# Music Mirror

Luke Marolda, Matt Hegi, and Thomas Lee

Department of Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—Music Mirror is a comprehensive speaker attachment that seamlessly manages song queueing, recommendations, and crowd engagement. Users are able to steer the system through a distributed web app that hosts a suite of song request and consensus voting capabilities. Using our expertise in software systems, machine learning, and hardware systems we were able to develop a final product that can mount to any speaker to provide a maximum of 200 concurrent users with 3 distinct song request formats, as well as the ability to provide live user feedback to alter the queue through vetoes and likes. Two of the three song request formats include song recommendation capabilities that are preferred to Spotify's recommendations by 73.3% of users. Further, we prioritized an easily usable mobile website with an average user onboarding time of 48.4 seconds, as well as user engagement through a light strobing system that transitions with songs in <100ms response time intervals. Finally, we also support an endless queue via automated song recommendations, volume adjustment through a button interface, as well as safety mechanisms to prevent unsafe usage of the light system and to avoid overly vulgar music content.**

***Index Terms*—DJ, song, queue, DMX capable light fixture, Audio Speaker, Raspberry Pi, web application**

## I. Introduction

THIS project aims to create an all-in-one music platform for events. We replace a costly, difficult to locate & coordinate, and not custom tailored human DJ for social events such as weddings, house parties, and reunions with a comprehensive smart jukebox system that handles song queueing, crowd engagement, and accurately represents users music tastes. Music is the centerpiece of such events and similar gatherings: it is responsible for setting the atmosphere of the event space, which dictates the mood of its guests and allows them to get out of their shells and enjoy themselves (and each other). Thus it is crucial that hosts employ a competent music system, whether a DJ or a jukebox, that will continuously be playing new songs, without allowing for a silent (or even worse, a dull) moment that could derail the entire momentum of the party. It is the system's responsibility to cultivate an exciting environment by playing crowd favorite song requests and the best songs from their personal collection that the guests will actually want to dance and sing along to.

Traditional DJs can only accomplish this well with years and years of experience mixing, listening to large collections of different song genres, and reading diverse crowds, and so there is a shortage of good DJ talent, especially in places outside of major cities with bustling young adult populations. Additionally, even the best DJs will be of no use in a crowd that does not match their target demographic, and to complete the dance floor of the event another professional must be hired, as a lighting designer must create the lighting rig to sync with the music and illuminate the space. Therefore, for most events, which are restricted by a combination of money, time, compatibility, and availability, having a high quality human DJ which garners a sufficient level of satisfaction from its guests is infeasible.

In regards to jukeboxes currently available on the market, they require users to be physically centralized at the device, leading to inefficiencies in contention and the lack of the ability to express opinions on songs other people queued. Additionally, these jukeboxes require the guests to pay money to queue songs, discouraging user engagement and acting to generate revenue for the jukebox company at the cost of a diminished listening experience for users.

The Music Mirror system addresses these problems by providing a custom tailored suite of services at a comparatively low cost, more efficiently, and with a much greater degree of convenience. As a self contained package, it is readily available, and with a flat component cost it is much cheaper than the exorbitant hourly rate of a popular DJ or an alternative solution such as an expensive jukebox, which has significantly less functionality than our system.

The guests of the event will interact with the system through our web application (in most cases on mobile platforms, which are ubiquitous) on which they will be able to queue their favorite songs, request more songs similar to ones that have been played already, generate session song recommendations, downvote songs to remove them from the queue (if it is vetoed by the majority of active users), and provide live feedback on the songs that have been played. As a result the guests will feel more satisfied as they will feel as if their voice is being heard, and be more likely to dance, sing, and enjoy the event as the songs they actually want to hear are being played. This democratization of the song queue will custom tailor the experience for the guests, as it reflects the crowd's tastes better than a single human operator can.

Music Mirror will also use the tracklist of songs queued by the users, their inputs (Upvotes and Downvotes to manually indicate to the system what songs in the queue they liked or didn't like) to insert songs of its own to the collective queue through the mentioned session recommendation feature. The system will blend these characteristics to create comprehensive music choices that not only support the interests of the audience, but are novel and potentially new songs for the users. This will be accomplished using a two-tier recommendation system, pairing Spotify's API recommendation endpoint with a clever seed sampling model that utilizes the live user feedback.

Finally, Music Mirror will operate its own lighting fixtures via the DMX protocol automatically, manipulating the warmth, colors, strobing, intensity, and overall pattern of the lights to suit the atmosphere and the characteristics of the music currently playing. This will add the final dimension of engagement to our comprehensive system that does not come standard with a regular DJ.

Full-scope physical automated music systems similar to Music Mirror are not publicly available, and the archetype is a novel concept in the general market. However, there are other computer DJs that generate song recommendations (like the Spotify DJ) which exist as pure software, applications that allow human DJs to remotely collect song requests from the crowd and then make a decision on them, and jukeboxes that allow guests to walk up and directly queue songs from the central device itself (and thus, is not much different from a music player app just being open on a tablet that anyone can touch). Music Mirror is the first to combine these services into a single, comprehensive, automated platform, allowing for remote song requests concurrently to be added directly to the music queue, to inject its own novel song choices, and to operate its own lighting fixtures to provide a holistic listening experience.

## II. Use-Case Requirements

The target users of the Music Mirror system are hosts of social events like weddings, bar gatherings, high school reunions, corporate socials, and house parties, where music is a key factor in the overall enjoyment of guests. In such events, it is critical for the music being played to be enjoyed by the event participants, but also to be representative of what the majority of people want to hear. This multifaceted use-case environment guided us in developing our system requirements.

Music Mirror is a much more convenient, cost-effective, and intelligent solution than a traditional human DJ or electronic jukebox. We combine a set of features that allows our system to be incredibly reflective of the event guest's music preferences. Event hosts will be able to simply pay a low flat rate for the physical device instead of spending hours and hundreds (or thousands) of dollars negotiating a time and rate with a real DJ, buying an overpriced jukebox, or settling for an alternative sub-par solution. To accomplish this, we settled on a set of core use-case requirements to guide our development process. The companion web app will be intuitive and quick to acquire and learn to use, and guests will be able to queue songs on their own without external guidance in under a minute. We will support 3 distinct song request formats, as well as the ability to provide live user feedback to alter the queue through vetoes and likes. Two of the three song request formats include song recommendation capabilities that we aim to be preferred to Spotify's recommendations by users. Further, we prioritized user engagement through our light strobing system that transitions with songs and matches the tempo and emotion of the corresponding songs. Finally, we also support an endless queue via automated song additions, volume adjustment through a button interface, as well as safety mechanisms to prevent unsafe usage of the light system and to avoid overly vulgar music content.

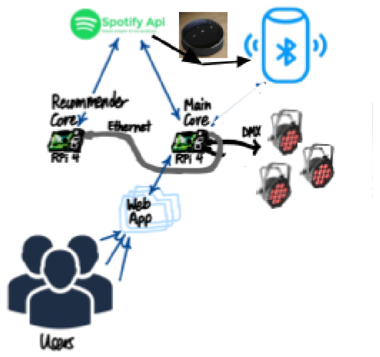


Fig. 1. Overall physical system illustration

## III. Architecture and/or Principle of Operation

The complete physical system of Music Mirror is depicted in Fig. 1. Event hosts will turn on and power the system, and potentially swap out the speaker or the lighting fixture for their own if they have a pre-existing device setup. Event guests, which are the users of our system, interact with it by accessing our web application. These users will type in any songs they want to add to the queue, get song similarity or session recommendations, and press enter, which will forward the song request to Music Mirror. The songs on the centralized queue will be collected from all the users and be displayed on the web app. The rest of the functionality will be operated automatically by the system, without requiring user intervention.

