A Functional Music Player

Nate Wagner

Abstract — The Function Music Player (FM Player) is a new music streaming service. It allows the user to create a new type of playlist called “Functions.” Functions are written in a Turing complete language and use a drag-and-drop interface to allow the easy creation of complex playback rules.

Introduction

For a long while, I’ve been looking for a music streaming service that would allow me to create complex rules about how playlists are played. Unable to find such a service, I set out to build my own. Before I begin explaining the project, there are a few things to note. First, this project is still a work in progress, so some large parts are unfinished. That being said, it has become my primary music streaming service, and it’s far enough through development for me to feel comfortable sharing it. Second, one of the first problems I faced in development was that to create a music player, you need music. My solution to this problem was to use a version of the open-source YouTube DL project to extract audio from YouTube videos. As you may have guessed, this solution poses a few questions about legality. While I am not a lawyer, I did research the subject extensively before starting. To the best of my knowledge, using YouTube DL for personal use does not pose any legal issues, but releasing public access to that section of my project would. For this reason, I will not be giving public access to the project website or the AWS code for downloading audio. With all that out of the way, let’s start with the tech explanation!

Part I: YouTube Integration

A Functional Music Player

One of the first problems I tackled in the project was audio. Once I decided to use YouTube as a source for audio files, I laid out what an implementation might look like. Through this process, I would need three main things: a place to store audio files, a system to automate downloading audio from YouTube, and a system to standardize the format of the audio files. I considered a few options for storage, such as a network-accessible drive on my local network or paying for a dedicated server with a platform like Dreamhost. After some research, I discovered that Amazon offers a service called S3 that is specifically for this kind of project. Using Amazon's online cost calculator, I found that even storing more than 10,000 songs would cost less than $1 per month.

Having chosen Amazon Web Services (AWS) as my storage solution, when I went to look for where to run the audio downloader, AWS was the first on my list. Through my research, I found AWS Lambda Functions, a way of paying for computing time without worrying about physical servers. As it turned out, this fits my downloading and standardizing audio requirements: two birds with one stone!

Now that I knew the platform I'd be using, it was time to start with the implementation. Before this project, I had never used AWS, so navigating their large, verbose platform was challenging as I began working with their tools. For about two weeks, all my work was reading tutorials and taking deep dives into Lambda, S3, IAM, and ECR documentation. Afterward, I felt more comfortable with AWS and was ready to begin a prototype.

I created two functions, one for downloading YouTube videos and one for converting files to the .ogg OPUS format. I had previously learned about FFmpeg, a multipurpose tool for operating on audio and video files, so that was my go-to for conversion. Without much trouble, I got a compressed version of FFmpeg to successfully convert a given input file to the OPUS

format and save the output to the correct location. The YouTube downloader, on the other hand, was much more difficult. The biggest problem I ran into had to do with AWS Lambda runtimes. My entire project thus far had been written in Typescript (compiled to Javascript), and I wanted to keep things more straightforward by writing all of my Lambda functions in Typescript. This choice began to pose problems when I found that the YouTube DL software was written in Python, and even the compiled binaries required a Python runtime. By default, Amazon provides a Node JS or a Python runtime for its Lambda functions, but not both simultaneously. The only way to use both would be to create a custom runtime, a much more involved process using Docker, a tool I had never used before.

