

CARDIFF UNIVERSITY
SCHOOL OF COMPUTER SCIENCE AND
INFORMATICS

Development and Evaluation of a Web Application for Creating Click Tracks with Advanced Rhythmic Features

Daniel Redly

SEPTEMBER 16, 2022



Supervisor: Prof David Marshall
Student ID: c2074914

Abstract

Musicians sometimes listen to a series of rhythmic pulses known as a click track while practicing or recording music to make sure they stay in time. Often the music has a constant tempo, meaning it stays at the same speed, but this is not always the case. If the music features sections which speed up or slow down, tempo curves can be used to model the change in tempo over time. Polyrythms are another feature of some music that could be incorporated into a click track. On a basic level, a polyrhythm is when two distinct rhythms are playing at the same time. Polyrythms feature in musical genres from around the world such as Afro-Cuban and Indian Classical music, and can be an effective rhythmic device to make a song sound more interesting.

Creating a click track for a piece of music with constant tempo is simple, as a pulse is simply played with a constant interval, however adding polyrythms and tempo changes presents new challenges in the creation of click tracks. The aim of this project was therefore to develop a web application which makes it easy and convenient for musicians of all levels to create click tracks with advanced rhythmic features.

An initial version of the web application was developed which allowed users to create and edit click tracks in the browser as well as download them in a variety of different formats including MIDI and wav. To place this work into context, a review was done of some existing literature in the field, particularly on gradual tempo change. A selection of existing musical applications, especially for the web, were also examined to see how they implement tempo changes, polyrythms and other interesting rhythmic features.

After completing and deploying an initial version of the application, a user evaluation survey was sent to fellow students and musicians. 100% of the participants were able to generate tempo changes which sounded natural, and ten out of the fifteen participants found the application "fairly easy" to use.

More work was done to improve and expand on the application while user evaluation took place on the initial version. Firstly, user accounts and a database were added so that users could save click tracks and come back to them later. Afterwards, end to end and performance testing were done in an effort to improve the speed and reliability of the application.

Acknowledgements

A special Thank you to my supervisor Prof David Marshall, and to all who participated in my user survey.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 6 |
| 1.1 | Keeping Time in Music | 6 |
| 1.2 | Metronome | 6 |
| 1.3 | Click Tracks | 6 |
| 1.3.1 | What is a click track? | 6 |
| 1.3.2 | Pros and Cons of Click Tracks | 7 |
| 1.4 | Tempo Change | 9 |
| 1.5 | Time signature and polyrhythms | 9 |
| 2 | Background | 11 |
| 2.1 | Academic Research | 11 |
| 2.1.1 | Local Tempo and Expressive Deviations | 11 |
| 2.1.2 | Gradual Tempo Change | 11 |
| 2.1.3 | Polyrhythms | 13 |
| 2.2 | Existing Software | 14 |
| 2.2.1 | Music and Rhythm Web applications | 14 |
| 2.2.2 | Digital Audio Workstations | 15 |
| 2.2.3 | Other applications | 16 |
| 3 | Aims and Objectives | 18 |
| 3.1 | Summary | 18 |
| 3.2 | Functional Requirements | 18 |
| 3.3 | Non Functional Requirements | 21 |
| 4 | Technical Implementation | 23 |
| 4.1 | Overall Structure and Design | 23 |
| 4.2 | Justification for Chosen Tools | 23 |
| 4.3 | Development Environment | 24 |
| 4.4 | Frontend | 24 |
| 4.4.1 | Frontend Structure | 25 |
| 4.4.2 | State Management | 25 |
| 4.4.3 | Audio Playback | 26 |
| 4.4.4 | User Interface | 26 |
| 4.4.5 | Rhythm Calculations | 27 |
| 4.4.6 | Data structure and default values | 32 |
| 4.4.7 | Tempo Visualisation | 33 |

| | | |
|----------|---|-----------|
| 4.4.8 | Help Dialogues | 35 |
| 4.5 | Audio Processing Microservice | 35 |
| 4.5.1 | Structure | 35 |
| 4.5.2 | MIDI file creation | 36 |
| 4.5.3 | Audio file synthesis | 39 |
| 4.5.4 | Audio file formats | 40 |
| 4.5.5 | File Upload | 41 |
| 4.6 | User Management Back-end | 42 |
| 4.6.1 | Structure | 42 |
| 4.6.2 | Authentication | 43 |
| 4.6.3 | TypeScript | 44 |
| 4.6.4 | Changes to Frontend | 44 |
| 4.7 | Deployment | 46 |
| 4.7.1 | Note on the Heroku free tier | 49 |
| 5 | Testing | 50 |
| 5.1 | End to End Testing | 50 |
| 5.2 | Performance analysis | 51 |
| 6 | User Evaluation | 57 |
| 7 | Future Improvements | 65 |
| 7.1 | Audio processing in the browser | 65 |
| 7.2 | Beat Detection | 65 |
| 7.3 | Native Mobile Application | 65 |
| 7.4 | Container Optimisation | 66 |
| 8 | Reflection | 67 |
| | References | 68 |
| 9 | Appendix | 74 |
| 9.1 | GitHub Repositories | 74 |
| 9.2 | User Survey and Related Documents | 74 |

1 Introduction

1.1 Keeping Time in Music

When playing an instrument or singing, it is usually important for a musician to stay in time, that is to adhere to the intended rhythm of a piece, playing with the correct tempo. Keeping time is an important skill, especially when playing with other musicians, and can be learned and practiced by anyone. Musical entrainment is the phenomenon that allows individuals to coordinate with rhythms from their environment, for example tapping their foot along to a drum beat or synchronising with other players in a band [1]. It is defined by McPherson et al [2] as a stable relationship between external periodic signals and an individual's internal rhythmic processes.

The tempo of music is typically measured in beats per minute, commonly abbreviated to bpm. Inter-Onset-Interval, or IOI, is another measurement of tempo, representing the time in seconds between two consecutive beats. Therefore the relation between tempo in bpm and IOI in seconds can be formulated as follows:

$$Tempo_{bpm} = \frac{60}{IOI} \quad (1.1)$$

Both of these terms will be used extensively throughout the project.

1.2 Metronome

Perhaps the simplest way to improve one's rhythmic skills is to practice to a metronome, a device which produces a regular pulse to aid the player with keeping in time. This is usually an auditory pulse, but can often be accompanied by visual cues too, such as a flashing light or swinging pendulum. A metronome is especially useful when playing unaccompanied or in an ensemble without a percussionist. Additionally, musicians often use metronomes for practice to increase the tempo at which they can play a piece, starting with a very slow tempo and gradually working their way up [3].

1.3 Click Tracks

1.3.1 What is a click track?

A click track is essentially a prerecorded audio file which plays the same role as a metronome. The use of a click track is prevalent in recording sessions and live

performance, where musicians, most commonly the drummer, listen to it through in-ear monitors [4]. In the simplest case, the click track is simply a constant pulse, e.g. 120bpm over the entire song. This is suitable for a large portion of commercially recorded music today, in which the tempo remains constant for the duration of a song. For more complex tempo manipulations, however, a Digital Audio Workstation (DAW) can be used. The details for how this works in different DAW's will be explained in Section 2.2.2. For reference, A DAW is an application for the recording, processing, and editing of digital audio, often replacing pieces of hardware found in a recording studio [5]. DAW's, especially in their more recent incarnations, can often be quite complex and all-encompassing in terms of their functionality, according to Reuter [6].

1.3.2 Pros and Cons of Click Tracks

Using a click track, tracks can be recorded independently of one another and mixed together later, with the assurance that they will be in time with each other. This would be especially useful when musicians are collaborating on a song remotely. Sometimes, the drums/percussion can be recorded first, and then other musicians can record their tracks over the drums instead of using a click track, but a click track can be very useful for the drummer to record the drum track in the first place. Additionally click tracks can still be preferable, as they usually use a fairly high pitched and percussive sound that will easily stand out in the mix to a musician, making it easier to play in time. Software-based click tracks, especially when implemented through a modern DAW, also tend to be more flexible than a simple metronome, as they can incorporate changes in time signature and tempo [7].

There are however limitations to the use of click tracks, especially ones with constant tempo and time signature. While they are common in contemporary and especially pop music, there are many musical traditions around the world that do not follow a constant tempo or regular time signature throughout a piece. In the world of classical music, where tempo is typically more flexible, the conductor can be thought of as playing the role of a click track, acting as the cue for all musicians in an orchestra to stay in time. While some pieces may have gradual and fairly predictable changes in tempo, such as *final ritardandi* (the gradual deceleration of a piece towards the end), another important feature of classical music, especially from the Romantic era, is *rubato*, which can be thought of as a push and pull of timing [8]. The former could be modelled fairly accurately in a click track with some mathematically defined tempo track as we will see later in Section 2.1.2, but the latter would present a greater challenge, as tempo fluctuates on a more granular

level.

Many people in the music community can be quite critical of click tracks, with the main criticism being that they tend to stifle individual expression and make music sound too mechanical or robotic. For example, James Beament [9] has a quote from his book *How we hear music: the relationship between music and the hearing mechanism* which illustrates this sentiment quite effectively:

And many recent recordings of pop music demonstrate how music is killed by a metronome for they are as square as a draftsman's T. For the convenience of recording engineers, each player has to record their part on a separate track while listening to a click track — a metronome — and the clicks are then used to synchronize the tracks while the technicians adjust them to their taste and mix them. I know talented young musicians who can't do it; we can understand why. Nothing compares with a recording of a live performance in which the players provide each other with the time-framework. if you want to kill a musical performance, give the player a click track!

Paul Lamere [10], the author of a music technology blog, performed an analysis on tempo deviations of performances from rock drummers to determine which do and do not use click tracks. This was done by using the python library remix to plot beat durations in songs, averaged over a short window. It was quite easy to tell which performances made use of a click track, as the plots were drastically different, as shown in Figures 1.1 and 1.2, his plots for Dizzy Miss Lizzie by the Beatles and One More Time by Britney Spears.

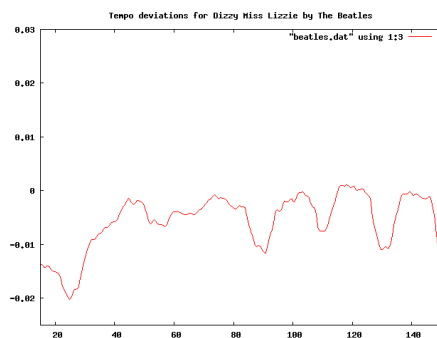


Figure 1.1: Without click track

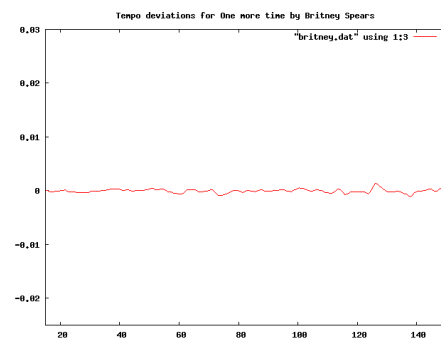


Figure 1.2: With click track

These artists are both appreciated by many people, which suggests that use of a click track is neither something essential for good music or something which ruins it, rather it all depends on the specific application.

1.4 Tempo Change

Quite often, music will speed up or slow down, and therefore a constant isochronous pulse is no longer sufficient to act as a click track. Rather than the local tempo fluctuations discussed in Section 1.3.2, this work will focus more on gradual tempo changes. In traditional musical vocabulary, an increase in tempo is called an *accelerando* or less commonly *stringendo*, while a decrease is called a *ritardando*, or sometimes *rallentando* [11]. Some famous examples of *accelerando* include Edvard Grieg's *In the Hall of the Mountain King* and Led Zeppelin's *Stairway to Heaven*. *Ritardando* is often seen in classical music, especially towards the end of a piece or sections, as in Albeniz's *Tango* or Chopin's *Etude op.10-3* [12].

1.5 Time signature and polyrhythms

A piece of music is typically subdivided into sections called measures which consist of a certain number of beats, most commonly four. If one tries counting along to any contemporary pop song, chances are that counting to four will feel most natural, as it is most likely in 4/4 time. The meaning of this fractional representation will be explained shortly. For any waltz, however, listeners will naturally find themselves counting to three, as there are only three beats per measure. At its simplest, a time signature represents how many beats are played per measure. Sometimes, the time signature can change in the middle of a piece of music, such as in Pink Floyd's song *Money*, where the song starts off with seven beats per measure but then switches to four beats per measure as the guitar solo starts.

We might often hear that 4/4 is the most common time signature in most contemporary Western music, but what does 4/4 actually mean? Formally, time signature is represented as a fraction, where the numerator represents the number of beats in a measure, and the denominator represents the type of beat. Most commonly this is a quarter note, represented by a number 4, but half notes and eighth notes are not too rare either, represented by numbers 2 and 8 respectively. Time signatures such as 3/4 and 4/4 are considered to be simple meters, which means that one pulse corresponds to one count. 6/8, on the other hand, as well as other time signatures counted in eighth notes are generally compound meters, where the beat is subdivided. This means that 6/8 is generally felt as 2 pulses, each subdivided into 3 beats [13]. Some well-known examples of music in compound meter are *We Are the Champions* by Queen in 6/8 [13], and *Claire de Lune* by Claude Debussy in 9/8 [12].

A slightly rarer occurrence, at least in most genres of music, is a polyrhythm, in

which two time signatures are playing at once. More specifically, a polyrhythm can be defined as the simultaneous use of multiple rhythms which are not perceived to be derived from each other. An example of this would be two measures of the same length, where one is divided into 4 beats and the other into 3. This would be called a 3 against 4 polyrhythm [14]. Polyrhythms are especially popular in genres such as progressive rock and math rock, where there is a strong emphasis on technical prowess and rhythmic intricacy. Polyrhythms are not to be confused with polymeters. While the beats in polyrhythms will always align on the downbeat (the first beat of a measure), polymeters instead consist of two parts of different time signature but the same tempo, so the beats themselves will occur simultaneously, but the downbeats will drift out of sync with each other. This difference is illustrated in Figures 1.3 and 1.4.

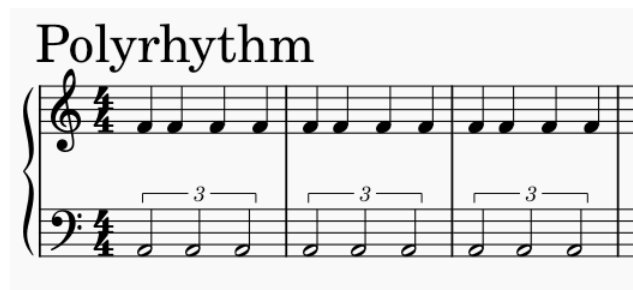


Figure 1.3: Basic example of polyrhythm in MuseScore

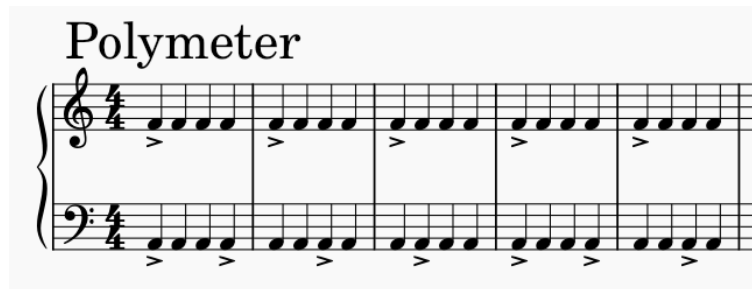


Figure 1.4: Basic example of polymeter in MuseScore. The placement of accents in the bass clef on every third note show that this rhythm is in 3/4 time.

2 Background

2.1 Academic Research

2.1.1 Local Tempo and Expressive Deviations

Repp [15] conducted extensive research into cognitive representation of musical time, and found that performances by musicians often contained deviations from the marked tempo and rhythm from a score, but that they were by no means random or haphazard. On the contrary it was found that musicians could replicate these deviations with relatively high precision. Repp proposed that musicians largely follow an implicit set of rules to generate timing variations from musical structure, and that listeners also subconsciously expect these rules to be adhered to. One especially important consideration was that lengthenings were much more common than shortenings, and are especially common at the end of pieces, sections of a piece or even the end of a phrase. This perhaps points to the idea of speech influencing music, since it is typical to slow down or pause at the end of a sentence when speaking. Many studies indicate a strong relationship between music and speech perception, such as this one by Hausen et al [16] which found a robust link between the two. Hannon [17] went further and conducted a study which showed that the rhythm of music from a given culture is influenced by the rhythm of its language. This was proven by having listeners attempt to classify purely instrumental pieces of music based on their language of origin.

Schreiber et al [18] describe the difficulties of quantifying local tempo in their case study of Chopin's Mazurkas. For reference, a Mazurka is a traditional Polish folk dance in 3/4 time [19]. A tempo estimation system powered by a convolutional neural network called DeepTemp was used. Tempo stability was analysed for two datasets: the Mazurkas and a second dataset called *Ballroom* which was comprised of pop, rock, EDM and ballroom music. It was found that after normalising the local tempi in the Mazurka dataset, only 15.5% of the local tempi fell within a typically used tolerance interval of $\pm 4\%$, compared to 90.9% for the *Ballroom* dataset. As part of their background research they cite Repp's effort to find a definition of "basic tempo", which compromises between global tempo and Inter Beat Interval.

2.1.2 Gradual Tempo Change

Berndt [20] states that the change of tempo throughout a piece of music can be represented by a tempo map, i.e. a sequential list of tempo instructions.

Alternatively, a tempo map can be thought of as a sequence of curve segments mapping symbolic time (number of notes) to tempos. The mean tempo condition is proposed as a way to model these tempo curves, defined as the relative position in the score at which half the tempo change has been processed.

Cope et al [21] conducted research into temporal prediction in music with gradual tempo change, comparing it with already known results for isochronous temporal prediction experiments. They found that participants were much better at detecting exaggerations, for example a note played too early in an *accelerando* section or a note played too late in a *ritardando* section.

Some more detailed findings were found by Schulze et al [22], in an experiment where five volunteers were asked to tap along to a gradually accelerating or decelerating click track. A different approach to Berndt was taken when modelling tempo curves, with the use of a somewhat more complex sigmoidal function which the authors claim was chosen to resemble tempo change in performed music. Asynchrony trajectories, meaning a plot of the error in timing, were generated. It was found that these trajectories followed characteristic M and W shapes for *accelerando* and *ritardando* respectively. The explanation given for this is that the participant is initially late in adjusting to the changing tempo, then after around 6-8 taps they catch up but overcompensate slightly. After this, they tend to fall behind and overcompensate a second time, but to a lesser degree. This consistent pattern of error shows that keeping time with changing tempo does not come as naturally to people, and therefore would be a useful skill to train for the performance of such music.

