ChatGPT

Master Task Plan: Cross-Platform AI-Enabled Media Player

The following is an exhaustive breakdown of the software project to build a cross-platform (Windows, macOS, Linux, Android, iOS) open-source media player. The architecture is highly modular [1] and designed for parallel development by AI coding agents. All major features (audio/video playback, smart playlists, visualizations, gesture/voice control, AI tagging, format conversion, streaming, subtitle sync, cross-device sync, etc.) are covered. Each module is divided into fine-grained tasks that can be developed independently in isolated environments. The plan includes estimated lines of code (LOC) per module, number of tasks, appropriate implementation languages, required libraries, and environment setup steps for each task. This modular approach also future-proofs the system for integrating AI-driven features (e.g. automatic media generation) by allowing new modules/plugins to be added easily [1] .

Module Overview and Estimates

Module	Primary Tech/Language	Estimated LOC	# of Tasks
Core Media Engine	C++17 (performance-critical), FFmpeg libs, OpenGL	**50,000**	**150**
Media Library & Playlists	C++17 (shared logic), SQLite, TagLib	**6,000**	**25**
Visualization (Audio Visuals)	C++17 (OpenGL, projectM library)	**2,500**	**12**
Desktop GUI (Win/Mac/Linux)	Qt 6 (C++/QML UI)	**10,000**	**40**
Android App	Kotlin (Android SDK, JNI)	**6,000**	**30**
iOS App	Swift (SwiftUI/UIKit, Obj-C bridge)	**6,000**	**30**
Gesture & Voice Control	C++17 (voice engine), Platform SDKs (Java/Swift for integration)	**3,000**	**15**
AI Tagging Service	Python (ML frameworks) or C++ (ONNX)	**4,000**	**15**
Format Conversion Tool	C++17 (FFmpeg encoding)	**1,000**	**5**
Streaming & Network	C++17 (Networking libs), Platform APIs	**2,000**	**10**
Subtitle Support	C++17 (libass or custom parse), Qt/QML overlay	**1,000**	**5**

Module	Primary Tech/Language	Estimated LOC	# of Tasks
Cross-Device Sync	C++17 (sockets, mDNS), Platform integration	**3,000**	**15**
DevOps & Infrastructure	Docker, CMake, Gradle/Xcode, CI scripts	**800**	**10**
Testing & QA	Various (GoogleTest, XCTest, etc.)	**2,500**	**20**
Total Estimated	*(Multi-language, cross-platform)*	**~97,800**	**~372**

(Lines of code are rough estimates for initial implementation. Tasks counts are approximate and represent fine-grained parallelizable units.)

Below, each module is detailed with its tasks. Each task listing includes a short description, chosen language and libraries, and explicit environment setup steps so it can be executed in isolation (e.g. in a containerized agent).

Core Media Engine (C++17)

Description: The Core Engine handles all low-level media functionality: file access, decoding audio/video, playback control, output to audio/video devices, and core features like format conversion and subtitle timing. It is written in C++ for performance and portability. The engine leverages the FFmpeg libraries (libavformat, libavcodec, etc.) for decoding a wide range of codecs, similar to how VLC uses libavcodec internally [2]. The design is highly modular (following VLC's plugin-like architecture [1]) to allow easy addition of new codecs, filters, or output modules. The engine exposes a C API and C++ interface for UIs to control playback (play, pause, seek, etc.) and retrieve status. Estimated ~50k LOC. Approximately 150 tasks are defined for maximum parallelism.

- **Setup & Infrastructure:**
- *Task:* **Initialize Core Engine Project Structure** – Create the core engine directory layout, initialize a Git repository, and write a CMakeLists.txt capable of building on all target platforms. Include placeholders for sub-modules (decoders, outputs, etc.). *Language:* C++17. *Libraries:* CMake, GoogleTest (for later tests). *Environment:* **Ubuntu 22.04 container** – install build-essential, cmake, git. **Windows container/VM** – install Visual Studio Build Tools 2022. **macOS** – Xcode CLI tools. (Each environment should be configured with C++17 support and have FFmpeg dev libraries available.)
- *Task:* **Define Core Engine API (Header)** – Design the core API in a header (e.g., `MediaPlayer.h`) with functions/classes for opening media, controlling playback, and event callbacks. This acts as the contract between engine and UIs. *Language:* C++17 (header only). *Environment:* Same as above (C++ dev). No external libs needed for this task.
- *Task:* **Stub Implementation of Core Classes** – Create empty implementations (.cpp files) for key classes (MediaPlayer, MediaDecoder, AudioOutput, VideoOutput, PlaylistManager, etc.) with function stubs that return default values or logs. This allows parallel development of each component. *Language:* C++17. *Environment:* C++ dev environment; ensure linking with FFmpeg libs is set up (even if not used yet). (E.g., run `apt-get install libavcodec-dev libavformat-dev libavutil-dev` on Linux container to have FFmpeg headers [2]).

- *Task:* **Build & Integration CI Setup** – Write a GitHub Actions (or similar CI) config to build the core engine on each platform (using CMake, and platform-specific toolchains). Ensure it compiles the stubbed code. *Language:* YAML/CI config. *Environment:* Docker image with cmake and compilers, plus actions for Windows and macOS runners. Steps: install dependencies (FFmpeg, etc.), run CMake, build.

- **Decoding Pipeline:**

- *Task:* **Integrate FFmpeg for Demuxing** – Use libavformat to open media files/streams and read packets. Implement a `MediaDemuxer` class that can enumerate streams (audio/video/subtitle) and read frames. *Language:* C++17. *Libraries:* FFmpeg (avformat). *Environment:* C++ dev (with `libavformat-dev` installed; e.g., run `apt-get install libavformat-dev`).
- *Task:* **Implement Audio Decoding (FFmpeg)** – Use libavcodec to decode audio frames (e.g. MP3, AAC, FLAC, etc.) to raw PCM. Implement `AudioDecoder` that sets up codec context and decodes packets. (Leverages FFmpeg's built-in codecs for broad format support [2].) *Language:* C++17. *Libraries:* FFmpeg (avcodec, swresample for audio resampling). *Environment:* C++ dev with FFmpeg libs (ensure `libavcodec-dev`, `libswresample-dev` installed).
- *Task:* **Implement Video Decoding (FFmpeg)** – Similar to audio: decode video frames (H.264, HEVC, VP9, etc.) to raw pixel frames (YUV). Implement `VideoDecoder` class. Use libswscale to convert frames to a format suitable for rendering (e.g. RGBA). *Language:* C++17. *Libraries:* FFmpeg (avcodec, swscale). *Environment:* C++ dev with FFmpeg libs (`libswscale-dev` etc.).
- *Task:* **Audio/Video Synchronization** – Implement timing control to sync audio and video. E.g., maintain a clock for audio (since audio out is often master), adjust video frame timing to match. *Language:* C++17. *Libraries:* std::chrono for timing. *Environment:* C++ dev. (No extra libs; just ensure engine can use high precision timers, which is default.)
- *Task:* **Buffering and Caching Logic** – Implement read buffers for streaming files: if reading from slow source or network, buffer data. Also handle network stream buffering (reading ahead). *Language:* C++17. *Libraries:* Boost.Asio or FFmpeg's internal IO callbacks. *Environment:* C++ dev, install `libboost-dev` if using Boost.
- *Task:* **Threading and Locking** – Set up decoding threads: one thread for demuxing, separate threads for audio decode, video decode (if needed), and rendering. Ensure thread-safe queues (e.g., packets and frames queue), and use mutexes/conditions for synchronization. *Language:* C++17 (std::thread, mutex). *Environment:* C++ dev, no extra libs.

- *Task:* **Hardware Decoding Support (Optional)** – For performance, add support for hardware-accelerated video decoding on each platform. E.g., use FFmpeg with DXVA2/D3D11VA on Windows, VideoToolbox on macOS, MediaCodec on Android. Implement conditional codec initialization that tries hardware decoders. *Language:* C++17. *Libraries:* FFmpeg (with hwaccel support), platform SDKs for HW decode. *Environment:* C++ dev plus platform-specific: e.g. Windows with Windows SDK (for DXVA), macOS with AVFoundation frameworks, Android NDK for MediaCodec via JNI. Steps: ensure FFmpeg is built with `--enable-hwaccels`.

- **Audio Output (Platform Interfaces):**

- *Task:* **Abstract Audio Output Interface** – Design an interface (abstract class `AudioOutput`) that defines methods for initializing audio device, writing PCM data, volume control, etc. This will have concrete implementations per OS. *Language:* C++17. *Environment:* C++ dev.

