UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Facultat d'Informàtica de Barcelona

Project Challenge Report

Cloud Computing

Project Group:
Andrea Armani
Valdemar Hernández
Ricardo Holthausen
Jesús Huete
Dimitrios Tsesmelis

Professor(s):
Angel Toribio

Project Period:
Winter Semester 2020

Date of Submission:
May 24th, 2020

Edifici B6 del Campus Nord, C/Jordi Girona, 1-3, 08034 Barcelona

TABLE OF CONTENT

FUNCTIONALITY & SCOPE

The goal of the developed application is to simulate the behavior of a virus outbreak within the population of a given location.

The first thing that a new user has to do in order to access our website, is to create a new user. After giving all the required information (username, name, email etc…), a verification email is sent to his email address. After validating the email, the user is able to authenticate himself and have access to the main dashboard.

In the dashboard, the user can run new simulations by providing the required parameters, which will be used to set up the simulation environment. The parameters to be received are the following:

- Simulation Name: Name given to the current simulation. This can be used later to retrieve past simulation results.
- Rates
    - Mortality: Percentage of the population that dies during one day.
    - Infection: Percentage of the population that gets infected during one day.
- Population Size: Amount of people that make up the simulation's population.
- Duration: Maximum amount of days to be simulated.
- Initial state
    - Susceptible: Percentage of the population susceptible to the disease.
    - Incubating: Percentage of the population incubating the disease. They cannot infect others at this stage.
    - Infected: Percentage of the population that may infect susceptible people.
    - Treated: Percentage of the population initially considered as treated against the disease.
    - Cured: Percentage of the total population initially considered as cured from the disease.

With these parameters, the users will have the opportunity to analyze how the spreading of the virus is affected depending on the values defined for them. After the user inputs these values, the simulation will run until one of the following happens:

- The simulation reaches the number of days input by the user in the "Duration" parameter.
- Every person in the simulation is either immune, susceptible or dead.

In the left-hand side of the dashboard, the user can see the older simulations that he has already run. By pressing the refresh list, he automatically receives the updated list of the simulations.

Moreover, on the bottom of the dashboard, the user is able to visualize the results of the different simulations that he ran in the past, by providing the name of the desired simulation to be plotted. The visualization depicts the overall progress of the simulation, by showing the evolution of the different statuses of the population (infected, cured, dead etc…) over the days. In addition, the pie charts present the aggregated statuses of the first and the last day of the outbreak, which gives an overall image to the user of how the infection started and how it ended.

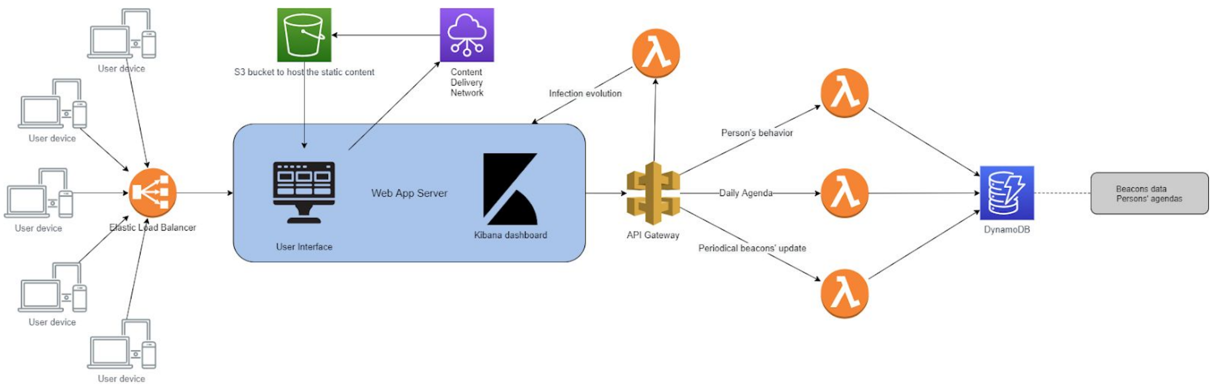
How does it work?

In order to create the simulation, firstly the information about the population is generated. Information about the status of the people, their locations throughout the day and the days they have spent in their current status is obtained. After this, an iterative process starts, first storing the current statistics in a list and then updating the statuses and the days in the current status arrays.
The first step of the update is creating a list of susceptible people that have been in contact with infected ones. Then, the list will be sampled without replacement and regarding the infection rate parameter. Those people who have been selected will change their status to incubating.
The update of the status works similarly to a finite-state machine:
- If the person is incubating for more than 3 days, they will become infected
- If the person is infected, the next status will change either to dead, treated or immune (or stay as infected), depending on a random generated number and the mortality rate.
- Similarly, if the person is treated, it can evolve to dead, immune or stay as treated (it is more likely to evolve to the immune status).

Finally, after the simulation stops (regarding the conditions stated in the previous section), the statistics per day are stored in the database.

Differences with first draft

At the beginning of the project, the intention was to develop the application using the components and services shown in the following diagram.



The intention was to deploy the application using AWS Elastic Beanstalk, which would take care of the load balancing whenever the users accessed the web application to setup and run the simulation. The static content of the web app would be stored in an S3 bucket and provided through the AWS Content Delivery Network (CDN) in order to reduce the load on the application's origin and deliver a local copy of the content from a nearby spot. The User

Interface (UI) in the web app would also contain a Kibana dashboard with plots showing the results of the simulation. Then, the UI would connect with several AWS Lambda functions through an API gateway in order to run the simulation; which would be in charge of managing the population's updates regarding their status and location according to their agenda. These updates would be stored in a DynamoDB database.

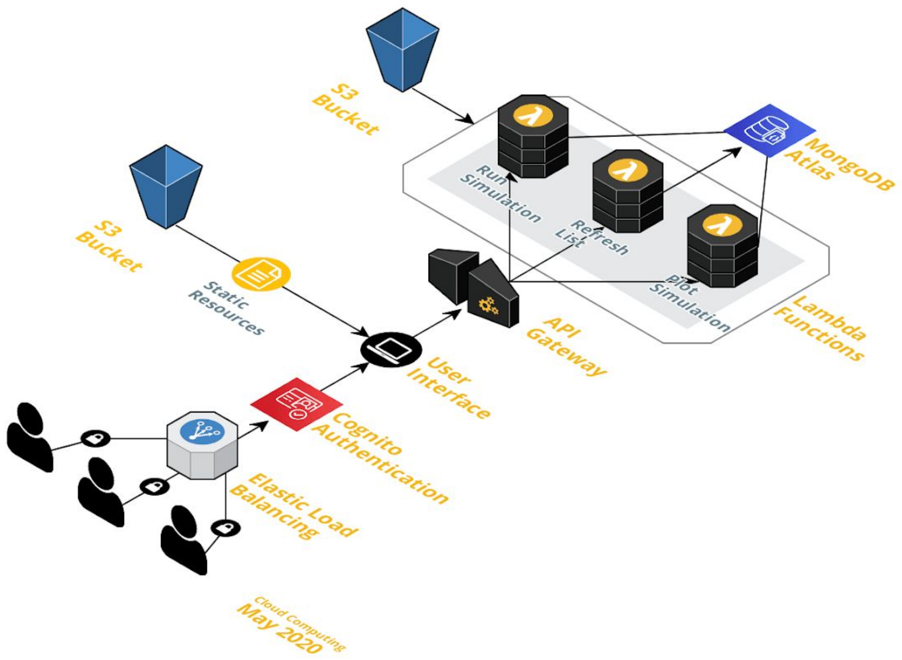
The differences between this design and the final implementation are listed below:

- The Kibana dashboard was no longer implemented. To show the results of the simulation, the web app uses a JavaScript library, Chart.js [1] instead.
- The web app is no longer hosted in an Elastic Beanstalk instance. The web app relies exclusively on the static content stored in an S3 Bucket to connect with the Lambda functions through the API Gateway.
- The number of Lambda functions needed to run the simulation was reduced from 4 to 3.
- The results of the simulation are stored in an Atlas MongoDB table instead of a DynamoDB table.
- AWS Cognito is used for the user's authentication, a functionality that was not described in the first draft.

Further details about the components used is provided in the upcoming sections of this document.

ARCHITECTURAL DESIGN

The proposed solution is based on a serverless service. The concept of "serverless" computing refers to not needing to maintain our own servers to run these functions. AWS Lambda is a fully managed service that takes care of all the infrastructure for us.



The end user interacts with the UI, the content of which is hosted inside an S3 bucket. However, before the user is able to run the simulation, it must go through an authentication step that is handled by Cognito. Once this step is completed, the user may choose between plotting the results from previously generated simulations or specifying the parameters to run a new simulation. In both cases, a trigger, activated through a button, calls a JavaScript function to validate the parameters and send the request to the API Gateway through AJAX containing the required parameters in JSON-type format.