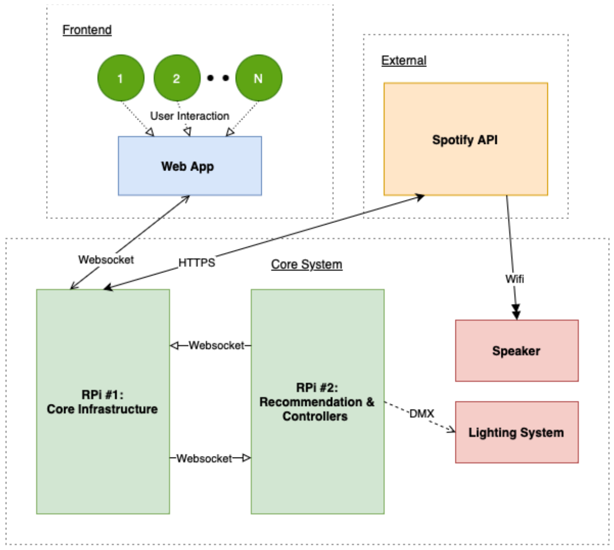


Fig. 2. Overall system diagram

The overall system is composed of four main subsystems: the web application which is the user interface, the main Raspberry Pi 4 ("RPi") which acts as the "brain" of the system receiving and managing the queue, the secondary Raspberry Pi which aggregates user inputs and engagement to generate novel recommendations (using a combination of our song queue data processing and the Spotify Web API),, and the physical interface, which actually plays the songs and flashes the lights. A user will submit a song request on the web app, which will communicate through a Web Socket to the main RPi, which will add it to the queue, forward a formatted request to the Spotify Web API to look for a playable song match, and send the updated queue view back to the web app client. The main RPi will query the Spotify API to receive song data (e.g. the song's genre, theme, tone, tempo, etc.) as well as to actually play the song once it is next up in the queue. The secondary recommender RPi will be available to continuously generate new song recommendations when a user requests, as well as when the queue runs out of songs from users and inserts recommendations of its own.

### *A. Web Application*

The web application (Hosted on the Core Pi) is the medium through which the users will be able to request and vote on songs as well as see the most updated version of the song queue. We used WebSockets to ensure the server holding the queue can initiate messages with the users at any point. This ensures the users always receive the newest version of the queue. The web app also has functionalities to keep track of users' last actions, responsive CSS, and reconnection to the app automatically after being away.

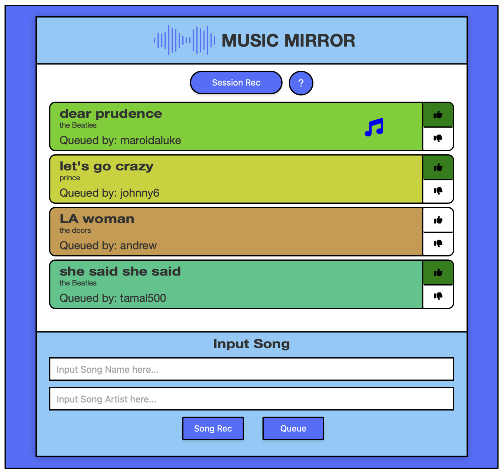


Fig. 3. Frontend Webapp

Above, we see how the web app looks for users of Music Mirror. The system participants can both add a song to the queue and vote for/against others in a user-friendly way.

### *B. Main Raspberry Pi Core*

The main Raspberry Pi (RPi) contains modules responsible for transferring the user song requests onto the queue, pruning the queue for vetoed songs, keeping track of user actions, issuing calls to the Spotify Web API to match requests to playable songs, retrieve metadata about the songs, and playing them on the audio speaker system. Additionally, this RPi core is responsible for semantically matching user song requests to queried resources from the Spotify Web API, to ensure the correct song is being played. Further, this core will house an authorization driver that utilizes Selenium and ChromeDriver to automate user authentication with the Spotify Web API. The main RPi core maintains these microservices using persistent Java processes that are spun up on boot, through Maven applications hosted on them.

### *C. Recommender Raspberry Pi Core*

The second core is responsible for generating song recommendations when queried by the main core. Whenever the queue needs a song recommendation (from either a user request, insertion between user requests, etc.), it will communicate with the second RPi which will build a seed query to send to the Spotify Recommendation endpoint, using our custom sampling and seed generation methodology, which then applies a refined ranking on the returned results from Spotify to provide enhanced recommendations. This will also involve housing an in-memory data structure to hold characteristics of songs that have been played, as well as those that have been recommended by our model. Additionally the core transmits the song to our DMX lighting control program hosted on this pi (Reccomender). After receiving the song in a similar fashion to the recommending program, it sends lighting signals to our physical lights [Described more in Physical Interface (Audio & Lights)]

### *D. Physical Interface (Audio & Lights)*

The system is highly modular and can connect to any external bluetooth or AUX speaker. This is accomplished via the Spotify Connect functionality, which allows us to control a wifi-connected speaker via the Web API. To further increase our modularity, we will be connecting with Spotify via a wifi-based audio streamer, which will then allow us to route the streamed audio to a bluetooth or directly-wired speaker. This essentially allows us to widen our possible speaker choices from just wifi-based speakers, which are expensive and more difficult to find, to virtually any external speaker, as the dominant speaker connection methods are bluetooth and physically wired.

The system's lighting controller operates the lighting fixtures included in the system via DMX signals generated by a long-running Java process propagated through an ENTTEC DMX USB Pro converter. The lighting controller selects from different sets of colors based on the characteristics of the current song (acousticness, danceability, valence, energy) pulled from the Spotify Web API, and modulates the strobing frequency of the lights in real-time with tempo changes throughout the song. Additionally, the ENTTEC DMX USB Pro translates between the USB standard from the Raspberry Pi to generic DMX signals, allowing Music Mirror to be connected to any DMX-capable lighting fixtures that our users may already have.

## IV. Design Requirements

To satisfy the use-case requirements there are several design requirements covering both the hardware and software (as well as the distributed system networking) aspects of the Music Mirror system.

The primary method of interaction between the users and the system is the web app, therefore it must be responsive as well as easy to understand and use. Hence the latency from placing a song request on the web app to the centralized collective queue, and then pushing the updated view of the queue back to the web app client must be under 1 second to be quick and to prevent users from being frustrated using the app. Additionally, it must take new users to take less than 1 minute on average to learn how to use the web app on their own. This will lower the barrier of entry and ensure that as many guests as possible are accommodated by the system.

The system must have a sufficient capacity to fulfill the use-case of an average sized social event. Primarily these consist of gatherings such as weddings, reunions, and parties. As a result our system needs to support a network of 100+ concurrently online users (interfacing through instances of the web app), as the average size of an American wedding is 75-150 guests. Furthermore, the queue must hold at least 100 songs, to reach the target of a 6 hour average reception at 3.5 minutes per song. We also need to ensure that the songs users request are actually the ones being played, so we require an 80% accuracy in semantic matches between the user requests and the actually queried Spotify resources.

User satisfaction is also a critical consideration, hence our system must ensure that the novel song recommendations it produces are high quality. Therefore our target user approval of the generated recommendations is that 75% of users prefer our recommendations to Spotify's naive recommendations. This will assure event hosts that their guests will be enjoying the songs that they are surprised with, with a small margin of error, and shows that our models introduce novelty to existing solutions.

The lighting must always be in sync with the music that the system is playing at the moment. This serves to make the guest experience feel immersive and coherent, and impress users with a more complete event. Hence our target is to have a <1 sec response time between our generated light signals and resulting light effects, in order to ensure that the lighting always tracks the currently playing song accurately.

Additionally, we require some new updated features since our last report at mid semester, which is that our system supports an endless queue, meaning that music will always be playing regardless of the number of users currently in our system. We also aim to support safe volume adjustment through a button interface.

