Spotify Queue Builder

Technical Specification/Design Documentation

Jaime Eli Mizrachi

June 6, 2025

1 Motivation

As someone who is constantly listening to music, I was initially excited by Spotify's AI DJ feature. In theory, it promised personalization and smart sequencing of songs, but in practice, I found it to be too single-note and impersonal. It seemed to push the same kinds of tracks repeatedly, seemed to lack a contextual understanding of my taste, and forced industry-pushed content rather than using my actual listening habits.

I created the Spotify Queue Builder to address these issues. This tool allows me to work with my full listening history, apply filters and weights based on parameters that matter to me (mood, flow, listening frequency, listening history), and generate long, coherent playlists that reflect my actual preferences in the moment. This allows me to be versatile; I can generate a 10-track vibe for a walk or a 60-track flow for a long road trip, and can shape the outcome directly and precisely. This lets me get granular with the details of what my queue should look like.

This system is also location-agnostic. Many of Spotify's most advanced features are only available in select countries. By building this from scratch, I make sure that there is consistent access and control, since many of these AI features are only available in select regions.

2 Pipeline Steps

2.1 Build Song Feature Database

The first step takes in the user's full listening history, and aggregates it into a structured dataset. To do this, users need to download their Spotify history and upload those files. The process begins by scanning all files matching the pattern `Streaming_History_Audio*.json`. Each file is parsed to extract the track title, artist, album, and timestamp fields. Songs are identified using a composite ID based on the title and artist (since same songs might exist in multiple albums). Each matching song object holds a list of all timestamps when it was played.

Once raw listening logs are assembled, the next step is enrichment. Every song is passed through an OpenAI GPT-4-search prompt. The model is queried to estimate six core features for the song: Key, BPM, Mood (0–1.0 scale), Popularity (0–1.0), Upbeatness (0–1.0), and Genre (comma-separated list). It also infers the release year.

These estimates are parsed line-by-line and mapped to internal formats, such as translating musical keys into integers using a 12-tone scale (e.g., C Major and A minor correspond to 0, G Major and E minor correspond to 7). Songs with fewer than 10 recorded plays or flagged for noise (e.g., live albums) are excluded. This filtered dataset is saved as `user_history.csv`, and will be the main source for future playlist generation.

Note: Ideally, the original project would've accessed Spotify's API directly for this information, rather than having to use GPT to guess this. However, support for this API was recently deprecated, so this was the best viable alternative.

2.2 Load Data

Once the dataset is finished, it's ingested into memory via the `load_data` function, which loads and enriches the dataset for the recommendation feature downstream.

The function first tries to load the dataset from the given file path. If this fails, it falls back to a predefined directory to handle the different environments.

Once loaded, the dataset is enriched again for behavioral insights:

- Timestamp strings are parsed into lists and converted to UTC-aware datetime objects.
- The most recent interaction is recorded as `last_played`.
- The total number of interactions is stored as `play_count`.

A `recency_score` is then computed using the formula

$$\text{recency\_score} = \frac{1}{1 + \Delta t},$$

where Δt represents the number of days since the song was last played. This score makes sure that more recently played songs receive higher weights, while older tracks gradually decay in influence.

The enriched dataset allows the model to determine track relevance based on both metadata and user listening behavior.

2.3 Interpret Prompt

Users describe what output they would want in a free-form natural language prompt. The `interpret_prompt` function turns this into a form of structured data, using the OpenAI GPT-4 model to extract semantic parameters.

Each playlist attribute (genre preferences, mood, energy level, popularity, release year range, recency, flow type, listening time frame, desired playlist length, etc) is evaluated on its own. This modular approach lets us assign specific confidence scores to each parameter, avoiding incorrect cross-inference between attributes that might be unrelated.

To minimize the response time, each parameter is processed concurrently using Python's `ThreadPoolExecutor`. This reduces overall latency since every parameter's prompt is running at the same time. Separating the parameters ensures that they are decided independently, which leads to less biasing between parameters.

Each evaluation task gives GPT-4 a detailed system prompt that defines the parameter with context and lists valid options (e.g., the phrase "with friends" implying a preference for popular music). The model returns a structured JSON response including: a `reasoning` field with a brief explanation, a boolean `suggested` flag, a `value` if applicable, and a `confidence` score between 0.0 and 1.0.

These confidence scores play two roles. They show how uncertain the model is, and they act as *weights* in the playlist scoring process. Using confidence as a weight adds nuance to how inferred preferences influence the final output, not just relying on binary yes-no decisions.

Genre values are specially handled, given the variety of subgenres, typos, and implicit associations. Inputs are passed through a normalization and expansion layer (e.g., "synth-pop" expands to include "pop" and "electronic"), which improves recall and filtering later on.

All inferred parameters and their weights are stored in a dictionary that's fed into the UI. At this point, the parameters are displayed back to the user, where they can change the associated values and weights. Users can also make choices about other features, such as prioritizing their favorite songs, the time ranges in which they listened to songs, the length of the queue, and their intended flow.

Once they're satisfied with the way the parameters are set up, the process can continue, and they can sit back and watch their optimal queue be created in front of them.

