Final Report for OnCore

Yutong Qin (Yoki), Yuqian Cui (Nana), Yingtong Shen (Sophie)

Core Features

Ticket Wallet

Ticket Wallet is one of the app's main features. It lets users add tickets manually or digitize paper tickets using their camera or photo gallery. Each ticket isn't just saved as an image—it's linked to key show details like title, date, venue, seat number, and price. This turns each ticket into a complete viewing record. It solves the problem of losing or damaging physical tickets and lays the groundwork for calendar sync and spending insights.

Personal Calendar

The calendar shows upcoming and past shows in both monthly and weekly views. Saved performances are automatically added to the calendar, making it easy for users to keep track of their plans or look back at what they've seen. Since all data is stored locally, the calendar remains accessible even when offline.

Budget Tracker

The budget feature automatically tracks how much users spend on shows based on ticket prices they enter. Spending is displayed visually through simple charts, giving users a clear overview of their costs. It helps them manage their budget while adding long-term value to the app.

Discovery Features

Daily Pick

Daily Pick brings fresh content recommendations using a third-party music API. It offers users a light and easy way to explore new musicals or plays. This feature shifts the app beyond just tracking experiences, encouraging users to keep discovering new ones.

Event Search

Powered by the Ticketmaster API, Event Search lets users browse shows currently playing or coming soon in New York. Each listing includes show name, date, venue, and official links. Users can save any show they're interested in directly to their calendar or Ticket Wallet for future planning.

Social Features

Community Feed

The community feed showcases records shared by other users, creating a live stream of recent activity. Users can scroll through others' experiences, react, and leave comments—building a space for casual interaction and shared enthusiasm.

Headcounts

For each show, the app displays how many users are planning to attend or have already gone. This adds a light social layer to the experience, helping users feel part of a broader audience community.

Reminders and Notifications

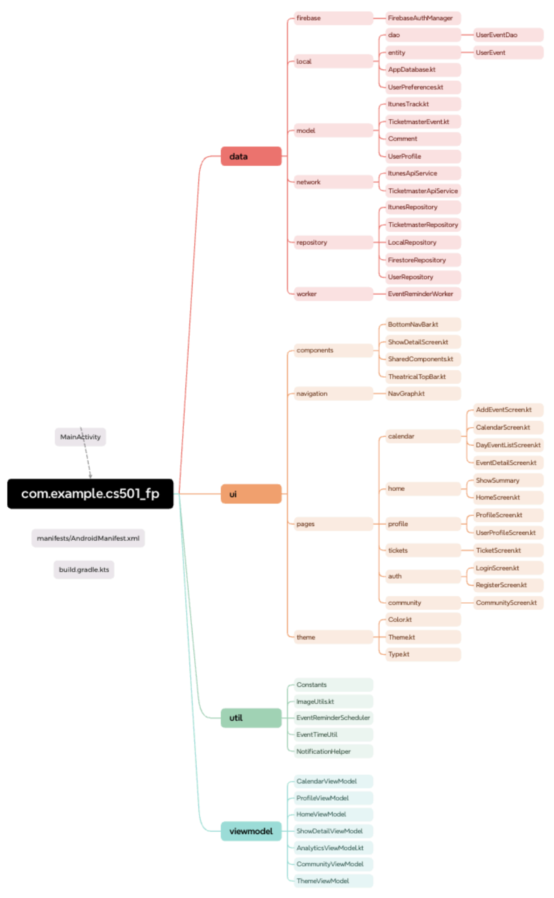
The app offers performance reminders via local notifications. Even when not running in the foreground, users will get timely alerts before a show starts. This system is powered by background scheduling and is designed to match real-life needs like travel time and preparation.

