University of Waterloo
Faculty of Engineering
Department of Electrical and Computer Engineering

TempoTracks

Group 2023.017

Garrick Clarke	g3clarke@uwaterloo.ca	20822323
Rizwan Kassamali	rkassama@uwaterloo.ca	20834079
Alexander Ma	a37ma@uwaterloo.ca	20822811
Matthew Panizza	mpanizza@uwaterloo.ca	20822324
Kuba Rogut	krogut@uwaterloo.ca	20844610
Sterlyn Tang	s224tang@uwaterloo.ca	20859326

Consultant: Douglas Wilhelm Harder

30 June 2023

Table of Contents

1 High-Level Description

1.1 Motivation

There is no easy way to create a playlist that would match your workout pace, whether that is to keep up with a target pace to reach a goal or for the music to keep up with the runner's pace. Furthermore, it is even more of a challenge, if the pace of the workout varies, songs will not speed up and slow down to keep the runner going at the current pace. Music can provide ergogenic, psychological, and psychophysical benefits during physical activity, especially when movements are performed synchronously with music [1]. The goal of this application is to help athletes in their workouts keep a specified tempo or be able to reach a time goal where they can run to a modular soundtrack. A study showed that music speed is directly related to how athletes perform. Speeding up the music program increased distance covered/unit time, power, and pedal cadence from 0.7-3.5% respectively; slowing the program produced falls of 3.8-9.8% [2]. By harnessing the power of music, we are able to help individuals train better and become stronger.

1.2 Project Objective

The goal of this project is to give the user an application where they will be able to have music that adapts to the pace of the workout or that pushes the user to hit a pace goal. An iPhone application and an Apple Watch application will be created, and these apps will work together to provide support for the user. Athletes will not have to carry their phones while working out and can just go with their watch to the training session. The phone application will be used to create custom workouts, import the music library, view and track progress over a period of time and more. All workouts will be saved and will be able to synchronize with Apple Health and merge with the user's current workout history.

The Apple ecosystem was chosen for this project due to a large user base, a vast selection of music via Apple Music and the ability to access and modify music data by using the Apple Music Toolkit. The project will leverage the fact that users are still using media they are familiar with, as they are able to use their own music playlists. A user can create a running or triathlon plan with ease and let the app do the rest. It will process these playlists by sorting them by beats per minute and pitch, according to the user's cardio plan. In most cases, the beats per minute will peak near the middle of their run and decrease exponentially at the end. If the user wants to do a different style of cardio, the app will accommodate that as well.