- *Task:* **Audio Output – Windows (WASAPI)** – Implement `AudioOutputWASAPI` using Windows WASAPI API for low-latency audio playback. *Language:* C++17. *Libraries:* Windows Core Audio APIs (WASAPI). *Environment:* **Windows 11 VM** with MSVC, Windows SDK. Steps: include `<audioclient.h>`, link `ole32.lib`, etc.
- *Task:* **Audio Output – macOS (CoreAudio)** – Implement `AudioOutputCoreAudio` using CoreAudio (AudioQueue or AVAudioEngine). *Language:* C++17/Obj-C++ if needed. *Libraries:* macOS CoreAudio framework. *Environment:* **macOS** with Xcode. Steps: link CoreAudio.framework, use Objective-C++ (.mm file) for easier Apple API calls if needed.
- *Task:* **Audio Output – Linux (PulseAudio/ALSA)** – Implement `AudioOutputPulse` using PulseAudio or ALSA. (PulseAudio is common on modern Linux.) *Language:* C++17. *Libraries:* PulseAudio client library. *Environment:* **Ubuntu 22.04** – install `libpulse-dev` (or `alsa-lib` if ALSA used).
- *Task:* **Audio Output – Android (OpenSL ES or AAudio)** – Implement audio output via Android's native audio. For older devices use OpenSL ES, for newer use AAudio. This will be called via JNI from core or integrated in a small Android-specific C++ component. *Language:* C++ (via NDK). *Libraries:* Android NDK (OpenSL ES headers or AAudio). *Environment:* **Android NDK r23+** in a container (install via `sdkmanager ndk-bundle`).
- *Task:* **Audio Output – iOS (AVAudio)** – Use iOS AVAudioSession/AudioUnits for output. Implement `AudioOutputiOS` (Obj-C++ likely). *Language:* C++/Obj-C++. *Libraries:* AVFoundation/AudioToolbox. *Environment:* **macOS with Xcode** (iOS SDK). Steps: link AVFoundation, use `.mm` file for Obj-C API.

- *Task:* **Audio Buffering & Mixing** – Implement buffering in audio output (use double-buffering or ring buffer to feed audio API smoothly). Also handle basic mixing if needed (e.g., system sounds or crossfade future features). *Language:* C++17. *Environment:* C++ dev.

- **Video Output (Rendering):**

- *Task:* **Abstract Video Output Interface** – Define interface `VideoOutput` for presenting video frames. Likely methods like `init(window)`, `renderFrame(Frame*)`, etc. The actual rendering may be delegated to UI in some cases, but define a common protocol. *Language:* C++17. *Environment:* C++ dev.
- *Task:* **Video Rendering – OpenGL (Desktop)** – Implement a cross-platform video renderer using OpenGL (for Windows, Linux, possibly macOS). Create an OpenGL context (or accept one from UI), upload decoded YUV frame as texture, use a shader to convert to RGB and display. *Language:* C++17/OpenGL (GLSL shader). *Libraries:* GLEW/GLFW (for context if headless), or use Qt's OpenGL if within Qt UI. *Environment:* **Ubuntu** – install `mesa-common-dev` (OpenGL), **Windows** – use WGL via Win32 or Qt, **macOS** – use CGL or Metal (see below).
- *Task:* **Video Rendering – DirectX (Windows optional)** – Alternatively on Windows, implement using Direct3D 11 for potentially better integration. This task can create a D3D11 texture from YUV and render. *Language:* C++17. *Libraries:* Direct3D 11. *Environment:* **Windows** with DirectX SDK (part of Windows SDK).
- *Task:* **Video Rendering – Metal (macOS)** – On macOS (and iOS), use Metal for rendering frames (since OpenGL is deprecated on Apple platforms). Implement a Metal layer that receives frames (possibly convert using vImage). *Language:* Objective-C++/Swift (for Metal setup). *Libraries:* MetalKit. *Environment:* **macOS** with Xcode (Metal framework).
- *Task:* **Frame Renderer – Android (OpenGL ES)** – Implement rendering in Android using OpenGL ES. Likely, we render on a `SurfaceView` or `TextureView` via an OpenGL ES context. *Language:* C++

(NDK OpenGL ES), with JNI hooks to an Android `Surface`. *Libraries:* OpenGL ES 3.0, EGL. *Environment:* **Android NDK** with OpenGL support.

- *Task:* **Frame Renderer – iOS (Metal/GL ES)** – On iOS, use either OpenGL ES or Metal. If using a unified approach, we can use OpenGL ES 3.0 on iOS as well via GLKit (though Apple prefers Metal). Implement an `AVSampleBufferDisplayLayer` alternative if not using custom GL. *Language:* Obj-C++/Swift. *Libraries:* OpenGL ES via GLKit, or Metal. *Environment:* **Xcode iOS** with appropriate frameworks.

- *Task:* **Video Output Integration** – Connect the decoder thread to video output: copy or share decoded frames to the rendering component. Ensure thread safety and possibly use double-buffering or queue to avoid frame drops. *Language:* C++17. *Environment:* C++ dev.

- **Playback Control & Media Session:**

- *Task:* **Implement Play/Pause/Seek Logic** – Code the logic for play, pause, stop, and seek. Manage decoder threads accordingly (e.g., pause should block decoding, seek should flush decoder buffers and jump to new timestamp). *Language:* C++17. *Environment:* C++ dev.
- *Task:* **Track and Playlist Management (Core)** – In core, implement support for playlist playback (sequential or shuffle). The UI or library module can supply a playlist (list of file URLs), and engine handles transitioning to next track, etc. *Language:* C++17. *Environment:* C++ dev.
- *Task:* **Playback State Notifications** – Implement callback or observer pattern to notify UIs of state changes (playback started, paused, track ended, position updated, etc.). E.g., a function pointer or signal that UIs can hook. *Language:* C++17. *Environment:* C++ dev.
- *Task:* **Volume and Audio Effects** – Implement volume control in core (scale PCM samples if not using system mixer's volume). Also provide hooks for audio effects (e.g., equalizer or DSP effects as future plugin). *Language:* C++17. *Environment:* C++ dev.

- *Task:* **Media Metadata Extraction** – When a media is loaded, extract metadata (title, duration, etc.) using FFmpeg's metadata or TagLib for audio files. Provide this info via the core API. *Language:* C++17. *Libraries:* FFmpeg (avformat metadata APIs) and TagLib (for more detailed tags). *Environment:* C++ dev (with TagLib installed – e.g., `apt-get install libtag1-dev` on Linux).

- **Format Conversion (Transcoding):**

- *Task:* **Audio Format Conversion Utility** – Implement function to convert an audio file to another format (e.g., WAV to MP3). Leverage FFmpeg's encoding capabilities. For example, decode using core pipeline, then initialize an encoder (lame for MP3 via libavcodec) to write output. *Language:* C++17. *Libraries:* FFmpeg (avcodec for encoders, avformat for muxing). *Environment:* C++ dev with libmp3lame or other codec libs available (FFmpeg usually includes popular encoders).
- *Task:* **Video Transcoding Utility** – Similar for video: transcode from one format to another or change resolution/bitrate. Use libavcodec encoders (H.264, etc.). Implement options for user (quality settings). *Language:* C++17. *Libraries:* FFmpeg. *Environment:* C++ dev with FFmpeg (ensure libx264, etc. are present if possible).
- *Task:* **Conversion Task Management** – Make conversion run in a separate thread or process to not block playback. Possibly implement this as a separate module or background worker. Ensure progress callbacks for UI. *Language:* C++17. *Environment:* C++ dev.

- *Task:* **Integration in UI** – (Coordination with UI tasks) Provide core API calls to start a conversion job and report progress. Stub out these calls here for UI to use. *Language:* C++17. *Environment:* C++ dev.

- **Streaming & Network Protocols:**

- *Task:* **Network Stream Input Support** – Enable FFmpeg's network support to play URLs (HTTP/HTTPS, RTMP, etc.). This may just require linking with networking libs (FFmpeg handles protocols if built with them). Verify that libavformat is configured with HTTP, HLS, etc. *Language:* C++17. *Libraries:* FFmpeg (with libcurl for HTTP if needed). *Environment:* C++ dev; ensure `libcurl4-openssl-dev` installed so FFmpeg can use it for network.
- *Task:* **YouTube Integration (Optional)** – (Optional advanced) Implement ability to play YouTube (and similar) links by integrating youtube-dl or its library to fetch direct stream URLs. *Language:* Python (if calling youtube-dl) or C++ (system call). *Libraries:* youtube-dl or youtube-dlp (as an external tool). *Environment:* Python 3 available in container (for calling youtube-dl via subprocess). Steps: `pip install youtube_dl`.
- *Task:* **Streaming Protocols (HLS/DASH)** – Test and ensure HLS (m3u8 playlists) and MPEG-DASH streams work. FFmpeg can demux these if configured. Possibly implement a custom handler for adaptive bitrate (choose representation). *Language:* C++17. *Environment:* C++ dev (no new libs beyond FFmpeg).

- *Task:* **Internet Radio Streams** – Allow playing audio streams (MP3/AAC over HTTP, etc.). Handle ICY metadata (song info) if present. *Language:* C++17. *Libraries:* None beyond FFmpeg. *Environment:* C++ dev.

- **Subtitle Handling:**

- *Task:* **Subtitle Parser (SRT)** – Implement support for external subtitle files (.srt). Parse SRT format (text file with timestamp cues). *Language:* C++17. *Environment:* C++ dev.
- *Task:* **Subtitle Renderer/Provider** – Provide decoded subtitle text and timing to UI. For text subtitles like SRT, core can send the subtitle text and timestamp to UI to display. For advanced subtitles (ASS/SSA), consider integrating **libass** to render to bitmaps. *Language:* C++17. *Libraries:* libass (for ASS subtitles) if needed. *Environment:* C++ dev, install `libass-dev` on Linux if using.
- *Task:* **Subtitle Sync Adjustment** – Implement mechanism to adjust subtitle timing (delay or advance) during playback. Core can maintain an offset value applied to subtitle timestamps. *Language:* C++17. *Environment:* C++ dev.

- *Task:* **Subtitle Track Selection** – If video has embedded subtitles (e.g., in MKV), allow selecting track via core API. Use FFmpeg to demux subtitle stream and decode (libavcodec can decode DVDSUB, etc.). *Language:* C++17. *Environment:* C++ dev.