Perhaps the most thoroughly researched aspect of tempo curves in music is the final *ritardando*, a common expressive device in Western classical music especially wherein a piece gradually slows down towards the end. Friberg and Sundberg [23] examined the possible connection between music performance and locomotion by comparing the final *ritardando* to a runner slowing down. A parameter q was proposed as a way to represent curvature. As part of their experiment they tried modelling tempo change with q values ranging from 1 to 4, which corresponded to quadratic IOI (Inter Onset Interval), $\text{linear}(x)$, $\text{linear}(t)$, $\text{square root}(t)$ and $\text{cubic root}(t)$. The parameter x represents score position, or simply position of the runner, while t represents time. $\text{Linear}(t)$, corresponding to $q = 2$ and $\text{square root}(t)$, corresponding to $q = 3$ received the highest ratings. There is good physical justification for this, as $\text{linear}(t)$ alludes to constant braking force, while $\text{square root}(t)$ alludes to constant braking power. This is reminiscent of a study by Feldman et. al [24], in which the authors examined musical tempo

changes modelled with force dynamics. The force profile is integrated to yield a tempo curve, for example a linearly increasing force produces a quadratic tempo field. Therefore to produce a tempo curve which is smooth on both ends, that is to say of cubic order or higher, the tempo profile needs to be of quadratic order or higher. A key difference with this study is that measures were used as the smallest unit, whereas Friberg and Sundberg made their measurements on a note by note basis instead.

A variety of models was reviewed by Honing [25] in 2003, including those of Friberg and Sundberg, and Feldman et al. The main criticism presented was their lack of consideration for the details of the musical material's rhythmic structure.

2.1.3 Polyrythms

Møller et al [26] performed a study on beat perception in polyrythms. When someone listens to a polyrhythm, there are essentially two different rhythms competing to be heard as the "main" rhythm. It was found that listeners had a propensity to rhythms divisible by 2, so in a 4 against 3 polyrhythm, for example, they would be more likely to count to 4 along with it. It was also found that pitch had a significant effect, with lower pitched rhythms more likely to be heard as the main rhythm. This could simply be because of the prevalence of bass drum in contemporary music, so it would be interesting see if subjects from a different musical background without such a prevalence of bass drum would give the same results.

Kennedy et al [27] studied the effectiveness of auditory and visual stimuli, as well as a combination of the two, in training participants to reproduce a simple 3 against 2 polyrhythm. More specifically, this was a bimanual coordination exercise, meaning the participants were asked to tap the two rhythms simultaneously, one with each hand. It was found that retention performance for the purely visual stimulus was inferior. This is unsurprising, as studies before have found that auditory stimulus is generally more effective for rhythmic entrainment, such as Hove et al [28], who found that auditory beats resulted in more stable tapping synchronization when compared to visual flashes. The results of Kennedy's study could perhaps be made more meaningful by studying other more complex polyrythms, as 3 against 2 is relatively common in contemporary music, so results could be affected by the participants playing based on memory.

2.2 Existing Software

2.2.1 Music and Rhythm Web applications

The landscape for browser based audio applications is changing very rapidly, but ToneJS comes up often as a popular and up to date audio library. The Web Audio API, introduced by Google in 2011 has been crucial for much of the subsequent technical achievements in browser-based audio applications. In their paper from 2015, Roberts et al [29] describe the development of two new libraries built on the Web Audio API, Gibberish.js and Interface.js, for audio synthesis and creation of user interfaces for music performance respectively. Though these two libraries are no longer being maintained, the authors cite ToneJS as a promising new technology in their related works section.

ToneJS began development in early 2014, while the author Yotam Mann [30] was working on another project called *Echo*, as a way to encapsulate and reuse Web Audio functionality that he had been using for years. The first test of this library was his next project *Jazz.Computer*, an interactive song which responds to a listener's scrolling, entirely synthesised in the browser with ToneJS.

One example of a very ambitious web application with these new technologies is the JSS-01 JavaScript Software Synthesizer developed by Michael Kolesidis [31]. He uses among other technologies, ToneJS, and goes so far as to describe it as "the soul of our project". An even more ambitious project is Kameyama's [32] online MIDI editor *Signal*, which works almost like a DAW (digital audio workstation), and can even connect to a user's physical MIDI devices using the Web MIDI API. A year on from its launch, it has nearly 10,000 users per month [33]. Users are able to manipulate the tempo of a song very intuitively by drawing curves on a separate tempo tab. One limitation of this approach is that users will not be able to mathematically define the tempo curves and it could be hard to replicate them exactly.

ToneJS was also used by Lesterberg [34] in her Master's Thesis from 2021 on developing new musicking technologies, where it is described as being "easy and intuitive to code".

The most used web audio library however is HowlerJS, which has more features for general audio playback, such as spatial audio and full codec support [35]. ToneJS seems to be more focused on the creation and synthesis of music from scratch, however. In his article comparing web audio libraries, Arek Nawo [36] likens coding with ToneJS to "being a conductor with code as the baton".

Dr Stephane Pigeon [37] has included a Polyrythm Beat Generator as part of

his site myNoise.net. Though it is not explicitly stated, it has likely been achieved with the php programming language as the url ends in .php. He states that he has taken inspiration from Sub-Saharan African music, in which the polyrhythms can create an almost trance-like or hypnotic feeling. The author does acknowledge that timing drifts can occur after a while depending on the user's browser and computing power. An interesting feature of this application is that up to ten different rhythms can be played simultaneously, where most examples of polyrhythms tend to only have two simultaneous rhythms.

2.2.2 Digital Audio Workstations

The ability to manipulate tempo in interesting ways is nothing new for DAW's. In Petersen's [38] review of new features added to major DAW's from 2010, some form of tempo manipulation was included in the new features for three of these application: *Logic Pro*, *Cakewalk* and *PreSonus*. In *Logic Pro* a new feature called Flex Time was added for timing and tempo manipulation. In *Cakewalk*, resolution settings were added for tempo changes, so users could choose to set tempo either on every beat, every measure or every clip. Meanwhile in *PreSonus*, functionality was added for changing tempos within an event.

Tempo manipulation is now a standard and expected feature in any DAW. For example, even in LMMS, a lightweight open source alternative to commercial DAWs, the tempo can be automated just like any other property [39]. In Reaper, another popular and affordable DAW, manipulating tempo is intuitive and flexible. In fact, Jeff Kaiser [40] has uploaded a tutorial specifically on creating complex click tracks in Reaper. This tutorial outlines a few different ways to interact with the tempo. The tempo envelope be edited with standard envelope editing tools. Alternatively the user has a few existing menu options such as "Insert -> Tempo/-time signature change marker" and "Gradually transition tempo to next marker". In Cubase, another DAW, points can be drawn on a tempo track to control tempo change through time. This works well for MIDI data, and can work for audio tracks too provided they are quantized to the tempo track [41].

Features for creating polyrhythms in DAWs tend to be less standardised than those for manipulating tempo curves. Some online resources, such as this article by Trandafir [42], suggest the approach of finding the lowest common multiple of the two rhythms and creating the polyrhythms on a grid of that subdivision. For example, for a 4 against 3 polyrhythm, the user would need to subdivide measures into 12, with one rhythm being 4 notes of length 3, and the other being e notes of length 4. An illustration of this concept in MuseScore is shown in Figure 2.1.

While this works fine, it is still quite a bit of work for the user and could certainly be made easier. Additionally this approach becomes more difficult for polyrhythms with a less simple ratio, or when more than 2 distinct rhythms are involved, as in Dr Pigeon's Polyrhythm Beat Generator discussed in Section 2.2.1.



Figure 2.1: Using the lowest common multiple approach for a 4 against 3 polyrhythm

A different approach is to use a time stretching tool to manipulate the lengths of notes in a measure, for example making 5 notes fit into a measure in 4/4 time. Ableton Live seems to be a popular DAW for this approach. In his tutorial for warping tempo tracks in Ableton Live, Rory PQ [43] explains that any audio sample can be time-stretched to play in sync with a given tempo. There are in fact six different warp modes for different types of audio, accounting for factors such as amount of transients and whether or not to preserve pitch.

2.2.3 Other applications

MuseScore is an open source musical notation editor, and while it does not have a built in feature for adding tempo curves, a plugin has been developed for it by Johan Temmernman [44], allowing for both linear and exponential tempo changes, using the concept of mean tempo condition, which we have already encountered in Berndt's research in section 2.1.2. As MuseScore does not explicitly have support for continuous tempo changes, this had to be achieved by adding a hidden tempo marker to each note.

A more complex feature which will not be considered for this project is automatic beat detection. In music with a regular drum beat this can be an easy task, as the bass drum will almost always play on the downbeat of every measure, and the snare will often play on the third beat. If a drum kit or similar percussion is not present on the track, however, this becomes a much harder task. This feature has been implemented in the Moises App. The publishers claim to have developed the world's first Smart Metronome and Audio changer, which can generate click

tracks which follow any variation in tempo. They have used AI to achieve this feature along with other useful tasks for musicians such as separating vocal tracks from a mix [45]. Mounir et al [46] outline the differences between data-driven and non data-driven note onset detection (NOD) systems, and state that data-driven systems have been found to slightly outperform their non data-driven counterparts. However this is only the case if they are trained using large annotated databases, with the data needing to be annotated manually.

3 Aims and Objectives

3.1 Summary

The main aim of this project is to create a useful and convenient web application which allows users to create and edit click tracks. To set it apart from other web based solutions, I aim to add more advanced rhythmic functionality that may typically be found in a desktop based DAW (Digital Audio Workstation). This will be done by dividing the track into sections, each of which can be edited individually through a simple form. A major aim of this project is ensuring that the user can simply open a url and start editing click tracks, without the need for any downloads or installations, or to sign up for any account.

The first and most important feature to be implemented will be the ability to plan out a click track in sections, and to be able to edit and delete these sections, changing values such as the time signature or number of measures. The user should have more advanced rhythmic options available to them, such as polyrhythms and smooth tempo change.

After creating a click track, a user should be able to listen to it directly in the browser, as well as download it in a variety of file formats. Finally, a user should be able to sign up for an account so that they can save click tracks they have been working on to come back to later, but this should be strictly optional, to minimise any friction to using the application.

3.2 Functional Requirements

1. The user **must** be able to add a section, to act as the basic building block for the click track.
 - (a) The user **must** be able to add this section to any point in the click track, for example at the beginning, or between two pre-existing sections, to allow for more flexibility when creating the click track.
 - (b) The user **must** be able to modify the section's tempo (in bpm), duration (in measures) and time signature, so that each section can have different rhythmic properties.
 - (c) If the user selects the tempo change option, they **must** be able to change the starting and ending tempos, as well as the mean tempo condition, as this allows for the creation of accelerando and ritardando sections with a variably shaped tempo curve.

- (d) If the user selects the polyrhythm option, they **must** be able to select a different time signature for the second rhythm, in order to have two distinct rhythms for the polyrhythm.
 - (e) If the polyrhythm option is not selected, the user **must** be able to change the accented beats, so that different notes besides the first beat of every measure can be played louder.
- 2. The user **must** be able to edit a section of a click track to allow more flexibility in the construction of the click track.
- 3. The user **must** be able to delete a section of a click track, so that mistakes can be easily undone and to allow more flexibility in the construction of the click track.
 - (a) The form for editing the section **must** be pre-populated with the section's current data, to make it clear which section the user is editing.
- 4. The user **must** be able to toggle help dialogues on and off, as the help icons might be a nuisance to the user once they are more familiar with the application.
 - (a) The toggle for this **must** always be visible in the upper navbar, so that the user can easily toggle help icons back on if they are stuck.
- 5. The user **must** be able to select samples for playback, so that they can customise the sound of the click track.
 - (a) A reasonable sample **must** be selected by default for the primary sample, so that the user can listen to their click track right away even if they do not select any samples.
 - (b) The user **must** be able to listen to a sample from the sample selection menu to preview it, by clicking on the listen icon, so that they will be able to see whether they want to use that sound before playing the entire click track.
 - (c) The user **must** be able to select a primary sample by clicking on it.
 - (d) The user **should** be able to optionally select a second sample, also by clicking, so that they can play the click track with a different sound for strong and weak beats or for polyrhythms.

- (e) The user **must** be able to deselect the second sample by either clicking it again, or by selecting a new primary sample, in case they want to go back to only using one sample.
6. The user **must** be able to play the click track directly in the browser, so that the application can easily be used without downloading any files.
 - (a) This action **should** only be available to the user if the click track contains at least one section, to prevent any errors from trying to play an empty click track.
 - (b) The user **must** be able to stop the playback immediately by pressing the stop button, so that they will not have to close or reload the tab to stop playback.
 7. The user **must** be able to export the click track to a file to download, so that they can save it to their device and potentially import it into other applications.
 - (a) This action **should** only be available to the user if the click track contains at least one section, to prevent any errors from trying to export an empty click track.
 - (b) The user **must** have a choice of file formats, including MIDI and at least one audio file format such as flac or mp3, as MIDI files and audio files can be useful for different applications.
 - (c) The user **should** see a progress bar while the conversion is in progress, so that they know their request is being processed.
 - (d) Once finished, the user **must** be provided with a link to download the file.
 8. The user **must** be able to view a visualisation of the tempo curve for the entire track, so that they can see how the tempo changes throughout the entire track.
 9. This action **should** only be available to the user if the click track contains at least one section, to prevent any errors from trying to render the visualisation for an empty click track.
 10. The user **must** be able to toggle the visualisation on and off, so that it does not take up extra space on the screen when it is not needed.

11. When toggled on, the visualisation **should** include two line charts, one for symbolic time, the other for physical time, as it will be interesting for the user to see the difference between the two.
12. The user **must** be able to create an account to save their click tracks and come back to them later.
 - (a) On successful registration, the user **must** be logged in automatically, so they can start saving click tracks right away.
 - (b) The logged in user **should** now have an option to save a click track to their account on the main editor page, so that they can continue using the application in the same way as before, but with the option to save their click tracks to the server.
 - (c) The user **must** be required to select a title when saving the click track, to prevent click tracks being saved with an empty string as their title.
 - (d) Saving the click track **must** save the click track to a database and redirect the user to a page with a view of all their saved click tracks, so that they can be confident that it is saved persistently.
 - (e) On the saved click tracks page, a user **must** be able to click the "edit" button on any of the click tracks, which will take them to the editor page with that click track loaded in.
 - (f) When editing a click track, the user **must** be able to click on save changes, which will update the click track in the database, and redirect the user back to their saved click tracks page.
 - (g) The user **must** be able to delete a click track. Clicking the "delete" button **should** first bring up a confirmation dialogue, so that the user cannot accidentally delete a click track with a single misclick.
 - (h) Upon confirming, the click track **must** be deleted from the database.

3.3 Non Functional Requirements

1. The main page should load fast, ideally with TTI (time to interactive) [47] less than 2 seconds, as studies have shown that the probability for a user to click away increases by 32% when a page's load time goes from 1 second to 3 seconds [48].
2. A help icon should show up next to parts of the application that are likely to need explanation for the user, so that it is easy to find documentation

when needed. The help tooltips should be informative without being overly verbose.

3. The user should be informed about responses to their actions, for instance through error messages in form fields and alerts at the top of the page, to maintain a good overall user experience.
4. The layout should be responsive and appear just as good on mobile devices as on desktops, with the exception of the tempo visualisation, as this requires a wide viewport for best results.
5. The playback of the click track should start without noticeable latency.
6. The timings of the notes should be precise, as this application could be used for practicing fast and intricate rhythms.
7. The user should be able to easily distinguish between strong and weak beats, to make playing along to the click track in any given time signature easier.

4 Technical Implementation

4.1 Overall Structure and Design

A significant aspect of the application's structure is the decoupling of the frontend and backend. In this particular case, the backend needs to complete two main functionalities, audio processing and user management, which are quite separate in their scope, so it was decided to go further with the idea of decoupling and follow a microservice architecture. This greatly improves the resilience and fault isolation of the application [49]. For example, if the audio processing microservice were to crash, users could still edit and play click tracks, as well as log in and save click tracks to their account. Likewise, if the user management backend crashed, users could still edit and play click tracks as well as export them to files.

In fact, the application could even be run offline provided the sample files from Cloudinary were cached, which could be done with service workers. This would be an important step to transforming the application into a Progressive Web App (PWA), as the offline experience criteria is stated as "Where connectivity isn't strictly required, your app works the same offline as it does online" [50]. Alternatively, the audio files could just be served statically, but this would be a less flexible option, as in the future I may want to allow users to upload their own samples to use.

As convenience is a core goal in this project, I ensured that having an account was purely optional, and that any user, logged in or not, could start editing clicktracks as soon as they open the application.

4.2 Justification for Chosen Tools

ReactJS was chosen for the frontend as it is well suited to applications with rich interactivity and complex state management. Additionally it is one of the most mature and popular frontend frameworks, so has plenty of documentation and extra libraries to extend its functionality, of which I used a few, including Redux Toolkit for state management and Formik for managing forms.

For the audio processing microservice, Flask was chosen as the server-side web framework, mainly as it is a python framework, as python has many excellent libraries for music processing and analysis, such as librosa, music21 and pydub. Additionally, flask was chosen over a more "batteries-included" framework such as Django because of the limited need for traditional web server functionality such as user management and database access.

ExpressJS was chosen for the user management backend, as I am most familiar with writing typical CRUD APIs with it, after following the excellent *Full Stack Open* course from the University of Helsinki.

Finally, Cloudinary was chosen as the solution for static file storage, primarily due to its generous free tier, as well as its functionality for media transformations, for example returning an audio clip with modified pitch or volume. Another reason for choosing Cloudinary is its integrated CDN (content delivery network), which achieves faster load times on these static files for users around the world. The integrated CDN is a major advantage it has over competing services such as Amazon's S3 Buckets [51].

Figure 4.1 illustrates on a high level how these technologies all link together for the application.

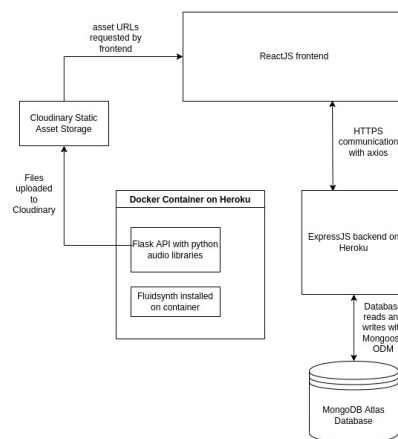


Figure 4.1: Outline of the technology stack

4.3 Development Environment

One drawback of using a microservice architecture is that it resulted in more setup when developing locally, as this involved opening three separate terminal windows and typing commands into each. After learning some basic bash scripting, however, I was able to set up the development environment, including the docker container for the audio conversion service (see Section 4.7) by typing only a single command, shown in Figure 4.2.

4.4 Frontend

User interaction is handled with a frontend built in React. Perhaps the biggest challenge here was managing state of the application, with many pieces of data


```

$ dev.sh
1  #!/bin/bash
2  gnome-terminal \
3      --tab --title="Audio Backend" -- bash -c "cd ../Advanced-Clicktrack-Audio-Backend; ./rebuild.sh; $SHELL"
4  gnome-terminal \
5      --tab --title="User Backend" -- bash -c "cd ../Advanced-Clicktrack-User-Backend; npm run dev; $SHELL"
6  gnome-terminal \
7      --tab --title="Frontend" -- bash -c "npm start; $SHELL" \

```

Figure 4.2: Shell script for starting the development environment

being accessed by different parts of the application. For this purpose, I used a state management library called Redux along with its extension Redux Toolkit. Styling was handled by the Material UI library, eliminating the need to spend too much time writing CSS.

