DALHOUSIE UNIVERSITY

CSCI 5409 – Cloud Computing

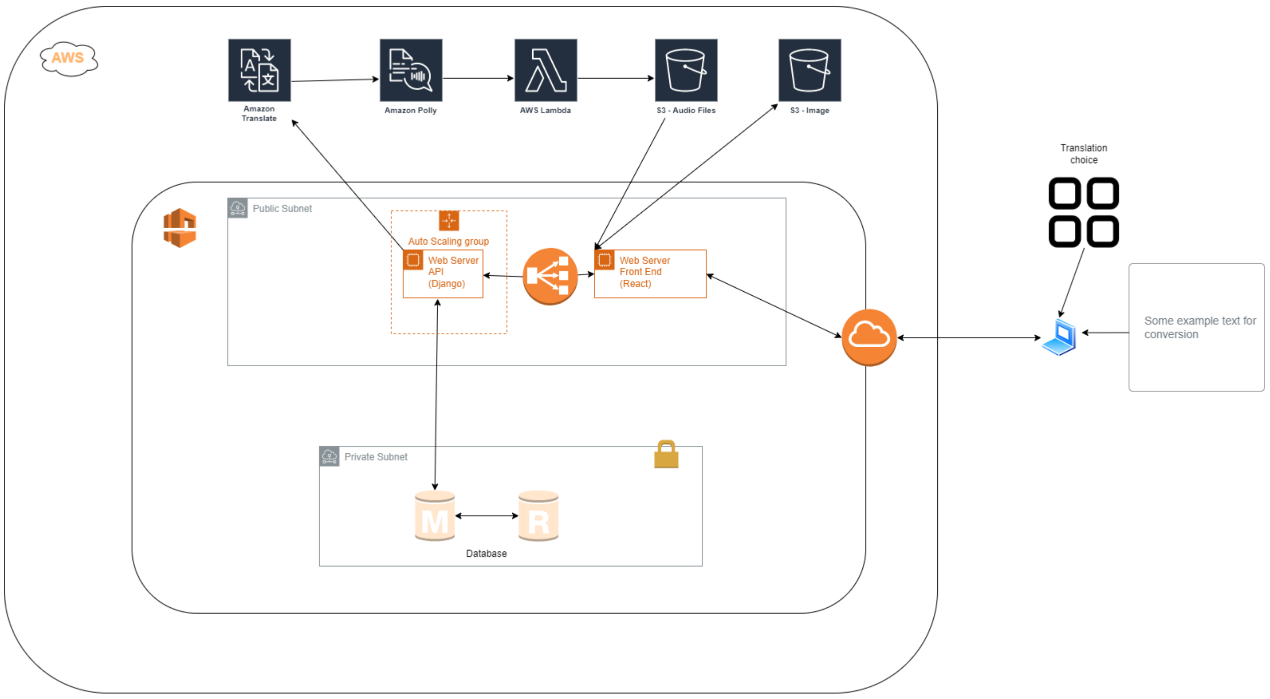
Architecture Critical Analysis and Response

Work done by,

GROUP 31

- **Guturu Rama Mohan Vishnu – B00871849**
- **Aditya Deepak Mahale – B00867619**
- **Sumit Singh – B00882103**

1. The architecture of our project can be described as in the following diagram.



In this document, we are going to discuss the three different flows flowing through the architecture and we are going to discuss all three of them below:

Case A: Registration

If a new user wants to register for our website, he will be directed to the sign-up page and after entering all the required details, the details will be sent to the network gateway present in the Virtual Private Cloud (VPC) in AWS. The network gateway communicates with the front-end web server which is built in ReactJS and the details are passed through this to the Load Balancer present behind the front-end server. The load balancer provides a single endpoint through which the front-end application can connect to the backend application built in Django. Now the details will be passed on to the Redundant Database present in the Private Subnet and all the user details will be stored in the respective table in the database.

Case B: Login

The flow of the login scenario is as similar as the flow of registration. But the only difference in this case is that when an existing user tries to login, the user details are authorized with the details already present and if those are valid credentials, then the redundant database will generate a JWT (JSON Web Token) which will be returned to the network gateway all through the flow again. If a new user who didn’t register first tries to login, the details will be checked with the available user details in the database and if these user details aren’t present, the system will let the user know that they have to register before logging in to the website.

Case C: Translation

This is the scenario where our whole project will be used. A user logs in to the website using their valid credentials, uploads their text which needs to be translated and they will be given a dropdown list of options to choose from, to translate their text into the respective language. The text and the selected option details will pass through the network gateway, to the front-end web server and to the load balancer which allows us to talk to the back-end application. The front-end web server, load balancer and the back-end API are all present in the Public Subnet in AWS. Our backend API is in Auto Scaling group, which communicates with Amazon Translate to translate the given text into the respective selected language. After successfully translating the given text, Amazon Translate will trigger Amazon Polly which takes the translated text as an input and converts it into an audio file in its respective language. After completing the task, Amazon Polly will trigger AWS Lambda which stores the converted audio file in an S3 bucket in AWS. Now, the link of the audio file present in the public S3 bucket will be sent back to the front-end web server which displays the link to the user. The user can access the link and will be able to download the audio file and listen to it.

