

2. Software Requirements

2.1. Tools and Technology

To build our karaoke-integrated music application with sharing capabilities, we selected a set of robust, well-supported Android tools and libraries that together provide IDE support, a modern language, backend services, cloud media management, and efficient media handling.

2.1.1. Android Studio

Android Studio is Google's official IDE for Android development, built on JetBrains' IntelliJ IDEA. It offers a Gradle-based build system, a visual layout editor with real-time previews, integrated emulators for multiple device profiles, and comprehensive profiling tools (CPU, memory, network). These features streamline coding, debugging, UI design, and performance tuning in a single environment [8].

2.1.2. Kotlin

Kotlin is a statically typed JVM language officially supported by Google for Android development since 2017. It reduces boilerplate through type inference, data classes, and extension functions, and enforces null-safety to prevent common runtime errors. Kotlin's coroutine framework greatly simplifies asynchronous tasks—such as network calls and media processing—by allowing sequential code style without blocking the main thread. Its full interoperability with Java also facilitates gradual migration of existing codebases [9].

2.1.3. Firebase

Firebase is Google's Backend-as-a-Service platform. We use Firebase Authentication for secure user sign-in, Cloud Firestore for real-time, NoSQL data storage of user profiles, playlists, and performance metadata, and Cloud Storage for hosting recorded audio and image files. Firestore's offline support and real-time synchronization ensure that social feeds and collaborative playlists remain up-to-date across devices [10].

2.1.4. Cloudinary

Cloudinary provides managed media workflows with its Android SDK, offering

uploads, storage, on-the-fly transformations (format conversion, cropping), and global CDN delivery. By offloading media processing to Cloudinary, our app efficiently handles user-generated audio and image assets, applies optimized transformations, and leverages built-in caching for low-latency playback and display [11].

2.1.5. Glide

Glide is an open-source image loading and caching library for Android. It asynchronously fetches, decodes, and displays images (including animated GIFs and video thumbnails), managing bitmap pooling and memory to minimize `OutOfMemoryErrors`. Glide's integration into `RecyclerViews` and other scrolling containers makes it ideal for displaying album art and user avatars in our social feed with smooth scrolling performance [12].

2.1.6. Media3 ExoPlayer

Jetpack Media3's ExoPlayer is a highly extensible media playback library that supports DASH, HLS, and SmoothStreaming, adaptive bitrate switching, and custom renderers. We leverage the media3-exoplayer and UI modules (XML and Compose) to implement our karaoke engine—mixing background tracks, user recordings, and synchronized lyrics—while maintaining low latency and precise audio synchronization [13].

2.2 Functional Requirements

2.2.1 Authentication and Authorization

The application implements a secure, Firebase-based authentication and authorization system to manage user identities and control access to personalized features. New users register by providing an email address, password, and a unique username; additional profile details (e.g., display name and profile picture) may be added after account creation. Upon registration, Firebase Authentication issues a secure token that the app uses to manage sessions, ensuring that only authenticated users can access karaoke recording and sharing functionality. A “Forgot Password” workflow allows users to reset their credentials via an emailed link, while email verification ensures account validity before users can publish any performance. All profile updates—including avatar changes or edits to display name—are immediately synchronized to Firestore, preserving data consistency and securing access through Firebase Security Rules.

2.2.2 Music Playback

Users can seamlessly browse and stream background tracks through an ExoPlayer-powered playback engine. The app presents a searchable catalog of licensed instrumental and full-track audio, allowing filtering by title, artist, or genre. Playback controls—including play/pause, seek, volume adjustment, and shuffle/repeat modes—are integrated into both the main player UI and the karaoke interface. To support offline use, users may download select tracks; these are cached locally and play without network connectivity. ExoPlayer’s adaptive bitrate streaming ensures uninterrupted playback even under fluctuating network conditions, while the UI displays buffering states and network status to keep users informed.

2.2.3 Karaoke Recording & Performance

In karaoke mode, the app overlays synchronized lyrics on the screen, time-coded to the currently playing track. When users initiate recording, the system captures microphone input in parallel with the background audio using Media3 APIs. After recording, the app automatically mixes the vocal track with the instrumental, producing a single combined audio file. Before upload or local save, users may preview their performance, re-record if desired, or trim the beginning and end of the track. The final mix, along with accompanying metadata (track ID, duration, timestamp), is prepared for cloud upload.

2.2.4 Media Upload & Storage

To handle user-generated content, the application integrates Cloudinary’s Android SDK for secure, efficient media management. Upon user confirmation, mixed audio files and optional performance snapshots are uploaded directly to Cloudinary, with real-time progress indicators and automatic retries in case of network interruption. Cloudinary transforms and optimizes assets on the fly—converting audio to the appropriate format and resizing images for thumbnails—before delivering them through a global CDN. The resulting URLs and associated metadata are stored in Firestore, enabling instant access and playback across all of the user’s devices.

