Song Recommendation System

CMPE 48A Cloud Computing

Term Project Report

Fall 2022

Altay Acar - 2018400084

Umut Deniz Şener - 2018400225

Introduction

This project's scope is a web application that provides song recommendations to users based on a random song chosen from the available songs in a dataset that we have trained using machine learning algorithms. The frontend of the web application was implemented using ReactJS and most of the styling is implemented with custom CSS classes, while the backend was built using various services provided by Amazon Web Services (AWS) such as SageMaker, Lambda, API Gateway, and EC2.

The song recommendation engine utilizes machine learning techniques to analyze audio features data of the songs and provide recommendations based on the affinity of each song to other songs in the dataset. To achieve this, we trained a recommendation model using SageMaker and deployed it on a SageMaker endpoint.

The web application also provides users a vibrant user interface that features song cards for each recommended song that allows users to access the song's Spotify page for playback or any streaming related action.

Overall, the project demonstrates the power and flexibility of cloud computing in building and deploying a complex web application with advanced functionality such as machine learning algorithms and models. In the following sections, we will describe the design and implementation of our term project in more detail.

Project Description

The goal of our project is to develop a system using cloud services that are provided by a cloud service provider such as AWS and observe the behavior of the system that we have built when it is subjected to heavy usage using the stress testing tools like JMeter. To that extent, our project can be examined under two main parts: architecture and testing.

In the architecture part of our project, we have focused on designing and implementing a cloud-based system using the cloud services provided by AWS as much and optimal as possible. For that we have conducted research on the numerous AWS services available to us within the free credits limitation. Since the system that we have aimed to build would be a web application that will be subject to a large number of user requests simultaneously, we aimed to utilize EC2 for hosting the web application, Bucket to store and manage our dataset, Lambda and API Gateway to handle API requests, and SageMaker to provide machine learning services.

In the testing part of our project, we have used stress testing tools such as JMeter to evaluate the behavior and performance of our cloud-based system under incrementing heavy usage. We will be simulating different scenarios with different levels of load to test the scalability and reliability of our system based on different parameters such as user amount, sample amount, and ramp-up time.
We will be collecting various metrics such as response time, error rate, and average throughput during the testing phase to analyze the behavior of our system. Based on the results of the stress tests, we have made inferences on how our system handles the expected levels of traffic.

We can say that our project aimed to demonstrate the behavior of various cloud services such as machine learning services or serverless computing services for building a scalable and reliable system that can handle a large number of user requests.

Architecture

Our cloud based song recommendation system leverages several Amazon Web Services (AWS) components to provide a seamless user experience. Our project's architecture feature the following main components:

- Data Preprocessing: A Python script in our local system that performs various preprocessing steps (e.g., dropping unnecessary columns, one-hot encoding) on a dataset downloaded from Kaggle, and stores the resulting dataset in an AWS S3 bucket.

- Model Training: An AWS SageMaker notebook instance that trains a K-means clustering model on the preprocessed dataset stored in the AWS S3 bucket and stores the trained model in the same S3 bucket.
- Song Recommendation: An AWS Lambda function that generates a random number between the limits of entry indexes in the trained dataset, retrieves the trained model from the S3 bucket, and uses it to recommend the most similar 8 songs.
- API Endpoint: An AWS API Gateway endpoint that exposes the Lambda function as a publicly accessible API, allowing users to make GET requests to retrieve the recommended songs.
- Web Application: A React-based web application that displays the recommended songs as clickable cards, with album cover images, song names, and artists' names. The web application also provides a button that allows users to trigger the recommendation process by making a GET request to the API endpoint.

Our architecture was developed incrementally, starting from the ground up. Given that the integration of cloud-based machine learning models and serverless computation is necessary for the smooth operation of higher level tasks such as API masks, request handling, and frontend design in our web application, we created a roadmap to guide the development process. We implemented each component in a step-by-step manner to ensure that the final product met our requirements and provided a user-friendly experience.

Implementation

In the following part of our project report, we will discuss how we have implemented our project leveraging various cloud services, client-side code, and data store and management in detail.

Cloud Services

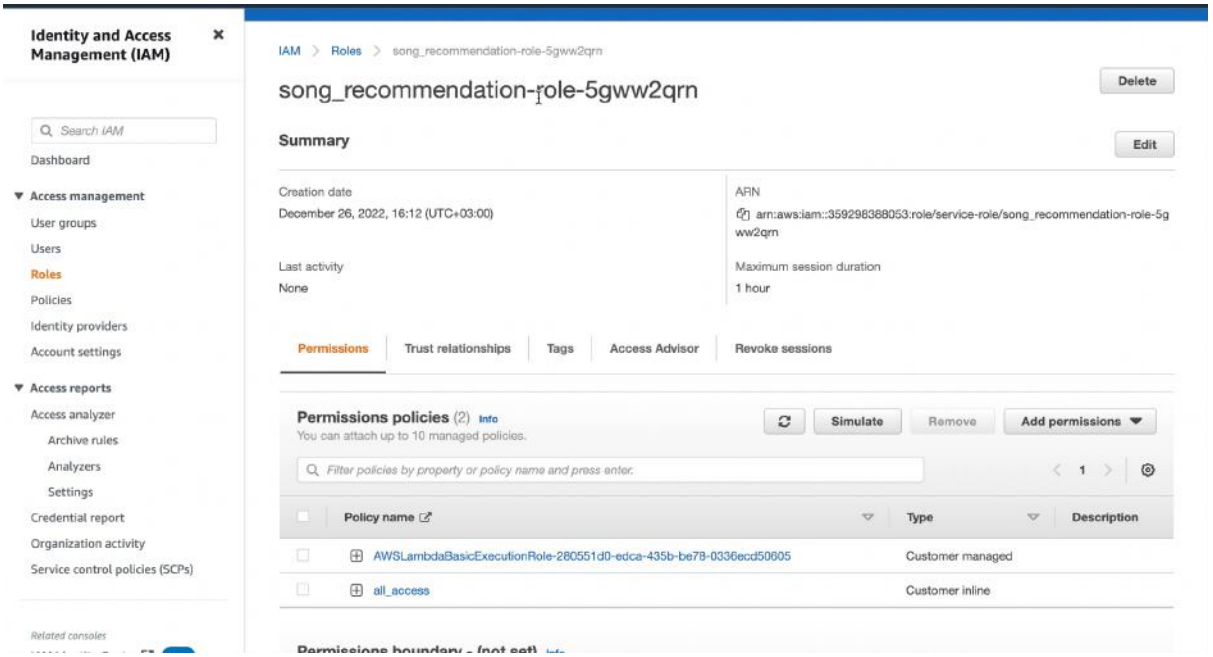
As the first step of the implementation phase of our cloud computing project, we carefully researched and compared various cloud service providers to determine the best fit for our needs. After thorough evaluation, we decided to proceed with the Amazon Web Services for our project because of the cost-free tier eligibility of the services we are going to use and the broad range of available services in the platform.

We are utilizing several cloud services on this platform, including AWS IAM, AWS SageMaker, AWS S3 Bucket, AWS Lambda, AWS API Gateway, AWS EC2 Instance, AWS Elastic IP Address.

AWS IAM

In our cloud computing project, we utilized the AWS Identity and Access Management (IAM) service to create roles and manage the various services that we utilized. This allowed us to better organize and control access to the resources and functions needed for our project. By using IAM, we were able to set fine-grained permissions for individual users and resources, ensuring that only authorized personnel had access to sensitive information and operations.

When we were dealing with the AWS SageMaker endpoint, we encountered some permission errors. In order to resolve these issues and ensure that our project could proceed smoothly within the time limitations, we granted full access to a specific role that we had created. This allowed the role to be used across all AWS services and eliminated any issues related to permissions. While granting broad access to a role can potentially increase security risks, in this case it was necessary to ensure that our project could continue without further delay. By carefully managing and monitoring the use of this role, we were able to successfully implement the SageMaker endpoint and complete our project.



AWS SageMaker

In our project, we leveraged Amazon SageMaker to train a preprocessed dataset that was stored in an AWS S3 bucket using its de facto K-Means clustering algorithm. AWS SageMaker notebook instance retrieves the preprocessed dataset from the S3 bucket and loads it into memory. Then, it divides the dataset into three subsets: a training set, a test set, and a validation set. The training set is used to fit the model, the test set is used to evaluate the model's performance, and the validation set is used to fine-tune the model's hyperparameters.

After SageMaker notebook created a model using this dataset, we used this model to train the dataset further. The use of SageMaker allowed us to take advantage of its powerful machine learning capabilities and easily train our dataset in the cloud just by executing several commands via the command line interface of our local system. By storing the dataset in S3 bucket, we were able to access it easily and efficiently from within SageMaker, resulting in a seamless and streamlined training process.

