Weather Analytics Web Application

Master in Data Science, FIB
Cloud Computing and Big Data Analytics

May 26th, 2023

Miona Dimic, Mateusz Jerzy Galinski, Poly Kinya, Odysseas Kyparissis, Joan Oliveras

Table of Contents

Introduction

The purpose of this project is to create an informative and user-friendly web application about the current and historical weather conditions in all 50 provinces of Spain. This application is intended to be available on the AWS[1] cloud platform, taking into account the following advantages that AWS offers:

- **Scalability**: AWS provides auto-scaling features for automatically adjusting the number of resources allocated.
- **Reliability**: AWS guarantees that web apps stay available even if there are hardware failures.
- **Flexibility**: Wide selection of available services and tools, allow us to fulfill all our needs of design in almost every possible way, considering the optimization of the performance and cost.
- **Cost-effectiveness**: Following pay-as-you-go pricing model, we can pay only for the resources that we use. Transparency is essential when it comes to estimation of the possible costs, so AWS cost management is one of the most important aspects.
- **Security**: Given the possibility of using specific security features offered by AWS, inside of IAM, VPC, KMS, WAF, etc, we are certain that desired security can be covered and properly monitored.
- **Global Infrastructure**: In case we want to expand the reach of users, AWS has a vast global infrastructure with data centers located in multiple regions worldwide.
- **Developer-Friendly**: Variety of SDKs (Software Development Kits) offered within AWS, enables us to make it easy to build, deploy and manage web applications.

Taking this into consideration, the goal of the project is to design and implement a system that can offer the desired quality for the web application. By combining and integrating available AWS services we intended to create a design able to avoid single points of failure, provide security of information and infrastructure, as well as elasticity, decoupled components and performance optimization.

In the following sections, we will explain, in more detail, the description and the main goal of the project, the designed implementation pipeline, as well as its deviation from the original design idea. In the designed implementation pipeline section of the report, the resources used for each AWS service are presented, as well as how the project benefits from those and why we've chosen them. We will also focus on the task allocation and time management tool used to properly manage the project's development between the team members. Moreover, in the last section of the report, a listing of the hours invested, by each member, for all the main tasks of the project is taking place. Additionally, there, we comment out how the team deviated from the original time management, and how this deviation could be minimized. Finally, in each respective section of the report an explanation of how the project is following the Twelve Factors

[1] https://aws.amazon.com

Methodology Framework[2], together with a presentation of the main problems encountered during the project's evolution is taking place .

One must take into account that certain modifications from previous structured project proposals were made, due to occurrence of problems, such as the lack of AWS permissions. As we were given the student account to carry out this project, some of the desired functionalities offered by AWS were forbidden to be used. Nevertheless, we managed to find a proper alternative and will also present the more advanced possibilities assuming that missing permissions are given.

Last but not least, in the following links one can find the software generated for the implementation of the specific project:

- https://github.com/mionaD-upc/ccbdaProject
- https://github.com/odyskypa/django-weather-app

More information about the contents of the repositories is presented in the following chapters.

[2] https://en.wikipedia.org/wiki/Twelve-Factor_App_methodology

Description and Main Goal of the Project

As previously mentioned, the main focus of the project is to create a weather-related web application. The idea is to apply Extract, Transform, and Load (ETL) operations to information gathered from an external trusted resource and generate insights to be presented to the end user within our own cloud-based application. The external source used is the website OpenWeather[3], which provides an Application Programming Interface (API) offering several weather data collections, both real-time and historical. For the specific implementation, we solely focus on retrieving information about real-time weather conditions, leaving the rest of the collections to be used for future expansion of the application.

Moreover, to be able to create a functional demo of the application, the initial location chosen for which we obtain the weather information is Spain. This should not be seen as a limitation, but a focused use case, to guarantee the functionality on a small scale. Additionally, once we obtain the desired look, expanding the scope of data retrieval would enable us to include more geographical locations, thus attracting more users worldwide. In addition, the existence of the AWS centers worldwide would support this scalability.

On the user side perspective, the application would provide, simple dashboarding interface, containing graphical visualization, as well as filtering functionality, for the current and historical weather metrics such as temperature, precipitation, pressure, etc., for all the 50 provinces of Spain. As the external resource chosen provides multilingual data, ideally our application would enable the same interface represented in different languages in the future.

In addition to the basic weather measurements, the application should be able to capture extreme weather conditions from the retrieved data and notify the users accordingly. Finally, we would like to personally attract users, enabling them to use this feature. One idea would be to make the clients sign in into our web application, and access these additional features. In this way everyone would be able to obtain essential weather information, but only those subscribed are to be notified for additional information such as daily recommendations given the current weather data. Although, due to lack of time during the project development, this feature is to be implemented in a second phase.

Once we set up the initial motivation and desired functionality of our application, in the following section we provide a detailed description of the implementation pipeline and the system's design for accomplishing the above-mentioned utilities.

[3] https://openweathermap.org/

Implementation Pipeline

Overview

In this section, both the original proposed and the final system design of the web application are presented. To begin with, in Figure 1. the original proposed design is depicted, consisting of several decoupled components, which are collaborating, having as a common goal to offer a smooth and user-friendly functionality of the application.

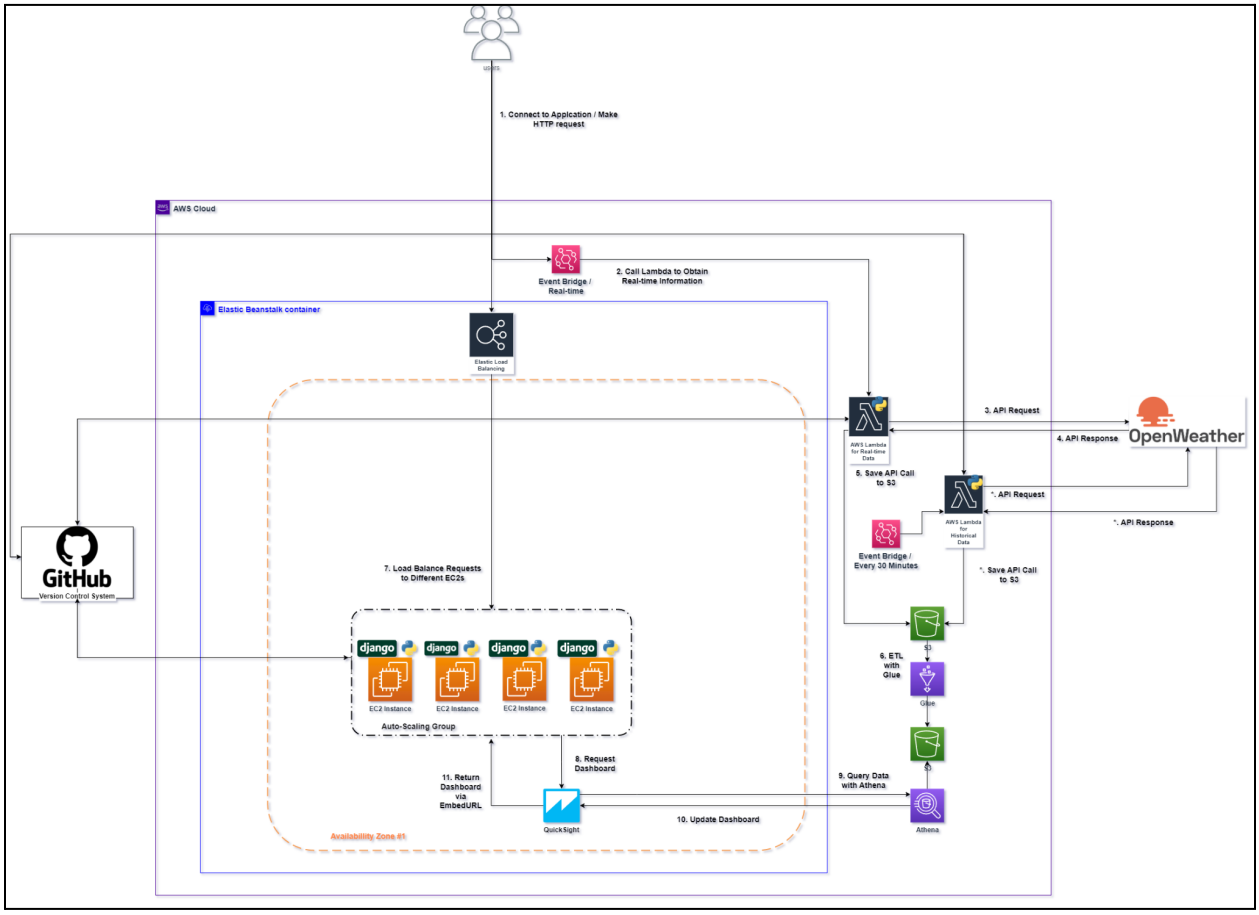

Figure 1. Original Proposed Application Design

As it can be seen in the diagram, the desired components would include an Elastic Beanstalk[4] (EBS) component that consists of an Elastic Load Balancing[5] (ELB) component and an Auto-scaling Group[6] (ASG) of EC2[7] Instances. The ELB would redirect the users' requests to the EC2 Instances of the ASG. The ASG would increase or decrease the number of EC2 instances being available to the users, based on several metrics and key performance

[4] https://aws.amazon.com/elasticbeanstalk/
[5] https://aws.amazon.com/elasticloadbalancing/
[6] https://docs.aws.amazon.com/autoscaling/ec2/userguide/auto-scaling-groups.html
[7] https://aws.amazon.com/ec2/

indicators (KPIs), such as website traffic, latency, etc. Those instances would contain the latest version of the application deployed, and they would all be connected to the QuickSight component, which is providing the dashboard with the insights of the weather information via an embedURL[8] format. Furthermore, every time a user would connect to the website or make an HTTP[9] request via the Load Balancer, the application would call the Lambda[10] function which is sending an API call to the OpenWeather API requesting for the current weather information of all Spanish provinces in real-time. Then once the response was returned, the same function would create a JSON[11] file into an S3[12] bucket in AWS. Consequently, with the help of the Glue[13] service, an ETL process would start, transforming the JSON files into CSVs and reassuring the appropriate format of the information. The result of this ETL process would again be stored in a separate S3 bucket. Then, with the usage of specific queries implemented by using the Athena[14] service, QuickSight would be able to retrieve the new information and present the updated dashboard to the user. Finally, a separate lambda function would be calling the OpenWeather Api, independently of users requests, every 30 minutes via an Event Bridge[15] for gathering information which can be used as weather history for further usage. For clarification, the Lambda functions would be written in Python[16], and the application running on the EC2 instances with the Python framework Django[17]. Both the application's and the lambdas' code would be uploaded to GitHub[18], for version controlling reasons. The design schema includes numbers in each step and component of the process, indicating the order of execution of the processes.

