Parrot: PDF Text to Speech Conversion Using AWS

Andrew Li and Miya Liu

CS 310 Scalable Software Architectures

Parrot: PDF Text to Speech Conversion Using AWS

Text-to-speech synthesis is commonly used across various industries, helping to create increasingly accessible software. Research on text-to-speech systems indicates two main components of a TTS software: a natural language processing (NLP) component and a speech synthesis component, which must work hand-in-hand to produce intelligible speech [1]. Indeed, literature indicates a long history of research into speech synthesis [2], and many websites across the internet offer free services to convert text-based input to audio-based output. Here, we use Amazon Web Services (AWS) to create a micro-scale text to audio conversion service, creating a mini-service that allows users to upload small PDF files and convert them into human-sounding audio files.

1 Software Stack

This software is split into a web-based frontend, serverless API, manually built gateway[1], and SQL database. Here, the frontend is hosted on DigitalOcean, which has built in support for load balancing. The API and database are hosted using AWS Lambda functions and an RDS database respectively, which have built in support for scalability. Finally, the custom-built API gateway is hosted on DigitalOcean. Further steps can be taken to increase the scalability of this software stack, though by choosing DigitalOcean and AWS, these steps are easier than manually spinning up servers, as DigitalOcean makes load balancing very easy; it just costs more money.

1.1 Frontend

The frontend is written using NextJS[2], a superset of React.js, which is widely used in the software engineering industry, in TypeScript. This choice was made because a JavaScript based framework is easy to use, and more importantly, is easy to connect to a backend. The frontend is then split into four main components: a file upload view, text content view, upload history view, and a play audio view, all of which are effectively sibling components.

1. The file upload view does as the name implies; the user can upload a PDF file (max size 200 KB)
2. The text content view takes the text content (not the raw bytes) of the PDF and displays it to the user, though it does not allow for editing. There is a submit button that converts the content into audio.
3. The upload history view stores data of all previously uploaded files, and has two buttons that activate the corresponding audio and text content, and a button to delete that file from the stored upload history.
4. The play audio view allows for the user to, after the audio is generated by clicking the aforementioned submit button, play the audio file generated from an AWS Lambda function.

The fact that these are sibling components creates complexity in passing props between them, especially because some of them are actually cousins. For this reason, on top of a NextJS/React based frontend, we used Redux to manage global state, which effectively wraps the entire app in a `useContext` React hook. Doing this simplifies passing state between sibling components, which makes it easy to, for example, store a user's upload history from one component, and easily access it from another. A persistence library (`redux-persist`) was used to persist the global state across refreshes, which does so by storing the global state in the browser's cookies[3]. Here, we define the structure of the global state as

```
fileHistory: IUploadedFile[];     list of uploaded files          (1.1)
apiEndpoint: string;              API gateway endpoint            (1.2)
mostRecentFile?: IUploadedFile;   active file (text and audio)    (1.3)
errorMsg?: string;                an error message to appear in an error box   (1.4)
```

where `IUploadedFile` is a struct (interface in TypeScript) with the fields

[1] For a reason we believe is related to CORS, API Gateway and Lambda both were not working natively, so we built our own solution.

[2] Version 12, due to DigitalOcean only having NodeJS 16, and v14 requiring NodeJS 18.

[3] Indeed, clearing cookies resets the global state.

1. `localKey: string | number` This acts as a hash for every uploaded file, without hashing the contents of the file. While we could use the job ID that is returned from the Lambda function, having a client-side generated key allows for a future feature where a user can manually input an AWS job ID, add it to their job history, and have it treated separately from duplicates of that job ID, which would be a critical feature if text editing was implemented.
2. `fileName: string` The name of the uploaded PDF file.
3. `fileContent: string` The **raw content** of that PDF file, converted to base 64.
4. `content?: string` The **text content** of that PDF file, returned from an AWS Lambda function.
5. `audioS3Key?: string` The S3 key for the `mp3` file generated by an AWS Lambda function.
6. `awsJobId?: string` The job ID of the uploaded file, which is the primary key in the database, which is described in that corresponding section.
7. `awsTextKey?: string` The key of the text file in S3. This is passed to the Lambda function that converts text to audio instead of a job ID, as it effectively caches the results of that query by passing it through the frontend, eliminating the need to query the database again.
8. `audioRaw?: string` The raw bytes, in base 64, of the audio file returned by the text-to-audio Lambda function.

The frontend is hosted on DigitalOcean, which has built in support for load balancing, though can be further enhanced by building the frontend into a Docker image stored on AWS ECS, and using a service like AWS App Runner which handles scaling.

1.2 Lambda Backend

The meat of the backend API is written as a series of AWS Lambda functions using Python 3.11. Specifically, there are four Lambda functions:

1.2.1 Extract Text

The extract text function takes `POST` requests, because it passes the base-64 encoding of a PDF file into the Lambda function. The Lambda function then creates a record in the SQL database and adds the extracted text as a text file to the S3 bucket, and returns this text to the client.

1.2.2 Convert Text to Audio

The convert text to audio function takes `POST` requests, because it updates the record in the database, creates an `mp3` file in the S3 bucket, and returns a string to indicate success or failure. The actual audio, though downloaded immediately after in this implementation of the frontend, could theoretically be downloaded separately which could be useful for users on mobile data or with other bandwidth limitations.

1.2.3 Download Audio

The download audio function takes `GET` requests, as it simply takes a job ID as a parameter, finds the corresponding audio file, and returns the raw bytes of that file encoded in base 64.

1.2.4 Reset

The reset function takes `DELETE` requests, as it removes records from S3. The route takes `jobid` as a URL parameter, and deletes the files corresponding to that job.