Once the request is received by the API Gateway, the appropriate Lambda function will execute their python code to either refresh the names assigned to previous simulations, retrieve previous results to be plotted in the UI or run a new simulation according to the input parameters using the logic described in the How does it work? section of this document.

The Lambda functions can read or write data into the MongoDB cluster. It was decided to use documents to store the data because of the structure of the information and how we decided to access it.

In-memory computation logic

During the fourth sprint, we changed our approach regarding the simulation. Up until that point we relied on the database to store the information about the population. However, this approach generated a great time execution overhead, as a large amount of calls to the database were needed. We explored alternatives such as Memcached and Redis (topic covered later on). Finally we decided to use the main memory available for the lambda functions. The population information would be then generated at the beginning of each simulation, and the information that would need to be accessed later would be written to the database at the end of the process, in order to comply with the sixth factor from the Twelve-Factor App methodology [2] (the one related to organize the app as a set of stateless processes).
For the sake of simplicity and in order not to generate an excessive overhead regarding the memory usage, we assume that each person in the simulation population is represented with an integer from 0 to (N-1), being N the population size. In this manner, all the information about the people will be stored in numpy arrays in which the index in the array indicates the person to which it is related.
The information about the population is composed by three numpy arrays:

- One 1-dimensional numpy array of int8, in order to store the statuses (we only need 6 different values to store the status of a person, from 0 to 5. Thus, int8 data type allows us to do so while maintaining a low memory consumption (only one byte per person)
- One 2-dimensional array of 24 * N int16/int32 (the data type is decided on runtime), depending on the value of N. The amount of different locations depends directly on the population size. As we can address (215 - 1) different array indexes with int16 integers, if N is greater than that number, we will need to use int32 data type instead. Again, in order to maintain a simple approach, there are three kinds of locations, namely homes (80%), work places (10%) or recreational places (10%).

- One 1-dimensional array of int8, initialized with zeros, for storing the days each person has spent in their current status.

Numpy arrays were chosen as the approach for storing this data in order to have the possibility of changing the data type at will, and therefore not waste memory (the default data type for integers is int64).

Another optimization that dramatically increased the performance of the simulations' execution is the process parallelization. Specifically, we introduced parallelization during the phase of the hourly computations in the different places. In a few words, this means that whenever a process is calculating the interactions of people in Place1 (e.g Workplace), another process can do the same calculation for Place2 (e.g. Home).
In total, we launch N processes in parallel, where N is equal to the number of available processors. The results are very satisfying, as the **Speedup** of the parallel program is around **2.6**.

Studying the capabilities and limitations of our simulator.

Our simulator runs in a lambda function. These functions, at least regarding the free-tier, have just 512MB of RAM. Our logic makes an efficient usage of this main memory, nonetheless, there is an upper bound around 5 million people. A simulation with a larger population could be possible by splitting it in several executions. This approach has not been implemented, but we have done research regarding the possibilities of doing so. If we wanted to increase this limit for a simulation of a bigger city, we could run several simulations which would model different parts of the city. The communication between cities would be done by means of information about those people from one part of the city that move to other parts in the database. In this sense, each simulation would perform two actions on the database, a query at the beginning to retrieve information of those people from other parts of the city that moved to the new part, and another in the end to store the information of the people belonging to that simulation that moved to other parts of the city.

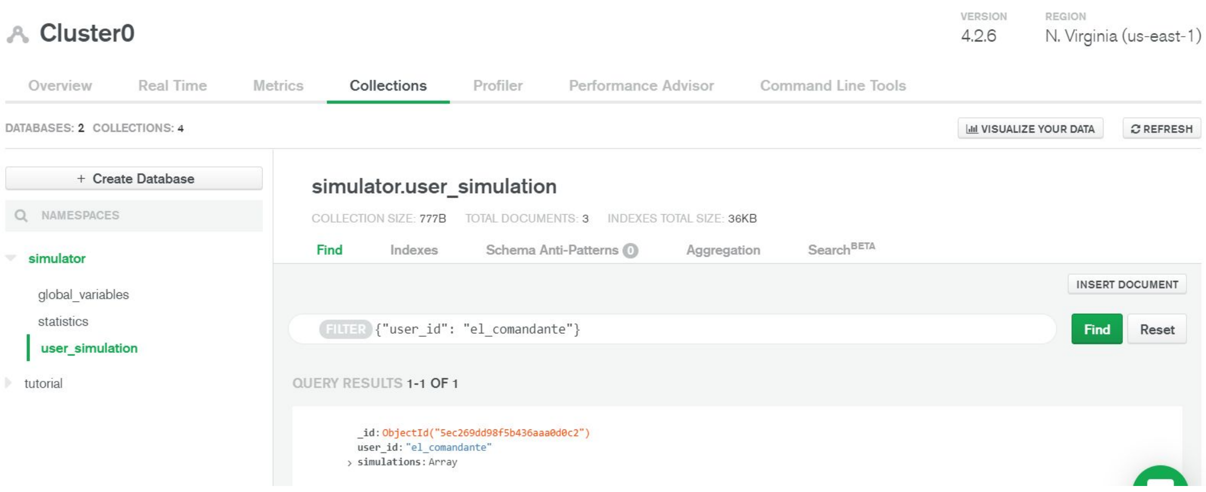
Regarding the way of scheduling the execution of different simulations, it would be managed by a queue system such as AWS SQS [3]. In this way, the complete simulation, done with several runs of the same lambda function, would be done step by step, running the simulation for the next part of the city when the previous one had finished.

Components & Services

To develop our solution we used several components that can be splitted in different groups. This section describes the components and services of each group and some additional details of the advantages of their implementation against other options.

Database

Regarding the database, we store the required data in documents inside a MongoDB cluster. To host our cluster, we used MongoDB Atlas [4] which is a global cloud database service for modern applications and specifically it is Database-as-a-Service (DBaaS).

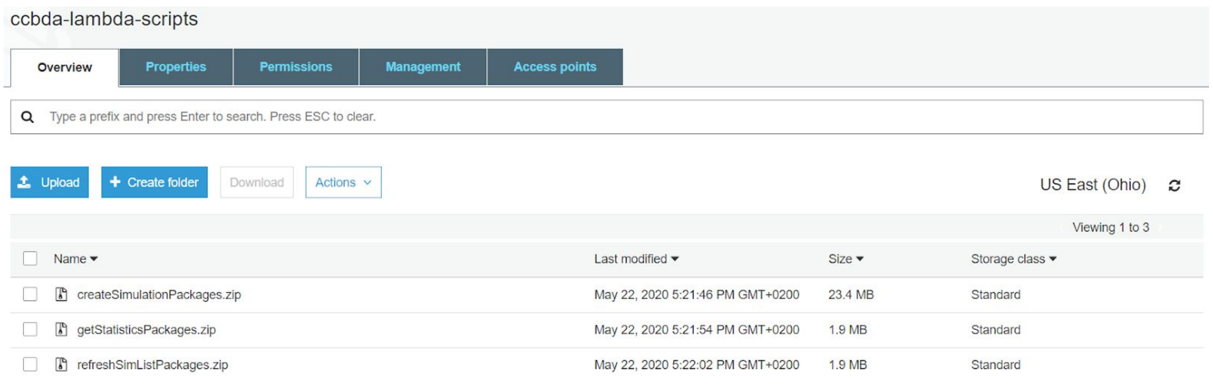


```
{"user_id": "el_comandante"}
```

```
_id: ObjectId("5ec269dd98f5b436aaa0d0c2")
user_id: "el_comandante"
> simulations: Array
```

In the beginning, we implemented the simulation by storing and accessing the data related to the population to the database. As we want to run a pandemic simulation with a big population (millions of people), we decided to use different documents to store that data. In this way, we would achieve to access faster the population data of each simulation, which in the case of a relational database would be much slower (a very big table with different partitions). However, later we changed the logic of the simulation, as we executed it entirely in memory to get better performance.
Having the data stored in documents was beneficial for our application as we did not have to define a schema in advance. In this way, we did not have to decide in advance what information we would like to store for the results of the simulations but we did it just in time that we had the real data and their structure. This gave us flexibility, because even during the last sprint, we were able to slightly change the structure of the statistics that we store in the database, to enhance the visualization in the UI.

Not only do we use databases for the storage of the data used by the simulator, but also to store the scripts for the whole application. In this case, we created an S3 bucket that would store the different zip files that update the Lambda functions. Each function has its corresponding zip, with the libraries that it requires to run.



