Step 1: Importing Modules

```
# SYSTEM & UTILITIES
import os  # file and path operations
import time  # measure time or add delays
import json  # read/write JSON data
import requests  # make HTTP API calls
from IPython.display import clear_output  # clear notebook output

# CORE SCIENTIFIC LIBRARIES
import numpy as np  # numerical computations
import pandas as pd  # data manipulation and analysis
import torch  # GPU acceleration and tensor operations

# MACHINE LEARNING & NLP
from sentence_transformers import SentenceTransformer, util  # text embeddings and similarity
from sklearn.metrics.pairwise import cosine_similarity  # cosine similarity computation
from sklearn.preprocessing import normalize  # normalize data vectors
from sklearn.decomposition import PCA  # dimensionality reduction (e.g., for visualization)
from huggingface_hub import hf_hub_download  # download pretrained models
from openai import OpenAI  # interact with OpenAI API

# VISUALIZATION & UI
import plotly.graph_objs as go  # interactive visualizations
import plotly.io as pio  # plotly rendering interface
import gradio as gr  # build interactive web UIs
```

Step 2: Viewing Dataset

```
#Reading CSV
csv_path = '/content/top_4500_songs.csv'
df = pd.read_csv(csv_path)
df.head()
```

	Unnamed: 0	track_id	artists	album_name	track_name	popularity	duration_ms	explicit	dan
0	81051	3nqQXoyQOWXiESFLIDF1hG	Sam Smith;Kim Petras	Unholy (feat. Kim Petras)	Unholy (feat. Kim Petras)	100	156943	False	
1	51664	2tTmW7RDtMQtBk7m2rYeSw	Bizarrap;Quevedo	Quevedo: Bzrp Music Sessions, Vol. 52	Quevedo: Bzrp Music Sessions, Vol. 52	99	198937	False	
2	89411	5ww2BF9slyYgNOk37BlC4u	Manuel Turizo	La Bachata	La Bachata	98	162637	False	
3	30003	4uUG5RXrOk84mYEfFvj3cK	David Guetta;Bebe Rexha	I'm Good (Blue)	I'm Good (Blue)	98	175238	True	
4	88405	6Sq7ltF9Qa7SNFBsV5Cogx	Bad Bunny;Chencho Corleone	Un Verano Sin Ti	Me Porto Bonito	97	178567	True	

5 rows × 21 columns

```
Start coding or generate with AI.
```

Step 3: Dropping null values

```
num_songs = len(df)
print(f"No of Songs: {num_songs}")

df = df.dropna(subset=['artists', 'album_name', 'track_name', 'track_genre'])
```

```
No of Songs: 4500
```

Step 4: Importing models from Huggingface

Before we move to **audio embeddings (MuQ)**, let's load our **text-based embedding models**.
These models help our app understand the *meaning* of words and phrases —

like when someone says “chill songs like Blinding Lights” — even if those exact words don’t appear in the dataset.

🎯 **What are embeddings?** Embeddings are numerical representations of text.
They capture *semantic meaning* — so similar words or phrases have closer embeddings in vector space.

For example:

> “energetic pop song” and “upbeat track” will have embeddings that are close to each other.

💡 **The models we’re loading:**

Model	Description	Use Case
MiniLM	Small & fast model	Great for quick testing or limited resources
MPNet	Balanced accuracy model	Good all-rounder for general text similarity
BGE (Base)	High-performing English model	Ideal for nuanced understanding of prompts

⚙️ **How it works:**

1. The code defines a dictionary of available models.
2. It downloads and initializes each model using `SentenceTransformer`.
3. These models are now ready to convert any text (e.g., song titles, lyrics, or prompts) into embedding vectors for comparison.

✅ Once you see the message:

```
# Define a dictionary of available embedding models and their Hugging Face repo IDs
embedding_models = {
    "MiniLM": "sentence-transformers/all-MiniLM-L6-v2",  # Fast and lightweight
    "MPNet": "sentence-transformers/all-mpnet-base-v2",  # High accuracy general-purpose model
    "BGE": "BAAI/bge-small-en-v1.5"  # Compact but strong performance for English
}

# Choose which models you want to load (you can pick one or multiple)
selected_models = ['BGE', 'MPNet', 'MiniLM']

# Loop through and download + load each selected model
for model_name in selected_models:
    repo_id = embedding_models[model_name]

    # Try downloading model config to check if it exists
    try:
        hf_hub_download(repo_id=repo_id, filename='config.json')
    except:
        pass

    # Load the SentenceTransformer model from Hugging Face
    embedding_models[model_name] = SentenceTransformer(repo_id)
    print(f"✅ Embedding model '{model_name}' loaded successfully.")
```

```
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/t
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
config.json: 100%                                                 743/743 [00:00<00:00, 37.8kB/s]
modules.json: 100%                                                349/349 [00:00<00:00, 26.8kB/s]
config_sentence_transformers.json: 100%                                   124/124 [00:00<00:00, 11.5kB/s]
README.md:       94.8k/? [00:00<00:00, 9.12MB/s]
sentence_bert_config.json: 100%                                   52.0/52.0 [00:00<00:00, 5.58kB/s]
model.safetensors: 100%                                           133M/133M [00:01<00:00, 151MB/s]
tokenizer_config.json: 100%                                       366/366 [00:00<00:00, 47.1kB/s]
vocab.txt:       232k/? [00:00<00:00, 11.2MB/s]
tokenizer.json:       711k/? [00:00<00:00, 41.0MB/s]
special_tokens_map.json: 100%                                     125/125 [00:00<00:00, 13.1kB/s]
config.json: 100%                                                 190/190 [00:00<00:00, 21.0kB/s]
✅ Embedding model 'BGE' loaded successfully.
config.json: 100%                                                 571/571 [00:00<00:00, 69.4kB/s]
modules.json: 100%                                                349/349 [00:00<00:00, 32.7kB/s]
config_sentence_transformers.json: 100%                                   116/116 [00:00<00:00, 13.3kB/s]
README.md:       11.6k/? [00:00<00:00, 1.04MB/s]
sentence_bert_config.json: 100%                                   53.0/53.0 [00:00<00:00, 5.94kB/s]
model.safetensors: 100%                                           438M/438M [00:04<00:00, 83.4MB/s]
tokenizer_config.json: 100%                                       363/363 [00:00<00:00, 43.4kB/s]
vocab.txt:       232k/? [00:00<00:00, 15.8MB/s]
tokenizer.json:       466k/? [00:00<00:00, 31.7MB/s]
special_tokens_map.json: 100%                                     239/239 [00:00<00:00, 25.7kB/s]
config.json: 100%                                                 190/190 [00:00<00:00, 13.1kB/s]
✅ Embedding model 'MPNet' loaded successfully.
config.json: 100%                                                 612/612 [00:00<00:00, 40.4kB/s]
modules.json: 100%                                                349/349 [00:00<00:00, 23.0kB/s]
config_sentence_transformers.json: 100%                                   116/116 [00:00<00:00, 6.37kB/s]
README.md:       10.5k/? [00:00<00:00, 639kB/s]
sentence_bert_config.json: 100%                                   53.0/53.0 [00:00<00:00, 3.71kB/s]
model.safetensors: 100%                                           90.9M/90.9M [00:00<00:00, 214MB/s]
tokenizer_config.json: 100%                                       350/350 [00:00<00:00, 32.7kB/s]
vocab.txt:       232k/? [00:00<00:00, 14.5MB/s]
tokenizer.json:       466k/? [00:00<00:00, 30.5MB/s]
special_tokens_map.json: 100%                                     112/112 [00:00<00:00, 14.2kB/s]
config.json: 100%                                                 190/190 [00:00<00:00, 20.5kB/s]
✅ Embedding model 'MiniLM' loaded successfully.
```

Note: Test how embedding works here :)

Before we dive into *music embeddings*, let's first **see how text embeddings behave**.
This step helps visualize what our models (MiniLM, MPNet, BGE) are actually doing behind the scenes.

🧠 **Concept:** An *embedding* is a numerical representation of meaning.
When two words (or phrases) have similar meanings, their embeddings are *close* together in vector space.
For example:

- "AI" and "machine learning" → very close
- "banana" and "keyboard" → far apart

🔍 **What this code does:**

1. Creates a list of words (`word_dict`) — some fruits 🍌 and some tech terms 💻.
2. Converts each word into an embedding using all three models.
3. Asks for a user input (e.g., "apple" or "AI") and finds the *most semantically similar* words.
4. Prints the top 5 matches with similarity scores for each model.

💡 **Try this:**

- Type **"AI"**, **"computer"**, or **"machine learning"** → you'll see tech words rank high.
- Type **"banana"** or **"grape"** → fruit-related words will appear.

