Cloud Computing

Project Final Report

Marc Garnica Caparrós
Quang Duy Tran
Syeda Noor Zehra Naqvi
Todi Thanasi

1st June, 2018

Submitted to: Angel Toribio Gonzalez

Table of Contents

Introduction

Information retrieval and open data platforms are key points for developing the next generation of smart cities. Open data services and frameworks on top of a city data and cloud infrastructure can speed up the innovation and growth of the city bringing and implicit improvement of the citizens life.

One of the sectors where open data and innovation can provoke significant improvements is on the Mobility across the city. In the city of Barcelona, Transports Metropolitans de Barcelona (TMB) is opening to the community a repository of data sets and information defining and describing at real time the details of the public transport in the city. Our project aims to use this open portal as the starting point of a service to provide a useful service to the citizens.

Public transport is one of the most used ways of moving around the city of Barcelona. Sometimes a shower longer than expected, lazy walking or other factors can make us lose the desired train or bus. In other occasions, the time waiting for the bus in the stop can make us lose a lot of time during the day. Our project builds a simple but robust service providing the real-time arrival times of the buses in the city. In addition to facilitating the citizens, this project also includes a dashboard to visualize data analysing user requests and provide feedback to Transports Metropolitans de Barcelona.

Objective

The main goal of this project is to gather information regarding how many search requests were made for a bus or a line for being able to measure the usage of public transport in barcelona and provide feedback to Transports Metropolitans de Barcelona as follows:

- Frequency of requests per bus stop and line.
- Visualize this data on charts and map

To achieve our goal, these main tactiques have been described:

- Deploy a trustful and usable service for citizens to check the arrival times of the desired buses and stops.
- Implement an dashboard visualizing the frequency of line and bus stop requests on charts and map.

Another goal of the project is to learn about automation capabilities within cloud. The TMB Helper app has been implemented to achieve all the described objectius in an integrated platform where the data gathering and data analysis are offered.

Functionalities and Scope

As defined in the project objective section the implementation of the TMB helper is focused on two main parts. First of all a service for citizens where they are able to see at real time the arrival times in any Bus station. This platform works as a data gathering system which is used in the second part of the project. The second part consist of a dashboard with a chart showing most frequently requested bus stops and another chart showing most frequently requested line in order to realize the number of people using a particular route. The visualization of frequently requested stops are also provided by a heat map.

<u>TMB Helper: Citizen service</u>

The user entry point in the system is a website. Users are usually facing memory problems in their mobile phones so they are not able to download all the application they require. It is for that reason that the service interface is provided as a web page where users can easily access through any web browser. The functionalities of the service is following:

- User friendly and device independent interface.
- Search by stop name.
- Search by line.
- Get arrival times by stop.
- Receive other stop recommendation depending on the arrival times and location.

TMB Helper: Dashboard/Analysis service

With the data gathered through the citizen service, the dashboard provides visualization of this data through charts and a heatmap. The features of this dashboard are the following:

- See top 6 most frequent requested stops as a bar chart and heat map
- See top 6 most frequently requested lines as a bar chart
- Date filter to filter the plots by different date

Architecture

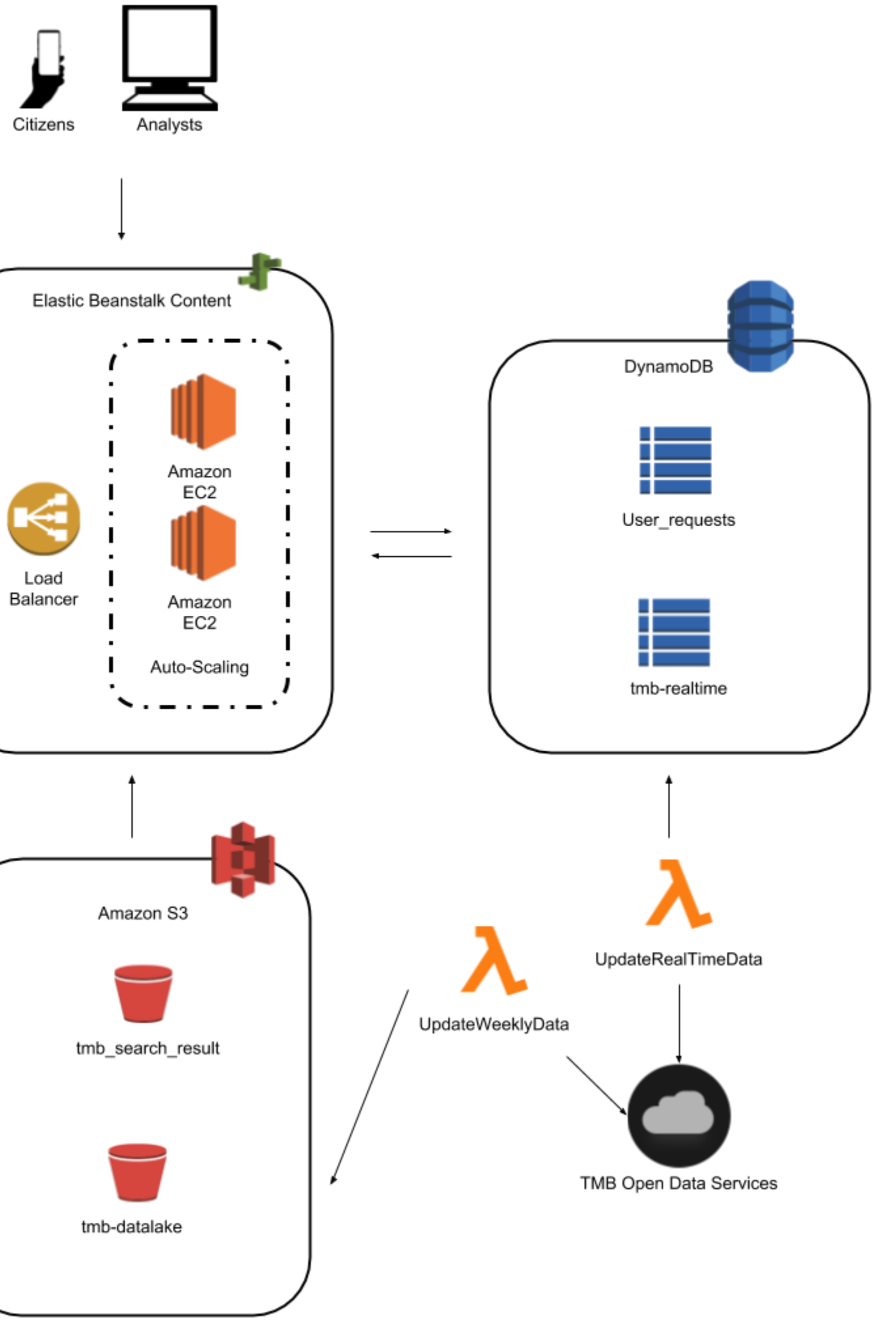
The main components of the deployment to achieve the functionalities of the project are shown in the next diagram. The interface of the system is a web browser, either on mobile device or computers. Most usually citizens will use their mobile phones and the analysis dashboard is designed for computer screens. (Both of these components are part of a django web application). Several changes have been performed from the initial draft architecture description due to first, adaptation and flexibility for the maturity of the application and responding to different needs.

