# System Verification and Validation Plan for Software Engineering

Team 8 – Rhythm Rangers

Ansel Chen
Muhammad Jawad
Mohamad-Hassan Bahsoun
Matthew Baleanu
Ahmed Al-Hayali

November 6, 2024

# Revision History

| Date | Version | Notes |
|---|---|---|
| 2024-11-04 | 0.0 | Revision 0 |

# Contents

# List of Tables

# 1 Symbols, Abbreviations, and Acronyms

| symbol | description |
|---|---|
| VnV | Verification and validation |
| TDD | Test-driven development |
| API | Application programming interface |
| SRS | Software requirements specification |
| MG | Module Guide |
| MIS | Module Interface Specification |
| HAD | Hazard Analysis |
| UI | User Interface |
| UX | User Experience |
| PEP | Python Enhancement Proposal |
| SLOC | Source lines of code |
| LLOC | Logical lines of code |
| CI | Continuous integration |
| FR | Functional requirement |
| NFR | Non-functional requirement |
| WCAG | Web Content Accessibility Guidelines |
| Los ojos | The eyes |

This document details the Verification and Validation plan, outlining the strategies, tools, and processes to ensure the system meets its design specifications and quality standards. Automated testing tools, including linters, unit testing frameworks, and continuous integration (e.g., GitHub Actions), will maintain code quality and enable efficient verification of functionality. The roadmap covers key phases: initial setup and planning to define testing tools and team responsibilities; systematic unit, integration, and automated testing to validate module interactions and overall functionality; structured usability assessments to gauge user satisfaction; and a final verification phase to ensure all functional and non-functional requirements are met. This phased approach, combined with both automated and manual validation techniques, aligns the project with its technical, usability, and compliance objectives.

# 2 General Information

## 2.1 Summary

This Verification and Validation (VnV) Plan outlines the testing strategies, tools, and processes designed for *GenreGuru*, a software application that enables users to analyze musical tracks by extracting key features. The system aims to assist users in understanding and classifying musical genres through feature analysis. In addition to core functionalities, *GenreGuru* includes stretch goals for generating genre-based recommendations based on the extracted features and for music generation, where the application would produce new tracks inspired by selected genres. This VnV Plan is structured to ensure that *GenreGuru* meets all specified functional and non-functional requirements, from feature accuracy to user experience.

## 2.2 Objectives

The objectives of this VnV Plan are twofold: to provide confidence that the project satisfies all specified functional and nonfunctional requirements (verification) and to confirm that the end product aligns with user expectations and project goals (validation).

**Verification** Ensuring the product is built correctly according to the defined requirements and specifications, i.e.,

- Assessing each atomic item (unit testing) to confirm it functions as intended.
- Testing interactions between atomic items (integration testing) to verify that components work together seamlessly.
- Performing end-to-end (system) testing to confirm that the entire system functions correctly and meets project specifications.
- Applying Test-Driven Development (TDD) practices where feasible to enhance code reliability from early stages.
- Refactoring code regularly to eliminate code smells and maintain clean, efficient, and maintainable code.

**Validation** Ensuring the project meets needs of the stakeholders within the intended environment, typically through acceptance testing. This process is often performed by individuals external to the project team.

## 2.3 Challenge Level and Extras

**Challenge Level** As stated in the Problem Statement document, the challenge level of this project is general. In light of the revised scope of the project, i.e., omitting the generation component and delaying the recommendation component until after acceptable completion of the featurizer, as described in the hazard analysis document, we have not changed the challenge level of our project. In retrospect, perhaps it was likely the case that our project should have been labelled as a challenging project instead.

**Extras** As stated in the problem statement document, the team plans to include user & API documentation for reference, usability testing for easy startup, and design thinking to build an intuitive user interface. Note that the generation component is now a stretch goal instead of a core feature, as discussed in the hazard analysis document.

## 2.4 Relevant Documentation

This section lists the documents relevant to the VnV Plan for *GenreGuru*. Each document provides vital information for implementing effective testing strategies, defining specific requirements, or guiding system design.

- **Software Requirements Specification (SRS)** (Author, 2019): The SRS document outlines all functional and non-functional requirements for *GenreGuru*. It serves as the primary reference for establishing test cases, as the VnV plan will verify that each requirement has been met. This document ensures that our testing covers all specified behaviors and performance expectations.

- **Module Guide (MG)**: The MG defines the architecture and organization of the system into modules, detailing each module's responsibilities and dependencies. This structure is essential for integration testing, as it maps out how each component interacts within the system. By following the MG, the VnV plan will verify that module connections and data flows work as designed.

- **Module Interface Specification (MIS)**: The MIS provides detailed descriptions of module interfaces, including inputs, outputs, and expected behavior. This document is critical for interface testing, as it enables the VnV team to check that modules interact correctly. Test cases will focus on validating that modules communicate as specified, using the MIS as a guideline.

- **Hazard Analysis Document (HAD)**: The HAD identifies potential risks and proposes mitigations for *GenreGuru*. The VnV team will use this document to develop test cases targeting high-risk areas, such as data integrity and error handling. This approach ensures that the system is robust and resilient under various scenarios.

- **Design Documents (DD)**: Detailed design documents include diagrams and flowcharts that describe the inner workings of each module. These documents support white-box testing, offering a basis for verifying the internal logic and efficiency of code structures. The VnV team will reference these documents to validate that each module's implementation aligns with its design.

- **Test Plan (TP)**: The TP outlines the overall testing strategy, methodologies, and test case responsibilities. It complements the VnV plan by providing specific instructions for executing and managing the tests, ensuring that testing aligns with project goals and requirements as defined in the SRS and MG.

