SAPIENZA
UNIVERSITÀ DI ROMA

Department of Computer Science

Project Report

Cloud Computing

Professors:
Emiliano Casalicchio

Students:
Barreca Federico - 1736423
Bevilacqua Paolo Pio - 2002288
De Sio Ilaria - 2064970

Academic Year 2022/2023

1 Description of the problem addressed

The proposed project called "*SoundFlow*" aims to offer song recommendations from Spotify associated with emotions by a text inserted by the user.

Once the user-entered text is classified into an emotion, it will be assigned a floating-point value between 0 and 1. This value will serve as the basis for recommending a song from the Spotify dataset. The association between emotions and songs will be established using these three fields:

1. *Valence*, which represents the degree of similarity and positivity of a song (Spotify automatically provides the valence value for each song);
2. *Energy*, which is a perceptual measure of intensity and activity. Generally, energy tracks are fast, loud and noisy, ranging from measure 0.0 to 1.0;
3. *Tempo*, that is the overall estimated tempo of a track in beats per minute (BPM). In musical terminology, tempo is the speed or pace of a given piece and derives directly from the average beat duration.

We decided to use Spotify's API, which provides a music player connected to the Spotify platform, allowing users with premium accounts to listen to the suggested song in its entirety.

The application uses many AWS services to process the pipeline, read songs by API, and elaborate the prediction and process it by making it available to the end user that uses the system.

2 Design of the solution

Starting from the description of the addressed problem, we have developed the architecture of the system proposed (See the Figure 1).

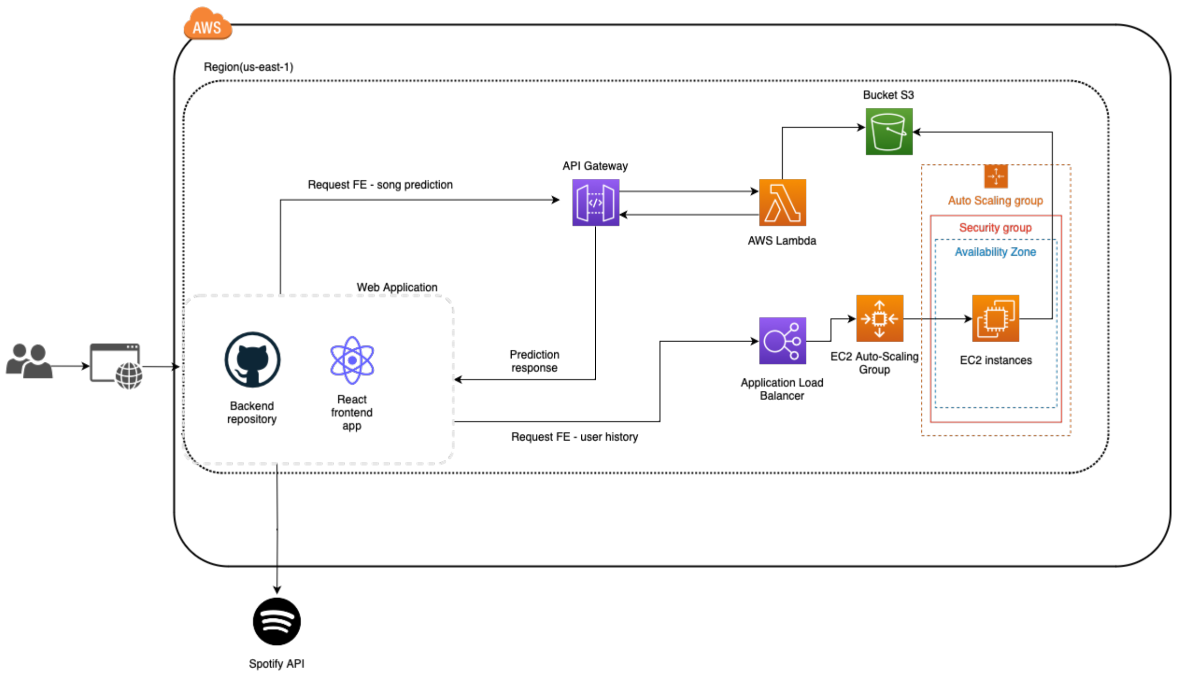


Figure 1: Cloud architecture diagram

3 Description of the Implementation of the solution

For the realization of our project, we made use of different technologies that led us to achieve our goal. Technologies used in our project are:

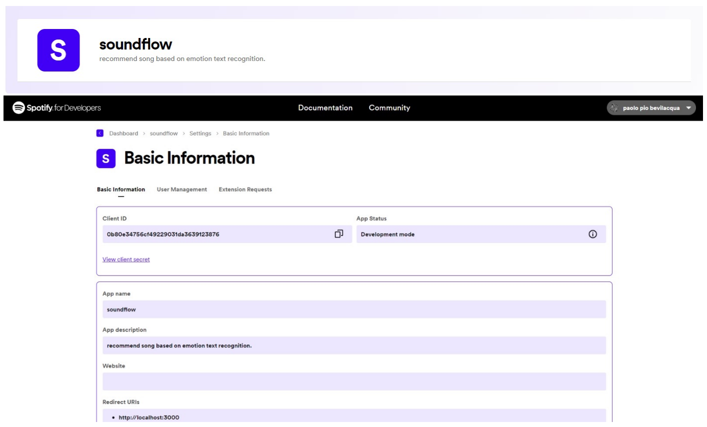
- *Spotify API* : In this project, the Spotify API allowed us to integrate directly within the Web application a player containing the suggested song, and since it is also connected to the user's premium account it also allows him to save it directly to his favorite songs in the Spotify library. This integration will allow users to seamlessly listen to the suggested music within the familiar Spotify environment but within the Web Application.
- *AWS EC2*: we chose EC2 because it is a flexible computing service that allows us to run virtual machines with different hardware configurations. We configure it to deploy the front-end application and a simple functionality about reading data from the bucket.
- *Amazon API Gateway*: this service in the project was useful to us because it simplifies the creation, publication, maintenance, monitoring, and protection of APIs at any scale, in fact, it allowed us to manage HTTP request and response requests.
- *AWS Lambda*: we required this service to seamlessly integrate the code logic, enabling us to utilize the machine learning model within the AWS environment. This integration facilitates the prediction of scores for each individual song.
- *Amazon S3*: we created an S3 bucket so that we could put inside it the dataset containing all the Spotify songs, with all the specifications related to it, but in this case, the most relevant was the track-id, and the valence used in combination with the energy and tempo of the song itself.
- *Amazon Elastic Load Balancing*: intelligently distributes incoming request traffic across application instances. This ensures the continuous availability of applications,
- *Amazon EC2 Auto-Scaling*: allows us to use the automatic scaling of computing resources, such as EC2 instances, based on demand. This ensures that applications remain available and responsive during traffic spikes.
- *Amazon AWS Cloudwatch*: is a resource monitoring and management service, and it has been useful for monitoring resource performance and has allowed us to organize and view metrics and alerts to ensure system reliability.

