DALHOUSIE UNIVERSITY

CSCI 5409 – Cloud Computing

Final Project Report

Work done by,

GROUP 31

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

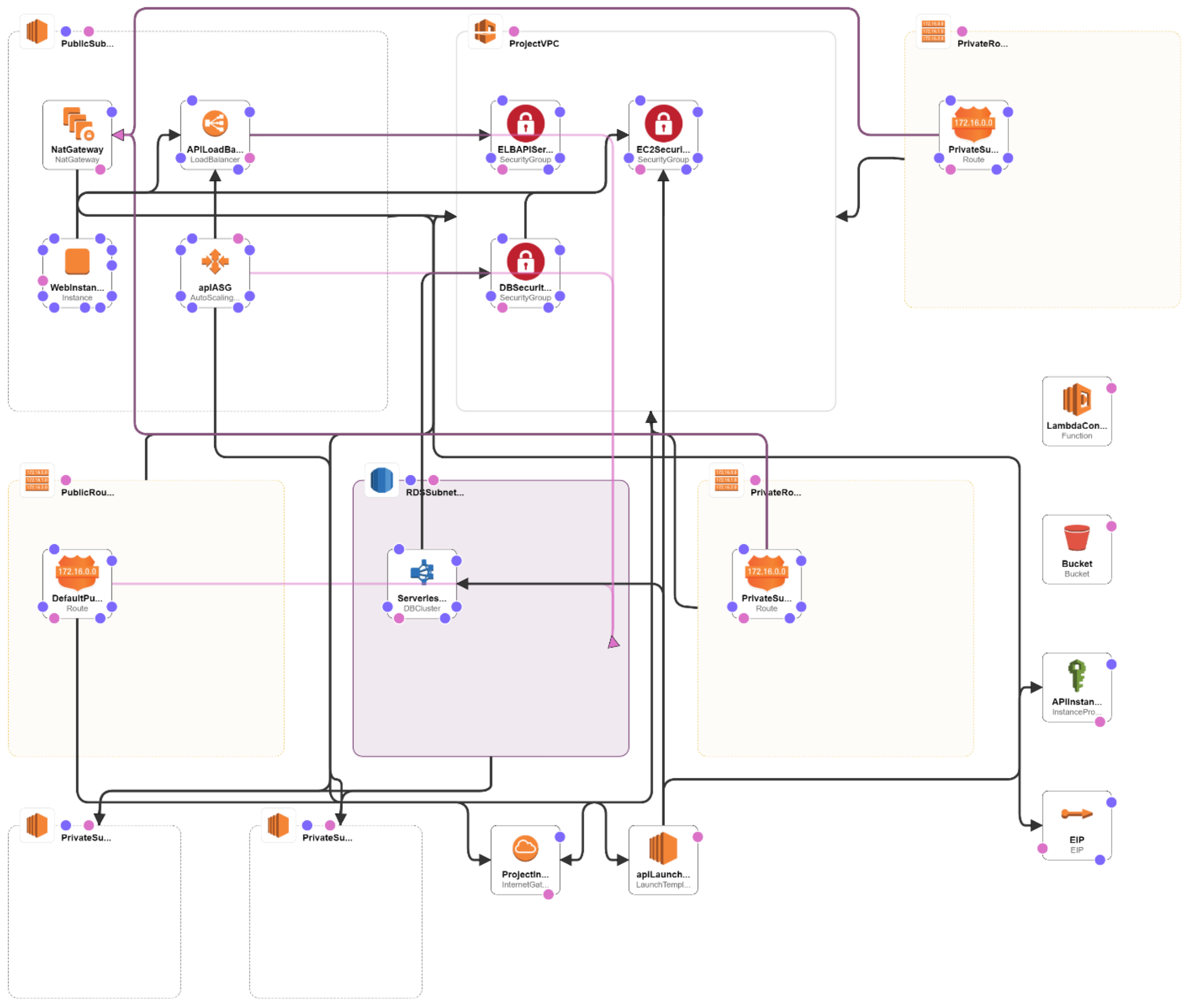
1. Describe your final architecture

(a) How do all of the cloud mechanisms fit together to deliver your application?

Firstly, we will list the services we choose from the list of services available under different categories to complete this project.