# 3 Plan

The subsections to follow outline, in order, VnV team members and their corresponding roles, the verification process for the SRS, the design & its components, the VnV document and process itself, the implementation, associated automated testing & verification tools, and software validation tools.

## 3.1 Verification and Validation Team

- *Ansel Chen*: Joint responsibility for client application testing; joint featurizer algorithm testing.
- *Muhammad Jawad*: Joint featurizer algorithm testing.
- *Mohamad-Hassan Bahsoun*: Joint responsibility for client application testing; external service API interface testing; joint featurizer algorithm testing.
- *Matthew Baleanu*: Unit testing for compute; joint featurizer algorithm testing.
- *Ahmed Al-Hayali*: Establish testing infrastructure; Maintain testing infrastructure; Functional testing for server storage; Server networking interface testing; joint featurizer algorithm testing.
- *Supervisor/Dr. Martin V. Mohrenschildt*: Featurizer algorithms validation and limited verification runthroughs.
- *Stakeholders*: Participate in usability testing, acceptance testing, and offer limited design validation input.

## 3.2 SRS Verification Plan

The following approaches will be used for SRS verification:

- Formal reviews with the supervisor
- A checklist that will be given to the supervisor and any peer reviewers. It will also serve as a guide for the developers of the system

- Using feedback from grading to create new checklists and update existing checklists
- Ad-hoc reviews from peers and other teams in the course

This is the initial SRS checklist that reviewers will use. It will be updated as reviews are performed:

- ☐ Does each functional requirement have a detailed and accurate description, rationale and fit criteria?
- ☐ Is each requirement (both functional and non-functional) relevant and necessary?
- ☐ Are all functional requirements traceable to at least one use case?
- ☐ Are all fit critera unambiguous and verifiable?
- ☐ Have all issues opened by reviewers been closed?

## 3.3 Design Verification Plan

The design verification will ensure that each component of the system meets the requirements outlined in the Software Requirements Specification (SRS). Verification will occur through systematic reviews, testing, and validation checklists tailored to each component.

### 3.3.1 Client Application Design Verification

User-facing component for inputting track information and displaying system output.

**Verification Approach** The verification will proceed as we:

- Conduct usability testing with users to assess ease of input and clarity of output.
- Review design documents to ensure UI/UX aligns with requirements.

**Checklist** The design is considered verified if the:

- ☐ User can input track information without errors.
- ☐ System output is clear and comprehensible.
- ☐ User feedback is collected and assessed.

### 3.3.2 Server Networking Design Verification

Represents communication between the server and external components, e.g., between the client application and external service APIs.

**Verification Approach** The verification will proceed as we:

- Review network architecture to confirm it supports required protocols.
- Test for successful data transmission between components.

**Checklist** The design is considered verified if:

- ☐ All intended connections are established.
- ☐ Data loss during transmission is within acceptable limits.
- ☐ Network security measures are implemented and verified.

### 3.3.3 Server Compute Design Verification

Component responsible for processing user requests and issuing featurization and data access requests.

**Verification Approach** The verification will proceed as we:

- Review processing algorithms for efficiency.
- Conduct code reviews to ensure best practices are followed.

**Checklist** The design is considered verified if the:

- ☐ Processing time is optimized for expected load.
- ☐ Code is clean and adheres to project standards.
- ☐ Edge cases are handled appropriately.

### 3.3.4 Server Storage Design Verification

Represents the database for storing data.

**Verification Approach** The verification will proceed as we:

- Review database schema against the SRS.
- Test data retrieval and storage for accuracy.

**Checklist** The design is considered verified if the:

- ☐ Schema supports all required data types and relationships.
- ☐ Data retrieval times meet performance criteria.
- ☐ Data integrity is maintained during transactions.

#### 3.3.5 External Service APIs Design Verification

Represents integration with external services like Spotify or Deezer.

**Verification Approach** The verification will proceed as we:

- Review API documentation for compliance.
- Test integration for successful data exchange.

**Checklist** The design is considered verified if:

- ☐ All API endpoints are functional as expected.
- ☐ Data that is exchanged is accurate and formatted correctly.
- ☐ Error handling for API failures is in place.

#### 3.3.6 Featurizer Design Verification

Handles the featurization process.

**Verification Approach** The verification will proceed as we:

- Review featurization algorithms for correctness.
- Test output against known input cases to validate accuracy.

**Checklist** The design is considered verified if:

- ☐ All features are correctly extracted from input data.
- ☐ Featurization time is within acceptable limits.
- ☐ Results match expected outputs for test cases.

A group discussion will be held bi-weekly to review the verification process and address any outstanding issues. Feedback from classmates will be incorporated throughout the development process, especially during review sessions.

## 3.4 Verification and Validation Plan Verification Plan

The primary method of verification for the VnV plan should be reviews and mutation testing. The main goals these reviews are used for is to capture how complete the test cases are and if they are correct. A checklist would be provided to the reviewers in order to make their job simpler. To these ends, we would like to use the following methods to review the VnV plan:

- Classmate reviews: The VnV plan will be reviewed by another group in order to identify any gaps or inconsistencies. As we cannot expect other classmates to fully grasp the full nature of our project, we mainly expect the peer review to be convering consistency issues and sanity checking the test cases.
- TA review: The course instructors will review the VnV plan to check if it meets the rubric. The rubric would be turned into a checklist for the TA and as the course instructors would understand the project better than the peer reviewers, they could help determine whether the test cases fully cover all the requirements and if the plan is feasible within the capstone timeline.
- Mutation testing: this method would be used to test the quality of our test cases. By introducing small changes to the code of the project, we would be able to test if the resulting output would be the different under the same inputs, thus allowing us to verify whether the test cases that were derived during the VnV plan are truly capable of detecting faults or not.
- Internal team review: Our current process assigns at least one reviewer for each new section before committing changes to the VnV plan. This practice ensures that updates align with the overall project requirements and verifies any new test cases or methodologies introduced.

