Final Report CCBDA

Andreas Haukeland
MIRI Student
andreas.haukeland.95@gmail.com

Haftamu Hailu Tefera
Erasmus Mundus Master Student-BDMA
haftish.birhan@gmail.com

Jorrit Palfner
MIRI Student
Universitat Politècnica de Catalunya
palfner.jorrit@est.fib.upc.edu

Mar Vidal Segura
MIRI Student
Universitat Politècnica de Catalunya
mar.vidal@est.fib.upc.edu

Contents

I. Introduction

As the last part of the course in CCBDA we have made a simulation tool for the COVID-19 outbreak. The tool simulates how infections spreads over time with user-interaction and can be used to determine the effectiveness of tools for slowing the spread of the virus. In particular the impact of a currently much discussed measure, contact tracing, can be evaluated using the tool. The tool features the functionality of quarantining users who have been nearby newly infected people. The tool is hosted on AWS and is scalable for a large number of simultaneous users. The simulator can be accessed on https://dtgtual5qkvjd.cloudfront.net/.

II. Final Result

The architecture of our final product follows the flow described in diagram 1. The user interacts with the simulator through a static website distributed using CloudFront and stored in S3. The first time the user accesses the website he can input the parameters he wants for a new simulation. The static website triggers a Lambda-function that stores these parameters as a new job in a SQS-queue. The job is stored with a simulation-id which is sent to the user via the SES and he is notified that the job is starting. An EC2 instance is listening to the SQS-queue and picks up the new jobs as they arrive. This instance writes the results of the simulation into a RDS MySQL database as it progresses. When the whole simulation is finished a notification is sent to the user via email to let them know that the simulation is finished. Through the URL in the final email or with the simulation id the user can then review the data in the static website.

A. Architecture

B. Design Choices

1) Static website: Our choice for the front-end of the application was a static website. The deployment is done with CloudFront, we have an S3-bucket for storing the website code and we have connected it with Lambda-functions to

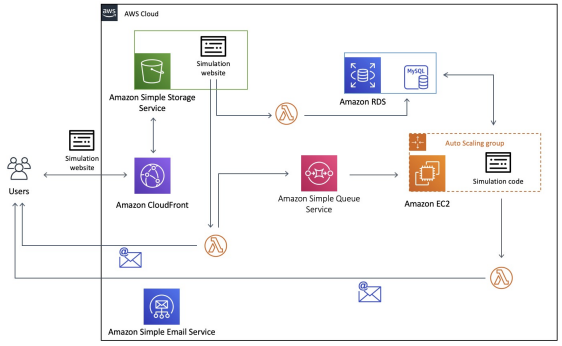

Fig. 1. Architecture

trigger the functionality of starting a simulation and retrieving information. A static website provides the benefit of a lightweight interface to our application, which distributed over a CDN is quickly available all over the world. It also reduces the costs of the cloud services as there is not a EC2 instance running the whole time hosting the website. This fits very well with the purpose of the website, which is to send data to the simulator and visualize the results, as this is not a very complex task and the website will probably not have a high amount of users on it at the same time. Given the nature of the application it is also likely that it will be idle a lot of the time and it therefore would be a waste to have a server listening for requests all the time.

2) SES: The Simple Email Service is used to inform the user of the status of his simulation and is triggered from Lambda functions. It is first used to inform the user that his simulation is ready to be run and give him the id of his simulation. After the simulation is finished the user is again notified so that he can check the results. The reason why we wanted to notify the user by mail is that for bigger jobs it would not make sense for the user to wait on the website until the job had finished. We therefore needed a way for him to know that he could come back to review the data. There are a number of ways to notify a user by email, but the SES-service is very easy to integrate in Lambda-functions and as it is part of the AWS-stack. The main reason for choosing the SES-service as our email-provider is therefore the simplicity of implementation.

