

D2: Design Document

Project Overview and Constraints

The AudioSegment Marker Tool is a macOS desktop application for reviewing audio and audio/video media files, placing segment markers by listening, and exporting a CSV file for use with the client's existing ffmpeg pipeline. The tool is strictly read-only with respect to the input file. All markup is driven by audio content only.

The following constraints apply to all stack decisions:

- macOS is the required deployment target
- The tool must not modify the original media file
- CSV output must follow the client's ffmpeg pipeline format: timestamps in HH:MM:SS.mmm
- Keyboard-first interaction — most actions must be achievable with a single keypress
- The UI has been prototyped in Figma — the stack must support faithful implementation of a custom interface
- Lightweight and responsive is a hard requirement — play/pause and marker placement must feel instantaneous
- Frameworks and libraries should do as much heavy lifting as possible to reduce custom engineering

Stack 1: Tauri (chosen)

Technologies

- **Languages:** TypeScript for the UI and frontend logic; Rust for the audio engine and file I/O
 - Developers write two separate codebases that talk to each other over IPC — more complex to debug than a single-language app
- **Application Shell:** Tauri v2
 - Manages the native window, file dialogs, and message passing between the Rust backend and the web frontend
 - Replaces both Electron's main process and its bundled Chromium engine
- **Frontend Framework:** Svelte with TypeScript
 - Standard web component development
 - Figma designs translate directly to HTML and CSS with no widget toolkit constraints
- **Waveform Rendering:** WaveSurfer.js 7.x with Regions plugin

- Runs in the web frontend — provides scrolling waveform, playhead, zoom, and region/marker management out of the box
- The Regions plugin directly models start and end segment markers, which maps cleanly to the spec
- Developers get most of the waveform UI for free without writing custom rendering code
- **Audio Playback and Decoding:** cpal crate + Symphonia (Rust)
 - cpal interfaces with CoreAudio on macOS for sample-accurate, low-latency output
 - Symphonia decodes MP3, MP4, WAV, FLAC, OGG, and AAC natively in Rust — no ffmpeg required at runtime
 - Audio runs entirely in Rust, separate from the web frontend, so UI rendering never blocks playback
- **Speed and Pitch Control:** rubato crate (Rust)
 - Provides high-quality pitch-preserving time-stretch for refinement mode
 - Runs in the Rust backend — results are sent to the frontend as processed audio chunks
- **Keybinding:** Tauri global shortcut plugin and frontend keydown listeners
 - System-wide shortcuts registered through Tauri; in-window shortcuts handled in TypeScript
- **CSV Export:** Rust csv crate
 - RFC 4180 compliant, fast, and requires no configuration — just format the struct and write
- **Packaging and Deployment:** Tauri bundler → macOS .app and .dmg
 - Supports Apple code signing and notarization
 - Final bundle is 5 to 20 MB — the smallest of all options — because no browser engine is bundled

Pros

- Smallest bundle size of all options (5 to 20 MB) because Tauri uses macOS's built-in WebView instead of bundling Chromium
- Audio runs in Rust with CoreAudio — garbage collection never interrupts playback, giving the most consistent low-latency behavior
- WaveSurfer.js handles nearly the entire waveform UI including regions, zoom, and looping — developers write very little custom rendering code
- cpal with CoreAudio achieves the lowest possible audio output latency on macOS, which is critical for the spec's requirement that marker placement feel instantaneous
- Symphonia provides broad format support without requiring ffmpeg as a runtime dependency
- The HTML and CSS frontend means Figma designs translate directly to code with no widget toolkit limitations
- TypeScript provides compile-time type safety across all frontend logic
- Very small final application — fast to install and launch

Cons

- Two separate languages mean two separate mental models — the team must context-switch between Rust's ownership system and TypeScript's async patterns in the same session

- Debugging requires two separate toolchains — Rust uses cargo and lldb while TypeScript uses browser DevTools — tracing a bug that crosses the IPC boundary is harder than debugging a single-language app
- Tauri is significantly less mature than Electron — there are fewer community examples of audio tools built with it, and StackOverflow coverage is thin
- WKWebView's Web Audio API support is less complete than Chromium's — some audio scheduling features available in Electron are not available here
- WaveSurfer.js running inside WKWebView rather than Chromium should be tested early to confirm rendering performance on the target machine

Stack 2: Python + PyQt6

A fully Python-based desktop application using PyQt6 as the GUI framework. PyQt6 provides Python bindings for Qt6 — a mature, cross-platform native widget toolkit with solid macOS integration. Audio playback is handled by sounddevice, a Python binding for PortAudio, which communicates with CoreAudio on macOS. Waveform rendering is done by PyQtGraph, an OpenGL-accelerated graphics library that lives inside the Qt widget tree. All format conversion goes through ffmpeg-python, which shells out to ffmpeg for any input that soundfile cannot read natively. The entire codebase is one language, which makes this the most accessible stack.