4.4.1 Frontend Structure

As the frontend was far more complex and feature rich than the backend in this project, it was especially important to use a well organised and extensible file structure and adhere to the principal of separation of concerns. All source code went into the src directory of the project, which itself contained the directories components, config, reducers, services and utils. The purpose of each directory is briefly described in Table 1. The idea of directories for services and reducers is taken from the University of Helsinki's Full Stack Open course [52].

| Directory | Purpose |
|------------|---|
| components | Stores components, the modular building blocks of the user interface |
| config | Stores constants that can be changed to affect the whole application |
| pages | Stores components which take up a full page in the routing system, i.e. the Main Page or the Login Page |
| reducers | Stores logic for interacting with the application's global state tree |
| services | Stores logic for communicating with the backend, and potentially any external APIs in the future |
| utils | Stores various utility functions, but mainly for making calculations related to click tracks |

Table 1: Purpose of directories in the frontend source code

4.4.2 State Management

The state of the application can be visualised at any time in Chrome Devtools by using the Redux Chrome extension. We can see the global state tree in Figure

4.3, which illustrates how the state is comprised of three distinct categories. Each category was handled by a separate file in the reducers directory. The first reducer, sections, stores data on the sections of the click track, as well as where the form for adding or editing sections should be rendered. The latter could also logically fit into the ui category. The samples reducer keeps track of which samples are selected, and finally the ui reducer is concerned with the visibility of different ui elements.

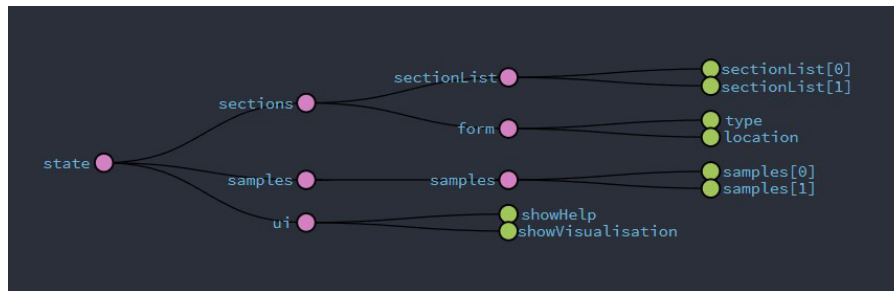


Figure 4.3: Visualisation of the application state in Redux DevTools

4.4.3 Audio Playback

The ToneJS library powered the playing and loading of audio samples. Thanks to an article by Chris Wilson [53], a mistake I managed to avoid early was using built-in JavaScript functions such as `setInterval()` and `setTimeout()` for scheduling playback. The execution of their callback functions can be delayed by tens of milliseconds or more, as they are on the main thread, so can be blocked by any number of processes such as rendering content or processing HTTP requests. Instead, it is suggested to use the WebAudio API, which runs in a separate thread. ToneJS is built on the WebAudio API, so I was confident that the timings of scheduled notes would be precise. High precision was important, as the application allows the user to select a tempo as high as 400bpm. With the addition of polyrhythms, notes become even more dense, and any deviations from the intended timing would be immediately noticeable.

4.4.4 User Interface

The layout of the user interface was done quite differently from a typical DAW or sequencer, which will generally have a left to right view of the entire track. Instead, the layout here is based on sections, organised vertically, which makes for easier viewing on mobile devices. Information for each section is rendered

in a `SectionDisplay` component, and can be modified by interacting with the `SectionForm` component. It was important that the form for adding or editing a section is rendered in the correct location, to make it clear to the user which section they are working on. Additionally, the decision was made to display information concerning the section inside a collapsible accordion component, to allow the `SectionDisplay` component to take up less vertical space. Figure 4.4 shows both of these components in action, with one section displayed in its collapsed state and the other expanded.

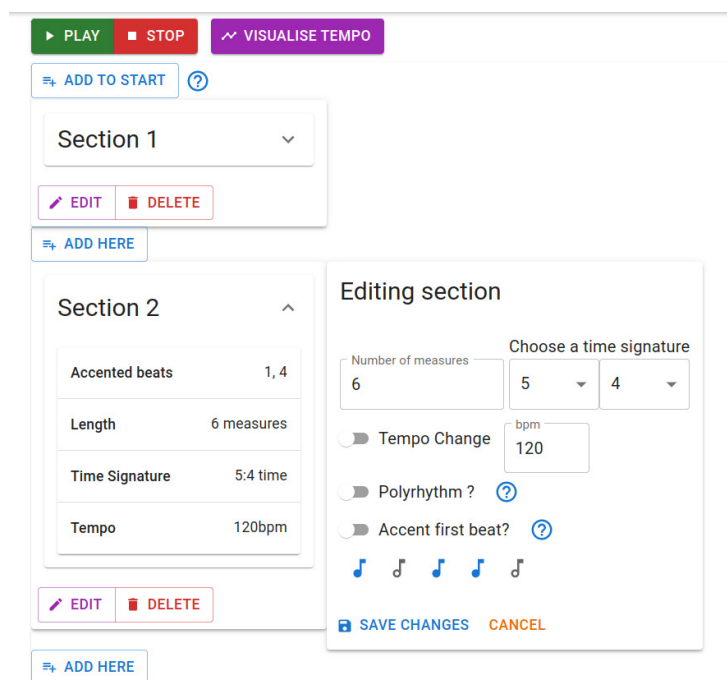


Figure 4.4: Editing a section

An important aspect of the user interface is visual feedback. To this end, a notification banner at the top of the page was implemented, relaying information about events to the user, such as a successful login or saving the changes made to one of their click tracks. Additionally, the Formik library was used on the login and register forms to help render informative error messages next to relevant form fields, for instance notifying a user that their password is too short, as in Figure 4.6.

4.4.5 Rhythm Calculations

Throughout most of the development, time signature was represented simply as beats per measure, with beats always being hardcoded to quarter notes. This greatly simplified the code for audio processing, but would not provide a good user

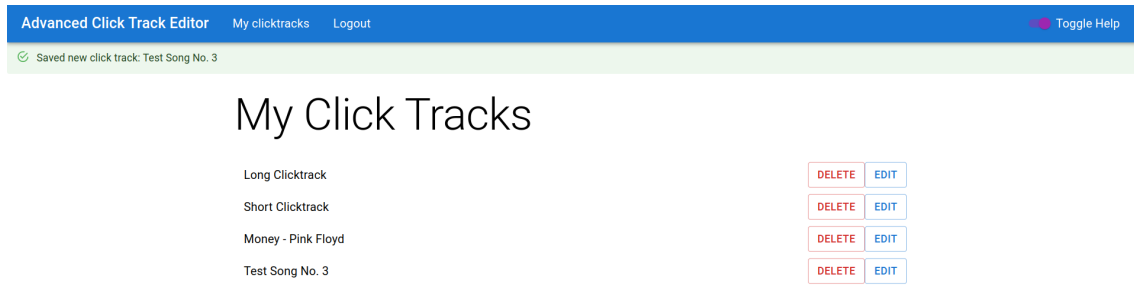


Figure 4.5: Notification informing the user that their click track has been saved successfully

Register

Name

Username

Field is required

Password

Password must be at least 8 characters

Confirm Password

Field is required

REGISTER

Figure 4.6: Error messages displayed on the register form

experience, as musicians are used to a different notation for time signature, that is a fraction where the numerator represents the number of beats per measure and the denominator represents the duration of these beats, i.e. 4 for quarter notes or 8 for eighth notes. Therefore I implemented a selection for the denominator with the options 2, 4 and 8, as it is very rare to see values other than this in music. For all calculations and business logic, it was however much easier to always represent the beats as quarter notes. This lead to the need to recalculate bpm when displaying it in the user interface. So for example, if a user created a section in 6/8 time with bpm = 110, internally the program would store the time signature as 6/4 and double the bpm. When displaying the bpm to the user, it would need to be multiplied by a factor of 4 / denominator.

The function for playing the click track takes an array of times to schedule the playback of the samples, which meant that these times would need to be calculated based on the data stored in the section data structure. This was simple enough for sections with constant tempo, but slightly more complicated for the sections with tempo change. On a basic level, an empty array was initialised, then for each section the onset time of each note was calculated, by first calculating the bpm and then the interval for that note. This means that the resolution for changing tempo is based on individual beats, similar to the feature in the *Cakewalk* DAW described in Section 2.2.2. Additionally, a boolean `downBeat` value was assigned based on input from the section form. Finally, an offset time had to be included, since aside from the first section, the starting time would not be 0 but instead would be equal to the ending time of the previous section. For that reason, each time the `buildClickTrackSection` function is called, it takes `startTime` as an argument and returns `endTime`.

```

3  export const buildClickTrackSection = (startTime, sectionData, last=false) => {
4    const bpmArray = makeBpmArray(sectionData)
5    const intervalArray = bpmArray.map(bpm => 60/bpm)
6    const timeArray = intervalArray.map((interval, idx) => {
7      return idx > 0
8        ? startTime + intervalArray.slice(0, idx).reduce((a, b) => a + b)
9        : startTime
10   })
11   const accentArray = sectionData.rhythms[0].accentedBeats
12
13   const endTime = timeArray[timeArray.length - 1] //Last entry of the timeArray
14
15   // To keep the final click on for visualisation
16   if (last) {
17     const sectionTimeArray = timeArray
18     .map((time, idx) => {
19       accentArray.includes(idx % sectionData.numBeats)
20         ? { time, bpm: bpmArray[idx], downBeat: true }
21         : { time, bpm: bpmArray[idx], downBeat: false }
22     })
23     return { sectionTimeArray, endTime }
24   }
25
26   const sectionTimeArray = timeArray
27     .slice(0, timeArray.length - 1)
28     .map((time, idx) => {
29       accentArray.includes(idx % sectionData.rhythms[0].timeSig[0])
30         ? { time, bpm: bpmArray[idx], downBeat: true }
31         : { time, bpm: bpmArray[idx], downBeat: false }
32     })
33
34   return { sectionTimeArray, endTime }
35 }
36

```

Figure 4.7: Function to build a section of the click track

The variable `bpmArray` represents the current bpm at each note and is calculated based on Berndt's formulation of the mean tempo condition. Berndt [20] gives the Tempo at an arbitrary symbolic time d in the equation below,

$$Tempo(d) = \left(\frac{d - d_m}{d_{m+1} - d_m} \right)^{p(i_m)} (t_{2,m} - t_{1,m}) + t_{1,m} \quad (4.1)$$

where d_m and d_{m+1} are the symbolic times at the beginning and end of the section, respectively, and $t_{1,m}$ and $t_{2,m}$ are the tempi at the beginning and end of

the section. The exponent $p(i_m)$ is derived from the mean tempo condition i_m as shown:

$$p(i_m) = \ln 0.5 / \ln i_m \quad (4.2)$$

If we consider a case where the starting time is 0, Equation 4.1 can be simplified as shown [20]:

$$Tempo(d) = \left(\frac{d}{d_{m+1}} \right)^{p(i_m)} (t_{2,m} - t_{1,m}) + t_{1,m} \quad (4.3)$$

We can see the implementation of Equations 4.2 and 4.3 in the functions `calcExponent` and `bpmAtCurrentBeat` in Figure 4.8. This function can then be mapped onto each note to obtain the bpm array.

```
src > utils > .js tempoCurveCalculator.js > [0] getFullTempoDataSymbolic
1  const calcExponent = meanTempoCondition => Math.log(0.5) / Math.log(meanTempoCondition)
2
3  const bpmAtCurrentBeat = (currentBeat, exponent, numBeats, startTempo, endTempo) => {
4    return ((currentBeat / numBeats) ** exponent) * (endTempo - startTempo) + startTempo
5  }
6
7  const makeBpmArray = (sectionData) => {
8    const numNotes = sectionData.overallData.numMeasures * sectionData.rhythms[0].timeSig[0]
9    const exponent = calcExponent(sectionData.overallData.mtc)
10   const bpmArray = Array.from({ length: numNotes + 1 })
11   .map((_, val, idx) => {
12     return bpmAtCurrentBeat(
13       idx,
14       exponent,
15       numNotes,
16       sectionData.rhythms[0].bpm[0],
17       sectionData.rhythms[0].bpm[1]
18     )
19   })
20   return bpmArray
21 }
```

Figure 4.8: Function to calculate the bpm array of a section

The addition of polyrhythms as a feature introduced a new layer of complexity to calculations, which will be explored below.

Firstly, we can see in Figure 4.9 that in our function `getClickTimesPoly`, each section is checked for whether it is a polyrhythm, by checking if the array of rhythms on it has length greater than one. A different function is called to build that section depending on the result.

The function to build a polyrhythmic section itself is shown in Figure 4.9, where we can see that the section data is first split into its two different rhythms, which are passed as arguments to another function which returns a time array for each rhythm. Depending on how the different rhythms line up, there can be some extra times added to the end, so lines 140-144 of Figure 4.9 include logic for removing them. Finally, and most importantly, the two time arrays are combined with the `combineTimeArrays` function. The functionality for this is different depending on

```

22 export const getClickTimesPoly = (sections, numInstruments) => {
23   const clickTimesPoly = []
24   let startTime = 0
25
26   for (let i = 0; i < sections.length; i++) {
27     const { sectionTimeArray, endTime } = sections[i].rhythms.length > 1
28       ? buildPolyrhythmicSection(startTime, sections[i], numInstruments)
29       : buildClickTrackSection(startTime, sections[i])
30     clickTimesPoly.push(...sectionTimeArray)
31     startTime = endTime
32   }
33
34   return clickTimesPoly
35 }

```

Figure 4.9: Function to generate click times taking polyrhythmic sections into consideration

whether the click track is to be played with one or two distinct samples.

```

63 export const combineTimeArrays = (timeArrays, numInstruments) => {
64   if (numInstruments === 1) {
65     // Combine the two time arrays into one
66     const combinedArray = timeArrays
67       .reduce((a, b) => a.concat(b))
68       .sort((a, b) => a - b)
69     // Round off the numbers to prevent weird floating point imprecisions
70     .map(time => Math.round(time * 10 ** 6) / 10 ** 6)
71     .filter(t => !isNaN(t)) // Remove the NaN weirdness from the end
72
73     // stringify the clickTime objects so that they can be compared and the duplicates
74     // can be deleted. Downbeats are when the times of two clicks are the same, as this
75     // means that the polyrhythms are lining up
76     const clickTimeArrayWithDuplicates = combinedArray.map((time, idx) => {
77       // Check if time is equal to the next time
78       if (idx < combinedArray.length - 1 && time === combinedArray[idx + 1]) {
79         return JSON.stringify({ time, downBeat: true })
80       }
81
82       // Check if time is equal to previous time
83       if (idx > 0 && time === combinedArray[idx - 1]) {
84         return JSON.stringify({ time, downBeat: true })
85       }
86       return JSON.stringify({ time, downBeat: false })
87     })
88
89     // Remove duplicates
90     const clickTimeArray = [... new Set(clickTimeArrayWithDuplicates)]
91     .map(ct => JSON.parse(ct))
92     return clickTimeArray
93   }

```

Figure 4.10: Logic for combining time arrays when only one sample is selected for playback

Let us first consider the case of one sample, shown in Figure 4.10. To start, the two time arrays are concatenated together, then the resulting array is sorted in ascending order. Then the `downBeat` property with a value of `true` is given to any duplicate times, essentially creating accents where the notes line up. Finally duplicate notes are removed by converting the array into a set and then back into an array.

In the case of two distinct samples, shown in Figure 4.11, we want to achieve something a bit different. Instead of using the `downBeat` property, we will instead use a property `secondInstrument` to instruct the playback function on which sample to use. Downbeats will naturally be emphasised due to having two samples playing simultaneously. Once the `downBeat` property has been applied, the arrays are concatenated and sorted as before, only this time duplicates are not deleted,

```

94   } else if (numInstruments === 2) {
95     const strongClickArray = timeArrays[0].map(time => {
96       return { time, secondInstrument: false, downBeat: false }
97     })
98     const weakClickArray = timeArrays[1].map(time => {
99       return { time, secondInstrument: true, downBeat: false }
100     })
101     const combinedArray = [strongClickArray, weakClickArray]
102       .reduce((a, b) => a.concat(b))
103       .sort((a, b) => a.time - b.time)
104     // Round off the numbers to prevent weird floating point imprecisions
105     .map(click => {
106       return { ...click, time: Math.round(click.time * 10 ** 6) / 10 ** 6 }
107     })
108     .filter(click => !isNaN(click.time)) // Remove the NaN weirdness from the end
109     // Add 0.00001 to the times of all downbeats on the second instrument, to prevent
110     // ToneJS from throwing an error
111     .map((click, idx, arr) => {
112       if (idx > 0 && click.time === arr[idx - 1].time) {
113         return { ...click, time: click.time + 0.00001 }
114       }
115       return click
116     })
117     return combinedArray
118   }
119 }

```

Figure 4.11: Logic for combining time arrays when two samples are selected for playback

since we want the two different samples to play at the same time. ToneJS requires that each time given is strictly greater than the previous time though, so a tiny increment is added to the second sample each time the two rhythms align.

4.4.6 Data structure and default values

The current click track which the user is working on gets stored in the redux store of the application as an array of Section objects. When adding a new click track section, default values are displayed in the form. Figure 4.12 shows the JavaScript object which stores these defaults and also serves to illustrate the data structure of an individual section.

```

src > config > JS sectionDefaults.js > [⌕] defaults
1  export const defaults = {
2    overallData: {
3      numMeasures: 4,
4      mtc: 0.5
5    },
6    // Can be array of length > 1 for polyrhythms
7    rhythms: [
8      {
9        bpms: [120, 120], //bpm at start and end of section
10       timeSig: [4, 4],
11       accentedBeats: [0]
12     }
13   ]
14 }

```

Figure 4.12: Default Values for a click track section

As shown in Figure 4.12, the data for a section is divided into 2 nested objects to account for possible polyrhythms. The first of these, `overallData`, includes data that remains constant for the whole section and is not affected by the individual rhythms. The second, `rhythms`, is an array of objects each representing a rhythm, with the first being the primary rhythm. This allows for polyrhythms to be optional without including many null variables. Each section is also assigned a universally unique id (`uuid`) on creation, to allow for easy and predictable manipulation of the section list. This is shown in Figure 4.13, where a `uuid` is generated for a newly added section in the `addSection` function, and then is used to look up a section in the `updateSection` function.

```
11 const sectionSlice = createSlice({
12   name: 'sections',
13   initialState,
14   reducers: {
15     addSection(state, action) {
16       const data = action.payload
17       const newSection = { ...data, id: uuidv4() }
18       const idx = state.form.location
19       state.sectionList.splice(idx, 0, newSection)
20     },
21     updateSection(state, action) {
22       const data = action.payload
23       state.sectionList = state.sectionList.map(section =>
24         section.id !== data.id ? section : data
25       )
26     },
27   },
28 })
```

Figure 4.13: Logic for creating and updating sections in the state

The default values are based on what would be most common in existing music. A default time signature of 4/4 was an easy decision, as it is by far the most common time signature in western music. Likewise, tempo was set as constant by default, at 120bpm, considered a fairly average tempo, and which corresponded to a neat, round IOI of 0.5 (half a second between notes). Polyrhythms and custom note accents were also off by default. A JavaScript object saved in a separate file within the config folder of the app was used to store these default values, so they could be easily changed at any time.