3) SQS: For our application, we wanted to be able to have any given number of users at a time and we wanted them to be able to perform long-running jobs. We therefore needed a way to store the incoming jobs for the simulator-nodes to start processing when they were ready. To solve this we have implemented an SQS-queue which stores all the jobs received from the frontend. Simple queue service (SQS) is a simple queue that performs well, and is used to deliver messages reliably to for example EC2 instances or Lambda-functions. We have used the SQS-queue in the simplest way, which is to just query it for messages from our worker-nodes. The queue is therefore used to distribute work from the static website to an instance running the simulation.

4) EC2 autoscaling group: Since many users could be using our services at the same time we needed a way to scale our processing power with the number of jobs waiting to be processed. With an application deployed on the cloud there are numerous options to tap in to the seemingly infinite resources available. We have therefore chosen to distribute the workload from the queue into an autoscaling group of EC2-instances. As the first instance reaches it's limit of processing power a new instance is started that can poll a new job from the queue. This approach enables us to fully exploit the processing power of the EC2 instances, but also gives us an easy way to manage our services. The autoscaling group can for example be configured with a maximum number of units.

5) RDS: To persistently store agent simulation data the right database technology is required. In our first sprint meeting we were considering using DynamoDB NOSQL databases, but later we identified that we only need to store structured data of the simulation and decided to shift into a relational database approach.
Hence, in our research, we started by identifying the available relational database systems (DBMS). Finally, among the available ones, we decided to use MySQL as our storage engine. Why do we choose MySQL over other relational database management systems (RDBMS) could be an interesting question? MySQL is the most widely used open source RDBMS product which fully supports atomicity, consistency, isolation, and durability. Furthermore, the tool easily interacts with many business Intelligence tools such as tableau and Elastic stack to visualize simulation results to create better insights about COVID-19 spread. Moreover, the team is well aware of the technology and we found it easy to work.
After we chose MySQL, the next research was towards which Amazon MySQL suits best to our project. We research the aws MySQL implementation of Amazon Relational Database Service (RDS) MySQL instance and EC2 hosted-MySQL. And, we finally decided to use RDS MySQL instance for the following promising reasons:

- Amazon RDS makes it easy to perform automatic backups and manual snapshots of the DB instance whereas in EC2 hosted MySQL manually setup is required to do backup and snapshot through the help of EBS.
- DB instance scaling is another functionality that makes Amazon RDS MySQL a better choice. They have an Elastic Volumes that you can adjust performance, or change the volume type while the volume is in use
- Database setup, Database provisioning and software patching are automatically managed by AWS but in case of hosted MySQL manual installation and setup is required
- Automatic Multi-AZ deployments for 24/7 availability. while a hosted Amazon EC2 MySQL database you can

use partial replication and the configuration is still manual
- Easy to connect with EC2 instances to fetch and manipulate data
- One among the 12 factor methodology is treating the backing service as pluggable resources to minimize dependency with components (l.e EC2) and RDS Mysql support this one.
- In addition to what we mentioned above, RDS MySQL gives us another benefit to easily connect with a tableau which is a data visualization tool.

[] Overall, In Amazon RDS MySQL AWS takes full responsibility for the database and taking all the above RDS advantage we choose MySQL RDS. The entire process of configuration, management, maintenance, scalability, availability and security is automated by AWS. All these features helped us to focus on more important tasks instead of dealing with cumbersome and time consuming technical details.

- Day of simulation
- Currently Infected agents
- Total Isolated agents
- Total agents with symptoms
- Total infected agents
- Total dead agents

To easily interact with the database using GUI, we use MySQL workbench to retrieve data in a structured way. Furthermore, we can access the data from tableau Desktop and Tableau online.

III. Development through the project & Main problems encountered

The result described in the previous sections is our final result for the project. However, throughout the project we have tried to have an iterative approach and we have therefore tried to start with a simple solution and then gradually build more complex services. In this section, we will therefore describe our work throughout the project which includes some ideas we didn't have time to implement. Additionally we will describe problems we encountered along the way.