- **Visualization Interface:** (See *Visualization Module* below for actual visualizer implementation)

- *Task:* **Audio Visualization Feed** – In core, implement a way to feed audio PCM data to the visualization module. For example, after audio decode (before output), copy PCM samples to a shared buffer or call a visualization callback. *Language:* C++17. *Environment:* C++ dev.

- *Task:* **Visualization API Hook** – Define an interface in core for plug-in visualizers (e.g., a function pointer that receives PCM data chunks or an abstract `Visualizer` class with a callback). *Language:* C++17. *Environment:* C++ dev.

- *Task:* **Spectrum Analysis (optional)** – As part of core, implement an FFT on audio samples to produce frequency spectrum if not using an external library. (Optional, since projectM can handle internally, but core could provide a basic visualization mode itself.) *Language:* C++17. *Libraries:* FFT library (like KissFFT) or use FFmpeg's avfilter for audio FFT. *Environment:* C++ dev, install any needed math lib for FFT if chosen.

- **Performance & Optimization:**

- *Task:* **Memory Management Audit** – Ensure large buffers (video frames, audio buffers) are allocated efficiently (reuse where possible). Implement memory pool for frames to reduce allocations. *Language:* C++17. *Environment:* C++ dev.
- *Task:* **CPU/GPU Optimization** – Enable compiler optimizations (O3) and consider SIMD for heavy operations (like audio sample conversion). Possibly use FFmpeg's optimized conversions via libswresample and libswscale which use SIMD internally [3] [4]. *Language:* C++17. *Environment:* C++ dev, ensure compiler flags for NEON (ARM) and SSE2/AVX (x86) where applicable.
- *Task:* **Stress Testing Tools** – Write a simple load test (in C++ or Python) that opens multiple media instances to ensure engine handles it (if intended to allow multiple plays). *Environment:* C++ dev or Python with bindings.

(The core engine's modular design lays the foundation for all features. Its use of FFmpeg provides broad codec support out of the box [2], and the plugin-like structure allows adding new modules (codecs, filters, outputs) as needed [1].)

Media Library & Smart Playlists (C++17, SQLite)

Description: This module manages the media library database, metadata, playlists (including "smart playlists"), and search functionality. It indexes media files (audio and video), stores metadata (title, artist, genre, etc.), and supports queries for dynamic playlists. Using a small C++ module for library ensures logic can be shared across platforms. It will use **SQLite** as an embedded database to store media info and playback history, and **TagLib** to read audio metadata tags (ID3, etc.) for accurate media info [5]. Smart playlists allow filtering or AI-based recommendations. Estimated ~6k LOC, ~25 tasks.

- **Library Database Setup:**
- *Task:* **Integrate SQLite Database** – Add SQLite as a dependency and set up a database file (e.g., `media_library.db`) for storing media entries (with fields for path, metadata, play count, last played, etc.). Implement a `LibraryDB` class to wrap SQLite operations (open DB, run queries). *Language:* C++17. *Libraries:* SQLite3. *Environment:* C++ dev; ensure `libsqlite3-dev` installed (on Windows, include SQLite amalgamation).
- *Task:* **Define DB Schema** – Define tables for `MediaItem` (tracks/videos), `Playlist`, `PlaylistItems`, etc. Write SQL schema and have the code execute it at first run. *Language:* SQL (embedded as string in C++ or separate .sql file). *Environment:* C++ dev.
- *Task:* **Implement Library Scanning** – Write a function to scan given directories for media files. For each file, retrieve metadata (via TagLib for audio, via FFmpeg for video duration/resolution) and

insert or update the database. *Language:* C++17. *Libraries:* TagLib for metadata (fast C++ library for audio tags [5]), FFmpeg (for video metadata if needed). *Environment:* C++ dev; ensure TagLib is installed (`apt-get install libtag1-dev`).

- *Task:* **Metadata Extraction** – Use TagLib to get title/artist/album for audio files, and basic metadata for video (maybe just filename as title if no tags). Ensure Unicode support (TagLib handles Unicode tags [6]). *Language:* C++17. *Environment:* C++ dev with TagLib.

- *Task:* **Update/Remove Entries** – Implement functions to update metadata (e.g., user edits tags) or remove entries (if file deleted). Ensure changes reflect in DB. *Language:* C++17 (SQL execution). *Environment:* C++ dev.

- **Smart Playlist & Search:**

- *Task:* **Basic Playlist Management** – Implement a `Playlist` class for user-created playlists (a list of media item IDs). Support saving playlists to the DB and loading them. *Language:* C++17. *Environment:* C++ dev.
- *Task:* **Smart Playlist Criteria** – Allow creation of playlists based on filters, e.g., "genre = rock and rating >= 4". Implement a simple query builder that fetches matching items from the DB. *Language:* C++17/SQL. *Environment:* C++ dev.
- *Task:* **Auto Playlists (Recent, Frequent)** – Create dynamic lists like "Recently Added", "Most Played" by querying DB (order by added_date, play_count, etc.). *Language:* C++17/SQL. *Environment:* C++ dev.
- *Task:* **AI Recommendations Hook** – (Future-proofing) Provide an interface to accept AI-generated playlists or song suggestions. For now, perhaps a stub that could take e.g. a mood or user listening history and produce a list. (The actual AI logic might be in the AI Tagging module or future extension.) *Language:* C++17. *Environment:* C++ dev.
- *Task:* **Search Functionality** – Implement a search query function that searches songs/videos by title, artist, album, etc. Use SQL `LIKE` queries (and possibly FTS (Full Text Search) extension of SQLite for efficiency). *Language:* C++17/SQL. *Environment:* C++ dev (enable SQLite FTS module if needed).

- *Task:* **Rating System** – (Optional) Add support for rating tracks (1-5 stars) and store in DB. This can feed into smart playlists (e.g., "Highly Rated"). *Language:* C++17/SQL. *Environment:* C++ dev.

- **Library API & Sync:**

- *Task:* **Library-Core Integration** – Connect the library with the core engine: when media is played, update play count, last played timestamp in DB. When new media added via UI, call library scan to add to DB. *Language:* C++17. *Environment:* C++ dev.
- *Task:* **Expose Library API to UI** – Provide methods to retrieve lists of media (for UI to display library content), playlist contents, etc. These could be exposed through the core's API (e.g., core engine has a pointer to LibraryDB or a facade methods like `getAllSongs()`). *Language:* C++17. *Environment:* C++ dev.
- *Task:* **Threading for DB** – Ensure library operations (like scanning or heavy queries) run off the main UI thread. Possibly use a worker thread in this module. *Language:* C++17 (std::thread). *Environment:* C++ dev.
- *Task:* **Smart Playlist Evaluation** – Implement re-evaluation triggers for smart playlists (e.g., if metadata changes or after each new addition, update any smart playlists). *Language:* C++17. *Environment:* C++ dev.

(By using TagLib for audio metadata and SQLite for storage, this module can efficiently manage a large media collection [5]. "Smart playlists" are essentially saved queries on the metadata, which can later be enhanced with AI tagging data for mood/genre-based mixes.)

Visualization Module (C++17 OpenGL)

Description: The visualization module generates real-time graphical visualizations of audio output (e.g. waveforms, spectrums, psychedelic effects) that sync with the music. It will integrate **projectM**, the popular open-source music visualizer library that reimplements Winamp Milkdrop visuals [7]. Using projectM allows a rich set of pre-defined visualizations and is cross-platform (works on Windows, Mac, Linux, Android, iOS with OpenGL ES). Estimated ~2.5k LOC for integration/glue code, ~12 tasks (not counting the projectM library code itself, which is external).

- **Visualization Engine Integration:**
- *Task:* **Include projectM Library** – Add projectM as a dependency (either as a submodule or linking to installed library). projectM provides a library (`libprojectM`) that can be fed audio PCM data and returns visual rendering via OpenGL [7]. *Language:* C++17. *Libraries:* projectM (C++). *Environment:* C++ dev; install `libprojectM-dev` if available (or build from source). Ensure OpenGL/GL ES available in environment.
- *Task:* **Initialize projectM Instance** – Write code to create a projectM context with a configuration (width, height, config settings like mesh size). This context will generate visualization frames. *Language:* C++17. *Environment:* C++ dev.
- *Task:* **PCM Data Feeding** – Implement a function to send audio PCM samples to projectM's input (e.g., `projectM::pcm()` API). Connect this to the core engine's visualization hook (so each audio frame or chunk decoded is forwarded). *Language:* C++17. *Environment:* C++ dev.

- *Task:* **OpenGL Context Sharing** – Ensure the visualization can render to a texture or screen. On desktop, possibly create an OpenGL texture the module draws into; on mobile, integrate with an OpenGL ES view. This may involve obtaining an OpenGL context from the UI. *Language:* C++17/OpenGL. *Environment:* C++ dev with OpenGL (for desktop) and OpenGL ES (for mobile via NDK).

- **Rendering & Display:**

- *Task:* **Render Visualization Frame** – Use projectM to render the next frame. Typically projectM can render into the current OpenGL context. Implement a method `Visualizer::render()` that calls projectM's render function (which draws using GL). *Language:* C++17/OpenGL. *Environment:* C++ dev.
- *Task:* **Offscreen Rendering (Optional)** – If direct on-screen drawing is not convenient, render into an offscreen framebuffer/texture. Then the UI can display that texture in its UI framework (e.g., Qt Quick texture, Android Canvas). *Language:* C++17/OpenGL. *Environment:* C++ dev.
- *Task:* **Switch Visualization Presets** – Allow cycling through different visual presets (projectM has many). Implement a control to load next preset (projectM API provides preset loading). *Language:* C++17. *Environment:* C++ dev.

