School of Computing and Mathematical Sciences

CO7201 Individual Project

Final Report

SongAssist

Gabriel Knight
gk247@student.le.ac.uk
249047672

Project Supervisor: Newman Lau
Principal Marker: Dr Martina Palusa

Word Count: 10594
25/07/25

DECLARATION

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s). Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s). I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Gabriel Knight
Date: 27/06/25

Contents

Introduction

1.1 Problem Statement

For guitarists looking to learn a new song, considerable ambiguity can arise around deciding the most effective mode of *musical information retrieval* (*MIR),* especially for those with an untrained ear who cannot rely solely on the raw audio of a song to infer guitar-playing intricacies [*Wiggins, 2024*]. The presence of other instruments in this raw audio also frequently obscure the guitar part, making for a frustrating experience in attempting to identify what is being played [*Oxenham, et al, 2003*]. Recent developments in the field of *stem-separation*, technology that can take a raw audio file and isolate instruments from each other, have presented a promising solution to such MIR related issues [*Kharat, et al, 2021*].

Stem-separation has existed as a technology since the mid-1990s but it was with the emergence of AI driven models in the late-2010s that isolating instruments from mixed audio achieved professional-grade quality and practicality [*Araki, et al, 2025*]. Web applications such as *Moises.ai* feature stem-separation that can be performed on any audio file uploaded by the user, *Moises* advertises the separation offered as ‘studio-quality’, the application also delivers other advanced features such as automatic section identification [*Watcharasupat, Lerch, 2024*].

While existing stem-splitting applications offer impressive quality and valuable insight, no existing applications cater specifically to guitarists by providing additional assistance on learning each part of the song. After hearing an isolated guitar track using these programs, a guitarist will know what the guitar-part sounds like but won’t be much closer to actually replicating that in their own playing; searching the web or a separate application for accurate notation that clearly conveys chords and timing of the song is usually required [*Vasquez, et al, 2023*]. Research has shown that it can take over 20 minutes to re-enter deep focus after an interruption from context-switching [*Mark, Gonzalez, Harris, 2023*], for a guitarist constantly switching between a stem-splitting and notation application this can make for an inefficient learning workflow and a frustrating experience.

On top of seeking out chords for a song, it is necessary to understand how a guitarist has played each part, this is not always obvious by listening to an isolated part alone. Whether a song has been played using a plectrum or finger-picking technique can be very useful information when learning but this usually requires further research and context-switching which ultimately presents more interruptions to the workflow [*Josel, Tsao, 2021*].

The fact there is no leading application available on the market that attempts to consolidate the learning process of a guitarist into a single program that utilizes existing modern technologies often leads to a fractured user-experience with MIR taking longer than it should [*Martelloni, McPherson, Barthet, 2023*].

This report covers a proposed solution to this issue in the form of a web application called *SongAssist*.

1.2 SongAssist Overview

SongAssist is an all-in-one practice web application for guitarists to learn songs in a single focused environment with minimal interruptions to workflow. The application allows users to upload an audio file of any format with which the guitar will be isolated from all other instruments and vocals, *SongAssist's* user friendly interface makes it easy to quickly adjust the relative volumes of the backing track and guitar.

Once the user's file has been stem-separated, *SongAssist* can also provide AI-powered advice on how to play the song and is prompt-engineered to answer any of the user's questions throughout the learning process. Chord sheets for learning the user's song can also be AI-generated and saved within the corresponding project, if the user wishes to change the arrangement of the song or any inaccuracies are spotted in generated chords then manual changes can be made and accessed in the future.

SongAssist provides the functionality for users to create loops of the section they are currently learning, these sections can then be saved and labelled. The playback speed of the loaded song can be adjusted to fit the user's preference with straightforward UI elements for doing so. If the user wishes to download their separated stems to their local machine then this is also possible.

Figure 1 shows *SongAssist's* user interface when a project is loaded.

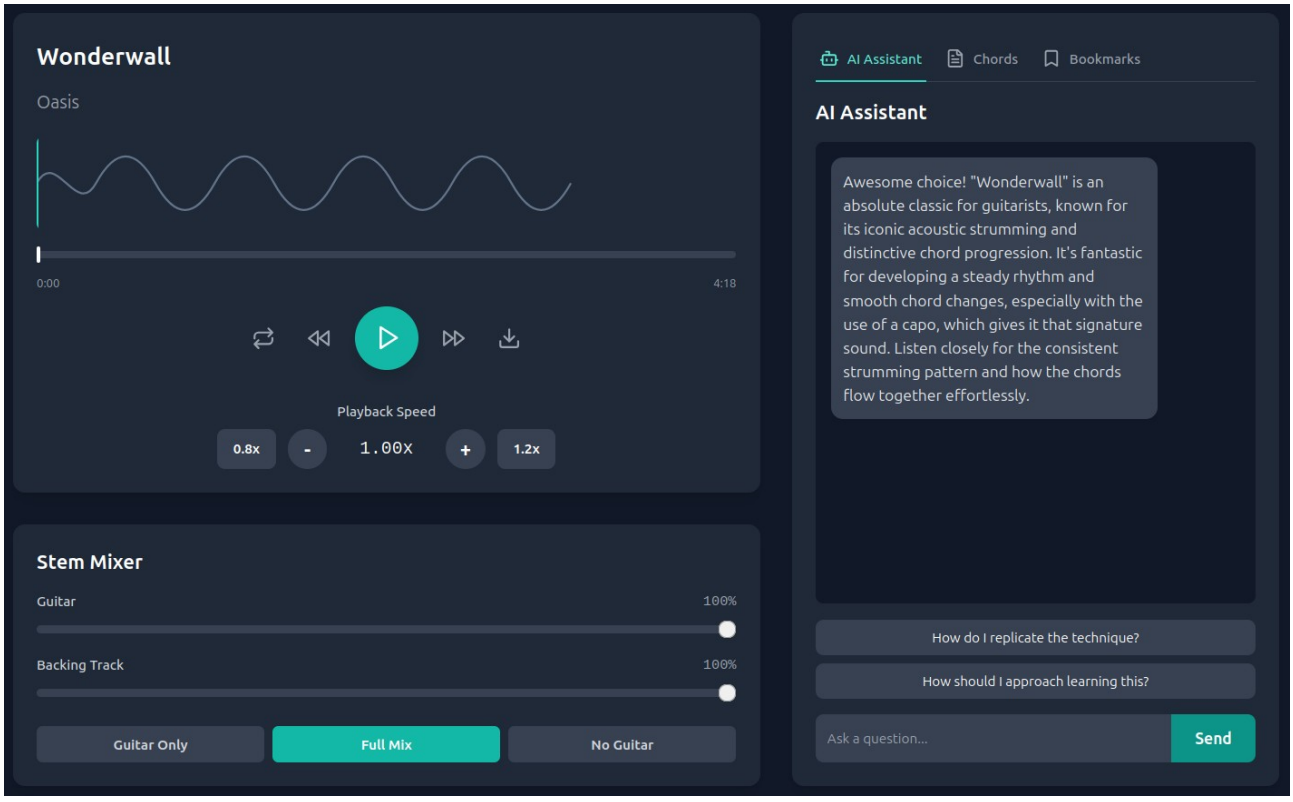

Figure 1: Screenshot of SongAssist application

Requirements

At the start of the project, requirements for *SongAssist* were divided into 3 categories: *essential, desirable and optional*. This section contains a list of the requirements within each category along with justification for each.

Essential

- *Implement AI-assisted stem-splitting instrument isolation functionality.*
  - This is *SongAssist's* core feature. As mentioned previously, the presence of other instruments when listening out for just the guitar part can be very distracting and sometimes even drown out the guitar part altogether. By allowing guitarists to listen to the part they are learning in isolation, distractions are prevented and workflow efficiency is increased.
- *Provide an AI assistant that gives advice on playing the song.*
  - Once the user has listened to the isolated guitar part and developed a proper understanding of how it should sound, immediate access to information on how it was played along with applied techniques provides a structured foundation for the next stage of the learning process.
- *Develop functionality for generating guitar tablature based on the MP3 file provided.*
  - Presenting tablature to the user clearly informs them of notes that need to be played at specific times using the previously advised techniques, further aligning with *SongAssist's* goal of consolidating all necessary forms of *MIR* within a single application.
- *Provide a straightforward user-interface.*
  - *SongAssist* is intended for use by all guitarists, including those who are not necessarily technologically proficient; therefore it is an essential requirement to ensure the interface is simple to use with all functionality presented clearly and logically.

Desirable

- *Enable users to create, edit, and manage multiple playlists of songs they are learning.*
  - This could be beneficial for musicians learning numerous songs simultaneously, such as performing musicians; by having all songs for each concert grouped together, a more organised workflow is supported.
- *Include an interactive stem mixer.*
  - As opposed to switching between the two separated stems for playback, an interactive stem mixer provides the user with a lot more freedom in how they want the song to sound. The prominence of guitar in studio-recorded songs can vary significantly; as a result, some backing tracks will require more volume reduction than others for the guitar to be heard clearly.
- *Incorporate a feature to adjust the playback speed of the MP3 file without distorting the pitch.*
  - Particularly fast or complex sections of a guitar-part can be difficult to assimilate even after isolation, thereby justifying the need for playback-speed adjustment. If the frequency of the notes can be unaffected by speed-adjustment then this enhances the accuracy of playback delivered to the user, allowing them to play along at the same pitch.
- *Implement a feature to add and delete bookmarks at specific timestamps within a song.*
  - Learning a song usually requires practising multiple distinct sections, functionality for quickly accessing these different sections in the form of user-defined bookmarks facilitates compartmentalization of the song and reducing cognitive-load.

Optional

- *Develop a feature to allow users to pitch shift their MP3 files without affecting playback speed.*
  - For guitarists seeking to transpose a song into a different key, this feature could show them how it would sound by shifting the isolated part to this key while maintaining the same playback-speed; as a result, the user can decide which key is optimal prior to learning.
- *Allow users to name or label their bookmarks for easier identification.*
  - Allowing users to name created bookmarks enhances navigational efficiency with a string of text being easier to identify than a timestamp.
- *Implement a waveform display of the audio track to allow for more precise bookmark placement.*
  - It is possible that a song's different sections could be visible within this waveform display, making it easy for the user to identify and instantly jump to the section they wish to hear, leading to further enhancement to user-experience.
- *Develop a customizable user-interface.*
  - To support different workflows that may prioritize certain features over others, providing users with a customizable UI could be beneficial for making these features more instantly accessible.

Literature Review

3.1 Stem-Separation

Early stem-separation methods in the 1990s relied on mathematical techniques such as *Independent Component Analysis (ICA)* whereby statistical patterns are identified in the signal of the audio in order to differentiate instruments from each other. Separation is performed by trying to find a linear transformation that can isolate instruments by maintaining maximum statistical independence of each estimated source [Hyvärinen & Oja, 2000].

It is assumed using the *ICA* technique that sources are independent from each other in that the behaviour of one signal does not contain any insight on the behaviour of another; this means that for audio where instruments overlap with one another, *ICA* isn't as effective [*Sawada, et al, 2019*]. An ideal audio mix for *ICA* is known as an overdetermined mixture, this is where the number of channels matches or exceeds the number of independent sources, stem separation quality in this case is proportionate to the ratio of channels to sources. Conversely, if there are less channels than sources then this is called an underdetermined mix whereby *ICA* performance drops significantly [*Cruces, 2015*]. With most studio recordings being mixed in stereo which consists of 2 channels, any more than 2 independent sources being present within the audio results in suboptimal performance using *ICA*; as a result, any stereo recording of a full band (vocals, guitar, drums, bass) playing will likely exceed the number of sources required for high quality separation [*Feng, Kowalski, 2018*]. These limitations are intensified in frequency-domain *ICA* by the permutation problem, a systemic issue in which signal origins become scrambled across different frequency bins (a discrete range of frequencies representing the energy of the signal in a short sample), making it difficult to reconstruct coherent stems [Sawada et al., 2004].