Summary of quantitative requirements:

| **Specification** | **Target Value** |
|---|---|
| Web App to Queue Latency | < 1 sec |
| User Web App Onboarding | < 1 min |
| User Network Capacity | > 100 users |
| Song Queue Capacity | > 100 songs |
| Semantic Match Accuracy | > 80% |
| Song Recommendations | 75% Preference |
| LED Behavior music match | < 1 sec response time<br>Accurate song classification |

## V. Design Trade Studies

### A. *Using WebSockets rather than HTTP*

We decided to go with WebSockets over HTTP for two reasons. The first is that we want for both the clients and server (Raspberry Pi) to have the ability to initiate communication. The client needs to be able to request songs and the server needs to be able to update the client queue sometimes independently of client requests. An example of the server needing to update on its own is when it recommends songs to the clients. We understand that the server can still do that in the HTTP protocol, but that brings us to our second reason: we want real-time communication between the client and server. Our app maintains a real-time queue for songs to be played and songs to be removed from the queue. So users must be looking at an accurate representation of what the current state of the server is. So if one user queues a song or puts the final dislike vote to remove a song from the queue, we want it to be immediately updated for everyone. The best way to support all of this functionality is through WebSockets [1]. So even though WebSockets are harder to implement than HTTP requests, it allows us to have real-time updates to all users.

### B. *Choosing the veto consensus protocol*

When choosing the consensus protocol we thought about who should have a say and how we could make that happen. Here are a few possibilities we thought of: everyone with access to the website, everyone who was ever at the event, everyone who is currently at the event, and everyone who is currently at the event and interacting with the app. We decided that we wanted only people who are currently at the event (since they are the only ones hearing the music) and only the people interacting with the app (since they are the ones who are actively voting). So to accomplish both of these we decided to:

1. only host the website on a local host for the wifi so only people on the wifi can access the website.
2. Implement a heartbeat system to check what users have interacted with the app in a certain period of time. So user's votes will not count if they have not interacted with the app. Once they are again active their actions will be re-added. The way they can be active is any interaction with the page more than just being on the screen.

We chose 30 minutes (about 10 songs) because people won't constantly be on their phones during Events. Also every 30 minutes our Spotify token updates so this is also a convenient spot to mark users inactive to keep our async timing functions to a minimum. If people cared what songs were playing next, they would check at least once for every few songs playing. We also have to consider what percentage of votes are needed to remove a song from the queue. Since we already have narrowed down the votes that count to only users that have interacted with the app for the last 30 minutes we know that they have had the chance to look at the soon-to-be-played songs, so we think that a majority rule would work best. If there are more dislikes than likes for a song then it will be permanently removed from the queue.

### C. *Using a custom recommendation system*

As mentioned, the Music Mirror system will incorporate a model to generate novel song recommendations for the users. A naive vanilla solution is to simply use Spotify's recommendation endpoint. However, we took this a step further due to one core concept: the lack of real time user feedback that goes into the Spotify model. In our system, as more songs are played by the user, and more upvotes and downvotes are provided for the songs that have been played, our system gains critical context and insight into the music taste of our users as well as the broader opinions of the collective audience. We also have live sensor data such as our loudness measure that can be utilized. This real time feedback is something that would not be included into a naive API call to Spotify's model, which takes in an input seed of songs, artists, albums, and other song characteristics such as BPM, tone, acousticness, and a dozen other parameters. Therefore, we have decided to build a second component of the model, which incorporates this real-time feedback to generate more effective seeds to be passed into this model. For example, if a user specifies that they want to hear a song that is similar to the last 5 songs played, how do we accomplish this? There is no input to the Spotify model that would allow us to distinguish between which of these 5 songs resonated the most effectively with the audience. So, we implement a custom sampling mechanism that takes a weighted sample of the song characteristics that is directly correlated to the approval of the songs (ie. the number of upvotes or downvotes each song has). Further, we will include our physical measures (ie. the noise sensor) into this seed generation as well, adding another dimension of live feedback. This initial filtering provides much better input data to the Spotify model, in turn generating better song recommendations that are more representative of the collective event opinion.

### D. *Choosing the semantic matching algorithm*

A core tradeoff that we identified was the differences between different semantic matching algorithms. We tested with three different techniques, a simplistic string comparison, a 1-gram character model, and using an embedding transformer model. To analyze the performance of each, we considered system performance in terms of latency and memory usage, as well as matching accuracy between expected matches and expected failures. We found that the simplistic string comparison was too inefficient, but the 1-gram character model and the transformer model met our accuracy requirements. However, it was noted that the transformer approach utilized a lot of system memory and had a slower latency, which meant the system performance was worse than the 1-gram approach. But with the 1-gram model, our accuracy wasn't quite as good as the transformer, despite the requirements being met. Therefore, we ended up choosing the 1-gram character model for the default setting, but allow users to still utilize the embedding model technique if they prefer higher accuracy versus more user capacity and better system performance.

Now on to the actual way we chose to do the veto. At the start of the second 30-minute period, every 30 minutes the backend will remove likes for users that have not been removed from an inactive list, making sure we only remove likes/dislikes once. We then add all users to the inactive dictionary for the next 30-minute period. Users are taken off this hashmap when they perform a page interaction on the front end which gets sent over. We realize this is not a true 30-minute timeout. If a user interacted with the page at the start of the first 30-minute period and then did not interact until 59 minutes later their likes would not get removed. We chose to do it this way after considering things like JavaScripttiming functions and keeping every single last user action on the backend and somehow checking those every 30 minutes. The second one would have put too much unnecessary stress on the backend application. The issue with the first problem is that when users go away from their screen on some phones (either locking their phone or even just going to a different app) the JavaScriptcode stops leaving the timing function useless. Having the backend check user interactions in 30 minute buckets gives us the best of both worlds (simplicity and reliability).

### E. *Choosing the DMX signal generation library*

Multiple different DMX signal generation software packages were considered when building our real-time adaptive lighting controller. These included the Open Light Architecture framework, PyDMX, and native DmxPy, in addition to the Java ported DmxPy version we ended up using. While all of these libraries were capable of interfacing with our ENTTEC DMX USB Pro converter and propagating control signals, we found the Java version of DmxPy to be best suited for our needs. While the other packages boasted more powerful features that could potentially allow for more complex lighting orchestration, they required more dependencies and were much harder to learn how to use and set up. As our lighting controller would be manipulating the DMX signals itself, we found that the fine-grained and direct channel controls provided by the DmxPy library were sufficient. Additionally, as we made the lights match the tempo of the music at the beat level, we found the more lightweight & quicker DmxPy to work best. Finally, as the rest of our code base and Raspberry Pi communication protocols we had implemented were in

Java, we found the Java port of DmxPy to integrate the most seamlessly with our system.

## VI. System Implementation

Below, we discuss the system implementation, all of which is housed nicely in a 3D-printed casing that can be seen below. We split our discussion into the core subsystems of Music Mirror.

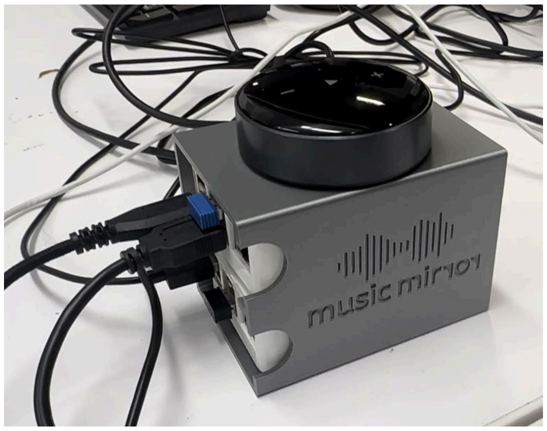


Figure 4. Music Mirror Casing

### A. *Web App (Frontend)*

As shown in Fig. 7. the web app will be hosted on the Raspberry Pi. We are using the Spring Boot chat app [2] to serve as a starting point for the web application because it has a working implementation of Web Sockets using Java Springboot. It starts the WebSocket in the Java backend and can listen for events and messages that happen through the connected JavaScript that the users will be able to interact with through the HTML. The frontend has these functionalities:

- Web Socket communication with the backend. The front end uses Java Springboot's Web Socket by initializing with SockJS and using that socket connection to subscribe to a bunch of actions the backend can make to send messages to the front end. These are actions like song removal and queue updates which will then call specific functions on the front end to update the queue that all users see. This communication also works the other way in that the JavaScriptfunctions can send messages over this socket connection to invoke specific functions in the backend Java code. Examples of this would be liking, queueing, and sending user activity updates.
- Web socket reconnection when users are away and Web Socket gets disconnected. Of course, we want users to be able to go on different apps and close their phones to go dancing, but during that time their WebSocket could lose connection because of the JavaScriptcode stopping execution. To combat this, once the user comes back on to the screen we reconnect the socket and load all user progress back + what song queuing they missed.
- Responsive and colorful UI components. By comprising most of our CSS with flex containers, we are able to fit our app to any width/height screen within reason. Also when the screen width is too small for the text queued, our app uses an animation to have the song scroll for users to see the whole song name rather than making the text very small. For colorfulness, users are assigned one of 15 diverse colors (not blue since that is for the music mirror queue) randomly which will be the color of the song element that is queued. The Music Mirror recommendations will always be blue so they stick out amongst other queues.

