Project Report: Mood-Based Music Detector

Project Title	**Mood-Based Music Detector**
Course	Computer Science and Engineering (CSE)
Submitted By	Gaurav Jain
Enrollment Number	25BET10044
Date of Submission	November 2025

Introduction

The Mood-Based Music Detector is a command-line application developed in Python. Its primary function is to analyze a user's current emotional state, or "mood," through a series of simple, interactive questions and subsequently generate a personalized playlist from a predefined music database.

This project demonstrates core programming concepts such as data structures (dictionaries), file handling (JSON for history), basic logic, and algorithm design (mood scoring). The application provides a simple, functional example of a recommendation system, making music suggestions tailored to the user's input, and includes a mechanism to save and track historical usage.

Problem Statement

In today's fast-paced digital environment, users often seek music that perfectly matches their current emotional or task-oriented state (e.g., relaxing when stressed, focusing while working). The problem this project addresses is the lack of a quick, simple, and direct utility to curate a relevant micro-playlist based on an immediate subjective self-assessment of mood, without relying on complex, resource-intensive analysis of external data (like audio features or real-time biometrics).

Goal: To develop a lightweight Python program that accurately maps subjective user responses to a dominant mood and suggests a small, curated playlist accordingly.

Functional Requirements

Requirement ID	Description
FR1	**User Input:** The system must accept user input for a series of Yes/No questions (represented as 1 or 2) to determine their current emotional state.
FR2	**Mood Scoring:** The system must calculate a score for each predefined mood category (happy, sad, relax, focus, energetic, old, romantic).
FR3	**Mood Detection:** The system must identify the single dominant mood based on the highest calculated score.
FR4	**Playlist Generation:** The system must randomly select a fixed number of songs (defaulting to 3) from the music database corresponding to the detected mood.
FR5	**History Logging:** The system must save the user's name, detected mood, score, generated playlist, and timestamp into a persistent JSON file (music_history.json).
FR6	**History Retrieval:** The system must be able to load existing history from the JSON file at startup.
FR7	**Output Display:** The system must clearly display the detected mood and the resulting generated playlist to the user.

Non-functional Requirements

Requirement ID	Description
NFR1	**Usability:** The interface must be simple and text-based (command line) with clear instructions (1 for No, 2 for Yes).
NFR2	**Maintainability:** The music database (music_db) must be easily modifiable by a developer to add, remove, or change songs and mood categories.
NFR3	**Performance:** Mood calculation and playlist generation must be instantaneous, as the data set is small.
NFR4	**Data Integrity:** History data must be stored in a structured, standard format (JSON) to prevent data corruption between runs.