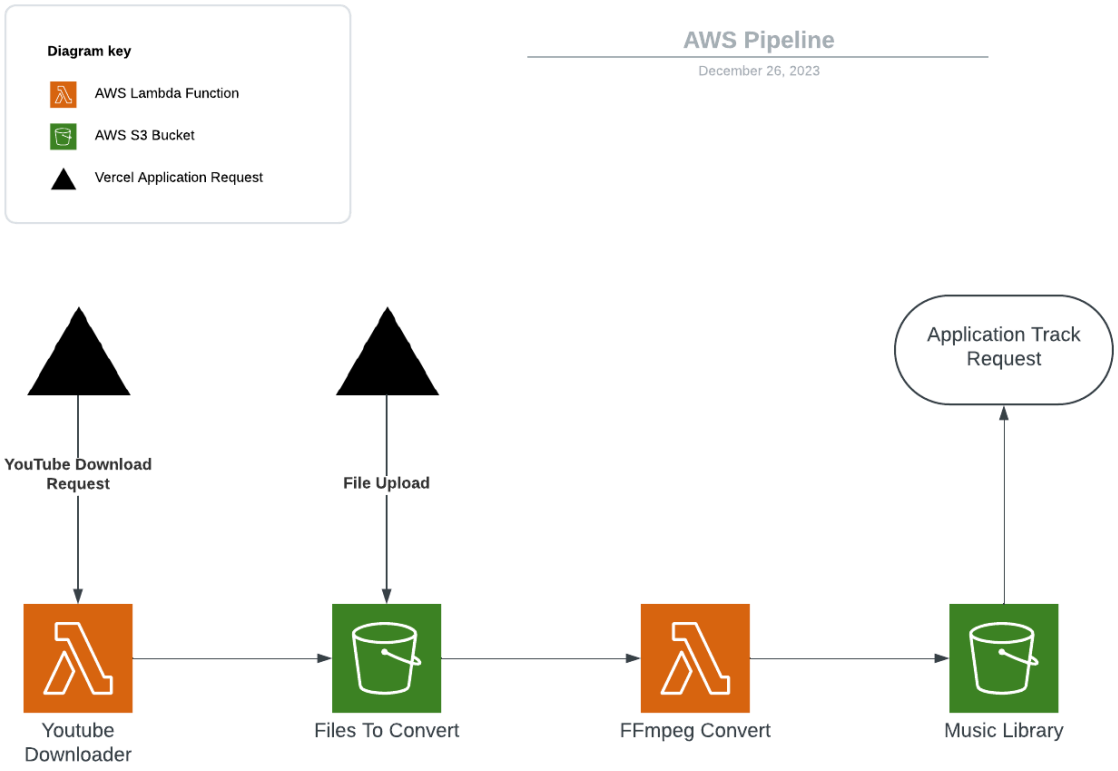

Figure 1. A diagram of the pipeline of AWS services currently used by my application. It was created using lucidchart.com.

I ended up going with the custom runtime route. In hindsight, keeping all of my code in Typescript likely wasn't worth the work required, but along the way, I learned more about AWS and Docker than I ever would have otherwise. When I finished, I had a reasonably robust AWS pipeline (see Figure 1) that would allow for the easy addition and storage of new tracks.

Part II: Data structures

When I began working on this project, I knew I wanted to be able to create programmable playlists, but not much else. One of the first problems I ran into was where to get audio of the tracks in my library. I looked into solutions, such as utilizing Apple Music's vast library through their MusicKit API. Unfortunately, all such services I could find required you to be 18 years old or had costs meant for large businesses. As mentioned above, I eventually settled on the YouTube DL for raw audio but quickly arrived at a second problem. While YouTube could provide the audio, it wasn't cataloged well for use in a music player. I realized I would need some way to store track metadata, artists, albums, and playlists. I quickly wrote a JSON file to store my entire library and called it good enough.

As you might imagine, this did not turn out to be "good enough," as I began developing a user interface, I quickly discovered many of its shortcomings. After running into many problems with data validation, updating, and synchronization, I decided to look for other ways of storing data. On other projects, I've had success working with SQL databases, and Vercel, the hosting service I use, was offering a 256 MB database as part of their free tier. This offer made an SQL database (specifically Postgres) the obvious choice for metadata storage.

Once I had a database, I wanted to find an object-relational mapping (ORM) tool to allow easier access to my data. After some research, I found a tool called Prisma to generate a

specialized typescript ORM and update the database based on a provided schema. I converted my JSON file into a Prisma schema, wrote some wrapper code to open a secure access point, and got the database running.

