Music Mirror

Luke Marolda, Matt Hegi, and Thomas Lee

Department of Electrical and Computer Engineering,
Carnegie Mellon University

***Abstract*—Traditional social events require an experienced, human DJ to continually mix and play songs, while also queueing more songs for the future. Not only is hiring a DJ expensive and hard to source, it is also risky as the DJ's personal music taste may not align well with that of your guests. With the Music Mirror comprehensive DJ system, hosts will be able to conveniently and cheaply have music playing continuously, chosen directly by their guests, as well as novel recommendations and party lighting operated by the DJ itself. This will afford their guests a more satisfying and engaging experience.**

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

I. Introduction

This project aims to replace a costly, difficult to locate & coordinate, and not custom tailored human disc jockey for social events such as weddings, house parties, and reunions with a comprehensive smart DJ system. The DJ is the centerpiece of such events and similar gatherings: they are responsible for setting the atmosphere of the event space, which dictates the mood of its guests, and using their music as social lubrication, which allows the guests to get out of their shells and enjoy themselves (and each other). Thus it is crucial that hosts employ a competent DJ 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 DJ'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.

The Music Mirror DJ 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.

The guests of the event will interact with our DJ 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, 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 DJ can read the audience.

Music Mirror will also use the tracklist of songs queued by the users, their inputs (Upvotes and Downvotes to manually indicate to the DJ system what songs in the queue they liked or didn't like) as well as their level of engagement for each song collected by a noise sensor to generate novel recommendations to insert songs of its own to the collective queue. The DJ 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 DJ 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 complete the full spectrum of the DJ set.

II. Use-Case Requirements

The target users of the Music Mirror DJ system are hosts of social events that commonly have a DJ and a dance floor, but where the guests aren't attending to see a specific popular celebrity DJ or musical artist. For example, the smart DJ system would not replace a famous human DJ like Fred again.. or Skrillex at a nightclub where attendees are going solely to see that artist, but instead events like weddings, high school reunions, corporate socials, and house parties, where the main focus is not the DJ but it is still necessary to have exciting music playing.

As the DJ itself is not the main reason for hosting the event, not much time, money, and organizational brain power can be afforded to securing a good human disc jockey. Hence, users should find Music Mirror much more convenient, cost-effective, and reflective of their 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, and quickly turn on the system instead of coordinating with and providing creature comforts for an actual DJ. 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. As Music Mirror learns the music tastes of the audience through their song requests and inputs, it will have a greater rate of matching the crowd's preferences than a single host making a judgment about which human DJ's style best suits their needs. The system will also come standard with a lighting fixture system controlled by Music Mirror itself, and so will not require extra time and money for hosts to hire a lighting designer to collaborate with a human DJ to set up the party lights.

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 DJ system, interact with the DJ by accessing our web application. These users will type in any songs they want to add to the queue and press Submit, which will forward the song request to the Music Mirror DJ. 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. 1. Overall physical system illustration

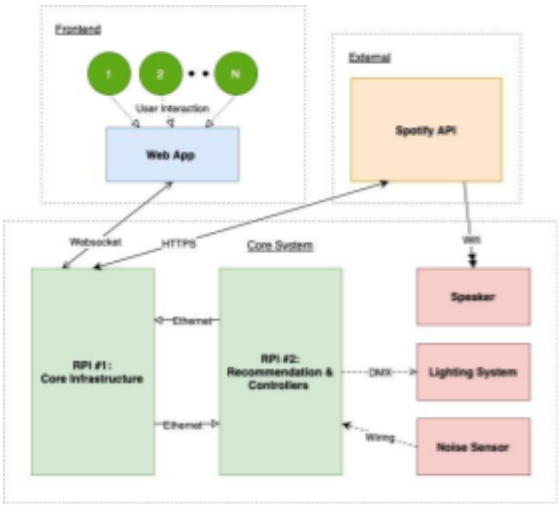


Fig. 2. Overall system diagram

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, 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 monitoring the songs on the queue, as well as how loud the crowd is (correlated with how much they are enjoying the currently playing song), to continuously generate new song recommendations. The recommender will insert these new songs occasionally onto the collective queue.