4 Description of the deployment of your solution

Starting with the description phase, we will go in more detail by showing the implementation of the various components.

4.1 Step 1: Spotify API

In order to use the Spotify API, the first step is to register on the Spotify Developers platform (`https://developer.spotify.com`) and create a new App in order to obtain valid credentials to develop our application.

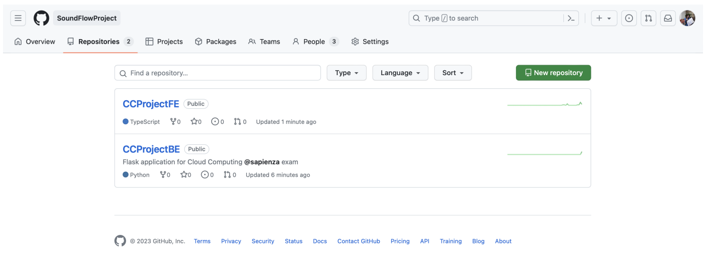

soundflow
recommend song based on emotion text recognition.

Client ID: 0b80e34756cf49229031da3639123876
App Status: Development mode
View client secret
App name: soundflow
App description: recommend song based on emotion text recognition.
Website:
Redirect URIs:
- http://localhost:3000

Figure 2: Settings on Spotify Developers

4.2 Step 2: Versioning and S3 buckets

In the next step, we decided to create an organization on Git Hub, a structure that allows you to manage and coordinate projects and software repositories with a group of people efficiently. Respectively creating a repository for the Backend in Flask, and one for the Frontend in React (See the Figure 3).



CCProjectFE Public
TypeScript 0 0 0 0 Updated 1 minute ago

CCProjectBE Public
Flask application for Cloud Computing @sapienza exam
Python 0 0 0 0 Updated 6 minutes ago

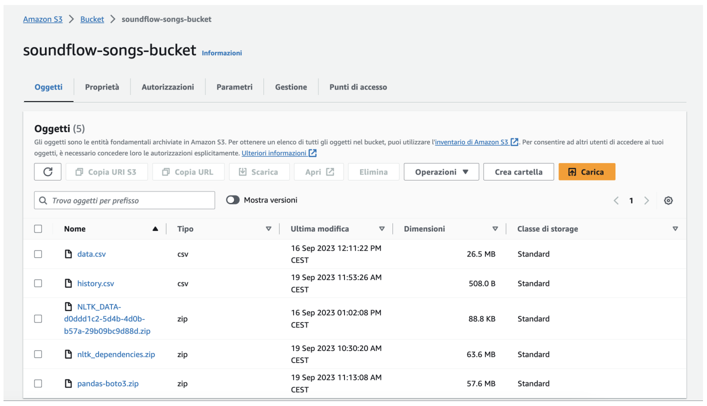
© 2023 GitHub, Inc. Terms Privacy Security Status Docs Contact GitHub Pricing API Training Blog About

Figure 3: Settings of organization GitHub

Next, we proceeded to establish an Amazon S3 bucket (See the Figure 4) containing several files:

- *Data.csv*: containing Spotify song data previously acquired from Kaggle (`https://www.kaggle.com/datasets/vatsalmavani/spotify-dataset`). This data includes essential information about the songs, including track-id, valence, tempo, energy, and other relevant attributes;
- *History.csv*: created by us containing user navigation data about the text, song prediction, and date;
- *Nltk dependencies*: containing the NLTK library used later in the prediction of the lambda;
- *Pandas-Boto3*: is a Python package that allows integration between the Pandas(for data management and manipulation) and Boto3 libraries (allows connection with AWS storage services).

One of the key steps of this project was the development of the Lambda function get_emotion_score, as you can see in the figs. 5a and 5b, layers were inserted, they are a feature that allows to separate and manage in a modular way the dependencies and external resources needed, in this case for space problems they were inserted separately.

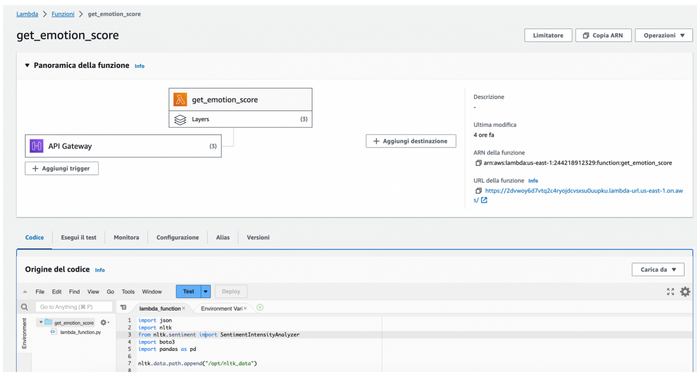


Oggetti (5)

Gli oggetti sono le entità fondamentali archiviate in Amazon S3. Per ottenere un elenco di tutti gli oggetti nel bucket, puoi utilizzare l'inventario di Amazon S3. Per consentire ad altri utenti di accedere ai tuoi oggetti, è necessario concedere loro le autorizzazioni esplicitamente. Ulteriori informazioni

Nome	Tipo	Ultima modifica	Dimensioni	Classe di storage
data.csv	csv	16 Sep 2023 12:11:22 PM CEST	26.5 MB	Standard
history.csv	csv	19 Sep 2023 11:53:26 AM CEST	508.0 B	Standard
NLTK_DATA-d0ddd1c2-5d4b-4d0b-b57a-29b09bc9d88d.zip	zip	16 Sep 2023 01:02:08 PM CEST	88.8 KB	Standard
nltk_dependencies.zip	zip	19 Sep 2023 10:30:20 AM CEST	63.6 MB	Standard
pandas-boto3.zip	zip	19 Sep 2023 11:13:08 AM CEST	57.6 MB	Standard