In comparison with the original, Figure 2. presents the final design, which is described by certain limitations, mainly due to permission problems opposed by AWS Identity and Access Management[19] (IAM) restrictions, forced by AWS itself for the Learner Lab account.

As it can be clearly seen from the diagram of the final design, the delivered implementation is completely different from the proposed one. The limitation of security policies attached to the Learner Lab user, created a chain of problems that lead to several changes in the original design. To be more specific, the initial problem occurred after the finalization of the first version of the Django web application. Initially the application was developed and tested on local machines and it was working properly. Then, the deployment of the application with EBS took place. Although the application was functioning properly in the local environment, when it was deployed on AWS premises using EBS, the IAM policies of the Lab Learner role were applied, leading to the failure of ingesting the embedURL of QuickSight dashboard inside the application.

[8] https://schema.org/embedUrl
[9] https://en.wikipedia.org/wiki/HTTP
[10] https://aws.amazon.com/lambda/
[11] https://www.json.org/json-en.html
[12] https://aws.amazon.com/s3/
[13] https://aws.amazon.com/glue/
[14] https://aws.amazon.com/athena/
[15] https://aws.amazon.com/eventbridge/
[16] https://www.python.org/
[17] https://www.djangoproject.com/
[18] https://github.com/
[19] https://aws.amazon.com/iam/

After this failure, it was understood that QuickSight was blocked in general from accessing the remaining services of AWS. For that reason, we had to find an alternative for both the creation of the database and the generation of the dashboard.

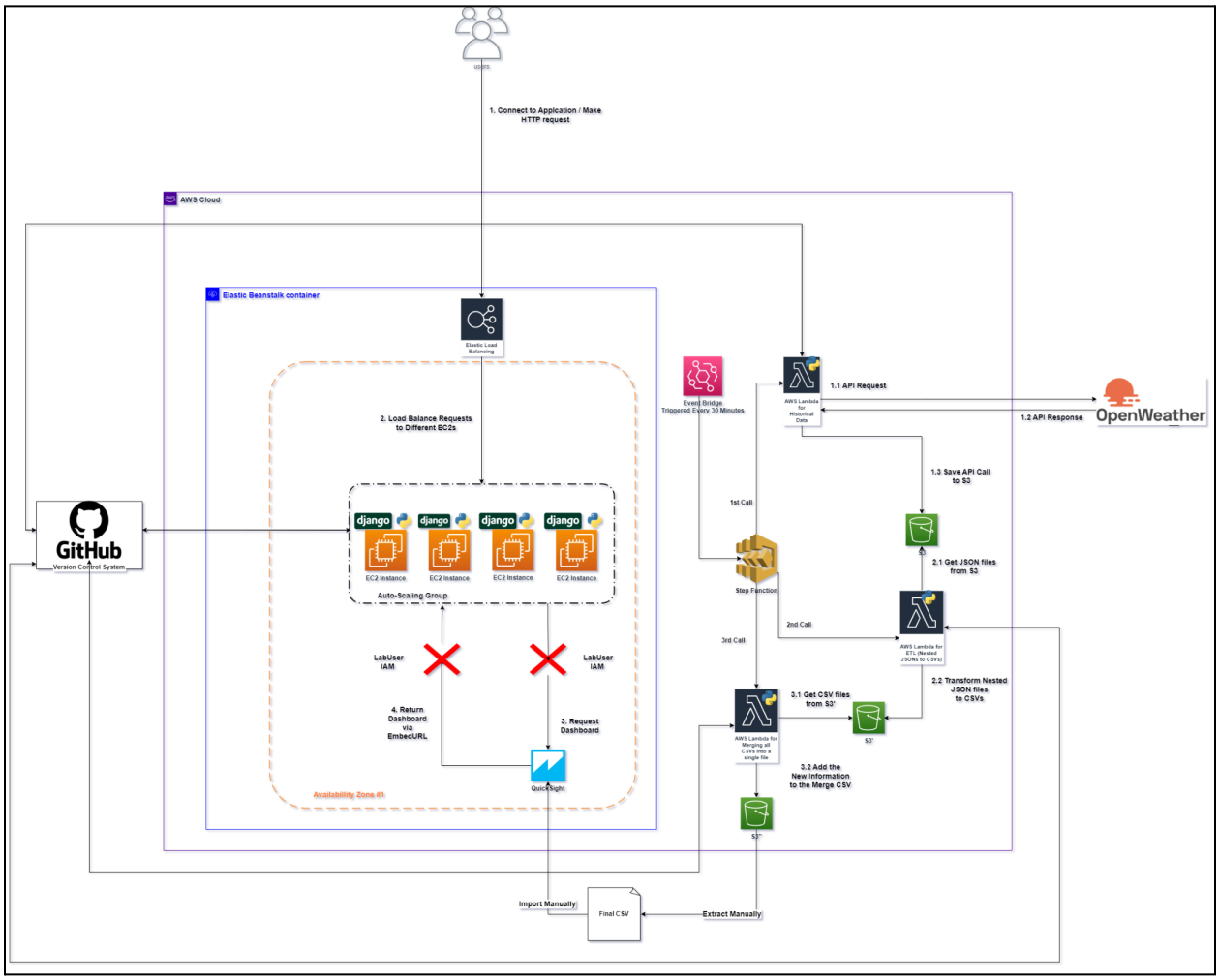

Figure 2. Final Application Design

Consequently, taking into account the limited amount of time that was available for the development of the project, we ended up with the following implementation. Instead of using AWS Glue to apply the ETL processes and create the database, which would be queried directly from QuickSight with the usage of Athena, the creation of a Step Function[20] occurred. It is important to note here, that for the above-mentioned reason the Django application can still be automatically deployed to AWS with EBS, however an error message indicating that the authentication process has failed is presented from AWS. However, the creation of the dashboard in QuickSight was completed and only the connection between the web application and the dashboard is missing for the completion of the solution. Although this problem was faced, the Step Function was able to provide an alternative solution. More precisely, every 30 minutes an Event Bridge is triggered, calling the Step Function which consists of 3 individual lambda functions. Each function is called in a serialized manner. The first function makes a call

[20] https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html

to the OpenWeather API requesting the weather information. When the result of the request is returned from the API the function saves the response in JSON format inside an S3 bucket. Then, the second function is called, which applies an ETL process. It reads the JSON file generated by the call of the 1st function from the S3 bucket where it was placed, flattens its nested structure and transforms it into a CSV with the appropriate format for each variable. Finally, the 3rd function concatenates the result of the 2nd function into a single unified CSV file containing the total amount of information gathered. With this approach, we were able to manually extract the latest version of the data and upload them manually into QuickSight to present a final solution for the project. We acknowledge that this is not a solution that would work in a real-case scenario and especially in a Cloud Computing Analytics, where the size of the files are huge. To conclude, once the merged CSV file is uploaded to QuickSight, the generation of the dashboard is completed. Unfortunately, the connection of the application with the QuickSight service is not possible and for that reason two different demos are going to be presented, one for each individual component.

To sum up, in the following subsections, each distinct component of the system is described and compared with the original proposed components, including the benefits that they offer to the project as well as their limitations.

AWS Lambda for Historical Data Retrieval

Using the available free pricing plan provided by OpenWeather's API , we could access the following information, with restriction of making 60 calls per minute:

- Current Weather
- 3-hour Forecast 5 days
- Basic weather maps
- Weather Dashboard
- Air Pollution API
- Geocoding API
- Weather widgets

Simply by creating an account one can obtain the personal API key to be used for every desired call.

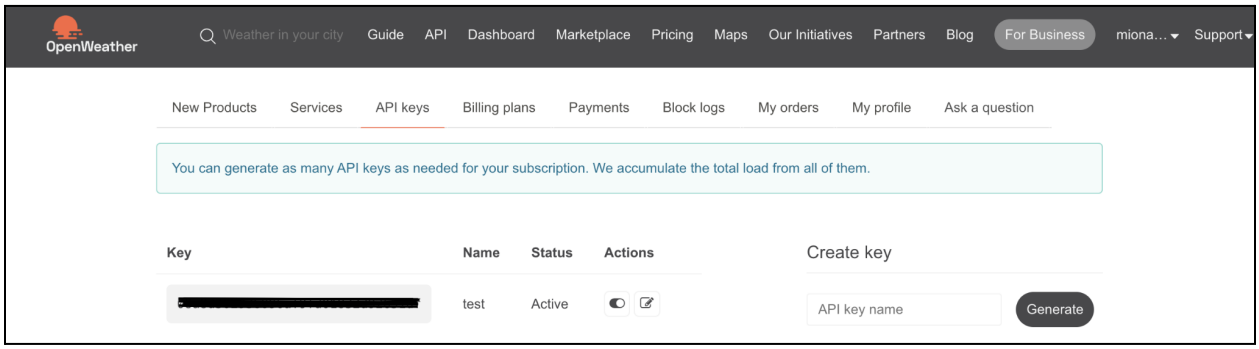

Figure 3. Obtaining personal API key

Getting the Data from OpenWeatherMap API with a Simple Python Script

In order to fully understand the API response structure, before using AWS services, we created a simple script that would make an API call for the current data.

As previously mentioned, location focus is Spain, and specifically the capitals of all provinces. To obtain coordinates of all provinces, it was firstly needed to make API calls with all fifty province names. Retrieving coordinates of those was possible using Geocoding API - another tool created by OpenWeather.

An example of one of those API calls is the following. Here we obtain coordinates given the location Barcelona.

Example of API call for obtaining coordinates

```
http://api.openweathermap.org/geo/1.0/direct?q=Barcelona&limit=5&appid={API key}
```

Format of a Geocoding API response and available fields are accessible through the following link: https://openweathermap.org/api/geocoding-api.

Since the coordinates are not going to be changed, we decided to create a single json file consisting of all the responses received when passing fifty province names. These were stored in coordinates.json file.

Reading the coordinates from this file, and passing those to another API call, enabled us to retrieve weather information for each location. The responses are then appended to another single json file, representing the current weather data of all provinces in Spain.

Example of API call for obtaining current weather data

```
https://api.openweathermap.org/data/2.5/weather?lat={lat}&lon={lon}&appid={API key}
```

Format of an API response and available fields are accessible through following link: https://openweathermap.org/current

Since real-time responses are not available for Free pricing plan, we decided to simulate real-time weather conditions, just by scheduling API calls every 30 minutes.

Once we managed to perform this locally ensuring that the response is in the right format, we proceeded to implement this process equivalently in a specific AWS service.

Getting the Data from OpenWeatherMap API with AWS Lambda Function

To be able to perform previously mentioned data retrieval inside of the AWS, we decided to use **AWS Lambda**.

