1. **Introduction**

Music has always played a pivotal role in human culture and identity. In the digital era, **music recommendation systems (MRS)** have emerged as indispensable tools that personalize listening experiences for millions of users. These systems serve as intermediaries between massive music libraries and individual listeners, offering tailored suggestions that match the user’s tastes, needs, and contexts. Music Recommendation Systems (MRS) have become essential tools in modern digital music platforms, enabling users to discover new content tailored to their preferences. From generating playlists based on historical listening patterns to suggesting trending songs through social signals, MRS have become critical to user engagement in platforms like Spotify, Apple Music, Deezer, and YouTube Music.

Yet, despite their ubiquity and ongoing advances, **existing MRS still fall short of capturing the full complexity of human musical preference**. The way people choose music is shaped by a confluence of factors that go beyond mere history or sound features. These include psychological traits (such as personality or mood), situational variables (e.g., activity, location, or time of day), and cultural influences (e.g., language, ethnicity, or regional music traditions). This thesis explores how these **underrepresented dimensions** can be integrated into a recommendation engine to produce richer, more human-aware music suggestions.

Traditional recommendation techniques often rely on collaborative filtering, content-based filtering, or hybrid methods. While these approaches have proven effective to some extent, they often fail to consider the complex, multidimensional nature of music preferences, which can be influenced by psychological states, situational contexts, and cultural backgrounds. As a result, many systems struggle with problems like the cold-start issue, lack of diversity in recommendations, and failure to adapt to users’ changing moods or environments.

This thesis aims to address these limitations by advancing MRS through the integration of psychological, situational, and cultural dimensions. We propose a user-centric, text-based recommendation approach that allows users to input free-text queries, such as "energetic workout music" or "calm jazz for reading," and receive dynamically generated suggestions that align with the intended mood or activity.

The main objective of this research is to design and implement a modular web-based system that leverages modern web technologies (Vue 3, Node.js, MongoDB) for user interaction and service orchestration, and Python for recommendation logic. By combining natural language processing (NLP) with music metadata analysis, the system seeks to interpret user intent and offer relevant recommendations that go beyond surface-level similarity.

The remainder of the thesis is structured as follows: Chapter 2 presents a literature review of existing approaches and current challenges in music recommendation. Chapter 3 details the technical stack and system architecture. Chapter 4 describes the methodology and implementation of the recommendation engine. Chapter 5 provides an analysis of experimental results, and Chapter 6 concludes with key findings and directions for future work.

### 1.1 Historical Context and Evolution of MRS

The journey of music recommendation has evolved considerably over time. Initially, music discovery relied on **manual curation**—radio DJs, magazine lists, or peer suggestions. As digital media gained ground, static algorithms offered **genre-based filtering**, which grouped users based on similar genre preferences. Then, **collaborative filtering (CF)** and **content-based filtering (CBF)** rose to prominence in the early 2000s. CF uses similarity among users or items (e.g., “people who liked this also liked…”), while CBF analyzes item features (e.g., genre, tempo, key).

However, CF suffers from the **cold start problem**—new users or items lack sufficient data—and often reinforces **popularity bias** and filter bubbles. Hybrid systems that combine CF and CBF address some of these issues but still remain **weak in contextual awareness**. In recent years, **deep learning-based models**, including sequence-aware systems and transformers (e.g., BERT4Rec), have been introduced to model user sessions and long-term preferences. Platforms like Spotify have also implemented **real-time playlist generation and mood-based analysis**, yet these features often remain proprietary and opaque.

The current academic discourse has shifted toward **context-aware** and **explainable** recommendations. This includes user intent modeling, **emotion detection from queries**, and **cross-domain recommendation** (e.g., combining music with mental health or fitness apps). Still, most real-world systems fail to fully integrate **psychological, situational, and cultural factors** in a cohesive way.

