is tasked with the read requests carried from users. Differently from the first lambda function, it supports two work modes: the users can either request all of their entries' metadata, or more in-depth data for a single entry.

The function receives as parameters the session token and the "type" of the request, either it being `single` or `all`, depending on the user prompt in the front-end interface. When the `single` mode is selected, also the unique ID of the entry is inputted to the function.

When requesting all of the data, through the `all` mode, only the DynamoDB instance is interrogated. A query executed, and all of the data regarding a single user is then copied in a list, which is then returned to the front-end through a JSON structure.

A single entry, requested via the `single` mode, is instead polled from the S3 bucket by its ID, which is the filename as defined in the first Lambda function.

After our testing, which is described below in this report, we have slightly modified the Lambda functions' configuration to allocate 1024MB of RAM instead of the default 128MB, to accommodate the heavy tasks put in place by the functions, especially in regards of the `getDiaryEntry` Lambda, which has to deal with hundreds of thousands of entries.

2.3 DynamoDB

The DynamoDB instance has been created to provide a solid database for the whole application to rely on. The database is designed to function as a standard SQL database, with a single table which holds all the needed data.

The table, named *diary-associations*, relates a single user with their diary entry and the sentiment data which is generated by the back-end. The table is structured with the following columns: *id* (which is also the primary key), *sentiment* (composed of the four sentiment values calculated by NLTK), *title* and *username*.

This last value is used as a *global secondary index*, and is used to speed up the operations carried out by the `getDiaryEntry` Lambda, when trying to retrieve all the data belonging to a certain user. In previous iterations of the application, we experimented with the Scan function, instead of using a classic Query, which gave us a bit more flexibility with the parameters to the expense of the overall computation time. Since the Lambda function, however, is capped to run in a maximum of 3 seconds, we had to opt for this option which uses the secondary index, as it allows us to obtain way more entries (we limited the Scan to only the newest 300 entries) with a single request to the database, also using way less resources.

2.4 S3 Bucket

Two buckets have been created for the purpose of the application.

The first bucket, named *diary-frontend*, simply acts as a hosting provider for the Web Interface of the application.

The hosting has been set up fairly simply, and a HTTP endpoint has been automatically generated. Even though the pages are fairly dynamic, S3 has been chosen as the HTML and JavaScript part of the front-end interface are static in their own, and modifications are only applied at run time, on the user's end. As the requests are fairly simple and lightweight in their nature, this bucket was not part of our testing.

The second bucket, however, is a bit more interesing.

Named *diary-entries*, it holds the main diary data for each user. The bucket is set up to be intuitive to navigate, and to be easy to search for in a temporal order.

Each user is given its own folder when they send their first entry, where all their entries are stored. Each entry is saved as a simple text file, and the file is given as a name the timestamp of the moment the file has been registered in the Lambda function. The correct title of the entry, as previously stated, is instead stored in the DynamoDB table.

2.5 Cognito

The AWS Cognito implementation has been quite straightforward. Users can both sign-up and log-in from the user interface, and their authentication data is then sent to the lambda to effectively authenticate the requests.

The user pool has been created mostly with standard specifications, but a few settings has been tweaked. At signup, for security reasons, both a username and an email address have to be provided. However, only the username is then used for authentication, and the email is automatically marked as verified (Cognito by default requires an email address to be verified, even for fairly simple applications such as this one) through a trivial Lambda function which is triggered when the sign-up operation is completed.

To aid our testing, the session tokens that Cognito produces when a user logs in have been extended to the maximum allowed duration, which is of 24 hours. With the default settings of 60 minutes, our most extensive testing would fail before their intended completion.

