

Improving Scalability and UX of the QDAcity Transcription Service

MASTER THESIS

Shu-Man Cheng

Submitted on 20 June 2025



Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:

Dr. Andreas KAUFMANN, M.Sc.

Prof. Dr. Dirk Riehle, M.B.A.



Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Erlangen, 20 June 2025

License

This work is licensed under the Creative Commons Attribution 4.0 International license (CC BY 4.0), see <https://creativecommons.org/licenses/by/4.0/>

Erlangen, 20 June 2025

Abstract

Qualitative research usually consists of multiple activities to extract information from unstructured data. The transcription process is one of the steps in qualitative research and is crucial for the results. With a large data set, the transcription process can be time-consuming and tedious. To enhance the efficiency, Qualitative Data Analysis (QDA) software provides tools for the user to organize and process the data as well as to optimize the workflow.

QDAcity is such a cloud-based web application for QDA. It provides a transcription service to convert audio to text, a customized dashboard and collaborative research but it is nevertheless under constant development for improvement. For this thesis, a new transcription service was designed and implemented for QDAcity. The transcription editor was also enhanced with a new UI design, a speaker diarization feature, error handling and a guided tour to explain features. It was important, when adding those features, to ensure compatibility with the existing application.

These new features were broken down into several steps. First, the transcription service of the QDAcity Java backend was migrated to a new transcription service in Node.js to improve the scalability and performance. Second, a new feature was implemented: Speaker diarization. This was achieved by restructuring the data format from the cloud-based service and using a React transcription editor library to display the results on the frontend. Third, the UI elements were designed and adjusted to be more intuitive and modern to improve the user experience. After the implementation, the user can now use these new features in QDAcity.

Contents

1	Introduction	1
1.1	QDAcity	1
1.2	Problem Statement	2
1.3	Objective	2
1.4	Thesis Structure	3
2	Requirements	5
2.1	Functional Requirements	5
2.1.1	New Transcription Service Infrastructure	6
2.1.2	Authentication and Security	7
2.1.3	Speaker Diarization	7
2.1.4	Error Handling	8
2.1.5	UI/UX Improvements	8
2.2	Non Functional Requirements	8
3	Architecture	13
3.1	Current System	13
3.2	Tech Stack	14
3.3	Google Cloud Speech-to-Text and Google Cloud Storage	15
3.4	Get Audio File Meta Data	17
3.5	Transcription Editor Library	18
3.6	Upload Progress and Transcription Progress	20
3.7	Google STT Error Messages	21
3.8	Common Code Conventions in Transcription Service	22
3.9	New Transcription Service System Architecture	22
4	Design and Implementation	25
4.1	High Level Overview	25
4.2	Media Upload Process	26
4.3	Authentication and Security	26
4.4	Transcription Process and Speaker Diarization	27
4.4.1	Transcription Process	27
4.4.2	Speaker Diarization	28
4.4.3	Rename Speaker	30
4.5	Error Handling & Logging	31
4.6	UI UX Improvement	32
4.6.1	Transcription Editor Redesign	32

4.6.2	Transcription Editor Tour	34
4.7	Cloud Deployments and Release	36
5	Evaluation	37
5.1	Functional Requirements	37
5.1.1	New Transcription Service Infrastructure	37
5.1.2	Authentication and Security	38
5.1.3	Speaker Diarization	39
5.1.4	Error Handling	39
5.1.5	UI/UX Improvements	40
5.2	Non Functional Requirements	40
6	Future Improvement	43
6.1	Role Based Access Control	43
6.2	Unsupported Language for Speaker Diarization	43
6.3	Transcription Service on Different Browser	44
7	Conclusion	45
	References	47

List of Figures

2.1	FunctionMASTeR Template based on Rupp (2014)	5
2.2	Conditions of requirements based on Rupp (2014)	6
2.3	EnvironmentMASTeR template based on Rupp (2014)	8
2.4	PropertyMASTeR Template based on Rupp (2014)	9
2.5	Product Quality Model based on ‘Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model’ (2023)	9
3.1	Current System Component Diagram	14
3.2	Speaker Count Input Box	17
3.3	GCP STT Output 1	19
3.4	GCP STT Output 2	19
3.5	BBC kaldi Format 1	20
3.6	BBC kaldi Format 2	20
3.7	Transcription Service Component Diagram	23
4.1	High-Level Flow	25
4.2	Upload Process	26
4.3	Authenticate Process	27
4.4	Transcription Process	28
4.5	Transcription Request	29
4.6	Original Speaker Labels Display on Transcription Editor	30
4.7	New Speaker Labels Display on Transcription Editor	30
4.8	Rename Speaker Window	31
4.9	Rename All Instances of Speaker	31
4.10	Error Handling Process Diagram	32
4.11	Error Notification Window	32
4.12	Original Transcription Editor UI	33
4.13	New Transcription Editor UI	33
4.14	Transcription Editor Tour - Step 1	35
4.15	Transcription Editor Tour - Step 2	35
4.16	Transcription Editor Tour - Step 3	36
4.17	Cloud Deployment	36
6.1	Unsupported Language Speaker Diarization	44

List of Tables

3.1	Technology Stack Overview	15
3.2	Google STT Transcription Models	16
3.3	Class LongRunningRecognize Metadata	21
3.4	Google Cloud STT Error Messages	22
4.1	Comparison of keyboard shortcuts for media playback controls	34
5.1	New Transcription Service Infrastructure - Requirements Fulfillment Overview	38
5.2	Authentication and Security - Requirements Fulfillment Overview	39
5.3	Speaker Diarization - Requirements Fulfillment Overview	39
5.4	Error Handling - Requirements Fulfillment Overview	40
5.5	UI/UX Improvements - Requirements Fulfillment Overview	40
5.6	Non-Functional Requirements - Fulfillment Overview	42

List of Code Examples

3.1	Google STT Config	16
3.2	Speaker Diarization Config	16
3.3	Music-Metadata Response	18

Acronyms

API	Application Programming Interface
BBC	British Broadcasting Corporation
CES	Collaborative-Editing-Service
CLI	Command Line Interface
CI	Continuous Integration
ES	ECMAScript
FLAC	Free Lossless Audio Codec
GAE	Google App Engine
GCP	Google Cloud Platform
GCS	Google Cloud Storage
HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notation
JWT	JSON Web Token
MP3	MPEG-1 Audio Layer III
NPM	Node Package Manager
QDA	Qualitative Data Analysis
RBAC	Role-Based Access Control
STT	Speech-to-Text
TBC	To Be Confirmed
URL	Uniform Resource Locator
UI	User Interface
UX	User Experience
WAV	Waveform Audio File Format

1 Introduction

QDA is an integral part of many qualitative research methods that includes multiple activities to extract relevant information from non-numerical data, interpret the data and abstract the information from it (Lacey & Luff, 2007). Qualitative data is usually gathered using methods like interviews, questionnaires or surveys (Güler, 2015). Qualitative research consists of several steps, from data collection, data transcription to data analysis (Bailey, 2008). Transcription is a crucial step in qualitative research, as quality significantly impacts the analysis. Inaccuracies in the transcription could lead to incorrect results (Stuckey, 2014).

With large data sets, the transcription process is time-consuming and monotonous. It is not just about typing the exact words spoken; the importance is to capture how the words are conveyed. Tone and inflection play key roles in the correct transcription process (Stuckey, 2014). Using QDA software, it becomes easier and faster for researchers, as they can skip the tedious task of transcribing (Evers, 2010).

1.1 QDAcity

QDAcity¹ is a cloud-based web application for QDA, developed by the Professorship for Open-Source Software in Friedrich-Alexander University Erlangen-Nürnberg. Running on Google Cloud Platform (GCP), QDAcity requires only a web browser to enable the researchers to perform QDA. It has several target groups such as students and research groups with each having different use cases such as marketing or teaching. One of the main features of QDAcity is the interview transcription automation. Using Google Cloud Speech-to-Text (STT) technology to transcribe the interview files from audio to text, QDAcity aims to save the users' time while transcribing. After the automated transcription, the user can then review and modify the results as well as export them for further analysis. In addition, it also allows multiple collaborators to work on the same project at the same time. Collaborative research is one of the strengths of QDAcity. Other features in QDAcity like coding, customized dashboard and recommendation system exist as well. However, they are not in scope of this thesis and will not be explained further. The main focus of the thesis is the transcription feature.

In terms of technology, QDAcity is built upon Google App Engine (GAE) environment using the Java Spring Boot framework in the backend. The frontend is built using JavaScript and TypeScript based on the React framework. It is deployed on GCP. Google Application Programming Interface (API)s are used for various features such as Google

¹QDAcity <https://qdacity.com/>

Cloud STT for transcription and Google Cloud Storage (GCS) for data storage.

1.2 Problem Statement

There are some problems that needed to be addressed in the transcription feature of QDAcity, including the speaker diarization, transcription service migration, error handling and User Interface (UI) and User Experience (UX) improvement.

First, although QDAcity already had the transcription feature in its transcription editor, the speaker diarization feature did not work correctly in the existing application because the transcription editor library that QDAcity used did not support the Google STT data format. Using Google STT, audio files were correctly transcribed, but the speakers were not distinguished. Instead, each sentence was assigned to a new speaker, which made it difficult to analyze which sentence was spoken by which person. Prior to this thesis, the unknown speakers were shown as To Be Confirmed (TBC) with an increment in the number for each sentence. This issue is addressed in this thesis to provide better transcription results.

Second, when it comes to the processing speed and scalability, QDAcity could profit from improvement. Compared to Java, Node.js is lighter, faster and more scalable in a serverless environment such as Google Cloud Run (Resende, 2015). Some studies show the migration from Java to Node has significant improvement in terms of performance. Node using Express.js framework shows slightly better CPU efficiency and lower memory consumption compared to Java using the Spring Boot framework in simple operations and real-time data processing (Shyam Mohan & Goswami, 2025). This is thus something that could be optimized in the application. However, the migration of the transcription service from Java to Node.js introduces new challenges in the system integration, particularly in terms of security communication and authentication, which have to be addressed to ensure a full and reliable implementation.

Apart from that, the UI and UX of the QDAcity transcription editor also required improvement. The UI of the transcription editor was not well integrated with the QDAcity design system. The export button on the transcription editor was not obvious with only a small exit door icon, making it difficult for the user to navigate. Furthermore, error handling and communication through the UI had room for improvement prior to this thesis. Only a limited number of audio formats were supported. When the user uploaded an unsupported type of file, the transcription did not work. This presented significant limitations to the ease-of-use. Without the feedback, the user did not know what the problem was. This was improved as part of the thesis.

1.3 Objective

Derived from the problem statements, the objective of this thesis is to improve the scalability and UX of the QDAcity Transcription Service with five derived work packages:

- **Migrate the Transcription Service from the Java backend to Node.js** - Improving the scalability and performance
- **Ensure authentication and security** - Ensuring secure access and data protec-

tion by implementing JSON Web Token (JWT)-based authentication and token validation.

- **Implement speaker diarization** - Identifying and labeling different speakers in the transcription by converting Google STT format to customized format for the transcription editor library
- **Display error handling message** - Providing error message for user feedback
- **UI and UX Improvement** - Enhancing user interface and experience design

1.4 Thesis Structure

Following the introduction in Chapter 1, Chapter 2 formulates the requirements for the thesis. The chapter is divided into two sections, namely (2.1) Functional Requirements and (2.2) Non-Functional Requirements. Chapter 3 focuses on the architecture of the new transcription service, which further discusses the system architecture, GCP and STT. Building upon this, Chapter 4 begins with a high level overview of the implementation, followed by detailed descriptions of each process. Chapter 5 focuses on the evaluation of the implementation referring to the requirements of Chapter 2. Based on the evaluation, Chapter 6 discusses the further needed improvements to QDAcity. Finally, Chapter 7 summarizes and concludes this thesis.

2 Requirements

This chapter states the requirements of the new transcription service infrastructure, authentication and security, speaker diarization, error handling and UI UX improvement. The requirements are categorized as Functional Requirements (FR) and Non-Functional Requirements (NFR). Based on ‘IEEE Standard Glossary of Software Engineering Terminology’ (1983), functional requirement specifies a function that a system or system component must be able to perform. Non-functional requirements are constraints on the services or functions offered by the system and often apply to the overall system, rather than individual features or services (Alashqar et al., 2015). In this thesis, both types of requirements are specified using the requirements templates by Rupp (2014) to avoid linguistic ambiguity. The templates will be further explained in chapter 2.1 and 2.2.

2.1 Functional Requirements

In this section, functional requirements are discussed and listed, using the Functional-MASTeR template by Rupp (2014) which is first explained.

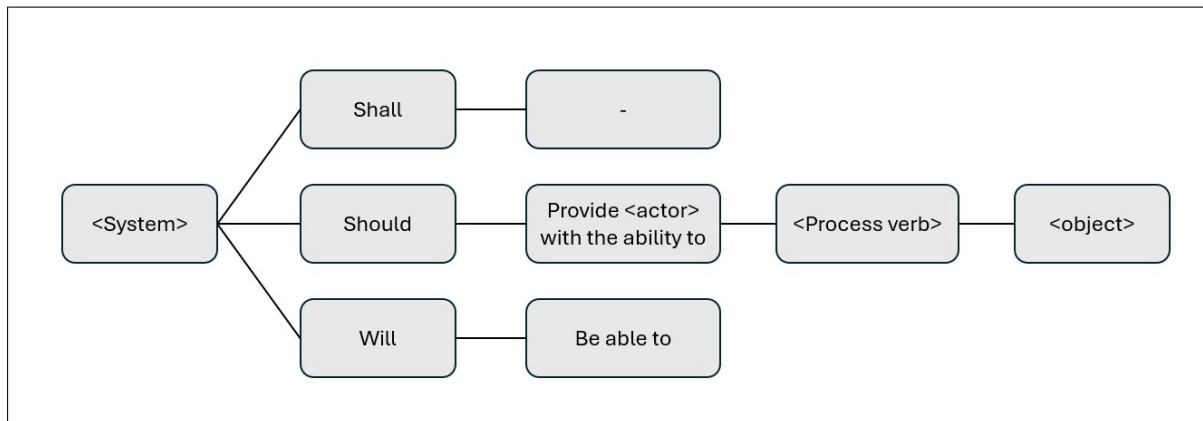


Figure 2.1: FunctionMASTeR Template based on Rupp (2014)

Figure 2.1 shows the FunctionalMASTeR template by Rupp (2014). The definition of the verbs (SHALL/SHOULD/WILL) have different degrees of importance for a specific requirement. The verb SHALL indicates the requirement that needs to be fulfilled. The verb SHOULD represents strongly recommended implementations. However, they are not mandatory. The verb WILL describes a requirement that will be implemented in the future, not during the current implementation. In this thesis, the “system” refers to the

current system of QDAcity. The actor (user) refers to the system admin, developers and the user who uses QDAcity.