Using AWS Lambda for data retrieval with API calls offers us automatic scalability, enabling efficient handling of varying levels of traffic without the need for manual infrastructure management. It also provides cost efficiency by charging only for the actual compute time used during each request.

As our Lambda function requires certain dependencies such as **requests** library that by default is not supported by AWS, it was necessary to create a deployment package with required dependencies[21].

To the created deployment package, one must additionally add **lambda_function.py** file. Example of a lambda function created for this task is listed below. As it uses coordinates for all 50 provinces, a json file containing those has been also added to the deployment package.

```
import json
import requests

def process_json_file(file_path):
   # Retrieving current weather data for all provinces in Spain
   with open(file_path, 'r') as f:
       data = json.load(f)

   results = []

   for inner_list in data:
       dictionary_object = inner_list[0]
       lat = dictionary_object['lat']
       long = dictionary_object['lon']
       response = make_api_call(lat, long)
       results.append(response)
   return results

def make_api_call(lat, lon):
     API_URL  =  f"https://api.openweathermap.org/data/2.5/weather?lat={lat}&lon={lon}&appid={API
key}&units=metric"
   response = requests.get(API_URL)
   return response.json()


def lambda_handler(event, context):
   json_file_path = 'coordinates.json'
   currentWeather = process_json_file(json_file_path)
   return {
       'statusCode': 200,
       'body': 'Successful function execution'
   }
```

[21] https://docs.aws.amazon.com/lambda/latest/dg/python-package.html

Uploading the created zip of the deployment package into the AWS Lambda created on AWS console, led us to accomplish the data retrieval process.

Storing the API Responses in AWS S3 Bucket

We decided to make an API call every 30 minutes, and thus it was needed to add an additional trigger to the lambda function. As the AWS lambda working file system is read only, storing json files was impossible to do unless we define a storage unit in which we would keep the API responses.

As a storage unit we decided to use an **Amazon S3 bucket**. S3 is preferred over other options as it supports scalability, durability, ease of accessibility, comprehensive feature set (versioning, lifecycle management), and cost-effectiveness. Additionally, S3's integration with other AWS services and its widespread adoption in the industry make a good fit for our application. The usage of a database, for example RDS or DynamoDB, could be an alternative option at this stage. On the other hand, a DataBase Management System (DBMS) would offer a more structured solution for storing information, as well as more efficient data retrieval for the creation and update of the dashboards with the usage of queries via the Athena service. However, due to the IAM limitations mentioned before that forbidden the access of QuickSight to databases, we concluded to develop the current solution using S3 buckets.

Before adding a trigger, we modified the lambda accordingly, so that it stores the json response in the specified S3 bucket. It was necessary to have an empty S3 bucket available in order to perform this task.

```
import boto3
import pytz
import json
import requests
from datetime import datetime, timezone

s3_client = boto3.client("s3")
LOCAL_FILE_SYS = "/tmp"
S3_BUCKET = "weatherdatabase"  # please replace with your bucket name


def process_json_file(file_path):
   # Retrieving current weather data for all provinces in Spain
   with open(file_path, 'r') as f:
       data = json.load(f)

   results = []

   for inner_list in data:
       dictionary_object = inner_list[0]
       lat = dictionary_object['lat']
       long = dictionary_object['lon']
       response = make_api_call(lat, long)
       results.append(response)

   return results
```

```
def make_api_call(lat, lon):
     API_URL = f"https://api.openweathermap.org/data/2.5/weather?lat={lat}&lon={lon}&appid={API
key}&units=metric"
   response = requests.get(API_URL)
   return response.json()

def _get_key():
   spain_tz = pytz.timezone('Europe/Madrid')
   dt_now = datetime.now(tz=spain_tz)
   KEY = (
       dt_now.strftime("%Y-%m-%d")
       + "_"
       + dt_now.strftime("%H")
       + "_"
       + dt_now.strftime("%M")
       + "_"
   )
   return KEY


def lambda_handler(event, context):
   key = _get_key()
   json_file_path = 'coordinates.json'
   currentWeather = process_json_file(json_file_path)

   file_name = key+'.json'
   file_path = '/tmp/' + file_name  # Provide the desired file path

   with open(file_path, 'w') as f:
       json.dump(currentWeather, f)
   s3_client.upload_file(file_path, S3_BUCKET, file_name)

   return {
       'statusCode': 200,
       'body': 'Successful function execution'
   }
```

Adding an EventBridge Trigger

Additionally, a trigger which is executed every 30 minutes, was created, and configured throughout AWS Console, integrating the **Amazon EventBridge** with AWS Lambda. This service provides a scalable and flexible event-driven architecture, allowing us to easily connect and trigger Lambda functions based on events from various sources.

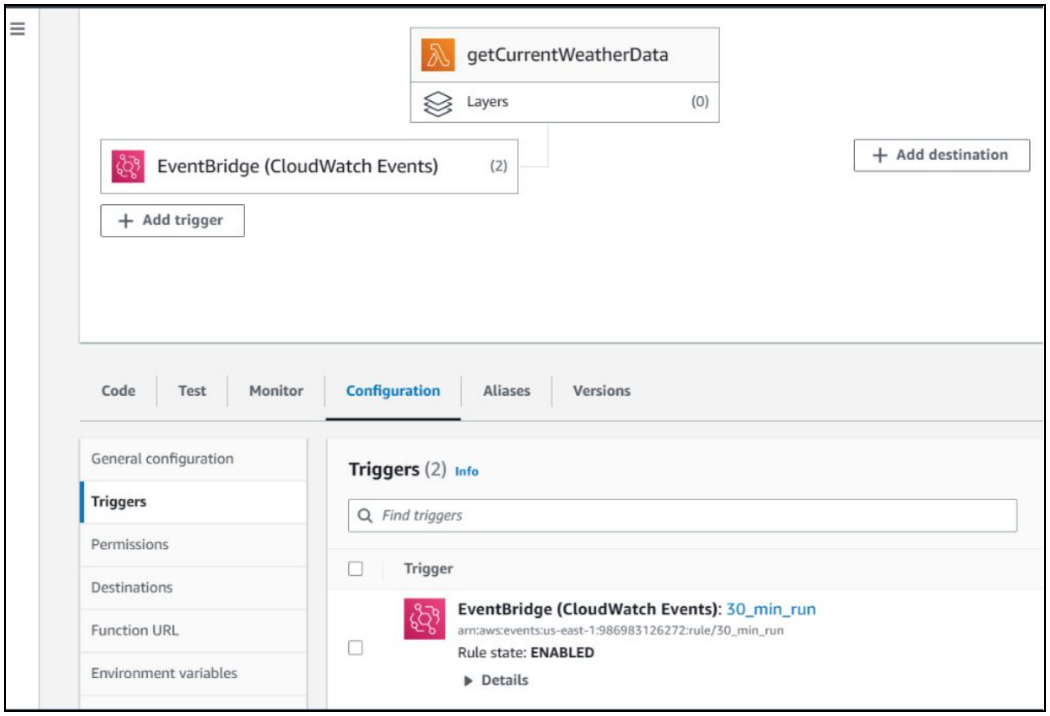

Figure 4. EventBridge Trigger

Result of this integration to previously defined function, resulted with S3 bucket populated, as represented in the following picture. Each file is uniquely identified with a timestamp of the data retrieval (transaction) time and consists of the weather data recorded in that exact time.

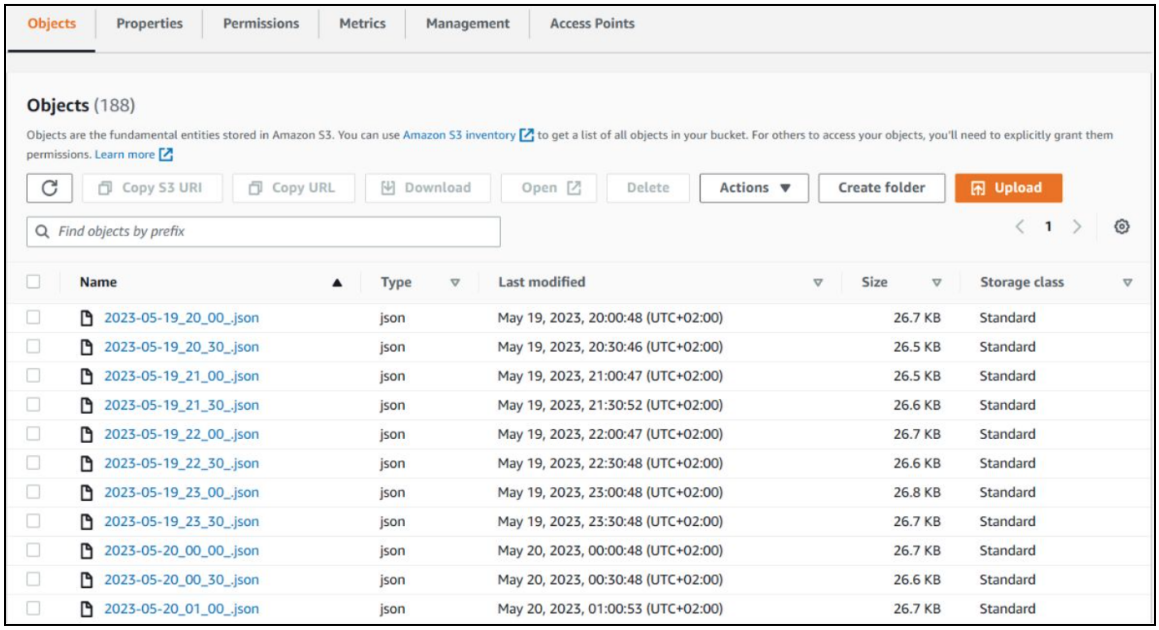

Figure 5. Populated AWS S3 bucket

ETL Pipeline Design with Step Function

After ensuring that the job for collecting the data from API calls was working and properly scheduled, we started designing the pipeline to be followed in order to use the data for visualization purposes with Quicksight. Here, we considered multiple options, AWS Athena and AWS Glue.

Initial design

Firstly, we planned to incorporate AWS Glue for the initial data transformation. AWS Glue is a managed ETL service that automatically discovers, catalogs, and transforms data from various sources. We intended to leverage its capabilities to cleanse and standardize the API data.

By creating a Glue job, we would process the raw data from the API calls, performing transformations such as filtering irrelevant information, removing duplicates, and converting data types as needed. This would ensure the data's consistency and suitability for visualization.

After the initial transformation using AWS Glue, we intended to integrate AWS Athena into the pipeline. AWS Athena is an interactive query service enabling SQL-based analysis of data directly in Amazon S3. Leveraging its serverless nature, we would conduct ad-hoc querying and analysis on the transformed data.