At the beginning of the project our idea was to create a dynamic web-server with Django, inspired by earlier assignments in the course. The deployment of this dynamic website would be with Elastic Beanstalk and EC2. As described in the section on Explanation of choice this did not fit very well with the purpose of the website as it would mean running an instance all the time. After some consideration we therefore decided to change it to a static website. Given the change in design, we therefore got some extra work since we had to implement the frontend again. Luckily, this was fairly early in the project and the developing time of the static website was reduced as we already had our HTML-templates designed.

The interaction of the static website with the other parts of the project is implemented with two Lambda functions. The first Lambda function sends the message to the SQS with the information needed in the simulation and it also sends an email to the user though SES. The second Lambda function gets the simulation-id and gets the simulation results from the database hosted in RDS. The implementation of this second Lambda function was trickier than the first one as it has to have a special configuration to interact with RDS and also it has code dependences that they have to be added inside Lambda.

After we had decided to receive jobs from a SQS queue we wanted to create the simulator in an iterative approach to have a running pipeline working as soon as possible. The first version of the simulator was therefore a single EC2 instance that pulled one job at a time from the SQS queue, initialized the simulation with the user-defined parameters and sent the results to the database.

However, we identified several drawbacks of this first version which we wanted to improve upon. The first issue identified was that for jobs with very large populations it would be very slow because we had no parallelization in the internal code. Furthermore, we stored all the agent data locally therefore it is possible to run out of memory. The second drawback was that it only processed one job at a time. For this reason our application would not perform well in case of high user-activity. The last drawback was that the EC2 instance was kept alive all the time listening to new jobs and therefore had to run even if there were no new jobs upcoming.

For our next iteration we took on scaling by removing the data from in-memory and rather fetch it from the database when needed. This would allow us to simulate on any population size and also make it easier to distribute the work because every worker could fetch the relevant data from the database. The first two drawbacks were somewhat interconnected and we had two different ideas for solving them. The first idea was the architecture we have today with an SQS queue and an autoscaling-group with EC2 instances that perform the work there. The job of coordinating the new job is very light and therefore only one instance is necessary. To parallelize the code we have separated the heavier calculations out of the coordinating simulator into stand-alone functions residing in an autoscaling-group of other EC2 instances. The coordinating EC2 unit therefore puts all the heavy work into a new SQS-queue which the autoscaling-group reads from. The autoscaling-group reads and write the needed data from a RDS MySQL database. The other idea was to do the heavy work in Lambda-functions called by the coordinating instance. Both approaches were quite similar in complexity and achieved the same things. However, both approaches encountered the same problem, accessing the database instead of the using the local memory negates any gains we might have achieved from splitting the work. Another problem we encountered when trying the approach with 10 worker instances and with one coordinator was that the coordinator became a bottleneck as well. The issue is that the coordinator in this shape distributes too many tasks creating too many confirmation messages. The confirmation messages are in particular for one request to a SQS queue can at most return 10 messages. Concluding with

the distributed approach a simulation of a 1000 agents takes minutes instead of seconds. Therefore, we kept the prior non-distributed version of the simulator deployed in combination with the website.
Another problem encountered through the project was that even when we had things working locally we found it very hard to get it to work with deployment with Elastic Beanstalk. To solve this we rather created an image of an EC2 instance which updates itself on startup by pulling from git, installing any new dependencies from a requirements.txt file in the repo and then running the latest version of the script. This worked pretty well, but was not ideal since it made it hard to work on separate branches from the master-file. Also looking at log-files was challenging in the cloud since we didn't log everything that happened on the EC2 instance and it was hard to receive the logs when SSH-ing into the instance.
In order to get smoother visualization of the simulation results, we wanted to include interactive graphs. To do this we needed an appropriate data visualization tool and we found Tableau to be a perfect match as it easily integrates with MySQL database and we had some experience with it from our tutorial.

Tableau was very easy to use and to create the line graph below we did the following:

1) Drag and drop "measures and dimensions" into "columns and rows" section of Tableau Workspace
2) Select the type of visualization, tableaus selects the default visualization based on the data we provide
3) Apply custom formatting such as changing colors, and filters.