This step shows how models like **BGE** understand *meaning*, not just spelling.
In the next steps, we'll extend this same concept from *text* → to *music* 🎶 using MuQ.

```
word_dict = [
    "apple", "banana", "cherry", "fruit salad", "computer",
    "machine learning", "deep learning", "AI model", "openAI", "python programming",
    "grape", "orange", "mango", "keyboard", "mouse", "laptop", "notebook"
]

# 4. Precompute embeddings for dictionary (for each model)
dict_embeddings = {}
for model_name in selected_models:
    model = embedding_models[model_name]
    dict_embeddings[model_name] = model.encode(word_dict, convert_to_tensor=True)

# 5. Function to find top N matches
def find_top_matches(user_input, top_n=5):
    results = {}
    for model_name in selected_models:
        model = embedding_models[model_name]
        input_emb = model.encode(user_input, convert_to_tensor=True)
        similarities = util.cos_sim(input_emb, dict_embeddings[model_name])[0]
        top_indices = torch.topk(similarities, k=top_n).indices
        top_words = [word_dict[idx] for idx in top_indices]
        top_scores = [similarities[idx].item() for idx in top_indices]
        results[model_name] = list(zip(top_words, top_scores))
    return results

user_input = input("Enter your text: ")
matches = find_top_matches(user_input)
for model_name, top_matches in matches.items():
    print(f"\nTop matches using {model_name}:")
    for word, score in top_matches:
        print(f"{word} — {score:.4f}")
```

```
Enter your text: potable PC

Top matches using BGE:
computer — 0.7662
laptop — 0.7552
keyboard — 0.6944
mouse — 0.6902
notebook — 0.6668

Top matches using MPNet:
computer — 0.4821
laptop — 0.4701
notebook — 0.4006
keyboard — 0.2951
mouse — 0.2747

Top matches using MiniLM:
computer — 0.4926
laptop — 0.4174
keyboard — 0.3738
apple — 0.3587
notebook — 0.3446
```

🎵 Step 5 – Embedding Songs

Now that our embedding models are ready, we'll use them to *represent each song numerically*
— a crucial step before we can recommend music intelligently.

🧠 **What's happening here:** Each song in our dataset has metadata like genre, tempo, loudness, danceability, and energy.
We're converting this metadata into a descriptive sentence for the model to read, such as:

> "Genre: Pop, 85% danceable, 78% energy, 120 BPM, clean track."

This allows the text-based embedding models (BGE, MPNet, MiniLM) to understand the *musical meaning*
of each song — even before we introduce audio embeddings.

📦 **How it works:**

1. `song_to_text()` converts song features into natural language summaries.
2. Each summary is passed through the embedding models to create high-dimensional vectors.
3. The vectors are **normalized** and saved into `.jsonl` files (one per model).
   These embeddings represent the *musical fingerprint* of each track.

🪄 **Why this matters:**

- Songs that "feel" similar (tempo, mood, energy) will have embeddings closer together.

- Later, we'll compare new songs or prompts using **cosine similarity** to find matches.
- This creates the foundation for an AI model that *understands* music structure and mood.

🎨 **Bonus:**
The notebook also includes a 3D visualization function (`visualize_embeddings_3d_interactive`) that reduces embeddings into 3 dimensions using PCA and displays them in a scatter plot — so you can actually *see* clusters of similar songs!

✅ Once you see a message like:

```
# Install and import required libraries
!pip install jsonlines
import jsonlines
from sklearn.preprocessing import normalize
import numpy as np
from sentence_transformers import SentenceTransformer
import plotly.graph_objects as go
from sklearn.decomposition import PCA

# --- Function to convert song metadata into a descriptive text string ---
def song_to_text(row):
    """
    Convert a song's metadata row into a descriptive text string
    that summarizes its genre, audio features, and explicitness.
    """
    explicit_text = "explicit" if row['explicit'] else "clean"
    return (
        f"genre: {row['track_genre']}. "
        f"This song is {row['danceability']*100:.0f}% danceable, "
        f"energy level {row['energy']*100:.0f}%, "
        f"loudness {row['loudness']} dB, "
        f"speechiness {row['speechiness']*100:.0f}%, "
        f"acousticness {row['acousticness']*100:.0f}%, "
        f"instrumentalness {row['instrumentalness']*100:.0f}%, "
        f"liveness {row['liveness']*100:.0f}%, "
        f"valence {row['valence']*100:.0f}%, "
        f"tempo {row['tempo']} BPM. "
        f"It is an {explicit_text} track."
    )

# Generate descriptive texts for all songs in the dataframe
song_texts = df.apply(song_to_text, axis=1)

# Dictionary to store embeddings for different models
model_embeddings = {}

# Set plotly renderer for Google Colab (for interactive plots)
pio.renderers.default = "colab"

# --- Generate embeddings for each selected model ---
for name in selected_models:
    print(f"Generating embeddings with {name}...")

    # Load the embedding model
    model_path = embedding_models[name]
    if isinstance(model_path, SentenceTransformer):
        model = model_path
    else:
        model = SentenceTransformer(str(model_path))

    # Encode all song texts to embeddings and normalize them
    emb = model.encode(song_texts.tolist(), show_progress_bar=True, convert_to_numpy=True)
    emb = normalize(emb)  # Normalize to unit vectors
    model_embeddings[name] = emb

    # Save embeddings to a JSONL file with track_id, text, and embedding
    output_file = f"{name}_song_embeddings.jsonl"
    with jsonlines.open(output_file, mode='w') as writer:
        for idx, (_, row) in enumerate(df.iterrows()):
            writer.write({
                "track_id": row['track_id'],
                "text": song_texts.iloc[idx],
                "embedding": emb[idx].tolist()
            })

    print(f"Text embeddings for {name} saved successfully as JSONL: {output_file}")

print("All selected model embeddings generated, normalized, and stored in JSONL format.")
```

```
# --- Function to visualize embeddings from multiple models in 3D ---
def visualize_embeddings_3d_interactive(embeddings_dict, df, n_samples=None):
    """
    Create an interactive 3D scatter plot of embeddings from multiple models.
    Allows selection of models and comparison between them.

    Parameters:
    - embeddings_dict: dict of model_name -> embeddings (numpy arrays)
    - df: dataframe containing song metadata
    - n_samples: number of random samples to visualize (for performance)
    """

    all_traces = []  # List to store Plotly traces
    model_names = list(embeddings_dict.keys())

    # Choose a subset of songs if n_samples is specified
    if n_samples is not None and n_samples < len(df):
        sample_indices = np.random.choice(len(df), n_samples, replace=False)
        sample_indices = np.sort(sample_indices)
    else:
        sample_indices = np.arange(len(df))
        n_samples = len(df)

    sampled_df = df.iloc[sample_indices].reset_index(drop=True)
    song_names = sampled_df['track_name'].tolist()

    color_schemes = ['Viridis', 'Plasma', 'Inferno', 'Magma', 'Cividis']

    # Create a 3D scatter trace for each model
    for idx, model_name in enumerate(model_names):
        emb = embeddings_dict[model_name][sample_indices]

        # Reduce embedding dimensionality to 3 for visualization using PCA
        pca = PCA(n_components=3)
        emb_3d = pca.fit_transform(emb)

        # Prepare hover text with song and artist info
        hover_texts = []
        for i, row_idx in enumerate(sample_indices):
            row = df.iloc[row_idx]
            hover_text = (
                f"<b>{row['track_name']}</b><br>"
                f"Artist: {row['artists']}<br>"
                f"Genre: {row['track_genre']}<br>"
                f"Model: {model_name}"
            )
            hover_texts.append(hover_text)

        # Create the Plotly scatter3d trace
        trace = go.Scatter3d(
            x=emb_3d[:, 0],
            y=emb_3d[:, 1],
            z=emb_3d[:, 2],
            mode='markers+text',
            name=model_name,
            text=sampled_df['track_name'],
            hovertext=hover_texts,
            hoverinfo='text',
            visible=True if idx == 0 else 'legendonly',
            marker=dict(
                size=6,
                color=np.arange(len(emb_3d)),
                colorscale=color_schemes[idx % len(color_schemes)],
                showscale=True,
                colorbar=dict(
                    title=model_name,
                    x=1 + (idx * 0.15),
                    len=0.5
                ),
                line=dict(width=0.5, color='white')
            ),
            textposition='top center',
            textfont=dict(size=8)
        )
        all_traces.append(trace)

    # Build the figure with all traces
    fig = go.Figure(data=all_traces)

    # Layout settings for the 3D scatter plot
    fig.update_layout(
        title={
```

```
            'text': f'3D Embeddings Comparison - {n_samples} Songs Across {len(model_names)} Models',
            'x': 0.5,
            'xanchor': 'center',
            'font': {'size': 20}
        },
        scene=dict(
            xaxis=dict(title='PC1', backgroundcolor="rgb(230, 230,230)", gridcolor="white"),
            yaxis=dict(title='PC2', backgroundcolor="rgb(230, 230,230)", gridcolor="white"),
            zaxis=dict(title='PC3', backgroundcolor="rgb(230, 230,230)", gridcolor="white"),
            camera=dict(eye=dict(x=1.5, y=1.5, z=1.5)),
            aspectmode='cube'
        ),
        width=1400,
        height=900,
        showlegend=True,
        legend=dict(
            x=0.02,
            y=0.98,
            bgcolor='rgba(255, 255, 255, 0.8)',
            bordercolor='black',
            borderwidth=1
        ),
        hovermode='closest',
        updatemenus=[  # Dropdown menu for selecting models
            dict(
                buttons=[  # Buttons for all models, individual models, and pairwise comparison
                    dict(
                        label="All Models",
                        method="update",
                        args=[{"visible": [True] * len(model_names)}]
                    )
                ] + [
                    dict(
                        label=model_name,
                        method="update",
                        args=[{"visible": [i == idx for i in range(len(model_names))]}]
                    )
                    for idx, model_name in enumerate(model_names)
                ] + [
                    dict(
                        label=f"Compare: {model_names[i]} vs {model_names[j]}",
                        method="update",
                        args=[{"visible": [idx in [i, j] for idx in range(len(model_names))]}]
                    )
                    for i in range(len(model_names))
                    for j in range(i+1, len(model_names))
                ],
                direction="down",
                showactive=True,
                x=0.02,
                xanchor="left",
                y=0.85,
                yanchor="top",
                bgcolor='rgba(255, 255, 255, 0.9)',
                bordercolor='black',
                borderwidth=1
            ),
        ],
        annotations=[  # Label for dropdown
            dict(
                text="Select Models:",
                x=0.02,
                y=0.88,
                xref="paper",
                yref="paper",
                showarrow=False,
                font=dict(size=12, color="black"),
                bgcolor='rgba(255, 255, 255, 0.8)'
            )
        ]
    )

    # Save the interactive plot to an HTML file
    filename = f'embeddings_3d_all_models_{n_samples}_songs.html'
    fig.write_html(filename)

    return fig

# --- Visualize embeddings interactively ---
# Note: Rendering in Colab can be slow for large datasets
# It's safer to visualize only a sample (e.g., 30 songs)
```

```
fig = visualize_embeddings_3d_interactive(model_embeddings, df, n_samples=30)
```

```
Collecting jsonlines
  Downloading jsonlines-4.0.0-py3-none-any.whl.metadata (1.6 kB)
Requirement already satisfied: attrs>=19.2.0 in /usr/local/lib/python3.12/dist-packages (from jsonlines) (25.4.0)
Downloading jsonlines-4.0.0-py3-none-any.whl (8.7 kB)
Installing collected packages: jsonlines
Successfully installed jsonlines-4.0.0
Generating embeddings with BGE...
Batches: 100%                                          141/141 [00:05<00:00, 27.25it/s]
Text embeddings for BGE saved successfully as JSONL: BGE_song_embeddings.jsonl
Generating embeddings with MPNet...
Batches: 100%                                          141/141 [00:18<00:00,  8.05it/s]
Text embeddings for MPNet saved successfully as JSONL: MPNet_song_embeddings.jsonl
Generating embeddings with MiniLM...
Batches: 100%                                          141/141 [00:03<00:00, 47.22it/s]
Text embeddings for MiniLM saved successfully as JSONL: MiniLM_song_embeddings.jsonl
All selected model embeddings generated, normalized, and stored in JSONL format.
```