4.4.7 Tempo Visualisation

Line charts were used to give a visual representation of the tempo throughout the entire click track, created with the help of the Recharts library. The use of an existing chart library made it relatively quick to get visualisations into the application, which was perfect for the time constraint of this project, however for better performance and more customisation, the tempo curves could be drawn

from scratch on an HTML canvas element. The application includes both a chart in symbolic time, i.e. time measured in notes, and physical time, time measured in seconds, as this difference can be quite significant, especially for drastic tempo changes. An example of both these charts can be seen in Figure 4.14. This was one instance in which the responsive design of the application was a bit weaker, as it was crucial to have a wide viewport to visualise the tempo change throughout the track, especially for longer tracks. For extra clarity, regions of the charts are shaded in different colours to represent the sections in the track. In the chart with symbolic time, vertical lines are also included to represent measures within each section.

Two different approaches were considered when passing data to the chart components. The first was to simply plot a point for each note in the click track, and use the linear line option to connect them, which would still give the appearance of smooth curves for tempo curve sections with a lot of notes. This would result in the most accurate charts, but would also require more computing power to plot each individual point and recalculate them all when any tempo data in the click track changes. This approach was used for the chart with physical time, to be able to accurately show the exact bpm of notes at each time. The second approach was to only give the bare minimum data to plot the sections, and use one of the built-in curve fitting algorithms provided by Recharts to draw the lines. This bare minimum data was simply the x and y values at the boundaries between sections and at the point of mean tempo condition. This approach was used for the chart with symbolic time, as it is there simply to show the user the overall shape of the tempo curves, and did not need to have every single intermediate point plotted with perfect accuracy.

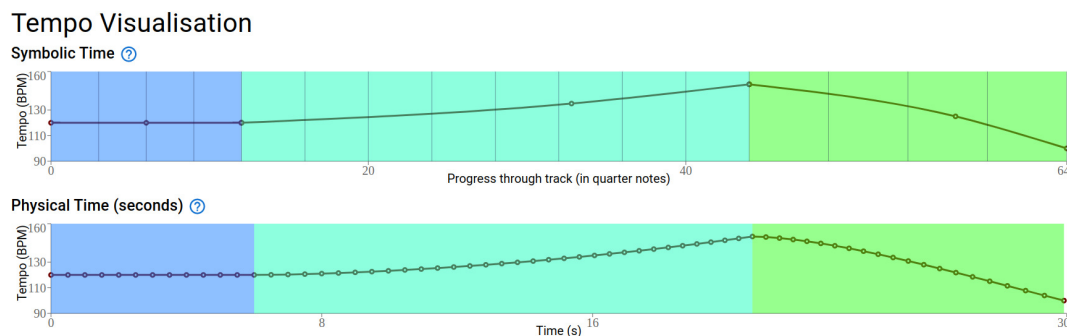


Figure 4.14: Line charts for tempo visualisation

A much smaller line chart was also integrated into the section form component to help the user visually understand the effect of changing the mean tempo condition, as show in Figure 4.15. Without it, a slider would not be very indicative

at all.

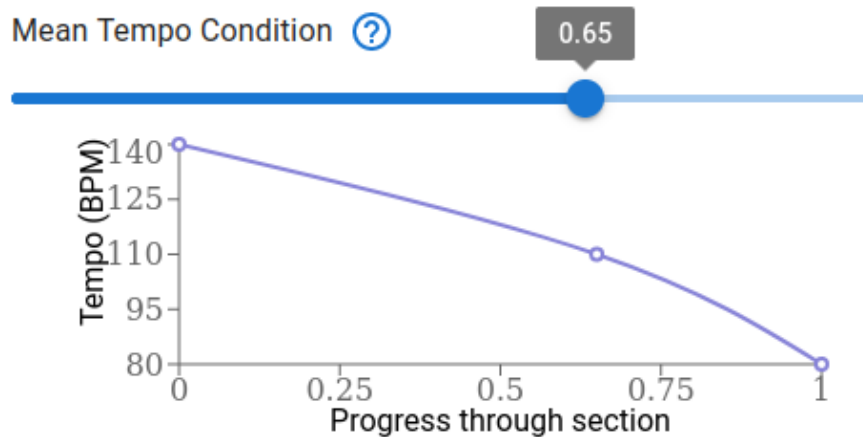


Figure 4.15: Selection of mean tempo condition

4.4.8 Help Dialogues

To provide adequate documentation for users without avoid cluttering up the page, I decided to implement a help dialogue system where users could hover over a question mark icon to get more information about a certain part of the application. These icons could be toggled on or off via a button at the top of the page. This was an instance in which Redux's state was useful, as the Boolean value `showHelp` was accessed by many different components in the component tree.

4.5 Audio Processing Microservice

4.5.1 Structure

Though this backend service is a significantly smaller codebase than the frontend, it was still important to structure it in a logical way, with functionality separated into 5 main python files (not including `__init__.py`) in the app directory. Since the backend was used more as a file conversion microservice than a traditional backend, well established design patterns such as model-view-controller or n-tier architecture were not relevant here. This made Flask especially well-suited to the particular use case, since it is lightweight and unopinionated and does not force the user to adopt any of these structures in the way that a framework such as Django or Ruby on Rails would.

Since most of the complexity resides in the audio and symbolic music processing functions, it was decided to follow a facade design pattern, described by

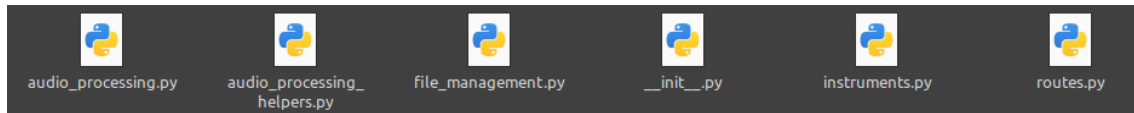


Figure 4.16: Python files used for backend

Gamma et al [54] as providing a higher level interface that makes the subsystem easier to use. This meant the service could only be interacted with through 4 clearly named API routes: `make_midi()`, `make_wav()`, `make_flac()` and `make_ogg()`. This proved to be a good design decision, as it meant I was able to change the underlying algorithms and functionality for these endpoints without having to change anything on the frontend.

Simply looking at one of the four route handlers, shown in Figure 4.17, can illustrate how the data flows in the backend. First the data sent from the frontend is parsed, then passed to the appropriate audio processing function, which resides in the file `audio_processing.py`. Behind the scenes, some helper functions from `audio_processing_helpers.py` are also called. A filename is returned, which is then passed to the file upload function, residing in `file_management.py`. If successfully uploaded, the url is returned to the route handler, so it can then finally return the url to the frontend.

```

30 @app.route("/api/make_wav", methods=["POST"])
31 def make_wav() -> dict:
32     data = request.json
33     section_data = data["sectionData"]
34     note_bpms = data["noteBpms"]
35     instrument_vals = data["instruments"]
36     wav_filename = make_file_with_fluidsynth(section_data, note_bpms, 'wav', instrument_vals)
37     wav_url = upload_file(wav_filename)
38     return (
39         {"url": wav_url}
40         if wav_url != "error"
41         else {"error": "Something went wrong with the file"}
42     )

```

Figure 4.17: API route handler for requesting a wav file

4.5.2 MIDI file creation

The music21 python library was heavily used when it came to creating MIDI files out of the JSON data sent from the frontend. The music21 library allows for representing a piece of music symbolically in an object oriented way with a hierarchy of stream > parts > measures > notes and rests, where stream is an abstract data structure which allows any sort of musical instruction or information to be stored at any offset. An offset represents a point in time, measured in quarter notes [55]. For this project parts were used to separate what was played by different

instruments as well as for the secondary rhythm of polyrhythms. Measures did not need to be explicitly defined. Instead, time signature markers were used, which automatically subdivided the part into measures.

The `make_midi_file` function is quite lengthy, but pseudocode for it in the case of of a click track with no polyrhythms is shown in Figure 4.18.

```

1  function make_section(time_signature, num_measures, note_pitch):
2      quarter_note_length = 0.5 / (4 / denominator of the time_signature)
3      number_of_notes = numerator of time_signature * num_measures
4      initialise empty list
5      for count in range(0, number_of_notes):
6          add a note followed by a rest, each of length = quarter_note_length to the list
7      return list of notes and rests
8
9  function make_midi_file(section_data, note_bpms, instrument?):
10
11      initialise empty stream object
12      initialise empty main rhythm part object
13
14      set note_pitch to default value of middle C
15      if instrument is specified
16          change note_pitch to instrument.playback_note
17
18      for entry in section_data:
19          section = make_section(entry.time_signature, entry.num_measures, entry.note_pitch)
20          add section to main rhythm part
21
22      for bpm in note_bpms:
23          calculate offset at which to insert the tempo marker, based on duration of notes being used
24          insert a tempo marker at into main rhythm part at the correct offset
25
26      insert_at = 0
27      for entry in section_data:
28          insert time_signature marker into main rhythm part with value entry.time_signature and offset of insert_at
29          increment value of insert_at based on entry.time_signature and entry.num_measures
30
31      insert main rhythm part into stream object
32      write the stream object a midi file
33
34      calculate indices of the notes to be accented
35
36      open midi file with mido library
37      get the midi track with the notes
38      loop through the messages of the track to get only note messages, i.e. messages with attribute velocity > 0
39      for note_message in note_messages:
40          if index of note_message in indices of accented notes:
41              increase velocity value to make accented note
42          else:
43              decrease velocity value
44
45      save changes to midi file
46
47      return midi file name

```

Figure 4.18: Pseudocode for the algorithms to make a MIDI file

I found that music21 was generally quite powerful and intuitive to use, but one problem which I came across was that changing the volume of notes did not seem to work, and this was crucial for adding accents to downbeats in the click track. My solution to this was to use another python library called Mido, which is created especially for manipulation of MIDI data, generally on a lower level than music21. Due to the lower level nature, using Mido requires some basic knowledge of how the MIDI format works. On a basic level, a MIDI file consists of tracks, which are themselves simply a series of instructions called messages. The messages are typically only 2-3 bytes long and consist of simple instructions such as NOTE ON or NOTE OFF along with basic data for properties like pitch and volume, which

in MIDI terminology are usually called note number and velocity respectively [56]. In this particular use case, we are interested in the velocity property of these messages, so once the messages corresponding to accented notes are located, it is simply a case of directly modifying the velocity property.

This is of course not an ideal solution, as a file is first written to by music21, then opened again by Mido before being modified and saved one more time, which is likely not good for performance.

To evaluate that the MIDI files were created correctly, they could be opened and examined in MuseScore. For click tracks with gradual tempo changes, it was possible to see all the discrete tempo markings as in Figure 4.21, essentially achieving the same results as Temmerman's [44] tempo change plugin discussed in Section 2.2. Polyrhythms were represented in the MIDI file by stretching the length of notes in the secondary rhythm so that they would all fit into a measure the same length as the primary rhythm, essentially recreating Ableton Live's time stretch feature discussed in section 2.2.2. For example, in a 5 against 4 polyrhythm, each note of the rhythm in 5 would have a length of 80% of a quarter note, determined by the ratio $4/5$. This is illustrated in Figure 4.19. Using the approach of lowest common multiple, also mentioned in Section 2.2.2, would work too, and the extra maths involved would not be a problem since it could all be calculated automatically, however the stretching approach was chosen as it results in more readable scores in applications such as MuseScore or Sibelius.



Figure 4.19: MIDI file generated generated from a click track with 5 against 4 polyrhythm, opened in MuseScore

The addition of polyrhythms and multiple instruments certainly added some complexity to the backend. Fluidsynth could only synthesise a MIDI file with one soundfont at a time, so constructing a click track with two samples from different soundfonts required generating a separate MIDI file for the notes played by each sample, then creating 2 separate audio files and combining them. This was done through the soundfile module, with the code shown in Figure 4.20, which reads the data from both audio files into NumPy arrays. These can then be overlaid

using the + operator. To play them sequentially, the concat operator would be used instead.

```

161 audio_data1, sample_rate = sf.read(f"part1.{file_format}")
162 audio_data2, sample_rate = sf.read(f"part2.{file_format}")
163
164 # May be a slightly different amount of silence at the end of each track, so need to trim down
165 # the longer one so that the np arrays can be added
166 shorter_length = min(len(audio_data1), len(audio_data2))
167 audio_data = audio_data1[:shorter_length] + audio_data2[:shorter_length]
168
169 sf.write(f"output.{file_format}", audio_data, sample_rate)

```

Figure 4.20: Code for overlaying two audio files

Another potential solution to generating tracks with 2 different samples would be to create a custom soundfont, as a different sample could be assigned to each note of the keyboard, not to mention multiple instruments, so this would be plenty of samples. This would require the use of a gui soundfont editor such as Polyphone [57], unless an effective way was found to generate a soundfont file automatically from a collection of samples.

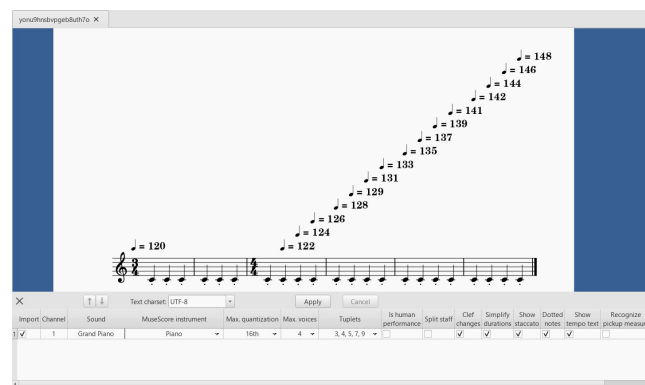


Figure 4.21: MIDI file generated from a click track with accelerando section

4.5.3 Audio file synthesis

Currently, data from the frontend is first converted to MIDI regardless of the requested file format, with the MIDI file then being used to synthesise audio. This was not always the case during the development of the application, as the first approach I tried was to generate wav files directly using the wavfile module of the ScyPy python library. This worked well for initial simple test cases such as 4 measures of a constant tempo, but problems quickly became apparent. The initial algorithm was very crude and did not scale well. The first major bug which occurred happened when notes were too close together. The duration of the sample itself was longer than the interval between the two click times, so the generate_silence function was given a negative value, causing the application to

crash. This was later fixed by allowing consecutive samples to "cut each other off", but was far from the end of problems with this approach.

Another attempted solution was to use an external API for converting from MIDI to audio file formats. While this did work fairly smoothly it lacked in flexibility, as it was impossible to specify a soundfont file, so the samples used for synthesis could not be controlled.

The better solution was to use fluidsynth, a command-line-based software synthesiser with many use cases such as playback and sequencing. The case relevant to this project was providing a MIDI file and a soundfont file, and getting an audio file as an output, in this case either a flac or a wav file. At first I tried using a python library called midi2audio, which acts as a lightweight wrapper for fluidsynth, but it did not have an option for controlling the volume of the output, which was coming out far too quiet. Thankfully it was not too difficult to simply execute fluidsynth directly from the python code using the subprocess module, using the `-g` flag to change the gain to 1 instead of its default value of 0.2. This was achieved using the command shown in Figure 4.22.

```
subprocess.run([
    "fluidsynth",
    "-ni",
    "-g",
    "1",
    instruments[0].soundfont_file,
    midi_filename,
    "-F",
    f"output.{file_format}",
])
```

Figure 4.22: Command used to generate an audio output file from a MIDI input file, using fluidsynth

4.5.4 Audio file formats

The three audio file formats available in the initial version of the application are explained below.

1. WAV - The wav format is uncompressed audio data, and therefore results in the large filesizes. It is the preferred choice for use when mixing and editing audio as all the original data is present [58].
2. FLAC - The flac format uses a lossless compression algorithm, resulting in

smaller filesizes with no irreversable loss of audio quality. It is popular among audiophiles and is used for the HD option in some streaming services [59].

3. OGG - The ogg format is an open source format that uses lossy compression, similar to mp3, to achieve much smaller filesizes than either wav or flac. Lossy compression does mean that some audio information is irreversibly lost, but this is rarely noticeable [60]. Ogg files are often used for browser-based playback, for instance Wikipedia uses ogg files when including audio samples in their articles.

4.5.5 File Upload

Once either the MIDI or audio file has been created, the final step is to upload it to Cloudinary and return the url of the uploaded file so that it can be displayed on the frontend. The function to achieve this is shown in Figure 4.23.

```
12 def upload_file(filename: str) -> str:
13     """Returns the https url of the file once it is uploaded to cloudinary"""
14
15     print("Uploading file", filename)
16     # Cloudinary considers audio to be a subset of the video resource type
17     upload_response = cloudinary.uploader.upload(
18         filename, resource_type="video", folder="clicktracks"
19     )
20
21     try:
22         return upload_response["secure_url"]
23     except:
24         return "error"
```

Figure 4.23: Function for uploading a file to Cloudinary

The user-created click tracks do not actually need to be stored persistently on Cloudinary, as it is just used to serve the file for download. Consequently, it was possible to save space by automatically deleting a newly uploaded click track after a predetermined period of time had elapsed. Thanks to this, running out of the space allocated to a free tier Cloudinary account was no longer a concern. The implementation for this feature was not completely trivial, as Flask does not run asynchronously by default, so simply calling `time.sleep(30)` followed by the code to delete the track would actually block the entire application for 30 seconds. Therefore, it was necessary to gain familiarity with the basics of the multiprocessing module for python, so that this delayed deletion functionality could be started in parallel as a separate process. The additions to the file uploading functionality are shown in Figure 4.24.

```

17 DELETE_TIMEOUT = 15 * 60 # Automatically deletes the track from cloudinary after 15 minutes
18
19 def delete_file_after_timeout(public_id):
20     time.sleep(DELETE_TIMEOUT)
21     cloudinary.uploader.destroy(public_id, resource_type = 'video')
22     log(f'Deleted {public_id}')
23     return
24
25 def upload_file(filename: str) -> str:
26     """Returns the https url of the file once it is uploaded to cloudinary"""
27
28     print("Uploading file", filename)
29     # Cloudinary considers audio to be a subset of the video resource type
30     upload_response = cloudinary.uploader.upload(
31         filename, resource_type="video", folder="clicktracks"
32     )
33
34     mp.Process(target=delete_file_after_timeout, args=(upload_response['public_id'],)).start()
35
36     try:
37         return upload_response["secure_url"]
38     except:
39         return "error"

```

Figure 4.24: Added functionality for deleting the uploaded file after 15 minutes

4.6 User Management Back-end

The user management backend was implemented after the user evaluation, as it had no impact on the core functionality of the click track editor.

4.6.1 Structure

As this is a backend with more traditional functionality such as CRUD, user authentication and interaction with a database, the Model View Controller (MVC) design pattern was chosen, with the directory structure shown in Figure 4.25.

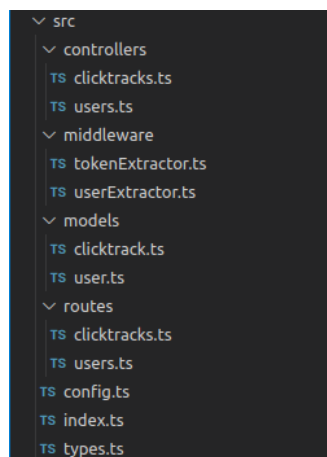


Figure 4.25: User management backend directory structure

Only two models were needed, one for users and one for click tracks. The software has been designed to be easily scalable though, so adding more models would be simple. For example in the future if we want the user to be able to practice along to a click track, and keep track of their performance, another model

called UserStats, for instance, could be added.