Figure 4: S3 Bucket



Descrizione
-

Ultima modifica
4 ore fa

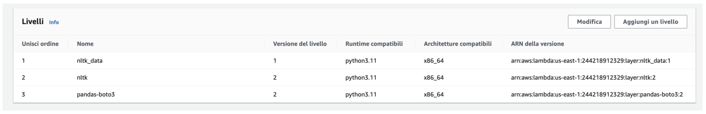
ARN della funzione
arn:aws:lambda:us-east-1:244218912329:function:get_emotion_score

URL della funzione Info
https://2dvwoy6d7vtq2c4ryojdcvsxsu0uupku.lambda-url.us-east-1.on.aws/

```python
import json
import nltk
from nltk.sentiment import SentimentIntensityAnalyzer
import boto3
import pandas as pd

nltk.data.path.append("/opt/nltk_data")
```

(a) Interface of the Lambda function



Unisci ordine	Nome	Versione del livello	Runtime compatibili	Architetture compatibili	ARN della versione
1	nltk_data	1	python3.11	x86_64	arn:aws:lambda:us-east-1:244218912329:layer:nltk_data:1
2	nltk	2	python3.11	x86_64	arn:aws:lambda:us-east-1:244218912329:layer:nltk:2
3	pandas-boto3	2	python3.11	x86_64	arn:aws:lambda:us-east-1:244218912329:layer:pandas-boto3:2

(b) Lambda layers

4.3 Step 3: Lambda function

After setting up all the dependencies we needed, we moved on to implementing the lambda function code. One of the initial steps involves the utilization of NLTK's **SentimentIntensityAnalyzer**, a tool designed for the analysis of sentiment within textual content. This instrument relies upon a lexicon containing words assigned with positive and negative sentiment scores, which it employs to assess the text provided as input. Its output includes a **polarity score** (indicating whether the sentiment is positive, negative, or neutral) and a **sentiment intensity score**. These scores are notably beneficial when integrated with valence and other influential factors during the process of emotion analysis.
Some variables are configured externally to the Lambda function, such as retrieving the song dataset from the S3 bucket. This approach is aimed at performing particularly onerous operations during the start-up (warm-up) phase in order to reduce latency.

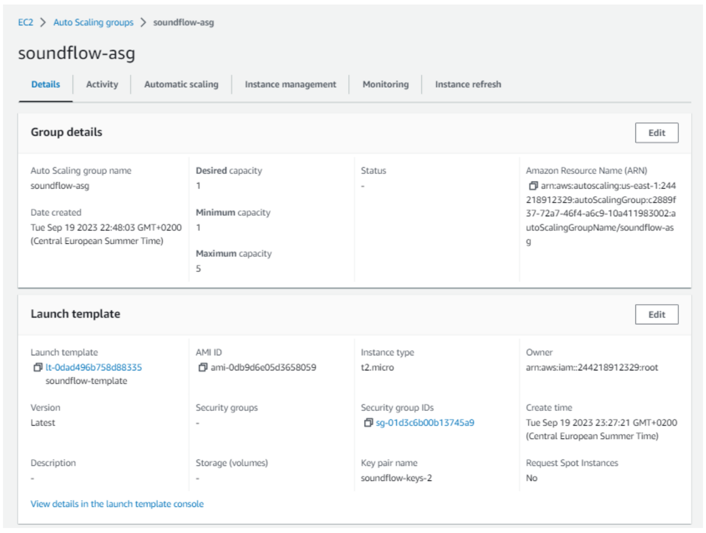
Subsequently, we proceed with the acquisition of the text provided by the user, taking into account the scenario in which no text is provided. We then calculate the polarity score based on the input text, followed by normalization to work on a common scale with valence. Once these operations are completed, we obtain a data frame containing all the songs with the same obtained score, falling within a defined range as described below

$$[normalized\_score - increment, normalized\_score + increment]$$

while avoiding scenarios where there are no songs with that score, and as the last step the song closest to the assigned valence is selected.

4.4 Step 4: Auto-Scaling group

In order to facilitate interconnection and optimized resource management, we implemented an Auto-Scaling Group. The Auto-Scaling Group is configured to have a minimum capacity of 1 a maximum capacity of 5, and a desired capacity of 1.



soundflow-asg

Group details

Auto Scaling group name: soundflow-asg	Desired capacity: 1	Status: -	Amazon Resource Name (ARN): arn:aws:autoscaling:us-east-1:244218912329:autoScalingGroup:c2889f37-72a7-46f4-a6c9-10a411983002:autoScalingGroupName/soundflow-asg
Date created: Tue Sep 19 2023 22:48:03 GMT+0200 (Central European Summer Time)	Minimum capacity: 1		
	Maximum capacity: 5		

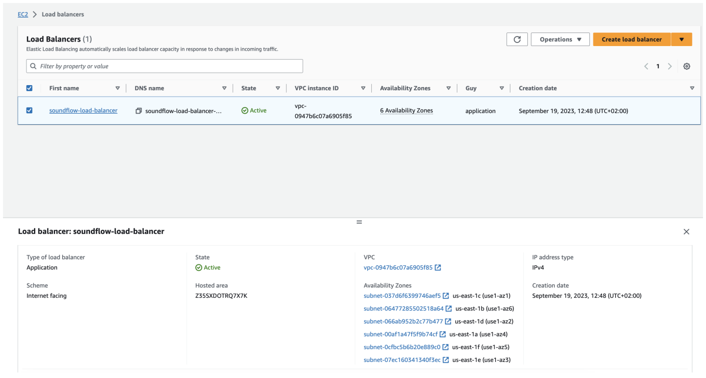
Launch template

Launch template: lt-0dad496b758d88335 soundflow-template	AMI ID: ami-0db9d6e05d3658059	Instance type: t2.micro	Owner: arn:aws:iam::244218912329:root
Version: Latest	Security groups: -	Security group IDs: sg-01d3c6b00b13745a9	Create time: Tue Sep 19 2023 23:27:21 GMT+0200 (Central European Summer Time)
Description: -	Storage (volumes): -	Key pair name: soundflow-keys-2	Request Spot Instances: No

View details in the launch template console

Figure 6: Setting of Auto-Scaling group

This group was configured to be connected to the Application Load Balancer, thus providing the user with a single point of access to the application. If traffic increases, the system will self-scale using auto-scaling, thus ensuring continuous service availability and optimal performance.