Name	Last modified	Size	Storage class
createSimulationPackages.zip	May 22, 2020 5:21:46 PM GMT+0200	23.4 MB	Standard
getStatisticsPackages.zip	May 22, 2020 5:21:54 PM GMT+0200	1.9 MB	Standard
refreshSimListPackages.zip	May 22, 2020 5:22:02 PM GMT+0200	1.9 MB	Standard

Having such files allowed the update of the Lambda functions utilizing an AWS CLI script, rather than manually having to upload the zip every time we wanted to do a deployment. Such scripts for both uploading the files and updating the Lambdas are as follows:

```
#Update Zips in S3
aws s3 cp createSimulationPackages.zip s3://ccbda-lambda-scripts
aws s3 cp getStatisticsPackages.zip s3://ccbda-lambda-scripts
aws s3 cp refreshSimListPackages.zip s3://ccbda-lambda-scripts

#Update Lambdas
aws lambda update-function-code --function-name simulation-main
--s3-bucket ccbda-lambda-scripts --s3-key
createSimulationPackages.zip
aws lambda update-function-code --function-name get-statistics
--s3-bucket ccbda-lambda-scripts --s3-key getStatisticsPackages.zip
aws lambda update-function-code --function-name run-simulation
--s3-bucket ccbda-lambda-scripts --s3-key refreshSimListPackages.zip
```

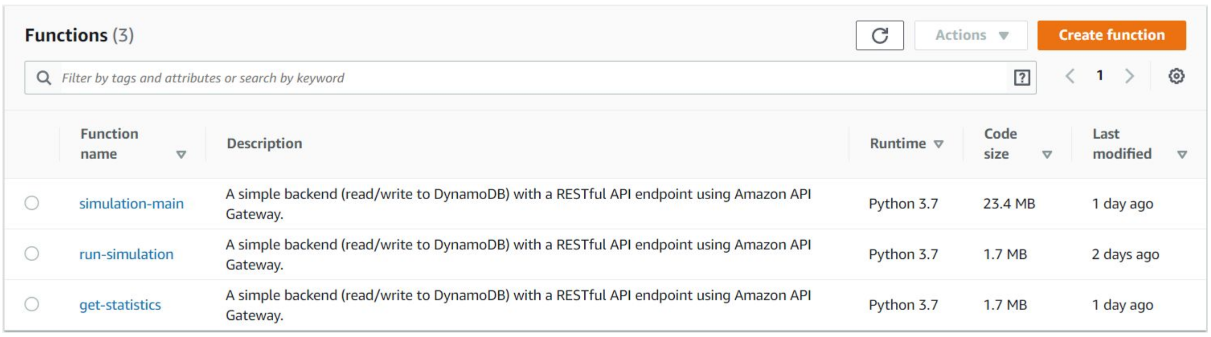
Apart from Atlas MongoDB, other alternatives were also taken into consideration. For instance, we were initially thinking of using AWS DocumentDB service that supports MongoDB, however we rejected this scenario, as its cost was much higher than the one of Atlas. Similarly, as we have already mentioned, AWS DynamoDB was another alternative which was more expensive than Atlas, hence it was also rejected.

Regarding the storage of the population data, we researched the use of AWS ElastiCache as an alternative to use the Lambdas memory (512MB). Within AWS ElastiCache we had Redis and Memcached available as options for deploying our in-memory data structures related to the population interactions, statuses, agendas, etc. Both options persist data in RAM, belong to the NoSQL data storage family, and keep data stored in a key-value data model. While Memcached is suggested to be used for scenarios where the data you keep is static and you want to be able to scale quickly, Redis is designed for more complex storage scenarios, with its multiple data capabilities and more storage per key compared to Memcached. Though both choices seem to be useful for our requirement, the free tier of both options only allows the use of a t1.micro instance, which is limited to 512MB of storage (same as the Lambda functions). Moving forward, there is definitely a need for the application to be able to scale for a population of hundreds of millions, but for this specific prototype application it didn’t make sense to deploy them, as the memory of the Lambdas serves the purpose.

Serverless Architecture

When you look at examples of AWS serverless architecture applications [5], the main component that drives the execution of the code are AWS Lambdas. Such assets allow users to run code for virtually any type of application or backend service - all with zero administration and provisioning or managing servers. We would only pay for the compute time consumed by executing the code. Thus, for this specific scenario where the users would trigger a simulation and wait to see the results, it seemed the right choice to move forward. In this case, three Lambda functions are being used to manage the whole application, each with a different size depending on the amount of libraries that the script requires for running. The zip uploaded to update each function will have the complete list of libraries the scripts utilize:

- simulation-main: its purpose is to execute the whole simulation logic. It receives several parameters (population size, mortality rate, infectious %, etc.) and with such, generates a population with different statuses and executes the simulation over the desired period of time or until convergence (no more infected entities).
- run-simulation: it is utilized to retrieve the different simulations that a Cognito user has stored within the database. With it, a user can plot a simulation previously run, rather than having to re-run a simulation with the same parameters used before.
- get-statistics: this function makes a request to Atlas MongoDB and retrieves the daily parameters of the simulation specified. Such parameters are utilized by the UI to plot the behavior of the simulation.



Function name	Description	Runtime	Code size	Last modified
simulation-main	A simple backend (read/write to DynamoDB) with a RESTful API endpoint using Amazon API Gateway.	Python 3.7	23.4 MB	1 day ago
run-simulation	A simple backend (read/write to DynamoDB) with a RESTful API endpoint using Amazon API Gateway.	Python 3.7	1.7 MB	2 days ago
get-statistics	A simple backend (read/write to DynamoDB) with a RESTful API endpoint using Amazon API Gateway.	Python 3.7	1.7 MB	1 day ago

A single API Gateway with three different routes is utilized to manage the three Lambda functions. Each Lambda has a distinct API endpoint which is called by the UI, which triggers an event by the API Gateway to execute the lambda_handler within each script. Depending on which one of the Lambdas is triggered, it will return or not a body in the response, together with the 200 status code if everything runs accordingly.

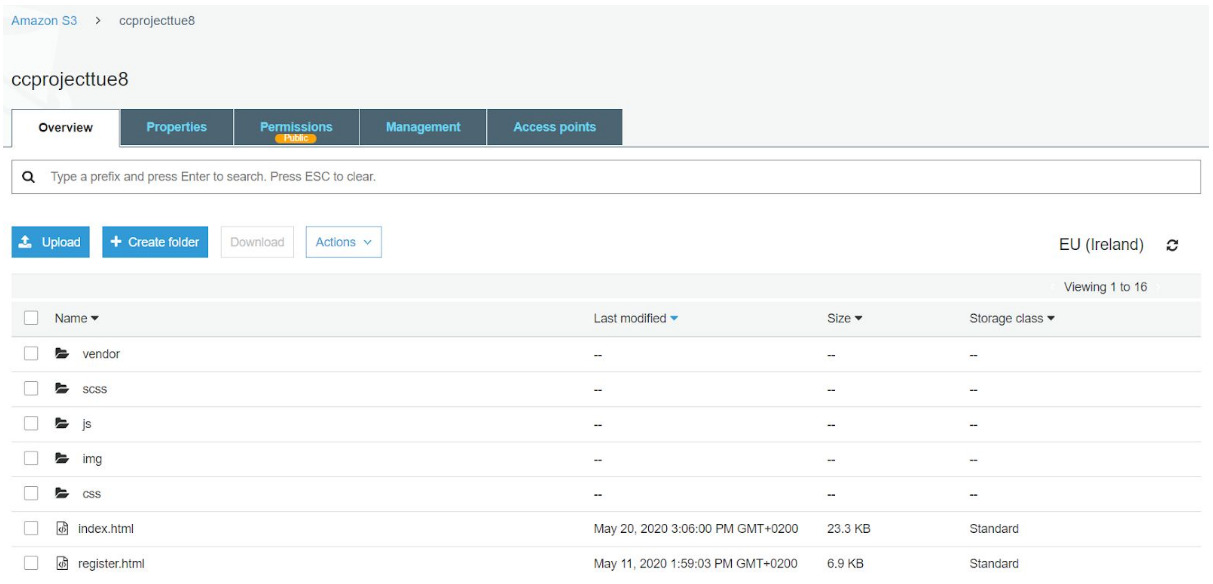
Besides AWS Lambda, we had other options available from AWS to use for the purpose of the application. Specifically, we researched regarding AWS Batch [6] and AWS Fargate [7]. These are the reasons why we decided to continue utilizing AWS Lambda rather than any of those two options:

- **AWS Batch**, though it would provide the specific computational resources needed for executing our scripts, it appears in the documentation available that this option is more suitable for queue jobs. From a design standpoint, we could've adjusted our code to execute in a queue fashion to process the interactions, but it would cause a lot of data communication between those jobs and our database or repository where the population data would be stored. Because of this exchange of data within the jobs, which continues to grow as the simulation runs, it didn't seem to be the best option from a workload perspective.
- **AWS Fargate**, while providing a container to deploy the whole application which can run for as long as the script needs it to, appears to be more of a scheduled job purpose like. Examples shown on the AWS documentation refer to cron jobs where they are executed every day/week/etc. for specific business scenarios. As our requirement wouldn't fit into a cron job, but more a request-service one, Lambda still appears to be the better option.