### 1.2 Interdisciplinary Relevance and User-Centric Motivation

Understanding music preference demands an interdisciplinary lens. **Psychology** explains how traits like extraversion or neuroticism shape genre choices. For example, studies show that extraverts tend to prefer energetic music, while introverts might lean toward instrumental or mellow tracks. **Situational context** is equally critical—people often seek different music when they are studying, driving, exercising, or socializing. A track that energizes a morning run may feel intrusive during a meditation session. Meanwhile, **cultural background** plays a role in musical syntax understanding, language comprehension in lyrics, and rhythm preferences. Western users may prefer harmonic progression-based music, while Middle Eastern cultures might favor maqam-based melodic scales.

In practice, a music recommendation system that understands that a user is “preparing for an exam and feeling anxious” could provide calming instrumental music in a preferred language, filtered by culturally relevant instruments. This **nuanced personalization** is missing from most MRS and forms the motivation for this thesis.

### 1.3 Objectives and Contributions

To address these limitations, the thesis sets out to develop and evaluate a **context-aware, user-centric music recommendation system** that integrates the following components:

1. **Natural Language Query Interface** – Accepts free-form text input from the user (e.g., “relaxing Turkish music for a rainy day”) and infers intent.
2. **Psychological Modeling** – Interprets mood, emotional tone, or personality markers within the query.
3. **Situational Awareness** – Infers activity, environment, or time-based cues (e.g., “workout,” “bedtime,” “study”) to enhance contextual relevance.
4. **Cultural Tagging and Filtering** – Incorporates metadata like song language, region, or cultural genre.
5. **Modular Architecture** – Implements the above logic using a decoupled architecture (Vue 3 frontend, FastAPI backend, Python NLP recommender).
6. **Evaluation** – Uses both quantitative metrics (e.g., cosine similarity accuracy, diversity) and qualitative case studies to evaluate relevance.

The novelty of this work lies in combining **natural language understanding with cultural and emotional sensitivity**, building a recommender system that responds not only to what the user wants, but why they want it.

## CHAPTER 3 Technical Stack and Architecture

To support the implementation of the proposed music recommendation system, we have designed a modern web-based architecture leveraging widely used and efficient technologies for both frontend and backend development, along with machine learning capabilities.

### 1. Frontend

The user interface will be developed using the following technologies:

* **HTML & CSS**: These technologies will be used to structure and style the layout of the application.
* **JavaScript**: Enables dynamic content rendering and interactivity.
* **Vue 3**: A progressive JavaScript framework will be used for building the frontend, specifically using the Composition API with the <script setup> syntax for clearer and more concise logic organization.

This frontend stack provides a responsive, modular, and maintainable design, which is ideal for handling user inputs, such as text queries, and rendering personalized music recommendations.

Vue 3 was chosen over alternatives like React or Angular due to its lightweight structure, reactivity system, and smooth integration of the Composition API, which promotes cleaner code organization for medium-sized applications such as ours.

### 2. Backend

The backend will serve as the communication layer between the frontend and the recommendation engine. It will be built using:

* **Node.js**: A JavaScript runtime environment suitable for handling asynchronous operations and scalable applications.
* **Express.js**: A minimal and flexible Node.js web application framework used to define routes and API endpoints.
* **MongoDB**: A NoSQL database chosen for its flexibility in handling unstructured data and ease of integration with JavaScript-based applications. It will be used to store music metadata, user queries, and recommendation logs if needed.

We chose MongoDB instead of relational databases like PostgreSQL due to its schema-less design, which is ideal for evolving datasets typical in recommendation systems. Its native compatibility with JSON and JavaScript simplifies integration with Node.js.

### 3. Recommendation Engine

The recommendation engine is the core analytical component of the system. It will be developed using:

* **Python**: A powerful programming language widely used in data science and machine learning.
* **Natural Language Processing Libraries**: Libraries such as transformers, sentence-transformers, or spaCy will be used to process and vectorize text queries.
* **Similarity Analysis**: Cosine similarity or other distance metrics will be applied to compare user input against a dataset of song metadata or precomputed embeddings to generate relevant music recommendations.