2.4 Filter Tracks

Filters are only applied when the related confidence weight is greater than zero, ensuring the hard filtering constraints are matching clear user intent and avoiding unnecessary exclusions.

Genre Filtering. Genre preferences are matched using case-insensitive regular expressions, given the variations in naming (e.g., matching "alt rock" to "Alternative Rock"). If that fails, the system falls back to basic substring checks against inconsistent inputs.

Year Filtering. Tracks are kept only if their release year falls within the user's chosen range. This helps make sure the results match the era or trend the user is interested in.

Timestamp Filtering. If the user prompt includes a listening time frame (e.g., "songs I listened to last fall"), the system filters for tracks with at least one listening timestamp during that period. This uses historical listening data to make recommendations more context-aware. Parsing handles variety in timestamp formats and missing data.

If filtering eliminates all candidate tracks at any stage, the system exits early with a warning, not wasting computation on an empty set. By design, songs are not filtered out based on other parameters, such as mood and energy, since those are more subjective.

2.5 Score Tracks

After filtering, the remaining songs are scored using a soft penalty framework. For each parameter, such as mood, energy, popularity, and year, penalties are found based on how far a song's value deviates from the target. Let p_i denote the penalty for feature i, and w_i be the associated weight. The total penalty is:

$$\text{penalty} = \sum_i w_i p_i$$

Each penalty term is computed as follows:

- **Mood penalty:** $p_{\text{mood}} = |m - m^*|$, where m is the song's mood and m^* is the target mood.
- **Energy penalty:** $p_{\text{energy}} = |e - e^*|$, where e is the song's energy and e^* is the target.
- **Popularity penalty:** $p_{\text{popularity}} = |p - p^*|$, based on the song's popularity.

- **Recency penalty:** $p_{\text{recency}} = 1 - \text{recency\_score}$, which decreases as last played date gets older.
- **Favoriteness penalty:** $p_{\text{fav}} = 1 - \frac{c}{c_{\max}}$, where c is the play count of the track and $c_{\max}$ is the maximum play count across all songs.

The final track score is then found via:

$$\text{score} = \exp(-\text{penalty})$$

The exponential transformation maps total penalties to a [0, 1] scale, where lower penalties give higher scores. This mapping is also non-linear, so that small penalty differences are emphasized.

2.6 Pick Songs

Once all songs are scored, a subset is sampled for the final playlist. To maintain non-determinism (so as to keep this tool interesting between uses), the top $3 \times N$ tracks, where N is the desired playlist length, form a candidate pool. Then, sampling is performed without replacement using squared score weights to choose N tracks:

$$P_i = \frac{\text{score}_i^2}{\sum_j \text{score}_j^2}$$

2.7 Order Queue (Flow Sequencing)

The ordering of songs is controlled by the `flow` parameter, which supports three modes: `smooth`, `varied`, and `random`.

The similarity score S between two songs is found as a weighted combination:

$$S = 35 \cdot s_{\text{bpm}} + 35 \cdot s_{\text{key}} + 2 \cdot s_{\text{genre}} + 1 \cdot s_{\text{artist}},$$

where each component s_i ranges from 0 (completely dissimilar) to 1 (identical or maximally similar). BPM similarities are found using normalized absolute differences. Key similarities use custom cyclic key distance functions based around the distance of two keys in the circle of fifths. Genre similarity uses Jaccard index, and artist similarity is binary.

In **smooth** mode, the goal is to create gentle transitions between the N songs. Starting from a random track, each subsequent song is the unpicked one that has the maximum similarity to the track that was just chosen.

In **varied** mode, the goal is add contrast. Each track is selected to be the least similar to the last 3 songs, based on the same similarity metric:

$$\min_{k \in \text{Unpicked}} \left(\frac{1}{3} \sum_{j \in 3 \text{ latest tracks}} S(\text{track}_k, \text{track}_j) \right)$$

In **random** mode, songs are shuffled arbitrarily using `sample(frac=1)`. Similarity between tracks isn't taken into account.

2.8 End

After sequencing, the final playlist is created directly on Spotify, and is displayed for the user to play directly on it. On the terminal, the final playlist is output as a DataFrame, containing selected metadata fields like title, artist, album, BPM, and genre, and its scoring from earlier.

3 Future Steps

While the current system is a strong pipeline for personalized playlist generation, here are some potential improvements:

- **Deterministic and scalable song feature extraction:** The current enrichment pipeline uses GPT-4 to infer musical features, which can be slow and non-deterministic. This was a workaround due to Spotify's deprecation of the audio features API. Future versions should explore alternate deterministic sources.
- **Include modality of key (major vs. minor):** Currently, only the pitch class is represented numerically. Major and minor keys relative to one another are grouped together. Future versions will encode modality explicitly, allowing for more nuanced similarity scoring and mood control.
- **Web-based interface:** The existing CLI version requires local setup. A hosted interface could allow access from anywhere, including mobile devices, without having to run Python scripts.
- **Visual flow mapping:** A visual representation of how tracks transition over time, based on BPM, key, or mood, can help users better understand and adjust flow dynamics.
- **Collaborative playlists:** Extend functionality to let multiple users merge their histories and generating a queue optimized for a whole group.