From the line graph (figure 2) we can see a correlation among the number of deaths, isolated, and infected using line graphs with time.

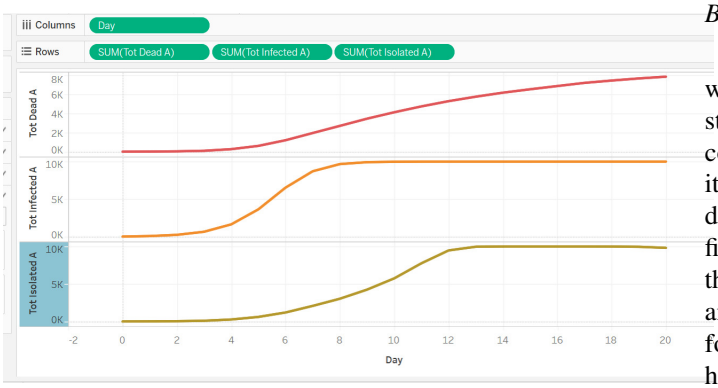

Fig. 2. Tableau Demo

Unfortunately, we did not have time to deploy it in our final version with it being connected to the correct simulations. This will be part of future work, because it is a great tool for visualizing real time generated data.

A. Difference to project draft

When reviewing the first draft, the biggest difference to our final solution is in the complexity of the actual simulation. After finishing the first iteration of the simulator and getting feedback in the 1st sprint meeting we realized that it was very time-consuming to create an advanced simulator and that it was not very beneficial for the main purpose of the project which was to work with cloud services. The final simulator therefore has less user-defined parameters than we initially thought and things like different user-types, customizable population density and varied travel patterns were therefore emitted.
On design choices we mostly stuck to the initial architecture, but with two important distinctions. Initially we wanted to create a dynamic website in Django, but given that we simply wanted to show data to the user we decided that a static website was sufficient with the added benefit of not needing to constantly run a web-server.
The other difference in technology was using a RDS-database instead of DynamoDB. We found that the benefits of the two were very similar since we didn't need real-time data and we had very structured data that could easily be stored in RDS.

IV. The 12 Factor-Methodology

A. Code base

In this project we have used Git for our code base. We have used a private repository to have all the project code and documentation to have a version control of them. Using Git came with the benefit of providing us with the possibility to create branches to work simultaneously in the same repository preventing conflict with others' work.

B. Dependencies

Three parts of the project have dependencies: the static website, one of the Lambda functions and the simulator. The static website has included its dependencies inside the HTML code. To deal with Lambda function with dependencies, first it was created locally. A virtual environment was created to download all the libraries and packages needed. After, a zip file was created with the packages of the dependencies and the lambda function. This zip file was uploaded in the cloud and it contains the dependencies needed. The dependencies for the simulator can be found in a requirement.txt file which has been generated using pip ¡ freeze. This way dependencies are easily installed when the simulator is deployed to a new EC2 instance. Dependencies include the boto library to interact with amazon services such as the SQS queue and pymysql to interact with the database.

C. Configuration

The configuration files also affect the same parts as before. The static website contains the Lambda urls to execute the

functionality. These urls are on top of the JS file (submitToAPI.js), the file that interacts with the Lambda functions. A configuration file was created for each Lambda function. These configuration files contain all the endpoints to interact with other cloud services and they also contain necessary information such as database username and password. The simulator is deployed from an image stored in AWS. In this image the environment variables are set when starting up the instance so that it accesses other AWS services. This allowed us to remove the variables from the code, but made it hard to update the variables as you needed to create a new image. The variables have however been pretty similar through the last parts of the project.