🔗 Step 6 – Combining Text and Audio Embeddings

Now that we've created:

- **Text embeddings** (from models like BGE, MPNet, and MiniLM)
- **Audio embeddings** (from MuQ)

...it's time to bring them together!

🎯 **Goal:** This step converts all our `.jsonl` files into compact `.npy` (NumPy) format and aligns text and audio embeddings for each track — allowing the model to use both meaning and sound.

🧠 **What's happening in this function:**

1. **Detect & load embeddings**
   - Finds all available text embedding files (`*_song_embeddings.jsonl`)
   - Finds the audio embedding file (`Final_Audio_Embeddings.jsonl`)
   - Loads them into memory and maps them by `track_id`
2. **Merge the two types of embeddings**
   - For each track, matches the song's text and audio vectors
   - Creates three separate `.npy` outputs:
     - `text_only` embeddings
     - `text+audio` combined embeddings
     - `audio_only` embeddings
3. **Save optimized files**
   - Saves the merged embeddings in a structured `converted_embeddings` folder
   - Ensures everything is ready for similarity search and recommendation

💡 **Why this matters:** By combining text + audio embeddings:

- The system can now understand both *semantic* (lyrics, genre, metadata) and *acoustic* (energy, tone, rhythm) properties of a song.
- This is what makes **MuQ-powered hybrid recommendations** more accurate and context-aware.

✅ Once you see:

```
import json
import numpy as np
import os

def auto_convert_embeddings():

    # --- detect files in current directory ---
    base_dir = os.getcwd()
    all_files = os.listdir(base_dir)

    text_jsonls = [f for f in all_files if f.endswith("_song_embeddings.jsonl")]
    audio_jsonl = next((f for f in all_files if f == "Final_Audio_Embeddings.jsonl"), None)

    if not text_jsonls:
        print("⚠️ No text embedding JSONL files found.")
```

```
            return
        if not audio_jsonl:
            print("⚠️ No audio embedding JSONL file named 'audio_embedding.jsonl' found.")
            return

        # Create output directory for .npy files
        output_dir = os.path.join(base_dir, "converted_embeddings")
        os.makedirs(output_dir, exist_ok=True)

        # --- Load audio embeddings into a dict keyed by track_id ---
        print("Loading audio embeddings...")
        audio_data = {}
        with open(audio_jsonl, "r") as f:
            for line in f:
                data = json.loads(line)
                track_id = data.get("file_name", "").replace(".mp3", "")
                audio_data[track_id] = data.get("embedding", [])
        print(f"✅ Loaded {len(audio_data)} audio embeddings from {audio_jsonl}")

        # --- Process each text embedding model ---
        for text_jsonl in text_jsonls:
            model_name = text_jsonl.replace("_song_embeddings.jsonl", "")
            print(f"\nProcessing model: {model_name}")

            # Load text embeddings into a dict keyed by track_id
            text_data = {}
            with open(text_jsonl, "r") as f:
                for line in f:
                    data = json.loads(line)
                    track_id = str(data.get("track_id", ""))
                    text_data[track_id] = data

            print(f"Loaded {len(text_data)} text embeddings for {model_name}.")

            # --- Save text-only embeddings as .npy ---
            text_only_list = [
                {
                    "track_id": tid,
                    "embedding": entry.get("embedding", None),
                    "text": entry.get("text", "")
                }
                for tid, entry in text_data.items()
            ]
            text_only_path = os.path.join(output_dir, f"{model_name}_text_embeddings.npy")
            np.save(text_only_path, text_only_list)
            print(f"💾 Saved text-only embeddings → {text_only_path} ({len(text_only_list)} entries)")

            # --- Save matched text+audio embeddings ---
            matched = []
            for tid, entry in text_data.items():
                if tid in audio_data:
                    matched.append({
                        "track_id": tid,
                        "text_embedding": entry.get("embedding", None),
                        "audio_embedding": audio_data[tid],
                        "text": entry.get("text", "")
                    })
            matched_path = os.path.join(output_dir, f"{model_name}_text_audio_embeddings.npy")
            np.save(matched_path, matched)
            print(f"💾 Saved text+audio embeddings → {matched_path} ({len(matched)} matches)")

        # --- Save audio-only embeddings ---
        audio_only_list = [
            {"track_id": tid, "audio_embedding": emb}
            for tid, emb in audio_data.items()
        ]
        audio_npy_path = os.path.join(output_dir, "audio_embeddings.npy")
        np.save(audio_npy_path, audio_only_list)
        print(f"\n💾 Saved standalone audio embeddings → {audio_npy_path} ({len(audio_only_list)} entries)")

        print("\n✅ All conversions complete!")
        print(f"📂 Output directory: {output_dir}")
        print(f"Total files generated: {len(text_jsonls) * 2 + 1}")

    # --- Run automatically ---
    auto_convert_embeddings()
```

```
Loading audio embeddings...
✅ Loaded 1334 audio embeddings from Final_Audio_Embeddings.jsonl

Processing model: MPNet
Loaded 4500 text embeddings for MPNet.
```

```
💾 Saved text-only embeddings → /content/converted_embeddings/MPNet_text_embeddings.npy (4500 entries)
💾 Saved text+audio embeddings → /content/converted_embeddings/MPNet_text_audio_embeddings.npy (1334 matches)

Processing model: BGE
Loaded 4500 text embeddings for BGE.
💾 Saved text-only embeddings → /content/converted_embeddings/BGE_text_embeddings.npy (4500 entries)
💾 Saved text+audio embeddings → /content/converted_embeddings/BGE_text_audio_embeddings.npy (1334 matches)

Processing model: MiniLM
Loaded 4500 text embeddings for MiniLM.
💾 Saved text-only embeddings → /content/converted_embeddings/MiniLM_text_embeddings.npy (4500 entries)
💾 Saved text+audio embeddings → /content/converted_embeddings/MiniLM_text_audio_embeddings.npy (1334 matches)

💾 Saved standalone audio embeddings → /content/converted_embeddings/audio_embeddings.npy (1334 entries)

✅ All conversions complete!
📁 Output directory: /content/converted_embeddings
Total files generated: 7
```

🧩 Step 7 – Explain Recommendations with an AI Model

In this step, we're giving our recommender system a voice! 🎤

So far, our app can find songs that are similar based on embeddings. Now we'll add a short explanation for why a user who likes one song might also like another based on mood, energy, and style.