Independent Vector Analysis (*IVA*) was introduced as a natural extension of *ICA* in 2006 by [*Kim, Eltoft, Lee, 2006*] in an attempt to directly address the weaknesses in frequency-domain blind source separation. Where *ICA* treats each frequency bin independently of each other, *IVA* treats them together as vectors, this leads to much higher quality separation where the timbre of instruments like guitar is preserved significantly better. By using vectors to analyse statistical dependencies across bins, the *IVA* technique solved the permutation problem [*Hao, et al, 2010*]. A year after the emergence of this technology, *IVA* moved from the slower *gradient-based* scheme to become *auxiliary-function-based* which lead to notable improvements in speed and quality; it is stated that by employing auxiliary-functions, resulting separated stems suffered from less distortion [*Ono, 2012*]. While the introduction of *IVA* marked a consequential shift in the field of stem separation, quality of output from this technique still suffered from many similar limitations to that of *ICA*. One of the main drawbacks with using statistical methods is the dependence on mathematical assumptions that don't consistently apply across all songs, techniques such as *IVA* particularly struggle with dense musical arrangements [*Erateb, 2019*].

To address the discussed limitations of *ICA* and *IVA*, experimentation by researchers began to take place in the mid-2010s with Convolutional Neural Network (*CNN*) based approaches. A CNN is a type of neural network used for detecting patterns in structured data. These networks are trained using large-scale datasets enable them to efficiently generalize relevant features for segmentation. *U-Net* is a CNN based architecture that was initially developed for image segmentation but was found to yield successful results with stem-separation when *U-Net* architectures were adapted for spectrogram inputs [*Jansson, et al., 2017*]. Spectrograms present the signal data for audio in a quantifiable manner by converting a time-domain waveform into a time-frequency representation that informs models of the pitch and harmonic structure [*Wyse, 2017*]. Using *U-Net*, it was found that separation can be performed first by compressing and encoding the original audio spectrogram data into abstract representations of the contents. The encoded data is then decoded into separate

stems based on this abstracted data; during this, symmetric skip connections link the encoder and decoder layers by passing over the rest of the data that wasn't abstracted in order for the decoder to reconstruct the audio at a high-quality [*Cohen-Hadria, Roebel, Peeters, 2019*]. Although the use of a spectrogram provides a number of benefits such as an abundance of timbre representation, reconstructing a song using a spectrogram can result in phase inconsistencies due to the phase being discarded in favour of time and frequency; additionally, the time-frequency trade off with processing audio in the frequency-domain exclusively often leads to the sound of drums or percussive instruments being attenuated [*Simpson, 2015*]. Spectrogram-based separation models, such as *Spleeter,* have seen continuous support as a less computationally-expensive option with web applications such as *Moises.ai* implementing their own proprietary spectrogram-based models [*Hennequin, et al., 2020*].

Around the late-2010s, arguably the biggest shift in stem-separation occurred with the emergence of time-domain w*aveform-based* separation models. Waveform separation retains the use of *U-Net* architecture but uses a waveform of the song as input in place of a spectrogram. Having access to the unaltered audio signal that comes with a waveform provides the model with a substantial amount of temporal data that otherwise would've been lost during the spectrogram transformation [*Stöter, et al., 2019*]. Moreover, the use of a waveform supplies the model with all necessary phase information, eliminating the need for phase reconstruction and drastically reducing the chance of phase inconsistencies occurring in separated audio [*Stoller, Ewert, Dixon, 2018*].

One of the most powerful waveform-based separation models available today is *Demucs,* an open source deep neural network developed in 2019 for end-to-end source separation. Distinguishing it from other existing waveform models, *Demucs* consists of a hybrid-architecture that takes both waveforms and spectrograms as input [*Défossez, et al., 2019*]. The model's ability to focus on both time-domain and frequency-domain means that temporal data and timbral patterns can both be recognised which combines the advantages of both waveform-based and spectrogram-based approaches [*Défossez, et al., 2021*].

Historically, isolating guitar from other instruments posed a significant challenge for developers; due to there being a wide variety of sounds a guitar can emit, for example electric and acoustic, it can be difficult for models to identify the sound [*Ferrante, 2025*]. Compounding this, considerable spectral overlap can occur between the guitar and vocals or piano. *Demucs'* hybrid-architecture has since made way for the introduction of specialized models such as *htdemucs_6s* that can split the guitar-part of an audio file into its own stem. *Demucs* models prior to *htdemucs_6s* placed the separated guitar audio in a stem labelled *'other'* that consisted of audio left over after vocals, bass and drums had been isolated. Alongside publicly available large-datasets, the *htdemucs_6s* model is specially trained on a private *Meta* dataset that features guitar-rich content to support the model's ability to identify and isolate a guitar-part, this is not the case for other *Demucs* models which are usually trained solely on public datasets such as *MUSDB18* [*Rouard, Massa, Défossez, 2023*].

While existing stem separation applications like *Moises.ai* present the user with isolated guitar, their primary purpose is this one task. To date, no leading applications in this field offer any meaningful assistance on learning how to play the separated audio. This means that, for many guitarists, the use of an app like *Moises.ai* is only useful for one stage of the learning process with technique, chords and fingering guidance advice being largely absent.

3.2 Large Language Models

The rise of *Large Language Models (LLMS)* in the early-2020s offers a powerful solution to this issue of incomplete user-experience. *LLMs* are already being used for *MIR* with *Text-to-Music Retrieval++ (TTMR++)* shown to be effective in giving musical advice from its pre-trained knowledge base based just off of metadata from the audio file [*Doh, et al., 2024*]. With *TTMR++*, the need for contextual information in existing stem-separation applications could be addressed with stem-separation and playing advice both informing the user within a single application, making for a more streamlined user-experience.

Contemporary advancements in this field have seen the development of *multimodal* LLMs which can process multiple different types of data concurrently, an example of this is Google's *Gemini*. Both audio and text prompts can be passed to these models; as a result, more accurate advice can be given from this cross-referencing of different sources [*Xu, et al., 2025*]. Audio analysis functionality has facilitated the use of multimodal LLMs for *automatic music transcription (AMT)* which benefits from also having a contextual understanding of the song it is processing [*Mao, et al., 2025*]. Although multimodal LLMs have delivered promising recent results, it must be noted that that they have a tendency to 'hallucinate' and generate inaccurate information, an exacerbating factor in this is the high volume of data within the training-set leading to spurious associations [*Vasilakis, et al., 2024*]. An LLMs tendency to hallucinate can also be exacerbated when there isn't enough data in its knowledge-base to answer a query; this means that for songs that are less popular, the risk of inaccuracies is higher [*Gumaan, 2025*].

3.3 Information Assimilation

Despite the broad accessibility of these technologies, the fact that they typically aren't synthesized into a single application leads to a fragmented workflow for guitarists looking to utilize these systems as they will have to constantly keep context-switching between applications. Repeated context-switching is shown to impair an individual's ability to take in new information by disrupting cognitive flow [*Parekh, 2024*].

In addition to context-switching, research consistently shows that an excessive amount of information being presented to an individual at once can impair their ability to take in new information, this is known as *cognitive load theory* [*Plaas, Moreno, Brünken, 2010*]. For this reason, a fragmented workflow is entirely unconducive to learning a new song as a guitarist. Cognitive load theory can be especially apparent in music where trying to identify what is being played in a particularly fast song is often a significant challenge; research has shown that by slowing down the playback of a song, it can be easier to understand the music behind it [*Allingham, Wöllner, 2022*].

Developing a user-experience that supports simplicity and intuitiveness has shown to increase a user's capacity to learn with many users in this study stating their lowered barriers to engagement after experiencing a straightforward user-interaction [*Chu, et al, 2019*]. If a user interface doesn't provide the user with a good experience then the value of novel features is diminished.

3.4 Practice Tools

When a guitarist is looking to learn the chords for a song, a form of music notation called a *'chord-sheet'* can be useful. Chord-sheets show the chords of a given song in alignment with the lyrics to clearly represent the timing of chord changes [*Kukreti, 2015*]. Many websites and applications exist as practice environments by providing user-uploaded chord sheets along, some examples include *Ultimate Guitar* and *Songsterr* [*Avila, et al., 2019*]. Applications such as this face the opposite limitations to that of stem-separation programs whereby a a plethora of chords and playing advice is delivered but no functionality exists for isolating the guitar part and listening to it.

Design

SongAssist was designed to directly address the problem statement by providing a unified and focused practice environment for guitarists learning a song using a combination of both contextual playing advice and isolated audio stems. This section delivers a comprehensive look at the distinct design elements contained within the *SongAssist* application.

4.1 Architecture

SongAssist adopts a client-server architecture in which the back-end features are separated from the front-end, this is illustrated in *figure 2*. The primary reason for using this architecture was to ensure computationally expensive tasks such as *Demucs* processing and, to a lesser extent, *Gemini* API calls. With the frontend freed up from such tasks, the user-interface should be significantly more responsive and far less prone to crashes, leading to a satisfying user-experience. Furthermore, this architecture enables potential future deployment of backend processes to cloud infrastructure separate from the user's device which ensures that performance stays consistent across all devices with access to a browser.

The back-end of *SongAssist* was designed with all core features decoupled from each other. This makes *SongAssist* more robust by mitigating the risk that failure of one component will lead to the whole application crashing; feature decoupling also makes the system more scalable with each feature potentially being deployed separately in a distributed infrastructure.

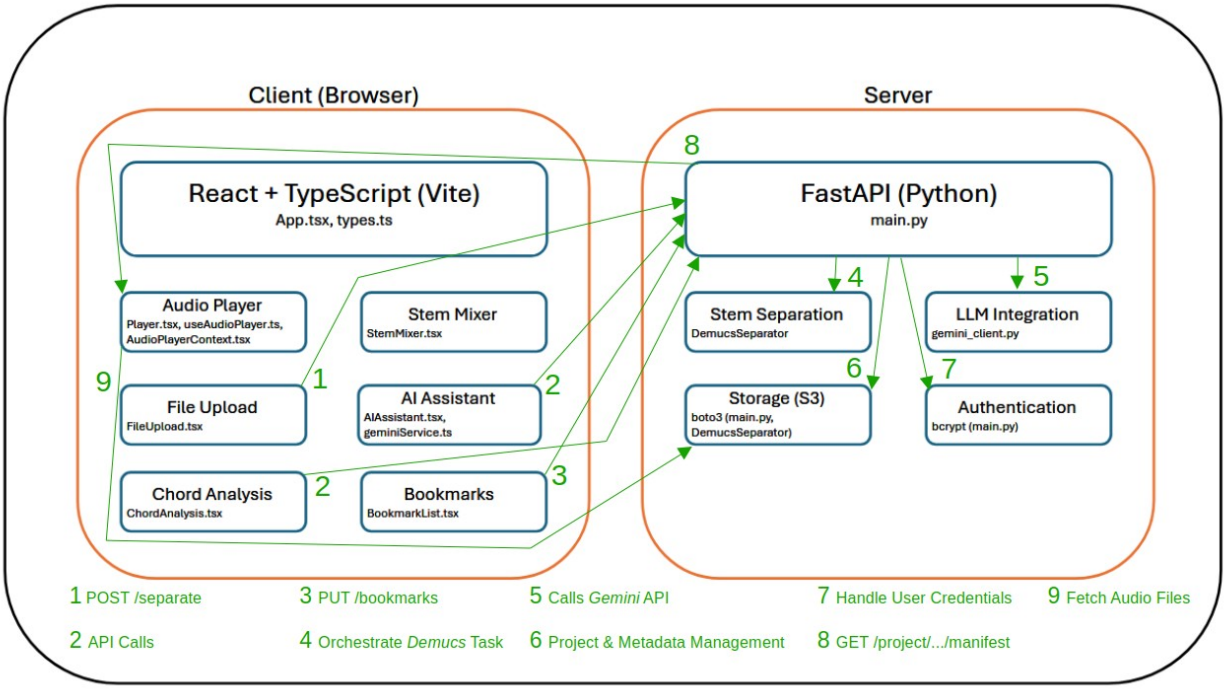

Figure 2: System architecture diagram for SongAssist

4.2 API

SongAssist's API, built using *FastAPI,* acts as a control layer for the entire application. It handles all core functionality such as *user authentication, audio upload* and *separation, Gemini interactions, bookmark storage* and *project management*. Aside from request-response handling, the API also generates and maintains manifest files for each project which link together important metadata associated with each project including bookmarks and generated chords.