4.6.2 Authentication

Upon creating an account, a user's password is hashed using the bcrypt hashing algorithm with 10 salt rounds. The code for this is shown in Figure 4.26. As bcrypt salts passwords by default and can be slowed down deliberately by increasing the number of salt rounds, it is often recommended as the most secure method for hashing passwords to be stored in a database [61].

```
30 const register = async (username: string, name: string, password: string) => {
31   const saltRounds = 10;
32   // eslint-disable-next-line @typescript-eslint/no-unsafe-call
33   const passwordHash = await bcrypt.hash(password, saltRounds);
34   const newUser = new UserModel({
35     username,
36     name,
37     passwordHash
38   });
39
40   const savedUser = await newUser.save();
41   return savedUser;
42 };
```

Figure 4.26: Code for registering a user

When logging in, the entered password is compared with the password hash in the database using the `bcrypt.compare` function. If successful, the server returns a token which is associated with the user. This token can then be stored in the browser's local storage. For requests requiring authentication, the token is sent in the Authorization header, as shown in Figure 4.27. Back on the server, requests to protected routes must first pass through `tokenExtractor` and `userExtractor` middleware functions, shown in Figures 4.28 and 4.29, where the token is first extracted from the request, and then the server tries to decode it and associate it with an existing user. If either of these steps fails, an appropriate error response is returned.

```
6 const getConfig = () => {
7   {
8     headers: {
9       Authorization: 'bearer ' + window.localStorage.getItem('loggedInClicktrackUserToken'),
10    }
11  }
12 }
13
14 const getAll = async () => {
15   const response = await axios.get(`${baseUrl}/clicktracks`, getConfig())
16   return response.data
17 }
```

Figure 4.27: Code for sending the token in the Authorization header

```

src > middleware > TS tokenExtractor.ts > ...
1 import * as express from 'express';
2
3 const tokenExtractor = (req: express.Request, _res: express.Response, next: express.NextFunction) => {
4   const authorization = req.get('authorization');
5   if (authorization && authorization.toLowerCase().startsWith('bearer ')) {
6     req.token = authorization.substring(7);
7   }
8   next();
9 };
10
11 export default tokenExtractor;

```

Figure 4.28: tokenExtractor middleware

```

src > middleware > TS userExtractor.ts > ...
1 import * as express from 'express';
2 import config from '../config';
3 import jwt, { JwtPayload } from 'jsonwebtoken';
4 import UserModel from '../models/user';
5
6 const userExtractor = async (req: express.Request, res: express.Response, next: express.NextFunction) => {
7   try {
8     if (req.token) {
9       const decodedToken: JwtPayload = jwt.verify(req.token, config.SECRET) as JwtPayload;
10      if (!decodedToken.id) {
11        return res.status(401).json({ error: 'token missing or invalid' });
12      }
13      const user = await UserModel.findById(decodedToken.id);
14      if (!user) {
15        return res.status(404).json({ error: 'User not found' });
16      }
17      req.userId = decodedToken.id as string;
18      return next();
19    } else {
20      return res.status(401).json({
21        error: 'invalid token'
22      });
23    }
24    // eslint-disable-next-line @typescript-eslint/no-explicit-any
25  } catch (err: any) {
26    if (err.name === 'JsonWebTokenError') {
27      return res.status(401).json({
28        error: 'invalid token'
29      });
30    } else if (err.name === 'TokenExpiredError') {
31      return res.status(401).json({
32        error: 'token expired'
33      });
34    }
35  }
36  next();
37 };
38
39 export default userExtractor;

```

Figure 4.29: userExtractor middleware

4.6.3 TypeScript

To enable better error handling and more powerful autocomplete in the IDE [62], the server was written in TypeScript, with the models defined using the typegoose library. Thanks to the use of TypeScript interfaces and typegoose classes, shown in Figures 4.30 and 4.31, to create a type system, the server would throw an error upon receiving malformed or invalid data.

4.6.4 Changes to Frontend

Some additions were needed on the frontend in order for users to access the features offered from this backend. To start, the pure single page model was no longer suitable for the application, as there would need to be separate login and register pages as well as an index view for all of a user's saved click tracks.

```

src > TS types.ts > Section
1  interface OverallData {
2      numMeasures: number;
3      mtc: number;
4  }
5
6  interface Rhythm {
7      bpm: [number, number];
8      timeSig: [number, number];
9      accentedBeats: number[];
10 }
11
12 export interface Section {
13     overallData: OverallData;
14     rhythms: Rhythm[];
15     id: string;
16 }

```

Figure 4.30: TypeScript interface for click track section

```

6  class Clicktrack {
7      @prop()
8      public title!: string;
9
10     @prop()
11     public sections!: Section[];
12
13     @prop({ref: () => User})
14     public author!: Ref<User>;
15 }

```

Figure 4.31: Typegoose class for clicktrack model

All of the routes as well as the routing logic can be seen inside the project's base component `App.jsx`, shown in Figure 4.32. Therefore, client-side routing was implemented with the React Router library. Client-side routing is faster than server-side routing as an entire new page does not need to be downloaded when navigating to a different url. It also makes it easier to share state between pages [63]. A new directory entitled `pages` was created in the source code, to hold pages, which are still React components, but distinguished as the ones associated with each route.

The route `/myclicktracks`, shown in Figure 4.33, is used as an index view of all the click tracks belonging to the logged in user. When a user clicks on the edit button for one of these click tracks, they are taken to a route of the form `/myclicktracks/:id`, so that it is clear they are editing a specific click track.

More information needed to be stored in the applications global state tree. This was easy to accomplish thanks to the separation of the state into logical categories as discussed in Section 4.4.2. A new reducer for users was added, which shows up as a new branch on the tree visualisation shown in Figure 4.34.

```

src > App.jsx > ...
1 > import { useEffect } from 'react'
16
17
18 const App = () => {
19   const dispatch = useDispatch()
20   const user = useSelector(state => state.user.user)
21   const flash = useSelector(state => state.ui.flash)
22
23   useEffect(() => {
24     clicktrackService.startUp()
25     userService.ping()
26     // Try to get current user from local storage
27     if (window.localStorage.loggedInClicktrackUser) {
28       dispatch(setUser(JSON.parse(window.localStorage.loggedInClicktrackUser)))
29     }
30   }, [])
31
32   return (
33     <Router>
34       <Navbar />
35       {flash
36         ? <Flash message={flash.message} severity={flash.severity}/>
37         : null
38       }
39       <Routes>
40         <Route path="/" element={MainPage} />
41         <Route path="/register" element={RegisterPage} />
42         <Route path="/login" element={LoginPage} />
43         <Route path="/myclicktracks" element={MyClicktracks user={user}} />
44         <Route path="/myclicktracks/:id" element={MainPage} />
45       </Routes>
46     </Router>
47   )
48 }
49
50 export default App

```

Figure 4.32: App component with routing implemented

4.7 Deployment

Separating the frontend and backend allowed for many appealing options for deploying the frontend, since it was essentially a static site. These options included Netlify, Render and Vercel, among many others. Vercel was chosen for this project due to its ease of use and the availability of preview as well as production deployments.

One challenge when deploying the frontend was dynamically changing the backend url to that of the production backend. This was done using the `window.location.href` attribute in JavaScript, to check the current URL. Therefore the API url could be changed dynamically based on whether the production or development version of the frontend was being used. We can see the solution to both of these challenges illustrated in Figure 4.35.

Deploying the backend came with a few more challenges. I decided early on to deploy it to Heroku, due to its free tier and my familiarity with the platform. The main disadvantage of the free tier is cold starts, meaning the server takes a long time (3-4 seconds, sometimes even more) to respond if the application has not received any requests for 30 minutes. This is due to Heroku apps being run in lightweight Linux containers called dynos, which are powered down during periods of inactivity to save computing power [64]. Unsurprisingly, this has a negative effect on user experience. Studies have shown that increasing the load time of a site from one second to five seconds increases the probability of a user clicking away by 90% [48]. There exist tools to ping a Heroku application every 30 minutes,

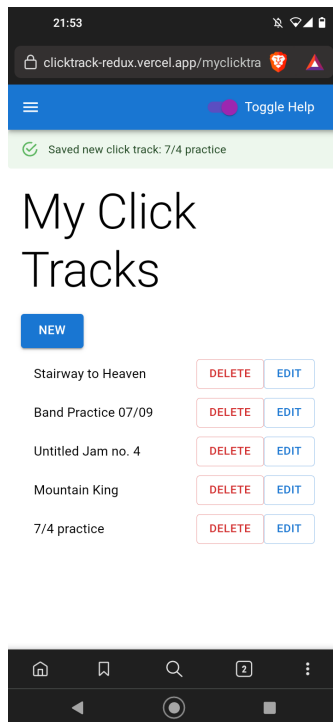


Figure 4.33: View of a user's clicktracks (on a mobile device).

but this will result in the limited number of hours given in the free tier being used up. Instead a better solution was found. Typically a user would spend some time creating, editing and previewing a click track, which is all handled by the frontend, before requesting a file to download. Therefore when the frontend web page is first opened, it sends a trivial HTTP get request to the server to "wake it up", so by the time a request for file generation is made, the server is fully up and running. This was the initial idea at least, but results were inconsistent, as the first file conversion still took quite long sometimes. Further research revealed that this was due to HTTP GET requests being cached, so the server was not receiving this initial "wake up" request. The solution to this was to change to using a POST request, as POST requests are not cached by default [65].

While a simple deployment of the Flask application to Heroku worked well at first, a major roadblock occurred when adding the functionality to synthesise wav and flac files from MIDI, as fluidsynth needed to be installed directly on the server, not just as a python dependency in `requirements.txt`. The trade off for Heroku's ease of use is that it does not offer this level of control over the servers, at least on the free tier. An alternative option would be to use a virtual private server (VPS), but this requires significantly more knowledge and work to configure properly. In the end, I decided to opt for a third approach which is increasingly popular in modern backend development, which is containerization.

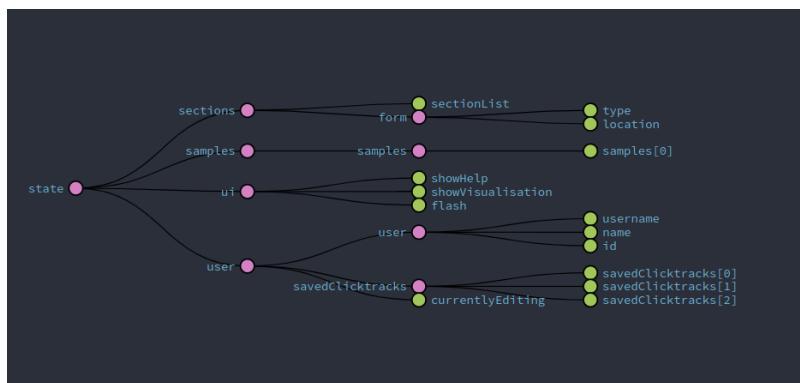


Figure 4.34: Visualisation of the application state after changes to the frontend

```

1 import axios from 'axios'
2 const baseUrl = window.location.href.includes('clicktrack-redux')
3   ? 'https://clicktrack-audio-backend.herokuapp.com/'
4   : 'http://127.0.0.1:5001'
5
6 // Sends a trivial request to the backend to prevent heroku from
7 // cold starting once a user requests a wav file
8 const startUp = () => {
9   try {
10     axios.get(baseUrl)
11   } catch (err) {
12     console.log('Server not running')
13   }
14 }
15
16 const getFile = async (data, fileExt) => {
17   const response = await axios.post(`${baseUrl}/api/make_${fileExt}`, data)
18   return response.data
19 }
20
21 export default { startUp, getFile }

```

Figure 4.35: Functionality for frontend interaction with the audio processing API

I used Docker as the containerization engine, allowing me to write a Dockerfile, essentially a set of instructions for how to set up a lightweight virtual machine, in this case with FluidSynth and python installed. The Dockerfile for this application is shown in Figure 4.36. This gave me full control of the Linux environment in which the application runs, while still taking advantage of Heroku's ease of use, as pushing to the Heroku container registry was quite straightforward, requiring only the commands shown in Figure 4.37.

Finally, the user management backend was a simple Heroku deployment, as the process for deploying NodeJS applications is well documented.

To enable communication between the frontend and the two backend services, cross origin resource sharing (CORS) needed to be enabled. For improved security, it is possible for a server to implement a whitelist of external origins which can request resources, as opposed to opening up the server to all external traffic. The logic for this in the Flask application is shown in Figure 4.38.


```

1 FROM ubuntu
2
3 #Install dependencies necessary for wav synthesis from midi and sound conversion
4 RUN apt-get update -y \
5     && apt-get install -y fluidsynth timidity libsndfile1 \
6     #Install pip for managing pythonpackages
7     python3-pip
8
9 WORKDIR /app
10
11 COPY ./requirements.txt requirements.txt
12
13 RUN pip install -r requirements.txt
14
15 COPY . .
16
17 EXPOSE 5000
18
19 CMD gunicorn wsgi:app

```

Figure 4.36: Dockerfile used for setting up the container environment for the python server

```

$ redeploy.sh
1 #!/bin/sh
2 heroku container:login
3 heroku container:push web --app clicktrack-audio-backend
4 heroku container:release web --app clicktrack-audio-backend

```

Figure 4.37: Shell script for redeploying the audio microservice

4.7.1 Note on the Heroku free tier

At the time of writing, Heroku has recently announced that they will be ending their free tier as of the 28th of November 2022 [66]. Had this been announced earlier, a different deployment platform would certainly have been chosen for the backend services of the application. As it stands, these services will continue to run on Heroku for the immediate future, but migrating them over to a different platform such as Fly.io, or Render is something I now intend to do after submission.

```

9  app = Flask(__name__)
10 CORS(app)
11 if os.environ["FLASK_ENV"] == "production":
12     CORS(app, origins=["https://clicktrack-redux.vercel.app"])

```

Figure 4.38: CORS whitelist implementation in Flask

5 Testing

5.1 End to End Testing

Given more time to complete the project, I would have liked to implement a comprehensive suite of unit tests. However due to the limited time frame of the project, end to end was the testing method of choice. End to end testing also has many advantages over unit testing, such as testing the interaction between multiple components and more closely resembling real life situations [67]. With strong end to end testing in place, I could have more confidence in the reliability of the application, and therefore focus the user survey on more subjective questions. Though end to end testing was implemented relatively late in the development of the application, it was immediately beneficial when adding the last few features, as I could quickly verify that changes did not break existing functionality without the need to manually test out all the interactions myself.

Cypress was chosen as the end to end testing library. It works by opening an automated browser window and performing interactions specified in a spec file to simulate a user interacting with the website. It is then easy to see which tests, if any, failed and why.

Since the application needs to be running locally to run Cypress tests on it, setting up the testing environment requires even more steps than setting up the development environment, so I extended the `dev.sh` script from Section 4.3 with another terminal window to open the cypress tests, as shown in Figure 5.1.

```

$ eZe.sh
1  #!/bin/bash
2  gnome-terminal \
3  | --tab --title="Audio Backend" -- bash -c "cd ../Advanced-Clicktrack-Audio-Backend; ./rebuild.sh; $SHELL"
4  gnome-terminal \
5  | --tab --title="User Backend" -- bash -c "cd ../Advanced-Clicktrack-User-Backend; npm run dev; $SHELL"
6  gnome-terminal \
7  | --tab --title="Frontend" -- bash -c "npm start; $SHELL"
8  gnome-terminal \
9  | --tab --title="Cypress" -- bash -c "cypress open; $SHELL"

```

Figure 5.1: Shell script to start the testing environment

To avoid one excessively large file for the tests, I decided to divide them into testing the ideal functionality, i.e. the general flow of a user interacting with the application as expected, and testing edge case, i.e. what if the user tries to enter

an empty value or a negative value where they are not supposed to? Testing the edge cases immediately raised some bugs to my attention. For example, when refactoring the forms to use MaterialUI components, I removed the `required` HTML attribute without noticing. This led to a user being able to clear the number inputs in the form for adding a new section and submit it, creating sections with length of 0 measures or tempo of 0 bpm, which was of course undesired. Without implementing end to end testing, this bug may have gone unnoticed for much longer.

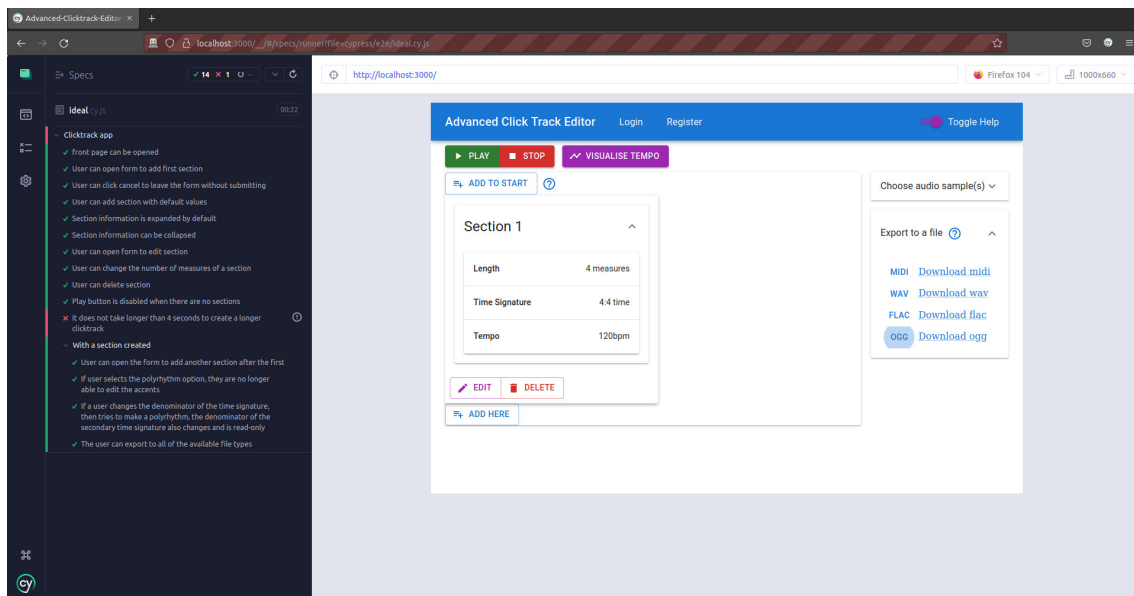


Figure 5.2: Cypress tests for ideal functionality

In Figure 5.2, we can see the full automated browser window running the aforementioned test file for ideal functionality. The one failing test is because the audio processing service still needs some optimisation when dealing with longer click tracks, so it takes longer than Cypress's default maximum response time of 4000ms. Figure 5.3 shows just the results of running the test suite for edge cases (after fixing the bugs which it revealed).

5.2 Performance analysis

The slow response time for longer click tracks was quite disappointing to see, so some further analysis was done to locate the performance bottleneck. There were three main areas of the code that I thought could be slowing the performance down.