<u>Components:</u>

We are following three-tier architecture for building our application. Using a layered architecture will help us in creating a scalable application.

<u>Web server (Presentation layer: EC2):</u>

We are hosting our frontend web server on an EC2 instance. The instance will act as an entry point to the application. It will host a react application served by Nginx. To add an additional layer of security, we'll be using security groups.

<u>Application layer (Backend: ELB, Auto scaling groups, EC2):</u>

The application layer will provide API endpoints for the front end. We'll be hosting the Django application on EC2 instances for this layer.

<u>Elastic load balancer:</u>

To improve the workload distribution, we'll deploy an elastic load balancer in front of the API instance. We'll be using an application load balancer to route HTTPS traffic to API EC2 instances in the backend. Using a load balancer ensures that the client request gets routed to the instance with less load to maintain the application performance.

<u>Auto-scaling groups:</u>

To meet the demand during peak leads, we'll be using an auto-scaling group. The EC2 instances in the group will automatically scale based on the demand.

AWS Translate:

The API instance will pass on the text to AWS translate for conversion. The converted text will then be passed on to AWS Polly.

AWS Polly:

AWS Polly will receive the translated text and convert it to an audio file.

AWS Lambda:

AWS Lambda will act as an intermediate trigger point for storing audio files in the S3 bucket.

S3:

S3 bucket will host the audio files.

Database Layer:

We’ll be using Multi-AZ deployment for the database in the backend to handle disaster recovery.

2. Determine which architecture you've learned about in class your group's project is most similar to. List it, and then describe any differences you see between your design and the fundamental or advanced architecture from class.

Since we are using a layered architecture, within two layers of our project we can observe architectures discussed in the class. In the application layer, we are using an elastic load balancer and auto-scaling group to dynamically scale the EC2 instances based on the incoming load. This architecture is similar to dynamic scalability architecture. In our case, the elastic load balancer will act as a listener for scaling instances in the auto-scaling group. The difference between dynamic scalability and our architecture and ours is that we are only scaling instances horizontally instead of scaling vertically. Also, we are not dynamically relocating servers in this layer.

In addition, in the database layer, we'll be using Multi-AZ deployment to manage disaster recovery. This architecture is similar to redundant storage architecture. The data will be replicated in an instance in a different availability zone. However, the replicated data will only be utilized in case of a disaster when the primary node fails. Automatic failover will make the redundant node the primary node. The difference between redundant storage architecture and ours is that we're not replicating data to a different region for increasing availability.

The proposed architecture is a hybrid. In this architecture we are using three fundamental architecture that were taught in the class. This hybrid architecture is a collection of the following architecture

- Workload distribution architecture
- Redundant storage architecture
- Dynamic scaling architecture

This architecture is also derived from dynamic scaling architecture. The reason behind going with these many fundamental architectures is to ensure higher availability, scalability, and fault tolerance. Though we have single point of failure from user interface, we wanted to ensure that users who translate their text once should have hinger availability of data with low latency.

Mostly of the architecture discussed in the class were individual and mostly independent, so this hydride architecture is different from them because of its combination of various fundamental architecture. Having these many architectures incorporated into one enables us to overcome limitations of one another resulting in better performance.

3. As we all know, there is always scope for improvement in life. This is also true in the case of architecture proposed. Even in our architecture we have scope for improvement that can result in better performance, availability, and scaling.

I. <u>Performance:</u>

When I look at all the architecture taught, I feel one of the most powerful and best performing is **cloud balancing architecture**. This architecture ensures that we have our service running all the time, with better performance. Cloud balancing architecture uses special multiple cloud services with load balancing on top of it. This ensures high performance with low latency.

One more improvement that I can think of is adding **dynamic data normalization architecture,** this ensures that de-duplicated data being stored in the database, which ensures no redundant data is stored which improves the database operations.

II. <u>Availability:</u>

One of the best advanced architectures is **cloud balancing architecture**. This ensures high availability across the network all the time. Requests are routed to redundant implementations of the service distributed across multiple clouds by an automated scaling listener. This failover system increases resiliency by allowing for cross-cloud failover.

III. Scaling:

One of the major limitations of workload distribution architecture is spin up time. Spinning up time referred to waiting time for server to start up during scaling. Use of **Resource pooling architecture** could have easily overcome this issue, it also ensures synchronization over the pools.