Building a Spotify Playlist-Based Music Recommendation System

In this guide, we will walk through creating a music recommendation system using a Spotify playlist as the starting point. We'll cover two implementation options – running the project in **Google Colab** (a free Jupyter notebook environment) and in **Google Cloud Vertex AI** (managed ML platform) – and detail how to integrate the Spotify API and Genius API. We will also discuss cost considerations for each approach. The playlist in question contains ~3,200 songs, so our solution must handle that data scale.

1. Environment Setup: Google Colab vs. Google Vertex AI

Google Colab (Free Option): Google Colab is a cloud-based Jupyter notebook service with free GPU/TPU availability. It's ideal for development and small-scale experiments. You can run Python code in your browser without setup. However, Colab sessions are temporary (12-hour limit) and have limited resourcesanalyticsvidhya.com. If needed, you can upgrade to Colab Pro for around $9.99 per month to get longer runtimes and more memory/CPU (and some dedicated Compute Units) – but the free version is usually sufficient for a dataset of this size.

Google Vertex AI (Cloud Option): Google Vertex AI Workbench provides managed JupyterLab notebooks on Google Cloud. This is suitable for more robust or scalable workflows. You can choose a VM instance with desired CPU/GPU specs, and the environment integrates well with other GCP services (BigQuery, Cloud Storage, etc.)levelup.gitconnected.comlevelup.gitconnected.com. Vertex AI allows you to train and deploy models at scale, and offers MLOps features like pipelines and model registrylevelup.gitconnected.com. To use Vertex AI, you'll need a Google Cloud project – new users get $300 free creditscloud.google.com, which is plenty for this project's scope.

Cost Considerations: Running a notebook on Vertex AI incurs charges for the compute instance per hour. For example, a standard 4-vCPU machine costs on the order of $0.15–$0.19 per hourdev.todev.to (so a 10-hour run costs about $1.50). Smaller instances can be as cheap as ~$0.094/hourpump.co. If you only need a couple hours of CPU processing for 3,200 songs, the cost will be well under $1 in most cases. Using a GPU (for deep learning on audio or NLP) would increase cost (e.g. a Tesla T4 ~$0.35–$0.40/hour)pump.co, but for our recommendation approach a GPU isn't strictly necessary. Colab, by contrast, is free (with resource limits), so you might start in Colab to prototype and then consider Vertex AI for long-term runs or deployment.

Summary: **If cost is a concern and the workload is moderate, Colab's free tier is zero-cost and quick to start.** If you need a persistent environment, better security, or want to scale up (or deploy a model as an API), **Vertex AI Workbench is appropriate** – and the costs for a project of this size are modest (likely under a few dollars, often covered by free credits).

2. Spotify API Integration – Data Collection from a Playlist

The first major step is gathering data about the songs in the Spotify playlist. We will use the **Spotify Web API** to fetch track details and audio features. To access Spotify's API, you need to create a **Spotify Developer** account and register an app to obtain API credentials (a *Client ID* and *Client Secret*)spotipy.readthedocs.io. This allows you to authenticate your requests.

Setup Spotify API Access: Install the Spotify API client library for Python. A popular choice is **Spotipy**, a lightweight Python wrapper for Spotify's Web APIspotipy.readthedocs.iospotipy.readthedocs.io. In your Colab or Vertex notebook, install Spotipy via pip and import it:

```
!pip install spotipy
import spotipy
from spotipy.oauth2 import SpotifyClientCredentials
```

After installing, you should authenticate. For simplicity, we can use the **Client Credentials flow** (since we only need read-only access to public data). Set your Spotify Client ID and Secret as environment variables or pass them to SpotifyClientCredentialsspotipy.readthedocs.io. For example:

```
import os
os.environ['SPOTIPY_CLIENT_ID'] = 'YOUR_CLIENT_ID'
os.environ['SPOTIPY_CLIENT_SECRET'] = 'YOUR_CLIENT_SECRET'
spotify = spotipy.Spotify(client_credentials_manager=SpotifyClientCredentials())
```