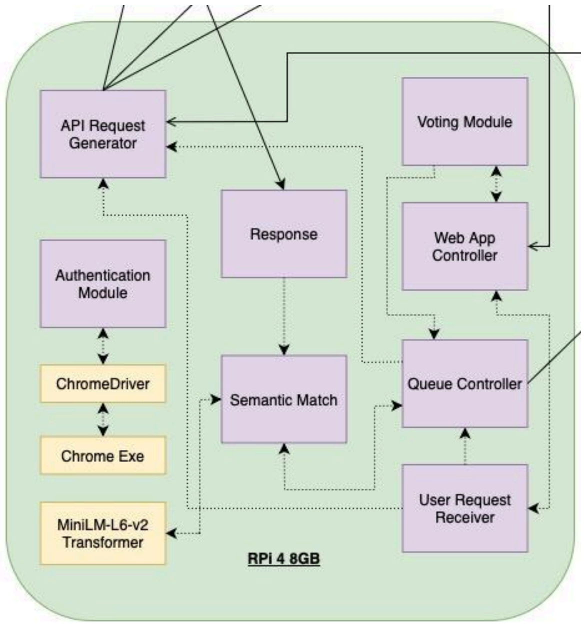


Fig. 5. Main Raspberry Pi core

### B. *Main Raspberry Pi Core*

#### 1. *Queue Controller*

This is our backend for the web application that uses all the other modules seen in Fig. 4 (besides Web App Controller) to provide our backend functionality which is:

1. Keep the queue in a ConcurrentLinkedQueue data structure since multiple requests will be added at the same time. This will hold the songs as well as votes for and against them which will be held in a concurrent list
2. Interact with this song queue to mark the current song that is playing so it can queue it on the Spotify API and send it to the lighting system. Also gets the song requests and song resources from Spotify to update the queue and show users immediately after.

3. Keep track of users in ConcurrentHashMap that has a key of user_id and a value list that holds votes against specific songs
4. Use another Concurrent dictionary to map queue_id to the song object inorder to have O(1) queue removal on the backend. All data structures will be sharing the same song objects, not copies, to ensure correctness and space efficiency
5. Keep track of users that have yet to send a heartbeat for this 30 minute period. If a user's heartbeat times out it will mark all of their votes as not counting and adjust each song accordingly. Right when they interact with the page again, their votes will be added back to the songs still on the queue
6. Continuously listen for new users through Web Sockets controlled in the User Request receiver to add them to the dictionary and let them start to vote/queue. This works because once the user joins the web page it will send a connection request to the backend to initialize another socket connection and all other user functionalities
7. Any change in the queue it updates the frontend accordingly
8. Use the Authorization model to make sure our Spotify API connection always works by periodically (30 min) refreshing our API key.

2. Semantic Matching & API Request Generator

Although it may seem trivial to find a song on Spotify that a user requests, this is in fact not the case. The Spotify database maintains song data in a very particular manner, and any discrepancies in the way songs, artists, and albums are named

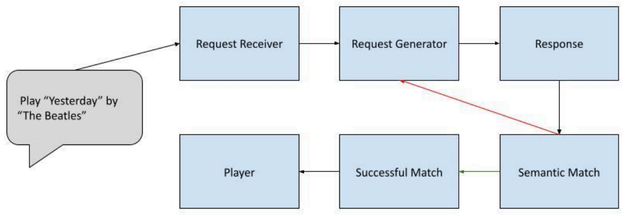


Fig. 6. Semantic Match

may cause unintended difficulty when querying for song resources. For example, say a user requests "Yesterday" by "The Beatles". Well, this song may be directly stored on Spotify as "Yesterday", or perhaps it contains extra information such as "Yesterday (Remastered)", or even "Yesterday (10th Anniversary Edition)". Even further, Spotify's search mechanism is imperfect. There could be many different search results that are close matches, such as "Yesterday - Remastered" by J Dilla or "Lost in Yesterday" by Tame Impala. Obviously, a naive string matching algorithm will not give us a high success rate in actually choosing the songs that the users actually intended to play. That is why we have the system interaction detailed above. We need a semantic matching algorithm to choose between the songs that Spotify's API call responded with, and then if the desired song is still not found, we will need to re-query the endpoint. Thus, we paired Cosine Similarity with a tokenization process to match between constructed strings of the desired and returned song name, artist name, and album in which the song is from. For the tokenization, we support the use of both an embedding transformer, as well as a 1-gram character model. For the transformer, we used the MiniLM-L6-v2 model which takes in an input string and embeds it in a 384 dimension vector space. For the 1-gram model, we simply create vectors representing the character frequency of the input and output strings. Regardless of the embedding choice, we will have a parameterized minimum similarity for us to choose a song, which sits at 85%. We essentially iterate through the Spotify search results, and choose the highest similarity that surpasses the 85% boundary in order to determine a match. Once we reach a successful match, the Spotify response also includes a unique song ID which can then be used to actually access the song resources via the player.

3. Voting Module (Veto Consensus)

As described earlier we will keep a few data structures to keep track of the song queue and songs that should be vetoed. At every user action, we will be updating votes for and against each song. If we find that there are more active likes than active dislikes of a current song then it will be removed from the queue.

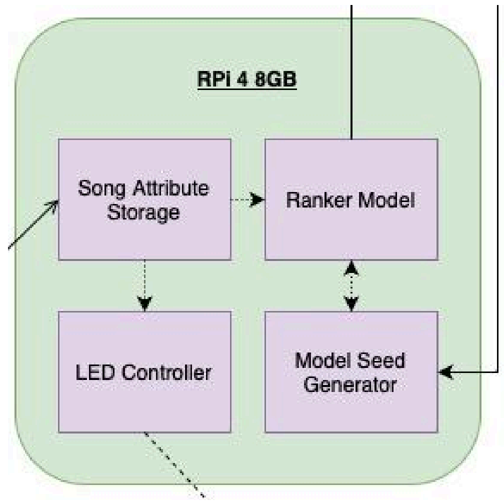


Fig 7: Recommender Pi

4. Authorization Module

To access the Spotify Web API, proper authorization is needed. Essentially, we have a singular Spotify premium account associated with the system that needs to allow the system to access its resources. Typically, because this is a Web API, it would be implemented via some graphic interface that can be displayed to a user. Once you start up the system, an authorization request is sent to Spotify to obtain an authorization code that will be used to generate access tokens. However, Spotify's response to the authorization request is a redirection to a callback URI, where the user can physically click the proper approvals and proceed. However, our device needs to be able to handle the auth process solely on the RPi core because the system itself is the 'user' in the context of the API and we don't have a physical user interface where we could access the internet and follow the callback URI. Therefore, we accomplish this process by using Selenium web driver, in accompaniment with ChromeDriver to automate this authentication process. The driver itself attaches to the callback URI response, and then clicks on the necessary buttons to approve of the needed provisions for the system. Following this, the session is redirected back to our server. This authorization process only needs to occur once, and then the remainder of the system utilizes a returned refresh token to then regenerate access tokens.