Google's *Gemini API* is used to directly address the gap in knowledge after stems have been separated. After receiving song metadata, Gemini can give technique and tone replication advice alongside chord generation. This LLM has been chosen to its multimodal capabilities that have been proven to yield promising results for musical transcription, *Gemini* is also constantly updated by Google which means *SongAssist* can benefit from these improved iterations.

Web Audio API is utilized within the browser to manage all lightweight playback related features. This is managed within the browser in order to ensure maximum responsiveness; if the user experiences any latency when using playback features then this could interrupt workflow and lead to a frustrating user experience. By leaving less resource-intensive tasks like this to the browser, the *SongAssist* API is freed up to centre on orchestration and persistence.

4.3 Storage

AWS S3 Cloud Storage is used as *SongAssist's* storage medium for separated stems as well as metadata and temporary uploads. *S3* was chosen due to its persistent and durable object storage capabilities which ensures reliable access to project assets and extensive scalability, this is vital for an application that is designed to store multiple audio files per user. Each user has their own folder within the S3 bucket in which all their projects are stored along with associated metadata, this prevents user data clashing with other users while keeping retrieval simple.

4.4 Front-End

SongAssist's front-end consists of a *single page application (SPA)* developed using the *React* JavaScript library and TypeScript. React was primarily chosen due to its extensive ecosystem of developer tools and component-based architecture in which each UI element can be separated into its own component. Such a modular approach allows for a separation of concerns between each section of the user-interface which supports maintainability and makes each component simpler to refactor without affecting all other UI elements.

SongAssist is served locally by *Vite*, a JavaScript build tool used for rapid development that provides instantaneous server startup times along with *hot module replacement (HMR).* The developer-friendly features that *Vite* promotes an efficient development process. Once deployed in a production environment, *Vite* is also a strong choice for its lightweight output which, in combination with *Content Delivery Networks (CDN)* compatibility, provides *SongAssist's* front-end with a scalable and production-ready environment.

Naturally, all user interactions with *SongAssist* are initiated from the front-end which includes functionality like audio file uploads, bookmark creation, playback features and adjusting the stem mixers. Whenever an interaction occurs, the front-end makes an *asynchronous API call* to the back-end where resource-intensive tasks such as stem-separation or chord generation are processed. This makes for a computationally efficient front-end that is responsive and isn't overburdened with heavy tasks.

The design of *SongAssist's* user-interface is crucial for addressing the core aim of supplying a simple, easy-to-use practice environment where guitarists can focus purely on the song they are

learning. Consistent design patterns are employed through *TailwindCSS* across *SongAssist* to ensure the application is intuitive even for users unfamiliar with computers. Once a project is loaded in, all functionality is contained within a single page which aligns with *SongAssist's* priority of facilitating a focused and uninterrupted practice session. Functionality for the AI assistant, chord generator and saved bookmark access are divided into 3 tabs within this page and can be easily cycled between while maintaining state-persistence. Intuitive controls for setting specified loops within the stem player together with buttons for instantly switching to an exclusively guitar or backing track mix make for a fulfilling user-experience. The front-end is also responsible for changing the playback speed of audio by utilizing the *HTML5 <audio>* element through the powerful *Web Audio API.*

The choice to develop a web-based front-end was based ion making *SongAssist* platform-independent. With only a browser being required for access to the application, *SongAssist* caters to a wide range of guitarists and lowers barriers to entry.

4.5 Back-End

The back-end of *SongAssist* acts as the central processing layer for the whole application where all computationally expensive processes are handled. As mentioned previously, *FastAPI*, a Python web framework, serves as the foundation of the backend; *FastAPI* was chosen due to its native asynchronous execution support which allows *SongAssist* to handle concurrent user requests efficiently. This ensures that if one user tries to perform an action such as chord generation while another is separating stems then the API can handle both.

The *SongAssist* API delivers a structured set of endpoints for tasks like user management, project handling, stem-separation and chord analysis. User authentication is managed through the API by implementing bcrypt hashing which ensures user details are stored securely; this user data is stored in a JSON file in the corresponding user folder within *AWS S3.* Manifest files to consolidate all metadata and URLs associated with a project are also generated by the API, such files function as a single source of truth for the frontend to query.

Stem-splitting functionality is implemented using *htdemucs_6s* which, as mentioned earlier, is the first and only widely available *Demucs* model that is able to isolate the audio of guitar into its own stem. Once a user has uploaded a file to be separated, a background task is started whereby the temporary file is written to the local disk and uploaded to the *S3* bucket; from here the backend can receive the file. Upon downloading the uploaded audio, the backend first runs a check to see if the duration of the audio exceeds 7 minutes, if this is true then stem-separation process splits into segments in order to avoid memory issues caused by these larger files. The *htdemucs_6s* model then splits the downloaded audio into 2 stems, one containing just guitar and the the other containing all other instrumentation and vocals, this 2 stem approach was decided upon to cover both use cases of listening along to an isolated guitar track and playing along to the backing track in place of the original guitar track. If *SongAssist* was to provide more stems then there is concern this could contribute to excess cognitive load and result in a less focused user experience. As soon as separation has completed and separated stem files have been uploaded, a *manifest.json* file is uploaded into the project folder which contains the location of both stems for the front-end to retrieve project data consistently. Local temporary files are cleaned up following completion of this process to optimize server resource utilization.

Alongside stem-separation, the backend also implements a service orchestration layer with the *Gemini API* that provides playing advice and chord generation for *SongAssist*. Custom prompt engineering is given for each function that *Gemini* is used for such as chord generation and artist assumptions, *Gemini* is specifically prompted to return in a structured JSON format in order to ensure predictable and machine-readable results which can be effectively parsed by the front-end

without too much difficulty. The *Gemini* instance used for analysing a guitar stem is also passed 90 seconds of the separated guitar stem by the back-end in order to assist with harmonic and textural analysis, the limitation of 90 seconds was imposed to reduce the risk of maximum token errors from the model that can result from larger audio files. Information about this audio is transcribed in a readable format and uploaded to the chord generation *Gemini* instance and is used in combination with its existing knowledge base to result in fully informed chord generation for each song. Chords that have either been generated by *Gemini* or edited by the user are saved to their own analysis file in markdown format within the project folder, these objects are stored separately in order to ensure flexibility. *Gemini* also predicts the name of the artist associated with an uploaded song if this is not already provided, this artist name is passed to other *Gemini* endpoints for more relevant advice and chord generation; if the predicted name is incorrect then the user has the option to edit this field, mitigating the risk of AI hallucinations affecting user-experience.

The back-end manages the creation and retrieval of bookmarked loops which are stored in an array within the project's *bookmarks.json* file. This is implemented through a *FastAPI* endpoint that takes a user-created bookmark as a structured request, bookmark objects consist of an identifier, start point, end point and an optional label that can be input by the user. Whenever a project is loaded, the JSON file is retrieved and saved bookmarks are reconstructed by the array to ensure persistence.

4.6 Methodology

Development of *SongAssist* largely followed a waterfall methodology. At the start of the project, key features were identified as *stem-separation, AI-assisted contextual* advice and *comprehensive playback features.* These features were all implemented as planned with some minor exceptions being in the choice of technologies used such as the shift from firebase to AWS S3.

4.7 Technology Stack

- *Front-end*
  - ***TypeScript***
    - The main programming-language used for *SongAssist's* front-end, *TypeScript* provides *static typing* along with *type-annotations*. Use of this language should ensure enhanced maintainability [*Microsoft, 2025*].
  - ***Vite***
    - A lightweight build tool that also delivers a development server for *SongAssist* to be hosted locally. Rapid hot module replacement ensures an efficient development workflow [*Vite Contributors, 2025*].
  - ***React***
    - A *JavaScript* Library for building dynamic user-interfaces. Used for handling the logic behind *SongAssist's* front-end as a single page application with modular components [*Facebook, Inc., 2013*].
  - ***TailwindCSS***
    - An open-source CSS framework integrated with the *Vite* build process. *TailwindCSS* is used for designing *SongAssist's* visual presentation due to the variety of pre-defined styling classes that it offers, assuring *SongAssist* features a consistent design-language to support a pleasant user-experience [*Tailwind Labs, 2025*].
- *Back-end*
  - ***Python (3.12)***
    - Serves as the primary programming language for *SongAssist's* back-end. All server-side logic is implemented using Python due to the large selection of *machine-learning (ML)* and *audio processing* libraries it offers. Compatibility with *Demucs* and *FastAPI* was also a significant factor in choosing Python [*Python Software Foundation, 2024*].
  - ***FastAPI***
    - A python web framework that implements fast asynchronous request handling and robust data validation. Such functionality allows the *SongAssist* API to process multiple requests simultaneously and reduces the risk of runtime errors [Tiangolo, 2018].
  - ***Demucs (htdemucs_6s)***
    - A CNN-Transformer hybrid model tasked with stem-separation functionality within *SongAssist*. The model processes an uploaded file by separating it into two stems, one containing isolated guitar and another containing the rest of the track.
    - [*Rouard, Massa, Défossez, 2023*]
  - ***Google Gemini API***
    - Acts as the API gateway layer for all LLM implementation throughout *SongAssist*. The API provides tools for identifying song data using an uploaded file name, using multimodal audio analysis on an isolated guitar stem, generating chords and playing advice [*Google, 2025*].
  - ***AWS S3***
    - S3 provides persistent cloud storage to store project and user data. *SongAssist's* API interfaces with S3 by saving all relevant data here and for the front-end to access.
  - ***Boto3***
    - Operates as a translation layer between *S3* and *FastAPI* by converting Python operations into low-level API calls for *S3* to process and vice-versa[*Boto3 Developers, 2025].*
  - ***Passlib (bcrypt)***
    - A password hashing library that ensures user passwords entered into *SongAssist* are hashed before they are stored, this ensures robust user authentication [*Passlib Developers, 2025*].

- ***FFmpeg***
  - A multimedia framework that offers a vast selection of libraries for processing audio. In *SongAssist*, *FF*mpeg is used to implement audio-trimming for *Gemini* to access and an audio length check that informs how stem-separation is processed [*FFmpeg Developers, 2025*].