User Interface

Regarding the User Interface (UI), it was decided to use HTML and CSS code for the forms and JavaScript and JQuery to dynamically manage the information inside the forms. Additionally, JavaScript charting (through Chart.js) is used to visualize the results of the simulation.

These files, that make up the static content of the application, were stored in an S3 bucket, which was configured for Static Website Hosting. Once public access was granted to the bucket and the *login.html* file was configured as the Index document, the files that had been previously loaded to the bucket could be accessed through the endpoint provided by AWS. In later stages of development, the team was able to make simulation tests directly from this website if necessary, given that the current usage of the bucket was not as high to make the team incur in costs because of its usage.



Name	Last modified	Size	Storage class
vendor	--	--	--
scss	--	--	--
js	--	--	--
img	--	--	--
css	--	--	--
index.html	May 20, 2020 3:06:00 PM GMT+0200	23.3 KB	Standard
register.html	May 11, 2020 1:59:03 PM GMT+0200	6.9 KB	Standard

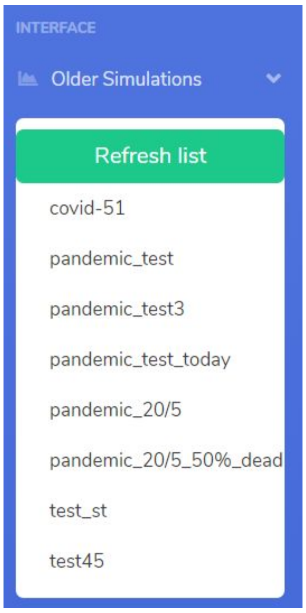
Since the content of the bucket was also stored in the project's Github repository, making updates to the static content could be done easily through the use of the following command:

```
#Update S3 Bucket
aws s3 sync . s3://bucket-name --exclude ".idea/*"
```

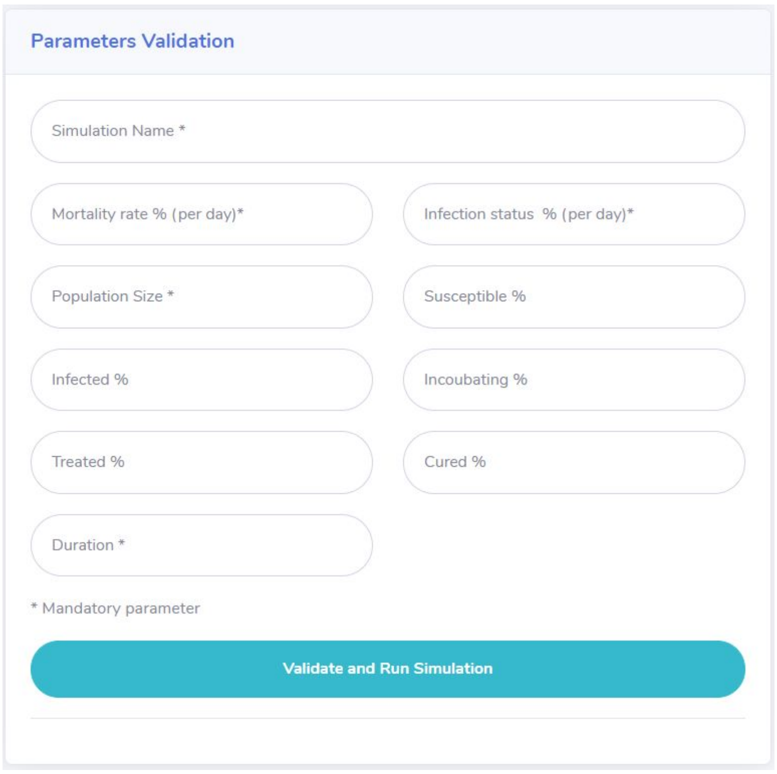
Additionally, it is possible to establish an SSL connection for the static website through the creation of an AWS CloudFront Distribution. By specifying the S3 Bucket as the Origin Domain Name for the Distribution, AWS gives developers the ability to allow end users to use either HTTP or HTTPS to access the contents of the bucket. The Distribution uses a default SSL CloudFront Certificate, providing a secure link between the web application and the end user's browser.

The main page of the website contains several components that interact with the API Gateway described in Serverless Architecture. Those components are described below:

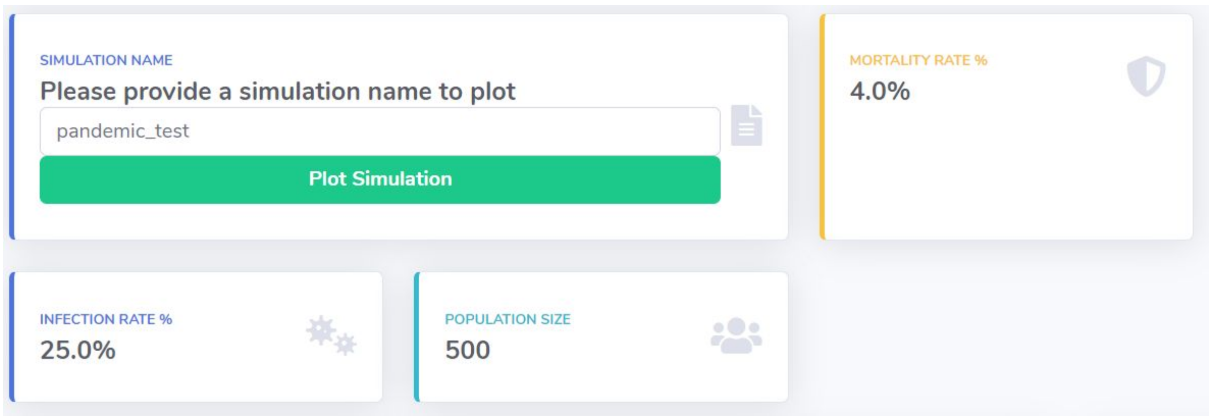
- <u>Refresh List:</u> Clicking on the button will trigger a JavaScript function that retrieves the username from the current session and sends a request to the API Gateway through the jQuery ajax() method to obtain a list with all simulations associated with that username.



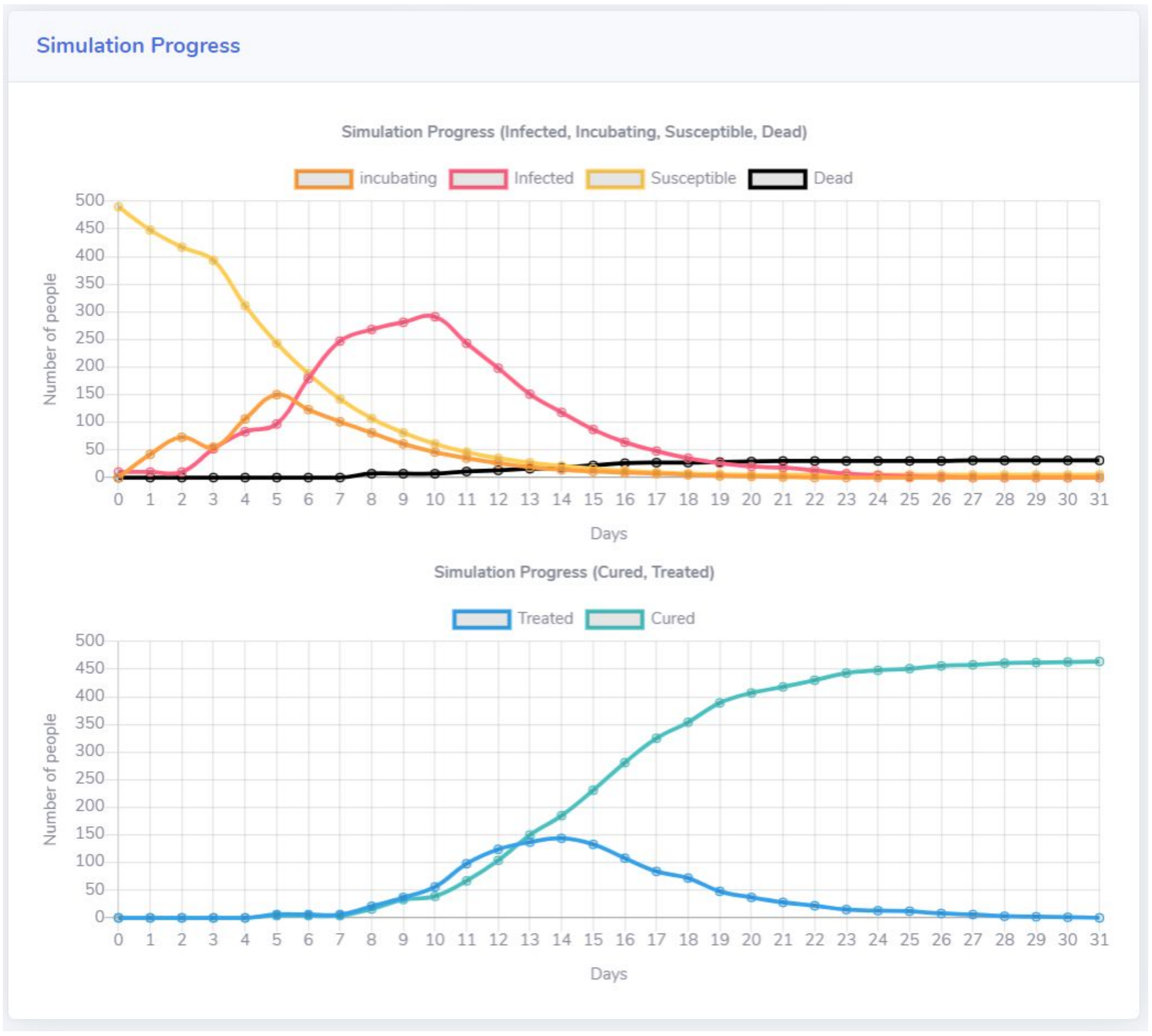
- <u>Validate and Run Simulation:</u> Clicking on this button will trigger a JavaScript function that validates the correctness of all the parameters in the form before sending the request to the API Gateway. Once the request is sent, the website will show a loading screen until it receives a response. The ajax() method has different code implemented to alert the user in case of success or error.

