Real Time Podcast Translating, Transcribing, and Recommendation System

Daniel Cha
Columbia University
dc3765@columbia.edu

Serrana Aguirregaray
Columbia University
sa4117@columbia.edu

Yizhi Tang
Columbia University
yt2822@columbia.edu

***Abstract*—We have created a real-time podcast translating, transcribing, and recommendation system that can scale to millions of users. Specifically, users are able to upload podcasts files, share it with other users, and listen to other people's audio files through the recommendation system. They are also able to translate foreign podcasts to another language of their choice. Section one of this paper gives an introduction to the problem that we are trying to solve; section two explains the architecture of our large-scaled system; section three explains some key design details whereas section four gives implementation details and code structures. In the end, the result and conclusion section, we summarize this paper by showing you the impacts and things.**

***Index Terms*—Cloud Computing, Amazon Web Services, DynamoDB, Lambda Function, Simple Storage Service, CloudFormation, CodePipelines, CloudFront, Amazon Kinesis, SQS, SNS, Amazon Lex, Amazon Polly, Amazon Trascribe, Amazon Translate, Databases, React, JavaScript, Python, Machine Learning, Natural Language Processing, Recommendation System.**

I. Introduction

A podcast is a series of spoken word, audio episodes, focused on a particular topics such as a romantic story, a scientific introduction to computers, etc. One current problem in the podcast industry is the imbalance of language. For instance, large majority of the educational podcast content is in English or Spanish versus Chinese, Korean, or other minor languages. Our project aims to solve this problem by providing user a transcribing and translating tool. Specifically, users can upload a podcast in any language, and convert it into the language they are able to understand. Users are also able to share podcasts to the public, and see others' podcasts through our recommendation system. With recent progress in machine learning technologies, the quality of the translated audio files are excellent. We are able to keep the voice timbre the same as the original speaker but in another language. With the help of cloud infrastructures specifically Amazon Web Services, we are able to scale our podcast upload, transcribe, and recommendation system to millions of users.

II. Architecture

The architecture shown in Figure 1 comprises AWS services integrated to process audio files uploaded by users, perform transcription, translation, custom voice generation, storage, and user management functionalities.

A. Audio Processing Workflow

The audio processing workflow involves several steps:

1) **Audio Input:** Users upload audio files to the corresponding bucket in Amazon S3.
2) **Transcription:** An AWS Lambda function (Transcription Lambda) is triggered by an SQS event. This SQS event is triggered by the Upload Lambda which is in charge of updating DynamoDB. Once DynamoDB is ready, a message is pushed to SQS to start the transcription process using Amazon Transcribe. Results are stored in the transcription folder within S3 and the second lambda (Translation lamba) is triggered.
3) **Translation:** Translated output from Transcribe is stored in S3. Amazon Translate is used for the translarion.
4) **Custom Voice Generation:** Amazon Polly generates a custom voice audio based on translated text. Results are stored in an S3 bucket with presigned URLs in DynamoDB, triggering an SNS notification upon completion.

B. Frontend Integration

The frontend, deployed via Amazon CloudFront and S3, enables users to interact by uploading audio files, selecting source/target languages, and providing email addresses. User-uploaded files are stored in S3 with metadata via API Gateway.

C. User Services

AWS Cognito manages user accounts, authentication, and authorization. The interaction between user request and the server is handled with API Gateway. The audio files with metadata are stored in Amazon S3 and all user data and audio data for user services and recommendations are stored in DynamoDB, facilitating functionalities like making files public, search, likes, and interactions. The AWS Lambda and Amazon SQS are used to trigger other lambda functions or to send data necessary for function implementation.

D. Recommendation Services

Lambda function(LF1) is triggered upon user-uploaded audio files, storing metadata in two DynamoDB tables. The first DynamoDB(D1) maintains user-specific audio file data, while the second DynamoDB(D2) categorizes audio files by labels. Upon a user liking an audio file, lambda function(LF2) is activated via API Gateway, updating user preferences in a

third DynamoDB (D3) based on audio file labels and logging viewed files in a fourth DynamoDB (D4) to refine recommendations. Finally, lambda function(LF3), also triggered via API Gateway, assembles personalized audio file recommendations for the user by cross-referencing user preferences with available audio files.

E. Overall Integration

The proposed architecture demonstrates a seamless integration of AWS services to process audio files, manage user interactions, and ensure efficient audio processing and storage. By leveraging these services, the system achieves scalability, reliability, and ease of maintenance.