## *C. Recommender Raspberry Pi Core*

### 1. Song Attribute Storage

To most effectively generate seeds for our recommendation model, we need readily available access to song characteristics and attributes that will be inputs to the model. Therefore, whenever we add a song from Spotify onto the queue, we will also send a request to gather the song's analysis, and will store these attributes in an in-memory map. We do not need to utilize a database because the number of songs in which we will store will not exceed the memory capabilities of the pi. The actual attributes that are stored will be discussed in the next section, but they will be easily accessible for the input generator's use.

### 2. Model Input Generator

As previously mentioned, our recommendation system utilizes the Spotify recommendation endpoint, as well as a clever sampling mechanism to generate the best possible seeds to input into the model. We will have access to 15 different parameters for the model, including: track, genre, artist, acousticness, danceability, energy, instrumentalness, key, liveness, loudness, mode, popularity, speechiness, tempo, and valence. To select the values we will actually feed into the model for a given user request, we will utilize the live user feedback to build an exponentially weighted combination of these attributes for each song being utilized in the seed. For example, if the user requests a song to be played that is similar to the last 5 songs that have been played, then to choose the parameters to build a seed with, we will weight them by the number of thumbs up / thumbs downs they have, with an exponential factor used to parameterize how concentrated the selected values are around the most highly rated of these 5. This is an important distinction than something as naive as a normal average, because this would produce very dull results. To see this, consider the averaging of a song's BPM. If you had 5 songs, 2 with very slow BPMs and 3 with very fast, then the average of these would simply be a dull medium paced song. That is why we are interactively using context provided by our users' experience to inform which of these songs we should place the highest weight on. In a way, it is a reinforcement learning approach to improving Spotify's naive recommendations by introducing live feedback on the songs being played and the recommendations provided.

### 3. Refined Song Similarity Recommendation

One of our requirements was to ensure song recommendations that are more refined than Spotify's. Therefore, to accomplish this we used a two-tier model that takes Spotify's generic recommendations and then refines them with some mathematical operations. Essentially, for a song similarity recommendation we do similar to the above and generate a seed to feed into Spotify's song recommendation endpoint. This will return 20-30 recommended songs, primarily based off of their proprietary user data as well as song characteristics. However, it is then our job to further refine the results to ensure the returned song is the most 'similar' to the input song. Thus, we developed a model that maps songs to a 9-dimensional vector space, where each dimension represents one of the following characteristics: acousticness, danceability, energy, instrumentalness, liveness, loudness, speechiness, tempo, and valence. Now, our similarity problem has become a math problem. To find the most similar song to the input of the returned Spotify recommendations, we then apply a standard L2-norm minimization to find the song with the shortest distance away from the input song in this vector space. We used min-max normalization during this process, to avoid issues from the different ranges of the 9 characteristics. Once we find the song with the minimum distance, we deem it as the most 'similar' and return it as the refined recommendation.

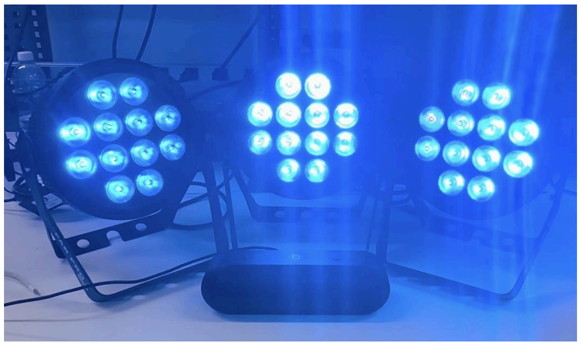


Figure 8. LED System and Speaker

### 4. LED Controller

The lighting fixtures attached to the recommender RPi were controlled via DMX signals transmitted over a DMX cable. These signals will be generated on board the recommender RPi using a Java program which controls the different channels (independently controllable groups of LEDs) of the fixture by using the DmxPy interface (ported to Java) to generate specific DMX outputs. The DMX channel signals, which control the behavior and colors of the lights, were determined based on the characteristics (acousticness, danceability, valence, energy) of currently playing songs, which were derived from the Spotify Web API. (Valence is defined as the level of musical positiveness conveyed by the song)

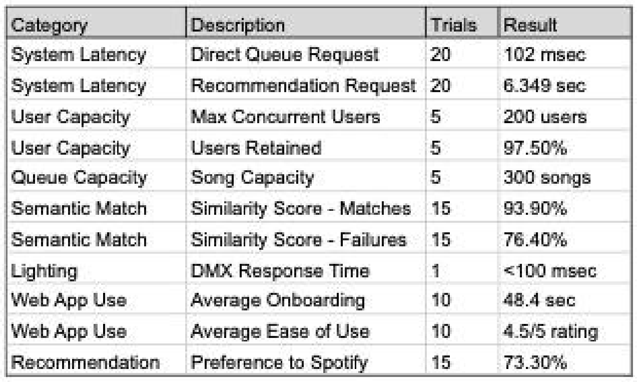
| Type | Color Scheme | Characteristics |
|---|---|---|
| 0 | Full Range | default |
| 1 | Acoustic/Warm | high acousticness |
| 2 | Dance/Disco | high danceability |
| 3 | Positive/Upbeat | high valence |
| 4 | Sad/Moody | low valence |
| 5 | Energetic | high energy |

Additionally, the lighting controller would maintain an internal timer throughout the duration of the song's runtime. This would allow the controller to modulate the time delay in between color changes in real time, such that the lights would match the bpm as the currently playing song progresses throughout its different tempo sections. Below is the formula for the time delay in milliseconds as used by our lighting controller, with a 10ms offset for switching the lights off after the current beat:

timeDelay = Math.round((60 / tempo) * 1000) - 10

## VII. Test, Verification and Validation

The software, hardware, and networking aspects of the



| Category | Description | Trials | Result |
|---|---|---|---|
| System Latency | Direct Queue Request | 20 | 102 msec |
| System Latency | Recommendation Request | 20 | 6.349 sec |
| User Capacity | Max Concurrent Users | 5 | 200 users |
| User Capacity | Users Retained | 5 | 97.50% |
| Queue Capacity | Song Capacity | 5 | 300 songs |
| Semantic Match | Similarity Score - Matches | 15 | 93.90% |
| Semantic Match | Similarity Score - Failures | 15 | 76.40% |
| Lighting | DMX Response Time | 1 | <100 msec |
| Web App Use | Average Onboarding | 10 | 48.4 sec |
| Web App Use | Average Ease of Use | 10 | 4.5/5 rating |
| Recommendation | Preference to Spotify | 15 | 73.30% |

Figure 9: Summary of Test Results

Music Mirror system were rigorously tested to verify intended behavior and validate the quality of our submodules. The objective was to confirm that the user experience is intuitive, smooth, and satisfying, and that the system can stand up to the stressors of our target use-case scenarios. Below, we will go in depth in regards to each test we performed, all of which are summarized in the table above.

### *A. Tests for Web App to Queue Latency*

Timestamped test song queue requests were issued from a mock web application instance to the DJ system, and were used to measure the time elapsed between inputting a request and seeing the corresponding queue update return to the web app. Over a set of 20 trials of direct queue song requests the average latency was measured to be 102 ms, significantly faster than our 1 second roundtrip time benchmark. Our latency testing verified that our system would feel responsive and seamless for users of our web application. This ensured that our user operation throughput would remain high, and that event guests would not be discouraged or frustrated when engaging with our system.

### *B. Tests for User Web App Onboarding*

In order to test the intuitiveness of our web app we planned to collect data using real survey participants to determine how quickly it takes an average new user to learn how to queue songs and access the different functions of the app. We accomplished this by surveying fresh users who have never been exposed to our web app and measured how long it takes them to feel confident about their understanding of it and be able to make song requests and navigate the queue on their own. We also measured their satisfaction with the ease of use of the system. The target amount of time for this onboarding was less than 1 minute. We met these goals by interviewing 10 participants, and found that their average onboarding time was 48.4 seconds. More specifically, the minimum onboard time was 21 seconds and the maximum was 83 seconds. We found that the average ease of use rating was 4.5/5. This was fantastic to see and confirmed our use-case that our web app must be easy to use as well as visually appealing and enjoyable.