1.3 API Gateway

Using AWS API Gateway is the obvious choice for a project like this, but a reason related to CORS prevented it from working properly. That is, when running our app on a browser with CORS disabled, everything worked, but on a normal browser, everything failed. To solve this, we built our own API Gateway using Express.js, which forwards the request to the corresponding Lambda function (or, for the route with a path parameter, AWS API Gateway), as because the Express.js server is not running in a web browser, CORS is ignored. Implementing this was quite trivial, and this gateway was deployed to DigitalOcean.

1.4 Database

For the database, we used a MySQL database hosted using AWS RDS. We did not do any replication or sharding, as this is a very small scale project, though either could be added using technology such as Orchestrator[4] that automatically handles replication. Within our database, we have one table for jobs, as we did not implement authentication for this project. This table's schema contains a job ID, the original PDF file name, and S3 keys for each of the files that were created by one of the Lambda functions.

Job ID (PK)	**Status**	**Original PDF Name**	**Text File S3 Key**	**Audio File S3 Key**
...	...	...	...	...

1.5 Software Diagrams

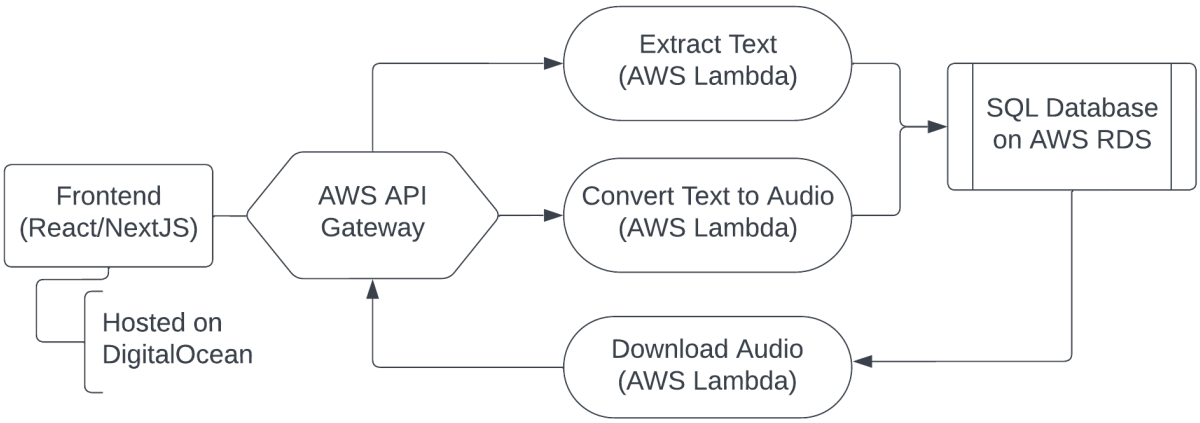


Figure 1: Software architecture diagram for Parrot, illustrating the relationship between different layers of the software stack.

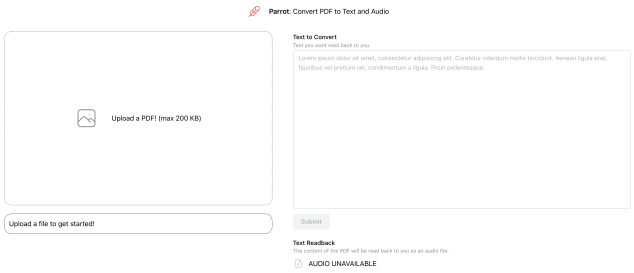
[4] https://github.com/openark/orchestrator

2 End Product

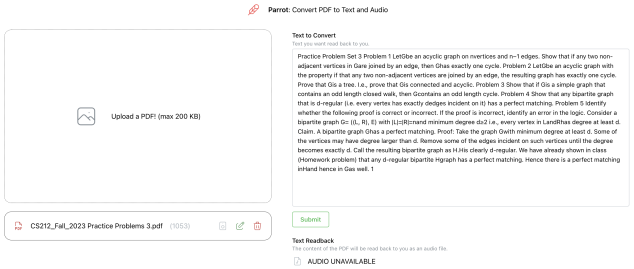
The end product, hosted on DigitalOcean, can be found at

https://squid-app-xzsji.ondigitalocean.app/

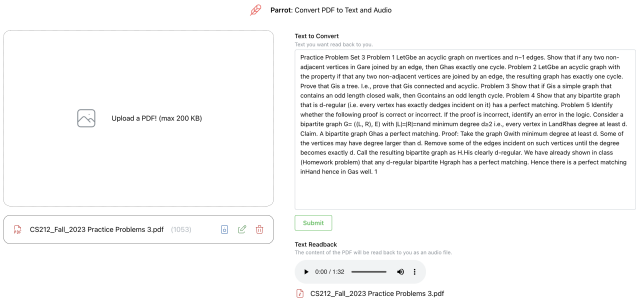
Initially, the website will appear like



After uploading a practice problem set from CS 212, it looks like



After clicking the submit button (and waiting), it looks like



which, upon playing the audio, reads out the text in that box.

References

[1] C. Ungurean and D. Burileanu, "An advanced NLP framework for high-quality text-to-speech synthesis," in *2011 6th Conference on Speech Technology and Human-Computer Dialogue (SpeD)*, May 2011, pp. 1–6.

[2] T. Dutoit, "NLP architectures for TTS synthesis," in *An Introduction to Text-to-Speech Synthesis*. Dordrecht: Springer Netherlands, 1997, pp. 57–70. [Online]. Available: https://doi.org/10.1007/978-94-011-5730-8_3