1.3 Block Diagram

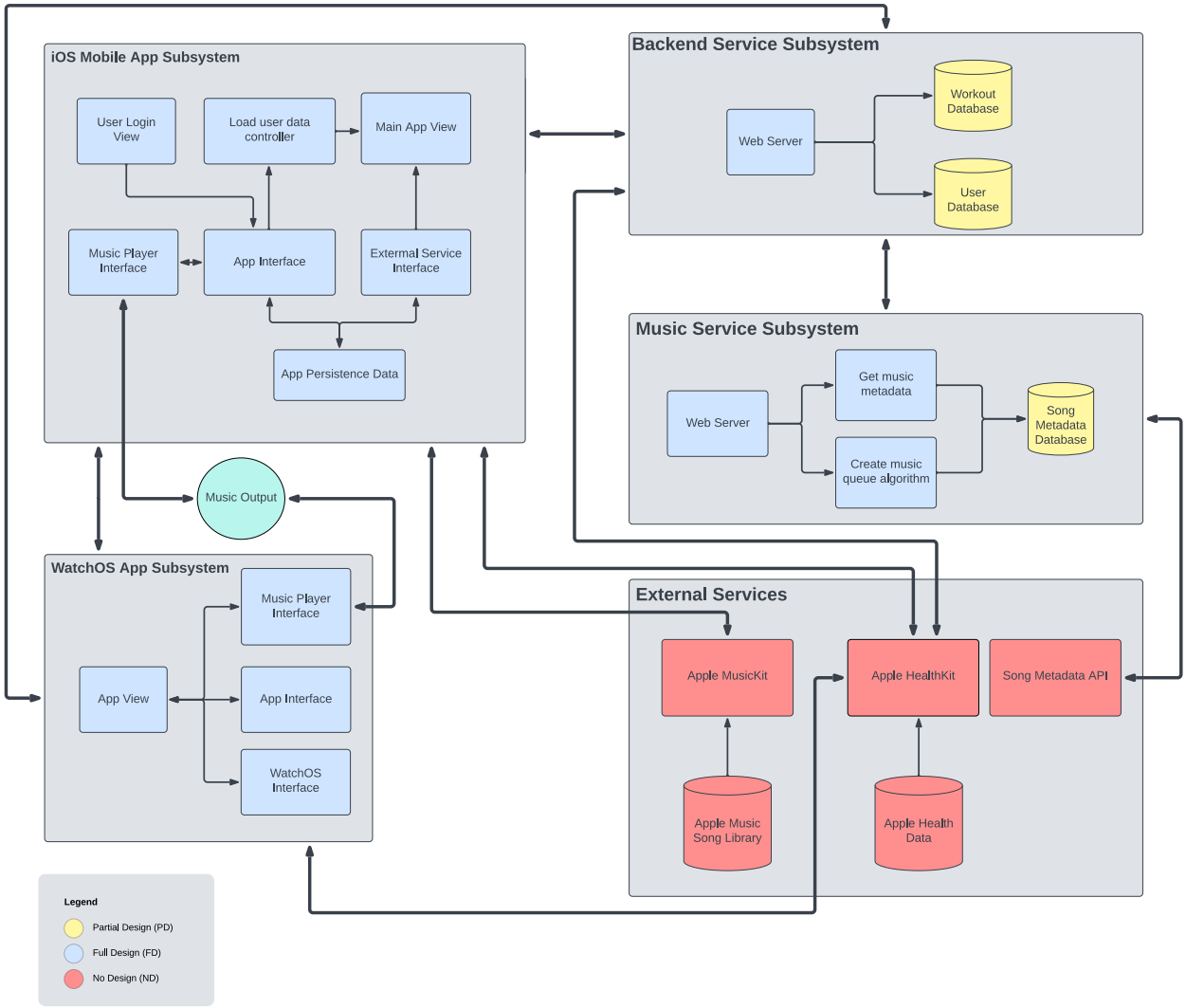


Figure 1-1: *Block Diagram of TempoTracks system*

1.3.1 iOS Mobile Application Subsystem

The iOS mobile application will be one of the main ways a user interacts with our application. From this interface, the user will be able to create, start and stop their workouts, which will also create a queue of music for their workout. To access the user's health data, the application will prompt the user to give us access to their stored health data using Apple's HealthKit. This app will be connected to the user's Apple Music, where we will be controlling the play of music along with the playback speed of the music. The app will also have to connect to the user's Apple Watch so that the data can sync across the devices. Also, we must leverage the iPhone's local data storage so that parts of our app can work offline.

1.3.2 WatchOS Application Subsystem

The watch application is the other way the user will interact with our application. This application will act as the supplementary application to the main iOS application and will only contain the essential features for a user to perform their workouts. Integration with wearable technology is important because users will best experience the app while exercising. The Apple Watch application must be simple, as a wearer should not have to focus on navigating it, only on their exercise. It will be able to create, start and stop workouts, along with playing the music queue. It should also be connected to the user's health data. Furthermore, when separated from the paired iPhone, the watch should be able to save any workout data and resync when the connection is re-established.

1.3.3 Backend Service Subsystem

The backend service subsystem will be responsible for connecting the database to a web server where the front-end applications will be able to request information from the backend. The backend service will consist of two databases. The first database will house the user data, including login information and the health data they choose to share with the application. The second database will have all the workouts for each user and the associated data for each workout. Each database will have its respective tables to help organize the data and make it easily accessible to the front end. The web server will use these databases and create both a public and private interface that will be used by the iOS application and the WatchOS subsystem.

Furthermore, the backend service subsystem will connect to the music service subsystem to get the songs and playlists for each workout as well as the data and metrics of each song such as beats per minute. Lastly, the backend system will need health-related information from Apple HealthKit. Health data such as the heart rate, and the stride length of the user will be used in real-time to adjust and tune the tempo of the music.

1.3.4 Music Service Subsystem

The last subsystem is the music service subsystem. This subsystem will provide the music from the backend system and this subsystem is responsible for obtaining all the metadata by using the external API. It is responsible for speeding up and slowing down the music accordingly. This subsystem will ensure that the output audio still is pleasant to listen to by adjusting other factors such as pitch. Users are expected to create their playlists along with recommended playlists, so this service will provide an entire rebalance of songs.

Moreover, the Music Service will be able to develop playlists for the user when pre-run, or when the user is planning a run. Like other music services, it will create playlists automatically for the user. The service will map the running intensity to the song’s beats per minute and pitch to best sync with the planned expiries. If a user decides to remove, add or swap songs in a generated playlist, this service can perform rebalancing of the songs’ BPM. When a user does modify a playlist the app created, it will treat the playlist as if the user created it (as mentioned above), but most of the work will already be done.

2 Project Specifications

In this section, the functional and non-functional requirements for TempoTracks will be discussed. Table 2-1 details the functional requirements that are needed and Table 2-2 outlines the non-functional requirements. Each specification will be classified as essential or non-essential. Tools, languages, and services were chosen based on several factors such as ease of use, how well they fit the planned design and how scalable they will be in the future. Expansion of this app to other operating systems is kept in mind.

2.1 Functional Specifications

Specification	Description	Classification
Access to health data	The user's health data such as heart rate and stride length will be required and imported from Apple Health to be able to create a playlist matched to their intensity.	Essential
Variety of music	A large variety of music will be needed as the tempo in a workout can change based on the type of workout and the phase of the workout such as warm-up, high intensity or cooldown.	Non-Essential
Import from Apple Music	Apple Music is required as the user must import their music library from Apple Music. This import allows us to control the tempo and pitch of the tracks through Apple Music Kit. The music will still be played through the Apple Music application.	Essential
Ability to speed up/slow down music tempo	In order to match the pace and tempo of the user, the application must be able to adjust the pitch and tempo of the music. This can be achieved using Apple Music Kit where parameters such as pitch and tempo can be set for the songs in the workout playlist.	Essential
Evaluate desired speed to determine next song choices	A graph will be used to determine the desired speed at any given point in the run. This function should choose an appropriate song for the run and modify the speed and the pitch as necessary.	Essential
Create a workout for the user	The user must be able to create a workout within the app. The workout should contain the activity, the duration, the intended goals, and details specific to the activity. The users must also pair songs or a playlist to this workout through the Apple Music import.	Essential
Start workout	The application must immediately start the workout upon the user's command and start to play the music with the correctly adjusted tempo.	Essential

Play a song to match the user's pace	The application must be able to play a song to match the user's pace.	Essential
Present workout and relevant health data to the user	The application must show the workout data to the user on the watch in real-time. The app must also show the complete workout data and the health data such as heart rate variability on the phone app after the workout is complete.	Non-Essential

Table 2-1: *Functional Specifications*

2.2 Non-Functional Specifications

Specification	Description	Classification
Watch Application Size	The watch app should not be more than 300MB as smartwatches tend to have less storage space than other devices such as smartphones.	Essential
Phone Application Size	The phone app should not be more than 700MB since larger app sizes can defer users from downloading the app.	Essential
Battery Consumption	The watch app should not consume more than 15% of battery capacity per hour. Watches tend to have a smaller battery size in comparison to other devices (e.g. smartphones). Since the watch screen will be on during the workout, this will also place extra strain on the battery.	Essential
Internet Access	Both the mobile and watch apps should utilize the internet to allow the user to log in, sync their Apple Music library and fully utilize some of the other services/subsystems.	Essential
Offline Usability	The app could be accessed offline. This would be convenient for the user but can be avoided with a data plan.	Non-Essential
Scalability	Must be able to support over 5000 users at the same time.	Non-Essential
Backend Server Uptime	The backend server should be up and running 99% of the time to ensure that users are able to store and retrieve data which is not stored locally on their device.	Essential
Supports various Apple watches	The app should work on the Apple Watch series, the Apple Watch SE series, and the Apple Watch Ultra. The watch application must consider the difference between the 3 series of watches, specifically the always-on display, screen size and their respective battery sizes.	Non-Essential