- ***Pydantic***
  - Used to define the models of data that the *SongAssist* API will accept, *Pydantic* is a data validation library for Python. Use of this library ensures that user and project data flowing between the front and back-end of *SongAssist* is correctly structured [*Pydantic Developers, 2025*].

Implementation

Implementation of *SongAssist's* design was focused on 3 key technical contributions: *the stem-separation pipeline, integration of a LLM across the system* and *developing intuitive and responsive front-end playback features*. Developing these areas of *SongAssist* required the creation of original, purpose-built code for ensuring the core features within the application integrate cohesively. To ensure that all components of *SongAssist* integrate together seamlessly and predictably, the data shape of all playback or project objects is defined in a '*types.ts*' file within the front-end. *Figure 3* presents an excerpt from this file, in this the data shape of entities such as stems are constrained to be either '*guitar'* or '*backingTrack*'.

The following 3 sections will demonstrate areas to which a key technical contribution was made, selected to show the technical challenges that were overcome.

```
export type Stem = 'guitar' | 'backingTrack';
export const ALL_STEMS: Stem[] = ['guitar', 'backingTrack'];

export interface Song {
  name: string;
  artist: string;
  duration: number;
  artistConfirmed: boolean;
  stemUrls?: Record<string, string>;
}

export interface Bookmark {
  id: number;
  start: number;
  end: number;
  label: string;
}
```

Figure 3: Excerpt from types.ts file

5.1 Technical Contribution 1: Stem-Separation Pipeline

The stem-separation pipeline is *SongAssist's* single most important feature, allowing for the user to upload any audio file that is then split into two stems with the guitar isolated from the second stem containing a backing track, this process is demonstrated in *figure 4*. While the stem-separation model responsible for isolating guitar is provided by *Demucs*, integrating this separation process into a multi-user platform with cloud storage proved to be a substantial software engineering operation.

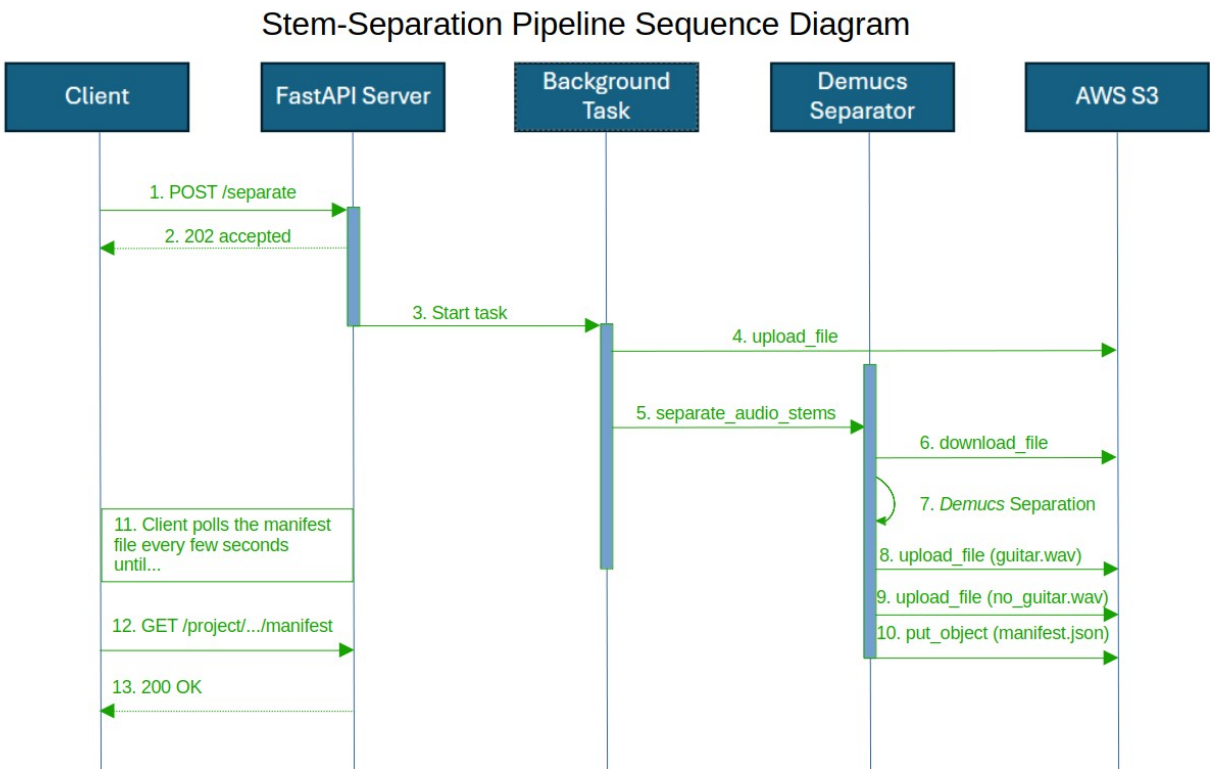

Figure 4: Stem-separation sequence diagram

Given that *htdemucs_6s* is run as a command line tool, the back-end required a tailored implementation capable of orchestrating subprocesses that run *Demucs* as if it was being used in a terminal; this was achieved by constructing command-line arguments in the *DemucsSeparator* class such as *'["python", "-m", "demucs.separate", "-n", self.model, "--two-stems", "guitar"]'* and executing them using *'subprocess.run(..., check=True)'*. Upon file upload, *FFprobe* inspects the length of the track using this command *'["ffprobe", "-v", "error", "-show_entries", "format=duration", "-of", "default=noprint_wrappers=1:nokey=1", file_path]'*. If the track exceeds 7 minutes in length then these flags are appended to the *Demucs* command *'["--segment", "7", "--mp3"]'*, this initiates a segmented separation process that outputs separated stems as MP3 files, this is to prevent memory overflows and performance crashes that were encountered when processing longer songs. For audio files not at risk of causing memory overflow, stem-separation is processed in a single pass without segmentation which is faster, separated stems are also outputted as lossless WAV files as this retains quality and saves compression time.

Because *Demucs* requires disk based input, it was deemed necessary to locally write uploaded audio to the backend server temporarily. This ensures that for each separation process, a stable local file path is always available to operate on. Access keys for *AWS S3* along with *Gemini* and *Vite* are stored in a separate *'.env'* file in order to maintain security.

Once the stem-separation process has successfully concluded, both the *guitar.wav* and *no_guitar.wav* files are uploaded to S3. UUIDs are used to generate task identifiers which prevents

duplicate data collisions. Each project created is stored in a path with the location *'stems/{username}/{task_id}'*, this is completed using *Boto3* with added metadata headers that ensure the content is recognised by S3. Figure 5 shows an excerpt from the *DemucsSeparator* class that demonstrates how stems are uploaded to S3:

```
# Upload guitar and no_guitar stems
for stem_name in ["guitar", "no_guitar"]:
    local_file_path = local_stems_dir / f"{stem_name}.{output_extension}"
    if local_file_path.exists():
        stem_key = f"stems/{username}/{task_id}/{stem_name}.{output_extension}"
        self.s3_client.upload_file(str(local_file_path), bucket_name, stem_key, ExtraArgs={'ACL': 'public-read', 'ContentType': content_type})
        stem_urls[stem_name] = f"{base_url}/{stem_key}"

if "no_guitar" in stem_urls:
    stem_urls["backingTrack"] = stem_urls.pop("no_guitar")
```

Figure 5: Code excerpt defining upload logic

The next stage in the pipeline after stems have been uploaded is creation of a *manifest.json* file that is stored in this same project folder. This manifest file will act as a central authority for the front-end by providing URLs to all relevant project assets such as bookmarks and chord sheets. Figure 6 shows how the manifest file creation process is defined in the *DemucsSeparator* class:

```
manifest_content = {"stems": stem_urls, "originalFileName": original_filename}
manifest_key = f"stems/{username}/{task_id}/manifest.json"
self.s3_client.put_object(Bucket=bucket_name, Key=manifest_key,
Body=json.dumps(manifest_content), ContentType='application/json', ACL='public-read')
```

Figure 6: Code excerpt that defines manifest file creation

It is necessary to clean up temporary audio files created for *Demucs* to process once they are no longer needed. Both the input file and intermediate output directory are deleted to conserve resources and prevent disk saturation.

Following completion of this process, the pipeline can support the next stage which is *Gemini* audio analysis. Upon the user requesting chord generation in the front-end, the isolated guitar stem is retrieved by the back-end using the *manifest.json* file where it is truncated down to 90 seconds. The truncated audio file is then passed to Gemini for multimodal analysis using the *analyze_guitar_stem* for contextual analysis.

5.2 Technical Contribution 2: LLM Integration

Implementation of the multimodal *Gemini Large Language Model* is primarily intended to directly extend the value of stem-separation by providing contextual playing advice to build on insight gained from the isolated audio. *SongAssist* effectively uses the *Gemini* API to provide this functionality. By integrating these features, the application reshapes from a playback tool to a comprehensive learning environment.

Instead of prompting *Gemini* to perform these functions in a single prompt, multiple separate endpoints are defined in the back-end. For each endpoint, it is then possible to specifically prompt-engineer the *Gemini* instance to deliver the required service, ensuring structured and reliable output.

When a user requests automatic chord generation from *Gemini, SongAssist* exposes a back-end route in *'/gemini/analyze-stem'* that initiates and handles the LLM pipeline. As briefly covered at the end of the stem-separation pipeline, the route first retrieves the isolated *guitar.wav* file and uses FFprobe to trim it down to 90 seconds and delegates to the *Gemini* wrapper *'analyze_guitar_file(..)'*. This wrapper defines a multimodal prompt that consists of both the 90 second audio clip and a text-prompt requesting use of knowledge base with the isolated stem audio

to return a structured JSON response that can be parsed properly in the front-end. An excerpt of the *'analyze_guitar_file(..)'* wrapper method is shown in *Figure 7*. *Figure 8* shows a screenshot of generated chords for the song 'Wonderwall' by *Gemini*.

```
uploaded = genai.upload_file(local_audio_path)

parts = []
parts.append({"text": PROMPT_BASE})
if user_prompt:
    parts.append({"text": f"User request: {user_prompt}"})
if extra_context_json:
    context_text = json.dumps(extra_context_json)
    parts.append({"text": f"Supplemental JSON Data:\n{context_text}"})

parts.append(uploaded)

config = GenerationConfig(max_output_tokens=8192)
```

Figure 7: Excerpt from 'analyze_guitar_file(..)'

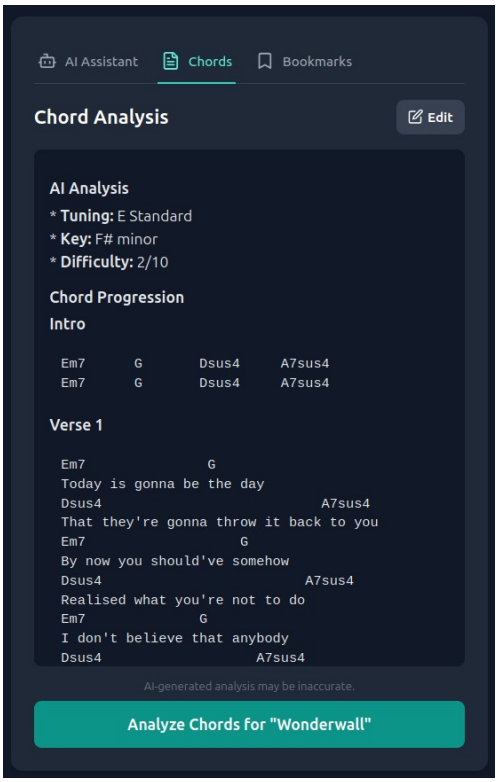

Figure 8: Screenshot of generated chords

Within the *'gemini_client.py'* file, structured prompt engineering is employed to ensure that chord generation always is always presented in a consistent, structured format to be easily understood by the user. The *'PROMPT_BASE'* lays out rules in the prompt such as making sure chord lines and lyric lines alternate, prompting also asks for a difficulty rating to be provided and for inferred contextual information on the passed audio file to be used during chord generation. Supplying *Gemini* with such strict prompts concerning formatting and musical information retrieval is aimed to reduce the risk of hallucinations by the model.

