TED UNIVERSITY

CMPE 492 / SENG 492 Senior Project

<<BooTunes>>

Test Plan Report

Spring 2025

Team Members:

Mehlika Eroğlu, 58717180232, Software Engineering

Melih Aydın, 28180576276, Computer Engineering

Batuhan Dalkılıç, 11491297756, Computer Engineering

Elif Alptekin, 31376233198, Computer Engineering

Supervisor: Venera Adanova

Jury Members:

Gökçe Nur Yılmaz

Eren Ulu

Contents

CMPE 492 / SENG 492 Senior Project Cover

1. Introduction

We have prepared this assessment in order to systematically plan, execute and preview the results of the test processes of the "Emotion-Based Music Experience Application" that we are developing. As stated in our previous reports, the application analyzes the PDF format books or text documents uploaded by the users page by page, determines the dominant emotion on the pages and plays a music genre appropriate to this emotion. In this way, it is aimed to provide the user with a personalized music experience that provides emotional harmony while reading.

This plan aims to ensure that the software is both technical and compliant with user expectations by testing the functional accuracy, system stability and user experience of the application. It will also enable us to foresee potential risks that may arise during the testing process and take the necessary precautions in advance.

2. Features to Be Tested

In this section of our report, we have detailed the basic functional features planned to be tested in the developed emotion-based music application. The main purpose of the application is to perform an emotional analysis of each page in a PDF document uploaded by the user, play music genres appropriate to the analysis result, and thus emotionally enrich the reading experience.

The main features to be tested are:

2.1. PDF File Uploading

The application's user interface should allow PDF format files to be uploaded without any problems. The system should parse the uploaded file page by page and make it suitable for analysis.

2.2. Page/Chapter Based Sentiment Analysis

The text of each page and chapter in the uploaded PDF must be analyzed using natural language processing (NLP) techniques to accurately identify the dominant emotion. The application must be able to identify and distinguish basic emotions such as joy, sadness, anger, fear, surprise, and calm.

2.3. Matching the Music Genre to the Emotion

Determining the music genre that corresponds to the detected emotion is one of the most important components of this feature. For example, the system should match genres such as slower-paced music or classical music for the emotion of “sadness”; and genres such as pop or dance music for the emotion of “joy.”

2.4. Playing Music from the Music Pool

A playlist suitable for the selected music genre should be called from the music clusters that we will create and presented to the user instantly. Personalization according to the user's choices is also among the plans.

2.5. Page and Music Synchronization

When the user moves to different pages or chapters within the PDF, the sentiment analysis of the new section should be done instantly and the current music should be changed accordingly or remain the same depending on the situation. In addition, the user's song selections should be processed. This synchronization should be tested.

2.6. User Interface and Usability

In order for the user to use the application easily, the interface must be understandable, responsive and error-free. The correct display of feedback, visuals and error messages is also within the scope of the test.

2.7. Interaction with Volume Control and Music Integration

The user is expected to be able to change the volume, pause and resume music, and interact with music content (e.g. skip or like songs). The accuracy and stability of these functions will be tested.

For each of these features, detailed test scenarios will be created, success criteria will be determined, and test results will be reported separately. Thus, the functional reliability of the system and the user experience will be ensured.

3. Testing Methodology

In this section, we outline the testing techniques and strategies that will be used to ensure the functional correctness, reliability, performance, and usability of the BooTunes application. Testing will be conducted at multiple levels, including unit, integration, system, performance, user acceptance, and beta testing, using both manual and automated tools.

3.1 Unit Testing

Unit testing focuses on verifying the correctness of individual functions, methods, or classes in isolation from the rest of the application. In the BooTunes system, these units belong to subsystems such as emotion analysis, music selection, visual generation, user interaction, and data management.

The objective is to ensure that each building block performs its intended logic under various conditions, including edge cases, incorrect input, and boundary values.