WatchOS / iOS supportability	The app should work on WatchOS 9.0.0 and future updates to the operating system. The phone app should work on iOS 16.0.0+.	Essential
Data Loss	If the health or accelerometer data is not updated, the app should continue playing music at the predetermined intervals set in the workout or at the last known pace for an open-ended workout.	Essential

Table 2-2: *Non-Functional Specifications*

3 Detailed Design

In this portion of the document, each subsystem will be discussed in depth and a rigorous explanation will be given of the choices that were made for each decision. The 4 subsystems are the iOS application, WatchOS application, backend service and music service.

3.1 iOS Application Subsystem

Problem Definition:

The iOS application will run on the user's Apple iPhone. This application will have to synchronize and retrieve data from the database similar to the WatchOS application. It will also have to synchronize with the WatchOS application itself for the music portion of TempoTracks. The iPhone application will also have more detailed statistics about the user's workouts and will show them in-depth information that will not be shown on the watch application due to its limited size.

Framework/Language Choice:

One of the key decisions of the iOS application will be the framework/language choice. The framework/language of choice should be easy to develop in, have good, extensive documentation, is scalable, is up to date, and be lightweight. These are the key things that should be considered when choosing a framework.

Framework/Language	Pros	Cons
Swift [3]	- Apple's native development language - Easy to learn - Up-to-date documentation - Highly Scalable - Very lightweight - Easy to synch with Apple Watch	- Not very versatile. Hard to compile for other OSes like Android - Poor third-party tool interoperability - Not a very common language. Would have to learn this language
React Native [4][5]	- Extremely popular framework	- Limited access to some native APIs for iOS, etc.

	- Built on JavaScript, one of the most widely used languages today - Supports multiple OSes (Easy to port existing code to different OSes like Android) - Easy to learn - Up-to-date documentation - Constantly updated - Highly Scalable - Lightweight	- Hard to debug. The combination of JavaScript and native code can make debugging challenging.
Flutter [5][6]	- Extremely popular framework - Built on Dart, a popular language for mobile and web development - Supports multiple OSes - Easy to learn - Up-to-date documentation - Constantly update - Highly Scalable - Lightweight - Flutter builds UI from the ground up so debugging is easier than React Native	- Our group does not have experience with using Dart. So, there would be a learning curve - Relatively new framework - Limited access to platform-specific features
Xamarin [5]	- Built on C#, which is a common language used in industry - Supports multiple OSes - Documentation is a bit worse than other framework counterparts - Provides access to platform-specific APIs	- Limited community support - Most of our group does not know C#

Table 3-1: *iOS Framework Choice*

Given the above pros and cons, React Native was chosen over the other frameworks and languages. It provides the best of all worlds in terms of cross-platform compatibility, ease of use, being lightweight, and having up-to-date documentation, despite the cons.

Authentication:

The iOS application will rely on the backend server for user authentication. The user's data will be secured through their account, and we will use a service like AWS Cognito to help achieve this goal. This allows us to have a central place for all user data and by using an existing system, security flaws will be mitigated.

User Interface:

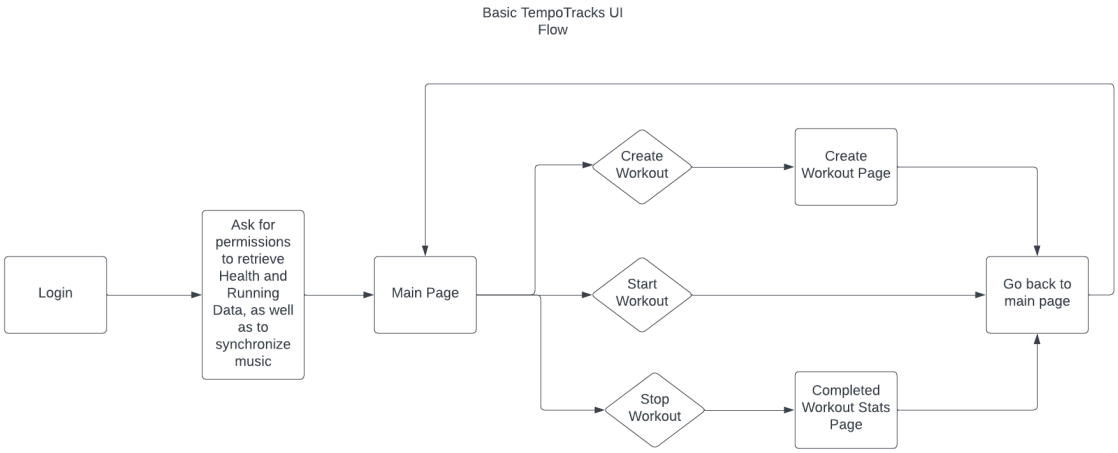


Figure 3-1: *User Interface Flow*

The basic iOS application flow is shown in Figure 3-1 above. The user will land on a login page. Upon logging in, the application will ask for permissions and the user will then be redirected to the main page.

Main Page:

The main page will contain general user information as well as settings, navigation buttons to other pages listed in the diagram, general workout stats, past workout performances, current workout stats if the user started a workout, and a list of user-created workouts to choose from to start.

Create Workout Page:

This page is a form that allows the user to input workout stats such as pace at different parts of the workout. For example, the user can set a custom interval plan, with a 2-minute sprint followed by 3 minutes of low-intensity for 6 sets and then a 10-minute cooldown. The user could also set their run to gradually increase in speed over time until they hit a desired pace. This data will be used to help determine what songs should be played at what time and how much the speed of the song will have to change.

Completed Workout Stats Page:

This page will show workout stats to the user after they have completed their workout. This includes calories burned, pace, distance covered, and other health data. Data can also be broken down into increments where the user can view their pace and heart rate per kilometre or specified time segment.