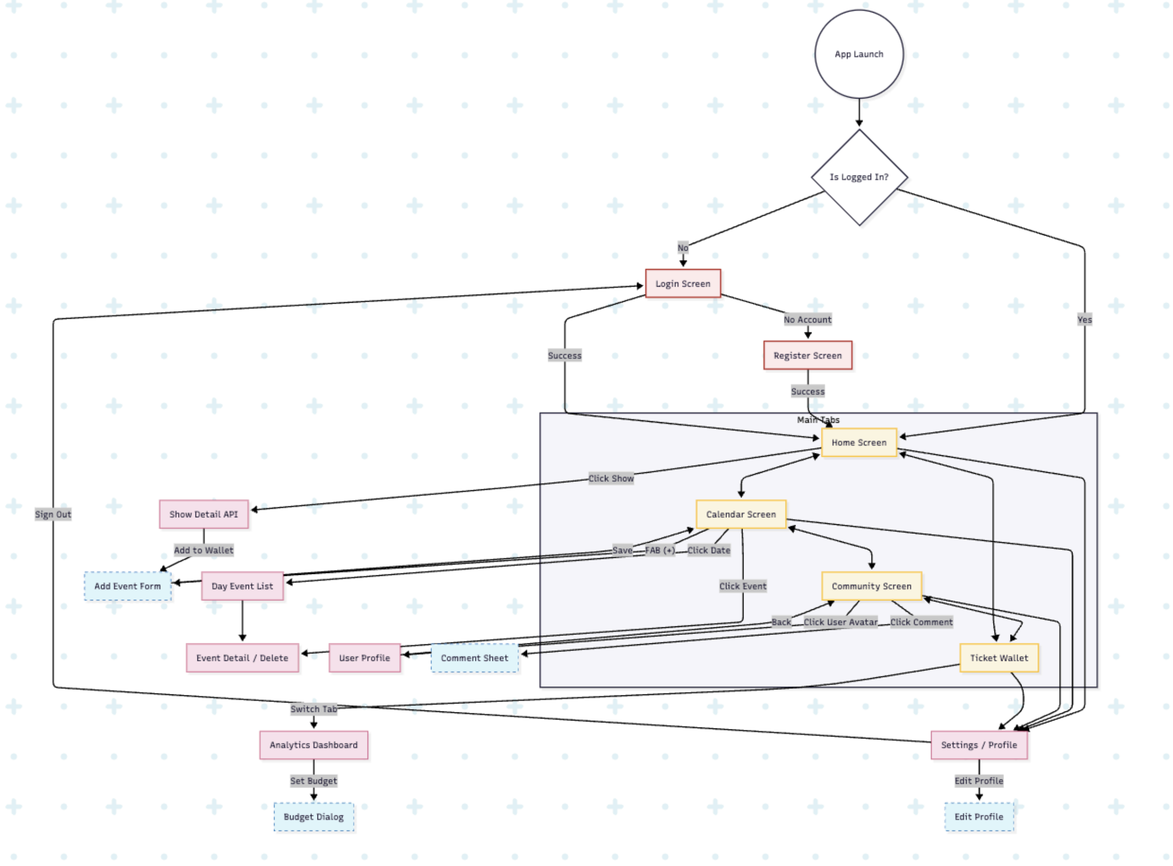
Interface and Interaction Design

The UI follows a clean, modern mobile design approach, focused on simplicity, clarity, and consistency. Key content is shown using card-based layouts, and users can switch between main sections through a bottom navigation bar. Color schemes and typography follow Material Design standards for a readable, user-friendly experience. Subtle animations also help make browsing and interacting feel smoother and more engaging.

Architecture And Technologies Used

Architecture





Technologies Used
Android / Kotlin / Jetpack Compose / Material 3 / Jetpack Navigation (Compose) / MVVM Architecture / ViewModel / AndroidViewModel / Kotlin Coroutines / Kotlin Flow / StateFlow / Room Database / SQLite / Firebase Authentication / Firebase Firestore / Firebase Storage / WorkManager / Android Notifications / Ticketmaster Discovery API / iTunes Search API / Retrofit / Gson / MediaPlayer / Coil (Image Loading) / Internal Storage (File I/O) / java.time API / Canvas (Custom Drawing) / Android Runtime Permissions / Dark / Light Theme (Material Theme)

Reflection on Teamwork and Agile Practices

Our team consisted of three members, and we initially divided responsibilities based on individual strengths. One team member focused on UI and front-end development using Jetpack Compose, another was responsible for database design and data persistence, and the third handled API integration and data fetching. This role-based division helped us make quick progress early in the project and ensured that each core technical area had clear ownership.

At the start of development, we followed a plan to complete and polish the UI first before fully integrating backend functionality. We also organized our GitHub workflow using separate branches based on roles, which allowed us to work in parallel during the early stages. However, as we began connecting the database and APIs, we encountered limitations with this approach. Many UI components relied on placeholder data, and once real data flows were introduced, we realized that the UI did not always align with actual data structures, loading states, and edge cases. This resulted in discrepancies that required UI revisions.

In response to these challenges, we adapted our development plan, which reflects an Agile mindset. Instead of finalizing the UI upfront, we shifted our focus to stabilizing backend logic, data persistence, and API integration first. Once the data flow and state management were reliable, we returned to refining the UI so that it accurately reflected real user interactions and data behavior. This adjustment significantly reduced rework and improved the overall consistency of the app.