- *Task:* **Performance Optimization** – projectM can be heavy; adjust settings (mesh size, frame rate) for low-power devices. Possibly auto-reduce quality on mobile. *Language:* C++17. *Environment:* C++ dev, no extra libs.

- **Integration with UI:**

- *Task:* **Expose Visualizer to UI** – Provide a way for UI to enable/disable visualization and to embed the visualization output in the player screen. For Qt, this might be a QML OpenGL underlay; for mobile, a custom view. The core can offer a texture handle or simply run the visualizer drawing when active. *Language:* C++17 (with hooks for UI frameworks). *Environment:* C++ dev and platform dev (for connecting to UI, e.g., JNI or Qt signals).
- *Task:* **Control Interface** – (UI side mostly) but ensure core can respond to visualization settings (like toggling on/off, switching presets as above). Possibly just stub functions in core like `enableVisualization(true/false)`. *Language:* C++17. *Environment:* C++ dev.

(By leveraging projectM, we gain "psychedelic and mesmerizing visuals" driven by audio input [7] without writing visualizer logic from scratch. This module acts as a plugin to the core engine, activated on demand.)

Desktop GUI (Qt/QML, C++)

Description: The desktop application (Windows, macOS, Linux) will use **Qt 6** with **QML** for a modern, fluid UI. Qt is chosen for its cross-platform native performance and flexibility (VLC also uses Qt for its interface on Windows/Linux) [8]. The UI will interact with the core engine via C++ calls (since we can link the core as a library). QML (Qt's declarative UI language) will define the interface, with C++ providing integration logic and custom components (e.g., video output widget, OpenGL visualization widget). We will ensure a native look on each OS (using Qt's adaptive styling or custom QML themes). Estimated ~10k LOC (QML + C++). ~40 tasks.

- **Environment Setup (Qt):**
- *Task:* **Qt Project Initialization** – Create a Qt Quick application project. Set up qmake or CMake for Qt, define main application entry (C++ `main.cpp` launching a QML UI). *Language:* C++ (for main) + QML. *Environment:* **Qt 6 environment** – e.g., use a Docker image with Qt 6 SDK or install Qt (Qt 6.5) on each platform. On Ubuntu, install `qt6-base-dev`, `qt6-declarative-dev`; on Windows, use Qt installer; on macOS, brew install qt or use Qt installer. Ensure QML modules (Qt Quick, Controls) are available.

- *Task:* **Integrate Core Engine Library** – Link the core C++ engine to the Qt app. If core is built as a static or shared library, link it and include the core headers. Verify that the app can call core functions. *Language:* C++ integration. *Environment:* Qt/C++ dev – ensure the core library (.lib, .a or .dylib/.so) is accessible. Update build scripts to find core includes and lib.

- **UI Layout & Components:**

- *Task:* **Design Main Player UI (QML)** – Create QML for the main playback screen: controls (play/pause button, next/prev, seek slider), volume slider, album art/video display area, etc. Use Qt Quick Controls 2 for standard components (buttons, sliders) to ensure cross-platform native feel. *Language:* QML (declarative). *Environment:* Qt Designer or text editor, no extra deps.
- *Task:* **Implement Playback Controls (C++ Logic)** – Write C++ slots or QML JavaScript that connect UI buttons to core engine calls (e.g., on Play button click, call core.play()). In Qt, we can expose a C++ class (MediaPlayerController) as a context property to QML, bridging UI and core. *Language:* C++ (Qt signals/slots) + QML integration. *Environment:* Qt dev.

- *Task:* **Media Library View (QML ListView)** – Build a library browsing screen in QML. Use `ListView` or `GridView` to display songs/videos retrieved from the Library module. Implement a model (QAbstractListModel in C++) that fetches data from the library DB and exposes it to QML. *Language:* C++ (model) + QML (UI). *Environment:* Qt dev.
- *Task:* **Search UI** – Add a search bar QML element. On text input, call a search function (from library module) and update the list view with results. *Language:* QML + C++. *Environment:* Qt dev.
- *Task:* **Playlist Management UI** – Create a view for playlists: list existing playlists, allow creating new ones. Also a "Now Playing" queue view. Provide drag-and-drop or buttons to add/remove items. *Language:* QML. *Environment:* Qt dev.
- *Task:* **Smart Playlist Editor** – UI to define smart playlist rules: e.g. a dialog where user picks field (genre, rating) and condition. This UI will produce a filter query for the library. *Language:* QML + some C++ for applying filter. *Environment:* Qt dev.
- *Task:* **Settings Dialog** – Provide a settings UI (QML) for various preferences: audio output device selection, theme (light/dark), enabling features (visualizations, cross-device sync login, etc.), key shortcuts, etc. *Language:* QML. *Environment:* Qt dev.
- *Task:* **Video Player Window** – For video playback, possibly use a separate window or fullscreen mode. Implement either as part of main window or a full-screen overlay. The video output from core needs to be displayed; in Qt we can use QML's `ShaderEffect` or `QQuickFramebufferObject` to show the video frame or texture provided by the core. *Language:* C++/Qt (for a custom QQuickItem to display frames) + QML. *Environment:* Qt dev (with OpenGL enabled for Qt Quick).
- *Task:* **Visualization Canvas** – Integrate the visualization module: e.g., create a QML item that either wraps an OpenGL context (perhaps using QQuickFramebufferObject to allow custom OpenGL rendering inside QML). Connect it to core's visualizer. *Language:* C++ (QQuickFramebufferObject subclass) + QML. *Environment:* Qt dev with OpenGL.

- *Task:* **Responsive Layout** – Ensure the UI adapts to different window sizes and HiDPI. Use anchors and Layouts in QML to make it responsive. *Language:* QML. *Environment:* Qt dev.

- **Platform-specific UI Adjustments:**

- *Task:* **Windows Integration** – Use QtWinExtras if needed for better Windows integration (e.g., taskbar progress, thumbnail toolbar buttons for media controls). *Language:* C++ (QtWinExtras). *Environment:* Qt on Windows.
- *Task:* **macOS Integration** – Use Mac native menus or touch bar. Possibly integrate with macOS media keys and Now Playing Center. *Language:* C++/Objective-C (for macOS AppKit integration). *Environment:* Qt on Mac (with Objective-C++ for native calls).

- *Task:* **Linux Integration** – Ensure compatibility with different distros/desktop environments. Perhaps support MPRIS (Media Player Remote Interfacing Specification) DBus interface so that desktop environments can control playback (e.g., sound menu in Ubuntu). *Language:* C++ (Qt DBus). *Environment:* Linux with DBus dev packages.

- **Core-UI Binding:**

- *Task:* **MediaPlayer Controller Class** – Implement a singleton C++ class in the UI project that holds an instance of core `MediaPlayer` and mediates between QML and core. This class has slots like `playFile(path)` which calls core->open(path)+play, etc., and signals for core callbacks (e.g., onTrackEnded -> signal to QML to update UI). *Language:* C++ (Qt). *Environment:* Qt dev.

- *Task:* **Expose Properties to QML** – Use `Q_PROPERTY` to expose playback state, current track info, progress, etc., to QML bindings (so UI elements can auto-update). Update these properties based on callbacks from core. *Language:* C++. *Environment:* Qt dev.
- *Task:* **Event Handling** – Map key presses (space = play/pause, arrow keys = seek, etc.) and other input to core actions. Qt allows handling key events in QML or C++. Implement accordingly. *Language:* QML (Keys.onPressed) or C++. *Environment:* Qt dev.

- *Task:* **Error Handling UI** – Display error dialogs or notifications for errors reported by core (file not found, codec error, etc.). *Language:* QML. *Environment:* Qt dev.

- **UI Theming and Assets:**

- *Task:* **Light/Dark Theme** – Use Qt Quick Controls theming or custom QML styling to support dark mode and light mode based on user preference or system setting. *Language:* QML (use `QtQuick.Controls.Material` or custom). *Environment:* Qt dev.
- *Task:* **Icons and Graphics** – Import icon assets (play, pause, etc.) or use Qt's icon fonts. Ensure icons are properly licensed (open source) since project is open-source. *Environment:* design assets included, used in QML.

- *Task:* **Localization Support** – Set up Qt translation (QSTr) for UI text, to allow future i18n. Provide an English translation file as default. *Language:* Qt Linguist (.ts files). *Environment:* Qt dev.

- **Desktop App Packaging:**

- *Task:* **Windows Installer Setup** – Create an NSIS or MSI installer script or use windeployqt to package the Windows build with Qt DLLs and necessary codecs (if any). *Environment:* Windows with Qt. Steps: run `windeployqt` on build output, then NSIS script to create installer.
- *Task:* **macOS Bundle** – Create a .app bundle, ensure Qt frameworks are copied (macdeployqt), and prepare a signed/notarized build (if necessary for distribution). *Environment:* macOS with Qt.
- *Task:* **Linux Package/AppImage** – Provide either an AppImage or .deb package. Possibly use linuxdeployqt for AppImage creation. *Environment:* Linux.