3.1.1 Scope of Unit Testing

Subsystem	Module / Class	Unit Test Target
Emotion Analysis	EmotionAnalyzer	Detecting correct emotion label from text input Handling ambiguous or multi-emotion sentences

Emotion Analysis	TextPreprocessor	Cleaning and tokenizing raw text Removing noise (punctuation, HTML tags, etc.)
Emotion Analysis	EmotionLabelMapper	Mapping model output scores to standardized emotion categories
Music Management	MusicSelector	Mapping emotions to genre correctly Selecting the correct track from playlist pool
Music Management	PlaybackController	Play, pause, skip logic Handling edge cases (empty playlist, repeated songs)
Music Management	VolumeManager	Adjusting volume levels Toggling mute/unmute states
Visual Management	VisualGenerator	Generating appropriate visuals for given emotions Caching and fallback behavior
Visual Management	ThemeSelector	Assigning visual themes to each emotion
PDF Processing	PDFParser	Extracting clean text per page Handling corrupt or non-standard PDFs
PDF Processing	PageNavigator	Page-forward and page-backward navigation logic
User Interaction	BookmarkManager	Saving and retrieving bookmarks Handling boundary conditions (first/last page)
User Interaction	SummaryGenerator	Generating accurate and concise summaries Handling edge cases (short books, disconnected text)
User Interaction	UserPreferenceHandler	Updating and retrieving user preferences (genre, emotion intensity)
Data Layer	DatabaseHandler	Insert, read, update, delete operations (CRUD) for bookmarks, summaries, and playback data
Data Layer	SessionManager	Managing per-user reading sessions and state
API Integration	SpotifyService	Fetching playlists, refreshing tokens, handling Spotify errors
API Integration	EmotionModelAPI	Sending requests to the NLP model Parsing and validating emotion predictions
Utility Functions	TextCleaner, Logger	Text normalization Log formatting Helper method accuracy

To ensure comprehensive validation of each unit, our testing strategy includes positive, negative, boundary, and exception-handling test cases. Mocking techniques are applied where external dependencies are involved, such as Spotify's API and emotion-to-visual services, allowing isolated and reliable tests. We aim to achieve at least 80% code coverage in core modules like EmotionAnalyzer, MusicManager, and BookController, monitored through automated tools such as coverage.py and Jest's coverage reports. Unit tests are implemented using pytest for backend Python components and Jest for React-based frontend logic. Detailed test cases, each labeled with a unique Test ID, including their input conditions, expected outputs, and priorities, are presented in Section 9 – Test Cases.

3.1.2 Tools & Frameworks

To support structured and maintainable unit testing in the BooTunes application, we use a mix of tools suited for our backend and frontend environments. These tools enable effective test creation, mocking, and validation across modules such as emotion detection, music recommendation, visual generation, and user interaction logic.

Backend:

- pytest: Used for writing and executing unit tests in backend components such as emotion analysis and book parsing.
- unittest.mock: Enables mocking of Spotify API calls, file access, or database operations to isolate and test functions individually.
- coverage.py: Provides code coverage metrics to ensure critical paths in the backend logic are adequately tested.

Frontend:

- Jest: The core framework for testing JavaScript functions, UI components, and state logic in React-based interfaces.
- React Testing Library: Enhances Jest by simulating actual user interactions and DOM behavior.

Mocking & Dependency Management:

- Manual Stubs / Mock JSONs: Used to simulate consistent responses from external APIs such as Spotify or emotion analysis engines.
- Environment Variables: Sensitive data (e.g., API keys) is replaced by test-safe mock values during testing phases.

CI/CD & Automation:

- GitHub Actions (or similar CI tools): Can be integrated to automatically run test suites during pull requests or mergers.
- ESLint / Flake8: Ensure code formatting and syntax standards are followed throughout the codebase.

3.2 System Testing

System testing focuses on validating the end-to-end functionality of the BooTunes application as a fully integrated and operational system. It ensures that the system meets all specified functional and non-functional requirements, operates correctly in realistic user scenarios, and delivers the desired user experience across all platforms.

This phase verifies that individual modules interact properly and form a cohesive product. The goal is to confirm that the application behaves as expected under actual usage conditions.

The main objective of system testing is to validate that the software behaves consistently across a variety of realistic scenarios and platforms. For BooTunes, this involves cross-platform validation, ensuring that emotion analysis influences music and visuals properly, that the synchronization between pages and media is accurate, and that personalization and bookmarking features work seamlessly.
The following table outlines key system test scenarios, categorized by feature, input conditions, and expected outcomes. These serve as a basis for detailed test execution, which is further expanded in Section 9: Test Cases.