Technologies

- **Language:** Python 3.11+ throughout — no secondary language required
- **GUI Framework:** PyQt6
 - Provides native macOS window chrome, menus, file dialogs, and a complete widget toolkit
 - Qt's widget model is well-documented with many examples of audio applications on GitHub
 - Custom UI designs from Figma must be re-implemented using Qt widgets — some design details may require compromise compared to raw HTML and CSS
- **Waveform Rendering:** PyQtGraph
 - OpenGL-accelerated plotting library embedded as a Qt widget — no separate rendering loop to manage
 - Capable of 60fps scrolling waveform with a moving playhead
 - Marker and region overlays require more custom code than WaveSurfer.js — developers write the marker drawing logic themselves
- **Audio Playback:** sounddevice + soundfile
 - sounddevice wraps PortAudio and talks to CoreAudio on macOS — provides sub-10ms play/pause latency
 - Precise current playback position is queryable from the audio callback thread
 - Audio I/O must run in a background thread to avoid blocking the Qt event loop — requires Queue-based communication between threads
- **Audio Decoding and Format Support:** soundfile + ffmpeg-python
 - soundfile handles WAV and FLAC natively

- ffmpeg-python shells out to ffmpeg for MP3, MP4, AAC, and any other format — the client already has ffmpeg installed, so this is zero additional dependency for the end user
- **Speed and Pitch Control:** librosa (phase vocoder)
 - Pitch-preserving time-stretch is available but must be pre-computed offline — librosa processes the audio segment and writes a new buffer before playback begins
 - This means entering refinement mode for the first time on a new marker has a brief processing delay
- **Keybinding:** PyQt6 QShortcut and keyPressEvent
 - Single-keypress actions are wired directly in the Qt event loop with minimal boilerplate
- **CSV Export:** Python standard library csv module
 - Zero external dependency — built into Python, RFC 4180 compliant
- **Packaging and Deployment:** PyInstaller → macOS .app
 - Bundles the Python runtime and all dependencies into a self-contained app — no Python required on the target machine
 - Bundle size is typically 80 to 180 MB due to numpy and librosa dependency chains

Pros

- Single language throughout — every developer can read, write, and debug every part of the codebase without switching mental models
- Lowest team ramp-up cost of all five options — Python syntax and ecosystem are already familiar
- PyQtGraph provides production-quality real-time scrolling waveform rendering comparable to WaveSurfer.js in visual quality
- sounddevice with CoreAudio delivers sub-10ms play/pause response, which is sufficient for the spec's requirement that playback controls feel instantaneous
- ffmpeg-python makes format support effectively unlimited and directly reuses the client's existing toolchain
- QShortcut makes keyboard-first interaction easy to implement — one line of code per shortcut
- Python standard library csv module makes compliant CSV export trivial — no extra library needed
- Large community and extensive documentation make it easy to find solutions to problems during development

Cons

- Python's Global Interpreter Lock means audio I/O and UI rendering must run in separate threads coordinated by a Queue — threading bugs are a common source of hard-to-reproduce issues
- PyQt6 applications feel slightly less native on macOS than SwiftUI — system animations, scrollbar behavior, and font rendering differ subtly from what macOS users expect
- librosa time-stretch is an offline operation — it cannot adjust playback speed in real time, which means entering refinement mode at reduced speed requires a preprocessing step before audio starts

- PyInstaller bundles are large (80 to 180 MB) because the Python runtime and scientific libraries like numpy are included
- Implementing marker and region overlays on the waveform canvas requires more custom code than WaveSurfer.js — developers must handle hit-testing, drag interaction, and visual updates manually
- Figma UI designs must be re-implemented using Qt widgets, which imposes layout and styling constraints not present in HTML and CSS approaches

Stack 3: Electron + TypeScript + React

Electron bundles a full Chromium browser engine and Node.js runtime into a standalone desktop application. The UI is written in HTML, CSS, and TypeScript running inside Chromium's renderer process. Node.js in the main process handles file system access, spawning ffmpeg subprocesses, and writing the CSV output. This is the same architecture used by LosslessCut — an open-source application that solves a nearly identical problem to this project and is the closest existing reference implementation. The most significant advantage of this stack is WaveSurfer.js, which provides waveform rendering, region markers, looping, and a playhead almost entirely out of the box.