A. Web Application

The web application is the medium for which the users will be able to request and vote on songs as well as see the most updated version of the song queue. It will use WebSockets to communicate with the backend controlled by the Main Raspberry Pi core so that current users can be accurately represented for vetoing and all users can see the same most updated queue.

Fig. 3. Frontend Wireframes

You can see a picture of how the website will look in Fig. 3. above. The users can both add a song to the queue and vote against others in a user-friendly way.

B. *Main Raspberry Pi Core*

The main Raspberry Pi contains modules responsible for transferring the user song requests onto the queue, pruning the queue for vetoed songs, and 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 core pi is responsible for semantically matching user song requests to queried resources from spotify, to ensure the correct song is actually 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.

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 pi which will build a seed query to send back to the Spotify Recommendation endpoint, using our custom sampling and seed generation methodology. 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. In addition, the processor also retrieves crowd loudness data in decibels over a USB connection to the noise sensor. Additionally the core transmits DMX control signals for the attached physical lighting fixture using a Python program leveraging the DmxPy library over a USB-to-DMX connection.

D. *Physical Interface (Audio & Lights)*

The system is highly modular and can connect to any external bluetooth 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 lighting fixture receives DMX transmissions from the main RPi core and is powered using a standard outlet connection.

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 smart DJ 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 snappy 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 DJ system.

The DJ system must have a sufficient capacity to fulfill the use-case of an average medium to large 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.

User satisfaction is also a critical consideration, hence our system must ensure that the novel song recommendations it produces are high quality. Therefore the DJ's target user approval of the generated recommendations must be 95%. This will assure event hosts that their guests will be enjoying the songs that they are surprised with, with a small margin of error..

The lighting system must always be in sync with the music that the DJ 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 80% accuracy for matching the lighting pattern to the current music and atmosphere of the event space.

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
Song Recommendations	95% Accuracy
LED Behavior music match	80% Accuracy

V. Design Trade Studies

A. Using WebSockets rather than HTTP

We decided to go with WebSockets over HTTP/HTTPS 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 we decided to use (the more complicated implementation) webSockets for those two reasons.

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.

We chose 15 minutes (about 5 songs) because people won't constantly be on their phones during weddings. So if people cared what songs were playing next, they would check at least once during every 5 songs. We also have to consider what percentage of votes are needed to remove a song from the queue. Our web app is designed to only let users vote against a song, so if they do not vote against the song then we count it as a vote for. Since we already have narrowed down the votes that count to only users that have interacted with the app for the last 15 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 >= 51% of people vote against the song then we will remove it 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.

VI. System Implementation

A. Web App

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. There are already event listeners for users joining, leaving, and sending messages. We will need to edit those. We will edit the backend functions to keep track of the song queue and users' votes (both that currently count towards the veto and not). Our backend will:

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. Keep track of users in ConcurrentHashMap that has a key of socket_id and a value list that holds votes against specific songs, and the last time they sent a heartbeat.
3. If a user's heartbeat times out it will mark all of their votes as not counting and adjust each song accordingly

4. Continuously listen for new users through web sockets to add them to the dictionary and enforce their votes
5. When there is a change in the queue it updates the frontend accordingly
6. Communicate with the other modules to get recommended songs

We will edit the front end to look like the wireframes in Fig. 3.

The users can both add a song to the queue and vote against others in a user-friendly way.