🧩 What We'll Do

- 🛰️ Connect to the OpenRouter API (to access an AI model).
- 💬 Send a short prompt describing both songs.
- 🧠 Receive a natural-language explanation from the model.

💡 Example: "If someone loves Blinding Lights, why might they also enjoy Save Your Tears?"

⚙️ How It Works

- Build a short descriptive prompt.
- Send it to OpenRouter's model.
- Return the explanation as text.

✅ Run the cell below to create the helper function: 👇

```
# Set the OpenRouter API key as an environment variable.
# This allows the client to authenticate requests securely without hardcoding the key in the code.
os.environ["OPENROUTER_API_KEY"] = "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"

# Initialize the OpenAI client with the OpenRouter base URL and the API key retrieved from environment variables
# This client will be used to send requests to the model for generating responses.
client = OpenAI(base_url="https://openrouter.ai/api/v1", api_key=os.getenv("OPENROUTER_API_KEY"))

# Define a function that explains why a user might like a recommended song based on a song they already like.
def explain_recommendation(user_song, recommended_song):
    # Create a prompt that asks the AI to provide a short explanation
    # focusing on aspects like mood, energy, or style of the recommended song.
    prompt = (
        f"User likes the song '{user_song}'. Suggest why they might like '{recommended_song}' "
        f"in 2-3 sentences focusing on mood, energy, or style."
    )

    # Send a request to the chat completion endpoint of the model.
    # The model used here is 'qwen/qwen3-coder:free', but it could be replaced with another model.
    # 'messages' is structured as a chat, with a user role sending the prompt.
    # 'max_tokens' sets the maximum length of the generated response.
    response = client.chat.completions.create(
        model="qwen/qwen3-coder:free",
        messages=[{"role": "user", "content": prompt}],
        max_tokens=999
    )

    # Extract the AI-generated explanation from the response object.
    explanation = response.choices[0].message.content

    # Return the explanation to whoever called the function.
    return explanation
```

💬 Step 11 – Building the AI Music Recommender

We've built embeddings, processed metadata, and merged text + audio features.
Now it's time for the fun part — turning everything into an **AI-powered chatbot** that understands songs 🎧

🧠 What happens in this step:

This section brings all components together into a *smart music recommender system* that can:

1. **Understand user intent**
   - Detects whether the user is asking for text-based, audio-based, or hybrid recommendations.
   - Recognizes phrases like:

     > "Recommend songs like *Blinding Lights*"
     > "Find tracks similar to this audio"
     > "Give me songs that sound energetic and chill"

2. **Load and process embeddings**
   - Uses **MuQ** to encode and compare audio embeddings (sound-based similarity).
   - Uses **BGE / MPNet** for text-based understanding of genre, mood, and metadata.
   - Matches both text + audio using **cosine similarity** to find top recommendations.

3. **Handle different input types:**
   - 📝 *Text mode*: Finds similar songs based on metadata and embedding meaning.
   - 🎵 *Audio mode*: Extracts MuQ embedding from the uploaded song or YouTube audio.
   - 🔀 *Hybrid mode*: Combines both for highly accurate, context-aware suggestions.

4. **Explain results using GPT (LLM integration)**
   - Generates natural, human-like explanations for why a song was recommended:

     > "This track matches your input by sharing a similar tempo (120 BPM) and energy level,
     > with bright synth patterns typical of 80s-inspired pop."

🪄 How the chatbot works:

Mode	What it uses	Example Query
Text-based	BGE embeddings + song metadata	"Songs like 'Levitating' by Dua Lipa"
Audio-based	MuQ embeddings	Upload a .wav or .mp3 file
Hybrid	Text + Audio embeddings combined	"Find songs that sound like 'Heat Waves' and feel energetic"

🧩 Key functions explained:

- `recommend_text_only()` → Uses text embeddings for semantic matching
- `recommend_audio_only()` → Uses MuQ for audio similarity
- `recommend_hybrid()` → Blends both (metadata + sound profile)
- `detect_recommendation_request()` → Interprets the user's natural query
- `chat_with_recommendations()` → The chatbot brain — manages conversation history, routes queries to the right recommender, and returns friendly, descriptive results

---

✅ **Outcome:** This step turns your notebook into a complete **AI-powered Music Recommender** that understands songs the way *humans do* — by meaning, sound, and emotion.