For the project scope, it was decided to postpone the usage of **Route 53.** After implementing and deploying the application and infrastructure, it was clear that there is no need for a robust static content management through **AWS Cloudfront** from a **AWS S3 bucket**.

The main functionalities are provided by an **Elastic Beanstalk** environment with Load Balancer and minimum of two **EC2 instances** within an auto-scaling environment. The implemented django-app is being deployed on this EBS environment.

The Elastic Beanstalk connect with **AWS DynamoDB** environment to get all the information needed to serve the user request. Inside the DynamoDB, two main tables are maintained: Real time data table and user request table. Weekly data is maintained in an S3 bucket.

The weekly S3 bucket tmb-datalake updated once per day by a lambda function connecting to the TMB Open Data service. This portal is also providing a real time interface to check the arrival times by stop and bus code. Although it is a powerful interface, it has some limitation in terms of requests and concurrency so the following approach has been initially decided: A lambda function is incrementally updating the DynamoDB real time data every minute. With this approach, the availability of the system is ensured no matter the deficits or issues in the Open Data portal and the request rate limits is fulfilled. Another S3 bucket is maintained to cache the queries made on dashboard and citizen search page to reduce to query load from database.



Methodology

We followed an Agile SCRUM methodology with sprints of one week in a total of 5 weeks. The tool selected for the project task management has been Trello (trello.com). Here is the link to out trello board: https://trello.com/b/VayZMpFJ/cc-project.
5 lanes were maintained in trello which were Backlog, To-Do,Doing, QC, Done. We maintained a backlog of all the stories and tasks to be done throughout the project life cycle. Different tags were assigned to the tasks in different categories. (new feature, research, documentation, administration work etc). All the tasks to be done during a sprint were moved from backlog to To-Do lane. Doing lane was maintained in order to avoid the problem of 2 team members ending up working on same task. Whenever a task was finished it is checked by another team

member in order to check if the developer missed some checks and then moved to the Done lane.

We had meetings every week to discuss the progress, explain each other on what we did and also discuss the occurring problems and any potential changes needed. Each team member was responsible for one main task and all of the subtasks related to the main task, however we shared the knowledge and help each other as a team in accomplishing all the tasks. We divided the tasks into 4 parts: Django webApp User services, Django webApp Dashboard analytics services, lambda functions for getting data from TMB (real time and weekly), automation, integration, monitoring and logging implementation.

Implementation Details

This section will discuss the implementation details of all the functionalities implemented for this project included all the problems faced while the development and what choices were made.

Serverless backend with AWS Lambda functions

As presented in the Architecture description, the implemented application cannot rely only on the user rate provided by TMB and it requires a layer of isolation in order to be robust and independent enough. In order to achieve those characteristics it was decided to perform a cache level with the interested request of the TMB open data endpoint. More specifically, two serverless lambda functions are periodically connecting to the TMB services and gathering the desired information.

TMB SOAP-WSDL endpoint

Simple Object Access Protocol is a standard-based Web services access protocol used for several years in the industry. REST was the newcomer in the block, indeed, REST seeked to fix the problems SOAP was experiencing, specially some additional complexities SOAP carries.

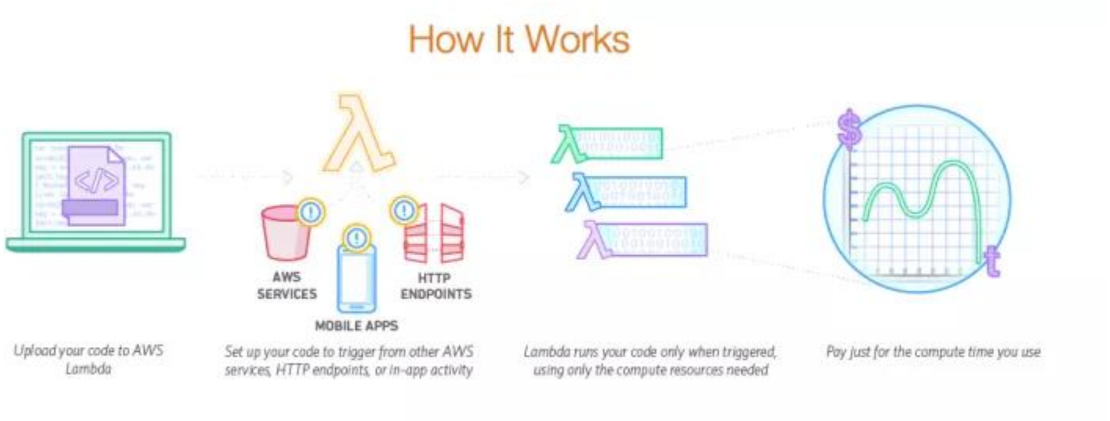
SOAP services rely exclusively on XML to provide all the resources and messaging services. One of the best functionalities of SOAP is the usage of WSDL files, key file for services discovery and retrieval. The SOAP WSDL files include all the needed definitions, methods and parameters to understand how the web services works. In other words, the process of discovering new services and executing its methods can be done automatically and programmatically.

For external reasons, TMB services is implemented with SOAP-WSDL technology. In order to consume this services from our backend Python Zeep package has been selected as the main tool to explore and consume the service. Zeep is basically inspecting the WSDl document provided by the service and generating the corresponding code to use all the methods defined on it. The parsing of the xml responses is performed with lxml library.

More technical details on the installation of Zeep and all the other packages needs can be found on the detailed Github repository.

AWS Lambda Functions

Amazon web services Lambda functions provides you with a user-friendly way of run code in the cloud without provisioning or managing servers.

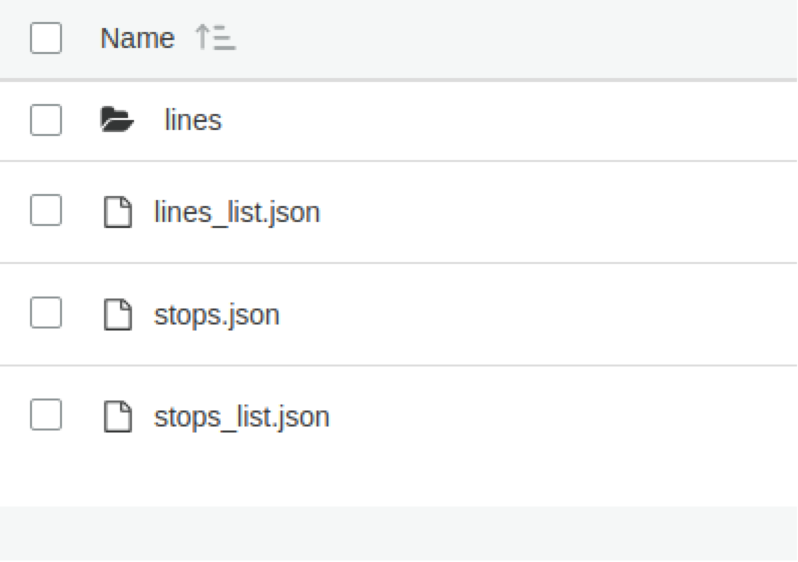

How It Works
AWS SERVICES
MOBILE APPS
HTTP ENDPOINTS
Upload your code to AWS Lambda
Set up your code to trigger from other AWS services, HTTP endpoints, or in-app activity
Lambda runs your code only when triggered, using only the compute resources needed
Pay just for the compute time you use

Weekly data ingestion

This is function is in charge to get the the more stable information coming from the TMB services such as bus lines, bus programs, stops and stops by line. This information does not require to be updated frequently so the function has been configured to run on a weekly basis.