This system worked great for the next few months, but as I outlined in more detail what I wanted the project to look like, I started to find problems with my schema. At the time, my database structure allowed me to store tracks, albums, artists, and playlists, but with the growing scope of the project, I realized I needed to store more. I began planning out my new schema, first by researching. One of the more helpful resources was from the people at musicbrainz.org, who have published their database schema. I also learned about ER diagrams and found them a beneficial planning tool (see Figure 2).

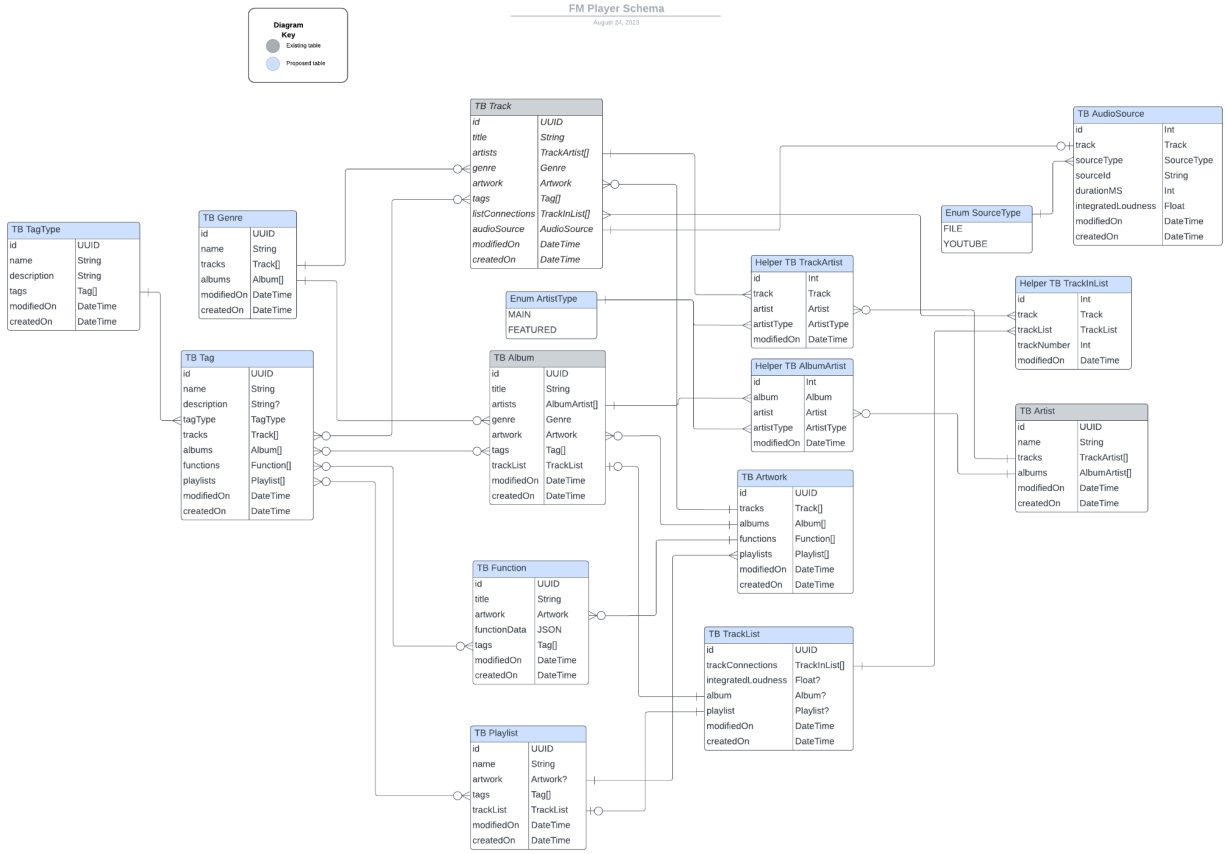

Figure 2. Planning document for the current version of the database schema. It was created using lucidchart.com.