D. Backing services

Different backing services are used in this project such as EC2 instances, SQS, RDS with MySQL, SES for emailing and CloudFront as CDN. The code is not fully separated from the characteristics of these services. Some code like the lambda functions use the specific code to interact with them, such as specific MySQL queries, so they can not be fully replaced for other providers that offer the same service. Even though they can be replaced for other instances of the same services as the configuration files are separated from the main code. Other services like CloudFront can be easily replaced for another CDN as the base code is stored in S3.

E. Build, Release, Run

We have not followed a strict separation between build, release and run. This is mainly because we have had separated instances with defined end-points and therefore did not feel we would gain a lot from creating heavy startup-scripts for deploying the code. As mentioned before, this somewhat reduced our flexibility because if did not want our EC2 instance to use the newest GitHub code we needed to SSH in to change it manually.

F. Processes

The application consists out of the static website which is running in as many independent stateless processes as necessary. The lambda functions used are two more stateless processes which only rely on the input of the call. The SQS can be viewed as a stateful buffer in case multiple simulations are requested at once. The simulator itself uses the EC2 instances memory to preserve state within a simulation. However, every new simulation is independent. The results of every simulation are saved in a database. The result data can be requested from and viewed on the static website.

G. Port binding

CloudFront is used to configure port binding and listens to the requests and provides them to the static website stored in S3 bucket. This bucket has restricted access, it can only be accessed via CloudFront. Other services that interact with each other such as Lambda function and RDS, they are in the same VPC and security group and the database can not be accessed by addresses out of VPC or addresses not configured in the security group.

H. Concurrency (Scale out via the process model)

The static website is distributed using CloudFront. It is a content delivery network massively scaled and globally distributed so it can deal with a lot of requests. The static website triggers Lambda function to execute the project functionalities. For each submit action, a different Lambda function is executed so this part of the project scales out automatically. An SQS queue supplies a simulator instance with the parameters of the tasks. Therefore, several simulator instances could run in parallel conducting independent simulations. However, in our final deployment new simulator instances need to be launched manually. The simulator deployed does not scale. However, we explored two possible approaches, using lambda functions or workers, to scale the simulator which are described in other sections in detail. The results of the simulations are stored in individual tables which can be accessed concurrently.

I. Disposability (Maximize robustness with fast startup and graceful shutdown)

For the client side of the project, the static website distributed with CloudFront is the whole time available. It executes Lambda functions which are always available and they are started and stopped automatically. For the simulator part we did not have time to implement graceful shutdown. A process that is killed will not return it to the queue and the job can therefore potentially be lost. To implement this is part of our future work.

J. Develop and production parity

Initially we started using two different branches: main for production and develop. The initial idea was creating different branches from develop branch to implement different functionalities of the project.
Finally, this way to work was not fully accomplished. The main branch was used for the development of the simulator code. The static website branch to develop the static website and the develop branch to have the current website versions deployed in the cloud. This way of working has been a little bit tricky to finally organize the final (production) version of the project. This way of working has not been a problem for losing correct work as the correct versions of each part of the code were deployed in the cloud.
Using the develop-stage-production versions it is very important when different people are implementing the project and also to keep track of the final versions that you have deployed in the cloud. This way of working would have help

us to work more efficiently and organized and ergo having the repository fully organized and having in the main branch only the working code.

K. Logs

We have tried to have useful logs in the whole project, to easily face the problems during the executions. We have enabled a S3 bucket to store the automatic CloudFront logs. Also the monitoring dashboard that it offers it has been useful to solve problems.
For the Lambda functions, different Log Groups have been created to keep track of the different logs of each function. In difference with CloudFront, these logs were customized to easily visualize which part of the function does not work. For the simulator we have used the logging-module of python to get clearer overview of what is happening. We did not implement any way to collect logs of the EC2 instance without SSH-ing into the instance which made debugging harder to do in the cloud.

L. Admin processes

In this project we have not used admin processes and we have not developed them.

