Final Report: Streaming Music System

Group
Ellin Hu (eh479) and Chris Sciavolino (cds253)

System Goals

Our system allows a user to stream music from our web platform using the Spotify API. Based on the user's music preferences, we suggest and tailor playback based on what the user typically listens to and what he/she skips. We base our system's recommendations on interactions including skipping songs and listening to a song to completion. For example, if the user listens until the end of a song (i.e. the player terminated), it makes sense to suggest other songs like it. On the other hand, if the user listens 10 seconds and skips to the next song, it's likely that the system should not suggest songs like that one.

Data and Interface

Static Data

We store information needed to generate a user's login, the user's settings within the app, and all interactions necessary to recreate the model specific for the user. These are stored from session to session and will be kept up to date depending on the user's interactions with the website. Additionally, we keep track of the user's Spotify refresh token, which allows the system to dynamically generate a user's access token as necessary. This way, the user does not have to log into Spotify very often, spending more time interacting with our system.

Real-time Data

The user's information and preference is dynamically rendered based on the user's account information. Additionally, all the information associated with a given user will need to be requested and rendered upon logging in. Song information and streaming data can be directly requested from Spotify through the browser using AJAX. Other dynamic data include the machine learning models associated with each user. The models are trained and updated as the user interacts with the system. User interaction data is stored in the data store, and queried by the backend when servers fail. We used Amazon DynamoDB to store data about interactions.

Interface

The client accesses our system through an HTTP server. First, they need to log into the system or create an account (also logging into a valid Spotify account as well). Next, they are redirected to a main landing page that displays the suggested songs to the user. On this page, they can play individual songs, look through information about the suggested songs, or refresh the view with all new songs generated from our backend ML model. Upon playing a song on this page, an AJAX

request is made to Spotify and the song is streamed in milliseconds (see evaluation section below). Using a basic HTML5 player, they are able to pause, scrub through songs, or control volume from the browser.

Client-side Server Interface API

```
// javascript (Node.js)
/**
 * /record-interaction: Records a user interaction in the Interaction table
 * and notifies backend servers.
 * parameters: { username, song_id, interaction_type }
 * returns: 200 if successful
 */
app.get("/record-interaction", (req, res) => {
  // ...
});


/**
 * /refresh-feed: Request a new list of song IDs to present the user, using
 * a backend service with access to a user's model.
 * parameters: { username, access_token }
 *  returns: [ suggestedSongIds ]
 */
app.get("/refresh-feed", (req, res) => {
  // ...
});
```

The only expected API calls from the client browser to the follower (HTTP) servers are storing user interactions like skipping songs and listening to songs fully and generating a new suite of suggested songs. Spotify API requests are done on the local browser to optimize stream times using AJAX. In order to record interactions with the user, the HTTP server needs to know the user's username as well as information about the interaction (what type it was and with which song). For the refresh feed endpoint, the HTTP server makes an API call to Spotify to generate a large library of songs. To do this, the follower server needs the user's username as well as their access token with the proper scopes approved.

Backend Server Interface API

```
// javascript (Node.js)
/**
 * /subscribe: Fetches data from the datastore and saves a trained
 * model associated with the user in a local reference.
 * Should be called when a user logs into the client-side service
 * params: { username }
 * returns: 200 if successful
```