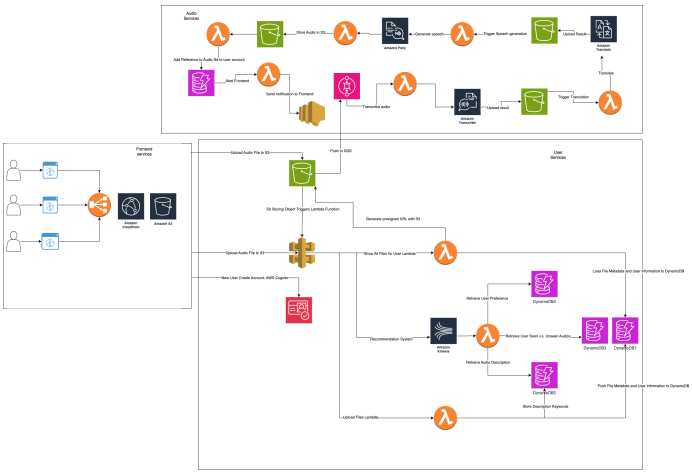

Fig. 1. Architecture

III. Key Design Details

A. Audio Services

The architecture seamlessly manages the automated processing of audio files utilizing various AWS services in a structured workflow. Amazon S3 acts as the primary repository for audio files stored within the 'audio.files.bucket'. Upon upload, these files undergo a series of processing stages facilitated by Amazon Transcribe, Translate, Polly, SQS, SNS, and DynamoDB.

Uploaded audio files trigger a Transcription Lambda function, updating user-specific audio file information in DynamoDB and queueing a message in SQS. This message includes the audio file path and necessary details for subsequent stages.

A dedicated Transcription Lambda, triggered by the SQS message, initiates an Amazon Transcribe job to convert the audio to text. The resulting transcripts are stored back into the S3 bucket under the 'transcription' folder. Post-transcription, the process seamlessly progresses to translation.

Translated text retrieval from S3 triggers the Amazon Translate service. The translated output is stored within a defined path in an S3 bucket.

Subsequently, the translated text undergoes custom voice synthesis through Amazon Polly. This phase involves generating custom voice audio based on the translated text. The resulting audio files are stored within the 'output.audio.files.bucket' in specific paths.

During this voice generation phase, presigned URLs are generated and stored in DynamoDB, corresponding to the relevant records. These presigned URLs facilitate direct access to the generated custom voice audio files.

Amazon SQS efficiently handles asynchronous message flow between stages, orchestrating the execution of specific Lambda functions. Simultaneously, Amazon SNS dispatches real-time notifications to the frontend upon the completion of the custom voice audio generation.

The architecture is adept at scaling without workflow interruptions. Utilizing SQS's queue management ensures consistent, reliable processing, even during heightened workloads. In summary, this architecture presents an automated, versatile, and scalable solution for processing multilingual audio content, seamlessly integrating AWS services to convert, translate, synthesize, and notify frontend applications efficiently.

B. User Services

The first page for the frontend is the login authentication managed by AWS Cognito. The user can create a new account or login with an account already created. After the login, users can upload audio file along with the metadata, such as source and target language, users email address, the option to make it public or not, and the description about the file. When the user upload the file, the audio file with the metadata is stored in the S3 via API Gateway. When a file is stored in S3, it triggers the lambda function. The lambda function fetches metadata from the S3 and store the audio file and the metadata in the DynamoDB. When this is done, it sends a SQS queue to the lambda function which starts the audio services in section A, along with necessary data such as S3 bucket name, audio file name, DynamoDB table name. These data are later used in the audio services for transcribe and translation.