3 Testing

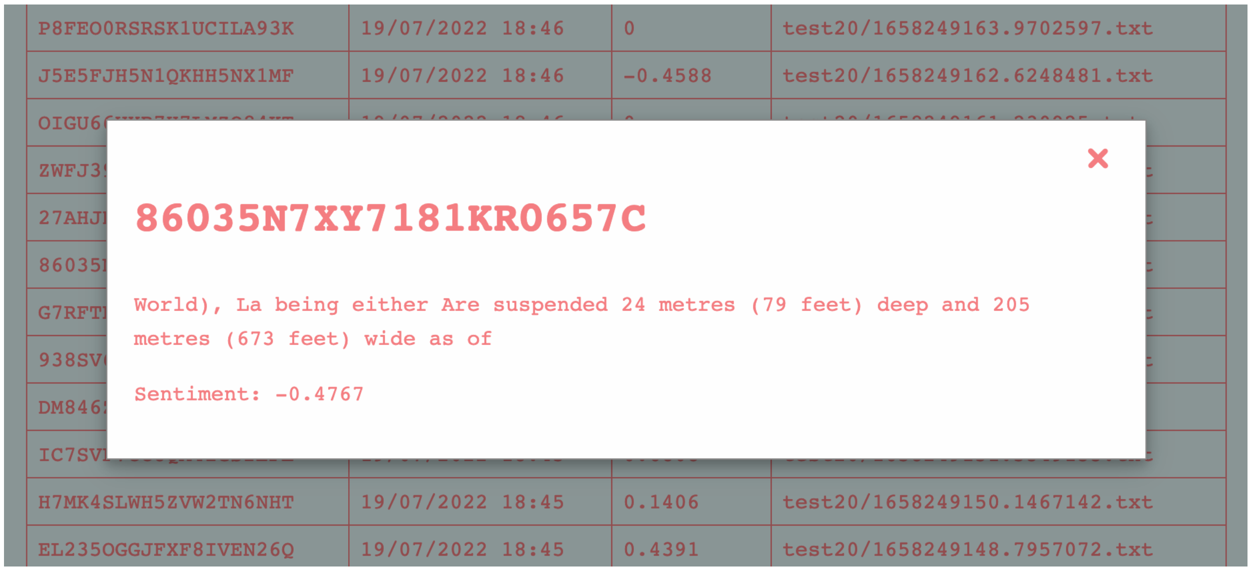
Testing has been a crucial part of our project. Testing the application gave us valuable insights of how real-world applications are built, and helped us modify the application itself to achieve the maximum performance.

3.1 Designing the tests

Two tests have been designed and put in place to put the application to its limits. Both tests are implemented as a Python script, and have been mainly run on our machines and on a VM instance created on the website **DigitalOcean** that helped us stress the system way better than our home computers would. The tests have been designed to stress the system in such a way to emulate many concurrent users accessing the application at the same time, conducting different operations from each other. The main goal of these tests was to **evaluate how the system would automatically adjust itself** to variation in user load and **if and how users would notice performance variations** if the system was under heavy load, or if it was poorly optimized.

A first script, named *graphical_benchmark*, was mainly designed to monitor the response time of the Lambda functions and the noticeable effects of the heavy load on the system.

In this script a number of threads would each generate a list of texts each, one different from the other, through the *DocumentGenerator* library, send them to the back-end, and await for a successful response. This was done to generate an interesting input dataset for the system to process, to avoid bias in the response, and to produce a graphical evaluation of how the system would respond to being under stress, thanks to a graphical output generated via the *matplotlib* library. We found out that the maximum concurrent executions generated with this method was about **35 requests per second**.



P8FEO0RSRSK1UCILA93K	19/07/2022 18:46	0	test20/1658249163.9702597.txt
J5E5FJH5N1QKHH5NX1MF	19/07/2022 18:46	-0.4588	test20/1658249162.6248481.txt
OIGU6[illegible]	[illegible]	[illegible]	[illegible]
ZWFJ3[illegible]	[illegible]	[illegible]	[illegible]
27AHJ[illegible]	[illegible]	[illegible]	[illegible]
86035[illegible]	[illegible]	[illegible]	[illegible]
G7RFT[illegible]	[illegible]	[illegible]	[illegible]
938SV[illegible]	[illegible]	[illegible]	[illegible]
DM846[illegible]	[illegible]	[illegible]	[illegible]
IC7SV[illegible]	[illegible]	[illegible]	[illegible]
H7MK4SLWH5ZVW2TN6NHT	19/07/2022 18:45	0.1406	test20/1658249150.1467142.txt
EL235OGGJFXF8IVEN26Q	19/07/2022 18:45	0.4391	test20/1658249148.7957072.txt

86035N7XY7181KR0657C

World), La being either Are suspended 24 metres (79 feet) deep and 205 metres (673 feet) wide as of

Sentiment: -0.4767

Figure 6: *An entry generated from this benchmark script. In the background, notice other entries generated in the same way.*

The second test script, named *simple_benchmark*, is fairly simpler from a code standpoint, but is more interesting in what produces on the server side.

The requests done to the server are the same for each thread, comprised of a pre-specified diary entry. This test does not provide a graphical output of any kind at the end of its execution, but is able to produce more requests per second, as the calculation executed in the hosting machine are way less intense. With this less intensive method, we found out that the system could handle a maximum of **200 concurrent executions** per second. With bigger request rates, the Cognito API would not be able to keep up with so many concurrent requests.