The conditions are classified into three groups as figure 2.2 shows. For logical expressions, the keyword IF is used. The keyword AS SOON AS is triggered when the condition is related to an event. The keyword AS LONG AS is used when the situation is time related. Figure 2.2 shows that flow of operators.

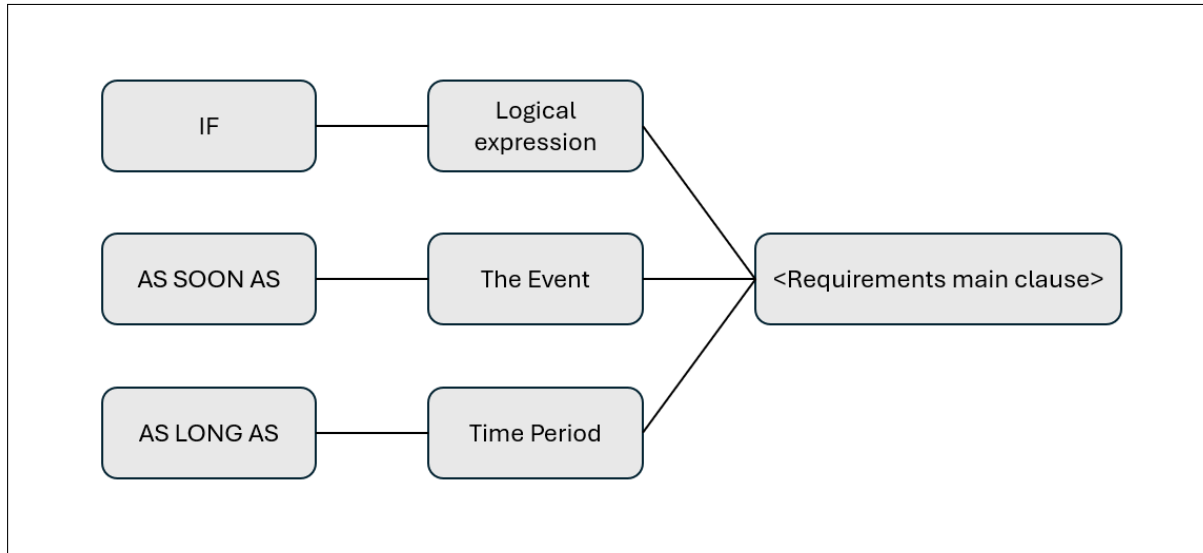


Figure 2.2: Conditions of requirements based on Rupp (2014)

Following the problem statement listed in the introduction, the five main topics with functional requirements are listed below.

2.1.1 New Transcription Service Infrastructure

FR-1.1: The Transcription Service shall be implemented using Node.js.

FR-1.2: The Transcription Service shall use TypeScript.

FR-1.3: The Transcription Service shall use Express.js.

FR-1.4: The Transcription Service shall initiate transcription tasks upon receiving HyperText Transfer Protocol (HTTP) requests for transcription jobs.

FR-1.5: The Transcription Service shall process transcription jobs through the Google Cloud STT API.

FR-1.6: The Transcription Service shall support the following audio formats: MPEG-1 Audio Layer III (MP3), Free Lossless Audio Codec (FLAC), and Waveform Audio File Format (WAV).

FR-1.7: The Transcription Service shall check file types and sample rate before making Google Speech-to-Text API requests.

FR-1.8: If the file is not supported due to its file type, sampling rate or it's a multichannel audio, the UI should pop up a notification window indicating the error message.

FR-1.9: The Transcription Service shall check the length of the audio file for the quota and billing control.

FR-1.10: The transcription results shall be returned to the client in JavaScript Object Notation (JSON) format.

FR-1.11: The Transcription Service shall be built, tested, and deployed using the QDA-city GitLab Continuous Integration (CI) pipeline.

FR-1.12: The Transcription Service shall run on Google Cloud Run.

2.1.2 Authentication and Security

FR-2.1: The system shall implement JWT-based authentication between the Java backend, client, and transcription service.

FR-2.2: The Java backend shall generate signed tokens or Uniform Resource Locator (URL) for each transcription request, with an expiration time.

FR-2.3: The Java backend shall provide an API endpoint that returns the public key for token validation.

FR-2.4: The Transcription Service shall expose an API that accepts requests for transcription.

FR-2.5: The Transcription Service shall cache the public key in memory for validation and update it when necessary.

FR-2.6: If the public key is missing in memory, the Transcription Service shall call the Java backend to retrieve it and validate tokens.

FR-2.7: The Transcription Service shall handle invalid or expired tokens by returning appropriate error messages.

FR-2.8: Only project members with “organizer” or “owner” roles shall be able to initiate or manage transcription.

2.1.3 Speaker Diarization

FR-3.1: The system shall integrate Google Cloud Speech-to-Text diarization capabilities to distinguish and label different speakers in the audio.

FR-3.2: Diarization data shall be accurately represented in the transcription output.

FR-3.3: The UI shall include an input box for users to specify the number of speakers in the audio file.

FR-3.4: The input box for speaker count shall allow only positive numbers.

FR-3.5: When the user renames a speaker on the UI, all instances of that speaker in the transcription shall be renamed at the same time, not just in one instance.

FR-3.6: The rename speaker prompt shall have a checkbox for user to choose if they want to rename all instances of speaker or not.

2.1.4 Error Handling

FR-4.1: The system shall provide error messages on the UI for issues like invalid file format, file type or sampling rate.

FR-4.2: The Transcription Service shall relay Google Cloud Speech-to-Text error messages to the frontend with simplified descriptions.

FR-4.3: Error messages shall be displayed as popup notifications for the users.

FR-4.4: All error messages shall be available in both German and English, based on user preferences.

FR-4.5: The system shall log all errors, including API failures and authentication errors, for debugging and monitoring.

2.1.5 UI/UX Improvements

FR-5.1: The transcription-related interfaces should be redesigned to improve usability and user experience.

FR-5.2: The application shall conform to the QDAcity design system.

FR-5.3: The UI should provide a transcription editor tour for the user who visits the transcription editor for the first time

2.2 Non Functional Requirements

In this section, non-functional requirements are presented. The sentence structure of non functional requirements are based on the EnvironmentMASTeR and the PropertyMASTeR templates by Rupp (2014) as shown in Figure 2.3 and Figure 2.4.

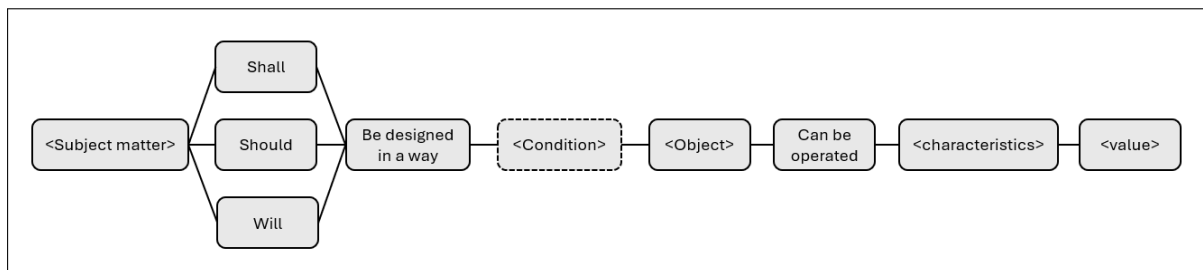


Figure 2.3: EnvironmentMASTeR template based on Rupp (2014)

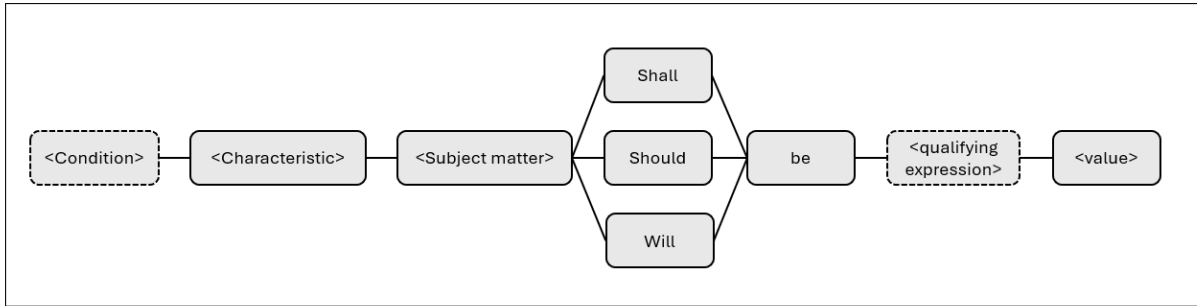


Figure 2.4: PropertyMASTeR Template based on Rupp (2014)

To have comprehensive software product quality properties, the non-functional requirements are designed to follow the Systems and software Quality Requirements and Evaluation according to ‘Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model’ (2023), which includes eight main characteristics and each of them has sub-characteristics as shown in Figure 2.5.

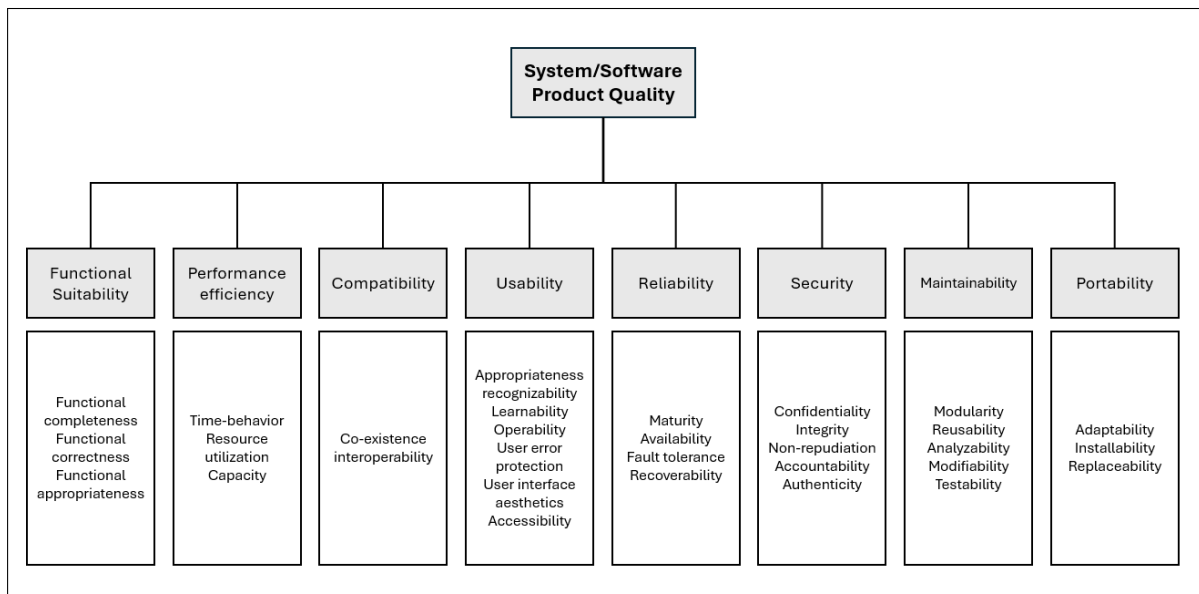


Figure 2.5: Product Quality Model based on ‘Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model’ (2023)

1. Functional Suitability

NRF-1.1: The Transcription Service shall integrate with the existing QDAcity application without breaking any part of the current system.

2. Performance Efficiency

NFR-2.1: The Transcription Service shall not exceed 500MB of memory usage during the transcription processing under normal load conditions (e.g., handling a 1-hour audio file)

NFR-2.2: Transcription response time shall not exceed 30 seconds for audio files up to 1 minute in duration.

3. Compatibility

NFR-3.1: The system shall be integrated into the existing QDAcity technology stack.

NFR-3.2: The Transcription Service shall be compatible with Node.js and Express.js

NFR-3.3: The frontend components shall be compatible with all Chrome, Firefox and Safari versions since 2022

4. Usability

NFR-4.1: The interface shall provide intuitive and consistent navigation, tooltips for all interactive UI elements, and real-time feedback to assist both technical and non-technical users in interacting with the transcription feature.

NFR-4.2: The interface shall be consistent with the rest of the QDAcity design.

NFR-4.3: The UI should provide a loading progress bar of upload and transcription progress for user feedback

NFR-4.4: The transcription results shall be displayed with speaker labels and timestamps to improve readability.

NFR-4.5: The media player in transcription editor shall provide keyboard hotkeys for the user.

5. Reliability

NFR-5.1: The system should not resume the upload if upload fails for more than 3 times in a row due to an unstable internet connection.

6. Security

NFR-6.1: Each endpoint shall check the authorization of a request

NFR-6.2: Requests must be sent via HTTP

7. Maintainability

NFR-7.1: The codebase shall follow standard naming conventions and modular design principles.

NFR-7.2: All API endpoints shall be documented using OpenAPI (Swagger), including request parameters, response formats, and error handling.

NFR-7.3: Code comments shall be provided for the functions to clarify their purpose.

NFR-7.4: The system's architecture and interactions between services shall be documented in the QDAcity Wiki.

NFR-7.5: Unit tests shall cover at least 80% of the transcription service logic, including handling different audio file formats.

NFR-7.6: Integration tests shall verify the authentication flow, role-based access, and STT API interaction.

NFR-7.7: Acceptance tests shall cover at least 80% of the new transcription service.

8. Portability

NFR-8.1: The system shall support deployment in a containerized environment (e.g. Docker) to allow flexibility on different platforms, not limited to Google Cloud Run

3 Architecture

This chapter presents the architecture of the new transcription service. The current system is discussed first, followed by the tech stack. Google Cloud STT, GCS, the music metadata library and the transcription editor library will be discussed in more detail. Lastly, new system architecture is described in this chapter as well.