With AWS Athena, we would define tables and schemas based on the transformed data in Amazon S3, allowing SQL queries to extract specific subsets of information for visualization. This would enable effortless exploration, aggregation, filtering, and joining of tables to generate meaningful visualizations.

Therefore, by combining AWS Glue and AWS Athena, we would establish an efficient file transformation pipeline. The Glue job would convert the raw data into a clean, structured format, while AWS Athena would offer flexibility and speed in querying and extracting insights from the transformed data.

Implementation problems

As researched, this state of art approach does design a correct and sound ETL & data analysis process, however, it is overcomplicated for our chosen project.

In our case, we mainly wanted to use AWS Glue to transform and process the files. Once we started designing the Glue job, we initially created a crawler that would automatically detect the structure of the json files and store that metadata to easily transform them. However, we found issues due to the way we had designed our json files.

In each file we were storing information about 50 cities. To do so, the information was stored as an array with 50 elements, each element using nested JSON structures with over 20 variables.

Thus, even though Glue has jobs for flattening and treating nested structures in JSON, our case was too complex and not easy to create such an ETL process.

Regarding AWS Athena, it was studied to provide direct access to Quicksight from the extracted JSON files, by querying the exact information needed and avoiding creating an ETL process. However, with the structure that we had designed in our json's, we quickly saw that this was not possible, as it was not straightforward to map the structure of each file.

Finally, what we decided to do is to create the transformation ourselves by using different Lambda functions and to orchestrate the jobs to ensure we have a sound ETL design.

AWS Lambda for ETL (Nested JSONs to CSVs)

First, we focused on creating a Lambda function that would transform the arrays of nested JSON's into a filetype that we can use for our further analysis, in our case, CSV.

To do so, we used the following code, which retrieves a JSON file from an S3 bucket (the one keeping the API call), processes its contents by mapping specific columns and converting nested structures, and generates a CSV file. The CSV file is then uploaded to a target S3 bucket, and the original JSON file is moved to an archive bucket (with all the past API calls).

Three points to consider:

- We designed a new column: Identifier. Not present in the API call, that would take the filename as identifier, thus giving us information about the timestamp in which the data was taken.
- By converting the file to a list of desired columns, we ensured that all the processed files had the same schema in the future, creating a solid transformation that would ensure consistency of the data.
- By keeping moving the processed JSON file to an archive bucket, we ensure reproducibility of our transformations, allowing us to go back and modify any changes if needed. We also kept the S3 bucket with the API calls empty, allowing us to trigger this function any time new files were arriving.

```
import json
import csv
import os
import boto3

s3 = boto3.client('s3')

def lambda_handler(event, context):
    # Get the name of the S3 bucket
    source_bucket = 'weatherdatabase'
    target_bucket = 'weather-csv-data'
```

```
    archive_bucket = 'json-check'

    # List objects in the source bucket
    response = s3.list_objects_v2(Bucket=source_bucket)
    if 'Contents' in response:
        # Assuming there is only one file in the bucket
        source_file_name = response['Contents'][0]['Key']
    else:
        return {
            'statusCode': 404,
            'body': 'No files found in the source bucket'
        }

    # Extract the file name and change the extension to .csv
    file_name = os.path.splitext(source_file_name)[0] + '.csv'

    # Download the JSON file from the source S3 bucket
    temp_file_path = '/tmp/' + source_file_name
    s3.download_file(source_bucket, source_file_name, temp_file_path)

    # Read the JSON file with UTF-8 encoding
    with open(temp_file_path, 'r', encoding='utf-8') as file:
        data = json.load(file)

    # Define the desired columns
    columns = [
     'Identifier', 'coord.lon', 'coord.lat', 'weather.id', 'weather.main', 'weather.description',
        'weather.icon', 'base', 'main.temp', 'main.feels_like', 'main.temp_min',
        'main.temp_max', 'main.pressure', 'main.humidity', 'visibility', 'wind.speed',
        'wind.deg', 'clouds.all', 'dt', 'sys.type', 'sys.id', 'sys.country',
        'sys.sunrise', 'sys.sunset', 'timezone', 'id', 'name', 'cod'
    ]

    # Generate the CSV file
    csv_data = []
    identifier = os.path.splitext(source_file_name)[0]
    for city_data in data:
        city_row = {'Identifier': identifier}
        for column in columns[1:]:
            keys = column.split('.')
            value = city_data
            for key in keys:
                if isinstance(value, list):
                    value = ', '.join([str(item[key]) for item in value])
                    break
                if key in value:
                    value = value[key]
                else:
                    value = None
                    break
            city_row[column] = value
        csv_data.append(city_row)

    # Generate the CSV content as a string
    csv_content = ''
    if len(csv_data) > 0:
        csv_content = ','.join(columns) + '\n'
        for row in csv_data:
            csv_content += ','.join(str(row.get(col, '')).encode('utf-8').decode('utf-8') for col in
```

```
columns) + '\n'

    # Upload the CSV file to the target S3 bucket
    s3.put_object(Body=csv_content.encode('utf-8'), Bucket=target_bucket, Key=file_name)

    # Move the previous file to the archive bucket
    s3.copy_object(Bucket=archive_bucket, Key=source_file_name, CopySource={'Bucket': source_bucket,
'Key': source_file_name})
    s3.delete_object(Bucket=source_bucket, Key=source_file_name)

    return {
        'statusCode': 200,
        'body': 'CSV file created successfully, uploaded to S3, and previous file moved to archive'
    }
```

AWS Lambda for Merging all CSV's into a single file

Secondly, we focused on creating a Lambda function to merge all the previously transformed CSV's into a single file, which would be used later for data analysis and visualization purposes.

To do so, we used the following Lambda function, which retrieves multiple CSV files from an S3 bucket, merges them into a single CSV file, and uploads the merged file to another S3 bucket. It provides a convenient way to consolidate data from multiple sources for further analysis and processing.

The output of this lambda function contained a CSV with the previously defined list of columns and all the historical API calls for all the 50 cities as individuals (rows). The name of the generated file was also containing the timestamp of creation, thus allowing us to compare between different files while always keeping the most recent information.

This generated CSV file was then used for data visualization.

```
import csv
import os
import boto3
from datetime import datetime

s3 = boto3.client('s3')

def lambda_handler(event, context):
    # Get the name of the S3 bucket
    source_bucket = 'weather-csv-data'
    target_bucket = 'final-database'

    # List objects in the source bucket
    response = s3.list_objects_v2(Bucket=source_bucket)
    if 'Contents' not in response:
        return {
            'statusCode': 404,
            'body': 'No files found in the source bucket'
        }
```

```
    # Create a list to store the data from all CSV files
    csv_data = []

    # Process each CSV file in the source bucket
    for file_obj in response['Contents']:
        # Download the CSV file from the source S3 bucket
        temp_file_path = '/tmp/' + file_obj['Key']
        s3.download_file(source_bucket, file_obj['Key'], temp_file_path)

        # Read the CSV file
        with open(temp_file_path, 'r', encoding='utf-8') as file:
            reader = csv.DictReader(file)
            for row in reader:
                csv_data.append(row)

    # Check if any CSV data was collected
    if len(csv_data) == 0:
        return {
            'statusCode': 404,
            'body': 'No CSV data found in the source bucket'
        }

    # Extract the columns from the first row
    columns = csv_data[0].keys()

    # Generate the CSV content
    csv_content = ''
    if len(csv_data) > 0:
        csv_content = ','.join(columns) + '\n'
        for row in csv_data:
            csv_content += ','.join(str(row.get(col, '')).encode('utf-8').decode('utf-8') for col in
columns) + '\n'

    # Generate a timestamp for the file name
    timestamp = datetime.now().strftime("%y%m%d%H%M")

    # Upload the merged CSV file to the target S3 bucket with timestamp in the file name
    target_file_name = f'merged_data_{timestamp}.csv'
    s3.put_object(Body=csv_content.encode('utf-8'), Bucket=target_bucket, Key=target_file_name)

    return {
        'statusCode': 200,
        'body': f'Merged CSV file created successfully and uploaded to
{target_bucket}/{target_file_name}'
    }
```

Orchestration of the Lambda functions

Step Function

Even though initially an EventBridge Trigger was defined to call the API every 30 minutes. This was not our final solution in the project. After creating the ETL process via different Lambda Functions, we decided to Orchestrate them externally to be able to keep track of the full process and errors that could be generated. Thus, we chose to use AWS Step Functions to do so.

AWS Step Functions is a fully managed service that allows you to coordinate and orchestrate multiple AWS services and serverless functions into serverless workflows. It provides a visual workflow editor to design and visualize the flow of your application's components, making it easier to build and coordinate complex, distributed applications.

By using AWS Step Functions we ensured the following benefits:

- Workflow Coordination: Ensured proper sequence and dependencies.
- Error Handling: Automatic retries, timeouts, and error management.
- Visual Workflow Design: Intuitive visual editor for workflow creation.
- Simplified Development: We just focused on individual Lambdas, not orchestration.
- Monitoring and Logging: Detailed tracking of progress and errors with logs.
- Integration with AWS Services: Seamless integration with other services.
- Scalability and Parallel Execution: Ability to handle large data and concurrent tasks.
- Step-Level IAM Permissions: Granular control over access and actions.

Even Though some of those points are not key for our project (mainly due to the reduced scope of it) it was interesting seeing which potential this tool could have in a larger process.

In our case, we created a State Machine which is a visual representation of a workflow. It defines the states and transitions that determine the execution flow of our system or application.

Thus in our case, when the State Machine gets triggered, it initializes each of the Lambdas consequently and finally ends the process.