Load Balancers (1)
Elastic Load Balancing automatically scales load balancer capacity in response to changes in incoming traffic.

First name	DNS name	State	VPC instance ID	Availability Zones	Guy	Creation date
soundflow-load-balancer	soundflow-load-balancer-...	Active	vpc-0947b6c07a6905f85	6 Availability Zones	application	September 19, 2023, 12:48 (UTC+02:00)

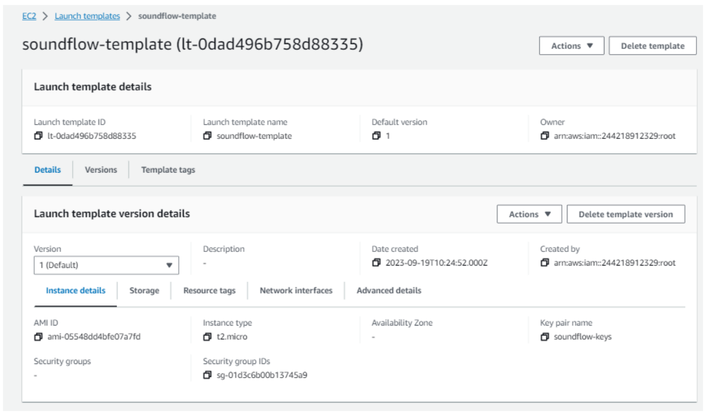
Load balancer: soundflow-load-balancer

Type of load balancer: Application	State: Active	VPC: vpc-0947b6c07a6905f85	IP address type: IPv4
Scheme: Internet facing	Hosted area: Z35SXDOTRQ7X7K	Availability Zones: subnet-037d6f6399746aef5 us-east-1c (use1-az1); subnet-06477285502518a64 us-east-1b (use1-az6); subnet-066ab952b2c77b477 us-east-1d (use1-az2); subnet-00af1a47f5f9b74cf us-east-1a (use1-az4); subnet-0cfbc5b6b20e889c0 us-east-1f (use1-az5); subnet-07ec160341340f3ec us-east-1e (use1-az3)	Creation date: September 19, 2023, 12:48 (UTC+02:00)

Figure 7: Setting of Elastic Load Balancer

To initiate the autoscaling group, a Launch template was created, in which various parameters were set, including the AMI (Amazon Machine Image) from which new EC2 instances associated with the autoscaling group should be created.

The used AMI was configured from a separate instance onto which the repositories for the React front-end and Flask back-end were cloned, and associated with the Apache and Gunicorn services,



EC2 > Launch templates > soundflow-template

soundflow-template (lt-0dad496b758d88335)

Launch template details

Launch template ID	Launch template name	Default version	Owner
lt-0dad496b758d88335	soundflow-template	1	arn:aws:iam::244218912329:root

Launch template version details

Version	Description	Date created	Created by
1 (Default)	-	2023-09-19T10:24:52.000Z	arn:aws:iam::244218912329:root

AMI ID	Instance type	Availability Zone	Key pair name
ami-05548dd4bfe07a7fd	t2.micro	-	soundflow-keys

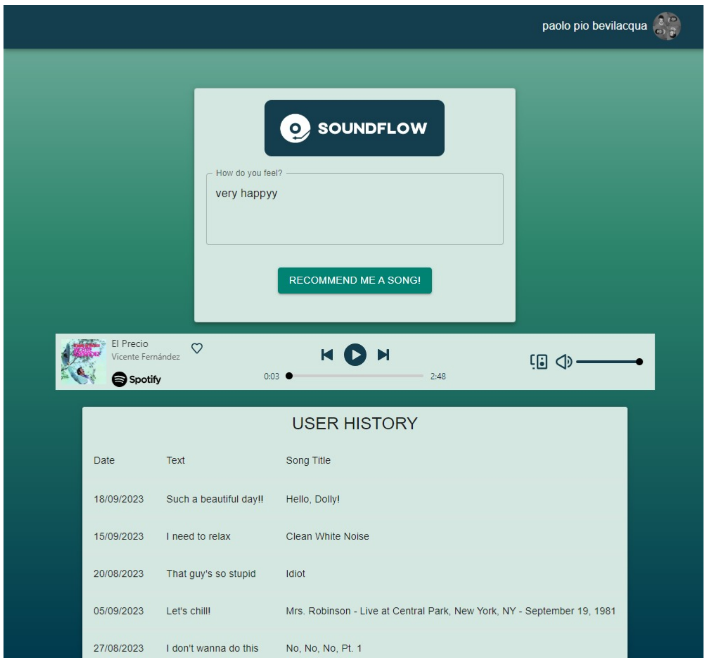
Security groups	Security group IDs
-	sg-01d3c6b00b13745a9

Figure 8: Setting of Launch Template

respectively.
These two services were configured to serve the front end (port 443 with TLS connection set using a self-signed certificate) and the back end (port 8080). The autoscaling group was linked to a load balancer, on which two target groups were configured for the two exposed services. A rule, similar to a proxy, was also configured for the back end because the call to the history service (/home) arrived on port 443 and was then internally redirected. In the Launch template of the autoscaling group, a service role was also configured on the EC2 instances to allow the new instances to access S3 buckets.