The function is connecting to the TMB services with Zeep and then getting all the information from the method '**getBusLinesAndStops'.** The response contains each stop information by line.

Once the response is parsed, the results are stored in an s3 data lake with the following content:



□ Name

□ lines

□ lines_list.json

□ stops.json

□ stops_list.json

- Lines folder contains a json file per each bus line in Barcelona. Each line file contains metadata of the line (code, name) and each stop code and name of the two directions of the line.

- Stops.json file is a detailed file contained all the stops metadata present in the bus services of TMB. Each stop is defined with its code, name, their coordinates with UTM and WGS formats, and finally all the lines going through the stop.
- Finally lines_list.json and stops_list.json are just useful files in order to speed up the queries and reduce the network usage within the fronted, containing a summary of lines/stops names and codes. These files are used for searching.

Real time data ingestion

As mentioned in the objectives of this project. TMB Helper aims to offer real time information about the arrival times of each stop and line. To perform that this function is connecting with the TMB services via the function ‘**getArrivalTimes’**. To achieve the most accurate real time information this function is updating their response every minute.

The response is parsed and then stored in a dynamodb table for further usage storing the arrival time per each line and stop of the TMB services in Barcelona.

The initial proposal was to create update a table item per each line and stop with its arrival time. This options is bringing the preferred granularity and reducing the network transfer between the frontend and the dynamodb table but it actually created a high number of items in the dynamodb table making bulk update take too many time and destroying the real time objectives.

After this reconsiderations the final granularity of the dynamodb table was to identify each item as a line array of arrival times. For each line and direction of TMB, dynamodb is storing each stop arrival time. We get use of the flexibility of dynamodb table schema with the following configuration:

- The hash key and also primary key of each item is the line code and direction.
- No sort key has been implemented due to the reduced number of items and conceptually not needed sorting in the partitioners dynamodb will potentially create.
- Each item has common schema keys: line code, direction id.
- Each item has then different keys depending on the the stop code of the referenced line. This is a key point to run the desired queries with lookup constant time. At the end each item schema looks like the following:

```
{
        “lineCode”:
        “directionId”:
        “Stop1id”:
        “...”:
        “StopNid”:
}
```

The update has been implemented efficiently making use of the **BatchWrite** functionality of AWS DynamoDB. With BatchWriteItem, you can efficiently write or delete large amounts of data, such as from Amazon Elastic MapReduce (EMR), or copy data from another database into DynamoDB. In order to improve performance with these large-scale operations, BatchWriteItem does not behave in the same way as individual PutItem and DeleteItem calls would. For example, you cannot specify conditions on individual put and delete requests, and BatchWriteItem does not return deleted items in the response.

TMB Helper frontend: Django Webapp

Django web application is the user entry point in the application. The home page of the application is search page for users to search for different stops and lines for their bus timings. There is a link to move to Dashboard from search page where the visualization of the most frequently requested stops and lines is available in form of charts and map.

User search for lines and buses

The search page consists of a search box, showing two options: search for line and for stop. Search results will navigate users to the stop or line they were searching for. These results are taken from a single json file on a S3 bucket which are updated weekly by the mentioned lambda function above. Depends on the search type, it can get data from lines_list.json or stops_list.json.

they are cached in another bucket. For each type of search (stop, line), results from each keyword will be stored in a json file. Example: results of finding stops containing the keyword “zona” will be stored in stops/zona.json.

The line page shows all the stop of it in both direction. For each stop in this list, other lines going through it are showed. In each stop page, we can get the arrival time of all upcoming buses to that stop. They are called from the DynamoDB table which getting data through TMB API. A refresh button is provided for users to update the newest time. Everytime an user visit a page of line or stop, a row is inserted to DynamoDB user request table for analytics.

Dashboard for Analytics

Dashboard is a web page which is part of our django application. It is a separate view in the django app. Dashboard consists of 2 charts showing the most frequently requested stops and most frequently requested lines respectively. The map shows the most frequently requested stops on the map in form of a heat map. The Dashboard also allows filtering of data by date range to re plot the graphs according to the data within the mentioned dates.

Technologies Used:

To build the dashboard the technologies used are python (django), S3 bucket, dynamo db, chart.js, leaflet.js.

Chart.js

Chat. js is a javascript charting library which is used in this product to make charts. We choose chart.js because it has vast capabilities at it is very easy to use and integrate with the project and it does not have any particular dependency issues. Chart.js has also very nice tutorials and big community. An alternative could be vincent, but we chose chart.js because vincent has python dependencies and it uses d3 charting library for javascript which is more complex if one has some particular requirements.

Leaflet.js

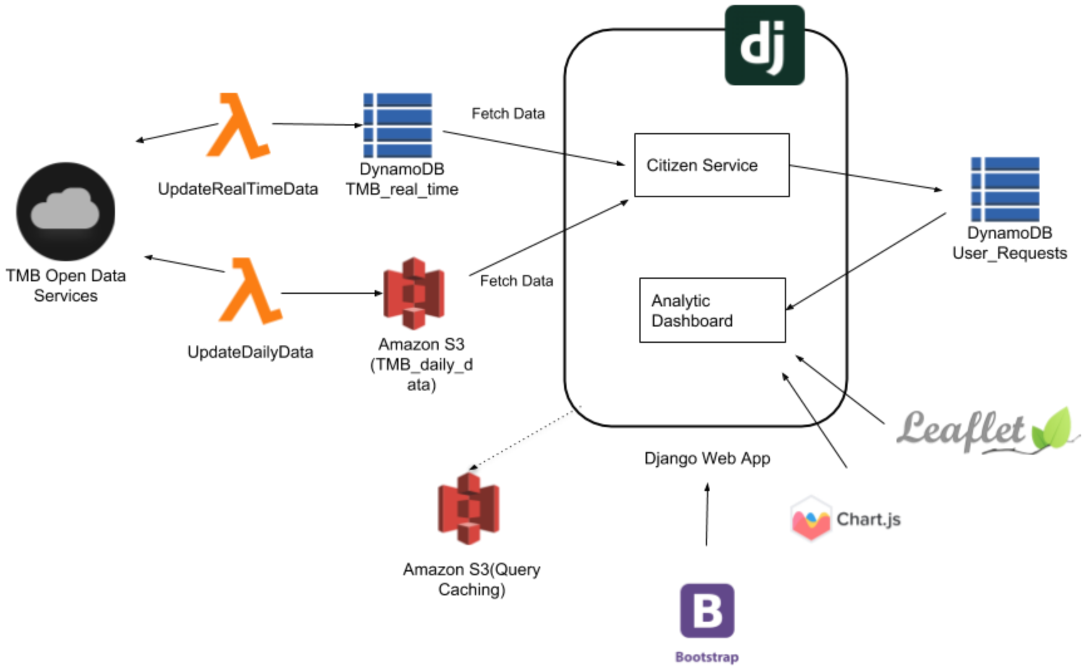
Leaflet.js is a javascript library for building interactive maps. We used this library to plot the map on the dashboard.We used the leaflet heat map extension to plot heat maps on top of leaflet maps to show the concentration of more frequently requested stops. The other option we explored was Google maps, it was a better option with easier usage, better APIs and more community, but from June 2016 onwards, they don't provide any free APIs, so we decided to stick to the free option. The problem faced while working with leaflet was that it has its own css classes and it's hard to understand and then adjust them. Another problem with leaflet is that it loads the background map by loading image tiles and putting them together which sometimes takes a lot of time.