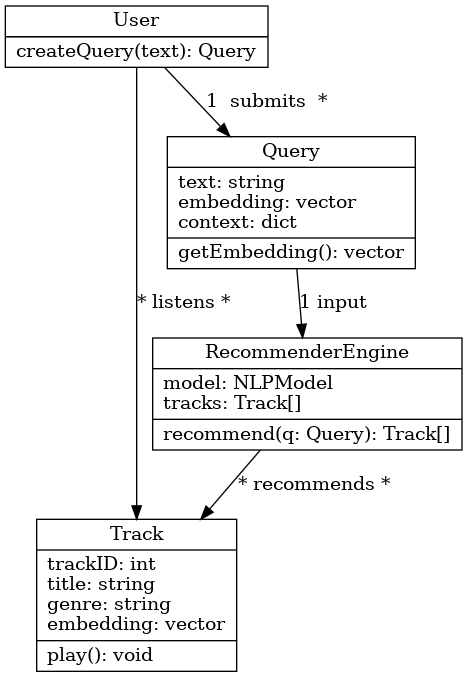


Figure 1

**Figure 1: Class diagram** of the core system components and their associations. In this UML class diagram, **User** represents the end-user of the system (in a broader implementation, this could include attributes like userID, name, or preference settings). A User can **submit** many Query instances (indicated by the “1 submits *” notation next to the association), meaning over time a user might issue multiple search queries. Each* ***Query*** *encapsulates a user’s input text (e.g. "energetic workout music") and possibly an embedding or context that is derived from that text. In our implementation, the Query text is transformed into an embedding vector via the NLP model; this could be represented by a method getEmbedding() within the Query class, which calls the language model. The* ***RecommenderEngine*** *(or simply Recommender) is the component that takes a Query and produces a list of Tracks. It holds references to the necessary resources for this task – for instance, it may have a loaded NLP model (model: NLPModel) and access to the collection of track data (tracks: Track[]). The class diagram shows that the RecommenderEngine has an operation recommend(q: Query): Track[], which when invoked will compute similarities between the given query and the tracks, returning a list of Track objects that are recommended. The association labeled “*recommends \*” between RecommenderEngine and Track indicates that the engine can recommend multiple tracks (and a track can be recommended to many users/queries over time).

This Python-based engine will run as a microservice, exposing an HTTP API. The Node.js backend will communicate with it using RESTful calls. This design isolates the machine learning logic from the core backend, facilitating independent development and scaling.

### 4. API Design

Communication between the frontend and backend, and between backend and the Python service, will follow RESTful principles. Example endpoints include:

* POST /recommend: Accepts user input text and returns a list of recommended tracks.
* GET /tracks/:id: Fetches metadata for a specific track.

All endpoints will return JSON responses and be secured using token-based authentication.