In order to configure our dataset for use with Amazon SageMaker, we added the following line of code:
train_input = TrainingInput(s3_data='s3://{}/train'.format(bucket), content_type='text/csv')

This allowed us to specify the location and format of our dataset stored in S3, ensuring that it was compatible with SageMaker. At first, we encountered some errors when attempting to create an endpoint using Amazon SageMaker. Upon further investigation, we discovered that our dataset included header rows for the entries, which were not compatible with SageMaker. In order to resolve this issue, we had to drop the header columns and re-process the dataset before proceeding with the endpoint creation. While this required some additional work and effort on our part, it ultimately allowed us to successfully use the dataset with SageMaker and move forward with our project.

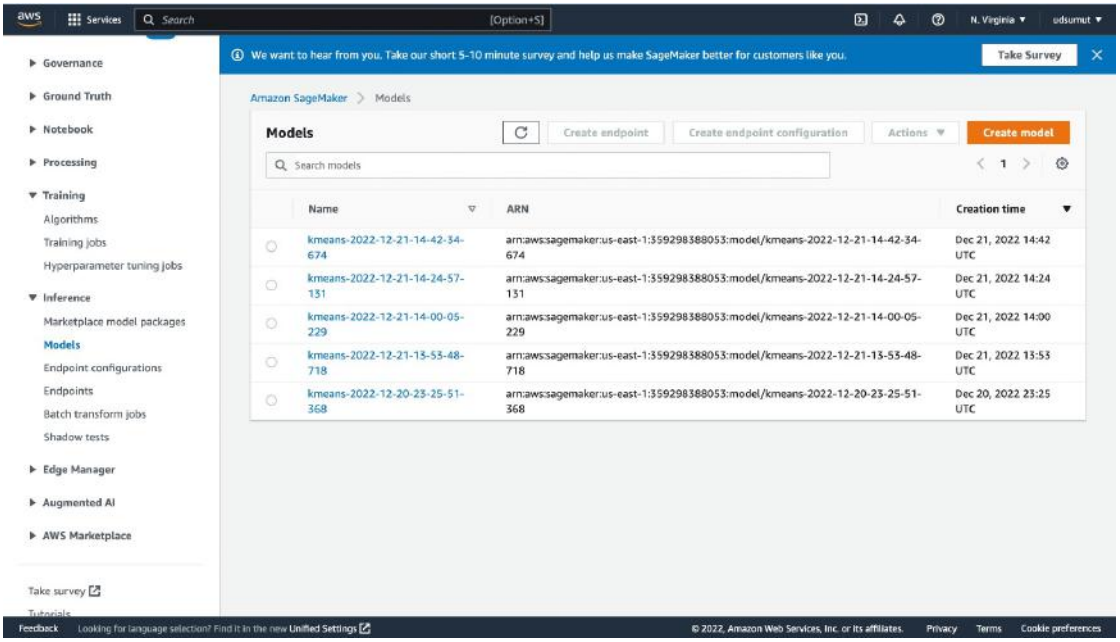
Next, we set up an estimator using sagemaker.estimator.Estimator, specifying the role that we had created during the IAM phase and selecting the train_instance_type='ml.m4.xlarge', which was within the free-tier limits. Once the estimator was configured, we used estimator.fit({'train': train_input}) to train the model, and then deployed the trained model using predictor = estimator.deploy. This allowed us to use the model for inference and make predictions on new data.

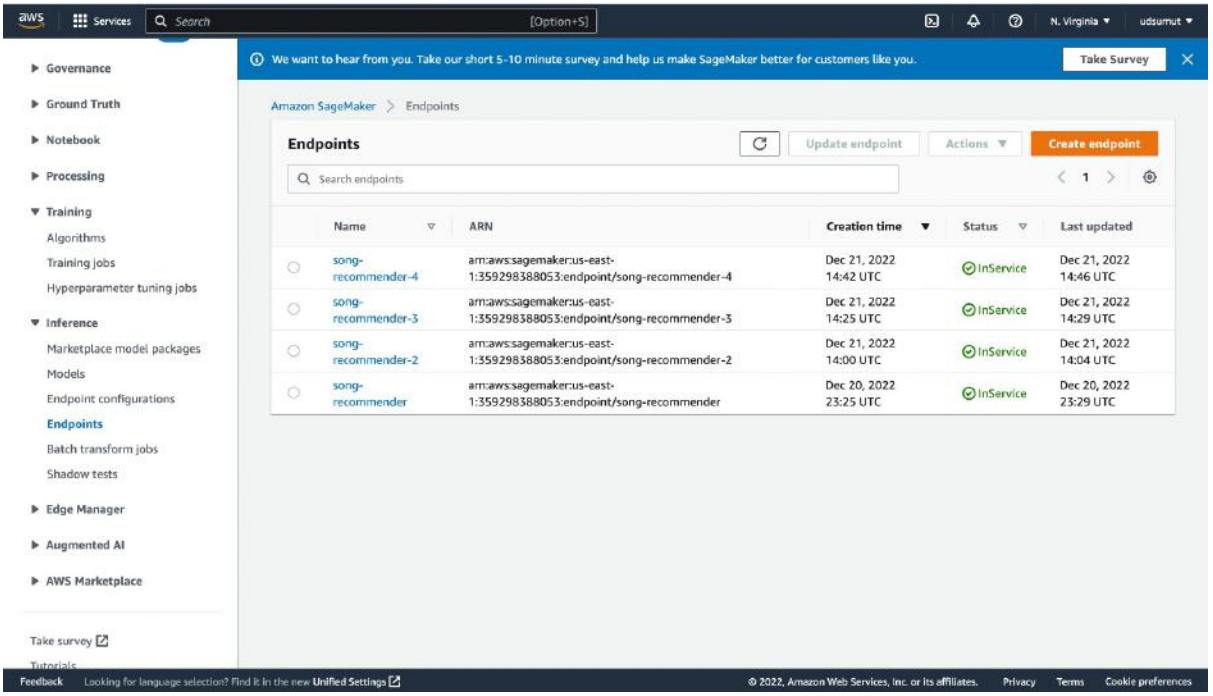
We also encountered errors during the training of our dataset. We have observed that predictor.predict() function only accepts byte string type of inputs to make predictions and create a trained dataset. In order to resolve this issue, we implemented the following code to train the dataset:

```
payload = train_data.values
byte_payload = payload.encode('utf-8')
predictions = predictor.predict(byte_payload)
```

By encoding the payload as a byte string and using the predictor.predict function, we were able to successfully train the dataset and make predictions using our trained model.
Creation of SageMaker endpoint:

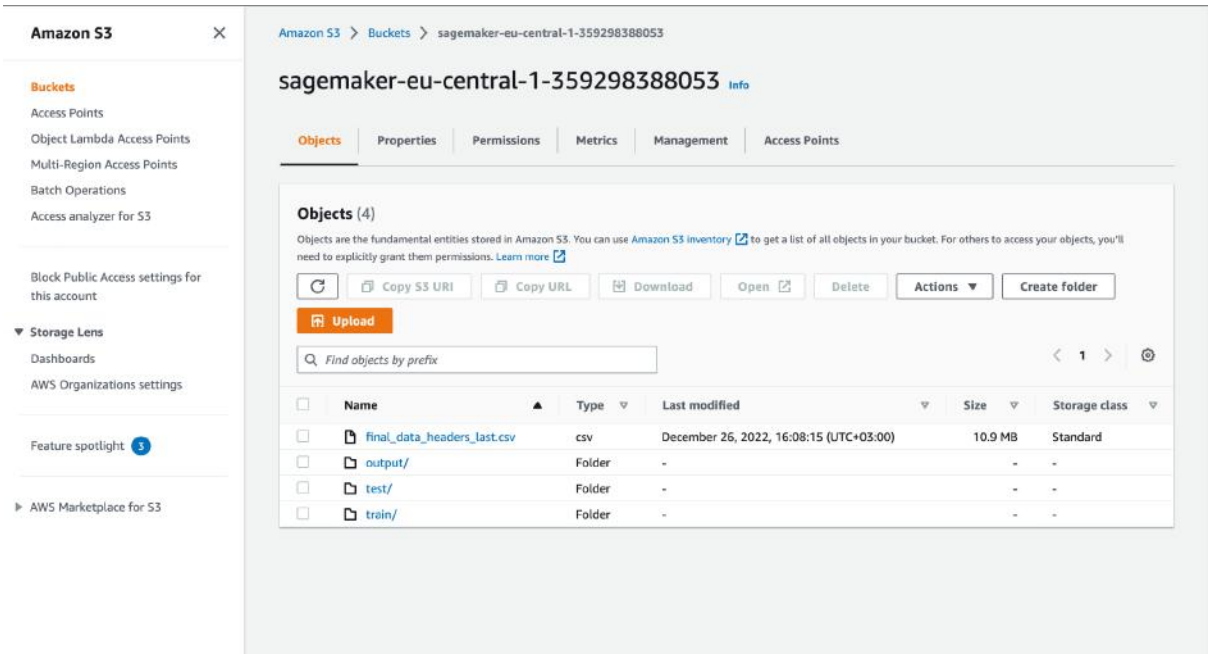
```
The method get_image_uri has been renamed in sagemaker>=2.
See: https://sagemaker.readthedocs.io/en/stable/v2.html for details.
train_instance_count has been renamed in sagemaker>=2.
See: https://sagemaker.readthedocs.io/en/stable/v2.html for details.
train_instance_type has been renamed in sagemaker>=2.
See: https://sagemaker.readthedocs.io/en/stable/v2.html for details.
INFO:sagemaker:Creating training-job with name: kmeans-2022-12-26-13-08-30-703
2022-12-26 13:08:32 Starting - Starting the training job...
2022-12-26 13:08:50 Starting - Preparing the instances for training.....
2022-12-26 13:10:57 Downloading - Downloading input data.....
2022-12-26 13:12:07 Training - Training image download completed. Training in progress..Docker entrypoint called with argument(s): train
Running default environment configuration script
[12/26/2022 13:12:09 INFO 140469342971712] Reading default configuration from /opt/amazon/lib/python3.7/site-packages/algorithms/resources/default-input.json: {'init_method': 'random', 'mini_batch_size': '5000', 'epochs': '1', 'extra_center_factor': 'auto', 'local_lloyd_max_iter': '300', 'local_lloyd_tol': '0.0001', 'local_lloyd_init_method': 'kmeans++', 'local_lloyd_num_trials': 'auto', 'half_life_time_size': '0', 'eval_metrics': '["msd"]', 'force_dense': 'true', '_disable_wait_to_read': 'false', '_enable_profiler': 'false', '_kvstore': 'auto', '_log_level': 'info', '_num_gpus': 'auto', '_num_kv_servers': '1', '_num_slices': '1', '_tuning_objective_metric': ''}
[12/26/2022 13:12:09 INFO 140469342971712] Merging with provided configuration from /opt/ml/input/config/hyperparameters.json: {'feature_dim': '12', 'k': '20'}
[12/26/2022 13:12:09 INFO 140469342971712] Final configuration: {'init_method': 'random', 'mini_batch_size': '5000', 'epochs': '1', 'extra_center_factor': 'auto', 'local_lloyd_max_iter': '300', 'local_lloyd_tol': '0.0001', 'local_lloyd_init_method': 'kmeans++', 'local_lloyd_num_trials': 'auto', 'half_life_time_size': '0', 'eval_metrics': '["msd"]', 'force_dense': 'true', '_disable_wait_to_read': 'false', '_enable_profiler': 'false', '_kvstore': 'auto', '_log_level': 'info', '_num_gpus': 'auto', '_num_kv_servers': '1', '_num_slices': '1', '_tuning_objective_metric': '', 'feature_dim': '12', 'k': '20'}
[12/26/2022 13:12:11 WARNING 140469342971712] Loggers have already been setup.
[12/26/2022 13:12:11 INFO 140469342971712] Final configuration: {'init_method': 'random', 'mini_batch_size': '5000', 'epochs': '1', 'extra_center_factor': 'auto', 'local_lloyd_max_iter': '300', 'local_lloyd_tol': '0.0001', 'local_lloyd_init_method': 'kmeans++', 'local_lloyd_num_trials': 'auto', 'half_life_time_size': '0', 'eval_metrics': '["msd"]', 'force_dense': 'true', '_disable_wait_to_read': 'false', '_enable_profiler': 'false', '_kvstore': 'auto', '_log_level': 'info', '_num_gpus': 'auto', '_num_kv_servers': '1', '_num_slices': '1', '_tuning_objective_metric': '', 'feature_dim': '12', 'k': '20'}
[12/26/2022 13:12:11 WARNING 140469342971712] Loggers have already been setup.
Process 7 is a worker.
[12/26/2022 13:12:11 INFO 140469342971712] Using default worker.
[12/26/2022 13:12:11 INFO 140469342971712] Loaded iterator creator application/x-recordio-protobuf for content type ('application/x-recordio-protobuf', '1.0')
[12/26/2022 13:12:11 INFO 140469342971712] Create Store: local
[12/26/2022 13:12:11 INFO 140469342971712] nvidia-smi: took 0.030 seconds to run.
[12/26/2022 13:12:11 INFO 140469342971712] nvidia-smi identified 0 GPUs.
[12/26/2022 13:12:11 INFO 140469342971712] Number of GPUs being used: 0
[12/26/2022 13:12:11 INFO 140469342971712] Checkpoint loading and saving are disabled.
[12/26/2022 13:12:11 INFO 140469342971712] Setting up with params: {'init_method': 'random', 'mini_batch_size': '5000', 'epochs': '1', 'extra_center_factor': 'auto', 'local_lloyd_max_iter': '300', 'local_lloyd_tol': '0.0001', 'local_lloyd_init_method': 'kmeans++', 'local_lloyd_num_trials': 'auto', 'half_life_time_size': '0', 'eval_metrics': '["msd"]', 'force_dense': 'true', '_disable_wait_to_read': 'false', '_enable_profiler': 'false', '_kvstore': 'auto', '_log_level': 'info', '_num_gpus': 'auto', '_num_kv_servers': '1', '_num_slices': '1', '_tuning_objective_metric': '', 'feature_dim': '12', 'k': '20'}
[12/26/2022 13:12:11 INFO 140469342971712] 'extra_center_factor' was set to 'auto', evaluated to 10.
[12/26/2022 13:12:11 INFO 140469342971712] Number of GPUs being used: 0
[12/26/2022 13:12:11 INFO 140469342971712] number of center slices 1
#metrics {"StartTime": 1672060331.5905643, "EndTime": 1672060331.5905988, "Dimensions": {"Algorithm": "AWS/KMeansWebscale", "Host": "algo-1", "Operation": "training", "Meta": "init_train_data_iter"}, "Metrics": {"Total Records Seen": {"sum": 5000.0, "count": 1, "min": 5000, "max": 5000}, "Total Batches Seen": {"sum": 1.0, "count": 1, "min": 1, "max": 1}, "Max Records Seen Between Resets": {"sum": 5000.0, "count": 1, "min": 5000, "max": 5000}, "Max Batches Seen Between Resets": {"sum": 1.0, "count": 1, "min": 1, "max": 1}, "Reset Count": {"sum": 0.0, "count": 1, "min": 0, "max": 0}, "Number of Records Since Last Reset": {"sum": 5000.0, "count": 1, "min": 5000, "max": 5000}, "Number of Batches Since Last Reset": {"sum": 1.0, "count": 1, "min": 1, "max": 1}}}
[12/26/2022 13:12:11 INFO 140469342971712] processed a total of 39087 examples
[12/26/2022 13:12:11 INFO 140469342971712] #progress_metric: host=algo-1, completed 100.0 % of epochs
#metrics {"StartTime": 1672060331.5910256, "EndTime": 1672060331.9226332, "Dimensions": {"Algorithm": "AWS/KMeansWebscale", "Host": "algo-1", "Operation": "training", "epoch": 0, "Meta": "training_data_iter"}, "Metrics": {"Total Records Seen": {"sum": 44087.0, "count": 1, "min": 44087, "max": 44087}, "Total Batches Seen": {"sum": 9.0, "count": 1, "min": 9, "max": 9}, "Max Records Seen Between Resets": {"sum": 39087.0, "count": 1, "min": 39087, "max": 39087}, "Max Batches Seen Between Resets": {"sum": 8.0, "count": 1, "min": 8, "max": 8}, "Reset Count": {"sum": 1.0, "count": 1, "min": 1, "max": 1}, "Number of Records Since Last Reset": {"sum": 39087.0, "count": 1, "min": 39087, "max": 39087}, "Number of Batches Since Last Reset": {"sum": 8.0, "count": 1, "min": 8, "max": 8}}}
[12/26/2022 13:12:11 INFO 140469342971712] #throughput_metric: host=algo-1, train throughput=117804.59199367657 records/second
[12/26/2022 13:12:11 WARNING 140469342971712] wait_for_all_workers will not sync workers since the kv store is not running distributed
[12/26/2022 13:12:11 INFO 140469342971712] shrinking 200 centers into 20
[12/26/2022 13:12:11 INFO 140469342971712] local kmeans attempt #0. Current mean square distance 0.087678
[12/26/2022 13:12:12 INFO 140469342971712] local kmeans attempt #1. Current mean square distance 0.088618
[12/26/2022 13:12:12 INFO 140469342971712] local kmeans attempt #2. Current mean square distance 0.085839
[12/26/2022 13:12:12 INFO 140469342971712] local kmeans attempt #3. Current mean square distance 0.092659
[12/26/2022 13:12:12 INFO 140469342971712] local kmeans attempt #4. Current mean square distance 0.084058
[12/26/2022 13:12:12 INFO 140469342971712] local kmeans attempt #5. Current mean square distance 0.089579
[12/26/2022 13:12:12 INFO 140469342971712] local kmeans attempt #6. Current mean square distance 0.087369
[12/26/2022 13:12:12 INFO 140469342971712] local kmeans attempt #7. Current mean square distance 0.087974
[12/26/2022 13:12:12 INFO 140469342971712] local kmeans attempt #8. Current mean square distance 0.087410
[12/26/2022 13:12:12 INFO 140469342971712] local kmeans attempt #9. Current mean square distance 0.083428
[12/26/2022 13:12:12 INFO 140469342971712] finished shrinking process. Mean Square Distance = 0
[12/26/2022 13:12:12 INFO 140469342971712] #quality_metric: host=algo-1, train msd <loss>=0.08342767506837845
[12/26/2022 13:12:12 INFO 140469342971712] batch data loading with context took: 30.9682%, (0.104513 secs)
[12/26/2022 13:12:12 INFO 140469342971712] predict compute msd took: 27.4820%, (0.092747 secs)
[12/26/2022 13:12:12 INFO 140469342971712] gradient: cluster size  took: 18.2987%, (0.061755 secs)
[12/26/2022 13:12:12 INFO 140469342971712] compute all data-center distances: inner product took: 10.8304%, (0.036551 secs)
[12/26/2022 13:12:12 INFO 140469342971712] collect from kv store took: 2.8994%, (0.009785 secs)
[12/26/2022 13:12:12 INFO 140469342971712] gradient: cluster center took: 2.7444%, (0.009262 secs)
[12/26/2022 13:12:12 INFO 140469342971712] splitting centers key-value pair took: 2.6142%, (0.008823 secs)
[12/26/2022 13:12:12 INFO 140469342971712] compute all data-center distances: point norm took: 1.4090%, (0.004755 secs)
[12/26/2022 13:12:12 INFO 140469342971712] gradient: one_hot took: 1.1938%, (0.004029 secs)
[12/26/2022 13:12:12 INFO 140469342971712] update state and report convergance took: 1.0240%, (0.003456 secs)
[12/26/2022 13:12:12 INFO 140469342971712] compute all data-center distances: center norm took: 0.4186%, (0.001413 secs)
[12/26/2022 13:12:12 INFO 140469342971712] update set-up time took: 0.0711%, (0.000240 secs)
[12/26/2022 13:12:12 INFO 140469342971712] predict minus dist took: 0.0461%, (0.000156 secs)
[12/26/2022 13:12:12 INFO 140469342971712] TOTAL took: 0.33748435974121094
[12/26/2022 13:12:12 INFO 140469342971712] Number of GPUs being used: 0
#metrics {"StartTime": 1672060331.5195923, "EndTime": 1672060332.5202692, "Dimensions": {"Algorithm": "AWS/KMeansWebscale", "Host": "algo-1", "Operation": "training"}, "Metrics": {"initialize.time": {"sum": 70.56093215942383, "count": 1, "min": 70.56093215942383, "max": 70.56093215942383}, "epochs": {"sum": 1.0, "count": 1, "min": 1, "max": 1}, "update.time": {"sum": 331.3732147216797, "count": 1, "min": 331.3732147216797, "max": 331.3732147216797}, "_shrink.time": {"sum": 594.6400165557861, "count": 1, "min": 594.6400165557861, "max": 594.6400165557861}, "finalize.time": {"sum": 596.930742263794, "count": 1, "min": 596.930742263794, "max": 596.930742263794}, "model.serialize.time": {"sum": 0.2009868621826172, "count": 1, "min": 0.2009868621826172, "max": 0.2009868621826172}}}
[12/26/2022 13:12:12 INFO 140469342971712] Test data is not provided.
#metrics {"StartTime": 1672060332.520373, "EndTime": 1672060332.5206082, "Dimensions": {"Algorithm": "AWS/KMeansWebscale", "Host": "algo-1", "Operation": "training"}, "Metrics": {"setuptime": {"sum": 2.3212432861328125, "count": 1, "min": 2.3212432861328125, "max": 2.3212432861328125}, "totaltime": {"sum": 1049.2372512817383, "count": 1, "min": 1049.2372512817383, "max": 1049.2372512817383}}}

2022-12-26 13:12:28 Uploading - Uploading generated training model
2022-12-26 13:12:28 Completed - Training job completed
Training seconds: 92
Billable seconds: 92
```





