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 4, 2024

Revision History

Date	Version	Notes
Date 1	1.0	Notes
Date 2	1.1	Notes

[The intention of the VnV plan is to increase confidence in the software. However, this does not mean listing every verification and validation technique that has ever been devised. The VnV plan should also be a **feasible** plan. Execution of the plan should be possible with the time and team available. If the full plan cannot be completed during the time available, it can either be modified to "fake it", or a better solution is to add a section describing what work has been completed and what work is still planned for the future. —SS]

[The VnV plan is typically started after the requirements stage, but before the design stage. This means that the sections related to unit testing cannot initially be completed. The sections will be filled in after the design stage is complete. the final version of the VnV plan should have all sections filled in. —SS]

Contents

List of Tables

[Remove this section if it isn't needed —SS]

List of Figures

[Remove this section if it isn't needed —SS]

1 Symbols, Abbreviations, and Acronyms

symbol	description
T	Test

[symbols, abbreviations, or acronyms — you can simply reference the SRS (Author, 2019) tables, if appropriate —SS]

[Remove this section if it isn't needed —SS]

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

[Say what software is being tested. Give its name and a brief overview of its general functions. —SS]

2.2 Objectives

[State what is intended to be accomplished. The objective will be around the qualities that are most important for your project. You might have something like: "build confidence in the software correctness," "demonstrate adequate usability." etc. You won't list all of the qualities, just those that are most important. —SS]

[You should also list the objectives that are out of scope. You don't have the resources to do everything, so what will you be leaving out. For instance, if you are not going to verify the quality of usability, state this. It is also worthwhile to justify why the objectives are left out. —SS]

[The objectives are important because they highlight that you are aware of limitations in your resources for verification and validation. You can't do everything, so what are you going to prioritize? As an example, if your system depends on an external library, you can explicitly state that you will assume that external library has already been verified by its implementation team. —SS]

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 probelm 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

[Reference relevant documentation. This will definitely include your SRS and your other project documents (design documents, like MG, MIS, etc). You can include these even before they are written, since by the time the project is done, they will be written. You can create BibTeX entries for your documents and within those entries include a hyperlink to the documents. —SS]

Author (2019)

[Don't just list the other documents. You should explain why they are relevant and how they relate to your VnV efforts. —SS]

3 Plan

[Introduce this section. You can provide a roadmap of the sections to come. —SS]

3.1 Verification and Validation Team

[Your teammates. Maybe your supervisor. You should do more than list names. You should say what each person's role is for the project's verification. A table is a good way to summarize this information. —SS]

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 verification and validation plan is an artifact that should also be verified. Techniques for this include review and mutation testing. —SS]

[The review will include reviews by your classmates —SS]

[Create a checklists? —SS]

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

[What tools are you using for automated testing. Likely a unit testing framework and maybe a profiling tool, like ValGrind. Other possible tools include a static analyzer, make, continuous integration tools, test coverage tools, etc. Explain your plans for summarizing code coverage metrics. Linters are another important class of tools. For the programming language you select, you should look at the available linters. There may also be tools that verify that coding standards have been respected, like flake9 for Python. —SS]

[If you have already done this in the development plan, you can point to that document. —SS]

[The details of this section will likely evolve as you get closer to the implementation. —SS]

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 Demo

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

[There should be text between all headings, even if it is just a roadmap of the contents of the subsections. —SS]

4.1 Tests for Functional Requirements

[Subsets of the tests may be in related, so this section is divided into different areas. If there are no identifiable subsets for the tests, this level of document structure can be removed. —SS]

[Include a blurb here to explain why the subsections below cover the requirements. References to the SRS would be good here. —SS]

4.1.1 Area of Testing1

[It would be nice to have a blurb here to explain why the subsections below cover the requirements. References to the SRS would be good here. If a section covers tests for input constraints, you should reference the data constraints table in the SRS. —SS]

Title for Test

1. test-id1

   Control: Manual versus Automatic

   Initial State:

   Input:

   Output: [The expected result for the given inputs. Output is not how you are going to return the results of the test. The output is the expected result. —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

2. test-id2

Control: Manual versus Automatic

Initial State:

Input:

Output: [The expected result for the given inputs —SS]

Test Case Derivation: [Justify the expected value given in the Output field —SS]

How test will be performed:

4.1.2 Area of Testing2

...

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

[Provide a table that shows which test cases are supporting which requirements. —SS]

5 Unit Test Description

[This section should not be filled in until after the MIS (detailed design document) has been completed. —SS]

[Reference your MIS (detailed design document) and explain your overall philosophy for test case selection. —SS]

[To save space and time, it may be an option to provide less detail in this section. For the unit tests you can potentially layout your testing strategy here. That is, you can explain how tests will be selected for each module. For instance, your test building approach could be test cases for each access program, including one test for normal behaviour and as many tests as needed for edge cases. Rather than create the details of the input and output here, you could point to the unit testing code. For this to work, you code needs to be well-documented, with meaningful names for all of the tests. —SS]

5.1 Unit Testing Scope

[What modules are outside of the scope. If there are modules that are developed by someone else, then you would say here if you aren't planning on

verifying them. There may also be modules that are part of your software, but have a lower priority for verification than others. If this is the case, explain your rationale for the ranking of module importance. —SS]

5.2 Tests for Functional Requirements

[Most of the verification will be through automated unit testing. If appropriate specific modules can be verified by a non-testing based technique. That can also be documented in this section. —SS]

5.2.1 Module 1

[Include a blurb here to explain why the subsections below cover the module. References to the MIS would be good. You will want tests from a black box perspective and from a white box perspective. Explain to the reader how the tests were selected. —SS]

1. test-id1

   Type: [Functional, Dynamic, Manual, Automatic, Static etc. Most will be automatic —SS]

   Initial State:

   Input:

   Output: [The expected result for the given inputs —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

2. test-id2

   Type: [Functional, Dynamic, Manual, Automatic, Static etc. Most will be automatic —SS]

   Initial State:

   Input:

   Output: [The expected result for the given inputs —SS]

Test Case Derivation: [Justify the expected value given in the Output field —SS]

How test will be performed:

3. ...

5.2.2 Module 2

...

5.3 Tests for Nonfunctional Requirements

[If there is a module that needs to be independently assessed for performance, those test cases can go here. In some projects, planning for nonfunctional tests of units will not be that relevant. —SS]

[These tests may involve collecting performance data from previously mentioned functional tests. —SS]

5.3.1 Module ?

1. test-id1

   Type: [Functional, Dynamic, Manual, Automatic, Static etc. Most will be automatic —SS]

   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

[Provide evidence that all of the modules have been considered. —SS]

References

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

6 Appendix

This is where you can place additional information.

6.1 Symbolic Parameters

The definition of the test cases will call for SYMBOLIC_CONSTANTS. Their values are defined in this section for easy maintenance.

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

[This section is not required for CAS 741 —SS]

The information in this section will be used to evaluate the team members on the graduate attribute of Lifelong Learning.

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?

2. What pain points did you experience during this deliverable, and how did you resolve them?

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.

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?