Persistence Cache:

A persistent cache will also be built to store all shown workout data that is updated on a per-time basis.

3.2 WatchOS Application Subsystem

Problem Definition:

The WatchOS application will be the application that runs on the user's Apple Watch. This application will need to be able to synchronize and retrieve data from the database similar to the iOS application. The app will feature a simplistic user interface intended for quick and easy use. It will have functionality such as chose playlist, stop, skip, resume song, start plan, and choose plan. Features such as creating running plans and modifying beats per minute and pitch will be limited to the full iOS app for safety and ease of use concerns. It is not intended that this app be used while the user is running but recommended to be used during exercise breaks.

Technology Used:

Apple Watch apps are only limited to two base programming languages. The most popular is Swift, a popular and easy language to build apps with. Swift was developed by Apple Inc. Objective C is another option. It is much older than Swift and very similar to other object-oriented languages such as Java or C#. Objective C was not developed by Apple Inc, but rather by Productivity Products International.

Framework/Language	Pros	Cons
Swift [7]	- Developed by Apple, used for iOS apps - Easy to learn - One of the only used languages for iOS, much more than alternative	- Development only pertains to iOS Watches (no Galaxy, etc.) - Poor third-party tool interoperability
Objective-C [8]	- Strongly typed language - Very old, with lots of documentation	- Not very popular - Slower than alternative

Table 3-2: *WatchOS Framework Choice*

Based on the table above and the nuances of coding for an Apple Watch, we have chosen to use Swift. Since the Apple Watch is a relatively new device, most development is done using

Swift, and using Swift would allow us to access new proprietary features. Furthermore, Swift would have better integration with Apple Music Kit.

Workout Home Page:

On this page, the user will be able to see a list of the custom workouts they have created. The user will be able to make small adjustments to a workout, such as changing the duration of the workout. The user will also be able to start a workout from this page which will directly launch them into the workout mode.

Workout mode:

The workout mode will show the user data pertaining to the current workout. In live time they will be able to see the duration, current calories burnt and heart rate. The user will also be able to adjust the song and change the volume. Finally, the user will also have controls for the workout such as the ability to pause/resume the workout.

3.3 Backend Service Subsystem

Problem Definition:

The backend service will the primary interface for the iOS and WatchOS apps to interact with. It will allow the frontend applications to interact with the rest of the subsystems and be able to perform functions like fetching a user's saved workouts or allowing a user to start a new workout. This subsystem will be responsible for any computation that does not directly involve music, such as user authentication or creating or updating a workout. There are many choices in how the backend is implemented in terms of the hosting, authentication, database type, database schema, as well as the API framework and its corresponding schema.

Backend Technology:

The toughest decision to be made regarding the backend is to decide where this backend service will live. There are many options to choose from, such as large cloud platforms like Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure. These are referred to as Infrastructure-As-A-Service (IaaS). These IaaS solutions allow nearly complete freedom in terms of framework choice, as well as great control in fine-tuning the backend. There are alternatives such as Firebase or Supabase, which are platforms built upon this IaaS solution that remove some of the complexity that comes with issues of scaling and security by simplifying the development process. Furthermore, these platforms come with additional tools that make the development experience easier, such as built-in authentication functionality with easy integrations to single-sign-on (SSO) through Google, Facebook, etc. These Platforms-as-a-Service (PaaS) have the downside that they may be easier to initially build upon, the simplification of the platform may cause issues down the line due to not being able to control some of the fine details. Similarly, these platforms are usually heavily opinionated and force you to use specific frameworks or technologies to build upon them. This decision cannot be made without first

analyzing the sub-problems of the backend and seeing which hosting providers match best with our sub-problem decision.

Authentication:

User authentication is a problem that needs to be solved by virtually all developers for any product. This is good news because there are many tools that make building user authentication easier. These tools are also important to use because they are proven to be secure, whereas building a custom authentication flow from scratch may lead to accidental security issues. These tools can be separated into several categories, where each category represents a different layer of abstraction and interoperability.

Category	Abstraction	Interoperability
Authentication libraries (Ex passport.js?)	- abstracts the authentication primitives - still requires custom implementation of the Authentication API (user login, signup, registration routes) as well as User database to store the username and passwords - free	- Able to be used in any setup
Authentication Platforms (Ex 0Auth, Clerk)	- abstracts the whole authentication API and User database - comes with great developer experience through great monitoring, user interface, and analytics - most expensive option - enterprise-level solution	- Used for any sort of cloud hosting provider, has good interoperability
Cloud Tools (Ex Supabase Auth, AWS Cognito)	- abstracts the whole authentication API and User database - comes with a decent developer experience and extras - good price	- Amazing when using other technology in the same cloud platform, developer experience gets much worse when crossing over

Table 3-3: *Authentication Tools*

After consideration, it would be best to use a cloud tool such as AWS Cognito or Supabase for our project. It abstracts away any of the headaches that comes with security and scalability for our authentication solution, while also having a good developer experience and great synergy with the other cloud technologies we will end up using. The specific cloud tool chosen will be selected at the end of this section using custom criteria.

Database Choice (SQL or NoSQL):

There are two main categories of databases which we can choose from, a relational database and a non-relational database. A relational database is a traditional database that allows for relationships between rows. Examples of this type of database are PostgreSQL or MySQL. The

main advantage of a relational database is that you are more easily and more efficiently query multiple sets of rows across multiple tables. The other important feature is that there are strict data models per table, so every row in a table has the exact same structure. This makes querying data easier and more consistent, but this makes it harder to quickly iterate the database schema. However, a non-relational database removes the concept of relationships in a database and allows for unstructured data. The downside is that it is harder to query the relationships between tables and rows. These databases are highly scalable and highly performant for most queries. For our project, we aim to go with a relational database as our data will have lots of relationships, and we will not be at the point where we will be running into scalability issues with relational databases.

Database Schema:

There are several tables that need to be made in our database, and as we are going to choose a relational database, we must define our data schema. Our databases must follow non-redundant forms. For this subsystem, we need a table to store users and their associated metadata, workouts, and workout templates.