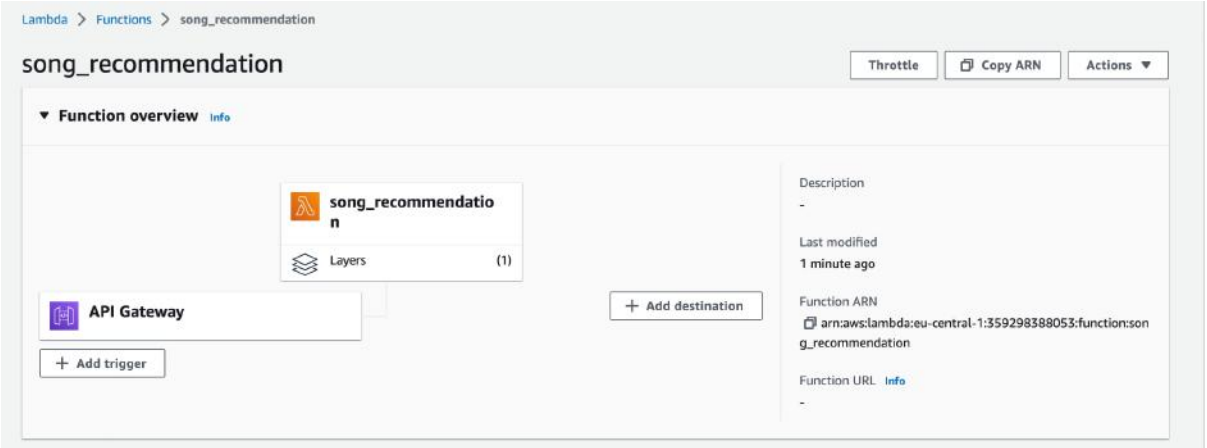
AWS S3 Bucket

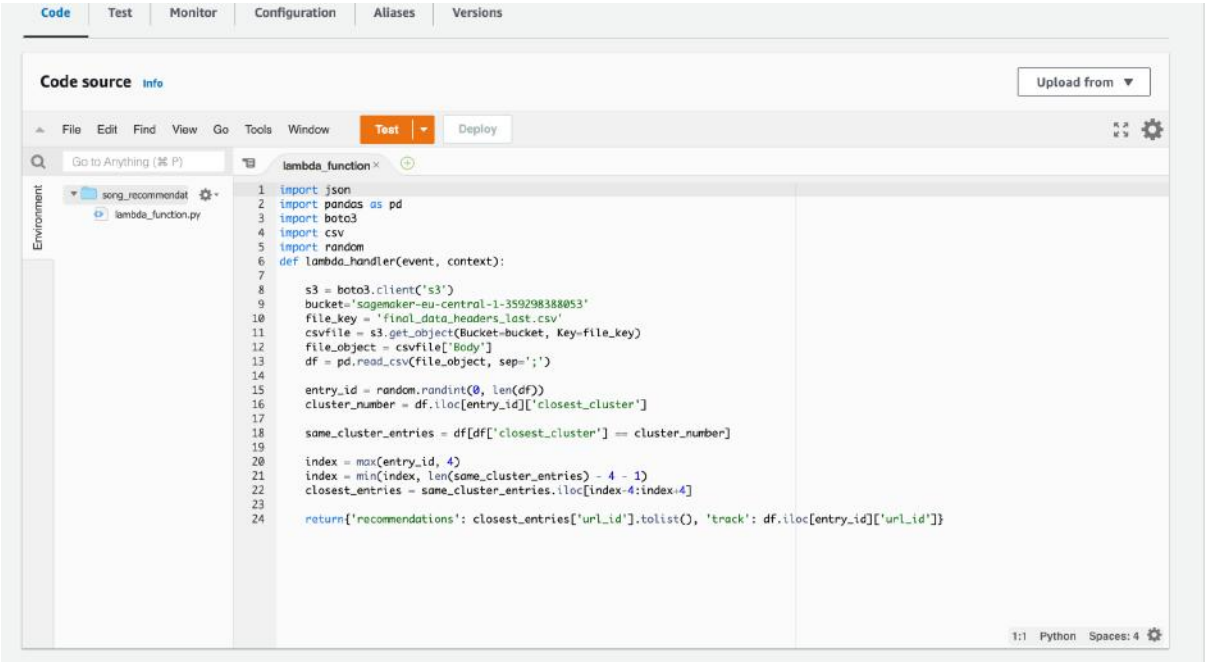
We have used Amazon S3 to store both preprocessed CSV files and the trained dataset produced by the SageMaker endpoint. S3 provided a convenient and reliable storage solution for our project, allowing us to easily store and access our data in the cloud. Since all the other services that we use to access the data were also provided by the same CSP, namely AWS, by using S3 to store both the preprocessed CSV files and the trained dataset, we were able to efficiently manage and organize our data, ensuring that it was readily available for use as needed.