3.1 Current System

The QDAcity is a cloud application built on GAE using Java for the backend logic, and JavaScript with the React Framework for the frontend. It uses Google Cloud API for different purposes, namely Google Datastore as a database, GCS to store files and Google Cloud STT to perform transcription job. GCS and STT will be discussed in further detail in chapter 3.3.

The current system of audio transcribing is illustrated in the component diagram in Figure 3.1.

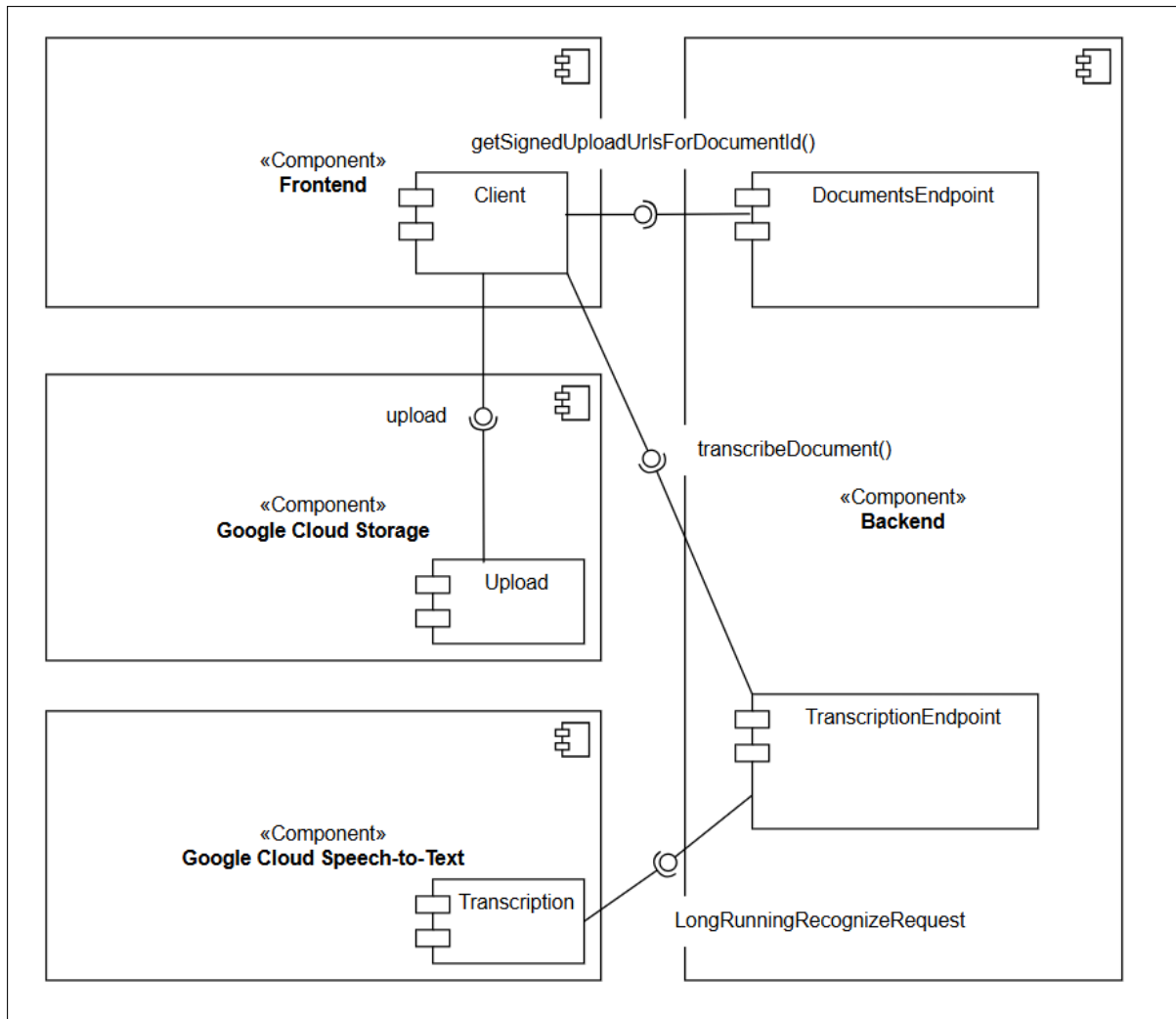


Figure 3.1: Current System Component Diagram

The `DocumentsEndpoint` in the Java backend generates the signed upload URL for frontend client to upload the audio file to GCS. Once the file is uploaded, the frontend calls `transcribeDocument` function in the `TranscriptionEndpoint` for the transcription job. The request is then sent to Google Cloud STT using `LongRuningRecognizeRequest`.

3.2 Tech Stack

The new Transcription Service shall be integrated into QDAcity current system. The choices of technology are thus constrained to the existing tech stack. The tech stack of this project is shown in Table 3.1.

Component	Technology
Frontend	
Programming Language	JavaScript, TypeScript
Framework	React
Backend	
Programming Language	Java, TypeScript
Cloud Infrastructure	Google App Engine
Persistence & Database	Google Cloud Storage, Google Datastore, Redis
API	Spring API

Table 3.1: Technology Stack Overview

Apart from the technology constraints, the Transcription Service also needs to follow the GitLab CI pipeline to streamline the building, testing and deployment.

3.3 Google Cloud Speech-to-Text and Google Cloud Storage

As described in the previous sub chapter, GCS and Google Cloud STT are used in the current system.

GCS¹ provides a storage management service on Google Cloud that can store many types of files (e.g. images, audio) which can be retrieved anytime. It is global, scalable and secure. The files are stored in buckets which are associated with the projects which are groups under an organization. Only the members of a team have permission to access the projects and files. An API call is needed to access the objects from GCS.

Google Cloud STT² uses Google AI to turn audio into text transcriptions. It supports 125 languages (e.g. English, German). The default language is set to English or German based on the user's chosen language in the system, but the user can change it with the language selector on the UI. There are also different transcription models as Table 3.2 lists. In this project, the telephony model is chosen because of its accuracy and reliability compared to other models.

¹Google Cloud Storage <https://cloud.google.com/storage>

²Google Cloud Speech-to-Text <https://cloud.google.com/speech-to-text>

Table 3.2: Google STT Transcription Models

Model Name	Description
latest_long	For long speech and conversations.
latest_short	For short utterances that are a few seconds in length.
telephony	Improved version of the <code>phone_call</code> model.
telephony_short	Dedicated version of the modern <code>telephony</code> model for short audio that originated from a phone call.
medical_dictation	For the notes dictated by a medical professional.
medical_conversation	For the conversation between a medical professional and a patient.
command_and_search	For short or single-word utterances.
default	For audio that does not fit the other audio models.
phone_call	For audio that originated from a phone call.
video	For audio from video clips that have multiple speakers.

Source: Google Cloud Speech-to-Text Transcription models Google Cloud, 2025c

Listing 3.1 shows the configuration of Google STT requests. It supports speaker diarization which can be integrated with the transcription request by adding a diarization configuration and specifying the minimum and maximum numbers of speakers in the audio file as Listing 3.2 shows. The user can fill in the numbers of the speakers in the audio file in the speaker count input box on the UI as Figure 3.2 shows.

```

1 const config = {
2   encoding: encoding,
3   sampleRateHertz: SampleRateHerzt,
4   languageCode: languageCode,
5   model: model
6 };

```

Listing 3.1: Google STT Config

```

1 {
2   "config": {
3     "features": {
4       "diarizationConfig": {
5         "minSpeakerCount": 2,
6         "maxSpeakerCount": 2
7       }
8     }
9   }
10 }

```

Listing 3.2: Speaker Diarization Config

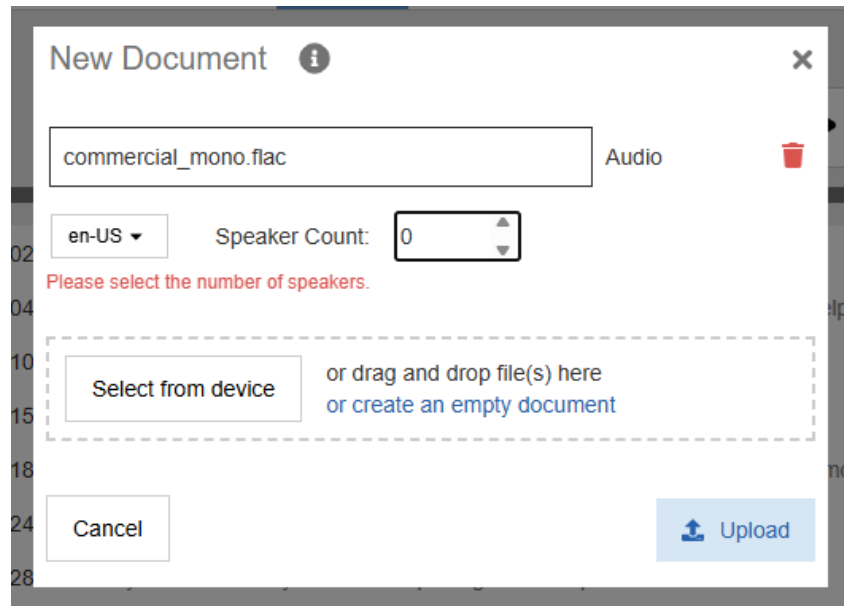


Figure 3.2: Speaker Count Input Box

The current system allows the users to upload files to GCS with a signed URL. The files will be stored in GCS. One of the reasons to use Google Cloud Speech-to-Text API is that through Google Cloud STT, it can transcribe the audio files that are stored in GCS directly without downloading the files or any further steps to transfer the file. Thus, in this project, the new transcription service follows the existing nature of using Google Cloud Speech-to-Text API and Google Cloud Storage for a more efficient implementation.

3.4 Get Audio File Meta Data

Before the transcription can be done, some meta data of the audio file is needed. There are two reasons why getting the audio file meta data before the transcription job is necessary. Firstly, some file information that needed to be sent in the request to Google Cloud STT. Secondly, the length of the audio file is needed to control the quota and billing system.

First, to send the request to Google Cloud STT API, certain file information needs to be provided (e.g. sample rate, encoding). As a result, using a library to get the file's metadata before uploading the file is a good method. For FLAC and WAV file, the encoding and sample rates are not required. Google STT determines the encoding and sample rate automatically for them.³ In this case, only getting the sample rate for MP3 files is essential.

Second, to integrate the quota and billing system for controlling the allowed transcription hours per user, the length of the uploaded audio file is required. The billed time that is provided in Google Cloud STT's response is not exactly the length of the audio file but diverges from it by about 2 seconds. Besides, the length of the audio should be determined before transcribing the file to avoid unnecessary billing. Consequently, another way of

³Google Cloud Speech-to-Text Documentation: Encoding. <https://cloud.google.com/speech-to-text/docs/encoding>

getting the length of audio file is needed, and using a library to get the file's metadata would be beneficial.

Taking the above two reasons into consideration, using a library to get the audio meta data is a good choice. The music-metadata library is chosen because of its simplicity and compatibility.⁴ It supports various audio formats like MP3, FLAC and WAV and it can parse the meta data of the audio in a Node.js readable stream. In other words, the meta data of the audio file that is stored in GCS can be parsed directly. Listing 3.3 shows the response of the music-metadata library. The sample rate and duration are the needed information.

```
1 Audio Format: {
2   tagTypes: [],
3   dataformat: 'WAVE/PCM',
4   bitsPerSample: 16,
5   sampleRate: 8000,
6   numberOfChannels: 1,
7   bitrate: 16000,
8   lossless: true,
9   numberOfSamples: 362400,
10  duration: 45.3
11 }
```

Listing 3.3: Music-Metadata Response

One noteworthy thing is that the music-metadata library is a pure ECMAScript (ES) module, which does not declare export functions for common JavaScript modules. To use this library, the Transcription Service is configured accordingly to be an ES module.

3.5 Transcription Editor Library

To display the transcription result on the transcription editor, the existing project uses the `bbc/react-transcript-editor` library⁵, which is a react component that displays transcription of videos and audios on the UI. With the `bbc/react-transcript-editor` library, the transcription editor can display the media player for audio files and the transcription results with speaker diarization. However, the speaker diarization of GCP STT is not working correctly with this library. Thus, the conversion of GCP STT output (as Figure 3.3 and Figure 3.4 show) into British Broadcasting Corporation (BBC) Kaldi data format (as Figure 3.5 and Figure 3.6 show) is necessary. BBC Kaldi is the default STT type of this library, which makes speaker diarization work correctly.

⁴music-metadata - npm package. <https://www.npmjs.com/package/music-metadata>

⁵bbc/react-transcript-editor library. <https://github.com/bbc/react-transcript-editor>

```
{
  "results": [
    {
      "alternatives": [
        {
          "transcript": "Anderson I've got the biggest serving Creator to buy everything in the store you know it no joke we are",
          "confidence": 0.6165882349014282,
          "words": [
            {
              "startTime": {
                "nanos": 400000000
              },
              "endTime": {
                "seconds": "1",
                "nanos": 100000000
              },
              "word": "Anderson"
            },
            { ... },
            { ... }
          ]
        }
      ],
      "resultEndTime": {
        "seconds": "32",
        "nanos": 230000000
      },
      "languageCode": "en-us"
    },
  ],
}
```

Figure 3.3: GCP STT Output 1

```
{
  "alternatives": [
    {
      "words": [
        {
          "startTime": {
            "nanos": 400000000
          },
          "endTime": {
            "seconds": "1",
            "nanos": 100000000
          },
          "word": "Anderson",
          "speakerTag": 1,
          "speakerLabel": "1"
        },
        { ... },
        { ... }
      ]
    }
  ],
  "totalBilledTime": {
    "seconds": "52"
  },
  "requestId": "7018945367944794353"
}
```

Figure 3.4: GCP STT Output 2

```
"action": "audio-transcribe",
"retval": {
  "status": true,
  "woid": "octo:2692ea33-d595-41d8-bfd5-aa7f2d2f89ee",
  "punct": "Good morning everyone. Welcome to our team meeting. Today we'll be discussing the quarterly results.",
  "words": [
    {
      "start": 0.52,
      "confidence": 0.98,
      "end": 0.78,
      "word": "good",
      "punct": "Good",
      "index": 0,
      "speakerTag": 1
    },
    { ... },
    { ... }
  ],
  "segmentation": {
    "metadata": {
      "version": "0.0.10"
    },
    "@type": "AudioFile",
    "speakers": [
      {
        "@id": "Speaker 1",
        "gender": "F"
      },
      { ... }
    ]
  }
},
```

Figure 3.5: BBC kaldi Format 1

```
],
"segments": [
  {
    "@type": "Segment",
    "start": 0,
    "duration": 8.45,
    "bandwidth": "S",
    "speaker": {
      "@id": "Speaker 1",
      "gender": "F"
    }
  },
  { ... }
]
},
"elapsed": 0.090955018997192,
"servertime": "20230501111317.9727"
```

Figure 3.6: BBC kaldi Format 2

3.6 Upload Progress and Transcription Progress

To provide the user with feedback on the UI before presenting the transcription results, uploading and transcribing progress bars are designed. For file uploads, a progress calculation component is used to compute the upload percentage, which is then reflected in the UI. Once the file is uploaded completely, the upload progress bar reaches

100% and the transcription progress bar begins to update for the user. To get the transcription progress update from Google Cloud API, the asynchronous speech recognition `LongRunningRecognizeResponse` is used instead of `Recognize` which performs synchronous transcription. The Long Running Operation object includes a `metadata.progressPercent` field which presents the progress percentage. Table 3.3 shows the attributes of long running operation.