At the same time, we also changed our branching strategy. As the project became more integrated, role-based branches became less effective. We transitioned to feature-based (functional) branches, where each branch represented a specific user-facing feature or workflow in the Musical Calendar app. This change improved collaboration, reduced merge conflicts, and encouraged shared ownership across UI, backend, and data layers.

Throughout the project, we followed Agile-inspired practices, including iterative development aligned with course milestones, frequent team communication, and continuous testing and refinement. Regular check-ins helped us discuss progress, resolve technical disagreements, and adjust priorities when needed. Rather than strictly following an initial plan, we embraced flexibility and made decisions based on real development constraints.

Overall, this project strengthened our teamwork and collaboration skills while giving us practical experience applying Agile principles in a real-world mobile development context. By adapting our workflow, communicating openly, and iterating based on feedback, we were able to deliver a more polished and realistic Musical Calendar application.

Testing Approach and Results

Our testing process for the Musical Calendar app followed a systematic debugging loop, which emphasized reproducing issues, localizing the root cause, and applying targeted fixes. This structured approach helped us debug efficiently and avoid repeated trial-and-error.

When an issue occurred, we first focused on reproducing the problem consistently. For example, one issue we encountered was that public repositories could not be displayed on the community page. By testing the app across different user states and data conditions, we confirmed that the problem was reproducible and not caused by random UI behavior.

Next, we worked on localizing the source of the issue. We inspected data-fetching logic, Firestore queries, and ViewModel state updates to identify where the data flow was breaking. Through logging and step-by-step verification, we discovered that the issue was caused by a naming conflict between Kotlin variables and Firestore field names, which prevented the correct mapping of public data.

After identifying the root cause, we applied a focused fix by renaming conflicting fields and updating the related data models and queries. We then retested the feature to ensure that public repositories were correctly fetched and displayed, and that the fix did not introduce new issues elsewhere in the app.

This reproduce–localize–fix loop was used repeatedly throughout development for other issues, including UI inconsistencies, state update timing, and backend integration problems. In addition to debugging, we performed manual functional testing, UI responsiveness testing, and informal peer testing to validate overall app stability and usability.

By following a systematic testing and debugging approach, we were able to resolve issues efficiently and improve the reliability of the Musical Calendar app. By the final version, core features functioned as expected, error handling was more robust, and the overall user experience was significantly improved.

AI Reflection Report

Throughout the development of OnCore, our team utilized AI tools - specifically the Android Studio’s Gemini Assistant - acting effectively for the process. Our primary goal was to resolve syntax-heavy challenges to enable high-level architectural decisions, and efficiently solve error messages from Logcats that are not highly readable. We integrated these tools across three main domains: UI generation (such as color selection, the one that looks most suitable in the design aspect), the complex date-time manipulations, and error analysis to decipher obscure Logcat stack traces. By treating AI as a consultative resource rather than a solution provider, we maintained full ownership of the codebase while significantly reducing development time.

AI is highly beneficial in handling mathematical logic and visual styling, which is the areas that are often time-consuming to implement from scratch. For instance, in the calendar page, we tasked the AI to discuss ideas on how to converting various time string formats into minutes to implement out “time conflict detection” logic, and this saved us hours of calculating the mathematical numbers to implement it correctly. Additionally, for the home screen, the AI assisted in writing complex Jetpack Compose Canvas code to create that gold and red gradient spotlight effect, which also requires intricate brush coordinate calculations, and helping us establish a cohesive visual identity that matched our design mockups.

However, despite its utility, the AI frequently provided misleading suggestions, such as its tendency to suggest Material 2 components, which is not compatible with our project’s material 3 dependency, and thus on those aspects we intend to design and handles on our own to save more time. Our team adopted a strict "Trust but Verify" protocol for all AI-generated code, focusing on security compliance and architectural integrity. A critical example occurred during the implementation of the Camera feature. The AI suggested using Environment.getExternalStorageDirectory() to save ticket images, a method that is deprecated and violates Android 10+ Scoped Storage security policies. We caught this during code review and refactored the logic to use context.openFileOutput and FileProvider, ensuring images were saved securely within the app’s internal sandbox.

In conclusion, AI tools served as a powerful productivity multiplier for the our team, particularly for syntax lookups and logic isolation. However, our experience highlighted that AI lacks the context of modern Android security standards and clean architectur. The final quality of the application depended entirely on our team’s ability to review, refute, and refactor the AI's suggestions into safe, idiomatic Kotlin code.