4.5 Final result

The final result of the application is shown in the figure below (See Figure 9). Two main functionalities are observed, the first one concerns the real-time Spotify player, directly linked to the user's premium Spotify profile. This allows the user to save all generated songs in their library, accessible from any device connected. The second functionality is the user's history, where each user can view the list of the latest generated songs, including the date and the text entered at the time of generation.



very happyy

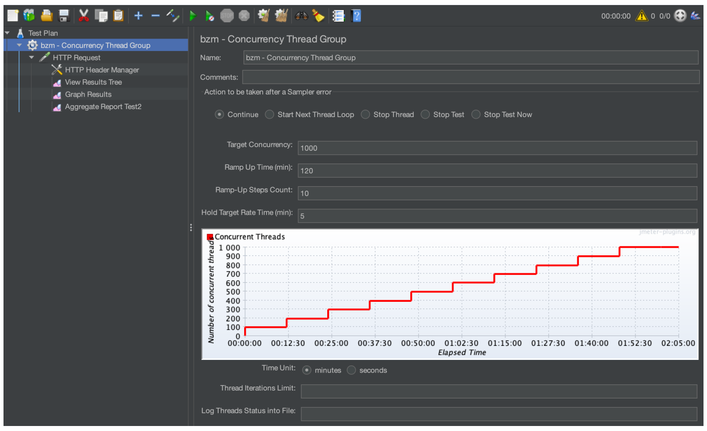
USER HISTORY

Date	Text	Song Title
18/09/2023	Such a beautiful day!!	Hello, Dolly!
15/09/2023	I need to relax	Clean White Noise
20/08/2023	That guy's so stupid	Idiot
05/09/2023	Let's chill!	Mrs. Robinson - Live at Central Park, New York, NY - September 19, 1981
27/08/2023	I don't wanna do this	No, No, No, Pt. 1

Figure 9: Interface of the application

5 Experimental design

Regarding testing, we have chosen to conduct tests using the JMeter software.
Below, you may find the configuration details. (See the Figure 10). As evident, we have employed a



Test Plan
- bzm - Concurrency Thread Group
  - HTTP Request
    - HTTP Header Manager
    - View Results Tree
    - Graph Results
    - Aggregate Report Test2

bzm - Concurrency Thread Group

Name: bzm - Concurrency Thread Group

Comments:

Action to be taken after a Sampler error: [✓ Continue] Start Next Thread Loop / Stop Thread / Stop Test / Stop Test Now

Target Concurrency: 1000

Ramp Up Time (min): 120

Ramp-Up Steps Count: 10

Hold Target Rate Time (min): 5

Time Unit: [✓ minutes] seconds

Thread Iterations Limit:

Log Threads Status into File:

Figure 10: JMeter test configuration

JMeter plugin referred to as the "**Concurrency Thread Group**", this plugin offers a streamlined method for configuring thread scheduling, with the primary objective of preserving concurrency levels. This entails the initiation of supplementary threads during runtime if an insufficient number of threads are concurrently active. In contrast to the conventional thread group, this approach refrains from pre-allocating all threads, thereby avoiding any superfluous memory consumption.
The main fields of our configuration are as follows :

- *Target Concurrency*: refers to the desired number of concurrent users or threads that should be active during the test, we set this to 1000, and in other tests try to 5000.
- *Ramp-Up Time*: means the duration over which new threads are gradually started until reaching the target concurrency level, in this test we decided to set it to 120 minutes i.e. 2 hours, imagining a real use case.
- *Ramp-Up Steps Count*: denotes the number of incremental steps in which new threads are introduced during the ramp-up phase, in this case, is set to 10.
- *Hold Target Rate Time*: represents the duration for which the test should maintain the specified target concurrency level once it's reached, enabling the evaluation of system performance under steady-load conditions, in this case, is set to 5 minutes.

In addition, the other components are *HTTP Request* used to send HTTP or HTTPS requests to a web server as part of a load test, and the corresponding *HTTP Header Manager* used to manage headers during the test.

6 Experimental results

6.1 Load Tests on Lambda service

The metrics chosen for the load tests to be performed on the Lambda function called **get_emotion_score** are:

- *Invocations*: represents the count of times a Lambda function is triggered or called in response to an event or request. This metric provides a measure of how often the Lambda function is activated, which is useful for assessing the demand for the service. (See the Figure 11).

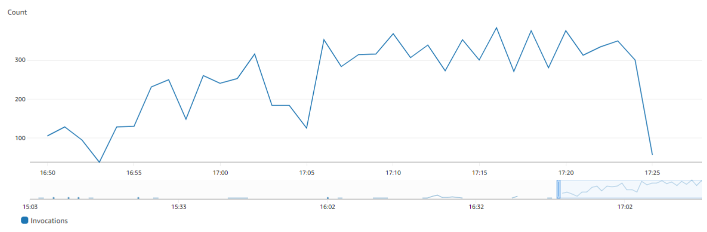


Figure 11: Graph on CloudWatch of Lambda invocations

- *Duration*: measures the time, in milliseconds, it takes for a Lambda function to execute its code logic in response to an invocation. This metric quantifies how long it takes to execute the prediction function and can be crucial to ensure that application performance meets requirements. (See the Figure 12).
  The graph shows spikes at 300,000 ms (5 minutes), which corresponds to the timeout set for the maximum execution time of the Lambda function. These points on the time axis, when compared to the success rate graph, indeed correspond to errors. The analysis through CloudWatch was conducted retrospectively, which did not allow for filtering out failed invocations to obtain a consistent response time graph.
  Through simple invocation tests using the API Gateway test tool and also by checking the minimum response time on this graph, we noticed that it takes only a few milliseconds to respond once the warm-up period has been surpassed. Failed invocations, which therefore register a duration of 5 minutes, may correspond to moments when the Lambda function was in a warm-up phase (in the case of this function, it took time to download objects from S3) or due to sudden load fluctuations.
  To avoid this situation, a possible solution could be to increase the memory available to the Lambda function.

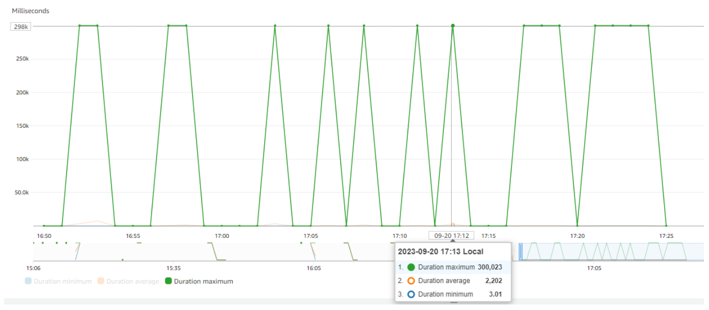


Figure 12: Graph on CloudWatch of Lambda requests duration

- *Error count and success rate*: These metrics provide an indication of the stability and reliability of Lambda functions. (See the Figure 13).
  As we can observe, the success rate closely follows the pattern of the invocation graph.

