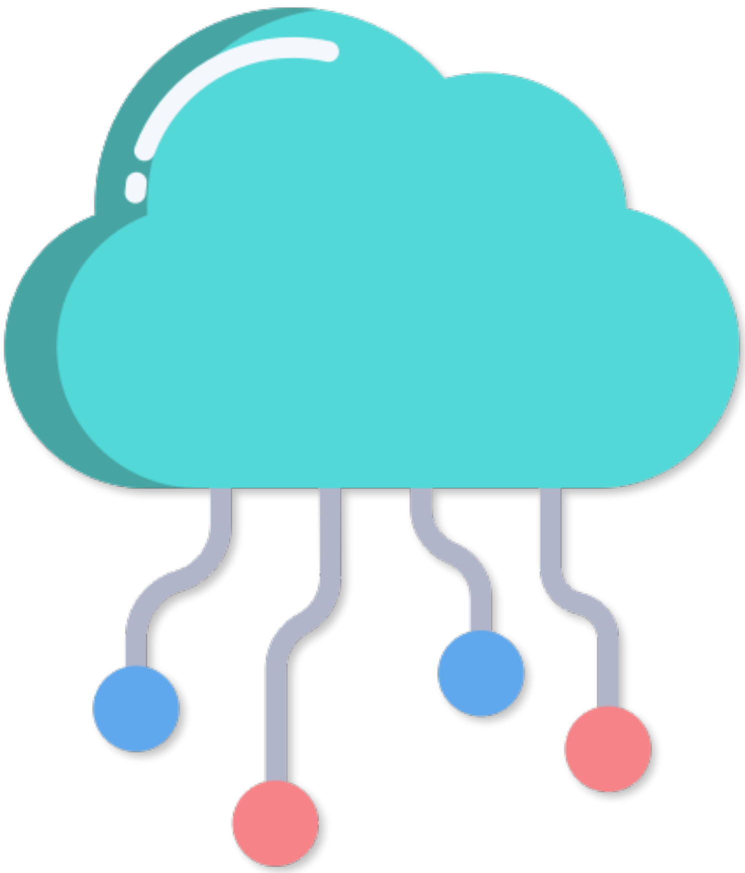
CMPE48A TERM PROJECT

FINAL REPORT



Yavuz Samet Topcuoglu & Ali Kaan Biber

Fall 2022

1. What Is Our Application?

Our application is a simple translation program which takes input as a string which can be provided in several languages, then converts the given text to desired target language. The outputs of the web application are the translated text and the sentiment analysis of the translated text.

Another feature is the program is that it can show the translation history with the show history button.

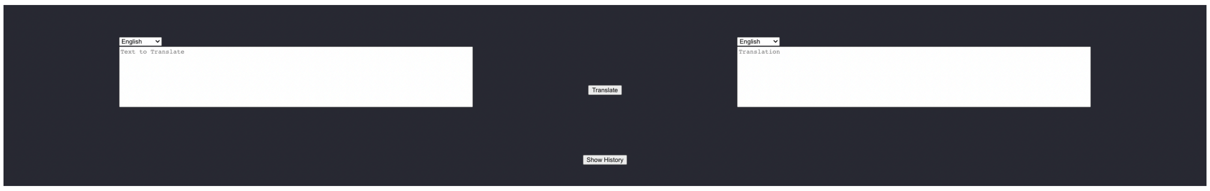
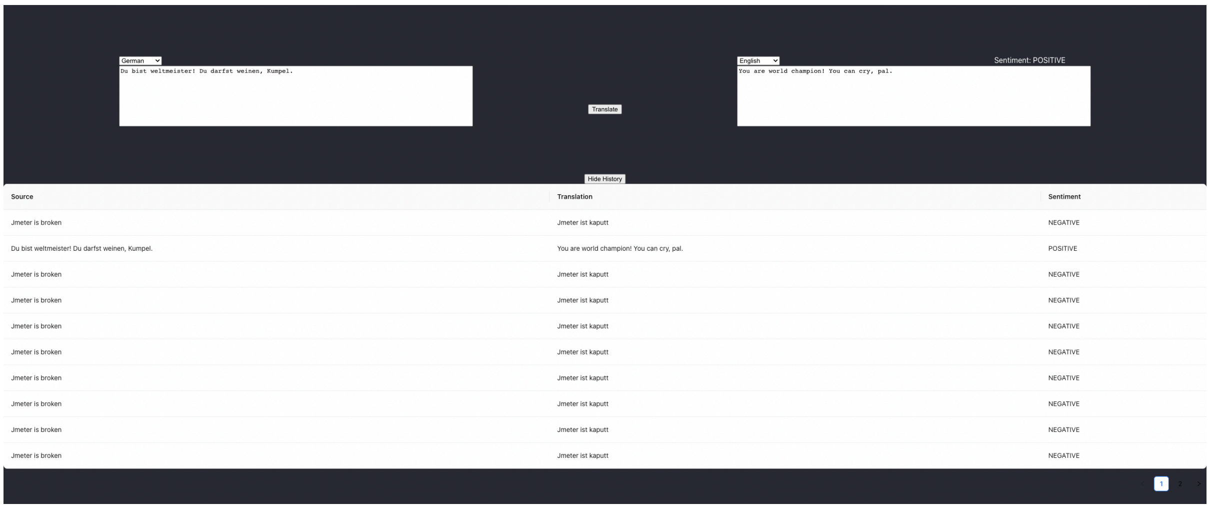


Fig 1. How web application welcomes you.