1. Creation of the MIDI file

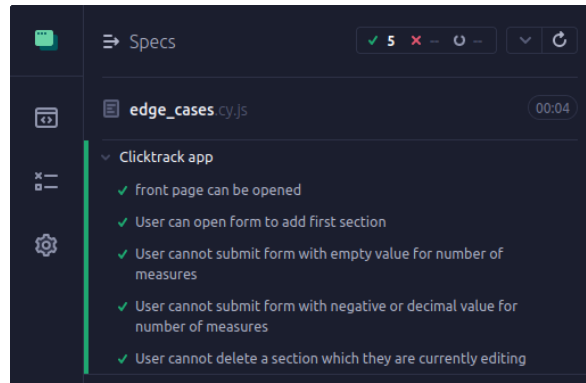


Figure 5.3: Cypress tests for edge cases

2. Synthesis of the audio file

3. Upload to Cloudinary

The first thing I wanted to find out was whether more time was taken in the creation of the MIDI file or in the synthesis of audio files. Since all audio files first require a MIDI file to be made in the first place, as described in Section 4.5.3, simply subtracting the time taken to create a MIDI file from the time taken to create and audio file would give a good estimate of how long the synthesis took. To this end, I used Cypress to automate the process of exporting 5 times to each file type, so that a more accurate average could be found of how long each file type takes to be created. The spec file for this is shown in Figure 5.4.

| Run | MIDI | Wav | Flac | Ogg |
|---------|--------|--------|--------|-------|
| 1 | 1.889 | 14.625 | 1.796 | 4.188 |
| 2 | 1.243 | 17.58 | 1.851 | 4.243 |
| 3 | 1.939 | 21.533 | 1.966 | 3.743 |
| 4 | 1.186 | 10.505 | 1.853 | 3.921 |
| 5 | 1.987 | 23.902 | 3.983 | 3.67 |
| Average | 1.6488 | 17.629 | 2.2898 | 3.953 |

Table 2: Comparison of time taken (in seconds) to generate a file of each format, for a click track of 20 measures

The results of the first test, shown in Table 2, were very eye-opening in terms of the difference between flac and wav files. While I knew that flac would always be faster due to its compression, the extent to which it was faster came as a shock. Making the assumption that synthesis time could be found by subtracting the MIDI creation time, we get an average time of 15.98 seconds for wav synthesis, compared to only 0.641 seconds for flac synthesis, so the wav file synthesis is

```

cypress > e2e > JS timing.cy.js > ...
1  describe('Timing Requests of the Click track app', function () {
2    beforeEach(function() {
3      cy.visit('http://localhost:3000')
4      cy.get('#add-to-start').click()
5      cy.get('.num-measures-input').clear().type('20')
6      cy.get('.section-form-submit').click()
7      cy.get('.expand-file-export').click()
8    })
9
10   //Run the test 5 times so results can be averaged
11   for (let i = 0; i < 5; i++) {
12     it('Time how long it takes to get a midi file', function () {
13       cy.get('.download-midi').click()
14       cy.contains('Download midi', { timeout: 60000 })
15     })
16
17     it('Time how long it takes to get a wav file', function () {
18       cy.get('.download-wav').click()
19       cy.contains('Download wav', { timeout: 60000 })
20     })
21
22     it('Time how long it takes to get a flac file', function () {
23       cy.get('.download-flac').click()
24       cy.contains('Download flac', { timeout: 60000 })
25     })
26
27     it('Time how long it takes to get an ogg file', function () {
28       cy.get('.download-ogg').click()
29       cy.contains('Download ogg', { timeout: 60000 })
30     })
31   }
32 })

```

Figure 5.4: Cypress spec file for timing the file creation of different file formats

nearly 25 times slower on average. This started to make more sense when I compared the file size of a flac and a wav file generated from the same 30 second click track. The flac file came to a size of 294kB, while the wav file was a much larger 5.3mB, or 18 times larger. In light of this, especially taking into account upload time, it makes sense that the requesting the wav file took 25 times longer. This large difference in filesize is itself still unexpected though, as multiple sources online [68], [69], quote the compression ratio for wav to flac as around 50%, so the wav file should only be twice as large, not 18 times. In the README file of his repository for the midi2audio python library, Zámečník [70] recommends exporting to flac over exporting to wav. Perhaps this could be because of an issue with fluidsynths's conversion to wav files specifically, although I could find no reference to this in any official documentation for fluidsynth.

While performing these tests, I stumbled upon another strange error. For click tracks with more than 20 measures, requesting an ogg file would simply cause the server to crash without logging any error messages whatsoever. I eventually tracked down this GitHub issue: <https://github.com/bastibe/python-soundfile/issues/130>, suggesting that this may be caused by a bug in the `soundfile` python module itself, which is being used to convert from flac to ogg.

Unfortunately due to time constraints these tests were carried out after deploy-

ing the initial version of the application for user evaluation. Surprisingly, none of the participants in the user evaluation survey seemed to notice these performance issues and bugs. This will be further discussed in Section 6. In the next release of the application, however I plan to at least temporarily remove the option for ogg export, and add a warning that wav export is very slow, with flac recommended instead. Additionally I will try to find ways to speed up the MIDI file creation too, as this is actually the main performance bottleneck for creation of audio files in formats with compression.

To gain more detailed insight I ran another performance test, in which only flac files were created for click tracks of increasing length, and I timed how much time was taken up by MIDI file processing, flac synthesis and uploading to Cloudinary. I increased the number of sections in intervals of 4 from 4 up to 100. Only the first half of the results, for the sake of brevity, is shown in Table 3. The upload times varied quite widely, but this is simply due to the weak and patchy WiFi network that the tests were run on, so these will be disregarded. The first time taken for flac synthesis should also be disregarded, as it is a very obvious anomaly. This is almost certainly due to the time taken to load fluidsynth on the server for the first time, as this test was run right after launching the server. Figure 5.5 shows a plot with the full dataset. The points for MIDI file creation fit best with a polynomial of order 2, meaning the algorithm has a time complexity of $O(n^2)$, while a linear fit was suitable for the flac file synthesis, corresponding to a much more acceptable time complexity of $O(n)$.

| Measures | MIDI | Flac | Upload | Total |
|----------|-------|-------|--------|-------|
| 4 | 0.04 | 2.846 | 1.42 | 4.306 |
| 8 | 0.083 | 0.113 | 1.822 | 2.018 |
| 12 | 0.152 | 0.218 | 1.272 | 1.642 |
| 16 | 0.278 | 0.214 | 2.557 | 3.049 |
| 20 | 0.35 | 0.313 | 2.888 | 3.551 |
| 24 | 0.462 | 0.312 | 2.134 | 2.908 |
| 28 | 0.633 | 0.413 | 2.89 | 3.936 |
| 32 | 0.709 | 0.413 | 3.436 | 4.558 |
| 36 | 0.972 | 0.413 | 3.269 | 4.654 |
| 40 | 1.116 | 0.513 | 2.431 | 4.06 |

Table 3: Time taken (in seconds) for various parts of flac file export, for click tracks of increasing length

$O(n^2)$ time complexity is of course something to be avoided if at all possible, so these findings necessitated a closer look at my function for making MIDI files.

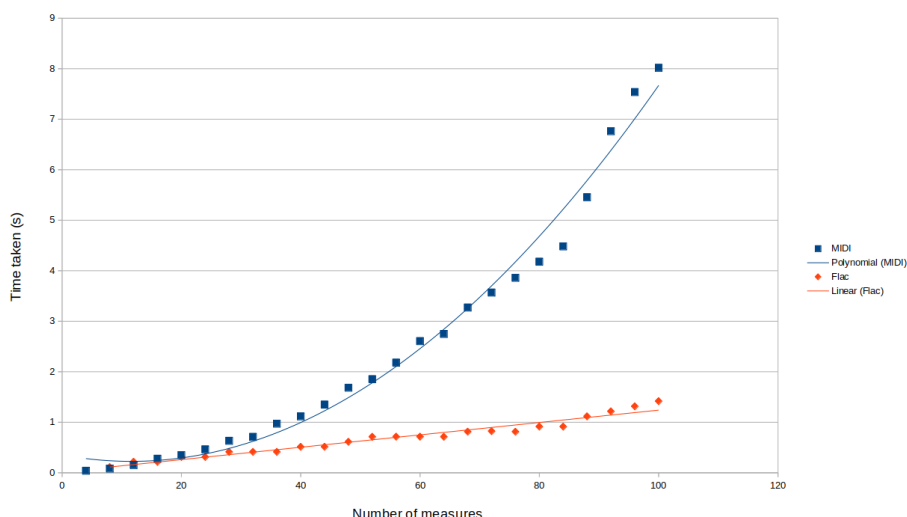


Figure 5.5: Comparison of time complexity of MIDI file creation and flac synthesis

Upon closer inspection, I hypothesised that it was the code shown in Figure 5.6 which was responsible for this. We can see that on line 68, the code loops through the list `note_bms`, which is linearly proportional to the length of the click track. Inside this loop, on line 71 of Figure 5.6, `music21`'s `offsetMap` method is called on the notes of the click track, which also essentially loops through them. Two nested loops through a list or array like structure is a typical example of an algorithm with $O(n^2)$ time complexity [71].

```

67 # Add tempo markers
68 for idx, bpm in enumerate(note_bms):
69     # Get the offset at which to insert the tempo marker, which can change depending on whether quarter,
70     # eighth, or half notes are being used
71     insert_at = main_rhythm_part.notes.offsetMap()[idx].offset
72     main_rhythm_part.insert(insert_at, tempo.MetronomeMark(number=bpm))

```

Figure 5.6: Code which is likely causing the quadratic time complexity

I will consider two possible options to resolve this issue. The first will be to rewrite the logic for constructing the MIDI file with `music21`, avoiding any nested loops. If this significantly improves performance then the second option will not need to be considered. If performance is still poor however, it may be beneficial to use `Mido` for the entire MIDI file creation process.

A new and improved algorithm for converting to MIDI was created. Instead of first adding all the notes and rests to the `music21` stream object and then looping through again to add time signature markers, tempo markers and accents, all of this information was instead added at the same time. This was achieved with a new helper function for making a section, shown in Figure 5.7. The same test of generating flac files of different lengths was run again, and we can see from Figure 5.8 that the performance of the algorithm has improved significantly, with

the time complexity now reduced to linear. Additionally, we can see that unlike in the previous test, the time taken to synthesise the first click track to flac is not abnormally high. The server was already running for this test, justifying my hypothesis that the anomaly in the first test was in fact due to the initial load time for fluidsynth. Finally, I was able to find a way to adjust note volumes with music21, which was an issue in the first edition of the MIDI creation algorithm (see Section 4.5.2), so Mido could be removed entirely as a dependency, making the application more lightweight overall.

```

74 def make_section_v2(
75     num_notes_before: int,
76     time_sig: List[int],
77     num_measures: int,
78     accented_notes: List[int],
79     note_pitch: str,
80     tempo_dict: dict,
81 ) -> list:
82     quarter_length = 4 / time_sig[1]
83     #Start with a time signature marker
84     numerator = time_sig[0]
85     denominator = time_sig[1]
86     notes_so_far = num_notes_before + time_sig[0] * num_measures
87     result = [meter.TimeSignature(f"{numerator}/{denominator}")]
88     for i in range(time_sig[0] * num_measures):
89         #Check if there is a tempo marker to add
90         if num_notes_before + i in tempo_dict.keys():
91             result.append(tempo.MetronomeMark(number=tempo_dict[num_notes_before + i]))
92         is_accented = i % time_sig[0] in accented_notes
93         click_note = note.Note(note_pitch, quarterLength=0.5*quarter_length)
94         click_note.volume = 120 if is_accented else 80
95         result.extend([
96             click_note,
97             note.Rest(quarterLength=0.5*quarter_length),
98         ])
99     return notes_so_far, result

```

Figure 5.7: New helper function for translating a section to music21 objects

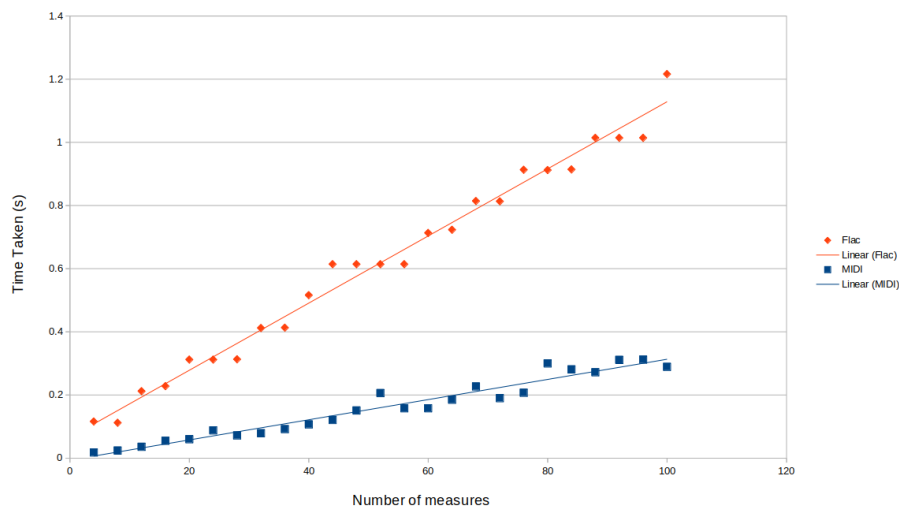


Figure 5.8: Comparison of time complexity of MIDI file creation and flac synthesis with the new and improved MIDI creation algorithm

6 User Evaluation

A survey with a link to the deployed web application was sent out to university mailing lists in order for users to evaluate the application. The questionnaire, recruitment email and participant information sheet can all be found in Section 9.2 of the appendix. The first question of the survey was simply to confirm that participants were 18 or older, so we will examine the responses to question 2 and onwards here.

Firstly, we can see from the answers shown in Figure 6.1 that 14/15 of the users surveyed had played music with a metronome or click track before. Likewise, 14 of them play music in which the tempo changes at least rarely, with the most common answer being "sometimes". This shows that the sample is fairly typical of hobby musicians, which are the main target audience of the application. To gain more insight from musicians who play music with tempo changes often, the user survey could be promoted in forums for specific genres like progressive rock, math rock or contemporary classical music, which typically employ a lot of rhythmic intricacy.

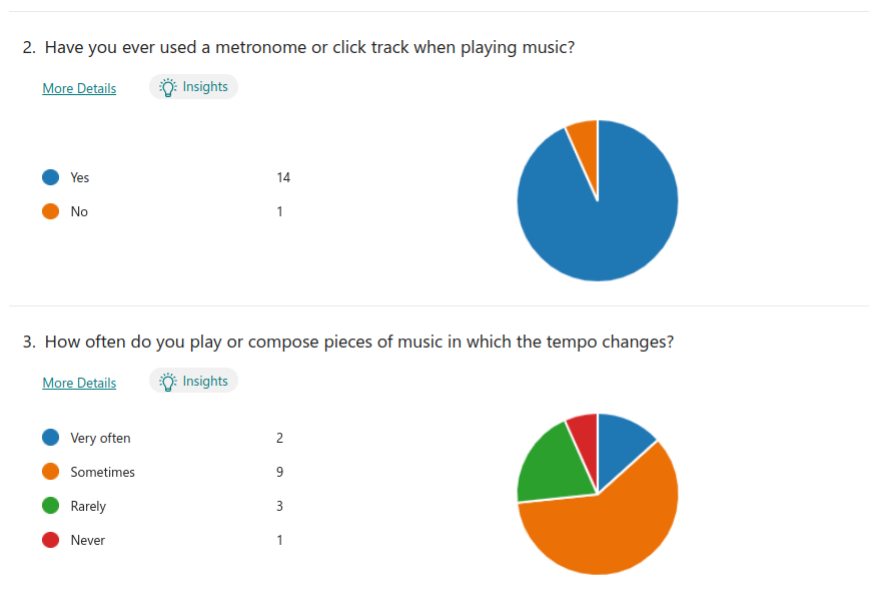


Figure 6.1: Responses to questions 2 and 3 of the user survey

In question 4, participants were required to choose from 5 qualifiers to describe how easily they could figure out how the application worked. We can see from Figure 6.2 that 10/15 participants selected "fairly easy", the second best option, while only one participant selected "difficult" and no participants selected "very difficult". This is quite satisfactory for a first round of user testing. In subsequent

questions which ask about improvements to the application, special attention will be paid to the answers of the two participants who answered "difficult" and "neutral".

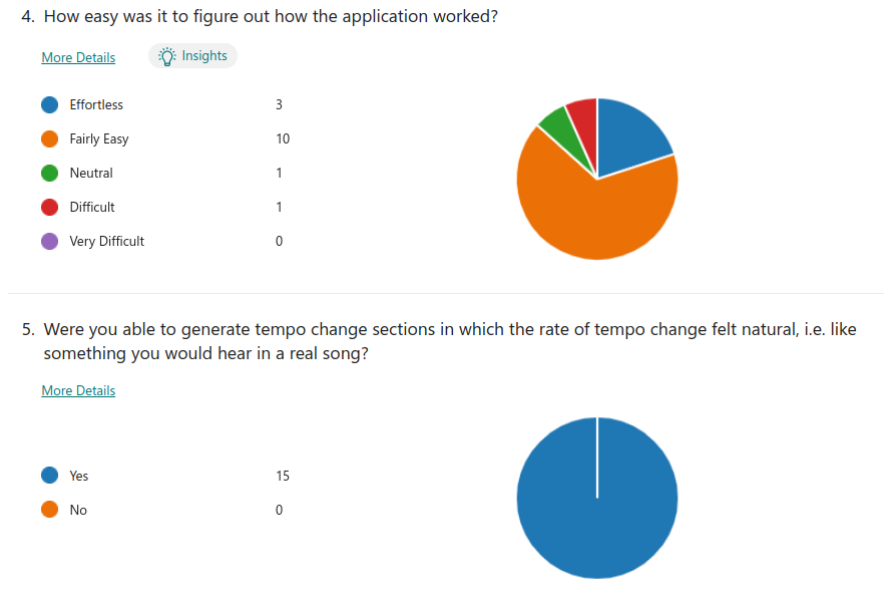


Figure 6.2: Responses to questions 4 and 5 of the user survey

The results for question 5 came as a pleasant surprise, as 100% of participants found that they could generate tempo changes that sounded natural. I feared that maybe some of the more experienced musicians would have answered negatively to this question, as the simple mean tempo condition approach meant the tempo curves did not have a smooth start and end. Feldman et al [24] claimed that this was needed for a natural sounding tempo transition, as discussed in Section 2.1.2. Perhaps a more open-ended format than a yes or no question could have worked better here, as a participant who was mostly satisfied with the tempo curves but still found some minor flaws would likely still answer yes.

In question 6, shown in Figure 6.3, users were given a list of potential uses for the application and asked to select any that they thought would be relevant to them. The most common use was to create click tracks for practicing, which is not surprising, as almost all musicians would regularly practice their instrument, while not all of them would be in bands or compose original music. The second most popular use case was for recording original music, and this was in fact a situation I found myself in that gave me the idea to develop this application in the first place. Comparatively less participants were interested in using the application for recording covers, likely because they could simply play along to the original recording to account for any potential tempo changes.