DynamoDB

We used DynamoDB to store the data which is needed to populate the charts and map on the dashboard. The reason to use DynamoDB is, it is very fast and efficient. And we needed filtering by date, which is very easy by DynamoDB filter expressions. And as we are using AWS to deploy out project and automate it, and DynamoDB is a part of AWS, it automatically makes it our first choice. The problem faced while using DynamoDB is that you can not have a data type date in DynamoDB, so it is not possible to compare the dates directly. To solve this problem we stored timestamps as a string and compared these string timestamps to filter by date.

Amazon S3

We used Amazon S3 bucket for caching our queries in order to reduce the query load from the database. We saved each query result in s3 bucket with the name having to-date and from-date in it, so next time if someone queries for exact same dates, get the data from s3 file instead of querying the database. This S3 bucket is also used for user searching request. As S3 bucket is part of Amazon and we used amazon to host our application, it was a good choice, because it is easier to integrate and create policies for it when using Amazon for automatic creation and deployment of the environment.



Automated infrastructure through Cloud Formation

It is very important while building an infrastructure in cloud to have the possibility to replicate the exact same infrastructure in a different environment. The best way to do it is without the need of a person interaction. This would be helpful to prevent any human errors while deploying and on the same time replicate everything in the shortest possible time.

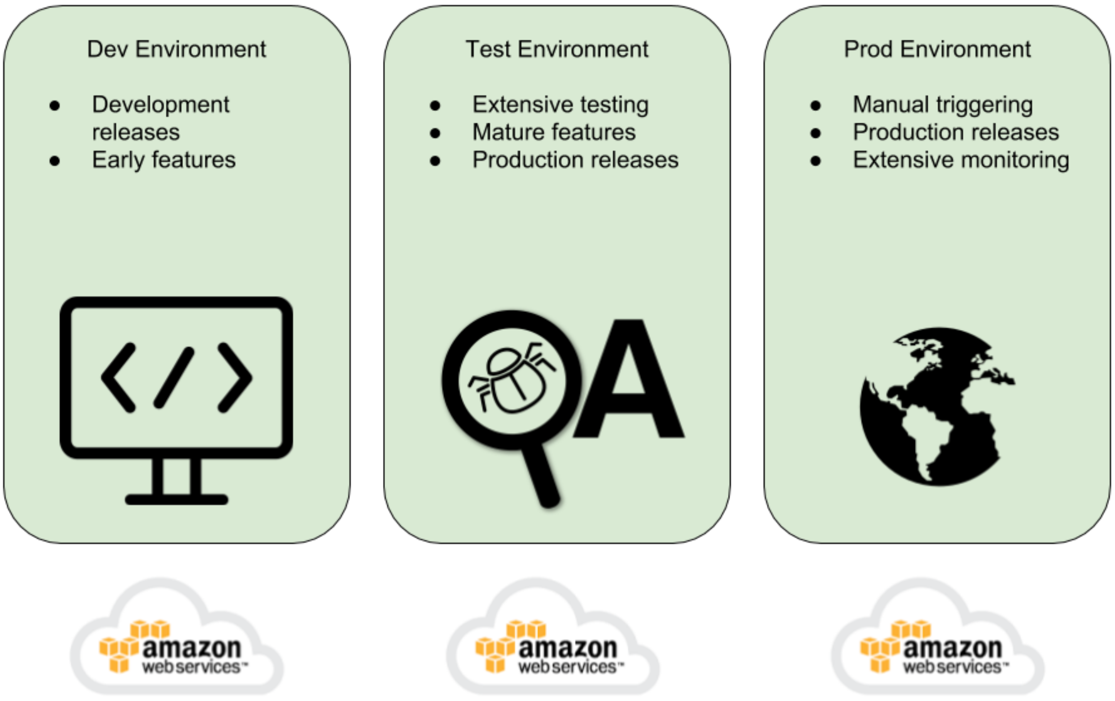
We have considered it as a very important part of our project and this is why we focused since the beginning in coding scripts that would help us to create the infrastructure later on. All this is possible thanks to AWS tool called Cloud Formation which is capable of building any kind of resource that AWS is offering only by using scripts in yaml/yml or json format. There exists a lot of documentation on how to build the scripts in order to create the basics but when it is required a custom and detailed deployment the complexity is increasing. We coded all the scripts to be able to build our infrastructure from the scratch without the need of human interaction to manually configure and deploy each of the used resources. It is only needed to execute this scripts throw aws cli to the AWS account where we want to deploy. The code to execute needs only to pass the parameters and everything is handled by Cloud Formation itself.

In addition, we thought that will be useful to have separate scripts for each of the services that we are creating because this would give us the benefit of reusing them in the future. So we have separate scripts for creating S3 buckets, Roles, DynamoDB, AWS Codepipeline, AWS EBS etc. All these scripts support multiple parameters in order to be able to be reused for different ways of configuration in the future. In the end, we merged the scripts by building a batch file that is automatically calling all of them in the needed order along with the with the appropriate parameters. The batch file is also taking care to configure appropriately the parameters and configurations for each of the environment developer, test and production.

Continuous Integration and Continuous Deployment

On of the main personal objectives this team was willing to achieve performing this project is to learn the CI/CD possibilities Amazon Web Services and Cloud technologies provide. This is sometimes and overlooked tasks in both project planning and project implementation. It is vital for any project having a robust application running with Continuous Integration and Continuous Deployment services managing in an easy manner the work of developers, testing of capabilities and new features and cloud infrastructure deployment.

Three basic environments have been designed for the development, testing and production of the project. Each environment exists within an independent AWS account. Each of them with clear assigned responsibilities and scopes.