Here is the simple example of how you can use the web application:



```
Du bist weltmeister! Du darfst weinen, Kumpel.
```

Sentiment: POSITIVE

```
You are world champion! You can cry, pal.
```

Source	Translation	Sentiment
Jmeter is broken	Jmeter ist kaputt	NEGATIVE
Du bist weltmeister! Du darfst weinen, Kumpel.	You are world champion! You can cry, pal.	POSITIVE
Jmeter is broken	Jmeter ist kaputt	NEGATIVE
Jmeter is broken	Jmeter ist kaputt	NEGATIVE
Jmeter is broken	Jmeter ist kaputt	NEGATIVE
Jmeter is broken	Jmeter ist kaputt	NEGATIVE
Jmeter is broken	Jmeter ist kaputt	NEGATIVE
Jmeter is broken	Jmeter ist kaputt	NEGATIVE
Jmeter is broken	Jmeter ist kaputt	NEGATIVE
Jmeter is broken	Jmeter ist kaputt	NEGATIVE

Fig 2. Both translated and show history enabled.

2. The Architecture

FINAL ARCHITECTURE

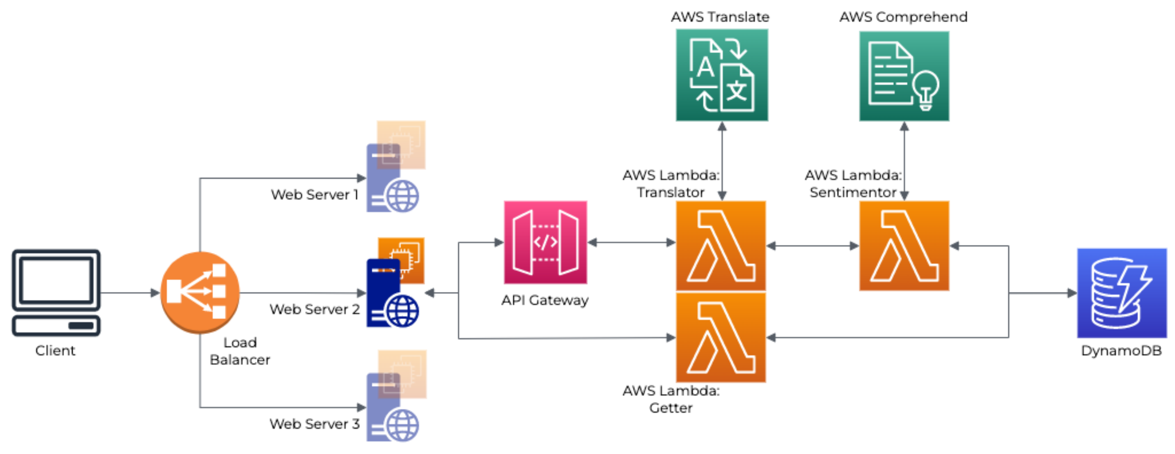


Fig 3. Final Architecture

To create the architecture given above, we have decided to use Amazon Web Services.

The Elastic Load Balancer, divides the load into three web servers which we have created as EC2 instances where our web application is deployed.

The web servers either makes a call to the API Gateway or Getter lambda function depending on the button pressed in the web application. In the initial architecture, there were no API Gateways in the plan since we were planning to enable the function URLs of the AWS lambda functions. Enabling function URLs in the Getter function worked without a problem, however in the Translator function, we had a problem since it requires posting. The API Gateway was the solution we came up with.

When 'Translate' button pressed, the client makes a POST request to API Gateway, which triggers the Translator function. The Translator lambda function uses AWS Translate to make the translation, then invokes the other lambda function called Sentimentor. The Sentimentor uses AWS Comprehend to make the analysis and writes the results to the DynamoDB. Then every function returns the total result to the client by the same path.

When 'Show History' button is pressed, the client makes a GET request to the Getter function, and that function retrieves the entries from the DynamoDB table and returns to the client.

The main problem that we have faced during the initialization of lambda functions was the handling of the role policies of the services. We handled the management of access to AWS resources with IAM.