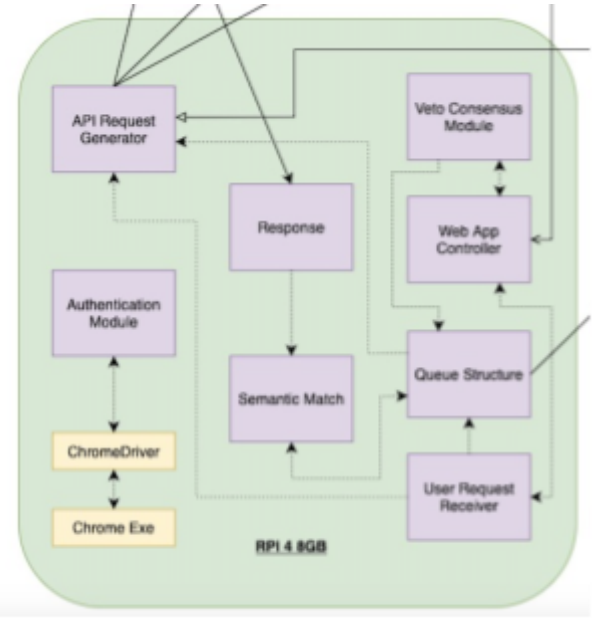


Fig. 4. Main Raspberry Pi core

B. Main Raspberry Pi Core

1. User Request Receiver

The main RPi receives incoming song requests from clients on the web application via a JSON formatted data payload. This request data is accepted by the User Request Receiver server hosted on the main RPi, which is continually listening for communication over the websockets, and forwarded to the Queue Structure Manager node. The User Request Receiver also sends this request data to the API Request Generator to begin the process of identifying the corresponding intended song on Spotify's internal database.

2. Queue Structure Manager

The Queue Structure Manager node retrieves successfully accepted song requests and adds them to the internal collective song queue. The centralized song queue utilized a Concurrent Linked Queue data structure for storage and manipulation operations in local memory, as the songs are String entries and are relatively lightweight since the actual audio files themselves will not be hosted on this RPi. The Queue Structure Manager performs all song addition and removal operations as requested by the User Request Receiver and Veto Consensus modules in collaboration with the web app backend service.

3. 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 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

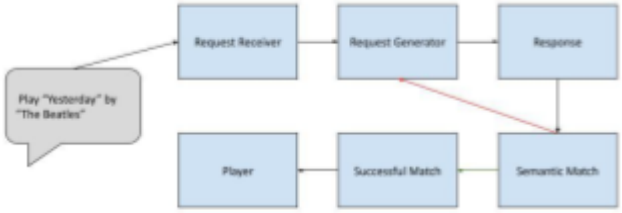


Fig. 5. Semantic Match

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 will be using Cosine Similarity to match between constructed strings of the desired and returned song name, artist name, and album in which the song is from. We will have a parameterized minimum similarity for us to choose a song, but also will not require 100% similarity to determine a match, but rather will parameterize a threshold that results in the most accurate selection process. 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.

4. 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 51% of active users have voted against the song we will remove it from the queue. This also goes for users who have not sent a heartbeat in the last 15 minutes. Their timer will run out and an event will be triggered to remove their votes from all of the songs. But their votes will still be stored for when they get back on the app.

5. 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.

C. Recommender Raspberry Pi Core

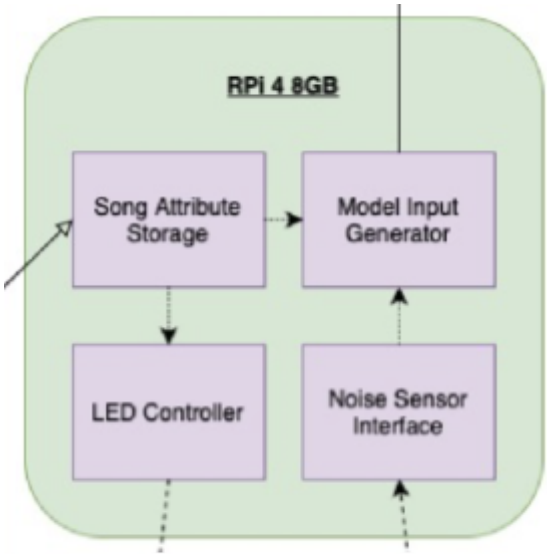


Fig 6: Recommender Pi

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. LED Controller

The lighting fixture attached to the recommender RPi will be controlled via DMX signals transmitted over a DMX cable. These signals will be generated on board the recommender RPi using a Python program which controls the different channels (independently controllable groups of LEDs) of the fixture by using the DmxPy interface to generate specific DMX outputs. The DMX channel signals, which control the behavior and colors of the lights, will be determined based on the characteristics (danceability, liveness, energy, etc.) of currently playing songs, which are derived from the Spotify Web API.