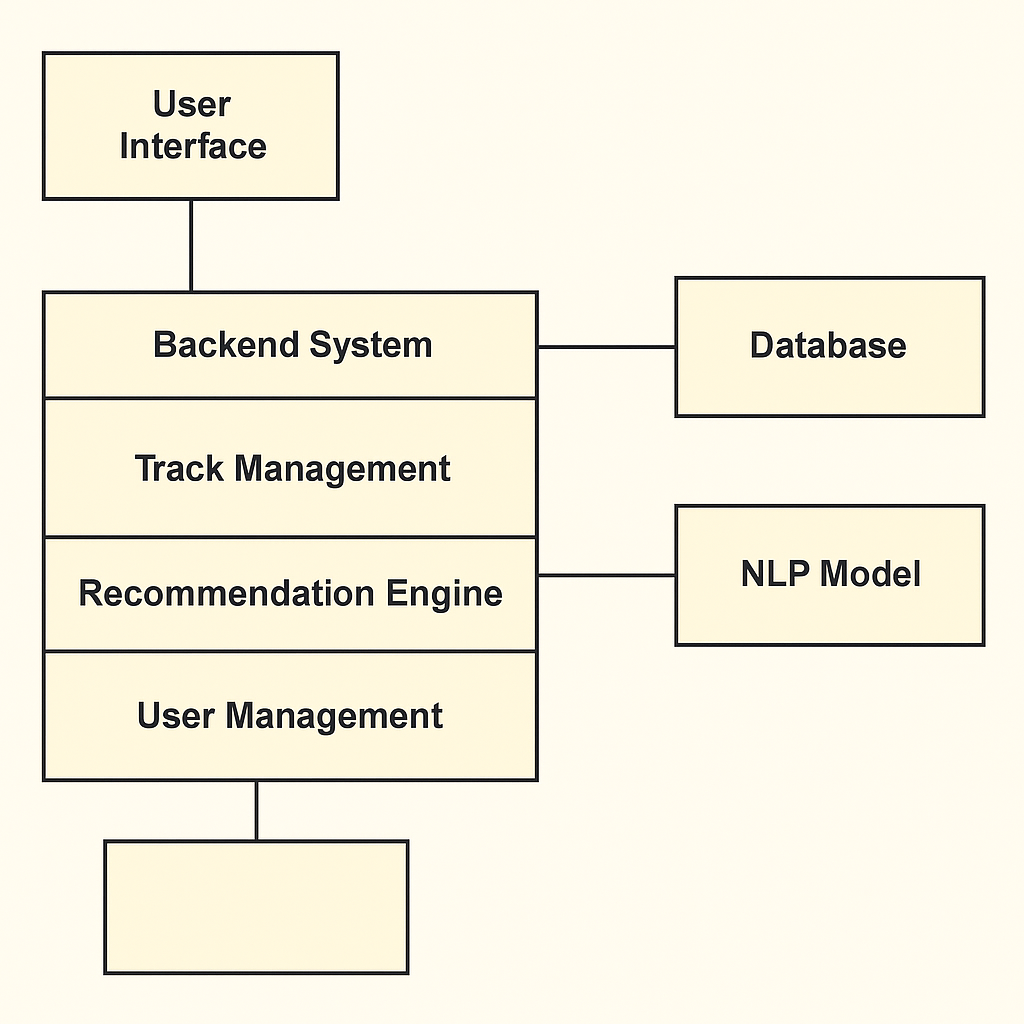


Figure 2

Figure 2: UML component diagram of the music recommendation system. It shows how the system is divided into front-end (User Interface), backend logic (including Track Management, Recommendation Engine, and User Management), and external dependencies (Database and NLP Model). The backend serves as the core that orchestrates user input, metadata retrieval, and similarity computation. This visual representation highlights the modular and decoupled nature of the architecture, making it scalable and maintainable.

### 5. Security and Scalability

To ensure reliability and data safety:

* Input validation and sanitation will be enforced to prevent injection attacks.
* Rate limiting will be used to avoid abuse.
* Authentication tokens (e.g., JWT) will protect API access.

The modular microservice architecture also allows each component to scale independently. For instance, the Python service can be replicated separately under heavier load.

### 6. Deployment Strategy

The system is designed to be deployed on cloud platforms such as **Render**, **Heroku**, or **Vercel**. The frontend and backend can be hosted separately or as part of a monorepo with environment-specific builds. A CI/CD pipeline via GitHub Actions will automate deployment.

### 7. System Architecture Diagram

A diagram representing the complete system flow will be added in the final version of the thesis. It will show the path from user input on the frontend through the backend, to the recommendation engine, and back to the UI with results.