Category	Services
General	Polly, Translate
Compute	Amazon EC2, Lambda
Network	AWS Virtual Private Cloud (VPC)
Storage	Amazon S3
Security	Security Groups, NACLs and Subnets in VPC
Cloud Deployment Model	AWS Public Cloud
Cloud Delivery Model	IaaS + FaaS

These are the services that we have used to build our project. Now, the real question of how all of our cloud mechanisms fit together to control our application can be known by the architecture below and also, we will discuss that in a bit detailed way.



Architecture diagram generated by AWS cloudFormation Designer

This is the architecture which was built by us through CloudFormation. As you can see here, there are many things which we have implemented and used. To start with, we would take the input of EnvironmentName, DBUsername and DBUserPassword from the user in the form of parameters. These fields will be used in the CloudFormation for the provisioning of the services. The first service which we create is the Virtual Private Cloud (VPC) named Project VPC. When we create VPC, the AWS automatically creates the NACL for our VPC by default. When it's done, we then create a new Internet Gateway and attach the Internet Gateway with the Project VPC. After this, a Public Subnet is created with

CidrBlock as 10.0.0.0/24 and linked to the Project VPC. The next step is to create a Public Route Table and after linking it to the Project VPC, we added a Default Public Route as 0.0.0.0/0 and also the Internet Gateway which was created above was linked here to the Route Table. Since we already have both the Public Subnet and Public Route Table, the one thing left to do is to assign these both things together. That step can be completed with the help of Public Subnet Route Table Association. Completing this step successfully means that we have a successful public subnet with everything that it would need and now we created two Private Subnets with one of them having the route as 10.0.1.0/24 and the other having 10.0.2.0/24. Creating a route table for each subnet and associating with their respective tables is done as explained in the public subnet part. After having all the 3 subnets, we then created Elastic IP address in the Project VPC domain. A NAT Gateway is now created with Elastic IP address Allocation ID and with the link to Public Subnet. We then linked this NAT Gateway to the two private subnets we created in the previous steps.

By the time this whole process is done, we will have a VPC with all the setup we need. So, the next main step is to create an Serverless DB Cluster to store our data in it. But to create a DB Cluster we would need a security group. So, we created a DB Security Group with the Project VPC reference with the inbound rules of 3306 and outbound rules to talk to anyone and we created this SG before creating the DB Cluster. A Serverless DB Cluster was created with the username and password which was taken as a parameter input from the user, with the configuration we wanted, and it was linked to the DB Security group. But the database part isn’t complete because we also need a Subnet Group for the database and that is done by creating a DB Subnet group with both private subnets of the Project VPC.

After being done with VPC and RDS, the next big thing is to create an EC2 Security Group which allows HTTP and HTTPS and SSH inbound and outbound traffic to the EC2 Instance. The inbound rules are also configured respective to those port numbers. An API Security Group is also being created to enable HTTP access via port number 80 with the Project VPC reference, with the help of creating an API Instance Profile of having the role as LabRole. Then, an API Launch Template is created with the respective instance types and instance profile we created just before this, and it runs a script which was written to get the code from GitHub and run it so that the back end of our application can start and keep running.

An API Load Balancer which depends on the Internet Gateway is being launched here with the public subnet reference and API Security Group. An Auto Scaling Group which again depends on the Internet Gateway attachment is being launched and the Launch Template from before is being used here and also the public subnet and load balancer from previous steps. Next, an S3 bucket is created in order to store the audio files and it is granted public access since any user who uses our application should be able to access their respective audio files.