Additional minor features are also implemented using Gemini such as taking the name of the uploaded file, cleaning it up by removing any numbering and inferring the artist of the song from this title. Figure 9 shows the implementation of this feature with *Gemini* being passed the filename and being prompted to store the output of this in a specific JSON format that the *geminiService.ts* front-end code expects.

```python
@app.post("/gemini/identify-from-filename")
def identify_song(req_body: IdentifyRequest):
    system_prompt = """You are a music expert. Your task is to identify a song title and artist from a raw audio filename.
    The filename might contain track numbers, garbage text, or underscores. Clean it up and provide the most likely song title and artist.
    Respond ONLY with a JSON object in the format: {"songTitle": "...", "artist": "..."}.
    If you cannot determine the artist, use "Unknown Artist".
    """
    user_prompt = f"Filename: \"{req_body.rawFileName}\""
    response_data = generate_text_from_prompt(system_prompt, user_prompt, model_name="gemini-2.5-flash")
    try:
        json_text = response_data.get("text", "{}")
        if "```json" in json_text:
            json_text = json_text.split("```json")[1].split("```")[0]
        parsed_json = json.loads(json_text)
        return parsed_json
    except (json.JSONDecodeError, IndexError):
        return {"songTitle": req_body.rawFileName, "artist": "Unknown Artist"}
```

Figure 9: Code excerpt that shows Gemini file identification implementation

5.3 Technical Contribution 3: Front-End Playback Features

For the purpose of delivering an intuitive yet feature-rich playback experience to all guitarists, it was necessary to implement numerous separate front-end components that extend the functionality of the *HTML5 <audio>* element. To make sure each component accesses the element consistently, it was necessary to create a dedicated React hook that interacts with the web audio API and exposes all main playback functions in the form of *'useAudioPlayer.ts'*.

Preventing individual front-end components from managing their own instance of the *<audio>* element was of significant importance; to solve this, the *'AudioPlayerContext.ts'* file provides a sole point of reference and reduces the risk of desynchronisation between components by providing a single instance for all components to use.

Playback Speed Control

In order for guitarists to properly assimilate the playing in a particularly complex song, it can be pragmatic to slow down playback with the intention of being able to process the song at a deeper level. *SongAssist* provides users with functionality for controlling playback speed granularly to complement stem-separation features, although in the current implementation pitch is also affected by changes in speed. The *'setPlaybackSpeed'* method inside *'useAudioPlayer.ts'* is responsible for playback speed control, as revealed in *Figure 10.*

```
const setPlaybackSpeed = useCallback((speed: number | ((prevSpeed: number) => number)) => {
    const context = audioContextRef.current;
    const newSpeed = typeof speed === 'function' ? speed(playbackStateRef.current.currentSpeed) : speed;

    if (playbackStateRef.current.isPlaying && context) {
        const elapsed = (context.currentTime - playbackStateRef.current.startedAt) * playbackStateRef.current.currentSpeed;
        playbackStateRef.current.pausedAt += elapsed;
        playbackStateRef.current.startedAt = context.currentTime;
    }

    Object.values(sourcesRef.current).forEach(source => {
        if (source) {
            source.playbackRate.value = newSpeed;
        }
    });

    setPlaybackSpeedState(newSpeed);

}, []);
```

Figure 10: setPlaybackSpeed method

This method directly manipulates the *playback* property of the Web Audio API source nodes in order to control playback speed of the song. During development, difficulties were encountered with changes in playback speed causing audio to skip forward in the song, causing a workflow disruption. To fix this, *playbackStateRef* is implemented for holding a reference to the playback state; a useRef was appropriate in this scenario as, unlike useState, they don't cause re-renders in the component when triggered which allows for *setPlaybackSpeed* to access up to date playback data. When playback speed is adjusted, the *'const elapsed = (…)'* line ensures that the position within the song is calculated based on the original playback speed before playbackStateRef is updated, ensuring a smooth transition.

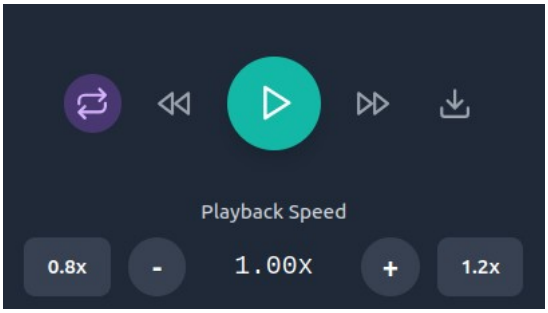

Figure 11: Screenshot of playback speed buttons

Interactive Looping

If a guitarist is particularly focused on a specific section of the song, they can intuitively create their own labelled loops that will activate upon click and save for when the project is next loaded up. Bookmarked loops can be used with the previously mentioned playback speed control functionality to let guitarists repeatedly practice a slowed down, more manageable pace. A main priority and challenge in implementing this feature was developing a simple interface for creating loops that functions as the user would expect it to, the *Player.tsx'* React component handles this functionality. This component defines user interaction with loops so that, when loop mode is activated for the first time in a project, a new default loop is instantly created in the middle of the track for the user to alter. Also when in loop mode, if the user clicks a position on the seeker bar, either the start or end point is dragged to this position depending on whichever has closest proximity to the clicked location. The use effect that implements this is presented in Figure 12.

```
useEffect(() => {
  const handleMouseMove = (e: MouseEvent) => {
    if (!dragStateRef.current?.isDragging) return;
    const timeFromCursor = calculateTimeFromEvent(e);
    const minLoopDuration = 2;

    switch (dragStateRef.current.type) {
      case 'new':
        onLoopChange({ start: Math.min(dragStateRef.current.initialTime, timeFromCursor), end: Math.max(dragStateRef.current.initialTime, timeFromCursor) });
        break;
      case 'start':
        if (loop) {
          const newStart = Math.min(timeFromCursor, loop.end - minLoopDuration);
          onLoopChange({ start: newStart, end: loop.end });
          if (currentTime < newStart) {
            onSeek(newStart);
          }
        }
        break;
      case 'end':
        if (loop) {
          const newEnd = Math.max(timeFromCursor, loop.start + minLoopDuration);
          onLoopChange({ start: loop.start, end: newEnd });
          if (currentTime > newEnd) {
            onSeek(newEnd);
          }
        }
        break;
      case 'seek':
        onSeek(timeFromCursor);
        break;
    }
  };
```

Figure 12: Use effect from Player.tsx that handles loop dragging

While implementing this useEffect hook, handling multiple user intentions at the same time as developing an easy to use seeking/looping interface proved a significant challenge. In initial implementations of the looping interface, the playback head would often fail to update to account for dragged loops and carry on playback outside of the loop. This issue was resolved by using the *onSeek* function to continuously check the *currentTime* value in comparison to both boundaries of the loop; ensuring that when a boundary is moved past the playback head, *onSeek* shifts the playback head to stay within the loop.

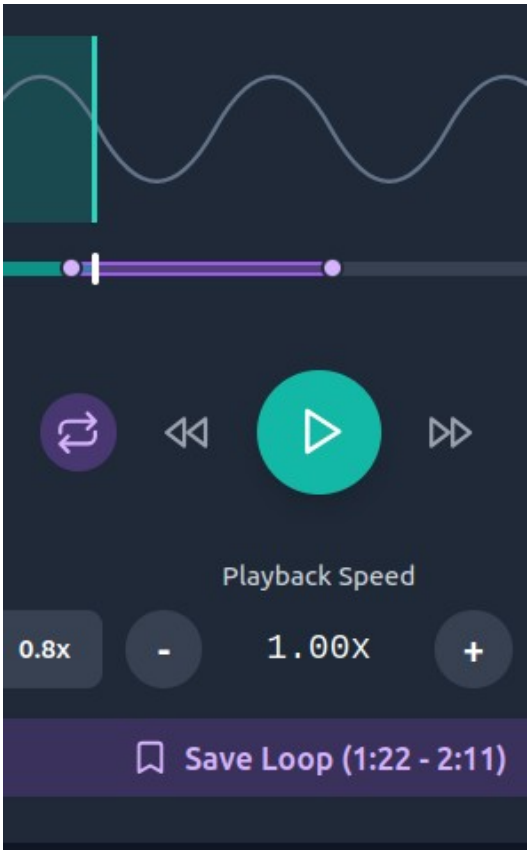

Figure 13: Screenshot of SongAssist showing looping functionality

Stem-Mixer

To handle cases where the user wishes to change the volume of guitar or backing track in the mix without completely muting one of the tracks, an interactive stem-mixer was implemented into *SongAssist*. Shown in *figure 14*, alongside the ability to meticulously control the mix of the guitar and backing track using stem sliders, buttons for instantly muting either of these stems is also present. This feature was implemented in a way that separates the stem volume change logic from the user-facing controls in order to enhance maintainability and robustness of the system. While UI components manage the state of the volume object, volume adjustment logic is handled by the *useAudioPlayer.ts* hook.