## 3.5 Implementation Verification Plan

### 3.5.1 Static Analysis

Static analysis is the analysis of program content without execution. Below is a description of static analysis techniques we plan to implement as part of our testing process.

**Linting** Linters like `Pylint`, `Ruff`, or `Flake8` check for errors, enforce a coding standard, identify code smells, and can make code refactoring suggestions.

**Formatting** Formatters like `Black` and `Ruff` (indeed, Ruff is also a formatter) standardize code appearance and adhere to style guides, allowing the code reader to focus on code content.

**Type Checking** Type checkers like `mypy` ensure correct use of variables and functions in code using type hints, as outlined in PEP 484. Type hinting can also serve as documentation when publishing an API reference or developer guide.

**Security Checking** Static security checkers like `Bandit` find common security vulnerabilities in code, e.g., framework misconfiguration (`B2XX`, e.g., exposing the Flask debugger in a production application, allowing remote code execution), blacklist calls (`B3XX`, e.g., loading serialized pickle files), blacklist imports (`B4XX`, e.g., importing `ftplib` for insecure file transfer), cryptography (`B5XX`, e.g., missing certificate validation), and injection (`B6XX`, e.g., testing for SQL injection).

**Code Metrics Analysis** A code metrics analysis tool like `radon` can provide insights on various aspects of the codebase:

> **Raw metrics** Number of lines of source code (SLOC), logic (LLOC), comments, and whitespace.
>
> **Cyclomatic complexity** Number of decisions (or linearly independent paths) in a code block.
>
> **Halstead metrics** Statically-generated program metrics.

**Composite Analysis Techniques** Tools like `Prospector` combine multiple analysis techniques into one, i.e., linting via `Pylint`, `Pyflakes`, or `Ruff`, PEP 8 and PEP 257 formatting via `pycodestyle` and `pydocstyle`, code complexity analysis via `McCabe`, simple security checking via `Dodgy`, packaging quality checking via `Pyroma`, unused modules checking via `Vulture`, type checking via `Mypy` or `Pyright`, and security checking via `Bandit`.

**Code Walkthroughs** Checklist-driven walkthroughs of featurization algorithms with other teammates and optionally the supervisor. Refer to the appendix section 6.3 for a sample checklist.

**Peer Desk Checks** Changes to the codebase will require approval from at least one other group member via a pull request review before merging to the main/production branch.

### 3.5.2 Dynamic Testing

Dynamic testing is the analysis of program runtime responses during and after execution. Below is a description of dynamic testing techniques we plan to implement as part of our testing process. For further details about the automated components, refer to section 3.6.

**System Testing** Tests are orchestrated via Testing orchestration tools like `tox` can manage all testing, from atomic unit tests to end-to-end system tests. System tests outlined in section 4 will be run to ensure necessary requirements are met.

**Unit Testing** Unit testing frameworks like `pytest` or `unittest` can verify that the implementation matches designs described in other system documents.

**User Interface Testing** UI components can be described as a set of discrete interactions, i.e., a transition model can capture user interactions as events then test it. Tools like the `Selenium` Python API automate web-based interactions, i.e., can serve as a testing framework to automate interaction sequences using the UI model.

**Integration Testing** Testing frameworks like `pytest` can be combined with modular fixtures or factories via `factoryboy` to simulate databases or other complex objects for testing operability between various interfaces, i.e., integration testing.

**Regression Testing** Persistence of tests across iterations of the project facilitates ease of regression testing, with automation via GitHub actions.

**Coverage Testing** Code coverage libraries like `coverage` or `pytest-cov` can be used to generate code coverage reports. These reports will inform developers of any possible code execution paths that have not been covered by unit tests.

## 3.6 Automated Testing and Verification Tools

The following automated tools will be used throughout the verification and validation process of the system:

- `Ruff` python linter and formatter
- `Mypy` python type checker
- `Pytest` unit testing framework
- `Coverage` code coverage library
- `Tox` orchestration tool
- github actions

Our automated testing tools will be used to strictly verify the implementation of the system (See the previous section of this document for more details). We decided to use `Ruff` because it is the highest performing python linter and it also doubles as a formatter. Mypy is the most popular python type checker, making it an obvious choice. `Pytest` and `Coverage` are the most popular tools for unit testing and code coverage, respectively. `Tox` was chosen because it is the tool the team has the most experience with.

When a pull request gets created, github actions are triggered. This invokes `Tox`. First, `Tox` will invoke `Pytest` and `Coverage`. This will build the project and run all the unit tests, and then perform code coverage analysis. Then, `Tox` will invoke `Ruff` to perform code linting and formatting. Finally, `Tox` will invoke `Mypy` to perform type checking. Every command will have its output piped to a text file to generate a full report. If no errors are reported by any of the tools, github will enable the ability to merge the pull request.

This automated testing setup allows for every pull request to undergo thorough checks before integration into the main codebase. By using `Ruff` and `Mypy`, we can ensure to maintain consistent code quality and catch common errors early. Using `Pytest` and `Coverage` ensures that the entire project is rigorously covered by unit tests. `Tox` automates this entire process, making it easier to identify any failures across modules and simplifying the process of tracking test results and code behavior. GitHub Actions integrates these tools, creating a reliable CI pipeline that allows the team to focus on development whilst maintaining quality.