User Table:

Column Name	Field Type	Description
user_id	string	Will be a unique identifier for each user. Will be related to the user id in our authentication method
email	string	The user's email must be unique per user
first_name	string	User's first name
last_name	string	User's last name

Table 3-4: *User Table*

Workout Table:

Column Name	Field Type	Description
workout_id	string	Unique identifier for each workout
user_id	string	Is key based on each user's unique identifier
status	ENUM["completed", ''in-progress"]	Represents whether or not the workout is currently in progress or has been completed
time_start	string	Time workout started
time_end	DateTime	Time workout ended
time_duration	integer	Duration of workout

workout_type	string	Will be a representation of what kind of workout the user performed, i.e. running, cycling, HITT
total_distance	Integer	Tracks distance travelled in metres
total_energy_burned	Integer	Tracks total calories burned
total_elevation_gain	integer	Tracks vertical distance travelled in metres
playlist_id	string	Identifier that relates to the associated playlist
training_intervals	JSON	Represents the structure of the changes in pace in the workout. i.e should show the high-intensity and low-intensity time intervals for a HITT workout

Table 3-5: Workout Table

Workout Template Table:

Column Name	Field Type	Description
workout_template_id	string	Unique identifier for each saved workout template
user_id	string	Is key based on each user's unique identifier
workout_type	string	Will be a representation of what kind of workout the user performed, i.e. running, cycling, HIIT
playlist_id	string	Identifier that relates to the associated playlist
training_intervals	JSON	Represents the structure of the changes in pace in the workout. i.e. should show the high-intensity and low-intensity time intervals for a HIIT workout

Table 3-6: Workout Template Table

User Table Code:

```
CREATE TABLE user
(
    user_id VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL,
    first_name VARCHAR(255) NOT NULL,
    last_name VARCHAR(255) NOT NULL,
    PRIMARY KEY (user_id)
);
```

Figure 3-2: *Code to create the User Table*

Above is the code to create the user table in the database. This initializes all the variables to a maximum length of 255 and sets the primary key of the table to the user_id, ensuring that it has to be unique. A similar code is used to create the workout table and the workout template table which have a respective id as the primary key, to ensure that all data is easily quarriable. Each workout and workout template database will have a foreign key of the user_id so that they can link each workout back to a specific user via the user_id.

API Framework

The first sub-problem we need to make a decision for is to decide whether we want our system to use server or serverless computing. Server computing is the traditional concept of having a dedicated piece of hardware host our software and respond to any web requests. Server computing requires a server to be up 24/7 to serve any requests, which leads to the evolution towards serverless computing. Serverless computing takes advantage of the latest technology in order to build each server's function on request. This leads to lower costs for smaller-scale systems, as you are only paying per request and only when the system is active, whereas before you might have been paying the same rate for your server to be idle 50% of the time. Serverless is able to do this by separating a large server into smaller "functions", which are able to handle the whole functionality of sub-processes inside the server. For example, if you had a server that handled authentication, you could break it up into server-less functions for user login, user logout and user registration. Each function is then built into a micro-container, which is essentially a small virtual machine when a user requests. This also allows serverless technology to scale well, as you can scale horizontally by building new containers to handle more and more user requests. Most serverless technology will automatically scale like this when needed, while classical server computing needs more manual setup in order to similarly scale.

Hosting Provider	Amazon Web Services [9]	Google Cloud Platform [10]
Server Cost	750 hours free, $0.02 per hour after	$0.04 per hour
Serverless Cost	1M free requests per month, $0.20 per 1M requests after	2M free requests per month, $0.40 per 1M requests after

Table 3-7: *Server vs. Serverless for AWS and GCP*

After consideration, serverless computing is the best choice at our scale, where the app can still handle thousands of users whilst keeping the costs low during and after the development process.

User Service API Schema/Endpoints:

API route	Description	Returned value
/user/register	This route will be a wrapper for creating a new user through our authentication tool. This wrapper is helpful and can facilitate functions like logging and better parameter validation	{ user: User, auth: Authenticated JWT token}
/user/login	Similar to the register route, this will be a wrapper over the login method through our authentication tool	{ user: User, auth: Authenticated JWT token }
/user/logout	Similar to the register route, this will be a wrapper over the logout method through our authentication tool	{ response: response code (200 for OK, etc.) }

Table 3-8: *API Endpoints for User Service*

Workout API Schema/Endpoints:

API route	Description	Returns value
/workout/create	This route will be used when a user initially creates a workout. It will also be responsible for setting off the method that will create the playlist for the workout. It should be able to create using completely new data or using a workout template	Will return a Workout data object
/workout/start	This route will be used when the user starts a workout and will update the workout's status.	Will return a Workout data object
/workout/end	This route will be used when a user ends a workout and will update statistics such as duration, calories burned, distance travelled, etc.	Will return a Workout data object
/workout/save	This route is used when a user	Will return a Workout-Template object

Table 3-9: *Workout API Endpoints*

Hosting Provider

Lastly, we need to finally choose the provider we will use to host the backend. The main options to choose from are Amazon Web Services, AWS Amplify, Google Cloud Platform, Firebase, and Supabase.

Hosting Provider	Amazon Web Services [11][12]	Google Cloud Platform [10][12]	Firebase [13]	Supabase [14]
Authentication Tools	AWS Cognito	Google Identity & Google SDK	Firebase Auth	Supabase Auth
Authentication Pricing	50,000 free MAU, $0.0055 per MAU afterwards	50,000 free MAU, $0.0055 per MAU afterwards	50,000 free MAU, $0.0055 per MAU afterwards	50,000 free MAU in the free plan, 100,000 MAU in 25$ / month
Authentication rating	8/10 - seems like a good option	6.5/10 - unclear documentation	8.5/10 - easiest to work with	9/10 - easy to work with and has great integration with their database solution
Database types	Amazon Aurora, PostgreSQL, MySQL, MariaDB, Oracle Database, and SQL Server	MySQL, PostgreSQL, and Microsoft SQL Server	NoSQL	PostgreSQL
Database pricing	20GB free storage space, $0.115 per GB-month thereafter	$0.09 per GB-month	1 GiB free storage space, then $0.1725 per GiB-month	500MB storage space in the free plan, 8GB in 25$ / month plan
Database rating	8/10 - amazing free tier	8.5/10 - cheap price if we use a lot of storage space	5/10 - We want to aim to use a relational database	7/10 - Easy to work with but not a lot of storage space included
Compute options (server vs serverless)	Server and serverless. Any language/frame work accepted	Server and Serverless. Any language/frame work accepted.	Only server-less using JavaScript/Types cript	Only server-less using Typescript
Compute pricing	Server- 750 hours free,	Server - $0.04 per hour	2M free requests per	500,000 free requests per