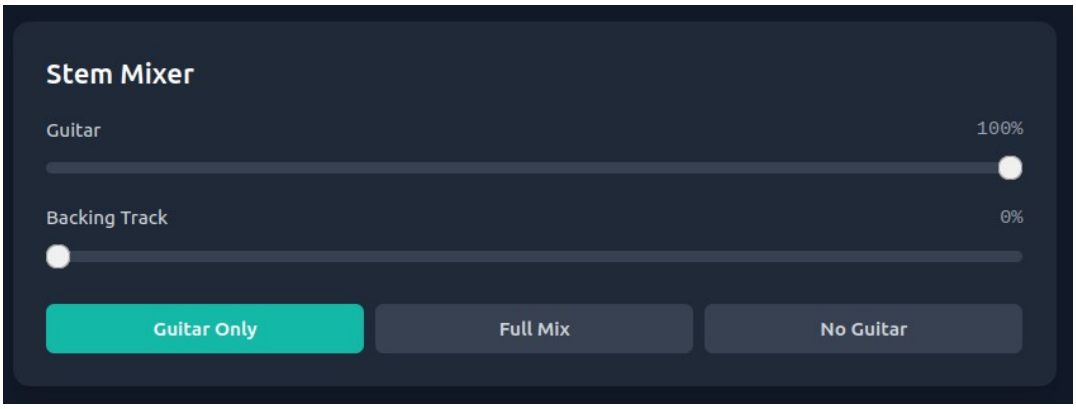

Figure 14: Screenshot of Stem Mixer

The stem-mixer as a visual UI element is rendered by the *StemMixer.tsx* component with an *onVolumeChange* prop that passes any user-input to the *App.tsx* file. Once this user-input is passed, the state of the *stemVolumes* object using a setter method provided by *useAudioPlayer.ts;* similarly, if the *handIsolationChange* function is called when one of the buttons are pressed then this also uses the *setStemVolumes* method, it is shown how this implemented in *Figure 15.*

```
// Update audible stems per selected isolation preset
const handleIsolationChange = (isolation: StemIsolation) => {
    setActiveIsolation(isolation);
    switch (isolation) {
        case 'guitar': player.setStemVolumes({ guitar: 100, backingTrack: 0 }); break;
        case 'backingTrack': player.setStemVolumes({ guitar: 0, backingTrack: 100 }); break;
        case 'full': default: player.setStemVolumes({ guitar: 100, backingTrack: 100 }); break;
    }
};
```

Figure 15: handleIsolationChange method

As for logic tasked with managing stem playback and controls, the *useAudioPlayer.ts* react hook keeps track of separate gain nodes for each stem. The use effect in this file that is responsible for volume control watches for changes to the volume slider in its dependency array. When an update is detected, a Web Audio API method *setTargetAtTime* is called, this method orchestrates a smooth transition between one audio value to another specified by the user; this prevents clicking sounds that can occur when the audio value changes too abruptly while also sounding like an instantaneous shift to the user.

```
// Effect for updating stem volumes
useEffect(() => {
    const context = audioContextRef.current;
    const guitarGain = gainsRef.current.guitar;
    const backingGain = gainsRef.current.backingTrack;

    if (context && context.state === 'running' && guitarGain && backingGain) {
        guitarGain.gain.setTargetAtTime(stemVolumes.guitar / 100, context.currentTime, 0.01);
        backingGain.gain.setTargetAtTime(stemVolumes.backingTrack / 100, context.currentTime, 0.01);
    }
}, [stemVolumes]);
```

Figure 16: Use effect for updating stem volumes

Testing

SongAssist employs both unit and integration testing in order to provide a robust framework for both front and back-end functions. By combining these strategies, valuable insight is gained on both *SongAssist's* technical and user-facing performance. In the current iteration of the system, all tests across the front-end and back-end are passing.

6.1 Framework and Tool Stack

- *Front-end*
  - **Vitest**
    - A powerful testing tool designed for use with the *Vite* build tool. *Vitest* provides APIs for defining and running of unit tests in *SongAssist* [*Vitest Contributors, 2025]*.
  - **React Testing Library (RTL)**
    - A testing library that focuses on testing user-facing components. RTL was used in *SongAssist* to ensure UI elements behave as expected during runtime [*Dodds, 2025*].
  - **jsdom**
    - JavaScript implementation of HTML and DOM standards that can simulate a browser environment. Use of jsdom enabled testing of components without an actual browser [*jsdom Contributors, 2025*].
  - **jest-dom**
    - A testing utility that with DOM-specific matchers. This was used to test UI elements in *SongAssist* with assertions such as .toBeInTheDocument() and .toBeDisabled() [*jest-dom Contributors, 2025*].
- *Back-end*
  - **Pytest**
    - The most powerful Python testing framework; offering simple test setup and mocking. *Pytest* also features better compatibility with FastAPI than other Python frameworks such as *unittest. SongAssist* uses *pytest* as the foundation for testing of the back-end, relying on it to run test functions and manage fixtures [*Pytest Developers, 2025*].
  - **FastAPI Test Client**
    - A testing utility bundled with *FastAPI* that allows for simulation of HTTP requests to an application. This test client is used to send test *GET, POST and PUT* requests to the API in order to validate the request-response cycle.
  - **monkeypatch**
    - A fixture built into Python that can dynamically modify and replace functions throughout a test. Use of this fixture while testing *SongAssist* enabled the *Gemini* API library to be replaced with a mock in order to test client logic without making actual API calls.

6.2 Front-End Testing

While testing *SongAssist's* front-end, it was imperative to make sure that user-facing components are robust and responsive as well as being just functional, this is to align with *SongAssist's* core vision of reducing workflow interruptions. The testing methodology implemented in order to achieve this focuses on the simulation of user-interactions and how the application responds to such interactions. By using the *React Testing Library* it was possible to ensure that front-end tests are consistent and resilient to any code refactoring that occurred throughout development. Every test created for *SongAssist's* front-end was run within a simulated browser environment provided by *jsdom*.

Player
SongAssist's player component is crucial to the application's core functionality. In order to sufficiently test that playback UI elements function as expected, it was necessary to validate that each button and input field changes state appropriately when triggered by a user-interaction. The test suite for this player component includes individual test cases for event handlers pertaining to play, pause, seek and changing metadata.

Figure 17 shows an excerpt from the *Player.test.tsx* suite. As is evident, constants such as *onToggleLoop* are assigned a *vi.fn()* mock function that keeps track of how the respective function is used during testing. When a function like loop toggling is called, the excerpt demonstrates how the component's interactivity during the test by asserting that the mock was successfully invoked. Such a test passing guarantees that the components are functioning as expected.

```
it('invokes primary controls and metadata handlers', () => {
  const onPlayPause = vi.fn()
  const onSeek = vi.fn()
  const onSpeedChange = vi.fn()
  const onAddBookmark = vi.fn()
  const onSongNameChange = vi.fn()
  const onArtistNameChange = vi.fn()
  const onLoopChange = vi.fn()
  const onToggleLoop = vi.fn()

  render(
    <Player
      song={baseSong}
      isPlaying={false}
      currentTime={20}
      playbackSpeed={1}
      loop={null}
      isLooping={false}
      allowLoopCreation={false}
      onPlayPause={onPlayPause}
      onSeek={onSeek}
      onSpeedChange={onSpeedChange}
      onAddBookmark={onAddBookmark}
      onSongNameChange={onSongNameChange}
      onArtistNameChange={onArtistNameChange}
      onLoopChange={onLoopChange}
      onToggleLoop={onToggleLoop}
    />
  )

  // Toggle loop
  fireEvent.click(screen.getByRole('button', { name: /toggle loop/i }))
  expect(onToggleLoop).toHaveBeenCalled()
```

Figure 17: Excerpt from Player.test.tsx suite

AudioPlayerContext State Logic

With the *AudioPlayerContext.ts* file serving as a single-source-of-truth for playback that all components can draw from, it was vital to make sure methods within this context all behave correctly so as to prevent desynchronisation. Toward the goal of testing *AudioPlayerContext.ts* and its internal logic in isolation, it was deemed necessary to mock the *useAudioPlayer.ts* hook that would usually be relied on, this allows for swift debugging by isolating any errors during testing to the AudioPlayerContext.ts file.

Excerpts from *figures 18, 19* and 20 show how *AudioPlayerContext.test.ts* tests in this way. First, the *useAudioPlayer.ts* hook is mocked using *vi.mock* by filling all fields with default parameters, this is shown in figure 18. Next, figure 19 shows a test component *Probe* is rendered to consume the state of the context as well as reveal any actions as a means to ascertain whether the correct data is being passed from *AudioPlayerContext.ts* to the rest of the application. Finally, simulated clicking of the loop toggle button occurs with the test suite checking that correct procedures have initiated as a result of this such as the playback head moving to the start of the loop.

```
// Mock useAudioPlayer to avoid Web Audio API
const seekMock = vi.fn()
vi.mock('../hooks/useAudioPlayer', () => {
  return {
    useAudioPlayer: () => ({
      song: { name: 'S', artist: 'A', duration: 100, artistConfirmed: true },
      isPlaying: false,
      currentTime: 0,
      isLoading: false,
      error: null,
      stemVolumes: { guitar: 100, backingTrack: 100 },
      playbackSpeed: 1,
      load: vi.fn(),
      play: vi.fn(),
      pause: vi.fn(),
      seek: seekMock,
      setStemVolumes: vi.fn(),
      setPlaybackSpeed: vi.fn(),
      setSong: vi.fn(),
    }),
  }
})
```

Figure 18: Excerpt 1 from AudioPlayerContext.test.ts

```
const Probe: React.FC = () => {
  const ctx = useAudioPlayerContext()
  return (
    <div>
      <div data-testid="isLooping">{String(ctx.isLooping)}</div>
      <div data-testid="loopStart">{ctx.loop ? ctx.loop.start.toFixed(2) : 'null'}</div>
      <div data-testid="loopEnd">{ctx.loop ? ctx.loop.end.toFixed(2) : 'null'}</div>
      <button onClick={ctx.onToggleLoop}>toggle</button>
    </div>
  )
}

describe('AudioPlayerContext', () => {
  it('throws if used outside provider', () => {
    const Spy: React.FC = () => {
      expect(() => useAudioPlayerContext()).toThrow()
      return null
    }
    render(<Spy />)
  })
```

Figure 19: Excerpt 2 from AudioPlayerContext.test.ts

```
// Toggle off stores savedLoop and clears active loop
fireEvent.click(screen.getByText('toggle'))
expect(screen.getByTestId('isLooping')).toHaveTextContent('false')
expect(screen.getByTestId('loopStart')).toHaveTextContent('null')

// Toggle on restores savedLoop and seeks
fireEvent.click(screen.getByText('toggle'))
expect(screen.getByTestId('isLooping')).toHaveTextContent('true')
expect(seekMock).toHaveBeenCalled()
)
```

Figure 20: Excerpt 3 from AudioPlayerContext.test.ts

6.3 Back-End Testing

With *SongAssist's* back-end acting as the systemic control tier for all significant data processing tasks such as data persistence and service orchestration, testing should guarantee the reliability of these processes. Back-end test suites for *SongAssist* focuses largely on validating the authenticity of API endpoints by checking that requests and responses are structured correctly. *Pytest* is the foundational framework behind testing of the *SongAssist* back-end and works to decouple application logic from external services with the aim of isolating functionality during testing, much in the same way as the aforementioned front-end tests suites. The appropriate tool for executing this is *monkeypatch* that is included with installation of *Pytest*, this fixture initiates mocks for services such as AWS S3 that behave in an identical way to the real service but don't require any real interaction with S3.

Gemini Client

Due to the fact that *Gemini* can often deliver varied responses for the same query, it can be difficult to develop repeatable tests that validate its output against prompts given in the code. For this reason, *Gemini* is mainly evaluated based on its ability to parse structured JSON data. Unit tests were implemented into the *test_gemini_client.py* test suite in the pursuit of performing this validation without the use of *Gemini API* calls.

Within the *test_gemini_client.py* test suite, a *test_analyze_guitar_file_parses_json* test method verifies whether the *analyze_guitar_file* method logic successfully results in a valid parsed JSON response from *Gemini*. *Monkeypatch* is used to replace the *Google Generative AI library* with a *FakeModel* class that returns a hardcoded JSON string instead of non-deterministic generated output that would be more difficult to reliably test. If the JSON output is returned in the format asserted at the bottom of *figure 21* then the test will pass.

```python
def test_analyze_guitar_file_parses_json(monkeypatch, tmp_path):
    import backend.gemini_client as gc

    # Stub upload_file to return a placeholder
    class FakeUpload:
        pass

    # Fake response object
    class FakeFinish:
        name = "STOP"

    class FakeCandidate:
        def __init__(self, text):
            self.finish_reason = FakeFinish()

    class FakeResp:
        def __init__(self, text):
            self.text = text
            self.candidates = [FakeCandidate(text)]

    class FakeModel:
        def __init__(self, name):
            pass
        def generate_content(self, parts, generation_config=None, safety_settings=None):
            # Return a valid JSON payload as text
            return FakeResp('{"tuning":"E Standard","key":"C"}')

    fake_genai = type("G", (), {
        "GenerativeModel": FakeModel,
        "upload_file": lambda path: FakeUpload(),
    })
    monkeypatch.setattr(gc, "genai", fake_genai)

    # Create a temporary dummy file to "upload"
    p = tmp_path / "a.wav"
    p.write_bytes(b"data")

    out = gc.analyze_guitar_file(str(p), model_name="any")
    assert out.get("tuning") == "E Standard"
    assert out.get("key") == "C"
```

Figure 21: test_analyze_guitar_file_parses test method

User Authentication Flow

Alongside testing individual components of the application, it is also necessary to ensure that all of these separate parts integrate with each other properly. In order to properly validate the user authentication process whereby a user logs in or creates a new account, integration tests focused on the interaction of tasks such as password hashing and S3 storage have been implemented.