6. Would you use this application for any of the following?

[More Details](#)

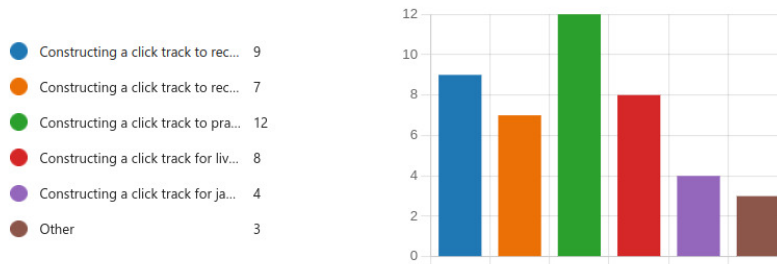


Figure 6.3: Responses to question 6 of the user survey

In question 7, users could optionally give any other use cases that were not included as an option in question 6. The four responses to this question are shown in Table 4. Interestingly participantss 1 and 3 both found that the application could be used to help come up with ideas for compositions, an idea I had not considered before. Participant 4 would use the application for rehearsals and warm ups for drumming. It would make sense to perhaps use a very gradually accelerating click track for a warm up session. Finally, while the tone of participant 2's response may not be entirely serious, it still contains some insight, that users will often find a native mobile app more convenient than a web application. In Section 7.3, a plan for creating a native version of the application is discussed.

| ID | Name | Response |
|----|-----------|---|
| 1 | anonymous | Interesting polyrhythms can be created easily with the app so it could be used to construct rhythmic patterns for compositions. |
| 2 | anonymous | Nah I'd just use a metronome app on my phone innit. It's more convenient, the click sounds phat and it's paired with a tuner. |
| 3 | anonymous | Unexpectedly, I found the tool encouraged me to think more deeply about the possibilities of changing / combining time signatures, and in turn this produced new creative ideas. I think it would be useful as a quick way to prototype the flow of a new composition (or a section thereof), which may in turn highlight otherwise overlooked possibilities. |
| 4 | anonymous | I play drums and use a metronome for rehearsal/warm up. |

Table 4: Other use cases suggested by survey participants

The last two questions of the survey were open-ended questions. In question 8,

participants were asked how they would improve existing features of the application. The responses to question 8 are shown in Table 5. Participant 2 is the one who rated the application as difficult to use, and from their response to this question it seems like a lot of the difficulty came from not having a reset button, as they had to resort to reloading the page to reset the click track. Additionally some more clear feedback would help when adding sections. Participant 3 also had problems to do with resetting the page, as well as an error from deleting sections that I was unable to reproduce. Participant 8 wanted to edit the structure of the click track in a more visual way, likely closer to how it would be done in a DAW, as they mentioned ProTools and Cubase sepcifically. This would require quite a bit of extra logic and restructuring for the application, but perhaps a more achievable feature would be drag and drop functionality for rearranging the sections. The participant who previously answered that they would use the application for their drumming warm up suggested adding the ability to define the length of sections using minutes and seconds instead of measures, which is logical for a warm up routine. This should not be too hard to implement, since time in seconds is already calculated in order to schedule audio events for playback. Finally, adjusting the volume of the click, suggested by participant 4, would certainly be a helpful addition to the application. Another user made a similar but more detailed suggestion as an answer to the next question, so it will be discussed further below.

In question 9, participants were asked which new feature they would most like to see added to the application. Responses are shown in Table 6.

The list below shows the order in which I would add these new features, starting with the ones that would be more easily implemented or require less restructuring of the existing codebase, ending with the more ambitious features.

1. Proper reset facility - This would just require a button to be added which clears the state of the click track being edited.
2. Example video - An example video could be uploaded to YouTube and linked with a url on the main page of the application.
3. Swing time - would require an extra input on the `SectionForm` component, as well as some extra logic for constructing the click tracks on the frontend and the audio processing microservice. I would likely take inspiration from MuseScore's implementation of swing rhythms, shown in Figure 6.4. For reference, swing time describes the technique of playing pairs of consecutive notes with unequal durations. In jazz and blues music this is quite common and usually the ratio of durations is close to 2:1 [72]. We can see that this

ratio is represented as a percentage in MuseScore, so their default value of 60% is logical. It would be important to be able to adjust this value though to allow for more interesting and experimental rhythmic possibilities.

4. Track volume sliders for main rhythm track and polyrhythm track - This would likely require using a numerical representation for click volume instead of a boolean choice of accented or unaccented. Making this change would open up a lot of other possibilities to do with manipulating volume so this would definitely be a good addition.
5. Popup tutorial and example - could add logic to the state management for an example click track. The popup tutorial would require more components on the frontend but not much added logic or calculation.
6. Ability for users to upload their own samples - Would be fairly easy to implement on a basic level, but difficult to ensure security and reliability. Would need robust validation to make sure the uploaded files are of the correct format and length.
7. Display progress on tempo visualisation graph - definitely achievable but not as easy as it might sound at first. This is because events on the WebAudio API run on a separate thread, so yet another timing system, the requestAnimationFrame API, is needed to accurately sync visual feedback with audio events [53].
8. Ability to import audio and MIDI files - Would require implementing MIDI libraries and audio processing functionality on the frontend, and possibly some form of beat detection algorithm if the user wanted to generate the click track automatically from the imported file.
9. Real time control of the bpm for live performances - would require a rethink in the structure of the frontend code, as currently all audio events are scheduled in advance before playing the click track. Scheduling events while playing is likely the better option anyways, as trying to play very long tracks in the browser in the current version of the application results in some lag due to scheduling all the audio events at once.

It is also worth considering what the participants did not request. For instance, none of the participants requested the addition of user accounts and saving tracks. This is perhaps indicative of a certain fatigue with constantly creating accounts that users of the modern web experience. It could also just be because creating and

editing a click track is generally a fairly quick task that the users see themselves doing in one sitting. For more advanced tasks, the users may likely turn to a DAW. With this information in mind, in future development I will shift my focus away from the user account features, and prioritise adding functionality to the editor itself. Perhaps later on, when more complex and interesting rhythmic ideas can be achieved with the editor, users will then find themselves wanting to save or perhaps even share their click tracks and then the user account features will become more important

Another surprise is that none of the users brought up the relative slowness of file export for longer click tracks. This is likely due to the simple and short nature of the application testing. In an effort to convince more participants to take part, the invitation email stated that testing out the click track would only take 10 minutes, so most participants probably just made short and simple click tracks.

Running an in person workshop where participants could sit down for a longer time would surely give more insightful results, and allow them to more fully explore the functionality as well as flaws of the application. This would of course make it harder to get a good number of participants, so more time would be needed for recruitment, along with potentially offering some form of compensation.

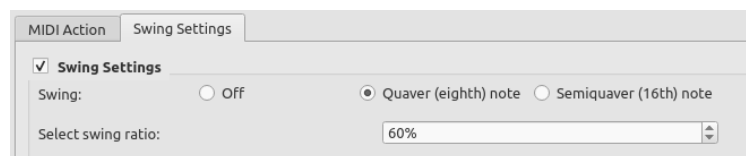


Figure 6.4: Swing implementation in MuseScore

| ID | Name | Responses |
|----|-----------|---|
| 1 | anonymous | No |
| 2 | anonymous | After clicking 'Add to Start' for the first time, choosing parameters and then 'Add this Section', it wasn't quite clear to me whether I had to do anything more to add the section afterwards. Perhaps it would have been clearer to me if the section thus created had been displayed in a more compact form. (I worked it out eventually.) It wasn't clear to me how to reset and start again with a completely empty click track. (<Ctrl>-R worked :-) |
| 3 | anonymous | When I deleted my rhythms, the whole page disappeared, but I figured out that I just had to refresh the feed if I wanted to start again. However, I think it would be helpful if the page stayed without having to refresh the feed. |
| 4 | anonymous | Phatter click. allow to adjust volume of the click. |
| 5 | anonymous | Be able to submit user's suggestions |
| 6 | anonymous | It would be nice to have changes made more dynamically, although this is maybe not a common requirement for metronomes. I just felt like it'd be good to have a slider or something? |
| 7 | anonymous | None |
| 8 | anonymous | The app is very good as it stands, but I did find I wanted to edit the structure visually (although the section dialogs were excellently laid out and easy to work with). Perhaps the click track itself could be displayed as a "tape track" (like ProTools, Cubase etc.), with sections added by clicking/dragging, and edited by selecting them? This might be a quicker way to generate such tracks, and enable the visualisation to be a bigger part of the process. |
| 9 | anonymous | would be handy to be able to add sections using minutes/seconds for length instead of measures. IE If I wanted to program a warm up routine something like 150 for 5 minutes then 160 for 5 minutes etc. |

Table 5: Responses to question 8 of the user survey

| ID | Name | Responses |
|----|-----------|--|
| 1 | anonymous | Track volume sliders for the main rhythm track and the polyrhythm track. |
| 2 | anonymous | Maybe a proper reset facility? |
| 3 | anonymous | See above |
| 4 | anonymous | A short popup tutorial in addition to the help buttons and maybe an example that you can play around with |
| 5 | anonymous | An example video of how to use new added features |
| 6 | anonymous | Add swing time or other similar metronomic variations? |
| 7 | anonymous | Upload your own sample! |
| 8 | anonymous | n/a |
| 9 | anonymous | The opportunity to import a WAV / MIDI file would be good to see; if creating a click track for an existing piece, it would be essential to be able to align it with that piece directly. Also, perhaps some kind of controller could be used to control the master tempo; that could enable the separation of BPM from arrangement, and allow a live musician to "tap" the actual tempo for a more fluid use of the software. Some coupling could be retained, perhaps to allow distinct tempo changes to be driven by the app (and only by the musician once established), and maybe to "dampen" any controller-driven change (for example, uneven tapping by the musician themselves). If developed as a plugin etc., the ability to start and synchronise the click track by timecode could be desirable, so it could run in parallel with an existing arrangement or application. |
| 10 | anonymous | Display progress on Tempo Visualization graph so I can easily see where I am in the pattern and when the next change is coming etc. |

Table 6: Responses to question 9 of the user survey

7 Future Improvements

Besides the features requested by participants in the user evaluation survey discussed in Section 6, some interesting potential features and improvements to the application are outlined below.

7.1 Audio processing in the browser

With recent additions to browser based audio functionality, it would actually be possible to do all the audio processing on the client side, eliminating the need for a separate python backend. *Signal*, the web based sequencer mentioned in Section 2.2, uses the Web MIDI API, for example. However, if I were to expand the application to include more advanced symbolic musical analysis in the future, it would still be beneficial to take advantage of the music21 library on the server.

7.2 Beat Detection

Currently the application works well for planning out click tracks, but there is the potential to do much more and tackle the more challenging problem of beat detection. Beat detection will likely require analysis and manipulation of raw audio data in the form of n-dimensional arrays, as well as frequency analysis through functions such as fast fourier transforms. NumPy is a python library well suited to these tasks, and is already installed on the python server as a dependency of music21, so the current application is a good starting point to implement this feature. There are still many challenges in this field, as humans can intuitively detect beats much of the time but it is hard to formalise exact conditions for when a beat occurs, especially in music without percussive instruments. This makes it a very promising field to apply machine learning to, and many researchers are attempting this currently.

7.3 Native Mobile Application

Finally, the experience for mobile users could be improved by the development of a native mobile application. A natural choice for this would be to use React Native, which would minimise the amount of changes need for the frontend code. Since the user management backend and the audio processing microservice are both deployed separately and decoupled from the frontend, they could remain unchanged.

7.4 Container Optimisation

One sub-optimal aspect of the project is the size of the docker image used for the backend. Typically, lightweight specialized container images are used for hosting web applications, such as alpine or python:slim [73], however I had difficulty getting all dependencies working on these images, so chose a basic Ubuntu image which ended up being around 1.4GB in size. For the moment it has no noticeable effect on the project, but if the need arose to scale up, this could quickly make hosting needlessly costly.

8 Reflection

The development of this application was a very instructive experience. One thing I learned was the difficulty of debugging as a codebase grows. The use of TypeScript for the user management backend certainly helped in this regard, and given the opportunity to restart this project from scratch I would use TypeScript for the frontend, where it would be even more beneficial simply due to the scale and complexity of the frontend.

Though I have plenty of experience playing music, as well as some programming experience, programming with audio was something new for me, and presented many challenges. On a fundamental level, the element of timing was the biggest obstacle. In my previous, mostly web-based, programming experience, the only concern with timing was to make everything as fast as possible, whereas in audio programming events need to be scheduled at specific times in the future. Additionally, slight variations in timing can be very noticeable to a user. I am aware that I have barely scratched the surface in terms of all the techniques and solutions present in the field of audio programming. Another challenge with working with audio, or more generally multimedia such as images and videos as well, was the increased need for optimisation. Since audio data, especially in uncompressed formats such as wav is quite large in terms of file size, badly optimised code can quickly slow down an application to unacceptable levels. This has helped me to appreciate why a lot of lower level audio code needs to be written in faster languages like C and C++. In most of my previous programming experience with text and numerical data, optimisation was not as pressing a concern.

A particularly interesting outcome of working on this project was finding a realistic opportunity to meaningfully contribute to an open source project, which was always an intimidating prospect for me before, looking at large and complicated codebases on GitHub. As mentioned in Section 4.5.3, the `midi2audio` library lacks the option to control the volume of the output, so I decided to extend the library with this functionality and make a pull request. The pull request, which at the time of writing is still awaiting approval, can be found at <https://github.com/bzamecnik/midi2audio/pull/9>.

References

- [1] *What is musical entrainment?* 2022. URL: <https://musicscience.net/projects/timing/iemp/what-is-musical-entrainment/>.
- [2] Trevor McPherson et al. "Intrinsic Rhythmicity Predicts Synchronization-Continuation Entrainment Performance". In: *Scientific Reports* 8.1 (2018). DOI: 10.1038/s41598-018-29267-z. URL: https://www.researchgate.net/publication/326851075_Intrinsic_Rhythmicity_Predicts_Synchronization-Continuation_Entrainment_Performance.
- [3] Shaun Letang. *What Is A Metronome, What Is A Metronome Used For, & Your Other Related Questions Answered - Music Industry How To*. 2022. URL: <https://www.musicindustryhowto.com/what-is-a-metronome-what-is-a-metronome-used-for-your-other-related-questions-answered/>.
- [4] Morris Wright. *What is A Click Track?* 2022. URL: <https://www.thedrummerguide.com/what-is-a-click-track/>.
- [5] Will Brook-Jones. *What is a DAW?* 2021. URL: <https://blog.andertons.co.uk/learn/what-is-a-daw>.
- [6] Anders Reuter. "Who let the DAWs Out? The Digital in a New Generation of the Digital Audio Workstation". In: *Popular Music and Society* 45.2 (2021), pp. 113–128. DOI: 10.1080/03007766.2021.1972701.
- [7] Nick Cesarz. *Click Tracks: Are They Useful or Just a Crutch?* 2019. URL: <https://drummingreview.com/click-tracks/>.
- [8] Sidney Prim. *Tempo Rubato - Definition and why you should care how its used!* 2017. URL: <https://www.libertyparkmusic.com/tempo-rubato/>.
- [9] James Beament. *How we hear music*. The Boydell Press, 2005.
- [10] Paul Lamere. *In search of the click track*. 2009. URL: <https://musicmachinery.com/2009/03/02/in-search-of-the-click-track/>.
- [11] *Italian musical terms*. 2022. URL: <https://www.musicca.com/musical-terms>.
- [12] *50 Greats for the Piano*. Yamaha, 2000.
- [13] West Troiano. *What is 6/8 Time Signature? | Liberty Park Music*. 2021. URL: <https://www.libertyparkmusic.com/what-is-6-8-time-signature/>.

- [14] Samuel Hunt. *Exploring Polyrythms, Polymeters, and Polytempi with the Universal Grid Sequencer framework*. Creative Technologies Laboratory. 2020, pp. 2–3. URL: <https://uwe-repository.worktribe.com/OutputFile/6829353>.
- [15] Bruno H. Repp. “Probing the cognitive representation of musical time: Structural constraints on the perception of timing perturbations”. In: *Cognition* 44.3 (1992), pp. 241–281. DOI: [10.1016/0010-0277\(92\)90003-z](https://doi.org/10.1016/0010-0277(92)90003-z). URL: <https://www.sciencedirect.com/science/article/pii/001002779290003Z>.
- [16] Maija Hausen et al. “Music and speech prosody: a common rhythm”. In: *Frontiers in Psychology* 4 (2013). DOI: [10.3389/fpsyg.2013.00566](https://doi.org/10.3389/fpsyg.2013.00566).
- [17] Erin E. Hannon. “Perceiving speech rhythm in music: Listeners classify instrumental songs according to language of origin”. In: *Cognition* 111.3 (2009), pp. 403–409. DOI: [10.1016/j.cognition.2009.03.003](https://doi.org/10.1016/j.cognition.2009.03.003).
- [18] Hendrik Schreiber, Frank Zalkow, and Meinard Müller. “Modeling and Estimating Local Tempo: a Case Study on Chopin’s Mazurkas”. In: *21st International Society for Music Information Retrieval*. International Audio Laboratories, 2020. URL: https://www.researchgate.net/publication/345672488_Modeling_and_Estimating_Local_Tempo_a_Case_Study_on_Chopin's_Mazurkas.
- [19] Maja Trochimczyk. *Mazur (Mazurka) - Polish Music Center*. 2018. URL: <https://polishmusic.usc.edu/research/dances/mazur/>.
- [20] Axel Berndt. “Musical Tempo Curves”. In: *International Computer Music Conference*. University of Huddersfield, 2011. URL: https://www.researchgate.net/publication/228844587_Musical_Tempo_Curves.
- [21] Thomas E. Cope, Manon Grube, and Timothy D. Griffiths. “Temporal predictions based on a gradual change in tempo”. In: *The Journal of the Acoustical Society of America* 131.5 (2012), pp. 4013–4022. DOI: [10.1121/1.3699266](https://doi.org/10.1121/1.3699266). URL: <https://asa.scitation.org/doi/10.1121/1.3699266>.
- [22] Hans-Henning Schulze, Andreas Cordes, and Dirk Vorberg. “Keeping Synchrony While Tempo Changes: Accelerando and Ritardando”. In: *Music Perception* 22.3 (2005), pp. 461–477. DOI: [10.1525/mp.2005.22.3.461](https://doi.org/10.1525/mp.2005.22.3.461). URL: <https://www.jstor.org/stable/10.1525/mp.2005.22.3.461>.