### 8. Data Flow Example

1. The user types: "relaxing music for focus"
2. Frontend sends this query to the Node.js API.
3. Node.js forwards the request to the Python microservice.
4. Python vectorizes the query, compares it to the song database, and returns top matches.
5. Node.js receives and relays the response to the frontend.
6. The UI renders the results with cover art, song names, and play links.

## Technical Stack and Architecture

To support the implementation of a real-time, intelligent, and audio-aware music recommendation system, a multi-layered technical architecture has been employed. This architecture leverages modern web development technologies, machine learning libraries, and pre-trained natural language processing models. The system integrates data processing, machine learning inference, backend API communication, and a dynamic frontend interface.

### 1. Frontend

The frontend of the system is developed using modern JavaScript technologies to ensure a responsive and user-friendly experience.

#### 1.1 Vue 3

Vue.js (version 3) was chosen as the primary frontend framework due to its performance, modularity, and ease of integration. Specifically, the project uses the Composition API with the <script setup> syntax. This approach promotes better logic organization and reusability of code within the single file components.

Vue 3 allows reactive handling of user inputs, which is ideal for building features such as real-time search, audio player controls, and dynamic recommendation updates. The application is served locally using Vite, a fast build tool and development server.

#### 1.2 HTML & CSS

The structure of the application is built with HTML, while CSS is used to style the user interface. Emphasis was placed on creating a clean and minimal design to ensure usability. Additional features such as audio control and responsive layouts were added to improve interaction quality.

#### 1.3 Audio Integration

The <audio> HTML element is used to stream 30-second MP3 clips of recommended tracks. These are fetched from the backend via public endpoints. This component allows users to immediately hear the recommendations without leaving the application interface.

### 2. Backend

The backend is built with Python using FastAPI, a modern, high-performance web framework for building APIs. FastAPI was selected due to its native support for asynchronous operations, automatic OpenAPI documentation generation, and excellent performance with Python-based machine learning workflows.

#### 2.1 FastAPI

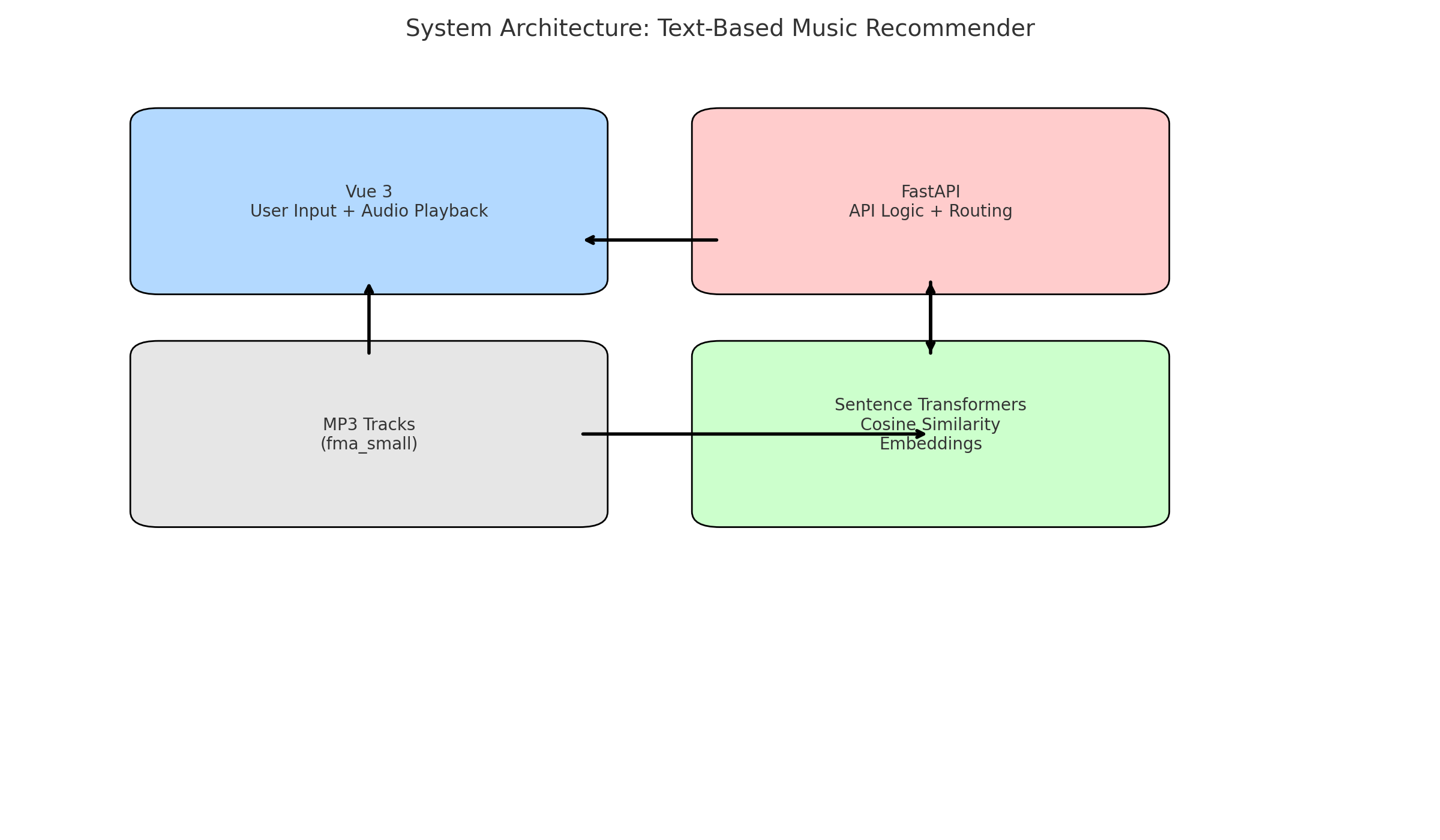
FastAPI manages two primary responsibilities:

* Serving recommendation results via the /recommend endpoint.
* Streaming audio files through /audio/{subfolder}/{filename} endpoint.

CORS middleware is used to allow requests from the Vue.js frontend (localhost:5173). All responses are returned in JSON format and designed to be consumed easily by the frontend.

#### 2.2 Dataset Handling (pandas)

The system uses the [FMA-Small](https://github.com/mdeff/fma) dataset, which includes 8,000 MP3 tracks, each 30 seconds long. Track metadata is loaded from a CSV file using the pandas library. Fields such as title and genre are combined to generate descriptive text used for matching with

user queries.

#### 2.3 Preprocessing and Description Vectorization

A descriptive string is constructed for each track, combining the title and genre. This is later used for semantic similarity comparisons against user queries. The path to each audio file is also programmatically generated using the track ID and formatted into a clean string (e.g., "003/003264.mp3").

### 3. Machine Learning Layer

The core intelligence of the system is built around natural language processing and vector similarity search.

#### 3.1 Sentence Transformers

The system uses the sentence-transformers library and the all-mpnet-base-v2 pre-trained model. This model maps natural language sentences to fixed-size embeddings in a 384-dimensional space. It is specifically designed for semantic similarity tasks, making it ideal for free-text music recommendation queries.

#### 3.2 Vector Embedding and Similarity

Every song description is encoded once into a vector using the transformer model. When a user submits a query (e.g., "calm jazz for studying"), the query is also embedded into the same vector space. Cosine similarity is then computed between the query vector and all precomputed song vectors. The top N most similar tracks are selected and returned as recommendations.

To avoid high memory usage, a subset of 300 tracks was used during early-stage development. Embeddings are computed serially to avoid multiprocessing issues on macOS with NumPy 2.x.

### 4. Audio File Serving

Each recommended song is returned with a valid path such as 003/003264.mp3, which corresponds to a local file in the fma\_small directory. FastAPI uses the FileResponse class to stream this file directly to the client via the /audio/ endpoint. This ensures fast access without requiring an external CDN.