Try it out with:

```
!pip install yt_dlp
!pip install muq
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
from yt_dlp import YoutubeDL
import librosa
import tempfile
import os
import json
import torch


# Global variables for audio embeddings
audio_embedding_data = None
audio_embeddings_matrix = None
audio_track_ids = None
```

```
# Initialize MuQ model (load once globally)
try:
    from muq import MuQ
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    muq_model = MuQ.from_pretrained("OpenMuQ/MuQ-large-msd-iter")
    muq_model = muq_model.to(device).eval()
    MUQ_AVAILABLE = True
    print(f"✓ MuQ model loaded on {device}")
except Exception as e:
    MUQ_AVAILABLE = False
    print(f"⚠️ MuQ model not available: {e}")


def load_audio_embeddings():
    """
    Load audio embeddings from JSONL file and create mapping to dataset.
    """
    global audio_embedding_data, audio_embeddings_matrix, audio_track_ids

    if audio_embedding_data is not None:
        return audio_embeddings_matrix, audio_track_ids

    try:
        print("📂 Loading audio embeddings from audio_embeddings.jsonl...")

        embeddings_list = []
        track_ids = []

        with open('Final_Audio_Embeddings.jsonl', 'r', encoding='utf-8', errors='ignore') as f:
            for line in f:
                try:
                    data = json.loads(line.strip())
                    filename = data['file_name']
                    embedding = data['embedding']

                    # Extract track_id from filename
                    track_id = filename.replace('.mp3', '').replace('.wav', '').replace('.flac', '')

                    # Convert to numpy array
                    if isinstance(embedding, list):
                        embeddings_list.append(np.array(embedding, dtype=np.float32))
                    else:
                        embeddings_list.append(np.array(embedding, dtype=np.float32))

                    track_ids.append(track_id)
                except json.JSONDecodeError:
                    continue  # Skip invalid lines

        # Stack into matrix
        audio_embeddings_matrix = np.vstack(embeddings_list)
        audio_track_ids = track_ids

        print(f"✓ Loaded {len(track_ids)} audio embeddings with shape {audio_embeddings_matrix.shape}")

        # Verify mapping
        dataset_track_ids = set(df['track_id'].astype(str).values)
        audio_track_ids_set = set(track_ids)
        matched = len(audio_track_ids_set.intersection(dataset_track_ids))
        print(f"✓ Matched {matched}/{len(track_ids)} embeddings with dataset")

        return audio_embeddings_matrix, audio_track_ids

    except FileNotFoundError:
        print("❌ audio_embeddings.jsonl not found")
        return None, None
    except Exception as e:
        print(f"❌ Error loading audio embeddings: {e}")
        import traceback
        traceback.print_exc()
        return None, None


def download_youtube_audio(song_name):
    """Download audio from YouTube."""
    try:
        print(f"🔍 Searching YouTube for: {song_name}")

        ydl_opts = {
            'format': 'bestaudio/best',
            'postprocessors': [{
                'key': 'FFmpegExtractAudio',
                'preferredcodec': 'mp3',
```

```
                'preferredquality': '192',
            }],
            'outtmpl': os.path.join(tempfile.gettempdir(), '%(id)s.%(ext)s'),
            'quiet': True,
            'no_warnings': True,
            'default_search': 'ytsearch1',
        }

        with YoutubeDL(ydl_opts) as ydl:
            info = ydl.extract_info(f"ytsearch1:{song_name}", download=True)
            if info and 'entries' in info and len(info['entries']) > 0:
                video_id = info['entries'][0]['id']
                audio_path = os.path.join(tempfile.gettempdir(), f"{video_id}.mp3")
                print(f"✓ Downloaded: {info['entries'][0]['title']}")
                return audio_path
        return None
    except Exception as e:
        print(f"❌ YouTube download failed: {str(e)}")
        return None


def extract_muq_embedding(audio_path, sr=24000):
    """Extract MuQ embedding from audio file."""
    if not MUQ_AVAILABLE:
        print("❌ MuQ model not available")
        return None

    try:
        wav, _ = librosa.load(audio_path, sr=sr)
        wavs = torch.tensor(wav).unsqueeze(0).to(device)

        with torch.no_grad():
            output = muq_model(wavs, output_hidden_states=True)
            embedding = output.last_hidden_state.mean(dim=1).squeeze().cpu().numpy()

        print(f"✓ Extracted MuQ embedding (shape: {embedding.shape})")
        return embedding
    except Exception as e:
        print(f"❌ MuQ extraction failed: {str(e)}")
        return None




def recommend_text_only(user_song, selected_model_name='BGE', top_k=5):
    """
    Text-based recommendation using embeddings + audio features.
    """
    if selected_model_name not in embedding_models:
        raise ValueError(f"Model '{selected_model_name}' not loaded.")

    selected_model = embedding_models[selected_model_name]
    embeddings = model_embeddings[selected_model_name]

    # Find song in dataset
    user_row = df[df['track_name'].str.lower() == user_song.lower()]

    # Semantic similarity
    user_embedding = selected_model.encode([user_song], convert_to_numpy=True)
    semantic_sims = cosine_similarity(user_embedding, embeddings).squeeze()

    # CASE 1: Song NOT found
    if user_row.empty:
        print(f"ℹ️ Song '{user_song}' not found. Using semantic matching...")
        popularity_scores = df['popularity'].values / 100.0
        combined_score = semantic_sims * 0.95 + popularity_scores * 0.05
        top_idx = combined_score.argsort()[::-1][:top_k]

        results = []
        for idx in top_idx:
            row = df.iloc[idx]
            results.append({
                "track_name": row['track_name'],
                "artists": row['artists'],
                "explanation": f"Semantic: {semantic_sims[idx]:.3f} | Genre: {row['track_genre']} | Tempo: {row[
            })
        return results

    # CASE 2: Song FOUND
    user_row = user_row.iloc[0]
    user_idx = user_row.name
    print(f"✓ Found '{user_row['track_name']}' by {user_row['artists']}")
```

```
        # Audio feature similarity
        feature_sims = np.zeros(len(df))
        for i in range(len(df)):
            candidate = df.iloc[i]
            tempo_sim = max(0, 1 - abs(user_row['tempo'] - candidate['tempo']) / 100)
            energy_sim = 1 - abs(user_row['energy'] - candidate['energy'])
            dance_sim = 1 - abs(user_row['danceability'] - candidate['danceability'])
            valence_sim = 1 - abs(user_row['valence'] - candidate['valence'])
            loudness_sim = max(0, 1 - abs(user_row['loudness'] - candidate['loudness']) / 30)
            acoustic_sim = 1 - abs(user_row['acousticness'] - candidate['acousticness'])
            speech_sim = 1 - abs(user_row['speechiness'] - candidate['speechiness'])
            live_sim = 1 - abs(user_row['liveness'] - candidate['liveness'])

            feature_sims[i] = (
                tempo_sim * 0.25 + energy_sim * 0.20 + dance_sim * 0.20 +
                valence_sim * 0.15 + loudness_sim * 0.10 + acoustic_sim * 0.50 +
                speech_sim * 0.03 + live_sim * 0.02
            )

        genre_boost = np.where(df['track_genre'] == user_row['track_genre'], 1.1, 1.0)
        popularity_scores = df['popularity'].values / 100.0

        combined_score = (
            semantic_sims * 0.50 + feature_sims * 0.45 + popularity_scores * 0.05
        ) * genre_boost
        combined_score[user_idx] = -np.inf

        top_idx = combined_score.argsort()[::-1][:top_k]

        results = []
        for i in top_idx:
            row = df.iloc[i]
            tempo_match = "identical" if abs(user_row['tempo'] - row['tempo']) < 5 else \
                          "very close" if abs(user_row['tempo'] - row['tempo']) < 15 else "similar"
            energy_match = "identical" if abs(user_row['energy'] - row['energy']) < 0.05 else \
                           "very close" if abs(user_row['energy'] - row['energy']) < 0.15 else "similar"

            explanation = (
                f"Semantic: {semantic_sims[i]:.3f} | {tempo_match} tempo ({row['tempo']:.0f} vs {user_row['tempo']:.
                f"{energy_match} energy ({row['energy']:.2f} vs {user_row['energy']:.2f}) | Genre: {row['track_genre
            )

            results.append({
                "track_name": row['track_name'],
                "artists": row['artists'],
                "explanation": explanation
            })

        return results



    def recommend_audio_only(user_input, top_k=5, use_muq=True):
        """
        Audio-only recommendation.
        user_input can be: song name (string) OR audio file path (.wav, .mp3)
        """
        is_audio_file = isinstance(user_input, str) and os.path.isfile(user_input)

        if is_audio_file:
            print(f"🎵 Using audio file: {user_input}")

            # Extract MuQ embedding from file
            if not use_muq or not MUQ_AVAILABLE:
                raise ValueError("MuQ model required for audio file input")

            user_muq_embedding = extract_muq_embedding(user_input)
            if user_muq_embedding is None:
                raise ValueError("Failed to extract embedding from audio file")

            user_row = None
            user_idx = -1

        else:
            # Text input - try to find in dataset
            user_song = user_input
            user_row = df[df['track_name'].str.lower() == user_song.lower()]

            if not user_row.empty:
                user_row = user_row.iloc[0]
                user_idx = user_row.name
```

```
            print(f"✓ Found '{user_row['track_name']}' by {user_row['artists']}")
            user_muq_embedding = None
        else:
            # Download from YouTube
            print(f"ℹ️ Song '{user_song}' not in dataset. Downloading from YouTube...")
            audio_path = download_youtube_audio(user_song)

            if audio_path is None:
                raise ValueError(f"Could not download '{user_song}'")

            if use_muq and MUQ_AVAILABLE:
                print("🎵 Extracting MuQ embedding...")
                user_muq_embedding = extract_muq_embedding(audio_path)
            else:
                user_muq_embedding = None

            try:
                os.remove(audio_path)
            except:
                pass

            user_idx = -1

    # Calculate audio scores
    audio_scores = np.zeros(len(df))

    if user_muq_embedding is not None:
        # MuQ embedding matching
        dataset_embeddings, track_ids = load_audio_embeddings()

        if dataset_embeddings is not None:
            track_id_to_idx = {tid: idx for idx, tid in enumerate(track_ids)}
            track_id_to_dataset_idx = {str(row['track_id']): idx for idx, row in df.iterrows()}

            temp_scores = cosine_similarity(
                user_muq_embedding.reshape(1, -1),
                dataset_embeddings
            ).squeeze()

            matched_count = 0
            for track_id, emb_idx in track_id_to_idx.items():
                if track_id in track_id_to_dataset_idx:
                    dataset_idx = track_id_to_dataset_idx[track_id]
                    audio_scores[dataset_idx] = temp_scores[emb_idx]
                    matched_count += 1

            print(f"✓ Matched {matched_count} songs using MuQ embeddings")

    elif user_row is not None:
        # Feature-based matching
        for i in range(len(df)):
            candidate = df.iloc[i]
            tempo_sim = max(0, 1 - abs(user_row['tempo'] - candidate['tempo']) / 100)
            energy_sim = 1 - abs(user_row['energy'] - candidate['energy'])
            dance_sim = 1 - abs(user_row['danceability'] - candidate['danceability'])
            valence_sim = 1 - abs(user_row['valence'] - candidate['valence'])
            loudness_sim = max(0, 1 - abs(user_row['loudness'] - candidate['loudness']) / 30)
            acoustic_sim = 1 - abs(user_row['acousticness'] - candidate['acousticness'])
            instr_sim = 1 - abs(user_row['instrumentalness'] - candidate['instrumentalness'])

            audio_scores[i] = (
                tempo_sim * 0.30 + energy_sim * 0.25 + dance_sim * 0.20 +
                valence_sim * 0.12 + loudness_sim * 0.08 + acoustic_sim * 0.03 + instr_sim * 0.02
            )

        genre_boost = np.where(df['track_genre'] == user_row['track_genre'], 1.15, 1.0)
        audio_scores *= genre_boost
        audio_scores[user_idx] = -np.inf