Technologies

- **Language:** TypeScript throughout both the main process and the renderer process
 - One language across the whole codebase — developers who know TypeScript can work anywhere in the project
 - TypeScript adds compile-time type checking on top of JavaScript, catching many bugs before runtime
- **Application Shell:** Electron v32+
 - Manages the native window, IPC between main and renderer processes, file dialogs, and macOS app lifecycle
 - Main process runs Node.js — handles file I/O, spawning ffmpeg, and writing CSV
 - Renderer process runs Chromium — renders the UI
- **Frontend Framework:** React 18 with TypeScript
 - Component-based UI that maps directly from the Figma prototype to JSX code
 - Developers familiar with web development can start building UI immediately
- **Waveform Rendering:** WaveSurfer.js 7.x with Regions plugin
 - The most fully-featured open-source waveform rendering library available
 - Regions plugin provides start and end segment markers, looping around a region, and playhead out of the box — developers write very little custom waveform code
 - This is the same library used in LosslessCut, confirming it meets the requirements of this type of application
- **Audio Playback:** Web Audio API (built into Chromium)
 - WaveSurfer.js uses Web Audio API internally — no separate audio library to install

- Playback rate change is available through `AudioBufferSourceNode` but pitch correction requires additional code or a library
- **Audio Decoding and Format Support:** Chromium native decode + ffmpeg subprocess
 - Chromium natively decodes MP4, MP3, WAV, WebM, and OGG
 - FLAC, ALAC, and some AAC variants are not supported by Chromium — ffmpeg must be invoked as a subprocess through Node.js `child_process` for these formats
- **Speed and Pitch Control:** SoundTouch.js or Web Audio API AudioWorklet
 - SoundTouch.js is a JavaScript port of the SoundTouch DSP library — provides pitch-preserving time-stretch in the browser
 - AudioWorklet allows custom audio processing in a separate thread inside Chromium — higher complexity to implement
- **Keybinding:** Mousetrap.js
 - Declarative keyboard shortcut library — developers register shortcuts in one line and attach a callback
 - Supports key sequences and modifier combinations
- **CSV Export:** Node.js built-in fs module
 - File writing happens in the main process — straightforward string formatting and file write
- **Packaging and Deployment:** Electron Builder → macOS .app and .dmg
 - Handles code signing and notarization for macOS Gatekeeper
 - Bundle size is typically 150 to 300 MB because Chromium and Node.js are both bundled regardless of how much of each is used

Pros

- WaveSurfer.js is the most capable off-the-shelf waveform library available — the Regions plugin alone covers a large portion of the spec's marker and segment requirements without custom code
- The entire UI is HTML, CSS, and TypeScript — Figma designs translate directly to code with no widget toolkit constraints, giving the highest fidelity to the prototype
- LosslessCut is built on this exact stack and solves a nearly identical problem — it is a working proof of concept and a direct reference for architecture decisions
- React's component model maps naturally to the segment list, marker panel, and waveform regions described in the spec
- Mousetrap.js makes keyboard-first interaction fast to implement
- TypeScript provides type safety across the full codebase
- Electron has the largest community and most Stack Overflow coverage of any desktop framework — finding solutions to problems is fast

Cons

- Chromium's audio scheduler introduces 10 to 20ms of timing jitter on audio events — this is the most significant technical risk for the spec's requirement that marker placement and play/pause feel instantaneous

- Bundle size (150 to 300 MB) is the largest of all options — Chromium and Node.js are bundled regardless of how little of each the app actually uses
- Memory usage is the highest of all options — Chromium's renderer process alone uses 150 to 300 MB at rest
- Pitch-preserving time-stretch for refinement mode requires SoundTouch.js or a custom AudioWorklet — neither is trivial to integrate and test
- Chromium does not natively decode FLAC, ALAC, or some AAC variants — ffmpeg subprocess fallback adds complexity and a second execution path to maintain
- IPC between Electron's main and renderer processes adds an architectural layer that makes simple file operations more verbose than they would be in a single-process app

Stack 4: Swift + SwiftUI

A fully native macOS application written in Swift using SwiftUI for the declarative user interface and AVFoundation — Apple's own audio/video framework — for playback, decoding, and speed control. This is the only stack that produces a true native macOS binary with no bundled runtime, no browser engine, and no web layer. Everything runs as compiled Swift code directly on the OS. The main trade-off is that waveform rendering must be built from scratch, because no off-the-shelf waveform library exists in the Apple ecosystem at the level of WaveSurfer.js or PyQtGraph.