Table 3.3: Class `LongRunningRecognize Metadata`

Attribute Name	Type	Description
<code>progress_percent</code>	<code>int</code>	Percentage of audio processed thus far. Show 100 when the transcription results are available.
<code>start_time</code>	<code>Timestamp</code>	Time when the request was received.
<code>last_update_time</code>	<code>Timestamp</code>	Time of the most recent processing update.
<code>uri</code>	<code>str</code>	The URI of the audio file being transcribed.
<code>output_config</code>	<code>TranscriptOutputConfig</code>	A copy of the <code>TranscriptOutputConfig</code> if it was set in the request.

Source: Google Cloud Class `LongRunningRecognize Metadata` Google Cloud, 2025a

3.7 Google STT Error Messages

Some types of files are not supported by Google Cloud STT API. If those unsupported types are uploaded, the system will get an error. In this project, a pop-up notification window for error message displays is designed to inform the user, so that the user can act accordingly. The error message received from Google Cloud STT is described in Table 3.4. As Table 3.4 shows, the messages from Google Cloud STT are long and too complicated for non-technical users to understand. As a result, an easier to understand and shorter message to display on the UI is necessary. In this project, a few common error messages are chosen to display using simple language in both English and German.

Table 3.4: Google Cloud STT Error Messages

Problem	Messages received
The Application Default Credentials are not available	{“The Application Default Credentials are not available. They are available if running in Google Compute Engine. Otherwise, the environment variable GOOGLE_APPLICATION_CREDENTIALS must be defined pointing to a file defining the credentials.”}
File does not exist	{“ERROR: File /path/to/key.json does not exist!”}
API key missing	{“error”: {“code”: 403, “message”: “The request is missing a valid API key.”, “status”: “PERMISSION_DENIED”}}
Forbidden: 403 POST API has not been used or is disabled	{“Forbidden: 403 POST Speech-to-Text API has not been used in project # before or it is disabled. Enable it by visiting [url] then retry. If you enabled this API recently, wait a few minutes for the action to propagate to our systems and retry.”}
Must use single channel (mono) audio	{“Must use single channel (mono) audio, but WAV header indicates 2 channels.”}
Must use 16 bit samples for LINEAR_PCM	{“INVALID_ARGUMENT: Must use 16 bit samples for LINEAR_PCM, but the WAV header indicates 8 bits per sample”}
Sync input too long	{“Sync input too long. For audio longer than 1 min use LongRunningRecognize with a ‘uri’ parameter.”}
Invalid recognition ‘config’: bad encoding	{“Invalid recognition ‘config’: bad encoding.”}

Source: Google Cloud Speech-to-Text Error Messages Google Cloud, 2025b

3.8 Common Code Conventions in Transcription Service

To conform to the standards of the existing code base, ESLint and Prettier are used in the new Transcription Service. An eslint configuration is added to extend the `.eslintrc` file in the root of the project. The use of `console.log` is restricted and should be replaced with `debug()`, `info()`, `warn()` or `error()` based on the intention of the log. Apart from that, the indentation should be in tabs instead of spaces. In this way, the transcription service follows the same code conventions as the rest of the code base.

3.9 New Transcription Service System Architecture

The new system architecture is illustrated in Figure 3.7. The components include the QDAcity frontend, the Java backend, the Transcription Service and two services of Google Cloud. Implementation details will be further discussed in chapter 4.

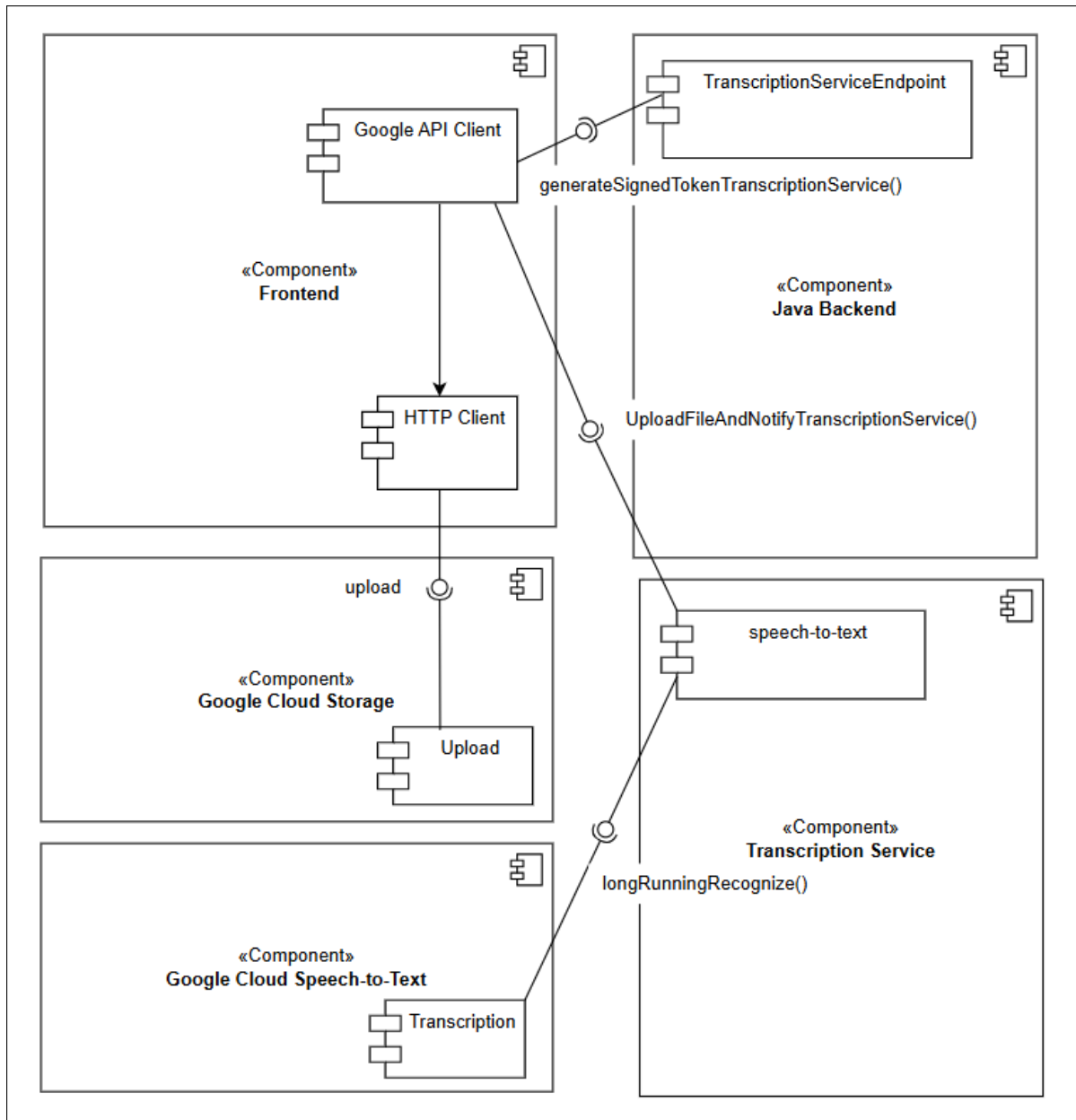


Figure 3.7: Transcription Service Component Diagram

In Figure 3.7, a component diagram illustrates five components in the new transcription service infrastructure. The **Transcription Service** endpoint in the Java backend provides **generateSignedTokenTranscriptionService()** to create a signed URL and the **Transcription Service** API URL and token are forwarded to the frontend client. The frontend client sends the request to the **speech-to-text** endpoint in the **Transcription Service**. Once the request is validated, the **speech-to-text** endpoint requests the transcription job by calling **longRunRecognize()** to the Google Cloud STT API.

4 Design and Implementation

This chapter discusses the implementation based on the requirements discussed in the previous chapters. The implementation follows the architecture described in chapter 3 and fulfills the requirements listed in chapter 2. First, the whole implementation will be presented as a big picture to see the overall flow. Second, the details of implementation for uploading, authentication, transcription, error handling and UI improvements will be discussed in the respective sub-chapters.

4.1 High Level Overview

The main flow of the transcription service is illustrated in the sequence diagram in Figure 4.1. The user uploads the file on the UI to trigger the process. The file is then stored in GCS using the signed URL generated by the Java backend. Google STT starts the transcription and speaker diarization job after receiving the request from the Transcription Service. The result is then displayed on the frontend for the user.

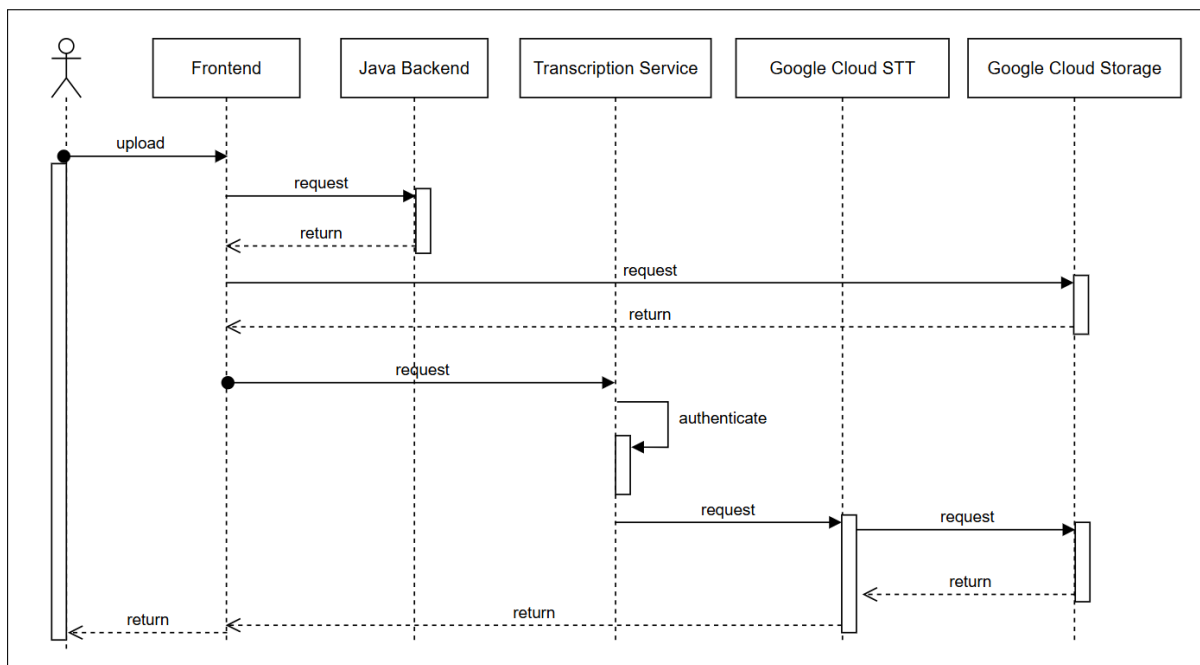


Figure 4.1: High-Level Flow

4.2 Media Upload Process

Figure 4.2 shows the upload process, presented as a sequence diagram. To ensure the security of file access to GCS, the signed URL is implemented. Accessing and uploading files to GCS is only possible with the authorization provided by the signed URL. The frontend is in charge of the file uploading, while the backend generates a signed URL and provides the file access.

When the user uploads an audio file, it triggers the `UploadFileAndNotifyTranscriptionService()` function in `upload.js` and passes the document related information as step 1 shows in the diagram. The signed URL to access GCS is generated through the Java backend and is then sent back to `upload.js` as step 2 to step 4 depicts. Subsequently, the signed URL is used to request the signed resumable upload URL by calling `getResumableUploadUrl()` and then uploading a file to GCS.