    top_indices = audio_scores.argsort()[::-1][:top_k]

    results = []
    for idx in top_indices:
        song = df.iloc[idx]

        if user_row is not None and not user_row.empty:
            # Check if user_row is a Series (single row) or DataFrame
            if isinstance(user_row, pd.DataFrame):
                user_data = user_row.iloc[0]
            else:
                user_data = user_row
```

```
            tempo_diff = abs(user_data['tempo'] - song['tempo'])
            explanation = (
                f"Tempo: {song['tempo']:.0f} BPM (diff: {tempo_diff:.0f}) | "
                f"Energy: {song['energy']:.2f} | Dance: {song['danceability']:.2f} | "
                f"Genre: {song['track_genre']}"
            )
        else:
            explanation = (
                f"Audio match score: {audio_scores[idx]:.3f} | "
                f"Tempo: {song['tempo']:.0f} BPM | Energy: {song['energy']:.2f} | "
                f"Genre: {song['track_genre']}"
            )

        results.append({
            "track_name": song['track_name'],
            "artists": song['artists'],
            "explanation": explanation
        })

    return results




def recommend_hybrid(user_song, selected_model_name='BGE', top_k=5):
    """
    Hybrid: text embeddings + audio features.
    """
    if selected_model_name not in embedding_models:
        raise ValueError(f"Model '{selected_model_name}' not loaded.")

    selected_model = embedding_models[selected_model_name]
    embeddings = model_embeddings[selected_model_name]

    user_embedding = selected_model.encode([user_song], convert_to_numpy=True)
    semantic_sims = cosine_similarity(user_embedding, embeddings).squeeze()

    user_row = df[df['track_name'].str.lower() == user_song.lower()]

    if user_row.empty:
        print(f"ℹ️ Song not found. Using semantic matching...")
        popularity_scores = df['popularity'].values / 100.0
        combined_score = semantic_sims * 0.95 + popularity_scores * 0.05
    else:
        user_row = user_row.iloc[0]
        user_idx = user_row.name
        print(f"✓ Found '{user_row['track_name']}' - using hybrid approach")

        feature_sims = np.zeros(len(df))
        for i in range(len(df)):
            candidate = df.iloc[i]
            tempo_sim = max(0, 1 - abs(user_row['tempo'] - candidate['tempo']) / 100)
            energy_sim = 1 - abs(user_row['energy'] - candidate['energy'])
            dance_sim = 1 - abs(user_row['danceability'] - candidate['danceability'])
            valence_sim = 1 - abs(user_row['valence'] - candidate['valence'])
            loudness_sim = max(0, 1 - abs(user_row['loudness'] - candidate['loudness']) / 30)
            acoustic_sim = 1 - abs(user_row['acousticness'] - candidate['acousticness'])
            speech_sim = 1 - abs(user_row['speechiness'] - candidate['speechiness'])
            live_sim = 1 - abs(user_row['liveness'] - candidate['liveness'])

            feature_sims[i] = (
                tempo_sim * 0.25 + energy_sim * 0.20 + dance_sim * 0.20 +
                valence_sim * 0.15 + loudness_sim * 0.10 + acoustic_sim * 0.05 +
                speech_sim * 0.03 + live_sim * 0.02
            )

        genre_boost = np.where(df['track_genre'] == user_row['track_genre'], 1.12, 1.0)
        popularity_scores = df['popularity'].values / 100.0
        interaction_bonus = feature_sims * semantic_sims * 0.05

        combined_score = (
            semantic_sims * 0.40 + feature_sims * 0.50 +
            popularity_scores * 0.05 + interaction_bonus
        ) * genre_boost
        combined_score[user_idx] = -np.inf

    top_idx = combined_score.argsort()[::-1][:top_k]

    results = []
    for i in top_idx:
        row = df.iloc[i]
```

```
            if not user_row.empty:
                explanation = (
                    f"Hybrid | Semantic: {semantic_sims[i]:.3f} | "
                    f"Tempo: {row['tempo']:.0f} vs {user_row['tempo']:.0f} BPM | "
                    f"Energy: {row['energy']:.2f} vs {user_row['energy']:.2f} | "
                    f"Genre: {row['track_genre']}"
                )
            else:
                explanation = (
                    f"Semantic: {semantic_sims[i]:.3f} | "
                    f"Tempo: {row['tempo']:.0f} BPM | Energy: {row['energy']:.2f} | "
                    f"Genre: {row['track_genre']}"
                )

            results.append({
                "track_name": row['track_name'],
                "artists": row['artists'],
                "explanation": explanation
            })

        return results


    def recommend_text_only_llm(user_song, selected_model_name='BGE', top_k=5, llm_client=None):
        """Text-based with LLM explanations."""
        base_recs = recommend_text_only(user_song, selected_model_name, top_k)

        user_row = df[df['track_name'].str.lower() == user_song.lower()]

        results = []
        for rec in base_recs:
            song_row = df[(df['track_name'] == rec['track_name']) & (df['artists'] == rec['artists'])].iloc[0]

            if llm_client and not user_row.empty:
                user_data = user_row.iloc[0]
                prompt = (
                    f"Explain why '{song_row['track_name']}' by {song_row['artists']} is recommended "
                    f"for someone who likes '{user_song}' by {user_data['artists']}. "
                    f"Original: {user_data['track_genre']}, {user_data['tempo']:.0f} BPM, energy {user_data['energy'
                    f"Recommended: {song_row['track_genre']}, {song_row['tempo']:.0f} BPM, energy {song_row['energy'
                    f"Focus on musical similarities in 2-3 sentences."
                )

                response = llm_client.chat.completions.create(
                    model="qwen/qwen-2.5-coder-32b-instruct",
                    messages=[{"role": "user", "content": prompt}],
                    max_tokens=150
                )
                explanation = response.choices[0].message.content.strip()
            else:
                explanation = rec['explanation']

            results.append({
                "track_name": rec['track_name'],
                "artists": rec['artists'],
                "explanation": explanation
            })

        return results




    def recommend_audio_only_llm(user_input, top_k=5, use_muq=True, llm_client=None):
        """Audio-based with LLM explanations."""
        base_recs = recommend_audio_only(user_input, top_k, use_muq)

        is_audio_file = isinstance(user_input, str) and os.path.isfile(user_input)
        if not is_audio_file:
            user_row = df[df['track_name'].str.lower() == user_input.lower()]
        else:
            user_row = None

        results = []
        for rec in base_recs:
            song_row = df[(df['track_name'] == rec['track_name']) & (df['artists'] == rec['artists'])].iloc[0]

            if llm_client and user_row is not None and not user_row.empty:
                user_data = user_row.iloc[0]
                prompt = (
                    f"Explain why '{song_row['track_name']}' matches '{user_input}' based on AUDIO characteristics.
                    f"Original: {user_data['tempo']:.0f} BPM, energy {user_data['energy']:.2f}, dance {user_data['da
```

```
                    f"Recommended: {song_row['tempo']:.0f} BPM, energy {song_row['energy']:.2f}, dance {song_row['da
                    f"Focus on rhythm and sonic qualities in 2-3 sentences."
                )

                response = llm_client.chat.completions.create(
                    model="qwen/qwen-2.5-coder-32b-instruct",
                    messages=[{"role": "user", "content": prompt}],
                    max_tokens=150
                )
                explanation = response.choices[0].message.content.strip()
            else:
                explanation = rec['explanation']

            results.append({
                "track_name": rec['track_name'],
                "artists": rec['artists'],
                "explanation": explanation
            })

        return results



    def recommend_hybrid_llm(user_song, selected_model_name='BGE', top_k=5, llm_client=None):
        """Hybrid with LLM explanations."""
        base_recs = recommend_hybrid(user_song, selected_model_name, top_k)

        user_row = df[df['track_name'].str.lower() == user_song.lower()]

        results = []
        for rec in base_recs:
            song_row = df[(df['track_name'] == rec['track_name']) & (df['artists'] == rec['artists'])].iloc[0]

            if llm_client and not user_row.empty:
                user_data = user_row.iloc[0]
                prompt = (
                    f"Provide a compelling explanation for why '{song_row['track_name']}' by {song_row['artists']} "
                    f"is recommended for someone who enjoys '{user_song}' by {user_data['artists']}. "
                    f"Original: {user_data['track_genre']}, {user_data['tempo']:.0f} BPM, energy {user_data['energy'
                    f"Recommended: {song_row['track_genre']}, {song_row['tempo']:.0f} BPM, energy {song_row['energy'
                    f"Combine thematic and sonic connections in 3-4 engaging sentences."
                )

                response = llm_client.chat.completions.create(
                    model="qwen/qwen-2.5-coder-32b-instruct",
                    messages=[{"role": "user", "content": prompt}],
                    max_tokens=200
                )
                explanation = response.choices[0].message.content.strip()
            else:
                explanation = rec['explanation']

            results.append({
                "track_name": rec['track_name'],
                "artists": rec['artists'],
                "explanation": explanation
            })

        return results




    def detect_recommendation_request(message):
        """
        Detect if user is asking for song recommendations.
        Improved: Better pattern matching, handles more variations, extracts song name more robustly.
        Returns: (is_request, song_name, recommendation_type)
        """
        message_lower = message.lower().strip()

        # Expanded patterns for recommendation requests
        patterns = {
            'recommend': ['recommend', 'suggest', 'suggestions for', 'find me', 'give me', 'what are some'],
            'similar': ['similar to', 'like', 'sounds like', 'reminds me of', 'in the style of'],
            'based_on': ['based on', 'if i like', 'from', 'inspired by'],
            'like': ['i like', 'i love', 'i enjoy', 'my favorite is', 'i\'m into']
        }

        # Song contexts to confirm it's music-related
        song_contexts = ['song', 'music', 'track', 'artist', 'band', 'album', 'playlist', 'tune', 'hit']
        has_song_context = any(context in message_lower for context in song_contexts) or any(pattern in message_lowe
```

```
        # Extract song name
        song_name = None
        is_request = False

        # Check patterns in order of specificity
        for category, phrases in patterns.items():
            for phrase in phrases:
                if phrase in message_lower:
                    is_request = True
                    parts = message_lower.split(phrase, 1)
                    if len(parts) > 1:
                        potential_song = parts[1].strip()
                        # Refine extraction: look for quotes, or split by common separators
                        if '"' in potential_song:
                            song_name = potential_song.split('"')[1]
                        elif "'" in potential_song:
                            song_name = potential_song.split("'")[1]
                        elif 'by' in potential_song:
                            song_name = potential_song.split('by')[0].strip()
                        else:
                            song_name = potential_song
                    break
            if is_request:
                break

        # Fallback: if no specific phrase but has "like [song]" or similar
        if not song_name and 'like' in message_lower and has_song_context:
            song_name = message_lower.split('like', 1)[-1].strip()
            is_request = True

        # Clean up song name
        if song_name:
            remove_words = ['songs', 'music', 'tracks', 'by', 'the', 'a', 'an', 'some', 'me', 'please', '?', '!', '.
            for word in remove_words:
                song_name = song_name.replace(word, '').strip()
            # Capitalize properly if needed
            song_name = song_name.title()
            if len(song_name) < 3:
                is_request = False