## 3.7 Software Validation Plan

### 3.7.1 External Data For Validation

The project cannot function very effectively without access to songs made available by Deezer's API, song snippets made available by Spotify's API, or the track audio features offered by Spotify's API that can serve as the ground truth for us to verify approximate correctness of our featurizer.

### 3.7.2 Requirement Reviews & Task-Based Inspections

Task-based inspections seek to verify the implementation of functional requirements and the degree to which the nonfunctional requirements are met. Requirement reviews, however, are conducted with the stakeholders to select and refine a subset of requirements.

- Requirement reviews with stakeholders can be conducted to offer assurance on the completeness of requirements documents. The goal is to survey stakeholders on the requirements to help with requirement prioritization, refinement, or omission.
- A task-based inspection can also be conducted to verify the correctness of the implementation with respect to the requirements. This would involve stakeholder being given a list of tasks designed to test functional and nonfunctional requirments. The stakeholder would document their experience carrying out these tasks. Ideally, the task-based inspection is conducted with the same stakeholder group as the one involved in the requirements review to obtain a revised opinion on the selection and granularity of requirements.

### 3.7.3 Project Supervisor Demonstration

The project supervisor is Dr. Martin V. Mohrenschildt. During the rev 0 demo, we should explain and justify the project requirements, followed by a demonstration of a project prototype, then a characterization of the correctness of the functional requirements implementation and adherance to the nonfunctional requirements. Unmet requirements must be revised or omitted with justification. Note that Dr. Martin V. Mohrenschildt is an expert in signals processing, thus we strive primarily to collect feedback on our signal processing components.

# 4 System Tests

## 4.1 Tests for Functional Requirements

### 4.1.1 User Input and Interaction Tests

1. Test ID: UII-01
   Control: Manual
   Initial State: The client application is open and ready for input.
   Input: User pastes a valid song reference link.
   Output: The system displays a confirmation that the link has been accepted, and the request process is initiated.
   Test Case Derivation: Confirms that the user can input a song link and initiate a request (Requirement #1).
   How test will be performed:

   - The tester will open the client application and input a song reference.
   - They will then request the system to perform an action and verify that a confirmation message appears, indicating that the reference is accepted.

2. Test ID: UII-02
   Control: Manual
   Initial State: The client application is running.
   Input: Some type of user interaction with the user interface elements (buttons, text, etc.)
   Output: The client application's user interface elements respond to the user's inputs.
   Test Case Derivation: Ensures that the client application is available for user interaction (Requirement #9).
   How test will be performed:

   - The tester will open the client application.
   - They will verify that input fields, buttons, and other elements are visible and responsive.

#### 4.1.2 Connection and Data Flow Tests

1. Test ID: CDH-01
   Control: Manual
   Initial State: Client application is connected to the server and external APIs.
   Input: User submits a request using a valid song link.
   Output: Successful data retrieval and confirmation message.
   Test Case Derivation: Verifies communication between client, server, and APIs (Requirement #2).
   How test will be performed:

   - The tester will submit a reference as input and observe the system processing the request.
   - After processing, the tester will check logs confirming the request has been received.

2. Test ID: EADR-01
   Control: Manual
   Initial State: Server and client are operational.
   Input: A song reference link.
   Output: Song data is retrieved from external API.
   Test Case Derivation: Ensures valid links trigger API data retrieval (Requirement #4).
   How test will be performed:

   - The tester will input a valid song link and check the system for API data retrieval confirmation.

#### 4.1.3 Data Processing and Feature Extraction Tests

1. Test ID: FE-01
   Control: Manual
   Initial State: Server is running.
   Input: A valid song reference link.
   Output: Extracted song features (tempo, pitch, genre, etc.).
   Test Case Derivation: Confirms feature extraction occurs as intended (Requirement #3).
   How test will be performed:

- The tester will input a valid reference song and submit a request for feature extraction.
- The tester will verify that the extracted features match a known expected output.

2. Test ID: AFD-01
   Control: Manual
   Initial State: Client receives analysis data.
   Input: None (after feature extraction).
   Output: Song features displayed in a readable format.
   Test Case Derivation: Ensures extracted features are presented clearly (Requirements #5 and #7).
   How test will be performed:

   - After feature extraction, the tester will confirm that the system returns extracted features.
   - The tester will check that the client displays these features clearly.

#### 4.1.4 4.1.4 Error Handling and Validation Tests

1. Test ID: IVED-01
   Control: Manual
   Initial State: Client application open.
   Input: Invalid or malformed song reference.
   Output: Error message guiding the user.
   Test Case Derivation: Checks input validation and error feedback (Requirements #6 and #8).
   How test will be performed:

   - The tester inputs an invalid reference and checks for messages and user guidance.

2. Test ID: EF-01
   Control: Manual
   Initial State: System is running.
   Input: Deliberate errors introduced in any component.
   Output: Specific error messages generated by each component.

Test Case Derivation: Confirms each component provides feedback (Requirement #6).
How test will be performed:

- The tester will introduce a failure/error scenario.
- The tester will verify that there are error messages specific to the affected component.

#### 4.1.5 Output Display Test

1. Test ID: OD-01
   Control: Manual
   Initial State: Analysis has been completed.
   Input: None.
   Output: The output is displayed to the user in the client application.
   Test Case Derivation: Verifies outputs are displayed for the user to see (Requirement #7).
   How test will be performed:

   - The tester ensures that all outputs from different components appear as intended on the client application.