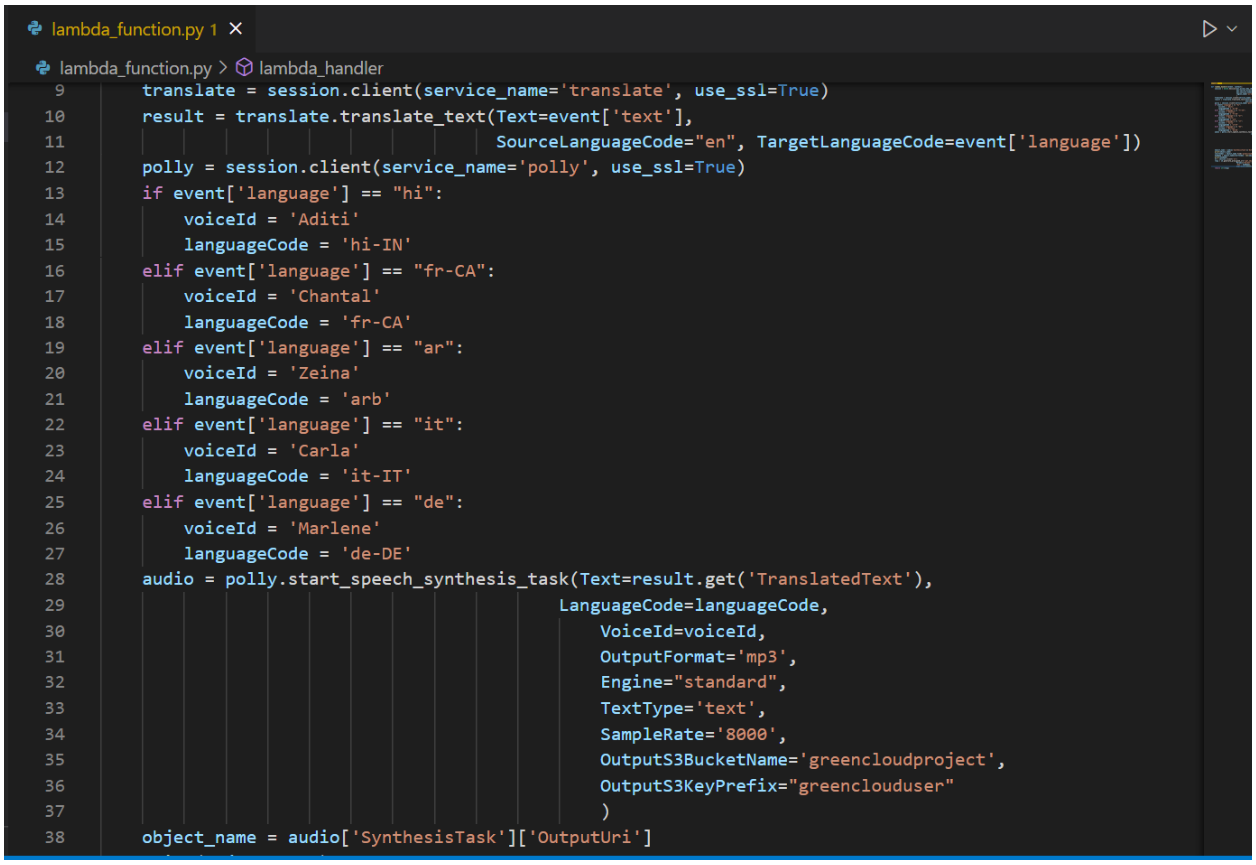
A Lambda function is being launched to handle the code which is followed by an EC2 Instance which depends on the S3 bucket, and it again runs a script which gets the code from GitHub and installs all the dependencies and runs the code in order to start the front end server of our application.

(b) Where is data stored?

We have two types of data; one is user related data and then we have audio file. Data related to user has all the details of the user like email, first name, last name and password and it is stored in serverless RDS DB. Once the user logs in and does their operation of translation and conversion of the text to desired language, it generates an audio file that we are storing in S3 bucket. S3 bucket is protected and has limited access. We are using boto3 S3 bucket generate_presigned_url function to expose proxy URL to audio file which is valid for 100 seconds.

(c) What programming languages did you use and why and what parts of your application required code?

We went forward with React for our front end and Django for backend. As we know React and Django are leading frameworks in Web development and has lot of cool features that make Web development a fun task. We did major coding in creating API to handle user login, registration, user details update and convert. Convert API is used to take user input text and target language and invoke the lambda function using Boto3 that handles the translations, speech conversion, storing audio file in S3 bucket and return the generated audio file URL. By using this URL, the user can access the processed audio file. In the front end, we have user interfaces to register user, login and conversion page. We are also handling data validation on our front-end application. We collect the data entered by user, validate it and call the respective API to handle the required operation in the backend. Logic of translation and Polly (Text to audio conversion) is handled in lambda function. We chose python language for lambda function. We are using Boto3 to call AWS translate and Polly to process the input, store the audio file in S3 bucket and return generated pressigned URL for audio file. In lambda, we get the user input of text and target language, translate the text into desired language and then based on details, we will populate the language code and voice ID which are needed for Polly. Then we call boto3 start_speech_synthesis_task with details like text, voiceID, language code, bucket name, file prefix and output format. This method generates the audio file and stores it in the S3 bucket mentioned. Then we get the file key name and then generate a preassigned URL and pass it to the backend, which return the value to UI where user can click on and gets redirected to new page where audio file plays.