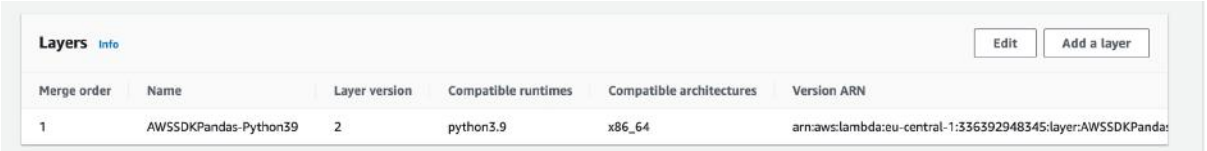


AWS Lambda

In our AWS-based application, we have utilized Lambda functions to handle API requests and provide machine learning services. One specific use case for our Lambda function is to retrieve and return a set of Spotify song URLs based on a processed and sorted dataset stored in an S3 bucket. To accomplish this, we first import the necessary libraries, including boto3 which allows us to interact with our S3 bucket. We then specify the bucket and file we want to access, and use boto3 to retrieve the contents of the file as a CSV. Next, we generate a random integer between 0 and the length of the dataset, and use this integer to retrieve a specific entry in the dataset. We then use the information in this entry to locate the closest cluster, and find the 4 entries above and below the current entry in the dataset. Finally, we return the Spotify song URLs for these 8 entries, as well as the URL for the entry itself. Overall, our Lambda function enables us to efficiently retrieve and return a set of relevant Spotify song URLs based on a processed and sorted dataset stored in S3, using the power of serverless computing and the flexibility of the AWS ecosystem.

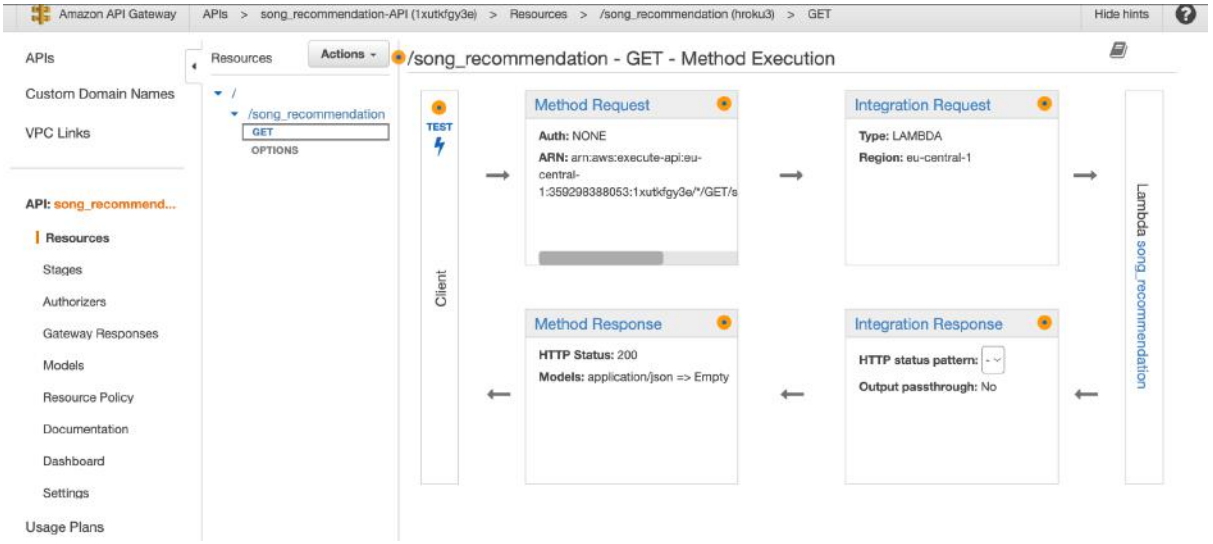






AWS API Gateway

In our AWS-based application, we have utilized API Gateway to trigger our Lambda function and handle incoming API requests. When a GET request is made to the API endpoint, API Gateway triggers the associated Lambda function, passing any relevant parameters and request data. The Lambda function then executes and returns a response, which is passed back through API Gateway and returned to the API requestor. API Gateway acts as a bridge between our Lambda function and the outside world, enabling us to easily build and deploy APIs that can trigger our serverless functions and return the results to clients. It also provides various features such as authentication, rate limiting, and caching, allowing us to easily secure and manage our API endpoint. Overall, API Gateway enables us to easily build and deploy APIs that can trigger our Lambda functions and handle incoming requests, providing a powerful and flexible way to expose our application's functionality to the world.



AWS EC2 Instance

We created an Elastic Compute Cloud (EC2) instance to host our web application's frontend code. To ensure that the instance was publicly accessible, we masked it behind an elastic IP address. We deployed our frontend code using a Docker image and container, which allowed us to easily package and deploy our application in a consistent and repeatable manner. Overall, this setup has provided us with a reliable and scalable infrastructure for hosting our web application.

Initially, we attempted to use the CodeBuild service offered by AWS to automate the deployment of our web application on the virtual machine. However, the configuration for CodeBuild proved to be quite complex and confusing, and we were unable to proceed beyond the build phase. As a result, we decided to use a more traditional approach and deployed our Docker image and container directly on the EC2 instance by accessing the

instance's command line interface through the AWS console. This allowed us to deploy our web application as needed within the time limitations, without the added complexity of the CodeBuild service. Overall, this approach has proven to be a reliable and efficient way to manage our application's deployment compared to the more experimental and rather newer option of CodeBuild.

AWS Elastic IP Address

To make our web application publicly accessible, we used the AWS Elastic IP address service to mask our EC2 instance, which contained our web application's frontend code. This allowed us to assign a static, publicly-accessible IP address to our EC2 instance, allowing users to access our application over the internet. By using an Elastic IP address, we were able to ensure that our application was consistently available and could be easily accessed by users from any location.

Client-Side Code

When we have successfully implemented all the cloud services mentioned above, we proceed with building the frontend code of our web application. Basically, the cloud services served as the backend of our web application, since it features only one API call regarding the recommendation process. That API call is binded to the AWS Lambda function within the API Gateway we have created.

Besides the computation we received from the cloud services, we also added an external API call to Spotify’s web API to provide a more user friendly web application.

Web Application

We implemented a basic React web application to provide a user-friendly interface for accessing the computation performed on the cloud. The web application is a single page application that makes an API call to the endpoint every time the user clicks the "echo songs" button. When the results are returned from the endpoint, the web application converts them into clickable song card components featuring the album cover image, song name, and artist name, using the Spotify Web API. These song cards are then displayed to the user, and when clicked, the user is redirected to the corresponding song's Spotify page.

The response of the endpoint we have created using the API Gateway is a JSON object with two fields: recommendations and track. Track has the Spotify song id of the randomly generated song and recommendations has the array of 8 recommended song’s Spotify song ids. Using this id values of the songs within the external API call to Spotify web API, we could successfully fetch the AWS Lambda returned songs’ data.

Overall, this web application provides a simple and intuitive way for users to access the computation performed on the cloud and interact with the results. By leveraging the Spotify Web API, we were able to enhance the user experience by providing additional information and functionality related to the songs returned by the endpoint.

Docker

To deploy our web application on the EC2 instance, we used Docker. Docker is a containerization platform that allows us to package our application and its dependencies into a self-contained unit, known as a container.

By using Docker, we were able to quickly and easily deploy our web application on the EC2 instance, without having to worry about installing and configuring the necessary dependencies and libraries on the server. This streamlined the deployment process and allowed us to focus on developing and testing our application. Overall, Docker proved to be an invaluable tool for deploying our web application on the EC2 instance.

Data Storage and Management

We have used a publicly available dataset of songs fetched from Spotify's API to build a song recommendation system. The dataset is taken from Kaggle. It featured 22 columns and all of them were not necessary for our implementation. Thus, we have preprocessed the data by dropping unnecessary columns, applying one-hot encoding to categorical variables, and handling missing values. These steps are necessary to ensure that the dataset is in a suitable format for training a machine learning model. With the Python script we have written, we have saved the preprocessed dataset to the same S3 bucket that has the untrained CSV files, so that it can be accessed by the AWS Lambda function to fetch recommended songs.

To make the computation process quicker and easier for the AWS Lambda function, we have reordered the trained data in accordance to the cluster numbers for each song entry in the trained dataset. Furthermore, we also arranged the song entries in ascending order of distance to the cluster field returned by the recommendation model within each cluster. This rearranged dataset allowed us to use AWS Lambda with more flexibility, since it does not feature the burden of processing each entry in the dataset to identify recommended, i.e. similar songs. Instead, it just finds the entry and gets the eight neighbors, i.e. eight most similar songs.