### 5. System Integration and Flow

#### 5.1 Request Flow

1. User types a search query (e.g., "calm jazz").
2. Vue frontend sends a GET request to /recommend?query=calm jazz.
3. FastAPI embeds the query, computes cosine similarity, and returns the top N matches.
4. Vue receives results, displays title/genre/score, and loads the correct MP3 path into the audio player.

#### 5.2 Deployment and Development

* Frontend: npm run dev via Vite on port 5173.
* Backend: uvicorn app.main:app --reload on port 8000.
* Audio files and metadata are stored locally for fast access.

### 6. Summary

The system leverages:

* Vue 3 + Vite for modern frontend architecture.
* FastAPI + Python for backend services.
* SentenceTransformers + PyTorch for embedding.
* FMA dataset with real MP3s for playback.
* Clean RESTful API design for communication.

This architecture offers a modular, scalable, and research-ready environment for exploring audio-based music recommendations driven by natural language. Future enhancements can include personalized models, expanded datasets, and real-time Spotify API comparison.

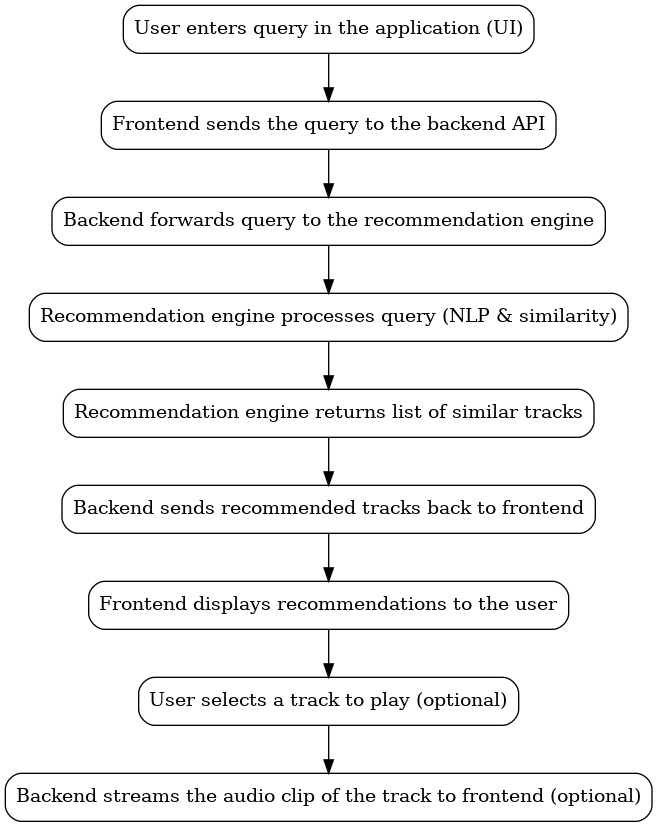


Figure 3

**Figure 3: Sequence diagram** (behavioral flow) for the music recommendation process. The steps are as follows: First, the *User* enters a query into the application’s search bar (e.g., *“relaxing music for focus”*). Upon submission, the **Frontend** sends the query to the **Backend API** over an HTTP request (Step 1 → 2 in the figure). The backend then **forwards the query to the Recommendation Engine** component (Step 3), which processes the query as described earlier – by converting the text to an embedding and comparing it with track embeddings (Step 4). Once the **Recommendation Engine finds similar tracks**, it returns a list of the top matches back to the Backend (Step 5). The **Backend** then sends this list of recommended tracks back to the **Frontend** in a JSON response (Step 6). The frontend receives the data and **displays the recommendations to the user** (Step 7), typically by rendering each track’s information in the UI. At this point, the user sees, for example, a list of 5–10 song suggestions that fit the query *“relaxing music for focus”*.