3. The Testing

3.1. Web Servers

We did not implement any auto-scaling mechanism for this project, however we tested two systems with the same amount of requests. One of the systems contain three Web Servers whereas the other contain just one. We sent 10000 requests to the both systems.

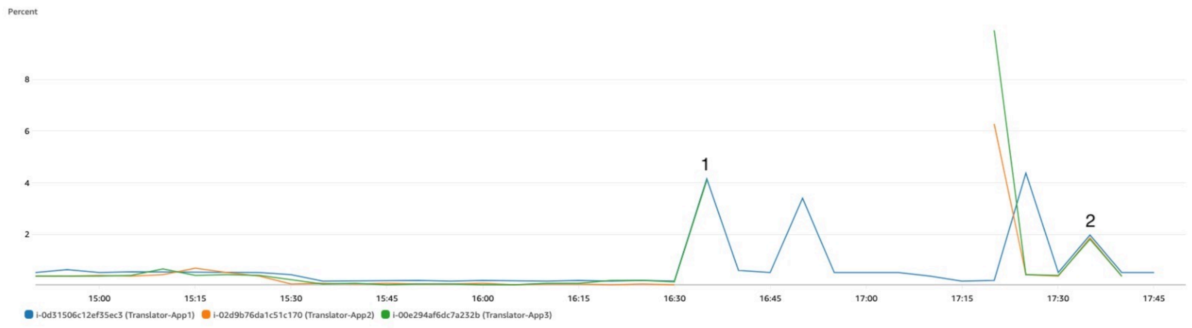


Fig 4. CPU Utilization Comparison

In the figure above, please concentrate on the peaks numbered 1 and 2. The others are irrelevant for our analysis. When we tested the system with only one web server, we had the graph numbered 1, stating the CPU Usage of 4%. However, when we tested the system with three web servers, we had the number 2 where all three web servers has CPU Usage of %2 each.

We can say that the CPU Utilization of the web servers are decreasing the when the load is shared among them.

Note that the reason why the green and orange line starts from significantly high values at some time is that the web servers are deployed in that very moment and the starting of the web servers caused that usage of the CPU. Before our testing, they were stabilized at a normal level where they are idle.

Also, we stored the elapsed time of each concurrent request in a csv file with the help of JMeter. We can see that average elapsed time of the three serviced system is lower than the half of the one serviced system. The average elapsed times are as follows: 324 ms and 671 ms.

3.2. AWS Lambda Functions

There are some limitations to testing that functions since the number of concurrent requests to the AWS Lambda Functions are limited by 10. As can be seen from the following figures:

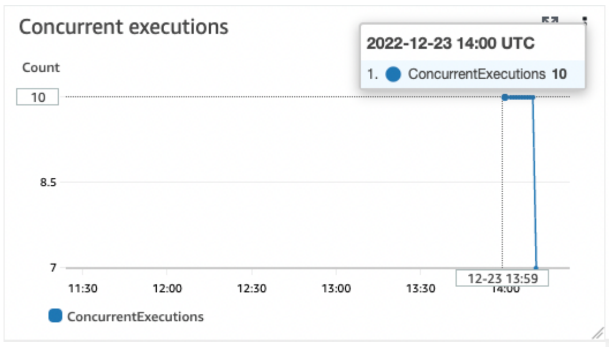


Fig 5. Concurrent Executions

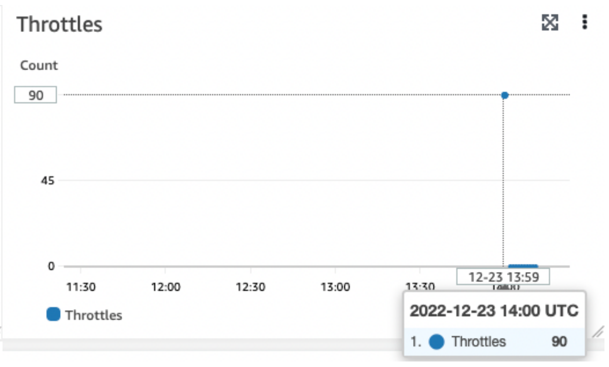


Fig 6. Throttles

When we tried to test a lambda function with 100 concurrent requests, we get the result above stating that the 10 requests are completed, however 90 of them are throttled due to limitation.

Since that was the situation, we tried two things. The first one is to test the lambda functions with one user and ten concurrent users. However, as we imagine, there were no significant change at all in the elapsed time between the function call and the function response. The average elapsed time for the Getter function was changing between 500 ms and 600 ms.

The second is to test lambda functions with ten concurrent users in a loop of 100 times with 1 second separation between concurrent group calls. Note that this configuration can be adjusted from JMeter. As expected, the average elapsed time was again the same. Because calling these functions with at most 10 concurrent users could not make a significant change since it is not a heavy load.

GitHub Links:	AWS Lambda Functions, Web Application
Demo Video:	Demo