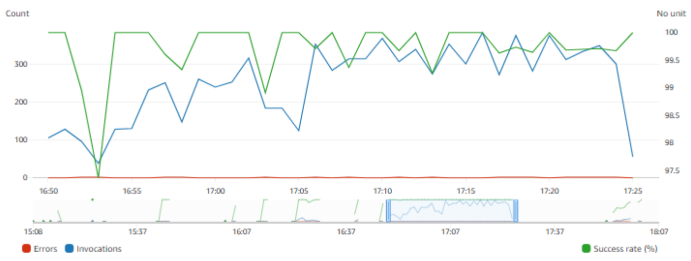


Figure 13: Graph on CloudWatch of error count and success rate

- *Concurrent executions*: represent the maximum number of instances of a Lambda function that are active simultaneously at a given moment. This metric is important to understand the capacity of AWS Lambda to handle concurrent workloads. We set the focus only on the maximum number of concurrent executions. (See the Figure 14). Looking at this graph, it is possible to see that the total number of concurrent executions is fairly constant, and, more importantly, it lets us know that our system holds up well to a large number of concurrent requests.

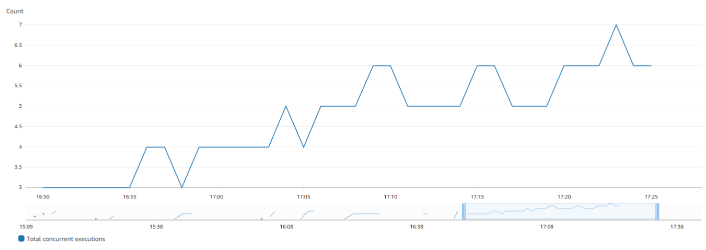
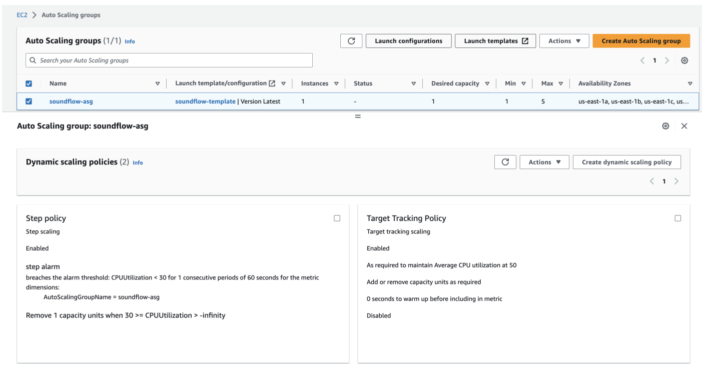


Figure 14: Graph on CloudWatch of Lambda concurrent executions

6.2 Testing the EC2 service

The AWS EC2 instances have been configured to deploy the front-end application and the user's browsing history retrieval function, returning them to the front end. The chosen instance types are t2.micro, which do not possess high computational capacity. For this reason, we decided to enhance the application's performance and scalability by setting up an auto-scaling group associated with a load balancer. This way, the application's scalability remains transparent to the end user. The testing phase of the auto-scaling group involves tracking the average CPU utilization as the load varies using testing tools, as well as monitoring the number of instances. Thanks to the policies (See Figure 15) configured on the auto-scaling group, instances are added or removed based on traffic, thus avoiding unnecessary resource consumption.

The autoscaling group tests were performed by invoking the '/home' service exposed by the Flask backend, which consumes more resources compared to the front end. This allowed for a better observation of the autoscaling behavior under varying loads.
The CPU utilization and response time tests under varying workloads were conducted using AWS Distributed Load Testing (`https://aws.amazon.com/solutions/implementations/distributed-load-testing-on-aws/`).


EC2 > Auto Scaling groups
Auto Scaling groups (1/1) Info
Launch configurations
Launch templates
Actions
Create Auto Scaling group
Search your Auto Scaling groups

Name	Launch template/configuration	Instances	Status	Desired capacity	Min	Max	Availability Zones
soundflow-asg	soundflow-template | Version Latest	1	-	1	1	5	us-east-1a, us-east-1b, us-east-1c, us…

Auto Scaling group: soundflow-asg

Dynamic scaling policies (2) Info

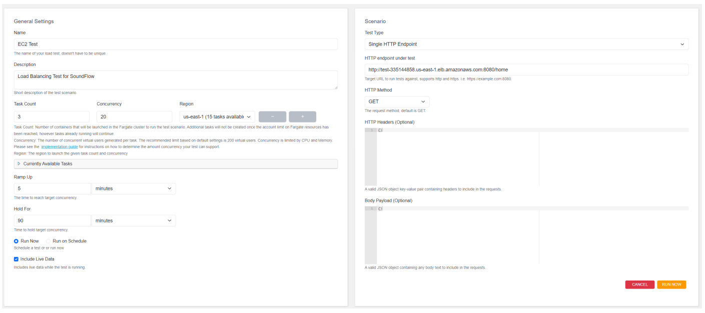
Actions
Create dynamic scaling policy

Step policy
Step scaling
Enabled
step alarm
breaches the alarm threshold: CPUUtilization < 30 for 1 consecutive periods of 60 seconds for the metric dimensions:
AutoScalingGroupName = soundflow-asg
Remove 1 capacity units when 30 >= CPUUtilization > -infinity

Target Tracking Policy
Target tracking scaling
Enabled
As required to maintain Average CPU utilization at 50
Add or remove capacity units as required
0 seconds to warm up before including in metric
Disabled

Figure 15: Dynamic scaling policies

Three tasks were imported, each with 20 virtual users, with the intention of simulating simultaneous and continuous traffic. The test lasted for 9 minutes with a 5-minute ramp-up time to reach a steady state.