                song_name = None

        # Determine recommendation type (improved: more keywords)
        rec_type = 'hybrid'  # default
        if any(word in message_lower for word in ['audio', 'sound', 'rhythm', 'beat', 'sonic', 'instrumental']):
            rec_type = 'audio_only'
        elif any(word in message_lower for word in ['text', 'lyrics', 'name', 'semantic', 'theme', 'story']):
            rec_type = 'text_only'

        return is_request, song_name, rec_type

    def chat_with_recommendations(message, history=None, selected_model_name='BGE'):

        if history is None:
            history = []

        # Format history from Gradio format [[user, bot], ...] to list of dicts
        formatted_history = []
        for pair in history:
            if isinstance(pair, list) and len(pair) == 2:
                user_msg, bot_msg = pair
                if user_msg:
                    formatted_history.append({"role": "user", "content": user_msg})
                if bot_msg:
                    formatted_history.append({"role": "assistant", "content": bot_msg})
            else:
                # If not standard Gradio, assume it's already dicts or skip
                pass

        is_rec_request, song_name, rec_type = detect_recommendation_request(message)

        if not is_rec_request or not song_name:
            # Normal chat (non-recommendation)
            if 'client' not in globals():
                return "Sorry, I can't respond right now without an LLM."
            system_prompt = "You are a helpful music chatbot. Respond naturally."
            messages = [{"role": "system", "content": system_prompt}] + formatted_history + [{"role": "user", "conte
            # print(messages)  # Debug: Uncomment to check structure
            try:
                response = client.chat.completions.create(
                    model="qwen/qwen3-coder:free",
                    messages=messages,
```

```
                max_tokens=300,
                temperature=0.7
            )
            return response.choices[0].message.content.strip()
        except Exception as e:
            return f"Error in API call: {str(e)}"

    # Recommendation request
    try:
        # Call the appropriate recommendation function
        if rec_type == 'text_only':
            recs = recommend_text_only(song_name, selected_model_name, top_k=5)
        elif rec_type == 'audio_only':
            recs = recommend_audio_only(song_name, top_k=5)
        else:
            recs = recommend_hybrid(song_name, selected_model_name, top_k=5)

        # Format results as context
        results_context = "Based on the user's query for recommendations similar to '{}', here are the top resul
        for i, r in enumerate(recs, 1):
            results_context += "{}. '{}' by {} - {}\n".format(
                i, r['track_name'], r['artists'], r.get('explanation', 'No explanation available')
            )

        # If no LLM, return formatted text
        if 'client' not in globals():
            return results_context

        # Send to LLM for natural response
        system_prompt = (
            "You are a friendly music recommendation assistant. "
            "Use the provided results to generate an engaging, conversational response. "
            "Start with something like 'Based on your interest in [song], here are some recommendations:' "
            "Explain briefly why each might appeal, and keep it fun and concise."
        )
        user_prompt = "{}\n{}".format(message, results_context)  # Include original message and results
        messages = [{"role": "system", "content": system_prompt}] + formatted_history + [{"role": "user", "conte
        # print(messages)  # Debug: Uncomment to check structure
        try:
            response = client.chat.completions.create(
                model="qwen/qwen3-coder:free",
                messages=messages,
                max_tokens=500,
                temperature=0.8
            )
            return response.choices[0].message.content.strip()
        except Exception as e:
            return f"Error in API call: {str(e)}"

    except Exception as e:
        return "Sorry, there was an error generating recommendations: {}".format(str(e))
```

```
Collecting yt_dlp
  Downloading yt_dlp-2025.10.22-py3-none-any.whl.metadata (176 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 176.0/176.0 kB 7.0 MB/s eta 0:00:00
Downloading yt_dlp-2025.10.22-py3-none-any.whl (3.2 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.2/3.2 MB 76.4 MB/s eta 0:00:00
Installing collected packages: yt_dlp
Successfully installed yt_dlp-2025.10.22
Collecting muq
  Downloading muq-0.1.0.tar.gz (55 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 55.8/55.8 kB 3.0 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: einops in /usr/local/lib/python3.12/dist-packages (from muq) (0.8.1)
Requirement already satisfied: librosa in /usr/local/lib/python3.12/dist-packages (from muq) (0.11.0)
Collecting nnAudio (from muq)
  Downloading nnAudio-0.3.3-py3-none-any.whl.metadata (771 bytes)
Requirement already satisfied: numpy in /usr/local/lib/python3.12/dist-packages (from muq) (2.0.2)
Requirement already satisfied: soundfile in /usr/local/lib/python3.12/dist-packages (from muq) (0.13.1)
Requirement already satisfied: torch in /usr/local/lib/python3.12/dist-packages (from muq) (2.8.0+cu126)
Requirement already satisfied: torchaudio in /usr/local/lib/python3.12/dist-packages (from muq) (2.8.0+cu126)
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-packages (from muq) (4.67.1)
Requirement already satisfied: transformers in /usr/local/lib/python3.12/dist-packages (from muq) (4.57.1)
Requirement already satisfied: easydict in /usr/local/lib/python3.12/dist-packages (from muq) (1.13)
Collecting x_clip (from muq)
  Downloading x_clip-0.14.4-py3-none-any.whl.metadata (724 bytes)
Requirement already satisfied: audioread>=2.1.9 in /usr/local/lib/python3.12/dist-packages (from librosa->muq) (3
Requirement already satisfied: numba>=0.51.0 in /usr/local/lib/python3.12/dist-packages (from librosa->muq) (0.60
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.12/dist-packages (from librosa->muq) (1.16.
Requirement already satisfied: scikit-learn>=1.1.0 in /usr/local/lib/python3.12/dist-packages (from librosa->muq)
Requirement already satisfied: joblib>=1.0 in /usr/local/lib/python3.12/dist-packages (from librosa->muq) (1.5.2)
Requirement already satisfied: decorator>=4.3.0 in /usr/local/lib/python3.12/dist-packages (from librosa->muq) (4
Requirement already satisfied: pooch>=1.1 in /usr/local/lib/python3.12/dist-packages (from librosa->muq) (1.8.2)
Requirement already satisfied: soxr>=0.3.2 in /usr/local/lib/python3.12/dist-packages (from librosa->muq) (1.0.0)
Requirement already satisfied: typing_extensions>=4.1.1 in /usr/local/lib/python3.12/dist-packages (from librosa-
Requirement already satisfied: lazy_loader>=0.1 in /usr/local/lib/python3.12/dist-packages (from librosa->muq) (0
Requirement already satisfied: msgpack>=1.0 in /usr/local/lib/python3.12/dist-packages (from librosa->muq) (1.1.2
Requirement already satisfied: cffi>=1.0 in /usr/local/lib/python3.12/dist-packages (from soundfile->muq) (2.0.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.12/dist-packages (from torch->muq) (3.20.0)
Requirement already satisfied: setuptools in /usr/local/lib/python3.12/dist-packages (from torch->muq) (75.2.0)
Requirement already satisfied: sympy>=1.13.3 in /usr/local/lib/python3.12/dist-packages (from torch->muq) (1.13.3
Requirement already satisfied: networkx in /usr/local/lib/python3.12/dist-packages (from torch->muq) (3.5)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.12/dist-packages (from torch->muq) (3.1.6)
Requirement already satisfied: fsspec in /usr/local/lib/python3.12/dist-packages (from torch->muq) (2025.3.0)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.6.77 in /usr/local/lib/python3.12/dist-packages (from t
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.6.77 in /usr/local/lib/python3.12/dist-packages (from
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.6.80 in /usr/local/lib/python3.12/dist-packages (from t
Requirement already satisfied: nvidia-cudnn-cu12==9.10.2.21 in /usr/local/lib/python3.12/dist-packages (from torc
Requirement already satisfied: nvidia-cublas-cu12==12.6.4.1 in /usr/local/lib/python3.12/dist-packages (from torc
Requirement already satisfied: nvidia-cufft-cu12==11.3.0.4 in /usr/local/lib/python3.12/dist-packages (from torch
Requirement already satisfied: nvidia-curand-cu12==10.3.7.77 in /usr/local/lib/python3.12/dist-packages (from tor
Requirement already satisfied: nvidia-cusolver-cu12==11.7.1.2 in /usr/local/lib/python3.12/dist-packages (from to
Requirement already satisfied: nvidia-cusparse-cu12==12.5.4.2 in /usr/local/lib/python3.12/dist-packages (from to
Requirement already satisfied: nvidia-cusparselt-cu12==0.7.1 in /usr/local/lib/python3.12/dist-packages (from tor
Requirement already satisfied: nvidia-nccl-cu12==2.27.3 in /usr/local/lib/python3.12/dist-packages (from torch->m
Requirement already satisfied: nvidia-nvtx-cu12==12.6.77 in /usr/local/lib/python3.12/dist-packages (from torch->
Requirement already satisfied: nvidia-nvjitlink-cu12==12.6.85 in /usr/local/lib/python3.12/dist-packages (from to
Requirement already satisfied: nvidia-cufile-cu12==1.11.1.6 in /usr/local/lib/python3.12/dist-packages (from torc
Requirement already satisfied: triton==3.4.0 in /usr/local/lib/python3.12/dist-packages (from torch->muq) (3.4.0)
Requirement already satisfied: huggingface-hub<1.0,>=0.34.0 in /usr/local/lib/python3.12/dist-packages (from tran
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.12/dist-packages (from transformers->muq
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.12/dist-packages (from transformers->muq) (6
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.12/dist-packages (from transformers->m
Requirement already satisfied: requests in /usr/local/lib/python3.12/dist-packages (from transformers->muq) (2.32
Requirement already satisfied: tokenizers<=0.23.0,>=0.22.0 in /usr/local/lib/python3.12/dist-packages (from trans
Requirement already satisfied: safetensors>=0.4.3 in /usr/local/lib/python3.12/dist-packages (from transformers->
Requirement already satisfied: beartype in /usr/local/lib/python3.12/dist-packages (from x_clip->muq) (0.22.4)
Collecting ftfy (from x_clip->muq)
  Downloading ftfy-6.3.1-py3-none-any.whl.metadata (7.3 kB)
Requirement already satisfied: torchvision in /usr/local/lib/python3.12/dist-packages (from x_clip->muq) (0.23.0+
Requirement already satisfied: pycparser in /usr/local/lib/python3.12/dist-packages (from cffi>=1.0->soundfile->m
Requirement already satisfied: hf-xet<2.0.0,>=1.1.3 in /usr/local/lib/python3.12/dist-packages (from huggingface-
Requirement already satisfied: llvmlite<0.44,>=0.43.0dev0 in /usr/local/lib/python3.12/dist-packages (from numba>
Requirement already satisfied: platformdirs>=2.5.0 in /usr/local/lib/python3.12/dist-packages (from pooch>=1.1->l
Requirement already satisfied: charset_normalizer<4,>=2 in /usr/local/lib/python3.12/dist-packages (from requests
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.12/dist-packages (from requests->transforme
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.12/dist-packages (from requests->tran
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.12/dist-packages (from requests->tran
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.12/dist-packages (from scikit-learn
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.12/dist-packages (from sympy>=1.13.3-
Requirement already satisfied: wcwidth in /usr/local/lib/python3.12/dist-packages (from ftfy->x_clip->muq) (0.2.1
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.12/dist-packages (from jinja2->torch->mu
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in /usr/local/lib/python3.12/dist-packages (from torchvision
Downloading nnAudio-0.3.3-py3-none-any.whl (43 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 43.6/43.6 kB 4.0 MB/s eta 0:00:00
Downloading x_clip-0.14.4-py3-none-any.whl (1.4 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.4/1.4 MB 35.1 MB/s eta 0:00:00
Downloading ftfy-6.3.1-py3-none-any.whl (44 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 44.8/44.8 kB 4.3 MB/s eta 0:00:00
Building wheels for collected packages: muq
  Building wheel for muq (setup.py) ... done
  Created wheel for muq: filename=muq-0.1.0-py3-none-any.whl size=60109 sha256=6b0ac4cd1db2e705907b596ad5451d8eaa
```