Both tests have been designed to operate in three different ways, or *stop-mode*s, to test the application in different ways.

- `time`: Each thread runs for a given amount of time. Used to test latency in the system with a stable workload.
- `num`: A given number of requests is given to each thread to complete. Produces interesting insights on average response times from the application back-end.
- `incremental`: A more standard testing approach, designed to monitor how the system scales and how it behaves with or without heavy load. In this mode, various timestamps are selected, in which some threads are put to sleep until they have to start working. Thread working times are espressed in percentages of the total number of threads. After some testing, we found out that the series `[10, 20, 50, 100, 100, 100, 50, 20, 10]` produced the most interesting results.

Test data have been collected via the built-in service **AWS CloudWatch** for the data collected on the server, and with the custom scripts (when planned) for the client data.

3.2 Test #1: Write test

This first test has been carried out to stress the *sentiment* Lambda function. As it is the most intensive of the two Lambdas, which touches both the DynamoDB table and the S3 bucket, it was deemed the most interesting to test.

The test has been executed on the DigitalOcean machine, with the *simple_benchmark* script in incremental mode, and has been running for a total duration of 1 hour with a maximum of 200 concurrent executions.

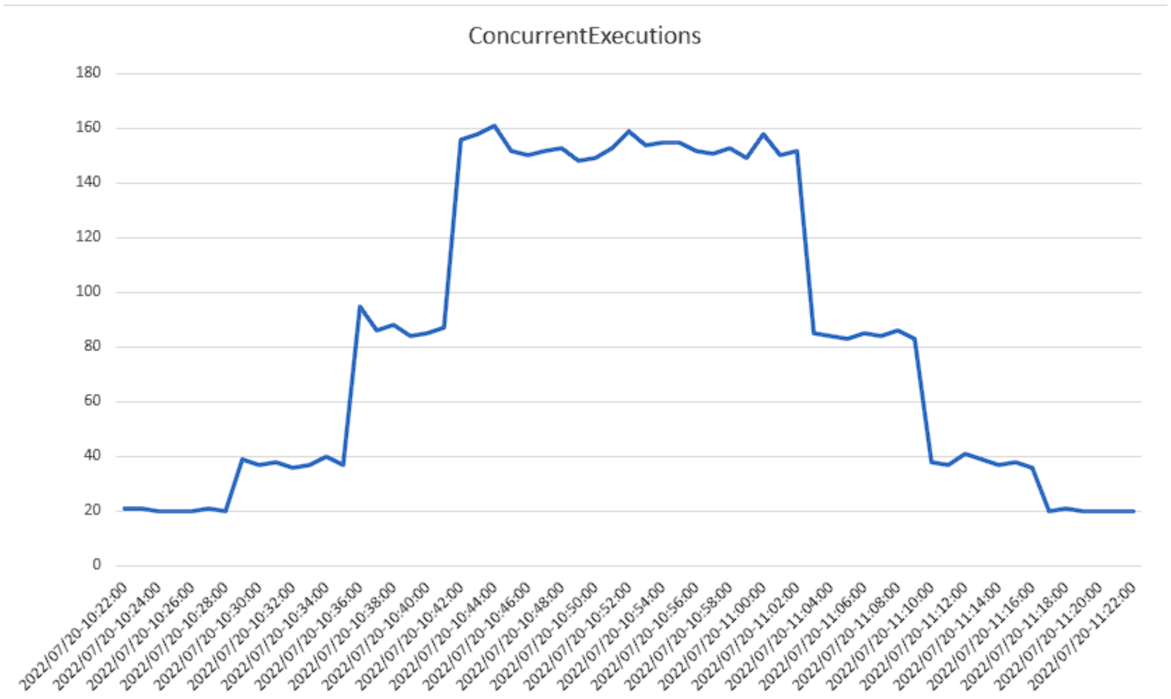


Figure 7: *The number of the Lambda function executions. Slight variations of the graph are due to latency given by the internet connection and the calculation necessary before sending the HTTP request.*

Interesting insights that have been gathered by this test are regarding the stability of the system, it being the total number of errors, and the scaling mechanism of the system itself.

Even though a Lambda function is set to scale automatically after a way larger number of concurrent executions (3000, per the Amazon documentation), when the test reaches its maximum load it automatically achieves some sort of optimization with heavier loads, as it is possible to notice when analyzing the execution times of the function itself: while the average times remain constantly around 1 second throughout all the execution, the minimum execution times seem to improve at least by 50% when the maximum load is achieved.