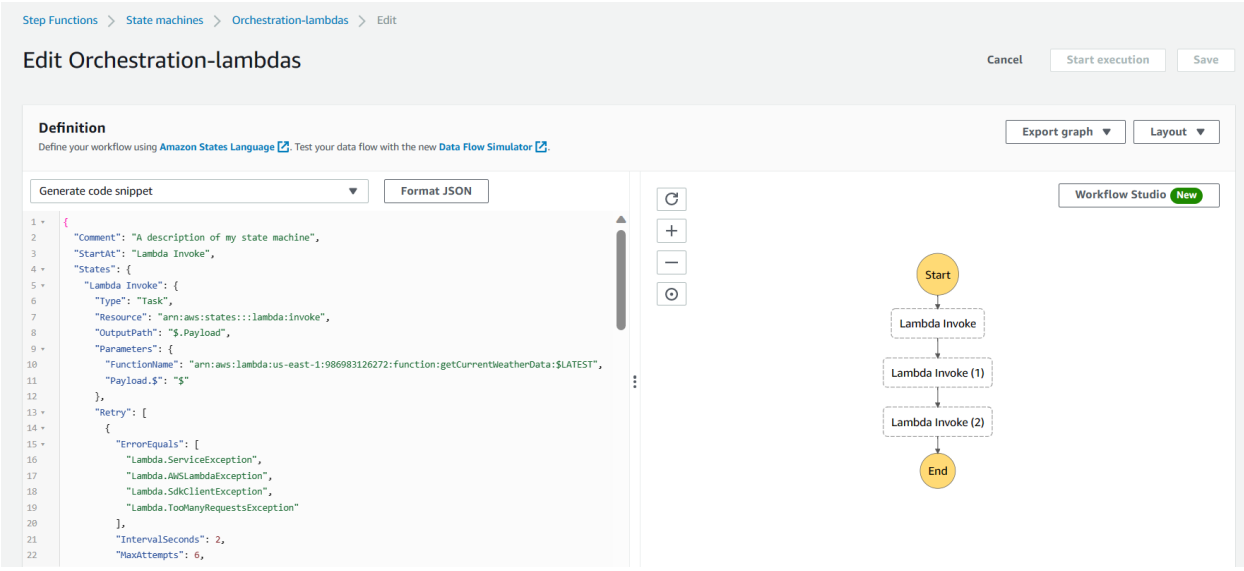

Figure 6. Design of our State Machine

In the following picture, you can find our State Machine with all the triggered events and its status, logs and information.

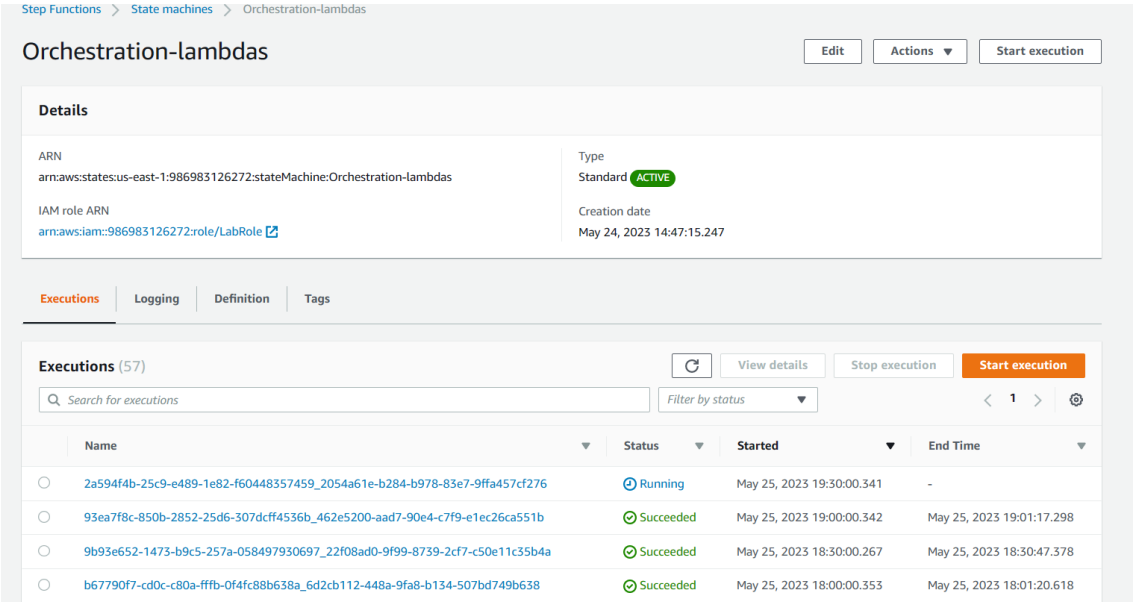

Figure 7. Overall info State machine

Finally, in order to trigger the State Machine, we need to create a new EventBridge rule. Similar to the one of 30 minutes created in the first Lambda, however, choosing as target the created State Machine.

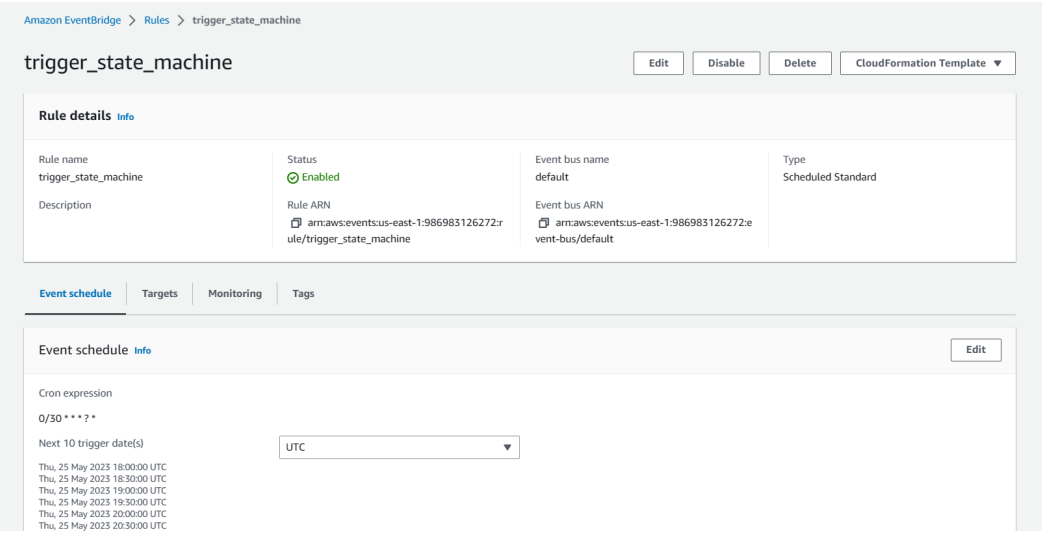

Figure 8. Design trigger of the State Machine

And with this, we have our orchestrated ETL Pipeline Design done and working.

Dashboard Creation with AWS QuickSight

The dashboard was initially conceived according to Figure 1, where the data would be ingested using Athena from how it was stored in the S3 buckets. This would allow for a handless

dynamical update of the data every 30 mins due to the API call cycle. However due to the limitations of the Student Learner's Lab account such automatic implementation was impossible to be achieved due to IAM authentication policy and the inability to change it with respect to the access to QuickSight. Hence the idea had to be adopted to now be based on manually uploaded CSV files that underwent the ETL process mentioned in the previous subsection.

After uploading the CSV file we had to properly encode the fields so that the dashboarding would be possible.

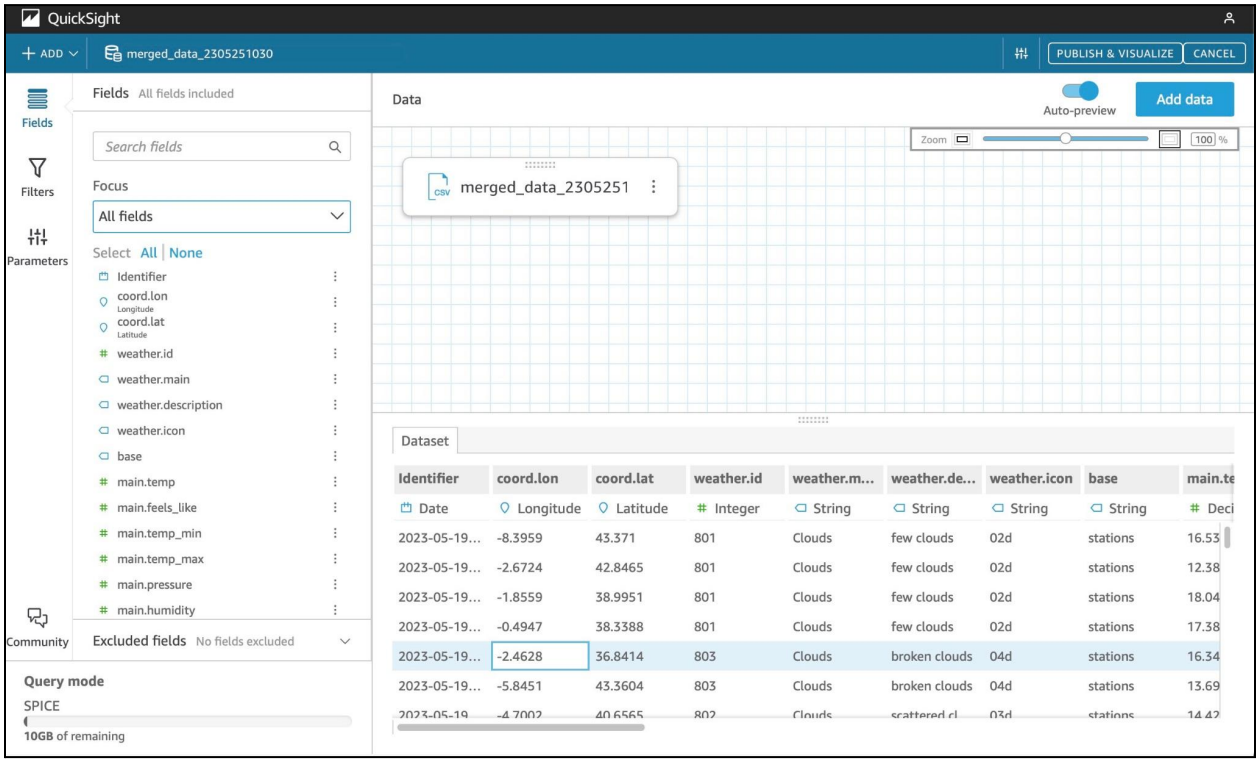

Figure 9. Ingestion of data

This involved properly encoding the *Identifier* as datetime column with appropriate format, the *coord.lon* and *coord.lan* as longitude and latitude respectively and finally the *sys.country* field as country in order to allow pre-filtering of the data to only include Spain. Since what we found was that sometimes inappropriate data would get ingested in the API call and it would pass undetected through the ETL pipeline which would result in some results for countries other than Spain, mainly Mexico or USA.

After that was done we could finally start creating the dashboard. QuickSight makes that process user-friendly and quicker to deploy and as a result easier to test different possibilities for dashboarding. After some trial and error we settled on the creation of 4 graphs in the delivered dashboard. The first 2 of them would consist of the map of Spain with the geographical locations of the provinces shown on it using the longitude and latitude coordinates. One of them would display the average *feels-like temperature* for the current day. The other one would display the *maximum wind speed* for the current day. Both charts are interactive, allowing

the user to view the detailed data for the wind or temperature simply by hovering their cursor over the place on the map. The following 2 figures present the first two graphs of the dashboard.