### 4.2 Tests for Nonfunctional Requirements

#### 4.2.1 APR1 - Minimalist Layout

- **Test ID:** TAPR1
- **Type:** Static, Manual
- **Initial State:** User interface loaded.
- **Input/Condition:** Visual inspection of layout.
- **Output/Result:** Interface presents a minimalist layout with minimal distractions.
- **How test will be performed:** Manually inspect interface layout to ensure it follows minimalist design guidelines.

#### 4.2.2 APR2 - High Contrast

- **Test ID:** TAPR2
- **Type:** Static, Manual
- **Initial State:** Interface set to default theme.
- **Input/Condition:** Check for visual contrast.
- **Output/Result:** All text and elements display high contrast for readability.
- **How test will be performed:** Perform a manual inspection of UI contrast using WCAG contrast standards.

#### 4.2.3 APR3 - Intuitive Navigation

- **Test ID:** TAPR3
- **Type:** Dynamic, Manual
- **Initial State:** System interface opened.
- **Input/Condition:** Navigate through different pages.
- **Output/Result:** Users can easily navigate and locate functions within 3 clicks.
- **How test will be performed:** Manually navigate to different features and confirm efficient accessibility.

#### 4.2.4 STR1 - Consistent Button Styles

- **Test ID:** TSTR1
- **Type:** Static, Manual
- **Initial State:** Interface loaded.
- **Input/Condition:** Check for style consistency in buttons.
- **Output/Result:** All buttons follow a consistent color and shape style.

- **How test will be performed:** Visually inspect all buttons to ensure they meet the style guidelines.

#### 4.2.5 EUR1 - Tooltip Visibility

- **Test ID:** TEUR1
- **Type:** Dynamic, Manual
- **Initial State:** Interface loaded with tooltips.
- **Input/Condition:** Hover over interactive elements.
- **Output/Result:** Tooltips display with descriptive content.
- **How test will be performed:** Manually hover over elements to confirm tooltip visibility.

#### 4.2.6 PIR1 - Customizable Color Themes

- **Test ID:** TPIR1
- **Type:** Dynamic, Manual
- **Initial State:** Interface with theme options.
- **Input/Condition:** User switches between themes.
- **Output/Result:** All themes display correctly with no visual errors.
- **How test will be performed:** Switch themes manually and verify consistent color scheme application.

#### 4.2.7 LR1 - Initial Tutorial

- **Test ID:** TLR1
- **Type:** Dynamic, Manual
- **Initial State:** First-time user experience loaded.
- **Input/Condition:** User accesses the system for the first time.

- **Output/Result:** System displays an introductory tutorial.
- **How test will be performed:** Manually confirm tutorial launches on initial access.

#### 4.2.8 LR2 - Tutorial Completion Time

- **Test ID:** TLR2
- **Type:** Dynamic, Manual
- **Initial State:** Tutorial in progress.
- **Input/Condition:** Measure time for tutorial completion.
- **Output/Result:** Users complete the tutorial within 5 minutes.
- **How test will be performed:** Track completion time for new users.

#### 4.2.9 UPR1 - Friendly Feedback

- **Test ID:** TUPR1
- **Type:** Static, Manual
- **Initial State:** Error states are accessible.
- **Input/Condition:** Trigger common errors.
- **Output/Result:** System provides friendly, clear feedback.
- **How test will be performed:** Manually review error messages to confirm they are polite and helpful.

#### 4.2.10 UPR2 - Standardized Iconography

- **Test ID:** TUPR2
- **Type:** Static, Manual
- **Initial State:** A subset of complete interface iconography is available.

- **Input/Condition:** Developer reviews subset of complete interface iconography.
- **Output/Result:** Subset of complete interface iconography is deemed standard and inoffensive.
- **How test will be performed:** Manually review individual icons to confirm they are standard and inoffensive.

#### 4.2.11 ACR1 - Accessible Fonts

- **Test ID:** TACR1
- **Type:** Static, Manual
- **Initial State:** Font-defining code is available.
- **Input/Condition:** Developer reviews font-defining code.
- **Output/Result:** Font-defining code is deemed to only contain accessible fonts in accordance to the WCAG 2.2 standard.
- **How test will be performed:** Perform a manual inspection of fonts available in the UI and cross-check them with the WCAG 2.2 standard.

#### 4.2.12 ACR2 - Color Blind Mode

- **Test ID:** TACR2
- **Type:** Dynamic, Manual
- **Initial State:** Interface set to default visibility mode.
- **Input/Condition:** Enable color blind mode.
- **Output/Result:** Interface set to color blind visibility mode.
- **How test will be performed:** Use a web driver like `Selenium` to perform input and record output.

#### 4.2.13 PAR1 - Query Request Time Precision

- **Test ID:** TPAR1
- **Type:** Dynamic, Automated
- **Initial State:** Server active and client application open.
- **Input/Condition:** Issue any valid query.
- **Output/Result:** Server returns response and client application displays query request time.
- **How test will be performed:** Use a web driver like `Selenium` to perform input and record output, capturing displayed time for verification.

#### 4.2.14 PAR2 - Rounding Accuracy

- **Test ID:** TPAR2
- **Type:** Dynamic, Automated
- **Initial State:** System loaded with rounding functions.
- **Input/Condition:** Input values with decimals.
- **Output/Result:** Values are rounded accurately according to specification.
- **How test will be performed:** Perform automated tests on rounding functions with pre-defined decimal values.

#### 4.2.15 RAR1 - Fault Tolerance

- **Test ID:** TRAR1
- **Type:** Dynamic, Manual
- **Initial State:** Server operational with any state or load, e.g., idle, under little load (2 or fewer users interleaving requests less than once every 30 minutes), under intermediate use (2 or more users interleaving requests at least once every 30 minutes), or under strenuous use (4 or more users interleaving requests at least once every 5 minutes).