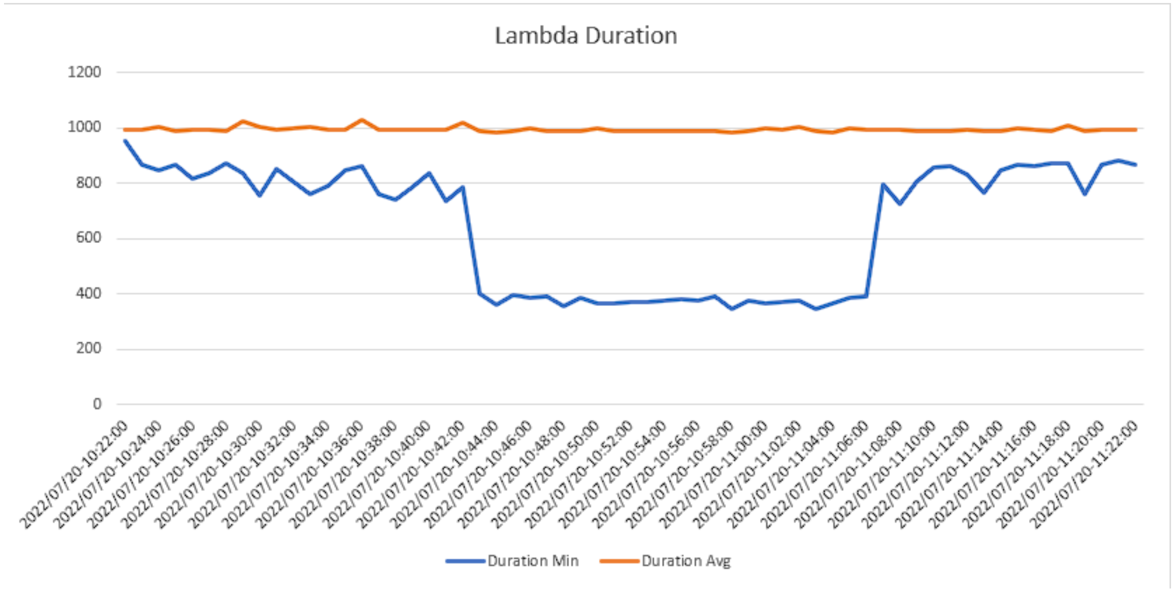


Figure 8: *Average (in orange) and minimum (in blue) duration of the Lambda function, in milliseconds.*

Interestingly however, even with higher concurrent executions, the system remains fairly stable, with a number of errors way below a noticeable rate. We have figured that this rate of error on heavy loads (about 1 every 200 executions) is acceptable, and the application maintains a >99% success rate, even with these high volumes of traffic.

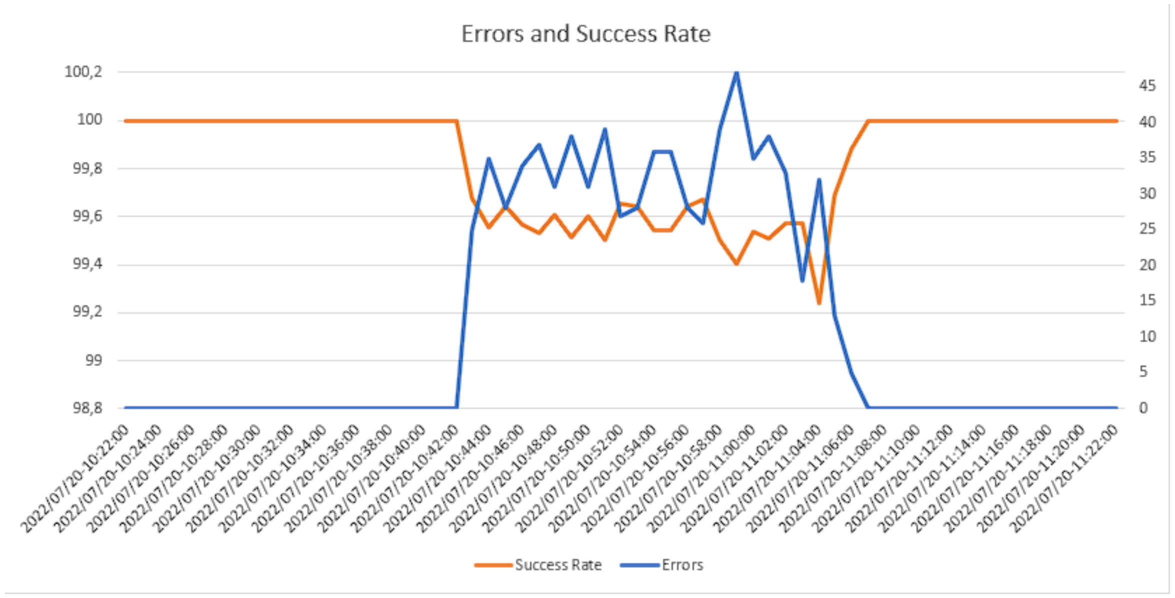


Figure 9: *The success rate percentage (in orange) and error count per minute (in blue).*