- [23] Anders Friberg and Johan Sundberg. “Does music performance allude to locomotion? A model of final ritardandi derived from measurements of stopping runners”. In: *The Journal of the Acoustical Society of America* 105.3 (1999), pp. 1469–1484. DOI: [10.1121/1.426687](https://doi.org/10.1121/1.426687). URL: <https://doi.org/10.1121/1.426687>.
- [24] Jacob Feldman, David Epstein, and Whitman Richards. “Force Dynamics of Tempo Change in Music”. In: *Music Perception* 10.2 (1992), pp. 185–203. DOI: [10.2307/40285606](https://www.jstor.org/stable/40285606). URL: <https://www.jstor.org/stable/40285606>.
- [25] Henkjan Honing. “The Final Ritard: On Music, Motion, and Kinematic Models”. In: *Computer Music Journal* 27.3 (2003), pp. 66–72. DOI: [10.1162/014892603322482538](https://www.jstor.org/stable/3681802). URL: <https://www.jstor.org/stable/3681802>.
- [26] Cecilie Møller et al. “Beat perception in polyrhythms: Time is structured in binary units”. In: *PLOS ONE* 16.8 (2021), pp. 1–24. DOI: [10.1371/journal.pone.0252174](https://doi.org/10.1371/journal.pone.0252174).
- [27] Deanna M. Kennedy, Jason B. Boyle, and Charles H. Shea. “The role of auditory and visual models in the production of bimanual tapping patterns”. In: *Experimental Brain Research* 224.4 (2012), pp. 507–518. DOI: [10.1007/s00221-012-3326-y](https://doi.org/10.1007/s00221-012-3326-y).
- [28] Michael J. Hove et al. “Synchronizing with auditory and visual rhythms: An fMRI assessment of modality differences and modality appropriateness”. In: *NeuroImage* 67 (2013), pp. 313–321. DOI: [10.1016/j.neuroimage.2012.11.032](https://doi.org/10.1016/j.neuroimage.2012.11.032).
- [29] Charles Roberts et al. “Designing Musical Instruments for the Browser”. In: *Computer Music Journal* 39.1 (2015), pp. 27–40. DOI: [10.1162/comj_a_00283](https://www.jstor.org/stable/24265496). URL: <https://www.jstor.org/stable/24265496>.
- [30] Yotam Mann. *Tone.js*. 2022. URL: <https://yotammann.info/tone>.
- [31] Michael Kolesidis. *JSS-01 | JavaScript Software Synthesizer*. 2022.
- [32] Ryohei Kameyama. *signal - Fully Open-sourced Online MIDI Editor*. 2021.
- [33] Ryohei Kameyama. *I spent five years building a web app, posted it on Hacker News, and got my first \$1*. 2022. URL: <https://codingcafe.jp/posts/signal-5yrs>.
- [34] Mari Lesteberg. “Micro and Macro: Developing New Accessible Musicking Technologies”. PhD thesis. University of Oslo, 2021.
- [35] Dylan Schiemann. *Howler.js Audio Library for the Modern Web*. 2018. URL: <https://www.infoq.com/news/2018/11/howlerjs-audio-modern-web/>.

- [36] Arek Nawo. *9 libraries to kickstart your Web Audio stuff*. 2019. URL: <https://areknawo.com/10-libraries-for-web-audio-stuff/>.
- [37] Stephane Pigeon. *Polyrhythm Pattern Generator - Design Your Own*. 2013. URL: <https://mynoise.net/NoiseMachines/polyrhythmBeatGenerator.php>.
- [38] George Peterson. "DAWS for Pro Audio Applications". In: *The Mix (Berkeley, Calif.)* 34.10 (2010), pp. 20–25. ISSN: 0164-9957.
- [39] *3.3 Toolbar - User Manual*. 2022. URL: <https://docs.lmms.io/user-manual/3-navigating-lmms/3.4#3.3.3-tempo-control>.
- [40] Jeff Kaiser. *9: Creating Complex Click Tracks - Two-Minute (or so) Tutorials for Reaper DAW*. 2021. URL: <https://whyreaper.com/tutorial/two-minute-or-so-tutorials-for-reaper-daw-9-creating-complex-click-tracks/>.
- [41] *gradual speedup audio*. 2014. URL: <https://forums.steinberg.net/t/gradual-speedup-audio/638016>.
- [42] Leticia Trandafir. *How to Use Polyrhythms to Create a Perfect Beat*. 2017. URL: <https://flypaper.soundfly.com/produce/how-to-use-polyrhythms-to-create-a-perfect-beat/>.
- [43] Rory PQ. *How to warp tracks in Ableton Live quickly*. 2019. URL: <https://iconcollective.edu/warp-tracks-in-ableton-live/>.
- [44] Johan Temmerman. *Musescore TempoChanges: How does it work?* 2021. URL: https://jeetee.github.io/MuseScore_TempoChanges/.
- [45] Moises Systems, Inc., 2022.
- [46] Mina Mounir, Peter Karsmakers, and Toon van Waterschoot. "Musical note onset detection based on a spectral sparsity measure". In: *EURASIP Journal on Audio, Speech, and Music Processing* 2021.1 (2021). DOI: [10.1186/s13636-021-00214-7](https://doi.org/10.1186/s13636-021-00214-7). URL: <https://doi.org/10.1186/s13636-021-00214-7>.
- [47] Radimir Bitsov. *Time to Interactive: Focusing on the Human-Centric Metrics*. 2018. URL: <https://calibreapp.com/blog/time-to-interactive>.
- [48] Nick Galov. *21+ Website Load Time Statistics and Facts for 2022*. 2022. URL: <https://webtribunal.net/blog/how-speed-affects-website/>.

- [49] Joe Nemer. *Advantages and Disadvantages of Microservices Architecture*. 2019. URL: <https://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback/>.
- [50] Sam Richard and Pete LePage. *What makes a good Progressive Web App?* 2022. URL: <https://web.dev/pwa-checklist/>.
- [51] Lawrence Wagerfield. *Cloudinary vs AWS S3 - Are they really comparable?* 2022. URL: <https://upload.io/blog/cloudinary-vs-s3/>.
- [52] Matti Luukkainen. *Fullstack part6*. 2022. URL: <https://fullstackopen.com/en/part6>.
- [53] Chris Wilson. *A tale of two clocks*. 2013. URL: <https://web.dev/audio-scheduling/>.
- [54] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [55] Dmitri Tymoczko. "Review of Michael Cuthbert, Music21: a Toolkit for Computer-aided Musicology (<http://web.mit.edu/music21/>)". In: *Music Theory Online* 19.3 (2013). DOI: [10.30535/mtol.19.3.11](https://doi.org/10.30535/mtol.19.3.11).
- [56] Francis Rumsey. *Desktop audio technology*. 1st ed. Focal Press, 2013.
- [57] Polyphone, 2022.
- [58] Arthur Fox. *WAV Or MP3: Which Is The Superior Audio Format?* 2022. URL: <https://mynewmicrophone.com/wav-or-mp3-which-is-the-superior-audio-format/>.
- [59] Mark Harris. *FLAC: A Superior Lossless Audio Format*. 2021. URL: <https://www.lifewire.com/what-is-flac-audio-format-2438548>.
- [60] James Nugent. *Lossless vs. lossy audio: FLAC, WAV, MP3, and other formats*. 2021. URL: <https://higherhz.com/lossless-vs-lossy-compression-audio-formats/>.
- [61] Dan Arias. *Hashing in Action: Understanding bcrypt*. 2021. URL: <https://auth0.com/blog/hashing-in-action-understanding-bcrypt/>.
- [62] Dominik Grzedzielski. *Why you should (probably) use Typescript*. 2021. URL: <https://thecodest.co/blog/why-you-should-probably-use-typescript>.
- [63] Joel Ross and Mike Freeman. *Client-Side Web Development*. 2022.

- [64] Elijah Trillionz. *Your Heroku App Is Slow to Load Because Of This*. 2021. URL: <https://dev.to/elijahtrillionz/your-heroku-app-is-slow-to-load-because-of-this-4lep>.
- [65] *Cacheable*. 2022. URL: <https://developer.mozilla.org/en-US/docs/Glossary/cacheable>.
- [66] Bob Wise. *Heroku's Next Chapter*. 2022. URL: <https://blog.heroku.com/next-chapter>.
- [67] Dirk Hoekstra. *Why Using Cypress Is Better Than Unit Testing*. 2019. URL: <https://betterprogramming.pub/why-using-cypress-is-better-than-unit-testing-e8234229be81>.
- [68] *WAV vs FLAC - Beginners Guide*. 2021. URL: <https://www.off-the-beat.com/wav-vs-flac/>.
- [69] *Is FLAC Better Than WAV? Here's What You Need To Know*. URL: <https://playbutton.co/is-flac-better-than-wav/>.
- [70] Bohumír Zámečník. *midi2audio*. 2016.
- [71] Ariel Salem. *An Easy-To-Use Guide to Big-O Time Complexity*. 2017. URL: <https://medium.com/@ariel.salem1989/an-easy-to-use-guide-to-big-o-time-complexity-5dcf4be8a444>.
- [72] Megan Lavengood. *Open Music Theory*. 2nd ed. 2021, p. 515.
- [73] Stuart Burns. *Use Docker and Alpine Linux to build lightweight containers*. 2020. URL: <https://www.techtarget.com/searchitoperations/tutorial/Use-Docker-and-Alpine-Linux-to-build-lightweight-containers>.

9 Appendix

9.1 GitHub Repositories

Initial versions for user evaluation

[Clicktrack editor frontend](#)

[Audio processing backend](#)

Current versions

Please note that these repositories may receive updates in the future. To see the source code at the time of submission refer to the attached zip archive.

[Clicktrack editor frontend](#)

[Audio processing backend](#)

[User Management backend](#)

9.2 User Survey and Related Documents

The following pages include the questionnaire and Participant Information Sheet sent out, as well as the recruitment email itself and the ethical approval from COMSC SREC.



Advanced Clicktrack Web App - User Evaluation

Form to critically evaluate the usefulness and user experience of an Advanced Clicktrack web application made for musicians

* Required

Initial Questions

Please answer these questions first, before trying out the web app.

1. I confirm that I am 18 or older *

☐ Yes

☐ No

2. Have you ever used a metronome or click track when playing music?

*

☐ Yes

☐ No

3. How often do you play or compose pieces of music in which the tempo changes? *

- ☐ Very often
- ☐ Sometimes
- ☐ Rarely
- ☐ Never

Evaluation of the web app

Please take some time to try creating some click tracks with the application and exploring its features. You can access the application here: <https://clicktrack-redux.vercel.app/>

4. How easy was it to figure out how the application worked? *

- ☐ Effortless
- ☐ Fairly Easy
- ☐ Neutral
- ☐ Difficult
- ☐ Very Difficult

5. Were you able to generate tempo change sections in which the rate of tempo change felt natural, i.e. like something you would hear in a real song? *

- ☐ Yes
- ☐ No

6. Would you use this application for any of the following? *

- ☐ Constructing a click track to record original music
- ☐ Constructing a click track to record a cover
- ☐ Constructing a click track to practice a piece of music
- ☐ Constructing a click track for live performance with other musicians
- ☐ Constructing a click track for jam sessions with other musicians
- ☐ Other

7. If you answered "other" to the above question, please elaborate.

8. Do you have any suggestions for how to improve existing features of the app?

9. Do you have any suggestions for additional features to be added?

This content is neither created nor endorsed by Microsoft. The data you submit will be sent to the form owner.



Microsoft Forms

PARTICIPANT INFORMATION SHEET

Development and Evaluation of a web application focused on advanced rhythmic features

You are being invited to take part in a research project. Before you decide whether or not to take part, it is important for you to understand why the research is being undertaken and what it will involve. Please take time to read the following information carefully and discuss it with others, if you wish.

Thank you for reading this.

1. What is the purpose of this research project?

I am an MSc Computing student with an interest in music, especially rhythm, and web development. In this project I aim to develop an application in which users can make click tracks with advanced rhythmic features such as gradual tempo change and polyrhythms (the playing of more than one rhythm simultaneously). The aim of the survey is to gather user feedback on the usefulness of the application and how it could be improved.

2. Why have I been invited to take part?

You have been invited because you are studying Music or play a musical instrument, and therefore might find this application useful, or at least be able to provide valuable insight into how it could be made useful.

3. Do I have to take part?

No, your participation in this research project is entirely voluntary and it is up to you to decide whether or not to take part. If you decide to take part, we will discuss the research project with you and ask you to sign a consent form. If you decide not to take part, you do not have to explain your reasons and it will not affect your legal rights. Involvement in this research project will have no effect on your education or progression through your degree course.

You are free to withdraw your consent to participate in the research project at any time, without giving a reason, even after signing the consent form.

4. What will taking part involve?

You will be given a link to the application and asked to use it based only on the included documentation. After this you will be asked to complete a survey about this experience, focused on ease of use and whether the application would be useful to you. This would be a one-time commitment of around of 10 minutes testing the application and then around 5 minutes to answer the questionnaire. This would likely occur during the months of August or September.

5. Will I be paid for taking part?

No. You should understand that any responses you give will be voluntary and you will not benefit financially now or in the future should this research project lead to the development of a profitable application.

6. What are the possible benefits of taking part?

There will be no direct advantages or benefits to you from taking part, but your contribution will help us understand what musicians are looking for in an advanced rhythm application and how to improve on the existing application.

7. What are the possible risks of taking part?

There shall be no risks in testing or answering the survey.

8. Will my taking part in this research project be kept confidential?

All information collected from (or about) you during the research project will be kept confidential and any personal information you provide will be managed in accordance with data protection legislation. Please see ‘What will happen to my Personal Data?’ (below) for further information.

9. What will happen to my Personal Data?

The only personal data collected will be your responses to the questionnaire, which will be strictly related to your evaluation of the application.

Cardiff University is the Data Controller and is committed to respecting and protecting your personal data in accordance with your expectations and Data Protection legislation. Further information about Data Protection, including:

- your rights
- the legal basis under which Cardiff University processes your personal data for research
- Cardiff University’s Data Protection Policy
- how to contact the Cardiff University Data Protection Officer
- how to contact the Information Commissioner’s Office

may be found at <https://www.cardiff.ac.uk/public-information/policies-and-procedures/data-protection>

Your survey responses will be processed during the week starting the 29th of August.

Anonymised information will be kept for a minimum of two weeks but may be published in support of the research project and/or retained indefinitely, where it is likely to have continuing value for research purposes.

Personal data and samples collected up until the point of participant can be withdrawn on request.

10. What happens to the data at the end of the research project?

Overall statistics from the data will be included in the final written dissertation, which may or may not be published. Additionally, findings from the question on potential improvements to the application may be used to create a plan for future development. The raw data itself will be deleted after completion and moderation of the dissertation.

11. What will happen to the results of the research project?

The results of this research could potentially be published from December 2022, after marking of the dissertation. Participants will not be identified in any report, publication or presentation.

12. What if there is a problem?

If you wish to complain, or have grounds for concerns about any aspect of the manner in which you have been approached or treated during the course of this research, please contact Prof. David Marshall or myself, Daniel Redly. If your complaint is not managed to your satisfaction, please contact Dr. Katarzyna Stawarz: comsc-ethics@cardiff.ac.uk

If you are harmed by taking part in this research project, there are no special compensation arrangements. If you are harmed due to someone's negligence, you may have grounds for legal action, but you may have to pay for it.

13. Who is organising and funding this research project?

The research is organised by Prof. David Marshall and MSc student Daniel Redly, through the School of Computer Science and Informatics. The research is currently not funded.

14. Who has reviewed this research project?

This research project has been reviewed and given a favourable opinion by the School Research Ethics Committee.

15. Further information and contact details

Should you have any questions relating to this research project, you may contact us during normal working hours:

Daniel Redly, 07765289285, RedlyDP@cardiff.ac.uk

Prof. David Marshall, +44 (0)29 2087 5318, marshallad@cardiff.ac.uk

Thank you for considering to take part in this research project. If you decide to participate, you will be given a copy of the Participant Information Sheet and a signed consent form to keep for your records.

User evaluation for advanced click track and rhythm web application - for one of my student's MSc Computing Dissertation Study

David Marshall <MarshallAD@cardiff.ac.uk>

Tue 8/30/2022 3:12 PM

To: **DG COMSC MSc** <dg.comsc.msc@cardiff.ac.uk>

Cc: **DG COMSC T Staff** <DG.COMSC.TeachingSchoolStaff@cardiff.ac.uk>; Daniel Redly <RedlyDP@cardiff.ac.uk>

 2 attachments (559 KB)

07_4_Appx3_Templ-PIS.docx; 07_5_Appx4_Templ-Consent-Form.docx;

Dear COMSC MSc Student/MSc Student Supervisor

Please could you help a fellow MSc student/one of my MSc students complete his MSc project by completing a user evaluation for his advanced click track and rhythm web application.

Many thanks

Dave Marshall

> Subject: User evaluation for advanced click track and rhythm web application - for MSc Computing Dissertation

>

> Dear all,

>

> I am a MSc Computing Student and am looking for volunteers to complete a survey for my dissertation. I am creating a web application which enables musicians to practice music with tempo changes. This would include the creation of click tracks (backing track in which a sound is played to help musicians stay in rhythm), with advanced rhythmic features. These would include gradual and abrupt tempo changes as well as polyrhythms (more than one different rhythm occurring at once).

>

> In order to assess the quality of this product, I would be really grateful if people with a knowledge of music could complete this survey.

>

> The survey is short and should take you no longer than 10 minutes to test the application and 5 minutes to complete the survey. All data will be anonymous. A consent form and participant information sheet have been attached to this email. Please look over them carefully and sign if you wish to proceed. Below is a link to the survey, which also contains a link to the application itself. The application and survey both work on mobile as well as desktop devices.

>

>

>

>

>

> Advanced Clicktrack Web App - User Evaluation

> Form to critically evaluate the usefulness and user experience of an Advanced Clicktrack web application made for musicians:

>

<https://forms.office.com/Pages/ResponsePage.aspx?id=MEu3vWiVVki9vwZ1l3j8vERp536xqKFFv04A4KIObJZURDZPRUNPMkhCWTQ1VIZTV0cyRk9HQUxCTi4u>

> Daniel Redly

>

Re: Ethics Approval for MSc Dissertation

COMSC Ethics <comsc-ethics@cardiff.ac.uk>

Wed 8/10/2022 1:16 PM

To: Daniel Redly <RedlyDP@cardiff.ac.uk>

Dear Daniel,

Research project title: Development and Evaluation of a web application focused on advanced rhythmic features

SREC reference: COMSC/Ethics/2022/067

The SCHOOL OF COMPUTER SCIENCE & INFORMATICS RESEARCH ETHICS COMMITTEE ('Committee') reviewed the above application at the meeting held electronically on 10/08/2022.

Ethical Opinion: FAVOURABLE

The Committee gave a favourable ethical opinion of the above application on the basis described in the application form, protocol and supporting documentation.

Additional approvals:

This letter provides an ethical opinion only. You must not start your research project until all appropriate approvals are in place. This is not a matter for the committee to advise on. For student projects, you should contact your supervisor. For staff projects/supervisors, contact the School Research Office comsc-research@cardiff.ac.uk.

Amendments:

Any substantial amendments to documents previously reviewed by the Committee must be submitted to the Committee via comsc-ethics@cardiff.ac.uk for consideration and cannot be implemented until the Committee has confirmed it is satisfied with the proposed amendments.

You are permitted to implement non-substantial amendments to the documents previously reviewed by the Committee but you must provide a copy of any updated documents to the Committee via comsc-ethics@cardiff.ac.uk for its records.

Monitoring requirements:

The Committee must be informed of any unexpected ethical issues or unexpected adverse events that arise during the research project. This notification should be made via email to us. The Committee must be informed when your research project has ended. This notification should be made to comsc-ethics@cardiff.ac.uk within 3 months of research project completion.

Complaints/Appeals:

If you are dissatisfied with the decision made by the Committee, please contact the school's Ethics Officer, Dr Katarzyna Stawarz in the first instance to discuss your complaint. If this discussion does not resolve the issue, you are entitled to refer the matter to the Head of School for further consideration. The Head of School may refer the matter to the Open Research Integrity and Ethics Committee (ORIEC), where this is appropriate. Please be advised that ORIEC will not normally interfere with a decision of the Committee and is concerned only with the general principles of natural justice, reasonableness and fairness of the decision.

Please use the Committee reference number (SREC reference) on all future correspondence.

The Committee reminds you that it is your responsibility to conduct your research project to the highest ethical standards and to keep all ethical issues arising from your research project under regular review.

You are expected to comply with Cardiff University's policies, procedures and guidance at all times, including, but not limited to, its Policy on the Ethical Conduct of Research involving Human Participants, Human Material or Human Data and our Research Integrity and Governance Code of Practice.

Yours sincerely,
COMSC SREC