Test ID	Feature	Test Scenario Description	Input/Action	Expected Result
ST-001	PDF Upload	Ensure the system parses PDF files correctly	Upload 100+ page book in .pdf format	Text content is parsed and displayed page-by-page
ST-002	Emotion Detection	Detect emotion across multiple pages	Navigate through 5 pages or chapters of emotionally varied text	Detected emotions vary by page and chapters match sentiment
ST-003	Music Matching	Change music based on emotion detected	Page with 'sad' sentiment	Calm/sad genre music is played
ST-004	Visual Theme Adaptation	Display a background visual based on the overall emotion or genre of the book	Open a book categorized under 'nature' or 'melancholy' genre	A nature-themed or emotion-appropriate visual is loaded once and remains consistent

ST-005	Offline Mode	Test playback and visuals without internet	Open pre-downloaded book in airplane mode	No error, music and visuals work seamlessly offline
ST-006	Summary Feature	Generate summary of previously read content	Tap 'Summarize Last 5 Pages'	Accurate 2–3 sentence summary is generated

3.3 Integration Testing

Integration testing is a critical phase in the software development lifecycle, particularly for applications like BooTunes that involve multiple interacting subsystems and external services. This phase is conducted after unit testing, which verifies individual modules in isolation, and acts as a bridge toward full system validation. The primary goal is to ensure that components interact correctly, data flows smoothly, and shared logic behaves consistently when modules are combined.

What makes integration testing especially important in BooTunes is the complexity of its architecture. The application is built upon distinct modules which must exchange data and respond to each other in real-time. The project also relies on external APIs, like Spotify and cloud-based AI models, which introduce variables such as network latency, rate limits, and service downtime. These are areas where integration issues often arise and cannot be reliably detected during unit testing alone.

By thoroughly executing integration testing, we aim to uncover inter-module incompatibilities, broken interfaces, and flawed assumptions about external service behavior before these problems reach the user. This not only prevents costly late-stage debugging but also directly contributes to the stability, reliability, and perceived quality of the final product.

The objectives of integration testing in our application are:

- Ensuring all module interfaces (internal and external) function as expected, triggering appropriate behaviors and returning correct outputs.
- Detect issues such as data format mismatches, logic inconsistencies, and invalid dependency handling that are often missed during unit testing.
- Ensure End-to-End workflow continuity to validate that user-triggered actions (e.g., flipping pages, updating preferences) lead to consistent backend responses and correct content presentation.
- Test the robustness of the system when interacting with external APIs like Spotify, ensuring proper handling of failures or latency through fallback mechanism which has validate external dependencies.

- Ensure that user-defined settings (e.g., preferred genres, brightness, emotion sensitivity) propagate correctly across all dependent components and influence their behavior accordingly.

As there has several approaches to integration testing we consider the all feautures and try to choose the most appropriate one considering the whole structure. Given the hybrid nature of BooTunes such as combining backend logic, AI models, multimedia orchestration, and external APIs, our integration testing adopts a targeted, interface-driven approach. Key integration types include:

- **Internal Integration:** Between backend modules such as BookController → EmotionAnalyzer, and EmotionAnalyzer → MusicManager/VisualManager.
- **External Integration:** Between backend services and third-party APIs (e.g., Spotify) and model providers (e.g., Hugging Face Transformers).
- **Frontend-Backend Synchronization:** Ensuring UI components in React/React Native accurately reflect backend logic and real-time content changes.

Core Integration Points

Integration Point	Integration Description	Expected Outcome
BookController → EmotionAnalyzer	Parsed book text sent for emotion classification	Dominant emotion correctly identified per page
EmotionAnalyzer → MusicManager	Emotion label triggers genre-based track selection	Audio track selected matches emotion and respects user preferences
EmotionAnalyzer → VisualManager	Emotion metadata triggers themed visual rendering	Visual theme reflects emotional tone or fallback to user-defined default
UserPreferencesController → All Components	User-defined settings (e.g., genre, brightness) applied to downstream modules	Preferences are respected across modules; rendering and audio behavior aligns
Backend → Spotify API	API calls made to retrieve music tracks based on detected emotion and genre	API returns expected track metadata, or fallback is used on error
React/React Native UI → Backend	UI requests emotion/audio/visual for page navigation or updates	Backend processes request and responds with appropriate content for immediate use
Backend → MongoDB	Session data, preferences, and history stored and retrieved	Accurate state retention across sessions and devices