3.3 Test #2: Read test

The testing on the read pipeline has been carried to monitor especially how the DynamoDB instance would react to high level of requests. Similarly to the first test, this was carried on the DigitalOcean instance, with a maximum of 200 concurrent executions, for half an hour.

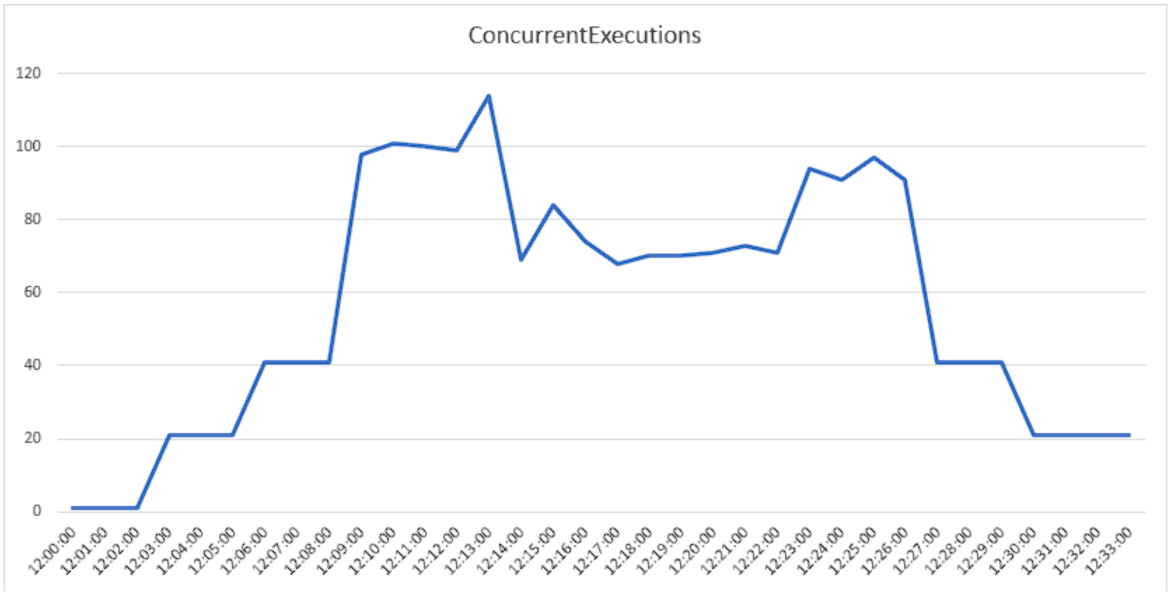


Figure 10: *The number of executions per time.*

This test highlighted how the read mechanism, however, would be more prone to errors. We figured out that the pipeline of operations executed by the read Lambda would be more delicate, as even small hiccups would make the function fail (Lambda functions are set to complete in 3 seconds, or they fail). When reaching higher level of requests, the function would produce way more errors, and cause concurrent executions to slow down (as they would wait longer for the server response). However, even if slightly worse than the previous test, we think that a >90% success rate is still highly acceptable.