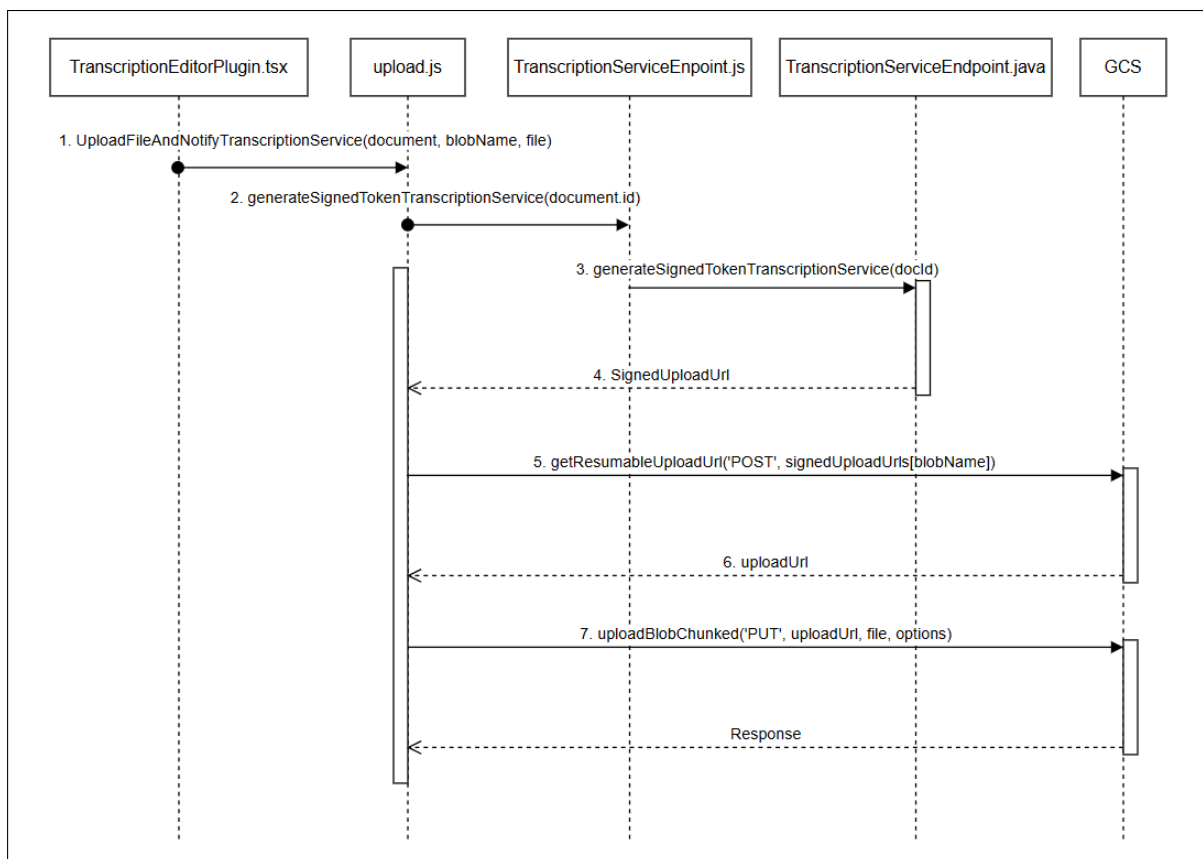


Figure 4.2: Upload Process

4.3 Authentication and Security

For authentication, JWT-based validation is used in the Java backend. It generates the token using the private key. The header of the request from the frontend should include the Bearer token for validation. In development, the Transcription Service gets the public key from the endpoint and validates the token locally. In production, the Transcription Service retrieves the public key from the Google Datastore and validates the token locally.

Once the user is validated, the transcription job is triggered. Only members of the project are allowed to access the project to ensure the security of the files.

Similarly to the previously discussed media upload process, the `UploadFileAndNotifyTranscriptionService()` is called to trigger the authentication process from the frontend. Figure 4.3 illustrates the authentication and security process. After triggering, the Java backend generates a signed token for the Transcription Service and passes the Transcription Service API URL back to `upload.js` as shown in step 2 to step 4. The request to the Transcription Service is sent based on the signed token and API URL received. Once the request is validated by `validateToken()` in `authController.ts`, the transcription request will pass to `speech-to-text.ts`.

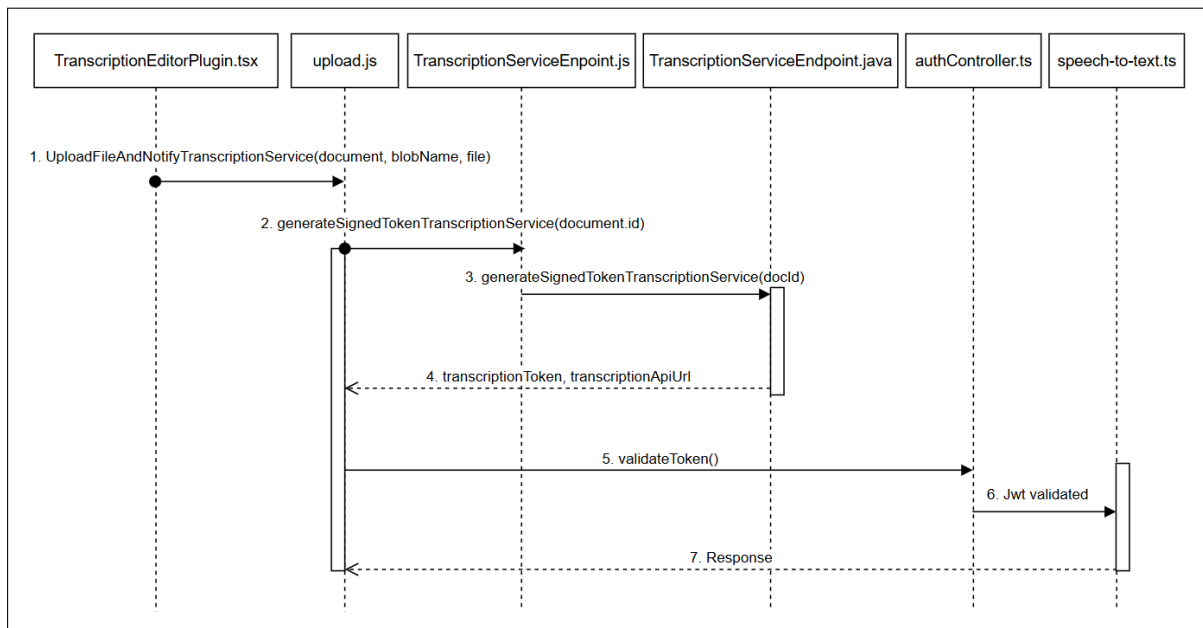


Figure 4.3: Authenticate Process

4.4 Transcription Process and Speaker Diarization

4.4.1 Transcription Process

The transcription service in the Java backend is refactored to a new Transcription Service using Node.js (TypeScript) and Express.js. Figure 4.4 shows this Transcription Service with its steps. The new endpoint `/TranscriptionService` is created in the Java backend to generate a signed token or a URL to the frontend. As discussed in previous sub-chapters, `UploadFileAndNotifyTranscriptionService()` in `upload.js` is called when receiving a request from a user. After uploading the media file, it then passes the request to the API of the Transcription Service for the transcription job as step 5 in the Figure 4.4 shows. A `jobId` is generated and sent back to `upload.js`, so that the frontend can check the progress of the transcription by calling `getTranscriptionStatus()` using the receiving `jobId`.

The transcription request to GCP STT API is generated through the `longRunningRecognize()` in `speech-to-text.ts`. Before receiving the transcription results from GCP STT API, it keeps receiving the transcription progress which is passed to the frontend to keep the user updated. Once the transcription job is done, the progress reaches 100%, and the

transcription results are sent back to `upload.js` as step 8 shows. The transcription results first get formatted by using `getFormattedTranscript()` function in `TranscriptionAdapter.js` and then its content is uploaded to GCS. After uploading, `TranscriptionEditorPlugin.tsx` gets the `contentDownloadUrl` by calling `getSignedDownloadUrlsForDocumentId()` and then downloads the content through that URL to display the results on the UI for the user.

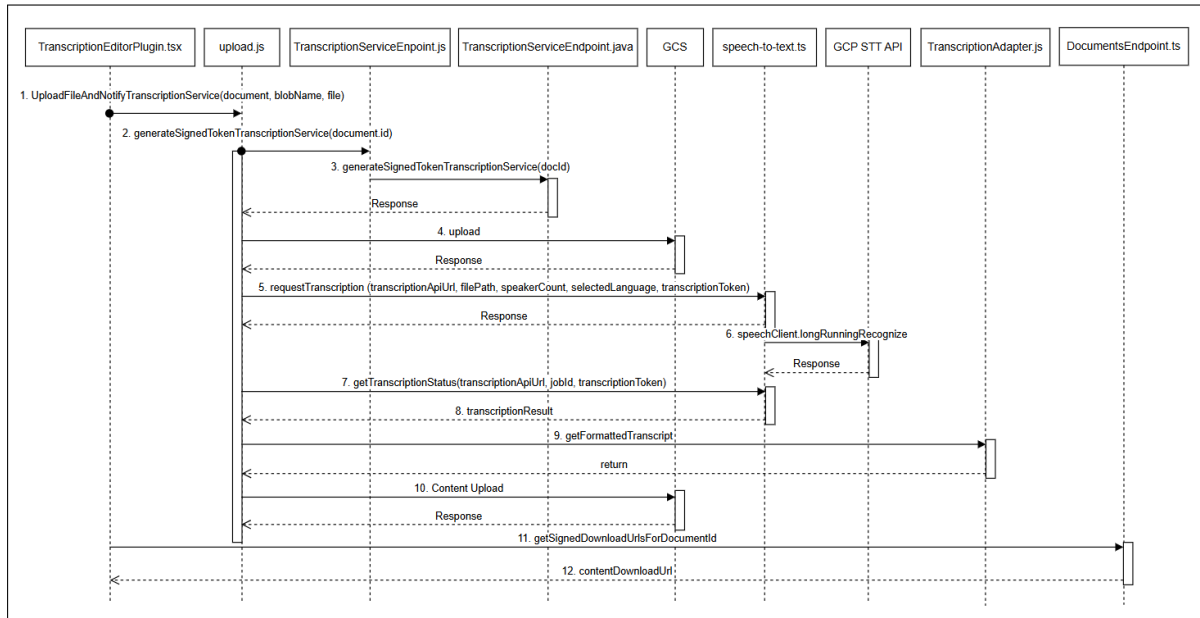


Figure 4.4: Transcription Process

4.4.2 Speaker Diarization

To make a transcription request with speaker diarization to GCP STT API, context information is needed as described in chapter 3. First, the basic information about the audio file, such as the file path, the sample rate and the language must be provided. The file path is where GCS stores the audio file. The sample rate and encodings are determined once the user uploads the file. Second, the language must be parsed. As described in chapter 3.3, the language is initially set to the user's default but can be changed at any time using the language selector. To enable speaker diarization, and as a third input, the `diarizationConfig` is needed along with the request. Fourth, the speaker count is also required. In this project, the user is asked to provide the speaker count before uploading the file. Fifth, to provide better transcription results, this project includes the telephony model in the transcription request. Figure 4.5 shows the code of the complete transcription request to GCP STT.

```

const [operation] = await speechClient.longRunningRecognize({
  audio: {
    uri: finalPath,
  },
  config: {
    encoding,
    sampleRateHertz: fileExtension === 'mp3' ? sampleRate : undefined,
    languageCode: selectedLanguage,
    diarizationConfig: {
      enableSpeakerDiarization: true,
      minSpeakerCount: 1,
      maxSpeakerCount: speakerCount,
    },
    model: 'telephony',
  },
});

```

Figure 4.5: Transcription Request

In `bbc/react-transcript-editor`, the speaker diarization display is sometimes inaccurate. Even though the data has speaker label, it sometimes displays as unknown speaker. The root cause lies in `groupWordsBySpeaker()` and `findSegmentForWord()` function in `group-words-by-speakers.js` in `bbc/react-transcript-editor` library. In the function, `paragraph.speaker` is set to unknown whenever the speaker changes, thus it does not correctly update it to the new speaker. To fix this issue, the code is modified to enforce the correct display of speaker diarization in `qdacity/react-transcript-editor`, a fork of `bbc/react-transcript-editor`. This own `qdacity` version is then published to a new version of Node Package Manager (NPM) package. Therefore, this project uses `qdacity/react-transcript-editor`¹ instead of `bbc/react-transcript-editor` to fix the unknown speaker issues.

Figure 4.6 shows the original display of speaker labels in the transcription editor, where unknown speakers were labeled as TBC0, TBC1, TBC2, and so on before the implementation. After successfully implementing the speaker diarization feature from Google Cloud STT, processing the data into `bbckali` structure, and fixing unknown speakers issues in `bbc/react-transcript-editor` library, the speaker labels are correctly displayed on the transcription editor as Figure 4.7 illustrates.

¹`qdacity/react-transcript-editor` - npm package. <https://www.npmjs.com/package/@qdacity/react-transcript-editor>












 TBC 0	00:00:02	I'm here.
 TBC 1	00:00:04	Hi. I'd like to buy a Chrome Cast and I was wondering whether you could help me with that.
 TBC 2	00:00:10	Hulu, which color would you like? We have you block and what?
 TBC 3	00:00:15	Let's get the black one.
 TBC 4	00:00:18	Okay, good. Would you like the new Concorde V remodel or the regular Comcast?
 TBC 5	00:00:24	Regular Chrome Cast is fine.
 TBC 6	00:00:28	Okay. Short. Would you like to ship it regular or Express?
 TBC 7	00:00:33	Express please.
 TBC 8	00:00:36	Terrific. It's on the way. Thank you very much. Thank you.
 TBC 9	00:00:42	Bye.
 TBC 10	00:00:43	Bye.

Figure 4.6: Original Speaker Labels Display on Transcription Editor










 SPEAKER_1	00:00:02	Okay I'm here.
 SPEAKER_2	00:00:04	hi I'd like to buy a Chromecast and I was wondering whether you could help me with that.
 SPEAKER_1	00:00:10	uh certainly which color would you like we have blue black and red.
 SPEAKER_2	00:00:15	let's get the Black 1.
 SPEAKER_1	00:00:18	uh okay great would you like the new Chromecast Ultra model or the regular Chromecast.
 SPEAKER_2	00:00:26	Chromecast is fine.
 SPEAKER_1	00:00:28	okay sure would you like to ship it regular or Express terrific it's uh on the way thank you very much.
 SPEAKER_2	00:00:38	thank you.
 SPEAKER_1	00:00:42	bye.

Figure 4.7: New Speaker Labels Display on Transcription Editor

4.4.3 Rename Speaker

When the user renames a speaker, the user now has the option to choose to rename all instances of this speaker, or only one instance of the speaker. As Figure 4.8 shows, the checkbox on the prompt window is implemented. Figure 4.9 illustrates the speaker diarization in the transcription editor after the user has renamed all instances of a speaker.

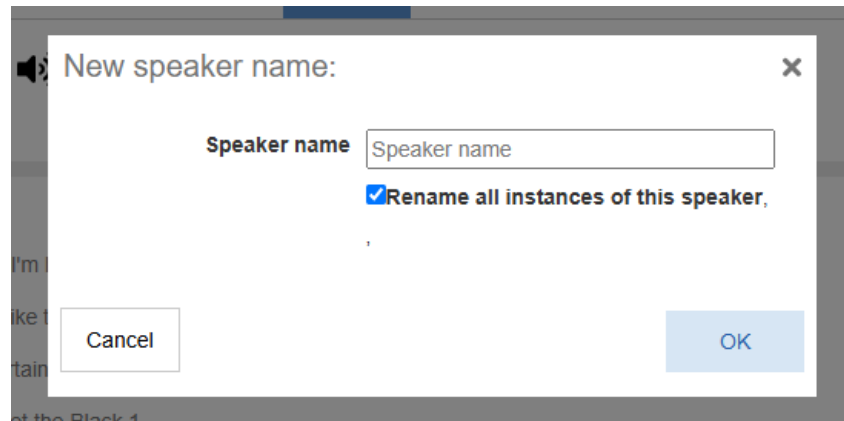


Figure 4.8: Rename Speaker Window

JOHN	00:00:02	Okay I'm here.
ALICE	00:00:04	hi I'd like to buy a Chromecast and I was wondering whether you could help me with that.
JOHN	00:00:10	uh certainly which color would you like we have blue black and red.
ALICE	00:00:15	let's get the Black 1.
JOHN	00:00:18	uh okay great would you like the new Chromecast Ultra model or the regular Chromecast.
ALICE	00:00:26	Chromecast is fine.
JOHN	00:00:28	okay sure would you like to ship it regular or Express terrific it's uh on the way thank you very much.
ALICE	00:00:38	thank you.
JOHN	00:00:42	bye.