Now, fetch the playlist tracks. Spotify playlists are identified by an ID (which is in the playlist's URL). For the given playlist (7MNBsBwgsqAsRZkdNE4E5Y), we can retrieve all track items. The Spotify API returns playlists in pages of 100 tracks, so Spotipy's playlist_items or playlist method will handle pagination (you may need to loop using spotify.next() for more than 100 tracks)spotipy.readthedocs.iospotipy.readthedocs.io. For example:

```
playlist_id = "7MNBsBwgsqAsRZkdNE4E5Y"
results = spotify.playlist_items(playlist_id, additional_types=['track'])
tracks = results['items']
while results['next']:
    results = spotify.next(results)
    tracks.extend(results['items'])
```

This gives a list of track entries. From each track item, extract the relevant info: track **ID**, **name**, and **artist name** (primary artist). We will need track IDs for audio feature lookup, and artist/name for lyric search.

Fetching Audio Features: Spotify provides pre-computed **audio features** for each track, which are very useful for content-based recommendations. These features include metrics like *danceability*, *energy*, *valence* (mood positivity), *tempo*, *instrumentalness*, *acousticness*, *liveness*, *speechiness*, and more, as well as the track's popularity and durationmedium.com. We can query these in bulk. Spotipy has audio_features(track_ids) which accepts a list of up to 100 track IDs and returns their feature metrics.

Collect the playlist's track IDs and split into chunks of 100 to call spotify.audio_features(). This will yield a dictionary for each track with its feature values. We can store these in a DataFrame or list. Key features to keep might be: danceability, energy, loudness, valence, tempo, key, mode, etc., since these quantify a

song's sonic characteristicsstratoflow.comstratoflow.com. Spotify's recommendation algorithm itself uses many of these audio attributes to understand a track's "feel"stratoflow.comstratoflow.com.

For example:

```
track_ids = [item['track']['id'] for item in tracks if item['track']]
features_list = []
for i in range(0, len(track_ids), 100):
    batch = track_ids[i:i+100]
    audio_feats = spotify.audio_features(batch)
    features_list.extend(audio_feats)
```

Now we have a list of feature dicts corresponding to our songs. We can merge this with track metadata (name, artist) by matching track IDs.

(Optional) Additional Metadata: You might also use the Spotify API to get each track's full metadata or the artists' info. For instance, artist genres or track release year could be useful. However, for a straightforward recommendation system, the audio features plus lyrics (next step) are sufficient.

Note: Accessing a public playlist's tracks via the Spotify API may not require the user's OAuth if the playlist is public. Using the client credentials token should work for public playlists. If the playlist were private, you'd need to authenticate via OAuth with the playlist-read-private scopedeveloper.spotify.com.

3. Genius API Integration – Fetching Lyrics for Tracks

To enrich our song data with **lyrics**, we will use the Genius API. Lyrics can provide insights into the song's themes, mood, and even help capture aspects of the music not obvious from audio features alone (Spotify has noted that analyzing lyrics and even cultural context can improve recommendationsstratoflow.comstratoflow.com). For example, Spotify's content-based filtering considers lyrical content as one factor in grouping similar songsstratoflow.comstratoflow.com.

Setup Genius API Access: Create a free account on Genius (genius.com) and register an API client to obtain an **Access Token**pypi.org. There is a Python library called **lyricsgenius** that makes it easy to use the Genius APIpypi.orgpypi.org. Install it with pip in your notebook:

```
!pip install lyricsgenius
import lyricsgenius
genius = lyricsgenius.Genius("YOUR_GENIUS_ACCESS_TOKEN")
```

Now, for each track, use the library to search for lyrics. We can call genius.search_song(title, artist) to find a song on Geniuspypi.org. It's wise to provide both song title and artist name to get accurate matches, especially since many songs have common titles. For example:

```
for track in tracks:
    title = track['track']['name']
    artist = track['track']['artists'][0]['name']
```

```
try:
    song = genius.search_song(title, artist)
    lyrics = song.lyrics if song else None
except Exception as e:
    lyrics = None
# store lyrics in your data structure
```

We wrap it in a try/except because some queries might fail or not find a result. The song.lyrics will contain the full lyrics text if foundpypi.org. You might want to remove section headers like "[Chorus]" – lyricsgenius has options to do that (e.g. genius.remove_section_headers = True)pypi.org.