Stress Testing

JMeter

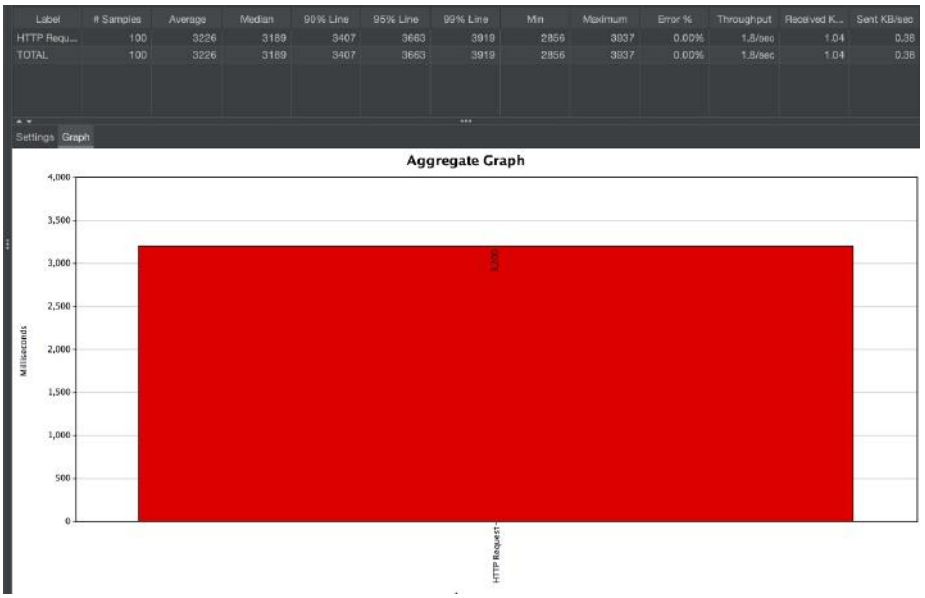
To evaluate the behavior and performance of our cloud-based system under heavy usage, we used the stress testing tool JMeter to simulate different scenarios with different levels of load. The goal of the stress testing was to test the scalability and reliability of our system, based on different parameters such as the number of users, the number of samples, and the ramp-up time.

In JMeter, we created a test plan with a thread group. In the thread group, we changed the parameters Number of threads (users) and ramp-up period (seconds) to simulate different levels of load. We also added a constant timer to pause the threads between requests, and

an HTTP request module to make GET requests to our API, which was deployed using API Gateway. In API Gateway, we trigger an AWS Lambda function to handle the requests. However, there is a limit for the Lambda function that allows only 10 users at the same time. Therefore, we had to take this constraint into consideration and adjust the ramp-up period accordingly. Finally, we added an Aggregate Report listener to collect and display the results of the stress test.

We tested our API with several different combinations of parameters in the thread group, and observed the performance of the system under each scenario. Some of the key findings from the stress testing are as follows:

10 users with 100 samples 10 sec ramp-up:
The system performed well, with an average response time of 3226 ms. The median response time is the middle value of all the response times measured, and in this case it was 3189 ms. The 90th percentile response time is the response time that a request completed in, such that 90% of requests completed in a shorter time and 10% of requests completed in a longer time. In this case, the 90th percentile response time was 3407 ms. The 95th percentile response time is the response time that a request completed in, such that 95% of requests completed in a shorter time and 5% of requests completed in a longer time. In this case, the 95th percentile response time was 3663 ms. The 99th percentile response time is the response time that a request completed in, such that 99% of requests completed in a shorter time and 1% of requests completed in a longer time. In this case, the 99th percentile response time was 3919 ms. The minimum response time was the shortest response time measured, which was 2856 ms in this case. The maximum response time was the longest response time measured, which was 3937 ms in this case. There were no errors reported and the throughput was 1.8 requests per second. The system received 1.04 kb/s and sent 0.38 kb/s.

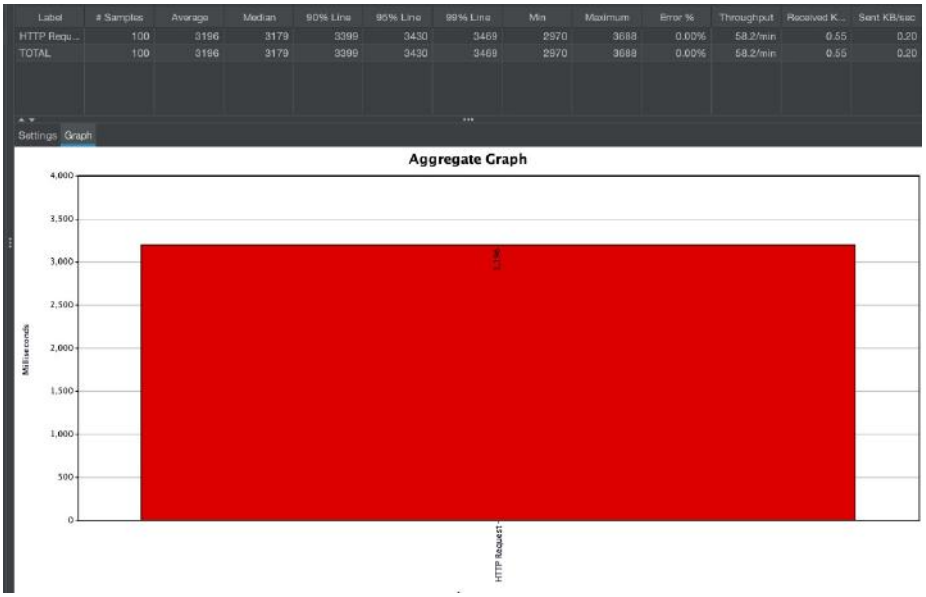


20 users with 100 samples and 100 sec ramp-up:
The system performed well, with an average response time of 3111 ms, a median response time of 3101 ms, and 90th, 95th, and 99th percentile response times of 3305 ms, 3330 ms,

and 3431 ms, respectively. There were no errors reported and the throughput was 48.7 requests per minute.

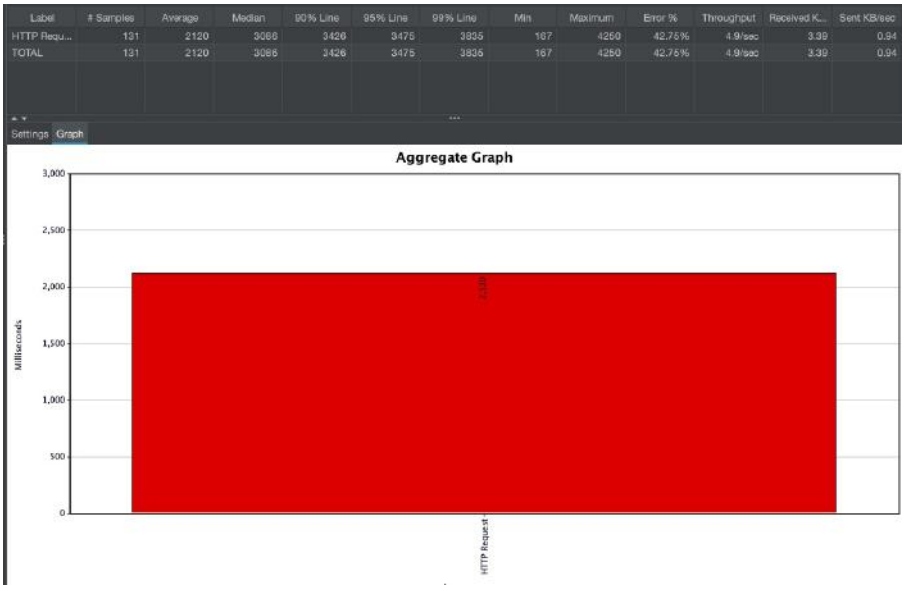
100 users with 100 samples and 100 sec ramp-up:
The system experienced some degradation in performance, with an average response time of 3196 ms, a median response time of 3179 ms, and 90th, 95th, and 99th percentile response times of 3399 ms, 3430 ms, and 3469 ms, respectively. There were no errors reported and the throughput was 58.2 requests per minute. The system received 0.55 kb/s and sent 0.20 kb/s.

Overall, the system performed well under this scenario, with slightly slower response times compared to the 10 and 20 user scenarios. The higher throughput indicates that the system was able to handle the increased load.



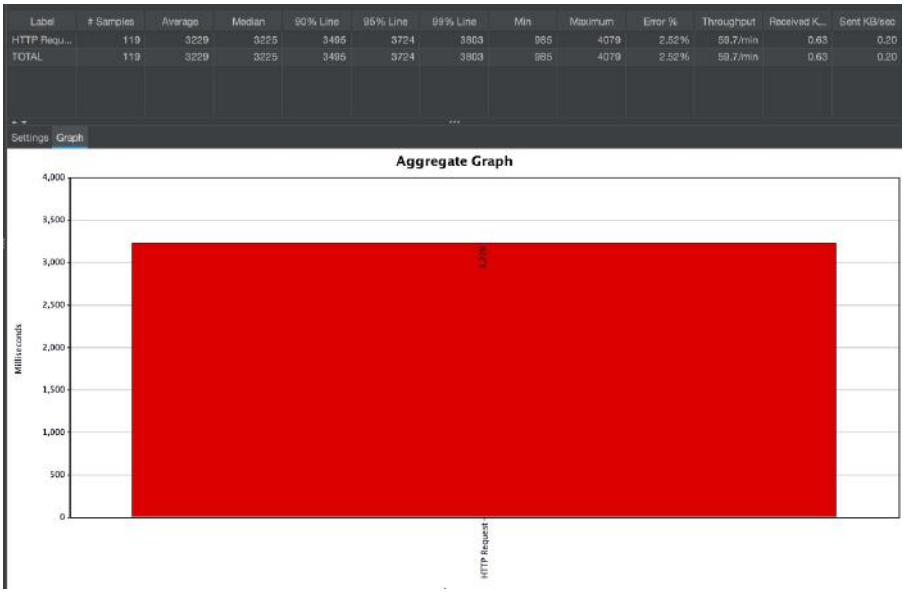
500 users with 131 samples and 100 sec ramp-up:
The system experienced significant degradation in performance, with an average response time of 2120 ms, a median response time of 3086 ms, and 90th, 95th, and 99th percentile response times of 3426 ms, 3475 ms, and 3835 ms, respectively. The minimum response time was 167 ms and the maximum response time was 9120 ms. There was a high error rate of 42.75% and the throughput was 4.9 requests per second. The system received 3.39 kb/s and sent 0.94 kb/s.