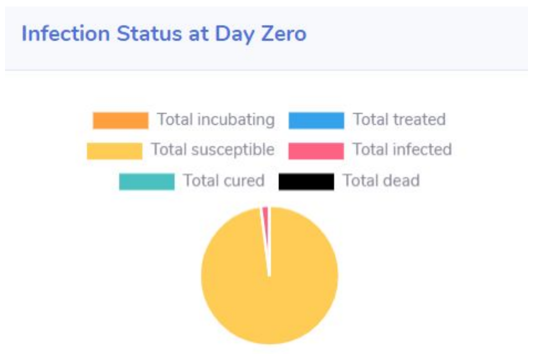


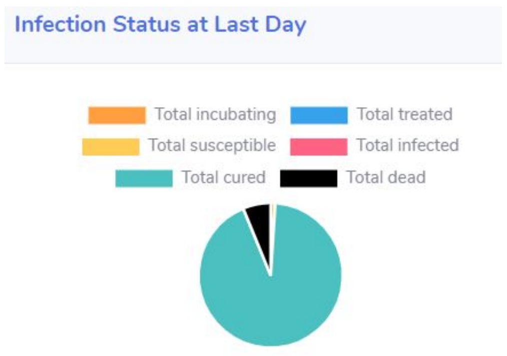
- Plot Simulation: Clicking on this button will trigger a JavaScript function that retrieves the username from the current session and simulation name introduced by the user to send a request to the API Gateway to obtain the simulation results and plot them in their corresponding charts.



The following images show an example of the plots displayed in the User Interface.







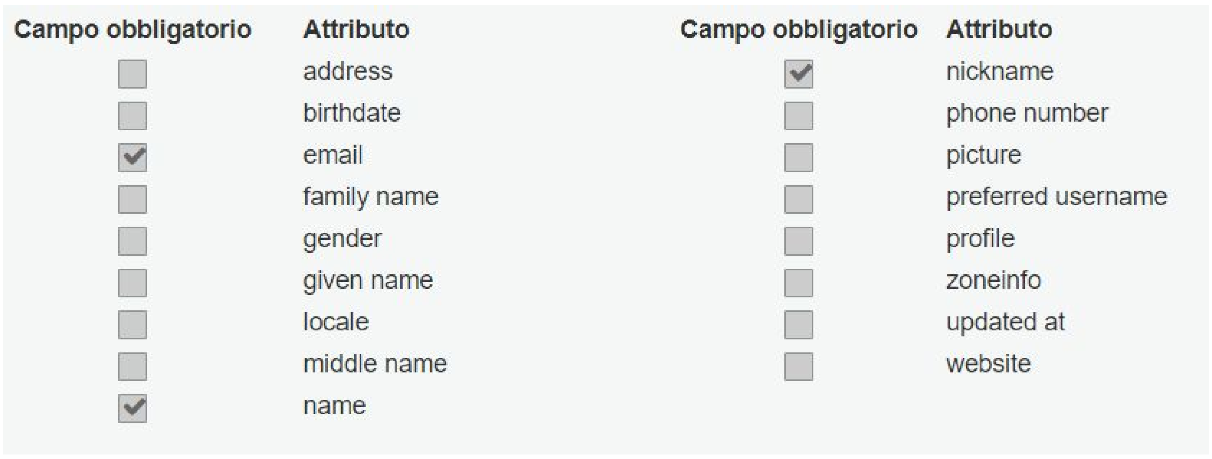
Other alternatives to develop the UI were discussed at early stages of the project, however, they were discarded for different reasons, which are explained below.

- **Web App Hosting**: Instead of hosting the app through an S3 bucket, it was also possible to use an Elastic Beanstalk instance. According to AWS [8], this alternative implementation would automatically handle the deployment, from capacity provisioning, load balancing, auto-scaling to application health monitoring, and would allow the team to configure this instance to run with Python, providing the necessary tools to run the simulation. This implementation was initially considered for the project, as it was developed during the first two sprints of the project, including a batch (and shell) script to create the instance with log monitoring in a matter of minutes. This implementation was discarded, however, when the team realized that the intended design of the project did not take advantage of Elastic Beanstalk, since the content of the web app was static and the python code was being developed inside the Lambda functions. Given that the use of the web app was not considered high, the fact that an S3 bucket provided the necessary scalability necessary for the project and the difference in prices (*"S3 Standard Bucket - $0.004 per 10,000 GET requests"* [9] against *"EC2 t2.micro instance - $0.0126 per Hour"* [10]), there was no reason to continue using an Elastic Beanstalk instance.
- **Charts Plotting**: The results from the simulation could have been presented to the user in a Kibana dashboard. Kibana offers intuitive charts that can be used to navigate through large amounts of log data and their pre-built aggregations and filters can run a variety of analytics like histograms [11]. However, the strong point of Kibana, analytics and monitoring, were not required for the current implementation of the project. Instead, the UI needed a tool capable of quickly updating the plots depending on the simulation to be displayed without the need to make additional connections with an Elasticsearch Service domain (which includes a built-in Kibana) that would further increase the complexity of the project.

Cognito

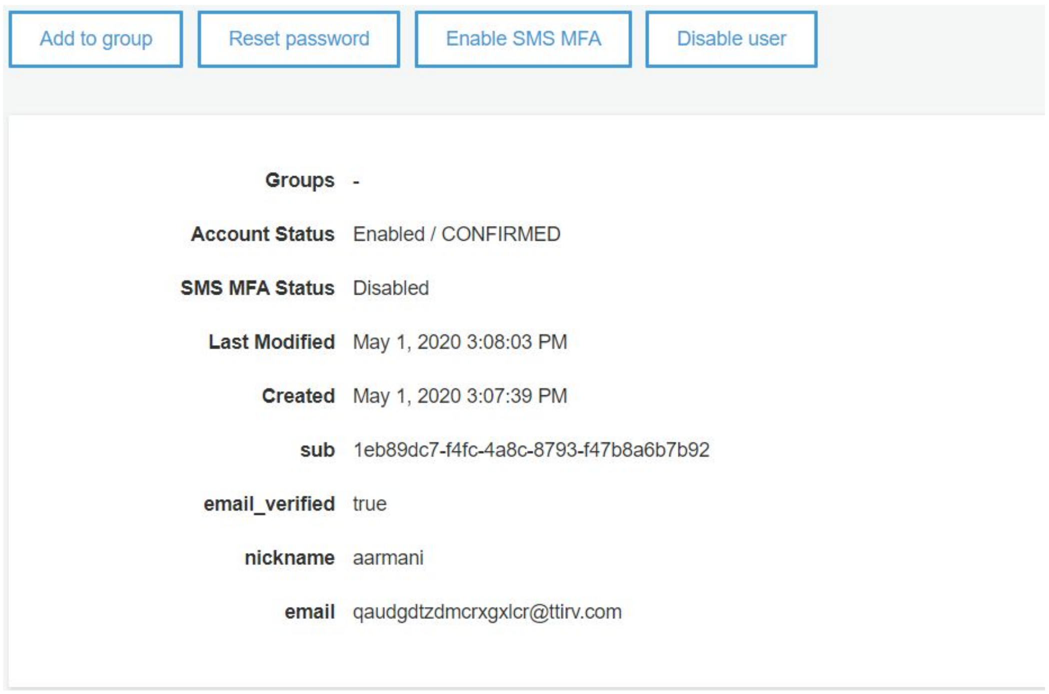
During the development of the UI, we faced the problem to create an authentication system and to do so we used AWS Cognito.
This service allows the user to login in two different ways: using a Federate Identity or a User Pool. The first approach allows the user to authenticate from a variety of providers, such as Google, Facebook, Twitter or Amazon.