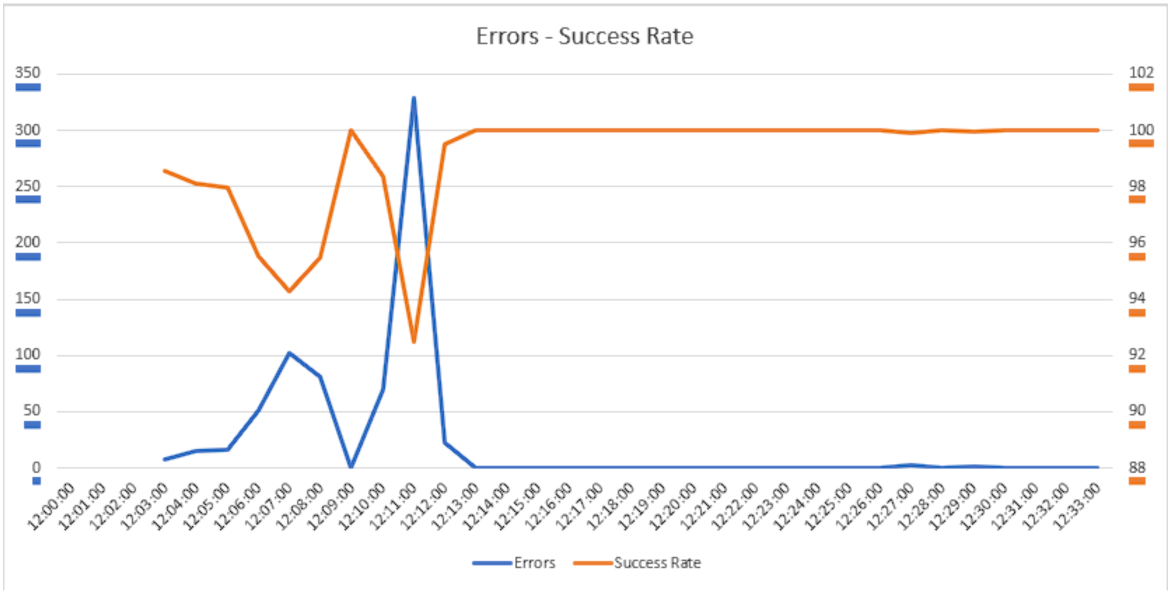


Figure 11: *The success rate percentage (in orange) and error count per minute (in blue).*

This test was carried mainly to monitor how the DynamoDB database would react to the traffic. We were delighted with noticing that not only the db would scale accordingly, but that in doing so, both the average and the minimum execution time of the Lambda function would improve as well.

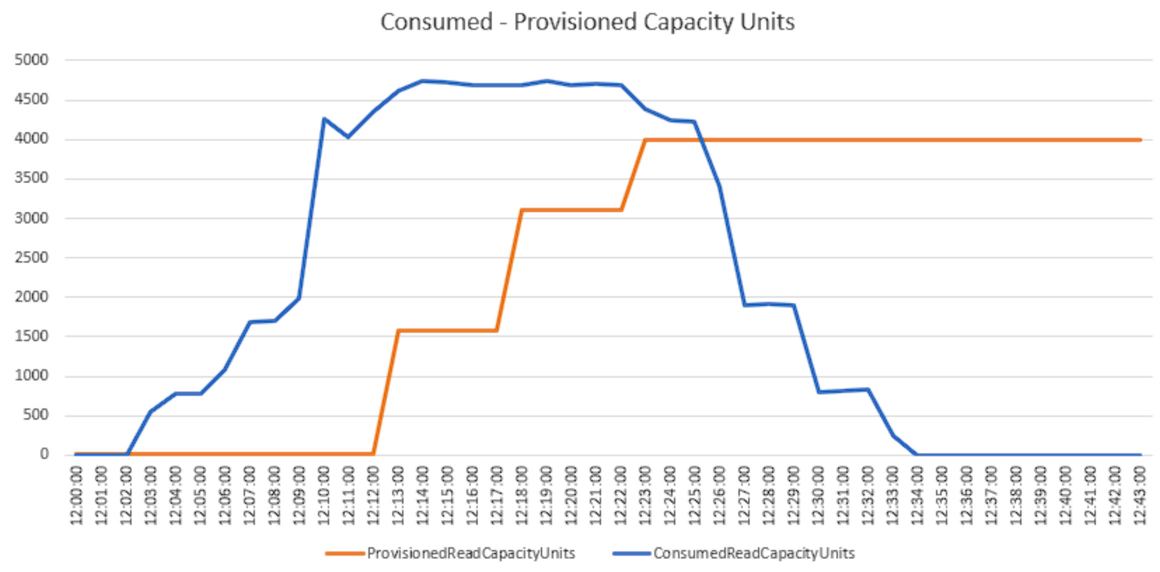


Figure 12: *The number of provisioned (in orange) and consumed (in blue) read units in the DynamoDB.*