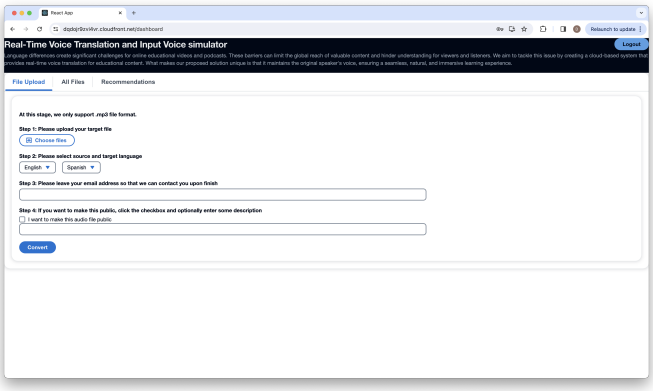

Fig. 2. Upload files example

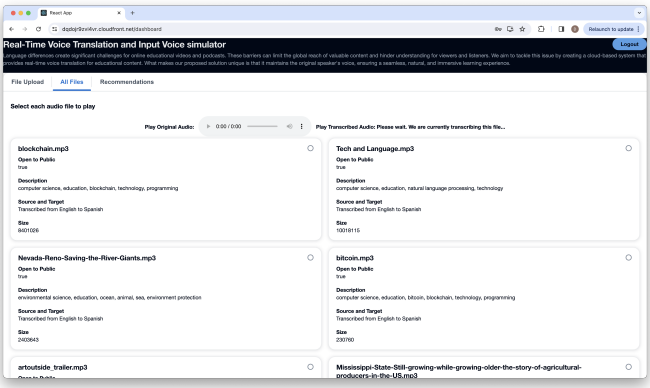

Fig. 3. Showing all files example

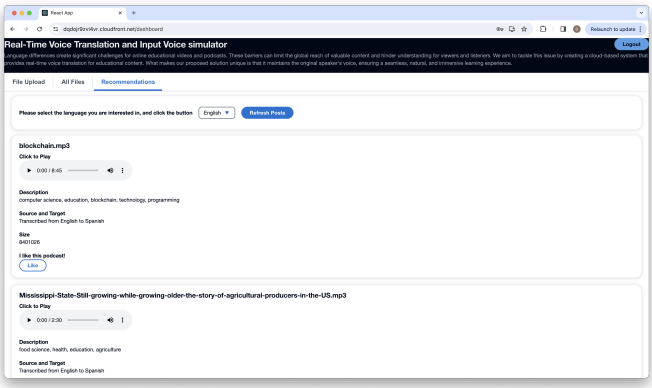

Fig. 4. Recommendation example

C. Recommendation Services

When files are uploaded by the user, the lambda function(LF1) is triggered, storing file information to two different DynamoDB. The first DynamoDB(D1) stores user information along with their audio files, with username as the primary key. Under each usernames, there is a list of audio files with key as the audio file name and the value as the audio file metadata such as file size, file name, source language, target language and etc. The second DynamoDB(D2) stores only name of the audio files based on their labels. The primary key is the label such as 'discrimination', 'computer science', 'math' and the value is the name of the audio file.

On the 'Recommendation' tab from the frontend, user can see a list of existing audio files in a grid view. When user clicks the 'like' button for a specific audio file, two actions are triggered. First, a lambda function(LF2) is triggered through API Gateway. LF2 get the labels attached to the liked audio file from the D2 and update the user preference to the DynamoDB(D3) according to the labels. D3 has the username as the primary key and under the username, there are list of labels that matches user preferences. The second action that is triggered from clicking the 'like' button is updating DynamoDB(D4), storing file names that each user has seen. This step was introduced to prevent the same file from keep appearing on the top of the recommendation. When the user send a request to show the list of recommendation of the audio files through the frontend, the lambda function(LF3) is triggered via API Gateway. The function shows the list of audio files in three steps. First, LF3 checks user preference through getting data from D3. Then, it gets audio file with the corresponding label from D2. Finally, it shows the list of audio files to the frontend.

IV. Code Structure

The system architecture relies on AWS Lambda functions to orchestrate the audio processing workflow, utilizing various AWS services for transcription, translation, and custom voice generation, as well as user and recommendation services.

Transcription Lambda Function

The Transcription Lambda, triggered by S3 events upon user audio file uploads, employs Amazon Transcribe for transcribing audio content. Its functionalities encompass:

- Event Handling and Retrieval: Processes the event data to extract necessary information like bucket, key, and metadata of the uploaded audio file.
- Transcription Initiation: Utilizes Amazon Transcribe service to start a transcription job, monitoring its status until completion.
- Result Handling and Storage: Retrieves the transcription output and stores the text in an S3 bucket dedicated for transcription results.
- Triggering Subsequent Function: Asynchronously invokes the Translation Lambda, passing pertinent payload information for further processing.

Translation Lambda Function

The Translation Lambda receives transcription data from the Transcription Lambda and performs language translation using Amazon Translate. Its functionalities encompass:

- Data Extraction and Translation: Extracts the transcription text from S3, translates it into the desired language using Amazon Translate.
- Translated Text Storage: Saves the translated text in a designated S3 bucket for translation results.
- Invoking Voice Generation Function: Invokes the Voice Generation Lambda asynchronously, passing necessary payload information for generating custom voice audio.