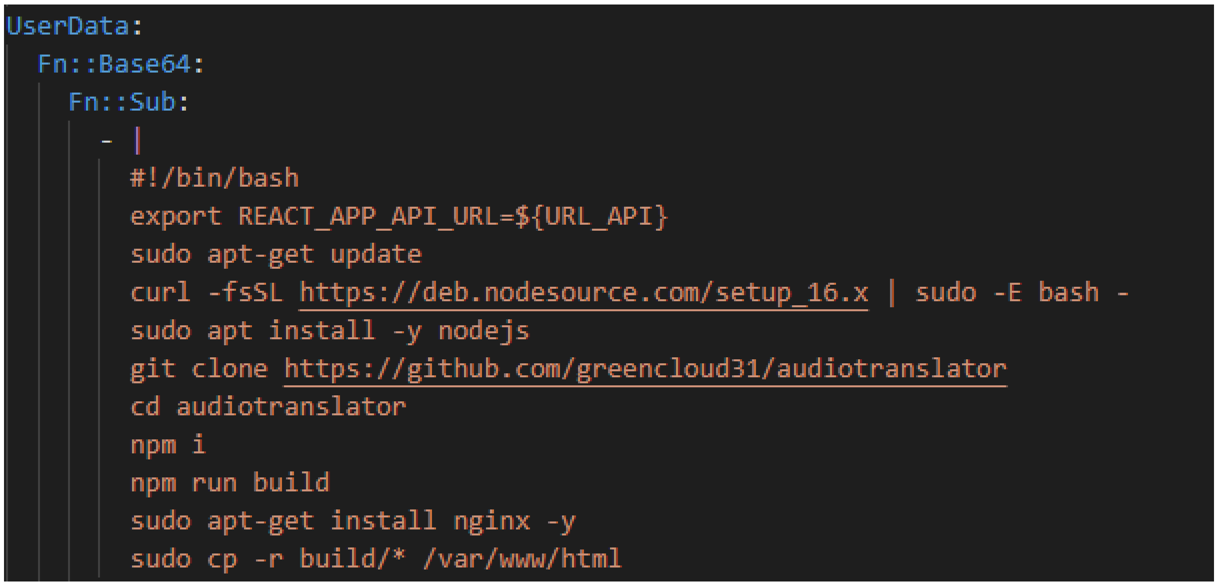


```python
    translate = session.client(service_name='translate', use_ssl=True)
    result = translate.translate_text(Text=event['text'],
                                      SourceLanguageCode="en", TargetLanguageCode=event['language'])
    polly = session.client(service_name='polly', use_ssl=True)
    if event['language'] == "hi":
        voiceId = 'Aditi'
        languageCode = 'hi-IN'
    elif event['language'] == "fr-CA":
        voiceId = 'Chantal'
        languageCode = 'fr-CA'
    elif event['language'] == "ar":
        voiceId = 'Zeina'
        languageCode = 'arb'
    elif event['language'] == "it":
        voiceId = 'Carla'
        languageCode = 'it-IT'
    elif event['language'] == "de":
        voiceId = 'Marlene'
        languageCode = 'de-DE'
    audio = polly.start_speech_synthesis_task(Text=result.get('TranslatedText'),
                                              LanguageCode=languageCode,
                                                  VoiceId=voiceId,
                                                  OutputFormat='mp3',
                                                  Engine="standard",
                                                  TextType='text',
                                                  SampleRate='8000',
                                                  OutputS3BucketName='greencloudproject',
                                                  OutputS3KeyPrefix="greenclouduser"
                                                  )
    object_name = audio['SynthesisTask']['OutputUri']
```

(d) How is your system deployed to the cloud?