The second approach is to create a User Pool that is specific for our application. In such a case, a user has its own credentials just for our webapp, so it requires a signup phase. Although this process is slightly more tricky than Federated Identity, it allows us to store much more detailed information about the user. By default, Cognito provides us some predefined fields to add to our user, however it's also possible to store new features:



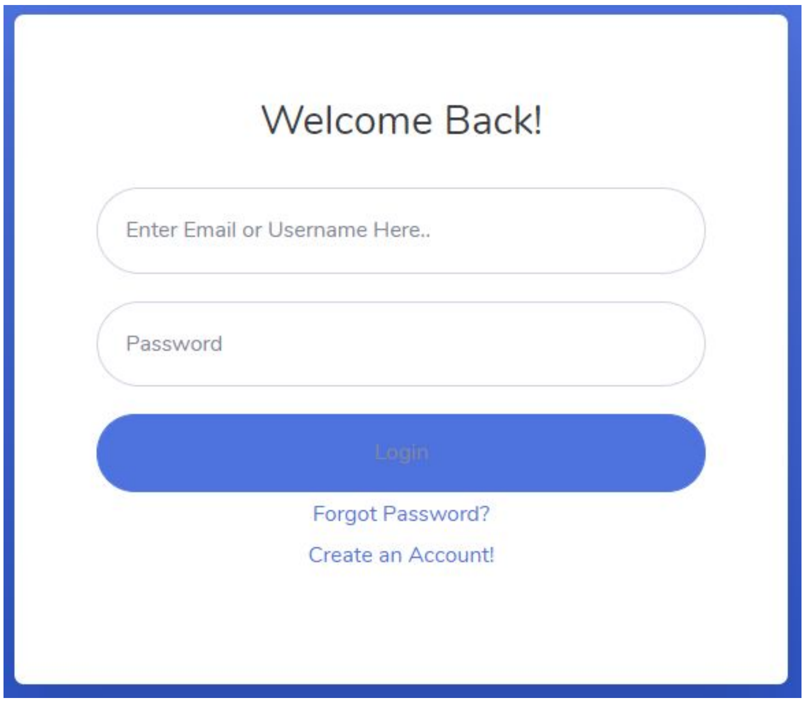
Campo obbligatorio	Attributo	Campo obbligatorio	Attributo
☐	address	✓	nickname
☐	birthdate	☐	phone number
✓	email	☐	picture
☐	family name	☐	preferred username
☐	gender	☐	profile
☐	given name	☐	zoneinfo
☐	locale	☐	updated at
☐	middle name	☐	website
✓	name		

In our webapp we check the user identity with a registration link to its mail, however Cognito allows us to perform 2-FA by sending a verification code to the cellphone number, but it is a very costly approach because the price for each message is defined by an international organization and it varies according to the company and the country in which the cell phone is registered. For instance, in Spain the cost for each SMS is very high because it varies between 0.07$ to 0.08$, while in the USA it is fixed at $0.00645 [12].
Our webapp implements the User Pool approach, however both techniques could be used.
As for the administration side, Cognito easily allows the management of users: it's possible to create specific roles, deactivate account and also delete them.



Groups	-
Account Status	Enabled / CONFIRMED
SMS MFA Status	Disabled
Last Modified	May 1, 2020 3:08:03 PM
Created	May 1, 2020 3:07:39 PM
sub	1eb89dc7-f4fc-4a8c-8793-f47b8a6b7b92
email_verified	true
nickname	aarmani
email	qaudgdtzdmcrxgxlcr@ttirv.com

Once the User Pool is ready, Cognito generates an API to several functions that will be called from the UI. For instance, our Login page is:



Once the form is correctly filled, a request to Cognito is sent and, if successful, we redirect the user to the simulation page. From a technical perspective, if the request is satisfied Cognito sets a session for the logged user inside the browser that will be later deleted in the logout phase.
With the same logic, Cognito APIs allow the user to change its password and create an account. During this latter operation, Cognito also sends a verification link to the user and as long as this link is not opened, the account will be disabled. Therefore, the user won't be able to access the simulation page.

TWELVE-FACTOR METHODOLOGY FOR SAAS

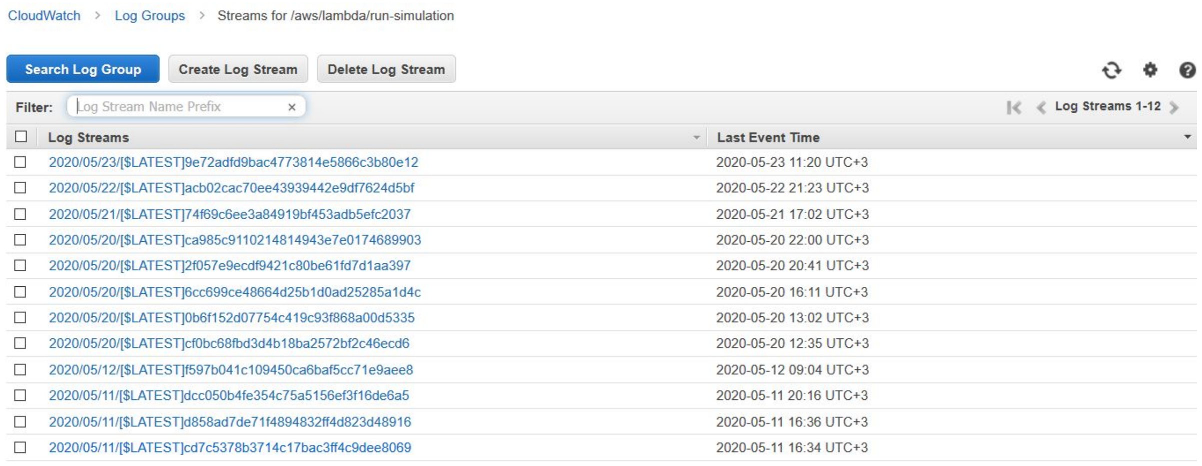
During the application development, the team strived to stick to the Twelve-Factor methodology. Our goal is the implementation of a software that maximizes the portability in different execution environments along with a continuous development approach that strives to minimize the divergence between development and production and the possibility of effectively scaling up without significant changes to tooling, architecture, or development practices.
Working as a team on a project implies that each member continuously updates the code and adds new features. As a consequence, there is an inherent risk caused by the heterogeneity of sources from whose final product is developed. Therefore, we considered vital to perform the version control of our project and we used Github for such purpose. There are several deployments of the same codebase, namely one development

deployment per team member, as well as a production deployment where the changes performed in the development ones are included when finished.
In order to declare our codebase dependencies and isolate them, regarding the Python code, we are managing them with pip for declaring, and Anaconda for isolating.
The configuration is stored in environment variables. At the beginning they were stored in a python file that was not uploaded to our repository, but then we changed the approach. All the database handlers, as well as the credentials for accessing MongoDB Atlas and AWS components are stored in environment variables. Thus, each deploy has its own environment variables, that can be (or not) different from the other ones.
Our software takes advantage of several backing services to work, such as S3, MongoDb Atlas or Cognito, as it has been explained earlier in the report.
The way we attempted to follow a correct build, release, run approach was through the use of Github and the S3 buckets to deploy the code for the Lambda functions. Such structure allowed us to automate the process, while making sure that there was a correct version deployment aligned to what we all had developed. Whenever code was fully tested locally, the developer pushed such commits to Github and notified the others that it was ready for deployment. Later on, such commits were pulled, archived on the zip and uploaded to the S3 bucket for the Lambda functions to fetch them and eventually be run.
Since we developed a webapp, we decided to store all the data generated by the users using MongoDb Atlas, a Database-as-a-Service resource that allows us to save data in a MongoDb fashion. However, the DbaaS is not suitable to store user's credentials, therefore we integrate our software with Cognito, an AWS service that is built to securely store and easily manage the authentication process.
We developed a Serverless architecture, where our UI is stored in an S3 bucket and the communication between the frontend and the backend is via Lambda APIs. This approach brings two benefits: on the one hand, the UI doesn't rely on runtime injection from a web server into the execution environment to create a web-facing service, and on the other hand Lambda APIs can naturally scale and avoid concurrency problems. Moreover, Lambdas are very helpful in terms of disposability, meaning that our webapp services can be started or stopped at any point in time in a graceful manner by disabling them.
Historically there have been substantial gaps between development and production. The gaps manifest in three different areas:

- The time gap: A developer may work for a long time before sending the code to production;
- The personnel gap: Developers write code, ops engineers deploy it;
- The tools gap: Developers may be using a different stack than production;

The way in which we structured our project and the organization of our team allow us to avoid these problems. All the members of our team are both developers and ops, therefore we never faced the personnel gap. Furthermore, we had scrum meetings every day in order to be on the same page and make decisions all together. Everytime we needed to add a new part to the system, we researched different options to achieve the result and later during a meeting we shared our findings and we decided what's the best software to use. Therefore, we ensure that our stack is the same. Finally, since the beginning of the project we didn't fix a role for each member because we wanted to be as flexible as possible: considered that we were dealing with a complex problem, we thought it was better that everyone could work in every aspect of the project. This extreme flexibility allowed us to reduce the time gap as much as possible by reassigning at every sprint a right number of people to perform a specific task.