	$0.02 per hour after Serverless – 1M free requests per month, $0.20 per 1M requests after	Serverless – 2M free requests per month, $0.40 per 1M requests after	month, $0.40 per 1M requests after	month, 2M free requests per month in a $25/month plan
Compute rating	8/10 - good free tier	8.5/10 - best free tier	8.5/10 - best free tier	7/10 – the most expensive option
Developer experience	7/10 - complicated to set up with a bad interface, but great external tools	6.5/10 - complex with poor documentation	8/10 - simplification of Google Cloud Platform	9/10 - Great documentation and development libraries
Developer tooling	8.5/10	7/10	7/10	8.5/10
Platform integrations	9/10	8/10	7/10	7.5/10
Team familiarity	8/10	4/10	7/10	8.5/10

Table 3-10: *Hosting Provider Choice*

The above scores were computed via a voting process in a team meeting. During this meeting, team members researched each of the above technologies and noted their features such as pricing, available options, availability of documentation, personal experience with technology and even integrations. These features were noted, and each member had to rate each of the features on a scale from 1 to 10. The average for each feature was then calculated and are the scores shown above.

After looking at the options and their scores, the team decided to choose Supabase for the backend subsystem. Supabase will allow the team to iterate on the project quickly with its easy development experience and great tooling. This was important because we want to prioritize the development process versus scalability and performance. Supabase also has a free tier that is good enough for up to a few thousand active users, making it a budget-friendly option that also allows us to scale more in the future if need be.

3.4 Music Service Subsystem

Problem Statement:

This service is intended to perform all logic related to the music sorting algorithm as part of our product. Using a stream of songs and their data from Apple Music Kit, the service will create custom playlists for the user to select. The playlists will be specialized to the user's running plans. The song selection is determined by three factors: what is trending, the genre/type of music the user listens to on their Apple Music account and which songs are most compatible with our algorithm. When a playlist is created by the service, it is then sorted by beats per minute, pitch and other factors to fit current running plans on the user's account. A playlist can also be created and modified by the user. In this case, the music service will have to sort the songs in the same way as automatically created playlists. If a song sounds "off" or unnatural at too low/high beats per minute, it may be recommended to be removed from the playlist.

Technology Used:

Extensive research was done to identify possible technologies which can be utilized for our Music Service Subsystem. Only the two most viable options were considered since other third-party services and APIs such as SoundCloud API lacked any of the capabilities the app required.

Third-Party Service/API	Pros	Cons
Spotify API [15]	- Easy to integrate into projects - Availability of extensive documentation	- API endpoints do not allow for the specific parameter controls our app requires such as flexible playback speeds
Apple Music Kit [16]	- Flexible options when modifying song properties - Integrated with the Apple ecosystem, users can play from Apple Music or even their local music library	- Very limited documentation and even example code snippets to follow

Table 3-11: *Spotify vs Apple Music Kit APIs*

After looking at the available options and interactively testing each service, the team opted for the Apple Music Kit since it was the only third-party service which allowed the full flexibility of parameters which the app needs.

Music Service Subsystem API Schema/Endpoints:

API route	Description	Returned value
/playlist/create	POST – Creates a new playlist based on user data	{ playlist: Playlist, response: response

		code (200 for OK, etc.) }
/playlists/delete/{playlistId}	DELETE – Deletes the playlist matching the playlist ID parameter	{ response: response code (200 for OK, etc.) }
/playlists/sort/{playlistId}	GET – Returns a sorted playlist for the specified playlist ID	{ playlist: Playlist, response: response code (200 for OK, etc.) }
/playlists/load/{playlistId}	GET – Returns a playlist	{ response: response code (200 for OK, etc.) }
/songs/load/{songId}	GET – Load song properties for a specific Apple Music track	{ response: response code (200 for OK, etc.) }
/play/song/{songId}/{playbackSpeed}/{pitch}	POST – Plays the song matching the song ID parameter, at the specified playback speed and pitch	{ response: response code (200 for OK, 404 for Song not found, etc.) }

Table 3-12: *Music Subsystem Endpoints*

4 Discussion and Project Timeline

4.1 Evaluation of Final Design

The objective of this app is for users to have an application that takes their music library and adjusts the tempo of the music to match the pace of their workout to help athletes train more efficiently and reach their goals faster. The proposed design in this document allows us to reach meet our objective based on the decisions made. We have chosen 4 subsystems that will work effectively together to be able to meet the function and non-functional requirements listed in Tables 2-1 and 2-2.

By using React Native for the iOS application, we are able to display all the data quickly and efficiently to the user. As React Native is a large and flexible framework we can leverage certain properties such as the virtual DOM that allows React Native to be lightweight and minimize battery usage, while also being faster than a traditional DOM. Similarly, with the WatchOS application, by choosing to use the Swift language, we are able to seamlessly integrate with the current workouts and “now playing” apps on the Apple Watch to help the user better control their workout. Furthermore, Swift will have better support for Apple Watch which means that we can support all Apple Watches that Apple currently supports.

The backend had the most difficult choice as this will connect to the 3 other subsystems. We chose Supabase as the overall winner because it had the most useful features. For example, a relational database was required and Supabase has a PostgreSQL database which is relational. Furthermore, Supabase uses server-less technology which was another choice we made, and our team has a high familiarity with Supabase. Finally, for the music subsystem, we chose to use Apple Music as this would be well integrated into the iOS and WatchOS ecosystem we are developing in. The Apple Music Kit allows us to edit the speed of songs seamlessly and makes it easy for users to import their existing music library.