General Settings
Name: EC2 Test
The name of your load test, doesn't have to be unique.
Description: Load Balancing Test for SoundFlow
Short description of the test scenario.
Task Count: 3
Concurrency: 20
Region: us-east-1 (15 tasks availabl
Task Count: Number of containers that will be launched in the Fargate cluster to run the test scenario. Additional tasks will not be created once the account limit on Fargate resources has been reached, however tasks already running will continue.
Concurrency: The number of concurrent virtual users generated per task. The recommended limit based on default settings is 200 virtual users. Concurrency is limited by CPU and Memory. Please see the implementation guide for instructions on how to determine the amount concurrency your test can support.
Region: The region to launch the given task count and concurrency
Currently Available Tasks
Ramp Up: 5 minutes
The time to reach target concurrency.
Hold For: 90 minutes
Time to hold target concurrency.
[✓ Run Now] Run on Schedule
Schedule a test or or run now
[✓ Include Live Data]
Includes live data while the test is running.

Scenario
Test Type: Single HTTP Endpoint
HTTP endpoint under test: http://test-335144858.us-east-1.elb.amazonaws.com:8080/home
Target URL to run tests against, supports http and https. i.e. https://example.com:8080.
HTTP Method: GET
The request method, default is GET.
HTTP Headers (Optional)
```
{}
```
A valid JSON object key-value pair containing headers to include in the requests.
Body Payload (Optional)
```
{}
```
A valid JSON object containing any body text to include in the requests.

Figure 16: Auto-Scaling group test configuration

The metrics chosen for the Distributed Load Testing are:

- *CPU Usage (per instance)*: In the CPU utilization graph, it can be observed that an increase in workload on the autoscaling group triggers the launch of new instances to distribute the load. CPU usage remains high and consistent even on the new instances, which are terminated as soon as the workload decreases.

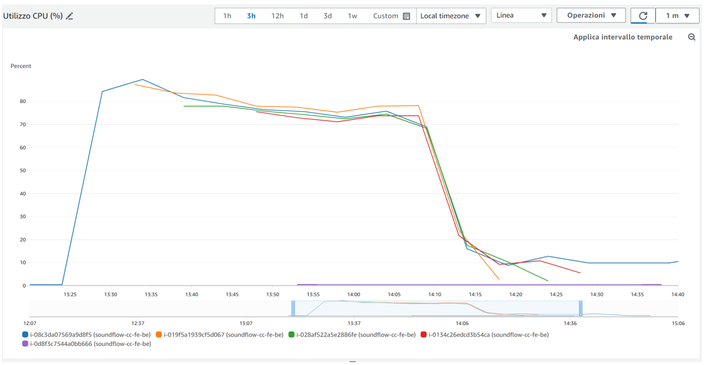


Figure 17: CPU Usage for each instance

- *Auto-Scaling group*: As the workload increases, the number of instances in the group also increases. The system performs well as the 'In Service' capacity of instances overlaps with the desired capacity. With varying workloads, the number of instances adjusts to provide computational capacity that preserves service availability and stability.

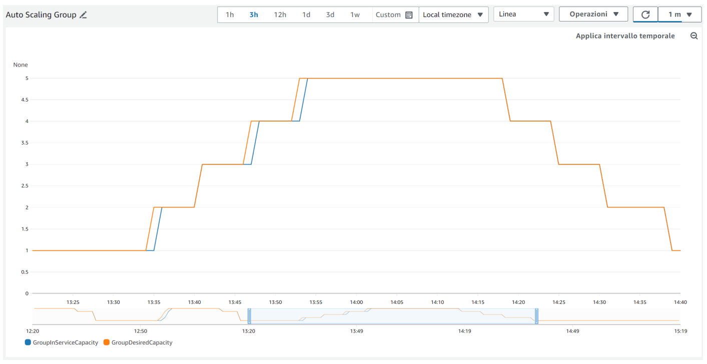


Figure 18: Desired Group compared to In Service Group

- *Distributed Load Testing*: as the number of requests increases, the success rate remains high. It is possible to observe the load increase ramps and a very low number of errors.

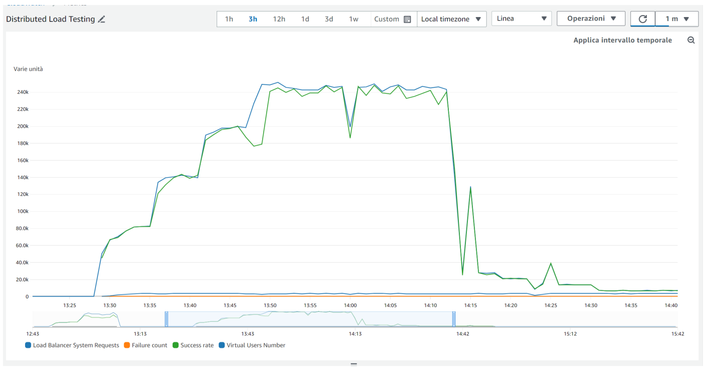


Figure 19: Number of the Requests, Failure count, Success rate and Virtual Users number

- *Target Response Time*: even the response time remains low and consistent over time. This aligns with one of the cloud computing objectives, which is to make the stress caused by an increase in workload transparent to every user, providing the impression of having an almost infinite amount of resources available. Towards the end, however, the response time increases, almost doubling. This behavior coincides with the termination of instances in the group, resulting in a decrease in available resources, which could be a possible cause. This is a point that requires further testing and perhaps the implementation of policies such as increasing the minimum capacity to prevent this behavior.