After about a week and a half of planning and development, I was happy with the structure I'd created and ready to integrate it into my project. Prisma was instrumental in this effort, as it automatically handled database connections and generated typescript types. The only problem was that Prisma is explicitly server-side. While Prisma handled connections and operations on the database, for security, those connections could not be made from the web client as it would mean sending out the database password to anyone accessing the app. Until then, I had solved this issue by creating individual access points with very narrow-scoped access to the database. While the number of tables was small, this worked fine, but with the new larger schema, I knew I needed a more robust and scalable system.

My eventual solution was to build a wrapper around the Prisma database client on the server and a Music Library component on the client that would allow for secure dynamic access to the entire database together. While planning, I knew it needed to be able to do four things: Create, Read, Update, and Delete (CRUD). Using CRUD as a starting point, I began to build the server-side component that would allow me to call various methods of the Prisma client with an arbitrary table. The client-side component follows a similar structure by providing helper methods for various CRUD operations and sending requests to the server when they are called.

You can find the current version of the server-side code at /api/db.ts. In that file, you will find a handler function at the top that receives and processes incoming HTTP requests, followed by methods for individually executing each CRUD method. The handler method also includes code using a service called Ably Realtime. This service sends an update notification to all active

clients to ensure they receive any new or changed data. Assuring the security of requests is handled in /middleware.ts using short-term JWTs for authorization and long-term refresh tokens for persistent logins.

The client-side code can be found at /src/music/library/crud-module.ts. This file constructs an object with a set of CRUD methods for each table. When a CRUD method is called first, a browser cache is checked for duplicate requests; if it fails, a request is sent using my VercelAPI helper object, which ensures the proper authorization is added. This whole system makes manipulating data within the application readable and easy to use.

```
1 const album = await MusicLibrary.db.album.get({id: albumId});
```

Figure 3. Example usage of the *MusicLibrary* object's database module. Specifically, this requests the album data for an album with the unique ID stored in the variable: *albumId*.

It has taken a few iterations to get this right, but the system I've created has worked very well and has been relatively easy to update.

Part III: Functional Playlists

As mentioned before, I started this project because I wanted the ability to write complex rules describing how my music is played. Most of my time on this project has been spent creating a base framework, so I wanted to ensure I did it well when creating the programmable playlists.

In the past, I have been interested in how computers interpret human-readable text as instructions, and I've had some experience writing lexers, parsers, and interpreters from scratch.

This knowledge gave me a starting point for organizing my language, and I began planning by writing out a grammar loosely based on the extended Backus-Naur form.

After deciding on the structure of my language, I need to determine how a user would interact with it. I created a few prototypes using text-based syntaxes but ultimately decided that a visual interface would fit a music player much better. I decided to build a drag-and-drop editor tool and, taking inspiration from MIT's Scratch editor, used color and shape coding to differentiate between types.