2.2.5 Performance Sharing & Social Feed

Authenticated users can publish their recorded performances to a personalized social feed. Shared items display a thumbnail, title, and brief description, and are immediately visible

to the user's followers. Viewers may engage by liking, commenting, or re-sharing to their own feeds, with engagement counts updated in real time via Firestore listeners. Each performance includes visibility settings—public, friends-only, or private—allowing users to control their audience. Notifications of new likes or comments are delivered through Firebase Cloud Messaging, ensuring prompt feedback and fostering community interaction.

2.2.6 Playlist Management

The application allows users to create, edit, and organize custom playlists of background tracks or favorite performances. Playlists can be assembled directly from the player interface or social feed, with drag-and-drop reordering supported in the playlist editor. Once created, playlists may be launched in sequence for consecutive karaoke sessions. All playlist data is stored in Firestore, ensuring that changes are synchronized across devices and available offline once tracks are cached.

2.2.7 User Profile

Each user has a dedicated profile page where they can view and edit personal information such as display name, avatar (handled via Glide for efficient image loading), and short bio. The profile also includes a history of all shared performances, sortable by date or popularity, and displays follower/following counts. Users may manage notification preferences—choosing whether to receive push alerts for new followers, likes, or comments—directly from the settings panel. All profile changes and preferences update in real time via Firestore and propagate immediately across the app.

2.2.8 Notifications & Interaction

To keep users engaged, the app provides both in-app and push notifications. Firebase Cloud Messaging delivers real-time alerts for social interactions—new followers, comments, likes—while in-app banners and badge counts highlight pending notifications. These mechanisms ensure that users remain connected to the community and are promptly informed of any activity related to their performances or social network. Future enhancements may include one-on-one direct messaging, enabling private conversations between users.

2.3 Non-Functional Requirements

2.3.1 Performance

The application must deliver a consistently smooth and responsive user experience under a variety of network conditions. All UI actions—such as tapping play/pause, navigating between screens, or saving a performance—should complete within two seconds. Streaming and playback controls powered by ExoPlayer should begin within one second of user request, and buffer seamlessly without stutter or interruption. Calls to cloud services—whether retrieving a catalog of tracks, fetching a friend’s feed, or uploading media—must complete within ten seconds or provide progressive feedback (e.g., a spinner or progress bar) to keep users informed. To achieve these targets, the app employs local caching for recently accessed media and Firestore queries optimized with appropriate indexing. Asynchronous coroutines ensure that long-running tasks (e.g., network fetches or media encoding) do not block the main thread.

2.3.2 Security

Protecting user data and privacy is paramount. All communications between the client and backend services occur over HTTPS with TLS 1.2 or higher. Authentication tokens issued by Firebase Authentication are securely stored in Android’s encrypted SharedPreferences and automatically refreshed as needed. Firestore Security Rules enforce that users can only read or write their own profile data, playlists, and performance records; shared performances are governed by visibility flags (public, friends-only, private) checked at query time. Cloudinary upload presets restrict transformations to approved formats and file sizes, and presigned URLs prevent unauthorized access to raw media. User passwords and sensitive profile information are never logged or exposed in analytics.

2.3.3 Usability

The user interface adheres to Material Design guidelines, ensuring clarity and consistency across the app. Primary actions such as recording, playback, and sharing are prominently placed and accompanied by intuitive icons and labels. Feedback is provided instantly on user interactions—buttons animate on tap, and toast messages confirm operations like “Performance saved.” In-app tutorials on first launch guide users through core features

(browse, record, share), with the option to revisit help topics via a dedicated support screen. Color contrast ratios, font sizes, and touch target dimensions meet accessibility standards to accommodate users with varying visual or motor abilities.

2.3.4 Maintainability & Reliability Requirements

The codebase follows the MVVM architecture, separating UI logic (ViewModel) from business logic (Model) and presentation (View). Features are modularized into packages—playback, recording, social, profile—enabling developers to locate and update components with minimal cross-impact. Continuous Integration pipelines run lint checks, unit tests, and UI tests on every commit to catch regressions early. The app targets 99.9% uptime: Firebase’s SLA-backed services power the backend, and Cloudinary’s CDN handles media delivery. Crashlytics monitors runtime errors in production, alerting the team and providing stack traces to expedite fixes. Regular automated backups of Firestore data guard against accidental data loss.

2.3.5 Error Handling & Recovery

Robust error handling ensures the application remains functional even when failures occur. Network timeouts, upload failures, or API errors trigger user-friendly dialogs explaining the issue (“Network unavailable – retry?”) and offer actionable options (retry, cancel). All errors are logged via Crashlytics with contextual information (user ID, device OS, app version), enabling swift diagnosis. Local operations—such as saving a draft recording—use transactional writes to prevent corruption, and uncommitted changes are restored on next launch. In the event of a critical failure, the app falls back to a safe state (e.g., returns to the home screen) rather than crashing outright, ensuring users can continue interacting with core features.