- **Input/Condition:** Server operational under any sequence of state transitions, e.g., from any of idle, under little, intermediate, or strenuous load, to idle, under little, intermediate, or strenuous load.
- **Output/Result:** Server operational with any state or load.
- **How test will be performed:** Schedule a 3-day monitoring period and use Ubuntu Server's `uptime` command to find the uptime across the monitoring period, allowing assessment and extrapolation of results from 3 days to 30 days. If possible, repeat the test with a 30-day (or longer) monitoring period. *Note*: Precise uptime metrics can only be achieved by formal checking which we do not have the expertise, time, or necessity for. A simple extrapolation result suffices for the given scope.

#### 4.2.16 CR1 - Minimum Concurrent Users Load

- **Test ID:** TCR1
- **Type:** Dynamic, Load Test
- **Initial State:** System idle.
- **Input/Condition:** Simulate multiple user logins.
- **Output/Result:** System handles expected concurrent user limit.
- **How test will be performed:** Use a load testing tool to simulate concurrent users.

#### 4.2.17 CR2 - Data Storage

- **Test ID:** TCR2
- **Type:** Dynamic, automated
- **Initial State:** Mock database active.
- **Input/Condition:** Issue any valid query.
- **Output/Result:** Database stores appropriate data related to songs and query information.

- **How test will be performed:** Use frameworks like `factoryboy` and `Pytest`'s fixtures to mock a database and test storage functionality.

#### 4.2.18 EPER1 - Server Device

- **Test ID:** TEPER1
- **Type:** Environment, Inspection
- **Initial State:** Server is available for inspection.
- **Input/Condition:** Inspect server specifications and body.
- **Output/Result:** Server is of make and model Dell OptiPlex 3050.
- **How test will be performed:** View server using los ojos.

### 4.3 Productization Requirements

#### 4.3.1 PRR1 - Production Readiness Verification

- **Test ID:** TPRR1
- **Type:** Dynamic, Manual
- **Initial State:** System deployed in staging environment.
- **Input/Condition:** Perform a complete end-to-end run.
- **Output/Result:** System operates without failure in a production-like setting.
- **How test will be performed:** Conduct end-to-end test in staging to verify production readiness.

#### 4.3.2 MR1 - Ease of Code Updates

- **Test ID:** TMR1
- **Type:** Static, Manual
- **Initial State:** Source code repository active.

- **Input/Condition:** Review update process.
- **Output/Result:** Codebase is structured for easy updates.
- **How test will be performed:** Review code structure and modularity to ensure maintainability.

#### 4.3.3 ACCR1 - Licensed Song Access

- **Test ID:** TMR1
- **Type:** Dynamic, Automated
- **Initial State:** Mock database active with multiple queries from different (at least 2 unique) users already loaded.
- **Input/Condition:** Issue queries to access songs requested other users, not the current user.
- **Output/Result:** Database returns an empty response because the requesting user does not have access (or a license) to the songs uploaded/licensed by the other user(s).
- **How test will be performed:** Use frameworks like `factoryboy` and `Pytest`'s fixtures to mock a database and entries, then issue queries, capture response, and ensure it is empty.

#### 4.3.4 IR1 - Database Backup

- **Test ID:** TIR1
- **Type:** Static & Manual, Dynamic & Automated
- **Initial State:** Database layout configured & backup code is complete.
- **Input/Condition:** Conduct code review to ensure backup functionality is correct, and simulate it running (on-command as opposed to weekly) to ensure it does backup.
- **Output/Result:** Database backup functionality is found to be correct, with ad-hoc generated backup artifacts to check it live.

- **How test will be performed:** Conduct a code review/walkthrough and use both unit and integration testing through `Pytest` with fixtures (alongside `factoryboy`) to ensure the correct backup artifacts are generated.

#### 4.3.5 IR2 - Database Deduplication

- **Test ID:** TIR2
- **Type:** Static & Manual, Dynamic & Automated
- **Initial State:** Database layout configured & deduplication code is complete.
- **Input/Condition:** Conduct code review to ensure deduplication functionality is correct, and insert duplicate records to ensure the mechanism prevents their insertion or "fails loudly".
- **Output/Result:** Database deduplication functionality is found to be correct, with induced duplicate insertions prevented.
- **How test will be performed:** Conduct a code review/walkthrough and use both unit and integration testing through `Pytest` with fixtures (alongside `factoryboy`) to ensure the correct deduplication behaviour is encountered.

#### 4.3.6 PR1 - Data Encryption Verification

- **Test ID:** TPR1
- **Type:** Static, Manual
- **Initial State:** Database system in use.
- **Input/Condition:** Inspect database for encryption protocols.
- **Output/Result:** All sensitive data is encrypted in storage.
- **How test will be performed:** Review encryption settings in database configuration.

#### 4.3.7 AUR1 - Access Logs for User Sessions

- **Test ID:** TAUR1
- **Type:** Dynamic, Manual
- **Initial State:** System active with user sessions.
- **Input/Condition:** Access user session logs.
- **Output/Result:** Logs capture all user activities accurately.
- **How test will be performed:** Review session logs for accuracy and completeness.

#### 4.3.8 CUR1 - Multilingual Support

- **Test ID:** TCUR1
- **Type:** Dynamic, Manual
- **Initial State:** System interface displayed.
- **Input/Condition:** Switch to different language options.
- **Output/Result:** System adapts to selected language without errors.
- **How test will be performed:** Switch languages manually and verify accurate translations.