Voice Generation Lambda Function

The Voice Generation Lambda, triggered by the Translation Lambda, utilizes Amazon Polly to create custom voice audio. Its functionalities encompass:

- Retrieval of Translated Text: Retrieves translated text from the S3 bucket, synthesizes speech using Amazon Polly to generate custom voice audio.

- Audio Storage and Metadata Handling: Stores the generated audio in a designated S3 bucket and updates metadata in DynamoDB with a presigned URL for efficient access.
- Notification Triggering: Sends a notification using SNS, signaling the completion of custom voice audio generation to the frontend application.

Upload File Lambda

Upload File Lambda, triggered by the S3 data insertion event, stores file data to the DynamoDB and triggers the Transcription Lambda.

- Storing in DynamoDB: The Upload File Lambda stores audio file along with file metadata into two different DynamoDBs, the D1 and D2.
- Triggering Transcription Lambda: It triggers Transcription Lambda to start the audio processing workflow by sending SQS message to the Lambda function.

Update User Like Lambda

Update User Like Lambda, triggered when user clicks the 'Like' button from the frontend, updates user preference and seen files for the user.

- Update user preference: The User Like Lambda update user preferences with the corresponding audio file labels by updating DynamoDB(D3).
- Update seen files: It updates the files that user have already seen by inserting data to DynamoDB(D4).

Recommend Lambda

The Recommend Lambda, triggered when user clicks the 'Recommendation' tab from the frontend, show all audio files recommended by the backend.

- Recommendation: The Recommend Lambda returns a list of the audio files that match the user preference by getting data from the DynamoDB(D3). Audio files are later shown in the frontend through the API Gateway.

Show User File Lambda

Show User File Lambda, triggered when user clicks the 'All files' tab from the frontend, shows all audio files the user have.

- Showing user's audio files: The Show User File Lambda returns a list of the audio files that the user have uploaded. Audio files are later shown in the frontend through the API Gateway

The orchestration of these Lambda functions within the AWS ecosystem showcases an efficient workflow for audio processing, integrating AWS services for transcription, translation, and voice generation tasks seamlessly.

V. Result and Changes

Our project goal was to construct a system of real-time voice translation using diverse Amazon Web Services. We designed and implemented a system that transcribes, translates, and synthesizes audio using AWS services, providing a seamless

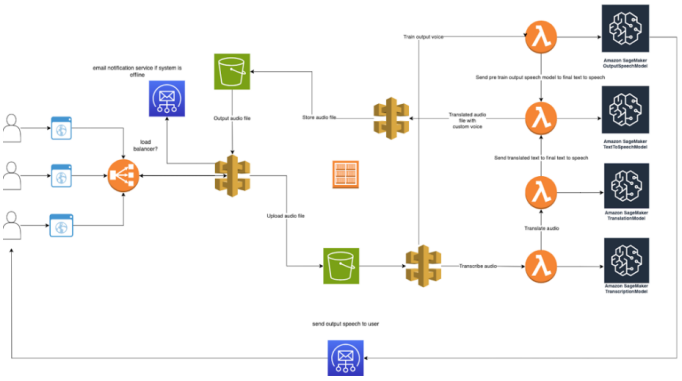

Fig. 5. Original Architecture

end-to-end solution. While we have done the job successfully, there were some major changes along the project.

The Figure 5 was the first architecture designed after consultation among our team members. However, this architecture had several issues. First, the original architecture had a more tightly coupled system where services might be excessively dependent on each other's availability. Second, the earlier architecture used multiple machine learning models for each user. Maintaining separate models for each user requires significantly more computational resources, storage, and memory. This could lead to higher costs and more complex infrastructure needs. Also, ensuring consistent performance across multiple user-specific models was difficult since some models might perform better than others due to differences in the amount and type of training data. To address these issues we have revised the architecture to Figure 6.