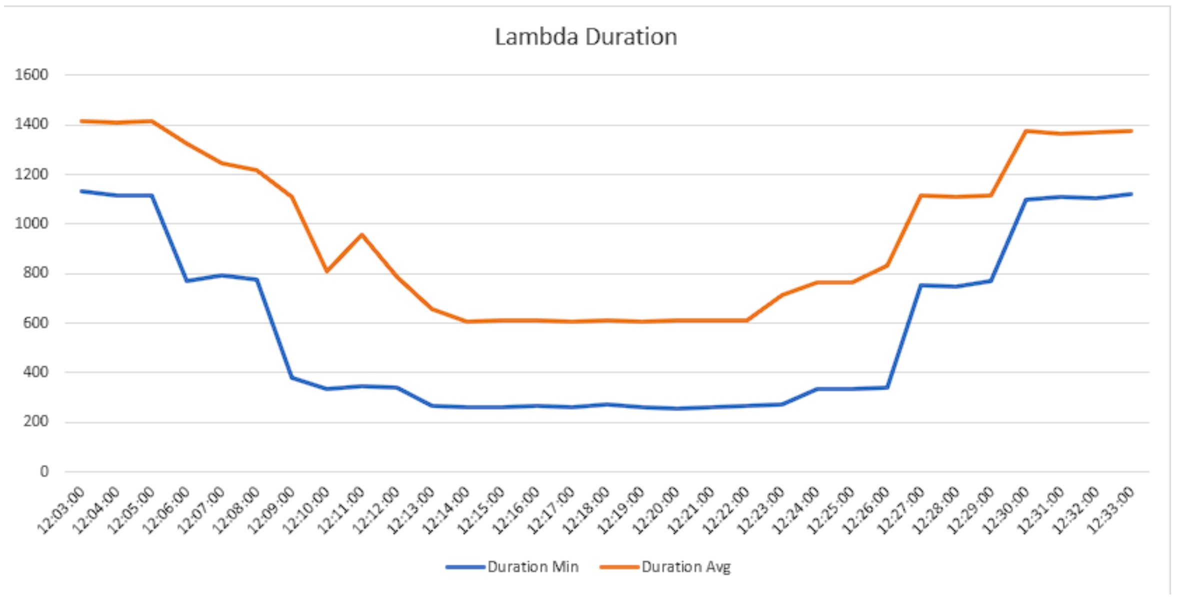


Figure 13: *Average (in orange) and minimum (in blue) duration of the Lambda, in milliseconds.*

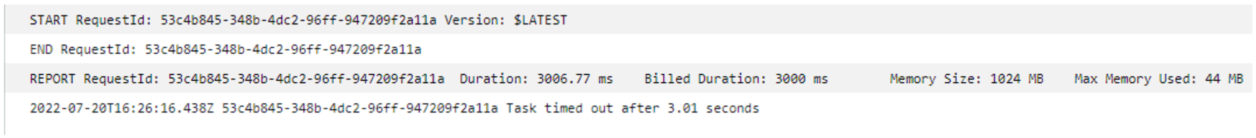
3.4 Test #3: Autoscaling test

Since DynamoDB was the component most stressed by the tests, and the one that seemed to benefit the most from having multiple operational units, we have decided to test this particular component by removing its scaling capabilities.

Normally, the db instance was set up to have up to 4000 read/write units available, with scaling being triggered at 70% usage. To test the extremes of the capabilities of the system as a whole, we decided to completely disable scalability, and momentarily set the read/write units to a steady 1000, without scaling options.

We decided to run a test identical to the specifications of Test #2 (read, incremental, 30 minutes, 200 threads) to compare the two.

The thing that was immediately noticeable was the sheer amount of errors happening to the requests. After analyzing the logs in CloudWatch, it was possible to see that it was not Dynamo that was failing, but either the Lambda function. By not having enough read units available, the execution time of the function would easily surpass the 3 second mark and fail automatically.



```
START RequestId: 53c4b845-348b-4dc2-96ff-947209f2a11a Version: $LATEST
END RequestId: 53c4b845-348b-4dc2-96ff-947209f2a11a
REPORT RequestId: 53c4b845-348b-4dc2-96ff-947209f2a11a  Duration: 3006.77 ms   Billed Duration: 3000 ms      Memory Size: 1024 MB    Max Memory Used: 44 MB
2022-07-20T16:26:16.438Z 53c4b845-348b-4dc2-96ff-947209f2a11a Task timed out after 3.01 seconds
```

Figure 14: *An error message, as seen from CloudWatch, that notifies the failure of a function instance.*

Even the executions that would complete, however, took much more time to run in respect to the previous test run. At the beginning of the test, when the executions would not saturate the read units, the execution time was fairly similar to the previous test, but after a few increment in the testing frequency, the execution time would significantly rise.