Regarding the logs, since the beginning of the implementation we realized their importance as the logs play mainly two roles, namely the debugging of the lambda functions as well as the system monitoring. For these reasons, we used AWS CloudWatch, which provides metrics to monitor the performance of the architecture, logs and many other features. A sample of the logs can be found in the following screenshot:



Log Streams	Last Event Time
2020/05/23/[$LATEST]9e72adfd9bac4773814e5866c3b80e12	2020-05-23 11:20 UTC+3
2020/05/22/[$LATEST]acb02cac70ee43939442e9df7624d5bf	2020-05-22 21:23 UTC+3
2020/05/21/[$LATEST]74f69c6ee3a84919bf453adb5efc2037	2020-05-21 17:02 UTC+3
2020/05/20/[$LATEST]ca985c9110214814943e7e0174689903	2020-05-20 22:00 UTC+3
2020/05/20/[$LATEST]2f057e9ecdf9421c80be61fd7d1aa397	2020-05-20 20:41 UTC+3
2020/05/20/[$LATEST]6cc699ce48664d25b1d0ad25285a1d4c	2020-05-20 16:11 UTC+3
2020/05/20/[$LATEST]0b6f152d07754c419c93f868a00d5335	2020-05-20 13:02 UTC+3
2020/05/20/[$LATEST]cf0bc68fbd3d4b18ba2572bf2c46ecd6	2020-05-20 12:35 UTC+3
2020/05/12/[$LATEST]f597b041c109450ca6baf5cc71e9aee8	2020-05-12 09:04 UTC+3
2020/05/11/[$LATEST]dcc050b4fe354c75a5156ef3f16de6a5	2020-05-11 20:16 UTC+3
2020/05/11/[$LATEST]d858ad7de71f4894832ff4d823d48916	2020-05-11 16:36 UTC+3
2020/05/11/[$LATEST]cd7c5378b3714c17bac3ff4c9dee8069	2020-05-11 16:34 UTC+3

Finally, we tried to distinguish the administrative tasks from the rest of the tasks. Specifically, we focus on scripting as much as possible the deployment of the required infrastructure, in order to minimize the time to market. For instance, before moving the HTML content to the S3 bucket, we were hosting our website in AWS Elastic Beanstalk and we had developed a python script to automatically create all the component's stack. Similarly, as we have already mentioned in a previous chapter, we developed scripts to generate and upload the packages that are used by the lambda functions. Furthermore, we developed an initialization database script that imports a couple of collections to the database that are later used by the simulations. Some other administrative tasks related to the project execution were cleaning and updating the information in Trello. Each one was responsible for completing their assigned tasks, while also adding or updating others if necessary within the tool. Moreover, while arranging our meetings, someone has always been in charge of hosting the call and making sure everyone was notified of the start time. Finally, we all made sure that the costs related to the assets used in AWS never went over the threshold for the free tier. This was a task executed not only by the owners of the account, but everyone had to be aware when developing or testing something.

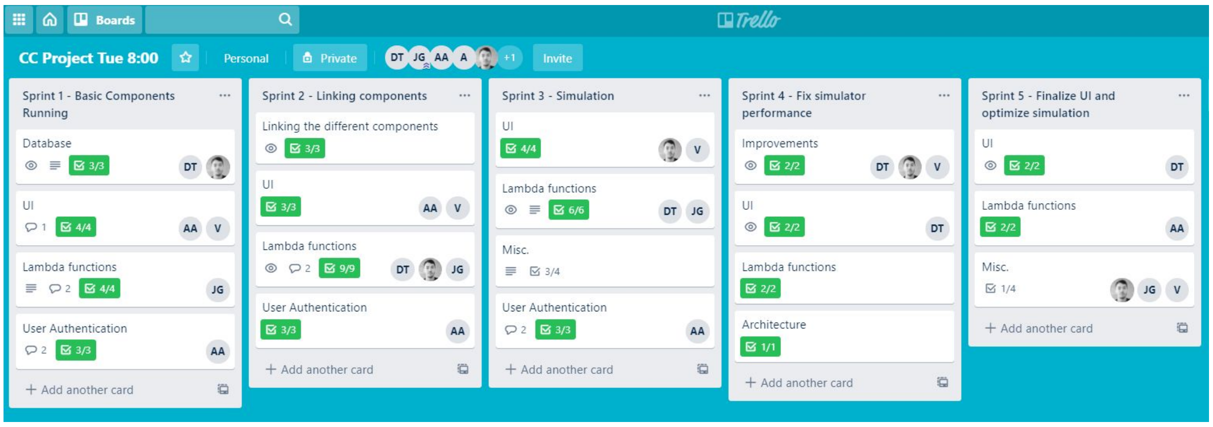
PROJECT METHODOLOGY

Division of responsibilities

Based on the activities defined during the first draft, at the beginning of every Sprint, the team had a Sprint Planning Meeting to discuss which components or functionality of the application would be addressed in that particular week. Once that was established, the team identified the complexity of such tasks in order to assign an adequate number of people to it, to ensure they could be completed in time.

No particular activity was permanently assigned to a team member, although in some cases it made sense to keep the same person working on the same activity for more than one sprint due to the increased knowledge acquired by that person in the previous weeks. However, being able to change from one activity to another provided the whole team with the flexibility and the required knowledge to be able to help in different areas if necessary.

To better organize and visualize the work of each member for every sprint, we used a Trello Board. Such a tool is vital for a project that adopts an Agile methodology as each member is able to see what he has to do weekly without keeping draft notes that cause confusion. Furthermore, each member is able to write notes and comments to the activities of the other members, which is also very useful because it allows everyone to see the changes that he has to do as well as track the proposals and the changes that have already been done in the past. In the end of the sprints, our board looks like this:


Boards
Trello
CC Project Tue 8:00
Personal
Private
DT JG AA A +1
Invite
Sprint 1 - Basic Components Running
Database 3/3 DT
UI 1 4/4 AA V
Lambda functions 2 4/4 JG
User Authentication 2 3/3 AA
Add another card
Sprint 2 - Linking components
Linking the different components 3/3
UI 3/3 AA V
Lambda functions 2 9/9 DT JG
User Authentication 3/3 AA
Add another card
Sprint 3 - Simulation
UI 4/4 V
Lambda functions 6/6 DT JG
Misc. 3/4
User Authentication 2 3/3 AA
Add another card
Sprint 4 - Fix simulator performance
Improvements 2/2 DT V
UI 2/2 DT
Lambda functions 2/2
Architecture 1/1
Add another card
Sprint 5 - Finalize UI and optimize simulation
UI 2/2 DT
Lambda functions 2/2 AA
Misc. 1/4 JG V
Add another card

Meetings

The team tried to stick to the SCRUM methodology by having a daily SCRUM meeting whenever possible. In these meetings, each team member described what he had been working on since the last meeting, what he planned to do for that day, and the roadblocks faced while working on the assigned tasks. Depending on the topics discussed during these meetings, additional meetings were scheduled between the team members involved to further discuss the topic, either to synchronize efforts on different fronts, suggest new ideas or solve issues of any kind. In this way, we managed to disengage the persons that were not needed for the individual discussions to focus on their tasks and hence make more productive work.

Exchange of ideas

The exchange of ideas occurred mostly during team meetings, where proposals to move the project forward were discussed in terms of feasibility, complexity, and available resources. Most of the time, these proposals were backed up by previous research made by the person(s) suggesting them, in order to expose to the other team members the benefits and downsides to its implementation and decide what steps to take next. These interactions resulted in different upgrades to the project overall, such as the use of MongoDB Atlas and the parallelization of tasks in the simulation to save execution time.

Documentation

During the implementation of our application, we created documentation especially for the different python functions that we developed. Taking into account that five people were working together at the same time, we had to orchestrate the tasks that each one had to do, as there were many dependencies between them. The biggest challenge that we faced was the development of the code for the different components. In other words, most of the time each member was coding something, the output of which later will be used by someone else. This implies that the functionality of the code had to be well documented otherwise the reusability or changes to the specific piece of code would be impossible. To tackle this problem, we decided to write extensive descriptions (comments) right before the definition of each function, including the structure of the input, the output and its functionality.