Performance Tip: Fetching lyrics for ~3200 songs involves 3200 API calls to Genius. Be mindful of rate limits. The Genius API allows about 60 requests per minute (per their documentation), so you may need to pause or throttle your requests to avoid being blocked. In practice, you can insert a short time.sleep(1) between calls, or fetch in batches. If speed is an issue, consider multithreading or saving interim results periodically (so you can resume if interrupted). Also note that not every song will be on Genius or correctly matched – expect a few misses for very obscure tracks or instrumental pieces.

After this step, you should augment your song data with a **Lyrics** field. This textual data will allow us to incorporate NLP techniques or at least consider song themes in recommendations. If storage is a concern, you can store just the lyrics or certain features of them (like sentiment or key words), but for our purposes keeping the full lyrics text for analysis is fine.

4. Building the Recommendation System

With data in hand (audio features + lyrics for each song in the playlist), we have two main paths to generate recommendations. We can either leverage Spotify's built-in recommendation engine to get similar songs, or build a **custom content-based recommender** using our data. We will outline both options:

Option A: Using Spotify's Recommendation API (Quick Start)

Spotify provides an endpoint to get recommended tracks based on seed songs/artists/genresdeveloper.spotify.comdeveloper.spotify.com. This is the same service that powers features like "Discover Weekly" and "Recommended Songs" on Spotify. It uses a mix of collaborative filtering and content-based methods internally. You can take a few songs from the playlist as "seeds" and ask Spotify for recommendations similar to those.

How to use: The Spotipy library offers spotify.recommendations() where you can specify up to 5 seed tracks, seed artists, or seed genres, and various target audio feature rangesspotipy.readthedocs.io. For example, to get 20 recommendations based on a couple of songs:

```
seed_tracks = [track_ids[0], track_ids[1], track_ids[2]]  # 1-5 seed track IDs
recs = spotify.recommendations(seed_tracks=seed_tracks, limit=20)
rec_tracks = [t['name'] + " - " + t['artists'][0]['name'] for t in recs['tracks']]
```

This will return tracks not in the seed list that Spotify considers musically related. Under the hood, Spotify's system is using patterns from millions of users and content analysis (audio & lyrics) to find songs that "go with" your seedsstratoflow.comstratoflow.com. In fact, Spotify's algorithm maps songs in a high-dimensional space of audio attributes and listening patterns, so it can pull songs that are close to your favorites on this "music map"stratoflow.comstratoflow.com.

Using the playlist as a basis: With 3200 songs, you have a lot of possible seeds. You might start by choosing a handful of representative tracks (perhaps the most popular ones, or a few from each genre cluster within the playlist). You can also use audio features to guide the recommendation call – for instance, if your playlist is mostly high-energy dance tracks, you might set target_energy=0.8 or min_tempo=100 in the API call to bias the recs. The Spotify recommendation API allows such feature filters to fine-tune resultsmedium.commedium.com.

To cover the breadth of a 3200-song playlist, you could iterate this process: cluster the playlist songs (explained below) and pick one seed from each cluster to get recs, or randomly sample seeds. Collect all unique recommendations returned. This could yield a large set of new songs that **Spotify** believes are similar in style to those in your playlist.

Pros: This approach is easy and takes advantage of Spotify's sophisticated algorithms (including collaborative filtering with other users' data, which you wouldn't have on your own). It's also fast to implement – essentially a few API calls.