Overall, the system struggled to handle the high load under this scenario, with slow response times and a high error rate. The lower throughput suggests that the system was unable to keep up with the demands of the 500 users.



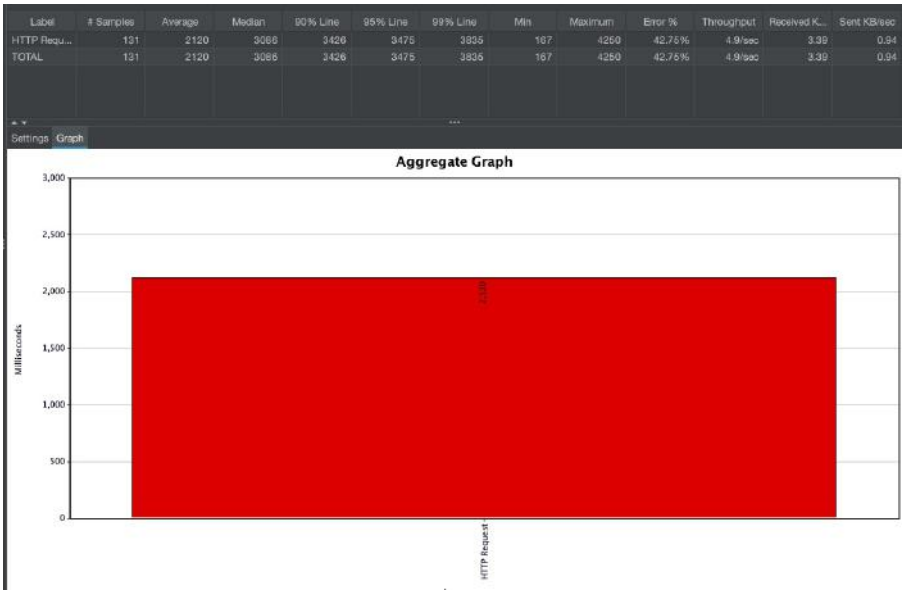
500 users with 119 samples and 500 sec ramp-up:
The system performed well, with an average response time of 3229 ms, a median response time of 3225 ms, and 90th, 95th, and 99th percentile response times of 3495 ms, 3724 ms, and 3803 ms, respectively. The minimum response time was 985 ms and the maximum response time was 4079 ms. There was a low error rate of 2.52% and the throughput was 59.7 requests per minute. The system received 0.63 kb/s and sent 0.20 kb/s.

Overall, the system performed well under this scenario, with fast response times and a low error rate. The higher throughput indicates that the system was able to handle the increased load, possibly due to the longer ramp-up period allowing the system to gradually scale up to meet the demands of the 500 users.



1000 users with 216 samples and 100 sec ramp-up:
The system experienced significant degradation in performance, with an average response time of 1178 ms, a median response time of 222 ms, and 90th, 95th, and 99th percentile response times of 3284 ms, 3367 ms, and 3783 ms, respectively. The minimum response time was 58 ms and the maximum response time was 9097 ms. There was a high error rate of 71.30% and the throughput was 9.7 requests per second. The system received 5.10 kb/s and sent 1.92 kb/s.

Overall, the system struggled to handle the high load under this scenario, with slow response times and a high error rate. The lower throughput suggests that the system was unable to keep up with the demands of the 1000 users.

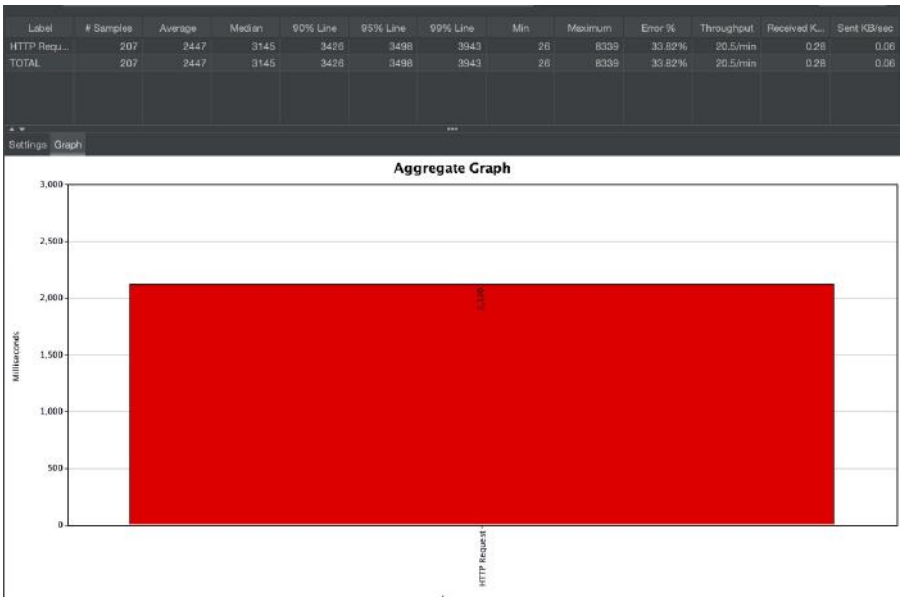


1000 users with 207 samples and 500 sec ramp-up:

1000 users, 207 samples, 500 sec ramp-up: The system experienced significant degradation

in performance, with an average response time of 2447 ms, a median response time of 3145 ms, and 90th, 95th, and 99th percentile response times of 3426 ms, 3498 ms, and 3943 ms, respectively. The minimum response time was 26 ms and the maximum response time was 8339 ms. There was a high error rate of 33.82% and the throughput was 20.5 requests per minute. The system received 0.28 kb/s and sent 0.06 kb/s.

Overall, the system did not perform well under this scenario, with slower response times and a high error rate compared to the 10 and 20 user scenarios. The lower throughput suggests that the system struggled to keep up with the demands of the 1000 users, possibly due to the increased load and shorter ramp-up period compared to the 500 user scenario.



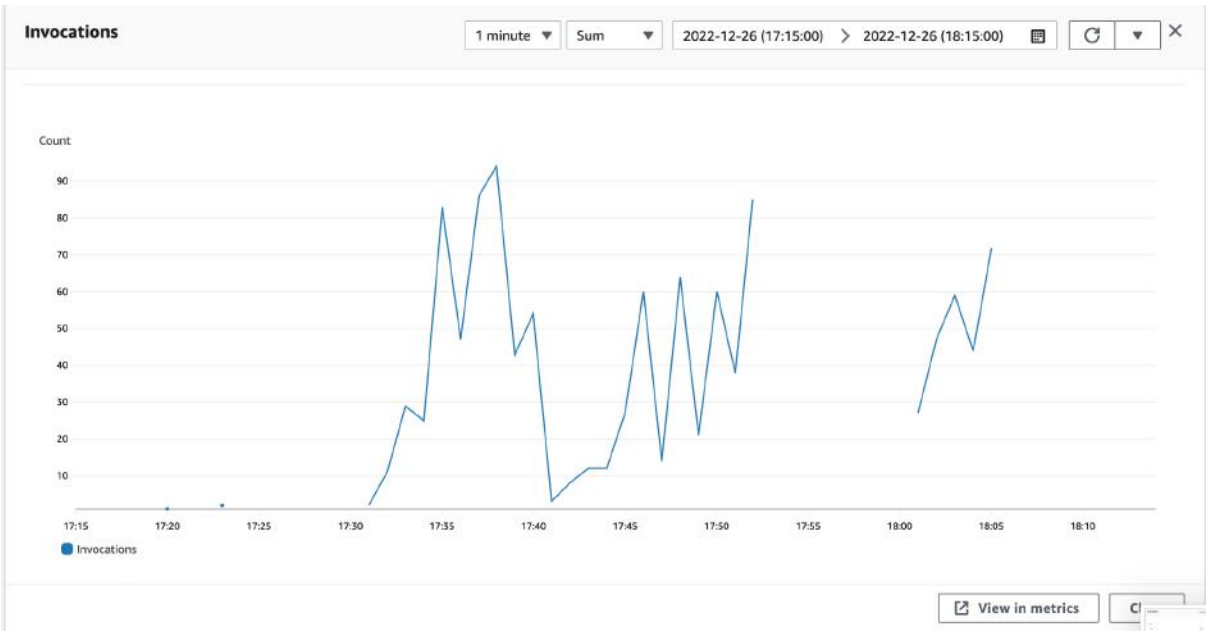
1000 users with 103 samples and 1000 sec ramp-up:

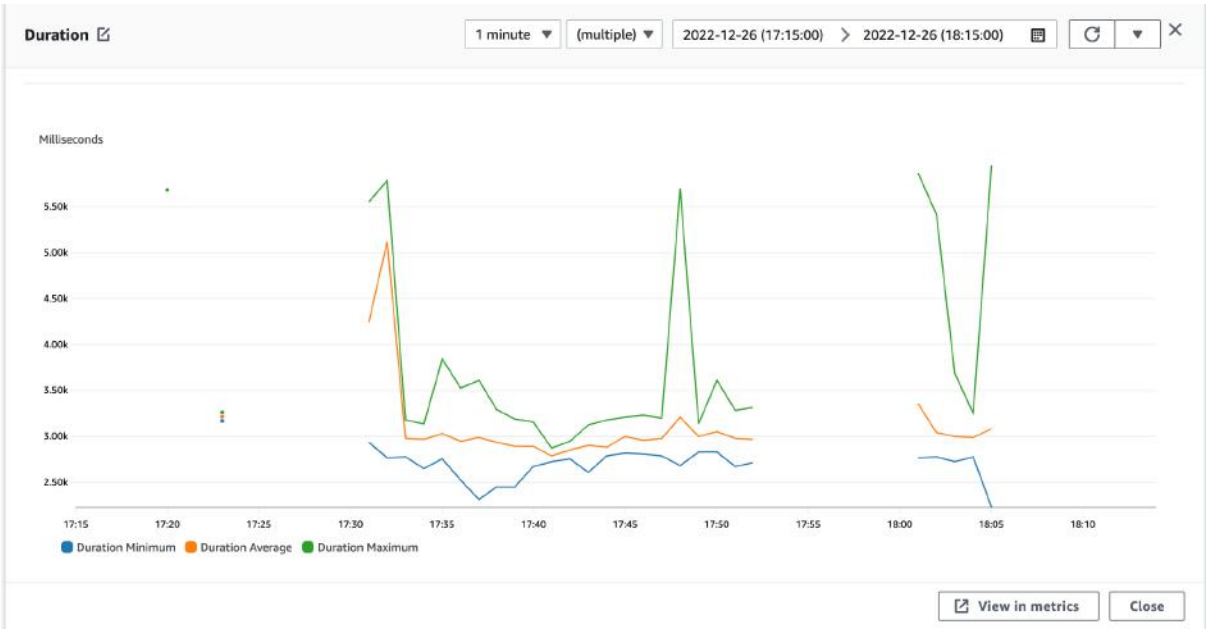
The system performed well, with an average response time of 3160 ms, a median response time of 3161 ms, and 90th, 95th, and 99th percentile response times of 3378 ms, 3427 ms, and 3864 ms, respectively. The minimum response time was 834 ms and the maximum response time was 3885 ms. There was a low error rate of 2.91% and the throughput was 59.8 requests per minute. The system received 0.64 kb/s and sent 0.20 kb/s.

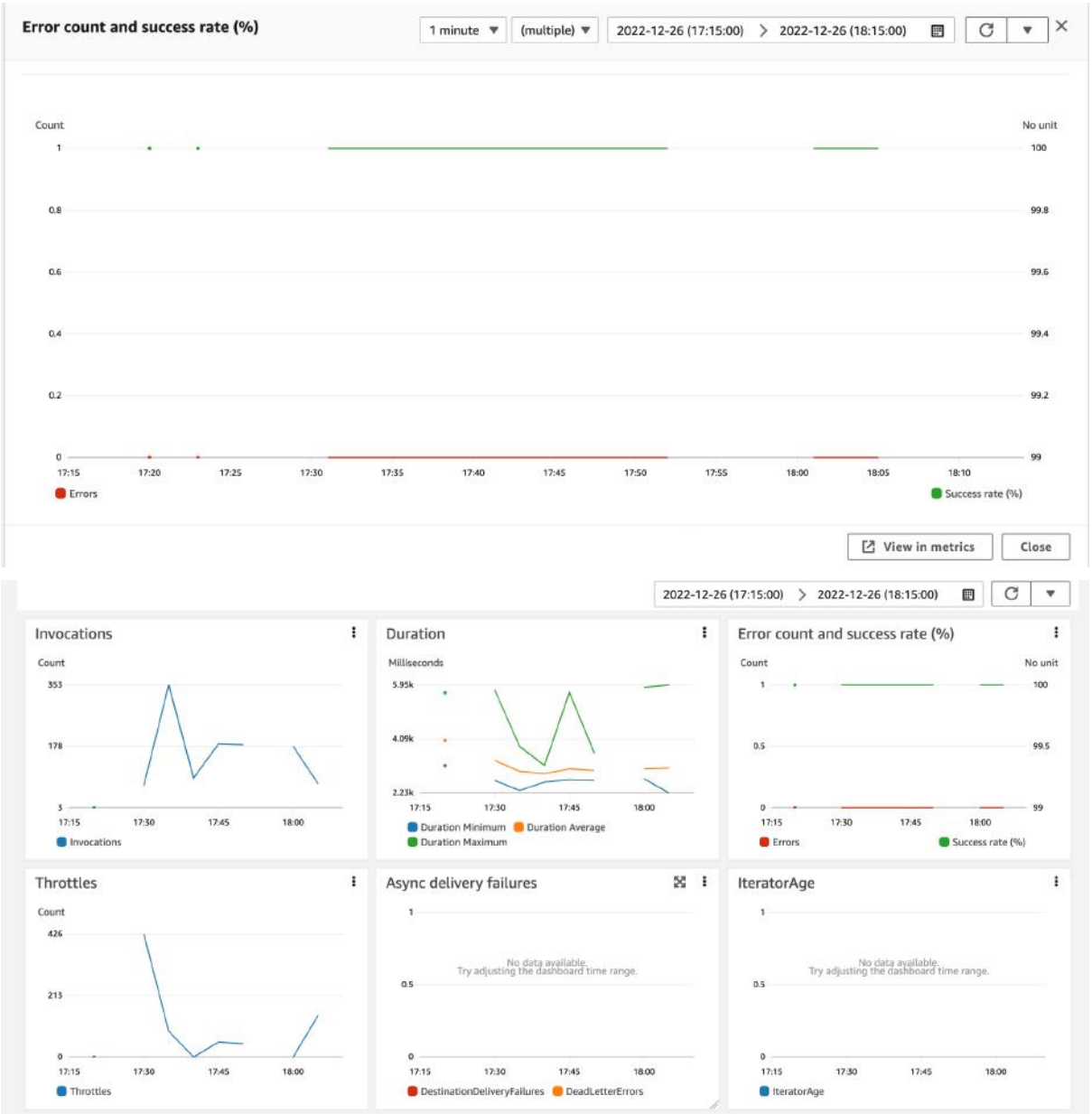
Overall, the system performed well under this scenario, with fast response times and a low error rate. The higher throughput indicates that the system was able to handle the increased load, possibly due to the longer ramp-up period allowing the system to gradually scale up to meet the demands of the 1000 users. This performance is similar to the 500 users with 119 samples and 500 sec ramp-up scenario, suggesting that the system is able to handle high levels of load with a longer ramp-up period.

In conclusion, the stress testing of the cloud-based system using JMeter revealed the performance and scalability characteristics of the system under different levels of load. The results showed that the system was able to handle lower levels of load, with fast response times and low error rates in the 10 and 20 user scenarios. However, as the load increased, the system struggled to keep up, with slower response times, higher error rates, and lower throughput. This was particularly evident in the 500 and 1000 user scenarios with shorter ramp-up periods. On the other hand, the system was able to handle higher levels of load with a longer ramp-up period, as seen in the 500 users with 119 samples and 500 sec ramp-up scenario and the 1000 users with 103 samples and 1000 sec ramp-up scenario. These findings suggest that the system's performance and scalability can be improved by carefully selecting the ramp-up period to allow the system to gradually scale up to meet the demands of the users.

AWS Interface







Conclusion

To conclude, we can say that the architecture of our system leverages the power of various AWS services to deliver a seamless and user-friendly experience. The use of SageMaker, Lambda, and API Gateway allowed us to scale the system up or down depending on our needs, and observe our system's behavior when it is subjected to various stress tests.

The web application, built using React, provided a clean and intuitive interface for users, and the integration with the Spotify Web API allowed users to easily access and listen to recommended songs.

The benefits of using these cloud services can be enlisted as scalability and accessibility. With the ability to easily scale up or down as needed within the free-tier limitations, we are able to save significant resources and expenses compared to traditional on-premises

infrastructure. Also, with machine learning platforms being at hand for us, we have saved significant time to build, train, and deploy our machine learning model to recommend songs.

Repository

For cloud related code please refer to this repository:
https://github.com/umutdenizsenerr/song-recommendation-system-1

For web application related code please refer to this repository:
https://github.com/xltvy/song-recommendation-system

Video Recording

For the screen recording of our project's AWS interface related part please refer to this link:
https://drive.google.com/file/d/1cGiYkZPPB73YeqH-fvwNxZVF7phzoKvc/view?usp=sharing