Working environments

In every IT project, it is vital to maintain different working environments, namely development (dev), user acceptance test (uat) and production (prod) environments. By applying this strategy, the number of troubles that can be caused when different environments overlap each other is minimized and the performance is increased, especially when the product is launched to the market and the workload of the prod environment is high.
In our case, we basically used dev and prod environments. Regarding the unit testing of the different components (e.g. UI, python code etc...), we used our local machines (personal computers) as dev environments . For hosting the website, we were running the HTML code in the local host of our computer (browser) while for developing and testing the python code, we used PyCharm IDE. Whenever each one of us had a component running, he uploaded the working code in Github and we also deployed the code to the corresponding prod environment (to the S3 bucket for the web content and to the Lambda functions for the python code).
In addition to that, we also used different environments for the database. Specifically, we created two different accounts to Atlas MongoDB, one to be the dev and another one to be the prod env. Similarly, whenever we wanted to develop or test a new functionality, we were using the dev db while whenever we had a ready component, we were switching to the prod

db. By following this strategy, we managed to have the latest version of our product ready to be used in one environment (prod) and we were also able to switch back to an older version, everytime that the dev environment crashed because of a new unsuccessful change that we tried.

Additional tools

Anaconda

One of the tools we utilized during the development for maintaining a structured environment of deployment is Anaconda. We created an environment with Python 3.7 in which we installed all the required libraries for the script to run. Examples of such libraries were: pymongo, numpy, json, bson, collections, etc. Furthermore, the use of this environment allowed us to later on store such libraries on a zip file to be deployed to the Lambda functions together with the scripts. It served not only the purpose of documentation and development management, but also as a deployment structure for our scripts.

AWS CLI

The AWS CLI allowed the creation of resources into AWS, such as S3 buckets and Lambda functions. Not only that, but it allowed us to automate scripts utilized for updating the Lambda functions. Everytime a new version of the code had to be deployed to the Lambda functions, we utilized commands from the AWS CLI to upload the zip containing the updated scripts to the S3 bucket and then requesting the Lambda function to pull the updated zip from that same bucket. That generated a more controlled and systematic approach to do updates, rather than manually uploading such zips to each function.

MAIN PROBLEMS ENCOUNTERED

<u>Population Storage:</u>
One of the biggest problems that had to be solved was the storage and the fetch of the population of each simulation, as each simulation can have a couple of million of persons. It was concluded that the best approach is to use MongoDB and have different documents for each population (e.g. population_4 for simulation with id = 4). In this way, whenever it is required to fetch the data of a simulation, it is not necessary to involve the other documents, otherwise this process would be extremely inefficient. For example, using relational tables, the database would have one single table for all the simulations,containing the information for the populations. This table would be extremely large because with 1.000 simulations with 1.000.000 persons each, the size of the table would be 1.000.000.000, which cannot be efficiently queried. Although this approach was later modified, solving this problem made the team learn about a different tool for storage in the shape of MongoDB, which was still used for the final implementation.

<u>Amazon S3 Requests:</u>
When the web app was migrated from the Elastic Beanstalk instance to an S3 Bucket, it was necessary to upload more than 1800 different files to the bucket, which, after some tests and

updates, quickly topped the Free Tier usage provided by AWS (2,000 Put, Copy, Post or List Requests of Amazon S3). After performing an analysis to identify a solution, it was identified that most of those files were exclusively style (CSS) files belonging to a particular set of tools, Font Awesome, which accounted for more than 1600 files of the static content. All these files were replaced by a couple of lines of code in the HTML files:

```
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.13.0/css/all.css">
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.13.0/css/v4-shims.css">
```

With those links in place, the visualization of the web app remained unchanged, but the difference in the use of the available requests was greatly improved.

Simulation performance

The biggest problem that we faced was the poor performance of the simulation, which is the main task of our website. The source of the problem was coming from the intensive I/O to the database, as in our initial approach, all the information for the simulation was stored in the database. We tried several ways to tackle this problem, like using Memcached to reduce the database I/O or parallelization of the request to the database. In the end, we concluded that a hybrid approach of them was the best way to run a simulation. We efficiently keep the population and the rest of the information of the simulation in memory, do all the required calculations and store the resulting statistics in the database. We also parallelize the computations, as we have already described in the architecture chapter.

Long scrum meetings

Another issue that we faced was related with the way we orchestrated the different tasks that we had to do to complete the project. Specifically, we made several meetings during one week that lasted more than the scheduled time (instead of 5 minutes, they took more than 30 minutes). The fact that we had additional work to do for other courses, especially during the start of each sprint, did not let us organize a long meeting to define in detail all the tasks for the current week. That is why we had to prolong some of the daily meetings in order to solve any misunderstanding or to clarify what is required.

Single account usage

While working on deploying and updating any of the resources from AWS that we were using, it became a “bottleneck” issue at some point due to the use of an account, or multiple, for different purposes. Though we manage to split the deployment of resources in different accounts, to try and prevent this issue from happening, whenever new code had to be deployed to the Lambdas or S3 buckets, tests had to be run between the UI and the Lambdas, verifications of data ingestions into the database, and many more cases, we encountered issues in which the user who created such resources had to be present to be able to move forward. It generated not the best environment for self-working, but made sure at least that we all were communicating frequently and aligning on what’s the status of what each one is working. In the future most probably a generic account with shared credentials would be the best approach to resolve such issues.

TIME INVESTED

The following table shows the hours invested by each team member in the different tasks of the project.

Activity	Team Member				
	Andrea	Dimitris	Jesus	Ricardo	Valdemar
Project Management	7	10	10	10	10
Architectural Design	5	2	2	2	5
Deliverables	5	5	5	5	5
User Interface	5	15	1	8	10
Database	1	4	5	4	0
Lambda Functions	15	20	20	20	0
Batch Operations	0	0	3	0	10
User Authentication	8	0	0	0	0
Testing	15	4	6	4	10

The next table shows the total amount of hours used for each task compared to the original forecast created for the first draft.

Activity	Forecast	Actual	Difference
Project Management	50	47	-3
Architectural Design	5	16	11
Deliverables	10	25	15
User Interface	25	39	14
Database	5	14	9
Lambda Functions	30	75	45
Batch Operations	20	13	-7
Kibana	30	0	-30
User Authentication	0	8	8

Testing	25	39	14
Total	**200**	**276**	**76**

How could the team have deviated less from the initial hour count breakdown?
The amount of hours assigned to each task at the beginning of the project was a tentative forecast. Agile methodology was used for the development of this project, thus, there were several tasks that changed overtime, as some requirements and functionalities were modified between sprints. In general, we spent more hours than those forecasted in most of the sections. The most remarkable one is the development of lambda functions, as they ended up comprising most of the functionalities of our simulator, and also, debugging these components can be a rather time-consuming task.
Another important difference between the forecasted and the actual time spent is that related to Kibana. As we were developing the UI of the simulator, we became aware that other technologies such as chart.js were more suitable regarding dynamic data visualization in a web environment. Thus, no time was spent on Kibana and most of that difference was spent on the lambda functions and the user interface.
Another reason for the extra hours devoted to each task was the “adaptation period” necessary when a member started working on an area for the first time. Throughout the whole process, the team members rotated over different tasks, so some additional hours were spent on learning the current status of that section. This may have been one of the main reasons for the deviation (a total of 76 hours).

References

[1] (2020). Chart.js Simple yet flexible JavaScript charting for designers & developers. Chart.js. Retrieved from https://www.chartjs.org/

[2] (2020). The Twelve-Factor App. Retrieved from:
https://12factor.net/

[3] (2020). Amazon Simple Queue Service. Retrieved from:
https://aws.amazon.com/sqs/

[4] (2020). MongoDB Atlas. mongoDB. Retrieved from:
https://www.mongodb.com/cloud/atlas

[5] (2020). Serverless Reference Architecture for creating a Web Application. Retrieved from:
https://github.com/aws-samples/lambda-refarch-webapp

[6] (2020). AWS Batch. Retrieved from: https://aws.amazon.com/batch/

[7] (2020). AWS Fargate. Retrieved from: https://aws.amazon.com/fargate/

[8] (2020). AWS Elastic Beanstalk. AWS. Retrieved from:
https://aws.amazon.com/elasticbeanstalk/?nc1=h_ls

[9] (2020). Amazon S3 pricing. AWS. Retrieved from:
https://aws.amazon.com/s3/pricing/?nc1=h_ls

[10] (2020). Amazon EC2 Pricing. AWS. Retrieved from:
https://aws.amazon.com/ec2/pricing/on-demand/

[11] (2020). Kibana. AWS. Retrieved from:
https://aws.amazon.com/elasticsearch-service/the-elk-stack/kibana/

[12] (2020). Amazon Worldwide SMS pricing
https://aws.amazon.com/it/sns/sms-pricing/