Cons: You have less control over the criteria, and you rely on Spotify's black-box. If you want a truly custom system (e.g., to experiment with using lyrics in a new way), you'd prefer Option B. Also, Spotify's recs might include songs that *are* already in your playlist (though usually it avoids that) or very popular songs you already know.

Option B: Custom Content-Based Recommendation Model

In a content-based approach, we use the features of the songs themselves (the content: audio attributes and lyrics) to find similarity and recommend music. Here's a step-by-step plan:

4.1 Feature Engineering: We have two modalities of data for each song:

- **Audio feature vector:** Using the Spotify audio features (danceability, energy, etc.), we can construct a numeric vector for each track. For example, Spotify provides about 11 key audio features. We might take a subset or all of them (perhaps [danceability, energy, valence, acousticness, instrumentalness, tempo, loudness, speechiness, liveness, mode, key] etc.). It can be useful to normalize these features (since they have different scales; e.g., loudness is in dB, tempo in BPM, others are 0-1).

- **Lyrics/text features:** The lyrics can be transformed into features using NLP techniques. A simple approach is to compute a **TF-IDF vector** for the lyrics (treat each song's lyrics as a document). This yields a high-dimensional sparse vector indicating important words. A more advanced approach is to use a pre-trained language model (like a Sentence-BERT or other transformer) to get an **embedding** of the lyric text – a dense vector (say 768 dimensions) that captures semantic meaning. For example, we could use a model that embeds the lyrics such that songs with similar

themes or sentiments end up with vectors close together. (As context, Spotify reportedly uses NLP to analyze song lyrics and even blog articles about music to extract descriptive terms, integrating that into their recommendation enginestratoflow.comstratoflow.com.)

For manageability, TF-IDF + dimensionality reduction might be enough here. You could generate TF-IDF vectors for the ~3200 lyrics, then use PCA or TruncatedSVD to reduce to, say, 50 components that capture the main variation. Alternatively, use a language model embedding directly to 300-768 dimensions and perhaps reduce further if needed.

4.2 Combining Features: Now we have an audio feature vector and a lyrics-based vector for each song. We should combine them into a single representation. One simple way: **concatenate** the vectors (after scaling each part appropriately). For instance, you have a 10-dimensional audio vector and a 50-dimensional lyrics vector, you make a 60-dim combined feature. Another approach is to weight them – e.g., you might give more weight to audio similarity than lyrical similarity if that matters more for your recommendations. Some experimentation can be done here (e.g., via cross-validation if you had a way to evaluate, or just intuition).

4.3 Similarity Computation: To recommend songs similar to a given song, compute the distance between feature vectors. A common choice is **cosine similarity** (measures the angle between two vectors). Cosine similarity works well for high-dimensional data like text. You can also use Euclidean distance (after normalization) which in this context is fairly similar.

Given a target song's combined feature vector, you calculate its cosine similarity with all other songs' vectors. Then rank the songs by similarity score. High scores indicate the songs are alike in sound and lyrical content. Since the goal is to **recommend new songs (not in the playlist)**, ideally we would have a *catalog* of candidate songs outside the original 3200 to compare against. This is an important point: A content-based system can only recommend items that it knows about. If we only build the feature database from the playlist itself, then we can only "recommend" songs that are already in that playlist (which isn't useful to expand the playlist). So, you should consider obtaining a larger pool of songs to serve as potential recommendations.

Getting a candidate pool: There are a few ways to get additional songs' data:

- Use the Spotify API to get tracks by **related artists**. For each artist in your playlist, fetch their top tracks or albums (Spotify has endpoints for an artist's top tracks and related artists). This could net a lot of songs that are not in your playlist but likely in similar genres.
- Use the **Spotify "Recommended" endpoint** (Option A) as a source of candidates. You could call it with many different seeds (as discussed) and union the results.
- Use an existing dataset: There are public music datasets (like Million Song Dataset, etc.), but integrating those might be overkill. However, something like the Million Song Dataset doesn't have lyrics, and linking to Spotify might be complicated. Sticking to Spotify API is simpler.