To carry out integration testing in BooTunes, we've used a combination of practical tools that help us test how different parts of the system interact with each other. Some of these tools are already being used, while others are planned to be introduced as the project progresses and becomes more stable. These tools help us test both frontend and backend components, simulate API behavior, and check how the system handles real-time communication between modules.

- **Postman / Swagger UI** : Used for manual testing of backend RESTful API endpoints, enabling validation of request-response structures, status codes, and authentication workflows during integration.
- **Jest & React Testing Library**: These tools are utilized for testing our React and React Native interfaces. They let us simulate user actions (like clicking buttons or flipping pages) and check whether the UI correctly sends or receives data from the backend.
- **Console Logging & MongoDB Inspection** Console output and backend logs are reviewed during integration tests to trace API calls, monitor control flows, and inspect real-time state transitions. MongoDB queries and inspection scripts are used to validate data persistence and user session handling across modules.
- **Mocking Libraries:** unittest.mock (Python) and axios-mock-adapter (JavaScript) to simulate Spotify API, AI model outputs, and failed dependencies.
- **Manual Scenario Scripts**: For more complex behaviors like switching between devices or using the app offline we write step-by-step test scripts and run them manually. This helps us test situations that are harder to automate.
- **Docker**: We also plan to use Docker to create a more stable testing environment. By containerizing our backend services, we'll be able to test interactions more easily without setup issues.
- **GitHub Actions (CI/CD Pipeline)** : A continuous integration setup is planned to automatically run integration and regression test suites upon every pull request or commit to the main branch. This will ensure that new changes do not introduce hidden integration-level bugs and improve test coverage over time.

By attentively validating interactions across internal services, APIs and workflows, this phase plays an important role in eliminating runtime issues, ensuring smooth delivery of a fully immersive, emotion-adaptive reading experience. The results of this phase directly contribute to the robustness, maintainability, and user satisfaction of the final product.

3.4 Performance Testing

Performance testing is the part of the testing process where we check how fast, responsive, and stable the system is when handling different levels of usage. While functional testing checks if BooTunes works correctly, performance testing helps us understand how well it

works under pressure for example, when multiple users are using the app, when a large book is uploaded, or when there are delays from APIs like Spotify.

Since BooTunes involves real-time emotion analysis, music selection, and visual rendering, it's important that the system performs smoothly. If performance is poor, users may experience delays, audio interruptions, or visual glitches—which would break the immersive experience we're aiming for.

Key areas of performance testing include:

- Response time after user actions like navigating pages or uploading a book.
- Speed of emotion analysis and how quickly music or visuals react to changes.
- Behavior under slow or failed Spotify API responses, ensuring fallback systems work smoothly.
- System stability when handling large PDFs or being used continuously over time.
- Resource usage (CPU and memory), particularly on mobile devices or in offline mode.

3.4.1 Tools & Methods

Manual Stopwatch Testing: For response time and load time of features like emotion detection and media playback.

Browser DevTools / Mobile Debuggers: To monitor memory and CPU usage on web and mobile platforms.

Simulated API Delay: Using mocking tools to add artificial delays and test fallback handling.

Offline Mode Emulation: Using browser/mobile developer tools to simulate no internet and observe behavior.

Logging and Console Monitoring: To track loading events, failures, and timeouts during performance-critical actions.

Testing so far has been mostly manual. We've observed load times and responsiveness across different conditions, used browser and tools to monitor performance, and simulated API delays to see how the system handles them. These tests have helped us catch issues like visual delays or audio interruptions during rapid page turns.

Looking ahead, we plan to introduce more automated methods like lightweight load testing or tracking resource usage under longer sessions. These efforts aim to make sure BooTunes stays responsive, scalable, and consistent across devices and environments.