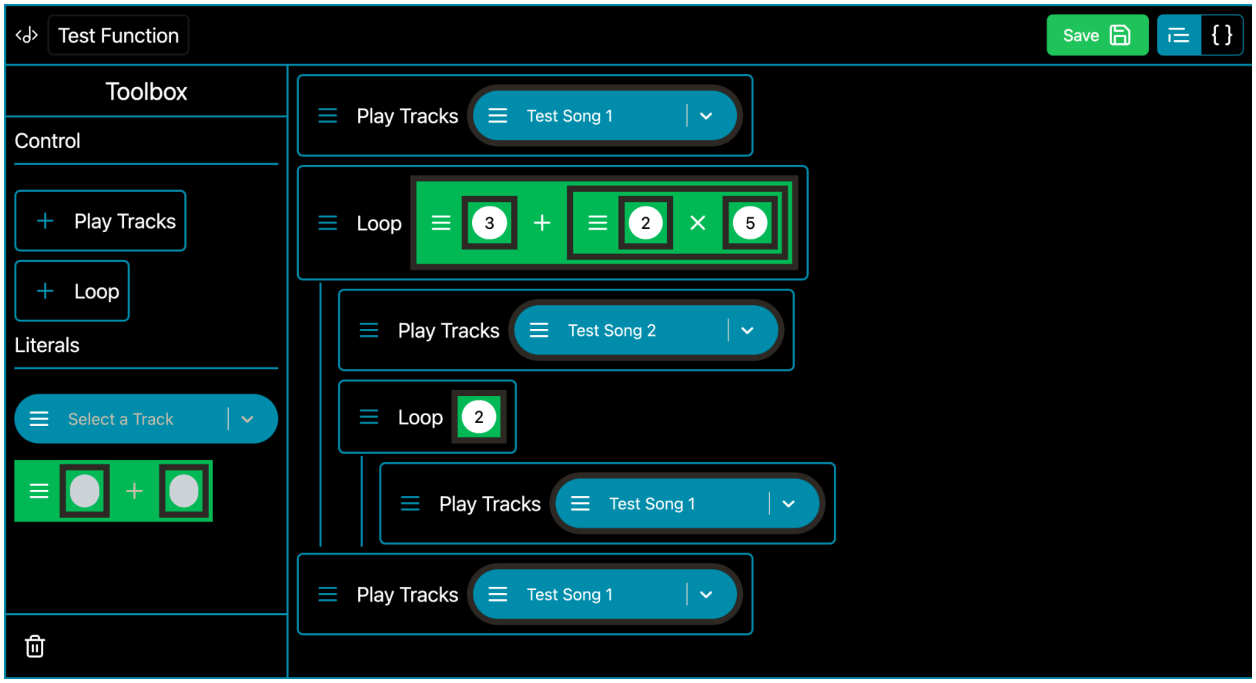

Figure 4. The current version of the visual function editor. It includes the usage of loops, nested loops, binary expressions, and playing tracks.

The function editor is the portion of the project I'm currently working on, so many parts are a work in progress. Nevertheless, I would like to highlight some of my considerations during its development.

A Functional Music Player

First, from the outset of building the function editor, I had been considering how to structure my system of compiling/interpreting the function code. I had already written some scaffolding for playing a flat list of tracks, but as I began planning the function editor, I realized this wouldn't work. In my planning document, I included a method of using outside data sources such as the current date, time, and weather. If I continued with my original plan of "compiling" a function into a flat list of tracks when clicking play, playback couldn't be nearly as dynamic as I was looking for. My solution to this was to write a playable function class to handle the playback of a function. The playable function is based on a system of states storing information about variable scopes, references to the current and next actions, and the current track ID. This state information is passed back and forth between the playable function and the audio player, allowing the execution of the function to be paused right until the next track is needed.

```
// Returns the next state that includes a track id and whether
// the program had looped back to the top.
public getNextTrack(
  currentState?: Functions.RuntimeState
): [Functions.RuntimeState, boolean] {
  if (!currentState) {
    currentState = this.createInitialState();
  }
  let trackId: string | null = null;
  let nextState = currentState;
  // If the currentState was the last state, capture it when returning the next state.
  let didEnd = currentState.isEnd;
  while (trackId === null) {
    nextState = this.evaluateState(nextState);
    trackId = nextState.currentTrackId;
    // trackId === null because we want to catch the end
    // when switching to the next track.
    if (nextState.isEnd && trackId === null) {
      didEnd = true;
    }
  }
  return [nextState, didEnd];
}
```

Figure 5. Method on the playable function class that handles evaluating the next track.

Second, the function editor is still unfinished and is missing key features, such as variables, conditionals, and track lists. That being said, much of the difficult work is done. While building the editor interface, I kept the end goal of supporting my complete language grammar in mind. I ensured components such as the action display could easily have new action types added. I also intentionally left the data attribute of function tree nodes unknown to allow new action types to store any needed data.