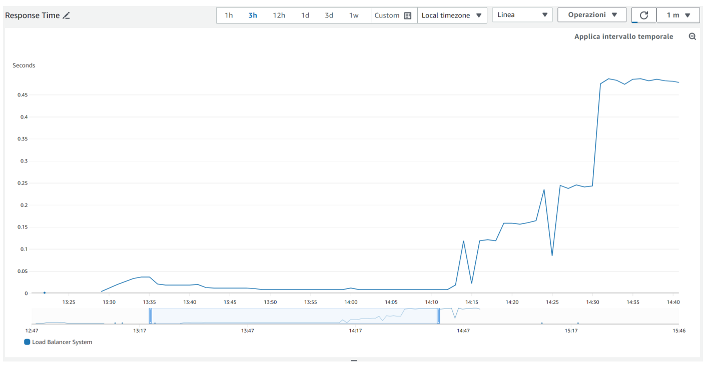
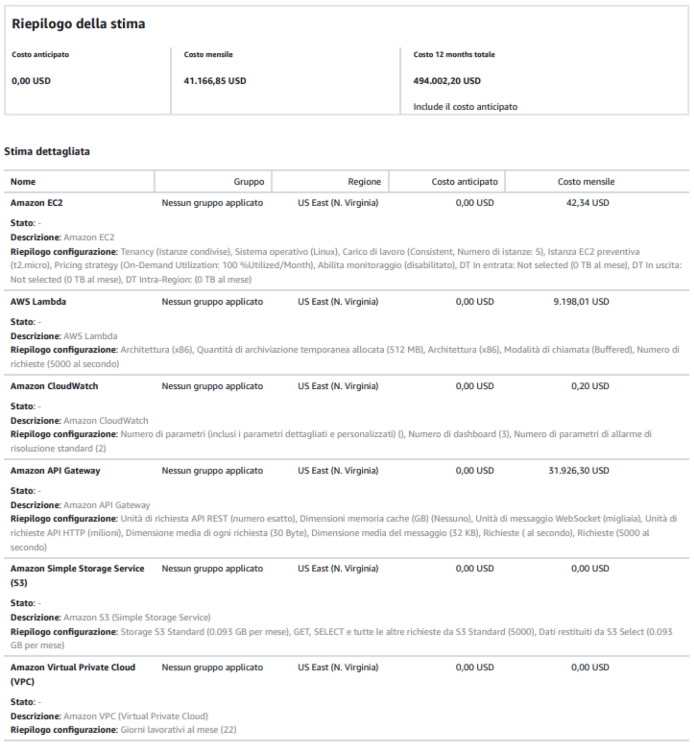


Figure 20: Response time of Load Balancer System

6.3 Cloud Cost Report

The cost analysis conducted through the AWS Pricing Calculator lets us compute the estimated prices associated with our architecture. The total cost is strictly related to the services utilized, a characteristic that renders the selection of cloud technology significantly more cost-effective than the traditional on-premise solution. It was particularly observed that opting for an on-demand EC2



Riepilogo della stima

Costo anticipato	Costo mensile	Costo 12 months totale
0,00 USD	41.166,85 USD	494.002,20 USD Include il costo anticipato

Stima dettagliata

Nome	Gruppo	Regione	Costo anticipato	Costo mensile
Amazon EC2	Nessun gruppo applicato	US East (N. Virginia)	0,00 USD	42,34 USD
Stato: - **Descrizione:** Amazon EC2 **Riepilogo configurazione:** Tenancy (Istanze condivise), Sistema operativo (Linux), Carico di lavoro (Consistent, Numero di istanze: 5), Istanza EC2 preventiva (t2.micro), Pricing strategy (On-Demand Utilization: 100 %Utilized/Month), Abilita monitoraggio (disabilitato), DT In entrata: Not selected (0 TB al mese), DT In uscita: Not selected (0 TB al mese), DT Intra-Region: (0 TB al mese)				
AWS Lambda	Nessun gruppo applicato	US East (N. Virginia)	0,00 USD	9.198,01 USD
Stato: - **Descrizione:** AWS Lambda **Riepilogo configurazione:** Architettura (x86), Quantità di archiviazione temporanea allocata (512 MB), Architettura (x86), Modalità di chiamata (Buffered), Numero di richieste (5000 al secondo)				
Amazon CloudWatch	Nessun gruppo applicato	US East (N. Virginia)	0,00 USD	0,20 USD
Stato: - **Descrizione:** Amazon CloudWatch **Riepilogo configurazione:** Numero di parametri (inclusi i parametri dettagliati e personalizzati) (), Numero di dashboard (3), Numero di parametri di allarme di risoluzione standard (2)				
Amazon API Gateway	Nessun gruppo applicato	US East (N. Virginia)	0,00 USD	31.926,30 USD
Stato: - **Descrizione:** Amazon API Gateway **Riepilogo configurazione:** Unità di richiesta API REST (numero esatto), Dimensioni memoria cache (GB) (Nessuno), Unità di messaggio WebSocket (migliaia), Unità di richieste API HTTP (milioni), Dimensione media di ogni richiesta (30 Byte), Dimensione media del messaggio (32 KB), Richieste (al secondo), Richieste (5000 al secondo)				
Amazon Simple Storage Service (S3)	Nessun gruppo applicato	US East (N. Virginia)	0,00 USD	0,00 USD
Stato: - **Descrizione:** Amazon S3 (Simple Storage Service) **Riepilogo configurazione:** Storage S3 Standard (0.093 GB per mese), GET, SELECT e tutte le altre richieste da S3 Standard (5000), Dati restituiti da S3 Select (0.093 GB per mese)				
Amazon Virtual Private Cloud (VPC)	Nessun gruppo applicato	US East (N. Virginia)	0,00 USD	0,00 USD
Stato: - **Descrizione:** Amazon VPC (Virtual Private Cloud) **Riepilogo configurazione:** Giorni lavorativi al mese (22)				

Figure 21: Report of estimated costs

pricing model proved more advantageous compared to pre-paid options, allowing us to pay solely for the resources actually utilized, thereby eliminating flat-rate charges.

]

7 Conclusion

The objective of our project was to provide users with the ability to use a website alongside an existing service like Spotify, allowing them to interact and add additional features. In our case, the analysis of user-input text to derive emotion using a machine learning model, and subsequently associating it with a song that has similar vibes to the written text, represents an interesting and useful concept. For this reason, given the popularity of Spotify, we decided to implement a reliable, scalable, and secure solution on AWS.
The project configuration phase focused on AWS's core services, which proved to be highly efficient and stable. Choosing to run the model on a Lambda function in a serverless environment provides us with greater flexibility and ease of implementation. Tests conducted on the Lambda function confirmed our expectations, with low response times and a high success rate.
Deploying the website on an auto-scaling group with a load balancer makes it available, reliable, and secure. It can handle variable and high data flows. AWS services, and cloud applications in general, proved to be a good choice for the proposed project.
Possible future developments may include integrating additional features into the application, such as user mood trend statistics or the option to apply filters to the songs returned by the prediction. Additionally, there could be improvements to the model used to extract emotion from text, which in a cloud environment could be continuously trained based on user feedback following predictions.