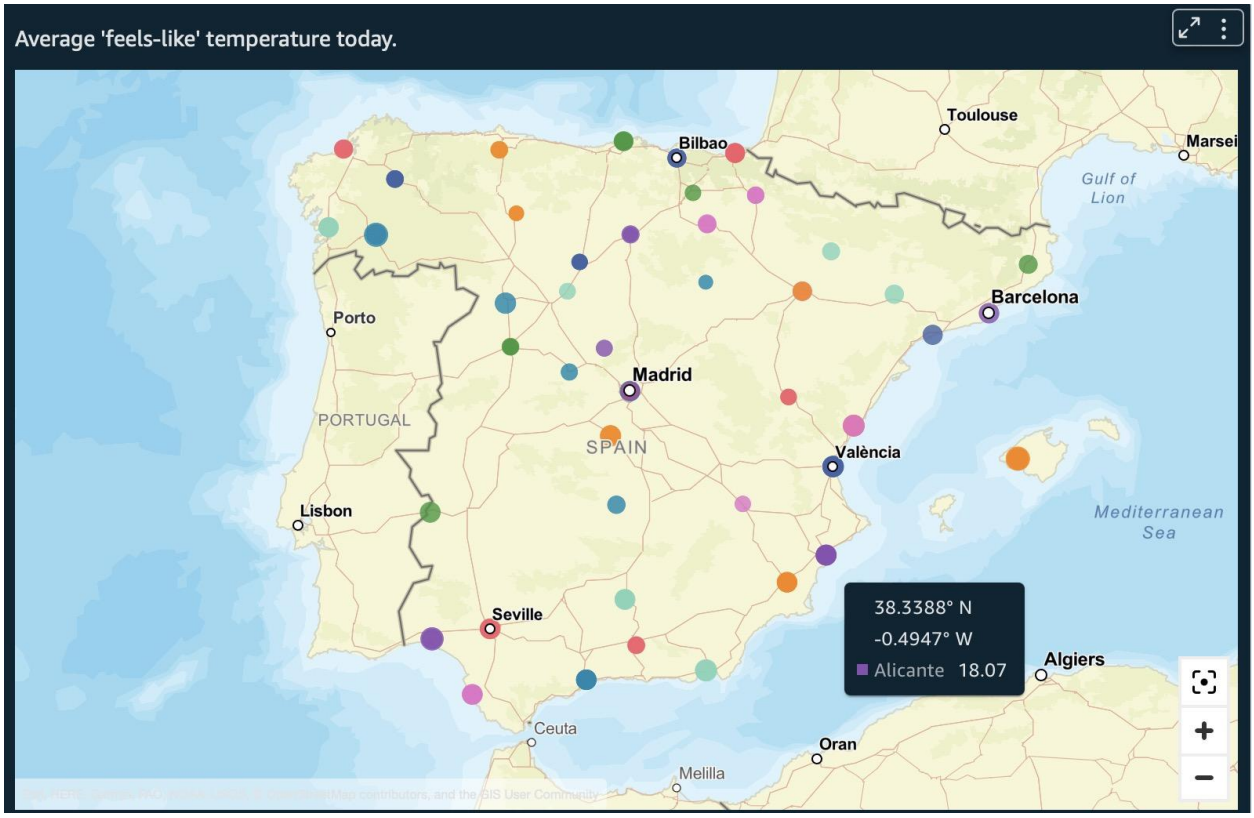

Figure 10. Graph showing ‘feels-like’ temperature

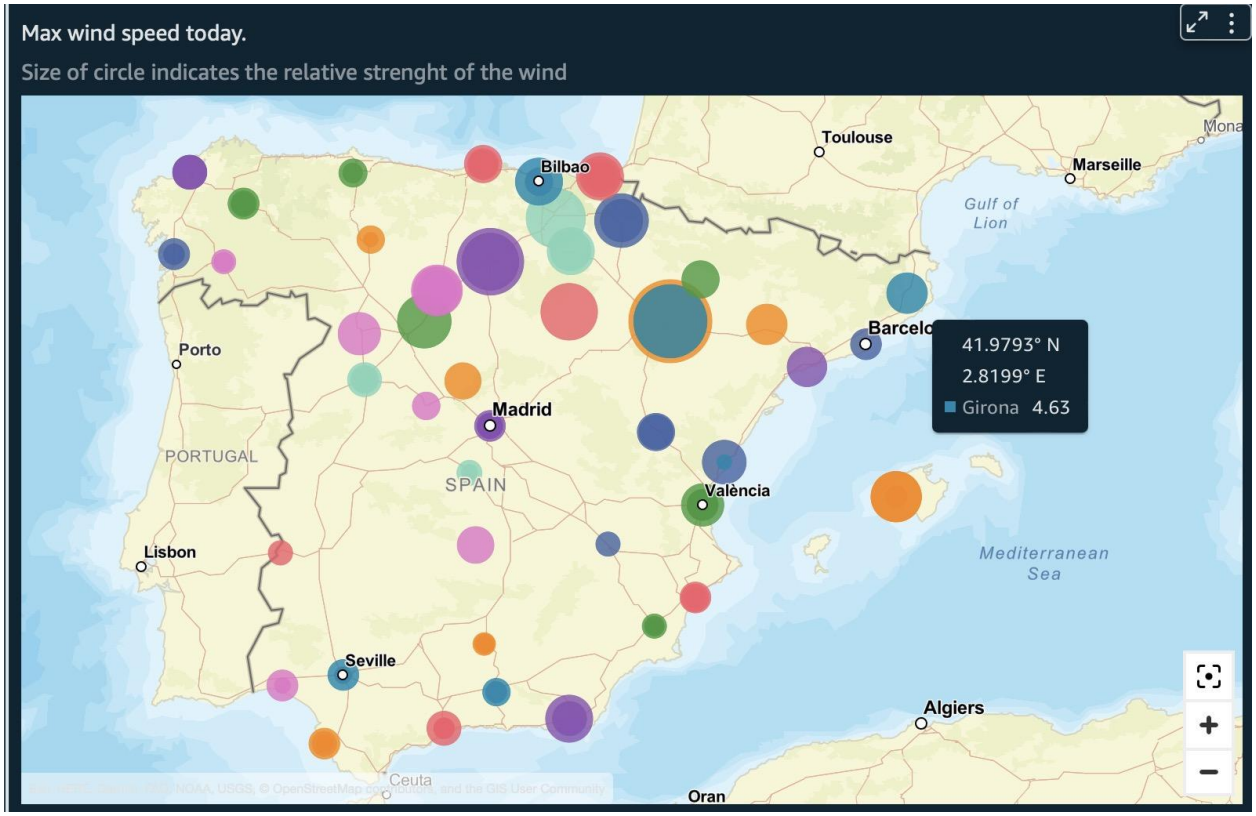

Figure 11. Graph showing max wind speed

After this was completed the creation of the remaining 2 graphs was completed. This time it was desired to use a different format, so not a location map, but one that would still allow us to see the data on the provinces level. In order to implement this in a visually accessible way a filter button was added for those 2 graphs that would allow the user to select which provinces they would like to show the data for. Alongside the button there was a text prompt outlining what the button is meant to be used for.

Figure 12. Province button

The following 2 graphs depict the *average daily humidity* per province and the *average hourly temperature* per province. Those graphs are also leveraging the interactive capabilities of the QuickSight service by allowing the user to view the exact data simply by hovering over the bar/line.