### *C. Tests for User Network Capacity*

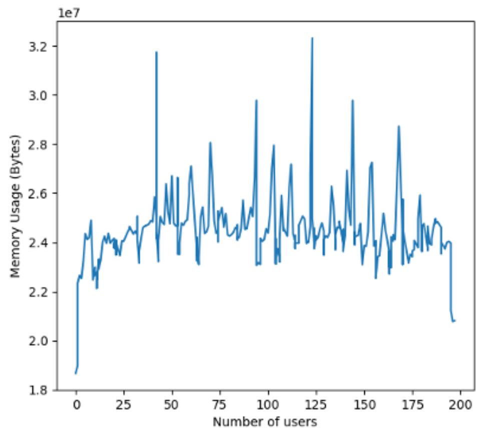


Figure 10: Memory Consumption vs. Users

To test network capacity a barrage of stress tests were conducted to determine whether or not the critical user interaction functions of our system hold up in the presence of many concurrent users and a large volume of incoming requests. In order to accomplish this, increasing numbers of dummy users (up to 200) were connected to the system, and we verified that the system can manage these large amounts of websockets and accept requests from any of them at any time, without decreasing system performance or running out of memory. Additionally, we will send multiple concurrent requests to the system all within one second of each other, and verify that none of these requests are dropped and that the system produces the correct behavior manipulating the queue. This will ensure that our system will be able to accommodate our use-case, which involves large numbers of guests at an event issuing requests at random times. We ran this test with a script that simulated the tests described above, and found the system capable of handling 200 concurrent users. This exceeded our target goal of >100 concurrent users. Further, we found our memory consumption to be fairly independent of the number of users, which is due to our lightweight design as well as explicit garbage collection processes throughout the system. These memory results are described in the above figure.

### D. *Tests for Song Queue Capacity and Veto System*

To test our song queue we used a shell script to simulate different loads of users performing actions that our clients would. Over the course of 5 trials, the entire Music Mirror system was rebooted, and using a set of 5 simulated users queuing 60 songs each the effective queue capacity was verified to be over 300 songs. This far exceeded our target of 100 songs (~6 hours at 3.5 minutes per song), capable of maintaining over 17 hours of play time, much longer than any anticipated application of our system. Outperforming our song capacity benchmark ensured that Music Mirror's collective song queue was sufficiently robust and voluminous in order to meet the requirements of our use case. This way event guests will be able to queue songs to their heart's content and keep their party going late into the night.

Music Mirror's veto system was visually inspected by connecting multiple users and attempted to Dislike & veto songs from the queue. We verified that our users would be able to prune the collective queue fairly and efficiently. This would ensure the best overall listening experience for guests, as well as improve the quality of the recommendations generated.

### E. *Tests for Song Recommendations Quality*

Because song recommendations are a subjective matter in nature, we tested the quality of them with user feedback surveys. To accomplish this, we had 5 in person interviews to present users with an input song, our similarity recommendation, and then Spotify's naive recommendation, and asked users which of the recommendations they preferred. For each person, we had 3 trials of this process. For each of the 3 trials, we tested with an alternative, rock, and rap song to see how the recommendations performed across genres. We saw that 11/15 trials resulted in our recommendations being preferred, which is a 73.3% preferred percentage. Although this fell slightly short of our >75% goal, we were happy with these results as recommendations are very subjective, so we felt this metric justified our improved ranking mechanism sufficiently.

### F. *Tests for LED Behavior Matching Music*

The lighting fixture's LEDs were visually inspected (checking that the color ranges displayed match Spotify's song attribute data) over a set group of songs played to verify that the patterns and colors they are emitting match the genre and tone of the songs playing. In addition, the response time of the DMX lighting fixture system was verified to be quicker than 100ms, in order to support a wide range of different song tempos. Songs with a high level of acousticness displayed warm colors, songs with a low valence score (the measure of musical positiveness of the song) displayed cooler colors, and songs with a high level of danceability or energy utilized a wider range of the available colors. Additionally, the lights were synchronized to a 120 bpm and a 140 bpm metronome, ensuring a sufficient level of fine-grained control over the DMX control line. These tests confirmed that Music Mirror's lighting system would be able to properly classify the genre of the currently playing song and match its tempo in real time, enhancing the user experience. This increased level of coherence would elevate the perceived level of professionalism of events using the Music Mirror system.

## VIII. Project Management

We have been maintaining efficient systems to keep track of our work progress and communicate our ideas, which are discussed below. Apart from these, we also have scheduled meeting times for Zoom calls every Wednesday and Friday evening for higher level design choices and progress. .

### A. *Schedule*

The schedule is shown in Fig. 8.. We have been using this schedule to guide and track our work progress.

### B. *Team Member Responsibilities*

| | |
|---|---|
| Thomas | • Light controller<br>• Web app & internal data structures<br>• Queuing/voting functionality |
| Matt | • User graphical interface<br>• Web app communication with backend<br>• Queuing/voting functionality<br>• Raspberry Pi communication between systems |
| Luke | • Recommendation RPi implementation |

| | |
|---|---|
| | • Authorization Driver<br>• Semantic matching<br>• Speaker pipeline connection |
| All Members | • System Integration<br>• User satisfaction surveys<br>• Testing |

## C. *Bill of Materials and Budget*

Fig. 11. Bill of Materials and Budget

| Description | Model | Manufacturer | Project Cost | User Cost |
|---|---|---|---|---|
| Raspberry Pi 4 | 8GB | CanaKit | $0 | $75 |
| Raspberry Pi 4 | 8GB | CanaKit | $0 | $75 |
| Spotify Subscription | Premium | Spotify | $10.99/Month | $10.99/Month |
| DMX Controlled Lights | | | $350 | Varied |
| Any Speaker | | | $0 | Varied |
| Wireless Audio Streamer | Mini Airplay2 | WiiM | $89 | $89 |
| | | | $471.97 | $239+10.99*Months + lights + speaker |

The difference between the project and user Cost is that (for example) we were provided with the Raspberry Pi's so they cost us $0 while they are 75 each to buy if a user were to replicate our system. So the project cost is what we spent while the user cost is what it will cost to build their own system. We labeled the lights and speakers for users as varied because any speaker will work and any DMX-controlled lights will work.

## D. Risk Management

A few of the risks were identified through our project implementation. The main one was us being able to control the lights. The lights were the last part of our system to initially show up plus we could not get the lights to work at first and had to pivot and get extra parts as we understood how the lights worked more and more. We were able to manage this risk by looking for help online and looking for help with a previous team. We knew there were a lot of DMX resources online in Python and a little in Java so we could use those to try and learn more about what we were doing. Also, we knew certain Python scripts worked so if we could not use the Java code we could have pivoted to a Python script. We were also in contact with an old capstone project that used DMX lights and they also gave us debugging advice. Another risk was our Web Socket set up, we originally tried to have our web app run independently of the Raspberry Pi and just have JavaScriptand html for the frontend which connects to our Java code in the core on the backend. Despite many efforts and hours we were not able to get the JavaScriptto JavaWeb Socket set up. So we looked across the internet and found a project with a tutorial [2] that had already implemented WebSockets using Java Spring Boot. Similarly, the Pi communication took some time to learn. For this one we looked up our error codes online to find other people with similar issues and were able to figure it out. So overall we managed our risks by choosing a well-documented project so we knew there was always another option if something did not work.

## IX. Ethical Issues

We took the time to ensure our design handled serious ethical considerations. We will discuss our system's ethical concerns in the context of public health, public safety, and public welfare.

While the potential public health consequences of the Music Mirror project are mild at the worst, there are still some issues that must be taken into consideration. Primarily, since the system operates the lights and sounds of the venue, attendees may be exposed to unsafe volume levels and nauseating or blinding flashing lights. Therefore there is a design tradeoff in determining the system's capacities for volume and light intensity, as louder performances and more vigorous lighting displays may be more entertaining for the users but potentially be unhealthy. Music Mirror addresses this issue by restricting volume to a healthy range, and its strobing frequency to prevent health complications such as epilepsy. Additionally, the energy footprint of the system can have negative effects on the environment. Again larger more complex systems may be more entertaining, but may have a higher energy cost. Music Mirror solves this issue by consuming comparable levels of power to similar music playing and sound systems.