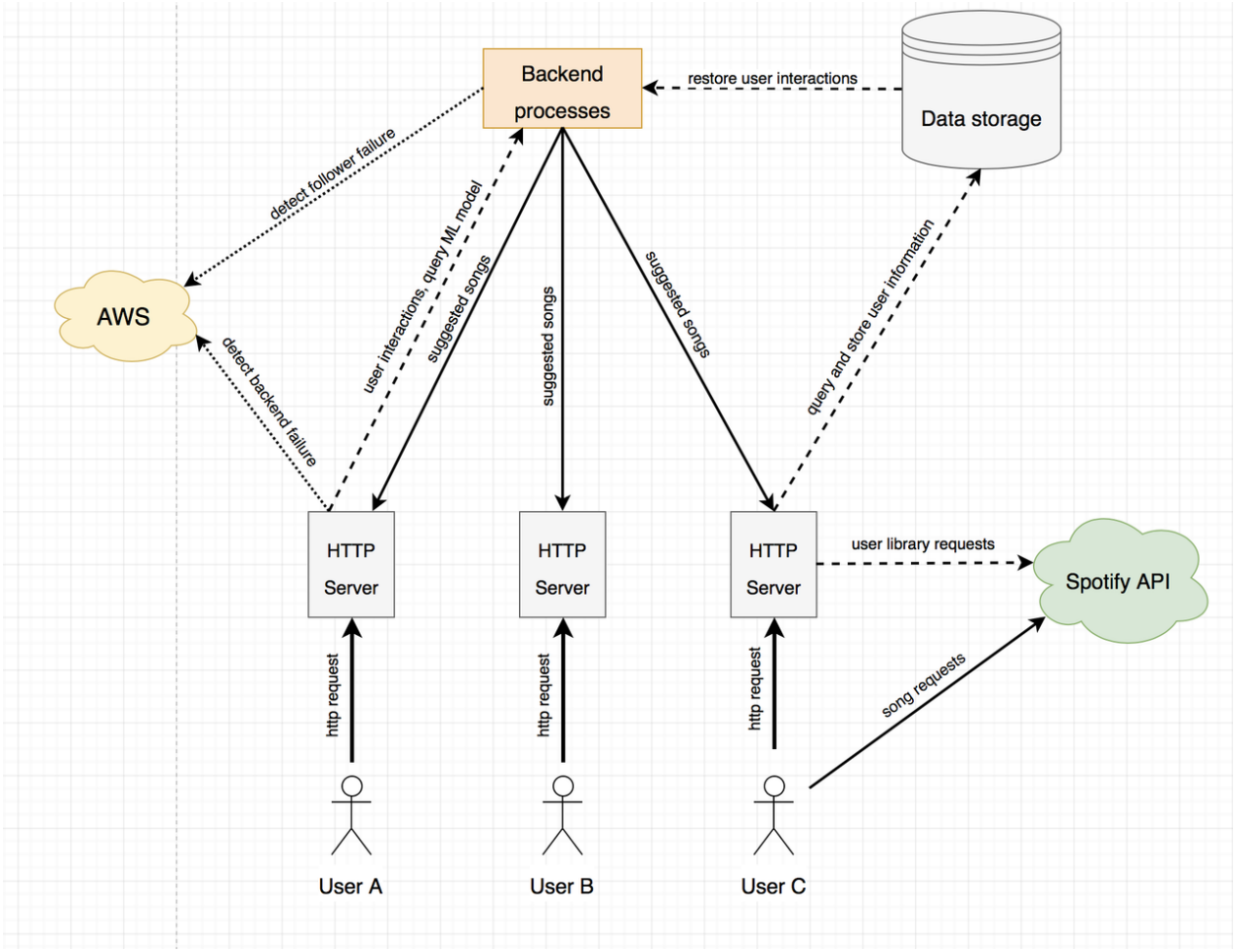
```
 */
app.get("/subscribe", (request, response) => {
  // ...
});

/**
 * /receive-info: Takes in a user interaction and trains the user model to
account
 * for this interaction.
 * params: { username, interactionType, songId }
 * returns: 200 if successful
 */
app.get("/receive-info", (request, response) => {
  // ...
});


/**
 * /suggest: Takes in a list of possible song suggestions and uses the
user's
 * individual ML model to determine which to suggest.
 * params: { username, songs }
 * returns: [ suggestedSongIds ]
 */
app.get("/suggest", (request, response) => {
  // ...
});
```

Implementation and Architecture

Our system takes place on a web platform (website) and the edge device is located on a browser on a computer or mobile device. We tried to make the webpage as responsive as possible while offline, but some aspects inherently require a network connection (like downloading and streaming unexpected songs). The way that Spotify distributes their music content is through streaming, which prevents downloading individual tracks and preloading them. Although we can save individual URLs, it still requires a network connection to stream any individual song.



From an architectural standpoint, we split our system into three main categories: the follower (HTTP) frontend servers, the data processing and ML backend servers, and the AWS DynamoDB storage layer. Each has very important and specific capabilities that emphasize their significance within the larger system.