3.5 User Acceptance Testing (UAT)

User Acceptance Testing is the final step in the validation process before the system is considered ready for release or public use. Unlike functional or integration testing, which is carried out by the development team, UAT focuses on verifying that the system meets the expectations of its end users in terms of usability, reliability, and overall experience.

For BooTunes, it is especially important because the success of the application depends not only on its technical functionality but also on how well it delivers an immersive and satisfying user experience. Since the app's core value lies in its ability to enhance reading with emotion-driven music and visuals, it's essential to observe how users interact with the system in real-world conditions and whether it aligns with their expectations.

3.5.1 User Acceptence Testing Process

To conduct UAT, we will provide a near-final version of the application to a small group of representative users. These testers are selected to reflect a range of potential users including casual readers, students in our university, and users unfamiliar with adaptive media.

Each participant will be asked to complete common tasks within BooTunes, such as:

- Uploading a PDF book or using a sample text
- Navigating between pages and observing how music and visuals respond
- Adjusting their preferences (e.g., genre filters, volume, mood intensity)
- Testing personalization settings and seeing their effect in real time
- Using the app in both online and offline modes

We will gather feedback through observation, structured forms, and follow-up questions. Testers will be encouraged to comment not only on functionality, but also on the emotional quality, clarity of the interface, and any moments of friction or confusion.

The system will be considered to have passed UAT if:

- Core features (uploading, emotion analysis, music/visual syncing, personalization) work without critical issues
- Users report a positive or meaningful emotional reading experience
- No major usability blockers or deal-breaking bugs are discovered
- At least 80% of test participants express confidence in using the app independently

3.6 Beta Testing

Beta testing is the final stage before the project is considered ready for delivery. At this point, BooTunes is shared with a small group of external users who were not involved in earlier development or testing. The aim is to observe how the app behaves in real-world conditions, across different devices and usage styles, and to gather practical, unbiased feedback.

This phase helps uncover issues that internal testing may miss such as unexpected user behaviors, device-specific bugs, or edge-case scenarios. It also allows us to validate the system's stability and emotional experience under normal, unscripted usage. Testers will be free to explore features on their own, including uploading books, flipping through pages, reacting to music/visual changes, and adjusting settings. Their feedback will be collected through short surveys and informal follow-ups. We're especially interested in whether the app feels smooth, responsive, and enjoyable to use without explanation.

Beta testing will help us fine-tune final details, catch any remaining bugs, and gain outside perspectives before presenting or releasing the project. The goal is not just to test functionality, but to confirm that BooTunes is truly ready to stand on its own in front of real users.

4. Test Environment

Testing for BooTunes has been carried out primarily in a local desktop environment using both backend and web-based frontend setups.The backend was developed and designed to run in Python using Flask/FastAPI, with MongoDB as the database.

Emotion analysis relies on pretrained models from using Hugging Face Transformers running locally or via Colab.The Spotify API was tested using a personal developer account within rate-limited boundaries. API validation will be carried and test calls planned to made through Postman and Swagger UI.

On the frontend side, testing would be performed on the web version of the application, developed in React and accessed via up-to-date versions of Google Chrome and Opera on Windows. We used tools like Jest and React Testing Library for automating interface behavior and checking UI–backend interactions. API behaviour will be tested to simulated API delays, offline mode, and caching behavior using browser developer tools and mock adapters.

As of now, mobile testing has not been conducted, though React Native components are in development. Ideally, further testing would include various physical and emulated mobile devices, containerized services such as Docker, CI/CD automation through GitHub Actions, and performance tracking tools for wider system observability.

5. Test Schedule

As of now, the development of BooTunes is ongoing, and testing has not yet finished. However, a structured and phased test schedule has been outlined to ensure that once the application reaches a stable version, all necessary testing activities can be executed in a manageable and prioritized way.The testing process will follow a phased and iterative approach. Rather than assigning fixed calendar dates, the schedule is organized around

development milestones. Each testing phase builds on the previous, ensuring thorough coverage from individual components to full-system evaluation and user-level validation.