V. Description of GitHub repository

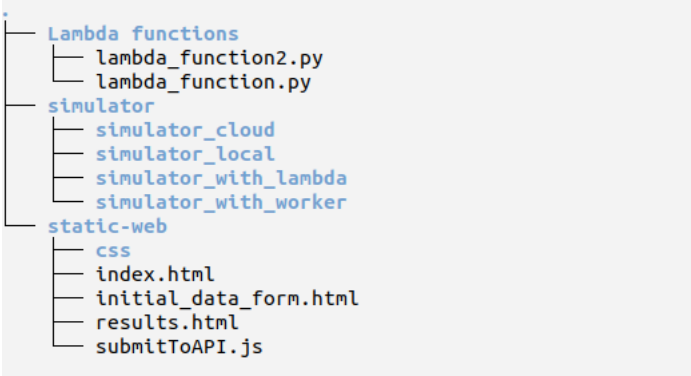

Fig. 3. Github Repo Tree

The project's code repository consists of three main folders: Lambda functions, simulator and static-web.
The"$staticweb$" is the folder for our static website. The folder consists of a generic CSS-file, three HTML-files: "index.html" which is our welcome page, "$results.html$" for showing the users results and "initial_data_form.html" for collecting input data from the user, and finally a JS-file. In the JS-file are defined the jQuery functions that trigger the Lambda functions when the forms are submitted. The jQuery function that is executed when the initial data form is submitted, it generates a simulation id, using uuid4 (pseudorandom 128 bit number), to have the simulation identified and prepares the data of the form to be sent to the Lambda function. The second jQuery function is executed when the user submits the simulation id to get the results. This function sends the simulation id to the Lambda function. If the Lambda result is successful, it hides the form and displays the results in a table.
The application architecture features two lambda functions which's code can be found in the Lambda Functions folder. The file lambdafunction specifies a lambda function sending a confirmation email that the simulation has started. Furthermore, the function sends a message with the simulation parameters in the queue which supplies the simulating instance with tasks.The second lambda function defined in lambdafunction2 fetches the result data of a specific simulation using the simulation id from the database to be displayed on the static website.
The simulator folder contains four versions of the simulator. The versions "simulatorcloud" and "simulatorlocal" are very similar, but the cloud version takes input from the SQS-queue while the local version takes input through the command line or specified in the code. We will start by describing these two versions first.
The input functionality is contained in the file "main" which given input parameters start the simulation. These input parameters are whether contact tracing is enabled, the number of agents, how many agents are initially infected, the number of agents in the same office, the number of agents residing in a home, the mortality rate in percent, the number an agent will be sick if infected, the number of days until an infected agent shows symptoms, the risk of infection at home, the risk of infection at work and whether verbose is enabled. Given these inputs the "initializer" is called creating the locations and agents for our simulation as objects. Both agents and locations are modelled as classes and can be found in their respective files. Every agent has an unique id generated with uuid4. This generates a pseudorandom 128 bit number which is stored in hexadecimal. Agent objects are stored in an array which is shuffled before assignment of agents to home and offices to introduce randomness in the distribution. Every home and office is created as an object with specified capacity, infection risk and agents which are also stored in an array.
A saver class is initialized with verbose and a simulation id. In this instance a table is created in a MySQL database which's parameters are specified in the folder Database. The table name includes the simulation id to enable display of results of a specific simulation. The simulator is also modeled as a class, taking whether contact tracing is enabled, the location array, agent array, saver object and simulation id as input. A function called step simulates a day in one step, by going over all locations to simulate whether a person, which is assumed to interact with all people in its home and office gets infected. Subsequently every agent object is updated and if contact tracing is enabled agents are isolated if an agent in a home or office shows symptoms. At

the end of each step the current status is saved to the database.

simulator_with_lambda The simulator with lambda functions is based on the two other versions. It is extended by parallelizing the simulator using lambda functions. There are 3 main steps which can be parallelized: the initialization, the simulation of infection and updating the status of an agent. All of these are in the previous code implemented as for-loops going over a lot of elements and we believed this could be improved by sharing this computation.
In our version of this, we did not have time to implement the change for all 3 places, but we tried for simulation of infection. We split the work in 2: one lambda functions simulated infections for houses and one for offices. However, when implementing this our simulator slowed down considerably, which we believe is due to use of RDS for gathering data instead of local memory.