Suppose we gather an extra few thousand songs as candidates and compute their features (audio + maybe fetch lyrics for those too if we want to include lyrics in similarity). This increases the upfront data collection effort, but results in a more comprehensive recommender.

If you prefer not to expand the dataset, another interpretation is that the recommendation system could operate **within the 3200-song playlist itself** – for example, it could be used to generate sub-recommendations: "if you like song X from this playlist, here are 5 others from the playlist you might enjoy" (basically finding similar songs within the playlist). This might be useful if the playlist is a broad collection and you want to highlight relationships among its songs. In that case, your candidate pool is the playlist itself (excluding the song in question).

4.4 Model or Algorithm: This content-based approach doesn't necessarily require "training" a machine learning model; it's mostly a similarity search problem. However, you can use algorithms like **K-Nearest Neighbors (KNN)** on the feature space. For example, train a KNN model on the combined feature vectors, so given a new song vector it can quickly find the K closest songs in the database. Since our data size is a few thousand, even brute-force similarity computation is fine (a few thousand dot products is trivial for modern CPUs).

Alternatively, you could cluster the songs using an algorithm like **K-means** on the feature space to identify distinct groups of musical style/lyrical theme within the playlist. With clusters, you might label each cluster by its dominant genre or mood, and then recommend songs from the same cluster. If you add outside songs into the clustering, they will mix in the clusters as well.

4.5 Recommendation Output: Once you have a similarity measure, you can generate recommendations in a couple of ways:

- **Song-to-song recommendation:** For each song in the playlist (or each song a user likes), find the top N most similar songs (from the candidate pool) that are not already known. For example, if a user picks a favorite song, you return "songs similar to that one."
- **Playlist-level recommendation:** Compute an "average" vector for the entire playlist or identify a few clusters in the playlist. Then find new songs that are closest to those centroids. Essentially, this would give you songs that fit the overall vibe of the playlist but aren't in it yet.

Because we also have lyrics, our recommendations can take into account thematic similarity. For instance, you could ensure that if the playlist has a theme (say, hopeful lyrics), the recommended songs also have similar sentiment. **Spotify's own system indeed uses lyrical analysis to refine recommendations**, e.g., ensuring songs have similar lyrical themes for more nuanced matchesstratoflow.comstratoflow.com.

4.6 Example: Imagine our playlist has a song with high danceability and positive lyrics about love. The audio features alone might match it with any upbeat dance track, but including lyrics might favor songs that are also about romantic themes. Our content-based model might then recommend a song by a different artist that has a similar dance beat *and* love-related lyrics – something Spotify's collaborative filtering might miss if that song isn't commonly playlisted with our original.

Evaluation: Without user feedback or a test set, evaluating the quality of recommendations is tricky. It will be somewhat subjective – you can listen or examine if the recs make sense. If you had multiple playlists or examples of "liked vs not liked" songs, you could validate that your system ranks liked songs

higher. Since this is a research/implementation project, the focus is on the process rather than quantitatively measuring accuracy.

5. Implementation in Colab vs Vertex, and Cost Estimation

Regardless of approach (Spotify API or custom model), you can implement the code in either Colab or Vertex AI.

Using Colab: You would execute all the above steps in a Colab notebook. Colab allows installation of libraries (!pip install ...) and will let you save results to Google Drive if needed. While running, keep an eye on memory – 3200 songs with lyrics and features is not too heavy (likely a few tens of MB of text at most, which Colab can handle). The main limitation might be runtime if the lyrics fetching takes a long time; but you can split the work or save intermediate data (e.g., after fetching features and lyrics, save a JSON or CSV to Drive so you don't have to re-fetch if the session resets).