Predicted Timeline	
1-2 Weeks	Unit + Initial Integration testing
1-2 Weeks	Full integration and system testing
2 Weeks	UAT, feedback loops, bug fixing, and optional beta testing

6. Control Procedures

To maintain the quality, consistency, and traceability of testing activities during the BooTunes project, control procedures have been implemented and followed throughout the ongoing testing phases. These procedures ensure that all identified issues are handled systematically, and that every change introduced as a result of testing is verified and documented with care.

All issues discovered during testing whether functional bugs, integration errors, or UI inconsistencies, are documented in a centralized and shared tracking system which is Github Issues. Each entry includes:

- A unique identifier and timestamp
- Description of the issue and affected component
- Steps to reproduce
- Severity classification as critical, major or minor
- Assigned team member
- Status updates (Open, In Progress, Resolved, Closed)

This process ensures that every issue is visible to the entire team, prioritized based on impact, and traceable throughout its resolution lifecycle.

Any change made in response to testing is subject to peer review before integration. This includes bug fixes, design adjustments, and updates to business logic. All code modifications are version-controlled through Git, and critical updates are documented in a changelog with clear descriptions of what was changed and why.

To avoid introducing regressions, changes are tested both at the point of modification and within the broader system workflow. Peer reviews help maintain code quality and foster team-wide awareness of updates.

Once an issue is marked as resolved, the associated test case is re-executed to confirm that the problem no longer occurs. If the fix affects other parts of the system, additional

regression testing is performed to validate overall stability. This helps ensure that resolving one issue does not unintentionally introduce new ones.

All test executions whether successful or failed are logged in a shared record that includes:

-Test case ID or scenario description

-Date and tester name

-Environment used

-Outcome and any notes or observations

This log serves as both a reference and a form of traceability during final evaluation and future maintenance planning. These control procedures have been essential in ensuring that testing activities are focused, collaborative, and productive. By combining issue tracking, re-testing and change management, we would able to iterate steadily improve quality and reliability as development continues.

7. Roles and Responsibilities

Within the BooTunes project team, responsibilities have been divided based on the functional components of the system. Each member focuses on a distinct area of the application such as emotion analysis, music integration, visual generation, text summarization and is responsible for both development and testing activities within that scope.

To support effective testing, each team member also contributes to defining test cases, executing validation steps, reporting issues, and documenting results relevant to their domain. Some roles overlap in areas such as user interface testing, database management, overall quality review etc.

UAT and beta feedback will be reviewed collectively to determine final updates.

8. Risks & Mitigation

Throughout the development and testing of BooTunes, several potential risks were identified that could impact the quality, stability, or on-time delivery of the application. These risks relate to unexpected limitations of tools and models, technical dependencies, time constraints, limited test coverage, and the complexity of delivering an emotionally adaptable experience. To manage these risks, we left flexibility in the plan to implement mitigation strategies that could be implemented throughout the rest of the project timeline.

- <u>Emotion Analysis Limitations</u>

Risk: Our initial goal of page-level emotion classification encountered practical limitations. Transformer models such as BERT were restricted by the 512-token input size, leading to a loss of emotional continuity and context. Some models also failed to capture subtle or layered emotions in complex literary works.
Mitigation: We transitioned to a hybrid approach by combining page-level analysis for

responsiveness and chapter-based aggregation for deeper accuracy. This allows us to preserve both emotional nuance and system performance. Emotion smoothing was implemented to prevent abrupt changes in playback or visuals, improving the reading flow.

- Visual Generation Problems

Risk: Our original plan to use image generation models such as Stable Diffusion or DALL·E to dynamically produce visuals based on emotion often led to low-quality or bizarre outputs. Generated visuals occasionally included unrealistic combinations of limbs or misplaced elements, such as animal parts embedded in human scenes.
Mitigation: We replaced dynamic emotion-driven visuals with book-contextual themed images. Instead of generating a new visual per page, we assign a static theme image based on the book's setting or tone. For example, an image of St. Petersburg is used for novels set there, providing a more stable and aesthetically pleasing experience.

- Music Integration API Limitations and Mapping

Risk: Spotify's API posed integration challenges, including rate limits, latency, authentication complexity, and the lack of fine-grained emotional metadata for songs. Our initial idea of dynamically selecting Spotify tracks based on emotion labels became unreliable during testing.
Mitigation: We developed a manual music database with labeled clusters of pre-selected tracks categorized by emotion and genre. This fallback ensures consistent playback even when the Spotify API is slow or inaccessible. The system prioritizes locally tagged tracks but uses Spotify streaming when available.