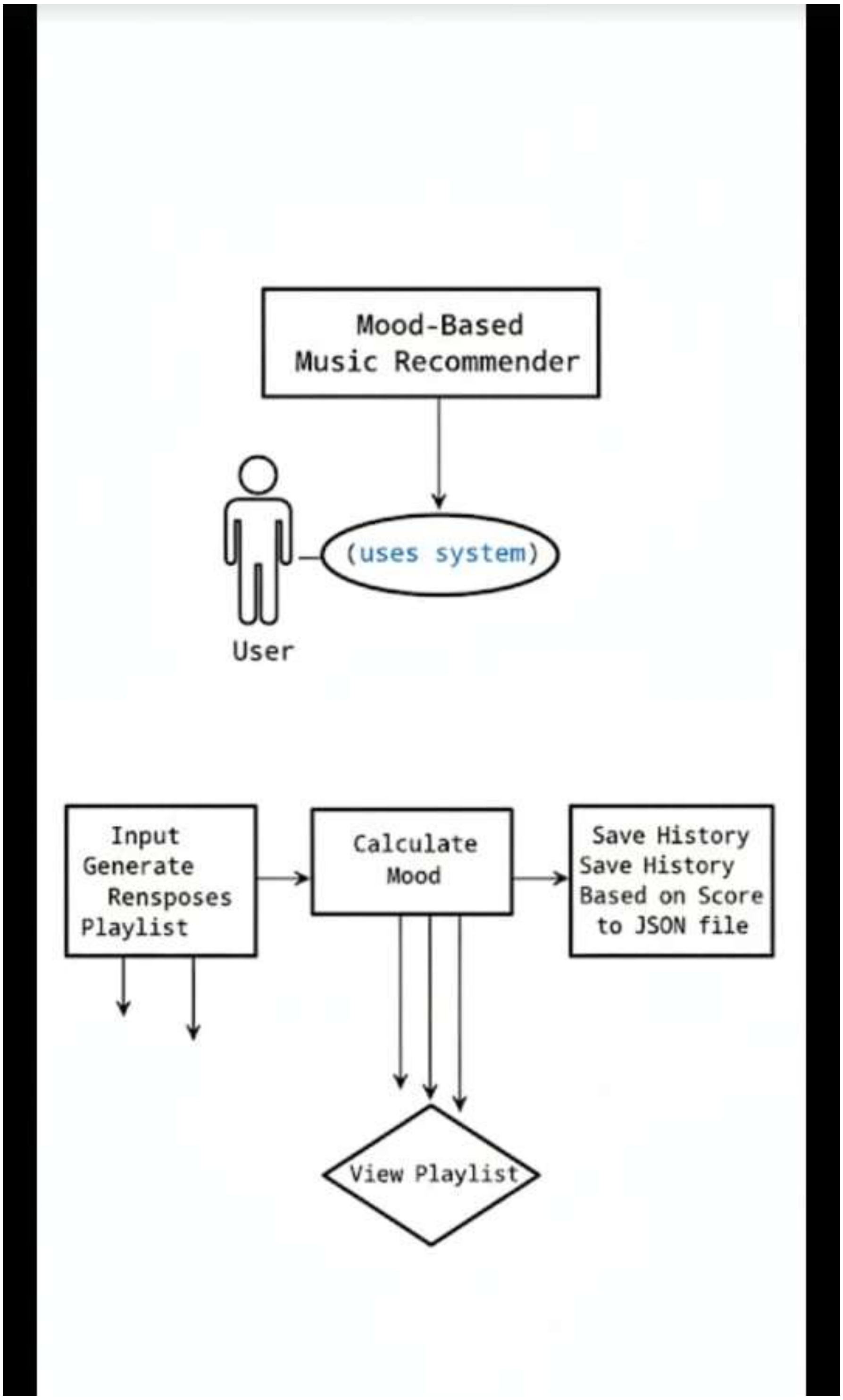
System Architecture

The system uses a **Monolithic Architecture** typical for small, standalone Python scripts. It consists of three main logical components:

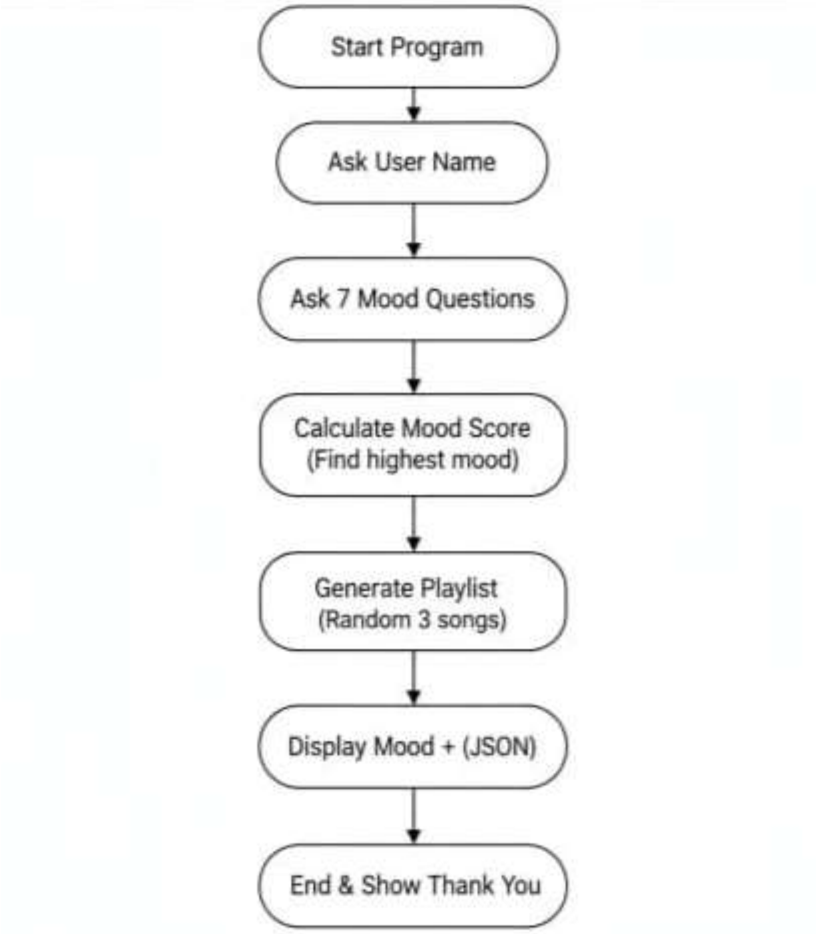
1. **Data Layer:** Stores the static song information (music_db dictionary) and manages the persistent history (music_history.json file).
2. **Logic Layer:** Contains the core functions for mood scoring (calculate_mood_score) and playlist selection (generate_playlist).
3. **Presentation Layer:** Handles user interaction via the console (main function, input() and print() statements).