(The Qt/QML UI will give a consistent, feature-rich interface across desktop OSes [8]. By using a common core controller in C++, we avoid duplicating logic. QML allows rapid UI development and dynamic updates (e.g., binding slider to playback position).)

Android App (Kotlin + NDK)

Description: The Android app provides a native Android user experience adhering to Material Design. It will be written mostly in **Kotlin** for UI and Android-specific logic, with the core engine integrated via **JNI** (Java Native Interface) calling into the C++ core library compiled for ARM/x86 Android. The UI will include activities/fragments for library, playback, settings, etc., using Android Jetpack libraries (e.g., RecyclerView, MediaSession for lock-screen controls, etc.). Estimated ~6k LOC (not counting core). ~30 tasks.

- **Project Setup:**
- *Task:* **Create Android Studio Project** – Initialize a new Android project (com.example.mediaplayer). Set minSdk (e.g., 24) and targetSdk (current). Choose a single-activity (with Navigation components)

or multi-activity structure. *Language:* Kotlin (for scaffold). *Environment:* **Android SDK** – ensure Android Studio or SDK command-line tools are installed. Use Gradle to manage. (In container, one could use gradle wrapper with SDK installed via sdkmanager).

- *Task:* **NDK Integration** – Enable C++ support. Add an NDK module or CMakeLists in the Android project to compile or include the core engine. Alternatively, compile core separately and include as prebuilt .so. Configure Gradle to include the native library (ABI splits for arm64-v8a, armeabi-v7a, x86_64). *Environment:* Android NDK (install via sdkmanager). Ensure `android.useAndroidX=true` and appropriate gradle plugin.

- **UI Implementation (Kotlin):**

- *Task:* **Main Activity & Navigation** – Implement MainActivity in Kotlin. Set up navigation graph for fragments: e.g., LibraryFragment, NowPlayingFragment, SettingsFragment. Use Jetpack Navigation component for transitions. *Language:* Kotlin. *Libraries:* AndroidX AppCompat, Navigation. *Environment:* Android SDK.
- *Task:* **Library Fragment (RecyclerView)** – UI to display media library (song list, video list). Use a RecyclerView with a list adapter bound to the media library data (from core or from a mirrored SQLite in Java if needed). For now, call core via JNI to get list, or possibly use an intermediate content provider. *Language:* Kotlin. *Environment:* Android SDK.
- *Task:* **Now Playing Fragment** – UI for currently playing track/video. Show controls (play/pause, next, prev), seek bar, title, etc. This fragment will call native methods for control. For video, use a SurfaceView or TextureView to show the video output (we'll connect this surface to the native video rendering). *Language:* Kotlin + small C++ (for rendering to Surface). *Environment:* Android SDK/NDK.
- *Task:* **Playlists Fragment** – UI to list playlists and their contents. Possibly combine with library UI or separate screen. *Language:* Kotlin. *Environment:* Android SDK.

- *Task:* **Settings Activity** – Preferences (use PreferenceFragmentCompat for convenience). Include toggles for features (enable visualization, voice control, etc.), and possibly a way to log in for cross-device sync if needed. *Language:* Kotlin. *Environment:* Android SDK.

- **Core Integration via JNI:**

- *Task:* **JNI Bridge Class** – Write a Kotlin `MediaPlayerNative` object that loads the native library (`System.loadLibrary("mediaplayer")`) and declares native methods for each core engine API function (e.g., `nativeOpen(String path)`, `nativePlay()`, `nativePause()`, etc.). *Language:* Kotlin (native method stubs) + C++ (JNI impl). *Environment:* Android NDK for C++ implementation, Android SDK for Kotlin side.
- *Task:* **JNI Implementations** – In C++ (within the core or separate JNI glue file), implement the JNI functions to call the actual core engine methods. For example, `Java_com_example_mediaplayer_MediaPlayerNative_nativePlay(JNIEnv*, jclass)` will call our `MediaPlayer::play()`. *Language:* C++17. *Environment:* Android NDK (CMake builds it).
- *Task:* **Thread Management** – Ensure JNI calls that may block (like open, or long processing) are done on background threads, not the UI thread. Use Kotlin coroutines or AsyncTask (deprecated) to call those JNI methods asynchronously. *Language:* Kotlin. *Environment:* Android SDK.

- *Task:* **Callbacks from Native** – Implement a mechanism for the native code to notify the Kotlin code of events (like playback complete, position changed). Options: use JNI to call Java methods or use

Android NDK's AAudio callbacks. E.g., define a Java interface for listeners and call via JNI (env->CallStaticVoidMethod on a stored global ref). Implement in MediaPlayerNative. *Language:* C++ (JNI) + Kotlin. *Environment:* Android SDK/NDK.

- **Android-Specific Features:**

- *Task:* **MediaSession & Notification** – Integrate Android MediaSession API for lock screen controls and notifications. Create a MediaSession in Kotlin, tie it to playback actions (so headphone controls and car playback controls work). Show a foreground service notification with play/pause, skip controls while music is playing. *Language:* Kotlin. *Environment:* Android SDK (MediaSessionCompat, NotificationCompat).
- *Task:* **Permission Handling** – Request storage permission (if needed for scanning media on device, depending on Android version and scoped storage rules). Possibly use Storage Access Framework for user to pick media directories. *Language:* Kotlin. *Environment:* Android SDK.
- *Task:* **Gesture Detection** – Implement touch gestures in the UI: e.g., swiping album art to next track, or swiping up/down for volume/brightness (common in video players). Use Android View touch listeners or gesture detectors. *Language:* Kotlin. *Environment:* Android SDK.

- *Task:* **Voice Control (Android)** – (See Voice Control Module, but integration here) Possibly use Android's SpeechRecognizer API for voice commands (if not using our internal Vosk directly in native). If using Vosk in native, then from UI trigger the native listening and then handle results. Or use Google Voice UI (ACTION_RECOGNIZE_SPEECH intent) for a quick solution. *Language:* Kotlin (for invoking voice input UI). *Environment:* Android SDK (SpeechRecognizer).

- **Testing on Device:**

- *Task:* **Instrumented UI Tests** – Write an Espresso test that opens the app, simulates a click on a song, and verifies playback started (perhaps via a stub or by checking notification or a playing indicator). *Language:* Kotlin. *Environment:* Android SDK (with Espresso).
- *Task:* **Compatibility Testing** – Ensure the app works on various Android versions (at least 8.0+). Use emulators or Firebase Test Lab for different API levels and device sizes. *Environment:* Android (emulators).

(The Android app will feel native, with Material Design components, while leveraging the powerful C++ core via JNI. MediaSession integration ensures it behaves like a proper Android music app with lock-screen controls.)

iOS App (Swift UI/UIKit + Core bridging)

Description: The iOS app will be implemented in **Swift** (with SwiftUI for modern UI or UIKit if needed for finer control). It will integrate the core C++ engine via a bridging header and Obj-C++ wrapper (since Swift can call C functions or Obj-C classes, and C++ can be wrapped in Obj-C++). The app will provide a clean iOS-native interface for library, playback, etc., consistent with iOS Human Interface Guidelines. Estimated ~6k LOC. ~30 tasks.

- **Project Setup:**

- *Task:* **Create Xcode Project** – New iOS project (e.g., using SwiftUI lifecycle for latest iOS). Set bundle id, deployment target (iOS 14+). *Language:* Swift. *Environment:* **Xcode** on macOS (ensure command-line tools installed for builds in automation).

- *Task:* **Include Core Engine** – Add the compiled core library (as static library or framework) to Xcode. Alternatively, compile core for iOS (arm64) as part of the project using a CMake ExternalProject or include source directly. Create a bridging header to expose needed C++ functions to Swift (via Obj-C). *Language:* Objective-C++ (for wrapper). *Environment:* Xcode with C++ support. Steps: ensure build settings allow mixed Swift/ObjC++.

- **UI Implementation (SwiftUI or UIKit):**

- *Task:* **Main UI (SwiftUI ContentView)** – Create a SwiftUI view for the main screen with playback controls, similar to the design on other platforms. SwiftUI can declaratively bind to state (which we will update from the core via a ViewModel class). Alternatively, if using UIKit: design a UIViewController with buttons, sliders, etc. *Language:* Swift (SwiftUI DSL). *Environment:* Xcode.
- *Task:* **Library List View** – SwiftUI List to show songs/videos. For data, likely use a view model that fetches from core (perhaps preloaded into an array). Possibly use `List` with search bar (UISearchBar via UIViewRepresentable). *Language:* Swift. *Environment:* Xcode.
- *Task:* **Now Playing View** – Shows current track and controls in SwiftUI. Include an Image for album art or video thumbnail, Text for title/artist, and playback controls. *Language:* Swift. *Environment:* Xcode.
- *Task:* **Navigation and Tab Bar** – Use a TabView for switching between Library and maybe a Settings tab. Or a NavigationView for drilling down into playlists, etc. *Language:* SwiftUI. *Environment:* Xcode.

- *Task:* **Settings UI** – SwiftUI Form for settings toggles (Dark mode, enable AI tagging, etc.). *Language:* Swift. *Environment:* Xcode.