- Mobile Testing Gap

Risk: Due to scheduling and resource constraints, mobile-specific testing was not conducted during early phases. As a result, there may be unverified issues related to layout scaling, responsiveness, or feature accessibility on mobile devices.
Mitigation: The mobile interface, developed in React Native, has been structured with responsive components. Dedicated mobile testing is planned during the final stage, focusing on UI consistency, offline mode behavior, and cross-device session continuity.

- Subjective Emotional Interpretation

Risk: User perception of music and visuals is subjective, especially when tied to abstract emotional signals. What feels "sad" or "inspiring" to one user may feel disconnected to another.
Mitigation: BooTunes includes multiple personalization options — such as genre preferences, emotion sensitivity controls, and the ability to disable music or visuals altogether. This

flexibility ensures that users can adjust the system to better match their individual emotional interpretations.

9. Test Cases

To verify the functional correctness and overall behavior of BooTunes, a set of structured test cases has been prepared and organized according to the core modules and user-facing features of the system. Each test case is designed to evaluate a specific function or interaction, with a focus on validating input/output behavior, integration flow, and user experience alignment.

Test ID	Feature	Test Description	Input	Expected Output	Actual Result	Status
TC-001	Emotion Analysis	Detect dominant emotion from chapter text	Chapter 1 content	Emotion: Sadness	Emotion returned: Sadness	Pass
TC-002	Emotion Analysis	Handle unsupported language or unreadable text	Page with OCR errors	Return neutral or warning	Returned: Neutral	Pass
TC-003	Music Matching	Play correct genre for 'Joy' emotion	Emotion: Joy	Pop or upbeat track is played	Upbeat track started	Pass
TC-004	Music Matching	Fallback to curated music if Spotify fails	No Spotify connection	Play from local emotion-mapped database	Not connected	Not tested
TC-005	Visual Generation	Display correct theme image based on book setting	Book set in Tokyo	Tokyo image is displayed	Tokyo theme image shown	Pass
TC-006	Summary Feature	Generate summary for selected chapter	Chapter 2 text	Brief summary that captures essence & tone of the chapter	Summary generated	Pass
TC-009	Personalization	Apply user's genre and emotion intensity preferences	Jazz preference, low intensity	Show mellow jazz tracks		Not tested
TC-010	Frontend-Backend	Update emotion/music/visual on page turn	User flips to page 3	New emotion analyzed, visuals/music updated		Not tested

TC-011	Offline Handling	Continue playback and emotion flow when offline	Device enters offline mode	Cached music/visuals used, no crash	Local tracks used, stable UI	Pass

Traceability Matrix

Feature	Related Test Case(s)
PDF File Uploading	ST-001, TC-001
Page/Chapter Based Sentiment Analysis	ST-002, TC-001, TC-002
Matching the Music Genre to the Emotion	ST-003, TC-003, TC-004
Playing Music from the Music Pool	TC-003, TC-004
Page and Music Synchronization	ST-003, TC-010
User Interface and Usability	TC-010, UAT Steps
Interaction with Volume Control and Music Integration	ST-006, TC-009
Summary Generation	ST-006, TC-006
Offline Functionality	ST-005, TC-011
Personalization and Preferences	TC-009, TC-006

References

Devlin, J., et al. (2018). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*

Bertolino, A. (2007). *Software Testing Research: Achievements, Challenges, Dreams.*

ACM Code of Ethics: https://www.acm.org/code-of-ethics

Hugging Face Transformers Documentation: https://huggingface.co/docs/transformers

MongoDB Documentation: https://www.mongodb.com/docs/

React Documentation: https://reactjs.org/docs/getting-started.html

React Native Documentation: https://reactnative.dev/docs/getting-started

pytest Documentation – https://docs.pytest.org/

Jest Documentation – https://jestjs.io/docs/getting-started

Postman API Platform – https://www.postman.com/

Spotify API Documentation: https://developer.spotify.com/documentation/web-api

IEEE Code of Ethics: https://www.ieee.org/about/corporate/governance/p7-8.html