

D3: Test Plan

Part 1 (What we would do with infinite time)

Unit Testing

- Maintain 100% test coverage for all backend components using cargo test
- Maintain 100% test coverage for all frontend functions using vitest
- Exhaustive tests for valid and invalid inputs as well as edge cases
- Run tests locally before each commit and set up Github actions to run tests when a PR is created

Integration and System Testing (with UI)

- Test that users can successfully upload an audio file of various types to the application (mp3, flac, etc..)
- Test that the audio quality produced by the application during playback matches other audio players and that the audio is not distorted when the playback speed is modified
- Test that users can add markers on top of the audio file and create segments within the application
- Test that all shortcuts for functionality within the application function properly and interact with the rust portion of the application as expected
- Test that the user is able to successfully export audio snippets and CSV files regardless of the input audio file type
- Test that the bundled FFmpeg binary successfully runs regardless of the end users machine or operating system

Tools Used

- Cargo Test: Used to test the backend logic for the application that is written in rust. Tests written with cargo test are written within the same file as the logic that they are testing
- Vitest: Used to test the functions written on the frontend that interact with the IPC commands that call the Rust backend functions. This requires creating mocks of the exposed Tauri IPC commands.

Types of End Users

The tests written will reflect the three primary concerns of our client which are

- To create audio snippets from a monolithic audio file recording of a live concert, with the audio possibly containing a large amount of crowd noise
- To create audio snippets from an educational talk at a conference where different portions of the talk address different topics

Performance/Reliability

- Test that files of larger sizes (greater than 2 hours long) can be successfully loaded, played/marked up, and processed into segments

Part 2 (What we are actually testing)

Unit Testing

For both the Typescript (frontend) and Rust (backend) portions of our codebase, we will aim for 85% unit test coverage (measured in lines). This allows us to cover the vast majority of our code, but leaves some wiggle room for the portion of our code that is registering IPC calls with Tauri and interacting with the device hardware (for things like playing music and writing to files). The remaining portion of the code will be further exercised by Our Integration and System Testing.

The Typescript tests will cover all of the frontend business logic (for example, the callbacks that are called when a given button is clicked). These tests are written using vitest. The Rust tests will cover all of the backend functionality for decoding audio, interacting with the ring buffer, marking segments, and calling into ffmpeg to generate audio snippets. These tests will be written using the test functionality built in with rust and cargo.

Integration and System Testing (Manual)

For each mp3, flac, and m4a, we upload an audio file of that type to the application to ensure that it can be loaded and played properly. Next we test out each of the shortcuts provided by the application to ensure that they function properly. Next, we add markers for segment starts, start/ends, and ends throughout the input file. When every start has a valid end marker the user should be able to press the export segments button in the top right corner of the screen and choose a path at which to save the audio files and the CSV.

Tools Used

For our typescript unit testing, we use vitest and Tuari IPC Mocks. This allows us to test the frontend functionality in isolation. Vitest also gives us the ability to generate a test coverage report directly out of the box.

For the Rust unit test, we use the built in cargo test functionality for writing the unit tests and the llvm-cov crate for generating coverage reports.

How to Run our Tests

Typescript Tests:

npm test

Typescript unit test coverage:

npm run coverage

Rust tests:

```
cd src-tauri  
cargo test
```

Rust test coverage (using llvm-cov). If you don't already have llvm-cov installed, you will need to install it by following the instructions in [this README](#)

```
cd src-tauri (if you haven't already done this by running the tests previously)  
cargo coverage
```

Acceptance Testing

There are three primary types of users that will test our application to verify it meets acceptance criteria: Developers, Clients (users), and Managers (Professor/TA).

For Developers: Automated unit tests on the frontend (Typescript) and backend (Rust) in addition to manual Integration/System testing is to be executed every PR to ensure core functionality and additional features produce expected behavior.

For Clients (users):

For all three types of users, the conditions which acceptance is dependent on are uniform and must be met:

- Users can successfully load a media file of the form .mp3, .flac, .wav from their local machine
- Users can play, pause, speed up/slow down playback, jump to specific timestamps using provided keyboard shortcuts
- Users can successfully place start, end and start/end markers with automatic validation using keyboard shortcuts
- Users can refer to the rendered waveform of the audio to review current playback position and location of markers placed
- Users can export segmented audio files based on their markers, where the contents of each audio file corresponds exactly to the content in the original file within the specified start and end timestamps