simulator_with_workers The folder simulator with workers, contains the version of our simulator made up of a coordinating instance and the workers instance it coordinates. The code can be found in the respective folders Coordinator and Worker. The coordinating instance follows the architecture of the simulator deployed with the difference that it is the coordinating instance distributing the work to create and update agents to other ec2 instances. The file agent.py, as well as the locations office.py and home.py specify the creation of tables in a mysql database. For interaction with the database we used sqlalchemy. The file sqs_handler contains the logic to send, receive and delete messages from an sqs queue. Initializer.py is called by main to firstly create the tables for agents and locations. Secondly, batches of work to create 100 agents each are created and send to a queue. Subsequently, the initialization process waits for confirmation messages that the work is done. Thirdly, messages containing instructions to assign agents home and offices are created, distributed and waited in similar fashion. After initialization the simulator is started. The simulation of a day is following the same pattern as initialization the simulator distributes the simulation of infections for all offices and home in batches of 10 including waiting for confirmation messages after each distribution of messages. After completion a message with the task of contact tracing distributed and waited for.
Every message is a dictionary encoded in .json. For task messages a message specifies the function to be executed and other parameters such as which agents in its message body. A confirmation message merely contains an attribute with the value "1". The folder worker contains a main file, which receives the messages and calls respective functions, an initializer file in which functions for creation of agents, as well as assignment to homes and offices are specified. The file simulator functions holds the implemented functions the same as in the local version but reading and writing, from and to the database.

VI. Future Work

In this section we present ideas we have to further improve the simulator but we did not have time to implement.

A. Scaling the simulator by distributing work

Finding good size for batches of work is critical to the success of a distributed approach. However, due to a lack of time we were not able to experiment with these parameters. Furthermore, a database with short access times seems to be just as critical. To solve the bottleneck of the syncing the coordinator and the workers an idea could be to sync the workers instead while the coordinator continuously fills the queue with tasks. Following this approach the work of the coordinator could also be distributed over multiple instances. A possible architecture could be an initial coordinator which receives the requirements of a simulation job and decides according to the number of agents to be simulated how many instances are necessary. Additionally, the initial coordinator provides all instances with the same parameter it generated at random to randomize the assignment of agents to offices and households. Every of these instances could in turn launch a number of worker instances. After this process every coordinating instance would perform the same functionality filling a queue with messages, possibly even not all to the same queue to avoid queue request conflicts. To conserve time continuity in the simulation workers need to wait for each other before performing tasks of a next step. The workers could switch between different modes belonging to different steps (office, home, contact tracing). Every task message would have an attribute marking the step it belongs to. As soon as a worker does not find any new messages belonging to the current step it is in it sends out a confirmation message to all other workers, in a confirmation queue, and waits for confirmation messages of every other worker. Because the number of workers created is deterministic from the number of total agents every worker knows how many peers it has therefore how many messages to expect. In case of using an auto-scaling group one would have to think of a way of registering a worker with all other workers when a new worker is booted up and unregister if a worker is shutdown.

B. Improving results with Tableau

In the simulation, we were thinking of automating the data ingestion pipeline in order to communicate updated simulation results to agents using an interacting graph in addition to regular emails in the form of URLs or endpoints, but due to time constraints we did not fully implement this feature. In the implemented approach, the data movement from EC2 to RDS is automated but sending data from RDS to Tableau is batch based because we fetch data into tableau after the simulation data is stored and we need to connect again and again to the database to see updates in the table to reflect those changes in the visualization. The automated data ingestion feature will enhance the functionality of the tool

when we deploy into the production environment.