- **Core Integration:**

- *Task:* **Bridging Header and Wrapper** – Write an Objective-C++ `.mm` file that includes core engine headers and implements Objective-C callable functions or a class. For example, create `MediaPlayerBridge` Objective-C class with methods +functions like `-(instancetype)init` (to create core MediaPlayer), `-(void)open:(NSString*)path`, `-(void)play`, etc. These methods call the C++ core. *Language:* Obj-C++ / C++. *Environment:* Xcode (enable C++17 in compiler flags).
- *Task:* **Expose to Swift** – Use a bridging header to expose the Objective-C interface to Swift. Then in Swift code, instantiate `MediaPlayerBridge` and call its methods for playback control. *Language:* Swift + Obj-C. *Environment:* Xcode.
- *Task:* **Callbacks from Core** – Similar to Android, implement callbacks. Perhaps have the Obj-C++ wrapper post NSNotifications or use delegation. E.g., when core signals track end, the wrapper could call `[NSNotificationCenter defaultCenter] postNotificationName:@"MediaPlayerTrackEnded" …]`. SwiftUI ViewModels can subscribe to these notifications to update state. *Language:* Obj-C++ and Swift. *Environment:* Xcode.

- *Task:* **Data Flow** – Implement a ViewModel (using ObservableObject in Swift) that holds current playback info state (e.g., currentTime, isPlaying). The ViewModel calls the bridge for actions and listens for notifications for updates (updating @Published properties accordingly, which SwiftUI UI binds to). *Language:* Swift. *Environment:* Xcode.

- **iOS Specific Features:**

- *Task:* **AVAudioSession Handling** – Configure AVAudioSession in AppDelegate/SceneDelegate to allow background audio playback. Set category to .playback and handle interruptions (phone calls) to pause/resume via core. *Language:* Swift. *Environment:* Xcode, iOS frameworks.
- *Task:* **Remote Command Center** – Integrate with MPRemoteCommandCenter so that control center (lock screen) has play/pause/next controls working. Update now playing info via MPNowPlayingInfoCenter with current track info (title, artwork). *Language:* Swift. *Environment:* iOS MediaPlayer framework.
- *Task:* **Gesture Support** – On iOS, e.g., a swipe gesture on album art to skip, or shake device for some action (if desired). Use SwiftUI gestures or UIKit gesture recognizers. *Language:* Swift. *Environment:* Xcode.

- *Task:* **Siri / Voice** – Optionally, add Siri integration (SiriKit Media Intents) to control playback via voice. This requires defining Siri intents for play/pause, etc. *Language:* Swift. *Environment:* Xcode with SiriKit. (Optional advanced feature.)

- **Testing:**

- *Task:* **Unit Tests (Swift)** – Test the ViewModel logic by injecting a dummy MediaPlayerBridge that simulates core responses. Ensure that play/pause toggle updates published state. *Language:* Swift (XCTest). *Environment:* Xcode.
- *Task:* **UI Tests (XCTest/UIAutomation)** – Automate tapping a sample track and verifying the now playing screen shows correct title. *Language:* Swift. *Environment:* Xcode (Simulator).

(The iOS app remains native in look-and-feel, using SwiftUI for a modern interface. By bridging to the C++ core, we avoid duplicating playback logic. The app integrates with iOS system features like Control Center and Siri for a seamless experience.)

Gesture & Voice Control Module

Description: This module encompasses user input beyond standard UI clicks: specifically, touch/mouse gestures and voice commands. Gesture control includes things like swiping the screen to change tracks or pinching for zoom (in video), as well as possibly mouse gestures on desktop (VLC supports mouse gestures for control [9]). Voice control allows the user to control playback with spoken commands (play, pause, skip, search by saying a song name, etc.). We will implement voice control using an **offline speech recognition** engine to avoid external dependencies – e.g., **Vosk API**, which supports 20+ languages and runs on-device (Android, iOS, desktop) [10]. Estimated ~3k LOC, ~15 tasks, plus usage of large external models (not counted in LOC).

- **Gesture Control:**
- *Task:* **Desktop Mouse Gestures** – Implement simple mouse gesture recognition on desktop. E.g., if user right-click-drags left (a "←" gesture) while video playing, interpret as "previous track"; drag right "→" as "next track"; an upward gesture maybe as volume up, etc. Use Qt's mouse events to track gesture paths and recognize a few simple patterns. *Language:* C++ (Qt). *Environment:* Qt dev.
- *Task:* **Mobile Touch Gestures** – Implement common gestures in mobile UIs: in Android and iOS UI code, add swipe detection on the Now Playing view (left/right for prev/next track), swipe up/down for

volume or brightness (for video). Use platform gesture recognizers (Android GestureDetector, iOS SwiftUI DragGesture). *Language:* Kotlin, Swift. *Environment:* Android Studio, Xcode.
- *Task:* **Shake to Shuffle (Mobile)** – (Fun optional) On iOS, use motion events to detect shake gesture to shuffle playback. On Android, use accelerometer sensor via SensorManager. *Language:* Swift, Kotlin. *Environment:* iOS/Android.

- *Task:* **Custom Gesture Mapping UI** – (Optional advanced) Provide settings where user can enable/disable certain gestures. For now, likely static, so skip UI.

- **Voice Control (Speech Recognition):**

- *Task:* **Integrate Vosk Speech Library (C++ or Python)** – Include Vosk (speech-to-text) offline engine. For example, use the Vosk C++ API or Python wrapper to perform recognition. We might run a lightweight model (50MB) for commands [11]. *Language:* C++ (for integration) or Python (if using a separate service). *Environment:* If C++ – include Vosk library and models in project (requires linking Kaldi-based libs). If Python – ensure Python env with `vosk` pip package available.
- *Task:* **Voice Command Grammar** – Define a limited vocabulary/grammar of commands: e.g., "play", "pause", "next track", "previous track", "volume up/down", "shuffle on/off", "play song <name>", etc. In Vosk, you can specify a list of key phrases for better accuracy [12]. Configure the recognizer with these. *Language:* Depends (could be JSON config to Vosk). *Environment:* C++ or Python environment with model.
- *Task:* **Microphone Capture (Desktop)** – Implement code to capture microphone audio from user when voice control is activated (e.g., user presses a voice button). On desktop, use PortAudio or Qt Multimedia to get audio input. Feed it to Vosk for processing. *Language:* C++17. *Libraries:* PortAudio or Qt Multimedia. *Environment:* C++ dev (install `portaudio-dev` if using PortAudio).
- *Task:* **Voice Input (Mobile)** – On Android, optionally start a recognition using Vosk's Android API (they provide a Gradle dependency). Or use native SpeechRecognizer (less ideal as it needs internet or Google services for offline packs). On iOS, use Vosk via an available iOS framework or Apple's Speech framework (which can do on-device in newer iOS). For consistency, if using Vosk, integrate it similarly. *Language:* Kotlin/Swift. *Environment:* Android with Vosk .aar, iOS with Vosk via swift package or use Speech.framework for a simple approach.
- *Task:* **Voice Command Processing** – Once text is recognized, map it to an action. E.g., recognized text "play next song" → call core.next(); "pause" → core.pause(); "play {song}" → search library for {song} and play if found. Implement this mapping logic in whatever part receives the recognition result (could be in C++ or at a higher level in each app). *Language:* C++ (if recognition done in core) or Kotlin/Swift (if done in app). *Environment:* Corresponding dev env.

- *Task:* **Feedback & Error Handling** – Provide some UI feedback when voice command is listening (e.g., a waveform or a beep) and if command not understood, show a message. *Language:* QML/Kotlin/Swift for UI feedback. *Environment:* All platforms UI.

- **Testing Voice/Gesture:**

- *Task:* **Simulate Voice Commands** – For testing, create a small script or use recorded audio samples of someone saying commands, and feed them to the recognizer to ensure correct action triggers. *Language:* Python (to feed audio to Vosk and verify output). *Environment:* Python with vosk installed, plus sample WAV files.

- *Task:* **Gesture Unit Tests** – (Where possible) Simulate gesture inputs. On Qt, perhaps simulate a sequence of mouse move events to mimic a gesture and verify action triggered. *Language:* C++ (Qt test framework) or manual testing. *Environment:* Qt test environment.

(Offline voice control via Vosk ensures privacy and availability without internet [11]. Combined with intuitive gesture controls (as also seen in VLC's gesture and hotkey interfaces [9]), this module makes the player more accessible and modern. Users can say "Pause music" or swipe to navigate, enhancing usability.)

AI Tagging Service (AI Metadata Module)

Description: The AI Tagging module automatically analyzes media (audio/video) and generates metadata tags: e.g., music genre detection, mood classification for songs, object and scene recognition in videos, speech transcription, etc. This enriches the media library with searchable tags without manual input [13]. To keep the system modular and leverage high-level frameworks, this may be implemented as a separate Python service or a C++ module using an inference engine (like ONNX Runtime). Given the complexity of AI models, we outline it as a separate component that communicates with the core (possibly via an API or by writing tags to the library DB). Estimated ~4k LOC (glue code; heavy lifting by libraries/models). ~15 tasks.

- **Setup & Architecture:**
- *Task:* **Choose AI Framework** – Decide on using Python (with libraries like PyTorch, TensorFlow) versus C++ (with ONNX Runtime or OpenCV's DNN). Python allows easier use of pretrained models; we'll proceed with a Python service for flexibility. Document this decision and ensure core can interface (maybe via a local HTTP or gRPC). *Language:* (Documentation/plan decision) – we choose **Python** for now.

- *Task:* **Environment for AI** – Prepare a Python environment with needed libraries: e.g., `pip install torchaudio torchvision transformers onnxruntime`. Also download any pre-trained models (e.g., a music genre classification model, an image recognition model, etc.). *Environment:* **Python 3.10** container – install PyTorch (CPU version), etc.