Figure 4.9: Rename All Instances of Speaker

4.5 Error Handling & Logging

In this section, the discussion focuses on the error handling. Figure 4.10 illustrates that process. If the user uploads an unsupported type of an audio file as discussed in Chapter 3.7, the pop-up notification window will display the corresponding error message. The upload and authentication process are the same as previously discussed in Chapter 4.2 and Chapter 4.3. As step 4 depicts in Figure 4.10, the message is sent from Google Cloud STT, which then gets formatted in `STTError.js` for simpler explanations in English and German. The formatted message is then sent as the parameter for the `openNotification` function to inform the user via a UI notification as illustrated in Figure 4.11.

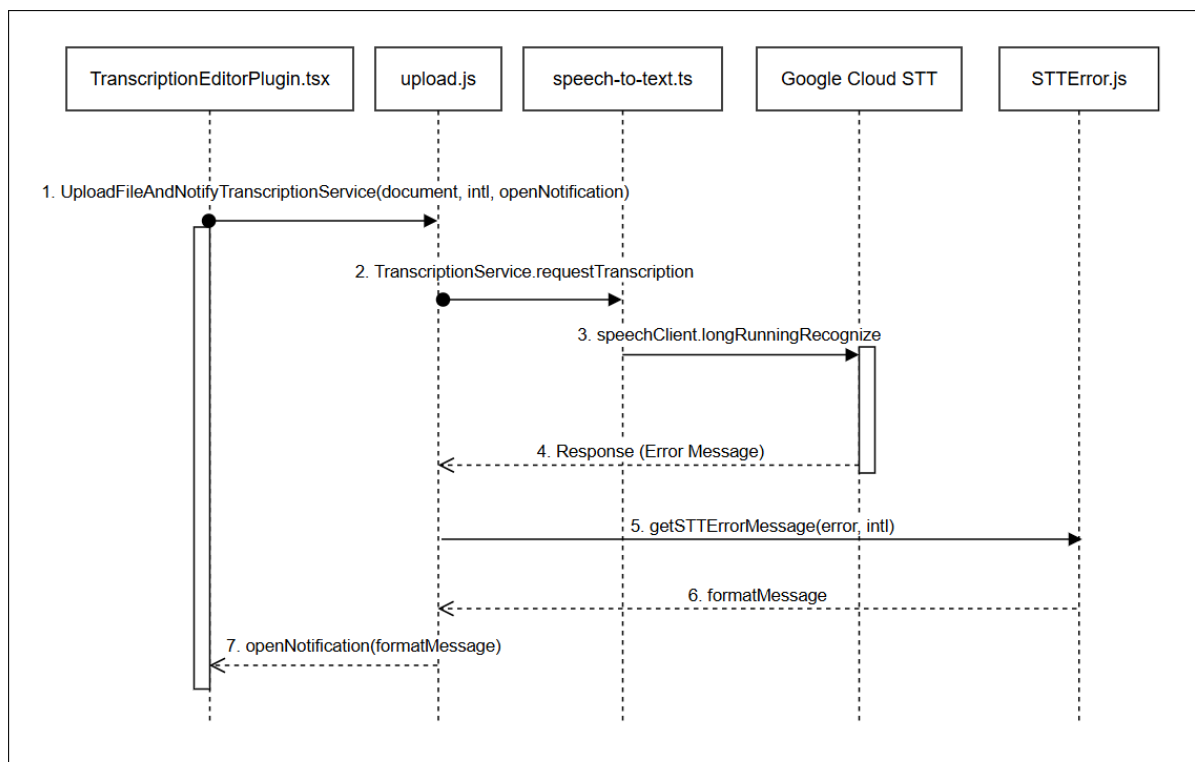


Figure 4.10: Error Handling Process Diagram

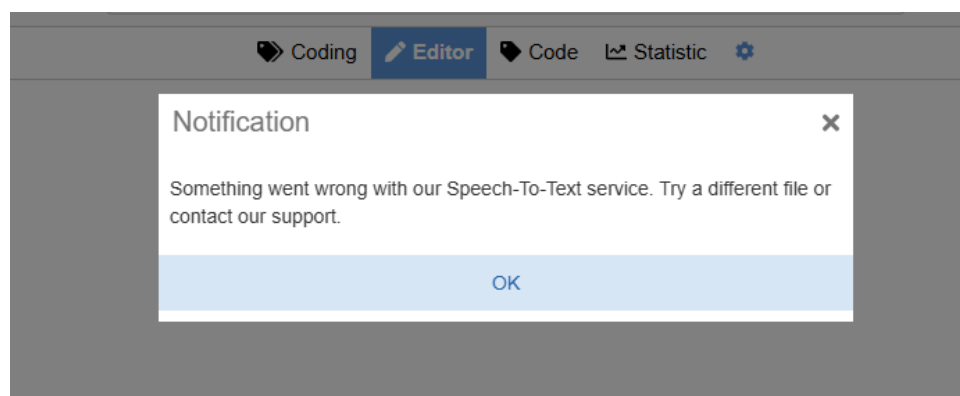


Figure 4.11: Error Notification Window

4.6 UI UX Improvement

4.6.1 Transcription Editor Redesign

The original design of the transcription editor lacked integration with the QDAcity design system and was not user-friendly. Hence, the UI of the transcription editor has been redesigned. Figure 4.12 shows the original design of the transcription editor, whereas Figure 4.13 shows the improved version. As these Figures illustrate, the media player has been redesigned.

Notable changes include:

- The background color of the media player has changed from light grey to white.
- The rewind button has been removed.
- The media player buttons no longer have borders.
- The export button is now larger and labeled “Convert to Text.”
- The progress bar is now light blue and visually reflects playback progress.
- The current time and total duration of the audio are displayed on the left and right sides of the progress bar, respectively.

The UI and UX is now more modernized and user-friendly.

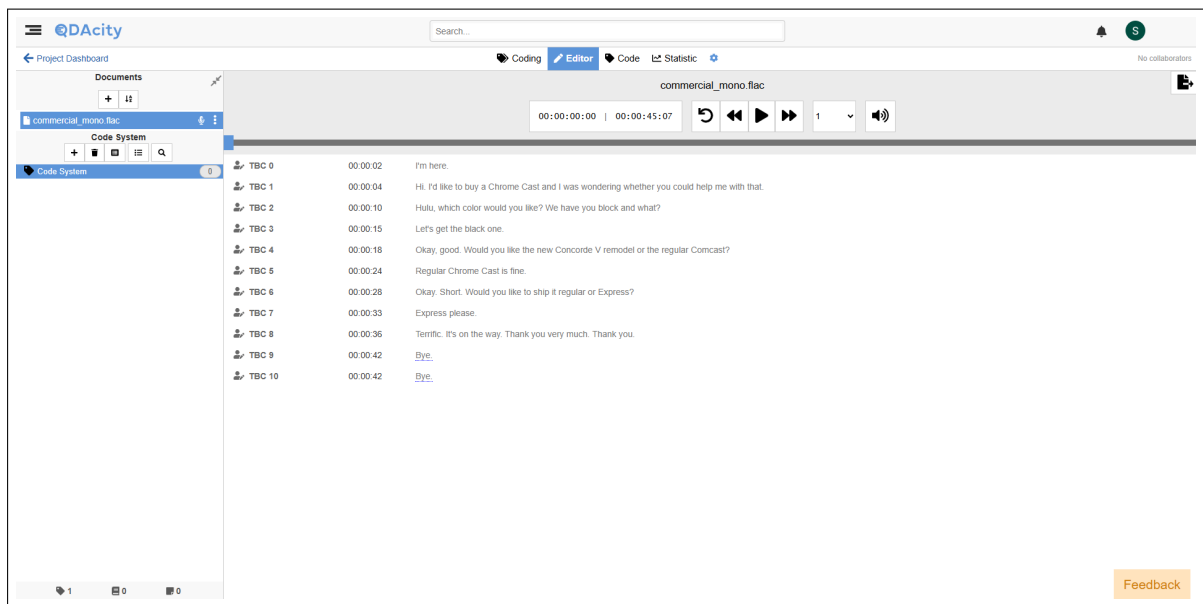


Figure 4.12: Original Transcription Editor UI

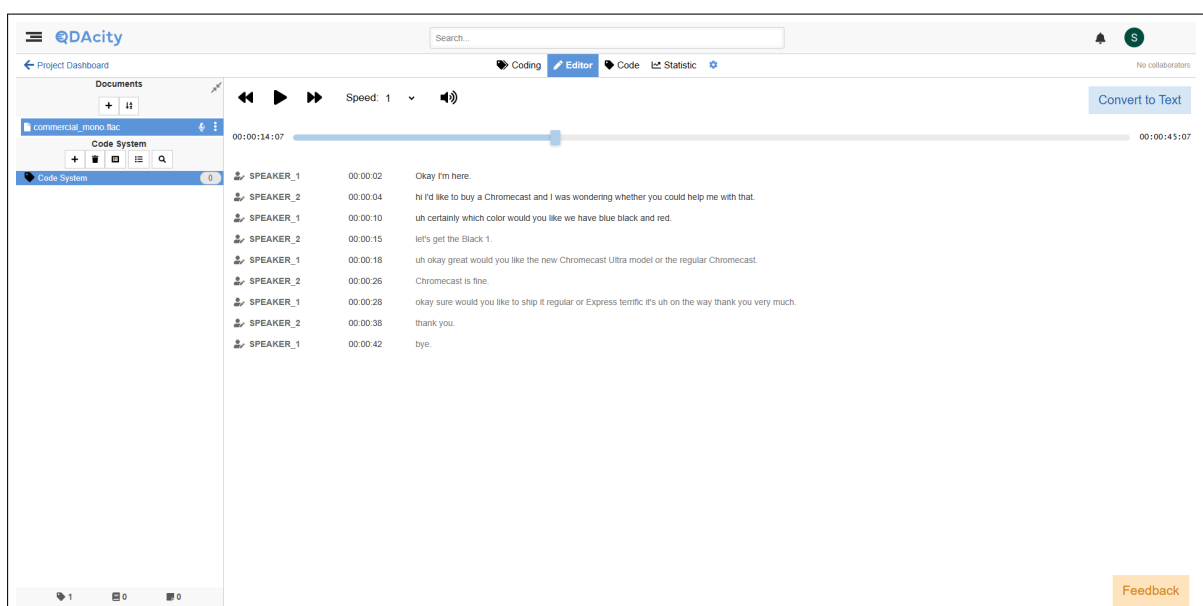


Figure 4.13: New Transcription Editor UI

There are also improvements for hotkeys in the transcription editor. The previous keyboard shortcuts were counterintuitive to users. In this thesis, the keyboard shortcuts have been revised to be more intuitive by replacing the unrelated English letter keys with arrow keys and substituting the `Alt` key with the `Ctrl` key.

As shown in Table 4.1, the shortcuts for play, skip forward, skip backward, decrease and increase the playback rate have been updated. However, the shortcut for setting the current time remains `Alt + T`, as `Ctrl + T` is already reserved within QDAcity.

Action	Original Hotkeys	New Hotkeys
Play/Pause	Alt + K	Ctrl + Space
Skip Forward	Alt + L	Ctrl + Right Arrow
Skip Backward	Alt + J	Ctrl + Left Arrow
Decrease Playback Rate	Alt + -	Ctrl + Down Arrow
Increase Playback Rate	Alt + =	Ctrl + Up Arrow
Set Current Time via Prompt	Alt + T	Alt + T (Remains the same)

Table 4.1: Comparison of keyboard shortcuts for media playback controls

4.6.2 Transcription Editor Tour

Figure 4.14, Figure 4.15 and Figure 4.16 show the transcription editor tour. The transcription editor tour is designed to help first-time users navigate the interface more effectively.

The first step introduces the media player, which is located on top of the transcription editor as Figure 4.14 illustrates. The user can interact with the media player by clicking or using hotkeys. Figure 4.15 shows the second step of the tour that introduces the transcription area where the transcription results are shown. In this area, the user can modify the speaker’s name by clicking on it and renaming it. The last step of the introduction, and the most important one, is the export button, depicted in Figure 4.16. The export button was small and not obvious in the old transcription editor UI, making it difficult for users to navigate. This was the main reason of implementing this transcription editor tour. The new transcription editor UI design not only makes the export button more visible but also more intuitive with the text label “Convert to text” on it, instead of an “exit door icon”. Now, with the improved UI design, and the transcription editor tour, the user can understand better the correct usage of this button.

Overall, the transcription editor tour briefly shows the features of the transcription editor to the user, providing a better user experience.