The Music Mirror system must ensure the public safety of the users it affects. The main public safety considerations are that of user privacy and protection from other users & abuse. In regards to privacy, the tradeoff balances the need to collect user data for satisfaction surveying, but protecting the user's privacy. Music Mirror will address this issue by hiding users' data from each other and securely storing their information. In regards to protecting users from each other, Music Mirror will not allow direct user to user communication on the web app, and making song voting anonymous. Also, there is the potential for users to try and queue vulgar music, so we implemented a safety check that will kick a user out of the app if they try to queue a song including a preconfigured list of "bad" words. This ensures safety for people who do not want to consume vulgar content, such as children.

The system must promote public welfare, and not have negative socio-political consequences. Music Mirror must cooperate with the music industry, and encourage healthy music consumption and production. The main potential issue is that of unfairly representing different genres or types of music, which may discriminate against different music fanbases. In order to be favorable to welfare considering the social factors, Music Mirror will completely democratize its song requesting and recommendation service, giving all users equal say, save for host privileges.

## X. Related Work

The Springboot Chat app [2] is similar to how we want to use our WebSockets. That is why we are using it for our WebSockets. This is a real-time chat app with a javascript and HTML frontend and a Java backend so it's very similar to our project.

## XI. Summary

In summary, we learned a lot about system design on our quest to democratize our users' listening experiences. We feel that Music Mirror has the potential to be a staple in future weddings, restaurants, parties, and other music listening venues. We also learned the importance of performance tradeoffs, such as memory consumption versus number of supported users, which were the backbone of our development process. Some future features we envision are not only allowing users to veto a song off the queue, but also to be able to vote on the positioning of a song in the queue. For example, if all users want to hear a song really badly, they can vote for it to be moved to the front of the queue. In addition, we aim to add a provisional feature, which would allow this system to be marketed better as a product. Currently, all users have equal voting weights but say an owner of Music Mirror wanted to still have some administrative control over the queue, then we could make some minor changes in our implementation that would allow for a specific user to have higher weighted votes, giving them more control. Moving forward, we have big aspirations for the future of Music Mirror.