Dev Environment

- Development releases
- Early features

Test Environment

- Extensive testing
- Mature features
- Production releases

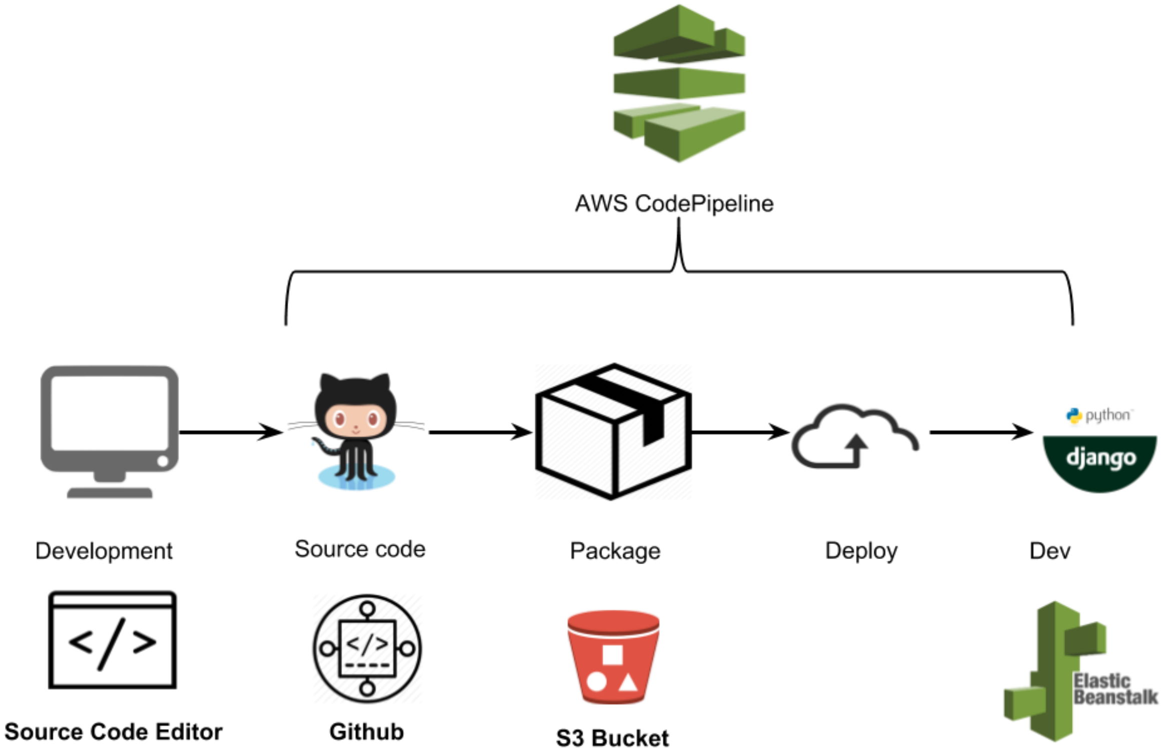
Prod Environment

- Manual triggering
- Production releases
- Extensive monitoring

The project divided the deployment schema in two parts, focusing on the frontend and the backend developed.

CI/CD to manage Django App frontend and Cloud infrastructure

The frontend functionalities are managed independently for its their deployment, including packaging the code and creation of the EBS environment.



This base schema has been implemented as follows:

- One deployment pipeline has been created in the Development environment, tracking the 'development' branch in github.
- Similarly another deployment pipelines has been created in the Testing environment, tracking commits on the master branch of each of the services.
- Finally, one deployment pipelines have been created in the Production environment, tracking as well the master branches of the web application. In this case, with a manual approval before starting the code deployment

CI/CD to manage Serverless application backend

Lambda functions in AWS helps the development of serverless with a high impact. But it adds some difficulty when it comes to deploy and configure serverless applications continuously integrated. In order to achieve this, Serverless Application Framework was adopted for the development and deployment of the serverless backend.

The Serverless framework is a client took allowing users to build and deploy serverless functions. It is actually platform independent as it provides the same interface for any of the event-driven functions providers such as AWS (Lambda) or Microsoft Azure Functions and Google Cloud Functions or Kubeless.

Focusing in the AWS services, Serverless provides structure and automation for building and deploying Lambda functions only focusing on the application. Mostly all Serverless framework functionality is gathered in a enriched YAML file called serverless.yml and it gathers this main concepts:

- Services
- Functions: Specifying the runtime and the callback.
- Events triggering the functions
- Resources needed to have the functions running in the cloud correctly.
- Plugins: Huge amount of open source plugins for different extra functionalities.

Local testing and developing with Serverless framework

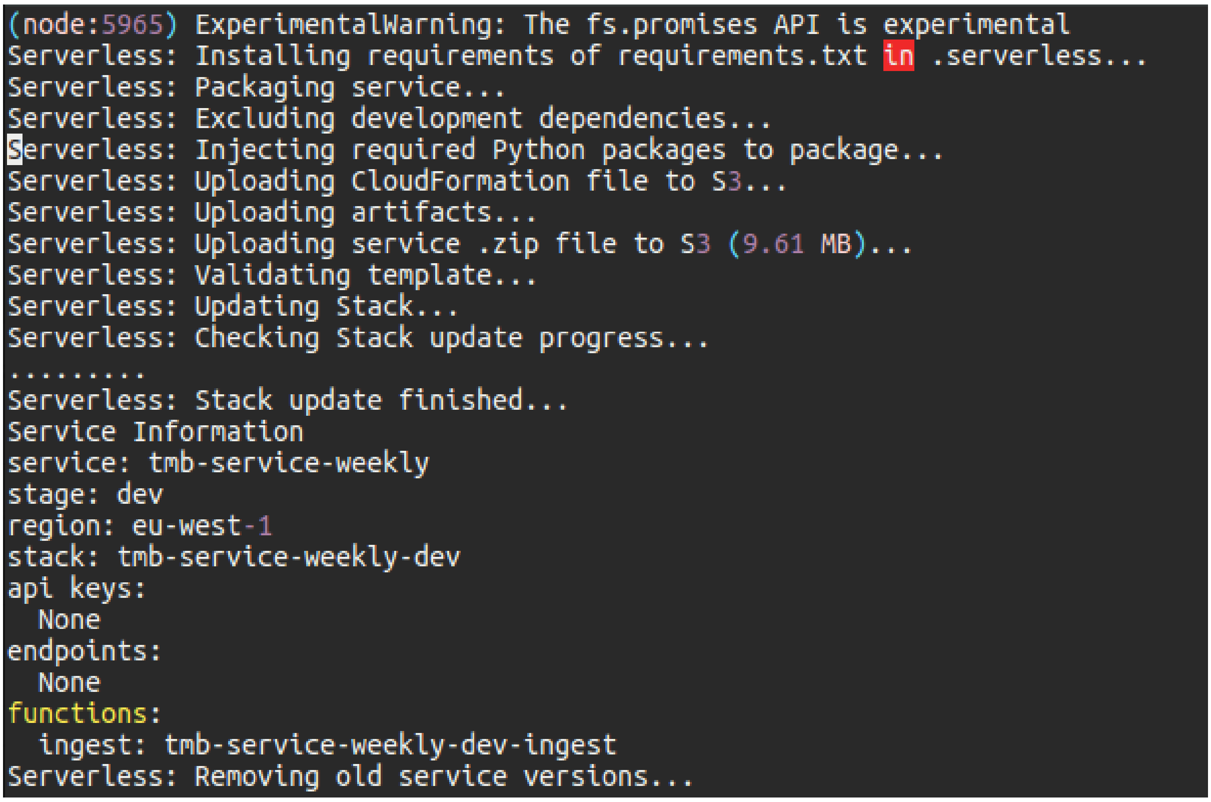
Serverless makes so easy to start coding and deploying lambda functions from your local environment and AWS account. It is only needed to configure the AWS cli locally and serverless.yml with your desired faculties. Without going into much detail this is the configuration on both Lambda functions developed:

Function	**Configuration**	**Value**
Weekly data	Service	tmb-service-weekly
	Function	handler.py#ingest
	Event	Via AWS Cron Syntax: *cron(0 7 ? * MON *)*
	Resources	S3-data-lake
Real time data	Service	tmb-services-realtime
	Function	handler.py#ingest
	Event	Via AWS Rate syntax: *rate(1 minute)*
	Resources	DynamoDB table

With your AWS profile configured correctly, the deployment of your desired serverless application is as easy as:
$> serverless deploy

This command will immediately connect to the specified AWS_PROFILE configured in the local client and create a **CloudFormation Stack** managing all the resources specified in the serverless.yml.

The following images is an example output after deploying with ‘dev’ alias (*$> serverless deploy --stage dev).*



```
(node:5965) ExperimentalWarning: The fs.promises API is experimental
Serverless: Installing requirements of requirements.txt in .serverless...
Serverless: Packaging service...
Serverless: Excluding development dependencies...
Serverless: Injecting required Python packages to package...
Serverless: Uploading CloudFormation file to S3...
Serverless: Uploading artifacts...
Serverless: Uploading service .zip file to S3 (9.61 MB)...
Serverless: Validating template...
Serverless: Updating Stack...
Serverless: Checking Stack update progress...
.........
Serverless: Stack update finished...
Service Information
service: tmb-service-weekly
stage: dev
region: eu-west-1
stack: tmb-service-weekly-dev
api keys:
  None
endpoints:
  None
functions:
  ingest: tmb-service-weekly-dev-ingest
Serverless: Removing old service versions...
```

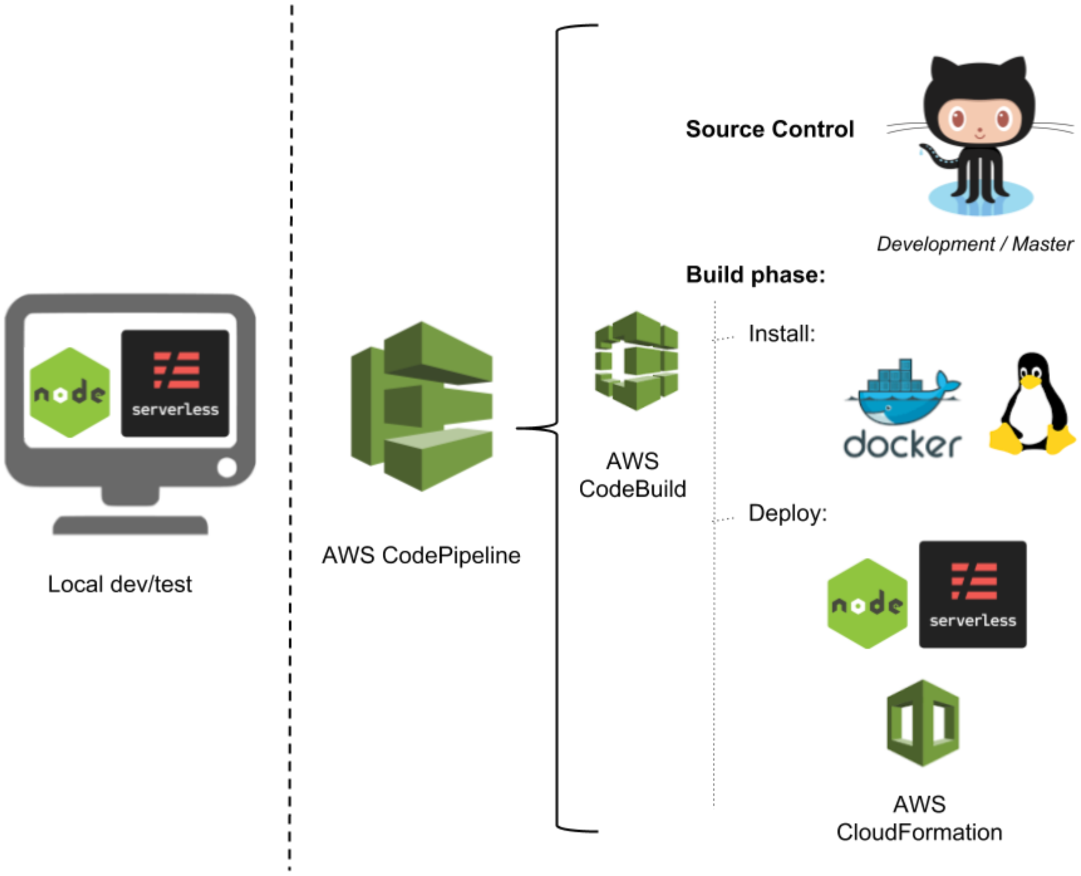
The serverless configurations files can be found respectively in the Github repositories of each services implemented as follows:
Weekly lambda function -> https://github.com/marcgarnica13/CCLambda-weekly/blob/master/serverless.yml

Realtime ingestion -> https://github.com/marcgarnica13/CCLambda-realtime

Continuous Integration and Deployment with Serverless Framework

When it comes to Cloud deployment and code integration for application management, serverless requires some packaging and configuration with AWS Codepipeline as the main actor. The main objective is to keep Serverless framework as the manager for all the infrastructure and code deploy into our serverless backend

For each Lambda function, its development is directly managed through a Github repository. This is the base deployment schema implemented for the project backend.



As explained in the previous section, the local test is completely managed by the Serverless framework. For versionioning and source control a Github repository has been created for each services provided.

Service	Github repository	Branches
Weekly ingestion of data.	https://github.com/marcgarnica13/CCLambda-weekly	development
		master
Realtime ingestion of data	https://github.com/marcgarnica13/CCLambda-realtime	development
		master

This base schema has been implemented as follows:

- Two deployment pipelines have been created in the Development environment , each of them tracking the 'development' branches of each serverless services in github.
- Similarly two deployment pipelines have been created in the Testing environment, tracking commits on the master branch of each of the services.