VII. Test, Verification and Validation

The software, hardware, and networking aspects of the Music Mirror DJ system will be tested to verify intended behavior and validate the quality of our submodules. The objective is 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.

A. Tests for Web App to Queue Latency

Timestamped test song queue requests will be issued from a mock web application instance to the DJ system, and will be used to measure the time elapsed between inputting a request and seeing the corresponding queue update return to the web app. As per the specification the latency will be clocked at under 1 second roundtrip time. This will allow us to benchmark the level of responsiveness of our web app to internal system pipeline. Fulfilling this latency test will ensure that users will have a seamless experience, and not be frustrated or discouraged from using the app.

B. Tests for User Web App Onboarding

In order to test the intuitiveness of our web app we will 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. To accomplish this we will time fresh users which have never been exposed to our web app and measure 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. The target amount of time for this onboarding is less than 1 minute. Additionally we will issue surveys for users to time themselves on how quickly they can learn to use the app to aggregate a larger volume of testing data. This will ensure that the barrier for entry into our DJ ecosystem is low, and that the vast majority of guests will have their song choices factored into the system, creating a much more accurate reflection of the crowd's music tastes.

C. Tests for User Network Capacity

To test network capacity a barrage of stress tests will be 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 100) will connect to the DJ, and we will verify that the system can manage these large amounts of websockets and accept requests from any of them at any time. Additionally, we will send multiple 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.

D. Tests for Song Queue Capacity and Veto System

To test our song queue we will use a Python script to simulate different loads of users performing actions that our clients would. So they will request songs as well as vote against others. Since there is no reference solution we will define certain steps that each user will take and will check the queue to see if it is what we expect for correctness. We will also make sure that our system can hold events from 150 users by doing simpler correctness tests and also doing stress tests where the 150 users request multiple songs to see if the system can handle it. Namely, we will test certain circumstantial sequences of requests, such as multiple users sending requests at very high-paced rates.

E. Tests for Song Recommendations Quality

Because song recommendations are a subjective matter in nature, we will be testing the quality of them with user feedback surveys. We will provide lists of songs that have been played, and then will display Spotify's naive recommendation results, and then our own improved two-tier system's results, and will have the user compare which of the two is preferred. We are looking for 80% preference between our model to Spotifty's, in order to justify the complexity of our design choices. These surveys will be sent out to 100 participants for statistically significant results.

F. Tests for LED Behavior Matching Music

The lighting fixture's LEDs will be visually inspected 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. If the fixture emits the incorrect lighting scheme prescribed to the song at any point during its duration it will be considered an error, and the accuracy will be determined by calculating which percentage of the songs in the predetermined test set produced an error. The target is at least an 80% success rate. This design requirement will ensure that the users will perceive a coherent and synergized DJ experience, leading to greater user engagement and satisfaction.

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 connection with the Raspberry Pi • Light controller • Communication protocol between RPis • Queuing/voting functionality
Matt	• User graphical interface • Web app

	communication with backend • Queuing/voting functionality
Luke	• Recommendation RPi implementation • Authorization Driver • Semantic matching • Speaker pipeline connection
All Members	• Loudness Sensor Integration • User satisfaction surveys • Testing

Fig. 1. Schedule example with milestones and team responsibilities

C. *Bill of Materials and Budget*

Fig. 5. 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/Mo
DMX Controlled Lights			$170	Varied
Any Speaker			$0	Varied
Wireless Audio Streamer	Mini Airplay2	WiiM	$89	$89
			$291.97	$239+10.99*Mo

D. *Risk Mitigation Plans*

The team has identified a few potential risks that could hinder the progress of our design implementation. These risks necessitate mitigation strategies to ensure continued advancement.