4.2 Use of Advanced Knowledge

This project utilizes many concepts of upper-year knowledge across all the subsystems. The topics studied in ECE 452 Software Architecture greatly assisted when designing the software architecture of each subsystem and the overall architecture through which they will communicate. ECE 356 Databases provided valuable concepts which assisted in the structuring of the database schemas. This course also provided insight on how to gauge the performance characteristics of different storage patterns which ultimately aided in the decision between a SQL or NoSQL database, a very crucial decision for the application's architecture.

Additionally, the subsystems communicate with each other through HTTP requests which was the main focus of ECE 358 Computer Networks. This course also provided a strong foundation and insight into understanding and debugging network requests and responses which greatly enhances the project's developer experience. Furthermore, the topics covered in ECE 458 Computer Security is also used to shield our subsystems and overall architecture from vulnerabilities. This knowledge was also utilized when deciding on which authentication technology to go with.

4.3 Creativity, Novelty and Elegance

4.3.1 Dynamic Music Tempo

One of the most creative elements is how the music will dynamically adapt to the user's desired need. For example, if the user is doing an interval training workout, the music will speed up automatically during the intense sections of the workout and slow down during the cool-down periods. Currently, some apps can speed up and slow down music, however, these applications use the user's local music stored on their device, however, we are integrating this with the user's Apple Music library. We will use both aspects, which are the pitch and tempo to ensure that the music sounds good when it is sped up.

4.3.2 Apple Watch Application

Another novelty is that the goal of the application is for the tempo change to be fully functional on a watch application. This would mean that the user doesn't need to have their phone with them when they are going on a run and just need to wear their watch. Currently, apps that modify music are done on a phone, by implementing this on a watch, it will be more seamless and convenient for users.

4.4 Student Hours

As of Friday, June 30, 2023, group members have put in the following hours into the design project. Given that the recommendation is 10 hours of work per week, per group member, we have planned meetings and individual work time to equate to approximately 10 hours per person, per week. This trend and strategy will be exercised for the rest of the term where work time and meetings can be adjusted as needed to ensure we are on track for our fourth-year design project.

Group Member	Garrick Clarke	Rizwan Kassamali	Alexander Ma	Matthew Panizza	Kuba Rogut	Sterlyn Tang
Hours logged to Date	61	66	64	64	69	67

Table 4-1: *Current Group Hours*

The times for each member are similar due to our group leveraging the use of sub-teams and pair programming sessions. An ideal week for us consists of a meeting at the start of the week, where we discuss the plan for the coming week, what was done last week and any issues. Tasks are then allocated with an estimated time for each member to complete them before the next meeting.

4.5 Potential Safety Hazards

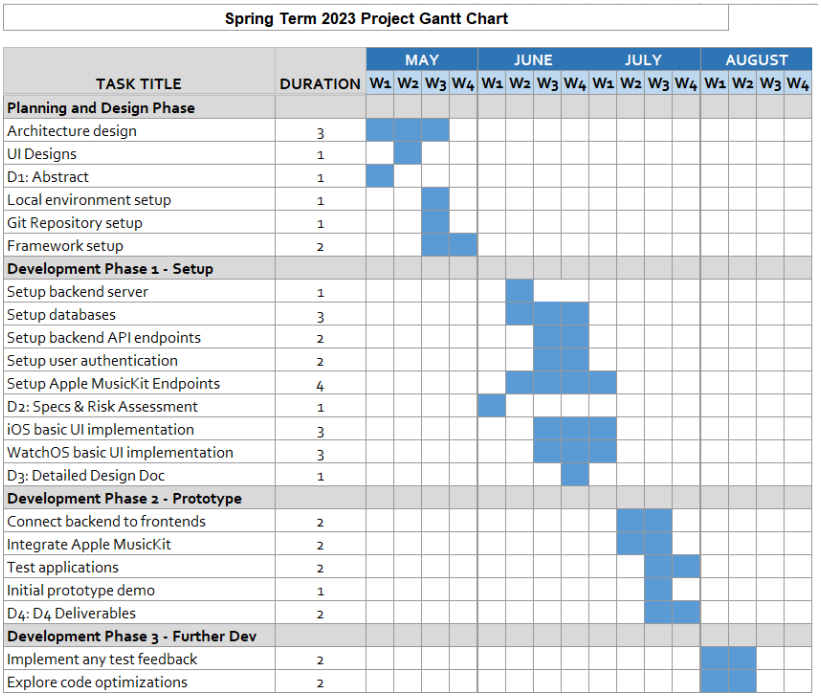
Given the fact that our Watch app will be primarily used while exercising, along with it being customizable by the user, safety concerns should be addressed. One potential hazard for our app is falling during a running session. On some occasions, it is possible for a user to react impulsively to a change in beats per minute. For example, if a user is running at a high pace, and the bpm of the current song playing suddenly drops, the user may try to drop speed too quickly. This reaction can lead to the user tripping and even sustaining injuries from the potential fall.

Another hazard that must be addressed is over-excretion by the user. In a case where the current song's beats per minute increase too fast, they may try to push themselves to a much higher running speed, even if it would be best to slow down. This concern will be considered when designing the application, using the person's weight, height, heart rate and more to cap the beats per minute. However, without proper human coaching fainting or collapsing from over-exercise can still occur, as algorithm prediction models are not perfect. If a user faints outside of supervision, the user is left vulnerable and prone to other humans, animals, and forces of nature.

Lastly, a user may become distracted by using the watch app's user interface while running, for example when switching between songs. While this is a concern for all music apps, it should still be noted as it is potentially the most hazardous on smartwatches, where users are expected to frequently divert their view to use (looking downwards to check the time, notifications, etc.). If this occurs while a user is moving at high speed, the potential of collision with another person, car, building or other object can be significant. Additionally, falling down a hill or incline is possible, which could prove lethal based on the height of the fall. If these cases occur, a user

could face broken bones or death at worst. While this is an unlikely event, it should still be noted on all apps used for running.