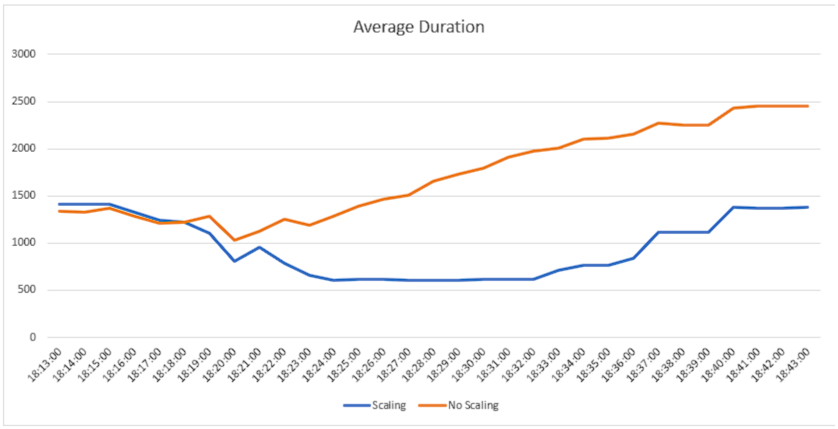


Figure 15: *Average run time comparison. In blue, the execution with scaling enabled, in orange, the one without.*

This was correlated by a higher error rate, with success percentages going as low as 40%. This would be obviously noticeable in a production application and would not be an optimal state for a service.

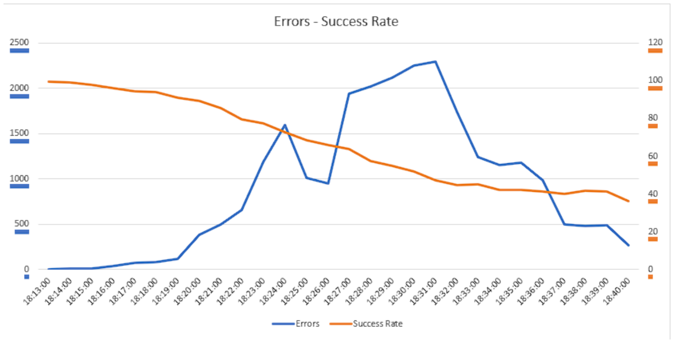


Figure 16: *The success rate percentage (in orange) and error count per minute (in blue).*

3.5 Test #4: Client test

This last test was the only one that was physically carried in our machines, and the one that was executed with the *graphical_benchmark* script. This test was executed to monitor if the end user would notice changes in responsiveness of the application if the services was under heavy load. This test was executed with 50 threads running at all times for one hour.

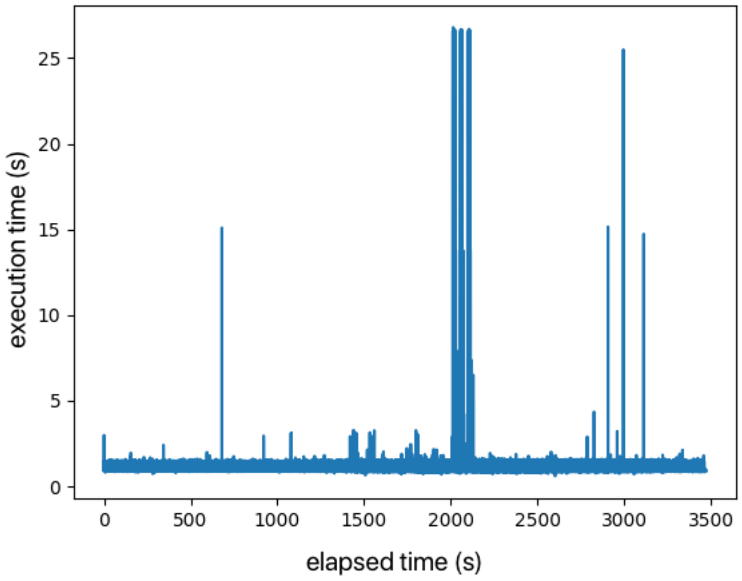


Figure 17: *Client test data.*

From this test, it is very easy to notice that the majority of the executions (>98%) would stably run between 1 and 1.5 seconds, similarly to the times previously analyzed in the Lambda functions.

However, it is also really interesting to notice that spike at around 2000 seconds after the start time: after some research, we found out that was due to a slight downtime experienced by the *us-east-1* AWS cluster, which is where all of the application resides.

We have concluded, however, that a normal user would not normally notice any slowdowns in the service, even at during peaks in traffic.

4 Conclusion

This project has been really interesting to carry out, allowing us to touch with hand how a real-life cloud environment works and has to be set up.

The project is open source and publicly available at the GitHub repository at this link.

A live version of the project is also publicly accessible by accessing the S3 endpoint at this link. If for some reason this link is not accessible, a demo video of the application can be found in the GitHub repository.