💬 Step 9 — The Chat interface (Connecting Everything)

Now that our recommendation logic and LLM explanations are ready,
we need a way for users to *chat* with our music recommender — this is where the **`chat()`** function comes in.

🧠 **What it does:**

- Acts as the *main entry point* for the chat interface (like Gradio).
- Passes the user's message and chat history to our function `chat_with_recommendations()`.
- Returns the response generated by the LLM or the recommender system.

⚙️ **How it fits in:**

1. The user types something like:
   - "Recommend me songs like *Blinding Lights*."
2. Gradio sends this message into `chat()`.
3. The function calls `chat_with_recommendations()` — which decides:
   - whether to use **MuQ embeddings**,
   - generate **recommendations**,
   - and return **a natural-language explanation**.
4. The result (recommendations + explanation) is displayed back in the chat window.

```
  Stored in directory: /root/.cache/pip/wheels/8c/83/a4/70dcef32ac0d79599e73d354880f0b28f61e2cce24c4c1aba5
Successfully built muq
Installing collected packages: ftfy, fairseq, x_clip, muq
Successfully installed ftfy-6.3.1 muq-0.1.0 npAudio-0.3.3 x_clip-0.14.4
config.json:       3.13k/? [00:00<00:00, 125kB/s]
```

💡 **In short:**

This is the **bridge** between our backend logic and the chat UI — it makes our AI feel like an interactive assistant instead of just a static script.

```
def chat(message, history=None, selected_model_name='BGE'):
    """
    Improved main chat handler:
    - Uses chat_with_recommendations for all logic.
    - Simplifies: directly calls it and returns the response.
    - Assumes Gradio or similar will handle history appending.
    """
    response = chat_with_recommendations(message, history, selected_model_name)
    return response
```

🎨 Step 10 – Launching the Gradio Chat Interface

Now that our chat function is ready, let's connect it to a simple **Gradio** interface.

🪄 **What this does:**

- Displays a chat box in your browser.
- Sends your message (e.g. "Recommend me songs like Blinding Lights") to the `chat()` function.
- Displays responses directly in a clean, interactive UI.

🌍 **Local vs. Share Links:**

- When you run `launch(share=True)`, Gradio creates a *temporary public link*.
- You (and others) can open this link to test the chatbot in real-time.
- If you use `share=False`, it runs *locally* on your device (good for debugging or private demos).

💡 **Tip:**

You can change the `title` or `description` to personalize your chatbot.

```
chat_interface = gr.ChatInterface(
    fn=chat,
    title="🎵 Music Recommendation Chatbot",
    description="Ask for song recommendations or chat with the bot."
)

chat_interface.launch(share=True, debug=True)
```

```
/usr/local/lib/python3.12/dist-packages/gradio/chat_interface.py:347: UserWarning:

The 'tuples' format for chatbot messages is deprecated and will be removed in a future version of Gradio. Please

Colab notebook detected. This cell will run indefinitely so that you can see errors and logs. To turn off, set de
* Running on public URL: https://204ffecaf17e4fe4e2.gradio.live

This share link expires in 1 week. For free permanent hosting and GPU upgrades, run `gradio deploy` from the term
```

No interface is running right now

```
ℹ️ Song not found. Using semantic matching...
Keyboard interruption in main thread... closing server.
Killing tunnel 127.0.0.1:7860 <> https://204ffecaf17e4fe4e2.gradio.live
```

🎧 Step 11 – Quick Local Test (Without Chat)

Before testing the chatbot, you can run this simple code block
to verify that your **recommendation engine** works on its own.

🔍 **What it does:**

- Uses `recommend_audio_only()` to find the top 5 similar songs for the given input.
- Prints the results directly in the notebook (no chat interface).