One potential risk is not being able to properly control the lighting fixture through a signal generator module hosted on our system. This is because the DMX protocol intended to control the lights is more of a hardware solution, and may not lend well to being controlled via software. Our mitigation strategy is familiarizing ourselves with other DMX signal generation libraries as backups, such as the Open Light Architecture framework and PyDMX, in addition to our primary DmxPy interface.

Another risk would be if we can not effectively transfer data between the RPis through ethernet. We don't think we should have any issues but if we do then there are other ways to transmit the data like Bluetooth or some sort of socket connection.

IX. 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.

X. Summary

The Music Mirror project aims to democratize the music listening experience by creating an all-inclusive smart DJ platform.

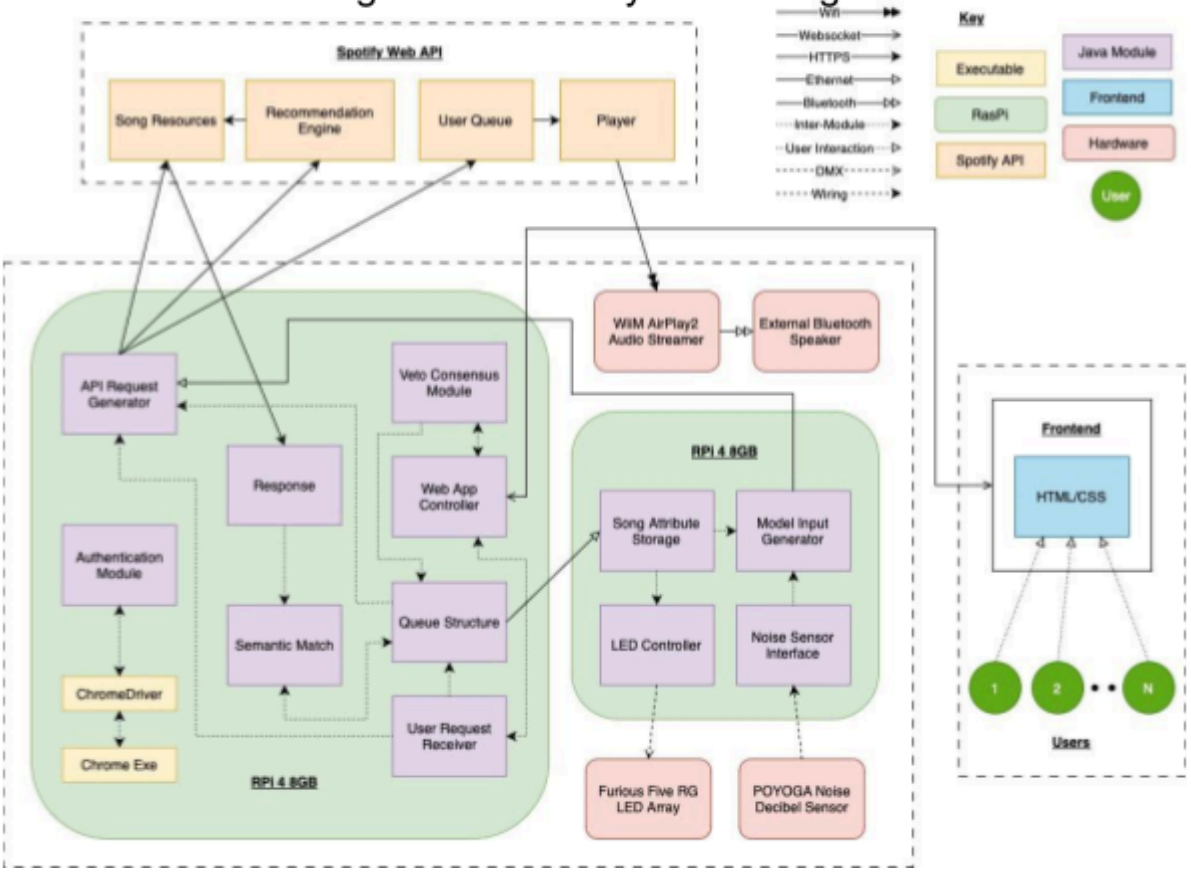
Glossary of Acronyms

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. 7. Detailed System Design



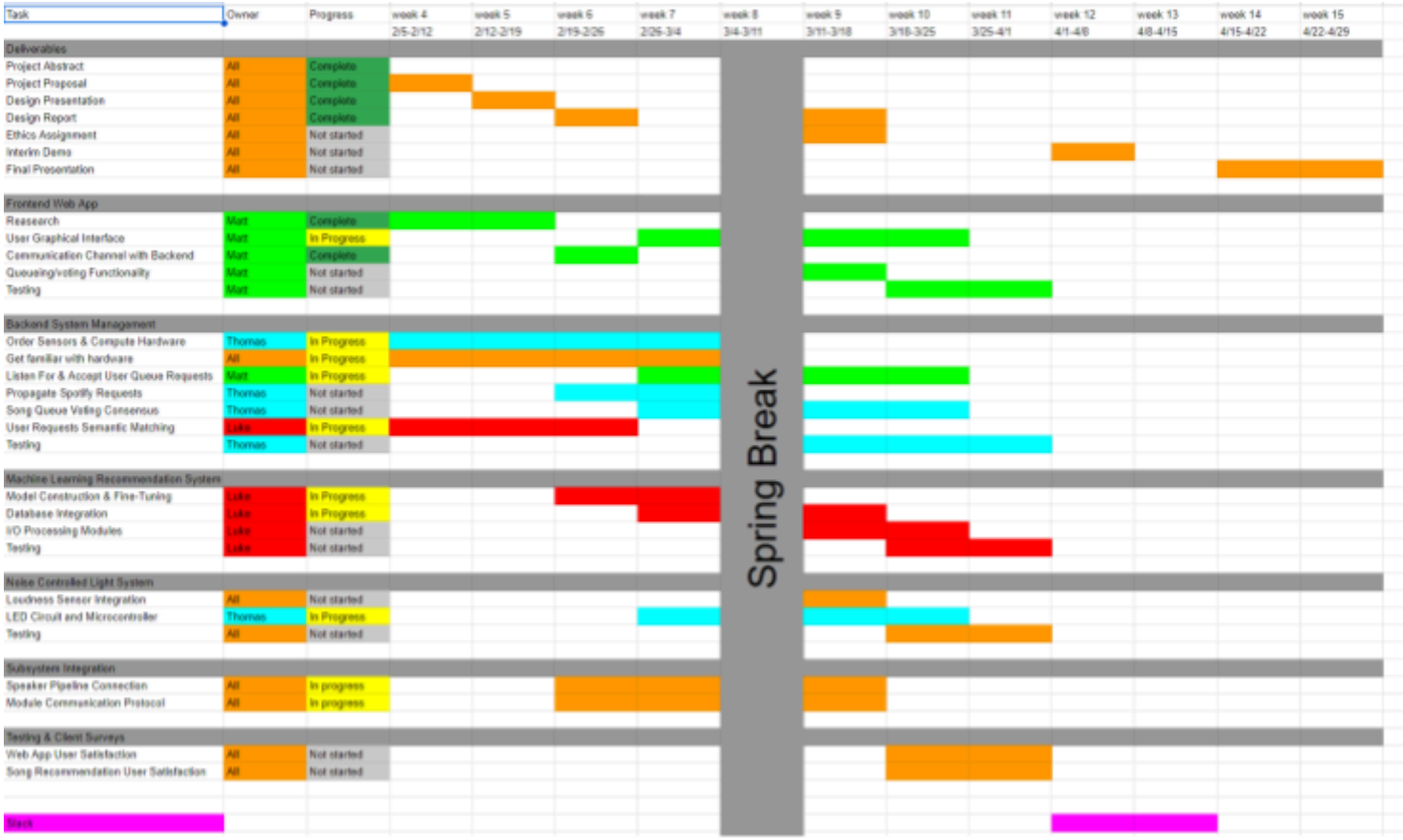


Fig. 8. Gentt Chart