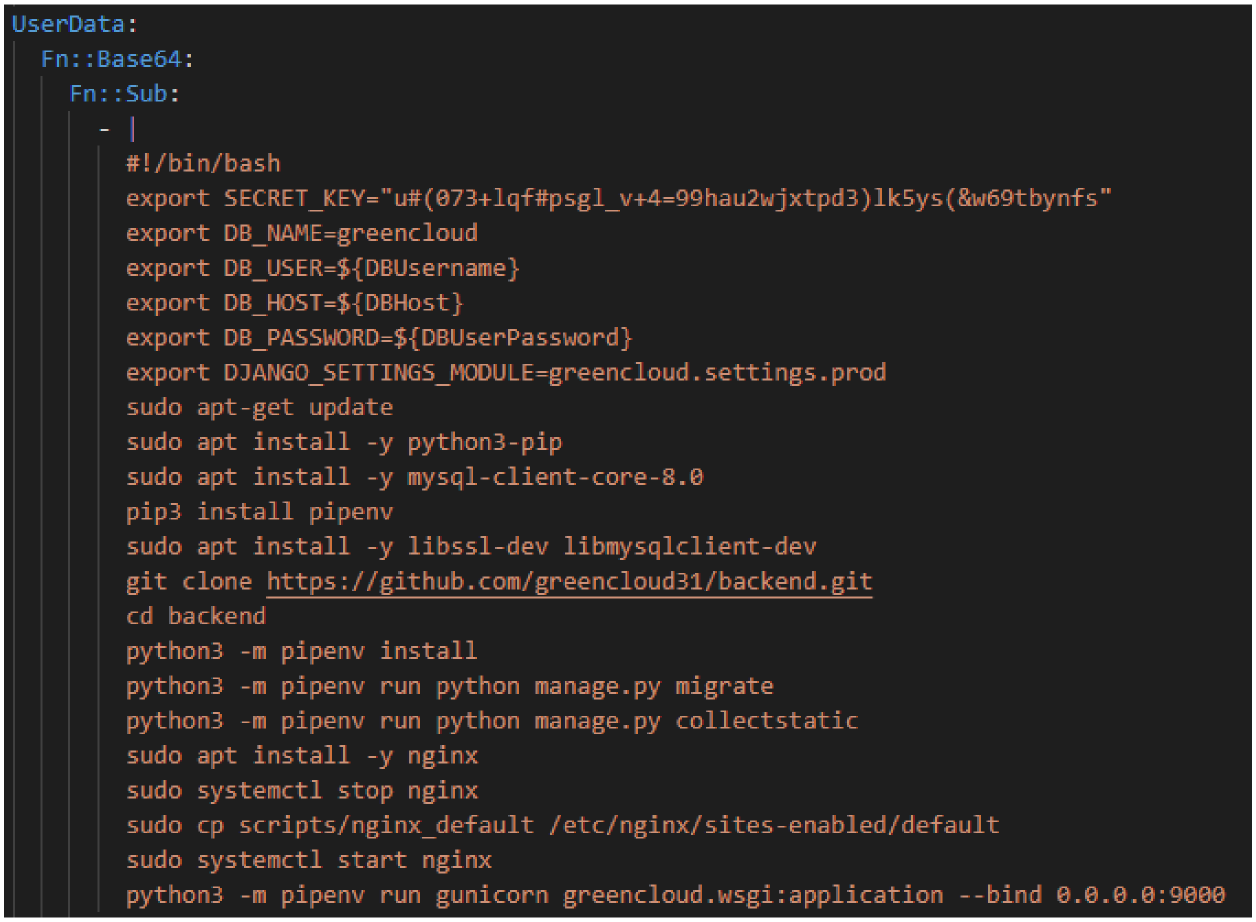
We are deploying our system into the AWS cloud using the CloudFormation template which we built. The template which we built will deploy all the AWS services into AWS when we upload the stack, and after building the stack and completing building the services, the CF template will run the script that we wrote in the template and the script will start both back-end and front-end and it will start our application. The script actually clones the code from the github repo in which we updated the final code, and then it will install all the dependencies required and when it is ready to be executed, it will run the code.