Follower (HTTP) Server

This is an image for a simple HTTP server. The primary goals with these servers are high availability and repeatability (fault tolerance). Each server is identical to one another and are able to:

- Respond to a client HTTP request and serve the website's content
- Recording and storing user interactions with the data storage layer
- Preprocessing intermediary requests between the client's browser and the backend servers
- Handle safe, encapsulated authentication and authorization with the service

We implemented these servers using Node.js, a framework that emphasizes asynchrony and quick responses. Given the goal of the system is to be incredibly scalable with high availability, Node.js complements this nicely. In most of our processes, we make an asynchronous request while

responding immediately with a quick response. This allows clients to receive quick, snappy responses while complex operations are loaded in as the responses come in.

Backend Servers (Data Processing and Machine Learning)

The backend servers aggregate updated information sent from the HTTP servers about user interactions with the website and update the user's individual machine learning model. The important tasks these backend servers handle include:

- Updating a user's individual machine learning model based on new user interactions
- Serving predictions for song recommendations that a given user may enjoy listening to
- Training user machine learning models from scratch upon first login
- Maintaining a list of machine learning models for active users to avoid unnecessary training

The primary machine learning model takes in a user's song preferences and should give possible song recommendations given these preferences. The training for this model considers interactions with the user's song and the song id. A positive feedback is instantiated if the user finishes a song entirely and the machine learning model categorizes the song as a potential recommendation. A negative feedback will initiate if the user doesn't finish a song and the model categorizes it as a poor recommendation. Then the model incorporates these interactions by training the existing model corresponding to the user. A list of subscribers and their corresponding machine learning models are kept locally.

As of right now, we use a RandomForstClassifier with the mljs library. A random forest classifier incorporates bagging, which reduces the variance by averaging many individual classifiers. Given the models start with very little data, reducing variance helps provide users with generally reliable recommendations while they build up an interactions list.

For a song's feature vector, we use Spotify's inherit feature vector. Using the Spotify API, we are able to request a song's sound analysis, which calculates characteristics about the song like loudness, repeatability, popularity, and genre. Using this vector for each song and a binary classification label (recommend (+1) and don't recommend (-1)), we produce a model that predicts whether a user will likely listen to a given song.

Database

We used DynamoDB for our database, and had two tables.

```
User - table containing all users of the Spoofify service
  Username - PrimaryKey String: unique identifier for the user
  HashPassword - String: hashed password for the user
  DisplayName - String: name to display to the user
  AccessToken - String: most recently used Spotify token the user has used
```

```
  RefreshToken - String: token used to refresh access to the user's
AccessToken


Interaction - table containing all interactions the user has had with the
system
  Username - PrimaryKey String: unique identifier for a list of interactions
  Interactions - Object List: list of all interaction dicts for the user

Note: Interaction dict mentioned in Interactions:
{
  SongId: String: unique identifier for a song, referenced through Spotify
  Label: String: type of interaction associated with this SongId (skipped |
finished)
}
```

Fault Tolerance

There are three possible ways for the network to cause outages throughout our system:

1. Follower servers get a timeout when requesting for information from backend servers
2. Browser gets a timeout when requesting information from follower servers
3. Backend servers get a timeout when sending information to follower servers

In all of these cases, the server receiving the timeout response notifies AWS of the failure. AWS first tries to redirect the requesting server to another available server. If AWS cannot find another available server that can satisfy the request, then it will create a new server with the same partition needed to satisfy the request.

AWS detects when it is notified that there is a timeout in communication between servers (for example, follower server's request to backend times out). In this case, we will have AWS start a new server to replace the server that has timed out, and it assumes that the old server is unresponsive. The new server takes data from the data store and trains it into a new machine learning model.

In our implementation, we tried to last as long as possible without a network connection to a follower server while still being a usable web service. In order to accomplish this, a lot of the music playback and requests are done with Asynchronous Javascript And XML (AJAX) requests. This allows communication with servers and databases without needing to reload a page or contact the rendering follower server. By using AJAX requests, we allow the user to listen to any songs presented on the main landing view while the follower server is crashing or terminated.

Finally, should our data storage become corrupted or crash, we could solve this by shadowing to backup data storage servers. In the event of a failure, AWS could designate the current "master" data storage as invalid and promote one of the backup servers to become the main data store that

handles answering requests. Additionally, should we lose out on a couple of user interactions in the downtime, they would likely not be missed very much. Considering the sheer quantity of user interactions possible for each user, missing 1 - 10 per user would have very little impact on the overall predictions of the ML model.

Spotify API

The Spotify API is the source of all music streaming data as well as metadata about the user (like playlists etc.). Whenever we request, stream, or recommend music, it is requested through Spotify using either the user's or a personal Spotify Premium account. This includes information needed for the machine learning model, such as song features, streaming CDNs, and songs from the user's library.