In *figure 22*, the *FastAPI* test client simulates a series of HTTP requests made to *SongAssist.* Each HTTP request and response is assessed based on whether back-end logic is successfully executed to produce the intended outcome. In this test, successful registration and login is tested in comparison to test failure cases where a *status_code* of 400 represents a failed registration due to a duplicate account.

```python
def test_register_and_login_flow(client, fake_s3):
    # Register new user
    r = client.post("/register/", json={"username": "alice", "password": "pass123"})
    assert r.status_code == 201
    assert "registered" in r.json()["message"]

    # Duplicate register should fail
    r2 = client.post("/register/", json={"username": "alice", "password": "pass123"})
    assert r2.status_code == 400

    # Login success
    r3 = client.post("/login/", json={"username": "alice", "password": "pass123"})
    assert r3.status_code == 200
    assert r3.json()["username"] == "alice"

    # Wrong password
    r4 = client.post("/login/", json={"username": "alice", "password": "nope"})
    assert r4.status_code == 401

    # Unknown user
    r5 = client.post("/login/", json={"username": "bob", "password": "x"})
    assert r5.status_code == 404
```

Figure 22: test_register_and_login_flow test method

Evaluation

Following development of *SongAssist*, it is important to reflect on the success of the project and how well all initial key requirements were delivered. Comparing *SongAssist* to similar existing MIR applications such as *Moises.ai* can also provide key insight as to the technical achievement of *SongAssist.* Limitations of the application will also be discussed at length in order to identify areas that could potentially be improved in the future.

7.1 Delivery of Requirements

This section will confer how successfully all requirements outlined at the beginning of the report were implemented. As previously mentioned, requirements were divided into 3 categories: *Essential, Desirable* and O*ptional.*

Essential Requirements
All essential requirements were implemented into *SongAssist* successfully. Most notably, the core stem-separation feature was integrated seamlessly into the application using the *htdemucs_6s* model, this specific model was initially identified in the preliminary report as highly suitable for guitar stem-separation which has been legitimised by its strong performance for isolating guitar from other instruments within *SongAssist*. The *Demucs* model performs especially well with 2 independent sources (usually guitar and vocals) but separation quality remains high even with audio of a full band, this mirrors the pattern identified with stem-separation found in the literature review. When requirements were first drafted, it was declared that the user could upload an MP3 file for separation; in the final implementation, this was extended so that any audio file format is compatible with *SongAssist.* The stem-separation pipeline maintained robustness from end-to-end with the upload project files promptly retrieved by the front-end without issue; although stem-separation speed on my laptop was slow, this was due to limited hardware and lack of a compatible, dedicated GPU that can increase source separation speed by up to 300x [*Raj, Povey, Khudanpur, 2022*].

The second essential requirement, configuration of an AI assistant to provide playing advice on the song, was also provisioned with great success. Implementation of this feature utilised *Gemini* API calls to prompt-engineer the assistant in providing optimal advice specifically catered to guitarists, the user can ask any question about playing or technique while listening along to the isolated guitar stem within the same application, this is not possible in other available stem-separation applications and provides deep insight without interruption of workflow or excessive cognitive load caused by context-switching.

While tablature generation was an initial essential requirement, during development it was decided that generated notation should follow the chord sheet format instead of tablature in order to enhance accuracy by letting the AI just list chords instead of fingering positions, this is less demanding for the *Gemini* model. Core functionality of this feature remains the same, it provides guitarists with the musical information they need to play a specific guitar part. *Gemini* is prone to hallucinating the wrong chords for some songs even after such precautions which replicates issues highlighted in the literature review; to combat this, generated chords were made editable and saveable within *SongAssist* so the user can make necessary alterations and keep a chord sheet they are satisfied with saved within each project. When generating chords for less popular songs, *Gemini* usually doesn't have as much information within its knowledge base about the chords or structure so is more likely to hallucinate. Chord generation was orchestrated by prompt-engineering *Gemini* to parse all musical information and chord sheets into a structured JSON output, this was efficacious even for less popular songs and has resulted in consistently structured generated output.

Lastly, the essential requirement for developing an intuitive and user-friendly interface was met triumphantly. Each project file is laid out clearly with each key feature assigned a dedicated component within the user-interface, features such as the AI assistant, chord generator and saved bookmarks each make up separate tabs within a UI element in order to avoid presenting an excess information to the user at once. All user-interactions with the system are responsive, this is attributable to the use of react which effectively manages useState for each object and initiates swift re-renders when state is updated.

Desirable Requirements

Most desirable requirements were delivered with success. An interactive stem-mixer was highlighted as a desirable feature for *SongAssist*, this was fully-implemented and has proved to be an incredibly useful feature for the application. Frequently, it can be useful to turn down one stem in comparison to the other while not completely muting the audio, especially for songs that have been mixed poorly. The stem-mixer supplies functionality for doing this in an intuitive component element featuring two volume sliders for each stem. For users that only want listen to one stem at a time, dedicated buttons are also provided for instantly switching between these stems which covers both use-cases.

Playback speed control with pitch retention was another desirable requirement. While playback speed can can be freely controlled by the user, altering the playback speed changes the pitch accordingly which means the feature was only implemented with some success. Nevertheless, the feature in its current state offers significant value with guitarists still able to slow down a guitar track to gain a greater understanding, the negative consequence is that the guitarist would have to retune their guitar in order to play along with pitch shifted playback.

Bookmarking was a outlined as a desirable feature intended to consist of a timestamp and a label, when the user clicks this bookmark it would take them to the specified timestamp in the song and playback from that point. This feature was implemented early on in the development lifecycle but provided limited usefulness; with bookmarks intended for guitarists who want to hone in on a specific section of a song, the feature in this state required users to repeatedly activate the bookmark to return to the section's starting point which interrupted workflow. To address this, it was decided to add an endpoint to bookmarks where playback would loop back to the start once this point was reached. Bookmarked loops are now a substantially useful feature allowing guitarists to repeatedly listen to a section of the song that they're focused on.

One desirable feature that wasn't implemented fully was playlist management of songs. Due to the fact that users can already navigate through the project list associated with their account, playlist management was deemed unnecessary and could've potentially cluttered the user-interface with a redundant feature.

Optional Requirements

No optional requirements were implemented into *SongAssist*. The waveform display originally proposed as an optional requirement was deemed to not offer that much value; with an increasingly large number of studio recordings using dynamic compression to keep the volume consistent [*Croghan, Arehart, Kates, 2012*], waveforms do not usually represent distinct sections clearly. Resultingly, a more visually appealing UI element that represents a waveform is in place of this feature.

Both the ability to pitch shift without affecting playback speed and to customize the user-interface were optional requirements not implemented. Pitch shifting playback was deemed not useful for practising or understanding a song and a customizable user interface could've overwhelmed users with unnecessary choices.

7.2 Unplanned Features

No user management features were initially planned for *SongAssist* although the need for this became apparent during development. Although user authentication is comparatively basic with only a username and hashed password being associated with each account, this functionality provides the facilities for users to have all songs stored neatly within their project list, providing persistent organisation and separation of user data.

7.3 Testing Outcomes

All unit and integration systems across the front and back-end passed. This shows that the current iteration of *SongAssist* behaves as expected under simulated conditions. Unit tests were especially useful for ensuring playback and looping function correctly while as integration testing provided valuable validation of the stem-separation pipeline. Using a mock for testing *Gemini* integration verified the format of potential output but prevented testing of AI generated accuracy, this could've aided in refining prompt-engineering for the chord analysis model instance.

7.4 Comparison to Existing Applications

SongAssist sits in the middle of 2 different application types: stem-separation services such as *Moises.ai* and guitar-learning platforms such as *Ultimate Guitar. Moises.ai* likely offers superior stem-separation quality to that of the *htdemucs_6s* model used in *SongAssist* but doesn't provide any insight on learning a specific song in the way *SongAssist* does; for guitarists, *SongAssist* is likely the superior choice but most other musicians would probably enjoy the separation quality and section identification that *Moises.ai* offers.

In terms of guitar-learning platforms, the sheer volume of chord-sheets available on applications like *Ultimate Guitar* means that the likelihood of finding accurate chords here is likely higher than *SongAssist's* chord generator providing accurate chords. However, users have to seek this tablature out themselves and decide between multiple different chord sheets that can all contain different voicing, *SongAssist* streamlines this process by presenting the user with generated chords at one click of a button, without the need to even type in a song title. For this reason, it could be argued that *SongAssist* supports a more fluid workflow than applications like *Ultimate Guitar*.

7.5 Limitations

During runtime, the duration of time that the *Demucs* model takes to complete separation is significantly protracted, this is solely due to my laptop proving a bottleneck for the separation process with subpar hardware and no compatible Nvidia GPU that can significantly boost performance [*Mezza, et al., 2024*]. When separating the stems of a 4 minute 22 second audio file, the *Demucs* model took 4 minutes 56 seconds to complete this process which is too slow for proper deployment.

User authentication features are not deployment-ready. Features such as password reset and email verification are not present which could cause security concerns if *SongAssist* were to be widely deployed.

As mentioned previously, *Gemini* hallucinations and inaccuracies when generating chords are still present even after strict prompt-engineering has been employed. The ability to edit generated chords mitigates this issue but user-experience would nevertheless be improved by a reduction in AI hallucinations. Attempts were made to implement *Essentia*, a JavaScript library for music analysis,

to analyse the isolated guitar stem and pass analysis data to Gemini but this could not be integrated within the time frame [*Music Technology Group, 2025*].

7.6 Future Work

There are numerous enhancements that could be made to *SongAssist* with the aim of directly addressing limitations outlined in the previous section. In order to prevent performance bottlenecks during stem-separation occurring in the future, processing of the stem-separation pipeline could be migrated to a cloud-based processing environment such as *AWS EC2* instances equipped with GPUS. This would be a more scalable approach and, more importantly, would lead to stem-separation completing significantly faster which would drastically improve user-experience.

Improving the depth of user management system is also a priority for future iterations of *SongAssist.* Allowing for more comprehensive account management features such as email verification and password reset functionality would markedly increase security and usability of *SongAssist*, making it marketable to a larger number of users and further delivering the core aim of making *SongAssist* accessible to all guitarists.

Use of Generative AI Declaration

Guidance and assistance was provided by the ChatGPT Codex extension for VSCode across the system, particularly in developing boiler-plate code for front-end components such as *Icons.tsx* and testing. All AI generated code was reviewed, adapted and maintained by me as developer of the project to ensure alignment with the original vision of SongAssist was carried out. Architectural and system design elements were orchestrated by me with AI used as a support tool.