A glimpse of the script from CF which starts our front-end server is:



```
UserData:
  Fn::Base64:
    Fn::Sub:
      - |
        #!/bin/bash
        export REACT_APP_API_URL=${URL_API}
        sudo apt-get update
        curl -fsSL https://deb.nodesource.com/setup_16.x | sudo -E bash -
        sudo apt install -y nodejs
        git clone https://github.com/greencloud31/audiotranslator
        cd audiotranslator
        npm i
        npm run build
        sudo apt-get install nginx -y
        sudo cp -r build/* /var/www/html
```

A glimpse of the script from CF which starts our back-end server is:



```
UserData:
  Fn::Base64:
    Fn::Sub:
      - |
        #!/bin/bash
        export SECRET_KEY="u#(073+lqf#psgl_v+4=99hau2wjxtpd3)lk5ys(&w69tbynfs"
        export DB_NAME=greencloud
        export DB_USER=${DBUsername}
        export DB_HOST=${DBHost}
        export DB_PASSWORD=${DBUserPassword}
        export DJANGO_SETTINGS_MODULE=greencloud.settings.prod
        sudo apt-get update
        sudo apt install -y python3-pip
        sudo apt install -y mysql-client-core-8.0
        pip3 install pipenv
        sudo apt install -y libssl-dev libmysqlclient-dev
        git clone https://github.com/greencloud31/backend.git
        cd backend
        python3 -m pipenv install
        python3 -m pipenv run python manage.py migrate
        python3 -m pipenv run python manage.py collectstatic
        sudo apt install -y nginx
        sudo systemctl stop nginx
        sudo cp scripts/nginx_default /etc/nginx/sites-enabled/default
        sudo systemctl start nginx
        python3 -m pipenv run gunicorn greencloud.wsgi:application --bind 0.0.0.0:9000
```

2. If your final architecture differs from your original project proposal, explain why that is. What did you learn that made you change technologies or approaches? If your final architecture is same as your original project proposal, reflect now on whether there is something you could have implemented with what you know now that would have resulted in a better architecture or a more cost-efficient product?

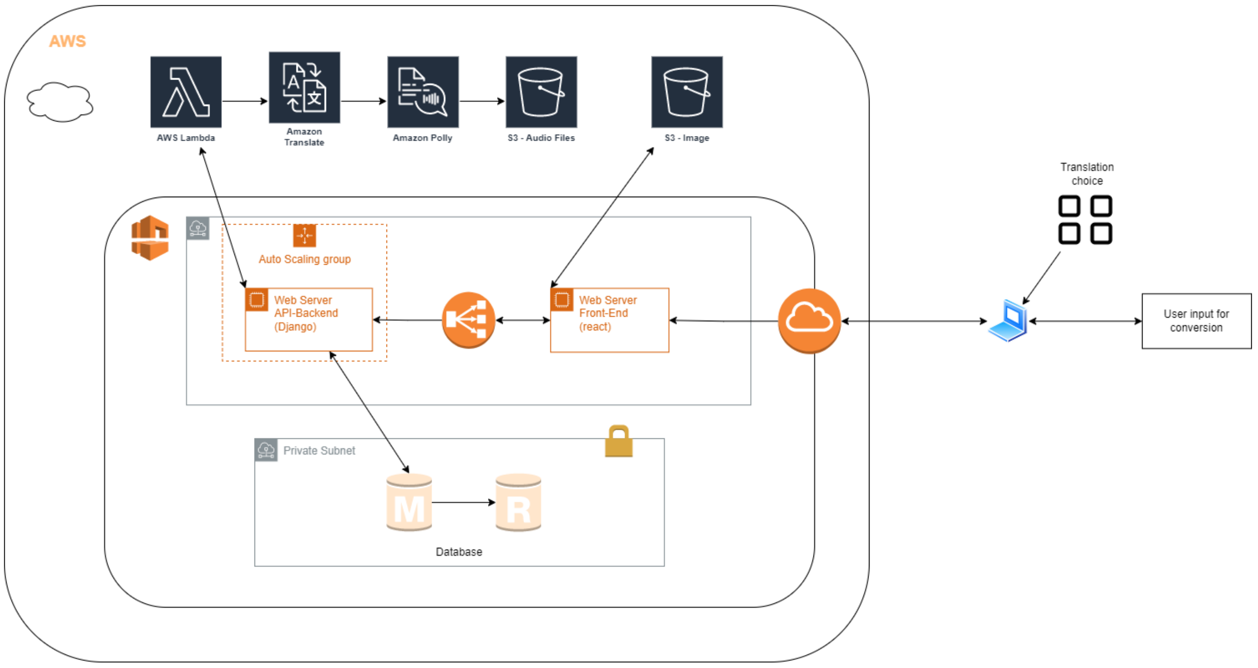
We used a serverless aurora DB instead of deploying a normal RDS instance. The reason for choosing serverless DB is that we realized that in our application, we are utilizing the database only for the purpose of registration. The authentication happens at the API layer using JWT. Hence, the application wouldn't require an always-available database. The serverless DB saved the cost significantly in our application.