Possible Improvements

If we want to add extra functionality that needs to be accessed real-time rather than cached, we could add microservice servers that can respond to requests for dynamic content (like real-time statistics on user data etc.). This would allow us to abstract the functionality away from the HTTP servers and leader servers and have a specialized server for a specific desired content. This server would act as a simply API request and return content to the user via a simple HTTP request.

Additionally, we would have included some more data for feature training in our model, such as search queries from a user or recommended songs moved into user playlists. As of right now, it's very difficult to quantify how well our machine learning model is doing. Whether a song should or should not be recommended is very subjective, so getting a definitive accuracy for the model would require a lot of data and would likely still be very subjective.

Finally, a major point brought up in peer reviews and reports are asking what the differentiating factor is between this service and what Spotify is doing right now. One of the neat features that Spotify leverages for high-quality machine learning models is social networking; however, they only do this through a music and artist social graph. The next major improvement that would set a service like this apart from Spotify would be full social media integration. This would significantly increase the number of interactions a user could have with our system and it would allow us to find new songs to present to the user using this kind of social graph. Just as services like Facebook and Instagram can feed immense amounts of information from user interactions with their social media site, having such immense amounts of data about a user's music preferences would likely bring on a new era of machine learning for music.

Evaluation and Testing

To test our system and see how it runs, we used two Amazon Web Service's LightSail instances under a free tier and tested it with them. We primarily looked at latency in order to address the availability and responsiveness of the service.

The primary features we are looking for in this system are short latency across all servers. Short latency is critical in terms of user experience because no users enjoy waiting. We also examined

how the latency is affected under stress. If the latency remains relatively constant under stress, then the system should be relatively scalable; however we were relatively unsuccessful when testing this. We found it very difficult to truly assess the latency of the system without having access to a larger user base with various computers all working at the same time.

In the data below, we record the average testing times of functions in the backend. Most of the functions had relatively low performance times that were very close to the base line (getting the subscriptions). This is because of most them send a response first, and then perform the rest of the actions asynchronously. However, the suggest function must call the machine learning model on the given data and return this information. Thus, it has slower performance than the other functions due to its synchronous nature. Nevertheless, the net speed of the calculations was quite fast and shows that our application scales well.

Action	Trial 1	Trial 2	Trial 3	Average Time (ms)
Get Subscriptions	82	82	85	83
Subscribe	86	48	86	73.3
Receive Info	85	84	80	83
Suggest	180	182	191	184.3

The data for the speeds of our front end pages and functions are also below. As we can see, loading pages with large images takes a long time (ex. index.html, the home page with large banner hero image). On the other hand, recording interactions does not take very long at all. This shows that the front end can communicate with the server pretty quickly.

	Trial 1	Trial 2	Trial 3	Average Time (ms)
Load main page (main.ejs)	298	235	246	259.667
Recording skip interaction	144	113	116	124.333
Refresh feed endpoint	283	337	384	334.667
Record finished interaction	141	125	110	125.333
Loading individual song from Spotify	64	32	47	47.6667
Play song	32	47	31	36.6667
Load login page	167	119	120	135.333
Load index.html (with large banner image)	495	542	567	534.667

Overview

Overall, our system scales relatively well, and is generally fault tolerant because of our extensive use of AJAX on the client-side. We are able to send requests querying information from Spotify for data from the browser directly. Using Node.js allows us to respond immediately to requests and

handle the computations asynchronously in the background. We were satisfied with the main interface provided to the user on the main landing page after logging in.

A couple places that we hoped to push further but were unable to were social media integration, extensive exception handling throughout the service, and building out a full Spotify service. For example, we only have a single suggested songs playlist rather than displaying all of the playlists in the user's library. We also could have built out a full search functionality, as a normal Spotify app would do. Our machine learning model suffers from a lack of interaction data for new users. This could be mitigated by providing a new user form that provides additional information about the user to skew the model to an age group, genre, or other general category.

Finally, a few implementation details we wish we could have built out further that would have increased scalability would be communication with AWS for fault tolerance and further AJAX support. Since a client's browser is incredibly stable and rarely crashes, having the browser execute various requests would increase the reliability of the system. More AJAX requests would also increase offline availability of the service for users.