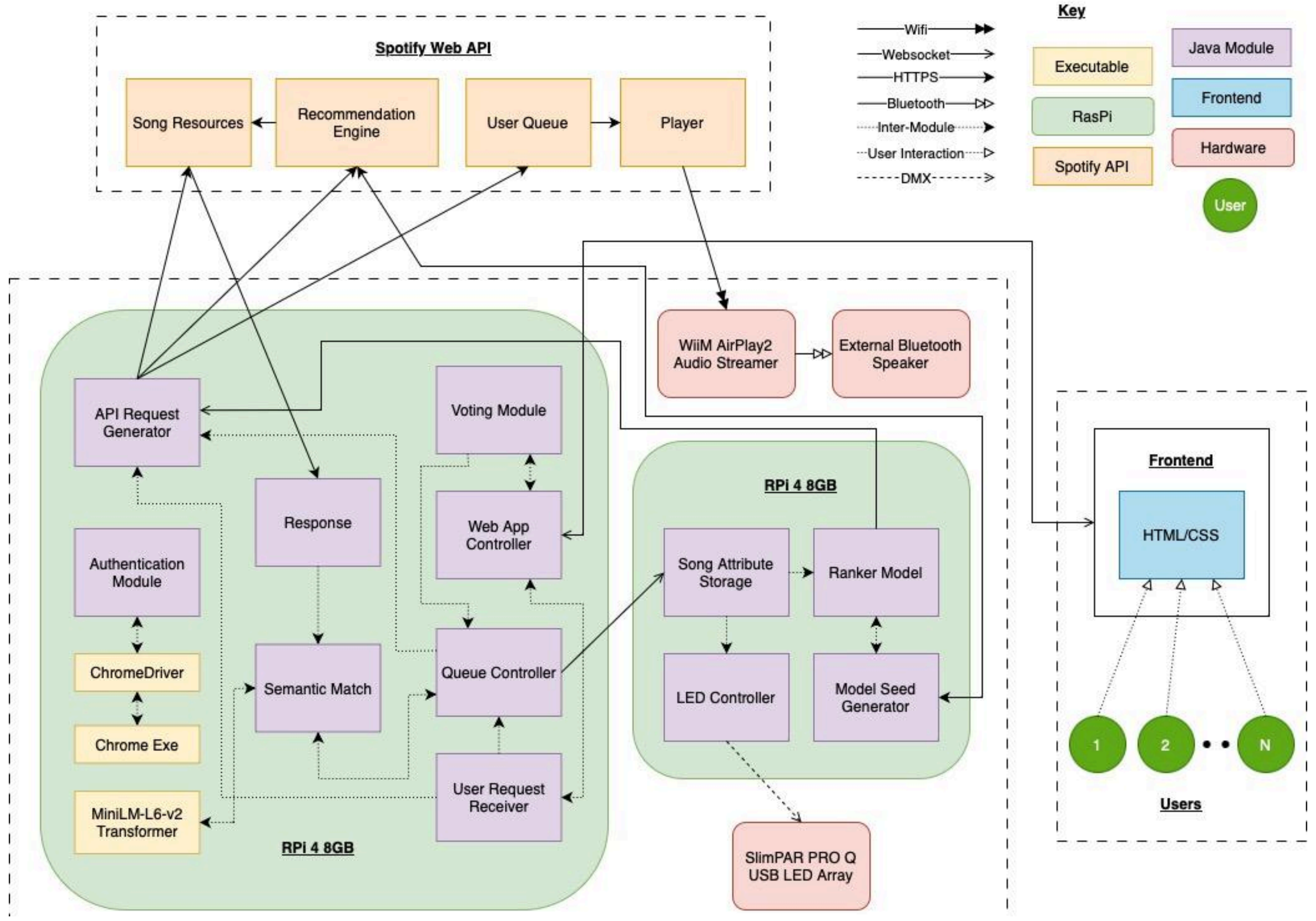
## Glossary of Acronyms

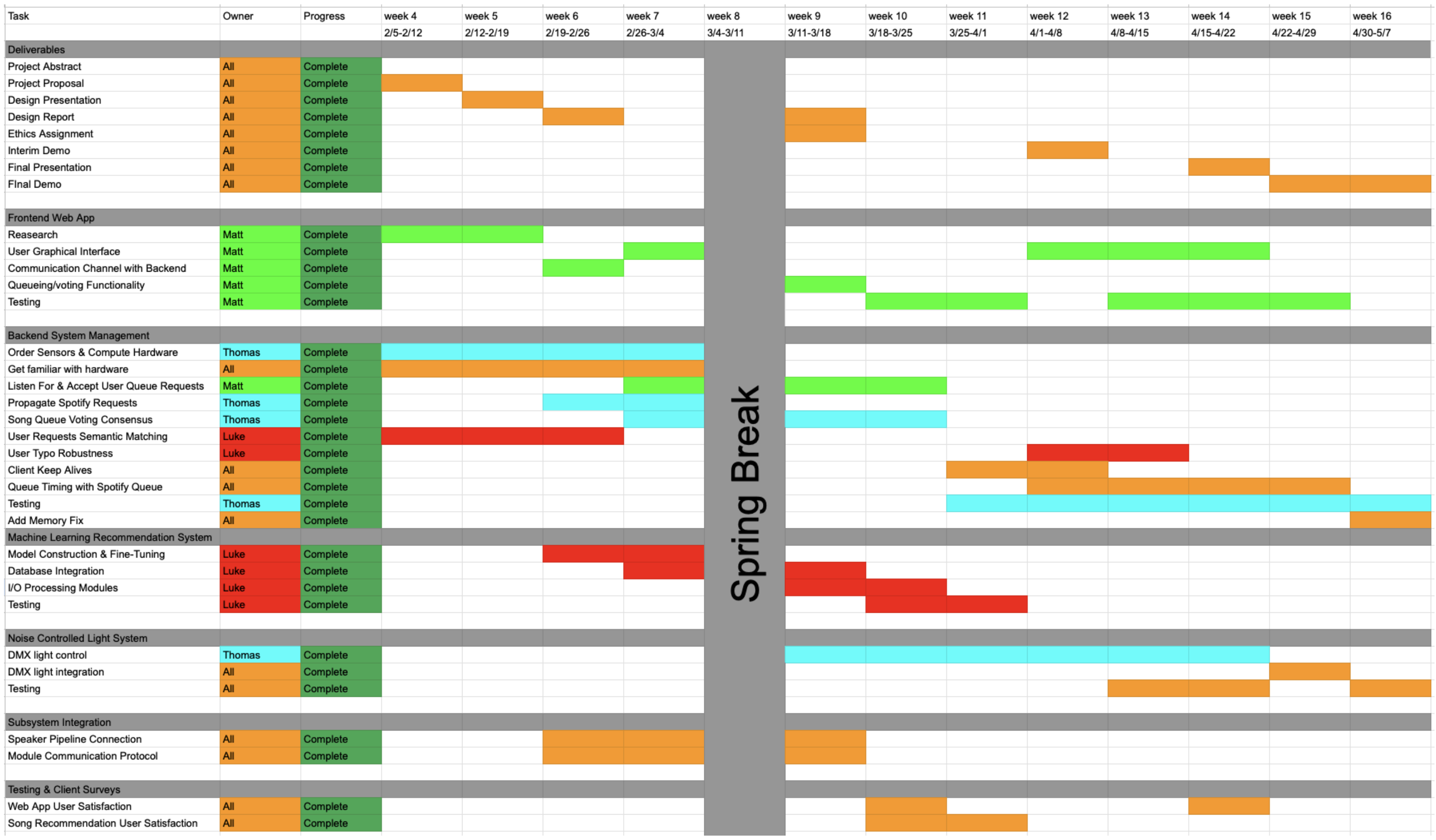
API – Application Programming Interface
DMX – Digital communication standard for controlling lighting fixtures and stage effects
HTML – HyperText Markup Language
JSON – JavaScript Object Notation
RPi – Raspberry Pi

## References

[1] Ably. (n.d.). WebSockets vs HTTP. Retrieved February 15, 2024, from https://ably.com/topic/websockets-vs-http
[2] Bouali, A. 2023. spring-boot-websocket-chat-app. GitHub. https://github.com/ali-bouali/spring-boot-websocket-chat-app.git
[3] rydercalmdown. (n.d.). DMX lights. GitHub. Retrieved March 1, 2024, from https://github.com/rydercalmdown/dmx_lights
[4] Open Lighting Project. (n.d.). OLA on Raspberry Pi. Retrieved March 1, 2024, from https://www.openlighting.org/ola/tutorials/ola-on-raspberry-pi/
[5] Spotify AB. (n.d.). Web API. Spotify for Developers. Retrieved March 1, 2024, from https://developer.spotify.com/documentation/web-api/

Fig. 12. Detailed System Design





| Task | Owner | Progress | week 4 | week 5 | week 6 | week 7 | week 8 | week 9 | week 10 | week 11 | week 12 | week 13 | week 14 | week 15 | week 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 2/5-2/12 | 2/12-2/19 | 2/19-2/26 | 2/26-3/4 | 3/4-3/11 | 3/11-3/18 | 3/18-3/25 | 3/25-4/1 | 4/1-4/8 | 4/8-4/15 | 4/15-4/22 | 4/22-4/29 | 4/30-5/7 |
| Deliverables | | | | | | | Spring Break | | | | | | | | |
| Project Abstract | All | Complete | | | | | | | | | | | | | |
| Project Proposal | All | Complete | 🟧 | | | | | | | | | | | | |
| Design Presentation | All | Complete | | 🟧 | | | | | | | | | | | |
| Design Report | All | Complete | | | 🟧 | | | 🟧 | | | | | | | |
| Ethics Assignment | All | Complete | | | | | | 🟧 | | | | | | | |
| Interim Demo | All | Complete | | | | | | | | | 🟧 | | | | |
| Final Presentation | All | Complete | | | | | | | | | | | 🟧 | | |
| FInal Demo | All | Complete | | | | | | | | | | | | 🟧 | 🟧 |
| Frontend Web App | | | | | | | | | | | | | | | |
| Reasearch | Matt | Complete | 🟩 | 🟩 | | | | | | | | | | | |
| User Graphical Interface | Matt | Complete | | | | 🟩 | | | | | 🟩 | 🟩 | 🟩 | | |
| Communication Channel with Backend | Matt | Complete | | | 🟩 | | | | | | | | | | |
| Queueing/voting Functionality | Matt | Complete | | | | | | 🟩 | | | | | | | |
| Testing | Matt | Complete | | | | | | | 🟩 | 🟩 | | 🟩 | 🟩 | 🟩 | |
| Backend System Management | | | | | | | | | | | | | | | |
| Order Sensors & Compute Hardware | Thomas | Complete | 🟦 | 🟦 | 🟦 | 🟦 | | | | | | | | | |
| Get familiar with hardware | All | Complete | 🟧 | 🟧 | 🟧 | 🟧 | | | | | | | | | |
| Listen For & Accept User Queue Requests | Matt | Complete | | | | 🟩 | | 🟩 | 🟩 | | | | | | |
| Propagate Spotify Requests | Thomas | Complete | | | 🟦 | 🟦 | | | | | | | | | |
| Song Queue Voting Consensus | Thomas | Complete | | | | 🟦 | | 🟦 | 🟦 | | | | | | |
| User Requests Semantic Matching | Luke | Complete | 🟥 | 🟥 | 🟥 | | | | | | | | | | |
| User Typo Robustness | Luke | Complete | | | | | | | | | 🟥 | 🟥 | | | |
| Client Keep Alives | All | Complete | | | | | | | | 🟧 | 🟧 | | | | |
| Queue Timing with Spotify Queue | All | Complete | | | | | | | | | 🟧 | 🟧 | 🟧 | 🟧 | |
| Testing | Thomas | Complete | | | | | | | | 🟦 | 🟦 | 🟦 | 🟦 | 🟦 | 🟦 |
| Add Memory Fix | All | Complete | | | | | | | | | | | | | 🟧 |
| Machine Learning Recommendation System | | | | | | | | | | | | | | | |
| Model Construction & Fine-Tuning | Luke | Complete | | | 🟥 | 🟥 | | | | | | | | | |
| Database Integration | Luke | Complete | | | | 🟥 | | 🟥 | | | | | | | |
| I/O Processing Modules | Luke | Complete | | | | | | 🟥 | 🟥 | | | | | | |
| Testing | Luke | Complete | | | | | | | 🟥 | 🟥 | | | | | |
| Noise Controlled Light System | | | | | | | | | | | | | | | |
| DMX light control | Thomas | Complete | | | | | | 🟦 | 🟦 | 🟦 | 🟦 | 🟦 | 🟦 | | |
| DMX light integration | All | Complete | | | | | | | | | | | | 🟧 | |
| Testing | All | Complete | | | | | | | | | | 🟧 | 🟧 | | 🟧 |
| Subsystem Integration | | | | | | | | | | | | | | | |
| Speaker Pipeline Connection | All | Complete | | | 🟧 | 🟧 | | 🟧 | | | | | | | |
| Module Communication Protocol | All | Complete | | | 🟧 | 🟧 | | 🟧 | | | | | | | |
| Testing & Client Surveys | | | | | | | | | | | | | | | |
| Web App User Satisfaction | All | Complete | | | | | | | 🟧 | | | | 🟧 | | |
| Song Recommendation User Satisfaction | All | Complete | | | | | | | 🟧 | 🟧 | | | | | |

Fig. 13. Gantt Chart