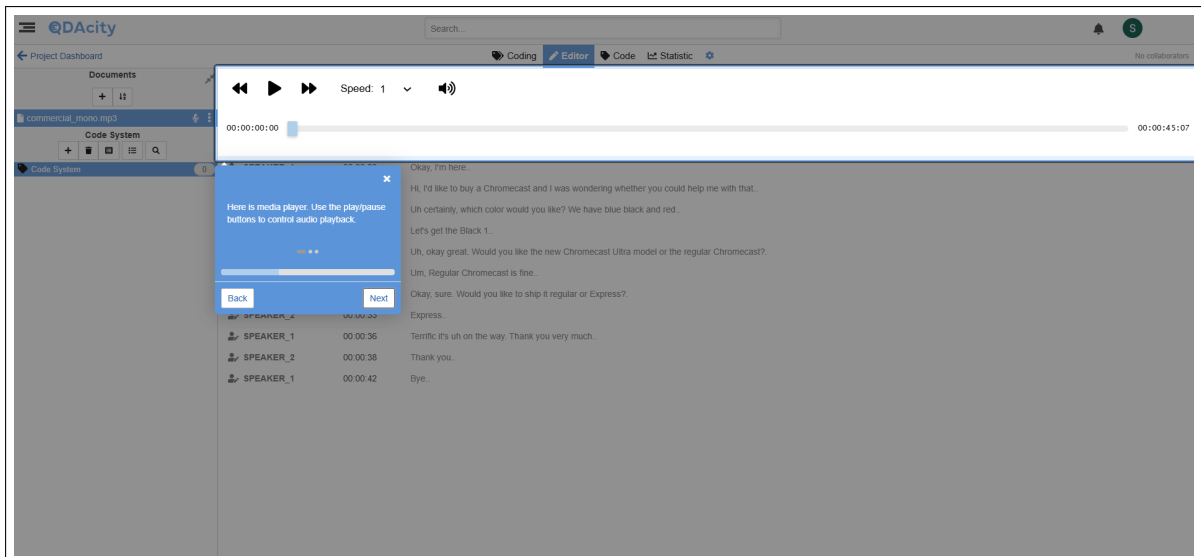


Figure 4.14: Transcription Editor Tour - Step 1

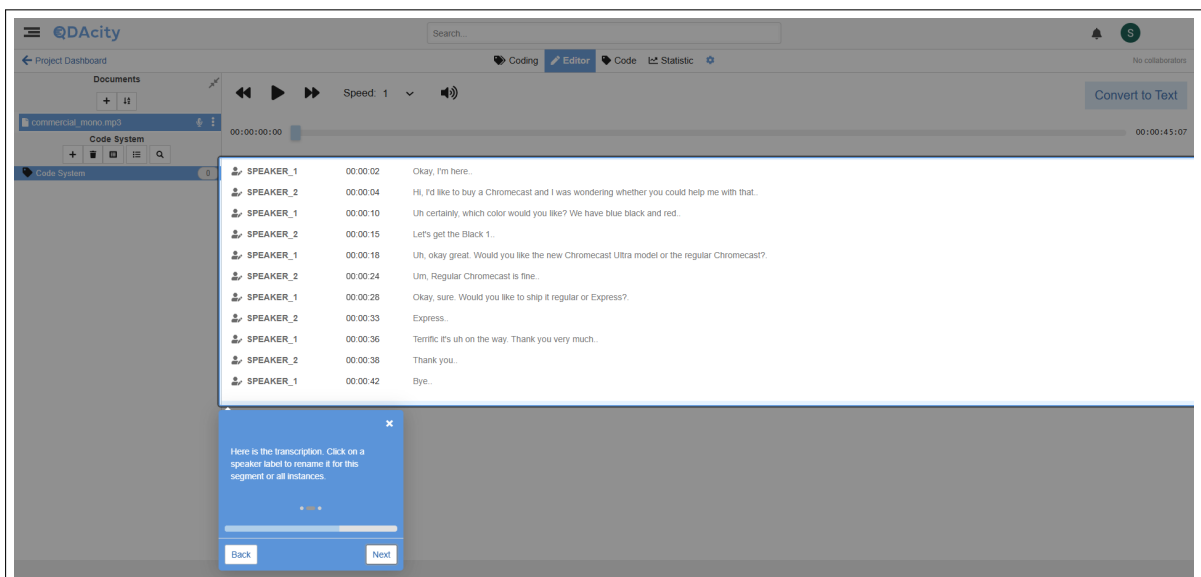


Figure 4.15: Transcription Editor Tour - Step 2

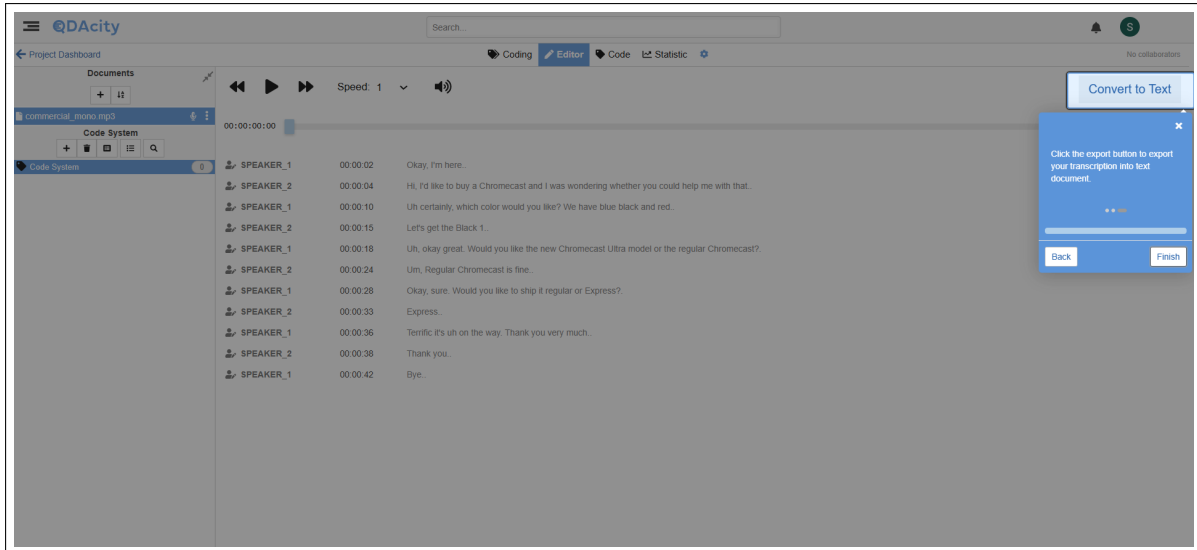


Figure 4.16: Transcription Editor Tour - Step 3

4.7 Cloud Deployments and Release

The Transcription Service is deployed and versioned in sync with the Java backend and the Collaborative-Editing-Service (CES) to make it easier. Thus, the versions are deployed and tagged like CES, according to the backend version number (GAE version). The version is determined using the `gcloud` Command Line Interface (CLI) in the deployment scripts. In terms of tagging the deployed version, for the CES it is tagged with `gae-` followed by the app-engine version number. Thus, to indicate compatibility, the CES revision corresponding to GAE version 100 is tagged as `gae-100`. This works the same way for the Transcription Service. The prefix in the Transcription Service tag is `gae-`.

In addition to the revision tags, the latest deployment of the Transcription Service is configured to receive the latest traffic. Through the tags, a new URL with a prefix is generated. As Figure 4.14 shows, the latest deployment has a revision tag `gae-1` with 100% traffic.

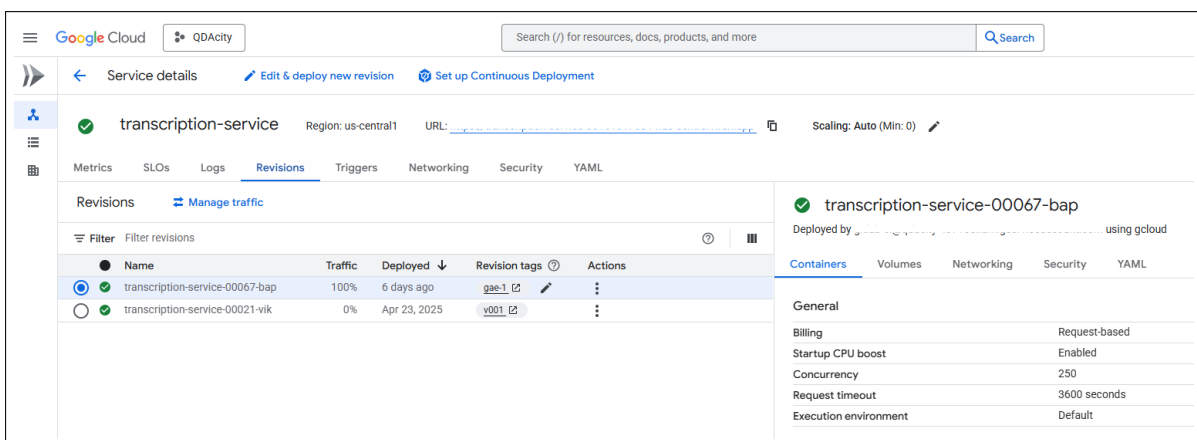


Figure 4.17: Cloud Deployment

5 Evaluation

This chapter evaluates the implementation of the requirements outlined in Chapter 2 by comparing them to the solution presented in Chapter 4. The evaluation is grouped by functional requirements and non-functional requirements. The goal of this chapter is to evaluate if the requirements are fulfilled.

5.1 Functional Requirements

This section evaluates the functional requirements, with a focus on the new transcription service infrastructure, the authentication and security, the speaker diarization, the error handling and the UI UX improvements.

5.1.1 New Transcription Service Infrastructure

The new transcription service infrastructure requirement fulfillment overview is shown in Table 5.1. The new service is implemented in TypeScript and runs in a containerized NodeJS environment, satisfying requirements FR-1.1 and FR-1.2. The API of the new service uses ExpressJS, according to FR 1-3. When the user uploads a file, HTTP requests are sent to the Transcription Service. The Transcription Service then sends the transcript requests to Google Cloud STT API. Thus, FR-1.4 and FR-1.5 regarding the initiation and the process of the transcription job are fulfilled.

The Transcription Service supports MP3, FLAC and WAV file format, and checks the file types and sample rate before sending the request to Google STT API. When the user uploads an unsupported audio format, the notification window is shown with an error message, satisfying FR-1.6 to FR-1.8. The Transcription Service uses the music metadata library to check the length of the audio when the user uploads an audio file according to FR-1.9. The transcription result is sent to the client as an object. When the client sends the data to be stored in GCS, it converts that object to a JSON string. Therefore, FR-1.10 is fulfilled.

The Transcription Service is built, tested and ultimately deployed to Google Cloud Run using the QDAcity GitLab CI pipeline, satisfying FR-1.11 and FR-1.12.

Table 5.1: New Transcription Service Infrastructure - Requirements Fulfillment Overview

Requirement	Description	Status
FR-1.1	Node.js implementation	Fulfilled
FR-1.2	TypeScript usage	Fulfilled
FR-1.3	Express.js framework	Fulfilled
FR-1.4	HTTP request initiation of transcription tasks	Fulfilled
FR-1.5	Google Cloud STT API processing	Fulfilled
FR-1.6	Audio format support (MP3, FLAC, WAV)	Fulfilled
FR-1.7	File type and sample rate validation	Fulfilled
FR-1.8	Error notification for unsupported formats	Fulfilled
FR-1.9	Audio length checking for quota control	Fulfilled
FR-1.10	JSON format transcription results	Fulfilled
FR-1.11	GitLab CI pipeline deployment	Fulfilled
FR-1.12	Google Cloud Run hosting	Fulfilled

5.1.2 Authentication and Security

Table 5.2 shows the requirement fulfillment overview for authentication and security. The system uses JWT-based validation for authentication. The Java backend generates the token using a private key, creates a public key API endpoint, provides an endpoint `/TranscriptionService` to send the signed token, signed upload URL and the Transcription Service API to the frontend.

In a development environment, the Transcription Service gets the public key from the endpoint and validates the token locally. In production, the public key is retrieved from the Google Datastore. Once the user is validated, it passes the request to speech-to-text for the transcription job. If the token is invalid, the system will receive an error message.

The requirements FR-2.1 to FR-2.7 regarding authentication and security have been fulfilled.

Only the members with organizer or owner roles of the project are allowed to start new transcriptions. This is not yet implemented. So far, every member with access to the projects can initiate a transcription. Thus, the requirement FR-2.8 regarding Role-Based Access Control (RBAC) has not been fulfilled.

Table 5.2: Authentication and Security - Requirements Fulfillment Overview

Requirement	Description	Status
FR-2.1	JWT-based authentication implementation	Fulfilled
FR-2.2	Signed tokens/URL with expiration time	Fulfilled
FR-2.3	Public key API endpoint	Fulfilled
FR-2.4	Transcription Service API exposure	Fulfilled
FR-2.5	Public key caching in memory	Fulfilled
FR-2.6	Public key retrieval when missing	Fulfilled
FR-2.7	Invalid/expired token error handling	Fulfilled
FR-2.8	Role-Based Access Control	Not Fulfilled

5.1.3 Speaker Diarization

The speaker diarization requirement fulfillment overview is shown in Table 5.3. The Transcription Service utilizes Google Cloud STT speaker diarization and displays the results correctly on the UI using qdacity/react-transcription-editor library, satisfying FR-3.1 and FR-3.2.

The speaker count is needed for making a GCP STT request with speaker diarization. Thus, an input box for the user to provide the number of speaker is implemented on the UI which only allows positive number. The number of speaker count is then sent to the Transcription Service. Therefore, the requirements FR-3.3 and FR-3.4 regarding the input box for speaker count have been fulfilled.

When the user renames a speaker, the rename speaker prompt window shows a checkbox for the user to choose if he or she wants to rename only one instance or all instances of the speaker, satisfying FR-3.5 to FR-3.6.

Table 5.3: Speaker Diarization - Requirements Fulfillment Overview

Requirement	Description	Status
FR-3.1	Google Cloud STT diarization integration	Fulfilled
FR-3.2	Accurate diarization data representation	Fulfilled
FR-3.3	UI input box for speaker count	Fulfilled
FR-3.4	Positive number validation for speaker count	Fulfilled
FR-3.5	Global speaker renaming functionality	Fulfilled
FR-3.6	Checkbox for rename all instances of speaker	Fulfilled

5.1.4 Error Handling

Table 5.4 shows the requirement fulfillment overview for error handling requirements. If the user uploads an unsupported type of audio file, a pop-up notification window with an error message will show up. Instead of just forwarding the messages from Google Cloud STT, the error message is simplified and formatted in English and German. In this way, the user receives a more readable feedback from the message display and understands how to solve the problems.

The requirements FR-4.1 to FR-4.5 regarding error handling have been accomplished.