4.6 Project Timeline

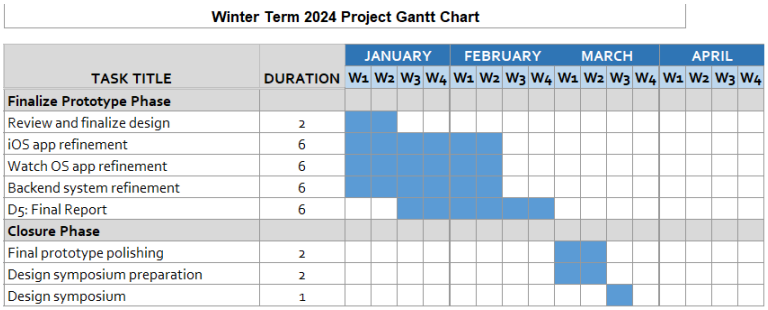


Spring Term 2023 Project Gantt Chart

TASK TITLE	DURATION	MAY				JUNE				JULY				AUGUST			
		W1	W2	W3	W4	W1	W2	W3	W4	W1	W2	W3	W4	W1	W2	W3	W4
Planning and Design Phase																	
Architecture design	3																
UI Designs	1																
D1: Abstract	1																
Local environment setup	1																
Git Repository setup	1																
Framework setup	2																
Development Phase 1 - Setup																	
Setup backend server	1																
Setup databases	3																
Setup backend API endpoints	2																
Setup user authentication	2																
Setup Apple MusicKit Endpoints	4																
D2: Specs & Risk Assessment	1																
iOS basic UI implementation	3																
WatchOS basic UI implementation	3																
D3: Detailed Design Doc	1																
Development Phase 2 - Prototype																	
Connect backend to frontends	2																
Integrate Apple MusicKit	2																
Test applications	2																
Initial prototype demo	1																
D4: D4 Deliverables	2																
Development Phase 3 - Further Dev																	
Implement any test feedback	2																
Explore code optimizations	2																

Figure 4-1: *Timeline for Spring 2023 4A Term*

Above is the timeline for the Spring 2023 term. This Gantt chart outlines the work completed so far and the work that is left to be completed this term. The Spring term is focused on completing all of the scaffolding and set-up work, and most of the application development. In the Spring term, 3 main development phases will be responsible for getting the minimum viable product ready and ensuring that it is in good shape for the Winter 2024 term. The following Gantt chart is for the Winter 2024 term. The Gantt chart depicts the timeline for the Winter 2024 term and has 2 main phases consisting of the final prototype phase, where final polishing touches is applied to the application and the closure phase, where we prepare to demonstrate how our application works and prepare for the design symposium.



Winter Term 2024 Project Gantt Chart

TASK TITLE	DURATION	JANUARY				FEBRUARY				MARCH				APRIL			
		W1	W2	W3	W4	W1	W2	W3	W4	W1	W2	W3	W4	W1	W2	W3	W4
Finalize Prototype Phase																	
Review and finalize design	2																
iOS app refinement	6																
Watch OS app refinement	6																
Backend system refinement	6																
D5: Final Report	6																
Closure Phase																	
Final prototype polishing	2																
Design symposium preparation	2																
Design symposium	1																

Figure 4-2: *Timeline for Winter 2024 4B Term*

5 References

[1] P.C Terry, C.I Karageorghis, A.M Saha, S. D'Auria, "Effects of synchronous music on treadmill running among elite triathletes", *Journal of Science and Medicine in Sport*, 2012. [Online] Available: https://www.sciencedirect.com/science/article/pii/S1440244011001186 (accessed Jun. 5, 2023).

[2] J. Waterhouse, P. Hudson, and B. Edwards, "Effects of music tempo upon submaximal cycling performance," *Wiley Online Library*, 2010. [Online] Available: https://onlinelibrary.wiley.com/doi/10.1111/j.1600-0838.2009.00948.x (accessed Jun. 7).

[3] "Swift Documentation," Swift, https://www.swift.org/documentation/ (accessed Jun. 26, 2023).

[4] "Performance Overview," React Native Dev, https://reactnative.dev/docs/performance (accessed Jun. 26, 2023).

[5] M. Reality, "Mobile reality - xamarin vs. Flutter vs. react native," Mobile Reality - Xamarin vs. Flutter vs. React Native, https://themobilereality.com/blog/xamarin-vs-flutter-vs-react-native#A-few-words-for-the-start (accessed Jun. 26, 2023).

[6] "Build apps for any screen," Flutter, https://flutter.dev/ (accessed Jun. 26, 2023).

[7] Apple Inc., "Creating a watchos app," Apple Developer Documentation, https://developer.apple.com/tutorials/swiftui/creating-a-watchos-app (accessed Jun. 28, 2023).

[8] "Programming with Objective-C," Apple Developer, https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC (accessed Jun. 28, 2023).

[9] "Amazon MQ Pricing | managed message broker service | Amazon Web Services," Amazon MQ Pricing, https://aws.amazon.com/amazon-mq/pricing/ (accessed Jun. 30, 2023).

[10] "Pricing | cloud functions | google cloud," Google, https://cloud.google.com/functions/pricing (accessed Jun. 30, 2023).

[11] H. Roose, "Cognito," Amazon, https://aws.amazon.com/cognito/details/ (accessed Jun. 30, 2023).

[12] "AWS vs GCP - which one to choose in 2023?," ProjectPro, https://www.projectpro.io/article/aws-vs-gcp-which-one-to-choose/477 (accessed Jun. 28, 2023).

[13] "Developer documentation for Firebase," Firebase https://firebase.google.com/docs (accessed Jun. 20, 2023)

[14] "Pricing & fees," Supabase, https://supabase.com/pricing (accessed Jun. 28, 2023).

[15] "Web api," Web API | Spotify for Developers, https://developer.spotify.com/documentation/web-api (accessed Jun. 25, 2023).

[16] A. Inc., "MusicKit," Apple Developer, https://developer.apple.com/musickit/ (accessed Jun. 30, 2023).