#### 4.3.9 LGR1 - Compliance with Copyright Laws

- **Test ID:** TLGR1
- **Type:** Static, Manual
- **Initial State:** Content library loaded.
- **Input/Condition:** Inspect all music content for licensing.
- **Output/Result:** All content has proper copyright attributions.
- **How test will be performed:** Check each music file and source for copyright compliance.

#### 4.3.10 LGR2 - Adherence to Data Protection Regulations

- **Test ID:** TLGR2
- **Type:** Static, Manual
- **Initial State:** User data system in place.
- **Input/Condition:** Inspect data management policies.
- **Output/Result:** User data management meets legal requirements.
- **How test will be performed:** Review data management practices to confirm legal compliance.

## 4.4 Traceability Between Test Cases and Requirements

| | FR1 | FR2 | FR3 | FR4 | FR5 | FR6 | FR7 | FR8 | FR9 |
|---|---|---|---|---|---|---|---|---|---|
| UII-01 | X | | | | | | | | |
| CDH-01 | | X | | | | | | | |
| FE-01 | | | X | | | | | | |
| EADR-01 | | | | X | | | | | |
| AFD-01 | | | | | X | | X | | |
| IVED-01 | | | | | | X | | X | |
| EF-01 | | | | | | X | | | |
| OD-01 | | | | | | | X | | |
| UII-02 | | | | | | | | | X |

Table 1: Traceability Matrix Showing the Connections Between the Tests and Functional Requirements

| | APR1 | APR2 | APR3 | STR1 | EUR1 | PIR1 | LR1 | LR2 | UPR1 | UPR2 | ACR1 | ACR2 | PAR1 | PAR2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| APR1 | X | | | | | | | | | | | | | |
| APR2 | | X | | | | | | | | | | | | |
| APR3 | | | X | | | | | | | | | | | |
| STR1 | | | | X | | | | | | | | | | |
| EUR1 | | | | | X | | | | | | | | | |
| PIR1 | | | | | | X | | | | | | | | |
| LR1 | | | | | | | X | | | | | | | |
| LR2 | | | | | | | | X | | | | | | |
| UPR1 | | | | | | | | | X | | | | | |
| UPR2 | | | | | | | | | | X | | | | |
| ACR1 | | | | | | | | | | | X | | | |
| ACR2 | | | | | | | | | | | | X | | |
| PAR1 | | | | | | | | | | | | | X | |
| PAR2 | | | | | | | | | | | | | | X |

Table 2: Traceability Matrix Showing the Connections Between the Tests and Non-Functional Requirements 1-14

| | RAR1 | CR1 | CR2 | EPER1 | PRR1 | MR1 | ACCR1 | IR1 | IR2 | PR1 | AUR1 | CUR1 | LGR1 | LGR2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RAR1 | X | | | | | | | | | | | | | |
| CR1 | | X | | | | | | | | | | | | |
| CR2 | | | X | | | | | | | | | | | |
| EPER1 | | | | X | | | | | | | | | | |
| PRR1 | | | | | X | | | | | | | | | |
| MR1 | | | | | | X | | | | | | | | |
| ACCR1 | | | | | | | X | | | | | | | |
| IR1 | | | | | | | | X | | | | | | |
| IR2 | | | | | | | | | X | | | | | |
| PR1 | | | | | | | | | | X | | | | |
| AUR1 | | | | | | | | | | | X | | | |
| CUR1 | | | | | | | | | | | | X | | |
| LGR1 | | | | | | | | | | | | | X | |
| LGR2 | | | | | | | | | | | | | | X |

Table 3: Traceability Matrix Showing the Connections Between the Tests and Non-Functional Requirements 15-28

# 5 Unit Test Description

## 5.1 Unit Testing Scope

## 5.2 Tests for Functional Requirements

### 5.2.1 Module 1

1. test-id1

   Type:
   Initial State:
   Input:
   Output:
   Test Case Derivation:
   How test will be performed:

2. test-id2

   Type:
   Initial State:
   Input:
   Output:
   Test Case Derivation:
   How test will be performed:

3. ...

### 5.2.2 Module 2

...

## 5.3 Tests for Nonfunctional Requirements

### 5.3.1 Module ?

1. test-id1

   Type:

   Initial State:

   Input/Condition:

   Output/Result:

   How test will be performed:

2. test-id2

   Type: Functional, Dynamic, Manual, Static etc.

   Initial State:

   Input:

   Output:

   How test will be performed:

### 5.3.2 Module ?

...

## 5.4 Traceability Between Test Cases and Modules

# References

Author Author. System requirements specification. https://github.com/..., 2019.

# 6 Appendix

## 6.1 Symbolic Parameters

No symbolic constants were used in defining the test cases.

## 6.2 Usability Survey Questions

To validate the usability requirements outlined in the SRS, the following survey questions have been structured as 'check criteria', allowing for quick evaluation with follow-up prompts to gather additional details if necessary.

- **Tooltip Effectiveness (EUR1)**:
  - *Survey Question:* "Were the tooltips helpful in understanding the function of each feature? (Yes/No)
    - If No, please specify which tooltips were unclear or missing."
- **Ease of Personalization (PIR1)**:
  - *Survey Question:* "Were you able to find and switch between different display color themes easily? (Yes/No)
    - If No, please describe any difficulties you encountered or suggestions for improvement."
- **First-Time User Guide Clarity and Timing (LR1, LR2)**:
  - *Survey Question:* "Did the first-time user guide help you understand the main features within 10 minutes? (Yes/No)
    - If No, please indicate what aspects were unclear or took longer than expected."
- **Content Politeness and Appropriateness (UPR1, UPR2)**:
  - *Survey Question:* "Was all content and iconography appropriate and free from offensive material? (Yes/No)
    - If No, please specify any instances where content was found inappropriate."
- **Accessibility of Font and Color Options (ACR1, ACR2)**:

- *Survey Question:* "Did you find the font choices easy to read, and was the color-blind mode accessible if you used it? (Yes/No)
    - If No, please describe any accessibility challenges you faced with font readability or color options."