We had to modify our architecture for the accommodation of serverless RDS in the private subnets. The private subnet does not get access to the internet and can only communicate with the instances in the same VPC. Hence, we had to create an additional NAT gateway so that instances in the private subnet get access to download resources from the internet, but the same instance does not get the capability to communicate with any other resources other than the VPC. Further, to associate the NAT gateway with the VPC, we created a route table and subnet route table association.

Since we created a VPC from scratch, for the public subnet, we created a route table and subnet route table association for associating the public subnet with the internet gateway.

In our application, our initial plan was to utilize lambda to store the generated audio file in the S3 bucket. However, we modified our architecture so that the code for calling Translate, Polly, and storing an audio file in S3 resided in the lambda function itself. To accommodate this, we also had to modify the EC2 instances in the API layer so that they have permission to call the lambda function without passing a secret key or token. We created an instance profile and attached it to the instance.

In our initial proposal document, we didn't have an elastic load balancer and auto-scaling group. To increase the application availability, we included these two components in our architecture.



Final modified architecture

3. How would your application evolve if you were to continue deployment? What features might you add next and which cloud mechanisms would you use to implement those features?

In our current implementation, we are not embedding the audio player on the website. Instead, we are redirecting the user to an S3 link with the audio file in it. If we continue the development, we'll embed the player on our website and directly stream the audio from the S3 bucket.

Since AWS academy has some issues with IAM roles, we didn't define proper security measures for the resources. For example, in the API layer, the EC2 instances have instance profiles with LabRole assigned to them. However, the purpose of the EC2 instance is just to access the lambda function for the execution. Hence, we'll only assign a role to the EC2 instance with the permission to execute the lambda function and nothing more than that. This will enforce the principle of least privilege in those instances.

In our current implementation, we have configured high availability configuration only for the backend API instances using auto-scaling group and elastic load balancer. However, the frontend EC2 instance will act as a single point of failure in this setup. To avoid that, we'll also create an autoscaling group for our frontend layer along with an elastic load balancer.

As an alternative configuration for the frontend deployment, we are also thinking about deploying the bundled frontend react application in an S3 bucket instead of deploying it on an EC2 instance. We’ll reduce the latency of the application by caching the S3 content on the CloudFront edge locations.

We’ll serve more static content such as images in an S3 bucket.

CloudFormation repository link:

https://git.cs.dal.ca/mahale/group31-cf-template

Frontend repository link:

https://github.com/greencloud31/audiotranslator

Backend repository link:

https://github.com/greencloud31/backend

References:

[1] "Getting started — djoser 2.0.1 documentation", Djoser.readthedocs.io, 2022. [Online]. Available: https://djoser.readthedocs.io/en/latest/getting_started.html. [Accessed: 05- Mar- 2022]

[2] "Simple JWT — Simple JWT 5.1.0.post2+g3fc9110 documentation", Django-rest-framework-simplejwt.readthedocs.io, 2022. [Online]. Available: https://django-rest-framework-simplejwt.readthedocs.io/en/latest/. [Accessed: 18- Mar- 2022]

[3] T. Christie, "Home - Django REST framework", Django-rest-framework.org, 2022. [Online]. Available: https://www.django-rest-framework.org/. [Accessed: 20- Mar- 2022]

[4] R. more, "How to Extend Django User Model", Simple is Better Than Complex, 2022. [Online]. Available: https://simpleisbetterthancomplex.com/tutorial/2016/07/22/how-to-extend-django-user-model.html. [Accessed: 21- Mar- 2022]

[5] "GitHub - adityadmahale/adopt: Indoor plants adoption website(Frontend - R eact.js)", GitHub, 2022. [Online]. Available: https://github.com/adityadmahale/adopt. [Accessed: 03- Feb- 2022]

[6] "The Ultimate Django Series: Part 2", Codewithmosh.com, 2022. [Online]. Available: https://codewithmosh.com/p/the-ultimate-django-part2. [Accessed: 12- Mar- 2022]