Third, because the editor is not text-based, there is no need for a lexer or parser. The data used by the editor to display the function also behaves as the abstract syntax tree (AST). Unfortunately, this means errors commonly caught during syntax or semantic analysis, such as a loop missing a value for its loop count, would not be caught. To fix this problem, I wrote a validation library to catch these errors before the editor is saved. The validation has two phases. The first phase is a precheck; it assumes the passed-in function could be anything and only succeeds if the function has the correct object structure. This phase is akin to syntax analysis. The second phase then makes sure each statement is configured correctly. For example, it checks to ensure that a binary arithmetic expression contains a left term, a right term, and an operator. This phase is akin to semantic analysis.

Creating playable functions and the function editor was by far the most exciting part of the project. It provided many novel challenges to tackle, and being the goal of the past nine months of work, it has felt very satisfying to see it work well.

Part IV: Workflow and Best Practices

While this project started to satisfy my desire for a new music streaming service, I stayed invested in the project because of the new knowledge I gained. Some of this learning has been in

new programming techniques and patterns, but much more has been in project management, project structure, and workflows. Despite this being a solo project, I know that in the future, most projects I work on will be with a team. For learning purposes, many of my project management and structure decisions were made as though I were working with others. Much of my knowledge about this subject has come from reading articles, examining similar projects, and talking to industry professionals about their strategies and workflows. (Side note: coincidentally, MIT's *The Missing Semester of Your CS Education* was one of the best sources I could find on the basics of Git and CLI usage.) I want to share some techniques I've learned about and implemented throughout this project.

One of the earliest tools I discovered was a Kanban board for task management. In my past projects, I have often struggled to organize tasks or focus on a single section; this has sometimes led to the project's abandonment. At the beginning of this project, however, I had the opportunity to take a deep dive into the project management of Sibcy Cline Realtor's website rewrite initiative. What stood out most to me was a tool they used called Asana, a task management tool built for teams. I decided to try it out for my project and set up an account. It provides an excellent way of managing priorities, tracking bugs, breaking down tasks, and staying focused. As of 12/30/2023, I have created and completed over 50 tasks throughout my project and have many more in progress.