Table 5.4: Error Handling - Requirements Fulfillment Overview

Requirement	Description	Status
FR-4.1	UI error messages for common issues	Fulfilled
FR-4.2	Simplified Google Cloud STT error relaying	Fulfilled
FR-4.3	Popup notification error display	Fulfilled
FR-4.4	Bilingual error messages (German/English)	Fulfilled
FR-4.5	Error logging for debugging/monitoring	Fulfilled

5.1.5 UI/UX Improvements

Table 5.5 shows the requirement fulfillment overview of the UI and UX improvements. The transcription editor interface has been redesigned, with everything conforming to the QDAcity style (e.g., colors, button design). As a result, FR-5.1 and FR-5.2 have been fulfilled. Furthermore, a transcription editor tour has been introduced to support users visiting the transcription editor for the first time. The tour consists of three steps introducing the media player, the transcription area and the export button. This tour makes it easier for the user to navigate the interface and understand that he/she needs to click the export button to export the transcription in text format. FR-5.3 has thus been satisfied.

Table 5.5: UI/UX Improvements - Requirements Fulfillment Overview

Requirement	Description	Status
FR-5.1	Transcription interface redesign	Fulfilled
FR-5.2	QDAcity design system conformance	Fulfilled
FR-5.3	Transcription editor tour	Fulfilled

5.2 Non Functional Requirements

In this section, non-functional requirements are evaluated, with a focus on eight categories of software product quality. Table 5.6 shows the requirement fulfillment overview of the non-functional requirements.

Functional Suitability. The functional suitability requirement NFR-1.1 has been fulfilled. The Transcription Service integrates with the existing QDAcity application, providing the necessary transcription capabilities without breaking the current workflow or system architecture.

Performance Efficiency. The performance efficiency requirement NFR-2.1 has been fulfilled. With a configuration of 500 MB memory limit for the container. The cloud run server shows peak usage around 28%, which is less than 150 MB. NFR-2.2 regarding response time has been partially fulfilled. While the system generally operates within acceptable parameters, transcription processing can occasionally take up to 32 seconds, which exceeds the optimal response time expectations for user experience.

Compatibility. The compatibility requirements NFR-3.1 to NFR-3.2 have been satisfied. The Transcription Service integrates into the existing QDAcity technology stack,

using Node.js, Express.js and Google Cloud Run. This ensures operation within the current infrastructure and maintains consistency with the established development framework. Being tested on different browser, the frontend components are compatible with Google Chrome (version 137.0.7151.120) , Firefox (version 139.0.4) and Safari (version 18.5). Therefore, NFR-3.3 has been fulfilled. However, the new backend transcription service works only in Google Chrome, but does not operate correctly in Firefox or Safari. Although this was not part of NFR-3.3, future development should take that into account.

Usability. The usability requirements NFR-4.1 to NFR-4.5 have been addressed. The new transcription editor UI design remains visual consistency with the existing QDAcity style. The upload and transcription progress bars are displayed to provide real-time feedback during the transcription process. The transcription results are presented with speaker labels and timestamps for identification and navigation. Additionally, the keyboard hotkeys for the media player in the transcription editor have been redesigned to be more intuitive, enhancing the overall user experience.

Reliability. The reliability requirement NFR-5.1 has not been fulfilled. The system currently lacks an upload limit of 3 attempts, requiring further development.

Security. The security requirements NFR-6.1 to NFR-6.2 have been implemented. Each endpoint checks the authorization of requests and the requests are sent via HTTPS.

Maintainability. The maintainability NFR-7.1 to NFR-7.4 have been fulfilled. The codebase follows standard naming convention. All API endpoints are documented. The Transcription Service's setup and interactions are documented in the QDAcity Wiki. NFR-7.5 and NFR-7.6 have not been fulfilled, however. The unit tests and integration tests have not been implemented, requiring further development to fulfill the requirements.

Portability. The portability requirement NFR-8.1 has been done. The system architecture supports deployment across different environments and platforms, ensuring flexibility in system deployment and operation.

Table 5.6: Non-Functional Requirements - Fulfillment Overview

Requirement	Description	Status
Functional Suitability		
NFR-1.1	Integration with existing QDAcity system	Fulfilled
Performance Efficiency		
NFR-2.1	Memory usage limit (500MB max)	Fulfilled
NFR-2.2	Response time limit (30s for 1min audio)	Partially Fulfilled
Compatibility		
NFR-3.1	QDAcity technology stack integration	Fulfilled
NFR-3.2	Node.js and Express.js compatibility	Fulfilled
NFR-3.3	Browser compatibility	Fulfilled
Usability		
NFR-4.1	Clear navigation and user assistance	Fulfilled
NFR-4.2	Consistent QDAcity design	Fulfilled
NFR-4.3	Progress bar for user feedback	Fulfilled
NFR-4.4	Speaker labels and timestamps display	Fulfilled
NFR-4.5	Keyboard hotkeys	Fulfilled
Reliability		
NFR-5.1	Upload retry limit (3 attempts)	Not Fulfilled
Security		
NFR-6.1	Endpoint authorization checking	Fulfilled
NFR-6.2	HTTP requirement	Fulfilled
Maintainability		
NFR-7.1	Standard naming conventions	Fulfilled
NFR-7.2	OpenAPI documentation	Fulfilled
NFR-7.3	Code comments for functions	Fulfilled
NFR-7.4	Architecture documentation in Wiki	Fulfilled
NFR-7.5	Unit test coverage (80% minimum)	Not Fulfilled
NFR-7.6	Integration test coverage	Not Fulfilled
Portability		
NFR-8.1	Containerized deployment support	Fulfilled

6 Future Improvement

Although the new Transcription Service, authentication, speaker diarization and error handling have been implemented in QDAcity, there are still some aspects that can be improved. In this chapter, the future improvements are presented. They provide a foundation for future research and development of QDAcity.

6.1 Role Based Access Control

As described in Chapter 5, requirement FR-2.8 regarding the project member roles in Chapter 2.1.2 has not been fulfilled. To manage collaborator's rights, RBAC should be implemented, so that only project members with organizer or owner roles can create new transcriptions. This should not only be implemented in the backend of QDAcity, but also documented on QDAcity's documentation¹.

6.2 Unsupported Language for Speaker Diarization

Google Cloud STT speaker diarization feature works for English, but it does not support other languages at the moment. If the selected language is not supported for speaker diarization in Google Cloud STT, the API still returns a transcription; however, speaker labels are omitted.

As shown in Figure 6.2, when the user uploads an audio file in an unsupported language, Brazilian Portuguese in this case, the transcription result is displayed as if spoken by a single speaker, even though the audio contains two distinct speakers.

¹QDAcity Documentation - Project Member Types <https://qdacity.com/docs/manage-project/project-members/>

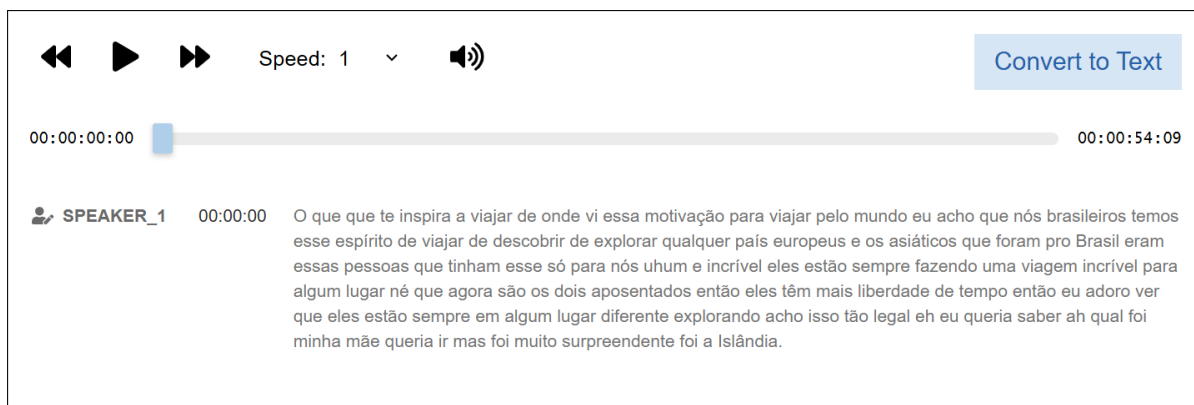


Figure 6.1: Unsupported Language Speaker Diarization

To address this issue, unsupported languages for speaker diarization have to be first filtered out. The list has to be continuously maintained based on Google Cloud STT's documentation ². Subsequently, the logic in the `transcriptionAdapter` needs to be modified for those unsupported languages. For example, words can be grouped into paragraphs when the time gap between the `endTime` of one word and the `startTime` of the next exceeds a certain threshold (e.g., 0.8–1.0 seconds). For each paragraph, the speaker tag count is incremented by one. In this way, the transcription result will be displayed in paragraphs, each with an increasing speaker tag number. Even though the speaker diarization does not work, the transcription will no longer appear as if there is only one speaker.

6.3 Transcription Service on Different Browser

As described in Chapter 5, requirement NFR-3.3 regarding the browser compatibility requires further development. The new backend Transcription Service does not function correctly in Firefox or Safari. Although this was not a requirement for the work of this thesis, the frontend components work on different browsers as required. Further investigation is required to resolve this issue in the backend. The root cause have to be first identified, and cross-browser feature compatibility testing should also be conducted. If necessary, browser-specific fallbacks or conditional logic can be implemented. In addition, third-party library compatibility should be reviewed to ensure full support across all major browsers.

²Google Speech-to-Text supported languages <https://cloud.google.com/speech-to-text/docs/speech-to-text-supported-languages>

7 Conclusion

QDAcity uses a new transcription service in Node.js, and the transcription editor now has the speaker diarization feature, error handling windows, a guided tour and a new UI design.

To successfully implement these features in QDAcity, the problems that could be improved with a clear objective in Chapter 1 were first analyzed. The necessary changes identified were to improve the scalability and UX of the QDAcity Transcription Service with five main areas, namely the refactoring of the Transcription Service, the enhancement of the authentication and security, the implementation of speaker diarization and the error handling as well as UI and UX improvements. To achieve the objectives, the functional requirements and non-functional requirements were specified in detail in Chapter 2, which lay the ground for the architecture and implementation part of Chapter 3 and 4. Those requirements were built using the template by Rupp (2014) to avoid linguistic ambiguity, so that the requirements are clear and testable.

In Chapter 3, the architecture for the implementation was discussed, The original system architecture of QDAcity was first presented, followed by the technology stack, Google STT and GCS. The reasons for using the music-metadata library and the transcription editor library were also explained in this chapter. In chapter 4, the implementations were described in depth. The audio file is first uploaded to GCS using the signed URL for security reasons. The communication among the Java backend, new transcription service and the frontend is handled via API calls and JWT-based validation. After validation, the Transcription Service sends the transcription request with speaker diarization enabled to the Google Cloud STT API. The transcription results are displayed on the frontend using `qdacity/react-transcription-editor` library. The UI and UX improvement of the transcription editor were also discussed in this chapter.

As discussed in detail in Chapter 5, evaluating the implementation of the requirements, out of 34 functional requirements, 33 have been fulfilled. FR-2.8 regarding RBAC has not been implemented due to time constraints. Chapter 6 outlines possible implementation strategies to address this shortcoming. Besides, out of 21 non-functional requirements, 17 have been fulfilled. Requirement NFR-2.2 regarding the response time has been partially fulfilled. While the response time is most of the time within 30 seconds, it sometimes can take up to 32 seconds. NFR-5.1 regarding the upload retry limit, NFR-7.5 and NFR-7.6 regarding the tests have not been implemented due to time constraints.

In conclusion, although some requirements have not been fulfilled, most of the requirements have been successfully implemented. The Transcription Service of QDAcity has been refactored from Java to Node.js, which is expected to improve the efficiency and

memory usage. The speaker diarization, error message popup window and new UI design have also been integrated into the current QDAcity software to provide better user experience. The user can now use the improved feature of QDAcity to conduct QDA.

References

- Alashqar, A. M., Abo Elfetouh, A., & El-Bakry, H. M. (2015). Requirement engineering for non-functional requirements [Information Systems Department, Faculty of Computer and Information Sciences, Mansoura University, EGYPT]. *International Journal of Information and Communication Technology Research*, 5(2).
- Bailey, J. (2008). First steps in qualitative data analysis: Transcribing. *Family Practice*, 25(2), 127–131. <https://doi.org/10.1093/fampra/cmn003>
- Evers, J. C. (2010). From the past into the future. how technological developments change our ways of data collection, transcription and analysis. *Forum Qualitative Sozialforschung / Forum: Qualitative Social Research*, 12(1). <https://doi.org/10.17169/fqs-12.1.1636>
- Google Cloud. (2025a). *Class longrunningrecognizemetadata* [Accessed: March 26, 2025]. Retrieved March 26, 2025, from https://cloud.google.com/python/docs/reference/speech/latest/google.cloud.speech_v1p1beta1.types.LongRunningRecognizeMetadata
- Google Cloud. (2025b). *Error messages* [Accessed: February 07, 2024]. Retrieved February 7, 2025, from <https://cloud.google.com/speech-to-text/docs/error-messages>
- Google Cloud. (2025c). *Speech-to-text transcription models* [Accessed: May 14, 2025]. Retrieved May 14, 2025, from <https://cloud.google.com/speech-to-text/docs/transcription-model>
- Güler, M. (2015). Case study: Ambitious growth target of bnp paribas in germany. *International Journal of Sales, Retailing and Marketing*, 4(9), 79–87.
- Ieee standard glossary of software engineering terminology. (1983). *ANSI/ IEEE Std 729-1983*, 1–40. <https://doi.org/10.1109/IEEESTD.1983.7435207>
- Lacey, A., & Luff, D. (2007). *Qualitative research analysis*. The NIHR RDS for the East Midlands / Yorkshire & the Humber.
- Resende, D. (2015, August). *Node.js high performance*. Packt Publishing Ltd.
- Rupp, C. (2014). *Requirements-engineering und -management: Aus der praxis von klassisch bis agil* (6., aktualisierte und erweiterte Auflage). Carl Hanser Verlag GmbH & Co. KG.
- Shyam Mohan, J. S., & Goswami, K. (2025). Performance analysis and comparison of node.js and java spring boot in implementation of restful applications. *Software: Practice and Experience*. <https://doi.org/10.1002/spe.3418>
- Stuckey, H. L. (2014). The first step in data analysis: Transcribing and managing qualitative research data. *Journal of Social Health and Diabetes*, 2(1), 6–8.
- Systems and software engineering — systems and software quality requirements and evaluation (square) — product quality model* (2nd ed.) [Status: Published; Stage: International Standard published [60.60]; Technical Committee: ISO/IEC JTC 1/SC

7; ICS: 35.080]. (2023, November). International Organization for Standardization and International Electrotechnical Commission.