- Finally, two deployment pipelines have been created in the Production environment, tracking as well the master branches of the two services. In this case, with a manual approval before starting the build phase of the deployment.

Twelve Factors

We have implemented our project by following the twelve factors recommendations. The section below is discussing each of these factors in further details.

1. **Codebase**
   The project code is tracked and maintained in three repositories in the version control system of GitHub. We have chosen GitHub because each of us was familiar with this provider and we all had a student account to benefit from premium features offered there.
   The repository is composed of several branches for each of the team members, one additional branch for the development account and of course the master branch. The reason behind having a separate branch for each developer is to give them the independence to work and commit on the git repository without having the need to merge every time even when the functionality that they are working on is not completely finished. The developers are pushing the changes to their own branch and when they finish developing they merge with the developer branch. The reason why we use a development branch it is because we have configured a continuous integration pipeline in AWS which is automatically deploying the new application version of the branch on the development account so it can be tested by the developers.
   This approach helped us to prevent possible conflict within different people working on the same part of the code, it was not creating dependency on the work of each other and we had the possibility to go back to earlier working versions in case something was not ok. We consider this technique as very useful and we would highly recommend this approach for further projects.

2. **Dependencies**
   The 12 factors methodology suggest that an application should explicitly declare and isolate dependencies. We have declared all the required packages with their specific version in a file called requirements.txt. In addition, virtualenv were used by all members to isolate the project and ensure that the exact same setup was running on every local machine and later on in the EBS in the cloud. The main benefit is that with this approach it doesn't matter if someone added new features with dependencies to the project since the above pip command can be run once before the start of work to assure the application is running in the same environment as on the other members' machines. The step of installing the new packages is very simple, you need to run just the command: pip install -r requirements.txt.
   This is making the application portable and facilitating developers team growth. Every new team member can easily configure their environments. We didn't face any difficulty in following this approach.

Furthermore each backend services deployed to AWS Lambda functions contains a configuration file serverless.yml.

3. **Configuration**
   Every kind of configuration from the code are separated. No part of the code has hard-coded table names, paths or configurations in general. All the configurations are stored in environment variables which will be different for every environment the application is running(development, test and production).
   We built shell and batch script files which are automatically configuring the values of these environment variables into the local machines of each developer.

4. **Backing services**
   We are using several external and internal resources in our project. But our code and infrastructure is independent of these resources. We are using Amazon S3 bucket and dynamoDB to store our data, and our application is independent of the structure, location and names of these resources. As dynamoDB and S3 bucket both are schemaless storages, they don't need any modifications if something changes in the data. On the other hand if we change the names of the table or s3 bucket or move it from one host to another, the code does not need any changes since it can be changed from the environment variables.

5. **Build, release, run**
   The agile methodology that we used to develop our application makes mandatory to follow the approach of build, release, run in order to be able to deliver well tested new functionalities. We configured Continuous Integration and Continuous Deployment to be able to correctly follow this methodology. We have created separated GitHub branches in order to manage CI/CD in different accounts. For deployment on AWS we used AWS data pipeline that allowed automated and continuous integration upon changes in GitHub. This way of working makes possible to test the application and new functionalities in an independent way.

6. **Processes**
   The application that we have developed is relying only on stateless processes. The data we collect or process are always saved S3 buckets or in DynamoDB database. In case something goes wrong, backing store, which gives you the right to scale out anything and do what you need to do. During stateless processes, you do not want to have a state that you need to pass along as you scale up and out. the information can be easily re-retrieved from these resources.

7. **Port binding**
   We are using EBS which is taking care of the port binding for our application. This service of AWS made it very convenient for us to switch between the different deployment stages since we don't need to configure different ports in those stages. We did not need to worry about port bindings in the different set-ups. In addition, we have configured our application for the declared hosts to allow port binding during the runtime. This is done in the settings file at the configuration line ALLOWED_HOSTS = ['.elasticbeanstalk.com','localhost','127.0.0.1'].

8. **Concurrency**

   We are using AWS Elastic beanstalk to deploy our application. AWS Elastic Beanstalk is an orchestration service offered from Amazon Web Services for deploying infrastructure which orchestrates various AWS services including auto scaling. So for concurrency elastic beanstalk is taking care of scaling up and scaling down the application automatically, according to the usage and requirements. So our application is automatically scaling out according to the work diversity because of elastic beanstalk.

9. **Disposability**

   We are assuring fault tolerance of our web application by using EBS as already mentioned in the previous point. The deployed application through EBS benefits from it's features of automatically creating replicas in other availability zones within the same region. This ensures the high availability of our application. Said that we are aware that the scalability factor is limited due to the free tier that we are actually using to build our cloud infrastructure. This is a limitation that can be easily overcome by deploying the all infrastructure in another account or simply upgrading the current account. We can handle both options because we have created Cloud Formation files to be able to deploy everything in a few minutes in another account by assuring that the configuration are the same.

   On the same time EBS is offering even the possibility of scaling up in case the assigned resources are always used in the maximum limits from the application. This can be easily configured and changed throw the GUI or Cloud Formation templates. The only thing what is needed to be changed is the type of the EC2 instance that EBS is using. The above mentioned configurations allow our application to assure a high level of fault tolerance.

10. **Dev/prod parity**

    We have tried to keep development, staging, and production environments as similar as possible. We have been using an agile methodology with weekly sprints which has obligated us to have several deployments in a short period of time to be able to cope with this development methodology. We have achieved this parity by implementing AWS Codepipeline integration which is automatically building, testing and deploying the new code pushed in the Github. The changes happening in the developer branch are immediately handled by the AWS Codepipeline configured in the AWS account. When the changes are checked in the development account they are pushed in the master branch and from here it is triggered the AWS Codepipeline in the Test environment and Production environment. There is a difference between Test and Production because nothing is released in the Production without having a manual confirmation by the responsible person for that process. This way of working guarantees us that nothing is transferred in the production without being fully tested. On the same time we assure a parity between development and production where they have very small differences between them.

    In addition to the continuous integration/continuous delivery configuration done by AWS Codepipeline, we have used cloud formation to replicate between different environments so that they do not have any differences and we prevent like this any possible human error.

11. **Logs**
We are using AWS CloudWatch tool in order to monitor the application that we have developed and all the services in general. This tool is allowing us to collect metrics and logs files in real time and automatically. This is done by the agents that are installed from AWS in the EC2 instances that are launched by the EBS service. Also other logs are collected automatically from other services like DynamoDB. In addition, we have enabled CloudTrail which records account activity about everything what you create, modify and delete in any of the resources used in AWS.

12. **Admin processes**
We are using another tool of AWS which is helping us to manage in a centralized way different alert that can be raised from the used resources. The used tool is AWS CloudWatch in which is possible to configure rules which can be connected to deliver notifications throw SNS service. This approach is mandatory in an environment like AWS Cloud where we use several resources and it makes it very difficult to handle all these resources.

Project Plan

Hours Distribution

We had a total of 160 hours assigned to this project. As we are a team of 4 people, each person spent 40 hours on the project. Following are the details of all the hours spent on the project.