Technologies

- **Language:** Swift 5.9+ throughout
 - Apple's modern compiled language — strongly typed with good tooling in Xcode
 - Requires Xcode for development, which means every developer needs a Mac
 - Unfamiliar to those who have not done Apple platform development before
- **GUI Framework:** SwiftUI with AppKit interop
 - SwiftUI is Apple's declarative UI framework — component-based and fast to prototype standard layouts
 - For the waveform canvas and precise keyboard handling, AppKit must be used through `NSViewRepresentable`, adding friction
 - Figma designs can be implemented but SwiftUI's layout system differs from CSS — pixel-perfect recreation of custom designs requires more effort
- **Waveform Rendering:** Core Graphics and Accelerate framework — custom built
 - No off-the-shelf waveform library exists for Swift — the team builds the scrolling waveform, playhead, and marker overlays from scratch
 - Accelerate's vDSP functions speed up peak extraction from audio buffers
 - This is the highest-effort component of this stack and the highest risk item for the project timeline
- **Audio Playback and Decoding:** AVFoundation and AVAudioEngine
 - Apple's own AV framework — decodes MP3, MP4, WAV, AIFF, AAC, FLAC, and ALAC natively on macOS without any external dependency

- No ffmpeg subprocess needed — format support is handled by the OS
- Provides sample-accurate current playback time, which makes precise marker placement straightforward
- **Speed and Pitch Control:** AVAudioUnitTimePitch
 - A built-in AVAudioEngine audio unit that adjusts playback rate while preserving pitch in real time
 - No preprocessing step, no third-party library — just configure the unit and connect it to the audio graph
 - This is the most robust and lowest-latency implementation of pitch-preserving speed control across all five stacks
- **Keybinding:** NSEvent global monitor and SwiftUI keyboardShortcut modifier
 - System-level and in-window keyboard shortcuts are both handled natively
- **CSV Export:** Swift standard library FileManager and String formatting
 - No external library needed — format the output string and write to disk
- **Packaging and Deployment:** Xcode Archive → native macOS .app
 - Xcode handles code signing and notarization through its built-in Distribute workflow
 - Bundle size is under 10 MB — no runtime is bundled, just compiled Swift code and assets

Pros

- Best macOS performance of all options — compiled native binary, instant launch, no runtime overhead
- AVAudioUnitTimePitch provides real-time pitch-preserving speed control with zero configuration — the most complete and lowest-effort implementation of refinement mode speed control in this comparison
- AVFoundation decodes all macOS-native formats without ffmpeg — one fewer system dependency and one fewer failure mode
- Smallest bundle size (under 10 MB) — nothing bundled except the compiled app
- Best macOS HIG compliance — Dark Mode, system fonts, scrollbar behavior, and accessibility all work exactly as macOS users expect
- No IPC, no subprocess management, no WebView — the app is a single unified process with direct access to all system APIs
- AVAudioEngine's sample-accurate scheduling makes low-latency play/pause and precise playback position queries straightforward

Cons

- The scrolling waveform with playhead must be built from scratch using Core Graphics or Metal — there is no off-the-shelf waveform library for Swift, making this the highest-effort and highest-risk component in the entire spec
- Steepest learning curve of all five options for a team without prior Apple platform experience — Xcode, SwiftUI, AVFoundation, and AppKit each have their own substantial learning requirements

- macOS only — no path to cross-platform support even as a long-term stretch goal without a complete rewrite in a different stack
- Xcode is the only practical IDE — every developer must have a Mac running a recent macOS version
- Swift Package Manager has a smaller library ecosystem than npm or PyPI — fewer ready-made solutions for application-level UI patterns
- SwiftUI still has gaps for complex custom views — the waveform canvas, drag-to-place marker interaction, and zoom will require AppKit interop, adding boilerplate

Summary

Each stack represents a different set of trade-offs between developer accessibility, audio precision, UI flexibility, and custom engineering effort. The table below highlights the key differentiator for each stack and its most significant concern.

- **Tauri:** Smallest bundle and best audio precision, but requires the team to learn Rust and manage a two-language codebase
- **Python + PyQt6:** Single language and lowest ramp-up, but waveform marker overlays and pitch-shift speed control require custom code and offline preprocessing
- **Electron + React:** WaveSurfer.js covers the most spec requirements out of the box and has the strongest real-world reference in LosslessCut, but audio timing jitter and bundle size are the biggest concerns
- **Swift + SwiftUI:** Best native macOS experience and the strongest pitch-preserving speed control, but the waveform must be built from scratch and the team must learn the Apple toolchain

Architecture Diagram