These questions are designed to verify specific usability requirements established in the SRS.

## 6.3 Code Walkthrough Checklist

Here is a sample code walkthrough checklist. It will be updated as reviews are performed:

Functionality

- ☐ Does the code perform its intended task?
- ☐ Can this code be traced to a requirement?
- ☐ Is there redundant or unnecessary code?

Readability

- ☐ Do the functions and variables have meaningful names?
- ☐ Does the source code contain sufficient commenting?
- ☐ Does the code follow PEP8 style guidelines

Modularity

- ☐ Are there excessively long functions that can be broken down?
- ☐ Is the code organized?
- ☐ Is the code easy to change or expand upon?

Error Handling

- ☐ Are exceptions and errors handled gracefully?
- ☐ Do exceptions and erros have useful error messages?

Testing

- ☐ Are there enough unit tests such that the code coverage tool reports 100% code coverage?
- ☐ Does `Pytest` report that 100% of unit tests are passing?
- ☐ Are there enough system tests to cover all requirements?
- ☐ Are all system tests passing?

Reliability

- ☐ Is the code fault-tolerant? I.e., does the system continue to operate despite failures/faults?
- ☐ Does the code have effective exception-handling and error recovery mechanisms?

Efficiency

- ☐ How much memory or processor capacity does the program consume?
- ☐ Are algorithms optimized, avoiding unnecessary operations?

Reusability

- ☐ Can components be reused in other applications/other parts of the application?
- ☐ Does the program have a well-partitioned, modular design with *strong cohesion* and *loose coupling*?

Scalability

- ☐ Can the system grow to accomodate more users, servers, data or other components?
- ☐ Can it do so with acceptable performance and at acceptable cost?

# Appendix — Reflection

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing "what you think the evaluator wants to hear."

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

1. What went well while writing this deliverable? We benefited from refining the scope of our Functional Requirements (FRs) in the SRS document, allowing us to focus on relevant test cases. This saved significant time and helped avoid creating test cases for requirements outside of the project's intended scope.

2. What pain points did you experience during this deliverable, and how did you resolve them?
   A major challenge was maintaining an abstract perspective when defining test cases for requirements. It was tempting to delve into detailed unit test cases rather than focusing on verification at a broader level. To resolve this, we regularly reviewed each section and discussed ways to keep our approach consistent with high-level testing.

3. What knowledge and skills will the team collectively need to acquire to successfully complete the verification and validation of your project? Examples of possible knowledge and skills include dynamic testing knowledge, static testing knowledge, specific tool usage, Valgrind etc. You should look to identify at least one item for each team member.
   Muhammad Jawad: To effectively conduct verification and validation, the team needs to develop a solid understanding of dynamic testing techniques and best practices.
   Mohamad-Hassan Bahsoun: To successfully complete the verification

and validation for GenreGuru, I will need to gain expertise in API testing to ensure that data exchanges between the application and external services are accurate, secure, and reliable.
Matthew: To conduct effective testing through static analysis methods, the team will also need to develop a strong understanding of static testing techniques and best practices
Ansel: Ansel will have to learn how to unit testing in order to properly apply the tests that were drafted up in the VnV plan. *Ahmed*: I am fairly acquainted with most technology involved in the verification process, but will need to research and draft templates for systematic validation processes. It is imperative to make validation processes very informaion-dense without overwhelming stakeholders, but certainly not unstructured and unprepared, wasting the stakeholder's time. I cross-referenced the courses *SFWRENG 3RA3* and *SFWRENG 3S03*, and will continue to do so in the future.

4. For each of the knowledge areas and skills identified in the previous question, what are at least two approaches to acquiring the knowledge or mastering the skill? Of the identified approaches, which will each team member pursue, and why did they make this choice?
   To gain expertise in dynamic testing, Jawad will:

   - Perform practice-based learning through testing frameworks such as 'pytest'
   - Watch Youtube tutorials on dynamic testing tools and practices

   This will help ensure basic foundational understanding of dynamic testing techniques.
   Matthew will:

   - Practice integrating `Ruff` and `Mypy` into toy projects
   - Watch videos and reading documentation on best practices for static analysis tools

   Matthew has minimal experience in static analysis tools, so it would be worthwhile to become proficient.
   Two approaches for acquiring API testing knowledge include completing online courses or tutorials that cover key API testing principles, or peer learning (e.g. learning from a classmate). Mohamad-Hassan

Bahsoun has chosen to pursue online tutorials to build foundational skills in a structured, and paced format. This is because this will allow him to be flexible while ensuring he covers essential concepts required for the project's testing needs.
As stated in section 3.1, Ansel is expected to test the featurizer and client application. For the client application testing, it is expected that Ansel will apply the knowledge from SFWR 4HC3: Human computer interfaces in order to help test and design a client application that is satisfactory. For the featurizer testing, it is expected that Ansel will learn about how to featurize songs and then process that data, which can include reading papers or asking Dr. MvM to learn. *Ahmed*: Look into user validation techniques and researcher data collection practices when collection project validation data; From experience, I recognize that some validation will result in conflicts or disagreements across developers and stakeholders, stakeholders and stakeholders, or developers amongst themselves - a crucial technique to refer back to is professional negotiation through discussion; continue to cross-reference *SFWRENG 3RA3* and *SFWRENG 3S03* and their relevant resources, e.g., and Ammann & Offutt's "Introduction to Software Testing".