Design Diagrams

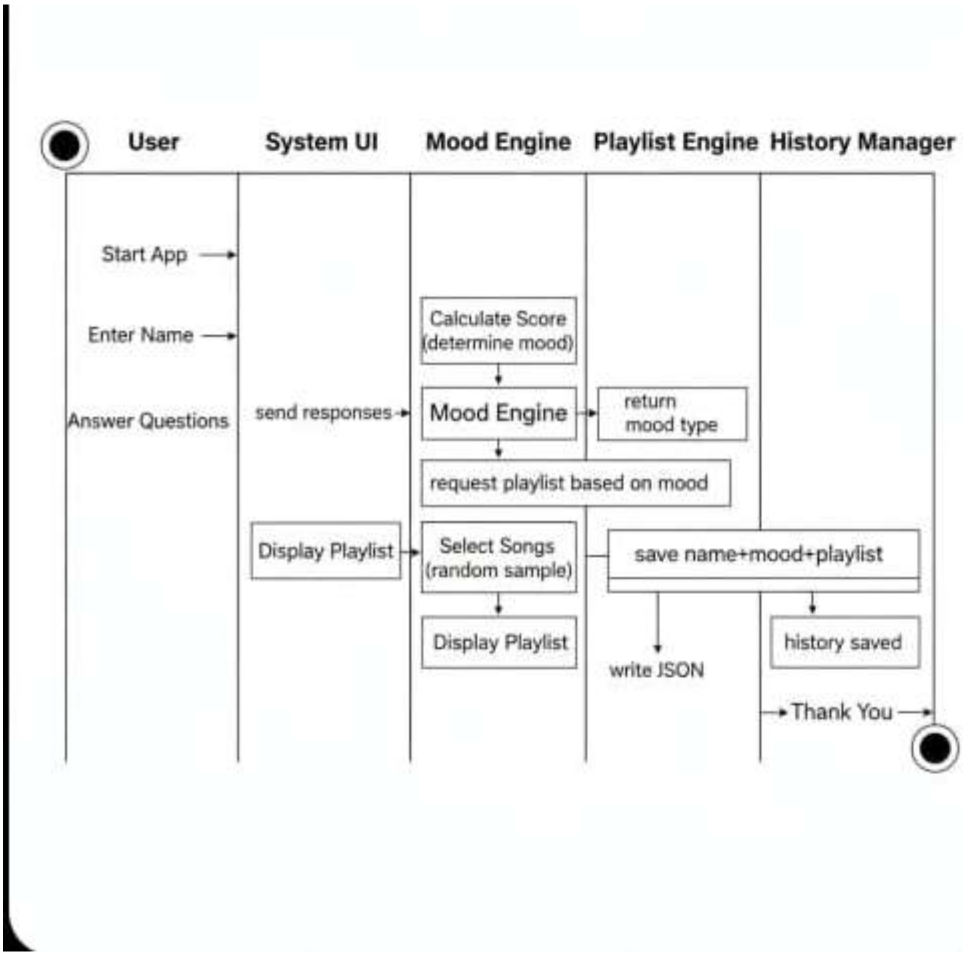
Use Case Diagram



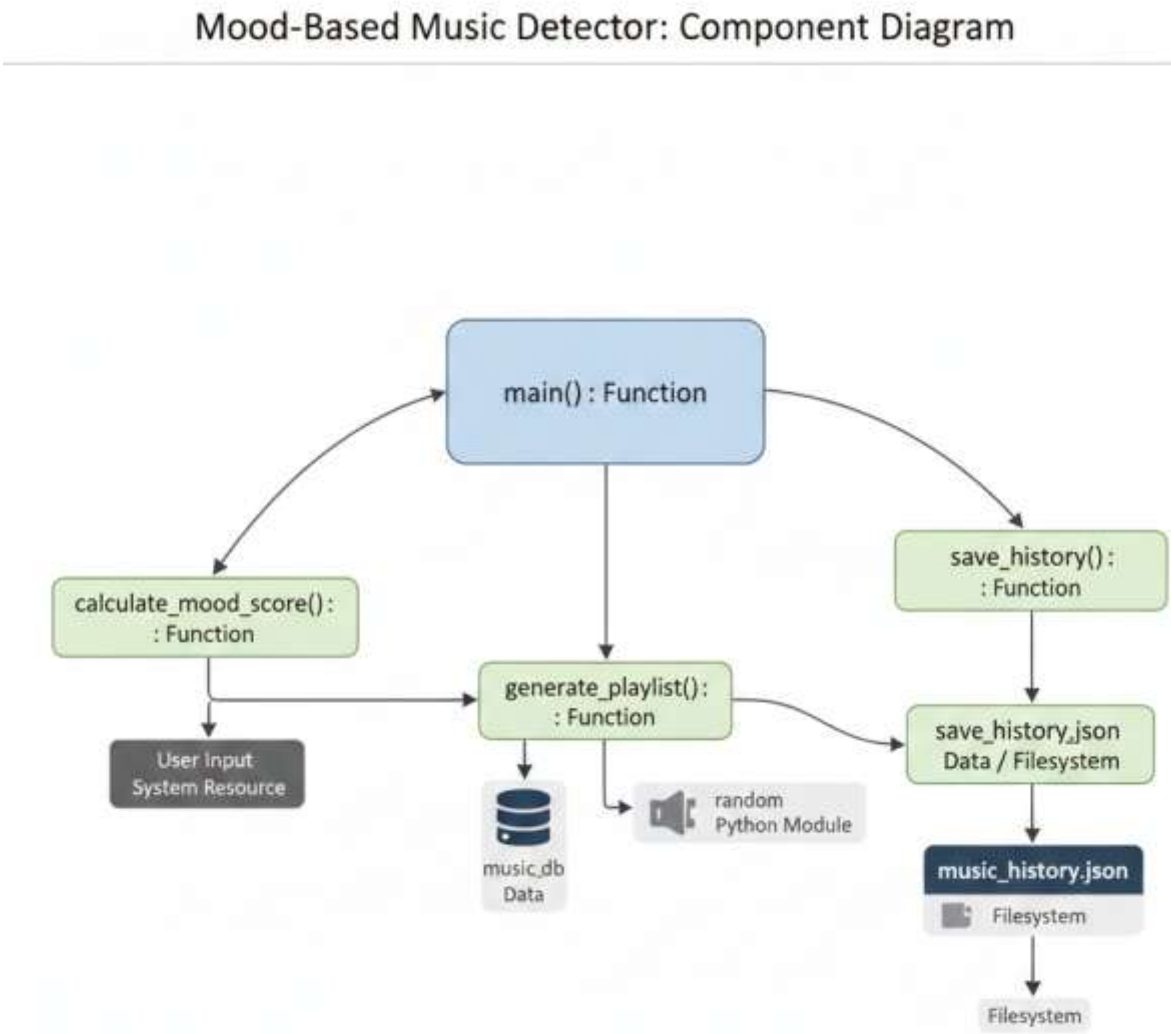
Workflow Diagram



Sequence Diagram



Class/Component Diagram



Conceptual Data Schema (JSON Document Structure)

The data layer uses a simple document structure for the history file (music_history.json), as it does not rely on a relational database.

Field Name	Data Type	Description
name	String	The user's name.
mood	String	The dominant mood detected (e.g., "energetic").
score	Integer	The highest score achieved for the dominant mood.
playlist	Array of Strings	The list of recommended songs.
time	String	The timestamp of the session (YYYY-MM-DD HH:MM:SS).

8. Design Decisions & Rationale

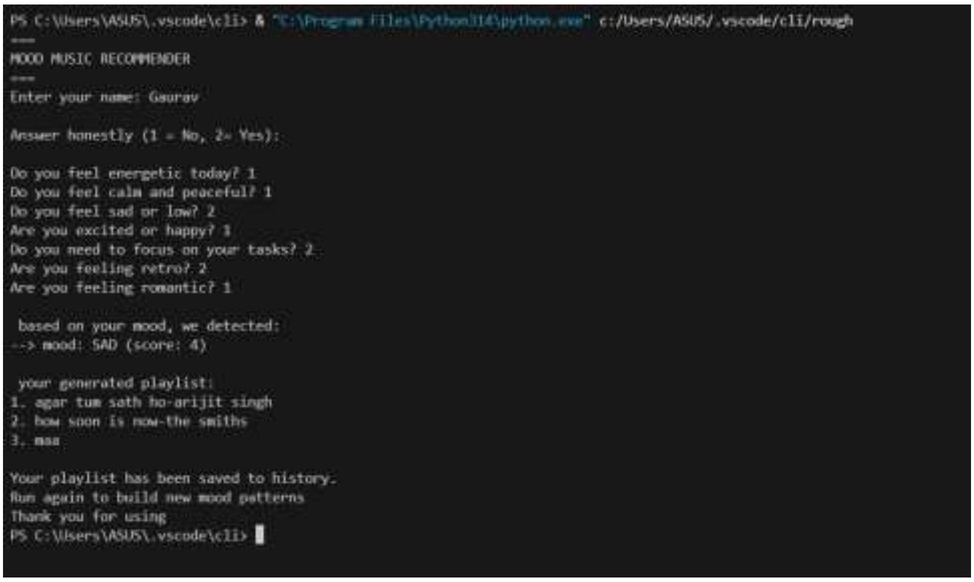
Decision	Rationale
Use of Dictionary for music_db	Dictionaries provide O(1) average time complexity for lookups, making mood-to-playlist retrieval extremely fast and efficient.
Question-Based Scoring	The scoring system (calculate_mood_score) is simple and allows for weighted prioritization (e.g., sad is multiplied by 2) to ensure a strong, dominant mood is selected even if multiple positive feelings are present.
JSON for History	JSON is a human-readable, lightweight, and language-agnostic format, ideal for persistent storage in small Python applications. It naturally supports nested structures like the playlist array.
Random Sampling in Playlist	Using random.sample() ensures that users receive variety within the same mood category across runs, increasing replayability and reducing predictability.
Single-File Implementation	All logic and data are contained in one file, simplifying deployment, sharing, and maintenance for a small-scale class project.