- **Audio Analysis:**

- *Task:* **Music Genre Classification** – Use an open model or library to classify music genre from audio. For instance, use `torchaudio` pipelines or a pre-trained model (like musicnn or an MIR library). Implement a function that takes an audio filepath (or array of features) and returns genre/mood tags (e.g., "rock", "classical", "happy"). *Language:* Python. *Libraries:* torchaudio, numpy. *Environment:* Python container with torchaudio. Possibly download a pretrained model (e.g., an .onnx or a PyTorch state dict for a music tagger).
- *Task:* **Audio Fingerprinting (Song ID)** – (Optional) Integrate an audio fingerprinting to identify songs (like acoustic fingerprint to match a database). There are open source like dejavu or Chromaprint. This could allow identifying track name if metadata missing. *Language:* Python. *Libraries:* AcoustID/ Chromaprint. *Environment:* Python with `pyacoustid` or similar.
- *Task:* **Speech-to-Text (for Videos)** – Use a speech recognition model (like Vosk or wav2vec) to transcribe speech in videos (or podcasts) to generate subtitle or keyword tags. *Language:* Python. *Libraries:* vosk (if not already used), or HuggingFace transformers with Wav2Vec2. *Environment:* Python with model downloaded.

- *Task:* **Mood Detection** – Implement or use a model to predict the mood of a song (happy, sad, energetic, etc.). Some research models exist; perhaps fine-tune or use an existing one. *Language:* Python. *Environment:* Python with ML libs. (This might overlap with genre classification model if multi-label tagging model is used.)

- **Video Analysis:**

- *Task:* **Object & Scene Recognition** – Use a computer vision model (e.g., pre-trained on ImageNet or COCO) to identify objects or scenes in video frames. Implement by extracting key frames (maybe one every X seconds) and running them through a model like ResNet or YOLO for objects. Collect labels (e.g., "beach", "car", "dog"). *Language:* Python. *Libraries:* OpenCV (for frame extraction), a deep learning model (TensorFlow or PyTorch with a pretrained CNN). *Environment:* Python with OpenCV and TF. Possibly use ONNX for a lightweight solution.
- *Task:* **Face Recognition (Optional)** – If applicable, detect if videos have faces of known people (could tie into a celebrity recognition API or just label "people present"). This likely out of scope unless we integrate an API, but mention for completeness. *Language:* Python.

- *Task:* **Video Scene Segmentation** – (Optional) Determine scene changes or highlights in videos, maybe to tag "scene 1: outdoor", "scene 2: indoor" etc. Could be done via a combination of object detection results.

- **Integration & Data Flow:**

- *Task:* **Service API** – Create a simple API for the AI tagging service. For instance, a REST endpoint (using Flask/FastAPI) or a CLI that core can call. Example: core calls `aitagging.tag_file(file_path)` either via Python bindings or HTTP; the service returns a list of tags/metadata. *Language:* Python (Flask) or even just a command-line interface. *Environment:* Python.
- *Task:* **Asynchronous Tagging** – Tagging can be slow (especially on large files or using deep models). Implement it asynchronously: e.g., when new media added to library, spawn a background tagging job (either a separate thread/process or queue in the AI service). The results can be stored in the DB when ready. *Language:* Python (multiprocessing or threading). *Environment:* Python.
- *Task:* **Core–AI Communication** – Implement in core (C++) a way to send files for tagging and receive results. This could be as simple as launching the Python script with file path and getting output JSON. Or if using an HTTP service, use libcurl in core to HTTP POST and parse JSON response. *Language:* C++ (for integration) + Python. *Libraries:* nlohmann::json (to parse JSON in C++) if needed. *Environment:* C++ dev (with libcurl), Python service running.

- *Task:* **Update Library with Tags** – Once tags are obtained (e.g., "genre: Rock", "mood: Happy", "objects: beach"), update the media's entry in the library database (possibly storing tags in a separate table or as a JSON blob). This allows these tags to be used in search or smart playlists (e.g., playlist of all "Happy" mood songs). *Language:* C++ (library module). *Environment:* C++ dev.

- **Testing AI Tagging:**

- *Task:* **Test on Sample Media** – Prepare a couple of media files (one song, one video) and run the tagging pipeline. Verify that sensible tags are produced (manually check). This ensures the models

and integration are working. *Language:* Python test script. *Environment:* Python with models, and maybe a known song (e.g., a classical music file should be tagged "Classical").
- *Task:* **Performance Check** – Time the tagging of a file and log resource usage. Ensure it's not unreasonably slow. If it is, consider using smaller models or limiting analysis length (e.g., analyze first 30 seconds of a song). *Environment:* Python.

(AI auto-tagging will generate valuable metadata automatically [13] *. This is forward-looking: for example, tagging videos with objects and scenes can make search much more powerful ("show me videos with beach scenes"). The design keeps the AI analysis separate from core playback to maintain clean modularity and the ability to scale or disable it as needed.)*

Streaming & Networking Module

(Note: Some streaming tasks were already touched in core, but here we consolidate network-related features, including cross-device sync below. This module focuses on connectivity and streaming service integration.)

Description: This covers networking capabilities beyond basic file access: streaming from online sources and syncing or casting to devices. It includes support for standard streaming protocols (handled via core/FFmpeg), as well as potential integration with streaming services or remote control protocols. Estimated ~2k LOC (excluding reused core parts). ~10 tasks.

- **Online Streaming Features:**
- *Task:* **Open URL UI** – Add UI in desktop/mobile apps for "Open Stream URL", where user can input an http/rtsp link. Ensure that is passed to core (core can handle http via FFmpeg). *Language:* QML/Kotlin/Swift UI. *Environment:* respective UI envs.
- *Task:* **YouTube DL Integration** – (If not done in core) Implement a feature where a YouTube link can be entered, and the app will resolve it to actual stream URL (using youtube-dl). Possibly done by calling a Python script from the app. *Language:* Depends (maybe Python script). *Environment:* Ensure youtube-dl installed, accessible.
- *Task:* **Local DLNA/UPnP Support** – (Optional) Allow discovering UPnP/DLNA media servers on local network and listing media from them. This could use a library like Platinum UPnP or similar. *Language:* C++ (or Python). *Libraries:* MiniDLNA client or Qt UPnP if any. *Environment:* C++ dev with network libs.

- *Task:* **HTTP Server for Remote Control** – (Bridging to cross-device sync) perhaps run a small HTTP server in the app that can accept remote commands (play/pause, etc.) or serve the current playlist. This could be used by a companion app for remote control. *Language:* C++ (e.g., using civetweb or QtNetwork) or Python (Flask). *Environment:* C++ dev or Python.

- **Cross-Device Sync (Playback Sync):**
  Description: Cross-device sync lets a user pause on one device and resume on another, or remote-control one device from another. Inspired by features like Plex Sync or Spotify Connect. This likely requires devices to share state via a network. We'll implement a simple approach: if devices are on the same network, they discover each other (via mDNS/Bonjour) and then use a simple API (HTTP or a custom protocol) to share playback status. For truly remote (different networks), a cloud server could be used, but that's beyond scope here (one could self-host a small server to relay messages). Estimated ~3k LOC included above.

- *Task:* **Discovery (mDNS)** – Use multicast DNS to advertise presence of a device running the player. E.g., each instance advertises a service "MediaPlayer" with its IP/port. Use an mDNS library (Apple's Bonjour API on iOS/macOS, Avahi on Linux, or a cross-platform like `cpp-mdns`). *Language:* C++ for core service or platform-specific (Bonjour in Obj-C, NSNetService). *Environment:* For iOS/mac, use built-in Bonjour (no addl install); for Linux/Win, possibly include a small mDNS responder library.

- *Task:* **Announcement & Discovery Handlers** – When a new device is discovered, notify the UI (e.g., show an icon "Device available: LivingRoomPC"). Conversely, show a list of discovered devices in a "Devices" menu. *Language:* C++ core + UI integration. *Environment:* C++ dev plus UI.
- *Task:* **Device Info Exchange** – Define a small info payload (e.g., device name, current media, current position) that devices can query from each other. Could be via a REST GET on the advertised port (e.g., `GET /status` returns JSON with current track and position). *Language:* C++ or Python (for server). *Environment:* C++ dev with libmicrohttpd or Qt HTTP server; or Python Flask if used for control server.
- *Task:* **Sync Playback Position** – Implement function to send current playback state to another device. For example, if user chooses "Send to Device X", the app will POST the current media ID and position to Device X's `/play` endpoint. Device X then loads that media (if available locally or streams if possible) and seeks to that position, starting paused or playing as requested. *Language:* C++ (using HTTP client) on sender, and C++ (server handler) on receiver. *Environment:* C++ dev with libcurl (for sending) and whichever server lib (for receiving).

- *Task:* **Cloud Sync Option** – (Optional, advanced) Provide a way to sync via cloud by using a simple central server to store last played position per user. This requires user login. Could use a free service or self-host. Possibly skip due to complexity, but leave code hooks (like if user logged in, sync to cloud through an API). *Language:* C# or Node.js (if we had server) – out of scope to implement fully, but design is noted.

- *Task:* **Testing Sync** – Test with two instances: play on one, then attempt to sync to another and verify the second jumps to same time. Also test discovery works across OS (may need all on same LAN, ensure firewall open). *Environment:* Two devices/emulators on LAN.

