Cornelius Design Justifications

**Backend Framework**

For our app to function properly, we needed to have a backend framework for our server that could interact with our database and serve the requests of our frontend. In the end, we decided to use Django for our backend framework.

We decided to use Django because it comes with a built-in ORM and makes database management easier. If we were to use another backend framework like Express or Flask, it would mean that we would have to import specific packages or third party ORMs in order to interact with our database. Django streamlined this process. Although Django is usually used as a full-stack framework, it works just as well to create simple REST API endpoints that can serve a separate frontend application.

As mentioned above, we also considered using Express or Flask. The benefits of Express are the fact that it is a Node.js framework that would be intuitive to use along with a JavaScript frontend. Because React Native was our choice for our frontend, this would mean consistency across our stack and less code-switching for full-stack developers. However, we did not think that this reason outweighed the added hassle of added libraries and configuration for database management in comparison to Django. Similarly, Flask seemed like a very lightweight and intuitive way to spin up a server using Python with a pretty shallow learning curve. However, we would still need another ORM like SQLAlchemy to manipulate our database which did not seem necessary given that Django could do this for us automatically.

At this point, our team had already decided on using React Native for our frontend, which affected the perspective we had coming into this conversation. Because of this, the decision to use Django meant that our tech stack would be a little more nuanced because we needed to code in Python as well as JavaScript. However, our team is familiar enough in both languages that we did not think it would be too difficult to switch between the two as we developed the full-stack mobile app. This decision impacted our choice for our database management system, which is described below.

**Database Management System**

An important part of any application is database management, which is a topic that we needed to discuss after we had settled on Django as our backend framework. We wanted a robust and well-documented DBMS and in the end we decided on PostgreSQL.

We decided to use PostgreSQL because it has a higher capacity constraint than other SQL variants, and also because there is a lot of documentation surrounding it that would help in our development process. We also liked that it was a relational database system because we wanted to be able to have an organized schema for our datasets that could support more complex queries in the future. Non-relational database systems are considered more scalable and flexible but our team thought they might be too volatile to be used reliably in our production application.

We considered using other SQL variants like SQLite and non-relational database management systems like MongoDB as alternatives to PostgreSQL. For SQLite, we were concerned that the capacity constraint could be a limitation as the app is scaled up in the future. It is a very easy database system to set up because we can store the database in a file on our local machine and do not need to create an instance of the database separate from our application. However, it is less commonly used in production than databases like PostgreSQL and thus has less documentation. MongoDB is a popular non-relational database system that we also considered. Because of the way it stores data, it can often scale horizontally more easily than PostgreSQL and is more flexible because it does not have set schemas or tables in place. However, this can lead to data inconsistencies and make the process of more complex queries like joins more difficult to execute. Because we wanted to preserve the relations between tables in the form of primary and foreign keys, we decided to stick to a SQL database like PostgreSQL.

At this point, our team was already assuming that we would use Django in the backend, which affected how we addressed our conversation about databases. Django by default uses SQL and the ORM is meant to interact with a relational database. This meant that we were biased against non-relational databases like MongoDB and ultimately made the decision for PostgreSQL more natural. This decision impacted how we developed our APIs using Django but overall made the process easier for backend development.

**Audio Processing APIs**

This was perhaps the most important issue since it regarded the core functionality of our application. We needed to find an accurate and reliable audio processing API to transcribe our patient appointments because our application deals with sensitive information and the transcription could greatly affect our user’s health. We ultimately chose to use the Google Speech-to-Text API because the documentation was very comprehensive and frequently updated, the API was developed earlier than other alternatives that we looked at, and Google is a well-known and reliable software company.

The advantages of Google Speech-to-Text described above were important to us because we all have had limited experience working with audio processing APIs, so detailed documentation would not only help us have a smoother initial take off, but would also be crucial in the debugging process. React Native Voice was one of the main alternate APIs that we considered, and it was free as opposed to Google’s rates for its API services. Google charges $0.006 for every 15 seconds of audio data processed after the first hour each month, so React Native Voice would be much cheaper and sustainable in the long run. Eventually, we had to discard because documentation was limited to GitHub repos and Medium tutorials that were not as reliable as the Google distributed documentation. Furthermore, because the Google API was released in 2017, there has been enough time for others to have tried out the API and discuss potential bugs and errors on platforms like StackOverflow. Having this community is crucial for us since we are all working remotely and have limited assistance from sources offline. Our clients also preferred a more reliable API like Google’s despite the monetary incentive to use React Native Voice.

This decision affected how we implemented the API in the frontend and backend. Instead of being able to install an npm package in our project, we needed to create Google Cloud Platform accounts and configure the API keys for our project. This required more manual configuration work upfront, but we decided this was worth it for a more robust and well-documented audio processing API.

**Frontend Framework**

Our client requested a mobile application due to the fact that potential users would want to access their medical visits whenever and wherever and also be able to record as seamlessly as possible. Furthermore, our client wanted this application to be accessible to users with both Android and iOS devices. This meant that we needed to look for a frontend framework that could help us develop our UI easily while being able to deploy to these main operating systems.

Since we wanted to deploy to both Android and iOS, we mainly considered React Native because of its popularity, extensive documentation, and flexibility with both platforms. However, after attending our cohort meeting, another group mentioned Flutter, which became the main other consideration for us in terms of our frontend development. The advantages of Flutter are that it has more UI component options and has a Hot Reload feature that makes it faster and easier to develop the frontend. However, we ultimately decided to go with React Native because many of our team members have had experience with React and Flutter was released only a couple of years ago. This means that there is likely to be less documentation and consequently, a harder time dealing with bugs because of how recently Flutter was released. Having been released more than five years ago, React Native is a much more mature framework with a better community which means that there are plenty of tutorials and libraries to work with. Furthermore, Flutter is written in Dart, which meant that our team members would have to learn a new language, which just did not seem practical given our current timeline. We also considered using Swift because it is another well-documented mobile framework, but we quickly discounted it because it did not support Android development.

The frontend framework was the first decision that we made in our development process, which dictated the decisions we made regarding our backend. However, due to the flexibility of the React Native framework, we were still able to make the decisions about our audio processing API independently of our front end.