9. Implementation Details

The project is implemented in Python, utilizing built-in modules only:

- **json**: Used for reading from and writing to the music_history.json file. The indent=4 argument in json.dump() ensures the history file is formatted cleanly and is human-readable.
- **random**: The random.sample() function is used to select a non-repeating subset of songs for the playlist.
- **datetime**: Used to generate and format the current timestamp for history logging.
- **os**: Used in load_history() to check if the history file exists, preventing errors on the first run.

10. Screenshots / Results (Simulated Output)



11. Testing Approach

The testing approach was primarily **Unit Testing** and **Black Box Testing**.

Test Type	**Description**
Unit Testing (Mood Logic)	Inputting known combinations (e.g., all '2's for energetic questions, all '2's for sad questions) to verify that calculate_mood_score() correctly identifies the expected dominant mood and returns the correct score.
Unit Testing (Playlist)	Ensuring generate_playlist() correctly handles cases where the requested count (3) is larger than the available songs (e.g., for "old" mood with only 2 songs) without crashing.
Edge Case Testing (History)	Deleting the music_history.json file before a run to ensure load_history() handles the file not existing gracefully (i.e., returns an empty list).
Black Box Testing (User Flow)	Executing the program as an end-user, ensuring the prompts are clear, input handling is robust against non-integer inputs (though not explicitly handled, basic console error handling applies), and the final output is formatted as expected.

12. Challenges Faced

1. **History File Management:** The primary challenge was ensuring the program did not crash when the music_history.json file did not exist (on the very first run). This was solved by using the os.path.exists() check in the load_history function.
2. **Weighted Scoring:** Designing the scoring system to accurately reflect a user's *single* dominant mood was tricky. For example, a high "sad" score should likely override a low "happy" score. This was mitigated by applying a multiplier (*2) to the sad question score.
3. **Preventing Duplicates:** While random.sample() inherently prevents duplicates in the playlist, ensuring the same song wasn't recommended *too* frequently across sessions (a future enhancement) was a consideration for scalability.

13. Learnings & Key Takeaways

1. **Importance of File Handling:** Learned how to safely manage external data files (.json) in Python, including checking for file existence and handling serialization/deserialization.
2. **Data Structure Efficiency:** Reinforced the importance of using appropriate data structures (dictionaries) for fast key-value lookups in recommendation systems.
3. **Algorithm Design:** Gained experience in designing a simple, weighted algorithm (calculate_mood_score) to map subjective input to an objective outcome.
4. **Modularity:** The project's structure, with separate functions for loading, saving, calculating, and generating, demonstrates good modular programming practices.

14. Future Enhancements

1. **Advanced Input Validation:** Implement robust input handling using try-except blocks to prevent crashes if the user enters text or numbers outside of the expected '1' or '2'.
2. **Weighted Random Selection:** Instead of simple random sampling, songs could be weighted (e.g., based on frequency in history or user ratings) to provide smarter recommendations.
3. **User-Defined Moods:** Allow the user to input custom mood categories and assign songs to them during runtime, storing this expanded database in a configuration file.
4. **CLI Interface Improvement:** Utilize libraries like Click or Typer to create a more professional and robust command-line interface.

15. References

1. Python Standard Library Documentation (for json, random, datetime, os modules).
2. The provided mood-basedmusic detector source code file.