Meeting hours (common for all team members):

Date	Meeting Purpose	Hours per person
20 April, 2018	Idea finalization and first draft	1.5
23 April, 2018	Architecture Design	2
25 April, 2018	Architecture finalization	1
10 May, 2018	Weekly update meeting	1
18 May, 2018	Weekly update meeting	1
24 May, 2018	Weekly update meeting	1
28 May, 2018	Integration discussion	2.5
Σ		10

So each team member spent 10 hours in the meetings which makes it **40 hours** out of the 160 allocated hours.

Detailed distribution of the rest 120 hours (30 hours per person) are as following:

Duy:

Research	Working with JSON Display data in template with jQuery AJAX with Django Best practices for S3 and DynamoDB	2 hours 2 hours 2 hours 2 hours
Coding	Django app creation and configuration for deployment Search function Stop page Line page	2 hours 4 hours 3 hours 3 hours
Documentation	Report Presentation	4 hours 2 hours
Integration	Cloud Formation	4 hours
Σ		30 hours

Marc:

Research	SOAP WSDL services with Python Clients Python XML parsing Local deployment and testing of AWS Lambda functions (without using the AWS console) Best practices for AWS DynamoDB and S3 Data organization and schema. Continuous Integrations and Deployment with Serverless applications	2 hours 2 hours 2 hours 1 hour 3 hours

Coding	Getting the correct data from the TMB Services Parsing weekly and real time information Uploading weekly information to AWS S3 Uploading realtime information to AWS DynamoDB	1 hour 1 hour 30 minutes 30 minutes
Documentation	Report Presentation	4 hours 2 hours
Integration	Serverless Framework Code Pipeline Code Build (Docker images and installation through the buildspec.yml)	4 hours 1 hour 8 hours
Σ		30 hours

Noor:

Research	Charting libraries and understanding how it works Mapping libraries and understanding the functionalities Automated Integration	2 hours 2 hours 1 hour
Coding	Django App creation Dashboard charts Dashboard heatmaps Dashboard s3 caching Dashboard date filtering Integration Testing and Dashboard bug fixes	1 hour 2 hours 5 hours 1.5 hour 2 hours 2.5 hours
Documentation	Report Presentation	4 hours 1 hour

Integration	Cloud Formation (with Todi) Cloud Pipeline	6 hours
Σ		30 hours

So hours spent in total = meeting hours + 30 = **40 hours**

Todi:

Research	Continuous Integrations and Deployment. Cloud Formation.	2 hours 6 hours
Cloud Formation Scripts	S3 Buckets Roles Policy DynamoDB Codepipeline Elastic Beanstalk Cloud Watch	1 hour 1 hour 2 hour 1 hour 4 hours 2 hours 1 hour
Documentation	Report Presentation	3 hours 1 hour
Integration and infrastructure deployment		6 hours
Σ		30 hours

Total = meeting hours + 30 = **40 hours**

Deviation from Initial Plan

Initially we planned to finish everything in the week of 21st May, but we slipped until the week of 28th May, due to the workload and problems occurred during configuration and development. Apart from that, the execution of the initial plan have been followed with high accuracy and the project main objectives did not change.

Unimplemented Features

- Displaying bus lines on maps.
- Autocomplete search box.
- Domain and routing configuration through AWS Routing 53.
- Extensive cloud monitoring tools for the production environment.

Conclusion

Learning Outcomes

As mentioned during this document, this project was started with the main objective to gain a positive learning on how to deploy robust, reliable and integrated solutions to the cloud. There is no denying in that the cloud services are more and more conquering all the software solutions because of its resilience to failure and infinit (with reasonable limits) resources.

The main outcome extracted from this project is the first experience with an end-to-end implementation of cloud software and the work done into building the most stable and comfortable environment to work. Not only on code committing through Github and AWS Code pipeline but also in how to store and back any cloud infrastructure with CloudFormation Stacks in order to (1) be able to replicate a desired environment in seconds (2) deploy the infrastructure changes in the same way any developer will deploy his new feature in the code.

More specifically, the experience gained with AWS Lambda Functions and serverless applications in general has been very useful and interesting as well. Deploying serverless applications to the cloud gives to the developer the opportunity to **only** focus in the application functionalities and forget about the code infrastructure and execution runtimes. More importantly serverless functions such as AWS Lambda manage with an outstandingly easy manner how to trigger certain functionalities of our backend: API gateways, scheduled events, SNS queues or any other possible event that we can imagine.

Serverless framework is still a new tool with a lot of features to add but it already has a huge community behind due to the fact that supports mostly all the Cloud providers offering serverless solutions. It has been also a very good outcome on how to manage serverless application reducing the vendor lockin implicitly added for example through the AWS console.

Major Problems and Solutions

One of the first major problems encountered was the high dependency with the TMB services, this project decided to consume their services in order to provide an additional service to the citizens and some feedback to the TMB management. The experience dealing with Transport Metropolitans de Barcelona has not been perfect and we realised some lack of documentation and description of the services, something crucial in the Open Data methodologies. To overcome that slowdown in the project we first struggled to get the specific services that we required for our functionalities but the team designed a cache level making the connection to the TMB services completely transparent and resilient to any external failure. Obviously if TMB service stop providing information, TMB Helper would not make any sense.

Another huge step in the project was the design of the final deployment schema and the tools required to build a nice development workflow that enables the developer team to focus on the code and infrastructure design. Due to the team experience on this topics, this was a very difficult task that after a lot of research was managed correctly. At the end, the environment established is enough for a first MVP creation and deployment but also for further additions to the projects, not only code but also changes in the infrastructure. The team is satisfied with the codepipeline created in both backend and frontend channels.

Another problem we faced was that we were trying to have a single code pipeline in one account acting as devOps and it should push the code on all other accounts (testing, staging, production etc) according to the requirements. While trying to achieve this we faced major problems in cross account communication specially when creating the roles policies from one account in other account to give access of resources of that account. To solve this problem, instead of having single pipeline, we create separate pipelines for each account i-e now dev, staging, testing, production have their own code pipeline by running single click deployment script to create the infrastructure.

Future Extensions

- As future extension we plan to make the project more robust and even more automated by exploiting additional technologies.
- In terms of features we plan to add a feature to detect exact location of the user using GPS in order to help them more in finding nearest stop, as well as it will provide us data on the exact locations where most of the users search for particular stops, and may be this data will help in better locating the bus stops in the cities.
- Review the security aspect of our cloud environment and application. In terms of cloud environment we must review all roles that are being used by different services like EBS, Codepipeline, DynamoDB, S3 buckets etc. The existing role may have more than needed access on some resources that they are not using and this may create security issues. On the application side as well is needed some review in the security aspect and some penetration tests should be mandatory in a real life production environment.

Code links:

- https://github.com/todithanasi/bcn-bus
- https://github.com/marcgarnica13/CCLambda-realtime
- https://github.com/marcgarnica13/CCLambda-weekly
- https://github.com/todithanasi/cloudformation