Therefore, for future works, we will automate the data pipeline from EC2 to RDS, and then to tableau and dashboards will get updated automatically. And, agents will access simulated data using endpoints or http kind requests in addition to email notifications. We are also thinking of adding geolocation data to have a complete picture about COVID-19 spread using Interacting maps. Hence, users can get instant updates about the spread of using dashboards in order which will help them to take appropriate measures.

C. More features for the static website

We have three different improvements for the static website. The first one is about accessing the user results. The idea is to generate a url with the simulation id as a parameter, for example: $https : //dtgtual5qkvjd.cloudfront.net/results.html?simulationid = 473907d4 - 0eb2 - 4dad - 93f1 - cda2baa5b545$. The GET request would be processed by jquery and it would trigger a Lambda function that automatically includes the simulation id to the MySQL query. If the url does not contain the $simulationid$, the current form would be shown. This feature would make the user experience more efficient and easier. Related to this last improvement, it would also be a good improvement to add AWS Cognito to treat each user experience as a guest session and therefore keep track automatically of the data of the session and make the website more user friendly.
The errors shown to the user can be also improved. Currently, the website treats all the errors that came from the Lambda functions interaction in the same way and showing the same message. It would be a good improvement to show to the user the exact error that is being raised, for example, server error, database error, a user error introducing the input parameters or the $simulationid$.

D. Optimize Lambda functions

The current Lambda functions can also be improved and optimized. They could be generalized and execute only one functionality. The best way would be:

- Lambda function that only sends messages to SQS.
- Lambda function to send emails, the whole email information is provided.
- Lambda function to fetch data from the database. This is already implemented in this way.

VII. Listing of Time invested

TABLE I
TIME INVESTED ANDREAS HAUKELAND

# Sprint	Task	Hours
1	Coding simulator logic	5h
1	Architecture research for project	4h
2	Deploying simulator on cloud Coding database, connecting EC2 and simulator	5h
3	Implementing lambda-functions, trying to parallelize EC2 code	4h
4	Implementing final version, cleaning code	6h
Draft		2h
Report		6h
Meetings	Group meeting Every Thursday (1h 30min approx)	6h
Meetings	Sprint meeting Every Friday (30min)	2h

TABLE II
TIME INVESTED HAFTAMU HAILU TEFERA

# Sprint	Task	Hours
1	Research web application architecture	4h
2	Database setup, database and EC2 Research	6h
3	Coding Database	6h
4	Data visualization with tableau	4h
Draft		2h
Report		6h
Meetings	Group meeting Every Thursday (1h 30min approx)	6h
Meetings	Sprint meeting Every Friday (30min)	2h

TABLE III
TIME INVESTED JORRIT PALFNER

# Sprint	Task	Hours
1	Coding Simulator	6h
2	Research Parallelization Simulator	4h
3	Implement Queue functionality and start on reboot	8h
4	Implement parallelization with workers	8h
Draft		2h
Report		6h
Meetings	Group meeting Every Thursday (1h 30min approx)	6h
Meetings	Sprint meeting Every Friday (30min)	2h

Acknowledgment

We would like to thank our Professor Angel Toribo for patiently guiding us through the project with helpful advice in the weekly meetings.

TABLE IV
TIME INVESTED MAR VIDAL SEGURA

# Sprint	Task	Hours
1	Set up Elastic Beanstalk to launch the simulator.	2h
2	Develop the dynamic website using Python + Django. Implement the templates using HTML and Bootstrap. Deploy it in EC2. Arrange the simulator for being deployed together with the web server in EC2.	6h
3	Develop a static website with HTML, jQuery and Bootstrap. Deploy it on S3. Develop the lambda functions to do the functionalities. Send confirmation emails with SES. Send to SQS the initial data provided by the user.	6h
4	Develop the results page with HTML, jQuery and using Bootstrap. Implement the lambda function to fetch data from RDS. Deploy the static website using CloudFront and store it on S3. Improve the implementation of the messages of SQS.	8h
Draft		2h
Report		6h
Meetings	Group meeting Every Thursday (1h 30min approx)	6h
Meetings	Sprint meeting Every Friday (30min)	2h