Colab's free tier should be sufficient; the whole process might take on the order of an hour or two (most of that waiting for API calls). This is well under the 12-hour limit. If you were to do heavier ML training (say training a deep neural network on audio spectrograms, which we are **not** doing here), you might need Colab Pro for more GPU time. But for our feature-based approach, CPU is fine.

Using Vertex AI Workbench: In Vertex AI, you can create a notebook instance (choose a Machine Type, e.g., "n1-standard-4" which has 4 vCPUs, 15 GB RAM). That instance will cost about ~$0.15–$0.20 per hour as noted. You can run the same code there. The advantage is the instance can run longer and won't be preempted, and you can shut it down or schedule it as needed. Vertex notebooks have persistent storage for the notebook files, but remember to save any large data (like the lyrics dataset) to durable storage (e.g., Cloud Storage bucket) or else snapshot the VM, because the instance's disk will be deleted if you delete the notebook.

One neat Vertex feature: you could store your song data in **BigQuery** and even use **BigQuery ML** for certain tasks (for example, performing clustering or similarity search via SQL). However, given the moderate size, it's probably easier to do in pure Python/pandas in memory.

Deploying a Model (optional): If you build a content-based recommender and you want to serve it (e.g., via an API endpoint), Vertex AI can host your model. You'd package the similarity logic into a model server (perhaps a custom Flask app or a Vertex Prediction custom model). Vertex AI endpoints cost additional (roughly $0.25 per node hour for deployment plus per prediction costs)reddit.comdev.to. For a hobby project, deployment on Vertex might not be necessary – you could simply use the results within your notebook or export them.

Cost Summary: Running the entire pipeline in Vertex AI notebook for a one-time analysis: assume ~2 hours of an n1-standard-4 instance. That's roughly $0.30–$0.40 totaldev.to. Fetching data from Spotify/Genius APIs has no direct monetary cost (they're free to use, with rate limits). Storing data: a few MB in Cloud Storage or BigQuery costs only cents (first 10 GB of many services are often free). If you decided to train more complex ML models or handle audio files, costs would increase accordingly (e.g., training a neural net on audio with a GPU might cost a few dollars as shown in examplesdev.todev.to). But our feature-based method is very light.

In Google Colab, the cost is essentially zero unless you opt for Pro. If you did use Colab Pro for a month at $9.99, that's the upper bound – still relatively low.

Conclusion: For this project of ~3200 songs, **you can absolutely implement it on Colab for free**, using Spotipy and lyricsgenius to gather data and scikit-learn/pandas for the recommender logic. If you want to integrate into a larger pipeline or need more stability, **Vertex AI** offers a managed solution with minimal cost (likely covered by free credits). In either case, integrating the **Spotify API** to get audio attributes and the **Genius API** to get lyrics will significantly enhance the recommendation system's ability to find musically and thematically similar songs. By following the steps above, you can build a personalized music recommendation system that starts from your favorite playlist and suggests new tracks you'll likely enjoy – just as Spotify does, but tailored with your own tunable criteria.

Sources:

- Spotify API documentation – retrieving playlists and audio featuresmedium.commedium.com. These features (danceability, energy, etc.) characterize songs and are used in recommender systemsstratoflow.comstratoflow.com.
- Guide to using Spotipy (Python Spotify API library)spotipy.readthedocs.iospotipy.readthedocs.io and code examples for playlist extraction.
- Medium article on Spotify & Genius API usage (Raizel Bernstein) – explains installing Spotipy and lyricsgenius, and obtaining API keysraizelb.medium.comraizelb.medium.com.
- **LyricsGenius** documentation – usage of the Genius API for fetching song lyricspypi.orgpypi.org.
- Discussion of Spotify's recommendation algorithm combining collaborative filtering and content (audio analysis + lyrics)stratoflow.comstratoflow.com, demonstrating the importance of integrating both audio features and lyrical data for music recommendations.
- Google Cloud pricing examples for Vertex AI notebook and training – showing that even a multi-hour run on a standard machine costs only a few dollars at mostdev.todev.to (and likely free with credits).