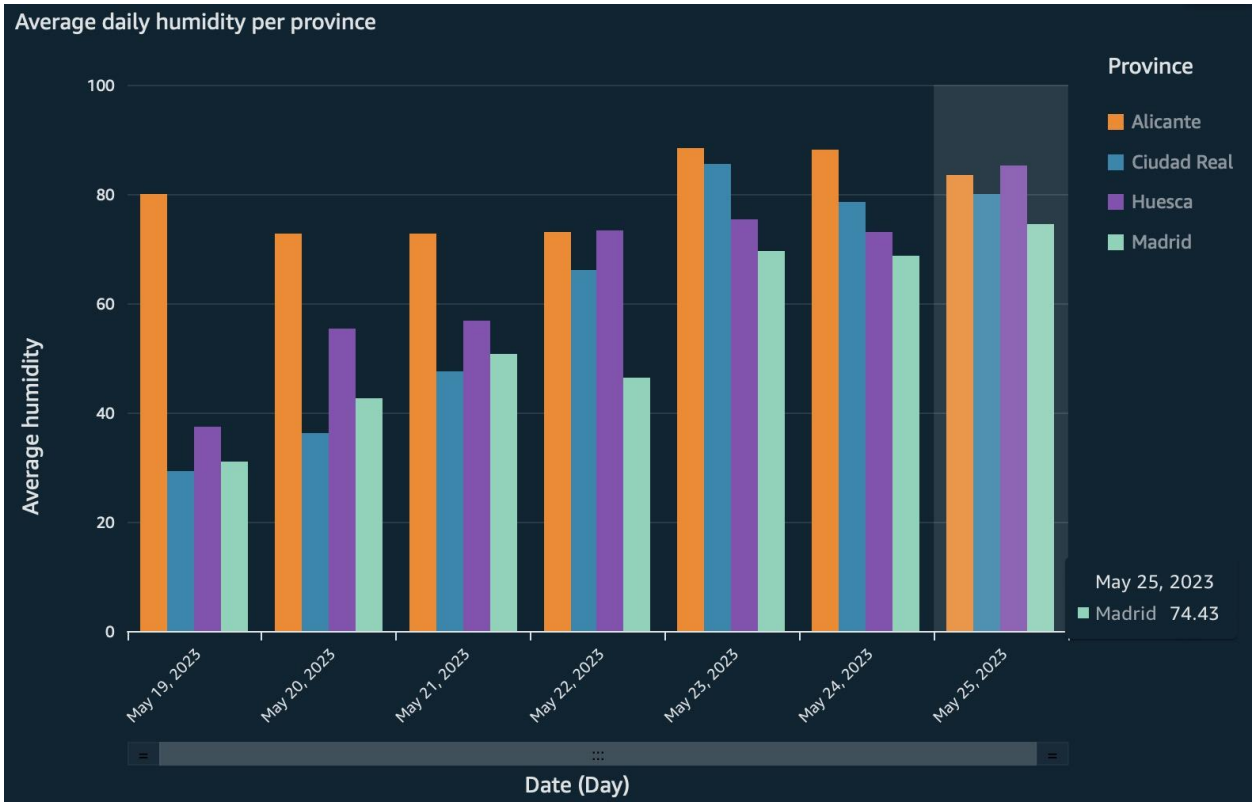

Figure 13. Bar plot for average daily humidity

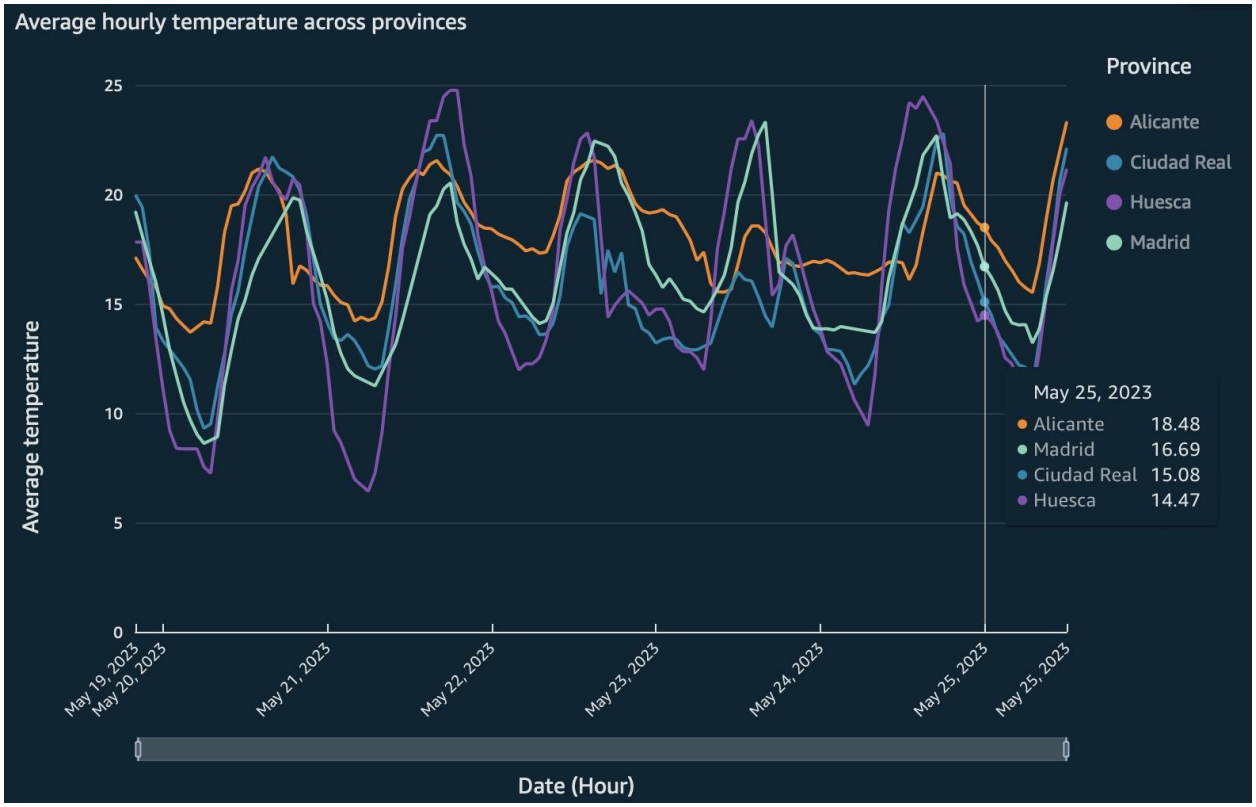

Figure 14. Line chart for average hourly temperature

Finally the following figure presents what the complete dashboard ended looking like.

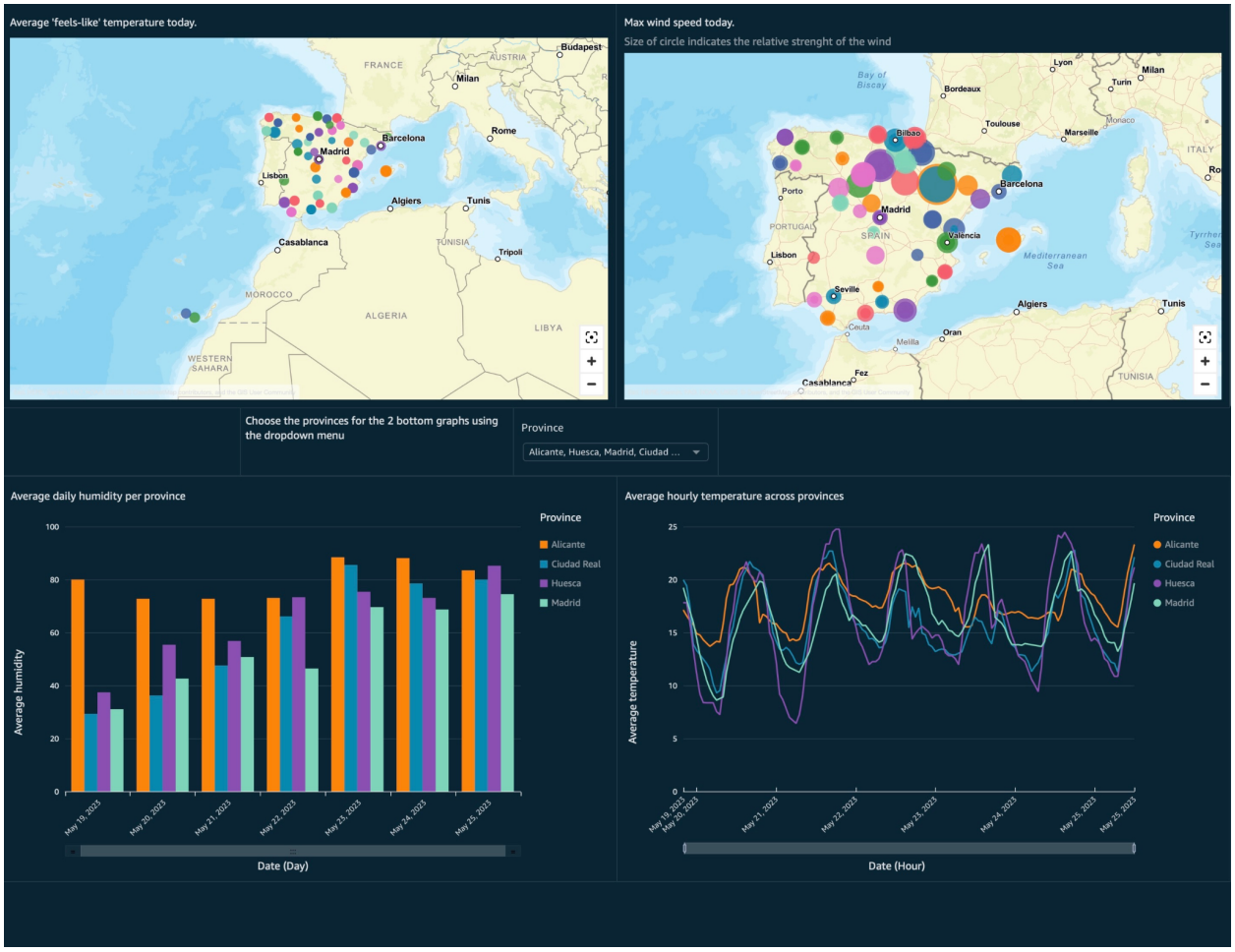

Figure 15. Final dashboard

The dashboard can be viewed from a web browser however due to the missing IAM permissions it is not possible to generate a QuickSight token which is necessary in order to embed the url of the dashboard into the application. Without that token, AWS prevents access to the dashboard from embedded links.

Development of Web Application with Django

The development of the web application with Django forms a crucial aspect of the overall project, as depicted in both Figure 1. and Figure 2.. The envisioned architecture comprises a comprehensive set of components, including EBS, ELB, ASG of EC2 Instances, QuickSight, Lambda functions, and various AWS services. This section focuses on outlining the key elements and functionalities involved in building the web application. Additionally, it highlights the integration of different services and the flow of data within the system. Moreover GitHub was

employed for version control purposes. By examining the following paragraphs, a deeper understanding of the development journey and the underlying infrastructure can be gained.

The Django framework offered numerous benefits throughout the development process. Firstly, it facilitated the use of a single codebase, which could be tracked in GitHub. The codebase used solely for the Django application can be found by following this link. This allowed for efficient collaboration among team members and simplified version control management. Additionally, the framework enabled the explicit declaration and isolation of dependencies. By using the virtualenv[22] Python library, together with a *requirements.txt* file, the project's dependencies were clearly defined, making it easier to manage and update whenever deemed necessary.

Furthermore, Django encouraged the practice of storing configuration variables in the environment. This approach provided flexibility and security by separating sensitive information from the codebase, making it easier to manage different environments such as development, staging, and production. For this part a script named *eb-env.py* was used for exporting the sensitive information as environment variables, such as AWS credentials and API keys, allowing at the same time the easier management of different environments as just mentioned. Furthermore, this design idea led to keeping development, staging, and production environments as similar as possible. By employing environment variables, the application could adapt to different settings and configurations without code modifications, enabling seamless transitions between development stages.

The initial project design was intended to leverage Django's ability to treat backing services as attached resources. Meaning that the application would be integrated with various AWS services such as QuickSight, Lambdas, and the OpenWeather API. In that way, Django would facilitate seamless communication and interaction with these services, enabling the application to gather weather information, transform data, and present it on the QuickSight dashboard. In the current solution, Django is interacting solely with QuickSight, however the ingestion of the dashboard into the application's homepage is forbidden from AWS.

On the other hand, The development process strictly separated the build and run stages. Django's development server facilitated rapid development and testing, while the application deployment to AWS Elastic Beanstalk (EBS) ensured a consistent and scalable environment for production. This separation allowed for efficient scaling and easy management of the application's resources. More information about the usage of the application in both the build and run stages can be found in the README file of the application's repository.

Moreover, Django's execution model, treating the application as a stateless process, further enhanced scalability and reliability. By adhering to this model, the application could be scaled out to handle increasing traffic demands, as well as provide robustness through fast startup and graceful shutdown of processes. Consequently, smooth operation and minimizing downtime during deployment or scaling processes was ensured.

[22] https://virtualenv.pypa.io/en/latest/

The framework also facilitated the export of services via port binding, enabling the web application to be accessed through a specified port. This allowed for easy integration with load balancers and other components within the AWS infrastructure, ensuring optimal performance and availability. More information about those aspects can be found in the following subsection.

Django also supported the treatment of logs as event streams. The framework provided logging capabilities that allowed the application to capture and analyze logs effectively, ensuring proper monitoring and troubleshooting during the development stage. However, in the production stage EBS takes care of this aspect.

Lastly, Django offers the possibility of execution of admin or management tasks as one-off processes. By using Django's management commands, various administrative tasks, such as data updates or data migrations are enabled. However, they were not used in this specific solution. Ultimately, the current state of the application is described in the *Comments on Current State of the Application* section of the README file of the repository.