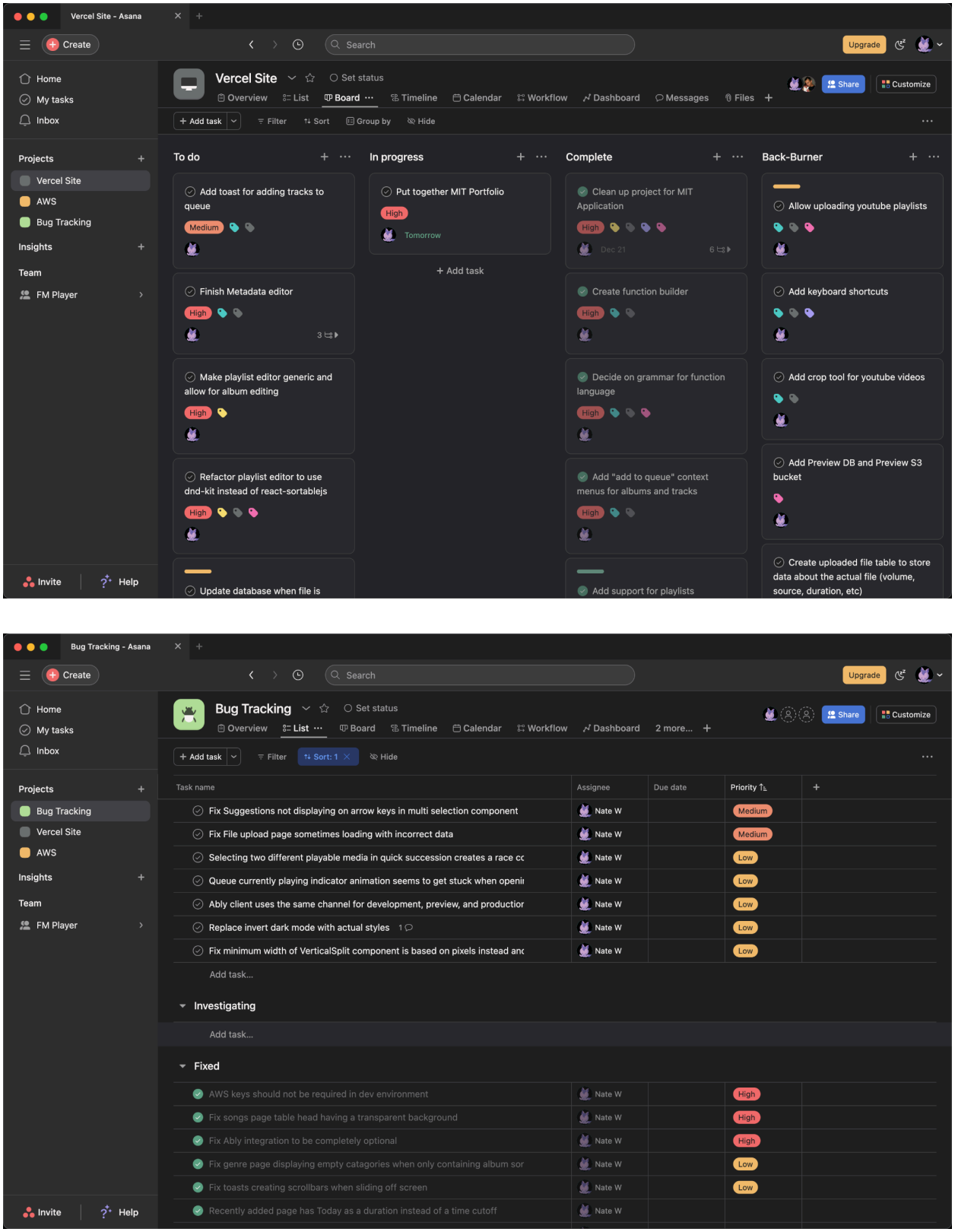

Figure 6. My Asana Kanban boards for bug tracking and front-end development planning.

A Functional Music Player

This project was my first time learning about continuous integration and deployment (CI/CD). When I began my project, my hosting platform, Vercel, offered an integration with GitHub. With the integration enabled, every commit made to the GitHub repo would automatically have a test deployment made for it. This tool has made worrying about whether my changes have made it into production a non-issue. Sometime after starting the project, I learned about git hooks and commit linters. After talking to a professional web developer about the importance of clear commit messages, I decided to add a commit linter to my project. Since adding a commit linter, the quality of my commit messages has increased dramatically. For example, compare this message from April, "Major updates to UI," to that from December, "feat: add track title to context menu."

Starting this project, I had a passing knowledge of the purpose of tests, but I had never written a test myself. Initially, I didn't see a need for tests, but as the project became more complex, I realized I could save myself many headaches by writing some tests. I set up the testing framework Jest and began writing tests for some of the more complicated pieces of code. Around this time, I also discovered GitHub Copilot. While I found it unhelpful for most tasks, I got decent results for writing tests. All of the tests for my application are found at /src/tests/. Any tests written with the assistance of an AI are labeled as such in comments.

The learning I've discussed in this section is perhaps the most important part of this project. I have thoroughly enjoyed building and using this project, but I don't see myself working on it for the rest of my life. On the other hand, the knowledge and skills I've gained during this journey have already improved my work in other projects and areas of study and will stay with me long into the future. Even this paper itself has been a valuable learning experience. I worried about college when I was younger because I didn't think I could write a paper like this.

A Functional Music Player

Having the chance to push my skills as a writer and share about a passion project has been a fun (and sometimes stressful) experience, so regardless of the outcome, I am glad to have had the opportunity to write it.