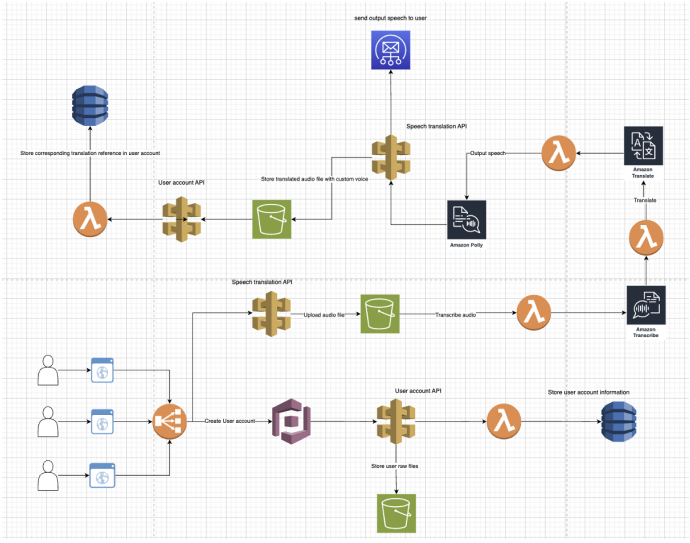

Fig. 6. Intermediate Architecture

The Figure 6 shows a more advanced architecture than the former one with some major changes. Originally, we had a more tightly coupled system where services might be excessively dependent on each other's availability. We

have introduced a greater variety of intermediate services at each stage, allowing us to resolve issues with minimal code modifications should problems arise. Additionally, the process of testing the system's functionality has been made more efficient by breaking it down into smaller, discrete services that facilitate smoother debugging. Also, in the previous architecture, we had another problem of using multiple machine learning models for each user. We have addressed this issue by going through one pipeline of pre-trained machine learning models, with three processes of transcribing, translating, and going through speech generation. This architecture is more efficient in terms of computation and storage, reducing the overall cost and complexity of the system compared to the precedent architecture. It is also easier to manage, update, and deploy than to handle many individual models for every users. Also, as users increase, the system does not need to allocate additional models, making it highly scalable. However, we felt that there were too few features available for users, so we added functions for user convenience and were able to create the final architecture.

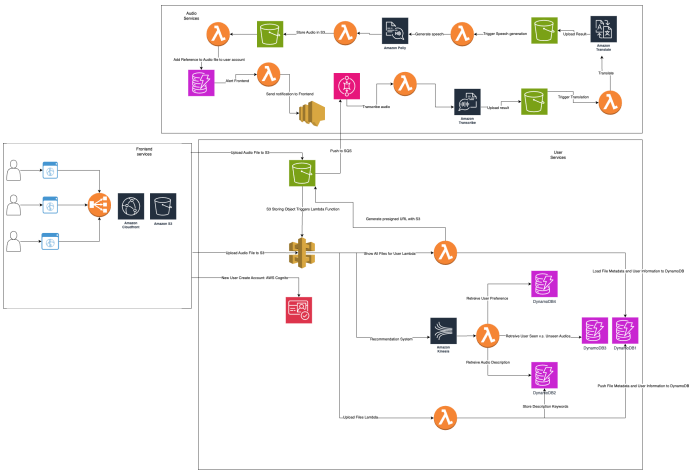

Fig. 7. Final Architecture

In Figure 7, we created pages to display all audio files before conversion and pages to show user's audio files after they have been converted. Additionally, we introduced a system that recommends audio files based on the labels of other audio files when users click 'likes'. To achieve this, we added three DynamoDB tables, connected them to the API Gateway, and thus enhanced user convenience.

VI. Conclusion

For our future works, scalability remains a key objective for our cloud architecture. We can further implement the elastic capabilities of cloud resources to ensure that our architecture can dynamically adapt to the fluctuating demands of our user base. This will involve implementing auto-scaling policies and fine-tuning our load balancing strategies to handle spikes in traffic gracefully, without sacrificing performance. Additionally, we can leverage container orchestration tools such as Kubernetes, connected with AWS EKS cluster, which can provide a more efficient way to deploy and manage applications at scale. This approach not only simplifies deployment patterns but also enhances our application's fault tolerance. By utilizing multiple pods across different nodes, we can ensure high availability and distribute traffic efficiently.

Performance optimization of our machine learning models is another crucial area for development. We can focus on enhancing model training efficiency and inference speed. This will likely involve transitioning to more powerful and specialized hardware such as GPUs or TPUs for training, and optimizing our models to run on edge devices for faster inference. We can also use Apache Kafka, since Kafka clusters can be expanded without downtime, which makes it highly scalable for streaming large amounts of data. The use of quantization and model pruning techniques can also be implemented to reduce model size and computational requirements without compromising accuracy.

In wrapping up, we have leveraged AWS microservices to create a real-time podcast translating, transcribing, and recommendation system, enhancing the overall efficiency and user experience. We've streamlined the data management with a single-model system and improved the system's functionality through careful design refinements. Our efforts have not only addressed current needs but also laid the groundwork for future enhancements. The modular approach has increased the system's reliability, and by integrating modern cloud services, we've prepared the architecture to handle growth and change.