Deployment of the Application with Elastic Beanstalk

The deployment of the web application with Elastic Beanstalk led in ensuring a scalable and reliable infrastructure for the project. By utilizing EBS, the application benefited from various factors that enhanced its deployment process and overall performance.

To begin the deployment process, the project's code was configured within a virtual environment, ensuring a clean and isolated environment for the application to run. Dependencies were installed based on the *requirements.txt* file provided in the repository, as mentioned before, explicitly declaring and isolating the necessary packages and libraries.

Next, the project leveraged the power of Elastic Beanstalk's configuration capabilities. The deployment command *eb create* initiated the creation of the application's environment on EBS, while passing the required environmental variables, such as AWS access credentials and region information. More information about the deployment of the application can also be found in the README file of the repository. Additionally, the application code, including the Django framework and the defined views, models, and templates, was bundled and deployed to the created EBS environment. In that way, the separation of build and run stages was enabled and ensured a smooth transition from development to production and provided a consistent and scalable environment for the application.

Furthermore, by utilizing the EBS environment, the project was able to execute the web application as a stateless process. This approach maximized scalability, as the application could handle increasing traffic demands by automatically scaling the number of EC2 instances within the ELB component. Elastic Beanstalk's ASG monitored various metrics, such as website traffic and latency, to determine the optimal number of EC2 instances needed to serve user requests effectively.

Moreover, Elastic Beanstalk facilitated the export of services via port binding. The application's traffic was efficiently distributed to the EC2 instances by the ELB, which is automatically configured in EBS, ensuring load balancing and high availability. This integration with the ELB component allowed the project to transparently integrate with other AWS services and further enhance the application's performance.

To continue with, the application benefited from fast startup and error-free shutdown processes, allowing for efficient scaling and ensuring a smooth user experience during deployment or scaling events. EBS's management of EC2 instances and load balancing is also taking care of maintaining the application's availability and minimizing disruptions. However, in the specific solution, it was not possible to test this functionality, due to the obvious reason that not multiple users are accessing the application simultaneously. The usage of EBS integrated logging capabilities to the project's software. EBS is continuously keeping track of logs as event streams, capturing and analyzing the executed processes effectively. Instructions on how to access the logging system of the application are included as well in the README file of the repository.

To sum up, the deployment of the web application with Elastic Beanstalk provided a scalable, robust, and streamlined infrastructure. By leveraging Elastic Beanstalk's features and integration with other AWS services, the project could achieve efficient deployment, scalability, and high availability while ensuring a consistent and reliable user experience.

Task Allocation and Time Management Tool

Project management is a key to ensuring that a project runs efficiently within an allocated time frame, budget while ensuring that the goals are met and a working product is delivered.

Delivering the working product is crucial but it is not the only aspect of a project that we need to consider ensuring that we meet our target. Task management, time management and resource management are also key.

As soon as the project team was assigned, we initialized the project management which coincidently matches with the 12-step cycle. The 1st step was brainstorming and defining the project scope and functionalities. It is in this step that we were able to identify the list of tasks and estimate the resources and time required to meet the project goals.

With a team of 5 resources, it was crucial to define a working structure that enabled definition and assignment of different tasks to be carried out in parallel as smaller independent project units to be later integrated, forming the final functional project. We decided to encompass the Agile Framework following the values defined in the Manifesto and the 12 principles behind it. This approach is centered around incremental and iterative steps to ensure that each simple unit works properly on its own, one could easily back track, modify each one separately, and generate the new desired project look. Secondly, unlike other approaches in the market this approach is focused on how team members work together. Lastly, the approach is driven toward quick delivery, adapting to change and collaboration.

Currently there are different tools such as Github issues, Asana, Jira, Monday.com, and Microsoft Planner, which offer similar project management and collaboration features. For this specific project and our task allocation needs we decided to use **Trello**[23] a web-based project management and collaboration tool. Trello has a simple to use interface and the ability for easy integration with other tools. It also offered flexibility on how to manage and organize our tasks once added to the board. The collaboration on the tool was also extensive where it allowed assignment of tasks to specific users as well as an interactive space where comments were shared on the progress of the tasks and any issues or insights gained.

In the picture below we list an example of a board defined at a specific stage of the project, representing the flow of the tasks from To Do to Done. Each member of the team is assigned to a certain task, and has the overview of the entire project task allocation.
Additionally, one can add notes, comments etc.. referring to useful links, or doubts to be answered.

[23] https://trello.com/

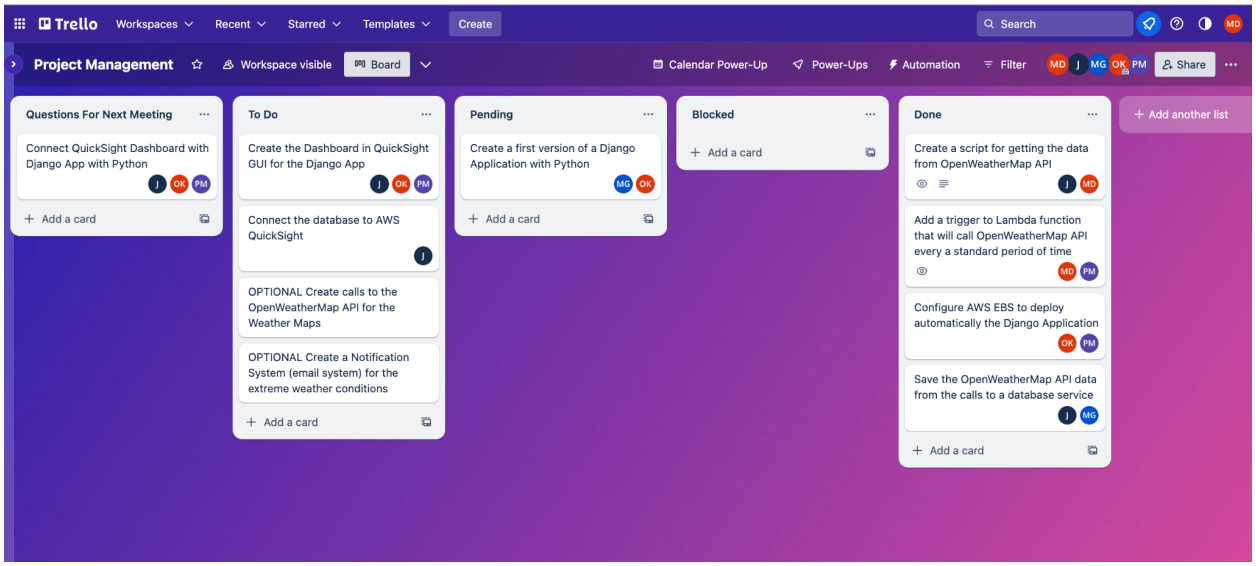

Figure 16. Task allocation and time management of the project

What is specifically useful when using tools like Trello is that each movement of the task is time recorded, giving us the overview of the duration of time consumed till completion therefore, a measure of time to transition from one project stage to the next.

Throughout the project implementation, we tried to follow most of the tasks as initially defined, changing the order when necessary. Duration of each task depends on the complexity and the challenges that arise and require more effort.

Final Listing of the Hours Invested

As observed during the project proposal and definition, setting up a schedule is easy but adhering to it can prove challenging as the project proceeds.

We utilized the Gantt chart to prioritize the tasks and also redefine the project schedule. It allows for the assignment of tasks in an overall overview therefore giving insight in what needs doing taking into account the time to completion.

Please see the Gantt Chart below with the hours invested in the implementation of the project.

			03/04/2023	10/04/2023	17/04/2023	24/04/2023	01/05/2023	08/05/2023	15/05/2023	22/05/2023	
	Proposed hours	Names	1	2	3	4	5	6	7	8	Final hours
		Miona	2	2							4
		Poly	1	1							2
Scope and functionality definition	0	Joan	1	1							2
		Joan	1								1
Capacity planning	0	Poly								2	2
		Odysseas	2								2
Writing project proposal	0	Joan	1								1
		Poly			1		1				2
		Odysseas	1	1	1		1				4
Identify the main components of the application	5	Joan					1				1
		Odysseas							1	1	2
		Miona		2						1	3
Document the services and resources that will be used	3	Joan						1			1
		Miona		2		1					3
		Odysseas	1	1							2
Research the OpenWeatherMap API and its capabilities	6	Joan			2						2
Document the data that can be obtained from the API	3	Miona			2	2				1	5
		Odysseas	1	1	1	1	1				5
		Miona			2				1		3
		Joan						1			1
Research AWS Cloud Services and their capabilities	8	Mateusz						2			2
		Miona			2	2					4
Integrate the OpenWeatherMap API	10	Joan					2				2
		Odysseas					3				3
Implement the back-end of the application using Django	30	Mateusz					5				5
		Odysseas	1	1	1			1	1		5
Decide on the appropriate AWS services to use	4	Mateusz				1	1	2			4
		Miona		1		1					2
Define the interactions between the components	7	Joan							1		1
		Joan					1		2	1	4
		Odysseas							1		1
Define and implement data transformations	20	Poly							3		3
Design the database schema	4	Joan					1		2		3
		Joan						2	2	2	6
Integration of AWS Cloud Services	12	Odysseas							1	1	2
		Joan							2	2	4
Deployment on AWS Cloud Services	16	Odysseas						1	1	1	3
		Miona			3			1	2		6
		Odysseas					1	1			2
Best practices to ensure easy maintenance	5	Poly		1		1			1		3
		Joan					1				1
Applications designed as stateless processes	4	Odysseas					1	1			2
Conducting testing on the application to ensure functionality and performance	12	Poly					1	1	1	1	4
		Odysseas						1	1		2
Performing quality assurance checks	8	poly		1				1	1		3
		Odysseas	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5	4
		Miona	1			0,5		1	0,5		3
Communication with team members regarding tasks and responsibilities	6	Poly	0,5		0,5	0,5	0,5	0,5	0,5	0,5	3,5
		Mateusz		0,5			0,5		0,5		1,5
Written documentation and logs implementation	9										0
		Odysseas								5	5
		Miona								3	3
		Joan								2	2
		Poly								4	4
Writing the final report	15	Mateusz							2	3	3
Implement the dashboard for weather analytics	8	Mateusz							2	4	6

A contingency plan was put in place to ensure that we could deal with any issues that we encountered. This entailed the addition of extra hours that went into the codebase to ensure that we were ready to deal with any errors and crashes in the project.

Hindrances:
During the implementation of the codebase, we encountered several integration issues related to the permission issued in the development environment. This meant that at least 4 resources spent 12 hrs each of the time allocated to the project searching for alternatives. This threatened the completion and delivery of the project.

Due to poor planning at the beginning of the project most of the actual development was left to later development which meant a rush through to completion.