Conclusion

The original concept for *SongAssist* was to address the interruptions to workflow that many guitarists face when learning a new song, primarily caused by context-switching between stem-separation and musical notation services. As a completed project, *SongAssist* successfully delivers a comprehensive learning environment that caters specifically to guitarists by isolating the guitar from any uploaded file while also offering *Gemini* AI-assisted playing advice and chord generation, removing the need for context-switching by keeping musical notation and isolated audio within one application. The application consists of a client-server architecture whereby the *Demucs* stem-separation model and other computationally expensive back-end tasks are separated from the front-end in order to maintain a robust system. Modern and responsive playback features are implemented into *SongAssist's* front-end using the *Web Audio API*, allowing for intuitive looping functionality with the option to save user-defined loops and also playback speed control. Overall, the *SongAssist* project has proven a success, demonstrating that the integration of both AI and audio-isolation technologies can succeed in delivering a feature-rich practice tool.

References:

- *Oxenham, A.J., Fligor, B.J., Mason, C.R. and Kidd Jr, G., 2003. Informational masking and musical training. The Journal of the Acoustical Society of America, 114(3), pp.1543-1549.*
- Kharah, D., Parekh, D., Suthar, K. and Shirsath, V., 2021. Audio stems separation using deep learning. *Int. J. Eng. Res. Technol.(IJERT)*, *10*(03).
- Watcharasupat, K.N. and Lerch, A., 2024. A stem-agnostic single-decoder system for music source separation beyond four stems. *arXiv preprint arXiv:2406.18747*.
- Kefalis, C. and Drigas, A., 2019. Web Based and Online Applications in STEM Education. *Int. J. Eng. Pedagog.*, *9*(4), pp.76-85.
- Araki, S., Ito, N., Haeb-Umbach, R., Wichern, G., Wang, Z.Q. and Mitsufuji, Y., 2025, April. 30+ years of source separation research: Achievements and future challenges. In *ICASSP 2025-2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 1-5). IEEE.
- Vásquez, M.A.V., Baelemans, M., Driedger, J., Zuidema, W.H. and Burgoyne, J.A., 2023, November. Quantifying the Ease of Playing Song Chords on the Guitar. In *ISMIR* (pp. 725-732).
- Mark, G., Gonzalez, V.M. and Harris, J., 2005, April. No task left behind? Examining the nature of fragmented work. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 321-330).
- Josel, S.F. and Tsao, M., 2021. *The techniques of guitar playing*. Bärenreiter-Verlag.
- Martelloni, A., McPherson, A.P. and Barthet, M., 2023. Real-time percussive technique recognition and embedding learning for the acoustic guitar. *arXiv preprint arXiv:2307.07426*.
- Wiggins, A.F., 2024. *Leveraging Unlabeled Data to Improve Guitar Tablature Transcription* (Doctoral dissertation, Drexel University).
- Plass, J.L., Moreno, R. and Brünken, R. eds., 2010. Cognitive load theory.
- Chu, A., Biancarelli, D., Drainoni, M.L., Liu, J.H., Schneider, J.I., Sullivan, R. and Sheng, A.Y., 2019. Usability of learning moment: features of an E-learning tool that maximize adoption by students. *Western Journal of Emergency Medicine*, *21*(1), p.78.
- Cano, E., FitzGerald, D., Liutkus, A., Plumbley, M.D. and Stöter, F.R., 2018. Musical source separation: An introduction. *IEEE Signal Processing Magazine*, *36*(1), pp.31-40.
- Rumbold, E.J., 2022. *A Critical Analysis of Objective Evaluation Metrics for Music Source Separation Quality* (Doctoral dissertation, NORTHWESTERN UNIVERSITY).
- Défossez, A., Usunier, N., Bottou, L. and Bach, F., 2019. Music source separation in the waveform domain. *arXiv preprint arXiv:1911.13254*.
- Kadandale, V.S., Montesinos, J.F., Haro, G. and Gómez, E., 2020, September. Multi-channel u-net for music source separation. In *2020 IEEE 22nd international workshop on multimedia signal processing (MMSP)* (pp. 1-6). IEEE.
- Sawada, H., Ono, N., Kameoka, H., Kitamura, D. and Saruwatari, H., 2019. A review of blind source separation methods: two converging routes to ILRMA originating from ICA and NMF. *APSIPA Transactions on Signal and Information Processing*, *8*, p.e12.
- Défossez, A., Usunier, N., Bottou, L. and Bach, F., 2019. Demucs: Deep extractor for music sources with extra unlabeled data remixed. *arXiv preprint arXiv:1909.01174*.
- Cruces, S., 2015. Bounded component analysis of noisy underdetermined and overdetermined mixtures. *IEEE Transactions on Signal Processing*, *63*(9), pp.2279-2294.
- Feng, F. and Kowalski, M., 2018. Revisiting sparse ICA from a synthesis point of view: Blind Source Separation for over and underdetermined mixtures. *Signal Processing*, *152*, pp.165-177.

- Kim, T., Eltoft, T. and Lee, T.W., 2006, March. Independent vector analysis: An extension of ICA to multivariate components. In *International conference on independent component analysis and signal separation* (pp. 165-172). Berlin, Heidelberg: Springer Berlin Heidelberg.
- Hao, J., Lee, I., Lee, T.W. and Sejnowski, T.J., 2010. Independent vector analysis for source separation using a mixture of Gaussians prior. *Neural computation*, *22*(6), pp.1646-1673.
- Sawada, H., Mukai, R., Araki, S. and Makino, S., 2004. A robust and precise method for solving the permutation problem of frequency-domain blind source separation. *IEEE transactions on speech and audio processing*, *12*(5), pp.530-538.
- Ono, N., 2012, December. Auxiliary-function-based independent vector analysis with power of vector-norm type weighting functions. In *Proceedings of The 2012 Asia Pacific Signal and Information Processing Association Annual Summit and Conference* (pp. 1-4). IEEE.
- Erateb, S., 2019. *Enhanced IVA for audio separation in highly reverberant environments* (Doctoral dissertation, Loughborough University).
- Jansson, A., Humphrey, E., Montecchio, N., Bittner, R., Kumar, A. and Weyde, T., 2017. Singing voice separation with deep u-net convolutional networks.
- Cohen-Hadria, A., Roebel, A. and Peeters, G., 2019, September. Improving singing voice separation using deep u-net and wave-u-net with data augmentation. In *2019 27th European signal processing conference (EUSIPCO)* (pp. 1-5). IEEE.
- Simpson, A.J., 2015. Time-frequency trade-offs for audio source separation with binary masks. *arXiv preprint arXiv:1504.07372*.
- Stöter, F.R., Uhlich, S., Liutkus, A. and Mitsufuji, Y., 2019. Open-unmix-a reference implementation for music source separation. *Journal of Open Source Software*, *4*(41), p.1667.
- Stoller, D., Ewert, S. and Dixon, S., 2018. Wave-u-net: A multi-scale neural network for end-to-end audio source separation. *arXiv preprint arXiv:1806.03185*.
- Hennequin, R., Khlif, A., Voituret, F. and Moussallam, M., 2020. Spleeter: a fast and efficient music source separation tool with pre-trained models. *Journal of Open Source Software*, *5*(50), p.2154.
- Wyse, L., 2017. Audio spectrogram representations for processing with convolutional neural networks. *arXiv preprint arXiv:1706.09559*.
- Défossez, A., 2021. Hybrid spectrogram and waveform source separation. *arXiv preprint arXiv:2111.03600*.
- Ferrante, B.M., 2025. Brandon M Ferrante, Improving Real-time Music Source Separation with Knowledge Distillation.
- Rouard, S., Massa, F. and Défossez, A., 2023, June. Hybrid transformers for music source separation. In *ICASSP 2023-2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 1-5). IEEE.
- Doh, S., Lee, M., Jeong, D. and Nam, J., 2024, April. Enriching music descriptions with a finetuned-llm and metadata for text-to-music retrieval. In *ICASSP 2024-2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 826-830). IEEE.
- Xu, W., Zhao, L., Song, H., Song, X., Lu, Z., Liu, Y., Chen, M., Lim, E.G. and Yu, L., 2025, April. Mozualization: Crafting Music and Visual Representation with Multimodal AI. In *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems* (pp. 1-7).
- Mao, Z., Zhao, M., Wu, Q., Wakaki, H. and Mitsufuji, Y., 2025. Deepresonance: Enhancing multimodal music understanding via music-centric multi-way instruction tuning. *arXiv preprint arXiv:2502.12623*.
- Vasilakis, Y., Bittner, R. and Pauwels, J., 2024. Evaluation of pretrained language models on music understanding. *arXiv preprint arXiv:2409.11449*.

- Parekh, R., 2024. The impact of task-switching on executive functions: exploring the effects on cognitive flexibility. *Essex Student Journal*, *15*(1).
- Gumaan, E., 2025. Theoretical Foundations and Mitigation of Hallucination in Large Language Models. *arXiv preprint arXiv:2507.22915*.
- Allingham, E. and Wöllner, C., 2022. Slow practice and tempo-management strategies in instrumental music learning: Investigating prevalence and cognitive functions. *Psychology of music*, *50*(6), pp.1925-1941.
- Raj, D., Povey, D. and Khudanpur, S., 2022. GPU-accelerated guided source separation for meeting transcription. *arXiv preprint arXiv:2212.05271*.
- Croghan, N.B., Arehart, K.H. and Kates, J.M., 2012. Quality and loudness judgments for music subjected to compression limiting. *The Journal of the Acoustical Society of America*, *132*(2), pp.1177-1188.
- Kukreti, M., 2015, July. Affective analysis of musical chords. In *2015 Science and Information Conference (SAI)* (pp. 379-385). IEEE.
- Avila, J.P.M., Hazzard, A., Greenhalgh, C. and Benford, S., 2019, September. Augmenting guitars for performance preparation. In *Proceedings of the 14th International Audio Mostly Conference: A Journey in Sound* (pp. 69-75).
- Mezza, A.I., Giampiccolo, R., Bernardini, A. and Sarti, A., 2024. Toward deep drum source separation. *Pattern Recognition Letters*, *183*, pp.86-91.
- Saini, R. and Behl, R., 2020, January. An Introduction to AWS-EC2 (Elastic Compute Cloud). In *ICRMAT* (pp. 99-102).
- Python Software Foundation (2023) *Python version 3.12*. [Programming language].
- Tiangolo, S. (2018) *FastAPI* [web framework].
- Boto3 Developers (2025) *Boto3: AWS SDK for Python* [library]. Amazon Web Services.
- Google (2025) *Google Generative AI (Gemini API)* [API service]. Google.
- FFmpeg Developers (2025) *FFmpeg* [software library].
- Passlib Developers (2025) *Passlib: Password Hashing Library for Python* [library].
- Pydantic Developers (2025) *Pydantic* [library].
- Pytest Developers (2025) *Pytest* [testing framework].
- Facebook, Inc. (2013) *React* [JavaScript library]. Meta.
- TypeScript Developers (Microsoft) (2025) *TypeScript* [programming language]. Microsoft.
- Vite Contributors (2025) *Vite* [build tool].
- Tailwind Labs (2025) *Tailwind CSS* [CSS framework].
- Vitest Contributors (2025) *Vitest* [testing framework].
- Kent C. Dodds and Contributors (2025) *React Testing Library* [library].
- jsdom Contributors (2025) *jsdom* [JavaScript implementation of the DOM].
- jest-dom Contributors (2025) *jest-dom* [testing utility].
- Essentia Developers (Music Technology Group, Universitat Pompeu Fabra) (2025) *Essentia* [audio analysis library].