(Cross-device resume is akin to Brightcove's XDR feature which lets you continue on another device where you left off [14]. Our implementation uses local network discovery and direct communication to avoid requiring external servers. It's a modular add-on that can be disabled if not needed.)

DevOps & Infrastructure

Description: Tasks for setting up development environments, build automation, and continuous integration. This ensures each parallel task has a reproducible environment and that the entire project can be built and tested across platforms.

- **Development Containers and Environment Scripts:**
- *Task:* **Dockerfile for Core/C++** – Write a Dockerfile for a container that has C++ build essentials, CMake, FFmpeg dev, TagLib, SDL/Qt if needed for headless builds, etc. This will be used by Codex agents for core engine tasks. *Environment:* Docker with base image (e.g., ubuntu:22.04). Steps: `apt-get update && apt-get install -y build-essential cmake libavcodec-dev`

`libavformat-dev libswresample-dev libsqlite3-dev libtag1-dev libpulse-dev libass-dev`.

- *Task:* **Dockerfile for Qt** – Container with Qt 6 SDK and Xvfb (for any GUI build tests). This is heavier (Qt binaries ~). Alternatively, use a pre-made Qt Docker image. *Environment:* Docker – base could be `qt:6.5` image or install Qt via Qt online installer in non-GUI mode.
- *Task:* **Android Build Environment** – Docker image or script that sets up Android SDK and NDK. E.g., use `openjdk:11` base, install commandline-tools, use sdkmanager to install platform-tools, platforms (android-33), build-tools, ndk. *Environment:* Docker with Android SDK.

- *Task:* **iOS Build Setup** – (Cannot easily dockerize since Xcode requires macOS). Use a MacOS runner with Xcode installed. Possibly a script to setup any Ruby cocoapods if needed (if using pods, but likely not). *Environment:* MacOS VM/CI runner – ensure Xcode command-line tools present, and a script to build iOS project via xcodebuild.

- **Continuous Integration (CI):**

- *Task:* **GitHub Actions CI Workflow** – Create a YAML workflow that builds the project for all platforms: use matrix to cover (linux, windows, mac) for core and maybe desktop app; separate jobs for Android (using macos or ubuntu with Android SDK) and iOS (macos runner). *Environment:* CI (GitHub Actions). Steps: checkout code, set up environment (like use setup-msbuild for Windows, brew install dependencies on Mac, docker run for Linux builds), then build via CMake/Gradle/Xcodebuild.
- *Task:* **Automated Tests in CI** – Integrate test execution in the CI pipeline: run unit tests after building core (e.g., ctest for C++ tests), run Android instrumentation tests on emulator (maybe use headless emulator or Firebase Test Lab), run iOS tests on a simulator (xcodebuild test). *Environment:* CI runners with appropriate setups.
- *Task:* **Static Analysis & Lint** – Set up tools: e.g., run `clang-tidy` or `cppcheck` on C++ code, Android Lint on Android code, SwiftLint for iOS. Add this to CI to maintain code quality. *Environment:* CI (install tools in pipeline steps).

- *Task:* **Code Formatting** – Provide a clang-format configuration and maybe a Prettier config for QML, etc., and have CI check formatting (or autoformat) to keep code style consistent. *Environment:* local dev/CI.

- **Repository Management:**

- *Task:* **Git Submodules for Libraries** – If using submodules (e.g., for projectM or TagLib if not using system libs), set them up and document how to update. *Environment:* Git.
- *Task:* **Issue Templates and Contribution Guide** – Create docs (CONTRIBUTING.md) outlining how to set up dev environment and how tasks are structured (for future contributors or AI agents). *Environment:* Markdown docs in repo.
- *Task:* **Merge Strategy for AI Agents** – Define how parallel contributions will merge (maybe use a script to auto-merge if no conflict, or have a nightly integration build that merges stubs with implementations). *Environment:* documentation.

Testing & Quality Assurance

Description: Outline of testing tasks (unit tests, integration tests, user acceptance tests) to ensure each module works correctly and the whole system is reliable.

- **Unit Testing (Core):**
- *Task:* **Core Unit Tests Setup** – Set up Google Test (or Catch2) in the project. Write basic tests for core components: e.g., test that opening a known media file returns correct duration, test that pausing actually stops progress, test format conversion on a short audio clip produces expected output file. *Language:* C++ (tests). *Environment:* C++ dev with GoogleTest (`apt-get install libgtest-dev` and compile).
- *Task:* **Library Module Tests** – Write tests for library database operations (using an in-memory or temp DB): adding media, searching, smart playlist filtering results. *Language:* C++ or Python (could also test via Python by opening DB file). *Environment:* C++ test.

- *Task:* **Visualization Tests** – Hard to fully automate visual output, but test that feeding a known sine wave PCM to visualizer produces any non-zero output or triggers render calls (could hook into projectM's PCM and ensure no error). *Language:* C++. *Environment:* Possibly manual check.

- **Integration Testing:**

- *Task:* **Playback Integration Test** – A test that launches the core engine in a separate thread with a known test media file (e.g., a small WAV or MP4 included in tests), plays it for a few seconds, and verifies that playback state changes to Playing, then to Ended at file end. This could be part of a GoogleTest or a separate harness. *Language:* C++. *Environment:* Possibly needs a way to run without actual audio device (use a dummy audio output that just discards data for test).

- *Task:* **End-to-End Script** – Write a Python or shell script that simulates a user scenario: add a folder to library, play a song, seek, pause, etc., by driving either the UI via automation (for instance, using Qt's QTest to simulate button clicks or Android UIAutomator). This can catch high-level integration issues. *Language:* Python (for desktop using something like PyAutoGUI) or use platform-specific automation. *Environment:* For Android, maybe use ADB commands to simulate input if possible; for desktop, maybe just trust unit tests.

- **Performance Testing:**

- *Task:* **Startup Time Test** – Measure how quickly the app opens and is ready to play, especially on low-end device. Possibly instrument the code to log timestamps. Ensure under a threshold (e.g., <2s on desktop). *Environment:* Any (manual measure or simple logs).
- *Task:* **Memory/CPU Profiling** – Run the app with a large playlist or 4K video and monitor memory usage, CPU usage. Identify any leaks or bottlenecks. Use Valgrind (for memory) on Linux, Instruments on macOS, Profiler on Android. *Environment:* platform-specific.

- *Task:* **Battery Test (Mobile)** – Play a video on Android/iOS for X minutes and measure battery drop, ensuring no excessive background activity beyond decoding. *Environment:* Manual or using power profiling tools (Android Profiler).

- **User Acceptance and Edge Cases:**

- *Task:* **Formats Compatibility Test** – Create a list of various media formats (mp3, flac, mp4, mkv, avi, etc.), attempt to play each on each platform. Verify playback works or at least fails gracefully with an error dialog if unsupported (shouldn't happen if FFmpeg covers it). *Environment:* Manual or small automated script.

- *Task:* **UI Responsiveness Test** – Verify the UI remains responsive during heavy operations (scanning library, downloading stream, etc.). If not, identify where to add threading. *Environment:* Manual.

- **Regression Testing:**

- *Task:* **Automate Regression Suite** – As features stabilize, maintain a suite of tests that must pass before each release (perhaps a combination of above tests). Set up CI to run this suite on pull requests. *Environment:* CI.

(Testing ensures reliability – from unit tests for small components to integration tests that mimic real usage. The modular design also helps test components in isolation, e.g., one can test the core engine with synthetic inputs without the UI.)

Sources: Key design choices were informed by industry practices and analogous projects. For instance, using FFmpeg's libavcodec is a proven method to support numerous codecs in media players [2]. The overall modular architecture (plugins for codecs, outputs, etc.) is inspired by VLC's design [1], ensuring future extensibility. TagLib was chosen for metadata management due to its efficiency and broad format support [5]. For visualizations, projectM offers cross-platform, music-reactive graphics [7]. Voice control leverages Vosk for offline recognition, which supports many languages and runs on mobile/embedded devices [10]. AI auto-tagging aligns with modern content management trends [13]. Cross-device sync follows patterns used in streaming services (e.g., Brightcove XDR) to resume playback on different devices [14]. All these choices contribute to a robust, future-proof media player architecture. Each task above can be executed in parallel, given the clearly defined module boundaries and environment setups, to rapidly develop the system using AI coding agents. [2] [1] [5] [7] [10] [13] [14]

---

[1] [8] [9] VLC media player - Wikipedia
https://en.wikipedia.org/wiki/VLC_media_player

[2] How do VLC and ffmpeg work together? - Stack Overflow
https://stackoverflow.com/questions/11525769/how-do-vlc-and-ffmpeg-work-together

[3] [4] GitHub - media-kit/media-kit: A cross-platform video player & audio player for Flutter & Dart.
https://github.com/media-kit/media-kit

[5] [6] TagLib | TagLib
https://taglib.org/

[7] GitHub - projectM-visualizer/projectm: projectM - Cross-platform Music Visualization Library. Open-source and Milkdrop-compatible.
https://github.com/projectM-visualizer/projectm

10 11 12 VOSK Offline Speech Recognition API
https://alphacephei.com/vosk/

13 AI Auto-Tagging: Essential for Efficient Content Management
https://www.veritone.com/blog/ai-auto-tagging/

14 Overview: Cross-Device Resume - Xdr - Brightcove APIs
https://apis.support.brightcove.com/xdr/getting-started/xdr-overview.html